Arithmetic for Computers

Exceptions

- Control is the most challenging aspect of processor design: it is both the hardest part to get right and the toughest part to make fast
- One of the demanding tasks of control is implementing exceptions and interrupts—events other than branches that change the normal flow of instruction execution
- Ex: undefined instruction, I/O devices to communicate with the processor

- Intelx86- interrupts
- RISC-V- exception- unexpected change(internal/external)
- Interrupt- external

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

- Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a processor, which determines the clock cycle time and thus performance.
- Without proper attention to exceptions during design of the control unit, attempts to add exceptions to an intricate implementation can significantly reduce performance, as well as complicate the task of getting the design correct

How Exceptions are Handled in the RISC-V Architecture

- Exceptions generated are execution of an undefined instruction or a hardware malfunction
- The basic action that the processor must perform when an exception occurs is to save the address of the unfortunate instruction in the supervisor exception cause register (SEPC) and then transfer control to the operating system at some specified address
- The os can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to a malfunction, or stopping the execution of the program and reporting an error

- After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the SEPC to determine where to restart the execution of the program
- For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it.
- There are two main methods used to communicate the reason for an exception- Supervisor Exception Cause Register or (SCAUSE), vectored interrupts

- SCAUSE-include a register which holds a field that indicates the reason for the exception
- vectored interrupt- An interrupt for which the address to which control is transferred is determined by the cause of the exception

Exception type	Exception vector address to be added to a Vector Table Base Register
Undefined instruction	$00\ 0100\ 0000_{two}$
System Error (hardware malfunction)	$01\ 1000\ 0000_{two}$

Implementing the exception system

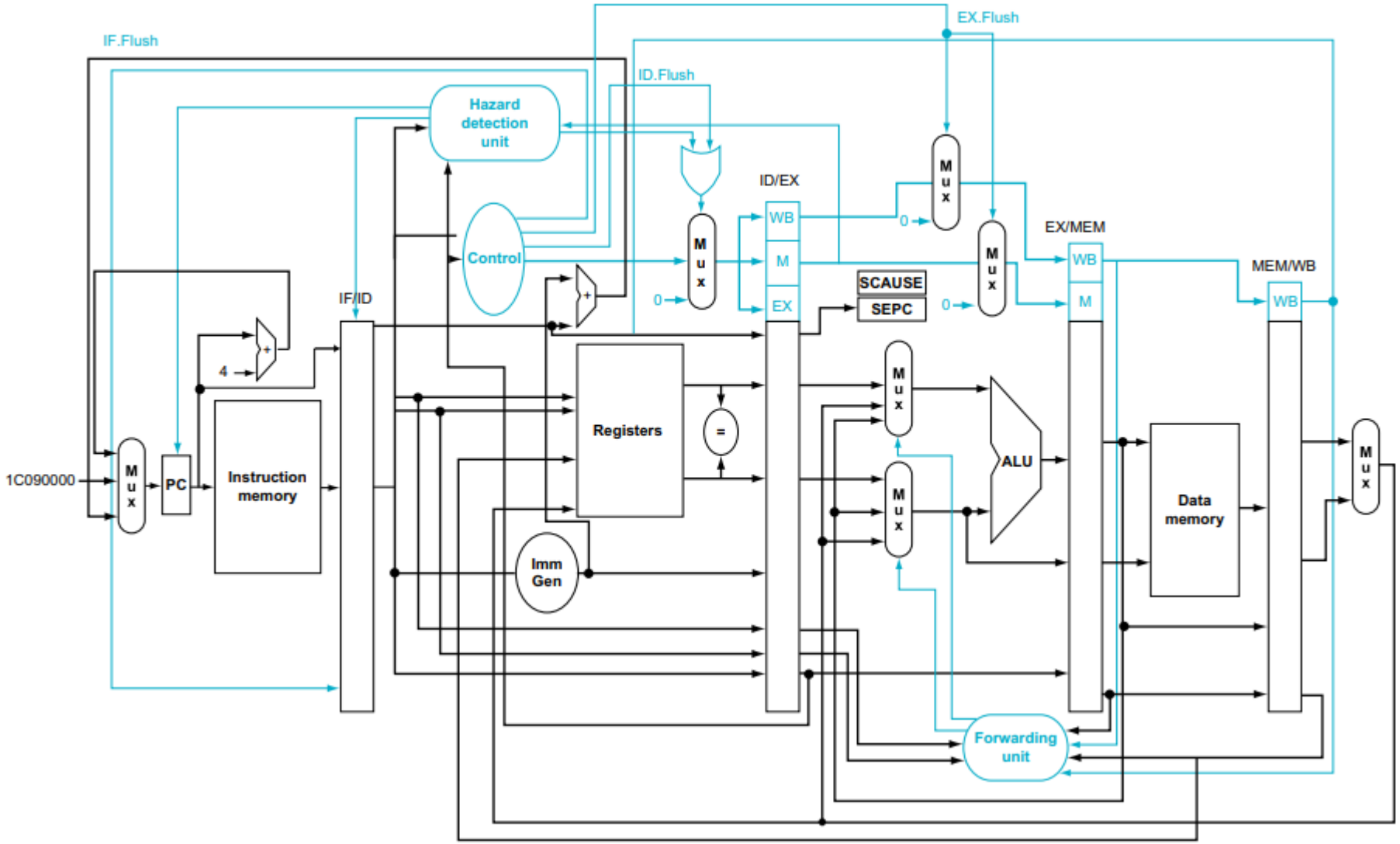
- implementing the exception system with the single interrupt entry point being the address 0000 0000 1C09 0000hex.
- SEPC: A 64-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- SCAUSE: A register used to record the cause of the exception. In the RISC-V architecture, this register is 64 bits, although most bits are currently unused. Assume there is a field that encodes the two possible exception sources mentioned above, with 2 representing an undefined instruction and 12 representing hardware malfunction

Exceptions in a Pipelined Implementation

- A pipelined implementation treats exceptions as another form of control hazard
- we must flush the instructions that follow the interrupted instruction from the pipeline and begin fetching instructions from the new address
- use the same mechanism we used for taken branches, but this time the exception causes the deasserting of control lines

- flush the instruction in the IF stage by turning it into a nop
- To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls
- A new control signal, called ID.Flush, is ORed with the stall signal from the hazard detection unit to flush during ID
- To flush the instruction in the EX phase, we use a new signal called EX.Flush to cause new multiplexors to zero the control lines
- To start fetching instructions from location 0000 0000 1C09 0000hex, which we are using as the RISC-V exception address, we simply add an additional input to the PC multiplexor that sends 0000 0000 1C09 0000hex to the PC

The datapath with controls to handle exceptions



Exception in a Pipelined Computer

Given this instruction sequence,

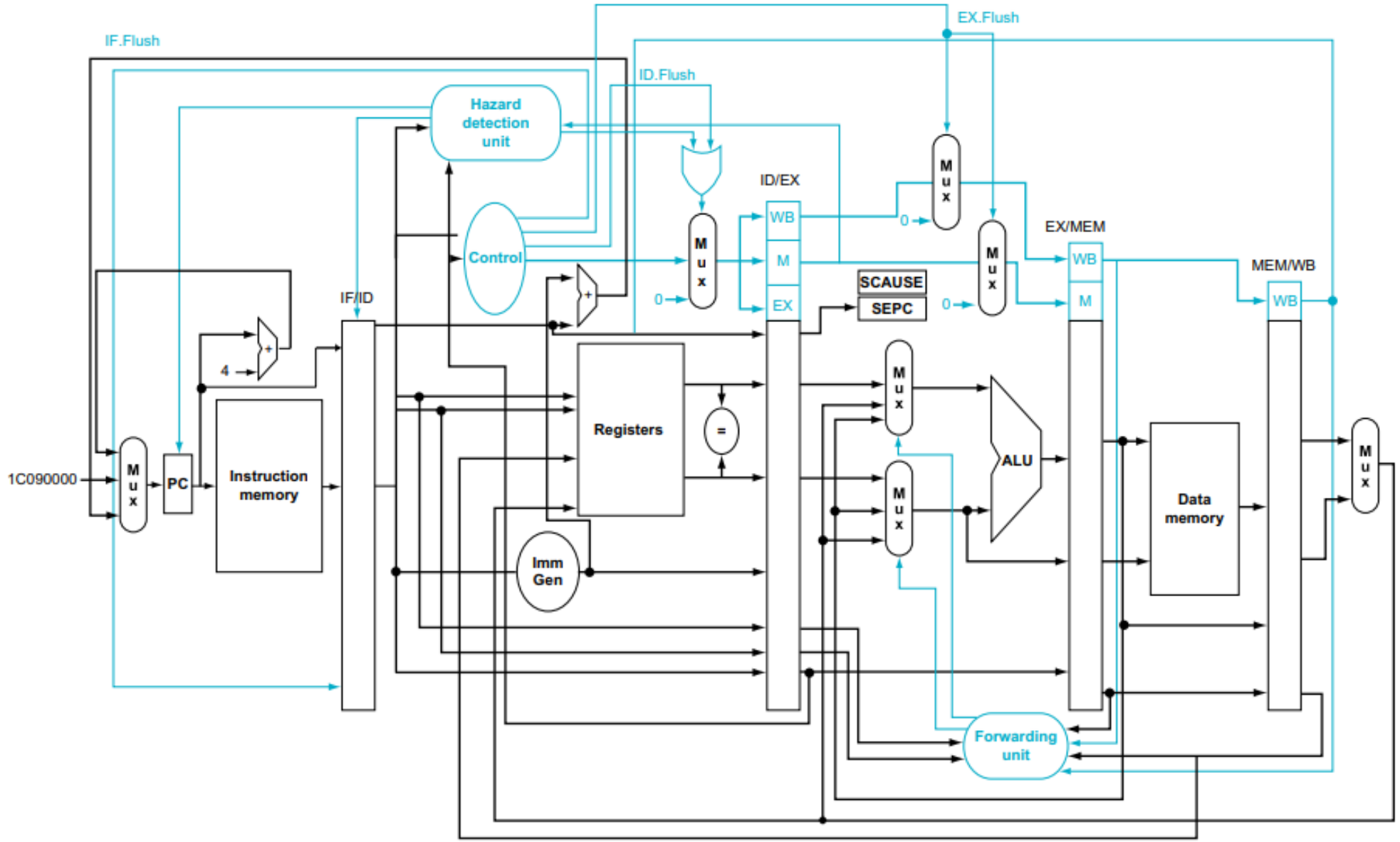
```
40hex   sub    x11, x2, x4
44hex   and    x12, x2, x5
48hex   or     x13, x2, x6
4Chex   add    x1,  x2, x1
50hex   sub    x15, x6, x7
54hex   lw     x16, 100(x7)
. . .
```

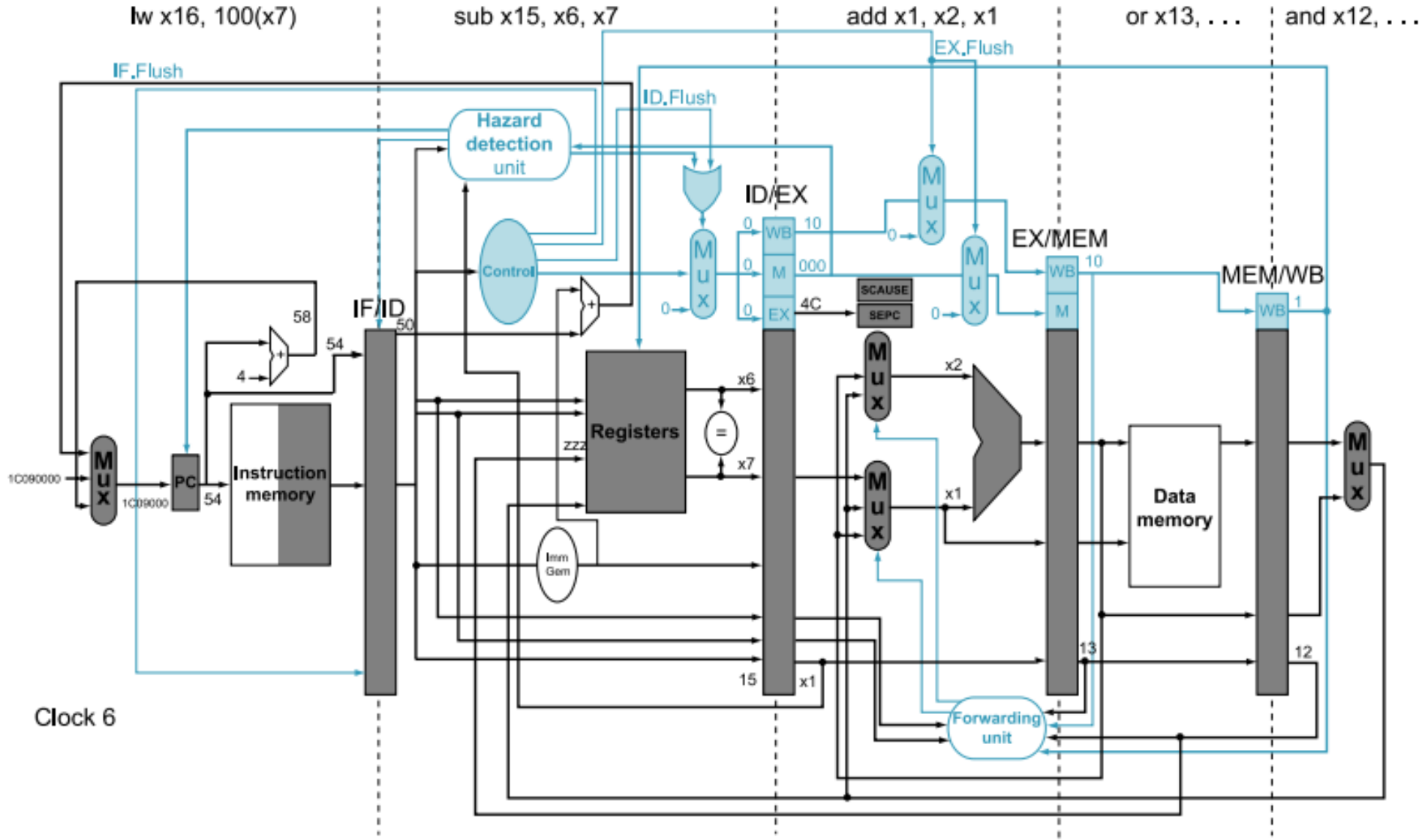
assume the instructions to be invoked on an exception begin like this:

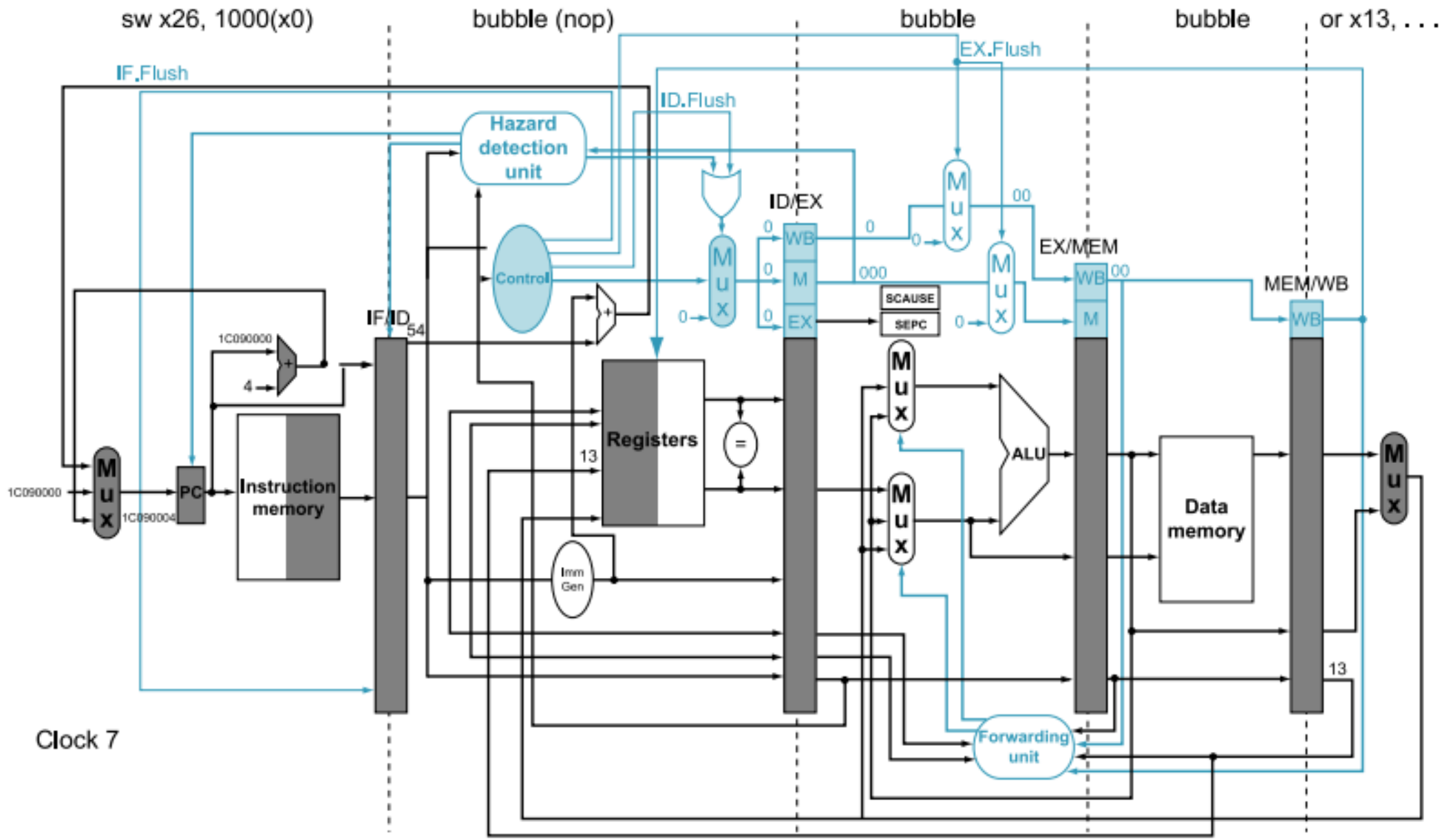
```
1C090000hex   sw    x26, 1000(x10)
1C090004hex   sw    x27, 1008(x10)
. . .
```

Show what happens in the pipeline if a hardware malfunction exception occurs in the `add` instruction.

- Assume the hardware malfunction is detected during that phase, and 0000 0000 1C09 0000hex is forced into the PC.
- Clock cycle 7 shows that the add and following instructions are flushed, and the first instruction of the exception handling code is fetched.
- Note that the address of the add instruction is saved: 4Chex.


IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
WB
M
EX
EX/MEM
MEM/WB
SCAUSE
SEPC
IF/ID
1C090000
PC
Instruction memory
Registers
Imm Gen
ALU
Data memory
Forwarding unit
Mux
4
0
=


lw x16, 100(x7)
sub x15, x6, x7
add x1, x2, x1
or x13, . . .
and x12, . . .
IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
000
10
4C
SCAUSE
SEPC
Registers
Instruction memory
Imm Gen
Data memory
Forwarding unit
PC
Mux
x1
x2
x6
x7
zzz
58
54
50
15
13
12
1C090000
Clock 6


sw x26, 1000(x0)
bubble (nop)
bubble
bubble
or x13, . . .
IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
54
WB
M
EX
SCAUSE
SEPC
1C090000
4
1C090004
PC
Instruction memory
Registers
13
Imm Gen
=
ALU
Data memory
Forwarding unit
Mux
00
000
Clock 7

- imprecise interrupt Also called imprecise exception. Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.
- precise interrupt Also called precise exception. An interrupt or exception that is always associated with the correct instruction in pipelined computers

Parallelism via Instructions

- Pipelining exploits the potential parallelism among instructions. This parallelism is called, naturally enough, instruction-level parallelism (ILP).
- two primary methods for increasing the potential amount of instruction level parallelism
- The first is increasing the depth of the pipeline to overlap more instructions
- replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is multiple issue

- static multiple issue An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution
- dynamic multiple issue An approach to implementing a multiple issue processor where many decisions are made during execution by the processor
- issue slots The positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint

- Two primary and distinct responsibilities must be dealt within a multiple-issue pipeline:
- how does the processor determine how many instructions, and which instructions can be issued in a given clock cycle?

-static issue processors, this process is at least partially handled by the compiler; in dynamic issue designs, it is normally dealt with at runtime by the processor

- Dealing with data and control hazards: in static issue processors, the compiler handles some or all the consequences of data and control hazards statically. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time

The Concept of Speculation

- speculation is an approach that allows the compiler or the processor to “guess” about the properties of an instruction, to enable execution to begin for other instructions that may depend on the speculated instruction
- Ex1: speculate on the outcome of a branch, so that instructions after the branch could be executed earlier
- Ex2:speculate that a store that precedes a load does not refer to the same address, which would allow the load to be executed before the store

- The difficulty with speculation is that it may be wrong.
- So, any speculation mechanism must include both a method to check if the guess was right and a method to unroll or back out the effects of the instructions that were executed speculatively. The implementation of this back-out capability adds complexity.
- when the speculation is wrong-the processor usually buffers the speculative results until it knows they are no longer speculative
- If the speculation is correct, the instructions are completed by allowing the contents of the buffers to be written to the registers or memory

- speculating on certain instructions may introduce exceptions that were formerly not present

Static Multiple Issue

- Static multiple-issue processors all use the compiler to assist with packaging instructions and handling hazards
- Very Long Instruction Word (VLIW) A style of instruction set architecture that launches many operations that are defined to be independent in a single-wide instruction, typically with many separate opcode fields

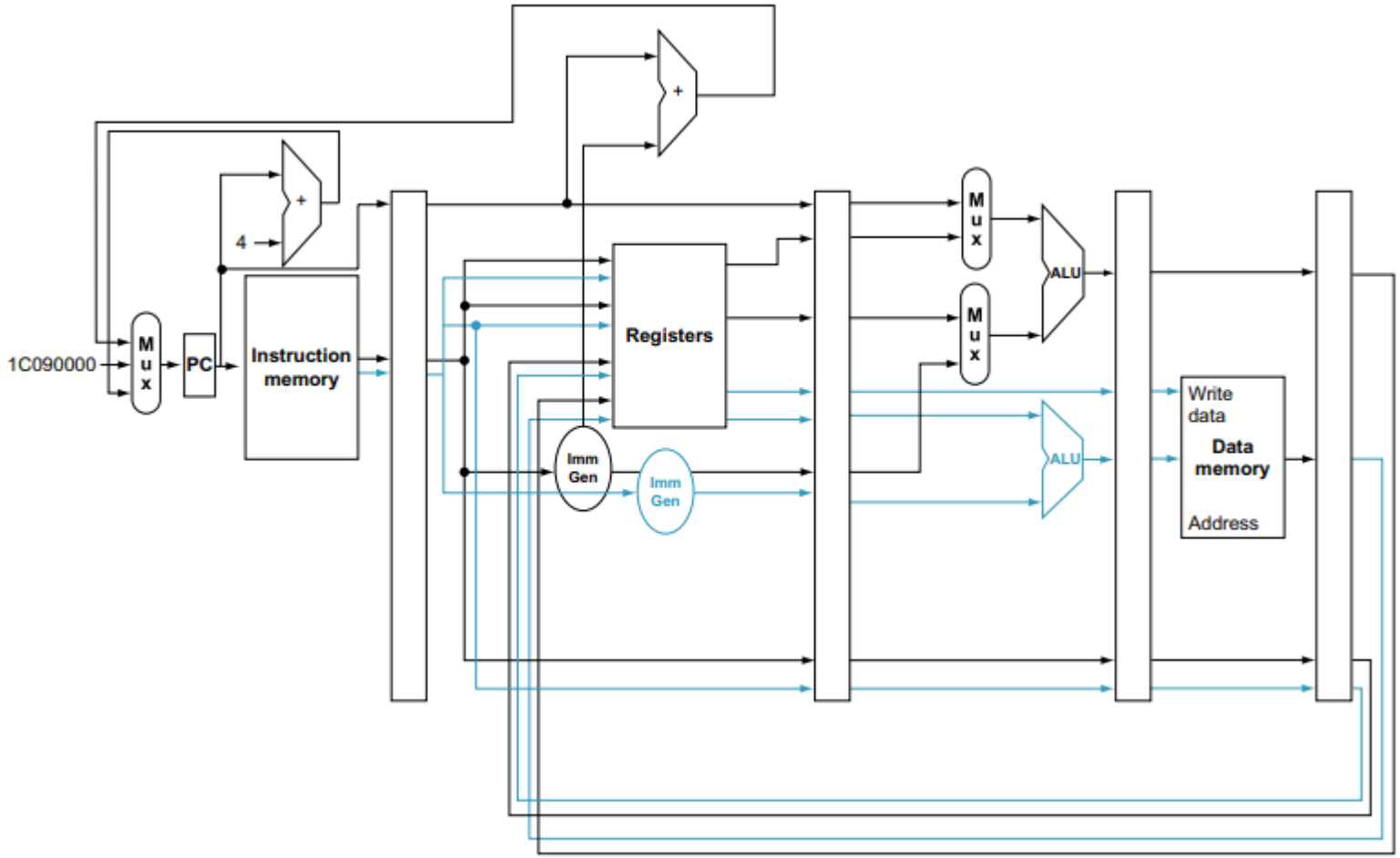
Static two-issue pipeline in operation

- consider a simple two-issue RISC-V processor, where one of the instructions can be an integer ALU operation or branch and the other can be a load or store

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

- Static multiple-issue processors vary in how they deal with potential data and control hazards
- In some designs, the compiler takes full responsibility for removing all hazards, scheduling the code, and inserting no-ops so that the code executes without any need for hazard detection or hardware-generated stalls
- the hardware detects data hazards and generates stalls between two issue packets, while requiring that the compiler avoid all dependences within an instruction packet

A static two-issue datapath



How would this loop be scheduled on a static two-issue pipeline for RISC-V?

```
Loop: lw   x31, 0(x20)     // x31=array element
      add  x31, x31, x21   // add scalar in x21
      sw   x31, 0(x20)     // store result
      addi x20, x20, -4    // decrement pointer
      blt  x22, x20, Loop  // compare to loop limit,
                           // branch if x20 > x22
```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

The scheduled code as it would look on a two-issue RISC-V pipeline

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw x31, 0(x20)	1
	addi x20, x20, -4		2
	add x31, x31, x21		3
	blt x22, x20, Loop	sw x31, 4(x20)	4

- we get the disappointing CPI of 0.8 versus the best case of 0.5, or an IPC of 1.25 versus 2.0.

- loop unrolling A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together
- register renaming The renaming of registers by the compiler or hardware to remove antidependences.
- antidependence Also called name dependence An ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions.

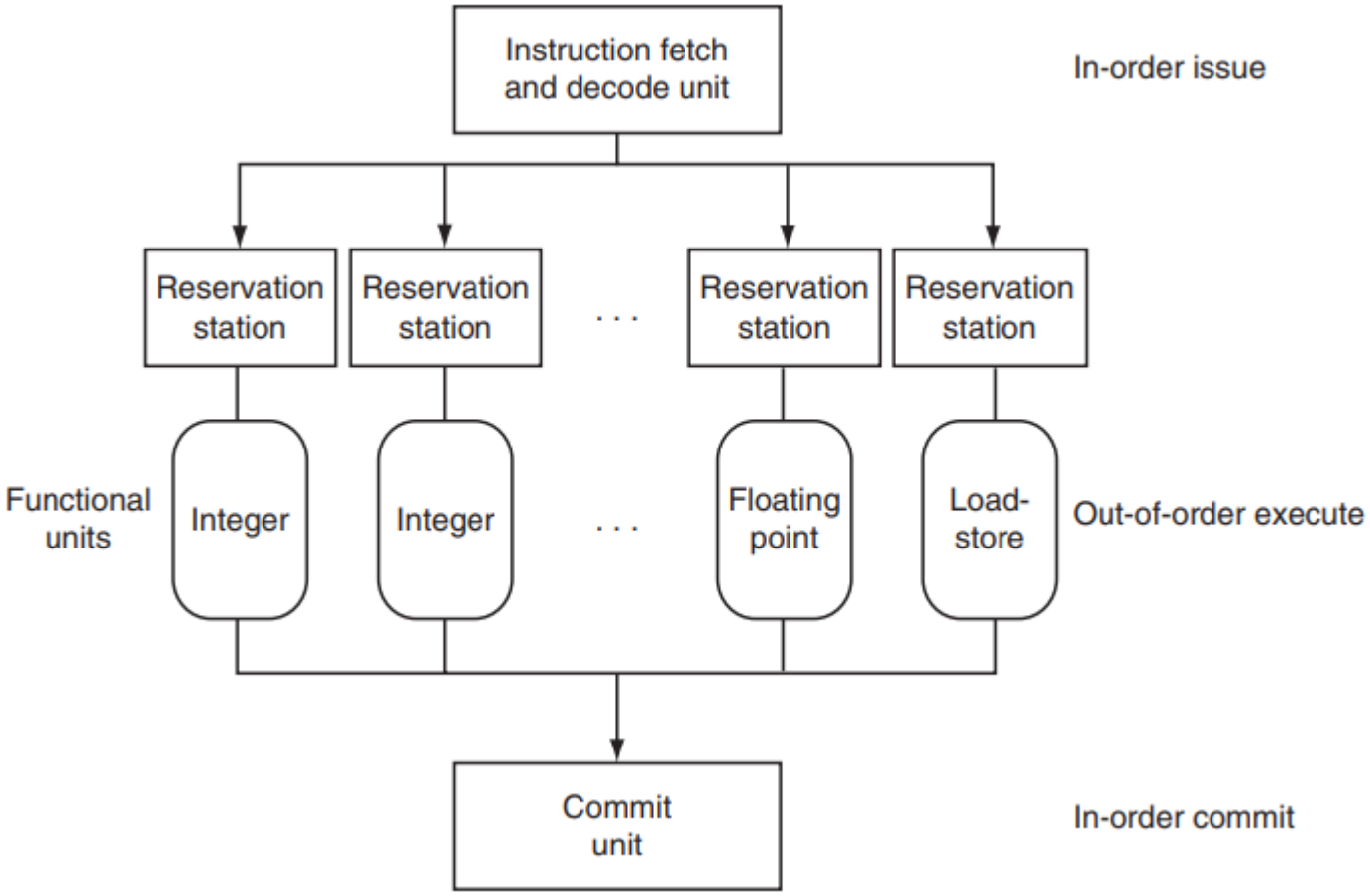
Dynamic Multiple-Issue Processors

- superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution
- dynamic pipeline scheduling Hardware support for reordering the order of instruction execution to avoid stalls

dynamic pipeline scheduling

```
lw    x31, 0(x21)
add   x1,  x31, x2
sub   x23, x23, x3
andi  x5,  x23, 20
```

The three primary units of a dynamically scheduled pipeline



- commit unit The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer visible registers and memory
- reservation station A buffer within a functional unit that holds the operands and the operation
- reorder buffer The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.

- out-of-order execution A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait
- in-order commit A commit in which the results of pipelined execution are written to the programmer visible state in the same order that instructions are fetched

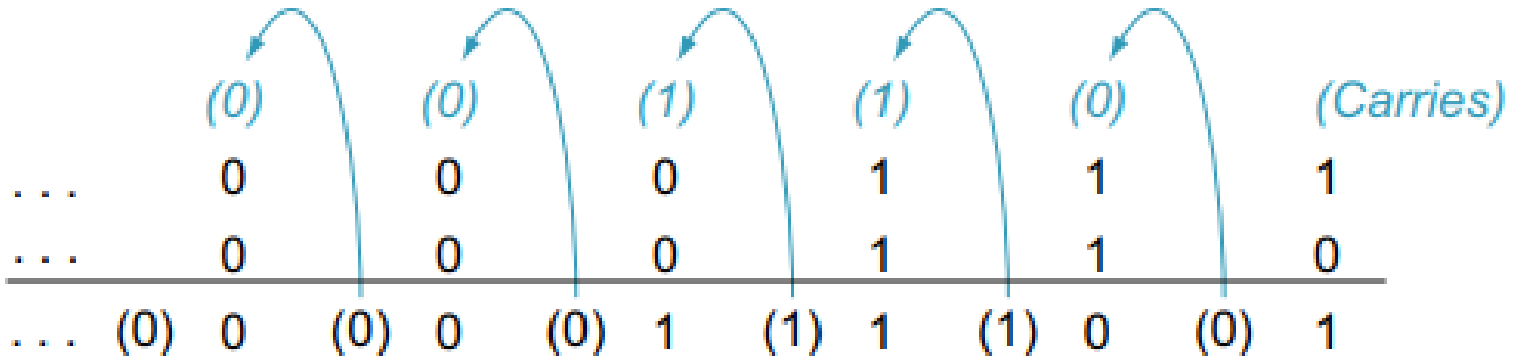
Addition and Subtraction

- Binary Addition and Subtraction

Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

$$
\begin{array}{rll}
 & 00000000\ 00000000\ 00000000\ 00000111_{two} & = 7_{ten} \\
+ & 00000000\ 00000000\ 00000000\ 00000110_{two} & = 6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00001101_{two} & = 13_{ten}
\end{array}
$$

Binary addition, showing carries from right to left



Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\ - & 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\ \hline = & 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten} \end{array}$$

or via addition using the two's complement representation of −6:

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\ + & 11111111\ 11111111\ 11111111\ 11111010_{two} = -6_{ten} \\ \hline = & 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten} \end{array}$$

Overflow conditions for addition and subtraction

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

- RISC-V does have data transfer operations for bytes and halfwords.
- What RISC-V instructions should be generated for byte and halfword arithmetic operations?

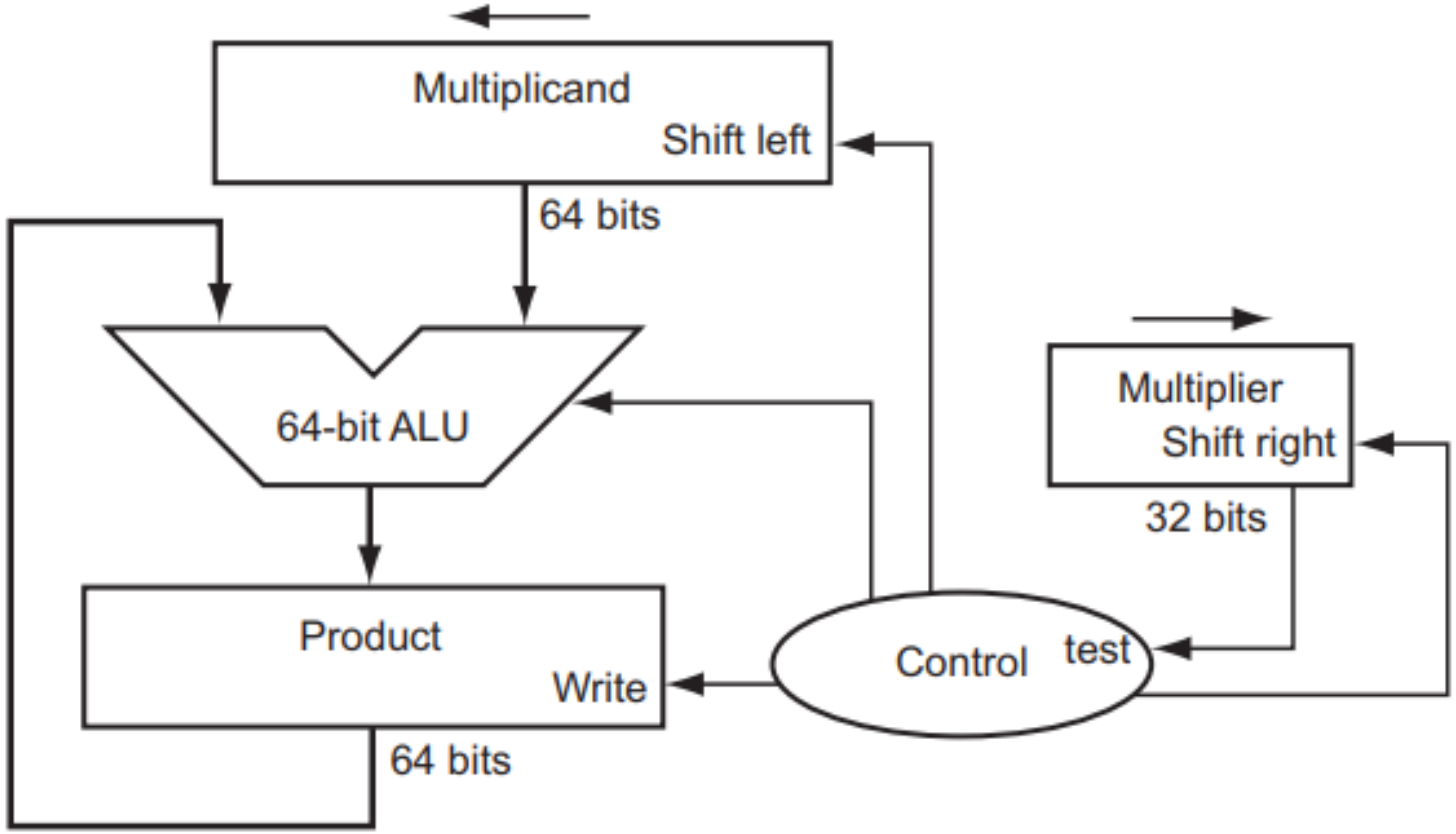
1. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`, using `and` to mask result to 8 or 16 bits after each operation; then store using `sb`, `sh`.
2. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mul`, `div`; then store using `sb`, `sh`.

- Saturation means that when a calculation overflows, the result is set to the largest positive number or the most negative number, rather than a modulo calculation as in two’s complement arithmetic

Multiplication

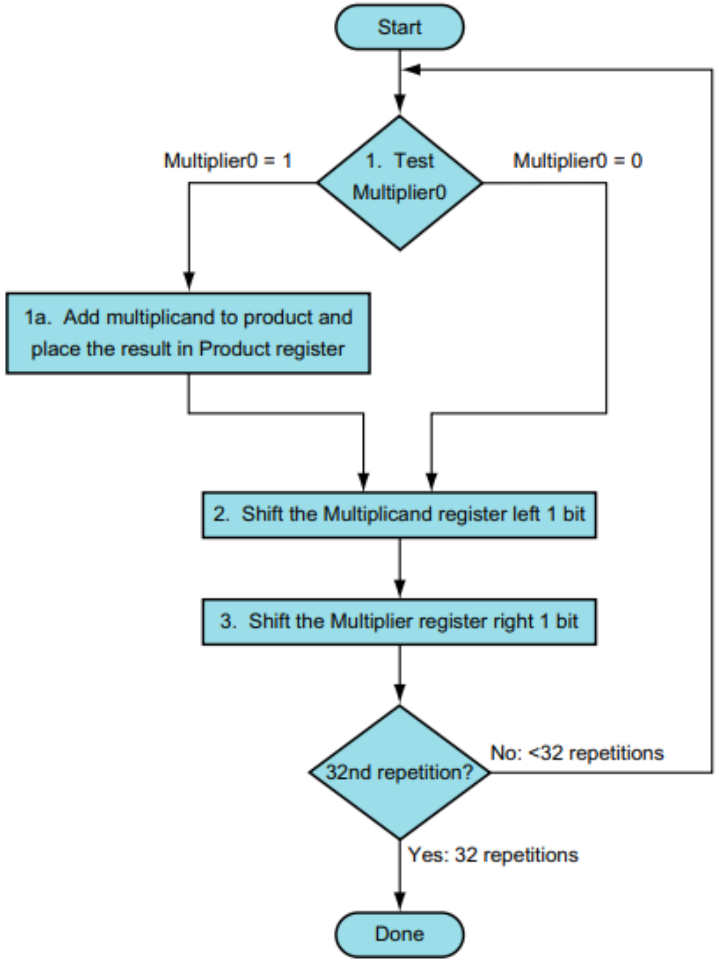
Multiplicand		1000_{ten}
Multiplier	x	1001_{ten}
		1000
		0000
		0000
		1000
Product		1001000_{ten}

Sequential Version of the Multiplication Algorithm and Hardware



- assume that the multiplier is in the 32-bit multiplier register and that the 64-bit product register is initialized to 0
- Over 32 steps, a 32-bit multiplicand would move 32 bits to the left. Hence, we need a 64-bit multiplicand register, initialized with the 32-bit multiplicand in the right half and zero in the left half.
- This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit product register

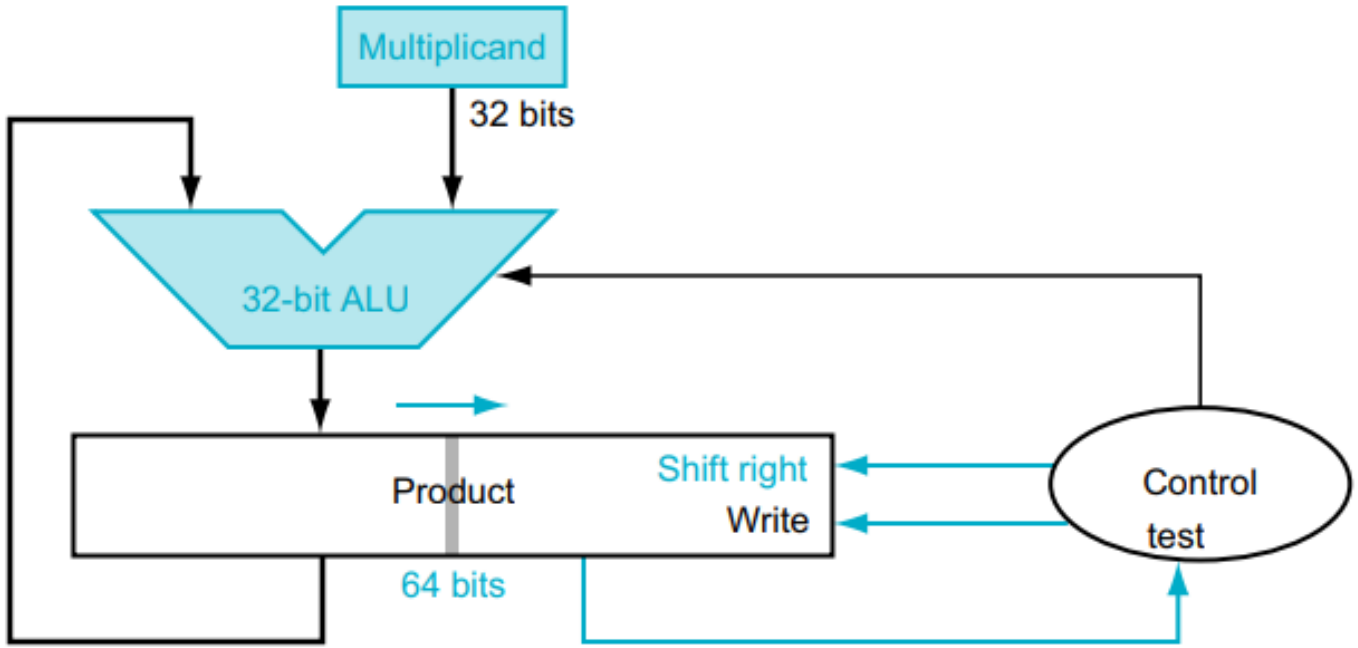
The first multiplication algorithm



Multiply example Using 4-bit numbers to save space, multiply 2ten × 3ten, or 0010two × 0011two.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Refined version of the multiplication hardware



