UConn CSE 3666 Exam 2 Review Notes

Notes by Professor Jerry Shi, Compiled by Jacob Gerow 4/14/25

This is intended to be an overview of the important topics on Exam 2 but not an exhaustive list. I.e. if you don't understand something in this presentation you should definitely take the time to go back and understand it, but there is likely information not covered here on the exam

Digital Circuits

Identity laws	A + 0 = A	A · 1 = A
Zero and One laws	A + 1 = 1	A · 0 = 0
Inverse laws	$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$
Commutative laws	A + B = B + A	A · B = B · A
Associative laws	A + (B + C) = (A + B) + C	A · (B · C) = (A · B) · C
Distributive laws	A·(B + C) = (A·B) + (A·C)	A+(B·C) = (A + B)·(A+C)
DeMorgan's laws	$\overline{A + B} = \overline{A} \cdot \overline{B}$	$\overline{A \cdot B} = \overline{A} + \overline{B}$

Product term and sum term

Product term : A single literal or a product (AND) of two or more literals

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

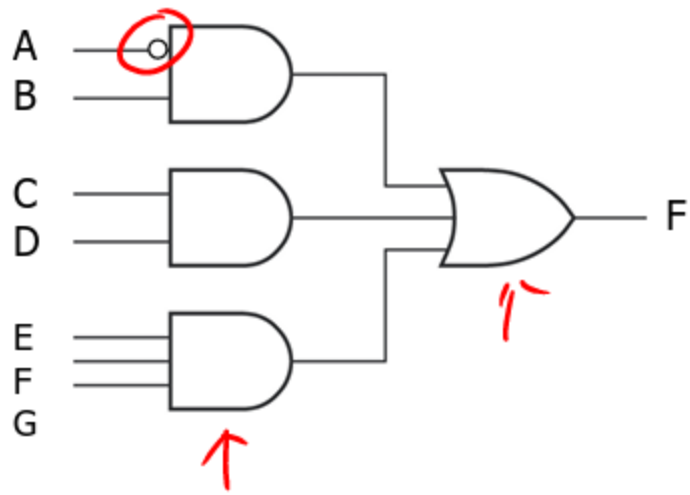
Sum term : A single literal or a logical sum (OR) of two or more literals

$$A, \quad \overline{B}, \quad A + B, \quad X + Y + \overline{Z}$$

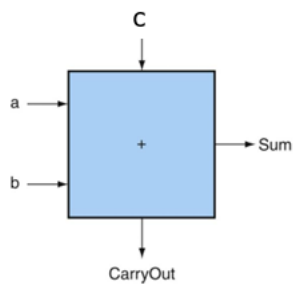
$$F = \overline{X} \cdot \overline{Y} \cdot Z + \overline{X} \cdot Y \cdot \overline{Z} + X \cdot \overline{Y} \cdot \overline{Z} + X \cdot Y \cdot Z$$

	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

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



Full Adder:



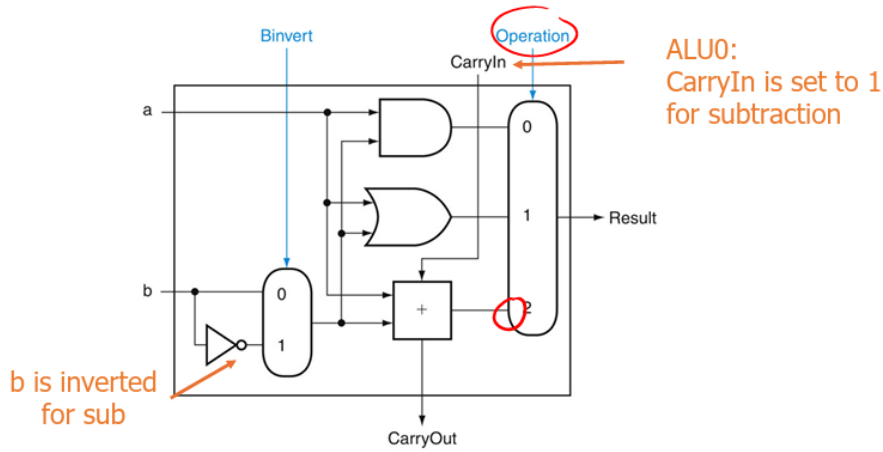
$$\text{Sum} = \overline{a} \cdot \overline{b} \cdot c + \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b} \cdot \overline{c} + a \cdot b \cdot c$$

