Brownie STD 32

Reference Manual

---------- ***Ver. 1.1***

Change Log:

2008/07/22 v.1.1 : Released
2008/04/01 v.1.0 : Released

ASIP Solutions, Inc.
6F Sanko Osaka Hommachi Bldg 2-3-8 Hommachi, Chuo-ku, Osaka, 565-0871 Japan
support@asip-solutions.com

Release Notes

v.1.0:

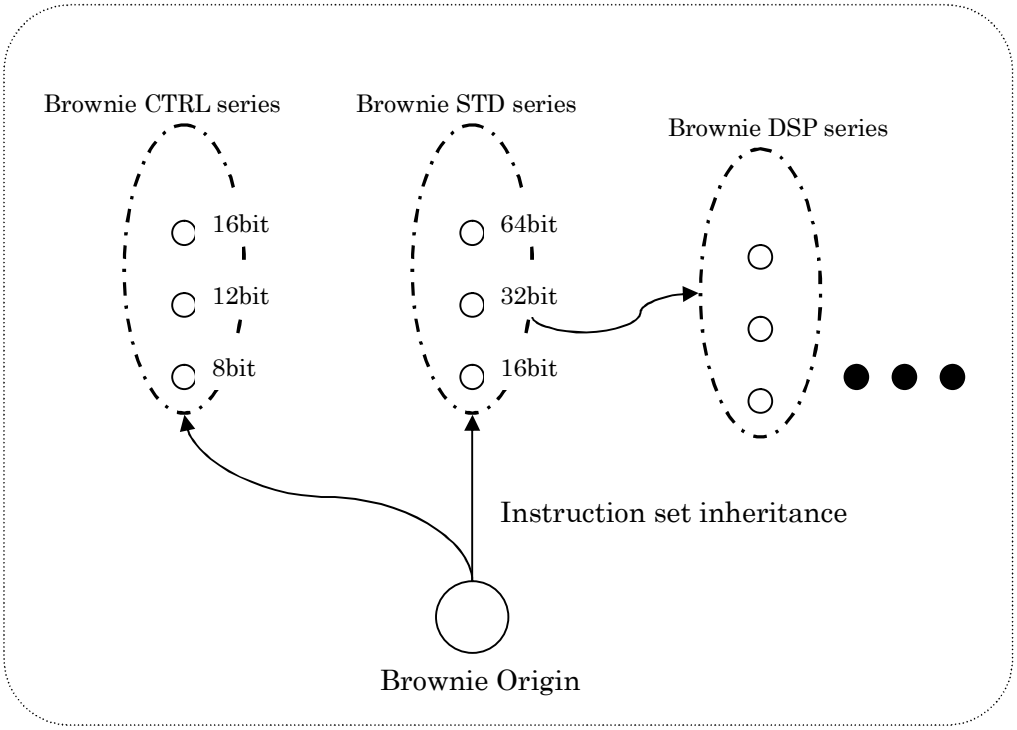
- Release

Introduction

An application specific instruction-set processor (ASIP) is a processor with an architecture specialized for various purposes. It is spotlighted as a processor for an embedded system enabling small area, low power consumption and high performance. Whereas ASIP enables efficient design of embedded system, design of ASIP itself can be time-consuming. A designer often cannot focus on design of specific instructions, spending most of design time for developing basic specifications of a processor or its implementation. It is desirable that a designer can focus on the design of specific instructions in designing ASIP and that, for this purpose, ASIP can be designed by adding specific instructions to a base processor. While ASIP development by extending a base processor is widely used as an efficient design method, several problems have been pointed out.

One problem is lack of options of base processors. The use of a base processor with a specification close r to the desired goal can help reduce a designer's burden, shorten the entire development period and improve performance of ASIP. However, conventional base processors are often distant from the target specification, forcing a designer to waste his energy. It is desirable that a designer can select a base processor from numerous candidate architectures designed for various purposes and needs in designing ASIP.

Another common problem in ASIP development by the extension of base processor is that a strong constraint by base process architecture hampers flexible design of specific instructions. Flexibility of design is essential in designing ASIP efficiently. ASIP Meister is well known as a tool to design ASIP flexibly and easily. It is a tool to generate HDL descriptions and assemblers of processors automatically from the specifications, and enables a designer to add resources, change architectures and extend instruction sets flexibly. Even in case of ASIP development using ASIP Meister, much labor is required for developing basic specifications of a processor etc. To develop ASIP flexibly in a short period, a base processor dedicated to ASIP Meister is necessary



Brownie family

Brownie is a base processor family designed on this requirement. The above figure shows the concept of Brownie family. Brownie family has instruction set series for various purposes, including Brownie CTRL series with an instruction set for control use, Brownie STD series with an instruction set for general use and Brownie DSP series with an instruction set for signal processing use. In each instruction set series, processors with different data bus bit width are available.A designer can choose Brownie best suited for base processor from these. The instruction set series of Brownie are mutually compatible. For example, the instruction set of Brownie DSP series maintains upper compatibility with that of Brownie STD series.Among features of Brownie is a strong affinity with ASIP Meister. A designer can design specific instructions by using ASIP Meister based on Brownie, and design complex specific instructions accompanied by changes in architectures and resources easily.Among features of Brownie is a strong affinity with ASIP Meister. A designer can design specific instructions by using ASIP Meister based on Brownie, and design complex specific instructions accompanied by changes in architectures and resources easily. Brownie has a necessary instruction set, allowing instruction set extension by an ASIP designer.

In addition to variation in architecture and flexibility and facility in customizing a base processor, another challenge in designing ASIP is the development of software development environment. Without software

development environment, such as a compiler, ASIP is hard to use in the real system. GNU software development environment (gcc, binutils, gdb, newlib) are included and usable in Brownie STD series upper. Brownie allows easier compiler extension through simple instruction sets, a simple architecture and instruction set compatibility within Brownie family. The simplification of instruction sets is important in improving performance of a processor and in developing a compiler. In designing ASIP based on Brownie, software development environment can be easily constructed by extending GNU environment provided with Brownie.

Thus, Brownie supports the demands as a base processor for ASIP and supports a designer powerfully with high development efficiency and design flexibility, used in combination with ASIP Meister.

Brownie STD, a universal instruction set series in Brownie family, has integer/logical operation, load/store and branch instructions and handles external/internal interrupts and general integer and peripheral operation. Brownie STD can be used as the core of the system.

1. Architecture

1.1. Architecture Overview

Brownie STD is a RISC-type pipeline processor architecture.Basic architecture parameters are shown in the following table.

Table. Architecture parameters

Basic architecture	RISC
Memory architecture	Harvard
Instruction length	32
Data length	32
Addressing	Byte address
Register Type	General-purpose register
The number of general-purpose registers	32
The number of pipeline stages	4
The number of delayed branch slot	0
Floating-point unit	N/A
Forwarding	full forwarding

The memory architecture of Brownie STD is Harvard. Both of the instruction length and the data length are 32 bit. Addressing can be performed in byte. Brownie has 32 integer general-purpose registers and a 4-stage pipeline structure, each stage of which is named IF, ID, EXE and WB. Full forwarding to the pipeline makes the operation results of all the instructions immediately usable. Brownie STD executes delayed load for load type instruction, and the number of delayed slot is one. Brownie STD does not have a delayed branch slot (does not execute delayed branch). Brownie STD does not have a floating-unit operation. A floating-point execution instruction can be added as a custom instruction.

1.2. Registers

Brownie STD has 32 integer registers, each of which is available for a programmer as GPR0 through GPR31. Some registers are reserved at hardware level. The values of registers reserved at hardware level can be automatically updated by the processor.

Zero Register:

GPR0 is reserved as a register to retain the value “0” and always outputs “0.” No value can be written in this register.

Status Register:

GPR1 is reserved as a register to reflect the status of the processor (Status Register). Status Register retains a flag, an interrupt mask and a processor mode. An operation execution accompanied by a flag change automatically updates a flag. An interrupt permission flag represents permission when it is "1," prohibition when it is "0." An execution mode flag represents user mode when it is "01," kernel mode when it is "00." Write into ALU flag is prohibited. Write into execution mode flag or interrupt permission flag is permitted only in kernel mode. Write is prohibited in modes other than kernel mode. Refer to Appendix B. Status Register for read/write of Status Register and the update timing.
The following table shows the meaning of each bit of Status Register.

Status Register [3:0]:	ALU flag (Carry, Zero, Sign, Overflow)
Status Register [8]:	External interrupt permission flag
Status Register [9]:	Internal interrupt permission flag
Status Register [15:14]:	Execution mode flag

Interrupt Return Register:

GPR2 retains the address of the instruction to which external or internal interrupts occur. The value of the register is automatically updated when interrupt occurs.

Link Register:

GPR3 retains the return address from JPL or JPRL instructions (See 2.2. Instructions). The address of the next instruction following JPL or JPRL instruction is stored. The execution of JPL and JPRL automatically updates the address.

GPR4, GPR5 and GPR6 are reserved as Return Register, Frame Pointer and Stack Pointer for the compiler. However, not being reserved at hardware level, they can be used by a programmer freely when the compiler is not in use. The following shows a list of reserved registers.

Table. Reserved registers

Reserved register	Use	Reservation
GPR0	Zero Register	hardware
GPR1	Status Register	hardware
GPR2	Interrupt Return Register	hardware
GPR3	Link Register	hardware
GPR4	FramePointer	compiler
GPR5	Stack Pointer	compiler
GPR6	Return Value for Integer or Float/Double	compiler
GPR7	Return Value for Double	compiler
GPR8-15	Register Passing	compiler
GPR16-31	Free	N/A

1.3. Memory Model

Addressing

Byte address is used for addressing of Brownie STD.

