

Review Problems

Quiz 1

October 2nd 2020

ECE 6913

Agenda

- Processor Performance comparison

HW 4 Part A

- How is a RISC V Instruction executed in hardware?
- What are the hardware component latencies and how do they determine execution time of any given instruction?
- How can we make these faster? [look at single cycle simplified implementation]
 - Make cycle time unique to each instruction – rather than equal to the slowest instruction?
 - Make the slowest instruction faster?
 - Effects of Pipelining
- What could make the processor slower?
 - Add functionality

HW 4 Part B

- How do we determine program execution time penalty from Data Hazards – with *and* without forwarding hardware overheads
- How do structural hazards from use of a single Memory Array increase CPI & how is this Structural Hazard resolved by instruction scheduling?
- Lowering CPI penalty due to Load/Store Data Use Hazards by overlapping EX and MEM stages of the RISC pipeline
- Load-use-data hazard resolution for given instruction sequences including branches

Example Problem 1

- Consider three different processors P1, P2, and P3 executing the **same instruction set**. P1 has a 3 GHz clock rate and a CPI of 1.5. P2 has a 2.5 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.2.
 - a. Which processor has the highest performance expressed in instructions per second?
 - b. If the processors each execute a program in 10 seconds, find the *number of cycles* and the *number of instructions*.
 - c. We are trying to *reduce the execution time by 30%*, but this leads to an *increase of 20% in the CPI*. What clock rate should we have to get this time reduction?

1a

IPS = Cycles per Second/Cycles per Instruction

A measure of throughput – or rate of doing work

- Performance of P1: $3\text{GHz}/1.5 = 2 \times 10^9 \text{ Inst/sec}$
- Performance of P2: $2.5\text{GHz}/1.0 = 2.5 \times 10^9 \text{ Inst/sec}$
- Performance of P3: $4\text{GHz}/2.2 = 1.8 \times 10^9 \text{ Inst/sec}$
- **CPI** can be as relevant to processor performance as clock frequency
- Faster clocks may not always be a good thing – higher power dissipation, worse reliability, worse coupling noise... and not the best rate of processing instructions!

1b

- # of cycles = Cycles per second x time (in seconds)
- Cycles of P1: 3 GHz x 10 s = 30 B cycles
- Cycles of P2: 2.5GHz x 10 s = 25 B cycles
- Cycles of P3: 4 GHz x 10s = 40 B cycles

Assuming a metric of wall clock time to execute a given benchmark program,

P3 consumed more clock cycles – more power to do the same work

P2 consumed the least number of clock cycles to do the same work

Lower CPI translates into higher productivity and higher energy efficiency as a result

1b contd..

- # of Instructions = Cycles / CPI
- # of Instructions of P1: 30 B cycles/1.5 Cycles per Instruction = 20B
- # of Instructions of P2: 25 B cycles/1 Cycles per Instruction = 25B
- # of Instructions of P3: 40 B cycles/2.2 Cycles per Instruction = 18.18B

1c

- *Lower execution time trades off with higher CPI & higher F_{CLK}*
- Assuming 30% reduction in execution time requires 20% higher CPI

$$\# \text{ Instructions} \times \text{CPI}_{\text{new}} / \text{ET}_{\text{new}} = F_{\text{clk_new}}$$

$$F_{\text{clk_new P1}} = 20 \text{ B} \times 1.8 / 7\text{s} = 5.14 \text{ GHz}$$

$$F_{\text{clk_new P2}} = 25 \text{ B} \times 1.2 / 7\text{s} = 4.28 \text{ GHz}$$

$$F_{\text{clk_new P1}} = 18.18 \text{ B} \times 2.6 / 7\text{s} = 6.75 \text{ GHz}$$

*High CPI processors require **even higher Clock rates** to get the same % improvement in execution time*

Example Problem 2

Compilers can have a profound impact on the performance of an application. Assume that for a program, compiler A results in a dynamic instruction count of $1.0\text{E}9$ and has an execution time of 1.1 s, **while compiler B results in a dynamic instruction count of $1.2\text{E}9$ and an execution time of 1.5 s.**

- a. Find the average CPI for each program given that the processor has a clock cycle time of 1 ns.
- b. Assume the compiled programs run on two different processors. If the execution times on the two processors are the same, how much faster is the clock of the processor running compiler A's code versus the clock of the processor running compiler B's code?
- c. A new compiler is developed that uses only $6.0\text{E}8$ instructions and has an average CPI of 1.1. What is the speedup of using this new compiler versus using compiler A or B on the original processor?

2a

- $\text{CPI} = \text{ETime} \times \text{Fclk} / \text{Instr Count}$
- Compiler A $\text{CPI} = 1.1\text{s} \times 1\text{GHz} / 1\text{ B} = 1.1$
- Compiler B $\text{CPI} = 1.5\text{s} \times 1\text{GHz} / 1.2\text{ B} = 1.25$

On a given machine with a **given clock frequency**, different compilers that generate machine instructions using the **same instruction set architecture** *can differentiate* in achieving **lower CPI** and **higher performance** as a result

2b

- Assume the **processors are different** and **Execution times are now the same**
- How much faster is clock running B's code Vs clock running A's code?

$$\begin{aligned} F_{\text{clk_B}} / F_{\text{clk_A}} &= [IC_B \times CPI_B] / [IC_A \times CPI_A] \\ &= [1.2 \text{ B} \times 1.25] / [1 \text{ B} \times 1.1] \\ &= 1.36 \end{aligned}$$

2c

- New compiler uses only 0.6 B instructions, CPI = 1.1
- Speedup Versus A or B on the original processor:
- Instr Count x CPI = #cycles in original processor
- $T_A / T_{new} = \text{Speedup Vs A: } 0.66 \text{ B cycles Vs } 1 \text{ B} \times 1.1 \text{ or } 1.1 \text{ B cycles}$
 $= 1.1 / 0.6 = 1.67$
- $T_B / T_{new} = 0.66 \text{ B cycles Vs } 1.2 \text{ B} \times 1.25$
 $= 1.5 / 0.66 = 2.27$

Example Problem 3

Consider two **different implementations** of the **same instruction set architecture**. The instructions can be divided into four classes according to their CPI (classes A, B, C, and D). **P1 with a clock rate of 2.5 GHz and CPIs of 1, 2, 3, and 3**, and **P2 with a clock rate of 3 GHz and CPIs of 2, 2, 2, and 2**.

- a. What is the global CPI for each implementation?
- b. Given a program with a dynamic instruction count of 1.0E6 instructions divided into classes as follows: 10% class A, 20% class B, 50% class C, and 20% class D, **which is faster: P1 or P2?**
- c. Find the clock cycles required in both cases.

3a,b

INSTR CLASS →	A: 10%	B: 20%	C: 50%	D: 20%
P1 CPI	1	2	3	3
P2 CPI	2	2	2	2

- $\text{CPI of P1} = 0.1 \times 1 + 0.2 \times 2 + 0.5 \times 3 + 0.2 \times 3 = 2.6$
- $\text{CPI of P2} = 0.1 \times 2 + 0.2 \times 2 + 0.5 \times 2 + 0.2 \times 2 = 2.0$

$$\text{ET}_{\text{P1}} = [\text{CPI} / \text{FCLK}] \times \text{IC} = [2.6 / 2.5\text{G}] \times 1\text{M} = 1.04 \text{ ms}$$

$$\text{ET}_{\text{P2}} = [\text{CPI} / \text{FCLK}] \times \text{IC} = [2.0 / 3\text{G}] \times 1\text{M} = 0.66 \text{ ms}$$

P2 is faster!

3c

-
- Clock cycles required = CPI x IC
 - P1: $2.6 \times 1M = 2.6M$ cycles
 - P2: $2.0 \times 1M = 2.0M$ cycles

Example Problem 4

The **Pentium 4** Prescott processor, released in 2004, had a clock rate of **3.6 GHz** and voltage of 1.25 V. Assume that, on average, it consumed **10 W of static power and 90 W of dynamic power**. The **Core i5** Ivy Bridge, released in 2012, has a clock rate of **3.4 GHz** and voltage of 0.9 V. Assume that, on average, it consumed **30 W of static power and 40 W of dynamic power**.

- a. For each processor find the average capacitive loads.
- b. Find the percentage of the total dissipated power comprised by static power and the ratio of static power to dynamic power for each technology.
- c. If the total dissipated power is to be reduced by 10%, how much should the voltage be reduced to maintain the same leakage current? Note: power is defined as the product of voltage and current.

4a, b

- $DP = F \times \frac{1}{2} CV^2$
- $C = 2DP/[V^2F]$

$$P4: 2 \times 90W / [1.5625 \times 3.6G] = 3.2 \times 10^{-8} F$$

$$i5: 2 \times 40W / [0.81 \times 3.4G] = 2.9 \times 10^{-8} F$$

$$P4: 10W / 100W = 10\%$$

$$i5: 30W / 70W = 42.9\%$$

Leakage current increases as # of transistors on chip increases exponentially

Example Problem 5

Instruction	Frequency	Cycles per Instruction
ALU operations	30%	1
Load	20%	2
Store	10%	2
Branches	20%	3
Floating point operations	20%	5

- What is the overall CPI of this machine?
- If the CPU runs at 750MHz, what is the MIPS rating of this machine? For this question, count floating point operations in the MIPS rating.
- Consider improving this computer's performance by enhancing the speed of the floating point instructions. What is the best possible overall speedup that we could obtain?

5a, b, c

- **CPI overall**

$$= 0.3 \times 1 + 0.2 \times 2 + 0.1 \times 2 + 0.2 \times 3 + 0.2 \times 5$$

$$= 0.3 + 0.4 + 0.2 + 0.6 + 1.0 = 2.5$$

- **MIPS (millions of instructions per second)**

$$= \text{Clock rate} / \text{CPI} = 750 \times 10^6 / 2.5 = 3 \times 10^8$$

$$= 300 \text{ MIPS}$$

- **Biggest increase in CPI contributed by floating point instructions (need more cycles per instruction)**

Improvements in CPI of floating-point instruction CPI = infinite, i.e., CPI of FP instructions $\rightarrow 0$

How much does the CPI of machine improve?

$$\text{Speedup} = \text{CPI old} / \text{CPI new}$$

$$\text{CPI new} =$$

$$0.3 \times 1 + 0.2 \times 2 + 0.1 \times 2 + 0.2 \times 3 + 0.2 \times 0 = 1.5$$

$$\text{Speedup} = 2.5 / 1.5 = 1.667$$

Example Problem 6

Two enhancements, E1 and E2, with the following speedups are proposed for a new architecture:

Speedup1 = 10

Speedup2 = 5

Only one of the enhancements is usable at any point in time (maybe because they use some of the same hardware).

- a. If E1 can be used 20% of the time and E2 can be used 10% of the time, what would be the overall speedup?
- b. If the percentage of time that E1 can be used decreased to 15%, what percentage of the time would the use of E2 have to be to get the same overall speedup as in part (a)?
- c. Suppose we are free to choose between E1 or E2, whenever we want (the percentages of time for using E1 or E2 can be varied as desired, but in total cannot be more than 100% of the time). What would be the maximum achievable overall speedup?

6 a, b, c

a. Speedup = $T_e \text{ old} / T_e \text{ new}$

$$T_e / [20\% (T_e/10) + 10\% (T_e/5) + 70\% x (T_e/T_e)]$$
$$= 1 / [0.02 + 0.02 + 0.7] = 1/0.74 = 1.35$$

b. $1/[0.15/10 + x/5 + (0.85 -x)] = 1 / [0.74]$

$$[0.15/10 + x/5 + (0.85 -x)] = 0.74$$

$$x = 0.125 / 0.8 = 0.15625$$

Enhancement 2 would need to increase its percentage time from 10% to 15.625% to make up for a decrease in time of Enhancement 1 from 20% to 15%

c. speedup = $T_e / [100\% x (T_e/10)] = 10$

Example Problem 7

- Suppose a program (or a program task) takes 1 billion instructions to execute on a processor running at 2 GHz. Suppose also that 50% of the instructions execute in 3 clock cycles, 30% execute in 4 clock cycles, and 20% execute in 5 clock cycles. What is the execution time for the program or task?

