RISC-V ASSEMBLY LANGUAGE

Programmer Manual
Part I

developed by: SHAKTI Development Team @ iitm '20

shakti.org.in

contact @ shakti[dot]iitm[@]gmail[dot]com

0.0.1 Proprietary Notice

0.0.2 Release Information

Version	Date	Changes
0.1	October 12, 2020	Initial Release
0.2	December 07, 2020	Updates and adding new programs
0.21	January 14, 2021	Update MUL descriptions for unsigned

Table of Contents

List of Figures

List of Tables

CSR	Control and Status Register
GP	Global Pointer
HART	Hardware Thread
IMM	Immediate Data
ISA	Instruction Set Architecture
MARCHID	Machine Architecture ID
MCAUSE	Trap cause code, Machine Mode
MCOUNTEREN	Counter enable, Machine Mode
MCYCLE	Clock cycle counter, Machine Mode
MEIP	Machine external interrupt
MEPC	Machine Exception Program counter
MHARTID	Hardware thread ID
MIE	Interrupt-enable register, Machine Mode
MIMPID	Implementation ID
MIP	Interrupt pending, Machine Mode
MISA	ISA and extensions
MSTATUS	Status register, Machine Mode
MTIP	Machine timer interrupt
MTVAL	Bad address or bad instruction, Machine Mode
MTVEC	Machine Trap Vector base address
MVENDORID	Machine Mode Vendor ID
NA	Not Applicable
NMI	Non Maskable Interrupt
RISC	Reduced Instruction Set Computer
RV128 / RV128I	Instructions present only on 128 bit machines
RV64 / RV64I	Instructions present only on 64 and 128 bit machines
RV32 / RV32I	Basic 32 bit instruction set, present on all machines
SP	Stack Pointer
TP	Thread Pointer
XLEN	Instruction (X) Length.

Table 1: List Of Abbreviations

chapter 1

Introduction

1.1 RISC-V

RISC-V pronounced as “RISC-five”, is an open-source standard Instruction Set Architecture (ISA), designed based on Reduced Instruction Set Computer (RISC) principles. With a flexible architecture to build systems ranging from a simple microprocessor to complex multi-core systems, RISC-V caters to any market. The RISC-V ISA provides two specifications, one, the User Level Instructions which guides in developing simple embedded systems and connectivity applications and two, the Privilege Level Instructions which guides in building secure systems, kernel, and protected software stacks.

RISC-V currently supports three privilege levels, viz.. Machine/Supervisor/User, with each level having dedicated Control Status Registers (CSRs) for system state observation and manipulation. In addition, RISC-V provides 31 read/write registers. While all can be used as general-purpose registers, they have dedicated functions as well. RISC-V is divided into different categories based on the maximum width of registers the architecture can support, for example, RV32 (RISC-V 32) provides registers whose maximum width is 32-bits and RV64 (RISC-V 64) provides registers whose maximum width is 64-bits. Processors with larger register widths can support instructions and data of smaller widths. So an RV64 platform supports both RV32 and RV64.

Note: This book uses the term *XLEN* to refer to the platform register width, in bits.

PART-I of the RISC-V programmer’s manual, details RISC-V assembly instructions, registers in use and the machine privilege level. Advanced concepts on Privilege levels, Memory Management unit and Trap delegation will be dealt with in PART-II of the manual.

The objective of the RISC-V ASM (assembly language) programmer manual is to aid users in writing extensive assembly programs and provide necessary information to write simple embedded applications.

1.2 Registers

RISC-V architecture provides 31 user modifiable general-purpose (base) registers, namely, $x1$ to $x31$, and with an additional read-only register $x0$, hard-wired to zero. One common use of $x0$ register is to aid in initializing other registers to zero.

In comparison to other ISAs, RISC-V uses a larger number of integer registers which helps in performance, where extensive use of loop unrolling and software pipelining is required.

In RISC-V systems, the following are the available base registers:

- There are 31 general purpose registers.
- Out of which 7 are temporary registers ($t0 - t6$).
- $a0 - a7$ are used for function arguments.
- $s0 - s11$ are used as saved registers or within function definitions.
- There is one stack pointer, one global pointer and one thread pointer register.
- A return address register ($x1$) to store the return address in a function call.
- One program counter (pc). pc holds the address of the current instruction.
- All the registers can be used as a general purpose register.

The Base registers can hold either data or a valid address and are usually identified with the letter *'x'* prefixing the register number. A brief description of the registers and their additional functions are as follows.

1.2.1 Stack Pointer Register

In RISC-V architecture, the $x2$ register is used as Stack Pointer (sp) and holds the base address of the stack. When programming explicitly in RISC-V assembly language, it is mandatory to load $x2$ with the stack base address while the C/C++ compilers for RISC-V, are always designed to use $x2$ as the stack pointer. In addition, stack base address must aligned to 4 bytes. Failing which, a load/store alignment fault may arise.

The $x2$ register can hold an operand in the following ways:

- As a base register for load and store instruction. In this case, the load/store address must be 4 byte aligned.
- As a source or destination register for arithmetic/logical/csr instructions.

1.2.2 Global Pointer Register

Data is allocated to the memory when it is globally declared in an application. Using pc-relative or absolute addressing mode leads to utilization of extra instructions, thus increasing the code size. In order to decrease the code size, RISC-V places all the global variables in a particular area which is pointed to, using the $x3$ (gp) register. The $x3$ register will hold the base address of the location where the global variables reside.

1.2.3 Thread Pointer Register

In multi-threaded applications, each thread may have its own private set of variables which are called "thread specific variables". This set of variables will be pointed to by the register $x4$ (tp). Hence, each thread will have a different value in its $x4$ register.

1.2.4 Return Address Register

The $x1$ (ra) register is used to save the subroutine return addresses. Before a subroutine call is performed, $x1$ is explicitly set to the subroutine return address which is usually `'pc + 4'`. The standard software calling convention uses $x1$ (ra) register to hold the return address on a function call.

1.2.5 Argument Register

In RISC-V, 8 argument registers, namely, $x10$ to $x17$ are used to pass arguments in a subroutine. Before a subroutine call is made, the arguments to the subroutine are copied to the argument registers. The stack is used in case the number of arguments exceeds 8.

1.2.6 Temporary Register

As the name suggests, the temporary registers are used to hold intermediate values during instruction execution. There are seven temporary registers ($t0 - t6$) in RISC-V.

Register Name	ABI Name	Description
x0	zero	Hard-Wired Zero
x1	ra	Return Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x5	t0	Temporary/Alternate Link Register
x6-7	t1-t2	Temporary Register
x8	s0/fp	Saved Register (Frame Pointer)
x9	s1	Saved Register
x10-11	a0-a1	Function Argument/Return Value Registers
x12-17	a2-a7	Function Argument Registers
x18-27	s2-s11	Saved Registers
x28-31	t3-t6	Temporary Registers

Table 1.1: RISC-V Base Integer Registers Of Size XLEN

1.3 Privilege mode

Inter-process security for a system necessitates the extent to which each process can use the system resources, to maintain the system and data integrity. These processes are grouped into different modes/levels, from low to high, and possess varying levels of privilege. Higher privilege modes have a greater system leveraging capacity in addition to their own. A mode trying to access a region it has no permission for, causes exceptions/traps. The three privilege levels are listed below,

Privilege	Value	Encoding	Abbreviation
User mode	0	00	U
Machine mode	3	11	M
Supervisor mode	1	01	S

Table 1.2: RISC-V Privilege Levels

With reference to the Table 1.2, the value field states the value of a privilege level. Encoding is used to encode the privilege level in a CSR registers. Machine level has the highest privilege and is also mandatory. Machine mode is inherently trusted, as it has low level access to the machine implementation. All software by default start in Machine Mode. This book deals with the Machine Mode. The other two modes are used for developing conventional applications and system software.

1.4 Control and Status Registers (CSRs)

The Control and Status Register (CSR) are system registers provided by RISC-V to control and monitor system states[1]. CSR's can be read, written and bits can be set/cleared. RISC-V provides distinct CSRs for every privilege level. Each CSR has a special name and is assigned a unique function. In addition to the machine level CSRs described in this section, M-mode code can access the CSRs at lower privilege levels. Other privilege levels and related CSR's are dealt with in part 2 of the manual.

Reading and/or writing to a CSR will affect processor operation. CSR's are used in operations, where a normal register cannot be used. For example, knowing the system configuration, handling exceptions, switching to different privilege modes and handling interrupts are some tasks for which a CSR is needed. The CSR cannot be read/written the way a general register can. A special set of instructions called `csr instructions` are used to facilitate this process. CSR instructions require an intermediate base register to perform any operation on CSR registers. Further, it is possible to write immediate values to CSR registers. table1.3 lists the CSRs present in machine mode.

1.4.1 CSR Field Specifications

An attempt to access a CSR that is not visible in the current mode of operation results in privilege violation. Similarly, in the current mode of operation, a privilege violation occurs when an attempt is

[1]Here, system/processor refers to a computing system built using RISC-V ISA

Register	Description
misa	Machine ISA
mvendorid	Machine Vendor ID
marchid	Machine Architecture ID
mimpid	Machine Implementation ID
mstatus	Machine Status
mcause	Machine trap cause
mtvec	Trap vector base address

Register	Description
mhartid	Machine Hardware thread ID
mepc	Machine exception program counter
mie	Machine interrupt enable
mip	Machine interrupt pending
mtval	Machine trap value
mscratch	Scratch register

Table 1.3: RISC-V Machine Mode Registers

made to write to a "read-only" labeled CSR. This attempt results in an illegal instruction exception. In addition to restrictions on how a CSR register is accessed, fields within some registers come with their own restrictions which are as listed as follows.

1.4.1.1 Reserved Writes Ignored, Reads Ignore Values (WIRI)

Read-only fields within some read-only and read/write registers, have been reserved for future use. Such fields have been named as `Reserved Writes Ignored, Reads Ignore Values (WIRI)`. A read or write to these fields must be ignored. In case the entire CSR is a read-only register, an attempt to write to the WIRI field will raise an illegal instruction exception.

1.4.1.2 Reserved Writes Preserve Values, Reads Ignore Values (WPRI)

Although, there are fields labeled "read/write" in some registers, they are reserved for future use and are not available for software modifications. Such fields are called as `Reserved Writes Preserve Values, Reads Ignore Values (WPRI)`. Values returned on a reading such fields must be ignored, while an attempt to write to the whole register containing such fields must preserve the original value.

1.4.1.3 Write/Read Only Legal Values (WLRL)

Some fields restrict the values that can be read/written to a field. Such values are called "legal" values and are specified by the processor. Fields with this restriction are labeled as `Write/Read Only Legal Values (WLRL)`. A read on such a field returns a legal value if legal values are written to it. Caution should be exercised to write only legal values as illegal writes may not return legal values.

1.4.1.4 Write Any Values, Reads Legal Values (WARL)

Some read/write fields offer the freedom of writing any value to it while reading them, will only return values which are legal. Such fields are labeled as `Write Any Values, Reads Legal Values (WARL)`. Implementations will not raise an exception on writes of unsupported values to an WARL field. Implementations must always deterministically return the same legal value after a given illegal value is written.

1.5 CSR Instructions

CSR instructions are used to read and write to CSR registers. These instructions are broadly classified as register-register and register-immediate instructions.

1.5.1 Register to Register instructions

Register-register instructions perform indicated operations on two registers of the system and leaves the result in the specified register.

1.5.1.1 CSRRC

`CSR Read and Clear Bits (CSRRC)` is used to clear a CSR.

Syntax

`csrrc rd, csr,` rs_1

Alias

`csrc csr,` rs_1

where,

rd	destination register
csr	csr register
rs_1	source register 1

Description

The `CSRRC` instruction clears bits of the specified CSR. It can be used to simply read a CSR without updating it. If (rs_1) is x0, then no update to the CSR will occur. The previous value of the CSR is copied to the destination register and then some selected bits of the CSR are cleared to 0, the value in (rs_1) is used as a bit mask to select which bits are to be cleared in the CSR. Other bits are unchanged. This is an atomic operation.

Usage

```
csrrc x1, mcause, zero        # mcause ⟵ (Invert (zero) Logical-AND mcause)
                              # x1 ⟵ old value of mcause
```

1.5.1.2 CSRR

`CSR Read (CSRR)` is used to read from a CSR.

Syntax

`csrr rd, csr`

where,

rd	destination register
csr	csr register

Description

The `CSRR` instruction is used to read the value of CSR. The previous value of the CSR is copied to the destination register. This is an atomic read operation.

Usage

```
csrr x5, mstatus        # x5 ⟵ mstatus
```

1.5.1.3 CSRRW

`CSR Read and Write (CSRRW)` is used to read from and/or write to a CSR.

Syntax

`csrrw rd, csr,` rs_1

Alias

`csrw csr,` rs_1

where,

rd	destination register
rs_1	source register 1
csr	csr register

Description

The previous value of the CSR is copied to destination register and the value of the source register (rs_1) is copied to the CSR, this is an atomic write operation. To read a CSR without writing to it, the source register (rs_1) can be specified as x0. To write a CSR without reading it, the destination register (rd) can be specified as x0. This is an atomic operation.

Usage

```
auipc t0, %pcrel_hi(mtvec)
addi t0, t0, %pcrel_lo(1b)
csrrw zero, mtvec, t0          # mtvec ⟵ t0
```

Exceptions

In lower privilege modes some of the CSRs are inaccessible. An attempt to read from or write to those CSR may cause an illegal instruction exception.

1.5.1.4 CSRRS

`CSR Read and Set Bits (CSRRS)` sets bits in the specified CSR.

Syntax

csrrs rd, csr, rs_1

Alias

`csrr rd, csr`

where,

rd	destination register
csr	csr register
rs_1	source register 1

Description

The `CSRRS` instruction can be used to simply read a CSR without updating it. If (rs_1)is x0, then no update to the CSR will occur. The previous value of the CSR is copied to the destination register and then some selected bits of the CSR are set to 0. The value in (rs_1) is used as a bit mask to select which bits are to be set in the CSR. Other bits are unchanged. This is an atomic operation.

Usage

```
csrrs zero, mstatus, x1        # mstatus ⟵ (x1 (Logical-OR) mstatus)
```

1.5.2 Immediate Instructions

1.5.2.1 CSRRCI

`CSR Read and Clear Immediate (CSRRCI)` clears any CSR using a zero-extended immediate value (imm[4:0]) encoded in the rs_1 field, instead of a value from an integer register.

Syntax

`csrrci rd, csr, imm`

Alias

`csrci csr, imm`

where,

rd	destination register
csr	csr register
imm	immediate value

Description

The `CSRRCI` instruction makes bits[4:0] in any CSR particularly easy to modify. The previous value of the CSR is copied to the destination register and then the CSR is cleared using immediate value. The 5-bit field that is normally used for rs_1 is zero-extended and used as the source value that is moved into the CSR. This is an atomic operation.

Usage

```
csrrci x1, mie, 3          # mie ⟵ (3 (Logical-AND) mie)
                           # x1 ⟵ old value mie
```

1.5.2.2 CSRRSI

`CSR Read and Set bits Immediate (CSRRSI)` can be used to make bits [4:0] in any CSR particularly easy to set "1".

Syntax

`csrrsi rd, csr, imm`

Alias

`csrsi csr, imm`

where,

rd	destination register
csr	csr register
imm	immediate value

Description

The `CSRRSI` instruction makes bits[4:0] in any CSR particularly easy to set to "1". The previous value of the CSR is copied to the destination register and then some selected bits of the CSR are set to 1. The 5-bit field that is normally used for rs_1 is zero-extended and used as a bit mask to select which bits are to be set in the CSR. This is an atomic operation.

Usage

```
csrrsi zero, mstatus, 3          # mstatus ⟵ (3 (Logical-OR) mstatus)
```

1.5.2.3 CSRRWI

`CSR Read and Write bits Immediate (CSRRWI)` copies the old value of a csr, then overwrites the csr with the specified immediate value.

Syntax

`csrrwi rd, csr, imm`

Alias

`csrwi csr, imm`

where,

rd	destination register
csr	csr register
imm	immediate value

Description

The `CSRRWI` is a variant of the `CSRRW` instruction, which is used to overwrite to a csr with the specified immediate value. The previous valueof the csr is copied to the destination register and then the entire csr is written to. The 5-bit field that is usually used for source register (rs_1) is zero-extended and used as the immediate value that is moved into the register. This is an atomic operation.

Usage

```
                                # x5 ⟵ old value of mstatus)
csrrwi x5, mstatus, 3           # mstatus ⟵ 3
```

1.5.3 Machine Information Registers

1.5.3.1 MISA

`Machine Instruction Set Architecture (MISA)` register lists the basic architecture of the RISC-V processor.

XLEN-1 XLEN-2	XLEN-3 26	25 0
MXL[1:0] (**WARL**)	**WIRI**	Extensions[25:0] (**WARL**)
2	XLEN-28	26

Figure 1.1: Machine ISA Register (`misa`)

Description

`MISA` also informs the register width and the implementation of RISC-V extensions. Individual bits in this CSR indicate the various options and extensions detailed by the RISC-V specification have been implemented.

I	Base Integer Instruction Set
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
C	Standard Extension for Compressed Instructions
S	Standard Extension for Supervisor mode
L	Standard Extensions for Decimal arithmetic instructions

Table 1.4: RISC-V ISA extensions

The register width of the machine is encoded in the most significant two bits of this CSR. The `MISA` register shows the widest register width, the core is capable of running. For example, an RV64 machine may be capable of running as an RV32 machine.

Off the 32 bits, the lower-order 26 bits correspond to the letters A, B, . . . , Y, Z (“A”=bit 0, “B”=bit 1, etc.). Each bit will be set to indicate whether a particular RISC-V extension is implemented in the core. For example, bit 5 will be set if the core supports the “F” extension.

Operation	ASM_Command	Usage
Read	csrr rd, misa	csrr x5, misa
Write	NA	NA
Set	NA	NA
Clear	csrrc rd, misa, rs_1	csrrc x0, misa, x5

Table 1.5: Basic Commands and Usage with misa Register

1.5.3.2 MVENDORID

`Machine Vendor Id (MVENDORID)` identifies the manufacturer of the RISC-V chip.

XLEN-1 7	6 0
Bank	Offset
XLEN-7	7

Figure 1.2: Machine VendorID register (`mvendorid`)

Description

`MVENDORID` stores the Identity number assigned to a vendor by the semiconductor engineering trade organization called `JEDEC`. Research and non-commercial implementations will have zero encoded.

Operation	ASM_Command	Usage
Read	csrr rd, mvendorid	csrr x5, mvendorid
Write	NA	NA
Set	NA	NA
Clear	NA	NA

Table 1.6: Basic Commands and Usage with mvendorid Register

1.5.3.3 MARCHID

`Machine Architecture Id (MARCHID)` identifies the particular architecture of the part and is essentially the "part number" or "model number".

XLEN-1   0
Architecture ID
XLEN

Figure 1.3: Machine Architecture ID Register (`marchid`).

Description

For commercial designs, this number is assigned by the vendor. For some non-commercial or open-source projects, a number may be assigned by the RISC-V Foundation. Otherwise, this register will contain zero.

Operation	ASM_Command	Usage
Read	csrr rd, marchid	csrr x5, marchid
Write	NA	NA
Set	NA	NA
Clear	NA	NA

Table 1.7: Basic Commands and Usage with marchid Register

1.5.3.4 MIMPID

`Machine Implementation Id (MIMPID)` identifies the particular implementation or version of the processor.

XLEN-1	0
Implementation	
XLEN	

Figure 1.4: Machine Implementation ID Register (`mimpid`).

Description

Given a particular vendor (as identified in mvendorid) and a part/model number (as identified in marchid), there may be several versions. It may be zero.

Operation	ASM_Command	Usage
Read	csrr rd, mimpid	csrr x5, mimpid
Write	NA	NA
Set	NA	NA
Clear	NA	NA

Table 1.8: Basic Commands and Usage with mimpid Register

1.5.3.5 MHARTID

`Machine Hardware Thread Id (MHARTID)` identifies which core is executing.

XLEN-1	0
Hart ID	
XLEN	

Figure 1.5: Hart ID Register (`mhartid`).

Description `MHARTID` register does not reflect a higher level (eg., operating system) concept of thread. In a single-core system with a single, simple FETCH-DECODE-EXECUTE pipeline, there only one HART. In a multi-core system, where each core will execute a single flow-of-control, each core will have its own HART. Each core's HART will execute concurrently with the other cores' HARTs.

It may be important to identify one thread as a “master thread”. One HART must be given an ID of zero. The number of hardware threads is fixed but the application software will need an unpredictable and changing number of threads. The OS will map traditional OS threads onto the available hardware threads.

Operation	ASM_Command	Usage
Read	csrr rd, mhartid	csrr x5, mhartid
Write	NA	NA
Set	NA	NA
Clear	NA	NA

Table 1.9: Basic Commands and Usage with mhartid Register

1.5.3.6 MSTATUS

`Machine STATUS (MSTATUS)` register details the machine status and helps in manipulating the state of the machine. The mstatus register has several bits to operate the different states of the machine.

63 ... 11	10 9	8	7	6	5	4	3	2	1	0
... ...	**WPRI**		MPIE	**WPRI**			MIE	**WPRI**		

Figure 1.6: Machine-Mode Status Register (`mstatus`) for RV64

31 ... 11	10 9	8	7	6	5	4	3	2	1	0
... ...	**WPRI**		MPIE	**WPRI**			MIE	**WPRI**		
	2	1	1	1	1	1	1	1	1	1

Figure 1.7: Machine-Mode Status Register (`mstatus`) for RV32.

Description

`MSTATUS` contains a number of fields that can be read and updated. By modifying these fields, the software can do things like enable/disable interrupts and change the virtual memory model.

Operation	ASM_Command	Usage
Read	csrr rd, mstatus	csrr x5, mstatus
Write	csrrw mstatus, rs_1	csrrw x0, mstatus, x5
Set	csrrs mstatus, rs_1	csrrs x0, mstatus, x5
Clear	csrrc mstatus, rs_1	csrrc x0, mstatus, x5

Table 1.10: Basic Commands and Usage with mstatus Register

For example, by writing to this CSR, the software can turn on virtual memory and page-table translation. Two of the fields are only used for 64 and/or 128 bit machines. These two fields reside in bits positions [35:32], so they are not even present in 32-bit machines.

1.5.3.7 MCAUSE

`Machine CAUSE (MCAUSE)` register contains the reason for the exception or interrupt that happened in the system.

XLEN-1	XLEN-2 0
Interrupt	Exception Code (WLRL)
1	XLEN-1

Figure 1.8: Machine Cause Register (`mcause`).

Description

When a trap is taken into Machine mode, `MCAUSE` is written by hardware with a code indicating the event that caused the trap. The list of numeric codes are listed below,

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Available for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*

Table 1.11: Machine cause register (`mcause`) values after trap.

