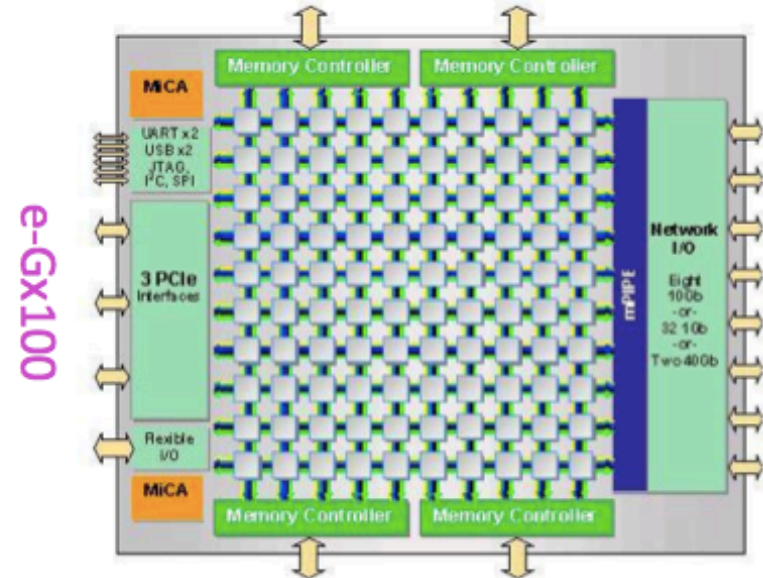
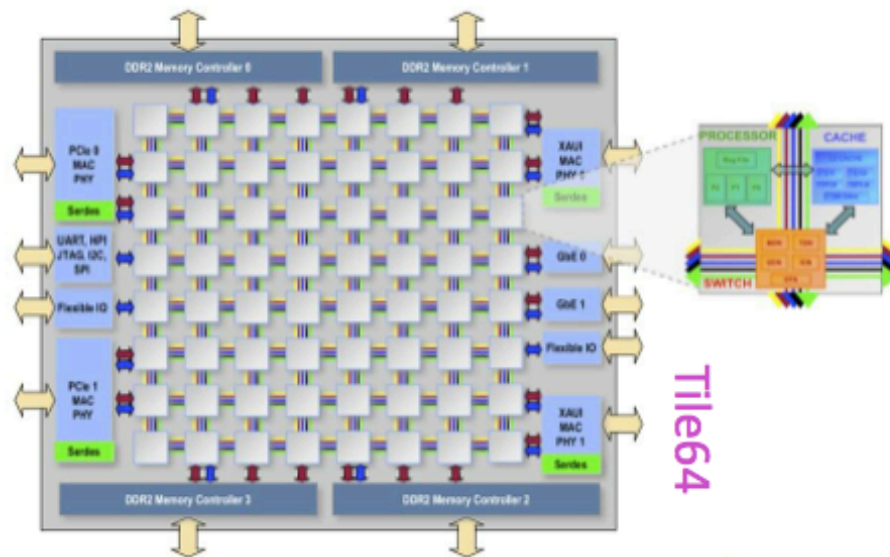


How to read a paper (Dally, Stanford)

- 1. Before diving in, think about what your goals are, and what you want to get out of the paper. You need to approach the reading with an aim to extract info; you may want to target specific areas of the paper in case you are looking for something particular.
- 2. Read the abstract first. This should be an advertisement for the paper.
- 3. Read the conclusions. Figure out what the authors accomplished, so you know what their goals were.
- 4. Find the “paper overview” section in the introduction and read it if you need to. If you don't know about a certain topic, you may find background material here.
- 5. Make a quick pass through the paper, so you understand their presentation of the topics. Figure out how the equations fit into the discussion, but don't try to understand the math. On the second pass, read the paper more carefully.
- 6. Highlight important parts of a paragraph so you don't have to reread the entire paragraph each time. This helps in class discussions as well.
- 7. Write down an outline as you go- this will help you remember what you have read. You can refer back to this outline as you go through the paper.
- 8. Look up key references if you are looking for a particular topic. This paper may not have what you are looking for, but it may build on top of a topic you need to research. In the WEB, you can find guides that list forward references for each paper, which may also be useful in your quest.

Multi-Core versus Many Core



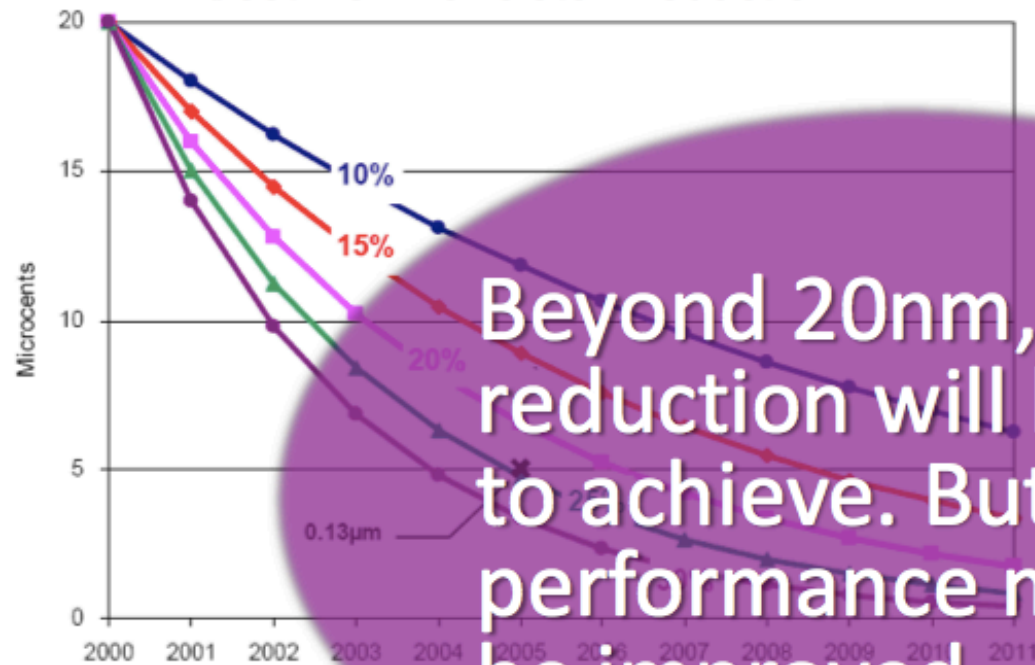
Comparison between Tile and Intel Core

Processor	Number of Cores	Frequency	Avg Power Consumption
TILEPro36	36	500MHz	9-13W
TILE64/Pro64	64	700MHz/866MHz	15-23W (all cores)
TILE-Gx36	36	1.25GHz/1.5GHz	10-55W
TILE-Gx64	64	1.25GHz/1.5GHz	10-55W
TILE-Gx100	100	1.25GHz/1.5GHz	10-55W
Intel Core i7-920	4	2.66GHz	130W (max TDP)

As processor working frequency increases, it is impossible to keep power consumption in the reasonable limits. Multi-core becomes a natural choice in order to use the power advantage implied in parallelization.

Cost Per Transistor Reduction

(US Dollar)



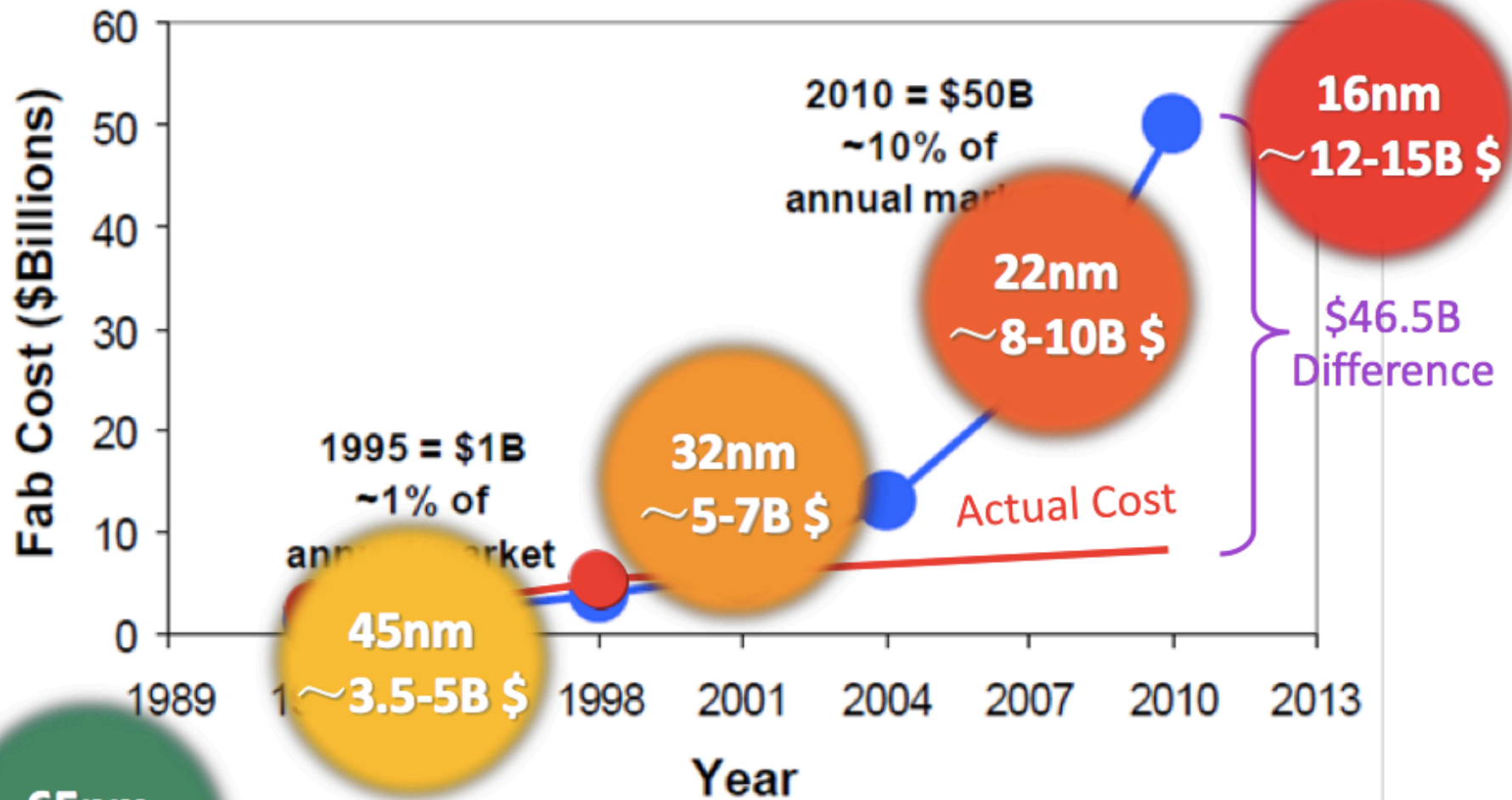
130nm	2,450
90nm	2,810
65nm	3,106
45nm	4,024
32nm	4,817
22nm	6,638

300mm Foundry Wafer Price Trends (2012 Perspective)

	0.13µm	90nm	65nm	45nm	32nm	22nm
Cost per gate (microcent)	18.88	12.54	9.34	7.67	6.89	6.66
• Percent decline (%)	NA	(33.6)	(25.6)	(17.9)	(10.1)	(3.3)

Cost Per Gate Trends

Source: Industry Restructuring, IBS Report, 2007



65nm
~2.5-3B \$

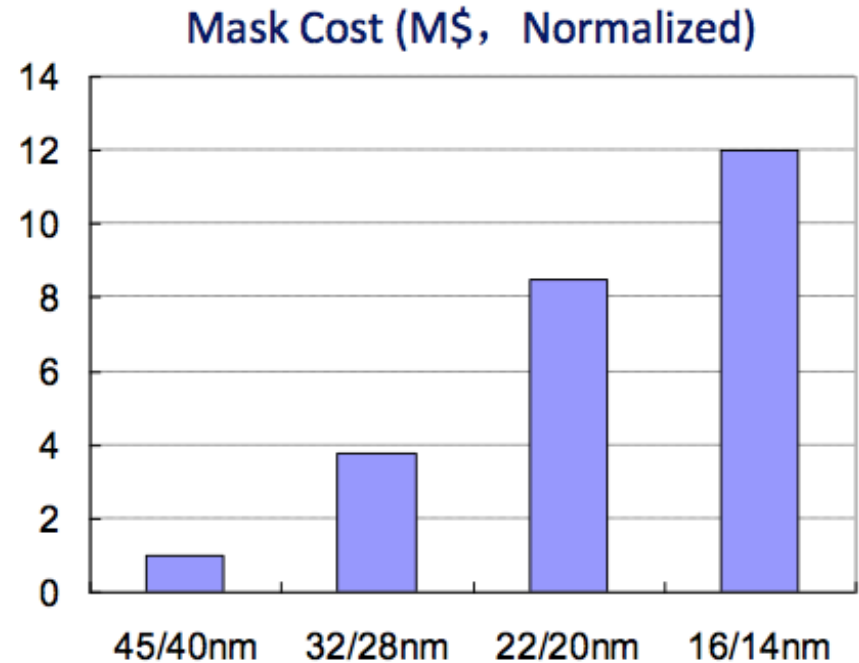
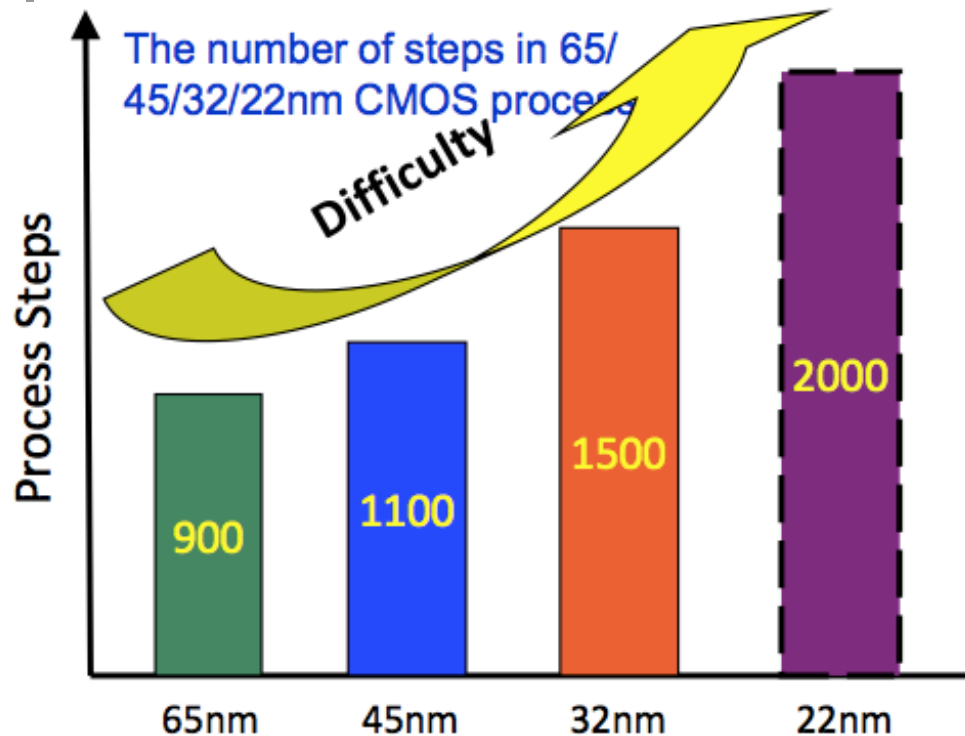
projection, Karen Brown, "Beyond the Wall: Technologies for the Future"

2001 Symposium on Microlithography; 2009 Actual, IC Insights

Wally Rhines, Chairman & CEO, Mentor Graphics, August 2010

Recently, many IDMs' have announced that they would not invest in developing new fabrication process and become Fab-lite or fabless.





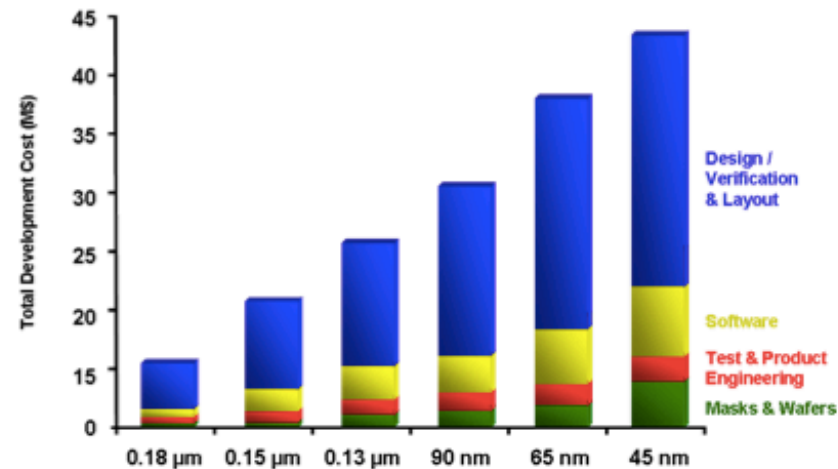
Wafer Cost Perspective for 450mm Facility

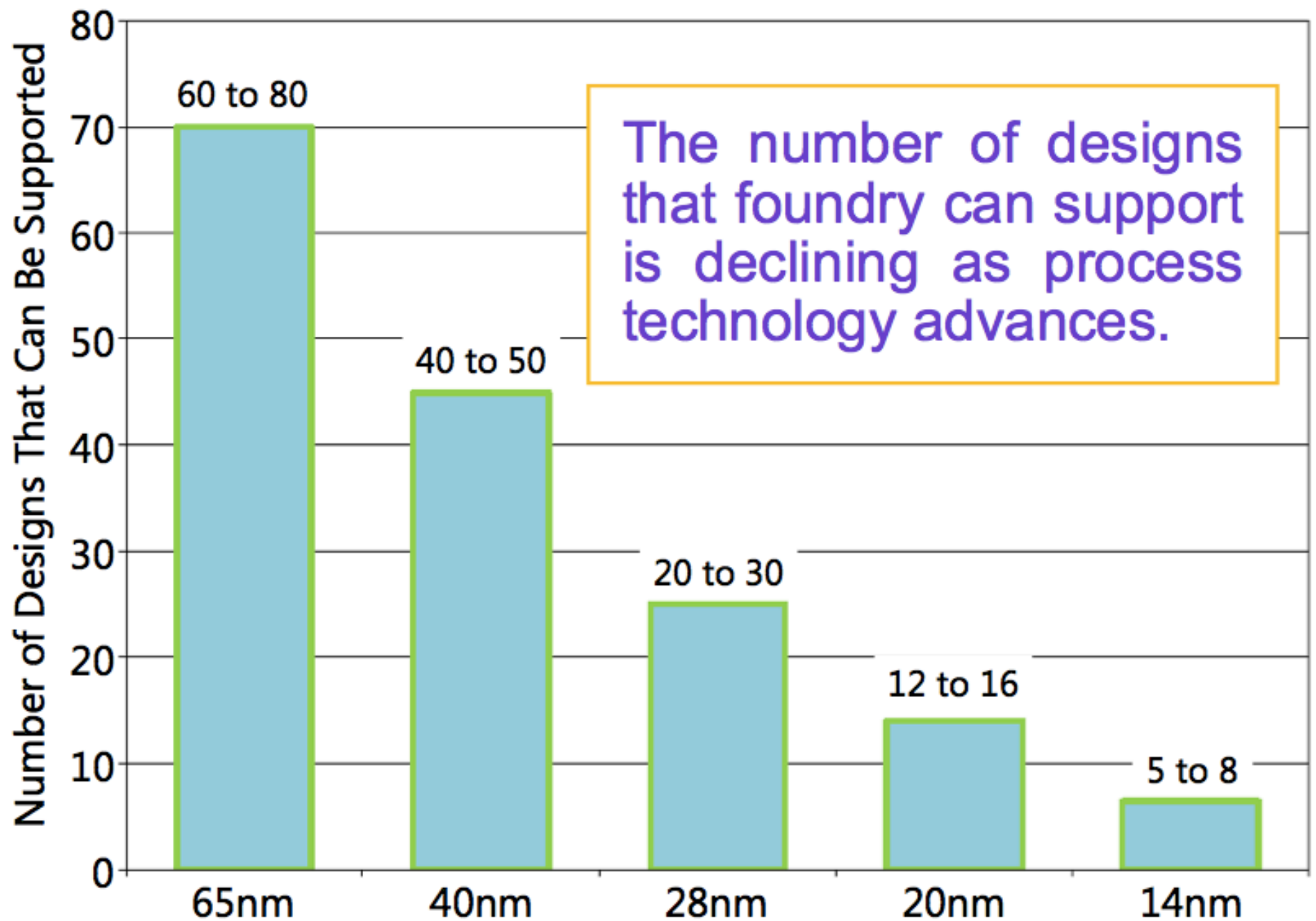
Source: IBS Inc, 2012

	Costs (\$M)					Cost/ wafer (\$)	Revenues/ wafer (\$) (50% GPM)	Annual revenues/ fab (\$M)
	Facilities	Equipment	Annual depreciation	Operating costs	TOTAL cost/year			
10K WPM	1,024	1,729	448	1,127	1,575.2	13,126.7	26,253.3	3,150.4
20K WPM	1,175	3,420	802	2,084	2,885.5	12,022.9	24,045.8	5,771.0
40K WPM	1,364	7,796	1,696	3,753	5,448.6	11,351.3	22,702.5	10,897.2
80K WPM	1,581	15,348	3,228	6,567	9,794.7	10,202.8	20,405.6	19,589.4

Technology node (μm)	K gates per sq. mm			Utilization (%)	Gate count (M)		
	Potential	Actual	Average		5mm x 5mm	10mm x 10mm	20mm x 20mm
0.25	112	92 to 97	94.2	81.8 to 86.4	2.3 to 2.4	9.2 to 9.7	36.6 to 38.7
0.18	164	133 to 140	136.7	81.1 to 85.6	3.3 to 3.5	13.3 to 14.0	53.2 to 56.2
0.13	248	199 to 209	204.0	80.2 to 84.3	5.0 to 5.2	19.9 to 20.9	79.6 to 83.6
0.090	443	344 to 361	352.2	77.6 to 81.4	8.6 to 9.0	34.4 to 36.1	137.5 to 144.2
0.065	694	500 to 545	522.9	72.1 to 78.6	12.5 to 13.6	50.0 to 54.5	200.1 to 218.2
0.045	1,179	778 to 871	824.7	66.0 to 73.9	19.5 to 21.8	77.8 to 87.1	311.3 to 348.5
0.032	1,723	1,023 to 1,173	1,098.4	59.4 to 68.1	25.6 to 29.3	102.3 to 117.3	409.4 to 469.3
0.022	2,726	1,450 to 1,682	1,566.1	53.2 to 61.7	36.3 to 42.0	145.0 to 168.2	580.1 to 672.8