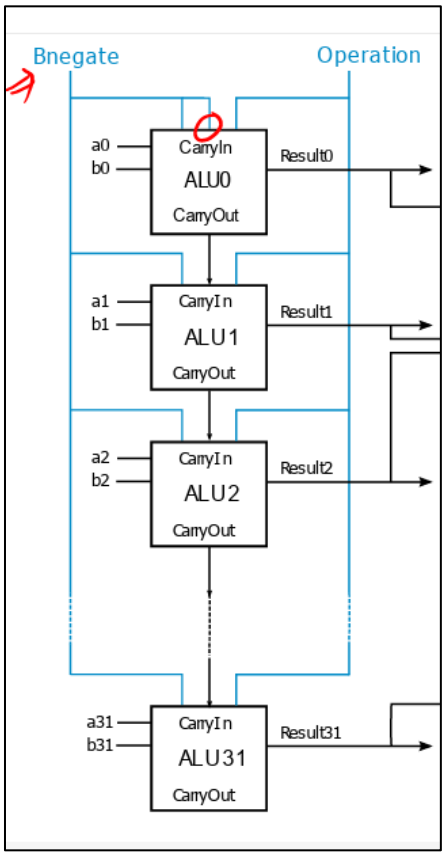
$$\text{CarryOut} = a \cdot b \cdot \overline{\text{CarryIn}} + a \cdot \overline{b} \cdot \text{CarryIn} + \overline{a} \cdot b \cdot \text{CarryIn} + a \cdot b \cdot \text{CarryIn}$$

Subtraction

To negate a two's complement number, flip all the bits and add 1.

$$a - b = a + (-b) = a + \overline{b} + 1$$





Sequential Circuits

- Combinational circuit: the outputs depend on the current input values
- Sequential circuit: the outputs also depend on the history of inputs
  - Two identical sequential circuits may produce different outputs even if their current inputs are the same

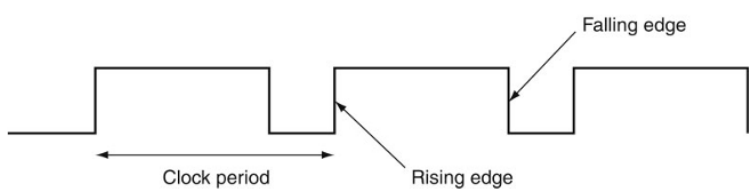
***** **SEQUENTIAL CIRCUITS USE REGISTERS** *****

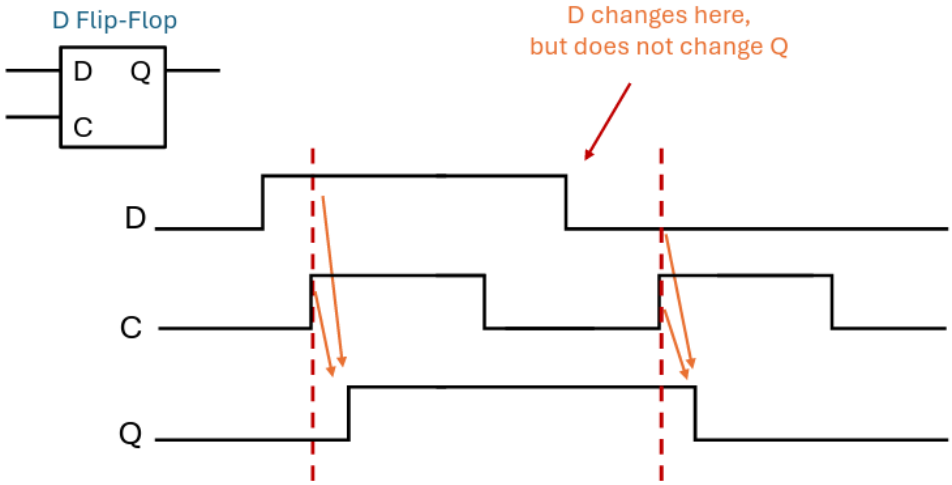
Clock

- A clock signal oscillates between high and low values
- The clock period is the time for one full cycle
  - Also called clock cycle time
  - The clock rate is the reciprocal of the cycle time

If the clock cycle time is 1 ns, the clock rate is 1 GHz.

If the clock rate is 2 GHz, the clock cycle time is 0.5ns.