Operation	ASM_Command	Usage
Read	csrr rd, mcause	csrr x5, mcause
Write	csrrw rd, mcause, rs_1	csrrw x0, mcause, x5
Set	csrrs rd, mcause, rs_1	csrrs x0, mcause, x5
Clear	csrrc rd, mcause, rs_1	csrrc x0, mcause, x5

Table 1.12: Basic Commands and Usage with mcause Register

1.5.3.8 MTVEC

`Machine Trap Vector Base Address (MTVEC)` register is used to store the address of the Trap handler.

XLEN-1 2	1 0
BASE [XLEN-1:2] (WARL)	MODE (WARL)
XLEN-2	2

Figure 1.9: Machine Trap-Vector Base-Address Register (`mtvec`)

Value	Name	Description
0	Direct	All exceptions set `pc` to BASE.
1	Vectored	Interrupts set `pc` to BASE+4×cause.
≥2	—	*Reserved*

Table 1.13: Encoding of `mtvec` MODE field.

Description

The `MTVEC` register has the address of the trap handler. When a trap occurs (and is to be handled, not ignored), the Hardware set's the program counter (PC) set to the value in the `MTVEC` register. This causes a jump to the first instruction in the trap handler routine.

Operation	ASM_Command	Usage
Read	csrr rd, mtvec	csrr x5, mtvec
Write	csrrw rd, mtvec, rs_1	csrrw x0, mtvec, x5
Set	csrrs rd, mtvec, rs_1	csrrs x0, mtvec, x5
Clear	csrrc rd, mtvec, rs_1	csrrc x0, mtvec, x5

Table 1.14: Basic Commands and Usage with mtvec Register

1.5.3.9 MEPC

`Machine Exception Program Counter (MEPC)` is an XLEN-bit read/write register, which holds the address of the instruction which resulted in a trap.

XLEN-1   0
mepc
XLEN

Figure 1.10: Machine Exception Program Counter Register (`mepc`).

Description

When a trap (exception) is taken into machine mode, the virtual address of the instruction which resulted in an exception, is written into the mepc register. It serves the same purpose for the exception handler that the return address (ra) register serves for subroutine calls. There can be certain traps, which can lead to system halt. In that case, `MEPC` cannot be used to return back.

Operation	ASM_Command	Usage
Read	csrr rd, mepc	csrr x5, mepc
Write	csrrw rd, mepc, rs_1	csrrw x0, mepc, x5
Set	csrrs rd, mepc, rs_1	csrrs x0, mepc, x5
Clear	csrrc rd, mepc, rs_1	csrrc x0, mepc, x5

Table 1.15: Basic Commands and Usage with mepc Register

Exceptions

`MEPC` register cannot hold a program counter (pc) value that would cause an *Instruction Address Misaligned* exception.

1.5.3.10 MIE

`Machine Mode Interrupt Enable (MIE)` is an XLEN read/write register, containing interrupt enable bits. Bits which are read-only, are hardwired to 0.

15... ...12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIE	0		0	MTIE	0		0	MSIE	0		0
4	1	1	1	1	1	1	1	1	1	1	1	1

Figure 1.11: Standard portion (bits 15:0) of `mie`.

Description

The `MIE` register has a list of bits to enable/disable interrupts. Using this register, individually Timer, Software and External interrupts can be controlled. `MIE`. For the bits in the `MIE` register to take effect, the *MIE* bit in `MSTATUS` register has to be set. In general, the *MIE* bit in `MSTATUS` controls the interrupt at global level. The bits in `MIE` register control interrupt at local level.

Operation	ASM_Command	Usage
Read	csrr rd, mie	csrr x5, mie
Write	csrrw rd, mie, rs_1	csrrw x0, mie, x5
Set	csrrs rd mie, rs_1	csrrs x0, mie, x5
Clear	csrrc rd, mie, rs_1	csrrc x0, mie, x5

Table 1.16: Basic Commands and Usage w.r.t MIE Register

1.5.3.11 MIP

`Machine Mode Interrupt Pending (MIP)` is an XLEN-bit read/write register which hols the information regarding interrupts which are pending.

15–12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIP	0		0	MTIP	0		0	MSIP	0		0
4	1	1	1	1	1	1	1	1	1	1	1	1

Figure 1.12: Standard portion (bits 15:0) of `MIP`.

Description

The `MIP` pending interrupt requests. The interrupt cause number, as reported in the `MCAUSE`, corresponds with the same bit in the `MIP` register. An interrupt will be considered if the particular bit is set both in `MIP` and `MIE`, and when the interrupts are globally enabled. Individual bits in `MIP` maybe writable or read-only. When the bit is writable, the pending interrupt can be cleared once the interrupt is addressed. In case the bits are read-only, the implementation must provide means to clear the pending interrupt.

Operation	ASM_Command	Usage
Read	csrr rd, mip	csrr x5, mip
Write	csrrw rd, mip, rs_1	csrrw x0, mip, x5
Set	csrrs rd, mip, rs_1	csrrs x0, mip, x5
Clear	csrrc rd, mip, rs_1	csrrc x0, mip, x5

Table 1.17: Basic Commands and Usage with MIP Register

Exceptions

Since the non-maskable interrupt is implicit, when executing the non-maskable interrupt (NMI) handler, it is not made visible in MIP.

1.5.3.12 MTVAL

The `Machine Trap Value (MTVAL)` register holds exception specific information.

XLEN-1 … 0
mtval
XLEN

Figure 1.13: Machine Trap Value register (`mtval`).

Description

When an exception is encountered, this register can hold exception-specific information to assist software in handling the trap. In the case of errors in the load-store unit `MTVAL` holds the address of the transaction causing the error. If this transaction is misaligned, the `MTVAL` holds the address of the missing transaction part. In the case of illegal instruction exceptions, it holds the actual faulting instruction. For all other exceptions, `MTVAL` register is 0.

Operation	ASM_Command	Usage
Read	csrr rd, mtval	csrr x5, mtval
Write	csrrw rd, mtval, rs_1	csrrw x0, mtval, x5
Set	csrrs rd, mtval, rs_1	csrrs x0, mtval, x5
Clear	csrrc rd, mtval, rs_1	csrrc x0, mtval, x5

Table 1.18: Basic Commands and Usage with mtval Register

1.5.3.13 MSCRATCH

A `Scratch Register (MSCRATCH)` for Machine Mode Trap Handler. This register allows us to store the context of trap handlers in other privilege levels. This is of much use only in case of system switching privilege modes.

XLEN-1 0
mscratch
XLEN

Figure 1.14: Machine-mode scratch Register (`mscratch`).

Description

- In order to prevent overwrite and lose of the previous values, when a machine mode trap handler is invoked, the use of at least one general purpose register is needed.
- `MSCRATCH` gives the software a register loaded with a base value, which can subsequently be used to save all remaining processor state.
- Mostly, it may contain a frame or stack pointer to the "register save area".

Operation	ASM_Command	Usage
Read	csrr rd , mscratch	csrr x5, mscratch
Write	csrrw rd, mscratch, rs_1	csrrw x0, mscratch, x5
Set	csrrs rd, mscratch, rs_1	csrrs x0, mscratch, x5
Clear	csrrc rd, mscratch, rs_1	csrrc x0, mscratch, x5

Table 1.19: Basic Commands and Usage with mscratch Register

Exceptions

`MSCRATCH` is a read/write Register, which is never used directly by the hardware. It only serves as an XLEN bit temporary scratch space to be used by the machine mode software. It is protected from other privilege modes and can be accessed without destroying contents of any register using CSR swap instructions.

chapter

2 Load and Store instructions

This section of manual covers the memory access instructions available in RISC-V Architecture. There are different instructions available for 8 bit, 16 bit, 32 bit and 64 bit access.

2.1 RV 32I

RV32I deals with the 32 bit instruction that are used for load and store operations. The instructions are broadly classified as register-register and immediate instructions

2.1.1 Load-Store Instructions

Load-store instructions transfer data between memory and processor registers. The LW instruction loads a 32-bit value from memory into the destination register (rd). LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in rd. LB and LBU are for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register to memory.

The load or store address should always aligned for each data type (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The processor will generate a misaligned access, if the addresses are not aligned properly. If the load or store instruction tries to access an invalid memory, a load/store access fault is generated. An invalid memory can arise because of PMP access controls or unavailable memory address.

2.1.1.1 LB

The `Load Byte (LB)` instruction, moves a byte from memory to register. The instruction is used for signed integers.

Syntax

`lb rd, imm(`rs_1`)`

where,

rd	destination register
imm	immediate data
rs_1	source register 1

Description

The `LB` is a data transfer instruction, defined for 8-bit values. It works with signed integers and places the result in the LSB of rd and fills the upper bits of rd with copies of the sign bit.

Usage

```
lb x5, 40(x6)       # x5 ⟵ valueAt[x6+40]
```

2.1.1.2 LBU

The `Load Byte, Unsigned (LBU)` instruction, moves a byte from memory to register. The instruction is used for unsigned integers.

Syntax

`lbu rd, imm(`rs_1`)`

where,

rd	destination register
imm	immediate data
rs_1	source register 1

Description

The `LBU` instruction, is defined for 8-bit values. It works with unsigned integers and places the result in the LSB of rd and zero-fills the upper bits of rd.

Usage

```
lbu x5, 40(x6)       # x5 ⟵ valueAt[x6+40]
```

2.1.1.3 LH

In RISC-V 16-bit numbers are known as half-words and the `Load Half-Word signed (LH)` instruction, loads a half-word from memory to register. The instruction is used for signed integers.

Syntax

`lh rd, imm(`rs_1`)`

where,

rd	destination register
imm	immediate data
rs_1	source register

Description

The `LH` instruction, treats the half-word as a signed number and loads a half-word from memory, placing it in the rightmost 16-bits of a register rd while the leftmost 48-bits of the register rd are sign extended.

Usage

```
lh x5, 0(x6)       # x5 ⟵ valueAt[x6+0]
```

2.1.1.4 LHU

`Load Half-Word Unsigned (LHU)` instruction, loads a half-word from memory to register. The instruction is used for unsigned numbers.

Syntax

`lhu rd, imm(`rs_1`)`

where,

rd	destination register
imm	immediate data
rs_1	source register 1

Description

The `LHU` instruction, treats the half-word as an unsigned number and loads it from memory, placing it in the rightmost 16-bits of a register rd while the leftmost 48-bits of the register rd are filled with zeros.

Usage

```
lhu x5, 0(x6)       # x5 ⟵ valueAt[x6+0]
```

2.1.1.5 LW

The `Load Word (LW)` instruction, moves a word, 32-bit value, from memory to register. The instruction is used for signed values.

Syntax

`lw rd, imm(`rs_1`)`

where,

rd	destination register
imm	immediate data
rs_1	source register 1

Description

The `LW` instruction, is defined for 32-bit values. It works with signed integers and places the result in the LSB of rd and fills the upper bits of rd with copies of the sign bit.

Usage

```
lw x5, 40(x6)       # x5 ⟵ valueAt[x6 + 40]
```

2.1.1.6 SB

`Store Byte (SB)` instruction, stores 8-bit values from a register to memory.

Syntax

`sb` rs_2`, offset(`rs_1`)`

where,

rs_1	base register
rs_2	source register
$offset$	12-bit integer value

Description

The `SB` is a store type instruction which stores 8-bit values from the low bits of a register rs_2 to memory. The low-order byte of the register rs_2 is copied to memory while the rest of the register is ignored and is unchanged. The address to which the byte will be stored to in the memory, is calculated at run time by adding an `offset` to a rs_1.

Usage

```
sb x1, 0(x5)       # x1 ⟵ valueAt[x5 + 0]
```

Store the 8-bit value in x1 register to location pointed to by x5.

2.1.1.7 SH

`Store Half-word (SH)` instruction, stores 16-bit values from a register to memory.

Syntax

`sh` rs_2`, offset(`rs_1`)`

where,

rs_1	base register
rs_2	source register
$offset$	12-bit integer value

Description

The `SH` is a store type instruction which stores 16-bit values from the low bits of a register rs_2 to memory. The low-order half-word of the register rs_2 is copied to memory while the rest of the register is ignored and is unchanged. The address to which the half-word will be stored to in the memory, is calculated at run time by adding an `offset` to a base register.

Usage

Store the 16-bit value in x1 register to location pointed to by x5.

```
sh x1, 0(x5)       # x1 ⟵ valueAt[x5 + 0]
```

2.1.1.8 SW

`Store Word (SW)` instruction, stores 32-bit values from a register to memory.

Syntax

`sw` rs_2`, offset(`rs_1`)`

where,

rs_1	base register
rs_2	source register
$offset$	12-bit integer value

Description

The `SW` is a store type instruction which stores 32-bit values from the low bits of register rs_2 to memory. The word from the register rs_2 is copied to memory. The address to which the word will be stored to in the memory, is calculated at run time by adding an `offset` to a base register.

Usage

Store the 32-bit value in x1 register to location pointed to by x5.

```
sw x1, 0(x5)       # mem[x5 + offset] ⟵ x1
```

2.1.2 Immediate instructions

Immediate instructions are those which contain the actual data to be operated upon, rather than the addresses of the data. It is directly encoded as part of an instruction.

2.1.2.1 LUI

The `Load Upper Immediate (LUI)` instruction, copies the 20-bit immediate value to the upper 20 bits of the destination register (*rd*) and resets the lower 12 bits to zero.

Syntax

`lui rd, imm`

where,

rd	destination register
imm	immediate Data

Description

The `LUI` instruction, copies the immediate value to the upper 20 bits of the destination register (*rd*). The lower 12 bits of the destination register is reset to zero. This instruction is usually used, when a register needs to be populated with a large value. The immediate value can be represented in hexadecimal or decimal format. In a RV64 systems, the most significant bit is sign extended to fill the most significant 32 bits (bits 63 - 32) 2.1.2.1. The destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register.

Usage

```
                        # imm = 0x11000
lui x5, 0x11000         # x5 ⟵ 0x11000
```

Assuming *x5* was zero before this instruction. *x5* will have a value 0x11000000, after executing above instruction.

```
                        # imm = 0x80011
lui x5, 0x80011         # x5 ⟵ 0x80011
```

Assuming *x5* was zero before this instruction. In RV64 systems, *x5* will have a value `0xffffffff80011000`, after executing above instruction. This example, further demonstrates that least 12 bits are always reset to zero.

2.1.2.2 AUIPC

`Add Upper Immediate to PC (AUIPC)` adds the 20-bit immediate value to the upper 20 bits of the program counter (*pc*) and stores the result in the destination register (*rd*).

Syntax

`auipc rd, imm`

where,

rd	destination register
imm	immediate value

Description

`AUIPC` is used to build pc-relative addresses. `AUIPC` forms a 32-bit temporary offset, by adding the 20-bit immediate value to the upper 20 bits of temporary offset, filling in the lower 12 bits with zeros. The temporary offset is added to the *pc*, to form the pc-relative address. The result is placed in the destination register (rd). In a 64 bit architecture, the temporary offset is sign extended and added to pc. The destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register.

Usage

Assuming *pc* is at `0x800000ff`.

```
auipc x5, 0x00110        # imm = 0x00110
                         # x5  ⟵ 0x00110000 + 0x800000ff
```

x5 will have `0x801100ff`.

Another example needed, which demonstrates that least 12 bits are unaffected is needed.

2.2 RV 64I

RV 64I deals with the 64 bit instructions that are used for load and store operations. The instructions are broadly classified as register-register and immediate instructions

2.2.1 Load-Store Instructions

Load-store instructions transfer data between memory and processor registers. The LD instruction loads a 64-bit value from memory into the destination register (rd). The SD instructions store 64-bit value in the register to memory.

The load or store address should always aligned for 64 bits. The processor will generate a misaligned access, if the addresses are not aligned properly.

2.2.1.1 LD

The `Load Double word (LD)` instruction does the fetching of 64-bit value from memory and loads into the destination register (rd).

Syntax

`ld rd, offset(`rs_1`)`

Description

A 64-bit value is fetched from memory and loaded into destination register, the memory address is formed by adding the offset to the contents of (rs_1). This instruction is available only for 64-bit and 128-bit machines.

Usage

```
ld x4, 1352(x9)        # x4 ⟵ valueAt[x9+1352]
```

2.2.1.2 SD

The `Store Double word (SD)` instruction does the copying of 64-bit value from register (rs_2) and loads into the memory(rs_1).

Syntax

`sd` rs_2`, offset(`rs_1`)`

Description

A 64-bit value is copied from register (rs_2) and loaded into memory. The memory address is formed by adding the offset to the contents of (rs_1). For a 128-bit machine the upper bits of the register are ignored. This instruction is available only for 64-bit and 128-bit machines.

Usage

```
sd x4, 1352(x9)        # mem[x9+1352] ⟶ x4
```

2.2.2 LWU

The `Load Word Unsigned (LWU)` instruction does the fetching of 32-bit value from memory and loads into the destination register (rd).

Syntax

`lwu rd, offset(`rs_1`)`

Description

A 32-bit value is fetched from memory and moved into destination register, the memory address is formed by adding the offset to the contents of (rs_1). 32-bit registers machine don't require either signextension or zeroextension is necessary for value that is already 32 bits wide, therefore the "signed load" instruction `LW` does the same thing as the "unsigned load" instruction `LWU`, making `LWU` redundant. This instruction is available only for 64-bit and 128-bit machines.

Usage

```
lwu x4,1352(x9)        # x4 ⟵valueAt[x9+1352]
```

2.3 Pseudo Instructions

RISC-V provides several pseudo-instructions which are simple to understand, easy to use and translate or expand to their base instructions. Pseudo instructions supported by RISC-V have the format shown as follows.

`OpCode destination_register, source_register`

Where content of the source register is copied into the destination register, and is read as,

`destination_register` $\longleftarrow$ `source_register`

2.3.1 Load pseudo instructions

2.3.1.1 MV

`Move (MV)` instruction to copy contents of one register to another.

Syntax

`mv rd,` rs_1

Translation

`addi rd,` rs_1`, 0`

where,

rs_1	source register 1
rd	destination register

Usage

```
mv x6, x5        # x6 ⟵ x5
```

Description

`Move (MV)` instruction is a simple "Copy Register", assembler pseudo-instruction which copies the contents of one register to another register. This assembler pseudo-instruction translates to add immediate `ADDI` instruction. This instruction translates to addi x6, x5, 0. Assuming x5 has a value 3 and x6 is initialized to 0, after move instruction, x6 will have the value 3.

2.3.1.2 LI

The `Load Immediate (LI)` loads a register (rd) with an immeidate value given int the instruction.

Syntax

li rd, CONSTANT

Description

The `LI` instruction loads a register (rd) with an integer value. With this instruction both positive and negative values can be loaded into the register.

Usage

```
li x5,100        # x5 ⟵100
li x5,-170       # x5 ⟵-170
```

2.3.1.3 LA

The `Load Address (LA)` loads the location address of the specified SYMBOL.

Syntax

la rd, SYMBOL

Description

The `LA` directive is an assembler pseudo-instruction which computes a pointer-sized effective address of the SYMBOL, but does not perform any memory access. The effective address itself is then stored in register *rd*. Depending on the addressing mode, the instruction expands to

```
lui rd, SYMBOL[31:12]
addi rd, t0, SYMBOL[11:0]
```

where SYMBOL[31:12] is the upper 20 bits of SYMBOL, and SYMBOL[11:0] is the lower 12 bits of SYMBOL.

Usage

```
.data
NumElements:  .byte 6
.text
la x5, NumElements          # x5 ⟵ addr[NumElements]
```

As an example, 'NumElements' SYMBOL has a location address '10010074'. When `LA` is given, this address, '10010074' is loaded into register x5.

2.3.1.4 SEXT.W

`Sign Extend Word (SEXT.W)` instruction sign extends a 32-bit value to 64-bits or 128-bits.

Syntax

`sext.w rd,` rs_1

where,

rs_1	source register 1
rd	destination register

Translation

`addiw` rd`,` rs_1`, x0`

Description

`SEXT.W` is an assembler pseudo-instruction which is available only for 64-bit and 128-bit machines. This instruction sign extends the lower 32 bits of value in rs_1 to 64 or 128 bits with the result being placed in the register rd. `SEXT.W` is useful when a 32-bit signed value must be extended to a larger value on 64-bit or 128-bit machine.

Usage

```
sext.w x6, x5        # x6 ⟵ x5
```

Assuming register x5 is loaded with value `0xfda961a6e88e974d`, `SEXT.W` sign extends this value to `0xffffffffe88e974d`, and is stored in x6. As this instruction translates to `ADDIW`, the sign extension translates to, x6 = x5+0

2.3.1.5 NEG

`Negate (NEG)` instruction computes two's complement of a value.

Syntax

`neg rd,` rs_1

Translation

`sub rd, x0,` rs_1

where,

rs_1	source register 1
rd	destination register

Description

`NEG` instruction arithmetically negates the contents of rs_1 and places the result in register rd. This instruction translates to instruction `Subtraction (SUB)` where the contents of rs_1 is subtracted from zero.

Usage

```
neg x6, x5        # x6 ⟵ x5
```

Assuming x5 is initialized to 1, negating x5 results in -1 which is stored in x6. As this instruction translates to instruction `SUB`, the negation is computed as, x6 = 0-x5.

Exception

Overflow can only occur when the most negative value is negated. Overflow is ignored.

2.3.1.6 NEGW

`Negate Word (NEGW)` instruction computes the two's complement of a 32-bit value.

Syntax

`negw rd,` rs_1

Translation

`subw` rd, `x0`, rs_1

where,

rs_1	source register 1
rd	destination register

Description

Similar to instruction `NEG`, the `NEGW` is used to negate a 32-bit number stored in rs_1 with the result being stored in register rd. `NEGW` translates to `SUBW` where the 32-bit number in rs_1 is subtracted from zero.

Usage

```
negw x6, x5        # x6 ⟵ x5
```

Assuming register x5 is initialized to the value 168496141, negating x5 results in -168496141 which is stored in x6. As this instruction translates to `SUBW`, the negation is computed as, x6 = 0-x5.

2.3.1.7 SEQZ

`Set If Equal to Zero (SEQZ)` instruction provides an indication if a register's content is zero.

Syntax

`seqz` rd, rs_1

Translation

`sltiu` rd, rs_1, `1`

where,

rs_1	source register 1
rd	destination register

Description

RISC-V provides a simple pseudo-assembler instruction, `SEQZ`, to check if the contents of the register rs_1, is zero or not. Indication is provided by a single bit value 0 if the register content is not 0 or value 1, if the register content is zero. `SEQZ` performs an unsigned comparison against 1. Since the comparison is unsigned, the only value less than 1 is 0. Hence if the comparison holds true, register rs_1 must contain 0.