Instruction	Frequency	Cycles per Instruction
A	50%	3
B	30%	4
C	20%	5

Problem 7

Average Cycles Per Instruction (CPI) of the Program

$$= 0.5 \times 3 + 0.3 \times 4 + 0.2 \times 5 = \mathbf{3.7}$$

1 billion instructions \times CPI = number of cycles required by Program = 3.7×10^9

at 2 GHz, *one clock cycle* consumes = $1 / [2 \times 10^9]$ seconds or 0.5×10^{-9} seconds or 0.5 nanoseconds

So, 3.7×10^9 cycles consumes $3.7 \times 10^9 \times 0.5 \times 10^{-9}$ seconds
 $= 3.7 \times 0.5 = 1.85$ seconds

Example Problem 8

- Suppose the processor in the previous example is redesigned so that all instructions that initially executed in 5 cycles now execute in 4 cycles. Due to changes in the circuitry, the clock rate has to be decreased from 2.0 GHz to 1.9 GHz. What is the overall percentage improvement?

Instruction	Frequency	Cycles per Instruction
A	50%	3
B	30%	4
C	20%	4

P8

Now, the Average Cycles Per Instruction (CPI) of the Program

$$= 0.5 \times 3 + 0.3 \times 4 + 0.2 \times 4 = 3.5$$

So,

1 billion instructions \times CPI = number of cycles required by Program = 3.5×10^9

at 1.9 GHz,

one clock cycle consumes = $1 / [1.9 \times 10^9]$ seconds or 0.526315×10^{-9} seconds

So,

3.5×10^9 cycles consumes $3.5 \times 10^9 \times 0.526315 \times 10^{-9}$ seconds = 1.8421025 sec

so improvement is $1.85/1.8421025 = 1.0042872$ or $\sim 0.43\%$ improvement

Instruction execution in single cycle datapath

In this exercise, we examine in detail how an instruction is executed in a single-cycle datapath.

Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word: **0x00c6ba23**.

What are the values of the ALU control unit's inputs for this instruction?

What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

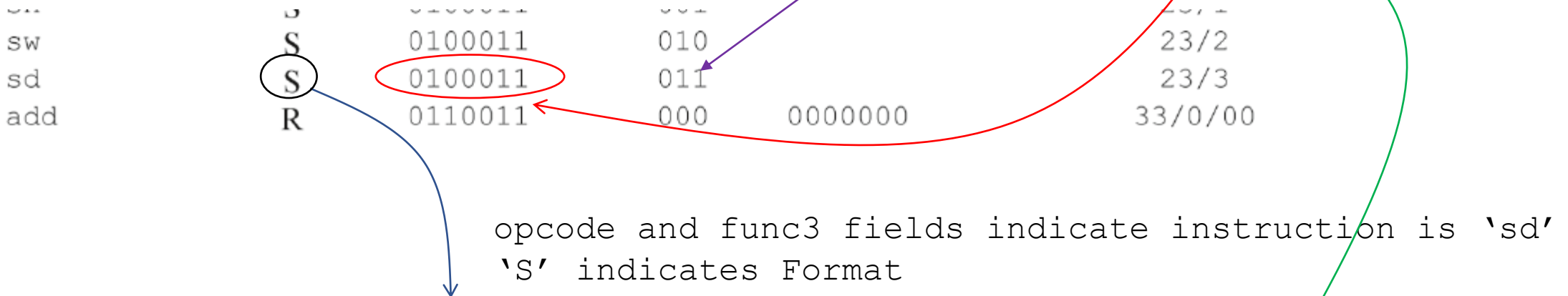
For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at Reg [xn].

What are the input values for the ALU and the two add units? of all inputs for the register's unit?

Instruction Fetch & Decode

- Processor fetches the following instruction word: **0x00c6ba23**.

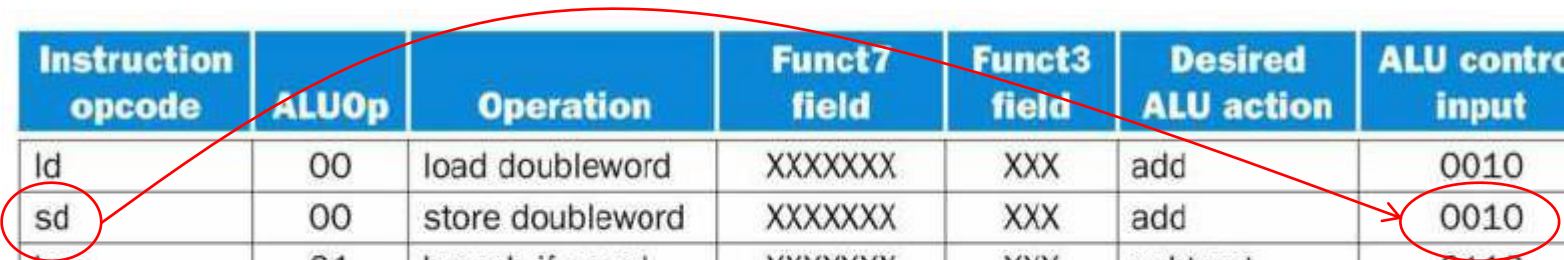
$00c6ba23_{16} = 0000\ 0000\ 1100\ 0110\ 1011\ 1010\ 0010\ 0011$



immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- The source register fields **rs1** & **rs2** identify register #s **x13** and **x12**
- S format splits the 12 bit **immediate** field in [31:25] & [11:7] of instruction
- So, instruction is `sd x12, 20(x13)`

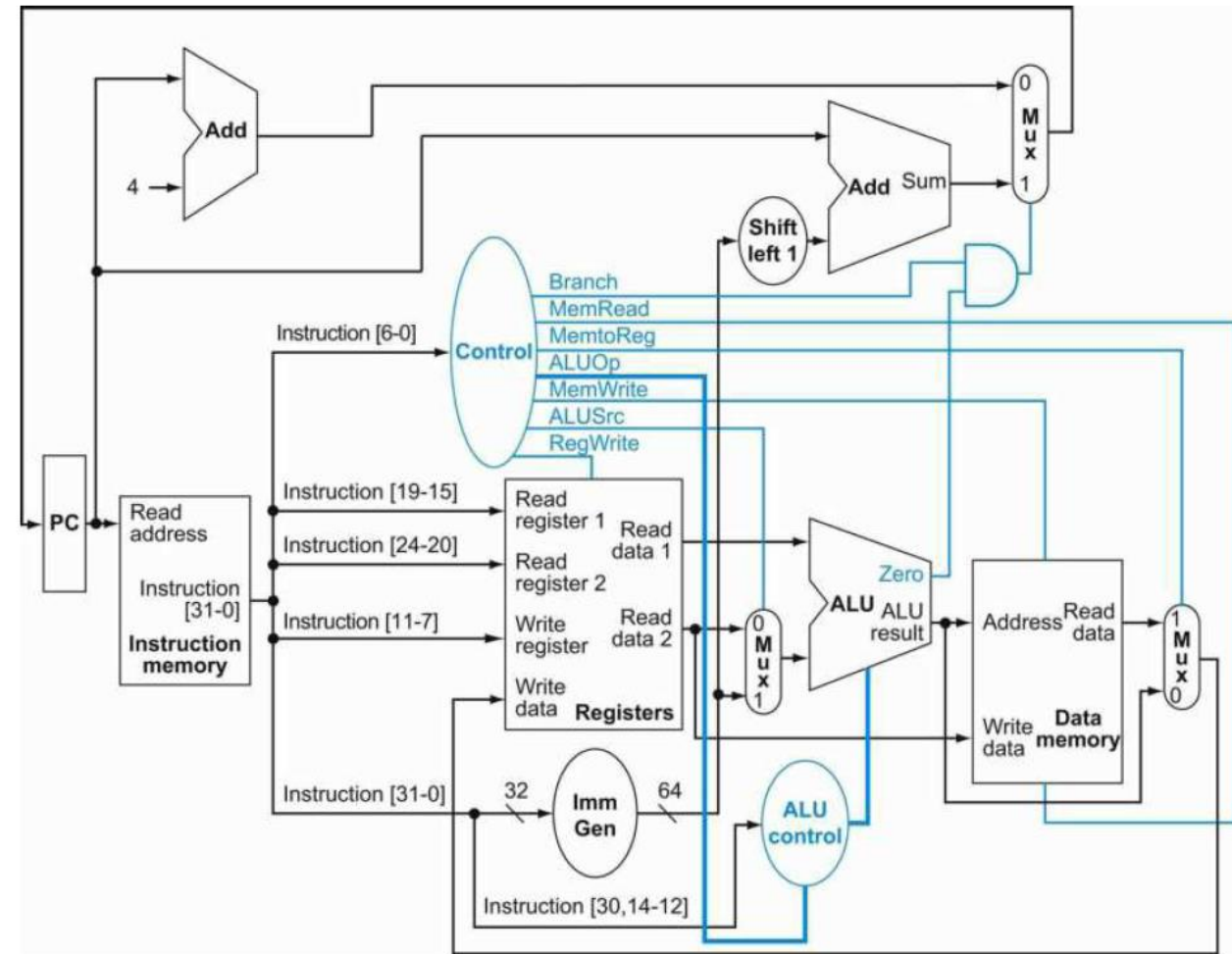
Instruction Execute



Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Since the '**add**' operation is executed in the ALU for a **sd** instruction (add offset in **immediate** 5 bit field (20) to base address (in **rs1**), the ALU Control lines take the value **0010**

- Since the **sd** instruction does not take any branch, the *next PC address* after this instruction is executed is **PC+4**
- Values of mux inputs and outputs during the execution : **ALUsrc**, **PCsrc** and **MemtoReg**
- **ALUsrc** is '1' and takes the input to this mux from the sign extend unit (0x0..0**14**)
- so that ALU can add **imm field** to contents of the base register **rs1** in the ALU to determine the address in memory to store contents of register **rs2**.



- **PCsrc** is '0' since this mux is asserted only for branch instructions
- **MemtoReg** is value is irrelevant (don't care)

Mux	Control input	input 1	input 2	output
ALUsrc	1	rs2	0x00..014	0x0..014
PCsrc	0	PC+4	offs sll 1	PC + 4
MemtoReg	δ	rs1 +0x..14	unknown	unknown

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

- **Branch adder** inputs: **PC** and 0x0..014 shifted by 1 (**0x0..28**)
- **PC adder** inputs: PC and 4
- Read register 1 (**rs1**) input port has the 5-bit instruction field [19-15] identifying the register **x13**, Read register 2 (**rs2**) input port has the 5-bit instruction field [24 – 20] identifying the register **x11**.
- Write register input port has undefined/irrelevant bits since Reg Write control signal is disabled for the **sd** instruction.

Datapath latencies for Instructions

- Problems in this exercise assume that the logic blocks used to implement a processor's datapath have the following latencies:

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

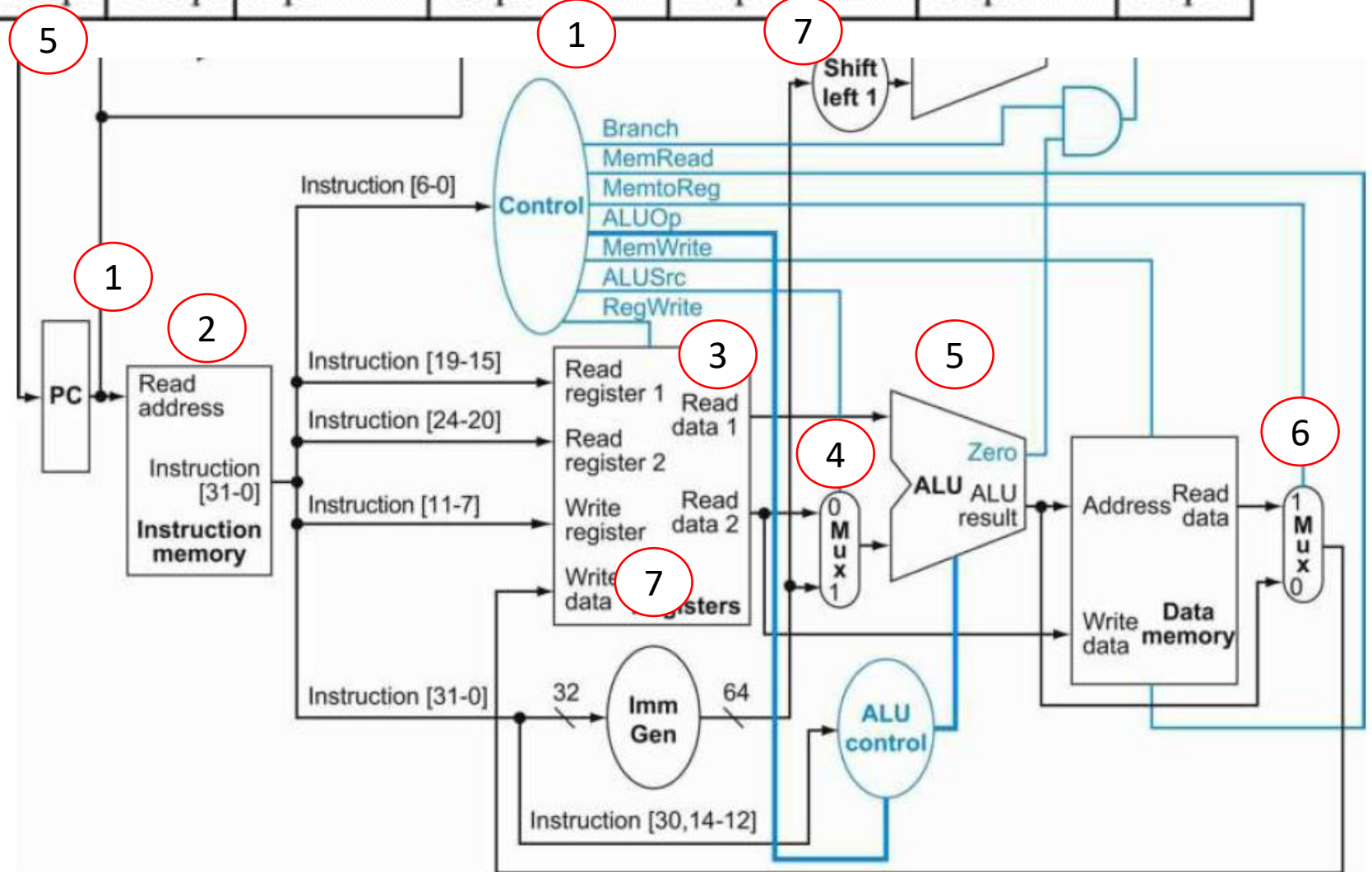
- “Register read” is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only.
“Register setup” is the amount of time a register's data input must be stable before the rising edge of the clock. This value applies to both the PC and Register File.

