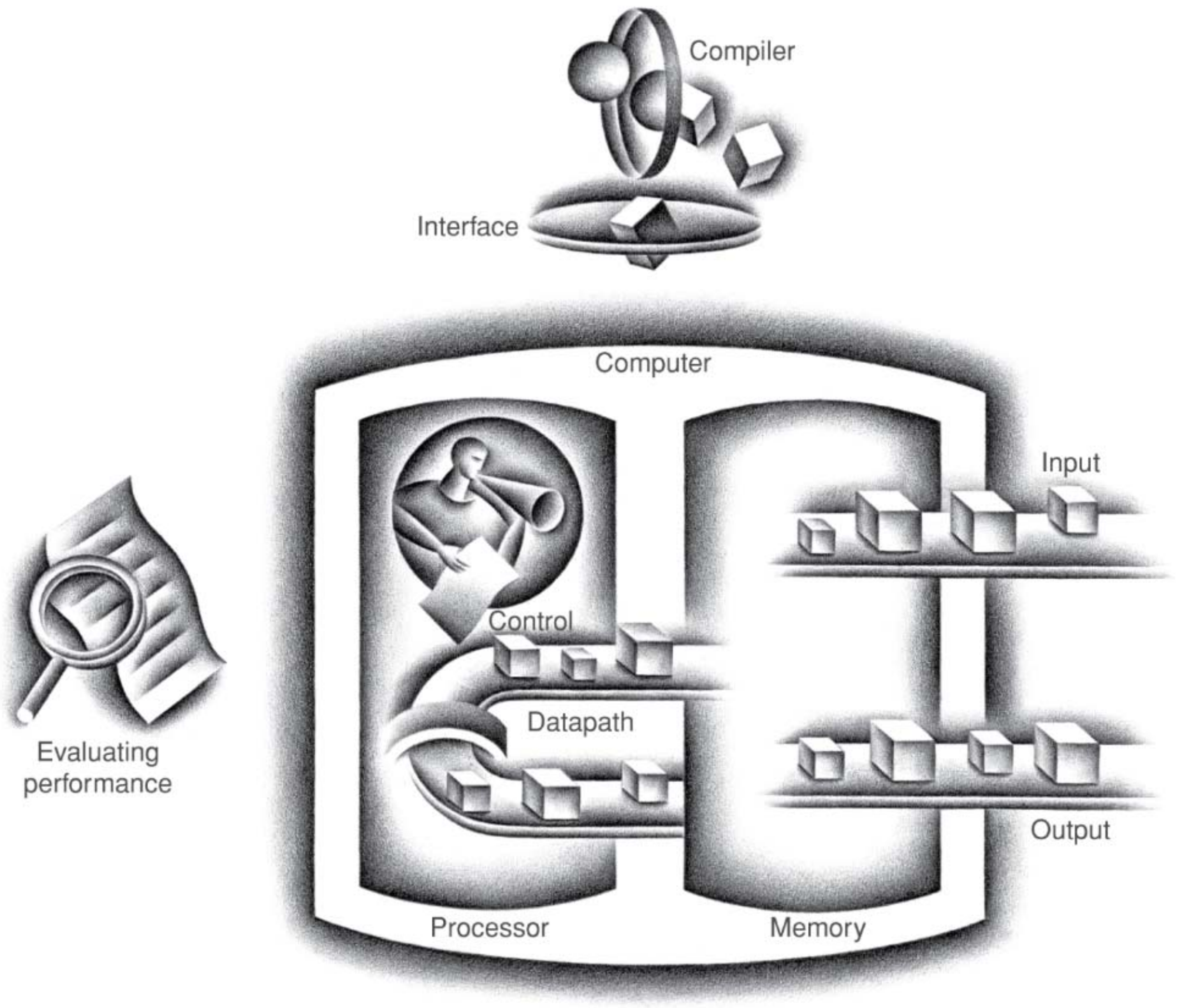
Chapter 4

The Processor

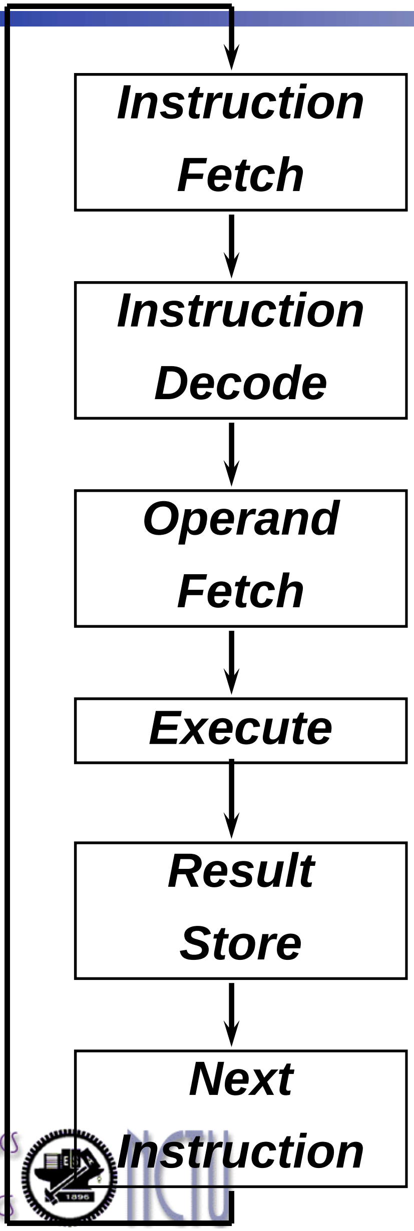
The Processor ?



Introduction

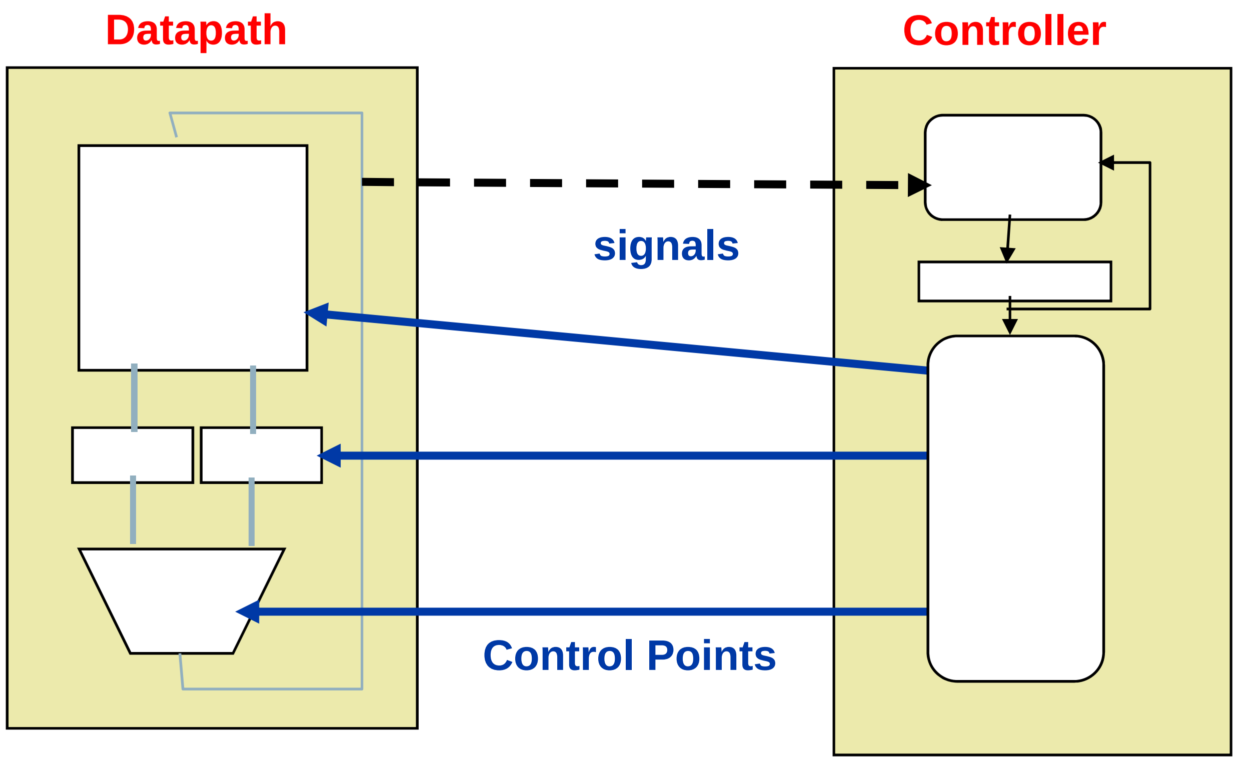
- We will learn
  - How the ISA determines many aspects of the implementation
  - How the choice of various implementation strategies affects the clock rate and CPI for the computer
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical operation: `add`, `sub`, `and`, `or`, `slt`
  - Program flow control: `beq`, `j`

Instruction Cycle



- For every instruction, the first three phases are identical:
  - Instruction fetch: send PC to the memory and fetch the instruction from the memory
  - Instruction decode and operand fetch: read one or two registers, using fields of the instruction to select the register from the register file (RF)
- Use ALU, depending on instruction class, to calculate
  - Arithmetic result
  - Memory address for load/store
  - Branch target address
- Access data memory only for load/store
- Write the ALU or memory back into a register,
  - using fields of the instruction to select the register
- PC $\leftarrow$ target address or PC + 4

Datapath vs Control



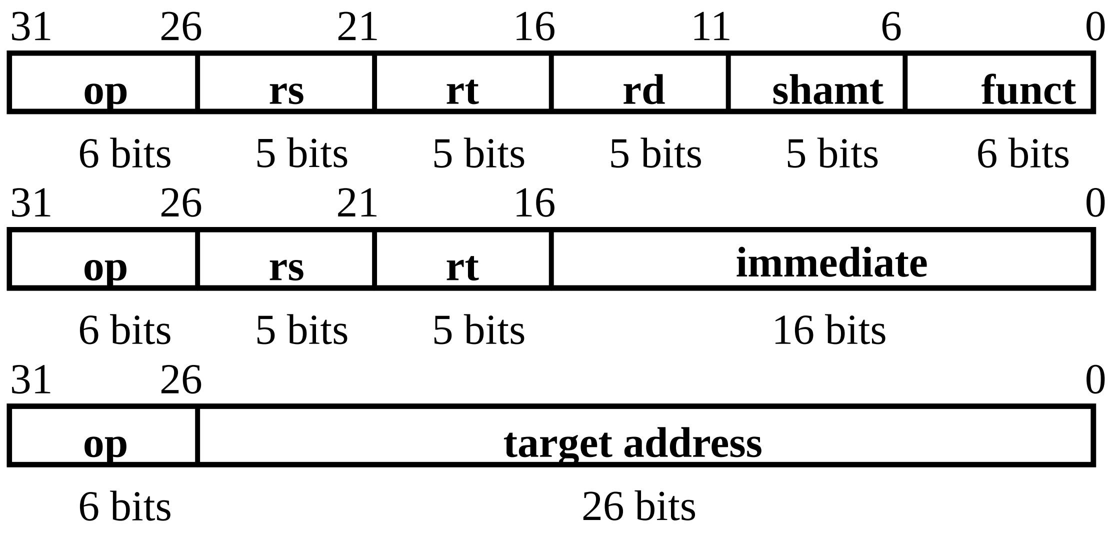
- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- Controller: State machine to orchestrate/control operation on the data path
  - Based on desired function and signals

Five Steps to Implement a Processor

1. Analyze the instruction set (datapath requirements)
   - The meaning of each instruction is given by the *register transfers*
   - Datapath must include storage element
   - Datapath must support each register transfer
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath meeting the requirements
4. Analyze the implementation of each instruction to determine setting of control points effecting register transfer
5. Assemble the control logic

Step 1: Analyze the Instruction Set

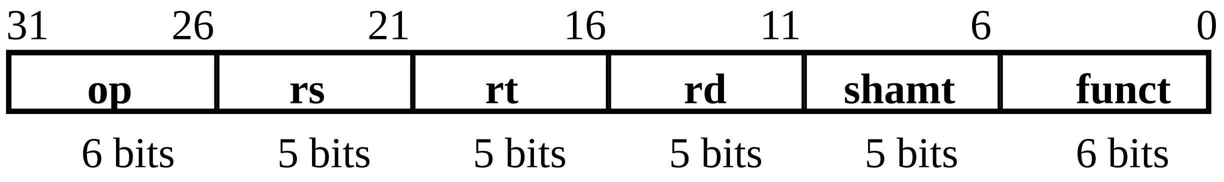
- All MIPS instructions are 32 bits long with 3 formats:
  - R-type:
  - I-type:
  - J-type:

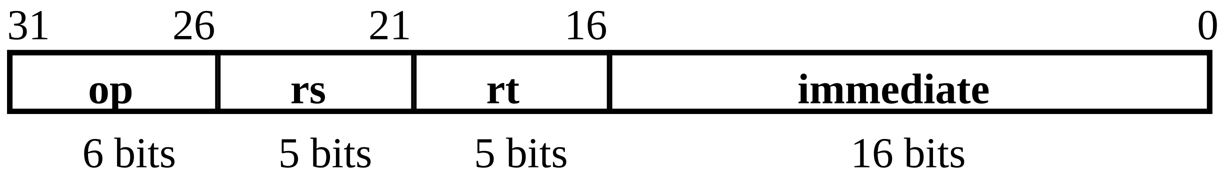


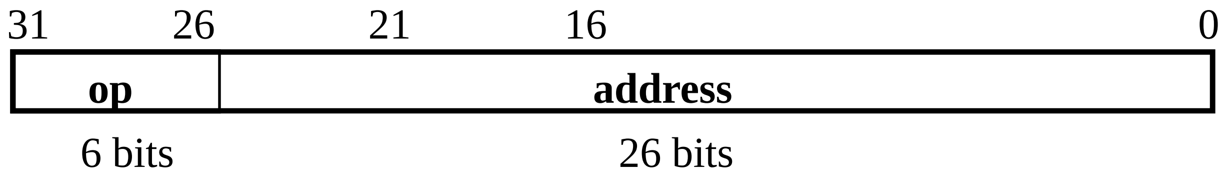
- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: source and/or destination register
  - shamt: shift amount
  - funct: selects variant of the “op” field
  - address / immediate
  - target address: target address of jump

Step 1: Analyze the Instruction Set

- Arithmetic/logical operation:
  - `add rd, rs, rt`
  - `sub rd, rs, rt`
  - `and rd, rs, rt`
  - `or rd, rs, rt`
  - `slt rd, rs, rt`
- Load/Store:
  - `lw rt,rs,imm16`
  - `sw rt,rs,imm16`
- Imm operand:
  - `addi rt,rs,imm16`
- Branch:
  - `beq rs,rt,imm16`
- Jump:
  - `j target`







Logical Register-Transfer Level (RTL)

- RTL is a design abstraction, which gives the hardware description of the instructions

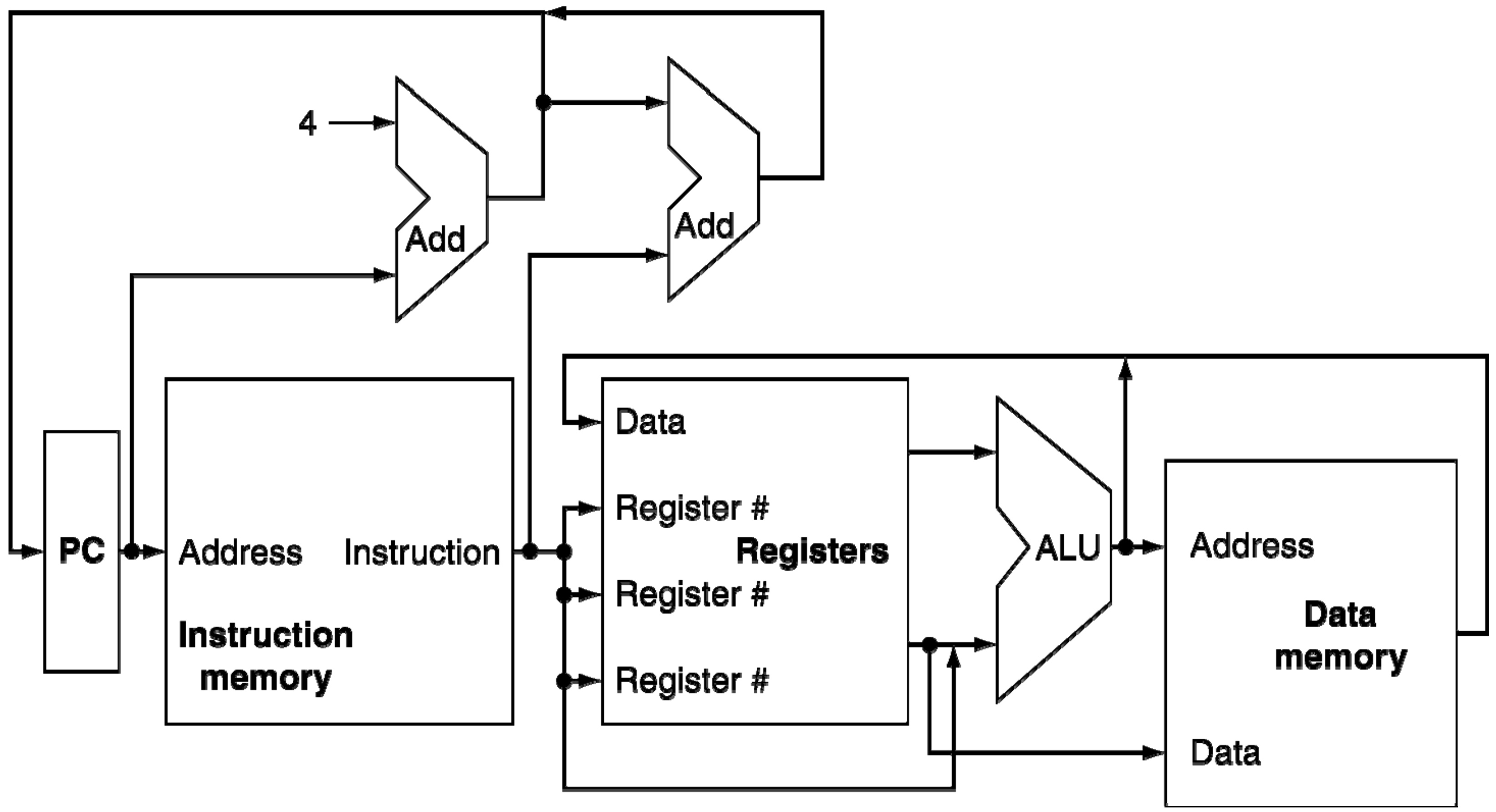
MEM[PC] = op | rs | rt | rd | shamt | funct
or = op | rs | rt | Imm16
or = op | Imm26 (added at the end)

Inst	Register transfers
ADD	R[rd] <- R[rs] + R[rt]; PC <- PC + 4
SUB	R[rd] <- R[rs] - R[rt]; PC <- PC + 4
LOAD	R[rt] <- MEM[R[rs] + sign_ext(Imm16)]; PC <- PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <-R[rt]; PC <- PC + 4
ADDI	R[rt] <- R[rs] + sign_ext(Imm16)]; PC <- PC + 4

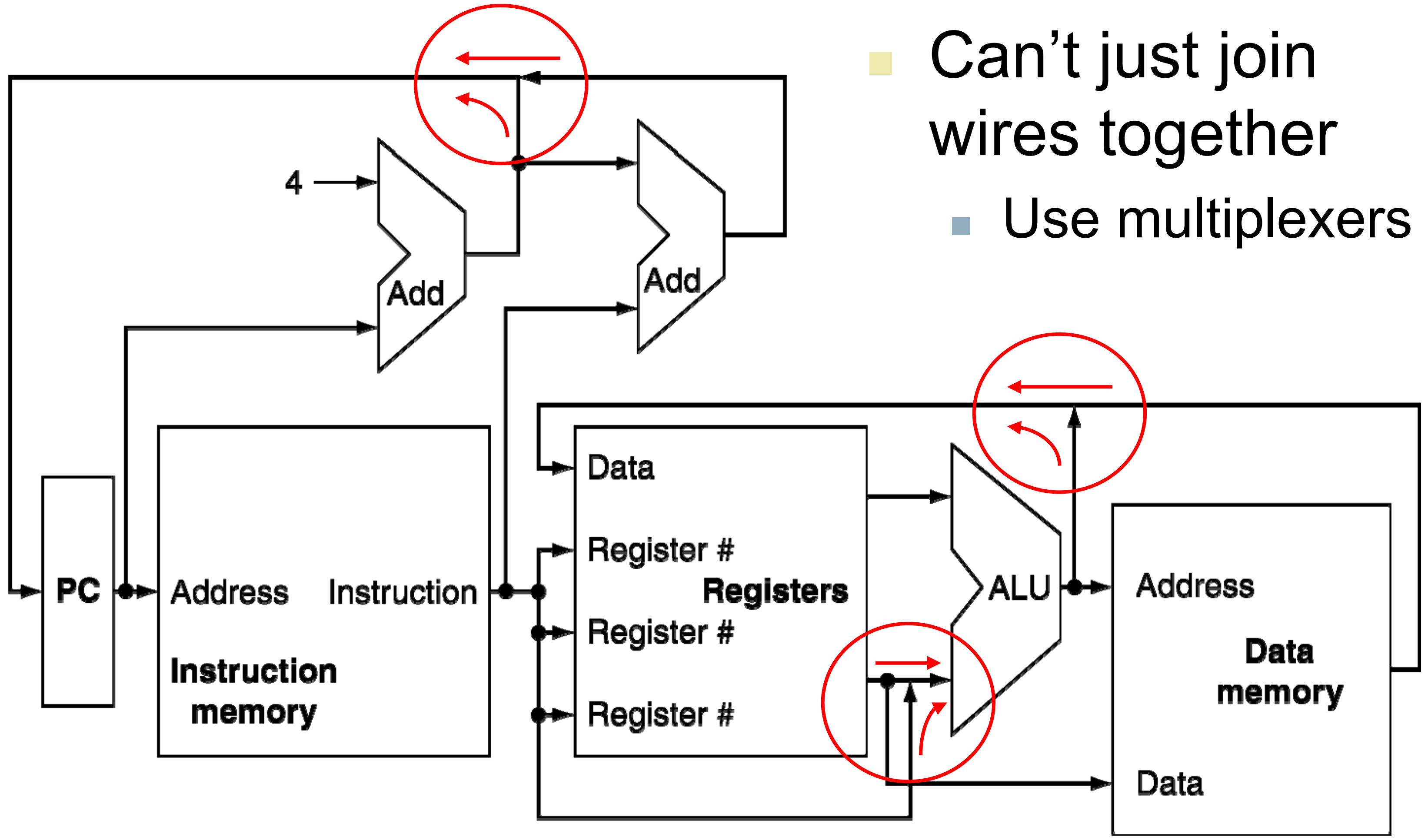
BEQ	if (R[rs] == R[rt]) then PC <- PC + 4 + sign_ext(Imm16)] || 00
	else PC <- PC + 4

J	PC <- PC[31..28] || Imm 26 || 00

Fig. 4.1 MIPS Datapath (Simplified)

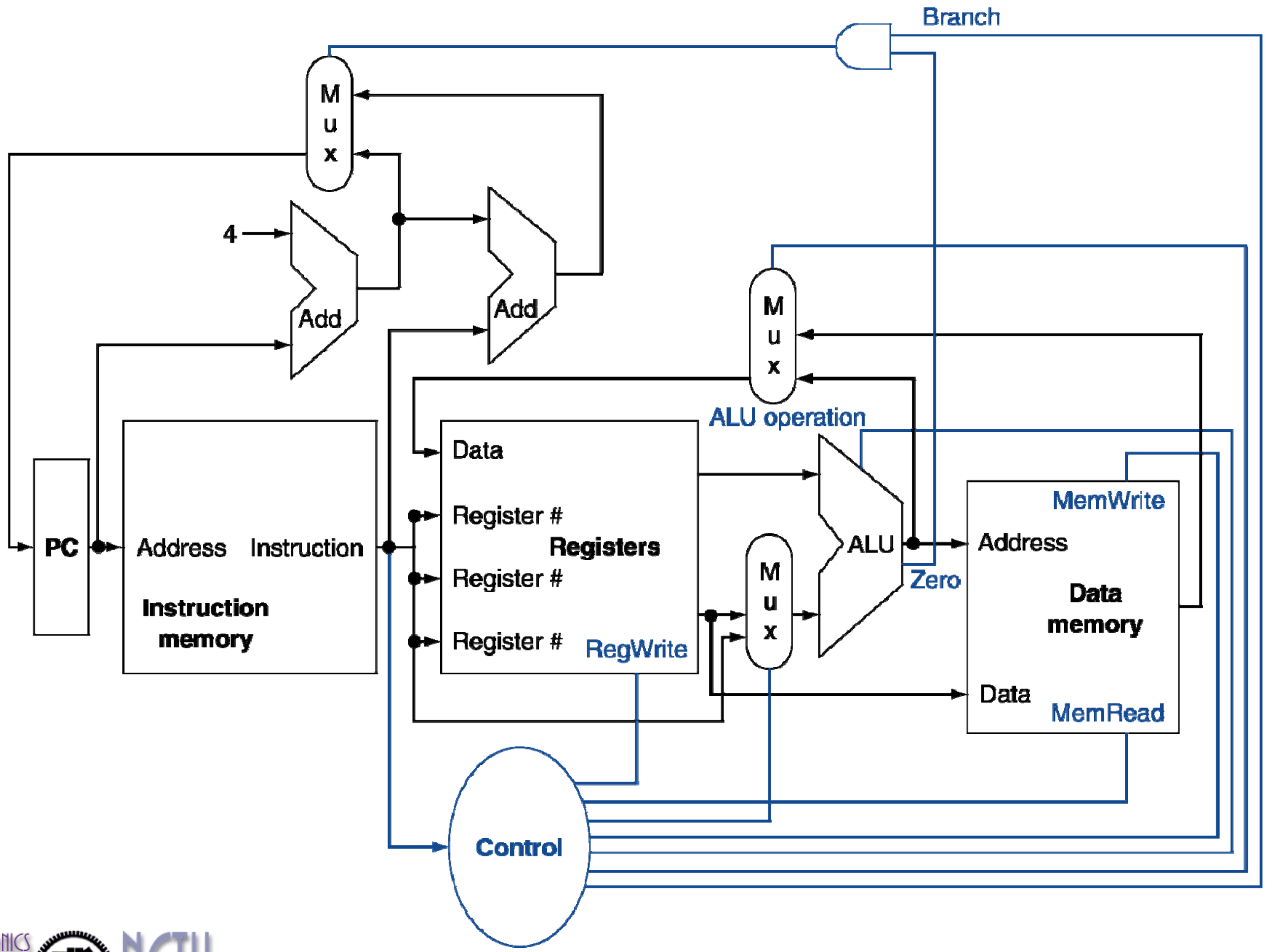


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control

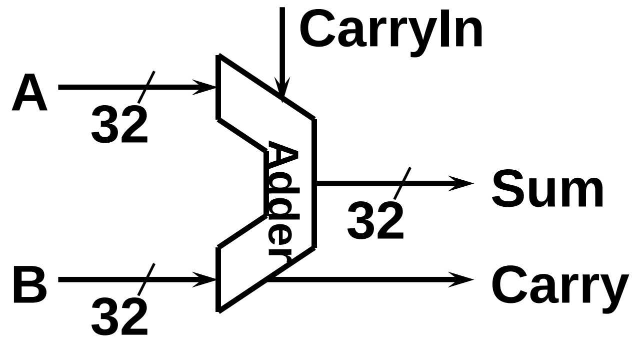

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Step 2: Datapath Elements

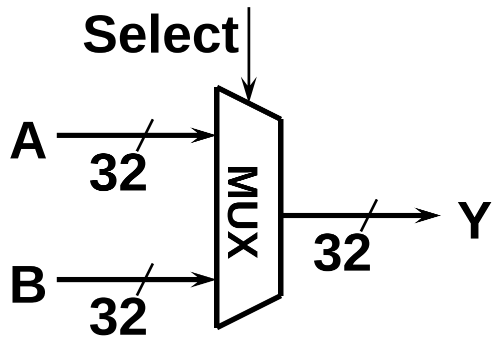
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit; Multi-bit data encoded on bus
- Two different types of datapath elements
  - Combinational elements
    - For computation, the output depends only on the current inputs
    - The output is a function of the input(s)
  - State (sequential) elements
    - For storing state/information
    - The output depends on both the input(s) and the contents of the internal state

Combinational Elements

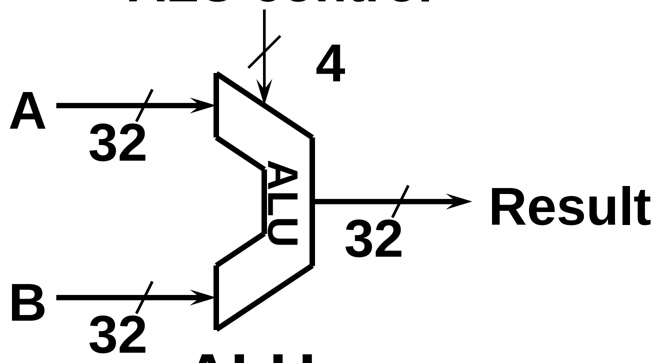
- Example of combinational logic elements :



Adder



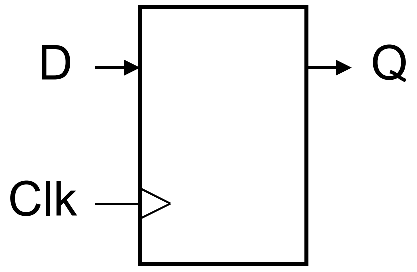
MUX

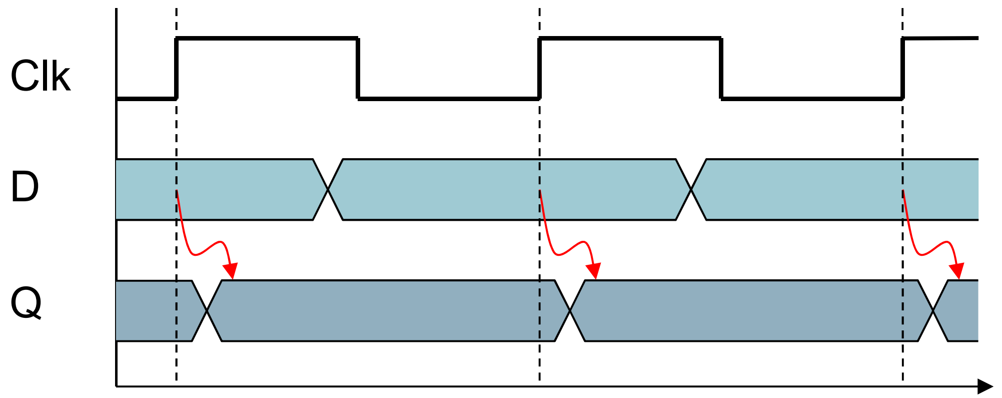


ALU

Sequential Elements (1)

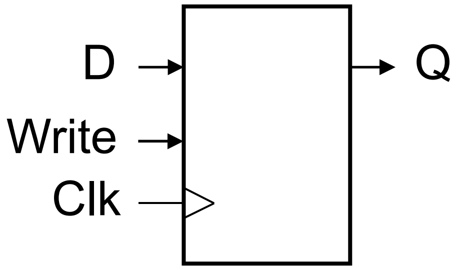
- D-type flip-flop: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

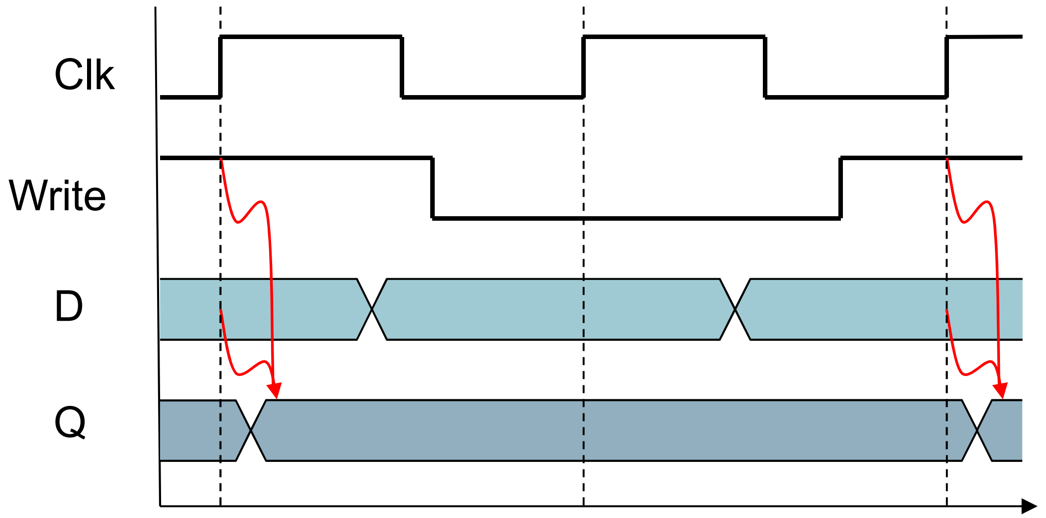




Sequential Elements (2)

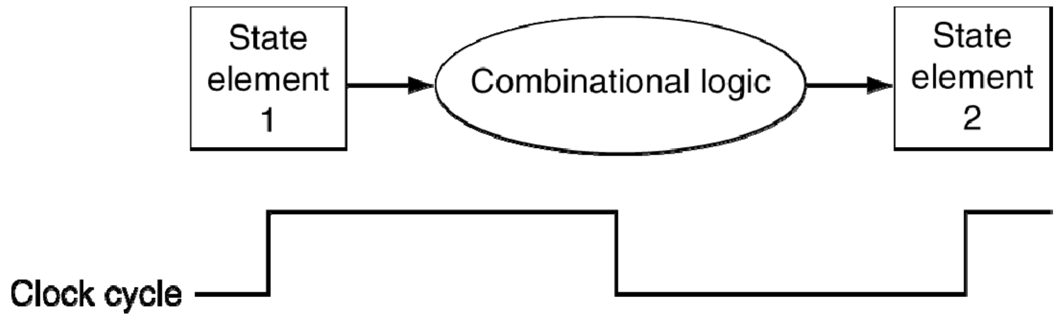
- Registers (or register file) and Memory with write control
  - Only updates on clock edge when write_enable control input is 1
  - Used when stored value is required later

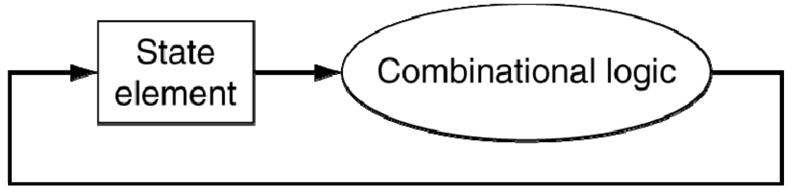




Clocking Methodology

- A clocking methodology defines when signals can be read and when they can be written
- Combinational logic transforms data during clock cycles
  - Between clock edges (edge-triggered clocking methodology)
  - Input from state elements, output to state element
  - Longest delay determines clock period





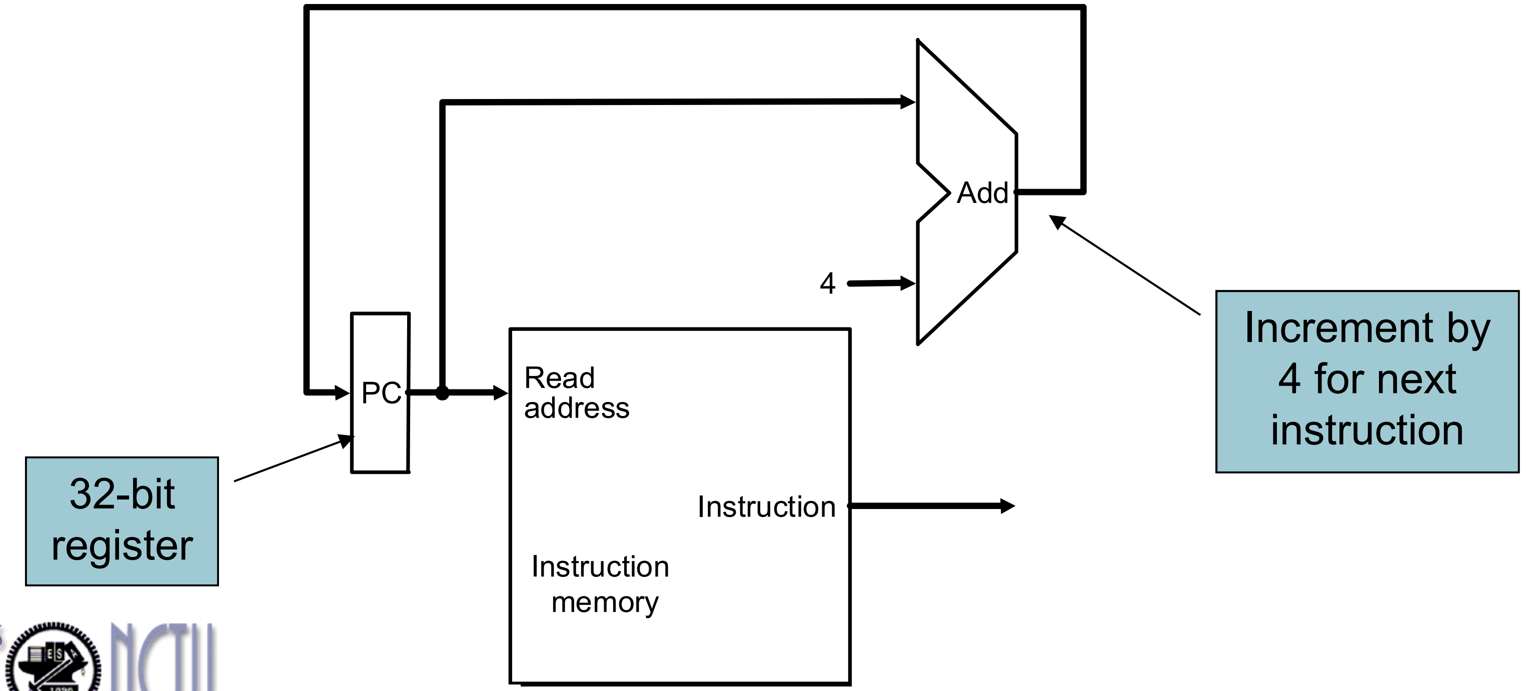
May be encountered a race problem

Step 3: Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - **Refining the overview design**