Usage

```
seqz x6, x5        # x6 ⟵ (x5 = 0) ?  1:0
                   # x6 = 1
```

Assuming register x5 contains 0, `SEQZ` instruction writes value 1 into register x6.

2.3.1.8 SNEZ

`Set If Not Equal to Zero (SNEZ)` instruction provides an indication if a register contains non-zero value.

Syntax

snez rd, rs_1

Translation

sltu rd, x0, rs_1

where,

rs_1	source register 1
rd	destination register

Description

`SNEZ` is a pseudo-assembler instruction that is used to check if the contents of a rs_1, is a non-zero value. This instruction sets value of register rd to 1 if the rs_1 is a non-zero value or sets rd to 0 otherwise. This instruction is implemented with an unsigned comparison against 0 using its base instruction `SLTU`. Since it is an unsigned comparison, the only value less than 0 is 0 itself. Therefore, if the less-than condition holds, the value in rs_1 must not be 0.

Usage

```
snez x6, x5        # x6 ⟵ (x5 ≠ 0) ?  1:0
                   # x5 = 9
                   # x6 = x0<x5 = 0<9 = 1
                   # x6 = 1
```

Assuming rs_1 (x5) is initialized to value 5, since this is greater than 0 value 1 is written into rd (x6).

2.3.1.9 SLTZ

`Set If Less Than Zero (SLTZ)` is a signed instruction which examines if a register's content is less then zero and indicates accordingly.

Syntax

sltz rd, rs_1

Translation

slt rd, rs_1, x0

where,

rs_1	source register 1
rd	destination register

Description

SLTZ is a signed pseudo-assembler instruction which translates to SLT, examines if the value in register rs_1 is less than zero. If register value found to be less than zero, a value 1 is stored in register rd. Otherwise the value 0 is stored.

Usage

```
sltz x6, x5        # x6 ⟵ (x5 < 0) ?  1:0
                   # x5 = -2
                   # x6 = x5<0 = -2<0 = 1
                   # x6 = 1
```

Assuming rs_1 (x5) is initialized with the value -2. Since the value -2 is less than 0, rd (x6) is entered with a value 1.

2.3.1.10 SGTZ

Set If Greater Than Zero (SGTZ) instruction examines if a register contains a value is greater than zero and indicates it accordingly.

Syntax

sgtz rd, rs_1

Syntax

slt rd, x0, rs_1

where,

rs_1	source register 1
rd	destination register

Description

SGTZ is a signed pseudo-assembler instruction which examines if the value in register rs_1 is greater than zero. If found true, value 1 is stored to register (rd) or value 0 is stored otherwise.

Usage

```
sgtz x6, x5        # x6 ⟵ (x5 > 0) ?  1:0
                   # x5 = 9
                   # x6 = x0<x5 = 0<9 = 1
                   # x6 = 1
```

Assume rs_1 (x5) is initialized to 9, since this is greater than 0. Value 1 will be stored in rd (x6).

chapter

3 Bitwise Instructions

3.1 RV 32I

RV 32I deals with the 32 bit instruction that are used for bit manipulation. The instructions are broadly classified as register-register and immediate instructions

3.1.1 Register to Register Instructions

Register operations involve both the operands as registers. The operation is performed on the value in the register and result is stored in destination register (rd). The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. 32 bits of result is written to the destination register.

3.1.1.1 SLL

`Shift Logical Left (SLL)` performs logical left on the value in register (rs_1) by the shift amount held in the register (rs_2) and stores in (rd) register.

Syntax

`sll rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

A `SLL` of one position moves each bit to the left by one. The low-order bit (the right-most bit) is replaced by a zero bit and the high-order bit (the left-most bit) is discarded.

Usage

li x5, 4	# $x5 \longleftarrow 2$
li x3, 2	# $x3 \longleftarrow 2$
sll x1, x5, x3	# $x1 \longleftarrow x5 << x3$

x1 will have a value 16.

3.1.1.2 SRL

`Shift Logically Right (SRL)` performs logical Right on the value in register (rs_1) by the shift amount held in the register (rs_2) and stores in (rd) register.

Syntax

`srl rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register2

Description

A `SRL` of one position moves each bit to the Right by one. The high-order bit (the left-most bit) is replaced by a zero bit and the low-order bit (the Right-most bit) is discarded.

Usage

li x5, 4	# $x5 \longleftarrow 4$
li x3, 2	# $x3 \longleftarrow 2$
srl x1, x5, x3	# $x1 \longleftarrow x5 >> x3$

x1 will have a value 1.

3.1.1.3 SRA

`Shift Right Arithmetic (SRA)` performs right shift on the value in register (rs_1) by the shift amount held in the register (rs_2) and stores in (rd) register.

Syntax

`sra rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`SRA` directive performs an arithmetic shift right by 0 to 32 places. The vacated bits at the most significant end are filled with zeros if the original value (the source operand) was positive. The vacated bits are filled with ones if the original value was negative. This is known as “sign extending” because the most significant bit of the original value is the sign bit for 2’s complement numbers, i.e. 0 for positive and 1 for negative numbers. Arithmetic shifting therefore preserves the sign of numbers.

Usage

```
li x5, 4            # x5 ⟵ 4
li x3, 2            # x3 ⟵ 2
sra x1, x5, x3      # x1 ⟵ x5 >> x3
```

x1 will have a value 1.

3.1.1.4 OR

`OR` directive performs bit-wise logical OR operation between contents of register (rs_1) and contents of register (rs_2) and stores in (rd) register.

Syntax

`or rd,` rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

A `bit-wise OR` is a binary operation that takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits.

Usage

```
li x5, 0x0100       # x5 ⟵ 0x0100
li x3, 0x0010       # x3 ⟵ 0x0010
or x1, x5, x3       # x1 ⟵ x5|x3
```

x1 will have a value 0x0110.

3.1.1.5 XOR

`XOR` performs bit-wise binary Exclusive-OR operation on the source register operands.

Syntax

`xor rd,` rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

A bit-wise `XOR` is a binary operation that takes two bit patterns of equal length and performs the logical inclusive XOR operation on each pair of bits.

Usage

```
li x5, 0x0100        # x5 ⟵ 0x0100
li x3, 0x0010        # x3 ⟵ 0x0010
xor x1, x5, x3       # x1 ⟵ x5|x3    (x1 ⟵ 0x0110)
```

3.1.1.6 NOT

`NOT` is a bit-wise invert operation, which performs a one's complement arithmetic.

Syntax

`not rd,` rs_1

Translation

`xori rd,` rs_1`, -1` # [-1 = `0xFFFFFFFF`]

where,

rs_1	source register 1
rd	destination register

Description

`NOT` instruction flips each bit of a register. This instruction translates to an exclusive OR operation `XORI` and implements the negation. The result is loaded into the destination register (rd).

Usage

```
not x6, x5        # x6 ⟵   ∼ x5
```

Assuming register x5 (rs_1) is initialized to value 1, on applying the `NOT` instruction on x5, 1 will be xored (since `XORI` is the base instruction for `XORI`) with -1, resulting to -2 (stored in x6). Now let's assume x5 is initialized to value -1, on applying `NOT` to it results in a value 0.

3.1.1.7 SLT

`Set Less Than (SLT)` perform the signed and unsigned comparison between (rs_1) and (rs_2) and stores the result in (rd).

Syntax

`slt rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`SLT` perform signed and unsigned compares respectively, writing 1 to rd if $rs_1 < rs_2$, 0 otherwise.

Usage

```
li x5, 3            # x5 ⟵ 3
li x3, 5            # x3 ⟵ 5
slt x1, x5, x3      # x1 ⟵ x5 < x3
```

x1 will have a value 1.

3.1.1.8 SLTU

`Set Less Than Unsigned (SLTU)` perform the signed and unsigned comparison between (rs_1) and (rs_2) and stores the result in (rd).

Syntax

`sltu rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`SLTU` sets rd to 1 if rs_2 is not equal to zero, otherwise sets rd to zero .`SLTU` perform signed and unsigned compares respectively, writing 1 to rd if rs_1¡rs_2, 0 otherwise.

Usage *x1* will have a value 1.

```
li x5, 3            # x5 ⟵ 3
li x3, 5            # x3 ⟵ 5
slt x1, x5, x3      # x1 ⟵ x5 < x3
```

3.1.2 Immediate instructions

Any instruction which contains an operand that is directly encoded as part of an instruction is called an immediate instruction and the operand as immediate operand. This section covers shift and logical operations with immediate operands as part of the instruction.

3.1.2.1 SLLI

`Shift Logically Left Immediate (SLLI)` performs logical left on the value in register (rs_1) by the shift amount held in the register (imm) and stores in (rd) register.

Syntax

`slli rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	immediate data

Description

A `SLLI` of one position moves each bit to the left by one. The low-order bit (the right-most bit) is replaced by a zero bit and the high-order bit (the left-most bit) is discarded.

Usage

```
slli x1, x1, 1    # x1 ⟵ x1<<1
```

3.1.2.2 SRLI

`Shift Logically Right Immediate (SRLI)` performs logical Right on the value in register (rs_1) by the shift amount held in the register (imm) and stores in (rd) register.

Syntax

`srli rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	immediate data

Description

A `Shift Right Logical Immediate (SRLI)` of one position moves each bit to the Right by one. The most significant bit is replaced by a zero bit and the least significant bit is discarded.

Usage

```
srli x1, x1, 1    # x1 ⟵ x1>>1
```

3.1.2.3 SRAI

`Shift Right Arithmetic Immediate (SRAI)` performs right shift on the value in register (rs_1) by the shift amount held in the (imm) and stores in (rd) register.

Syntax

`srai rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	Immediate data

Description

`SRAI` is arithmetic shift right of a number by 'N' places. The vacated bits at the most significant end are filled with value of sign bit (0 for +ve sign and 1 for -ve sign). This is known as "sign extending".The most significant bit of the original value is the sign bit for 2's complement numbers.

Usage

```
srai x1, x1, 1        # x1 ⟵ x1>>1
```

3.1.2.4 ANDI

`AND Immediate (ANDI)` performs binary operation between contents of register (rs_1) and immediate data (imm) and stores in (rd) register.

Syntax

`andi rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	immediate data

Description

A `Bitwise ANDI` is a binary operation that takes two bit patterns of equal length and performs the logical inclusive AND Immediate operation over each bits. The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. 32 bits of result is written to the destination register.

Usage

andi x5, x5, 4   # $x5 \longleftarrow x5\&4$

3.1.2.5 ORI

`OR Immediate (ORI)` performs binary operation between register (rs_1) and Immediate data (imm) and stores in (rd) register.

Syntax

`ori rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	Immediate data

Description

A `bitwise ORI` is a binary operation that takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits.

Usage

li x5, 0x0100	# $x5 \longleftarrow 0x0100$
ori x1, x5, 0x0010	# $x1 \longleftarrow x5\|2$

x1 will have a value 0x0110.

3.1.2.6 XORI

`Exclusive-OR Immediate (XORI)` performs bit-wise binary operation between register contents (rs_1) and Immediate data (imm) and stores in (rd) register.

Syntax

`xori rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	Immediate data

Description

A `bitwise XORI` is a binary operation that takes two bit patterns of equal length and performs logical inclusive XOR operation on each pair of corresponding bits.

Usage

xori x5, x5, 0b100000	# $x5 \longleftarrow x5\|0x0b100000$

3.1.2.7 SLTI

`Set Less than Immediate (SLTI)` compares contents of register (rs_1) and Immediate data (imm) and sets value in (rd) register.

Syntax

`slti rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	Immediate data

Description

A `SLTI` is a signed comparison between contents of the specified registers. If the value in register is less than the immediate value, value 1 is stored in destination register, otherwise, value 0 is stored in the destination register.

Usage

slti x5, x1, 2 $\# \ x5 \longleftarrow x1 < 2$

3.1.2.8 SLTIU

`Set Less Than Immediate Unsigned (SLTIU)` does comparison between register contents (rs_1) and Immediate data (imm) and sets value in (rd) register.

Syntax

`sltiu rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	Immediate data

Description

A `SLTIU` is a comparison to the contents of register using unsigned comparison. If the value in register is less than the immediate value, the value 1 is stored in destination Register, otherwise, the value 0 is stored in destination register.

Usage

slti x5, x1, 2 $\# \ x5 \longleftarrow x1 < 2$

3.2 RV 64I

RV 64I deals with the 64 bit instruction that are used for bit manipulation arithmetic operations. The instructions are broadly classified as register-register and immediate instructions.

3.2.1 Register to Register Instructions

The RV64I register-register operations involve both the operands as 64 bit registers. The operation is performed on the value in the register and result is stored in a destination register (rd). The source and destination registers can be any of the 31 base registers. x0 is read only.

3.2.1.1 SLLW

`Shift Left Logical Word (SLLW)` performs logical left on the value in register (rs_1) by the shift amount held in the register (rs_2) and stores in (rd) register.

Syntax

`sllw rd,` rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

A `SLLW` of one position moves each bit to the left by one. The low-order bit (the right-most bit) is replaced by a zero bit and the high-order bit (the left-most bit) is discarded.

Usage

```
li x3,5              # x3 ⟵ 5
li x1,3              # x1 ⟵ 3
sllw x1, x1, x3      # x1 ⟵ x1<<x3
```

3.2.1.2 SRLW

`Shift Right Logically Word (SRLW)` performs logical right on the value in register (rs_1) by the shift amount held in the register (rs_2) and stores in (rd) register.

Syntax

`srlw rd,` rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

A SRLW of one position moves each bit to the Right by one. The High-order bit (the left-most bit) is replaced by a zero bit and the low-order bit (the Right-most bit) is discarded.

Usage

```
li x1, 3            # x1 ⟵ 3
li x3, 5            # x1 ⟵ 5
srlw x1, x1, x3     # x1 ⟵ x1>>x3
```

3.2.1.3 SRAW

`Shift Right Arithmetic Word (SRAW)` performs Arithmetic right on the value in register (rs_1) by the shift amount held in the register (rs_2) and stores in (rd) register.

Syntax

`sraw rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

SRAW is an arithmetic shift right of a word by 'N' places. The vacated bits at the most significant end are filled with value of sign bit (0 for +ve sign and 1 for -ve sign). This is known as "sign extending". The most significant bit of the original value is the sign bit for 2's complement numbers. **Usage**

```
li x1, 3            # x1 ⟵ 3
li x3, 5            # x1 ⟵ 5
sraw x1, x1, x3     # x1 ⟵ x1>>x3
```

3.2.2 Immediate instructions

A 64-bit system involves 64-bit constant operands as part of their instructions.

3.2.2.1 SRLIW

`Shift Right Logical Immediate Word (SRLIW)` performs Logical right on the value in register (rs_1) by the shift amount held in the immediate data (imm) and stores in (rd) register.

Syntax

`srliw rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	immediate data

Description

A `SRLIW` does one position move of each bit to the left by one. The low-order bit (the right-most bit) is replaced by a zero bit and the high-order bit (the left-most bit) is discarded.

Usage

```
li x3,5                # x3 ⟵ 5
li x1,3                # x1 ⟵ 3
srliw x1, x1, x3       # x1 ⟵ x1>>x3
```

3.2.2.2 SRAIW

`Shift Right Arithmetic Immediate Word (SRAIW)` performs Arithmetic right on the value in register (rs_1) by the shift amount held in the Immediate (imm) and is stored in (rd) register.

Syntax

`sraiw rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	immediate data

Description

`SRAIW` is an arithmetic shift right immediate by 0 to 64 places. The vacated bits at the most significant end are filled with zeros if the original value (the source operand) was positive. The vacated bits are filled with ones if the original value was negative. This is known as ”sign extending” because the most significant bit of the original value is the sign bit for 2’s complement numbers, i.e. 0 for positive and 1 for negative numbers. Arithmetic shifting therefore preserves the sign of numbers.

Usage

```
li x1, 3              # x1 ⟵ 3
sraiw x1, x1, x3      # x1 ⟵ x1>>x3
```

chapter 4

Arithmetic Instructions

4.1 RV 32I

RV 32I deals with the 32 bit instruction that are used for arithmetic operations. The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. The instructions are broadly classified as register-register and immediate instructions

4.1.1 Register to Register instructions

Register to register instruction involves, both the operands as a register. The contents of the register holds the content of the operands.

4.1.1.1 ADD

`Addition (ADD)` adds the contents of two registers and stores the result in another register.

Syntax

`add rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

The `ADD` instruction adds content of the two registers rs_1 and rs_2 and stores the resulting value in rd register. The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. Overflows are ignored and the lower 32 bits of result is written to the destination register.

Usage

```
li x2, 3          # x2 ⟵ 3
li x3, 4          # x3 ⟵ 4
add x1, x2, x3    # x1 ⟵ x2 + x3
```

Assuming rs_1 (x2) and rs_2 (x3) contain values 3 and 4 respectively, an addition operation on them will result in value 7 which will be stored in rd (x1). *x1* will have a value 7.

4.1.1.2 SUB

`Subtraction (SUB)` subtracts contents of one register from another and stores the result in another register.

Syntax

`sub rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

The `SUB` instruction subtracts content of the source register rs_2 from rs_1 and stores the value in the register rd. Overflows are ignored and the lower XLEN bits of the result is written to rd. The source and destination registers can be any of the 31 base registers. The x0 register can be used as a source register only, but not as a destination register. The overflows as well as borrow are ignored and the lower 32 bits of result is written to the destination register.

Usage

```
li x2, 4          # x2 ⟵ 4
li x3, 3          # x3 ⟵ 3
sub x1, x2, x3    # x1 ⟵ x2 - x3
```

x1 will have a value 1.

4.1.1.3 MUL

`Multiplication (MUL)` calculates the product of the multiplier in source register 1 (rs_1) and multiplicand in source register 2 (rs_2), with the resulting product being stored in destination register (rd).

Syntax

`mul rd,` rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`MUL` calculates the product of two XLEN-bit operands in the source registers 1 and 2 (rs_1, rs_2). This instruction stores the less significant part of the result in the destination register and any overflow is ignored.

Usage

```
mul x4, x9, x13        # x4 ⟵ Low Bits [x9 * x13]
```

4.1.1.4 MULH

`Multiply signed and return upper bits (MULH))` calculates the product of signed values in source registers (rs_1) and (rs_2) and stores result in the specified destination register (rd).

Syntax

`mulh rd,` rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`MULH` calculates the product of signed multiplier and signed multiplicand (present in the two source registers specified respectively), and places the upper XLEN bits of the full 2*XLEN product, into the destination register. MULH has to be used with MUL to get the complete 2*XLEN bits result.

Usage

```
li x1,-80              # x1 ⟵ -80
li x5,20               # x5 ⟵ 20
mulh x5, x5, x1        # x5 ⟵ High Bits[x5*x1]
```

4.1.1.5 MULHU

`Multiply Unsigned and return upper bits (MULHU))` calculates the product of two unsigned values in source registers rs_1 and rs_2. The resulting value is placed in the specified destination register (rd).

Syntax

`mulhu rd,` rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`MULHU` multiplies two unsigned operands in the source registers and the most significant part of result is stored in the destination register.

Usage

```
li x1,-80              # x1 ⟵ -80
li x5,20               # x5 ⟵ 20
mulhu x5, x5, x1       # x5 ⟵ High Bits [x5*x1]
```

4.1.1.6 MULHSU

`Multiply Signed-Unsigned and return upper bits (MULHSU))` calculates the product of a signed value in source register rs_1 with an unsigned value in source register rs_2 and the resulting product is stored in destination register, rd.

Syntax

`mulhsu rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`MULHSU` computes the product of the signed, most significant word of the multiplier and the unsigned, least significant word of the multiplicand. The most significant part of the resulting product is stored in the specified destination register. The resulting value is a signed value.

Usage

```
li x1,-80              # x1 ⟵ -80
li x5,20               # x5 ⟵ 20
mulhsu x5, x5, x1      # x5 ⟵ High Bits[x5*x1]
```

4.1.1.7 DIV

`Division (DIV)` performs division on the value in source register (rs_1) with the value in the source register (rs_2) and stores quotient in (rd) register.

Syntax

`div rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

DIV does the division of operands in source registers and stores quotient in the destination register. Both operands and the result are signed values.

Usage

```
li x9, -400          # x9 ⟵ -400
li x13, 200          # x13 ⟵ 200
div x4, x9, x13      # x4 ⟵ x9/x13
```

4.1.1.8 DIVU

Division Unsigned (DIVU) performs unsigned Division on the value in source register (rs_1) by the value in the source register (rs_2) and stores quotient in the destination register (rd).

Syntax

divu rd, rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

DIVU does the division of unsigned operands in source registers and stores quotient in the destination register. Both operands and the result are unsigned values.

Usage

```
li x9, 400            # x9 ⟵ 400
li x13,200            # x13 ⟵ 200
divu x4, x9, x13      # x4 ⟵ x9/x13
```

4.1.1.9 REM

Reminder (REM) performs division on the value in source register (rs_1) with the value in the source register (rs_2) and stores remainder in (rd) register.

Syntax

rem rd, rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

REM does the signed division of operands in source registers and stores the remainder in the destination register. Both operands and the result are signed values.

Usage

```
li x9, 400          # x9 ⟵ 400
li x13,200          # x13 ⟵ 200
rem x4, x9, x13     # x4 ⟵ x9%x13
```

NOTE:

Sometime's a programmer needs both quotient and remainder. In such cases it is recommended to perform `DIV` first and `REM` later.

4.1.2 Immediate Instructions

Instructions involving a constant operand are immediate instructions. Here we are going to load and store immediate instructions.

4.1.2.1 LI

`Load Immediate (LI)` load register rd with a value that is immediately available

Syntax

`li rd, imm`

where,

rd	destination register
imm	Immediate data

Description

The `LI` instruction loads a positive or negative value that is immediately available, without going into memory. The value maybe a 16-bit or a 32-bit integer.

Usage

```
li x5, 24        # x5 ⟵ 24
```

4.1.2.2 ADDI

`Add Immediate (ADDI)` adds content of the source registers rs_1, immediate data (imm) and store the result in the destination register (rd).

Syntax

`addi rd,` rs_1`, imm`

where,

rd	destination register
rs_1	source register 1
imm	Immediate data

Description

The `ADDI` instruction adds content of a source register with an absolute value and stores the result in the destination register. Overflows are ignored and the lower 32 bits of result is written to the destination register.

Usage

```
li x2,24              # x2 ⟵ 24
addi x1, x2,64        # x1 ⟵ x2 + 64
```

x1 will have a value 88.

4.2 RV 64I

RV 64I deals with the 64 bit integer instructions that are used for arithmetic operations. The instructions are broadly classified as register-register and immediate instructions.

4.2.1 Register to Register instructions

The register operations involve both the operands as registers. The operation is performed on the value in the register and result is stored in destination register (rd).

