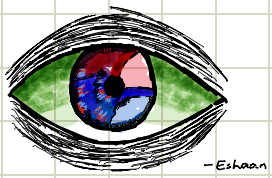
U3 - Interrupt and Arithmetic for Computer

Data Hazards

→ When an instruction depends on result of previous instruction that hasn't completed it's pipeline journey yet.

→ So basically, data hazards occur from dependence of one instruction on an earlier one that is still in pipeline (Not available)

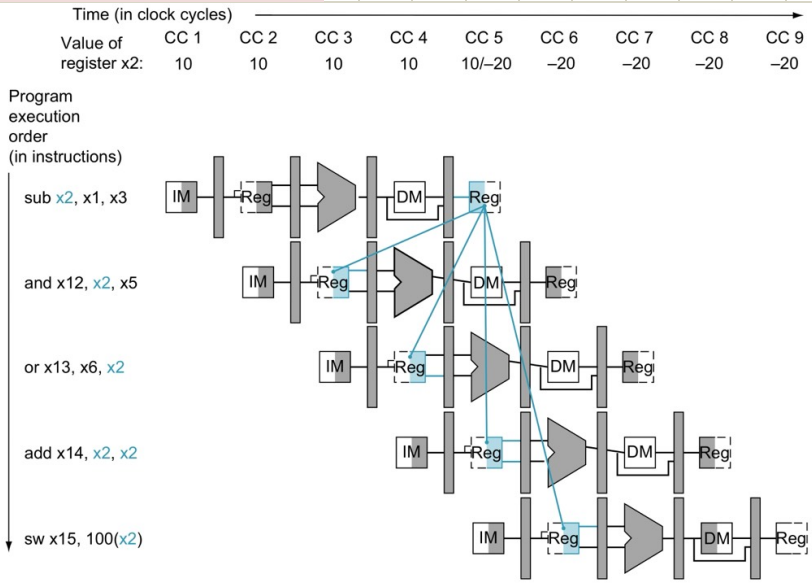


ex -

```
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sw x15, 100(x2)
```

In this example, last 4 instructions depend on resultant of x2 from 1st instruction

↳ There is a RAW (Result After Write) Dependancy



(CC: Clock Cycle)

Here, x2 is updating in CC5 but other instructions are using the value of x2 in CC3 & CC4 which result in Data Hazard

The hazard can be resolved by designing the register file hardware such that after value updates in first half of CC, it can be read in second half.

→ We can prevent all this if there was some stalling such that it can wait for the instruction to complete & then that new updated value is taken

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
SUB	IF	ID	EX	MEM	WB			
AND		IF	Stall	Stall	ID	EX	MEM	WB
OR					IF	ID	EX	MEM
ADD						IF	ID	EX
SW							IF	ID

For the same example, Now 2 stalls are added, so that and instruction can execute after completing WB in sub

Q.

```
add x15, x12, x11
lw x13, 8(x15)
lw x12, 0(x2)
or x13, x15, x13
sw x13, 0(x15)
```

i) Insert NOPs to ensure correct execution

ii) If there is forwarding unit & no hazard detection unit, Does the code require stalls How many Clock Cycles to execute the program

A. i)

```
add x15, x12, x11
nop
nop
lw x13, 8(x15)
lw x12, 0(x2)
nop
or x13, x15, x13
nop
nop
sw x13, 0(x15)
```

ii) None because 2nd lw instruction doesn't have data dependancy

. . . . .

. . . . .

. . . . .

. . . . .

. . . . .

1 2 3 4 5 6 7 8 9

No. of Cycles = 5 + 4 = 9

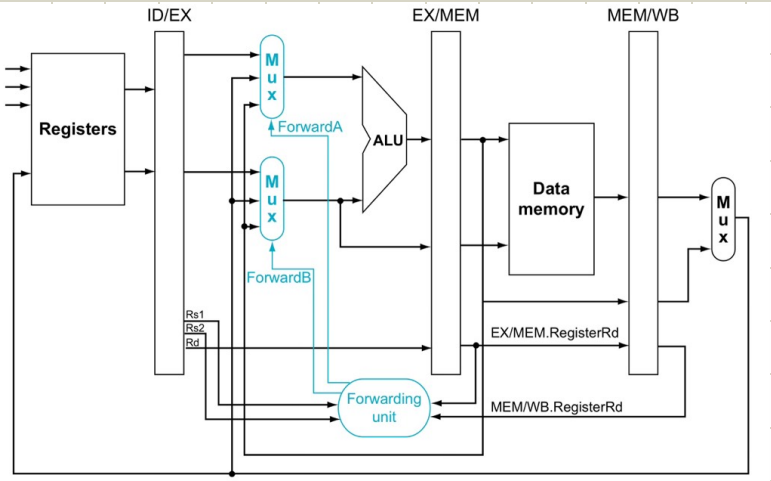
Forwarding

→ From our previous example, and & or instructions execute at CC4 & CC5 respectively without any stalls. We can also execute this segment without any stalls by forwarding the data as soon as it is available to any units that need it before it is ready from register file

→ Adding hardware to retrieve missing item early from resources is called forwarding or bypassing

↳ Most recent result is sent directly to where its needed instead of waiting for WB

→ Forwarding happens via multiplexers added before ALU inputs



→ There are 2 forwarding cases, in which either one of the pipeline registers hold the correct value before Write Back:

Hazard Type	Forward From	Condition	
EX Hazard	EX/MEM.Register Rd	Data produced in previous instruction's EX Stage	→ Forward from EX/MEM
MEM Hazard	MEM/WB.Register Rd	Data produced 2 stages earlier, now in WB Stage	→ Forward from MEM/WB

Data Hazard Detection Logic

→ The notations used are :

i) ID/EX.RegisterRs1 - Source Register 1 of instruction in EX Stage

ii) ID/EX.RegisterRs2 - Source Register 2 of instruction in EX Stage

iii) EX/MEM.RegisterRd - Destination Register of previous instruction

iv) MEM/WB.RegisterRd - Destination Register of instruction before that

a) Condition for EX Hazard (Forward from EX/MEM) (1st Priority)

→ if (EX/MEM.RegWrite == 1) // If we are writing to a Register
and (EX/MEM.RegisterRd != 0) // If destination register isn't x0
and (EX/MEM.RegisterRd == ID/EX.RegisterRs1) // And if dest.reg = reg 1
then ForwardA = 10 // Then forward

→ if (EX/MEM.RegWrite == 1)
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs2)
then ForwardB = 10

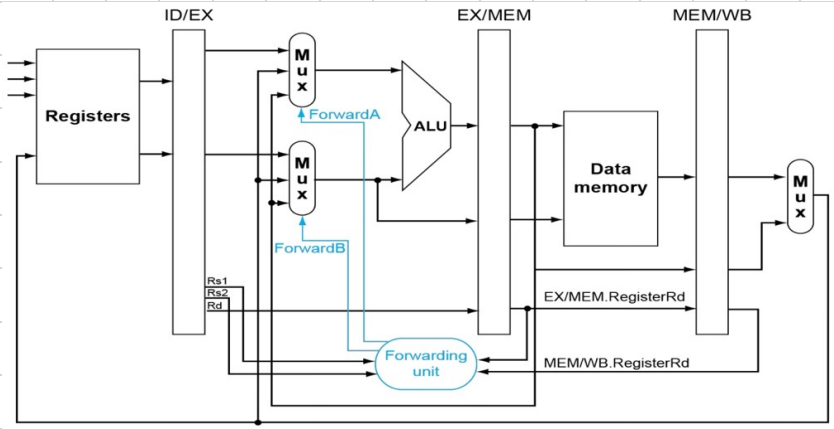
Note
RegWrite ensures we only forward if the instruction actually writes a register
!= avoids forwarding to x0

b) Condition for MEM Hazard (Forward from MEM/WB) (2nd Priority)

→ if (MEM/WB.RegWrite == 1)
and (MEM/WB.RegisterRd != 0)
and not ((EX/MEM.RegWrite == 1) and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd == ID/EX.RegisterRs1)
then ForwardA = 01

We add this to ensure it is NOT EX Hazard

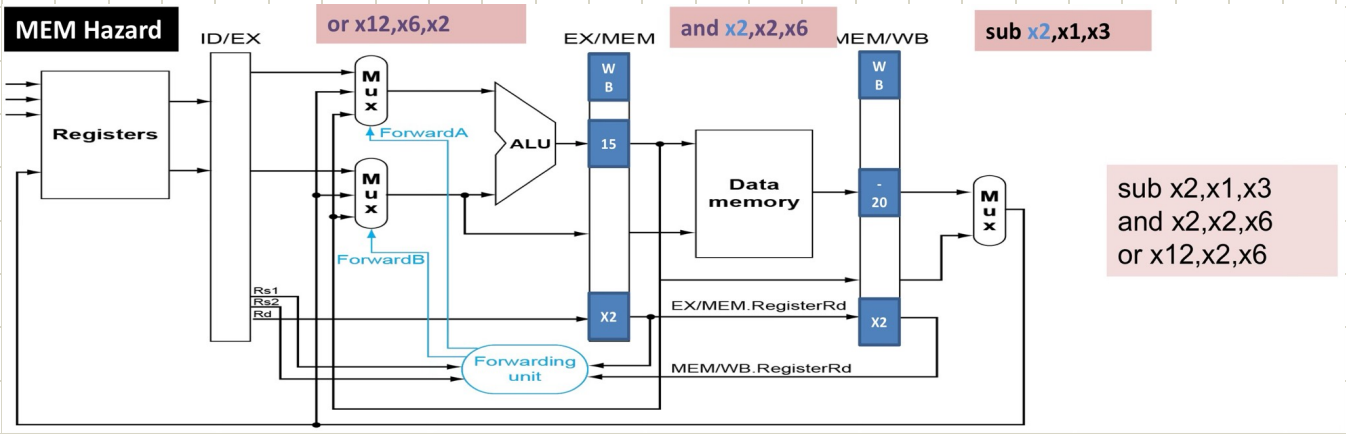
→ if (MEM/WB.RegWrite == 1)
and (MEM/WB.RegisterRd != 0)
and not ((EX/MEM.RegWrite == 1) and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd == ID/EX.RegisterRs2)
then ForwardB = 01



