CPSC 3300-001
Computer Systems Organization

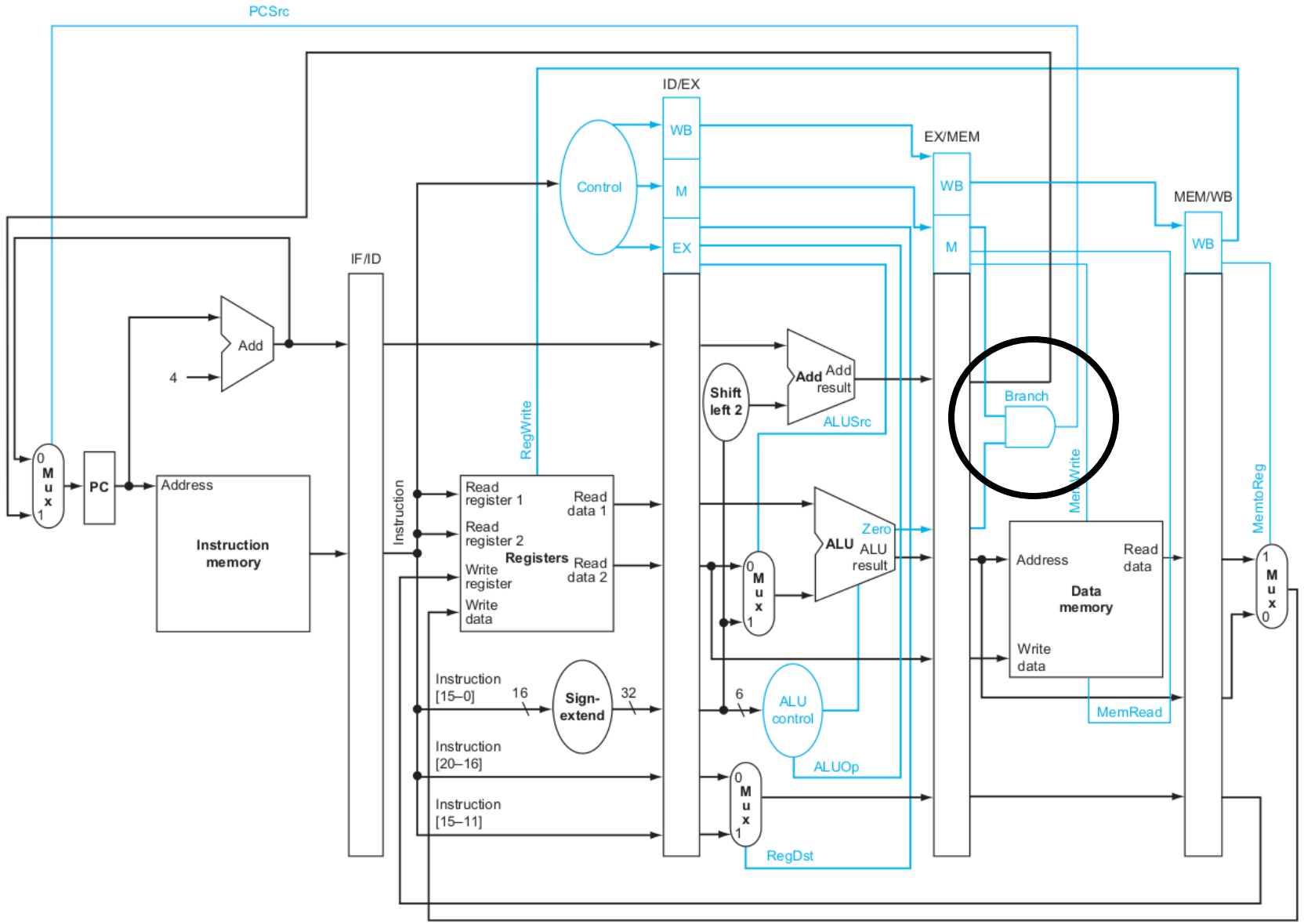
11. Control Hazards

Zhenkai Zhang

Announcement

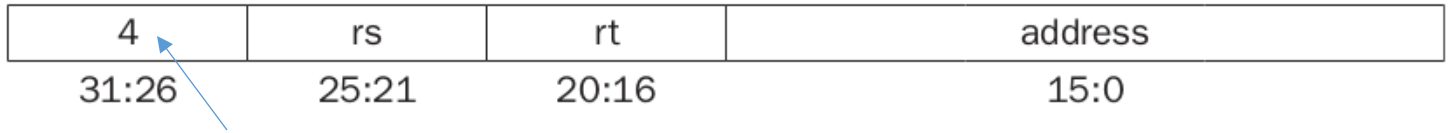
- Midterm exam on 10/5
  - Chapter 1, Appendix-B, and Chapter 4

Pipelined Datapath & Control Review



Conditional Branch Instruction Review

- A conditional branch reads two register operands and compares them
  - If the condition is met, PC is changed to the branch target address
    - Sign-extended displacement
    - Shift left by 2 bits
    - Add to PC + 4
      - PC-relative addressing mode
  - Otherwise, PC is just PC + 4 (i.e., falling through)



beq (opcode is 5 for bne)

- Assume Loop at $(80000)_{10}$

```
while (save[i] == k)
    i += 1;
```

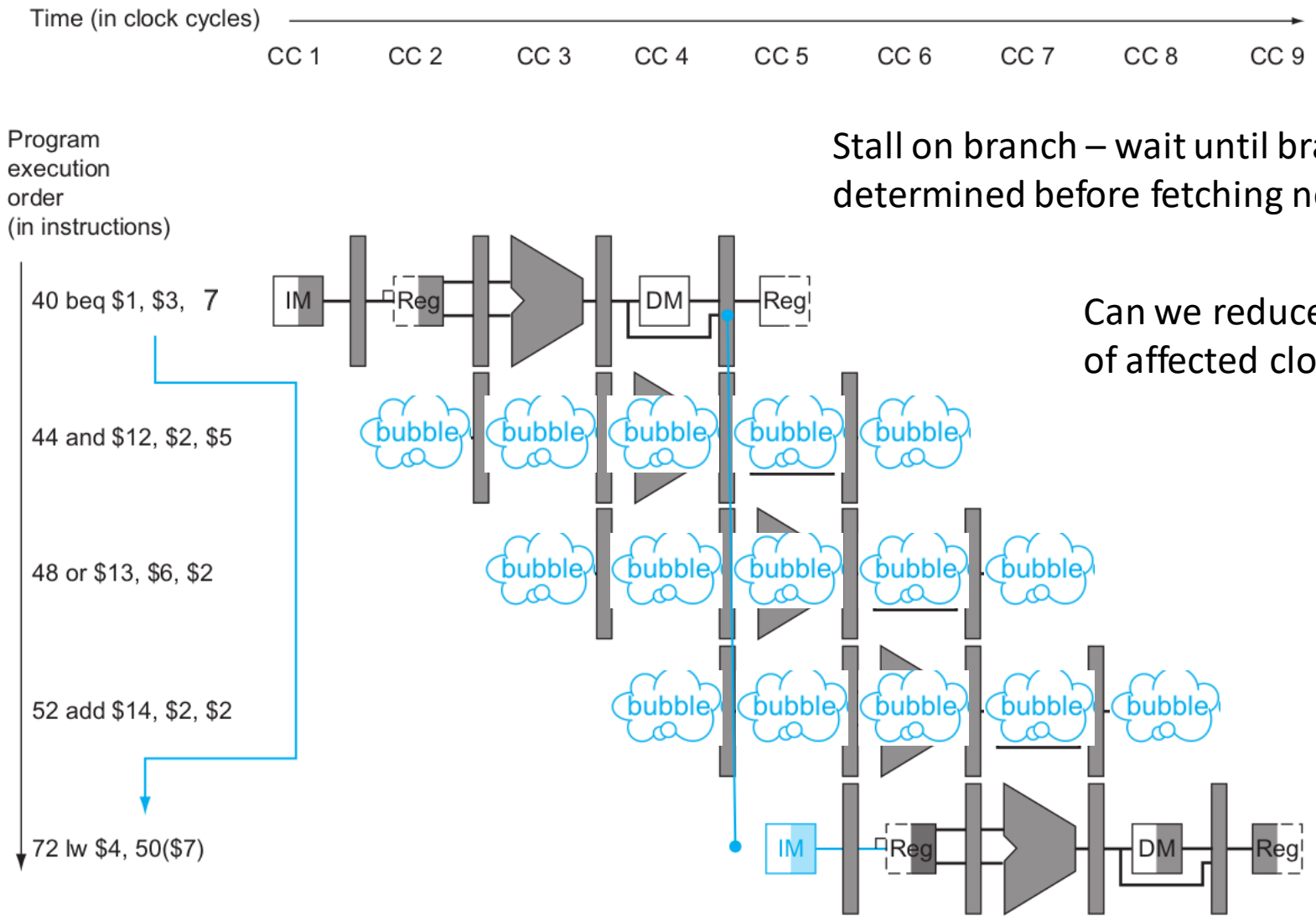
```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Control Hazards

- Branches determine the flow of control
- When a conditional branch is executed, it may or may not change the PC to something other than its current value plus 4
  - Fetching next correct instruction depends on a branch's outcome
  - The pipeline cannot always fetch the correct instruction
    - If a conditional branch is taken, the correct ones start from the branch target address
- If no intervention, the pipeline will execute incorrect instructions if a conditional branch turns out to be taken
  - How many incorrect instructions?
  - How to guarantee correctness?

The Impact of Pipeline on Branch Instruction

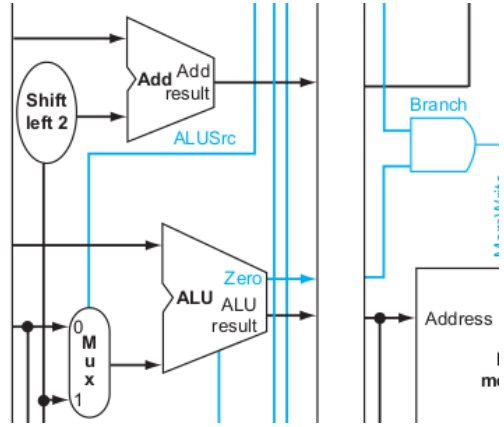


Stall on branch – wait until branch outcome determined before fetching next instruction

Can we reduce the number of affected clock cycles?

How to Reduce Taken Branch Cost?

- Why we resolve the branch in MEM in the current pipeline?
  - Branch target address calculation
    - PC + 4 and sign-extended immediate shifted left by 2 bits
  - Condition check
    - Using ALU to check the two operands' relationship
      - `beq` – if the two operands are equal (subtraction gives 0)



- Can we move the branch resolution to an earlier stage?
  - What is the earliest stage in which we can calculate the target address?
    - ID (PC + 4 and immediate field from the just fetched instruction)
  - Can we check branch's condition in the ID stage? (Let us focus on `beq`)

Simple Equality Test Logic

- Consider when we really see the two register operands in ID?
  - The second half of a clock cycle
    - The clock cycle is determined by the stage taking the most time
      - Usually that stage is MEM
  - After registers are read out, we may still have some time for some simple combinational logic
- How to check equality?
  - Subtraction and then NORing the bits
  - XORing individual bits and then NORing the XORed result
    - A simple two-level logic circuit

Data Hazard Caveats

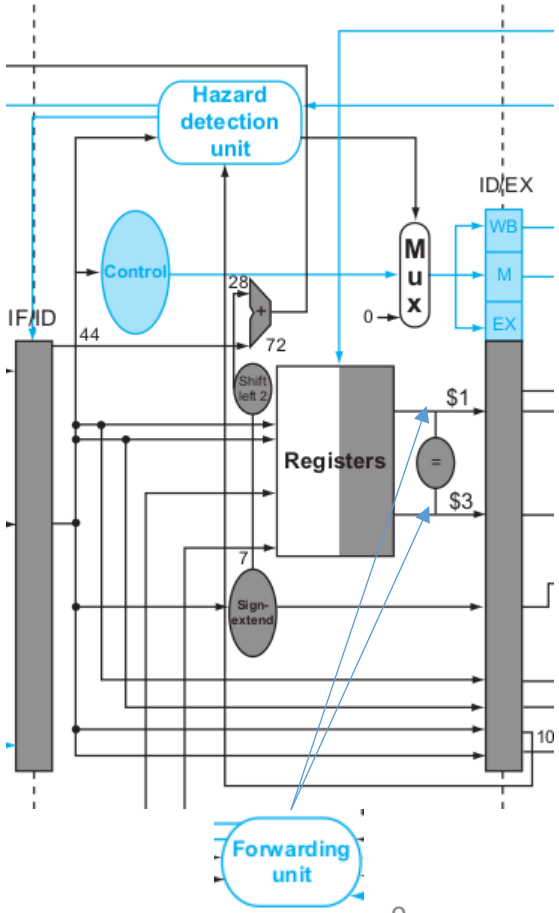
- If we move branch resolution to the ID stage, we need to deal with data hazards in the ID stage
  - Add new forwarding logic
    - Operands can come from EX/MEM or MEM/WB pipeline register
  - Add hazard detection unit
    - Stall the pipeline if forwarding cannot solve the potential data hazards

```
sub $4, $5, $6
beq $1, $2, 100
```

```
sub $4, $5, $6
beq $4, $2, 100
```

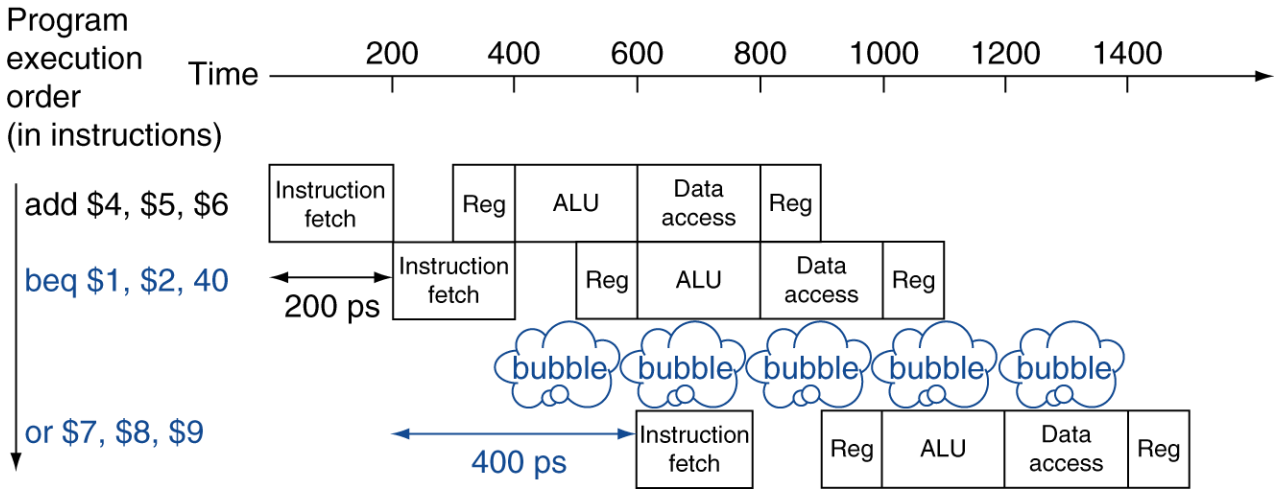
```
lw  $4, 20($5)
beq $4, $2, 100
```

```
sub $4, $5, $6
add $7, $8, $9
beq $4, $2, 100
```

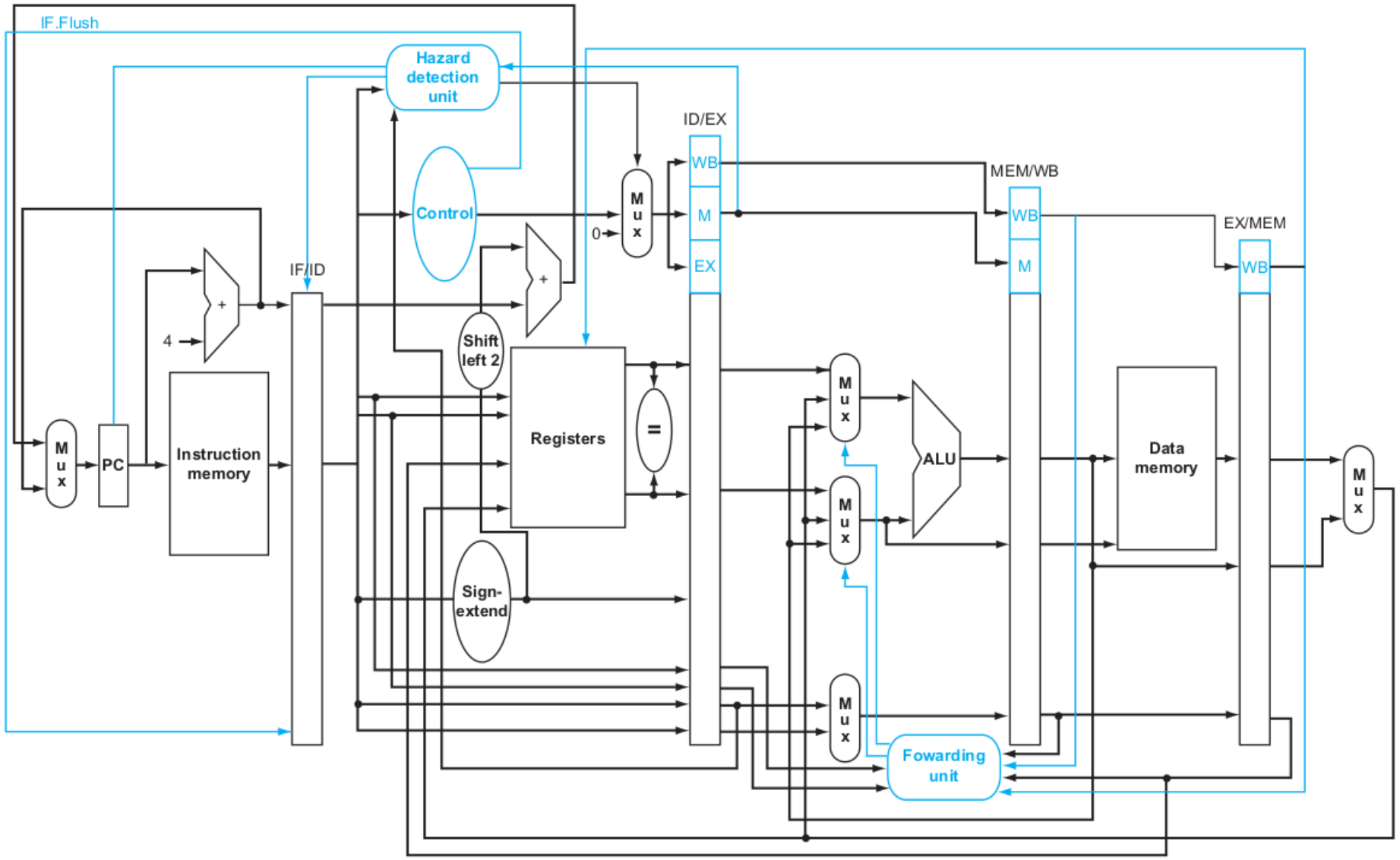


Flushing Fetched Instruction

- Even though branches can be resolved in the ID stage, there is still a stall – we need to flush instructions in the IF stage
  - Add a control line whose assertion will zero the instruction field of the IF/ID pipeline register
    - Transforms the fetched instruction into a `nop`

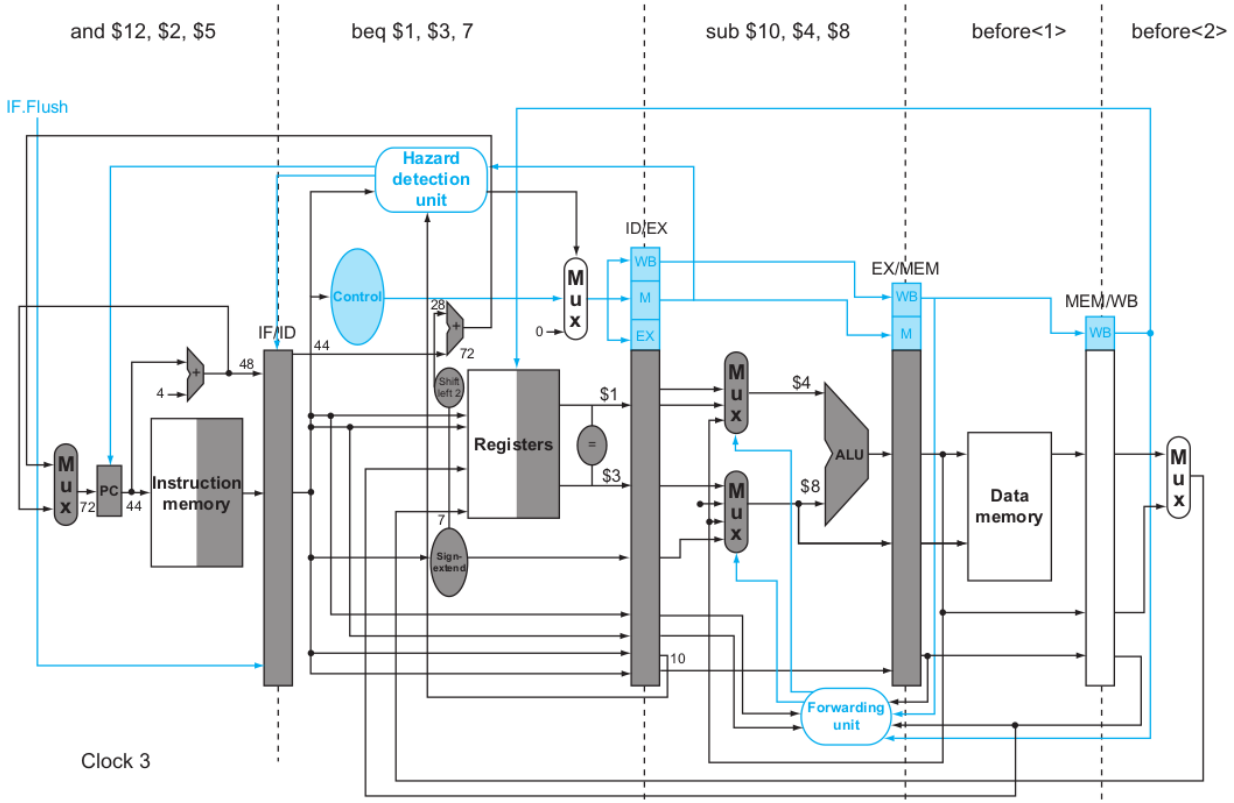


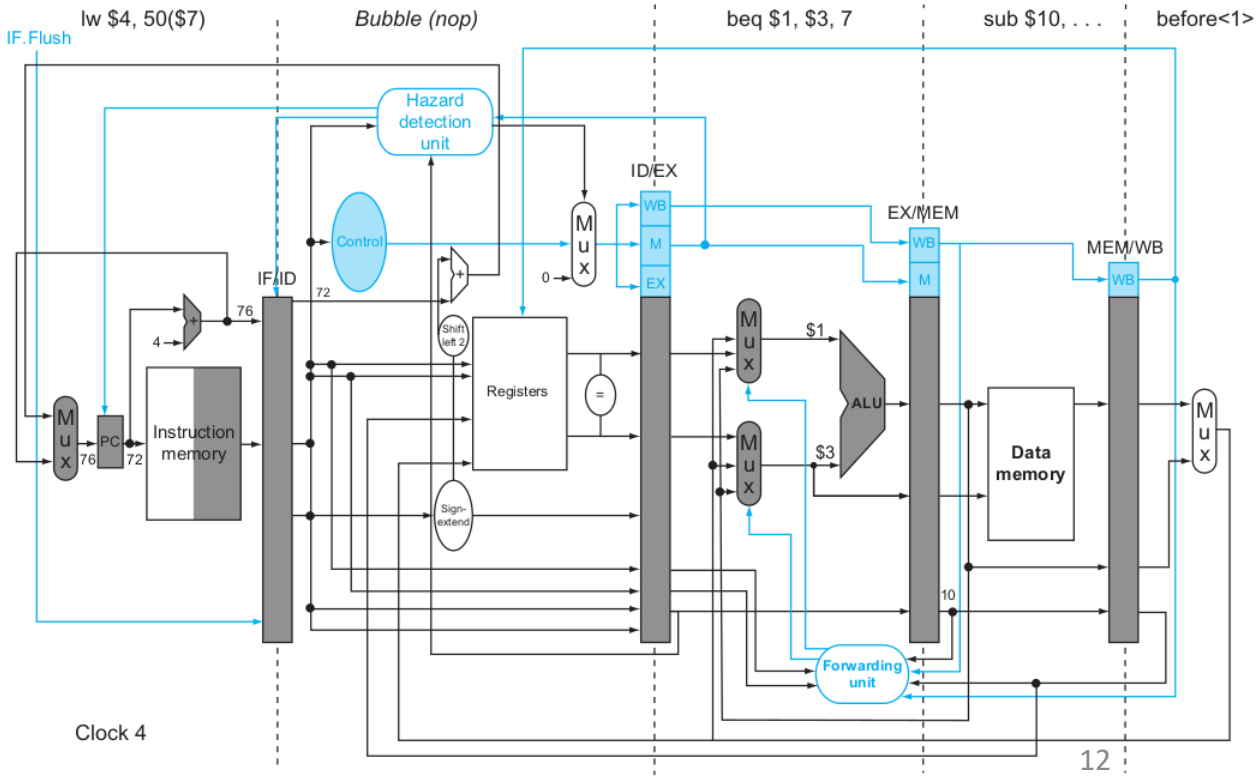
Final Datapath and Control



Stall on Branch

```
36 sub $10, $4, $8
40 beq $1, $3, 7  # PC-relative branch to 40 + 4 + 7 * 4 = 72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```





Branch Delay Slot

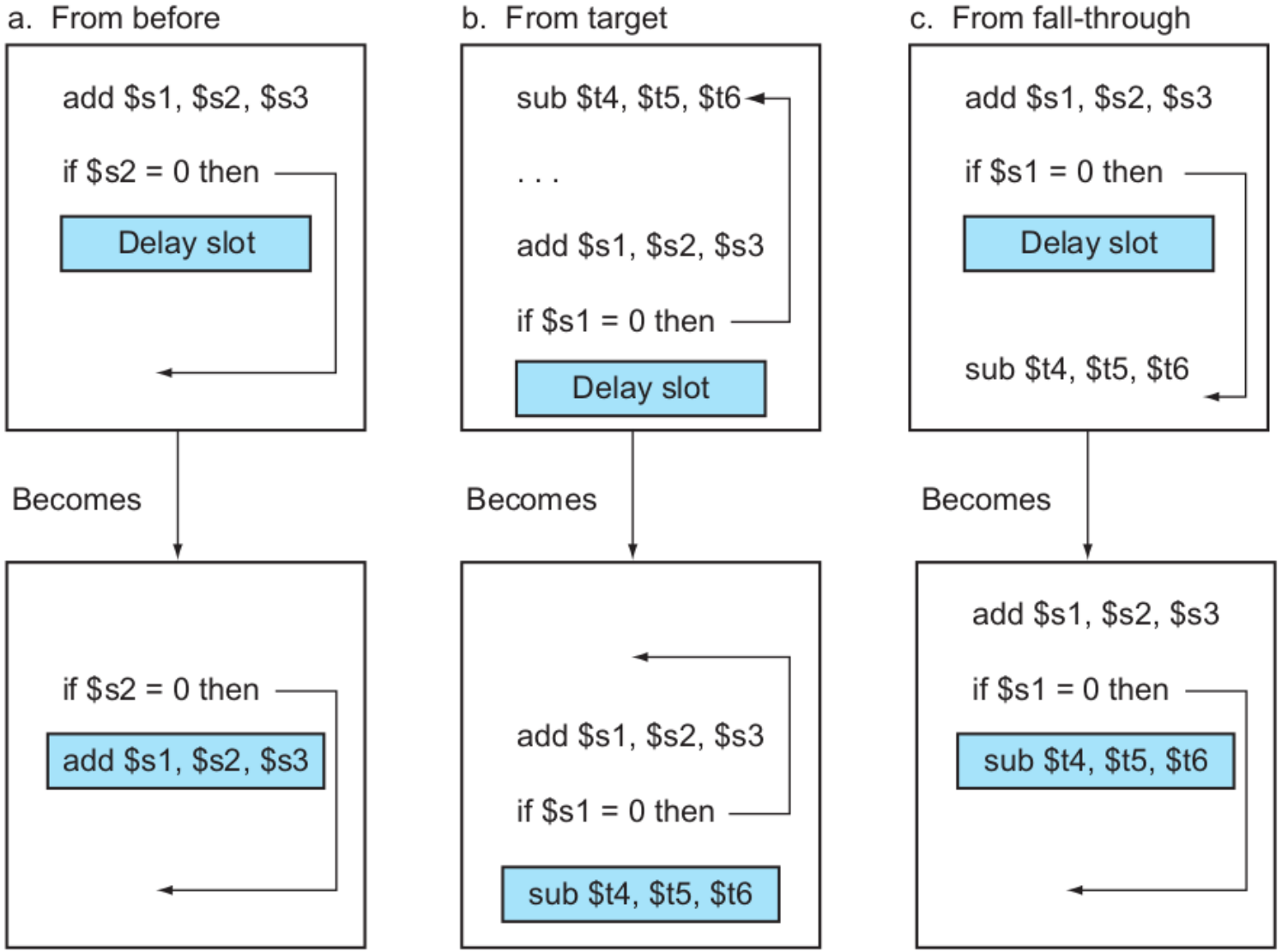
- Delayed branch technique was heavily used in early RISC processors and works reasonably well in the five-stage pipeline

branch instruction
sequential successor
branch target if taken

- In a delayed branch, there is a branch delay slot
  - A sequential successor is in the branch delay slot
  - This instruction is executed whether or not the branch is taken

Although it is possible to have a branch delay longer than one,
in practice almost all processors with delayed branch have a single instruction delay

Scheduling Branch Delay Slot

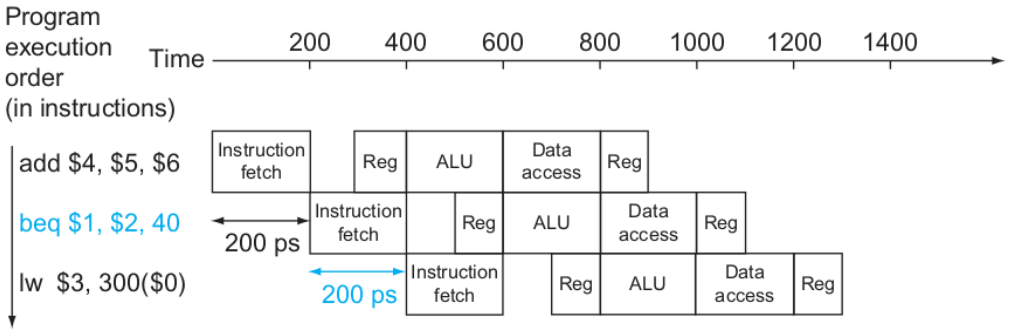


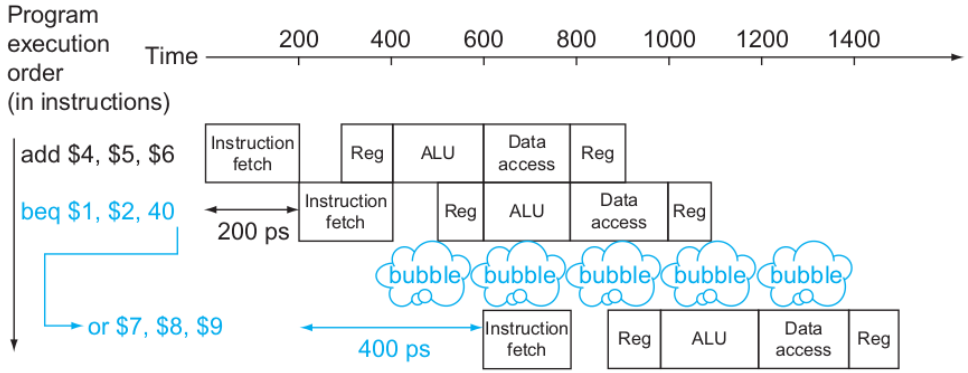
Branch Prediction

- The pipelines in modern processors get deeper and the potential branch penalty becomes much longer
  - Stall penalty becomes unacceptable
  - Using delayed branches and similar schemes becomes insufficient
- Why not try to predict the outcome of a branch
  - Both its decision and target address
  - Speculatively execute instructions along the predicted path
    - If prediction is correct, no loss
    - If prediction is wrong, flush the incorrect instructions in the pipeline, and then fetch the correct ones

Static Branch Prediction

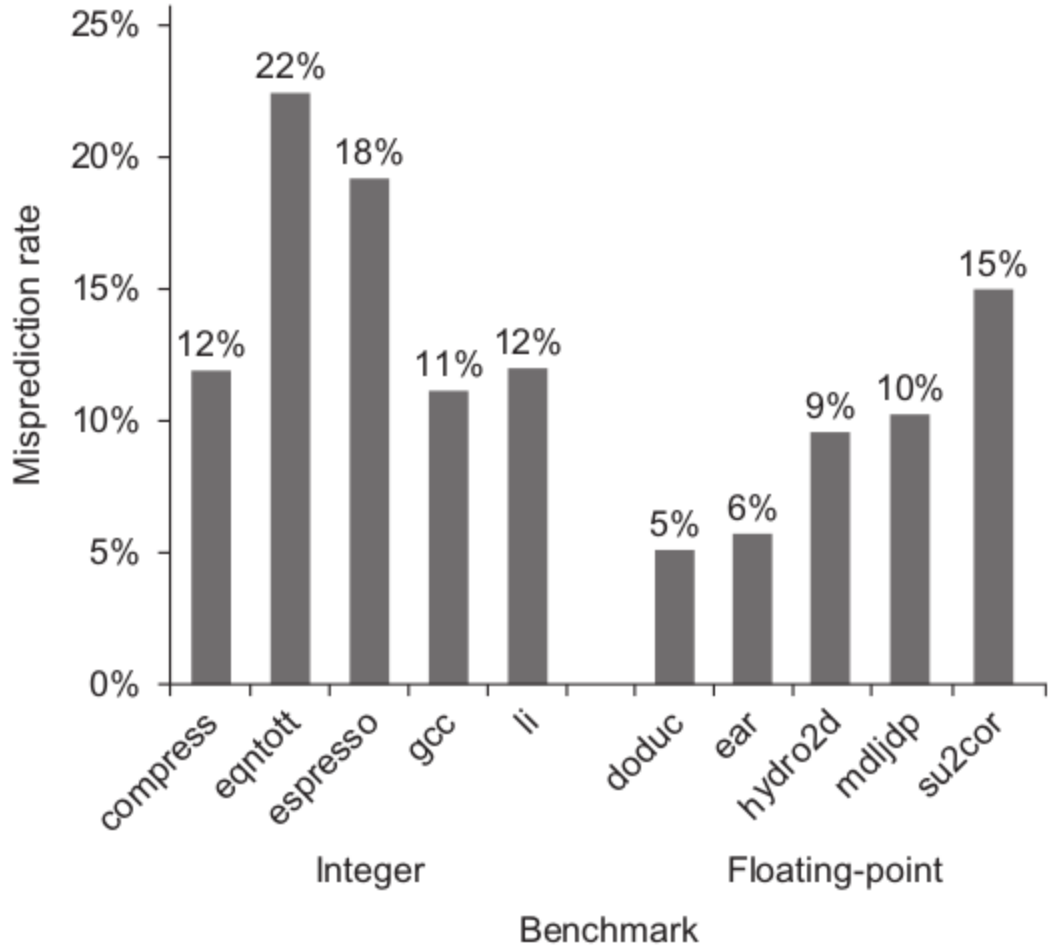
- The simplest scheme is to always predict "not taken" and continue execution down the sequential instruction stream





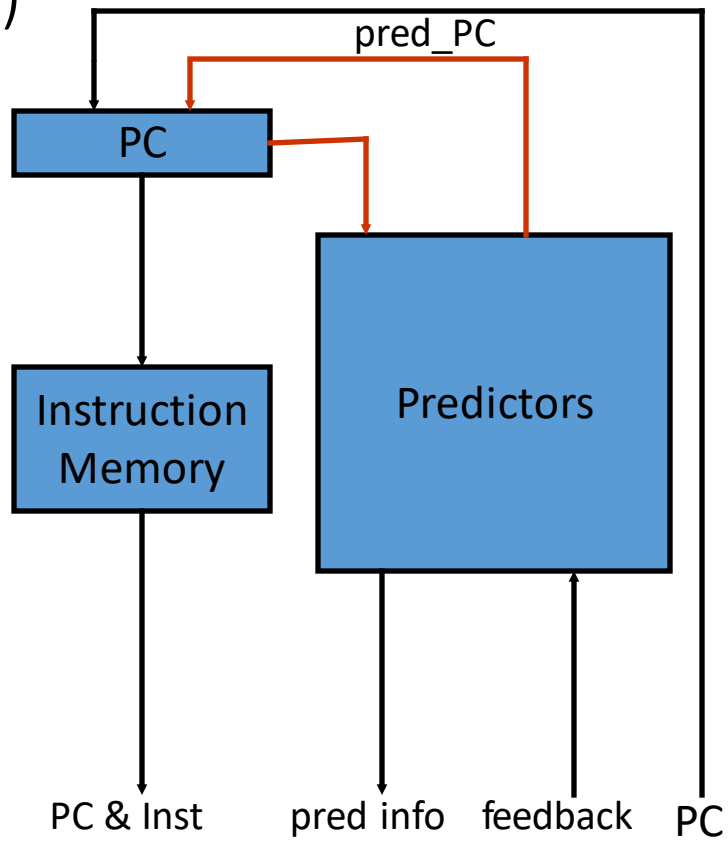
- A key observation is that the behavior of branches is often bimodally distributed
  - That is, an individual branch is often highly biased toward taken or untaken
  - We can use profile information collected from earlier runs to do compile-time branch prediction

Misprediction Rate (Profile-based Predictor)



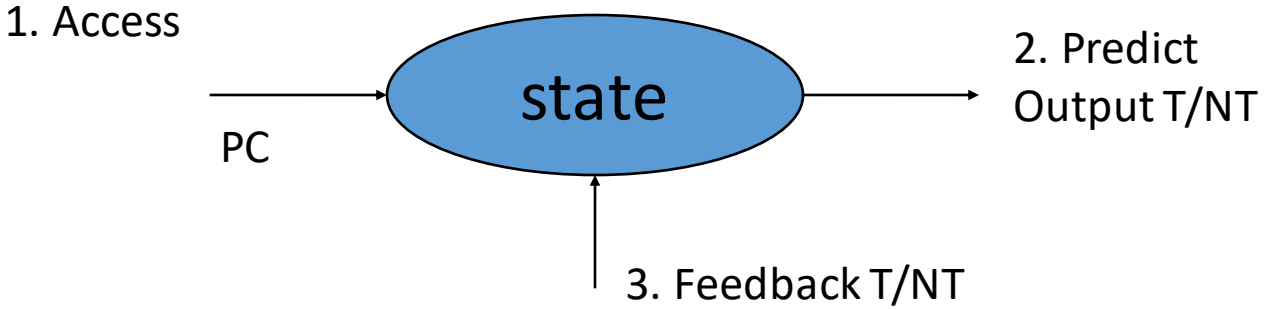
Dynamic Branch Prediction

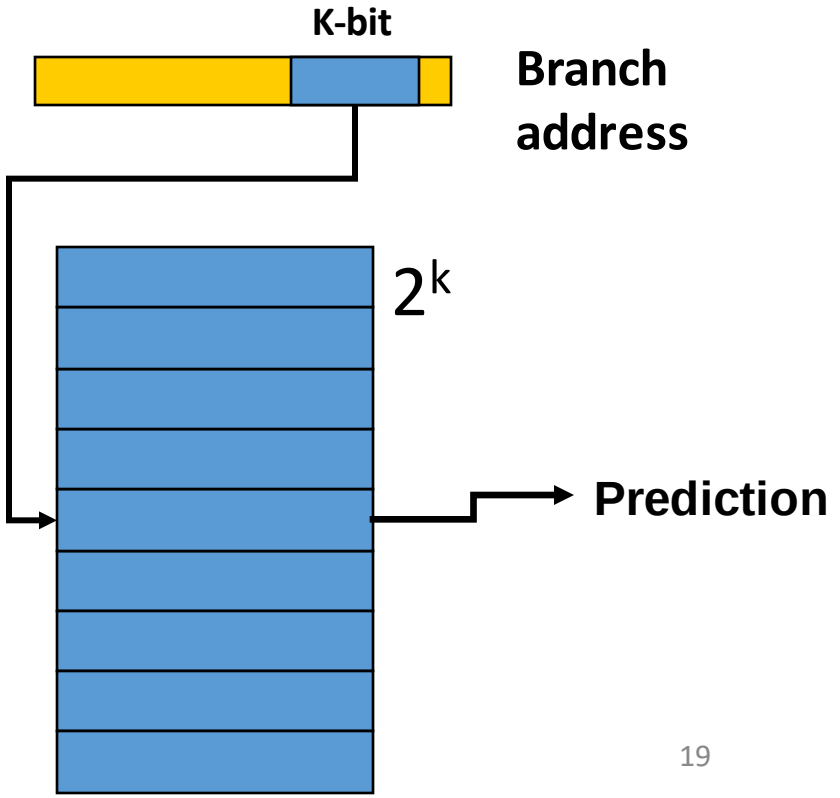
- We can improve the performance of branch prediction by using runtime information (learn lessons from the past)
- Available information:
  - Current PC
  - Past branch history (direction and target)
- What to predict:
  - Conditional branch instruction's branch direction and target address
  - Jump instruction's target address
  - Procedure call/return's target address



Branch Prediction Buffer

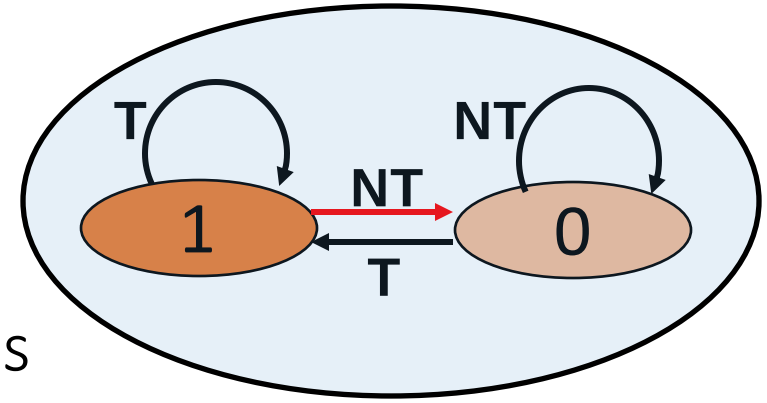
- Branch prediction buffer (branch/pattern history table) uses a table of simple predictors, indexed by bits of PC
  - Predict the branch direction (taken or not-taken)
  - More entries, more cost, but less conflicts, higher accuracy





1-bit Prediction

- The simplest way is to use one bit
  - 0 – not-taken 1 – taken
  - A misprediction flips the bit
- In a loop, 1-bit prediction will cause 2 mispredictions

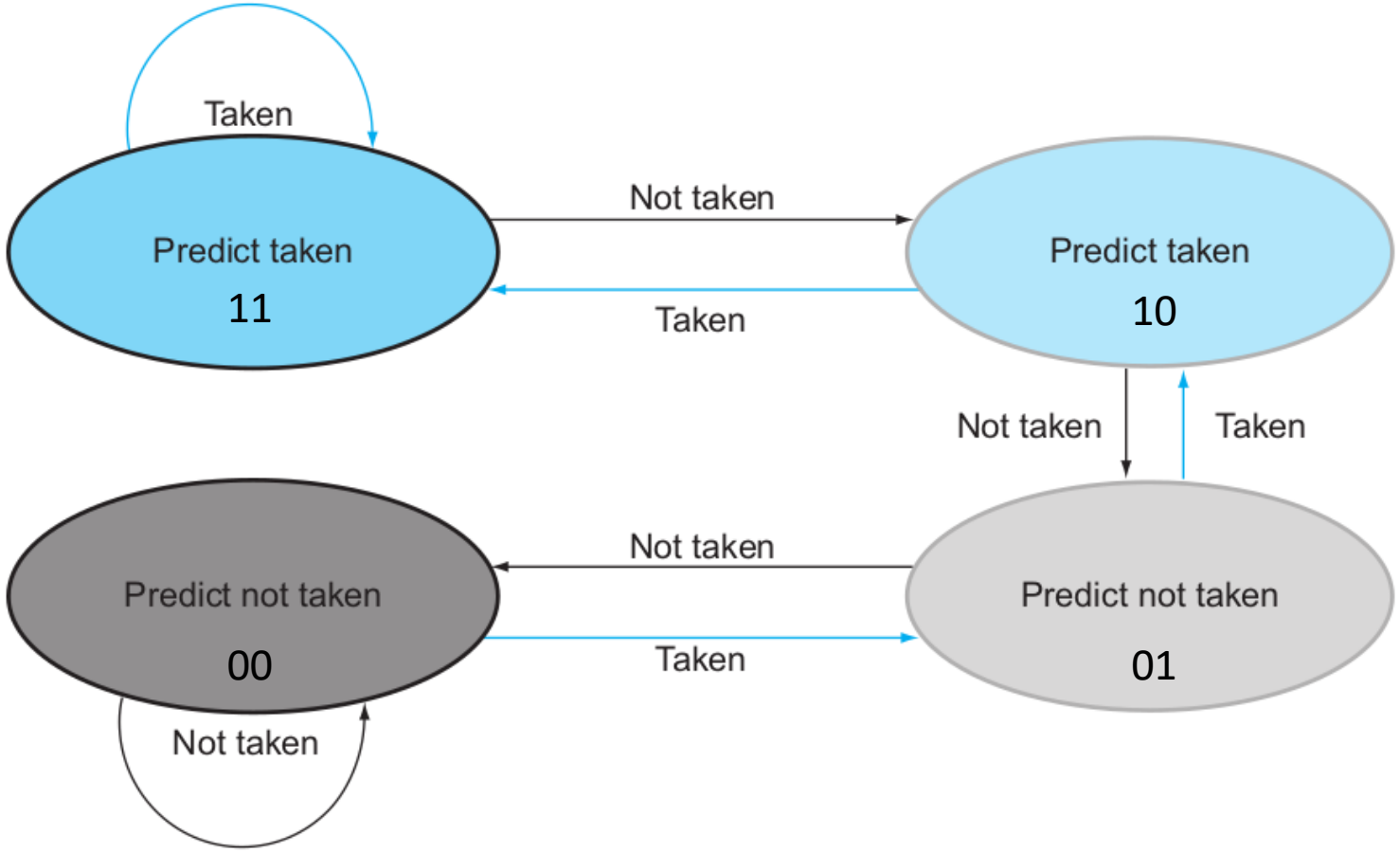



```
for (…){
  for (i = 0; i < 9; i++)
    a[i] = a[i] * 2;
}
```

  - Last iteration of the inner loop
  - First iteration of the inner loop
  - Only 80% accuracy even if the branch is taken 90% of the time

2-bit Saturating Counter

- 2-bit scheme may not change prediction direction if just got a single misprediction



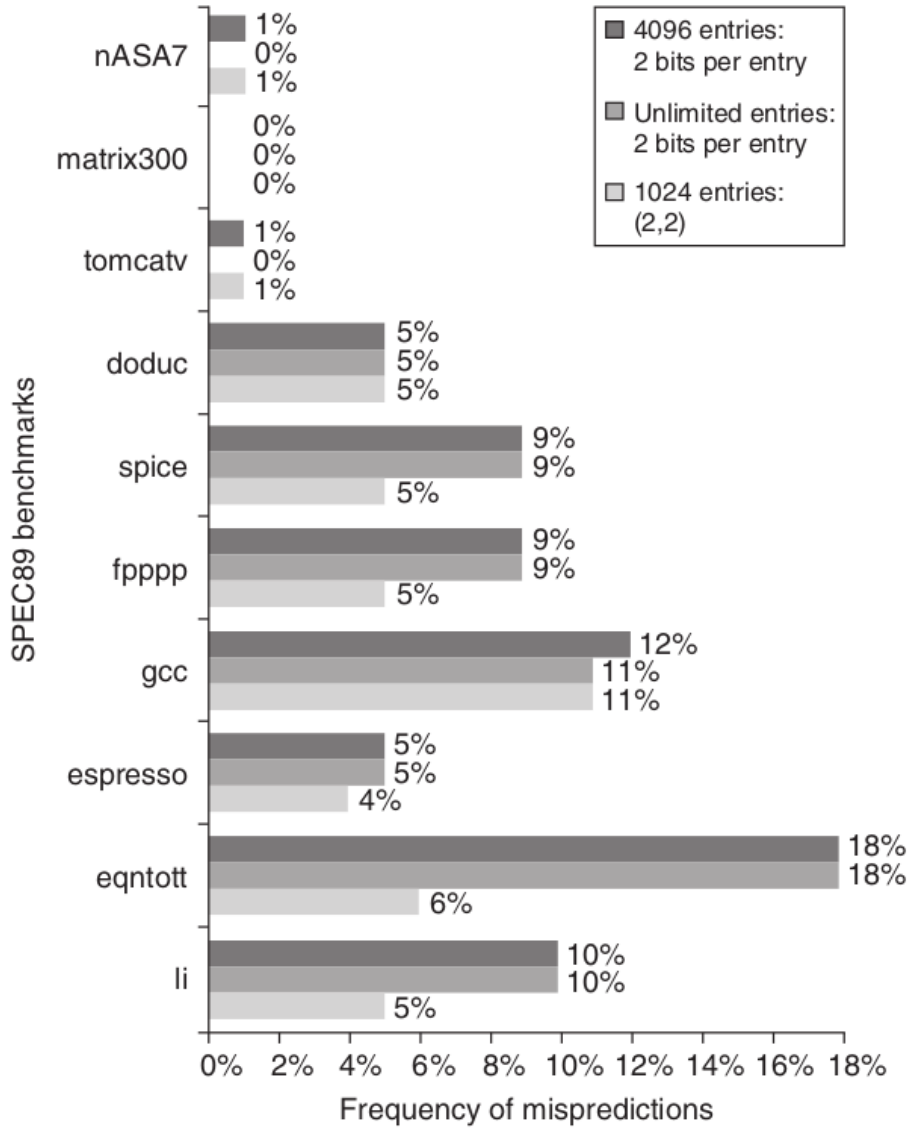
Correlating Branch Predictor

- The 2-bit predictor scheme uses only the recent behavior of a single branch to predict the future behavior of that branch

```
if (aa==2)
  aa=0;
if (bb==2)
  bb=0;
if (aa!=bb) {
```

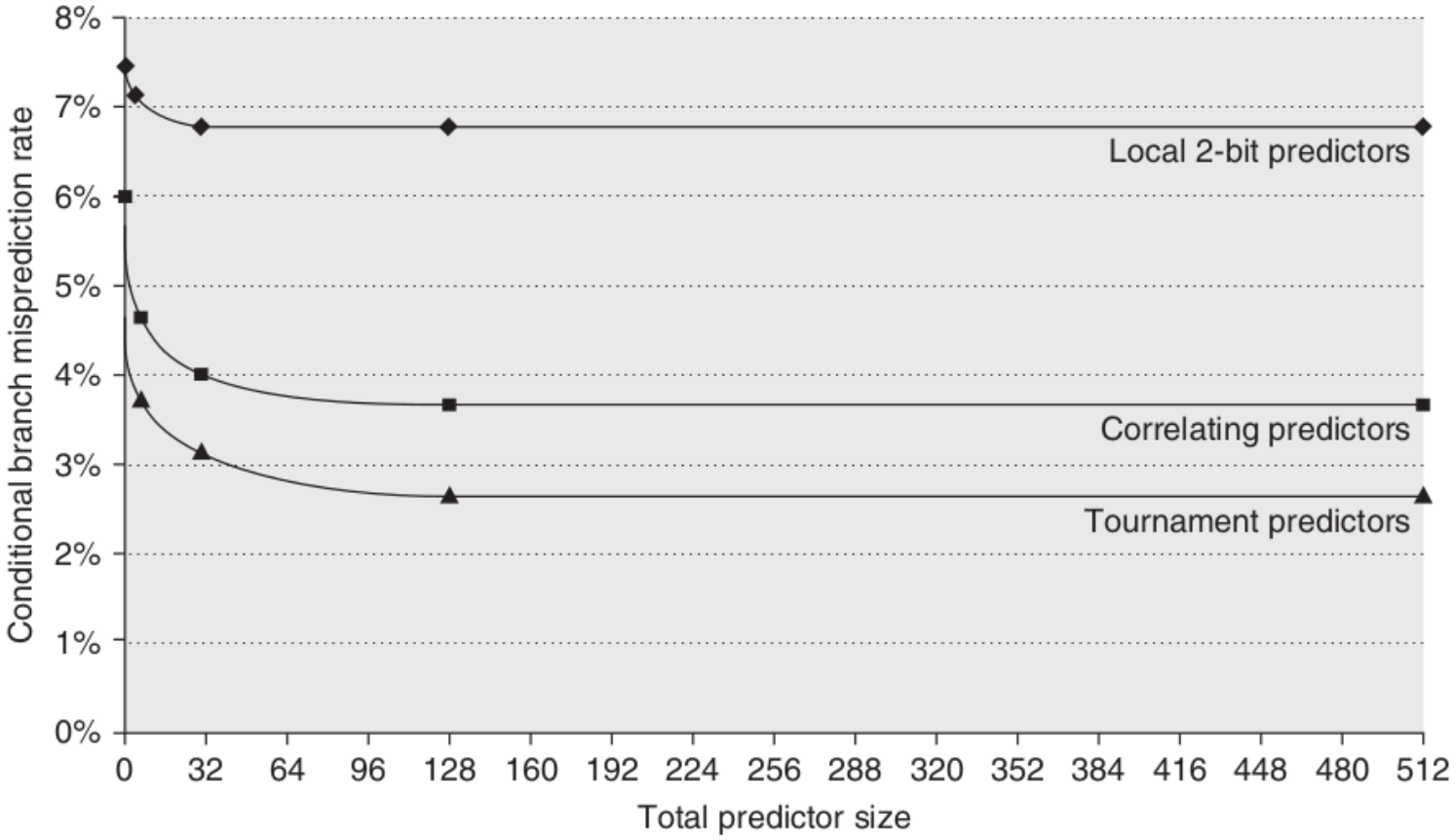
- How about looking at the recent behavior of other branches rather than just the branch we are trying to predict
  - An (m,n) predictor uses the behavior of the last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor for a single branch
    - A (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors in predicting a particular branch