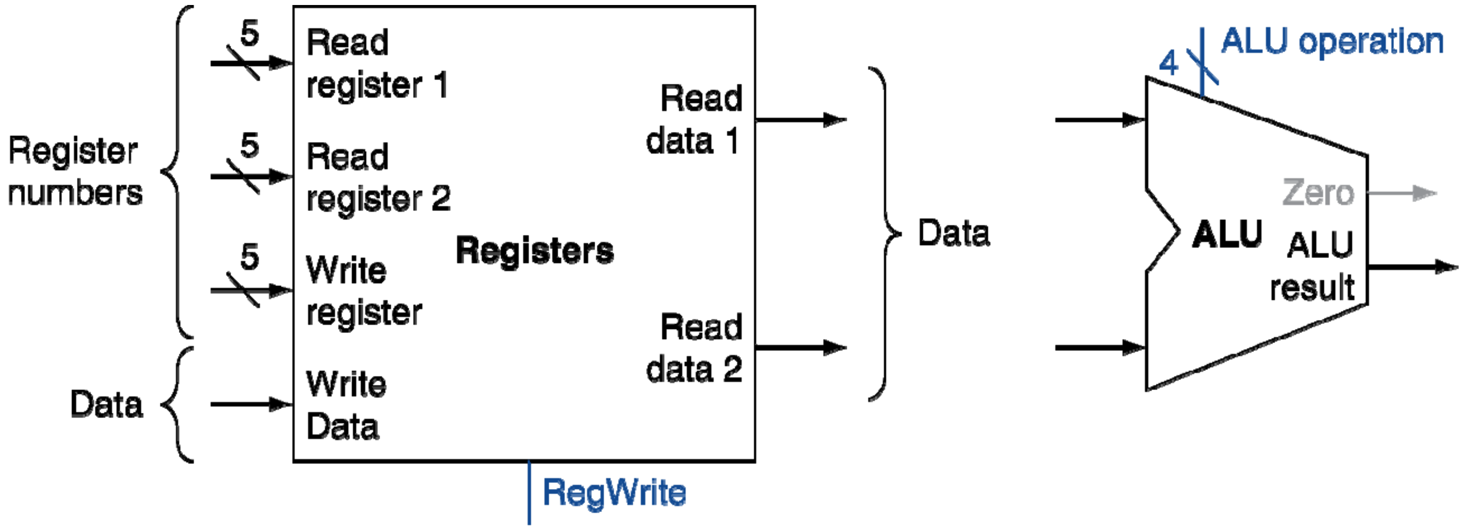
Instruction Fetch Unit

- Instruction fetch unit is used by other parts of the datapath
  - Fetch the instruction: **mem[PC]**
  - Update the program counter:
    - Sequential code: **PC <- PC + 4**
    - Branch and Jump: **PC <- "Something else"**



Step 3a: R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



a. Registers

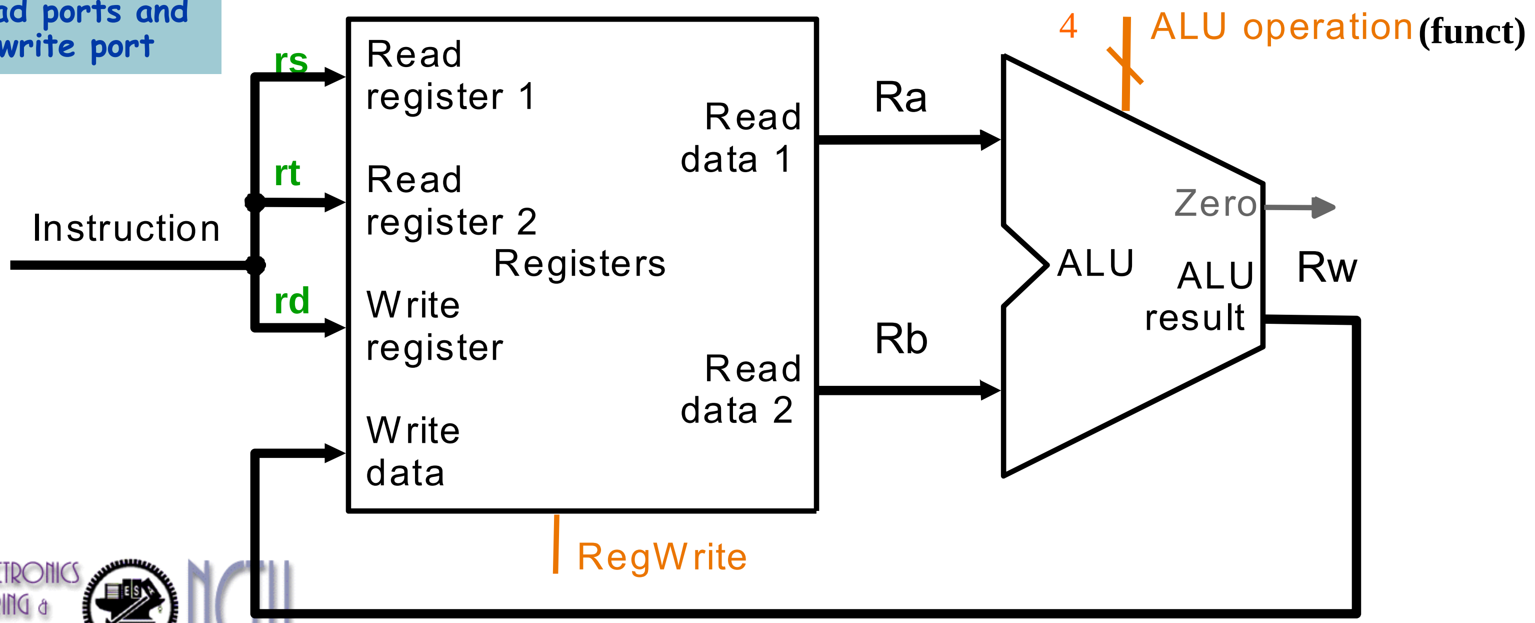
b. ALU

Add and Subtract

- **R[rd] <- R[rs] op R[rt]** Ex: **`add rd, rs, rt`**
  - Ra, Rb, Rw come from inst.'s rs, rt, and rd fields
  - ALU and RegWrite: control logic after decode

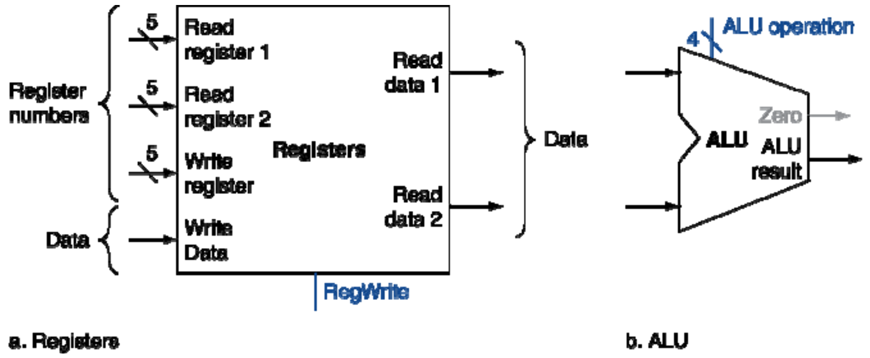
31 26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**

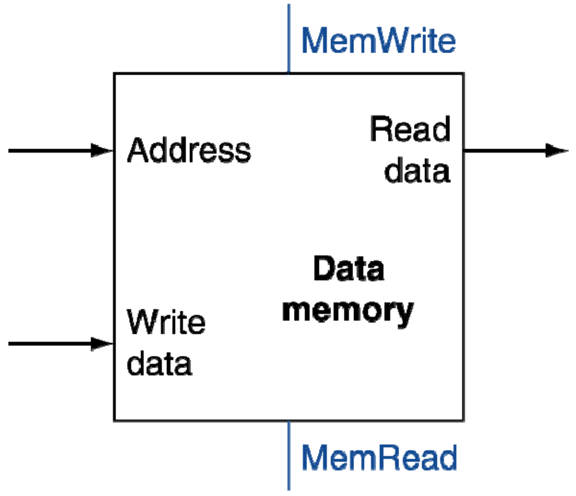
Two read ports and one write port



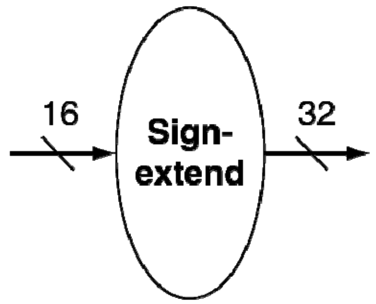
Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory





a. Data memory unit

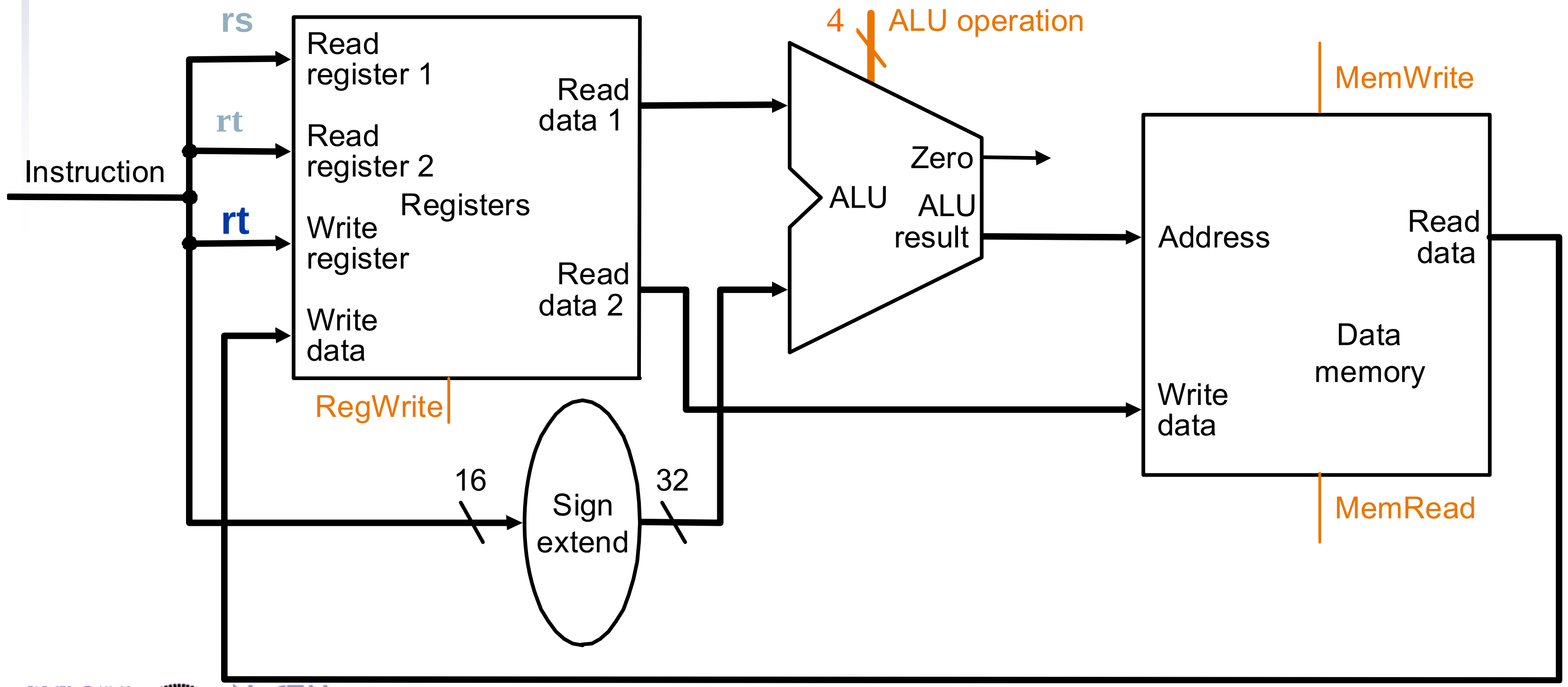


b. Sign extension unit

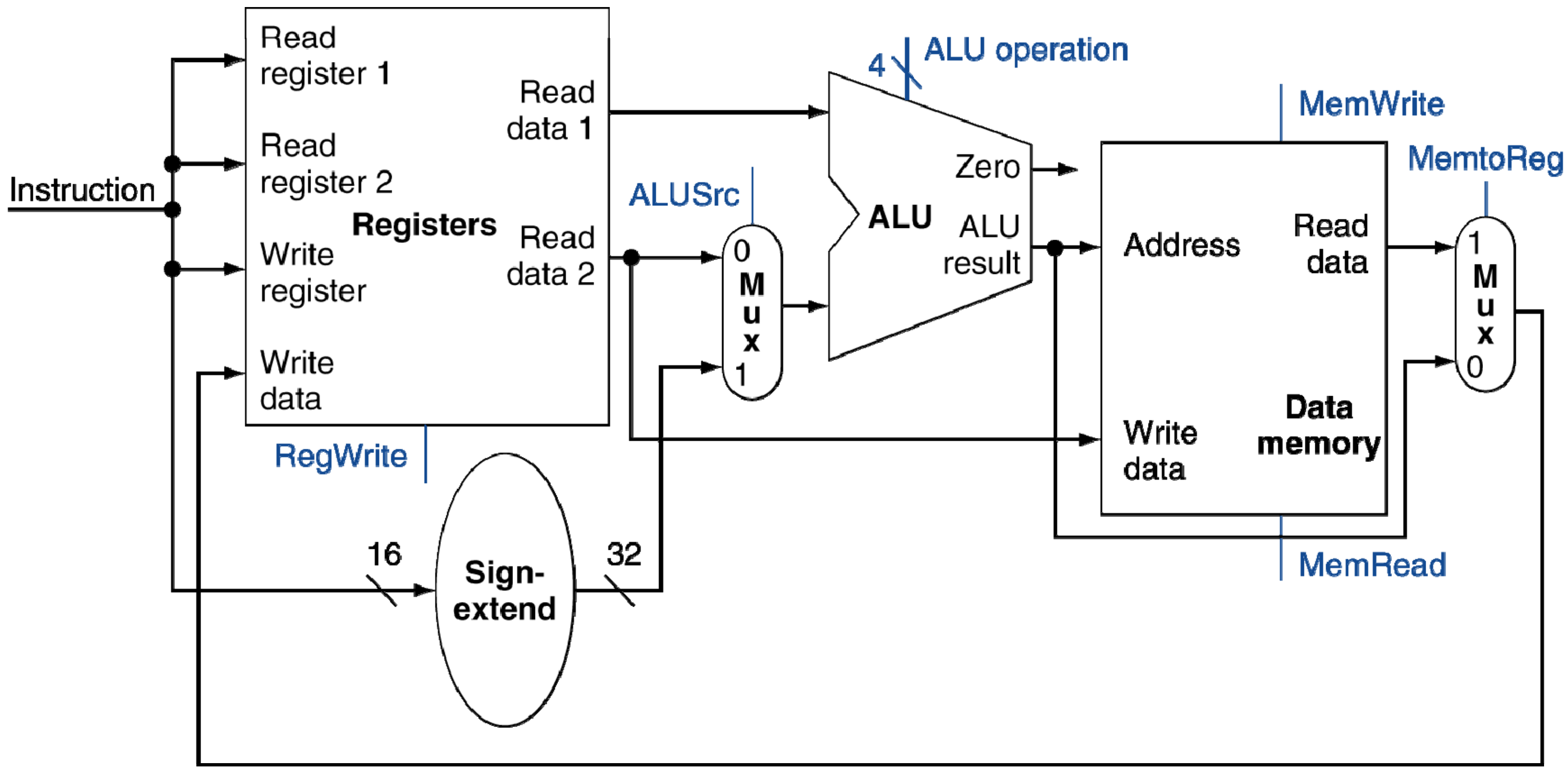
Step 3b: Store/Load Operations

- **R[rt]<-Mem[R[rs]+SignExt[imm16]]** Ex: `lw rt,rs,imm16`

31	26	21	16	0
op	rs	rt	immediate	
6 bits	5 bits	5 bits	16 bits	



R-Type/Load/Store Datapath



Recall Branch Operations

- `beq  rs, rt, imm16`

31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

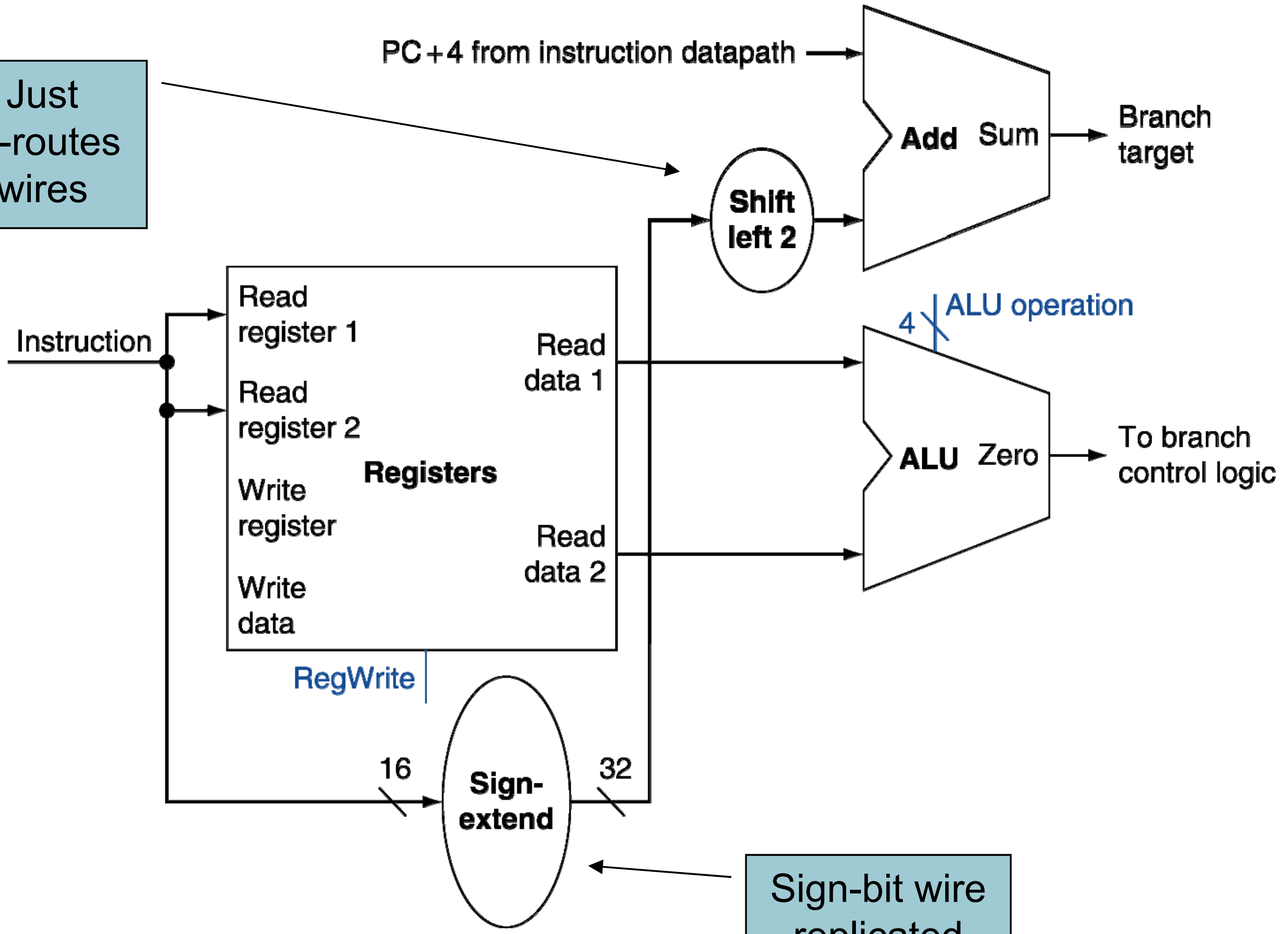
```
mem[PC]                                Fetch inst. from memory

COND <- R[rs] == R[rt]                 Calculate branch condition

if (COND == 0)                         Calculate next inst. address
    PC  <-  PC + 4 + ( SignExt(imm16) x 4 )
else
    PC  <-  PC + 4
```

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Step 3c: Branch Instructions

Just re-routes wires

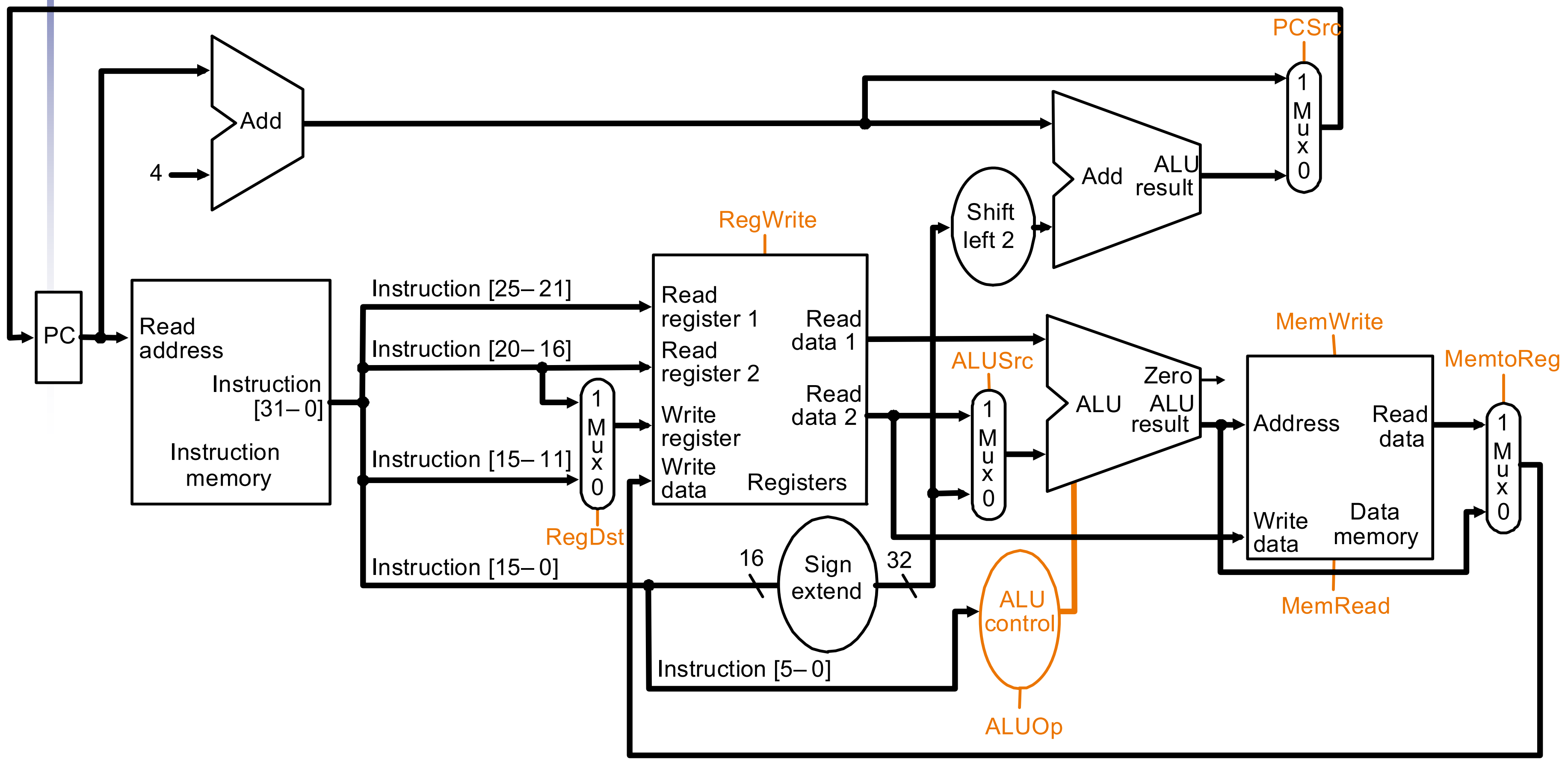


Sign-bit wire replicated

Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

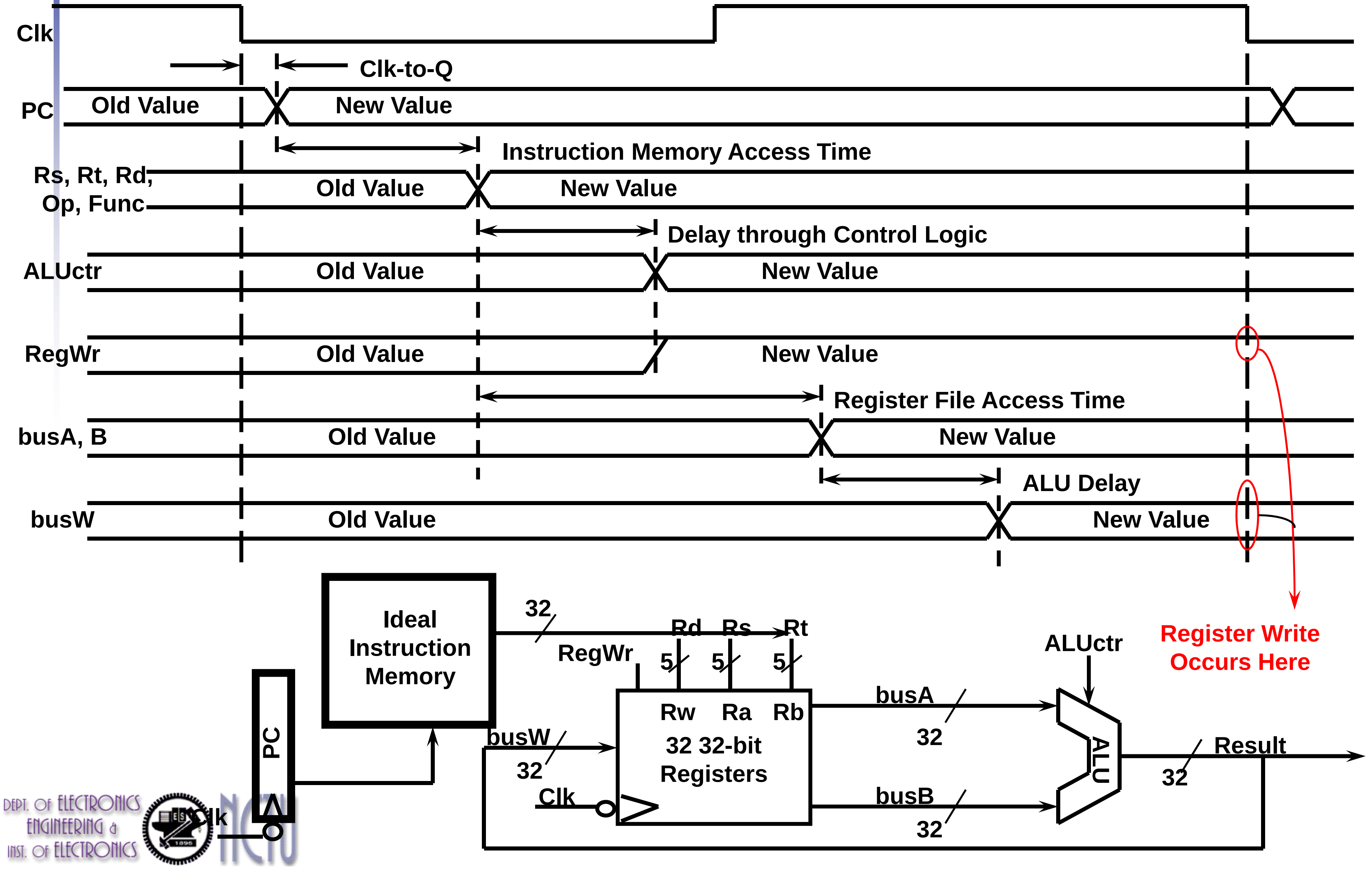
A Single Cycle Full Datapath



Clocking Methodology

- Define when signals are read and written
- Assume edge-triggered (synchronous design):
  - Values in storage (state) elements updated only on a clock edge => clock edge should arrive only after input signals stable
  - Any combinational circuit must have inputs from and outputs to storage elements
  - *Clock cycle*: time for signals to propagate from one storage element, through combinational circuit, to reach the second storage element
  - A register can be read, its value propagated through some combinational circuit, new value is written back to the same register, all in same cycle => no feedback within a single cycle

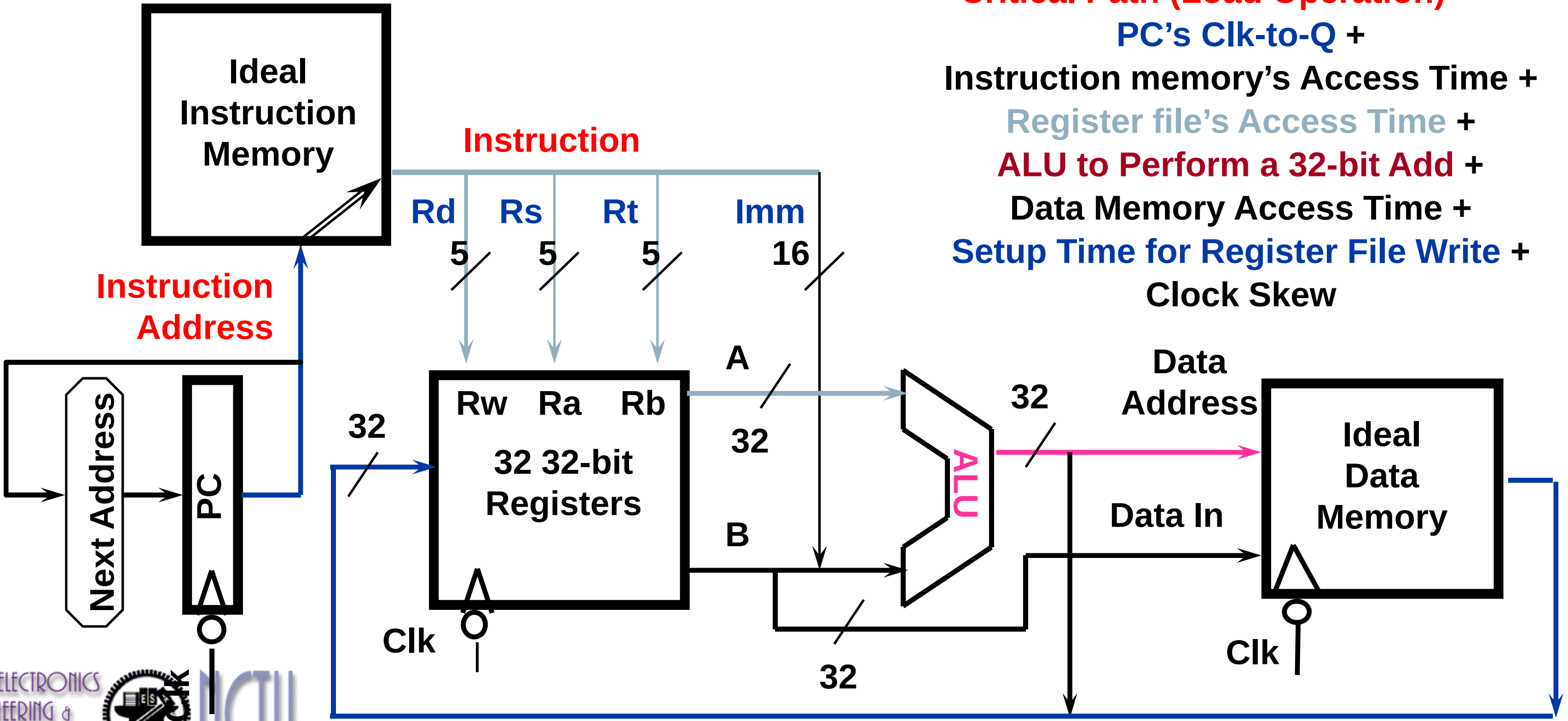
Register-Register Timing



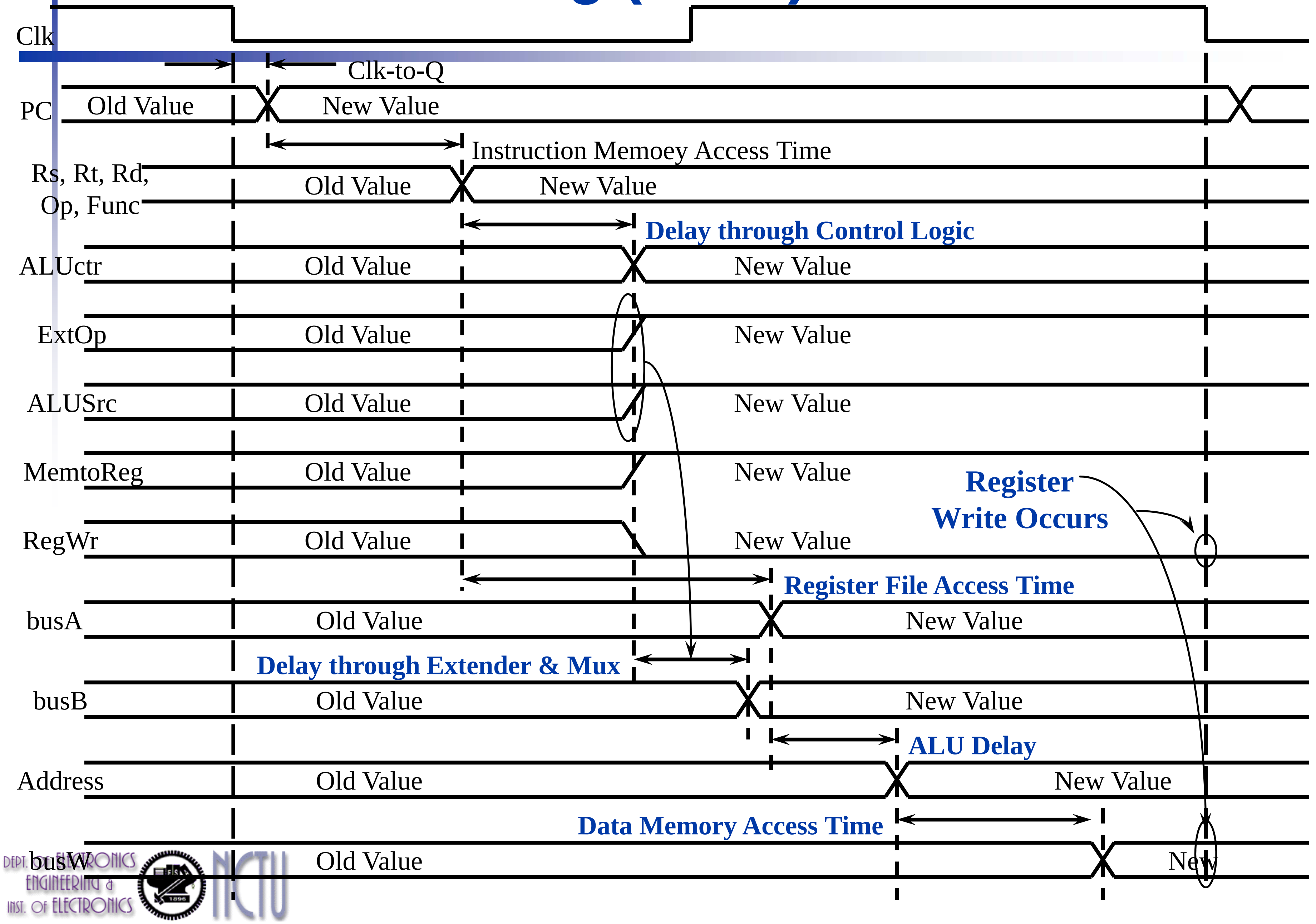
The Critical Path

- Register file and ideal memory:
  - During read, behave as combinational logic:
    - Address valid => Output valid after access time

Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction memory's Access Time +
Register file's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

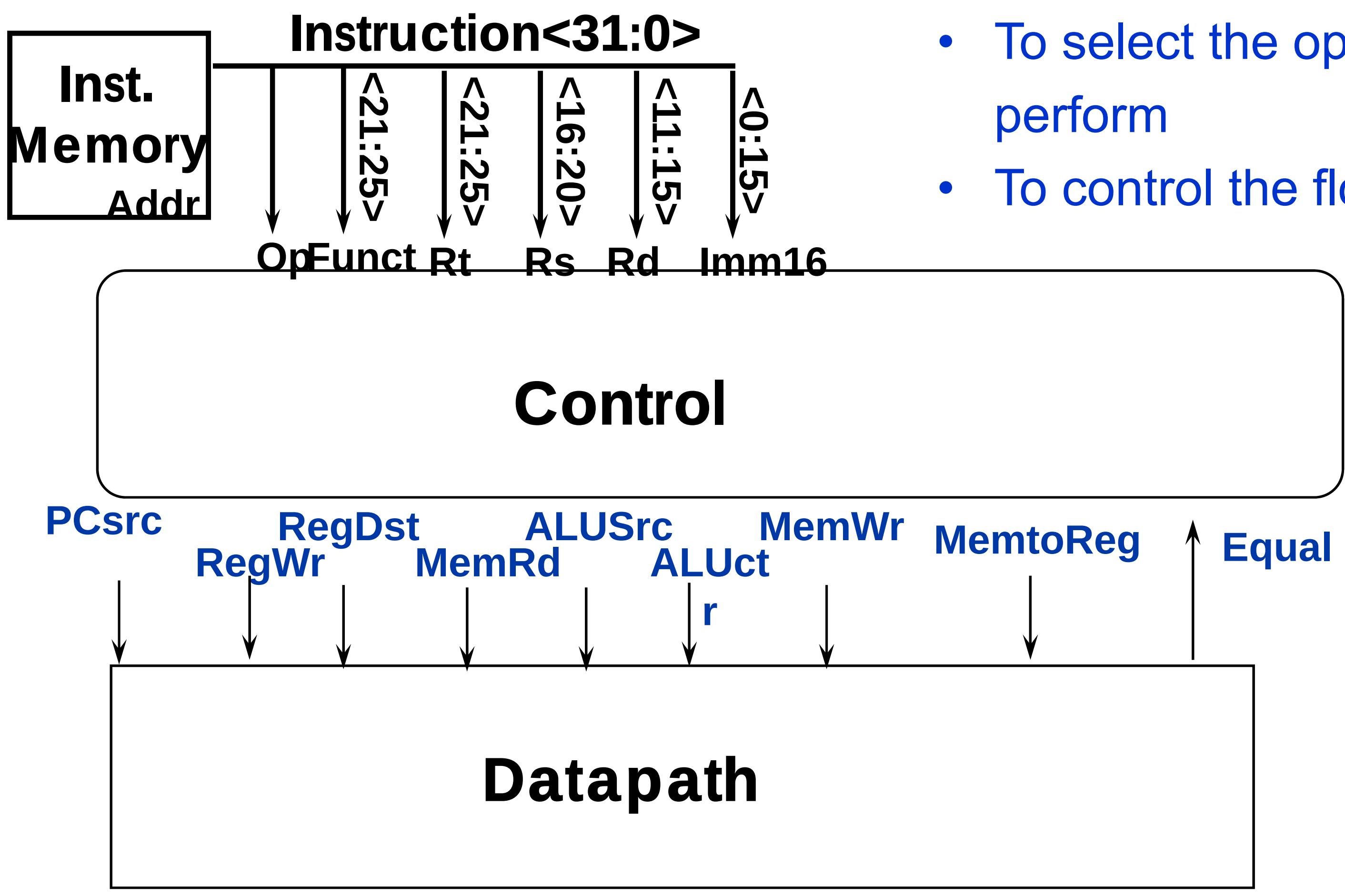


Worst Case Timing (Load)



DEPT. of ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Step 4: Control Points and Signals



- To select the operations to perform
- To control the flow of data

7 Control Signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
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, lower 16 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.

ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

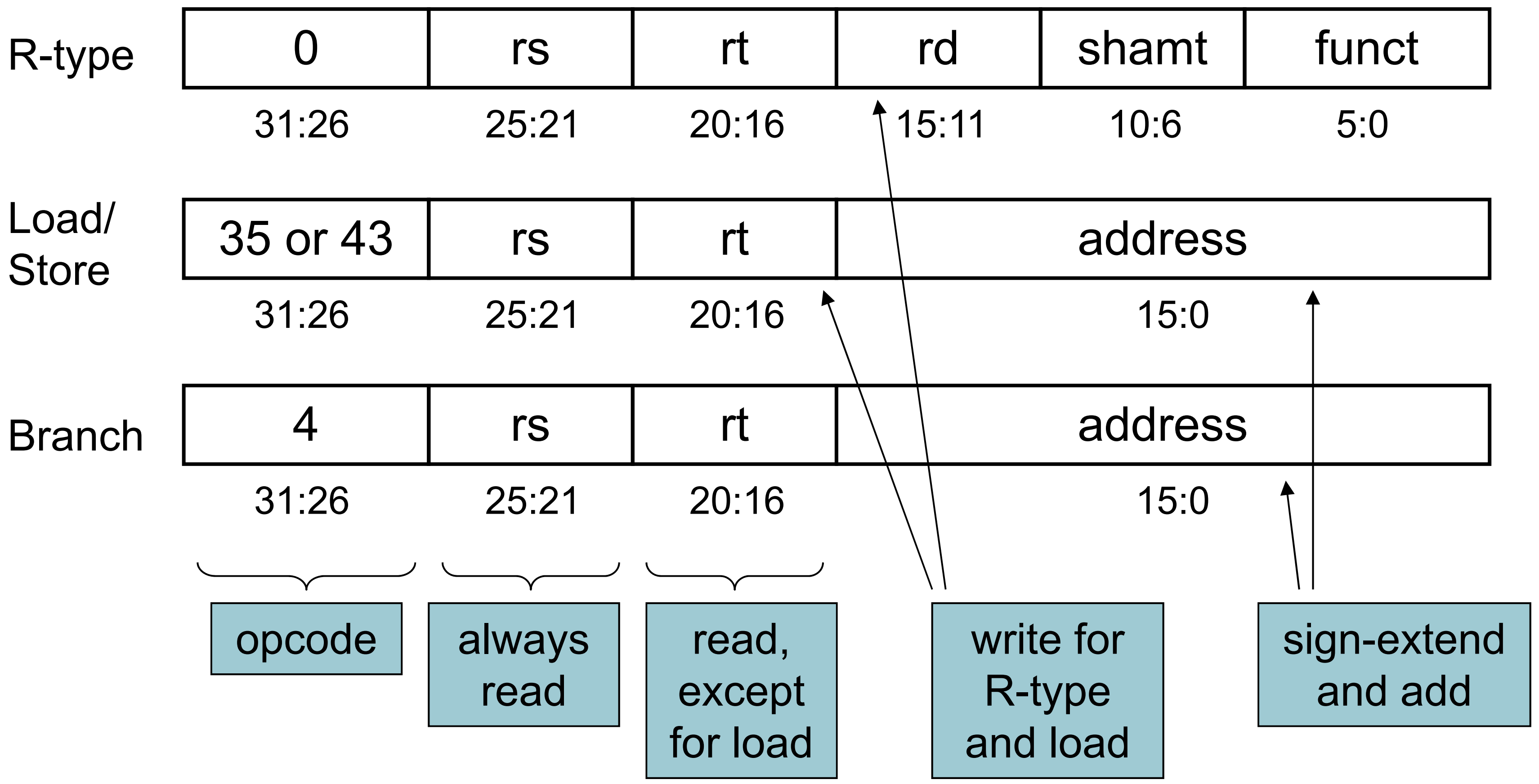
ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

- Control signals derived from instruction

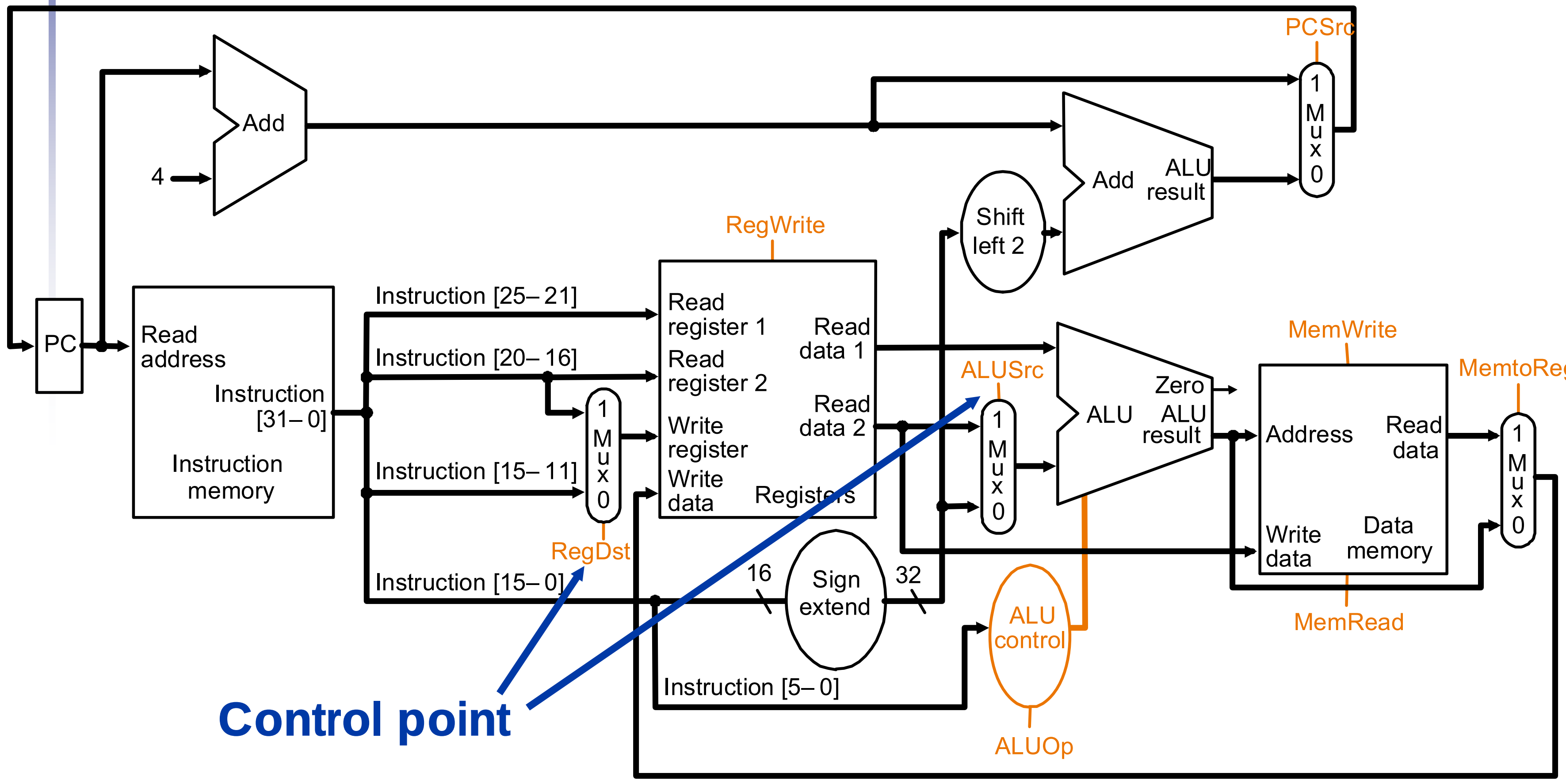


Designing Main Control

- Some observations:
  - opcode (Op[5-0]) is always in bits 31-26
  - two registers to be read are always in rs (bits 25-21) and rt (bits 20-16) (for R-type, beq, sw)
  - base register for lw and sw is always in rs (25-21)
  - 16-bit offset for beq, lw, sw is always in 15-0
  - destination register is in one of two positions:
    - lw: in bits 20-16 (rt)
    - R-type: in bits 15-11 (rd)

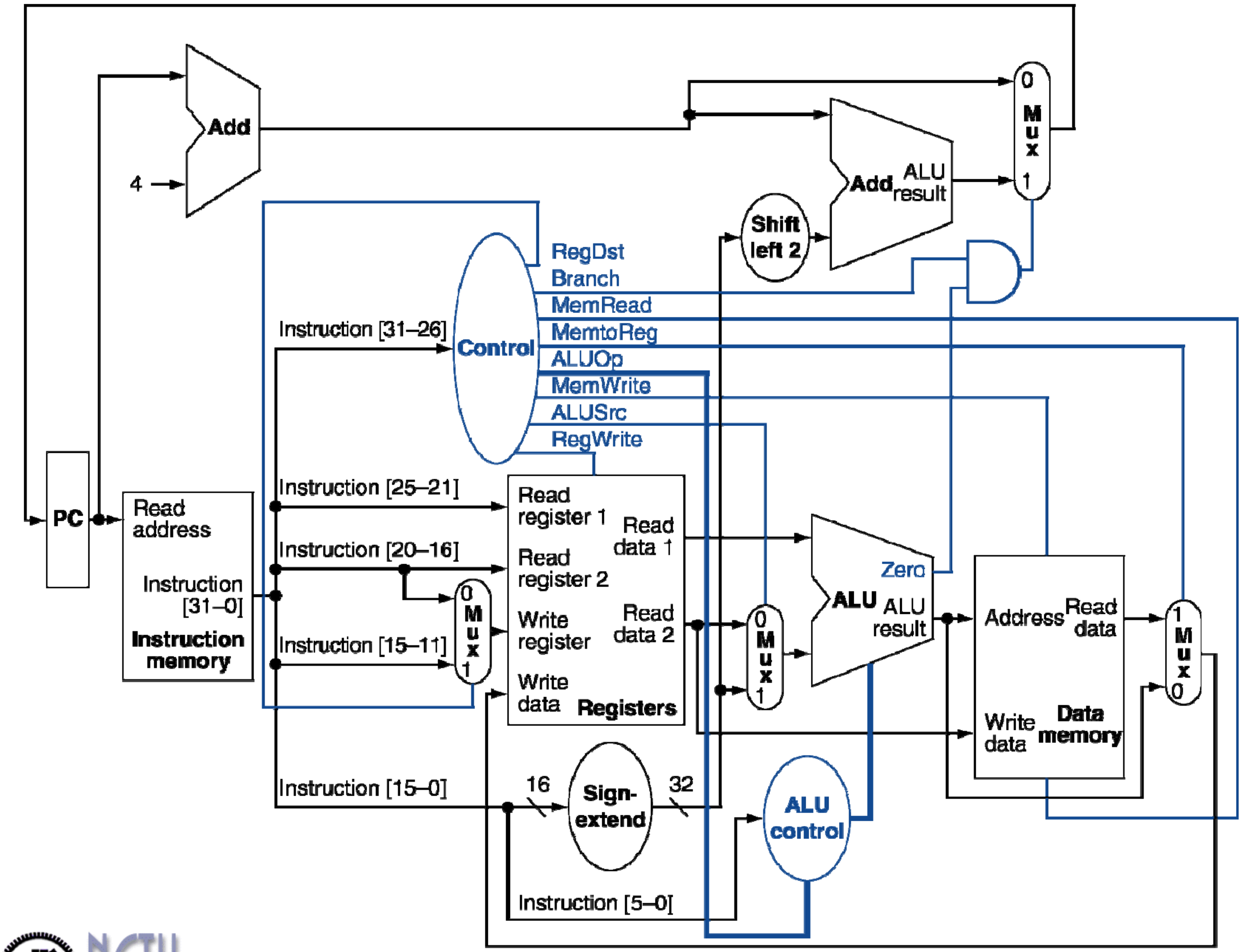
    => need a multiplex to select the address for written register

Datapath with Mux and Control

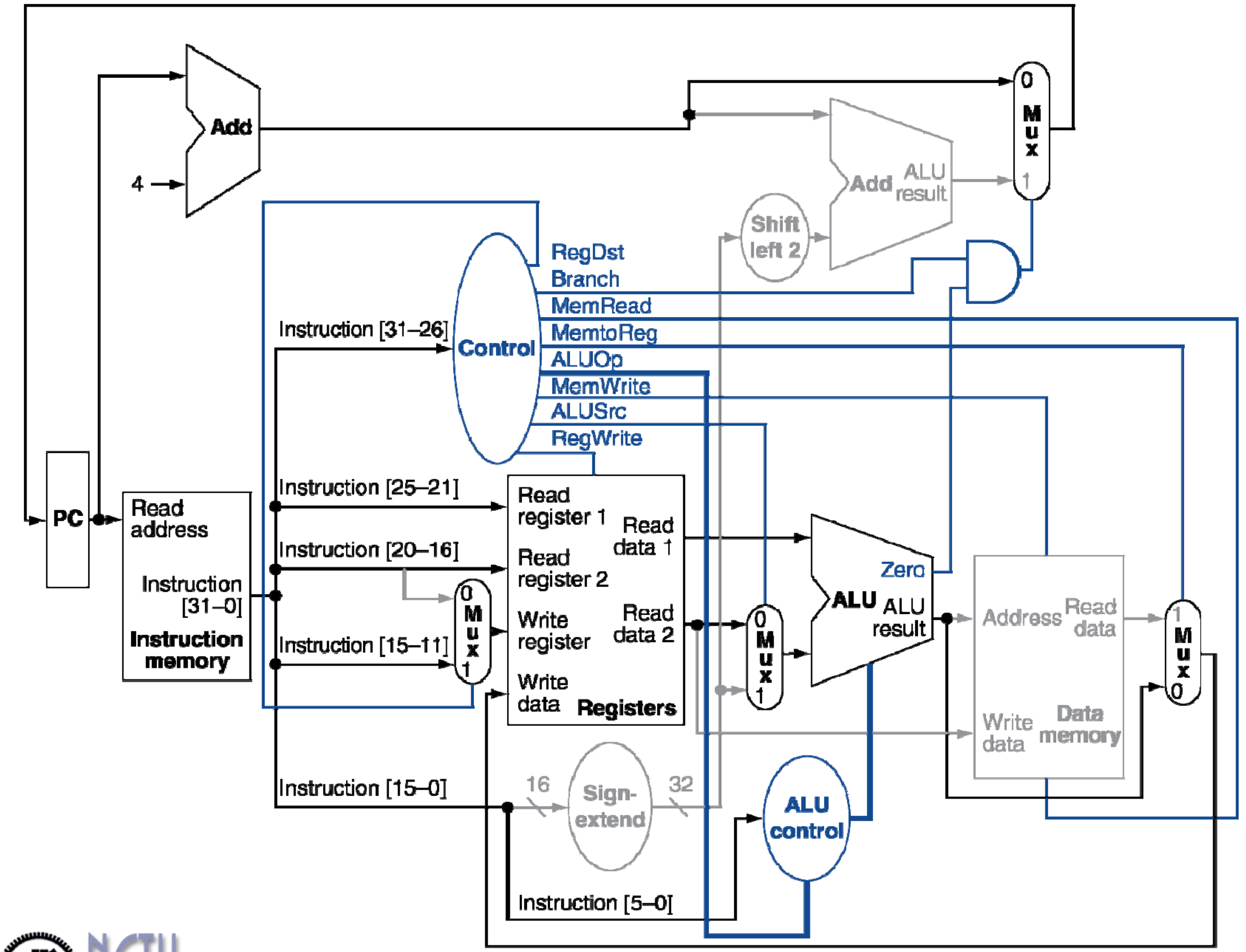


Control point

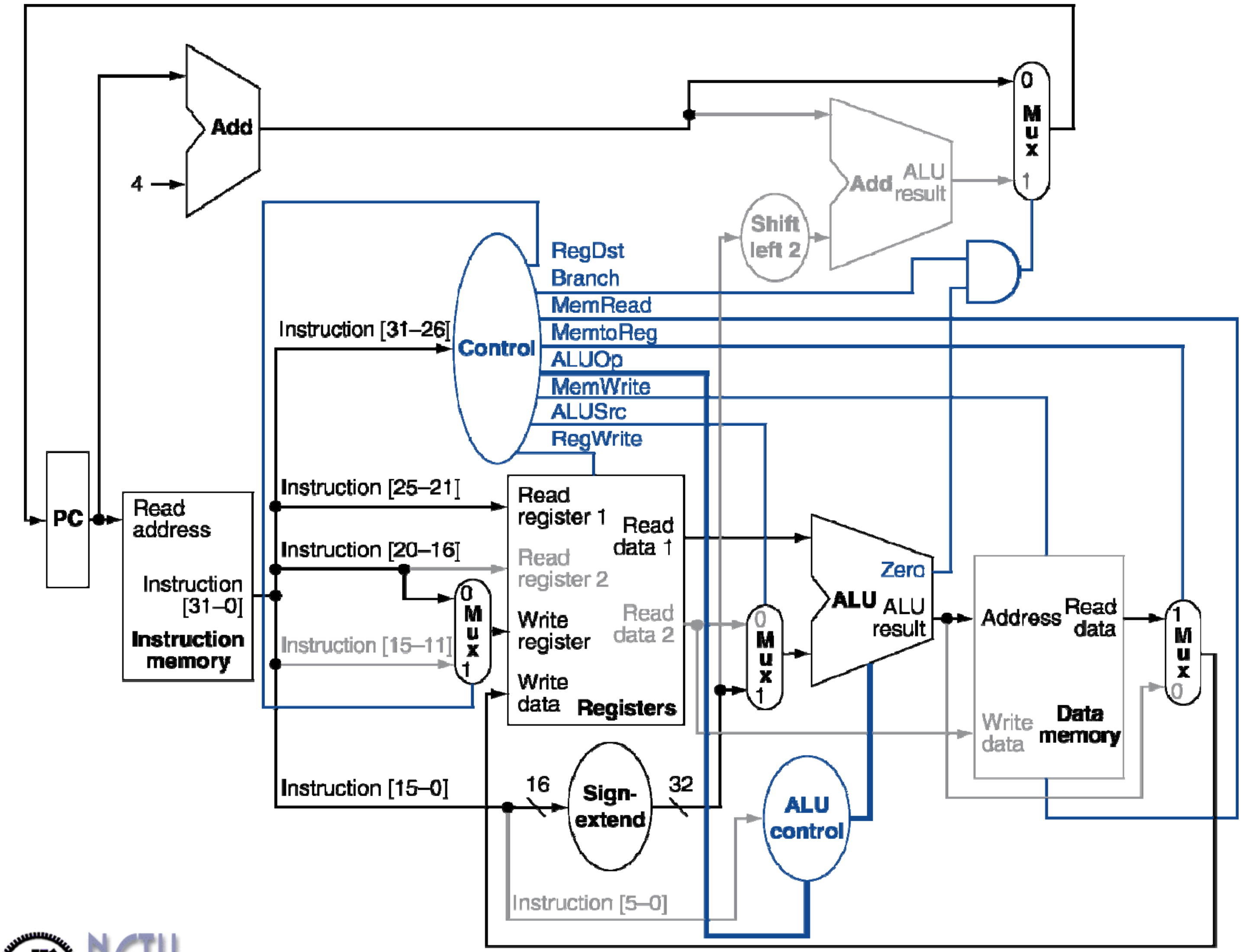
Datapath With Control



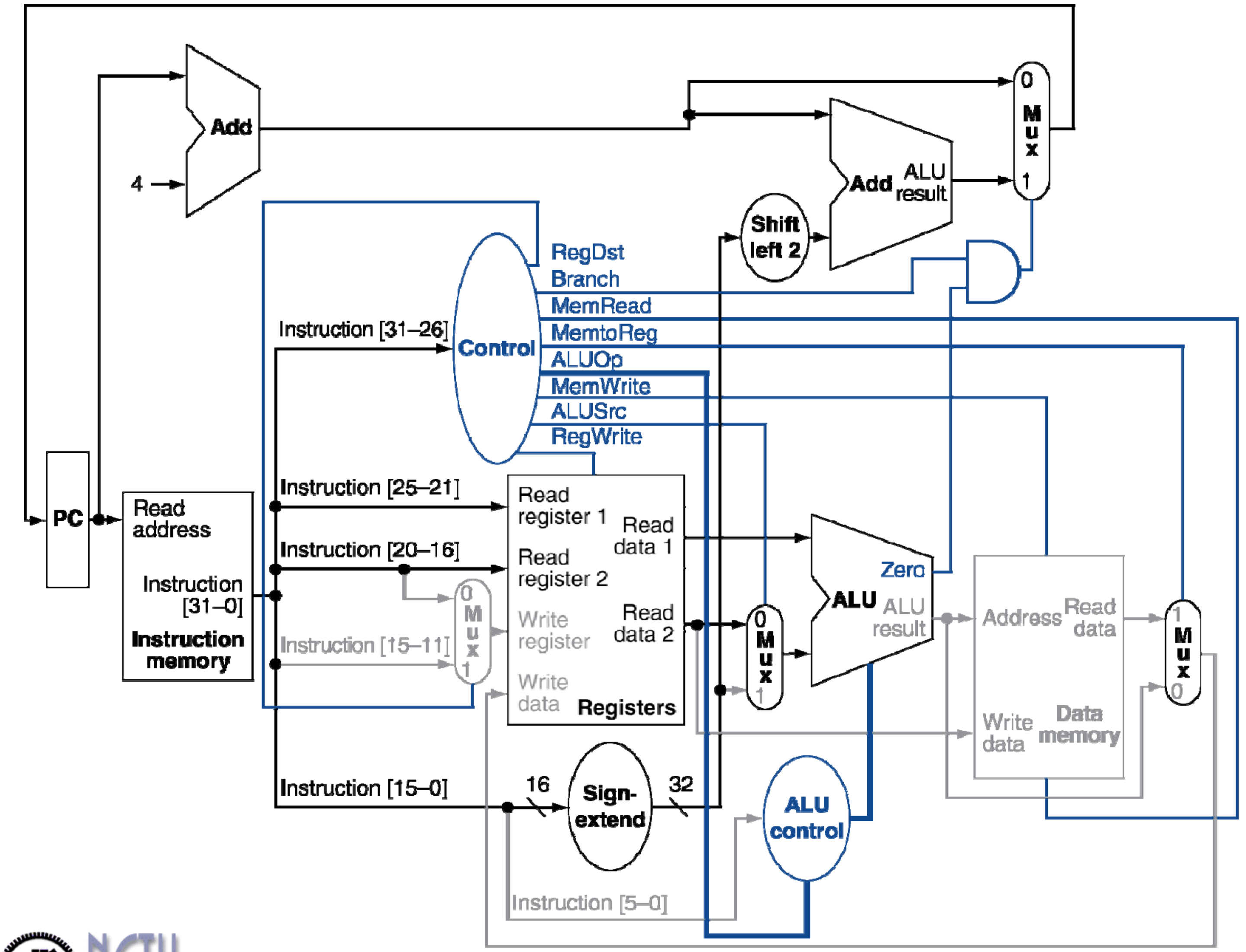
R-Type Instruction



Load Instruction



Branch-on-Equal Instruction

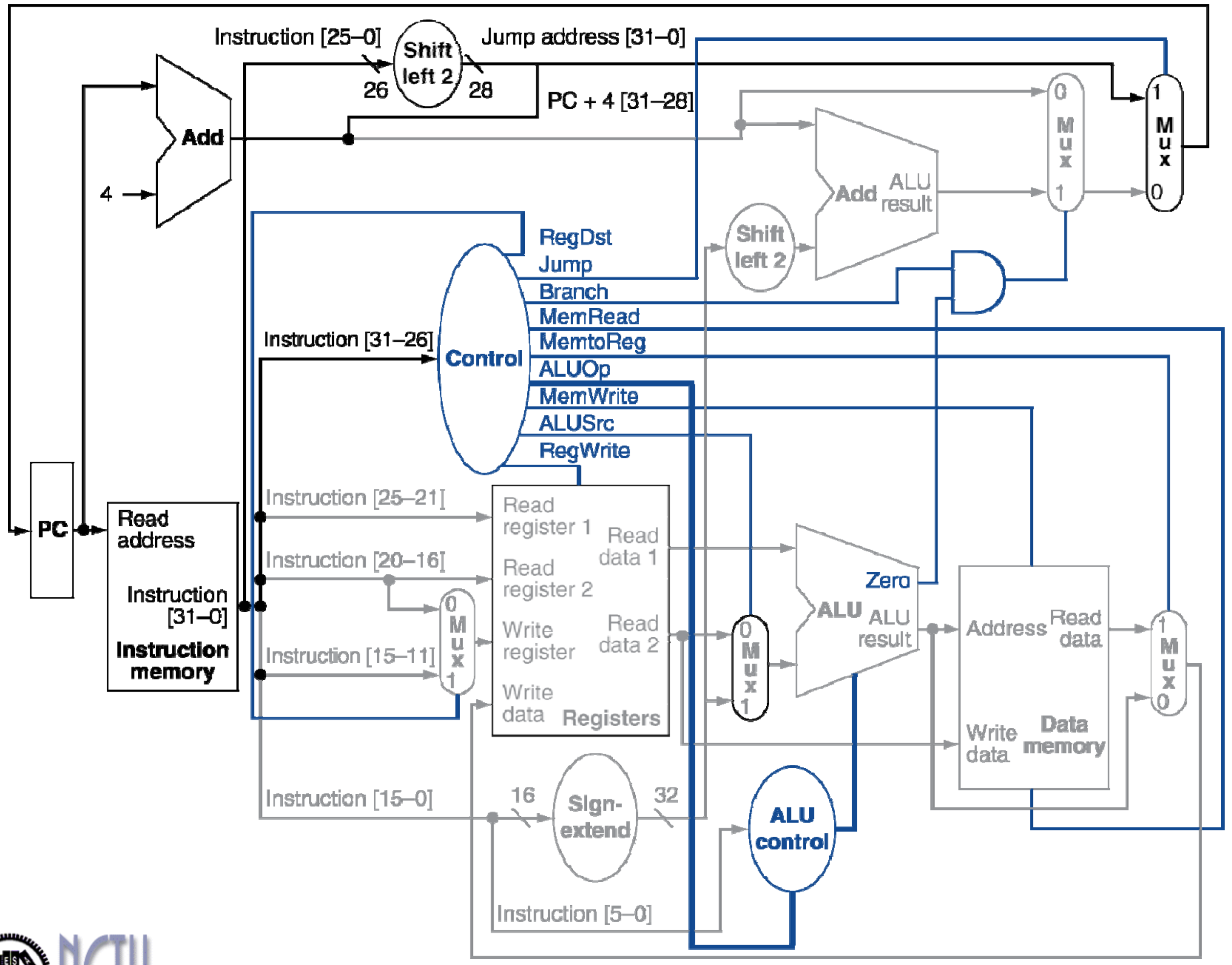


Implementing Jumps

Jump	2	address
	31:26	25:0

- Jump looks somewhat like a branch, but always computes the target PC (i.e. not conditional)
- Jump uses word address
- Update PC with concatenation of top 4 bits of old PC, 26-bit jump address, and 00_2
- Need an extra control signal decoded from opcode

Datapath With Jumps Added

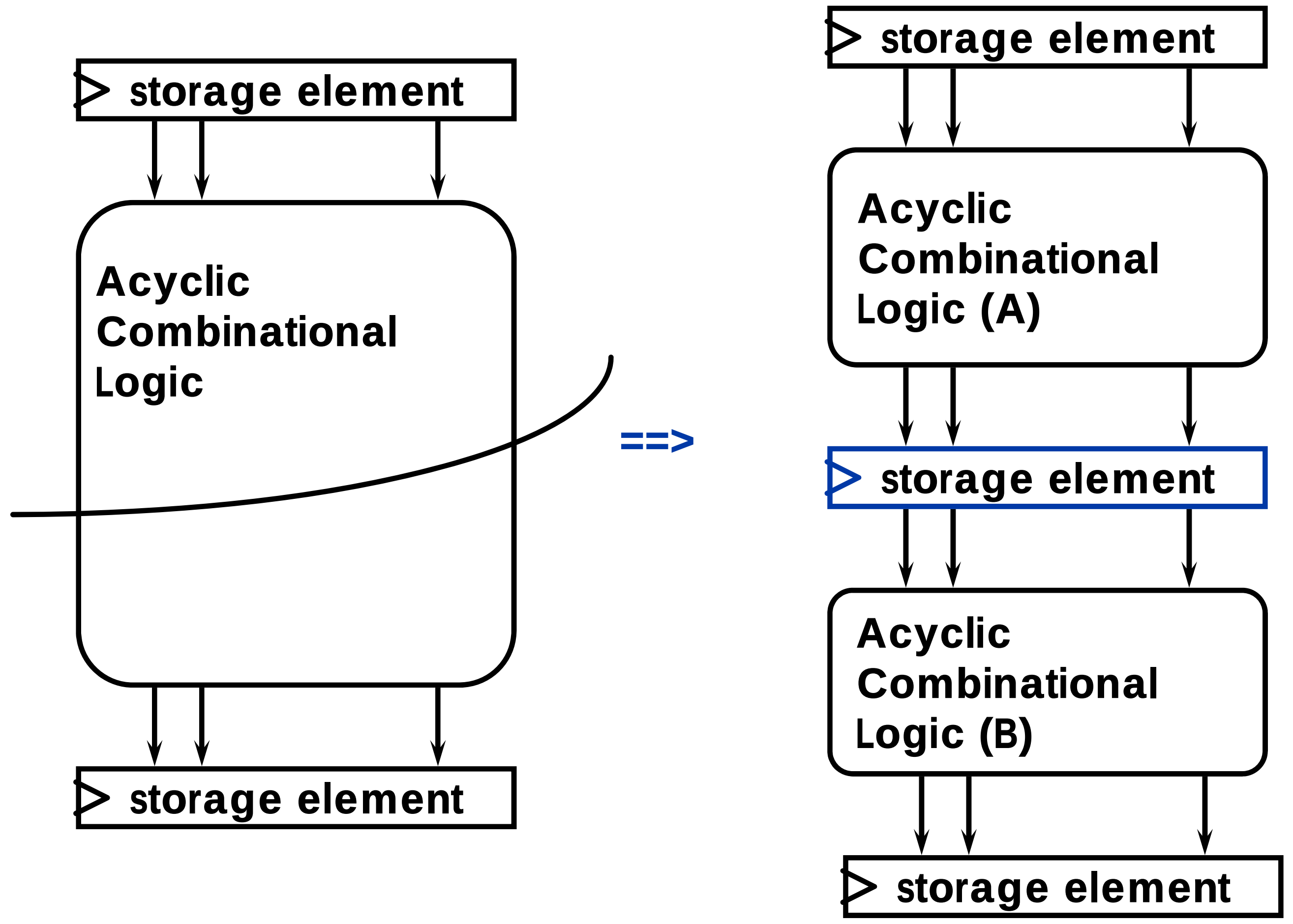


Concluding Remarks

- Not feasible to vary clock period for different instructions
- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- "Making the common case fast" cannot improve the worst-case delay ➔ Single cycle implementation violates the design principle
- We will improve performance by pipelining

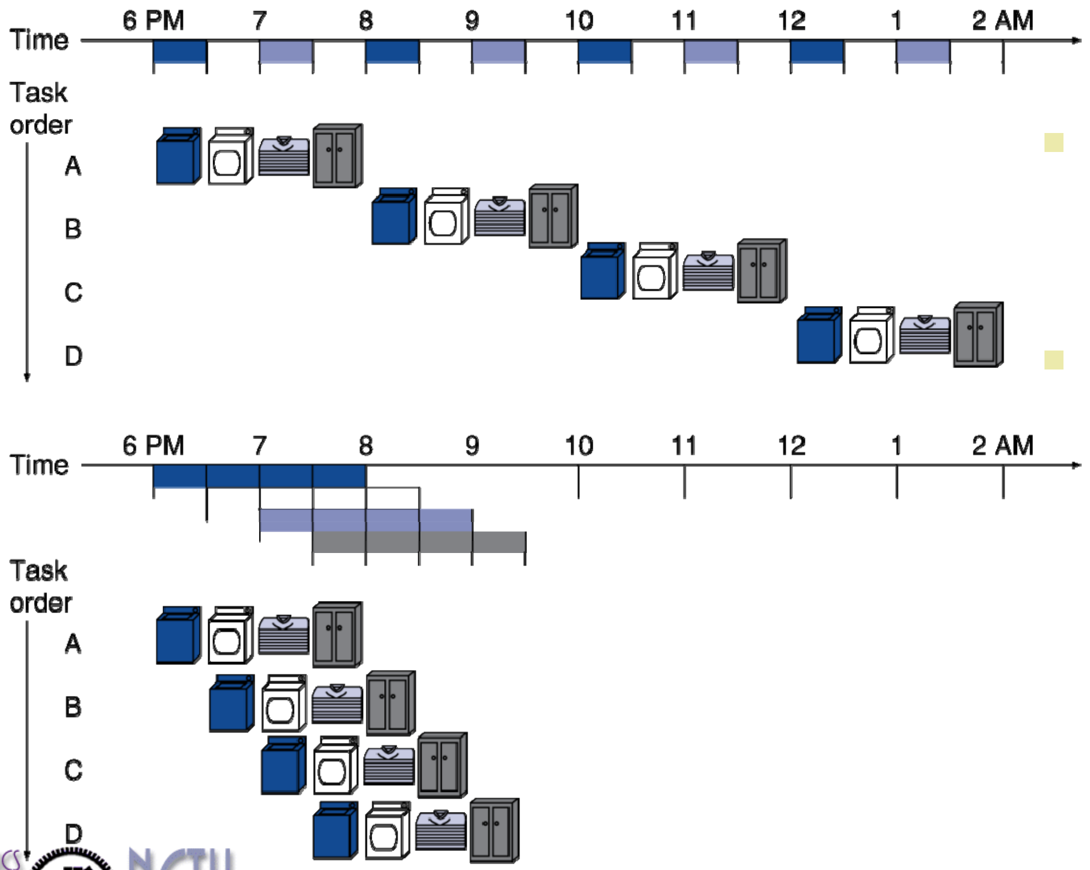
Pipelining Implementation

- Critical path reduction


storage element
Acyclic Combinational Logic
storage element
==>
storage element
Acyclic Combinational Logic (A)
storage element
Acyclic Combinational Logic (B)
storage element

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

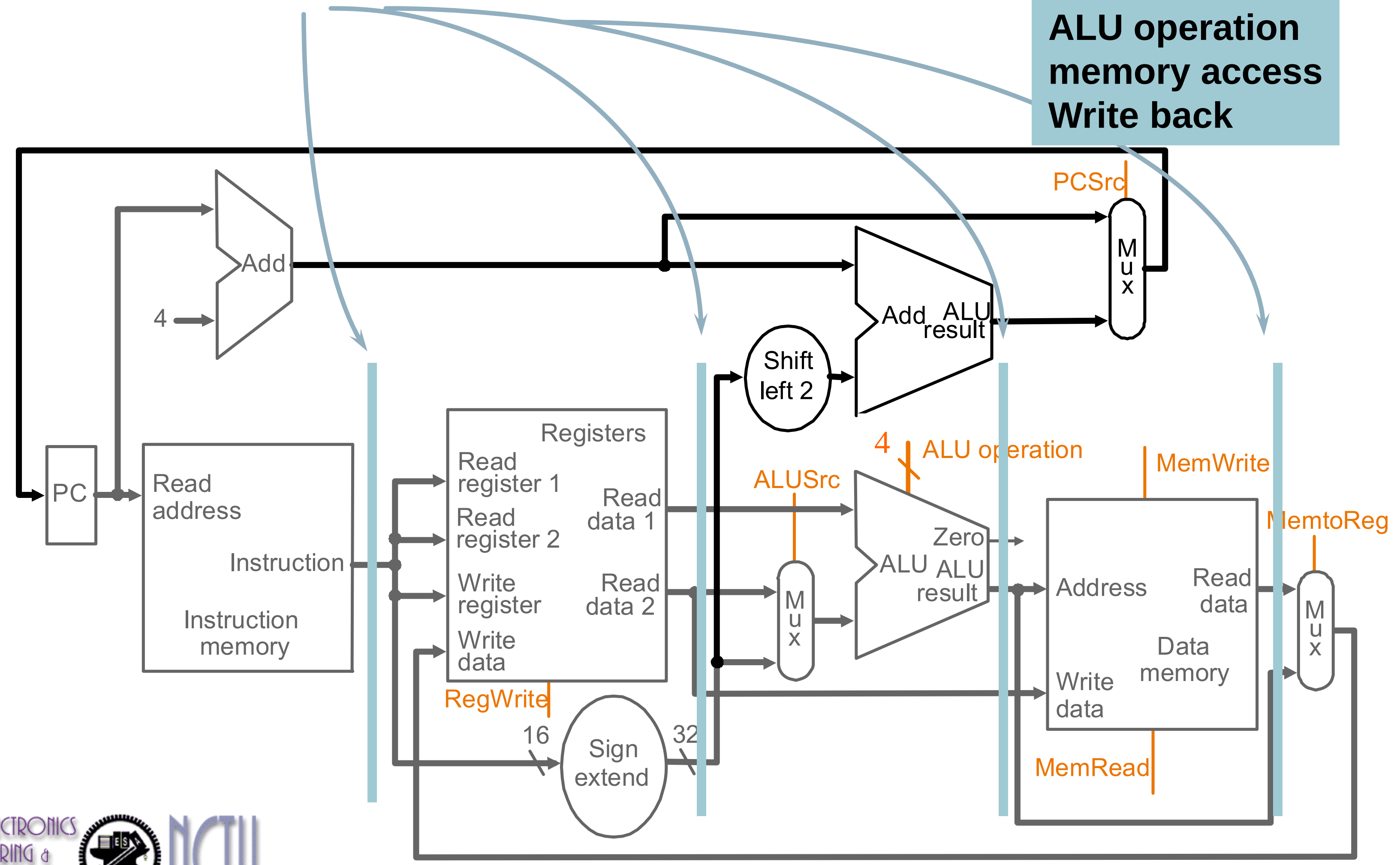
Steps for Designing a Pipelined Processor