→ Now we look at a case when both hazards occur (both EX and MEM hazards might match)
This is also called double hazard

ex: sub x2, x1, x3
and x2, x2, x6
or x12, x2, x6

Here, EX/MEM.Rd = x2 & MEM/WB.Rd = x2
We should always use the most recent value which is EX/MEM
Hence, we add the extra condition: Only forward from MEM/WB if EX/MEM hazard isn't true



→ Now we must also understand how it looks in the datapath,
→ 2 New MUXes are added before ALU inputs (one for A, other for B)
→ These 2 MUXes select among:
i) 00 - Normal Register value from ID/EX
ii) 10 - Forwarded value from EX/MEM
iii) 01 - Forwarded value from MEM/WB

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

→ The case that forwarding can't fix - Load-Use Hazard ★
Forwarding works only when value is produced in EX or MEM stage before it's needed
BUT in load instruction, data is read from memory only during MEM stage which is 1 cycle late
ex: lw x2, 20(x1) (lw gets data from memory in cycle 4 (MEM))
and x4, x2, x5 (and needs it for ALU in cycle 4 (EX))
The data isn't ready yet, so forwarding can't help
Here we use stalls

Stalling

→ When the next instruction depends on a load's result, hardware must pause one cycle

→ Mechanism :

i) Detects hazard during ID stage

ii) Hold the PC and IF/ID register, basically freezing them

iii) Converts the ID/EX stage register into a bubble (sets all control signals to 0)

(The bubble moves down the pipeline, doing nothing (NOP) while data becomes ready

→ If there was a dependancy b/w load and subsequent instruction, even in presence of forwarding, a stall needs to be introduced

→ Condition to detect a load-use hazard :

```
if (ID/EX.MemRead == 1 and
    ((ID/EX.RegisterRd == IF/ID.RegisterRs1) or
     (ID/EX.RegisterRd == IF/ID.RegisterRs2)))
  then stall the pipeline
```

If the instruction in EX (ID/EX) is a load (MemRead = 1) and
Destination Register = Either Source Register of current instruction
Then bubble must be inserted

→ During the stall,

PC Write = 0

IF/ID Write = 0

ID/EX control lines = 0

And on the next cycle, pipeline resumes & load data is now available in MEM/WB and can be forwarded to EX

(For simplicity, assume it like a washing machine where all the clothes are taken out and let to run without anything to wash, representing NOPs)

→ Now let's consider with forwarding + hazard detection together,

i) Normal ALU-ALU dependencies which are resolved by forwarding (No stall)

ii) Load-use dependancies resolved by 1-cycle stall

iii) Independant instructions flow freely

Which means performance is close to ideal sampling

→ Special case : Forwarding for Load → Store

ex: lw x10, 0(x11)

sw x10, 0(x12)

Here data has just been loaded & must be immediately used somewhere else

Since store uses the data only in MEM stage, we can forward from MEM/WB (load) → EX/MEM (store) directly

So, a forwarding unit has to be designed b/w MEM and WB to decide on forwarding result produced by load if there was dependancy

→ In summary,

	Type	Example	Hazard Location	Solution	How?
i)	EX Hazard	add → sub	EX/MEM → EX	Forwarding	Forward from EX/MEM
ii)	MEM Hazard	add → or	MEM/WB → EX	Forwarding	Forward from MEM/WB
iii)	Load-Use Hazard	lw → and	MEM → EX	Stall	1-cycle bubble
iv)	Load → Store	lw → sw	MEM/WB → MEM	Forwarding	Forward from MEM/WB

→ Hardware Units Added,

i) Forwarding Unit

→ Inputs : Register numbers from ID/EX, EX/MEM, MEM/WB

→ Outputs : Control lines ForwardA and ForwardB for ALU MUXes

→ Location : in EX Stage

ii) Hazard Detection Unit

→ Inputs : MemRead (from ID/EX), Register Numbers (from IF/ID, ID/EX)

→ Outputs : Stall Signals (PCWrite, IF/IDWrite, Control = 0)

→ Location : in ID Stage

Both are purely combinational units & don't store state

→ Along with Forwarding & Stalling, we also have a technique known as Instruction Scheduling (Compiler Technique)
In this, if there are any dependancies, the instructions are re-ordered to naturally avoid hazards

Assume the C code ⇒ `a = b+e;` ⇒ Both depend on b
`c = b+f;`

The ALP code is as shown ⇒

```
lw   x1, 0(x31)    // load b
lw   x2, 4(x31)    // load e
add  x3, x1, x2    // b+e
sw   x3, 12(x31)   // store a
lw   x4, 8(x31)    // load f
add  x5, x1, x4    // b+f
sw   x5, 16(x31)   // store c
```

Both add instructions have a hazard because their respective dependencies are on previous lw instruction

Exceptions

→ They are unexpected events that cause an unexpected change in control flow

ex: Undefined instructions, Hardware Malfunctions, Requests from the OS, External signals (Interrupts)

Type of event	From where?	RISC-V terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Either

→ Examples

→ Exception refers to any unexpected change in control flow without distinguishing whether the cause is internal or external. They are synchronous & tied to assembly instruction

→ Interrupt only refers to external events only. They are asynchronous

→ **Privilege Modes**

→ Privilege modes determine what level of authority an instruction has when it executes

→ They provide protection b/w different components of software stack & attempts to perform the operations not permitted by current privilege mode will cause exception to be raised.

→ RISC-V supports many privilege modes but here we focus on the relavant ones:

→ In simple terms, they prevent user programs can't mess with hardware. and when something serious happens (exception/interrupt), CPU switches to S-mode/M-mode, Runs safe Kernel code and then Returns Back

i) **Machine Mode (M-mode)**

→ Highest Privilege Mode

→ Has complete access to all registers, memory & features

→ Execution & interrupt handling defaults to M-mode

→ Firmware & low-level control run here

ii) **Superviser Mode (S-Mode)**

→ Used by Operating Systems

→ Can't access certain hardware unless explicitly allowed

→ Handles page faults, system calls, timer interrupts etc.,

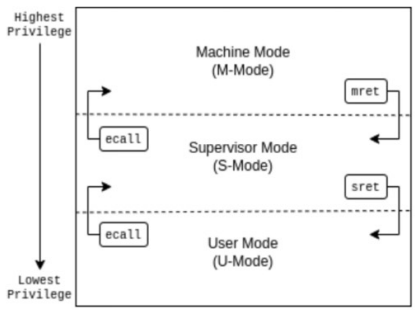
iii) **User Mode (U-Mode)**

→ Lowest privilege level

→ Applications run here

→ Can't execute privileged instructions

→ Traps move processor into supervisor/machine mode



→ Ecall is used to TRAP the processor which can be used to switch b/w privilege Models They are used to implement system call to pass/access system resources (Pass arguement b/w diff privileged modes)

→ Steps for Interrupt / Exception Handling

i) STOP normal execution

ii) SAVE where you were (into SEPC)

iii) SAVE why it happened (into SCAUSE / MCAUSE)

iv) GO TO handler code (using mvtec)

v) Handler fixes the issue or stops the program

Think of it like :

i) PAUSE Program

ii) Remember the address (SEPC)

iii) Remember the reason (SCAUSE)

iv) Jump to OS

v) OS handles it

→ Direct vs Vectored Trap Handling

→ These simply decide which address the CPU jumps to for the handler

→ Direct Mode (simplest)

→ All traps go to one address = mtvec.BASE

→ Handler must mcause/scause to know what happened

→ So like ,

Trap happened ⟶ Jump to mvtec.BASE

→ Vectored Mode

→ The trap handler address = base + (cause × 4)

Different cause has different address

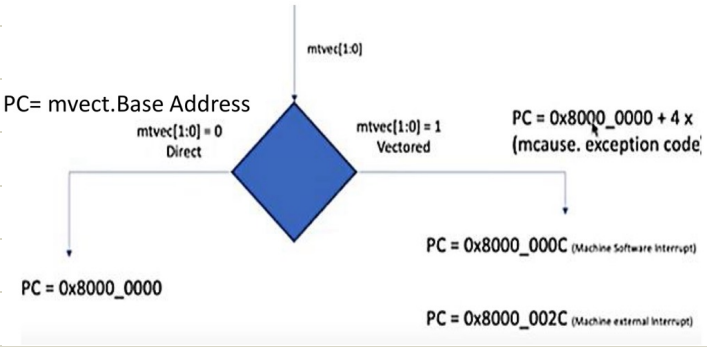
→ So think of the VECTOR TABLE like :

0x80000000 : handler for cause 0

0x80000004 : handler for cause 1

0x80000008 : handler for cause 2

→ Now the CPU doesn't have to check the cause (mcause/scause)



→ Interrupt Enable / Disable

→ Interrupts only trigger if :

Global Enable (mstatus. MIE) = 1

AND

Specific Enable (mie register bits) = 1

→ mie (Machine Interrupt Enable) register bits :

- MTIE : Enable Timer Interrupt
- MSIE : Enable Software Interrupt
- MEIE : Enable External Interrupt

→ mip (Machine Interrupt Pending) register bits :

- MTIP : Timer Interrupt Pending
- MSIP : Software Interrupt Pending
- MEIP : External Interrupt Pending

So basically if both ENABLE & PENDING = 1 , then interrupt happens.

