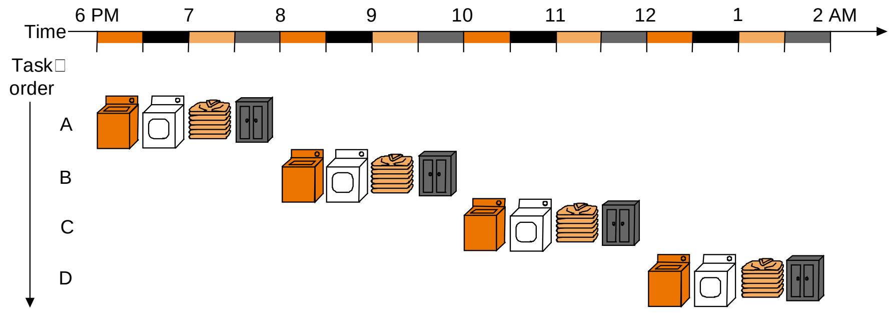
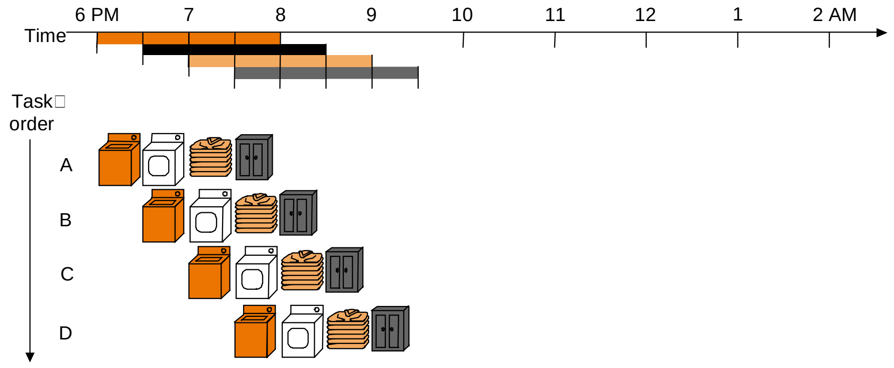
182.690 RECHNERSTRUKTUREN – PIPELINING

Thomas Polzer
tpolzer@ecs.tuwien.ac.at
Institut für Technische Informatik

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance
- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = number of stages





MIPS Pipeline

- Five stages, one step per stage
  - IF: Instruction fetch from (instruction) memory
  - ID: Instruction decode & register read
  - EX: Execute operation or calculate address
  - MEM: Access (data) memory operand
  - WB: Write result back to register

Pipeline Performance

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

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw						
sw						
R-format						
beq						

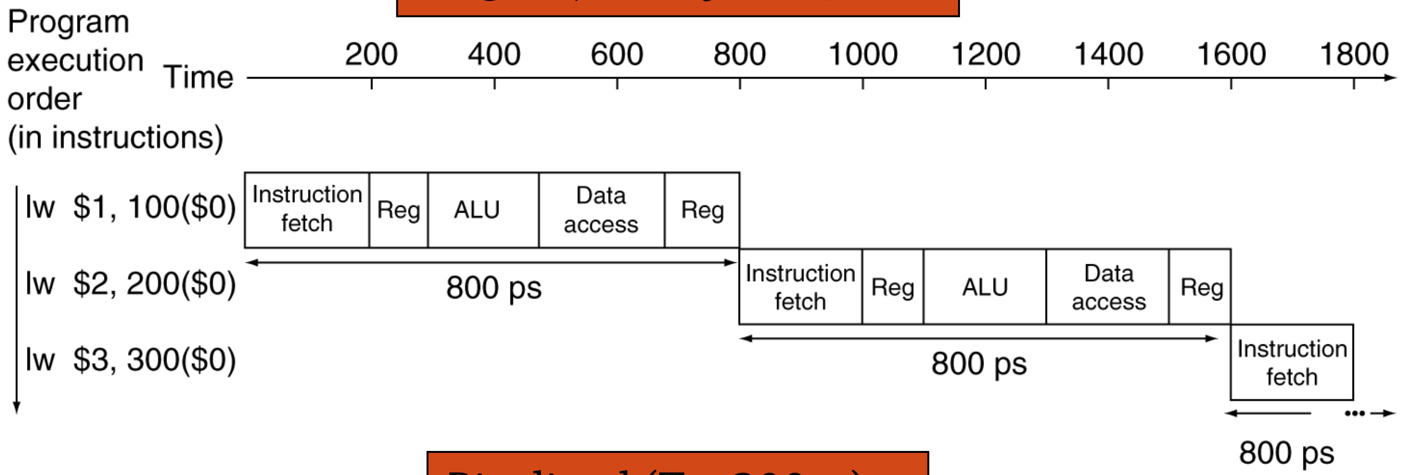
Pipeline Performance

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

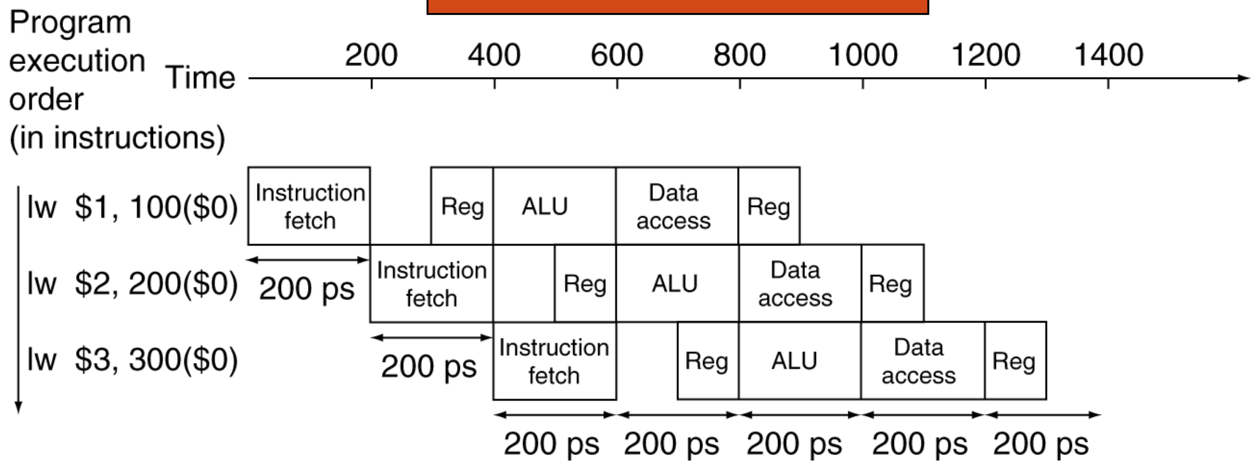
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



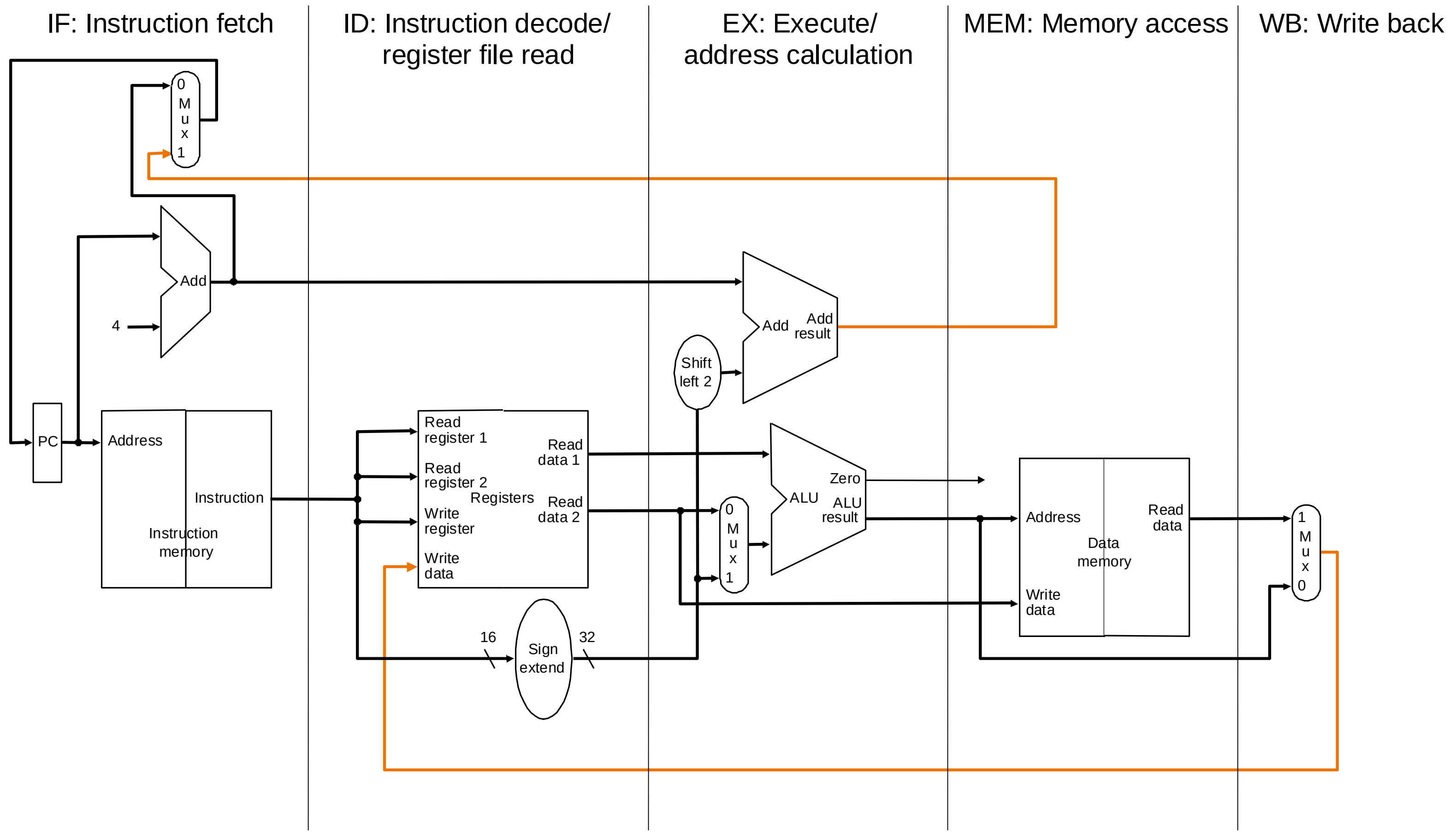
Pipeline Speedup

- k-stage pipeline increases the throughput by factor k (ideally)
- Execution time of instructions unchanged
- Limits:
  - Balance of the stages
  - Filling the pipeline
  - Dependencies (Hazards)

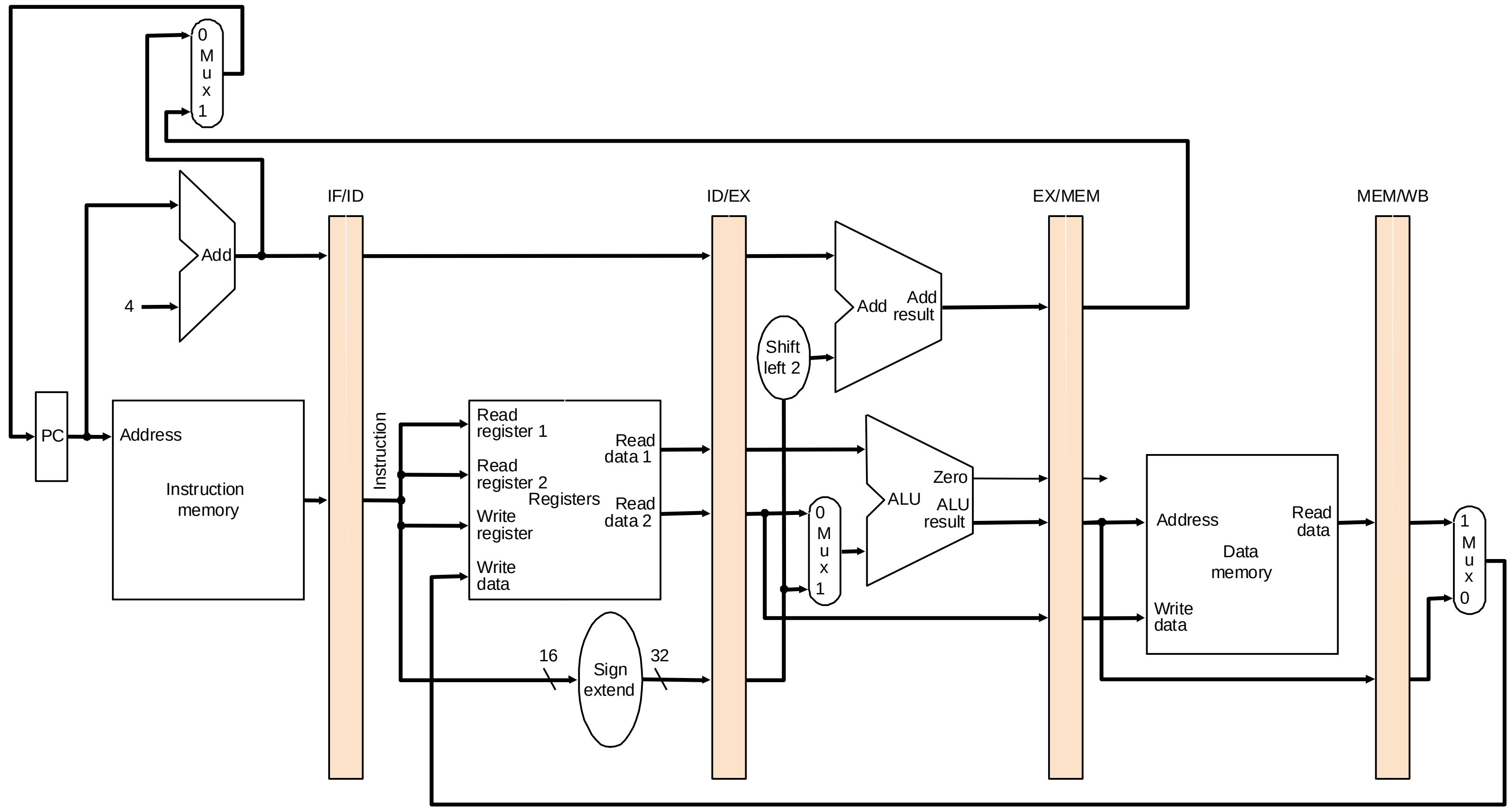
Pipelining and ISA Design

- 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

Segmentation of the Datapath



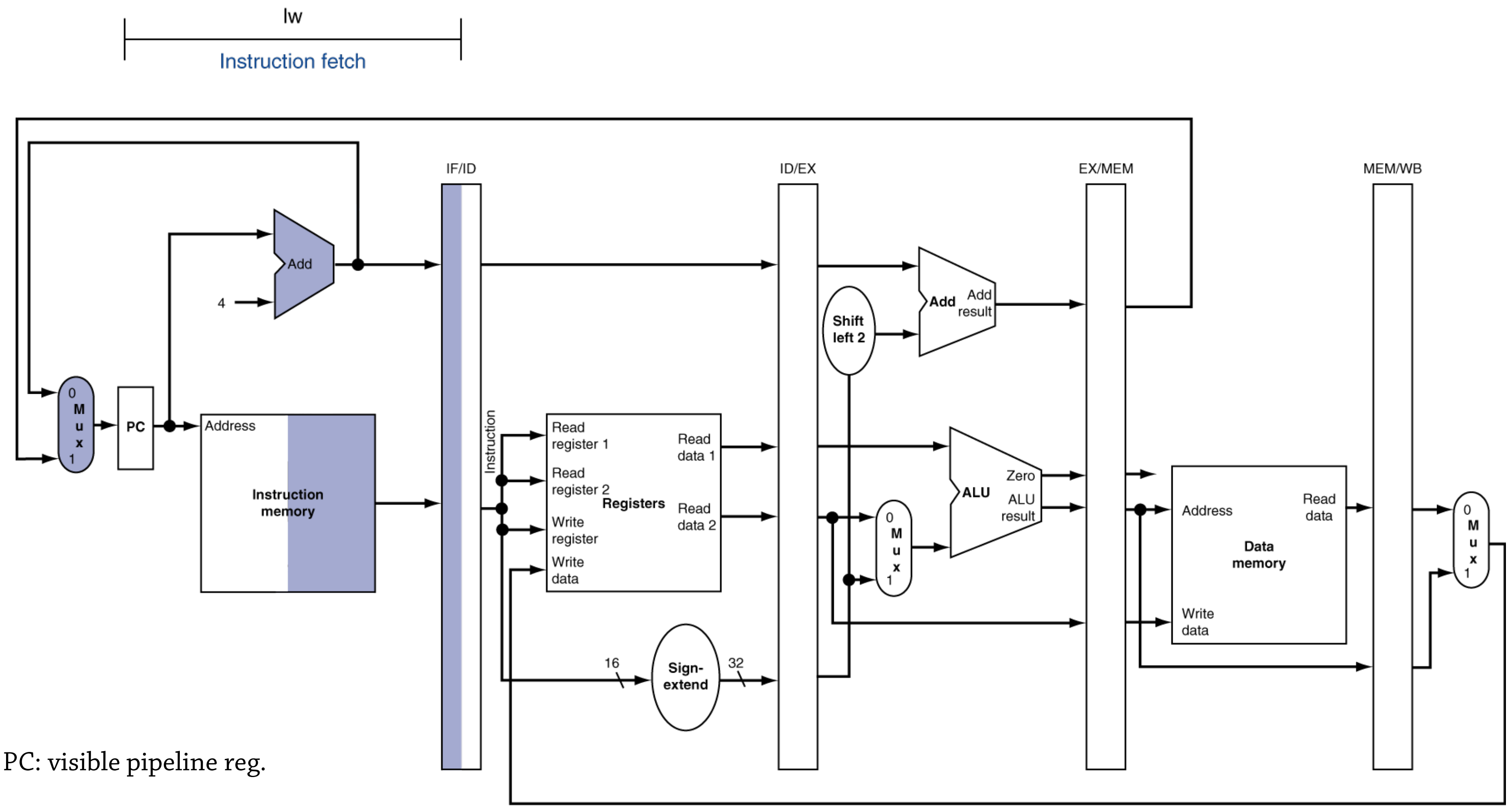
Pipeline Registers



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
- We’ll look at “single-clock-cycle” diagrams for load & store