%	0.13 μm	90nm	65nm	45nm	32nm	22nm
Gross profit margin	46.0	48.0	50.0	52.0	55.0	57.0
• Product development R&D	14.0	17.0	20.0	23.0	27.0	30.0
• SG&A	12.0	11.0	10.0	9.0	8.0	7.0
• Operating income	20.0	20.0	20.0	20.0	20.0	20.0
Cost of goods sold	54.0	52.0	50.0	48.0	45.0	43.0
• Wafer costs	35.0	35.0	35.0	34.0	30.0	26.0
• Packaging costs	10.0	9.0	8.0	8.0	9.0	10.0
• Testing costs	9.0	8.0	7.0	6.0	6.0	7.0
Revenues	100.0	100.0	100.0	100.0	100.0	100.0



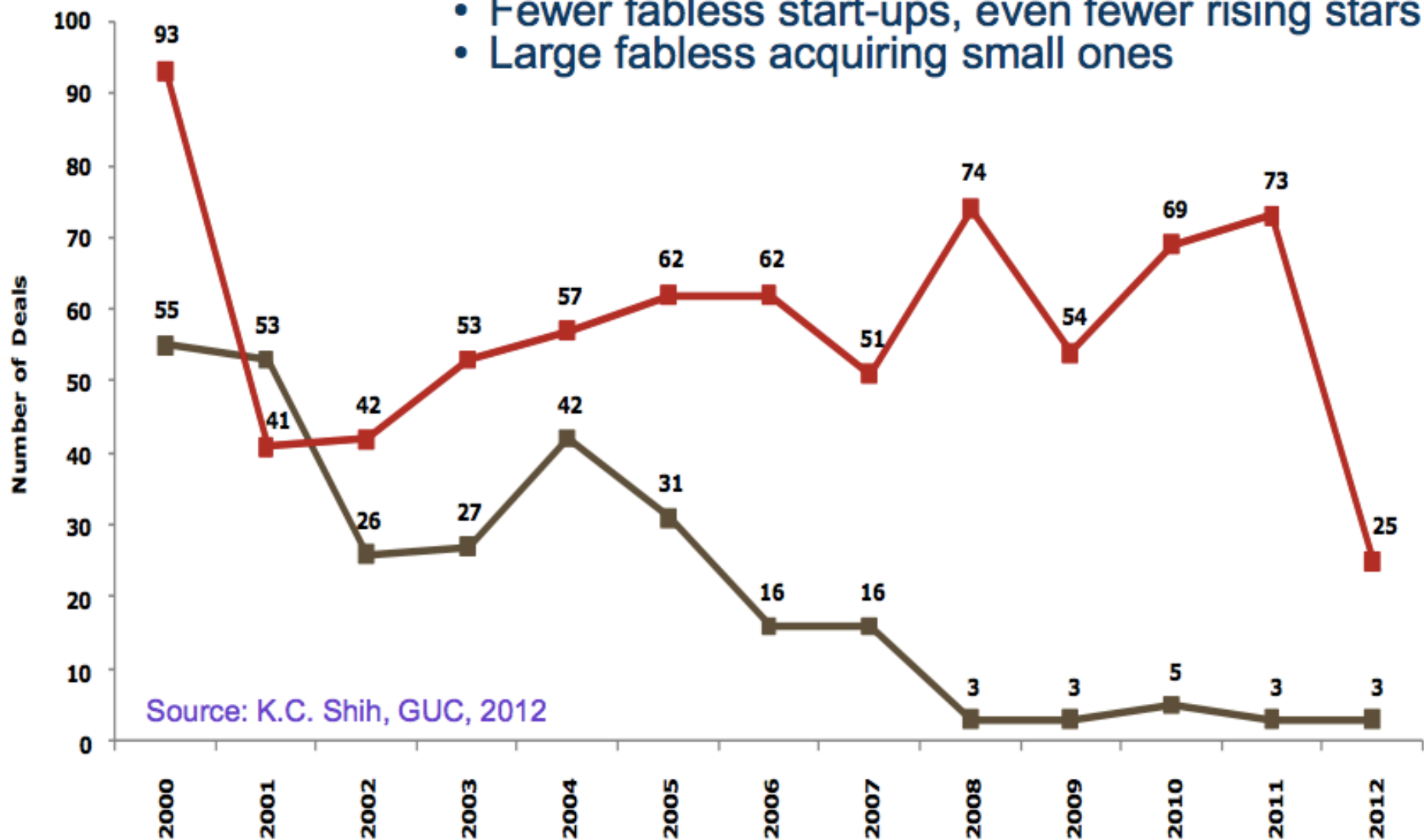


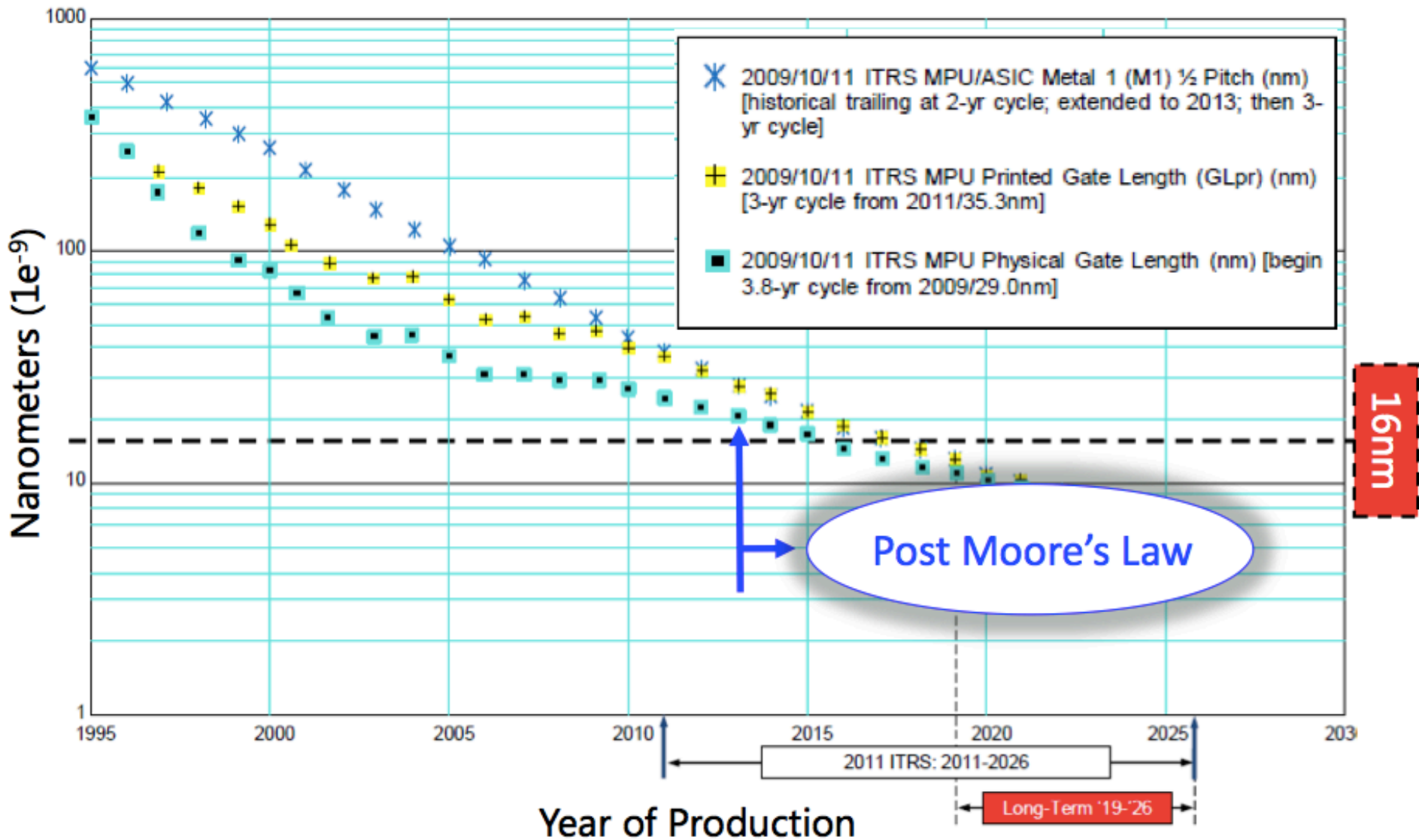
Source : Process Technology and Ecosystems, IBS, 2012

■ Fabless Semiconductor Series A Fundings

■ Fabless Semiconductor M&As by Year

- Fewer fabless start-ups, even fewer rising stars
- Large fabless acquiring small ones



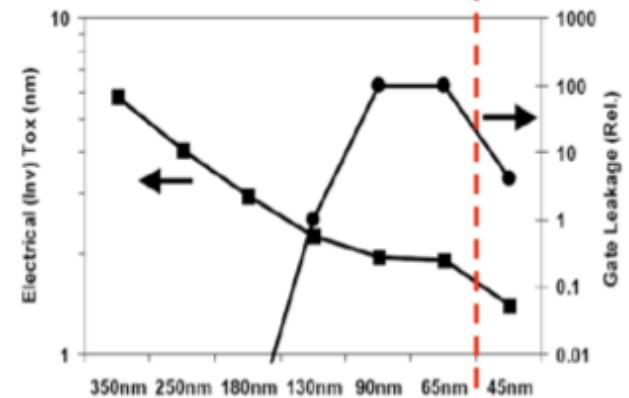




- 1) New integration and process technology development
- 2) New material introduction

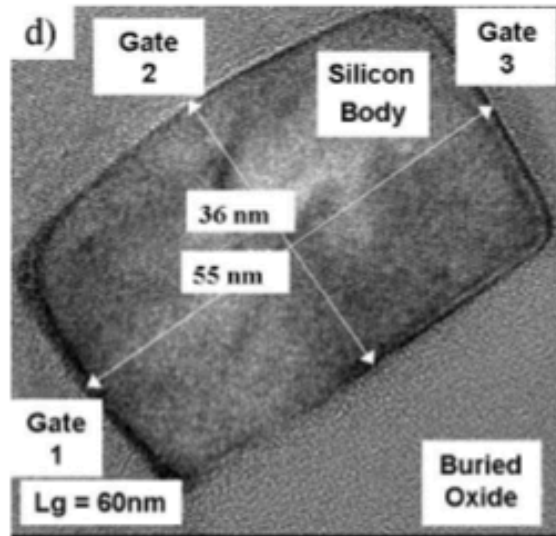
1980s2000s

Poly/SiON ↔ MG/HK



- High-K/MG enables to resume the T_{inv} scaling required to sustain Moore's law

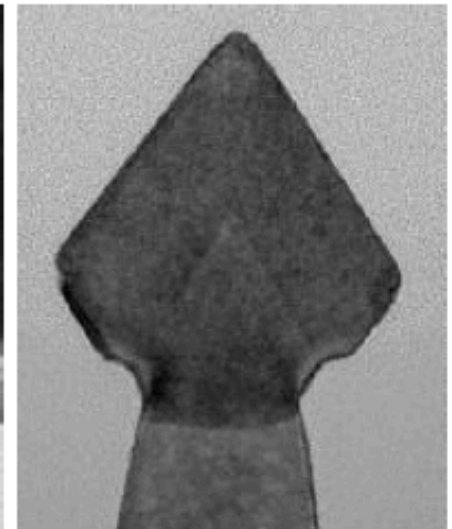
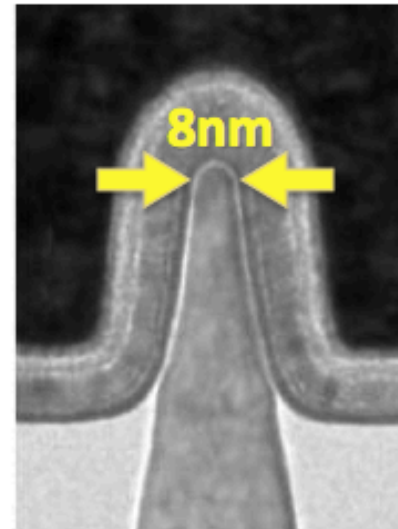
FinFET



Doyle, B., IEDM, 2003



8 years R&D



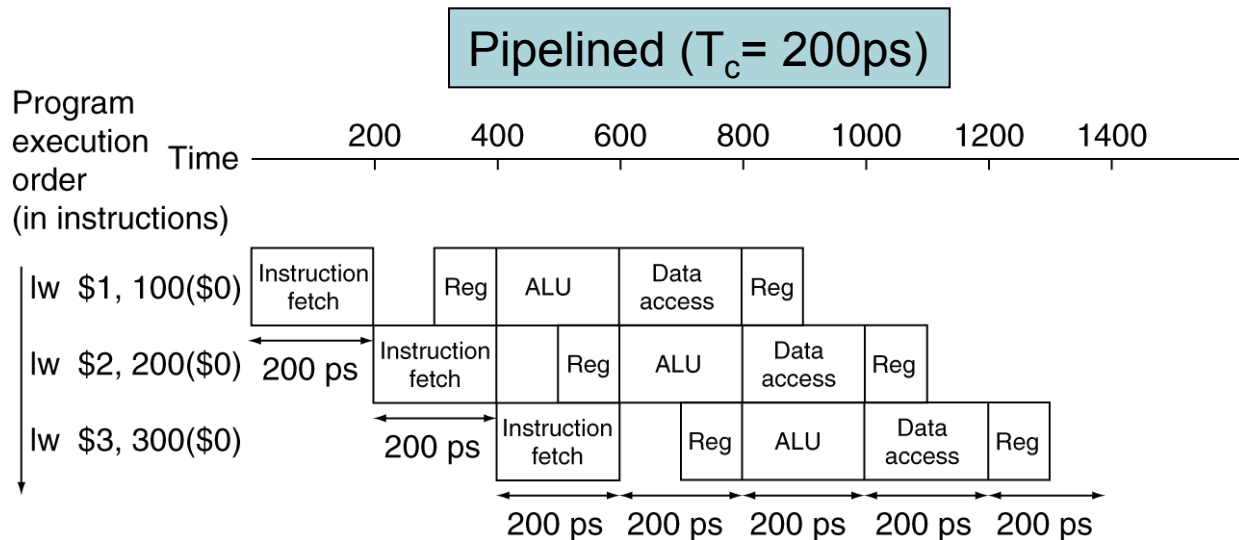
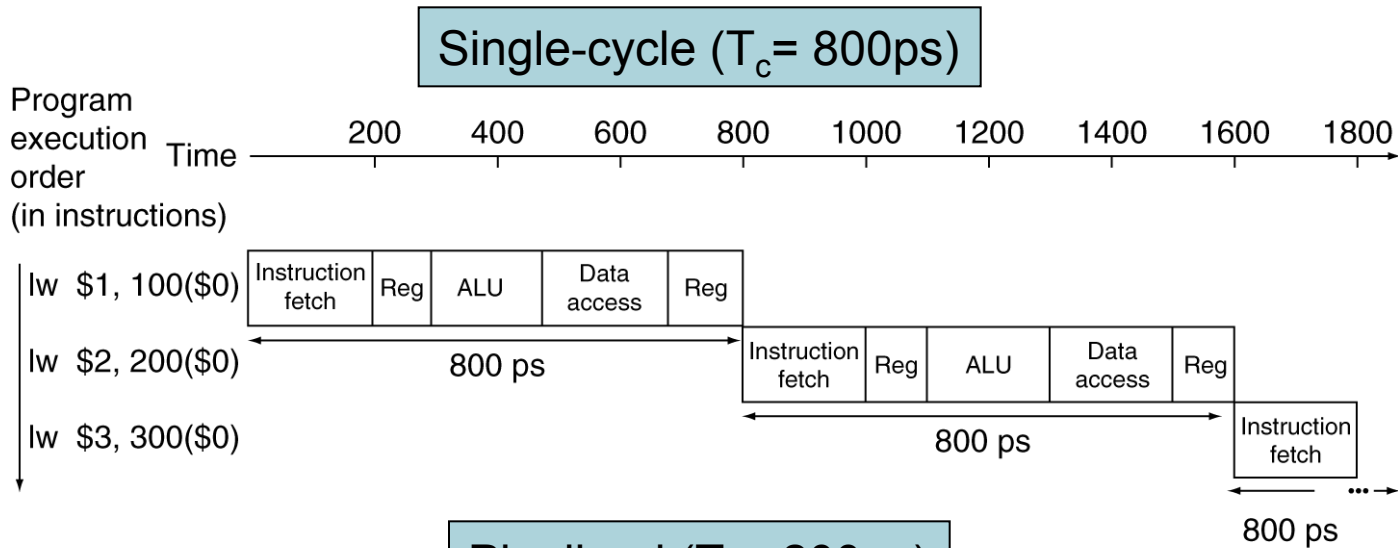
C. Auth, VLSI, 2012

Review of Pipelining

MIPS Pipeline

- Five stages, one step per stage
 1. IF: Instruction fetch from memory
 2. ID: Instruction decode & register read
 3. EX: Execute operation or calculate address
 4. MEM: Access memory operand
 5. WB: Write result back to register

Pipeline Performance



Pipeline Speedup

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

Pipelining and ISA Design

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

Hazards

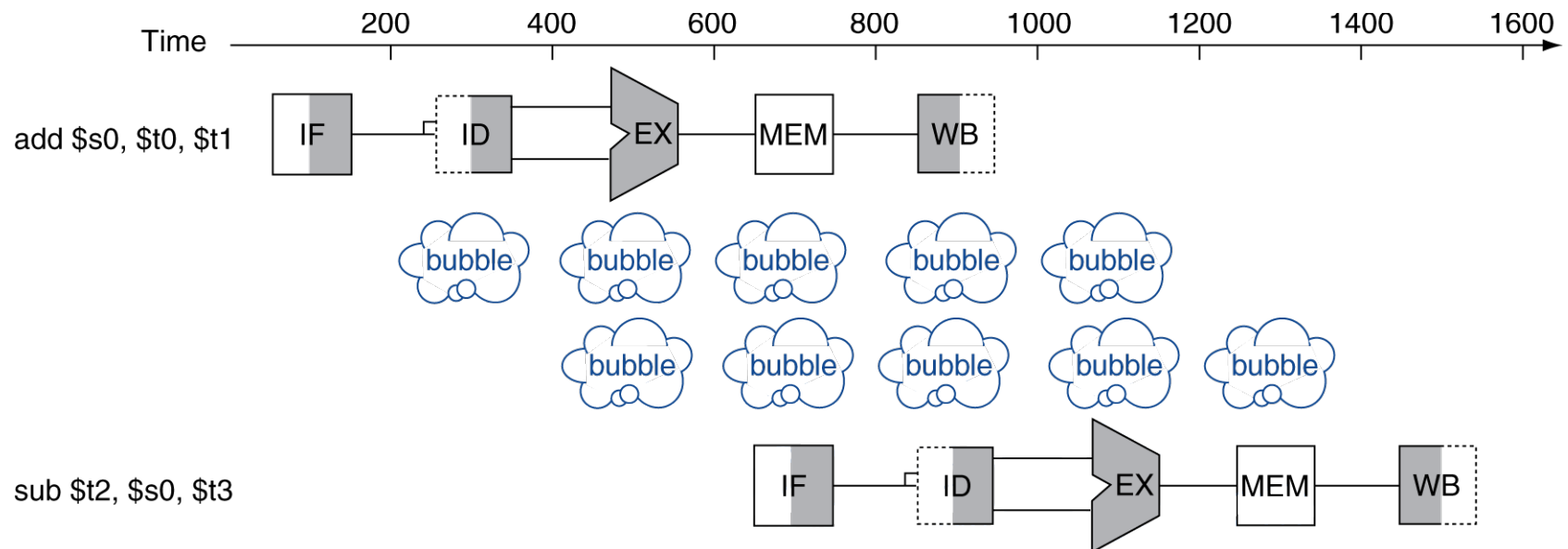
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
 - A required resource is busy
- Data hazard
 - Need to wait for previous instruction to complete its data read/write
- Control hazard
 - Deciding on control action depends on previous instruction

Structure Hazards

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

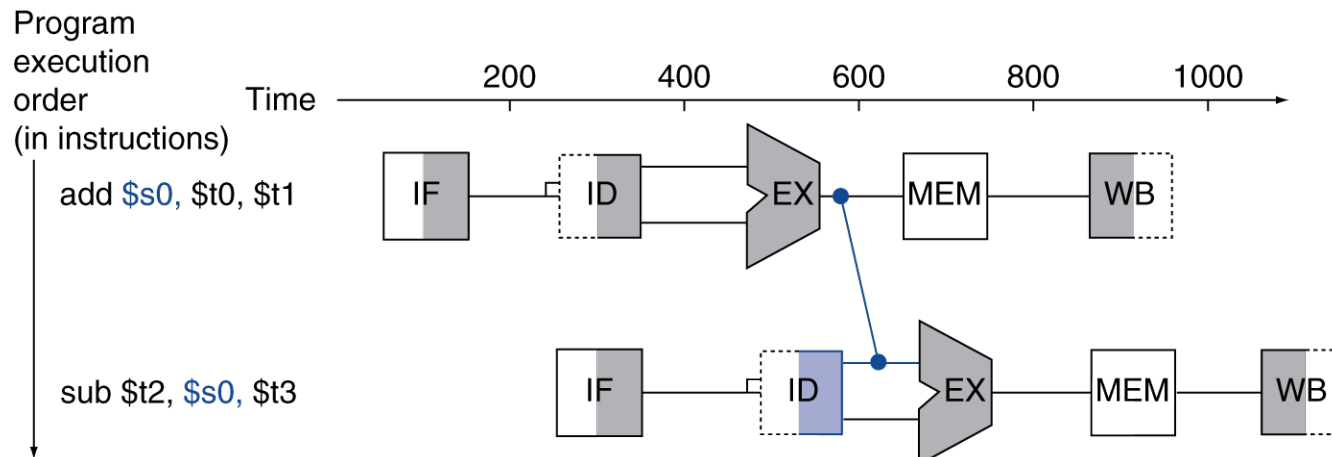
Data Hazards

- An instruction depends on completion of data access by a previous instruction
 - add $\$s0$, $\$t0$, $\$t1$
sub $\$t2$, $\$s0$, $\$t3$