→ Timer Interrupt

→ It is like an OS Alarm Clock

→ 2 Registers :

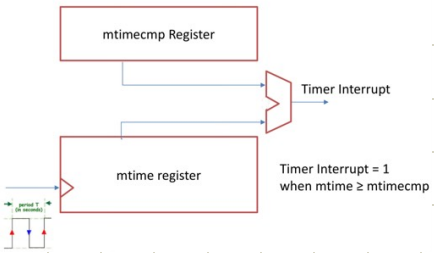
i) mtime

→ 64 bit increasing counter

→ Always increments

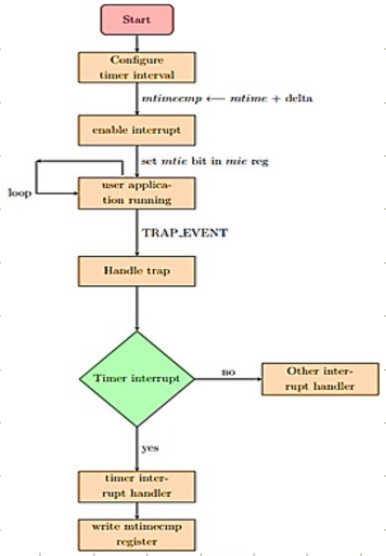
ii) mtimecmp

→ OS writes a future time here



→ Now, timer interrupt happens if mtime ≥ mtimecmp

CPU then jumps to timer handler



→ What happens in Pipeline during Execution ?

→ Pipeline has multiple instructions running at same time

→ Suppose an ADD instruction causes an exception in EX stage

Pipeline must ensure the following :

1. Instructions before ADD finish normally
2. ADD itself mustn't write any result
3. Instructions after ADD must be flushed (turned into NOP)
4. Save PC of ADD into SEPC
5. Save cause into SCAUSE
6. Jump to handler

→ This behavior is known as Precise exception because exception is attributed to correct instruction

→ Multiple Exceptions

→ If multiple instructions have exceptions at same time, then the standard approach is to handle the earliest instruction & flush all the instructions after it.

→ This makes behaviour predictable

In summary,

```
EXCEPTION = internal event (illegal instruction, misaligned, syscall)
INTERRUPT = external event (timer, external device)

Modes: U-mode → S-mode → M-mode (increasing privilege)

Trap Handling Steps:
1. Save PC to SEPC
2. Save cause to SCAUSE/MCAUSE
3. Jump to mtvec.BASE (Direct)
   or BASE + 4*cause (Vectored)
4. OS handler runs
5. Return with SRET

Interrupt Enable:
mstatus.MIE = 1 AND mie.MTIE/MSIE/MEIE = 1

Timer Interrupt:
mtime increments
if mtime >= mtimecmp → interrupt

Pipeline Exceptions:
Finish older instructions
Kill offending instruction result (don't write it)
Flush younger instructions
Go to handler
```

2. Consider the program given executed on a 5-stage pipelined architecture

Q.

```
40hex  sub   x11, x2, x4
44hex  and   x12, x2, x5
48hex  or    x13, x2, x6
4Chex  add   x1,  x2, x1
50hex  sub   x15, x6, x7
54hex  lw    x16, 100(x7)
. . .
```

assume the instructions to be invoked on an exception begin like this

```
1C090000hex  sw   x26, 1000(x10)
1C090004hex  sw   x27, 1008(x10)
. . .
```

Answer the following assuming a hardware malfunction exception has been invoked when the add instruction is in EX stage of the pipeline

a) What will be the content of SEPC
b) What will be the status of Interrupt bit in SCAUSE?
c) Where is the control of execution transferred?
d) Which instructions in the pipeline get flushed & gets executed?

A. a) SEPC = 0x4C (contains offending instruction)

b) Interrupt bit = 0 (Because this is an exception, not interrupt)

c) PC is transferred to 0x1C090000 (the TRAP handler base address)

OS exception routine begins here

d) Normal execution : sub @ 40 , and @ 44 , or @ 48

Flushed : add @ 4C , sub @ 50 , lw @ 54

Executed next : instructions @ 1C090000 & 1C090004 (Handler code)

Q. Explain the steps involved in handling the TRAP using the Vector Table approach.

A.

⭐ Vector Table TRAP Handling Steps

1. **Trap occurs** (exception/interrupt detected)
2. Hardware sets:
   - **SEPC = PC of faulting instruction**
   - **SCAUSE/MCAUSE = cause code (interrupt bit + code)**
3. Determine trap address:

```
PC = mtvec.BASE + 4 × cause_code
```

4. Control jumps to this entry in the vector table.
5. Handler at that vector entry:
   - Saves registers if needed
   - Fixes the cause OR terminates the program
6. If restartable:
   - Handler uses **SEPC** to return to the faulting instruction
7. Use **SRET** to return to normal execution.

Q. Explain with a flow chart how a timer can be used in Interrupt mode to generate the time delay.

A timer interrupt happens when:

```
mtime ≥ mtimecmp
```

Here is the **simple flow**:

⭐ Flowchart Steps

1. **Initialize timer**
   - Write current time (or future target time) to `mtimecmp`
2. **Enable timer interrupt**
   - Set `mie.MTIE = 1`
   - Set `mstatus.MIE = 1`
3. **Normal program executes**
4. **Hardware increments mtime continuously**
5. **If mtime ≥ mtimecmp → MTIP = 1**
6. **Interrupt check**

```
If (MIE=1) and (MTIE=1) and (MTIP=1)
   → Timer Interrupt occurs
```

7. **Trap to handler**
   - Save SEPC
   - Save SCAUSE
   - Jump to mtvec (or vector entry)
8. **Timer Handler runs**
   - Perform required delay tasks
   - Set next mtimecmp = mtime + delay
9. **Return to main program**
   - via MRET/SRET

MCQs

Q. 5. Assume there is an illegal instruction getting executed, what value will be updated in the **scause** register?
Note: Use the table given

Interrupt bit	Exception Code	
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
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt

a. Interrupt bit =1 and Exception Code=2
b. ✓ Interrupt bit =0 and Exception Code=2
c. Interrupt bit =0 and Exception Code=1
d. Interrupt bit =1 and Exception Code=4

Q. 6. If the **scause** MSB is 1 what is the cause for the trap?
a. ✓ Interrupt
b. Exception
c. Both
d. None of these

Q. 7. Which of the following is not an Exception?
a. System Reset
b. Invoking the OS from use program
c. Using undefined Instruction
d. ✓ I/O device request

Q. 8. Which of the following is either an Exception/ Interrupt?
a. I/O device request
b. Using undefined Instruction
c. I/O device request
d. ✓ Hardware Malfunctions

Q. 9. Interrupts are unexpected events that occur _____ any of the RISC-V harts.
a. Asynchronously Inside
b. Synchronously outside
c. ✓ Asynchronously outside
d. None of these

Q. 10. Which of the following modes has the highest privilege?
a. Supervisor mode (S-Mode)
b. ✓ Machine mode (M-Mode)
c. User mode (U-Mode)
d. All of them

Q. 11. Which of the following modes is a must to have in a RISC-V SoC?
a. Supervisor mode (S-Mode)
b. ✓ Machine mode (M-Mode)
c. User mode (U-Mode)
d. All of them

Q. 12. ecall is an instruction used to _____
a. ✓ switch between privilege Modes
b. transfer control to debugger
c. transfer control to subroutine
d. None of these

Q. 13. In the case of the Vectored Mode of TRAP Handling, control of execution transfer to the handler by updating PC with _____
a. ✓ VT_BaseAddress + 4 x Exception Code from scause
b. VT_BaseAddress
c. VT_BaseAddress + Exception Code from scause
d. None of these

Q. 14. In which of the following TRAP handlers the SEPC is copied into PC to restart the TRAPPED program
a. ✓ User timer Interrupt TRAP handler
b. Illegal Instruction TRAP handler
c. Instruction addresses misaligned TRAP handler
d. None of these

Q. 15. Which Timer register provides the current real-time in ticks?
a. mtimecmp
b. ✓ mtime
c. Both mtime and mtimecmp
d. None of these

Q. 16. Which Timer register is used to hold the Initial count as reference w.r.t time period to generated?
a. ✓ mtimecmp
b. mtime
c. Both mtime and mtimecmp
d. None of these

Q. 17. Under which of the following conditions the timer Interrupt is invoked?
a. mtime<mtimecmp
b. ✓ mtime>=mtime+delta
c. mtime=stime
d. None of these

Q. 18. mie register is used for _____
a. holding the pending exception to handled
b. ✓ enable/ disable the unexpected events using corresponding bits
c. to globally enable/ disable the unexpected events using a single bit.
d. None of these.