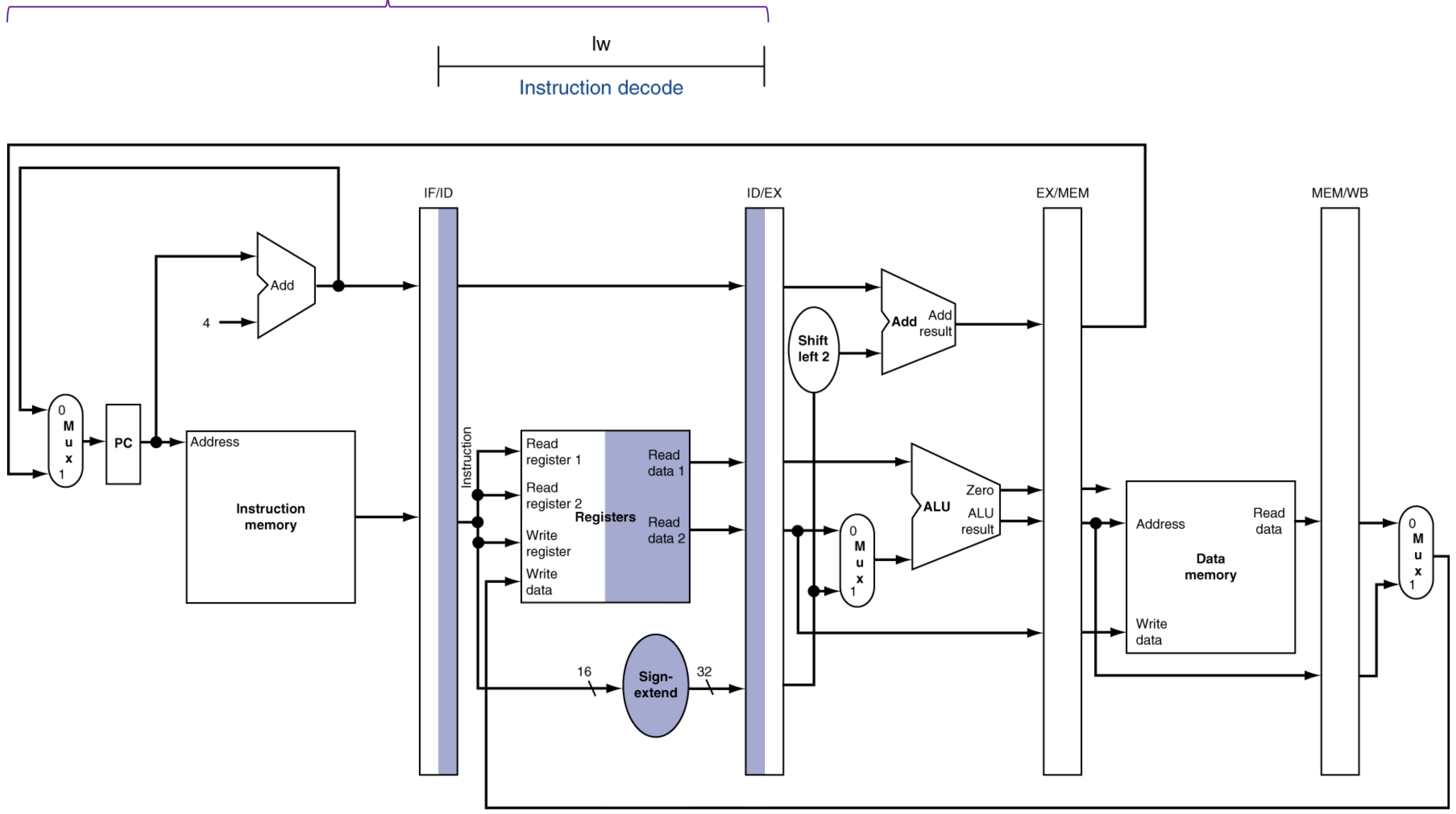
Instruction Fetch for Load, Store, …



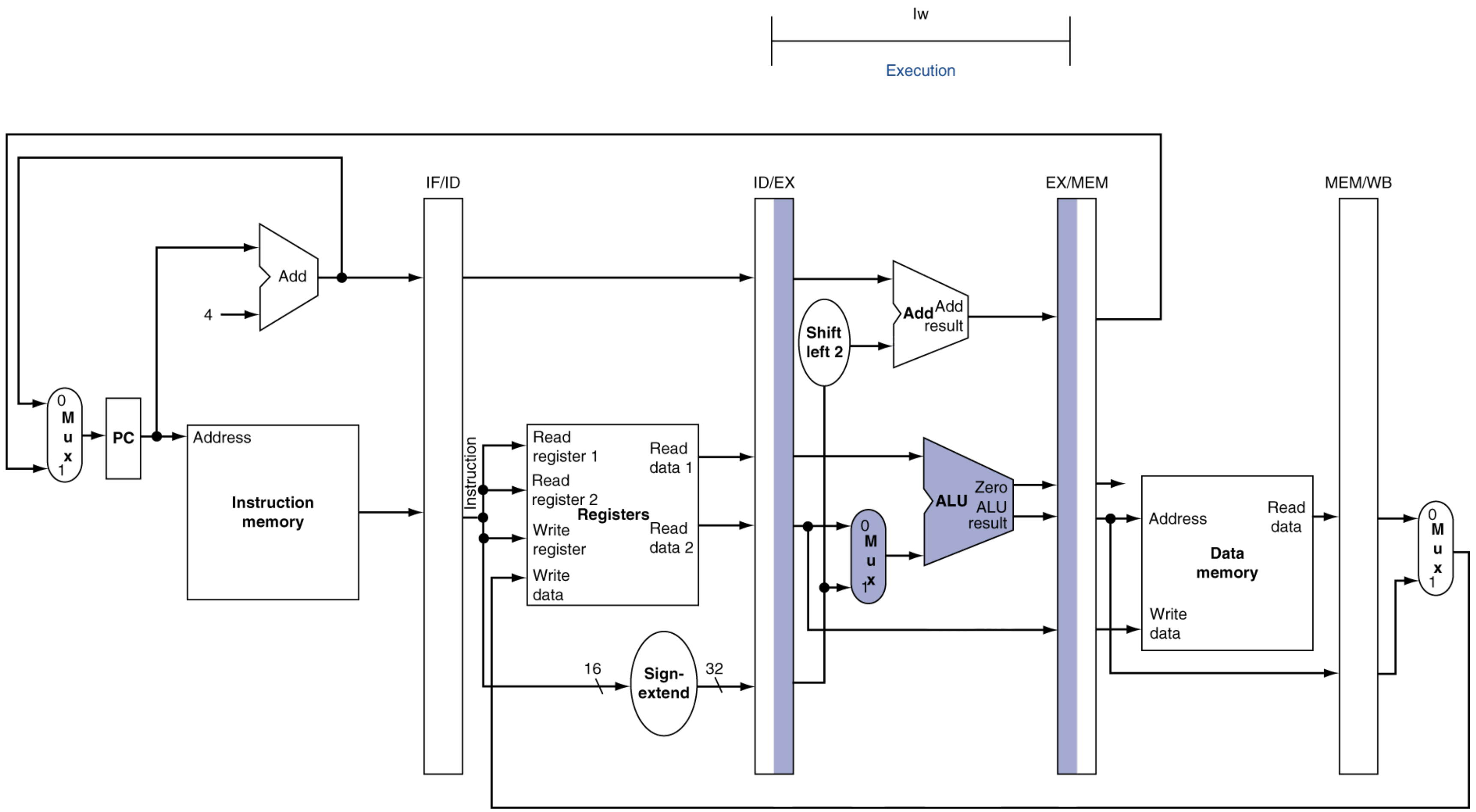
PC: visible pipeline reg.

