PEAKTOP Instruction Set Architecture Manual

PEAKTOP ISA v1.3.10.4

Rev. 190814

IHP - Innovations for High Performance Microelectronics

Preface

This document is a detailed specification of the PEAKTOP Instruction Set Architecture (ISA).

Audience

The document is to be used by architects, system designers, hardware developers, compiler and operating system developers, software writers (especially in assembly) for systems based on the PEAKTOP ISA.

Version

The version of the PEAKTOP ISA is given by four numbers. The first number specifies the version regarding the general PEAKTOP philosophy and foundations such as basic architectural properties, view of registers, memory, exception and interrupt handling mechanism, etc. The second, third and fourth number specify the versions of the data transfer, arithmetic/logic and control instructions, respectively. The initial version was v1.0.0.0, and up to v1.3.10.4, the PEAKTOP ISA was in the development phase. The first public document describing the PEAKTOP ISA is for version v1.3.10.4.

History

The PEAKTOP ISA has it origins at the Faculty of Electrical Engineering and Information Technologies (FEEIT) – Skopje, Macedonia (www.feit.ukim.edu.mk), in the magister work of Aleksandar Simevski under the mentorship of Prof. Dr. Aristotel Tentov which started in 2007. In January 2010, the defense of the magister thesis took place in Skopje, with co-mentoring from Prof. Dr. Rolf Kraemer, also a member of the thesis committee (together with Prof. Tentov and the Dean of FEEIT at that time Prof. Dr. Mile Stankovski). Prof. Kraemer is a professor at the Brandenburgische Technische Universität (BTU) Cottbus-Senftenberg, Germany (www.b-tu.de), and a head of the System Design Department in the state research institute IHP Microelectronics (www.ihp-microelectronics.com). The cooperation between IHP and FEEIT was initiated in a project funded by Deutscher Akademischer Austauschdienst (DAAD). At that time, the PEAKTOP ISA was designed, but an implementation was lacking. Aleksandar Simevski after the defense of the magister thesis moved to Germany for obtaining the PhD degree at BTU under the mentorship of Prof. Kraemer, with a scholarship provided by the German state of Brandenburg. An 8-core multiprocessor based on the PEAKTOP ISA v1.2.5.2 was then implemented, produced and tested successfully in IHP 130 nm technology. This chip was used as a demonstrator for a dynamically-adaptable multiprocessor framework, named Waterbear which was developed in the PhD thesis.

Name

If one assumes that the word PEAKTOP is written in Cyrillic, he will read it as "REACTOR", which is actually the original name of the architecture, inspired from the nuclear reactor. However, virtually all of the people whose native language is written in Cyrillic assume that the word is written in English, and they like the name because it represents the "top of the peak", or "the highest peak of all". Thus, the name remained, and it can be read in both ways. Therefore, in this document it is always written with upper case letters because of the "double" meaning.

Style convention

`CODE font` is used to display code in assembly or high-level program languages, as well as names of variables, bit-fields, constants, binary values, etc.

`ADD reg0, reg1` is an example of a valid assembly line. However, listings of program parts or code blocks in assembly or higher-level languages (incl. pseudo-languages) are given in special figure-like environment.

`<replace here>` denotes a placeholder which should be replaced with one of at least two optional terms.

Warning blocks are used to stress specific or exceptional situations.

Information blocks are used used to display important information or additional explanation.

Supplementary materials

PEAKTOP-related literature:

- PEAKTOP Assembler (PAS v4.3.2.0)
- PEAKTOP Multiprocessor Debugger (PMD v2.3.0.0)
- PEAKTOP DW Execution Pipeline

Thank you!

Thank you for using the PEAKTOP ISA and the products based on it. You are highly encouraged to send us feedback, suggestions, error reports, etc. to:

`simevski@ihp-microelectronics.com`

IHP - Innovations for High Performance Microelectronics

Contents

Preface 3

Contents 5

1 INTRODUCTION 11

2 ARCHITECTURAL PROPERTIES 12

2.1 Machine modes . . . 12

2.1.1 Natural machine mode . . . 12

2.1.2 FP machine mode . . . 12

2.1.3 Regularity . . . 13

2.2 Register files . . . 13

2.2.1 Enumeration, labeling and representation . . . 13

2.2.2 Registers operating in lower machine modes . . . 14

2.2.3 Circularity . . . 14

2.2.4 GPR file . . . 15

2.2.5 Special register file . . . 16

2.2.6 DSP, FPR and implementation-specific register files . . . 16

2.3 Memory addressing . . . 16

2.3.1 Address space . . . 16

2.3.2 Data addressing modes . . . 17

2.3.3 Instruction addressing . . . 17

2.3.4 Address alignment . . . 17

2.3.5 Endianness . . . 18

2.3.6 Orthogonality . . . 18

2.4 Program flow . . . 18

2.4.1 `INSTRUCTION COUNTER` . . . 18

2.4.2 Data and control inter-dependencies . . . 18

2.4.3 Pausing execution . . . 19

2.5 Operating system support . . . 19

2.6 Multiprocessing support . . . 19

3 BINARY LAYOUT 21

3.1 Data transfer instructions . . . 21
3.1.1 Memory transfer . . . 22
3.1.2 Inter-register transfer . . . 23
3.1.3 Load immediate . . . 24
3.2 Arithmetic/logic instructions . . . 24
3.2.1 Integer unit . . . 25
3.2.2 Floating point unit . . . 26
3.2.3 DSP unit . . . 26
3.3 Control instructions . . . 27
3.3.1 Program transfer . . . 27
3.3.2 Return from routine . . . 28
3.3.3 Pause instruction execution . . . 29
3.4 Summary . . . 29

4 EXCEPTIONS AND INTERRUPTS 33

4.1 Non-Maskable Interrupt (NMI) . . . 33
4.2 Exceptions . . . 34
4.2.1 DEBUG MODE EXCEPTION . . . 36
4.2.2 INVALID INSTRUCTION . . . 37
4.2.3 SYSTEM INSTRUCTION . . . 37
4.2.4 UNIMPLEMENTED GPR BANK . . . 37
4.2.5 UNIMPLEMENTED INSTRUCTION . . . 37
4.2.6 UNIMPLEMENTED REGISTER . . . 38
4.2.7 INVALID OPERATION . . . 38
4.2.8 DIVISION BY ZERO . . . 38
4.2.9 OVERFLOW . . . 39
4.2.10 FP INVALID OPERATION . . . 39
4.2.11 FP DENORMALIZED OPERAND . . . 40
4.2.12 FP DIVISION BY ZERO . . . 40
4.2.13 FP OVERFLOW . . . 40
4.2.14 FP UNDERFLOW . . . 41
4.2.15 FP INEXACT RESULT . . . 41
4.2.16 DSP EXCEPTION 0 . . . 41
4.2.17 DSP EXCEPTION 1 . . . 41
4.2.18 DSP EXCEPTION 2 . . . 42
4.2.19 DSP EXCEPTION 3 . . . 42
4.2.20 I SYSTEM BUS ERROR . . . 42
4.2.21 D SYSTEM BUS ERROR . . . 42

4.3 Interrupts . . . 43
4.4 Handling mechanism . . . 44
4.4.1 Hierarchy and priority . . . 44
4.4.2 Postponed handling . . . 45
4.4.3 Nesting . . . 46

5 SPECIAL REGISTERS 48

5.1 IMPLEMENTATION REGISTER . . . 49
5.2 EXECUTION STATUS . . . 50
5.3 EXCEPTION INSTRUCTION . . . 51
5.4 EXCEPTION REGISTER . . . 51
5.5 EXCEPTION MASKS . . . 52
5.6 EXCEPTION TABLE BASE ADDRESS . . . 52
5.7 INTERRUPT TABLE BASE ADDRESS . . . 52
5.8 CORE ID . . . 53
5.9 PROCESS ID . . . 53
5.10 SYSTEM CONTROL REGISTER . . . 53
5.11 NMI RETURN POINTER . . . 54
5.12 EXCEPTION RETURN POINTER . . . 54
5.13 USER CONTROL REGISTER . . . 54
5.14 CALL RETURN POINTER . . . 55
5.15 INTERRUPT RETURN POINTER . . . 56
5.16 DSP CONFIGURATION REGISTER . . . 56

6 INSTRUCTION SET 57

6.1 Detailed instruction specification . . . 57
6.1.1 MOV – Move data . . . 58
6.1.2 ADD – Add . . . 64
6.1.3 SUB – Subtract . . . 67
6.1.4 MUL – Multiply . . . 70
6.1.5 DIV – Divide . . . 73
6.1.6 SL – Shift left . . . 77
6.1.7 SR – Shift right . . . 80
6.1.8 RL – Rotate left . . . 83
6.1.9 RR – Rotate right . . . 86
6.1.10 AND – AND bitwise . . . 89
6.1.11 NAND – Negated AND bitwise . . . 92
6.1.12 OR – OR bitwise . . . 95
6.1.13 XOR – Exclusive OR bitwise . . . 98

6.1.14 SB – Set bit . . . 101
6.1.15 RB – Reset bit . . . 104
6.1.16 TB – Test bit . . . 107
6.1.17 RVB – Reverse bits . . . 110
6.1.18 FADD – FP Add . . . 113
6.1.19 FSUB – FP Subtract . . . 115
6.1.20 FMUL – FP Multiply . . . 117
6.1.21 FDIV – FP Divide . . . 119
6.1.22 FREM – FP Remainder . . . 121
6.1.23 FCMP – FP Compare . . . 123
6.1.24 FSQR – FP Square root . . . 125
6.1.25 FABS – FP Absolute . . . 127
6.1.26 FNEG – FP Negate . . . 129
6.1.27 FRND – FP Round to integer . . . 131
6.1.28 FF2I – FP to integer . . . 133
6.1.29 FI2F – Integer to FP . . . 135
6.1.30 FEXT – Extend FP format . . . 137
6.1.31 FSQZ – Squeeze FP format . . . 139
6.1.32 MAD – Multiply-add . . . 141
6.1.33 MSU – Multiply-subtract . . . 144
6.1.34 FMAD – FP Multiply-add . . . 147
6.1.35 FMSU – FP Multiply-subtract . . . 149
6.1.36 JMP – Jump . . . 151
6.1.37 BZ – Branch if Zero . . . 154
6.1.38 BNZ – Branch if Not Zero . . . 157
6.1.39 BM – Branch if MSB . . . 160
6.1.40 BMZ – Branch if MSB or Zero . . . 163
6.1.41 BNM – Branch if Not MSB . . . 166
6.1.42 BNMO – Branch if Not MSB or all Ones . . . 169
6.1.43 BL – Branch if LSB . . . 172
6.1.44 BLZ – Branch if LSB or Zero . . . 175
6.1.45 BNL – Branch if Not LSB . . . 178
6.1.46 BNLO – Branch if Not LSB or all Ones . . . 181
6.1.47 BO – Branch if all Ones . . . 184
6.1.48 BNO – Branch if Not all Ones . . . 187
6.1.49 RET – Return from procedure . . . 190
6.1.50 RETI – Return from interrupt handler . . . 192
6.1.51 RETE – Return from exception handler . . . 194

6.1.52 `RETN` – Return from Non-Maskable Interrupt (NMI) handler 196
6.1.53 `WAIT` – Wait 198
6.2 System instructions 200
6.3 Assembly conventions 200
6.3.1 Instruction options 200
6.3.2 Instruction arguments 202
6.3.3 Summary 203
6.4 Pseudo-instructions 207
6.4.1 Single pseudo-instructions 207
6.4.2 Multiple pseudo-instructions 208
6.5 Examples 211

List of Acronyms **216**

List of Figures **217**

List of Tables **220**

List of Examples **222**

List of Listings **225**

References **226**

Index **227**

1. INTRODUCTION

The PEAKTOP ISA is a general-purpose Load/Store Reduced Instruction Set Computer (RISC) architecture, suitable for real-time embedded processing as well as for general data processing. It is designed primarily for 32-bit and 64-bit implementations. However, this is not a limitation for shorter or wider implementations. It provides operating system and multiprocessing support.

The following points briefly describe the PEAKTOP ISA.

Simple The design of the ISA is driven by simplicity. It can be seen by the fact that in total only 53 mnemonics are used for all native (non-pseudo) instructions. Simplicity is further promoted by the principles of regularity, circularity and orthogonality.

Flexible Another motivation behind the PEAKTOP ISA besides simplicity is flexibility in tailoring the implementations according to their purpose and functionality. That is, they can choose which of the arithmetic/logic instructions to implement provided that an exception is raised on each unimplemented instruction, thus enabling simulation of the instruction by software. The Floating Point Unit (FPU) and the Digital Signal Processing (DSP) unit are also optional. Up to eight register files (2 mandatory and 6 optional) may be used.

Complete Although simple, the PEAKTOP ISA is complete and provides all the functionality as any other RISC architecture. It has a full operating system and multiprocessing support.

Regular Machine modes of $2^0, 2^1, 2^2, \ldots, 2^7$ bytes are possible. Each implementation defines its natural machine mode. However, the PEAKTOP ISA demands that all modes up to and including the defined natural machine mode must be implemented. For example, if the natural machine mode is 32-bit (4 bytes), the machine modes of 8- and 16-bits must also be fully implemented.

Circular The last register in the register file sees the first register as its subsequent neighbor, in the same manner that the first register sees the second. This applies for all register files.

Orthogonal All addressing modes consider equal all General-Purpose Register (GPR)s. Not only that, all instructions also consider equal all GPRs regarding width and functionality. That is, there are no "special" GPRs (which is even contradictory by definition).

The PEAKTOP ISA is **scientific**: regularity, orthogonality and circularity make the architecture a natural platform for scientific problems.

These "scientific" properties of the ISA further enable regularity, completeness, simplicity and flexibility during compiler and program construction as well as during their (formal) verification.

The properties of the PEAKTOP ISA are described in Section 2. The binary representation, instruction layout and description is given in Section 3. The exceptions, interrupts, their priorities and handling are defined and described in Section 4. The special registers are thoroughly described in Section 5. Finally, Section 6 gives the details of each instruction in the architecture, as well as the assembly conventions.

2. ARCHITECTURAL PROPERTIES

All instructions in the PEAKTOP ISA are 32-bit wide with 0,1,2 or 3 operands. In the arithmetic/logic operations one register is used both as an operand source and as a destination for the result of the operation. This type of machines are usually called **two-address machines**.

The binary representation of signed integers is in **second complement**. Common implementations usually choose 32-bit or 64-bit GPR width as well as 32-bit or 64-bit Arithmetic/Logic Unit (ALU) width, although both wider and shorter widths are also possible. The ISA defines 8, 16, 32, . . . , 1024-bit wide machine modes.

2.1 Machine modes

Each instruction has a 3-bit field `MMODE` specifying the **machine mode** in which the instruction is executed (see Table 1).

Table 1: *Machine modes*

MMODE	Nr. bits	Nr. bytes	Option	Description
000	8	1	B	Byte
001	16	2	H	Halfword
010	32	4	W	Word
011	64	8	D	Doubleword
100	128	16	Q	Quadword
101	256	32	1	Sentence
110	512	64	2	Doublesentence
111	1024	128	4	Quadsentence

For example, a data transfer instruction in word mode transfers 32 bits (4 bytes) of data; an arithmetic/logic instruction in halfword mode performs an operation on 16-bit wide (2 byte) operands; a branch instruction in byte mode examines an 8-bit (1 byte) argument, etc.

2.1.1 Natural machine mode

Implementations define the **natural machine mode**. It is determined by the GPR width and the ALU width. If both of them are, e.g., 32-bit, then the natural mode is word (`W`). However, if the widths differ, e.g., the ALU width is 32-bit and the GPR width is 64-bit or 16-bit, the natural mode should be defined carefully according to other criteria (for instance, from the target application).

2.1.2 FP machine mode

Implementations with an FPU additionally may have a Floating Point (FP) machine mode designated only for the FP instructions with bit widths other than the "power-of-two" widths in Table 1 (e.g., 80 bits for extended FP precision). Therefore, in the text, the machine modes of Table 1 are also referred to as **integer machine modes**. See also Subsection 3.2.2.

2.1.3 Regularity

The property of **regularity** of the ISA imposes that all machine modes up to (and including) the defined natural machine mode must be implemented. For example, an implementation with a 32-bit natural machine mode must also implement 8- and 16-bit machine modes.

2.2 Register files

A set of registers grouped by their function is called **register file**. The PEAKTOP ISA predefines two register files:

- GPR file
- Special register file

Furthermore, implementations may define up to six additional register files, e.g., DSP file or Floating Point Register (FPR) file, or other implementation-specific register files, e.g., additional GPR file. Thus, with the predefined two register files, in total up to eight register files are possible. All registers within a register file must have the same width, which is a consequence of the regularity property.

The **register state** is the state of all registers in all register files at a given point of time.

2.2.1 Enumeration, labeling and representation

The instructions of the PEAKTOP ISA have 6-bit fields that specify the register(s) to be used. This means that the maximal number of registers in each of the register files is 64. However, implementations can have lower number of registers. All registers within a register file are enumerated from `0` to max. 63. Thus, the 6-bit instruction fields directly specify the number of the register to be used.

The register names (labels) `reg<nr>` and `spc<nr>` denote the `<nr>`-th register of the GPR file and the special register file, respectively. For example, `reg3` denotes GPR 3. Furthermore, the labels `REG` and `SPC` (without numbers) refer to the entire GPR and special register file, respectively. If the implementation opts to use DSP or FPR files, the register names and labels are `dsp<nr>` and `fpr<nr>` for the registers, and `DSP` and `FPR` for the entire files, respectively. For other implementation-specific register files, the register names and labels are left unspecified.

In the register representations, the bit significance of the register increases with sliding from the left to right. Thus, the Most Significant Bit (MSB) of the register is on the far left side, while the Least Significant Bit (LSB) is on the far right side. Fig. 1 shows an 8-bit register with indicated MSB (`7`) and LSB (`0`). Bit enumeration starts from zero.

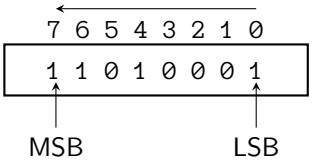
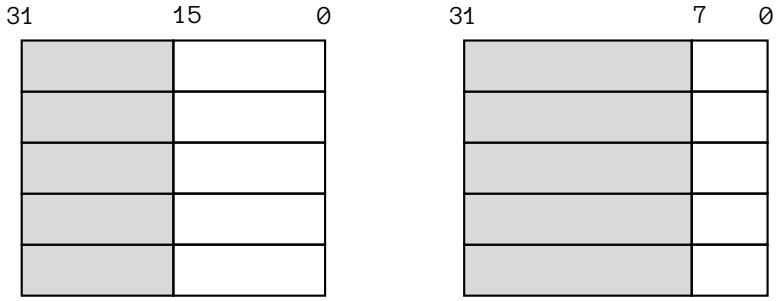


Fig. 1: *Graphical representations of registers, instructions and bit-fields*

This is also true for the representation of instructions and instruction bit-fields.

2.2.2 Registers operating in lower machine modes

In machine modes that are shorter than the GPR width, only the corresponding LSBs of the registers are used. For example, Fig. 2 shows a 32-bit register file operating in 16-bit and 8-bit mode, in which only the lower 16 and 8 bits (white fields) are used, and the upper 16 and 24 bits (grey fields) are not used, respectively.



(a) 32-bit register file in 16-bit machine mode (b) 32-bit register file in 8-bit machine mode

Fig. 2: *Registers operating in lower machine modes*

On the other side, if the machine mode is greater than the GPR width, then the principle of circularity applies (see Subsection 2.2.3).

2.2.3 Circularity

In machine modes that are wider than the register file width, more than one registers are used. For example, when writing a 32-bit register file in 64-bit machine mode, the higher 32-bit part will not be truncated but written to the subsequent register. That is, two 32-bit registers will be written: the lower 32-bits will be written in the register specified by the instruction (e.g., register 2) and the higher 32-bits will be written in register 3. On the other hand, when reading register 2 in 64-bit machine mode, the 64-bit data is formed by concatenating register 3 (higher 32 bits) and register 2 (lower 32 bits). Similarly, in 128-bit machine mode, four registers are used, etc.

Fig. 3 shows an example with an 8-bit register file in which register 2 is written/read in 16-bit machine mode. The ordering is **little-endian**.

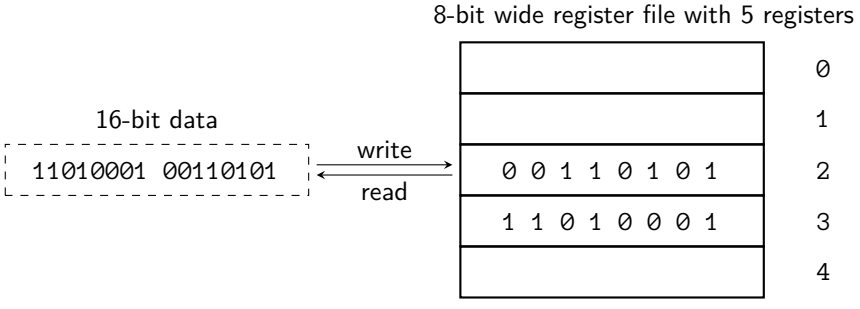


Fig. 3: *Read/write of data wider than the register file width*

However, the number of registers in the file is limited, as said, there can be maximum 64 registers. If the last register is written/read in a wider machine mode, then the higher data part comes from or goes to the first register, as if they are subsequent neighbors. This is the property of register **circularity** of the ISA. Fig. 4 shows an 8-bit register file with five registers, in which the last register (4) is written/read in 16-bit machine mode.

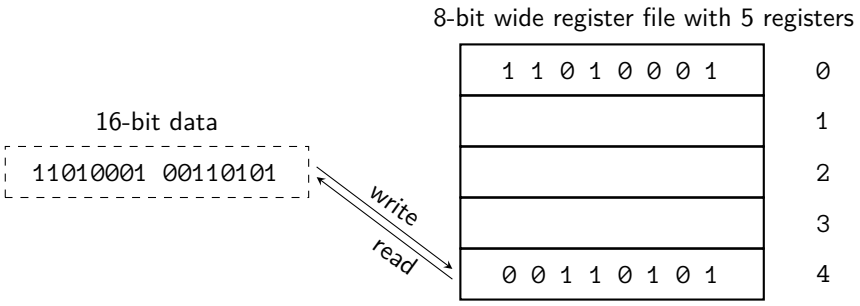


Fig. 4: *Read/write of wider data in the last register*

In this case, the higher 32 bits are written in register 0. Thus, the register file of Figs. 3 and 4 can be represented as in Fig. 5, showing more clearly which data parts go where. In Fig. 5 the bit significance of the registers' content and the register number increase with moving clock-counterwise.

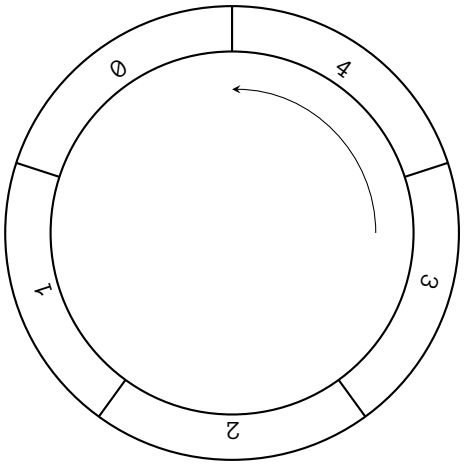


Fig. 5: *Register circularity*

If more than two registers are involved in a read/write operation, the same principle applies in the same manner.

Circularity of a GPR file with a non-standard width in integer machine mode

FPUs with extended precision may impose implementations using a "non-standard", i.e., not a power-of-two width of the GPR file. In such a case, all register bits are used in the FP machine mode. However, for integer machine modes that are shorter than the GPR width, only the corresponding subset of bits at the least significant end of the GPRs are "visible" (see Subsection 2.2.2). On the other hand, if the integer machine mode is greater than the GPR width, the property of circularity is here applied in the same manner for the "visible" part of the GPR file.

2.2.4 GPR file

The GPR file is used directly by most of the instructions. In fact, all other register files can be accessed only through the GPR file by inter-register transfer instructions[1]. As said, the maximal number of GPRs can be 64, but the minimal number is 3, i.e., all implementations must have at least three GPRs.

[1] An exception to this is when implementations opt to have an FPR file which is accessed by the FP instructions (see Subsection 2.2.6).

GPR width

The **minimal GPR width** is 8 bits and the **maximal** is 1024 bits (according to the machine modes in Table 1). However, the GPR width does not have to be a "power-of-two" number. For instance, implementations with FPU may require extended FP precision, e.g., 80-bit, in which case the GPR width can be set to 80 bits. Alternatively, implementations may opt to add another FPR file, independent from the GPR file. In this case, the FP instructions use the FPR file and not the GPR file (see Subsection 2.2.6).

2.2.5 Special register file

Table 2 shows the special registers predefined by the PEAKTOP ISA. Section 5 shows all the details of the special registers.

Table 2: *Special registers*

Nr.	Register
1	IMPLEMENTATION REGISTER
2	EXECUTION STATUS
3	EXCEPTION INSTRUCTION
4	EXCEPTION REGISTER
5	EXCEPTION MASKS
6	EXCEPTION TABLE BASE ADDRESS
7	INTERRUPT TABLE BASE ADDRESS
8	CORE ID
9	PROCESS ID
10	SYSTEM CONTROL REGISTER
11	NMI RETURN POINTER
12	EXCEPTION RETURN POINTER
13	USER CONTROL REGISTER
14	CALL RETURN POINTER
15	INTERRUPT RETURN POINTER
16	DSP CONFIGURATION REGISTER

2.2.6 DSP, FPR and implementation-specific register files

These register files are optional and can be accessed only through the GPR file by inter-register transfer instructions (see Subsection 3.1.2). However, the PEAKTOP ISA specifies that if a FPR file is implemented, then the FP instructions use the FPR file, and not the GPR file. On the other hand, If FPR file is not implemented, the FP instructions use the GPR file.

Furthermore, the PEAKTOP ISA provides a 4-bit auxiliary opcode which specifies the inter-register transfer between the GPR file on one hand, and the DSP and FPR file on the other (see Table 8). Thus, in total 7-out-of-16 codes are predefined, while the other 9 codes are left for (up to four) implementation-specific register files.

2.3 Memory addressing

2.3.1 Address space

The PEAKTOP ISA supports up to 128-bit **virtual address space**.

The lower part of the **virtual address** is the data or instruction address (see Subsections 2.3.2 and 2.3.3). The higher part of the virtual address is contained in the PROCESS ID special register.

The **physical address** can be formed either directly from the virtual address (as a subset of the virtual address), or, by a translation with a Memory Management Unit (MMU) or Memory Protection Unit (MPU).

The Input/Output (IO) devices are **memory-mapped** and the instructions make no difference in accessing memory or IO.

2.3.2 Data addressing modes

Data is accessed in memory with **memory transfer** instructions. The **data address** is formed in one of three ways:

Register The address is contained in a GPR.

Displacement base + offset: the address is formed as a sum of the base address contained in a GPR, and a 12-bit signed offset.

Indexed base + index: the address is formed as a sum of the base address contained in a GPR, and the index contained also in a GPR.

The register and indexed addressing modes include forms with automatic pre- and post-increment/decrement of the register/index GPRs according to the machine mode, i.e., according to the number of accessed bytes.

Furthermore, an immediate access to data and operands is also provided. The *load immediate* instruction has an 18-bit signed/unsigned immediate, while arithmetic/logic instructions have a 14-bit signed/unsigned immediate.

2.3.3 Instruction addressing

The **instruction address** is formed by multiplying the INSTRUCTION COUNTER by four (see Subsection 2.3.4). The INSTRUCTION COUNTER is automatically incremented by one. However, a **program transfer** instruction (unconditional or taken branch) overwrites it with the target instruction address. The program transfers can be of two types, depending on the way the instruction address is formed:

Relative In relative program transfer, a signed offset (20-bit for unconditional transfer and 14-bit for branches) is added to the current value of the INSTRUCTION COUNTER.

Absolute In absolute program transfer, the INSTRUCTION COUNTER is overwritten with the contents of a GPR specified by the program transfer instruction.

Finally, the INSTRUCTION COUNTER is changed upon entering interrupt, exception or NMI handling (see Subsection 2.4).

2.3.4 Address alignment

It is required that the instruction address is always word-aligned (four-byte-aligned), i.e., the two LSBs of the instruction address are always zero. Word-alignment of instructions facilitates implementation since all instructions are 32-bit wide.

The data address, on the other hand, does not have to be aligned in any way.

2.3.5 Endianness

In respect to memory addressing implementations can choose either **big** or **little endianness**. Static or dynamic configurability of the endianness is also allowed.

However, in respect to the internal operation of the register files, it is strictly specified to be little-endian[2] for the purposes of program compatibility, portability and simplicity. See Subsection 2.2.3.

2.3.6 Orthogonality

The property of **orthogonality** of the ISA imposes that all addressing modes treat and consider equal all GPRs in the GPR file in respect to their width, functionality and purpose. That is, there are no GPRs with special widths, functionalities or purpose, but all are equally-wide general-purpose registers. This means that all instructions (not only addressing modes) treat and consider equal all GPRs.

Of course, at a higher level, e.g., during compiler construction, some GPRs may be assigned special roles such as stack or frame pointers, but this view remains only at the higher level.

2.4 Program flow

A **routine** is a set of instructions performing some task. The program routines are called **procedures**, while the interrupt/exception handling routines are called **handlers**.

2.4.1 INSTRUCTION COUNTER

After system reset, fetching instructions from memory starts at address zero. The INSTRUCTION COUNTER is also reset to zero. After each successfully executed instruction, the INSTRUCTION COUNTER is automatically incremented by one. The value of the INSTRUCTION COUNTER can be changed in four more ways:

- by a *program transfer* instruction (see Subsection 2.3.3);
- by a *return from routine* instruction;
- upon entering *interrupt handling* when it is overwritten with the INTERRUPT TABLE BASE ADDRESS;
- upon entering *exception handling* or *NMI handling* when it is overwritten with the EXCEPTION TABLE BASE ADDRESS.

The program transfer instructions have the option to save the current INSTRUCTION COUNTER in the CALL RETURN POINTER. On the other hand, the INSTRUCTION COUNTER is automatically saved in the INTERRUPT RETURN POINTER, EXCEPTION RETURN POINTER or NMI RETURN POINTER on entering interrupt, exception or NMI handling, respectively. Returning from a routine to the point of the routine call, or to the point of interruption/exceptional instruction is done by executing a *return from routine* instruction.

2.4.2 Data and control inter-dependencies

The **data inter-dependencies** between instructions can be deduced from the register specifiers within the instructions.

[2] Therefore, little endianness in the PEAKTOP is slightly more preferred than big also for the memory.

Although there is an EXECUTION STATUS register containing flags from execution of arithmetic/logic instructions, the branch instructions do not use them, which means that implementations do not have to take care of the EXECUTION STATUS flags when dealing with **control inter-dependencies**.

However, software should take care in implementations with out-of-order execution if it decides to use the EXECUTION STATUS register, since this register reflects the status of the last executed arithmetic/logic instruction. That is, in out-of-order execution, a subsequent arithmetic/logic instruction can be executed before the arithmetic/logic instruction upon which a branch makes a decision according to an execution flag.

2.4.3 Pausing execution

Program execution can be paused (definitely or indefinitely) by the WAIT instruction. A definite pause is finished when the predefined pause period expires. Furthermore, both definite and indefinite pauses are finished by an interrupt/NMI, or by reset.

2.5 Operating system support

The PEAKTOP ISA defines two **operating modes**:

system In system mode, all instructions can be executed, and all registers can be accessed.

user In user mode, system instructions cannot be executed and some special registers cannot be accessed.

An attempt in user mode to execute a system instruction or to access some special registers which cannot be accessed in user mode, raises an exception.

The execution starts in **system mode**. Writing the 0-th bit of the SYSTEM CONTROL REGISTER with zero switches to execution in **user mode** (see Subsection 5.10). However, only a raised, potent exception, potent interrupt or NMI switches to execution in system mode. Thus, for example, an attempt to write to the SYSTEM CONTROL REGISTER in user mode will raise an exception, which if potent, will transfer execution to the exception handler in which the operating system can determine the operating mode.

With these simple mechanisms, the PEAKTOP ISA satisfies the Popek and Goldberg's virtualization requirements (classic virtualization – trap-and-emulate) [1].

MMU/MPU

Implementations may include MMU/MPU for address translation and memory/IO access protection which can be largely used by the operating system. The PROCESS ID special register is used to specify the process and to form the virtual address (see Subsection 2.3.1).

2.6 Multiprocessing support

The PEAKTOP ISA provides pairs of load/store instructions in which the store returns one or zero in order to tell whether the load/store at the given memory location was performed atomically or not, respectively. This "atomic" load is called *load-locked*, while the store is called *store-conditional*. Load-locked and store-conditional can be executed in any machine mode. However, the machine modes in a load-locked/store-conditional pair must be the same. On the other side, the addressing modes may differ in a single pair of load-locked and

store-conditional. Nevertheless, the computed (effective) data address still must be the same for the pair, otherwise the store-conditional will always fail. Multiprocessor **synchronization** routines and libraries may be built out of these pairs of instructions [2].

Additionally, the USER CONTROL REGISTER has a writable bit SYNC which may be used by the hardware and software to build synchronization mechanisms (see Subsection 5.13). The state of this bit is reflected on the output **sync line**. Furthermore, in a multiprocessor environment, often it is required that the processing element is identified by a single unique ID within the system. In this direction, the CORE ID special register can be used by the hardware or software.

3. BINARY LAYOUT

This Section details the binary layout of the instruction set. All instructions in the PEAKTOP ISA are 32-bit wide. In the representations of the bit fields, the bit significance increases with sliding to the left. Thus, the MSB of the field is on the first position on the left side, while the LSB is on the first position on the right side (see Fig. 1). The eight MSBs of the 32-bit wide instruction is the instruction OPCODE. There is also an additional, 4-bit auxiliary opcode AUXCODE.

The instructions are divided according to their function in three groups:

- Data transfer instructions
- Arithmetic/logic instructions
- Control instructions

Note that further division and more detailed instruction specification is given in Section 6!

3.1 Data transfer instructions

Fig. 6 shows the layout of the data transfer instruction and Table 3 shows the description of their bit fields.

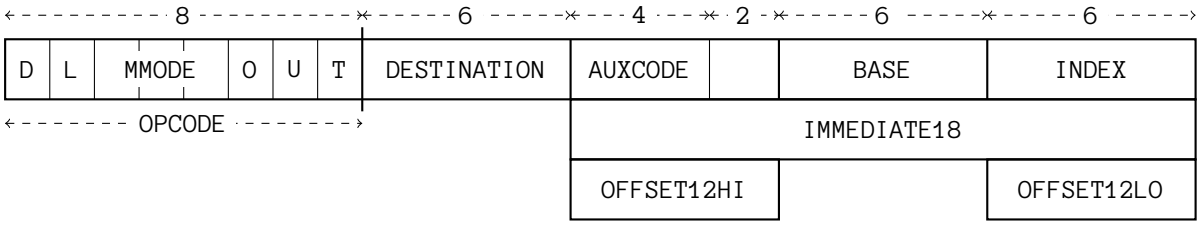


Fig. 6: *Layout of data transfer instructions*

OFFSET12 is a **12-bit signed offset** used for data address formation. It is a concatenation of the 6-bit wide fields OFFSET12HI and OFFSET12LO.

Three types of data transfer are possible:

- Memory transfer
- Inter-register transfer
- Load immediate

The width of the data which is transferred is specified by the MMODE field. The data width is a power-of-two and is min. $8 \times 2^0 = 8$ bits for MMODE = 000, and max. $8 \times 2^7 = 1024$ bits for MMODE = 111 (see Table 1).

Table 3: *Bit field description of data transfer instructions*

Bit field	Width	Description
D	1	Data transfer (always 1 for data transfer instructions)
L	1	Load
MMODE	3	Machine mode
O	1	Offset
U	1	Unsigned immediate data / Atomic memory transfer
T	1	Type of transfer
DESTINATION	6	Destination register specifier
AUXCODE	4	Auxiliary opcode
BASE	6	Base register specifier
INDEX	6	Index register specifier
IMMEDIATE18	18	Immediate signed/unsigned data
OFFSET12HI	6	High part of the 12-bit signed address offset OFFSET12
OFFSET12LO	6	Low part of 12-bit signed address offset OFFSET12

Maximal transfer width

Implementations define a **maximal transfer width** parameter. In order to preserve the regularity property (see Subsection 2.1.3), all transfer widths up to the defined maximal transfer width must be implemented. For example, an implementation with 64-bit maximal transfer width must implement also 8-, 16- and 32-bit transfer widths. An attempt to execute an instruction specifying a width greater than the maximal transfer width (e.g., 128-bit in the example) raises the UNIMPLEMENTED INSTRUCTION exception.

3.1.1 Memory transfer

If the transfer type bit T = 0, a memory transfer is inferred:

load
L = 1
: A memory value is loaded into a REG register (GPR) specified by the DESTINATION field.

store
L = 0
: The value of the REG register specified by the DESTINATION field is stored to memory.

Addressing

The address is formed in three different ways:

displacement
O = 1
: MEM[REG[BASE] + OFFSET12]: the value of the signed 12-bit offset contained in the OFFSET12HI and OFFSET12LO fields is added to the value of the GPR specified by the BASE field to form the address.

register
O = 0
: MEM[REG[INDEX]]: the address is contained in the GPR specified by the INDEX field. For register addressing, the MSB of AUXCODE should be 0.

indexed
O = 0
: MEM[REG[BASE] + REG[INDEX]]: the address is a sum of the values of the GPRs specified by the BASE and INDEX fields. For indexed addressing, the MSB of AUXCODE should be 1.

The expressions REG[<nr>], SPC[<nr>] and DSP[<nr>] refer to the value of the <nr>-th REG, SPC and DSP register, respectively. Furthermore, the expression REG[<nr>][<bitnr>] denotes the <bitnr>-th bit of the <nr>-th GPR, while REG[<nr>][<bithi>:<bitlo>] denotes a range of bits <bithi> down to <bitlo> of the <nr>-th register.

The expression MEM[<address>] refers to the value in memory at a location <address>. Furthermore, expressions like MEM[REG[INDEX]] or MEM[REG[BASE] + REG[INDEX]] are shortly written as MEM[INDEX] and MEM[BASE + INDEX] since the data address is always formed by using registers from the GPR file.

In register and indexed addressing the value of the index register can be automatically incremented or decremented before or after the memory transfer. The increment/decrement value is determined according to the MMODE field. For example, a pre-increment in a 16-bit transfer will add 2 to the value of the INDEX register before forming the address, while a post-decrement in a 32-bit transfer will subtract 4 from the value of the INDEX register after forming the address. The selection of the pre-/post- increment/decrement is done by the AUXCODE field (see Table 8).

The expressions INDEX++ and INDEX−− denote post-increment and post-decrement of the index GPR, respectively. Similarly, ++INDEX and −−INDEX denote pre-increment and pre-decrement.

Atomic memory transfer

Synchronization primitives for memory transfer in multiprocessing applications are specified by setting the bit U = 1: **load-locked** for L = 1 and **store-conditional** for L = 0. All other fields have the same function as for the "normal" memory transfer for U = 0. From the ISA point of view, the single difference is that store-conditional writes the GPR specified by the DESTINATION field with 1 if the atomicity of the load-store couple is preserved, or 0 otherwise, while in normal store this GPR is not changed. The width of the written 0 or 1 is also specified by MMODE.

3.1.2 Inter-register transfer

Inter-register transfer is inferred by T = 1 and O = 0. Data can be transferred (copied) from any REG register to any other REG register. Furthermore, data can be transferred from any REG register to any register in other register files. Similarly, data from any register in any register file can be transferred to any REG register. However, transfer between other register files (not involving a REG register) is not possible. The AUXCODE field differentiates these possibilities (see Table 8). The destination register number is specified by the DESTINATION field, while the data source register number is specified by the INDEX field. For inter-register transfer L = 1 and U = 0.

In fact, writing and reading a non-REG register requires data transfer to/from a REG register since the instructions for memory transfer use only REG registers. Thus, for example, if a SPC register is to be loaded from memory, the memory value has to be firstly loaded into a REG register, and then copied by an inter-register transfer instruction to the destination SPC register. Similarly, before storing a SPC register to memory, its value has to be firstly loaded to a REG register, after which the REG register is stored to memory.

3.1.3 Load immediate

Loading of an immediate value is inferred by `T = 1`, `O = 1` and `L = 1`. The immediate value specified in the 18-bit `IMMEDIATE18` instruction field is directly loaded into a `REG` register specified by the `DESTINATION` field. By specifying `U = 0` the immediate value is sign-extended to the width of the transfer specified by `MMODE`, according to the MSB of the `IMMEDIATE18` field. By specifying `U = 1` the value is zero-extended. Of course, if the transfer width is 16- or 8-bit, the MSBs of the immediate value are truncated, i.e., the lower 16 or 8 bits are correspondingly taken. See Table 4.

Table 4: *Sign/zero extension and truncation of an immediate value according to `MMODE`*

MMODE	U = 0	U = 1
	IMMEDIATE18 = 011001001000110001	
8	00110001	00110001
16	1001001000110001	1001001000110001
32	00000000000000001100100100011000 1	00000000000000001100100100011000 1
	IMMEDIATE18 = 111001001000110001	
8	00110001	00110001
16	1001001000110001	1001001000110001
32	11111111111111111001001000110001	00000000000000001100100100011000 1

Table 4 shows two examples of `IMMEDIATE18`, one with MSB of `0` and the other with MSB of `1`, while the rest 17 bits are the same in both cases, i.e., `110010010001100 01`. Note that in the first case for `MMODE = 16` the MSB of the truncated value is `1` and will be treated as negative number in signed operations, although the supplied immediate was originally positive (with MSB of 0). The same holds for the latter case for `MMODE = 8`, where the truncated value is positive, and the immediate was originally negative. Of course, in unsigned operations this does not matter.

3.2 Arithmetic/logic instructions

Fig. 7 shows the layout of the arithmetic/logic instructions and Table 5 shows the description of their bit fields.

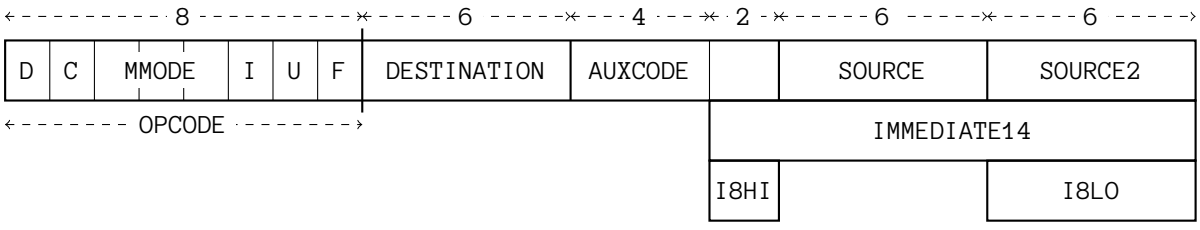


Fig. 7: *Layout of arithmetic/logic instructions*

`IMMEDIATE8` is an **8-bit signed/unsigned immediate** operand. It is a concatenation of the 2-bit wide field `I8HI` and the 6-bit wide field `I8LO`. Bit `U` signals whether `IMMEDIATE8` is signed (`0`) or unsigned (`1`), i.e., the same as for `IMMEDIATE14`.

The arithmetic/logic instructions are used for:

- Integer arithmetic

Table 5: *Bit field description of arithmetic/logic instructions*

Bit field	Width	Description
D	1	Data transfer (always 0 for arithmetic/logic instructions)
C	1	Control (always 0 for arithmetic/logic instructions)
MMODE	3	Machine mode
I	1	Immediate operand
U	1	Unsigned operation
F	1	Floating point operation
DESTINATION	6	Destination register specifier
AUXCODE	4	Auxiliary opcode
SOURCE	6	Source operand register specifier
SOURCE2	6	Second source operand register specifier
IMMEDIATE14	14	Immediate signed/unsigned operand
I8HI	2	High part of 8-bit immediate operand IMMEDIATE8
I8LO	6	Low part of 8-bit immediate operand IMMEDIATE8

- Shift/rotate
- Logic operations
- Bit operations
- Floating point operations
- Fused multiplication-addition/subtraction

Each arithmetic/logic operation is performed on at least one operand residing in a GPR specified by the DESTINATION field. The result of the operation is written back in the same register that supplied the first operand, i.e., the register specified by the DESTINATION field is overwritten after instruction execution. The number of bits written back to the register is specified by MMODE which also determines the width of operation. Each arithmetic/logic instruction also updates the EXECUTION STATUS register (see Subsection 5.2).

An implementation may choose not to implement all of the arithmetic/logic instructions. However, it shall raise the UNIMPLEMENTED INSTRUCTION exception on each encountered unimplemented instruction.

3.2.1 Integer unit

The integer unit executes the integer arithmetic, shift/rotate, logic and bit operations[3].

The second operand comes either from a GPR specified by the SOURCE field, or, as an immediate specified by the IMMEDIATE14 field. The I bit distinguishes the two alternatives:

I=0: REG[DESTINATION] ← REG[DESTINATION] ‹operation› REG[SOURCE]

I=1: REG[DESTINATION] ← REG[DESTINATION] ‹operation› IMMEDIATE14

Signed/unsigned operation

Signed/unsigned operation is specified by setting the bit U to 0/1, respectively. That is, the integer arithmetic instructions can be executed either as signed or unsigned. Here, not only the operands are treated as unsigned, but also the operation is affected, e.g., the ADD/SUB instructions will overflow differently in unsigned operation compared to signed.

Furthermore, for I = 1 the immediate value is sign-extended for U = 0 and zero-extended for U = 1 to the width of the operation specified by MMODE, according to the MSB of the IMMEDIATE14 field. Of course, for 8-bit operation (byte mode), the six MSBs of the immediate operand are truncated (i.e., only the lower 8 bits are taken). This applies not

[3] The reverse bits instruction RVB is considered to be a part of the DSP unit.

only for integer arithmetic instructions, but also for the logic operations. See Table 4 as an example of sign/zero extension and truncation of an immediate value.

However, for shift/rotate, logic and bit instructions, the input operands are always considered by the integer unit to be unsigned. Here, the U bit does not specify signed/unsigned operation but differentiates these instructions (see Table 9), or has another meaning, e.g., arithmetic/-logic shift. That is, for the SL and SR instructions, U = 1 specifies logic shift, while U = 0 specifies arithmetic shift. The arithmetic right shift pulls the MSB, while the logic right shift which pulls 0. On the other side, the arithmetic left shift triggers the OVERFLOW exception on a change of the MSB value, while the logic left shift does not trigger exceptions.

3.2.2 Floating point unit

The FP machine mode is specified by the F bit (not by MMODE). That is, only for the FP instructions the F bit is 1, which also infers the FP machine mode. For non-FP instructions (F=0), the integer machine mode is specified by the MMODE field. Nevertheless, some FP instructions still use the MMODE field: e.g., conversions from integer to FP formats specify the integer width with MMODE.

In conversions from floating point to integer format (FF2I) or vice versa (FI2F) the bit U specifies whether the integer result or source operand, respectively, is signed (U = 0) or unsigned U = 1. while the MMODE field specifies the integer width.

In conversions between floating point formats with different widths by the "extend" (FEXT), and "squeeze" (FSQZ) instructions, the MMODE field specifies the FP width. However, here only the 16-, 32-, 64- and 128-bit formats, i.e., H, W, D, Q and 1 machine modes are possible[4].

3.2.3 DSP unit

The DSP unit is optional and contains the DSP registers whose functionality is here not an object of specification. Subsection 3.1.2 only specifies the inter-register transfer between the GPR and the DSP registers.

Furthermore, the DSP applications can be also supported by instructions. The reverse bits instruction RVB and the fused multiply-add/subtract instructions (both integer and FP) are especially used by DSP applications, e.g., for Fast Fourier Transform (FFT) computations. Therefore, they are called DSP instructions.

The fused multiply-add (MAD) and multiply-subtract (MSU) instructions use a third operand which comes either from a GPR specified by the SOURCE2 field, or, as an immediate specified by the IMMEDIATE8 field. Here too, the I bit distinguishes the two alternatives:

I=0: REG[DESTINATION] $\leftarrow$ REG[DESTINATION] $\times$ REG[SOURCE] $\pm$ REG[SOURCE2]

I=1: REG[DESTINATION] $\leftarrow$ REG[DESTINATION] $\times$ REG[SOURCE] $\pm$ IMMEDIATE8

The signed/unsigned operation of the MAD and MSU instructions is completely the same as for the instructions for integer arithmetic (see Subsection 3.2.1), with the single difference that now the 8-bit IMMEDIATE8 field is used instead of IMMEDIATE14.

However, there are also floating point versions of these instructions (FMAD and FMSU) whose third operand can come only from a GPR specified by SOURCE2, and not from the IMMEDIATE8 field. Actually, no floating point instruction uses an immediate operand.

[4] This is based on the FP format definitions of the IEEE Std 754-2008 standard [3].

3.3 Control instructions

Fig. 8 shows the layout of the control instructions and Table 6 shows the description of their bit fields.

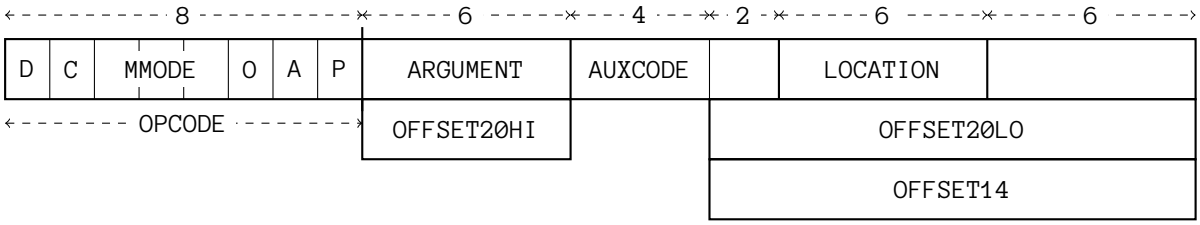


Fig. 8: *Layout of control instructions*

Table 6: *Bit field description of control instructions*

Bit field	Width	Description
D	1	Data transfer (always 0 for control instructions)
C	1	Control (always 1 for control instructions)
MMODE	3	Machine mode
O	1	Offset
A	1	Absolute transfer
P	1	Procedural transfer
ARGUMENT	6	Branch argument register specifier
AUXCODE	4	Auxiliary opcode
LOCATION	6	Location transfer register specifier
OFFSET20HI	6	High part of 20-bit signed (instruction) offset OFFSET20
OFFSET20LO	14	Low part of 20-bit signed (instruction) offset OFFSET20
OFFSET14	14	Signed (instruction) offset

OFFSET20 is a **20-bit signed offset** used for unconditional program transfer. It is a concatenation of the 6-bit wide field OFFSET20HI and the 14-bit wide OFFSET20LO.

The control instructions are used for program control and are of four types:

- Unconditional program transfer
- Conditional (branch) program transfer
- Return from routine
- Pause instruction execution

3.3.1 Program transfer

In program transfers, the opcode bit O specifies whether the value comes from a **register** or from a signed **offset**:

register **O = 0**	The value comes from the GPR specified by the 6-bit LOCATION field of the instruction.
offset **O = 1**	The value comes from the offset field, i.e., OFFSET20 for unconditional, and OFFSET14 for conditional transfers.

If the width of the specified values is shorter than the width of the INSTRUCTION COUNTER, the supplied value is sign-extended. For example, OFFSET14 is 14-bit, while the INSTRUCTION COUNTER is usually greater than 20-bits. In this case, the offset will be sign-extended according to the MSB of OFFSET14. On the other side, if the GPR specified by LOCATION is wider than the INSTRUCTION COUNTER, the LSBs of the GPR will be written to the INSTRUCTION COUNTER.

The program transfers (both conditional and unconditional) could be either **relative** to the INSTRUCTION COUNTER or **absolute**, which is specified by the opcode bit A:

relative **A = 0**	Program execution is transferred to an address obtained by addition of the signed offset or the GPR value to the current value of the INSTRUCTION COUNTER.
absolute **A = 1**	Program execution is transferred to an absolute address contained in the specified GPR or in the signed offset.

Finally, in **procedural** program transfers, the value of the INSTRUCTION COUNTER incremented by 1, is additionally written to the CALL RETURN POINTER (see Subsection 5.14) which can be later used by the *return from routine* instructions (Subsection 3.3.2). The procedural transfer is specified by setting the opcode bit P to 1.

Branching

Table 10 summarizes all control instructions. Branches use the ARGUMENT specifier to investigate the specified GPR whether to make the program transfer or not. For example, the Branch if MSB instruction (BM) checks if the MSB of the GPR is 1 or 0. If 1, the program transfer is made, otherwise the next instruction after BM is fetched.

The MMODE field specifies the machine mode, i.e., the (sub)width of the GPR to be investigated. For example, if the GPR is 32-bit wide, and an 8-bit machine mode is specified (MMODE = 000), then bit 7 of the GPR (not bit 31) will be investigated by the BM instruction. In order to investigate bit 31, MMODE should be set for 32-bit, i.e., MMODE = 010 (see Table 1). Furthermore, for 8-bit machine mode, the Branch if MSB or Zero instruction (BMZ) will additionally check if the eight LSBs are zero. Thus, BMZ will branch if the bit 7 is 1, or if the eight LSBs are all zero. Similarly, for 32-bit machine mode, all GPR bits will be additionally checked if they are all zero.

Branch if (Not) all Ones (BO/BNO) instructions are similar to Branch if (Not) Zero (BZ/BNZ). That is, BZ/BNZ checks whether all the bits of the register (in the specified machine mode) are zero, while BO/BNO checks if all the bits are ones.