Q. 19. Which of the following privileged modes **has virtually no restrictions**?
a. ✓ Machine
b. Supervisor
c. User
d. None of these

Q. 20. Assume a RISC-V core running in User privilege mode. Which is a TRUE statement
a. ✓ A process running in User mode cannot modify any of the control and status registers
b. A process running in User mode can modify any of the control and status registers
c. A process running in User mode can modify only supervisor-level control and status registers
d. A process running in User mode can modify only machine-level control and status registers

Multiplication

→ In addition, 2 n-bit number's sum is usually n-bit result

But in multiplication, 2 n-bit number's product is usually 2n-bit result

So, 32 bit × 32 bit = 64 bit product

→ Hardware must also generate partial products, shift appropriately, accumulate results & handle unsigned/signed no.

Basically multiplication = repeated addition + shifting

Binary Multiplication

→ It follows similar principle to multiplication,

ex :

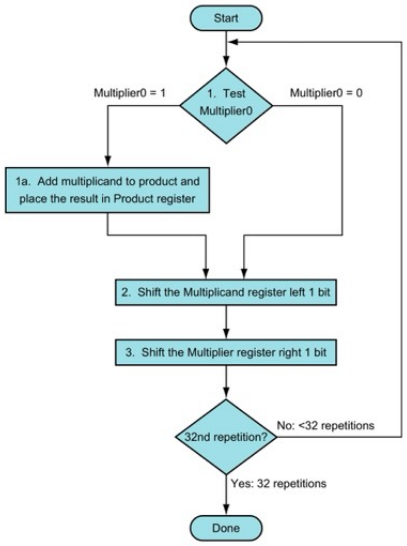
```
  1001   (9)   M
X 110    (6)   Q
-----
  0000
 1001 x
1001 xx
-------
1 1011 0 (54)  A
```

→ From the above example we can see that if multiplier bit is :

i) 1 → Add shifted multiplicand

ii) 0 → Add 0

Using this, lets build a Shift-and-Add Multiplier Algorithm



→ Assume the following registers,

A : Accumulator (initially 0)

Q : Multiplier

M : Multiplicand

Q(-1) : A single extra bit

n : no. of bits

Algorithm → Repeat n times :

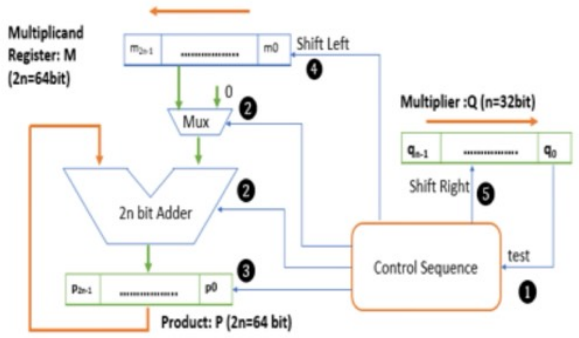
1. if Q0 = 1 → A ← A+M
2. Shift A, Q, Q(-1) right by 1 bit

Final Product = concatenation of A || Q

For the same example 1001 × 110, Take 1001 as multiplicand & 110 as multiplier

Iteration	Multiplier	Steps	Multiplicand	Product
0	11(0)	Initialization	00001001	00000000
1		a) Test bit = 0 ; Product = Product (No operation)		00000000
		b) Shift Left Multiplicand	00010010	
	01(1)	c) Shift Right Multiplier		
2		a) Test bit = 1 ; Product = Product + Multiplicand		00010010
		b) Shift Left Multiplicand	00100100	
	00(1)	c) Shift Right Multiplier		
3		a) Test bit = 1 ; Product = Product + Multiplicand		00110110
		b) Shift Left Multiplicand	01001000	
	000	c) Shift Right Multiplier		

→



→ The multiplicand, ALU and Product Register are 64 bits Only multiplier is 32 bits wide

→ 32-bit multiplicand starts at right half of multiplicand register & shifted left 1 bit on each step

→ Multiplier is shifted right at each step (opposite). Algorithm starts with product initialized to 0

→ Control decides when to shift Multiplicand & Multiplier and when to write new values to product register

Refined / Faster Version of Sequential Circuit Multiplier

→ To cut down on the resources, the circuit performs multiplication using

i) Single n-bit adder (instead of 2n-bit adder)

ii) Carry (1 bit) + Accumulator (n bit) + Multiplier (n bit) = Register (2n+1 bit)

iii) Multiplicand remains the same

ex : M = 1101 × Q = 1011

Initially C & A are 0

So,

C	A	Q	M = 1101
0	0000	1011	Initialise
	0000		Here Test bit = 1
	+ 1101		→ Add M to A, then put Q in right half of register
0	1101	1011	
0	0110	1101 SR	Finally Shift Right
	0110		Again Test bit = 1
	+ 1101		→ Add M to A, then put Q in right half of register
1	0011	1101	
0	1001	1110 SR	Finally Shift Right
	1001		Here Test bit = 0
	+ 0000		No Add
0	1001	1110	
0	0100	1111 SR	Only Shift Right
	0100		
	+ 1101		
1	0001	1111	
0	1000	1111	= Final Answer

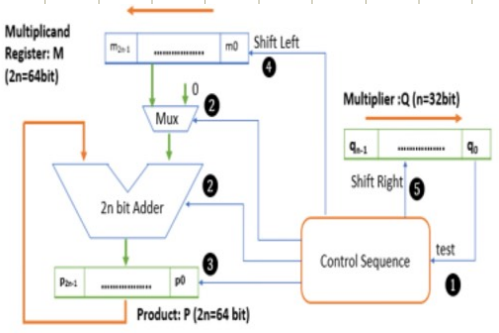
→ Comparing with first version,

i) Multiplicand register & ALU reduced to 32 bits

ii) Product is shifted right

iii) Seperate Multiplier register disappeared. Instead it is placed on right half of new register

iv) Product register is grown by 1 bit to 33 bit



Now for the same example, let's construct a table similar to one in simple multiplication

Iteration	Steps	Multiplicand	Product / Multiplier
0	Initialization	1101	0 0000 1011
1	a) LSB = 1, Prod = Prod + Mcand	1101	0 1101 1011
	b) Rshift Prod / Multiplier	1101	0 0110 1101
2	a) LSB = 1, Prod = Prod + Mcand	1101	1 0011 1101
	b) Rshift Prod / Multiplier	1101	0 1001 1110
3	a) LSB = 0, No OP	1101	0 1001 1110
	b) Rshift Prod / Multiplier	1101	0 0100 1111
4	a) LSB = 1, Prod = Prod + Mcand	1101	1 0001 1111
	b) Rshift Prod / Multiplier	1101	0 1000 1111

= Final Product

Q. **3.14** [10] <§3.3> Calculate the time necessary to perform a multiply using the approach given in Figures 3.3 and 3.4 if an integer is 8 bits wide and each step of the operation takes four time units. Assume that in step 1a an addition is always performed—either the multiplicand will be added, or a zero will be. Also assume that the registers have already been initialized (you are just counting how long it takes to do the multiplication loop itself). If this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously. If this is being done in software, they will have to be done one after the other. Solve for each case.

A. For hardware,

Add - 1 Clock

Shift - 1 Clock ⟶ (3 × 8) × 4 = 96 time units

Decide Looping - 1 Clock

For software,

Deciding what to add - 1 Clock

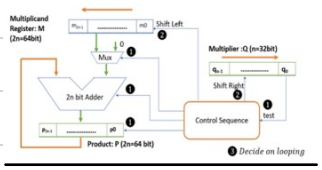
Add - 1 Clock

Shift Right - 1 Clock ⟶ (5 × 8) × 4 = 160 time units

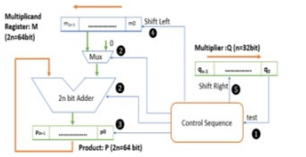
Shift Left - 1 Clock

Decide Looping - 1 Clock

Sequential
a) Hardware



Sequential
a) software



Faster Multiplication

→ Case i : Stacked Adders

Here we use 31 adders, each adding its own partial product & they are stacked vertically

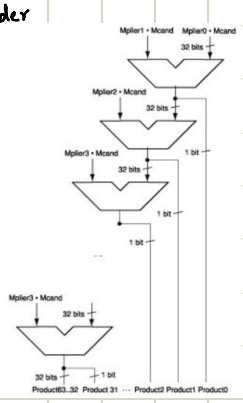
One input is multipicand ANDed with multiplier bit

other input is output of prior adder

→ For the time required,

It takes B time units to get through one adder & there will be A-1 adders, then

time req. = B × (A-1)



→ Case ii : Parallel Tree

Another way is to organize these additions in parallel tree instead of waiting for 32 add times, Now only $\log_2(32) = 5$

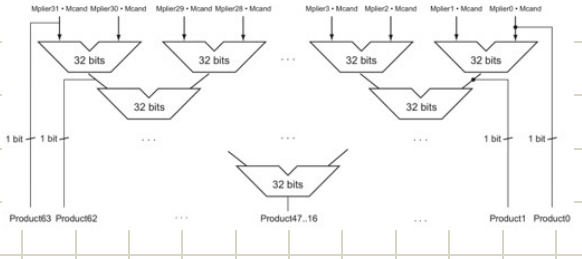
→ No. of levels = $\log_2(n)$

No. of adders in level 1 = $\frac{n}{2}$

No. of adder in subseq. level = $\frac{1}{2}$ of prev

→ Time taken, assume B units through an adder & $\log_2 A$ levels, then

time req = $B \times \log_2 A$



1) For multiplying two n-bit integers, the size of the adder required in the sequential circuit multiplier and refined (modified) sequential multiplier is _____ and _____ respectively?
   a) n bit and n nit
   b) 2n bit and 2n bit
   c) n bit and 2n bit
   d) 2n bit and n bit

