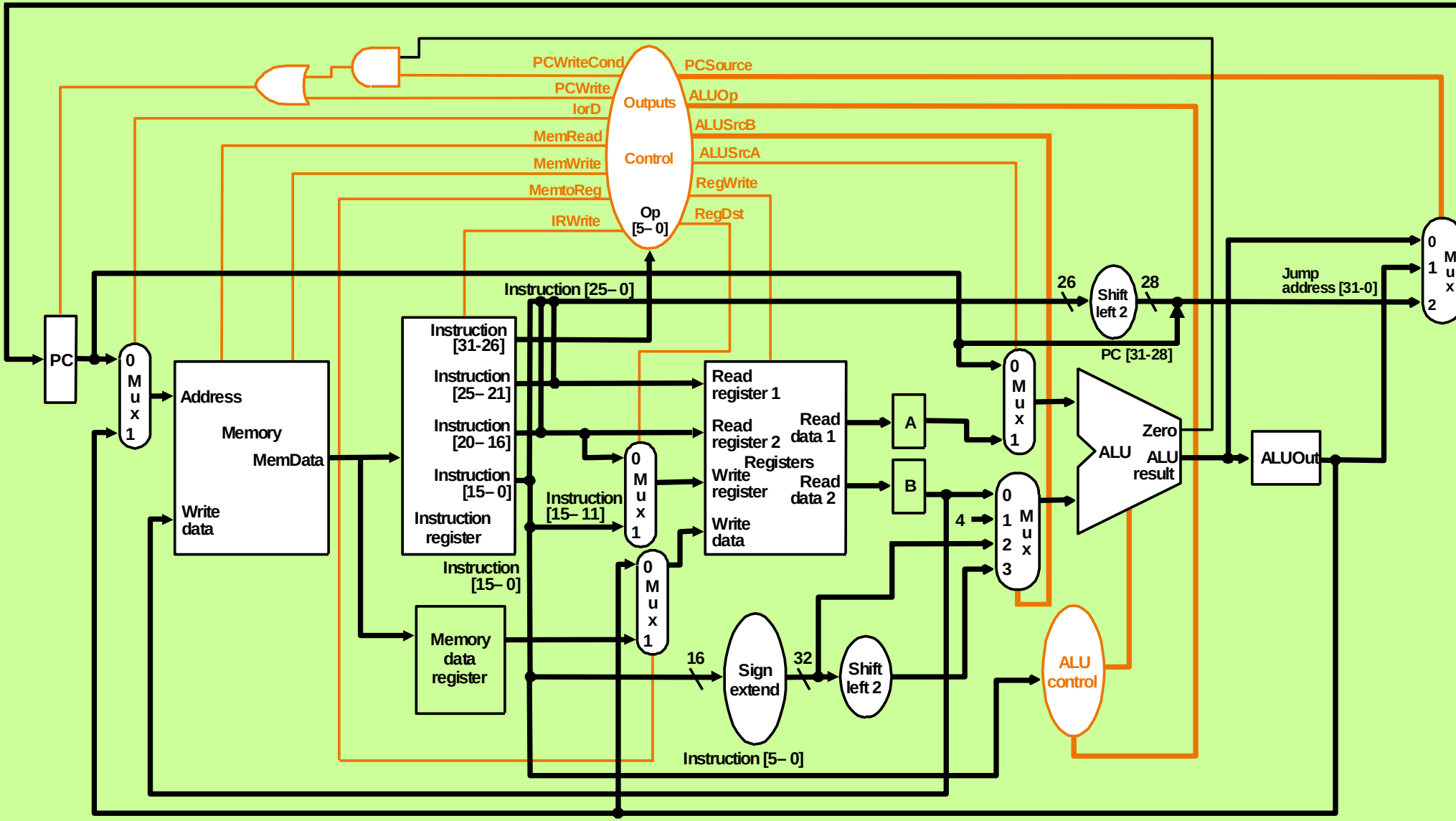
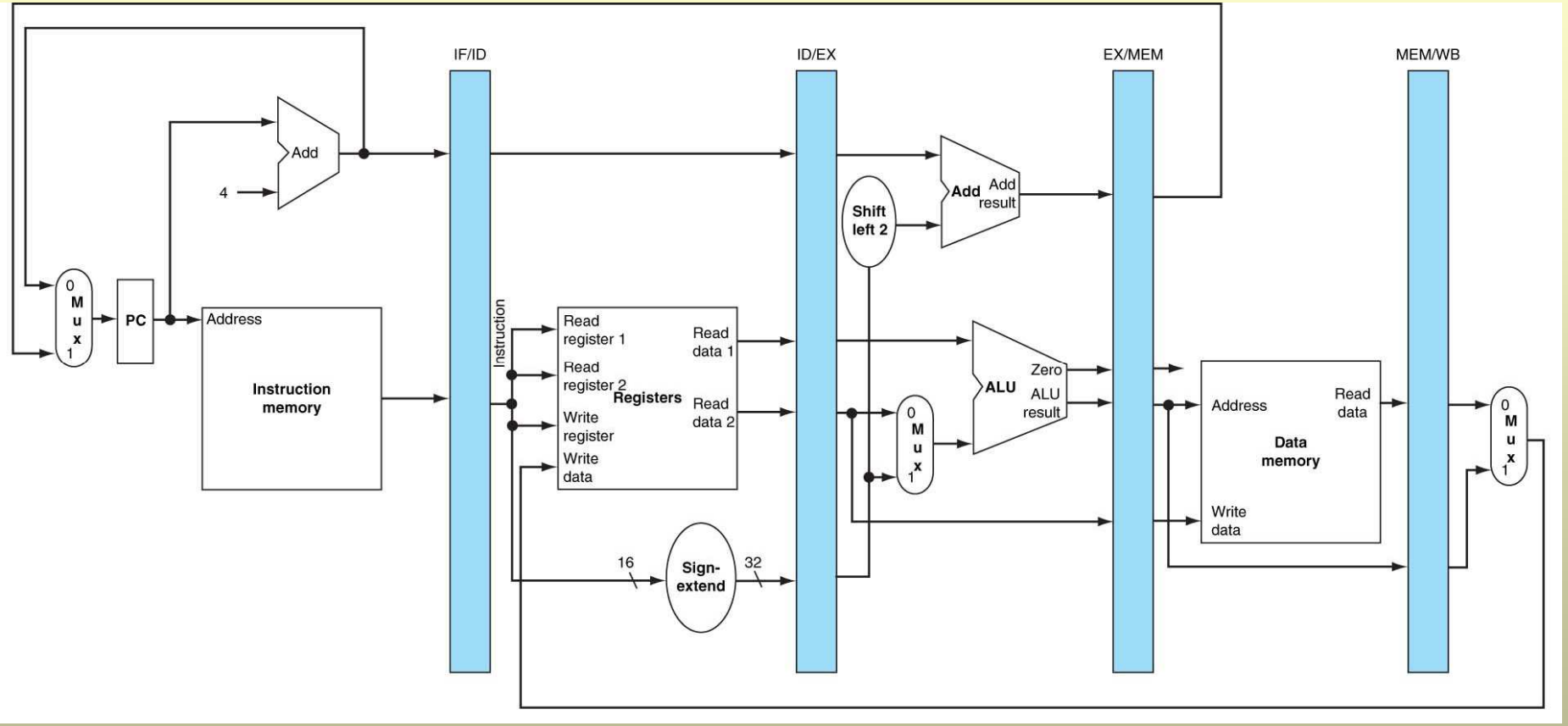
CSE 2021
Computer Organization