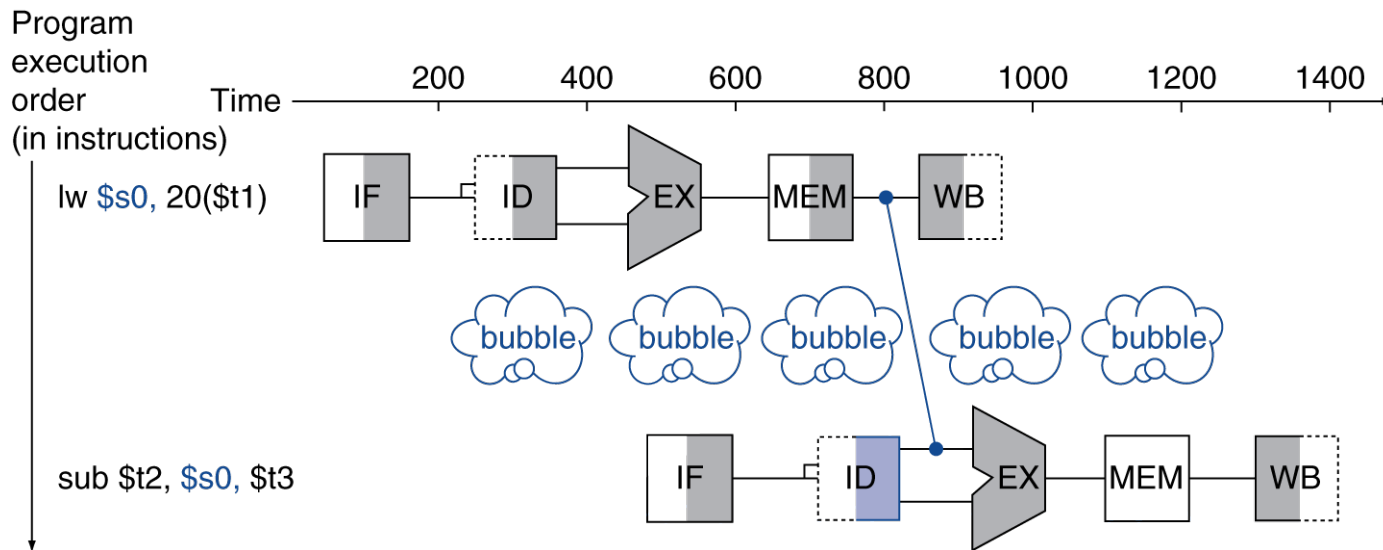
Forwarding (aka Bypassing)

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



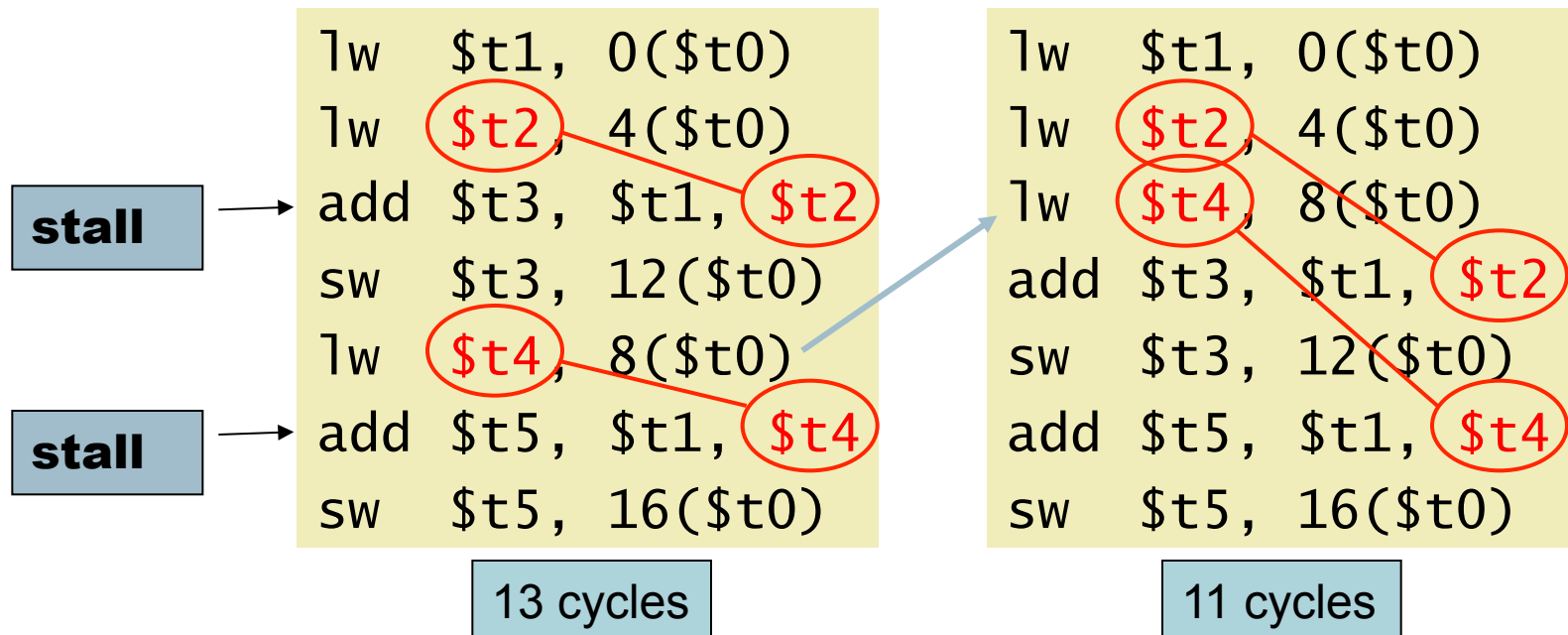
Load-Use Data Hazard

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



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $A = B + E$; $C = B + F$;

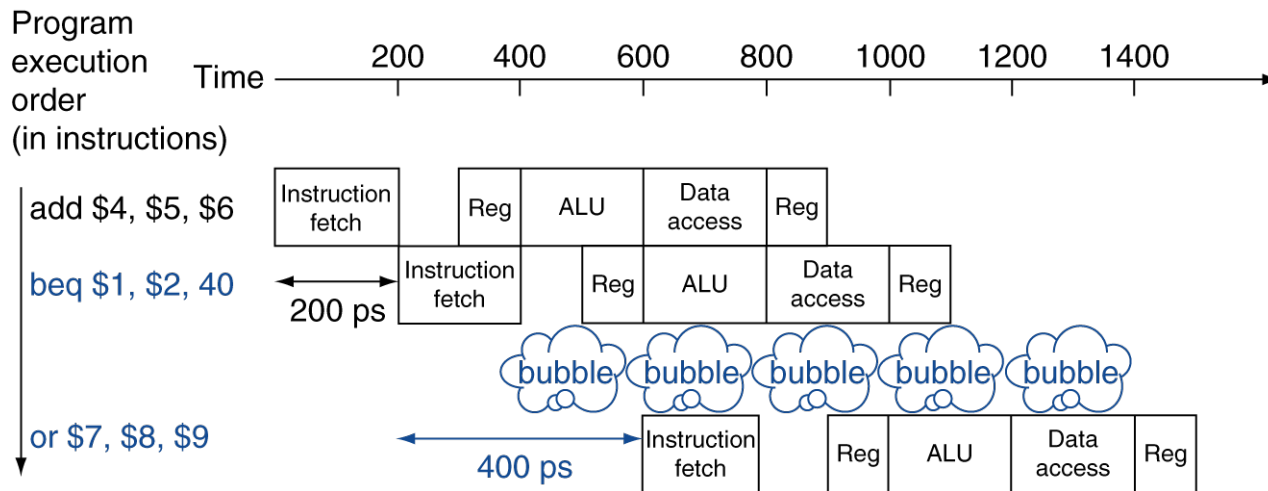


Control Hazards

- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - Still working on ID stage of branch
- In MIPS pipeline
 - Need to compare registers and compute target early in the pipeline
 - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

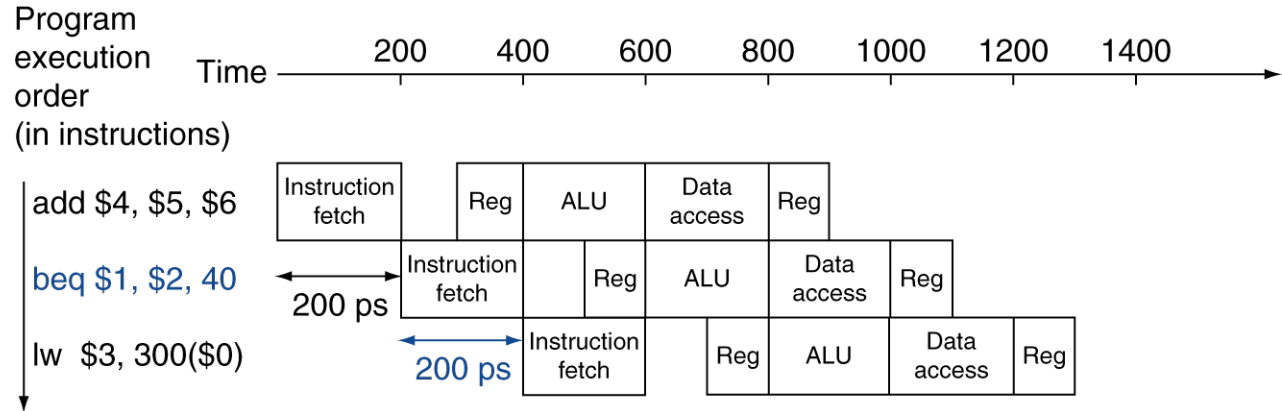


Branch Prediction

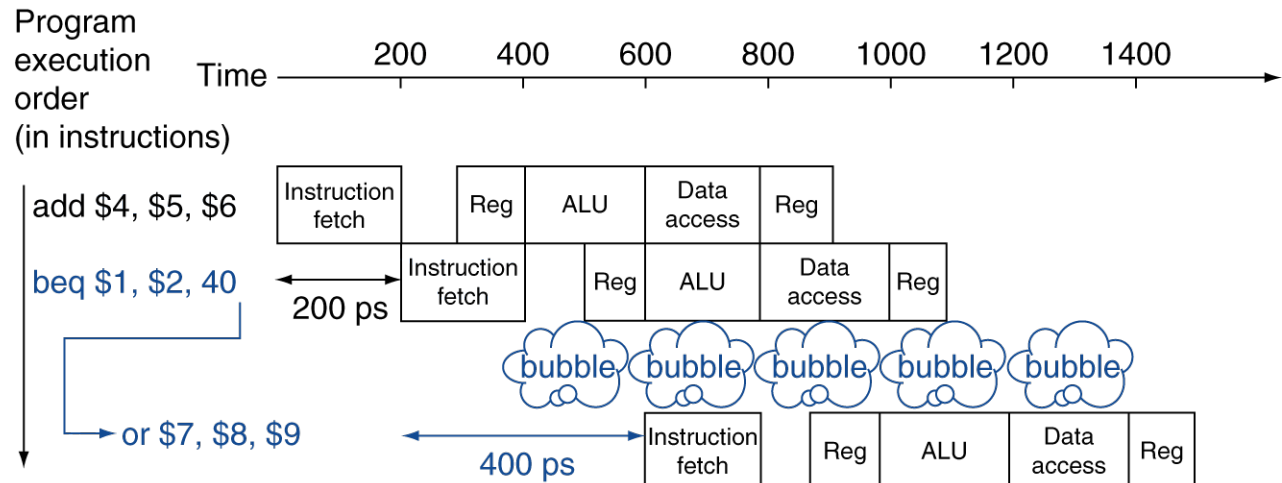
- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
 - Only stall if prediction is wrong
- In MIPS pipeline
 - Can predict branches not taken
 - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

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

Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

MIPS Pipelined Datapath

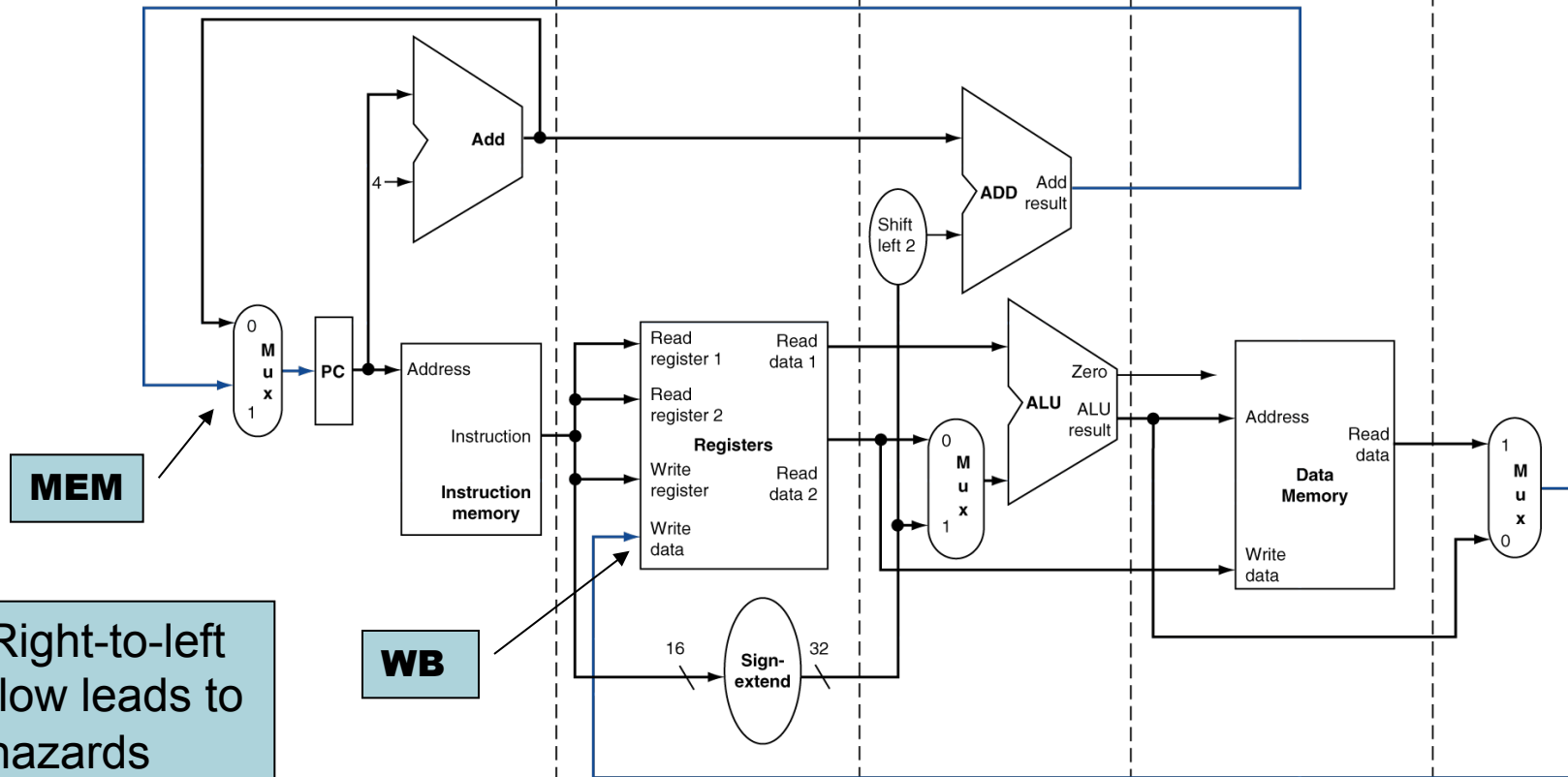
IF: Instruction fetch

ID: Instruction decode/
register file read

EX: Execute/
address calculation

MEM: Memory access

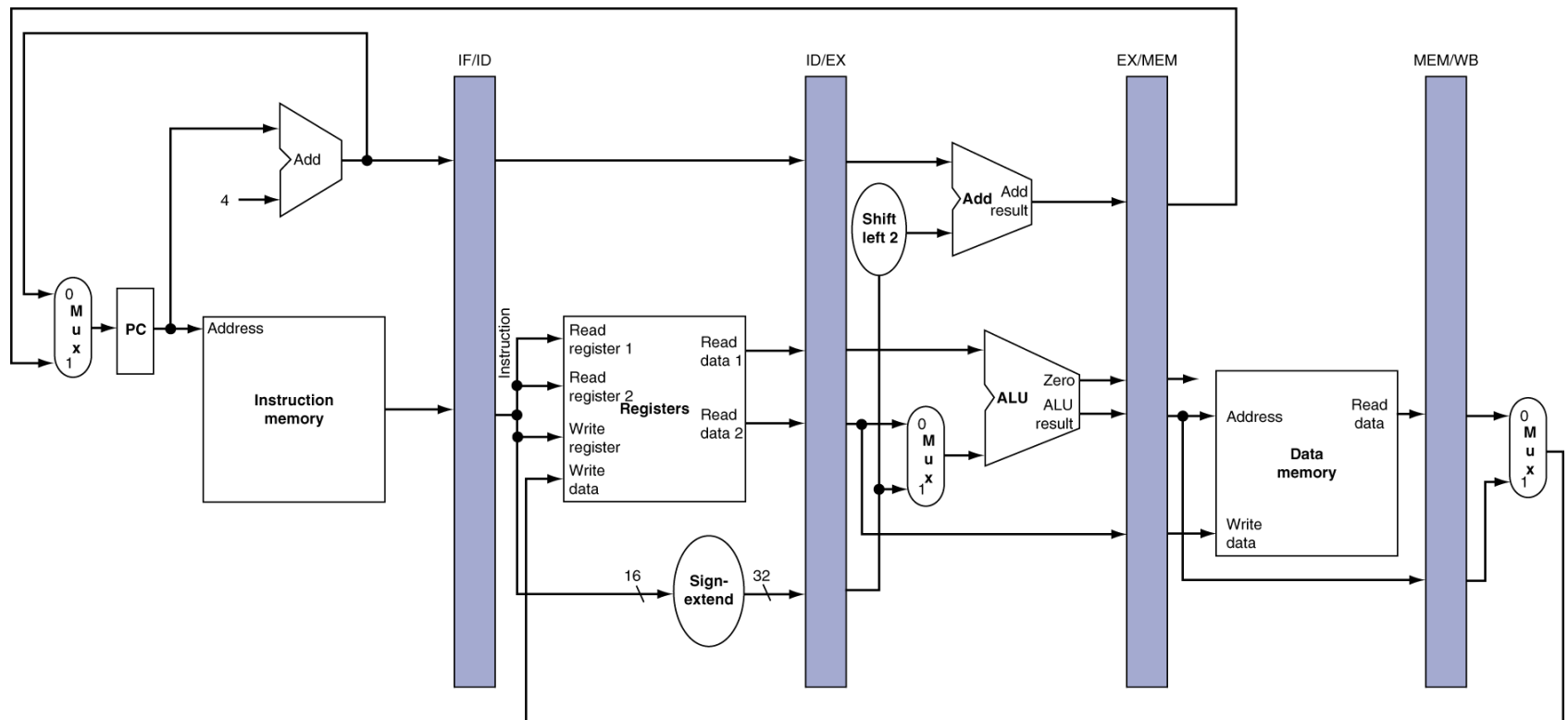
WB: Write back



Right-to-left
flow leads to
hazards

Pipeline registers

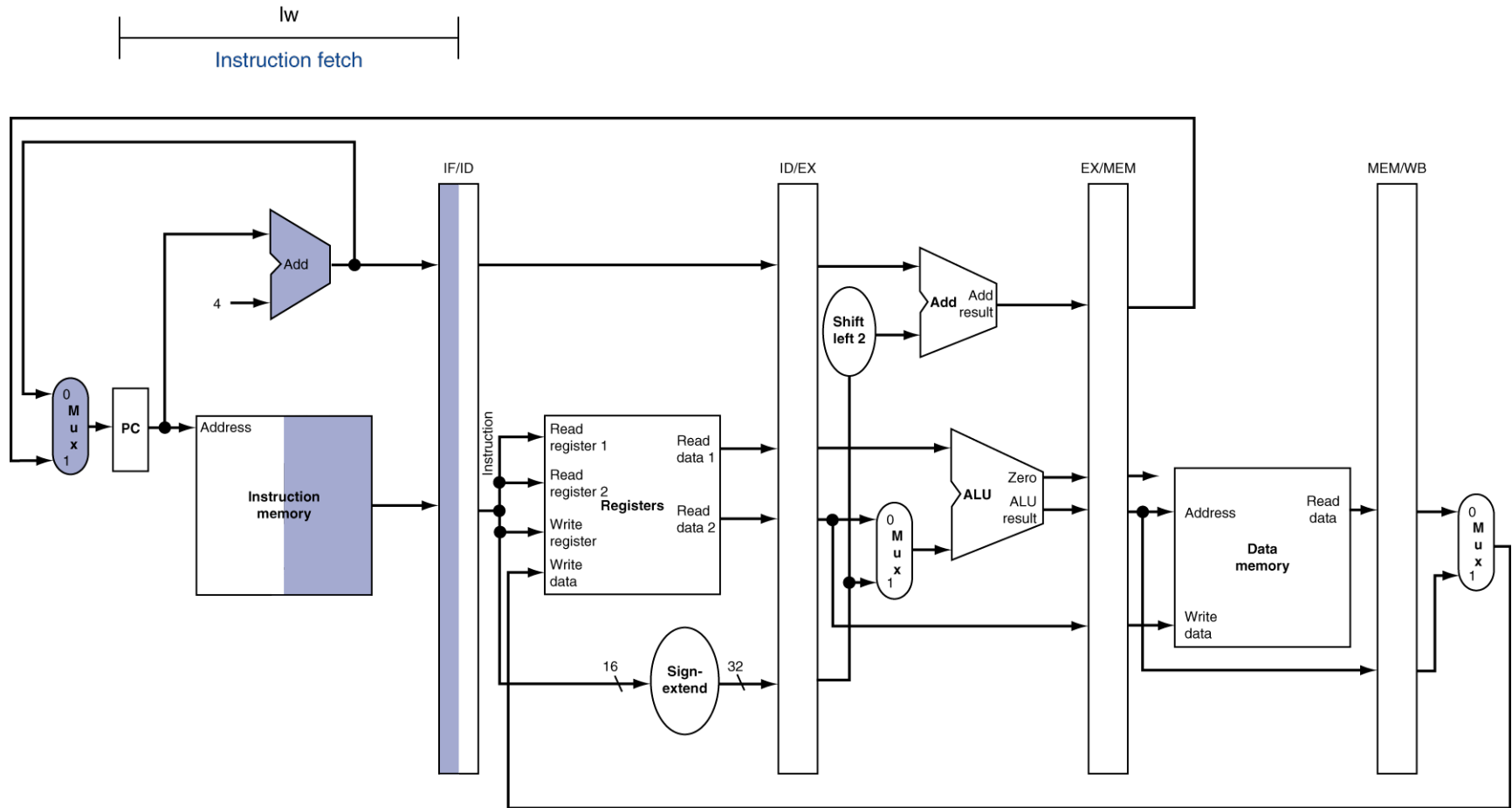
- Need registers between stages
 - To hold information produced in previous cycle