2) In a refined Sequential circuit multiplier, What value is initialized in (2n+1) shift right register before the iteration starts? Assume the multiplier is 0111b and the multiplicand is 1001b
   a) 0 0000 0111
   b) 0 0000 0000
   c) 0 0000 1001
   d) 0 1001 0111

3) In the First version of the Sequential circuit multiplier, the adder used should be _____ bit size if data is of n bit size.
   a) n bit
   b) 2n bit
   c) 4n bit
   d) n/2 bit size

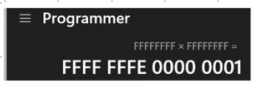
4) In the first version of the sequential circuit multiplier, Multiplicand Register=0000 0100b, Product Register = 0000 0010b, and Multiplier =0001b. What will be the state of these registers after an iteration?
   a) Multiplicand Register=0000 0100b, Product Register = 0000 0010b and Multiplier =0001b.
   b) Multiplicand Register=0000 1000b, Product Register = 0000 0110b and Multiplier =0000b.
   c) Multiplicand Register=0000 0010b, Product Register = 0000 0110b and Multiplier =0001b.
   d) None of these

5) What is the computation time taken by the Refined version of the Multiplier if this is being done in hardware the shift right and shift left required can be done simultaneously. Assume each step takes 10-time units and integers are of 32-bit size
   a) 32 x 3 x 10tu
   b) 32 x 5 x 10tu
   c) 32 x 10 tu
   d) log2(32) x 10tu

11) What is the computation time taken by the parallel tree multiplier? Assume each step takes 10-time units and integers are of 32-bit size.
   a) 32 x 10tu
   b) 31 x 10tu
   c) 5 x 10tu
   d) 6 x 10tu

M Extension

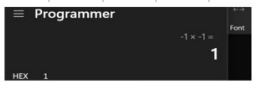
→	Multiply	mul x5, x6, x7	x5 = x6 × x7	Lower 32 bits of 64-bit product
	Multiply high	mulh x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit signed product
	Multiply high, unsigned	mulhu x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit unsigned product
	Multiply high, signed-unsigned	mulhsu x5, x6, x7	x5 = (x6 × x7) >> 32	Upper 32 bits of 64-bit signed-unsigned product



Case1: Unsigned Numbers

Source code
```
1 li x5,0xFFFFFFFF
2 li x6,0xFFFFFFFF
3 mul x7,x5,x6
4 mulhu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0xfffffffe



Case2: Signed Numbers

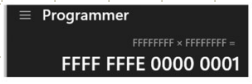
Source code
```
1 li x5,0xFFFFFFFF
2 li x6,0xFFFFFFFF
3 mul x7,x5,x6
4 mulh x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0x00000000

Case3: Signed Numbers x Unsigned Number

Source code
```
1 li x5,0xffffffff
2 li x6,0xffffffff
3 mul x7,x5,x6
4 mulhsu x8,x5,x6
```

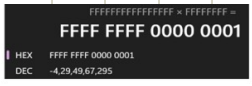
x5	t0	0xffffffff
x6	t1	0xffffffff
x7	t2	0x00000001
x8	s0	0xffffffff



Case1: Unsigned Numbers

Source code
```
1 li x5,0xffffffff
2 li x6,0x0f
3 mul x7,x5,x6
4 mulhu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0x0000000f
x7	t2	0xfffffff1
x8	s0	0x0000000e



Case2: Signed Numbers

Source code
```
1 li x5,0xffffffff
2 li x6,0x0f
3 mul x7,x5,x6
4 mulh x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0x0000000f
x7	t2	0xfffffff1
x8	s0	0xffffffff

Case3: Signed Numbers x Unsigned Number

Source code
```
1 li x5,0xffffffff
2 li x6,0x0f
3 mul x7,x5,x6
4 mulhsu x8,x5,x6
```

x5	t0	0xffffffff
x6	t1	0x0000000f
x7	t2	0xfffffff1
x8	s0	0xffffffff

Q.

13) What is the content of the register x7 and x8 after execution of the following program

Source code
```
1 li x5,0xffffffff
2 li x6,0xffffffff
3 mul x7,x5,x6
4 mulhsu x8,x5,x6
```

   a) x7= 0x0000 0001 and x8=0xFFFF FFFF
   b) x7= 0x0000 0001 and x8=0xFFFF FFFE
   c) x7= 0x0000 0001 and x8=0x0000 0000
   d) x7= 0x0000 0000 and x8=0xFFFF FFFF

Division

→ Division is slow because it :

i) is Repeated subtract + shift

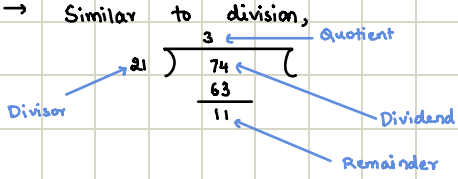
ii) Requires comparing & adjusting

iii) Needs to detect when remainder becomes negative

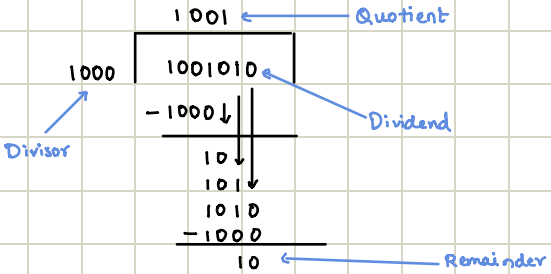
iv) Requires sign handling

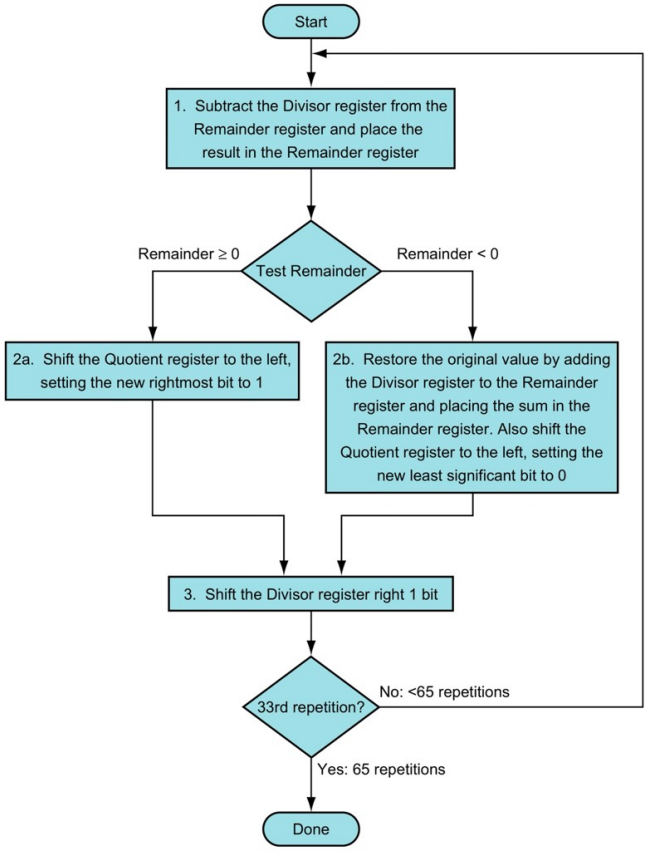
Division Principle

→ Similar to division,



Now in binary,





(i) Rem = Rem - Divisor

(ii) If Rem < 0 :

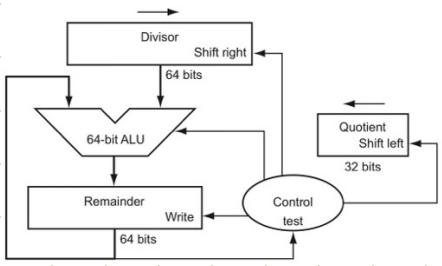
Rem = Rem + Divisor (Restore)

and shift left Quotient & $Q_0 = 0$

else :

shift left Quotient & $Q_0 = 1$

(iii) Shift Right Divisor 1 time



→ Divisor Register, ALU, Remainder Register are all 2n bit wide

Only Quotient Register is n bit

→ Control decides when to shift Divisor & Quotient registers and when to write new value into Remainder register

Now let's make a table similar to the one in multiplication

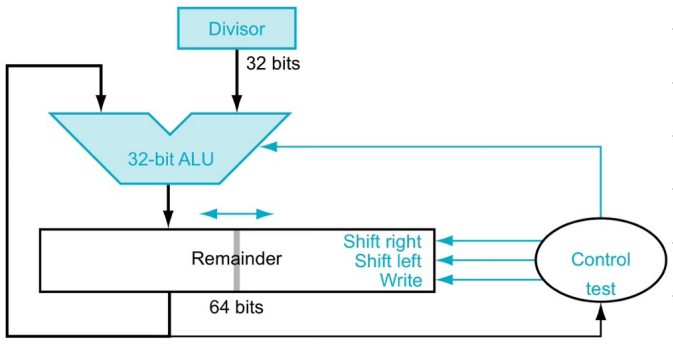
→ To divide 7_{10} by 2_{10} ⇒ 00000111_2 by 0010_2