Hugh Chesser, CSEB
1012U

Multicycle Implementation: Control Units added



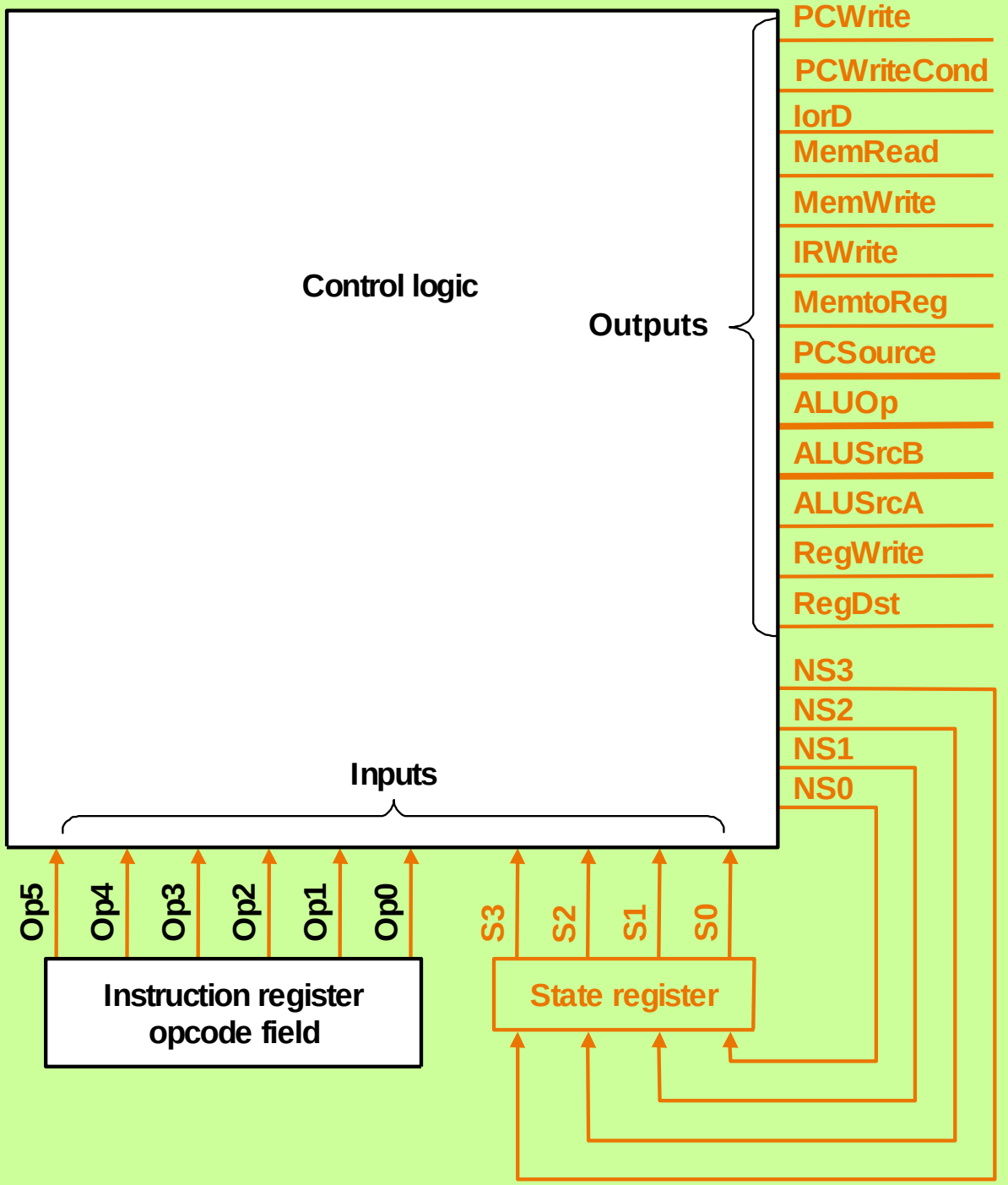
Multicycle Implementation – 5 Steps

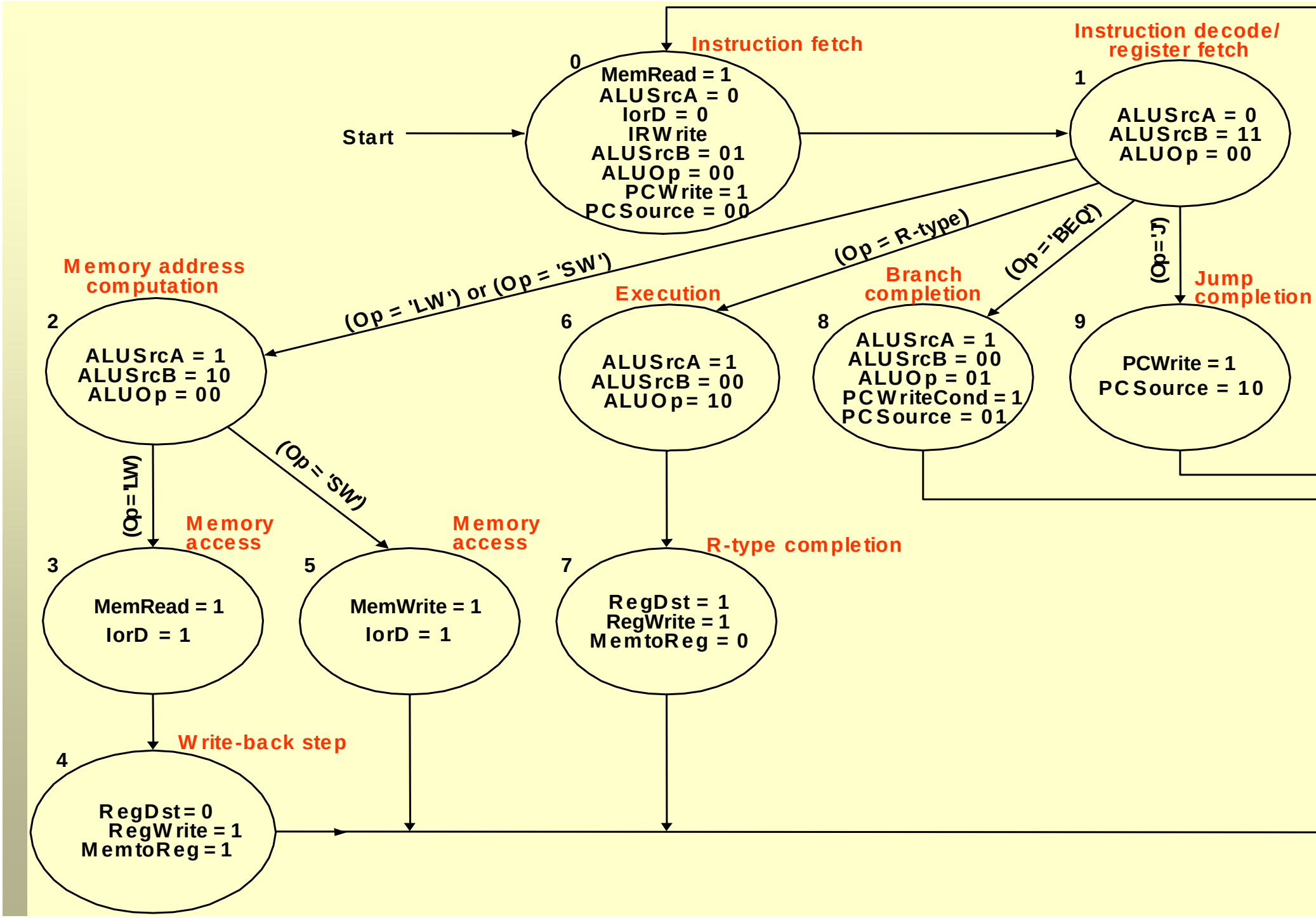


Multicycle implementation – "unwound" to show datapath in each step

Finite State Machine

Control of Multicycle Datapath (5)




Start
0
Instruction fetch
MemRead = 1
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite = 1
PCSource = 00
1
Instruction decode/
register fetch
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
2
Memory address
computation
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
Execution
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
8
Branch
completion
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond = 1
PCSource = 01
9
Jump
completion
PCWrite = 1
PCSource = 10
(Op = 'LW')
(Op = 'SW')
3
Memory
access
MemRead = 1
IorD = 1
5
Memory
access
MemWrite = 1
IorD = 1
7
R-type completion
RegDst = 1
RegWrite = 1
MemtoReg = 0
4
Write-back step
RegDst = 0
RegWrite = 1
MemtoReg = 1

Control Logic – Truth Table

Note that control outputs depend only on current state (Op column is blank for all output rows)

Next state depends on current state and inputs (opcode from instruction)

Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

NS0 Example (1)

Give the logic equation for the NS0 bit for the FSM