4.2.1.1 ADDW

`Add Word (ADDW)` adds content of the source registers (rs_1, rs_2) and stores the result in the destination register (rd).

Syntax

`addw rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

The `ADDW` instruction adds content of the two source registers and stores the value in the destination register. The overflows are ignored and the lower 64 bits of result is stored in destination register.

Usage

```
addw x4, x9, x13      # x4 ⟵ x9 + x13
```

4.2.1.2 SUBW

`Subtract Word (SUBW)` subtracts content of the source registers (rs_1, rs_2) and store the result in the destination register (rd).

Syntax

`subw rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

The `SUBW` instruction subtracts content of the source register rs_2 from rs_1 and stores the value in the destination register (rd). The overflows as well as borrow are ignored and the lower 64 bits of result is written to the destination register.

Usage

```
li x2, 456          # x2 ⟵ 456
li x3, 123          # x3 ⟵ 123
subw x1, x2, x3     # x1 ⟵ x2 - x3
```

x1 will have a value 333.

4.2.1.3 REMU

`Reminder Unsigned (REMU)` performs division on the value in source register (rs_1) with the value in the source register (rs_2) and stores remainder in (rd) register.

Syntax

`remu rd,` rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`REMU` does the division of operands in source registers and stores remainder in the destination register. Both operands and the result are unsigned values.

Usage

```
li x9, 400          # x4 ⟵ 400
li x13,200          # x4 ⟵ 200
remu x4, x9, x13    # x4 ⟵ x9%x13
```

Note:
Sometime's a programmer needs both quotient and remainder. In such cases it is recommended to perform `DIV` first and `REM` later.

4.2.1.4 MULW

`Multiplication Word (MULW)` directive multiplies contents of register rs_1 with that of register rs_2 and stores result in register rd. Only the lower order 32-bits of the result are used, which is sign extended to the full length of the register.

Syntax

`mulw rd,` rs_1, rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`MULW` does the multiplication of operands in source registers and stores result in the destination register. Only the lower order 32-bits of the result are used the lower 32 bits are signed extended to the full length of the register. This instruction is used to properly emulate 32-bit multiplication on a 64-bit or 128-bit machine. Only the least-significant 32 bits of Reg1 and Reg2 can possibly affect the result. If you want the upper 32-bits of the full 64-bit result use the MUL instruction on a 64-bit machine.

Usage

```
mulw x4, x9, x13        # x4 ⟵ x9*x13
```

4.2.1.5 DIVW

`Divide Word (DIVW)` performs Division on the value in source register (rs_1) with the value in the source register (rs_2) and stores quotient in (rd) register.

Syntax

`divw rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`DIVW` does the division of operands in source registers and stores quotient in the destination register. Both operands and the result are signed values, only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended to fill the destination register.

Usage

```
li x9, 400              # x9 ⟵ 400
li x13,200              # x13 ⟵ 200
divw x4, x9, x13        # x4 ⟵ x9/x13
```

4.2.1.6 DIVUW

`Divide Unsigned Word (DIVUW)` performs division on the value in source register (rs_1) with the value in the source register (rs_2) and stores quotient in (rd) register.

Syntax

`divuw rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`DIVUW` does the division of operands in source registers and stores quotient in the destination register. Both operands and the result are unsigned values, only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended to fill the destination register.

Usage

```
li x9, 400             # x9 ⟵ 400
li x13,200             # x13 ⟵ 200
divuw x4, x9, x13      # x4 ⟵ x9/x13
```

4.2.1.7 REMW

`Reminder Word (REMW)` performs Division on the value in source register (rs_1) with the value in the source register (rs_2) and stores remainder in (rd) register.

Syntax

`remw rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`REMW` does the division of operands in source registers and stores remainder in the destination register. Both operands and the result are signed values. Only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended to fill the destination register.

Usage

```
li x9, 400            # x9 ⟵ 400
li x13,200            # x13 ⟵ 200
remw x4, x9, x13      # x4 ⟵ x9%x13
```

NOTE:

Sometime, a programmer might need both quotient and remainder. In such cases it is recommended to perform `DIV` first and `REM` later.

4.2.1.8 REMUW

`Reminder Unsigned Word (REMUW)` performs Division on the value in source register (rs_1) with the value in the source register (rs_2) and stores remainder in (rd) register.

Syntax

`remuw rd,` rs_1`,` rs_2

where,

rd	destination register
rs_1	source register 1
rs_2	source register 2

Description

`REMUW` does the division of operands in source registers and stores remainder in the destination register. Both operands and the result are unsigned values. The least significant 32 bits of the operands are used and the 32-bit result is signed-extended.

Usage

```
li x9, 400              # x9 ⟵ 400
li x13,200              # x13 ⟵ 200
remuw x4, x9, x13       # x4 ⟵ x9%x13
```

NOTE:
Sometime, a programmer might need both quotient and remainder. In such cases it is recommended to perform `DIV` first and `REM` later.

4.2.2 Immediate Word Instructions

Instructions which involve a 32-bit constant operand have the "W" to specify 32-bit operations to be performed on them.

4.2.2.1 ADDIW

`Add Immediate Word (ADDIW)` adds content of the source registers rs_1, imm and store the result in the destination register (rd).

Syntax

`addiw rd,` rs_1, imm

where,

rd	destination register
rs_1	source register 1
imm	Immediate data

Description

The `ADDIW` instruction adds content of the two source registers and stores the value in the destination register. This instruction is only present in 64-bit and 128-bit machines. The operation is performed using 32-bit arithmetic. The result is then truncated to 32-bits, signed-extended to 64 or 128-bits and placed in destination register. The overflows are ignored and the lower 64 bits of result is written to the destination register.

Usage

```
li x9,456               # x9 ⟵ 456
addiw x4, x9,123        # x4 ⟵ x9 + 123
```

5

chapter

Control Transfer Instructions

5.1 Branch Instructions

A branch instruction in a program causes the system to execute a different instruction sequence, making the system deviate from its normal course of action of executing instructions in sequence. Branches are useful for implementing logical constructs since the architecture allows compares and dependent branches to be scheduled in the same cycle.

5.1.0.1 BEQ

`Branch If Equal (BEQ)` the contents of source register rs_1 is compared with source register rs_2, if found equal, the control is transferred to the specified label.

Syntax

`beq` rs_1`,` rs_2`, label`

where,

rs_1	source register 1
rs_2	source register 2
label	

Description

The `BEQ` instruction compares contents of (rs_1) is compared to the contents of (rs_2). If equal, control jumps. The target address is given as a PC-relative offset. More precisely, the offset is sign-extended, multiplied by 2, and added to the value of the PC. The value of the PC used is the address of the instruction following the branch, not the branch itself. The offset is multiplied by 2, since all instructions must be half word aligned.

Usage

```
loop:  addi x5, x1, 1        # x5 ⟵ x1 + 1
beq x0, x0, loop             # x0 = x0 jump to loop
```

5.1.0.2 BNE

`Branch If Not Equal (BNE)` the contents of source register rs_1, is compared with source register rs_2 if they are not equal control is transferred to the label as mentioned.

Syntax

`bne` rs_1`,` rs_2`, label`

where,

rs_1	source register 1
rs_2	source register 2
label	

Description

The `BNE` instruction compares contents of (rs_1) is compared to the contents of (rs_2). If not equal, control jumps. The target address is given as a PC-relative offset.

Usage

```
label:  addi x4, x9,123       # x4 ⟵ x9 + 123
bne x4, x9, label             # x4 ≠ x9 jump to label
```

5.1.0.3 BLT

`Branch If Less Than (BLT)` the contents of source register rs_1, is compared with contents of source register rs_2. If (rs_1) is less than (rs_2) control is transferred to the label as mentioned.

Syntax

`blt` rs_1`,` rs_2`, label`

where,

rs_1	source register 1
rs_2	source register 2
label	

Description

The `BLT` instruction compares contents of (rs_1) is compared to the contents of (rs_2). If (rs_1) contents is less than (rs_2)(signed comparison), control jumps. The target address is given as a PC-relative offset.

Usage

```
label:  addi x4, x9, 123        # x4 ⟵ x9 + 123
blt x4, x9, label               # x4 < x9 jump to label
```

5.1.0.4 BLTU

`Branch If Less Than Unsigned (BLTU)` the contents of source register rs_1, is compared with contents of source register rs_2 if (rs_1) is less than (rs_2) control is transferred to the label as mentioned.

Syntax

`bltu` rs_1`,` rs_2`, label`

where,

rs_1	source register 1
rs_2	source register 2
label	

Description

The `BLTU` instruction compares contents of (rs_1) is compared with the contents of (rs_2). If (rs_1) contents is less than (rs_2), (unsigned comparison) control jumps. The target address is given as a PC-relative offset.

Usage

```
loop:  addi x1, x0, 1        # x1 ⟵ x0 + 1
addi x5, x0, 3               # x5 ⟵ x0 + 3
bltu x1, x5, loop            # x1 < x5 jump to loop
```

5.1.0.5 BGE

`Branch If Greater Than or Equal, signed (BGE)` the contents of source register rs_1, is compared with contents of source register rs_2 if (rs_1) is greater than (rs_2) control is transferred to the label as mentioned.

Syntax

`bge` rs_1`,` rs_2`, label`

where,

rs_1	source register 1
rs_2	source register 2
label	reference to a valid memory location

Description

The `BGE` instruction compares contents of (rs_1) with the contents of (rs_2). If (rs_1) contents is greater than or equal to contents of (rs_2), (signed comparison) control jumps to the specified location. The target address is given as a PC-relative offset.

Usage

```
label:  addi x4, x9, 123        # x4 ⟵ x9 + 123
bge x4, x9, label               # if x4 ≥ x9 jump to label
```

5.1.0.6 BGEU

`Branch If Greater Than or Equal, Unsigned (BGEU)` the contents of source register rs_1, is compared with contents of source register rs_2. If rs_1 is greater than or equal to rs_2, control is transferred to the label as mentioned.

Syntax

`bgeu` rs_1`,` rs_2`, label`

where,

rs_1	source register 1
rs_2	source register 2
label	

Description

The `BGEU` instruction compares contents of (rs_1) is compared with the contents of (rs_2). If (rs_1) contents is greater than (rs_2), (unsigned comparison) control jumps. The target address is given as a PC-relative offset.

Usage

```
label:  addi x4, x9,123         # x4 ⟵ x9 + 123
bgeu x4, x9, label              # x4 ≥ x9 jump to label
```

5.1.1 Pseudo Instructions

Branching instructions in this section are pseudo or convenient instructions to be used in place of the base instructions.

5.1.1.1 BEQZ

`Branch if Equal to Zero (BEQZ)` instruction jumps to a specified location in the program if the condition, equal to zero is met.

Syntax

`beqz` rs_1`, label`

Translation

`beq` rs_1`, x0, label`

where,

rs_1	source register
label	Address to JUMP to

Description

The `BEQZ` translates to `beq` rs_1`, x0, label`, as the expansion reveals, the (rs_1) contents is compared with the zero register ($x0$) and the program counter branches to the specified label if the condition equal to zero is met.

Usage

```
li x6, 0                          # x6 = 0
loop:  li x5, x5, 100             # Example operation
beqz x6, loop                     # x6 = 0 branch to loop
```

Assume rs_1 (x6) is initialized to 0 and there is an example operation within the specified label (loop). `BEQZ` on register rs_1 (x6) will shift the program counter to the specified label since the contents of rs_1 (x6) is indeed 0.

5.1.1.2 BNEZ

`Branch if Not Equal to Zero (BNEZ)` jumps to a specified location in the program if the condition, not equal to zero is met.

Syntax

`bnez` rs_1`, label`

Translation

`bne` rs_1`, x0, label`

where,

rs_1	source register 1
label	Address to JUMP to

Description

The `BNEZ` instruction translates to `BNE`. As the translation reveals, the contents of rs_1 is compared with the zero register ($x0$) and branches to the specified label, if the condition that the contents of rs_1 register is not equal to zero, is met.

Usage

```
li x6, 50                           # x6 = 50
loop:  addi x5, x6, 100             # Example operation
bnez x6, loop                       # x6 ≠ 0 jump to loop
```

Assume rs_1 (x6) is initialized to 50 and there is an example operation within the specified label (loop). `BNEZ` on register rs_1 (x6) will shift the program counter to the specified label since the contents of rs_1 (x6) is indeed not equal to 0.

5.1.1.3 BLEZ

`Branch if Less Than or Equal to Zero (BLEZ)` the program counter branches to the specified location if the condition, less than or equal to zero.

Syntax

blez rs_1, label

Translation

bge x0, rs_1, label

where,

rs_1	source register 1
label	Address to JUMP to

Description

The BLEZ expands to BGE. This instruction is a signed comparison instruction which shifts the program counter to the specified location if value in rs_1 is less than or equal to 0.

Usage

```
li x6, -50                  # x6 = −50
loop:  addi x5, x6, 100     # Example operation
blez x6, loop               # x6 ≤ 0 jump to loop
```

Assuming rs_1 (x6) is initialized to -50, BLEZ, shifts the program counter to label (loop) since the condition that rs_1 (x6) should to either less than or equal to 0, is met.

5.1.1.4 BGEZ

Branch if greater than or equal to Zero (BGEZ) checks if register rs_1 is greater than or equal to zero, if the condition is met, the program counter branches to the specified label.

Syntax

bgez rs_1, label

Translation

bge rs_1, x0, label

where,

rs_1	source register 1
label	Address to JUMP to

Description

The BGEZ expands to BGE. This instruction compares if contents of rs_1 is greater than or equal to zero ($x0$). If the conditions are met, the program counter branches to the specified label.

Usage

```
li x6, 50                   # x6 = 50
loop:  addi x5, x6, 100     # Example operation
bgez x6, loop               # x6 ≥ 0 jump to loop
```

Assuming that rs_1 (x6) is initialized to a value 50, BGEZ instruction shifts the program counter to label (loop) since the condition, rs_1 (x6) must be greater than or equal to 0, is satisfied.

5.1.1.5 BLTZ

`Branch if Less Than Zero (BLTZ)` shifts the program counter to a specified location if the value in a register is less than zero.

Syntax

`bltz` rs_1`, label`

Translation

`blt` rs_1`, x0, label`

where,

rs_1	source register 1
label	Address to JUMP to

Description

`BLTZ` is a signed comparison instruction with its base instruction being `BLT`. The value in rs_1 is compared with x0 and shifts the program counter to the specified location in case its contents are less than 0.

Usage

```
li x6, -20                  # x6 = -20
loop:  addi x5, x6, 100     # Example instruction
bltz x6, loop               # x6 < 0 jump to loop
```

Assuming rs_1 (x6) is initialized to -20, `BLTZ` shifts the program counter to label (loop) since the contents of rs_1 (x6) is indeed less than 0. The program then executes the instructions within the label (loop).

5.1.1.6 BGTZ

`Branch if Greater Than Zero (BGTZ)` shifts the program counter to a specified location, if the contents of a register is found to be greater than zero.

Syntax

`bgtz` rs_1`, label`

Syntax

`blt x0,` rs_1`, label`

where,

rs_1	source register 1
label	Address to JUMP to

Description

The `BGTZ` is a signed comparison instruction which translates to its base instruction `BLT`. If the contents of rs_1 is greater than x0, the program counter shifts and continues its execution with the instructions in the location specified.

Usage

```
li x6, 5                        # x6 = 5
loop:  addi x5, x6, 100         # Example instruction
bgtz x6, loop                   # x6 > 0 jump to label
```

Assuming that rs_1 (x6) is initialized to value 5, the `BGTZ` instruction shifts the program counter to label (loop), since rs_1 (x6) is greater than 0. Program execution continues with what label (loop) contains.

5.1.1.7 BGT

`Branch if Greater Than (BGT)` instruction shifts the program counter to the specified location if the value in a register is greater than that of another.

Syntax

`bgt` rs_1`,` rs_2`, label`

Translation

`blt` rs_2`,` rs_1`, label`

where,

rs_1	source register 1
rs_2	source register 2
label	Address to JUMP to

Description

The `BGT` is a signed comparison instruction which translates to `BLT`. In this instruction, it is examined if the contents of rs_2 is less than the contents of register rs_1. If the condition is satisfied, program counter branches to the location specified.

Usage

```
li x5, 30                       # x5 = 30
li x6, -25                      # x6 = -25
loop:  addi x7, x6, 100         # Example instruction
bgt x5, x6, loop                # x6 < x5 jump to loop
```

Assuming rs_1 (x5) is initialized to 30 and rs_2 (x6) is initialized to -25. Since the condition rs_2 (x6) should be less than rs_1 (x5) to branch, is true (`BGT` translates to `BGT`), the program branches to label (loop) and continues execution

5.1.1.8 BLE

`Branch if Less Than or Equal (BLE)` instruction shifts the program counter to the specified location if the value in a register is less than or equal to that of another.

Syntax

`ble` rs_1`,` rs_2`, label`

Translation

`bge` rs_2, rs_1, `label`

where,

rs_1	source register 1
rs_2	source register 2
label	Address to JUMP to

Description

The `BLE` is a signed comparison instruction which examines if the contents of rs_1 is less than or equal to the contents of register rs_2. If the condition is satisfied, program counter branches to the location specified.

Usage

```
       li x5, -25           # x5 = -25
       li x6, 30            # x6 = 30
loop:  ble x5, x6, loop     # Example instruction
```

Assume rs_1 (x5) is initialized to -25 and rs_2 (x6) is initialized to 30, the program branches to the specified label (loop) since rs_1 (x5) is less than rs_2 (x6).

5.1.1.9 BGTU

`Branch if Greater Than, Unsigned (BGTU)` an unsigned comparison instruction to examine if contents of one register is greater than the other, according to which the program counter branches to the specified label.

Syntax

`bgtu` rs_1, rs_2, `label`

Translation

`bltu` rs_2, rs_1, `label`

where,

rs_1	source register 1
rs_2	source register 2
label	Address to JUMP to

Description

The `BGTU` is an unsigned comparison instruction which examines if the contents of rs_1 is greater than rs_2. If the condition is satisfied, the program counter shifts to the specified location and continues executing instructions from there on.

Usage

```
       li x6, 50             # x6 = 50
       li x7, 10             # x7 = 10
loop:  bgtu x6, x7, loop     # x6 > x7 Jump to loop
```

Assume rs_1 (x6) is initialized to 50 and rs_2 (x7) is initialized to 10. The program shifts to the specified label (loop) as rs_1 is greater than rs_2.

5.1.1.10 BLEU

`Branch if Less Than or Equal, Unsigned (BLEU)` instruction examines whether the of one register is less than or equal to the other and the program counter shifts accordingly.

Syntax

`bleu` rs_1`,` rs_2`, label`

Translation

`bgeu` rs_2`,` rs_1`, label`

where,

rs_1	source register 1
rs_2	source register 2
label	Address to JUMP to

Description

`BLEU` is an unsigned comparison instruction which examines if contents of rs_1 is less than or equal to that of rs_2. If the condition is satisfied, the program counter branches to the specified label.

Usage

```
li x6, 20                  # x6 = 20
li x7, 25                  # x7 = 25
loop:  addi x5, x7, 100    # Example instruction
bleu x6, x7, loop          # x6 ≤ x7 Jump to loop
```

Assuming rs_1 (x6) is initialized to 20 and rs_2 (x7) is initialized to 25. Since rs_1 (x6) is less than rs_2 (x7), the `BLEU` instruction branches the program counter to the specified label (loop).

5.1.1.11 RET

`Return from Subroutine (RET)` pseudo-instruction used at the end of a subroutine to return to its caller.

Syntax

`label:  ret`

where,

label	sub-routine

Description

The `RET` translates to `jalr x0, 0(ra)`. This instruction jumps to the address in the `ra`, but does not save a return address. The instruction will ensure that execution continues from where the call was made.

Usage

```
li x6, 50
li x7, 20
addi x5, x7, 100
ret                     # Return back to caller
```

5.2 Unconditional Jump Instructions

Unconditional Jump Instructions transfers the program sequence to the specified memory address without a condition.

5.2.0.1 Jump and Link

`Jump and Link (JAL)` is used to call a subroutine (i.e., function).

Syntax

`jal rd, offset`

where,

rd	destination register
$offset$	offset value

Description

The `JAL` instruction is used to call a subroutine (i.e., function). The return address (i.e., the PC, which is the address of the instruction following the JAL) is saved in the destination register. The target address is given as a PC-relative offset, more precisely, the offset is sign-extended, multiplied by 2, and added to the value of the PC. The value of the PC used is the address of the instruction following the JAL, not the JAL itself. The offset is multiplied by 2, since all instructions must be half word aligned.

Usage

```
loop:  addi x5, x4, 1        # x5 ⟵ x4 + 1
jal x1, loop                 # Goto loop x1 ⟵ address[loop]
```

5.2.0.2 JALR

`Jump and Link Register (JALR)` is used to invoke a subroutine call (i.e., function/method/procedure).

Syntax

`jalr rd, offset`

where,

rd	destination register
$offset$	offset value

Description

The `JALR` instruction is used to call a subroutine (i.e., function). The return address (i.e., the PC, which is the address of the instruction following the JALR) is saved in the destination register. The target address is given as a PC-relative offset, more precisely, the offset is sign-extended and added to the value of the destination register. The offset is not multiplied by 2.

Usage

```
addi x1, x0, 3              # x1 ⟵ x0 + 3
loop:  addi x5, x0, 1       # x5 ⟵ x0 + 1
jalr x0, 0(x1)              # x0 ⟵ mem[x1 + 0]
```

5.2.0.3 J

`Jump (J)` is a pseudo-instruction which uses `Jump and Link (JAL)` instead and sets the destination register to zero to discard return address.

Syntax

`j label`

where,

j	Jump
label	A string that points to an instruction

Description

`J` is a plain unconditional jump (UJ-type) instruction used to jump to anywhere in the code memory. This instruction translates to `jal x0, label`, which sets the return address to zero thus discarding the return address.

Usage

```
loop:  li x6, 100       # x6 ⟵ 100
li x7, 100              # x7 ⟵ 100
li x1, 1000             # x1 ⟵ 1000
add x5, x6, x7          # x5 ⟵ x6 + x7
bge x5, x1, load1       # x5 ≥ x1
load1:  li x5, x0       # x5 ⟵ 0
j loop                  # Jump to loop
```

5.2.0.4 JR

`Jump Register (JR)` is a pseudo-instruction which translates to `Jump and Link Register (JALR)` which jumps to the address and places the return address in a general purpose register (GPR).

Syntax

`jr` rs_1

where,

jr	Jump Register
rs_1	Return Address

Description

`JR` is translated to `jalr` rd, rs_1, `imm` where, rd is zero register, rs_1 contains the target address and imm is given the value 0. In this instruction, the rd field is set to zero thereby performing the jump to the address in ra register but does not save a return address.