Iteration		Steps	Quotient	Divisor	Remainder
0		Initialization	0000	0010 0000	0000 0111
1	a)	Rem = Rem - Div	0000	0010 0000	1110 0111
	b)	Rem < 0 ⇒ +Div, SLL Q, Q0 = 0	0000	0010 0000	0000 0111
	c)	Shift Div Right	0000	0001 0000	0000 0111
2	a)	Rem = Rem - Div	0000	0001 0000	1111 0111
	b)	Rem < 0 ⇒ +Div, SLL Q, Q0 = 0	0000	0001 0000	0000 0111
	c)	Shift Div Right	0000	0000 1000	0000 0111
3	a)	Rem = Rem - Div	0000	0000 1000	1111 1111
	b)	Rem < 0 ⇒ +Div, SLL Q, Q0 = 0	0000	0000 1000	0000 0111
	c)	Shift Div Right	0000	0000 0100	0000 0111
4	a)	Rem = Rem - Div	0000	0000 0100	0000 0011
	b)	Rem ≥ 0 ⇒ SLL Q, Q0 = 1	0001	0000 0100	0000 0011
	c)	Shift Div Right	0001	0000 0010	0000 0011
5	a)	Rem = Rem - Div	0001	0000 0010	0000 0001
	b)	Rem ≥ 0 ⇒ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	c)	Shift Div Right	0011	0000 0001	0000 0001

The requirement of this algorithm is it takes n+1 (here 4+1 = 5) steps to get proper quotient & remainder

Refined Version of Division Hardware

→



An the improved version,

i) Divisor Register needs to be n-bit size only

ii) ALU is of n-bit size

iii) Remainder register of 2n in which most significants n-bits initially hold zero & least significant n-bits holds the dividend.

iv) At end of n^{th} iteration, Most significant n-bits hold the remainder & least significant hold Quotient

Steps :

i) Rem = Rem << 1

ii) Rem = Rem - Div
↳ Left half of Rem

iii) If Rem < 0 : Rem + D
Shift Rem Left
Set New LSB as 0
else : Set R0 = 1

→ Advantages

i) ALU & Divisor are halved in comparision with version 1 & remainder is shifted left

ii) It also combines Quotient register with right half of the Remainder register

→ Now take 60 ÷ 17 , and build a table for this

Iteration		Steps	Divisor	Remainder / Quotient
0		Initialization	010 001	000 000 111 100
1	a)	R <<	010 001	000 001 111 000
	b)	Rem = Rem - Div	010 001	(1)10 000 111 000
	c)	Rem < 0 , Rem + Div	010 001	000 001 111 000
2	a)	R <<	010 001	000 011 110 000
	b)	Rem = Rem - Div	010 001	(1)10 010 110 000
	c)	Rem < 0 , Rem + Div	010 001	000 011 110 000
3	a)	R <<	010 001	000 111 100 000
	b)	Rem = Rem - Div	010 001	(1)10 110 100 000
	c)	Rem < 0 , Rem + Div	010 001	000 111 100 000
4	a)	R <<	010 001	001 111 000 000
	b)	Rem = Rem - Div	010 001	(1)11 110 000 000
	c)	Rem < 0 , Rem + Div	010 001	001 111 000 000
5	a)	R <<	010 001	011 110 000 000
	b)	Rem = Rem - Div	010 001	(0)01 101 000 000
	c)	Rem > 0 , R0 = 1	010 001	001 101 000 001
6	a)	R <<	010 001	011 010 000 010
	b)	Rem = Rem - Div	010 001	(0)01 001 000 010
	c)	Rem > 0 , R0 = 1	010 001	001 001 000 011

Remainder Quotient

→ M Extension

Divide	div x5, x6, x7	x5 = x6 / x7	Divide signed 32-bit numbers
Divide unsigned	divu x5, x6, x7	x5 = x6 / x7	Divide unsigned 32-bit numbers
Remainder	rem x5, x6, x7	x5 = x6 % x7	Remainder of signed 32-bit division
Remainder unsigned	remu x5, x6, x7	x5 = x6 % x7	Remainder of unsigned 32-bit division

→ Signed Division

→ Equation that holds : Dividend = Quotient × Divisor + Remainder

ex : Case i: +7/+2 ⇒ Quotient = + 3 , Remainder = +1

Case i: -7/+2 ⇒ Quotient = - 3 , Remainder = -1

Case i: +7/-2 ⇒ Quotient = - 3 , Remainder = +1

Case i: -7/-2 ⇒ Quotient = + 3 , Remainder = -1

Dividend	Divisor	Quotient	Remainder
+	+	+	+
-	+	-	-
+	-	-	+
-	-	+	-

Floating Point

→ We use floating point representation, because integers can't represent very large no.s (10^{30}) & very small fractions (10^{-30})

→ Floating point uses scientific notation : $(-1)^{sign} \times mantissa \times 2^{exponent}$

→ The most widely used floating point format in computers is IEEE 754 Single Precision (32-bit)

Sign	Exponent (8)	Fraction / Mantissa (23)

There are other standards as well

Precision	Sign	Exponent	Mantissa	Total
single	1	8	23	32
double	1	11	52	64
long double	1	15	64	80

→ In Single precision,

bit 31 : S ⟶ Sign bit

If S = 1 → Negative

If S = 0 → Positive

bit [30:23] : Biased Exponent ⟶ 8 bits allow us to represent $2^8 = 256$ different integers

We also need to represent +ve & -ve integers & 2 special cases

So we assign 254 only (-126 to +127) but this is unbiased

We also add a bias of 127, for ease of representation,

$E' = E + 127 \Rightarrow 1 \le E' \le 254$

Now we include special cases, then range is $0 \le E' \le 255$

bit [22:0] : 23 Bit Mantissa ⟶ Fractional part of significant bit

It always has a leading 1 with the binary point immediately to right

So, $V(B) = 1 + (b_{-1} \times 2^{-1}) + (b_{-2} \times 2^{-2}) + \dots + (b_{-23} \times 2^{-23})$

To pack more bits into the number, it is normalized to make the leading 1 bit of binary number implicit.

↳ For single precision, 24 bits long (1 + 23-bit fraction)

For double precision, 53 bits long (1 + 52-bit fraction)

1 + 23 → Fraction

↓

Significand

Since 0 doesn't have leading 1, it is given reserved exponent value 0, so hardware won't attach leading 1 to it.

So, value represented = $\pm 1.M \times 2^{E'-127}$

In summary ⇒ Signed bit (0 for +ve & 1 for -ve)

Exponent (biased) (Store exponent + bias ⇒ Ensures exponent is always non-negative)

Fraction/Mantissa (Only stores digits after binary point & leading 1 is hidden)

Q. **3.22** [10] <§3.5> What decimal number does the bit pattern 0 × 0C000000 represent if it is a floating point number? Use the IEEE 754 standard.

Q. **3.23** [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 single precision format.

Q. **3.24** [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 double precision format.

A. 3.22

For 0x0C000000 = 0000 1100 0000 0000 0000 0000 0000 0000

We can see 31st bit = 0 ⇒ Positive

Exponent = 00011000b

= 24d

Exponent unbiased = 24 − 127 = −103

Mantissa = 0

There is a hidden 1

So answer = $+1.0 \times 2^{-103}$

A. 3.23

Given, 63.25d ⇒ 63.25×10^0

In binary, 63 = 111111

0.25 = 0.01

So, $63.25 \times 10^0 = 111111.01 \times 2^0 = 1.1111101 \times 2^5$

Mantissa = 11111010000000000000000

Exponent (Biased) = 5 + 127 = 132 = 1000 0100

Sign = 0

Answer = 0100 0010 0111 1101 0000 0000 0000 0000

= 0x427D0000

A 3.24

Given, 63.25d ⇒ 63.25×10^0

In binary, 63 = 111111

0.25 = 0.01

So, $63.25 \times 10^0 = 111111.01 \times 2^0 = 1.1111101 \times 2^5$

Mantissa = 11111010000000000000000000000000000000000000000000000

Exponent (Biased) = 5 + 1023 = 1028

Sign = 0

Answer = 0100 0000 0100 1111 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

= 0x404FA00000000000

Overflow & Underflow

→ Overflow is a situation in which positive exponent becomes too large to fit in exponent field

→ Underflow is a situation in which negative exponent becomes too small to fit in exponent field

→ In order to let the user know that overflow or underflow occurred, computers signal these events by raising an exception

→ RISC-V computers don't raise exceptions, instead, software can read Floating-point Control and Status Register to check whether overflow or underflow occurred

→ Computer must provide atleast single precision representation to confirm to IEEE Standard

→ The standard also provides an extended version to increase precision

Floating Point Addition

→ Addition is MUCH harder than integer addition because exponents may differ

→ Steps for FP Addition:

i) Align Exponents
   → Shift the mantissa of the number with smaller exponent
   → Increase smaller exponent until both equal

ii) Add / Subtract Mantissas
   → Signs must be considered
   → Similar to integer addition

iii) Normalize Result
   → If result > 2 ⟹ Shift Right
   → If result < 1 ⟹ Shift Left and adjust exponent

iv) Round the Result
   → Use Guard, Round, Sticky (G,S,R) bits

v) Check Overflow / Underflow
   → Too large exponent ⇒ ∞
   → Too small exponent ⇒ 0 or denormalized

Guard, Round, Sticky bits (GRS)

→ Used during rounding:
   Guard: First bit shifted out of mantissa
   Round: Next bit of Guard
   Sticky: OR of all remaining bits

Rounding Decisions:

→ Round to nearest even (IEEE Default)
→ If GRS bits represent ≥ 0.5 → Round up

Q. Add 0.5d and −0.4375d

A. $0.5d = 0.1b = 1.0 \times 2^{-1}$

$-0.4375d = -0.0111b = -1.110 \times 2^{-2}$

i) Align Exponents ⇒ $(1.0 \times 2^{-1}) - (0.111 \times 2^{-1})$

ii) Add Both ⇒ 0.001×2^{-1}

iii) Normalize ⇒ 1.000×2^{-4}

Convert to decimal ⇒ 0.0625 d

https:// ← Click this if you want to learn how to convert decimal to binary (Floating)

Q. Add $A = 1.101 \times 2^3$ & $B = 1.001 \times 2^1$

A. i) Align Exponents ⇒ $(1.101 \times 2^3) + (0.01001 \times 2^3)$

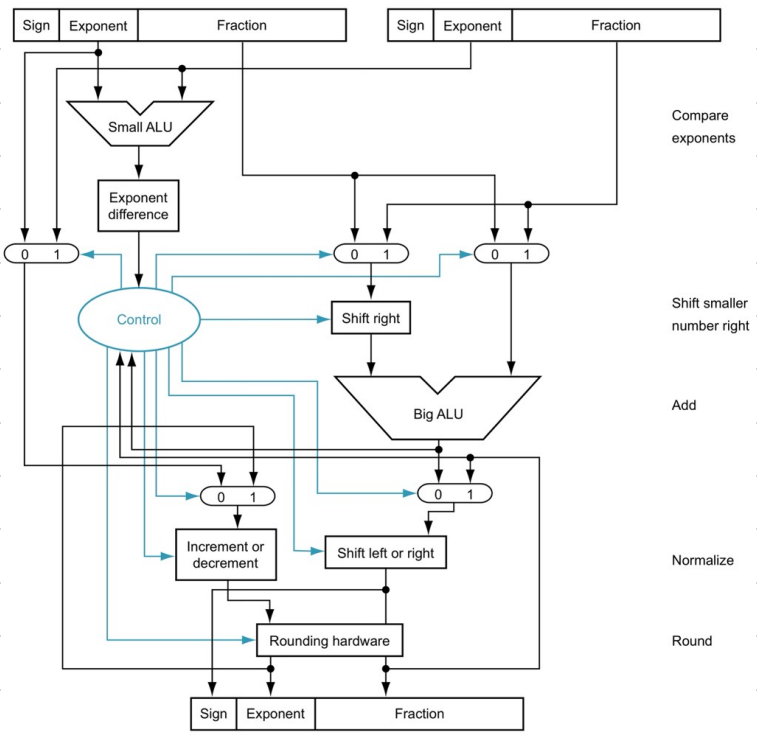
ii) Add Both ⇒ (1.11101×2^3)

iii) Normalize ⇒ 1.11101×2^3 Already Normalized

iv) Rounding ⇒ 1.11101×2^3 Already Rounded

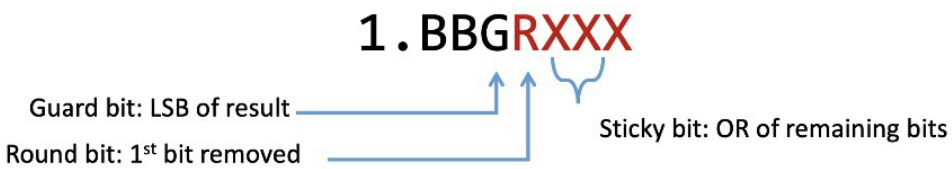
$= 1.90625 \times 8$

$= 15.25d$



→ Block Diagram of an arithmetic unit dedicated to floating point addition

Rounding



- Round up conditions
  - Round = 1, Sticky = 1 ⇒ > 0.5
  - Guard = 1, Round = 1, Sticky = 0 ⇒ Round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Systems@ETH Zürich

G	R	S	Incr?
0	0	0	No
0	0	1	No
0	1	0	No
0	1	1	No
1	0	0	No
1	0	1	Yes
1	1	0	Yes
1	1	1	Yes

Guard : First bit shifted out of mantissa

Round : Next bit of Guard

Sticky : OR of all remaining bits (Even if one 1 is there, S = 1)

Q. **ii)** In a particular floating point representation assume that there are only 4 bits fraction field. If floating point arithmetic has resulted in the following results ,how are they accurately rounded off using extra bits like guard bit ,round bit and sticky bit?
1.0101001
1.1110101
1.0011110

A. 1.0101001

→ They have asked 4 bit fraction

1.0101001 ⇒ G = 0 , R = 0 , S = 1 ⇒ No Rounding , 1.0101
(G R S)

1.1110101

→ 1.1110101 ⇒ G = 1 , R = 0 , S = 1 ⇒ Round , 1110 + 1 = 1111 ⇒ 1.1111
(G R S)

1.0011110

→ 1.0011110 ⇒ G = 1 , R = 1 , S = 0 ⇒ Round , 0011 + 1 = 0100 ⇒ 1.010
(G R S)

Q. **3.29** [20] <§3.5> Calculate the sum of 2.6125×10^{1} and $4.150390625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision described in Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps.

A. $2.6125 \times 10^{1} + 4.150390625 \times 10^{-1}$

$2.6125 \times 10^{1} = 26.125 = 11010.001 = 1.1010001000 \times 2^{4}$

$4.150390625 \times 10^{-1} = 0.4150390625 = 0.011010100111 = 1.1010100111 \times 2^{-2}$

Shift binary point six to the left to align exponents,

```
              GR
1.1010001000 00
1.0000011010 10 0111 (Guard 5 1, Round 5 0,
Sticky 5 1)
-------------------
1.1010100010 10
```

In this case the extra bit (G,R,S) is more than half of the least significant bit (0).

Thus, the value is rounded up.

$1.1010100011 \times 2^{4} = 11010.100011 \times 2^{0} = 26.546875 = 2.6546875 \times 10^{1}$

Q. **3.32** [20] <§3.10> Calculate $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3}$ by hand, assuming each of the values is stored in the 16-bit half-precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating-point format and in decimal.

A. **3.32** $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3}$

$3.984375 \times 10^{-1} = 1.1001100000 \times 2^{-2}$

$3.4375 \times 10^{-1} = 1.0110000000 \times 2^{-2}$

$1.771 \times 10^{3} = 1771 = 1.1011101011 \times 2^{10}$

shift binary point of smaller left 12 so exponents match

```
(A)         1.1001100000
(B)        +1.0110000000
           -------------
           10.1111100000 Normalize,
(A+B)       1.0111110000 × 2^-1
(C)        +1.1011101011
(A+B)         .0000000000 10 111110000 Guard = 1,
                          Round = 0, Sticky = 1
          ---------------
(A+B)+C +1.1011101011 10 1 Round up
```

$(A+B)+C = 1.1011101100 \times 2^{10} = 0110101011101100 = 1772$

To understand GRS,

0.00000000001011111 0000

↑↑ G R — if any of these bits 1, then S = 1

So, G = 1, R = 0, S = 1

We round up if G=1 AND (R=0 OR S=1) = 1

In this case 1 AND (0 OR 1) = 1 ✓

Then A+B = 0.0000000001

+C = 1.1011101011

1.1011101100 × 2^{10}

= 1 1011101100

= 1772d

Q. **3.33** [20] <§3.10> Calculate $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^{3})$ by hand, assuming each of the values is stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating-point format and in decimal.

A. $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^{3})$

$3.984375 \times 10^{-1} = 1.1001100000 \times 2^{-2}$

$3.4375 \times 10^{-1} = 1.0110000000 \times 2^{-2}$

$1.771 \times 10^{3} = 1771 = 1.1011101011 \times 2^{10}$

shift binary point of smaller left 12 so exponents match

```
(B)         .0000000000 01 0110000000 Guard = 0,
                           Round = 1, Sticky = 1
(C)        +1.1011101011
           -------------
(B+C)      +1.1011101011
 (A)         .0000000000  011001100000
           -------------
A + (B + C) + 1.1011101011 No round
```

$A + (B + C) + 1.1011101011 \times 2^{10} = 0110101011101011 = 1771$

Floating Point Multiplication

→ It is much easier than addition

→ Steps for FP Multiplication :

i) Determine the sign

→ sign (result) = sign (a) XOR sign (b)

ii) Add the exponents (subtract bias)

→ IEEE 754 stores a biased exponent

→ So, $E = E_a + E_b - bias$

→ Bias for single precision = 127

double precision = 1023

iii) Multiply the Mantissas

→ Multiply (1.fractionA) x (1.fraction B)

→ The product can be b/w 1 and 4 , if product ≥ 2 then shift right & increment exponent

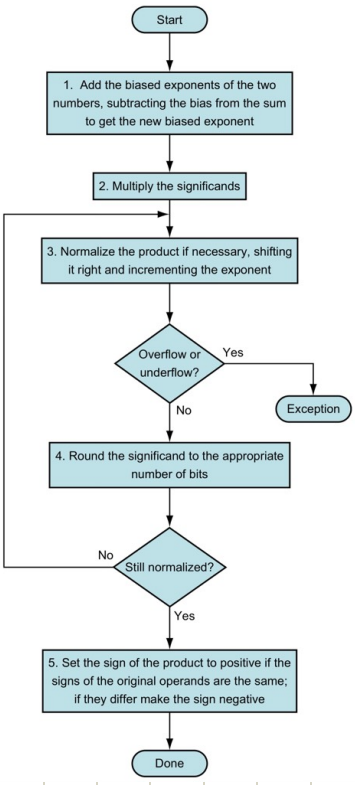
iv) Normalize

→ If mantissa ≥ 2 , mantissa = mantissa/2

exponent = exponent + 1

If mantissa < 1 , shift left until leading bit is 1

exponent = exponent - shifts



Q. Multiply 0.5d and -0.4375d

A. $0.5 \Rightarrow 1.000 \times 2^{-1}$

$-0.4375 \Rightarrow -1.110 \times 2^{-2}$

i) Determining sign,

0 XOR 1 = 1 ⇒ Negative

ii) Add exponent w/o bias

-1 + (-2) = -3

Now add bias, -3 + 127 = 124

iii) Multiply significands

```
  1.000
x 1.110
-------
 0000
 1000
10000
10000
-------
1.110000
```