Instruction Decode for Load, Store, …

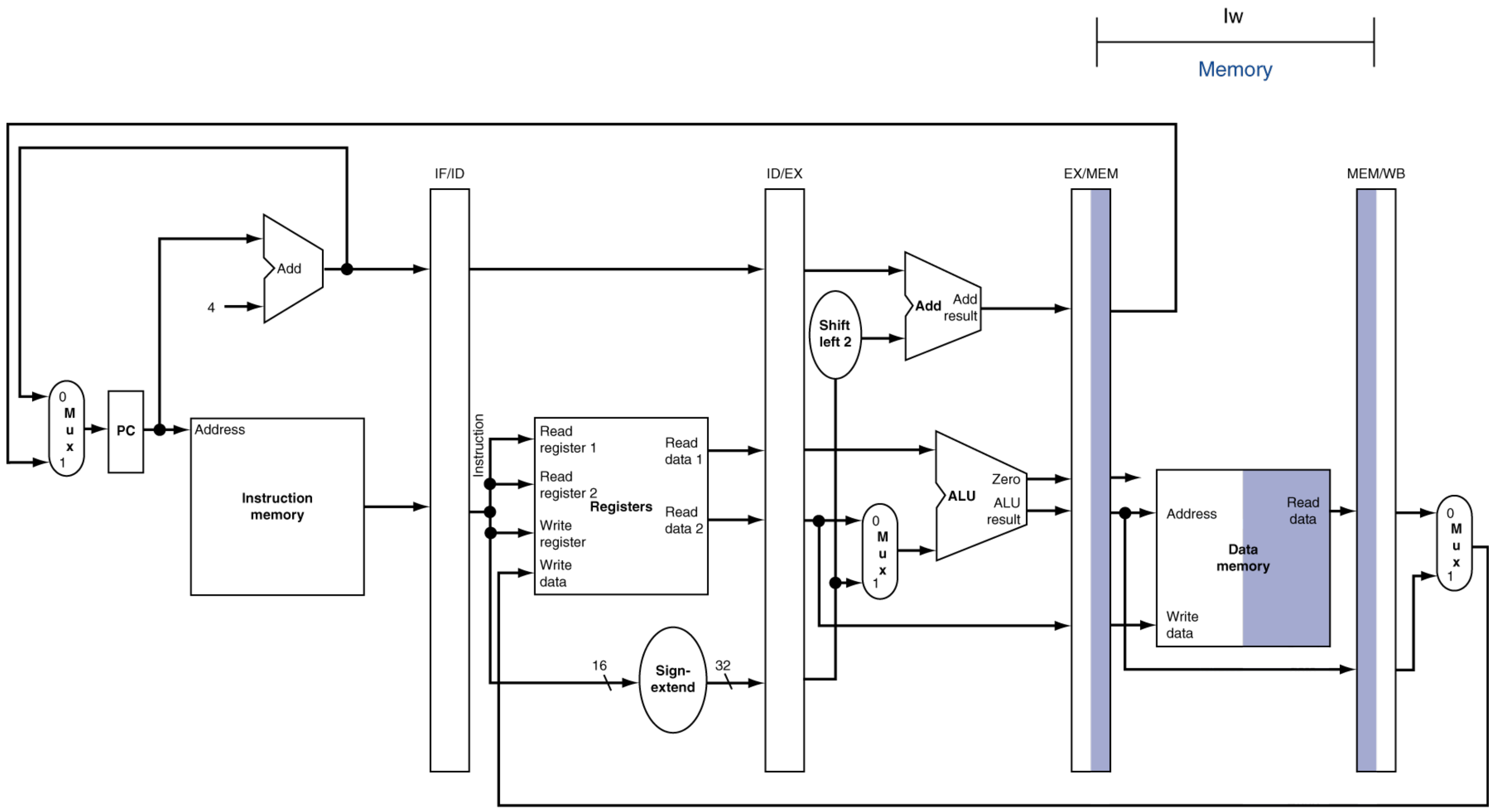
Executed by all instructions



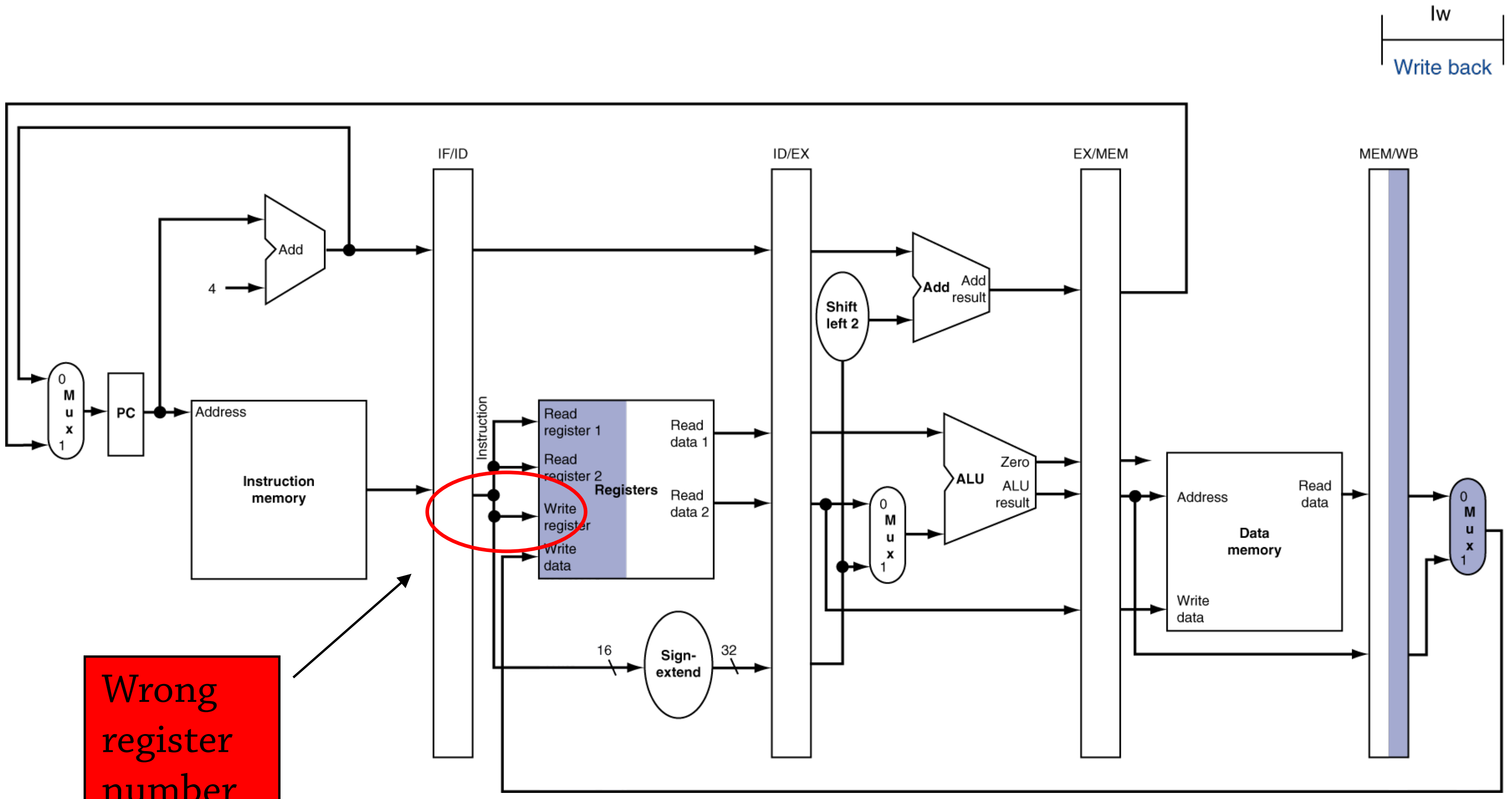
Execute for Load



Memory Access for Load

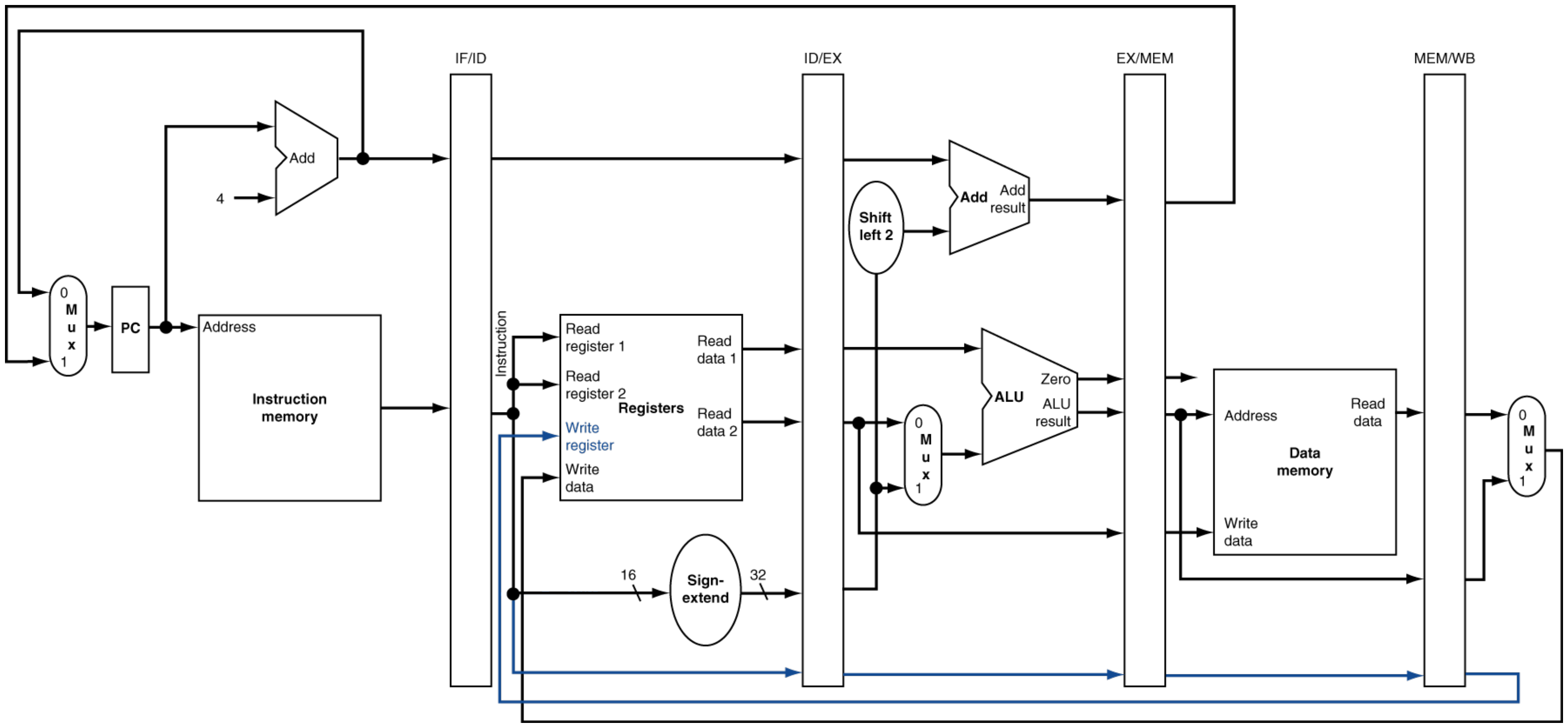


Writeback for Load

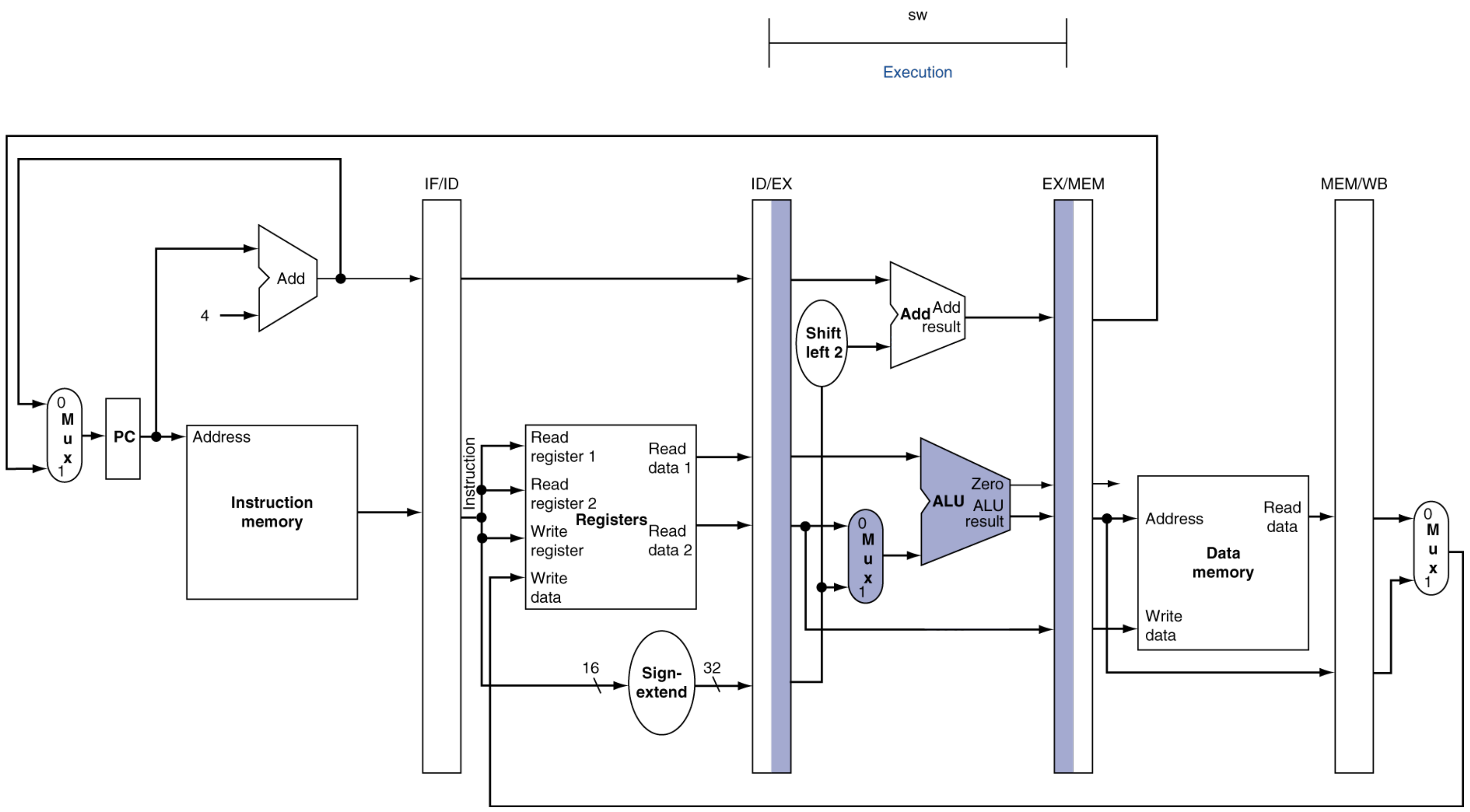


Wrong register number

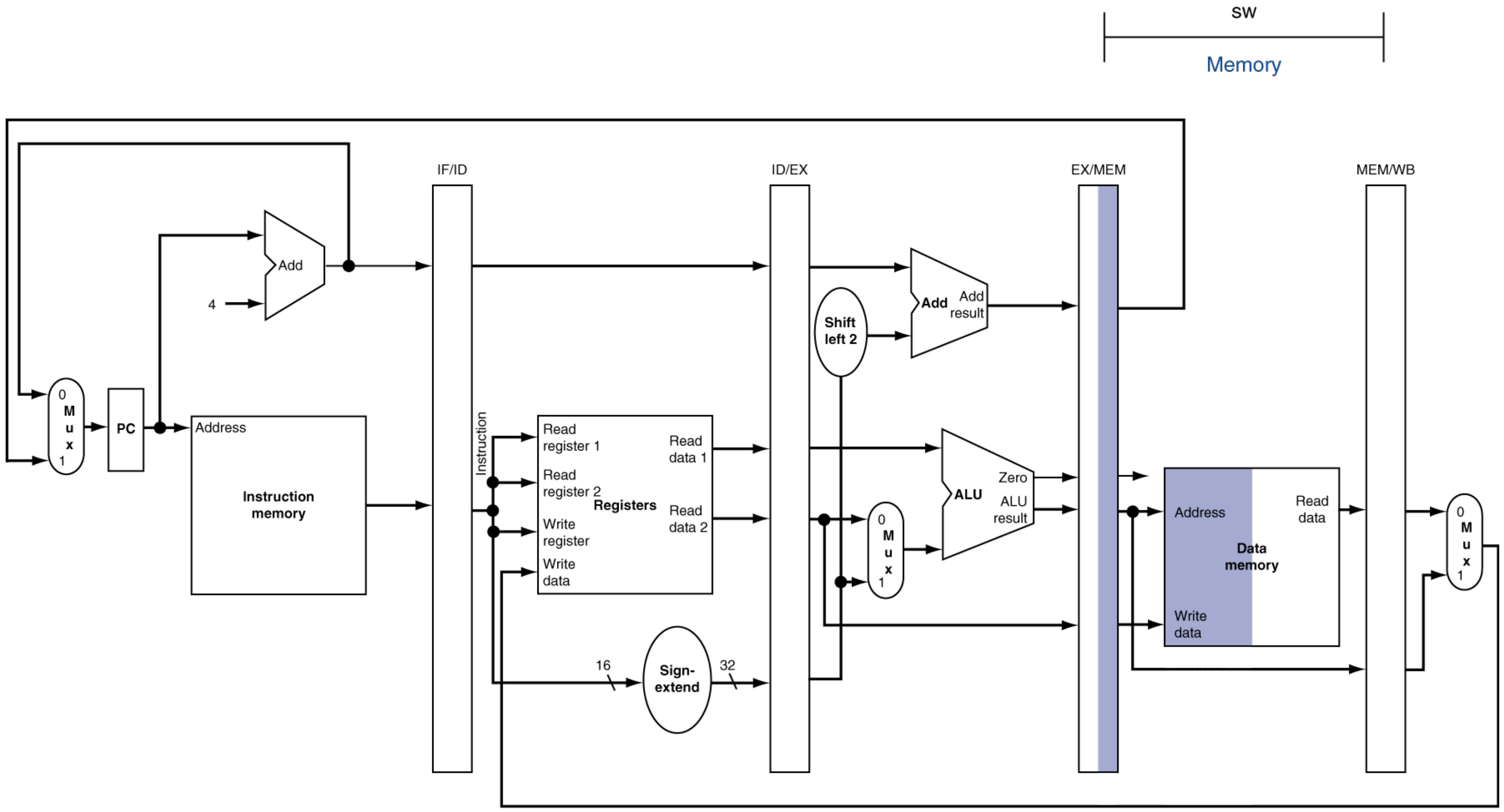
Corrected Datapath for Load



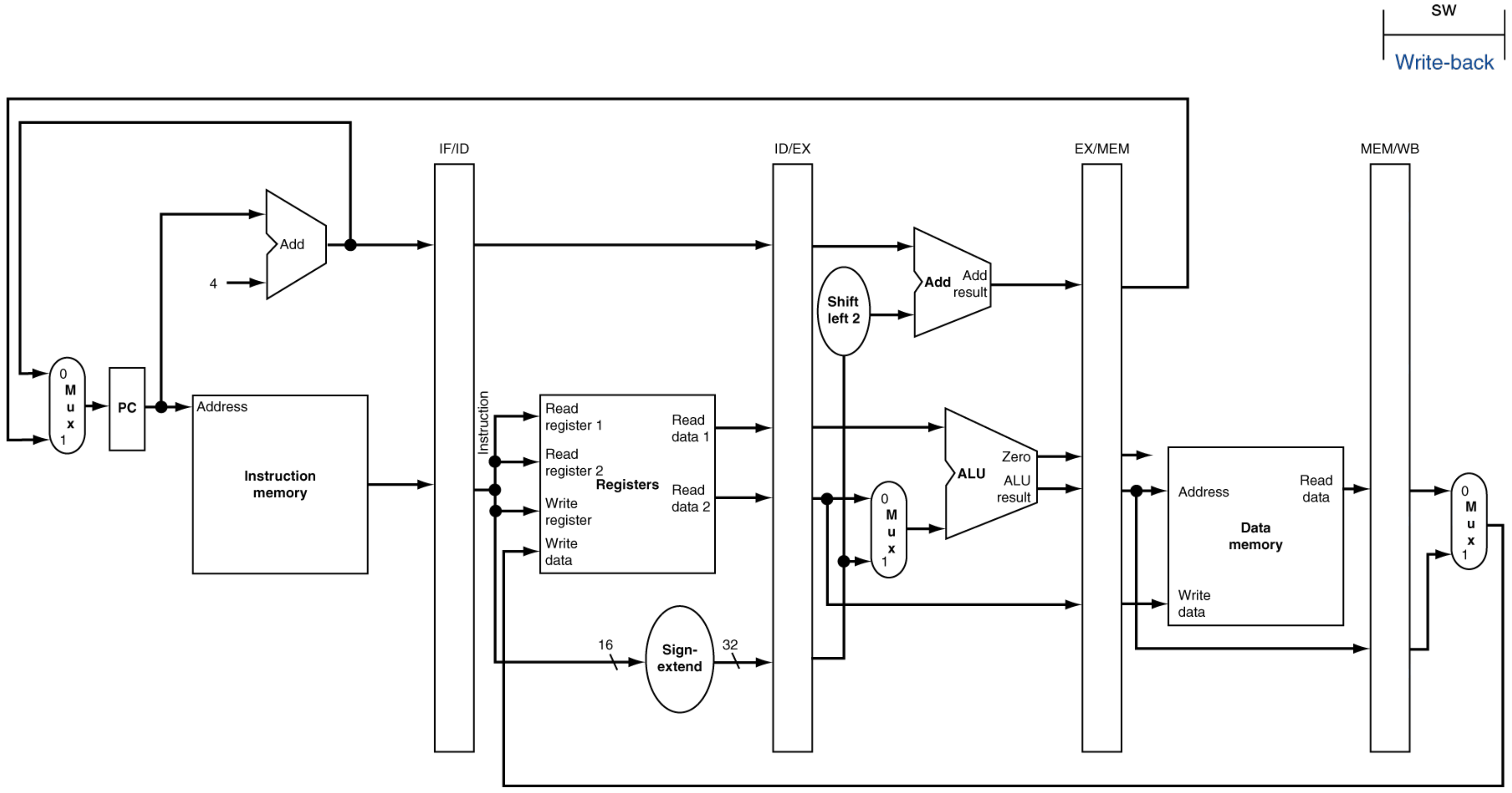
Execute for Store



Memory Access for Store

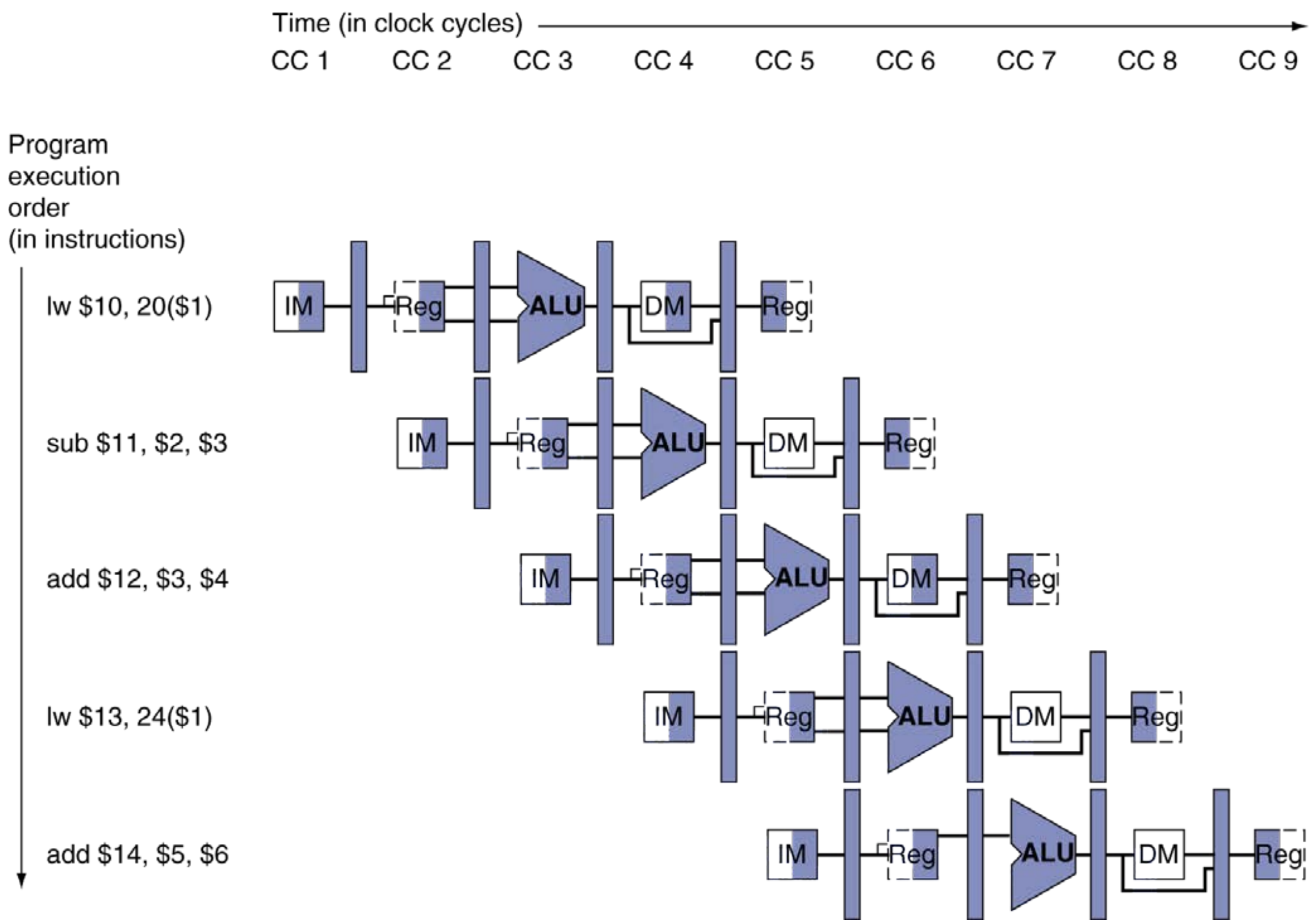


Writeback for Store



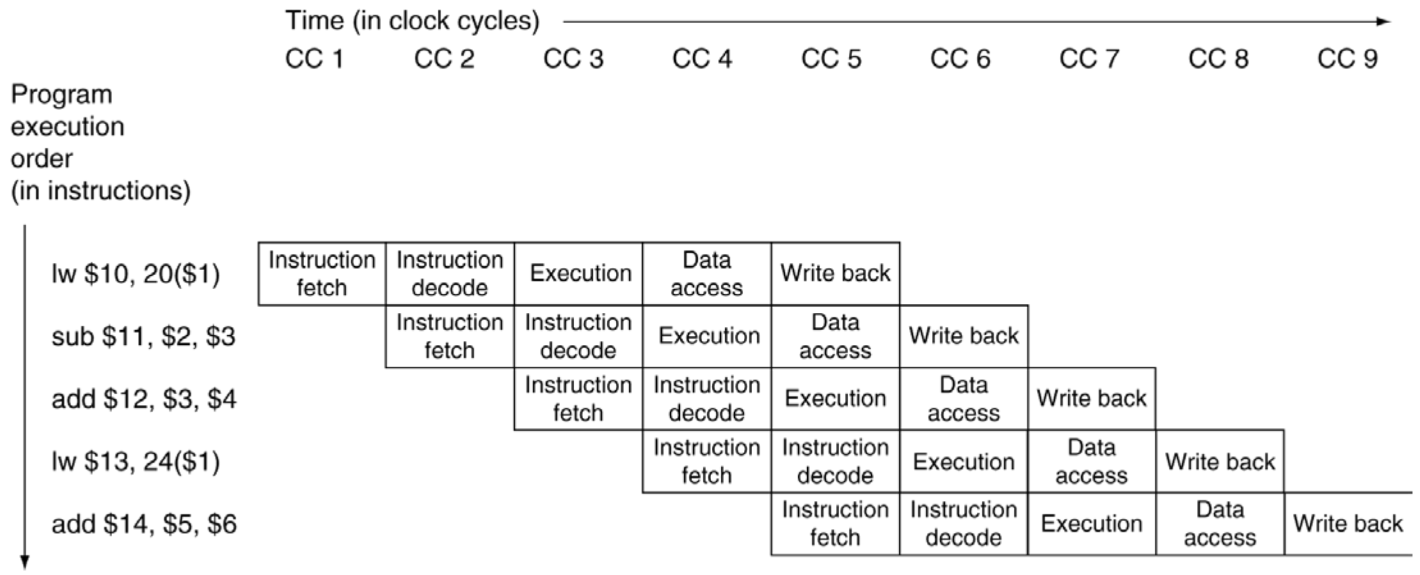
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