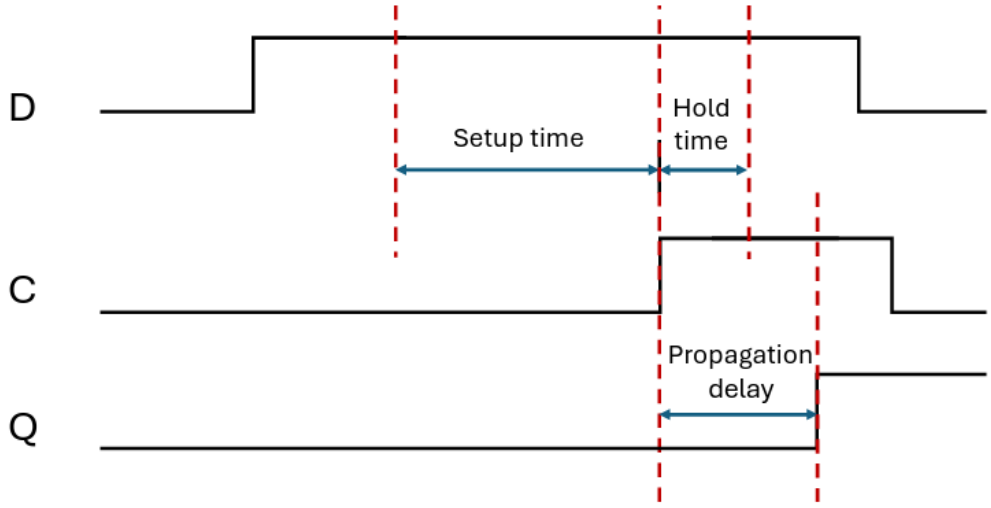




Setup time: D has to be steady for some time before the edge

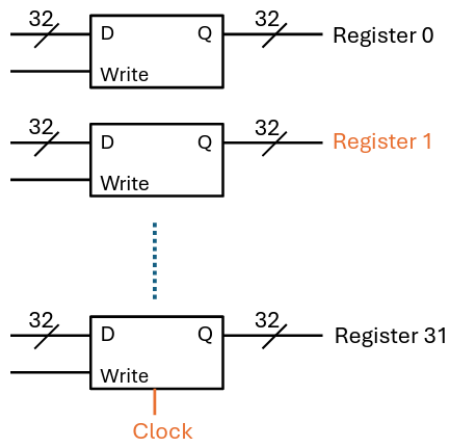
Hold time: D has to be steady for some time after the edge

Propagation delay: Time for input to propagate to output



Register File

- The register file has 32 32-bit Registers
- How do we select the register we need?



$$\text{Clock cycle time} \geq t_{prop} + t_{combinational}$$

{ + t(setup) } (if given)

Multiplier

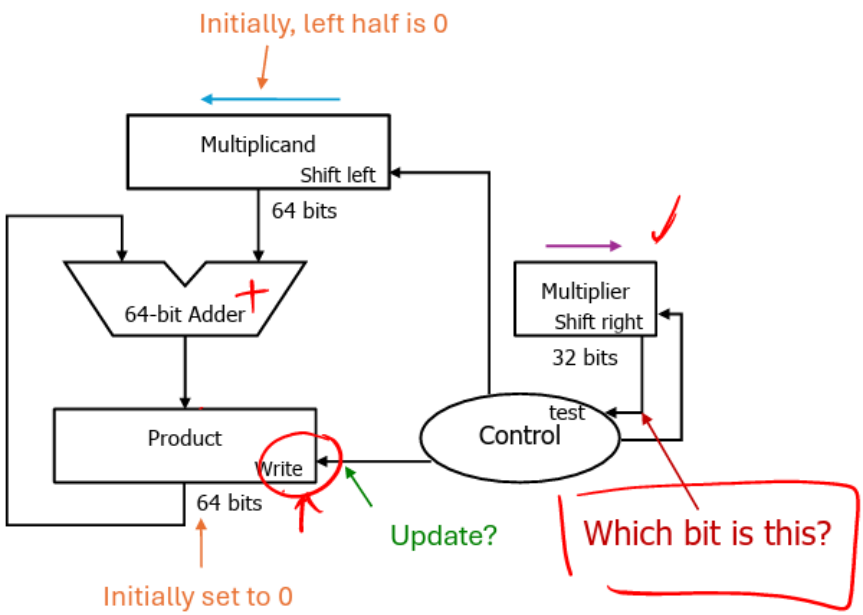
```
        1000   ← multiplicand
×       1001   ← multiplier
        1000
       0000
      0000
     1000
     1001000   ← product
```

- V2

```
1.  product = 0
2.  For i in 0 .. n-1
    2.1a t = multiplicand * multiplier[0]
    2.1b multiplier >>= 1
    2.2 product += t
    2.3 multiplicand <<= 1
```

Always check multiplier[0]

a • b



```
# lower 32 bits of the product
mul    rd, rs1, rs2


# higher 32 bits depend on signs of rs1 and rs2

mulh   rd, rs1, rs2    # both are signed
mulhu  rd, rs1, rs2    # both are unsigned
mulhsu rd, rs1, rs2    # rs1 is signed, rs2 is unsigned

 # signed
 div   rd, rs1, rs2    # rs1 / rs2
 rem   rd, rs1, rs2    # rs1 % rs2

 # unsigned
 divu  rd, rs1, rs2
 remu  rd, rs1, rs2
```

MyHDL

Behavioral Design

```
@block
def Mux2(z, a, b, sel):
    """ 2-input multiplexer

    z = sel ? b : a

    The width of signals do not mattter.
    """

    @always_comb
    def comb():
        if sel:
            z.next = b
        else:
            z.next = a

    return comb
```

Combinational Design

```
@block
def Mux2(z, a, b, sel):
    """ 2-input multiplexer

    z = sel ? b : a

    The width of signals do not mattter.
    """

    @always_comb
    def comb():
        z.next = (not sel and a) or (sel and b)

    return comb
```

```
myMux = Mux2(output_sig, input0_sig, input1_sig, sel_sig)
```

Floating Point

Binary to decimal

Example: `0b101.1101`

bits	1	0	1	1	1	0	1
weights	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

Multiply each bit with weight:

Integer part

Fractional part

0b101.1101

$= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$+ 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$

$= 4 + 0 + 1 + 0.5 + 0.25 + 0 + 0.0625$

$= 5.8125$

Convert the decimal number 0.8 to a binary number

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 ...