Prediction Accuracy



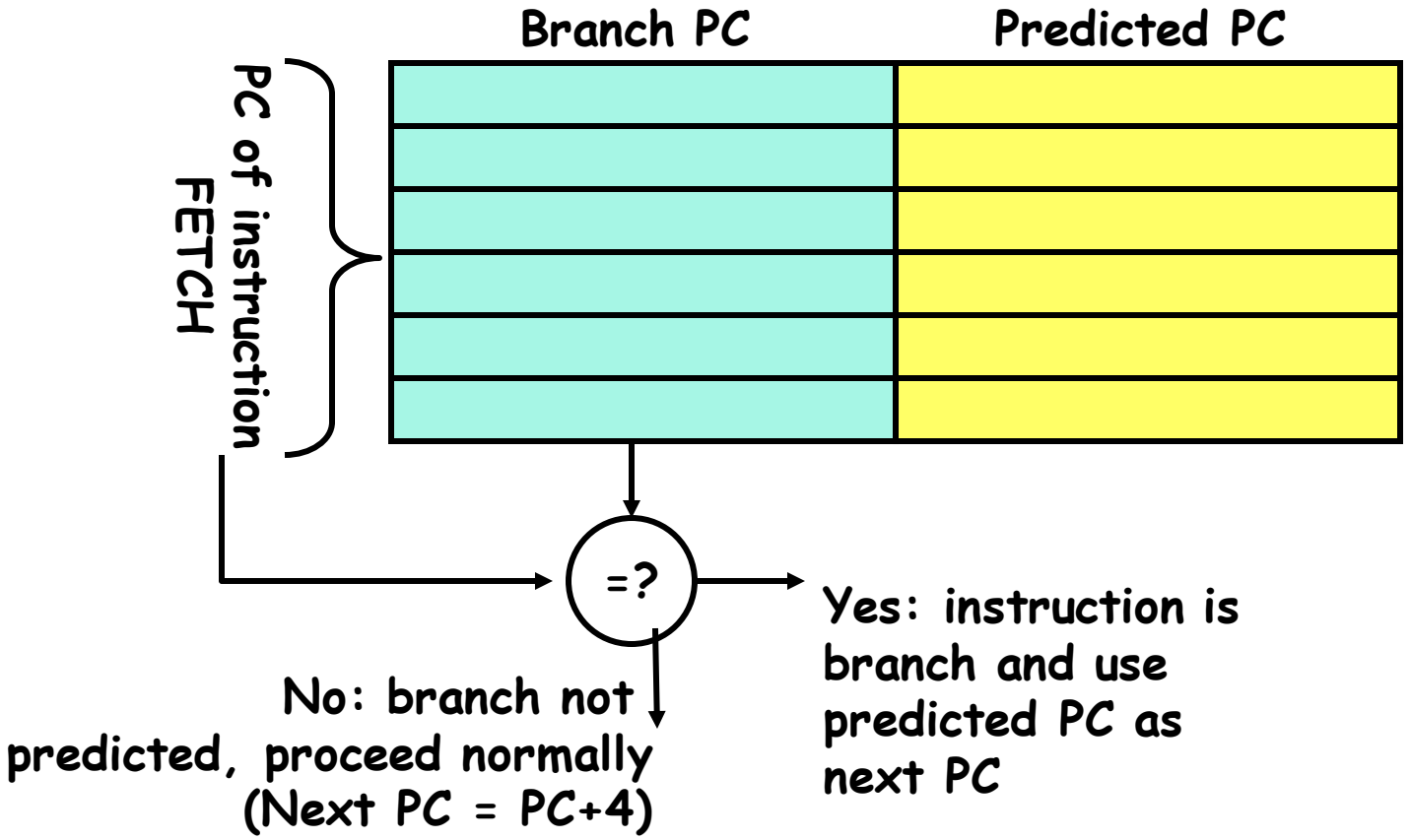
Tournament Predictor

- Tournament predictors use multiple predictors, usually one based on global information and one based on local information, and combining them with a selector

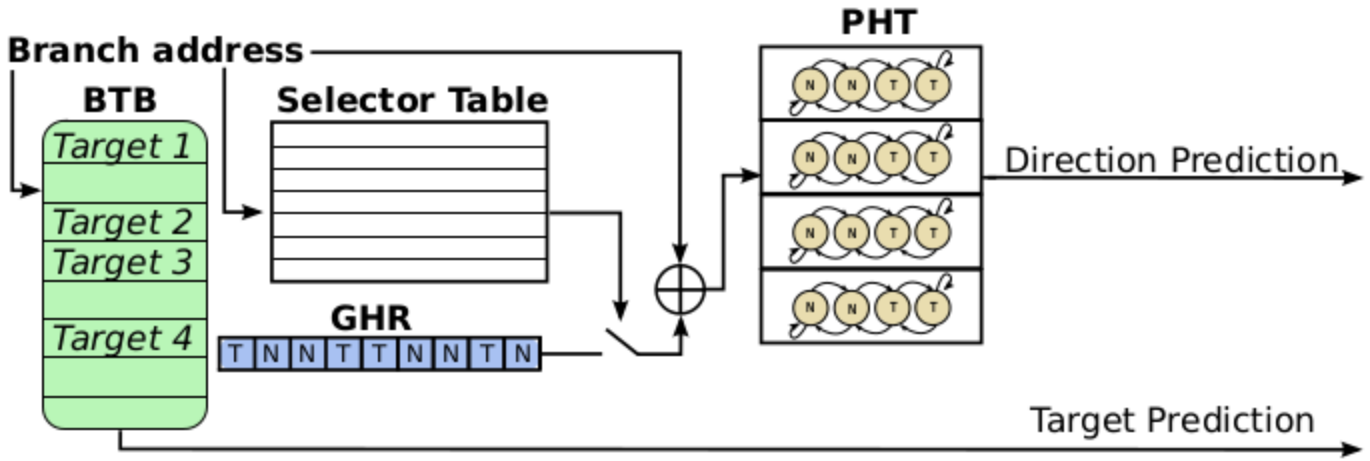


Branch Target Buffer

- Branch target buffer (BTB) caches the destination PC or destination instruction for a branch

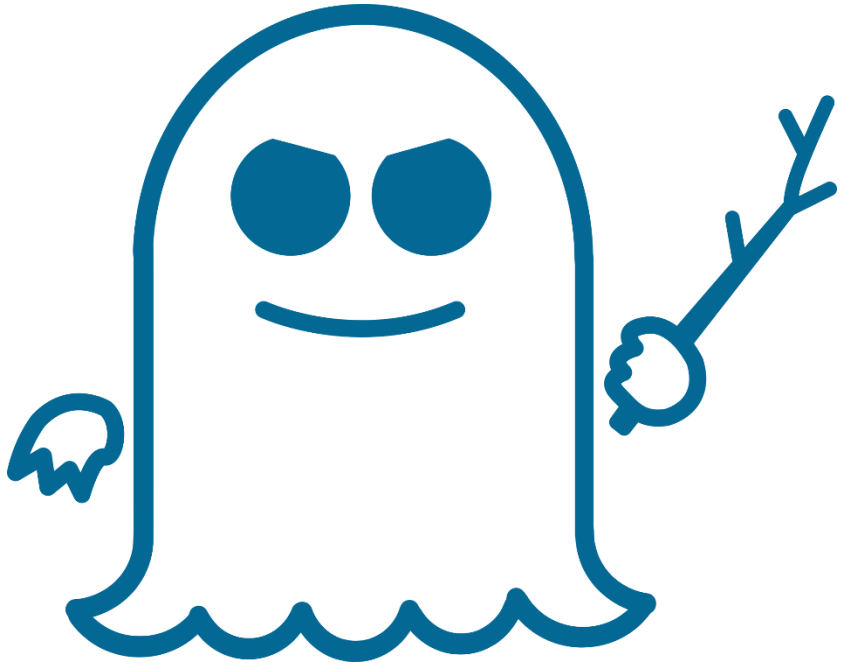


A Combined Branch Predictor



Spectre

- Speculative execution improves performance but introduces new security vulnerabilities
  - Out-of-order execution will also introduce vulnerabilities (Meltdown)



Go to read “How the Spectre and Meltdown Hacks Really Worked” to learn a general idea

Next Lecture

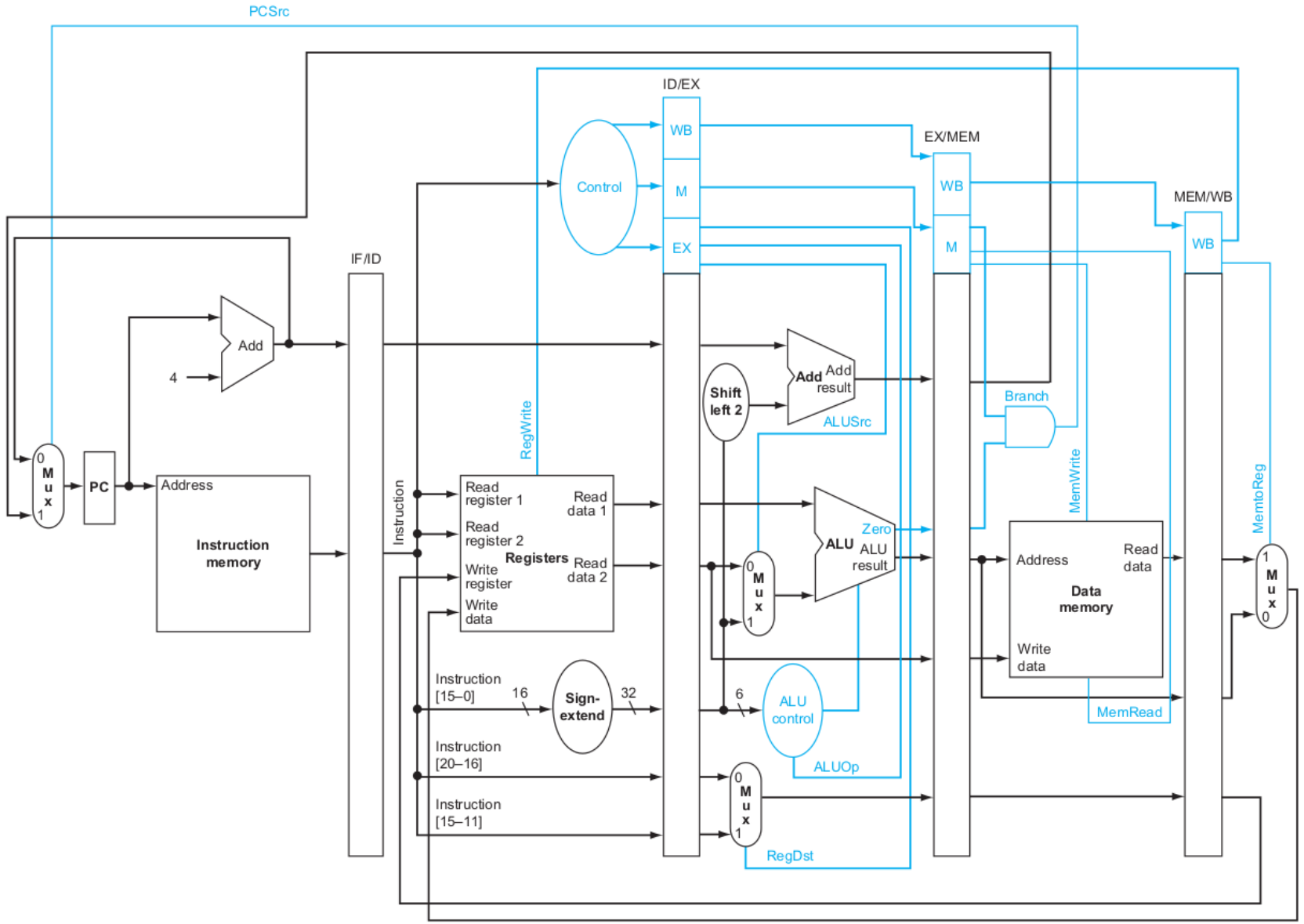
- Section 4.10 – 4.11
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

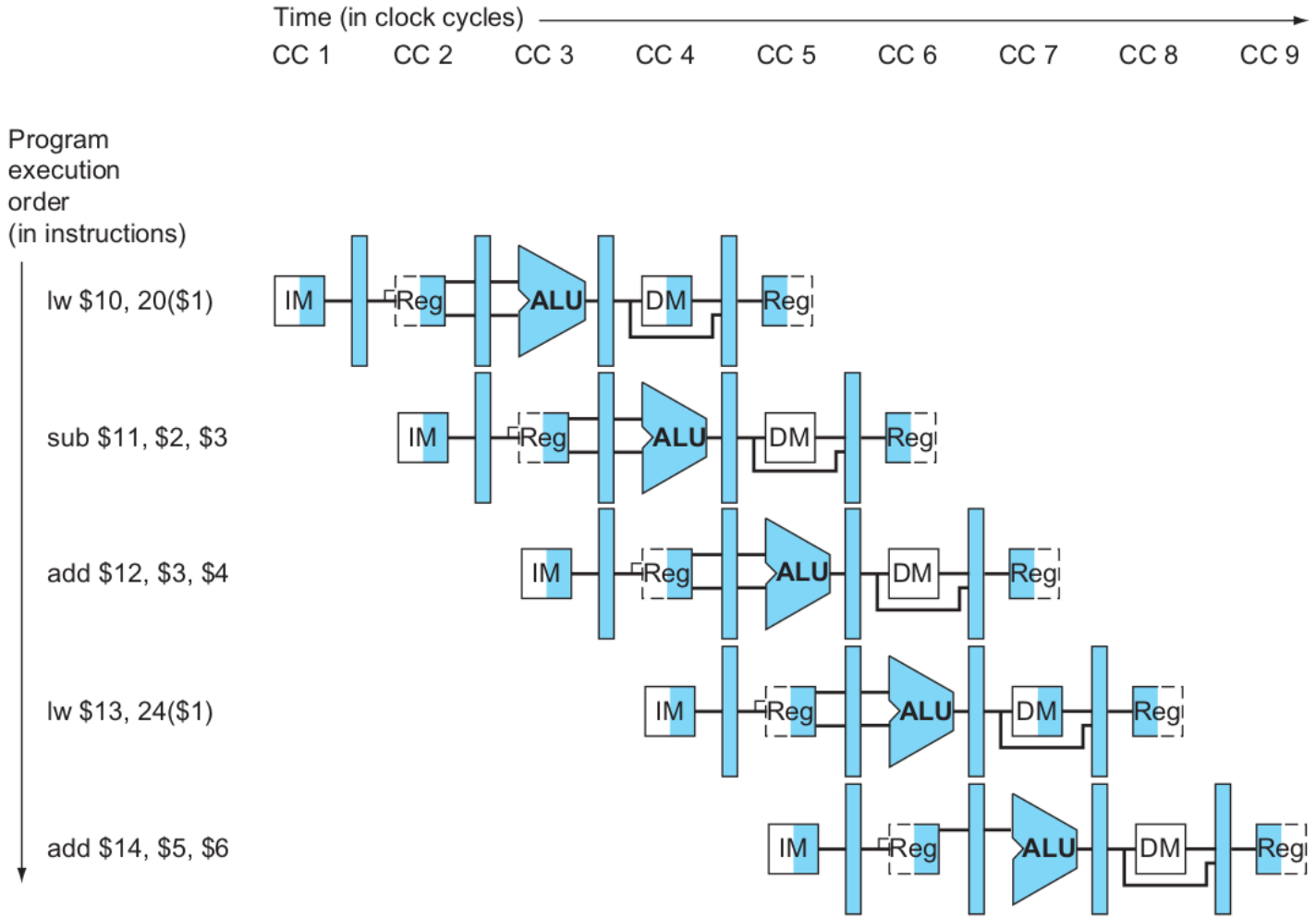
10. Data Hazards

Zhenkai Zhang

Pipelined Datapath & Control Review



Overlapping Instruction Execution



Pipeline Hazards

- There are situations, called hazards, that prevent the next instruction from executing during its designated clock cycle
  - Structural hazards
    - They arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution
      - In modern processors, structural hazards occur primarily in special purpose functional units that are less frequently used (e.g., floating point divide)
  - Data hazards
    - They arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - Control hazards
    - They arise from the pipelining of branches and other instructions that change the PC

Data Dependence

- An instruction j is data dependent on instruction i if either of the following holds
  - Instruction i produces a result that may be used by instruction j

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

The arrow points from an instruction that must precede the instruction that the arrowhead points to

  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i
    - A chain of dependences of the first type between the two instructions
    - This dependence chain can be as long as the entire program

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
sw  $t2, 12($t0)
```

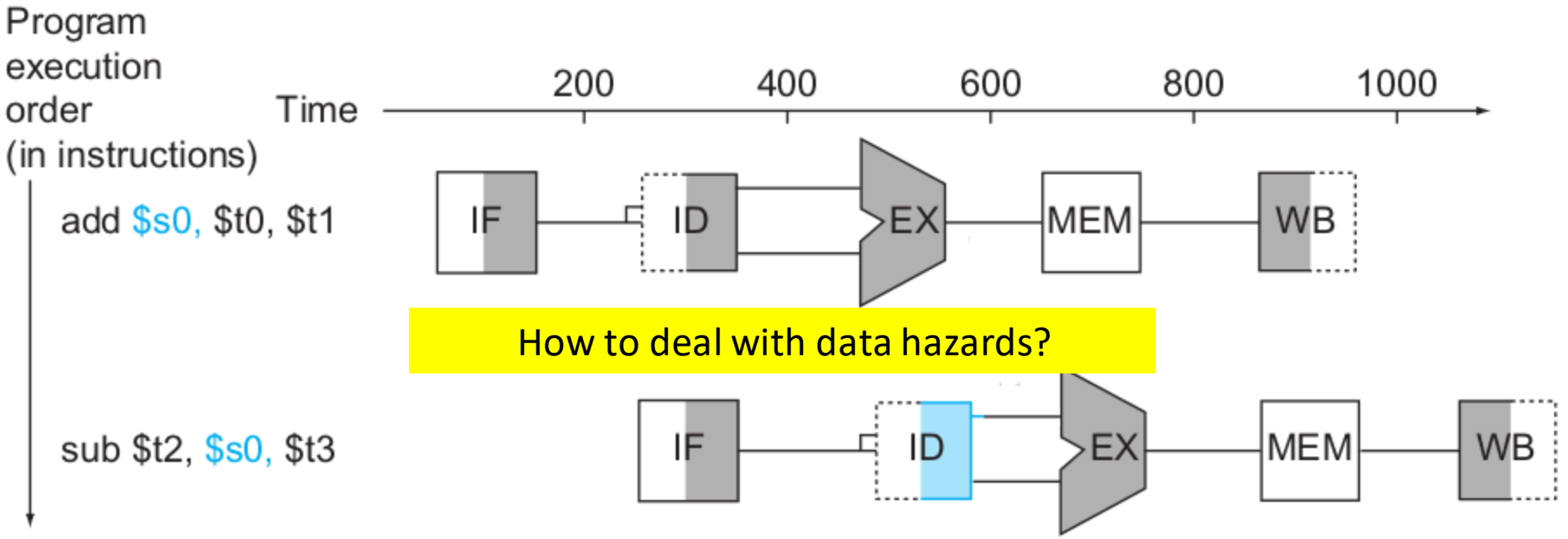
If two instructions are data dependent, they must execute in order

(Read-after-Write) Data Hazard

- A planned instruction cannot execute in the proper clock cycle since data that is needed to execute the instruction is not yet available

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

Can these two instructions be executed freely in the pipeline?



Scheduling Code

- A data dependence only indicates that a data hazard is possible
  - If not closely spaced, data hazards will not arise
- Scheduling code can avoid a hazard without altering a dependence,
  - Such scheduling can be done both by the compiler and/or by the hardware

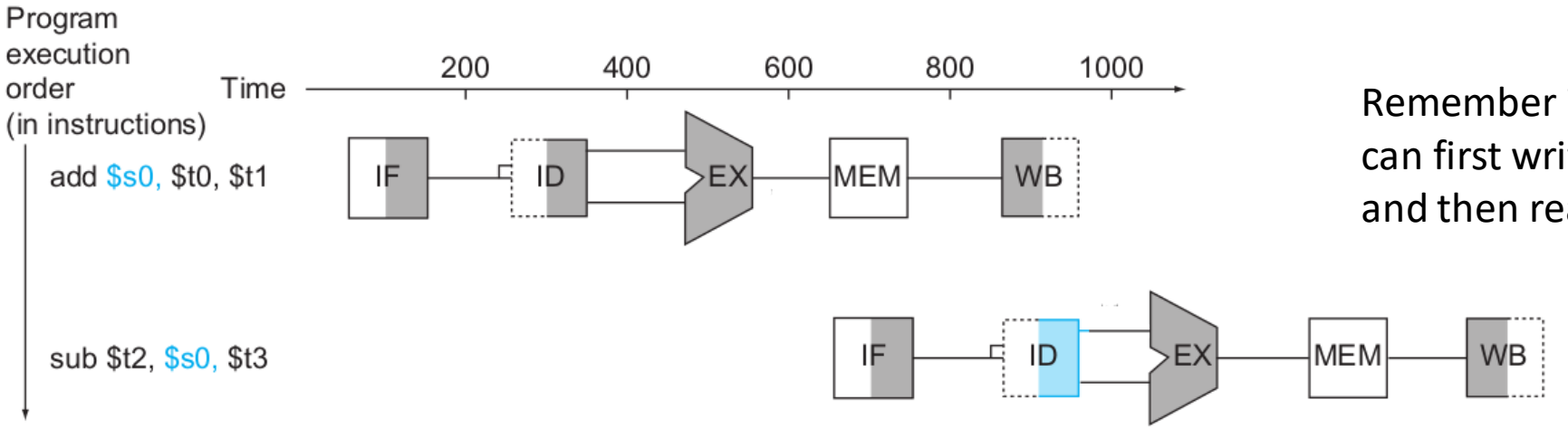
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t4, $t0, $t3
add $t5, $t1, $t7
and $t6, $t9, $t3
```

```
add $s0, $t0, $t1
add $t4, $t0, $t3
add $t5, $t1, $t7
and $t6, $t9, $t3
sub $t2, $s0, $t3
```

Pipeline Interlock

- The simplest hardware way is to add some logic, called a pipeline interlock, to preserve the correct execution pattern
  - In general, a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared
  - Stall is to pause parts of the pipeline by inserting `nop`

Can we do better?

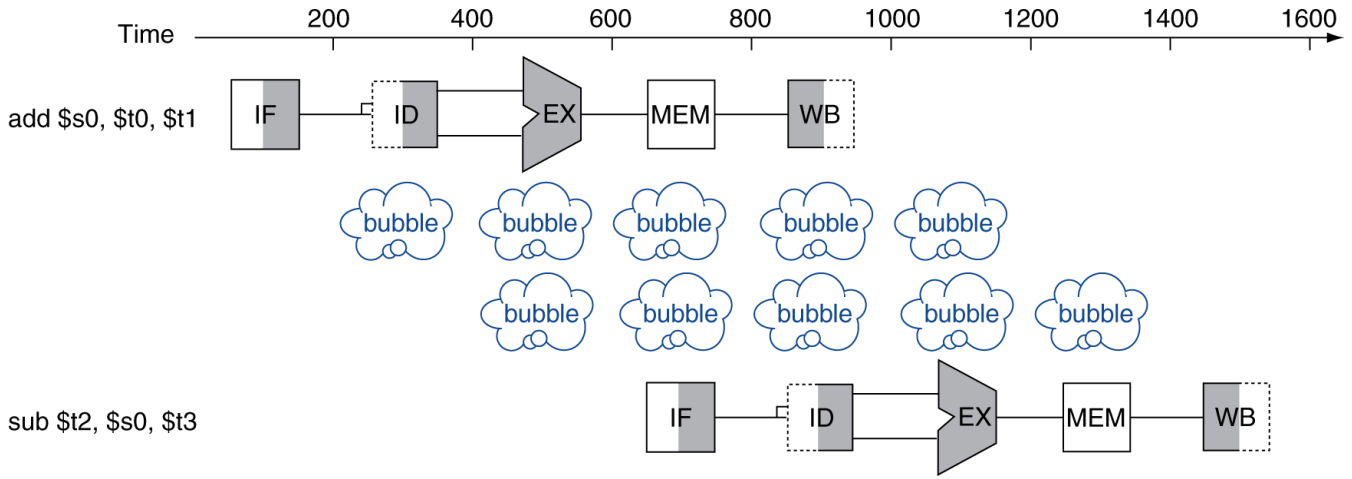


Remember in a clock cycle we can first write the register file and then read the register file

In this case, the interlock stalls the pipeline, beginning with the `sub` instruction that wants to use the data until the `add` instruction produces it

How to Stall?

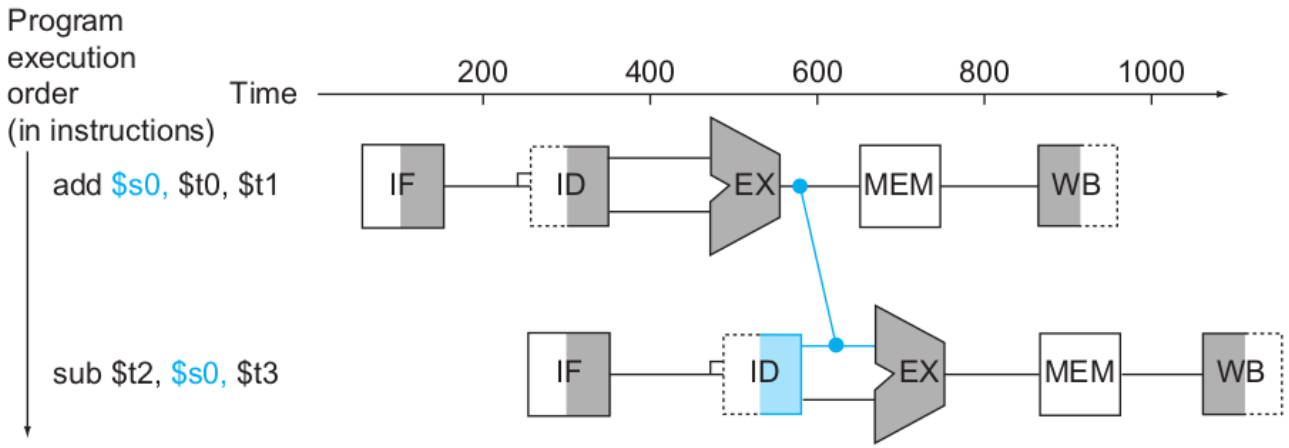
- If we can identify hazards in the ID stage, we can change the EX, MEM, and WB control fields of the ID/EX pipeline register to 0
  - We care about RegWrite and MemWrite (others can be don't care)
  - Actually, we also care about PC and IF/ID pipeline register
    - The instruction following the one who is encountering the hazard will continue to be read using the same PC
    - The IF/ID pipeline register will continue to give the same fields to the ID stage
- Pipeline stalls are often called "bubbles"



Why Wait?

- Observe that the `$s0` is not really needed by the `sub` until after the `add` actually produces it
  - If the result can be moved from the pipeline register where the `add` stores it to where the `sub` needs it, then the need for a stall can be avoided

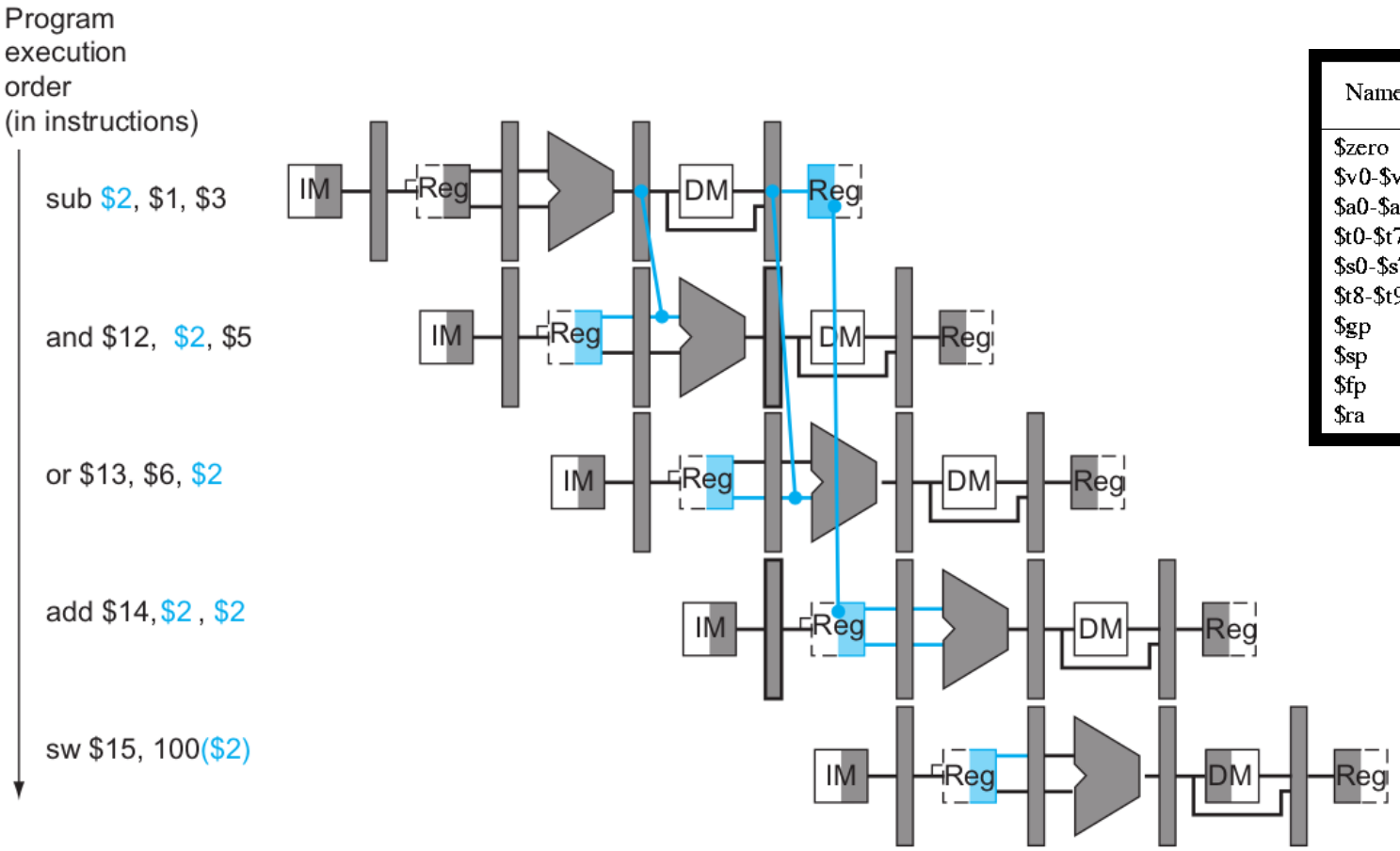
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```



Forwarding Results in Pipeline Registers

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

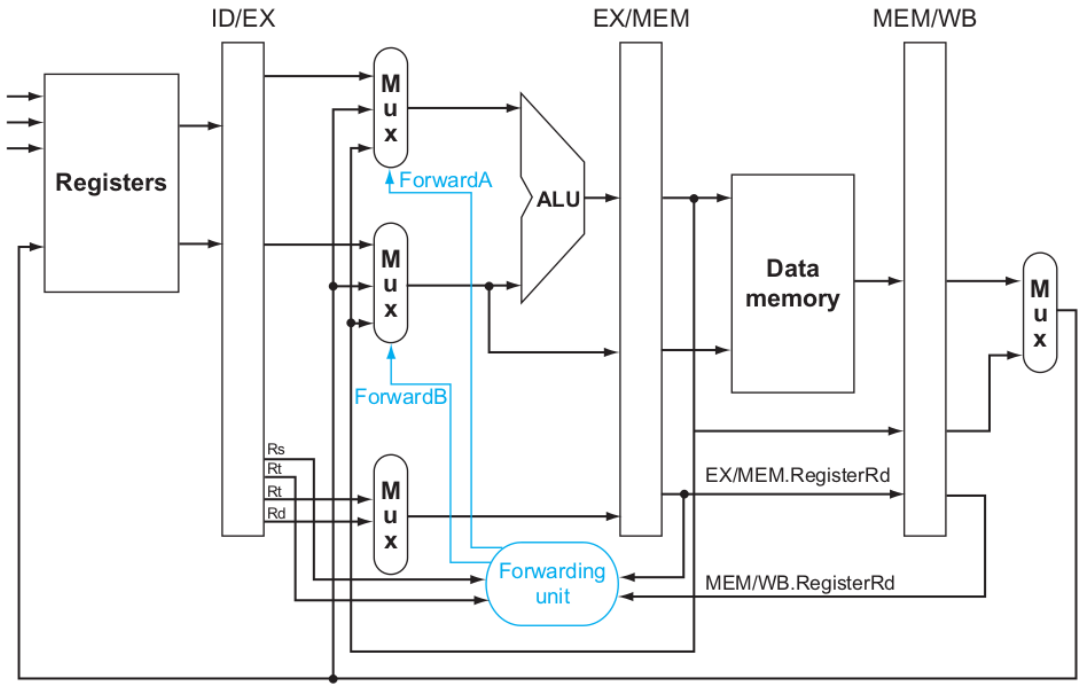


Name	Register	Usage	Preserved on call?
$zero	$0	the constant value 0	n.a.
$v0-$v1	$2-$3	values for results and expression evaluation	no
$a0-$a3	$4-$7	arguments	yes
$t0-$t7	$8-$15	temporaries	no
$s0-$s7	$16-$23	saved	yes
$t8-$t9	$24-$25	more temporaries	no
$gp	$28	global pointer	yes
$sp	$29	stack pointer	yes
$fp	$30	frame pointer	yes
$ra	$31	return address	yes

BTW, register names and numbers mapping in MIPS

Forwarding to ALU

- The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs

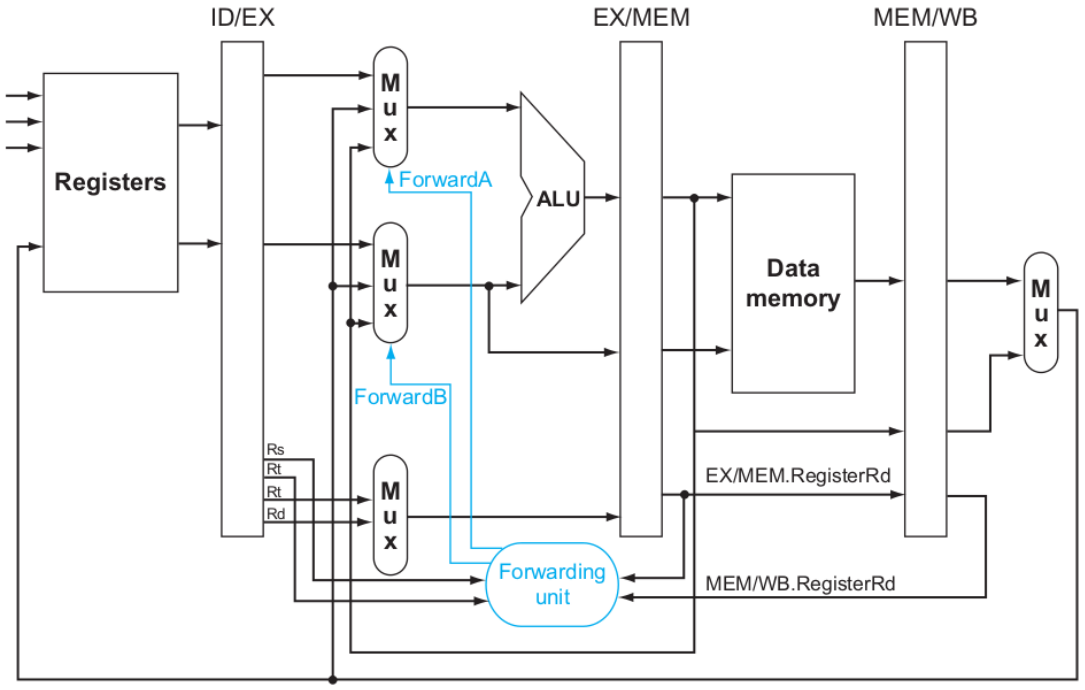


If

$$\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs1}$$
$$\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs2}$$
$$\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs1}$$
$$\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs2}$$

we want to use the forwarded value

Forwarding's Control



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
```

Why?

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

```
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
```

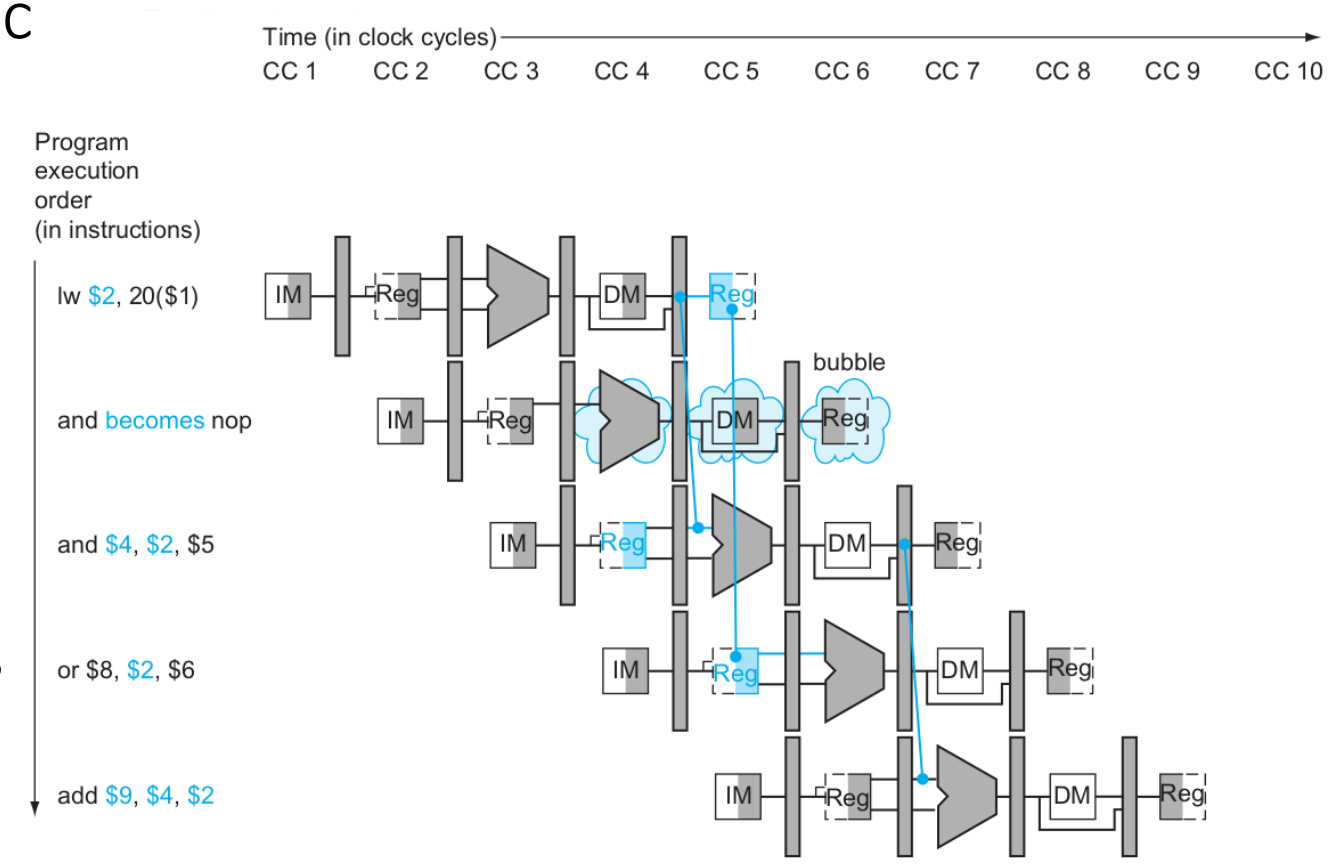
```
add  $1, $1, $2
add  $1, $1, $2
add  $1, $1, $2
```

Tricky

ForwardA should be 10 or 01?

Load-Use Data Hazard

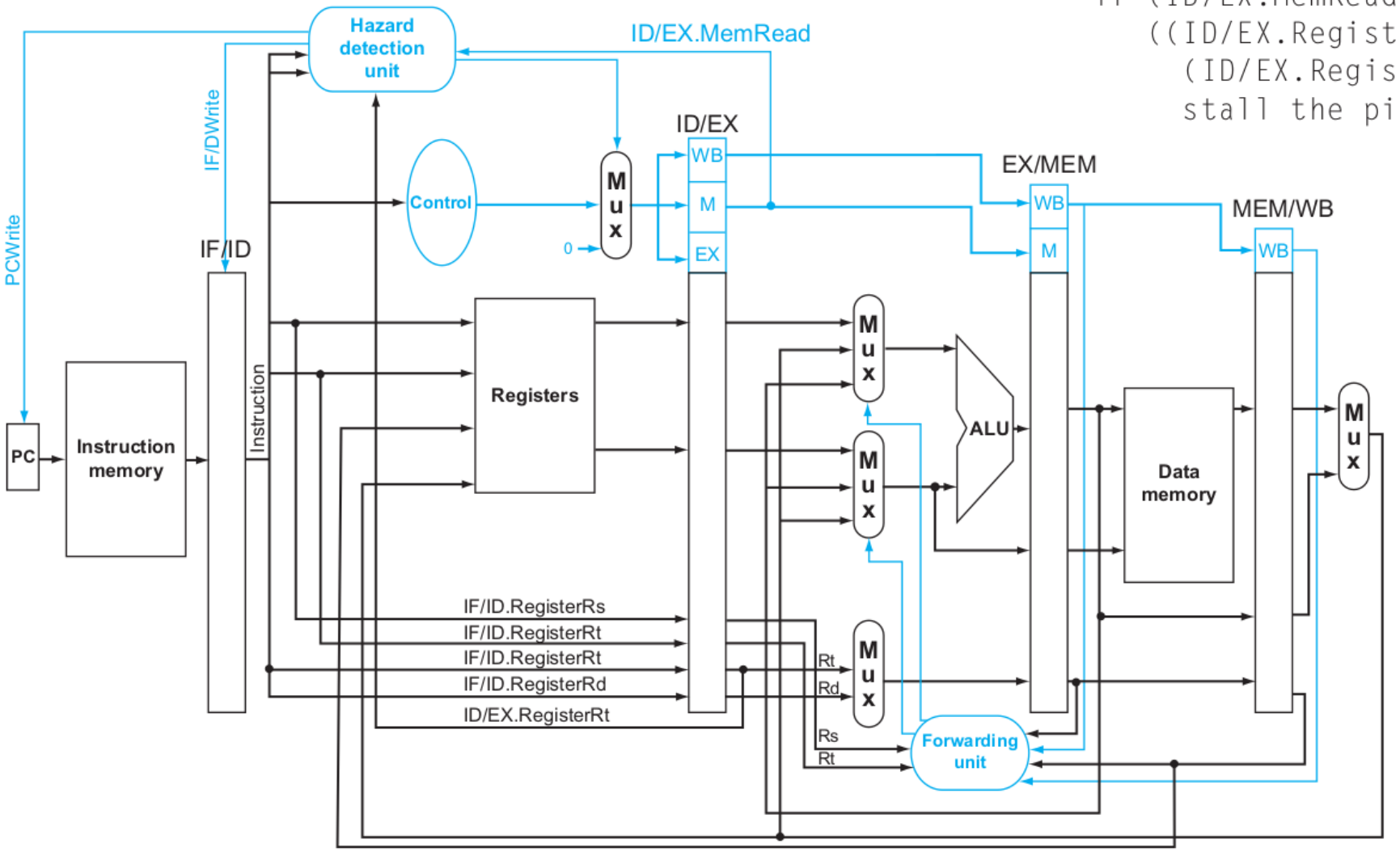
- Not all potential data hazards can be handled by forwarding
- Load-use data hazard is a specific form of data hazard
  - The data being loaded by a load instruction has not yet become available when it is needed by another instruction
- We need a stall even with forwarding when an R-format instruction following a load tries to use the data



Hazard Detection Unit

Type	31 26	25 21	20 16	15 11	10 06	05 00
R-Type	opcode	$rs	$rt	$rd	shamt	funct
I-Type	opcode	$rs	$rt	imm		
J-Type	opcode	address				

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
     stall the pipeline
```

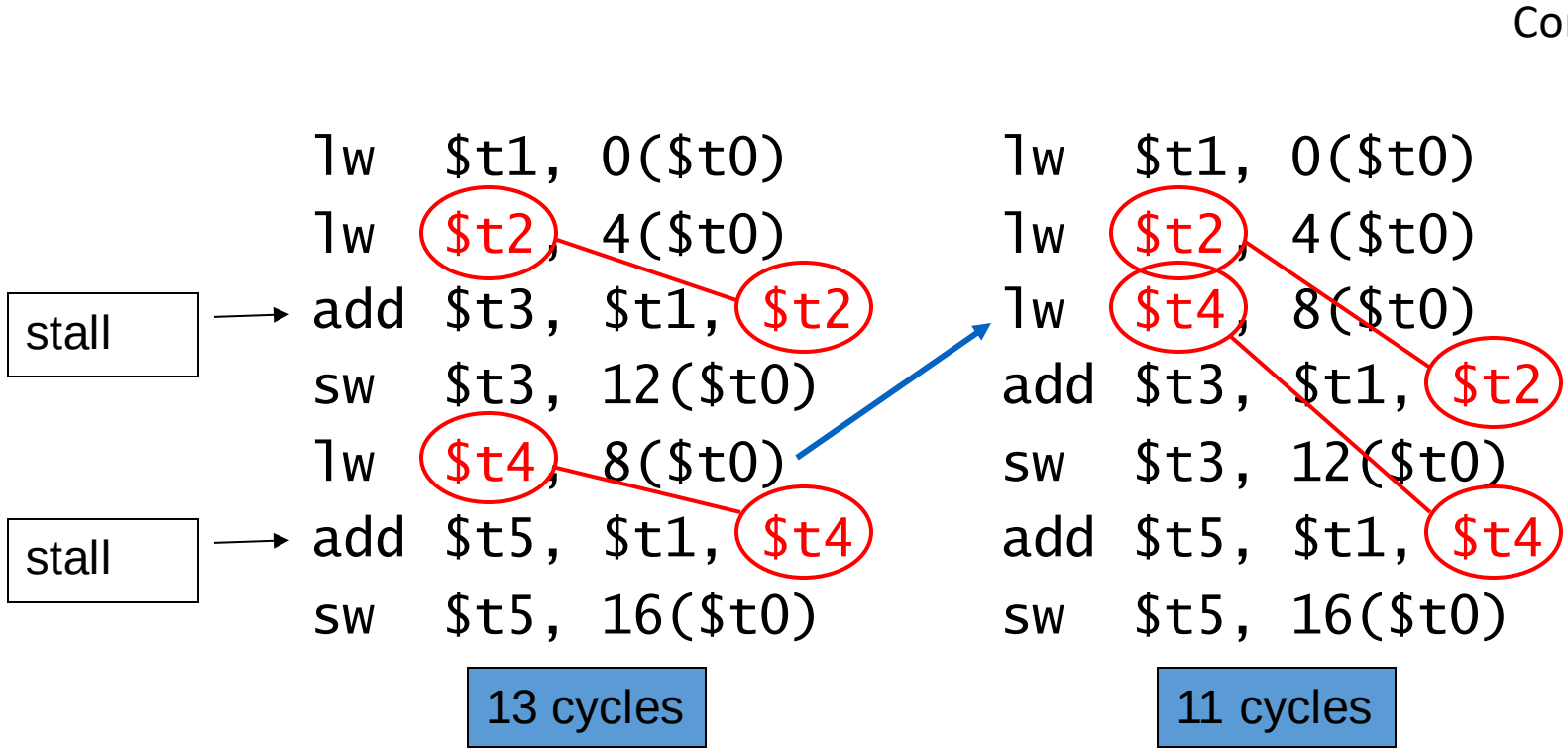


Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

Compiled code for C code

```
A = B + E;
C = B + F;
```



Forwarding for Store Instructions

- If no forwarding, will pipeline stalls be needed?
  - If so, can forwarding get rid of stalls? Or more like load-use?

```
add   $15, $1, $2
sw    $15, 100($3)
```

- How to implement forwarding here?
  - Where to put a mux?
  - Which pipeline register should be focused on?
  - How to check if forwarding is needed?

WAW and WAR Data Hazards

- Other than RAW data hazards, there are two more types
  - WAW (write after write)
    - A later instruction tries to write an operand before an earlier instruction writes it
      - The writes end up being performed in the wrong order, leaving the value written by the earlier instruction rather than the value written by the later instruction in the register
    - WAW hazards are present only in pipelines that write in more than one pipeline stage or allow an instruction to proceed even when a previous instruction is stalled
  - WAR (write-after read)
    - A later instruction tries to write an operand before an earlier instruction
      - The earlier instruction incorrectly gets the new value written by the later instruction
    - A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline and other instructions that read a source late in the pipeline, or when instructions are reordered

Register renaming can solve them!

They do not exist in the learnt classic 5-stage in-order pipeline

Next Lecture

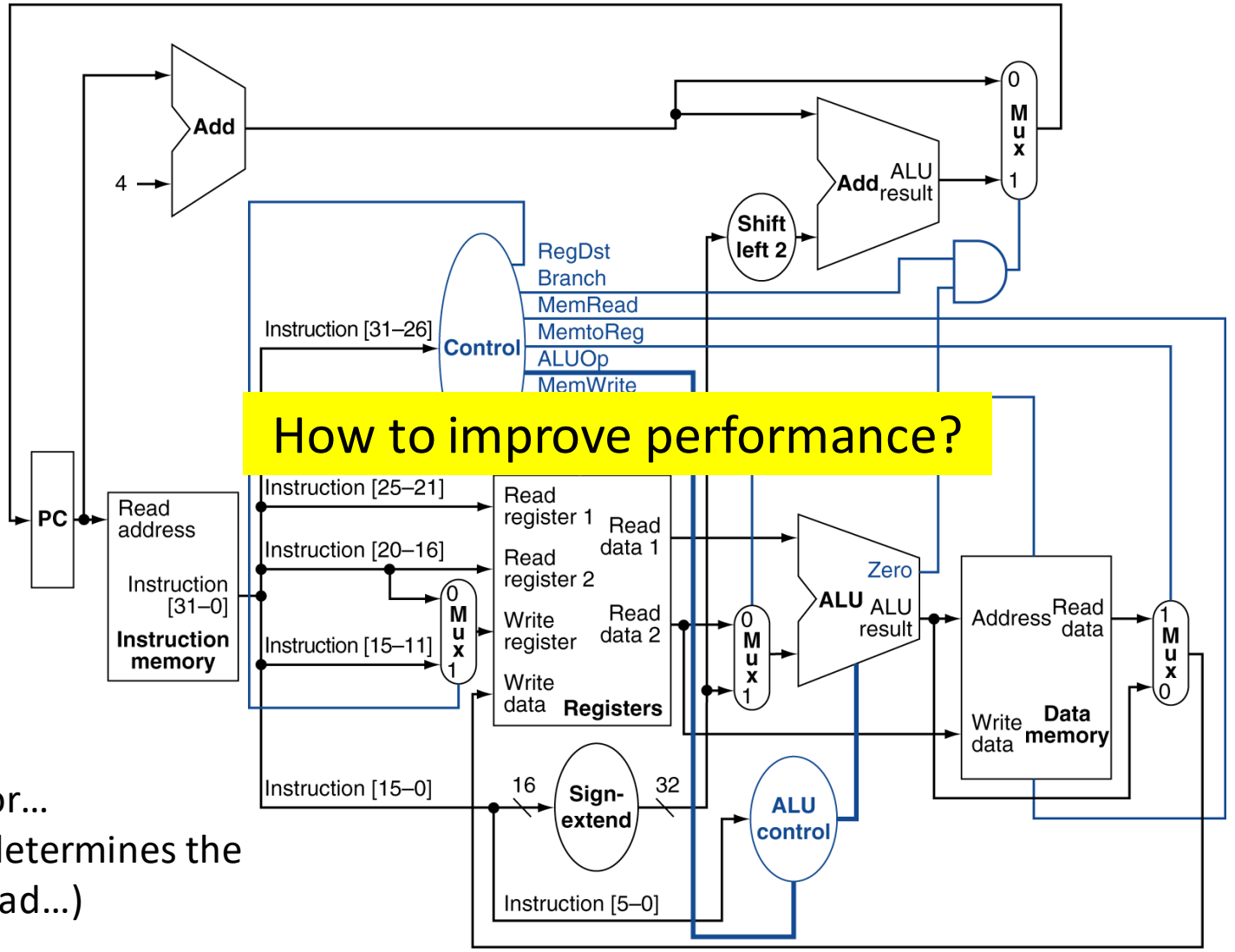
- Section 4.9
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

9. Pipelining

Zhenkai Zhang

Single-Cycle Design Review

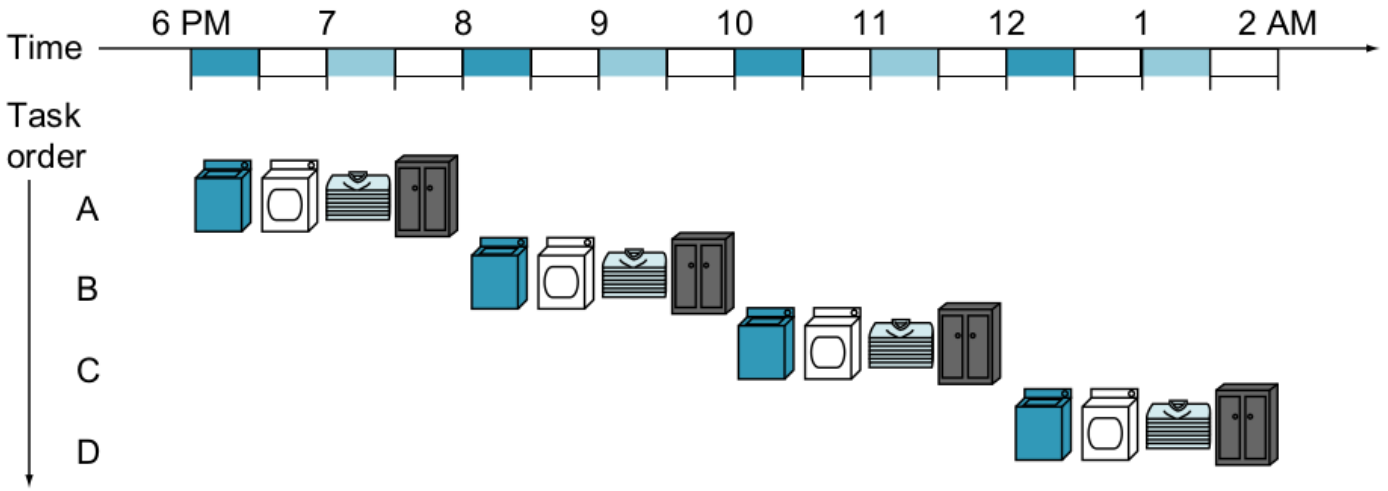


Performance is poor…
The longest delay determines the clock cycle time (load…)

How to Quickly Do Laundry?