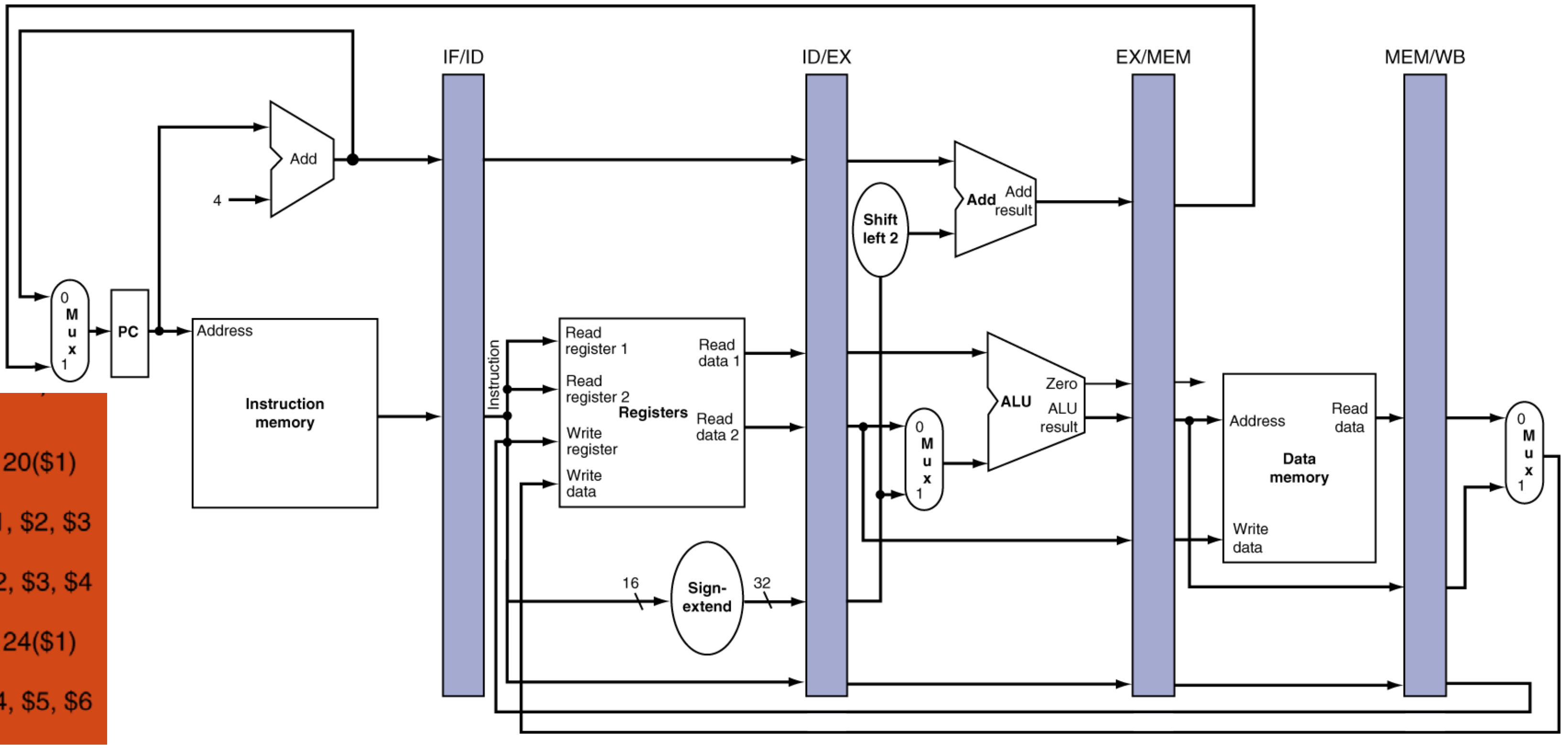
- Traditional form



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



lw $10, 20($1)

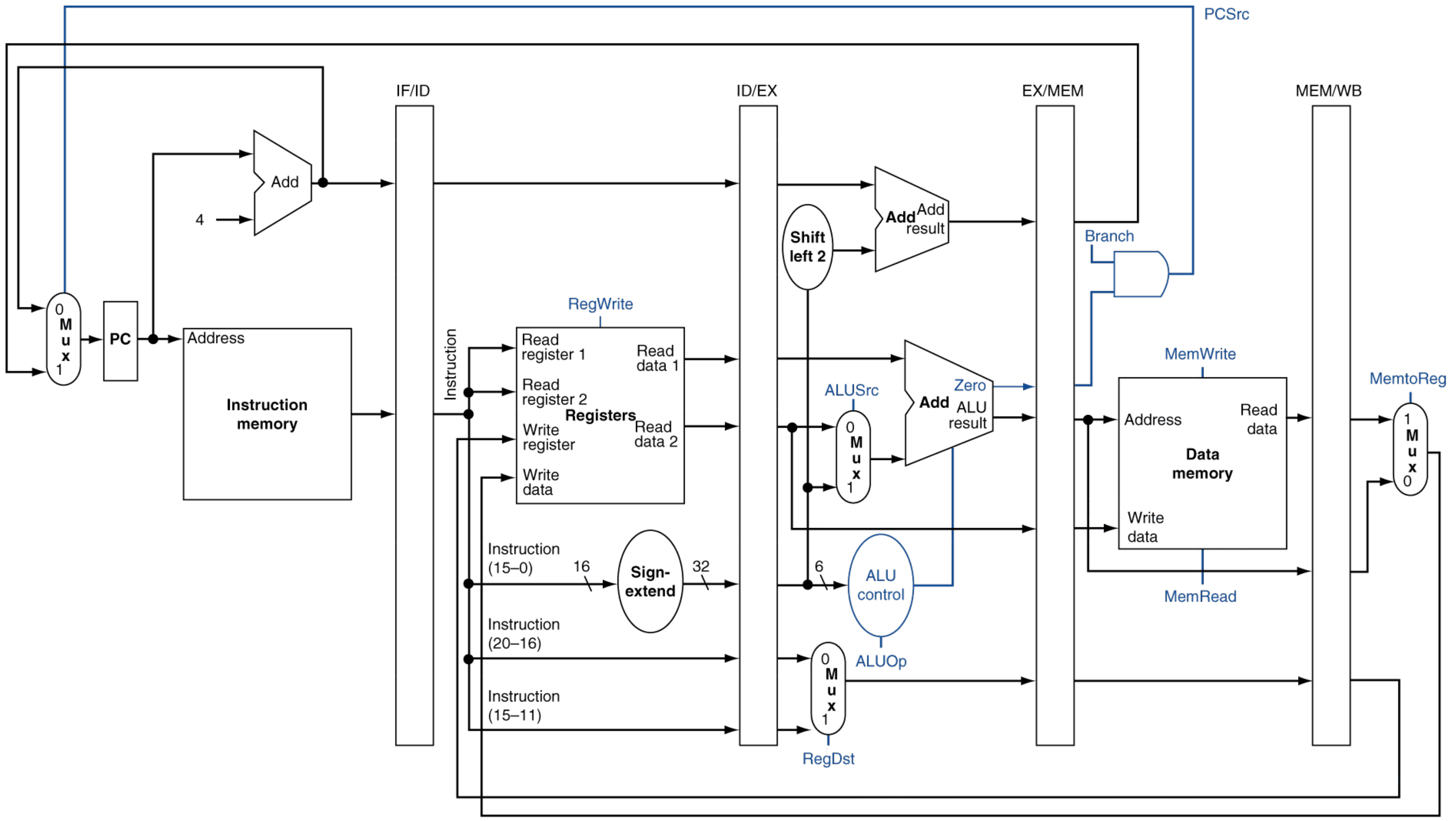
sub $11, $2, $3

add $12, $3, $4

lw $13, 24($1)

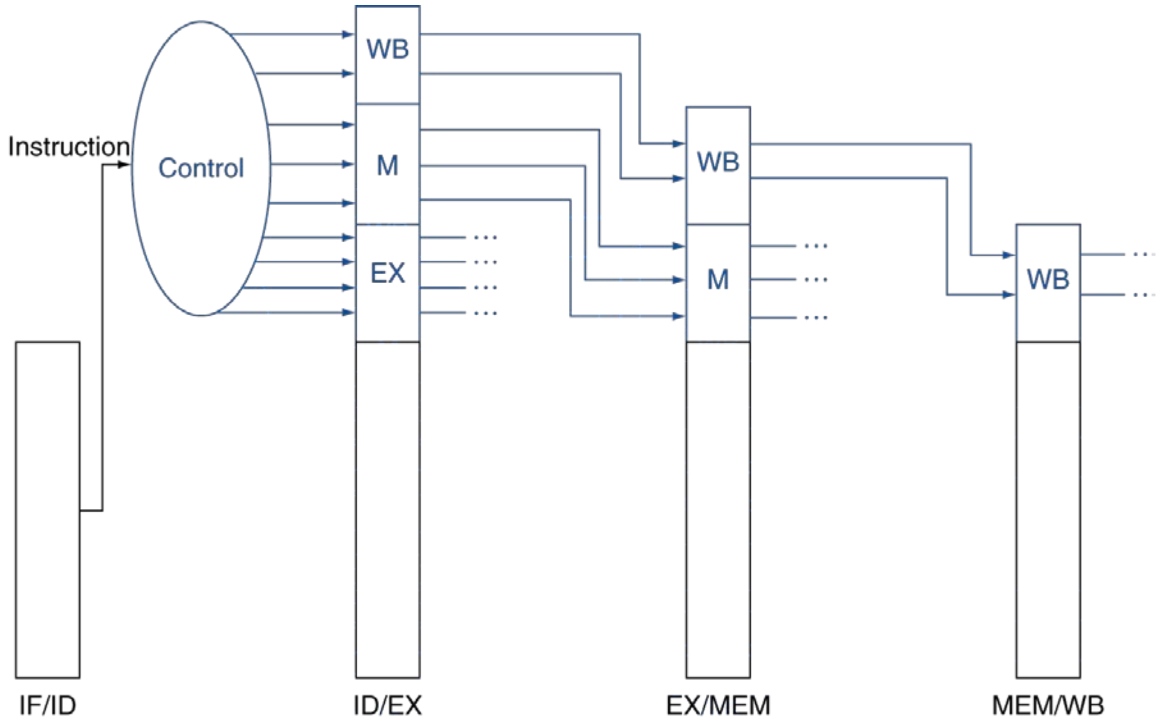
add $14, $5, $6

Pipelined Control (Simplified)

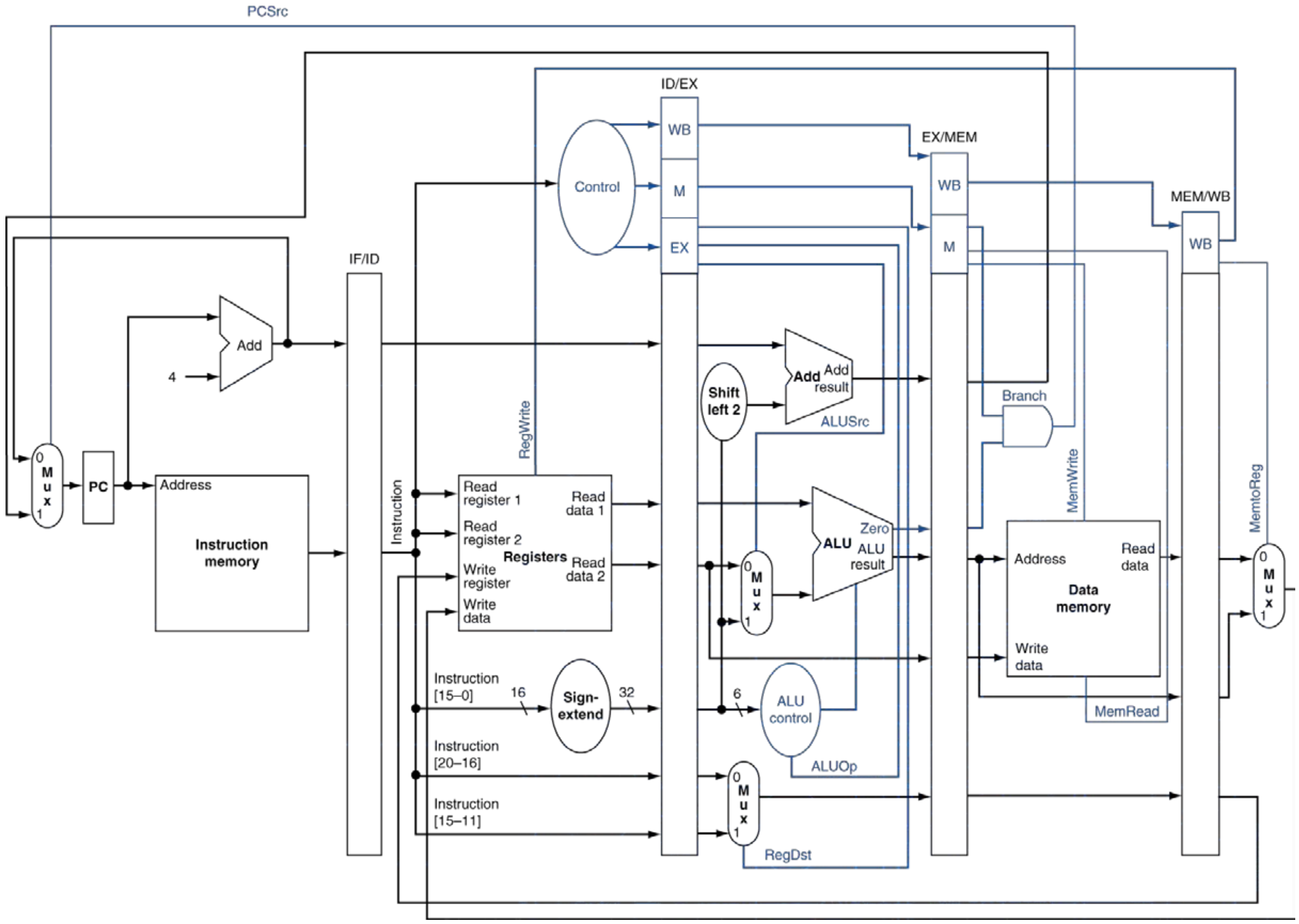


Pipelined Control

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



Pipelined Control



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 datapath require separate instruction/data memories
  - Or separate instruction/data caches