- Examine the datapath and control diagram
  - Starting with single cycle datapath
- Partition datapath into stages:
  - IF (instruction fetch), ID (instruction decode and register file read), EX (execution or address calculation), MEM (data memory access), WB (write back)
- Associate resources with stages
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

Partition Single-Cycle Datapath

- Add registers between <u>smallest steps</u>

Ins. fetch
RF access
ALU operation
memory access
Write back



5-Stage MIPS Pipeline

- Five steps, one stage per step
  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

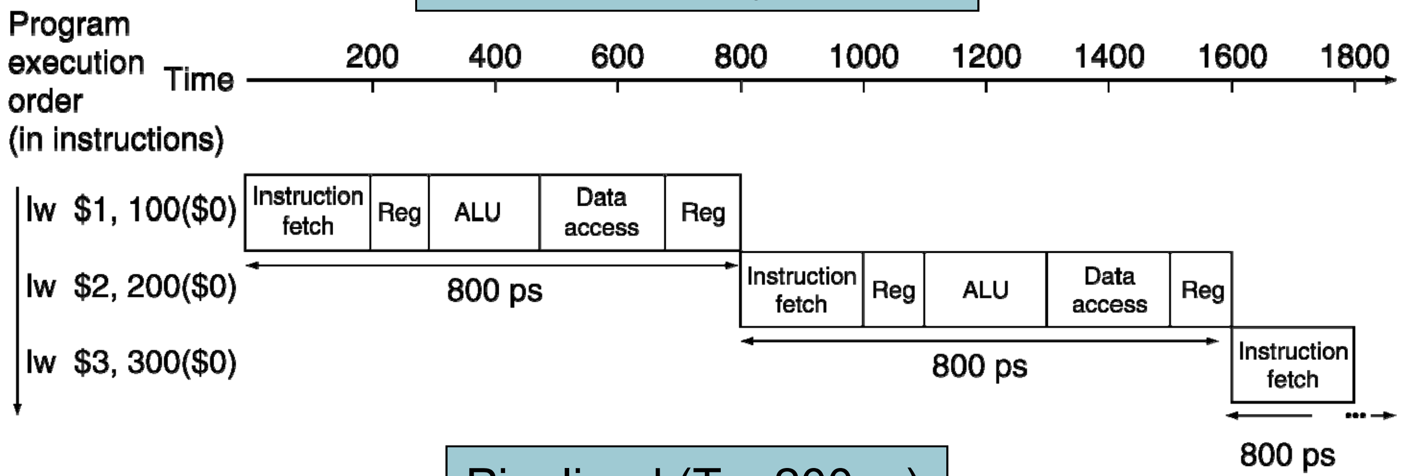
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

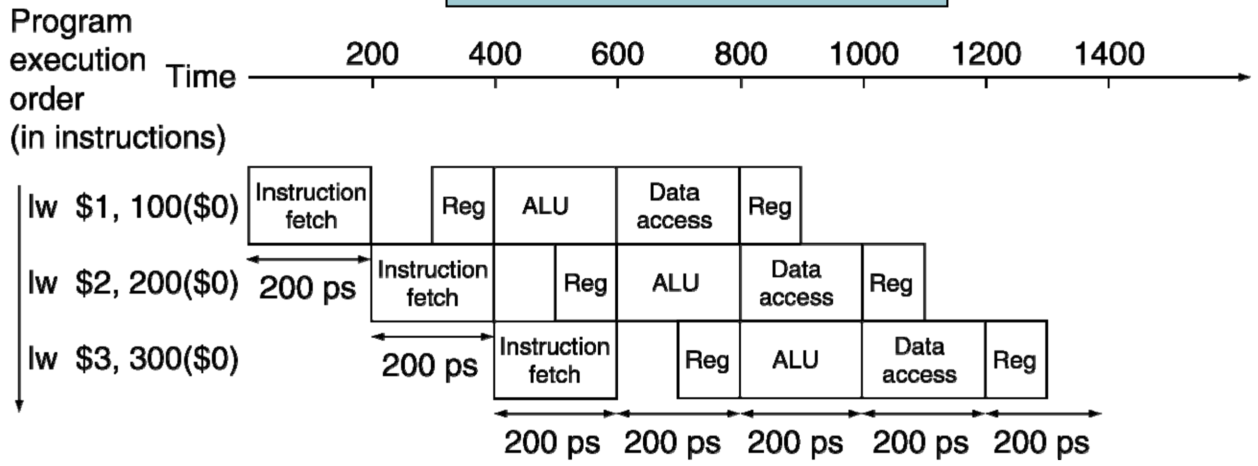
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{\text{pipelined}}$ $= \dfrac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining Lessons

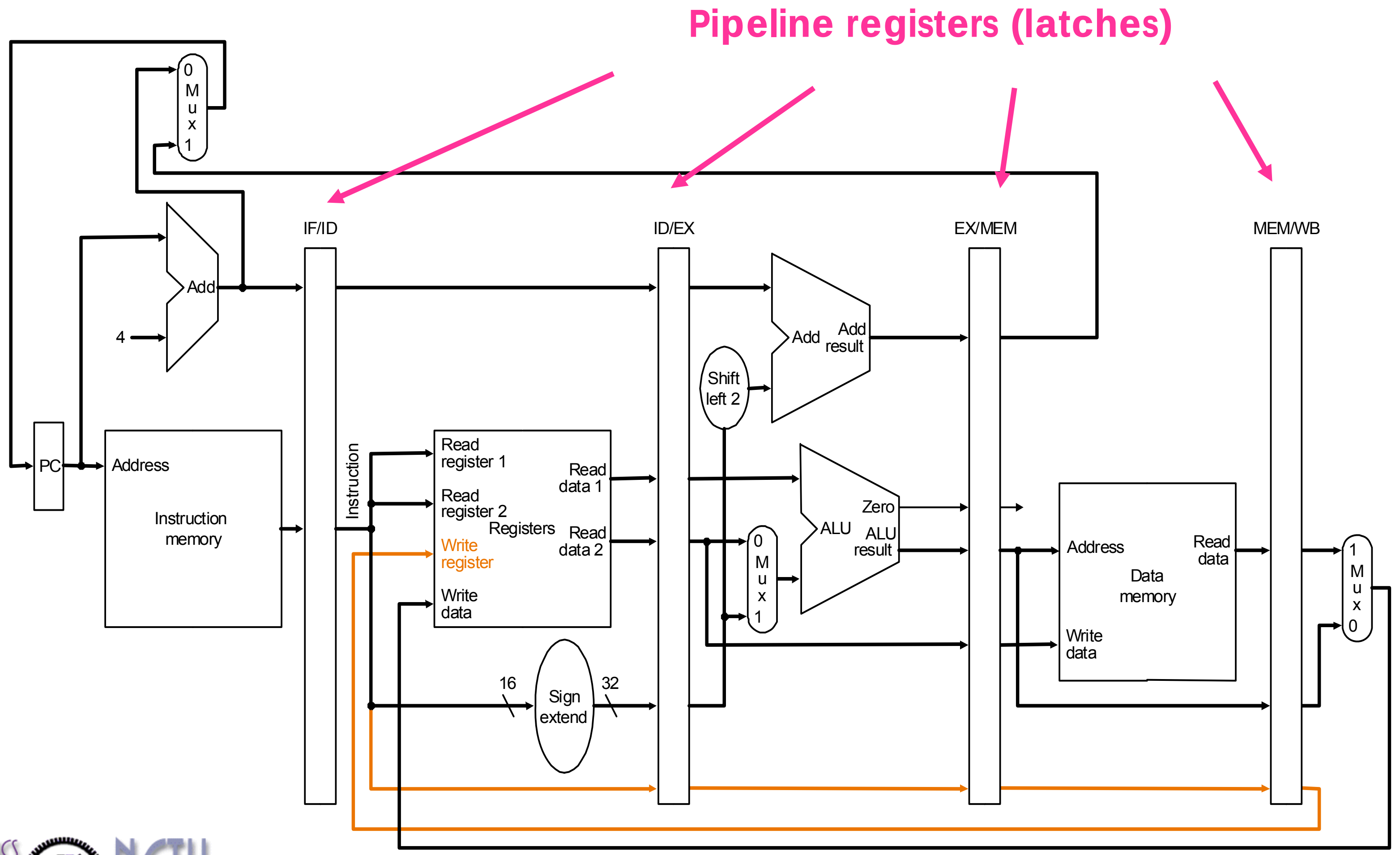
- Doesn’t help latency of single task, but throughput of entire
- Pipeline rate limited by slowest stage
- Multiple tasks working at same time using different resources
- Potential speedup = Number pipe stages
- Unbalanced stage length; time to “fill” & “drain” the pipeline reduce speedup
- **Stall for dependences or pipeline hazards**

MIPS ISA Designed for Pipelining

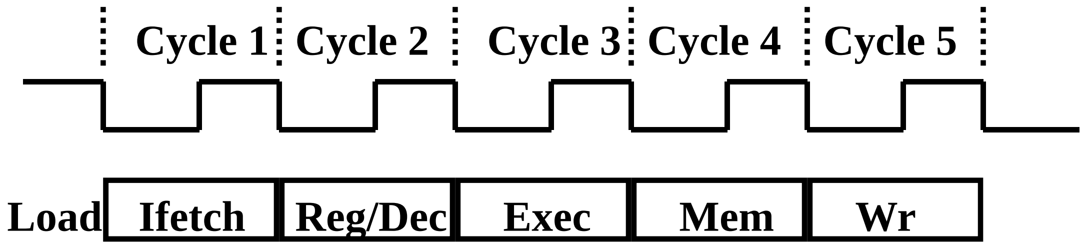
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - c.f. x86: 1- to 17-byte instructions
- Few and regular instruction formats
  - Can decode and read registers in one step
- Load/store addressing
  - Can calculate address in 3rd stage, access memory in 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

Pipelined Datapath

Use registers between stages to carry data and control

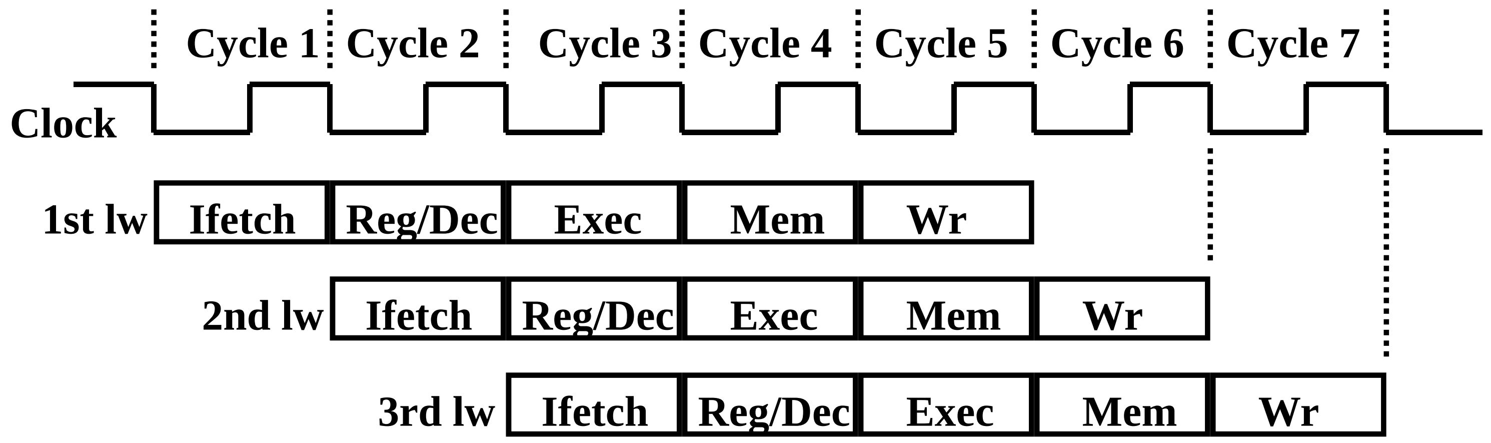


Consider Load Instruction



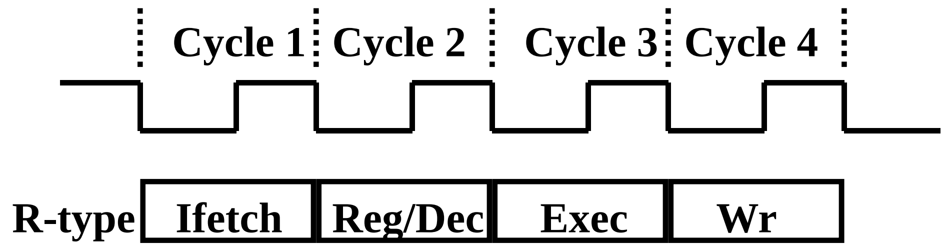
- IF: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- ID: Instruction Decode
  - Registers fetch and instruction decode
- EX: Calculate the memory address
- MEM: Read the data from the Data Memory
- WB: Write the data back to the register file

Pipelining `lw` Instructions



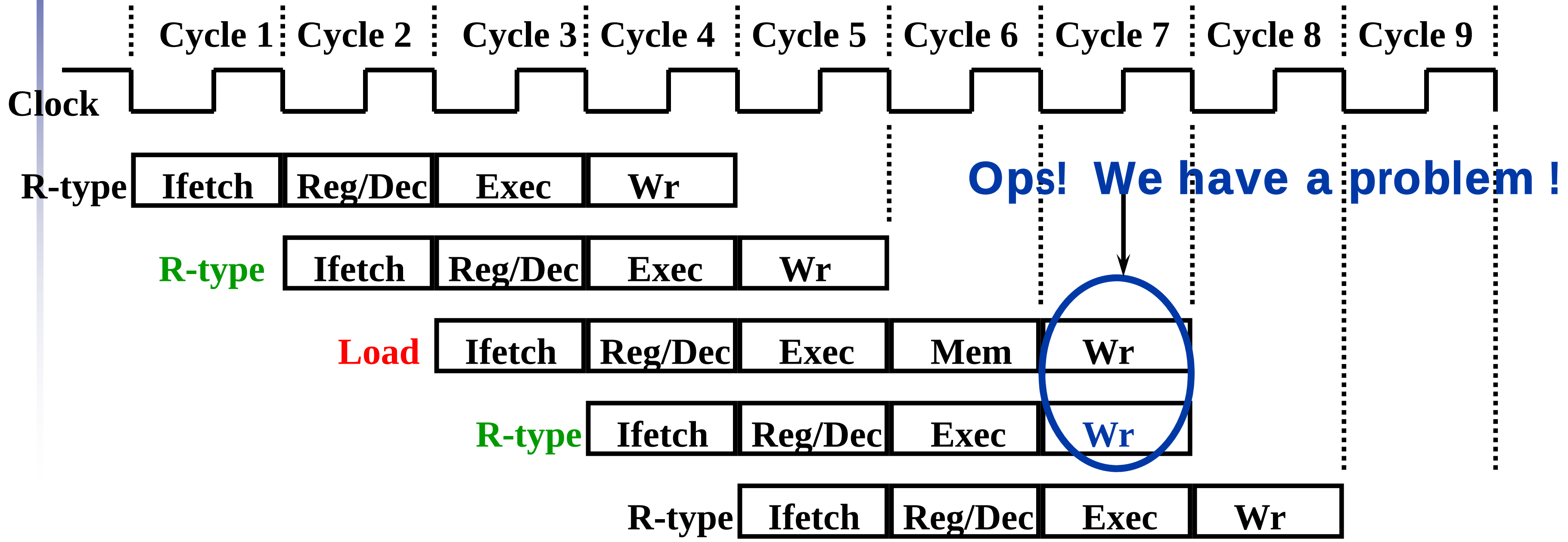
- 5 functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (busA and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the MEM stage
  - Register File's Write port (busW) for the WB stage

The Four Stages of R-type Instruction



- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: ALU operates on the two register operands
- WB: write ALU output back to the register file

Hazard Problem



- We have a *structural hazard*:
  - Two instructions try to write to the RF at the same time, but only one write port !

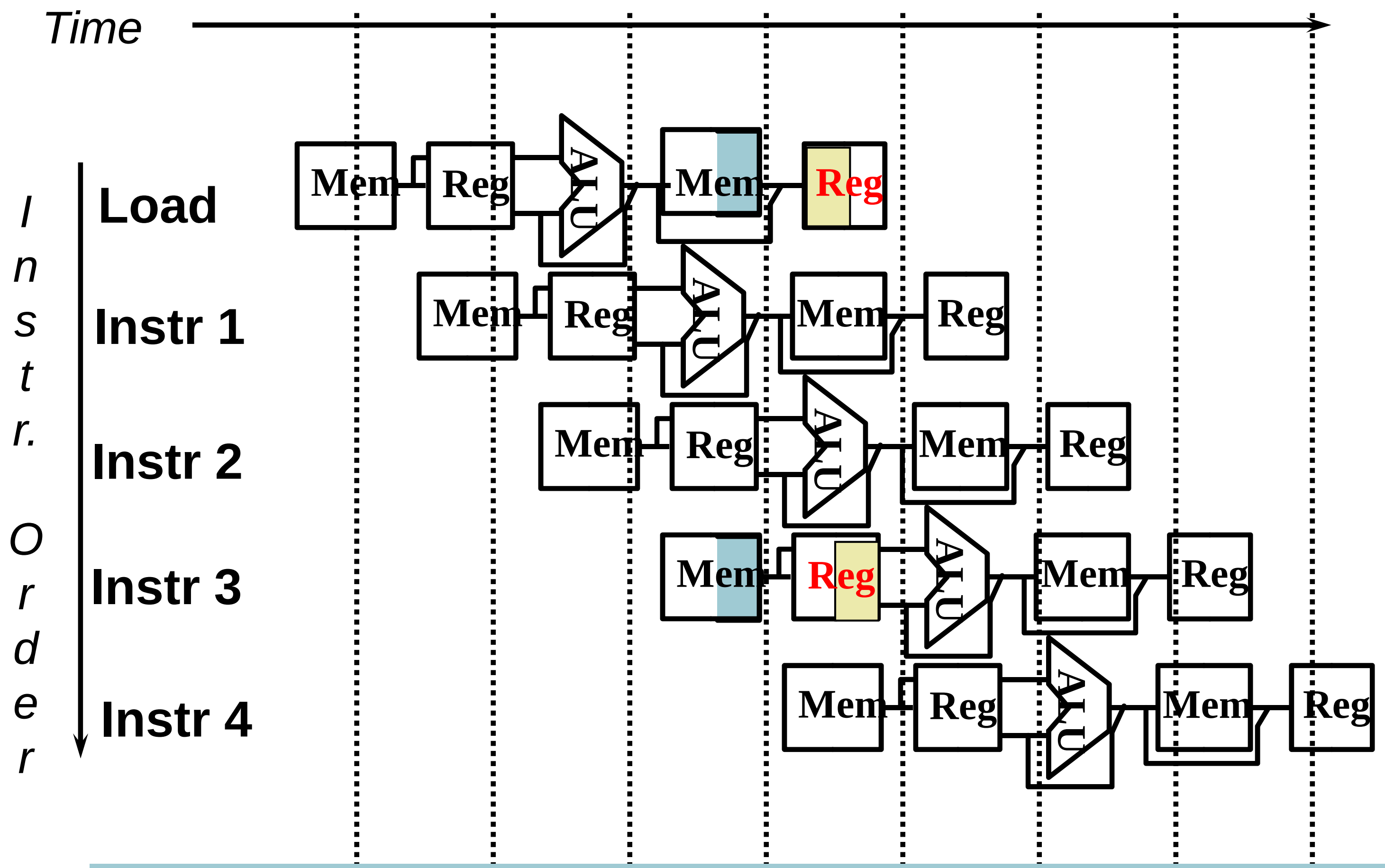
Pipeline Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazard
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction
- *Several ways to solve:* ***forwarding, adding pipeline bubble, making instructions same length***

Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

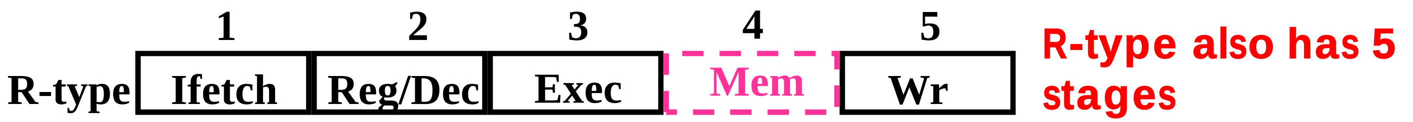
Structural Hazard Solution: Seperate I/D Memory

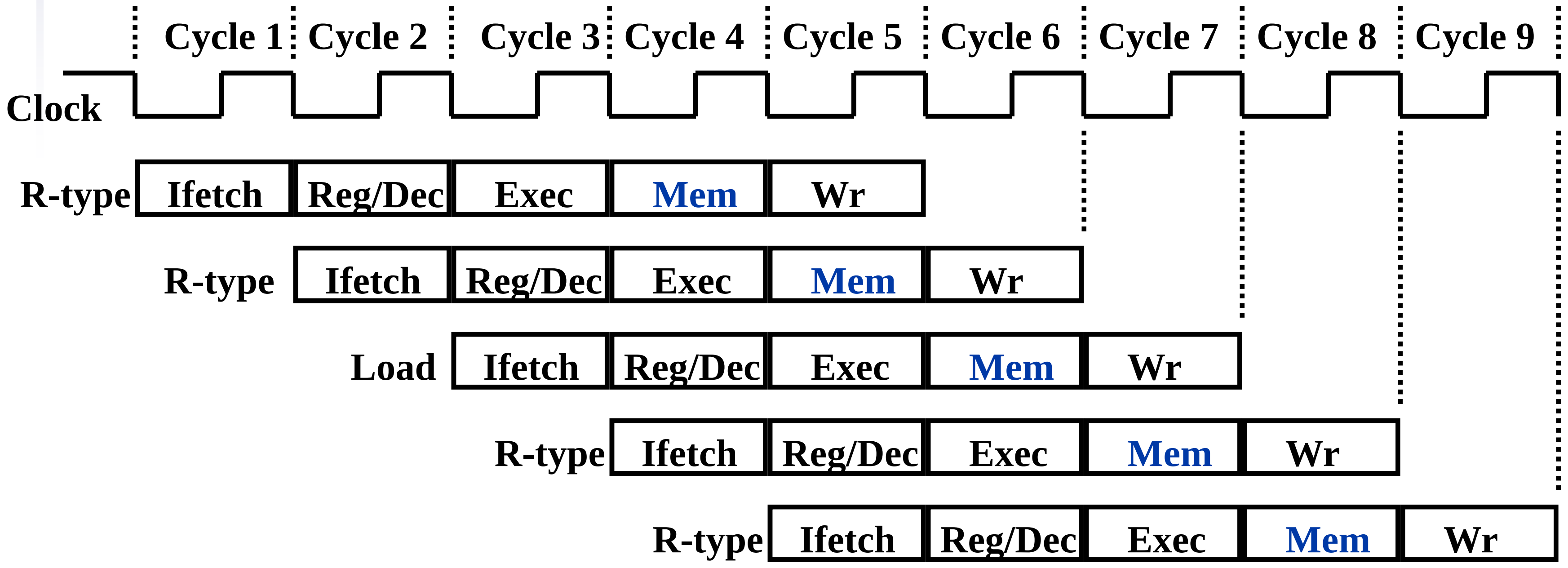


1. I/D separate memory: data memory and instruction memory
2. First half cycle for write and the second half cycle for read

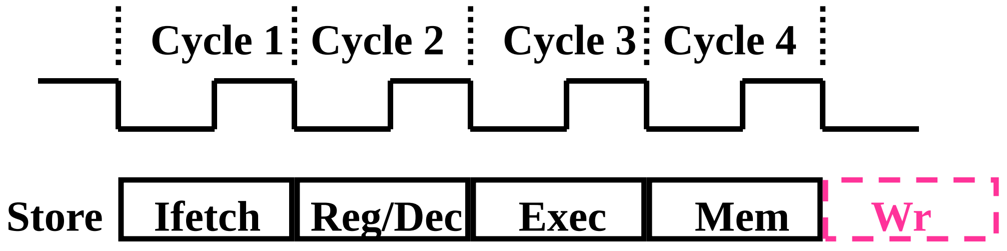
Structural Hazard Solution: Delay R-type's Write

- Delay R-type's register write by one cycle:
  - R-type also use Reg File's write port at Stage 5
  - MEM is a NOP stage: nothing is being done.





The Four Stages of `sw`

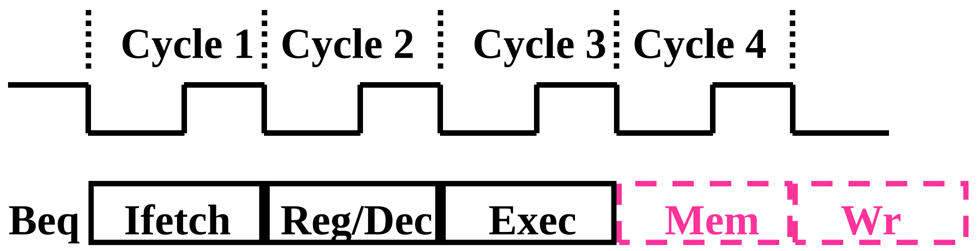


- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: calculate the memory address
- MEM: write the data into the Data Memory

Add an extra stage:

  - WB: NOP

The Three Stages of beq



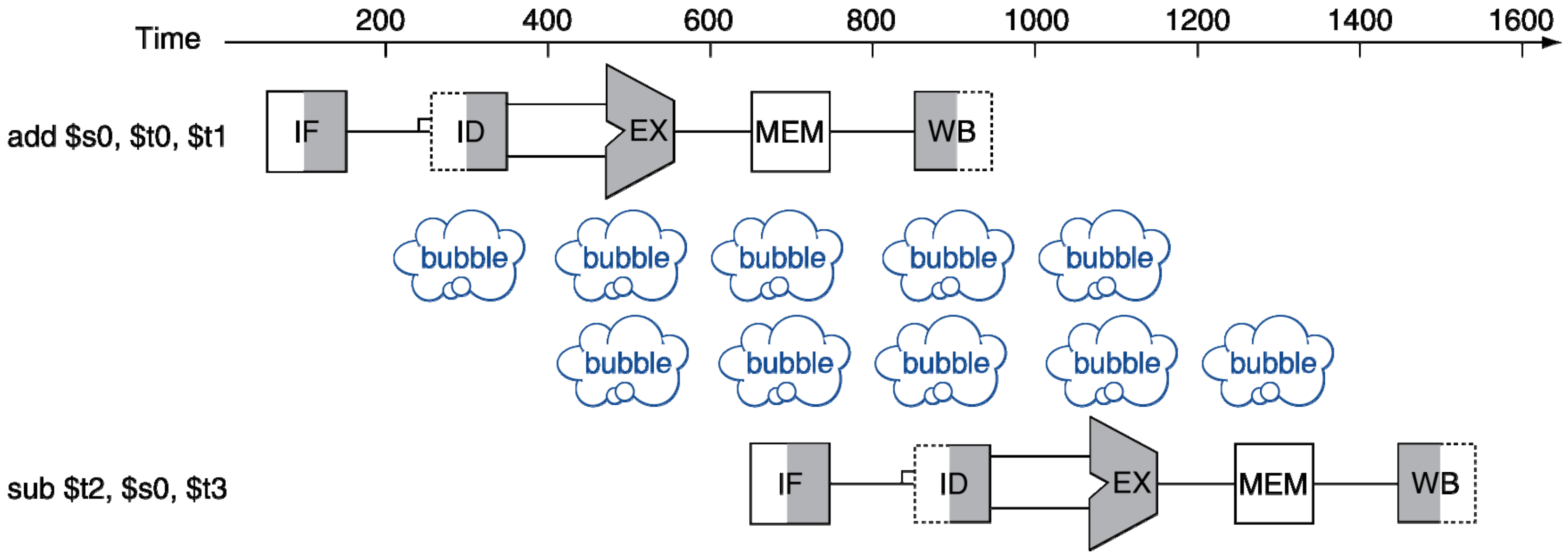
- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX:
  - compares the two register operand
  - select correct branch target address
  - latch into PC

Add two extra stages:
  - MEM: NOP
  - WB: NOP

Data Hazards

- An instruction depends on completion of data access by a previous instruction

  -
    ```
    add   $s0, $t0, $t1
    sub   $t2, $s0, $t3
    ```



Types of Data Hazards

Three types: (inst. i1 followed by inst. i2)

- **RAW (read after write):** **True data dependency**
  i2 tries to read operand before i1 writes it
- **WAR (write after read):** **Name dependency**
  i2 tries to write operand before i1 reads it
  - Gets wrong operand, e.g., autoincrement addr.
  - Can't happen in MIPS 5-stage pipeline because:
    - All instructions take 5 stages, and reads are always in stage 2, and writes are always in stage 5
- **WAW (write after write):** **Name dependency**
  i2 tries to write operand before i1 writes it
  - Leaves wrong result (i1's not i2's); occur only in pipelines that write in more than one stage
  - Can't happen in MIPS 5-stage pipeline because:
    - All instructions take 5 stages, and writes are always in stage 5
- **RAR?** **No dependency**

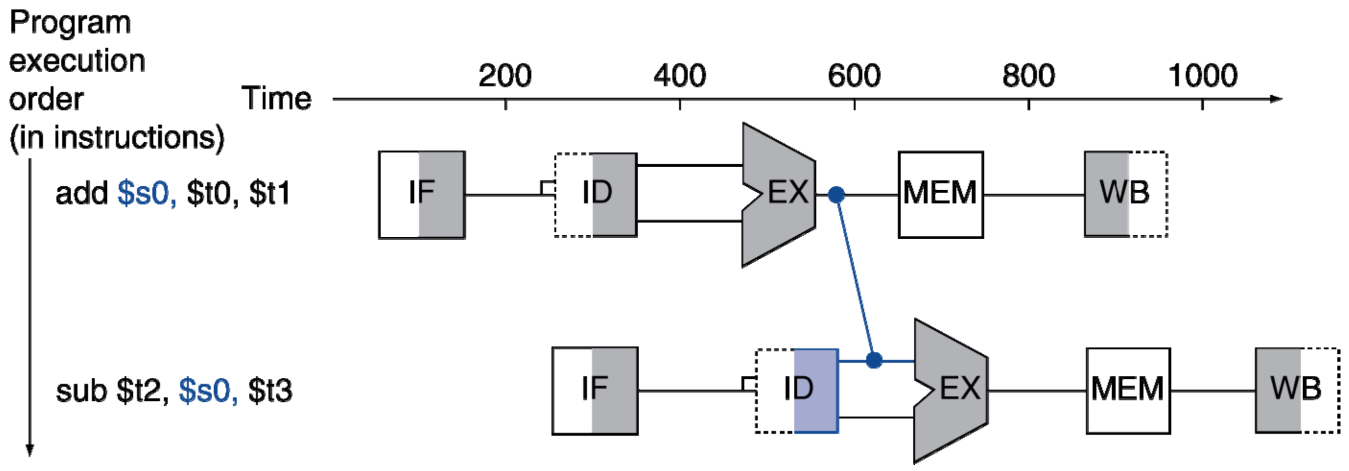
Handling Data Hazards

- Use simple, fixed designs
  - Eliminate WAR by always fetching operands early (ID) in pipeline
  - Eliminate WAW by doing all write backs in order (last stage, static)
  - These features have a lot to do with ISA design
- Internal forwarding in register file:
  - Write in first half of clock and read in second half
  - Read delivers what is written, resolve hazard between sub and add
- Detect and resolve remaining ones
  - Compiler inserts NOP, or reorders the code sequence
  - Forward
  - Stall

Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath

Hardware complexity ⇧

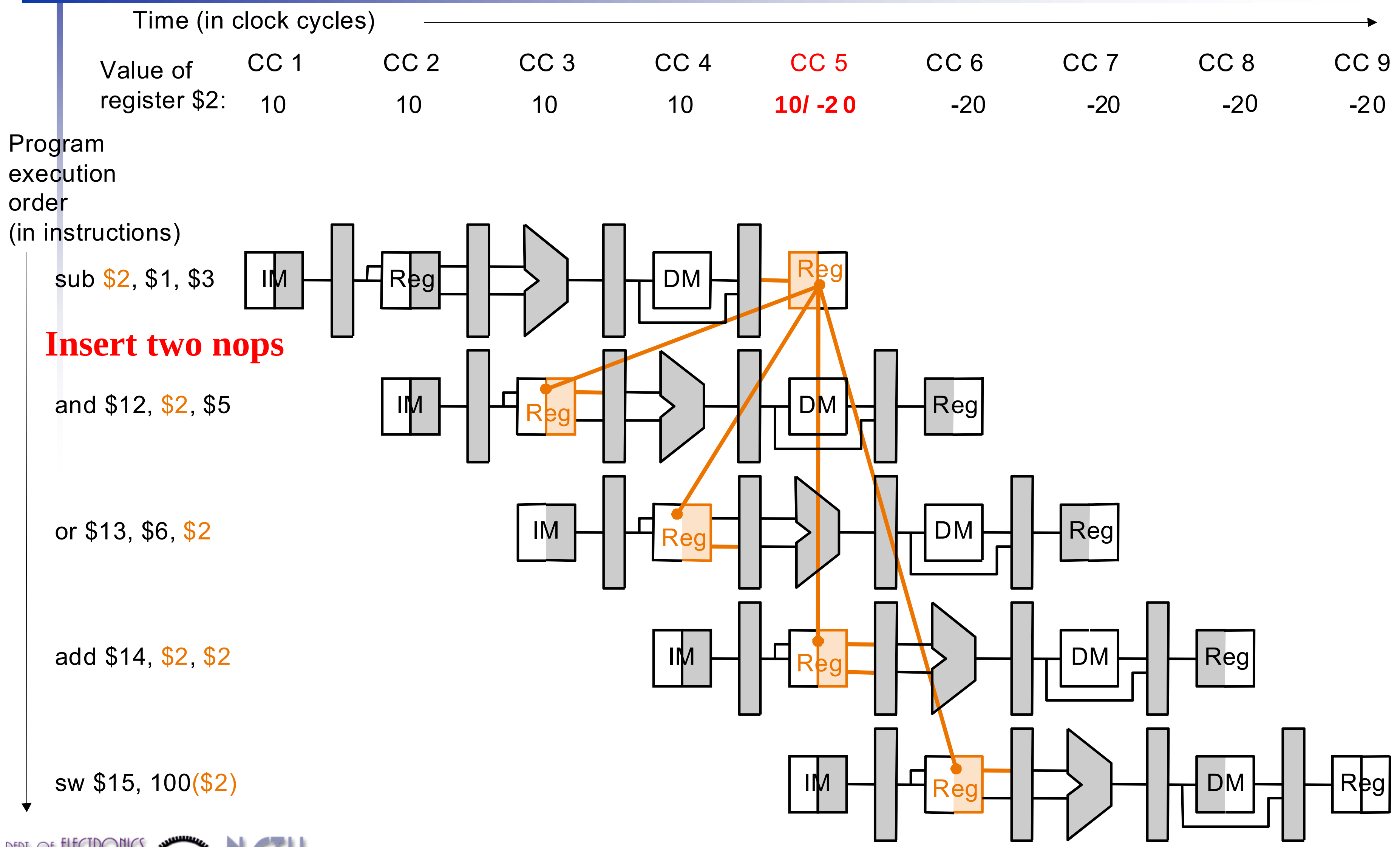


Example

- Consider the following code sequence
-

```
sub $2,  $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

Data Hazards Solution: Inserting NOPs by Software

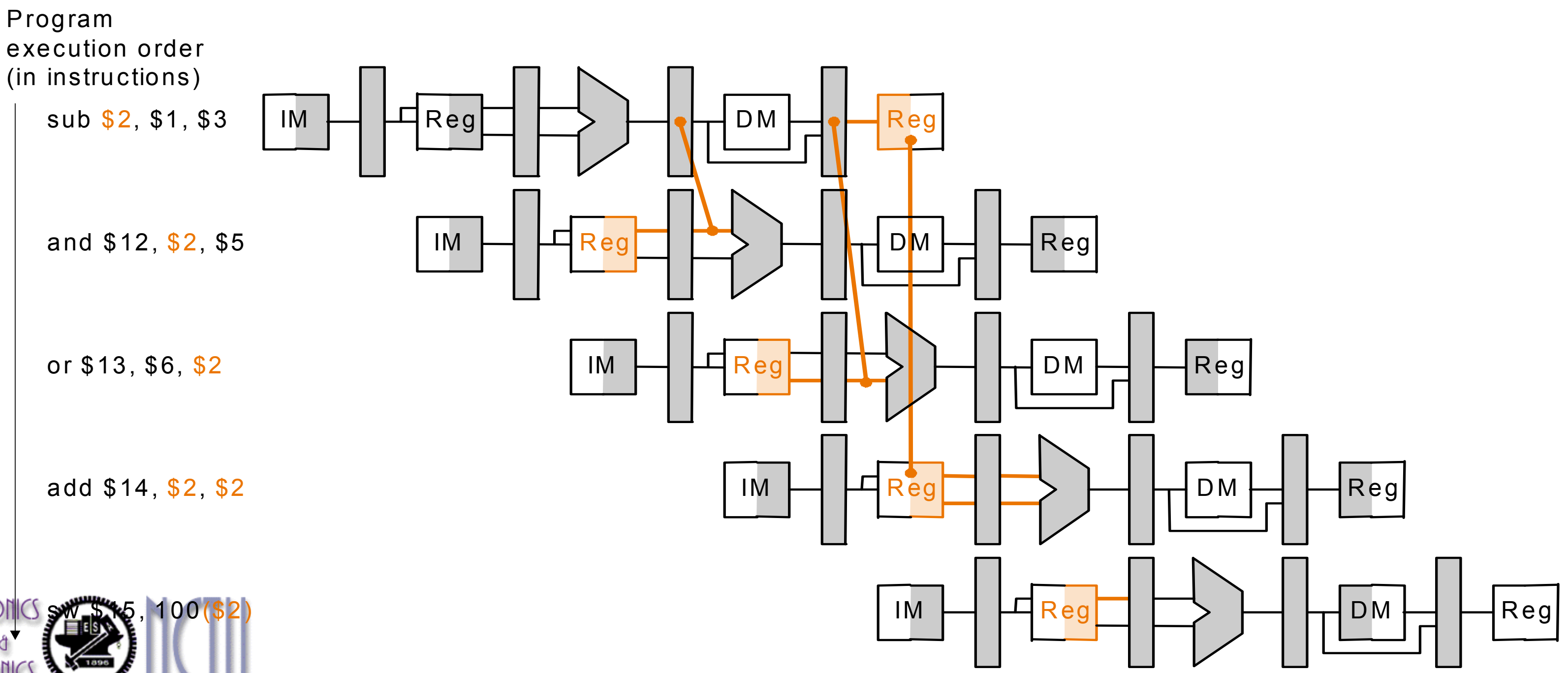


Data Hazards Solution: Internal Forwarding Logic

- Use temporary results, e.g., those in pipeline registers, don't wait for them to be written