*** remember that the last decimal bit you include may need to be rounded up

The normalized binary representation has a single 1 before the point

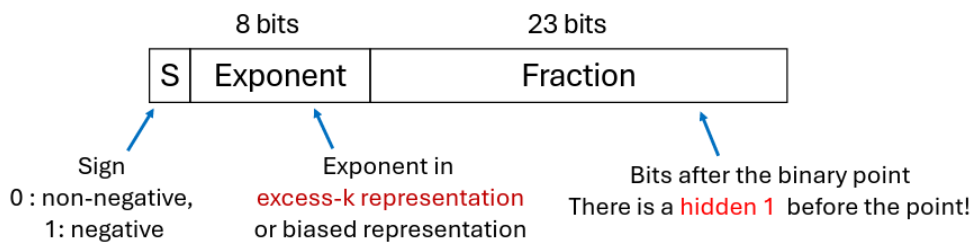
Sign

$$\pm \underbrace{1.x}_{\text{Significand}} \times 2^E$$

Exponent

is written in decimal for convenience

IEEE Floating-Point Format: single-precision



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

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

EncodedExponent = ActualExponent + 127

Denormalized/subnormal Numbers

- Denormalized number: the exponent field is 0b0000_0000
  - The actual exponent is always –126 for single precision numbers
  - The hidden bit is 0

$$v = (-1)^{S} \times (0.\text{Fraction}) \times 2^{-126}$$

Example:

0	0000 0000	100 0000 0000 0000 0000

$0.1 \times 2^{-126} = 1 \times 2^{-127}$

0	0000 0000	000 0000 0000 0000 0001

$0.00000000000000000001 \times 2^{-126} = 2^{-149}$

Infinities and NaN

Exponent field = 0b1111 1111 (255)

- If fraction = 000…0, ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check

0	1111 1111	000 0000 0000 0000 0000

- If fraction ≠ 000…0, Not-a-Number (NaN)
  - Indicates illegal or undefined result. Can be used in subsequent calculations.
    - e.g., 0.0 / 0.0

1	1111 1111	100 0000 0010 0100 0000

IEEE Floating-Point Format: double precision

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

Sign
0 : non-negative,
1: negative

Exponent in
excess-k representation

Bits after the binary point
There is a hidden 1 !

$$\text{value} = (-1)^{S} \times (1.\text{Fraction}) \times 2^{(\text{EncodedExponent}-\text{Bias})}$$

Exponent in single-precision: excess-127: Bias = 127.
Exponent in double-precision: excess-1023: Bias = 1023

** if exponent is n bits, bias is 2^(n-1) – 1

F and D Extensions in RISC-V

- F for float and D for double
  - D is a superset. If D is supported, F is supported
- Separate FP register file (RF) consisting of 32 FP registers
  - In F, each register can hold a float
  - In D, each register can hold a float or a double

```
f0, f1, … f30, f31
```

f0 is a regular register, not always 0!

- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact

- FP load and store instructions
  - w for SP and d for DP

```
flw, fsw, fld, fsd

# same memory addressing modes
# base address is an integer
flw     f8, 0(sp)          # single-precision
fsw     f8, 4(sp)

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

FP Arithmetic

- 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
```

FP Comparison and Branch

- Single- and double-precision comparison

```
f.eq.s, f.lt.s, f.le.s
f.eq.d, f.lt.d, c.le.d
```

- Result, 0 or 1, is saved in an integer destination register
  - Use beq or bne to branch on comparison result

```
# if f3 < f4, goto loop
f.lt.d   t0, f3, f4       # t0 = f3 < f4
bne      t0, x0, loop     # if t0
```

Compare with x0
No need to compare with 1

Conversion between datatypes

- Many conversion instructions. Study the reference card

```
fcvt.s.w, fcvt.d.w, fcvt.d.s, …
```

```
addi       t0, x0, 5    # t0 = 5

fcvt.s.w   ft0, t0      # word to single-precision
# ft0 now is a single-precision 5.0

fcvt.d.w   ft1, t0      # word to double-precision
# ft1 now is a double-precision 5.0
```

Performance

** don't overthink, try to think about logically

CPU Time = Cycles * (sec / cycle)

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

Comparing performance

"X is *n* time faster than Y"