3.3.2 Return from routine

A *return from routine* instruction transfers the program execution at an instruction address location specified by the return pointers (see Table 7). The MMODE field distinguishes which return pointer is used.

Table 7: *Return pointers used by 'return from routine' instructions*

MMODE	Instruction	Used return pointer	Return from
000	RET	CALL RETURN POINTER	Procedure
001	RETI	INTERRUPT RETURN POINTER	Interrupt handler
010	RETE	EXCEPTION RETURN POINTER	Exception handler
011	RETN	NMI RETURN POINTER	NMI handler

The CALL RETURN POINTER is written during an execution of a procedural program transfer instruction in which the bit P is 1 (see Subsection 3.3.1). It is written with the address of the instruction following the procedural program transfer instruction that is being executed. On the other hand, the interrupt/exception/NMI return pointers are written automatically upon

entering interrupt/exception/NMI handling with the address of the instruction following the interrupted or exceptional instruction, respectively. However, the return pointers can be also written by an inter-register transfer instruction (see Subsection 3.1.2).

If the bit P is set to 1, the instruction address found in the return pointer is decremented by 1. That is, execution returns to the **previous** instruction address found in the return pointers. At this address is the last executed instruction before the routine call (if the return pointer is not overwritten in the meantime). This is useful, for example, when the same instruction should be re-executed after handling the exception that it caused. Another useful case for a return from routine with P=1 is described in Subsection 3.3.3.

3.3.3 Pause instruction execution

The WAIT instruction pauses instruction execution. The pause period is specified by supplying a wait timer value. Similar to the JMP instruction, the opcode bit 0 specifies whether the wait timer value comes from a GPR or as an immediate value placed in the OFFSET20 field.

If the wait timer value is zero, instruction execution is paused indefinitely, i.e., it can be resumed only by an interrupt (or NMI). After servicing the interrupt, executing a RETI (or RETN) instruction at the end of the handling routine returns the program execution to the instruction following the WAIT instruction. Alternatively, the option P can be specified for the RETI (or RETN) instruction, which will return program execution again to the WAIT instruction in order to wait for the following interrupt.

On the other hand, specifying a non-zero wait timer value loads the wait timer to that value. The wait timer is decremented on each clock cycle. When the wait timer reaches zero, the pause is finished and execution is resumed with the instruction following the WAIT instruction.

Of course, a reset terminates the pause (either indefinite or definite) and an instruction fetch from address zero follows.

3.4 Summary

Table 8 summarizes the data transfer instructions: in places where the L or the U bit is not given, it means that both alternatives of the bit for the corresponding instruction are possible. N/A means that the field is not applicable, i.e., not used by the instruction. The MMODE field is always used by all data transfer instructions and it specifies the data transfer width according to Table 1.

The mnemonic MOV is assigned for all data transfer instructions. In assembly, the differentiation between transfer types is made according to the type and the ordering of the arguments (see Table 68). Subsection 6.4 gives further details on the use of pseudo-mnemonics which can be alternatively used for immediate visual distinction of the data transfer type.

Table 9 summarizes the arithmetic/logic instructions: in places where the I or the U bit is not given, it means that both alternatives of the bit for the corresponding instruction are possible. Similarly, if MMODE is not given, it means that more combinations for the MMODE are possible. Arithmetic/logic instructions always use the AUXCODE field.

Table 10 summarizes the control instructions: in places where the O,A or the P bit is not given, it means that both alternatives of the bit for the corresponding instruction are possible. Similarly, if MMODE is not given, it means that more combinations for the MMODE are possible. Control instructions always use the AUXCODE field.

Table 8: *Summary of data transfer (MOV) instructions*

L	O	U	T	AUXCODE	Description	Used fields
Memory transfer						
	1		0	N/A	REG[DESTINATION] ↔ MEM[BASE+OFFSET12]	DESTINATION, BASE,OFFSET12
	0		0	0000	REG[DESTINATION] ↔ MEM[INDEX]	
	0		0	0001	REG[DESTINATION] ↔ MEM[INDEX++]	DESTINATION,
	0		0	0010	REG[DESTINATION] ↔ MEM[INDEX−−]	AUXCODE,
	0		0	0101	REG[DESTINATION] ↔ MEM[++INDEX]	INDEX
	0		0	0110	REG[DESTINATION] ↔ MEM[−−INDEX]	
	0		0	1000	REG[DESTINATION] ↔ MEM[BASE+INDEX]	
	0		0	1001	REG[DESTINATION] ↔ MEM[BASE+(INDEX++)]	DESTINATION,
	0		0	1010	REG[DESTINATION] ↔ MEM[BASE+(INDEX−−)]	BASE,AUXCODE,
	0		0	1101	REG[DESTINATION] ↔ MEM[BASE+(++INDEX)]	INDEX
	0		0	1110	REG[DESTINATION] ↔ MEM[BASE+(−−INDEX)]	
Inter-register transfer						
1	0	0	1	0000	REG[DESTINATION] ← REG[INDEX]	
1	0	0	1	0001	REG[DESTINATION] ← SPC[INDEX]	
1	0	0	1	0010	SPC[DESTINATION] ← REG[INDEX]	DESTINATION,
1	0	0	1	0011	REG[DESTINATION] ← DSP[INDEX]	AUXCODE,
1	0	0	1	0100	DSP[DESTINATION] ← REG[INDEX]	INDEX
1	0	0	1	1110	REG[DESTINATION] ← FPR[INDEX]	
1	0	0	1	1111	FPR[DESTINATION] ← REG[INDEX]	
Load immediate						
1	1		1	N/A	REG[DESTINATION] ← IMMEDIATE18	DESTINATION, IMMEDIATE18

Thus, in total 53 mnemonics are used for all native instructions. Pseudo-instructions introduce additional pseudo-mnemonics, as described in Subsection 6.4.

Table 9: *Summary of arithmetic/logic instructions*

MMODE	I	U	F	AUXCODE	Mnemonic	Description	Used fields
Integer arithmetic							
			0	0000	ADD	Add	
			0	0001	SUB	Subtract	DESTINATION,
			0	0010	MUL	Multiply	SOURCE/IMMEDIATE14
			0	0011	DIV	Divide	
Shift/rotate							
			0	0100	SL	Shift left (arith./logic)	
			0	0101	SR	Shift right (arith./logic)	DESTINATION,
		0	0	0110	RL	Rotate left	SOURCE/IMMEDIATE14
		1	0	0110	RR	Rotate right	
Logic operations							
			0	0111	AND	AND bitwise	
			0	1000	NAND	Negated AND bitwise	DESTINATION,
			0	1001	OR	OR bitwise	SOURCE/IMMEDIATE14
			0	1010	XOR	Exclusive OR bitwise	
Bit operations							
		0	0	1011	SB	Set bit	
		1	0	1011	RB	Reset bit	DESTINATION,
		0	0	1100	TB	Test bit	SOURCE/IMMEDIATE14
		1	0	1100	RVB	Reverse bits	
Floating point operations							
000	0	0	1	0000	FADD	FP Add	
000	0	0	1	0001	FSUB	FP Subtract	
000	0	0	1	0010	FMUL	FP Multiply	DESTINATION,
000	0	0	1	0011	FDIV	FP Divide	SOURCE
000	0	0	1	0100	FREM	FP Remainder	
000	0	0	1	0101	FCMP	FP Compare	
000	0	0	1	0110	FSQR	FP Square root	
000	0	0	1	0111	FABS	FP Absolute	
000	0	0	1	1000	FNEG	FP Negate	
000	0	0	1	1001	FRND	FP Round to integer	DESTINATION
	0		1	1010	FF2I	FP to integer	
	0		1	1011	FI2F	Integer to FP	
	0	0	1	1100	FEXT	Extend FP format	
	0	0	1	1101	FSQZ	Squeeze FP format	
Fused multiplication-addition/subtraction							
			0	1110	MAD	Multiply-Add	DESTINATION,SOURCE,
			0	1111	MSU	Multiply-Subtract	SOURCE2/IMMEDIATE8
	0	0	1	1110	FMAD	FP Multiply-Add	DESTINATION,
	0	0	1	1111	FMSU	FP Multiply-Subtract	SOURCE,SOURCE2

Table 10: *Summary of control instructions*

MMODE	O	A	P	AUXCODE	Mnemonic	Description	Used fields
Unconditional program transfer							
000				0000	JMP	Jump	LOCATION/ OFFSET20
Conditional (branch) program transfer							
				0001	BZ	Branch if Zero	
				0010	BNZ	Branch if Not Zero	
				0011	BM	Branch if MSB	
				0100	BMZ	Branch if MSB or Zero	
				0101	BNM	Branch if Not MSB	
				0110	BNMO	Branch if Not MSB or all Ones	ARGUMENT,
				0111	BL	Branch if LSB	LOCATION/
				1000	BLZ	Branch if LSB or Zero	OFFSET14
				1001	BNL	Branch if Not LSB	
				1010	BNLO	Branch if Not LSB or all Ones	
				1011	BO	Branch if all Ones	
				1100	BNO	Branch if Not all Ones	
Return from routine							
000	0	0		1101	RET	Return from procedure	N/A
001	0	0		1101	RETI	Return from interrupt handler	
010	0	0		1101	RETE	Return from exception handler	
011	0	0		1101	RETN	Return from NMI handler	
Pause instruction execution							
000		0	0	1110	WAIT	Wait (do not execute)	LOCATION/ OFFSET20

4. EXCEPTIONS AND INTERRUPTS

An **exception** is a special case, situation, event, or deviation from the normal, standard and usual behavior of the instruction or the system, and requires special attention. An **interrupt** is a signal coming from an external component such as an IO device that requires attention. The **interrupt line** is usually driven by an **interrupt controller**.

Exceptions and interrupts can be **disabled** all at once or **masked** individually. A raised exception/interrupt is **potent** if the exceptions/interrupts are enabled, and the exception/interrupt is not masked. Similarly, a raised exception/interrupt is **impotent** if the exceptions/interrupts are disabled or the exception/interrupt is masked.

When a potent exception/interrupt is raised, the sequence of program instructions is interrupted and execution continues at an address specified by special registers. At this address usually an interrupt/exception handler **dispatcher** is placed, which further transfers the execution at the corresponding interrupt/exception handlers. On the other side, an impotent exception/interrupt does not interrupt the sequence of program instructions and the normal execution continues.

4.1 Non-Maskable Interrupt (NMI)

The NMI is a top priority interrupt intended for use in abnormal situations that should (theoretically) never happen, or, for events that are exceptional, highly-important, and occur only rarely. NMI is therefore treated like an exception with highest priority which cannot be masked nor disabled, but it is called an interrupt since it is signaled by a hardware line like the "maskable" interrupts. NMI is immediately handled no matter of processor state, after completion of the interrupted instruction.

The NMI is signaled by a single hardware **NMI line**, which is different from the (maskable) interrupt line. The interrupt controller can also drive the NMI line and enable multiple sources for the NMI.

NMI handling

Fig. 9 shows a program routine (procedure) that is interrupted by an NMI at the p_i instruction.

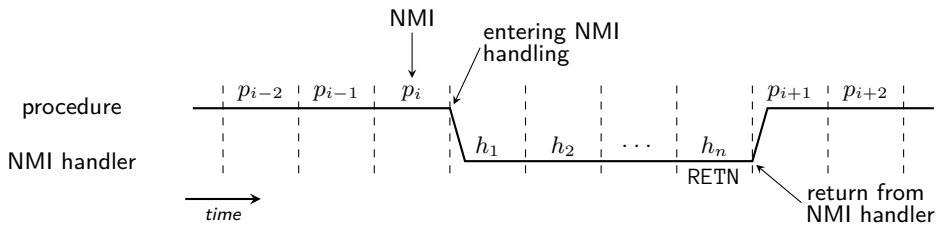


Fig. 9: *NMI handling*

The NMI handler has n instructions $h_1, h_2, \ldots, h_n$. After full completion of the interrupted instruction p_i, the NMI handler is entered. The following instruction which is executed is the h_1 instruction from the NMI handler. The `RETN` instruction (*return from NMI handler*) is the h_n instruction of the NMI handler which returns execution to the procedure at the p_{i+1} instruction.

Upon entering NMI handling:

- The NMI RETURN POINTER is written with the value of the INSTRUCTION COUNTER.
- The EXCEPTION TABLE BASE ADDRESS register specifies the address at which program execution is transferred (see Subsection 5.6).
- The operating mode is automatically switched to system mode.
- Bit 0 in the EXCEPTION REGISTER is set (see Subsection 5.4).
- The EXCEPTION INSTRUCTION register is written with the interrupted instruction (see Subsection 5.3).
- Both exceptions and interrupts are disabled.

Handler interruptability

The NMI handler is interruptable only by another NMI. Of course, if exceptions/interrupts are made potent by the handler itself, they can also interrupt the handler.

Disabling and masking

The NMI can be neither disabled nor masked, i.e., the NMI is always potent.

Acknowledging

Once the NMI is handled, it should be acknowledged by reseting bit 0 in the EXCEPTION REGISTER (see Subsection 5.4), otherwise the execution will enter exception handling of the NMI exception after finishing NMI handling. The acknowledgement should happen before executing the RETN instruction. Furthermore, acknowledging may be required in an implementation-specific manner, e.g., by writing corresponding registers in the entity that is raising the NMI.

Return from handling

The NMI handler is usually terminated with the RETN system instruction which restores back the operating mode and the enabled/disabled status of the exceptions and the interrupt line as they were before NMI handling was entered.

4.2 Exceptions

The PEAKTOP ISA predefines 21 exceptions which are summarized in Table 11. As said, the NMI is treated like an exception with highest priority and is therefore written in Table 11 as exception 0. Up to 64 exceptions may be defined. Exceptions 22 to 63 are left for implementation-specific purposes.

The exceptions are divided into two types according to the source by which they are raised:

instruction-raised — The exception is raised by an **exceptional instruction**, i.e., instruction that can raise exception(s).

hardware-raised — The exception is raised by a hardware mechanism (e.g., debug or memory access mechanism, or the DSP unit).

When a potent, instruction-raised exception occurs, the destination GPRs are **not** written-back by the exceptional instruction since the corresponding handler should decide what is to be done. On the other hand, if an impotent, instruction-raised exception occurs, the exceptional instruction is either *skipped*, or it performs a normal *write-back* to the destination GPR and the EXECUTION STATUS register. A skipped instruction does not change any register (except that the EXCEPTION REGISTER is updated to mark the exception occurrence). For example, a system instruction in user mode is skipped when the SYSTEM INSTRUCTION exception is

Table 11: *Exceptions*

Nr.	Exception	Raised by	If impotent
0	NMI	NMI line	N/A
1	DEBUG MODE EXCEPTION	debug mode	no change
2	INVALID INSTRUCTION	instruction	skipped
3	SYSTEM INSTRUCTION	instruction	skipped
4	UNIMPLEMENTED GPR BANK	instruction	skipped
5	UNIMPLEMENTED INSTRUCTION	instruction	skipped
6	UNIMPLEMENTED REGISTER	instruction	skipped
7	INVALID OPERATION	instruction	skipped/write-back
8	DIVISION BY ZERO	instruction	write-back
9	OVERFLOW	instruction	write-back
10	FP INVALID OPERATION	instruction	write-back
11	FP DENORMALIZED OPERAND	instruction	write-back
12	FP DIVISION BY ZERO	instruction	write-back
13	FP OVERFLOW	instruction	write-back
14	FP UNDERFLOW	instruction	write-back
15	FP INEXACT RESULT	instruction	write-back
16	DSP EXCEPTION 0	DSP unit	no change
17	DSP EXCEPTION 1	DSP unit	no change
18	DSP EXCEPTION 2	DSP unit	no change
19	DSP EXCEPTION 3	DSP unit	no change
20	I SYSTEM BUS ERROR	mem. acc.	undefined, error
21	D SYSTEM BUS ERROR	mem. acc.	no change, error

impotent, but an instruction causing impotent DIVISION BY ZERO or OVERFLOW will write-back an appropriate result (e.g., the maximal possible integer). Table 11 also summarizes this.

On the other side, impotent, hardware-raised exceptions *do not change* the registers, except the EXCEPTION REGISTER which notifies the occurrence of the exception. Of course, if a potent, hardware-raised exception occurs, the currently executing instruction is finished normally, after which the exception handling begins.

Finally, an impotent I SYSTEM BUS ERROR leads to an undefined state of execution in which the behavior of the system is not determined. This state is signaled on the hardware **error line**, and can be only exited by a reset.

Exception handling

Fig. 10 shows a program routine (procedure) that is interrupted by an exception at the p_i instruction.

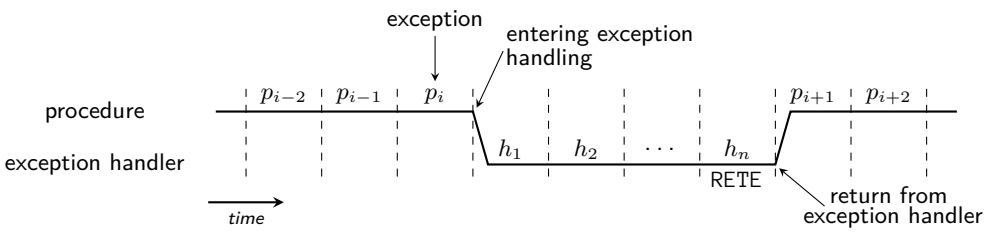


Fig. 10: *Exception handling*

The exception handler has n instructions $h_1, h_2, \ldots, h_n$.If the exception is hardware-raised, the exception handler is entered after full completion of the interrupted instruction p_i. However, if the exception is instruction-raised (by the p_i instruction) then the p_i instruction does not write-back anything to the registers, i.e., the p_i instruction is not completed when the

exception handler is entered (and will not be completed after returning from the handler). This is because it is left to the handler to decide what to do with the exceptional instruction. The following instruction which is executed is the h_1 instruction from the exception handler. The RETE instruction (*return from exception handler*) is the h_n instruction of the exception handler which returns execution to the procedure at the p_{i+1} instruction.

Upon entering exception handling:

- The EXCEPTION RETURN POINTER is written with the value of the INSTRUCTION COUNTER.
- The EXCEPTION TABLE BASE ADDRESS register specifies the address at which program execution is transferred (see Subsection 5.6).
- The operating mode is automatically switched to system mode.
- The corresponding exception bit in the EXCEPTION REGISTER is set (see Subsection 5.4).
- The EXCEPTION INSTRUCTION register is written with the interrupted instruction (see Subsection 5.3).
- Both exceptions and interrupts are disabled.

Handler interruptability

The exception handler is interruptable only by an NMI. Of course, if exceptions/interrupts are made potent by the handler itself, they can also interrupt the handler.

Disabling and masking

All exceptions can be disabled at once (except NMI) by writing a zero to the ENABLE EXCEPTIONS bit in the SYSTEM CONTROL REGISTER (see Subsection 5.10). They are disabled after reset, so this bit should be written with one in order to enable them. Furthermore, exceptions can be masked individually by writing a one to the corresponding bit in the EXCEPTION MASKS register (see Subsection 5.5).

Acknowledging

Once an exception is handled, it should be acknowledged in the EXCEPTION REGISTER (see Subsection 5.4) by reseting its bit, otherwise the dispatcher will select it again for handling (which is already done). The acknowledgement should happen before executing the RETE instruction.

Return from handling

The exception handler is usually terminated with the RETE system instruction which restores back the operating mode and the enabled/disabled status of the exceptions and the interrupt line as they were before exception handling was entered.

4.2.1 DEBUG MODE EXCEPTION

The exception is raised by the debug mode mechanism. The debug mode is activated by writing a one to the DEBUG MODE bit in the SYSTEM CONTROL REGISTER (see Subsection 5.10). In debug mode, this exception is raised and exception handling starts after each executed and fully-completed instruction. However, the handler is uninterrupted since exceptions and interrupts are automatically disabled when handling starts.

Raised by...

Hardware-raised: debug mode mechanism

If impotent...

The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made. However, debug mode is practically disabled if this exception is masked.

4.2.2 INVALID INSTRUCTION

An attempt to execute an invalid instruction raises this exception. Any combination of 32 bits that is read as an instruction but cannot be decoded as a valid instruction, i.e., it does not comply to the specification in this document, raises this exception.

Raised by...

Instruction-raised: invalid (out-of-specification) instruction.

If impotent...

The exceptional instruction is skipped. The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.3 SYSTEM INSTRUCTION

An attempt to execute a system instruction in user mode raises this exception.

Raised by...

Instruction-raised: by RETE, RETN and by any inter-register transfer (MOV) executed in user mode in which the destination is a special register that cannot be written in user mode (see Table 12).

If impotent...

The exceptional instruction is skipped. The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.4 UNIMPLEMENTED GPR BANK

An attempt to specify an unimplemented (physically non-existing) GPR bank raises this exception.

Raised by...

Instruction-raised: by an inter-register transfer (MOV) from a GPR to the SYSTEM CONTROL REGISTER (see Subsection 5.10) in which the GPR BANK field (bits 4 through 7) is changed.

If impotent...

The exceptional instruction is skipped. The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.5 UNIMPLEMENTED INSTRUCTION

An attempt to execute a valid but unimplemented instruction raises this exception. In other words, this exception is raised when the the specific implementation does not implement the

instruction.

Raised by...

Instruction-raised: by any unimplemented instruction.

If impotent...

The exceptional instruction is skipped. The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.6 UNIMPLEMENTED REGISTER

An attempt to specify an unimplemented register in any of the 6-bit instruction fields for register specification raises this exception. For example, an instruction that specifies a GPR with number greater than 31 in an implementation which has a GPR file with 32 registers will raise this exception. This holds for any register file, not only for the GPR. Furthermore, this exception is raised also in the case when the entire register file is not implemented. For example, if the DSP file is not implemented, and the instruction specifies access to a DSP register.

Raised by...

Instruction-raised: by any instruction that specifies an unimplemented register.

If impotent...

The exceptional instruction is skipped. The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.7 INVALID OPERATION

An attempt to perform an invalid operation specified by a (non-FP) instruction raises this exception.

Raised by...

Instruction-raised: by bit operations SB, RB, TB and RVB when the second argument specifies a bit weight which is greater than the specified machine mode or greater than the ALU width. Furthermore, when operating in system mode, it is raised by an inter-register transfer instruction (MOV) from a GPR to a non-writable special register (see Table 12).

If impotent...

The exceptional instruction is skipped. However, if the exception is raised by an arithmetic/-logic instruction, an undefined result may be written to the destination GPR. The INVALID OPERATION bit in the EXECUTION STATUS register is set (see Subsection 5.2), provided that the exception is raised by an arithmetic/logic instruction. The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.8 DIVISION BY ZERO

An integer division when the divisor is zero raises this exception.

Raised by...

Instruction-raised: by the integer division instruction DIV.

If impotent...

The result of division by zero is written to the destination GPRs: the quotient is the maximal representable (signed/unsigned) number in the specified machine mode, while the remainder is zero. The DIVISION BY ZERO bit in the EXECUTION STATUS register is set (see Subsection 5.2). The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4).

4.2.9 OVERFLOW

An integer operation which results in a number that is greater than the maximal representable number in the specified machine mode raises this exception.

Raised by...

Instruction-raised: by the addition/subtraction instructions (ADD, SUB), by signed division with DIV in which the dividend is the minimal representable number in the specified machine mode and the divisor is –1, and by the arithmetic left shift (SL) when the MSB changes state.

If impotent...

The result of the operation is written to the destination GPRs. That is, the maximal representable (signed/unsigned) number is written for ADD, SUB and for the quotient of DIV, while the remainder of DIV is zero. For arithmetic left shift, the result is the same as if logic left shift was executed. The OVERFLOW bit in the EXECUTION STATUS register is set (see Subsection 5.2). The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4).

4.2.10 FP INVALID OPERATION

An attempt to perform an invalid operation specified by an FP instruction raises this exception.

Raised by...

Instruction-raised:

- by any FP instruction in which one or more operands is a signaling Not a Number (NaN), or the result is NaN;
- by FADD, FSUB, FMAD or FMSU when the operands imply addition or subtraction of infinities with opposite signs, e.g., positive infinity plus negative infinity;
- by FMUL, FMAD or FMSU when one multiplication operand is zero and the other is infinity;
- by FDIV when both operands are zero or both are infinity;
- by FREM when none of the operands is NaN, and the first operand is infinity or the second operand is zero;
- by FSQR when the operand is less than zero;
- by FF2I when the operand is infinity or NaN, or, when the operand is greater than the maximal representable integer in the specified machine mode.

See page 37 of the IEEE Std 754-2008 standard [3] for further explanation.

If impotent...

A quiet NaN is written to the destination GPR if the format of the result is an FP number. If the format of the result is integer (e.g., FF2I raises the exception) then the instruction is skipped and no change to the destination GPR is made. The INVALID OPERATION bit in the EXECUTION STATUS register is set (see Subsection 5.2). The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4).

4.2.11 FP DENORMALIZED OPERAND

An FP operation in which one or more operands is a denormalized FP number raises this exception. However, if the operands are not denormalized and the instruction result is denormalized this exception is not raised.

Raised by...

Instruction-raised: all FP instructions except integer to FP conversion (FI2F).

If impotent...

The result of the operation is written to the destination GPR. The DENORMALIZED bit in the EXECUTION STATUS register is set (see Subsection 5.2). The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4).

4.2.12 FP DIVISION BY ZERO

An FP division when the divisor is zero and the dividend is a finite, nonzero FP number raises this exception (see page 37 in [3]).

Raised by...

Instruction-raised: by the FP division instruction FDIV.

If impotent...

The result ($\pm$infinity) is written to the destination GPR. The DIVISION BY ZERO bit in the EXECUTION STATUS register is set (see Subsection 5.2). The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4).

4.2.13 FP OVERFLOW

An FP operation in which the result (either FP or integer) exceeds the largest representable finite number of the destination format. (see page 37 in [3]).

Raised by...

Instruction-raised: by the FP arithmetic instructions FADD, FSUB, FMUL, FDIV, FMAD and FMSU, by round to integer FRND, and by the FP conversion instructions FF2I, FI2F and FSQZ.

If impotent...

The result of the operation is written to the destination GPR: the result is either $\pm$infinity or the most negative/positive number of the destination format, depending on the FP rounding mode. The OVERFLOW bit in the EXECUTION STATUS register is set (see Subsection 5.2). The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4).

4.2.14 FP UNDERFLOW

An FP operation in which the result is a tiny non-zero number raises this exception (see page 38 in [3]).

Raised by...

Instruction-raised: by the FP instructions FADD, FSUB, FMUL, FDIV, FSQR, FSQZ, FMAD and FMSU.

If impotent...

The rounded result is written to the destination GPR. The UNDERFLOW bit in the EXECUTION STATUS register is set (see Subsection 5.2) only if the result is inexact, in which case also the FP INEXACT RESULT exception is raised. If the result is exact, although the result is tiny (and denormalized), the UNDERFLOW bit is not set and the FP INEXACT RESULT exception is not signaled. The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4).

4.2.15 FP INEXACT RESULT

An FP operation in which the rounded result is not exact raises this exception. Furthermore, this exception is also raised when the result overflows and the FP OVERFLOW exception is impotent. Similarly, it is also raised when an inexact result underflows and the FP UNDERFLOW is impotent. (see pages 37 and 38 in [3]).

Raised by...

Instruction-raised: by the FP instructions FADD, FSUB, FMUL, FDIV, FSQR, FRND, FF2I, FI2F, FSQZ, FMAD and FMSU.

If impotent...

The rounded (or overflowed) result is written to the destination GPR. The INEXACT bit in the EXECUTION STATUS register is set (see Subsection 5.2). The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4).

4.2.16 DSP EXCEPTION 0

The exception is implementation-specific and is raised by the DSP unit.

Raised by...

Hardware-raised: by the DSP unit.

If impotent...

The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.17 DSP EXCEPTION 1

The exception is implementation-specific and is raised by the DSP unit.

Raised by...

Hardware-raised: by the DSP unit.

If impotent...

The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.18 DSP EXCEPTION 2

The exception is implementation-specific and is raised by the DSP unit.

Raised by...

Hardware-raised: by the DSP unit.

If impotent...

The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.19 DSP EXCEPTION 3

The exception is implementation-specific and is raised by the DSP unit.

Raised by...

Hardware-raised: by the DSP unit.

If impotent...

The exceptional instruction is written in the EXCEPTION INSTRUCTION register (see Subsection 5.3), and the exception is noted in the EXCEPTION REGISTER (see Subsection 5.4). No other register change is made.

4.2.20 I SYSTEM BUS ERROR

The exception is raised during instruction fetch. Read/write/execute access violation is a common cause of this exception raised by the MMU/MPU, or by external memory or IO devices. Furthermore, parity error checks, or other implementation-specific mechanisms can also raise this exception.

Raised by...

Hardware-raised: by the memory/IO access mechanism.

If impotent...

The state of execution is undefined. Error is signaled on the error line.

4.2.21 D SYSTEM BUS ERROR

The exception is raised during a memory data access. Read/write/execute access violation is a common cause of this exception raised by the MMU/MPU, or by external memory or IO devices. Furthermore, parity error checks, or other implementation-specific mechanisms can also raise this exception.

Raised by...

Hardware-raised: by the memory/IO access mechanism.

If impotent...

The state of execution is defined, but error is signaled on the error line anyway.

4.3 Interrupts

An Interrupt Request (IRQ) is signaled over a single hardware line called the interrupt line.

Interrupt handling

Fig. 11 shows a program routine (procedure) that is interrupted by an interrupt at the p_i instruction.

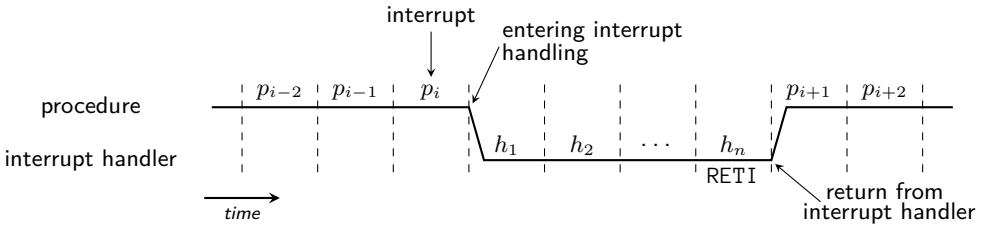


Fig. 11: *Interrupt handling*

The interrupt handler has n instructions $h_1, h_2, \ldots, h_n$. After full completion of the interrupted instruction p_i, the interrupt handler is entered. The following instruction which is executed is the h_1 instruction from the interrupt handler. The `RETI` instruction (*return from interrupt handler*) is the h_n instruction of the interrupt handler which returns execution to the procedure at the p_{i+1} instruction.

Upon entering interrupt handling:

- The `INTERRUPT RETURN POINTER` is written with the value of the `INSTRUCTION COUNTER`.
- The `INTERRUPT TABLE BASE ADDRESS` register specifies the address at which program execution is transferred (see Subsection 5.7).
- The operating mode is automatically switched to system mode.
- The interrupts are disabled, while the enabled/disabled status of the exceptions is not changed.

Handler interruptability

The interrupt handler is interruptable by an NMI and by potent exceptions. Of course, if interrupts are made potent by the handler itself, they can also interrupt the handler.

Disabling and masking

The interrupt line can be disabled by writing a zero to the `ENABLE INTERRUPTS` bit in the `SYSTEM CONTROL REGISTER` (see Subsection 5.10). The interrupt line is disabled after reset, so this bit should be written with one in order to enable it. The individual masking of the interrupts is done in the interrupt controller, which is here not an object of specification.

Acknowledging

Acknowledging the interrupts (if required) is done by writing corresponding registers in the interrupt controller and/or the interrupt-requesting device.

Return from handling

The interrupt handler is usually terminated with the RETI system instruction which restores back the enabled/disabled status of the interrupt line as it was before interrupt handling was entered.

4.4 Handling mechanism

The following points summarize the NMI, exception and interrupt handling mechanism:

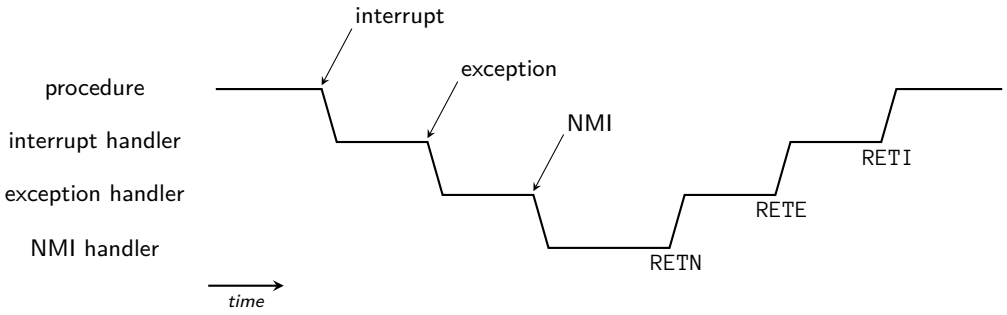
- Exceptions and interrupts are disabled automatically upon entering NMI or exception handling, i.e., bits 2 and 3 (ENABLE EXCEPTIONS and ENABLE INTERRUPTS) in the SYSTEM CONTROL REGISTER are reset (see Subsection 5.10).
- Interrupts are disabled automatically upon entering NMI, exception or interrupt handling, i.e., bit 3 (ENABLE INTERRUPTS) in the SYSTEM CONTROL REGISTER is reset.
- System mode is automatically switched upon entering NMI, exception or interrupt handling, i.e., bit 0 (SYSTEM MODE) in the SYSTEM CONTROL REGISTER is set. The system should decide when to switch to user mode. For instance, user mode could be switched after the system reads the interrupt controller information and the address of the handling procedure is known.
- The system should acknowledge exceptions by resetting the corresponding bit(s) in the EXCEPTION REGISTER (see Subsection 5.4). It may be also required to acknowledge the interrupts to the interrupt controller and/or to the interrupt-requesting device. A privileged access (in system mode) to the interrupt controller and/or device may be required.
- The RETE and RETN instructions restore the SYSTEM CONTROL REGISTER bits 0, 2 and 3 to the state before entering exception or NMI handling, while the RETI instruction restores only bit 3. However, if a RETE/RETN instruction is executed out of exception or NMI handling (i.e., in a program procedure), or RETI is executed out of interrupt handling, the SYSTEM CONTROL REGISTER bits are not changed. RETE and RETN are system instructions, while RETI is not.

4.4.1 Hierarchy and priority

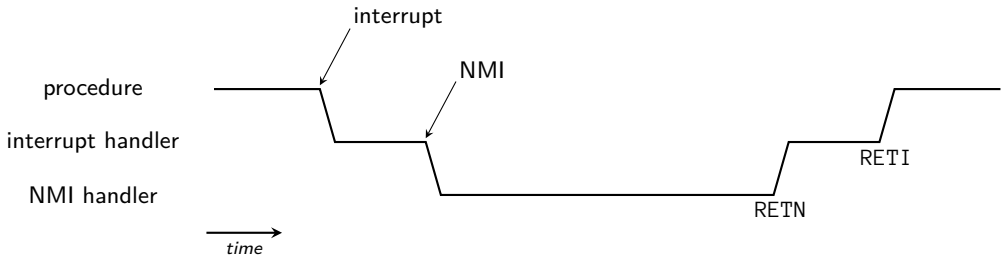
Assuming that exceptions and interrupts are potent (initially), and that the NMI, exception and interrupt handlers do not contain instructions that change the ENABLE EXCEPTIONS and ENABLE INTERRUPTS bits in the SYSTEM CONTROL REGISTER (see Subsection 5.10), an exception can interrupt the execution of an interrupt handler. A NMI can interrupt any routine including NMI handler (see Fig. 12).

It is also possible that several exceptions are raised at once. The exception handler dispatcher routine should then decide on the priority of the multiple exceptions that are raised, and transfer execution to the corresponding exception handler. After acknowledging the exception and executing RETE, the next exception (of the multiple raised) is selected for handling by the dispatcher, until all exceptions are handled. Fig. 13 shows a case in which two exceptions are raised simultaneously at instruction p_i.

They are handled one after another, without executing procedure instructions in-between. The handlers of exception 1 and 2 have n and m instructions, respectively. After handling both exceptions serially (one after another), the execution of the procedure continues at instruction p_{i+1}.



(a) Exception interrupting interrupt handler and NMI interrupting exception handler



(b) NMI interrupting interrupt handler

Fig. 12: *Hierarchy of NMI, exceptions and interrupts*

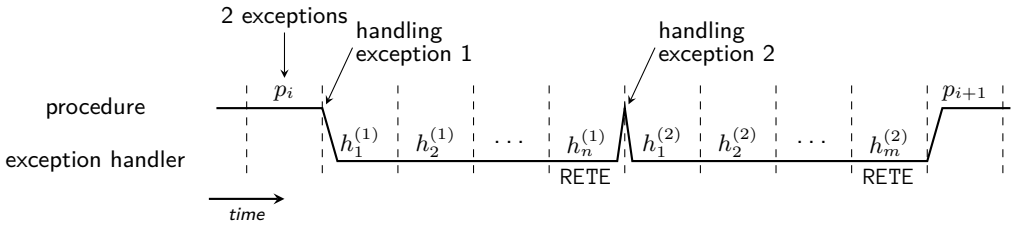


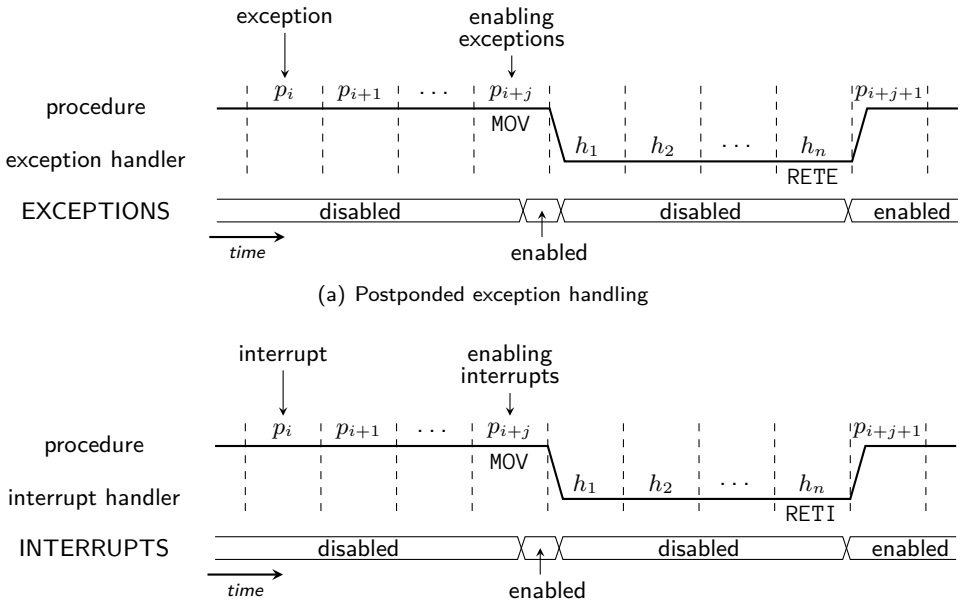
Fig. 13: *Multiple exceptions raised simultaneously*

4.4.2 Postponed handling

The occurrence of impotent exceptions is marked in the EXCEPTION REGISTER (see Subsection 5.4). Once the exceptions become potent, the exception handling mechanism is activated. Here too, the dispatcher decides on the priority if multiple (impotent) exceptions are marked in the EXCEPTION REGISTER. Similarly, if the interrupt line is disabled, a signaled interrupt is still marked internally. Once the interrupt line is enabled, the interrupt handling mechanism is activated.

Fig. 14 shows an exception/interrupt that is raised at instruction p_i when the exceptions/interrupts are disabled. Their enabling is done j instructions later by an inter-register transfer instruction p_{i+j} (MOV) which sets the ENABLE EXCEPTIONS/ENABLE INTERRUPTS bit in the SYSTEM CONTROL REGISTER. After full execution of this instruction, the exception/interrupt handling is done immediately, at which point, as said, the exceptions/interrupts are disabled (i.e., the ENABLE EXCEPTIONS/ENABLE INTERRUPTS bit is reset automatically). At the end of handling, the RETE/RETI instruction restores back the value of the ENABLE EXCEPTIONS/ENABLE INTERRUPTS bit as it was before entering exception/interrupt handling, which in this case is set by the MOV instruction to 1. It is also here assumed that the exception/interrupt handlers do not contain instructions that change the ENABLE EXCEPTIONS/ENABLE INTERRUPTS bit.

In this way, the execution of the exception/interrupt handlers is postponed: instead of ex-



(a) Postponded exception handling

(b) Postponed interrupt handling

Fig. 14: *Postponed execution of exception and interrupt handlers*

ecuting in the time of their occurrence, the execution happens after enabling the exceptions/interrupts. Postponing the handling of an individual exception is done by setting its mask bit in the EXCEPTION MASKS register (see Subsection 5.5) instead of disabling all exceptions.

Finally, NMI handling cannot be postponed.

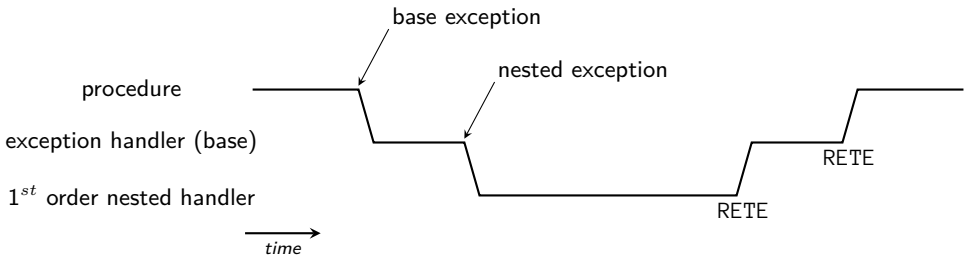
4.4.3 Nesting

As said, upon entering exception handling, the exceptions are disabled. However, the exception handler may decide to enable them. If a potent exception occurs during the execution of a previously invoked exception handler, the handler is interrupted and exception handling is re-entered. This is called **exception nesting**, and the latter exception is a **nested exception** of first order, while the first exception whose handler was executing is the **base exception**. If the nested exception is interrupted by another potent exception, it is called a nested exception of second order, etc.

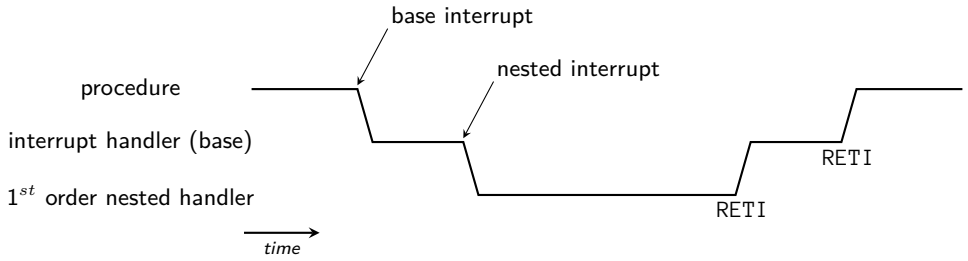
The same discussion holds for the interrupts, i.e., the execution of an interrupt handler may be interrupted by a **nested interrupt**, which is called **interrupt nesting**. Fig. 15 shows the nesting of exceptions and interrupts.

Nesting of exceptions and interrupts requires extra software since the return pointers (EXCEPTION RETURN POINTER/INTERRUPT RETURN POINTER registers) are automatically overwritten upon each re-entering into the exception/interrupt handling routines. Therefore, before making the exceptions/interrupts potent, the handler should preserve the current return pointers in order to be able to return to the correct addresses after handling the nested exceptions/interrupts. Nevertheless, a potent interrupt during exception or NMI handling, or potent exception during interrupt or NMI handling does not require extra software since the return pointers are different for NMI, exceptions and interrupts, and the correct use of the *return from routine* instructions is sufficient (see Fig. 12).

A **nested NMI**, however, is difficult to handle since the extra software required to save the return pointers may even not be executed when the NMI handler is interrupted by a nested NMI (due to the inability to disable or mask the NMI). Therefore, system designers should



(a) Nested exception handling



(b) Nested interrupt handling

Fig. 15: *Nesting exceptions and interrupts*

take into consideration whether nested NMI should be allowed, and if so, how they should be handled.

5. SPECIAL REGISTERS

According to Table 2 there are 16 predefined special registers. Their width can differ between implementations. However, in order to ensure greater portability of programs, a 32-bit width of the special register file is recommended, in which the special registers are either 32-bit or 64-bit wide (see Table 12). A 64-bit special register is composed of two 32-bit registers in the special register file in which the more significant part is with higher enumeration (little-endian ordering). In total, the 16 predefined special registers occupy 27 32-bit wide registers since 11 of them are 64-bit wide. Using the principle of circularity (see Subsection 2.2.3) the 64-bit registers can be accessed at once, or, the higher and lower parts can be accessed individually.

This recommendation limits the physical address space to 64 bits since the return pointers and exception/interrupt base addresses are 64-bit wide registers!

Table 12 also shows the recommended aliases of the special registers (i.e., additional register names) to be used. However, the register names spc0, spc1,. . . , spc63 should be valid for all implementations.

Table 12: *Recommended register aliases and access permissions in user/system mode of a 32-bit wide special register file*

Name	Alias	Register	User acc.	Sys. acc.
spc0	IMP	IMPLEMENTATION REGISTER	r-	r-
spc1	IMP2			
spc2	EST	EXECUTION STATUS	r-	rw
spc3	EXI	EXCEPTION INSTRUCTION	r-	rw
spc4	EXC	EXCEPTION REGISTER	r-	rw
spc5	EXC2			
spc6	EXM	EXCEPTION MASKS	r-	rw
spc7	EXM2			
spc8	ETB	EXCEPTION TABLE BASE ADDRESS	r-	rw
spc9	ETB2			
spc10	ITB	INTERRUPT TABLE BASE ADDRESS	r-	rw
spc11	ITB2			
spc12	CID	CORE ID	r-	rw
spc13	PID	PROCESS ID	r-	rw
spc14	PID2			
spc15	SCR	SYSTEM CONTROL REGISTER	r-	rw
spc16	NRP	NMI RETURN POINTER	r-	rw
spc17	NRP2			
spc18	ERP	EXCEPTION RETURN POINTER	r-	rw
spc19	ERP2			
spc20	UCR	USER CONTROL REGISTER	rw	rw
spc21	CRP	CALL RETURN POINTER	rw	rw
spc22	CRP2		rw	rw
spc23	IRP	INTERRUPT RETURN POINTER	rw	rw
spc24	IRP2			
spc25	DCR	DSP CONFIGURATION REGISTER	rw	rw
spc26	DCR2			

Furthermore, Table 12 shows the read/write access permissions in user and system mode. In user mode, prohibited write operation to a register triggers the SYSTEM INSTRUCTION

exception. In system mode, prohibited write operation to a register triggers the INVALID OPERATION exception. Specifying a non-existing register triggers the UNIMPLEMENTED REGISTER exception.

[<hi>:<lo>] denotes a range of bits within a register (a bit field) starting from bit number <hi> down to bit number <lo>.

Reset state

All special registers except the IMPLEMENTATION REGISTER (see Subsection 5.1) and the SYSTEM MODE bit in the SYSTEM CONTROL REGISTER (see Subsection 5.10) are reset to zero on system reset.

5.1 IMPLEMENTATION REGISTER

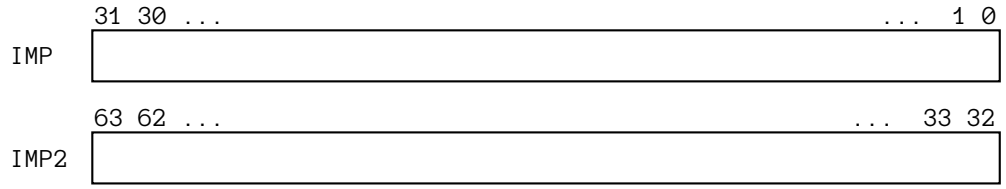


Fig. 16: *IMPLEMENTATION REGISTER*

The IMPLEMENTATION REGISTER (Fig. 16) is a read-only register both in system and in user mode. Its contents is fixed during design time and reflects the properties of the implementation. The bits and bit-fields of the IMPLEMENTATION REGISTER are as follows.

[2:0] **Maximal transfer width** (see Subsection 3.1). The machine mode encoding (see Table 1) in this field shows the maximal transfer width.

[5:3] **GPR width** (see Subsection 2.2.4). The machine mode encoding (see Table 1) in this field shows the GPR width.

[8:6] **ALU width**. The machine mode encoding (see Table 1) in this field shows the ALU width.

[14:9] **Physical address width**. The binary number in this field incremented by one gives the physical address width of the implementation.

15 **Separate instruction and data interface**. If this bit is 1, the implementation has separate instruction and data interface for memory/cache access. If this bit is 0, the implementation uses a single interface to memory/cache for both instruction and data.

[19:16] **FPU type**:

0000	No FPU	
0001	Halfword FPU	(16-bit FP format)
0010	Single FPU	(32-bit FP format)
0011	Double FPU	(64-bit FP format)
0100	Quadruple FPU	(128-bit FP format)
0101	Octuple FPU	(256-bit FP format)
1001	Extended halfword FPU	(> 16-bit FP format)
1010	Extended single FPU	(> 32-bit FP format)
1011	Extended double FPU	(> 64-bit FP format)
1100	Extended quadruple FPU	(> 128-bit FP format)
1101	Extended octuple FPU	(> 256-bit FP format)

[31:20] **Architecture number (low part)**. This number is used for further differentiation between implementations.

[37:32] **Number of GPRs**. The binary number in this field incremented by one gives the number of GPRs in the implementation.

[43:38] **PROCESS ID width**. The binary number in this field incremented by one gives the number of used bits in the PROCESS ID register (see Subsection 5.9). Thus, the number in this field plus the number contained in the field [14:9] plus two, gives the virtual address width of the implementation.

45 **Multiplier present**. If this bit is 1 it shows that a multiplier is present in the implementation which executes the MUL instruction. If this bit is 0 there is no multiplier in the implementation and execution of a MUL instruction raises the UNIMPLEMENTED INSTRUCTION exception.

46 **Divider present**. If this bit is 1 it shows that a divider is present in the implementation which executes the DIV instruction. If this bit is 0 there is no divider in the implementation and execution of a DIV instruction raises the UNIMPLEMENTED INSTRUCTION exception.

[49:47] **DSP width**. The binary number in this field shows the width of the DSP file. The encoding used to specify this width is according to Table 1.

[55:50] **Number of DSP registers**. The binary number in this field incremented by one gives the number of DSP registers in the implementation.

[63:56] **Architecture number (high part)**. This number is used for further differentiation between implementations.

5.2 EXECUTION STATUS

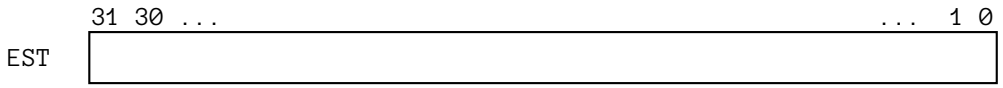


Fig. 17: *EXECUTION STATUS*

The EXECUTION STATUS register (Fig. 17) contains the flags of execution of arithmetic/-logic instructions. It is automatically updated with the completion of any arithmetic/logic instruction. However, if the instruction raises a potent exception, this register (and the destination GPR) are not updated since exception handling is entered. Data transfer and control instructions do not affect this register. The flags in the register are the following.

0 **INVALID OPERATION** is set by SB, RB, TB and RVB when the second argument specifies a bit weight which is greater than the specified machine mode or greater than the ALU width. Furthermore, it is also set whenever the conditions for raising the FP INVALID OPERATION exception are satisfied (see Subsection 4.2.10).

1 **UNIMPLEMENTED OPERATION** is set to 1 when the specified arithmetic/logic operation is not implemented.

2 **DIVISION BY ZERO** is set to 1 by DIV and FDIV when the divisor is zero.

3 **OVERFLOW** is set to 1 by ADD and SUB whenever the computed result cannot fit to the destination GPR according to the specified machine mode.

4 **UNDERFLOW** is set to 1 by FP instructions when the result is a tiny, inexact FP number.

5 **EQUAL** is set to 1 by all arithmetic/logic instructions with two operands when the operands are equal.

6 **GREATER THAN** is set to 1 by all arithmetic/logic instructions with two operands when the first operand is greater than the second operand.

7 **LESS THAN** is set to 1 by all arithmetic/logic instructions with two operands when the first operand is less than the second operand.

8 **INEXACT** is set to 1 by FP instructions when the produced result is inexact.

9 **UNORDERED** is set to 1 by FCMP when at least one of the operands is NaN.

10 **SIGN** is set to 1 if the result of the operation is a negative number (either integer or FP).

11 **ZERO** is set to 1 if the result of the operation is zero (either integer or FP).

12 **DENORMALIZED** is set to 1 if the result of the FP operation is denormalized.

13 **INFINITY** is set to 1 if the result of the FP operation is infinity.

14 **SIGNALING NAN** is set to 1 if the result of the FP operation is a signaling NaN.

15 **NAN** is set to 1 if the result of the FP operation is a NaN.

The bits in the range [31:16] are not used.

5.3 EXCEPTION INSTRUCTION

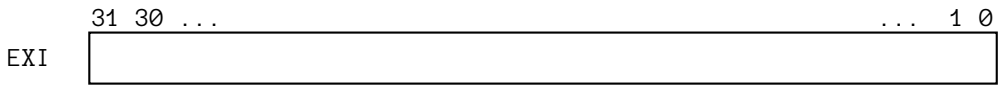


Fig. 18: *EXCEPTION INSTRUCTION*

The EXCEPTION INSTRUCTION register (Fig. 18) contains the exceptional instruction that caused the exception. Furthermore, if a hardware-raised exception or NMI occurred, this register contains the instruction following the last completed instruction before entering exception/NMI handling. An impotent exception does not update this register.