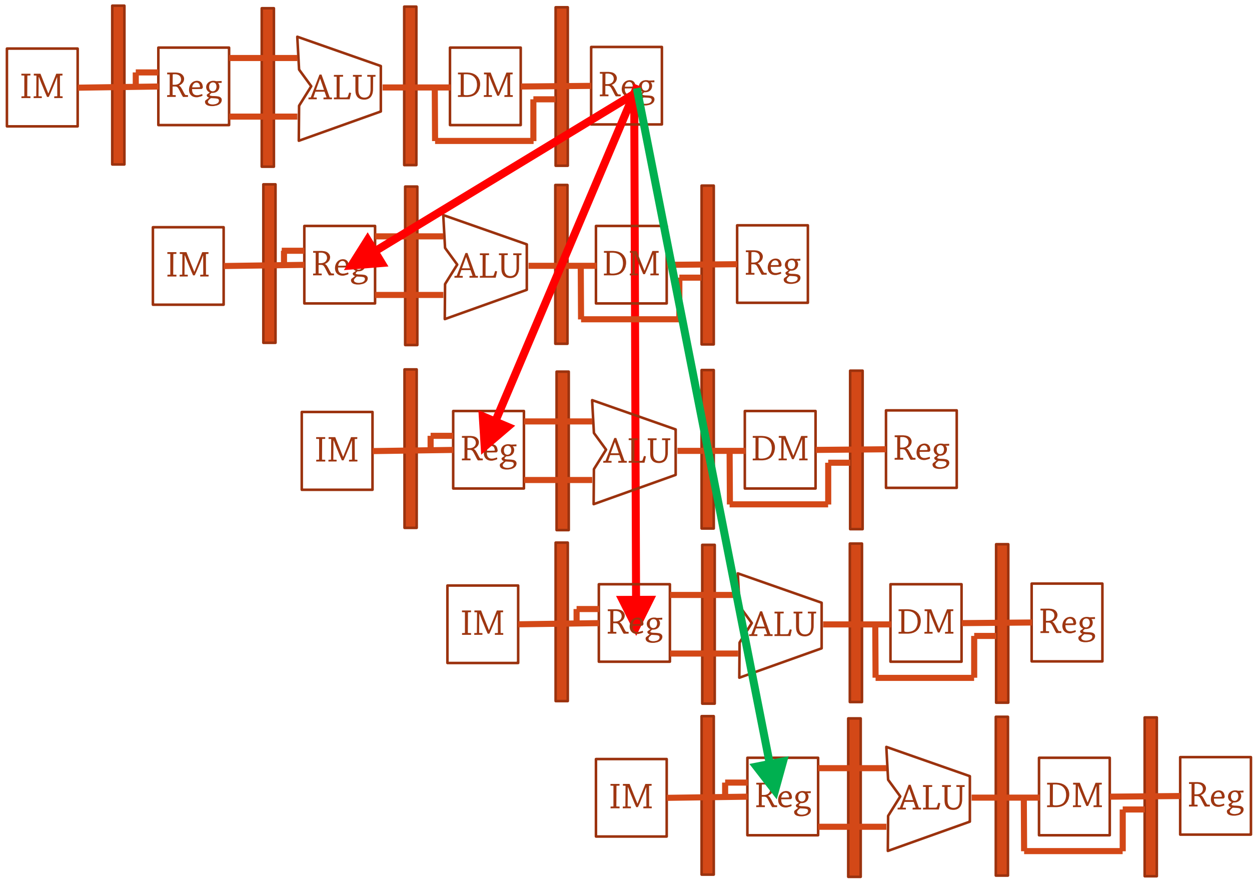
Types of Data-Hazards

- read-after-write (RAW)
  - Register is read before the previous instruction has written it
- write-after-read (WAR)
  - Register is written before the previous instruction has read it
- write-after-write (WAW)
  - Register written but then overwritten by a previous instruction

Register Access

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

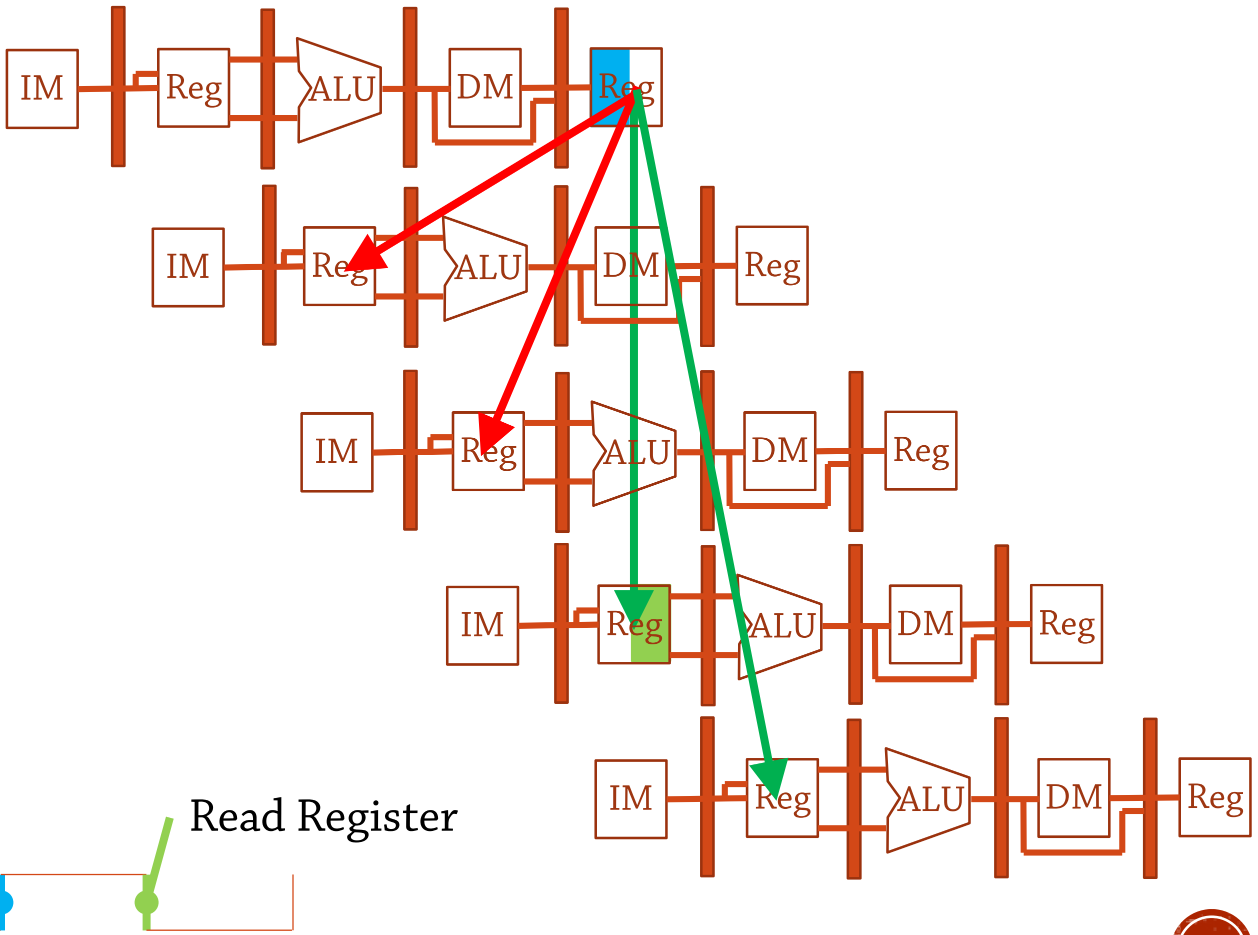
```
add $s0, $t0, $t1

sub $t2, $s0, $t3

or $t4, $s0, $s1

and $s3, $s0, $s2

sw $s4, 0($s0)
```



Register Access

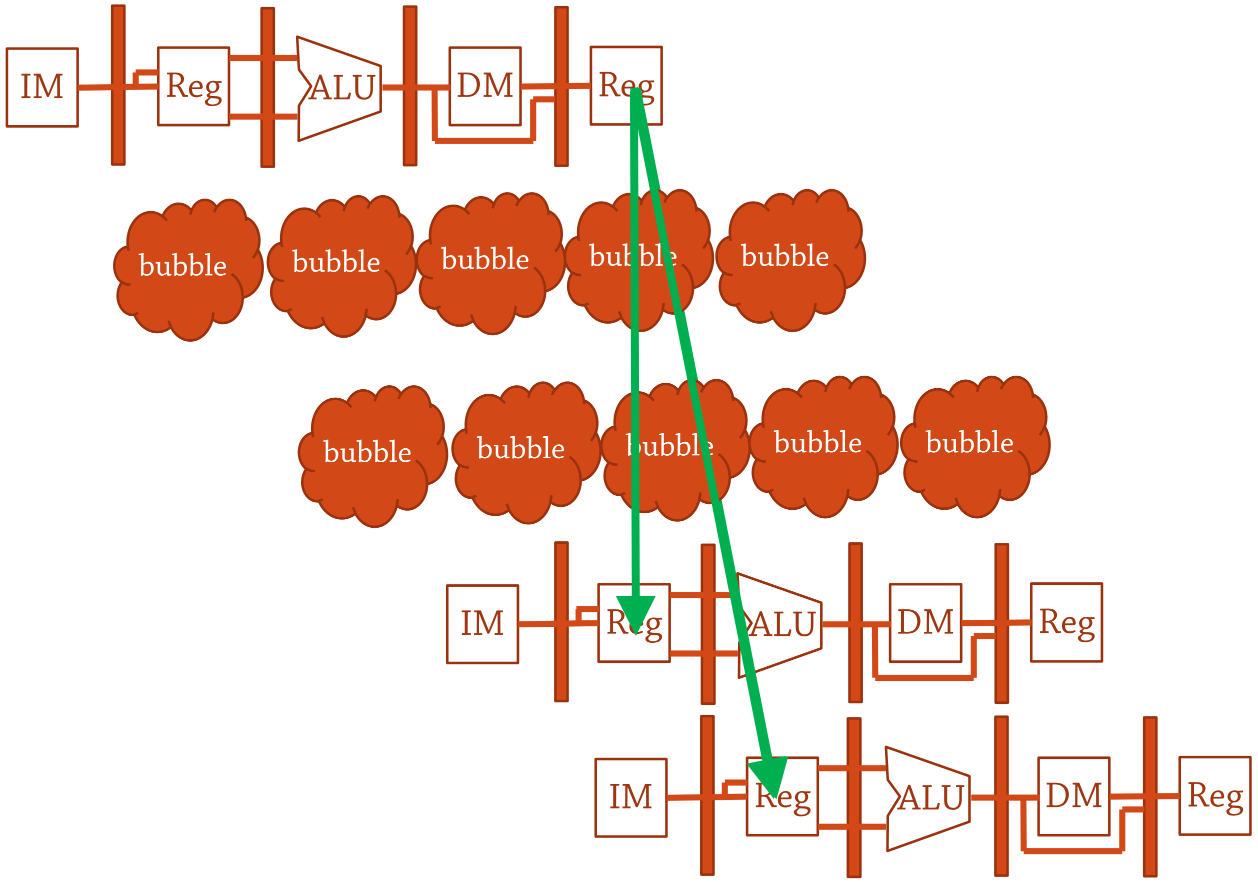
- Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half

```
add $s0, $t0, $t1

sub $t2, $s0, $t3

or $t4, $s0, $s1

and $s3, $s0, $s2

sw $s4, 0($s0)
```



Inserting NOPs

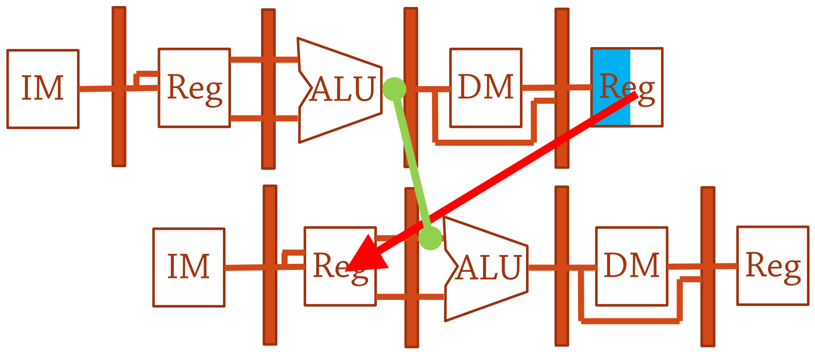
```
add $s0, $t0, $t1

nop

nop

sub $t2, $s0, $t3

or $t4, $s0, $s1
```



Forwarding

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

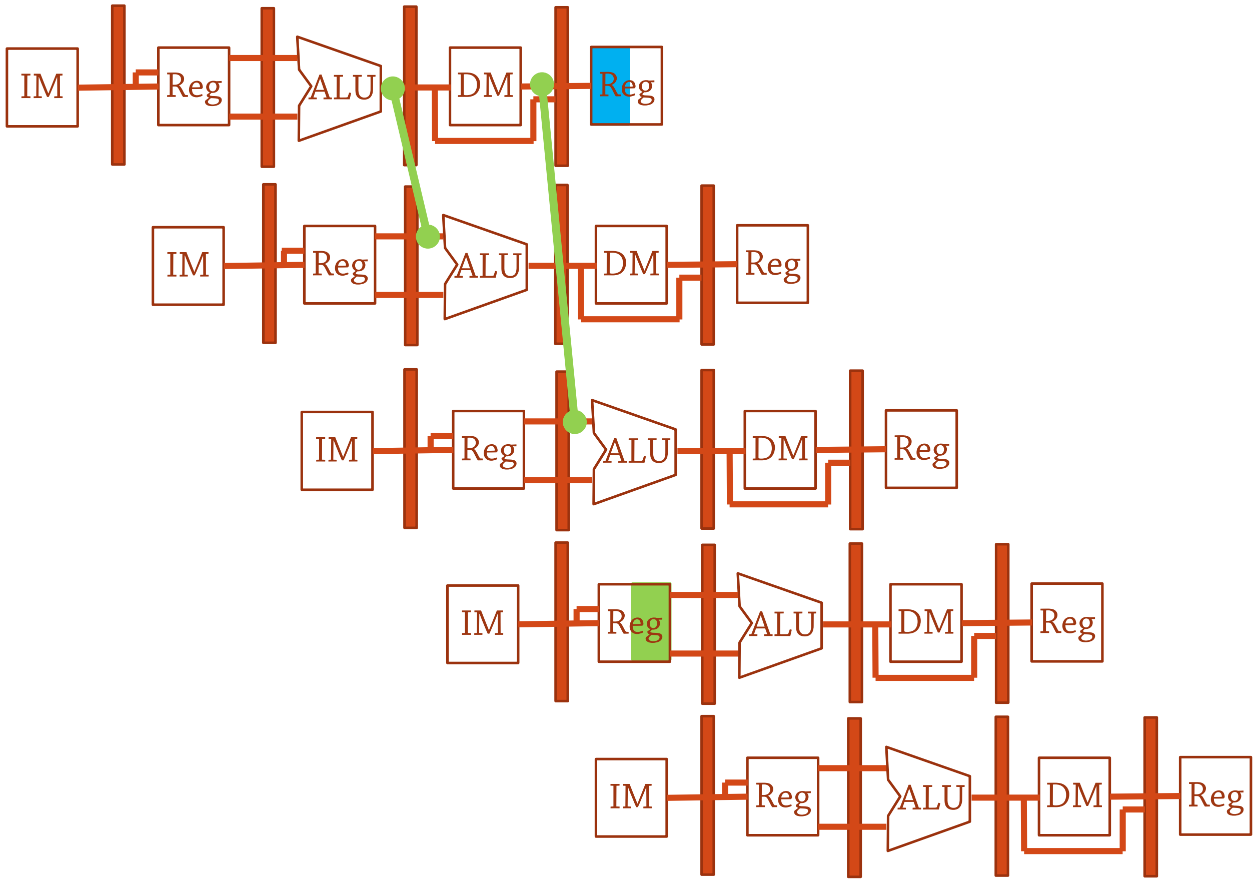
```
add $s0, $t0, $t1

sub $t2, $s0, $t3
```



Dependencies & Forwarding

- How do we detect when to forward?

```
add $s0, $t0, $t1

sub $t2, $s0, $t3

or $t4, $s0, $s1

and $s3, $s0, $s2

sw $s4, 0($s0)
```



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

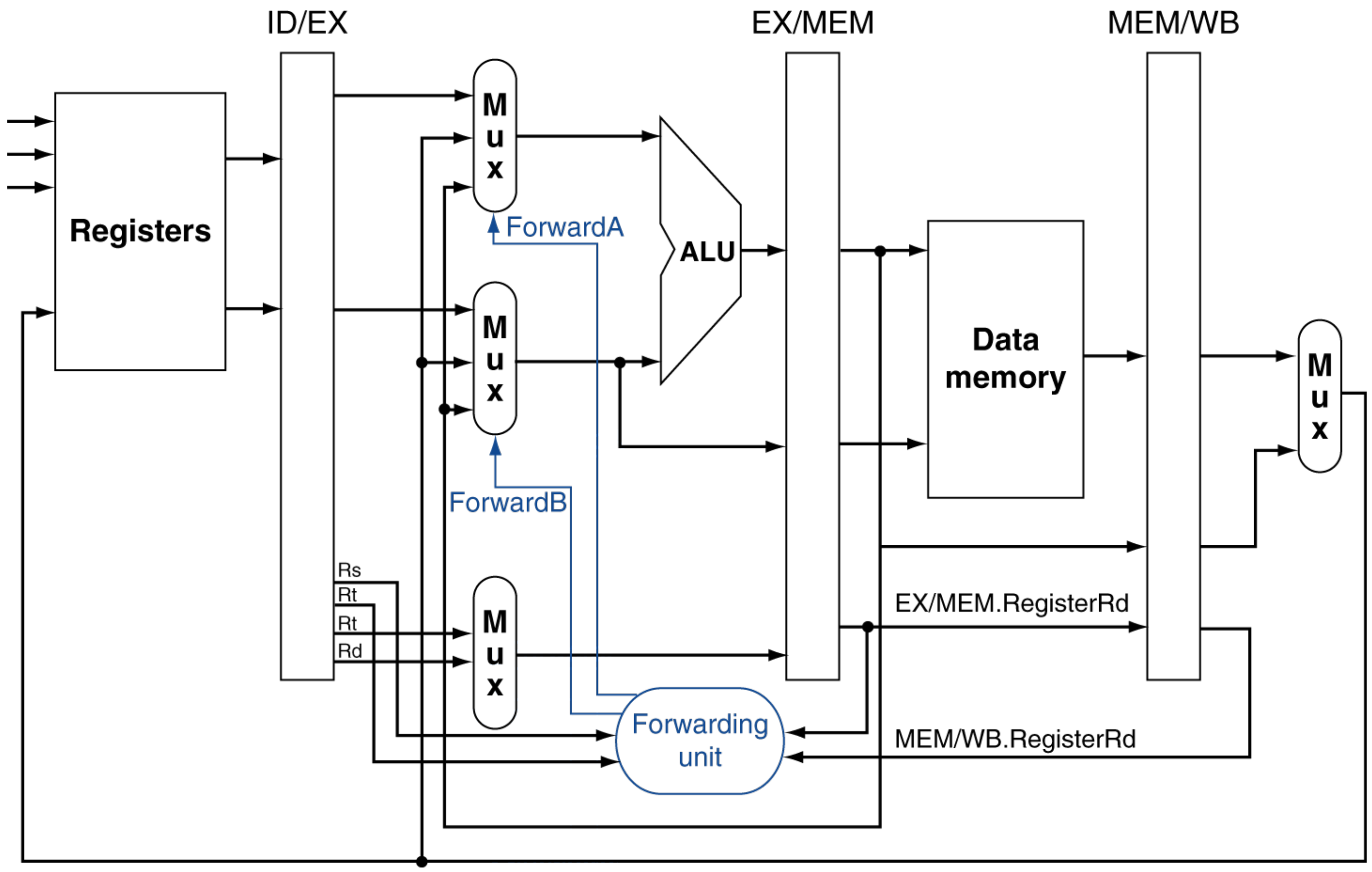
Fwd from EX/MEM pipeline reg (1a, 1b)