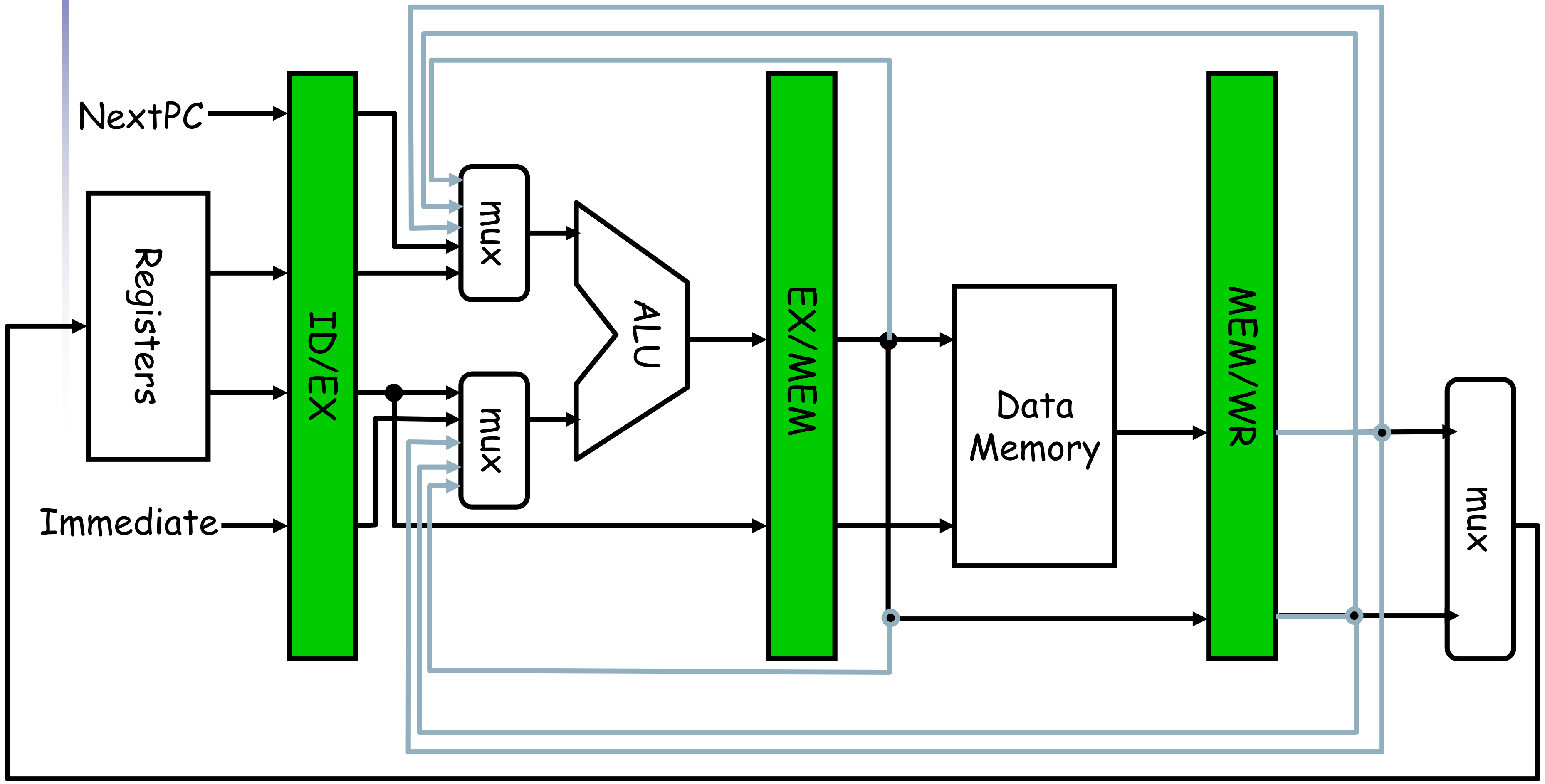
Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X



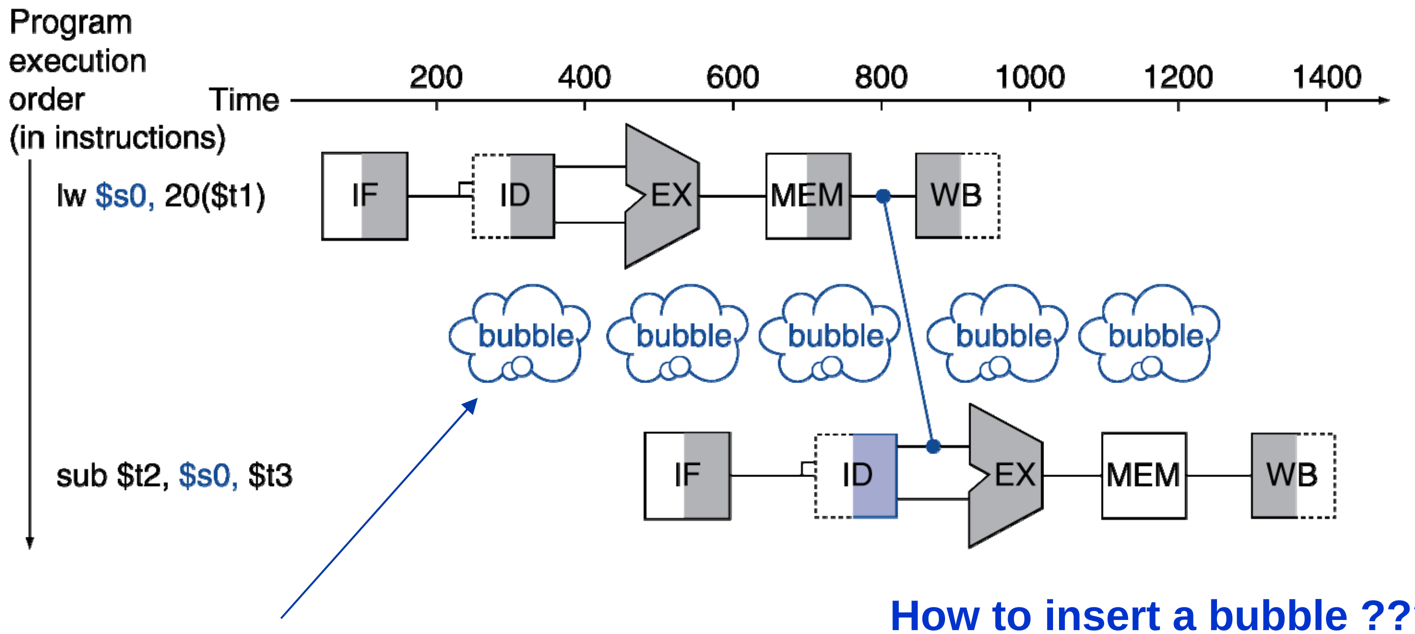
HW Change for Forwarding

Additional hardware is required.



Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!

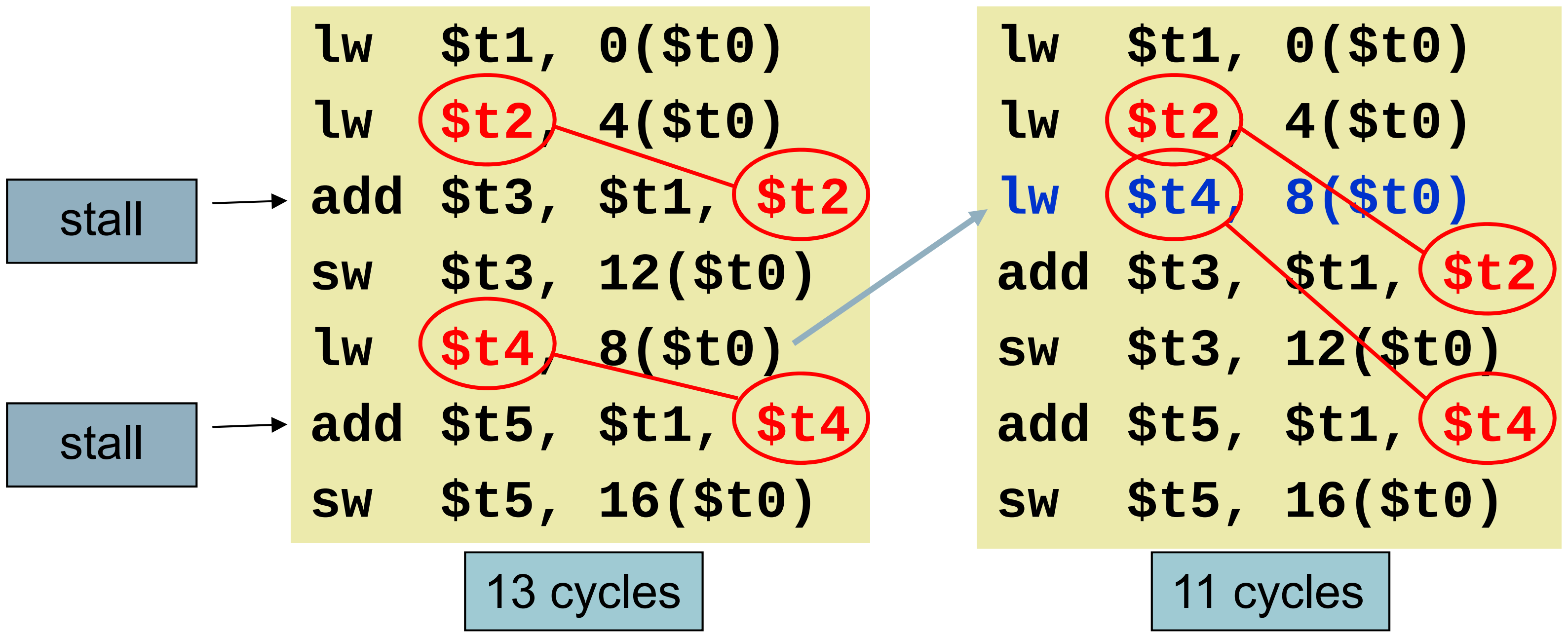


Software Check or Hardware Handling

How to insert a bubble ???

Rescheduling Code to Avoid Stalls

- Compiler reorders the code sequence to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`

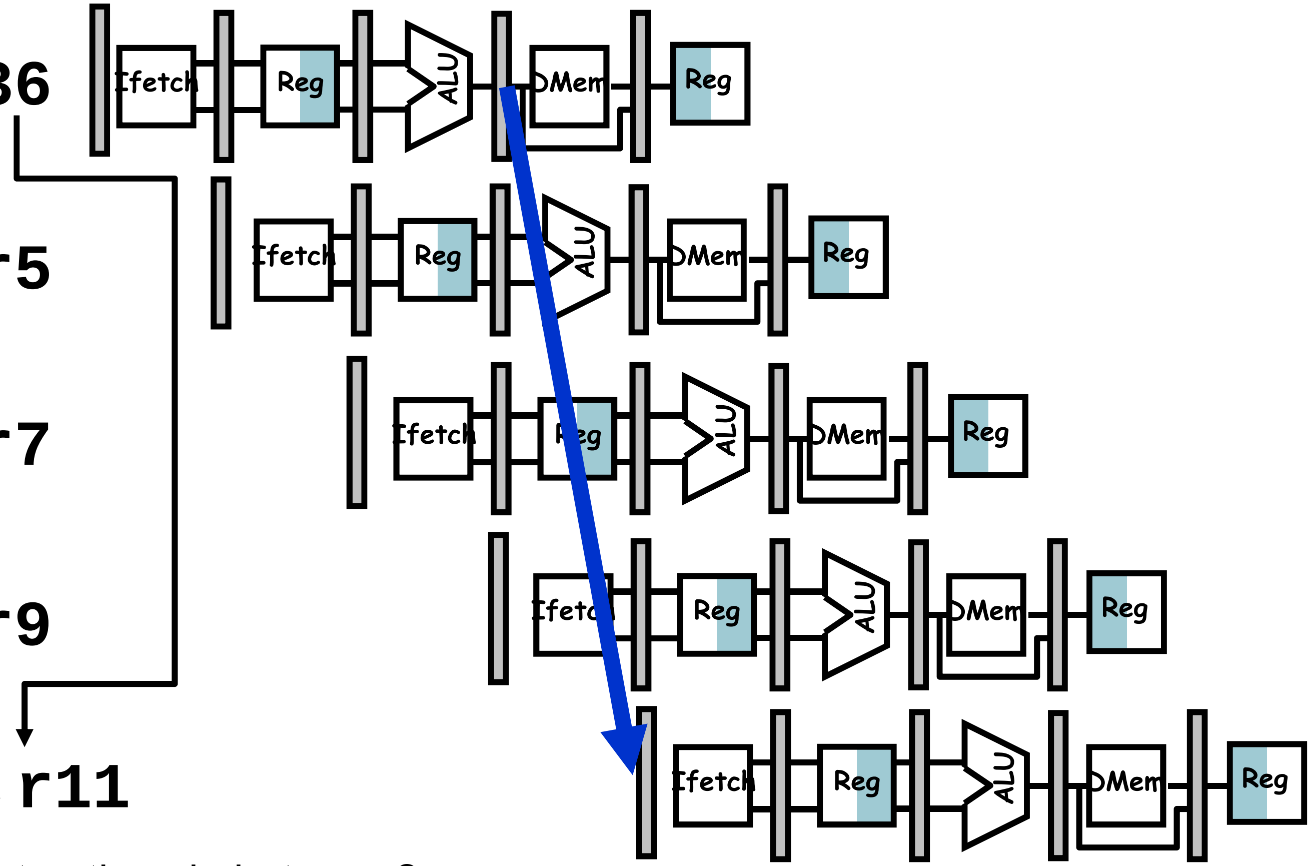


```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

Control Hazard on Branches

```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```



What do you do with the 3 instructions in between?

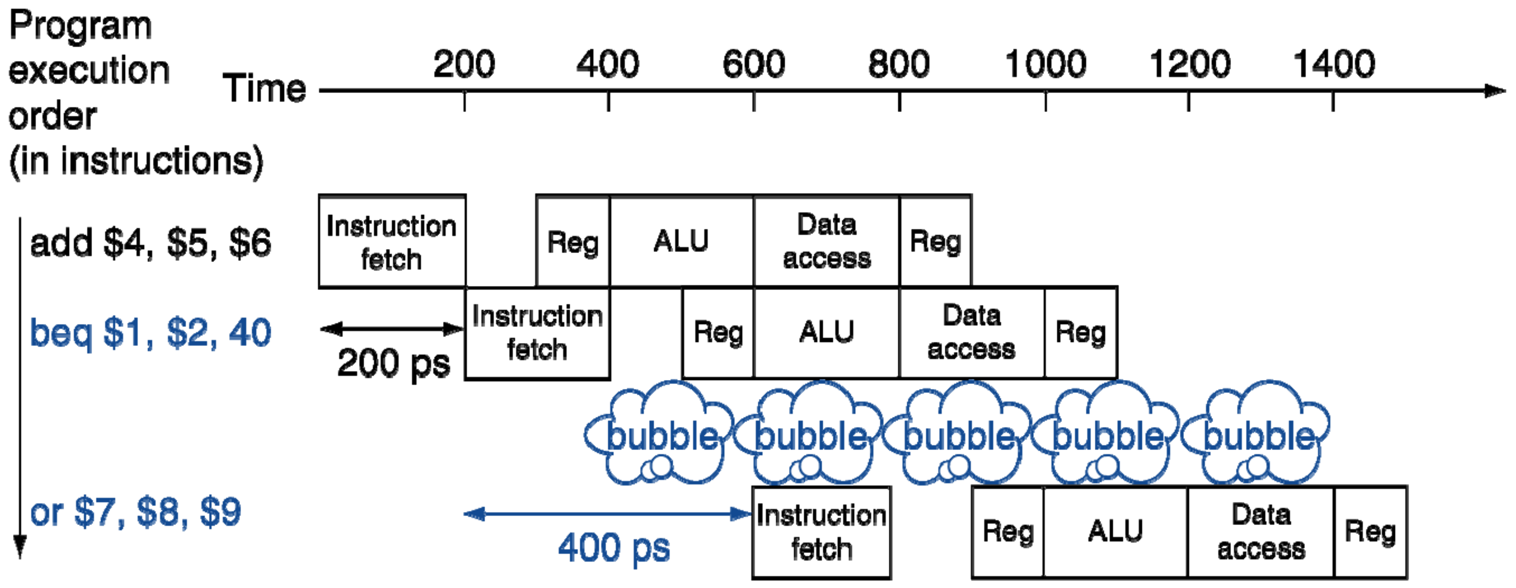
The simplest solution is to stall the pipeline as soon as a branch instruction is detected

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two-part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or $\neq$ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



How to add a stall cycle ?

Four Alternatives for Control Hazard

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- But haven't calculated branch target address in MIPS, it still incurs 1 cycle branch penalty
- Advantage of branch target is known before outcome

Four Alternatives for Control Hazard

#4: Delayed Branch – make the stall cycle useful

- Define branch to take place AFTER a following instruction

branch instruction
  sequential successor$_1$
  sequential successor$_2$
  ....
  sequential successor$_n$
branch target if taken

Branch delay of length *n*

These insts. are executed !!

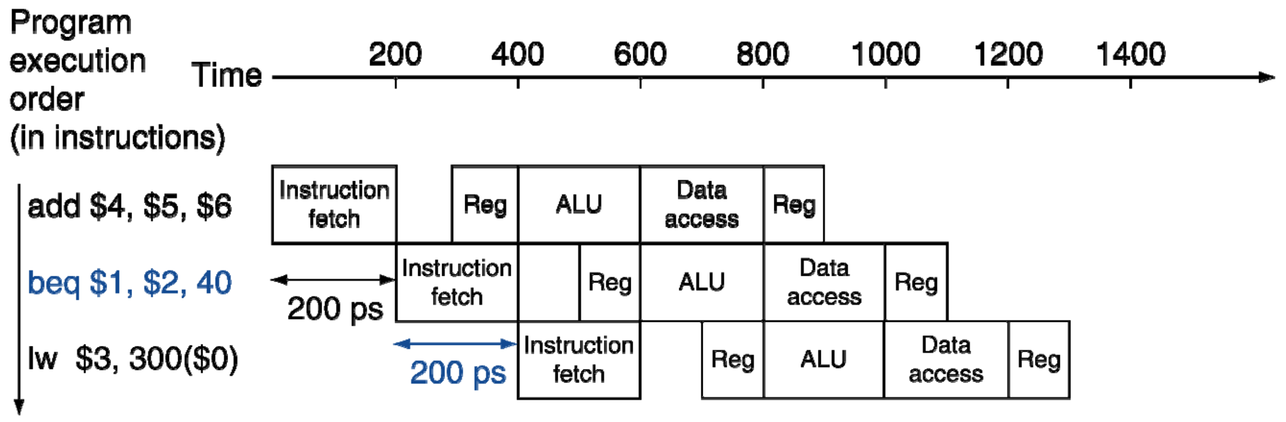
- 0-cycle latency, if all the stall cycles are useful

Handling Branch Hazard

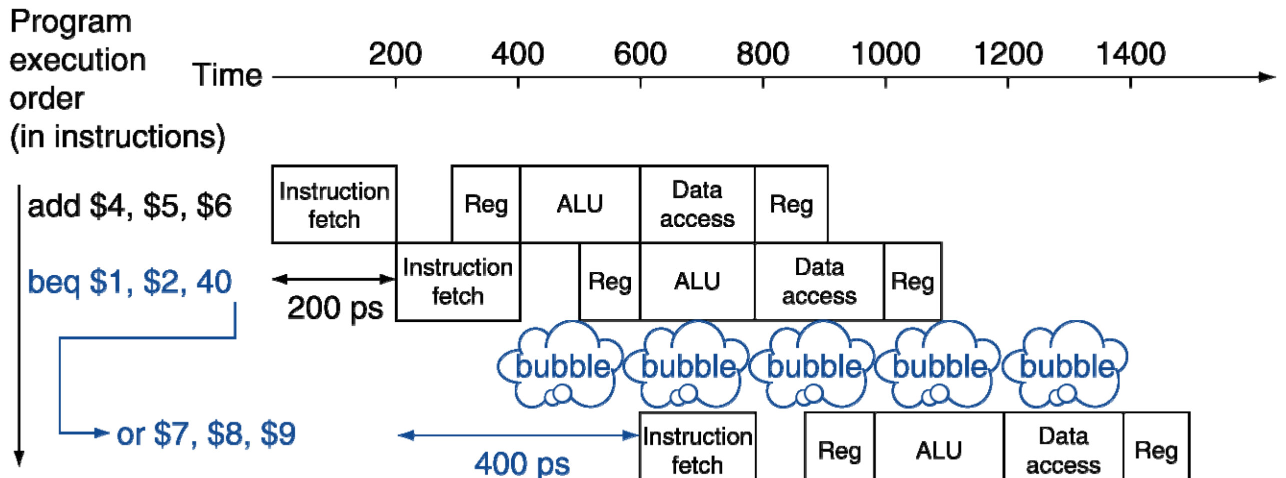
- Moving branch execution earlier in the pipeline
  - Move up branch address calculation to ID
  - Check branch equality at ID (using XOR for zero test) by comparing the two registers read during ID
  - Branch decision made at ID => one instruction to flush
  - Add a control signal, IF.Flush, to zero instruction field of IF/ID => making the instruction an NOP (i.e. bubble instruction)
- (Static) Predict branch always not taken
  - Need to add hardware for flushing inst. if wrong
- Compiler rescheduling and delay branch (discussed later)
- Dynamic branch prediction (discussed later)

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

- Static n-bit branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Pipeline Operation

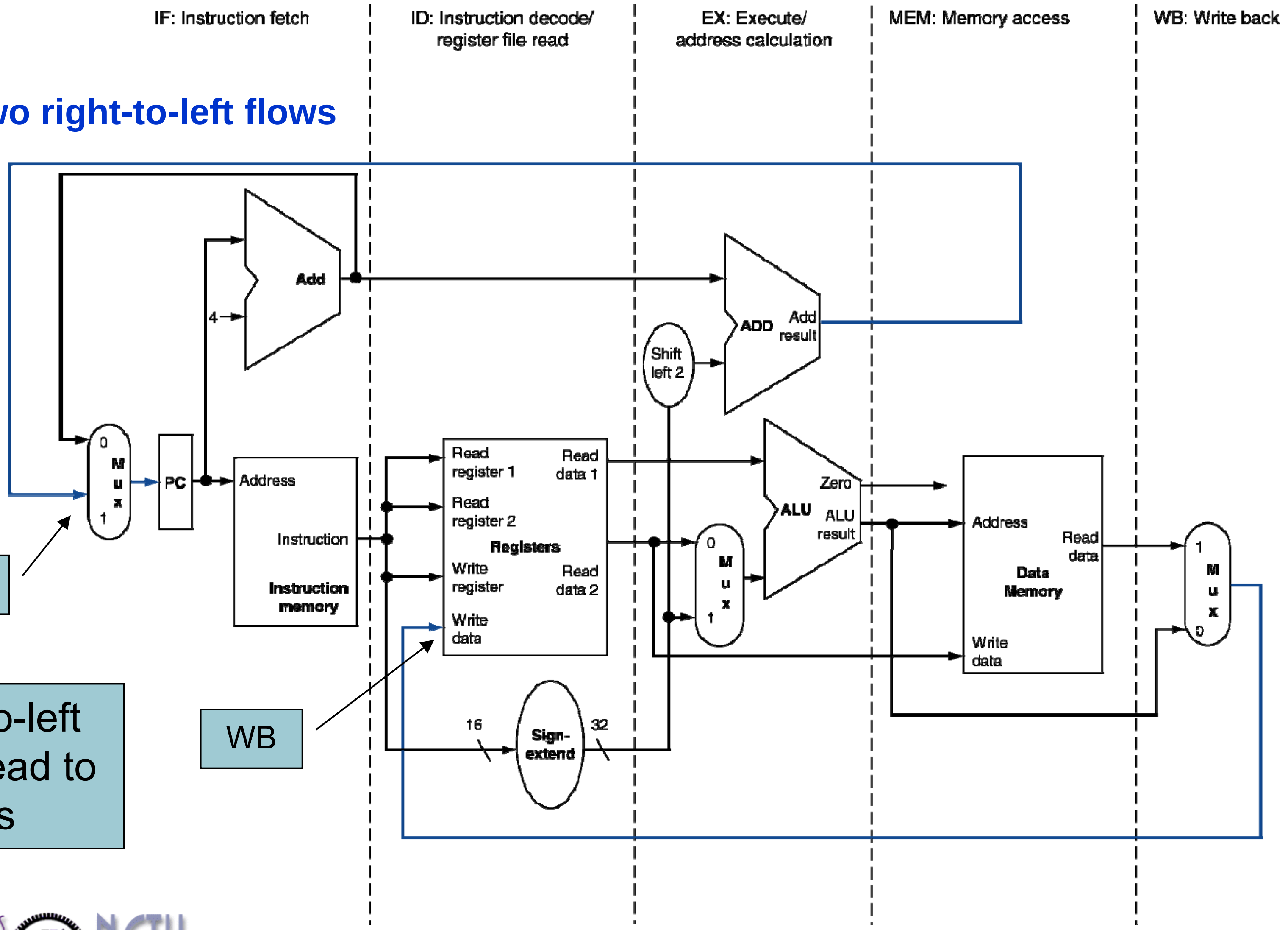
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time

Recall: Steps for Designing a Pipelined Processor

- Examine the datapath and control diagram
  - Starting with single cycle datapath
- Partition datapath into stages:
  - IF (instruction fetch), ID (instruction decode and register file read), EX (execution or address calculation), MEM (data memory access), WB (write back)
- Associate resources with stages
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

MIPS Single-Cycle Datapath

Two right-to-left flows



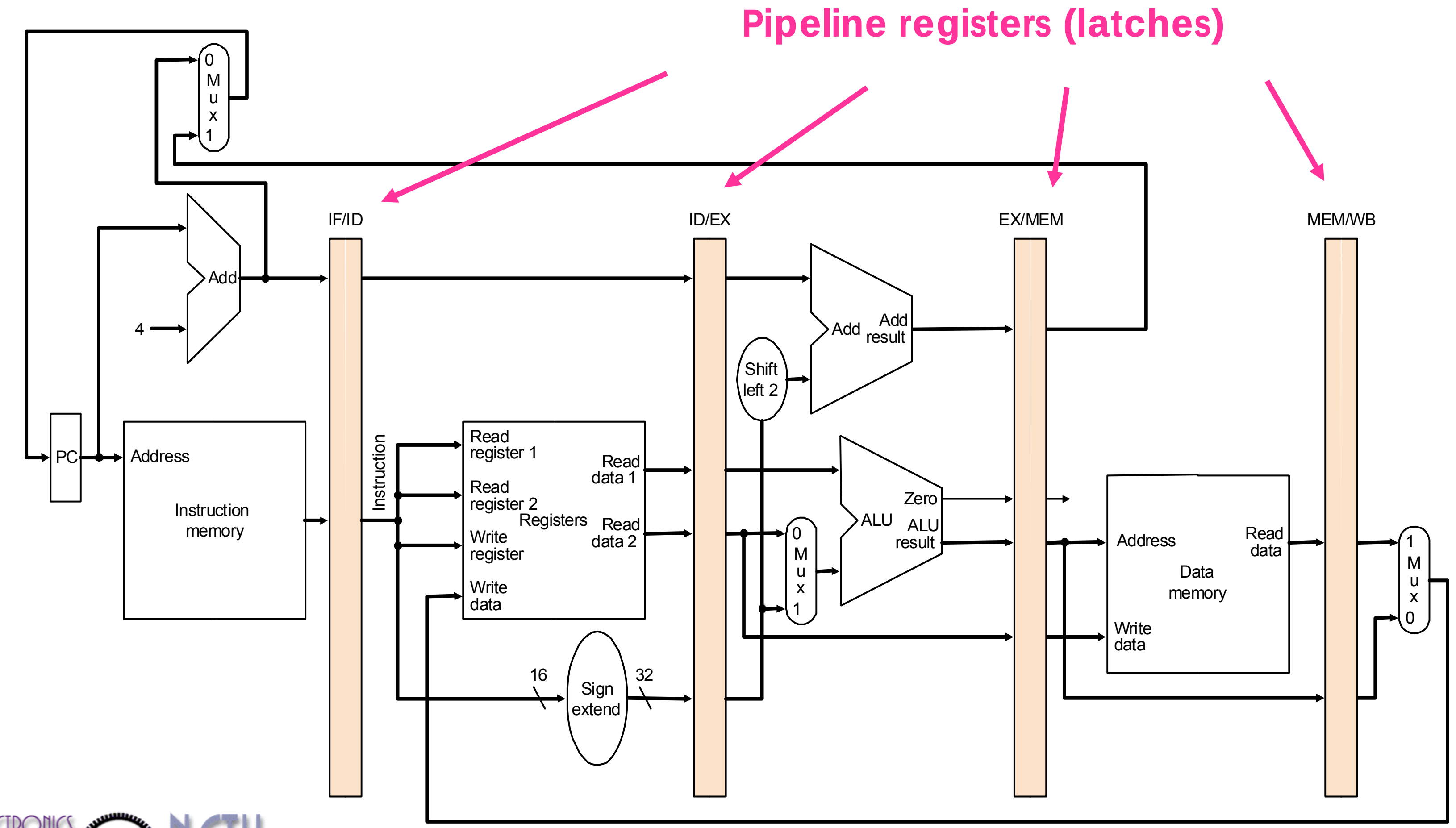
MEM

WB

Right-to-left flows lead to hazards

Pipeline Registers

Use registers between stages to carry data and control



MIPS ISA Micro-Operations

One way to show what happens in pipelined execution

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

Instruction fetch	IR = Memory[PC]
	PC = PC + 4			

Instruction decode/register fetch	A = Reg [IR[25-21]]
	B = Reg [IR[20-16]]
	ALUOut = PC + (sign-extend (IR[15-0]) << 2)			

Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

MIPS ISA Micro-Operations

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

Instruction fetch	IR = Memory[PC]
	PC = PC + 4			

Instruction decode & register fetch	A = Reg [IR[25-21]]; B = Reg [IR[20-16]]
	if (A ==B) then, ALUOut = PC + (sign-extend (IR[15-0]) << 2)			

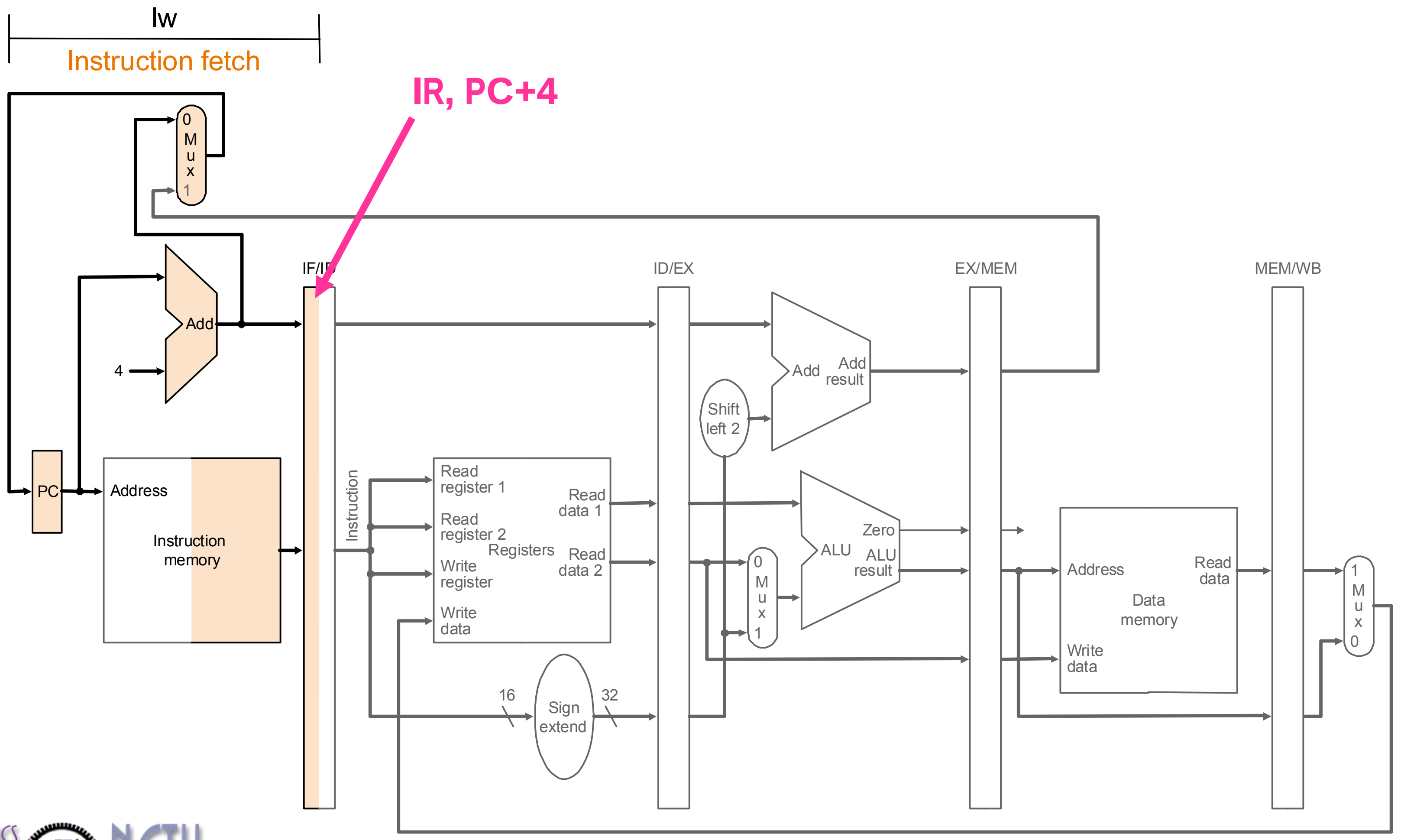
Execution/ address computation	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])		PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion		Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion/ R-type completion	Reg [IR[15-11]] = ALUOut	Load: Reg[IR[20-16]] = MDR		

Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - Shows pipeline usage in a single cycle (stage)
  - Highlight resources used
- We'll look at "single-clock-cycle" diagrams for load instruction