- NS0 is true for states 1(0001_{two}), 3 (0011_{two}), 5 (0101_{two}), 7 (0111_{two}), 9 (1001_{two})
- Referring to the truth table…

$$NS0_1 = state0 = \overline{S0} \cdot \overline{S1} \cdot \overline{S2} \cdot \overline{S3}$$

Output	Current states	Op
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

Take a couple of minutes to think about the other conditions…

NS0 Example (2)

Give the logic equation for the NS0 bit for the FSM

Output	Current states	Op
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw'
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

$$NS0_3 = state2 \cdot Op = \text{'lw'}(23_{hex}) = \overline{S0} \cdot \overline{S1} \cdot S2 \cdot \overline{S3} \cdot Op5 \cdot \overline{Op4} \cdot \overline{Op3} \cdot \overline{Op2} \cdot Op1 \cdot Op0$$

$$NextState5 = State2 \cdot (Op[5\text{-}0]=sw)$$
$$= \overline{S3} \cdot \overline{S2} \cdot \overline{\overline{S1}} \cdot \overline{S0} \cdot Op5 \cdot \overline{Op4} \cdot Op3 \cdot \overline{Op2} \cdot Op1 \cdot Op0$$

$$NextState7 = State6 = \overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0}$$

$$NextState9 = State1 \cdot (Op[5\text{-}0]=jmp)$$
$$= \overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot S0 \cdot \overline{Op5} \cdot \overline{Op4} \cdot \overline{Op3} \cdot \overline{Op2} \cdot Op1 \cdot \overline{Op0}$$

$$NS0 = NS1 + NS3 + NS5 + NS7 + NS$$

Agenda

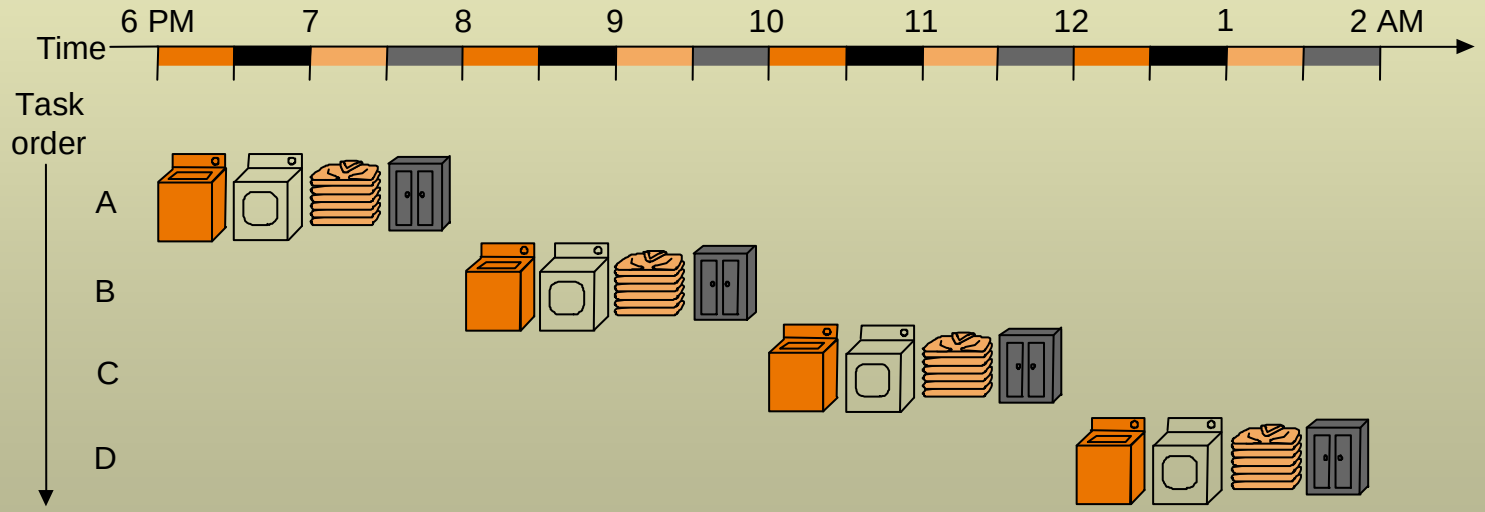
Topics:

1. Pipeline implementation

Patterson: 4.5

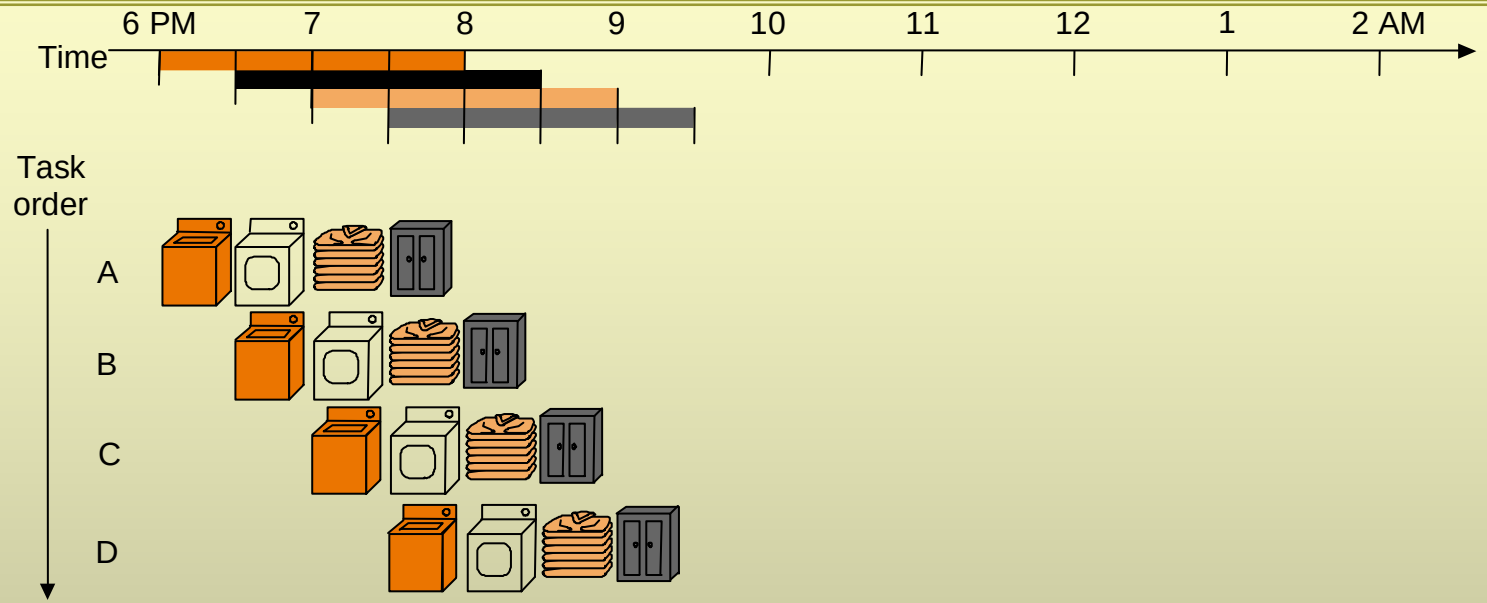
Why Pipelining (1)?

- Pipelining is an implementation technique in which multiple instructions are overlapped during execution.
- Pipelining enhances the throughput of the processors.
- To explain pipelining, consider an analogy with a laundry example where 4 users are asked to wash, dry, fold, and store several loads of clothes.



Sequential Laundry

Why Pipelining (2)?



Pipelined Laundry (assume equal time at each stage with 4 users: Ann, Brian, Cathy, and Don).

A. Ann places dirty load # 1 in the washer

B. When washer is finished, Brian places wet load # 1 in the dryer. Ann loads the washer with load # 2 of dirty clothes.

C. When load # 1 is dried, Cathy takes dried load # 1 out of the dryer and starts folding. Brian loads the dryer with wet load # 2. Ann loads the washer with load # 3 of dirty clothes.

D. When load # 1 is folded, Don starts storing folded load # 1 in the storer. Cathy takes dried load # 2 out of the dryer and starts folding. Brian loads the dryer with wet load # 3. Ann loads the washer with load # 4 of dirty clothes. Process continues.

Pipelining with Single Cycle Datapath (1)

MIPS pipelining has the following five stages:

1. Instruction Fetch (IF): Fetch instruction from memory.
2. Instruction Decode (ID): Read registers while decoding the instruction.
3. Execution (EX): Execute the operation or calculate the address.
4. Memory Access (MEM): Access an operand in data memory.
5. Write Back Stage (WB): Write the result into a register.