Usage

```
label:  li x28, 100          # x1 <-- 100
li x5, 200                   # x5 <-- 200
li x6, 50                    # x6 <-- 50
jal ra, loop                 # ra <-- loop
li x2, 10                    # x2 <-- 10
loop:  add x4, x28, x5       # x4 <-- x28 + x5
sub x7, x6, x4               # x7 <-- x6 + x4
jr ra                        # JumpRegister
```

5.3 System Instructions

SYSTEM instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions. CSR instructions are described in this

5.3.1 ECALL

`Environment Call (ECALL)` instruction is used to implement system calls. Also, ECALL is used to transfer control from lower privilege level to higher privilege level.

Syntax

```
ecall
```

Description

The `ECALL` instruction is used to implement system calls. System calls are subroutine calls made from a lower privilege code to a higher privilege code. The execution happens in the higher privilege level and result is given back to the lower privilege code. Once the desired operation is over, the control returns back to the lower privilege level. Generally, if an operation needs to be done at a higher privilege level, ECALL is used. For example, the implementations of libraries for FILE operations in a Unix operating system, uses ECALL. On execution of ECALL, one of the following exception arise:

- Environment Call from User Mode
- Environment Call from Supervisor Mode
- Environment Call from Machine Mode

As described in the section "mcause", the above exceptions have a dedicated exception code. The trap handler in higher privilege level handles the exception and redirects the call to the corresponding subroutine. The arguments are passed through argument registers (a_i) and result is saved in Saved register (s_i).

Usage

```
addi x5, x0, 4          # x5 ⟵ 0 + 4
ecall                   # Atomic jump to location 0x80000180
```

5.3.2 EBREAK

`Environment Break (EBREAK)` is an assembly instruction that is used to stop the execution suddenly.

Syntax

`ebreak`

Description

The `EBREAK` instruction is used to invoke a debugger, by causing a "Breakpoint" exception. Typically the debugging software will insert this instruction at various places in the application code sequence, in order to gain control from an executing program.

Usage

```
la x1, msg                  # x1 ⟵ address[msg]
li x2, 0x11100111           # x2 ⟵ 0x11100111
ebreak                      # Debugger Breakpoint to test code
sw x5, 0(x1)                # ValueAt[x1 + 0] ⟵ x5
.section .rodata
msg:  .string "Hello World!"
```

5.3.3 WFI

`Wait For Interrupt (WFI)` instruction causes the processor to suspend instruction execution. The processor will wake up when an asynchronous interrupt occurs and resumes execution.

Syntax

`WFI`

Description

On execution of `WFI` trap handler will be invoked and upon return to the code sequence containing the `WFI` instruction, the next instruction following the `WFI` will be executed.

5.3.4 NOP

The `No Operation (NOP)` instruction executes silently. It does not change registers, memory or processor statues. Only the program counter is advanced.

Syntax

`nop`

Description

`NOP` is a pseudo instruction that expands to `addi x0, x0, 0`. The x0 is a read-only register holding the value zero. Anything, written to x0 register is discarded. The NOP instruction does not change any architecturally visible state, except for advancing the pc and increment any applicable performance counters. As RISC-V has no arithmetic flags (i.e., carry, overflow, zero, sign flags), any arithmetic operation whose destination register is x0 will endup as a no operation instruction regardless of the source registers.

Usage

Lets say *pc* is at 0x80000000. After execution of below instruction.

nop $\quad\quad \# \ pc \longleftarrow pc + 2$

pc becomes 0x80000002. The state of the machine is unchanged.

chapter 6

Trap's in RISC-V

Trap is a specific scenario caused by a exceptional condition or interrupt. In RISC-V, the term `trap` refers to, transfer of control to a trap handler caused either by an exception or an interrupt. Exception is an unusual condition occurring at run time of an instruction in the current RISC-V hart. An exception disrupts the normal flow of instruction execution. Exceptions are usually synchronous. Interrupts are another form of a trap, where the origin of interrupt is from Timer or peripherals. Interrupt is a scenario designed to service a specific external input. All the Traps can be handled or ignored. It is upto the software to decide. A "trap handler" is a subroutine that handles the trap in a software. The way of handling a trap is left to the software designer and varies from one type of trap to another.

6.1 Exceptions

Exceptions are usually synchronous and always tied to an assembly instruction. A exception can arise at any stage of execution of an instruction. For example, during instruction decode stage, the hardware may detect a bad opcode field. This will trigger a "illegal instruction" exception. When an exception happens, the hardware sets the *mcause* register with the corresponding exception code. The *pc* is set to the trap handler base address. The exception code helps to identify the type of exception. The possible exceptions in RISC-V are listed in Table

- Illegal instruction
- Instruction/Load/Store address misaligned
- Instruction/Load/Store access fault
- Environment call
- Break point

6.1.1 Illegal Instruction Exception

The exception occurs when the programs tries to execute any illegal instruction. For example trying to write on a read-only CSR register will generate a illegal instruction exception.

Example:

```
li t0, 8                    # t0 ⟵ 8
csrrs x0, mhartid, t0       # Attempt to write to a read-only CSR, generates exception
```

6.1.2 Instruction Address Misaligned Exception

The exception occurs when the programs tries to execute an unconditional jump or take a branch, wherein the target address is not 4 byte aligned. For example, executing a program with start address as 0x80000001. This will generate a instruction address misalignment exception on a unconditional jump.

Note:

Instruction address misaligned exceptions are not possible on machines that support extensions with 16-bit aligned instructions, such as the compressed instruction-set extension, C.

Example:

```
# _start address set to 0x80000001 (_start not aligned to 4 byte boundary.

_start:    la x15, loop       # x15 ⟵ Address (loop)
jalr ra, x15 ,0               # Jumping to a label (loop) which is not 4 byte aligned
                              # This causes an Instruction address misalignment exception
loop:     addi x10, x10,1     # x10 ⟵ x10+1
j loop                        # Jump to loop
```

6.1.3 Load Address Misaligned Exception

The exception occur when the programs tries to execute an load instruction to access data from misaligned address or an address that is not 4 byte aligned. For example, trying to access a data section without using a properly aligning it would cause this exception.

Example:

```
la x15, _data1          # x15 ⟵ Address (_data1)
lw x10, 0 ( x15 )       # x10 ⟵ Content(x15)
                        # Trying to load from a misaligned address (_data1)
li t0, 8

_data1:                 # _data1 section is not aligned to 4 byte boundary
   .word 3              # Load access at _data1 causes a misaligned exception
   .word 2
```

6.1.4 Store Address Misaligned Exception

The exception occurs when the programs tries to execute an store instruction at a misaligned address (Address that is not four byte aligned). For example trying to store data into a data section without using proper alignment, would cause this exception.

Example:

```
la x15, _data1          # x15 ⟵ (_data1) memory address
sw x10, 0 ( x15 )       # mem[x15+0] ⟵ x10
                        # Trying to store at a misaligned address (_data1)
sw x10, 0 ( x15 )

_data1:                 # _data1 section is not aligned to 4 byte boundary
   .word 3              # Store access at _data1 causes a misaligned exception
   .word 2
```

6.1.5 Instruction Access Fault

The exception occurs when the programs tries to access an instruction on a invalid memory location. For example executing unconditional jump instruction to a memory location which is out of bounds of the physical memory.

Example:

```
la x15, _data1        # x15 ⟵ Address of label (_data1)
jalr ra,-1(x15)       # Jumping to wrong addr, decoding contents at that addr

_data1:
   .word 100
   .word 99
```

In the above case, _data1 holds data values. The data values are aligned at word boundary. Now, we jump to a location, that is *_data1 - 1 byte* memory location. Here, when we execute 'jalr', an instruction access fault happens. The jump should have happened at 4 byte aligned address.

6.1.6 Load Access Fault

The exception occurs when the programs attempt to do a load on a invalid memory location. For example trying to load from address which is more than the bound of memory or inaccessible by memory. Certain registers are 32 bits of size. A 64 bit load operation might thrown an error.

Example:

```
_start:
la x15, _start          # x15 ⟵ Address (_start)
ld x16, -16 ( x15 )     # x16 ⟵ Content(x15-16) -Exception generated
```

6.1.7 Store Access Fault

The exception occurs when the programs attempts to do a store on an invalid memory location. For example, trying store to address which is more than the bound of memory or inaccessible by memory.

Example:

```
_start:
la x15, _start              # x15 ⟵ Address (_start)
sd x16, -16 ( x15 )         # x16 ⟶ Content(x15-16) -Exception generated
```

6.1.8 Break Point

The exception occurs when the programs executes a break-point set in the program to enter debug mode.

6.1.9 Environment Call

This exception occurs when the programs executes a system call. The system call is realized in RISC-V using *ecall* instruction. The ecall instructions can also used to switch from lower privilege modes to higher privilege modes. An example ecall instruction is demonstrated below.

Example:

```
addi x10, x10, 2
ecall                     # Environment call exception generated
```

6.2 Handling Exceptions

Once an exception happens the processor stops execution and passes the control the trap handler. Inbetween this, the processor privilege is set to Machine mode and processor sets the mcause register with exception code. The mepc is set with the *pc* of the instruction that caused the exception. All exception's come to the Machine Mode trap handler first. This applies for exceptions that arise from different privilege levels. The Machine Mode trap handler executes in Machine Mode. In the trap handler, first the context of the registers are saved in stack. Then the trap is serviced. After this the saved context in stack is restored back. This way, the trap is handled without causing much trouble to the execution flow.

Now, a question may arise on how the hardware jumps to the trap handler. This is established by setting the *mtvec* register with Tap handler's physical address. Usually the value in *mtvec* is called as "Trap entry".

Incase, we may not want to handle the exception in Machine Mode. we might want to handle it in Supervisor Mode or even User Mode. As such, there is a facility to "delegate" some or all exceptions to the lower privilege levels. These things will be seen in PART II.

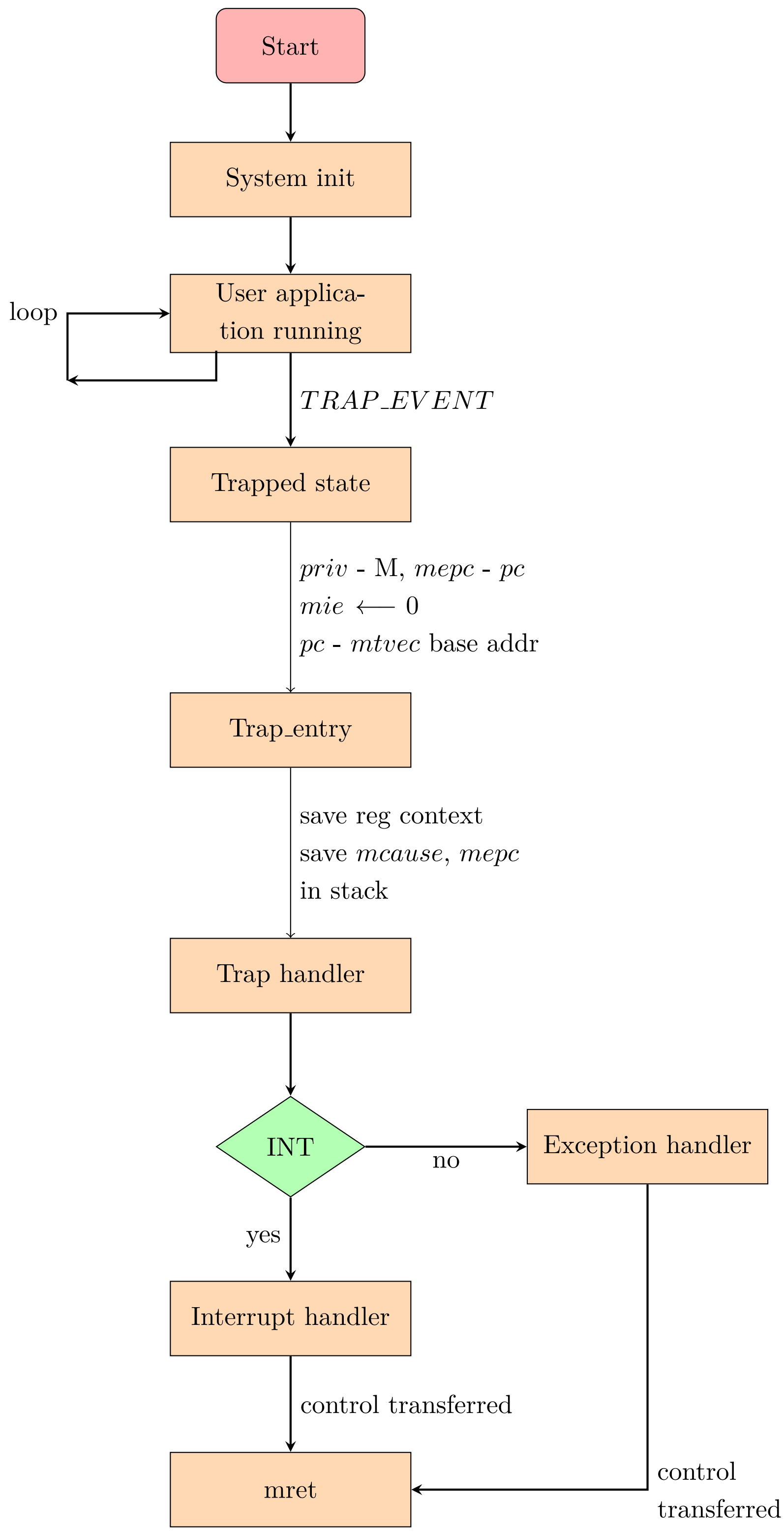


Figure 6.1: Trap occurrence and handling mechanism

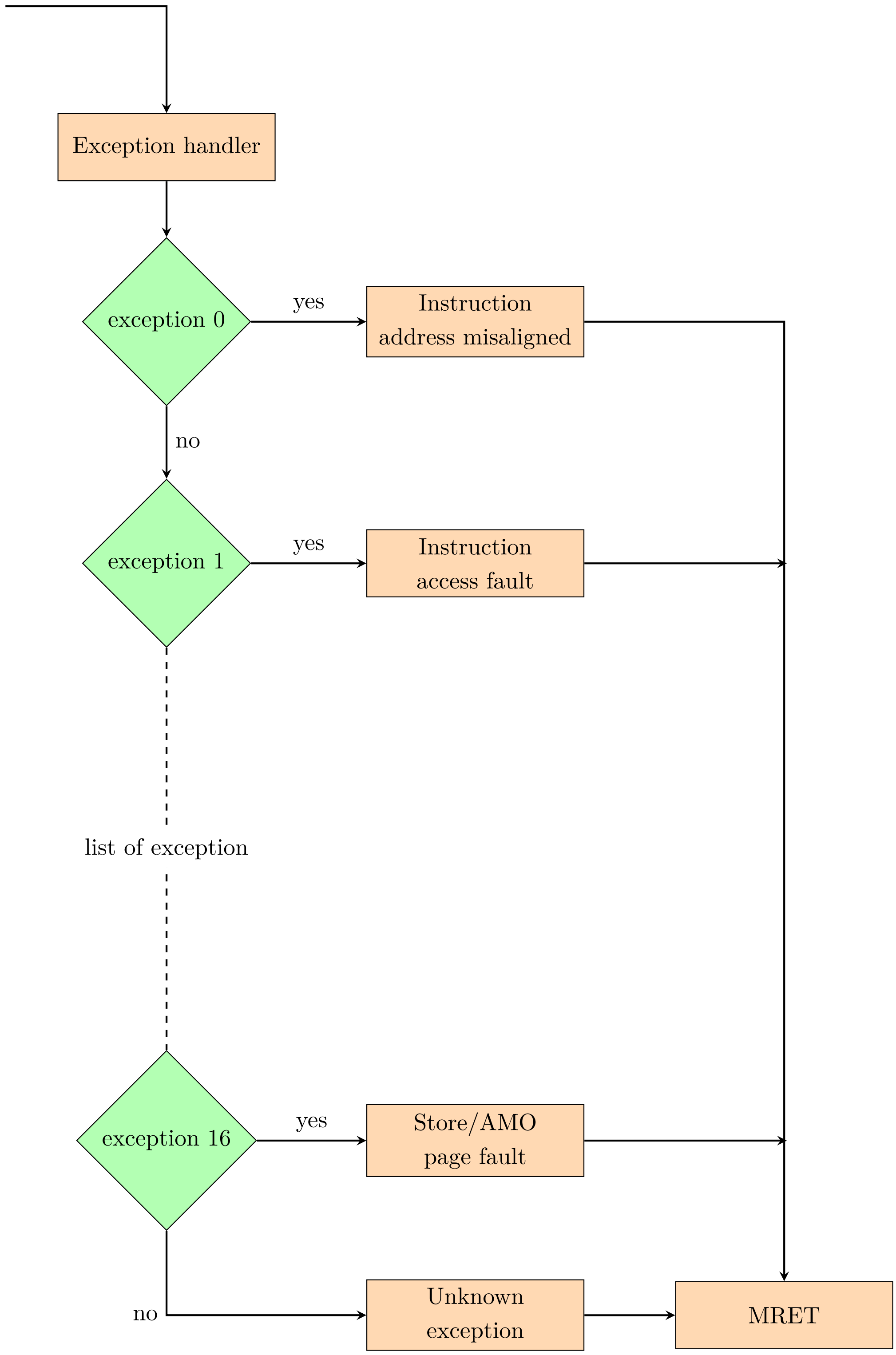


Figure 6.2: Exception handling part

The trap handler must begin on word aligned address boundary. This means that any address stored in the mtvec CSR must have “00” as the least significant two bits. Secondly, The RISC-V spec makes use of the last two bits in mtvec as follows.

- If the last two bits are “00”, then it means the CSR contains the address of a single trap handler.
- If the last two bits are “01”, then it means there is a collection of trap handlers, one for each type of asynchronous interrupt (Vectored Trap handler).
- The remaining bit patterns “10” and “11” are not used.

`Things to remember`:

When a trap occurs,

- The privilege mode is set to Machine Mode.
- The MIE (Interrupt enable) bit in the status word is set to 0.
- The MCAUSE register is set to indicate which event has occurred.
- The MEPC is set to the last instruction that was executing when system Trapped.
- The PC is set to MTVEC value. Incase of Vectored Traps handling, the PC is set mtvec base address + 4x(mcause).

6.2.1 Exception Handling Registers

The exception handling mechanism uses 4/5 registers to know all the information of a Trap. Those registers are CSR registers. A separate set of register is made available for each privilege level. Mstatus register has the Trap related information as bit information. Mepc register holds the physical address of the instruction, when exception happened. Mtvec has the base address of the Trap handler. It is usually referred to as the entry point of the Trap. Mcause has the exception of the Trap.

6.2.2 MSTATUS

`Machine Status Register (MSTATUS)` is used to enable/disable the interrupts. The mstatus register has many more bits. But these are the bits used with respect to a Trap.

Description

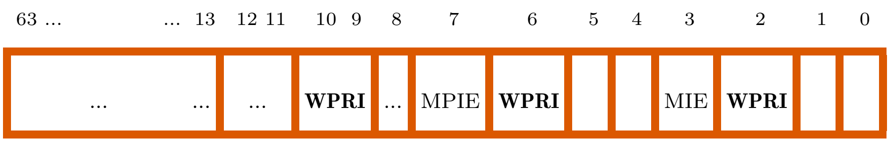


Figure 6.3: Machine-mode status register (`mstatus`) for RV64

`MSTATUS` contains a number of fields that can be read and updated. By modifying these fields, the software can do things like enable/disable interrupts and change the virtual memory model.

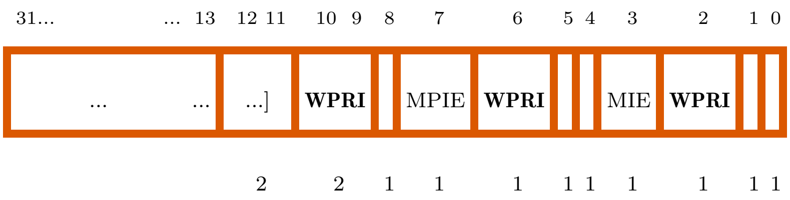


Figure 6.4: Machine-mode status register (`mstatus`) for RV32.

We use MSTATUS register while handling exceptions to read and set the MPP and SPP bits based on the requirement to switch privilege modes. This will be discussed in PART II.

Example:

```
li t0,0x800
csrrs zero, mstatus, t0        # Setting MPP bits on mstatus register
```

6.2.3 MRET

We were discussing earlier that mtvec register helps the hardware to locate the base address of the Trap handler. If there is an entry to a Trap, there should also be an exit. In the following section, we will be dealing with this part exactly.

`Machine Mode Trap Handler Return (MRET)` is used to return from a trap handler that is executing in the Machine Mode.

Syntax

`MRET`

Description

Once the trap is serviced and the saved context is restored. The mret instruction can be called. This instruction basically tells the processor to pass control back to the address in the mepc register. Incase of exception originating from a lower privilege level. The MRET instruction transfers control to that privilege level. The MPP field of the status register will be referred, to determine which mode to return to (either m, s, or u). The return will be effected by copying the saved program counter from `mepc` to the Program Counter (pc).

Exceptions

`MRET` may only be executed when running in Machine Mode.

6.3 Understanding Stack in RISC-V

6.3.1 Stack

Stack is an abstract data structure used to implement function calls in a program and holds data temporarily during a function call. Being a linear data-structure, a stack grows and shrinks during calls to function and is based on the last-in-first-out (LIFO) concept. The implementation of stack on an architecture is entirely at the software designer's disposal.

Availability of limited registers in an architecture, restricts the number of variables that can be used in a program. A stack serves the purpose of holding data temporarily during function calls. It is specifically used to store variables when a function or procedure call is made.

A stack is famously used for "UNDO" i.e., holding the history of an activity. For example, before switching over to a function, a stack is called upon to store the contents of the necessary registers as it may be modified during the execution of the function. After the function is executed, all registers can be restored with their values prior to the function call. This action of store and retrieval is called "PUSH and POP". Some architectures support the use of "PUSH" and "POP" keywords, while others use "LOAD" "STORE" instructions to do the same.

A program that implements a stack, sets aside a certain portion of the memory for its use. A register called "Stack Pointer" stores the address of the last program request in a stack. A program's stack is not generally hardware, but the Stack Pointer which points to the current area, is a CPU register. In RISC-V the stack is always kept 16-byte aligned.