Four loads:

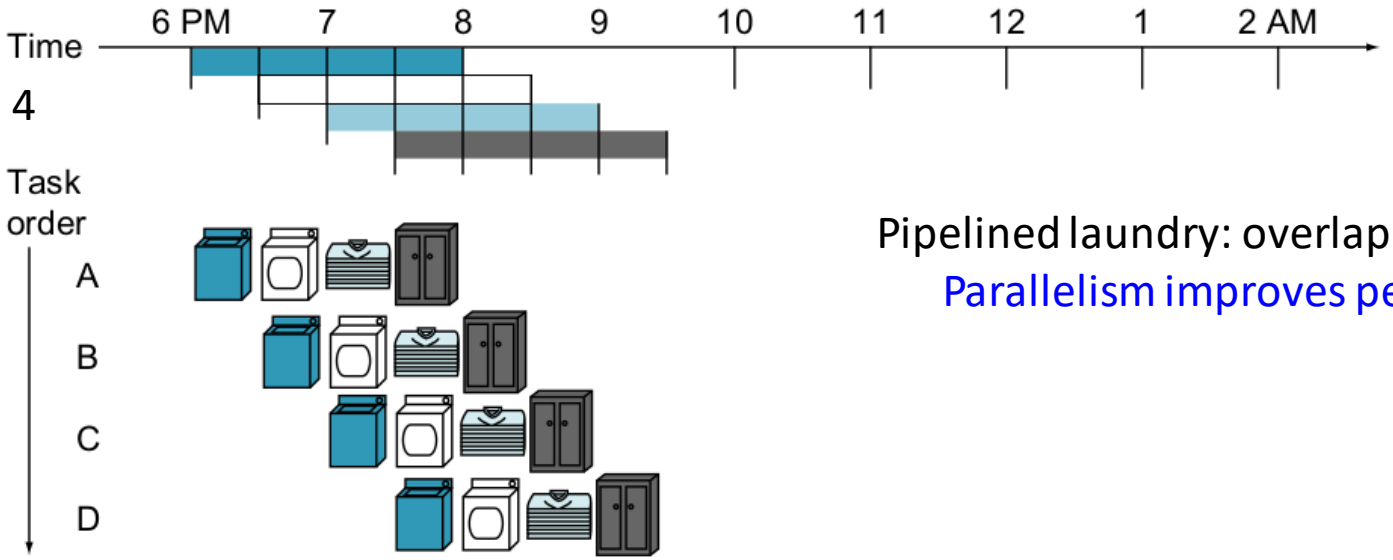
Speedup = 8/3.5 = 2.3



Non-stop:

Speedup = 2n/(0.5n + 1.5) ≈ 4

= the number of stages



Pipelined laundry: overlapping execution

Parallelism improves performance

Pipelining

- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution
  - It takes advantage of parallelism that exists among the actions needed to execute an instruction
- A pipeline consists of many stages (called pipeline stages/segments)
  - Each stage in the pipeline completes a part of an instruction
  - Different stages are completing different parts of different instructions in parallel
  - The stages are connected one to the next to form a pipe
    - Instructions enter at one end, progress through the stages, and exit at the other end, just as when you are doing laundry

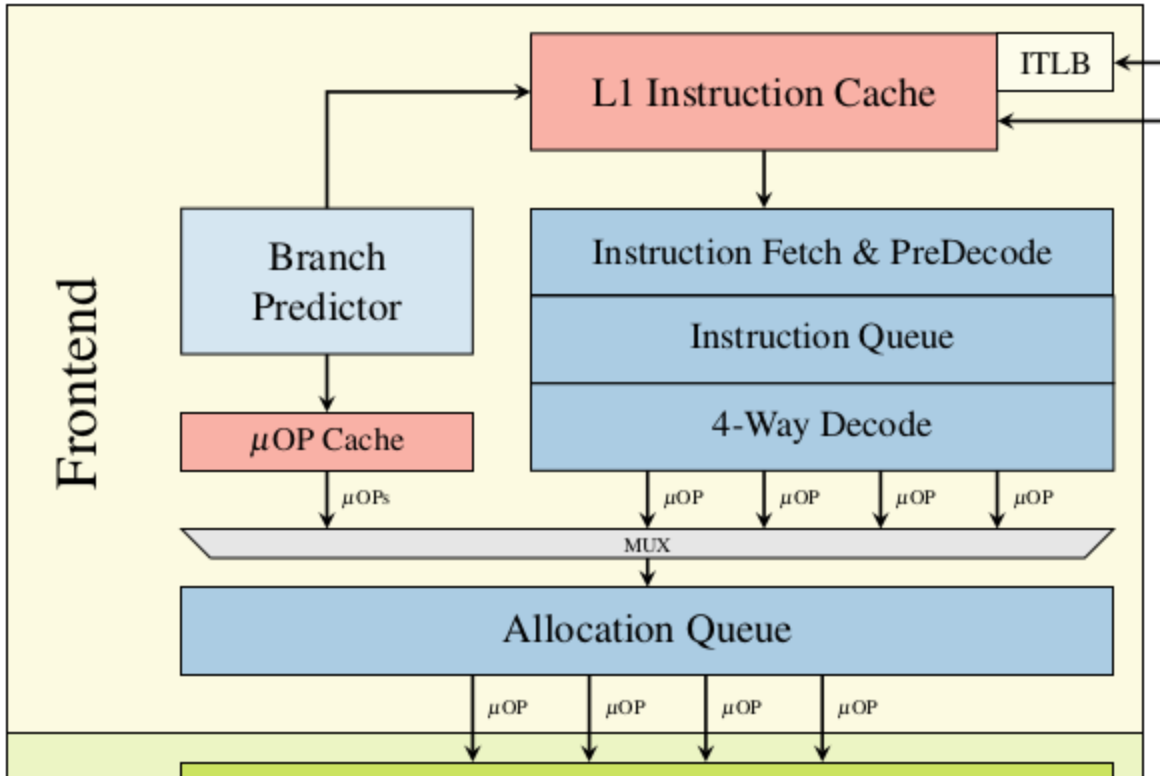
Why RISC Good for Pipelining

- All RISC architectures are characterized by a few key properties
  - Instructions have the same size
    - Easier to fetch in one cycle and convenient for decoding
  - The instruction formats are few in number and regular in general
    - Can decode and read registers in one step
  - Operations on data apply to data in registers and typically change the entire register
    - Can simplify the control
  - The only operations that affect memory are load and store instructions that move data from memory to a register or to memory from a register
    - Can calculate memory address and access memory in fixed stages

How about CISC like x86?

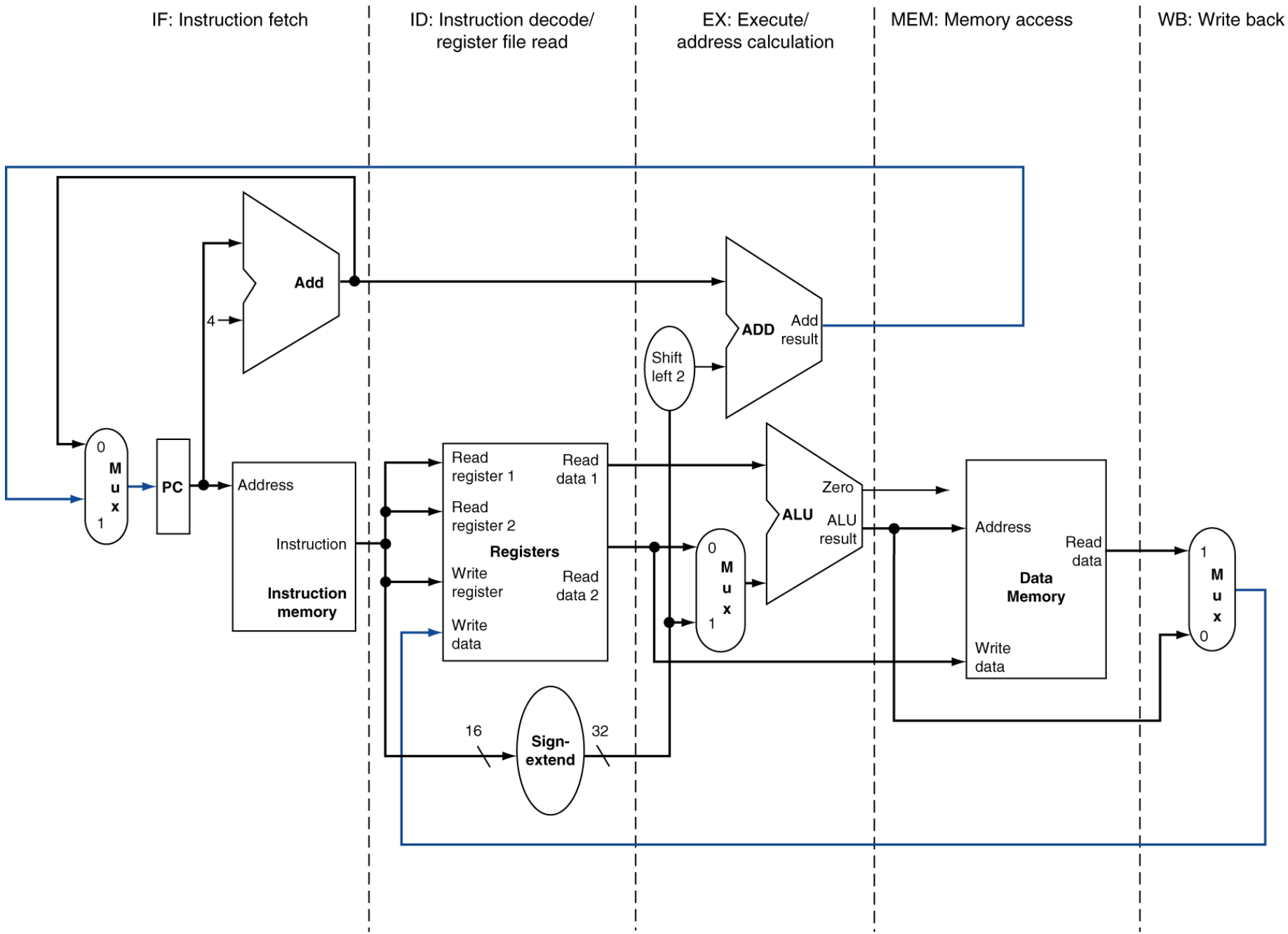
x86 Frontend

- Intel fetches x86 instructions and translates them into internal MIPS-like instructions that Intel calls micro-operations (µops)



Classic Five-Stage Pipeline

- The single-cycle datapath can be separated into five pieces, with each piece named corresponding to a stage of instruction execution
  - IF: Instruction fetch
  - ID: Instruction decode and register file read
  - EX: Execution or effective address calculation
  - MEM: Data memory access
  - WB: Write back

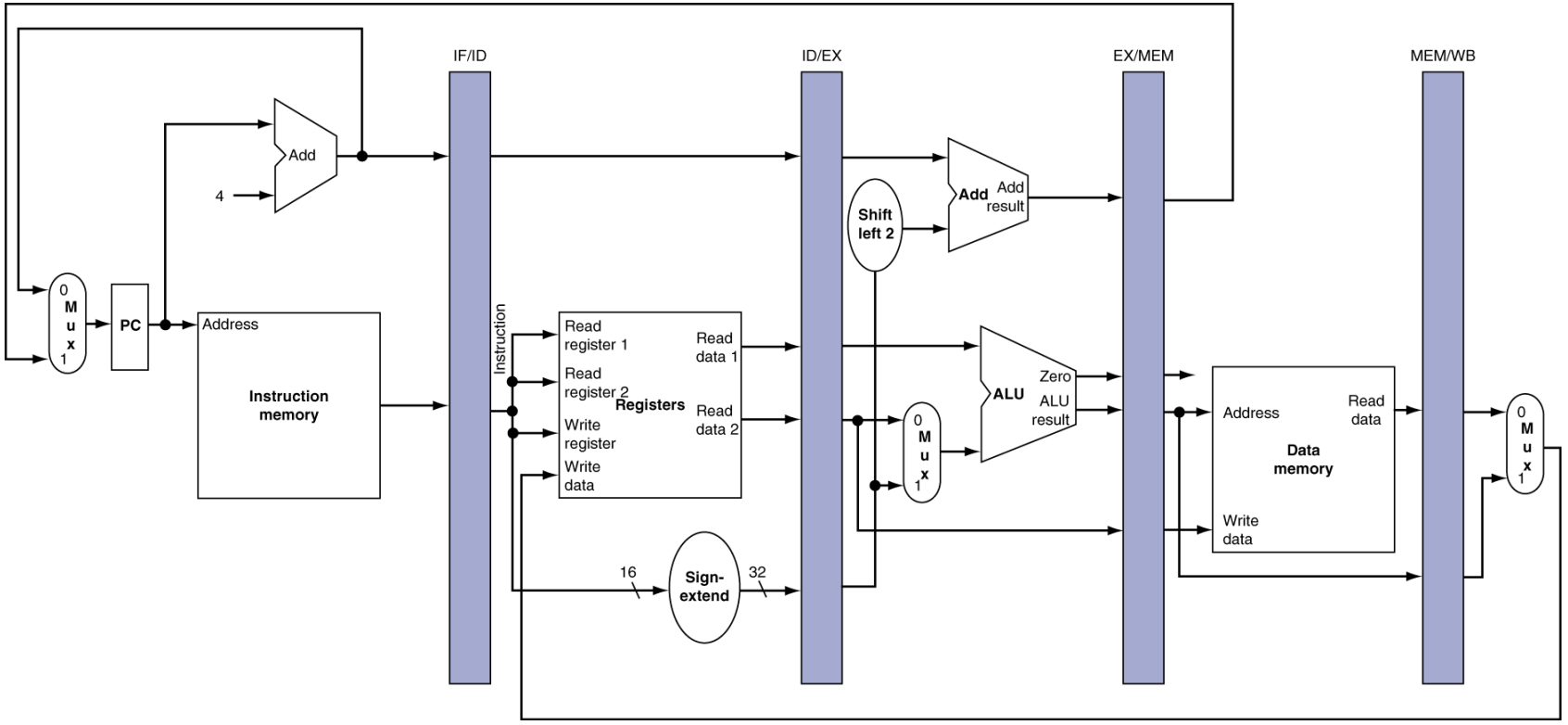

IF: Instruction fetch
ID: Instruction decode/
register file read
EX: Execute/
address calculation
MEM: Memory access
WB: Write back
Add
4
M
u
x
0
1
PC
Address
Instruction
Instruction
memory
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
Shift
left 2
ADD
Add
result
ALU
Zero
ALU
result
Address
Read
data
Data
Memory
Write
data
16
Sign-
extend
32

As Simple As Splitting the Datapath?

- We want to balance the stages
  - If not balanced, speedup is less (clock cycle depends on the longest stage)
- We must also ensure that instructions in different stages of the pipeline do not interfere with one another

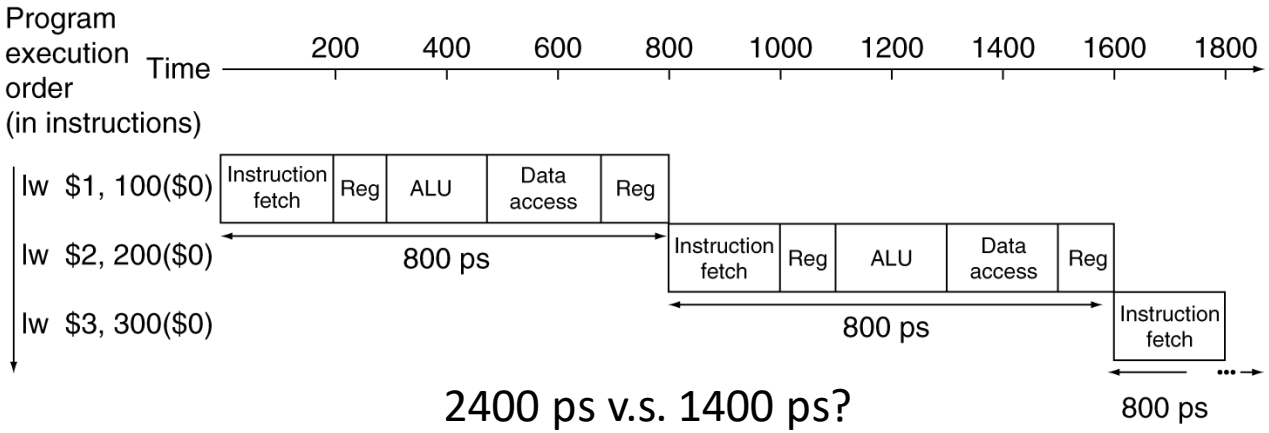
Pipeline Registers

- This separation is done by introducing pipeline registers between successive stages of the pipeline
  - At the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle

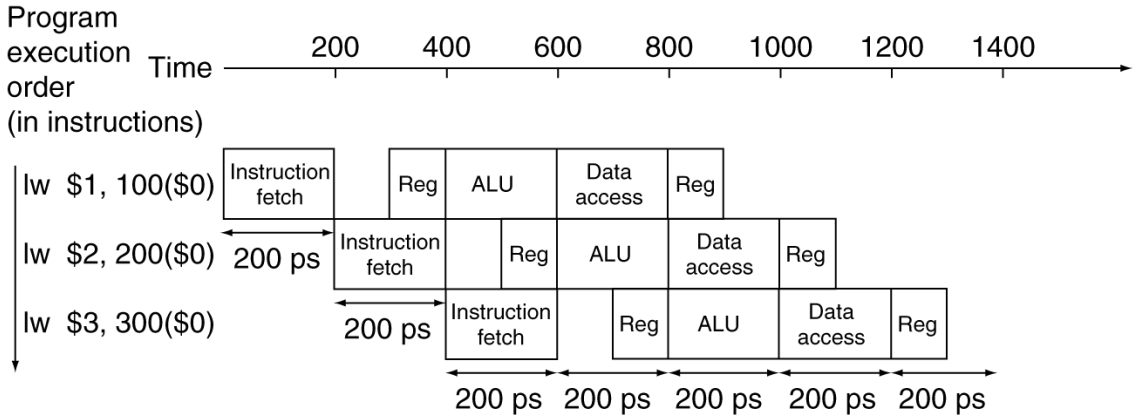


Single-Cycle v.s. Pipelined Performance

- Assume operation times for the major functional units
  - 200 ps for memory access, 200 ps for ALU operation, and 100 ps for register file read or write



2400 ps v.s. 1400 ps?



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

What would happen if the number of instructions are greatly increased?

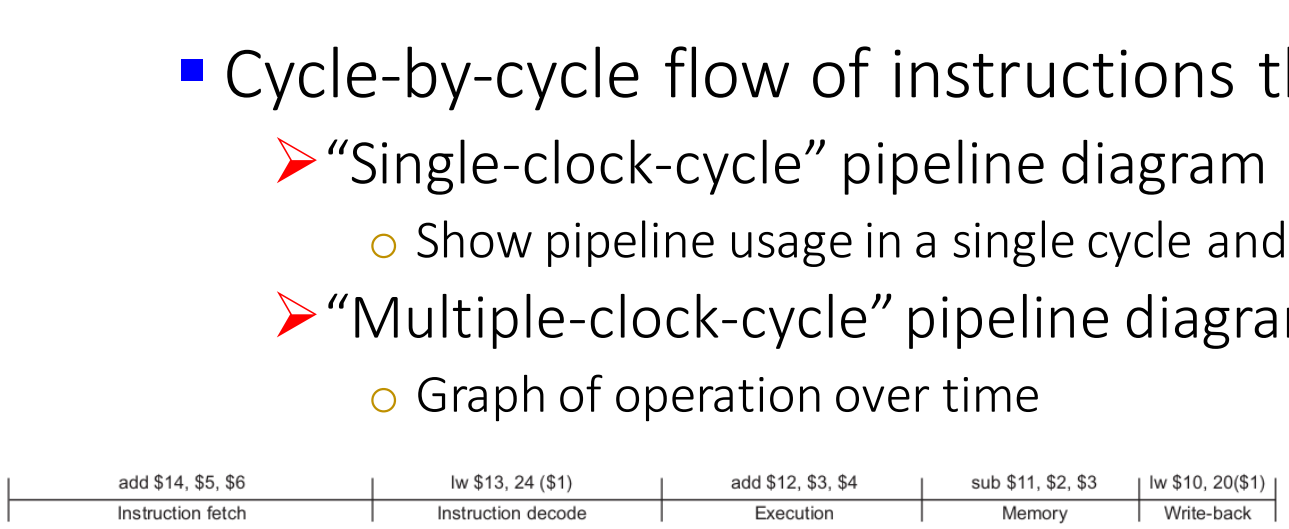
Every 200 ps, an instruction can start executing

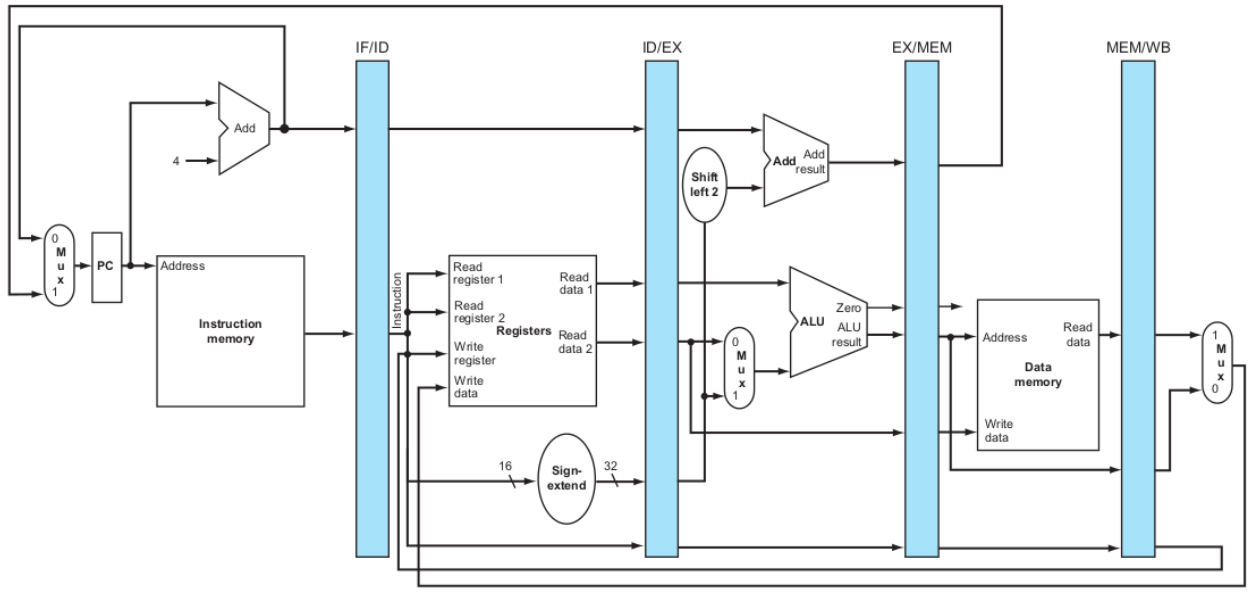
800n v.s. 200n+800

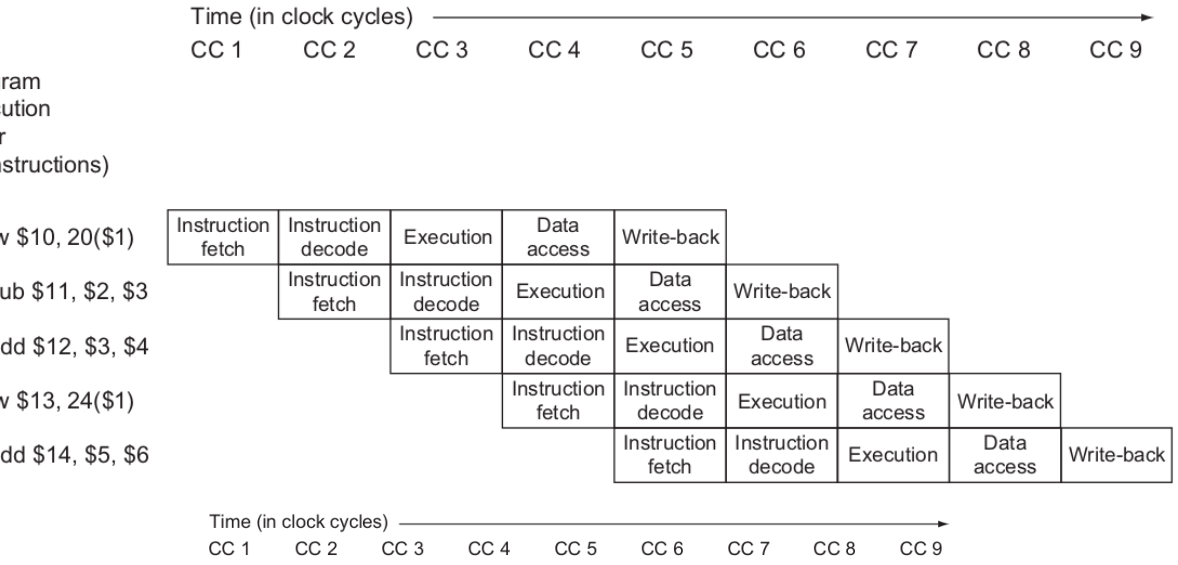
Why good?
Throughput, not time for one instruction
-Pipeline register overhead
-Imperfect splitting

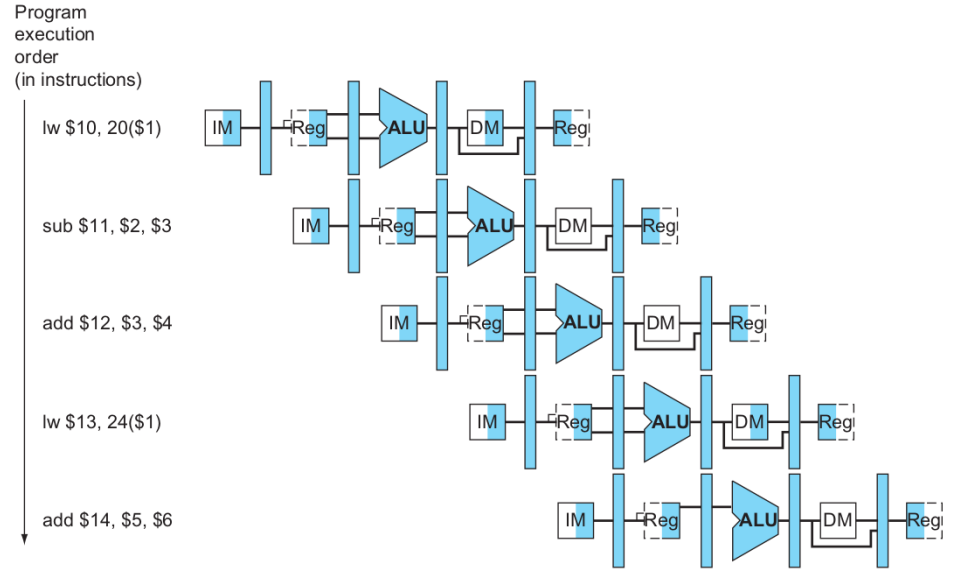
Graphically Representing Pipelines

- Cycle-by-cycle flow of instructions t
  - "Single-clock-cycle" pipeline diagram
    - Show pipeline usage in a single cycle and
  - "Multiple-clock-cycle" pipeline diagra
    - Graph of operation over time