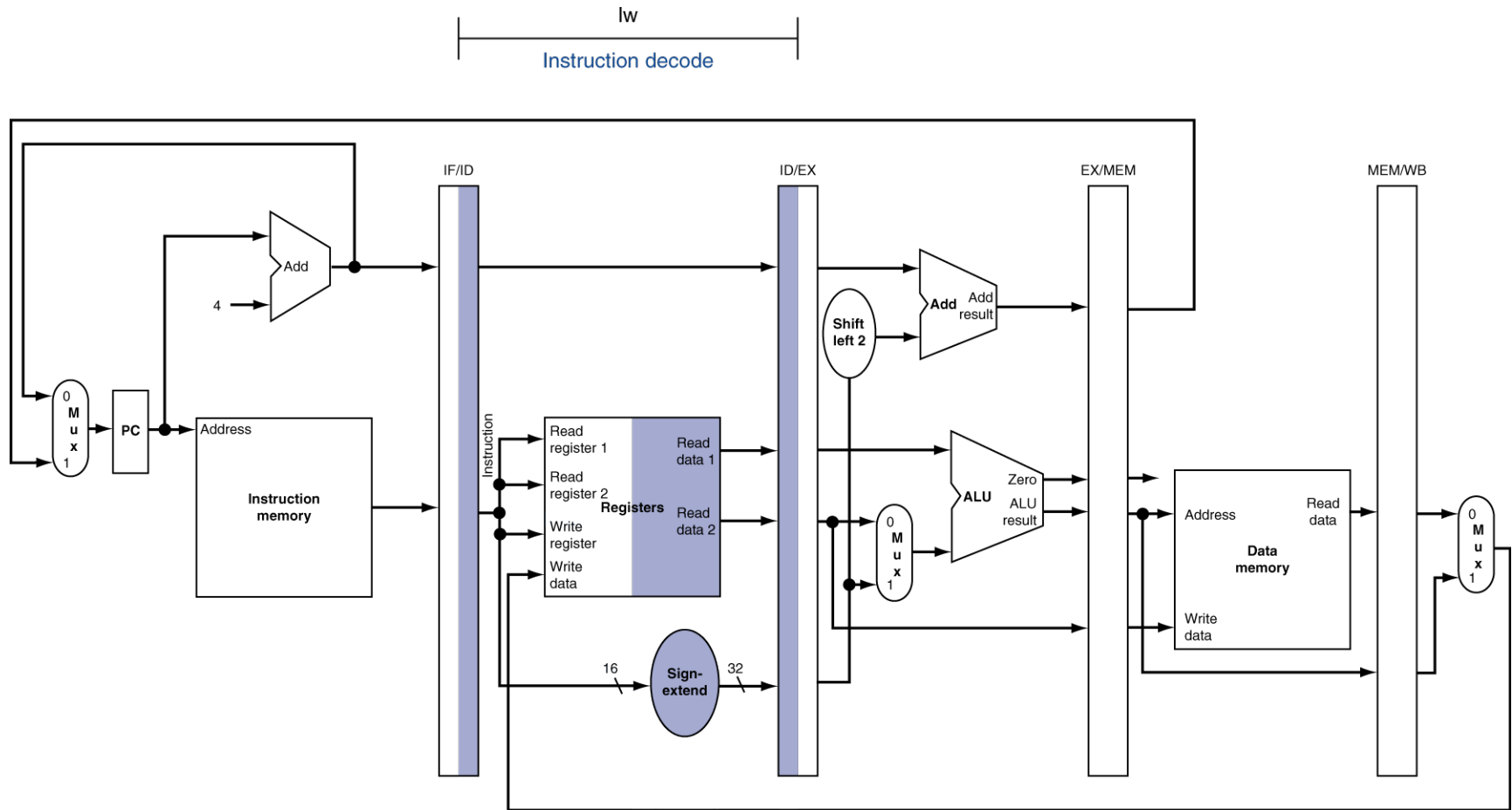
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
 - “Single-clock-cycle” pipeline diagram
 - Shows pipeline usage in a single cycle
 - Highlight resources used
 - c.f. “multi-clock-cycle” diagram
 - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store

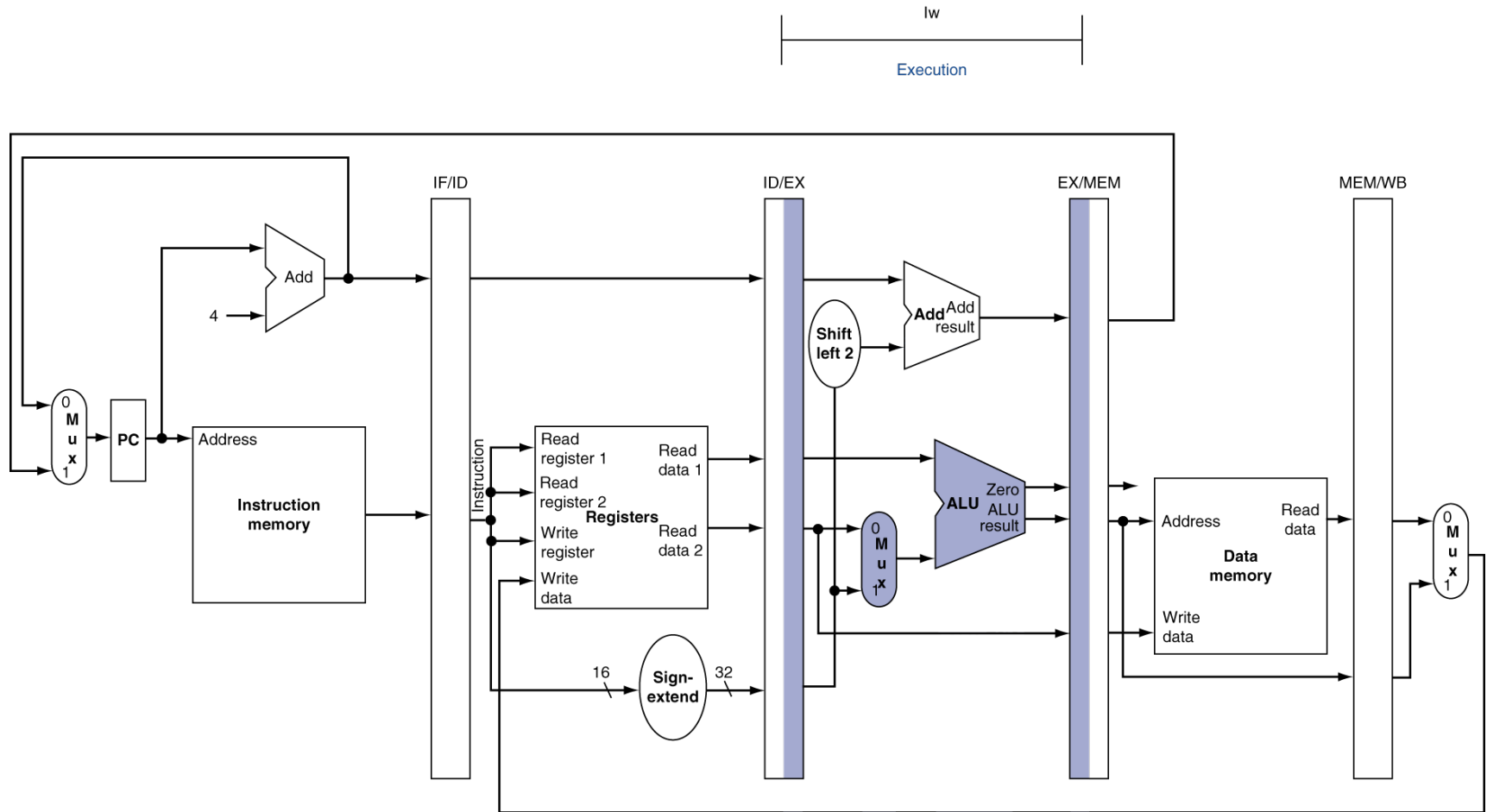
IF for Load, Store, ...



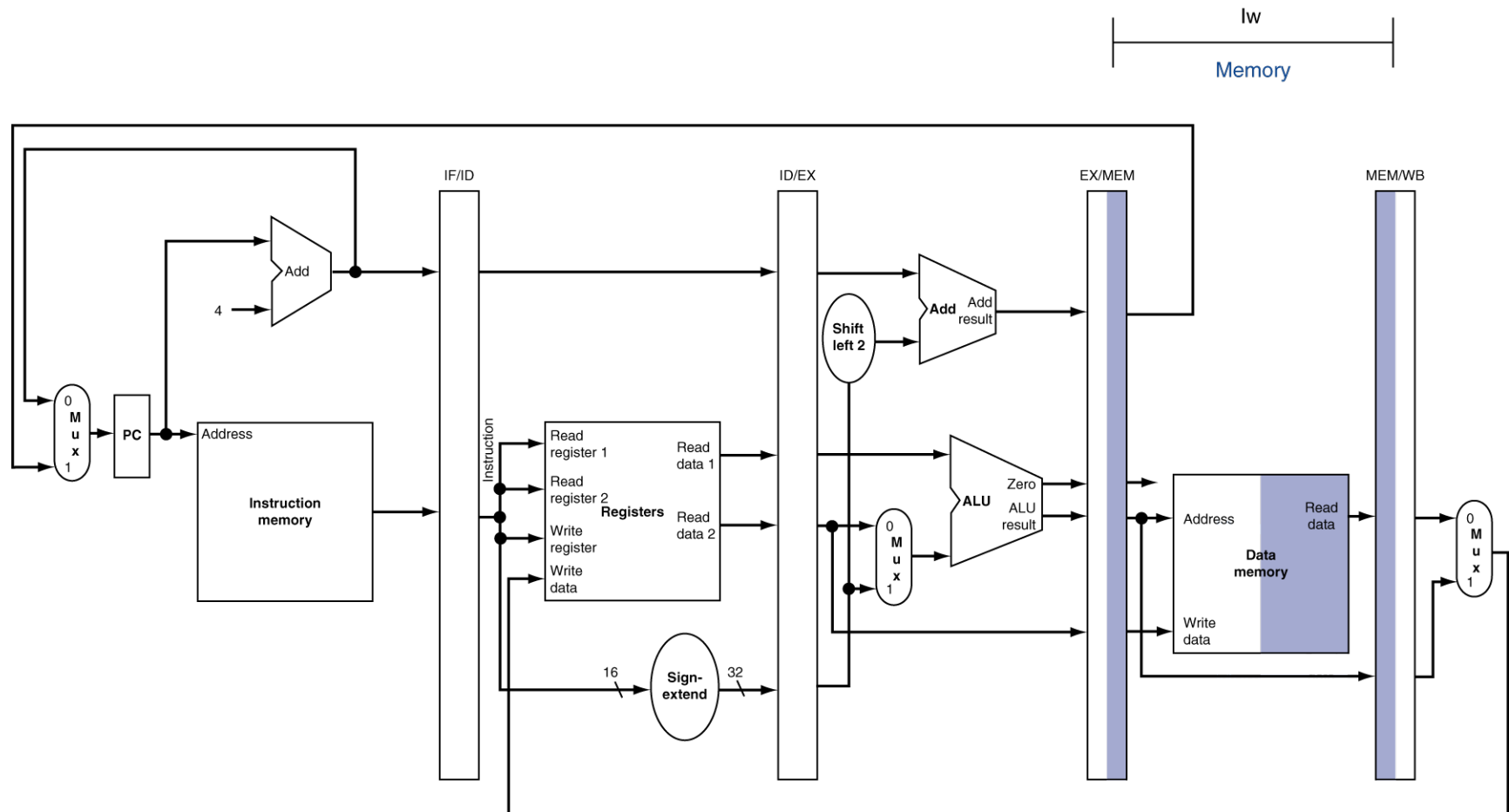
ID for Load, Store, ...