Stack is implemented the following way in a RISC-V assembly language program:

- Initialize the Stack Pointer (sp) to a memory address
- Allocate space for Stack, by decrementing the sp by the number of locations required multiplied by XLEN[1] bytes. This will allocate memory for stack temporarily in memory.
  * addi sp, sp, -3*XLEN
- PUSH data onto stack. This essentially writes the register values to the stack.
  * sd x1, 1*XLEN(sp)
  * sd x2, 2*XLEN(sp)
  * sd x4, 2*XLEN(sp)
- POP data from stack. This essentially restores the register values back from the stack.
  * ld x1, 1*XLEN(sp)
  * ld x2, 2*XLEN(sp)
  * ld x4, 2*XLEN(sp)
- To free the stack, increment sp by the same number of locations used earlier ('n locations' multiplied by XLEN bytes). This will reset the stack pointer to the bottom of the caller stack.
  * addi sp, sp 3*XLEN

[1]XLEN is 4 bytes in RV32 and 8 bytes in RV64

chapter 7

Interrupts

Interrupts are asynchronous events triggered by external source. The processor may tend to process or ignore interrupts. Interrupts can be both software and hardware. In RISC-V interrupts are classified into timer, software and external interrupts. The external interrupts are also called as global interrupts. Timer interrupts are handled in the core. Software interrupts are internal to the processor, and external interrupts are handled by the PLIC module. In this chapter, we are going to see about handling Timer and External interrupts in RISC-V.

7.1 Timer Interrupts

A "timer interrupt" is caused when a separate timer circuit indicates that a predetermine interval has ended. The timer subsystem will interrupt the currently executing code. The timer interrupts are handled by the OS which uses them to implement time-sliced multi threading.

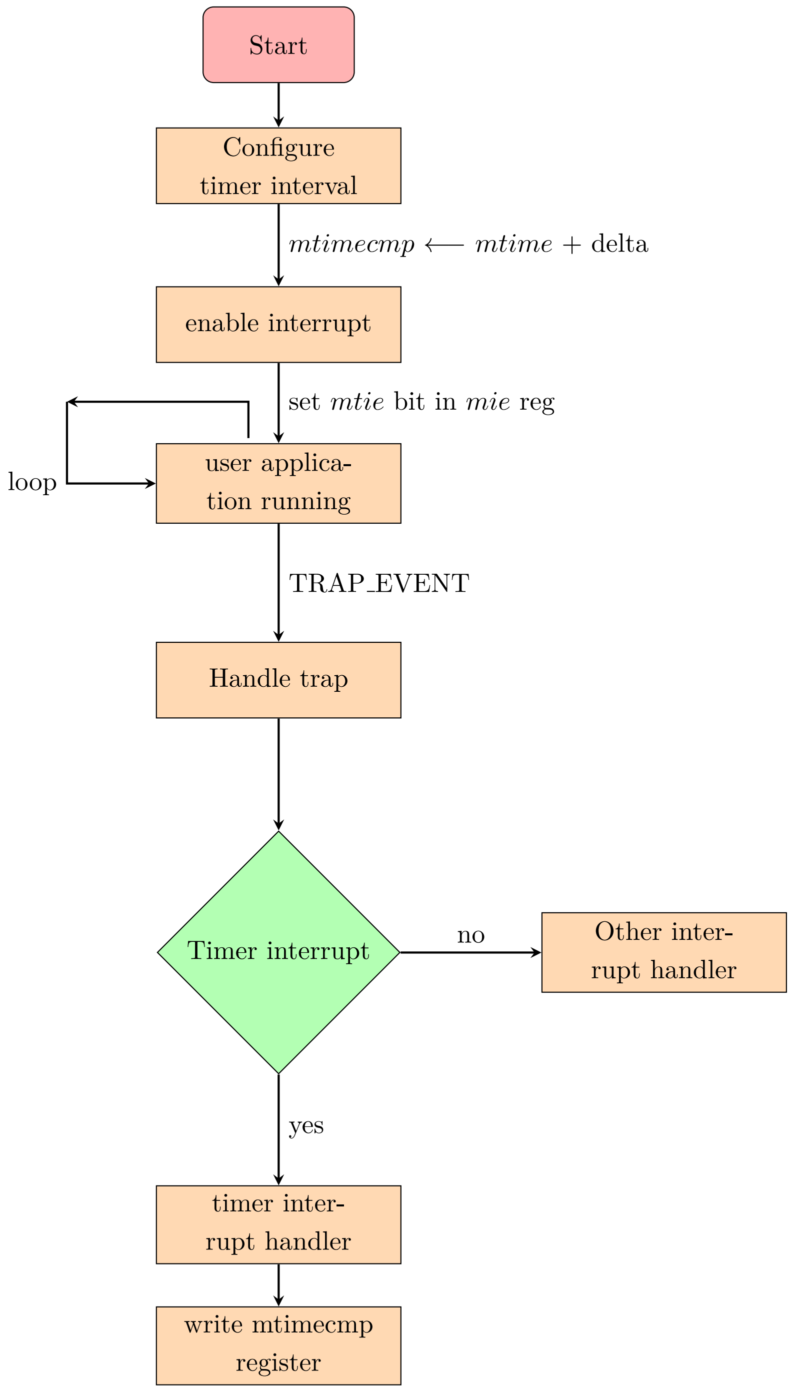
7.1.1 mtime Register

mtime register is a synchronous counter. It starts running from the time the processor is powered on and provides the current real time in ticks.

7.1.2 mtimecmp Register

This register is used to store the time period after which a timer interrupt should happen. The value of mtimecmp is compared with mtime register. When mtime value becomes greater mtimecmp, a timer interrupt happens. Both the mtime and mtimecmp registers are 64 bit memory mapped registers.

7.1.3 Timer Interrupt flow chart



7.1.3.1 Interrupt Enable Bits

Each of the Timer, Software, and External Interrupts can be enabled individually. Globally, all the interrupts can be enabled/disabled using the *MIE* bit in `MSTATUS` register. The MTIE, MSIE, MEIE bit enable's/disable's Timer, Software, and External interrupts individually.

7.1.3.2 Interrupt Processing Bits

When an interrupt occurs the *MPIE* bit will be set to hold the interrupt enable state. And the MIE bit is set to 0. This taken care by Hardware. This way the interrupt's are blocked and states are maintained.

7.2 External Interrupts

An “External Interrupt” comes from outside the processor and the precise nature of the cause will depend on the application. Such interrupts are asynchronous and are generated by external sources through the hardware, which maybe serviced by the processors. For example, a RISC-V processor used in an embedded process control system might receive external interrupts from various sensors demanding for appropriate action(s) to be taken. These interrupts are handled by the Platform Level Interrupt Controller (PLIC). The source of interrupts for PLIC are the devices connected to the SoC (IO, UART, SPI, etc...). As per the RISC-V specification these are termed as global interrupt sources, with each prioritised and routed by PLIC to the core. **For more detailed information on PLIC, kindly refer to the PLIC document provided in the link:** `http://shakti.org.in/documentation.html`

7.3 Software Interrupts

A “software interrupt” is caused by setting a bit in the machine status word. This can be useful in a multi-core chip where a thread running on one core needs to send an interrupt signal to another core.

Non-Maskable Interrupt Handling

Some traps are “maskable” and others are “non-maskable”. A maskable interrupt can either be handled, or can be ignored, or can be passed from a higher privilege level to a lower privilege level.

chapter

8 Assembler Directives

8.1 Object File section

Object files contain instructions and data. The instructions and data are stored in appropriate sections according to their use.

8.1.1 .TEXT

A read-only section containing the actual instructions of the program.

Syntax

.section .text or .text
data
instruction

Description

This portion of the object file or virtual address space is also known as the code segment or simply the text segment of the program. It contains executable instructions which cannot be modified at run-time. Any attempt to store into the .TEXT section will produce a "Segmentation" error and the program is terminated immediately. The code segment can contain constants in addition to instructions.

Usage

```
.text
li x5, 100
addi x5, x0, 100
```

8.1.2 .DATA

A read-write portion of the object file which contains data for the variables of the program.

Syntax
.section .data or .data
Variables

Description
The .DATA section contains initialized static variables that is global and static local variables.

Usage

```
.data
.word 1
helloworld:  .ascii "Hello World!"
```

8.1.3 .RODATA

Contains read-only data.

Syntax
.section .rodata or .rodata
data

Description
This section consists of read-only data for the program. But is not really enforced.

Usage

```
.rodata
mydata:  .asciz "Hello World!"
```

8.1.4 .BSS

The `Basic Service Set (.BSS)` is a read-write section containing uninitialized data.

Syntax

`.bss symbol, length, align`

where,

symbol	Local symbol
length	Reserve bytes to the length for symbol
align	Align to integer power two

Description

The `.BSS` directive is used for local common variable storage. When the program starts running, all the contents of this section are zeroed bytes. Since this section starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The .BSS section was invented to eliminate those explicit zeros from object files. In the program the .BSS section follows the data section.

Usage

```
.bss label1, 8, 4
```

8.1.5 .COMM

The `Common (.COMM)` common object to .BSS section, declares a common symbol named symbol.

Syntax

```
.comm symbol, length
```

where,

symbol	Local symbol
length	Reserve bytes to the length for symbol

Description

The `.COMM` declares a common symbol named symbol. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. The size of an object in the .BSS section is set by the .COMM directive.

Usage

```
.comm label1, 8
```

8.1.6 .COMMON

The `Common (.COMMON)` emit common object to .BSS section.

Syntax

```
.common symbol, length, .bss
```

where,

symbol	Local symbol
length	Reserve bytes to the length for symbol

Description

The `.COMMON` declares a common symbol named symbol. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. This directive behaves somewhat like .comm directive, but the syntax is different.

Usage

```
.common label1, 8
```

8.1.7 .SECTION

`Section (.SECTION)` directive assembles the following code into a section named `"name"`.

Syntax

```
.section name
```

where,

name	Name of section

Description

`.SECTION` instruction is only supported for targets that support arbitrarily named sections, on `"A.out"` targets.

Usage

```
.section A
```

8.1.8 Miscellaneous Functions

8.1.9 .OPTION

The `.OPTION` directive has a statically defined list of arguments with RISC-V options.

Syntax

```
.option argument
```

where,

argument	rvc, norvc, pic, nopic, push, pop

Description

The `.OPTION` directive modifies RISC-V specific assembler options inline with the assembly code. This is used when particular instruction sequences must be assembled with a specific set of options.

Usage

```
.option push
```

8.1.10 .FILE

The `.FILE` directive to start a new logical file.

Syntax

```
.file string
```

where,

string	new file name

Description

The `.FILE` directive, in general, the filename is recognized whether or not it is surrounded by quotes. But to specify an empty file name, the quotes must be given.

Usage

```
.file Hello
```

8.1.11 .IDENT

The `IDENT (.IDENT)` directive is accepted for source compatibility.

Syntax

```
.ident "string"
```

where,

string   file name

Description

The `.IDENT` directive is used by some assemblers to place tags in object files. It simply accepts the directive for source-file compatibility with such assemblers, but does not actually emit anything for it. At times it is used to place tags in object files. The behavior of this directive varies depending on the target.

Usage

```
.ident "GCC: (GNU) 7.2.0"       # "string" ⟵ GCC: (GNU) 7.2.0
```

8.1.12 .SIZE

The `.SIZE` is used to set the size associated with a symbol.

Syntax

```
.SIZE symbol, symbol
```

Description

The `.SIZE` directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside .def or .endef pairs.

Usage

```
memcpy:
mv x4, x5                    # x4 ⟵ x5
beqz x7, 1b                  # if x7 = 0; goto 1b
1:  add t1, t1, 1            # t1 ⟵ [t1+1]
add t2, t2, -1               # t1 ⟵ [t2-1]
.size memcpy, .-memcpy
```

8.1.12.1 .TYPE

The `.TYPE` directive is used to set the type of a symbol.

Syntax

`.type name, symbol`

where,

name	Type name
symbol	Value

Description

The `.TYPE` directive allows you to tell the assembler what type a symbol is.

Usage

```
.type int, 256       # 256 is of type int
```

8.1.13 Directives for Definition and Exporting of symbols

8.1.13.1 .GLOBAL

The `.GLOBAL` directive to globalize symbols.

Syntax

`.global symbol` or `.globl symbol`

where,

symbol	Variable, whose name is to be visible to entire program

Description

Usually, a defined symbol is visible only to partial program, only to the portion where it is defined. With the `.GLOBAL` directive its value is made available to other partial programs that are linked with it.

Usage

```
i:  word 5
.global i        # Variable i is made global
```

8.1.13.2 .LOCAL

The `.LOCAL` directive limit the visibility of symbols.

Syntax

`.local symbol`

where,

symbol	Local variable name

Description

The `.LOCAL` directive marks each symbol in the comma separated list of names as a local symbol, so that it will not be externally visible. If the symbols do not already exist, they will be created.

Usage

```
i:  word 5
.local i       # Variable i is made local
```

8.1.13.3 .EQU

The `EQUATE (.EQU)` directive sets the value of symbol to expression.

Syntax

```
.equ symbol, expression
```

where,

symbol   Local value

Description

The `.EQU` directive has two operands separated by a comma. Wherever the first operand appears in the program, the assembler replaces it with the second operand. Used only while assembling your code, once the symbol is defined, its value can not be changed in the remaining part of the source code.

Usage

```
.equ counter, 3      # counter ⟵ 3
```

8.2 Alignment Control

The ALIGN directive aligns the next instruction to a specified boundary by padding with zeros or NOP instructions.

8.2.0.1 .ALIGN

The `.ALIGN` directive aligns the next instruction by a given byte boundaries.

Syntax

```
.align size
```

where,

size   Byte boundary

Description

The `.ALIGN` directive gives the location counter desired alignment in bytes.

Usage

```
.align 2    # Align to 4-bytes
```

8.2.0.2 .BALIGN

The `.BALIGN` directive aligns member byte boundaries with padding.

Syntax

```
.balign size
```

where,

size	Byte boundary

Description

The `.BALIGN` directive pads location counter to a particular storage boundary.

Usage

```
.balign 8     # Align to 8-bytes
```

8.2.0.3 .P2ALIGN

The `.P2ALIGN` directive directive aligns member byte boundaries with padding. Alias for `.ALIGN` directive.

Syntax

```
.p2align size
```

where,

size	Byte boundary

Description

The `.P2ALIGN` directive pads location counter to a particular storage boundary. Alignment done to the power of 2.

Usage

```
.p2align 3   # Align to 8-bytes
```

8.3 Assembler Directives for Emitting Data

Assembler directives are instructions to the assembler to perform various bookkeeping tasks, storage reservation, and other control functions.

8.3.0.1 .2BYTE

The `.2BYTE` directive for unaligned 16-bit comma separated words.

Syntax

```
.2byte value
```

where,

value	Value to be initialized

Description

The `.2BYTE` directive initializes the specified value to 2 bytes or 16-bit unaligned integers. It can also store multiple comma-separated values. The operands specified can be decimal, hex, binary, or character constants, but not labels.

Usage

```
.2byte 0x1000
```

8.3.0.2 .4BYTE

The `.4BYTE` directive for unaligned 32-bit comma separated words.

Syntax

```
.4byte value
```

where,

value	Value to be initialized

Description

The `.4BYTE` directive initializes the specified value to 4 bytes or 32-bit unaligned integers. It can also store multiple comma-separated values. The operands specified can be decimal, hex, binary, or character constants, but not labels.

Usage

```
.4byte 0x1000000
```

8.3.0.3 .8BYTE

The `.8BYTE` directive for unaligned 64-bit comma separated words.

Syntax:

```
.8byte value
```

where,

value	Value to be initialized

Description

The `.8BYTE` directive initializes the specified value to 8 bytes or 64-bit unaligned integers. It can also store multiple comma-separated values. The operands specified can be decimal, hex, binary, or character constants, but not labels.

Usage

```
.8byte 0x1000000000000000
```

8.3.0.4 .HALF

The `.HALF` directive for naturally aligned 2byte or 16-bit comma separated words.

Syntax

```
.half value
```

where,

value	Value to be initialized

Description

The `.HALF` directive initializes the specified value to 2 bytes or 16-bit aligned integers. It can also store multiple comma-separated values. The operands specified can be decimal, hex, binary, or character constants, but not labels.

Usage

```
.half 0x1000
```

8.3.0.5 .WORD

The `.WORD` directive for naturally aligned 4-bytes or 32-bit comma separated words.

Syntax

```
.word value
```

where,

value	Value to be initialized

Description

The `.WORD` directive initializes the specified value to 4 bytes or 32-bit aligned integers. It can also store multiple comma-separated values and the operands specified can be decimal, hex, binary, or character constants, but not labels.

Usage

```
.word 0x1000000
```

8.3.0.6 .DWORD

The `Double Word (.DWORD)` directive for naturally aligned 8-bytes or 64-bit comma separated words.

Syntax

```
.dword value
```

where,

value	Value to be initialized

Description

The `.DWORD` directive creates a double word constant. They can also store multiple comma separated values. The operands specified can be decimal, hex, binary, or character constants, but not labels.

Usage

```
.dword 0x7000000000000000
```

8.3.0.7 .BYTE

The `.BYTE` directive for unaligned 8-bit comma separated words.

Syntax

```
.byte value
```

where,

value	Value to be initialized

Description

The `.BYTE` directive initializes the specified value to 1 bytes or 8-bit unaligned integers. It can also store multiple comma-separated values. The operands specified can be decimal, hex, binary, or character constants, but not labels.

Usage

```
.byte 0x10
```

8.3.1 .ASCIZ

`ASCIZ (.ASCIZ)` instruction is similar to the ascii instruction and emits the specified string within double quotes.

Syntax

```
.asciz "string"
```

where,

"String"	User specified string

Description

The `.ASCIZ` instruction is like the ascii instruction, but each string is followed by a zero byte. The "z" in .ASCIZ stands for zero. For this directive, the assembler increments the location counter by the length of the string, including the null character at the end. This directive is easier to read for text strings.

Usage

```
.asciz "Hello World"
```

8.3.2 .STRING

`String (.STRING)` instruction emits the specified string.

Syntax

`.string "String"`

where,

"String"	User specified string

Description

For the `.STRING` directive, the assembler increments the location counter by the length of the string, including the null character at the end.

Usage

```
.string "Hello World"
```

8.3.3 .INCBIN

`Include Binary (.INCBIN)` instruction emits the included file as a binary sequence of octets.

Syntax

`.incbin "file"`

where,

"file"	File to be included

Description

The `.INCBIN` instruction takes any file and includes it within the file being compiled. The file is included as it is, without being assembled.

Usage

```
.incbin "hello.c"        # File. ⟵ hello.c
```

This instruction includes the file "hello.c" into the file "File. ".

8.3.4 .ZERO

`Zero Bytes (.ZERO)` instruction reserves a block of memory.

Syntax

`.zero integer`

where,

integer	Number of bytes to reserve

Description

`.ZERO` instruction reserves a block of memory as an input buffer, it reserves and initializes a block of memory to zero.

Usage

```
.zero 100    # mem[100-bytes] ⟵ 0
```

This instruction reserves 100 bytes of memory and stores zeros in them.

chapter 9

Example Programs and Practice exercises

9.1 Important Prerequisites

1. The necessary files to compile and simulate ASM programs in spike environment, are hosted inside the **spiking** folder. Do the following in a terminal:
   (a) cd $HOME
   (b) git clone https://gitlab.com/shaktiproject/software/spiking.git
2. Move to spiking folder
   (a) cd spiking
3. Compile and generate dump for a program
   (a) riscv64-unknown-elf-gcc -nostdlib -nostartfiles -T spike.lds example.S -o example.elf
   (b) riscv64-unknown-elf-objdump -d example.elf & > example.dump
4. Debugging, Loading and Executing an ASM program. Open **three** separate terminals, ensuring each are within the spiking folder. Run the following commands individually in each terminal.
   (a) $(which spike) –rbb-port=9824 -m0x10010000:0x20000 bootload.elf $(which pk)
   (b) sudo $(which openocd) -f spike.cfg
   (c) riscv64-unknown-elf-gdb
      i. (gdb) target remote localhost:3333
      ii. (gdb) file example.elf
      iii. (gdb) load

(d) Execute a program line by line using "step in" command

i. si

(e) To check contents of registers

i. (gdb) info reg

For more detailed information, please visit: https://shakti.org.in/learn_with_shakti/intro.html

Note: All programs illustrated here have been tested on the spike simulator with a BRAM-memory starting address set to 0x10010000.