5.4 EXCEPTION REGISTER

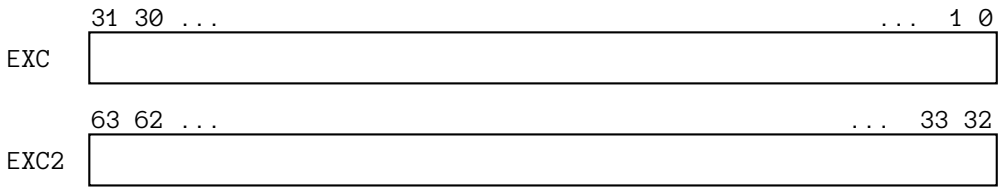


Fig. 19: *EXCEPTION REGISTER*

The corresponding bit for each raised exception is set to 1 in the EXCEPTION REGISTER (Fig. 19). The ordering of the bits in this register is as in Table 11. Thus, the bits in the range [63:22] are reserved for implementation-specific exceptions.

The exception handler should acknowledge the exception by resetting the corresponding bit to 0 in this register. The NMI handler should acknowledge the NMI by resetting bit 0.

5.5 EXCEPTION MASKS

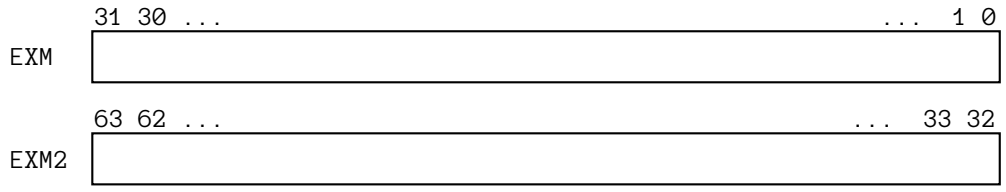


Fig. 20: *EXCEPTION MASKS*

The exception masks are defined in the EXCEPTION MASKS register (Fig. 20). Setting the bit to 1 sets the mask of the corresponding exception and makes it impotent. The ordering of the bits in this register is as in Table 11. Thus, the bits in the range [63:22] are reserved for implementation-specific exceptions.

However, bit 0 is for the NMI and is always zero, i.e., it cannot be set to one since the NMI is not maskable.

5.6 EXCEPTION TABLE BASE ADDRESS

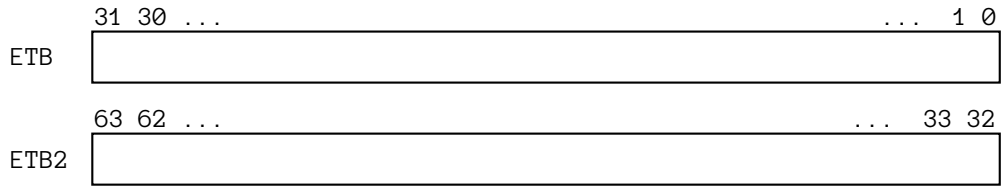


Fig. 21: *EXCEPTION TABLE BASE ADDRESS*

The EXCEPTION TABLE BASE ADDRESS register (Fig. 21) contains the address at which program execution is transferred upon entering NMI/exception handling. That is, it contains the base address of the exception handler dispatcher.

5.7 INTERRUPT TABLE BASE ADDRESS

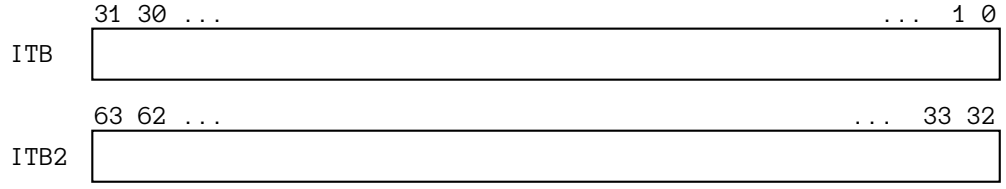


Fig. 22: *INTERRUPT TABLE BASE ADDRESS*

The INTERRUPT TABLE BASE ADDRESS register (Fig. 22) contains the address at which program execution is transferred upon entering interrupt handling. That is, it contains the base address of the interrupt handler dispatcher.

5.8 CORE ID

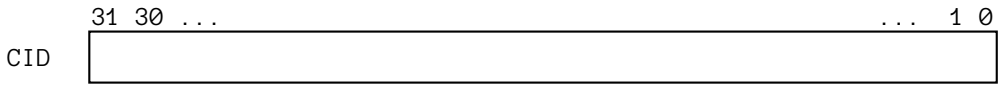


Fig. 23: *CORE ID*

The CORE ID register (Fig. 23) contains the ID of the processing element (core) which is relevant for multiprocessing environments.

5.9 PROCESS ID

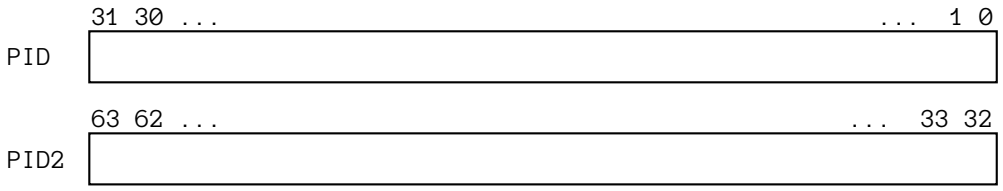


Fig. 24: *PROCESS ID*

The PROCESS ID register (Fig. 24) contains the high part of the virtual address, or viewed alternatively, the ID of the currently executing process.

Writing the PROCESS ID register resets the INSTRUCTION COUNTER to zero. The operating system should take care of jumping to the next instruction on context-switch.

5.10 SYSTEM CONTROL REGISTER

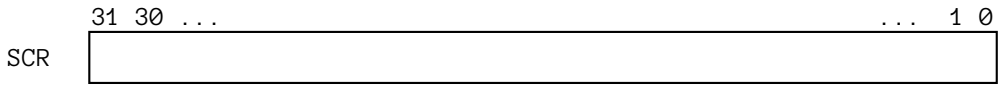


Fig. 25: *SYSTEM CONTROL REGISTER*

The SYSTEM CONTROL REGISTER (Fig. 25) has several functions that control the behavior of the system. It is only writable in system mode. The bits and bit-fields of the register are as follows.

0 SYSTEM MODE. If 1, the system mode is set. If 0, the user mode is set. This bit is set to 1 after reset, i.e., the system starts in system mode.

1 DEBUG MODE. If 1, the debug mode is set, in which the DEBUG MODE EXCEPTION is raised after each executed instruction. For normal operation this bit should be 0.

2 ENABLE EXCEPTIONS. If 1, the exceptions are enabled. If 0, the exceptions are disabled.

3 ENABLE INTERRUPTS. If 1, the interrupts are enabled. If 0, the interrupts are disabled.

[7:4] GPR BANK. The number of the currently used GPR bank. Theoretically, up to 16 banks can be implemented, but practically two to four banks are implemented. Specifying a number greater than or equal to the number of implemented banks raises the UNIMPLEMENTED GPR BANK exception.

The bits in the range [31:8] are not used.

5.11 NMI RETURN POINTER

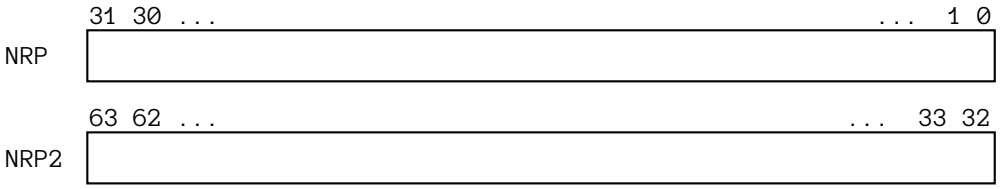


Fig. 26: *NMI RETURN POINTER*

The NMI RETURN POINTER register (Fig. 26) contains the address of the last executed instruction before NMI handling was entered. It is automatically written on entering NMI handling. The RETN instruction uses this pointer to return to the correct place of execution at the end of the NMI handler.

5.12 EXCEPTION RETURN POINTER

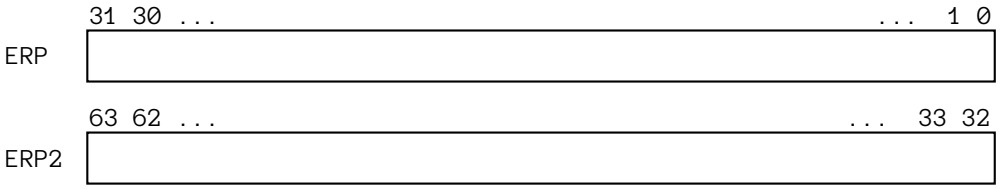


Fig. 27: *EXCEPTION RETURN POINTER*

The EXCEPTION RETURN POINTER register (Fig. 27) contains the address of the last executed instruction before exception handling was entered. It is automatically written on entering exception handling. The RETE instruction uses this pointer to return to the correct place of execution at the end of the exception handler.

5.13 USER CONTROL REGISTER

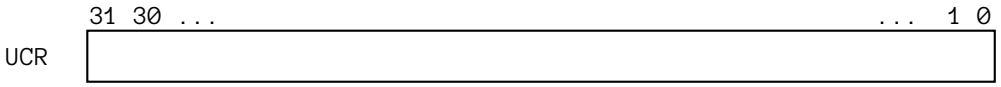


Fig. 28: *USER CONTROL REGISTER*

The USER CONTROL REGISTER (Fig. 28) adds more functions to the SYSTEM CONTROL REGISTER (see Subsection 5.10) that control the behavior of the system, but it is also writable in user mode. The bits and bit-fields of the register are as follows.

0 DONT CACHE INSTRUCTIONS. If 1, the system does not use the instruction cache. If 0, the system uses the instruction cache.

1 DONT CACHE DATA. If 1, the system does not use the data cache. If 0, the system uses the data cache.

2 DONT BUFFER INSTRUCTIONS. If 1, the system does not buffer the instructions. If 0, the system buffers the instructions.

3 DONT BUFFER DATA. If 1, the system does not buffer the data. If 0, the system buffers the data.

4 SYNC. The state of this bit is reflected on the output **sync line** used for synchronization purposes in multiprocessing environments.

[11:8] FP ROUNDING MODE:

0000	Nearest – even
0001	Nearest – odd
0010	Nearest – zero away
0011	Nearest – positive infinity
0100	Nearest – negative infinity
0101	Nearest – zero
1010	Zero-away
1011	Positive infinity
1100	Negative infinity
1101	To zero

[14:12] FP PRECISION:

000	Extended	
011	Octuple	(256-bit)
100	Quadruple	(128-bit)
101	Double	(64-bit)
110	Single	(32-bit)
111	Half	(16-bit)

The bits in the ranges [7:5] and [31:15] are not used.

5.14 CALL RETURN POINTER

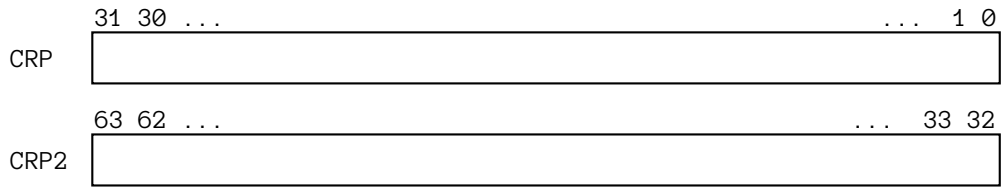


Fig. 29: *CALL RETURN POINTER*

The CALL RETURN POINTER register (Fig. 29) is automatically written by procedural program transfer instructions with the address of the instruction following the procedural program transfer instruction. Of course, it can be overwritten by inter-register transfer instruction in order to implement nested procedures. The RET instruction is usually placed at the end of the procedure and uses this pointer to return to the correct place of execution in the caller procedure.

5.15 INTERRUPT RETURN POINTER

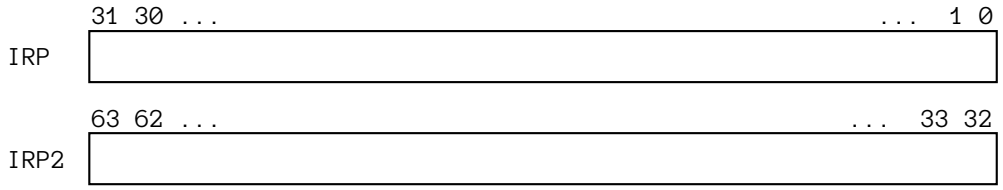


Fig. 30: *INTERRUPT RETURN POINTER*

The INTERRUPT RETURN POINTER register (Fig. 30) contains the address of the last executed instruction before interrupt handling was entered. It is automatically written on entering interrupt handling. The RETI instruction uses this pointer to return to the correct place of execution at the end of the interrupt handler.

5.16 DSP CONFIGURATION REGISTER

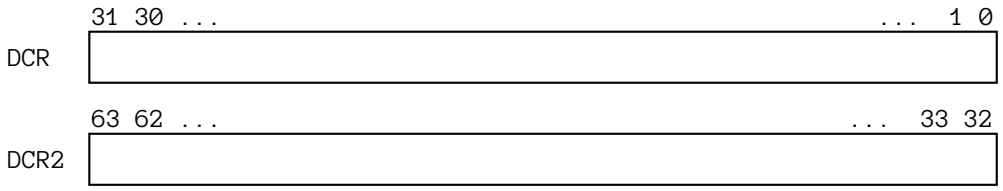


Fig. 31: *DSP CONFIGURATION REGISTER*

The DSP CONFIGURATION REGISTER (Fig. 31) is used to configure the DSP unit. The contents and the functionality of this register are .

6. INSTRUCTION SET

This Section explains in further details all instructions from the PEAKTOP ISA. The instruction dynamics is also explained, i.e., the changes that are made by executing an instruction, reading/writing special registers, conditions under which exceptions are triggered, side effects, etc. The construction of the assembly is also specified.

6.1 Detailed instruction specification

This Subsection gives detailed specification for each native instruction. The description of each instruction is organized in five paragraphs:

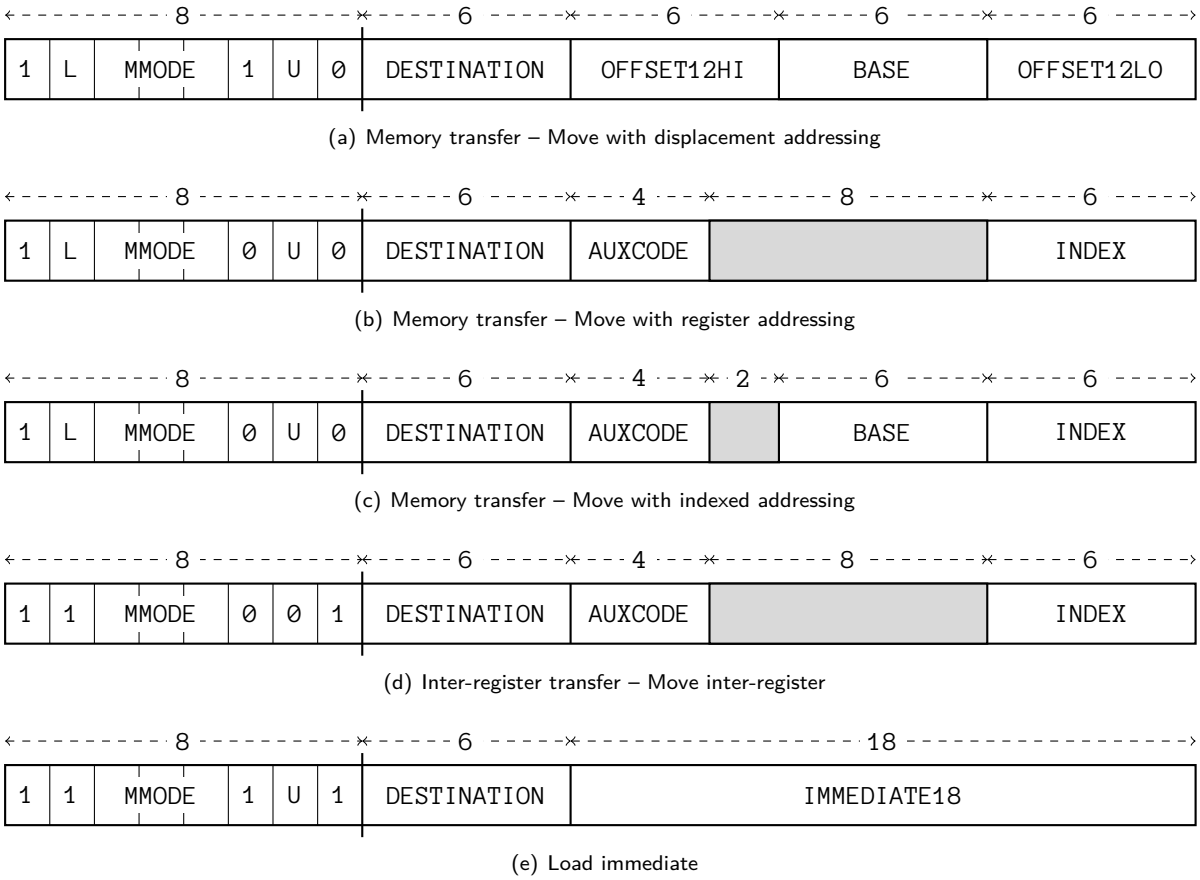
- The binary layout is specifically given for each instruction and its variants.
- The **Fields** paragraph explains the binary fields of the instruction.
- The **Execution** paragraph shows the operation of each instruction.
- The **Changes** paragraph summarizes all the changes that are done by executing the instruction.
- The **Exceptions** paragraph summarizes all exceptions that can be raised by the instruction.
- The **Examples** paragraph gives examples that illustrate the execution of the instruction.

Table 13 lists the arithmetic/logic operator symbols used to describe the functions of the instructions.

Table 13: *Arithmetic/logic operator symbols*

Symbol	Operator description
$\leftarrow$	right to left assignment
++	increment
−−	decrement
+	addition
−	subtraction
×	multiplication
÷	division
%	modulo
<<	shift left
>>	shift right
<<>	rotate left
<>>	rotate right
&	Bitwise AND
&~	Bitwise NAND
|	Bitwise OR
∧	Bitwise XOR
<==>	compare
==	is equal to
!=	is not equal to
e( )	exponent, e.g., xe$(y) \leftrightarrow x^y$

6.1.1 MOV – Move data



(a) Memory transfer – Move with displacement addressing

(b) Memory transfer – Move with register addressing

(c) Memory transfer – Move with indexed addressing

(d) Inter-register transfer – Move inter-register

(e) Load immediate

Fig. 32: *Move data (MOV) instructions*

The MOV mnemonic which comes from '*move data*' is used to denote all instructions that transfer data in a PEAKTOP system:

- memory to register (load)
- register to memory (store)
- between registers within the GPR file
- between a register in the GPR file and a register in another register file
- load of immediate values

The instructions for memory transfer specify one of the three data addressing modes (see Subsection 2.3.2). The instructions for inter-register transfer simply specify the registers (and the register files). The load immediate instruction specifies the immediate value to be loaded. All of them specify the machine mode of the transfer.

A grey field in binary instruction representation like in Fig. 32 is a don't care field.

Fields

L	For memory transfers, L=1 specifies **load** from memory and L=0 specifies **store** to memory. For inter-register transfer and load immediate, this bit is always 1.

MMODE	Specifies the integer machine mode according to Table 1.
U	For memory transfers, U=1 specifies the load-locked (L=1) and store-conditional (L=0) instructions, while U=0 specifies "normal" load and store. For load immediate, U=0 specifies signed (sign-extended) immediate, while U=1 specifies unsigned (zero-extended) immediate. (For inter-register move it is always 0.)
DESTINATION	Specifies the destination GPR for load (L=1). For store (L=0) it specifies the data source GPR which is not changed (except in atomic store-conditional (U=0) when it is written with 0 or 1).
INDEX	In memory transfers with register addressing this field specifies the GPR containing the memory address, while in indexed addressing it specifies the GPR containing the index. In inter-register transfer it specifies the data source GPR.
AUXCODE	In memory transfers with register and indexed addressing, this field specifies whether the index GPR is not changed, or whether it is pre- or post-incremented/decremented. In inter-register transfers, it specifies the source and destination register files.
BASE	In memory transfers with displacement and indexed addressing, this field specifies the GPR containing the base address.
OFFSET12HI	The six MSBs of the 12-bit OFFSET12 used as a signed displacement offset in memory transfers with displacement addressing.
OFFSET12LO	The six LSBs of the 12-bit OFFSET12 used as a signed displacement offset in memory transfers with displacement addressing.
IMMEDIATE18	An 18-bit immediate value for *Load immediate*.

Execution

Table 14 summarizes the execution of the MOV instructions.

Additionally to Table 14, for store-conditional (U=1,L=0), the destination GPR is written with 1 or 0 in MMODE width, if the load-locked/store-conditional pair of instructions was successful or not, respectively. That is,

REG[DESTINATION] ← 1, on successful load-locked/store-conditional pair, and

REG[DESTINATION] ← 0, on unsuccessful load-locked/store-conditional pair.

However, if the store-conditional returns 0, it means that writing to memory did not happen.

Changes

Destination register	All data transfer instructions except non-atomic store instructions change the specified destination register. Inter-register transfer instructions can also change the registers in other register files (not only in the GPR file), while all other data-transfer instructions change the registers only in the GPR file. If MMODE specifies shorter width than the destination register width, only the corresponding lower bits of the destination register are changed (see Subsection 2.2.2). If MMODE specifies wider width than the destination register width, the register circularity applies (see Subsection 2.2.3).
Memory	Only store instructions change the memory contents.
Index GPR	Memory transfer instructions with register or indexed addressing have the possibility to pre- or post-increment/decrement the index GPR according to the specified machine mode.

Table 14: *Execution of MOV instructions*

L	AUXCODE	Function
Move with displacement addressing		
0	N/A	MEM[BASE+OFFSET12] ← REG[DESTINATION]
1	N/A	REG[DESTINATION] ← MEM[BASE+OFFSET12]
Move with register addressing		
0	0000	MEM[INDEX] ← REG[DESTINATION]
	0001	MEM[INDEX++] ← REG[DESTINATION]
	0010	MEM[INDEX−−] ← REG[DESTINATION]
	0101	MEM[++INDEX] ← REG[DESTINATION]
	0110	MEM[−−INDEX] ← REG[DESTINATION]
1	0000	REG[DESTINATION] ← MEM[INDEX]
	0001	REG[DESTINATION] ← MEM[INDEX++]
	0010	REG[DESTINATION] ← MEM[INDEX−−]
	0101	REG[DESTINATION] ← MEM[++INDEX]
	0110	REG[DESTINATION] ← MEM[−−INDEX]
Move with indexed addressing		
0	1000	MEM[BASE+INDEX] ← REG[DESTINATION]
	1001	MEM[BASE+(INDEX++)] ← REG[DESTINATION]
	1010	MEM[BASE+(INDEX−−)] ← REG[DESTINATION]
	1101	MEM[BASE+(++INDEX)] ← REG[DESTINATION]
	1110	MEM[BASE+(−−INDEX)] ← REG[DESTINATION]
1	1000	REG[DESTINATION] ← MEM[BASE+INDEX]
	1001	REG[DESTINATION] ← MEM[BASE+(INDEX++)]
	1010	REG[DESTINATION] ← MEM[BASE+(INDEX−−)]
	1101	REG[DESTINATION] ← MEM[BASE+(++INDEX)]
	1110	REG[DESTINATION] ← MEM[BASE+(−−INDEX)]
Move inter-register		
1	0000	REG[DESTINATION] ← REG[INDEX]
	0001	REG[DESTINATION] ← SPC[INDEX]
	0010	SPC[DESTINATION] ← REG[INDEX]
	0011	REG[DESTINATION] ← DSP[INDEX]
	0100	DSP[DESTINATION] ← REG[INDEX]
	1110	REG[DESTINATION] ← FPR[INDEX]
	1111	FPR[DESTINATION] ← REG[INDEX]
Load immediate		
1	N/A	REG[DESTINATION] ← IMMEDIATE18

Exceptions

`SYSTEM INSTRUCTION` — It is raised when an inter-register transfer instruction executed in user mode tries to write a special register which is not writable in user mode (see Subsection 4.2.3 and Table 12).

`INVALID OPERATION` — It is raised when an inter-register transfer instruction executed in system mode tries to write a non-writable special register (see Subsection 4.2.7 and Table 12).

Depending on the implementation, this instruction can also raise the `UNIMPLEMENTED INSTRUCTION` and `UNIMPLEMENTED REGISTER` exceptions (see Subsections 4.2.5 and 4.2.6), as well as the `UNIMPLEMENTED GPR BANK` exception (see Subsection 4.2.4).

Examples

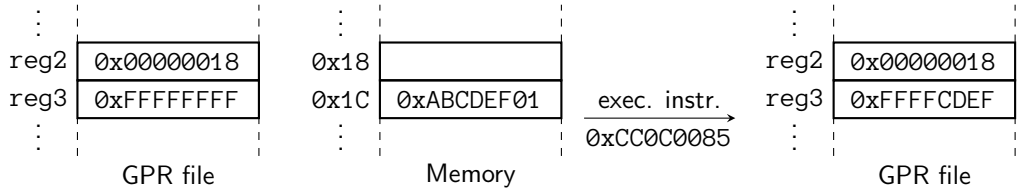
Example 1: *Load halfword from memory with displacement addressing*

Instruction in binary format: `11001100 000011 000000 000010 000101`

Instruction in hexadecimal format: 0xCC0C0085
Fields:

L	1	(load)
MMODE	001	(halfword)
U	0	(non-atomic)
DESTINATION	000011	(reg3)
OFFSET12HI	000000	(0x0)
BASE	000010	(reg2)
OFFSET12LO	000101	(0x5)

The instruction loads a halfword (16 bits) into the GPR 3 from memory at an effective address found by addition of the BASE register (here specified as GPR 2) and the OFFSET12. The concatenation of OFFSET12HI and OFFSET12LO gives OFFSET12 = 0x5. The following illustration shows an example state of a 32-bit wide GPR file and memory with little-endian ordering before and after execution of the instruction (0xCC0C0085).



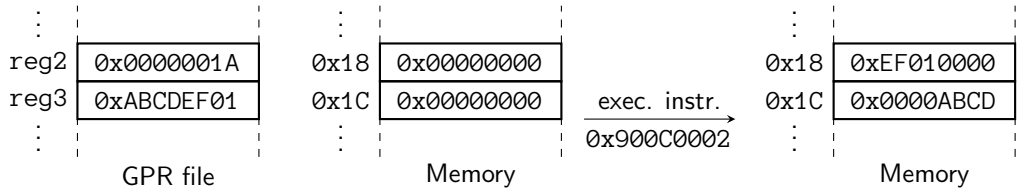
Thus, the effective address is 0x18 + 0x5 = 0x1D. At address 0x1C the memory content is 0xABCDEF01, so the 16 bits 0xCDEF starting at 0x1D will be read into the lower half of GPR 3, while its upper part remains unchanged, i.e., 0xFFFF.

Example 2: *Store word in memory with register addressing*

Instruction in binary format: 10010000 000011 0000 00000000 000010
Instruction in hexadecimal format: 0x900C0002
Fields:

L	0	(store)
MMODE	010	(word)
U	0	(non-atomic)
DESTINATION	000011	(reg3)
AUXCODE	0000	(no base, no change of the INDEX GPR)
INDEX	000010	(reg2)

The instruction stores a word (32 bits) residing into GPR 3 (specified by the DESTINATION field) into memory at an effective address contained in the GPR 2 (specified by the INDEX field). The following illustration shows an example state of a 32-bit wide GPR file and memory with little-endian ordering before and after execution of the instruction (0x900C0002).



Thus, all 32 bits of GPR 3 are written at address 0x1A taken from GPR 2. Of course, the GPR file is not changed by a non-atomic store instruction.

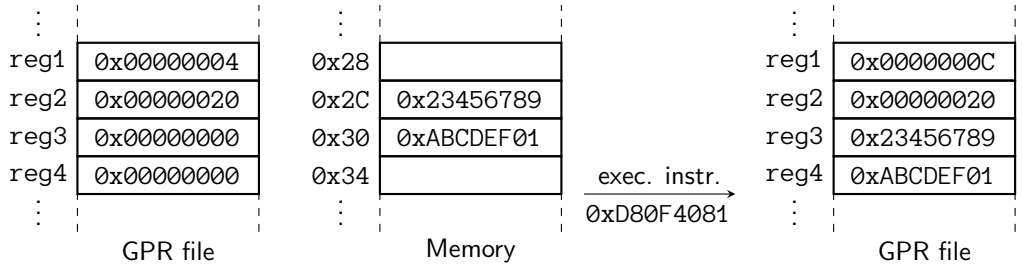
Example 3: *Load doubleword from memory with indexed addressing*

Instruction in binary format: 11011000 000011 1101 00 000010 000001
Instruction in hexadecimal format: 0xD80F4081

Fields:

L	1	(load)
MMODE	011	(doubleword)
U	0	(non-atomic)
DESTINATION	000011	(reg3)
AUXCODE	1101	(use base, pre-increment the INDEX GPR)
BASE	000010	(reg2)
INDEX	000001	(reg1)

The instruction loads a doubleword (64 bits) into the GPR 3 from memory at an effective address found by addition of the BASE register (here specified as GPR 2) and the INDEX register GPR 1 which is pre-incremented before forming the address. The following illustration shows an example state of a 32-bit wide GPR file and memory with little-endian ordering before and after execution of the instruction (0xD80F4081).



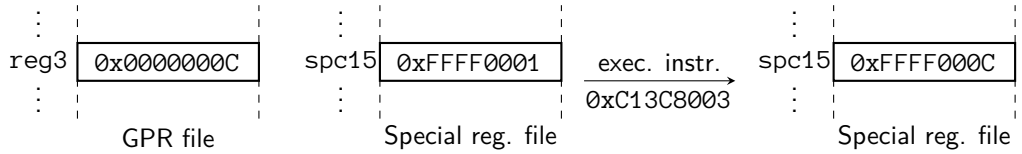
Thus, the INDEX found in GPR 1 (0x4) is incremented by 8 (since a doubleword has 8 bytes) and its value (0xC) is updated and added to the BASE GPR 2 (0x20) giving the effective address 0x2C. Since the GPR width is 32 bits, both GPR 3 and GPR 4 are loaded due to the property of circularity (see Subsection 2.2.3).

Example 4: *Inter-register transfer from a GPR to a special register*

Instruction in binary format: 11000001 001111 0010 00000000 000011
Instruction in hexadecimal format: 0xC13C8003
Fields:

MMODE	000	(byte)
DESTINATION	001111	(SYSTEM CONTROL REGISTER)
AUXCODE	0010	(copy GPR to special register)
INDEX	000001	(reg3)

The instruction copies a byte from the GPR 3 to the SYSTEM CONTROL REGISTER. The following illustration shows an example state of a 32-bit wide GPR and special register file before and after execution of the instruction (0xC13C8003).



Thus, by executing this instruction, the user mode is entered, and both the exceptions and the interrupt line are enabled. Only the lower eight bits are changed in the SYSTEM CONTROL REGISTER, as the machine mode is byte.

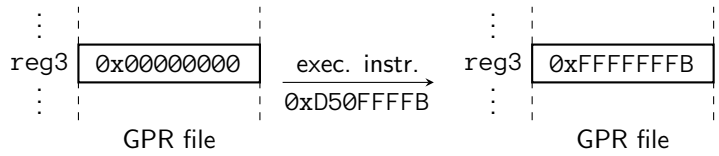
Example 5: *Load immediate*

Instruction in binary format: 11010101 000011 1111111111111011
Instruction in hexadecimal format: 0xD50FFFFB

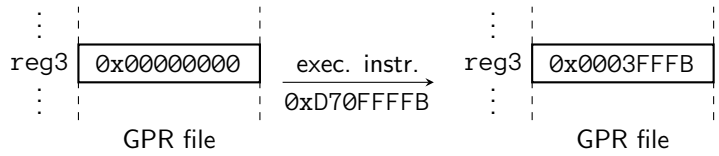
Fields:

MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
IMMEDIATE18	111111111111111011	(0x3FFFB)

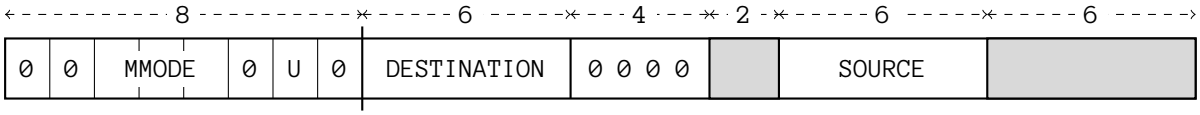
The instruction loads an immediate word value (32-bit) into GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0xD50FFFFB).



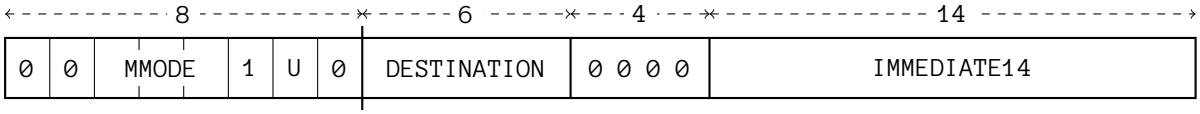
Thus, the 18-bit wide immediate value is sign-extended to 32-bits. If now only the U bit is changed to 1, the instruction (0xD70FFFFB) treats the immediate value as unsigned and its execution will do the following.



6.1.2 ADD – Add



(a) Add register



(b) Add immediate

Fig. 33: *Add (ADD) instructions*

The ADD instruction specifies *integer addition* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the addition. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **signed** operation (including sign-extended immediate for *Add immediate*). 1: **unsigned** operation (including zero-extended immediate for *Add immediate*). See Subsection 3.2.1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Add register*.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Add immediate*.

Execution

Table 15 summarizes the execution of ADD instructions.

Table 15: *Execution of ADD instructions*

Add register
REG[DESTINATION] ← REG[DESTINATION] + REG[SOURCE]
Add immediate
REG[DESTINATION] ← REG[DESTINATION] + IMMEDIATE14

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

OVERFLOW	It is raised when the result of the operation cannot be represented in the specified machine mode (see Subsection 4.2.9).

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

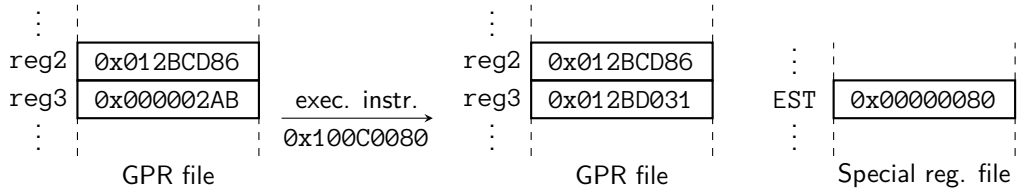
Examples

Example 6: *Add register*

Instruction in binary format: 00010000 000011 0000 00 000010 000000
Instruction in hexadecimal format: 0x100C0080
Fields:

MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The instruction adds the value of GPR 2 to GPR 3 and writes the computed result back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100C0080). It also shows the state of the special register file after instruction execution.



Thus, the GPR 3 is overwritten with the computed result in its full 32-bit width since the operation is in word machine mode.

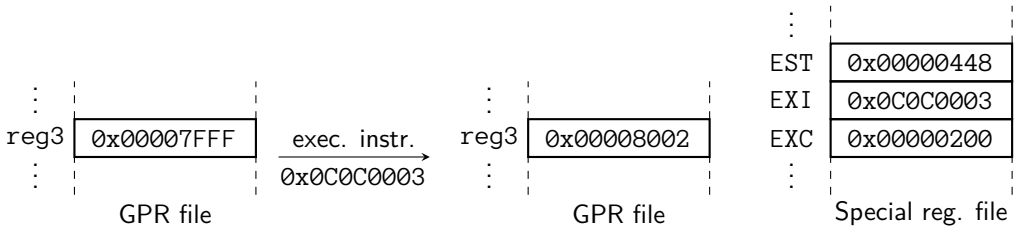
In the EXECUTION STATUS register (EST), only the LESS THAN flag is set since (before instruction execution) the first operand in GPR 3 specified by DESTINATION is lesser than the second operand in GPR 2 specified by SOURCE (see Subsection 5.2).

Example 7: *Add immediate*

Instruction in binary format: 00001100 000011 0000 00000000000011
Instruction in hexadecimal format: 0x0C0C0003
Fields:

MMODE	001	(halfword)
U	0	(signed)
DESTINATION	000011	(reg3)
IMMEDIATE14	00000000000011	(0x3)

The instruction adds the 14-bit wide, sign-extended IMMEDIATE14 value to the value in GPR 3, and writes the computed result back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x0C0C0003). It also shows the state of the special register file after instruction execution. It is assumed that the OVERFLOW exception is impotent.

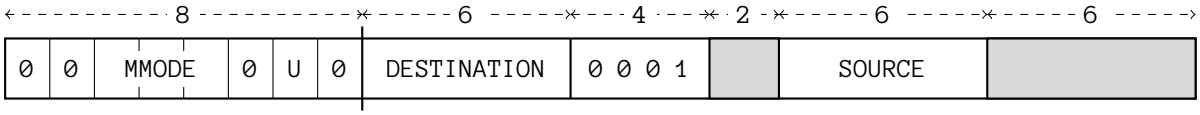


Thus, the lower 16 bits of GPR 3 are overwritten with the computed result since the operation is in halfword machine mode.

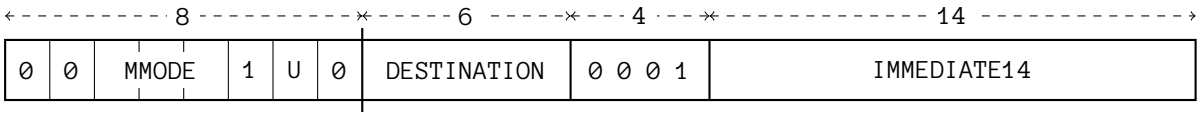
This instruction raises the OVERFLOW exception since the computed result overflows the maximal representable positive integer in halfword mode and becomes a negative number. The EXCEPTION INSTRUCTION register (EXI) is therefore written with the instruction code, and the OVERFLOW bit in the EXCEPTION REGISTER (EXC) is set (it is assumed that before executing the instruction the EXC register was zero). However, since it is assumed that the OVERFLOW exception is impotent, the result is written back to the DESTINATION GPR 3. If it was potent, the GPR 3 and the EXECUTION STATUS register (EST) would not be overwritten and the exception handling would have been started.

In the EXECUTION STATUS register (EST), the OVERFLOW, GREATER THAN and SIGN flags are set (see Subsection 5.2). The GREATER THAN flag is set since (before instruction execution) the first operand in GPR 3 specified by DESTINATION is greater than the second operand specified by IMMEDIATE14. On the other side, the SIGN flag is set since the result (obtained after instruction execution) is negative in halfword mode.

6.1.3 SUB – Subtract



(a) Subtract register



(b) Subtract immediate

Fig. 34: *Subtract (SUB) instructions*

The SUB instruction specifies *integer subtraction* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the subtraction. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **signed** operation (including sign-extended immediate for *Subtract immediate*). 1: **unsigned** operation (including zero-extended immediate for *Subtract immediate*). See Subsection 3.2.1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Subtract register*.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Subtract immediate*.

Execution

Table 16 summarizes the execution of SUB instructions.

Table 16: *Execution of SUB instructions*

Subtract register
REG[DESTINATION] ← REG[DESTINATION] – REG[SOURCE]
Subtract immediate
REG[DESTINATION] ← REG[DESTINATION] – IMMEDIATE14

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

OVERFLOW	It is raised when the result of the operation cannot be represented in the specified machine mode (see Subsection 4.2.9).

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

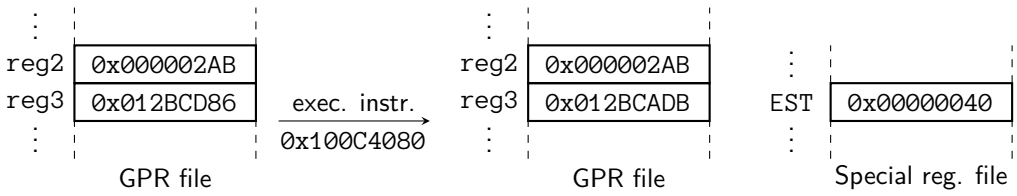
Examples

Example 8: *Subtract register*

Instruction in binary format: 00010000 000011 0001 00 000010 000000
Instruction in hexadecimal format: 0x100C4080
Fields:

MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The instruction subtracts the value in GPR 2 from GPR 3 and writes the computed result back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100C4080). It also shows the state of the special register file after instruction execution.



Thus, the GPR 3 is overwritten with the computed result in its full 32-bit width since the operation is in word machine mode.

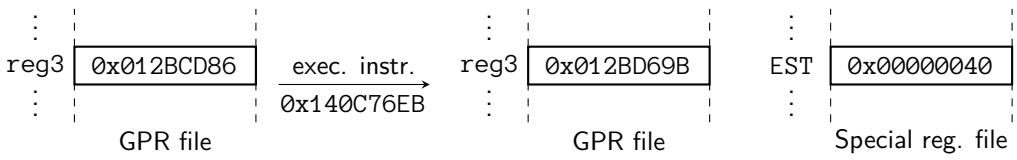
In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

Example 9: *Subtract immediate*

Instruction in binary format: 00010100 000011 0001 11011011101011
Instruction in hexadecimal format: 0x140C76EB
Fields:

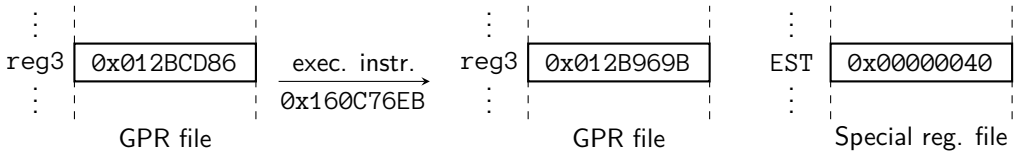
MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
IMMEDIATE14	11011011101011	(0x36EB)

The instruction subtracts the 14-bit wide, sign-extended IMMEDIATE14 value from the value in GPR 3, and writes the computed result back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x140C76EB). It also shows the state of the special register file after instruction execution.



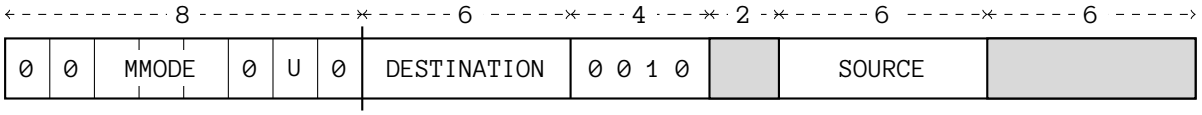
Thus, the GPR 3 is overwritten with the computed result in its full 32-bit width since the operation is in word machine mode.

Since the sign-extended immediate value is actually a negative number (decimal: -2325), this is actually an addition. If now only the U bit is changed to 1, the instruction (0x160C76EB) denotes unsigned subtraction and treats the immediate value as unsigned, and its execution will do the following.

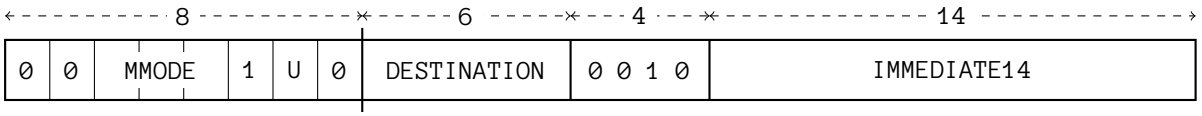


In both cases, only the GREATER THAN flag is set in the EXECUTION STATUS register (EST) (see Subsection 5.2).

6.1.4 MUL – Multiply



(a) Multiply register



(b) Multiply immediate

Fig. 35: *Multiply (MUL) instructions*

The MUL instruction specifies *integer multiplication* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the multiplication. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **signed** operation (including sign-extended immediate for *Multiply immediate*). 1: **unsigned** operation (including zero-extended immediate for *Multiply immediate*). See Subsection 3.2.1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Multiply register*.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Multiply immediate*.

Execution

Table 17 summarizes the execution of MUL instructions.

Table 17: *Execution of MUL instructions*

Multiply register
REG[DESTINATION] ← REG[DESTINATION] × REG[SOURCE]
Multiply immediate
REG[DESTINATION] ← REG[DESTINATION] × IMMEDIATE14

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. However, the MUL instruction always returns a result which is twice the width of the input operands specified by MMODE. Thus, depending on the machine mode and the GPR width, subsequent GPRs may be written according to the property of circularity (see Subsection 2.2.3) in little-endian ordering. For example, if both

the GPR width and the MMODE is 32 bits, then the result is 64-bit wide which will be written in two subsequent GPRs, i.e., the lower part in the GPR specified by the DESTINATION field, and the upper part in the subsequent GPR. On the other side, if the GPR width is 32 bits and MMODE is 16 bits, the result is 32-bit wide, and will be written in a single register specified by DESTINATION.

EXECUTION STATUS — Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

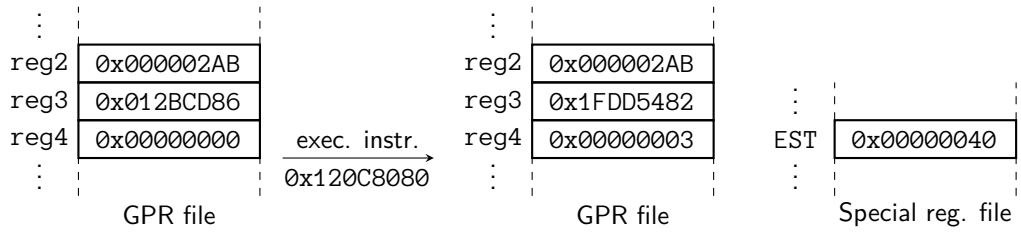
Examples

Example 10: *Multiply register (word machine mode)*

Instruction in binary format: 00010010 000011 0010 00 000010 000000
Instruction in hexadecimal format: 0x120C8080
Fields:

MMODE	010	(word)
U	1	(unsigned)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The instruction multiplies the value in GPR 2 to the value in GPR 3 and writes the computed result back in GPR 3 and in GPR 4. Since U = 0, the operation is unsigned, and the operands in GPR 2 and GPR 3 are considered unsigned. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x120C8080). It also shows the state of the special register file after instruction execution.



The computed result 0x31FDD5482 is 64-bit wide since a multiplication of two word-sized values gives a doubleword result which is written back into GPR 3 and GPR 4, of which GPR 4 contains the higher part in significance. In other words, the result is a concatenation of GPR 4 and GPR 3.

In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

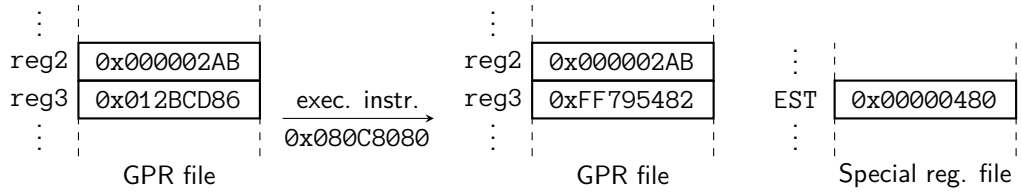
Example 11: *Multiply register (halfword machine mode)*

Instruction in binary format: 00001000 000011 0010 000010 000000
Instruction in hexadecimal format: 0x080C8080
Fields:

MMODE	001	(halfword)
U	0	(signed)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The instruction multiplies the 16-bit subvalue in GPR 2 to the 16-bit subvalue in GPR 3 and writes the computed 32-bit result back in GPR 3. The following illustration shows an example

state of a 32-bit wide GPR file before and after execution of the instruction (0x080C8080). It also shows the state of the special register file after instruction execution.



The 16-bit input operand value in GPR 3 (0xCD86) is negative and the input operand value in GPR 2 is positive (0x02AB), which implies a negative 32-bit wide result (0xFF795482) which is written back into GPR 3.

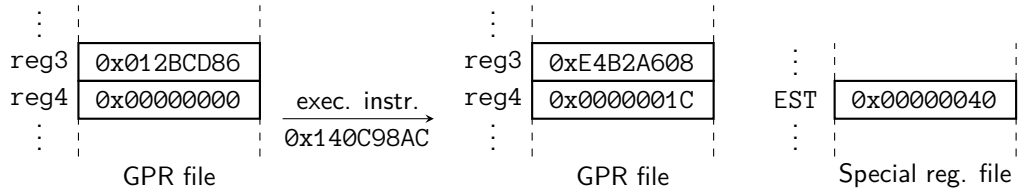
In the EXECUTION STATUS register (EST), the LESS THAN and SIGN flags are set (see Subsection 5.2).

Example 12: *Multiply immediate*

Instruction in binary format: 00010100 000011 0010 01100010101100
Instruction in hexadecimal format: 0x140C98AC
Fields:

MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
IMMEDIATE14	01100010101100	(0x18AC)

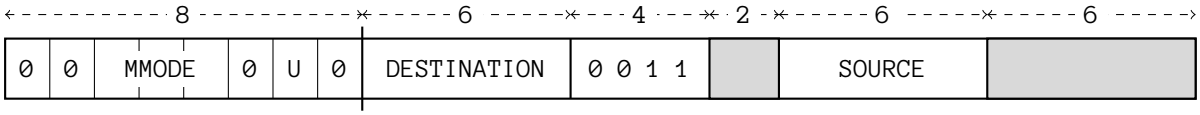
The instruction multiplies the 14-bit wide, sign-extended IMMEDIATE14 value to the value in GPR 3, and writes the computed result back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x140C98AC). It also shows the state of the special register file after instruction execution.



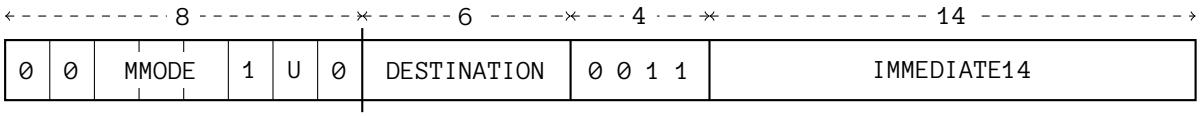
The computed result 0x1CE4B2A608 is 64-bit wide since a multiplication of two word-sized values gives a doubleword result which is written back into GPR 3 and GPR 4, of which GPR 4 contains the higher part in significance. In other words, the result is a concatenation of GPR 4 and GPR 3.

In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

6.1.5 DIV – Divide



(a) Divide register



(b) Divide immediate

Fig. 36: *Divide (DIV) instructions*

The DIV instruction specifies *integer division* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the division. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **signed** operation (including sign-extended immediate for *Divide immediate*). 1: **unsigned** operation (including zero-extended immediate for *Divide immediate*). See Subsection 3.2.1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Divide register.*
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Divide immediate.*

Execution

Table 18 summarizes the execution of DIV instructions.

Table 18: *Execution of DIV instructions*

Divide register
REG[DESTINATION] ← REG[DESTINATION] ÷ REG[SOURCE] REG[subseq(DESTINATION)] ← REG[DESTINATION] % REG[SOURCE]
Divide immediate
REG[DESTINATION] ← REG[DESTINATION] ÷ IMMEDIATE14 REG[subseq(DESTINATION)] ← REG[DESTINATION] % IMMEDIATE14

The subseq() function gives the subsequent GPR number. For example, for a 32-bit wide GPR file and a 32-bit machine mode (or lower), subseq(2) = 3. On the other hand, for a 32-bit wide GPR file and a 64-bit machine mode subseq(2) = 4, etc.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. The DIV instruction returns both the quotient and the remainder of the integer division in two subsequent registers according to the circularity property Subsection 2.2.3. In Table 18 the function subseq( ) gives the number of the subsequent register that contains the remainder. However, if the GPR width is lower than the machine mode of the operation, then accordingly more registers are used. For example, if the GPR width is 16 bits and the machine mode is 32 bits, four registers are used: the GPR specified by the DESTINATION field contains the lower part of the quotient. The first subsequent register contains the higher part of the quotient, while the third and fourth subsequent registers contain the lower and the upper part of the remainder, respectively. In this case, the function subseq( ) in Table 18 gives the number of the GPR that contains the lower part of the remainder.
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

DIVISION BY ZERO	It is raised when the divisor is zero.
OVERFLOW	It is raised on a signed division in which the dividend is the minimal representable number in the specified machine mode and the divisor is –1 .

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

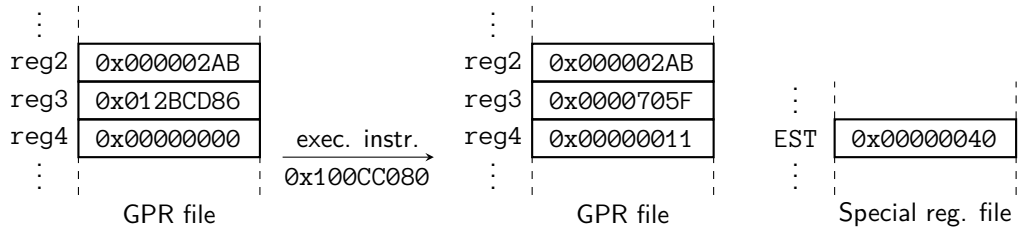
Examples

Example 13: *Divide register (word machine mode)*

Instruction in binary format: 00010000 000011 0011 00 000010 000000
Instruction in hexadecimal format: 0x100CC080
Fields:

MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The divisor in GPR 2 divides the dividend in GPR 3. The computed quotient is written back in GPR 3 while the computed remainder is written back in GPR 4. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100CC080). It also shows the state of the special register file after instruction execution.