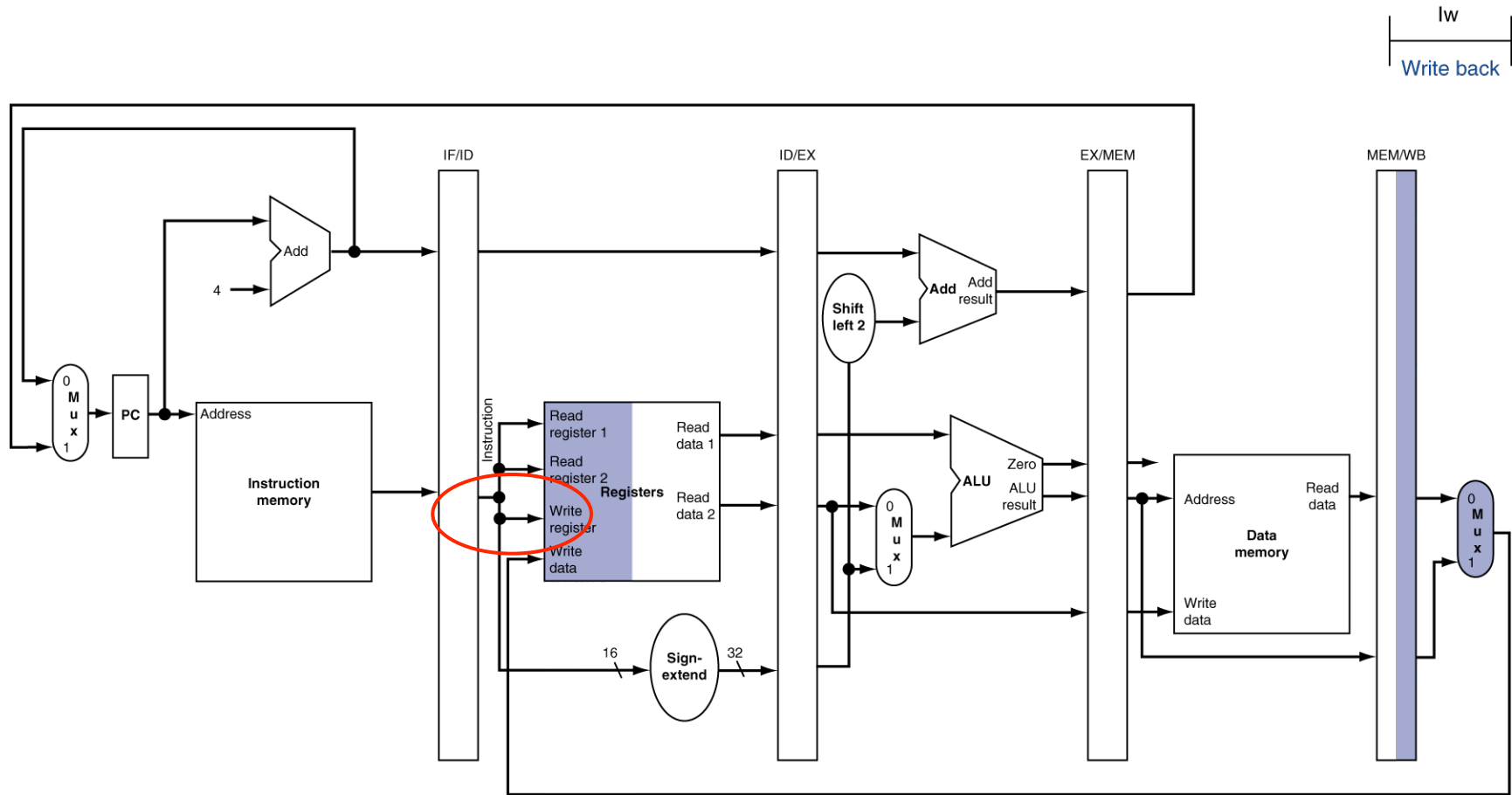
EX for Load



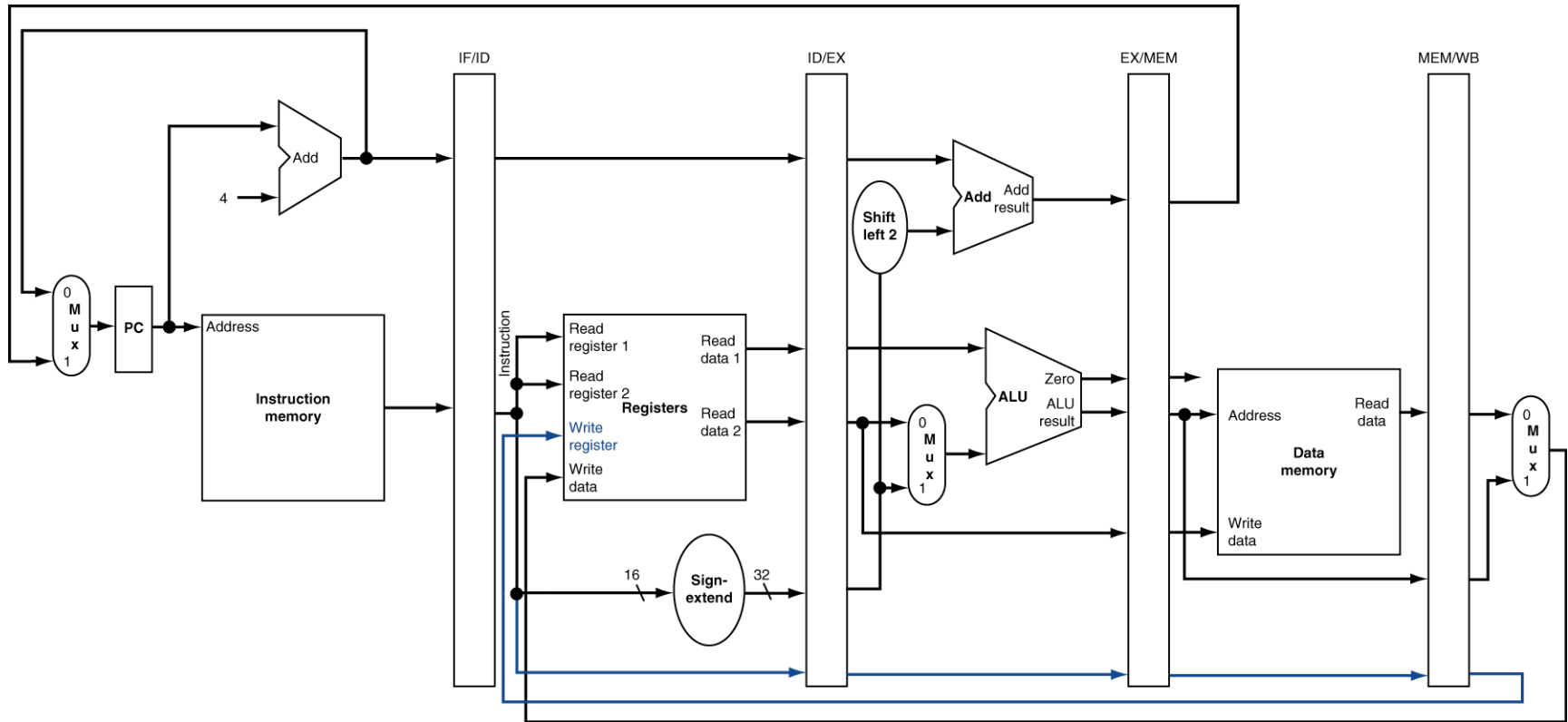
MEM for Load



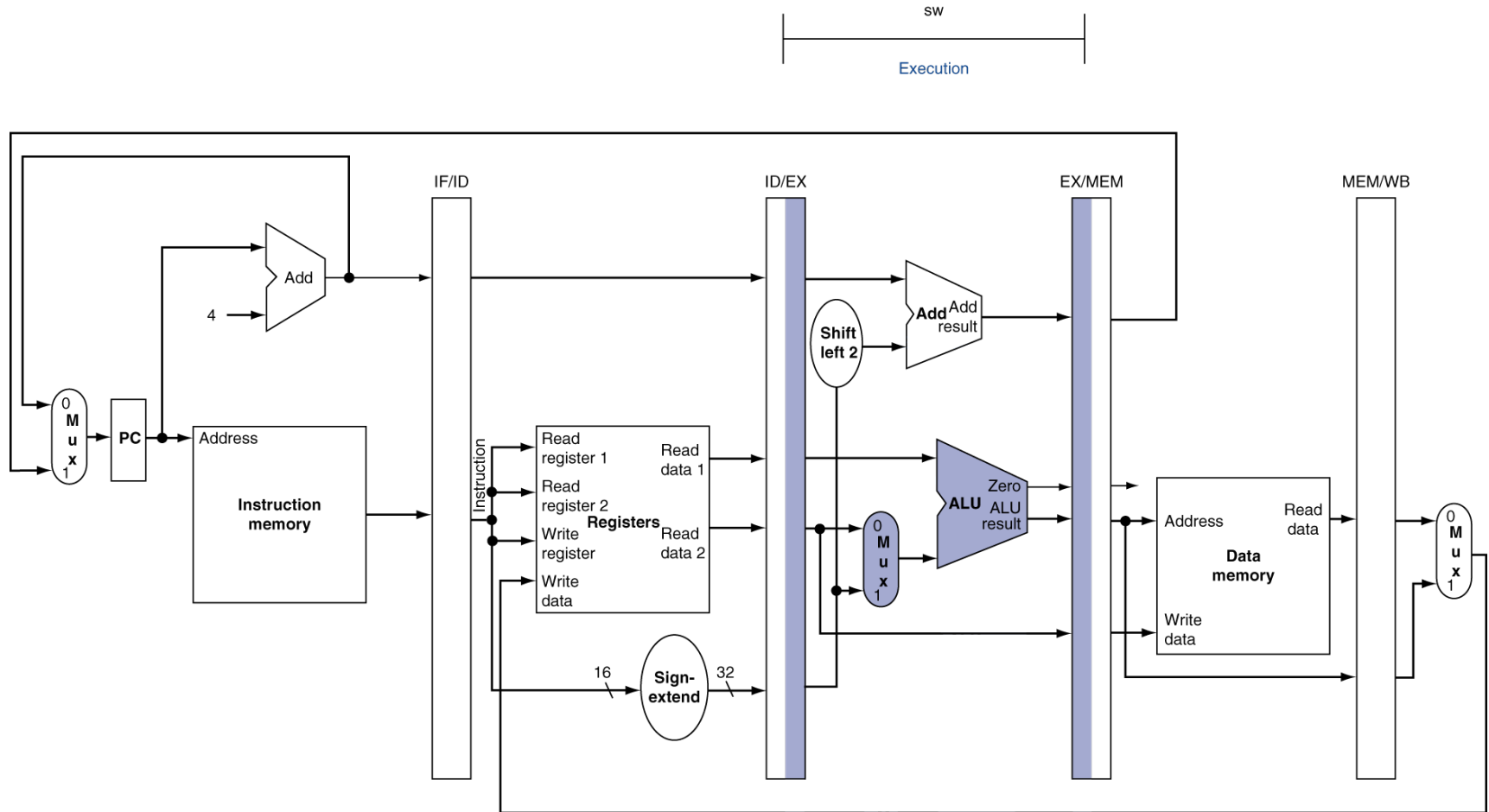
WB for Load



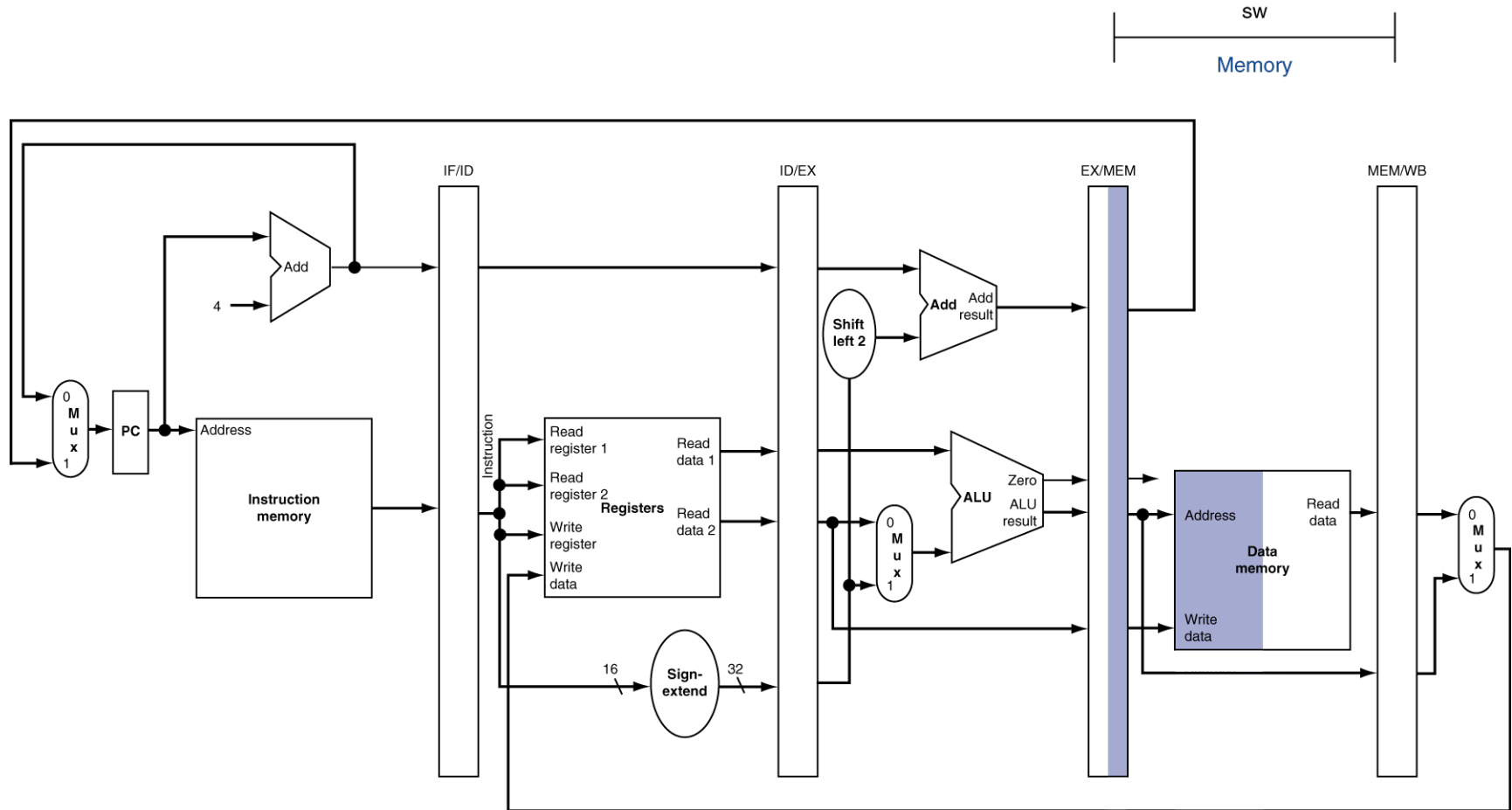
Corrected Datapath for Load



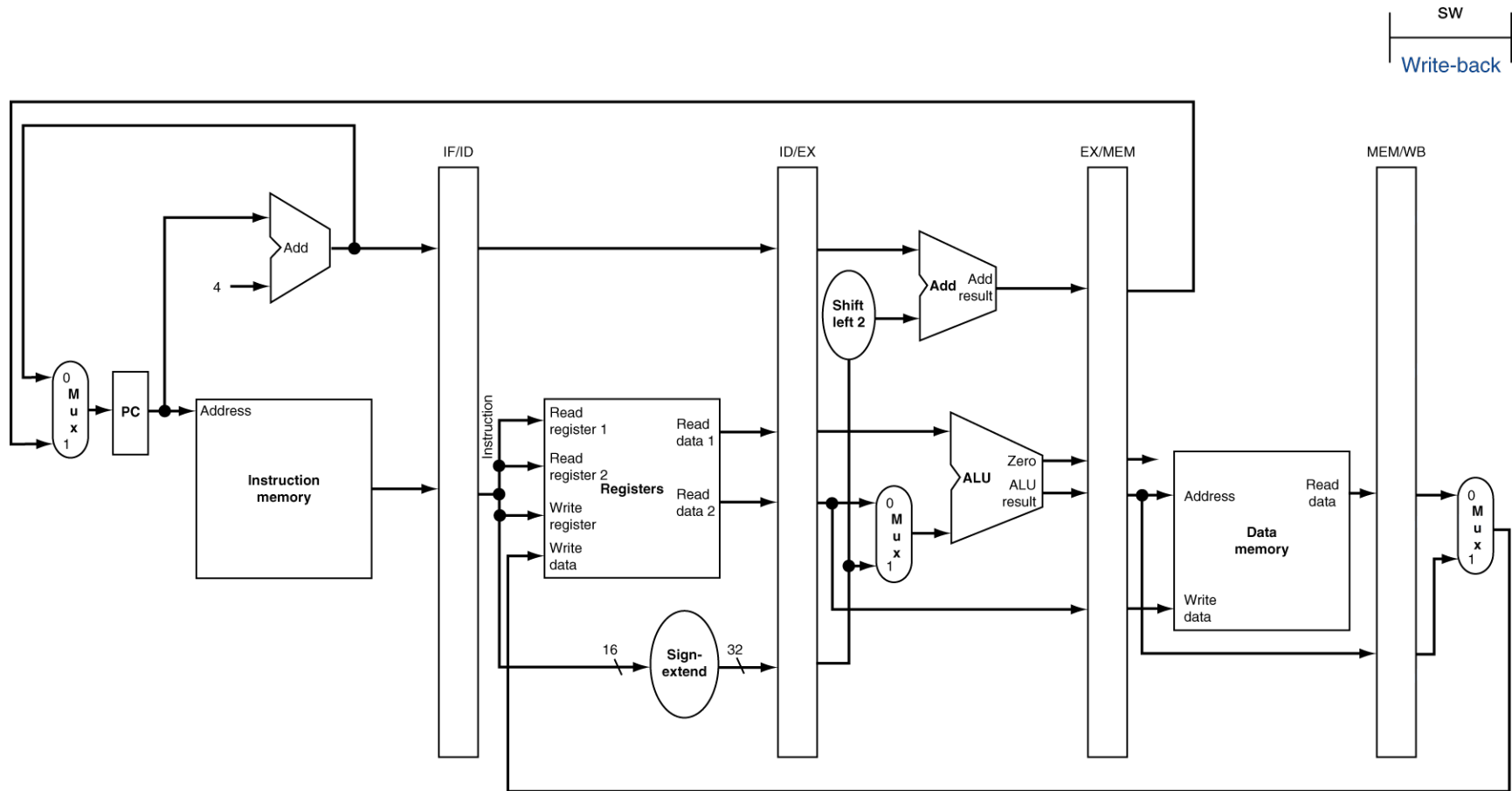
EX for Store



MEM for Store

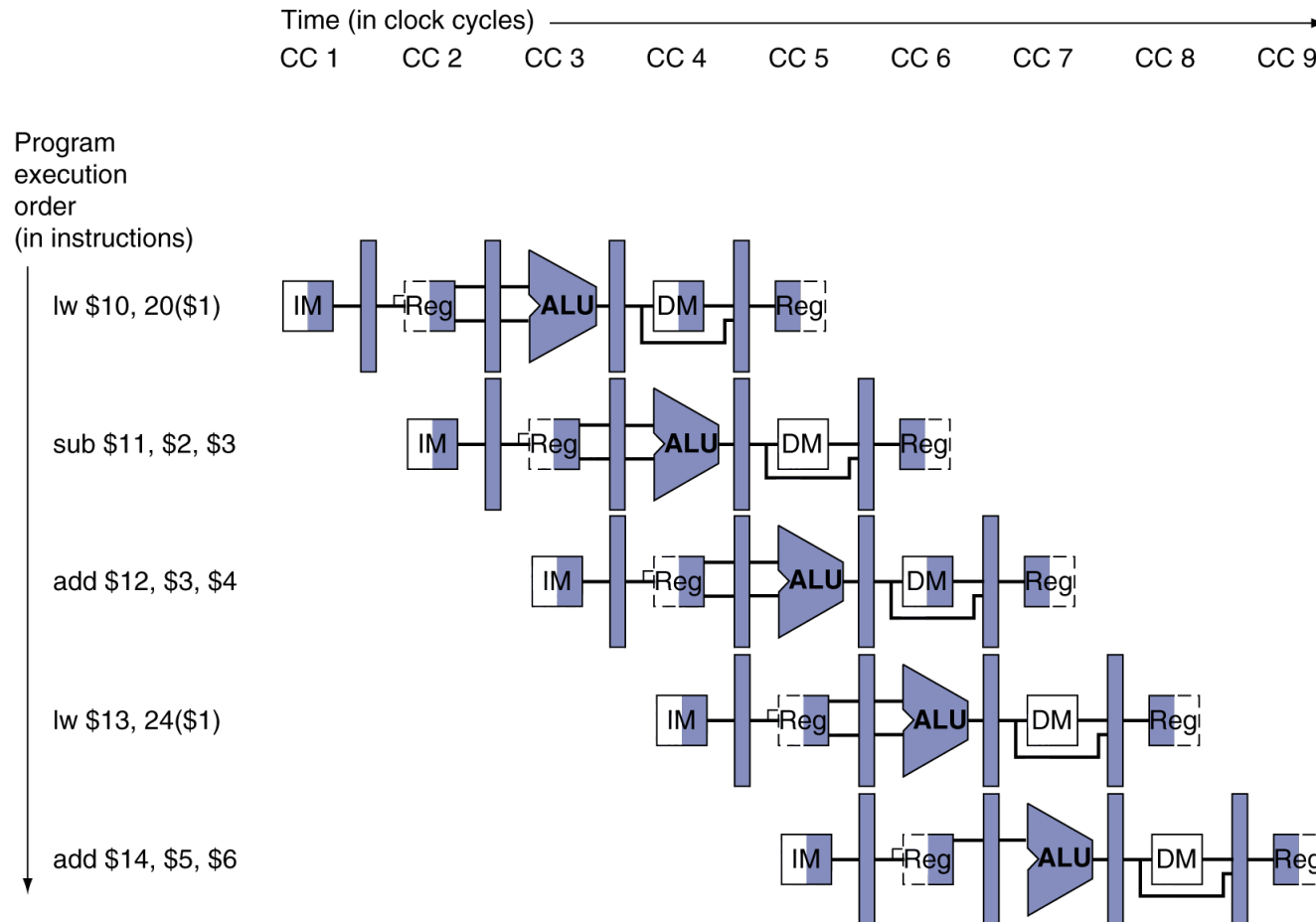


WB for Store



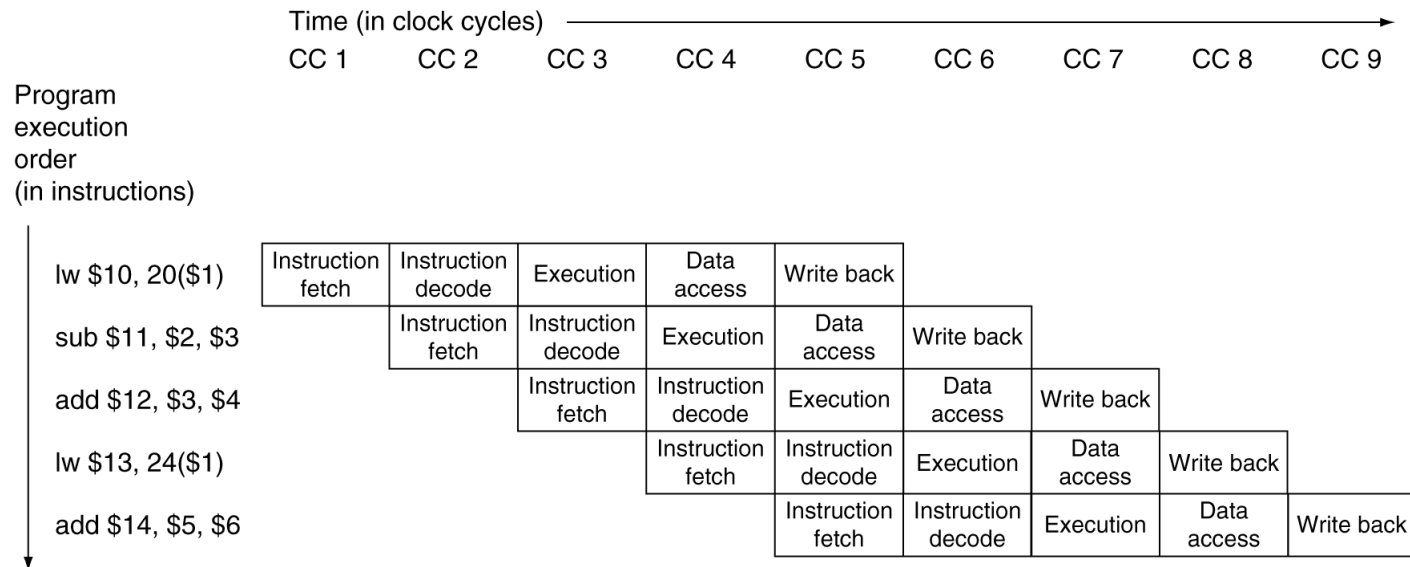
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

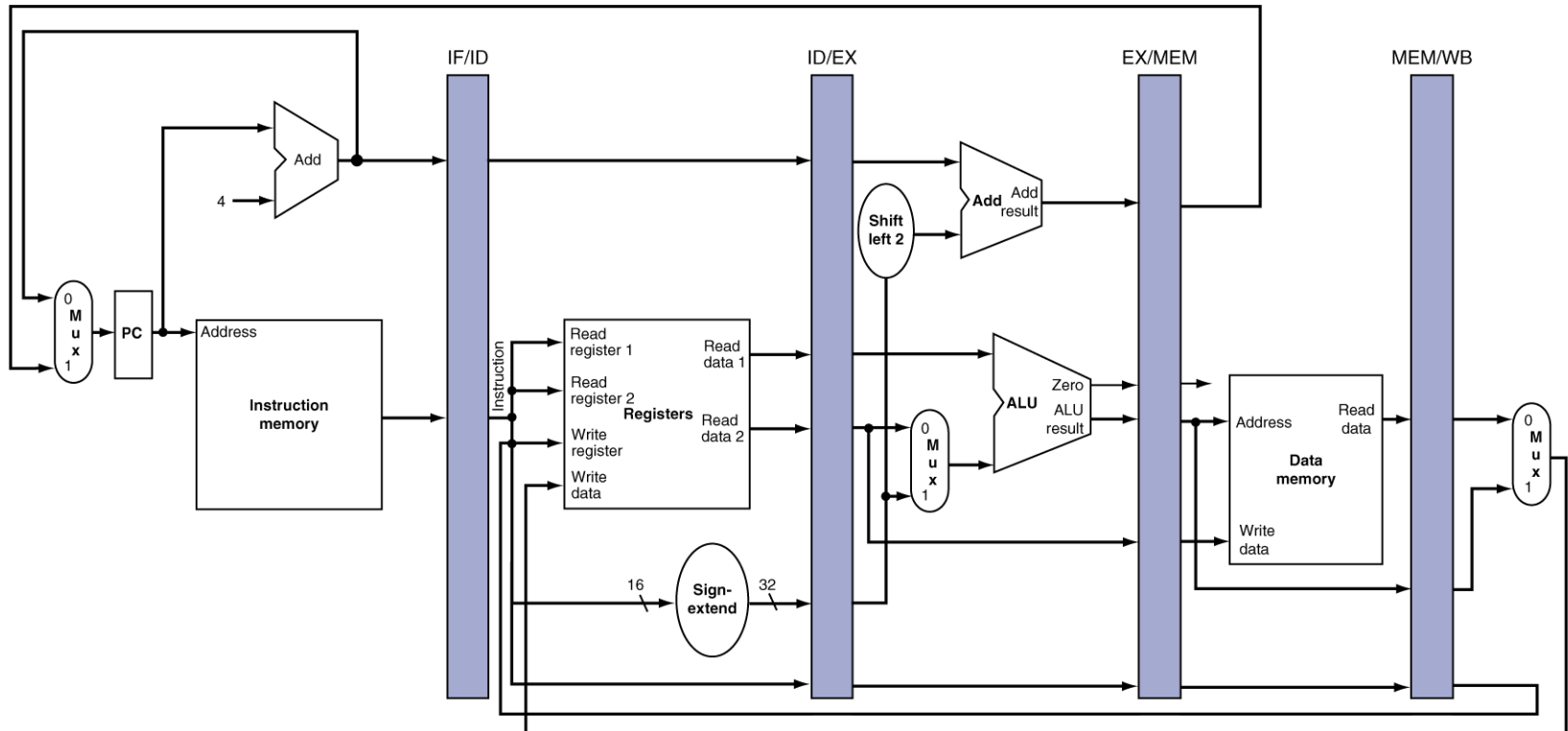
■ Traditional form



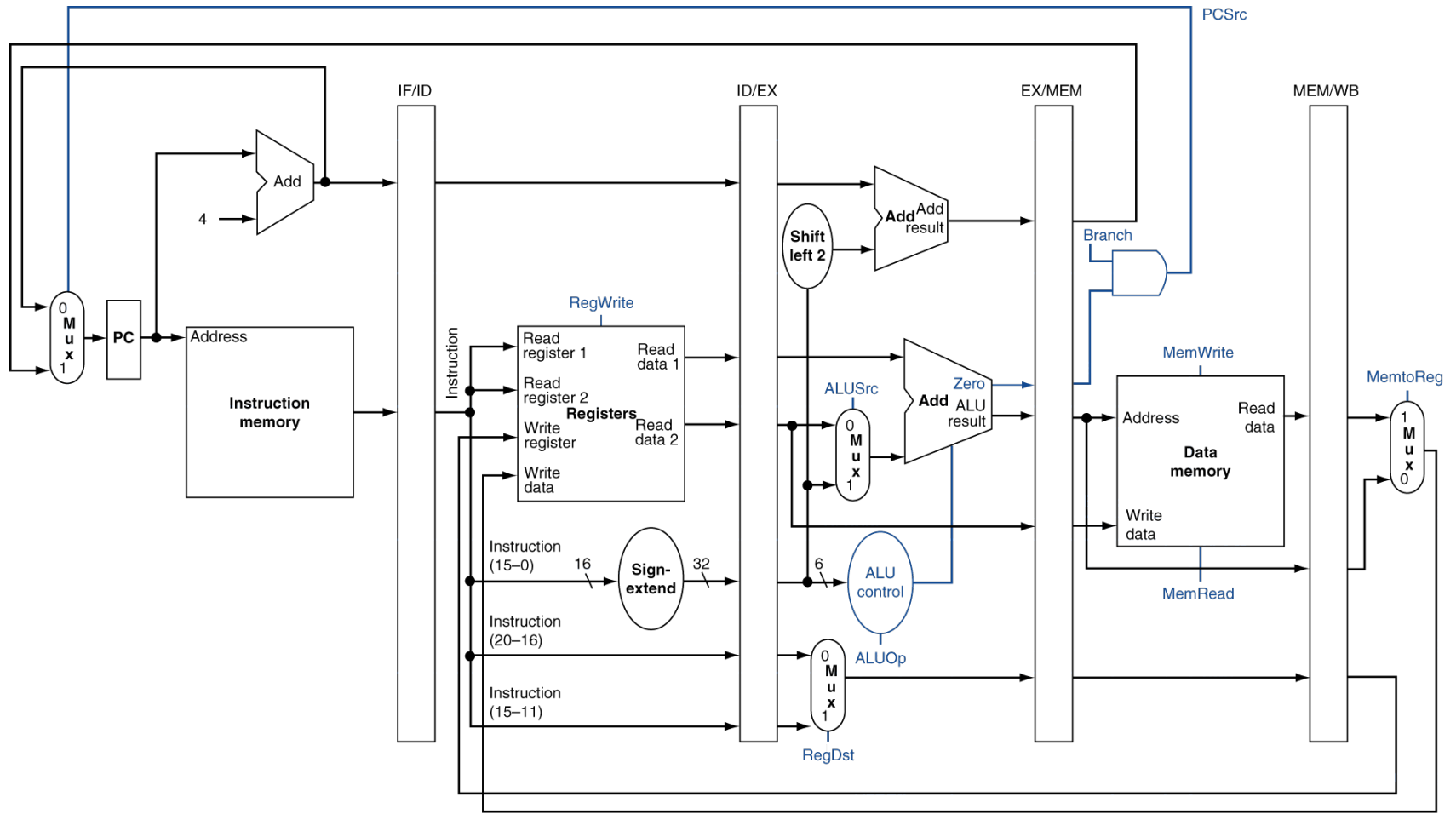
Single-Cycle Pipeline Diagram

■ State of pipeline in a given cycle

add \$14, \$5, \$6	lw \$13, 24 (\$1)	add \$12, \$3, \$4	sub \$11, \$2, \$3	lw \$10, 20(\$1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back

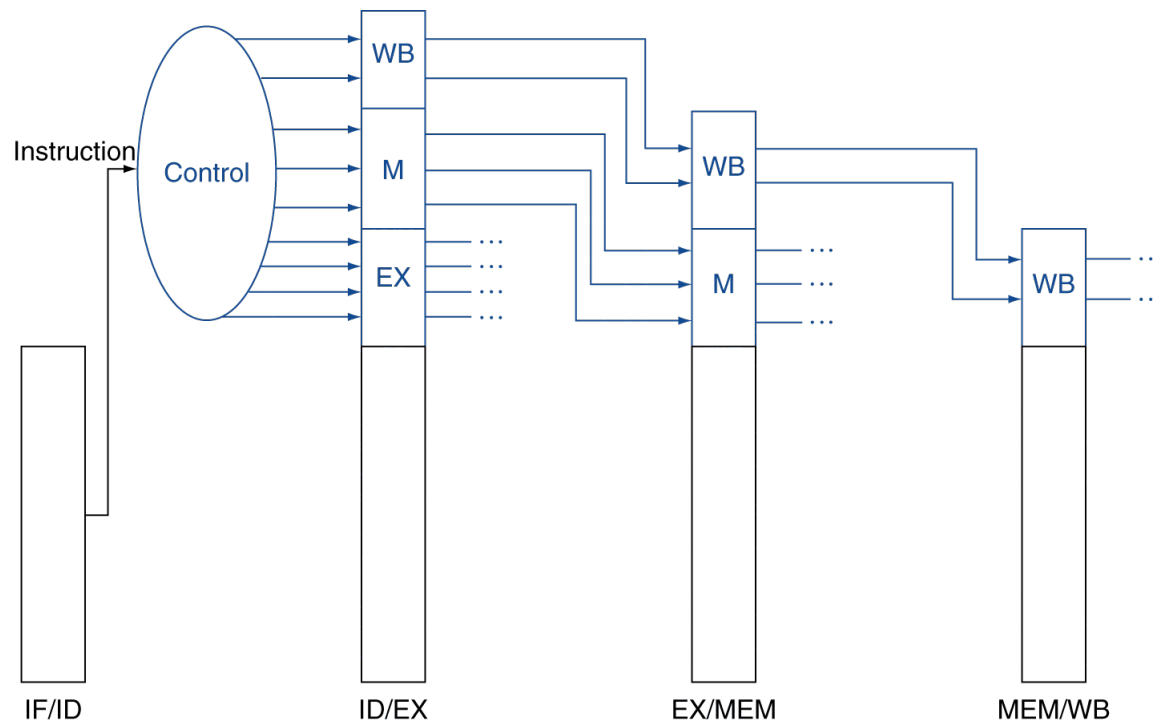


Pipelined Control (Simplified)