9.2 Assembly Language Example Programs

9.2.1 Data Transfer Instructions

9.2.1.1 To load 8, 16, 32 and 64 bit numbers into individual register

```
_start:
  andi t0, t0, 0               # Clear register t0
  andi t1, t1, 0               # Clear register t1
  andi t2, t2, 0               # Clear register t2
  andi t3, t3, 0               # Clear register t3
  li t0, 0xFF                  # Load a 8-bit number to t0
  li t1, 0xFFFF                # Load a 16-bit number to t1
  li t2, 0xFFFFFFFF            # Load a 32-bit number to t2
  li t3, 0x7FFFFFFFFFFFFFFF    # Load a 64-bit number to t3
```

9.2.1.2 Register to register data transfer

```
_start:
  andi t0, t0, 0      # Clear register t0
  andi t1, t1, 0      # Clear register t1
  li t0, 0x4A         # Load register t0 with a value
  mv t1, t0           # Copy contents of register t0 to register t1
```

9.2.1.3 Register to memory data transfer

a. Store Byte – 1 Byte

```
_start:
  andi t0, t0, 0          # Clear register t0
  andi t1, t1, 0          # Clear register t1
  li t0, 0x10011000       # Load register t0 with an address
  li t1, 0x71             # Load register t1 with a 1-Byte value
  sb t1, 0(t0)            # Store the byte in t1 into first byte slot of
                          address specified in t0
```

```
li t1, 0x79            # Load register t1 with another 1-Byte value
sb t1, 1(t0)           # Store the byte in t1 into second byte slot of
                       address specified in t0
```

b. Store Half-Word – 2 Bytes

```
_start:
  andi t0, t0, 0         # Clear register t0
  andi t1, t1, 0         # Clear register t1
  li t0, 0x10011000      # Load register t0 with an address
  li t1, 0x7971          # Load register t1 with a 2-Byte (half-word)
                         value
  sh t1, 0(t0)           # Store the half-word in t1 to the first
                         half-word slot of address specified in t0
  li t1, 0x7B7A          # Load register t1 with another 2-Byte
                         (half-word) value
  sh t1, 2(t0)           # Store the half-word in t1 to the second
                         half-word slot of address specified in t0
```

c. Store Word – 4 Bytes

```
_start:
  andi t0, t0, 0         # Clear register t0
  andi t1, t1, 0         # Clear register t1
  li t0, 0x10011000      # Load register t0 with an address
  li t1, 0x7B7A7971      # Load register t1 with a 4-Byte (1 word) value
  sw t1, 0(t0)           # Store the word in t1 to the first-word slot of
                         address specified in t0
  li t1, 0x7F7E7D7C      # Load register t1 with another 4-Byte (1-word)
                         value
  sw t1, 4(t0)           # Store the word in t1 to the second word slot
                         of address specified in t0
```

d. Store Double – 8 Bytes

```
_start:
  andi t0, t0, 0                   # Clear register t0
  andi t1, t1, 0                   # Clear register t1
  andi t1, t1, 0                   # Clear register t1
  li t0, 0x10011000                # Load register t0 with an address
  li t1, 0x7F7E7D7C7B7A7971        # Load register t1 with double word
                                   (8-bytes = 2 words) value
  sd t1, 0(t0)                     # Store the double word in t1 to
                                   address specified in t0
```

9.2.1.4 Register to stack memory data transfer

```
_start:
  andi t0, t0, 0                # Clear register t0
  andi t1, t1, 0                # Clear register t1
  li sp, 0x10012000             # Setting the stack pointer register to an
                                address

  li t0, 0x7776757473727170     # Load a 64-bit (8-bytes) value to register t0
  li t1, 0x7F7E7D7C7B7A7978     # Load a 64-bit (8-bytes) value to register t1

  .p2align 2                    # Aligning the stack - Storage boundary
  addi sp, sp, -2*8             # Setting depth of the stack
  nop
  sd t0, 1*8(sp)                # Storing contents of t0 into first stack
                                pointer slot

  sd t1, 2*8(sp)                # Storing contents of t0 into second stack
                                pointer slot

  addi sp, sp, 2*8              # Collapse stack
```

9.2.2 Arithmetic Instructions

9.2.2.1 Addition - Illustrating addition operation between contents of two registers and contents of a register with an immediate value

```
_start:
  andi t0, t0, 0                # Clear register t0
  andi t1, t1, 0                # Clear register t1
  andi t2, t2, 0                # Clear register t2
  andi t3, t3, 0                # Clear register t3
  li t0, 0x1A352A9C             # Loading register t0 with a value
  li t1, 0x1B2D4C6A             # Loading register t1 with a value
  addi t2, t0, 0x1CB            # Add t0 with an immediate value
  add t2, t0, t1                # Add -- t0 with t1 and place the result in t2
  addw t3, t0, t1               # Add -- t0 with t1 and place the 32-bit result
                                in t3
```

9.2.2.2 Subtraction - Illustration the subtraction operation between contents of two registers

```
_start:
  andi t0, t0, 0                # Clear register t0
  andi t1, t1, 0                # Clear register t1
  andi t2, t2, 0                # Clear register t2
  andi t3, t3, 0                # Clear register t3
  li t0, 0x1A03533A12054021     # Load register t0 with a value
  li t1, 0x3B14875C35286142     # Load register t1 with a value
  sub t2, t1, t0                # Subtract t0 from t1 and place the result in t2
  subw t3, t1, t0               # Subtract t0 from t1 and place the 32-bit
                                result in t3
```

9.2.2.3 Multiplication - Illustrating different multiplication operations between contents of two registers

```
_start:
  andi t0, t0, 0                # Clear register t0
  andi t1, t1, 0                # Clear register t1
  andi t2, t2, 0                # Clear register t2
  andi t3, t3, 0                # Clear register t3
  andi t4, t4, 0                # Clear register t4
  andi t5, t5, 0                # Clear register t5
  li t0, -43                    # Load register t0 with a negative value
  li t1, 187                    # Load register t1 with a positive value
  mulh t3, t0, t1               # Signed Multiplication of t0 with t1 and place
                                the most significant half of the result in t3
  mul t2, t0, t1                # Multiplication of t0 with t1 and place the
                                lower half of the result in t2
  mulhu t4, t0, t1              # Unsigned Multiplication of t0 with t1 and
                                place the most significant half of the result
                                in t4
  mulw t5, t0, t1               # Multiply-word, multiply t0 with t1 and place
                                the result in t5
```

9.2.2.4 Division - Illustrating different division operations between contents of two registers and procuring the quotient of the division operation into a register

```
_start:
  andi t0, t0, 0                # Clear register t0
  andi t1, t1, 0                # Clear register t1
  andi t2, t2, 0                # Clear register t2
  andi t3, t3, 0                # Clear register t3
  andi t4, t4, 0                # Clear register t4
  andi t5, t5, 0                # Clear register t5
  li t0, -2516                  # Load register t0 with a negative value
  li t1, 74                     # Load register t1 with a positive value
  div t2, t0, t1                # Divide t0 by t1 and place quotient in t2
  li t3, 1332                   # Load register t3 with a positive value
  li t4, 18                     # Load register t4 with a positive value
  divu t5, t3, t4               # Unsigned division of t3 by t4 and place
                                quotient in t5
```

9.2.2.5 Remainder - Illustrating different division operations between contents of two registers and procuring the remainder of the division operation into a register

```
_start:
  andi t0, t0, 0                # Clear register t0
  andi t1, t1, 0                # Clear register t1
  andi t2, t2, 0                # Clear register t2
  andi t3, t3, 0                # Clear register t3
  andi t4, t4, 0                # Clear register t4
```

```
andi t5, t5, 0          # Clear register t5
li t0, -2516            # Load register t0 with a negative value
li t1, 75               # Load register t1 with a positive value
rem t2, t0, t1          # Divide t0 by t1 and place the remainder in t2
li t3, 1332             # Load register t3 with a positive value
li t4, 118              # Load register t4 with a positive value
remu t5, t3, t4         # Unsigned divide t3 by t4 and place the
                        remainder in t5
```

9.2.3 Logical Operations - Illustrating various logical operations with immediate values and between contents of registers

9.2.3.1 ANDI

```
_start:
  andi t0, t0, 0                          # Clear register t0
  andi t1, t1, 0                          # Clear register t1

  li t0, 0x13372D6                        # Load t0 register with a value
  andi t1, t0, 0xFC                       # Logical AND-Immediate operation
                                          of contents of t0 with an immediate
                                          value.  Result is placed in t1
```

9.2.3.2 AND

```
_start:
  andi t0, t0, 0                          # Clear register t0
  andi t1, t1, 0                          # Clear register t1
  andi t2, t2, 0                          # Clear register t2

  li t0, 0x13372D6                        # Load t0 register with a value
  li t1, 0xFFFFFFC                        # Load t1 register with a value
  and t2, t0, t1                          # Logical AND operation between
                                          contents of registers t0 and t1, with
                                          the result placed in t2
```

9.2.3.3 ORI

```
_start:
  andi t0, t0, 0                          # Clear register t0
  andi t1, t1, 0                          # Clear register t1

  li t0, 0xC53D6                          # Load t0 register with a value
  ori t1, t0, 0x5C                        # Logical OR-Immediate operation of
                                          t0 with an immediate value, result is
                                          placed in t1
```

9.2.3.4 OR

```
_start:
  andi t0, t0, 0                          # Clear register t0
  andi t1, t1, 0                          # Clear register t1
  andi t2, t2, 0                          # Clear register t2

  li t0, 0xC53D6                          # Load t0 register with a value
  li t1, 0xD6332                          # Load t1 register with a value
```

```
or t2, t0, t1                  # Logical OR operation between
                               contents of registers t0 and t1, with
                               the result placed in t2
```

9.2.3.5 X-ORI

```
_start:
  andi t0, t0, 0               # Clear register t0

  xori t0, x0, 0xD6            # Logical X-OR operation with an
                               immediate value
```

9.2.3.6 X-OR

```
_start:
  andi t0, t0, 0               # Clear register t0
  andi t1, t1, 0               # Clear register t1
  andi t2, t2, 0               # Clear register t2

  li t0, 0xC53D6               # Load t0 with a number
  li t1, 0xD6332               # Load t1 with a number
  xor t2, t0, t1               # Logical X-OR operation between
                               contents of two registers
```

9.2.3.7 NOT

```
_start:
  andi t0, t0, 0               # Clear register t0
  andi t1, t1, 0               # Clear register t1
  li t0, 0xFFFFFFFFFFFFFFD3    # Load t0 register with a number
  not t1, t0                   # Logical NOT operation on the
                               contents of t0, result is placed in
                               register t1
```

9.2.4 Conditional Operations - Illustrating conditional operations between contents of registers

9.2.4.1 If...then...Else and the nested If

If statement

```
_start:
  andi t0, t0, 0               # Clear register t0
  andi t1, t1, 0               # Clear register t1
  li t0, -2                    # Load t0 register with a negative
                               value

  slt t1, t0, x0               # Set t1 to 1 if t0 is less than 0
  j Endif                      # Short jump to end of statement
```

```
    Endif:  j Endif                        # End of If
```

If-Else statement

```
  _start:
    andi t0, t0, 0                         # Clear register t0
    andi t1, t1, 0                         # Clear register t1
    andi t2, t2, 0                         # Clear register t2
    andi t3, t3, 0                         # Clear register t3
    li t0, 2                               # Load t0 with a number
    li t3, -2                              # Load t3 with a number
     slt t1, t0, x0                        # Set t1 to 1 if t0<0
     beq t1, x0, Else                      # If t1=0, goto "Else" statement
     j Endif                               # End If statement
     Else:  sgt t2, t3, x0                 # Else statement, t2=1 if t3>0
    Endif:  j Endif                        # End of If-Else conditional
                                           statements
```

If-ElseIf-Else statement

```
  _start:
    andi t0, t0, 0                         # Clear register t0
    andi t1, t1, 0                         # Clear register t1
    andi t2, t2, 0                         # Clear register t2
    andi t3, t3, 0                         # Clear register t3
    andi t4, t4, 0                         # Clear register t4
    andi t5, t5, 0                         # Clear register t5
    li t0, 2                               # Load t0 with a positive value
    li t3, -2                              # Load t3 with a negative value
     slt t1, t0, x0                        # Set t1 to 1 if t0 < 0
     beq t1, x0, ElseIf                    # Goto ElseIf statement if t1 = 0
     j Endif                               # End If statement
    ElseIf:  sgt t4, t3, x0                # Set t4 to 1 if t3 > 0
     beq t4, x0, Else                      # Goto Else statement if t4 = 0
     j Endif                               # End "Else" statement
    Else:  seqz t5, t4, x0                 # Set t5 to 1 if t4 = 0
    Endif:  j Endif                        # End of If-ElseIf-Else conditional
                                           statements
```

Nested If-Else statement

```
  _start:
    andi t0, t0, 0                         # Clear register t0
    andi t1, t1, 0                         # Clear register t1
    andi t2, t2, 0                         # Clear register t2
    andi t3, t3, 0                         # Clear register t3
    andi t4, t4, 0                         # Clear register t4
    li t0, 100                             # Load t0 with a value
    li t1, 200                             # Load t1 with a value
    If:  beq t0, t1, Else                  # Goto Else if t0 = t1
```

```
    IfIf:  sgt t2, t0, t1                   # Set t2 to 1 if t0 > t1
      beq t2, x0, IfElse                    # Goto IfElse if t2 = 0
      j Endif                               # End of If statement
    IfElse:  seqz t3, t2                    # Set t3 to 1 if t2 = 0
      j Endif                               # End of If statement
  Endif:  j Endif                           # End of Nested If conditional
                                            statements
```

While Loop

```
_start:
  andi t0, t0, 0                            # Clearing contents of register t0
                                            # Functions as index "i" for the loop
  andi t1, t1, 0                            # Clearing contents of register t1
                                            # Holds value to compare index with
  andi t2, t2, 0                            # Clearing contents of register t2
                                            # Functions as variable "sum"
  li t1, 100                                # Load t1 with value 100
  loop:  add t2, t2, t0                     # Sum = Sum+i
    addi t0, t0, 1                          # Increment index "i"
    blt t0, t1, loop                        # Iterate if t0<t1
  End:  j End                               # End of WHILE loop
```

For Loop

```
_start:
  andi t0, t0, 0                            # Clear register t0
                                            # Functions as index "i" for the loop
  andi t1, t1, 0                            # Clearing contents of register t1
                                            # Functions as variable "sum"

  loop:  andi t2, t2, 0                     # For loop begins
                                            # Clear t2 before starting the loop
    add t1, t1, t0                          # Compute sum=sum+i

    addi t0, t0, 1                          # Increment i by 1
    slti t2, t0, 100                        # Set t2 to 1 if t0<100
    bne t2, x0, loop                        # Iterate if t2≠0
  End:  j End                               # End of FOR loop
```

Switch Case

```
_start:
                                            # Clearing/Initializing contents of
                                            five registers to 0

  mv a0, x0
  mv a4, x0
  mv a5, x0
  mv a7, x0
  mv t3, x0
```

```
li a7, 9164                                # Loading a7 with a number
li t3, 58                                  # Loading t3 with a number

switch_case:  la a0, _data1                # Begin Switch Case
                                           # Load address of location where list
                                           of operators are stored
      lw a4, 16(a0)                        # Load choice of operator into a4

  case_add:  lw a5, 0(a0)                  # Addition case.  Load Addition
                                           operator to a5
    xor a5, a5, a4                         # If a5 = a4, XOR the two will result
                                           in zero
    bne a5, x0, case_sub                   # If a5 ≠ 0, goto Subtraction case
    add a5, a7, t3                         # Add a7 with t3 and store the result
                                           in a5
    j End                                  # Break

  case_sub:  lw a5, 4(a0)                  # Subtraction case.  Load Subtraction
                                           operator to a5
    xor a5, a5, a4                         # If a5 = a4, XOR the two will result
                                           in zero
    bne a5, x0, case_mul                   # If a5 ≠ 0, goto Multiplication case
    sub a5, a7, t3                         # Subtract t3 from a7 and store the
                                           result in a5
    j End                                  # Break

  case_mul:  lw a5, 8(a0)                  # Multiplication case.  Load
                                           Multiplication operator to a5
    xor a5, a5, a4                         # If a5 = a4, XOR the two will result
                                           in zero
    bne a5, x0, case_div                   # If a5 ≠ 0, goto Division case
    mul a5, a7, t3                         # Multiply a7 with t3 and store the
                                           product in a5
    j End                                  # Break

  case_div:  lw a5, 12(a0)                 # Division case.  Load Division
                                           operator to a5
    xor a5, a5, a4                         # If a5 = a4, XOR the two will result
                                           in zero
    bne a5, x0, default                    # If XOR ≠ 0, goto Default case
    div a5, a7, t3                         # Divide a7 by t3 and store the
                                           quotient in a5
    j End                                  # Break

  default:  li a5, 0xDEADBEEF              # Default case.  Load a5 with
                                           DEADBEEF if none of the cases match


  .p2align 0x2                             # Align data section to eight bytes
  _data1:                                  # Data section label
```

```
                                          # List of operators and user's choice
                                          of operator
    .word '+'
    .word '-'
    .word '*'
    .word '/'

    .word '*'                             # User's choice of operator
```

9.2.5 Exercises

9.2.5.1 A Program to find the number of even and odd elements in an array

a. Using the remainder method

```
_start:
  .data                                   # Data for the program
    Array:  .byte 12,19,45,69,98,23       # Array of even and odd numbers
  .text                                   # Code section of the program
    andi t0, t0, 0                        # Even number count
    andi t1, t1, 0                        # Odd number count
    andi t2, t2, 0                        # Holds the address and elements of the
                                          Array
    andi t3, t3, 0                        # For loop index i
    andi t4, t4, 0                        # Holds size of Array
    andi t5, t5, 0                        # Holds value to divide Array numbers with,
                                          to determine even or odd
  li t4, 6                                # Size of array
  li t5, 2                                # Value to divide array elements with

  FOR_loop:  bge t3, t4, END              # Condition to control loop iterations
    la t2, Array                          # Load address of Array
    add t2, t2, t3                        # Increment Array index
    lb t2, 0(t2)                          # Load an element from the Array
    rem t2, t2, t5                        # Divide the Array element by t5 and store
                                          remainder in t2

    IF: bnez t2, ELSE                     # Control execution of condition
      addi t0, t0, 1                      # Increment even number count
      addi t3, t3, 1                      # Increment index i
      j FOR_loop                          # Iterate FOR loop

    ELSE:
      addi t1, t1, 1                      # Increment odd number count
      addi t3, t3, 1                      # Increment index i
      j FOR_loop                          # Iterate FOR loop

  END: j END                              # End of program
```

b. Using the masking method

```
_start:
  andi t0, t0, 0                    # Even number count
  andi t1, t1, 0                    # Odd number count
  andi t2, t2, 0                    # Holds the address and elements of the
                                    Array
  andi t3, t3, 0                    # For loop index i
  andi t4, t4, 0                    # Holds size of Array
  andi t5, t5, 0                    # Holds value to divide Array numbers with,
                                    to determine even or odd

  li t4, 6                          # Size of array

  FOR_loop:  bge t3, t4, END        # Condition to control loop iterations
    la t2, Array                    # Load address of Array
    add t2, t2, t3                  # Increment Array index
    lb t2, 0(t2)                    # Load an element from the Array
    and t2, t2, 1                   # Mask t2 with 1 to check whether LSB is 1
                                    or not

    IF: bnez t2, ELSE               # Execute condition if number is even
      addi t0, t0, 1                # Increment even number count
      addi t3, t3, 1                # Increment index i
      j FOR_loop                    # Iterate FOR loop

    ELSE:                           # Execute condition if number is odd
      addi t1, t1, 1                # Increment odd number count
      addi t3, t3, 1                # Increment index i
      j FOR_loop                    # Iterate FOR loop

  END: j END                        # End of program
```

9.2.5.2 Program to find the Fibonacci series for a specified range, without recursion

```
_start:
  andi t0, t0, 0                    # Will hold address for an array
  andi t1, t1, 0                    # Number of elements in the series
  andi t2, t2, 0                    # First number in the series
  andi t3, t3, 0                    # Second number in the series
  andi t4, t4, 0                    # Third number in the series
  andi t5, t5, 0                    # Variable to control loop
  li t0, 0x10010                    # Setting an address to store
                                    elements in array
  li t1, 7                          # Number terms required in the series
  li t2, 0                          # Load first element in the series
  li t3, 1                          # Load second element in the series
  li t5, 1                          # Initializing loop index

  sb t2, 0(t0)
```

```
sb t3, 1(t0)

loop:  bgt t5, t1, END                  # Condition to control number of
                                        iterations
  addi t5, t5, 1                        # Increment index by 1
  add t4, t2, t3                        # Add terms in n and (n-1), store
                                        result in t4
  add t0, t0, t5                        # Move through terms in Array

  sb t4, 0(t0)                          # Update Array with computed number
                                        in the series

  mv t2, t3
  sub t0, t0, t5
  j loop                                # Iterate
END: j END                              # End of program
```

9.2.5.3 In Place Bubble Sort

```
_start:
  .data                                 # Data section of bubble-sort program
    Array:  .byte 6,7,3,2,9,8           # Array of unsorted data
    Arraysize:  .byte 6                 # Defining size of array
  .text                                 # Commands section of the program
    andi t0, t0, 0                      # Clear contents of register t0; Holds
                                        array location
    andi t1, t1, 0                      # Clear contents of register t1; Holds
                                        index of inner FOR loop
    andi t3, t3, 0                      # Clear contents of register t3; Holds
                                        content of current array location
    andi t4, t4, 0                      # Clear contents of register t4
    andi t5, t5, 0                      # Clear contents of register t5; Holds
                                        content of adjacent array location
    andi t6, t6, 0                      # Clear contents of register t6; Acts as
                                        temporary variable during swaps
    la t0, Array                        # Load address where unsorted Array is
                                        stored
    la t1, Arraysize                    # Load address where size of array is
                                        stored
    lb t1, 0(t1)                        # Load a number from the array
    addi t1, t1, -1                     # Number of swaps to be made
    andi x1, x1, 0                      # Clear contents of x1
    outerloop:                          # Outer FOR loop
      bge x0, t1, outerend              # Jump to end if t1=0
      andi t2, 0                        # Clear contents of register t2
      innerloop:                        # Inner FOR loop
        bge t2, t1, innerend            # Jump to end of inner FOR loop if t2=t1
        lb t3, 0(t0)                    # Load the first number from unsorted array
                                        to t3
        lb t5, 1(t0)                    # Load the second number from unsorted
                                        array to t5
```

```
        bgt t3, t5, swap          # Swap if t3>t5
        addi t0, t0, 1            # Increment index to move through the array
        addi t2, t2, 1            # Increment index of inner FOR loop
        j innerloop               # Loop through inner FOR loop
      swap:                       # Swap function
        mv t6, t3                 # Move t3 to t6 register
        mv t3, t5                 # Move t5 to t3 register
        mv t5, t6                 # Move t6 to t5 register
        sb t3, 0(t0)              # Store t3 to current array location
        sb t5, 1(t0)              # Store t5 to adjacent array location
        addi t0, t0, 1            # Increment index to point to next array
                                  location

        addi t2, t2, 1            # Increment index of inner FOR loop
        j innerloop               # Loop through inner FOR loop
    innerend:                     # End of inner FOR loop
      la t0, Array                # Load address of array
      addi t1, t1, -1             # Decrement outer index of outer FOR loop
      j outerloop                 # Loop through outer FOR loop
    outerend:  j outerend         # End of program
```

9.2.5.4 An implementation of Selection Sort Algorithm

```
_start:
  andi t0, t0, 0                         # Address of array to be sorted
  andi t1, t1, 0                         # Number of elements in array
  andi t2, t2, 0                         # Variable to hold minimum value
                                         during comparison with array elements

  andi t3, t3, 0                         # Position of minimum value in array
  andi t4, t4, 0                         # Temporary variable
  andi t5, t5, 0                         # Outer FOR loop Counter i
  andi t6, t6, 0                         # Inner FOR loop counter j

  addi t5, t5, -1                        # Initializing index i
  li t1, 6                               # Specifying number of terms in the
                                         array


  OUTER_FOR_LOOP: addi t5, t5, 1         # Increment index i
    bgt t5, t1, END                      # Condition to control loop
                                         iterations


    la t0, array                         # Load given array address
    add t0, t0, t5                       # Increment array index
    lb t2, 0(t0)                         # Load a term from the given array
    mv t3, t5                            # Update position of minimum value
    addi t6, t5, 1                       # Set index j for inner loop

    INNER_FOR_LOOP: bgt t6, t1, SWAP     # GoTo swap, if condition true

      IF: la t0, array                   # IF statement, load array address to
                                         t0
```

```
        add t0, t0, t6                 # Move to next term in the array
        lb t4, 0(t0)                   # Load a term from array into t4
        blt t2, t4, ELSE               # Move to statement ELSE, if
                                       condition true
        mv t2, t4                      # t2 contains minimum value
        mv t3, t6                      # t6 contains position of minimum
                                       value
        addi t6, t6, 1                 # Increment index j
        j INNER_FOR_LOOP               # Iterate inner loop
      ELSE: addi t6, t6, 1             # Increment index j
       j INNER_FOR_LOOP                # Iterate through inner loop
  SWAP: beq t3, t5, OUTER_FOR_LOOP     # GoTo outer loop, if condition true
    la t0, array                       # Load array address to t0
    add t0, t0, t5                     # Increment array index
    lb t4, 0(t0)                       # t4 - loaded with array value in
                                       position i
    sb t2, 0(t0)                       # Store t2 in location in t0
    sub t0, t0, t5
    add t0, t0, t3
    sb t4, 0(t0)                       # Store t4 in location in t0
    j OUTER_FOR_LOOP                   # Iterate outer loop
    END: la t0, array                  # Load array address into t0
.data
  array:  .byte 9,2,3,5,11,1,4         # Array for selection
```