"… than Y"
Y's time is the top

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

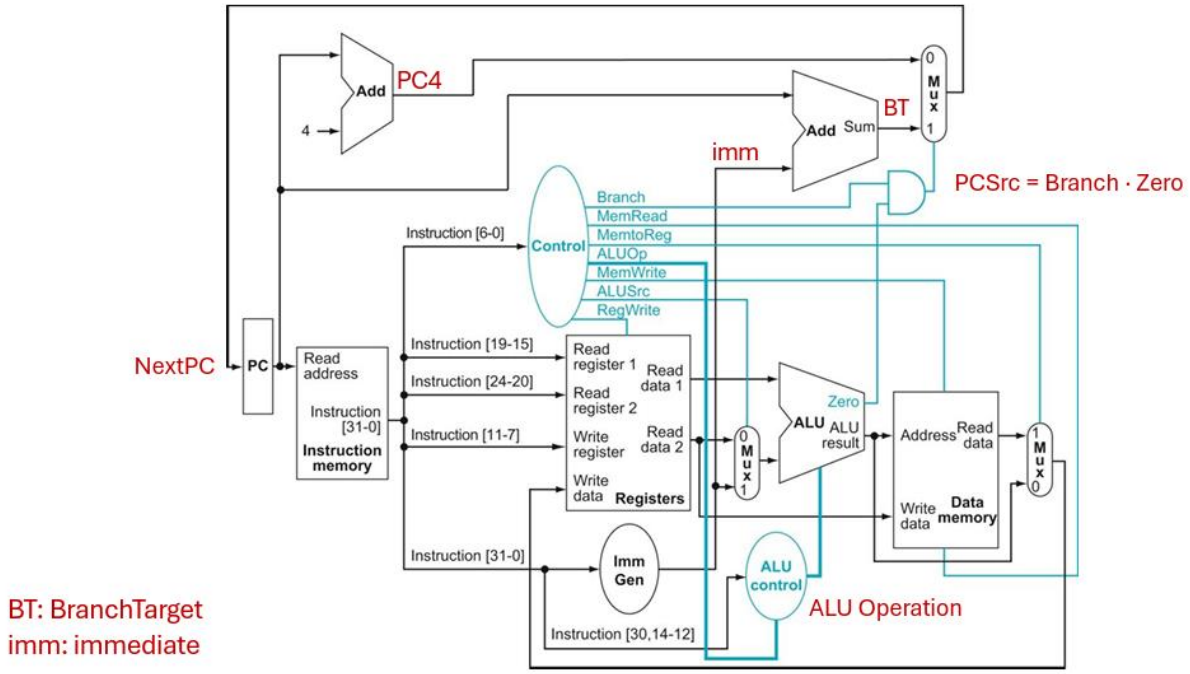
$$\text{Speedup} = \frac{\text{CPU Time}_{\text{before\_enhancement}}}{\text{CPU Time}_{\text{after\_enhancement}}}$$

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

** best speedup you can achieve by optimizing a section of the code is if that section has an **infinite** speedup

$$\text{T}_{\text{improved}} = \frac{\text{T}_{\text{affected}}}{\text{improvement factor}} + \text{T}_{\text{unaffected}}$$