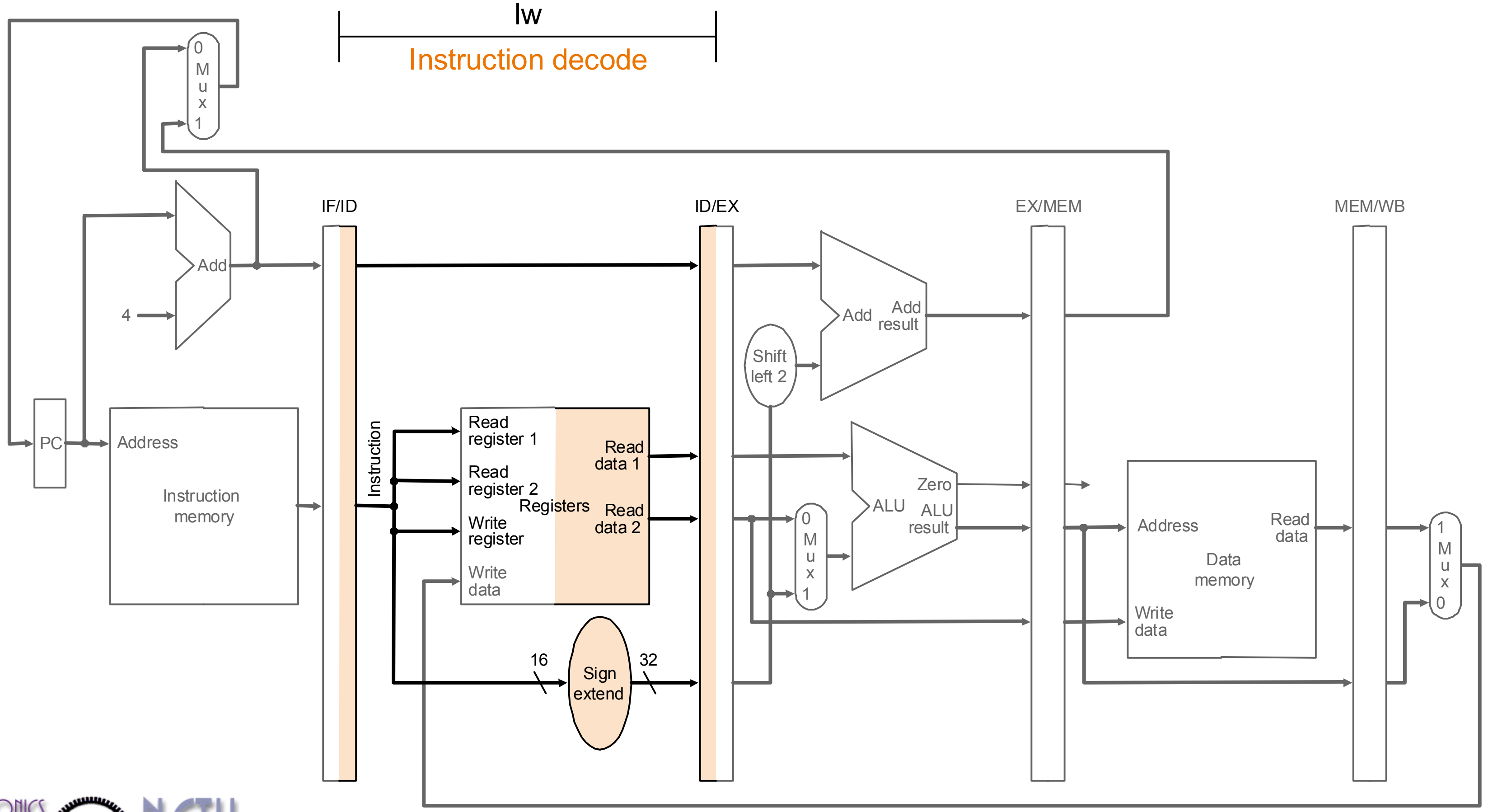
IF Stage of `lw`

- Ex: `lw rt,rs,imm16`



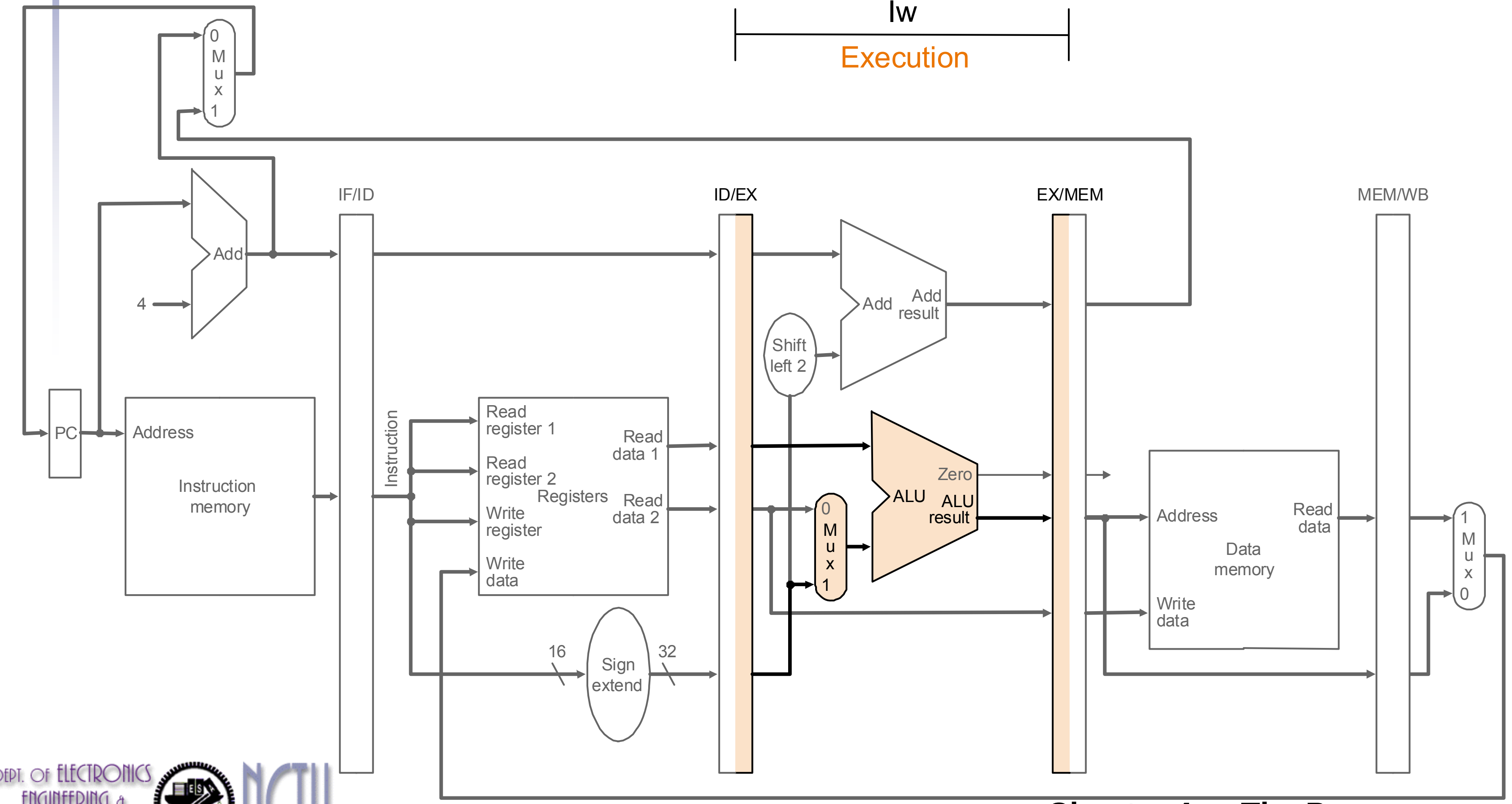
ID Stage of `lw`

- Ex: `lw rt,rs,imm16`
- `A = Reg[IR[25-21]];`



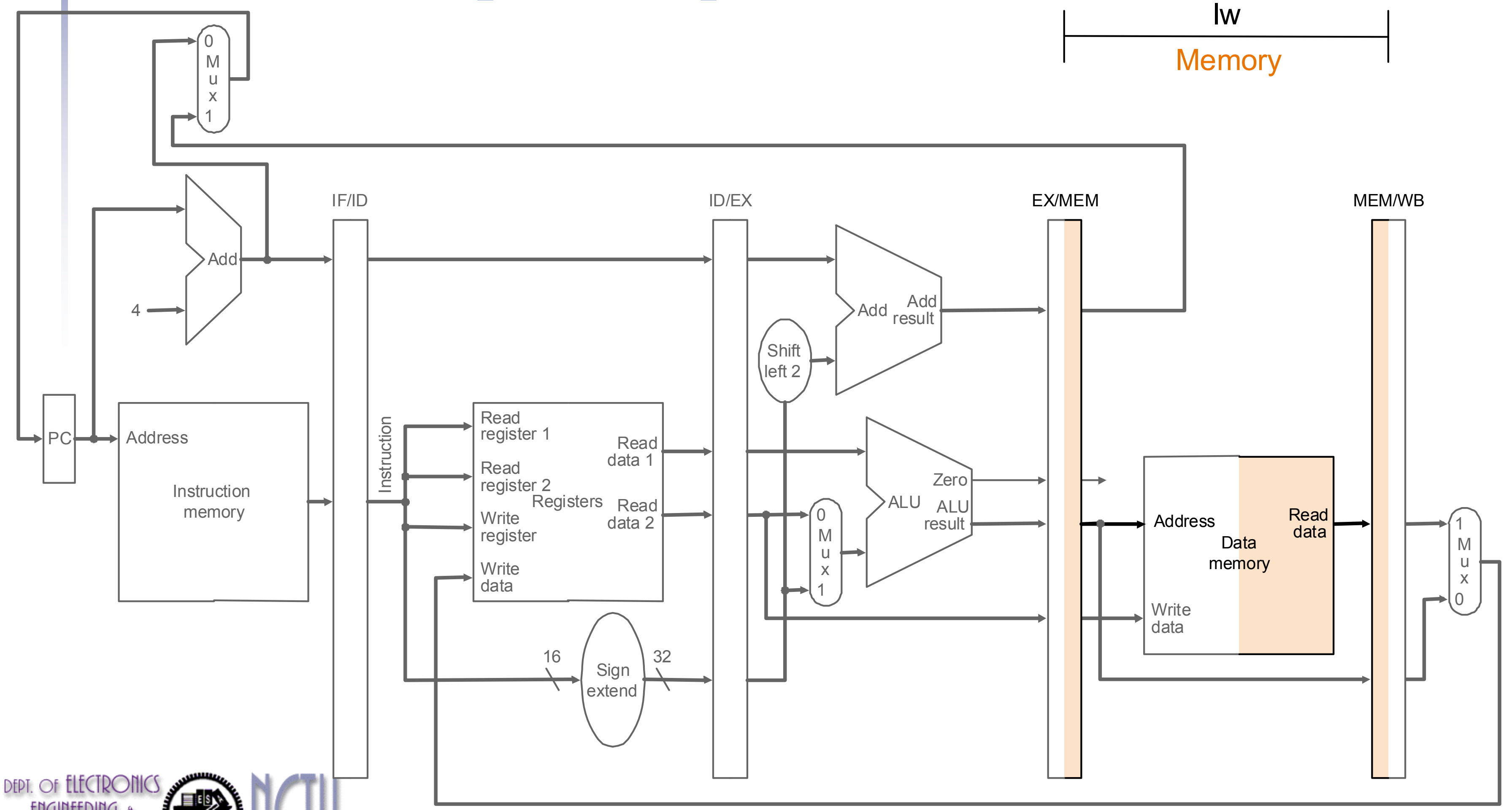
EX Stage of `lw`

- Ex: `lw rt,rs,imm16`
- `ALUout = A + sign-ext(IR[15-0])`



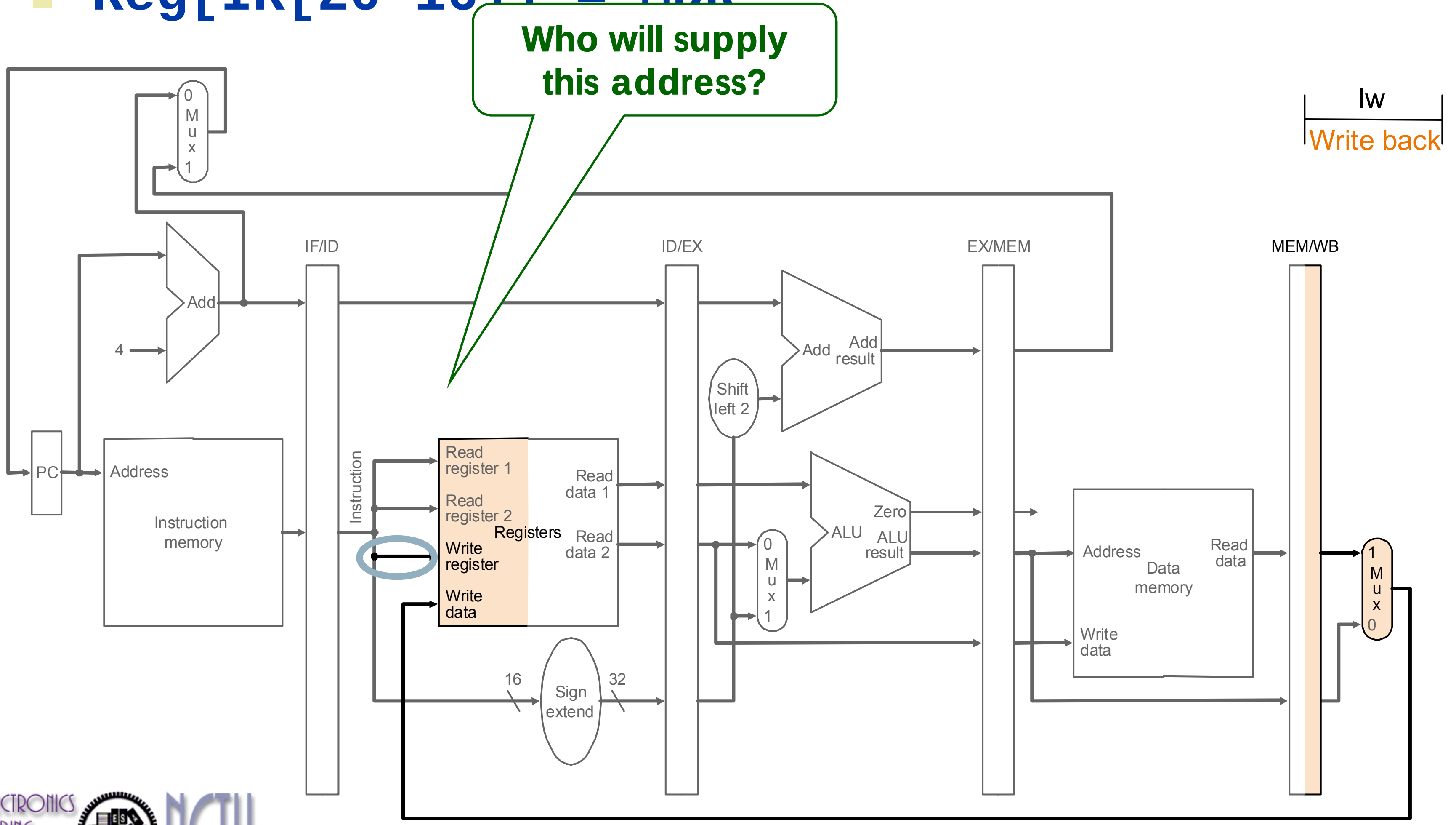
MEM State of `lw`

- Ex: `lw rt,rs,imm16`
- `MDR = mem[ALUout]`

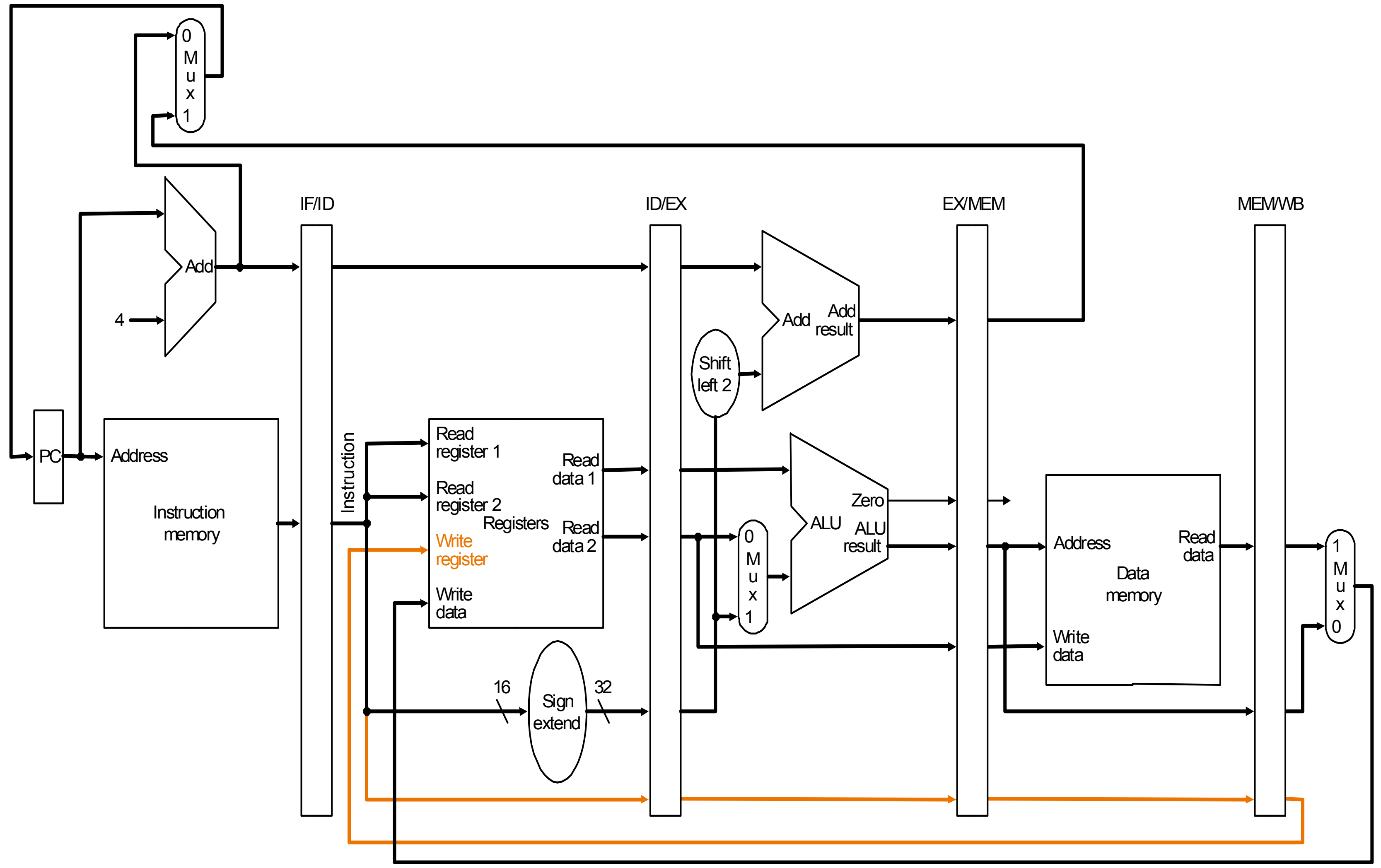


WB Stage of `lw`

- Ex: `lw rt,rs,imm16`
- `Reg[IR[20-16]] = MDR`

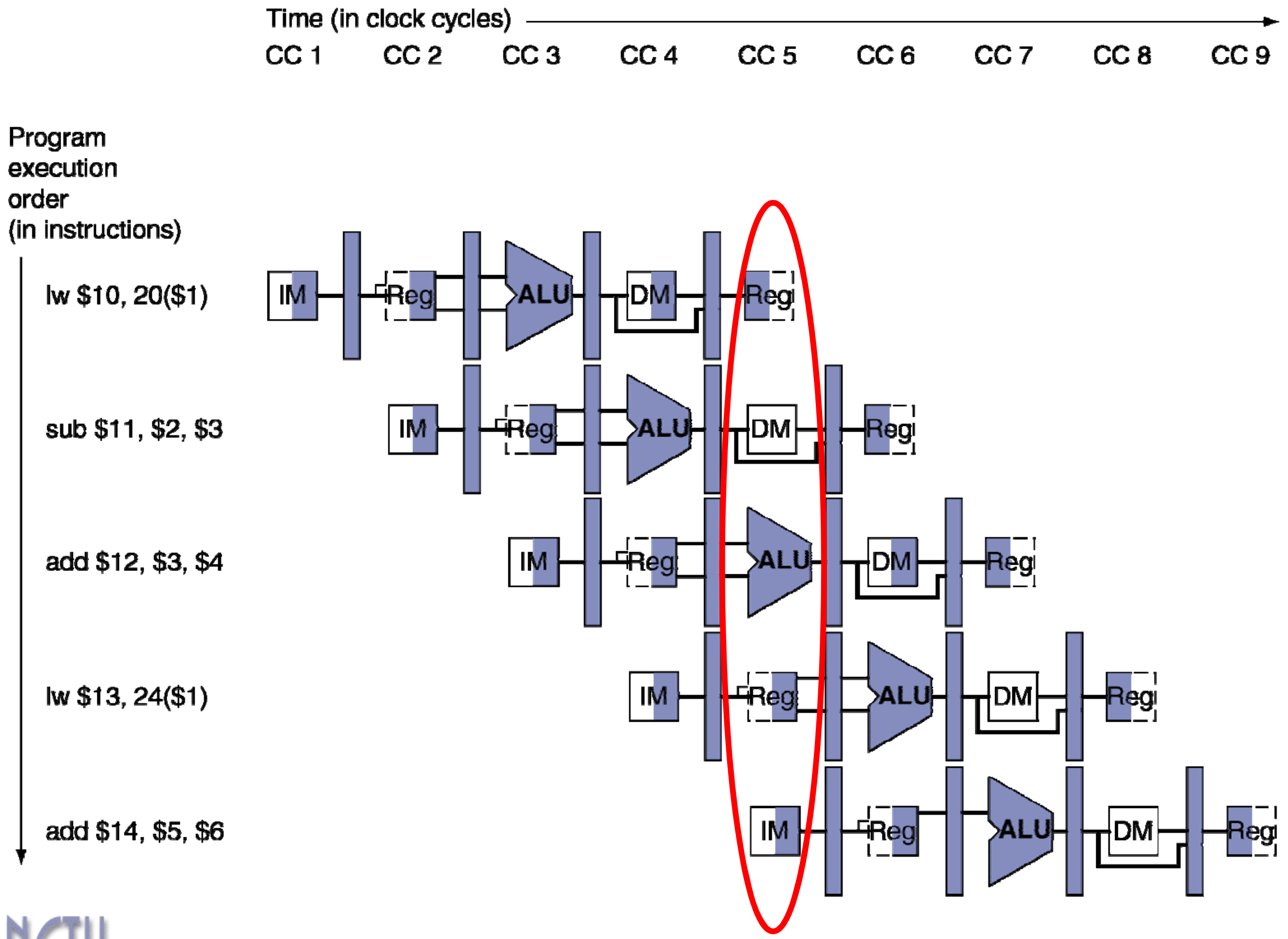


Pipelined Datapath



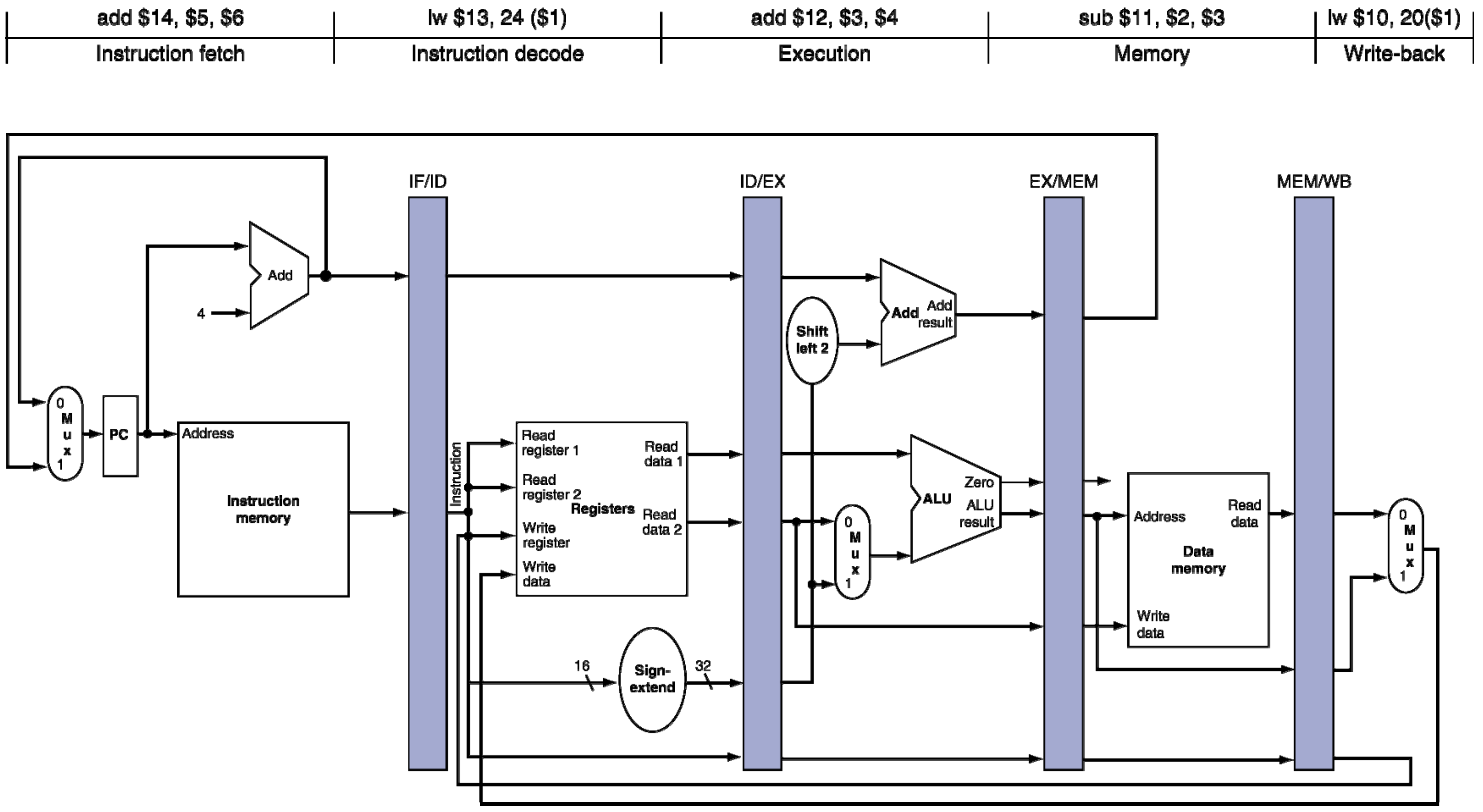
Pipeline Diagram

- Multi-clock-cycle pipeline diagram



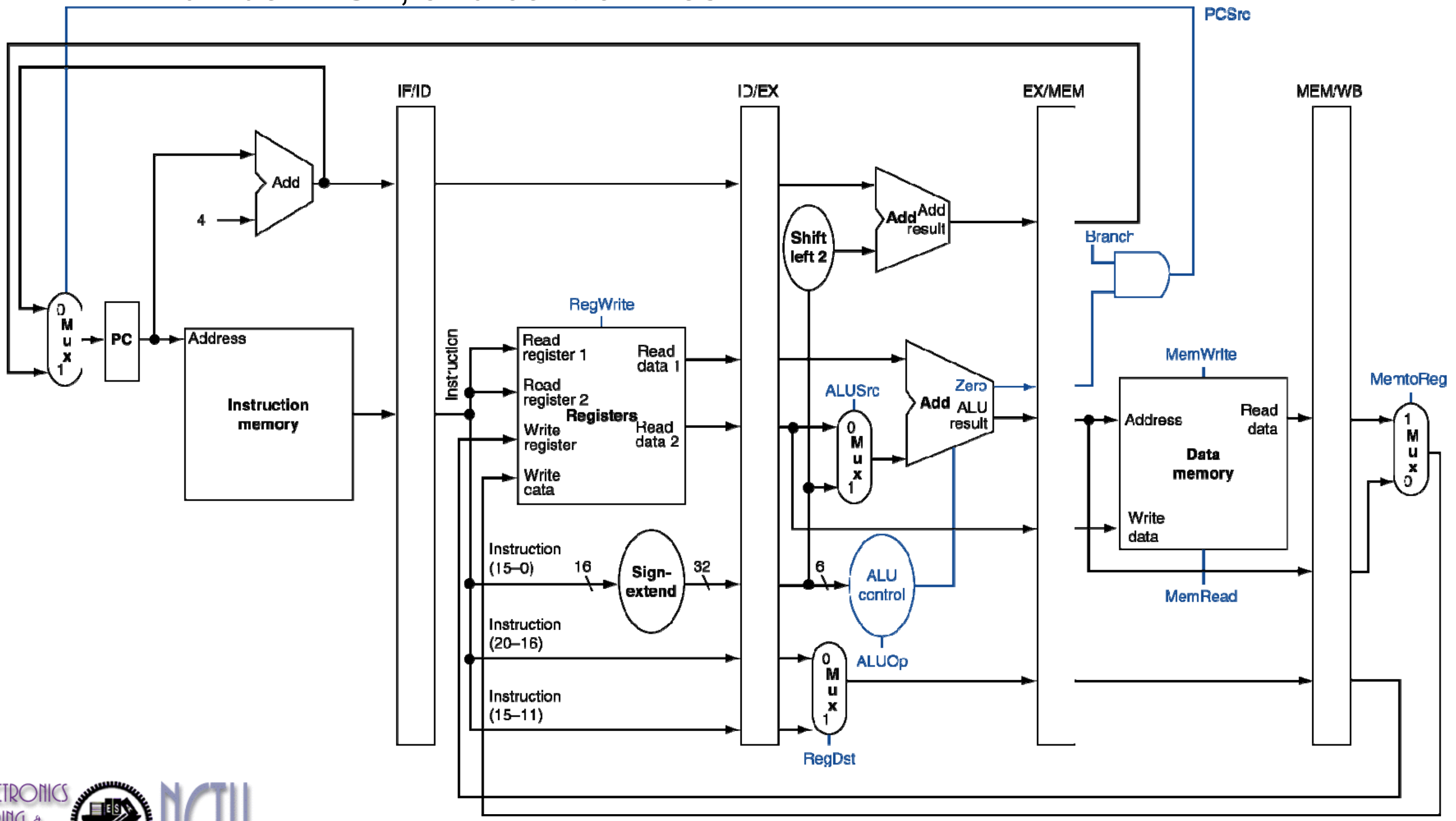
Pipeline diagram

- Single-clock-cycle diagram in a given cycle cc5



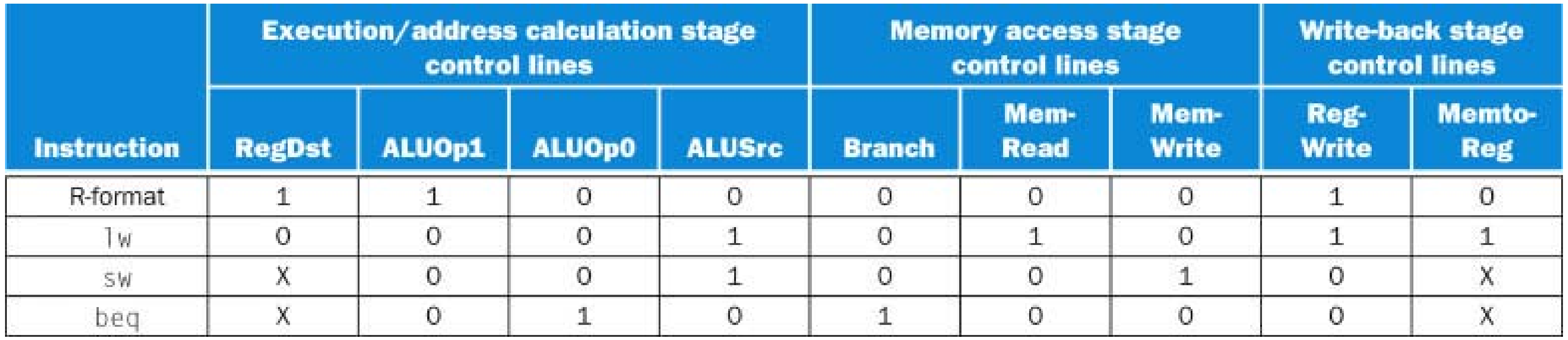
Pipelined Control (1)

- Start with the (simplified) single-cycle datapath
  - Use the same ALU control logic, branch logic, destination-register-number MUX, and control lines



Pipelined Control (2)

- To specify control for the pipeline, we need to set control values during each pipeline stage.

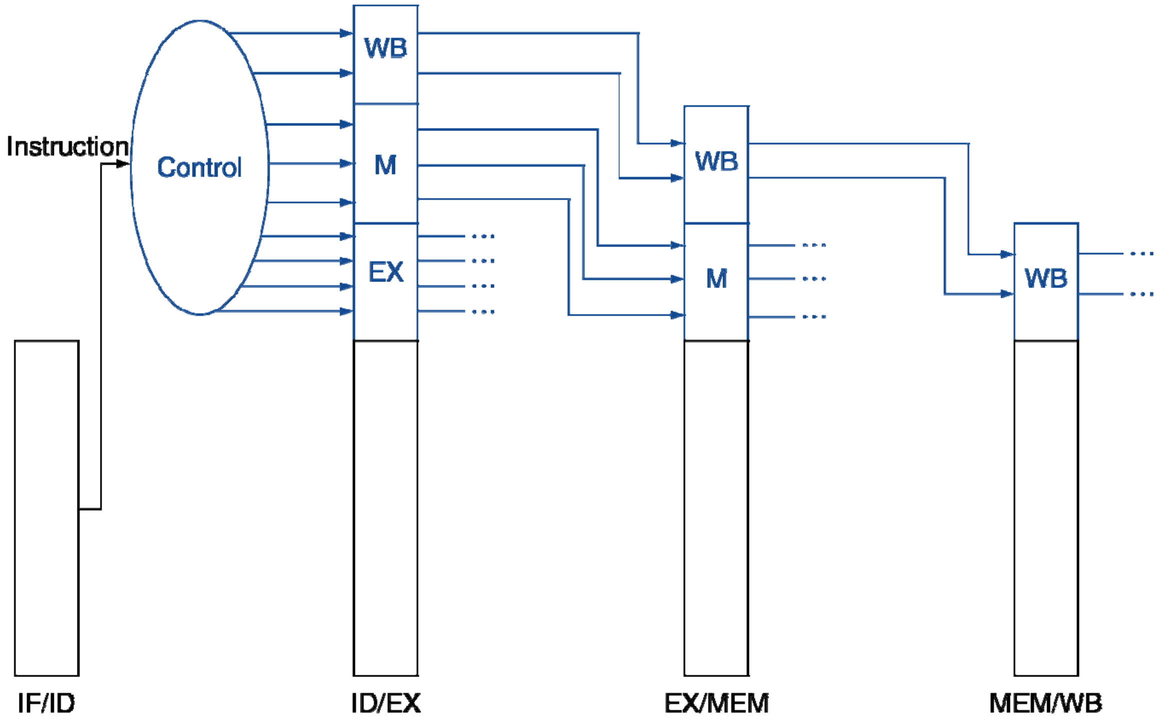


Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

- The simplest implementation way is data stationary pipelined control: to extend the pipeline registers to include control information

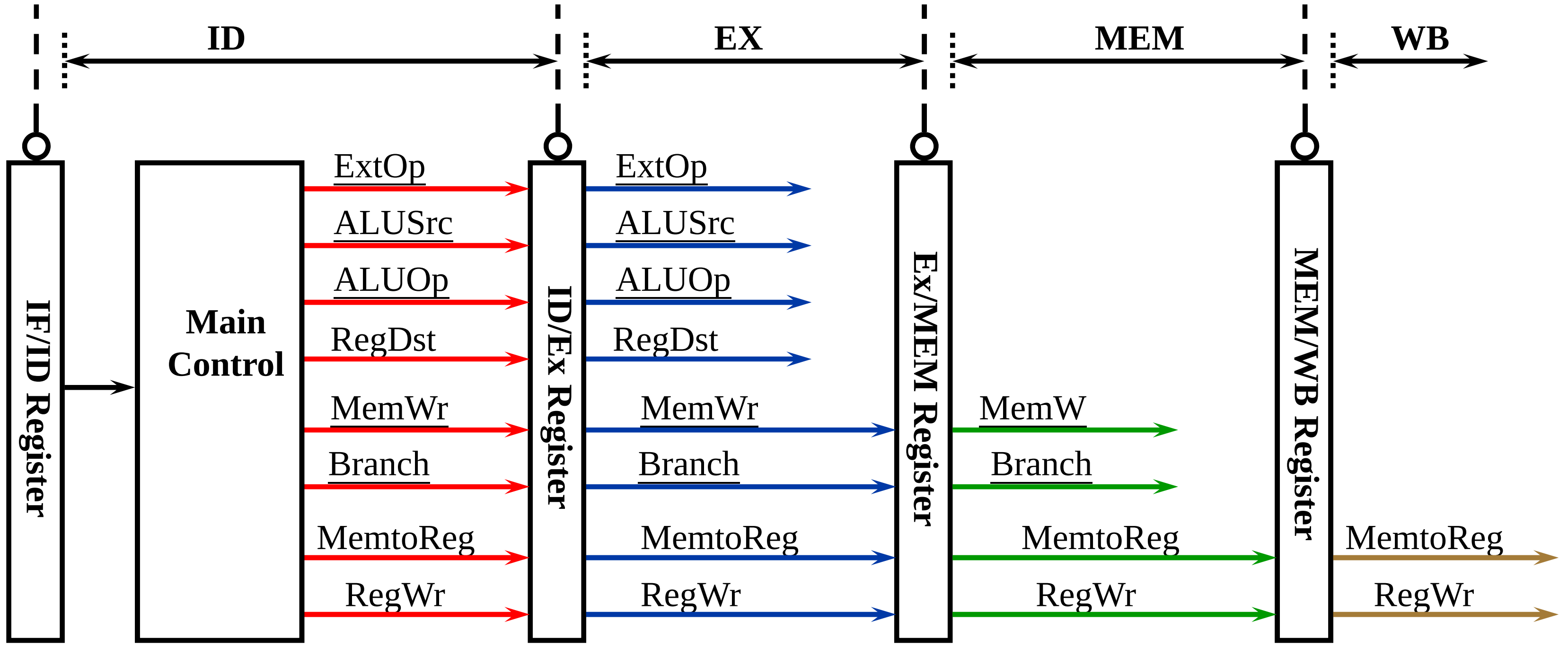
Data Stationary Pipelined Control

- Control signals derived from instruction
  - Main control generates control signals during ID
  - Pass control signals along just like the data

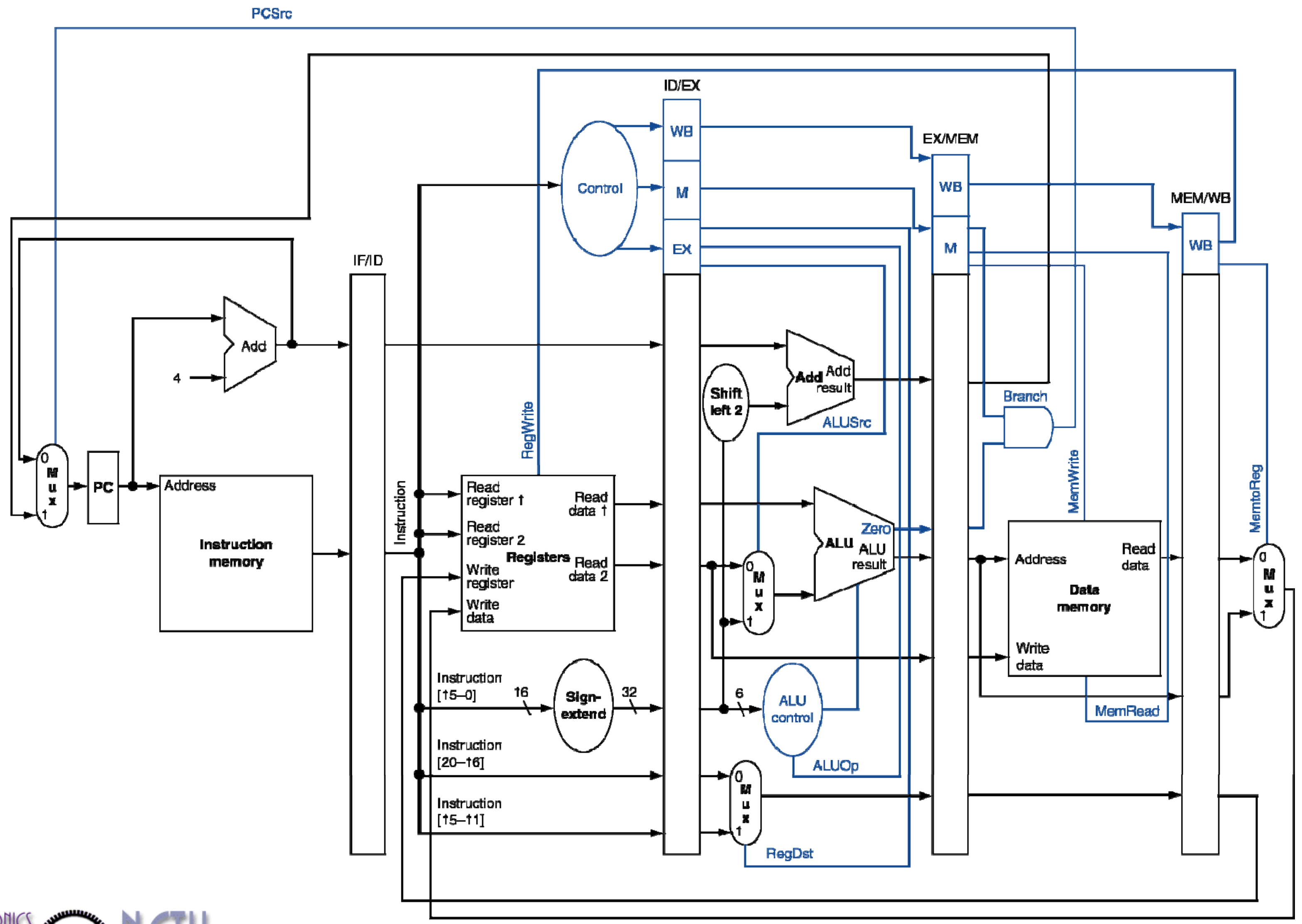


Data Stationary Control

- Signals for EX (ExtOp, ALUSrc, ...) are used 1 cycle later
- Signals for MEM (MemWr, Branch) are used 2 cycles later
- Signals for WB (MemtoReg, MemWr) are used 3 cycles later



Pipelined Control



Hazard Detection

- We can resolve hazards with forwarding, but how do we detect when to forward?
  - RAW (WAR, WAW) dependence check
    - i.e. to compare register number between instructions
  - Pass register numbers along pipeline
    - ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
    - e.g. ALU operand register numbers in EX stage are given by ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when

