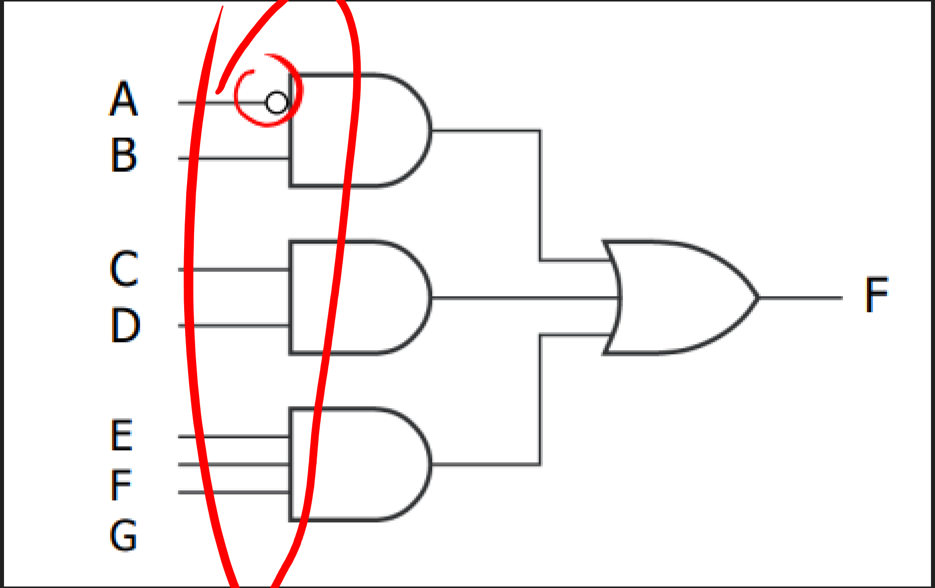
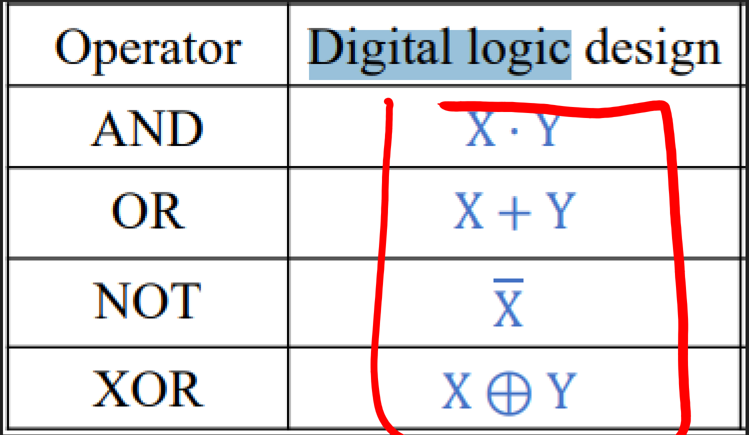
Exam 2

Computer Architecture

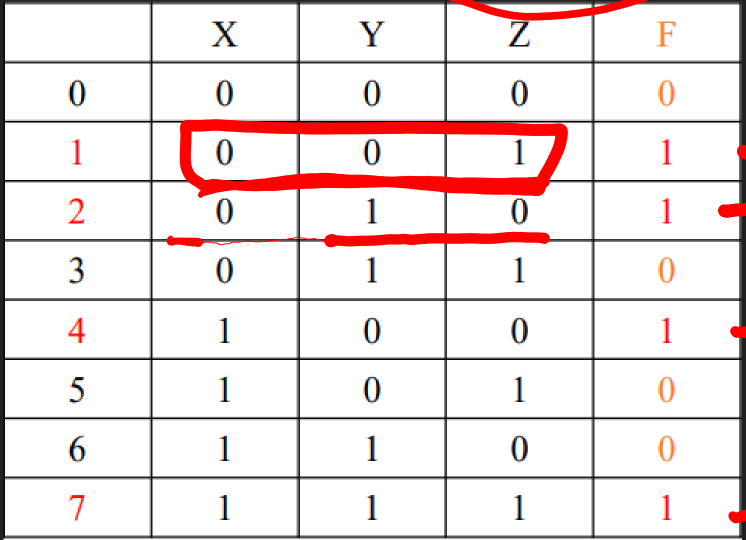
Digital Logic



$\overline{A} \cdot B + C \cdot D + E \cdot F$



Operator	Digital logic design
AND	$X \cdot Y$
OR	$X + Y$
NOT	$\overline{X}$
XOR	$X \oplus Y$

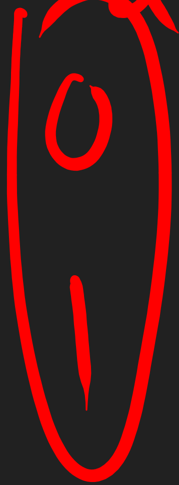


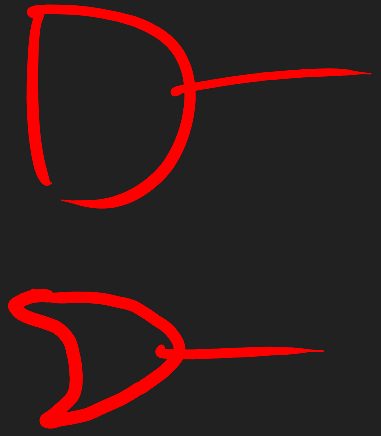
	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

resut.next= a and b a or b

MyHDL stuff

- Implementing small circuits in myHDL.
  - Implement a mux
  - Simple logic gates



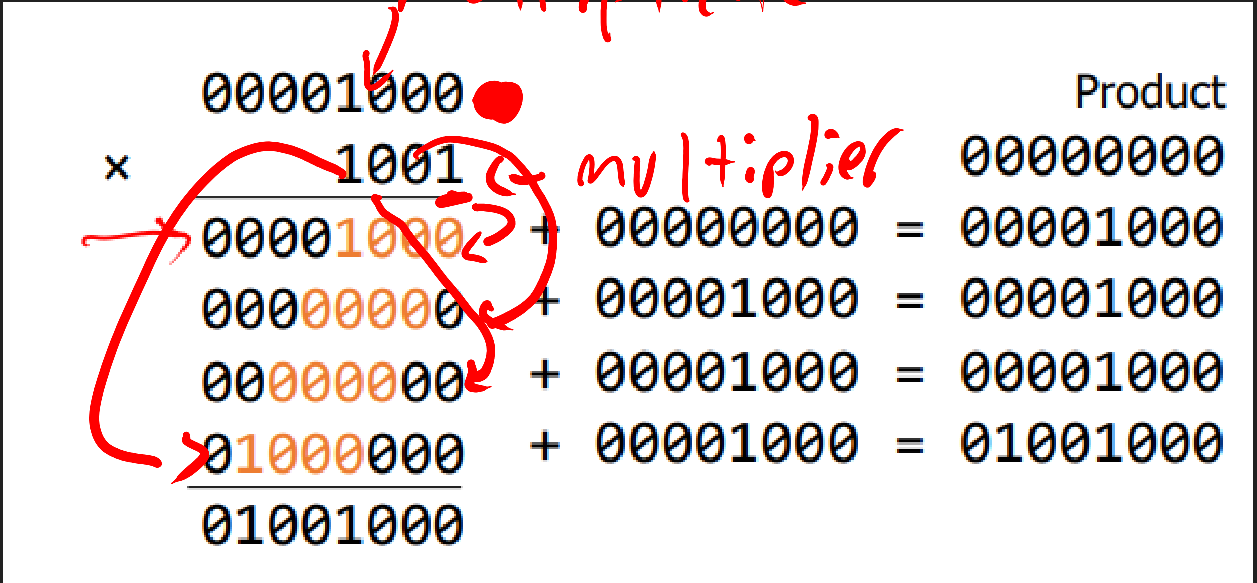


```
mux2(sig)
if sig==1:
    mux2out.next=opt1
else:
```