Product is $1.110000 \times 2^{-3} = 1.110 \times 2^{-3}$ (4 bit)

iv) Normalize :

Product already normalized & exponent is also 127 ≥ -3 ≥ -126 so no overflow or underflow

Now final product = -0.001110 b

= -0.21875 d

Floating Point Instructions in RISC-V

RISC-V supports the IEEE 754 single-precision and double-precision formats with these instructions:

- Floating-point *addition, single* (`fadd.s`) and *addition, double* (`fadd.d`)
- Floating-point *subtraction, single* (`fsub.s`) and *subtraction, double* (`fsub.d`)
- Floating-point *multiplication, single* (`fmul.s`) and *multiplication, double* (`fmul.d`)
- Floating-point *division, single* (`fdiv.s`) and *division, double* (`fdiv.d`)
- Floating-point square root, single (`fsqrt.s`) and square root, double (`fsqrt.d`)
- Floating-point equals, single (`feq.s`) and equals, double (`feq.d`)
- Floating-point less-than, single (`flt.s`) and less-than, double (`flt.d`)
- Floating-point less-than-or-equals, single (`fle.s`) and less-than-or-equals, double (`fle.d`)

ex:

```
flw     f0, 0(x10)  //  Load 32-bit F.P. number into f0
flw     f1, 4(x10)  //  Load 32-bit F.P. number into f1
fadd.s  f2, f0, f1  //  f2 = f0 + f1, single precision
fsw     f2, 8(x10)  //  Store 32-bit F.P. number from f2
```

RISC-V floating-point operands

32 floating-point registers	f0-f31	An *f*-register can hold either a single-precision floating-point number or a double-precision floating-point number.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

→ Other instructions

RISC-V floating-point assembly language

Arithmetic	FP add single	`fadd.s f0, f1, f2`	f0 = f1 + f2	FP add (single precision)
	FP subtract single	`fsub.s f0, f1, f2`	f0 = f1 - f2	FP subtract (single precision)
	FP multiply single	`fmul.s f0, f1, f2`	f0 = f1 * f2	FP multiply (single precision)
	FP divide single	`fdiv.s f0, f1, f2`	f0 = f1 / f2	FP divide (single precision)
	FP square root single	`fsqrt.s f0, f1`	f0 = √f1	FP square root (single precision)
	FP add double	`fadd.d f0, f1, f2`	f0 = f1 + f2	FP add (double precision)
	FP subtract double	`fsub.d f0, f1, f2`	f0 = f1 - f2	FP subtract (double precision)
	FP multiply double	`fmul.d f0, f1, f2`	f0 = f1 * f2	FP multiply (double precision)
	FP divide double	`fdiv.d f0, f1, f2`	f0 = f1 / f2	FP divide (double precision)
	FP square root double	`fsqrt.d f0, f1`	f0 = √f1	FP square root (double precision)
Comparison	FP equality single	`feq.s x5, f0, f1`	x5 = 1 if f0 == f1, else 0	FP comparison (single precision)
	FP less than single	`flt.s x5, f0, f1`	x5 = 1 if f0 < f1, else 0	FP comparison (single precision)
	FP less than or equals single	`fle.s x5, f0, f1`	x5 = 1 if f0 <= f1, else 0	FP comparison (single precision)
	FP equality double	`feq.d x5, f0, f1`	x5 = 1 if f0 == f1, else 0	FP comparison (double precision)
	FP less than double	`flt.d x5, f0, f1`	x5 = 1 if f0 < f1, else 0	FP comparison (double precision)
	FP less than or equals double	`fle.d x5, f0, f1`	x5 = 1 if f0 <= f1, else 0	FP comparison (double precision)
Data transfer	FP load word	`flw f0, 4(x5)`	f0 = Memory[x5 + 4]	Load single-precision from memory
	FP load doubleword	`fld f0, 8(x5)`	f0 = Memory[x5 + 8]	Load double-precision from memory
	FP store word	`fsw f0, 4(x5)`	Memory[x5 + 4] = f0	Store single-precision from memory
	FP store doubleword	`fsd f0, 8(x5)`	Memory[x5 + 8] = f0	Store double-precision from memory

ex: C Code to convert Fahrenheit to Celsius in ALP

```
float f2c (float fahr)
{
    return ((5.0f/9.0f) *(fahr - 32.0f));
}
```

→

```
f2c:
    flw f0, const5(x3) // f0 = 5.0f
    flw f1, const9(x3) // f1 = 9.0f
    fdiv.s f0, f0, f1 // f0 = 5.0f / 9.0f
    flw    f1, const32(x3) // f1 = 32.0f
    fsub.s f10, f10, f1    // f10 = fahr - 32.0f
    fmul.s f10, f0, f10 // f10 = (5.0f / 9.0f)*(fahr - 32.0f)
    jalr   x0, 0(x1)   // return
```

1) In a single precision floating point representation, If the Exponent is zero then what value is stored in the exponent field?
   a) Zero
   b) 127
   c) 254
   d) 1023

2) For $+(0.2)_{10}$ represented using IEEE 754 32-Bit Floating-Point Representation what will be bias Exponent value?
   a) 124ten
   b) -3 ten
   c) 130 ten
   d) +3 ten

3) For $+(0.2)_{10}$ represented using IEEE 754 32-Bit Floating-Point Representation what will be stored in the Mantissa field
   a) 100 1100 1100 1100 1100 1100
   b) 001 1001 1001 1001 1001 1001
   c) 000 0000 0000 0000 0000 0000
   d) None of these

4) Given 0x0C00 0000 single precision floating point number. What is the actual exponent
   a) 24ten
   b) -24ten
   c) -103ten
   d) 103ten

5) What decimal number does the bit patten 0x0C00 0000 single precision floating point number represents?
   a) $+1.0 \times 2^{-103}$
   b) -1.0×2^{-103}
   c) $+1.0 \times 2^{24}$
   d) -1.0×2^{-24}

6) The size of Mantissa in a single precision FPR including implicit bits is _____ size?
   a) 23
   b) 24
   c) 21
   d) 52

7) What is the bias exponent of 63.25 represented using double precision IEEE754 standard
   a) 1028 ten
   b) 5 ten
   c) 132 ten
   d) None of these

8) In **RISC-V computers, if** overflow or underflow condition occurs in IEEE754 standard it is indicated by _____
   a) By generating an exception/interrupt
   b) By Flags in F**loating-point Control and Status Register**
   c) There will be No overflow/underflow in floating point standards
   d) None of these

9) The size of Exponent and Mantissa field in double precision floating point reorientation is _____ and _____size
   a) 11 bytes and 52 bytes
   b) 11 bits and 52 bits
   c) 8 bits and 22 bits
   d) 8 bits and 52 bits

10) The first step in Adding two floating point numbers is _____
   a) Matching the exponent of two numbers by shifting the right Significand(mantissa) of the Number with a smaller exponent for n times where n=|Ea-Eb| and Ea and Eb are exponents of two numbers
   b) Matching the Mantissa of two numbers by shifting the right Significand(mantissa) of the Number with a smaller exponent for n times where n=|Ea-Eb| and Ea and Eb are exponents of two numbers Matching sign of two numbers
   c) None of these

11) In the conversion formula $(-1)^S F.2^E$ used for converting IEEE754 standards into decimal numbers, s indicates _____
   a) Sigma of Number
   b) Sign of numbers
   c) Symbol
   d) Significand

12) Represent the number-0.75 in $(-1)^S F.2^E$ form where E is actual exponent.
   a) $+1.1000 \times 2^{-1}$
   b) -0.1100×2^{-1}
   c) -1.1000×2^{-1}
   d) -1.1100×2^{-1}

13) For the bit patten 0xB000 0000 of Single precision IEEE754 standard, Identify the actual exponent value.
   a) -31d
   b) 60d
   c) 96d
   d) None of these

14) What is the Single precision FP bit patten of the number $+1.111110100000 \times 2^5$
   a) 1100 0010 0111 1101 0000 0000 0000 0000 =0XC27D 0000
   b) 0100 0010 0111 1101 0000 0000 0000 0000 =0X427D 0000
   c) 0000 0010 1111 1101 0000 0000 0000 0000 =0x02FD 0000
   d) 1000 0010 1111 1101 0000 0000 0000 0000 =0x82FD 0000

Q. 1.a. a) Write the RISC-V code to load three single precision numbers (a,b, and c) from memory, perform (axb+c), and then store the result in memory. Assume x10 holds the base address. 3M

b) What is the size of the machine code generated for C extension instruction. 1M

A. a)
```
flw   f0, 0(x10)      # f0 = a
flw   f1, 4(x10)      # f1 = b
flw   f3, 8(x10)      # f3 = c
fmul.s f2, f0, f1     # f2 = a * b
fadd.s f4, f3, f2     # f4 = (a*b) + c
fsw   f4, 12(x10)
```

b) 16 bits / 2 bytes