- The Multiplicand register and ALU have been reduced to 32 bits. Now the product is shifted right.
- The separate Multiplier register also disappeared.
- The multiplier is placed instead in the right half of the Product register, which has grown by one bit to 65 bits to hold the carry-out of the adder

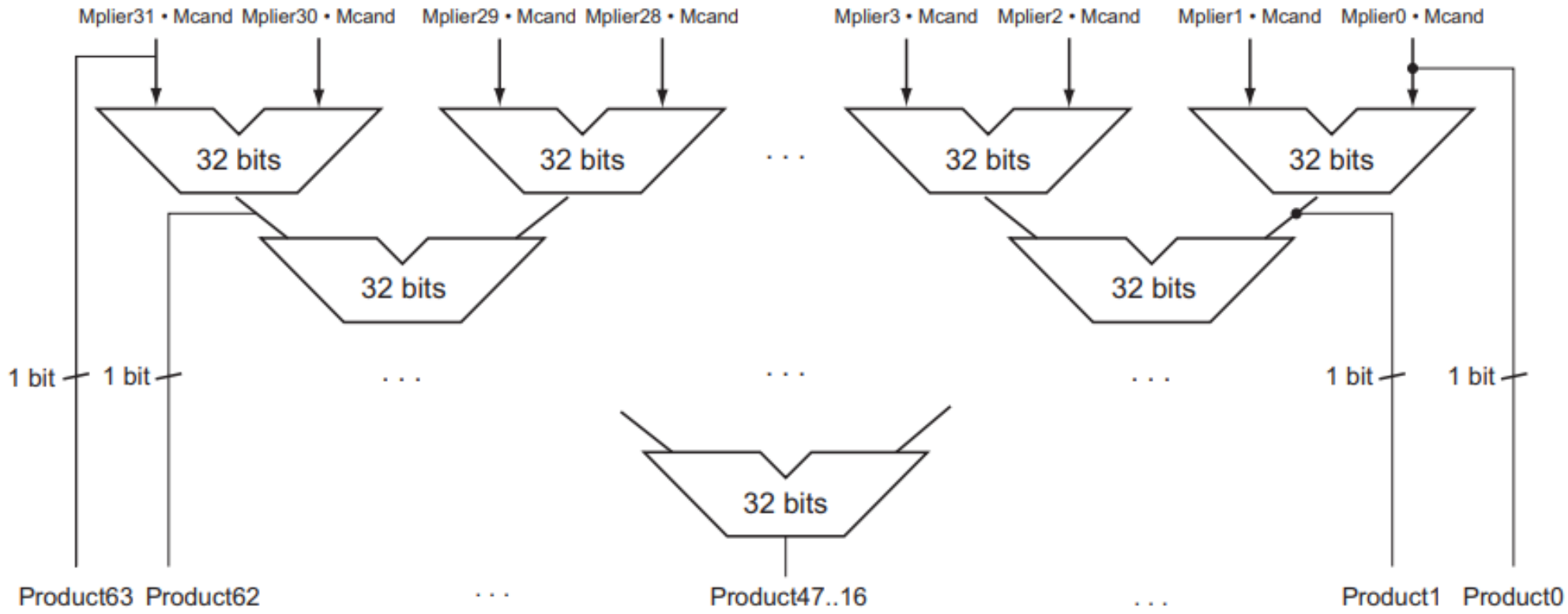
Signed Multiplication

- The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember their original signs
- The algorithms should next be run for 31 iterations, leaving the signs out of the calculation
- we need to negate the product only if the original signs disagree.

Faster Multiplication

- Faster multiplications are possible by essentially providing one 32- bit adder for each bit of the multiplier:
- one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder

Rather than use a single 32-bit adder 31 times, this hardware “unrolls the loop” to use 31 adders and then organizes them to minimize delay.



Multiply in RISC-V

- To produce a properly signed or unsigned 64-bit product, RISC-V has four instructions
- multiply (mul), multiply high (mulh), multiply high unsigned (mulhu), and multiply high signed-unsigned (mulhsu).

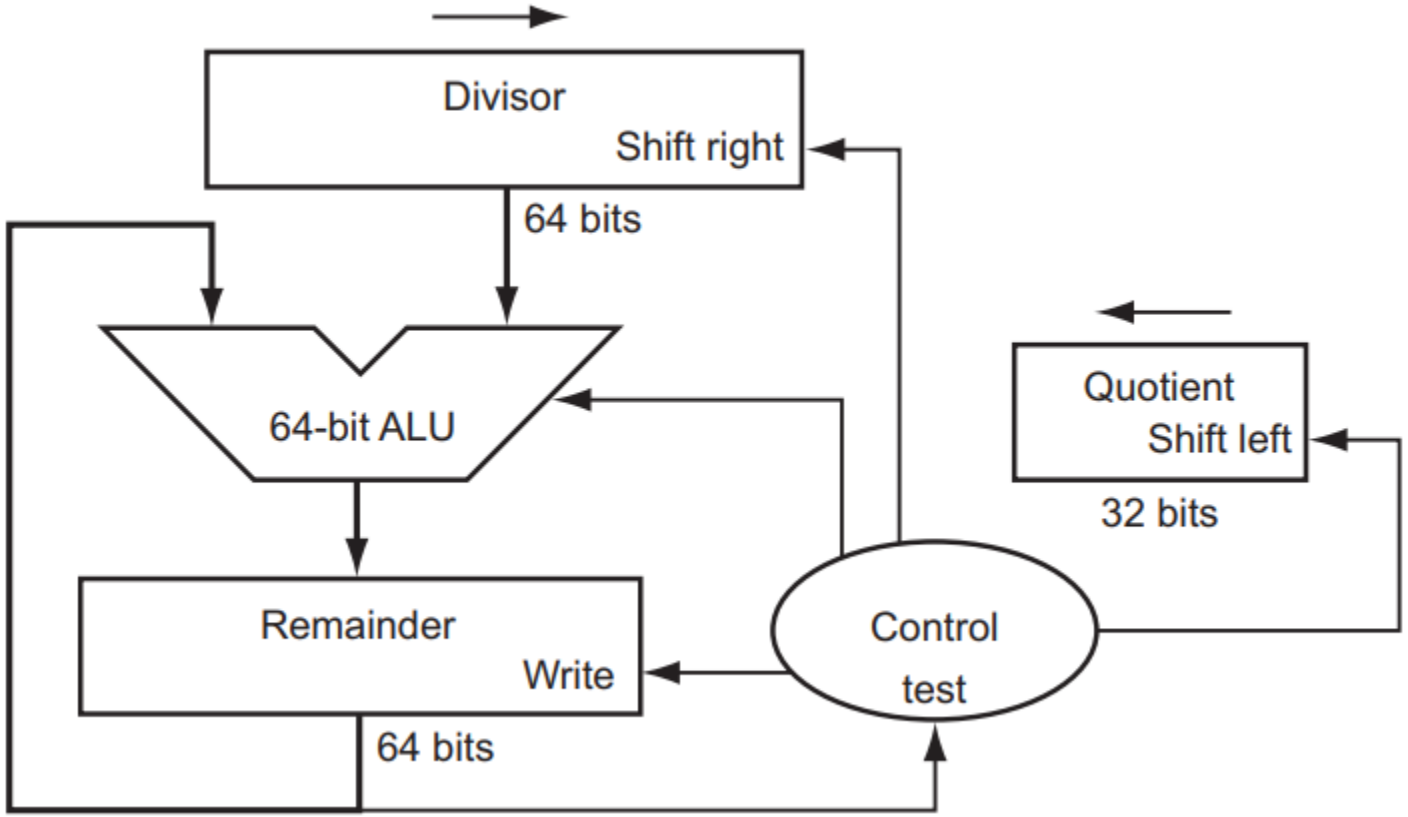
Multiply	`mul x5, x6, x7`	`x5 = x6 × x7`	Lower 32 bits of 64-bit product
Multiply high	`mulh x5, x6, x7`	`x5 = (x6 × x7) >> 32`	Upper 32 bits of 64-bit signed product
Multiply high, unsigned	`mulhu x5, x6, x7`	`x5 = (x6 × x7) >> 32`	Upper 32 bits of 64-bit unsigned product
Multiply high, signed-unsigned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

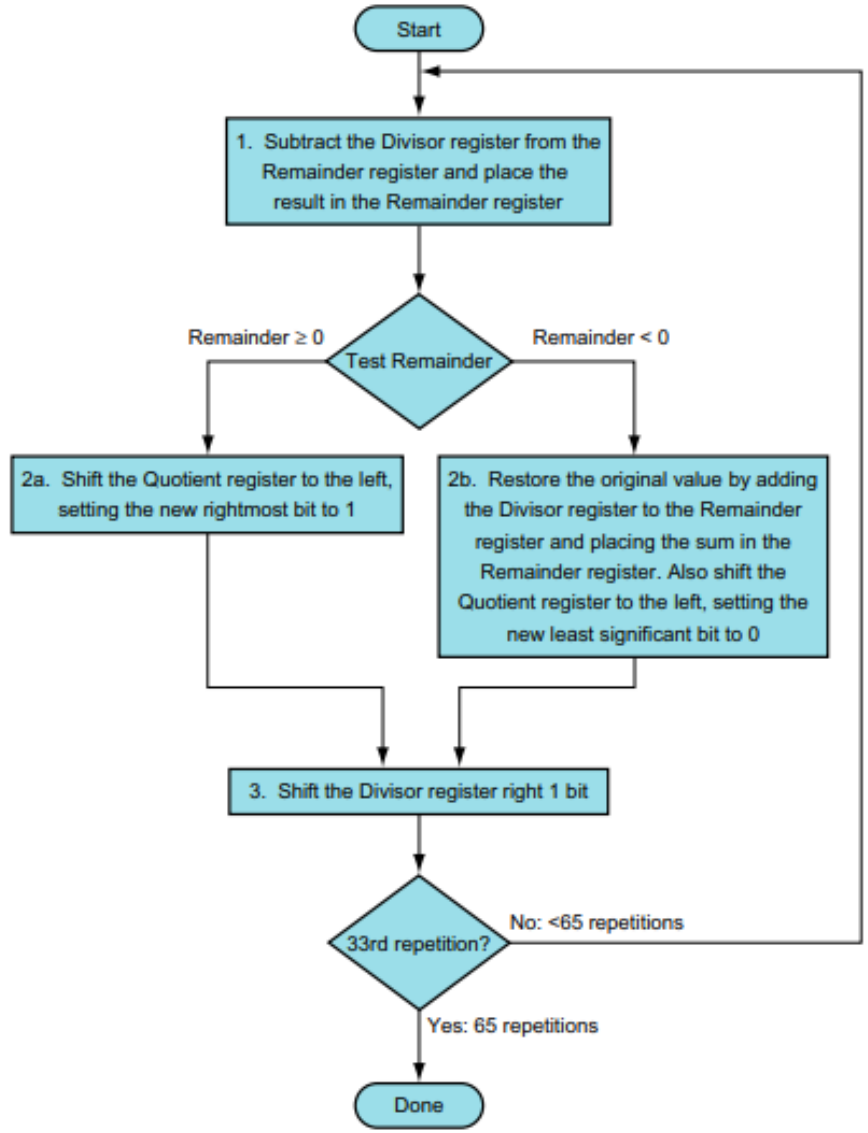
- divisor A number that the dividend is divided by
- quotient The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.
- remainder The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

$$\begin{array}{rrl}
 & 1001_{\text{ten}} & \text{Quotient} \\
\text{Divisor } 1000_{\text{ten}} & \overline{)\,1001010_{\text{ten}}} & \text{Dividend} \\
 & \underline{-1000} & \\
 & 10 & \\
 & 101 & \\
 & 1010 & \\
 & \underline{-1000} & \\
 & 10_{\text{ten}} & \text{Remainder}
\end{array}$$

A Division Algorithm and Hardware




Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: <65 repetitions
Yes: 65 repetitions
Done

- Using a 4-bit version of the algorithm try dividing 7ten by 2ten, or 0000 0111two by 0010two

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	⓪000 0011
	2a: Rem ≥ 0 ⟹ SLL Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	⓪000 0001
	2a: Rem ≥ 0 ⟹ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Signed Division

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

- The first case

$$+7 \div +2: \text{Quotient} = +3, + \text{Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$$-7 \div +2\text{: Quotient} = -3$$

Rewriting our basic formula to calculate the remainder:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3x + 2)\\ &= -7 - (-6) = \mathbf{-1}\end{aligned}$$

So,

$$-7 \div +2\text{: Quotient} = -3,\ \text{Remainder} = -1$$

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

- The reason the answer isn't a quotient of −4 and a remainder of +1, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor!

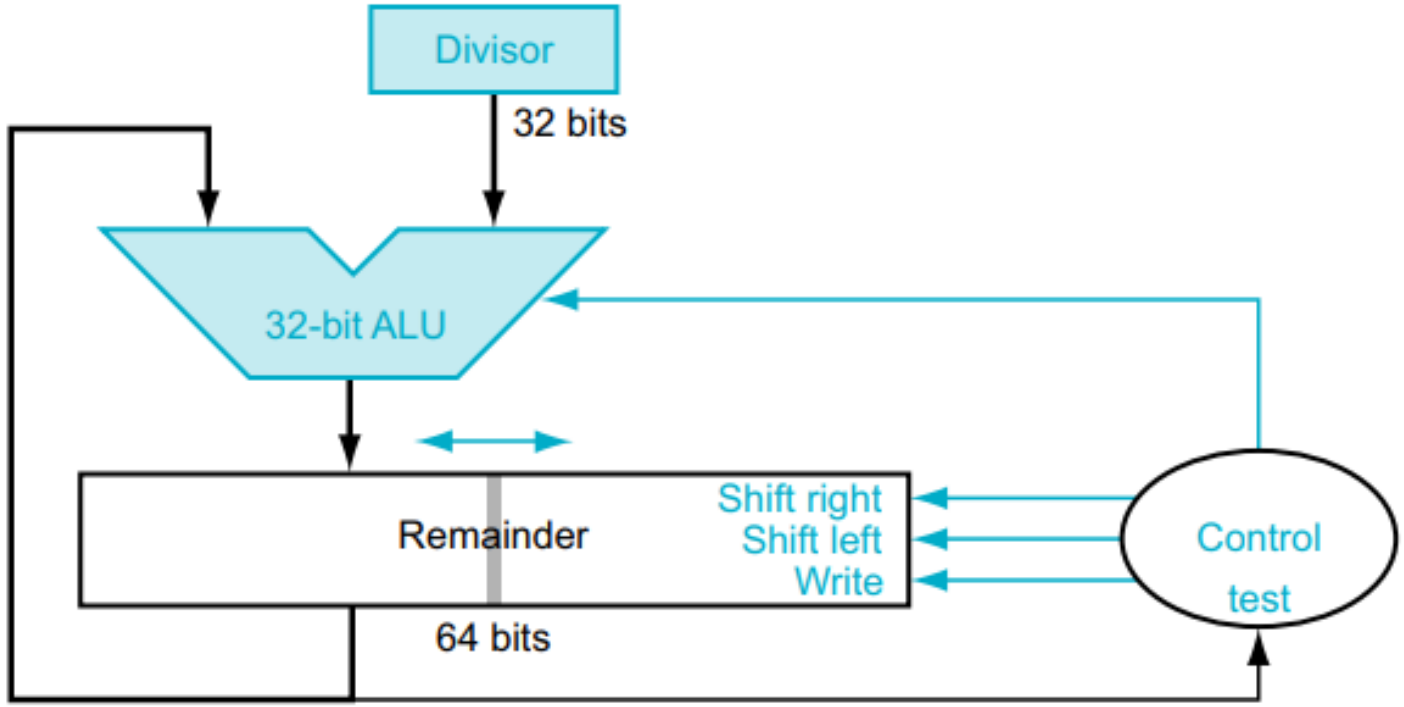
$$-(x \div y) \neq (-x) \div y$$

- dividend and remainder must have identical signs, no matter what the signs of the divisor and quotient

$$+7 \div -2\text{: Quotient} = -3,\ \text{Remainder} = +1$$
$$-7 \div -2\text{: Quotient} = +3,\ \text{Remainder} = -1$$

An improved version of the division hardware.

- The Divisor register, ALU, and Quotient register are all 32 bits wide



Divide in RISC-V

- The only requirement is a 64-bit register that can shift left or right and a 32-bit ALU that adds or subtracts
- To handle both signed integers and unsigned integers, RISC-V has two instructions for division and two instructions for remainder: divide (div), divide unsigned (divu), remainder (rem), and remainder unsigned (remu)

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

Floating Point

- scientific notation A notation that renders numbers with a single digit to the left of the decimal point
- normalized A number in floating-point notation that has no leading 0s.
- $1.0_{ten} \times 10{-}9$ is in normalized scientific notation, but $0.1_{ten} \times 10{-}8$ and $10.0_{ten} \times 10{-}10$ are not.
- floating point Computer arithmetic that represents numbers in which the binary point is not fixed

$$1.xxxxxxxxx_{two} \times 2^{yyyy}$$

Adv of Scientific notation

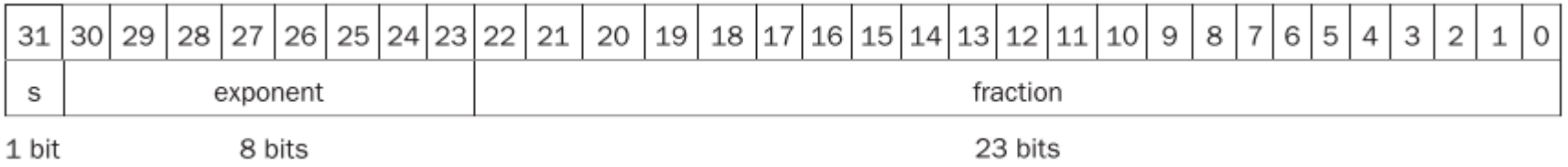
- It simplifies exchange of data that includes floating-point numbers
- it simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form
- it increases the accuracy of the numbers that can be stored in a word, since real digits to the right of the binary point replace the unnecessary leading 0

Floating-Point Representation

- fraction The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the mantissa.
- exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.
- increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented

- In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$



- overflow (floatingpoint) A situation in which a positive exponent becomes too large to fit in the exponent field.
- underflow (floatingpoint) A situation in which a negative exponent becomes too large to fit in the exponent field.
- double precision A floating-point value represented in a 64-bit doubleword
- single precision A floating-point value represented in a 32-bit word

- RISC-V double precision allows numbers almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$
- advantage is its greater precision because of the much larger fraction

Exceptions and Interrupts

- exception Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow, for example.
- interrupt An exception that comes from outside of the processor. (Some architectures use the term interrupt for all exceptions.)
- RISC-V computers do not raise an exception on overflow or underflow; instead, software can read the floating-point control and status register (fcsr) to check whether overflow or underflow has occurred

IEEE 754 Floating-Point Standard

- IEEE 754 makes the leading 1 bit of normalized binary numbers implicit.
- The fraction part is referred as significand

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

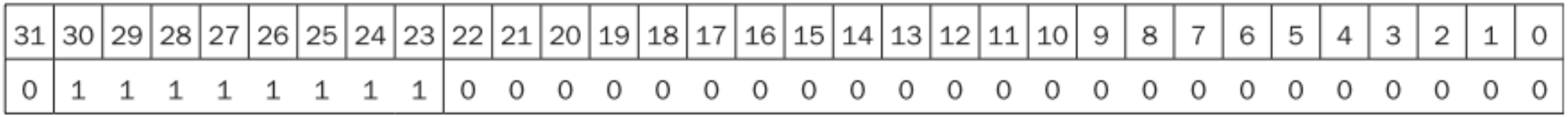
$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E$$

IEEE 754 encoding of floating-point numbers.