Multiplication

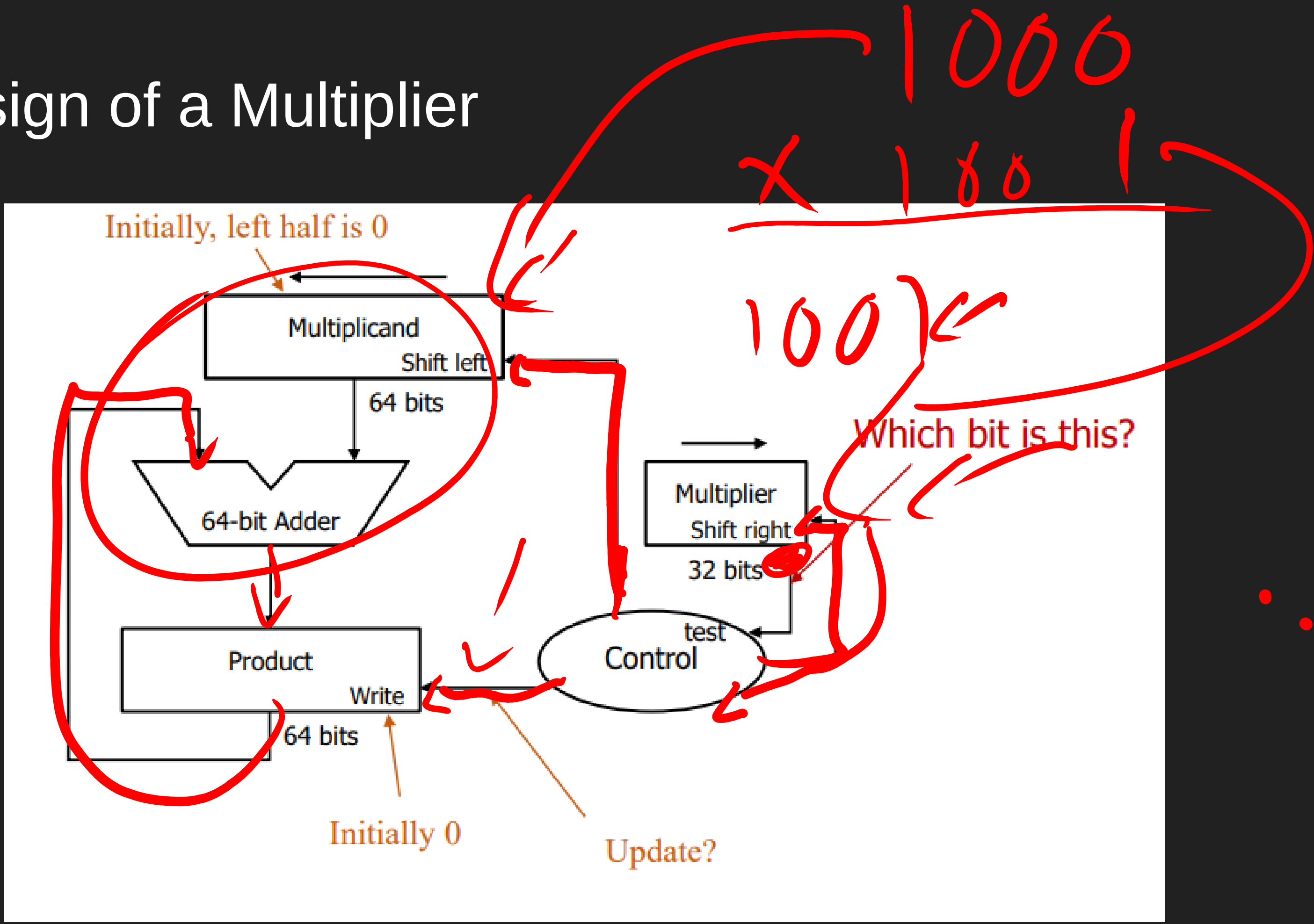
- Take the top number (multiplicand) and write it out each time there is a 1 in the multiplier ex:

multiplicand



```
   00001000                              Product
×      1001  multiplier                  00000000
   00001000  +  00000000  =  00001000
   00000000  +  00001000  =  00001000
   00000000  +  00001000  =  00001000
   01000000  +  00001000  =  01001000
   01001000
```

Design of a Multiplier



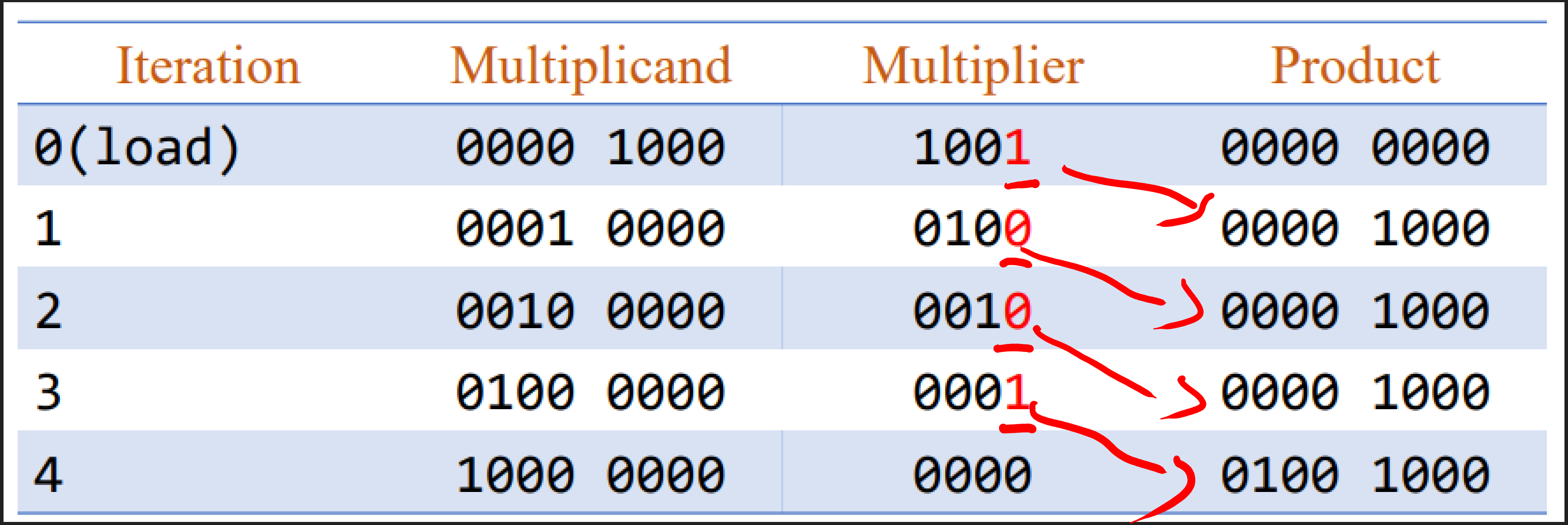
Initially, left half is 0

Which bit is this?