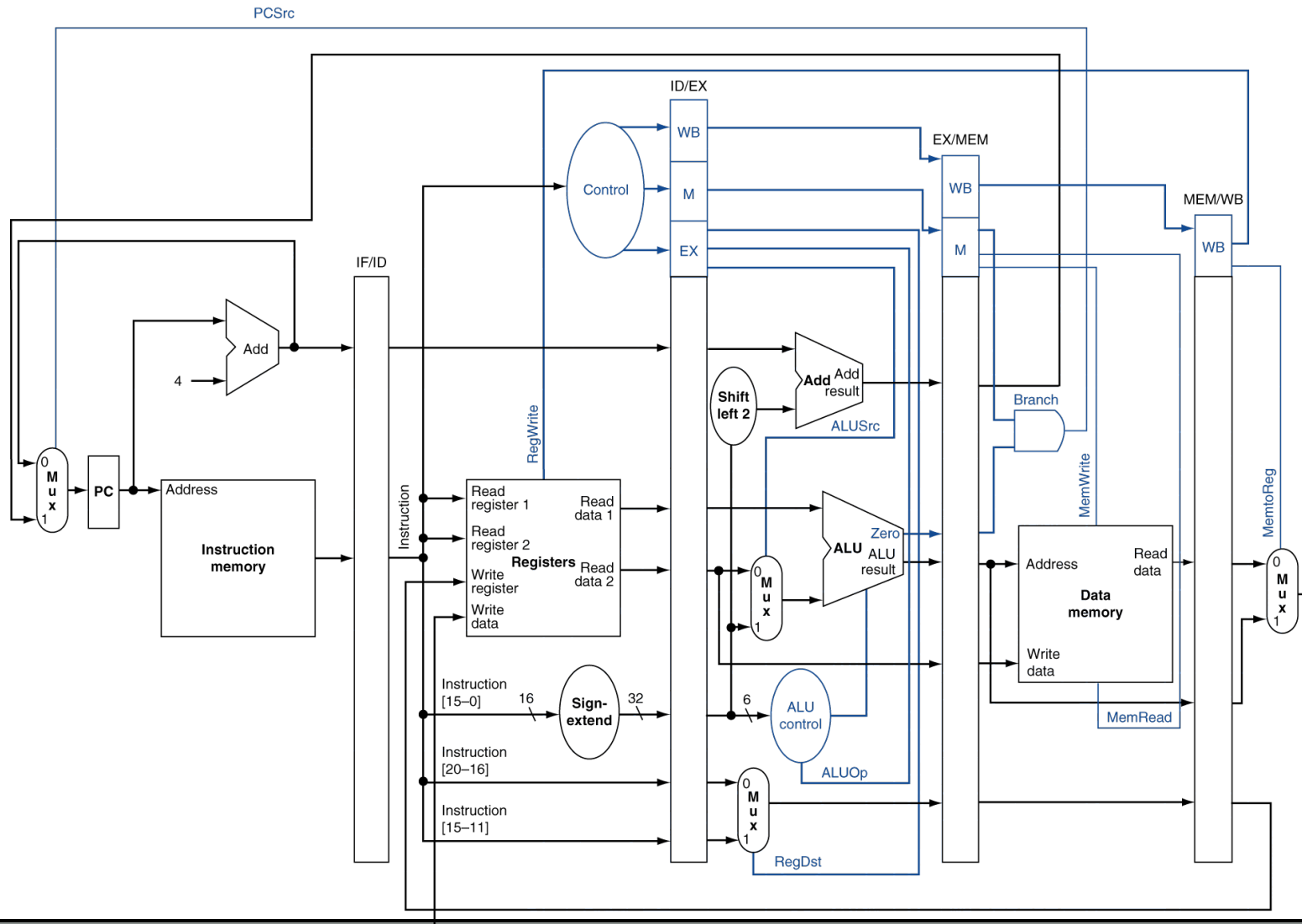


Pipelined Control

- Control signals derived from instruction
 - As in single-cycle implementation



Pipelined Control



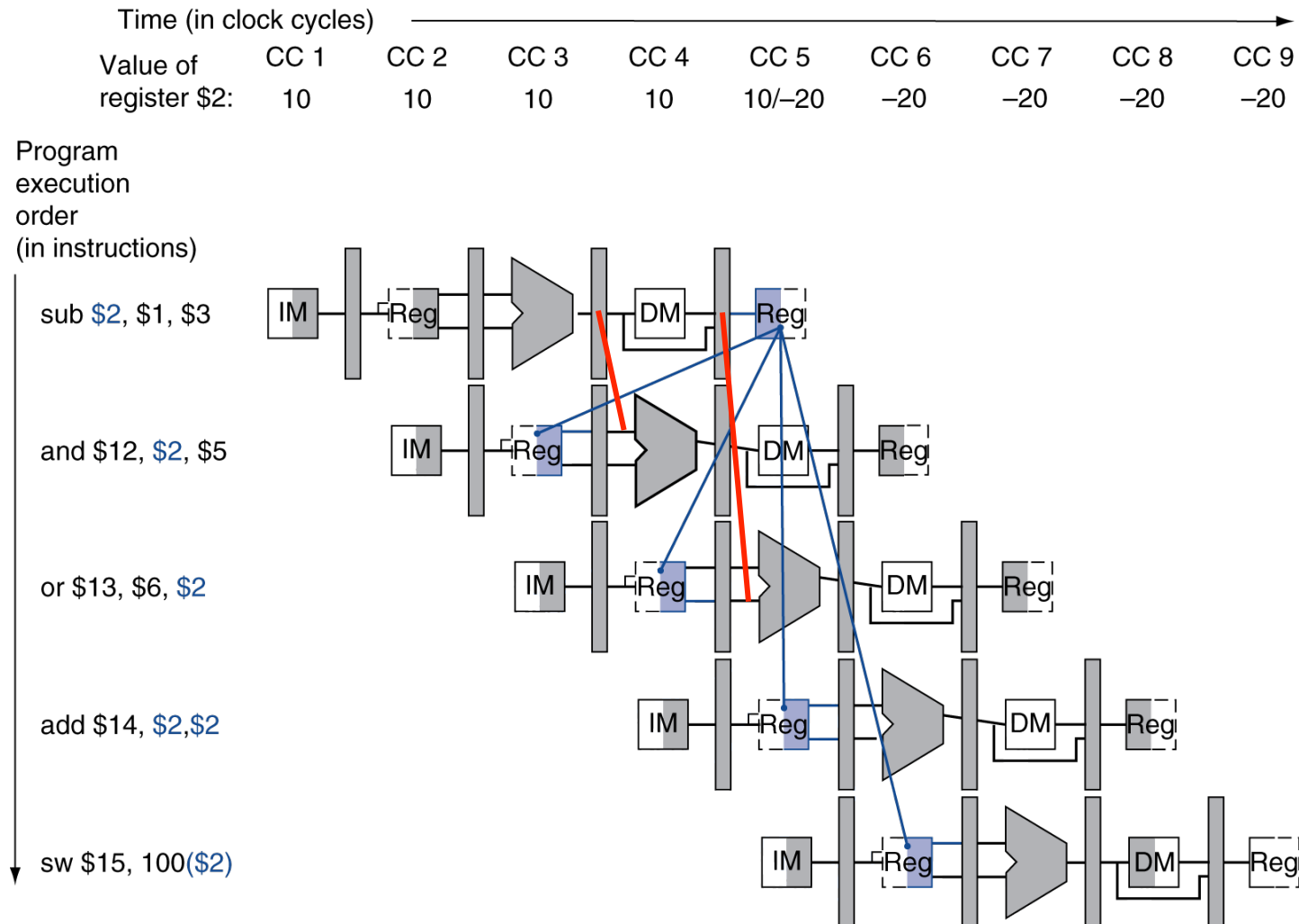
Data Hazards in ALU Instructions

- Consider this sequence:

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

- We can resolve hazards with forwarding
 - How do we detect when to forward?

Dependencies & Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
 - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
 - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
 - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
 - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
 - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
 - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

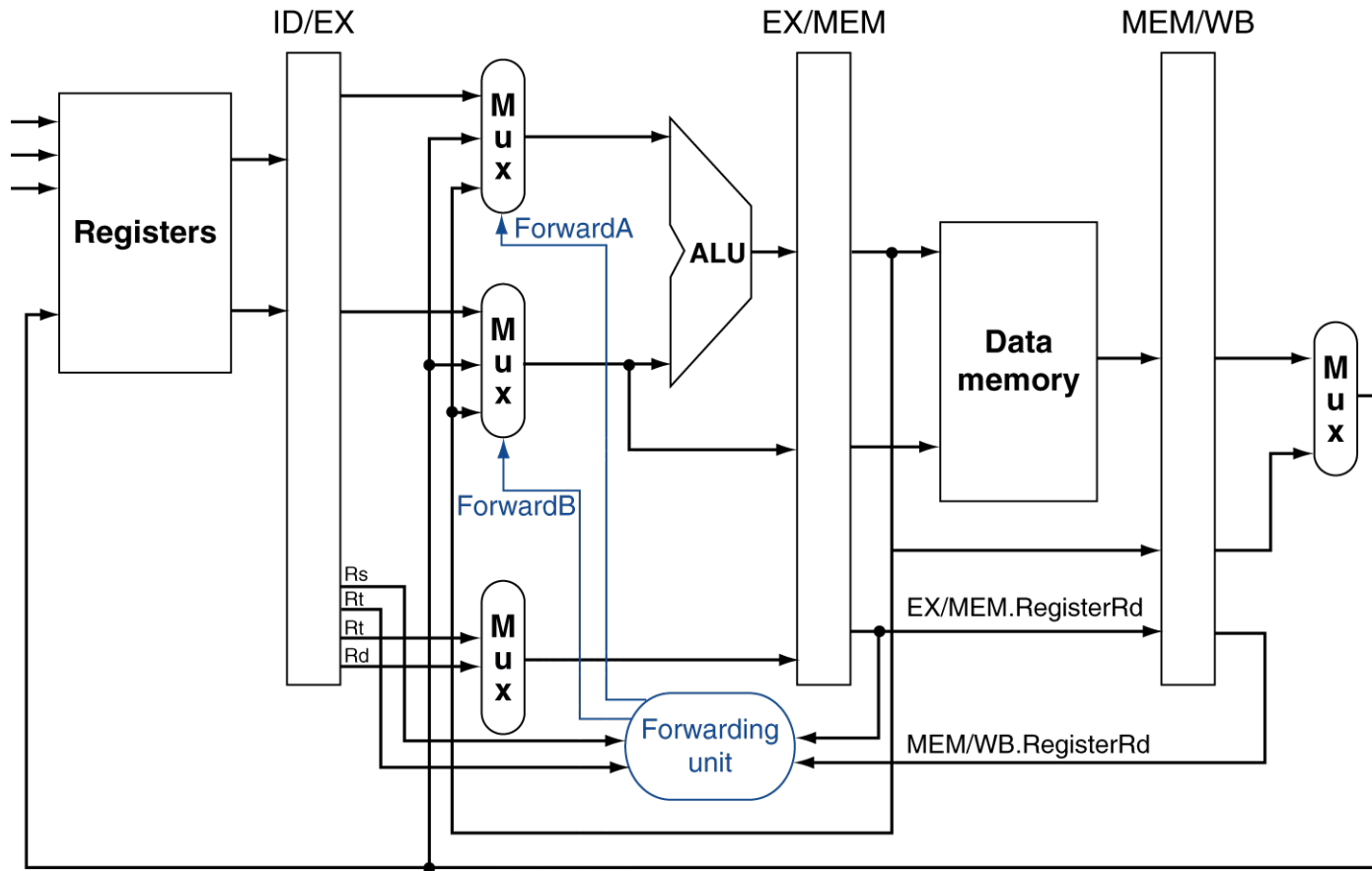
Fwd from
EX/MEM
pipeline reg

Fwd from
MEM/WB
pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
 - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not \$zero
 - EX/MEM.RegisterRd \neq 0,
MEM/WB.RegisterRd \neq 0

Forwarding Paths



b. With forwarding

Forwarding Conditions

- EX hazard

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

ForwardA = 10

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))

ForwardB = 10

- MEM hazard

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd \neq 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

ForwardA = 01

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd \neq 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

ForwardB = 01

Double Data Hazard

- Consider the sequence:
 - add \$1, \$1, \$2
 - add \$1, \$1, \$3
 - add \$1, \$1, \$4
- Both hazards occur
 - Want to use the most recent
- Revise MEM hazard condition
 - Only fwd if EX hazard condition isn't true

Revised Forwarding Condition

- MEM hazard

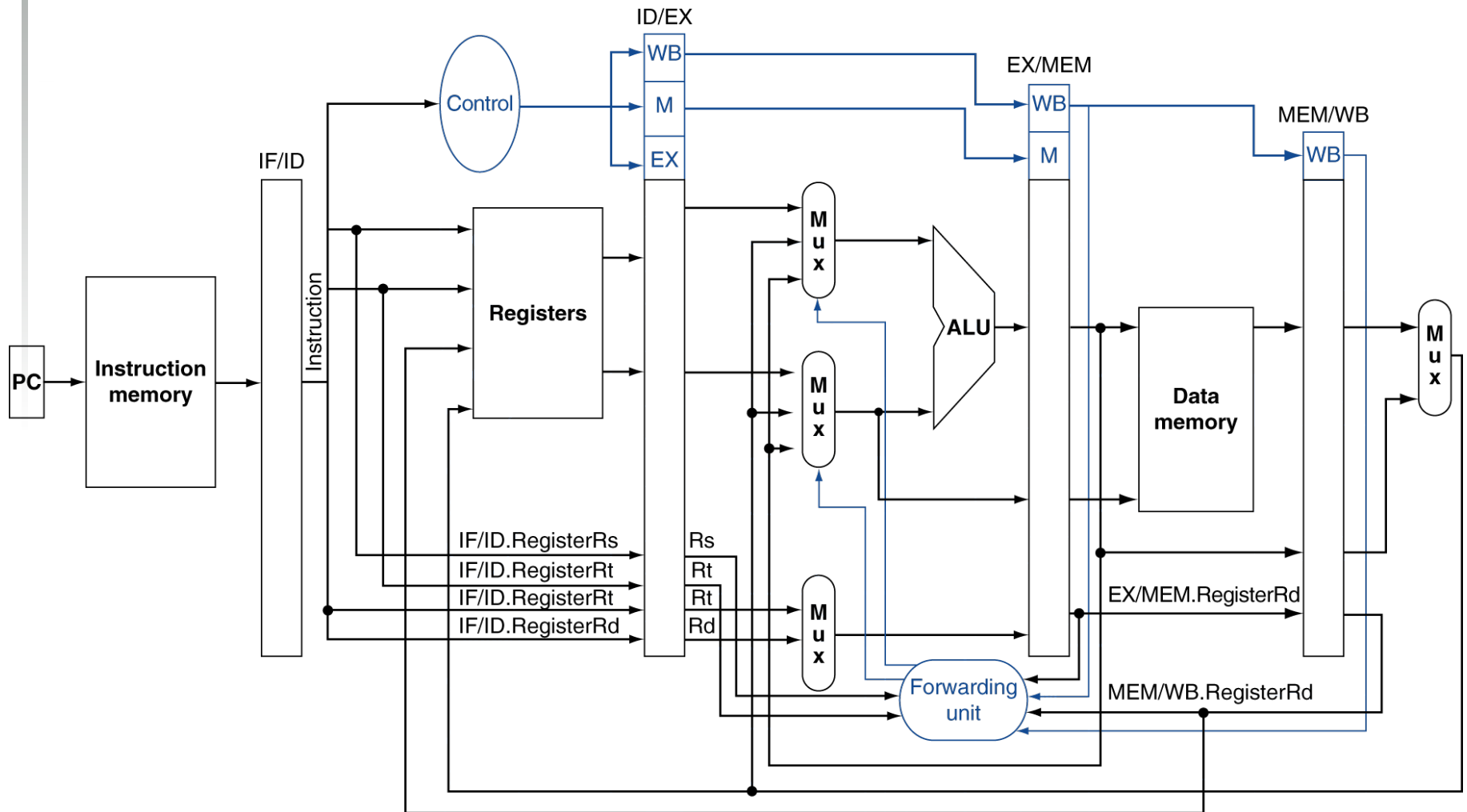
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd \neq 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

ForwardA = 01

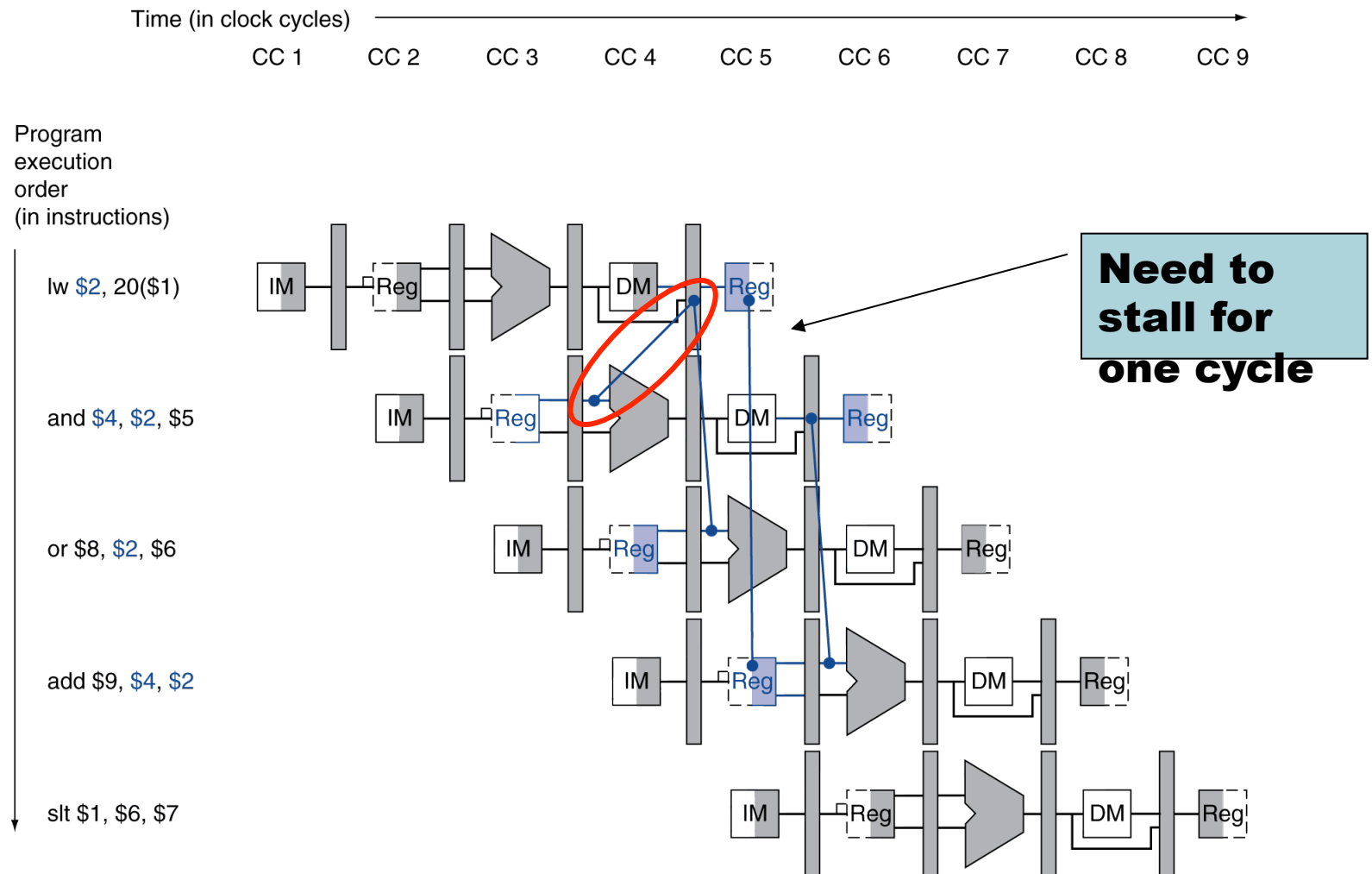
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd \neq 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