Alignment

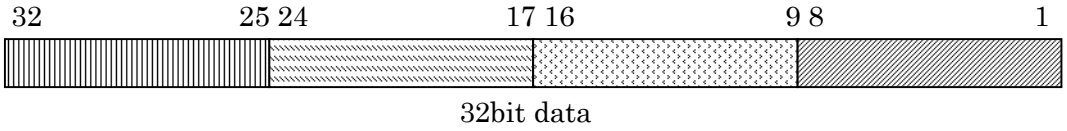
The alignment of Brownie STD has 1-byte (byte access), 2-byte (halfword access) and 4-byte boundaries (word access). Brownie STD calculates an effective address by rounding down accesses violating these alignments. For example, in word access, the alignment is

0x0000
0x0004
0x0008
0x000C
:

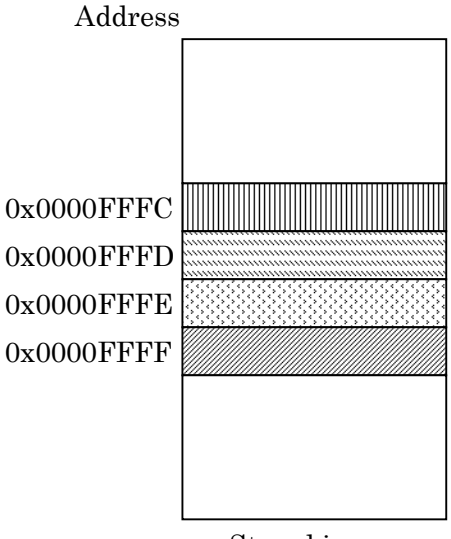
In this example, if the address 0x000A is specified, the last two digits are rounded down and the corrected address is used as an effective address.
Brownie STD does not except concerning alignment violation.

Endian

The byte order of Brownie STD is big endian.
In the figure below, 32-bit data are stored at the address 0x0000FFFC.



32bit data



Stored image

Memory Space

The following table shows the instruction memory space of Brownie STD.

The memory 0x00000000 through 0x0fffffff is used as kernel space; user space is 0x10000000 through 0xffffffff. An interrupt vector is located in 0x0ffe0000-0x0fff0000 in kernel space.

Table 1 Instruction Memory Space

Address	Space
0x00000000 - 0x0FFFFFFF	Kernel space
0x0FFE0000 - 0x0FFF0000	Interrupt vector space
0x10000000 - 0xFFFFFFFF	User space

1.4. Instruction Set

Instruction Type

Brownie STD has the following seven instruction types.

RR Type
RI Type
MA Type
BR Type
JP Type
JPR Type
SP Type
RT Type

RR (Register and Register) type instruction is a type of instruction to perform an operation using two source registers and write back the result to the registers. RI (Register and Immediate) type instruction is an instruction to perform an operation using a register and an immediate value and write back the result to the register. The immediate value is 16 bits. MA (Memory Access) type belongs to load/store type instruction. Register indirect addressing can be used for memory access. BR (Branch) type instruction branches (forks) by evaluating the value of the indexed register. JP (JumP) Type and JPR (JumP Register) Type indicate unconditional immediate value relative jump and register indirect absolute jump respectively. SP (Special) Type indicates instructions which do not index a source register and an output register (e.g. NOP). RT (Register Transfer) Type performs

an operation using one source register and write back the result to the register.

Addressing Mode

The instruction set of Brownie STD supports the following two addressing modes.

- Immediate value addressing
- Register indirect addressing

The offset can be indexed for register indirect addressing. The offset is expressed in byte address. An effective address is "the value of the register + offset." The offset is evaluated as signed value.

1.5. Interrupts

Brownie STD supports the following interrupts.

Priority	Type	Interrupt name
Highest	RESET	RESET
:	Internal	TRAP
Lowest	External	EXT

RESET interrupt is activated by an external reset signal. RESET interrupt is of highest priority and is not maskable. TRAP interrupt is of the second highest priority. TRAP interrupt is generated at software level. EXT interrupt is of the lowest priority. EXT interrupt is generated by external interrupt request. Brownie STD has an external interrupt request port and can index an interrupt factor from the interrupt controller outside the processor core. The interrupt controller is located in memory space and accessible from the data bus. Internal/external interrupts can be activated or invalidated by setting an appropriate flag in Status Register.

Brownie processes interrupts by the interrupt vectors. An interrupt vector is located in vector location space as below.

Table 2 Interrupt vector space

Address	Space
0x0FFE0000 - 0x0FFF000(	Interrupt vector location space
0x0FFE0000 - 0x0FFE03F(	Reset interrupt vector
0x0FFE0400 - 0x0FFE07F(	External interrupt vector
0x0FFE0800 - 0x0FFE0BF(	Internal interrupt vector

The space for TRAP exceptions is reserved as follows. Refer to 2.2.2. Instruction Descriptions for details on TRAP.

Address	Area
0x0FFE0800 - 0x0FFE0900	TRAP

1.6. Interface

The interfaces of Brownie STD are shown in the following table.

Port name	Direction	Bit width	Use
CLK	IN	1	Clock
RESET	IN	1	Reset
DMEM_ADDR_OUT	OUT	32	Data memory address bus
DMEM_ADDRERR_IN	IN	1	Data memory address error
DMEM_DATA_OUT	OUT	32	Data memory bus (in)
DMEM_DATA_IN	IN	32	Data memory bus (out)
DMEM_RW_OUT	OUT	1	Data memory R/W mode
DMEM_WMODE_OUT	OUT	2	Data memory access mode
DMEM_EMODE_OUT	OUT	1	Sign extension mode output
DMEM_CANCEL_OUT	OUT	1	Data memory cancellation
DMEM_REQ_OUT	OUT	1	Data memory Req
DMEM_ACK_IN	IN	1	Data memory Ack
IMEM_ADDR_OUT	OUT	32	Instruction memory address
IMEM_ADDRERR_IN	IN	1	Instruction memory address error
IMEM_DATA_IN	IN	32	Instruction memory bus
EXTINT_IN	IN	1	External interrupt request
EXTCATCH_OUT	OUT	1	Notification of the start of external interrupt processing

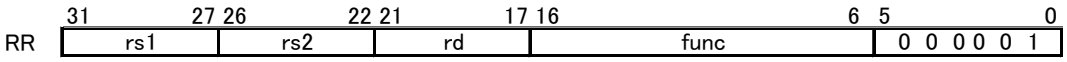
CLK and RESET refer to a clock and an external reset signal respectively. Brownie has respective memory interfaces for the instruction memory and the data memory. However, since Brownie STD assumes that the fetch is completed in one cycle, the instruction memory interface is without Req and Ack signals. An external interrupt request is input into EXTINT_IN. If the requested interrupt processing is started, Brownie STD notifies the start of the processing outward through EXTCATCH_OUT. Refer to 3. Interrupts and 4. Memory Access for details on interface protocol.

2. Instruction Set

2.1. Instruction Format

Below is a list of the formats of instruction types of Brownie STD and the numbers of instructions for the instruction types.

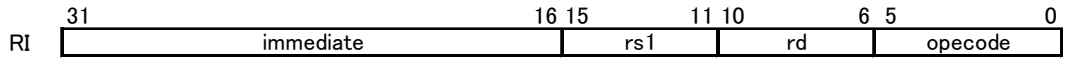
RR Type



rs1 and r2 indicate the indexes of source registers, while rd indicates the index of the register to store the operation results. func is a sub-opecode of RR Type. Lower 6 bits are an opecode, and fixed to “000001” in case of RR t ype.

Brownie STD has 15 instructions as RR type. 2033 extension instructions can be added to RR type.

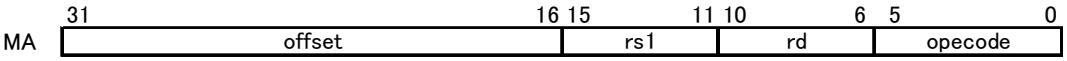
RI Type



rs1 indicates the index of a source register, while rd indicates the index of the register to store the operation results. Immediate is immediate value data. Lower 6 bits are an opecode, and “100000” through “111111” can be used in case of RI type.

Brownie STD has 9 instructions as RI type. The number of extension instructions that can be added to RI type is 32.

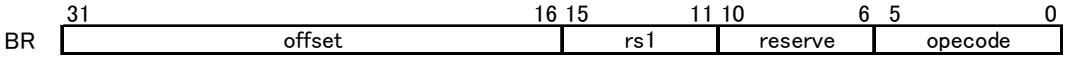
MA Type



rs1 and rd indicate the indexes of registers. What rs1 and rd indicate differs between load and store type in it. In load, rs1 and rd indicate the indexes of the source address and the store register respectively. In store, rs1 and rd indicate the register to store the destination address and the source register respectively. The offset is immediate data. Lower 6 bits are an opecode and “000010” through “000111” can be used in case of MA type. The addressing mode is register indirect access.

The number of instructions that can be added to MA type is shared with BR, JP and JPR. “010000” through “011111” in the opecode field is available for the number, which are 16 in total.

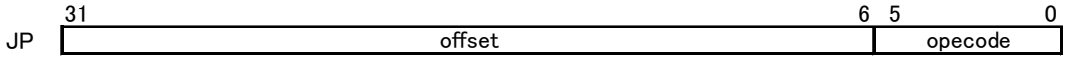
BR Type



rs1 indicates the index of the source register. The offset is immediate value data. Lower 6 bits are an opecode, and “001001” through “001010” can be used in case of BR type. The addressing mode is immediate value access.