9.2.5.5 An implementation of Insertion Sort Algorithm

```
_start:
                                       # Initializing registers
  mv t0, x0
  mv t1, x0
  mv t2, x0
  mv t3, x0
  mv t4, x0
  mv t5, x0
  mv t6, x0

For_Loop:  la t0, nums_size            # Load t0 with unsorted array size
  lw t1, 0(t0)                         # Load t1 with value in 0 offset of
                                       t0
  lw t2, 4(t0)                         # Load t2 with value in 4 offset of
                                       t0
  addiw t1, t1, 4                      # Add a constant value to t1
  sw t1, 0(t0)                         # Store t1 value to t0
                                       # With an offset 0 of t0
  bgt t1, t2, End                      # GoTo End if t1 value > t2 value
  la t2, nums                          # Load array address to t2
  addw t2, t2, t1                      # Add t1 with t2 and store answer in
                                       t2
```

```
  lw t3, 0(t2)                          # Load t3 with value at 0 offset of
                                        t2
  addiw t4, t1, -4                      # t4 = t1 + constant

While:  la t0, nums                     # t0 = unsorted array address
  addw t0, t0, t4                       # t0 = t0+t4
  lw t0, 0(t0)                          # Load t0 with value at 0 offset of
                                        t0
  sgt t1, t0, t3                        # t1 = 1, if t0>t3
  mv t6, x0                             # Clear t6
  addi t6, t6, -1                       # t6 = t6-1
  sgt t5, t4, t6                        # t5 = 1, if t4>t6
  and t5, t1, t5                        # t5 = (t1 & t5)
  beqz t5, While_End                    # GoTo While_End if t5 = NULL)
  la t2, nums                           # t2 = unsorted array address
  mv t6, x0                             # Clear t6
  addiw t6, t4, 4                       # t6 = t4+4
  addw t2, t2, t6                       # t2 = t2+t6
  sw t0, 0(t2)                          # Store t0 to 0 offset of t2
  addiw t4, t4, -4                      # t4 = t4+constant
  j While                               # GoTo While

While_End:  addiw t4, t4, 4             # t4 = t4+4
  la t2, nums                           # t2 = unsorted array address
  addw t2, t2, t4                       # t2 = t2+t4
  sw t3, 0(t2)                          # Store t3 to 0 offset of t2
  j For_Loop                            # GoTo For_Loop

End:  la t0, nums                       # Load sorted array address to t0
                                        # Load each value into individual
                                        register to view the sorted array

lw t1, 0(t0)
lw t2, 4(t0)
lw t3, 8(t0)
lw t4, 12(t0)
lw t5, 16(t0)
lw t6, 20(t0)
lw s2, 24(t0)
lw s3, 28(t0)
lw s4, 32(t0)
lw s5, 36(t0)
```

9.2.5.6 Implementation of Binary Search Algorithm

```
_start:
.data
 Array:  .byte 1,2,3,4,5,6,7,8,9,10
  .text
    andi t0, t0, 0                           # Holds sorted Array
    andi t1, t1, 0                           # Holds the 'low' value
```

```
andi t2, t2, 0                  # Holds the 'high' value
andi t3, t3, 0                  # Holds the 'mid' value
andi t4, t4, 0                  # Holds the 'key' to be searched
andi t5, t5, 0                  # Holds the index in which the key
                                resides
andi t6, t6, 0                  # Holds the value to find mid value
                                in the array
li t1, 0                        # Low Value
li t2, 9                        # High Value
li t3, 0                        # Mid Value
li t4, 1                        # Key = 1
li t6, 2

IF: bgt t1, t2, END

ELSE:
   add t3, t1, t2
   div t3, t3, t6
   la t0, Array
   add t0, t0, t3
   lb t0, 0(t0)
   find_key_if:
      bne t4, t0, find_key_if_else
      j END

   find_key_if_else:
      bgt t4, t0, find_key_else
      addi t2, t3, -1
      j ELSE

   find_key_else:
      add t1, t3, 1
      j ELSE                    # Loop to Else

END: j END                      # Register t3 will hold the index
                                which contains the key
```

9.2.5.7 Computing factorial of a number, WITH and WITHOUT recursion

a. Without Recursion

```
_start:
  la x5, _data1               # Load data section address to x5
  lwu a0, 0(x5)               # Load a0 with number "n" to
                              calculate its factorial
  addi a4, x0, 1              # Initialize a4 to 1, a4 will keep
                              track of the calculated factorial
  addi a5, x0, 1              # Initialize "index" a5 to 1, used in
                              FOR loop
```

```
  FOR_LOOP: bgt a5, a0, End             # GoTo "End" if "index" greater than
                                        "n"
    mul a4, a4, a5                      # Multiply a4 and a5, store answer in
                                        a4
    addi a5, a5, 1                      # Increment "index" by 1
    j FOR_LOOP                          # Iterate

  End:  mv a7, a4                       # Move computed factorial to a7 from
                                        a4
   j End

.section .data                          # Begin data section
.p2align 0x2                            # Align data section to two words
_data1:                                 # Data section label
.word 0x4                               # Number to compute factorial for
```

b. With Recursion

```
_start:
  la x5, _data1                         # Load data section address to x5
  lwu sp, 0(x5)                         # Set sp to address specified in
                                        first 4 bytes of x5
                                        # Initializing four registers to zero
  mv a0, x0
  mv a4, x0
  mv a5, x0
  mv a7, x0
  lw a0, 4(x5)                          # Load a0 with data from second 4
                                        bytes of x5
  jal ra, _fact                         # Store address of recursive function
                                        in ra
  mv a7, a0                             # Move answer from a0 to a7
  sw a7, 8(x5)                          # Store answer in third 4 byte slot
                                        of address present in x5
  ebreak                                #
  j _start                              # Loop back to start

_fact:
  addi sp, sp, -32                      # Allocate 4 locations each of size 2
                                        words
  sd ra, 24(sp)                         # Store return address(ra) to
                                        Memory[24+sp]
  sd s0, 16(sp)                         # Store contents of s0 to
                                        Memory[16+sp]
  addi s0, sp, 32                       # Making s0 as frame pointer
  mv a5, a0                             # Move a0 contents to a5
  sw a5, -20(s0)                        # Store a copy of a5 to onto stack at
                                        location = Memory[s0-20]
  beqz a5, J1                           # Branch to Function J1 if a5 is 0
  addiw a5, a5, -1                      # Decrement a5 by 1
```

```
    mv a0, a5                          # Move a5 to a0
    jal ra, _fact                      # Update return address(ra) to
                                       recursive function
    mv a4, a0                          # Move a0 temporarily to a4
    lw a5, -20(s0)                     # Load a5 with data in Memory[s0-20]
    mul a5, a5, a4                     # Multiply a5 and a4, store answer in
                                       a5
    mv a0, a5                          # Move a5 to a0, as return value
    ld ra, 24(sp)                      # Move up the stack, update return
                                       address(ra) with address stored in
                                       Memory[24+sp]
    ld s0, 16(sp)                      # Update frame pointer
    addi sp, sp, 32                    # Reduce stack height
    ret                                # Return to function

  J1:
    addi a0, x0, 1                     # Initialize a0 to 1
                                       # Prepare to pop values from
                                       stack, update respective registers
                                       accordingly and reduce stack height
    ld ra, 24(sp)
    ld s0, 16(sp)
    addi sp, sp, 32

  .section .data                       # Begin data section
  .p2align 0x2                         # Align data section to two words
  _data1:                              # Data section label
  .word 0x10011000                     # Address for initialize stack
                                       pointer to
  .word 0x4                            # Number for which factorial has to
                                       be calculated
```

9.2.5.8 Program to generate and solve various exceptions in RISC-V

a. Instruction Access Fault

```
  _start:
                                       # Shift right arithmetic immediate -
                                       Shifting X0 right by 1 bit and store it
                                       to x17
    srai x17, x0,1
    srai x12, x0,1
    srai x10, x0,1
    srai x15, x0,1
    srai x6, x0,1
                                       # Adding constant to source register and
                                       saving it in destination register
    addi x10, x10, 1
    addi x12, x10, 13
```

```
    addi x17, x10, 64
                                    # Loading constants from _data section
    la x15, _data1                  # Store _data1 location to x15
    addi x17,x0, 0x10               # Comparing register for end of loop
    addi x14,x0, 0x0                # Index
                                    # Jumping to PC+50 to cause instruction
                                    access fault
    jalr ra,50(x15)
    loop:  lw x16, 0(x15)           # Load value from x15 pointing location to
                                    x16 reg
    addi x15, x15, 0x04             # GoTo next location
    addi x14, x14, 0x04
    bne x14,x17,loop                # Check for equality
    sw x17, 0x60(x15)               # Store x17 value to x15+0x60 location
    lw x12, 0x60(x15)               # Load x15+0x60 location value to x12
    bnez x10, _start                # GoTo start of the program if x10 value is
                                    not NULL

  .p2align 0x2                      # Align data section to 8-bytes
  .section .data                    # Start of data section
  _data1:                           # Declaring data to be used in the program
  .word 7
  .word 6
```

b. Load Access Fault

```
  _start:
                                    # Shift right arithmetic immediate -
                                    Shifting X0 right by 1 bit and store it
                                    to x17
    srai x17, x0,1
    srai x12, x0,1
    srai x10, x0,1
    srai x15, x0,1
    srai x6, x0,1
                                    # Adding constant to source register and
                                    saving it in destination register
    addi x10, x10, 1
    addi x12, x10, 13
    addi x17, x10, 64
                                    # Loading constants from _data section
    la x15, _data1                  # Store _data1 location to x15
    addi x17,x0, 0x10               # Comparing register for end of loop
    addi x14,x0, 0x0                # Index
                                    # Instruction to cause load access fault
    la x13,_start
    ld x16,-16 (x13)
    loop:  lw x16, 0(x15)           # Load value from x15 pointing location to
                                    x16 register
    addi x15, x15, 0x04             # GoTo next location
```

```
  addi x14, x14, 0x04
  bne x14,x17,loop          # Check for equality
  sw x17, 0x60(x15)         # Store x17 value to x15+0x60 location
  lw x12, 0x60(x15)         # Load x15+0x60 location value to x12
  bnez x10, _start          # GoTo start of the program if x10 value is
                            not NULL

.p2align 0x2                # Align data section to 8-bytes
.section .data              # Start of data section
_data1:                     # Declaring data to be used in the program
.word 7
.word 6
```

c. Load Address Misaligned

```
_start:
                            # Shift right arithmetic immediate -
                            Shifting X0 right by 1 bit and store it
                            to x17
  srai x17, x0,1
  srai x12, x0,1
  srai x10, x0,1
  srai x15, x0,1
  srai x6, x0,1
                            # Adding constant to source register and
                            saving it in destination register
  addi x10, x10, 1
  addi x12, x10, 13
  addi x17, x10, 64
                            # Loading constants from _data section
  la x15, _data1            # Store _data1 location to x15
  addi x17,x0, 0x10         # Comparing register for end of loop
  addi x14,x0, 0x0          # Index
  loop:  lw x16, 0(x15)     # Load value from x15 pointing location to
                            x16 register
  addi x15, x15, 0x04       # GoTo next location
  addi x14, x14, 0x04
  bne x14,x17,loop          # Check for equality
  sw x17, 0x60(x15)         # Store x17 value to x15+0x60 location
  lw x12, 0x60(x15)         # Load x15+0x60 location value to x12
  bnez x10, _start          # GoTo start of the program if x10 value is
                            not NULL

                            # Load Address Misaligned error since
                            .p2align is missing
.section .data              # Start of data section
_data1:                     # Declaring data to be used in the program
.word 7
.word 6
```

d. Store Access Fault

```
_start:
                                    # Shift right arithmetic immediate -
                                    Shifting X0 right by 1 bit and store it
                                    to x17
  srai x17, x0,1
  srai x12, x0,1
  srai x10, x0,1
  srai x15, x0,1
  srai x6, x0,1
                                    # Adding constant to source register and
                                    saving it in destination register
  addi x10, x10, 1
  addi x12, x10, 13
  addi x17, x10, 64
                                    # Loading constants from _data section
  la x15, _data1                    # Store _data1 location to x15
  addi x17,x0, 0x10                 # Comparing register for end of loop
  addi x14,x0, 0x0                  # Index
                                    # Instruction to cause store access fault
  la x13,_start
  sd x17,-16 (x13)
  loop:  lw x16, 0(x15)             # Load value from x15 pointing location to
                                    x16 register
  addi x15, x15, 0x04               # GoTo next location
  addi x14, x14, 0x04
  bne x14,x17,loop                  # Check for equality
  sw x17, 0x60(x15)                 # Store x17 value to x15+0x60 location
  lw x12, 0x60(x15)                 # Load x15+0x60 location value to x12
  bnez x10, _start                  # GoTo start of the program if x10 value is
                                    not NULL

.p2align 0x2                        # Align data section to 8-bytes
.section .data                      # Start of data section
_data1:                             # Declaring data to be used in the program
.word 7
.word 6
```

e. Store Address Misaligned

```
_start:
                                    # Shift right arithmetic immediate -
                                    Shifting X0 right by 1 bit and store it
                                    to x17
  srai x17, x0,1
  srai x12, x0,1
  srai x10, x0,1
  srai x15, x0,1
```

```
    srai x6, x0,1
                                        # Adding constant to source register and
                                        saving it in destination register

    addi x10, x10, 1
    addi x12, x10, 13
    addi x17, x10, 64
                                        # Loading constants from _data section
    la x15, _data1                      # Store _data1 location to x15
    addi x17,x0, 0x10                   # Comparing register for end of loop
    addi x14,x0, 0x0                    # Index
    li x11,0x1                          # Load a constant to x11
    addi x13,x0,0xAB                    # Adding x13 value to a constant
    sd x13,0 (x15)                      # Store address misaligned when x13 value
                                        to stored to data section
    loop:  lw x16, 0(x15)               # Load value from x15 pointing location to
                                        x16 register
    addi x15, x15, 0x04                 # GoTo next location
    addi x14, x14, 0x04
    bne x14,x17,loop                    # Check for equality
    sw x17, 0x60(x15)                   # Store x17 value to x15+0x60 location
    lw x12, 0x60(x15)                   # Load x15+0x60 location value to x12
    bnez x10, _start                    # GoTo start of the program if x10 value is
                                        not NULL

                                        # Causes Store Address Misaligned error
                                        since .p2align is missing
  .section .data                        # Start of data section
  _data1:                               # Declaring data to be used in the program
  .word 7
  .word 6
```

9.2.5.9 PLIC: A simple code to illustrate the working of PLIC with UART as the peripheral

```
  #define SP_BASE_ADDR 0x10012000              # Stack pointer base
                                               address = 0x10012000
  #define UART_BASE_ADDR 0x10013000            # UART base address =
                                               0x10013000
  _start:
                                               # Initializing required
                                               registers to 0
    andi sp, sp, 0
    andi t0, t0, 0
    andi t2, t2, 0
    andi t3, t3, 0
    andi t3, t3, 0
    andi t4, t4, 0
    andi t5, t5, 0
    andi t6, t6, 0
    andi s1, s1, 0
```

```
andi s2, s2, 0
andi s3, s3, 0

li sp, SP_BASE_ADDR                 # sp ⟵ Stack pointer base address

la t0, trap_entry                   # t0 ⟵ trap entry address
csrw mtvec, t0                      # mtvec ⟵ t0
li t2, UART_BASE_ADDR               # t2 ⟵ UART base address

uart_init:  lb t1, 12(t2)           # Initialize UART
                                    # Load 12th byte of t2 to t1
                                    # t1 ⟵ 12(t2)
  andi t1, t1, 0x2                  # Initialize t1 to Hex 2 value
                                    # t1 ⟵ 0x2

  bnez t1, uart_init                # If t1 ≠ 0, GoTo uart_init
  andi t1, t1, 0                    # Clear t1
  addi t1, t1, 65                   # t1 ⟵ t1+65
                                    # Value 65 is ASCII for 10 for UART

  sb t1, 4(t2)                      # Store 4th byte of t2 to t1
                                    # t1 ⟶ 4(t2)
  jal ra, interrupt                 # GoTo label "interrupt"
                                    # ra ⟵ "interrupt" address
loop:  j loop                       # Infinite loop

interrupt:  li t0, 8                # t0 ⟵ 8
  csrrs x0, mstatus, t0             # mstatus ⟵ t0
  li t0, 0x800                      # t0 ⟵ 0x800
  csrrs x0, mie, t0                 # mie ⟵ t0
  csrr s8, mstatus                  # mstatus ⟵ s8
  andi t1, s8, 8                    # t1 ⟵ (s8 ∧ 8)
  bnez t1, uart_base_addr           # If t1 ≠ 0, GoTo uart_base_addr

begin:
andi t5, t5, 0                      # Clear t5
                                    # t5 ⟵ (t5 ∧ 0)
andi t6, t6, 0                      # Clear t6
                                    # t6 ⟵ (t6 ∧ 0)
addi t5, t5, 96                     # t5 ⟵ (t5+96)
andi t4, t4, 0                      # Clear t4
                                    # t4 ⟵ (t4 ∧ 0)
addi t4, t4, 2                      # t4 ⟵ (t4+2)

PLIC: li t3, 0x0C000000             # PLIC base address
                                    # t3 ⟵ 0x0C000000
  add t3,t3, t6                     # t3 ⟵ t3+t6
  sw t4, 0(t3)                      # Store-word t4 to first word-segment of t3
                                    # t4 ⟶ 0(t3)

  addi t6, t6, 4                    # t6 ⟵ t6+4
```

```
  bge t5, t6, PLIC                  # If t5 > t6 GoTo PLIC
  andi t4, t4, 0                    # Clear t4
  addi t4, t4, 0xff                 # t4 <— t4+0xff
                                    # Setting priority to 7 (highest)
                                    for all peripherals

  li t3, 0x0C002000
  sb t4, 0(t3)
  li t3, 0x0C002001
  sb t4, 0(t3)
  li t3, 0x0C002002
  sb t4, 0(t3)
  li t3, 0x0C002003
  sb t4, 0(t3)
  li t3, 0x0c010000
  li t4, 0x1
  sb t4, 0(t3)
  ret

.p2align 2
trap_handler:  li s3, 0x0c010010
  csrr t0, mcause
  li t3, 0x10010000
  and t0,t0,t3
  beqz t0, exception_handler
  beq t0, t3, interrupt_handler
  1:  ret

.p2align 2
exception_handler:  csrr t0, mcause
  la t1, _data1
  lw t2, 0(t1)
  addi t2, t2, 4
  sw t2, 0(t1)
  add t1, t1, t2
  sw t0, 0(t1)
  j 1b

                                    # Taking back-up of all registers
                                    onto the stack
.p2align 2
trap_entry:
addi sp, sp, -32*8
nop
sd x1, 1*8(sp)
sd x2, 2*8(sp)
sd x3, 3*8(sp)
sd x4, 4*8(sp)
sd x5, 5*8(sp)
sd x6, 6*8(sp)
sd x7, 7*8(sp)
sd x8, 8*8(sp)
```

```
sd x9, 9*8(sp)
sd x10, 10*8(sp)
sd x11, 11*8(sp)
sd x12, 12*8(sp)
sd x13, 13*8(sp)
sd x14, 14*8(sp)
sd x15, 15*8(sp)
sd x16, 16*8(sp)
sd x17, 17*8(sp)
sd x18, 18*8(sp)
sd x19, 19*8(sp)
sd x20, 20*8(sp)
sd x21, 21*8(sp)
sd x22, 22*8(sp)
sd x23, 23*8(sp)
sd x24, 24*8(sp)
sd x25, 25*8(sp)
sd x26, 26*8(sp)
sd x27, 27*8(sp)
sd x28, 28*8(sp)
sd x29, 29*8(sp)
sd x30, 30*8(sp)
sd x31, 31*8(sp)
jal trap_handler                    # Return here after handling trap

ld x1, 1*8(sp)
ld x2, 2*8(sp)
ld x3, 3*8(sp)
ld x4, 4*8(sp)
ld x5, 5*8(sp)
ld x6, 6*8(sp)
ld x7, 7*8(sp)
ld x8, 8*8(sp)
ld x9, 9*8(sp)
ld x10, 10*8(sp)
ld x11, 11*8(sp)
ld x12, 12*8(sp)
ld x13, 13*8(sp)
ld x14, 14*8(sp)
ld x15, 15*8(sp)
ld x16, 16*8(sp)
ld x17, 17*8(sp)
ld x18, 18*8(sp)
ld x19, 19*8(sp)
ld x20, 20*8(sp)
ld x21, 21*8(sp)
ld x22, 22*8(sp)
ld x23, 23*8(sp)
ld x24, 24*8(sp)
ld x25, 25*8(sp)
ld x26, 26*8(sp)
```

```
ld x27, 27*8(sp)
ld x28, 28*8(sp)
ld x29, 29*8(sp)
ld x30, 30*8(sp)
ld x31, 31*8(sp)
mret

isr_handler:  li t3, 0x0C001010          # Setting interrupt for UART as the peripheral

lw t4, 0(t3)                             # Load first word of t3 to t4
                                         # t4 <--- 0(t3)
li s2, UART_BASE_ADDR                    # Load s2 with UART base address
                                         # s2 <--- 0x10013000

uart:  lb s1, 12(s2)                     # Load UART status to s1
                                         # s1 <--- 12(s2)
  andi s1, s1, 0x2                       # s1 <--- (s1 ∧ 0x2)
  bnez s1, uart                          # Wait for interrupt
  andi s1, s1, 0                         # Clear s1
  add s1, s1, t4                         # s1 <--- s1+t4
  sb s1, 4(s2)                           # Store-byte s1 to 4th byte of s2
                                         # s1 ---> 4(s2)
  sw t4, 0(t3)                           # Store-word t4 to first word segment of t3
                                         # t4 ---> 0(t3)

  ebreak

uart_base_addr:  li s2, UART_BASE_ADDR   # s2 <--- 0x10013000

                                         # Check UART status and handle as before

uart_check:  lb s1, 12(s2)
  andi s1, s1, 0x2
  bnez s1, uart_check
  andi s1, s1, 0
  addi s1, s1, 66
  sb s1, 4(s2)
  j begin

.p2align 0x2
.section .data
_data1:
.word 0
.word 0
.word 0
.word 0
```