ForwardB = 01

Datapath with Forwarding



Load-Use Data Hazard



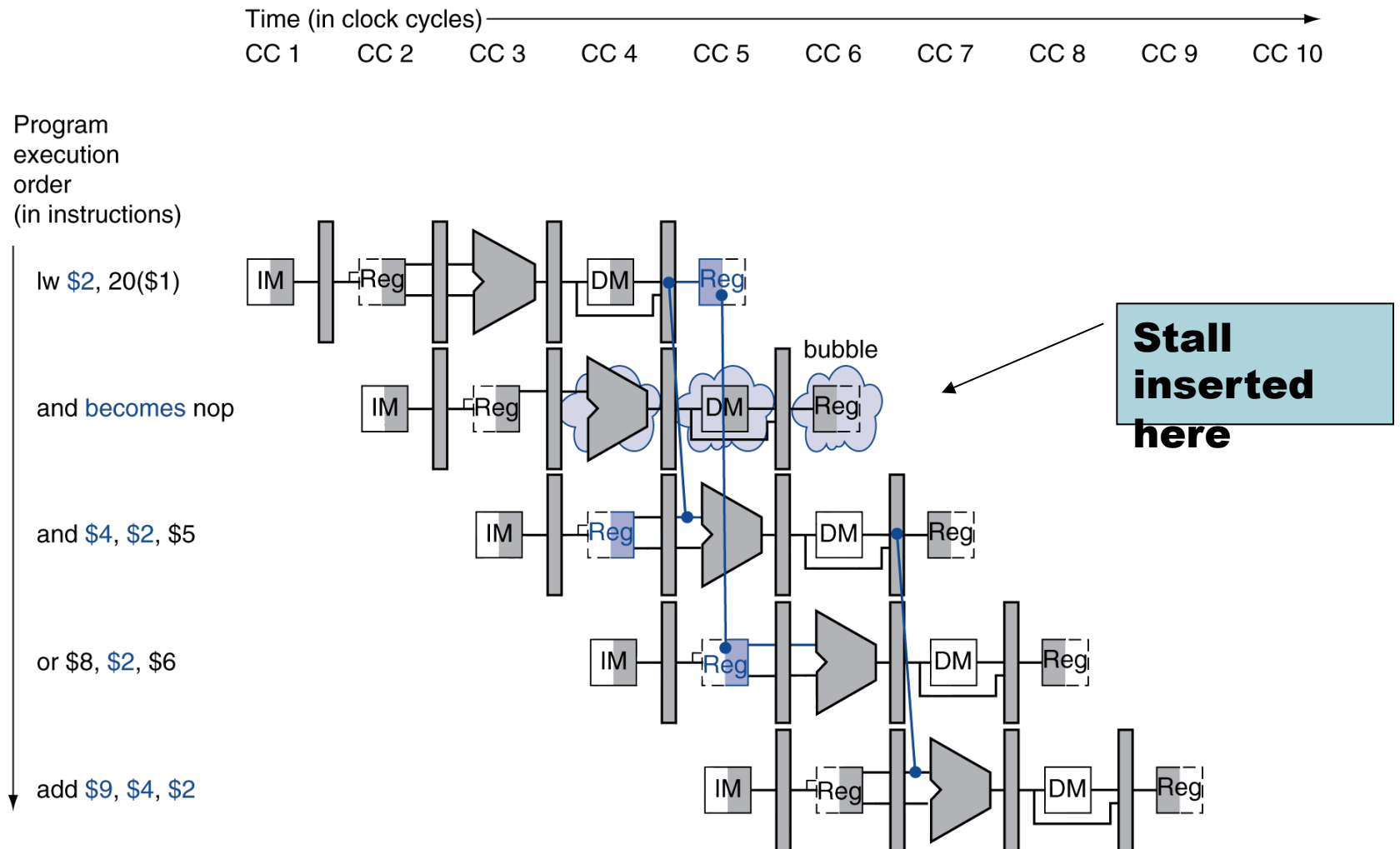
Load-Use Hazard Detection

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

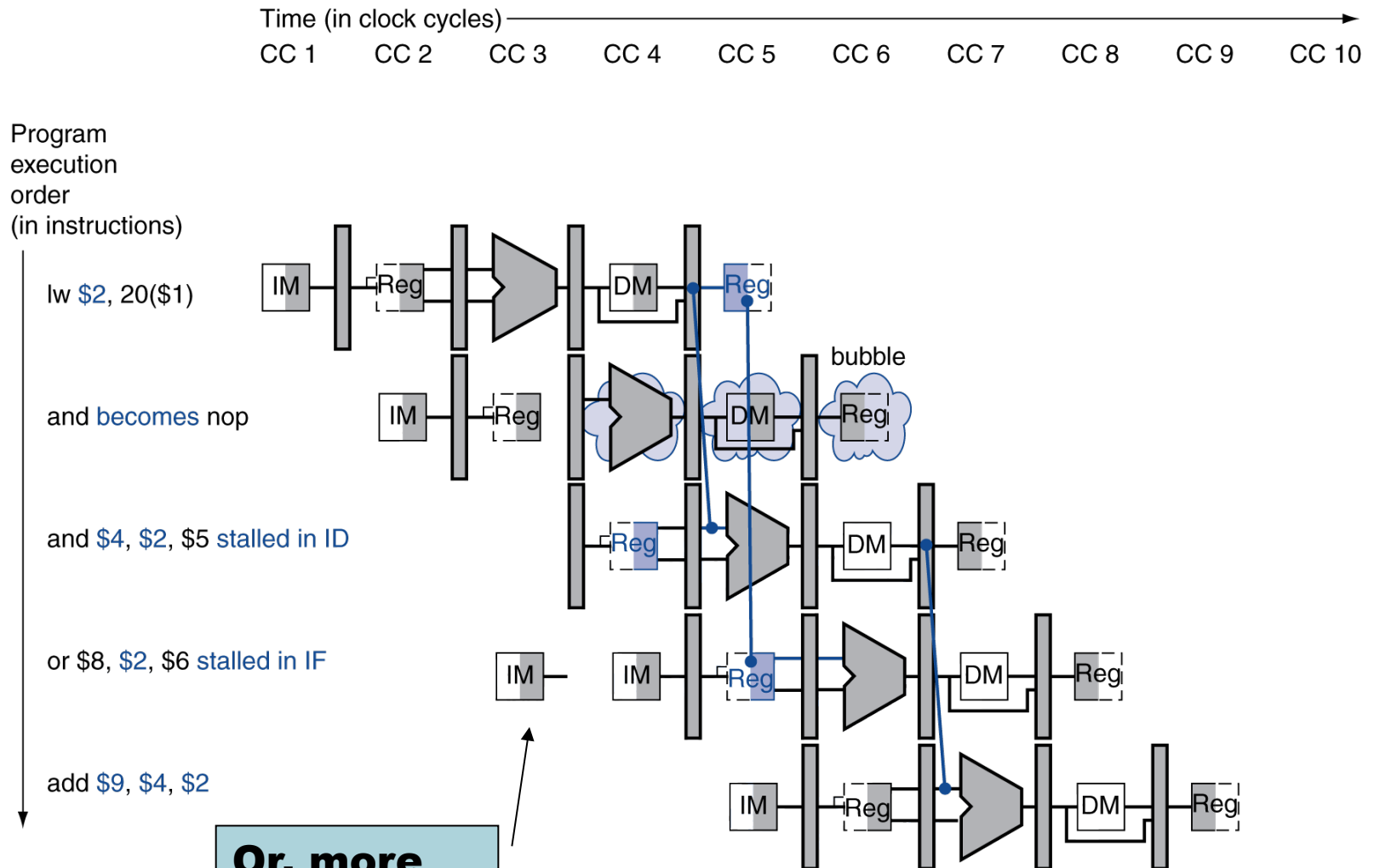
How to Stall the Pipeline

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

Stall/Bubble in the Pipeline

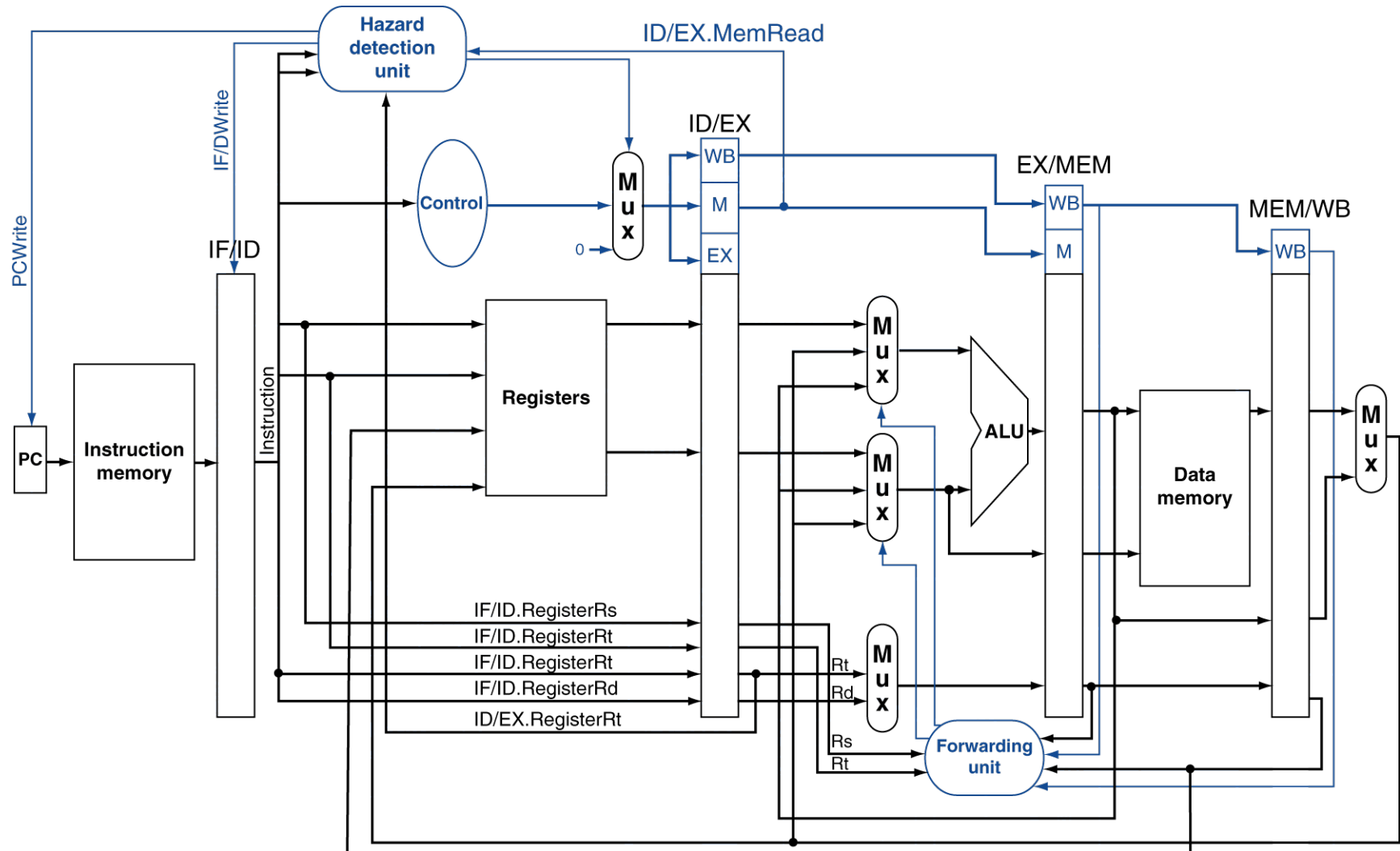


Stall/Bubble in the Pipeline



Or, more accurately

Datapath with Hazard Detection



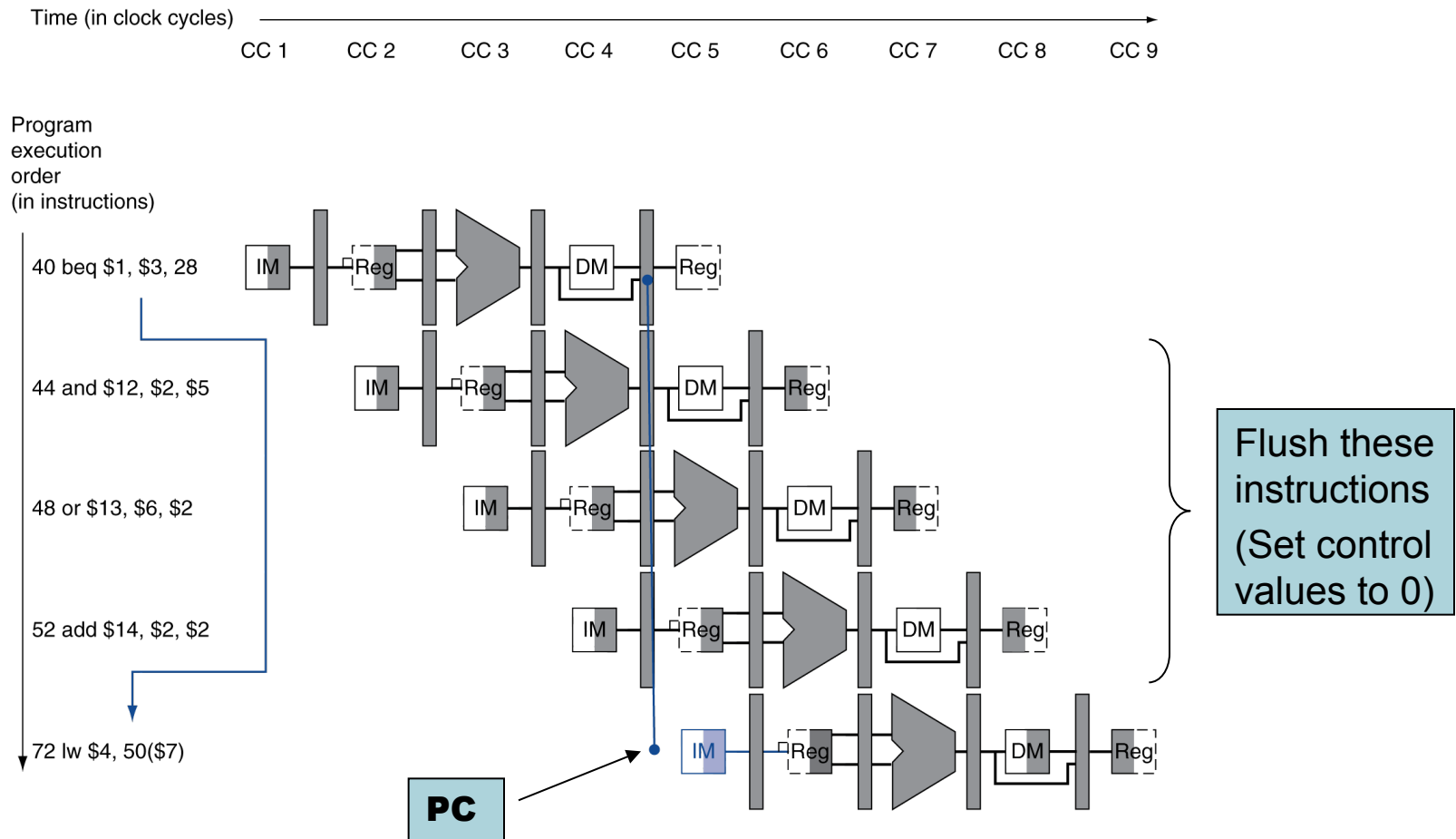
Stalls and Performance

The BIG Picture

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

Branch Hazards

- If branch outcome determined in MEM

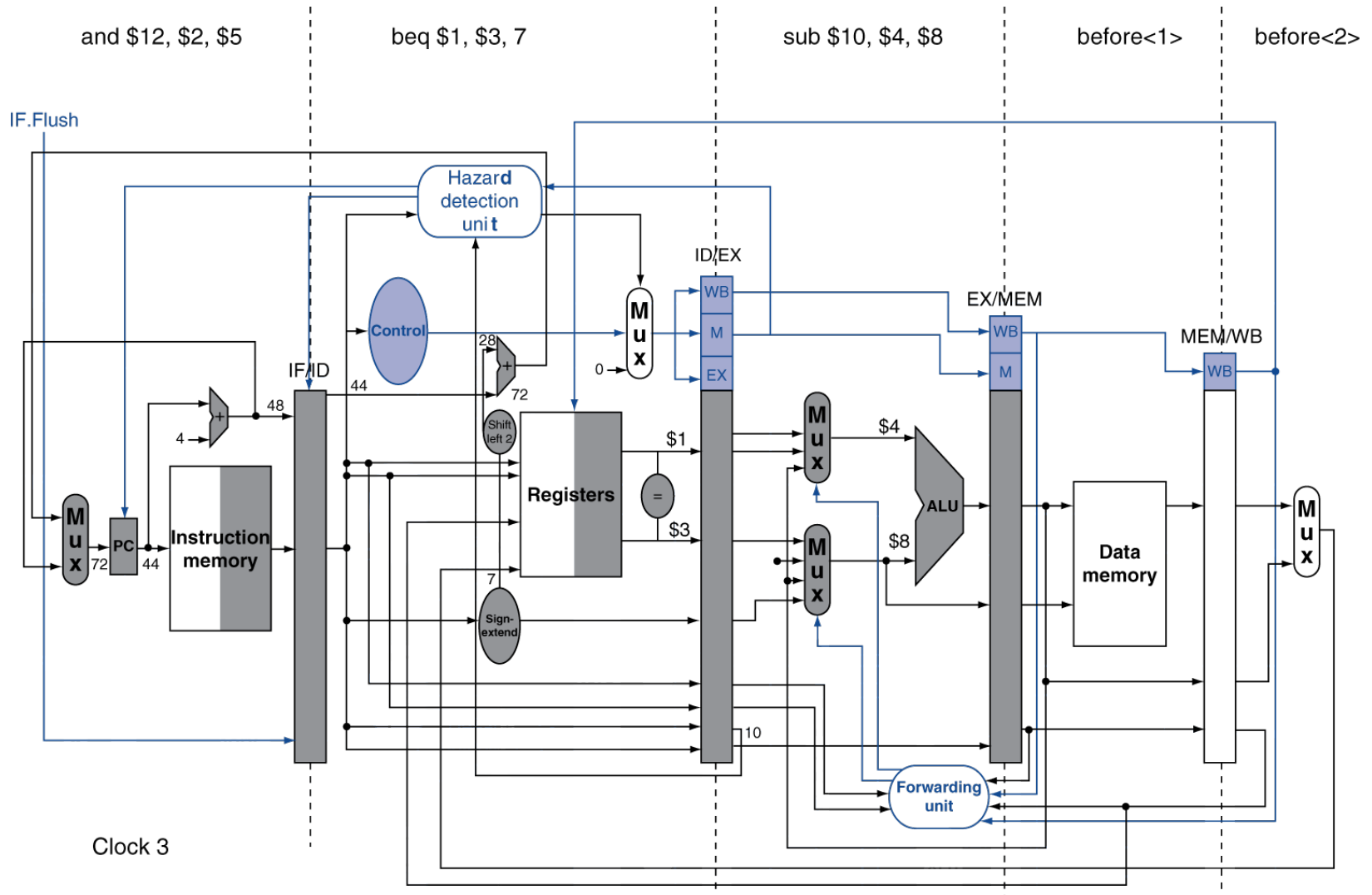


Reducing Branch Delay

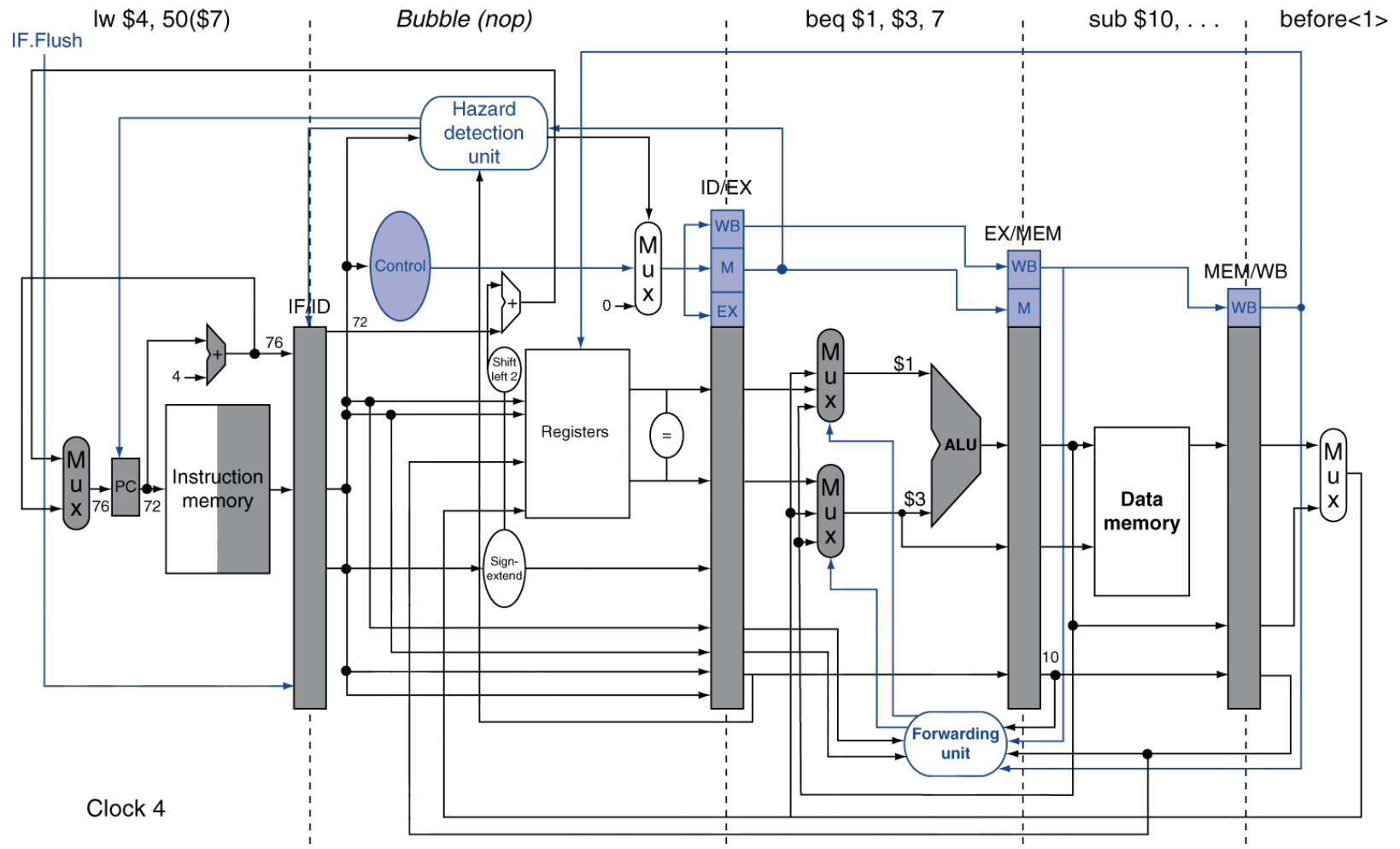
- Move hardware to determine outcome to ID stage
 - Target address adder
 - Register comparator
- Example: branch taken

```
36:  sub    $10, $4, $8
40:  beq    $1,  $3, 7
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)
```