- IEEE 754 even has a symbol for the result of invalid operations, such as 0/0 or subtracting infinity from infinity. This symbol is NaN, for Not a Number
- The purpose of NaNs is to allow programmers to postpone some tests and decisions to a later time in the program when they are convenient

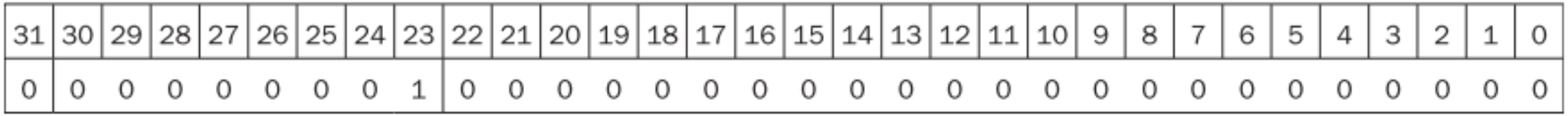
Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

- $1.0_{two} \times 2^{-1}$ would be represented in a single precision as



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- The value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- The desirable notation must therefore represent the most negative exponent as 00 … 00two and the most positive as 11 … 11two.
- This convention is called biased notation, with the bias being the number subtracted from the normal, unsigned representation to determine the real value
- IEEE 754 uses a bias of 127 for single precision, so an exponent of −1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$, and +1 is represented by 1 + 127, or $128_{ten} = 1000\ 0000_{two}$

- The exponent bias for double precision is 1023
- Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} \quad \text{Bias})}$$

The range of single precision numbers is then from as small as

$$\pm 1.00000000000000000000000_{\text{two}} \times 2^{-126}$$

to as large as

$$\pm 1.11111111111111111111111_{\text{two}} \times 2^{+127}.$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

- Show the IEEE 754 binary representation of the number −0.75ten in single and double precision

The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$ yields

$$(-1)^1 \times (1 + .1000\,0000\,0000\,0000\,0000\,000_{two}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{ten} is then

Converting Binary to Decimal Floating Point

- What decimal number does this single precision float represent?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- The sign bit is 1, the exponent field contains 129, and the fraction field contains

$1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$
\begin{aligned}
(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\
&= -1 \times 1.25 \times 2^2 \\
&= -1.25 \times 4 \\
&= -5.0
\end{aligned}
$$

Floating-Point Addition

- floating-point addition: $9.999_{ten} \times 10\text{^}1 + 1.610_{ten} \times 10\text{^}{-1}$

$$1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$$

$$0.016 \times 10^{1}$$

$$\begin{array}{rr} & 9.999_{ten} \\ + & 0.016_{ten} \\ \hline & 10.015_{ten} \end{array}$$

$$10.015_{\text{ten}} \times 10^{1} = 1.0015_{\text{ten}} \times 10^{2}$$

$$1.0015_{\text{ten}} \times 10^{2}$$

$$1.002_{\text{ten}} \times 10^{2}$$

Binary Floating-Point Addition

- adding the numbers 0.5_{ten} and -0.4375_{ten}

$$
\begin{aligned}
0.5_{ten} &= 1/2_{ten} &&= 1/2^{1}_{ten} \\
&= 0.1_{two} &&= 0.1_{two} \times 2^{0} &&= 1.000_{two} \times 2^{-1} \\
-0.4375_{ten} &= -7/16_{ten} &&= -7/2^{4}_{ten} \\
&= -0.0111_{two} &&= -0.0111_{two} \times 2^{0} &&= -1.110_{two} \times 2^{-2}
\end{aligned}
$$

$$-1.110_{\text{two}} \times 2^{-2} = -0.111_{\text{two}} \times 2^{-1}$$

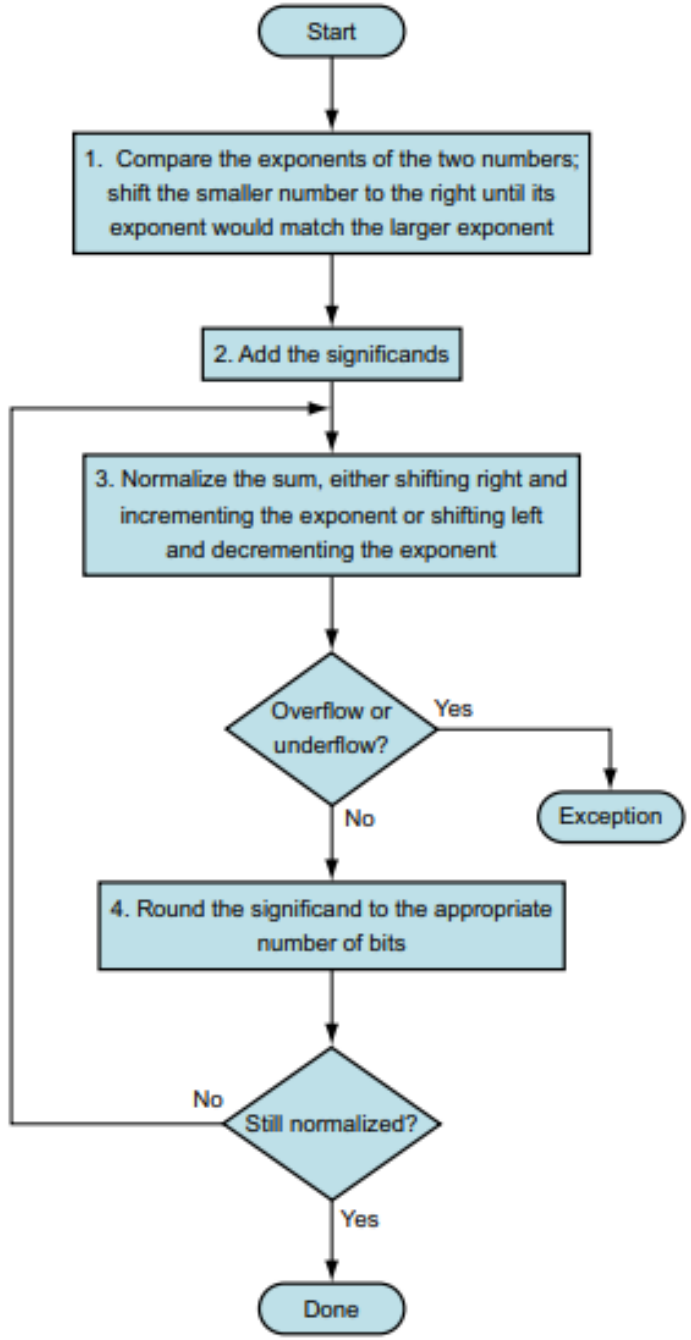
$$1.000_{\text{two}} \times 2^{-1} + (-0.111_{\text{two}} \times 2^{-1}) = 0.001_{\text{two}} \times 2^{-1}$$

$$\begin{aligned} 0.001_{\text{two}} \times 2^{-1} &= 0.010_{\text{two}} \times 2^{-2} = 0.100_{\text{two}} \times 2^{-3} \\ &= 1.000_{\text{two}} \times 2^{-4} \end{aligned}$$

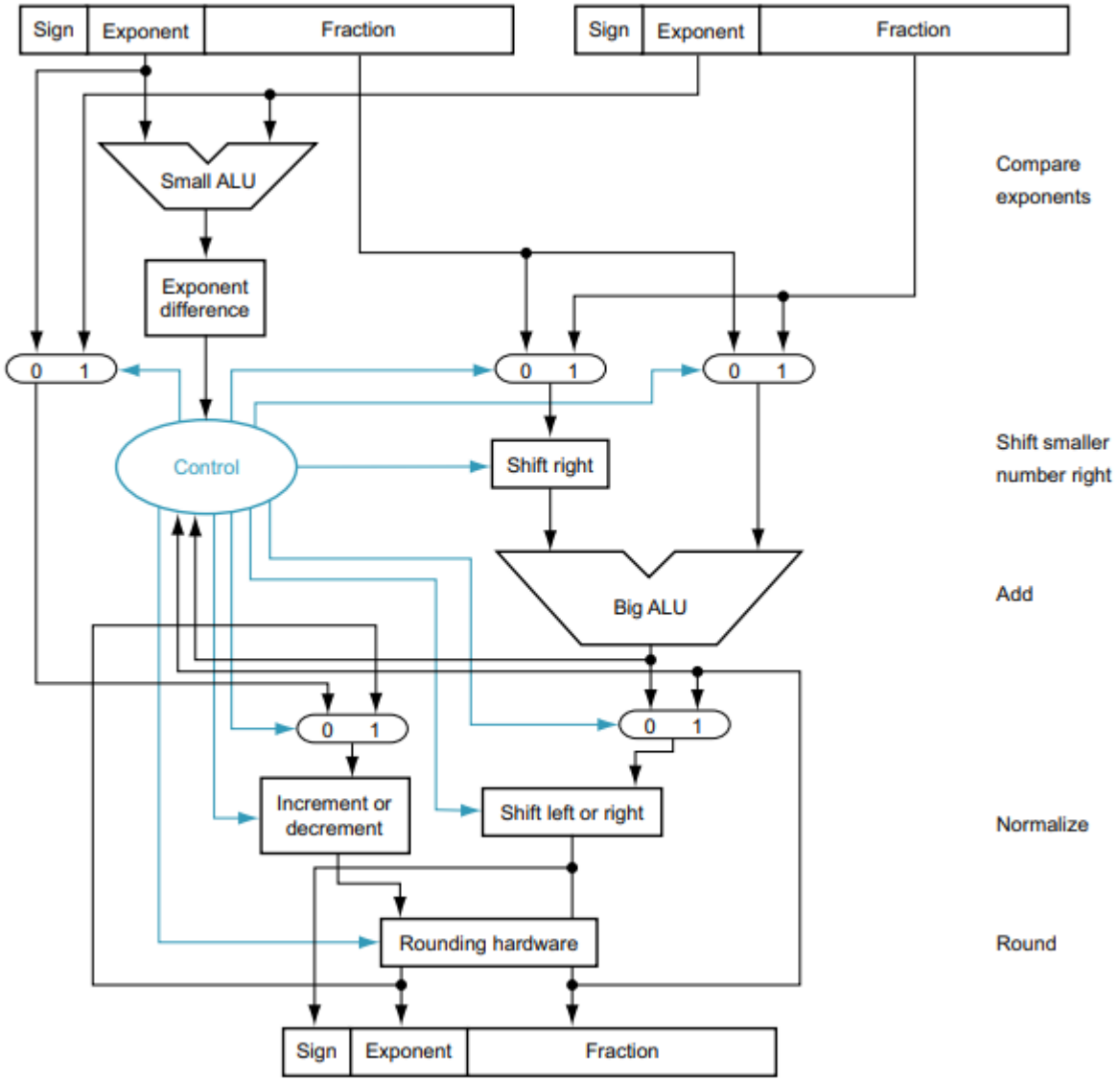
$$1.000_{\text{two}} \times 2^{-4}$$

$$1.000_{two} \times 2^{-4} = 0.0001000_{two} = 0.0001_{two}$$
$$= 1/2^{4}_{ten} = 1/16_{ten} = 0.0625_{ten}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.


Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

Block diagram of an arithmetic unit dedicated to floating-point addition



Floating-Point Multiplication

- Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together

$$1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}.$$

- Step 1: biase $\text{New exponent} = 10 + (-5) = 5$
- 10 + 127 = 137, and −5 + 127 = 122
- New exponent 137+ 122= 259
- This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

- Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum
- New exponent = 137 +122 - 127= 259 - 127= 132 = (5 +127)
- Step 2. multiplication of the significands:

$$
\begin{array}{r}
1.110_{ten} \\
\times \quad 9.200_{ten} \\
\hline
0000 \\
0000 \\
2220 \\
9990 \\
\hline
1110000_{ten}
\end{array}
$$

- 10.212000 ten
- If we keep only three digits to the right of the decimal point, the product is 10.212 × 10^5 .
- Step 3. normalize product:

$$10.212_{\text{ten}} \times 10^5 = 1.0212_{\text{ten}} \times 10^6$$

- check for overflow and underflow

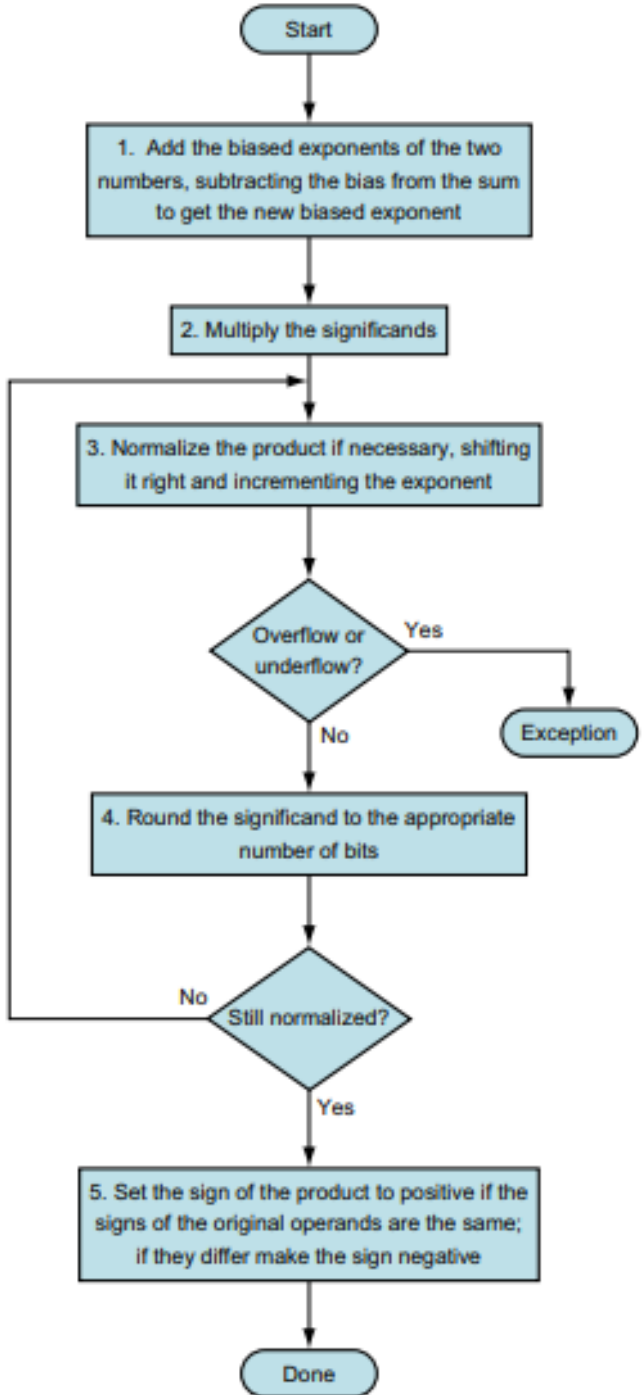
- Step 4:round the number
- We assumed that the significand is only four digits long (excluding the sign)

$$1.0212_{ten} \times 10^{6}$$

$$1.021_{ten} \times 10^{6}$$

- Step 5.: Sign of the product
- The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise, it's negative. Hence, the product is

$$+1.021_{\text{ten}} \times 10^{6}$$


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Binary Floating-Point Multiplication

- Multiply the numbers 0.5_{ten} and -0.4375_{ten},

  $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

-

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \times \quad 1.110_{two} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{two} \end{array}$$

The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{\text{two}} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence, the product is

$$-1.110_{\text{two}} \times 2^{-3}$$

Converting to decimal to check our results:

$$\begin{aligned} -1.110_{\text{two}} \times 2^{-3} &= -0.001110_{\text{two}} = -0.00111_{\text{two}} \\ &= -7/2^{5}_{\text{ten}} = -7/32_{\text{ten}} = -0.21875_{\text{ten}} \end{aligned}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

Floating-Point Instructions in RISC-V

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.

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

The RISC-V code to load two single precision numbers from memory, add them, and then store the sum

- A single precision register is just the lower half of a double-precision register

```
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
```

- The floating-point instructions use the same format as their integer counterparts: loads use the I-type format, stores use the S-type format, and arithmetic instructions use the R-type format
- One issue that architects face in supporting floating-point arithmetic is whether to select the same registers used by the integer instructions or to add a special set for floating point
- some computers convert all sized operands in registers into a single internal format

Compiling a Floating-Point C Program into RISC-V Assembly Code

Let's convert a temperature in Fahrenheit to Celsius:

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

Assume that the floating-point argument `fahr` is passed in `f10` and the result should also go in `f10`. What is the RISC-V assembly code?

```
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
```

Compiling Floating-Point C Procedure with Two-Dimensional Matrices into RISC-V

- C = C + A * B
- assume C, A, and B are all square matrices with 32 elements in each dimension.

```
void mm (double c[][], double a[][], double b[][])
{
        size_t i, j, k;
         for (i = 0; i < 32; i = i + 1)
          for (j = 0; j < 32; j = j + 1)
           for (k = 0; k < 32; k = k + 1)
              c[i][j] = c[i][j] + a[i][k] *b[k][j];
}
```

- The array starting addresses are parameters, so they are in x10, x11, and x12. Assume that the integer variables are in x5, x6, and x7, respectively. What is the RISC-V assembly code for the body of the procedure?

```
    mm:...
       addi x28, x0, 32    //  x28 = 32 (row size/loop end)
       addi x5, x0, 0    //  i = 0; initialize 1st for loop
L1:    addi x6, x0, 0    //  j = 0; initialize 2nd for loop
L2:    addi x7, x0, 0    //  k = 0; initialize 3rd for loop
```

```
slli  x30, x5, 5    //  x30 = i * 2^5(size of row of c)
add  x30, x30, x6     //  x30 = i * size(row) + j
slli  x30, x30, 3     //  x30 = byte offset of [i][j]
add  x30, x10, x30  //  x30 = byte address of c[i][j]
fld  f0, 0(x30)     //  f0 = 8 bytes of c[i][j]
```

```
L3: slli  x29, x7, 5     //  x29 = k * 2^5(size of row of b)
    add   x29, x29, x6   //  x29 = k * size(row) + j
    slli  x29, x29, 3    //  x29 = byte offset of [k][j]
    add   x29, x12, x29  //  x29 = byte address of b[k][j]
    fld   f1, 0(x29)     //  f1 = 8 bytes of b[k][j]
```

```
slli  x29, x5, 5     //  x29 = i * 2^5(size of row of a)
add   x29, x29, x7   //  x29 = i * size(row) + k
slli  x29, x29, 3    //  x29 = byte offset of [i][k]
add   x29, x11, x29  //  x29 = byte address of a[i][k]
fld   f2, 0(x29)     //  f2 = a[i][k]
```

```
fmul.d  f1, f2, f1  // f1 = a[i][k] * b[k][j]
fadd.d  f0. f0. f1  // f0 = c[i][j] + a[i][k] * b[k][j]

addi  x7, x7, 1    //  k = k + 1
bltu  x7, x28, L3  //  if (k < 32) go to L3
fsd   f0, 0(x30)   //  c[i][j] = f0

addi  x6, x6, 1    //  j = j + 1
bltu  x6, x28, L2  //  if (j < 32) go to L2
addi  x5, x5, 1    //  i = i + 1
bltu  x5, x28, L1  //  if (i < 32) go to L1
. . .
```

Accurate Arithmetic

- guard The first of two extra bits kept on the right during intermediate calculations of floatingpoint numbers; used to improve rounding accuracy
- round Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format. It is also the name of the second of two extra bits kept on the right during intermediate floatingpoint calculations, which improves rounding accuracy.

Add $2.56_{ten} \times 10^0$ to $2.34_{ten} \times 10^2$, assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

$$\begin{array}{r} 2.3400_{ten} \\ +0.0256_{ten} \\ \hline 2.3656_{ten} \end{array}$$

Rounding the sum up with three significant digits yields $2.37_{ten} \times 10^2$.

- Doing this without guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{r} 2.34_{\text{ten}} \\ +0.02_{\text{ten}} \\ \hline 2.36_{\text{ten}} \end{array}$$

The answer is $2.36_{\text{...}} \times 10^2$, off by 1 in the last digit from the sum above.

- units in the last place (ulp) The number of bits in error in the least significant bits of the significand between the actual number and the number that can be represented
- fused multiply add A floating-point instruction that performs both a multiply and an add, but rounds only once after the add.

C type	Java type	Data transfers	Operations
int	int	lw, sw	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
unsigned int	–	lw, sw	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
char	–	lb, sb	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
short	char	lh, sh	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
float	float	flw, fsw	fadd.s, fsub.s, fmul.s, fdiv.s, feq.s, flt.s, fle.s
double	double	fld, fsd	fadd.d, fsub.d, fmul.d, fdiv.d, feq.d, flt.d, fle.d

Parallelism and Computer Arithmetic: Subword Parallelism

- Architects recognized that many graphics and audio applications would perform the same operation on vectors of the data.
- By partitioning the carry chains within a 128-bit adder, a processor could use parallelism to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands.
- The cost of such partitioned adders was small yet the speedups could be large.
- parallelism occurs within a wide word, the extensions are classified as subword parallelism(data level parallelism)