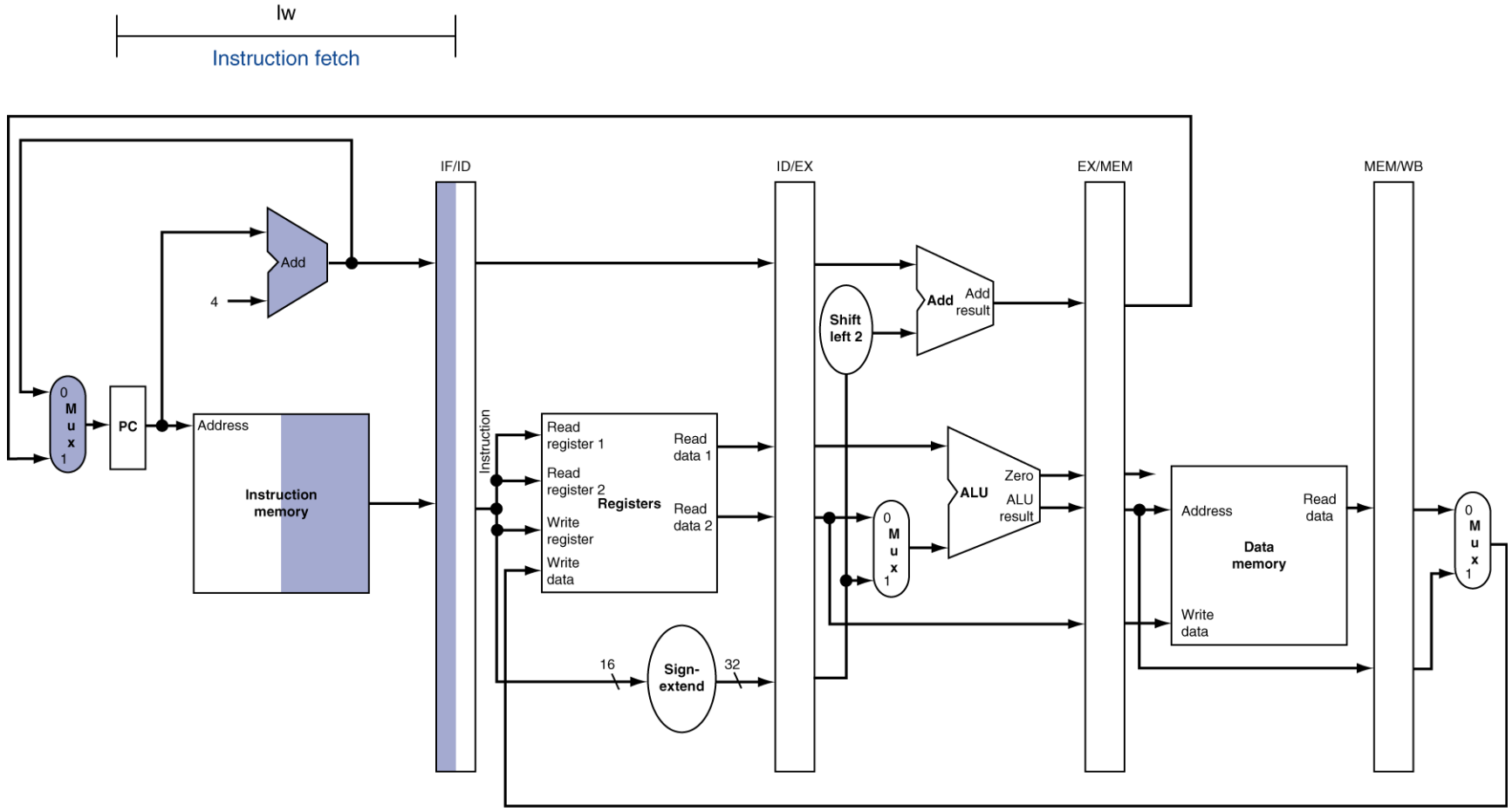




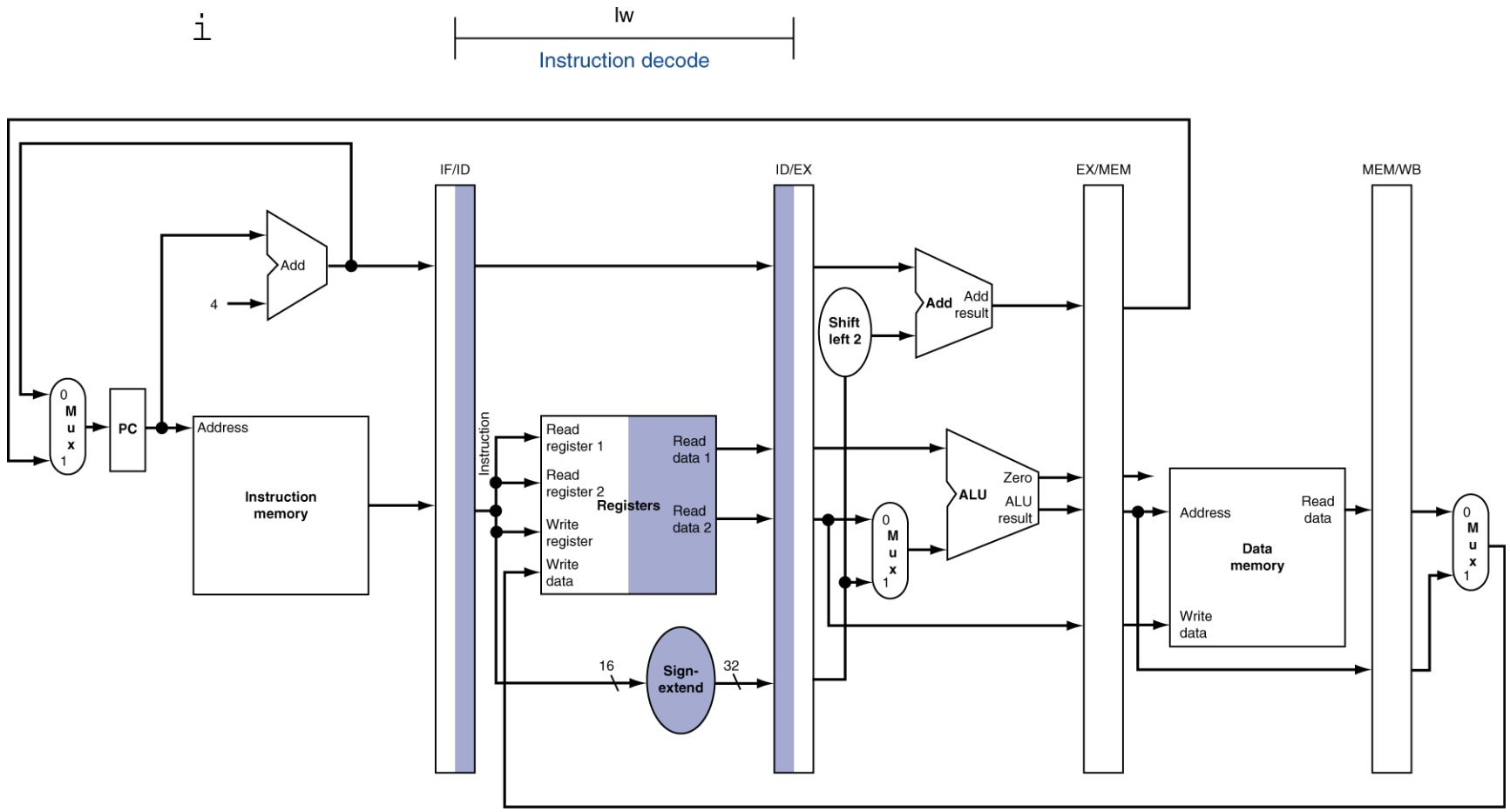




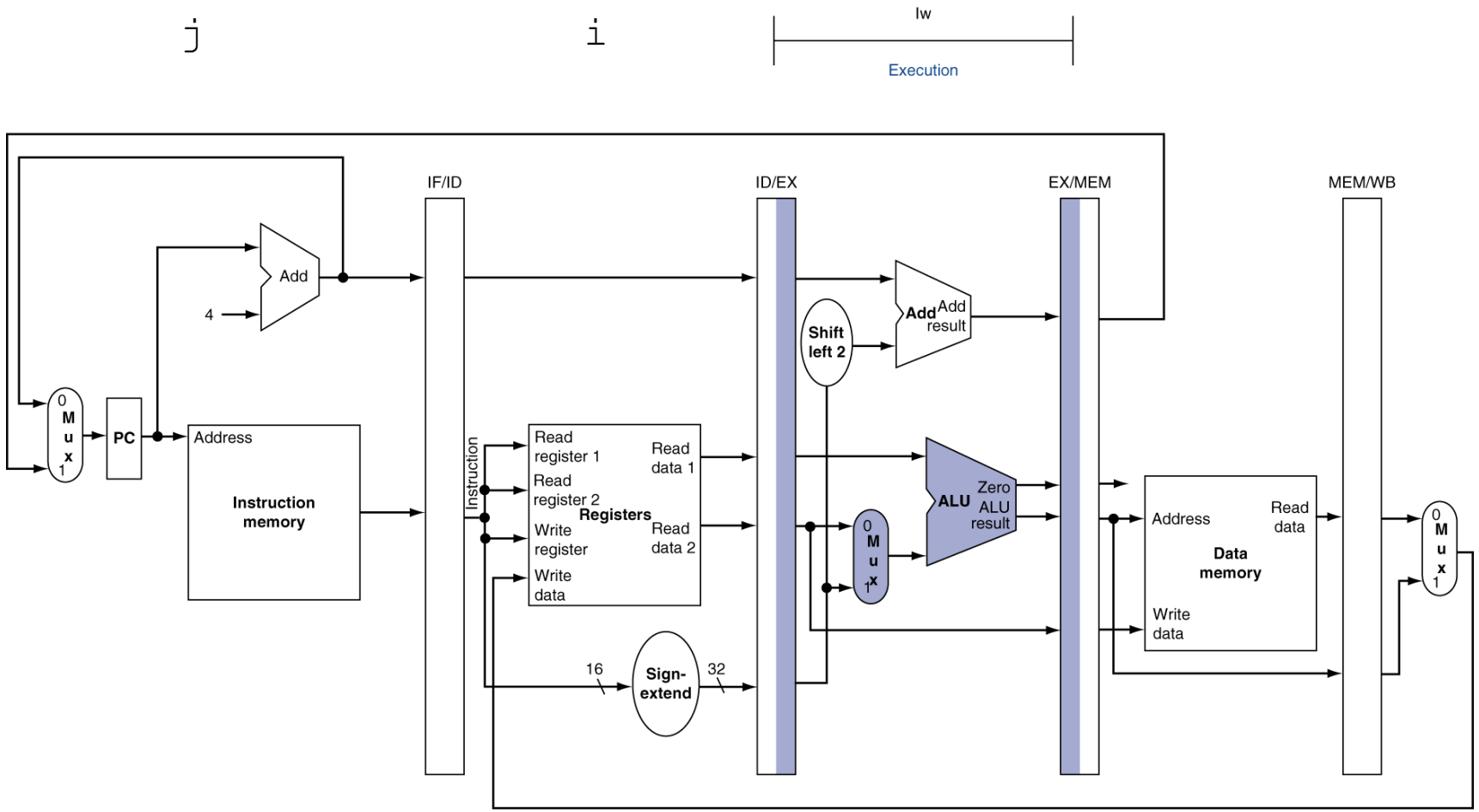
IF Stage of Load Instruction



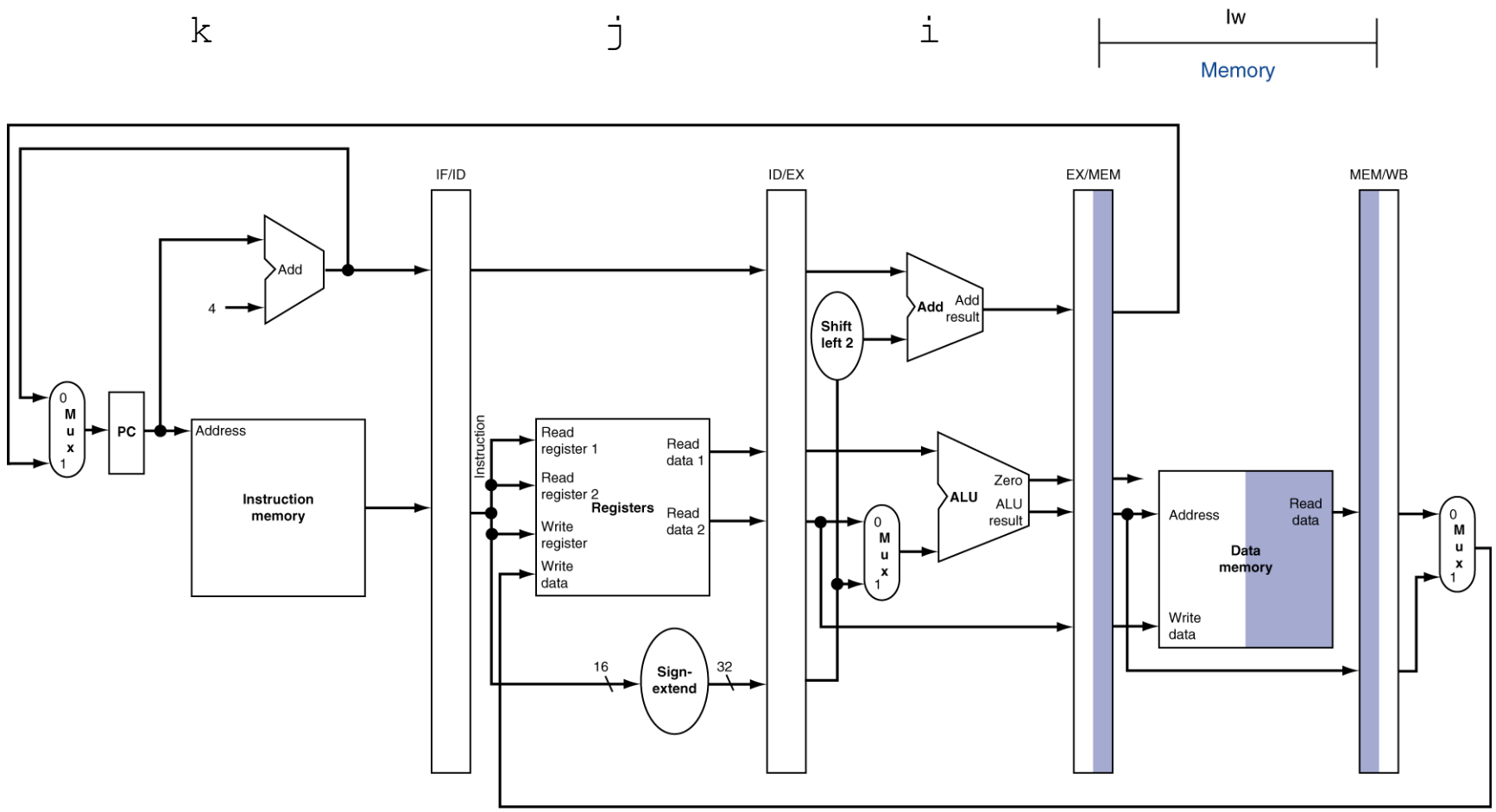
ID Stage of Load Instruction



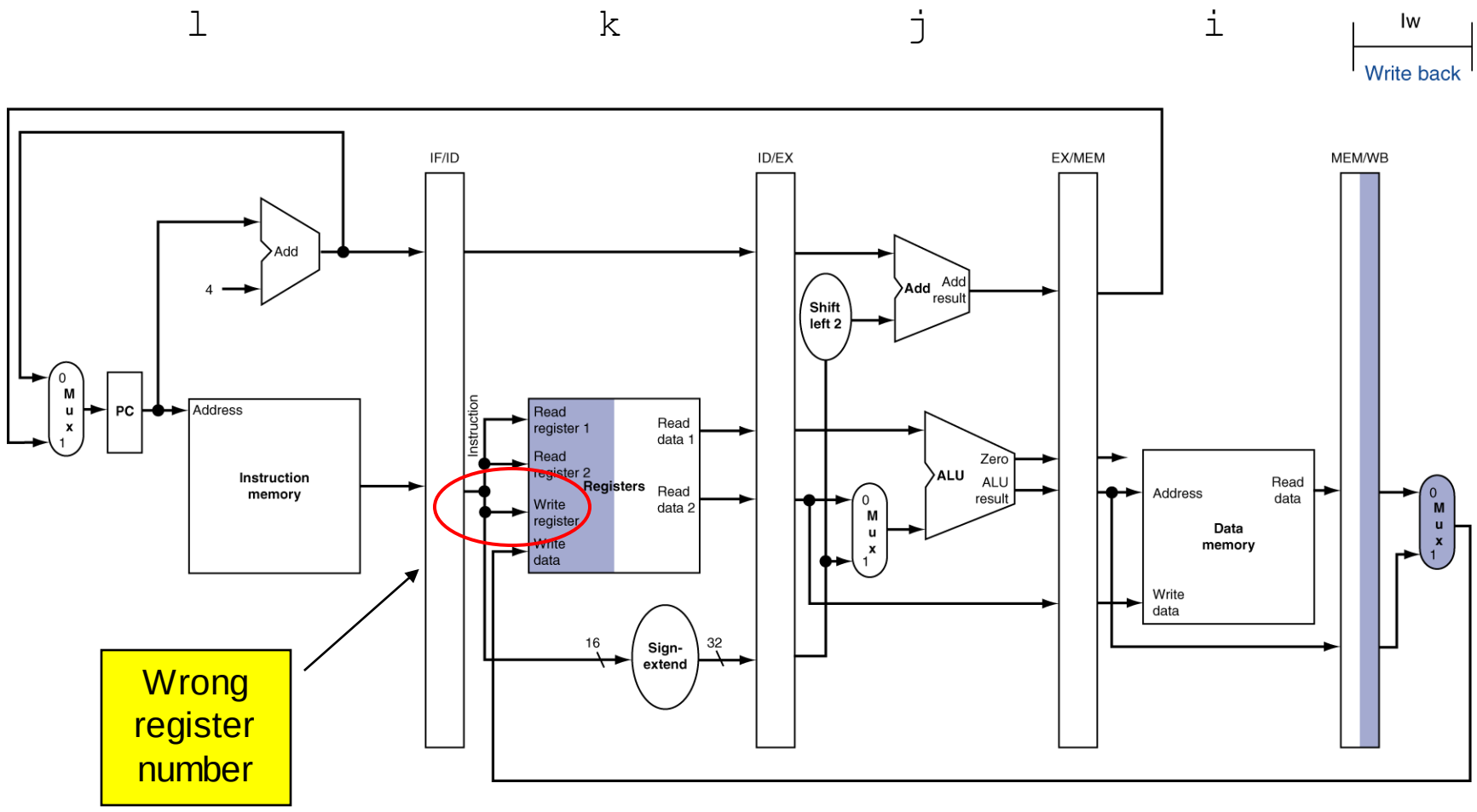
EX Stage of Load Instruction



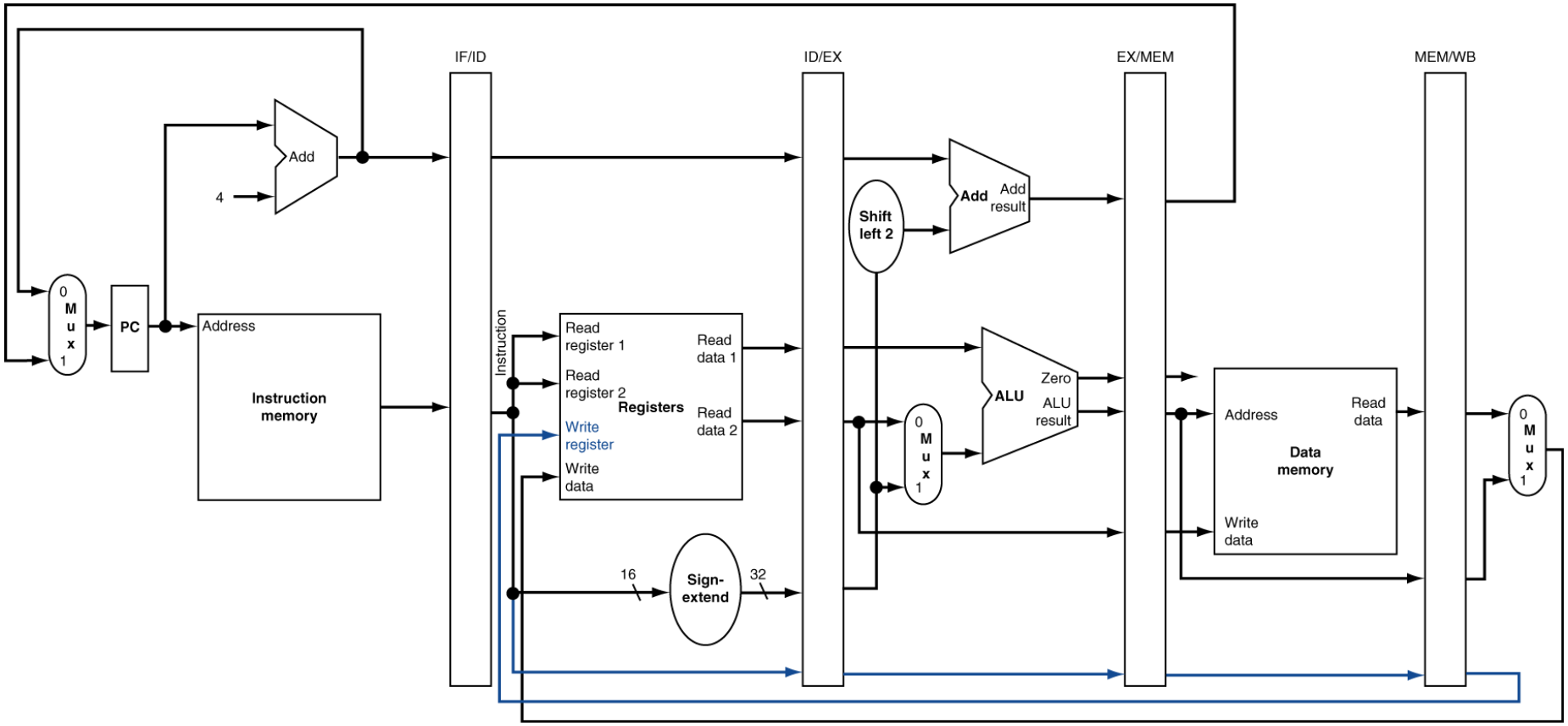
MEM Stage of Load Instruction



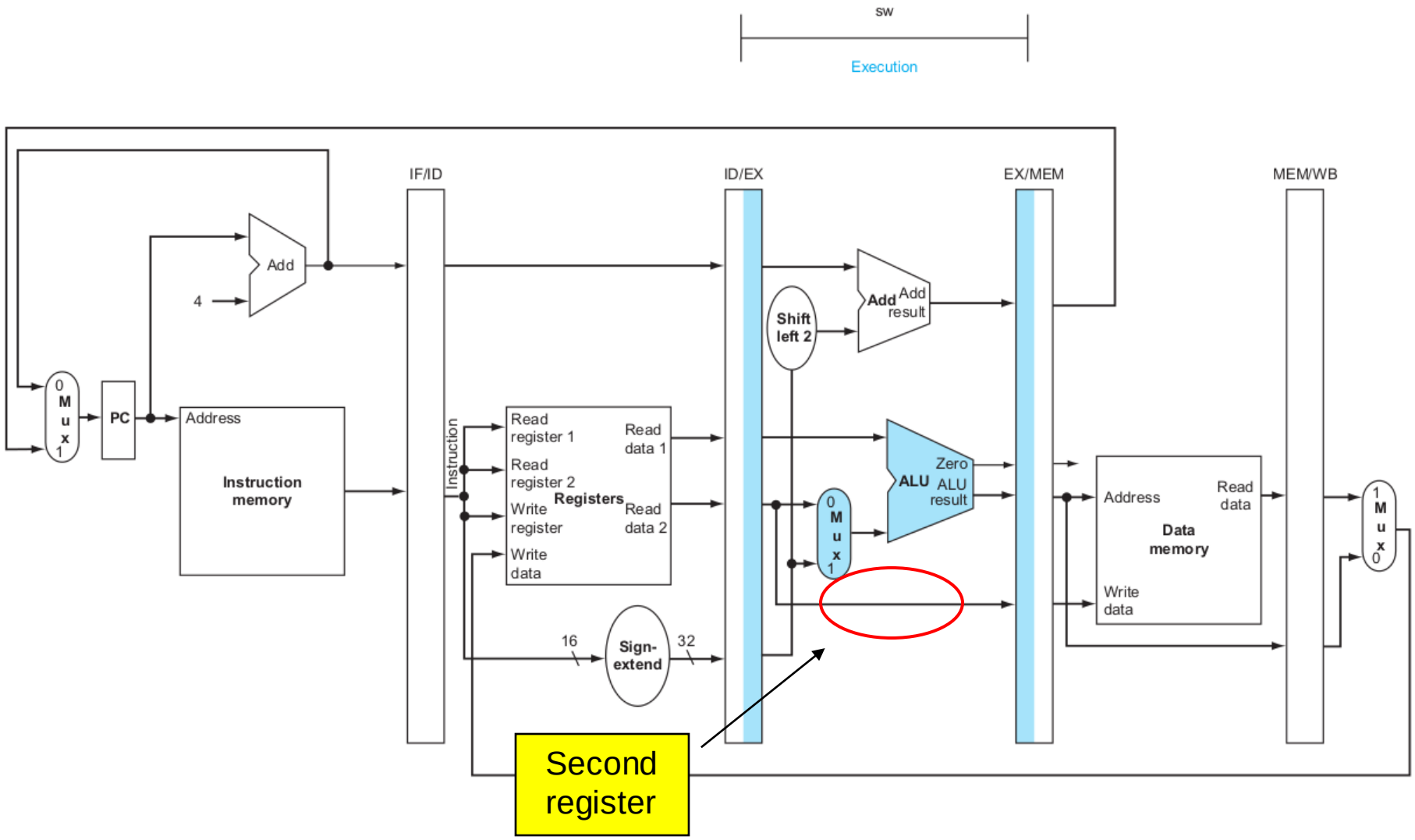
WB Stage of Load Instruction



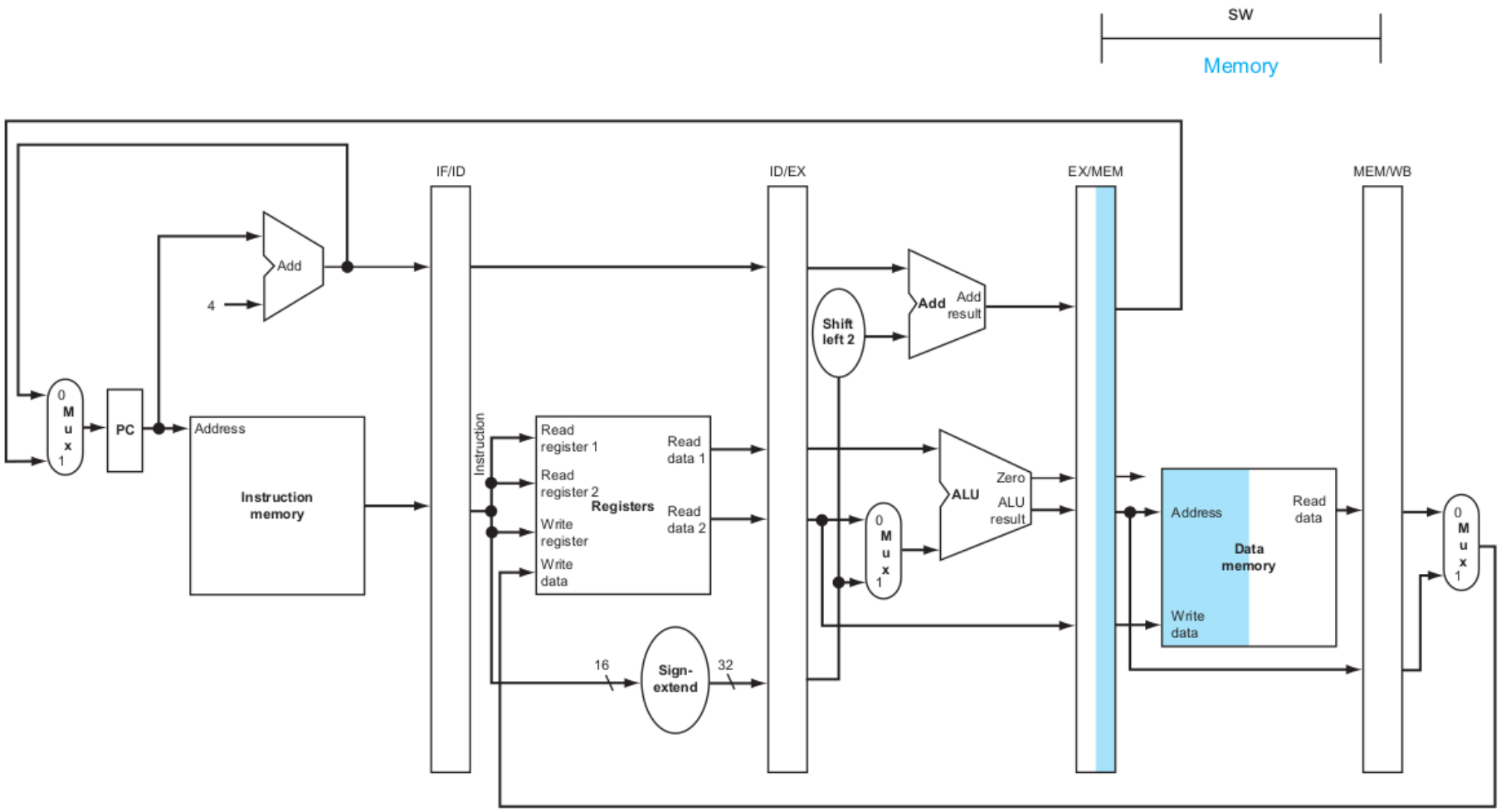
Fixed Datapath



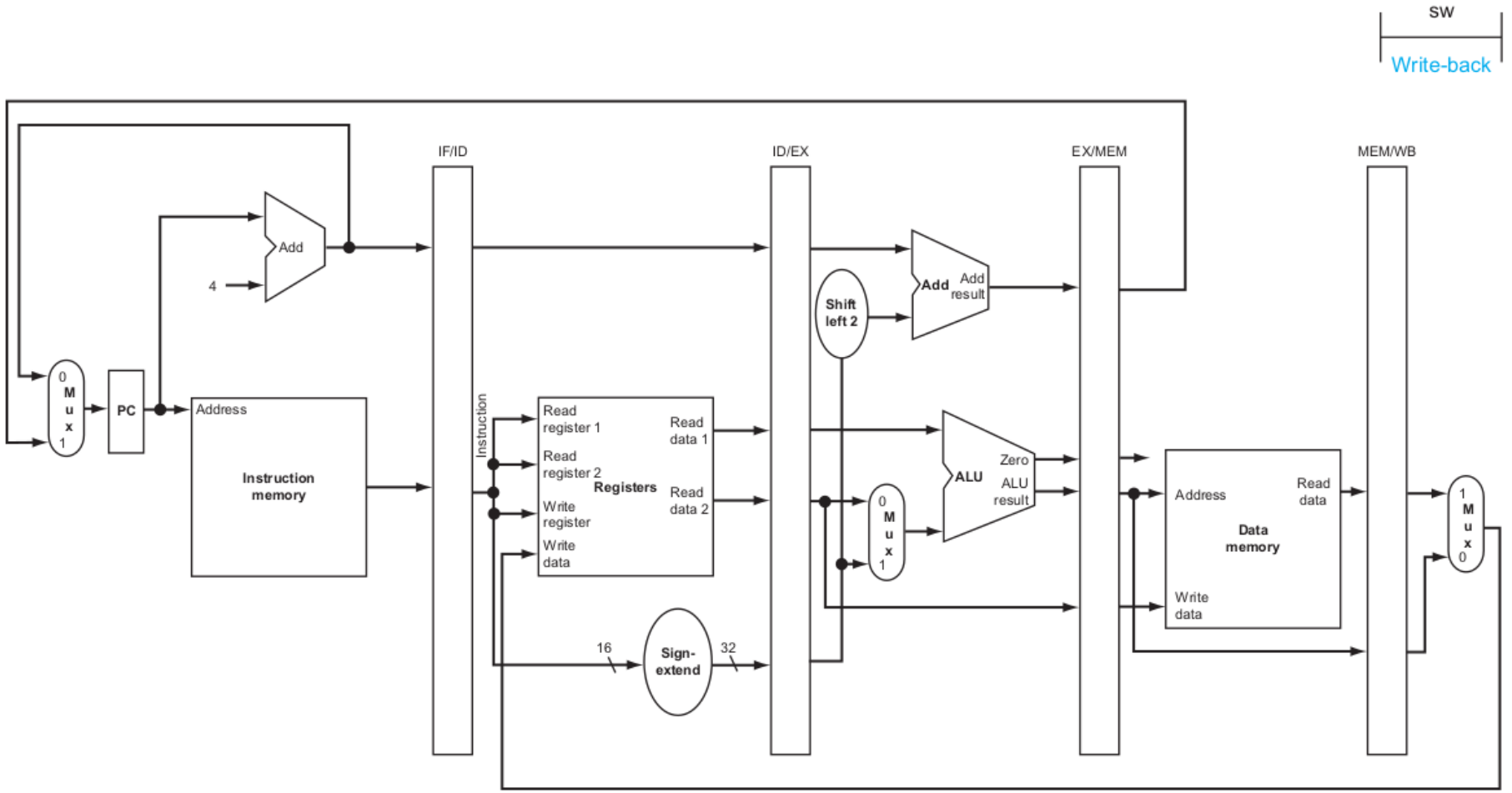
EX Stage of Store Instruction



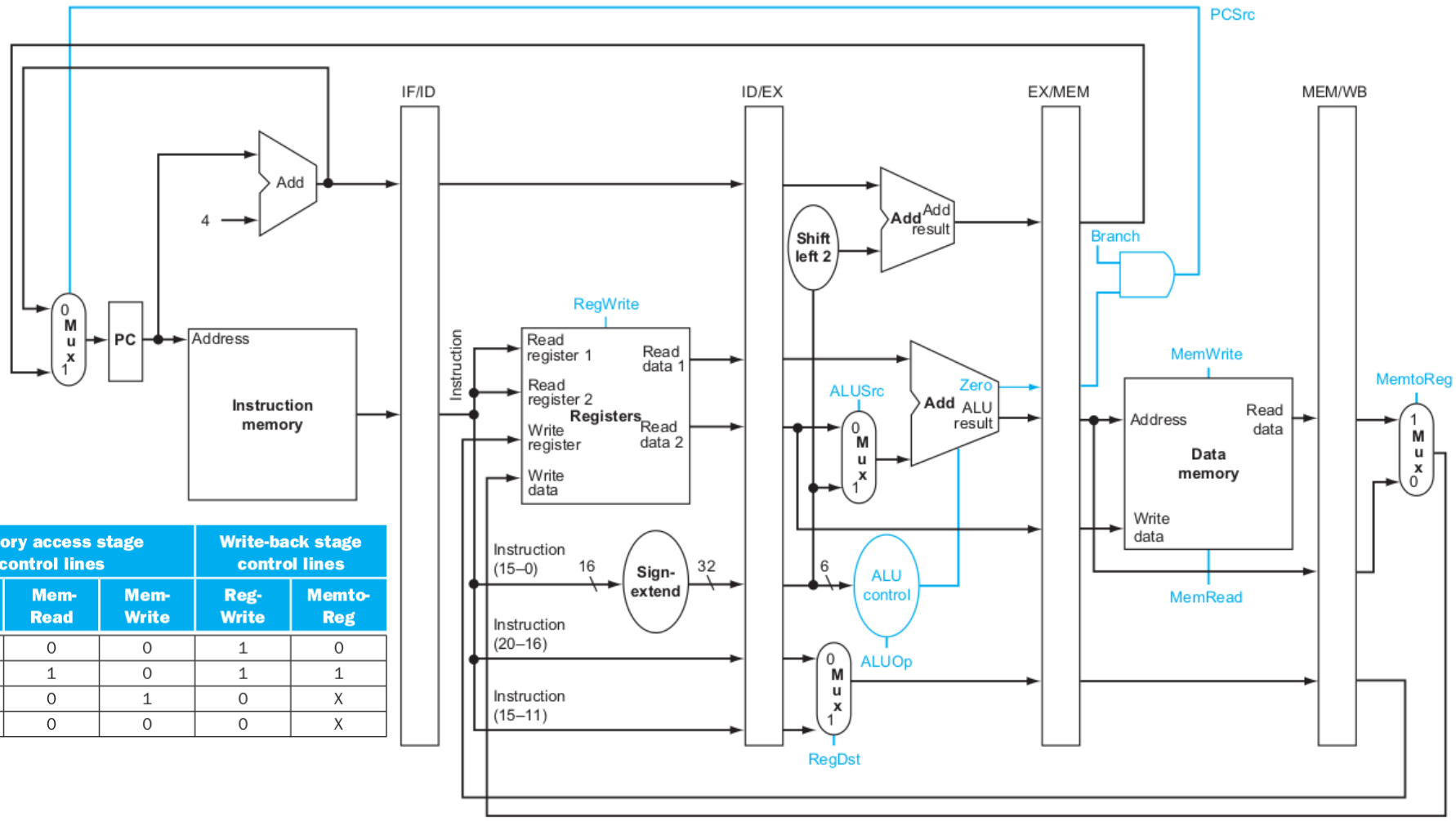
MEM Stage of Store Instruction



WB Stage of Store Instruction

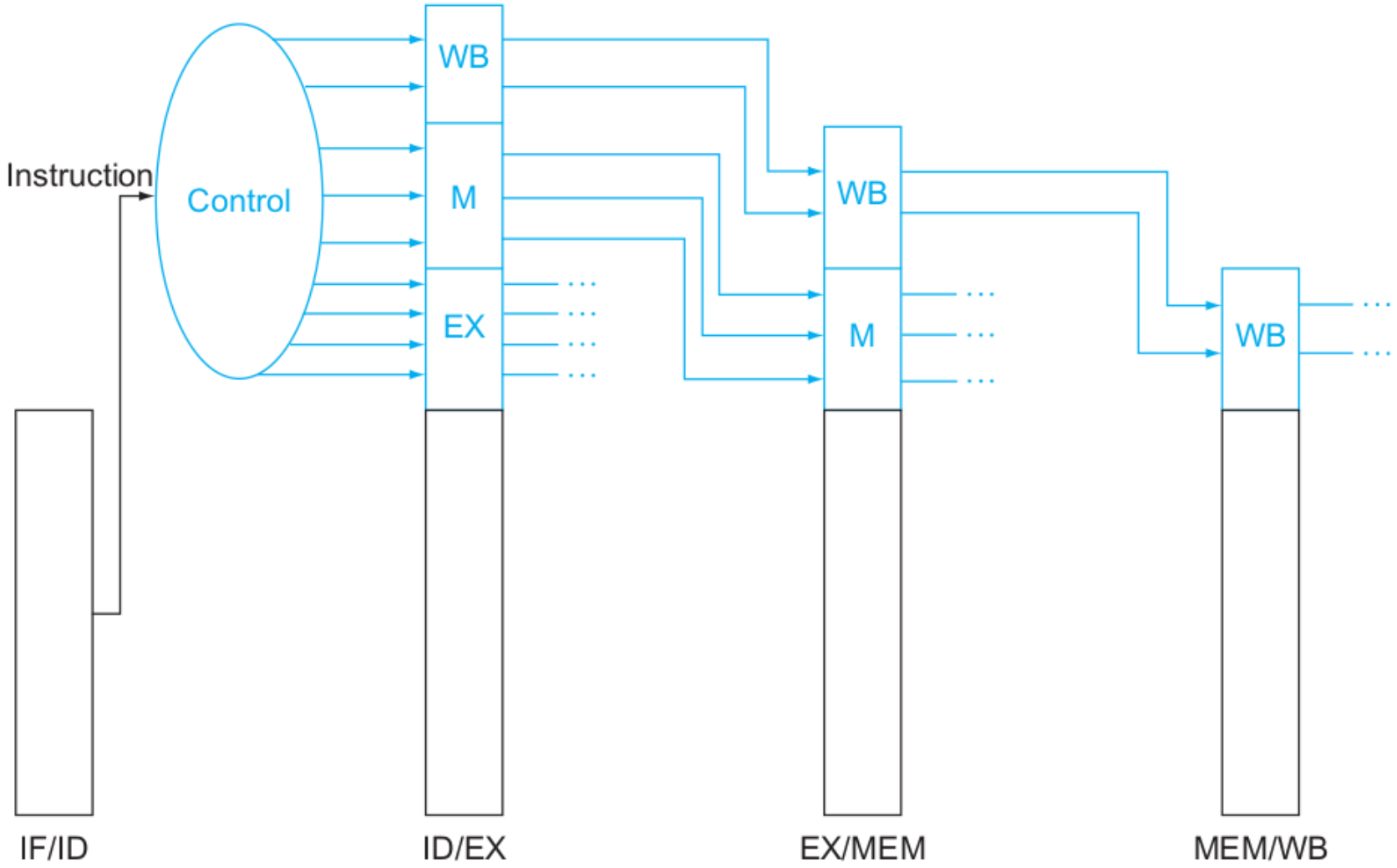


Control Lines

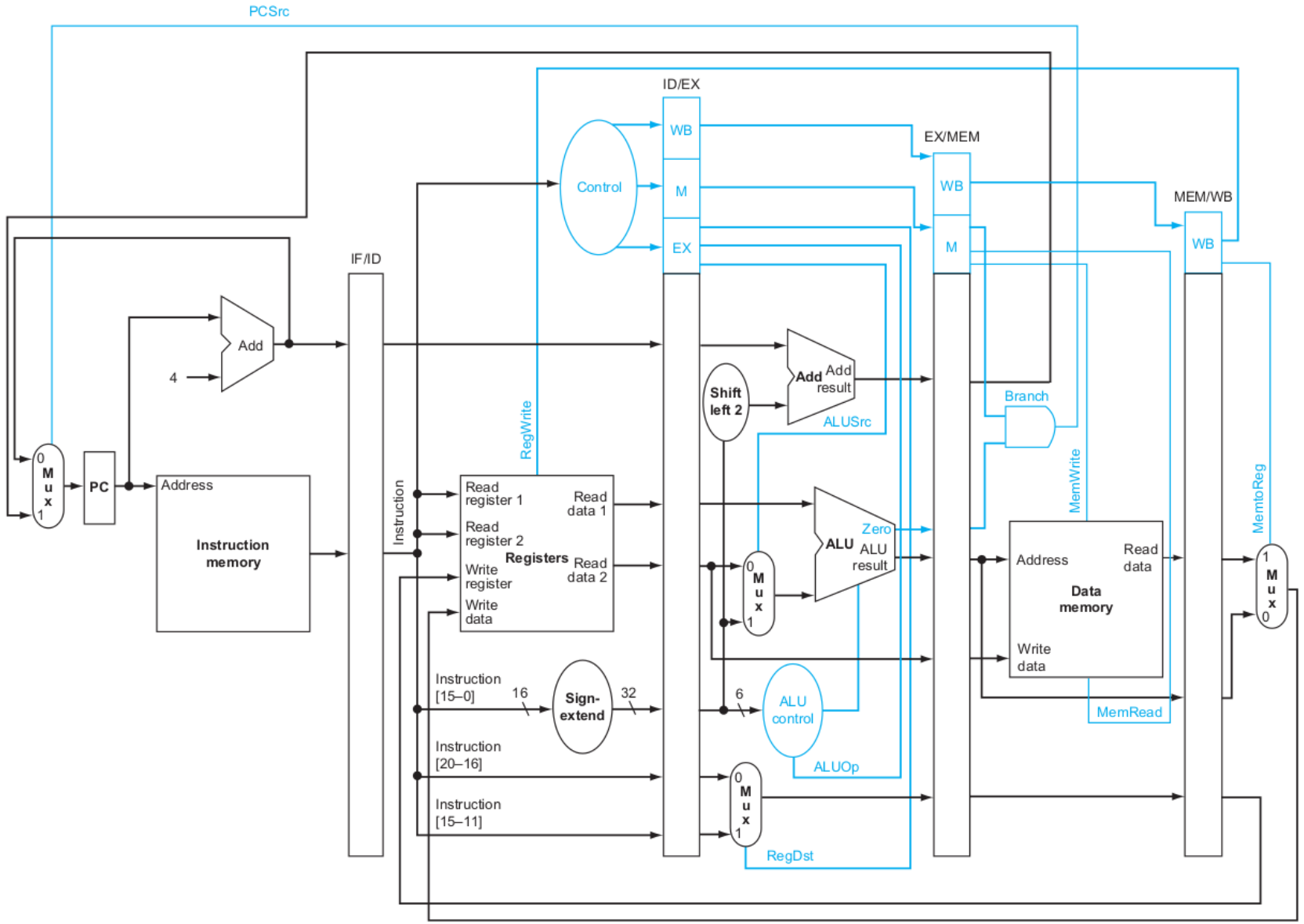


Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Pipelined Control



Pipelined Datapath & Control



Pipeline Hazards

- There are situations, called hazards, that prevent the next instruction from executing during its designated clock cycle
  - Structural hazards
    - They arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution
      - In modern processors, structural hazards occur primarily in special purpose functional units that are less frequently used (e.g., floating point divide)
  - Data hazards
    - They arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - Control hazards
    - They arise from the pipelining of branches and other instructions that change the PC

Next Lecture

- Section 4.8
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

6. Sequential Logic II

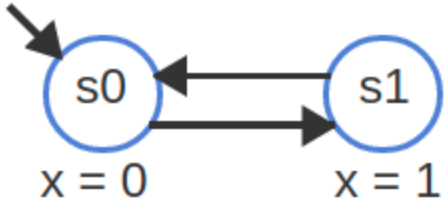
Zhenkai Zhang

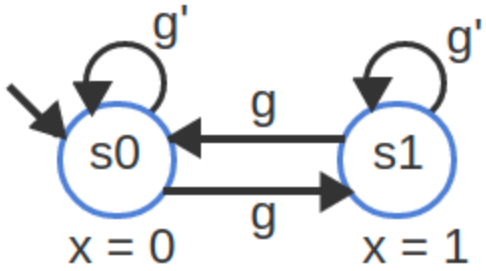
How to Describe Sequential Behavior

- To describe combinational behavior, we may use a logic function in the form of a truth table or a Boolean equation
  - But logic function lacks the capability of describing sequential behavior
- An FSM (finite-state machine) is a computation model capable of describing sequential behavior

FSM

- An FSM consists of
  - Inputs
  - Outputs
  - States
  - An initial state
  - Transitions

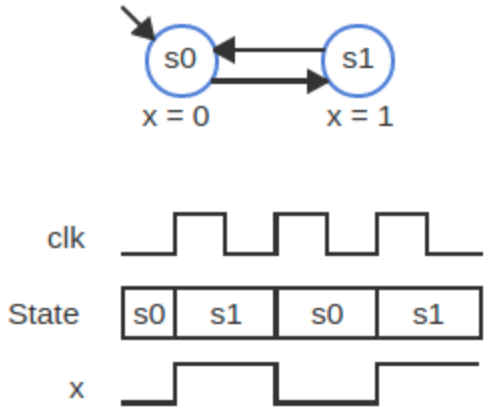


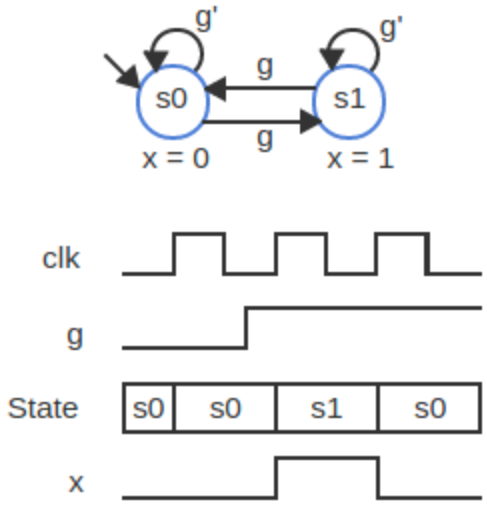


- Moore v.s. Mealy machines (they are equivalent in their capabilities)
  - The outputs only depend on the current state in a Moore FSM
    - Outputs are drawn with the states
  - The outputs depend on both the current state and the inputs in a Mealy FSM
    - Outputs are drawn on the transitions

FSM Execution

- An FSM starts in an initial state

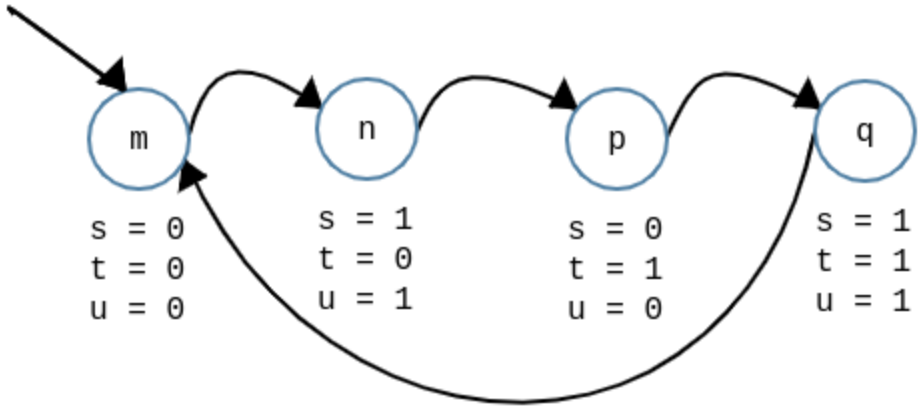




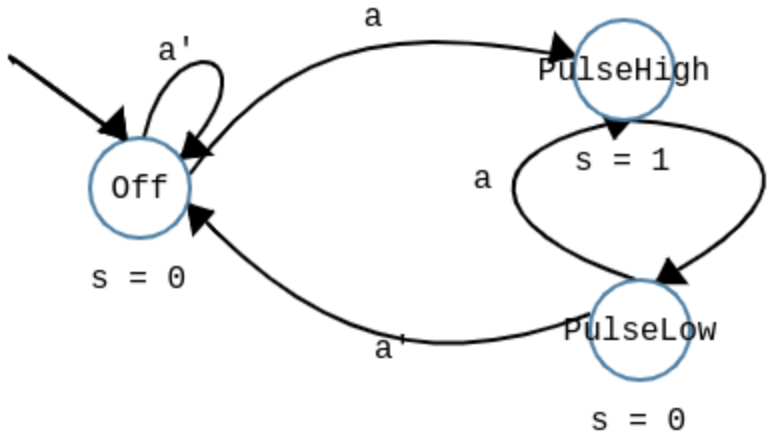
- Every time an implicit clock input rises/falls, the FSM changes to a next state pointed to by a transition whose condition evaluates to true
  - A transition without a listed condition has an implicit condition of true

Capturing Behavior with FSMs

- FSMs are commonly used to capture sequential behavior
  - To capture behavior means for a designer to describe desired behavior in some form



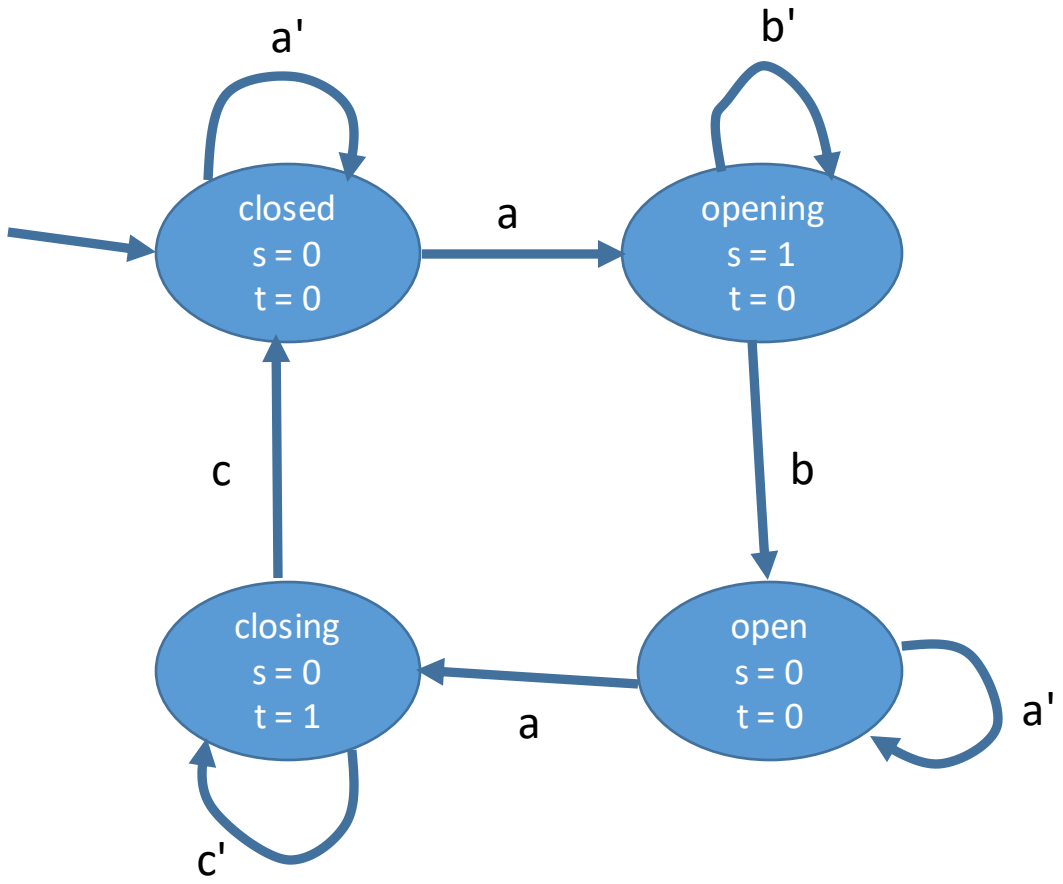
Generating a sequence of patterns on several outputs



Generating a pulsing output when an input is 1

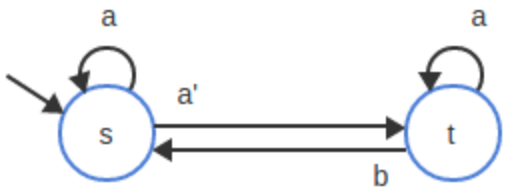
FSM Example -- Garage Door Opener

- A garage door opener has a single button causing a toggle
  - Inputs: a, b, c
    - a = 1 indicates the button is being pressed
    - b = 1 indicates fully-open
    - c = 1 indicates fully-closed
  - Outputs: s, t
    - s = 1 raises the door
    - t = 1 lowers the door



FSM Issues

- For a given state, exactly one outgoing transition should have a true condition at any given time
  - No more, no less!
  - A transition without a listed condition has an implicit condition of true

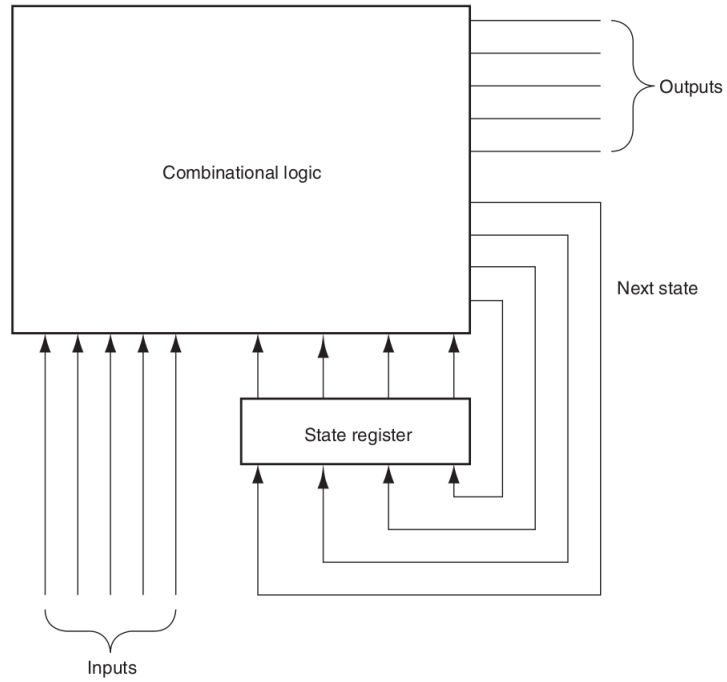


Is this FSM good?

- A common shorthand – if an output is not explicitly assigned in a state, the output is implicitly assigned with 0
  - Sometimes a designer still explicitly assigns an output with 0 for clarity

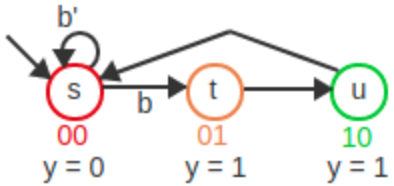
FSMs to Circuits

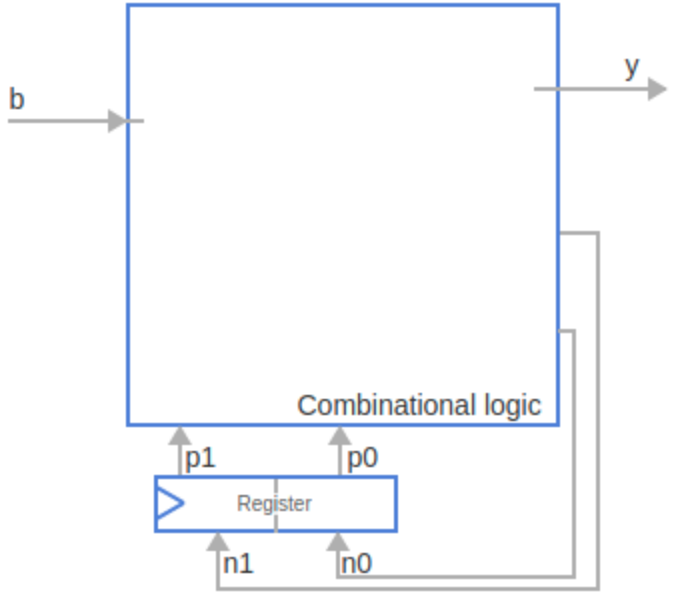
- An FSM can be converted to a sequential logic circuit (known as a controller) consisting of a state register and combinational logic
  - The state register holds an FSM's present state
  - The combinational logic computes
    - The FSM's outputs, based on the present state
      - Output function
      - Moore FSM
    - The next state, based on the present state and the FSM's inputs
      - Next-state function



State Register

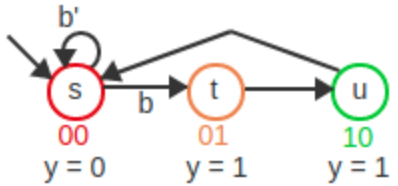
- Each state requires a unique bit encoding, which is then stored in the state register
  - Three or four states require 2 bits (00, 01, 10, 11)
  - Five, six, seven, or eight states require 3 bits (000, 001, ..., 111)
  - N states require how many bits?
    - $\lceil \log_2 N \rceil$





Output and Next-State Functions

- A truth table can be used for deriving the combinational logic
  - Output function
  - Next-state function

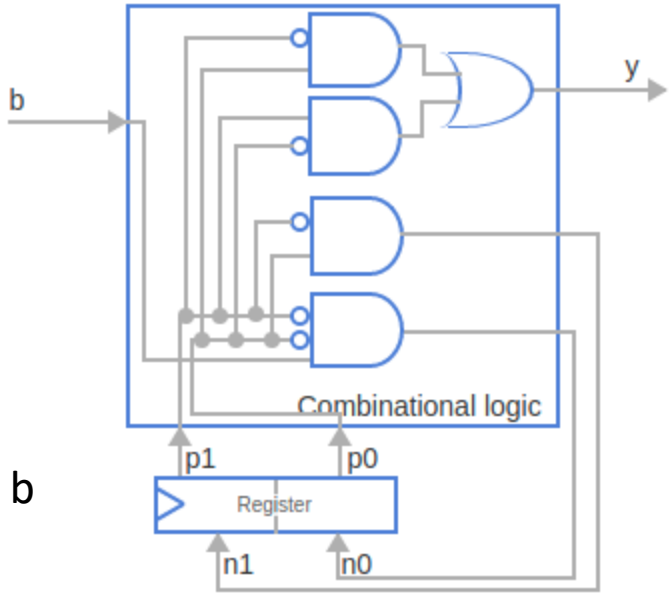


	current state		input	next state		output
	p1	p0	b	n1	n0	y
s	0	0	0	0	0	0
	0	0	1	0	1	0
t	0	1	0	1	0	1
	0	1	1	1	0	1
u	1	0	0	0	0	1
	1	0	1	0	0	1
Unused	1	1	0	0	0	0
	1	1	1	0	0	0

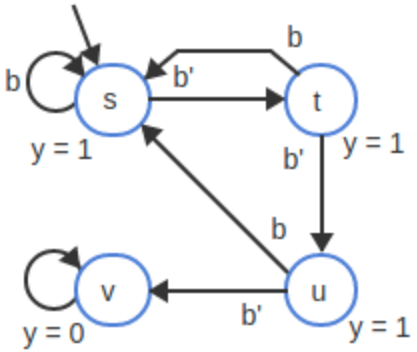
n1 = p1’ p0 b’ + p1’ p0 b = p1’ p0

n0 = p1’ p0’ b

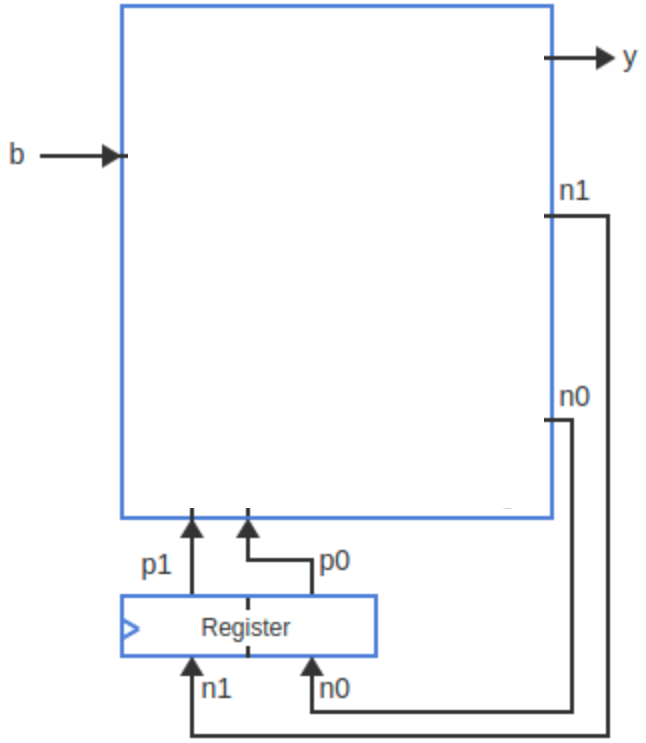
y = p1’ p0 b’ + p1’ p0 b + p1 p0’ b’ + p1 p0’ b
  = p1’ p0 + p1 p0’



FSM to Circuit Example

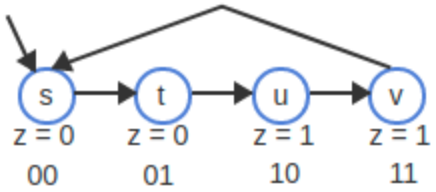


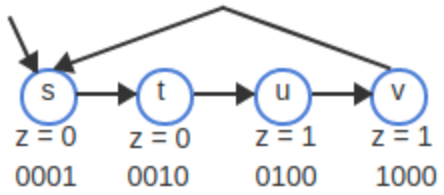
	p1	p0	b	n1	n0	y
s	0	0	0	0	1	1
	0	0	1	0	0	1
t	0	1	0	1	0	1
	0	1	1	0	0	1
u	1	0	0	1	1	1
	1	0	1	0	0	1
v	1	1	0	1	1	0
	1	1	1	1	1	0



State Encodings

- Binary state encodings
  - It uses the fewest bits possible for a given number of states
- One-hot state encodings
  - It uses N bits for N states, with each encoding having exactly one bit set to 1
  - Next-state function is simple to implement (using less logic gates)

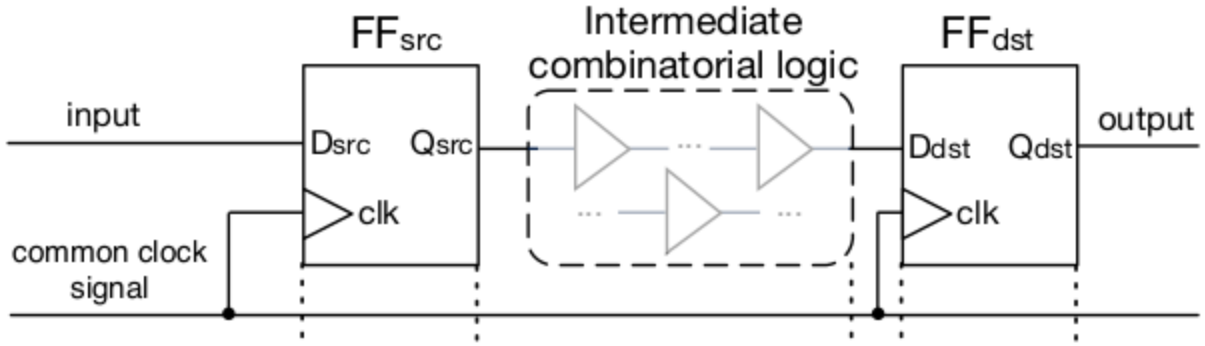




- FSM startup
  - Assuming an initial state's encoding is all 0's, then the circuit merely needs to start with 0's in the state register
    - Registers typically have a clear state

Synchronous Digital Circuit

- A synchronous system is one in which edge-triggered memory elements are controlled in synchrony with a global clock signal

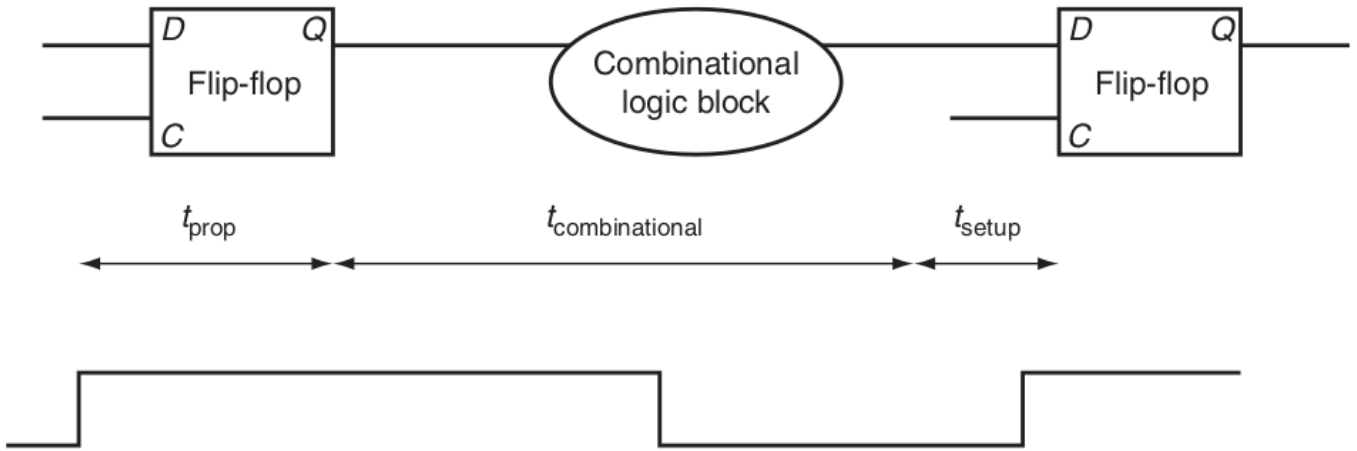


Can the global clock be arbitrarily fast?

Clock Period Constraints

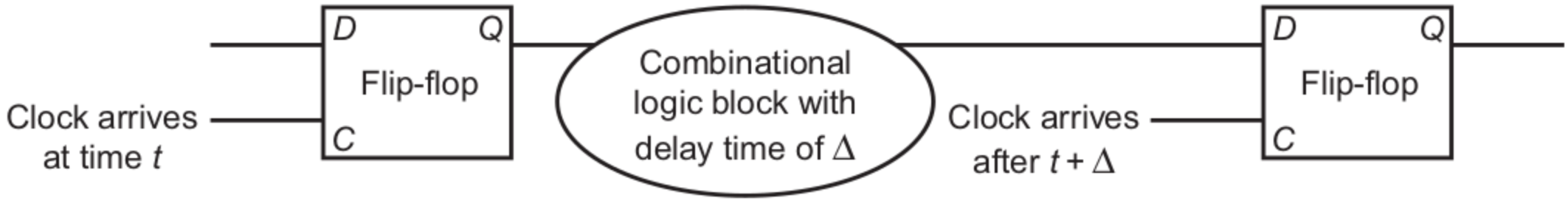
- We have mentioned t_{setup} and t_{hold}
- $t_{combinational}$ is the longest delay of combinational logic surrounded by two flip-flops (determined by the the critical path)
- t_{prop} is the time for a signal to propagate through a flip-flop

$$t \geq t_{prop} + t_{combinational} + t_{setup}$$



Clock Skew

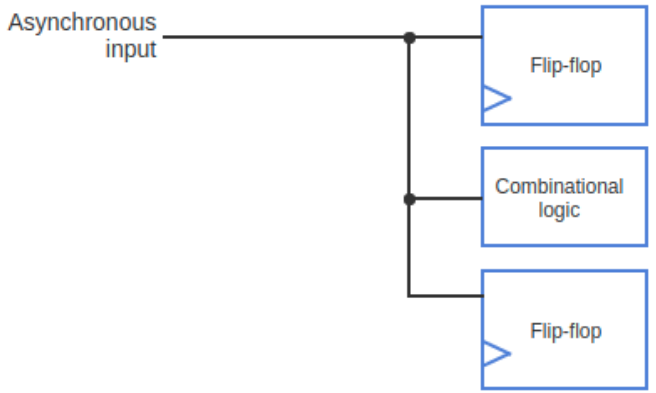
- Clock distribution is not perfect, and there are differences in absolute time between the times when state elements see a clock edge
  - Clock skew Δ



$$t \geq t_{prop} + t_{combinational} + t_{setup} + t_{skew}$$

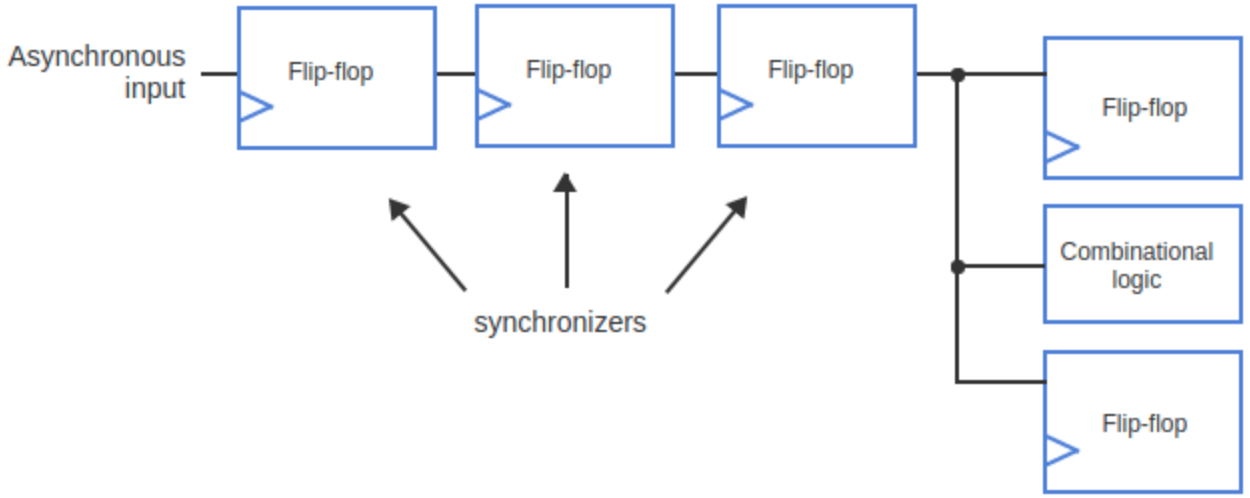
Asynchronous Inputs

- Recall what may happen if the setup time and/or hold time of a flip-flop are not met
- An asynchronous input is an input that is not synchronized with a circuit's clock
  - A button input can be pressed at any time by a user, which is not synchronized with the clock
  - An asynchronous input can change at any time, including during the setup time and hold time, which can lead to metastable states



Synchronizers

- A common way to synchronize an asynchronous input is to connect the asynchronous input into a synchronizer flip-flop
  - A synchronizer flip-flop is typically an extremely fast flip-flop with very small setup and hold times



Summary

- Combinational logic
  - Logic gates, truth table, Boolean equations
  - Sum of products, sum of minterms (truth table to sum of minterms)
  - Commonly used combinational logic circuits
  - ALU (adder)
- Sequential logic
  - Memory elements (latches and flip-flops)
  - Clock frequency constraints
  - FSM

Next Lecture

- Section 4.1 – 4.4
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

8. Datapath & Control

Zhenkai Zhang

Chapters 2 & 3

- CPSC 3300 does not cover chapters 2 and 3
  - Your prerequisite CPSC 2310 should have covered ISA and arithmetic
    - May not be MIPS
  - If you have forgotten most of it, please go to read Chapters 2 & 3
- We will review some necessary things along the way

MIPS Instruction Formats

- R-type (R for register)
  - Arithmetic instructions (add, sub)
  - Logic instructions (and, or)
  - Unconditional jump instructions (jr, jalr)
    - Addresses are in registers (jump register)

Type	31 26	25 21	20 16	15 11	10 06	05 00
R-Type	opcode	$rs	$rt	$rd	shamt	funct
I-Type	opcode	$rs	$rt	imm		
J-Type	opcode	address				

- I-type (I for immediate)
  - Arithmetic & logic instructions with constants (addi, andi)
  - Data transfer instructions (lw, sw)
  - Conditional branch instructions (beq, bne)
- J-type (J for jump)
  - Unconditional jump instructions (j, jal)
    - Addresses are directly specified

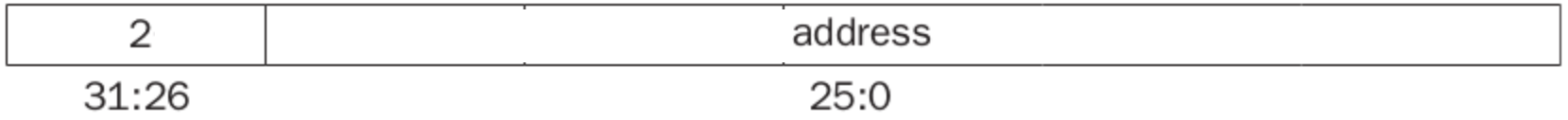
A Basic MIPS Implementation

- An implementation includes a subset of the core MIPS instruction set
  - Arithmetic & logical instructions – add, sub, AND, OR, and slt (R-type)

  - Data transfer instructions – lw and sw (I-type)

  - Conditional branch instruction – beq (I-type)

  - Unconditional branch instruction – j (J-type)

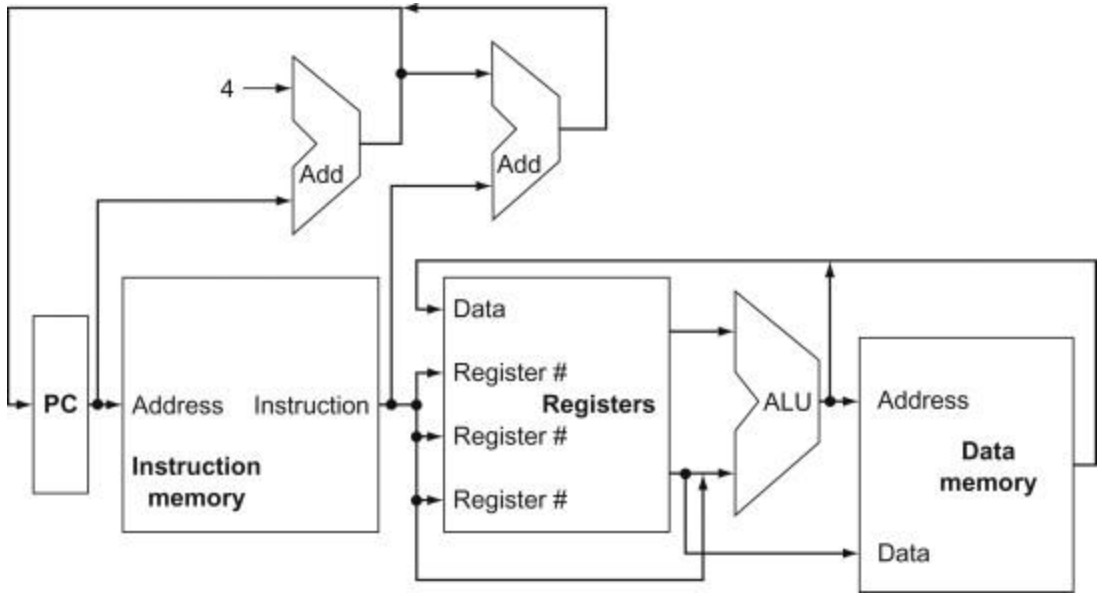


MIPS Instruction Execution

- PC → memory; fetch instruction
- Register numbers → register file; read registers
  - For some instructions, only 1 register is read, while most require reading 2
- Depending on the instruction class
  - Use ALU to calculate
    - Arithmetic/logic results
    - Memory addresses for load/store
    - Comparison for checking branch conditions
  - Access memory to read data for load or write data for store
  - Write data from the ALU (arithmetic/logical) or memory (load) into a register
  - PC ← branch target address or PC + 4

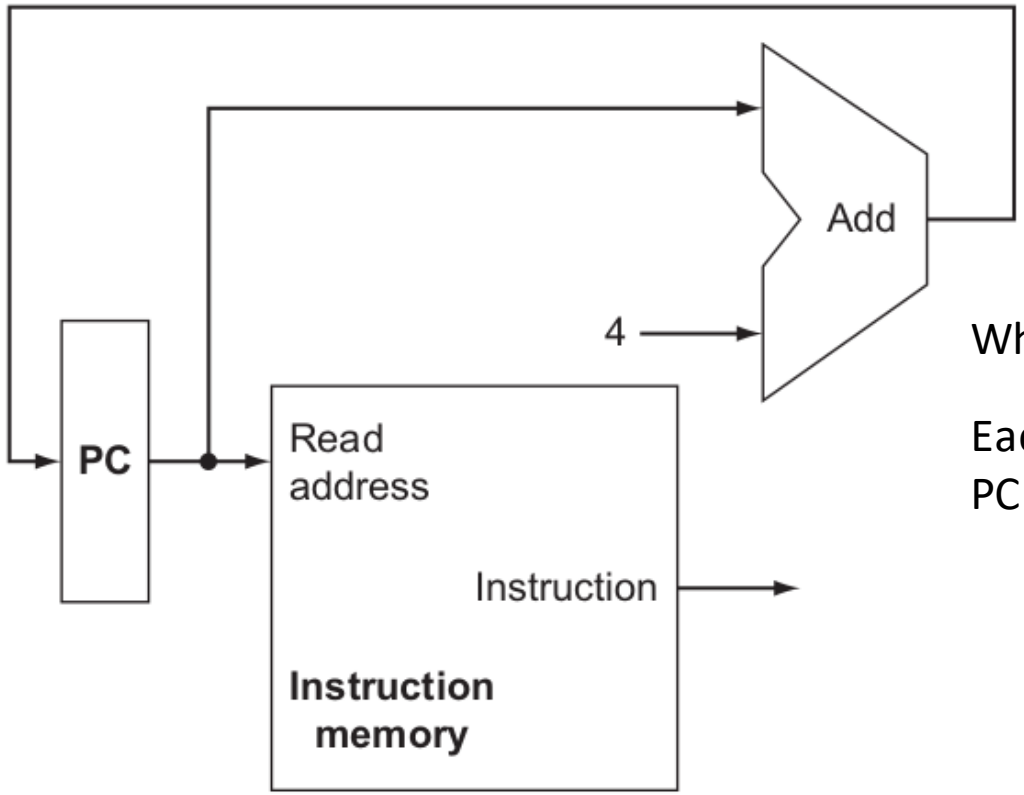
Building a Simple Datapath

- Datapath elements operate on or hold data within a processor
  - Registers, ALUs, instruction and data memories (SRAM), adders…
  - We focus on single-cycle design
    - Each datapath element can only do one function at a time
    - Hence, we need separate instruction and data memories
- We will build a MIPS datapath incrementally
  - Refining the overview design



Instruction Fetch

- The program counter (PC) is a register that contains the address of the next instruction in the program to be executed

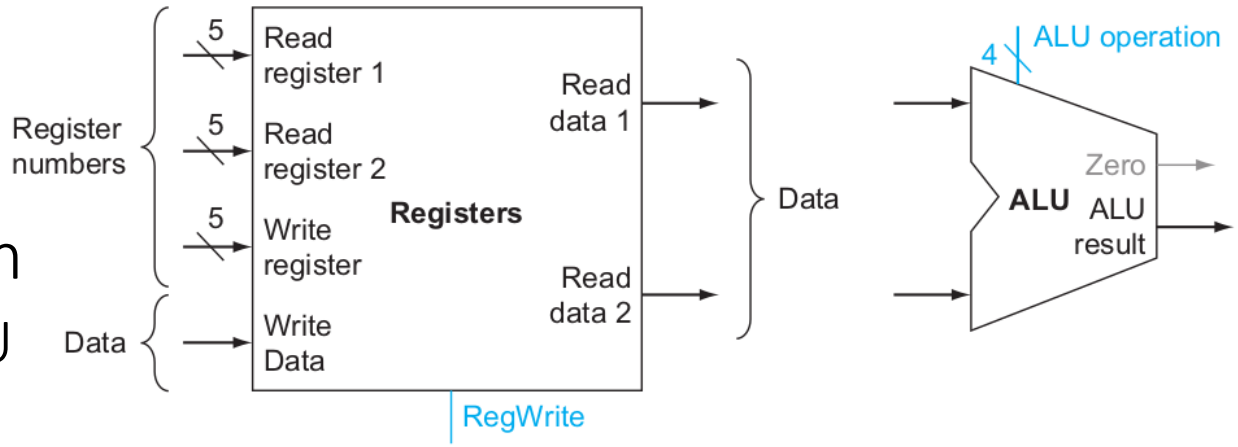


Why PC + 4?