Example: Branch Taken

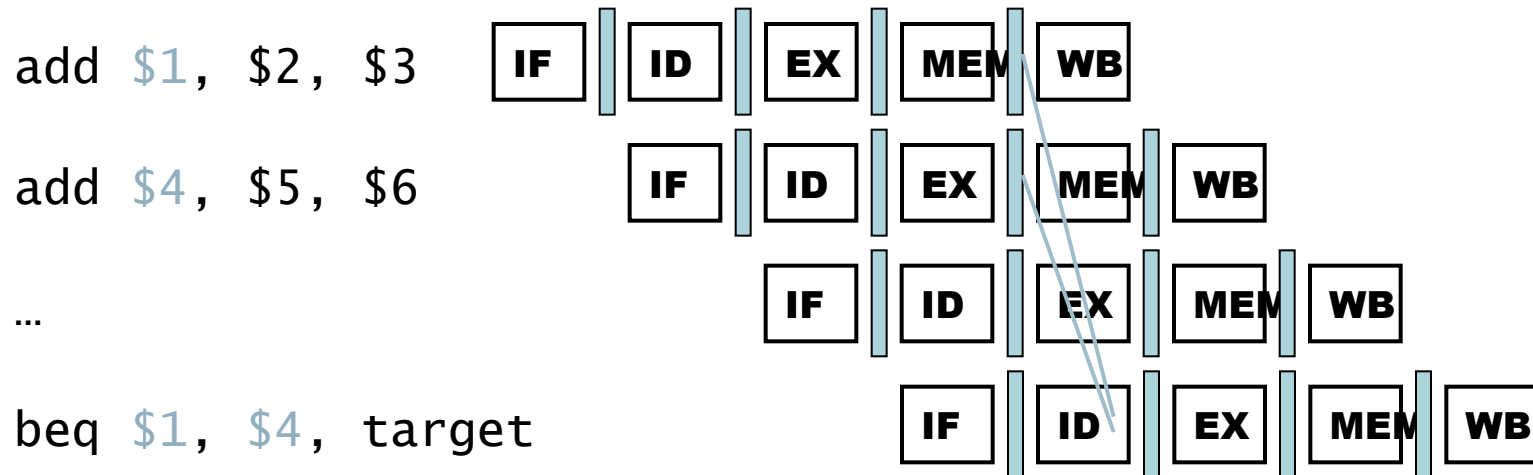


Example: Branch Taken



Data Hazards for Branches

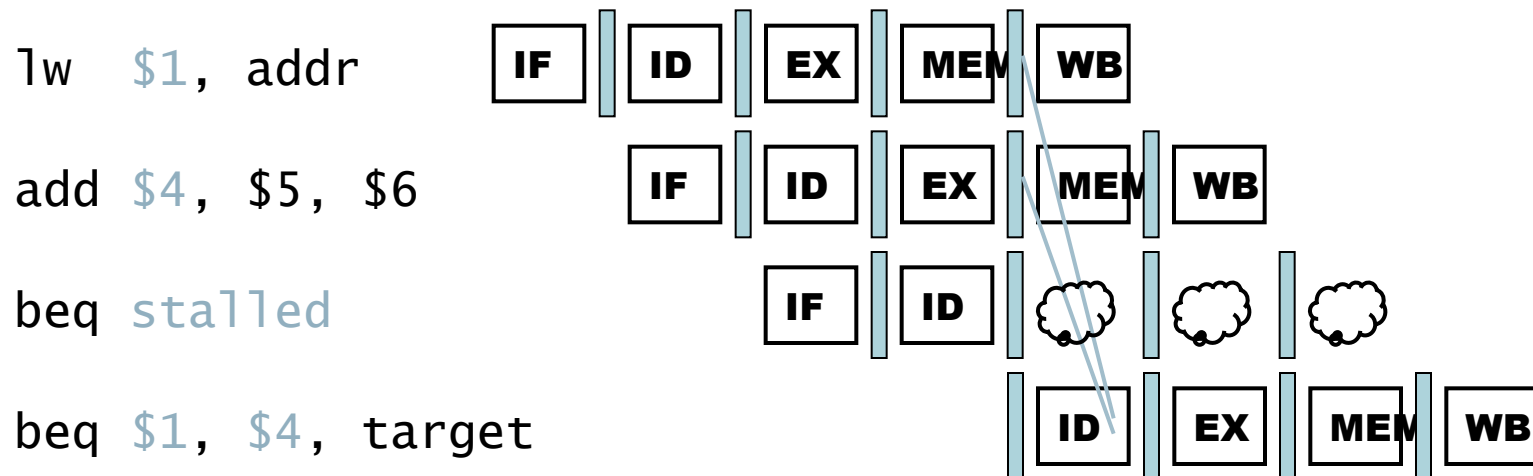
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction



- **Can resolve using forwarding**

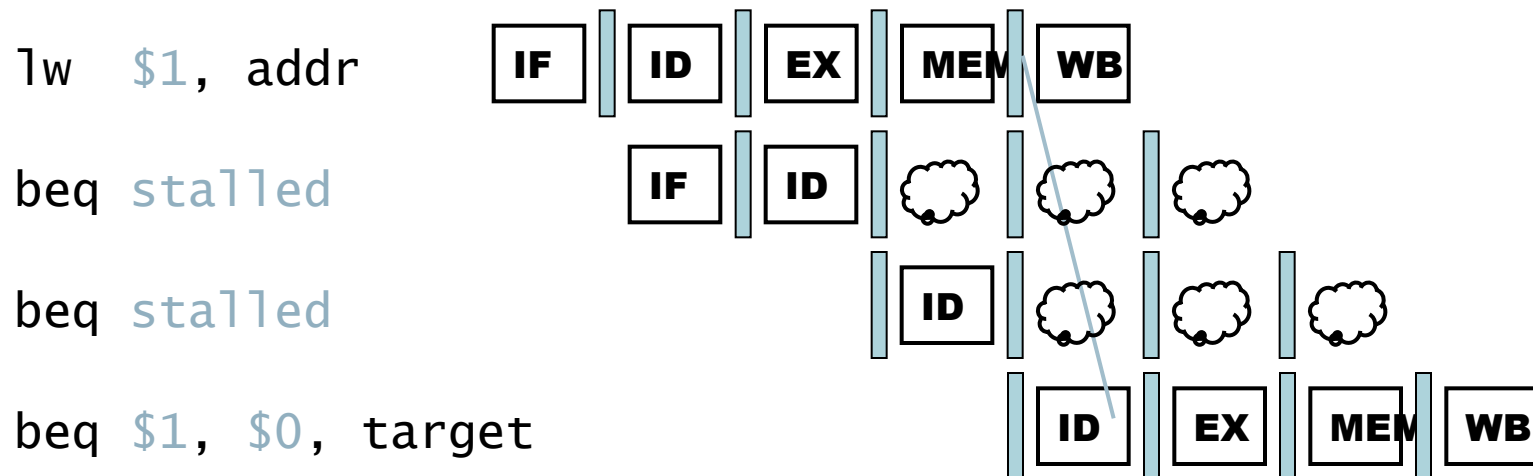
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
 - Need 1 stall cycle



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
 - Need 2 stall cycles

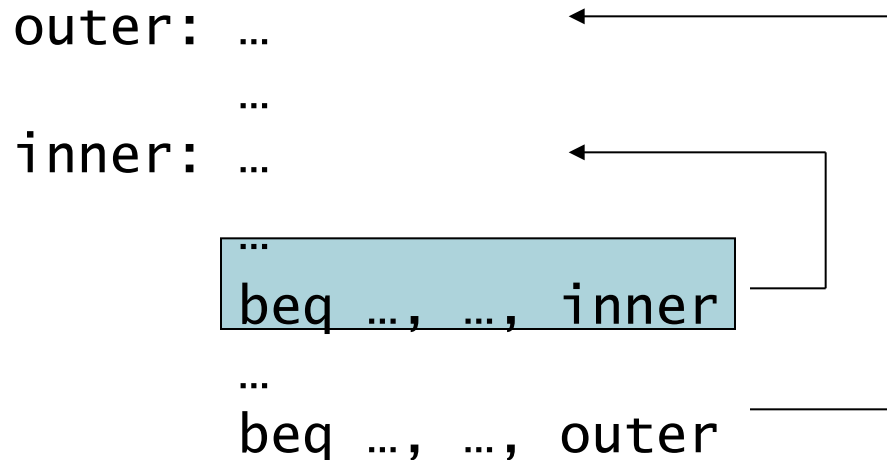


Dynamic Branch Prediction

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

1-Bit Predictor: Shortcoming

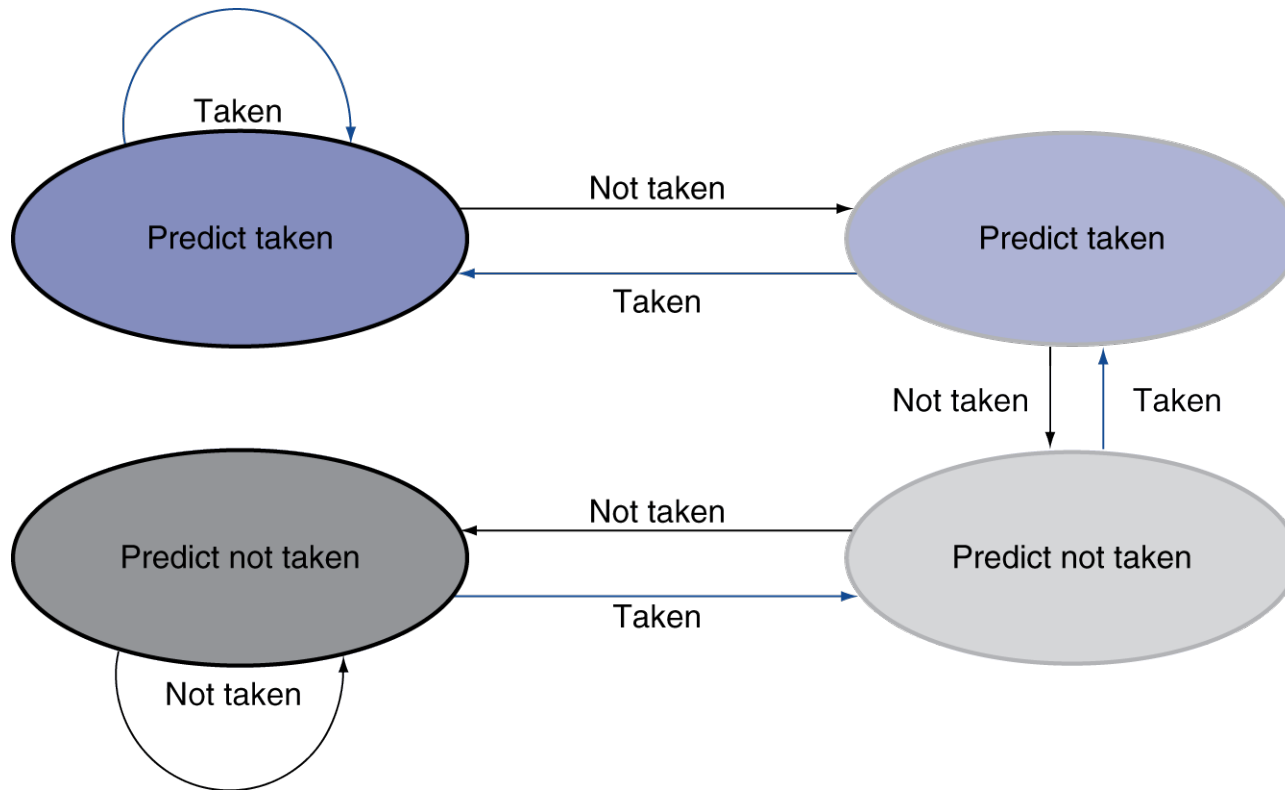
- Inner loop branches mispredicted twice!



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

2-Bit Predictor

- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
 - 1-cycle penalty for a taken branch
- Branch target buffer
 - Cache of target addresses
 - Indexed by PC when instruction fetched
 - If hit and instruction is branch predicted taken, can fetch target immediately

Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
 - Different ISAs use the terms differently
- Exception
 - Arises within the CPU
 - e.g., undefined opcode, overflow, syscall, ...
- Interrupt
 - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
 - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
 - In MIPS: Cause register
 - We'll assume 1-bit
 - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
 - Take corrective action
 - use EPC to return to program
- Otherwise
 - Terminate program
 - Report error using EPC, cause, ...

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
add \$1, \$2, \$1
 - Prevent \$1 from being clobbered
 - Complete previous instructions
 - Flush add and subsequent instructions
 - Set Cause and EPC register values
 - Transfer control to handler
- Similar to mispredicted branch
 - Use much of the same hardware

Speculation

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

Compiler/Hardware Speculation

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

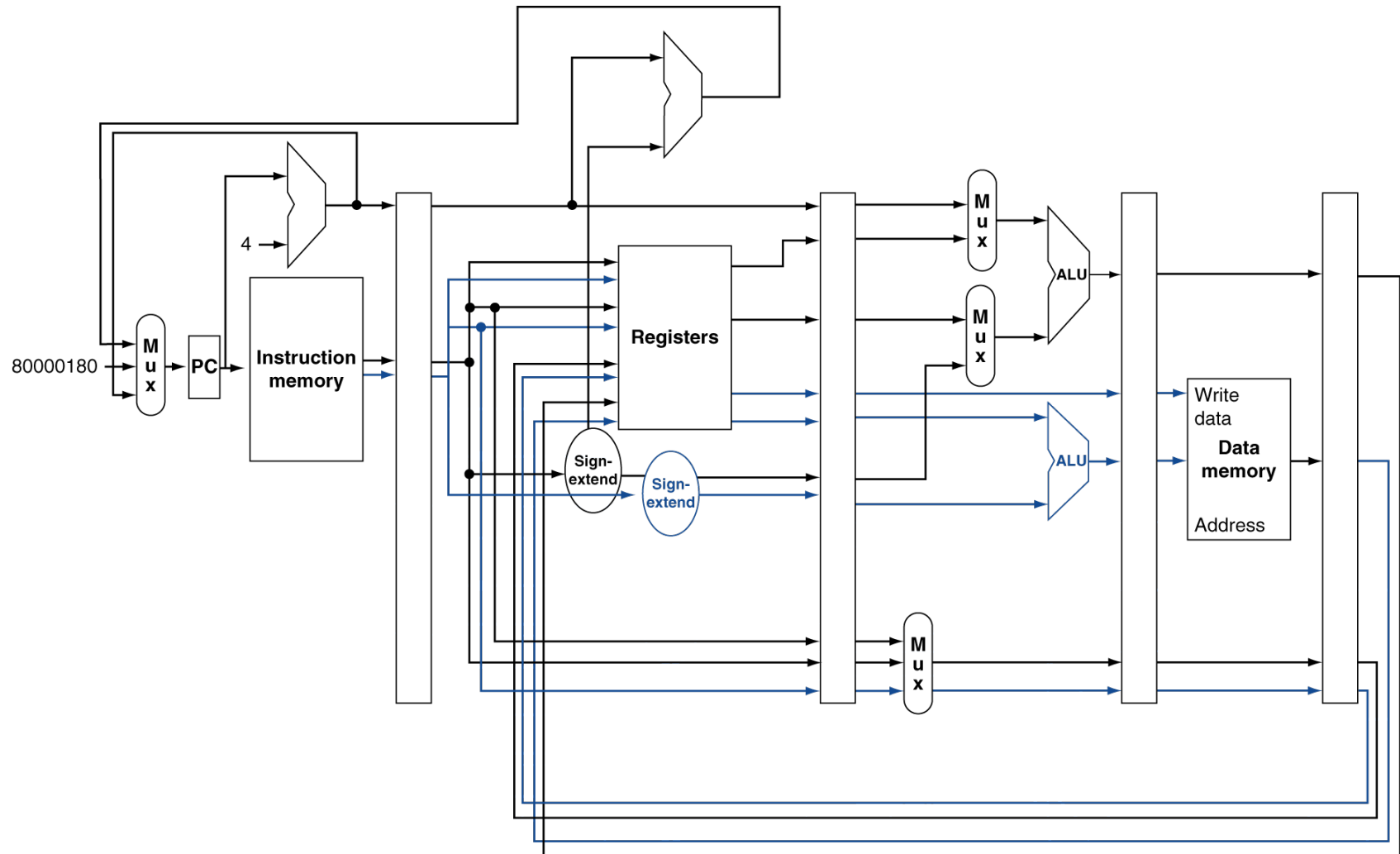
Static Multiple Issue

- Compiler groups instructions into “issue packets”
 - Group of instructions that can be issued on a single cycle
 - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
 - Specifies multiple concurrent operations
 - \Rightarrow Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
 - Reorder instructions into issue packets
 - No dependencies with a packet
 - Possibly some dependencies between packets
 - Varies between ISAs; compiler must know!
 - Pad with nop if necessary

MIPS with Static Dual Issue



MIPS with Static Dual Issue

- Two-issue packets
 - One ALU/branch instruction
 - One load/store instruction
 - 64-bit aligned
 - ALU/branch, then load/store
 - Pad an unused instruction with nop

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

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
 - Forwarding avoided stalls with single-issue
 - Now can't use ALU result in load/store in same packet
 - `add $t0, $s0, $s1`
`load $s2, 0($t0)`
 - Split into two packets, effectively a stall
- Load-use hazard
 - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw    $t0, 0($s1)      # $t0=array element
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 0($s1)      # store result
      addi  $s1, $s1, -4     # decrement pointer
      bne   $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1, -4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)**

Loop Unrolling

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

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi \$s1 , \$s1, -16	lw \$t0 , 0(\$s1)	1
	nop	lw \$t1 , 12(\$s1)	2
	addu \$t0 , \$t0 , \$s2	lw \$t2 , 8(\$s1)	3
	addu \$t1 , \$t1 , \$s2	lw \$t3 , 4(\$s1)	4
	addu \$t2 , \$t2 , \$s2	sw \$t0 , 16(\$s1)	5
	addu \$t3 , \$t4 , \$s2	sw \$t1 , 12(\$s1)	6
	nop	sw \$t2 , 8(\$s1)	7
	bne \$s1 , \$zero, Loop	sw \$t3 , 4(\$s1)	8

- $IPC = 14/8 = 1.75$
 - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, ... each cycle
 - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
 - Though it may still help
 - Code semantics ensured by the CPU

Speculation

- Predict branch and continue issuing
 - Don't commit until branch outcome determined
- Load speculation
 - Avoid load and cache miss delay
 - Predict the effective address
 - Predict loaded value
 - Load before completing outstanding stores
 - Bypass stored values to load unit
 - Don't commit load until speculation cleared

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predictable
 - e.g., cache misses
- Can't always schedule around branches
 - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture

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

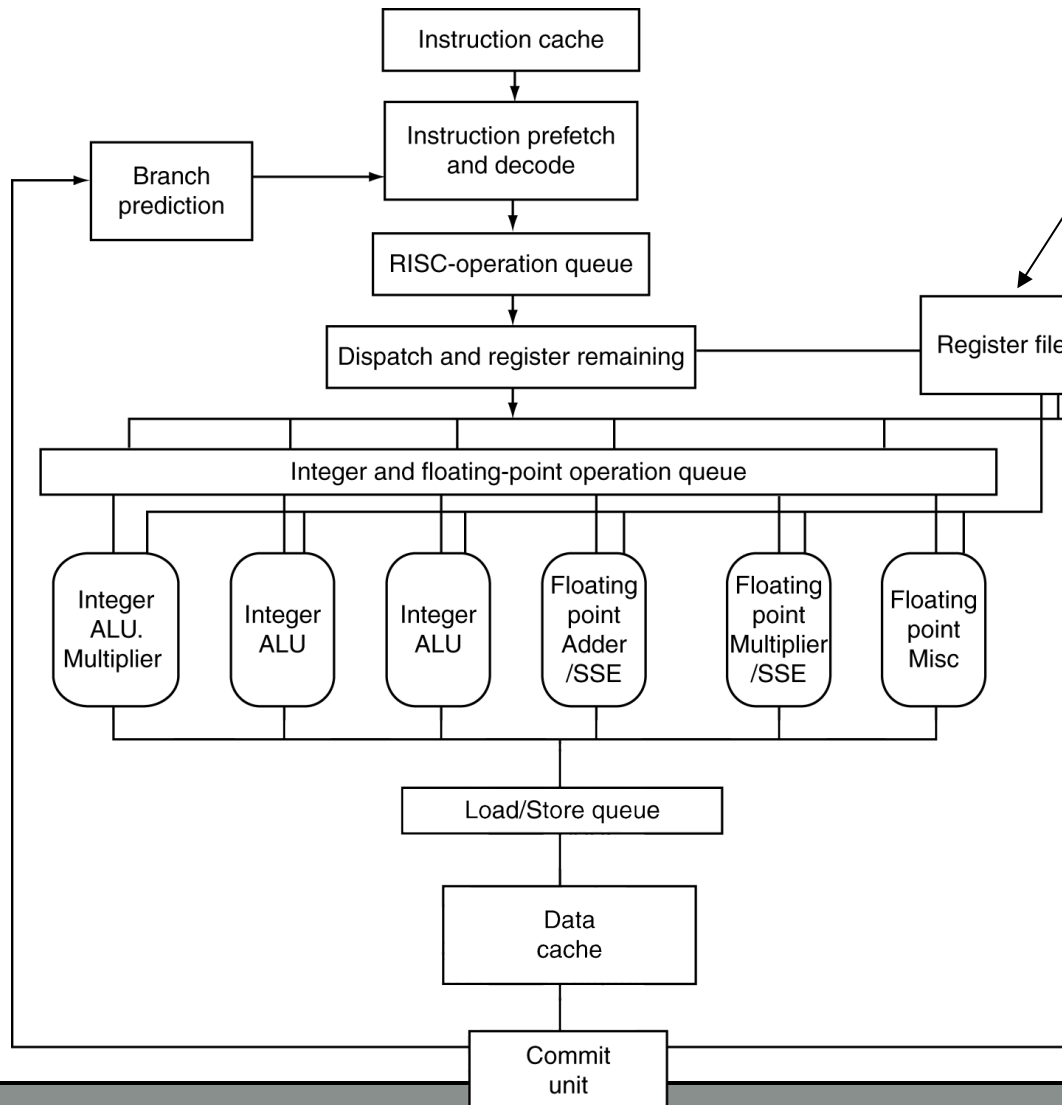
Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

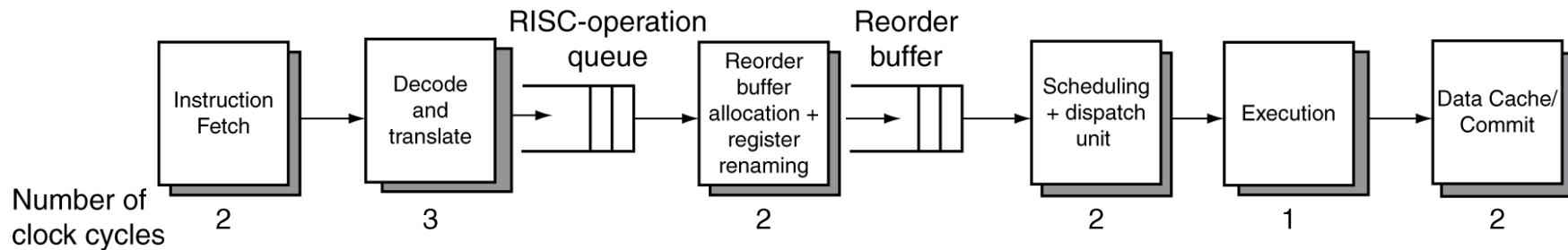
The Opteron X4 Microarchitecture

72
phys
ical
regi
ster
s



The Opteron X4 Pipeline Flow

- For integer operations



- **FP is 5 stages longer**
- **Up to 106 RISC-ops in progress**

- **Bottlenecks**

- **Complex instructions with long dependencies**
- **Branch mispredictions**
- **Memory access delays**

Fallacies

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