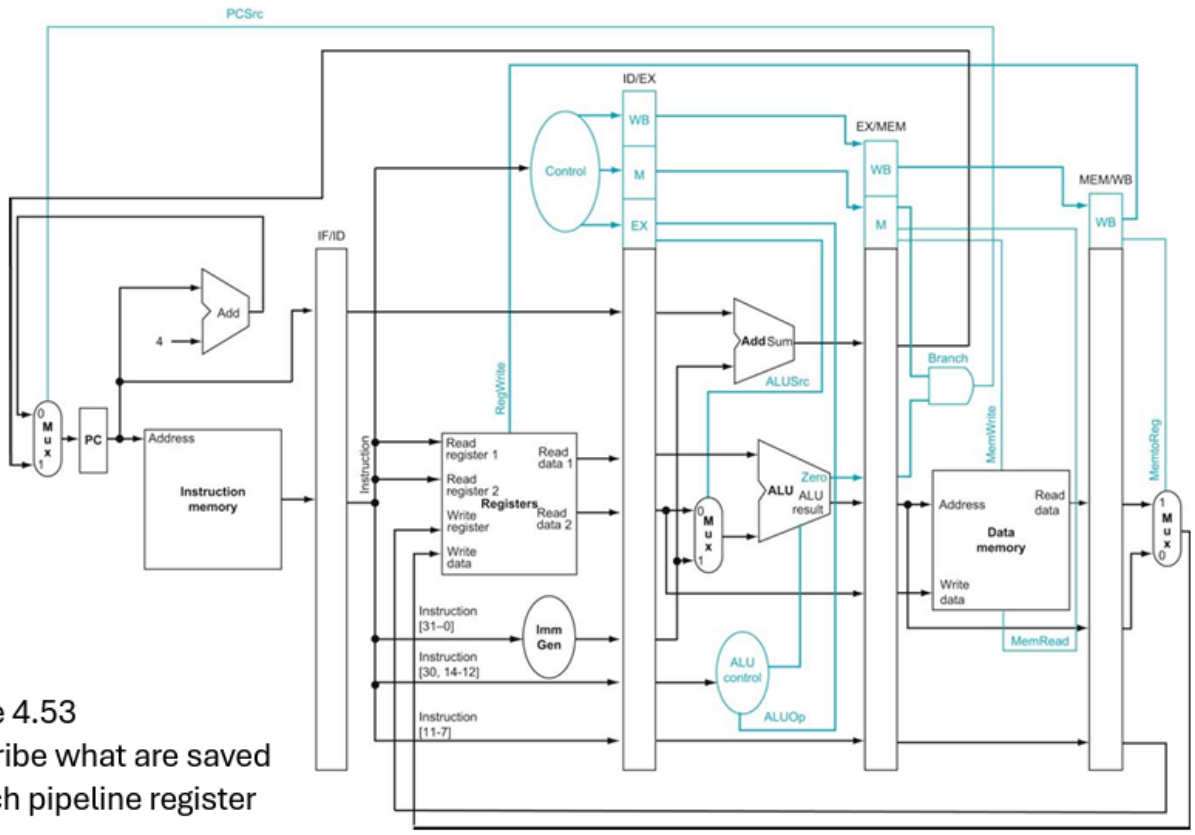
Single Cycle Implementation



- You should understand this figure **fully**

Pipelined Implementation

- Speedup = number of stages in pipeline



re 4.53
cribe what are saved
ach pipeline register

- Understand this figure fully

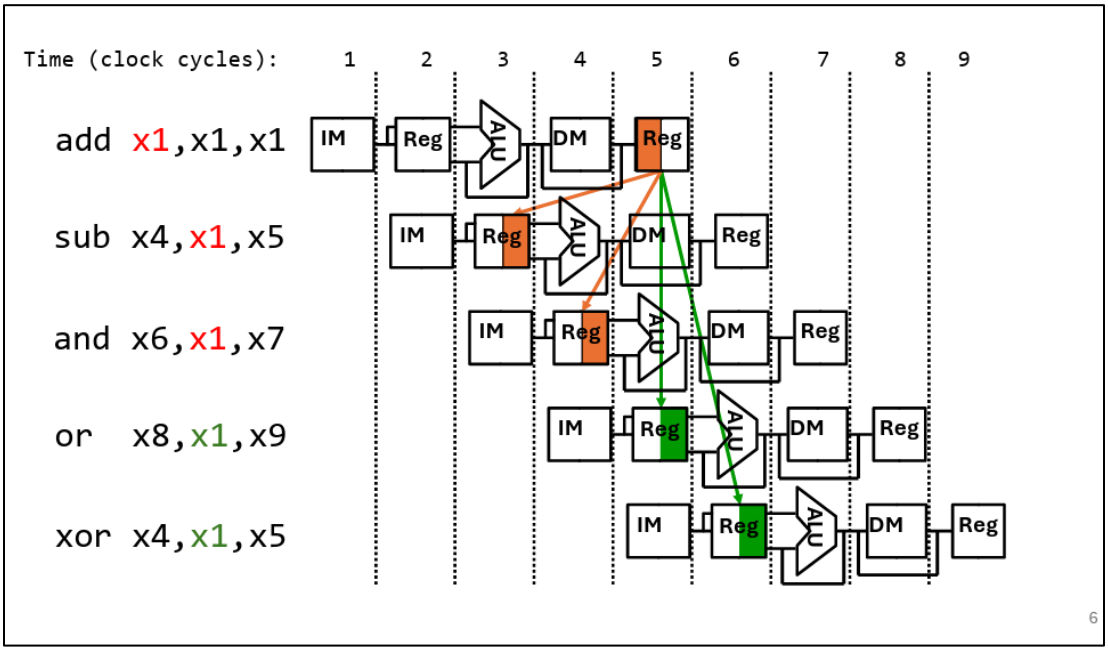
Hazards

Structural hazards:

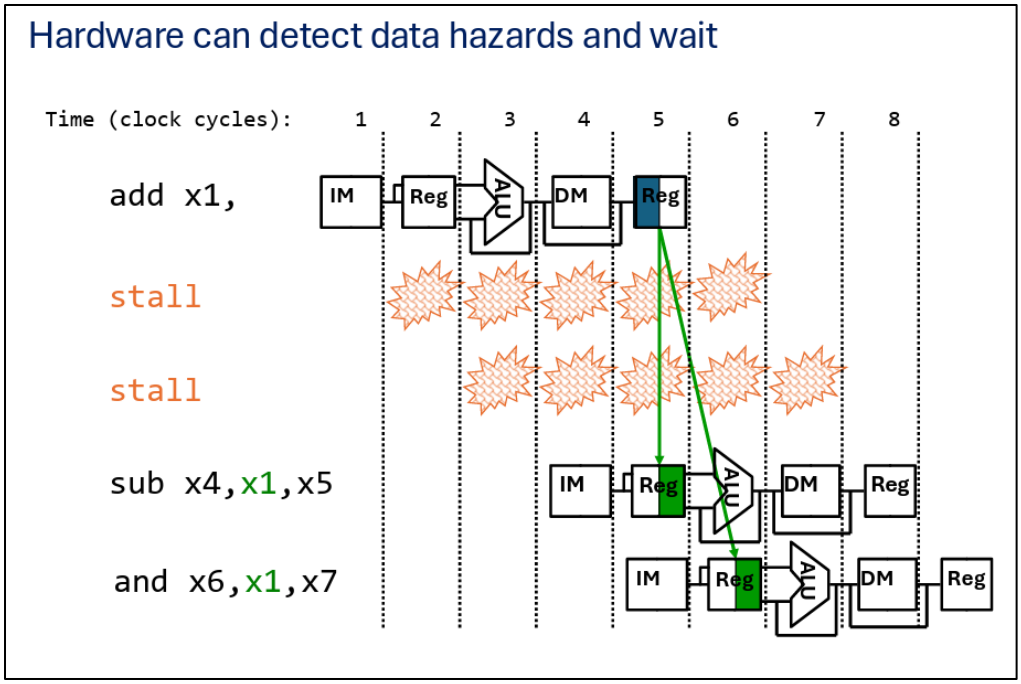
- 2 or more parts of the processor are attempting to access the same resource at the same time
- E.X. Register File, the write back stage potentially writes back in the same cycle the decode stage decodes.
    - We deal with this by letting the WB stage write back in the first half of the clock cycle and letting the register file decode the instruction in the second half of the clock cycle
- There are **no structural hazards** (that introduce delay) present in our processor

Data hazards:

- A decoding instruction is trying to decode a register which is already in the pipeline (and not yet written back)



Data hazard solutions:

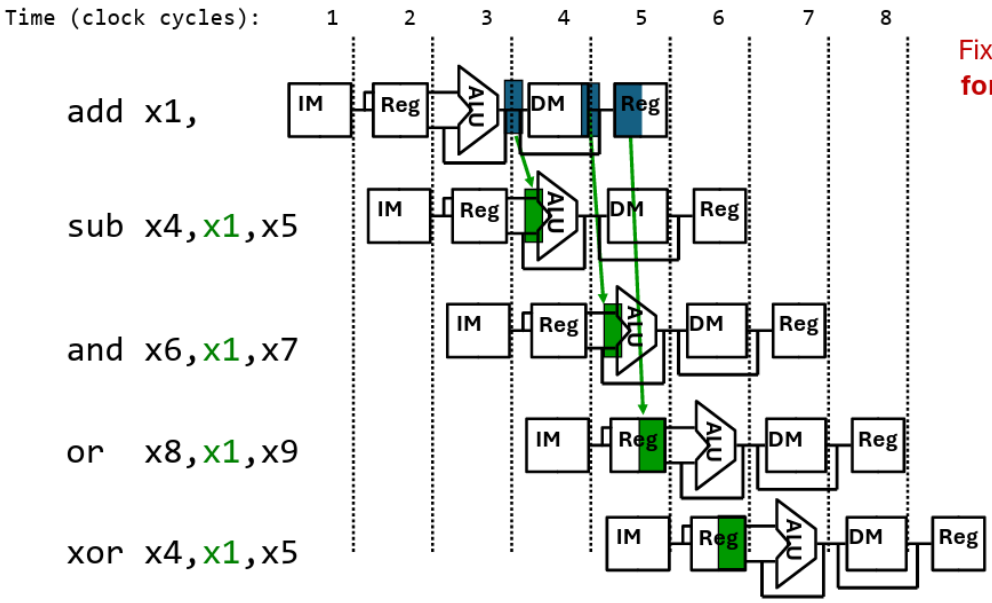


		1	2	3	4	5	6	7	8
add	x1,x2,x3	IF	ID	EX	MEM	WB			
sub	x4,x1,x5		IF	ID	-	-	EX	MEM	WB
and	x6,x1,x7			IF	-	-	ID	EX	MEM

“-“ means repeat the same stage

IF ID - - EX MEM WB ==== IF ID ID ID EX MEM WB (First two IDs yield incorrect result)