Each MIPS instruction has 4 bytes
PC is always a multiple of 4

R-Type Arithmetic/Logic Instructions

- Read two register operands
  - The first register source operand is *rs*
  - The second operand is *rt*
- Perform arithmetic/logical operation
  - The *funct* field selects the specific ALU operation
- Write the result to a register
  - The register destination operand is *rd*

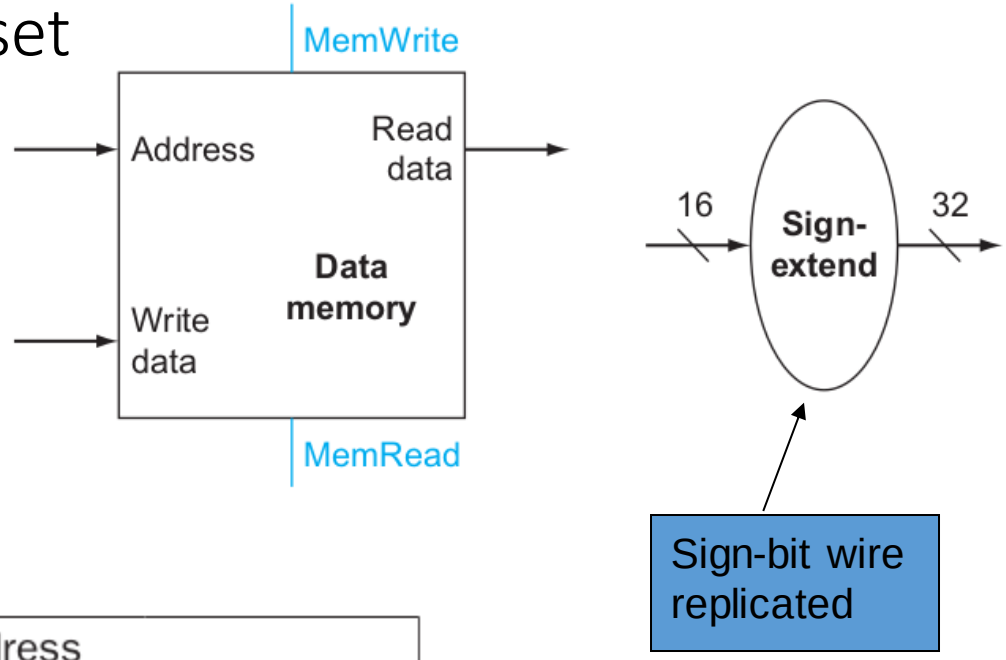


0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

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

I-Type Load/Store Instructions

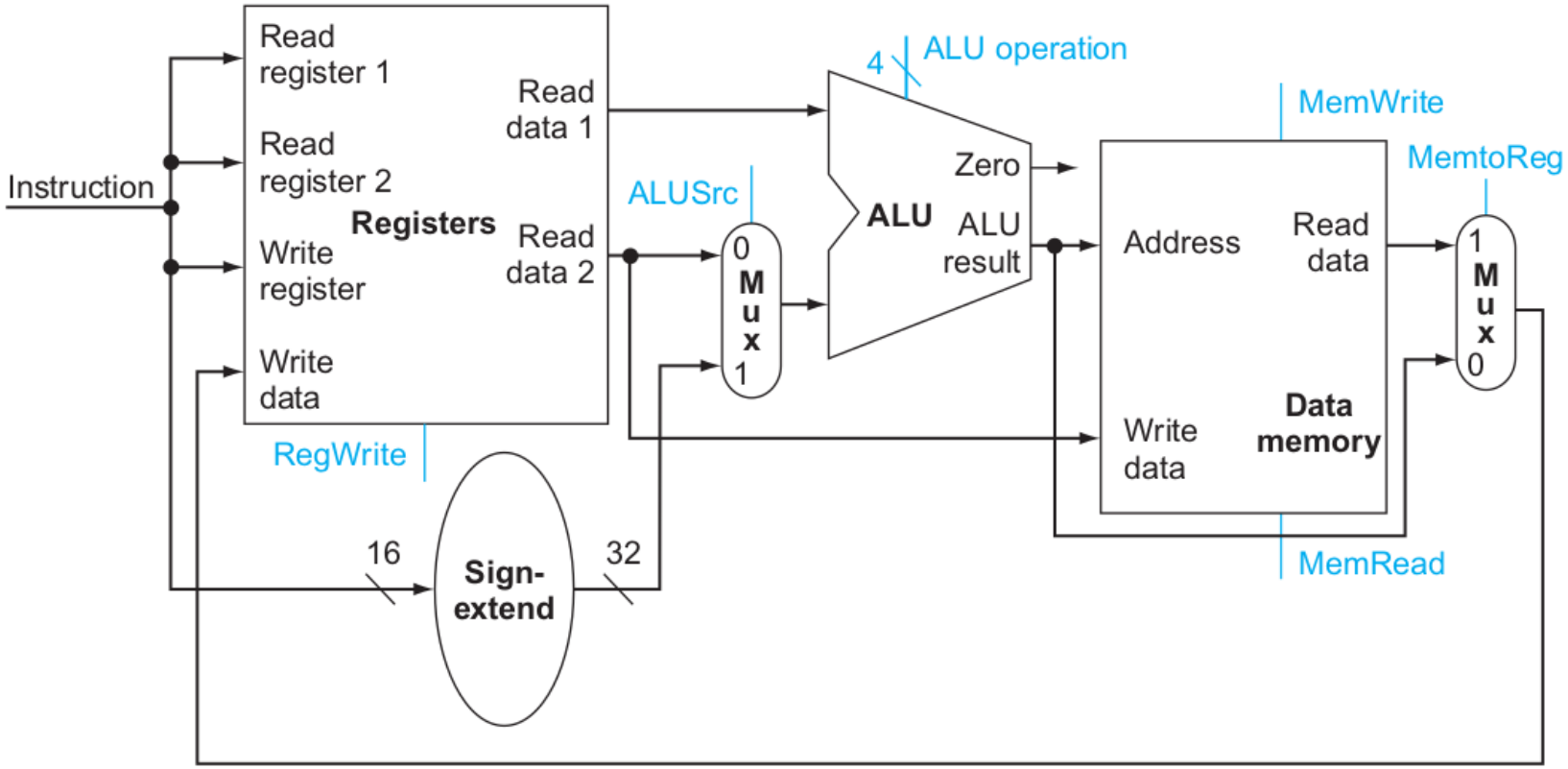
- Read base register operand
  - The base address is in register *rs*
- Calculate memory address using 16-bit offset
  - Use ALU, but sign-extended offset
- Update register file or memory
  - Load: Read memory and update register *rt*
  - Store: Write register *rt* to memory



35 or 43	rs	rt	address
31:26	25:21	20:16	15:0

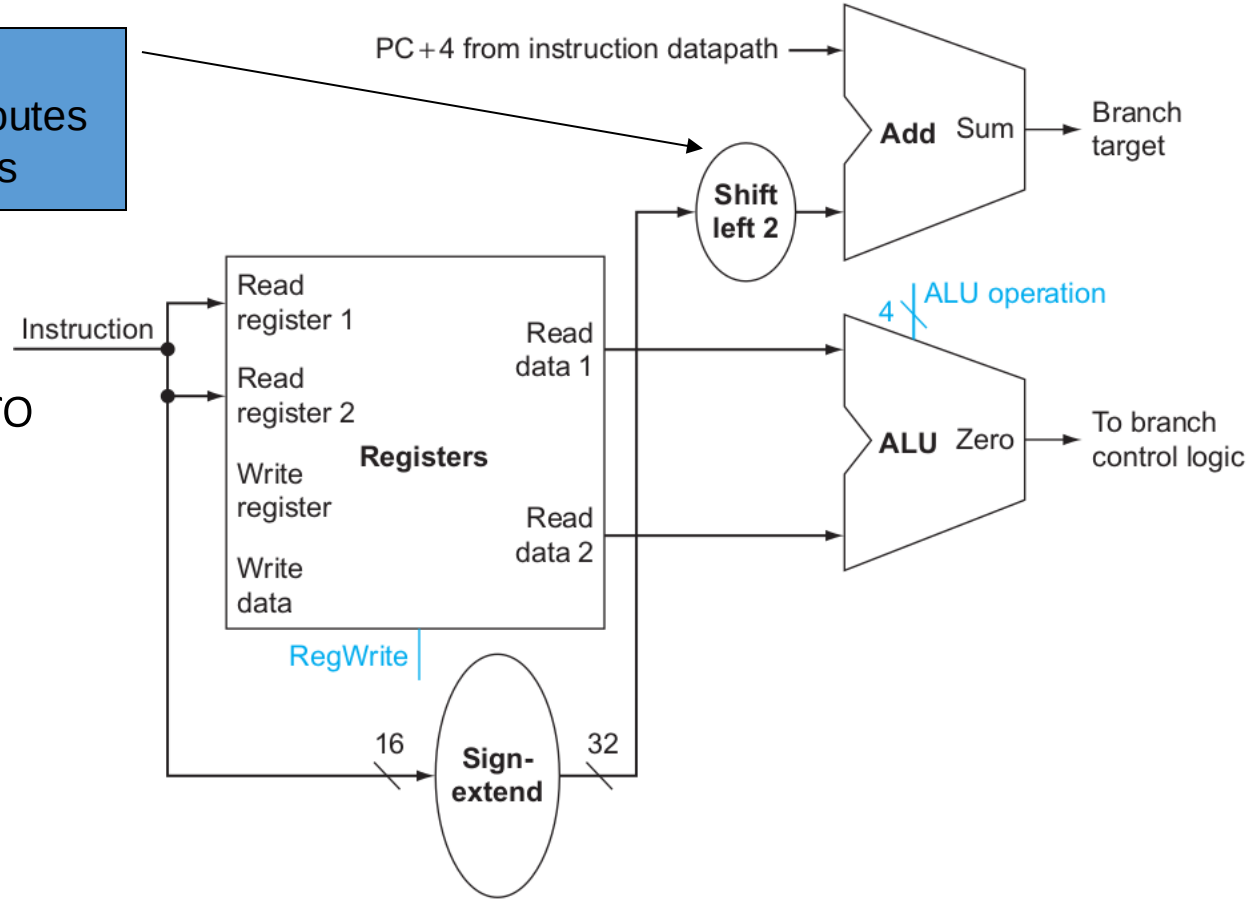
Combining Elements (R-Type & Load/Store)

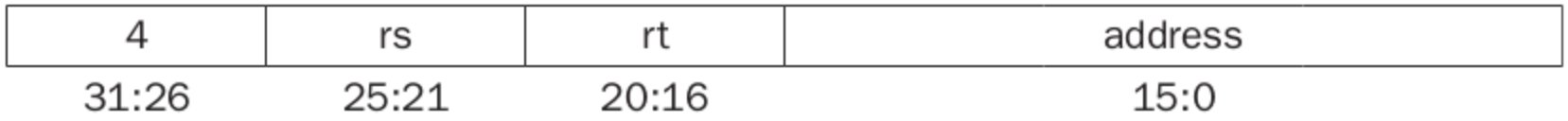
- Use multiplexers where alternate data sources are used for different instructions



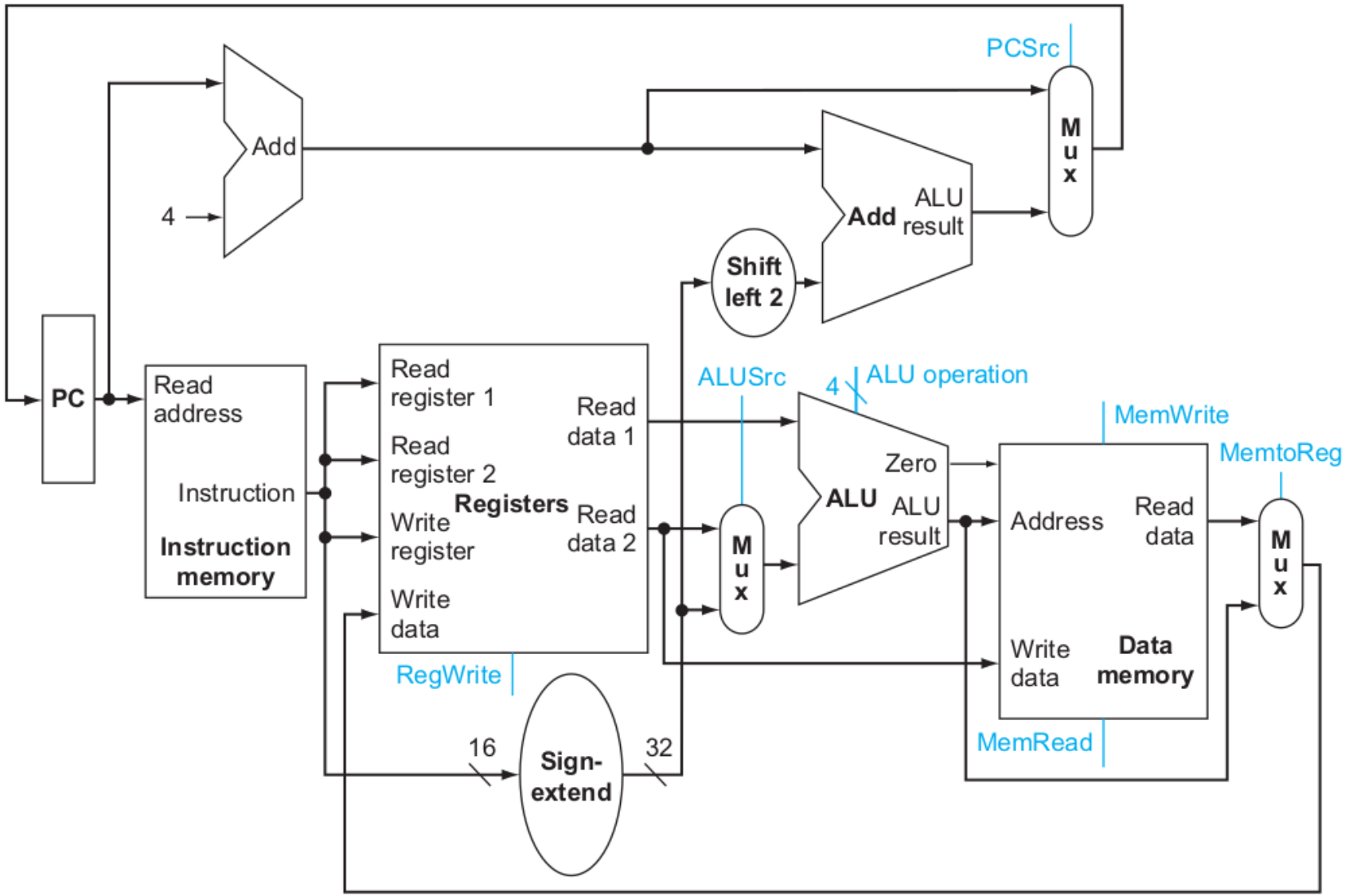
I-Type Conditional Branch Instructions

- Read two register operands
  - The first register source operand
  - The second operand is *rt*
- Compare operands
  - Use ALU, e.g., subtract and check Zero
- Calculate branch target address
  - Sign-extended displacement
  - Shift left by 2 bits (why?)
    - Word-alignment and increase the range
  - Add to PC + 4
    - PC-relative addressing mode



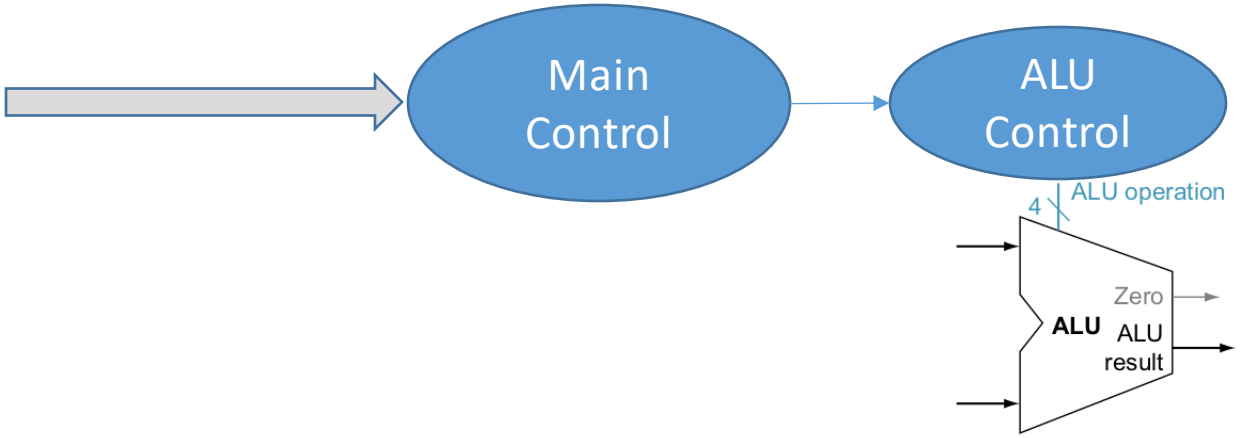


Combining Again



Adding a Simple Control Unit

- The control unit commands the datapath, memory, and I/O devices according to the executed instructions
- The control unit is often implemented via multiple levels
  - A main control unit with several smaller control units
    - The size of the main controller is reduced
    - The latency of the control unit may be potentially reduced



Desired ALU Action

- Recall what each instruction needs ALU for

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

Instruction opcode		Instruction	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	35	lw	load word	XXXXXX	add	0010
SW	43	sw	store word	XXXXXX	add	0010
Branch equal	4	beq	branch equal	XXXXXX	subtract	0110
R-type	0	add	add	100000	add	0010
R-type	0	sub	subtract	100010	subtract	0110
R-type	0	and	AND	100100	AND	0000
R-type	0	or	OR	100101	OR	0001
R-type	0	slt	set on less than	101010	set on less than	0111

The main control can use ALUOp control bits to tell ALU control what to do

ALU Control Unit

- ALU control is a sub one controlled by the main control via ALUOp
  - ALUOp is set by the main control according to the opcode
- ALU control functionality is captured by a truth table
  - We can use combinational logic to implement it

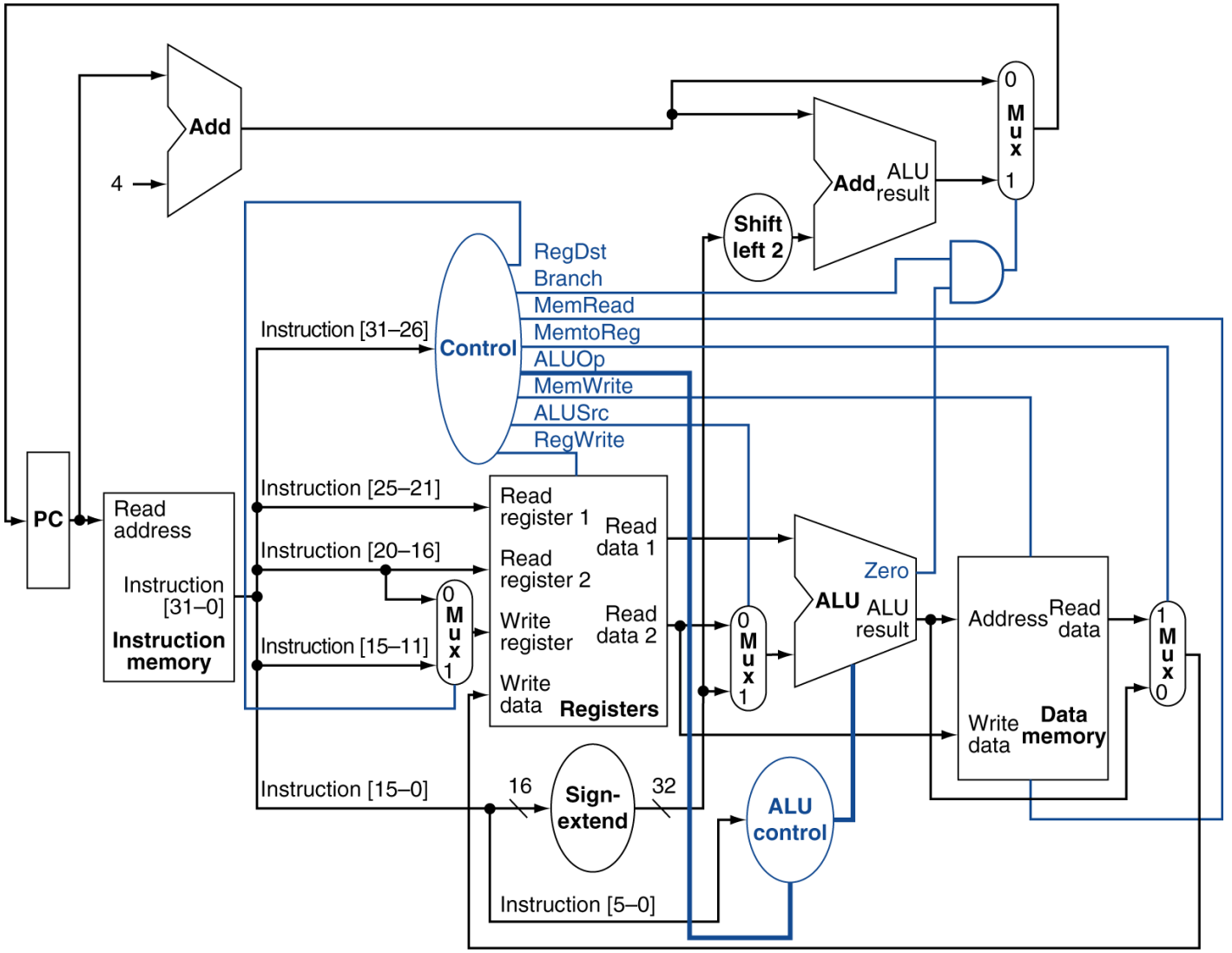
ALUOp	funct	ALU control
00	XXXXXX	0010
00	XXXXXX	0010
01	XXXXXX	0110
10	100000	0010
	100010	0110
	100100	0000
	100101	0001
	101010	0111

Main Control Unit

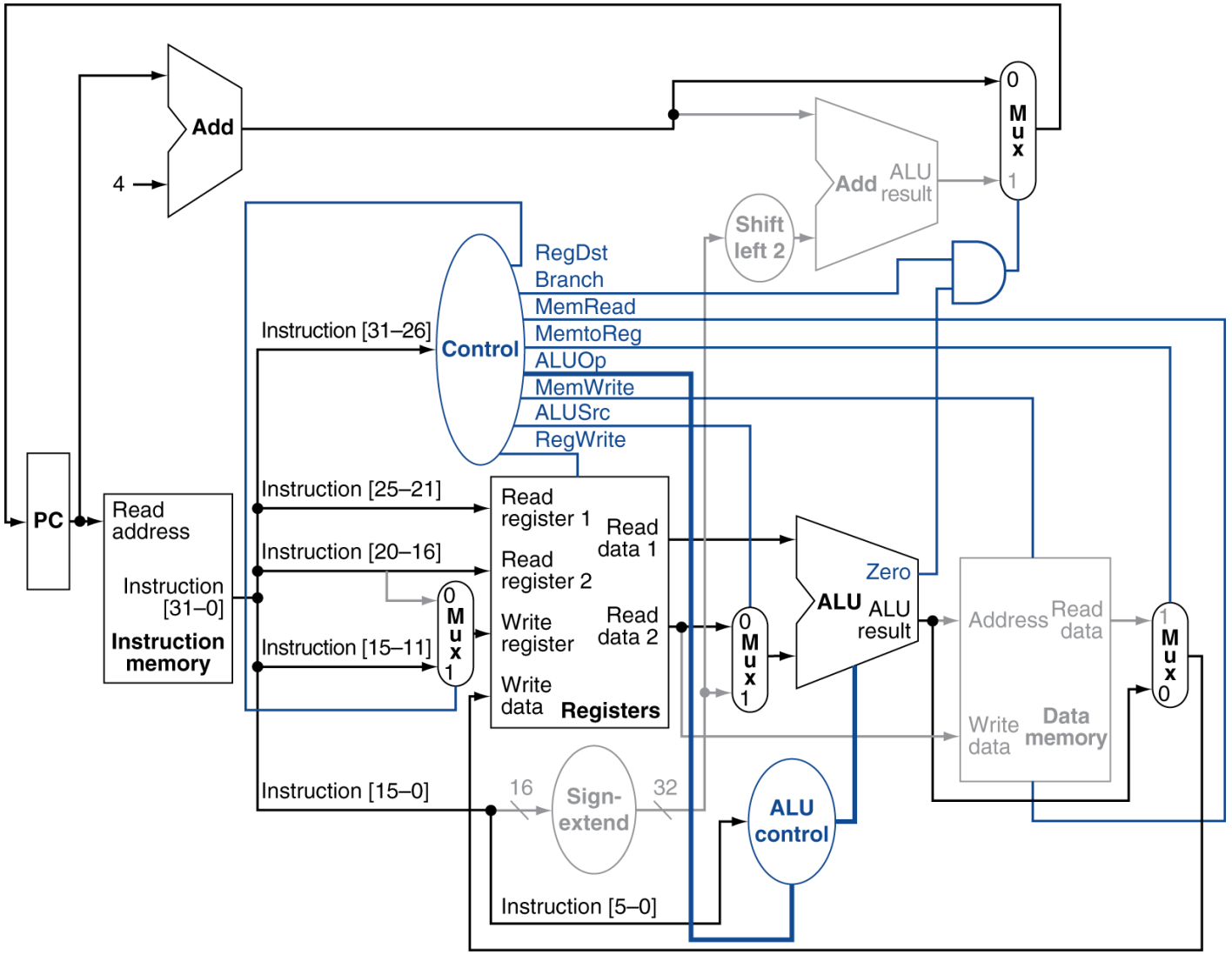
- Other than ALUOp, the main control unit set the following signals according to the instruction opcode

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.

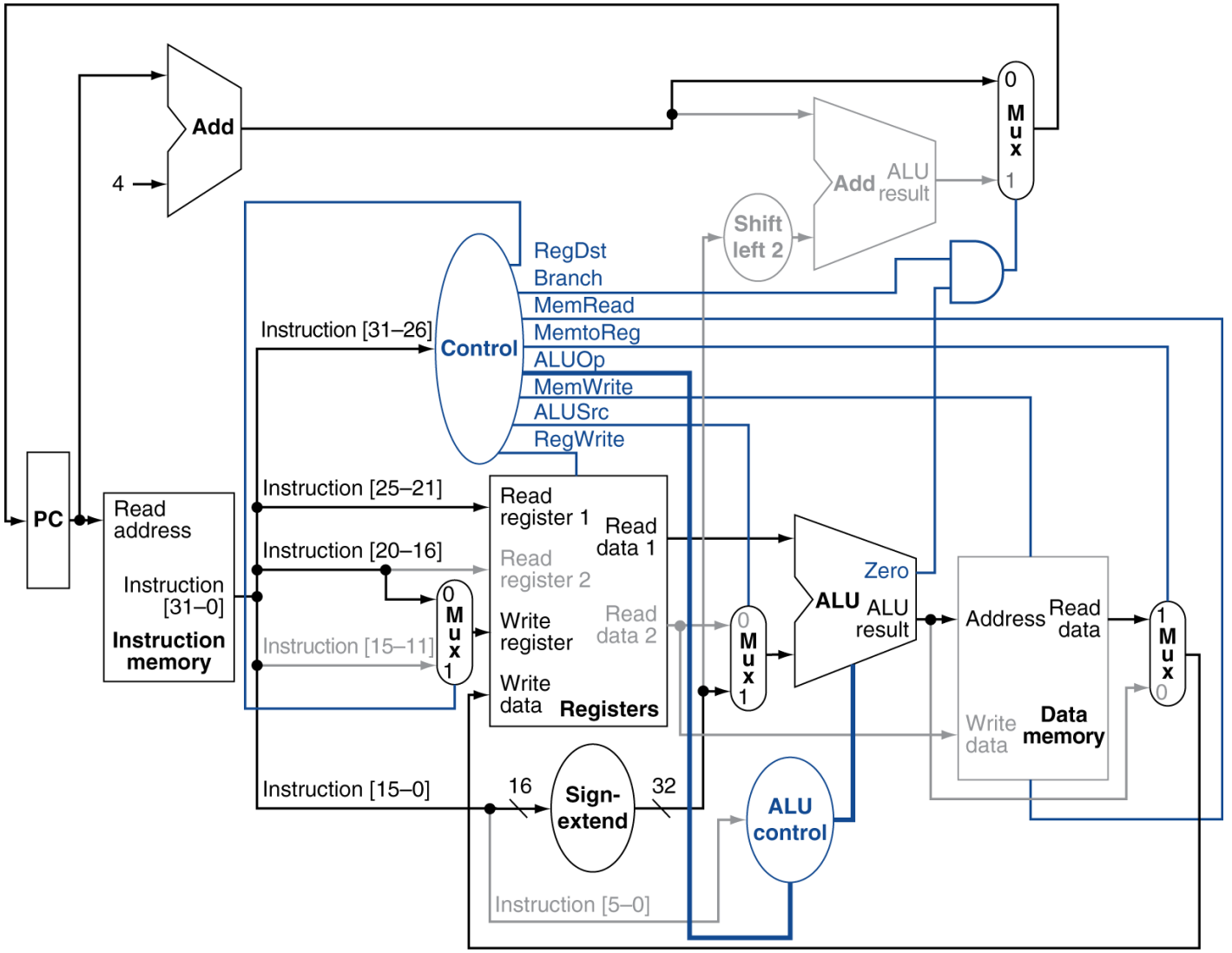
Datapath With Control



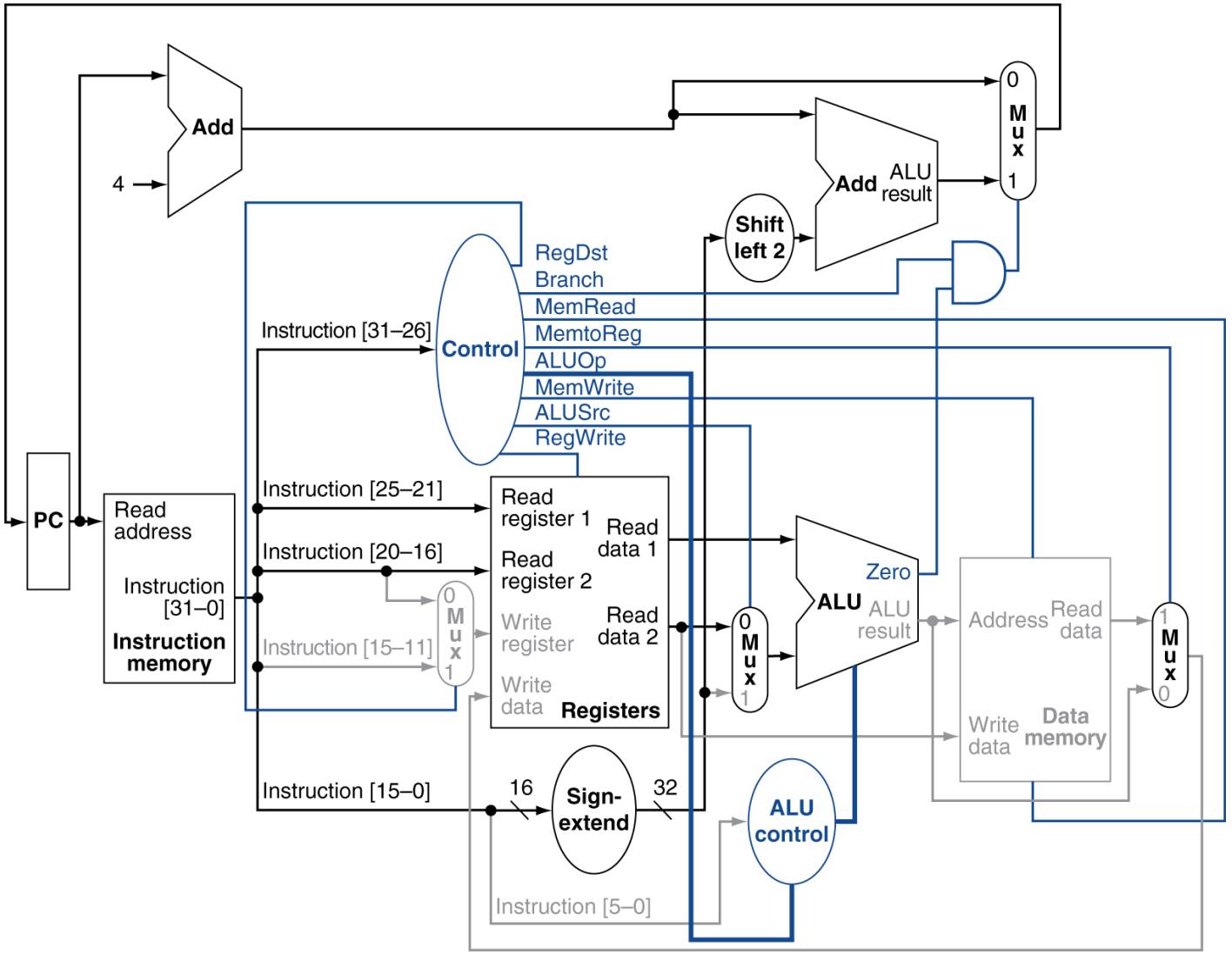
R-Type Instruction



Load Instruction

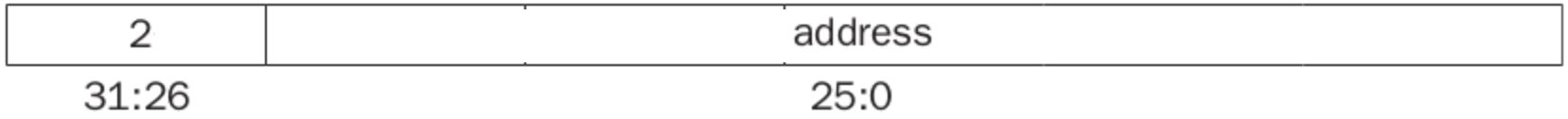


Branch-If-Equal Instruction

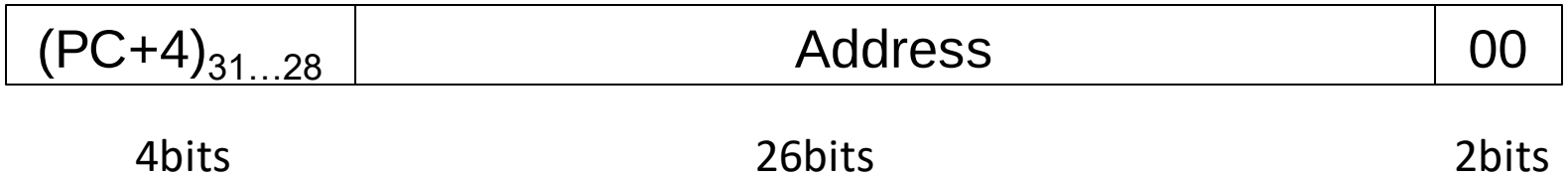


Implementing Jumps

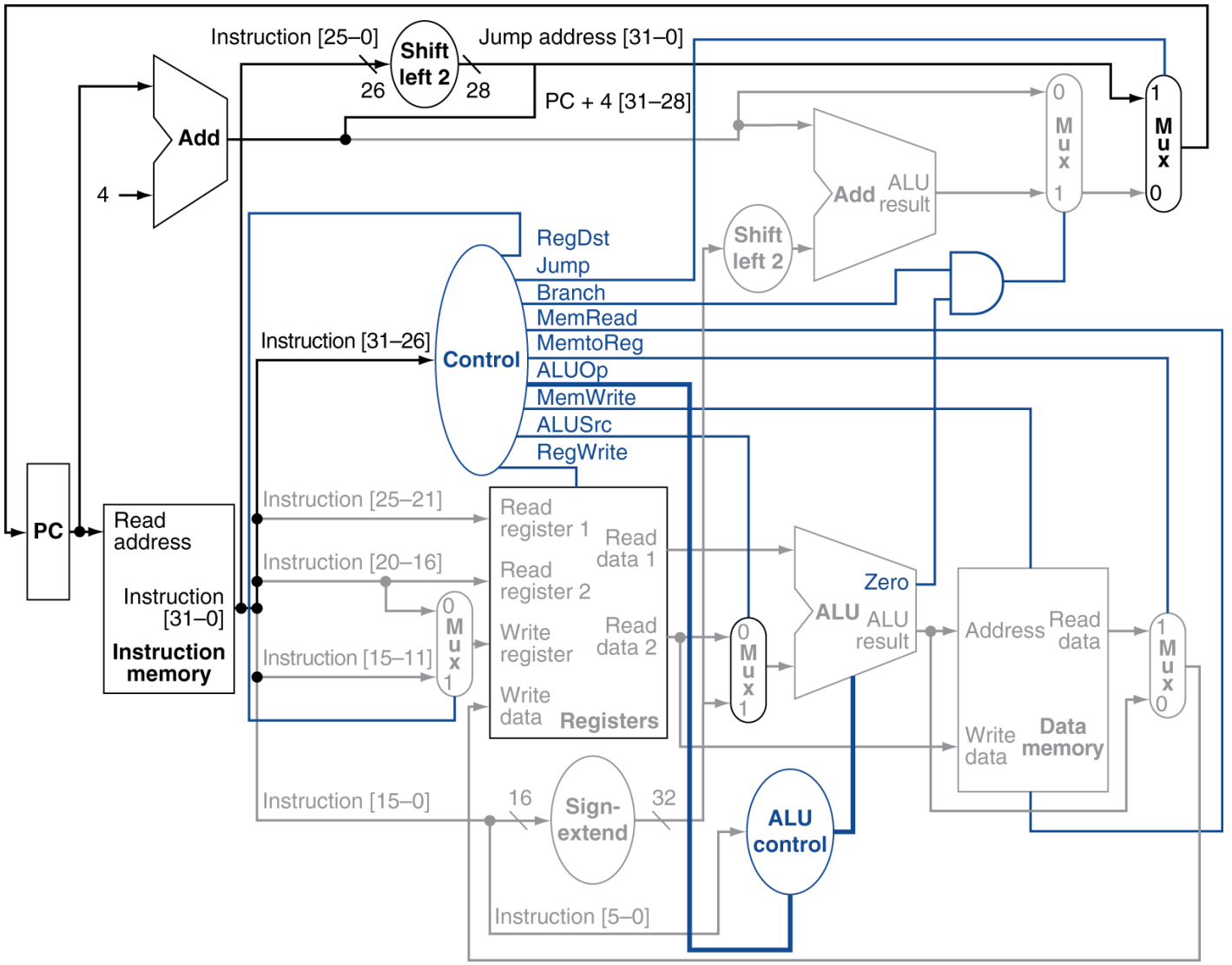
- Jump needs an extra control signal decoded from opcode



- How to derive the target address?
  - The most significant 4 bits of the next instruction's address (PC to jump + 4)
  - The 26-bit immediate fi eld of the jump instruction
  - The least significant 2 bits of 00



Datapath With Jumps Added



Next Lecture

- Section 4.6 – 4.7 (4.5 is optional)
  - Read the contents
  - Finish pre-class questions

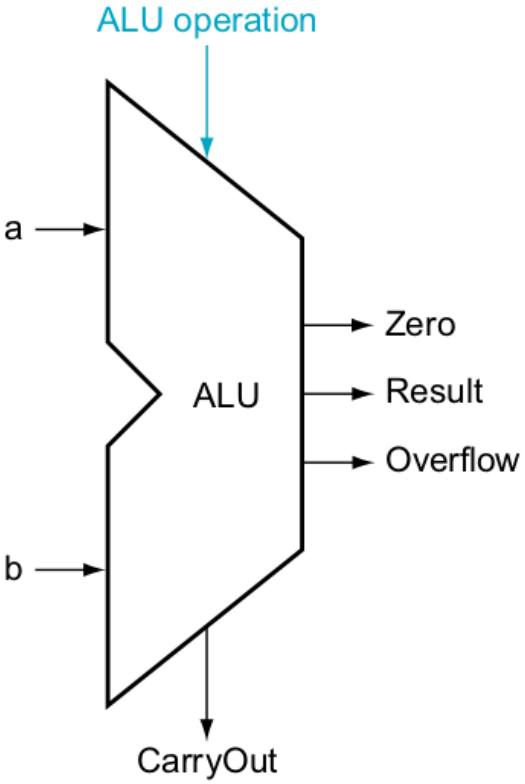
CPSC 3300-001
Computer Systems Organization

5. Arithmetic Logic Unit

Zhenkai Zhang

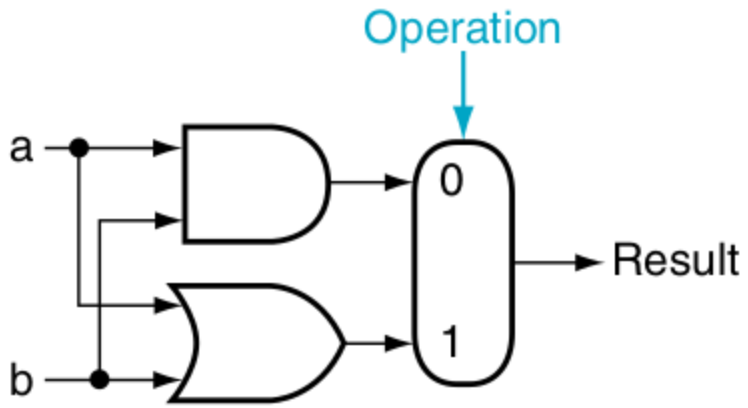
Arithmetic Logic Unit (ALU)

- The ALU performs the arithmetic operations or logical operations
  - Arithmetic operations
    - Addition, subtraction, etc.
  - Comparison operations
    - Less than, equal to, etc.
  - Logical operations
    - AND, OR, etc.
- If the registers are N-bit wide, we need a N-bit wide ALU



AND and OR

- Logic operations can map directly onto the hardware components



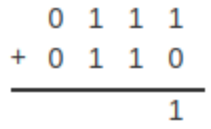
- 1-bit logic unit for AND and OR
  - If operation = 0, result = a AND b
  - If operation = 1, result = a OR b

Addition

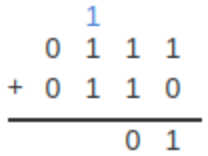
- We know how to add two decimal numbers
  - Starts at the right and adds each digit, possibly carrying a 1 to the next digit
  - How about adding two binary numbers?

7 + 6

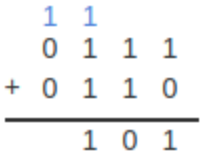
```
  0 1 1 1
+ 0 1 1 0
```



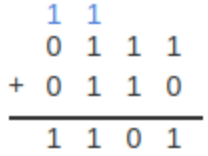
Add rightmost digit: 1 + 0 = 1



Add next digit: 1 + 1 = 10, so carry 1



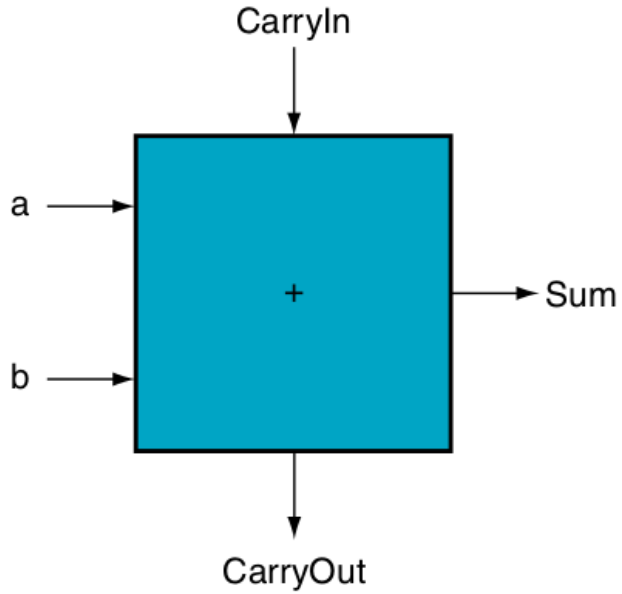
Add next digit: 1 + 1 + 1 = 11, so carry 1



Add the last digit: 1 + 0 + 0 = 1

1-Bit Adder

- A 1-bit adder has three inputs and two outputs
  - Two single-bit operands a and b, and a carry-in
  - A single-bit sum and a carry-out
  - This is called a full adder
    - If no carry-in input, it is called half adder



Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

CarryOut = a' b CarryIn + a b' CarryIn + a b CarryIn' + a b CarryIn

Sum = a' b' CarryIn + a' b CarryIn' + a b' CarryIn' + a b CarryIn

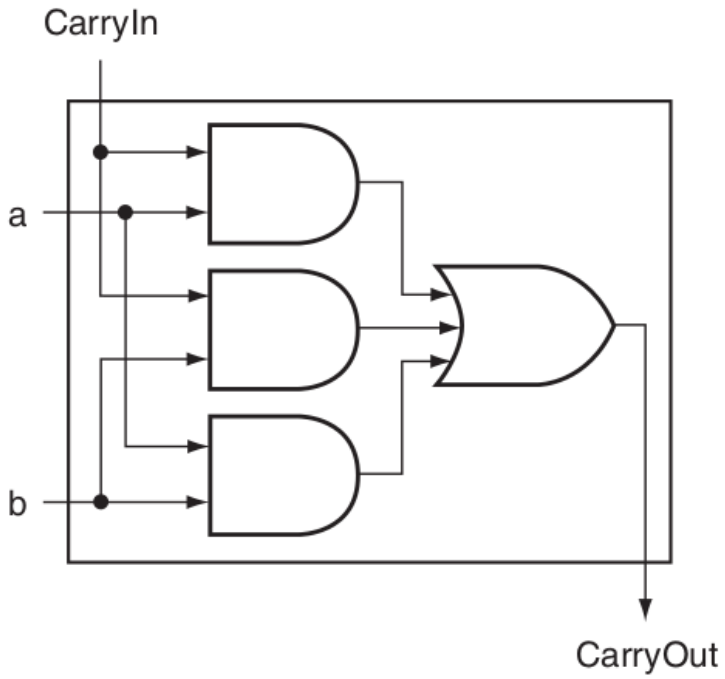
1-Bit Adder Carry-Out Signal

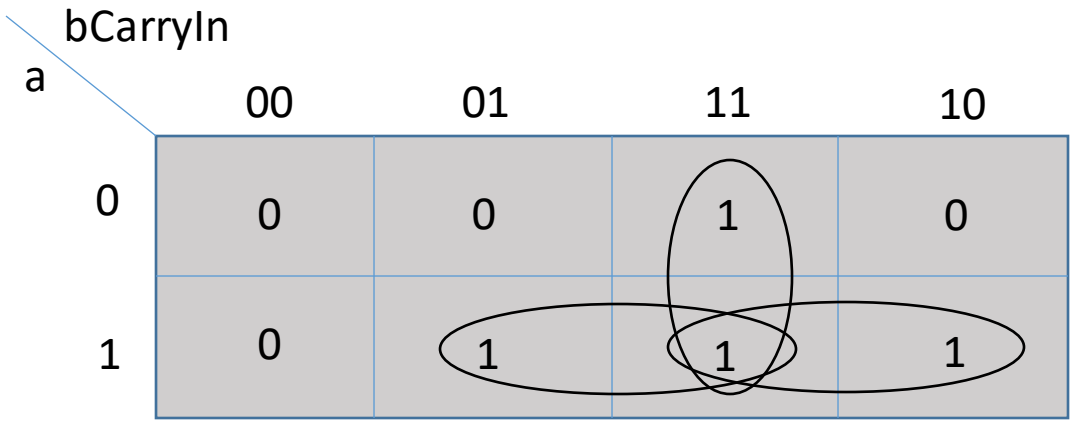
CarryOut = a' b CarryIn + a b' CarryIn + a b CarryIn' + a b CarryIn

= a' b CarryIn + a b' CarryIn + a b CarryIn' + (a b CarryIn + a b CarryIn + a b CarryIn) idempotent

= (a'+a) b CarryIn + (b'+b) a CarryIn + (CarryIn'+CarryIn) a b distributive

= a b + a CarryIn + b CarryIn complement



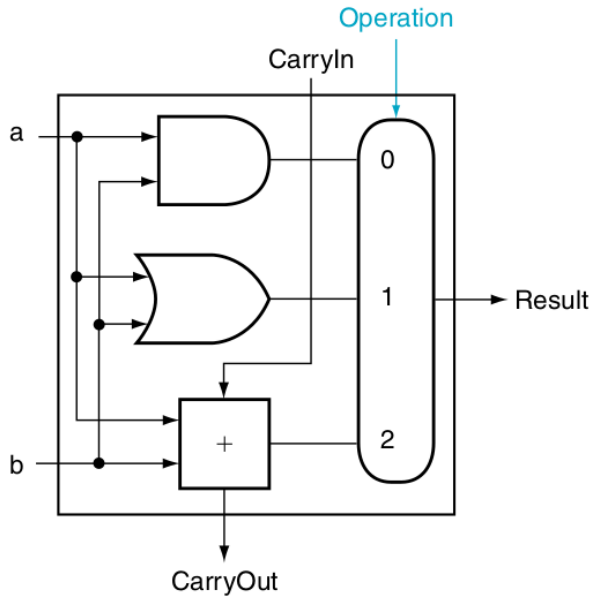


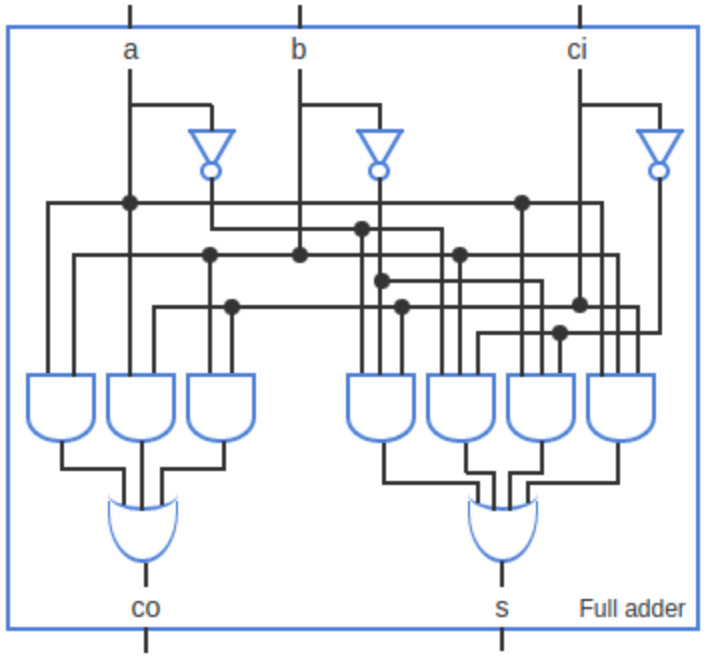
Using K-map can easily simplify the equation

1-Bit Adder Sum Signal

Sum = a' b' CarryIn + a' b CarryIn' + a b' CarryIn' + a b CarryIn

This Boolean equation cannot be simplified!



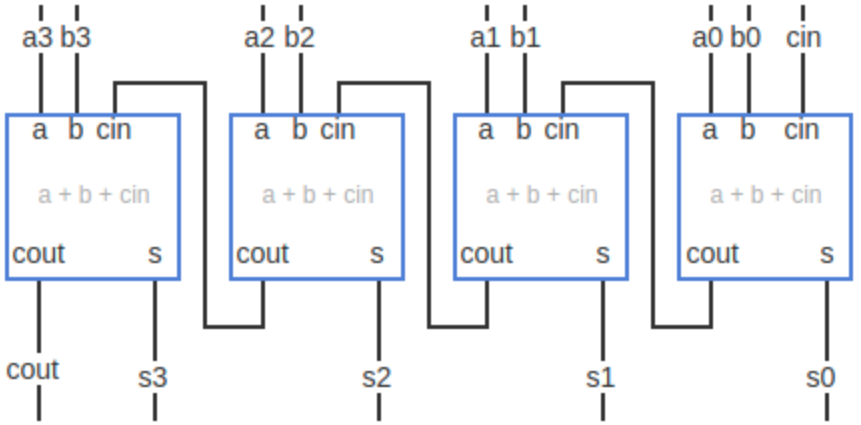


If we can use XOR, Sum is just a XOR b XOR CarryIn (odd number of 1's)

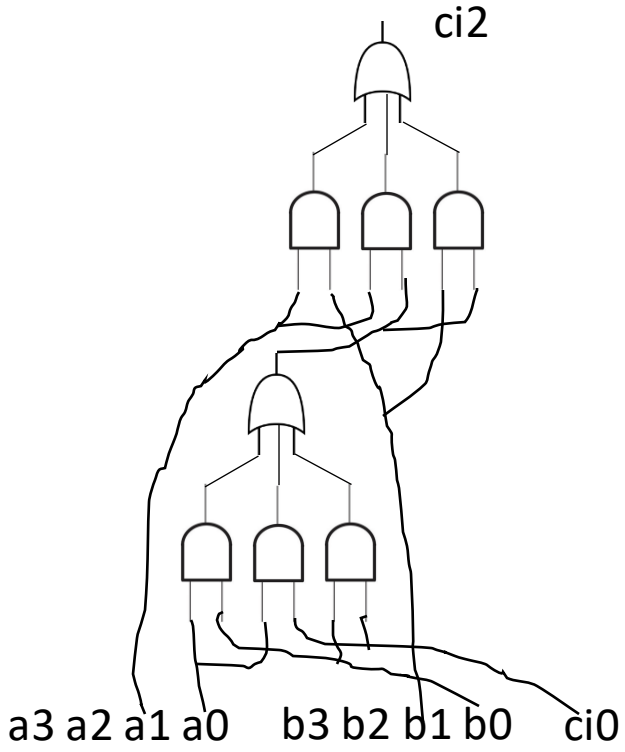
N-Bit Adder (Ripple Carry Version)

- We can directly link the carries of 1-bit adders to form an N-bit adder
  - This is called a ripple carry adder

ci1 = co0 = a0 b0 + a0 ci0 + b0 ci0
ci2 = co1 = a1 b1 + a1 ci1 + b1 ci1



What kind of problem does this implementation have?

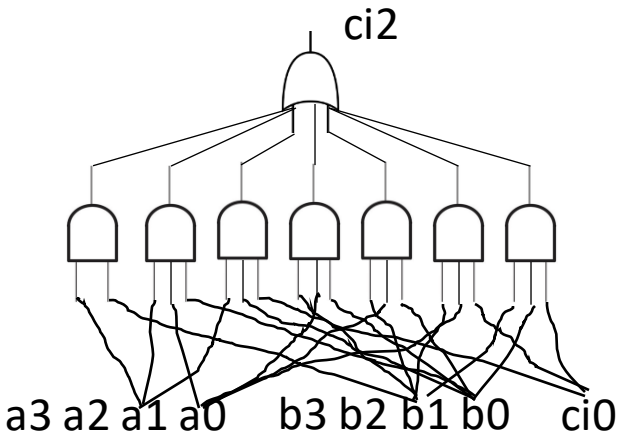


Faster Carry-In Determination

ci1 = co0 = a0 b0 + a0 ci0 + b0 ci0

ci2 = co1 = a1 b1 + a1 ci1 + b1 ci1 = a1 b1 + a1 (a0 b0 + a0 ci0 + b0 ci0) + b1 (a0 b0 + a0 ci0 + b0 ci0)

= a1 b1 + a1 a0 b0 + a1 a0 ci0 + a1 b0 ci0 + b1 a0 b0 + b1 a0 ci0 + b1 b0 ci0



- Each carry-in bit can be determined using a two-level logic
  - But it grows exponentially
    - ci1 – 3 product terms
    - ci2 – 7 product terms
    - ci3 – 15 product terms
    - ci4 – 31 product terms

Generate and Propagate

$ci1 = co0 = a0\ b0 + a0\ ci0 + b0\ ci0 = a0\ b0 + (a0 + b0)\ ci0$ distributive

$ci2 = co1 = a1\ b1 + a1\ ci1 + b1\ ci1 = a1\ b1 + (a1 + b1)\ ci1$

$= a1\ b1 + (a1 + b1)\ (a0\ b0 + (a0 + b0)\ ci0)$

$= a1\ b1 + (a1 + b1)\ a0\ b0 + (a1 + b1)\ (a0 + b0)\ ci0$

- We define two terms – generate (gi) and propagate (pi)
  - $ci_{i+1} = gi + pi\ ci_i$
  - $gi = ai\ bi$
    - If ai = 1 and bi = 1, a carry-out must be **generated** no matter what its carry-in is
  - $pi = ai + bi$
    - Suppose gi is 0 and pi is 1, then $ci_{i+1} = gi + pi\ ci_i = ci_i$
    - It **propagates** the carry-in (ci_i) to the carry-out (ci_{i+1})

4-Bit Carry-Lookahead Logic

ci1 = a0 b0 + (a0 + b0) ci0 = g0 + p0 ci0
ci2 = a1 b1 + (a1 + b1) ci1 = g1 + p1 g0 + p1 p0 ci0
ci3 = a2 b2 + (a2 + b2) ci2 = g2 + p2 g1 + p2 p1 g0 + p2 p1 p0 ci0
ci4 = a3 b3 + (a3 + b3) ci3 = g3 + p3 g2 + p3 p2 g1 + p3 p2 p1 g0 + p3 p2 p1 p0 ci0

- Both gi (an AND gate) and pi (an OR gate) can be easily got
  - ci1 – 2 product terms
  - ci2 – 3 product terms
  - ci3 – 4 product terms
  - ci4 – 5 product terms
- Each carry-in bit can be determined using a three level logic
  - 1st level – gi and pi
  - 2nd level – products
  - 3rd level – sum

Two-Level Carry-Lookahead Logic

- We can use 4-bit carry-lookahead logic to form a 16-bit one
  - We define super propagate (Pi) and generate (Gi) signals for 4-bit blocks

P0 = p3 p2 p1 p0
P1 = p7 p6 p5 p4
P2 = p11 p10 p9 p8
P3 = p15 p14 p13 p12

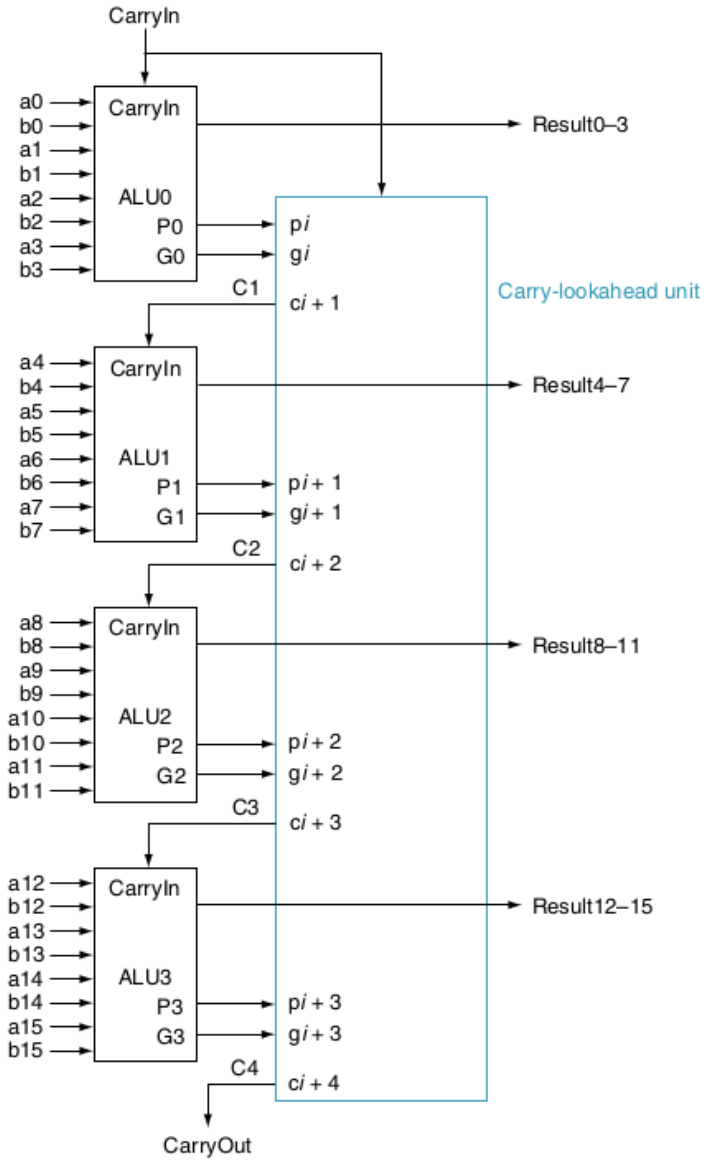
How to propagate the carry-in to the carry-out

G0 = g3 + p3 g2 + p3 p2 g1 + p3 p2 p1 g0
G1 = g7 + p7 g6 + p7 p6 g5 + p7 p6 p5 g4
G2 = g11 + p11 g10 + p11 p10 g9 + p11 p10 p9 g8
G3 = g15 + p15 g14 + p15 p14 g13 + p15 p14 p13 g12

How to generate a carry-out

Ci1 = G0 + P0 ci0
Ci2 = G1 + P1 G0 + P1 P0 ci0
Ci3 = G2 + P2 G1 + P2 P1 G0 + P2 P1 P0 ci0
Ci4 = G3 + P3 G2 + P3 P2 G1 + P3 P2 P1 G0 + P3 P2 P1 P0 ci0

Two-Level Carry-Lookahead Adder



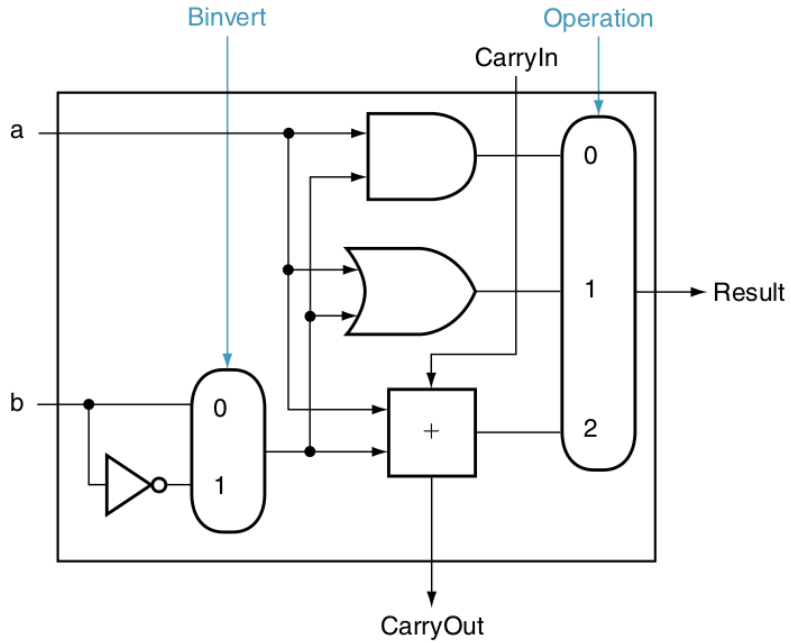
Speed of Ripple Carry v.s. Carry Lookahead

- Assume a 16-bit adder, if we use ripple carry, how many logic levels are needed to get the last carry-out bit (co15)?
  - To get ci15, we need 30 levels, then we need 2 additional levels for co15
    - 32 logic levels
- If we use two-level carry-lookahead, how many logic levels?
  - To get gi and pi, we need 1 level
  - To get Gi and Pi, we need 2 levels (although we just need 1 level to get Pi)
  - To get C4 (i.e., co15), we need 2 levels
    - 5 logic levels

Subtraction

- We know that a – b equals to a + (-b)
  - How to negate a two's complement number?
    - Invert every bit and then add 1
  - For each bit of subtrahend, we need is an invertor
  - We also need to add an additional 1

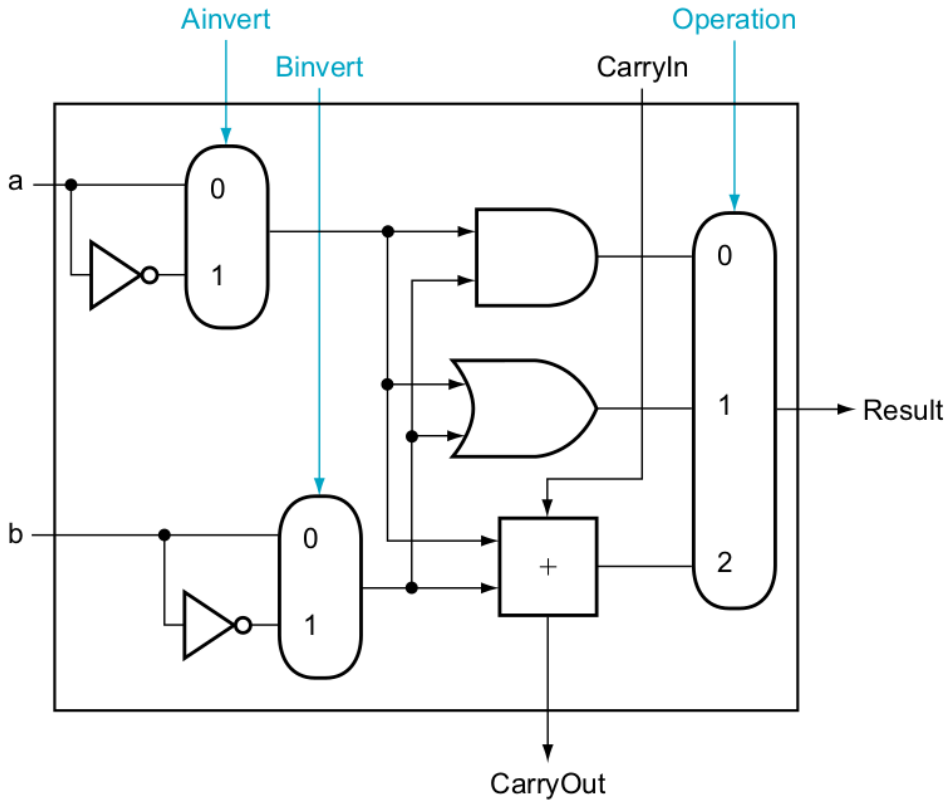
00001101
(1) 11110010
(2) 11110011



A carry-in to the full adder for the least significant bit suffices

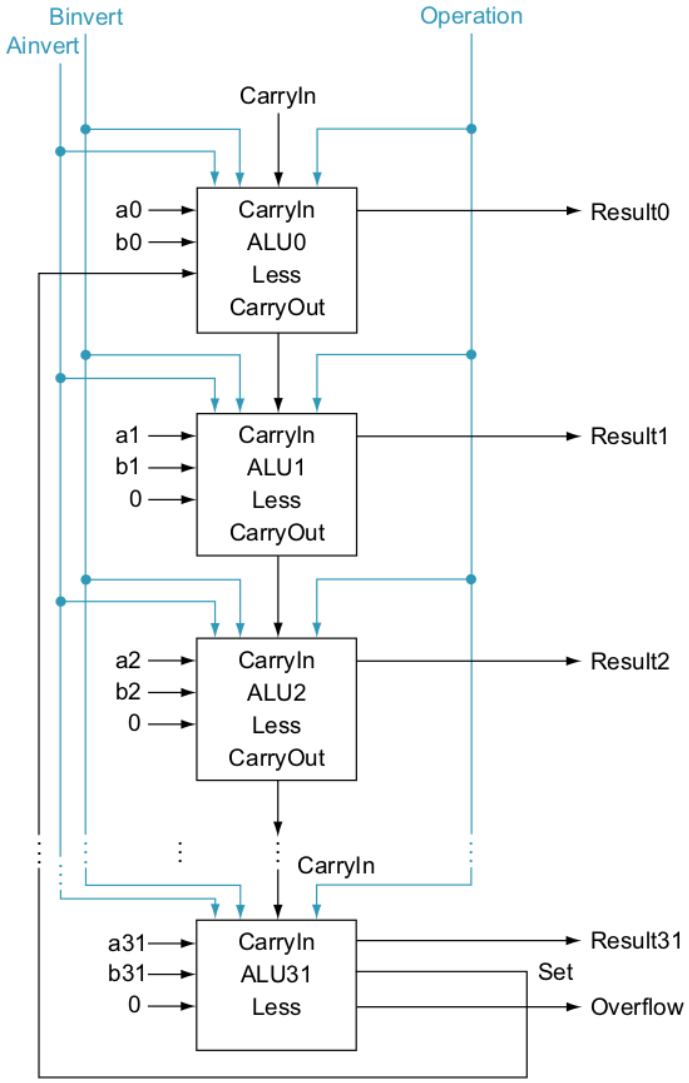
NOR and NAND

- To implement either NOR or NAND, DeMorgan's law can help
  - a NOR b = (a OR b)' = a' AND b'
  - a NAND b = (a AND b)' = a' OR b'



Less-than Comparison

- a < b means a – b < 0
  - Sign bit comes to rescue – the sign bit of a – b should be 1 if a < b; otherwise a >= b
    - The most significant bit
- The set less than instruction (slt) produces 1 if a < b, and 0 otherwise
  - If a < b, we also need to set the least significant bit, and other bits are 0



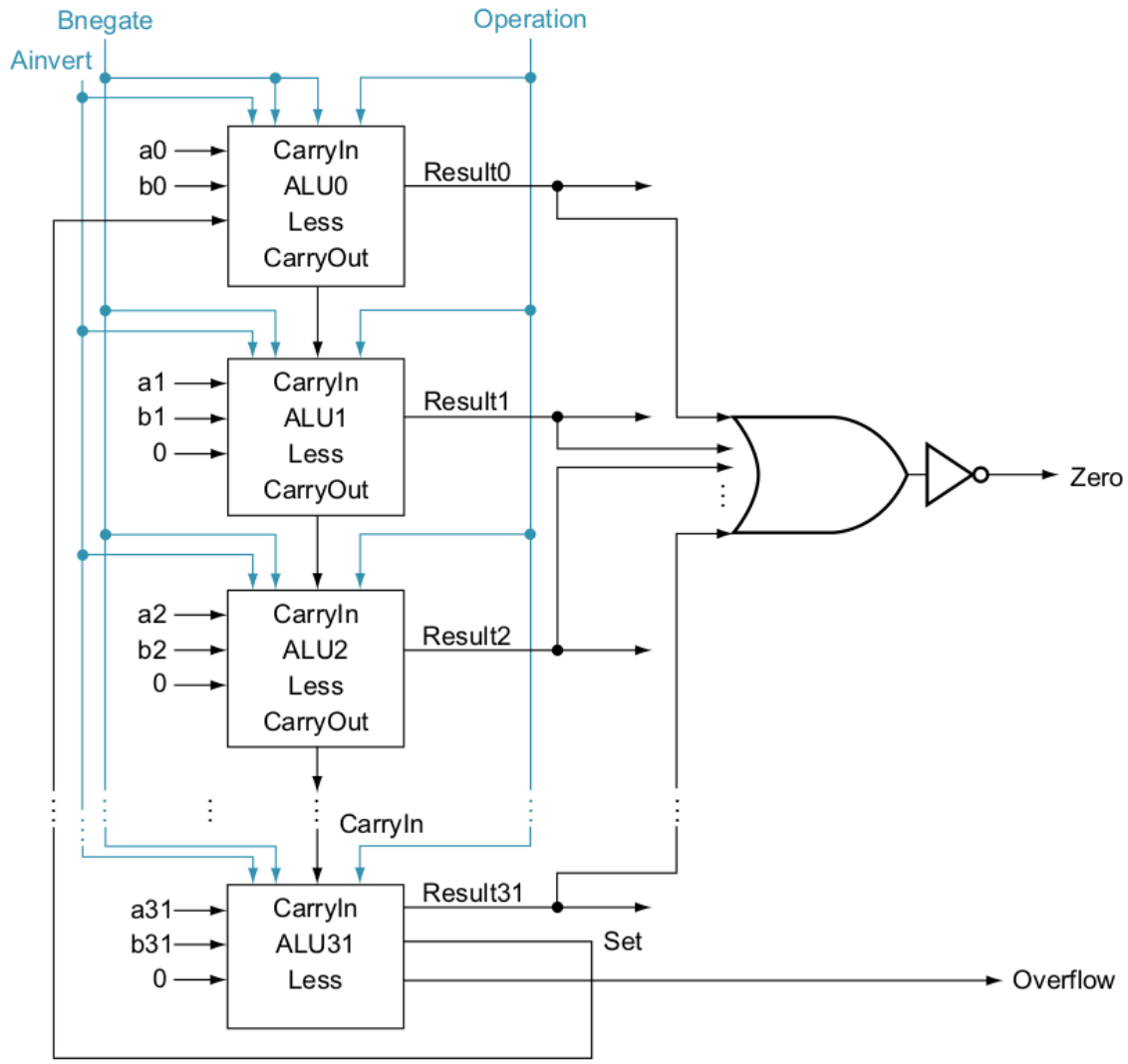
Overflow

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

- What if an overflow occurs during less-than comparison?
  - For example, a = -7, b = 6, on a 4-bit ALU
    - a – b = 1001 + 1010 = (1) 0011 → sign bit is 0, so -7 is not less than 6?
- A simple trick to check overflow – the carry-in to the most significant bit is not the same as the carry-out of the most significant bit
  - How can we quickly check if two bits are the same or not?
    - XOR
- What to do with Set bit and Overflow check bit?

Equal-to Comparison

- a = b means a – b = 0
  - NOR every bit of a – b
    - If a – b is 0, every bit is 0
      - NOR gives 1, namely Zero is true
    - If a – b is not 0, at least one bit is 1
      - NOR gives 0, namely Zero is false



Next Lecture

- Section B.7 – B.9
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

6. Sequential Logic I

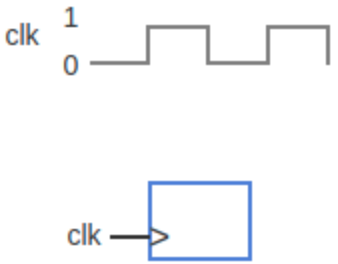
Zhenkai Zhang

Combinational Logic v.s. Sequential Logic

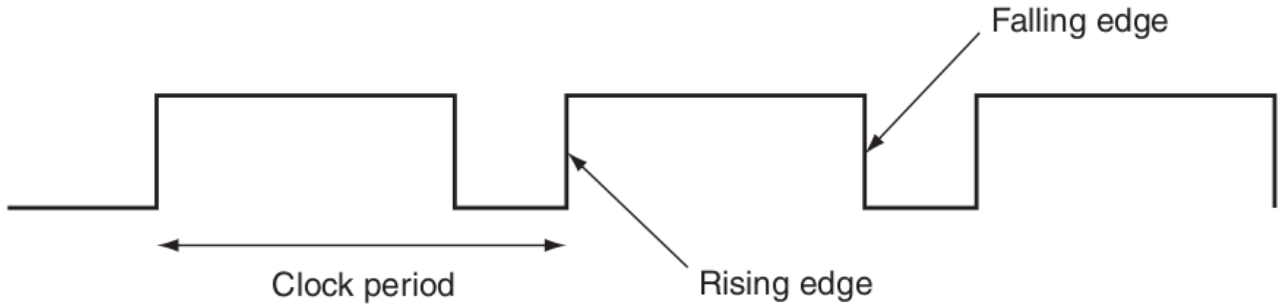
- Recall the difference between combinational logic and sequential logic
  - A combinational logic circuit's output depends only on the present combination of input values
  - A sequential logic circuit's output depends on the present and the past sequence of input values
- For sequential logic, we need memory elements to store states
  - Usually, a clock signal is used to control memory elements when to store

Clock

- A clock signal is an oscillating signal
  - Periodically alternates between 0 and 1
    - Clock cycle time is a period
    - Clock frequency is 1/period
  - Generated by a crystal oscillator



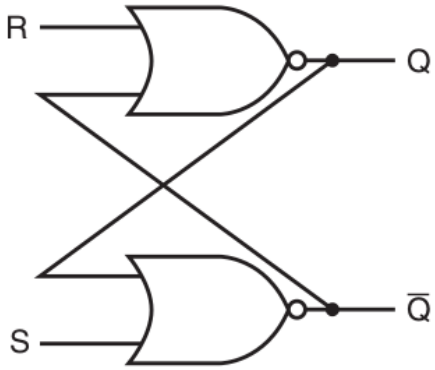
- Transition from 0 to 1 is called rising edge (positive edge)
- Transition from 1 to 0 is called falling edge (negative edge)



SR Latch

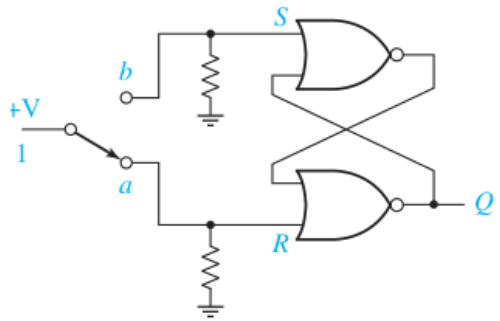
a	b	NOR
0	0	1
0	1	0
1	0	0
1	1	0

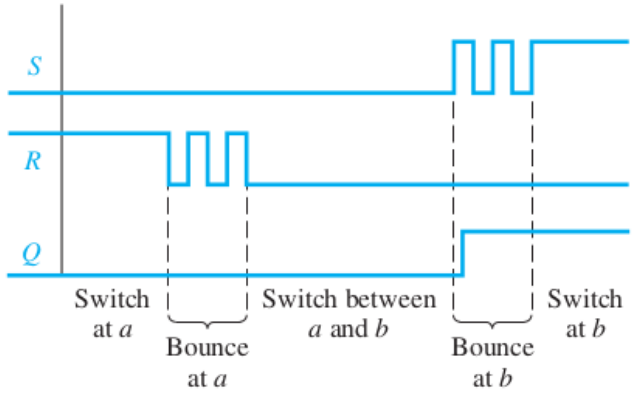
- An SR latch can be designed with two cross-coupled NOR gates
  - Inputs: S (set) and R (reset)
  - Outputs: Q and Q'
  - It is not clocked
- SR latch behavior
  - When S = 1 and R = 0, Q = 1 and Q' = 0
  - When S = 0 and R = 1, Q = 0 and Q' = 1
  - When S = 0 and R = 0, Q and Q' remain
  - What about S = 1 and R = 1?
    - No bit was stored, and instead the latch oscillates



Debouncing Switches

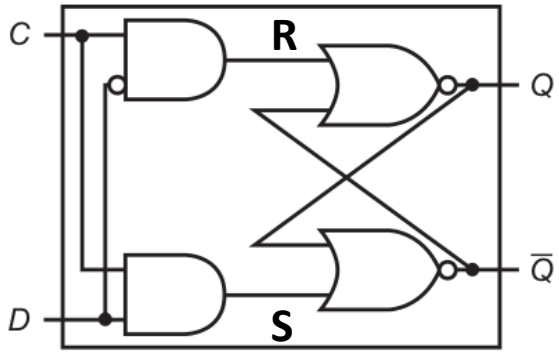
- Normally, SR latches are not directly used
  - They are often used for building more complex latches and flip-flops
- A direct use of SR latches is to build debouncing switches
  - When a mechanical switch is opened or closed, the switch contacts tend to bounce open and closed several times before settling down
  - This produces a noisy transition, and this noise can interfere with the proper operation of a logic circuit





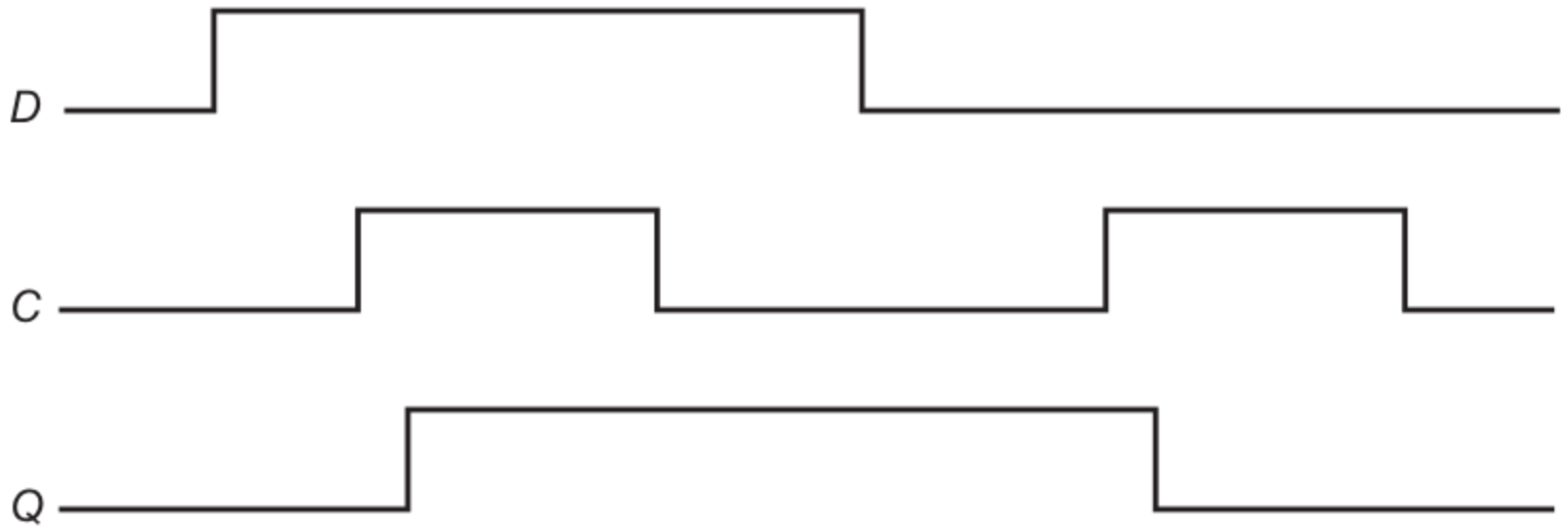
D Latch

- A D latch can be implemented using an internal SR latch
  - Inputs: D (data) and C (clock)
  - Outputs: Q and Q'
  - It is a clocked latch
- D latch behavior (to facilitate discussion, S and R are introduced)
  - When C = 0 (clock is low), S and R are always 0
    - Q and Q' remain
  - When C = 1 (clock is high), S is D and R is D'
    - S and R are always complements to each other
    - If D is 1, Q becomes 1, and if D is 0, Q becomes 0
- A clocked latch is said to be level-sensitive



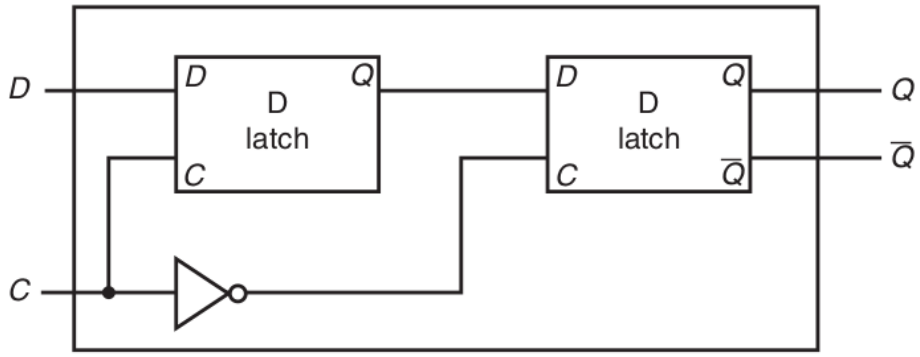
D Latch Timing diagram

- Assuming the output Q is initially deasserted



Edge-Triggered D Flip-Flops

- A D flip-flop can be implemented by cascading two D latches
  - The first latch's Q is the second latch's D
  - Other implementations exist

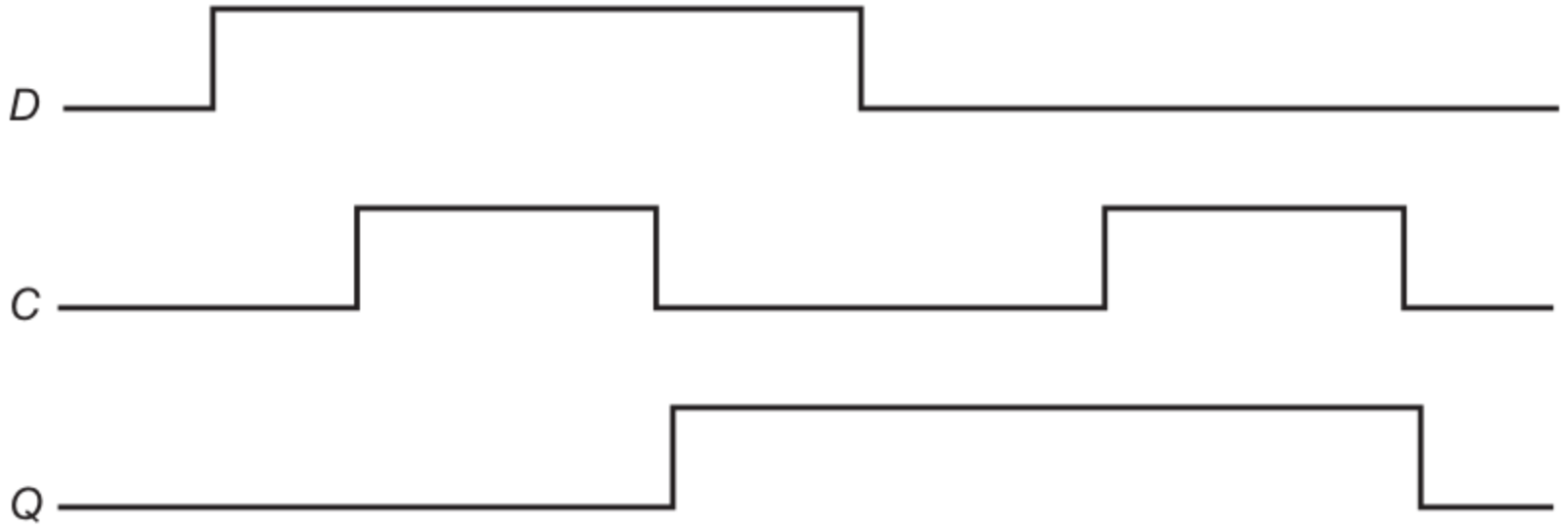


- (Falling-edge-triggered) D flip-flop behavior
  - When C = 1, the first latch is open
    - The first latch's Q gets D
  - When C = 0, the first latch is closed, but the second latch is open
    - The second latch's Q gets its input from the output of the first latch
  - It stores D only on the falling edge
- A flip-flop is said to be edge-triggered

How can we change this to a rising-edge-triggered flip-flop?

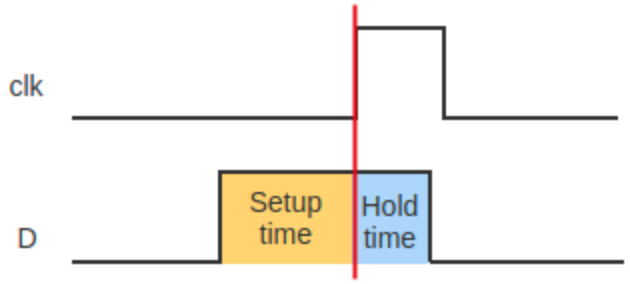
D Flip-Flop Timing diagram

- Assuming the output Q is initially deasserted



Setup Time and Hold Time

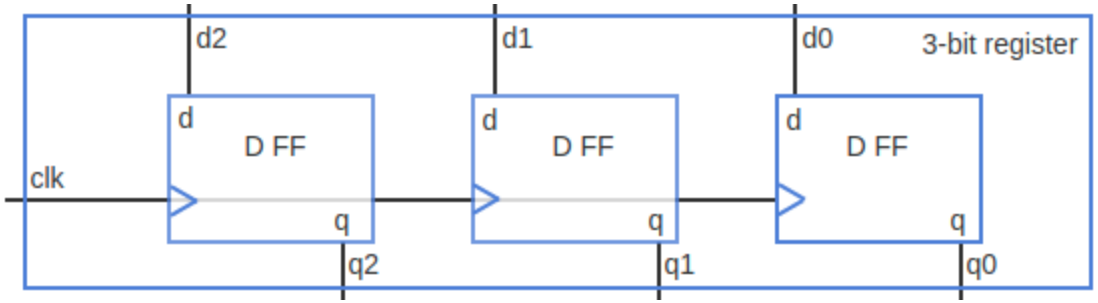
- There are some timing requirements for a D flip-flop
  - The input signal must be stable for at least t_{setup} before the clock edge
    - The minimum time t_{setup} is called the setup time
  - The input signal must hold stable for at least t_{hold} after the clock edge
    - The minimum time t_{hold} is called the hold time



What if we failed meeting such requirements? Flip-flop may go into a metastable state (can have a voltage that is a value between 0 and 1)

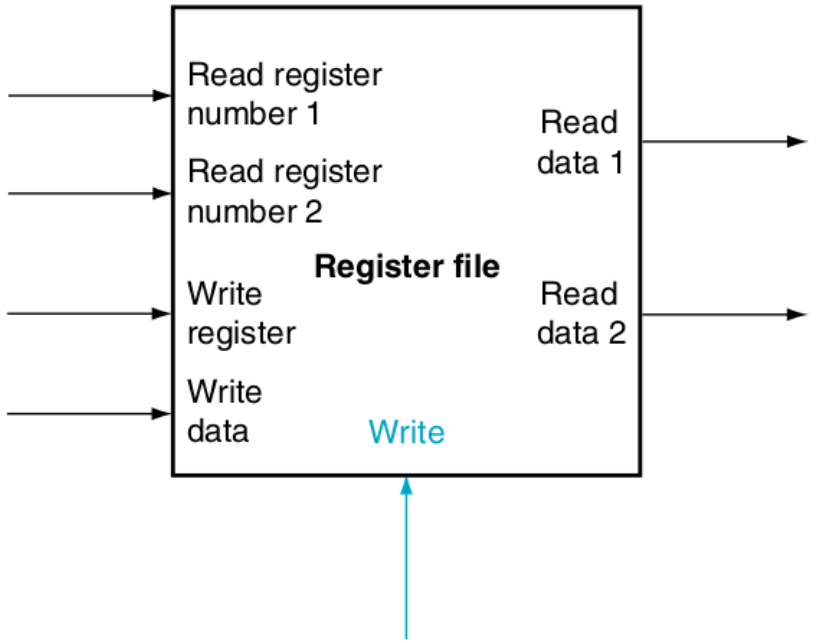
Register

- A register is a circuit that stores a group of bits (e.g., 32 bits or 64 bits)
  - On a rising/falling clock edge, all bits are stored simultaneously
    - Storing bits in a register is known as loading the register
  - A common register implementation uses flip-flops

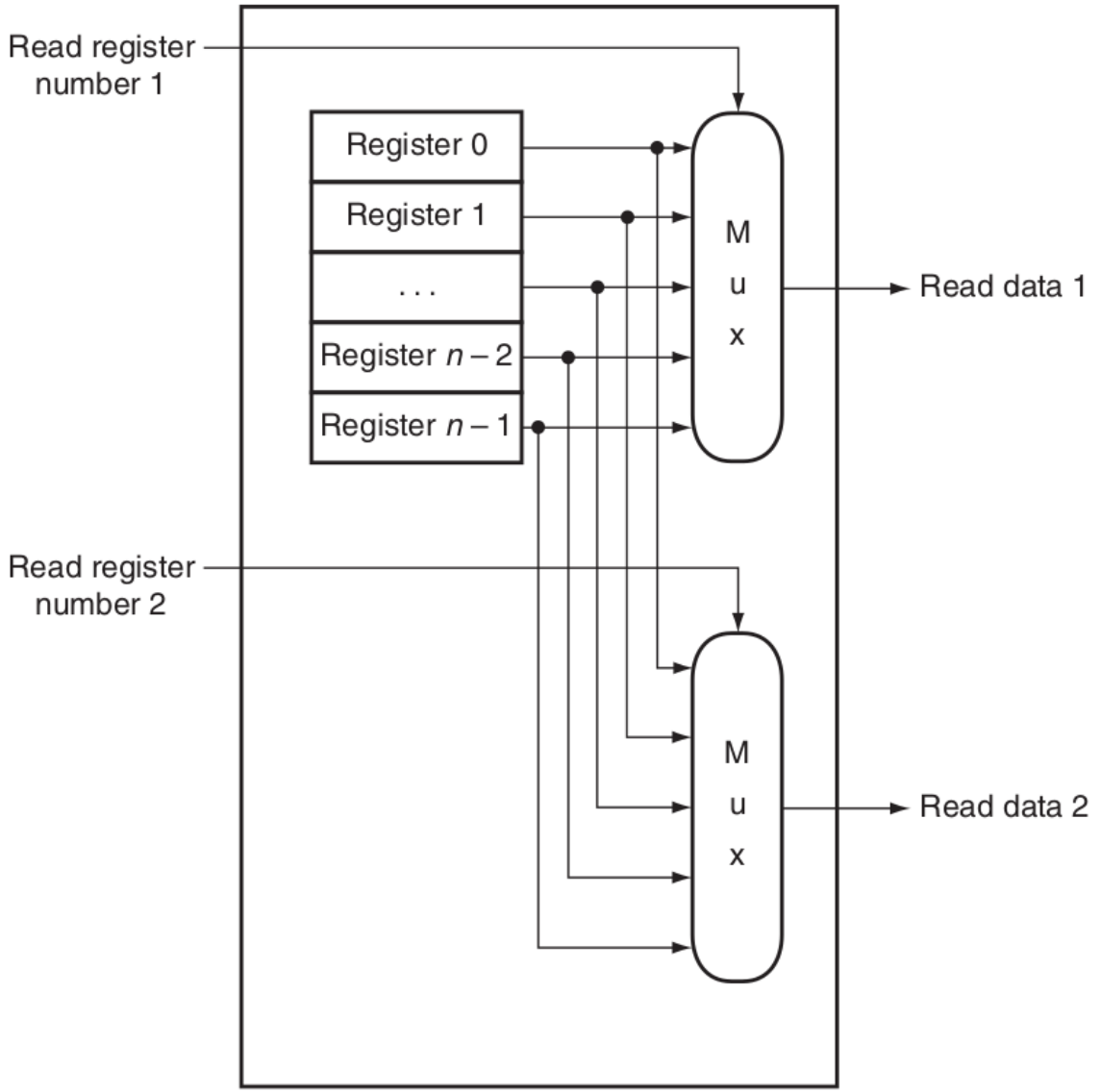


Register File

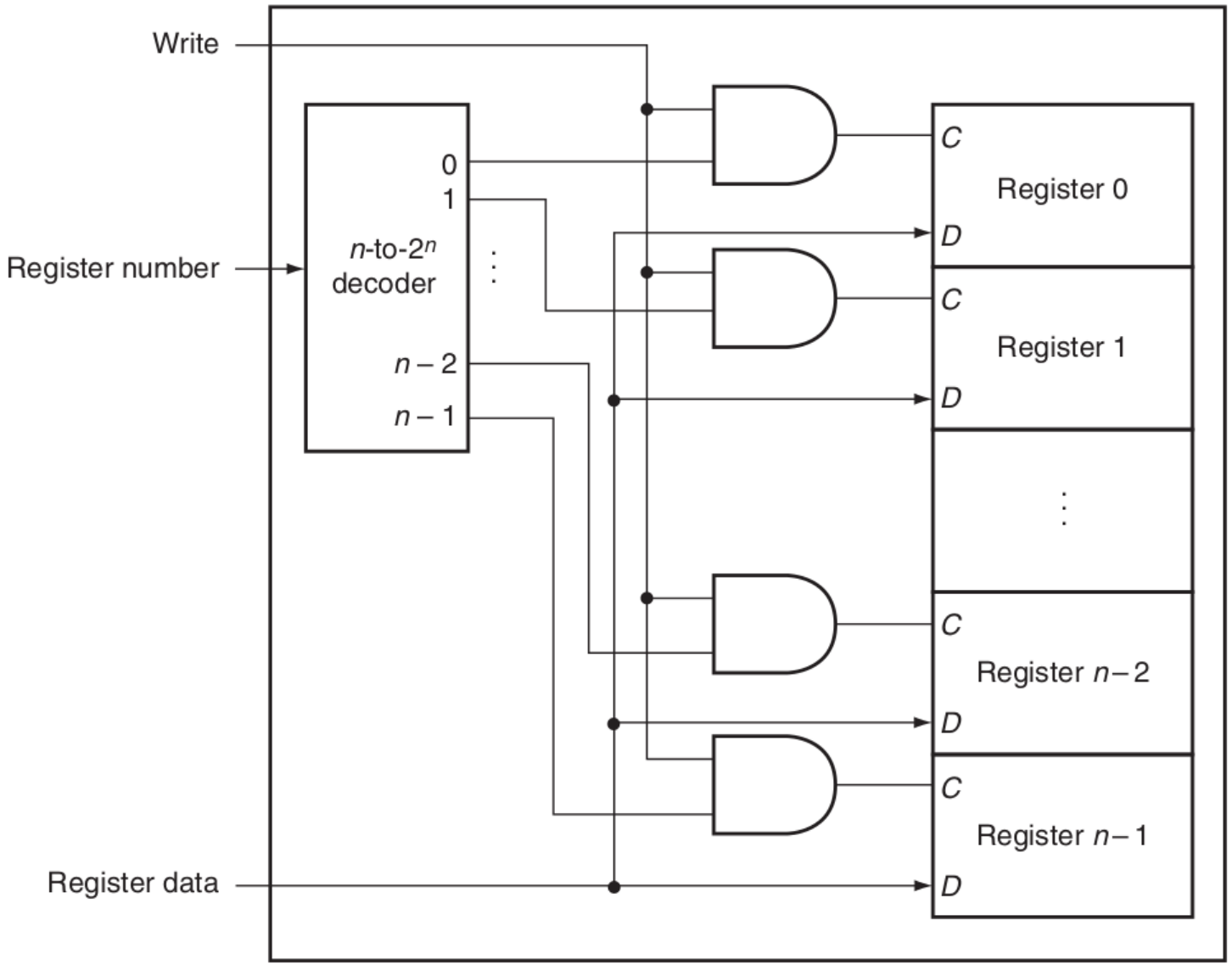
- A register file is a set of registers that can be read and written by supplying a register number to be accessed
  - A register file has read port(s) and write port(s)
    - Implemented with muxes and decoders



Read Port Implementation

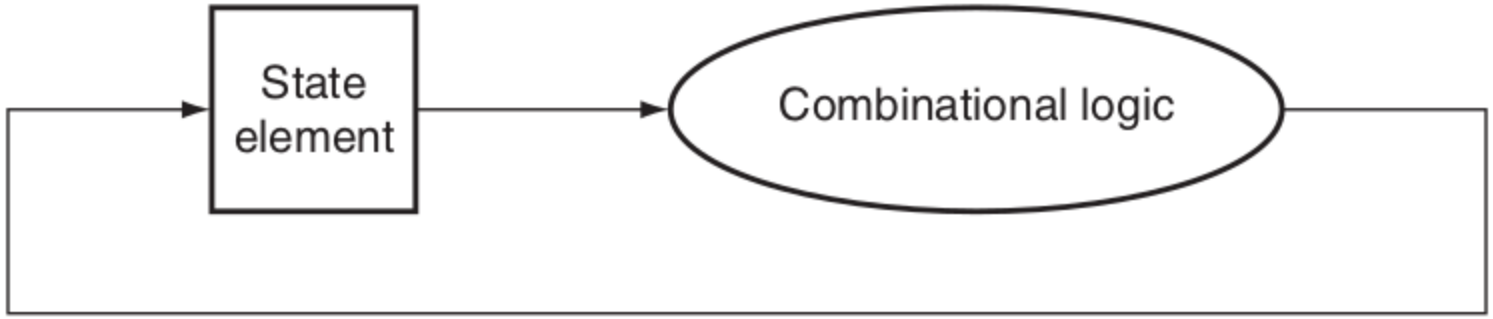


Write Port Implementation



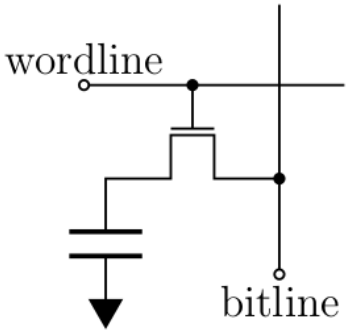
Sequential Logic Circuit

- A sequential logic circuit contains one or more combinational logic blocks and certain edge-triggered memory elements to store state
  - It normally has state in a feedback loop with the logic
  - The next state depends on the inputs and the present state
  - The output depends on the present state and perhaps also the inputs



SRAM v.s. DRAM

- SRAM stores each bit in a bi-stable memory cell
  - Not sensitive to disturbance
  - Each cell is implemented with a six-transistor circuit
  - An SRAM cell retains its value indefinitely, as long as it is kept powered
  - Used to build caches
- DRAM stores each bit as charge on a capacitor
  - DRAM storage can be made very dense
    - Each cell consists of a capacitor and a single access transistor
  - Unlike SRAM, a DRAM cell is very sensitive to any disturbance
  - A DRAM cell loses its charge over time
    - Need to be refreshed
  - Used in main memory and GPU memory

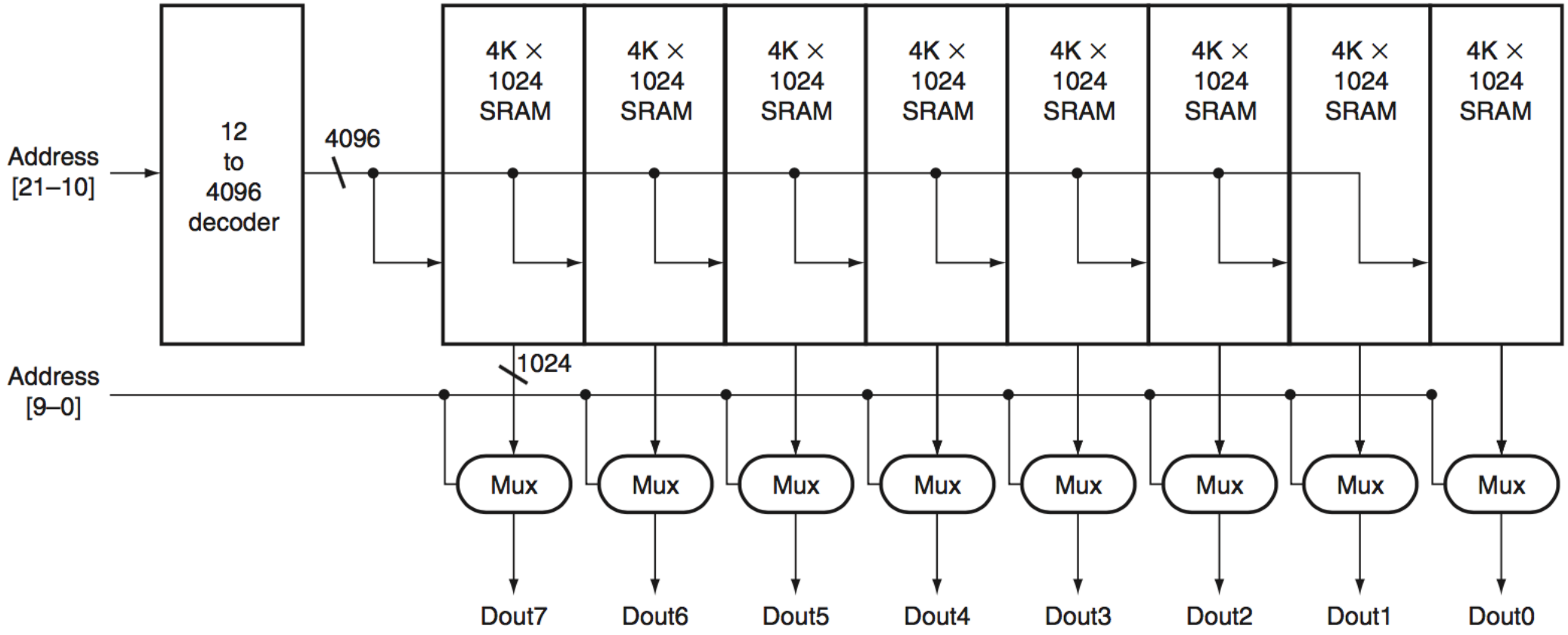


Two-Step Address Decoding

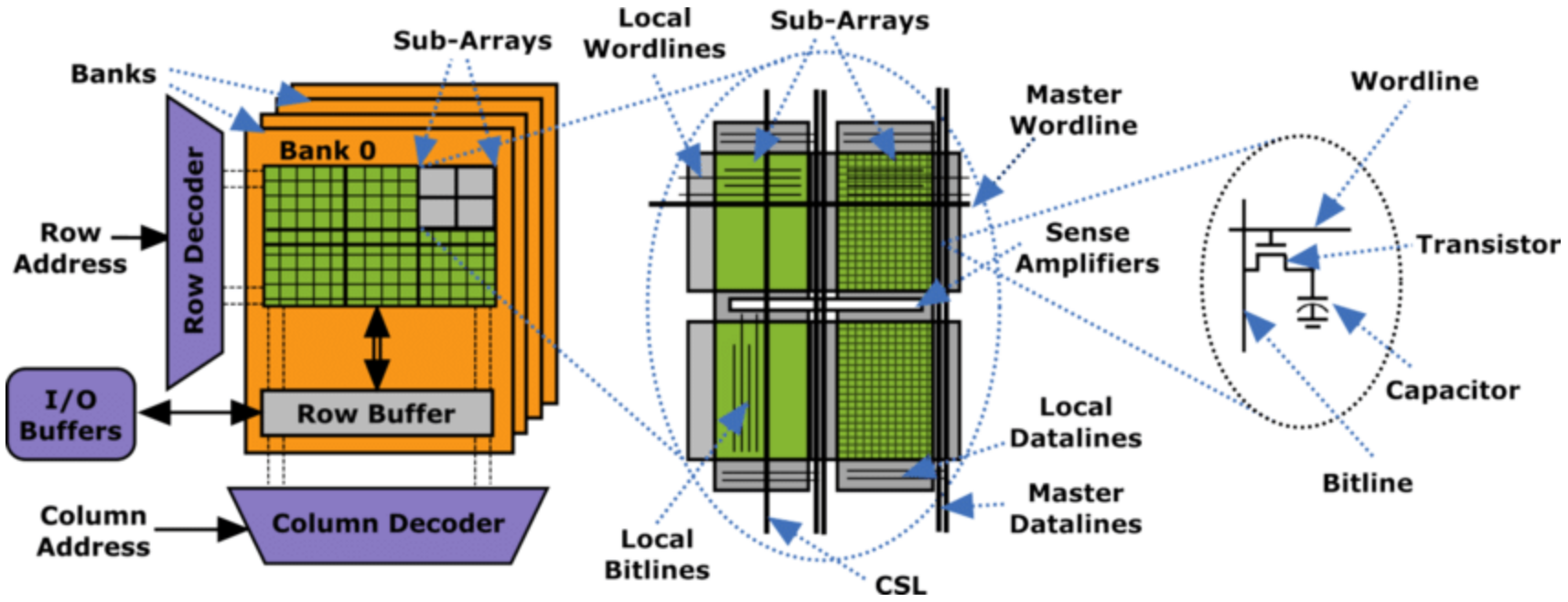
- Unlike a register file, it is impractical to use a single mux/decoder to cope with addresses of large memories
- Large memories with a huge amount of addresses are often organized as rectangular arrays and use a two-step decoding process
  - Higher order bits in an address
  - Lower order bits in an address

SRAM Example

- Organize a 4M X 8 SRAM as an array of 4K X 1024



DRAM Example



Source: Power Modelling of 3D-Stacked Memories with TLM2.0 based Virtual Platforms

Next Lecture

- Section B.10 – B.13
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

3. Multiprocessors & Benchmarking

Zhenkai Zhang

Homework Assignment 1

- Due on Sept. 3rd

Performance Measurement Review

- Execution time = instruction count × CPI × clock cycle time
  - The product of these three factors is the only unimpeachable measure of performance
- Many things can affect these three factors and thus performance
  - Algorithm
    - Affects instruction count and CPI
  - Programming language
    - Affects instruction count and CPI
  - Compiler
    - Affects instruction count and CPI
  - ISA
    - Affects instruction count, CPI, and clock cycle time

MIPS (Million Instructions per Second)

- An alternative for measuring performance is instruction execute rate
  - MIPS (million instructions per second)
    - $\text{MIPS} = \frac{instruction\ count}{CPU\ time \times 10^6} = \frac{clock\ rate}{CPI \times 10^6}$
  - The higher, the better performance
- This metric can be misused and lead to misleading claims
  - MIPS cannot be used to compare computers with different ISAs
    - 10 MIPS on ARM v.s. 8 MIPS on x86 means nothing
  - MIPS varies between programs on the same computer (same as CPU time)
    - Some vendors just brag its product can reach xyz MIPS without giving further info
  - MIPS can vary independently from performance
    - If a new program executes more instructions but each instruction is faster, we may have longer execution time but higher MIPS

MIPS Example

- Consider the following performance measurements for a program:

Measurement	Computer A	Computer B
Instruction Count	10 billion	8 billion
Clock Rate	4 GHz	4 GHz
CPI	1.0	1.1

- Which computer has higher MIPS?
- Which computer is faster?

Benchmarking

- What programs should we use to evaluate performance?
  - A benchmark is a program selected for comparing computer performance
- There are different types of benchmarks
  - Real programs which have input, output, and options that a user can select when running the program
    - Examples: compilers, text processing software, etc.
  - Kernels that are small, key pieces from real programs
    - Examples: Livermore loops and LINPACK
  - Toy benchmarks which are typically between 10 and 100 lines of code and produce a result the user already knows
    - Examples: sieve of Eratosthenes, puzzle, and quicksort
  - Synthetic benchmarks which try to match an average execution profile
    - Examples: Whetstone, Dhrystone, and Rhealstone

Benchmark Suites

- A collection of benchmarks measuring the performance of systems/subsystems with a variety of applications
  - Weakness of any one benchmark is lessened by the presence of the other benchmarks
  - Some benchmarks are kernels, but many are real programs
- Examples
  - SPEC
  - HPL
  - HPC Challenge
  - NASA Parallel Benchmarks

Benchmarking Pitfalls

- Optimization option on today's compilers can affect the results of benchmark tests
- Modification of the sources (public-domain software) produces different versions of the benchmark
- Many benchmarks are one-dimensional in nature (test only one aspect of a system)
  - Different aspects to test are: CPU, I/O, file system, web, etc.
- A compiler can "recognize" a benchmark suite and loads a hand-optimized algorithms for the test

SPEC

- Standard Performance Evaluation Cooperative (SPEC) develops benchmarks for CPU, I/O, web, ...
  - There are several SPEC groups
    - Open Systems Group, High Performance Group, Graphics Performance Groups, and SPEC Research Group
- All SPEC benchmarks are publicly available and well known/understood
  - Benchmarks designed such that external influences are kept at a minimum
  - For CPU, the latest is SPEC CPU2017

SPEC CPU Benchmarks

- There are 43 benchmarks in SPEC CPU2017, organized into 4 suites
  - SPECspeed 2017 Integer has 10 benchmarks
  - SPECspeed 2017 Floating Point has 10 benchmarks
  - SPECrate 2017 Integer has 10 benchmarks
  - SPECrate 2017 Floating Point has 13 benchmarks
    - SPECspeed suites always run one copy of each benchmark
    - SPECrate suites run multiple concurrent copies of each benchmark
      - The tester selects how many
  - Distinct 10 integer benchmarks: 5 written in C, 4 in C++, 1 in Fortran
  - Distinct 14 floating point benchmarks: 6 in Fortran, 5 in C++, 3 in C

SPECspeed 2017 Results (Intel Xeon E5-2650L)

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Perl interpreter	perlbench	2684	0.42	0.556	627	1774	2.83
GNU C compiler	gcc	2322	0.67	0.556	863	3976	4.61
Route planning	mcf	1786	1.22	0.556	1215	4721	3.89
Discrete Event simulation - computer network	omnetpp	1107	0.82	0.556	507	1630	3.21
XML to HTML conversion via XSLT	xalancbmk	1314	0.75	0.556	549	1417	2.58
Video compression	x264	4488	0.32	0.556	813	1763	2.17
Artificial Intelligence: alpha-beta tree search (Chess)	deepsjeng	2216	0.57	0.556	698	1432	2.05
Artificial Intelligence: Monte Carlo tree search (Go)	leela	2236	0.79	0.556	987	1703	1.73
Artificial Intelligence: recursive solution generator (Sudoku)	exchange2	6683	0.46	0.556	1718	2939	1.71
General data compression	xz	8533	1.32	0.556	6290	6182	0.98
Geometric mean	–	–	–	–	–	–	2.36

1774/627

Be Normal

- SPECratio can be considered as a normalized execution time
  - Dividing measured execution times by the reference time
  - SPEC uses a historical Sun Fire V490 with 2100 MHz UltraSPARC-IV+ chips as the reference machine

Example: Program P1 has the following execution times:

On machine A: 10 secs

On machine B: 100 secs

On machine C: 150 secs

Normalized to A: A=1, B=10, C=15

Normalized to B: A=.1, B=1, C=1.5

Execution time ratio

Get Mean

Normalized geometric mean

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

It gives the same relative answer no matter what reference machine is used

Example: the SPECspeed 2017 ratios running on a 1.8 GHz Intel Xeon E5-2650L

2.83, 4.61, 3.89, 3.21, 2.58,
2.17, 2.05, 1.73, 1.71, 0.98

Geometric mean is 2.36

SPEC Power Benchmarks

- Power consumption of servers at different workload levels
  - SPECpower_ssj 2008 benchmark suite with Java business applications
  - 0%, 10%, 20%, …, 100% (10% increments)
  - Performance: ssj_ops (sever side Java operations)
  - Power: watts (joules/sec)

$$Overall\ ssj\_ops\ per\ watt = \frac{\sum_{i=0}^{10} ssj\_ops_i}{\sum_{i=0}^{10} power_i}$$

SPECpower_ssj2008 Results (Intel Xeon X5650)

Target Load %	Performance (ssj_ops)	Average Power (watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1922
∑ssj_ops / ∑power =		2490

SPECpower_ssj2008 Results

- https://www.spec.org/power_ssj2008/results/power_ssj2008.html

Hardware Vendor / Test Sponsor	System / Enclosure (if applicable)	Nodes	JVM Vendor	Processor					Total Memory (GB)	Submeasurements			Result (Overall ssj_ops/watt)
				CPU Description	MHz	Chips	Cores	Total Threads		ssj_ops @ 100%	avg. watts @ 100%	avg. watts @ active idle	
ASUSTeK Computer Inc.	RS720-E9/RS8 Dec 6, 2018 | HTML | Text	1	Oracle Corporation	Intel Xeon Platinum 8180	2500	2	56	112	192	5,386,401	385	48.2	12,727
ASUSTeK Computer Inc.	RS720Q-E9-RS8 None May 10, 2019 | HTML | Text	4	Oracle Corporation	Intel Xeon Platinum 8176M	2100	8	224	448	768	19,257,841	1,417	193	11,935
ASUSTeK Computer Inc.	RS720-E9-RS8 Jul 3, 2019 | HTML | Text	1	Oracle Corporation	Intel Xeon Platinum 8280L	2700	2	56	112	192	5,862,238	412	62.4	14,066
ASUSTeK Computer Inc.	RS720-E9-RS8 Jul 17, 2019 | HTML | Text	1	Oracle Corporation	Intel Xeon Platinum 8280L	2700	2	56	112	192	5,845,032	405	47.4	14,274
ASUSTeK Computer Inc.	RS500A-E10-PS4 Jan 21, 2020 | HTML | Text	1	Oracle	AMD EPYC 7742 2.25Ghz	2250	1	64	128	128	6,064,779	214	51.4	20,908
ASUSTeK Computer Inc.	RS500A-E10-PS4 Jan 21, 2020 | HTML | Text	1	Oracle	AMD EPYC 7742 2.25Ghz	2250	1	64	128	128	5,879,510	206	56.4	21,168
ASUSTeK Computer Inc.	RS700A-E9-RS4V2 Apr 8, 2020 | HTML | Text	1	Oracle	AMD EPYC 7742 2.25Ghz	2250	2	128	256	256	11,702,137	430	106	20,214
ASUSTeK Computer Inc.	RS700A-E9-RS4V2 Apr 8, 2020 | HTML | Text	1	Oracle	AMD EPYC 7742 2.25Ghz	2250	2	128	256	256	11,945,201	425	92.9	20,350
ASUSTeK Computer Inc.	RS620SA-E10-RS12 RS620SA-E10 Oct 7, 2020 | HTML | Text	6	Oracle	AMD EPYC 7742 2.25Ghz	2250	6	384	768	768	36,876,127	1,140	228	24,238
ASUSTeK Computer Inc.	RS620SA-E10-RS12 RS620SA-E10 Oct 7, 2020 | HTML | Text	6	Oracle	AMD EPYC 7742 2.25Ghz	2250	6	384	768	768	36,645,962	1,141	250	24,507
ASUSTeK Computer Inc. ASUSTeK Company Inc.	ASUS RS160-E5 (Intel Xeon L5430 Processor, 2.66 GHz) Nov 20, 2008 | HTML | Text	1	Oracle Corporation	Intel Xeon L5430	2666	2	8	8	8	278,927	173	89.4	1,020
ASUSTeK Computer Inc. ASUSTeK Company Inc.	ASUS RS160-E5 (Intel Xeon L5420 Processor, 2.50 GHz) Nov 20, 2008 | HTML | Text	1	Oracle Corporation	Intel Xeon L5420	2500	2	8	8	8	270,621	170	86	1,009
ASUSTeK Computer Inc. ASUSTeK Company Inc.	ASUS RS100-E5 (Intel Xeon X3360 Processor, 2.83 GHz) Jan 2, 2009 | HTML | Text	1	Oracle Corporation	Intel Xeon X3360 Processor	2833	1	4	4	4	165,064	118	56.7	905

No Power at Idle?

- Look back at the results running on Intel Intel Xeon X5650
  - At 100% load: 258W
  - At 50% load: 170W (66%)
  - At 10% load: 121W (47%)
- Google data center
  - Mostly operates at 10% – 50% load
  - At 100% load less than 1% of the time
- Consider designing processors to make power proportional to load

Multiprocessors

- Recall that the “power wall” has limited the clock rate
  - To improve performance, we need to take advantage of multiprocessors
- Multicore processors
  - High end server processors (quad-core, six-core, 10-core processors)
  - Atom processors in netbooks (dual-core, quad-core, six-core)
  - Processors in cell phones (dual-core, quad-core, six-core)
- Requires explicitly parallel programming
  - Programming for performance
  - Load balancing
  - Optimizing communication and synchronization

Can a good parallel program improve the performance by 4 times on a quad-core?

Amdahl's Law

- We cannot improve an aspect of a computer and expect a proportional improvement in overall performance

$$T_{new} = \frac{T_x}{\text{factor}} + T_y \quad \text{where}$$

T_x is the time affected by the enhancement, and

T_y is the time spent using unenhanced portion of the computer

- Amdahl's law gives us a quick way to find the speedup from some enhancement

$$\text{speedup} = \frac{1}{(1 - \text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{\text{factor}}}$$

Amdahl's Law Example

Suppose that we want to enhance the processor used for web serving. The new processor is *10 times faster* on computation in the web serving application than the original processor. Assuming that the original processor is busy with computation *40% of the time* and is waiting for I/O *60% of the time*, what is the overall speedup gained by incorporating the enhancement?

$$\text{fraction}_{\text{enhanced}} = 0.4 \quad \text{factor} = 10 \quad \text{speedup} = \frac{1}{0.6+0.04} = 1.56$$

Another Example

Suppose a program runs in *100 seconds* on a computer, with multiply operations responsible for *80 seconds* of this time. How much do I have to improve the speed of multiplication if I want my program to run *5 times faster*?

$$\text{fraction}_{\text{enhanced}} = 0.8 \quad \text{speedup} = 5 = \frac{1}{0.2 + \frac{0.8}{\text{factor}}}$$

No way!

Summary

- Cost/performance is improving
  - Moore's law (slowed down), Dennard scaling (end)
  - Use parallelism to improve performance
- Hierarchical layers of abstraction
  - In both hardware and software
- Instruction set architecture
  - The hardware/software interface
- Execution time: the best performance measure
- Power is a limiting factor

Next Lecture

- Section B.1 – B.4
  - Read the contents
  - Finish pre-class questions

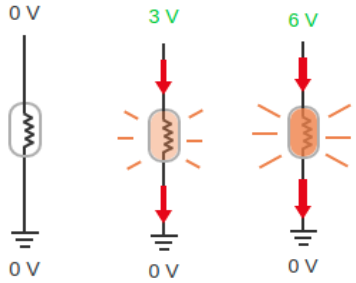
CPSC 3300-001
Computer Systems Organization

4. Combinational Logic

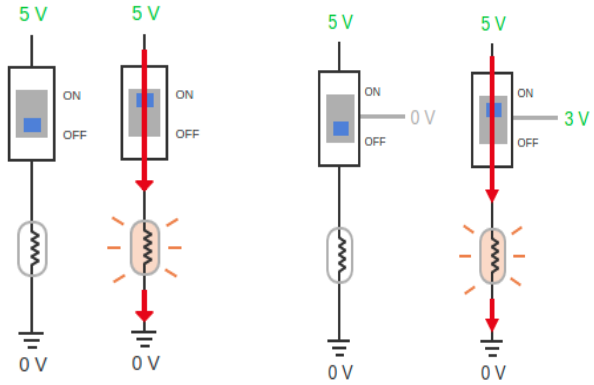
Zhenkai Zhang

Electrical Systems

- An electrical system involves movement of charged electrons through wires
  - Voltage is the potential for charge to move
  - Current is the amount of charge flow
  - Resistance is a wire's opposition to flow

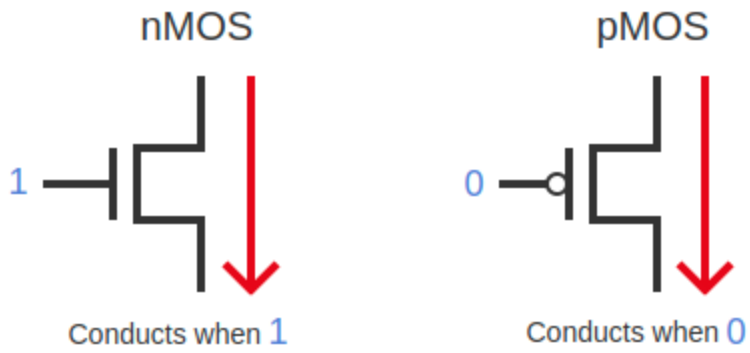


- A switch is an electronic device that conducts between two terminals if the switch is on
- An electronically-controlled switch has another input terminal whose voltage can turn the switch on
  - Relays, vacuum tubes, and transistors



Digital Circuits

- A digital circuit has voltages that are treated as either high or low, and is typically built as a connection of transistors
  - High/low voltage, 1/0, true/false, asserted/deasserted



- Building complex circuits from transistors is hard
  - In 1938, Claude Shannon described how transistor circuits could implement logic functions

How to Describe a Logic Function

- We can use a truth table to precisely specify a logic function
- A truth table lists all possible combinations of input values on the left, and lists the output value for each combination on the right

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

  - If there are n inputs, how many entries in the truth table?
    - 2^n
    - Grows exponentially (not good)
- Another approach to specifying a logic function is to use Boolean equations (also called logic equations)
  - This is done with the use of Boolean algebra
    - Shannon applied Boolean algebra to the design of digital circuits
    - Digital circuits are sometimes called logic circuits due to the roots in Boolean algebra

Boolean Algebra

- Boolean algebra is an algebra whose only values are true or false, and whose operators are AND, OR, and NOT
  - **NOT** gives 1 if the input is 0, and gives 0 if the input is 1
  - **AND** outputs 1 only if both the inputs are 1's
    - Logic product
  - **OR** outputs 1 if either, or both, of the inputs is a 1
    - Logic sum

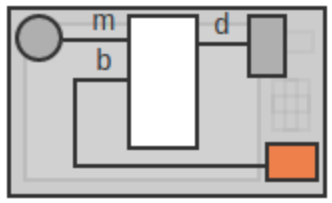
Operation	Shorthand	Notes
a AND b	ab	Intentionally looks like multiplication. Known as *abutment*.
a OR b	a + b	Intentionally looks like addition.
NOT(a)	a'	a' is also called the ***complement*** of a.

Basic Properties of Boolean Algebra

Property	Name	Description
$a(b + c) = ab + ac$ $a + (bc) = (a + b)(a + c)$	Distributive (AND) Distributive (OR)	(AND) Same as multiplication in regular algebra (OR) Different from regular algebra
$ab = ba$ $a + b = b + a$	Commutative	Variable order does not matter. Good practice is to sort variables alphabetically.
$(ab)c = a(bc)$ $(a + b) + c = a + (b + c)$	Associative	Same as regular algebra
$aa' = 0$ $a + a' = 1$	Complement (AND) Complement (OR)	(AND) Clearly one of a, a' must be 0 $1 \cdot 0 = 0 \cdot 1 = 0$ (OR) Clearly one of a, a' must be 1 $1 + 0 = 0 + 1 = 1$
$a \cdot 1 = a$ $a + 0 = a$	Identity (AND) Identity (OR)	(AND) Result of $a \cdot 1$ is always a's value $0 \cdot 1 = 0$ $1 \cdot 1 = 1$ (OR) Result of $a + 0$ is always a's value $0 + 0 = 0$ $1 + 0 = 1$
$a \cdot 0 = 0$ $a + 1 = 1$	Null elements	Result doesn't depend on the value of a.
$a \cdot a = a$ $a + a = a$	Idempotent	Duplicate values can be removed.
$(a')' = a$	Involution	$(0')' = (1)' = 0$ $(1')' = (0)' = 1$
$(ab)' = a' + b'$ $(a + b)' = a'b'$	DeMorgan's Law	

Boolean Equation

- A Boolean equation has a Boolean variable (left), an equal sign, and a Boolean expression (right), defining the left variable's value based on the right variables' values
  - A Boolean equation can describe a digital circuit, with the output on the left and the inputs on the right



Inputs: m: motor is operating
b: button is pressed

Output: d: open door

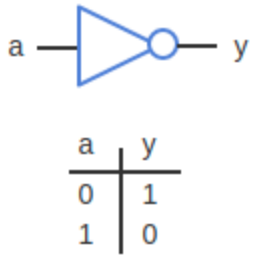
Goal: Open door if user presses door open button, but only if motor is not operating

d = b AND NOT(m)

d = bm’

Logic Gates

- A logic gate is a transistor circuit that implements a basic logic function



a	y
0	1
1	0

NOT
(invertor)

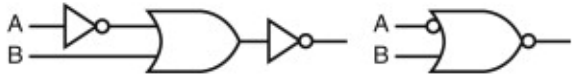
a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

AND

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

OR

Any logical function can be constructed using AND gates, OR gates, and invertors



Invertors are often drawn as bubbles at gate inputs/outputs

a	b	f
0	0	0
0	1	1
1	0	1
1	1	0

XOR
(y = ab' + a'b)

a	b	f
0	0	1
0	1	0
1	0	0
1	1	1

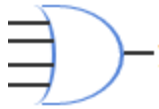
XNOR

a	b	f
0	0	1
0	1	1
1	0	1
1	1	0

NAND

a	b	f
0	0	1
0	1	0
1	0	0
1	1	0

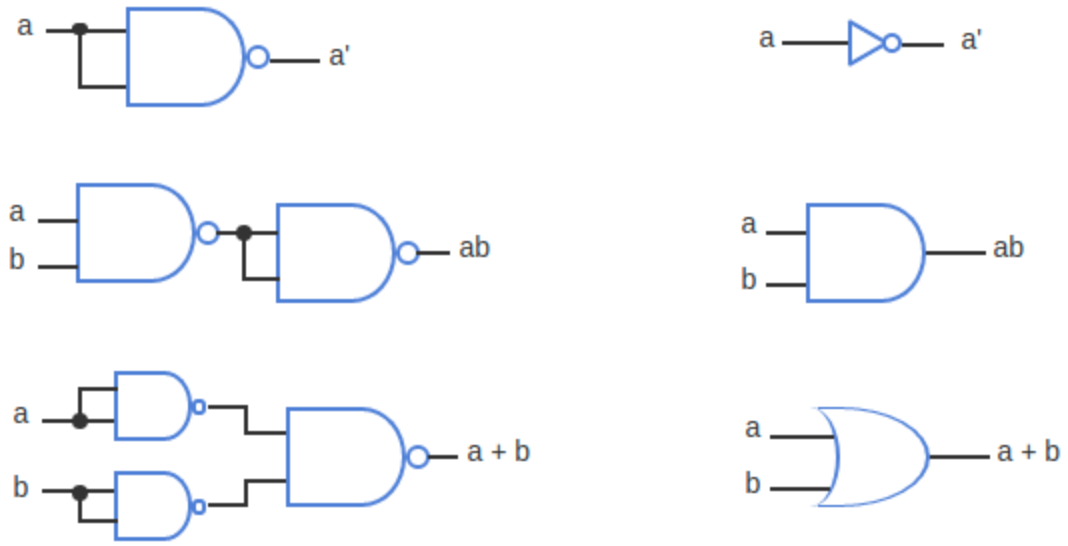
NOR



Except for NOT, other gates may take more than two inputs

Universal Gates

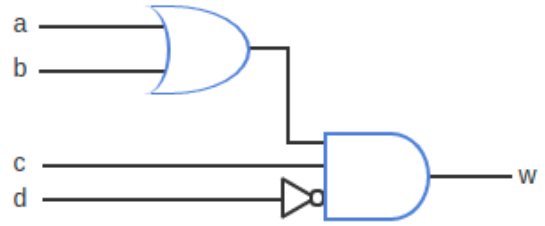
- A universal gate is a single gate type that can implement any combinational circuit
  - Both NAND and NOR gates are universal gates



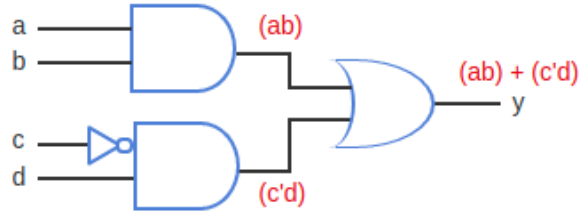
Equations to/from Circuits

- A Boolean equation can be converted to a digital circuit by converting each operation to a gate
  - Conversion is done first for items within parentheses
  - In a term like cd', NOT is converted before AND or OR

w = (a+b)cd’



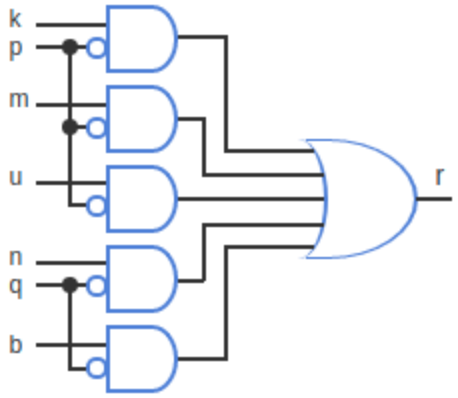
- A circuit can be converted to an equation
  - Starting from the inputs, the process replaces gates by terms while moving towards the output, labeling gate outputs along the way



y = (ab)+(c’d)

Sum-of-Products (SOP)

- Circuits are commonly designed by creating a simplified expression in SOP form, then converting to a two-level circuit
  - A product term (also called product or term) is an ANDing of (one or more) variables, like a, b', or ab'c
  - An expression in SOP form consists of an ORing of product terms, like ab'c + ab



r = kp' + mp' + up' + nq' + bq'

Sum-of-Minterms

- A canonical form of a Boolean equation is needed
  - Different equations may represent the same function
    - Ex: y = a + b, and y = a + a'b, represent the same function
    - The sameness is not obvious, so a standard equation form is desirable
- Sum-of-minterms form is a canonical form, which is a sum-of-products with each product a unique minterm
  - A minterm is a product term having exactly one literal for every function variable
  - A literal is a variable appearance, in true or complemented form, in an expression, such as b, or b'

A logic function y has inputs a and b

Is y = ab + a'b + a'b' + ab in sum-of-minterms form?

How about y = ab + a‘?

Transforming to Sum-of-Minterms

- A sum-of-products equation can be transformed to sum-of-minterms
  - Multiplying each product term by (v + v') for any missing variable v
  - Removing redundant minterms

y = ab + abc'
= ab(c+c') + abc'
= abc + abc' + abc'
= abc + abc'

- A truth table can be transformed to a sum-of-minterms equation
  - Summing the minterms in rows having a 1

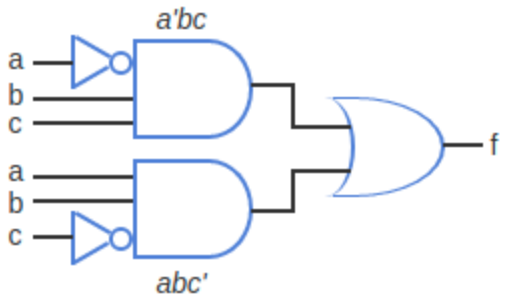
Truth table

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Equation

f = a'bc + abc'

Circuit



Example

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

D = A + B + C

D = A’B’C + A’BC’ + A’BC + AB’C’ + AB’C + ABC’ + ABC

E = A’BC + AB’C + ABC’

F = ABC

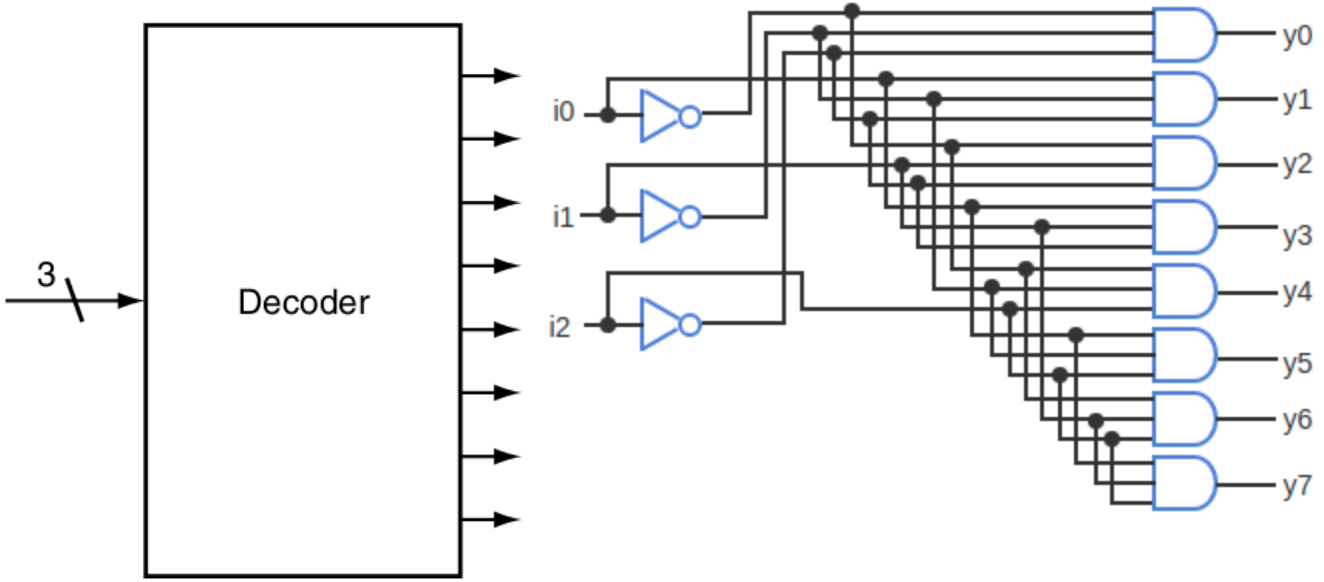
To simplify a sum-of-minterms, we can use Karnaugh maps (not covered)

Combinational v.s. Sequential Logic

- A combinational logic circuit's output depends only on the present combination of input values
  - It corresponds to a logic function (which can be specified by a truth table or a Boolean equation)
- A sequential logic circuit's output depends on the present and the past sequence of input values
  - It means the circuit has memory and stores at least one bit

Decoder

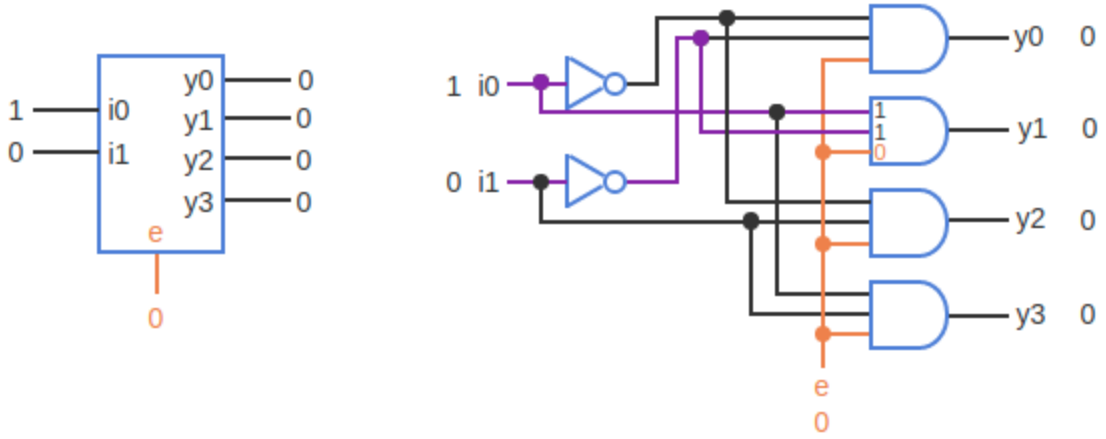
- A decoder is a combinational logic circuit that converts N inputs to a 1 on one of 2^N outputs
  - A 3-bit decoder converts three inputs to a 1 on exactly one of eight outputs



Outputs				
Out4	Out3	Out2	Out1	Out0
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

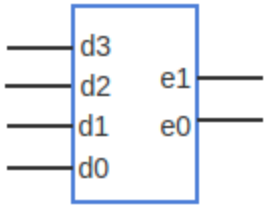
Decoder with Enable

- Some decoders have an additional input called an enable input
  - Enable = 0 sets all outputs to 0s
  - Enable = 1 enables the decoder for normal behavior



Encoder

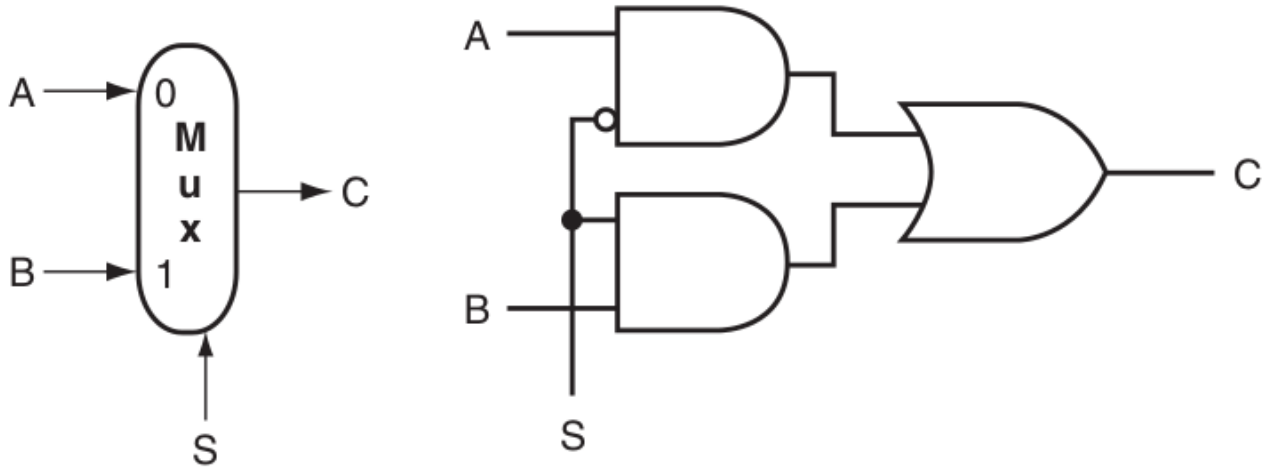
- An encoder is a combinational logic circuit that converts 1 of N inputs to a binary value using $\lceil \log_2 N \rceil$ outputs
  - An encoder has the reverse functionality of a decoder
  - An encoder assumes that exactly one input will be 1



d3	d2	d1	d0	e1	e0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

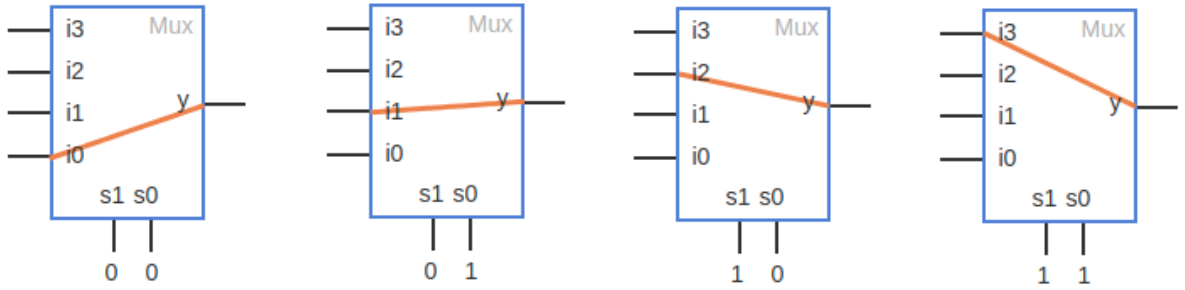
Multiplexor (Mux)

- A multiplexor (mux) is a combinational logic circuit that passes one of multiple data inputs through to a single output, selecting which one based on additional control inputs
  - A mux's control inputs are called select lines
    - If there are N data inputs, how many control inputs are needed?
      - $\lceil \log_2 N \rceil$

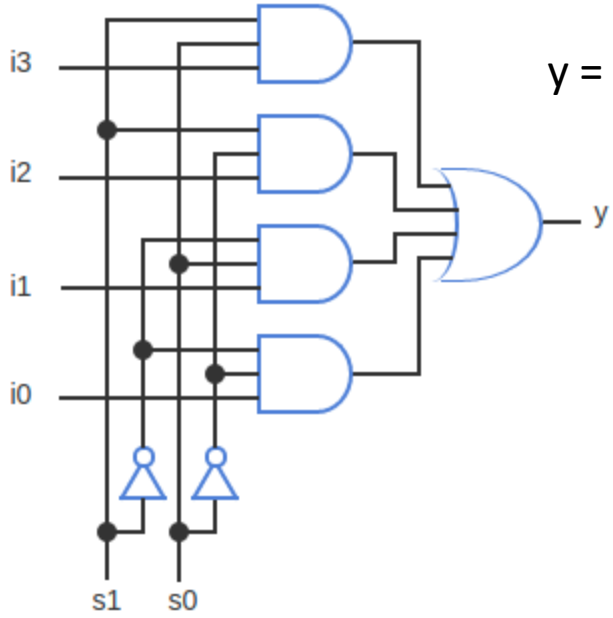


4-1 Mux

- A 4-1 mux, spoken as "4 to 1 mux", has 4 data inputs, 1 data output, and requires 2 control inputs



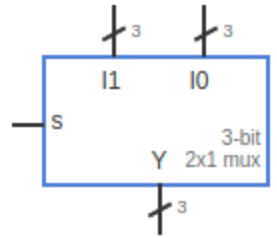
s1	s0	y
0	0	i0
0	1	i1
1	0	i2
1	1	i3

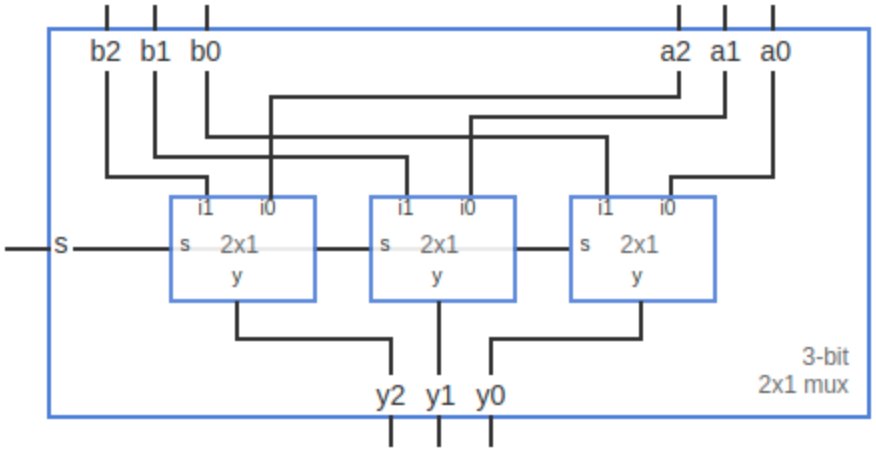


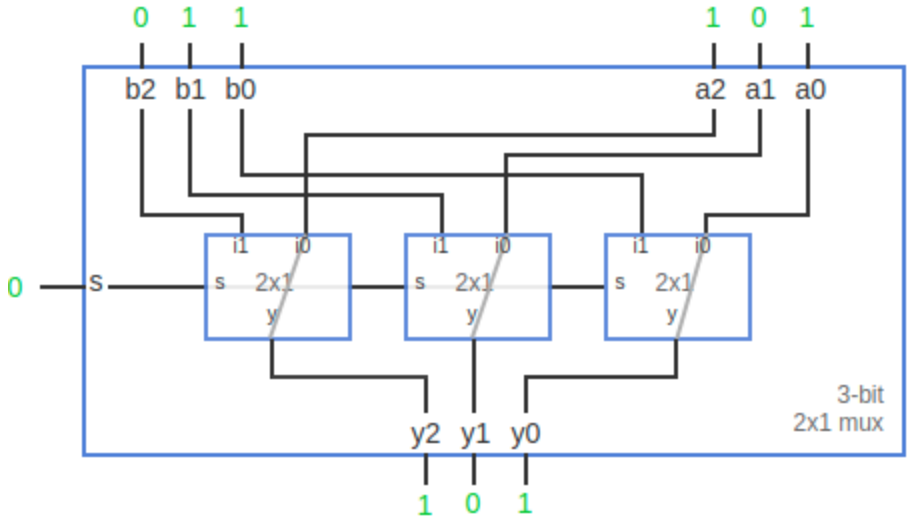
y = s1's0'i0 + s1's0i1 + s1s0'i2 + s1s0i3

N-bit MUX

- A N-bit mux is a mux where you select N bits at a time (as a group), instead of 1 bit
  - For example, you might want to select either a2a1a0 or b2b1b0
    - Data inputs: 6 (i.e., a0, a1, a2, b0, b1, b2) – there are 2 groups and each group has 3 bits
    - Control inputs: 1 (i.e., s) – we are selecting only one of the 2 groups
    - Outputs: 3 (i.e., y0, y1, y2) – each group has 3 bits
      - Constructing a 3-bit 2-1 MUX from single bit 2-1 muxes

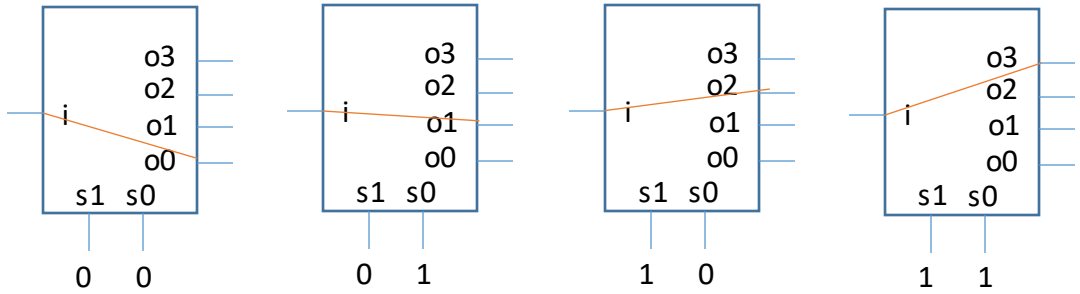






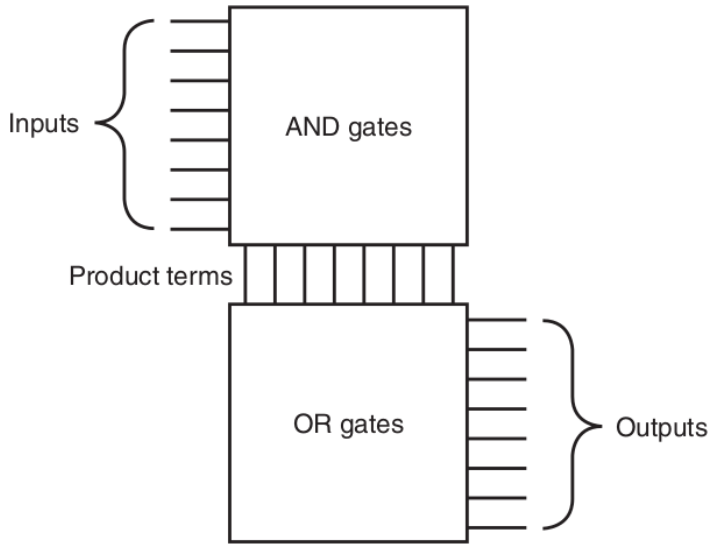
Demultiplexer (Demux)

- A demultiplexer (or demux) is a combinational logic circuit that passes a single input to one of multiple data outputs, selecting which one based on additional control inputs
  - A demux has the reverse functionality of a mux
  - 1 data input
  - N data outpus
  - $\lceil \log_2 N \rceil$ control inputs



Programmable Logic Array (PLA)

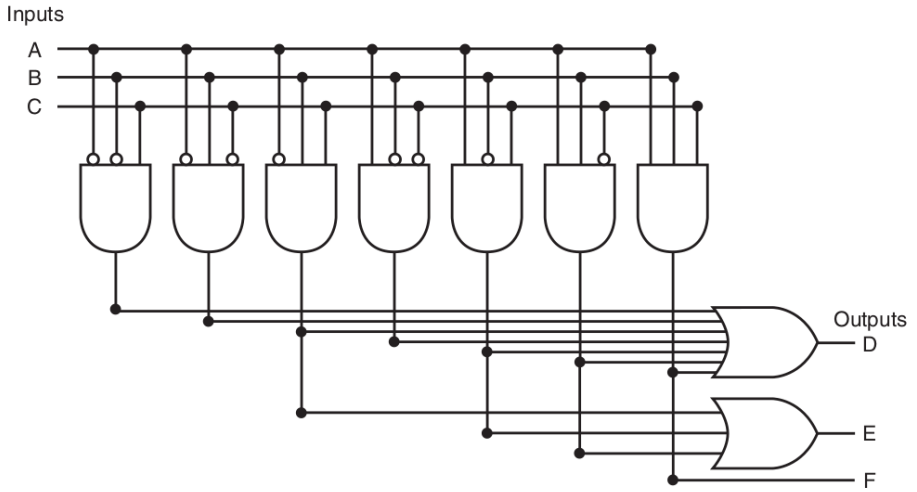
- A PLA has a set of inputs and corresponding input complements, and two stages of logic
  - The first stage is an array of AND gates that form a set of minterms
  - The second stage is an array of OR gates, each of which forms a sum of minterms



D = A'B'C + A'BC' + A'BC + AB'C' + AB'C + ABC' + ABC

E = A'BC + AB'C + ABC'

F = ABC



Read-Only Memory (ROM)

- A ROM can encode a collection of logic functions directly from the truth table
  - m address lines → 2^m ROM entries
  - Each entry has n bits → n logic functions

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

111	101
110	011
101	011
100	001
011	011
010	001
001	001
000	000

Next Lecture

- Section B.5 – B.6
  - Read the contents
  - Finish pre-class questions

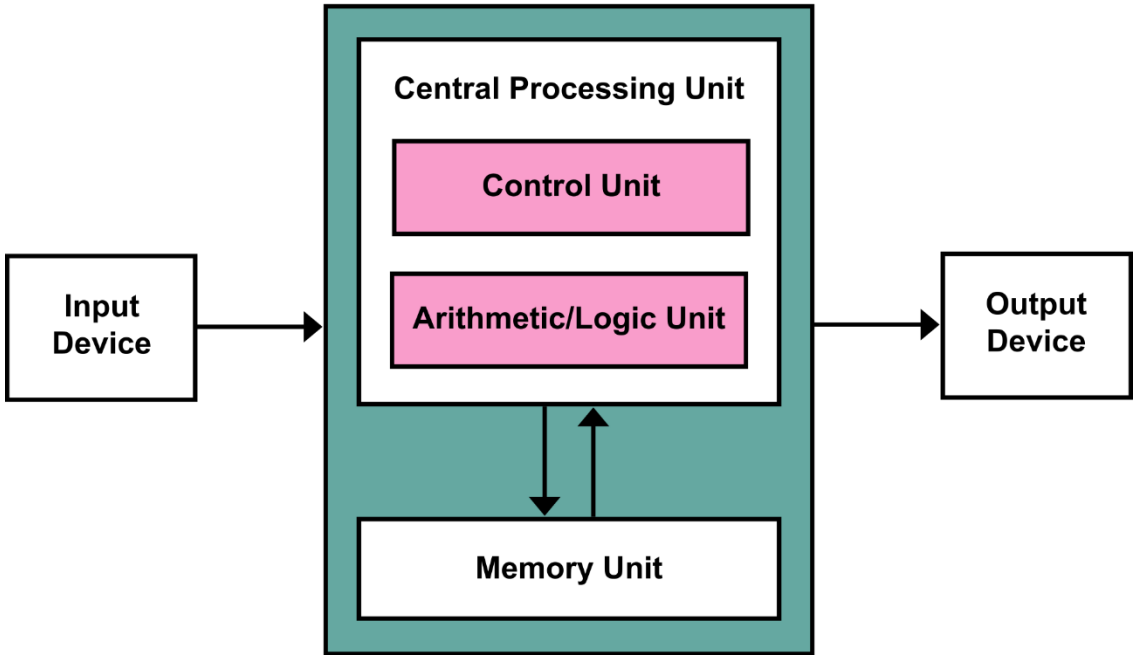
CPSC 3300-001
Computer Systems Organization

2. Performance & Power Wall

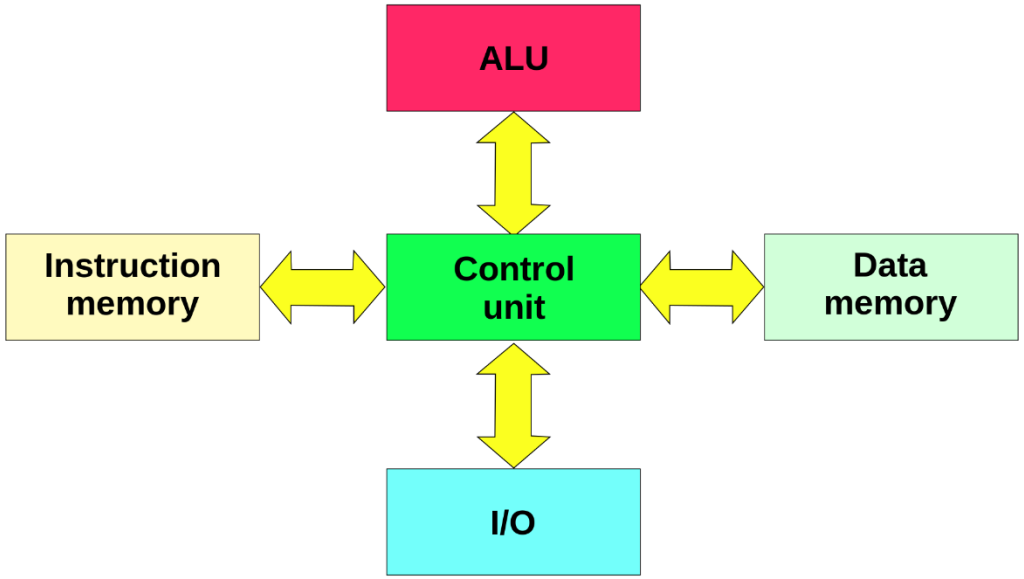
Zhenkai Zhang

Stored-Program Computer

- A stored-program computer stores instructions and data together in memory as numbers
  - Allows the computer to be "reprogrammed"



von Neumann Architecture

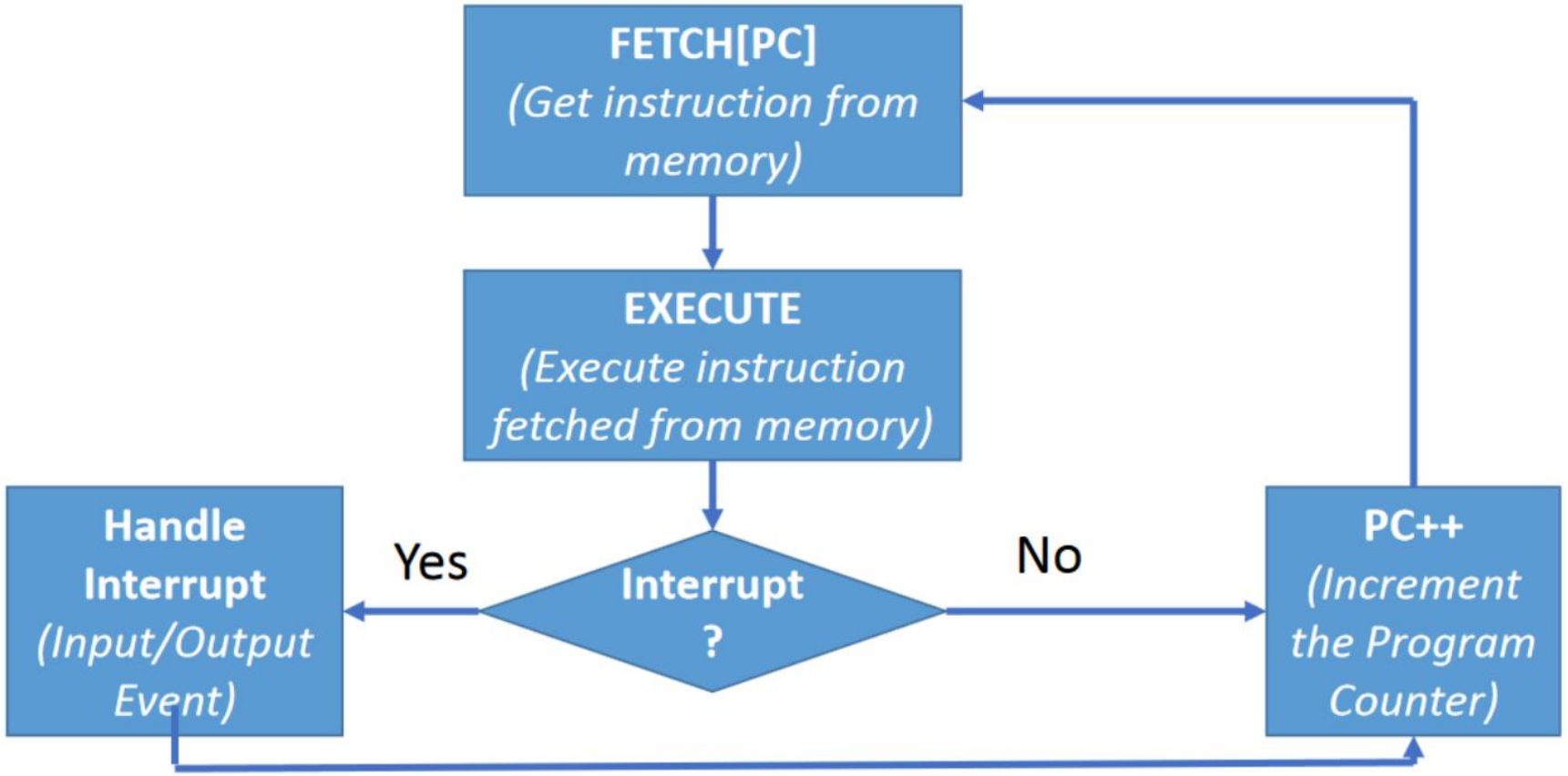


Harvard Architecture

Source: https://en.wikipedia.org/wiki/Von_Neumann_architecture

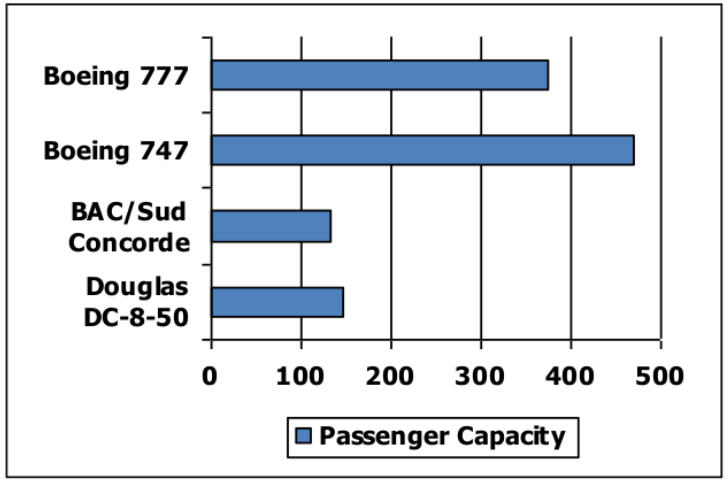
https://en.wikipedia.org/wiki/Harvard_architecture

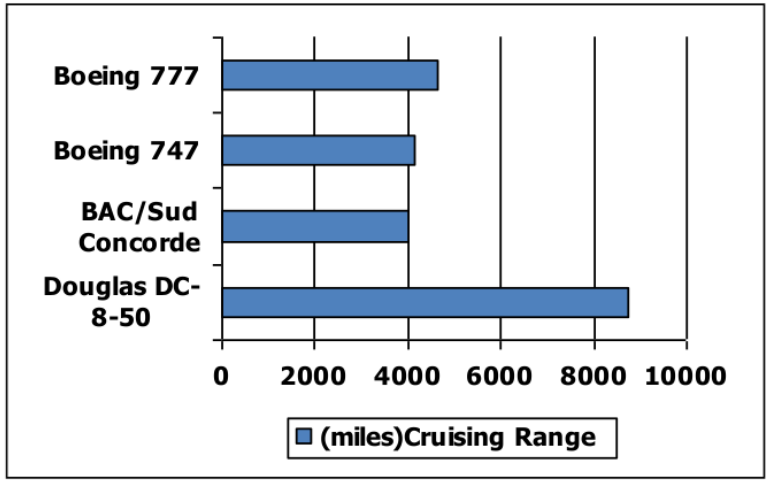
Processor Execution Cycle

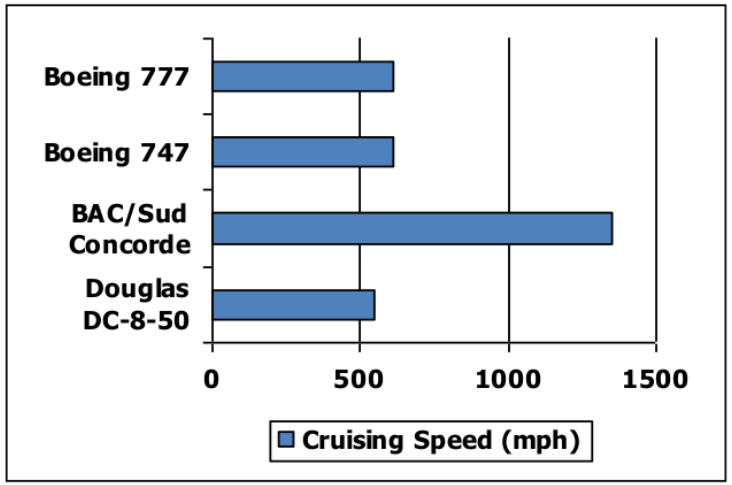


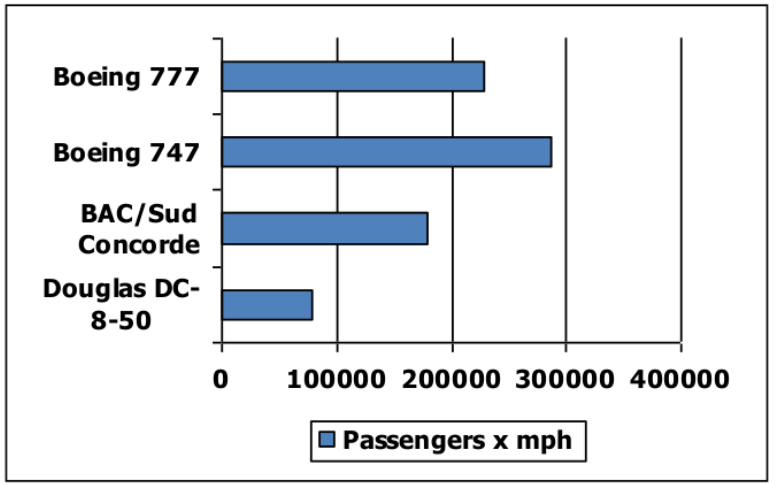
Defining Performance

- Which aircraft has the best performance?