Fwd from MEM/WB pipeline reg (2a, 2b)

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 ≠ 0,
    MEM/WB.RegisterRd ≠ 0

Forwarding Paths



b. With forwarding

Forwarding Conditions

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

Double Data Hazard

- Consider the sequence:

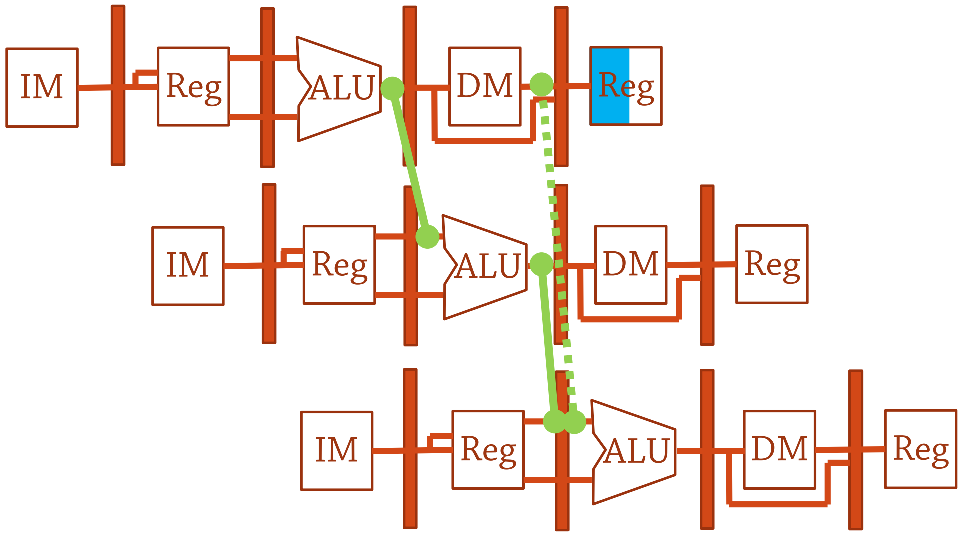
```
add $t1,$t2,$t3
add $t1,$t1,$t4
add $t1,$t1,$t5
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only forward if EX hazard condition is not true

Dependencies & Forwarding

- How do we detect when to forward?

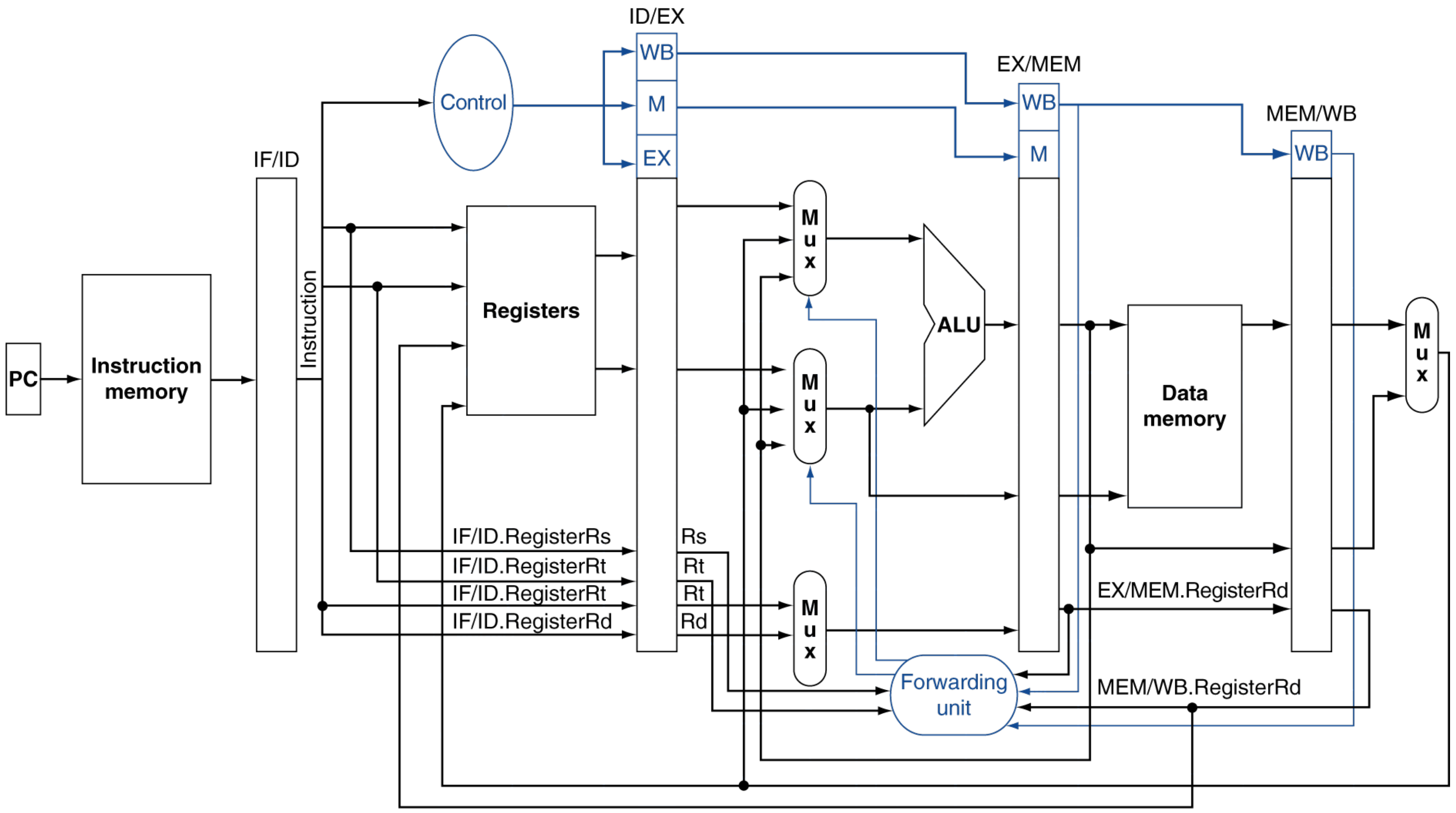
```
add $t1, $t2, $t3

add $t1, $t1, $t4

add $t1, $t1, $t5
```



Revised Forwarding Condition

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

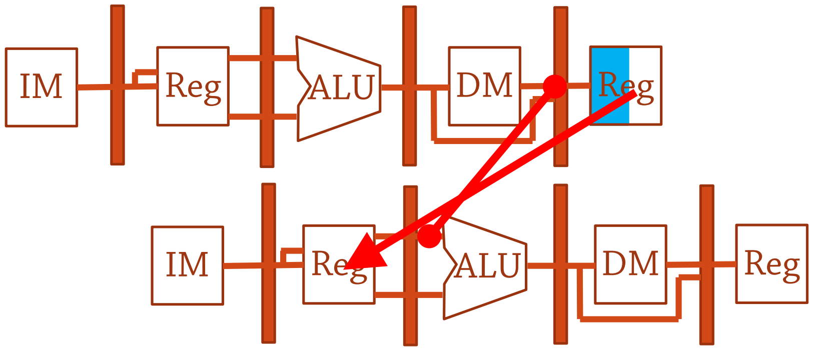
Datapath with Forwarding



Load-Use Data Hazard

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

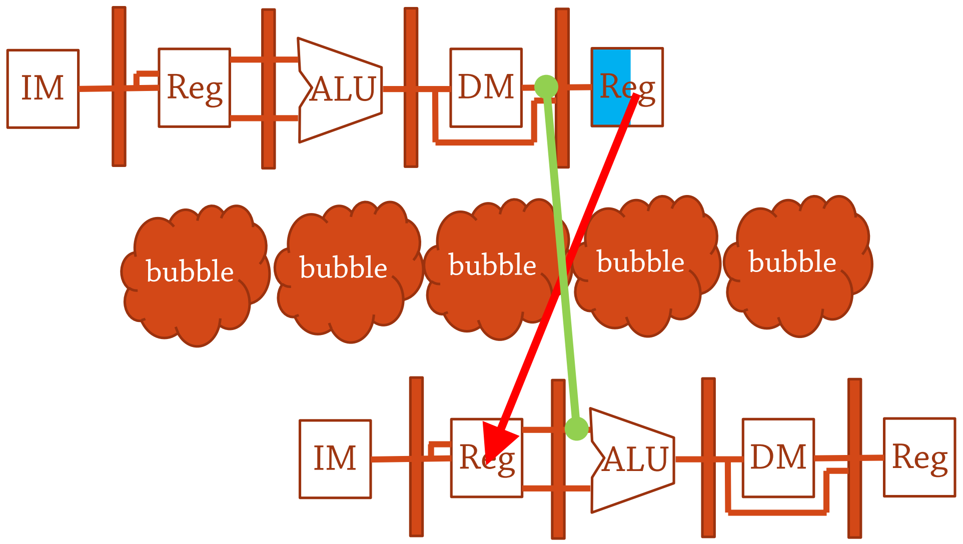
```
lw $s0, 0($t0)

sub $t2, $s0, $t3
```



Load-Use Data Hazard

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

```
lw $s0, 0($t0)

nop

sub $t2, $s0, $t3
```



Load-Use Hazard Detection

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

How to Stall the Pipeline

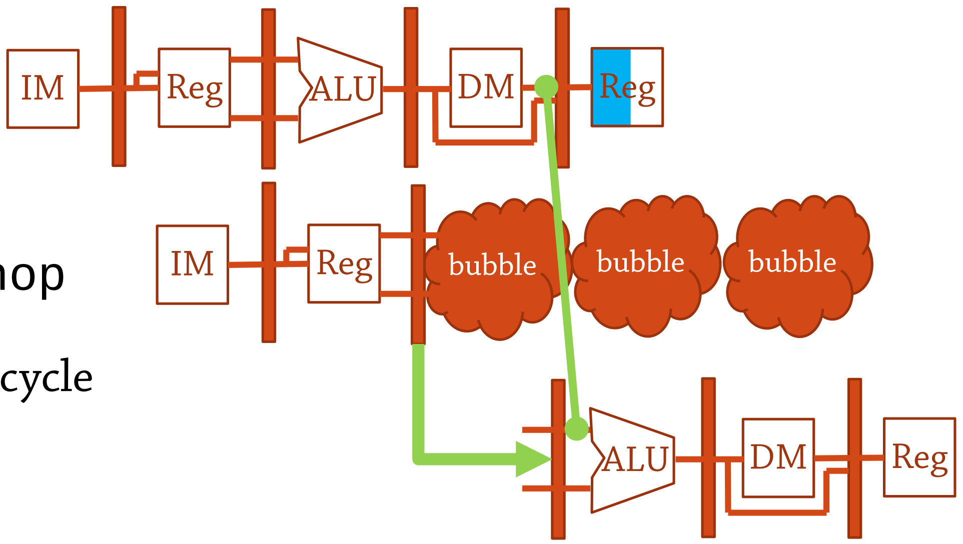
- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

Load-Use Data Hazard

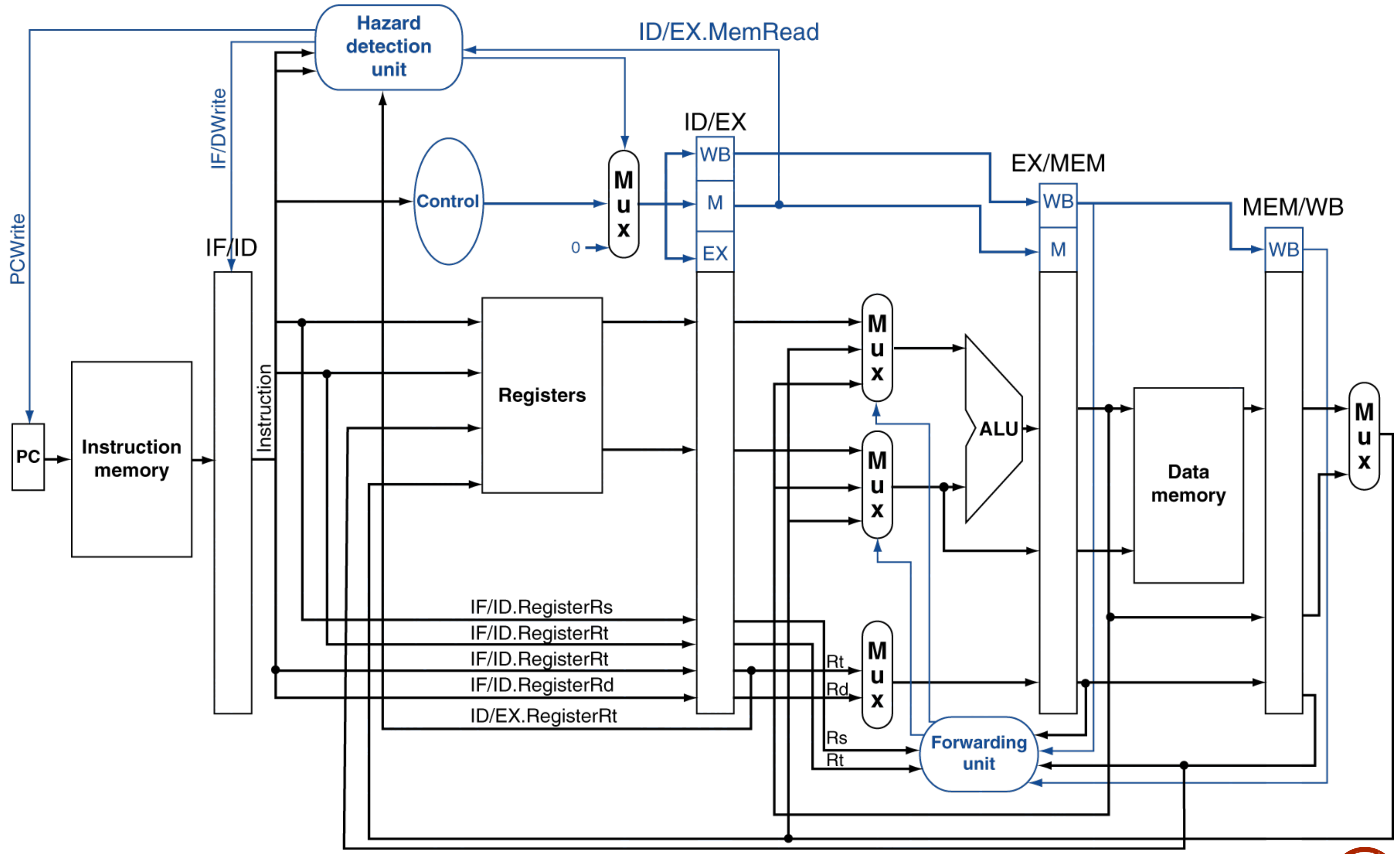
- Can not always avoid stalls by forwarding
  - If value not computed when needed
  - Can not forward backward in time!