What are the latencies of R-type, lw, sw, beq instructions (i.e., how long must the clock period be to ensure that this instruction works correctly)?

Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

R-type Instruction
Path Delay



Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

R-Type Instruction:

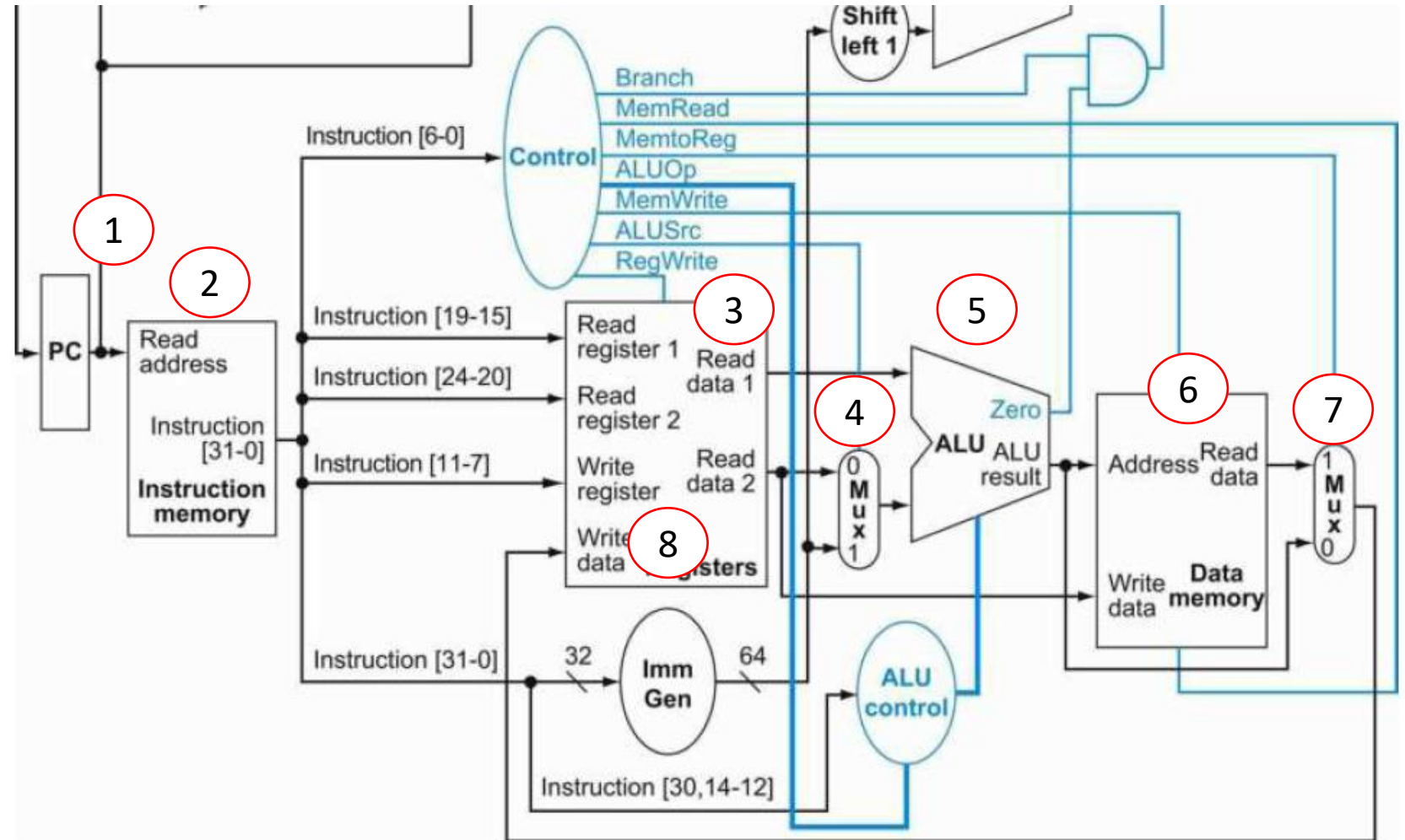
1. PC Register Read delay following rising edge of clock: 30 ps
 2. Instruction Memory: 250ps
 3. Read Register File: 150 ps
 4. Mux to pick data from either R2 or sign extended bit data from immediate field of instruction = 25 ps
- [note - there is no number given for sign extension delay, so we can assume it is less than the register file delay of 150 ps]
5. ALU delay: 200 ps
 6. Mux for WB to Register File of result from ALU: 25 ps
 7. Write back setup time for Register File registers: 20 ps

total = 700ps

Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

ld Instruction Path Delay



Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

1d Instruction:

1. PC Register Read delay following rising edge of clock: 30 ps
2. Instruction Memory: 250ps
3. Read Register File: 150 ps
4. Mux to pick data from either R2 or sign extended bit data from immediate field of instruction = 25 ps

[note - number given for sign extension delay of 50 ps is less than the 150 ps for register file read access, so we can assume that mux control signal is designed to fire only when both inputs are ready]

5. ALU delay: 200 ps

6. Read data from Data Mem, whose address provided by ALU: 250 ps

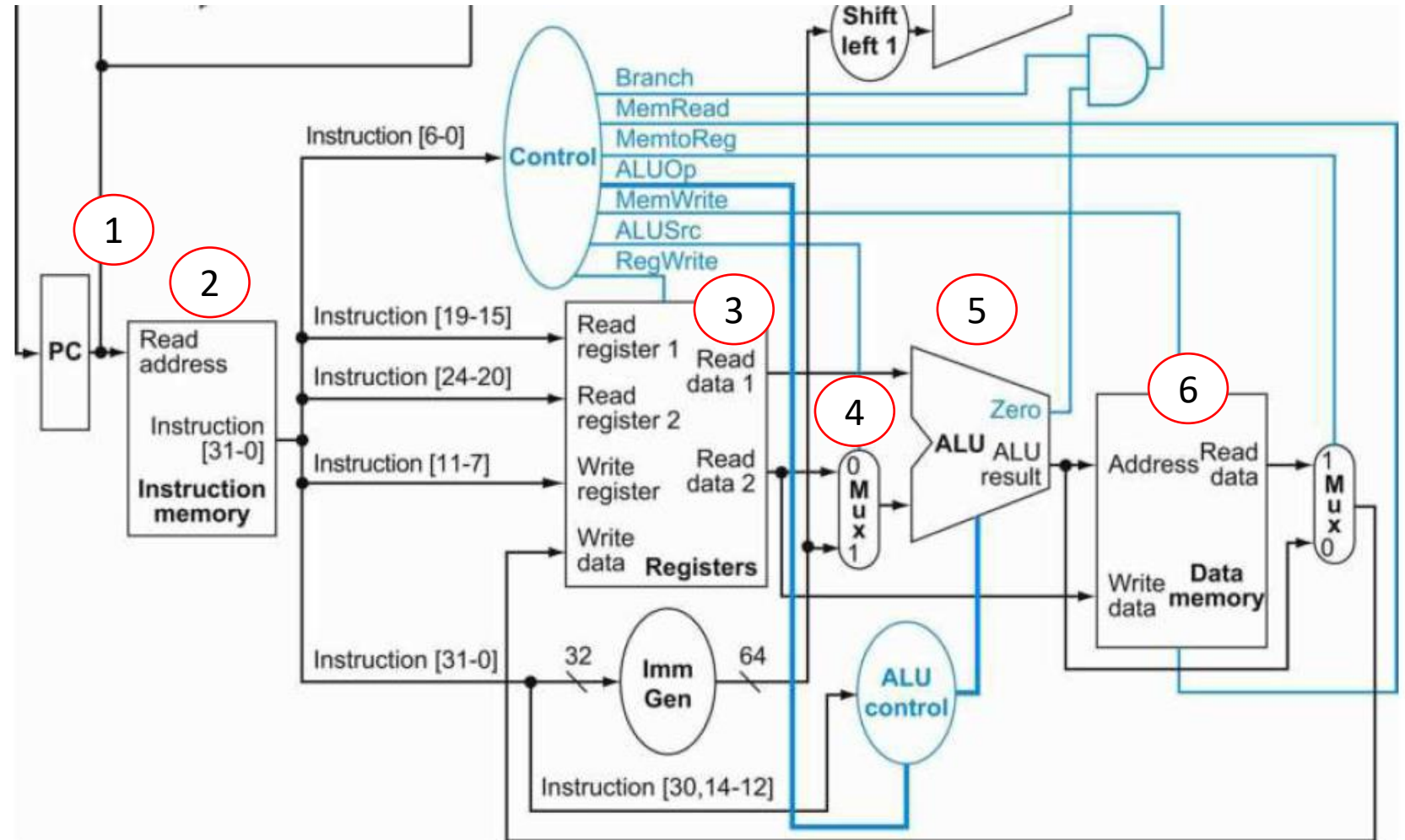
7. Mux for WB to *Register File* of result from Data Mem: 25 ps
8. Write back setup time *for Register File* registers: 20 ps

total 1d = 950ps

Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

sd Instruction
Path Delay



Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

sd Instruction:

1. PC Register Read delay following rising edge of clock: 30 ps
2. Instruction Memory: 250ps
3. Read Register File: 150 ps
4. Mux to pick data from either R2 or sign extended bit data from immediate field of instruction = 25 ps

[note - there is no number given for sign extension delay, so we can assume it is less than the register file delay of 150 ps]

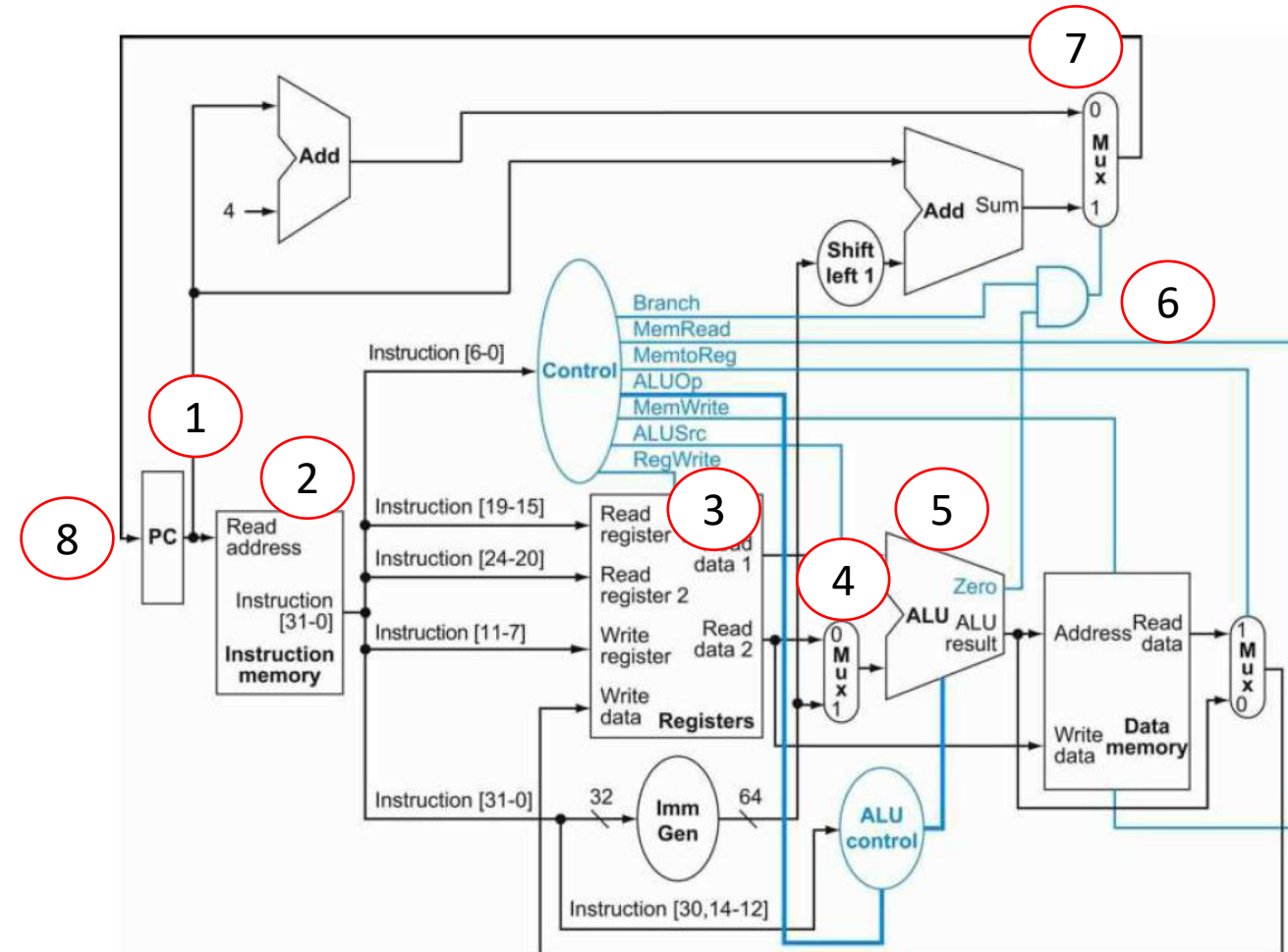
5. ALU delay: 200 ps
6. Write Data from R2 to Data Mem: 250 ps

Total for sd: 905 ps

Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

beq Instruction Path Delay



Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

beq Instruction:

1. PC Register Read delay following rising edge of clock: 30 ps
2. Instruction Memory: 250ps
3. Read Register File: 150 ps
4. Mux to pick data from either R2 or sign extended bit data from immediate field of instruction = 25 ps
[note - there is no number given for sign extension delay, so we can assume it is less than the register file delay of 150 ps]
5. ALU delay: 200 ps
6. Single gate delay AND of 'zero' output from ALU and 'Branch' control output from Control unit: 5ps
7. Mux for Branch address to PC: 25 ps
8. **Write back setup time for PC register: 20 ps** **Total for beq: 705 ps**

How can we make these faster?

- Suppose you could build a CPU where the clock cycle time was different for each instruction.
- What would the speedup of this new CPU be over the CPU presented in Figure 4.21 (in RISC-V text) given the instruction mix below?

R-type/I-type (non-ld)	ld	sd	beq
52%	25%	11%	12%

- Consider the addition of a multiplier to the CPU shown in Figure This addition will add 300 ps to the latency of the ALU, but will reduce the number of instructions by 5% (because there will no longer be a need to emulate the multiply instruction).

Cycle time = Instruction Latency?

- Cycle time of a single cycle processor were only as long as the instruction latency – with asynchronous clocking?
- No need to make cycle time equal to latency of slowest instruction
- Single cycle processor could be faster

R-type/I-type (non-lb)	lb	sb	beq
52%	25%	11%	12%

- Assuming only the above 4 instructions used with the given frequencies of their occurrence.
- Cycle time = $T_{\text{R-type}} \times f_{\text{R-type}} + T_{\text{lb}} \times f_{\text{lb}} + T_{\text{sb}} \times f_{\text{sb}} + T_{\text{beq}} \times f_{\text{beq}}$
- = $700\text{ps} \times 0.52 + 950\text{ps} \times 0.25 + 905\text{ps} \times 0.11 + 705\text{ps} \times 0.12 = 785.6\text{ps}$
- **Speed-up = $950\text{ps} / 785.6 \text{ ps} = 1.21$**

Make the Slowest Instruction [**ld**] faster?

- Eliminate the address calculation step in `ld`, `sd`
- So no instruction would use both – ALU and the Data Memory
- *Smaller cycle time* since slowest instruction is faster
- *But more instructions* since memory address for `ld`, `sd` must be calculated in a previous/following instruction with `ld/add` or `sd/add` combinations

What is new Cycle Time ?

- New latency for *ld* instruction:

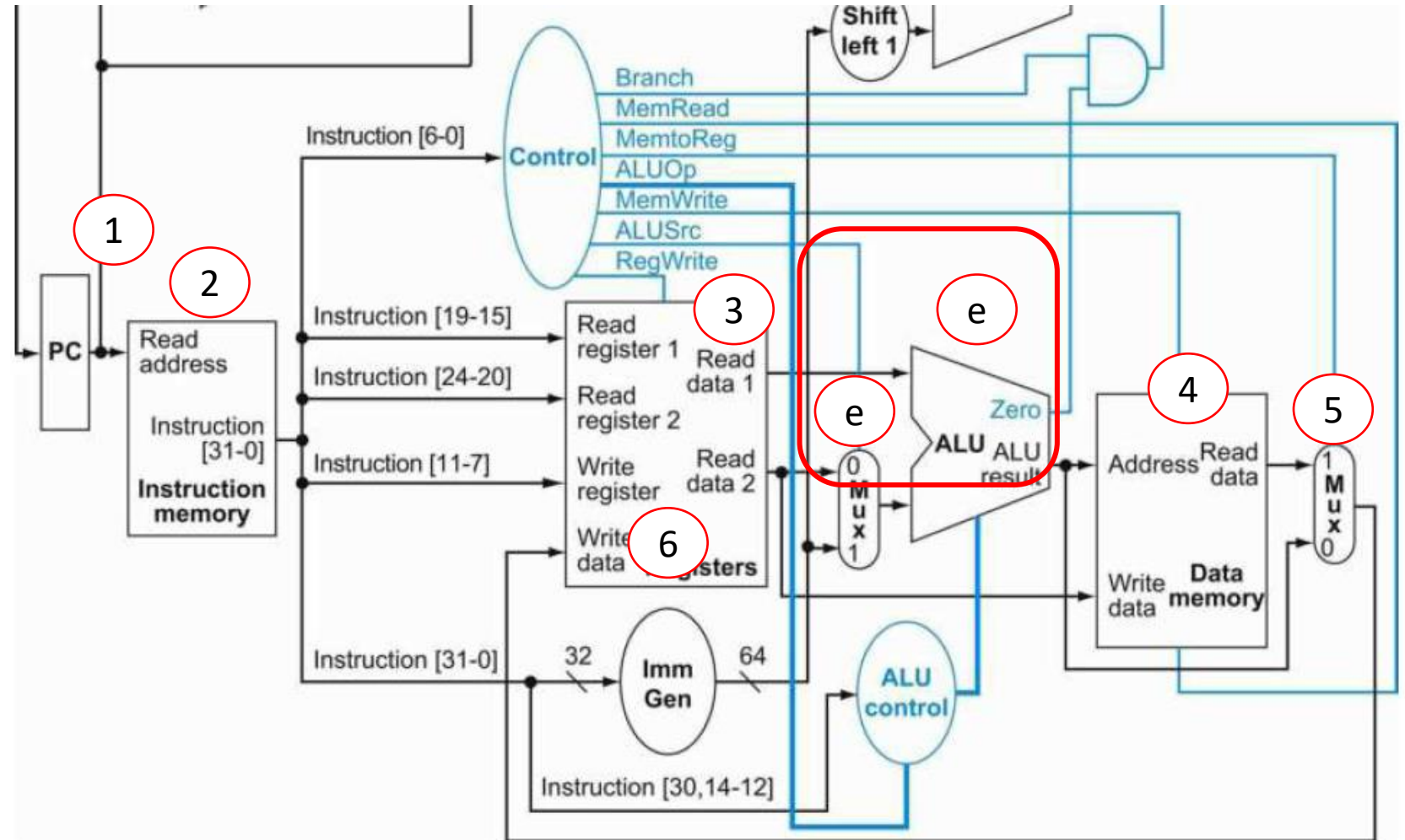
1. PC Register Read delay following rising edge of clock: 30 ps
2. Instruction Memory: 250ps
3. Read Memory address from Register File: 150 ps
 - eliminated* - Mux to pick data from either R2 or sign extended bit data from immediate field of instruction = 25 ps
 - eliminated* - ALU delay: 200 ps
4. Read data from Data Mem, whose address provided by ALU: 250 ps
5. Mux for WB *to Register File* of result from Data Mem: 25 ps
6. Write back setup time *for Register File* registers: 20 ps

total *ld* = 725ps

Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

new ld Instruction
Path Delay



New **sd** instruction latency

For the new **sd** instruction , *items 5 and 6 below proceed in parallel:*
with the faster of these (ALU delay) removed from the critical path

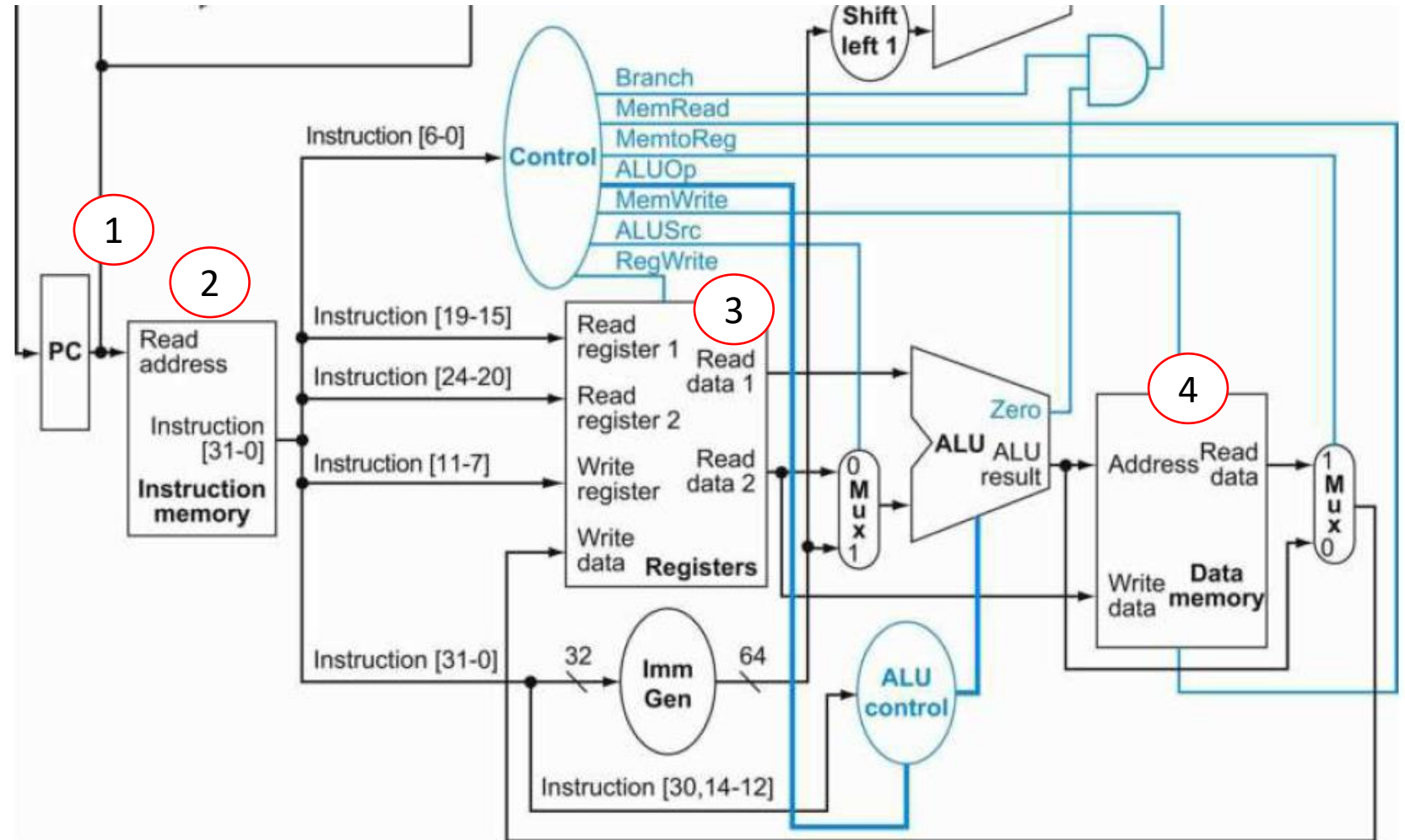
1. PC Register Read delay following rising edge of clock: 30 ps
2. Instruction Memory: 250ps
3. Read Memory address from Register File: 150 ps
eliminated - Mux to pick data from either R2 or sign extended bit data from immediate field of instruction = 25 ps
eliminated - ALU delay: 200 ps
4. Write data to Data Mem, whose address provided by ALU: 250 ps

total sd = 680ps

Processor Datapath Latencies

I-Mem/ D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

sd Instruction
Path Delay



Impact of Adding Functionality to speed

- Addition of Multiplier hardware adds 300ps to latency of ALU
- Reduces the number of instructions by 5% - no need to emulate multiply instruction
- Cycle Time:
- Previously – 950ps ; w Multiplier 1250ps
- Assuming CPI of 1, time to execute program requires 5% fewer cycles
- so Execution time from fewer instructions = $0.95 \times IC_{old} \times T_{cycle}$
- Speedup $ET_{old}/ET_{new} = [IC_{old} \times T_{cycle_old}]/[0.95 \times IC_{old} \times T_{cycle_new}]$
- $= 950/[0.95 \times 1250] = \mathbf{0.8}$

Pipelining

- Given: Latencies and Instruction mix

IF	ID	EX	MEM	WB
250 ps	350 ps	150 ps	300 ps	200 ps

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

- Cycle time in a pipelined and non-pipelined processor?
 - pipelined 350ps (slowest stage) non-pipelined 1250ps
- Latency of an ld instruction in a pipelined and non-pipelined processor?
 - pipelined and non-pipelined = 1250ps

Problems in HW 4 Part B

- How do we determine program execution time penalty from Data Hazards – with *and* without forwarding hardware overheads
- How do structural hazards from use of a single Memory Array increase CPI & how is this Structural Hazard resolved by instruction scheduling?
- Lowering CPI penalty due to Load/Store Data Use Hazards by overlapping EX and MEM stages of the RISC pipeline
- Load-use-data hazard resolution for given instruction sequences including branches

Speedup from Forwarding hardware resolving data hazards

Consider a version of the pipeline from *Section 4.5 in RISC-V text* that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). Suppose that (after optimization) **a typical n -instruction program requires an additional $0.4 \cdot n$ NOP instructions** to correctly handle data hazards.

1.1 Suppose that the cycle time of this pipeline without forwarding is 250 ps. Suppose also that **adding forwarding hardware will reduce the number of NOPs from $.4 \cdot n$ to $.05 \cdot n$, but increase the cycle time to 300 ps**. What is the speedup of this new pipeline compared to the one without forwarding?

Making the implicit assumption that the average CPI for the n -instruction Program is 1 and assuming no data hazards are encountered, in a Pipeline executing at cycle time of 250 ps takes

- $T_0 = n \times 250 \text{ ps}$
- Assuming 0.4 NOPS per instruction, in optimized program without forwarding
- $T_1 = 1.4n \times 250 \text{ ps}$
- Adding forwarding hardware reduces the number of NOPS to $0.05n$ but also increases the cycle time to 300ps yielding total execution time of:
- $T_2 = 1.05 \times n \times 300 \text{ ps}$
- Therefore, speedup = $T_1/T_2 = [1.4 \times 250] / [1.05 \times 300] = 1.11$

Limits on minimum NOPs while still improving exec time

Consider a version of the pipeline from *Section 4.5 in RISC-V text* that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). Suppose that (after optimization) a typical n -instruction program requires an additional $0.4 \times n$ NOP instructions to correctly handle data hazards.

1.2 Different programs will require different amounts of NOPs. How many NOPs (as a percentage of code instructions) can remain in the typical program before that program runs slower on the pipeline *with forwarding*?

Without Forwarding, typical program execution time,

- $T_0 = 1.4n \times 250\text{ps}$

With Forwarding, assuming γ = number of NOPs per instruction after forwarding.
Execution time:

- $T_F = (1 + \gamma) \times n \times 300\text{ps}$

Maximum number of NOPs per instruction, γ for faster hardware forwarding is restricted by

- $T_0 \geq T_F$

- $1.4n \times 250\text{ps} \geq (1 + \gamma) \times n \times 300\text{ps}$

- Solving, $\gamma \leq 0.167$

Limits on minimum NOPs while still improving exec time

Consider a version of the pipeline from *Section 4.5 in RISC-V text* that does not handle data hazards (i.e., the [programmer is responsible for addressing data hazards by inserting NOP instructions](#) where necessary). Suppose that (after optimization) [a typical n- instruction program requires an additional \$0.4 \cdot n\$ NOP instructions](#) to correctly handle data hazards.

1.3 Repeat 1.2; however, this time [let x represent the number of NOP instructions relative to n](#). (In 1.2, x was equal to 0.4) Your answer will be with respect to x.

- Replacing 0.4 for x in Problem 1.2, we get

- $(1+x)n \times 250\text{ps} \geq (1 + \gamma) \times n \times 300\text{ps}$

- solving for γ , we get the limits on min NOPs as a function of the NOPs/instruction in the program without forwarding

- More NOPs permissible with forwarding to improve performance for higher x
- If x goes below a minimum threshold, it is pointless to use forwarding hardware

[\[next 2 questions\]](#)

$$\gamma < (250x - 50)/300$$

[A]

Limits on minimum NOPs while still improving exec time

Consider a version of the pipeline from *Section 4.5 in RISC-V text* that does not handle data hazards (i.e., the **programmer is responsible for addressing data hazards by inserting NOP instructions** where necessary). Suppose that (after optimization) **a typical n -instruction program requires an additional $0.4*n$ NOP instructions** to correctly handle data hazards.

1.4 Can a program with only $.075*n$ NOPs possibly run faster on the pipeline with forwarding? Explain why or why not.

- In the best case, where **forwarding results in no NOPS at all**, the execution time will be $300n$ which is more than $250 \times 1.075n$
- So the given program cannot possibly run faster since the NOPs per instruction of the optimized program of 0.075 is too low to improve performance with forwarding – **essentially too few data hazards to begin with**

Limits on minimum NOPs while still improving exec time

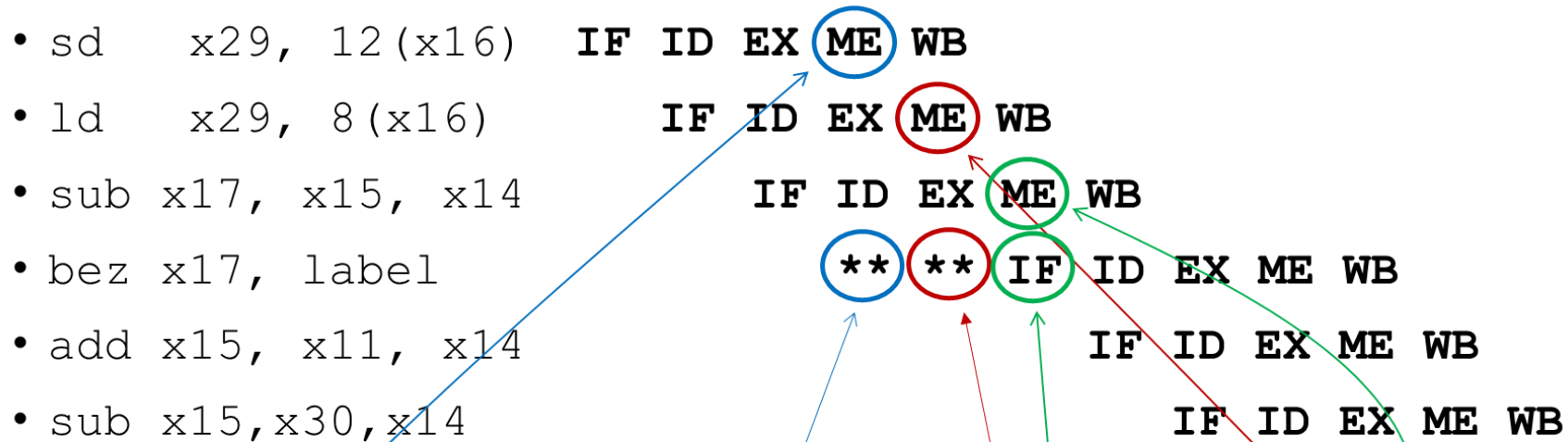
Consider a version of the pipeline from *Section 4.5 in RISC-V text* that does not handle data hazards (i.e., the [programmer is responsible for addressing data hazards by inserting NOP instructions](#) where necessary). Suppose that (after optimization) [a typical \$n\$ -instruction program requires an additional \$0.4*n\$ NOP instructions](#) to correctly handle data hazards.

1.5 At minimum, how many NOPs (as a percentage of code instructions) must a program have before it can possibly run faster on the pipeline with forwarding?

- The larger the number of NOPs per instruction, x without forwarding, the easier it is for forwarding hardware to work and lower the number of NOPs per instruction.
- As the number of NOPs per instruction without forwarding, x decreases, it becomes physically impossible to go faster with forwarding if there are very few data hazards. This minimum occurrence of data hazards is when x is at a minimum corresponding to the case of $\gamma=0$ in equation [\[A\]](#) {slide 5}.
- This minimum value of x can be solved in [\[A\]](#) for $\gamma = 0$ as $x = 0.2$

Pipeline with Structural hazards

2.1 Draw a pipeline diagram to show where the given code above will stall.



Structural hazard when '**bez**' instruction attempts to read instruction from memory in same clock cycle when '**sd**' instruction attempts to write data to memory

Structural hazard encountered again when '**bez**' instruction attempts to read instruction from memory in same clock cycle when '**ld**' instruction attempts to read data to memory

No structural hazard encountered in next cycle when '**bez**' instruction attempts to read instruction from memory in same clock cycle as '**sub**' instruction because **sub** instruction (R-Type) does not access memory

Pipeline with Structural hazards

2.2 In general, is it possible to reduce the number of stalls/NOPs resulting from this structural hazard by reordering code?

- The conflict for access to Memory will always be present between the first RISC pipeline stage and the Fourth – from 3 instructions upstream, *if that first instruction is a load/store instruction*.
- It is not possible to lower NOPs by reordering the code since every instruction 3 cycles after a load/store instruction must access memory for IF

Pipeline with Structural hazards

2.3 Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding NOPs to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.

- The only way to resolve this structural hazard in hardware is by adding a separate memory array for Data and for Instructions.
- Adding NOPs to avoid the structural hazard can also resolve it but adds significant penalty to execution time

Pipeline with Structural hazards

2.4 Approximately how many stalls would you expect this structural hazard to generate in a typical program? (Use the instruction mix shown below)

R-type/I-type (non-ld)	ld	sd	beq
52%	25%	11%	12%

- 36% of the instructions will stall corresponding to all of the load/store instructions in a typical program represented in table above

Overlapping EX and MEM stages of the RISC pipeline

3. If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the ALU. (See Problem 4 in HW 4) As a result, the MEM and EX stages can be overlapped and the pipeline has only four stages.

3.1 How will the reduction in pipeline depth affect the cycle time?

- Cycle time is dependent on the latency of the slowest of the 5 RISC pipeline stages **not on the number of stages**. Cycle time will not change by reduction of pipeline depth

Overlapping EX and MEM stages of the RISC pipeline

3.2 How might this change improve the performance of the pipeline?

- By overlapping the MEM with the EX pipeline stages, fewer pipeline stage enables the latency of an instruction to improve.
- Also, load-use-data hazards that require a NOP cycle inserted between a load instruction and an instruction that uses this data read from memory, **can eliminate the need of this NOP cycle** potentially lowering the Execution Time **penalty in NOPS per instruction for load-use-data hazards**

Overlapping EX and MEM stages of the RISC pipeline

3.3 How might this change degrade the performance of the pipeline?

- Removing the offset from `ld/sd` instructions can require `addi` to be used/paired with `ld/sd` instructions thus increasing the number of instructions in the program
- If the improvement in execution time from resolving load-use-data hazards is less than degradation from more instructions, this overlap would not be implemented

Load-use-data hazard resolution

4. Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

Choice 1:

ld x11, 0(x12):	IF	ID	EX	ME	WB	
add x13, x11, x14:		IF	ID	EX	..	ME WB
or x15, x16, x17:			IF	ID	..	EX ME WB

Choice 2:

ld x11, 0(x12):	IF	ID	EX	ME	WB	
add x13, x11, x14:		IF	ID	..	EX ME	WB
or x15, x16, x17:			IF	..	ID EX ME	WB

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
ld x11, 0(x12):	IF	ID	EX	ME	WB			
add x13, x11, x14:		IF	ID	EX	..	ME	WB	
or x15, x16, x17:			IF	ID	..	EX	ME	WB

ld x11, 0(x12):	IF	ID	EX	ME	WB			
add x13, x11, x14:		IF	ID	..	EX	ME	WB	
or x15, x16, x17:			IF	..	ID	EX	ME	WB

Choice 2 because:

In the first sequence, content of register x11 becomes available only at the end of CC4 but is being used at the start of CC4 in the execute stage of the add instruction – this is a 'load-use-data-hazard'. Insertion of a stall in CC5 is too late

In the second sequence, content of register x11 which becomes available at the end of CC4 is used at the start of CC5 (and not CC4 as above) since the EX stage has been delayed by insertion of a stall in CC4

Load-use-data hazard resolution

5. Consider the following loop.

```
LOOP: ld    x10, 0(x13)
      ld    x11, 8(x13)
      add   x12, x10, x11
      subi  x13, x13, 16
      bnez  x12, LOOP
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, that the pipeline has full forwarding support, and that branches are resolved in the EX (as opposed to the ID) stage.

Load-use-data hazard resolution

5.1 Show a pipeline execution diagram for the first two iterations of this loop.

[1] Since perfect branch prediction is used, we do not lose any cycles due to branch hazards – that is, the branch is always predicted and taken correctly

[2] Availability of full forwarding support is assumed, so we can assume data hazards that can be resolved with forwarding do not stall the pipeline

[3] Load-use-hazards cannot be resolved and are identified in next slide in red boxes – resolved by stalling the pipeline

[4a] pipeline stages unused by any instruction are identified in blue

[4b] any clock cycle that does not have all of the pipeline stages utilized is identified with 'N'

Load-use-data hazard resolution

5.1 Show a pipeline execution diagram for the first two iterations of this loop

5.2 Mark pipeline stages **that do not perform useful work**. How often while the pipeline is full do we have a cycle **in which all five pipeline stages are doing useful work**? (Begin with the cycle during which the `subi` is in the IF stage. End with the cycle during which the `bnez` is in the IF stage.)

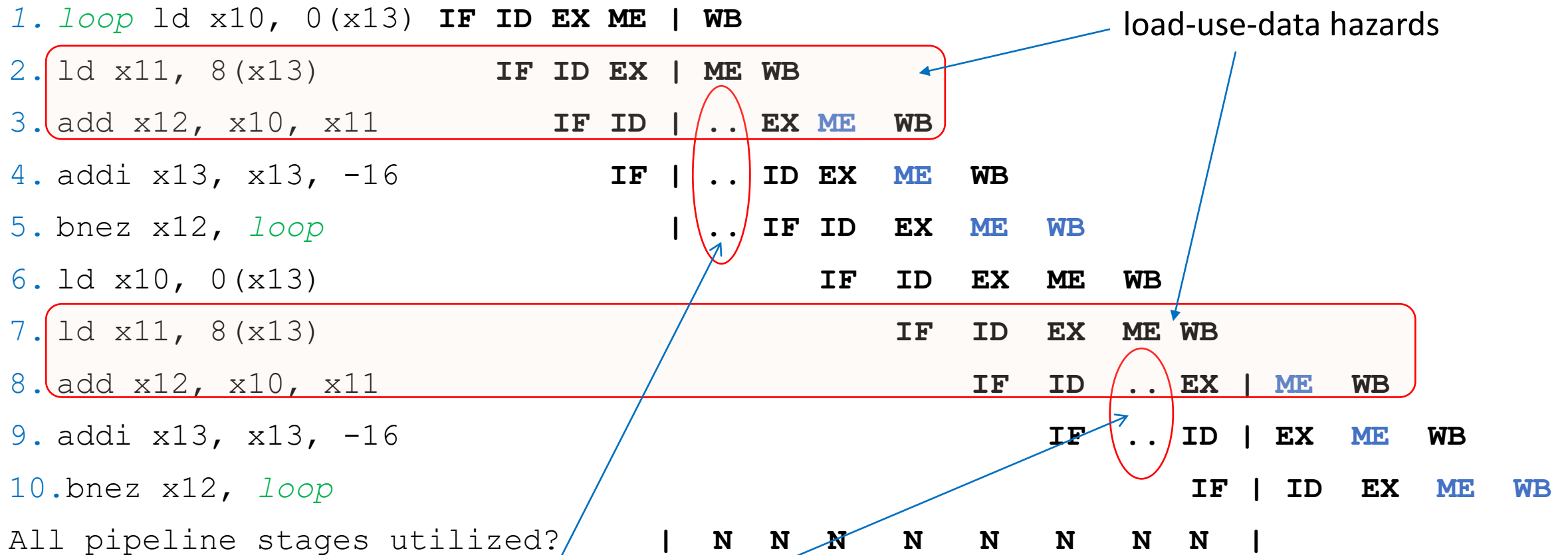
[1] Since perfect branch prediction is used, **we do not lose any cycles due to branch hazards** – that is, the branch is **always predicted and taken correctly**

[2] Availability of **full forwarding support** is assumed, so we can assume **data hazards that can be resolved with forwarding do not stall the pipeline**

[3] Load-use-hazards **cannot be resolved** and are identified in next slide in red boxes – **resolved by stalling the pipeline**

[4a] pipeline stages **unused by any instruction** are **identified in blue**

[4b] any clock cycle **that does not have all of the pipeline stages utilized** is identified with 'N'



Stalls to the pipeline execution to resolve load-use-data hazards

RAW Data Dependencies

6. This exercise is intended to help you understand the **cost/complexity/performance trade-offs of forwarding** in a pipelined processor. Problems in this exercise refer to pipelined datapaths from *Figure 4.53 in RISC-V text (reproduced below)*. These problems assume that, of all the instructions executed in a processor, **the following fraction of these instructions has a particular type of RAW data dependence**

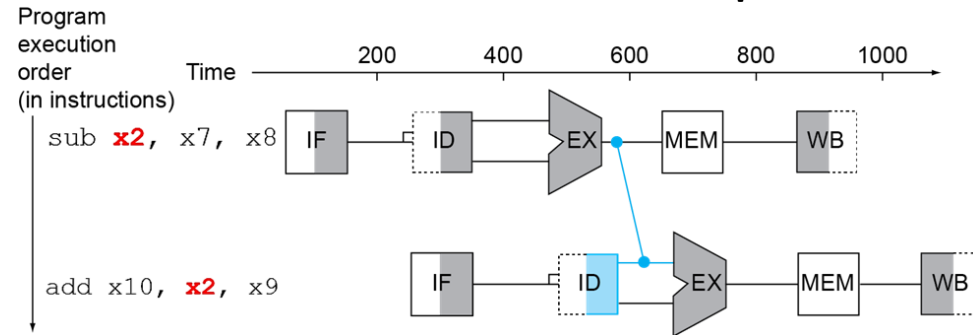
EX to 1 st Only	MEM to 1 st Only	EX to 2 nd Only	MEM to 2 nd Only	EX to 1 st and EX to 2 nd
5%	20%	5%	10%	10%

RAW Data Dependencies

6.1 For each RAW dependency listed in table previous slide, give a sequence of **at least three assembly statements** that exhibits that dependency.

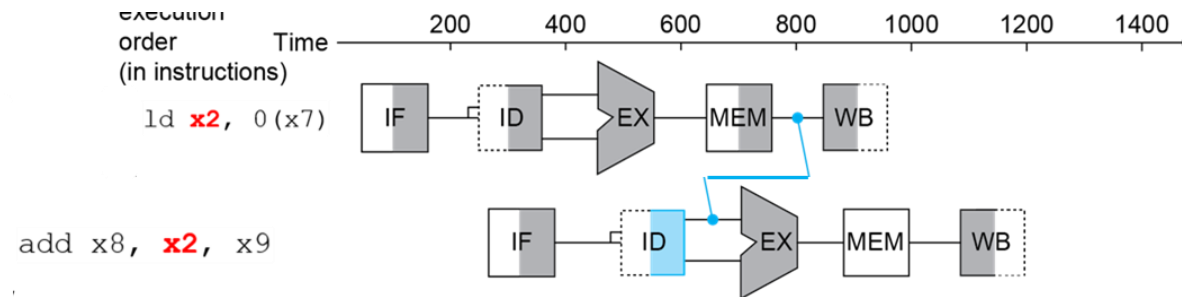
(1) EX to 1st only:

```
sub x2, x7, x8
add x10, x2, x9
add x27, x28, x29
```



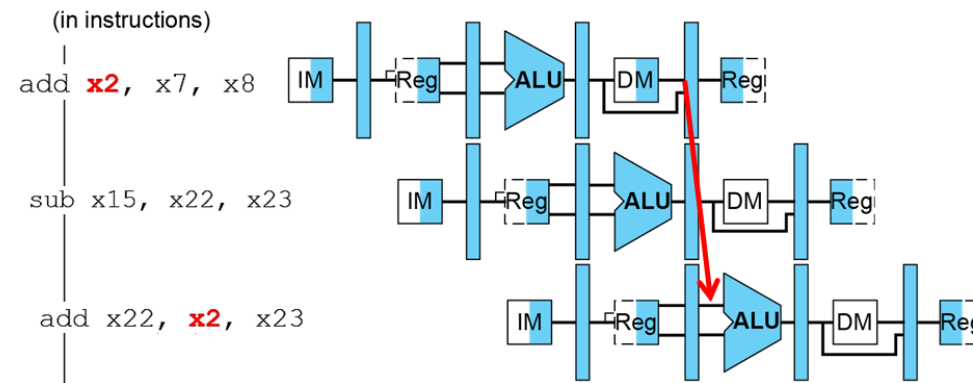
(2) MEM to 1st only:

```
ld x2, 0(x7)
add x8, x2, x9
sub x15, x22, x23
```



(3) EX to 2nd only:

```
add x2, x7, x8
sub x15, x22, x23
add x22, x2, x23
```



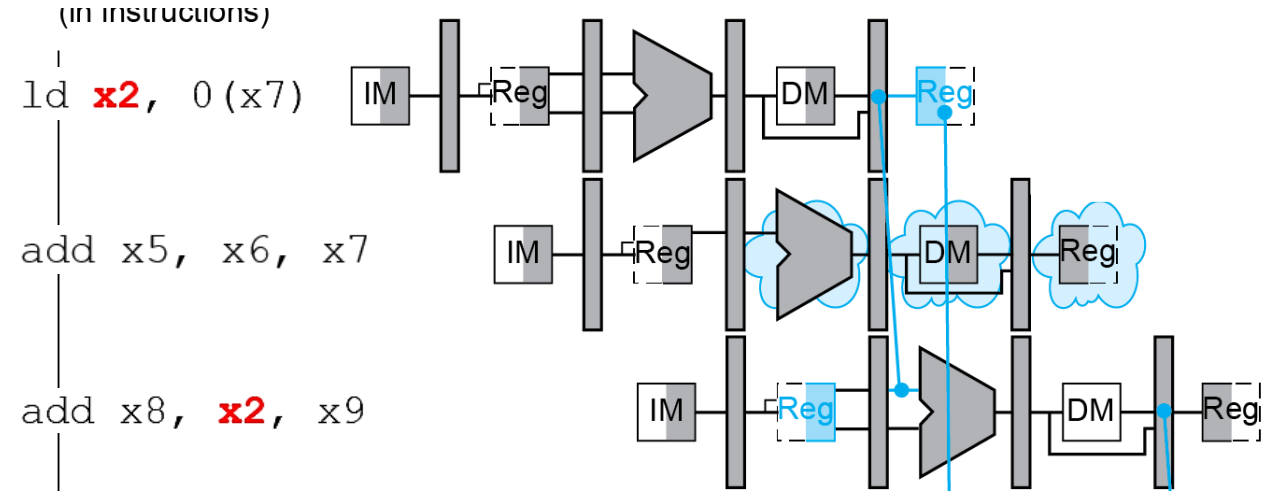
RAW Data Dependencies

(4) MEM to 2nd only:

ld **x2**, 0(x7)

add x5, x6, x7

add x8, **x2**, x9

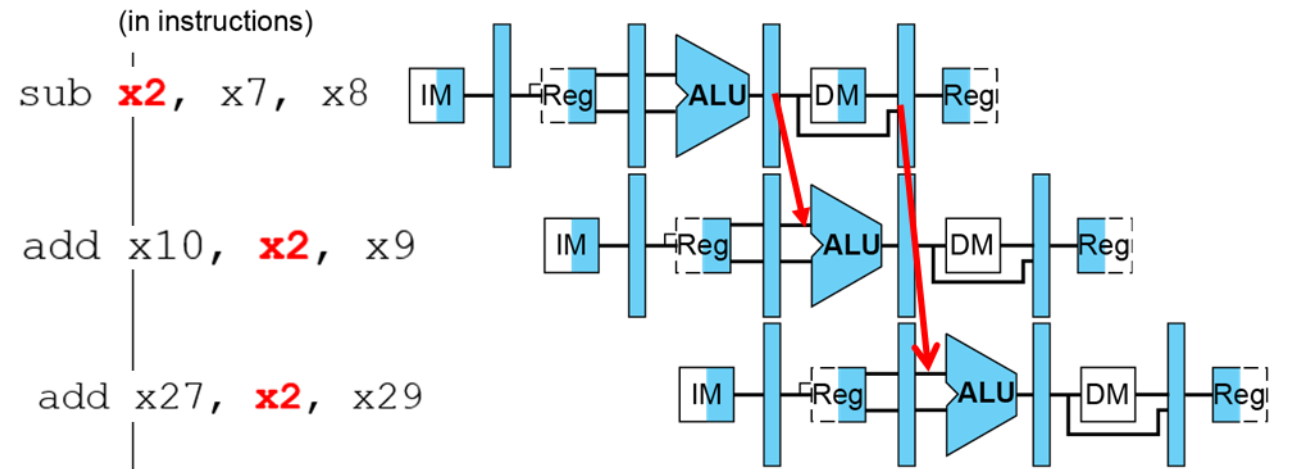


(5) EX to 1st and EX to 2nd :

sub **x2**, x7, x8

add x10, **x2**, x9

add x27, **x2**, x29



RAW Data Dependencies

6.2, 6.3 For each RAW dependency above, **how many NOPs would need to be inserted to allow your code from 6.1 to run correctly on a pipeline with no forwarding or hazard detection?** Show where the NOPs could be inserted.

EX to 1st only:

sub **x2**, x7, x8

NOP

NOP

add x10, **x2**, x9

add x27, x28, x29

MEM to 1st only:

ld **x2**, 0(x7)

NOP

NOP

add x8, **x2**, x9

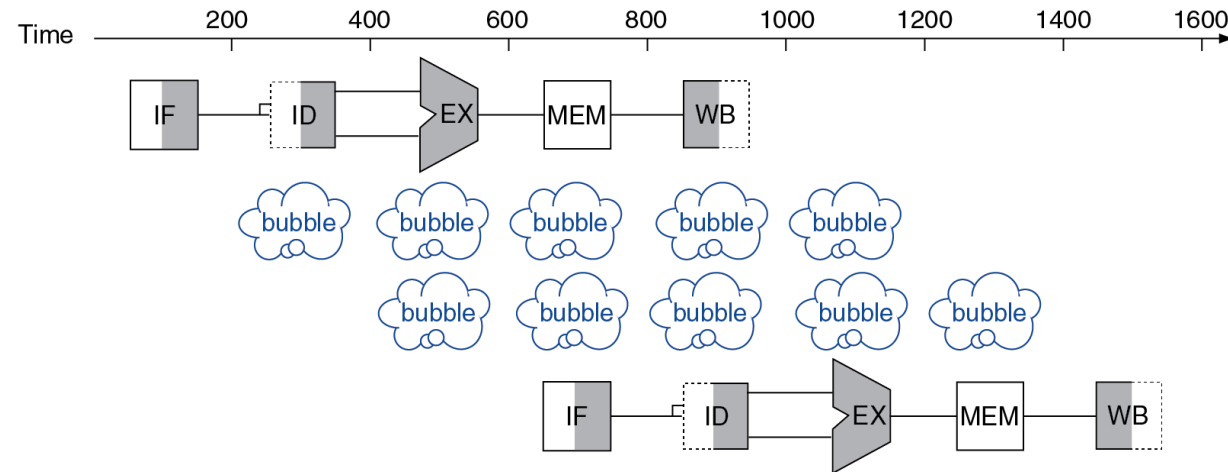
EX to 2nd only:

add **x2**, x7, x8

sub x15, x22, x23

NOP

add x22, **x2**, x23



RAW Data Dependencies

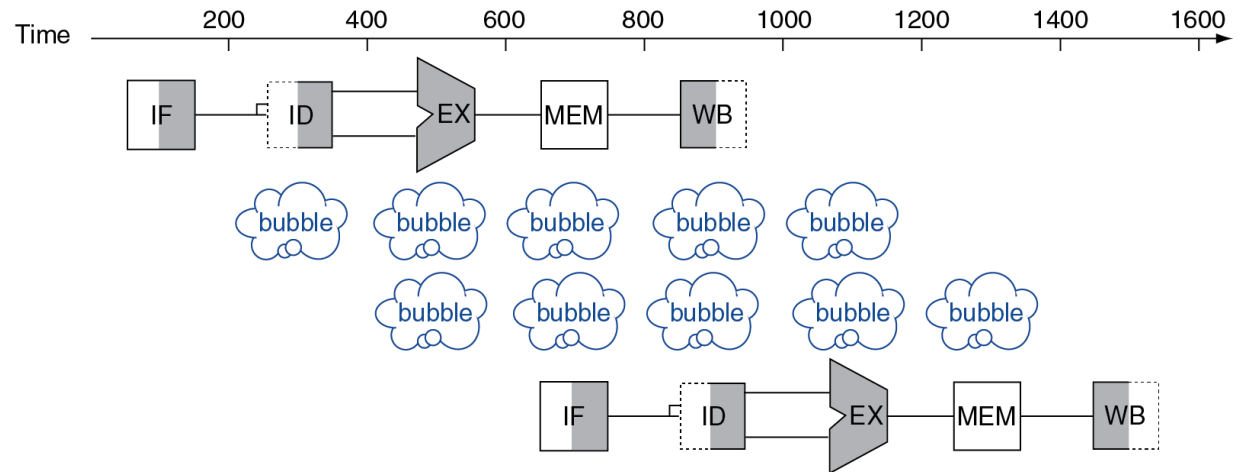
MEM to 2nd only:

ld **x2**, 0(x7)

add x5, x6, x7

NOP

add x8, **x2**, x9



EX to 1st and EX to 2nd:

sub **x2**, x7, x8

NOP

NOP

add x10, **x2**, x9

add x27, **x2**, x29

CPI Without Forwarding

6.4 Assuming no other hazards, what is the CPI for the program described by the table above when run on a pipeline with no forwarding? What percent of cycles are stalls? (For simplicity, assume that all necessary cases are listed above and can be treated independently.)

EX to 1 st Only	MEM to 1 st Only	EX to 2 nd Only	MEM to 2 nd Only	EX to 1 st and EX to 2 nd
5%	20%	5%	10%	10%

- Taking a weighted average of the number of NOPs for each from 6.2 gives $0.05 * 2 + 0.2 * 2 + 0.05 * 1 + 0.1 * 1 + 0.1 * 2 = 0.85$ NOPs per instruction
- a CPI of 1.85.
- So, $0.85 / 1.85$ cycles = 46%, are NOPs.

CPI improvement with Full Forwarding

6.5 What is the CPI if we use full forwarding (forward all results that can be forwarded)? What percent of cycles are stalls?

EX to 1 st Only	MEM to 1 st Only	EX to 2 nd Only	MEM to 2 nd Only	EX to 1 st and EX to 2 nd
5%	20%	5%	10%	10%

- The only RAW dependency that cannot be handled by load-use-data hazards with forwarding is from the MEM stage to the next instruction
- 20% of instructions will generate one NOP for a CPI of 1.2.
- 0.2 out of 1.2 cycles = 17% NOPs

Partial Forwarding – use EX/MEM only

6.6 Let us assume that we cannot afford to have **three-input** multiplexers that are needed for full forwarding. We have to decide **if it is better to forward only from the EX/MEM pipeline register** (next-cycle forwarding) **or only from the MEM/WB pipeline register** (two-cycle forwarding). What is the CPI for each option?

If we forward from the **EX/MEM register only**, we have the following stalls/NOPs

EX to 1st: 0 (eq 1) in slide 22

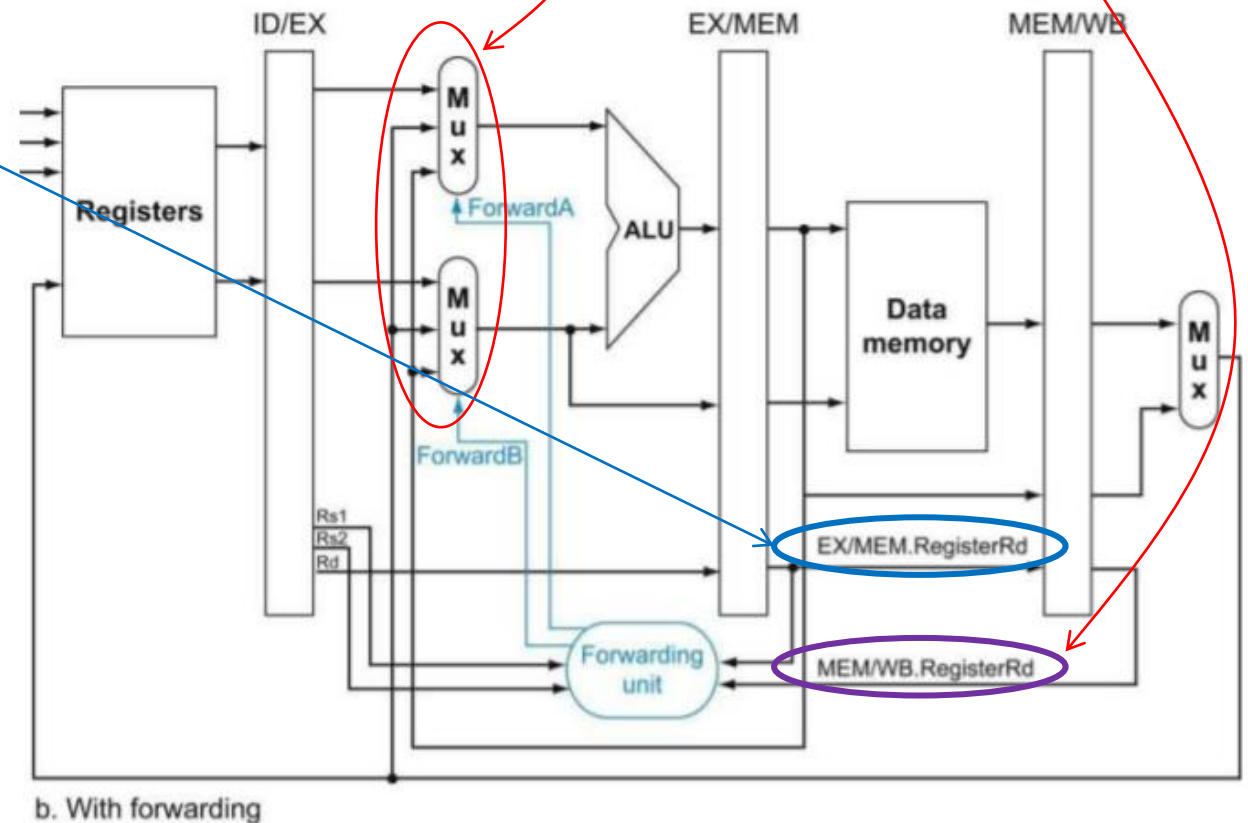
This hazard needs the **EX/MEM** register with which the forwarding unit eliminates the NOP, so NOPs = 0

MEM to 1st: 2 (eq 2) [must insert 2 NOPs] This hazard needs the **MEM/WB** register (load-use-data hazard) and cannot be resolved, so 2 NOPs must be inserted

EX to 2nd: 1 (eq 3) This hazard needs the **MEM/WB** register and cannot be resolved so a NOP must be inserted

MEM to 2nd: 1 (eq 4) This hazard needs the **MEM/WB** register and cannot be resolved, so a NOP is inserted

EX to 1st and 2nd: 1 (eq 5) EX to 1st can be resolved with the **EX/MEM** register but the EX to 2nd needs the **MEM/WB** register and cannot be resolved so a NOP is inserted to resolve the EX to 2nd hazard



Partial Forwarding – use MEM/WB only

6.6 Let us assume that we cannot afford to have three-input multiplexers that are needed for full forwarding. We have to decide if it is better to forward only from the EX/MEM pipeline register (next-cycle forwarding) or only from the MEM/WB pipeline register (two-cycle forwarding). What is the CPI for each option?

If we forward from the MEM/WB register, we have the following stalls/NOPs

EX to 1st: 1 (eq 1) in slide 22

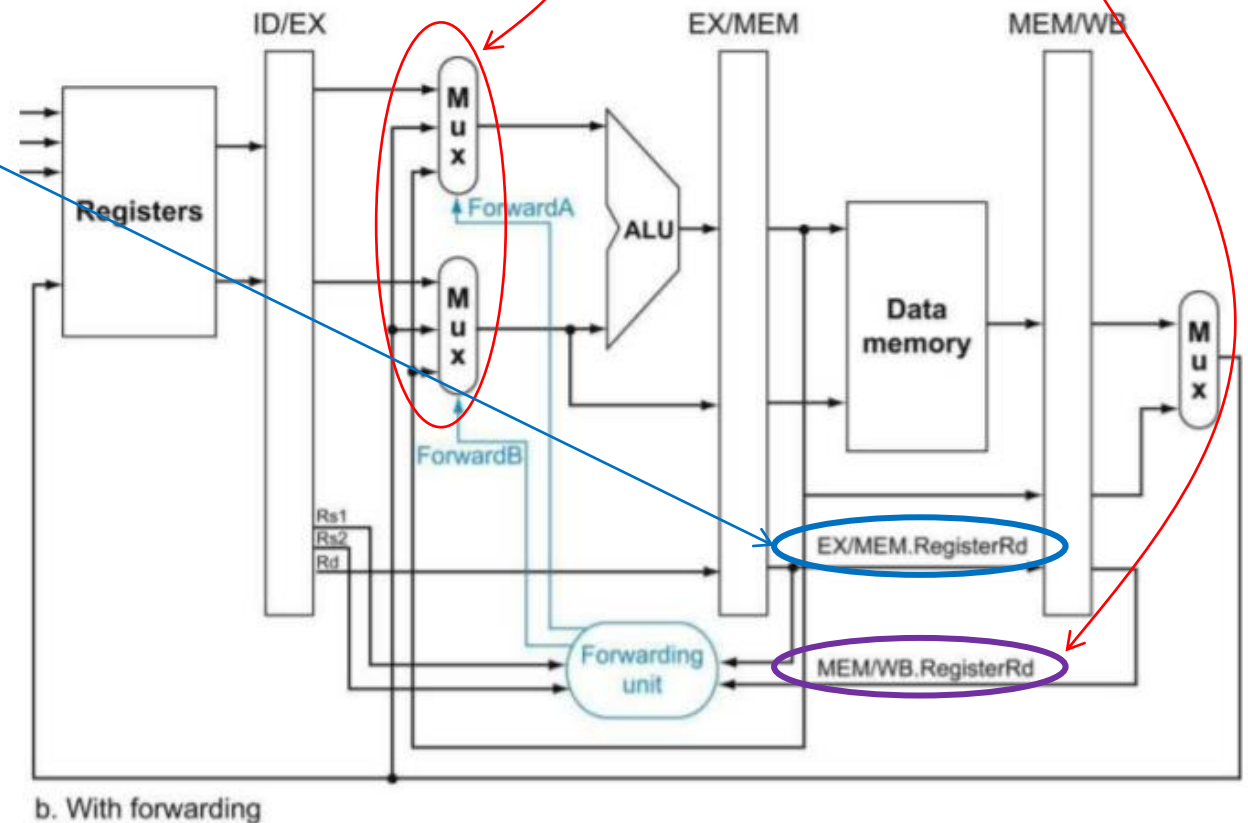
This hazard needs the EX/MEM register and can be resolved with the MEM/WB register, if the MEM/WB register forwards to the ALU after 1 NOP, so NOP = 1

MEM to 1st: 1 (eq 2) This hazard needs the MEM/WB register (load-use-data hazard) but 1 NOP must be inserted

EX to 2nd: 0 (eq 3) This hazard needs the MEM/WB register and is resolved with 0 NOPs

MEM to 2nd: 0 (eq 4) This hazard needs the MEM/WB register and is resolved with 0 NOPs

EX to 1st and 2nd: 1 (eq 5) EX to 1st cannot be resolved and needs 1 NOP, but the EX to 2nd can be resolved with the MEM/WB register



CPI with Partial Forwarding

- With MEM/WB register unavailable,
- Average NOPs of $0.05*0 + 0.2*2 + 0.05*1 + 0.10*1 + 0.10*1$
= 0.65 stalls/instruction

So, CPI = 1.65

- With EX/MEM register unavailable,
- Average NOPs of $0.05*1 + 0.2*1 + 0.1*1$
= 0.35 stalls/instruction

So, CPI = 1.35

Speedup with Full F, Partial F, NF

6.7 For the given hazard probabilities and pipeline stage latencies, (1) what is the speedup achieved by each type of forwarding (EX/MEM, MEM/WB, for full) as compared to a pipeline that has no forwarding?

- We calculated CPI for NF, EX/MEM only, MEM/WB only and with Full Forwarding
- Period for any of the above is the longest latency stage [FF needs 130 ps w FF unit]

	No Forwarding	EX/MEM	MEM/WB	Full Forwarding
CPI	1.85	1.65	1.35	1.2
Period	120ps	120ps	120ps	130ps
Time	222ps	198ps	162ps	156ps
Speedup	ref	1.12	1.37	1.42

Performance Limits with zero Hazards

6.8 What would be the additional speedup (relative to the fastest processor from 6.7) be if we added “timetravel” forwarding that eliminates all data hazards?

Assume that the yet-to-be-invented time-travel circuitry **adds 100 ps to the latency of the full-forwarding EX stage.**

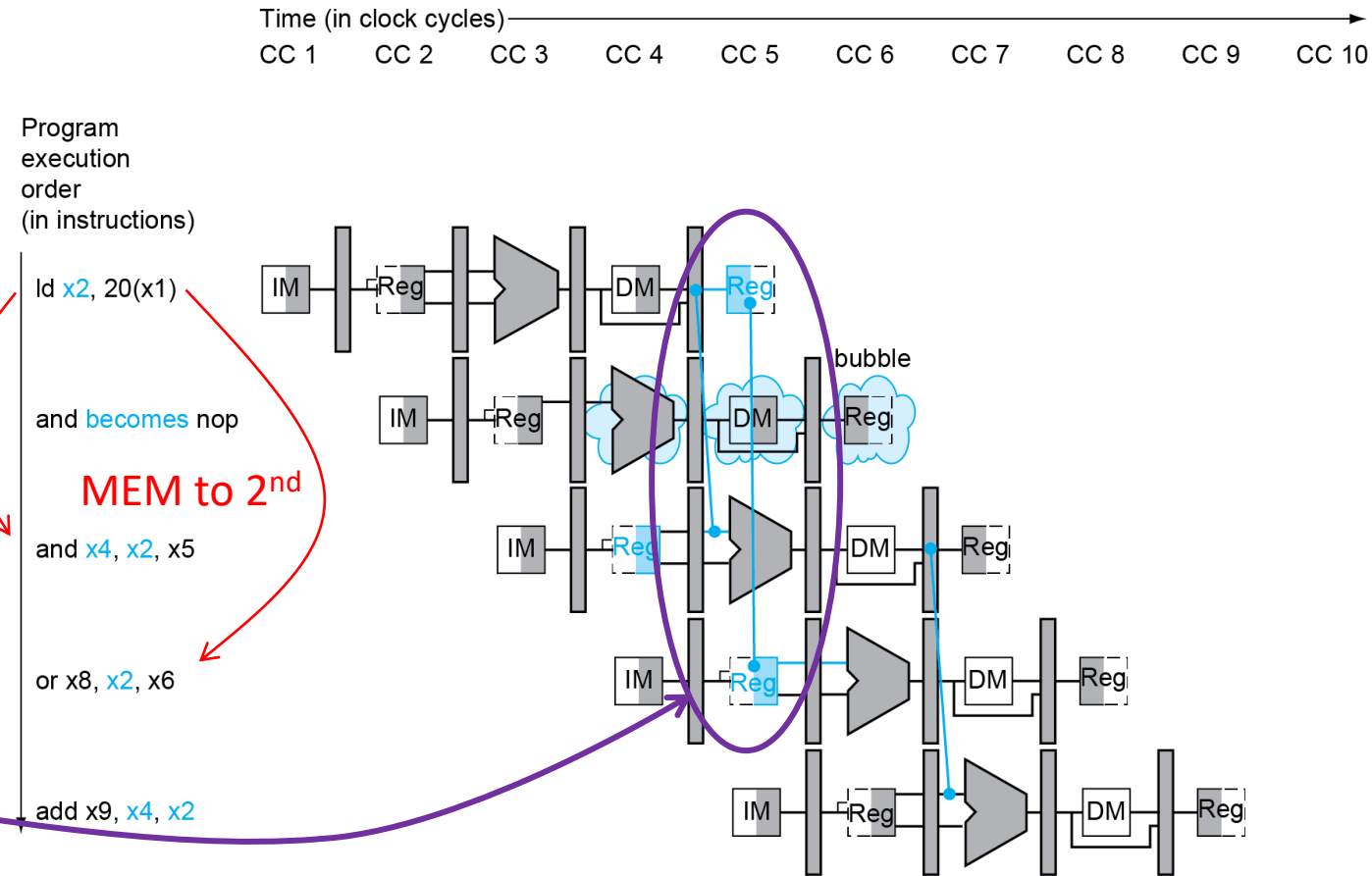
- CPI with full forwarding is 1.2
- CPI for “time travel” forwarding is 1.0 [Max with 0 Hazards]
- Clock period for full forwarding is 130
- **Clock period for zero hazards forwarding is 230**
- $\text{Speedup} = T_{\text{old}}/T_{\text{new}} = (1.2 \cdot 130) / (1 \cdot 230) = 0.68$
- Zero hazard forwarding actually slows the CPU

MEM to 1st and MEM to 2nd

6.9 The table of hazard types has separate entries for “EX to 1st” and “EX to 1st and EX to 2nd”. Why is there no entry for “MEM to 1st and MEM to 2nd”?

- All MEM to 1st instructions **will cause a stall** (load-use-data hazard)
- So instruction after `ld` must be a NOP
- So, 2nd instruction would have ID stage in same Clock Cycle as WB stage of `ld` instruction
- This is not a hazard anymore

So, if a MEM to 1st RAW hazard is present, the MEM to 2nd RAW hazard cannot be present!



Hazard Resolution

7. Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a five-stage pipelined datapath:

add **x15**, x12, x11

ld **x13**, 4(**x15**) **EX to 1st RAW Hazard**

ld x12, 0(x2)

or **x13**, x15, **x13** **MEM to 2nd RAW Hazard**

sd **x13**, 0(x15) **EX to 1st RAW Hazard**

7.1 If there is no forwarding or hazard detection, insert NOPs to ensure correct execution.

7.1 No Hazard Resolution – only NOP insertion

add x15, x12, x11

nop

nop

EX to 1st RAW Hazard resolution with 2 NOPs

ld x13, 4(x15)

ld x12, 0(x2)

nop

MEM to 2nd RAW Hazard resolution with 1 NOP

or x13, x15, x13

nop

nop

EX to 1st RAW Hazard resolution with 1 NOP

sd x13, 0(x15)

Code Optimization assuming NOP resolution

7.2 Now, change and/or rearrange the code to minimize the number of NOPs needed. You can assume register x17 can be used to hold temporary values in your modified code.

not possible to reduce the number of NOPs.

7.3 No Hazard Detection Unit

7.3 If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?

- The code executes correctly.
- Hazard detection relevant only to insert a stall for load-use-data hazards
- The given instruction sequence does not have **ld** followed by use of register written into