The number of instructions that can be added to BR type is shared with MA, JP and JPR. “010000” through “011111” in the opecode field is available for the number, which are 16 in total.

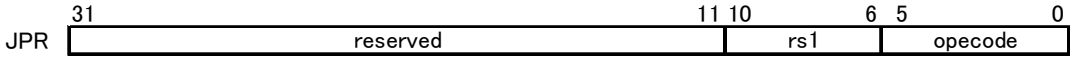
JP Type



The offset is immediate value data. Lower 6 bits are an opecode, and “001011” through “001101” can be used in case of JP type. The addressing mode is immediate value access.

The number of instructions that can be added to JP t ype is shared with MA, BR and JPR. ” 010000” through ” 011111” in the opecode field is available for the number, which is 16 in total.

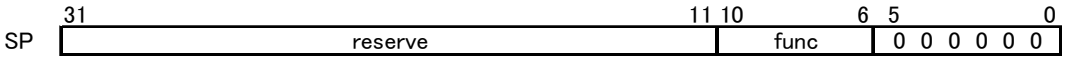
JPR Type



rs1 is the index of the register to store the address. Lower 6 bits are an opecode, and “001110” through “001111” can be used in case of JPR type. The addressing mode is register indirect access.

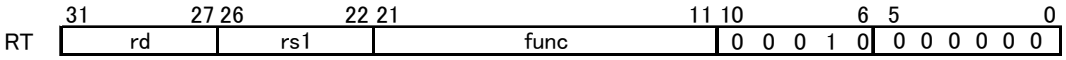
The number of instructions that can be added to JPR type is shared with MA, BR and JP. ” 010000” through ” 011111” in the opecode field is available for the number, which is 16 in total

SP Type



Lower 7 bits are an opecode, and fixed to “000000” in case of SP Type.

RT Type



Lower 11 bits are an opecode, and fixed to “00010000000” in case of RT type.

2.2. Instructions

2.2.1. Behavior Notation

The following table defines how to describe instruction behaviors of Brownie STD. Comparison operators marked with (unsigned) indicate unsigned comparison.

Table 3 Behavior notations and meanings

Behavior notations	Meanings
A + B	Arithmetic addition
A - B	Arithmetic subtraction
A * B	Arithmetic product
A / B	Arithmetic division
A % B	Modular multiplication
A & B	Bitwise AND
A | B	Bitwise OR
A ^ B	Bitwise XOR
~ A	Bitwise NOT
A >> B	Logical right shift of A by the value of B
A << B	Logical left shift of A by the value of B
A >>> B	Arithmetic right shift of A by the value of B
A <<< B	Arithmetic left shift of A by the value of B

zero(A, B)	Zero extension of data A to B-th bit
sign(A, B)	Sign extension of data A to B-th bit
conj(A, .., Z)	Concatenation from A to Z (A is higher level)
min(A, B)	Minimum value of {A, B}
max(A, B)	Maximum value of {A, B}

A == B	Truth value indicating whether A and B are equivalent
A != B	Truth value indicating whether A and B are not equivalent
A < B	Truth value indicating whether A is smaller than B
A > B	Truth value indicating whether A is larger than B
A >= B	Truth value indicating whether A is larger than or equal to B
A <= B	Truth value indicating whether A is smaller than or equal to B
A && B	Logical conjunction
A || B	Logical sum
! A	Logical negation
C ? A : B	A if C is true, B if false

A = B	Data transfer from right hand B to left hand A
if(exp)	Execute a sentence if an expression exp is true

PC	Value of program counter
LinkRegister	Value of link register (GPR3)
StatusRegister	Value of Status Register (GPR1)
GPR(n)	Value of universal register index of which is n
Data(addr)	Value of data memory stored at the address addr (in terms of by
A[B]	Value of B-th bit of data A
A[B..C]	Cutout of B-th bit through C-th bit

CURADDR	Address of instruction memory in which the instruction is locat
TRPBASE	Base address of TRAP instruction

rs1, rs2, rd	Source register number
immediate	Immediate value
offset	Offset value

2.2.2. Instruction Descriptions

The instructions of Brownie STD are shown. Detailed formats of the instructions are as follows.

Instruction Type: The type of instruction
Instruction Format: Assembly format of instruction
Flag Change: Changes in flag (C, Z, S, V), (-, -, -, V) etc.
Exception: The stage number in case the instruction causes an exception
Behavior:
Behavioral description by behavior notations

Description:
Overview of instructions

Examples in Flag Change field represent

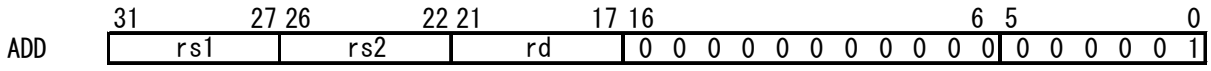
C: Carry
Z: Zero
S: Sign
V: Overflow
—: The previous values are retained
x: Contingent

Flag Change is reflected in Status Register after the second instruction. Check Flag Change after the second instruction, if you wish to. Refer to Appendix B.3. Update Timing of Flag Register for the timing of Flag Change.

The carry flag (C) indicates the following in case of addition and subtraction.

- addition
  - C = 1: Occurrence of carry
  - C = 0: No occurrence of carry
- subtraction
  - C = 1: No occurrence of borrow
  - C = 0: Occurrence of borrow

ADD — *Addition*



Instruction Type: *RR Type*

Instruction Format: ADD *rd*, *rs, rs2*

Flag Change: (C, Z, S, V)

Exception: No

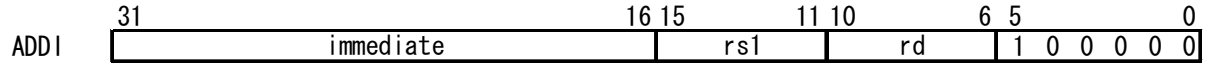
Behavior:

```
GPR(rd) = GPR(rs1) + GPR(rs2);
```

Description:

To perform the signed arithmetic addition of GPR (rs1) and GPR (rs2) and write the result to GPR (rd). The flag change accompanying this operation (Carry, Zero, Sign, Overflow) is reflected in Status Register.

ADDI — *Addition with Immediate*



Instruction Type: *RI Type*

Instruction Format: ADDI *rd, rs1, immediate*

Flag Change: (C, Z, S, V)

Exception: No

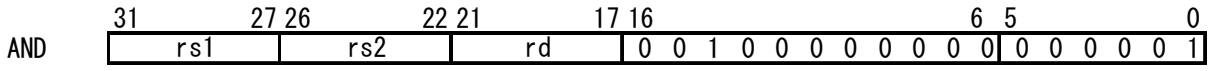
Behavior:

```
GPR(rd) = GPR(rs1) + sign(immediate, 32);
```

Description:

To perform the signed arithmetic addition of the values of GPR (rs1) and immediate and write the result to GPR (rd). The flag change accompanying this operation (Carry, Zero, Sign, Overflow) is reflected in Status Register. The value of immediate is handled as a signed value.

AND — *Logical AND*



Instruction Type: *RR Type*

Instruction Format: AND *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

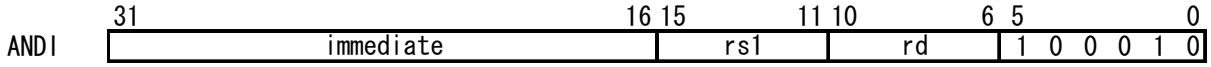
Behavior:

```
GPR(rd) = GPR(rs1) & GPR(rs2);
```

Description:

To calculate the logical AND of the values of GPR (rs1) and GPR (rs2) and write the result to GPR (rd).

ANDI — *Logical AND with Immediate*



Instruction Type: *RI Type*

Instruction Format: ANDI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

Behavior:

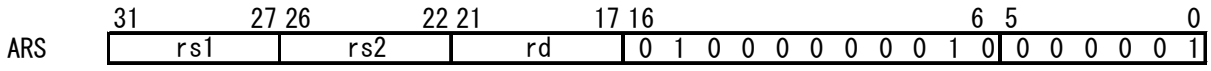
```
GPR(rd) = GPR(rs1) & zero(immediate, 32);
```

Description:

To calculate the logical AND of the values of GPR (rs1) and immediate and write the result to GPR (rd).

The value of immediate is handled as an unsigned value.

ARS — *Arithmetic Right Shift*



Instruction Type: *RR Type*

Instruction Format: ARS *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

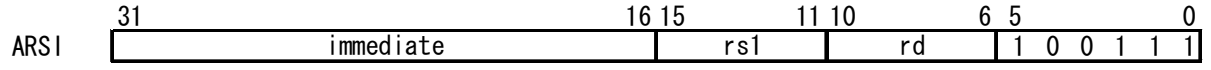
Behavior:

```
GPR(rd) = GPR(rs1) >>> GPR(rs2)[4..0];
```

Description:

To perform the arithmetic right shift of GPR (rs1) by the value of GPR (rs2) and write the result to GPR (rd). Only low bits of the GPR (rs2) value are evaluated, while the rest are ignored.