```
lw $s0, 0($t0)

sub $t2, $s0, $t3 → nop

sub $t2, $s0, $t3
```

Pause for one cycle

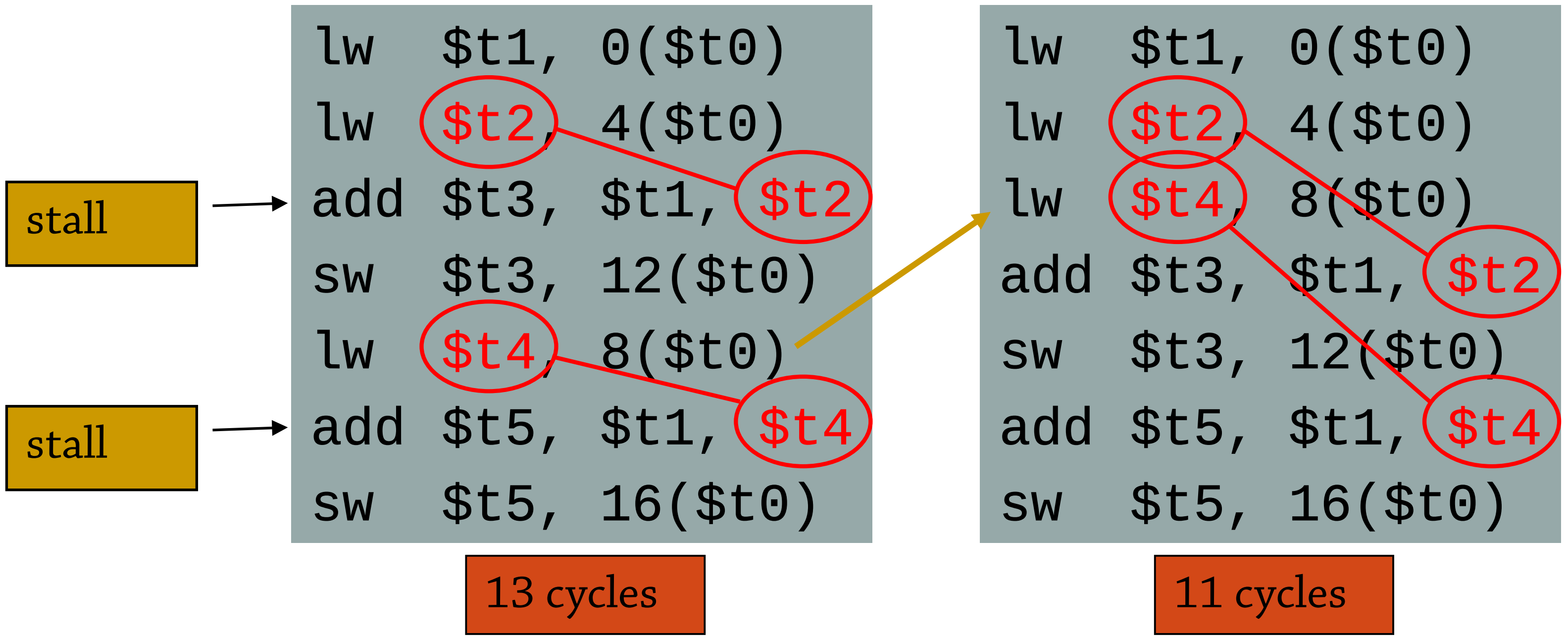


Datapath with Hazard Detection



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;



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

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

Stalls and Performance

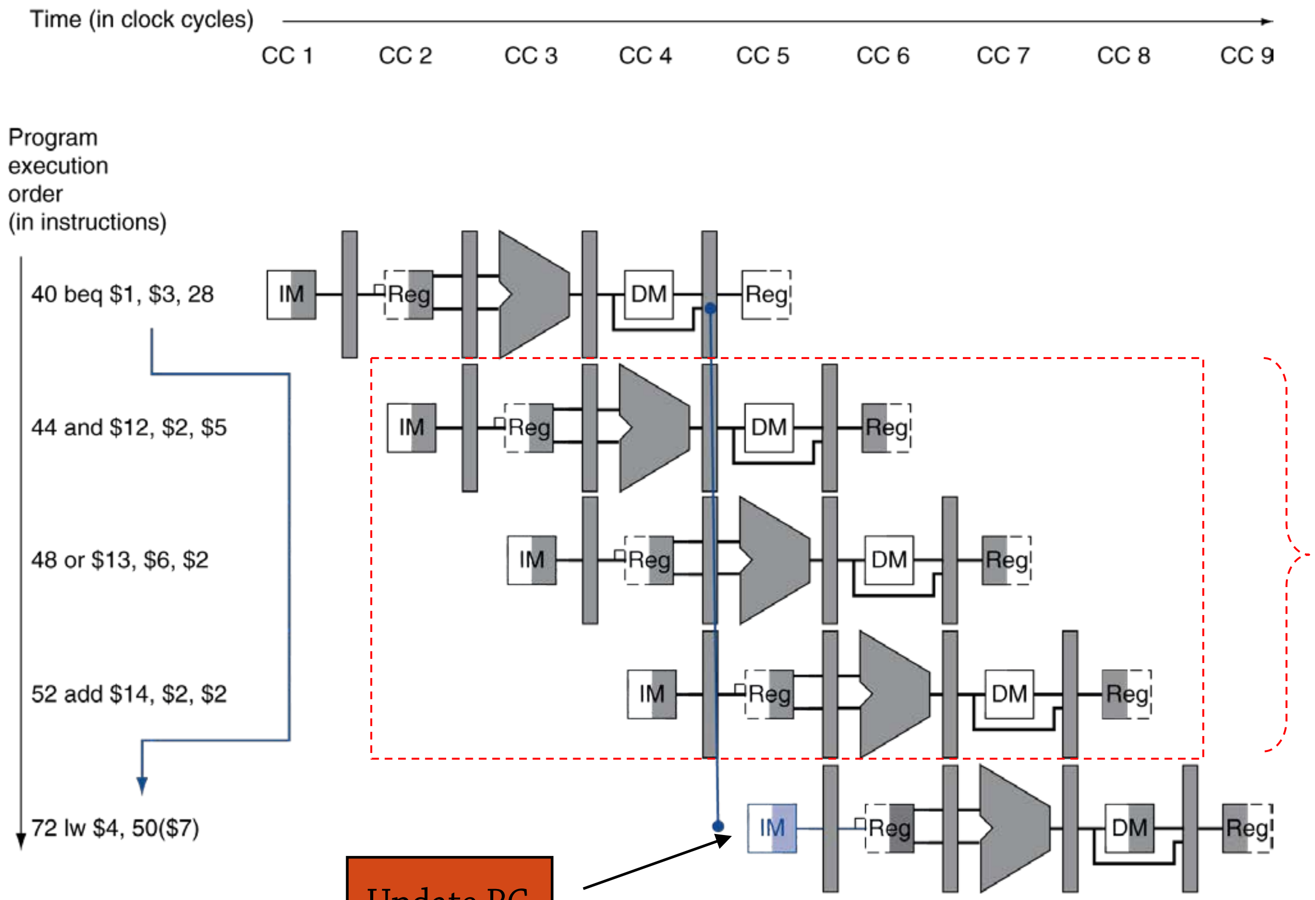
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction

Branch Hazards

- If branch outcome determined in MEM



Flush these instructions (Set control values to 0)

Update PC

Branch Hazards

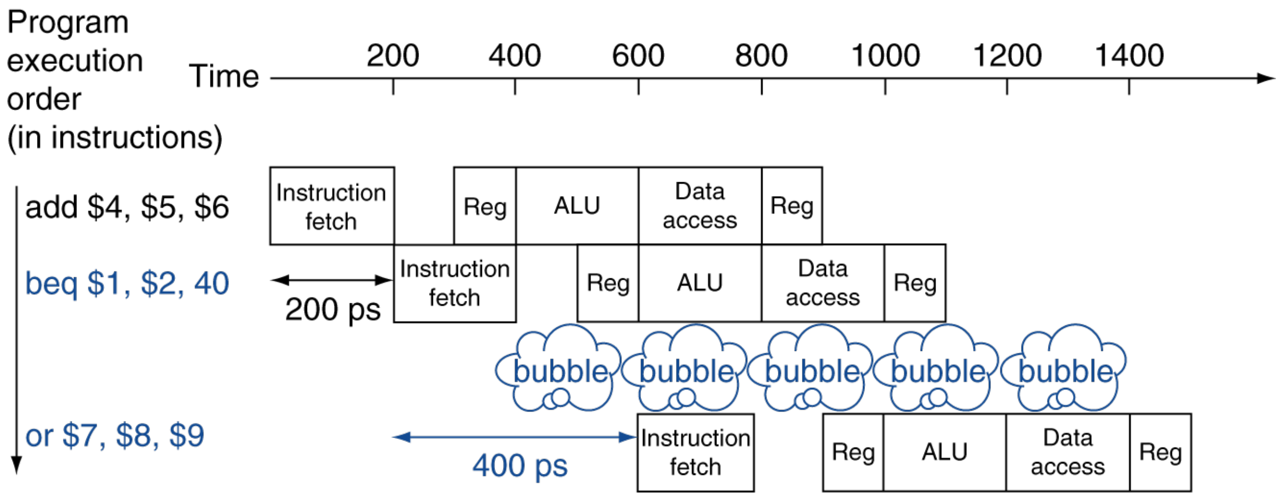
- When the flow of instruction addresses is not sequential
  - Unconditional branches (`j, jal, jr`)
  - Conditional branches (`beq, bne`)
  - Exceptions
- Possible approaches
  - Stall (impacts CPI)
  - Move decision point as early in the pipeline as possible
  - Delay decision (requires compiler support)
  - Predict and hope for the best!
- Control hazards occur less frequently than data hazards, but there is nothing as effective against control hazards as forwarding is for data hazards

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

  Conservative approach: Stall immediately after fetching a branch, wait until outcome of branch is known and fetch branch address.

- Extra HW (cmp and adder for address in ID stage)



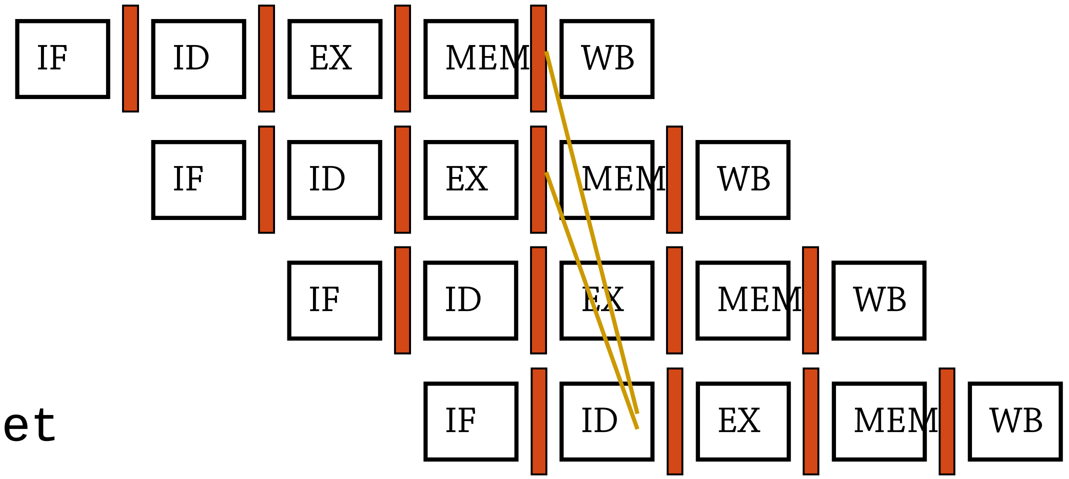
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - → Reduce cost of the taken branch
  - Target address adder
  - Register comparator

Data Hazards for Branches

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

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

add $4, $s1, $6

…

beq $1, $4, target
```

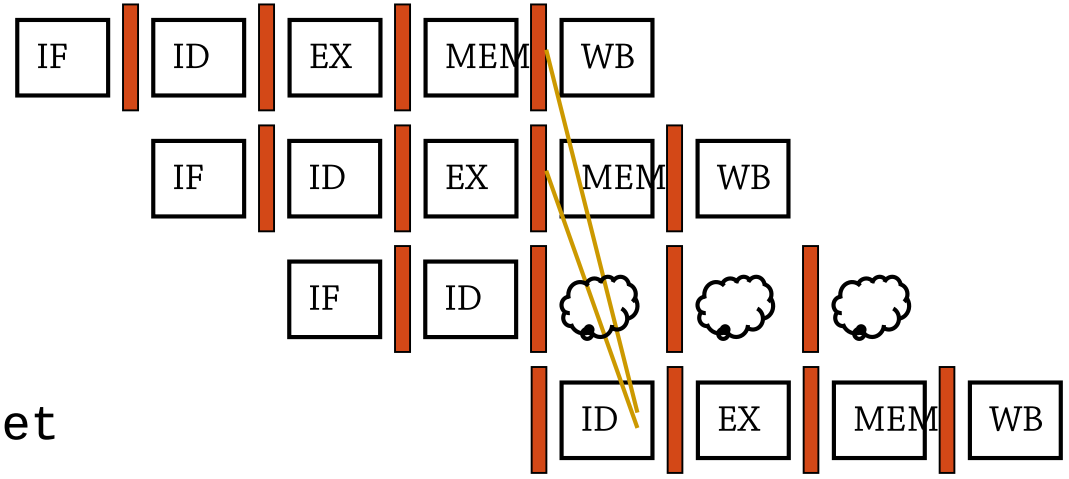


- Can resolve using forwarding

Data Hazards for Branches

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

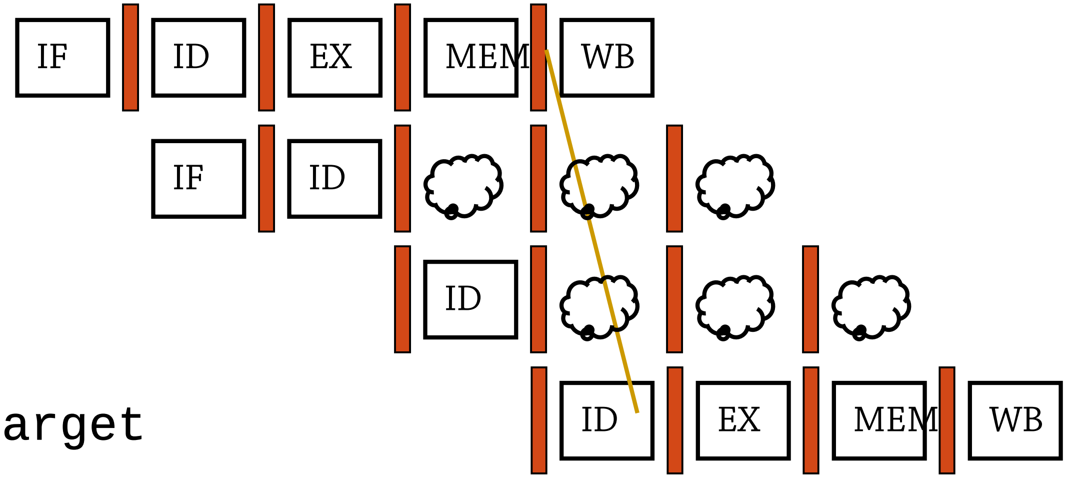
```
lw  $1, addr
add $4, $s1, $6
beq stalled
beq $1, $4, target
```



Data Hazards for Branches

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

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $zero, target
```

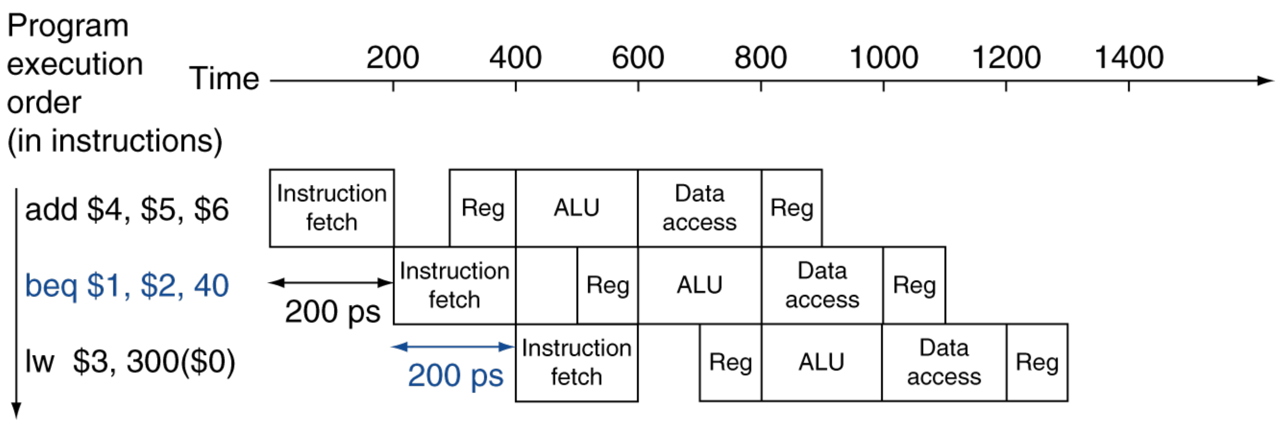


Branch Prediction

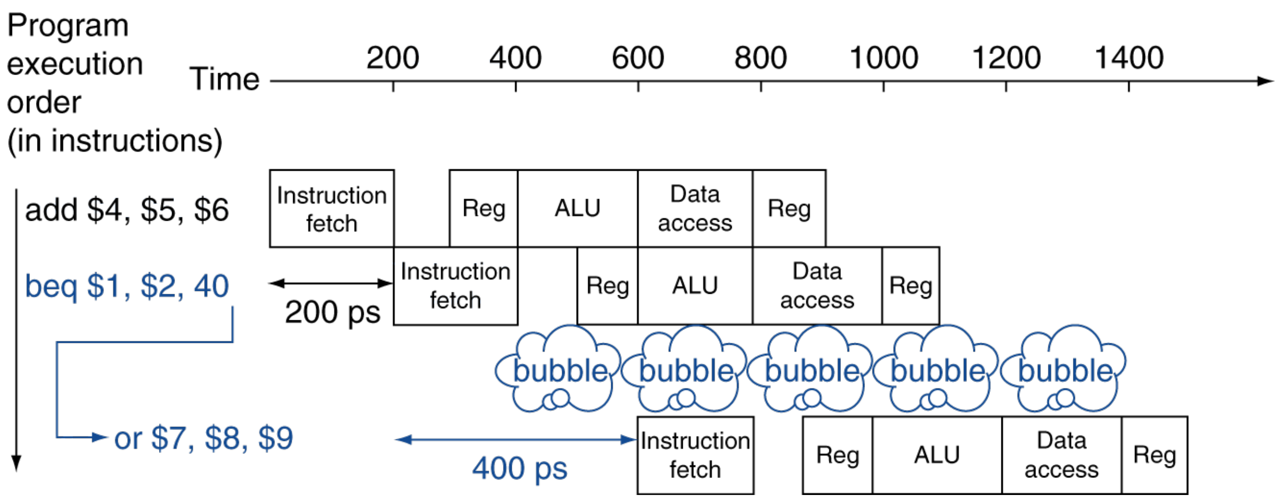
- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

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

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

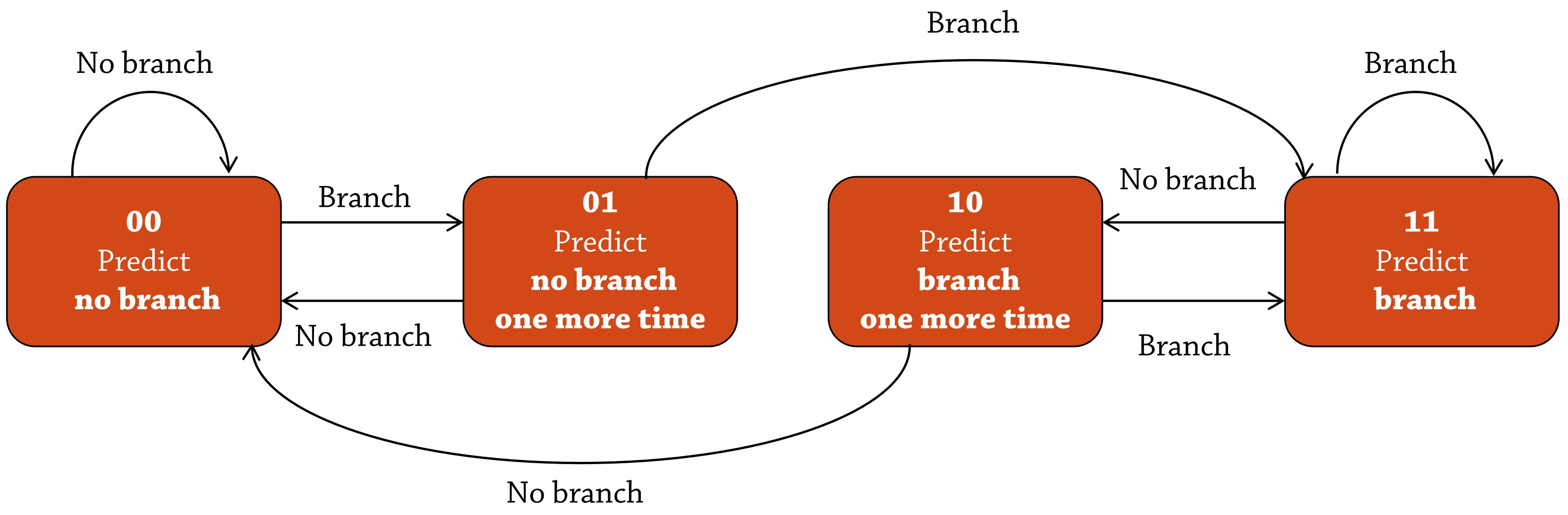
- A 1-bit predictor will be incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code
  - First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  - As long as branch is taken (looping), prediction is correct
  - Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