Initially 0

Update?

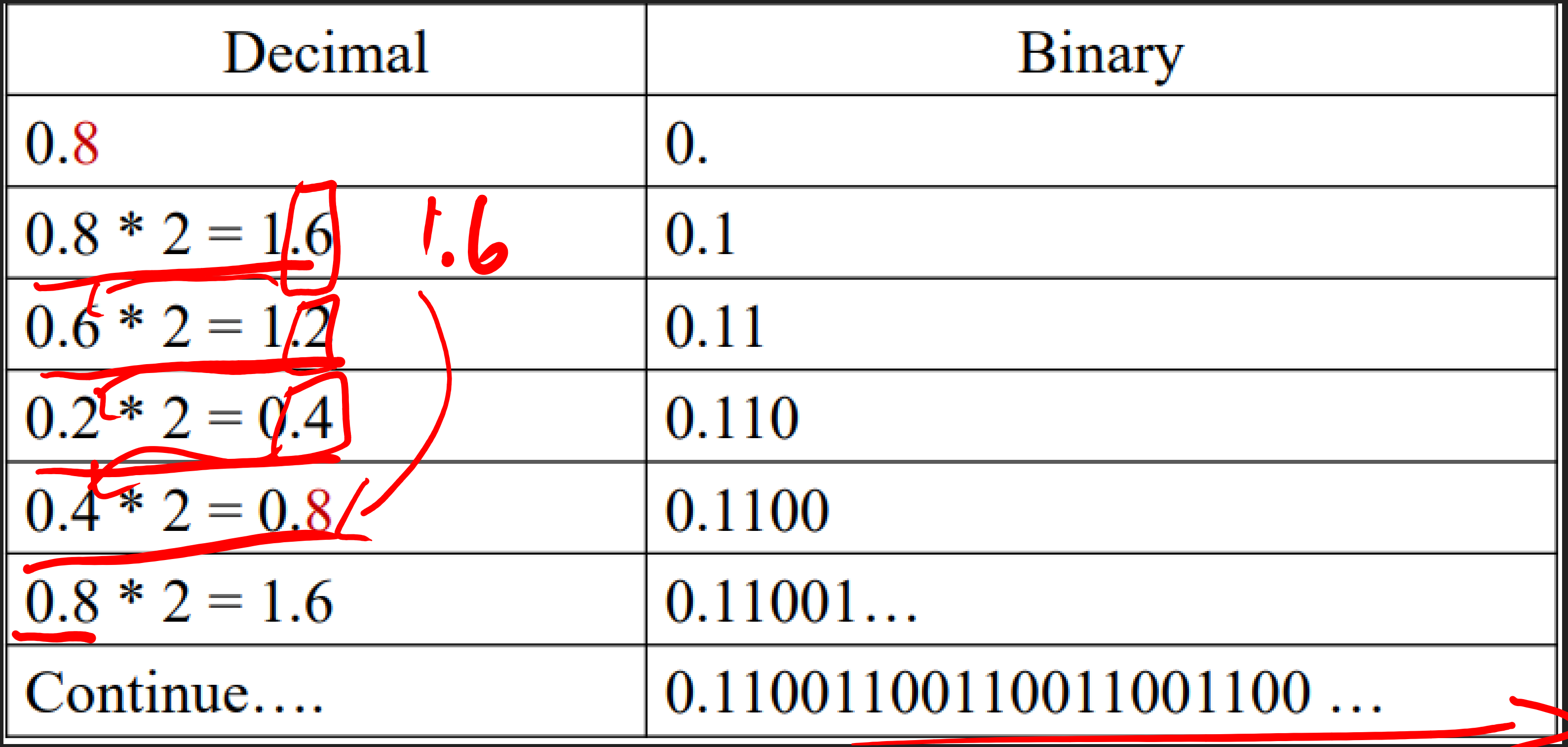
Register Values in a 4-bit Multiplier



Iteration	Multiplicand	Multiplier	Product
0(load)	0000 1000	1001	0000 0000
1	0001 0000	0100	0000 1000
2	0010 0000	0010	0000 1000
3	0100 0000	0001	0000 1000
4	1000 0000	0000	0100 1000

Decimal->Binary

0.



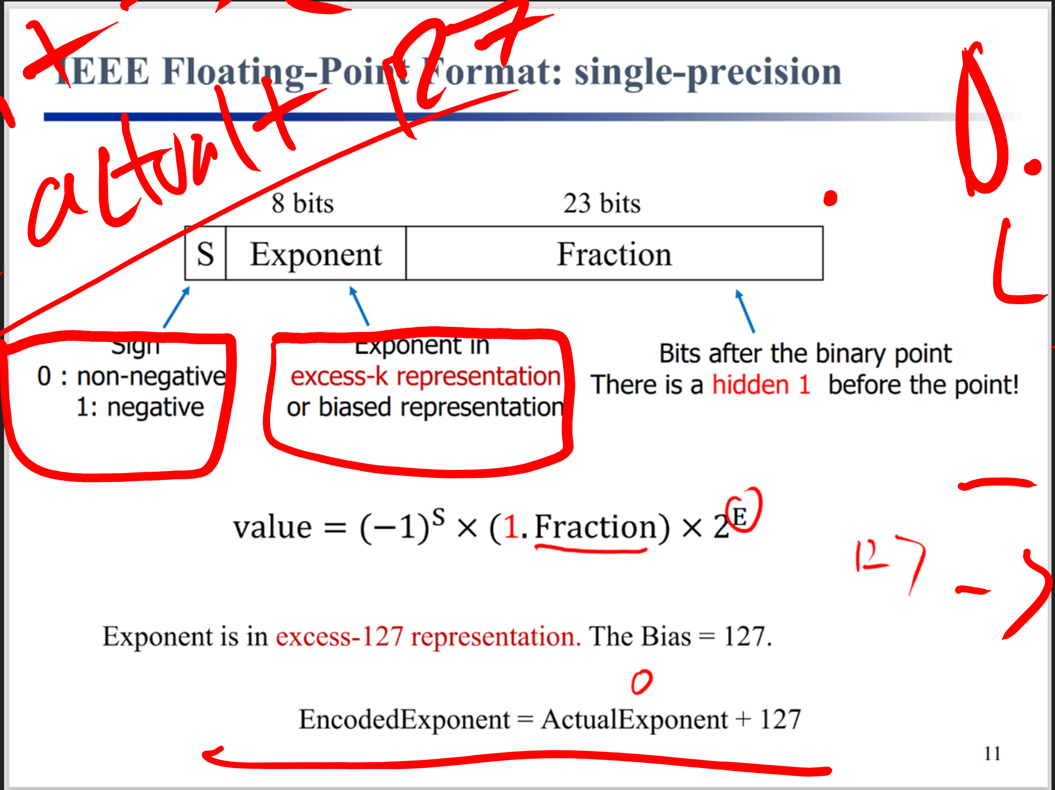
Decimal	Binary
0.8	0.
0.8 * 2 = 1.6	0.1
0.6 * 2 = 1.2	0.11
0.2 * 2 = 0.4	0.110
0.4 * 2 = 0.8	0.1100
0.8 * 2 = 1.6	0.11001…
Continue….	0.11001100110011001100 …