Metrics: Execution Time

- Define:

$$Performance = \frac{1}{Execution\ time}$$

- "X is n times faster than Y“

$$\frac{Performance_X}{Performance_Y} = \frac{Execution\ time_Y}{Execution\ time_X} = n$$

Relative Performance

If *computer A* runs a program in *10 seconds* and *computer B* runs the same program in *15 seconds*, how much *faster* is A than B?

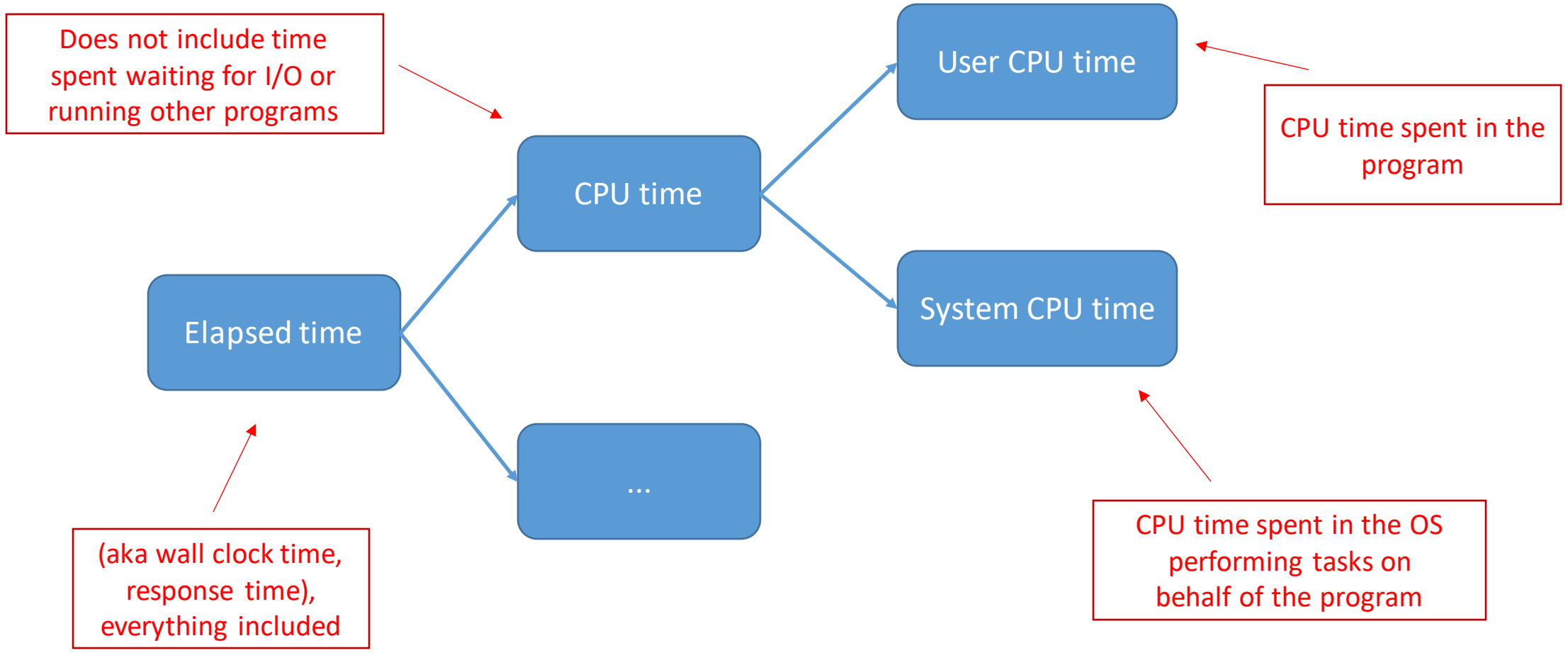
$$\frac{Performance_A}{Performance_B} = \frac{Execution\ time_B}{Execution\ time_A} = 1.5$$

So, A is 1.5 times faster than B

Measuring Execution Time

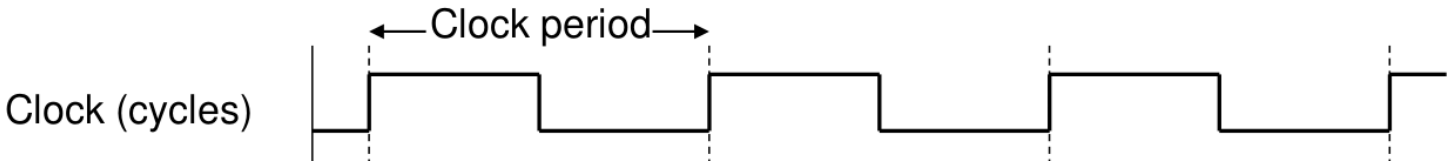
- Elapsed time
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- CPU time
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares
  - Comprises user CPU time and system CPU time

Measuring Execution Time



CPU Clocking

- Operation of digital hardware governed by a clock



- Clock period (= clock cycle) – duration of a clock cycle
  - e.g., 250 ps = 0.25 ns = 250×10^{-12} seconds
- Clock rate (frequency) – cycles per second
  - e.g., 4.0 GHz = 4000 MHz = 4×10^{9} Hz

CPU Time

Total CPU execution time for a program (in sec.) = the number of CPU clock cycles × the duration of 1 clock cycle

= the number of CPU clock cycles ÷ the clock rate

- Performance improved by:
  - Reducing the number of clock cycles
  - Increasing the clock rate
  - Hardware designer must often trade off clock rate against cycle count

CPU Time Example

Our favorite program runs in *10 seconds* on *computer A*, which has a *2 GHz clock*. We are trying to help a computer designer build *computer B*, which will run this program in *6 seconds*. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing *computer B* to require *1.2 times* as many clock cycles as *computer A* for this program. What clock rate should we tell the designer to target?

$$\frac{\#cycles\ on\ B}{\#cycles\ on\ A} = 1.2 \qquad 10 = \frac{\#cycles\ on\ A}{2\ GHz} \qquad 6 = \frac{\#cycles\ on\ B}{?}$$

Classic Performance Equation

- Let us include the number of instructions needed for the program

CPU clock cycles = the number of instructions × average clock cycles per instruction

- The number of instructions is often called instruction count
  - Given a binary, it is fixed
- Average clock cycles per instruction is called CPI
  - CPI provides a way to compare two different implementations of an ISA
    - AMD Ryzen 9 5900X v.s. Intel Core i9-11900K
- Therefore:

$$\text{CPU time} = \text{instruction count} \times \text{CPI} \times \text{clock cycle time}$$

Performance Equation Example

Suppose we have two implementations of *the same instruction set architecture*. *Computer A* has *a clock cycle time of 250 ps* and *a CPI of 2.0* for some program, and *computer B* has *a clock cycle time of 500 ps* and *a CPI of 1.2* for the same program. Which computer is faster for this program and by how much?

$$CPU\ time_A = IC_A \times 2.0 \times 250\ ps$$
$$CPU\ time_B = IC_B \times 1.2 \times 500\ ps$$

$IC_A = IC_B$ Why?

$$\frac{Performance_A}{Performance_B} = \frac{CPU\ time_B}{CPU\ time_A} = 1.2$$

CPI in More Detail

- Different instruction classes may take different numbers of cycles
  - Integer add may take 1 but floating-point div may take tens

$$\#clock\ cycles = \sum_{i=1}^{n} (CPI_i \times instruction\ count_i)$$

$$CPI = \frac{\#clock\ cycles}{\sum_{i=1}^{n} instruction\ count_i}$$

- CPI varies by instruction mix
  - A measure of the dynamic frequency of instructions across one or many programs

Example

Suppose a program 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?

$$CPI = 0.5 \times 3 + 0.3 \times 4 + 0.2 \times 5 = 3.7$$

$$CPU\ time = 10^9 \times 3.7 \times \frac{1}{2 \times 10^9} = 1.85\ s$$

Example (Cont.)

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*. No changes are made to the instruction set. What is the overall percentage improvement?

$$CPI_{new} = 0.5 \times 3 + 0.3 \times 4 + 0.2 \times 4 = 3.5$$

$$CPU\ time_{new} = 10^9 \times 3.5 \times \frac{1}{1.9 \times 10^9} = 1.84\,s$$

$$\frac{Performance_{new}}{Performance_{old}} = \frac{CPU\ time_{old}}{CPU\ time_{new}} = 1.005$$

Performance Equation Discussion

$$\text{CPU time} = \text{instruction count} \times \text{CPI} \times \text{clock cycle time}$$

- Instruction count for a program
  - Determined by program itself (algorithm and language), ISA, and compiler
- CPI
  - Determined by CPU hardware and also program itself (algorithm and language) and compiler
- Clock cycle time
  - Determined by CPU hardware

Power Consumption

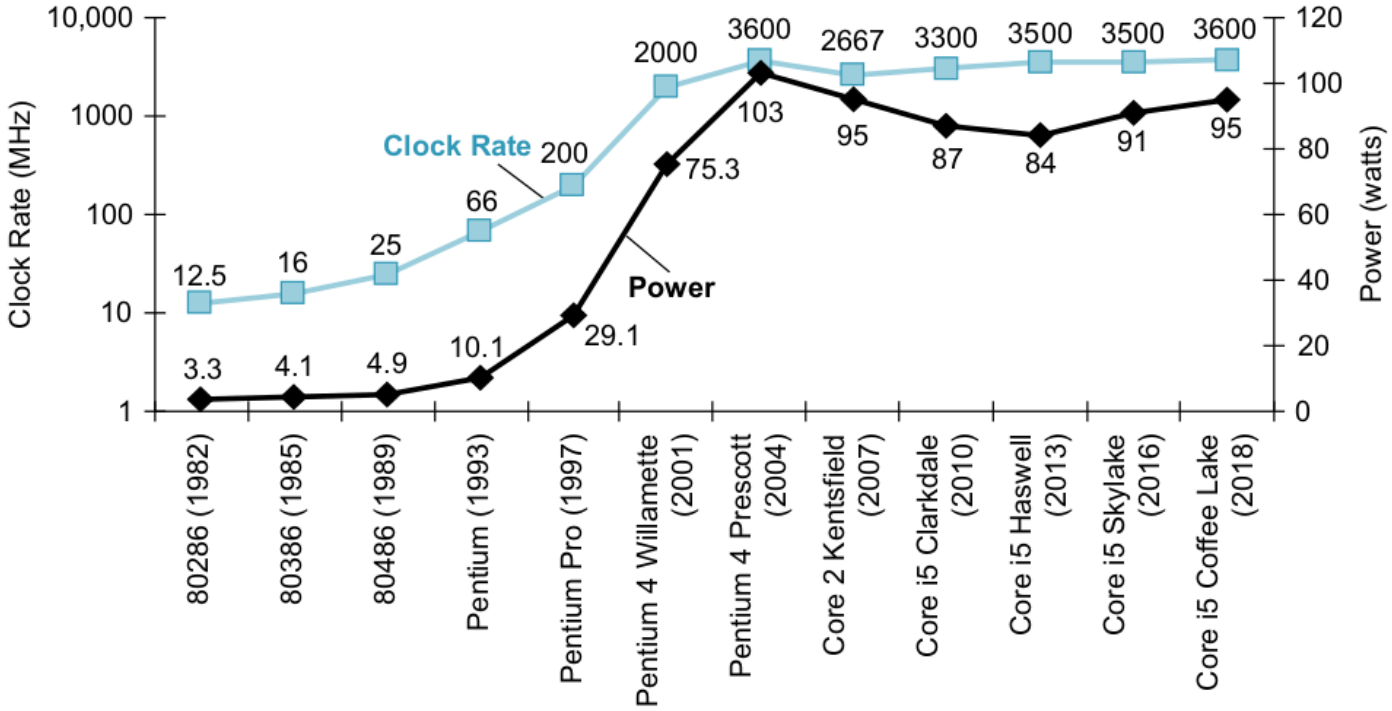
- Power consumption consists of dynamic power and static power
  - Dynamic power is caused by the charging and discharging of the capacitive load on each gate's output due to switching activity
    - $P_d \propto C \times V^2 \times f$
  - Static power captures the power lost from the leakage current irrespective of switching activity
    - $P_s \propto V \times I \times N$
- Dynamic power is the main component
  - DVFS (dynamic voltage and frequency scaling)
  - Clock gating

Dennard Scaling

- Robert Dennard in 1974 observed that voltage and current should be proportional to the linear dimensions of a transistor
  - Smaller transistors just need smaller voltage and current to work
- Operational voltage keeps being shrunk during the last two decades
  - From 5 V to 1 V
  - Circuits could operate at higher frequencies using the same power
  - But can we keep increasing the clock frequency?

The End of Dennard Scaling

- Dennard scaling did not consider leakage power and threshold voltage
  - Small voltage cannot shut off transistors completely
  - The aggregate leakage power can create big thermal problems
  - A “power wall” is created to make 4 GHz clock rate kind of the ceiling



Next Lecture

- Section 1.8 – 1.12
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

1. Introduction

Zhenkai Zhang

Computer Systems are Ubiquitous

Why Study Computer Organization?

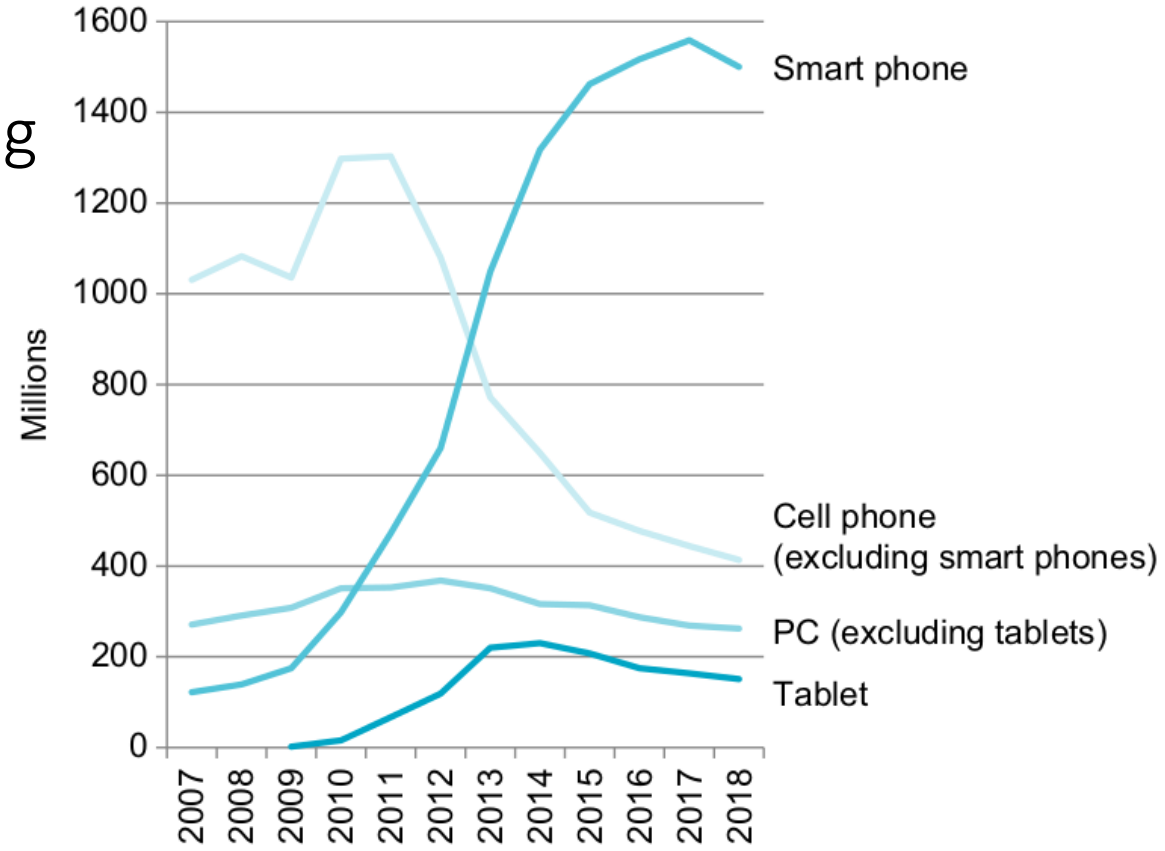
- You are in CS major (and you are forced to learn)
- You use them everyday (so you want to know the internals)
- You want to DIY a computer (so you know what to buy)
- You aim to be a world-class programmer (for better pay)
- You need to solve tricky issues related to performance/energy
- You don’t want to be embarrassed during interviews
- …

Traditional Classes of Computer Systems

- Personal computers (PC)
  - A computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse
- Servers
  - A computer used for running larger programs for multiple users, often simultaneously, and typically accessed only via a network
- Embedded computers
  - A computer inside another device used for running one predetermined application or collection of software

Post-PC Era

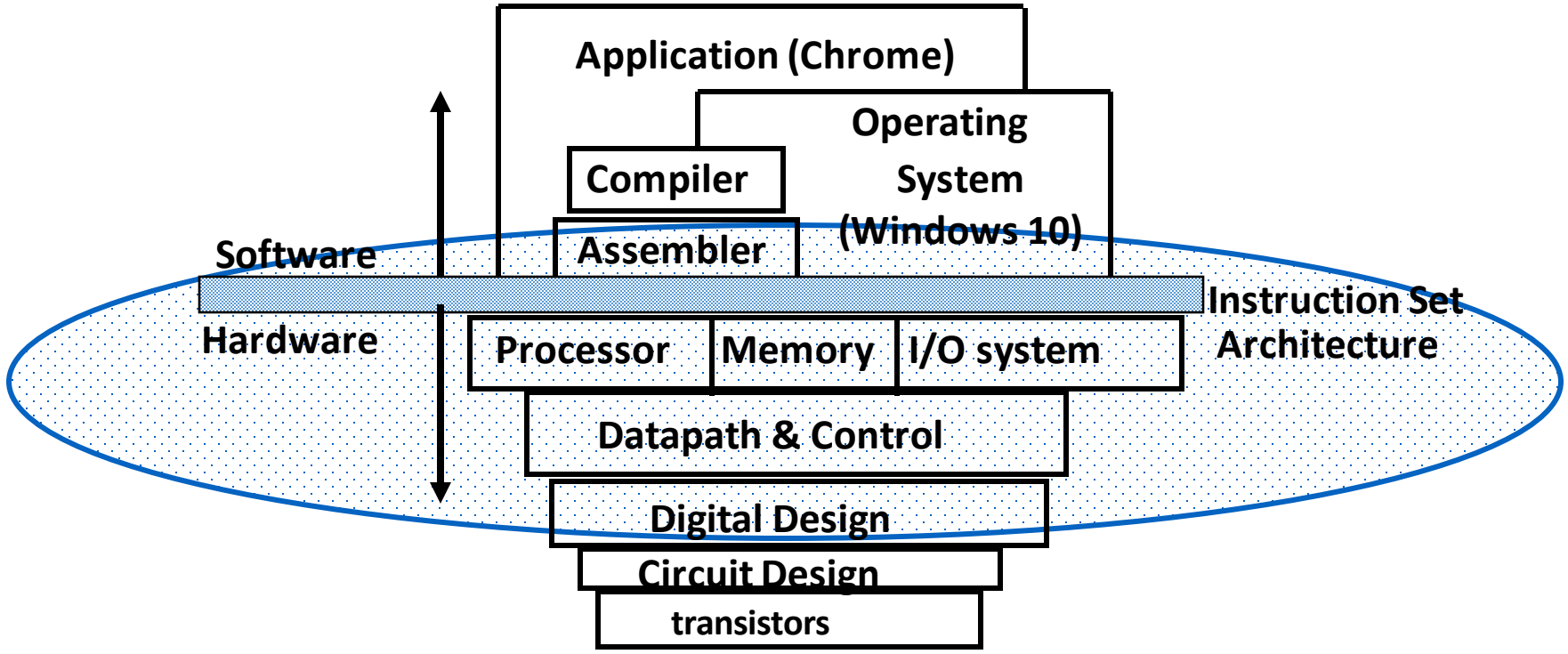
- PC → personal mobile devices
  - Tablets, smart phones…
- Traditional server → cloud computing
  - Infrastructure as a service (IaaS)
    - Virtual machines…
  - Platform as a service (PaaS)
    - Docker containers…
  - Software as a service (SaaS)
    - Google drives…



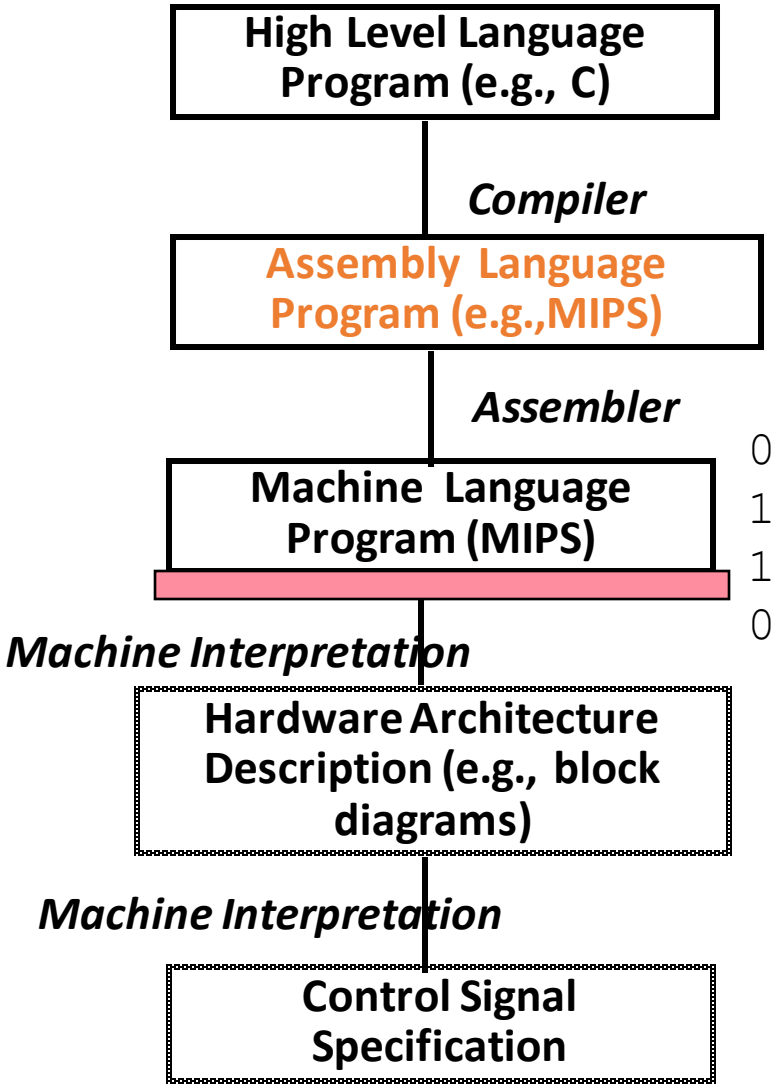
Great Ideas

- Principle of abstraction, used to build systems as layers
- Principle of locality, exploited via a memory hierarchy (cache)
- Making the common case fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Dependability via redundancy

Levels of Abstraction for A Computer



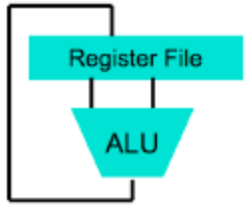
Levels of Representation

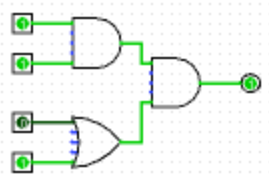



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

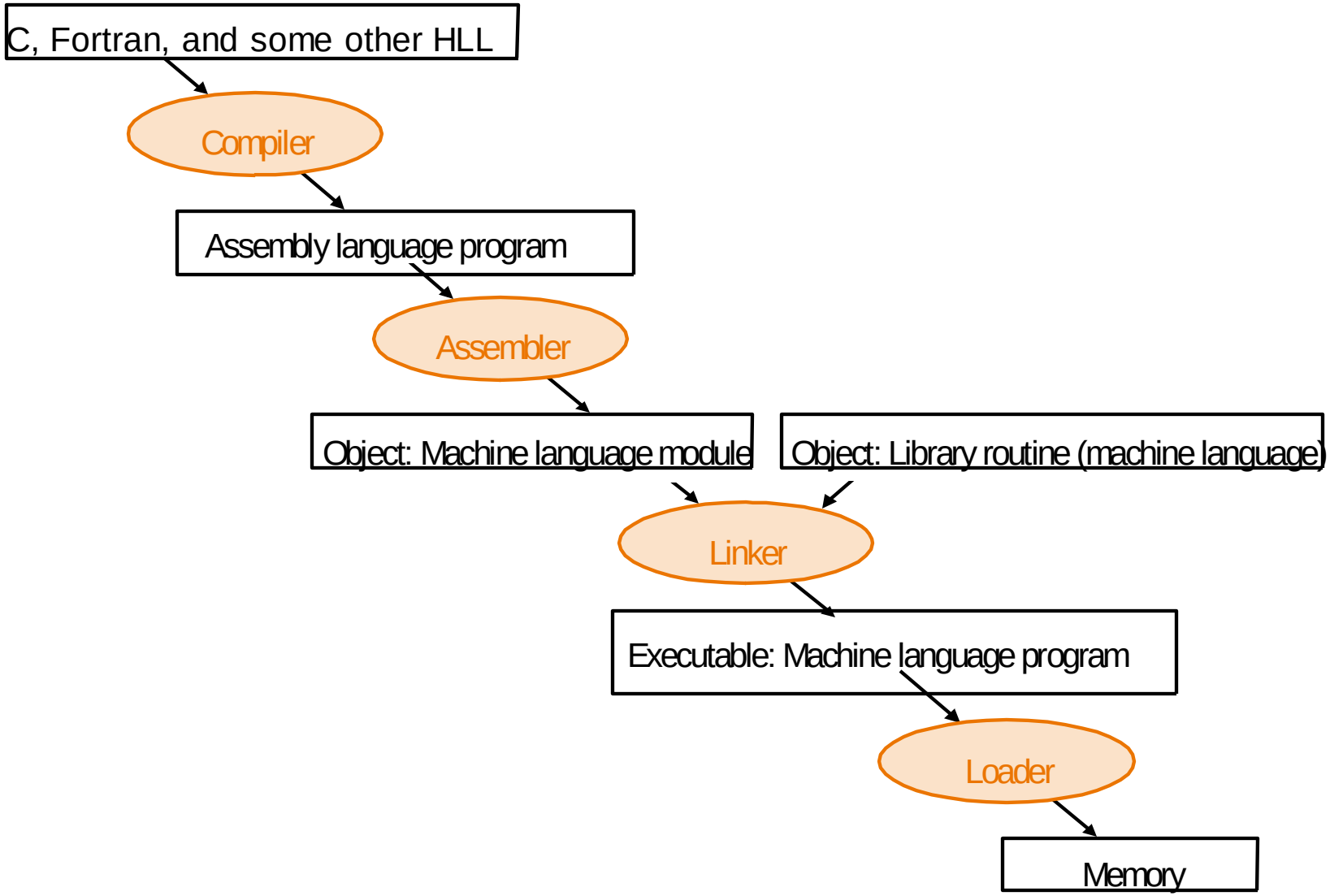
```
lw t0,      0($2)
lw t1,      4($2)
swt1,       0($2)
swt0,       4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

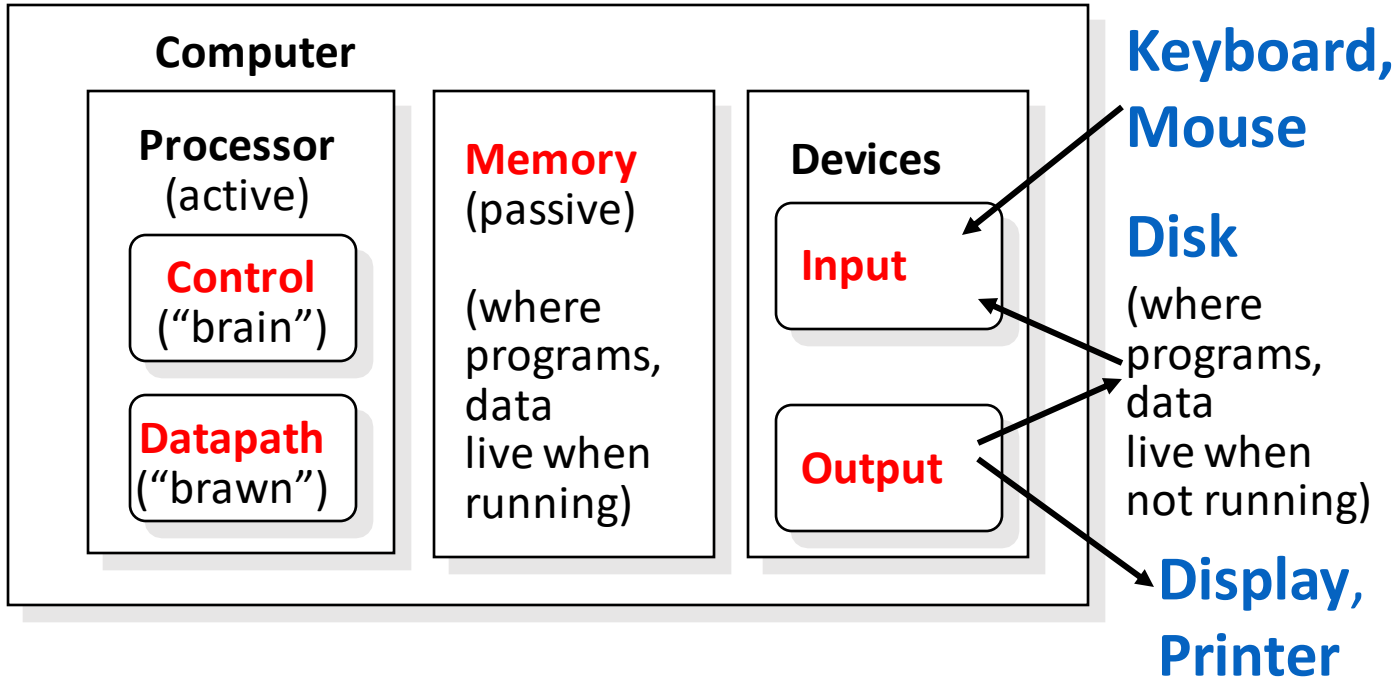




A Translation Hierarchy

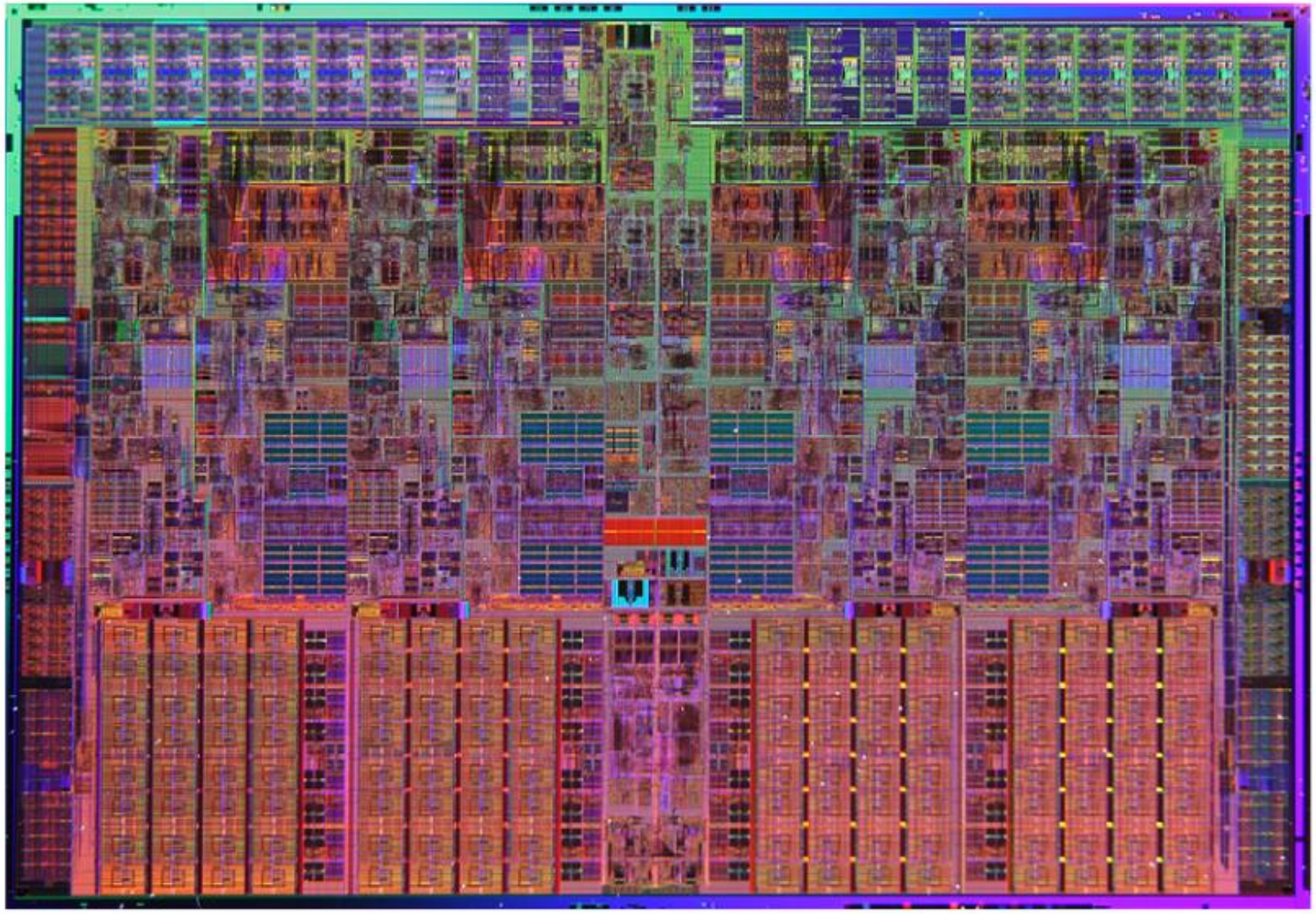


Anatomy: 5 Components of Any Computer

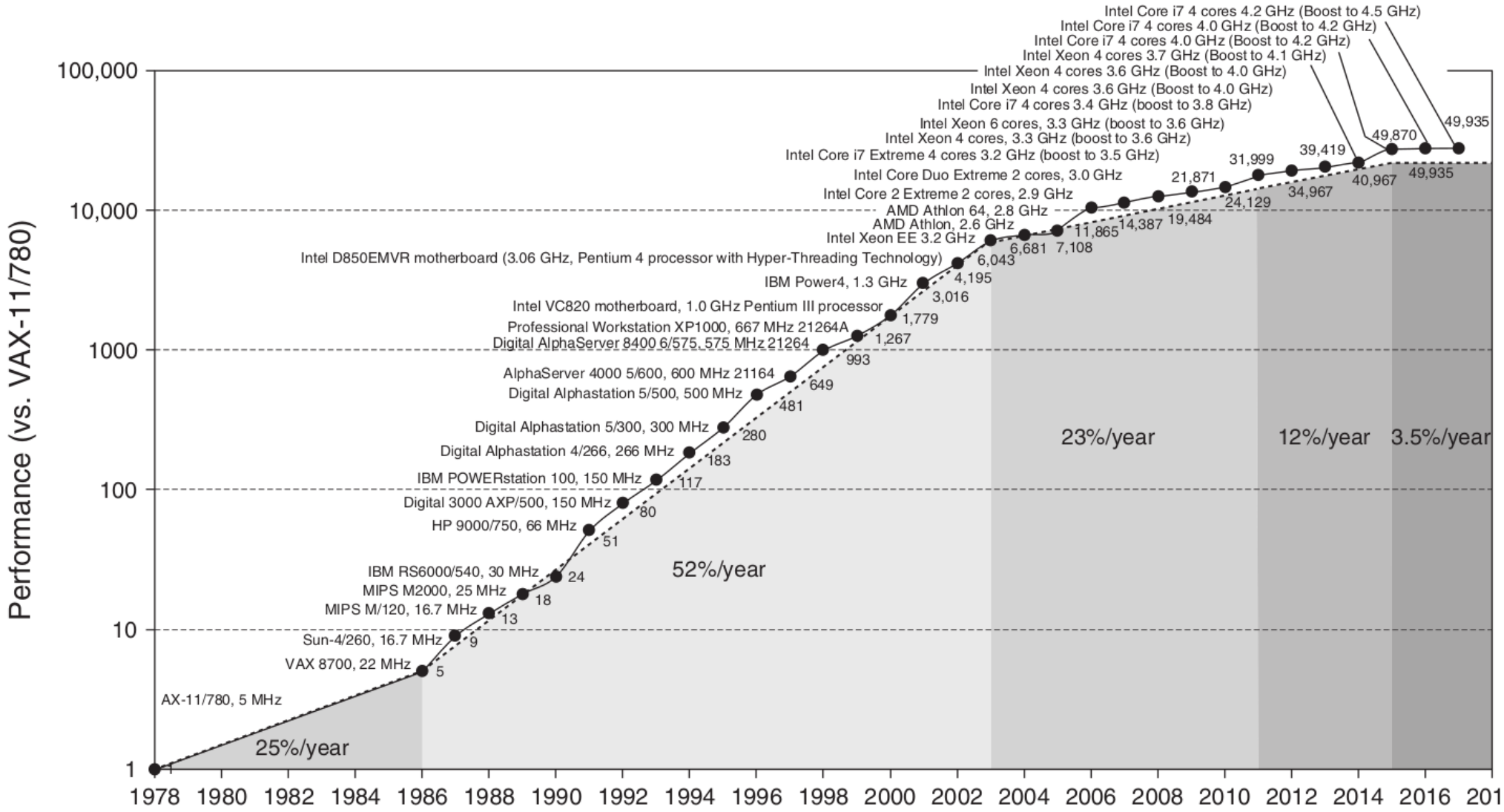


More about Processor (CPU)

- Datapath
- Control
- Registers
- Cache memory

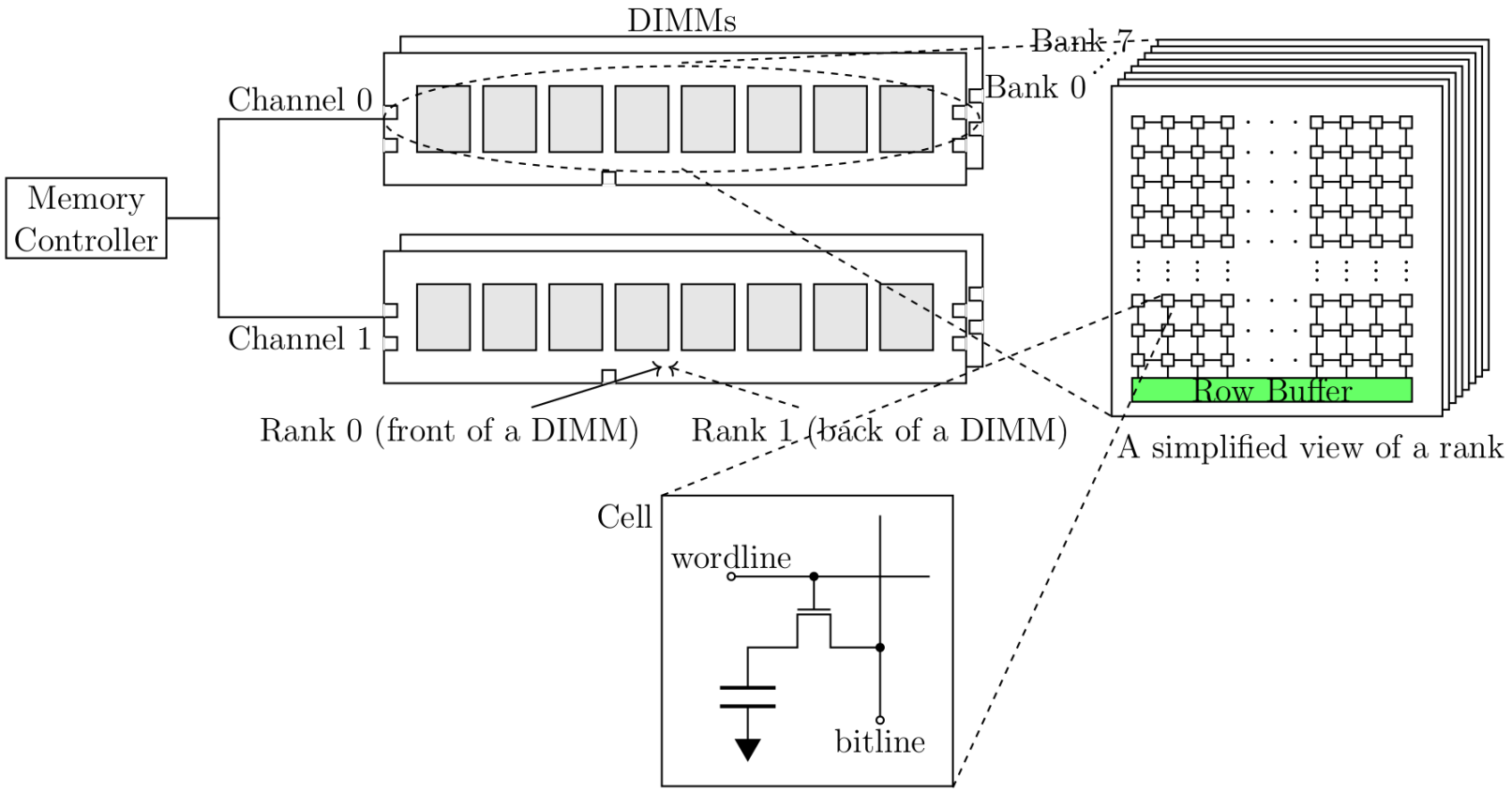


Growth in Processor Performance



More about Memory

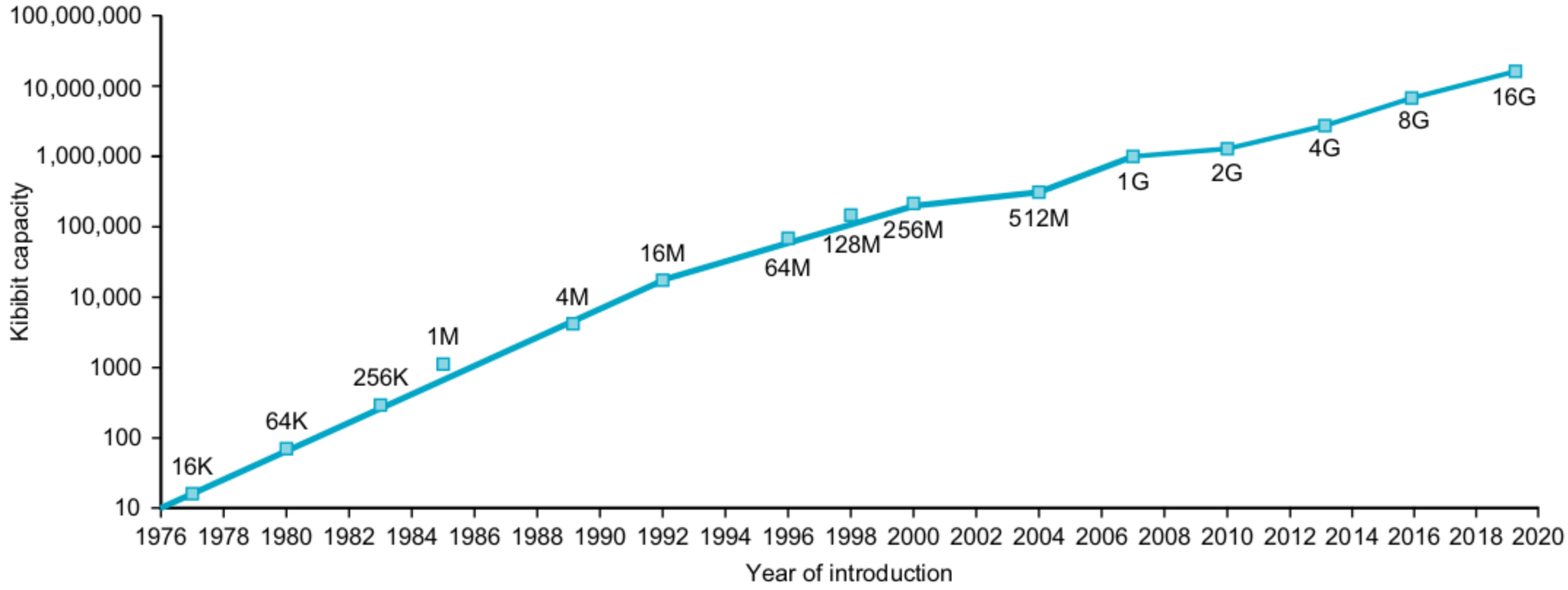
- DIMM
- Rank
- Bank
- Cell



Common Size Terms

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%
ronnabyte	RB	10^{27}	robibyte	RiB	2^{90}	24%
queccabyte	QB	10^{30}	quebibyte	QiB	2^{100}	27%

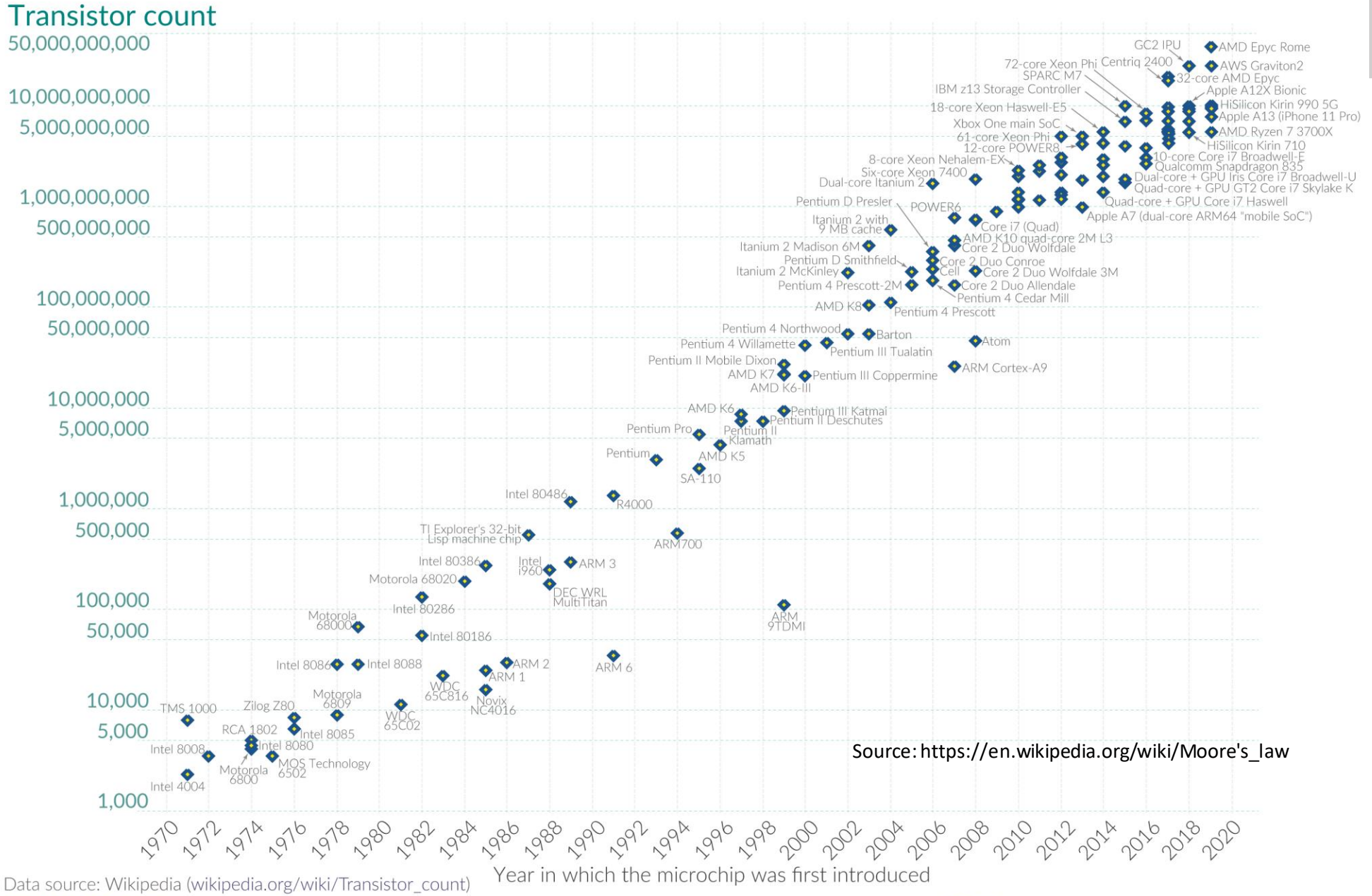
Growth in Memory Capacity



Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World in Data

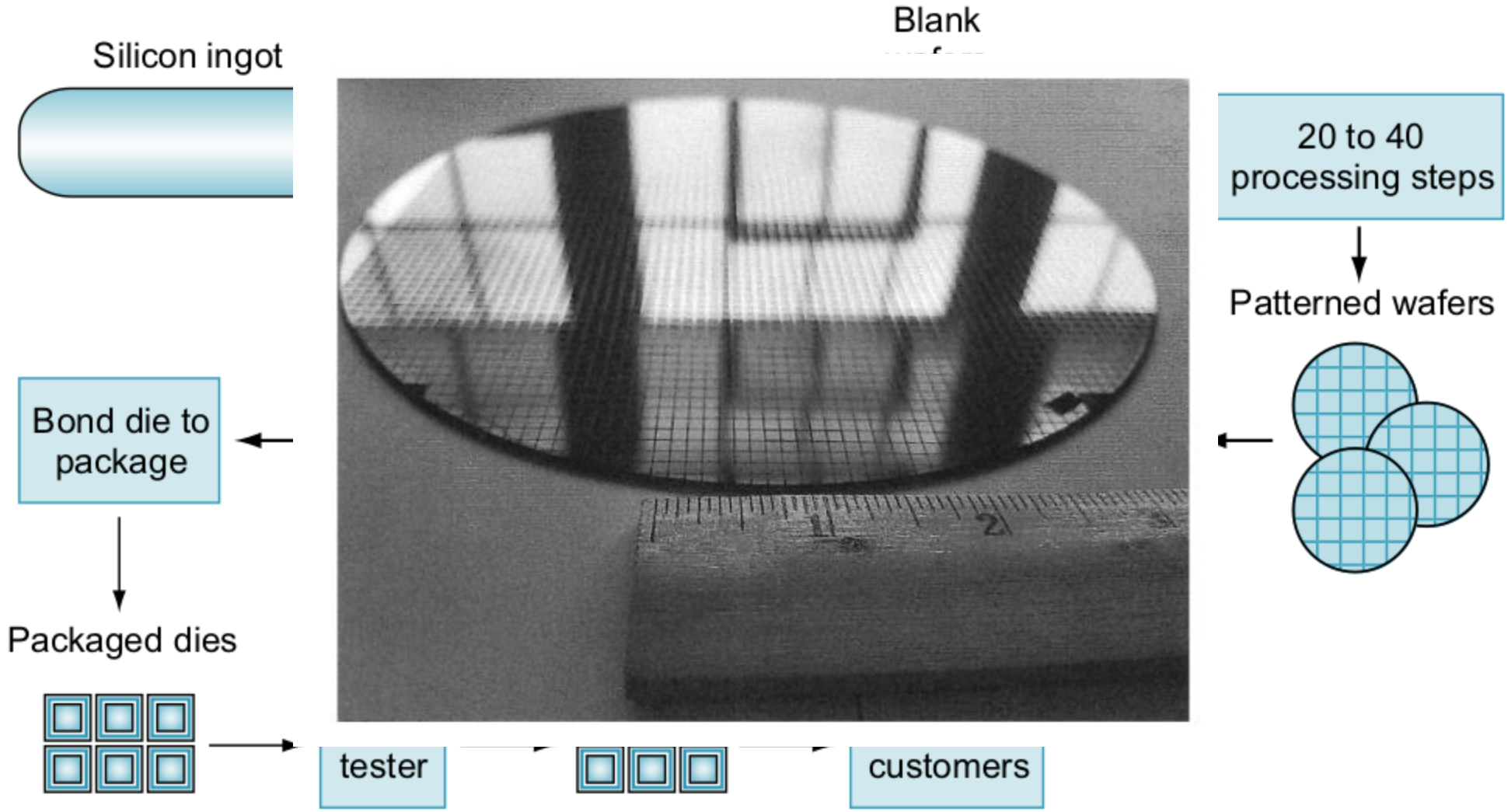


Source: https://en.wikipedia.org/wiki/Moore's_law

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

The Chip Manufacturing Process



Next Lecture

- Section 1.6, 1.7
  - Read the contents
  - Finish pre-class questions