ARSI — *Arithmetic Right Shift with Immediate*



Instruction Type: *RI Type*

Instruction Format: ARSI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

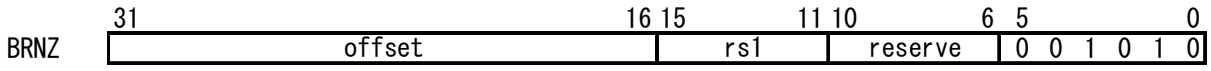
Behavior:

```
GPR(rd) = GPR(rs1) >>> immediate[4..0];
```

Description:

To perform the arithmetic right shift of GPR (rs1) by the value of immediate and write the result to GPR (rd). Only valid low bits of the immediate value are evaluated, while the rest are ignored.

BRNZ — Branch Not Zero



Instruction Type: *BR Type*

Instruction Format: BRNZ *rs1, offset*

Flag Change: (-, -, -, -)

Exception: No

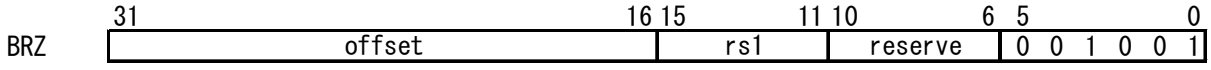
Behavior:

```
if(GPR(rs1) != 0)
  PC = CURADDR + 4 + sign (offset, 32);
```

Description:

Add the value of the offset converted by sign extension to that of PC, if the value of GPR (rs1) is not 0 (relative address jump). The reference point of this jump is the address of the next instruction.

BRZ — Branch Zero



Instruction Type: *BR Type*

Instruction Format: BRZ *rs1, offset*

Flag Change: (-, -, -, -)

Exception: No

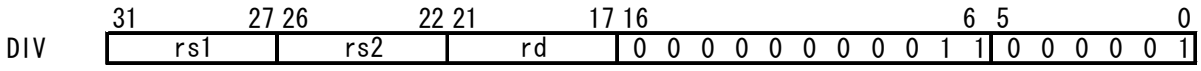
Behavior:

```
if(GPR(rs1) == 0)
  PC = CURADDR + 4 + sign (offset, 32);
```

Description:

If the value of GPR (rs1) is 0, the result of signed extension of the offset value is added to the value of PC (relative address jump). The reference point of this jump is the address of the next instruction.

DIV — *Division*



Instruction Type: *RR Type*

Instruction Format: DIV *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

Behavior:

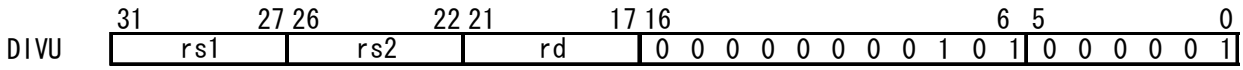
```
GPR(rd) = GPR(rs1) / GPR(rs2);
```

Description:

The result of the signed division of the values of GPR (rs1) and GPR (rs2) is written to GPR (rd).

If division by zero or overflow-causing operation (0x80000000 /0xFFFFFFFF) is performed, the result is contingent.

DIVU — *Unsigned Division*



Instruction Type: *RR Type*

Instruction Format: DIVU *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

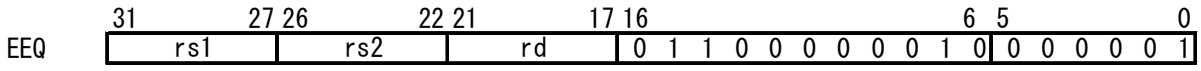
Behavior:

```
GPR(rd) = GPR(rs1) /(unsigned) GPR(rs2);
```

Description:

The result of the unsigned division of the values of GPR (rs1) and GPR (rs2) is written to GPR (rd).

EEQ — *Evaluate Equal to*



Instruction Type: *RR Type*

Instruction Format: EEQ *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

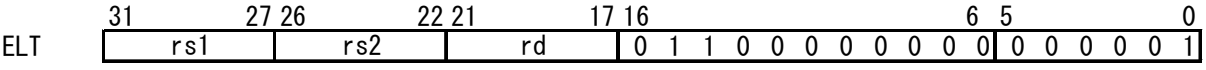
Behavior:

```
GPR(rd) = (GPR(rs1) == GPR(rs2)) ? 1:0;
```

Description:

If the value of GPR (rs) is equal to that of GPR (rs2), 1 is written to GPR (rd). Otherwise, 0 is written. The values of GPR (rs1) and GPR (rs2) are evaluated as signed values.

ELT — *Evaluate Less Than*



Instruction Type: *RR Type*

Instruction Format: ELT *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

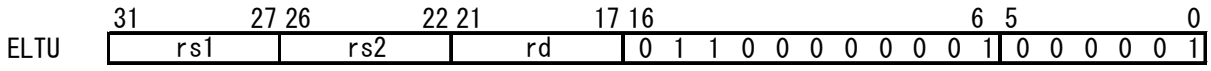
Behavior:

```
GPR(rd) = (GPR(rs1) < GPR(rs2)) ? 1:0;
```

Description:

If the value of GPR (rs1) is smaller than that of GPR (rs2), 1 is written to GPR (rd). Otherwise, 0 is written. The values of GPR (rs1) and GPR (rs2) are evaluated as signed values.

ELTU *Evaluate Less Than Unsigned*



Instruction Type: *RR Type*

Instruction Format: ELTU *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

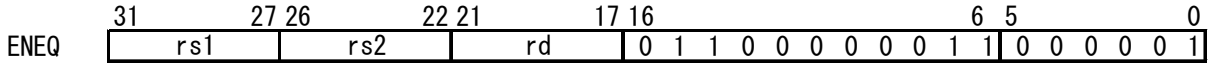
Behavior:

```
GPR(rd) = (GPR(rs1) <(unsigned) GPR(rs2)) ? 1:0;
```

Description:

If the value of GPR (rs1) is smaller than that of GPR (rs2), 1 is written to GPR(rd). Otherwise, 0 is written. The values of GPR (rs1) and GPR (rs2) are evaluated as unsigned values.

ENEQ *Evaluate Not Equal to*



Instruction Type: *RR Type*

Instruction Format: ENEQ *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

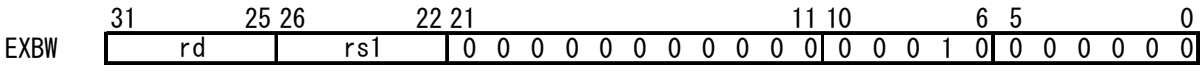
Behavior:

```
GPR(rd) = (GPR(rs1) != GPR(rs2)) ? 1:0;
```

Description:

If the value of GPR (rs1) is not equal to that of GPR(rs2), 1 is written to GPR (rd). Write "0" otherwise. The values of GPR (rs1) and GPR (rs2) are evaluated as signed values.

EXBW — *Extend Byte to Word*



Instruction Type: *RT Type*

Instruction Format: EXBW *rd, rs1*

Flag Change: (-, -, -, -)

Exception: No

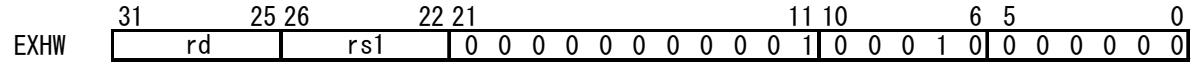
Behavior:

```
GPR(rd) = sign(GPR(rs1)[7:0], 32);
```

Description:

Lower 8 bits of GPR (rs1) are converted to 32-bit data by sign extension.

EXHW — *Extend Half word to Word*



Instruction Type: *RT Type*

Instruction Format: EXHW *rd, rs1*

Flag Change: (-, -, -, -)

Exception: No

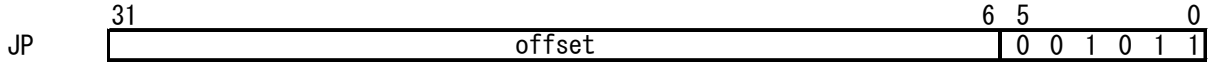
Behavior:

```
GPR(rd) = sign(GPR[15:0], 32);
```

Description:

Lower 16 bits of GPR (rs1) are converted to 32-bit data by sign extension.

JP — *Jump*



Instruction Type: *JP Type*

Instruction Format: JP *offset*

Flag Change: (-, -, -, -)

Exception: No

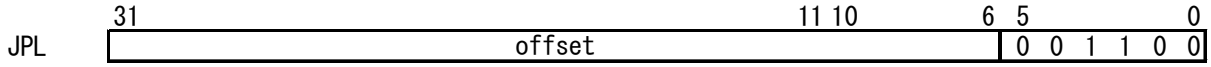
Behavior:

```
PC = CURADDR + 4 + sign(offset, 32);
```

Description:

To perform the sign extension of the offset value unconditionally and add the result to the value of PC (relative address jump). The reference point of this jump is the address of the next instruction. The offset value is handled as a signed value.

JPL — *Jump and Link*



Instruction Type: *JP Type*

Instruction Format: JPL *offset*

Flag Change: (-, -, -, -)

Exception: No

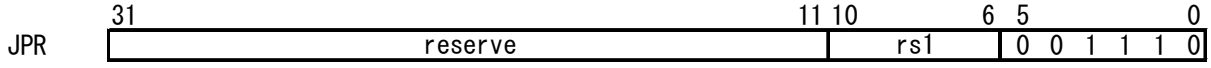
Behavior:

```
LinkRegister = CURADDR + 4;
PC           = CURADDR + 4 + sign(offset, 32);
```

Description:

To write the address of the following instruction (PC + 4) and add the value of the offset unconditionally converted by sign extension to the value of PC (relative address jump). The reference point of this jump is the address of the next instruction. The offset value is evaluated as a signed value.

JPR — *Jump with Register*



Instruction Type: *JPR Type*

Instruction Format: JPR *rs1*

Flag Change: (-, -, -, -)

Exception: No

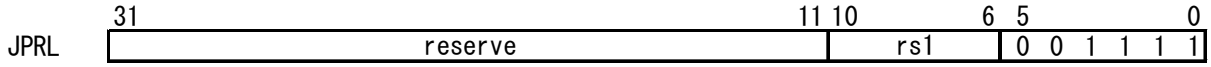
Behavior:

```
PC = GPR(rs1);
```

Description:

To write the value of GPR (rs1) to PC unconditionally (absolute address jump).

JPRL — *Jump with Register and Link*



Instruction Type: *JPR Type*

Instruction Format: JPRL *rs1*

Flag Change: (-, -, -, -)

Exception: No

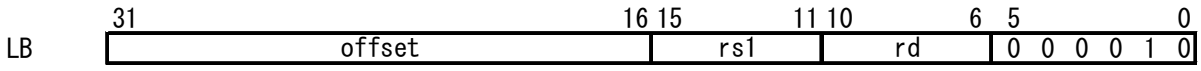
Behavior:

```
LinkRegister = CURADDR + 4;
PC           = GPR(rs1);
```

Description:

To write the address of the following instruction (PC + 4) to Linkregister and the value of GPR (rs1) unconditionally to PC (absolute address jump).

LB — *Load Byte*



Instruction Type: *MA Type*

Instruction Format: LB *rd, offset(rs1)*

Flag Change: (-, -, -, -)

Exception: No

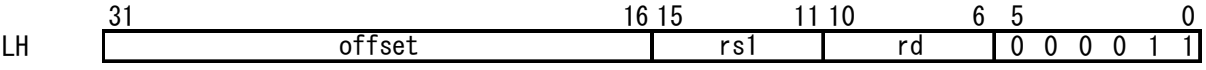
Behavior:

```
GPR(rd) = sign(Data(GPR(rs1) + offset), 32);
```

Description:

To access the data memory using the addition of the values of GPR (rs1) and the offset as an effective address. The offset value is evaluated as a signed value.

LH — *Load Half word*



Instruction Type: *MA Type*

Instruction Format: LH *rd, offset(rs1)*

Flag Change: (-, -, -, -)

Exception: No

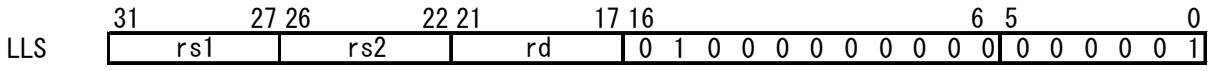
Behavior:

```
GPR(rd) = sign( conj(Data(GPR(rs1) + offset),
                     Data(GPR(rs1) + offset + 1), 32);
```

Description:

To access the data memory using the addition of the values of GPR (rs1) and the offset as an effective address. The offset value is evaluated as a signed value.

LLS — *Logical Left Shift*



Instruction Type: *RR Type*

Instruction Format: LLS *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

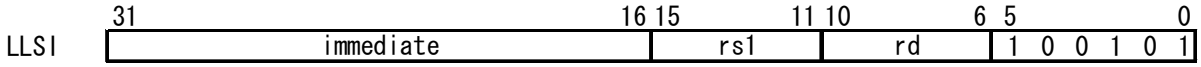
Behavior:

```
GPR(rd) = GPR(rs1) << GPR(rs2)[4..0];
```

Description:

To perform the logical left shift of GPR (rs1) by the value of GPR (rs2) and write the result to GPR (rd). Only valid low bits of the GPR (rs2) value are evaluated, while the rest are ignored.

LLSI — *Logical Left Shift with Immediate*



Instruction Type: *RI Type*

Instruction Format: LLSI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

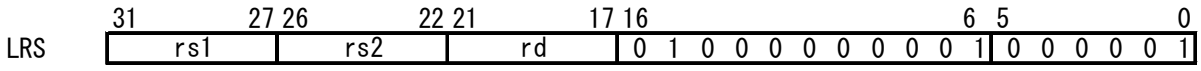
Behavior:

```
GPR(rd) = GPR(rs1) << immediate[4..0];
```

Description:

To perform the logical left shift of GPR (rs1) by the value of immediate and write the result to GPR (rd). Only valid low bits of the immediate value are evaluated, while the rest are ignored.

LRS — *Logical Right Shift*



Instruction Type: *RR Type*

Instruction Format: LRS *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

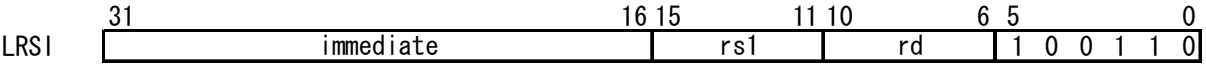
Behavior:

```
GPR(rd) = GPR(rs1) >> GPR(rs2)[4..0];
```

Description:

To perform the logical right shift of GPR (rs1) by the value of GPR (rs2) and write the result to GPR (rd). Only valid low bits of the GPR (rs2) value are evaluated, while the rest are ignored.

LRSI — *Logical Right Shift with Immediate*



Instruction Type: *RI Type*

Instruction Format: LRSI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

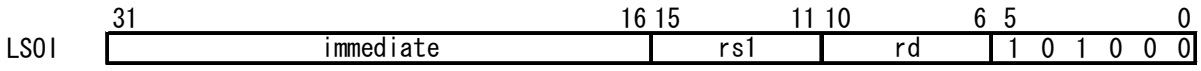
Behavior:

```
GPR(rd) = GPR(rs1) >> immediate[4..0];
```

Description:

To perform the logical right shift of GPR (rs1) by the value of immediate and write the result to GPR (rd). Only valid low bits of the immediate value are evaluated, while the rest are ignored.

LSOI *Left Shift and OR with Immediate*



Instruction Type: *RI Type*

Instruction Format: LSOI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

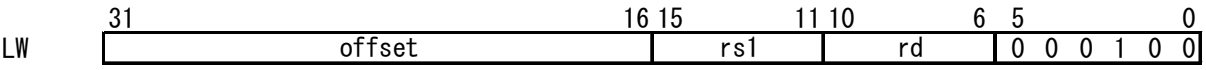
Behavior:

```
GPR(rd) = (GPR(rs1) << 16) | zero(immediate, 32);
```

Description:

To perform the 16-bit left shift of GPR (rs1), calculate the logical OR of the value and the immediate value and write the result to GPR (rd).

LW *Load Word*



Instruction Type: *MA Type*

Instruction Format: LW *rd, offset(rs1)*

Flag Change: (-, -, -, -)

Exception: No

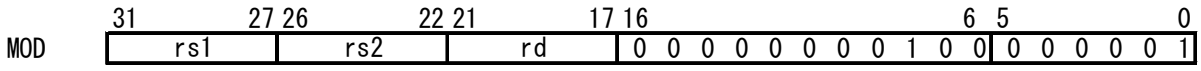
Behavior:

```
GPR(rd) = conj(Data(GPR(rs1) + offset),
               Data(GPR(rs1) + offset + 1),
               Data(GPR(rs1) + offset + 2),
               Data(GPR(rs1) + offset + 3));
```

Description:

To access the data memory using the addition of the values of GPR (rs1) and the offset as an effective address. The offset value is evaluated as a signed value.

MOD *Modulo arithmetic*



Instruction Type: *RR Type*

Instruction Format: MOD *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

Behavior:

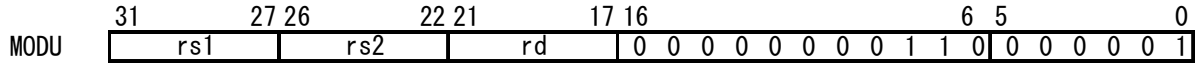
```
GPR(rd) = GPR(rs1) % GPR(rs2);
```

Description:

To calculate the signed subtraction of the values of GPR (rs1) and GPR (rs2) and write the result to GPR (rd).

The result of this operation is always positive. If division by zero or overflow-causing operation (0x80000000 % 0xFFFFFFFF) is performed, the result is contingent.

MODU *Unsigned Modulo arithmetic*



Instruction Type: *RR Type*

Instruction Format: MODU *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

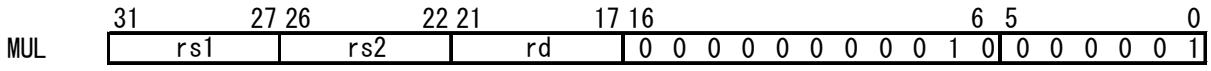
Behavior:

```
GPR(rd) = GPR(rs1) %(unsinged) GPR(rs2);
```

Description:

To calculate the unsigned subtraction of the values of GPR (rs1) and GPR (rs2) and write the result to GPR (rd). The result of this operation is always positive.

MUL — *Multiplication*



Instruction Type: *RR Type*

Instruction Format: MUL *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

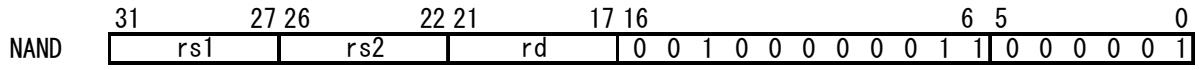
Behavior:

```
GPR(rd) = GPR(rs1) * GPR(rs2);
```

Description:

To calculate the singed multiplication of values of GPR (rs1) and GPR (rs2) and write the result to GPR (rd).

NAND — *Logical NAND*



Instruction Type: *RR Type*

Instruction Format: NAND *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

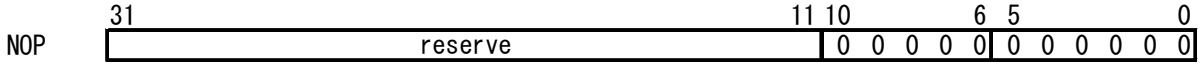
Behavior:

```
GPR(rd) = ~(GPR(rs1) & GPR(rs2));
```

Description:

To calculate the logical NAND of the values of GPR (rs1) and GPR (rs2) and write the result to GPR (rd).

NOP — *No Operation*



Instruction Type: *SP Type*

Instruction Format: NOP

Flag Change: (-, -, -, -)

Exception: No

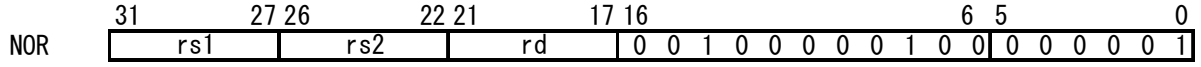
Behavior:

```
(Nothing)
```

Description:

NOP instruction has no effective behavior.

NOR — *Logical NOR*



Instruction Type: *RR Type*

Instruction Format: NOR *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

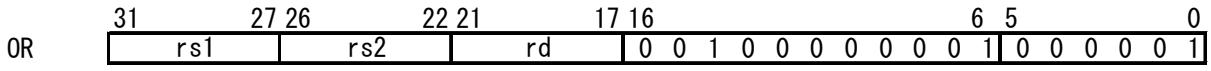
Behavior:

```
GPR(rd) = ~(GPR(rs1) | GPR(rs2));
```

Description:

To calculate the logical NOR of the values of GPR (rs1) and GPR (rs2) and write the result to GPR (rd).

OR — *Logical OR*



Instruction Type: *RR Type*

Instruction Format: OR *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

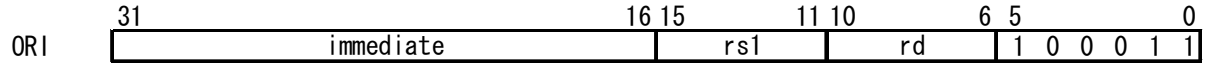
Behavior:

```
GPR(rd) = GPR(rs1) | GPR(rs2);
```

Description:

To calculate the logical OR of the values of GPR (rs1) and GPR (rs2) and write the result to GPR (rd).

ORI — *Logical OR with Immediate*



Instruction Type: *RI Type*

Instruction Format: ORI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

Behavior:

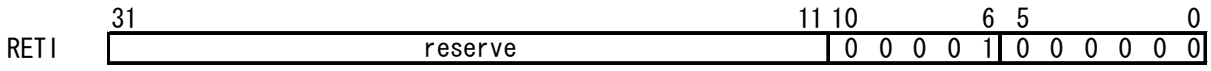
```
GPR(rd) = GPR(rs1) | zero(immediate, 32);
```

Description:

To calculate the logical OR of the values of GPR (rs1) and immediate and write the result to GPR (rd).

The immediate value is handled as an unsigned value.

RETI — *Return from Interrupt*



Instruction Type: *SP Type*

Instruction Format: RETI

Flag Change: (-, -, -, -)

Exception: No

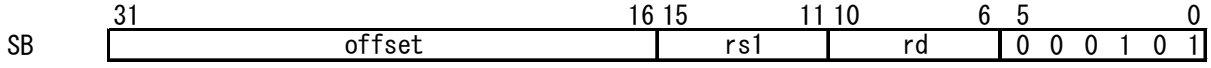
Behavior:

```
Status Register[8]          = 1;
Status Register[9]          = 1;
Status Register[15:14]      = "01";
PC                          = Interrupt Return Register;
```

Description:

RETI instruction activates all the interrupts and writes the value of Interrupt Return Register to PC.

SB — *Store Byte*



Instruction Type: *MA Type*

Instruction Format: SB *offset (rd), rs1*

Flag Change: (-, -, -, -)

Exception: No

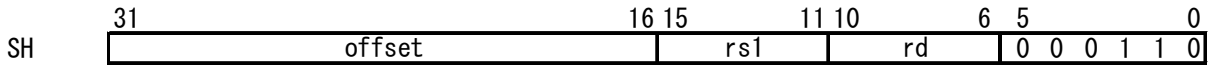
Behavior:

```
Data(GPR(rd) + offset) = GPR(rs1)[7..0];
```

Description:

To calculate the addition of the values of GPR (rd) and the offset and write the value of GPR (rs1) as an effective address. The offset value is evaluated as a signed value.

SH — *Store Half word*



Instruction Type: *MA Type*

Instruction Format: SH *offset(rd), rs1*

Flag Change: (-, -, -, -)

Exception: No

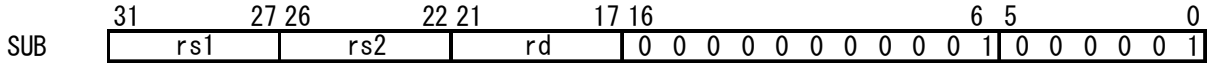
Behavior:

```
Data(GPR(rd) + offset)     = GPR(rs1)[15..8];
Data(GPR(rd) + offset + 1) = GPR(rs1)[7..0];
```

Description:

To calculate the addition of the values of GPR (rd) and the offset and write the value of GPR (rs1) to the data memory as an effective address.

SUB — *Subtraction*



Instruction Type: *RR Type*

Instruction Format: SUB *rd, rs1, rs2*

Flag Change: (C, Z, S, V)

Exception: No

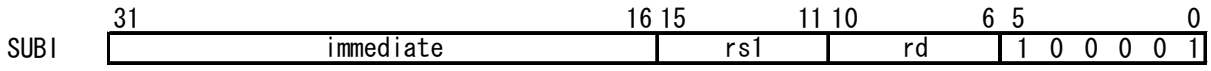
Behavior:

```
GPR(rd) = GPR(rs1) - GPR(rs2);
```

Description:

To calculate the signed subtraction of the values of GPR (rs1) and GPR (rs2) and write the result to GPR (rd). A flag change (Carry, Zero, Sign and Overflow) accompanying this operation is reflected in Status Register.

SUBI — *Subtraction with Immediate*



Instruction Type: *RI Type*

Instruction Format: SUBI *rd, rs1, immediate*

Flag Change: (C, Z, S, V)

Exception: No

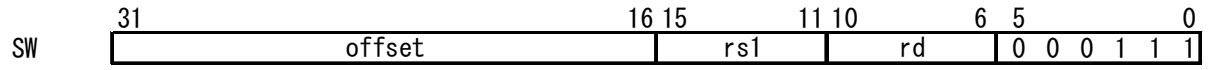
Behavior:

```
GPR(rd) = GPR(rs1) - sign(immediate, 32);
```

Description:

To calculate the singed subtraction of the values of GPR (rs1) and immediate and write the result to GPR (rd). A flag change (Carry, Zero, MSB and Overflow) accompanying this operation is reflected in Status Register. The immediate value is handled as a signed value.

SW — *Store Word*



Instruction Type: *MA Type*

Instruction Format: SW *offset(rd), rs1*

Flag Change: (-, -, -, -)

Exception: No

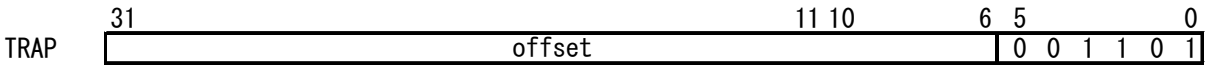
Behavior:

```
Data(GPR(rd) + offset)     = GPR(rs1)[31..24];
Data(GPR(rd) + offset + 1) = GPR(rs1)[23..16];
Data(GPR(rd) + offset + 2) = GPR(rs1)[15..8];
Data(GPR(rd) + offset + 3) = GPR(rs1)[7..0];
```

Description:

To calculate the addition of the values of GPR (rd) and the offset and write the value of GPR (rs1) to the data memory as an effective address. The offset is evaluated as a signed value.

TRAP *Trap*



Instruction Type: *JP Type*

Instruction Format: TRAP *offset*

Flag Change: (-, -, -, -)

Exception: Stage 3

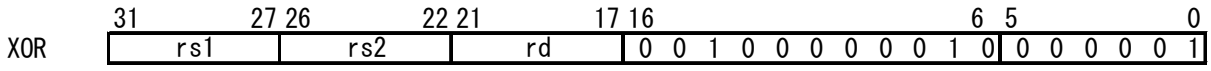
Behavior:

```
if(Status Register[9] == 1){
  Status Register[8]              = 0;
  Status Register[9]              = 0;
  Status Register[15:14]          = "00";
  Interrupt Return Register       = CURADDR;
  PC                              = TRPBASE + offset;
}
```

Description:

To generate software interrupt. The offset of the operand indicates the type of software interrupt (handler address).

XOR — *Logical Exclusive OR*



Instruction Type: *RR Type*

Instruction Format: XOR *rd, rs1, rs2*

Flag Change: (-, -, -, -)

Exception: No

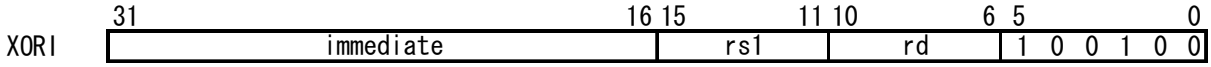
Behavior:

```
GPR(rd) = GPR(rs1) ^ GPR(rs2);
```

Description:

To calculate the exclusive OR of GPR (rs1) and GPR (rs2) and write the result to GPR (rd).

XORI — *Logical Exclusive OR with Immediate*



Instruction Type: *RI Type*

Instruction Format: XORI *rd, rs1, immediate*

Flag Change: (-, -, -, -)

Exception: No

Behavior:

```
GPR(rd) = GPR(rs1) ^ zero (immediate, 32);
```

Description:

To calculate the exclusive OR of GPR (rs1) and immediate and write the result to GPR (rd).

The immediate value is treated as an unsigned value.

2.2.3. Instruction Set Quick Reference

Below is a list of the instructions, the types, the formats, the code allocations and the mnemonic formats of Brownie STD.

Type	Insn	31–27	26–22	21–17	16–6	5–0	Format	Meanings
RR	ADD	rs1	rs2	rd	0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 1	OP *rd, rs1, rs2*	Addition
	SUB				0 0 0 0 0 0 0 0 0 0 1			Subtraction
	MUL				0 0 0 0 0 0 0 0 0 1 0			Multiplication
	DIV				0 0 0 0 0 0 0 0 0 1 1			Division
	DIVU				0 0 0 0 0 0 0 0 1 0 1			Unsigned division
	MOD				0 0 0 0 0 0 0 0 1 0 0			Modular multiplication
	MODU				0 0 0 0 0 0 0 0 1 1 0			Unsigned modular multiplication
	AND				0 0 1 0 0 0 0 0 0 0 0			Logical AND
	NAND				0 0 1 0 0 0 0 0 0 1 1			Logical NAND
	OR				0 0 1 0 0 0 0 0 0 0 1			Logical OR
	NOR				0 0 1 0 0 0 0 0 1 0 0			Logical NOR
	XOR				0 0 1 0 0 0 0 0 0 1 0			Logical XOR
	LLS				0 1 0 0 0 0 0 0 0 0 0			Logical left shift
	LRS				0 1 0 0 0 0 0 0 0 0 1			Logical left shift
	ARS				0 1 0 0 0 0 0 0 0 1 0			Arithmetic right shift
	ELT				0 1 1 0 0 0 0 0 0 0 0			Comparison (<)
	ELTU				0 1 1 0 0 0 0 0 0 0 1			Unsigned comparison (<)
	EEQ				0 1 1 0 0 0 0 0 0 1 0			Comparison operation (=)
	ENEQ				0 1 1 0 0 0 0 0 0 1 1			Comparison operation (!=)

Type	Insn	31–16	15–11	10–6	5–0	Format	Meanings
RI	ADDI	immediate	rs1	rd	1 0 0 0 0 0	OP *rd, rs1, immediate*	Immediate value operation
	SUBI				1 0 0 0 0 1		
	ANDI				1 0 0 0 1 0		
	ORI				1 0 0 0 1 1		
	XORI				1 0 0 1 0 0		
	LLSI				1 0 0 1 0 1		
	LRSI				1 0 0 1 1 0		
	ARSI				1 0 0 1 1 1		
	LSOI				1 0 1 0 0 0		left shift & immediate value OR

Type	Insn	31–16	15–11	10–6	5–0	Format	Meanings
MA	LB	offset	rs1	rd	0 0 0 0 1 0	OP *rd, offset(rs1)*	Byte load
	LH				0 0 0 0 1 1		Halfword load
	LW				0 0 0 1 0 0		Word load
	SB				0 0 0 1 0 1	OP *offset(rd), rs1*	Store byte
	SH				0 0 0 1 1 0		Store halfword
	SW				0 0 0 1 1 1		Store word

Type	Insn	31–16	15–11	10–6	5–0	Format	Meanings
BR	BRZ	offset	rs1	reserved	0 0 1 0 0 1	OP *rs1, offset*	Branch (rs1 = 0)
	BRNZ				0 0 1 0 1 0		Branch (rs1 != 0)

Type	Insn	31–6	5–0	Format	Meanings
JP	JP	offset	0 0 1 0 1 1	OP *offset*	Immediate value jump
	JPL		0 0 1 1 0 0		Immediate value JAL
	TRAP		0 0 1 1 0 1		TRAP

Type	Insn	31–11	10–6	5–0	Format	Meanings
JPR	JPR	reserved	rs1	0 0 1 1 1 0	OP *rs1*	Register indirect jump
	JPRL			0 0 1 1 1 1		Register indirect JAL

Type	Insn	31–11	10–6	5–0	Format	Meanings
SP	NOP	reserved	0 0 0 0 0	0 0 0 0 0 0	OP	NOP
	RETI		0 0 0 0 1			Interrupt return

Type	Insn	31–27	26–22	21–11	10–6	5–0	Format	Meanings
RT	EXBW	rd	rs1	0 0 0 0 0 0 0 0 0 0 0	0 0 0 1 0	0 0 0 0 0 0	OP *rd, rs1*	8bit -> 32bit Sign extension
	EXHW			0 0 0 0 0 0 0 0 0 0 1				16bit -> 32bit Sign extension

3. Interrupts

This chapter explains interrupt processing of Brownie STD.

3.1. Reset Interrupt

A reset interrupt is launched by inputting "1" into the reset input port.
Once a reset interrupt is launched, all the registers within the processor are cleared and the program counter is set to the reset vector address.

3.2. External Interrupt

An external interrupt can be invalidated by setting the external interrupt permission flag bit to "0." After the invalidation, an external interrupt asserted in EXTINT_IN is ignored. If an external interrupt is detected and the handler is launched, Brownie STD interrupts the current processing, stores the address of the instruction to return in Interrupt Return Register, branches into external interrupt vector address 0x0FFE0400 and invalidates an internal/external interrupt. Simultaneously, the execution mode flag of Status Register is changed to kernel mode. RETI instruction is used to return to user mode.
The Figure 1 shows the timing of external interrupt request of Brownie STD. When an interrupt request is notified to Brownie STD through EXTINT_IN, the processor core completes the preparation for interrupt processing and asserts EXTRCATCH_OUT at the start of interrupt processing. EXTINT_IN must be continuously asserted for the duration.It takes at least five cycles before the external interrupt processing is started after EXTINT_IN is asserted. The duration varies according to the state of the processor. Thus, if you negate EXTINT_IN without confirming EXTCATCH_OUT, the interrupt request may not be detected.

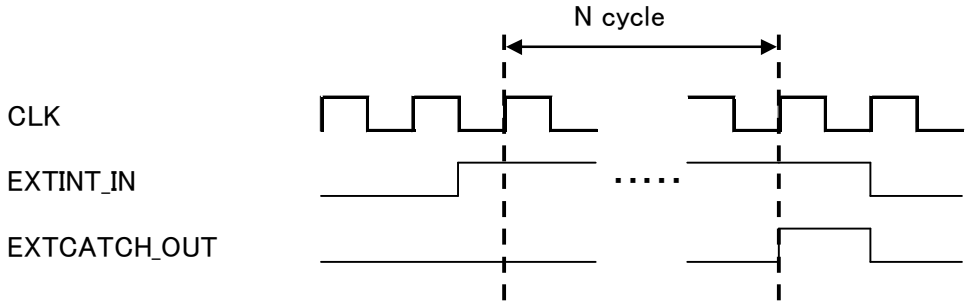


Figure 1 External interrupt request notification

3.2. Internal Interrupts

An internal interrupt can be invalidated by setting the internal interrupt permission flag bit to "0." In case of invalidation, all the internal interrupts are ignored. When the internal interrupt handler is launched, Brownie STD negates the current processing, stores the address of the negate instruction in Interrupt Return Register and branches into the internal interrupt vector.

3.2.1. TRAP

TRAP instruction can generate an interrupt at software level. When TRAP instruction is executed, Brownie STD stores the address of the instruction in Interrupt Return Register and branches into TRAP interrupt vector (0x0FFE0800 + offset). Simultaneously, the execution mode flag of Status Register is changed to kernel mode. RETI instruction is used to return to user mode.

The address stored in Interrupt Return Register by TRAP instruction is the address of the instruction that caused exceptions, namely the address of TRAP instruction itself. Thus, to return, the value of Interrupt Return Register needs to be reset to an appropriate value.

4. Memory Access

This chapter explains communication protocol between Brownie STD and external memory, timing, etc. Refer to 1.6. Interface for the list of external interfaces.

4.1. Instruction Memory

The instruction memory access is read-only. Data need to be obtained within the address transmission cycle. The following shows what the signal lines used for instruction memory access mean.

IMEM_ADDR_OUT	: Instruction memory address transmission bus
IMEM_ADDRERR_IN	: Instruction memory access error
IMEM_DATA_IN	: Instruction memory data receive bus

Figure 2 illustrates the timing of read access. The data of the address need to be determined within the cycle of the fetch address transmission. Since the fetch address is fixed while some factor is stalling the pipeline (Addr(D) in the figure), the fetch data need to be retained for the duration.

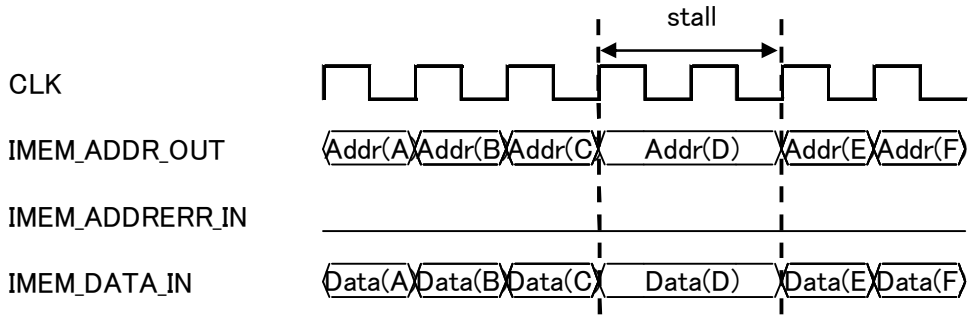


Figure 2 Instruction memory read access

4.2. Data Memory

The data memory is read/write accessible. Signal lines used for data memory access mean the following.

DMEM_ADDR_OUT	: Data memory address
DMEM_ADDRERR_IN	: Data memory access error
DMEM_DATA_OUT	: Data memory data transmission bus
DMEM_DATA_IN	: Data memory data receive bus
DMEM_RW_OUT	: Data memory R/W mode
DMEM_WMODE_OUT	: Data memory access mode
DMEM_EMODE_OUT	: Datamemory code extension mode
DMEM_CANCEL_OUT	: Data memory access cancellation
DMEM_REQ_OUT	: Data memory access request
DMEM_ACK_IN	: Data memory access completion

Table 4 Relation between memory access mode and signal value

	DMEM_RW_MODE_OUT	DMEM_WMODE_OUT	DMEM_EMODE_OUT
Signed byte read	0	"00"	1
Signed halfword read	0	"01"	1
Signed word read	0	"11"	1
Unsigned byte read	0	"00"	0
Unsigned halfword read	0	"01"	0
Unsigned word read	0	"11"	0
Byte write	1	"00"	0
Halfword write	1	"01"	0
Word write	1	"11"	0

The data memory has several access modes. The Table 4 illustrates the relation between modes and the signal values.

The Figure 3 shows data memory access timing. The example in the figure is signed word read. The address, the RW mode, the access mode and the sing extension mode are sent simultaneously with the ccess request (REQ). Then, Brownie STD retains the values until the notification of access completion (ACK). Brownie reads the data of the cycle and continues the execution after the determination of data by the memory and the ACK notification. The same timing applies to other access modes, with only control signal values varied. The Figure 4 illustrates an example of signed byte write. The write address, the write data, the RW mode, the access mode and the sign extension mode are sent during write.

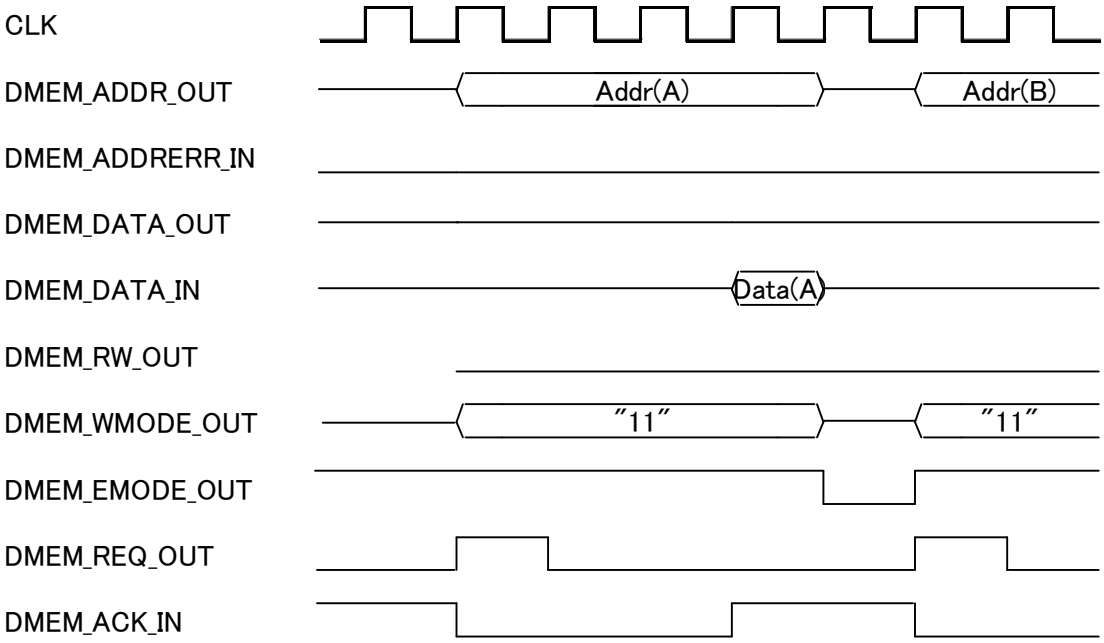


Figure 3 Signed word read

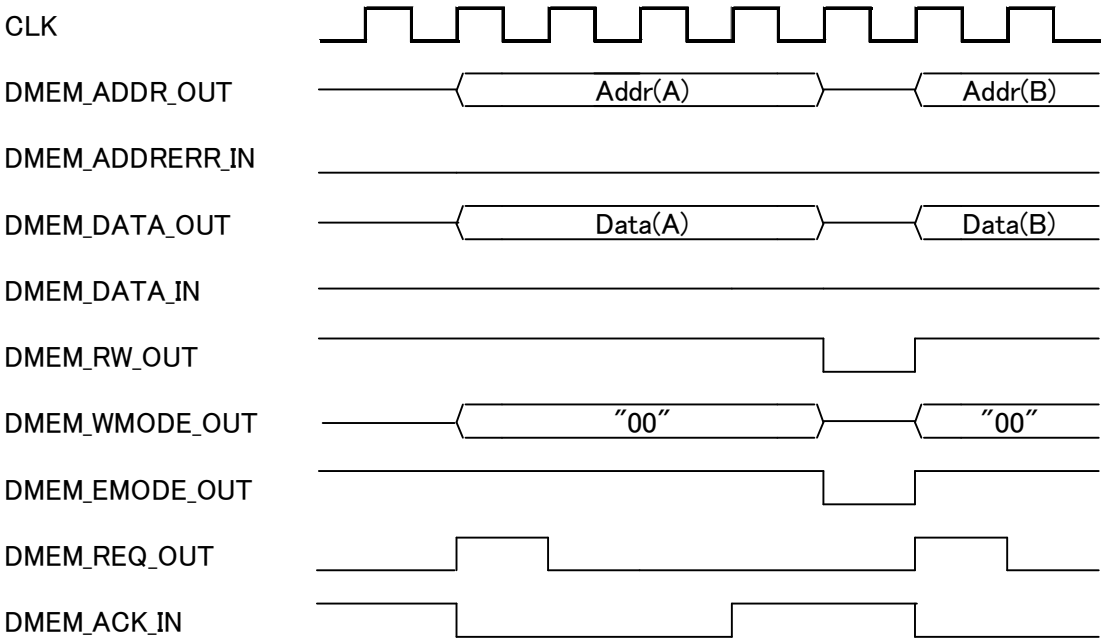


Figure 4 Signed word read example

Appendix

Appendix A. Delayed Load

In delayed load, the data loaded by a load instruction becomes usable after a given time interval (instruction count).

```
ope1   ----------------
ope2   ----------------
LW     %GPR1, 0(%GPR2)          ; Load instruction
ope3   ----------------          ; Delayed load slot
ope4   ----------------
```

The load instruction LW loads data onto GPR1.The Loaded data GPR1 is available for ope4 but unavailable for ope3 in the delayed load slot. Instructions that do not use GPR1 (either as a source or a destination) can be scheduled in the delayed load slot. When an instruction using GPR1 in the delayed load slot, data hazards occur as shown below.

```
ADD    %GPR2,%GPR0,%GPR1
LW     %GPR2,0(%GPR0)
ADD    %GPR3,%GPR0,%GPR2
```

(The loaded value is not available for an instruction immediately after a load instruction.)

```
LW     %GPR2,0(%GPR0)
ADD    %GPR2,%GPR0,%GPR1
ADD    %GPR3,%GPR0,%GPR2
```

(If the register is overwritten immediately after a load instruction, the overwriting value is valid.)

Appendix B. Status Register

Appendix B.1. Read and Write of Status Register

When read from and write to Status Register (GPR1), insert two instructions NOP for safety reasons. Below is an example.

```
ope1   ----------------
ope2   ----------------          ; should be NOP
ope3   ----------------          ; should be NOP
ope4   %GPR1, %GPRX, %GPRX       ; GPR1 is destination
ope5   ----------------
ope6   ----------------
```

If GPR1 is a destination or a source of ope4 in the above instruction sequence (in the example, ope4 is a destination), ope2 and ope3 should be NOP. If ope2 and ope3 are not NOP, read and write of Status Register can produce unintended consequences.

Appendix B.2. Reflection Timing of Write into Status Register

If Status Register (GPR1) is updated, the update is reflected form the third instruction. Below is an example.

```
ope1   %GPR1, %GPRX, %GPRX       ; GPR1 is destination
ope2   ----------------
ope3   ----------------
ope4   ----------------
```

If ope1 among the instruction sequence shown above is an instruction to have GPR1 as a destination, the value is reflected in GPR1 from ope4. If GPR1 is read by ope2 or ope3, the result is contingent.

Appendix B.3. Update Timing of Flag Register

An instruction accompanied by a flag change updates Status Register. The update of the flag is effective from the second instruction. Below is an example.

```
ope1   ----------------          ; change flag
ope2   ----------------
ope3   ----------------
ope4   ----------------
```

If ope1 among the instruction sequence shown above is an instruction to update the flag, the value is reflected in the flag register in GPR1 from ope3. If ope2 reads GPR1, the result of flag register is contingent.