1.6

Floating Point

actual exponent → [-126, 127]

encoded = actual + 127



IEEE Floating-Point Format: single-precision

Sign
0 : non-negative
1: negative

Exponent in
excess-k representation
or biased representation

Bits after the binary point
There is a hidden 1 before the point!

$$\text{value} = (-1)^S \times (1.\underline{\text{Fraction}}) \times 2^{\boxed{E}}$$

Exponent is in excess-127 representation. The Bias = 127.

$$\text{EncodedExponent} = \overset{0}{\text{ActualExponent}} + 127$$

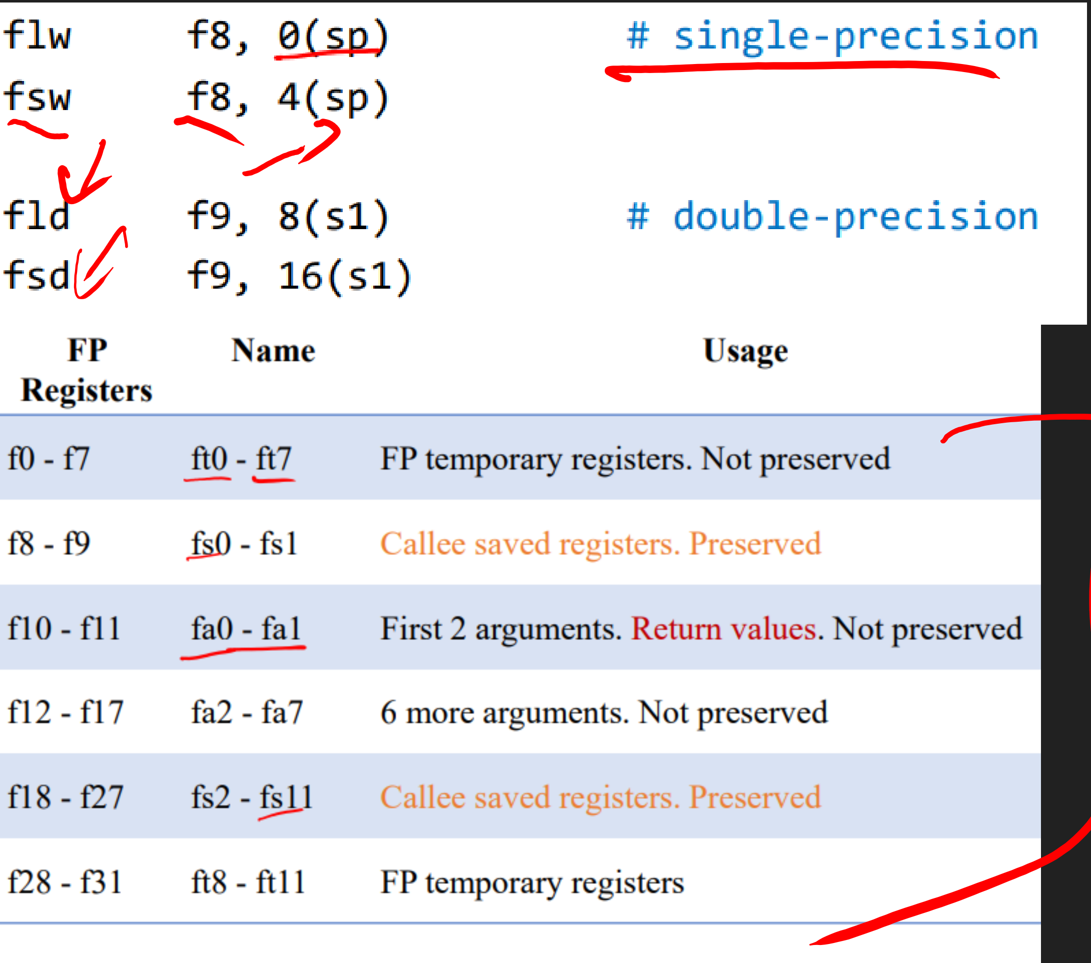
$0.11\boxed{10}$

→ 1.0

$-0.10\boxed{0}$

127 → -0.11

Floating-point Extension RISC-V



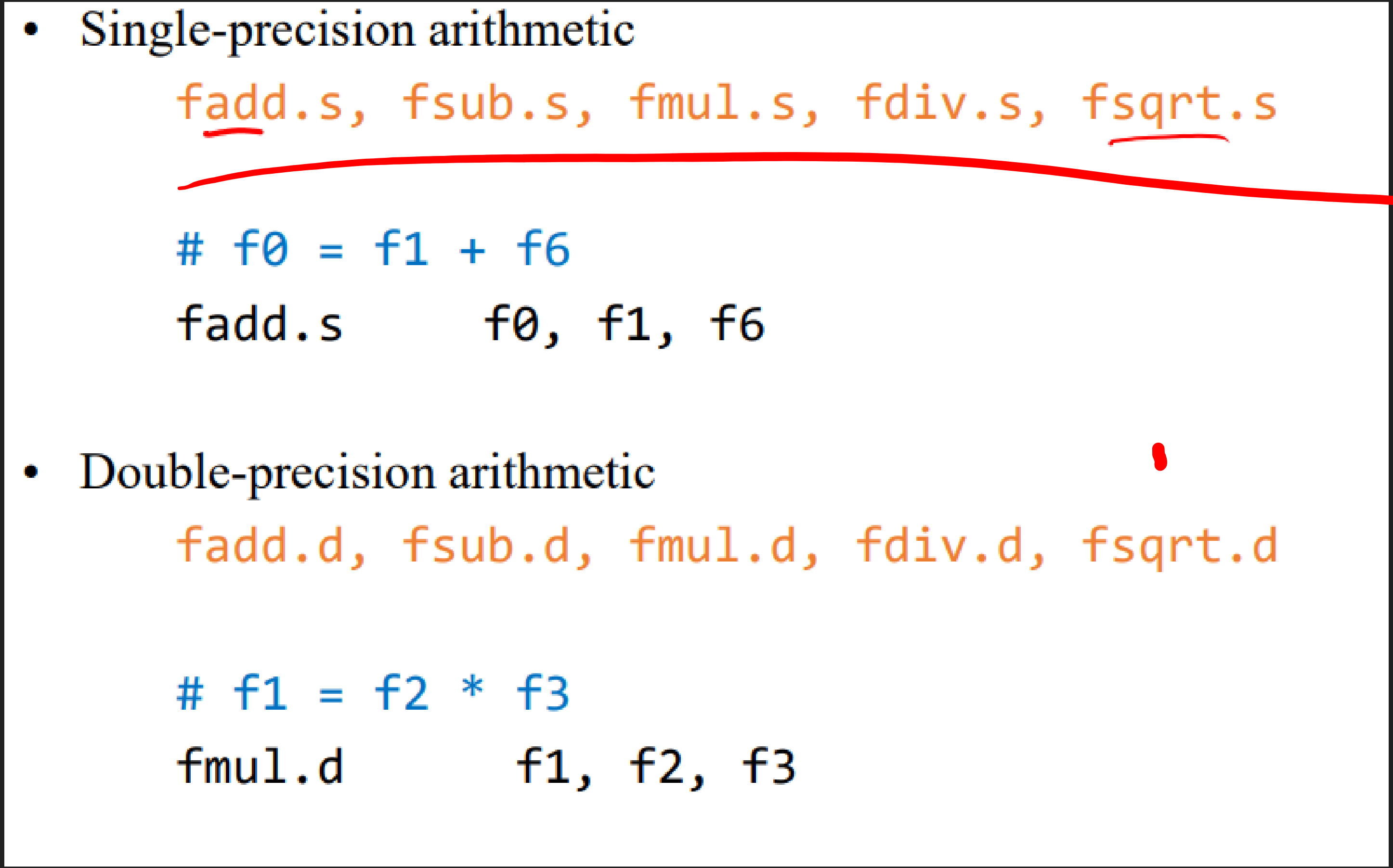
```
flw     f8, 0(sp)          # single-precision
fsw     f8, 4(sp)

fld     f9, 8(s1)          # double-precision
fsd     f9, 16(s1)
```