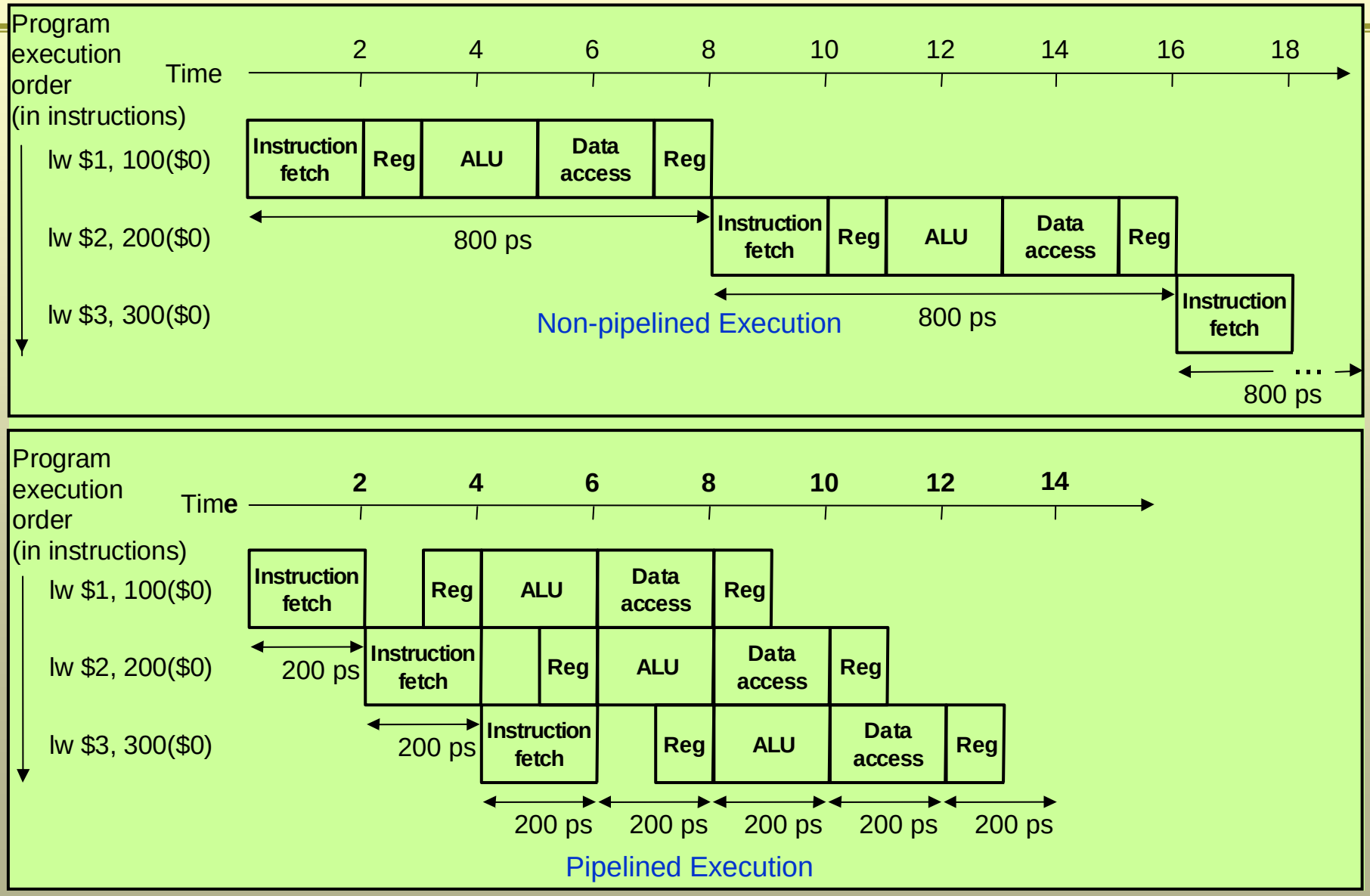
Activity: Compare the non-pipelined execution time with the pipelined execution time for the instructions:

```
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
```

assuming the following delays at different functional units.

Instruction Class	Instruction fetch	Register read	ALU	Data access	Register write	Total
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ns
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ns
R-format	200 ps	100 ps	200 ps		100 ps	600 ns
Branch (beq)	200 ps	100 ps	200 ps			500 ns

Pipelining with Single Cycle Datapath (2)



Pipelining with Single Cycle Datapath (3)

- Speedup obtained through pipelining equals the number of pipe stages if execution time of each stage is the same.
- In our previous example, speedup should be 5.

Actual speedup in our previous example = 24 / 14 = 1.71

Why? Number of instructions are too small.

Increase the number of instructions to 1003.

Then speedup = (1003 × 8)/(2 × 1003 + 8) = 8024/2014 = 3.98

- Pipelining added some overhead (additional 100ps for Register read)
- Note that pipelining increases the overall throughput. The execution time for each instruction stays the same.

Suitability of MIPS architecture towards Pipelining

MIPS	80x86
1. All MIPS instructions are of the same length. Instruction fetch (IF) in the first pipeline stage and decoding in the second stage is easier.	1. Instructions in 80x86 have variable length from 1 byte to 17 bytes. This makes the first two stages (instruction fetch and decoding) more challenging making pipelining difficult.
2. MIPS instructions have a limited number of formats with registers staying specified at almost the same bit positions. This allows the decoding stage to start reading the registers at the same time as HW is determining the type of instruction.	2. Due to variable instruction length in 80x86, the registers are specified at different bit positions.
3. MIPS do not allow operands to be directly used from the memory. Operands are first loaded into the registers.	3. 80x86 allows direct operation on operands while in memory. An additional address stage is therefore needed in 80x86.
4. Since operands are aligned in memory, data can be transferred from memory to registers in a single data transfer command.	

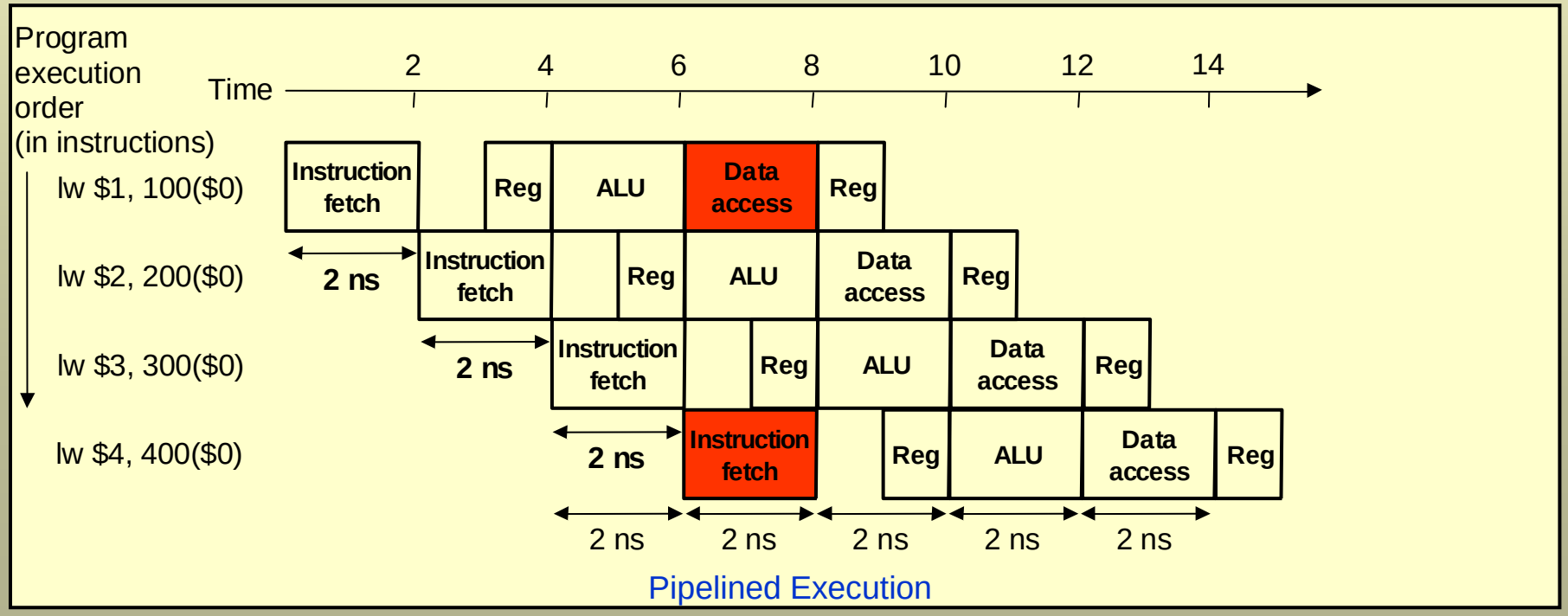
Pipelining Hazards (1)

— Hazards: Pipelining hazards occur when the next instruction in a pipelined program can not be executed until the prior instruction has been executed.

1. Structural Hazards: occur when hardware does not support combination of instructions to be executed in the same clock cycle.

   Laundry analogy: A washer-dryer combo is used where a load of clothes is washed and then dried in the same machine.

   MIPS: A single memory used for data and instructions results in structural hazard below.



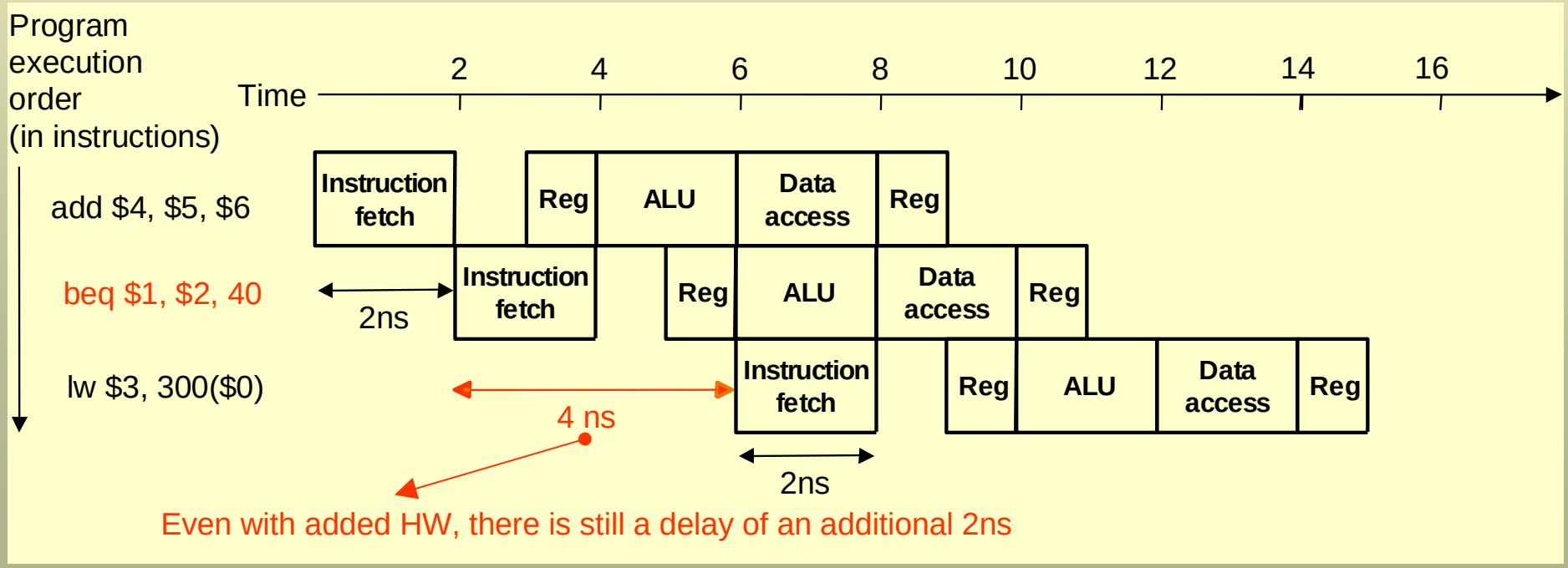
Pipelining Hazards (2)

2. Control Hazards: occur when the execution of the next instruction depends upon a decision in the previous instruction.

   Laundry analogy: Laundry crew is required to determine the correct detergent and temperature setting for perfect cleaning.

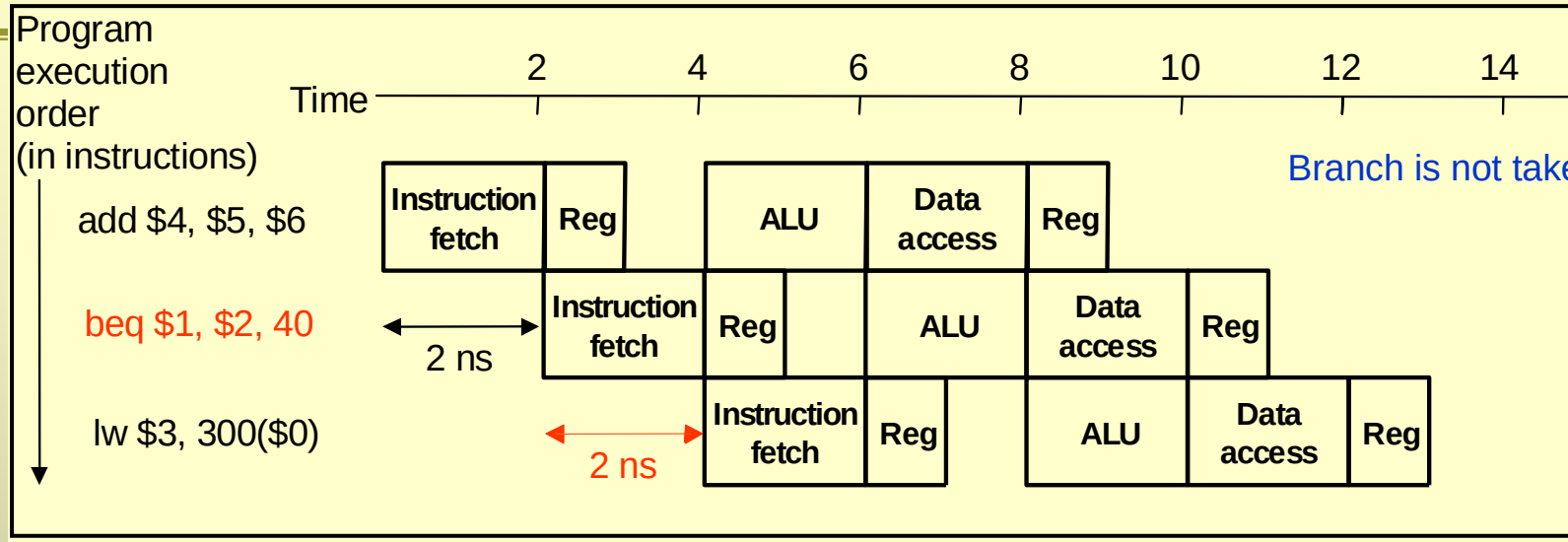
   MIPS: Branch instruction can cause a control hazard.