```
Loop: 1st loop instr
      2nd loop instr
           .
           .
           .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

2-Bit Predictor

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed
  - One bit for what the branch is supposed to be
  - One bit for what it did last time



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Or 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
- Jump to handler

Instruction-Level Parallelism (ILP)

- Pipelining:
  - Executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1 → use Instructions Per Cycle (IPC)
    - e.g. 4-way multiple-issue → peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue (at compile time)
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards
- Dynamic multiple issue (during execution)
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

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, execute instructions after branch
    - Roll back if path taken is different
  - Speculate on store that precedes load doesn't refer to same address
    - Roll back if location is updated

Compiler or Hardware Speculation

- Compiler can reorder instructions
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed (written to the registers or memory)
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

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

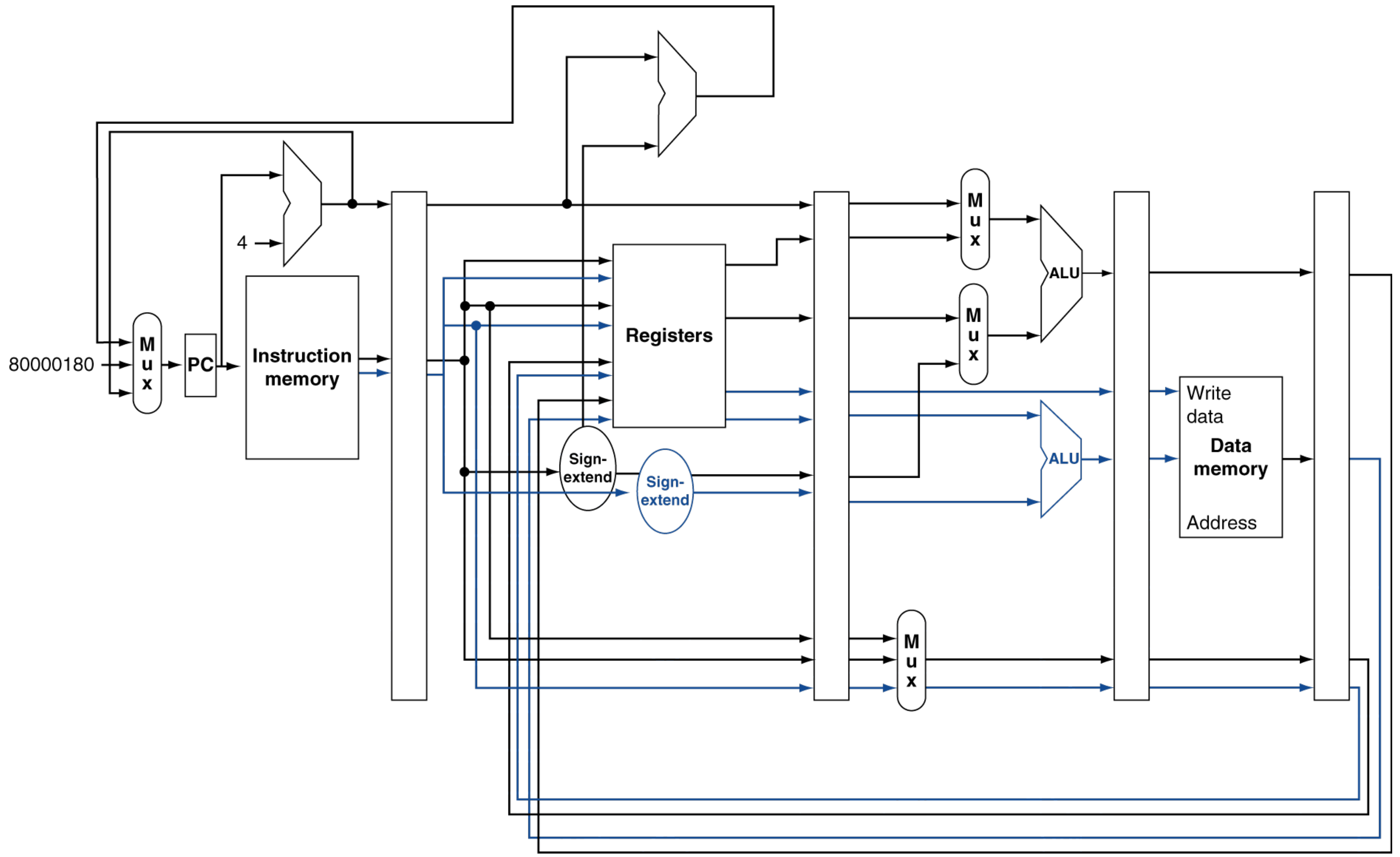
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
  - Pad with nop if necessary

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

MIPS with Static Dual Issue



Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- Execution 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
  - Compiler applies “register renaming” to eliminate all data dependencies that are not true data dependencies

Unrolled Code Example

```
lp:   lw $t0,0($s1)      # $t0=array element
      lw $t1,-4($s1)     # $t1=array element
      lw $t2,-8($s1)     # $t2=array element
      lw $t3,-12($s1)    # $t3=array element
      addu  $t0,$t0,$s2    # add scalar in $s2
      addu  $t1,$t1,$s2    # add scalar in $s2
      addu  $t2,$t2,$s2    # add scalar in $s2
      addu  $t3,$t3,$s2    # add scalar in $s2
      sw $t0,0($s1)      # store result
      sw $t1,-4($s1)     # store result
      sw $t2,-8($s1)     # store result
      sw $t3,-12($s1)    # store result
      addi  $s1,$s1,-16    # decrement pointer
      bne   $s1,$zero,lp      # branch if $s1 != 0
```

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

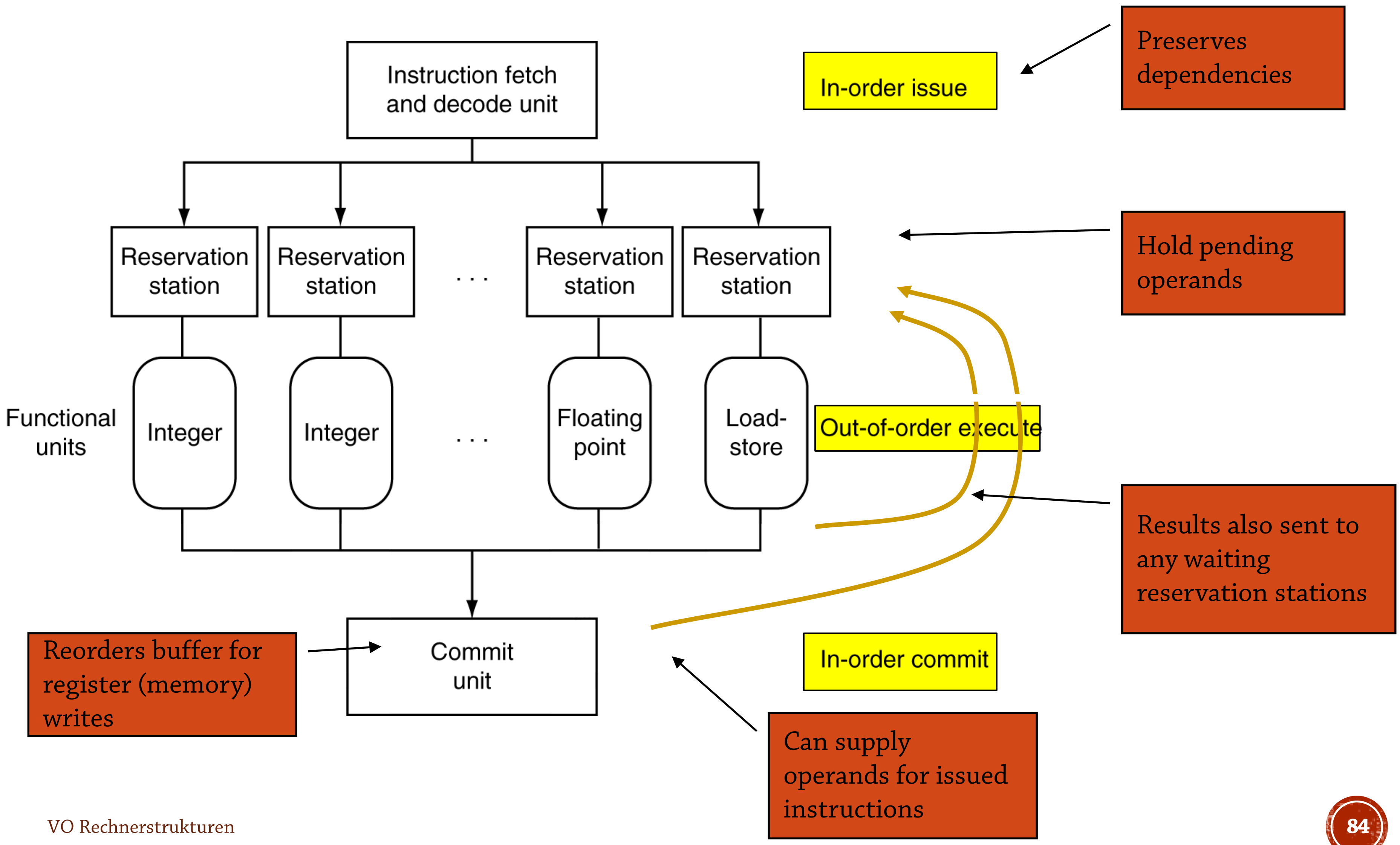
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
      lw    $t0, 20($s2)
      addu  $t1, $t0, $t2
      sub   $s4, $s4, $t3
      slti  $t5, $s4, 20
```

  - Can start `sub` while `addu` is waiting for `lw`

Dynamically Scheduled CPU



Preserves dependencies

Hold pending operands

Results also sent to any waiting reservation stations

Reorders buffer for register (memory) writes

Can supply operands for issued instructions

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - 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?

- Yes, but not as much as we'd like:
  - Programs have real dependencies that limit ILP
  - Some dependencies are hard to eliminate
  - Some parallelism is hard to expose
  - Memory delays and limited bandwidth, hard to keep pipelines full
  - Speculation can help if done well

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
  - Today, power concerns lead to less aggressive designs

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets
    (e.g. IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes

Concluding Remarks

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

EXAMPLES

Control Hazards

Consider a five stage pipeline. Unconditional jumps are calculated at the end of the second stage, while the target of conditional jumps is only known at the end of stage three. The jump prediction is implemented using the predict-not-taken strategy and the CPU has an ideal CPI of one. The frequency of conditional jumps is 20%, of unconditional jumps five percent and in 60% of the cases the conditional jumps are taken.

How much faster would the machine be without branch hazards?

Control Hazards

- Conditional jumps:
  - 20% of all instructions are conditional jumps
  - 60% of the predictions are incorrect
  - Costs for erroneous prediction: 2 cycles
- Unconditional jumps:
  - 5% of all instructions are unconditional jumps
  - All predictions are wrong
  - Costs for erroneous prediction: 1 cycles

Control Hazards

- Execution time without branch hazards:
  - $ET_{ideal} = N * CPI_{ideal} * T_{clk}$ $CPI_{ideal} = 1$
- Execution time including branch hazards:
  - $ET_{real} = N * CPI_{real} * T_{clk}$
  - $CPI_{real} = 0.75 + 0.2 * (0.4 * 1 + 0.6 * 3) + 0.05 * 2 = 1.29$
- Performance comparison:
  - $ET_{real} / ET_{ideal} = CPI_{real} / CPI_{ideal} = 1.29$

➔ The CPU would be 29% faster.

Instruction Scheduling

Assume a MIPS-processor with a five stage pipeline. There is no forwarding implemented and jump instructions are executed in stage 2.

Add `nop` instructions to the following program such that it is executed correctly. Optimize the program afterwards to minimize the overhead.

```
      ori  $s4,$zero,12
      or   $s1,$zero,$zero
sum:  lw   $t2,0($s4)
      add  $s1,$s1,$t2
      addi $s4,$s4,-4
      bne  $s4,$zero,sum
```

Instruction Scheduling

Inserting nop-instructions:

```
        ori    $s4,$zero,12
        or     $s1,$zero,$zero
        nop
sum:    lw     $t2,0($s4)
        nop
        nop
        add    $s1,$s1,$t2
        addi   $s4,$s4,-4
        nop
        nop
        bne    $s4,$zero,sum
```

Optimized code:

```
        ori    $s4,$zero,12
        or     $s1,$zero,$zero
        nop
sum:    lw     $t2,0($s4)
        addi   $s4,$s4,-4
        nop
        add    $s1,$s1,$t2
        bne    $s4,$zero,sum
```

Load-Delay-Slot

Reorder the following MIPS-code to minimize pipeline stalls. Forwarding is implemented and load-instructions are executed in stage four (→ one load-delay slot).

```
lw   $t1,0($t2)
lw   $t3,4($t2)
add  $t4,$t1,$t3
add  $t4,$t4,$s2
lw   $s1,0($t6)
lw   $t7,4($t6)
sub  $t8,$s1,$t7
sw   $t8,8($t6)
```

Load-Delay-Slot

- Reordered code:

```
lw   $t1,0($t2)
lw   $t3,4($t2)
lw   $s1,0($t6)
add  $t4,$t1,$t3
lw   $t7,4($t6)
add  $t4,$t4,$s2
sub  $t8,$s1,$t7
sw   $t8,8($t6)
```

Pipeline Diagram

Draw the pipeline diagram for the following program:

```
    ori  $s4,$zero,20
lp: lw   $t2,0($s4)
    mult $s1,$s1,$t2
    addi $s4,$s4,-4
    bne  $s4,$zero,lp
    nop
```

Forwarding is not implemented and the multiplication takes three cycles in stage EX. Branch targets are calculated in stage ID. If there are any RAW-dependencies, the pipeline is stalled until data is available. Assume that there the loop is executed five times.

Pipeline Diagram

```
    ori $s4,$zero,20  IF ID EX ME WB
Lp: lw  $t2,0($s4)       IF ID st st EX ME WB
    mlt $s1,$s1,$t2         IF st st ID st st EX EX EX ME WB
    addi $s4,$s4,-4                  IF st st ID st st EX ME WB
    bne $s4,$zero,Lp                          IF st st ID st st EX...
    nop                                                IF st st abort
    lw  $t2,0($s4)                                              IF ...
```

# of stalls: 2 + 5 * 3 * 2 cycles

# of control hazards: 4 * 1 cycles

Delayed Branch

- Draw the pipeline diagram of the following program
- How many stalls occur?
- Which value is stored at memory address zero, if the initial values in the memory are:

Address	Value
4	2
8	3
12	4

Delayed Branch

```
        lw   $t1,12($zero)
        addi $t2,$zero,8
loop:   lw   $t3,0($t2)
        addi $t2,$t2,-4
        bne  $t2,$zero,loop
        mult $t1,$t1,$t3
        ori  $t1,$t1,1
        sw   $t1,0($zero)
```

The multiplication needs three cycles in stage EX. Forwarding is implemented. The jump is executed at the end of the ID stage and the processor uses a delay-branch-slot of length one.

Delayed Branch

```
lw   $t1,12($zero)  IF ID EX ME WB
addi $t2,$zero,8       IF ID EX ME WB
lw   $t3,0($t2)           IF ID EX ME WB
addi $t2,$t2,-4              IF ID EX ME WB
bne  $t2,$zero,Loop             IF ID EX ME WB
mult $t1,$t1,$t3                   IF ID EX EX EX ME WB
lw   $t3,0($t2)                       IF ID st st EX ME WB
addi $t2,$t2,-4                          IF st st ID EX ME WB
bne  $t2,$zero,Loop                               IF ID EX ME WB
mult $t1,$t1,$t3                                     IF ID EX EX EX ME
ori  $t1,$t1,1                                          IF ID st st EX
sw   $t1,0($zero)                                          IF st st ID
```

# of stalls: 2 * 2 cycles

Address 0: 25

182.690 RECHNERSTRUKTUREN – PIPELINING

Thomas Polzer
tpolzer@ecs.tuwien.ac.at
Institut für Technische Informatik