FP Registers	Name	Usage
f0 - f7	ft0 - ft7	FP temporary registers. Not preserved
f8 - f9	fs0 - fs1	Callee saved registers. Preserved
f10 - f11	fa0 - fa1	First 2 arguments. Return values. Not preserved
f12 - f17	fa2 - fa7	6 more arguments. Not preserved
f18 - f27	fs2 - fs11	Callee saved registers. Preserved
f28 - f31	ft8 - ft11	FP temporary registers

a s

f



- Single-precision arithmetic

```
fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s

# f0 = f1 + f6
fadd.s     f0, f1, f6
```

- Double-precision arithmetic

```
fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d

# f1 = f2 * f3
fmul.d      f1, f2, f3
```

Performance Equations

$$\text{CPU Time} = \text{Clock Cycles} \times \text{Clock Cycle Time}$$

Number of clock cycles

$$\text{CPU Time} = \frac{\text{Clock Cycles}}{\text{Clock Rate}}$$

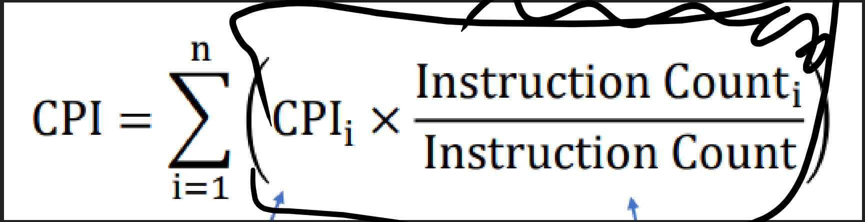
$$\text{Performance} = \frac{1}{\text{CPU Time}}$$

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}}$$

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{CPI}$$

cpu time

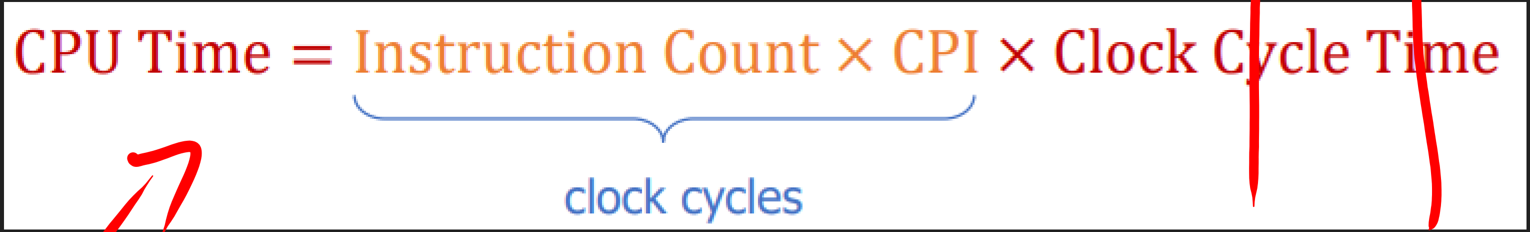
$$n = \frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\frac{1}{\text{ExecutionTime}_x}}{\frac{1}{\text{ExecutionTime}_y}} = \frac{\text{ExecutionTime}_y}{\text{ExecutionTime}_x}$$



$$\text{CPI} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

table

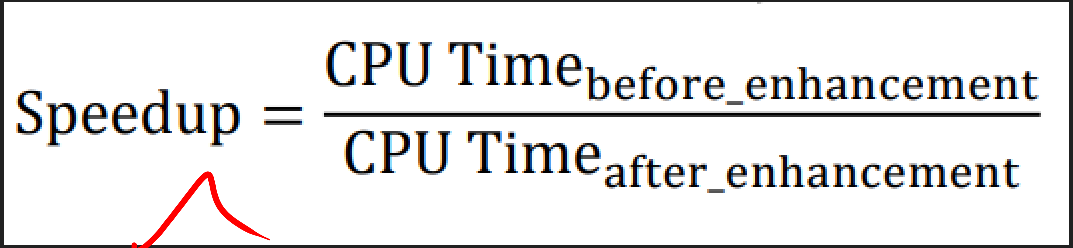
Class	A	B	C



$$\text{CPU Time} = \underbrace{\text{Instruction Count} \times \text{CPI}}_{\text{clock cycles}} \times \text{Clock Cycle Time}$$

clock rate

cycle time



$$\text{Speedup} = \frac{\text{CPU Time}_{\text{before\_enhancement}}}{\text{CPU Time}_{\text{after\_enhancement}}}$$