Another (better) Way to "Fix" a Data Hazard

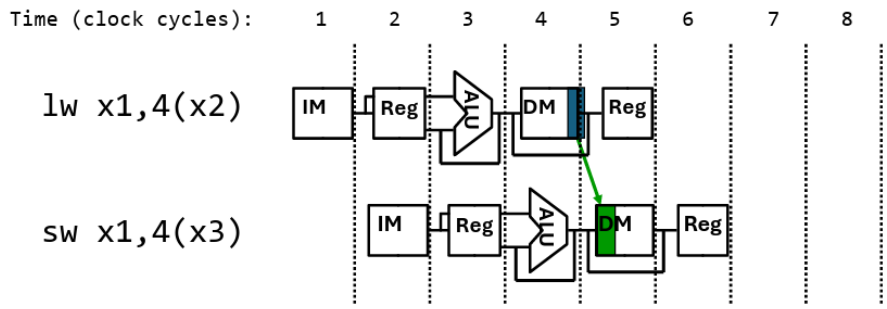


Fix data hazards by **forwarding** results from where they are **available** to where they are **needed**

	1	2	3	4	5	6	7	8
add x1,x2,x3	IF	ID	EX	MEM	WB			
sub x4,x1,x5		IF	ID	EX	MEM	WB		
or x6,x7,x1			IF	ID	EX	MEM	WB	

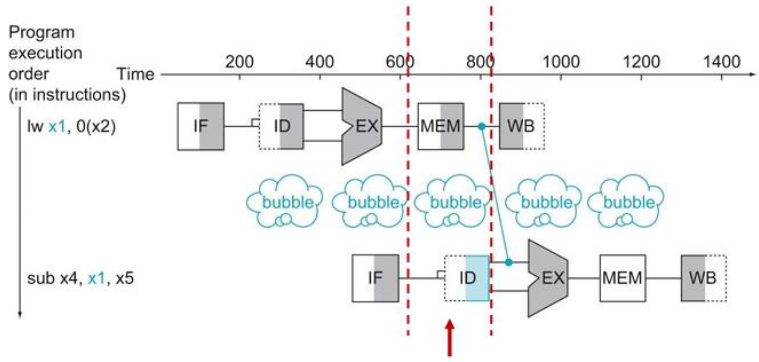
MEM/WB to MEM forwarding

- Loads that immediately followed by stores (memory-to-memory copies) can avoid a stall via forwarding
  - From the MEM/WB register to the data memory input
  - MUX and forwarding logic are added in MEM stage

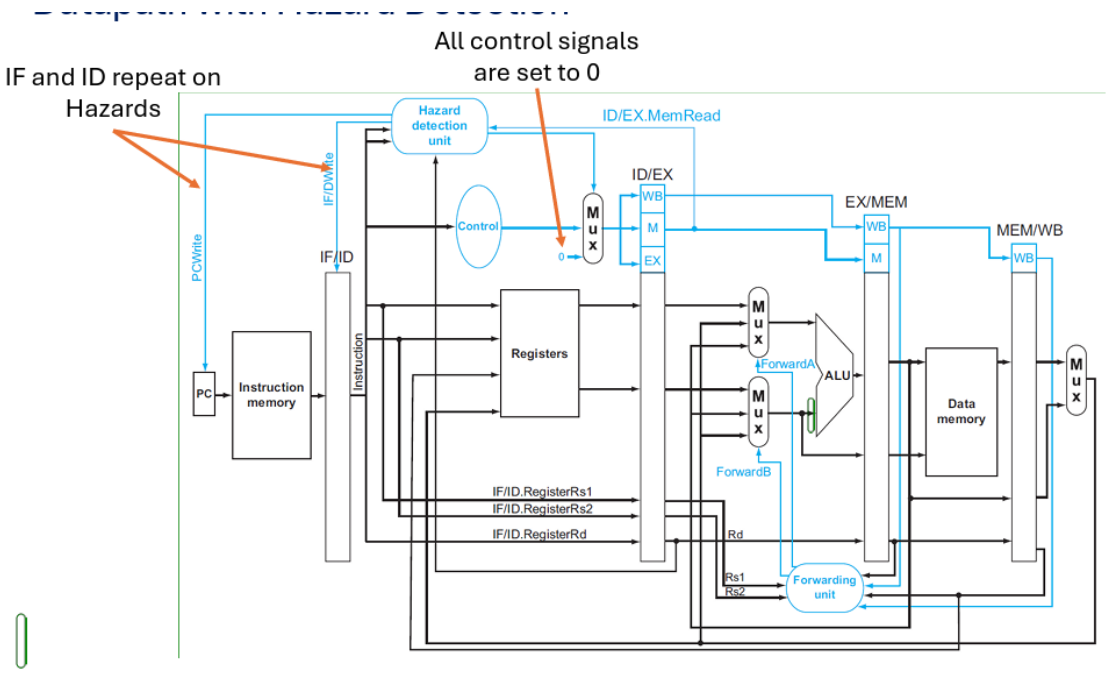


Load-Use Data Hazard

- Cannot always avoid stalls by forwarding
  - The value needed is not computed/loaded yet



SUB cannot get in Ex and use x1 in this cycle
Wait in ID (not in EX!)



Control Hazards:

- Caused when you are executing a branch instruction and the processor is not sure which instruction will come next
- Not covered in this exam