Thus, the computed quotient 0x0000705F and remainder 0x00000011 are both 32-bit wide, and occupy GPR 3 and GPR 4, respectively.

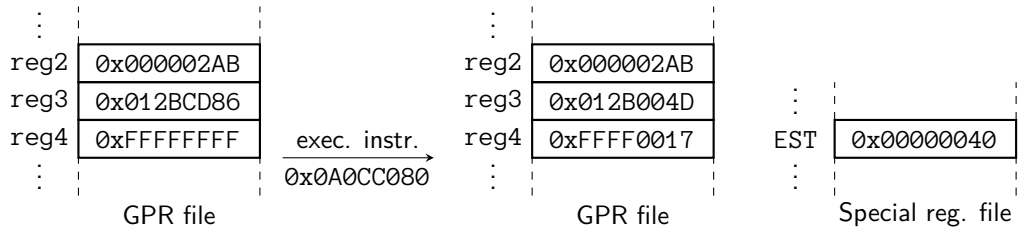
In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

Example 14: *Divide register (halfword machine mode)*

Instruction in binary format: `00001010 000011 0011 00 000010 000000`
Instruction in hexadecimal format: `0x0A0CC080`
Fields:

MMODE	001	(halfword)
U	1	(unsigned)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The 16-bit subvalue in GPR 2 is the divisor which divides the 16-bit subvalue dividend in GPR 3. The computed 16-bit quotient is written back in GPR 3 while the computed 16-bit remainder is written back in GPR 4. The division is unsigned and both of the 16-bit input operands are considered unsigned. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (`0x0A0CC080`). It also shows the state of the special register file after instruction execution.



Thus, the 16-bit input operand value in GPR 3 (`0xCD86`) is considered positive integer. After instruction execution, only the lower 16 bits of GPR 3 and GPR 4 are overwritten with the 16-bit quotient (`0x4D`) and the 16-bit remainder (`0x17`), respectively.

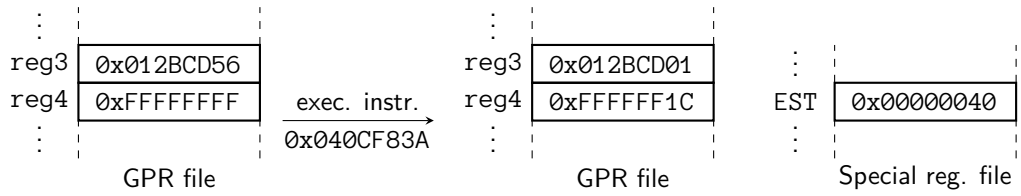
In the `EXECUTION STATUS` register (`EST`), only the `GREATER THAN` flag is set (see Subsection 5.2).

Example 15: *Divide immediate*

Instruction in binary format: `00000100 000011 0011 11100000111010`
Instruction in hexadecimal format: `0x040CF83A`
Fields:

MMODE	000	(byte)
U	0	(signed)
DESTINATION	000011	(reg3)
IMMEDIATE14	11100000111010	(0x383A)

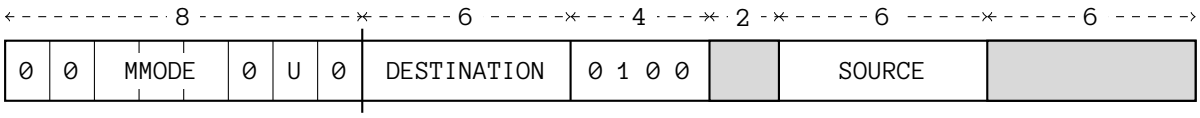
The 8-bit subvalue of the 14-bit `IMMEDIATE14` is the divisor which divides the 8-bit subvalue dividend in GPR 3. The computed 8-bit quotient is written back in GPR 3 while the computed 8-bit remainder is written back in GPR 4. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (`0x040CF83A`). It also shows the state of the special register file after instruction execution.



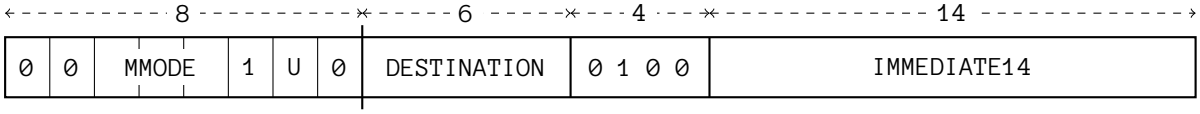
Thus, the 8-bit divisor given by `IMMEDIATE14` (`0x3A`) divides the 8-bit subvalue dividend in GPR 3 (`0x56`). After instruction execution, only the lower 8 bits of GPR 3 and GPR 4 are overwritten with the 8-bit quotient (`0x01`) and the 8-bit remainder (`0x1C`), respectively.

In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

6.1.6 SL – Shift left



(a) Shift left register



(b) Shift left immediate

Fig. 37: *Shift left (SL) instructions*

The SL instruction specifies a *left shift* of the first operand for a number of places specified by the second operand. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the shift. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **Arithmetic left shift** – triggers the OVERFLOW exception if the MSB (according to MMODE) of the destination GPR is changed. 1: **Logic left shift** – does not trigger exceptions.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Shift left register*. That is the number of bits to be shifted left in the GPR specified by DESTINATION.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Shift left immediate*. That is the number of bits to be shifted left in the GPR specified by DESTINATION.

Execution

Table 19 summarizes the execution of SL instructions.

Table 19: *Execution of SL instructions*

Shift left register
REG[DESTINATION] ← REG[DESTINATION] << REG[SOURCE]
Shift left immediate
REG[DESTINATION] ← REG[DESTINATION] << IMMEDIATE14

The SL instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the

corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).

EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

OVERFLOW	It is raised by arithmetic left shift when the MSB of the destination GPR changes its state. The MSB of the GPR is determined according to the specified machine mode.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

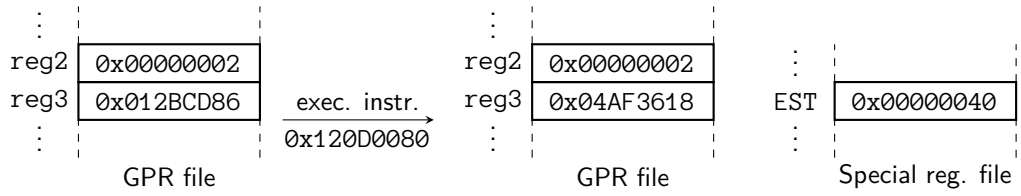
Examples

Example 16: *Shift left register*

Instruction in binary format: 00010010 000011 0100 00 000010 00000000
Instruction in hexadecimal format: 0x120D0080
Fields:

MMODE	010	(word)
U	1	(logic shift)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The value in GPR 3 is shifted left (logically) for a number of bit places given by GPR 2. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x120D0080). It also shows the state of the special register file after instruction execution.



If the number of bit shifts (here given by GPR 2) is greater or equal than the GPR width (or the width of the machine mode), the result will be zero.

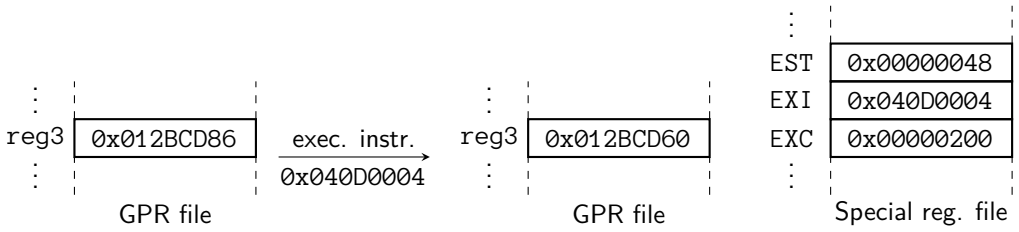
In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

Example 17: *Shift left immediate*

Instruction in binary format: 00000100 000011 0100 00000000000100
Instruction in hexadecimal format: 0x040D0004
Fields:

MMODE	000	(byte)
U	0	(arithmetic shift)
DESTINATION	000011	(reg3)
IMMEDIATE14	00000000000100	(0x4)

The value in GPR 3 is shifted left (arithmetically) for a number of bit places given by IMMEDIATE14. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x040D0004). It also shows the state of the special register file after instruction execution.

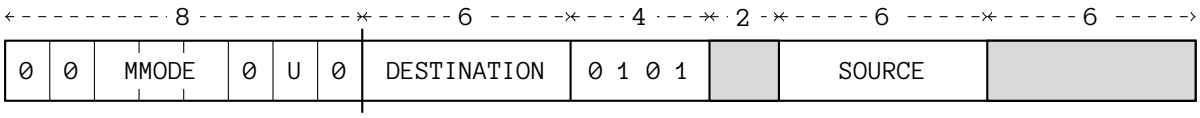


Thus, only the lowest byte of GPR 3 is shifted for 4 places.

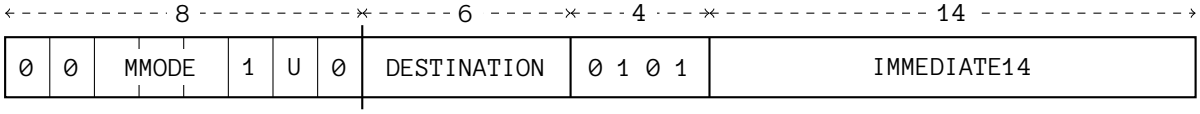
This instruction raises the OVERFLOW exception since the MSB (in byte width) of GPR 3 is changed from 1 to 0 after instruction execution. The EXCEPTION INSTRUCTION register (EXI) is therefore written with the instruction code, and the OVERFLOW bit in the EXCEPTION REGISTER (EXC) is set (it is assumed that before executing the instruction the EXC register was zero). However, since it is assumed that the OVERFLOW exception is impotent, the result is written back to the DESTINATION GPR 3. If it was potent, the GPR 3 and the EXECUTION STATUS register (EST) would not be overwritten and the exception handling would have been started.

In the EXECUTION STATUS register (EST), the OVERFLOW and GREATER THAN flags are set (see Subsection 5.2).

6.1.7 SR – Shift right



(a) Shift right register



(b) Shift right immediate

Fig. 38: *Shift right (SR) instructions*

The SR instruction specifies a *right shift* of the first operand for a number of places specified by the second operand. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the shift. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **Arithmetic right shift** – pulls the MSB (according to MMODE) of the destination GPR. 1: **Logic right shift** – pulls a zero.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Shift right register*. That is the number of bits to be shifted right in the GPR specified by DESTINATION.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Shift right immediate*. That is the number of bits to be shifted right in the GPR specified by DESTINATION.

Execution

Table 20 summarizes the execution of SR instructions.

Table 20: *Execution of SR instructions*

Shift right register
REG[DESTINATION] ← REG[DESTINATION] >> REG[SOURCE]
Shift right immediate
REG[DESTINATION] ← REG[DESTINATION] >> IMMEDIATE14

The SR instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the

corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).

EXECUTION STATUS — Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

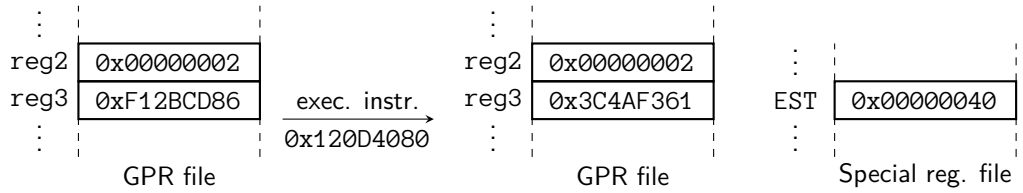
Examples

Example 18: *Shift right register*

Instruction in binary format: 00010010 000011 0101 00 000010 000000
Instruction in hexadecimal format: 0x120D4080
Fields:

MMODE	010	(word)
U	1	(logic shift)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The value in GPR 3 is shifted right (logically) for a number of bit places given by GPR 2. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x120D4080). It also shows the state of the special register file after instruction execution.



If the number of bit shifts (here given by GPR 2) is greater or equal than the GPR width (or the width of the machine mode), the result will be zero.

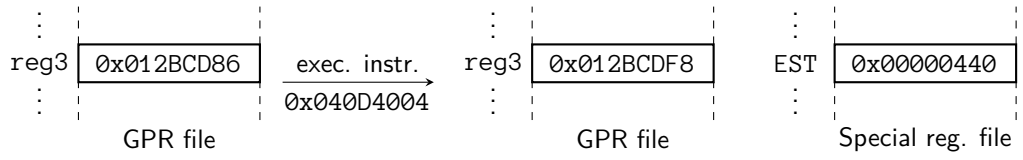
In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

Example 19: *Shift right immediate*

Instruction in binary format: 00000100 000011 0101 00000000000100
Instruction in hexadecimal format: 0x040D4004
Fields:

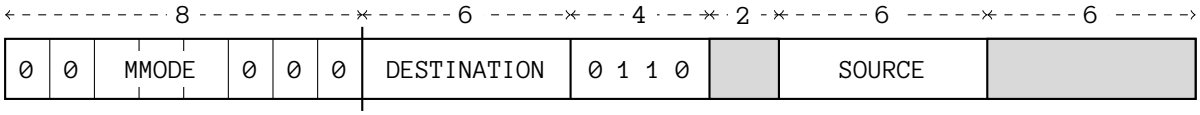
MMODE	000	(byte)
U	0	(arithmetic shift)
DESTINATION	000011	(reg3)
IMMEDIATE14	00000000000100	(0x4)

The value in GPR 3 is shifted right (arithmetically) for a number of bit places given by IMMEDIATE14. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x040D4004). It also shows the state of the special register file after instruction execution.

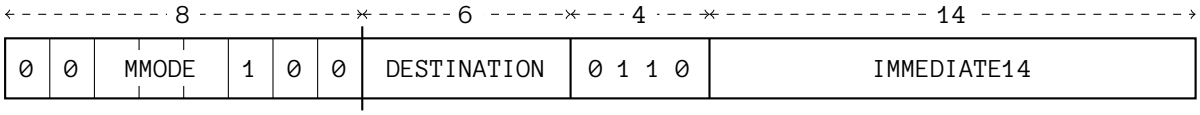


Thus, only the lowest byte of GPR 3 is shifted for 4 places. The MSB in byte width of GPR 3 is preserved and the result remains negative (in byte width).

6.1.8 RL – Rotate left



(a) Rotate left register



(b) Rotate left immediate

Fig. 39: *Rotate left (RL) instructions*

The RL instruction specifies a *left rotation* of the first operand for a number of places specified by the second operand. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the rotation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Rotate left register*. That is the number of bits to be rotated left in the GPR specified by DESTINATION.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Rotate left immediate*. That is the number of bits to be rotated left in the GPR specified by DESTINATION.

Execution

Table 21 summarizes the execution of RL instructions.

Table 21: *Execution of RL instructions*

Rotate left register
REG[DESTINATION] ← REG[DESTINATION] <<> REG[SOURCE]
Rotate left immediate
REG[DESTINATION] ← REG[DESTINATION] <<> IMMEDIATE14

The RL instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).

EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

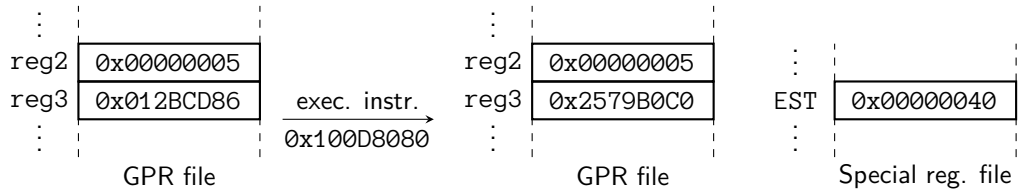
Examples

Example 20: *Rotate left register*

Instruction in binary format: 00010000 000011 0110 00 000010 000000
Instruction in hexadecimal format: 0x100D8080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The value in GPR 3 is rotated left for a number of bit places given by GPR 2. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100D8080). It also shows the state of the special register file after instruction execution.



Thus, the bits in GPR 3 that go out at the left side are inserted back on the right side.

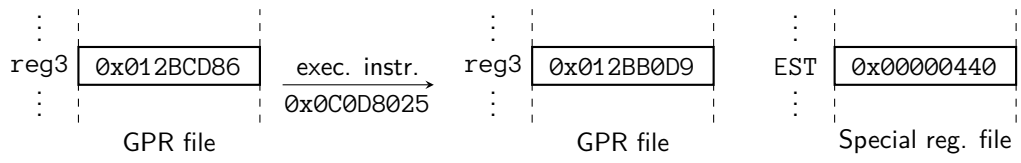
In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

Example 21: *Rotate left immediate*

Instruction in binary format: 00001100 000011 0110 00000000100101
Instruction in hexadecimal format: 0x0C0D8025
Fields:

MMODE	001	(halfword)
DESTINATION	000011	(reg3)
IMMEDIATE14	00000000100101	(0x25)

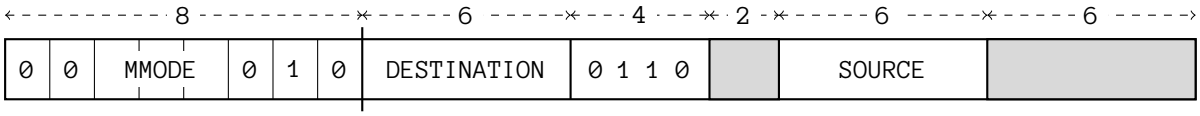
The value in GPR 3 is rotated left for a number of bit places given by IMMEDIATE14. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x0C0D8025). It also shows the state of the special register file after instruction execution.



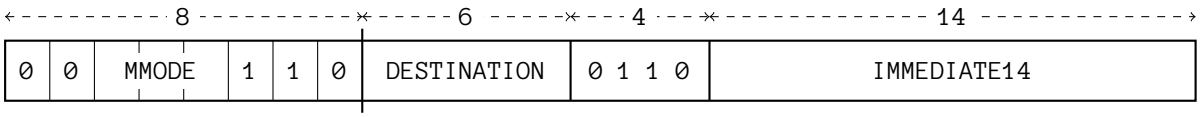
Thus, the bits in GPR 3 that go out at the left side are inserted back on the right side. Although the number of bit rotations (0x25, or, in decimal format 37) is greater than the width of the machine mode (16 bits), GPR 3 is rotated anyway. The result will be the same as if the number of bit rotations is 5, i.e., 37 % 16.

In the EXECUTION STATUS register (EST), the GREATER THAN and SIGN flags are set (see Subsection 5.2).

6.1.9 RR – Rotate right



(a) Rotate right register



(b) Rotate right immediate

Fig. 40: *Rotate right (RR) instructions*

The RR instruction specifies a *right rotation* of the first operand for a number of places specified by the second operand. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the rotation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Rotate right register*. That is the number of bits to be rotated right in the GPR specified by DESTINATION.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Rotate right immediate*. That is the number of bits to be rotated right in the GPR specified by DESTINATION.

Execution

Table 22 summarizes the execution of RR instructions.

Table 22: *Execution of RR instructions*

Rotate right register
REG[DESTINATION] ← REG[DESTINATION] <>> REG[SOURCE]
Rotate right immediate
REG[DESTINATION] ← REG[DESTINATION] <>> IMMEDIATE14

The RR instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see

Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).

EXECUTION STATUS — Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

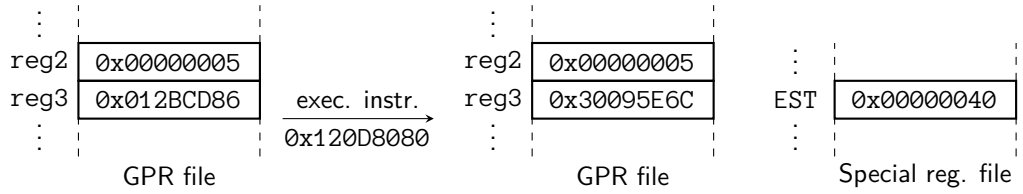
Examples

Example 22: *Rotate right register*

Instruction in binary format: 00010010 000011 0110 00 000010 000000
Instruction in hexadecimal format: 0x120D8080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The value in GPR 3 is rotated right for a number of bit places given by GPR 2. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x120D8080). It also shows the state of the special register file after instruction execution.



Thus, the bits in GPR 3 that go out at the right side are inserted back on the left side.

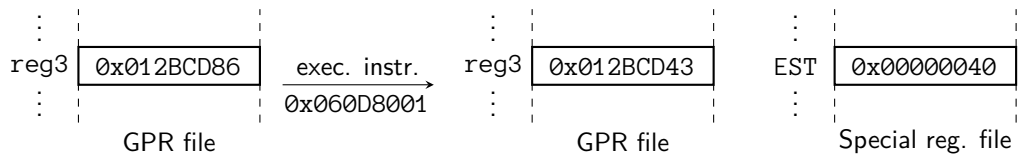
In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

Example 23: *Rotate right immediate*

Instruction in binary format: 00000110 000011 0110 00000000000001
Instruction in hexadecimal format: 0x060D8001
Fields:

MMODE	000	(byte)
DESTINATION	000011	(reg3)
IMMEDIATE14	00000000000001	(0x1)

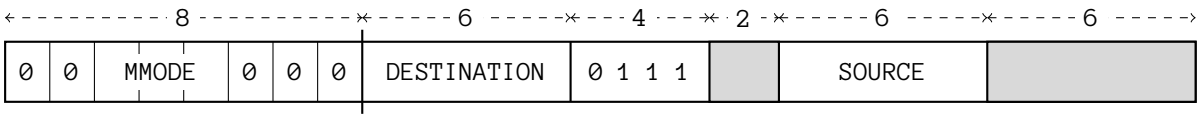
The value in GPR 3 is rotated right for a number of bit places given by IMMEDIATE14. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x060D8001). It also shows the state of the special register file after instruction execution.



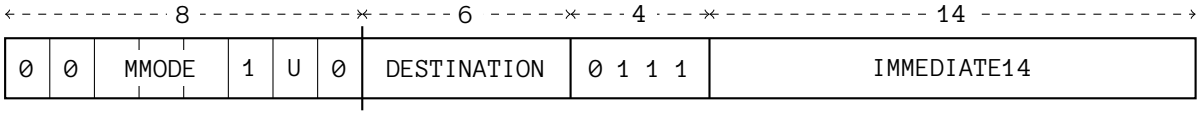
Thus, the bit in GPR 3 that goes out at the right side is inserted back on the left side.

In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

6.1.10 AND – AND bitwise



(a) AND bitwise register



(b) AND bitwise immediate

Fig. 41: *AND bitwise (AND) instructions*

The AND instruction specifies *logic 'AND' operation* between two operands. The operation is bitwise, i.e., between the corresponding bits of the operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the 'AND' operation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	Not used by *AND bitwise register* (it should be always U=0). For *AND bitwise immediate*, U=0 specifies **sign-extended** immediate, while U=1 specifies **zero-extended** immediate. However, although zero- or sign-extended, the immediate is still treated as unsigned.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *AND bitwise register.*
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *AND bitwise immediate*.

Execution

Table 23 summarizes the execution of AND instructions.

Table 23: *Execution of AND instructions*

AND bitwise register
REG[DESTINATION] ← REG[DESTINATION] & REG[SOURCE]
AND bitwise immediate
REG[DESTINATION] ← REG[DESTINATION] & IMMEDIATE14

The AND instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

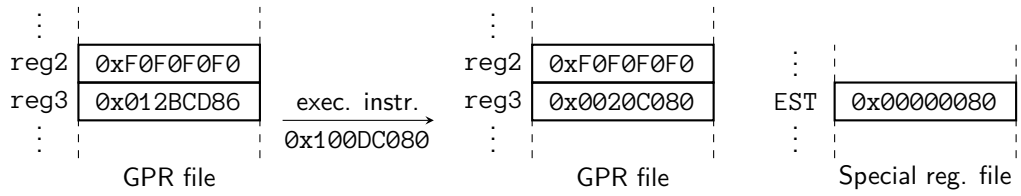
Examples

Example 24: *AND bitwise register*

Instruction in binary format: 00010000 000011 0111 00 000010 000000
Instruction in hexadecimal format: 0x100DC080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The value in GPR 3 is AND-ed bitwise with the value in GPR 2. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100DC080). It also shows the state of the special register file after instruction execution.



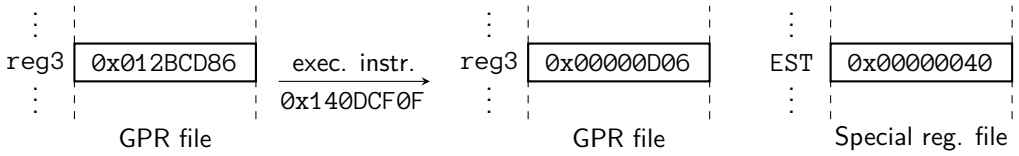
In the EXECUTION STATUS register (EST), only the LESS THAN flag is set (see Subsection 5.2). This is because logic operations always treat the operands as unsigned. The U bit can be used only for the logic operations with immediate values (see Examples 25, 27, 29 and 31) in which it signifies whether the immediate value is sign- or zero-extended, but even the sign-extended immediate values are treated as unsigned.

Example 25: *AND bitwise immediate*

Instruction in binary format: 00010100 000011 0111 00111100001111
Instruction in hexadecimal format: 0x140DCF0F
Fields:

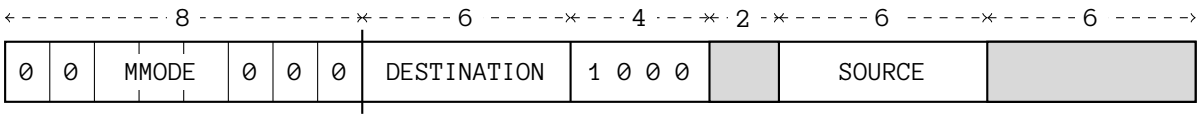
MMODE	010	(word)
U	0	(sign-extended immediate)
DESTINATION	000011	(reg3)
IMMEDIATE14	00111100001111	(0xF0F)

The value in GPR 3 is AND-ed bitwise with the sign-extended value given by IMMEDIATE14. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x140DCF0F). It also shows the state of the special register file after instruction execution.

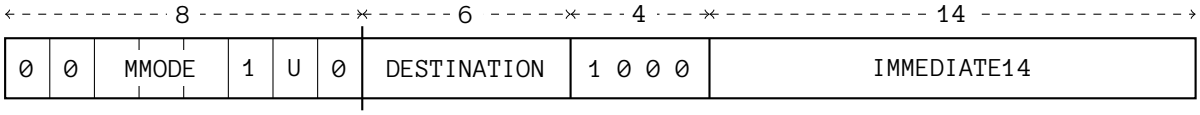


In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

6.1.11 NAND – Negated AND bitwise



(a) Negated AND bitwise register



(b) Negated AND bitwise immediate

Fig. 42: *Negated AND bitwise (NAND) instructions*

The NAND instruction specifies *logic 'negated AND' operation* between two operands. The operation is bitwise, i.e., between the corresponding bits of the operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the 'negated AND' operation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	Not used by *Negated AND bitwise register* (it should be always U=0). For *Negated AND bitwise immediate*, U=0 specifies **sign-extended** immediate, while U=1 specifies **zero-extended** immediate. However, although zero- or sign-extended, the immediate is still treated as unsigned.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Negated AND bitwise register*.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Negated AND bitwise immediate*.

Execution

Table 24 summarizes the execution of NAND instructions.

Table 24: *Execution of NAND instructions*

Negated AND bitwise register
REG[DESTINATION] ← REG[DESTINATION] &~ REG[SOURCE]
Negated AND bitwise immediate
REG[DESTINATION] ← REG[DESTINATION] &~ IMMEDIATE14

The NAND instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

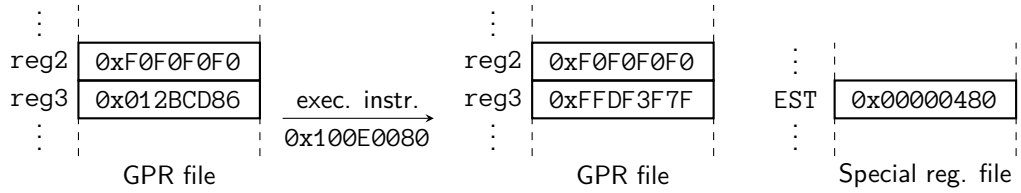
Examples

Example 26: *NAND bitwise register*

Instruction in binary format: 00010000 000011 1000 00 000010 000000
Instruction in hexadecimal format: 0x100E0080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The value in GPR 3 is NAND-ed bitwise with the value in GPR 2. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100E0080). It also shows the state of the special register file after instruction execution.



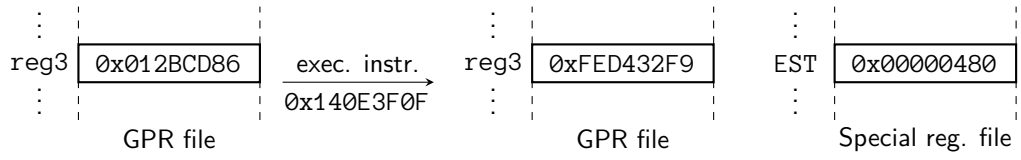
In the EXECUTION STATUS register (EST), the LESS THAN and SIGN flags are set (see Subsection 5.2).

Example 27: *NAND bitwise immediate*

Instruction in binary format: 00010100 000011 1000 11111100001111
Instruction in hexadecimal format: 0x140E3F0F
Fields:

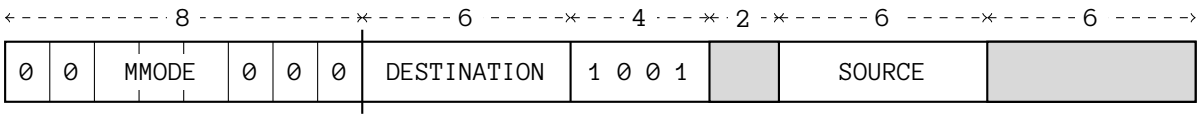
MMODE	010	(word)
U	0	(sign-extended immediate)
DESTINATION	000011	(reg3)
IMMEDIATE14	11111100001111	(0x3F0F)

The value in GPR 3 is NAND-ed bitwise with the sign-extended value given by IMMEDIATE14. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x140E3F0F). It also shows the state of the special register file after instruction execution.

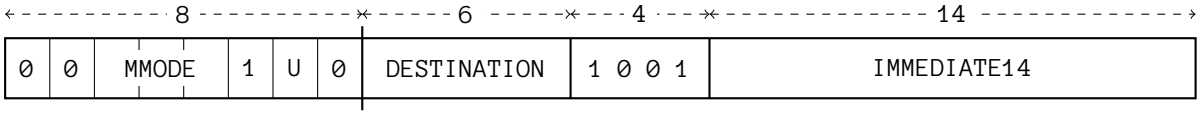


In the EXECUTION STATUS register (EST), the LESS THAN and SIGN flags are set (see Subsection 5.2).

6.1.12 OR – OR bitwise



(a) OR bitwise register



(b) OR bitwise immediate

Fig. 43: *OR bitwise (OR) instructions*

The OR instruction specifies *logic 'OR' operation* between two operands. The operation is bitwise, i.e., between the corresponding bits of the operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the 'OR' operation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	Not used by *OR bitwise register* (it should be always U=0). For *OR bitwise immediate*, U=0 specifies **sign-extended** immediate, while U=1 specifies **zero-extended** immediate. However, although zero- or sign-extended, the immediate is still treated as unsigned.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *OR bitwise register*.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *OR bitwise immediate*.

Execution

Table 25 summarizes the execution of OR instructions.

Table 25: *Execution of OR instructions*

OR bitwise register
REG[DESTINATION] ← REG[DESTINATION] | REG[SOURCE]
OR bitwise immediate
REG[DESTINATION] ← REG[DESTINATION] | IMMEDIATE14

The OR instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

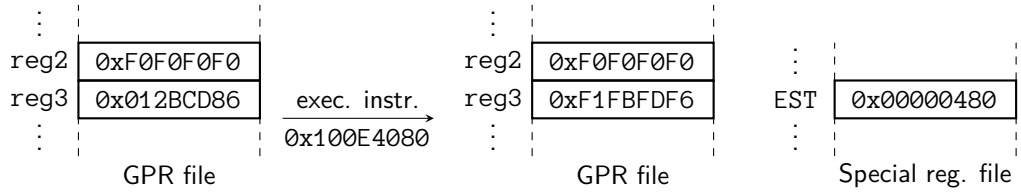
Examples

Example 28: *OR bitwise register*

Instruction in binary format: 00010000 000011 1001 00 000010 000000
Instruction in hexadecimal format: 0x100E4080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The value in GPR 3 is OR-ed bitwise with the value in GPR 2. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100E4080). It also shows the state of the special register file after instruction execution.



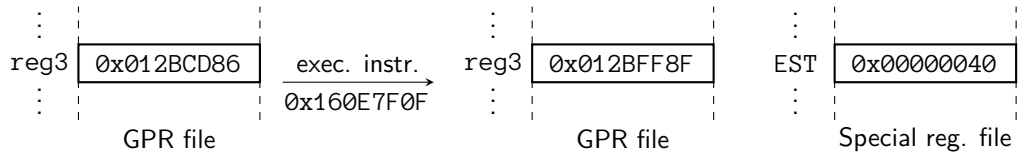
In the EXECUTION STATUS register (EST), the LESS THAN and SIGN flags are set (see Subsection 5.2).

Example 29: *OR bitwise immediate*

Instruction in binary format: 00010110 000011 1001 11111100001111
Instruction in hexadecimal format: 0x160E7F0F
Fields:

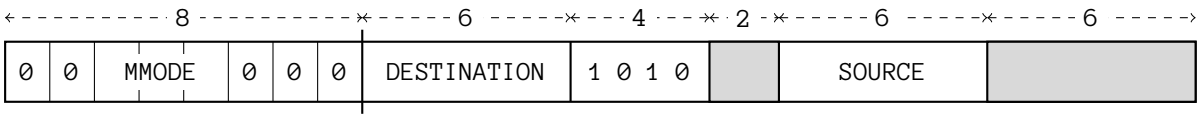
MMODE	010	(word)
U	1	(zero-extended immediate)
DESTINATION	000011	(reg3)
IMMEDIATE14	11111100001111	(0x3F0F)

The value in GPR 3 is OR-ed bitwise with the zero-extended value given by IMMEDIATE14. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x160E7F0F). It also shows the state of the special register file after instruction execution.

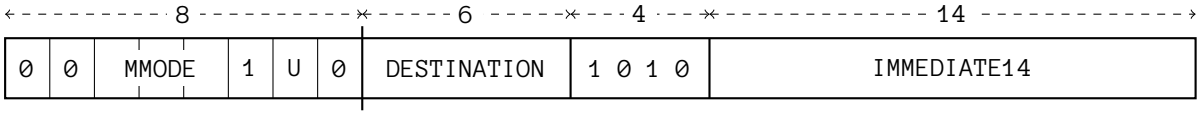


In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

6.1.13 XOR – Exclusive OR bitwise



(a) Exclusive OR bitwise register



(b) Exclusive OR bitwise immediate

Fig. 44: *Exclusive OR bitwise (XOR) instructions*

The XOR instruction specifies *logic 'exclusive OR' operation* between two operands. The operation is bitwise, i.e., between the corresponding bits of the operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the 'exclusive OR' operation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	Not used by *Exclusive OR bitwise register* (it should be always U=0). For *Exclusive OR bitwise immediate*, U=0 specifies **sign-extended** immediate, while U=1 specifies **zero-extended** immediate. However, although zero- or sign-extended, the immediate is still treated as unsigned.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Exclusive OR bitwise register*.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Exclusive OR bitwise immediate*.

Execution

Table 26 summarizes the execution of XOR instructions.

Table 26: *Execution of XOR instructions*

Exclusive OR bitwise register
REG[DESTINATION] ← REG[DESTINATION] ∧ REG[SOURCE]
Exclusive OR bitwise immediate
REG[DESTINATION] ← REG[DESTINATION] ∧ IMMEDIATE14

The XOR instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

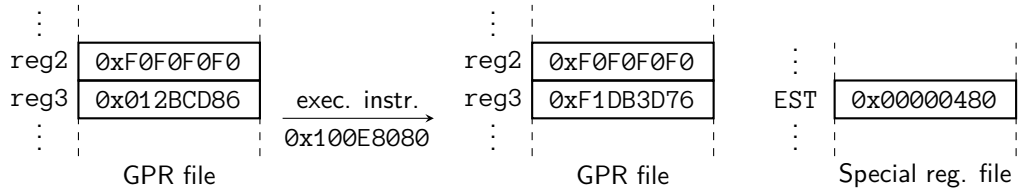
Examples

Example 30: *XOR bitwise register*

Instruction in binary format: 00010000 000011 1010 00 000010 000000
Instruction in hexadecimal format: 0x100E8080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The value in GPR 3 is XOR-ed bitwise with the value in GPR 2. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100E8080). It also shows the state of the special register file after instruction execution.



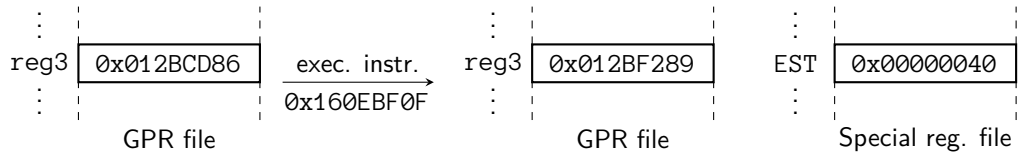
In the EXECUTION STATUS register (EST), the LESS THAN and SIGN flags are set (see Subsection 5.2).

Example 31: *XOR bitwise immediate*

Instruction in binary format: 00010110 000011 1010 11111100001111
Instruction in hexadecimal format: 0x160EBF0F
Fields:

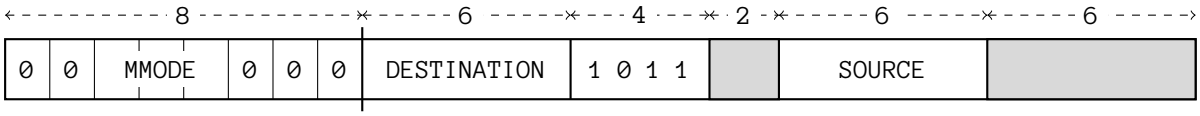
MMODE	010	(word)
U	1	(zero-extended immediate)
DESTINATION	000011	(reg3)
IMMEDIATE14	11111100001111	(0x3F0F)

The value in GPR 3 is XOR-ed bitwise with the zero-extended value given by IMMEDIATE14. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x160EBF0F). It also shows the state of the special register file after instruction execution.

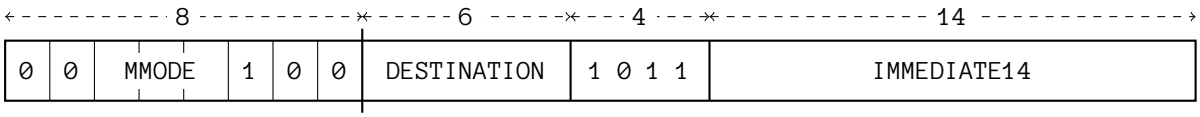


In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

6.1.14 SB – Set bit



(a) Set bit register



(b) Set bit immediate

Fig. 45: *Set bit (SB) instructions*

The SB instruction sets a bit in the first operand to 1. The number of the bit to be set in the first operand is specified by the second operand. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the 'set bit' operation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Set bit register*. That is the number of the bit to be set to 1 in the GPR specified by DESTINATION.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Set bit immediate*. That is the number of the bit to be set to 1 in the GPR specified by DESTINATION.

Execution

Table 27 summarizes the execution of SB instructions.

Table 27: *Execution of SB instructions*

Set bit register
REG[DESTINATION][REG[SOURCE]] ← 1
Set bit immediate
REG[DESTINATION][IMMEDIATE14] ← 1

The SB instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see

Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).

EXECUTION STATUS — Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

INVALID OPERATION — It is raised when the bit number (specified by the second operand) is greater than the specified machine mode or the ALU width (see Subsection 4.2.7).

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

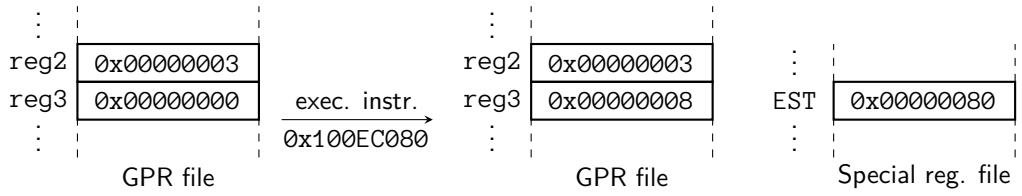
Examples

Example 32: *Set bit register*

Instruction in binary format: 00010000 000011 1011 00 000010 000000
Instruction in hexadecimal format: 0x100EC080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The bit specified by GPR 2 is set in GPR 3. Bit 0 is the LSB, while bit 31 is the MSB. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100EC080). It also shows the state of the special register file after instruction execution.



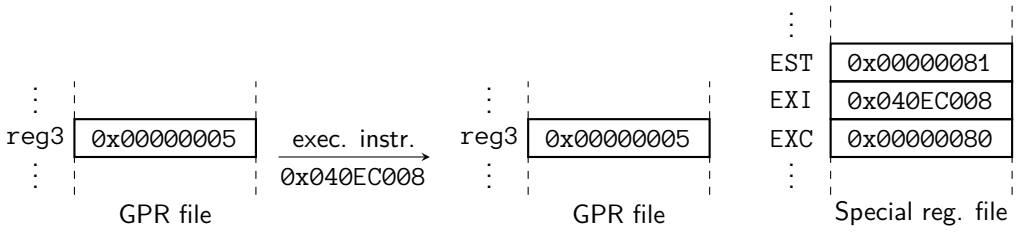
In the EXECUTION STATUS register (EST), only the LESS THAN flag is set (see Subsection 5.2).

Example 33: *Set bit immediate*

Instruction in binary format: 00000100 000011 1011 00000000001000
Instruction in hexadecimal format: 0x040EC008
Fields:

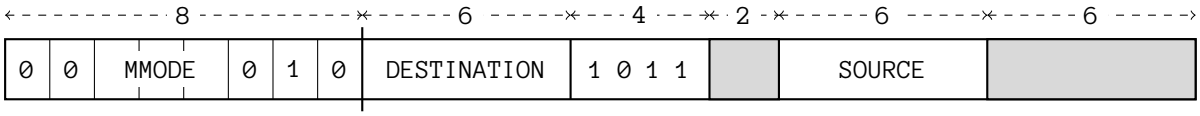
MMODE	000	(byte)
DESTINATION	000011	(reg3)
IMMEDIATE14	00000000001000	(0x8)

The bit specified by IMMEDIATE14 is set in GPR 3. Bit 0 is the LSB, while bit 31 is the MSB. The computed result is written back in GPR 3. However, in this example, the immediate value specifies a bit number which is beyond the byte machine mode. Therefore, the INVALID OPERATION exception is raised, and the DESTINATION GPR 3 is not overwritten. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x040EC008). It also shows the state of the special register file after instruction execution.

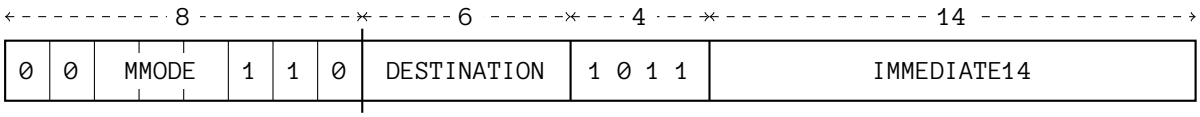


In the EXECUTION STATUS register (EST), the LESS THAN and INVALID OPERATION flags are set (see Subsection 5.2). The EXCEPTION INSTRUCTION register (EXI) is written with the instruction code, and the INVALID OPERATION bit in the EXCEPTION REGISTER (EXC) is set (it is assumed that before executing the instruction the EXC register was zero).

6.1.15 RB – Reset bit



(a) Reset bit register



(b) Reset bit immediate

Fig. 46: *Reset bit (RB) instructions*

The RB instruction resets a bit in the first operand to 0. The number of the bit to be reset in the first operand is specified by the second operand. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the 'reset bit' operation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Reset bit register*. That is the number of the bit to be reset to 0 in the GPR specified by DESTINATION.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Reset bit immediate*. That is the number of the bit to be reset to 0 in the GPR specified by DESTINATION.

Execution

Table 28 summarizes the execution of RB instructions.

Table 28: *Execution of RB instructions*

Reset bit register
REG[DESTINATION][REG[SOURCE]] ← 0
Reset bit immediate
REG[DESTINATION][IMMEDIATE14] ← 0

The RB instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see

Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).

EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

INVALID OPERATION	It is raised when the bit number (specified by the second operand) is greater than the specified machine mode or the ALU width (see Subsection 4.2.7).

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

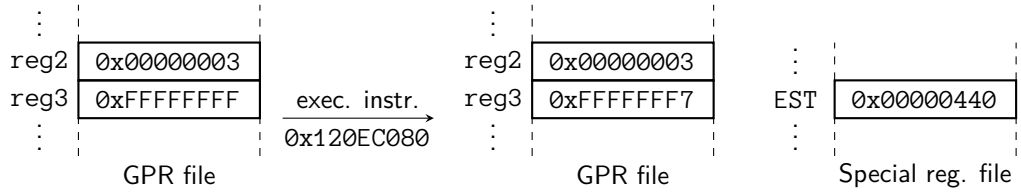
Examples

Example 34: *Reset bit register*

Instruction in binary format: 00010010 000011 1011 00 000010 000000
Instruction in hexadecimal format: 0x120EC080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The bit specified by GPR 2 is reset in GPR 3. Bit 0 is the LSB, while bit 31 is the MSB. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x120EC080). It also shows the state of the special register file after instruction execution.



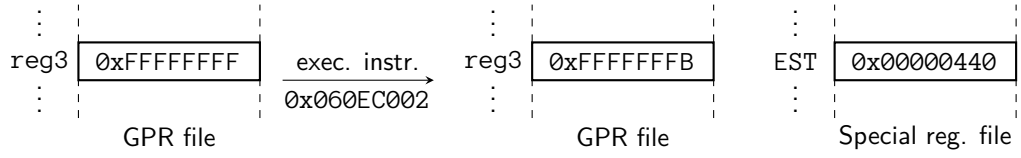
In the EXECUTION STATUS register (EST), the GREATER THAN and SIGN flags are set (see Subsection 5.2).

Example 35: *Reset bit immediate*

Instruction in binary format: 00000110 000011 1011 00000000000010
Instruction in hexadecimal format: 0x060EC002
Fields:

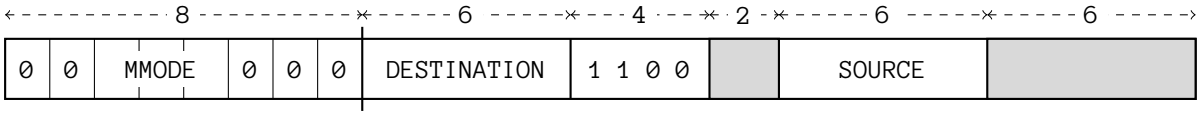
MMODE	000	(byte)
DESTINATION	000011	(reg3)
IMMEDIATE14	00000000000010	(0x2)

The bit specified by IMMEDIATE14 is reset in GPR 3. Bit 0 is the LSB, while bit 31 is the MSB. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x060EC002). It also shows the state of the special register file after instruction execution.

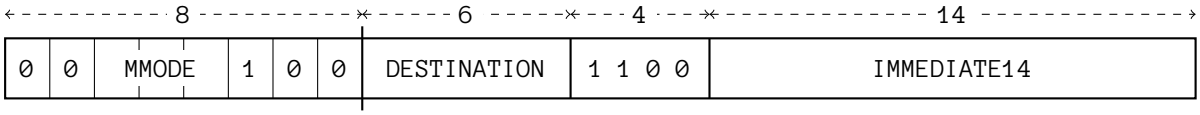


In the EXECUTION STATUS register (EST), the GREATER THAN and SIGN flags are set (see Subsection 5.2).

6.1.16 TB – Test bit



(a) Test bit register



(b) Test bit immediate

Fig. 47: *Test bit (TB) instructions*

The TB instruction tests a bit in the first operand to determine whether its value is 0 or 1 and overwrites the operand with the found value. The number of the bit to be tested in the first operand is specified by the second operand. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the 'test bit' operation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Test bit register*. That is the number of the bit to be tested in the GPR specified by DESTINATION.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Test bit immediate*. That is the number of the bit to be tested in the GPR specified by DESTINATION.

Execution

Table 29 summarizes the execution of TB instructions.

Table 29: *Execution of TB instructions*

Test bit register
REG[DESTINATION] ← REG[DESTINATION][REG[SOURCE]]
Test bit immediate
REG[DESTINATION] ← REG[DESTINATION][IMMEDIATE14]

The TB instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see

Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).

EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

INVALID OPERATION	It is raised when the bit number (specified by the second operand) is greater than the specified machine mode or the ALU width (see Subsection 4.2.7).

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

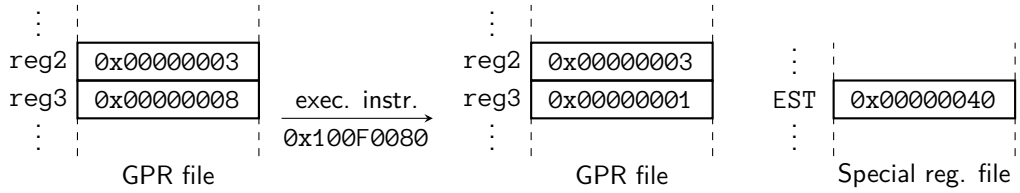
Examples

Example 36: *Test bit register*

Instruction in binary format: 00010000 000011 1100 00 000010 000000
Instruction in hexadecimal format: 0x100F0080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The bit specified by GPR 2 is checked in GPR 3 whether it is 0 or 1 and GPR 3 is rewritten with 0 or 1, respectively. Bit 0 is the LSB, while bit 31 is the MSB. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100F0080). It also shows the state of the special register file after instruction execution.



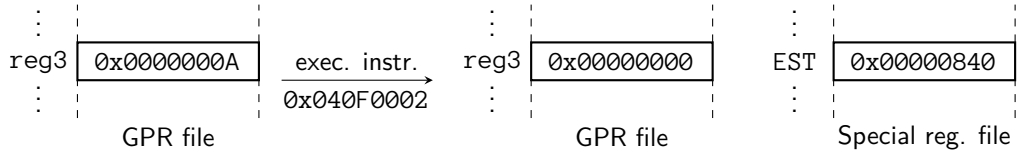
In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

Example 37: *Test bit immediate*

Instruction in binary format: 00000100 000011 1100 00000000000010
Instruction in hexadecimal format: 0x040F0002
Fields:

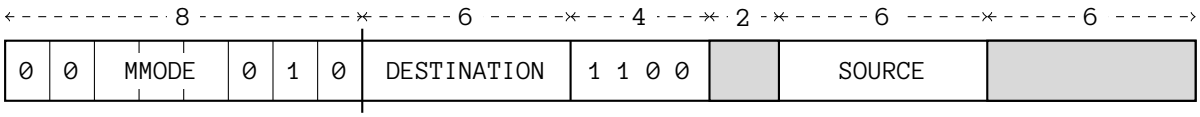
MMODE	000	(byte)
DESTINATION	000011	(reg3)
IMMEDIATE14	00000000000010	(0x2)

The bit specified by IMMEDIATE14 is checked in GPR 3 whether it is 0 or 1 and GPR 3 is rewritten with 0 or 1, respectively. Bit 0 is the LSB, while bit 31 is the MSB. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x040F0002). It also shows the state of the special register file after instruction execution.

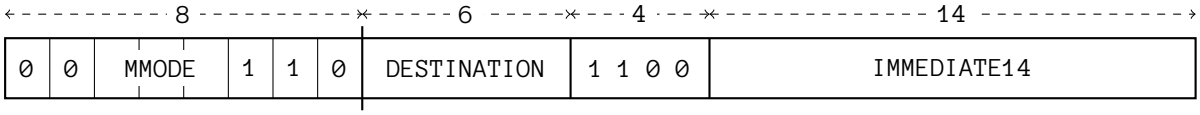


In the EXECUTION STATUS register (EST), the GREATER THAN and ZERO flags are set (see Subsection 5.2).

6.1.17 RVB – Reverse bits



(a) Reverse bits register



(b) Reverse bits immediate

Fig. 48: *Reverse bits (RVB) instructions*

The RVB instruction reverses the bits of the first operand. The number of bits to be reversed in the first operand (starting from bit 0) is specified by the second operand. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the 'reverse bits' operation. The second operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand for *Reverse bits register*. That is the number of the highest bit ⟨hi⟩ in the range [⟨hi⟩:0] that is to be reversed in the GPR specified by DESTINATION.
IMMEDIATE14	A 14-bit immediate value supplied as second operand for *Reverse bits immediate*. That is the number of the highest bit ⟨hi⟩ in the range [⟨hi⟩:0] that is to be reversed in the GPR specified by DESTINATION.

Execution

Table 30 summarizes the execution of RVB instructions.

Table 30: *Execution of RVB instructions*

Reverse bits register
REG[DESTINATION][REG[SOURCE]:0] ← reverse(REG[DESTINATION][REG[SOURCE]:0])
Reverse bits immediate
REG[DESTINATION][IMMEDIATE14:0] ← reverse(REG[DESTINATION][IMMEDIATE14:0])

The reverse() function reverses the bit order in a bit field. For example, reverse(110) = 011, reverse(0101) = 1010, reverse(00001) = 10000, etc.

The RVB instruction always treats both of the input operands as unsigned.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

INVALID OPERATION	It is raised when the bit number (specified by the second operand) is greater than the specified machine mode or the ALU width (see Subsection 4.2.7).

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

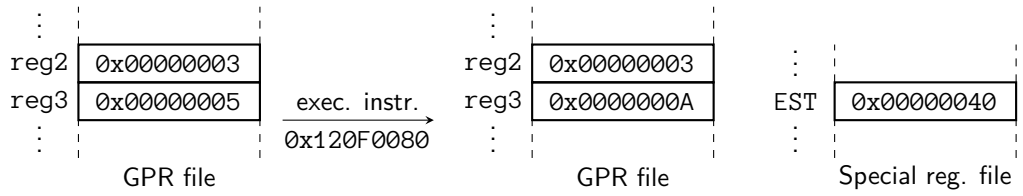
Examples

Example 38: *Reverse bits register*

Instruction in binary format: 00010010 000011 1100 00 000010 000000
Instruction in hexadecimal format: 0x120F0080
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The bits in GPR 3 starting at position zero and up to the bit specified by GPR 2 are reversed. Bit 0 is the LSB, while bit 31 is the MSB. The computed result is written back in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x120F0080). It also shows the state of the special register file after instruction execution.



In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

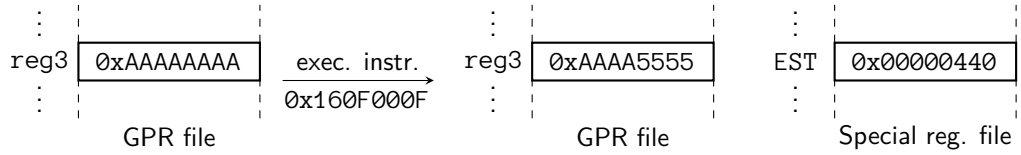
Example 39: *Reverse bits immediate*

Instruction in binary format: 00010110 000011 1100 00000000001111
Instruction in hexadecimal format: 0x160F000F
Fields:

MMODE	010	(word)
DESTINATION	000011	(reg3)
IMMEDIATE14	00000000001111	(0xF)

The bits in GPR 3 starting at position zero and up to the bit specified by IMMEDIATE14 are reversed. Bit 0 is the LSB, while bit 31 is the MSB. The computed result is written back in

GPR 3. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (`0x160F000F`). It also shows the state of the special register file after instruction execution.



In the `EXECUTION STATUS` register (`EST`), the `GREATER THAN` and `SIGN` flags are set (see Subsection 5.2).

6.1.18 FADD – FP Add

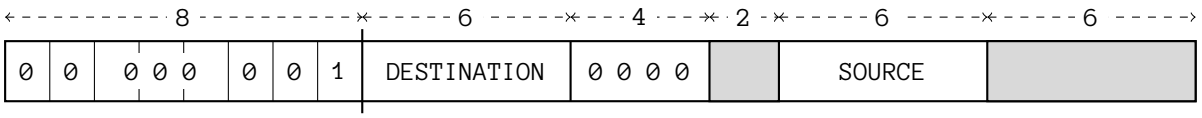


Fig. 49: *FP Add (FADD) instructions*

The FADD instruction specifies *FP addition* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the addition. The second operand is also in a GPR.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.

Execution

Table 31 summarizes the execution of FADD instructions.

Table 31: *Execution of FADD instructions*

FP Add
REG[DESTINATION] ← REG[DESTINATION] + REG[SOURCE]

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when (at least) one of the operands is a signaling NaN, or when the result is NaN. It is also raised when the operands imply addition of infinities with opposite signs, e.g., positive infinity plus negative infinity. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.
FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.
FP UNDERFLOW	It is raised when the result is a tiny non-zero number. See Subsection 4.2.14.

FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

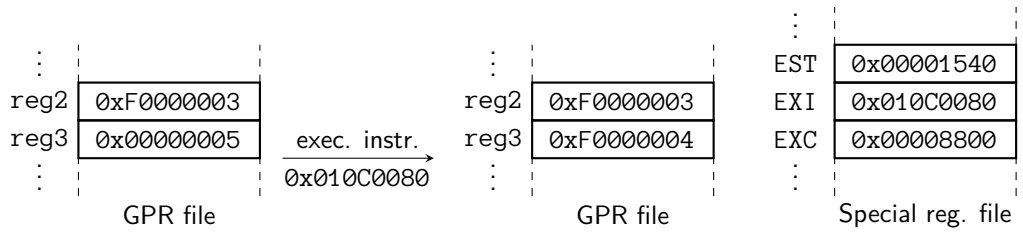
Examples

Example 40: *FP Add*

Instruction in binary format: `00000001 000011 0000 00 000010 000000`
Instruction in hexadecimal format: `0x010C0080`
Fields:

DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The instruction adds the value of GPR 2 to GPR 3 and writes the computed result back in GPR 3. The instruction assumes that the input operands are in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. This instruction raises the FP DENORMALIZED OPERAND and the FP INEXACT RESULT exceptions. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x010C0080). It also shows the state of the special register file after instruction execution.



In the EXECUTION STATUS register (EST), the GREATER THAN, the INEXACT, the SIGN and the DENORMALIZED flags are set.

Because of the raised exceptions, the EXCEPTION INSTRUCTION register (EXI) is written with the instruction code, and the FP DENORMALIZED OPERAND and FP INEXACT RESULT bits in the EXCEPTION REGISTER (EXC) are set (assuming that before executing the instruction the EXC register was zero). However, it is also assumed that the raised exceptions are impotent. Therefore, the result is written back to the DESTINATION GPR 3. If at least one of the exceptions was potent, GPR 3 and the EST register would not be overwritten and the exception handling procedure would have been started.

6.1.19 FSUB – FP Subtract

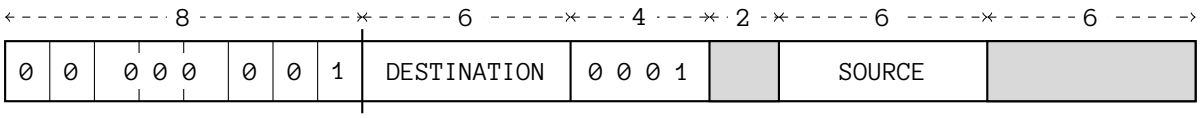


Fig. 50: *FP Subtract (FSUB) instructions*

The FSUB instruction specifies *FP subtraction* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the subtraction. The second operand is also in a GPR.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.

Execution

Table 32 summarizes the execution of FSUB instructions.

Table 32: *Execution of FSUB instructions*

FP Subtract
REG[DESTINATION] ← REG[DESTINATION] – REG[SOURCE]

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when (at least) one of the operands is a signaling NaN, or when the result is NaN. It is also raised when the operands imply subtraction of infinities with opposite signs, e.g., positive infinity minus positive infinity. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.
FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.
FP UNDERFLOW	It is raised when the result is a tiny non-zero number. See Subsection 4.2.14.

FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

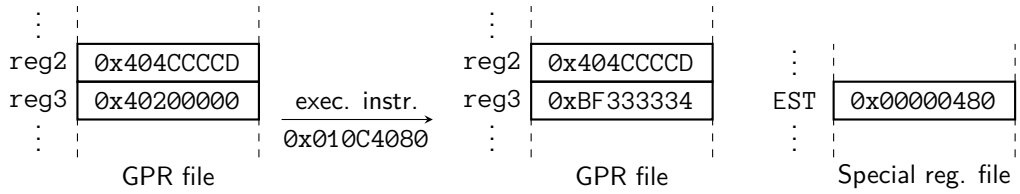
Examples

Example 41: *FP Subtract*

Instruction in binary format: 00000001 000011 0001 00 000010 000000
Instruction in hexadecimal format: 0x010C4080
Fields:

DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The instruction subtracts the value in GPR 2 from GPR 3 and writes the computed result back in GPR 3. The instruction assumes that the input operands are in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x010C4080). It also shows the state of the special register file after instruction execution.



In the EXECUTION STATUS register (EST), the LESS THAN and SIGN flags are set (see Subsection 5.2).

6.1.20 FMUL – FP Multiply

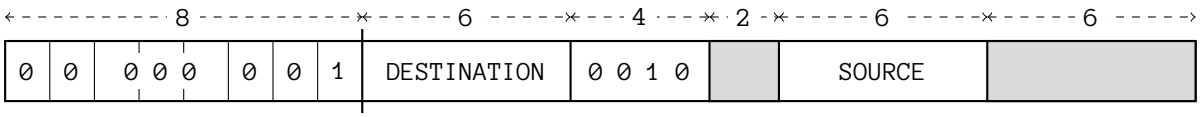


Fig. 51: *FP Multiply (FMUL) instructions*

The FMUL instruction specifies *FP multiplication* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the multiplication. The second operand is also in a GPR.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.

Execution

Table 33 summarizes the execution of FMUL instructions.

Table 33: *Execution of FMUL instructions*

FP Multiply
REG[DESTINATION] ← REG[DESTINATION] × REG[SOURCE]

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when (at least) one of the operands is a signaling NaN, or when the result is NaN. It is also raised when one multiplication operand is zero and the other is infinity. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.
FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.
FP UNDERFLOW	It is raised when the result is a tiny non-zero number. See Subsection 4.2.14.

FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

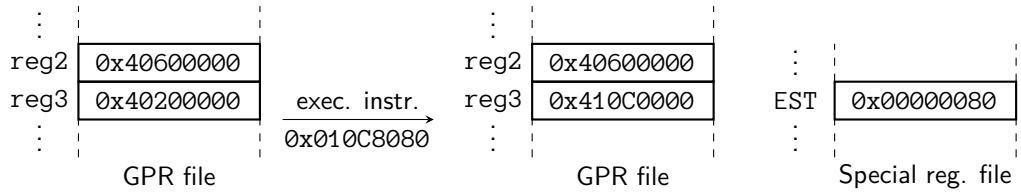
Examples

Example 42: *FP Multiply*

Instruction in binary format: 00000001 000011 0010 00 000010 000000
Instruction in hexadecimal format: 0x010C8080
Fields:

DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The instruction multiplies the value in GPR 2 to the value in GPR 3 and writes the computed result back in GPR 3. The instruction assumes that the input operands are in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x010C8080). It also shows the state of the special register file after instruction execution.



In the EXECUTION STATUS register (EST), only the LESS THAN flag is set (see Subsection 5.2).

6.1.21 FDIV – FP Divide

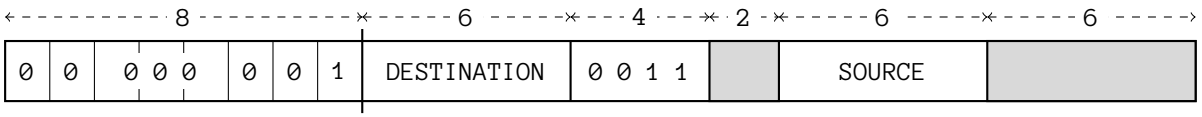


Fig. 52: *FP Divide (FDIV) instructions*

The FDIV instruction specifies *FP division* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the division. The second operand is also in a GPR.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.

Execution

Table 34 summarizes the execution of FDIV instructions.

Table 34: *Execution of FDIV instructions*

FP Divide
REG[DESTINATION] ← REG[DESTINATION] ÷ REG[SOURCE]

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when (at least) one of the operands is a signaling NaN, or when the result is NaN. It is also raised when both operands are zero or both are infinity. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.
FP DIVISION BY ZERO	It is raised when the divisor is zero and the dividend is a finite, nonzero FP number. See Subsection 4.2.12.
FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.

FP UNDERFLOW	It is raised when the result is a tiny non-zero number. See Subsection 4.2.14.
FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

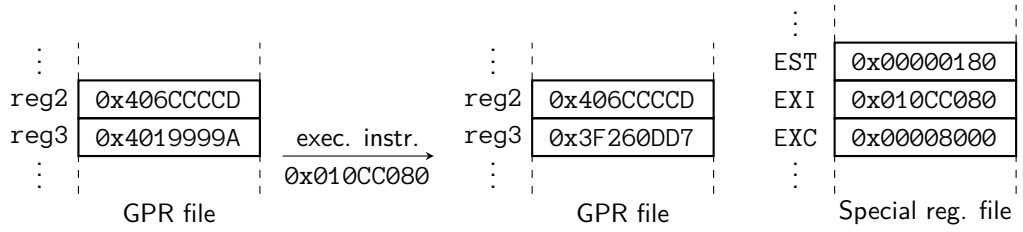
Examples

Example 43: *FP Divide*

Instruction in binary format: 00000001 000011 0011 00 000010 000000
Instruction in hexadecimal format: 0x010CC080
Fields:

DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The divisor in GPR 2 divides the dividend in GPR 3. The computed result of the division is written back in GPR 3. The instruction assumes that the input operands are in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x010CC080). It also shows the state of the special register file after instruction execution.



In the EXECUTION STATUS register (EST), the LESS THAN and INEXACT flags are set (see Subsection 5.2).

Because of the raised FP INEXACT RESULT exception, the EXCEPTION INSTRUCTION register (EXI) is written with the instruction code, and the FP INEXACT RESULT bit in the EXCEPTION REGISTER (EXC) is set (assuming that before executing the instruction the EXC register was zero). However, it is also assumed that the FP INEXACT RESULT exception is impotent. Therefore, the result is written back to the DESTINATION GPR 3.

6.1.22 FREM – FP Remainder

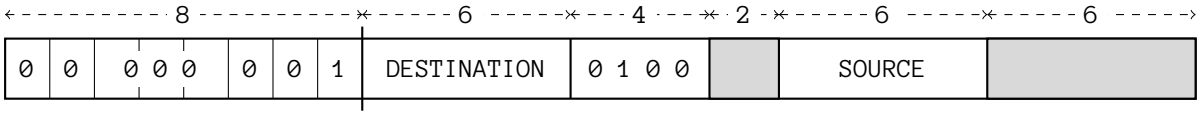


Fig. 53: *FP Remainder (FREM) instructions*

The FREM instruction specifies *FP remainder operation* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the remainder operation. The second operand is also in a GPR.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.

Execution

Table 35 summarizes the execution of FREM instructions.

Table 35: *Execution of FREM instructions*

FP Remainder
REG[DESTINATION] ← REG[DESTINATION] % REG[SOURCE]

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when (at least) one of the operands is a signaling NaN, or when the result is NaN. It is also raised when none of the operands is NaN, and the DESTINATION operand is infinity or the SOURCE operand is zero. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

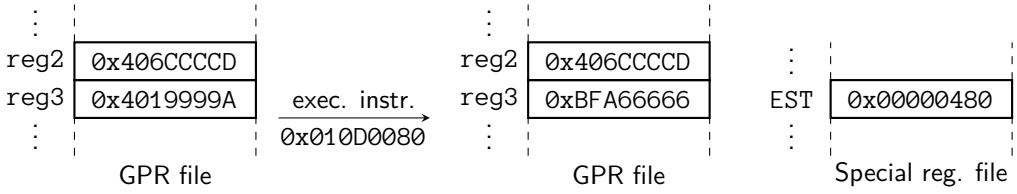
Examples

Example 44: *FP Remainder*

Instruction in binary format: `00000001 000011 0100 00 000010 000000`
Instruction in hexadecimal format: `0x010D0080`
Fields:

DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

The remainder of the division of GPR 3 by GPR 2 is computed and written back in GPR 3. The instruction assumes that the input operands are in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (`0x010D0080`). It also shows the state of the special register file after instruction execution.



In the EXECUTION STATUS register (EST), the LESS THAN and SIGN flags are set (see Subsection 5.2).

6.1.23 FCMP – FP Compare

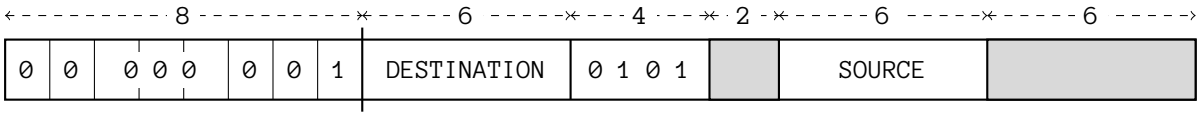


Fig. 54: *FP Compare (FCMP) instructions*

The FCMP instruction specifies *FP comparison* of two operands. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the comparison. The second operand is also in a GPR.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.

Execution

Table 36 summarizes the execution of FCMP instructions.

Table 36: *Execution of FCMP instructions*

FP Compare
REG[DESTINATION] ← REG[DESTINATION] <==> REG[SOURCE]

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. Only the lower 16-bits of the destination GPR are changed with the same content as the EXECUTION STATUS register. Of course, if the destination GPR is shorter than 16 bits, register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when (at least) one of the operands is a signaling NaN. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

Examples

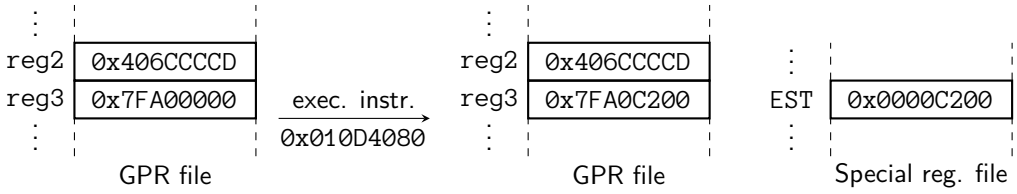
Example 45: *FP Compare*

Instruction in binary format: `00000001 000011 0101 00 000010 000000`
Instruction in hexadecimal format: `0x010D4080`

Fields:

DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)

An FP comparison between the floating point numbers in GPR 3 and GPR 2 is made. The result, i.e., the contents of the EXECUTION STATUS (EST) register is written back in GPR 3. The instruction assumes that the input operands are in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x010D4080). It also shows the state of the special register file after instruction execution.



Thus, the lower 16 bits of the GPR 3 are rewritten with the bits of the EST register.

In the EST register, the UNORDERED, the SIGNALING NAN and the NAN flags are set.

6.1.24 FSQR – FP Square root

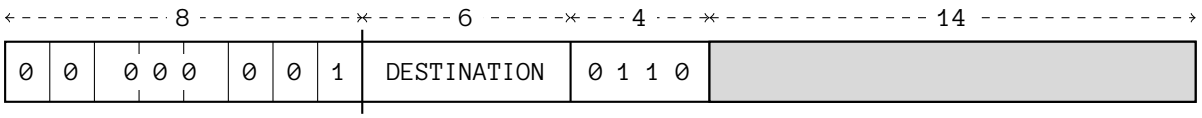


Fig. 55: *FP Square root (FSQR) instructions*

The FSQR instruction specifies finding the FP square root of an operand. The operand resides in a GPR in which the result will be written back.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).

Execution

Table 37 summarizes the execution of FSQR instructions.

Table 37: *Execution of FSQR instructions*

FP Square root
REG[DESTINATION] ← sqrt(REG[DESTINATION])

The sqrt() function computes the square root of the FP argument and returns the result also in FP format.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when the operand is a signaling NaN. It is also raised when the operand is less than zero. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.
FP UNDERFLOW	It is raised when the result is a tiny non-zero number. See Subsection 4.2.14.
FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW

exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

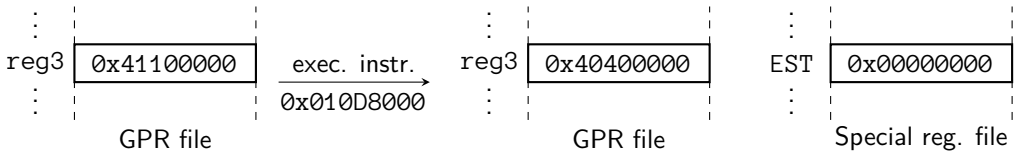
Examples

Example 46: *FP Square root*

Instruction in binary format: 00000001 000011 0110 00000000000000
Instruction in hexadecimal format: 0x010D8000
Fields:

DESTINATION 000011 (reg3)

The square root of the operand in GPR 3 is computed and written back in GPR 3. The instruction assumes that the input operand is in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x010D8000). It also shows the state of the special register file after instruction execution.



The EXECUTION STATUS (EST) register is set to zero (see Subsection 5.2).

6.1.25 FABS – FP Absolute

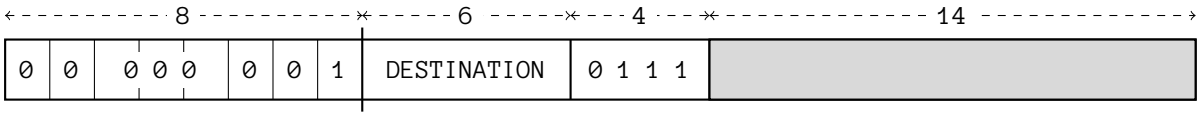


Fig. 56: *FP Absolute (FABS) instructions*

The FABS instruction specifies finding the FP absolute value of an operand. The operand resides in a GPR in which the result will be written back.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).

Execution

Table 38 summarizes the execution of FABS instructions.

Table 38: *Execution of FABS instructions*

FP Absolute
REG[DESTINATION] ← abs(REG[DESTINATION])

The abs() function computes the absolute value of the FP argument and returns the result also in FP format.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when the operand is a signaling NaN. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

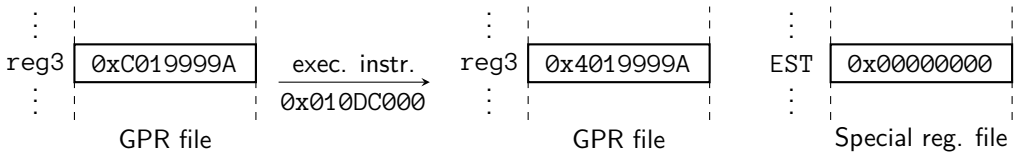
Examples

Example 47: *FP Absolute*

Instruction in binary format: `00000001 000011 0111 00000000000000`
Instruction in hexadecimal format: `0x010DC000`
Fields:

DESTINATION	000011	(reg3)

The absolute value of the operand in GPR 3 is written back in GPR 3. The instruction assumes that the input operand is in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (`0x010DC000`). It also shows the state of the special register file after instruction execution.



The `EXECUTION STATUS` (`EST`) register is set to zero (see Subsection 5.2).

6.1.26 FNEG – FP Negate

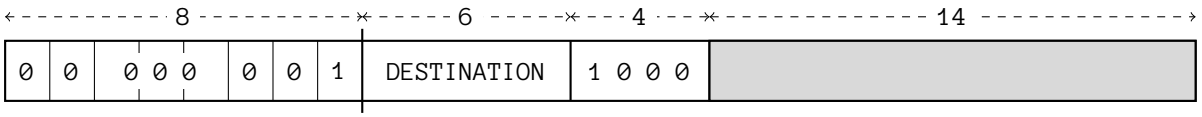


Fig. 57: *FP Negate (FNEG) instructions*

The FNEG instruction specifies finding the FP negated value of an operand. The operand resides in a GPR in which the result will be written back.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).

Execution

Table 39 summarizes the execution of FNEG instructions.

Table 39: *Execution of FNEG instructions*

FP Negate
REG[DESTINATION] ← −(REG[DESTINATION])

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when the operand is a signaling NaN. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

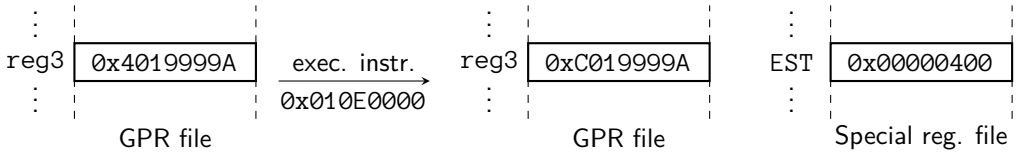
Examples

Example 48: *FP Negate*

Instruction in binary format: 00000001 000011 1000 00000000000000
Instruction in hexadecimal format: 0x010E0000
Fields:

DESTINATION	000011	(reg3)

The negated value of the operand in GPR 3 is written back in GPR 3. The instruction assumes that the input operand is in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (`0x010E0000`). It also shows the state of the special register file after instruction execution.



In the `EXECUTION STATUS` register (`EST`), only the `SIGN` flag is set (see Subsection 5.2).

6.1.27 FRND – FP Round to integer

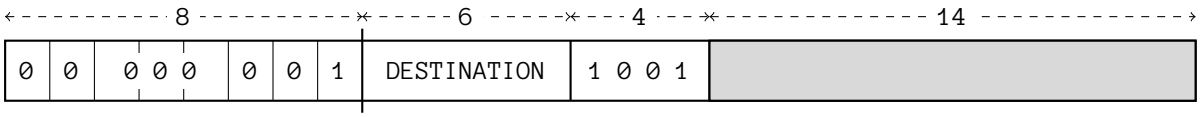


Fig. 58: *FP Round to integer (FRND) instructions*

The FRND instruction specifies rounding the FP operand to an integer value (also represented in FP format). The operand resides in a GPR in which the result will be written back.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).

Execution

Table 40 summarizes the execution of FRND instructions.

Table 40: *Execution of FRND instructions*

FP Round to integer
REG[DESTINATION] ← roundint(REG[DESTINATION])

The roundint() function rounds an FP number to integer number in FP format. For example, roundint(5.32) = 5.0, roundint(5.8) = 6.0, etc. It is also dependent on the FP rounding mode (see Subsection 5.13).

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when the operand is a signaling NaN. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.
FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.

FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

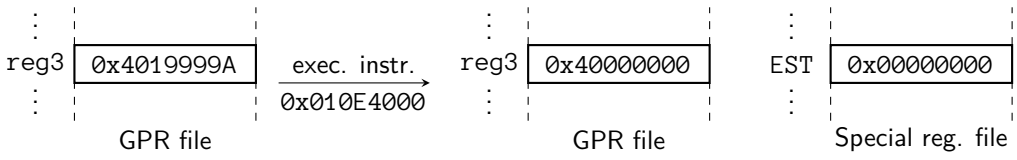
Examples

Example 49: *FP Round to integer*

Instruction in binary format: 00000001 000011 1001 00000000000000
Instruction in hexadecimal format: 0x010E4000
Fields:

DESTINATION 000011 (reg3)

The FP number supplied by GPR 3 is rounded (in FP format) and written back in GPR 3. The instruction assumes that the input operand is in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x010E4000). It also shows the state of the special register file after instruction execution.



The EXECUTION STATUS (EST) register is set to zero (see Subsection 5.2).

6.1.28 FF2I – FP to integer

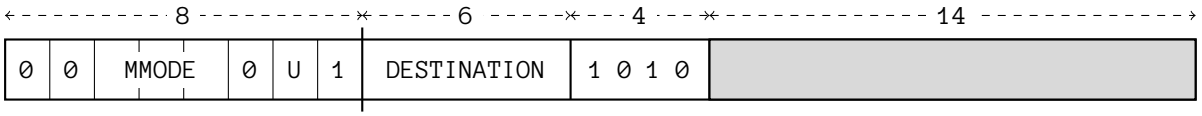


Fig. 59: *FP to integer (FF2I) instructions*

The FF2I instruction specifies rounding to integer and conversion of the FP operand to integer format. The operand resides in a GPR in which the result will be written back.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **signed** integer 1: **unsigned** integer
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).

Execution

Table 41 summarizes the execution of FF2I instructions.

Table 41: *Execution of FF2I instructions*

FP to integer
REG[DESTINATION] ← int(REG[DESTINATION])

The int() function rounds an FP number to integer number in integer format. For example, int(5.32) = 5, int(5.8) = 6, etc. It is also dependent on the FP rounding mode (see Subsection 5.13).

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If MMODE specifies shorter width than the GPR width, only the corresponding lower bits of the destination GPR are changed (see Subsection 2.2.2). If MMODE specifies wider width than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when the operand is a signaling NaN. It is also raised when the operand is infinity or NaN, or, when the operand is greater than the maximal representable integer in the specified machine mode. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.

FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.
FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

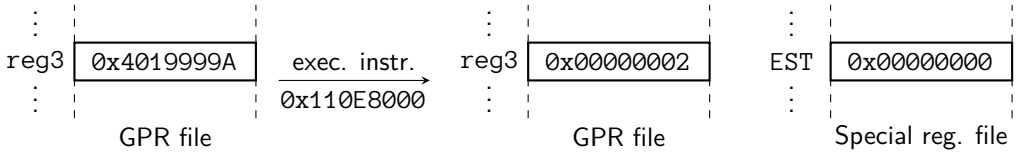
Examples

Example 50: *FP to integer*

Instruction in binary format: 00010001 000011 1010 00000000000000
Instruction in hexadecimal format: 0x110E8000
Fields:

MMODE	010	(to word integer)
U	0	(to signed integer)
DESTINATION	000011	(reg3)

The FP number supplied by GPR 3 is (rounded and) converted to integer format and is written back in GPR 3. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x110E8000). It also shows the state of the special register file after instruction execution.



The EXECUTION STATUS (EST) register is set to zero (see Subsection 5.2).

6.1.29 FI2F – Integer to FP

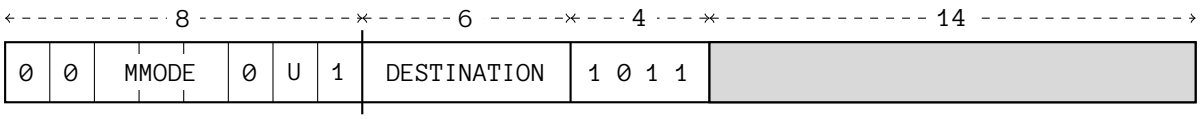


Fig. 60: *Integer to FP (FI2F) instructions*

The FI2F instruction specifies conversion of the integer operand to FP format. The operand resides in a GPR in which the result will be written back.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **signed** integer 1: **unsigned** integer
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).

Execution

Table 42 summarizes the execution of FI2F instructions.

Table 42: *Execution of FI2F instructions*

Integer to FP
REG[DESTINATION] ← fpn(REG[DESTINATION])

The fpn() function converts an integer number in integer format to its representation in FP format. For example, fpn(5) = 5.0.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.
FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

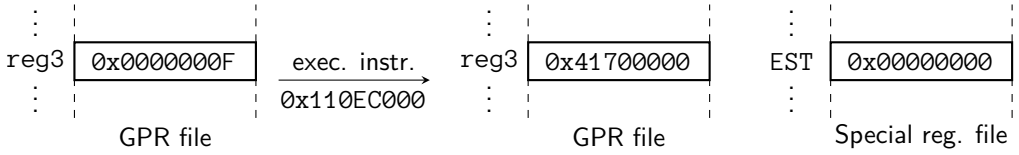
Examples

Example 51: *Integer to FP*

Instruction in binary format: `00010001 000011 1011 00000000000000`
Instruction in hexadecimal format: `0x110EC000`
Fields:

MMODE	010	(from word integer)
U	0	(from signed integer)
DESTINATION	000011	(reg3)

The number in integer format supplied by GPR 3 is converted to its representation in FP format and is written back to GPR 3. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (`0x110EC000`). It also shows the state of the special register file after instruction execution.



The `EXECUTION STATUS` (`EST`) register is set to zero (see Subsection 5.2).

6.1.30 FEXT – Extend FP format

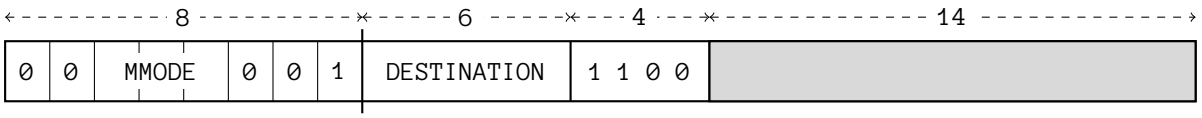


Fig. 61: *Extend FP format (FEXT) instructions*

The FEXT instruction specifies *extending* the operand's FP format specified by the machine mode (MMODE) field to the maximal supported FP format. The operand resides in a GPR in which the result will be written back.

Fields

MMODE	Specifies the FP machine mode. However, here only 16-, 32-, 64-, 128- and 256-bit machine modes, i.e., H/W/D/Q/1 are allowed. The encoding of the bit field is also according to Table 1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).

Execution

Table 43 summarizes the execution of FEXT instructions.

Table 43: *Execution of FEXT instructions*

Extend FP format
REG[DESTINATION] ← ext(REG[DESTINATION])

The ext() function extends the FP number in a given FP format to the maximal supported FP format in the implementation.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when the operand is a signaling NaN. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

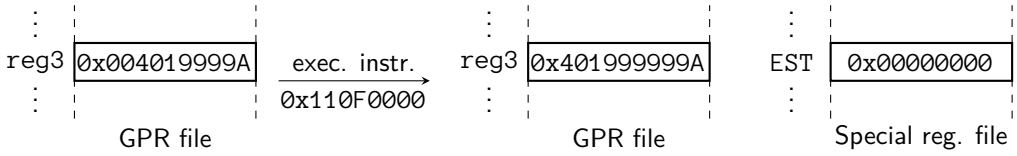
Examples

Example 52: *Extend FP format*

Instruction in binary format: `00010001 000011 1100 00000000000000`
Instruction in hexadecimal format: `0x110F0000`
Fields:

MMODE	010	(from 32-bit FP format)
DESTINATION	000011	(reg3)

The FP number supplied by GPR 3 is extended to the full width of the internal FP format and is written back to GPR 3. In this example, the format of the input operand is 32-bit wide according to the specified MMODE, while the internal FP format of the GPR file is 40 bits. Both formats are according to the IEEE Std 754-2008 standard [3].



Thus, assuming that a 32-bit FP number (with 23-bit wide mantissa) is loaded from memory to GPR 3, it is then reformatted, i.e., extended by the FEXT instruction (`0x110F0000`) to the internal 40-bit width (of which 31 bits are for the mantissa).

The EXECUTION STATUS (EST) register is set to zero (see Subsection 5.2).

6.1.31 FSQZ – Squeeze FP format

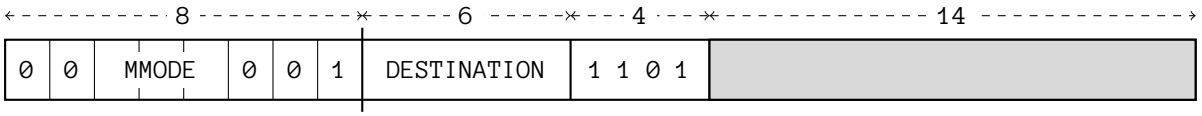


Fig. 62: *Squeeze FP format (FSQZ) instructions*

The FSQZ instruction specifies *squeezing* the operand's maximal supported FP format to the FP format specified by the machine mode (MMODE) field. The operand resides in a GPR in which the result will be written back.

Fields

MMODE	Specifies the FP machine mode. However, here only 16-, 32-, 64-, 128- and 256-bit machine modes, i.e., H/W/D/Q/1 are allowed. The encoding of the bit field is also according to Table 1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).

Execution

Table 44 summarizes the execution of FSQZ instructions.

Table 44: *Execution of FSQZ instructions*

Squeeze FP format
REG[DESTINATION] ← sqz(REG[DESTINATION])

The sqz() function squeezes the FP number in the maximal supported FP format to a shorter FP format.

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when the operand is a signaling NaN. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.
FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.

FP UNDERFLOW	It is raised when the result is a tiny non-zero number. See Subsection 4.2.14.
FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

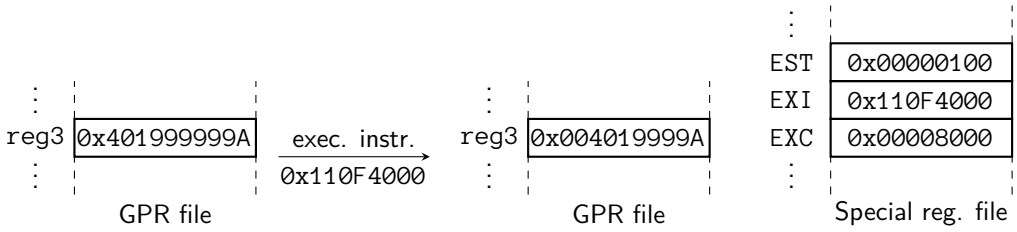
Examples

Example 53: *Squeeze FP format*

Instruction in binary format: 00010001 000011 1101 00000000000000
Instruction in hexadecimal format: 0x110F4000
Fields:

MMODE	010	(to 32-bit FP format)
DESTINATION	000011	(reg3)

The FP number supplied by GPR 3 is squeezed to the FP width specified by MMODE and is written back to GPR 3. In this example, the internal FP format of the GPR file is 40 bits, while the output format of the result is specified to be 32-bit wide. Both formats are according to the IEEE Std 754-2008 standard [3].

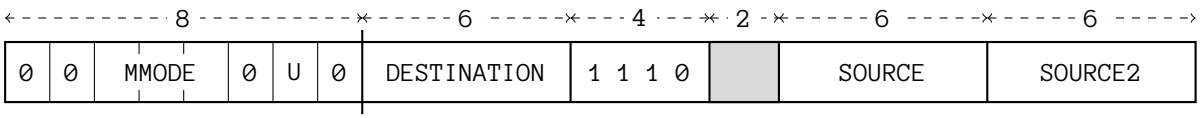


Thus, assuming that a 32-bit FP number needs to be stored in memory, the FP number supplied by GPR 3 is reformatted, i.e., squeezed by the FSQZ instruction (0x110F4000) from the internal 40-bit width (of which 31 bits are for the mantissa) to the 32-bit FP format (in which 23 bits are for the mantissa).

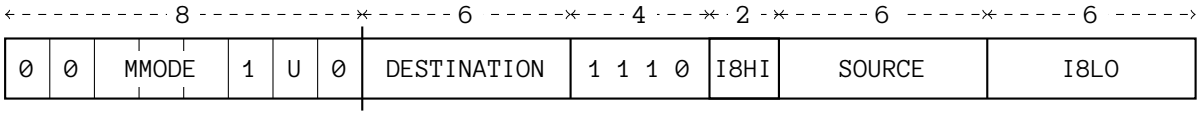
In the EXECUTION STATUS register (EST), only the INEXACT flag is set (see Subsection 5.2).

Because of the raised FP INEXACT RESULT exception, the EXCEPTION INSTRUCTION register (EXI) is written with the instruction code, and the FP INEXACT RESULT bit in the EXCEPTION REGISTER (EXC) is set (assuming that before executing the instruction the EXC register was zero). However, it is also assumed that the FP INEXACT RESULT exception is impotent. Therefore, the result is written back to the DESTINATION GPR 3.

6.1.32 MAD – Multiply-add



(a) Multiply-add register



(b) Multiply-add immediate

Fig. 63: *Multiply-add (MAD) instructions*

The MAD instruction specifies *fused multiplication-addition*, i.e., multiplication of two operands and addition of a third operand to the product. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the fused multiplication-addition. The second operand is also in a GPR. The third operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **signed** operation (including sign-extended immediate for *Multiply-add immediate*). 1: **unsigned** operation (including zero-extended immediate for *Multiply-add immediate*). See Subsection 3.2.1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.
SOURCE2	Specifies the number of the GPR containing the third operand.
I8HI	The two MSBs of the 8-bit IMMEDIATE8.
I8LO	The six LSBs of the 8-bit IMMEDIATE8.

Execution

Table 45 summarizes the execution of MAD instructions.

Table 45: *Execution of MAD instructions*

Multiply-add register
REG[DESTINATION] ← REG[DESTINATION] × REG[SOURCE] + REG[SOURCE2]
Multiply-add immediate
REG[DESTINATION] ← REG[DESTINATION] × REG[SOURCE] + IMMEDIATE8

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. However, the MAD instruction always returns a result which is twice the width of the input operands specified by MMODE. Thus, depending on the machine mode and the GPR width, subsequent GPRs may be written according to the property of circularity (see

Subsection 2.2.3) in little-endian ordering. For example, if both the GPR width and the MMODE is 32 bits, then the result is 64-bit wide which will be written in two subsequent GPRs, i.e., the lower part in the GPR specified by the DESTINATION field, and the upper part in the subsequent GPR. On the other side, if the GPR width is 32 bits and MMODE is 16 bits, the result is 32-bit wide, and will be written in a single register specified by DESTINATION.

EXECUTION STATUS — Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

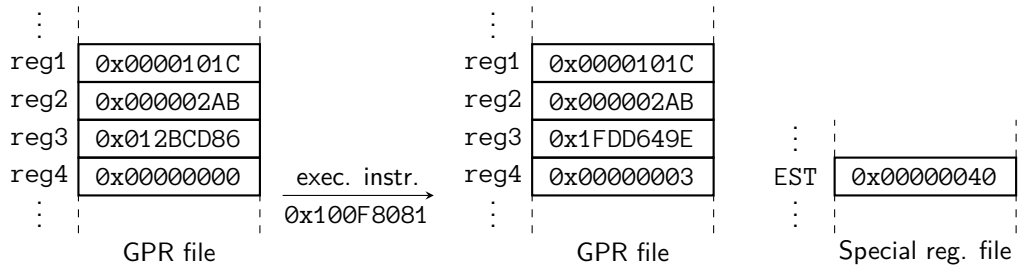
Examples

Example 54: *Multiply-add register*

Instruction in binary format: 00010000 000011 1110 00 000010 000001
Instruction in hexadecimal format: 0x100F8081
Fields:

MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)
SOURCE2	000001	(reg1)

The instruction multiplies the value in GPR 2 to the value in GPR 3, adds the value in GPR 1 to the product, and writes the computed result back in GPR 3 and in GPR 4. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100F8081). It also shows the state of the special register file after instruction execution.



The computed result 0x31FDD649E is 64-bit wide since a multiplication of two word-sized values gives a doubleword result which (updated with the addition of the GPR 1 value) is written back into GPR 3 and GPR 4, of which GPR 4 contains the higher part in significance. In other words, the result is a concatenation of GPR 4 and GPR 3.

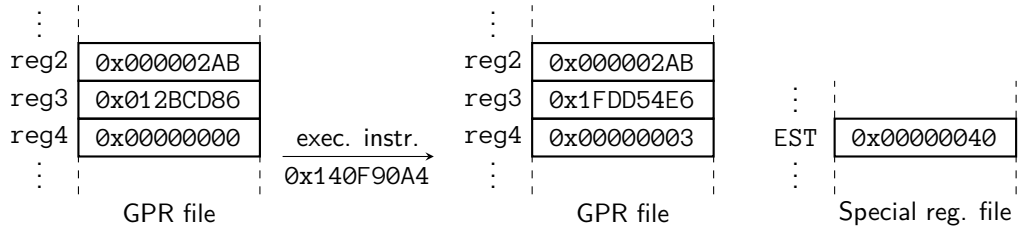
In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

Example 55: *Multiply-add immediate*

Instruction in binary format: 00010100 000011 1110 01 000010 100100
Instruction in hexadecimal format: 0x140F90A4
Fields:

MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)
I8HI	01	(0x1)
I8LO	100100	(0x24)

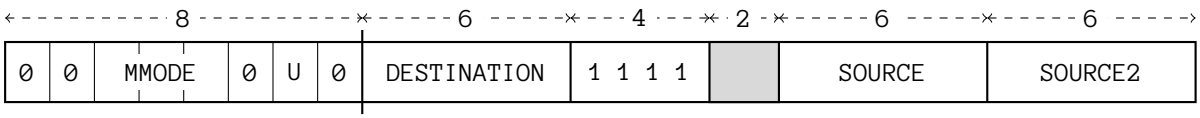
The instruction multiplies the value in GPR 2 to the value in GPR 3, adds the 8-bit wide (sign-extended) IMMEDIATE8 value to the product, and writes the computed result back in GPR 3 and in GPR 4. The concatenation of I8HI and I8LO gives IMMEDIATE8 = 0x64. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x140F90A4). It also shows the state of the special register file after instruction execution.



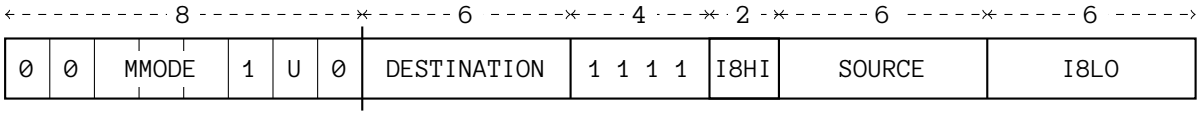
The computed result 0x31FDD54E6 is 64-bit wide since a multiplication of two word-sized values gives a doubleword result which (updated with the addition of the GPR 1 value) is written back into GPR 3 and GPR 4, of which GPR 4 contains the higher part in significance. In other words, the result is a concatenation of GPR 4 and GPR 3.

In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

6.1.33 MSU – Multiply-subtract



(a) Multiply-subtract register



(b) Multiply-subtract immediate

Fig. 64: *Multiply-subtract (MSU) instructions*

The MSU instruction specifies *fused multiplication-subtraction*, i.e., multiplication of two operands and subtraction of a third operand from the product. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the fused multiplication-subtraction. The second operand is also in a GPR. The third operand is either in a GPR or is an immediate value specified by the instruction itself.

Fields

MMODE	Specifies the integer machine mode according to Table 1.
U	0: **signed** operation (including sign-extended immediate for *Multiply-subtract immediate*). 1: **unsigned** operation (including zero-extended immediate for *Multiply-subtract immediate*). See Subsection 3.2.1.
DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.
SOURCE2	Specifies the number of the GPR containing the third operand.
I8HI	The two MSBs of the 8-bit IMMEDIATE8.
I8LO	The six LSBs of the 8-bit IMMEDIATE8.

Execution

Table 46 summarizes the execution of MSU instructions.

Table 46: *Execution of MSU instructions*

Multiply-subtract register
REG[DESTINATION] ← REG[DESTINATION] × REG[SOURCE] − REG[SOURCE2]
Multiply-subtract immediate
REG[DESTINATION] ← REG[DESTINATION] × REG[SOURCE] − IMMEDIATE8

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. However, the MSU instruction always returns a result which is twice the width of the input operands specified by MMODE. Thus, depending on the machine mode and the GPR width, subsequent

GPRs may be written according to the property of circularity (see Subsection 2.2.3) in little-endian ordering. For example, if both the GPR width and the MMODE is 32 bits, then the result is 64-bit wide which will be written in two subsequent GPRs, i.e., the lower part in the GPR specified by the DESTINATION field, and the upper part in the subsequent GPR. On the other side, if the GPR width is 32 bits and MMODE is 16 bits, the result is 32-bit wide, and will be written in a single register specified by DESTINATION.

EXECUTION STATUS — Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

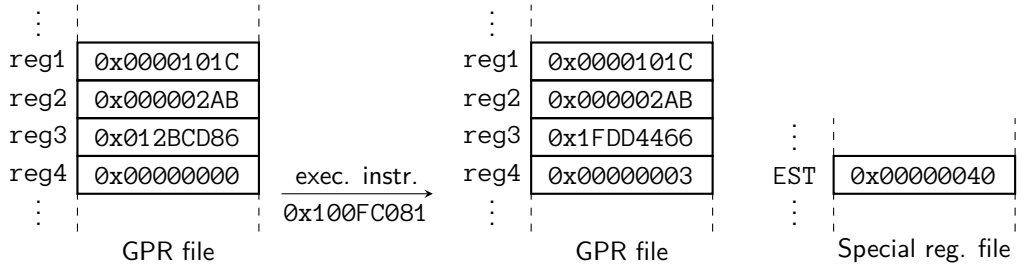
Examples

Example 56: *Multiply-subtract register*

Instruction in binary format: 00010000 000011 1111 00 000010 000001
Instruction in hexadecimal format: 0x100FC081
Fields:

MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)
SOURCE2	000001	(reg1)

The instruction multiplies the value in GPR 2 to the value in GPR 3, subtracts the value in GPR 1 from the product, and writes the computed result back in GPR 3 and in GPR 4. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x100FC081). It also shows the state of the special register file after instruction execution.



The computed result 0x31FDD4466 is 64-bit wide since a multiplication of two word-sized values gives a doubleword result which (updated with the subtraction of the GPR 1 value) is written back into GPR 3 and GPR 4, of which GPR 4 contains the higher part in significance. In other words, the result is a concatenation of GPR 4 and GPR 3.

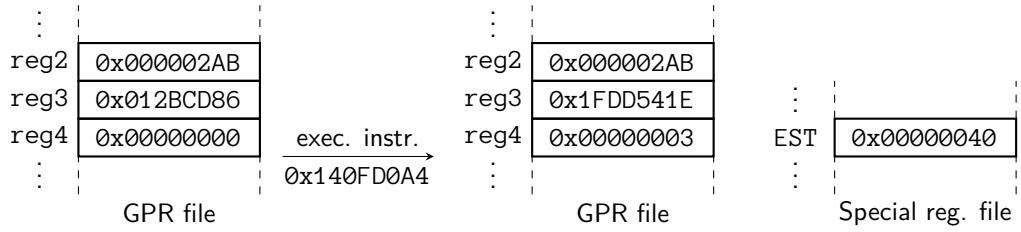
In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

Example 57: *Multiply-subtract immediate*

Instruction in binary format: 00010100 000011 1111 01 000010 100100
Instruction in hexadecimal format: 0x140FD0A4
Fields:

MMODE	010	(word)
U	0	(signed)
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)
I8HI	01	(0x1)
I8LO	100100	(0x24)

The instruction multiplies the value in GPR 2 to the value in GPR 3, subtracts the 8-bit wide (sign-extended) IMMEDIATE8 value from the product, and writes the computed result back in GPR 3 and in GPR 4. The concatenation of I8HI and I8LO gives IMMEDIATE8 = 0x64. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x140FD0A4). It also shows the state of the special register file after instruction execution.



The computed result 0x31FDD541E is 64-bit wide since a multiplication of two word-sized values gives a doubleword result which (updated with the subtraction of the GPR 1 value) is written back into GPR 3 and GPR 4, of which GPR 4 contains the higher part in significance. In other words, the result is a concatenation of GPR 4 and GPR 3.

In the EXECUTION STATUS register (EST), only the GREATER THAN flag is set (see Subsection 5.2).

6.1.34 FMAD – FP Multiply-add

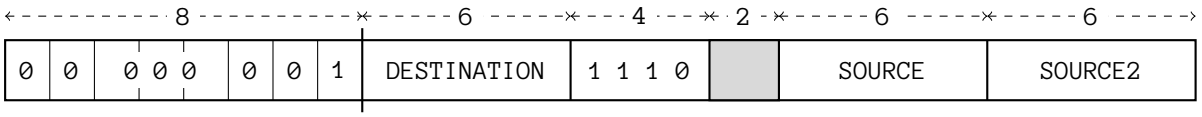


Fig. 65: *FP Multiply-add (FMAD) instructions*

The FMAD instruction specifies *FP fused multiplication-addition*, i.e., multiplication of two operands and addition of a third operand to the product. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the fused multiplication-addition. The second and the third operands are also in GPRs.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.
SOURCE2	Specifies the number of the GPR containing the third operand.

Execution

Table 47 summarizes the execution of FMAD instructions.

Table 47: *Execution of FMAD instructions*

FP Multiply-add
REG[DESTINATION] ← REG[DESTINATION] × REG[SOURCE] + REG[SOURCE2]

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when (at least) one of the operands is a signaling NaN, or when the result is NaN. It is also raised when the operands imply addition of infinities with opposite signs, e.g., positive infinity plus negative infinity, or when one multiplication operand is zero and the other is infinity. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.

FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.
FP UNDERFLOW	It is raised when the result is a tiny non-zero number. See Subsection 4.2.14.
FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

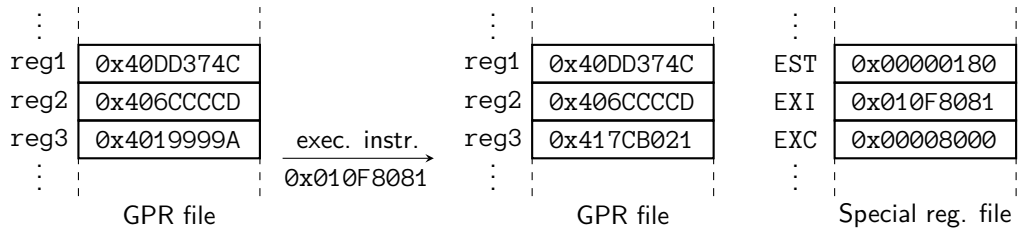
Examples

Example 58: *FP Multiply-add*

Instruction in binary format: 00000001 000011 1110 00 000010 000001
Instruction in hexadecimal format: 0x010F8081
Fields:

DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)
SOURCE2	000001	(reg1)

The instruction multiplies the value in GPR 2 to the value in GPR 3, adds the product to the value in GPR 1, and writes the computed result back in GPR 3. The instruction assumes that the input operands are in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x010F8081). It also shows the state of the special register file after instruction execution.



In the EXECUTION STATUS register (EST), the LESS THAN and INEXACT flags are set (see Subsection 5.2).

Because of the raised FP INEXACT RESULT exception, the EXCEPTION INSTRUCTION register (EXI) is written with the instruction code, and the FP INEXACT RESULT bit in the EXCEPTION REGISTER (EXC) is set (assuming that before executing the instruction the EXC register was zero). However, it is also assumed that the FP INEXACT RESULT exception is impotent. Therefore, the result is written back to the DESTINATION GPR 3.

6.1.35 FMSU – FP Multiply-subtract

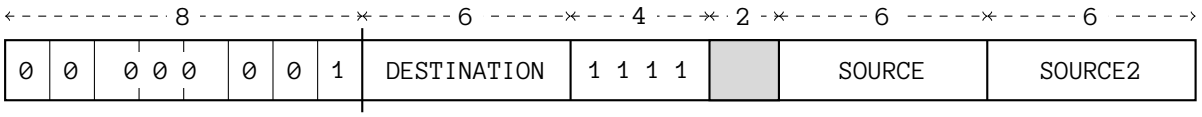


Fig. 66: *FP Multiply-subtract (FMSU) instructions*

The FMSU instruction specifies *FP fused multiplication-subtraction*, i.e., multiplication of two operands and subtraction of a third operand from the product. The first operand resides in a GPR in which the result will be written back, i.e., the first operand will be overwritten with the result of the fused multiplication-subtraction. The second and the third operands are also in GPRs.