Processors

S-type = braches

↳ state imm

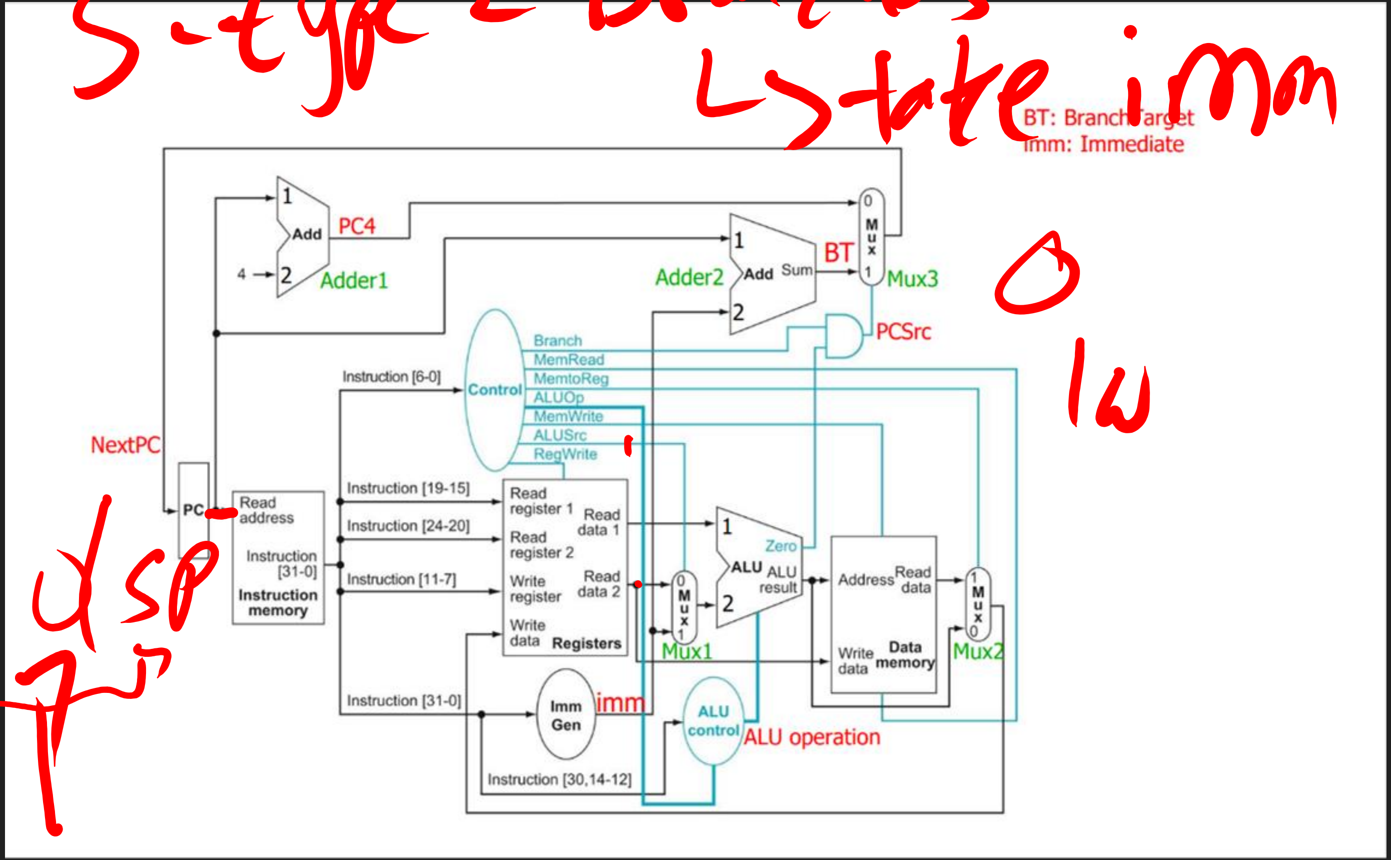
add

sw

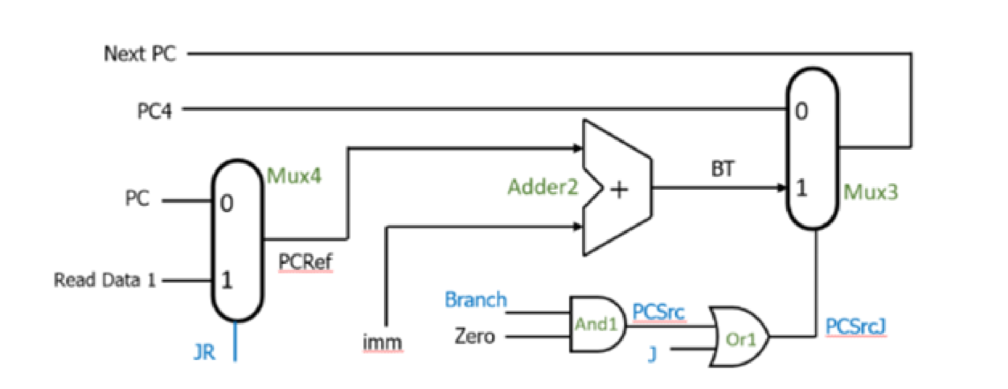
sw s1, 4(sp

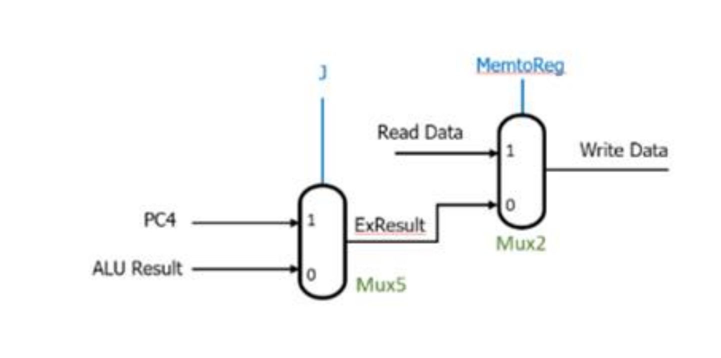
O

lw



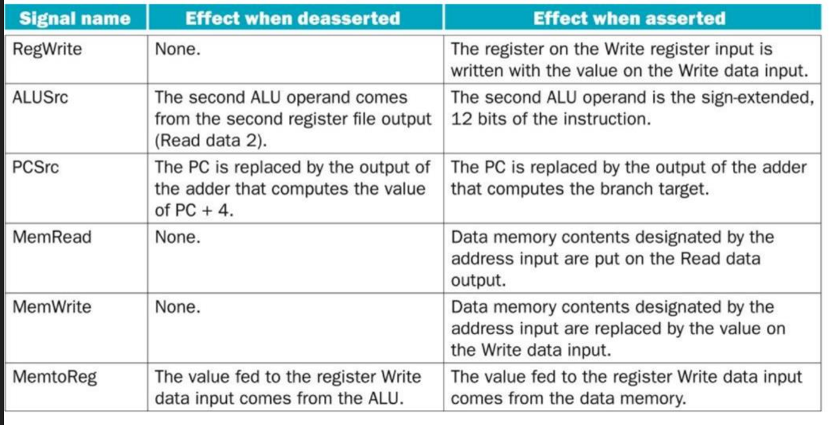
Designing Single Cycle Processors



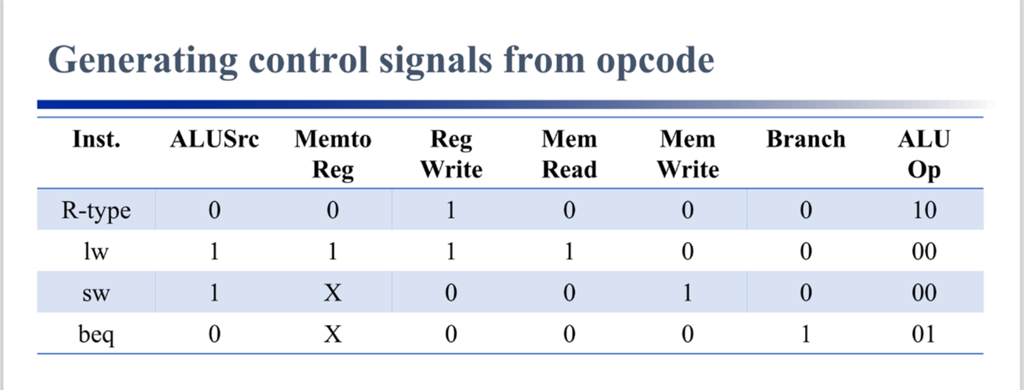


Processor Signals

R-type = add rd, rs1, rs2

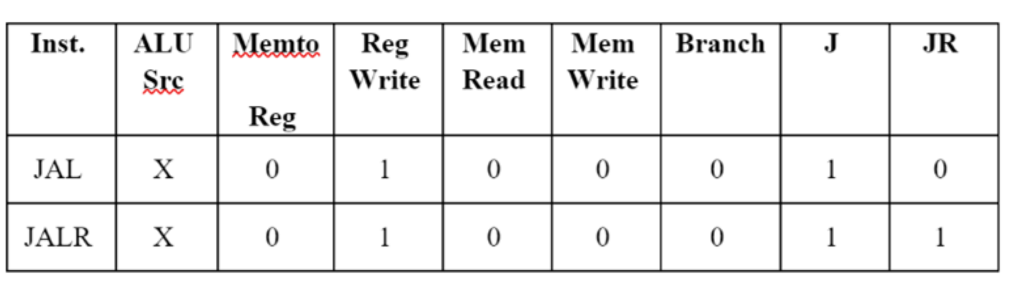


Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.