  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

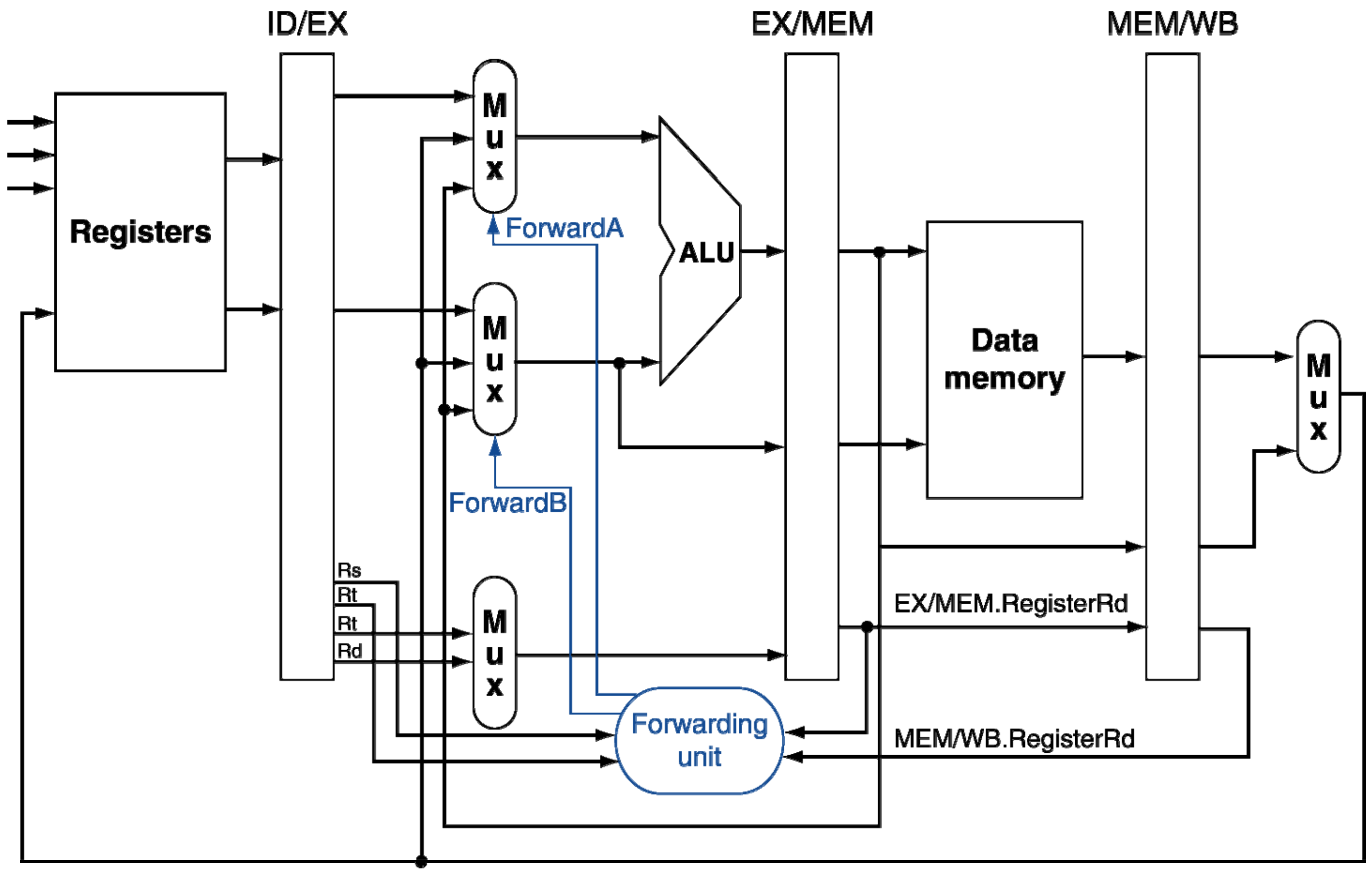
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Detecting the Need to Forward

- Not always do fordward if true data hazard
  - But only if forwarding instruction will write to a register!
    - EX/MEM.RegWrite, MEM/WB.RegWrite
  - And only if Rd for that instruction is not $zero
    - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

Forwarding Paths



b. With forwarding

Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Double Data Hazard

- Consider the sequence:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

- Both hazards occur ➔ Want to use the most recent
- Revise MEM hazard condition ➔ Only forward if EX hazard condition isn't true

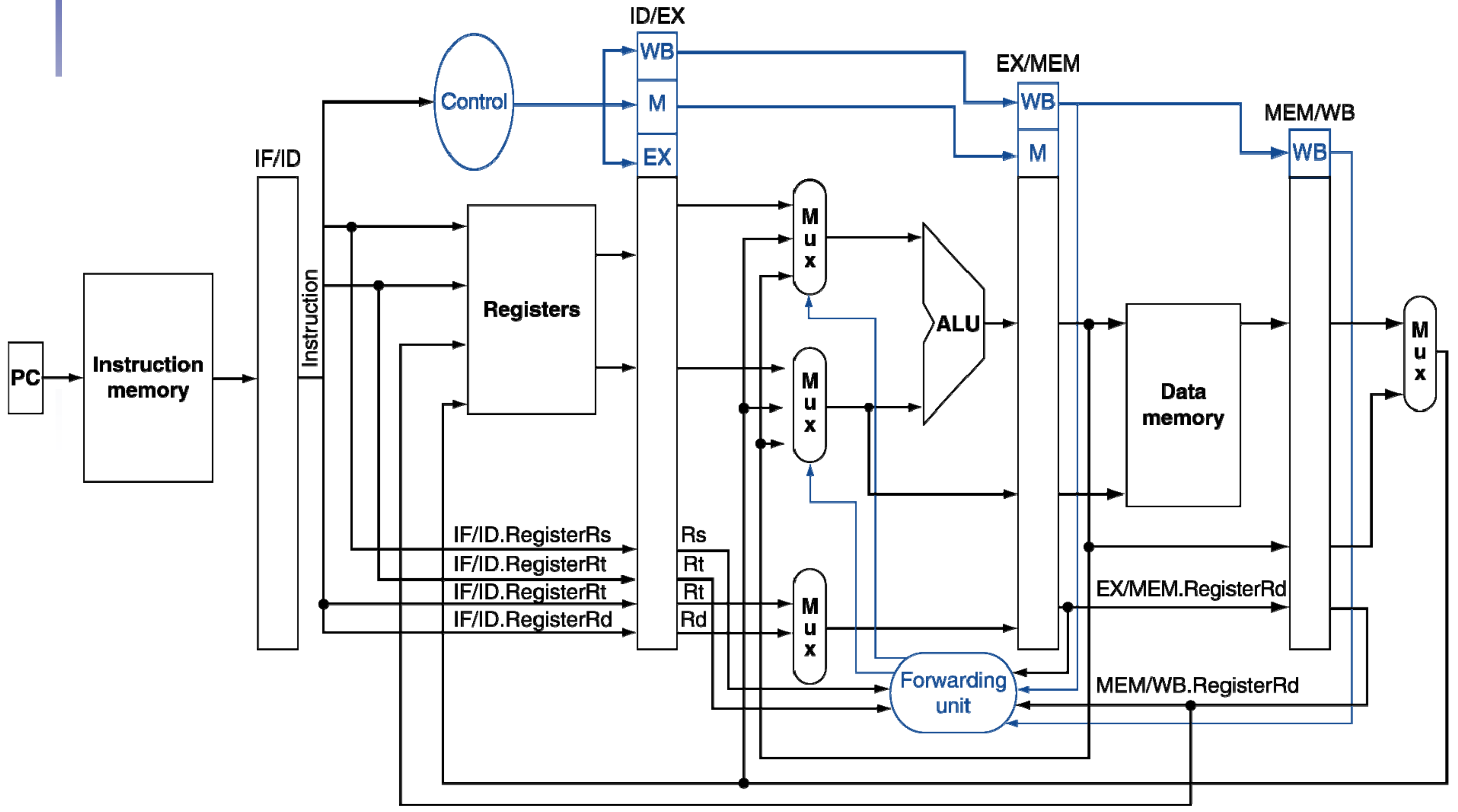
Revised Forwarding Condition

- MEM hazard

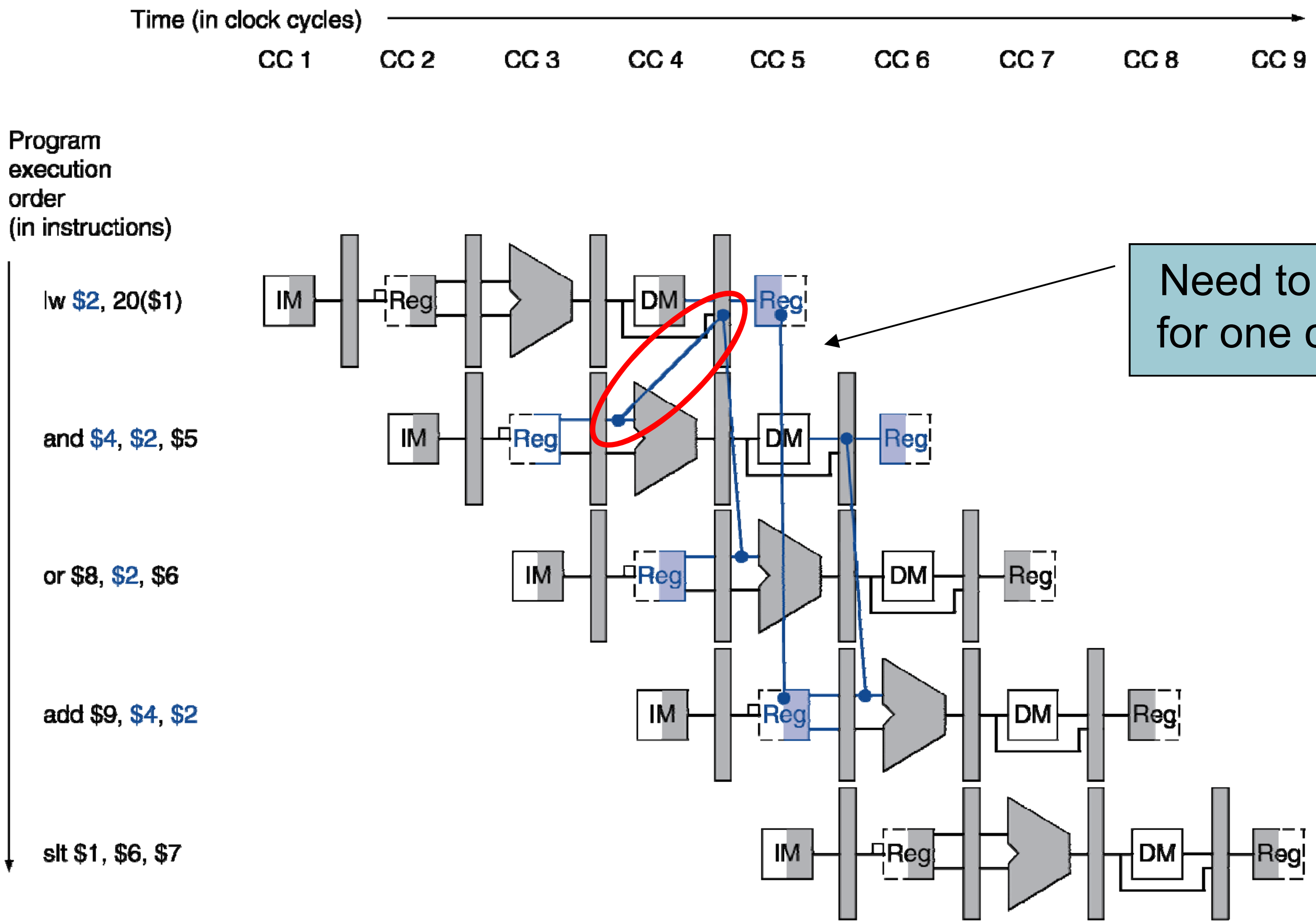
  -
    ```
    if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
                  and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
        and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01
    ```

  -
    ```
    if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
                  and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
        and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
    ```

Datapath with Forwarding



Load-Use Data Hazard



Need to stall for one cycle

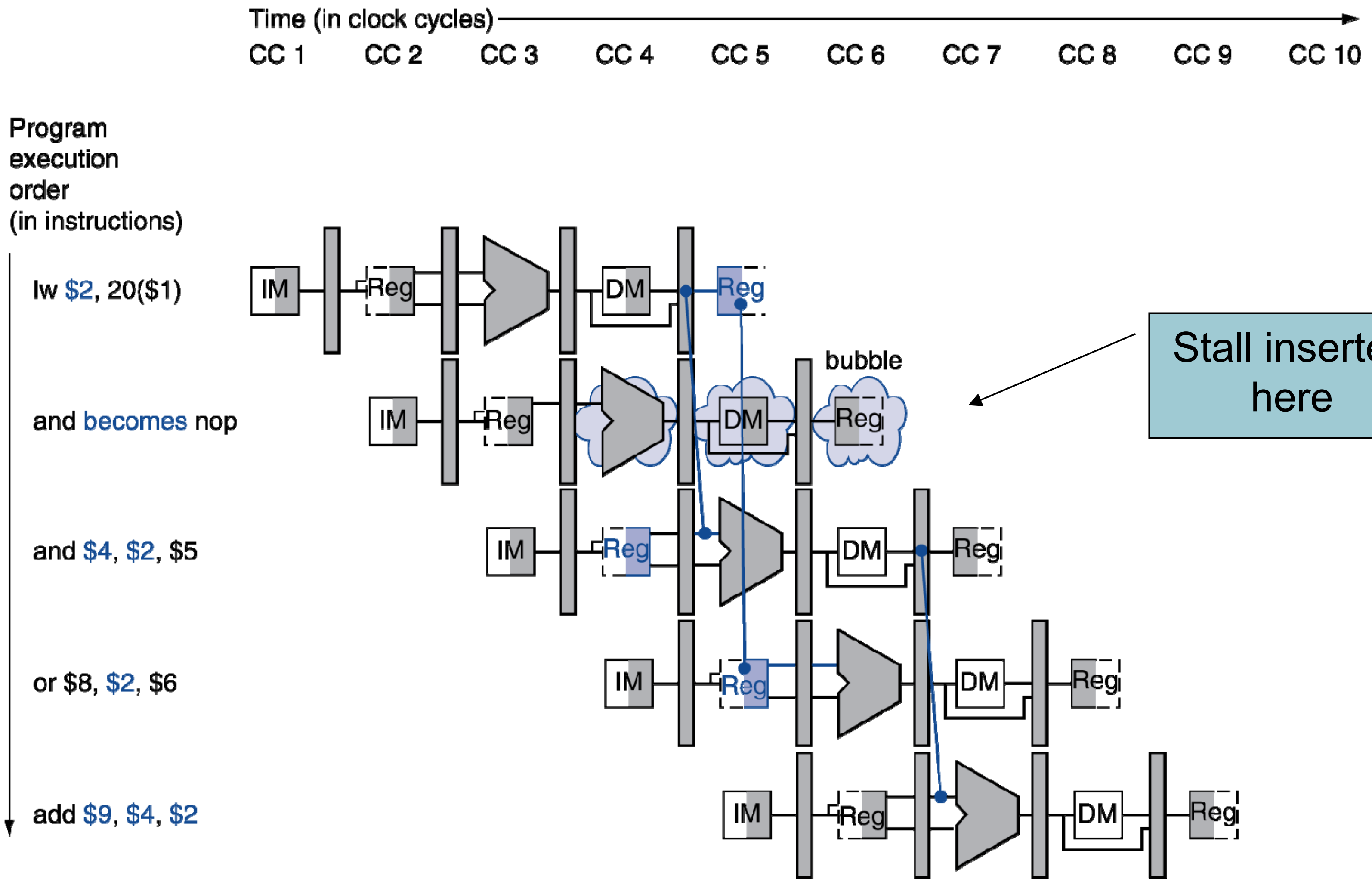
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

How to Stall the Pipeline

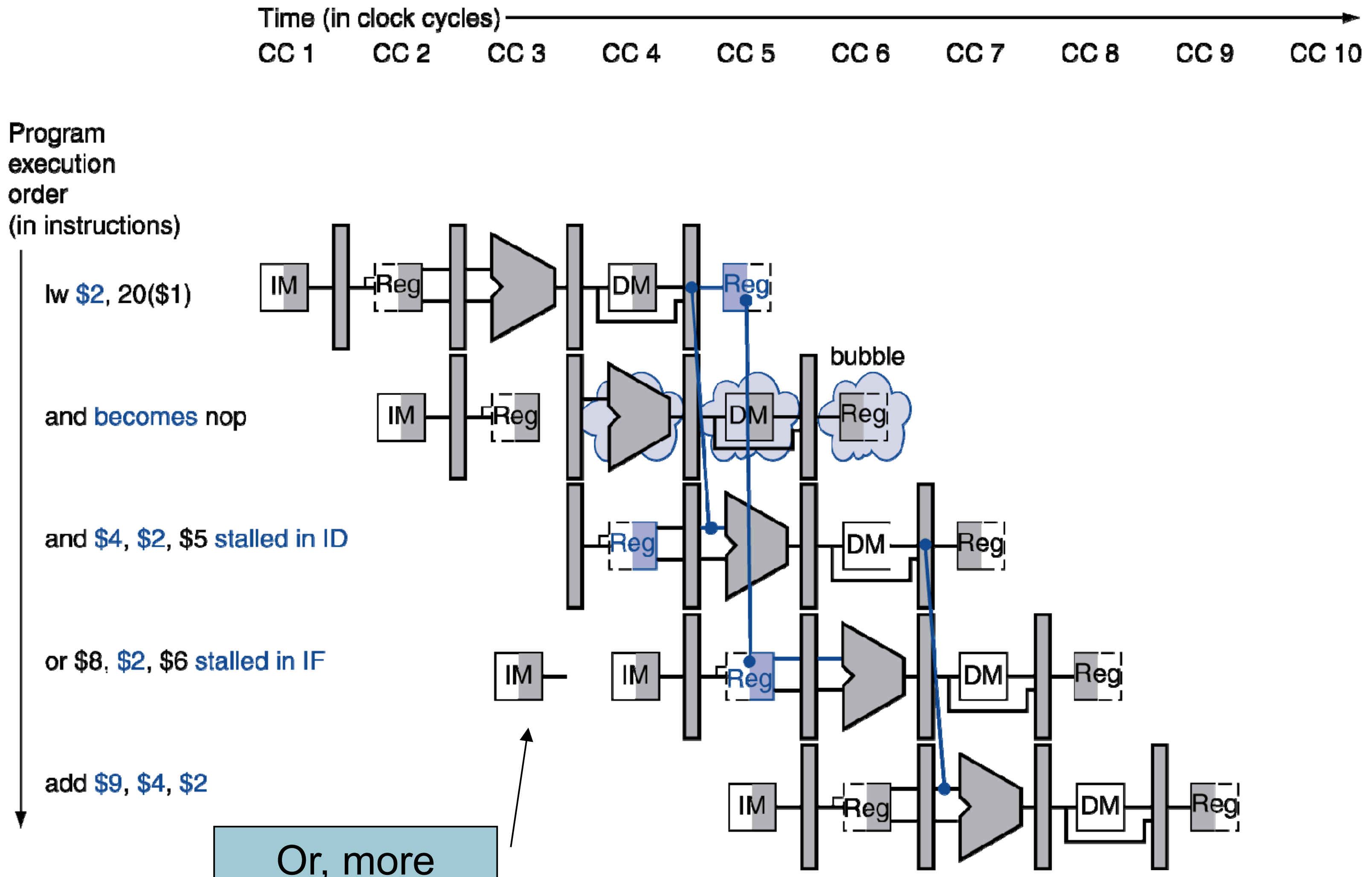
- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for `lw`
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline



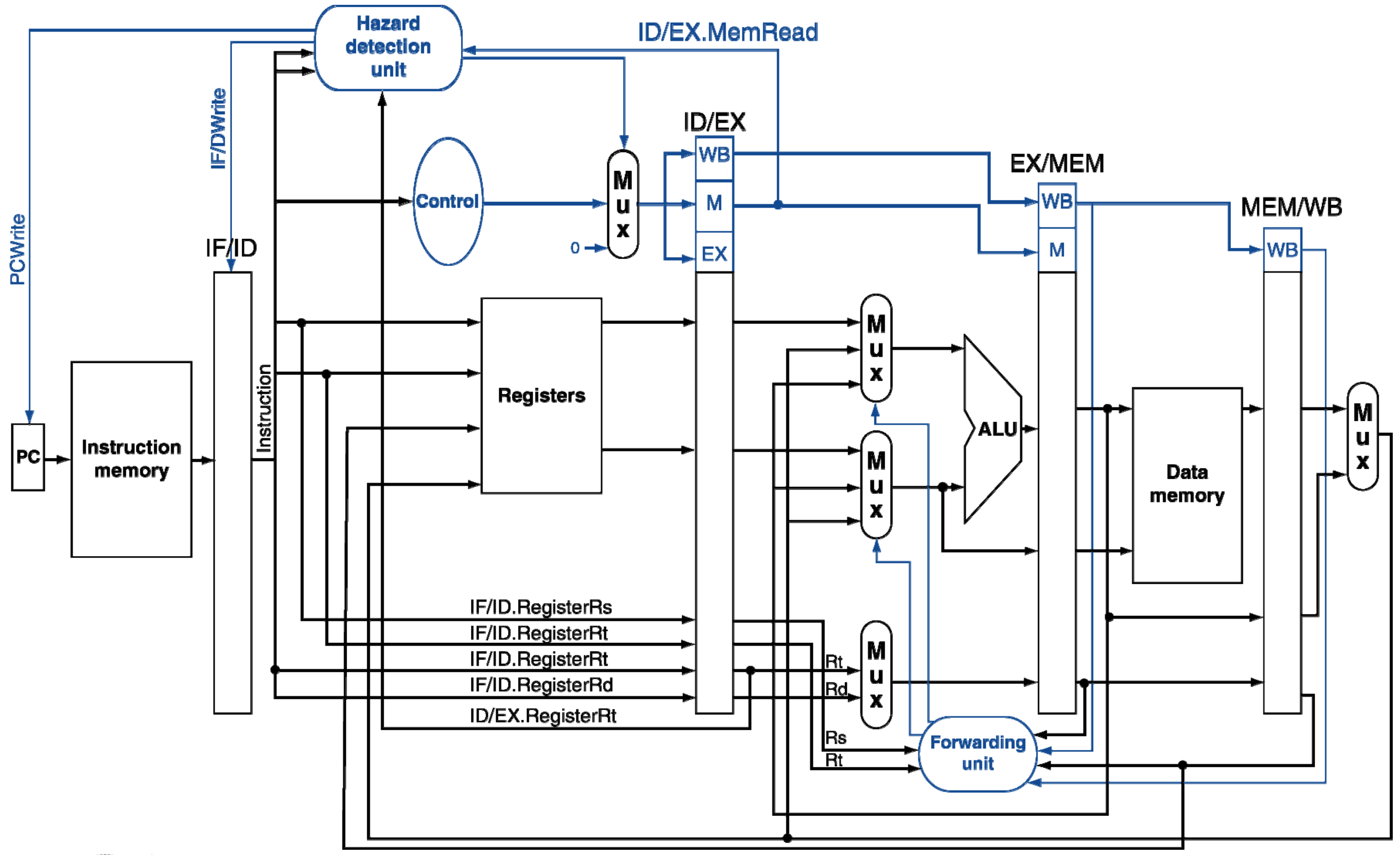
Stall inserted here

Stall/Bubble in the Pipeline



Or, more accurately…

Datapath with Hazard Detection



DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

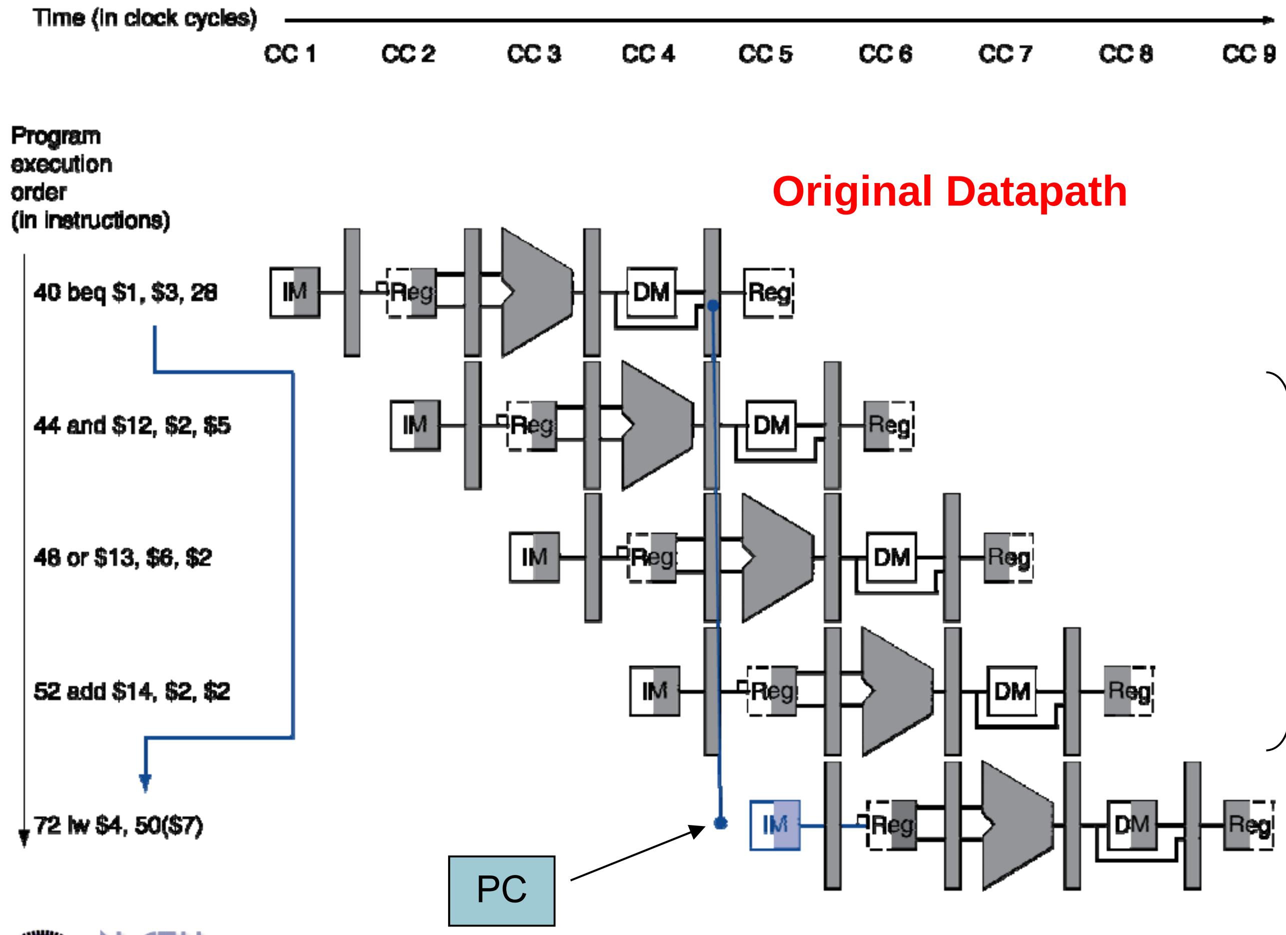
Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Branch Hazards

- If branch outcome determined in MEM

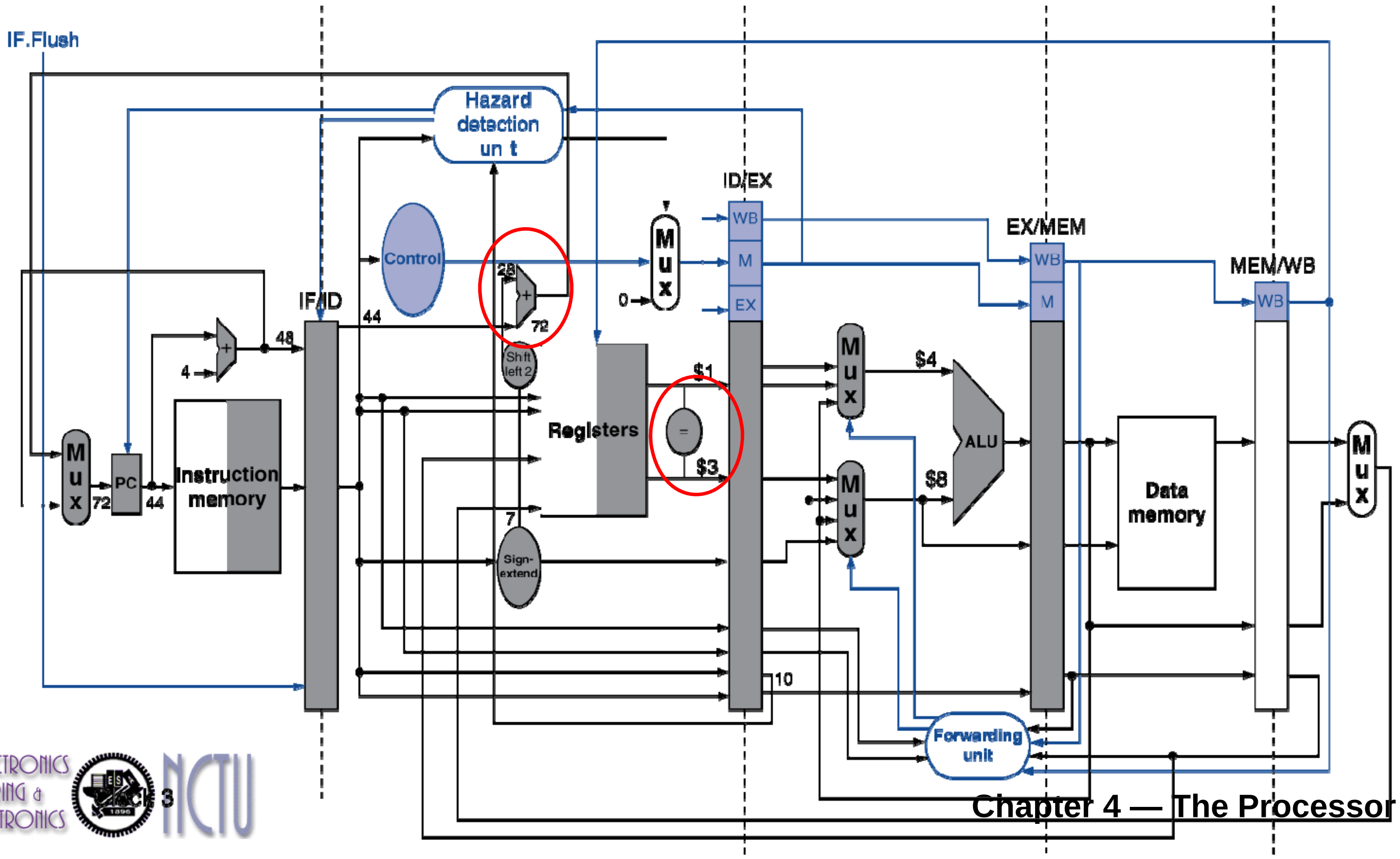


Flush these instructions (Set control values to 0)

Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

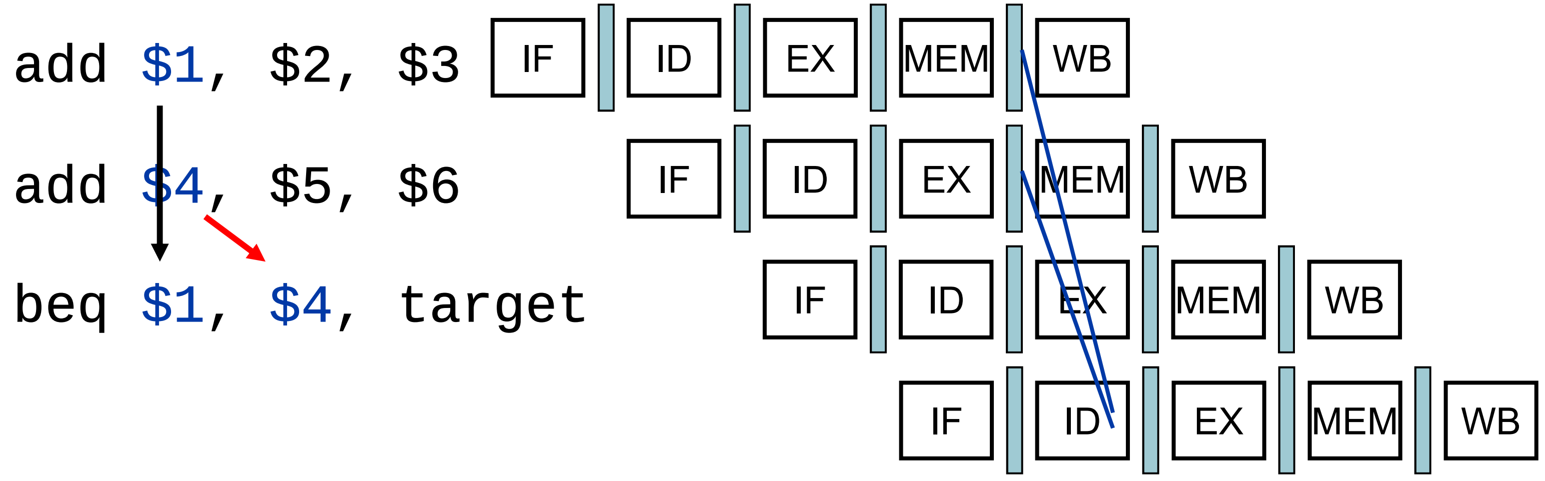
additional hardware



Data Hazards for Branch -- I

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3

add $4, $5, $6

beq $1, $4, target
```

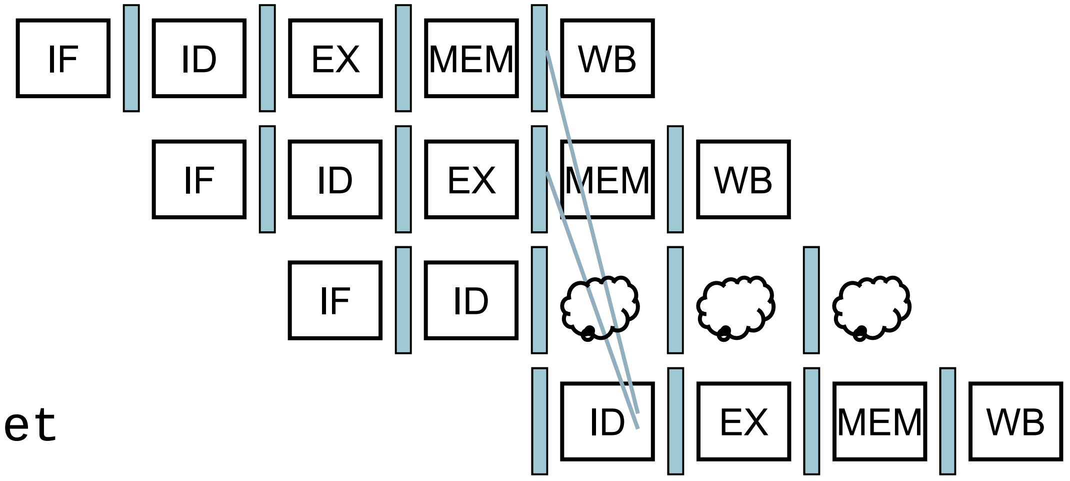


- Can resolve using forwarding, but need 1 stall cycle

Data Hazards for Branch -- I

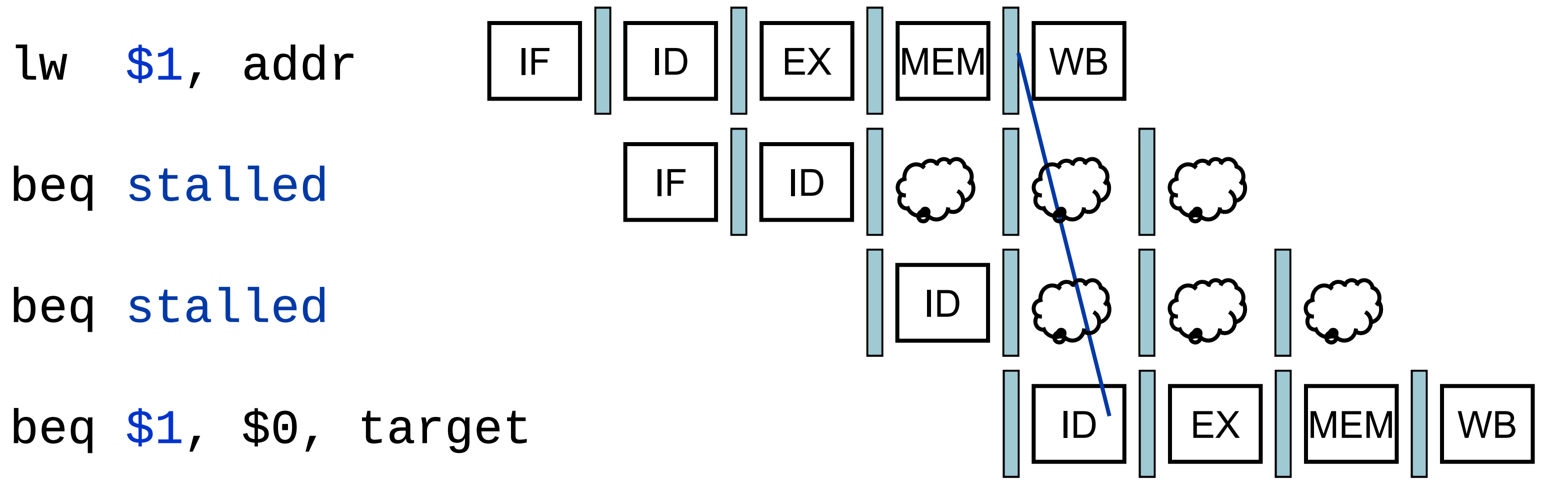
- 1 stall cycle + forwarding for ALU results