Fields

DESTINATION	Specifies the number of the GPR containing the first operand, which is also the destination GPR in which the computed result is written back after instruction completion (the value of the operand is overwritten).
SOURCE	Specifies the number of the GPR containing the second operand.
SOURCE2	Specifies the number of the GPR containing the third operand.

Execution

Table 48 summarizes the execution of FMSU instructions.

Table 48: *Execution of FMSU instructions*

FP Multiply-subtract
REG[DESTINATION] ← REG[DESTINATION] × REG[SOURCE] – REG[SOURCE2]

Changes

Destination GPR	Changes the destination GPR specified by the DESTINATION field. In this FP instruction the MMODE field is always zero since the operation is performed using the widest supported FP format, i.e., FP machine mode. If the FP width is shorter than the GPR width, only the corresponding lower bits of the GPR are changed (see Subsection 2.2.2). On the other side, if the FP width is wider than the GPR width, the register circularity applies (see Subsection 2.2.3).
EXECUTION STATUS	Changes the EXECUTION STATUS special register (see Subsection 5.2).

Exceptions

FP INVALID OPERATION	It is raised when (at least) one of the operands is a signaling NaN, or when the result is NaN. It is also raised when the operands imply subtraction of infinities with opposite signs, e.g., positive infinity minus positive infinity, when one multiplication operand is zero and the other is infinity. See Subsection 4.2.10.
FP DENORMALIZED OPERAND	It is raised when one or more operands is a denormalized FP number. However, if the operands are not denormalized but the operation produces a denormalized result, this exception is not raised. See Subsection 4.2.11.

FP OVERFLOW	It is raised when the result of the operation exceeds the largest representable finite number of the destination format. See Subsection 4.2.13.
FP UNDERFLOW	It is raised when the result is a tiny non-zero number. See Subsection 4.2.14.
FP INEXACT RESULT	It is raised when the rounded result is not exact. Furthermore, it is also raised when the result overflows and the FP OVERFLOW exception is impotent, or, when an inexact result underflows and the FP UNDERFLOW is impotent. See Subsection 4.2.15.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

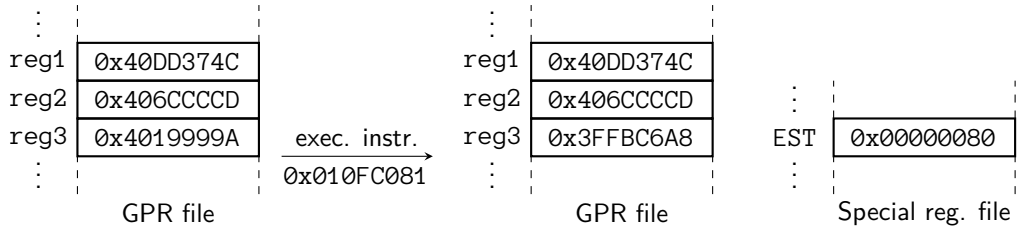
Examples

Example 59: *FP Multiply-subtract*

Instruction in binary format: 00000001 000011 1111 00 000010 000001
Instruction in hexadecimal format: 0x010FC081
Fields:

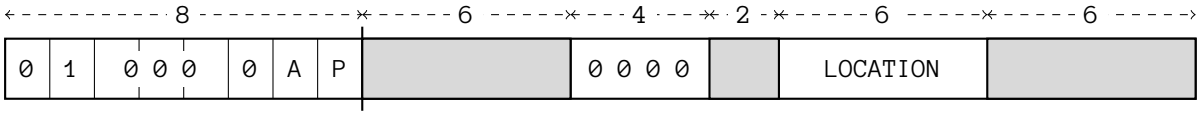
DESTINATION	000011	(reg3)
SOURCE	000010	(reg2)
SOURCE2	000001	(reg1)

The instruction multiplies the value in GPR 2 to the value in GPR 3, subtracts the value in GPR 1 from the product, and writes the computed result back in GPR 3. The instruction assumes that the input operands are in FP format and outputs the result also in FP format. In this example, the 32-bit FP format according to the IEEE Std 754-2008 standard is used [3]. The following illustration shows an example state of a 32-bit wide GPR file before and after execution of the instruction (0x010FC081). It also shows the state of the special register file after instruction execution.

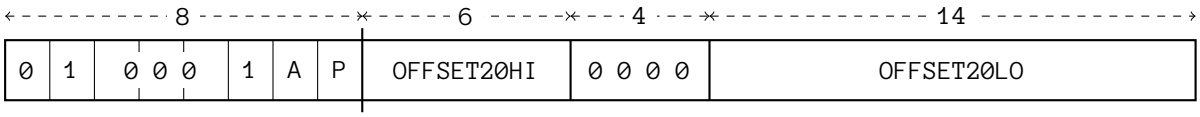


In the EXECUTION STATUS register (EST), only the LESS THAN flag is set (see Subsection 5.2).

6.1.36 JMP – Jump



(a) Jump according to register



(b) Jump according to offset

Fig. 67: *Jump (JMP) instructions*

The JMP instruction specifies unconditional program transfer to a location specified by a GPR or by an implicitly specified offset.

Fields

A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET20HI	The six MSBs of the 20-bit signed value OFFSET20.
OFFSET20LO	The 14 LSBs of the 20-bit signed value OFFSET20.

Execution

Table 49 summarizes the execution of JMP instructions.

Table 49: *Execution of JMP instructions*

Jump according to register
if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION] **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION] **if**(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
Jump according to offset
if(A == 1) INSTRUCTION COUNTER ← OFFSET20 **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET20 **if**(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1

Changes

INSTRUCTION COUNTER	It is loaded with the given/computed instruction address.
CALL RETURN POINTER	If P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

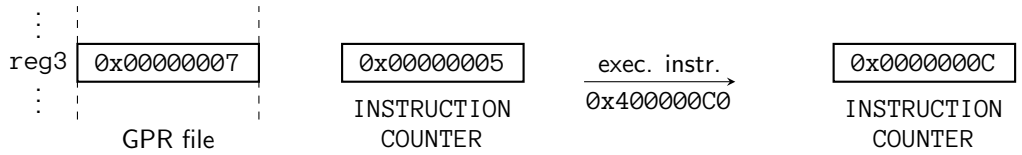
Examples

Example 60: *Jump relative according to register*

Instruction in binary format: `01000000 000000 0000 00 000011 000000`
Instruction in hexadecimal format: `0x400000C0`
Fields:

A	0	(relative transfer)
P	0	(non-procedural transfer)
LOCATION	000011	(reg3)

The instruction adds the value in GPR 3 to the INSTRUCTION COUNTER. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x400000C0).



That is, if the jump instruction is the 5-th instruction (at address 0x14), instructions 6 to 11 will be jumped, and the next instruction to be executed is the 12-th (at address 0x30).

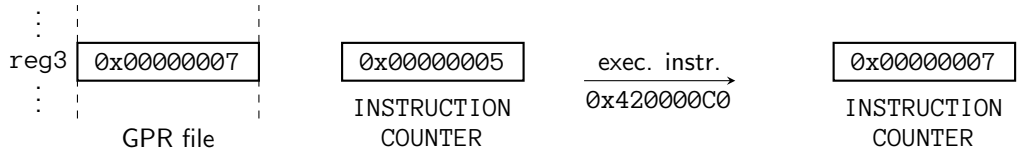
If the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

Example 61: *Jump absolute according to register*

Instruction in binary format: `01000010 000000 0000 00 000011 000000`
Instruction in hexadecimal format: `0x420000C0`
Fields:

A	1	(absolute transfer)
P	0	(non-procedural transfer)
LOCATION	000011	(reg3)

The value of GPR 3 is written to the INSTRUCTION COUNTER. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x420000C0).



That is, if the jump instruction is the 5-th instruction (at address 0x14), instruction 6 will be jumped, and the next instruction to be executed is the 7-th (at address 0x1C).

If the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

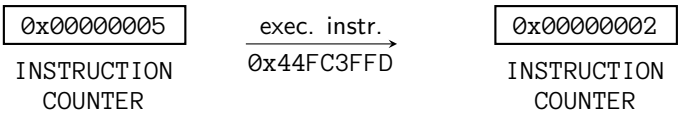
Example 62: *Jump relative according to offset*

Instruction in binary format: `01000100 111111 0000 1111111111101`
Instruction in hexadecimal format: `0x44FC3FFD`

Fields:

A	0	(relative transfer)
P	0	(non-procedural transfer)
OFFSET20HI	111111	(0x3F)
OFFSET20LO	11111111111101	(0x3FFD)

The instruction adds the signed OFFSET20 to the INSTRUCTION COUNTER. The concatenation of OFFSET20HI and OFFSET20LO gives OFFSET20 = 0xFFFFD. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x44FC3FFD).



That is, if the jump instruction is the 5-th instruction (at address 0x14), the program execution will be transferred back to instruction 2 (due to the negative OFFSET20 of -3), jumping instructions 4 and 3. Thus, the next instruction to be executed after the jump is the 2-nd instruction (at address 0x8).

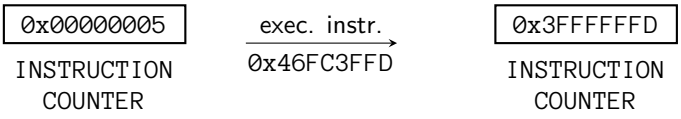
If the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

Example 63: *Jump absolute according to offset*

Instruction in binary format: 01000110 111111 0000 11111111111101
Instruction in hexadecimal format: 0x46FC3FFD
Fields:

A	1	(absolute transfer)
P	0	(non-procedural transfer)
OFFSET20HI	111111	(0x3F)
OFFSET20LO	11111111111101	(0x3FFD)

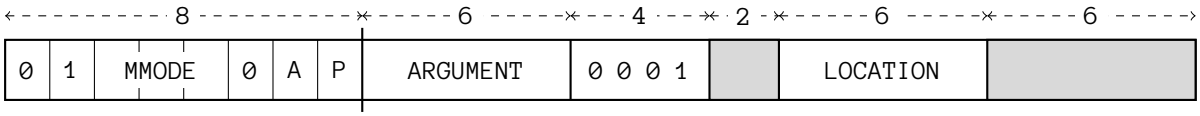
The signed OFFSET20 value is written to the INSTRUCTION COUNTER. The concatenation of OFFSET20HI and OFFSET20LO gives OFFSET20 = 0xFFFFD. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x46FC3FFD).



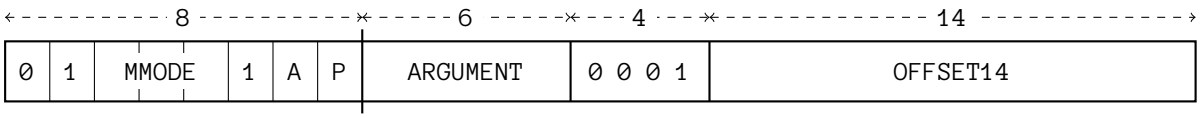
That is, the program will be transferred at instruction 0x3FFFFFFD (obtained by sign-extending OFFSET20 to the 30-bit width of the INSTRUCTION COUNTER) which resides at address 0xFFFFFFF4.

If the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.37 BZ – Branch if Zero



(a) Branch if Zero according to register



(b) Branch if Zero according to offset

Fig. 68: *Branch if Zero (BZ) instructions*

The BZ instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if all the bits of the argument GPR are 0 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 50 summarizes the execution of BZ instructions.

Table 50: *Execution of BZ instructions*

Branch if Zero according to register

if(REG[ARGUMENT][2e(MMODE+3)−1:0] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if Zero according to offset

if(REG[ARGUMENT][2e(MMODE+3)−1:0] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

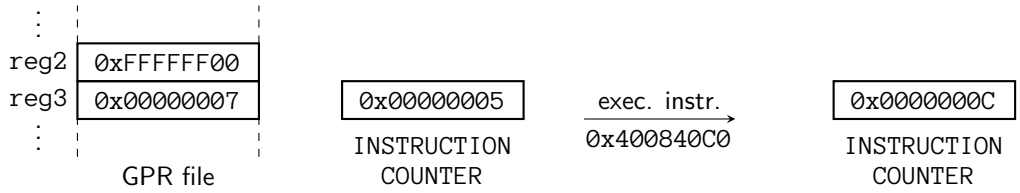
Examples

Example 64: *Branch if Zero according to register*

Instruction in binary format: 01000000 000010 0001 00 000011 000000
Instruction in hexadecimal format: 0x400840C0
Fields:

MMODE	000	(byte)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the 8-bit byte of GPR 2 is zero, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x400840C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the 8-bit byte of GPR 2 is zero" is met and the branch is taken, which means that the next instruction to be executed is the 12-th (at address 0x30), skipping instructions 6 to 11.

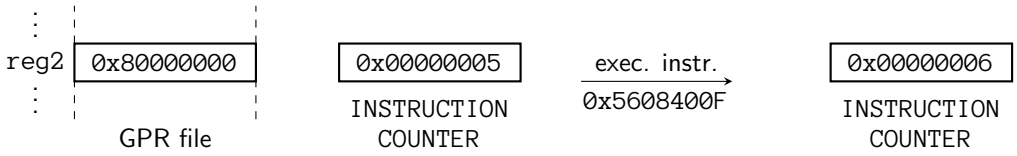
If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

Example 65: *Branch if Zero according to offset*

Instruction in binary format: 01010110 000010 0001 00000000001111
Instruction in hexadecimal format: 0x5608400F
Fields:

MMODE	010	(word)
A	1	(absolute transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000001111	(0xF)

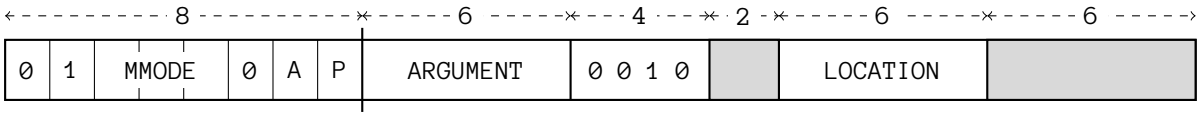
If the 32-bit word of GPR 2 is zero, the signed OFFSET14 value is written to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x5608400F).



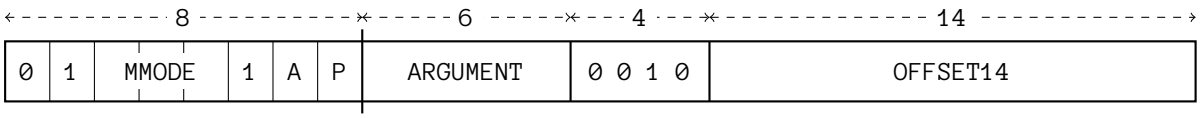
Thus, the branch instruction is the 5-th instruction (at address `0x14`). The condition "branch if the 32-bit word of GPR 2 is zero" is not met and the branch is not taken, which means that the next instruction to be executed is the 6-th (at address `0x18`).

If the branch is taken and if the transfer is procedural, i.e., the `P` bit is `1` instead of `0`, the `CALL RETURN POINTER` will be written with the value of the `INSTRUCTION COUNTER` before instruction execution, incremented by one (`0x00000006`).

6.1.38 BNZ – Branch if Not Zero



(a) Branch if Not Zero according to register



(b) Branch if Not Zero according to offset

Fig. 69: *Branch if Not Zero (BNZ) instructions*

The BNZ instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if not all the bits of the argument GPR are 0 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 51 summarizes the execution of BNZ instructions.

Table 51: *Execution of BNZ instructions*

Branch if Not Zero according to register

if(REG[ARGUMENT][2e(MMODE+3)−1:0] != 0) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if Not Zero according to offset

if(REG[ARGUMENT][2e(MMODE+3)−1:0] != 0) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

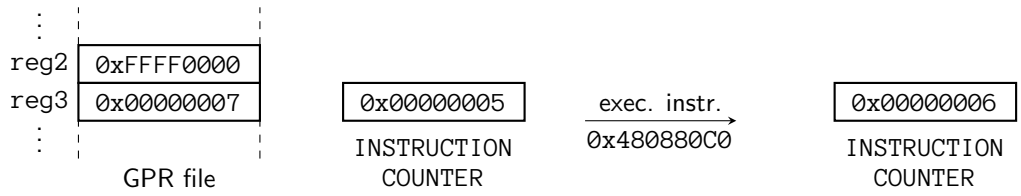
Examples

Example 66: *Branch if Not Zero according to register*

Instruction in binary format: 01001000 000010 0010 00 000011 000000
Instruction in hexadecimal format: 0x480880C0
Fields:

MMODE	001	(halfword)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the 16-bit halfword of GPR 2 is not zero, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x480880C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the 16-bit halfword of GPR 2 is not zero" is not met and the branch is not taken, which means that the next instruction to be executed is the 6-th (at address 0x18).

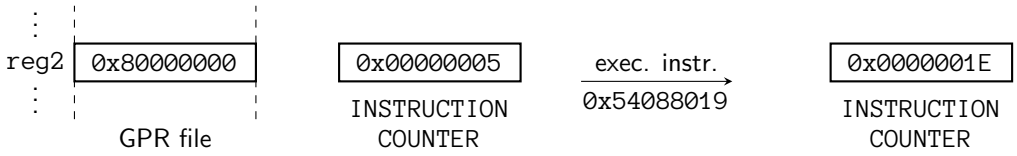
If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

Example 67: *Branch if Not Zero according to offset*

Instruction in binary format: 01010100 000010 0010 00000000011001
Instruction in hexadecimal format: 0x54088019
Fields:

MMODE	010	(word)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

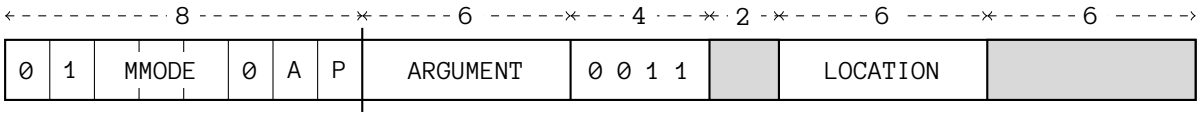
If the 32-bit word of GPR 2 is not zero, the signed OFFSET14 value is added to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x54088019).



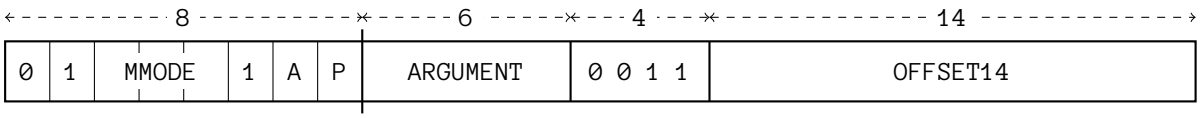
Thus, the branch instruction is the 5-th instruction (at address `0x14`). The condition "branch if the 32-bit word of GPR 2 is not zero" is met and the branch is taken, which means that the next instruction to be executed is the 30-th (at address `0x78`), skipping instructions 6 to 29.

If the branch is taken and if the transfer is procedural, i.e., the `P` bit is `1` instead of `0`, the `CALL RETURN POINTER` will be written with the value of the `INSTRUCTION COUNTER` before instruction execution, incremented by one (`0x00000006`).

6.1.39 BM – Branch if MSB



(a) Branch if MSB according to register



(b) Branch if MSB according to offset

Fig. 70: *Branch if MSB (BM) instructions*

The BM instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if the MSB of the argument GPR is 1 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 52 summarizes the execution of BM instructions.

Table 52: *Execution of BM instructions*

Branch if MSB according to register

if(REG[ARGUMENT][2e(MMODE+3)−1] == 1) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if MSB according to offset

if(REG[ARGUMENT][2e(MMODE+3)−1] == 1) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

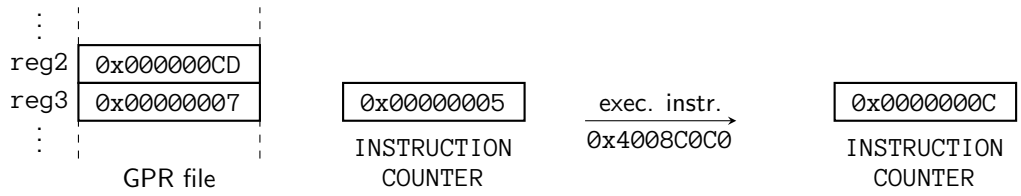
Examples

Example 68: *Branch if MSB according to register*

Instruction in binary format: 01000000 000010 0011 00 000011 000000
Instruction in hexadecimal format: 0x4008C0C0
Fields:

MMODE	000	(byte)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the MSB of the 8-bit byte of GPR 2 is one, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x4008C0C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the MSB of the 8-bit byte of GPR 2 is one" is met and the branch is taken, which means that the next instruction to be executed is the 12-th (at address 0x30), skipping instructions 6 to 11.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

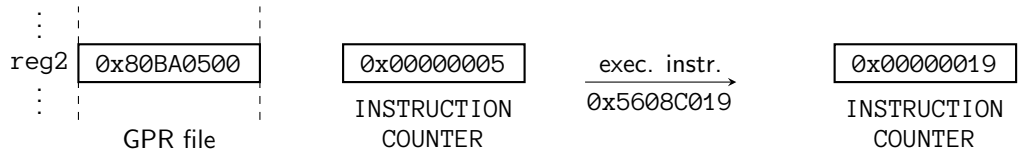
Example 69: *Branch if MSB according to offset*

Instruction in binary format: 01010110 000010 0011 00000000011001
Instruction in hexadecimal format: 0x5608C019
Fields:

MMODE	010	(word)
A	1	(absolute transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

If the MSB of the 32-bit word of GPR 2 is one, the signed OFFSET14 value is written to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value

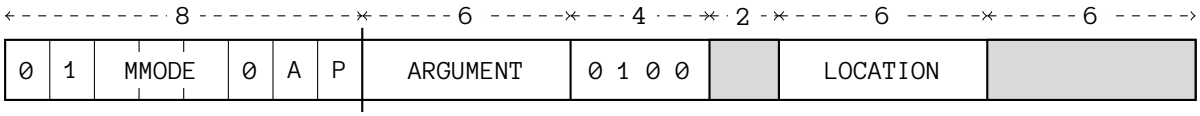
of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x5608C019).



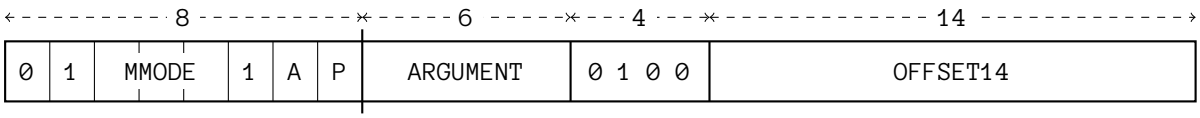
Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the MSB of the 32-bit word of GPR 2 is one" is met and the branch is taken, which means that the next instruction to be executed is the 25-th (at address 0x64), skipping instructions 6 to 24.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.40 BMZ – Branch if MSB or Zero



(a) Branch if MSB or Zero according to register



(b) Branch if MSB or Zero according to offset

Fig. 71: *Branch if MSB or Zero (BMZ) instructions*

The BMZ instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if the MSB of the argument GPR is 1, or, if all the bits of the argument GPR are 0 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 53 summarizes the execution of BMZ instructions.

Table 53: *Execution of BMZ instructions*

Branch if MSB or Zero according to register

if(REG[ARGUMENT][2e(MMODE+3)−1] == 1 **or** REG[ARGUMENT][2e(MMODE+3)−1:0] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if MSB or Zero according to offset

if(REG[ARGUMENT][2e(MMODE+3)−1] == 1 **or** REG[ARGUMENT][2e(MMODE+3)−1:0] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

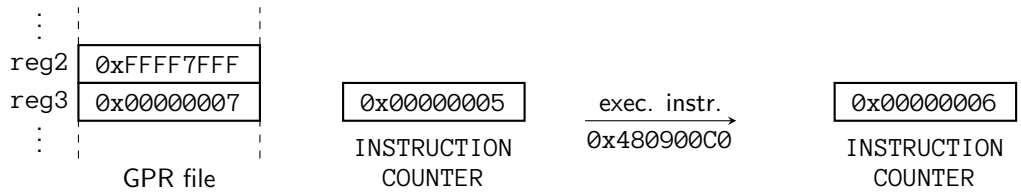
Examples

Example 70: *Branch if MSB or Zero according to register*

Instruction in binary format: 01001000 000010 0100 00 000011 000000
Instruction in hexadecimal format: 0x480900C0
Fields:

MMODE	001	(halfword)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the MSB of the 16-bit halfword of GPR 2 is one or the 16-bit halfword is zero, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x480900C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the MSB of the 16-bit halfword of GPR 2 is one or the 16-bit halfword is zero" is not met and the branch is not taken, which means that the next instruction to be executed is the 6-th (at address 0x18).

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

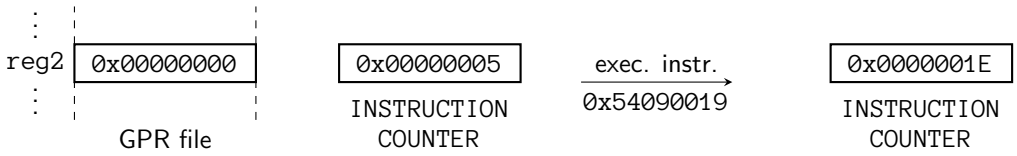
Example 71: *Branch if MSB or Zero according to offset*

Instruction in binary format: 01010100 000010 0100 00000000011001
Instruction in hexadecimal format: 0x54090019
Fields:

MMODE	010	(word)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

If the MSB of the 32-bit word of GPR 2 is one or the 32-bit word is zero, the signed OFFSET14 value is added to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR

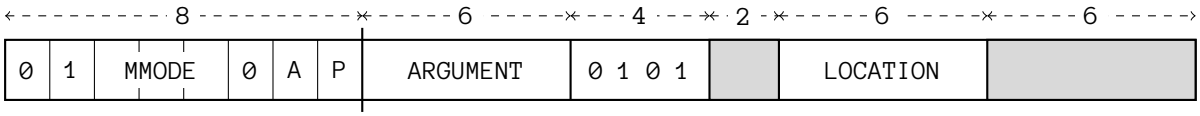
file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x54090019).



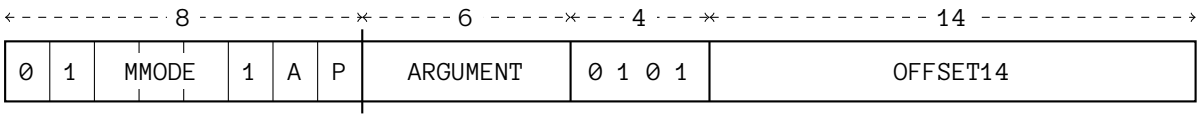
Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the MSB of the 32-bit word of GPR 2 is one or the 32-bit word is zero" is met and the branch is taken, which means that the next instruction to be executed is the 30-th (at address 0x78), skipping instructions 6 to 29.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.41 BNM – Branch if Not MSB



(a) Branch if Not MSB according to register



(b) Branch if Not MSB according to offset

Fig. 72: *Branch if Not MSB (BNM) instructions*

The BNM instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if the MSB of the argument GPR is 0 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 54 summarizes the execution of BNM instructions.

Table 54: *Execution of BNM instructions*

Branch if Not MSB according to register

if(REG[ARGUMENT][2e(MMODE+3)−1] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if Not MSB according to offset

if(REG[ARGUMENT][2e(MMODE+3)−1] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

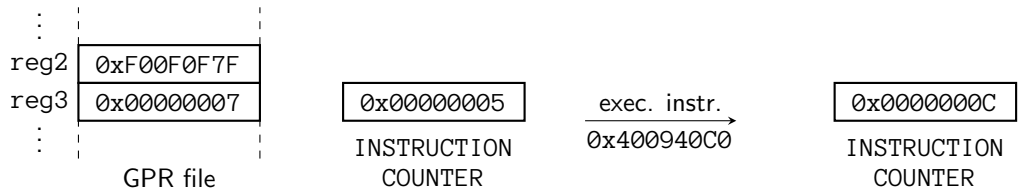
Examples

Example 72: *Branch if Not MSB according to register*

Instruction in binary format: 01000000 000010 0101 00 000011 000000
Instruction in hexadecimal format: 0x400940C0
Fields:

MMODE	000	(byte)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the MSB of the 8-bit byte of GPR 2 is zero, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x400940C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the MSB of the 8-bit byte of GPR 2 is zero" is met and the branch is taken, which means that the next instruction to be executed is the 12-th (at address 0x30), skipping instructions 6 to 11.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

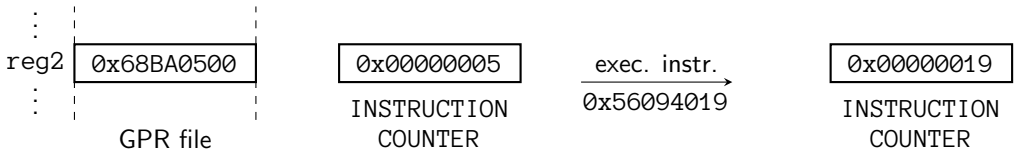
Example 73: *Branch if Not MSB according to offset*

Instruction in binary format: 01010110 000010 0101 00000000011001
Instruction in hexadecimal format: 0x56094019
Fields:

MMODE	010	(word)
A	1	(absolute transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

If the MSB of the 32-bit word of GPR 2 is zero, the signed OFFSET14 value is written to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value

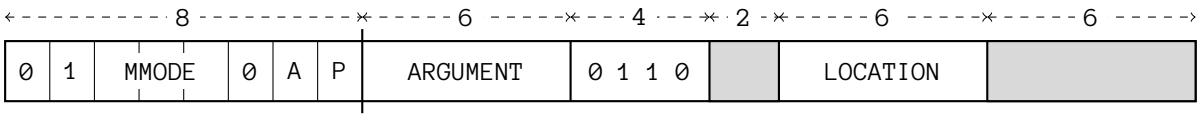
of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x56094019).



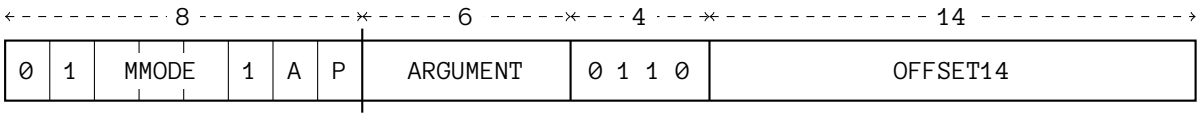
Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the MSB of the 32-bit word of GPR 2 is zero" is met and the branch is taken, which means that the next instruction to be executed is the 25-th (at address 0x64), skipping instructions 6 to 24.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.42 BNMO – Branch if Not MSB or all Ones



(a) Branch if Not MSB or all Ones according to register



(b) Branch if Not MSB or all Ones according to offset

Fig. 73: *Branch if Not MSB or all Ones (BNMO) instructions*

The BNMO instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if the MSB of the argument GPR is 0, or, if all the bits of the argument GPR are 1 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 55 summarizes the execution of BNMO instructions.

Table 55: *Execution of BNMO instructions*

Branch if Not MSB or all Ones according to register

if(REG[ARGUMENT][2e(MMODE+3)−1] == 0 **or** REG[ARGUMENT][2e(MMODE+3)−1:0] == −1) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if Not MSB or all Ones according to offset

if(REG[ARGUMENT][2e(MMODE+3)−1] == 0 **or** REG[ARGUMENT][2e(MMODE+3)−1:0] == −1) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

In the binary representation of the decimal –1 negative integer, all the bits are one (for any width), which is used as a shorthand notation in the condition evaluation in Table 55.

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

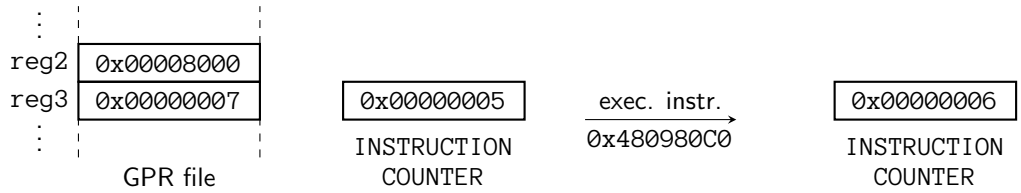
Examples

Example 74: *Branch if Not MSB or all Ones according to register*

Instruction in binary format: 01001000 000010 0110 00 000011 000000
Instruction in hexadecimal format: 0x480980C0
Fields:

MMODE	001	(halfword)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the MSB of the 16-bit halfword of GPR 2 is zero or all bits of the 16-bit halfword are ones, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x480980C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the MSB of the 16-bit halfword of GPR 2 is zero or all bits of the 16-bit halfword are ones" is not met and the branch is not taken, which means that the next instruction to be executed is the 6-th (at address 0x18).

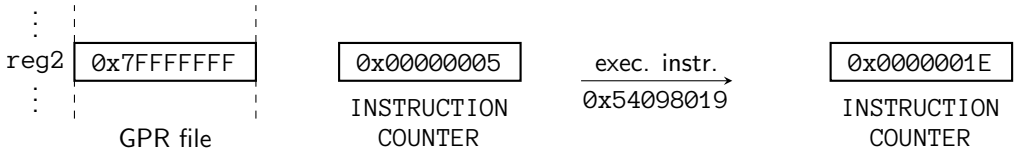
If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

Example 75: *Branch if Not MSB or all Ones according to offset*

Instruction in binary format: 01010100 000010 0110 00000000011001
Instruction in hexadecimal format: 0x54098019
Fields:

MMODE	010	(word)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

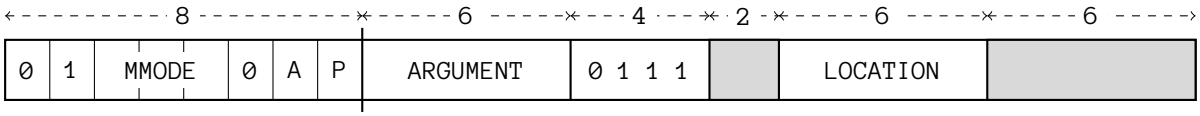
If the MSB of the 32-bit word of GPR 2 is zero or all bits of the 32-bit word are ones, the signed OFFSET14 value is added to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x54098019).



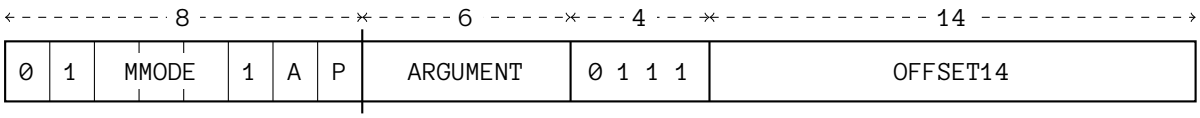
Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the MSB of the 32-bit word of GPR 2 is zero or all bits of the 32-bit word are ones" is met and the branch is taken, which means that the next instruction to be executed is the 30-th (at address 0x78), skipping instructions 6 to 29.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.43 BL – Branch if LSB



(a) Branch if LSB according to register



(b) Branch if LSB according to offset

Fig. 74: *Branch if LSB (BL) instructions*

The BL instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if the LSB of the argument GPR is 1 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 56 summarizes the execution of BL instructions.

Table 56: *Execution of BL instructions*

Branch if LSB according to register

if(REG[ARGUMENT][0] == 1) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if LSB according to offset

if(REG[ARGUMENT][0] == 1) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

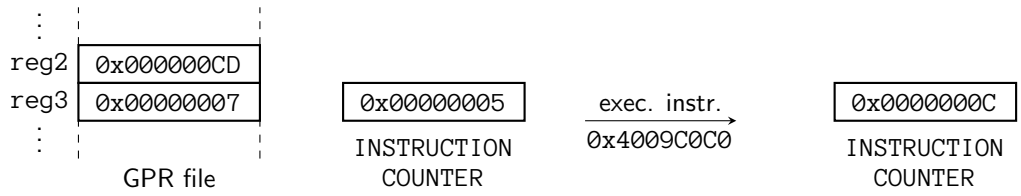
Examples

Example 76: *Branch if LSB according to register*

Instruction in binary format: 01000000 000010 0111 00 000011 000000
Instruction in hexadecimal format: 0x4009C0C0
Fields:

MMODE	000	(byte)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the LSB of the 8-bit byte of GPR 2 is one, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x4009C0C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the LSB of the 8-bit byte of GPR 2 is one" is met and the branch is taken, which means that the next instruction to be executed is the 12-th (at address 0x30), skipping instructions 6 to 11.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

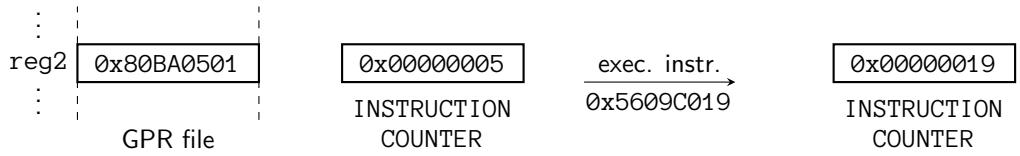
Example 77: *Branch if LSB according to offset*

Instruction in binary format: 01010110 000010 0111 00000000011001
Instruction in hexadecimal format: 0x5609C019
Fields:

MMODE	010	(word)
A	1	(absolute transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

If the LSB of the 32-bit word of GPR 2 is one, the signed OFFSET14 value is written to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value

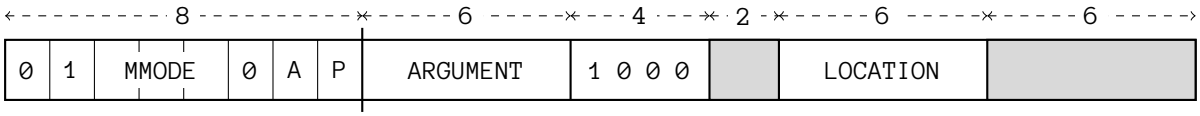
of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x5609C019).



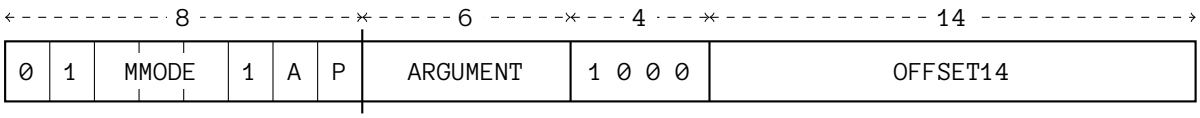
Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the LSB of the 32-bit word of GPR 2 is one" is met and the branch is taken, which means that the next instruction to be executed is the 25-th (at address 0x64), skipping instructions 6 to 24.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.44 BLZ – Branch if LSB or Zero



(a) Branch if LSB or Zero according to register



(b) Branch if LSB or Zero according to offset

Fig. 75: *Branch if LSB or Zero (BLZ) instructions*

The BLZ instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if the LSB of the argument GPR is 1, or, if all the bits of the argument GPR are 0 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 57 summarizes the execution of BLZ instructions.

Table 57: *Execution of BLZ instructions*

Branch if LSB or Zero according to register

if(REG[ARGUMENT][0] == 1 **or** REG[ARGUMENT][2e(MMODE+3)−1:0] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if LSB or Zero according to offset

if(REG[ARGUMENT][0] == 1 **or** REG[ARGUMENT][2e(MMODE+3)−1:0] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

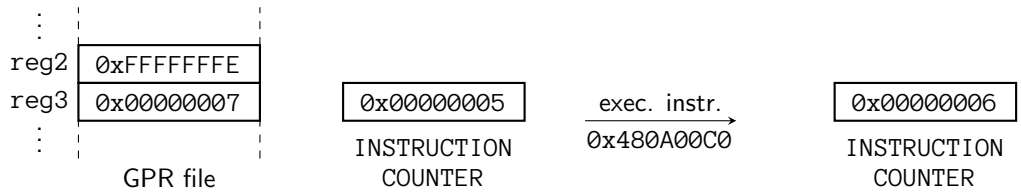
Examples

Example 78: *Branch if LSB or Zero according to register*

Instruction in binary format: 01001000 000010 1000 00 000011 000000
Instruction in hexadecimal format: 0x480A00C0
Fields:

MMODE	001	(halfword)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the LSB of the 16-bit halfword of GPR 2 is one or the 16-bit halfword is zero, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x480A00C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the LSB of the 16-bit halfword of GPR 2 is one or the 16-bit halfword is zero" is not met and the branch is not taken, which means that the next instruction to be executed is the 6-th (at address 0x18).

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

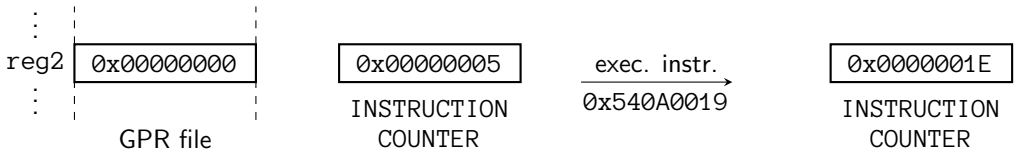
Example 79: *Branch if LSB or Zero according to offset*

Instruction in binary format: 01010100 000010 1000 00000000011001
Instruction in hexadecimal format: 0x540A0019
Fields:

MMODE	010	(word)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

If the LSB of the 32-bit word of GPR 2 is one or the 32-bit word is zero, the signed OFFSET14 value is added to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR

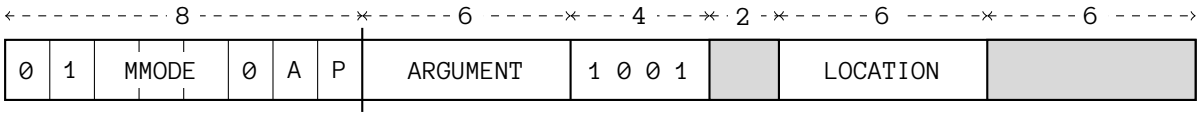
file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x540A0019).



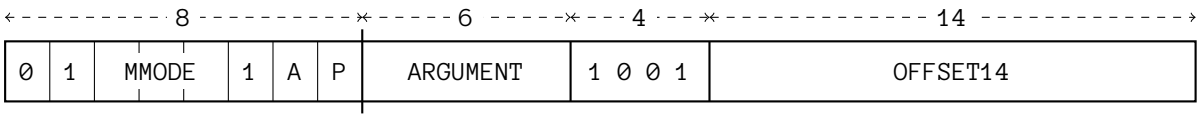
Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the LSB of the 32-bit word of GPR 2 is one or the 32-bit word is zero" is met and the branch is taken, which means that the next instruction to be executed is the 30-th (at address 0x78), skipping instructions 6 to 29.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.45 BNL – Branch if Not LSB



(a) Branch if Not LSB according to register



(b) Branch if Not LSB according to offset

Fig. 76: *Branch if Not LSB (BNL) instructions*

The BNL instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if the LSB of the argument GPR is 0 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 58 summarizes the execution of BNL instructions.

Table 58: *Execution of BNL instructions*

Branch if Not LSB according to register

if(REG[ARGUMENT][0] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if Not LSB according to offset

if(REG[ARGUMENT][0] == 0) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

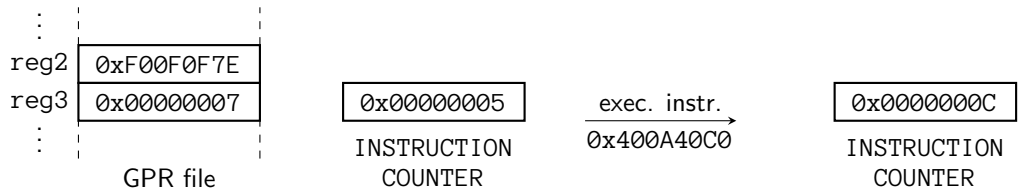
Examples

Example 80: *Branch if Not LSB according to register*

Instruction in binary format: 01000000 000010 1001 00 000011 000000
Instruction in hexadecimal format: 0x400A40C0
Fields:

MMODE	000	(byte)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the LSB of the 8-bit byte of GPR 2 is zero, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x400A40C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the LSB of the 8-bit byte of GPR 2 is zero" is met and the branch is taken, which means that the next instruction to be executed is the 12-th (at address 0x30), skipping instructions 6 to 11.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

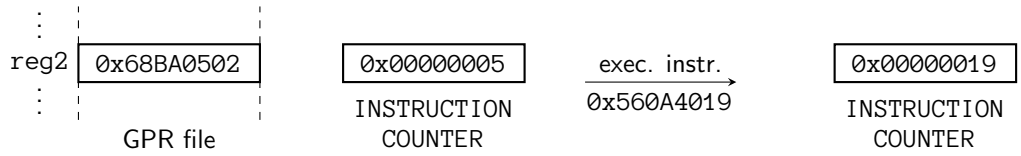
Example 81: *Branch if Not LSB according to offset*

Instruction in binary format: 01010110 000010 1001 00000000011001
Instruction in hexadecimal format: 0x560A4019
Fields:

MMODE	010	(word)
A	1	(absolute transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

If the LSB of the 32-bit word of GPR 2 is zero, the signed OFFSET14 value is written to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value

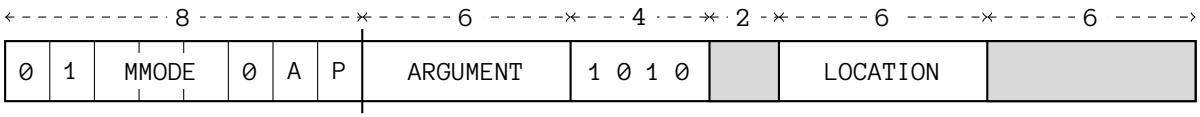
of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x560A4019).



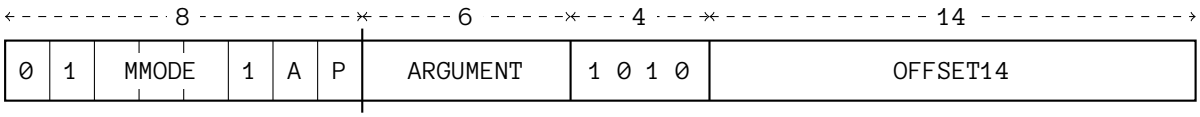
Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the LSB of the 32-bit word of GPR 2 is zero" is met and the branch is taken, which means that the next instruction to be executed is the 25-th (at address 0x64), skipping instructions 6 to 24.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.46 BNLO – Branch if Not LSB or all Ones



(a) Branch if Not LSB or all Ones according to register



(b) Branch if Not LSB or all Ones according to offset

Fig. 77: *Branch if Not LSB or all Ones (BNLO) instructions*

The BNLO instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if the LSB of the argument GPR is 0, or, if all the bits of the argument GPR are 1 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 59 summarizes the execution of BNLO instructions.

Table 59: *Execution of BNLO instructions*

Branch if Not LSB or all Ones according to register

if(REG[ARGUMENT][0] == 0 **or** REG[ARGUMENT][2e(MMODE+3)−1:0] == −1) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if Not LSB or all Ones according to offset

if(REG[ARGUMENT][0] == 0 **or** REG[ARGUMENT][2e(MMODE+3)−1:0] == −1) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

In the binary representation of the decimal –1 negative integer, all the bits are one (for any width), which is used as a shorthand notation in the condition evaluation in Table 59.

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

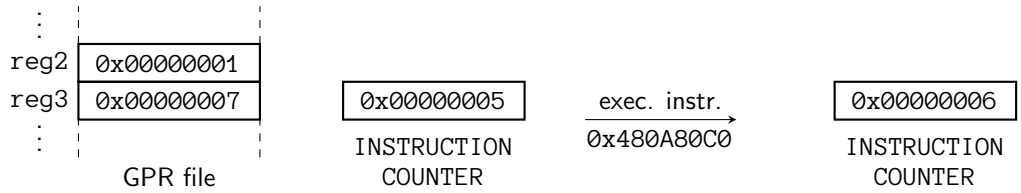
Examples

Example 82: *Branch if Not LSB or all Ones according to register*

Instruction in binary format: 01001000 000010 1010 00 000011 000000
Instruction in hexadecimal format: 0x480A80C0
Fields:

MMODE	001	(halfword)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If the LSB of the 16-bit halfword of GPR 2 is zero or all bits of the 16-bit halfword are ones, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x480A80C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the LSB of the 16-bit halfword of GPR 2 is zero or all bits of the 16-bit halfword are ones" is not met and the branch is not taken, which means that the next instruction to be executed is the 6-th (at address 0x18).

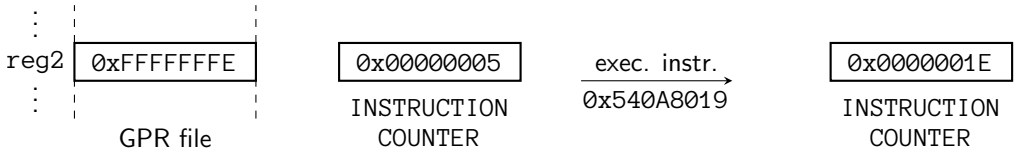
If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

Example 83: *Branch if Not LSB or all Ones according to offset*

Instruction in binary format: 01010100 000010 1010 00000000011001
Instruction in hexadecimal format: 0x540A8019
Fields:

MMODE	010	(word)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

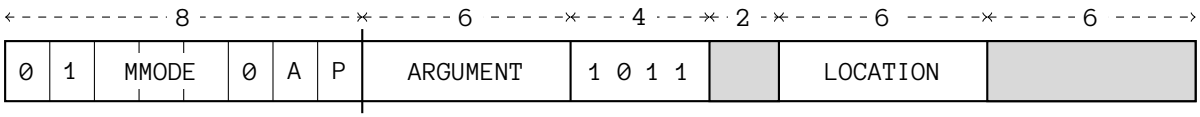
If the LSB of the 32-bit word of GPR 2 is zero or all bits of the 32-bit word are ones, the signed OFFSET14 value is added to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x540A8019).



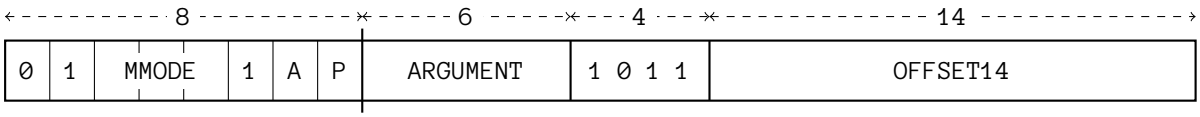
Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if the LSB of the 32-bit word of GPR 2 is zero or all bits of the 32-bit word are ones" is met and the branch is taken, which means that the next instruction to be executed is the 30-th (at address 0x78), skipping instructions 6 to 29.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.47 BO – Branch if all Ones



(a) Branch if all Ones according to register



(b) Branch if all Ones according to offset

Fig. 78: *Branch if all Ones (BO) instructions*

The BO instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if all the bits of the argument GPR are 1 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 60 summarizes the execution of BO instructions.

Table 60: *Execution of BO instructions*

Branch if all Ones according to register

if(REG[ARGUMENT][2e(MMODE+3)−1:0] == −1) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if all Ones according to offset

if(REG[ARGUMENT][2e(MMODE+3)−1:0] == −1) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

In the binary representation of the decimal –1 negative integer, all the bits are one (for any width), which is used as a shorthand notation in the condition evaluation in Table 60.

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

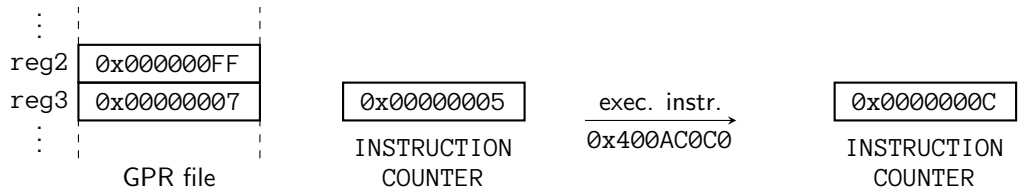
Examples

Example 84: *Branch if all Ones according to register*

Instruction in binary format: 01000000 000010 1011 00 000011 000000
Instruction in hexadecimal format: 0x400AC0C0
Fields:

MMODE	000	(byte)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If all bits in the 8-bit byte of GPR 2 are ones, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x400AC0C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if all bits in the 8-bit byte of GPR 2 are ones" is met and the branch is taken, which means that the next instruction to be executed is the 12-th (at address 0x30), skipping instructions 6 to 11.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

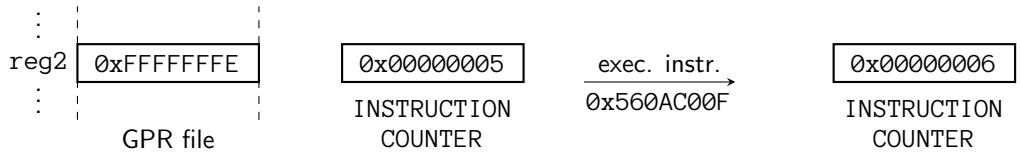
Example 85: *Branch if all Ones according to offset*

Instruction in binary format: 01010110 000010 1011 00000000001111
Instruction in hexadecimal format: 0x560AC00F
Fields:

MMODE	010	(word)
A	1	(absolute transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000001111	(0xF)

If all bits in the 32-bit word of GPR 2 are ones, the signed OFFSET14 value is written to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one.

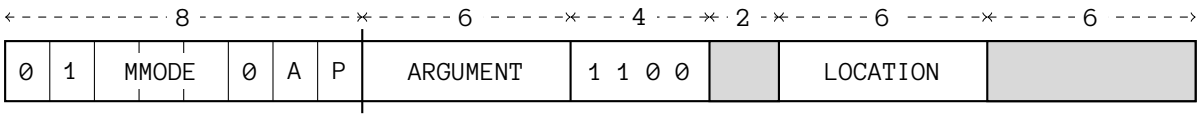
The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x560AC00F).



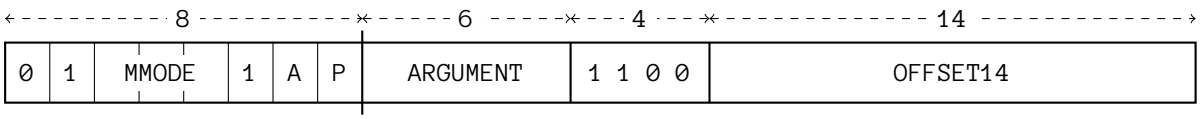
Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if all bits in the 32-bit word of GPR 2 are ones" is not met and the branch is not taken, which means that the next instruction to be executed is the 6-th (at address 0x18).

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.48 BNO – Branch if Not all Ones



(a) Branch if Not all Ones according to register



(b) Branch if Not all Ones according to offset

Fig. 79: *Branch if Not all Ones (BNO) instructions*

The BNZ instruction specifies conditional program transfer to a location specified by a GPR or by an implicitly specified offset. The branch condition is met if not all the bits of the argument GPR are 1 (in the specified machine mode).

Fields

MMODE	Specifies the integer machine mode according to Table 1.
A	0: **relative** transfer 1: **absolute** transfer (see Subsection 3.3.1).
P	If P=1, a **procedural** transfer is specified in which the INSTRUCTION COUNTER incremented by one is additionally written to the CALL RETURN POINTER (see Subsection 3.3.1).
ARGUMENT	Specifies the number of the GPR containing the argument which is investigated in order to decide whether to take the branch or not. MMODE specifies the machine mode, i.e., the integer width of the argument that is being investigated.
LOCATION	Specifies the number of the GPR containing the offset or absolute location for the transfer.
OFFSET14	A 14-bit signed offset value for branch according to offset.

Execution

Table 61 summarizes the execution of BNO instructions.

Table 61: *Execution of BNO instructions*

Branch if Not all Ones according to register

if(REG[ARGUMENT][2e(MMODE+3)−1:0] != −1) {
  if(A == 1) INSTRUCTION COUNTER ← REG[LOCATION]
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + REG[LOCATION]
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

Branch if Not all Ones according to offset

if(REG[ARGUMENT][2e(MMODE+3)−1:0] != −1) {
  if(A == 1) INSTRUCTION COUNTER ← OFFSET14
  else INSTRUCTION COUNTER ← INSTRUCTION COUNTER + OFFSET14
  if(P == 1) CALL RETURN POINTER ← INSTRUCTION COUNTER + 1
} **else** INSTRUCTION COUNTER ← INSTRUCTION COUNTER + 1

In the binary representation of the decimal –1 negative integer, all the bits are one (for any width), which is used as a shorthand notation in the condition evaluation in Table 61.

Changes

INSTRUCTION COUNTER	If the branch is taken, the INSTRUCTION COUNTER is loaded with the given/computed instruction address.
CALL RETURN POINTER	If the branch is taken and if P=1, the incremented value of the INSTRUCTION COUNTER is written to the CALL RETURN POINTER.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

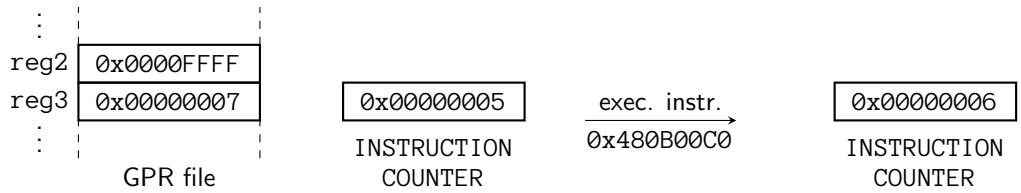
Examples

Example 86: *Branch if Not all Ones according to register*

Instruction in binary format: 01001000 000010 1100 00 000011 000000
Instruction in hexadecimal format: 0x480B00C0
Fields:

MMODE	001	(halfword)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
LOCATION	000011	(reg3)

If not all bits in the 16-bit halfword of GPR 2 are ones, the instruction adds the value in GPR 3 to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x480B00C0).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if not all bits in the 16-bit halfword of GPR 2 are ones" is not met and the branch is not taken, which means that the next instruction to be executed is the 6-th (at address 0x18).

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

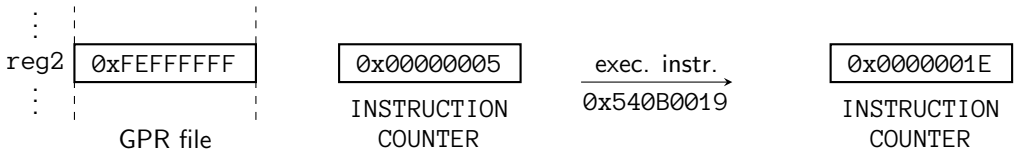
Example 87: *Branch if Not all Ones according to offset*

Instruction in binary format: 01010100 000010 1100 00000000011001
Instruction in hexadecimal format: 0x540B0019
Fields:

MMODE	010	(word)
A	0	(relative transfer)
P	0	(non-procedural transfer)
ARGUMENT	000010	(reg2)
OFFSET14	00000000011001	(0x19)

If not all bits in the 32-bit word of GPR 2 are ones, the signed OFFSET14 value is added to the INSTRUCTION COUNTER, otherwise the INSTRUCTION COUNTER is incremented by one. The following illustration shows an example state of a 32-bit wide GPR file and the value

of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x540B0019).



Thus, the branch instruction is the 5-th instruction (at address 0x14). The condition "branch if not all bits in the 32-bit word of GPR 2 are ones" is met and the branch is taken, which means that the next instruction to be executed is the 30-th (at address 0x78), skipping instructions 6 to 29.

If the branch is taken and if the transfer is procedural, i.e., the P bit is 1 instead of 0, the CALL RETURN POINTER will be written with the value of the INSTRUCTION COUNTER before instruction execution, incremented by one (0x00000006).

6.1.49 RET – Return from procedure

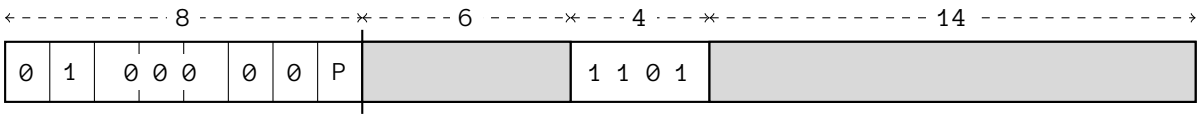


Fig. 80: *Return from procedure (RET) instructions*

The RET instruction specifies return from procedure. The point of return is specified by the CALL RETURN POINTER.

Fields

P	If P=1 the instruction address found in the return pointer is decremented by 1. That is, return to the last instruction before entering the which was **previously** executed.

Execution

Table 62 summarizes the execution of RET instructions.

Table 62: *Execution of RET instructions*

Return from procedure
if(P == 0) INSTRUCTION COUNTER ← CALL RETURN POINTER **else** INSTRUCTION COUNTER ← CALL RETURN POINTER – 1

Changes

INSTRUCTION COUNTER	It is loaded with the value of the CALL RETURN POINTER. If P=1, the value of the CALL RETURN POINTER is previously decremented by one.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION exception (see Subsection 4.2.5).

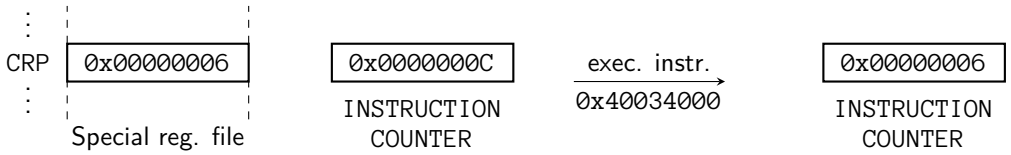
Examples

Example 88: *Return from procedure*

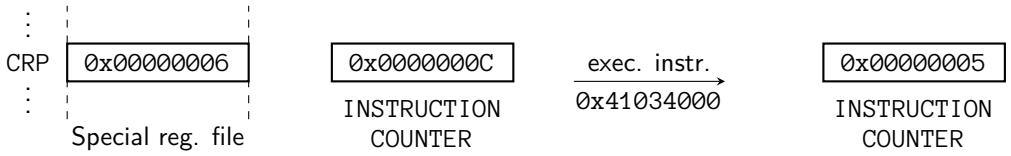
Instruction in binary format: 01000000 000000 1101 00000000000000
Instruction in hexadecimal format: 0x40034000
Fields:

P 0 (return after the procedure call)

The instruction returns to the place after the procedure call at a location given by the CALL RETURN POINTER (CRP). The following illustration shows the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x40034000).



If now only the P bit is changed and set to 1 (0x41034000), a return to the **P**revious instruction is specified since the value of the CRP is decremented by one. The following illustration shows the values of the INSTRUCTION COUNTER in this case.


CRP
0x00000006
Special reg. file
0x0000000C
INSTRUCTION COUNTER
exec. instr.
0x41034000
0x00000005
INSTRUCTION COUNTER

6.1.50 RETI – Return from interrupt handler

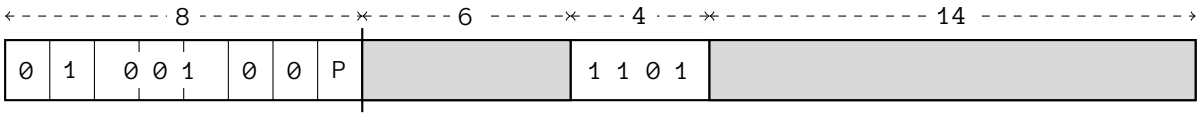


Fig. 81: *Return from interrupt handler (RETI) instructions*

The RETI instruction specifies return from an interrupt handler. The point of return is specified by the INTERRUPT RETURN POINTER.

Fields

P	If P=1 the instruction address found in the return pointer is decremented by 1. That is, return to the last instruction before entering the which was **previously** executed.

Execution

Table 63 summarizes the execution of RETI instructions.

Table 63: *Execution of RETI instructions*

Return from interrupt handler
if(P == 0) INSTRUCTION COUNTER ← INTERRUPT RETURN POINTER **else** INSTRUCTION COUNTER ← INTERRUPT RETURN POINTER – 1

The RETI instruction restores the enabled/disabled status of the interrupt line as it was before interrupt handling was entered. That is, bit 3 (ENABLE INTERRUPTS) of the SYSTEM CONTROL REGISTER (see Subsection 5.10) is restored to its value before entering interrupt handling. However, if RETI is executed out of the interrupt handler, i.e., if interrupt handling was not entered, the SYSTEM CONTROL REGISTER bit is not changed. RETI is not a system instruction like RETE and RETN, and can be also executed in user mode.

Changes

INSTRUCTION COUNTER	It is loaded with the value of the INTERRUPT RETURN POINTER. If P=1, the value of the INTERRUPT RETURN POINTER is previously decremented by one.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION exception (see Subsection 4.2.5).

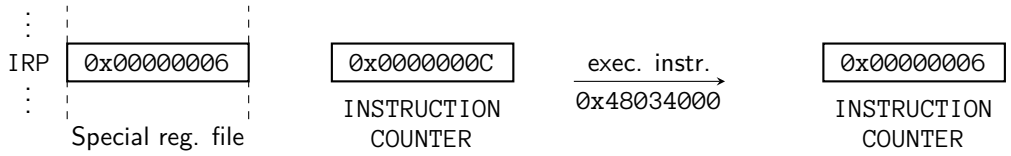
Examples

Example 89: *Return from interrupt handler*

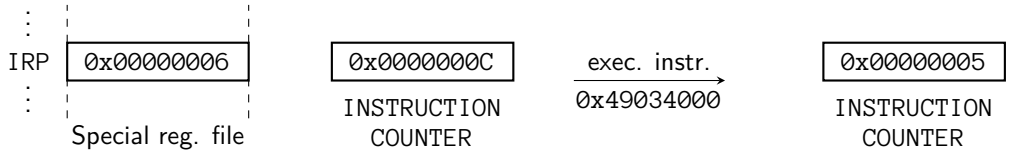
Instruction in binary format: 01001000 000000 1101 00000000000000
Instruction in hexadecimal format: 0x48034000
Fields:

P 0 (return after the interrupted instruction)

The instruction returns to the place after the interrupted instruction at a location given by the INTERRUPT RETURN POINTER (IRP). The following illustration shows the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x48034000).



If now only the P bit is changed and set to 1 (0x49034000), a return to the **P**revious instruction is specified since the value of the IRP is decremented by one. The following illustration shows the values of the INSTRUCTION COUNTER in this case.



6.1.51 RETE – Return from exception handler

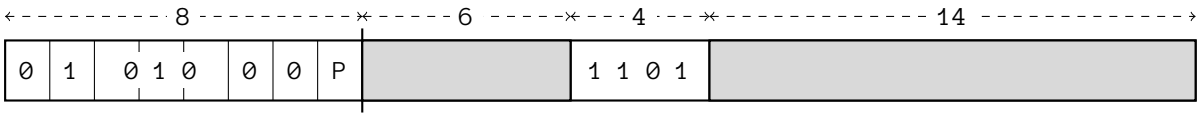


Fig. 82: *Return from exception handler (RETE) instructions*

The RETE instruction specifies return from an exception handler. The point of return is specified by the EXCEPTION RETURN POINTER.

Fields

P	If P=1 the instruction address found in the return pointer is decremented by 1. That is, return to the last instruction before entering the which was **previously** executed.

Execution

Table 64 summarizes the execution of RETE instructions.

Table 64: *Execution of RETE instructions*

Return from exception handler
if(P == 0) INSTRUCTION COUNTER ← EXCEPTION RETURN POINTER **else** INSTRUCTION COUNTER ← EXCEPTION RETURN POINTER – 1

The RETE instruction restores the operating mode and the enabled/disabled status of the exceptions and the interrupt line as they were before exception handling was entered. That is, bits 0, 2 and 3 (SYSTEM MODE, ENABLE EXCEPTIONS and ENABLE INTERRUPTS) of the SYSTEM CONTROL REGISTER (see Subsection 5.10) are restored to their values before entering exception handler. However, if RETE is executed out of the exception handler, i.e., if exception handling was not entered, the SYSTEM CONTROL REGISTER bits are not changed. RETE is a system instruction and can be executed only in system mode.

Changes

INSTRUCTION COUNTER	It is loaded with the value of the EXCEPTION RETURN POINTER. If P=1, the value of the EXCEPTION RETURN POINTER is previously decremented by one.

Exceptions

SYSTEM INSTRUCTION	It is raised when an attempt is made to execute the instruction in user mode.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION exception (see Subsection 4.2.5).

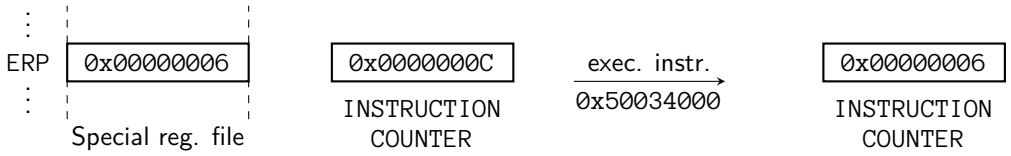
Examples

Example 90: *Return from exception handler*

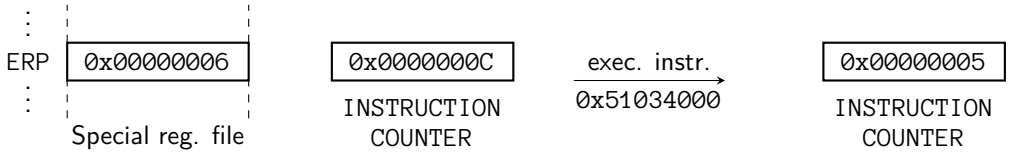
Instruction in binary format: 01010000 000000 1101 00000000000000
Instruction in hexadecimal format: 0x50034000
Fields:

P 0 (return after the exceptional instruction)

The instruction returns to the place after the exceptional instruction at a location given by the EXCEPTION RETURN POINTER (ERP). The following illustration shows the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x50034000).



If now only the P bit is changed and set to 1 (0x51034000), a return to the **P**revious instruction is specified since the value of the ERP is decremented by one. The following illustration shows the values of the INSTRUCTION COUNTER in this case.



6.1.52 RETN – Return from NMI handler

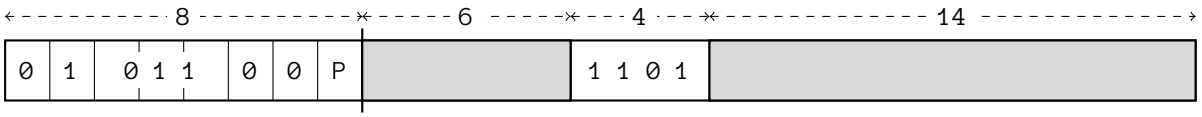


Fig. 83: *Return from NMI handler (RETN) instructions*

The RETN instruction specifies return from an NMI handler. The point of return is specified by the NMI RETURN POINTER.

Fields

P	If P=1 the instruction address found in the return pointer is decremented by 1. That is, return to the last instruction before entering the which was **previously** executed.

Execution

Table 65 summarizes the execution of RETN instructions.

Table 65: *Execution of RETN instructions*

Return from NMI handler
if(P == 0) INSTRUCTION COUNTER ← NMI RETURN POINTER **else** INSTRUCTION COUNTER ← NMI RETURN POINTER – 1

The RETN instruction restores the operating mode and the enabled/disabled status of the exceptions and the interrupt line as they were before NMI handling was entered. That is, bits 0, 2 and 3 (SYSTEM MODE, ENABLE EXCEPTIONS and ENABLE INTERRUPTS) of the SYSTEM CONTROL REGISTER (see Subsection 5.10) are restored to their values before entering NMI handler. However, if RETN is executed out of the NMI handler, i.e., if NMI handling was not entered, the SYSTEM CONTROL REGISTER bits are not changed. RETN is a system instruction and can be executed only in system mode.

Changes

INSTRUCTION COUNTER	It is loaded with the value of the NMI RETURN POINTER. If P=1, the value of the NMI RETURN POINTER is previously decremented by one.

Exceptions

SYSTEM INSTRUCTION	It is raised when an attempt is made to execute the instruction in user mode.

Depending on the implementation, this instruction can also raise the UNIMPLEMENTED INSTRUCTION exception (see Subsection 4.2.5).

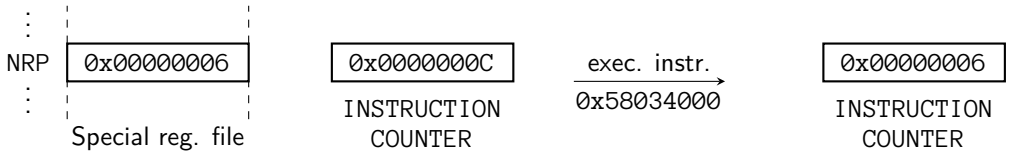
Examples

Example 91: *Return from NMI handler*

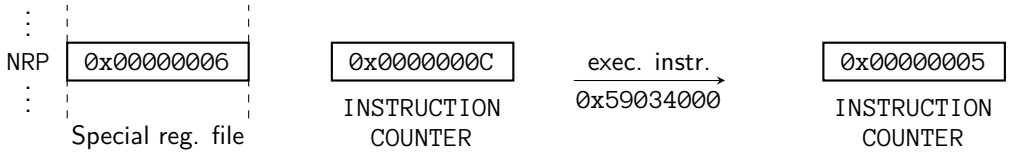
Instruction in binary format: 01011000 000000 1101 00000000000000
Instruction in hexadecimal format: 0x58034000
Fields:

P 0 (return after the interrupted instruction)

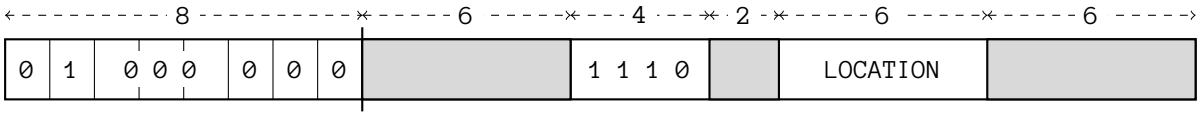
The instruction returns to the place after the interrupted instruction at a location given by the NMI RETURN POINTER (NRP). The following illustration shows the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x58034000).



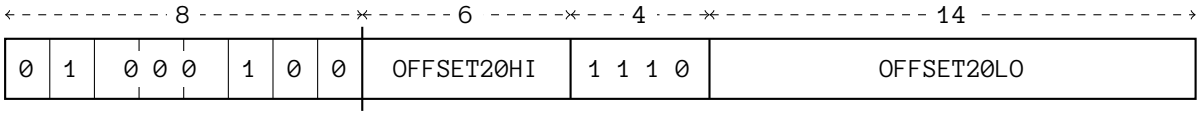
If now only the P bit is changed and set to 1 (0x59034000), a return to the **P**revious instruction is specified since the value of the NRP is decremented by one. The following illustration shows the values of the INSTRUCTION COUNTER in this case.



6.1.53 WAIT – Wait



(a) Wait according to register



(b) Wait according to offset

Fig. 84: *Wait (WAIT) instructions*

The WAIT instruction specifies a pause from instruction execution. The specified wait timer value (by a GPR, or by the OFFSET20 field of the instruction) determines if the pause is definite or indefinite. If the value is zero, the pause is indefinite, otherwise it is definite. If the pause is definite, instruction execution is resumed after expiration of the pause period, i.e., the wait timer reaches zero, or if an NMI or a potent interrupt is raised. On the other hand, an indefinite pause can be broken only by an NMI or a potent interrupt. Another way to break a pause is by a system reset.

Fields

LOCATION	Specifies the number of the GPR containing the wait timer value. The WAIT instruction always interprets the wait timer value as unsigned.
OFFSET20HI	The six MSBs of the 20-bit unsigned value OFFSET20.
OFFSET20LO	The 14 LSBs of the 20-bit unsigned value OFFSET20.

Execution

Table 66 summarizes the execution of WAIT instructions.

Table 66: *Execution of WAIT instructions*

Wait according to register

if(REG[LOCATION] == 0)
  forever INSTRUCTION COUNTER ← INSTRUCTION COUNTER
else
  for(REG[LOCATION]) INSTRUCTION COUNTER ← INSTRUCTION COUNTER

Wait according to immediate

if(OFFSET20 == 0)
  forever INSTRUCTION COUNTER ← INSTRUCTION COUNTER
else
  for(OFFSET20) INSTRUCTION COUNTER ← INSTRUCTION COUNTER

The for(x) ‹statements› construct in Table 66 implies that the ‹statements› are executed x times, i.e., the INSTRUCTION COUNTER remains unchanged for REG[LOCATION] / OFFSET20 clock cycles, as the wait timer is decremented on each clock cycle. On the other hand, the forever construct implies that the statements after are executed infinitely.

Changes

INSTRUCTION COUNTER	It is left unchanged (either definitely or indefinitely), i.e., the auto-increment functionality of the INSTRUCTION COUNTER is paused.

Exceptions

Depending on the implementation, this instruction can raise the UNIMPLEMENTED INSTRUCTION and UNIMPLEMENTED REGISTER exceptions (see Subsections 4.2.5 and 4.2.6).

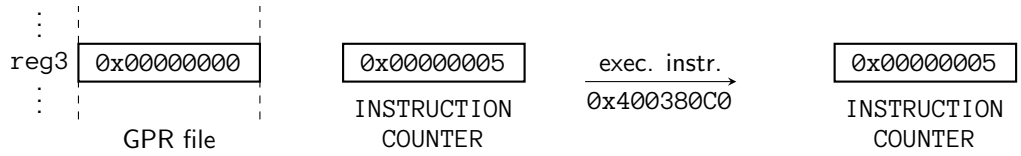
Examples

Example 92: *Wait according to register*

Instruction in binary format: 01000000 000000 1110 00 000011 000000
Instruction in hexadecimal format: 0x400380C0
Fields:

LOCATION	000011	(reg3)

The execution is paused for a definite or indefinite period, according to the value in GPR 3. The following illustration shows an example state of a 32-bit wide GPR file and the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction (0x400380C0).



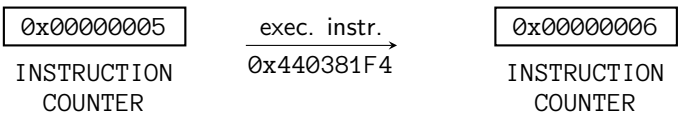
Thus, the execution is paused indefinitely since the value of GPR 3 is zero. Execution can be resumed only by an interrupt or NMI. Of course, a system reset also "resumes" execution.

Example 93: *Wait according to offset*

Instruction in binary format: 01000100 000000 1110 00000111110100
Instruction in hexadecimal format: 0x440381F4
Fields:

OFFSET20HI	000000	(0x0)
OFFSET20LO	00000111110100	(0x1F4)

The execution is paused for a definite or indefinite period, according to the value in OFFSET20. The concatenation of OFFSET20HI and OFFSET20LO gives OFFSET20 = 0x001F4. The following illustration shows the value of the 30-bit wide INSTRUCTION COUNTER before and after execution of the instruction, i.e., after expiration of the pause period. (0x440381F4).



Thus, the execution is paused for 500 clock cycles after which execution resumes. That is, the wait timer value is set to 500 (0x001F4). Execution can be also resumed before expiration of the defined pause period by an interrupt or NMI.

6.2 System instructions

System instructions are instructions that can be executed only in system mode and are used for operating system protection. An attempt to execute a system instruction in user mode raises the `SYSTEM INSTRUCTION` exception (see Subsection 4.2.3).

Any inter-register transfer instruction in which the destination is a special register that is non-writable in user mode is a system instruction (see Table 12).

The *return from exception* and *return from NMI handler* (`RETE` and `RETN`) instructions are also system instructions.

6.3 Assembly conventions

In assembly language, all PEAKTOP instructions are specified by a mnemonic concatenated with zero, one, two or three instruction options. An underscore concatenates the mnemonic with the instruction options. The assembly language is **case-insensitive**. The instructions can have zero, one, two or three arguments. Thus, they take one of the the following forms:

1. `<mnemonic>`
2. `<mnemonic> <arg1>`
3. `<mnemonic> <arg1>, <arg2>`
4. `<mnemonic> <arg1>, <arg2>, <arg3>`
5. `<mnemonic>_<option(s)>`
6. `<mnemonic>_<option(s)> <arg1>`
7. `<mnemonic>_<option(s)> <arg1>, <arg2>`
8. `<mnemonic>_<option(s)> <arg1>, <arg2>, <arg3>`

For example, an instruction that adds GPR 2 to GPR 1 in byte machine mode is written as:

```
ADD_B reg1, reg2
```

which will write the result back to GPR 1. For unsigned addition, where both operands are considered unsigned, the assembly line will be:

```
ADD_UB reg1, reg2
```

Permutation of the option letters is also possible:

```
ADD_BU reg1, reg2
```

Thus, the last two instructions translate to the same binary representation.

Note that all instruction mnemonics consist of 2, 3 or 4 letters, while each option is specified by one letter.

6.3.1 Instruction options

Instructions usually have one or more alternatives, or optional functionalities which are additionally specified by instruction options. For example, an integer arithmetic instruction can

be executed either as unsigned or signed (with or without the U option), or, during program transfer, the instruction may be specified to additionally write the INSTRUCTION COUNTER to the CALL RETURN POINTER by specifying the P option.

Machine mode options

The large bulk of the instructions in the PEAKTOP ISA can be executed in several machine modes. The machine mode in which the instruction is executed is specified by the machine mode options. Table 1 gives the option letters used for each machine mode.

Machine mode options are mutually exclusive, since one instruction cannot be executed in multiple machine modes simultaneously. Thus, writing ADD_BH reg1, reg2, for example, is illegal.

An option that specifies the natural machine mode option (one of B H W D Q 1 2 4) can be omitted in assembly. For instance, in implementations with W natural machine mode `ADD_W reg1, reg2` can be also written as `ADD reg1, reg2`.

However, omitting the machine mode option in assembly could affect the portability of programs between implementations with different natural modes, if the program uses mixed machine modes.

The U option

The U option is used to specify:

- Unsigned data
- Unsigned operation

Specifying the U option for a load immediate instruction sets the U bit to 1. If not specified, the U bit is set to 0. See Subsection 3.1.3.

The U option also specifies unsigned arithmetic operation. That is, specifying the U option for the integer arithmetic (ADD, SUB, MUL, DIV) and the fused multiply-add/subtract (MAD, MSU) instructions, sets their U bit to 1. If the U option is not specified, the U bit for these instructions is set to 0. See Subsection 3.2.1.

When the second operand of the logic instructions (AND, NAND, OR, XOR) is an immediate value, the U option can be specified in order to zero-extend the immediate value to the operation width. If the U option is not specified, the immediate value will be sign-extended. See Subsection 3.2.1.

Finally, conversions from FP format to integer format (FF2I) or vice versa (FI2F) have the U option in order to specify that the integer result or source operand, respectively, is unsigned. See Subsection 3.2.2.

The A option

The A option is used to specify:

- Atomic memory transfer
- Arithmetic shift
- Absolute program transfer

Specifying the A option for a memory transfer instruction sets the U bit to 1. If not specified, the U bit is set to 0. See Subsection 3.1.1.

Specifying the A option for the shift instructions SL and SR sets the U bit to 0, thus inferring an arithmetic left/right shift. If the A option is not specified, the U bit is set to 1 inferring a logic left/right shift. See Subsection 3.2.1.

Specifying the A option for a program transfer instruction sets the A bit to 1. If not specified, the A bit is set to 0. See Subsection 3.3.1.

The P option

The P option is used to specify:

- Procedural program transfer
- return from routine to the Previously executed instruction

Specifying the P option for a program transfer instruction (either unconditional or branch), or for a return from routine instruction, sets the P bit to 1. If not specified, the P bit is set to 0. See Subsections 3.3.1 and 3.3.2, respectively.

6.3.2 Instruction arguments

All instructions, except the return from routine (RET, RETI, RETE and RETN) instructions take at least one argument, and maximum three arguments. Instructions taking more than one argument are comma-separated. Only the fused multiply-add/subtract instructions, both integer and FP, (i.e., MAD, MSU, FMAD and FMSU) take three arguments. There are three types of arguments: register, numerical and address arguments.

Register arguments

The register arguments are given as explained in Subsection 2.2.1. That is:

reg<nr> The <nr>-th register from the GPR file.

spc<nr> The <nr>-th special register.

dsp<nr> The <nr>-th DSP register.

For example, copying the IMPLEMENTATION REGISTER spc0 to the fourth GPR is written as:

```
MOV reg4, spc0
```

This form of register naming must be applicable in all implementations. Furthermore, registers can have user-defined aliases, i.e., additional names, which may resemble the function of the register. Thus, the spc0 register can have the alias, e.g., IMP, while the reg4 can have the alias r4. Thus, the previous code line becomes:

```
MOV r4, IMP
```

The aliases can differ between implementations. However, it is recommended that the aliases proposed in this specification are used, simply for the purposes of assembly program compatibility. The proposed aliases for the GPR and the DSP registers are r<nr> and d<nr>, respectively, where <nr> is in the range [0, 63]. The proposed aliases for the special registers are given in Table 12 on Page 48.

Numerical arguments

Numerical arguments are used to specify immediate values, as well as offsets for data or program transfer. The WAIT instruction also can take a numerical argument. For example, loading reg0 with the immediate value of 5 will be:

```
MOV reg0, 5
```

The numerical arguments can be specified in five different formats given by Table 67.

That is, 0x and 0b before the number specify that the number is in hexadecimal and binary format, respectively, while a zero before the number signals the octal format. The exponential format is useful when FP operations are involved. Furthermore, numeric expressions that reduce to unambiguous numerical value are also acceptable. However, numeric expressions are not a subject of this specification.

Table 67: *Formats of numerical arguments*

Format	Example
hexadecimal	`0x1234`
decimal	`1234`
octal	`01234`
binary	`0b1110011`
exponential	`-12.34e56`

Address arguments

Address arguments are used by the memory transfer instructions to specify the address according to the addressing mode. Angle brackets are used to specify the register containing the address in register addressing, and to specify the offset and index in displacement and indexed addressing, respectively. That is

register `[reg‹nr›]`

displacement `reg‹nr-base-reg›[‹offset›]`

indexed `reg‹nr-base-reg›[reg‹nr-index-reg›]`

For example, loading from memory into `reg1` using the register addressing in which the address is placed in `reg3` is written as:

```
MOV reg1, [reg3]
```

For displacement addressing with offset 4 in which the base address is in `reg2`:

```
MOV reg1, reg2[4]
```

while, for indexed addressing in which the base address is in `reg2` and the index is in `reg3`:

```
MOV reg1, reg2[reg3]
```

On the other hand, storing the value of `reg1` to memory using the last three addressing modes is the same, except that the order of the arguments is swapped, i.e.,

`MOV [reg3], reg1`, `MOV reg2[4], reg1` and `MOV reg2[reg3], reg1`, respectively.

Pre- and post-increment/decrement of the index is specified by adding ++ and -- before and after the index register, respectively. For example,

```
MOV reg1, reg2[reg3++]
```

post-increments the index register `reg3`, while

```
MOV reg1, reg2[--reg3]
```

pre-decrements it. Or, for register addressing without base address

`MOV reg1, [++reg3]` and `MOV reg1, [reg3--]`

pre-increment and post-decrement `reg3`, respectively.

6.3.3 Summary

Table 68 gives the possible option combinations for each instruction, the type and number of arguments. Note that permutation of the option letters is allowed when there is more than one option. Furthermore, the natural machine mode option can be omitted.

Table 69 shows the assembly according to the proposed assembly conventions for all examples in Subsection 6.1.

Table 68: *Instruction options and arguments*

Instruction	Possible option combinations	Number and type of arguments
Data transfer instructions		
MOV (memory transfer – load)	B H W D Q 1 2 4 AB AH AW AD AQ A1 A2 A4	register, address
MOV (memory transfer – store)	B H W D Q 1 2 4 AB AH AW AD AQ A1 A2 A4	address, register
MOV (inter-register transfer)	B H W D Q 1 2 4	register, register
MOV (load immediate)	B H W D Q 1 2 4 UB UH UW UD UQ U1 U2 U4	register, numerical
Arithmetic/logic instructions		
ADD, SUB, MUL, DIV	B H W D Q 1 2 4 UB UH UW UD UQ U1 U2 U4	register, register/numerical
SL, SR	B H W D Q 1 2 4 AB AH AW AD AQ A1 A2 A4	register, register/numerical
RL, RR	B H W D Q 1 2 4	register, register/numerical
AND, NAND, OR, XOR	B H W D Q 1 2 4	register, register
	B H W D Q 1 2 4 UB UH UW UD UQ U1 U2 U4	register, numerical
SB, RB, TB, RVB	B H W D Q 1 2 4	register, register/numerical
FADD, FSUB, FMUL, FDIV, FREM, FCMP		register, register
FSQR, FABS, FNEG, FRND		register
FF2I, FI2F	B H W D Q 1 2 4 UB UH UW UD UQ U1 U2 U4	register
FEXT, FSQZ	H W D Q 1	register
MAD, MSU	B H W D Q 1 2 4 UB UH UW UD UQ U1 U2 U4	register, register, register/numerical
FMAD, FMSU		register, register, register
Control instructions		
JMP	A P AP	register/numerical
BZ, BNZ, BM BMZ, BNM, BNMO BL, BLZ, BNL BNLO, BO, BNO	B H W D Q 1 2 4 AB AH AW AD AQ A1 A2 A4 PB PH PW PD PQ P1 P2 P4 APB APH APW APD APQ AP1 AP2 AP4	register, register/numerical
RET, RETI, RETE, RETN	P	
WAIT		register/numerical

Table 69: *Assembly of the example instructions in Subsection 6.1*

Example	Page	Hexadecimal	Assembly
1	60	0xCC0C0085	`MOV_H reg3, reg2[5]`
2	61	0x900C0002	`MOV_W [reg2], reg3`
3	61	0xD80F4081	`MOV_D reg3, reg2[++reg1]`
4	62	0xC13C8003	`MOV_B spc15, reg3`
5	62	0xD50FFFFB	`MOV_W reg3, -5`
5	62	0xD70FFFFB	`MOV_UW reg3, -5`
6	65	0x100C0080	`ADD_W reg3, reg2`
7	65	0x0C0C0003	`ADD_H reg3, 3`
8	68	0x100C4080	`SUB_W reg3, reg2`

Table 69 continues on next page...

... Table 69 continued from previous page

Example	Page	Hexadecimal	Assembly
9	68	0x140C76EB	SUB_W reg3, -2325
9	68	0x160C76EB	SUB_UW reg3, -2325
10	71	0x120C8080	MUL_UW reg3, reg2
11	71	0x080C8080	MUL_H reg3, reg2
12	72	0x140C98AC	MUL_W reg3, 0x18AC
13	74	0x100CC080	DIV_W reg3, reg2
14	75	0x0A0CC080	DIV_UH reg3, reg2
15	75	0x040CF83A	DIV_B reg3, 0x383A
16	78	0x120D0080	SL_W reg3, reg2
17	78	0x040D0004	SL_AB reg3, 4
18	81	0x120D4080	SR_W reg3, reg2
19	81	0x040D4004	SR_AB reg3, 4
20	84	0x100D8080	RL_W reg3, reg2
21	84	0x0C0D8025	RL_H reg3, 37
22	87	0x120D8080	RR_W reg3, reg2
23	87	0x060D8001	RR_B reg3, 1
24	90	0x100DC080	AND_W reg3, reg2
25	90	0x140DCF0F	AND_W reg3, 3855
26	93	0x100E0080	NAND_W reg3, reg2
27	93	0x140E3F0F	NAND_W reg3, -241
28	96	0x100E4080	OR_W reg3, reg2
29	96	0x160E7F0F	OR_UW reg3, -241
30	99	0x100E8080	XOR_W reg3, reg2
31	99	0x160EBF0F	XOR_UW reg3, -241
32	102	0x100EC080	SB_W reg3, reg2
33	102	0x040EC008	SB_B reg3, 8
34	105	0x120EC080	RB_W reg3, reg2
35	105	0x060EC002	RB_B reg3, 2
36	108	0x100F0080	TB_W reg3, reg2
37	108	0x040F0002	TB_B reg3, 2
38	111	0x120F0080	RVB_W reg3, reg2
39	111	0x160F000F	RVB_W reg3, 15
40	114	0x010C0080	FADD reg3, reg2
41	116	0x010C4080	FSUB reg3, reg2
42	118	0x010C8080	FMUL reg3, reg2
43	120	0x010CC080	FDIV reg3, reg2
44	122	0x010D0080	FREM reg3, reg2
45	123	0x010D4080	FCMP reg3, reg2
46	126	0x010D8000	FSQR reg3
47	128	0x010DC000	FABS reg3
48	129	0x010E0000	FNEG reg3
49	132	0x010E4000	FRND reg3
50	134	0x110E8000	FF2I_W reg3
51	136	0x110EC000	FI2F_W reg3
52	138	0x110F0000	FEXT_W reg3

Table 69 continues on next page...

... Table 69 continued from previous page

Example	Page	Hexadecimal	Assembly
53	140	0x110F4000	FSQZ_W reg3
54	142	0x100F8081	MAD_W reg3, reg2, reg1
55	142	0x140F90A4	MAD_W reg3, reg2, 100
56	145	0x100FC081	MSU_W reg3, reg2, reg1
57	145	0x140FD0A4	MSU_W reg3, reg2, 100
58	148	0x010F8081	FMAD reg3, reg2, reg1
59	150	0x010FC081	FMSU reg3, reg2, reg1
60	152	0x400000C0	JMP reg3
61	152	0x420000C0	JMP_A reg3
62	152	0x44FC3FFD	JMP -3
63	153	0x46FC3FFD	JMP_A -3
64	155	0x400840C0	BZ_B reg2, reg3
65	155	0x5608400F	BZ_AW reg2, 15
66	158	0x480880C0	BNZ_H reg2, reg3
67	158	0x54088019	BNZ_W reg2, 25
68	161	0x4008C0C0	BM_B reg2, reg3
69	161	0x5608C019	BM_AW reg2, 25
70	164	0x480900C0	BMZ_H reg2, reg3
71	164	0x54090019	BMZ_W reg2, 25
72	167	0x400940C0	BNM_B reg2, reg3
73	167	0x56094019	BNM_AW reg2, 25
74	170	0x480980C0	BNMO_H reg2, reg3
75	170	0x54098019	BNMO_W reg2, 25
76	173	0x4009C0C0	BL_B reg2, reg3
77	173	0x5609C019	BL_AW reg2, 25
78	176	0x480A00C0	BLZ_H reg2, reg3
79	176	0x540A0019	BLZ_W reg2, 25
80	179	0x400A40C0	BNL_B reg2, reg3
81	179	0x560A4019	BNL_AW reg2, 25
82	182	0x480A80C0	BNLO_H reg2, reg3
83	182	0x540A8019	BNLO_W reg2, 25
84	185	0x400AC0C0	BO_B reg2, reg3
85	185	0x560AC00F	BO_AW reg2, 15
86	188	0x480B00C0	BNO_H reg2, reg3
87	188	0x540B0019	BNO_W reg2, 25
88	190	0x40034000	RET
88	190	0x41034000	RET_P
89	192	0x48034000	RETI
89	192	0x49034000	RETI_P
90	194	0x50034000	RETE
90	194	0x51034000	RETE_P
91	196	0x58034000	RETN
91	196	0x59034000	RETN_P
92	199	0x400380C0	WAIT reg3
93	199	0x440381F4	WAIT 500

6.4 Pseudo-instructions

The pseudo-instructions are assembly constructs that translate to one or more native instructions. Pseudo-instructions that translate to a single native instruction are called single pseudo-instructions, while pseudo-instructions that translate to more than one native instruction are called multiple pseudo-instructions.

An implementation can use the pseudo-instructions defined here (or a subset of them), but can also define and use implementation-specific pseudo-instructions. Not using pseudo-instructions is also an option.

The next enumeration gives the pseudo-instruction forms which are similar to the forms of the native instructions. The enumeration is actually continued from the native instruction forms in Subsection 6.3 on Page 200.

9. ‹pseudo–mnemonic›
10. ‹pseudo–mnemonic› ‹pseudo–arg1›
11. ‹pseudo–mnemonic› ‹pseudo–arg1›, ‹pseudo–arg2›
12. ‹pseudo–mnemonic›_‹option(s)›
13. ‹pseudo–mnemonic›_‹option(s)› ‹pseudo–arg1›
14. ‹pseudo–mnemonic›_‹option(s)› ‹pseudo–arg1›, ‹pseudo–arg2›

The options, if used, are the same as in the non-pseudo forms (see Subsection 6.3.1). On the other side, the pseudo-arguments can be either the same as the non-pseudo arguments, or different. In any case, the specification of the arguments follows the same rules as for the native instructions (see Subsection 6.3.2). The following tokens are used for the arguments in Tables 70 and 71.

reg‹nr›	The ‹nr›-th GPR.
spc‹nr›	The ‹nr›-th special register.
dsp‹nr›	The ‹nr›-th DSP register.
‹addr›	An address argument specified as described in Subsection 6.3.2.
‹num›	A numerical argument specified as described in Subsection 6.3.2.
SP	An alias for a GPR which is used as a stack pointer.
TMP	An alias for a GPR which is used as a register for storing temporary content.

6.4.1 Single pseudo-instructions

The purpose of single pseudo-instructions is to simplify the representation of the native instructions or to stress the operation of the native instructions. Table 70 shows the single pseudo-instructions and their one-to-one translation to native instructions.

LIMM and CALL ‹num› become multiple pseudo-instructions if the binary width of the numerical argument is greater than 18 bits (for LIMM) or 20-bits (for CALL), respectively.

Table 70: *Single pseudo-instructions*

Pseudo-instruction	Translation to native	Description
LOAD reg<nr>, <addr>	**MOV** reg<nr>, <addr>	Load from memory
STOR <addr>, reg<nr>	**MOV** <addr>, reg<nr>	Store in memory
PUSH reg<nr>	**MOV** [--SP], reg<nr>	Push on stack
POP reg<nr>	**MOV** reg<nr>, [SP++]	Pop from stack
COPY reg<nr>, reg<nr2>	**MOV** reg<nr>, reg<nr2>	Copy register (REG<–REG)
COPY reg<nr>, spc<nr2>	**MOV** reg<nr>, spc<nr2>	Copy register (REG<–SPC)
COPY spc<nr>, reg<nr2>	**MOV** spc<nr>, reg<nr2>	Copy register (SPC<–REG)
COPY reg<nr>, dsp<nr2>	**MOV** reg<nr>, dsp<nr2>	Copy register (REG<–DSP)
COPY dsp<nr>, reg<nr2>	**MOV** dsp<nr>, reg<nr2>	Copy register (DSP<–REG)
NOP	**MOV** reg0, reg0	No operation
LIMM reg<nr>, <num>	**MOV** reg<nr>, <num>	Load immediate
INCR reg<nr>	**ADD** reg<nr>, 1	Increment register
DECR reg<nr>	**SUB** reg<nr>, 1	Decrement register
SQR reg<nr>	**MUL** reg<nr>, reg<nr>	Square register
NOT reg<nr>	**NAND** reg<nr>, –1	NOT bitwise register
FCLS reg<nr>	**FCMP** reg<nr>, reg<nr>	Classify FP number
CALL reg<nr>	**JMP_AP** reg<nr>	Call procedure (register)
CALL <num>	**JMP_AP** <num>	Call procedure (immediate)
WAIT	**WAIT** 0	Wait indefinitely

6.4.2 Multiple pseudo-instructions

The purpose of multiple pseudo-instructions is to group several native instructions that perform a frequently used operation in order to simplify the assembly. Table 71 shows the multiple pseudo-instructions and their translation to native instructions.

Table 71: *Multiple pseudo-instructions*

Pseudo-instruction	Translation to natives
MEM <addr>, <addr2>	**LOAD** TMP, <addr2> **STOR** <addr>, TMP
LOAD spc<nr>, <addr>	**LOAD** TMP, <addr> **COPY** spc<nr>, TMP
LOAD dsp<nr>, <addr>	**LOAD** TMP, <addr> **COPY** dsp<nr>, TMP
LOAD reg<nr>, <num>	**LIMM** TMP, <num> **LOAD** reg<nr>, [TMP]
STOR <addr>, spc<nr>	**COPY** TMP, spc<nr> **STOR** <addr>, TMP
STOR <addr>, dsp<nr>	**COPY** TMP, dsp<nr> **STOR** <addr>, TMP
STOR <num>, reg<nr>	**LIMM** TMP, <num> **STOR** [TMP], reg<nr>
PUSH spc<nr>	**COPY** TMP, spc<nr> **PUSH** TMP
PUSH dsp<nr>	**COPY** TMP, dsp<nr> **PUSH** TMP
POP spc<nr>	**POP** TMP **COPY** spc<nr>, TMP
POP dsp<nr>	**POP** TMP **COPY** dsp<nr>, TMP
COPY spc<nr>, spc<nr2>	**COPY** TMP, spc<nr2>

Table 71 continues on next page...

. . . Table 71 continued from previous page

	COPY spc<nr>, TMP
COPY dsp<nr>, dsp<nr2>	**COPY** TMP, dsp<nr2> **COPY** dsp<nr>, TMP
LIMM reg<nr>, <num> multi-pseudo only if <num> width > 18 bits (see remark 8 below)	See remark 7 below.
LIMM spc<nr>, <num>	**LIMM** TMP, <num> **COPY** spc<nr>, TMP
LIMM dsp<nr>, <num>	**LIMM** TMP, <num> **COPY** dsp<nr>, TMP
SIMM <addr>, <num>	**LIMM** TMP, <num> **STOR** <addr>, TMP
NOP <num>	<num> times **MOV** reg0, reg0
NEG reg<nr>	**BZ** reg<nr>, 3 **NOT** reg<nr> **INCR** reg<nr>
ABS reg<nr>	**BNM** reg<nr>, 3 **NOT** reg<nr> **INCR** reg<nr>
<**ALU_OP**> reg<nr>, <num> multi-pseudo only if <num> width > 14 bits (see remarks 5 and 8 below)	**LIMM** TMP, <num> <**ALU_OP**> reg<nr>, TMP
MAD/MSU reg<nr>, reg<nr2>, <num> multi-pseudo only if <num> width > 8 bits (see remark 8 below)	**LIMM** TMP, <num> **MAD/MSU** reg<nr>, reg<nr>, TMP
TB spc<nr>, reg<nr> (see remark 9 below)	**COPY** TMP, spc<nr> **TB** TMP, reg<nr>
TB spc<nr>, <num> (see remark 9 below)	**COPY** TMP, spc<nr> **TB** TMP, <num>
SB/RB spc<nr>, reg<nr>	**COPY** TMP, spc<nr> **SB/RB** TMP, reg<nr> **COPY** spc<nr>, TMP
SB/RB spc<nr>, <num>	**COPY** TMP, spc<nr> **SB/RB** TMP, <num> **COPY** spc<nr>, TMP
JMP/WAIT <num> multi-pseudo only if <num> width > 20 bits (see remark 8 below)	**LIMM** TMP, <num> **JMP/WAIT** TMP
<**BRANCH**> reg<nr>, <num> multi-pseudo only if <num> width > 14 bits (see remarks 6 and 8 below)	**LIMM** TMP, <num> <**BRANCH**> reg<nr>, TMP
SYSM	**SB** SCR, 0
USRM	**RB** SCR, 0
DBGM	**SB** SCR, 1
NDBG	**RB** SCR, 1
EE	**SB** SCR, 2
DE	**RB** SCR, 2
EECE	**LIMM** TMP, 0 **COPY** EXC, TMP **SB** SCR, 2
DECE	**LIMM** TMP, 0 **COPY** EXC, TMP **RB** SCR, 2
EI	**SB** SCR, 3

Table 71 continues on next page. . .

. . . Table 71 continued from previous page

DI	**RB** SCR, 3
SYNC	**SB** UCR, 4
CSYN	**RB** UCR, 4
TEXC reg<nr>	**TB** EXC, reg<nr>
TEXC <num>	**TB** EXC, <num>
EACK reg<nr>	**RB** EXC, reg<nr>
EACK <num>	**RB** EXC, <num>
TEXM reg<nr>	**TB** EXM, reg<nr>
TEXM <num>	**TB** EXM, num
MASK reg<nr>	**SB** EXM, reg<nr>
MASK <num>	**SB** EXM, <num>
ENBL <num>	**RB** EXM, reg<nr>
ENBL <num>	**RB** EXM, <num>
SBNK <num> (see remark 10 below)	**COPY** TMP, SCR **SB/RB** TMP, 4 **SB/RB** TMP, 5 **SB/RB** TMP, 6 **SB/RB** TMP, 7 **COPY** SCR, TMP

Several remarks are in order.

1. Options to pseudo-mnemonics can be also applied (e.g., `ABS_B reg3`).
2. The numerical argument <num> can be also specified by symbolic names, like the enumeration type in the C programming language.
3. The aliases of the special registers (see Table 12) are used in Table 71.
4. The *Translation to natives* column in Table 71 may contain single pseudo-instructions which are to be firstly translated to native instructions according to Table 70. However, they are left in the column as pseudo-instructions for the sake of clarity and compactness.
5. The <ALU_OP> token in Table 71 represents an instruction mnemonic (including any applicable option) of any arithmetic/logic instruction that uses the IMMEDIATE14 field.
6. The <BRANCH> in Table 71 token represents an instruction mnemonic (including any applicable option) of any branch instruction.
7. For the **LIMM** reg<nr>, <num> instruction in which the width of <num> is greater than 18 bits, a sequence of instructions is computed. For example, assuming a 32-bit GPR width, `LIMM_W reg1, 0xAAAABBBB` will be translated to:

   ```
   MOV_H reg1, 0xAAAA
   SL_W  reg1, 16
   MOV_H reg1, 0xBBBB
   ```

8. The instructions in Table 71
   LIMM reg<nr>, <num>,
   <**ALU_OP**> reg<nr>, <num>,
   MAD/MSU reg<nr>, reg<nr2>, <num>,
   JMP/WAIT <num> and
   <**BRANCH**> reg<nr>, <num>
   are actually multiple pseudo-instructions only if the binary width of the numerical argument <num> is greater than the width of the immediate field of the instruction, i.e., greater than 18, 14, 8, 20 and 14 bits for IMMEDIATE18, IMMEDIATE14, IMMEDIATE8, OFFSET20 and OFFSET14, respectively.

9. The TB pseudo-instructions whose destination pseudo-argument is a special register can be freely executed in user mode since the translation to natives does not involve writing to special registers.
10. In the SBNK pseudo-instruction the correct SB or RB instruction for the 4-th, 5-th, 6-th and 7-th bit is determined according to the bit values of ‹num› at positions 0, 1, 2 and 3, correspondingly.
11. Some characteristics of the assembly are the following:
    - all mnemonics and pseudo-mnemonics have two to four letters;
    - only FP instructions begin with **F**;
    - only branch instructions begin with **B**;
    - system instructions can also have a **.S** suffix added to the mnemonic (or pseudo-mnemonic) for the purposes of visual differentiation of system instructions, e.g., `COPY SCR, reg3` could be also written as `COPY.S SCR, reg3`. Options are added behind the suffix, e.g., `COPY.S_B SCR, reg3`.
12. The following hint can be used for quick decoding of the meaning of the branch mnemonics. The letters in the mnemonic denote:

    B – Branch if

    Z – Zero

    N – Not

    M – MSB

    L – LSB

    O – all Ones

Generally, care should be taken when using multiple pseudo-instructions that translate into natives which use the TMP register since the TMP register may be also used in other program routines.