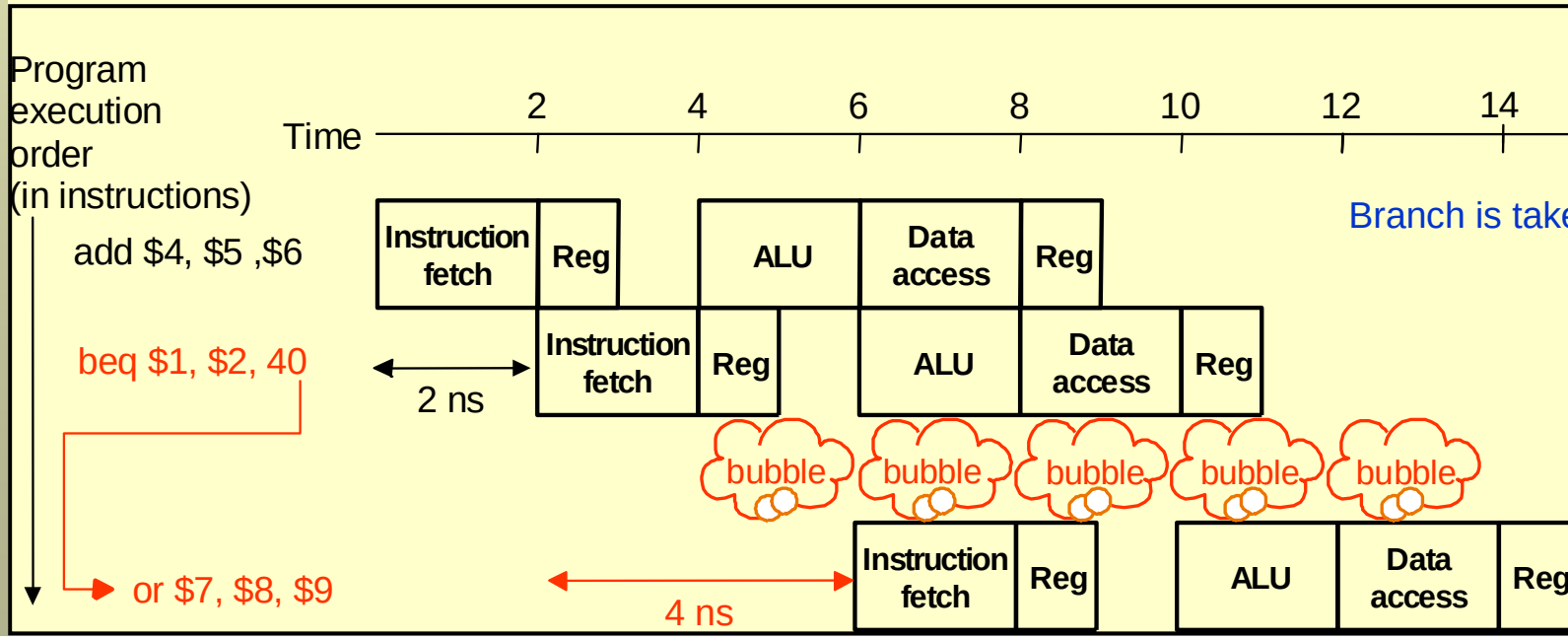
   Modification: Add extra HW such that: (1) registers are loaded and tested for equality and (2) PC is updated with the branch target address in the second step.



Pipelining Hazards (3)

Solution # 1 to Control Hazards: Always predict that the branch will fail and keep executing the program

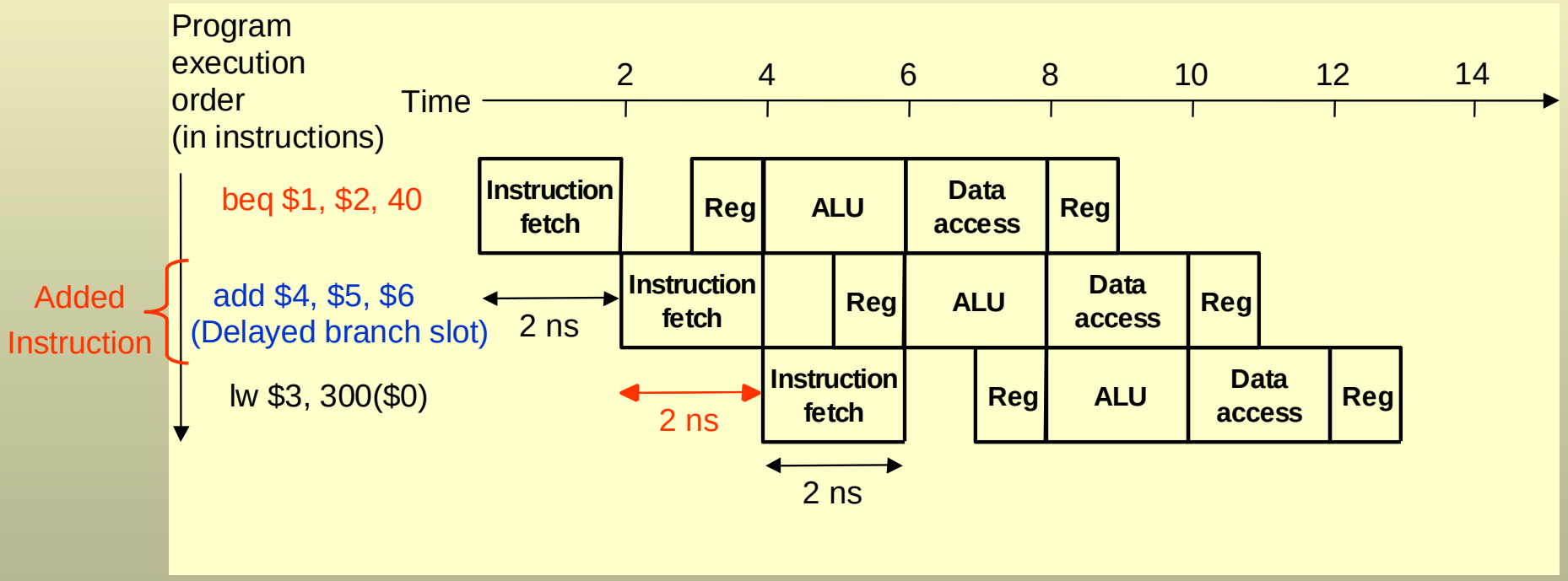




Registers must be cleared and an additional delay of 2ns

Pipelining Hazards (4)