Generating control signals from opcode

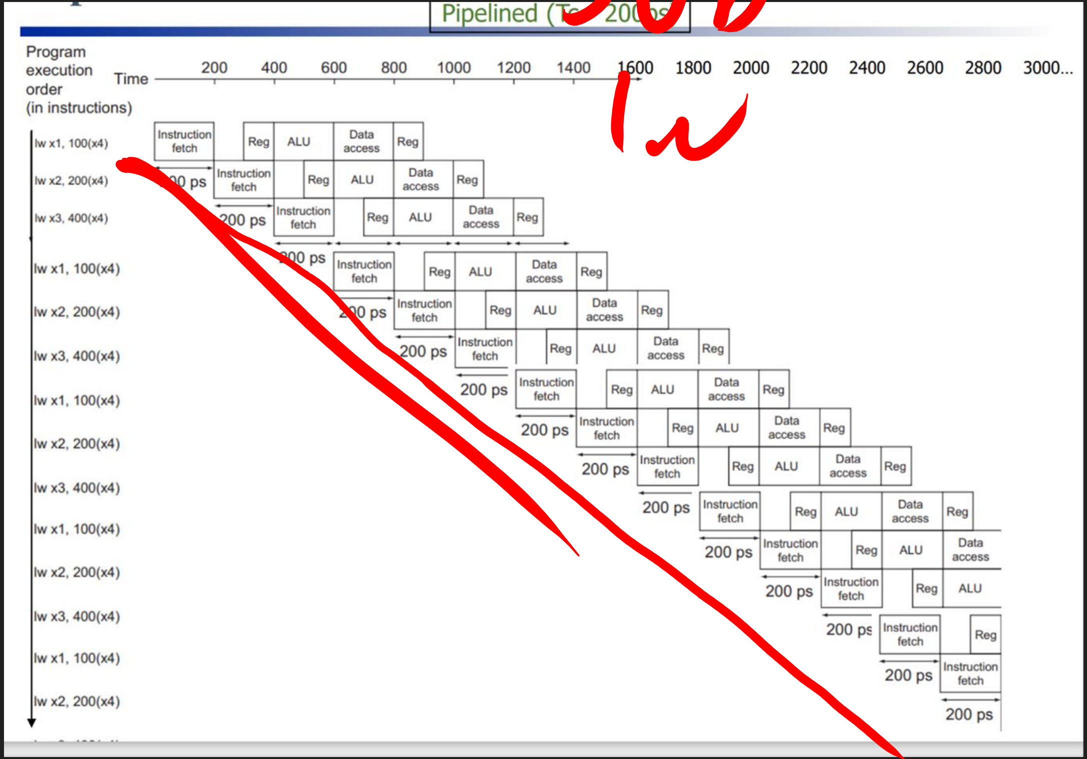
Inst.	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALU Op
R-type	0	0	1	0	0	0	10
lw	1	1	1	1	0	0	00
sw	1	X	0	0	1	0	00
beq	0	X	0	0	0	1	01

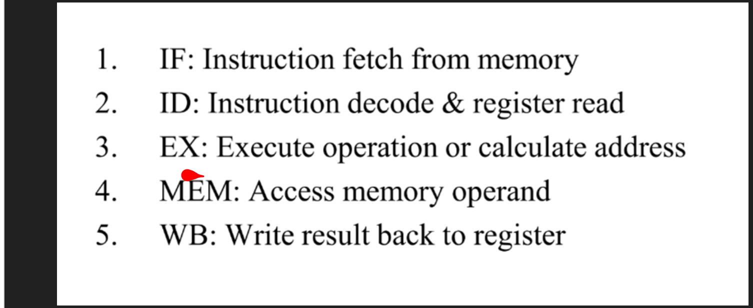


Inst.	ALU Src	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	J	JR
JAL	X	0	1	0	0	0	1	0
JALR	X	0	1	0	0	0	1	1

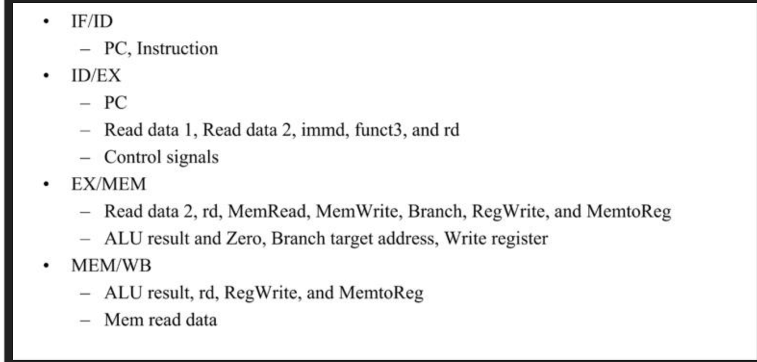
Pipelines

add
Sub
lw





1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register



- IF/ID
  - PC, Instruction
- ID/EX
  - PC
  - Read data 1, Read data 2, immd, funct3, and rd
  - Control signals
- EX/MEM
  - Read data 2, rd, MemRead, MemWrite, Branch, RegWrite, and MemtoReg
  - ALU result and Zero, Branch target address, Write register
- MEM/WB
  - ALU result, rd, RegWrite, and MemtoReg
  - Mem read data

Pipelines

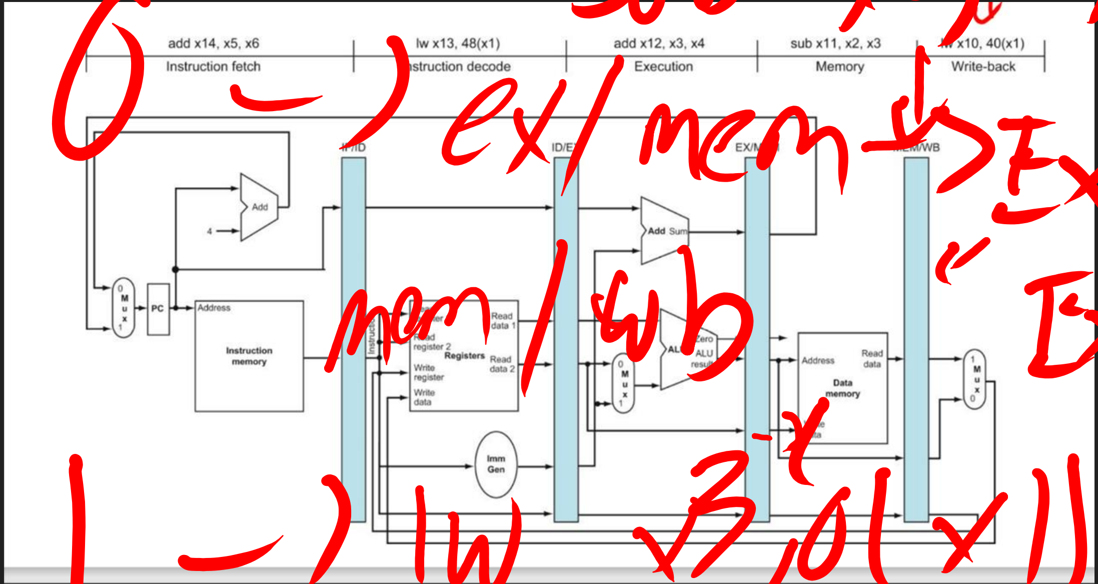
R-type add x1, x2, x3 Ex/mem

sub x2, x1, x3

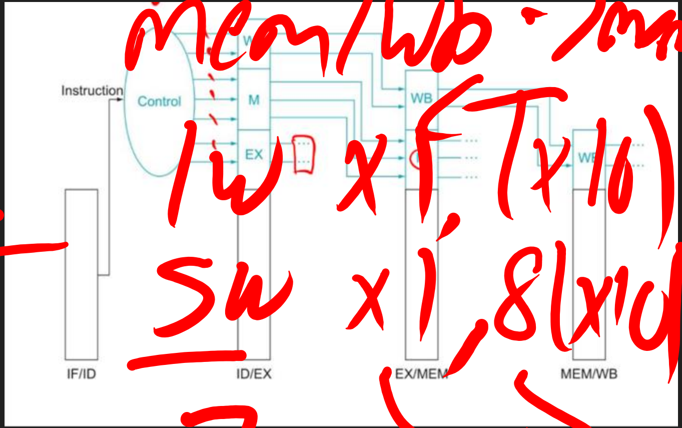
0 -> ex/mem -> Ex

mem/wb

Ex <-



mem/wb -> mem



lw xF, (x10)