6.5 Examples

Example 94: *Procedural transfer and return from procedure*

Listings 1 and 2 show code segments illustrating procedural program transfers and returning from procedures. BL_P investigates the LSB of reg2 and finds that the branch condition is met (since previously MOV put 0xF in reg2). Thus, a relative program transfer to the proced location[5] is done, jumping the load immediate instructions at Lines 3 and 4. Under the assumption that the instruction in Line 1 is at address 0x0, the BL_P instruction will write the CALL RETURN POINTER with 0x2, i.e., with the address of the second instruction at address 0x8 (Line 3).

Listing 1: *Procedural program transfer and return after the procedure call*

```
1   limm reg2, 0xF      //load immediate to reg2 (0xF)
2   bl_p reg2, proced   //branch is taken
3   limm reg3, 5        // <- return point
4   limm reg4, 100
5
```

[5] proced is a location label which is replaced by a number and is therefore used directly to branch according to offset by BL_P.

```
6   proced:
7      add reg0, 5     //next instruction executed after bl_p
8      add reg1, 3
9      ret             //ret returns according to the CRP value after bl_p
```

After execution of the add instructions in Lines 7 and 8, the RET instruction is executed, and the INSTRUCTION COUNTER is written with the value of the CALL RETURN POINTER (0x2). So, the next executed instruction after RET is the load immediate of reg0 in Line 3. In other words, the order of executed instructions according to their line numbers in Listing 1 is: 1, 2, 7, 8, 9, 3, 4.

If now the RET instruction is changed to RET_P (only the P bit is changed and set to 1, see Example 88), a return to the call is done, i.e., to the **P**revious instruction. In Listing 2, only the RET instruction is changed to RET_P compared to Listing 1.

Listing 2: *Procedural program transfer and return at the procedure call*

```
1   limm reg2, 0xF     //load immediate to reg2 (0xF)
2   bl_p reg2, proced  //branch is taken, here is also the <- return point
3   limm reg3, 5
4   limm reg4, 100
5
6   proced:
7      add reg0, 5     //next instruction executed after bl_p
8      add reg1, 3
9      ret_p           //ret returns according to the CRP value after bl_p
```

Thus, instead of returning at the load immediate instruction at Line 3, RET_P reduces the value of the CALL RETURN POINTER by one and returns again to the BL_P instruction. The order of the executed instructions according to line numbers in Listing 2 will be: 1, 2, 7, 8, 9, 2, 7, 8, 9, 2, 7, 8, 9, ... That is, a loop consisting of the instructions 2, 7, 8 and 9 is formed. If required, one way to break the loop is to change the LSB of reg2 within the proced procedure. Thus, the branch condition of BL_P will be false and instructions 3 and 4 will be executed.

Example 95: *Exception handling*

Listing 3 shows a skeleton of exception handling flow. At the beginning, the EXCEPTION TABLE BASE ADDRESS register (ETB) is set. The label of the address of the exception handler dispatcher is EXC_dispatcher, which is shifted by 2 places right (divided by 4) before it is written to the ETB. This has to be done because the INSTRUCTION COUNTER is automatically overwritten with the value of the ETB when handling of a potent exception is entered.

Listing 3: *Exception handling*

```
1   limm  reg0, EXC_dispatcher >> 2
2   copy  etb, reg0    //set EXCEPTION TABLE BASE ADDRESS
3   sb    reg1, 2000   //raises INVALID OPERATION exception
4   limm  reg3, -1
5   limm  reg4, -1
6   add_u reg3, reg4   //raises OVERFLOW exception
7   ee   //enable exceptions
8   nop  // <- return point from exception handlers
9   /*
10     <CONTINUE PROGRAM HERE>
11  */
12
13  EXC_dispatcher:
14     tb exc, 7                 //test if INVALID OPERATION exception
15     bl tmp, invalid_op_hndl   // ... and branch if so
16     tb exc, 9                 //else test if OVERFLOW exception
17     bl tmp, overflow_hndl     // ... and branch if so
18     /*
19        <TEST AND BRANCH TO OTHER HANDLERS HERE (PRIORITY DESCENDING)>
20     */
21     rete  //theoretically, this instruction should be never executed
```

```
22
23    invalid_op_hndl:
24       /*
25          <HANDLE INVALID OPERATION EXCEPTION HERE>
26       */
27       rb exc, 7  //acknowledge INVALID OPERATION exception
28       rete
29
30    overflow_hndl:
31       /*
32          <HANDLE OVERFLOW EXCEPTION HERE>
33       */
34       rb exc, 9  //acknowledge OVERFLOW exception
35       rete
36
37    /*
38       <OTHER EXCEPTION HANDLERS HERE>
39    */
```

Assuming that at the beginning none of the exceptions is masked, but they are all disabled through the ENABLE EXCEPTIONS bit in the SYSTEM CONTROL REGISTER, the SB instruction at Line 3 raises the INVALID OPERATION exception which is now impotent. Therefore, its handling will not be immediate, but postponed. The GPR 1 is not changed by the SB instruction, however, the EXECUTION STATUS, the EXCEPTION INSTRUCTION and the EXCEPTION REGISTER are updated accordingly. Afterwards, execution continues at Line 4.

Similarly, the ADD_U instruction at Line 6 will raise the OVERFLOW exception which is also impotent and its handling will be postponed. The GPR 3 is not changed by the ADD_U instruction, however, the EXECUTION STATUS, the EXCEPTION INSTRUCTION and the EXCEPTION REGISTER are updated accordingly. Afterwards, execution continues at Line 7.

The EE pseudo-instruction is translated to `SB SCR, 2` (according to Table 71), which is also a multiple pseudo-instruction finally translated to the following native instructions:

```
COPY tmp, scr
SB   tmp, 2
COPY scr, tmp
```

where the COPY single pseudo-instruction is simply replaced by MOV according to Table 70, and tmp is an alias of a GPR (see Subsection 6.4). Thus, the EE instruction is translated to these three instructions. As soon as the last one is fully executed (`COPY scr, tmp`), the exceptions are enabled and the program is immediately transferred to the address of the exception handler dispatcher EXC_dispatcher which is pointed by the EXCEPTION TABLE BASE ADDRESS register.

In EXC_dispatcher, each bit of the EXCEPTION REGISTER is tested with the TB instruction and if the bit is set, a program transfer to the corresponding handler is made with the BL instruction. In this example, for simplicity, only the INVALID OPERATION and OVERFLOW exceptions are shown which are exception 7 and 9 according to Table 11. Each exception handler after handling the exception, acknowledges it by resetting its corresponding bit in the EXCEPTION REGISTER with the RB instruction. At the end, the RETE instruction of the handler returns execution to the NOP instruction at Line 8. Thus, the RETE instruction at Line 21 in the EXC_dispatcher should be theoretically never executed.

Thus, after handling the INVALID OPERATION exception, the EXC_dispatcher is again re-entered at Line 14 (NOP at Line 8 is not executed yet) for handling the OVERFLOW exception since the EXCEPTION REGISTER is still not zero. Then, finally execution is continued at Line 8.

Now, if the branch instructions BL in EXC_dispatcher are made procedural (BL_P), and all the RETE instructions in the exception handlers are replaced by RET instructions, an improved version of exception handling is obtained since the EXC_dispatcher is not re-entered but all raised exceptions are handled sequentially according to their descending priority order. Listing 4 shows this version.

Listing 4: *Exception handling (improved flow)*

```
1  limm  reg0, EXC_dispatcher >> 2
2  copy  etb, reg0    //set EXCEPTION TABLE BASE ADDRESS
3  sb    reg1, 2000   //raises INVALID OPERATION exception
4  limm  reg3, -1
5  limm  reg4, -1
6  add_u reg3, reg4   //raises OVERFLOW exception
7  ee
8  nop  // <- return point from EXC_dispatcher
9  /*
10    <CONTINUE PROGRAM HERE>
11 */
12
13 EXC_dispatcher:
14    tb   exc, 7                 //test if INVALID OPERATION exception
15    bl_p tmp, invalid_op_hndl   // ... and branch if so
16    tb   exc, 9                 //else test if OVERFLOW exception
17    bl_p tmp, overflow_hndl     // ... and branch if so
18    /*
19       <TEST AND BRANCH TO OTHER HANDLERS HERE (PRIORITY DESCENDING)>
20    */
21    rete
22
23 invalid_op_hndl:
24    /*
25       <HANDLE INVALID OPERATION EXCEPTION HERE>
26    */
27    rb exc, 7  //acknowledge INVALID OPERATION exception
28    ret
29
30 overflow_hndl:
31    /*
32       <HANDLE OVERFLOW EXCEPTION HERE>
33    */
34    rb exc, 9  //acknowledge OVERFLOW exception
35    ret
36
37 /*
38    <OTHER EXCEPTION HANDLERS HERE>
39 */
```

Now the RETE instruction at Line 21 is executed after the handlers of all raised exceptions are executed, and program execution continues at Line 8 (the NOP instruction).

Example 96: *Wait (indefinitely) for peripheral interrupts*

Listing 5 shows an example where the program waits for an interrupt before continuing execution. At the beginning, the INTERRUPT TABLE BASE ADDRESS register (ITB) is set. The label of the address of the interrupt handler dispatcher is IRQ_dispatcher, which is shifted by 2 places right (divided by 4) before it is written to the ITB. This has to be done because the INSTRUCTION COUNTER is automatically overwritten with the value of the ITB when handling of a potent interrupt is entered. Then, after some program-specific system initialization (in system mode) like configuration of the interrupt controller, the interrupt line is enabled and the system is switched to user mode. Now, after some program-specific initialization in user mode, the system is instructed to wait for an interrupt, e.g., from an external IO device.

Listing 5: *Wait and handle a single interrupt*

```
1  limm reg0, IRQ_dispatcher >> 2
2  copy itb, reg0  // set INTERRUPT TABLE BASE ADDRESS
3  /*
4     <CONFIGURE INTERRUPT CONTROLLER>
5     <OTHER SYSTEM INITIALIZATION HERE>
6  */
7  ei    // enable the interrupt line
8  usrm  // switch to user mode
9  /*
```

```
10      <USER INITIALIZATION HERE>
11   */
12   wait  //wait for an interrupt
13   nop   //<- return point from IRQ_dispatcher
14   /*
15      <CONTINUE PROGRAM HERE>
16   */
17
18   IRQ_dispatcher:
19      /*
20         <INSPECT THE INTERRUPT CONTROLLER WHICH INTERRUPT(S) WERE RAISED AND
21          JUMP TO THE HIGHEST-PRIORITY INTERRUPT>
22      */
23      call IRQ_handler  //jump to the corresponding interrupt handler
24      usrm  //return to user mode (<- return point from IRQ_handler)
25      reti  //return from interrupt handler
26
27   IRQ_handler:
28      /*
29         <HANDLE INTERRUPT HERE>
30         <ACKNOWLEDGE INTERRUPT HERE>
31      */
32      ret   //return to the place of the dispatcher call
```

Thus, the program execution is paused indefinitely, until a potent interrupt is raised. The dispatcher `IRQ_dispatcher` inspects the interrupt controller and selects the (highest-priority) interrupt. Under the assumption that the interrupt selected for handling has a handler at address `IRQ_handler`, the dispatcher calls (jumps procedurally to) that address. After handling and acknowledging the interrupt, the program is transferred back to the dispatcher after the `CALL` (jump) instruction, i.e., at the `usrm` pseudo-instruction at Line 24. Thus, user mode is switched back since system mode was automatically switched upon entering interrupt handling, and, on the other side, `RETI` does not switch back the mode like `RETN` and `RETE` (see Subsection 4.4). Finally, the `RETI` instruction returns program execution to the `NOP` instruction at Line 13, continuing program execution.

Now, if only the `RETI` instruction at Line 25 is changed to `RETI_P` (see Listing 6), a different program flow is obtained, i.e., the program execution is returned again to the `WAIT` instruction, waiting again (indefinitely) for the next interrupt.

Listing 6: *Wait and handle interrupts*

```
1    limm reg0, IRQ_dispatcher >> 2
2    copy itb, reg0  // set INTERRUPT TABLE BASE ADDRESS
3    /*
4       <CONFIGURE INTERRUPT CONTROLLER>
5       <OTHER SYSTEM INITIALIZATION HERE>
6    */
7    ei    // enable the interrupt line
8    usrm  // switch to user mode
9    /*
10      <USER INITIALIZATION HERE>
11   */
12   wait  //wait for interrupts here (<- return point from IRQ_dispatcher)
13   nop   //this (and after this) instruction will not be executed
14
15   IRQ_dispatcher:
16      /*
17         <INSPECT THE INTERRUPT CONTROLLER WHICH INTERRUPT(S) WERE RAISED AND
18          JUMP TO THE HIGHEST-PRIORITY INTERRUPT>
19      */
20      call IRQ_handler  //jump to the corresponding interrupt handler
21      usrm  //return to user mode (<- return point from IRQ_handler)
22      reti_p  //return from interrupt handler
23
24   IRQ_handler:
25      /*
26         <HANDLE INTERRUPT HERE>
27         <ACKNOWLEDGE INTERRUPT HERE>
28      */
29      ret   //return to the place of the dispatcher call
```

List of Acronyms

ALU	Arithmetic/Logic Unit
BTU	Brandenburgische Technische Universität
DAAD	Deutscher Akademischer Austauschdienst
DSP	Digital Signal Processing
FEEIT	Faculty of Electrical Engineering and Information Technologies
FFT	Fast Fourier Transform
FP	Floating Point
FPR	Floating Point Register
FPU	Floating Point Unit
GPR	General-Purpose Register
IO	Input/Output
IRQ	Interrupt Request
ISA	Instruction Set Architecture
LSB	Least Significant Bit
MMU	Memory Management Unit
MPU	Memory Protection Unit
MSB	Most Significant Bit
NaN	Not a Number
NMI	Non-Maskable Interrupt
RISC	Reduced Instruction Set Computer

List of Figures

1 Graphical representations of registers, instructions and bit-fields 13
2 Registers operating in lower machine modes 14
3 Read/write of data wider than the register file width 14
4 Read/write of wider data in the last register 15
5 Register circularity 15
6 Layout of data transfer instructions 21
7 Layout of arithmetic/logic instructions 24
8 Layout of control instructions 27
9 NMI handling 33
10 Exception handling 35
11 Interrupt handling 43
12 Hierarchy of NMI, exceptions and interrupts 45
13 Multiple exceptions raised simultaneously 45
14 Postponed execution of exception and interrupt handlers 46
15 Nesting exceptions and interrupts 47
16 IMPLEMENTATION REGISTER 49
17 EXECUTION STATUS 50
18 EXCEPTION INSTRUCTION 51
19 EXCEPTION REGISTER 51
20 EXCEPTION MASKS 52
21 EXCEPTION TABLE BASE ADDRESS 52
22 INTERRUPT TABLE BASE ADDRESS 52
23 CORE ID 53
24 PROCESS ID 53
25 SYSTEM CONTROL REGISTER 53
26 NMI RETURN POINTER 54
27 EXCEPTION RETURN POINTER 54
28 USER CONTROL REGISTER 54
29 CALL RETURN POINTER 55
30 INTERRUPT RETURN POINTER 56
31 DSP CONFIGURATION REGISTER 56
32 Move data (MOV) instructions 58
33 Add (ADD) instructions 64
34 Subtract (SUB) instructions 67

35 Multiply (MUL) instructions . . . 70
36 Divide (DIV) instructions . . . 73
37 Shift left (SL) instructions . . . 77
38 Shift right (SR) instructions . . . 80
39 Rotate left (RL) instructions . . . 83
40 Rotate right (RR) instructions . . . 86
41 AND bitwise (AND) instructions . . . 89
42 Negated AND bitwise (NAND) instructions . . . 92
43 OR bitwise (OR) instructions . . . 95
44 Exclusive OR bitwise (XOR) instructions . . . 98
45 Set bit (SB) instructions . . . 101
46 Reset bit (RB) instructions . . . 104
47 Test bit (TB) instructions . . . 107
48 Reverse bits (RVB) instructions . . . 110
49 FP Add (FADD) instructions . . . 113
50 FP Subtract (FSUB) instructions . . . 115
51 FP Multiply (FMUL) instructions . . . 117
52 FP Divide (FDIV) instructions . . . 119
53 FP Remainder (FREM) instructions . . . 121
54 FP Compare (FCMP) instructions . . . 123
55 FP Square root (FSQR) instructions . . . 125
56 FP Absolute (FABS) instructions . . . 127
57 FP Negate (FNEG) instructions . . . 129
58 FP Round to integer (FRND) instructions . . . 131
59 FP to integer (FF2I) instructions . . . 133
60 Integer to FP (FI2F) instructions . . . 135
61 Extend FP format (FEXT) instructions . . . 137
62 Squeeze FP format (FSQZ) instructions . . . 139
63 Multiply-add (MAD) instructions . . . 141
64 Multiply-subtract (MSU) instructions . . . 144
65 FP Multiply-add (FMAD) instructions . . . 147
66 FP Multiply-subtract (FMSU) instructions . . . 149
67 Jump (JMP) instructions . . . 151
68 Branch if Zero (BZ) instructions . . . 154
69 Branch if Not Zero (BNZ) instructions . . . 157
70 Branch if MSB (BM) instructions . . . 160
71 Branch if MSB or Zero (BMZ) instructions . . . 163
72 Branch if Not MSB (BNM) instructions . . . 166

73 Branch if Not MSB or all Ones (BNMO) instructions 169
74 Branch if LSB (BL) instructions 172
75 Branch if LSB or Zero (BLZ) instructions 175
76 Branch if Not LSB (BNL) instructions 178
77 Branch if Not LSB or all Ones (BNLO) instructions 181
78 Branch if all Ones (BO) instructions 184
79 Branch if Not all Ones (BNO) instructions 187
80 Return from procedure (RET) instructions 190
81 Return from interrupt handler (RETI) instructions 192
82 Return from exception handler (RETE) instructions 194
83 Return from NMI handler (RETN) instructions 196
84 Wait (WAIT) instructions 198

List of Tables

1 Machine modes . . . 12
2 Special registers . . . 16
3 Bit field description of data transfer instructions . . . 22
4 Sign/zero extension and truncation of an immediate value according to MMODE . . . 24
5 Bit field description of arithmetic/logic instructions . . . 25
6 Bit field description of control instructions . . . 27
7 Return pointers used by '*return from routine*' instructions . . . 28
8 Summary of data transfer (MOV) instructions . . . 30
9 Summary of arithmetic/logic instructions . . . 31
10 Summary of control instructions . . . 32
11 Exceptions . . . 35
12 Recommended register aliases and access permissions in user/system mode of a 32-bit wide special register file . . . 48
13 Arithmetic/logic operator symbols . . . 57
14 Execution of MOV instructions . . . 60
15 Execution of ADD instructions . . . 64
16 Execution of SUB instructions . . . 67
17 Execution of MUL instructions . . . 70
18 Execution of DIV instructions . . . 73
19 Execution of SL instructions . . . 77
20 Execution of SR instructions . . . 80
21 Execution of RL instructions . . . 83
22 Execution of RR instructions . . . 86
23 Execution of AND instructions . . . 89
24 Execution of NAND instructions . . . 92
25 Execution of OR instructions . . . 95
26 Execution of XOR instructions . . . 98
27 Execution of SB instructions . . . 101
28 Execution of RB instructions . . . 104
29 Execution of TB instructions . . . 107
30 Execution of RVB instructions . . . 110
31 Execution of FADD instructions . . . 113
32 Execution of FSUB instructions . . . 115
33 Execution of FMUL instructions . . . 117

34 Execution of FDIV instructions . . . 119
35 Execution of FREM instructions . . . 121
36 Execution of FCMP instructions . . . 123
37 Execution of FSQR instructions . . . 125
38 Execution of FABS instructions . . . 127
39 Execution of FNEG instructions . . . 129
40 Execution of FRND instructions . . . 131
41 Execution of FF2I instructions . . . 133
42 Execution of FI2F instructions . . . 135
43 Execution of FEXT instructions . . . 137
44 Execution of FSQZ instructions . . . 139
45 Execution of MAD instructions . . . 141
46 Execution of MSU instructions . . . 144
47 Execution of FMAD instructions . . . 147
48 Execution of FMSU instructions . . . 149
49 Execution of JMP instructions . . . 151
50 Execution of BZ instructions . . . 154
51 Execution of BNZ instructions . . . 157
52 Execution of BM instructions . . . 160
53 Execution of BMZ instructions . . . 163
54 Execution of BNM instructions . . . 166
55 Execution of BNMO instructions . . . 169
56 Execution of BL instructions . . . 172
57 Execution of BLZ instructions . . . 175
58 Execution of BNL instructions . . . 178
59 Execution of BNLO instructions . . . 181
60 Execution of BO instructions . . . 184
61 Execution of BNO instructions . . . 187
62 Execution of RET instructions . . . 190
63 Execution of RETI instructions . . . 192
64 Execution of RETE instructions . . . 194
65 Execution of RETN instructions . . . 196
66 Execution of WAIT instructions . . . 198
67 Formats of numerical arguments . . . 203
68 Instruction options and arguments . . . 204
69 Assembly of the example instructions in Subsection 6.1 . . . 204
70 Single pseudo-instructions . . . 208
71 Multiple pseudo-instructions . . . 208

List of Examples

1 Load halfword from memory with displacement addressing 60
2 Store word in memory with register addressing 61
3 Load doubleword from memory with indexed addressing 61
4 Inter-register transfer from a GPR to a special register 62
5 Load immediate 62
6 Add register 65
7 Add immediate 65
8 Subtract register 68
9 Subtract immediate 68
10 Multiply register (word machine mode) 71
11 Multiply register (halfword machine mode) 71
12 Multiply immediate 72
13 Divide register (word machine mode) 74
14 Divide register (halfword machine mode) 75
15 Divide immediate 75
16 Shift left register 78
17 Shift left immediate 78
18 Shift right register 81
19 Shift right immediate 81
20 Rotate left register 84
21 Rotate left immediate 84
22 Rotate right register 87
23 Rotate right immediate 87
24 AND bitwise register 90
25 AND bitwise immediate 90
26 NAND bitwise register 93
27 NAND bitwise immediate 93
28 OR bitwise register 96
29 OR bitwise immediate 96
30 XOR bitwise register 99
31 XOR bitwise immediate 99
32 Set bit register 102
33 Set bit immediate 102
34 Reset bit register 105

35 Reset bit immediate . . . 105
36 Test bit register . . . 108
37 Test bit immediate . . . 108
38 Reverse bits register . . . 111
39 Reverse bits immediate . . . 111
40 FP Add . . . 114
41 FP Subtract . . . 116
42 FP Multiply . . . 118
43 FP Divide . . . 120
44 FP Remainder . . . 122
45 FP Compare . . . 123
46 FP Square root . . . 126
47 FP Absolute . . . 128
48 FP Negate . . . 129
49 FP Round to integer . . . 132
50 FP to integer . . . 134
51 Integer to FP . . . 136
52 Extend FP format . . . 138
53 Squeeze FP format . . . 140
54 Multiply-add register . . . 142
55 Multiply-add immediate . . . 142
56 Multiply-subtract register . . . 145
57 Multiply-subtract immediate . . . 145
58 FP Multiply-add . . . 148
59 FP Multiply-subtract . . . 150
60 Jump relative according to register . . . 152
61 Jump absolute according to register . . . 152
62 Jump relative according to offset . . . 152
63 Jump absolute according to offset . . . 153
64 Branch if Zero according to register . . . 155
65 Branch if Zero according to offset . . . 155
66 Branch if Not Zero according to register . . . 158
67 Branch if Not Zero according to offset . . . 158
68 Branch if MSB according to register . . . 161
69 Branch if MSB according to offset . . . 161
70 Branch if MSB or Zero according to register . . . 164
71 Branch if MSB or Zero according to offset . . . 164
72 Branch if Not MSB according to register . . . 167

73 Branch if Not MSB according to offset . . . 167
74 Branch if Not MSB or all Ones according to register . . . 170
75 Branch if Not MSB or all Ones according to offset . . . 170
76 Branch if LSB according to register . . . 173
77 Branch if LSB according to offset . . . 173
78 Branch if LSB or Zero according to register . . . 176
79 Branch if LSB or Zero according to offset . . . 176
80 Branch if Not LSB according to register . . . 179
81 Branch if Not LSB according to offset . . . 179
82 Branch if Not LSB or all Ones according to register . . . 182
83 Branch if Not LSB or all Ones according to offset . . . 182
84 Branch if all Ones according to register . . . 185
85 Branch if all Ones according to offset . . . 185
86 Branch if Not all Ones according to register . . . 188
87 Branch if Not all Ones according to offset . . . 188
88 Return from procedure . . . 190
89 Return from interrupt handler . . . 192
90 Return from exception handler . . . 194
91 Return from NMI handler . . . 196
92 Wait according to register . . . 199
93 Wait according to offset . . . 199
94 Procedural transfer and return from procedure . . . 211
95 Exception handling . . . 212
96 Wait (indefinitely) for peripheral interrupts . . . 214

List of Listings

1 Procedural program transfer and return after the procedure call 211
2 Procedural program transfer and return at the procedure call 212
3 Exception handling 212
4 Exception handling (improved flow) 214
5 Wait and handle a single interrupt 214
6 Wait and handle interrupts 215

References

[1] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, pp. 412-421, 1974

[2] J. H. Anderson and M. Moir. Universal Constructions for Multi-object Operations. *Proceedings of the 14-th Annual ACM Symposium on Principles of Distributed Computing*, pp. 184-193, 1995

[3] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008

Index

ABS, 208
ADD, 31, 39, 50, 64, 201, 204, 213
AND, 31, 89, 201, 204
ARGUMENT, 27, 28, 154, 155, 157, 158, 160, 161, 163, 164, 166, 167, 169, 170, 172, 173, 175, 176, 178, 179, 181, 182, 184, 185, 187, 188
AUXCODE, 21–23, 25, 27, 29–32, 58–62
BASE, 22, 58, 59, 61, 62
BLZ, 32, 175, 204
BL, 32, 172, 204, 211–213
BMZ, 28, 32, 163, 204
BM, 32, 160, 204
BNLO, 32, 181, 204
BNL, 32, 178, 204
BNMO, 32, 169, 204
BNM, 32, 166, 204
BNO, 28, 32, 187, 204
BNZ, 28, 32, 157, 187, 204
BO, 28, 32, 184, 204
BZ, 28, 32, 154, 204
CALL RETURN POINTER, 16, 18, 28, 48, 55, 151–190, 201, 211, 212
CALL, 207, 208, 215
COPY, 208, 213
CORE ID, 16, 20, 48, 53
CSYN, 208
D SYSTEM BUS ERROR, 35, 42
DBGM, 208
DEBUG MODE EXCEPTION, 35, 36, 53
DEBUG MODE, 36, 53
DECE, 208
DECR, 208
DENORMALIZED, 51, 114
DESTINATION, 22–25, 31, 58, 59, 61–68, 70–75, 77–81, 83, 84, 86, 87, 89, 90, 92, 93, 95, 96, 98, 99, 101, 102, 104, 105, 107, 108, 110, 111, 113–129, 131–150
DE, 208
DIVISION BY ZERO, 35, 38, 50, 74
DIV, 31, 38, 39, 50, 73, 74, 201, 204
DI, 208
DONT BUFFER DATA, 55
DONT BUFFER INSTRUCTIONS, 55
DONT CACHE DATA, 55
DONT CACHE INSTRUCTIONS, 55
DSP CONFIGURATION REGISTER, 16, 48, 56
DSP EXCEPTION 0, 35, 41
DSP EXCEPTION 1, 35, 41
DSP EXCEPTION 2, 35, 42
DSP EXCEPTION 3, 35, 42
EACK, 208
EECE, 208
EE, 208, 213
EI, 208
ENABLE EXCEPTIONS, 36, 44, 45, 53, 194, 196, 213
ENABLE INTERRUPTS, 43–45, 54, 192, 194, 196
ENBL, 208
EQUAL, 51
EXCEPTION INSTRUCTION, 16, 34, 36–42, 48, 51, 66, 79, 103, 114, 120, 140, 148, 213
EXCEPTION MASKS, 16, 36, 46, 48, 52
EXCEPTION REGISTER, 16, 34–42, 44, 45, 48, 51, 66, 79, 103, 114, 120, 140, 148, 213
EXCEPTION RETURN POINTER, 16, 18, 28, 36, 46, 48, 54, 194
EXCEPTION TABLE BASE ADDRESS, 16, 18, 34, 36, 48, 52, 212, 213
EXECUTION STATUS, 16, 19, 25, 34, 38–41, 48, 50, 64–69, 71, 72, 74–76, 78, 79, 81, 84, 85, 87, 88, 90, 91, 93, 94, 96, 97, 99, 100, 102, 103, 105, 106, 108, 109, 111–140, 142, 143, 145–150, 213
FABS, 31, 127, 204
FADD, 31, 39–41, 113, 204
FCLS, 208
FCMP, 31, 51, 123, 204
FDIV, 31, 39–41, 50, 119, 204
FEXT, 26, 31, 137, 204
FF2I, 26, 31, 39–41, 133, 201, 204
FI2F, 26, 31, 40, 41, 135, 201, 204
FMAD, 26, 31, 39–41, 147, 202, 204
FMSU, 26, 31, 39–41, 149, 202, 204
FMUL, 31, 39–41, 117, 204
FNEG, 31, 129, 204
FP DENORMALIZED OPERAND, 35, 40, 113–115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 137, 139, 147, 149
FP DIVISION BY ZERO, 35, 40, 119
FP INEXACT RESULT, 35, 41, 114, 116, 118, 120, 125, 132, 134, 135, 140, 148, 150

FP INVALID OPERATION, 35, 39, 50, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 137, 139, 147, 149
FP OVERFLOW, 35, 40, 41, 113–120, 125, 131, 132, 134, 135, 139, 140, 148, 150
FP PRECISION, 55
FP ROUNDING MODE, 55
FP UNDERFLOW, 35, 41, 113–118, 120, 125, 126, 132, 134, 135, 140, 148, 150
FREM, 31, 39, 121, 204
FRND, 31, 40, 41, 131, 204
FSQR, 31, 39, 41, 125, 204
FSQZ, 26, 31, 40, 41, 139, 204
FSUB, 31, 39–41, 115, 204
GPR BANK, 37, 54
GREATER THAN, 51, 66, 68, 69, 71, 72, 74–76, 78, 79, 81, 84, 85, 87, 88, 91, 97, 100, 105, 106, 108, 109, 111, 112, 114, 142, 143, 145, 146
I SYSTEM BUS ERROR, 35, 42
I8HI, 24, 25, 141, 143, 144, 146
I8LO, 24, 25, 141, 143, 144, 146
IMMEDIATE14, 24–26, 64–68, 70, 72, 73, 75, 77, 78, 80, 81, 83, 84, 86, 87, 89, 90, 92, 93, 95, 96, 98, 99, 101, 102, 104, 105, 107, 108, 110, 111, 210
IMMEDIATE18, 21, 22, 24, 58, 59, 63, 210
IMMEDIATE8, 24–26, 141, 143, 144, 146, 210
IMPLEMENTATION REGISTER, 16, 48, 49, 202
INCR, 208
INDEX, 22, 23, 30, 58, 59, 61, 62
INEXACT, 51, 114, 120, 140, 148
INFINITY, 51
INSTRUCTION COUNTER, 17, 18, 28, 34, 36, 43, 53, 151–190, 192–199, 201, 212, 214
INTERRUPT RETURN POINTER, 16, 28, 43, 46, 48, 56, 192
INTERRUPT TABLE BASE ADDRESS, 16, 18, 43, 48, 52, 214
INVALID INSTRUCTION, 35, 37
INVALID OPERATION, 35, 38, 49, 50, 60, 102, 103, 105, 108, 111, 213
JMP, 29, 151, 204
LESS THAN, 51, 65, 72, 90, 93, 94, 96, 99, 102, 103, 116, 118, 120, 122, 148, 150
LIMM, 207, 208
LOAD, 208
LOCATION, 27, 28, 151, 152, 154, 155, 157, 158, 160, 161, 163, 164, 166, 167, 169, 170, 172, 173, 175, 176, 178, 179, 181, 182, 184, 185, 187, 188, 198, 199
MAD, 26, 31, 141, 201, 202, 204
MASK, 208
MEM, 208
MMODE, 12, 21–29, 31, 32, 59, 61–65, 67, 68, 70–75, 77, 78, 80, 81, 83, 84, 86, 87, 89, 90, 92, 93, 95, 96, 98, 99, 101, 102, 104, 105, 107, 108, 110, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133–147, 149, 154, 155, 157, 158, 160, 161, 163, 164, 166, 167, 169, 170, 172, 173, 175, 176, 178, 179, 181, 182, 184, 185, 187, 188
MOV, 29, 30, 37, 38, 45, 58–60, 204, 211, 213
MSU, 26, 31, 144, 201, 202, 204
MUL, 31, 50, 70, 201, 204
NAND, 31, 92, 201, 204
NAN, 51, 124
NDBG, 208
NEG, 208
NMI RETURN POINTER, 16, 28, 34, 48, 54, 196
NOP, 208, 213–215
NOT, 208
OFFSET12HI, 21, 22, 58, 59, 61
OFFSET12LO, 21, 22, 58, 59, 61
OFFSET12, 21, 22, 59, 61
OFFSET14, 27, 28, 32, 154, 155, 157, 158, 160, 161, 163, 164, 166, 167, 169–173, 175, 176, 178, 179, 181–185, 187, 188, 210
OFFSET20HI, 27, 151, 153, 198, 199
OFFSET20LO, 27, 151, 153, 198, 199
OFFSET20, 27, 29, 32, 151, 153, 198, 199, 210
OPCODE, 21
OR, 31, 95, 201, 204
OVERFLOW, 26, 35, 39, 50, 65, 66, 68, 74, 77–79, 213
POP, 208
PROCESS ID, 16, 17, 19, 48, 50, 53
PUSH, 208
RB, 31, 38, 50, 104, 204, 208, 211, 213
RETE, 28, 32, 36, 37, 44, 45, 54, 192, 194, 200, 202, 204, 213–215
RETI, 28, 29, 32, 43–45, 56, 192, 202, 204, 215
RETN, 28, 29, 32–34, 37, 44, 45, 54, 192, 196, 200, 202, 204, 215
RET, 28, 32, 55, 190, 202, 204, 212, 213

RL, 31, 83, 204
RR, 31, 86, 204
RVB, 25, 26, 31, 38, 50, 110, 111, 204
SBNK, 208, 211
SB, 31, 38, 50, 101, 204, 208, 211, 213
SIGNALING NAN, 51, 124
SIGN, 51, 66, 72, 85, 93, 94, 96, 99, 105, 106, 112, 114, 116, 122, 130
SIMM, 208
SL, 26, 31, 39, 77, 201, 204
SOURCE2, 25, 26, 141, 142, 144, 145, 147–150
SOURCE, 25, 31, 64, 65, 67, 68, 70, 71, 73–75, 77, 78, 80, 81, 83, 84, 86, 87, 89, 90, 92, 93, 95, 96, 98, 99, 101, 102, 104, 105, 107, 108, 110, 111, 113–124, 141–150
SQR, 208
SR, 26, 31, 80, 201, 204
STOR, 208
SUB, 31, 39, 50, 67, 201, 204
SYNC, 20, 55, 208
SYSM, 208
SYSTEM CONTROL REGISTER, 16, 19, 36, 37, 43–45, 48, 49, 53, 55, 62, 192, 194, 196, 213
SYSTEM INSTRUCTION, 34, 35, 37, 48, 60, 194, 196, 200
SYSTEM MODE, 44, 49, 53, 194, 196
TB, 31, 38, 50, 107, 204, 208, 211, 213
TEXC, 208
TEXM, 208
UNDERFLOW, 41, 50
UNIMPLEMENTED GPR BANK, 35, 37, 54, 60
UNIMPLEMENTED INSTRUCTION, 22, 25, 35, 37, 50, 60, 65, 68, 71, 74, 78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 114, 116, 118, 120, 121, 123, 126, 127, 129, 132, 134, 135, 137, 140, 142, 145, 148, 150, 151, 155, 158, 161, 164, 167, 170, 173, 176, 179, 182, 185, 188, 190, 192, 194, 196, 199
UNIMPLEMENTED OPERATION, 50
UNIMPLEMENTED REGISTER, 35, 38, 49, 60, 65, 68, 71, 74, 78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 114, 116, 118, 120, 121, 123, 126, 127, 129, 132, 134, 135, 137, 140, 142, 145, 148, 150, 151, 155, 158, 161, 164, 167, 170, 173, 176, 179, 182, 185, 188, 199
UNORDERED, 51, 124
USER CONTROL REGISTER, 16, 20, 48, 54, 55
USRM, 208
WAIT, 19, 29, 32, 198, 202, 204, 208, 215
XOR, 31, 98, 201, 204
ZERO, 51, 109

absolute program transfer, 17, 27, 28, 152, 153, 155, 161, 167, 173, 179, 185, 201
address alignment, 17
address argument, 202–204, 207
address space, 16, 48
address translation, 17, 19
addressing mode, 11, 17–19, 58, 203
ALU, 12, 38, 49, 50, 102, 105, 108, 111
ALU width, 12, 38, 49, 50, 102, 105, 108, 111
argument, 12, 27, 29, 32, 38, 50, 154, 157, 160, 163, 166, 169, 172, 175, 178, 181, 184, 187, 200, 202–204, 207, 210, 211
arithmetic left shift, 26, 31, 39, 77, 78, 201
arithmetic right shift, 26, 31, 80, 81, 201
arithmetic shift, 26, 31, 39, 77, 78, 80, 81, 201
arithmetic/logic instruction, 3, 11, 12, 17, 19, 21, 24, 25, 29, 31, 38, 50, 51, 204, 210
arithmetic/logic operation, 12, 25, 50
arithmetic/logic operator symbols, 57
assembly, 3, 4, 11, 29, 57, 200–202, 207, 208, 211
atomic memory transfer, 19, 22, 23, 59, 201

base address, 17, 48, 52, 59, 203
base exception, 46
base register, 17, 21, 22, 30, 58–60, 203
big-endian, 18
branch, 12, 17, 19, 27, 28, 32, 154, 155, 157, 158, 160, 161, 163, 164, 166, 167, 169, 170, 172, 173, 175, 176, 178, 179, 181, 182, 184, 185, 187, 188, 202, 210, 211, 213

caller procedure, 55
circularity, 11, 14, 15, 48, 59, 62, 64, 67, 70, 74, 78, 81, 83, 87, 90, 93, 96, 99, 102, 105, 108, 111, 133, 141, 145
clock cycle, 29, 198, 199
compiler, 3, 11, 18
conditional program transfer, 17, 27, 28, 32, 154, 157, 160, 163, 166,

169, 172, 175, 178, 181, 184, 187
context-switch, 53
control instruction, 3, 21, 27–29, 32, 50, 204
control inter-dependency, 18, 19

data address, 17, 20, 21, 23
data addressing mode, 17, 58
data cache, 49, 55
data inter-dependency, 18
data transfer, 3, 12, 21–23, 29, 30, 50, 58, 59
data transfer instruction, 3, 12, 21, 22, 29, 30, 50, 58, 59, 204
data transfer width, 29
debug mode, 36, 37, 53
definite pause, 19, 29, 198, 199
disabled exceptions, 33, 34, 36, 43–46, 53, 194, 196
disabled interrupts, 33, 34, 36, 43–45, 54, 192, 194, 196
dispatcher, 33, 36, 44, 45, 52, 212–215
displacement addressing, 17, 22, 58–60, 203
DSP, 11, 13, 16, 25, 26, 34, 35, 38, 41, 42, 50, 56, 202, 207
DSP file, 13, 16, 38, 50
DSP instruction, 26
DSP register, 23, 26, 50, 202, 207
DSP unit, 11, 25, 26, 34, 35, 41, 42, 56
DSP width, 50

effective address, 20, 61, 62
enabled exceptions, 33, 34, 36, 43, 46, 53, 62, 194, 196, 213
enabled interrupts, 33, 34, 36, 43, 44, 54, 62, 192, 194, 196, 214
endianness, 14, 18, 48, 61, 62, 70, 142, 145
error line, 35, 42, 43
exception, 3, 11, 17–19, 22, 25, 26, 28, 29, 33–54, 57, 60, 65, 66, 68, 71, 74, 77–79, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 113–121, 123, 125–127, 129, 131–135, 137, 139, 140, 142, 145, 147–151, 155, 158, 161, 164, 167, 170, 173, 176, 179, 182, 185, 188, 190, 192, 194, 196, 199, 200, 212–214
exception acknowledgement, 36, 44, 51, 213
exception handler, 18, 19, 28, 32, 33, 35, 36, 44–46, 51, 54, 194, 200, 212–214
exception handler dispatcher, 33, 36, 44, 45, 52, 212, 213, 215
exception handling, 3, 17, 18, 29, 34–36, 44–46, 50–52, 66, 79, 114, 194, 212–214
exception nesting, 46, 47
exceptional instruction, 18, 29, 34, 37–42

FFT, 26
flag, 19, 50, 65, 66, 68, 69, 71, 72, 74–76, 78, 79, 81, 84, 85, 87, 88, 90, 91, 93, 94, 96, 97, 99, 100, 102, 103, 105, 106, 108, 109, 111, 112, 114, 116, 118, 120, 122, 124, 130, 140, 142, 143, 145, 146, 148, 150
FP, 12, 15, 16, 26, 35, 38–41, 50, 51, 113–140, 147–150, 201, 202, 211
FP format, 26, 40, 49, 113–132, 134–140, 147–150, 201
FP instruction, 12, 15, 16, 26, 31, 35, 38–41, 50, 51, 202, 211
FP machine mode, 12, 15, 26, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 137, 139, 147, 149
FP number, 40, 50, 51, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131–134, 137–140, 147, 149
FP operation, 40, 41, 51, 202
FP precision, 12, 16, 55
FP rounding mode, 40, 55, 131, 133
FP width, 26, 113, 115, 117, 119, 121, 125, 127, 129, 131, 135, 137, 139, 140, 147, 149
FPR, 13, 15, 16
FPR file, 13, 15, 16
FPU, 11, 12, 15, 16, 26, 49
FPU type, 49
frame pointer, 18
fused multiply-add, 25, 26, 31, 141, 142, 147, 148, 201, 202
fused multiply-subtract, 25, 26, 31, 144, 145, 149, 150, 201, 202

GPR, 11–18, 22, 23, 25–29, 34, 37–41, 49, 50, 54, 58, 59, 61–68, 70–75, 77–81, 83, 84, 86, 87, 89, 90, 92, 93, 95, 96, 98, 99, 101, 102, 104, 105, 107, 108, 110–189, 198–200, 202, 207, 210, 213
GPR bank, 37, 54, 60
GPR file, 13, 15, 16, 18, 23, 38, 58, 59, 61–63, 65, 68, 71–75, 78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 142, 143, 145, 146, 148, 150, 152,

153, 155, 158, 161, 164, 165, 167, 170, 171, 173, 176, 177, 179, 182, 183, 185, 186, 188, 199, 202
GPR width, 12, 14–16, 49, 62, 64, 67, 70, 71, 74, 77, 78, 80, 81, 83, 86, 87, 90, 93, 96, 99, 101, 102, 104, 105, 107, 108, 111, 113, 115, 117, 119, 121, 125, 127, 129, 131, 133, 135, 137, 139, 141, 142, 144, 145, 147, 149, 210

hardware-raised exception, 34–36, 41, 42, 51

IEEE Std 754-2008 standard, 26, 39, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 148, 150
implementation, 3, 11–13, 15–19, 22, 25, 37, 38, 48–50, 60, 65, 68, 71, 74, 78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 114, 116, 118, 120, 121, 123, 126, 127, 129, 132, 134, 135, 137, 140, 142, 145, 148, 150, 151, 155, 158, 161, 164, 167, 170, 173, 176, 179, 182, 185, 188, 190, 192, 194, 196, 199, 201, 202, 207
implementation-specific, 13, 16, 34, 41, 42, 51, 52, 56, 207
impotent exception, 33–35, 37–43, 45, 51, 52, 65, 66, 79, 114, 116, 118, 120, 126, 132, 134, 135, 140, 148, 150, 213
impotent interrupt, 33
indefinite pause, 19, 29, 198, 199, 214, 215
index register, 17, 21–23, 30, 58–60, 203
indexed addressing, 17, 22, 23, 58–61, 203
inexact result, 114, 116, 118, 120, 126, 132, 134, 135, 140, 148, 150
instruction address, 17, 28, 29, 151, 155, 158, 161, 164, 167, 170, 173, 176, 179, 182, 185, 188, 190, 192, 194, 196
instruction cache, 49, 55
instruction fetch, 18, 28, 29, 42
instruction option, 12, 18, 29, 200–204, 207, 210, 211
instruction-raised exception, 34, 35, 37–41
integer format, 26, 40, 133–136, 201
integer machine mode, 12, 15, 26, 59, 64, 67, 70, 73, 77, 80, 83, 86, 89, 92, 95, 98, 101, 104, 107, 110, 133, 135, 141, 144, 154, 157, 160, 163, 166, 169, 172, 175, 178, 181, 184, 187
integer unit, 26
integer width, 26, 154, 157, 160, 163, 166, 169, 172, 175, 178, 181, 184, 187
inter-dependency, 18, 19
inter-register transfer, 15, 16, 21, 23, 26, 29, 30, 37, 38, 45, 55, 58–60, 62, 200, 204
inter-register transfer instruction, 15, 16, 23, 29, 30, 37, 38, 45, 55, 58–60, 62, 200, 204
interrupt, 3, 11, 17–19, 28, 29, 33, 34, 36, 43–48, 52, 54, 192, 194, 196, 198, 199, 214, 215
interrupt acknowledgement, 43, 44, 215
interrupt controller, 33, 43, 44, 214, 215
interrupt handler, 18, 28, 32, 33, 43–46, 56, 192, 214, 215
interrupt handler dispatcher, 33, 52, 214, 215
interrupt handling, 3, 17, 18, 29, 43–46, 52, 192, 214, 215
interrupt line, 33, 34, 36, 43–45, 192, 194, 196, 214
interrupt nesting, 46, 47
interrupt request, 43
interrupt-requesting device, 43, 44
invalid instruction, 37
IO, 17, 19, 33, 42, 214
IO device, 17, 33, 42, 214
IRQ, 43
ISA, 3, 4, 11–14, 16, 18, 19, 21, 23, 34, 57, 201

little-endian, 14, 18, 48, 61, 62, 70, 142, 145
load from memory, 11, 19, 22, 23, 58–62, 138, 203, 204, 208
load immediate, 17, 21, 22, 24, 30, 58–60, 62, 201, 202, 204, 208, 211, 212
load-locked, 19, 23, 59
logic left shift, 26, 31, 39, 77, 78, 201
logic right shift, 26, 31, 80, 81, 201
logic shift, 26, 31, 39, 77, 78, 80, 81, 201
LSB, 13, 14, 17, 21, 28, 59, 102, 105, 108, 111, 141, 144, 151, 172–183, 198, 211, 212

machine mode, 11–17, 19, 22, 25–28, 38, 39, 49, 50, 58, 59, 62, 64–68, 70, 71, 73–75, 77, 78, 80, 83, 86, 89, 92, 95, 98, 101, 102, 104, 105, 107, 108, 110, 111,

113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 135, 137, 139, 141, 144, 147, 149, 154, 157, 160, 163, 166, 169, 172, 175, 178, 181, 184, 187, 200, 201, 203
masked exception, 33, 36, 37, 46, 52, 213
masked interrupt, 33, 43
maximal FP format, 137, 139
maximal GPR width, 16
maximal integer, 35, 39, 66, 133
maximal number of registers, 13–15
maximal transfer width, 22, 49
memory management unit, 17, 19, 42
memory protection unit, 17, 19, 42
memory transfer, 17, 21–23, 30, 58–60, 201, 203, 204
memory transfer instruction, 17, 22, 30, 58–60, 201, 203, 204
memory-mapped IO, 17
minimal GPR width, 16
minimal integer, 39, 74
MMU, 17, 19, 42
mnemonic, 11, 29–32, 58, 200, 210, 211
MPU, 17, 19, 42
MSB, 13, 21, 22, 24–26, 28, 39, 59, 77–80, 82, 102, 105, 108, 111, 141, 144, 151, 160–171, 198, 211
multiple pseudo-instruction, 207, 208, 210, 211, 213
multiprocessing, 11, 19, 20, 23, 53, 55

NaN, 39, 40, 51, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 137, 139, 147, 149
native instruction, 11, 30, 57, 207, 208, 210, 211, 213
natural machine mode, 11–13, 201, 203
nested exception, 46, 47
nested interrupt, 46, 47
nested NMI, 46, 47
nested procedure, 55
nesting, 46
NMI, 17–19, 28, 29, 33–36, 43–47, 51, 52, 196, 198–200
NMI acknowledgement, 34, 51
NMI handler, 18, 28, 32, 34, 44–46, 51, 54, 196, 200
NMI handling, 17, 18, 29, 33, 34, 44, 46, 51, 52, 196
NMI line, 33, 35
non-maskable interrupt, 33
numerical argument, 202–204, 207, 210

operating mode, 19, 34, 36, 38, 43, 48, 53, 194, 196
operating system, 3, 11, 19, 53, 200
orthogonality, 11, 18
out-of-order execution, 19

pause, 19, 27, 29, 32, 198, 199, 214, 215
pause period, 19, 29, 198, 199
PEAKTOP, 3, 4, 11–13, 16, 18, 19, 21, 34, 57, 58, 200, 201
physical address, 17, 48
physical address space, 48
physical address width, 49
portability, 18, 48, 201
post-decrement, 17, 23, 57, 59, 203
post-increment, 17, 23, 57, 59, 203
postponed handling, 45, 46, 213
potent exception, 19, 33–36, 43–46, 50, 66, 79, 114, 212
potent interrupt, 19, 33, 34, 36, 43, 44, 46, 198, 214, 215
pre-decrement, 17, 23, 57, 59, 203
pre-increment, 17, 23, 57, 59, 61, 62, 203
procedural program transfer, 27, 28, 55, 152, 153, 155, 156, 158, 159, 161, 162, 164, 165, 167, 168, 170, 171, 173, 174, 176, 177, 179, 180, 182, 183, 185, 186, 188, 189, 202, 211
procedure, 18, 33, 35, 43, 190, 211, 212
process, 19, 53
processor state, 33, 35, 42, 43
program compatibility, 18, 202
program routine, 33, 35, 43
program transfer, 17, 18, 27, 28, 32, 34, 36, 43, 55, 151–155, 157, 158, 160, 161, 163, 164, 166, 167, 169, 170, 172, 173, 175, 176, 178, 179, 181, 182, 184, 185, 187, 188, 201, 202, 211, 213, 215
programming language, 4, 210
pseudo-argument, 207, 211
pseudo-instruction, 11, 30, 207, 208, 210, 211, 213, 215
pseudo-mnemonic, 29, 30, 210, 211

register addressing, 17, 22, 23, 58–61, 203
register alias, 48, 202, 207, 210
register argument, 202, 204
register circularity, 11, 14, 15, 48, 59, 62, 64, 67, 70, 74, 78, 81, 83, 87, 90, 93, 96, 99, 102, 105, 108, 111, 133, 141, 145
register file, 11, 13–16, 18, 23, 38, 48, 58, 59
register file width, 14
register name, 13, 48, 202
register number, 13–15, 23
register state, 13

regularity, 11, 13, 22
relative program transfer, 17, 28, 152, 155, 158, 161, 164, 167, 170, 173, 176, 179, 182, 185, 188, 211
reset, 18, 19, 29, 35, 36, 43–45, 49, 53, 198, 199
reset state, 49
return from routine, 18, 27–29, 32, 46, 190, 192, 194, 196, 202, 211
return pointer, 16, 18, 28, 29, 34, 36, 43, 46, 48, 54–56, 151, 154, 155, 157, 158, 160, 161, 163, 164, 166, 167, 169, 170, 172, 173, 175, 176, 178, 179, 181, 182, 184, 185, 187, 188, 190, 192, 194, 196, 201
RISC, 11
routine, 18, 20, 27–29, 44, 46, 190, 192, 194, 196, 202, 211

scientific ISA, 11
second complement, 12
sign-extended immediate, 59, 89, 92, 95, 98
single pseudo-instruction, 207, 208, 210, 213
skipped instruction, 34, 35, 37, 38, 40
special register, 11, 16, 17, 19, 20, 33, 37, 38, 48, 49, 57, 60, 62, 64, 67, 71, 74, 78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 133, 135, 137, 139, 142, 145, 147, 149, 200, 202, 207, 210, 211
special register file, 13, 16, 48, 62, 65, 68, 71, 72, 74, 75, 78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 142, 143, 145, 146, 148, 150
special register file width, 48
stack pointer, 18, 207
store in memory, 11, 19, 22, 23, 58, 59, 61, 140, 204, 208
store-conditional, 19, 20, 23, 59
sync line, 20, 55
synchronization, 20, 23, 55
system instruction, 19, 34, 36, 37, 44, 60, 192, 194, 196, 200, 211
system mode, 19, 34, 36, 38, 43, 44, 48, 49, 53, 60, 194, 196, 200, 214, 215
system reset, 18, 49, 198, 199

temporary register, 207, 208, 211
transfer width, 22
two-address machine, 12

unconditional program transfer, 17, 27, 28, 32, 151, 202
undefined execution state, 35, 42
user mode, 19, 34, 37, 44, 48, 49, 53, 55, 60, 62, 192, 194, 196, 200, 211, 214, 215

virtual address, 17, 19, 50, 53
virtual address space, 16
virtual address width, 50

wait timer, 29, 198, 199
wait timer value, 29, 198, 199

zero-extended immediate, 59, 89, 92, 95, 98