Solution # 2 to Control Hazards: Insert an additional instruction that is not affected by the branch instruction. This solution is called delayed branch.

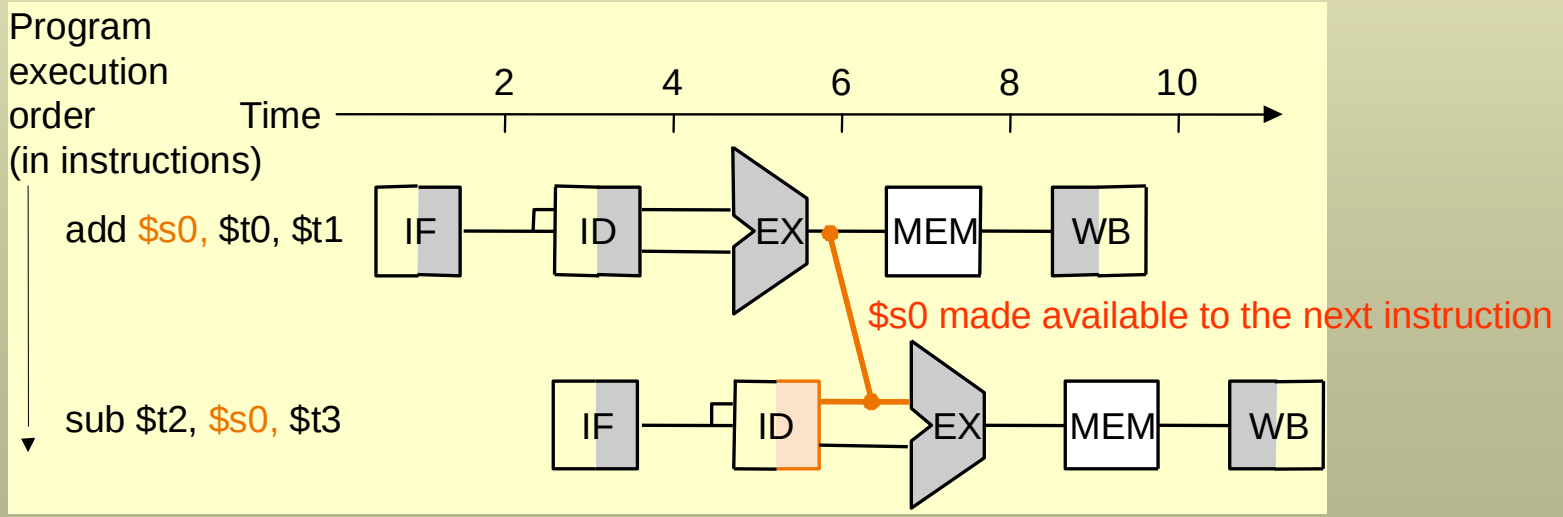


Pipelining Hazards (5)

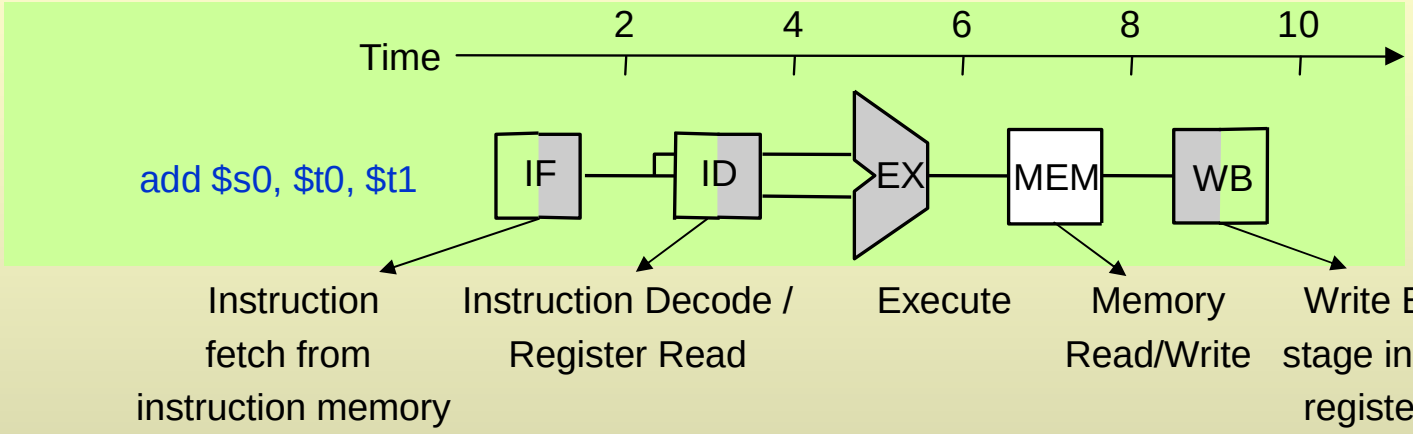
3. Data Hazards: occurs when an operand used in the next instruction is updated in the prior instruction.

```
add   $s0, $s1, $s2
sub   $t0, $s0, $t1
```

Solution: As soon as ALU generates data ($s0), it makes it available to the next instruction before storing it in the register file.



Graphical Representation



1. Shading in each block indicates the element is used for in the instruction. Since memory is not accessed in an add instruction, it is not shaded.
2. Shading on the left half of the block indicates that the element is being written. During instruction fetch, the instruction memory is read so the right half of IF block is shaded.
3. Shading on the right half of the block indicates that the element is being read. During write back stage, the register file is written so the left half of the WB block is shaded.

Activity 2

Using the graphical representation, show that the following swap procedure has a pipeline hazard. Reorder the instructions to avoid pipeline stalls.

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```