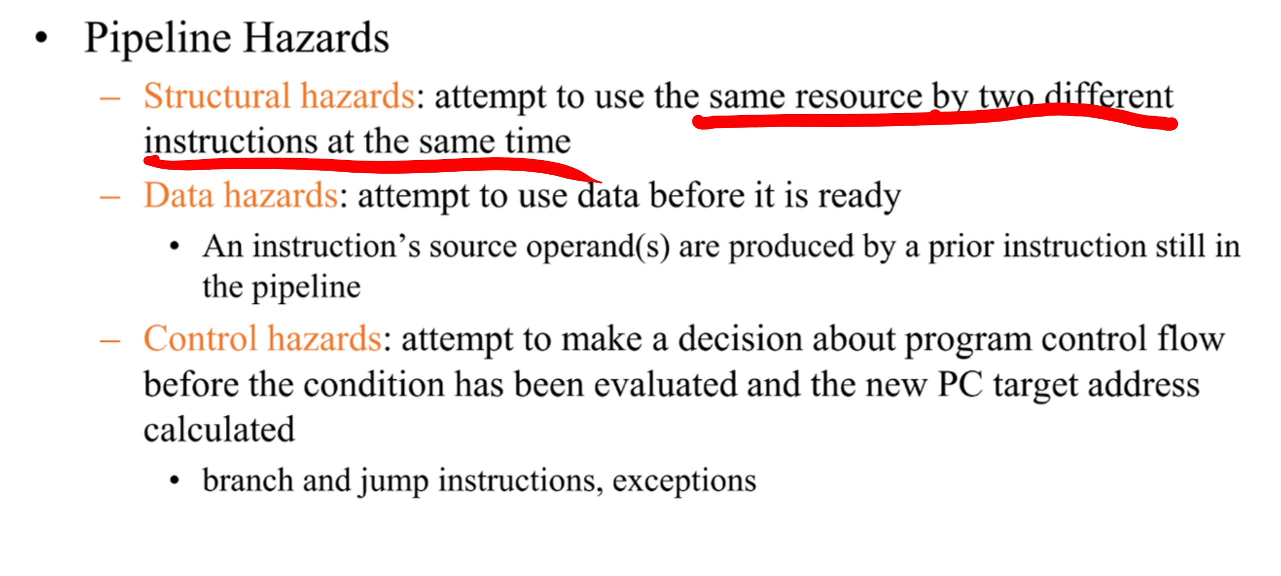
sw x1, 8(x10)

1 -> lw x3, 0(x1)

stall add x1, x2, x3

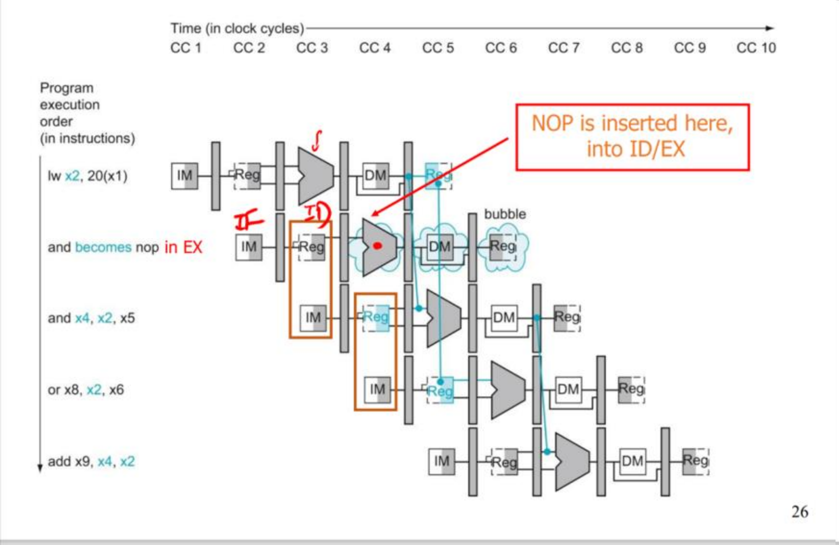
Hazards

mem/wb 2 times

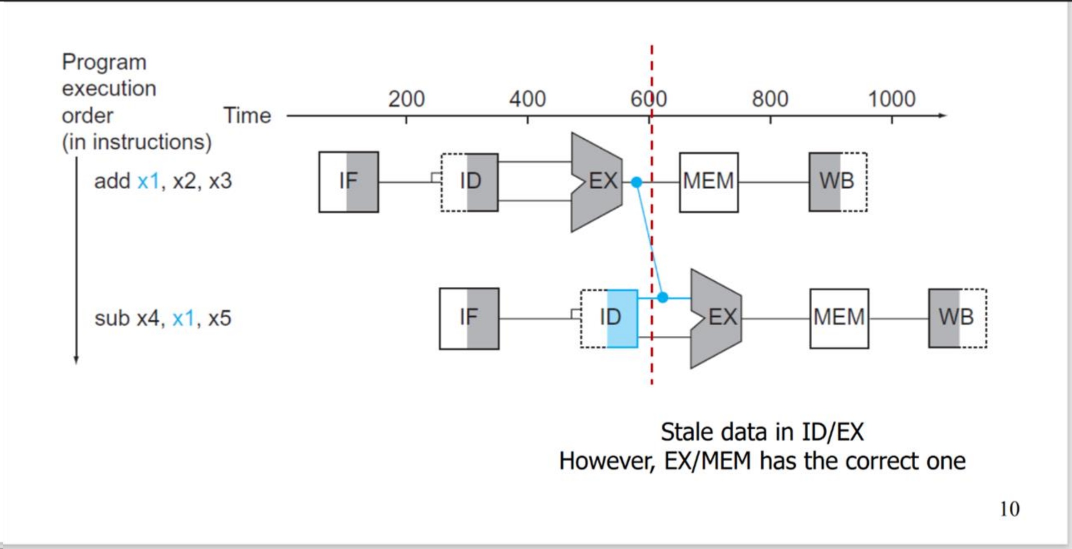


- Pipeline Hazards
  - Structural hazards: attempt to use the same resource by two different instructions at the same time
  - Data hazards: attempt to use data before it is ready
    - An instruction’s source operand(s) are produced by a prior instruction still in the pipeline
  - Control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions

Forwarding



NOP is inserted here, into ID/EX



Stale data in ID/EX
However, EX/MEM has the correct one