```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



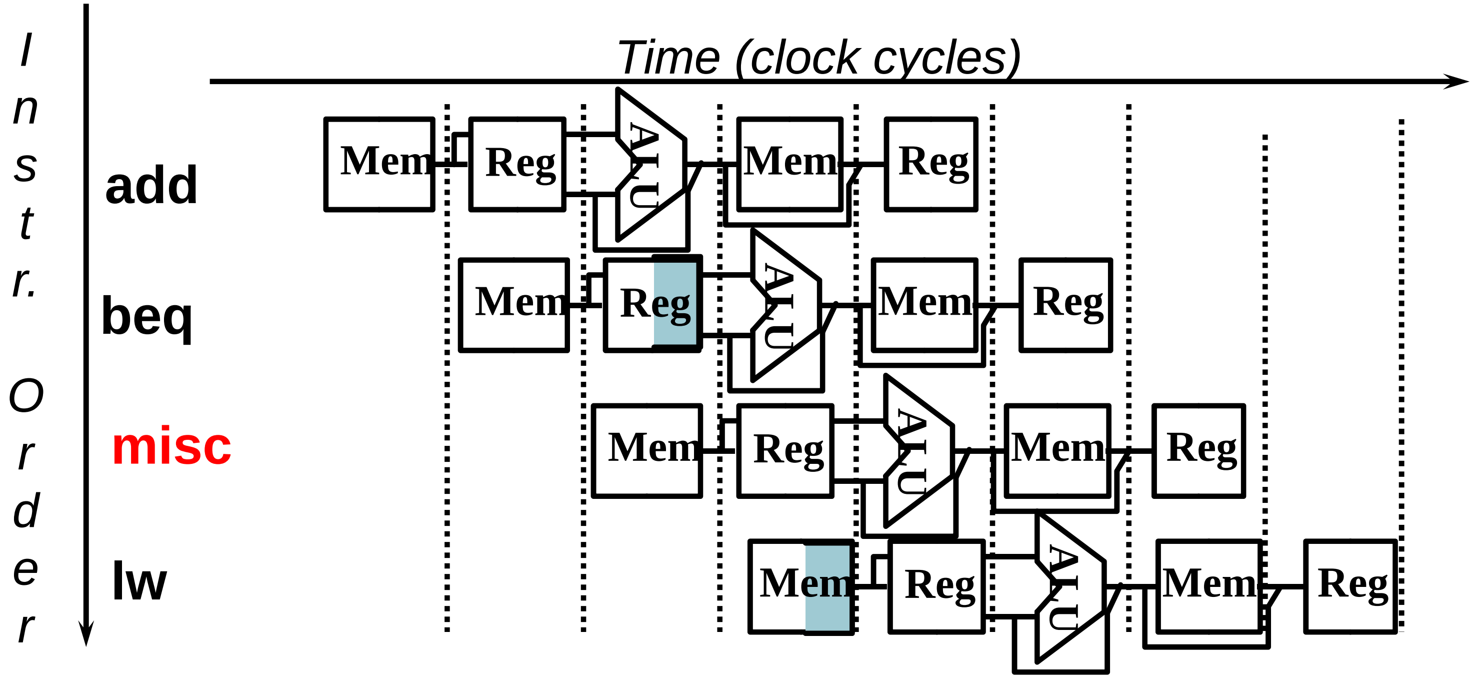
Data Hazards for Branch -- II

- If a comparison register is a destination of immediately preceding load instruction
  - Can resolve using forwarding, but need 2 stall cycles



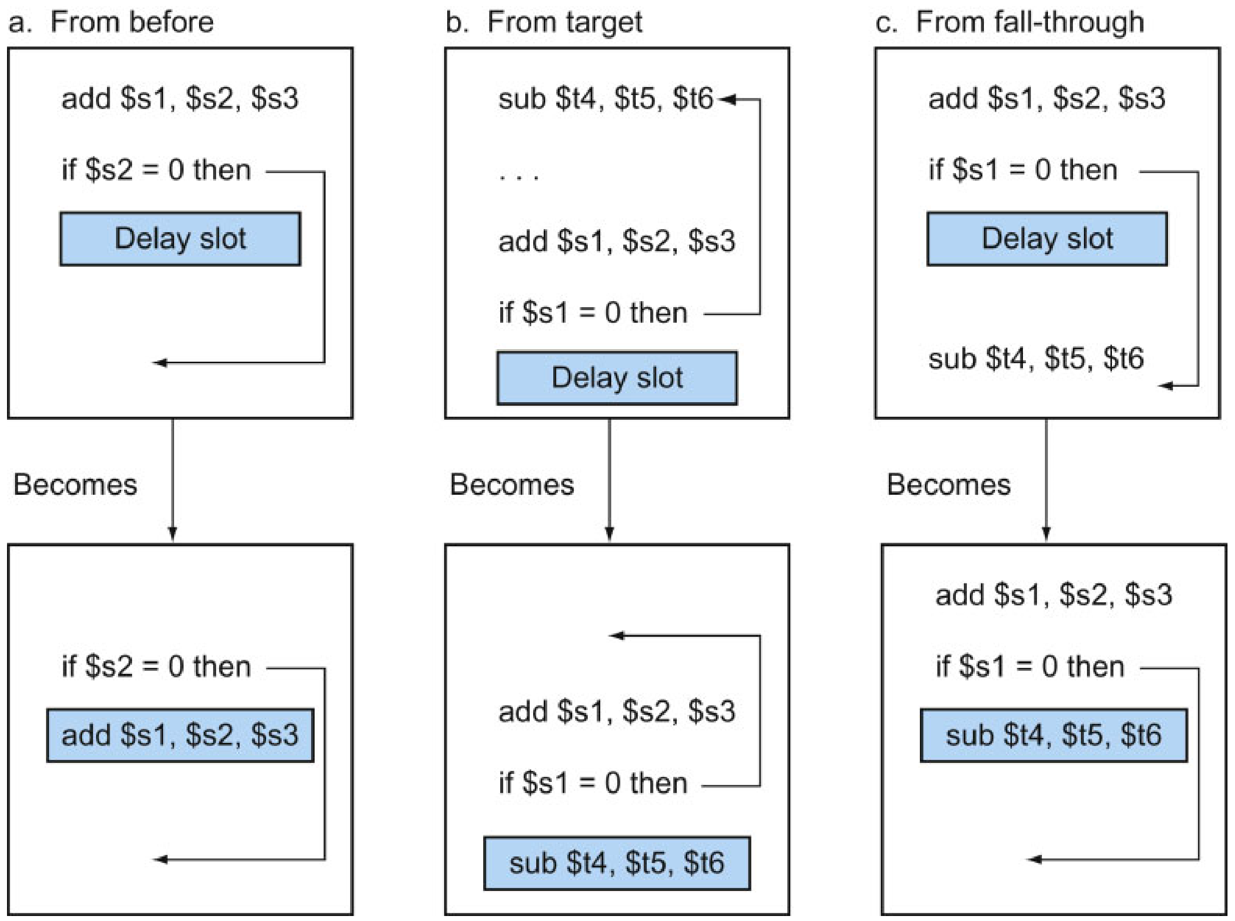
Delayed Branch

- Predict-not-taken + branch decision at ID
  => the following instruction is always executed
  => branches take effect 1 cycle later



  - 0 clock cycle penalty per branch instruction if can find instruction to put in slot

Scheduling the Branch Delay Slot



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the **sub** instruction may need to be copied, increasing IC
- In B and C, must be okay to execute **sub** when branch fails

Delay-Branch Scheduling Schemes

Scheduling Strategy	Requirements	Improve Performance When?
From before	Branch must not depend on the rescheduled instructions	Always
From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions	When branch is taken. May enlarge program if instructions are duplicated
From fall through	Must be OK to execute instructions if branch is taken	When branch is not taken.

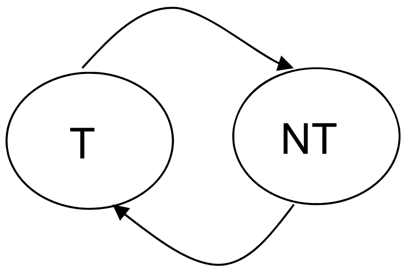
Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic branch prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction ➔ 1-bit predictor

Shortcoming for 1-Bit Predictor

- Inner loop branches mispredicted twice!

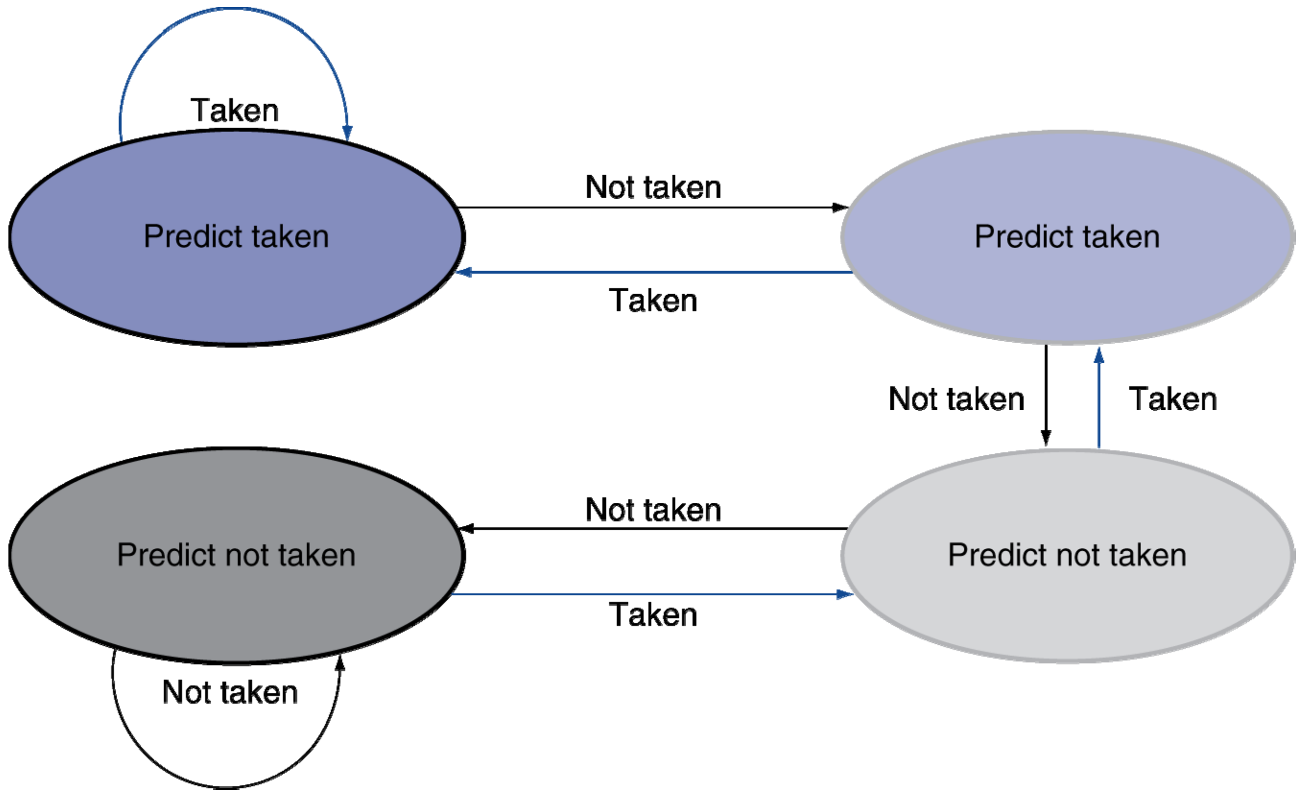
```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```



- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions

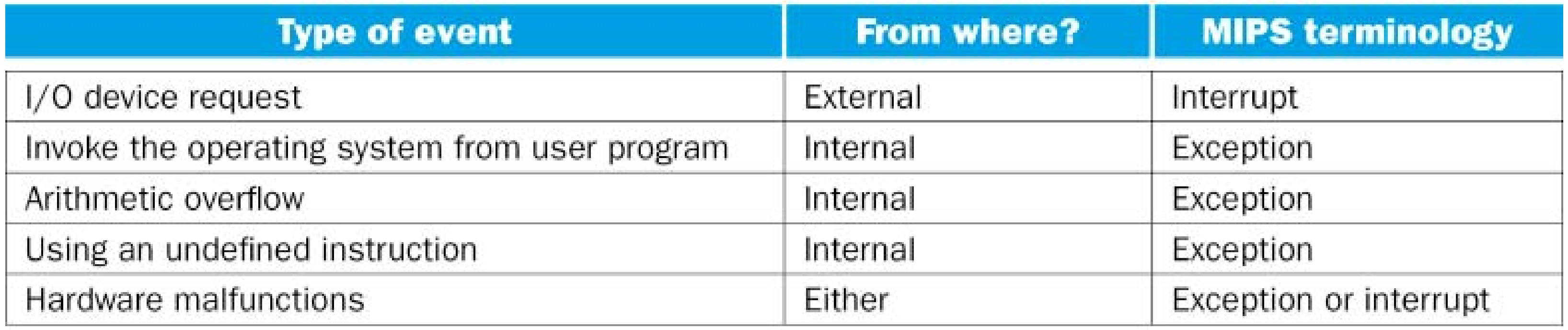


Calculating the Branch Target Address

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch in 5-stage MIPS processor
- Branch target buffer (discussed in CA course)
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately
  - 0-cycle penalty

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception: Arises within the CPU
  - e.g., undefined opcode, overflow, syscall, …
- Interrupt: From an external I/O controller



Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

- Dealing with execeptions without sacrificing performance is hard

Handling Exceptions in MIPS

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- 1. Save PC of offending (or interrupted) instruction in Exception Program Counter (EPC)
- 2. Save indication of the problem in Cause register
  - Must know the reason for the exception
  - Cause is a status register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- 3. Save registers in memory (similar to procedure call)
- 4. Jump to exception handler at 8000 00180

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:

Undefined opcode:	C000 0000
Overflow:	C000 0020
…:	C000 0040

- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program (also need to restore the saved registers from memory)
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

Must subtract 4 from EPC

Exceptions in a Pipeline

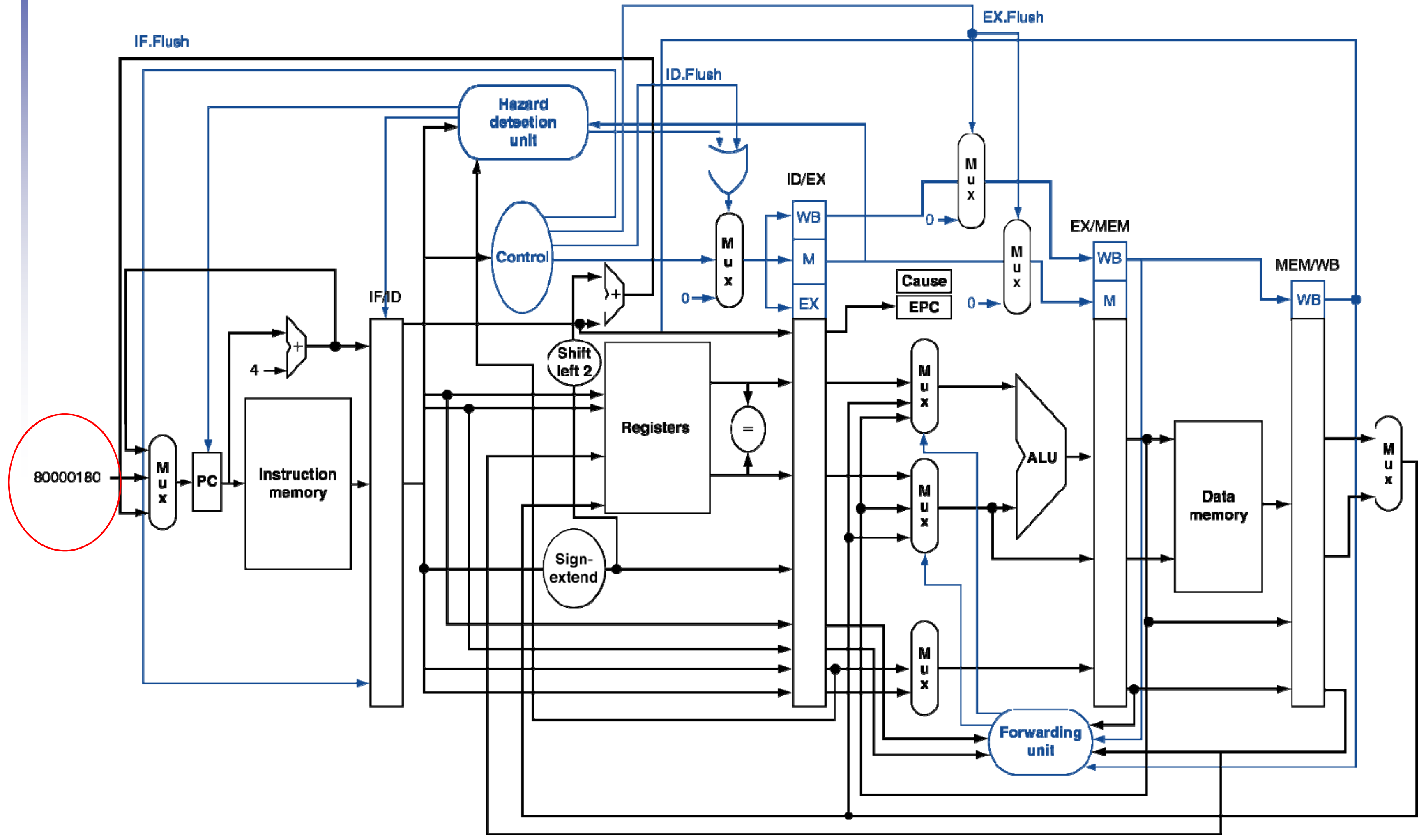
- Another form of control hazard
- Consider overflow on add in EX stage

```
add $1, $2, $1
```

  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush **add** and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions

Zeros control signals for flushing



Exception Summary

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Exception Example

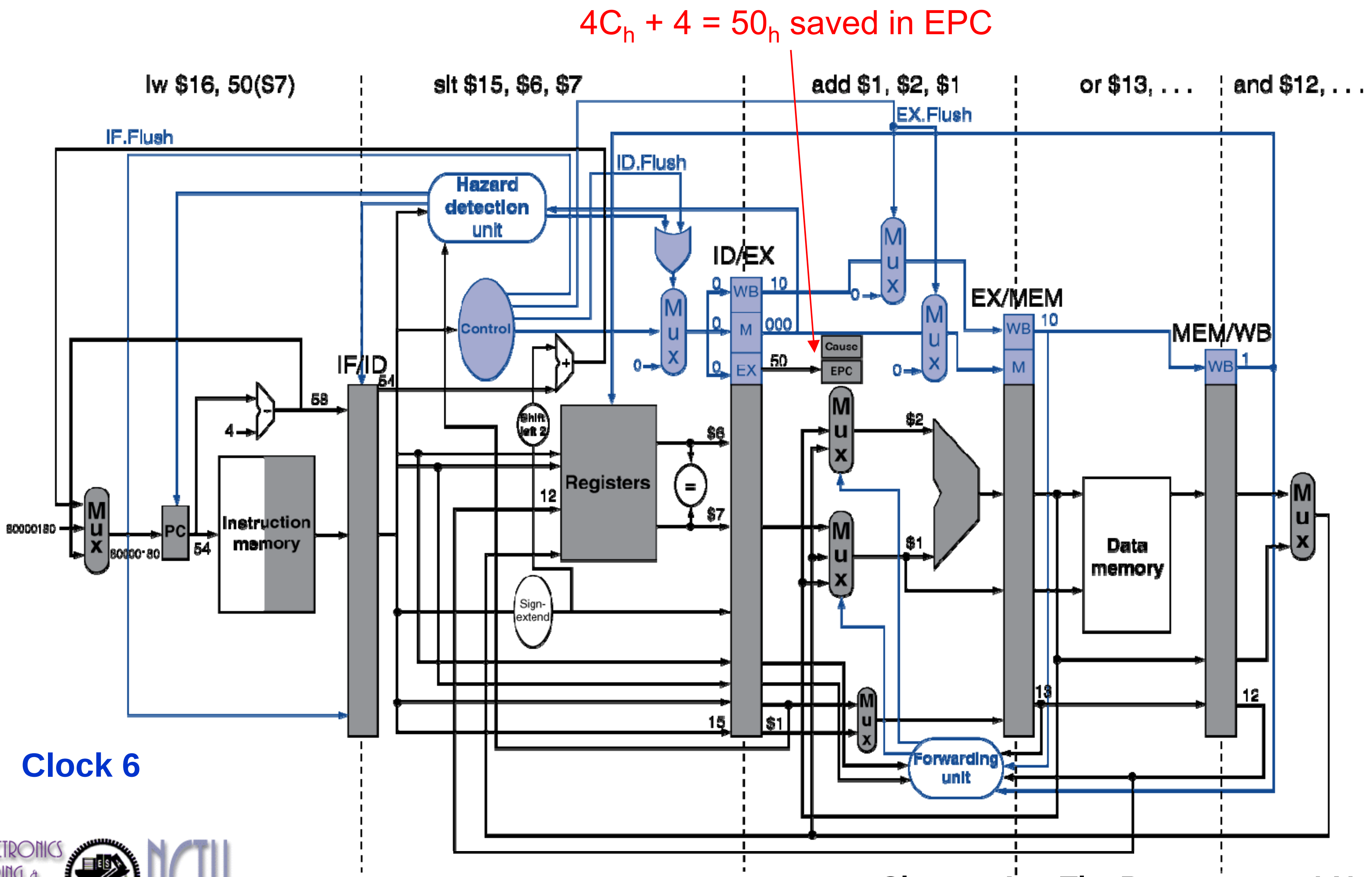
- Exception on add in

```
40      sub   $11, $2, $4
44      and   $12, $2, $5
48      or    $13, $2, $6
4C      add   $1,  $2, $1
50      slt   $15, $6, $7
54      lw    $16, 50($7)
…
```

- Handler

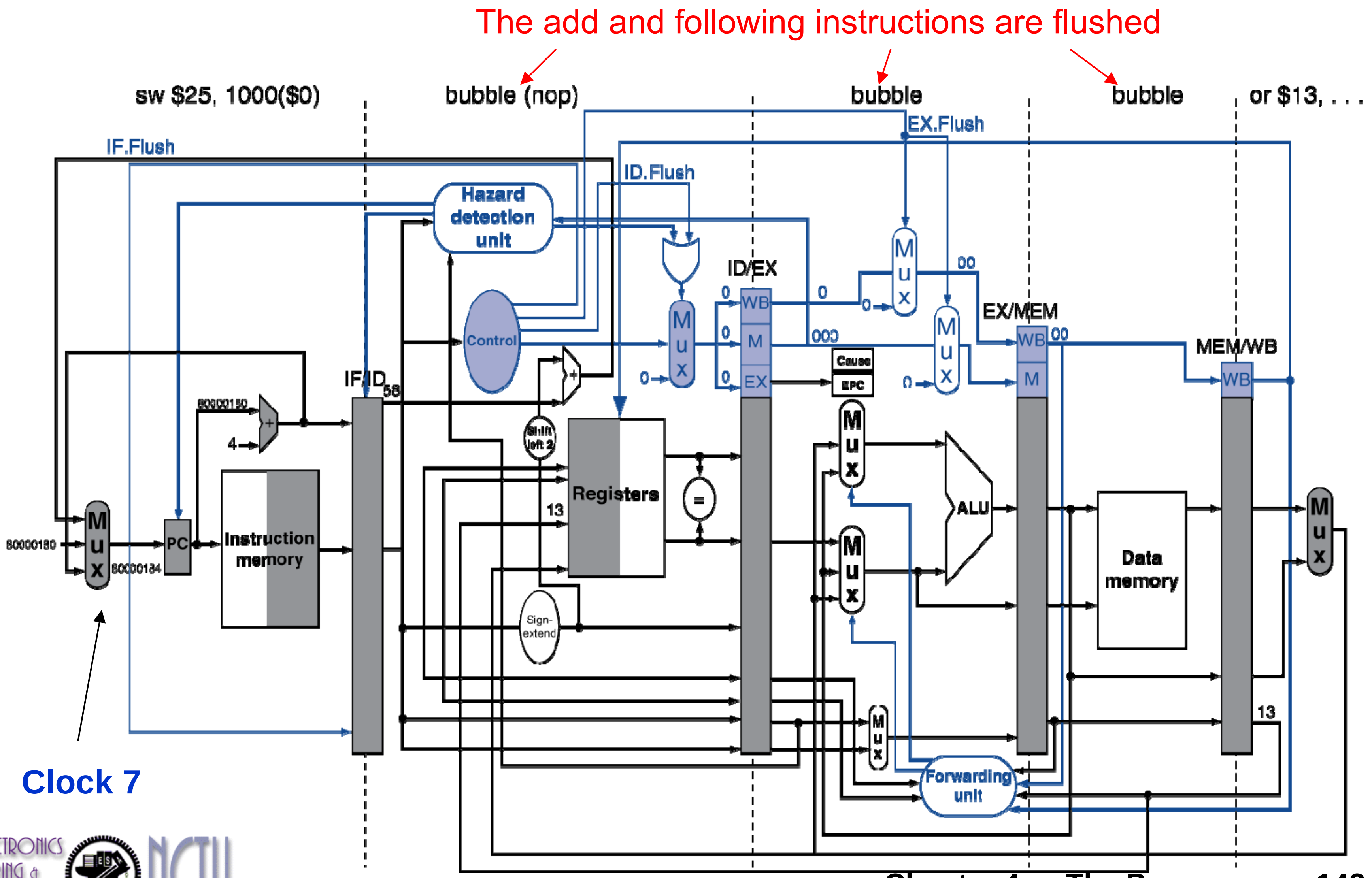
```
80000180   sw   $25, 1000($0)
80000184   sw   $26, 1004($0)
…
```

Exception Example



Clock 6

Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult! (discussed in CA course)

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require "manual" completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline (increase clock rate)
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue (using multiple ALUs)
    - Replicate pipeline stages $\Rightarrow$ multiple pipelined datapaths
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue (upto 4 parallel instructions)
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4, ideally
    - But dependencies reduce this in practice

Multiple Issue Processor

- Static multiple issue or VLIW processor
  - Compiler solves hazards, groups instructions to be issued together, and packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue or Superscalar processor
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime
- Rescheduling and loop unrolling techniques for multiple issue processors

MIPS with Static Dual Issue

- Two-issue packets
  - One for ALU/branch instruction and the other for load/store instruction
  - 64-bit aligned, 2-issue slot
    - Pad an unused instruction with nop
  - Peak IPC = 2

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Code Rescheduling

```
Loop: lw      $t0, 0($s1)
      addu    $t0, $t0, $s2
      sw      $t0, 0($s1)
      addi    $s1, $s1, -4
      bne     $s1, $zero, Loop
```

- After code rescheduling:

```
Loop: lw      $t0, 0($s1)
      addi    $s1, $s1, -4
      addu    $t0, $t0, $s2
      sw      $t0, 4($s1)
      bne     $s1, $zero, Loop
```

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Multiple-Issue Code Scheduling

- 2-issue processor

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

  - CPI: 4/5 = 0.8 (or IPC = 1.25)

Blank is nop

- Assume the loop index is a multiple of four
- After four-times loop unrolling and code scheduling

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi $s1,$s1,-16	lw $t0, 0($s1)	1
		lw $t1,12($s1)	2
	addu $t0,$t0,$s2	lw $t2, 8($s1)	3
	addu $t1,$t1,$s2	lw $t3, 4($s1)	4
	addu $t2,$t2,$s2	sw $t0, 16($s1)	5
	addu $t3,$t3,$s2	sw $t1,12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1,$zero,Loop	sw $t3, 4($s1)	8

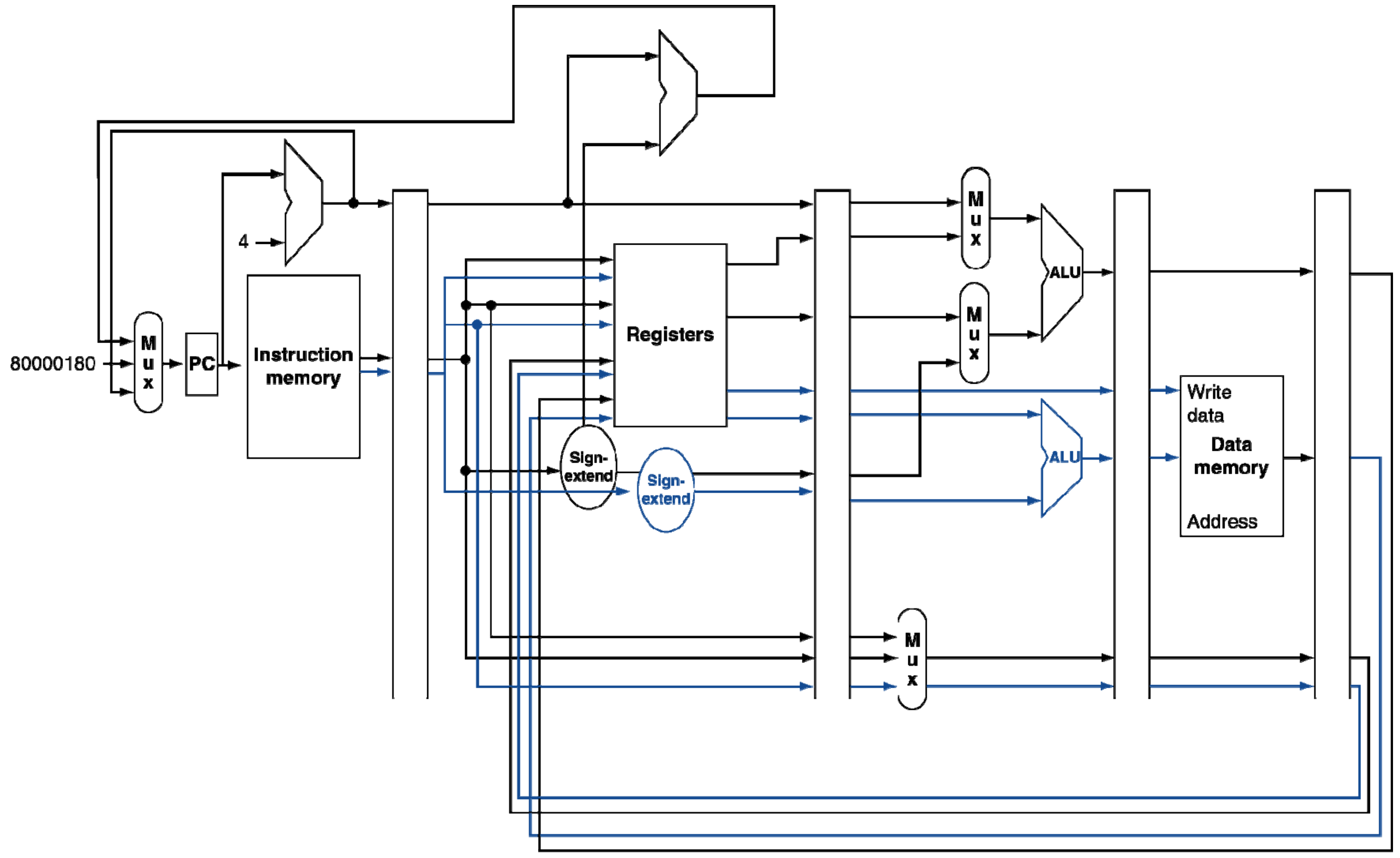
  - CPI: 8/14 = 0.57 (or IPC = 1.75)

Closer to 2, but at cost of registers and code size

Static Multiple Issue Processor

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Insert nop(s), if necessary
  - Software complexity ↑ Hardware complexity ↓

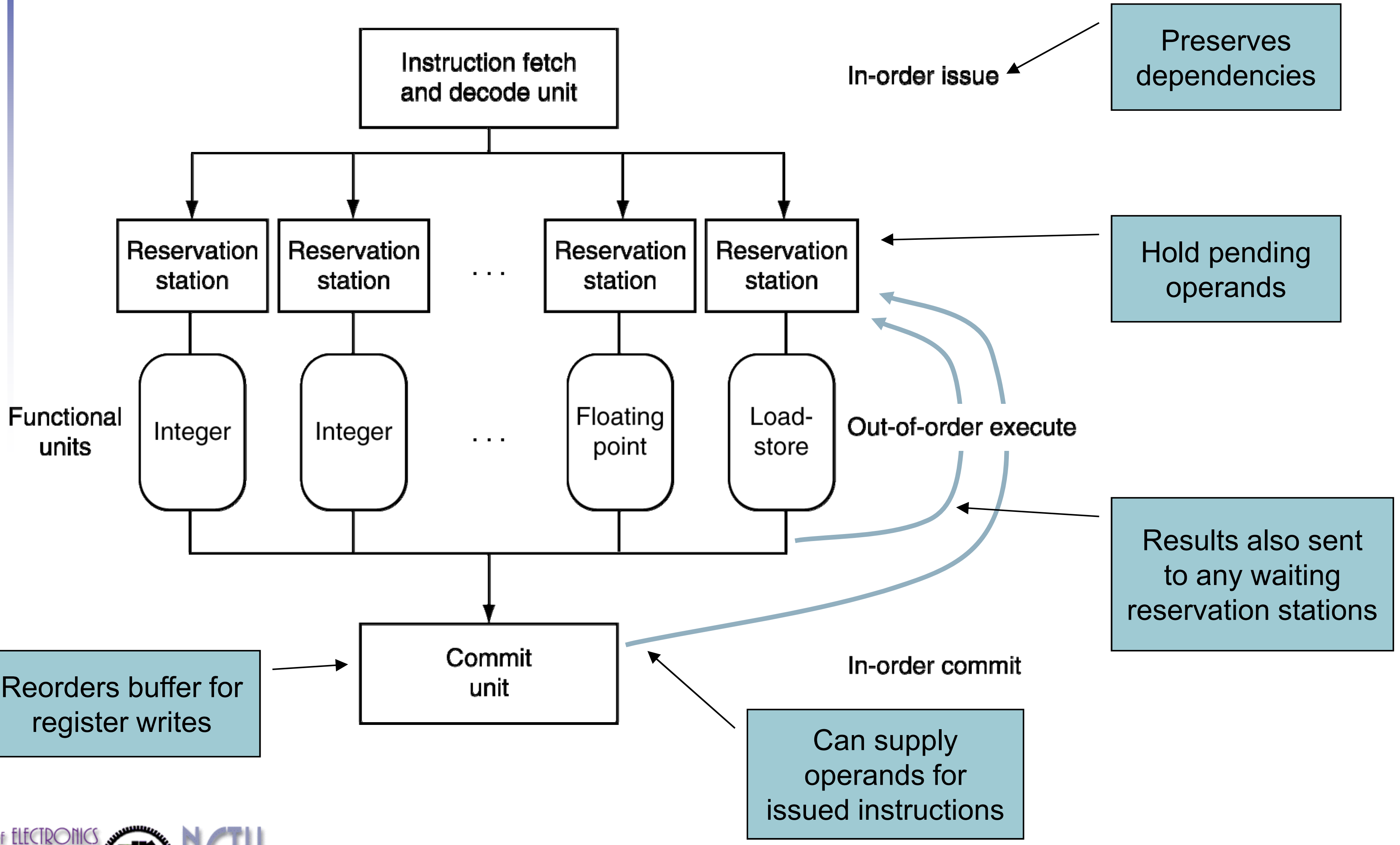
Two-Issue MIPS VLIW Processor



Dynamic Multiple Issue Processor

- CPU decides whether to issue 0, 1, 2, … each cycle (out-of-order execution and completion)
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU
- Old code still run
  - May not re-compile the code for new version
- Hardware complexity ↑ Software complexity ↓

Superscalar Processor



Speculation

- Predict and continue to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

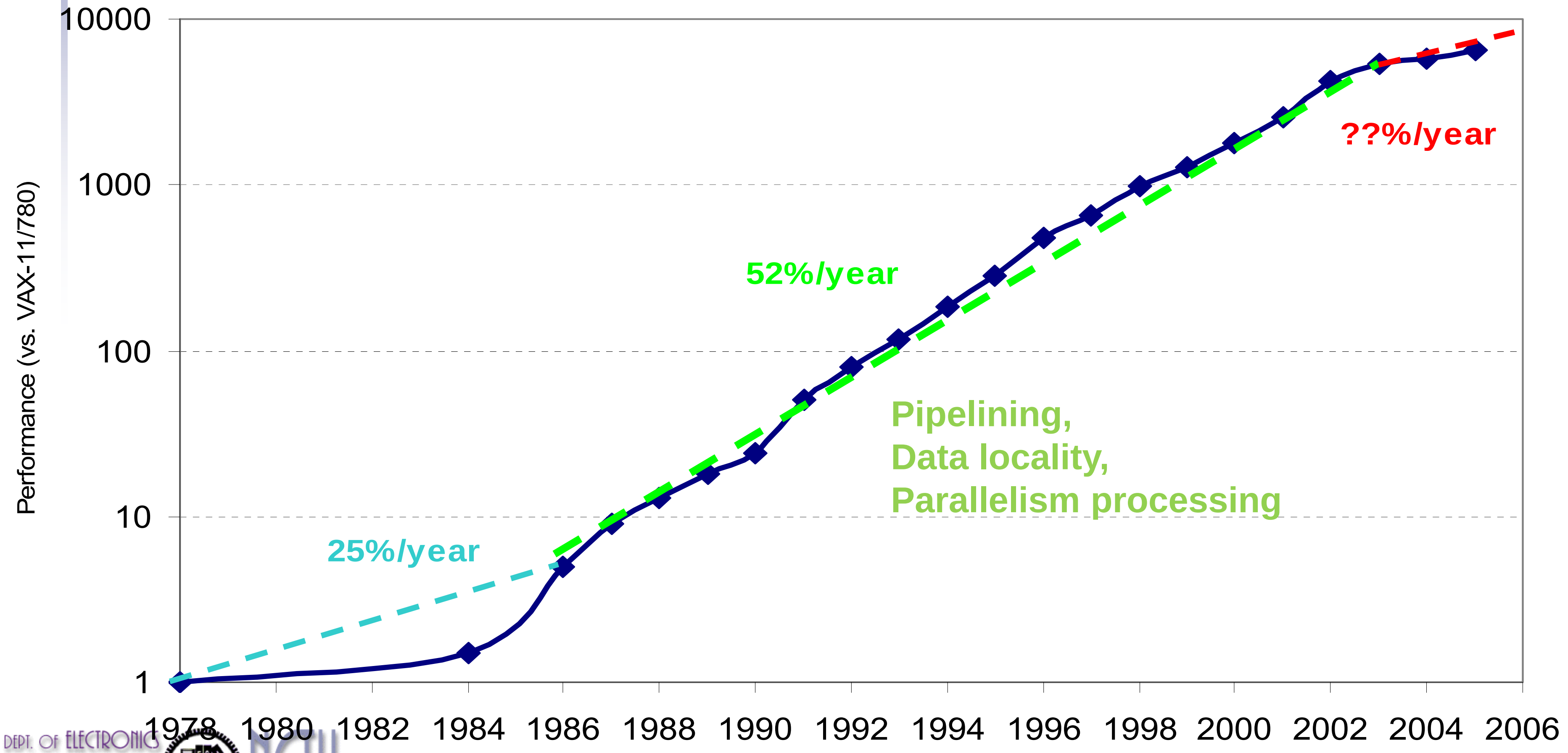
Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Multicore/Multiprocessor is the Trend

- Complexity of multiple-issue processors requires power
- Multiple simpler cores may be better



Cortex A8 and Intel i7

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	2-level	2-level
1st level caches/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-1024 KiB	256 KiB
3rd level caches (shared)	-	2- 8 MB

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall