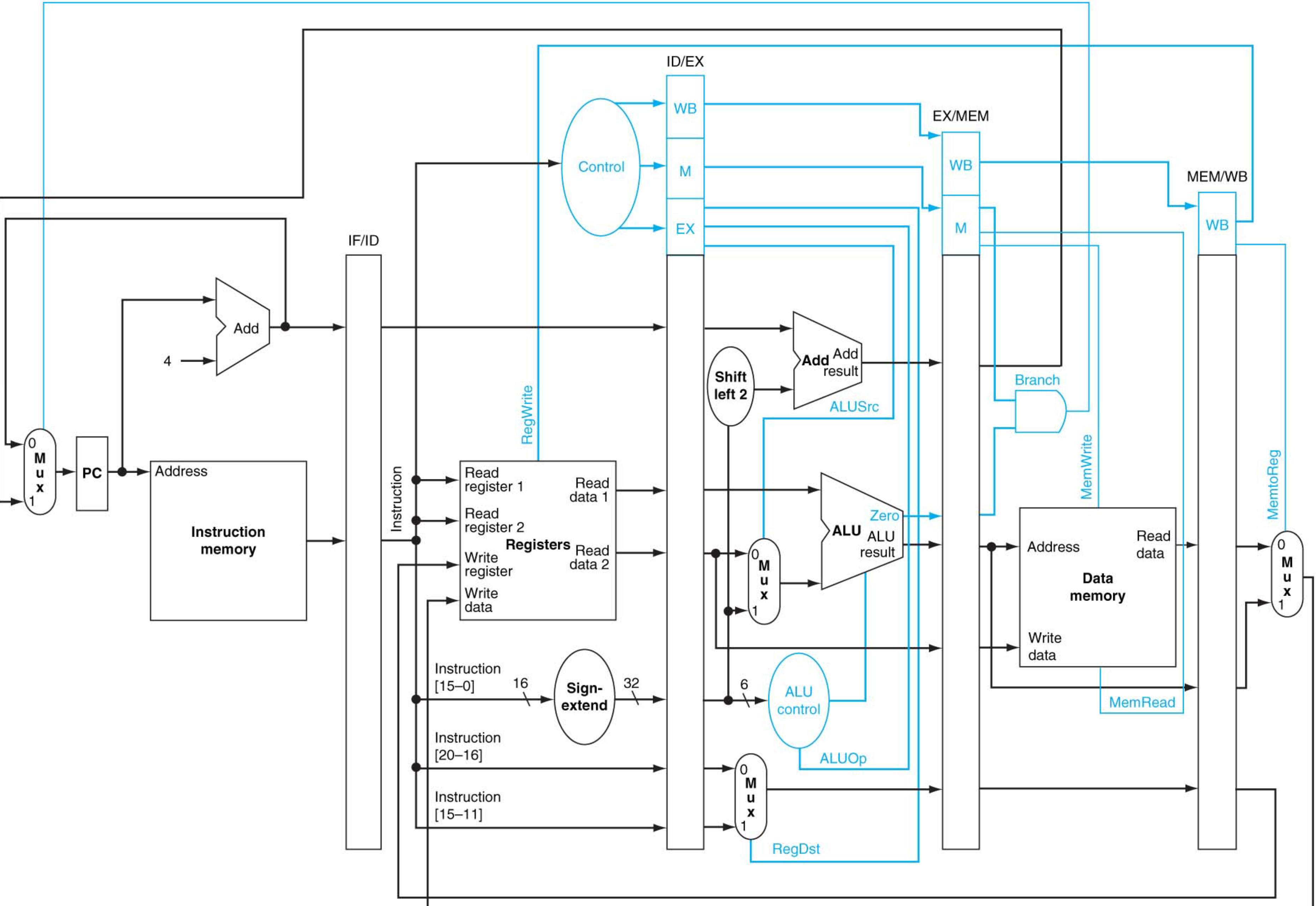
Pipeline

Chapter 4

Reminder



Data Hazard and Forwarding

- Interesting instruction sequence

  SUB $2, $1, $3

  AND $12, $2, $5

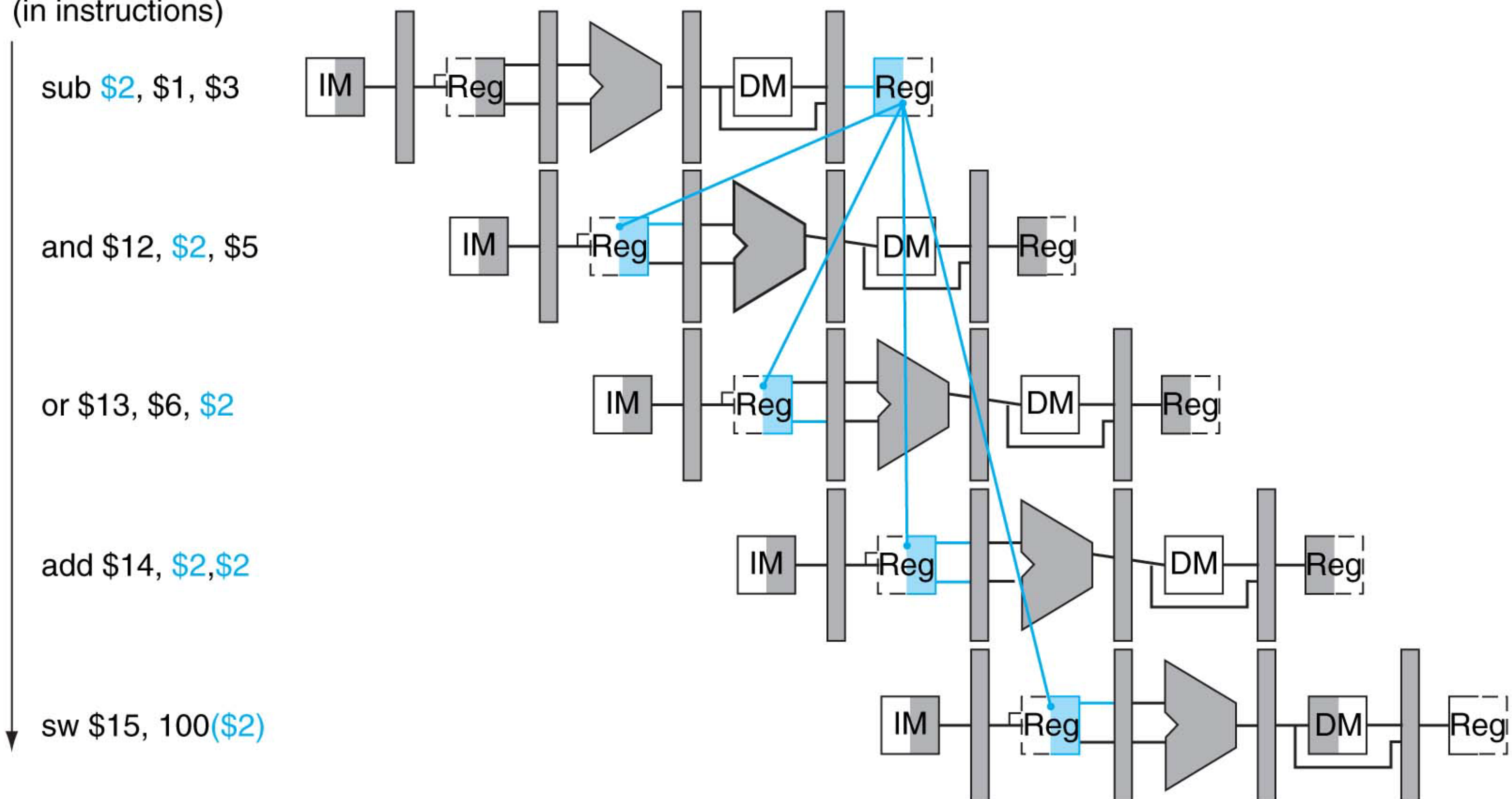
  OR $13, $6, $2

  ADD $14, $2, $2

  SW $15, 100 ($2)

- Last four instructions depend on $2
- Availability of new value after 5th cycle

Usage of the first result



Solution 1

- Solution on compiler level
- Forbid code sequences as given in example
- Insert nop operations
- Result

```
SUB $2, $1, $3
NOP
NOP
AND $12, $2, $5
OR $13, $6, $2
ADD $14, $2, $2
SW $15, 100 ($2)
```

Solution 2

- Detection of hazard
- Forwarding of the result
- Hazard types

    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

- Pipeline register name.register field

Hazards

- First hazard

  SUB $2, $1, $3

  AND $12, $2, $5

- Detectable:
  - And in EX stage and
  - Prior instruction in MEM stage
  - Hazard 1a

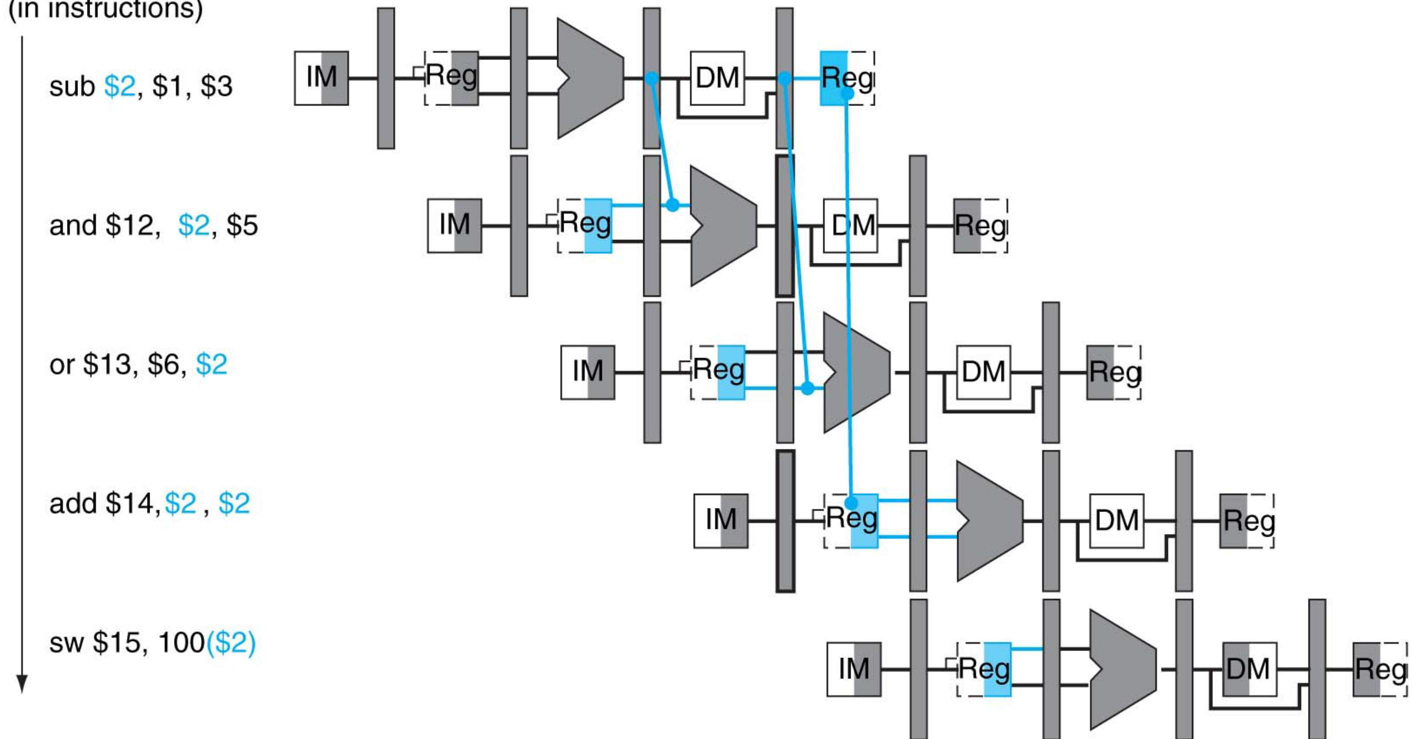
    EX/MEM.Register.Rd = ID/EX.RegisterRs = $2

Dependencies

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

Program execution order (in instructions)



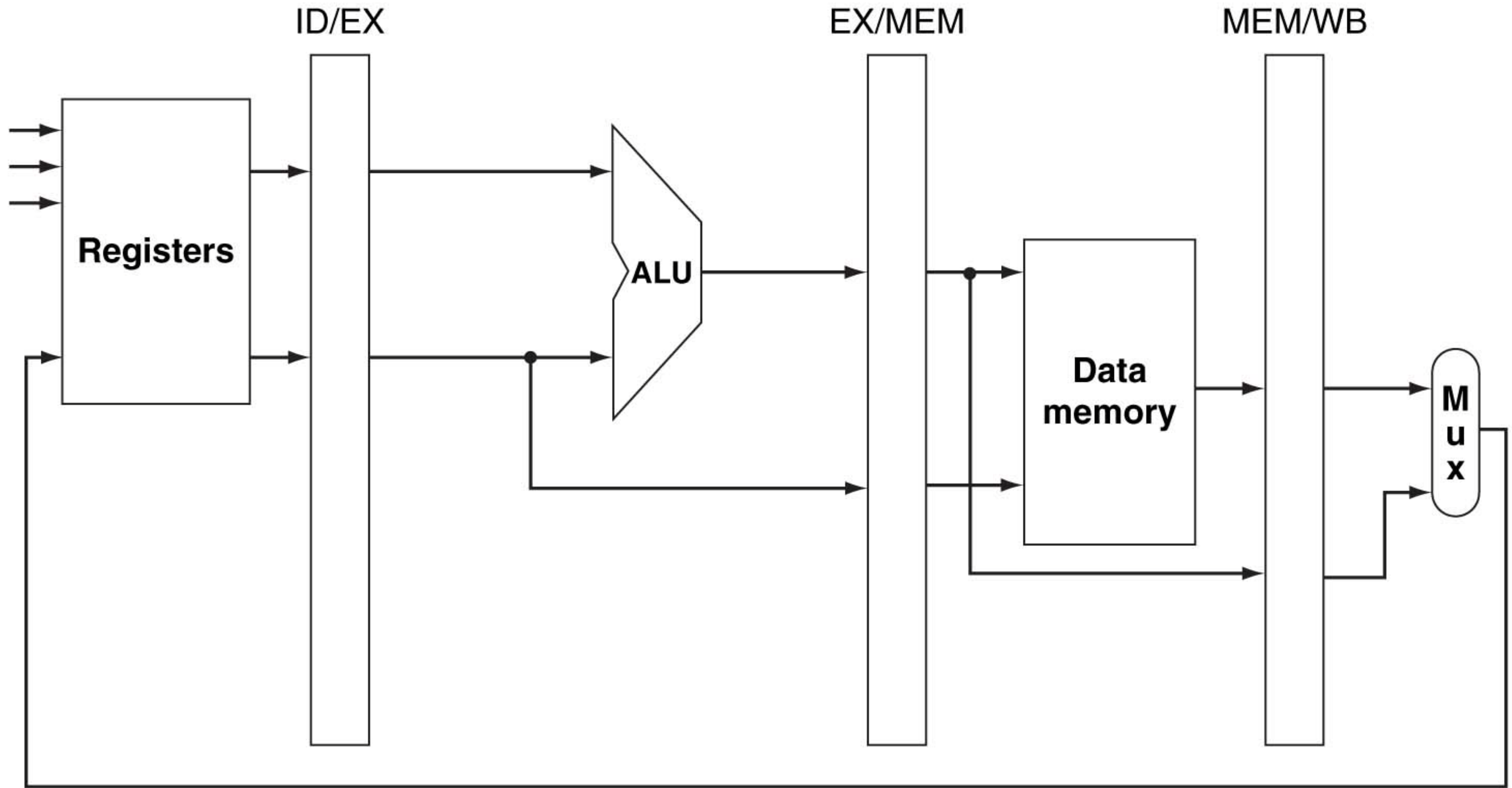
Detection conditions

- 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

Detection conditions

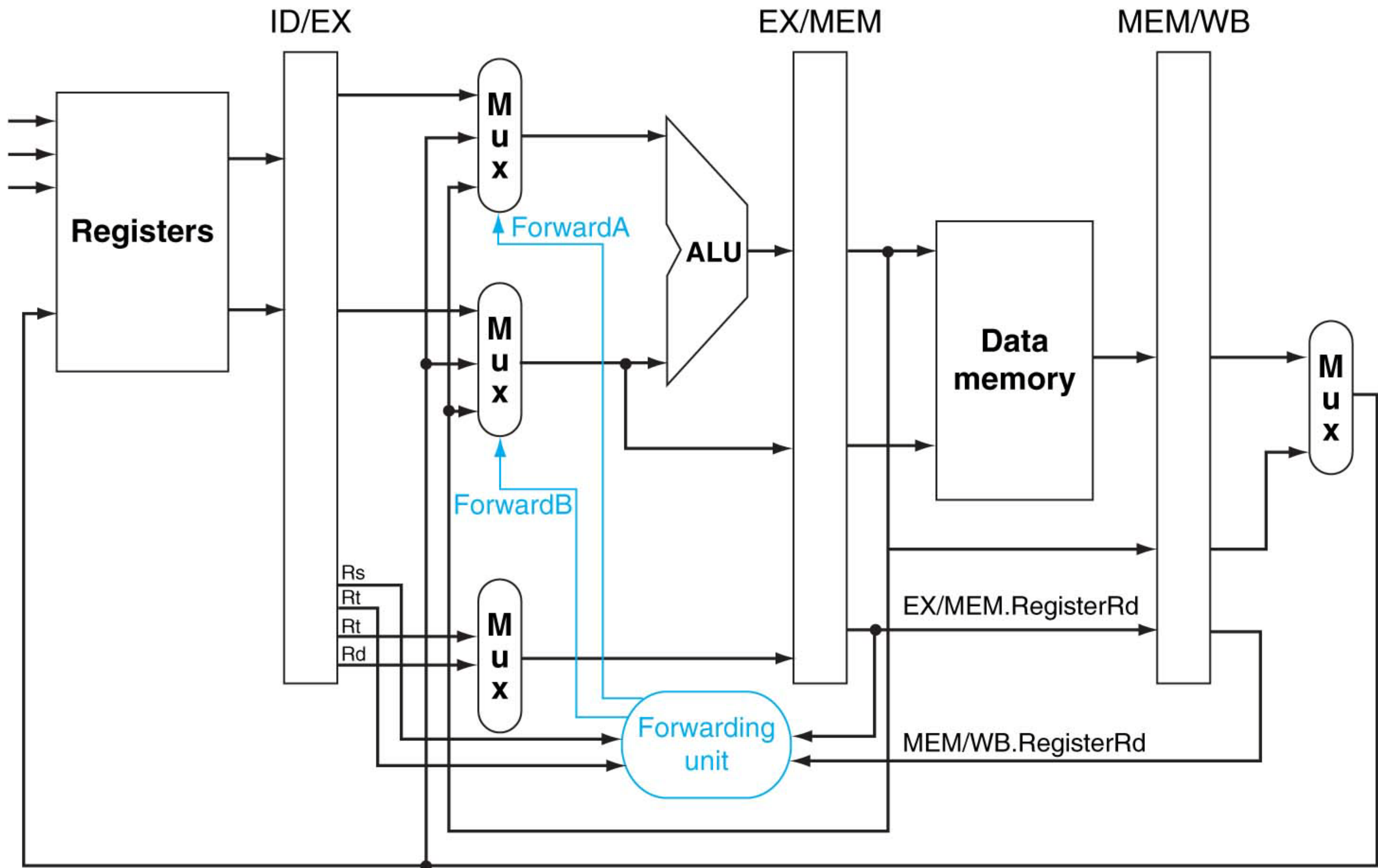
- MEM hazard
  - If (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd <> 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - If (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd <> 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

ALU without forwarding



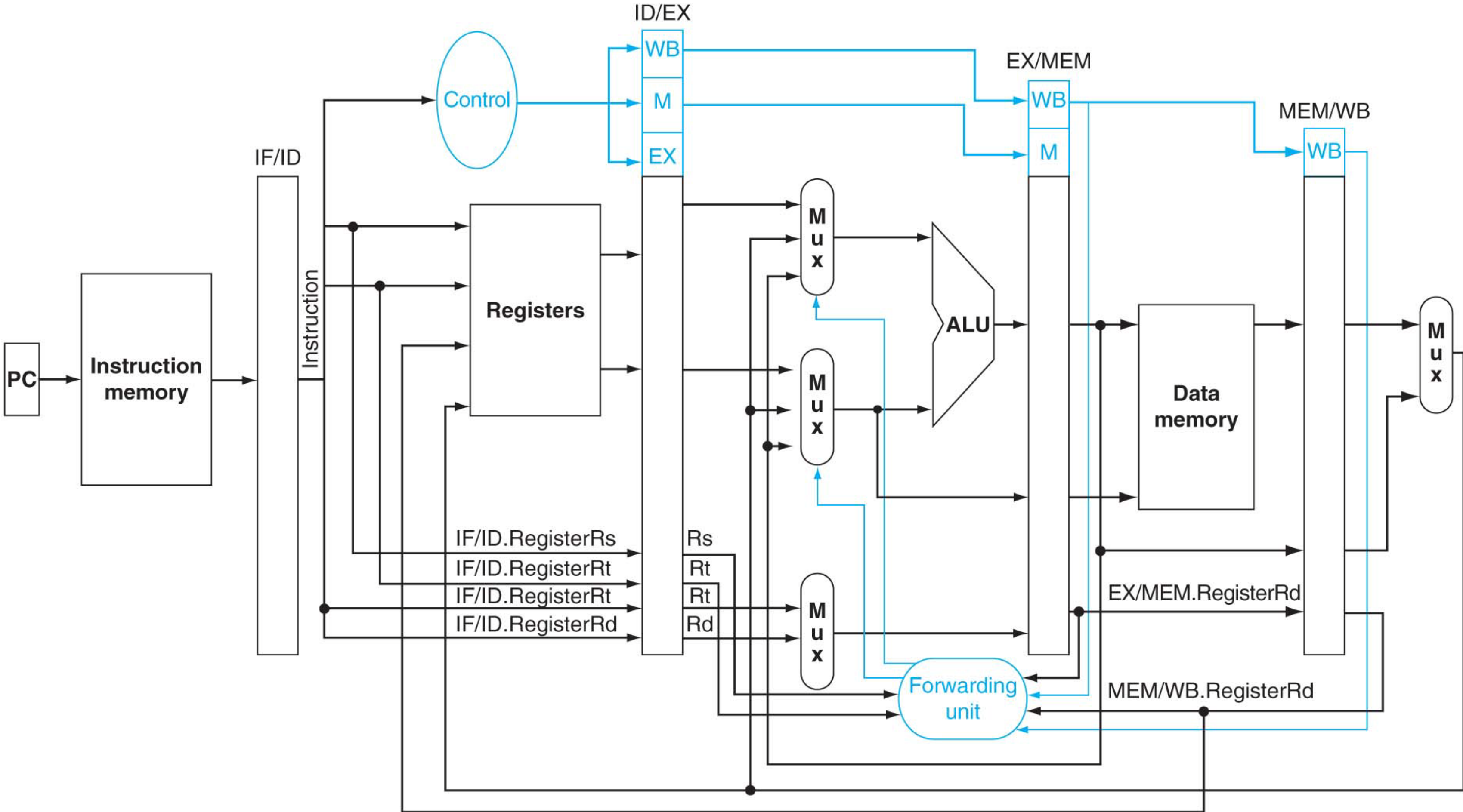
a. No forwarding

ALU without forwarding



b. With forwarding

Datapath with forwarding



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.

Example with forwarding

- Instructions with double dependency

  sub $2, $1, $3

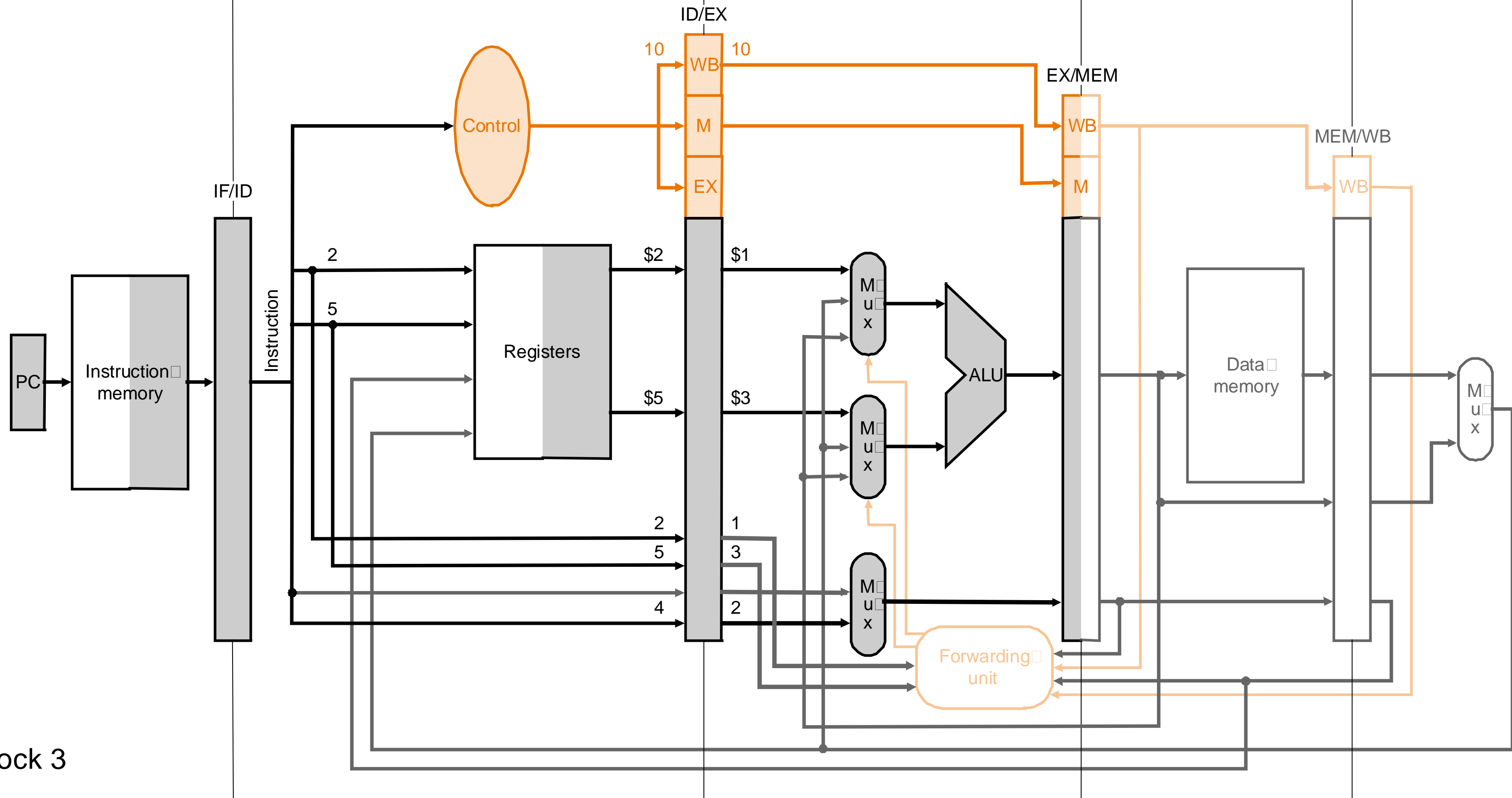
  and $4, $2, $5

  or $4, $4, $2

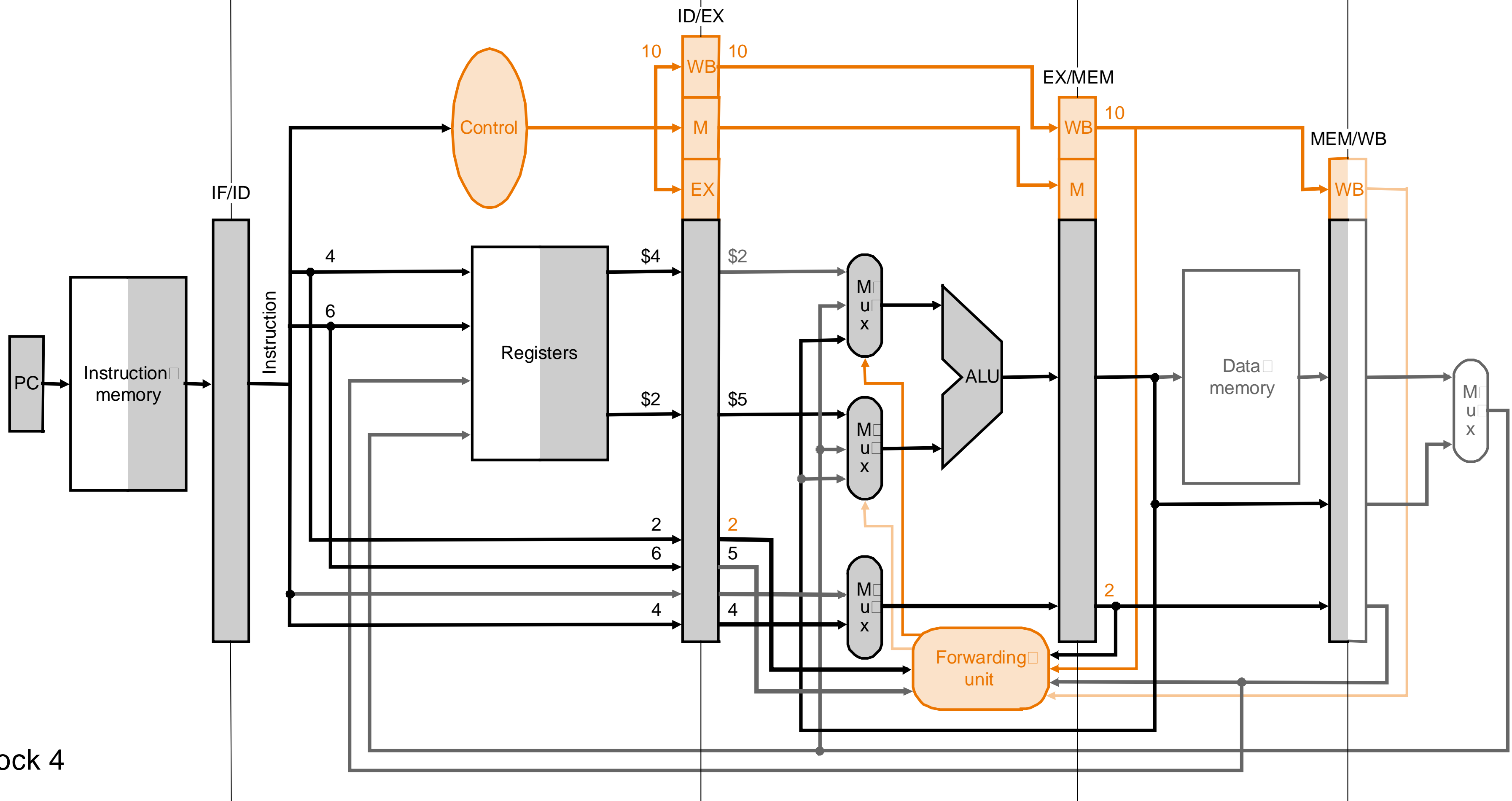
  add $9, $4, $2

- Instructions in execution cycle
  - Cycle 3: sub
  - Cycle 4: and
  - Cycle 5: or
  - Cycle 5: add

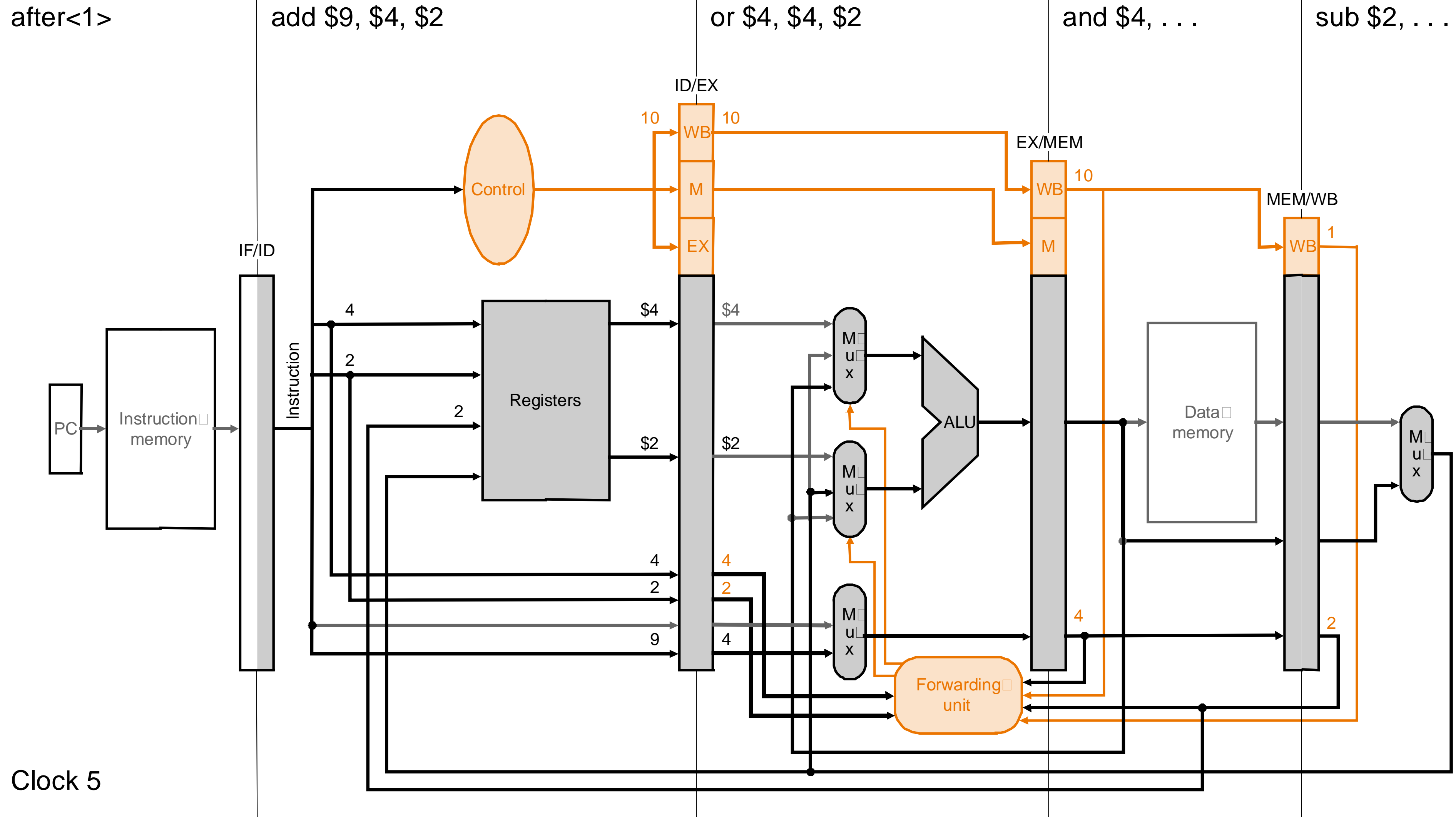
Example with forwarding



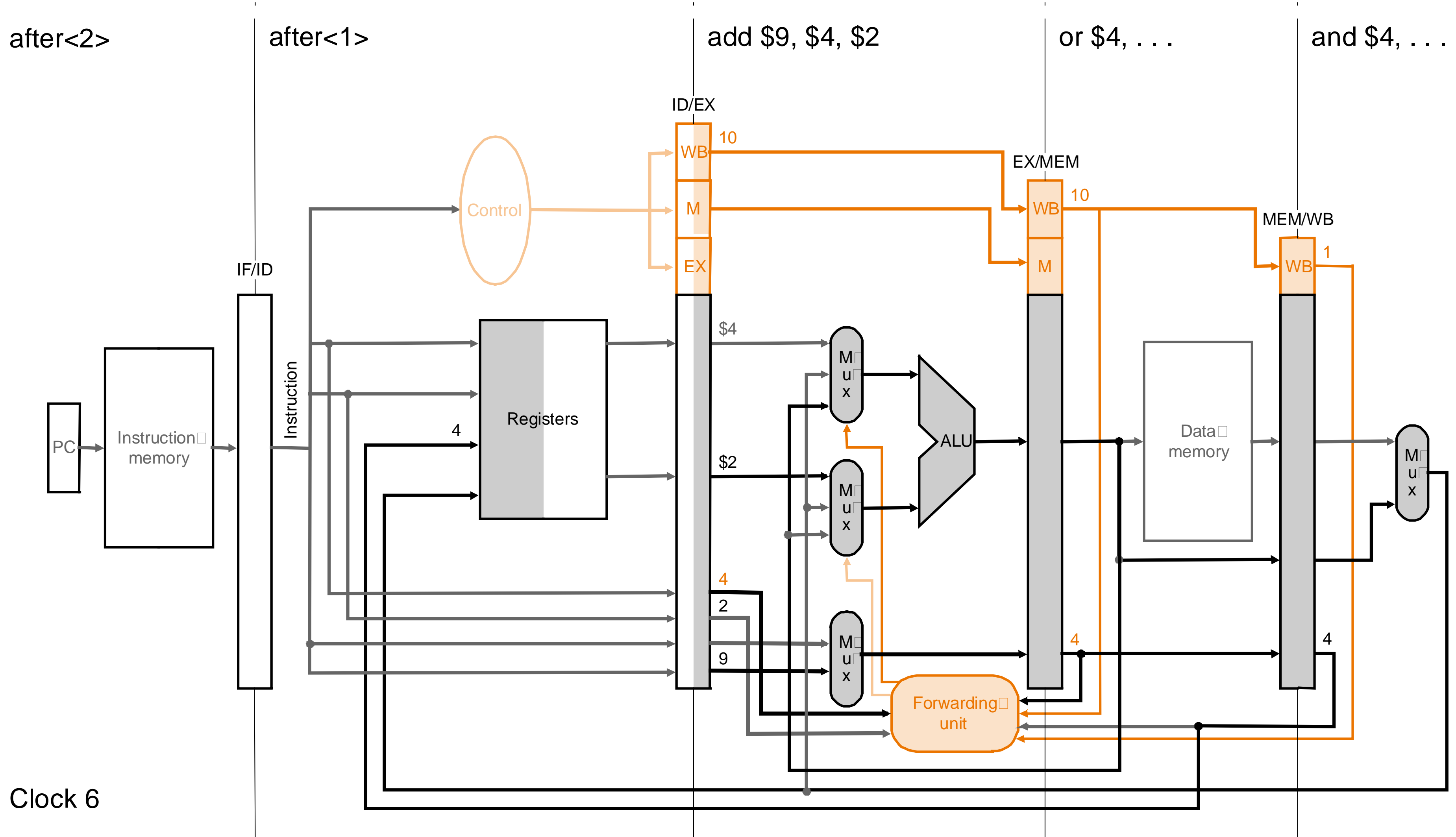
Example with forwarding



Example with forwarding



Example with forwarding



Summary

- Pipelining
  - Data
  - Control signals
  - Register references
- Forwarding
  - Detect a hazard
  - Assemble the operands from
    - Register
    - EX/MEM register
    - MEM/WB register

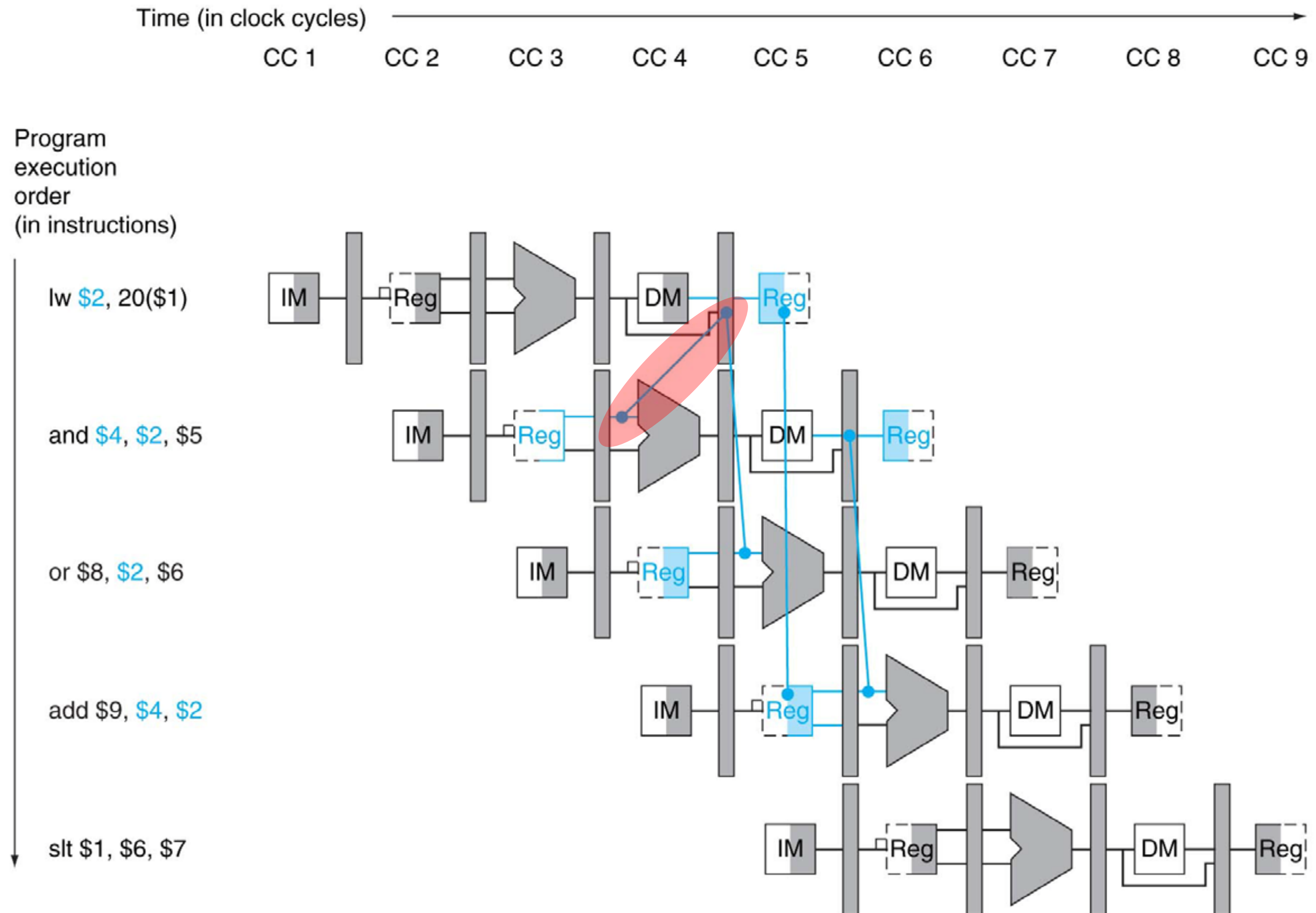
Data hazard and stalls

- Avoiding of hazards
  - Code reordering
  - Forwarding
- One problem is left over

```
lw $5, 100 ($4)
add $6, $7, $5
```

  - lw writes not before cycle 5 -> data hazard!
  - The hazard has to be detected and the pipeline to be stalled.

Pipeline with problem



Data hazard

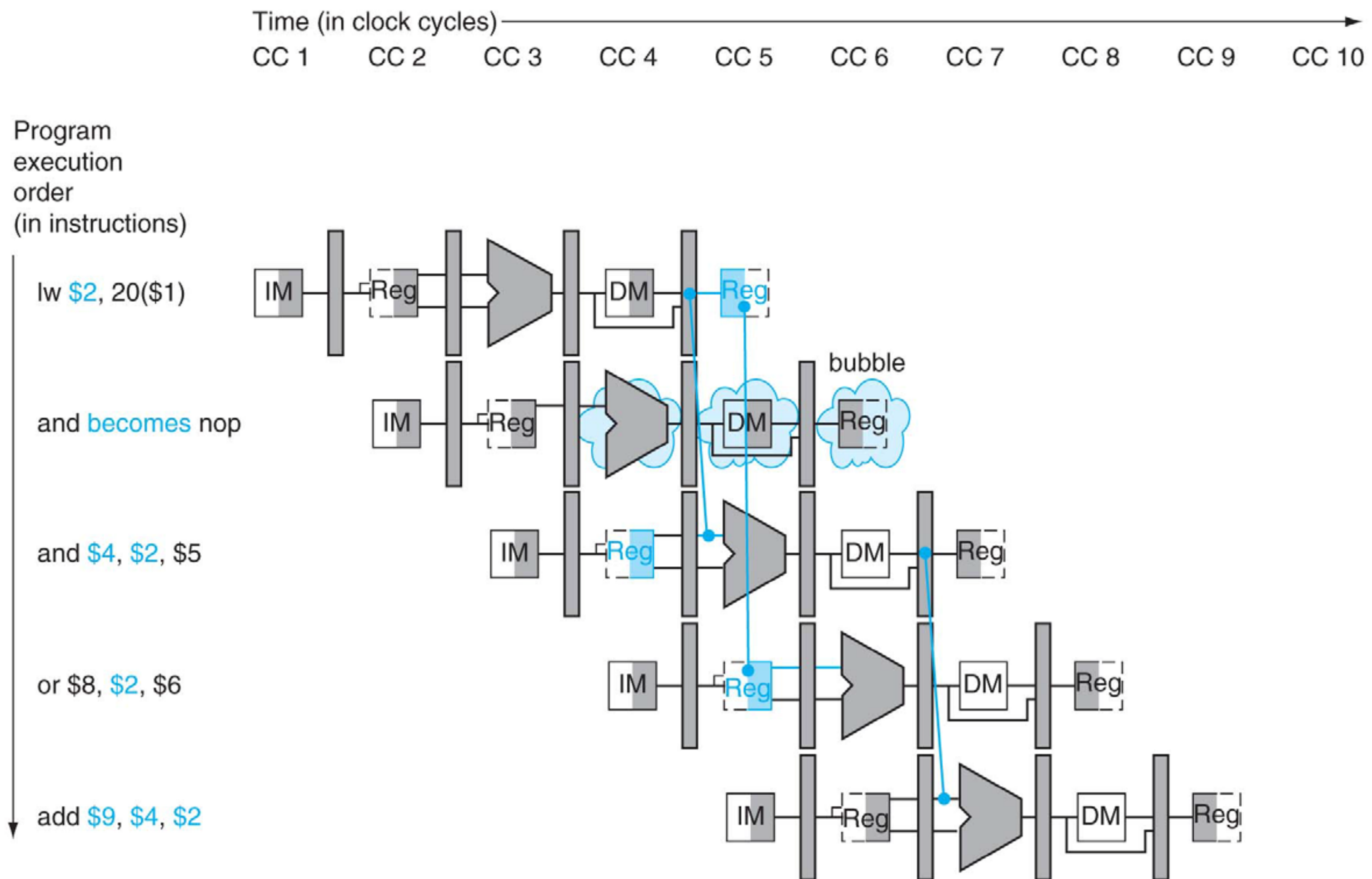
- Conditions for the hazard detection

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

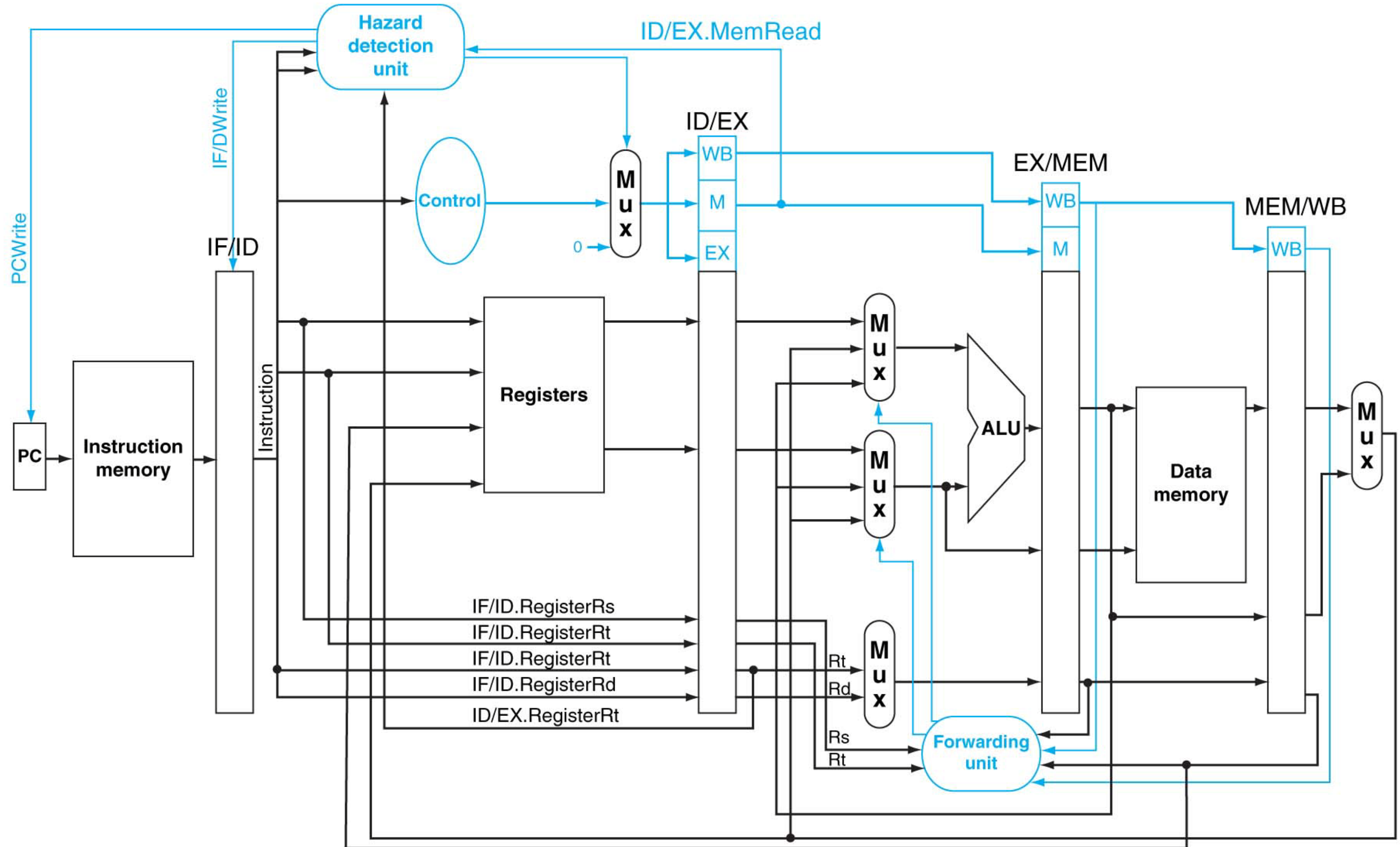
- Check for load instruction
- Check if the register to be loaded is part of the current instruction
- If it is, stall the pipeline

Implementation of a stall

- For one cycle (set all control signal to zero)
  - Stop update of PC
  - Freeze IF/ID pipeline register



Datapath with stall

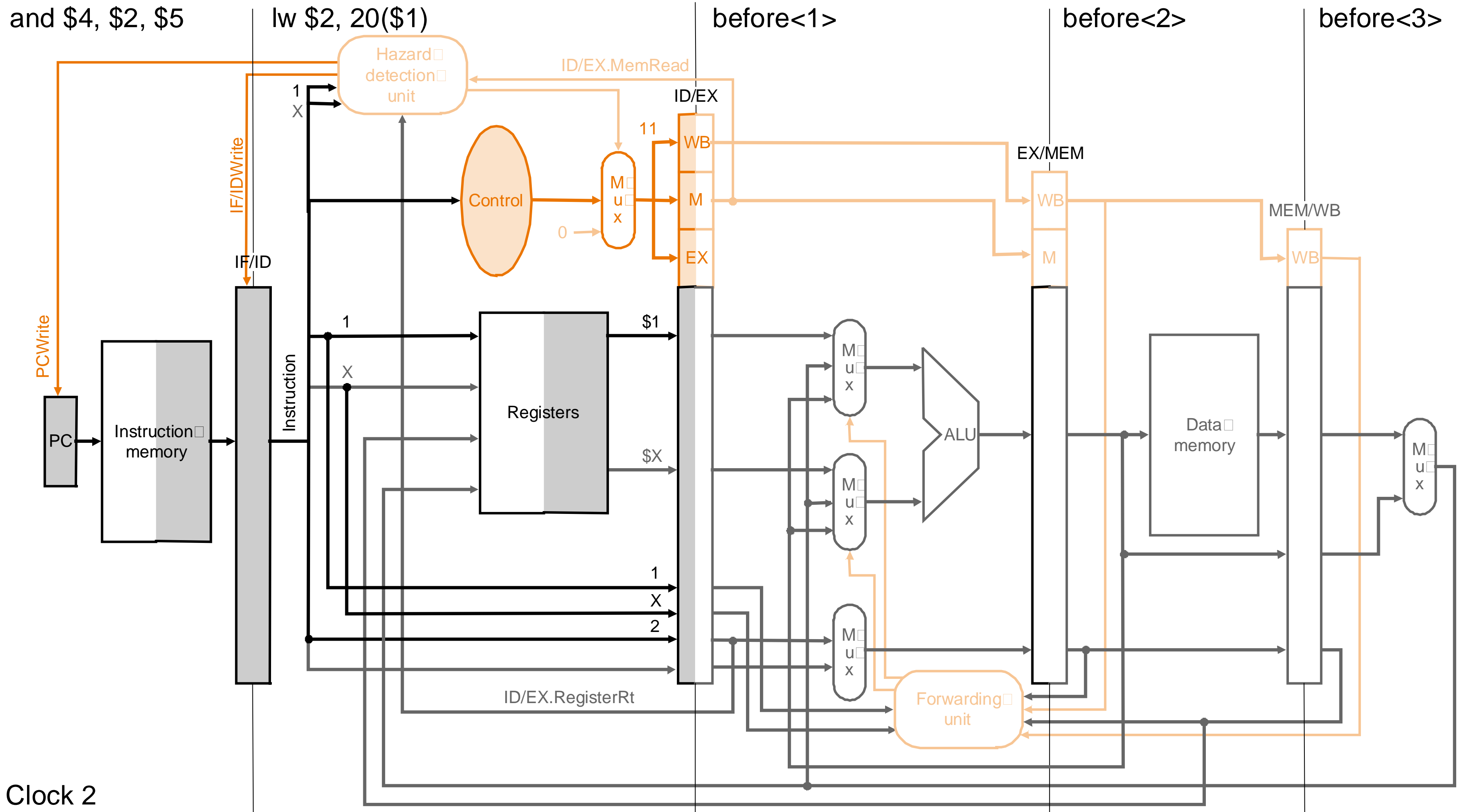


Example with stall

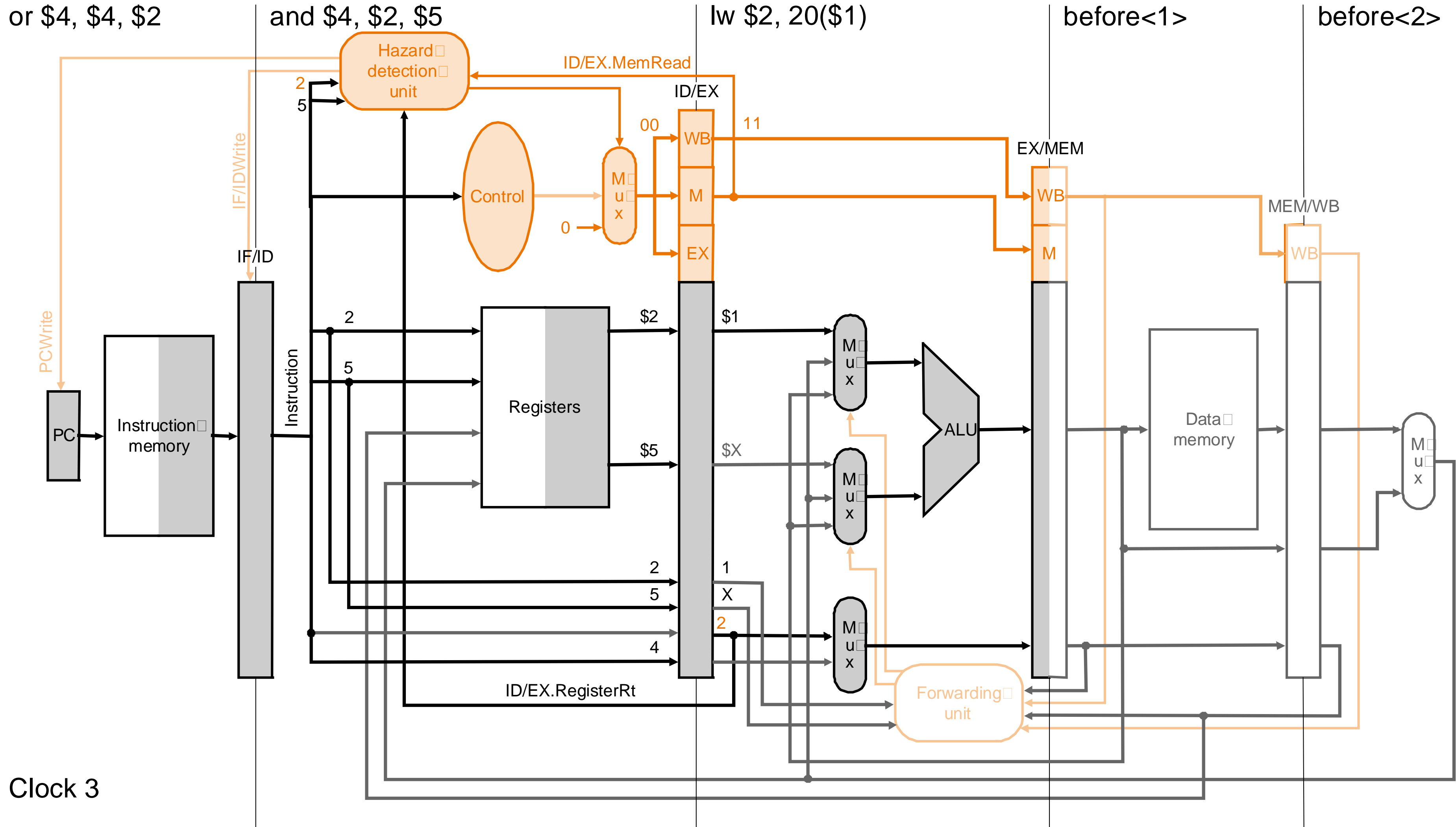
- Code sequence

```
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```

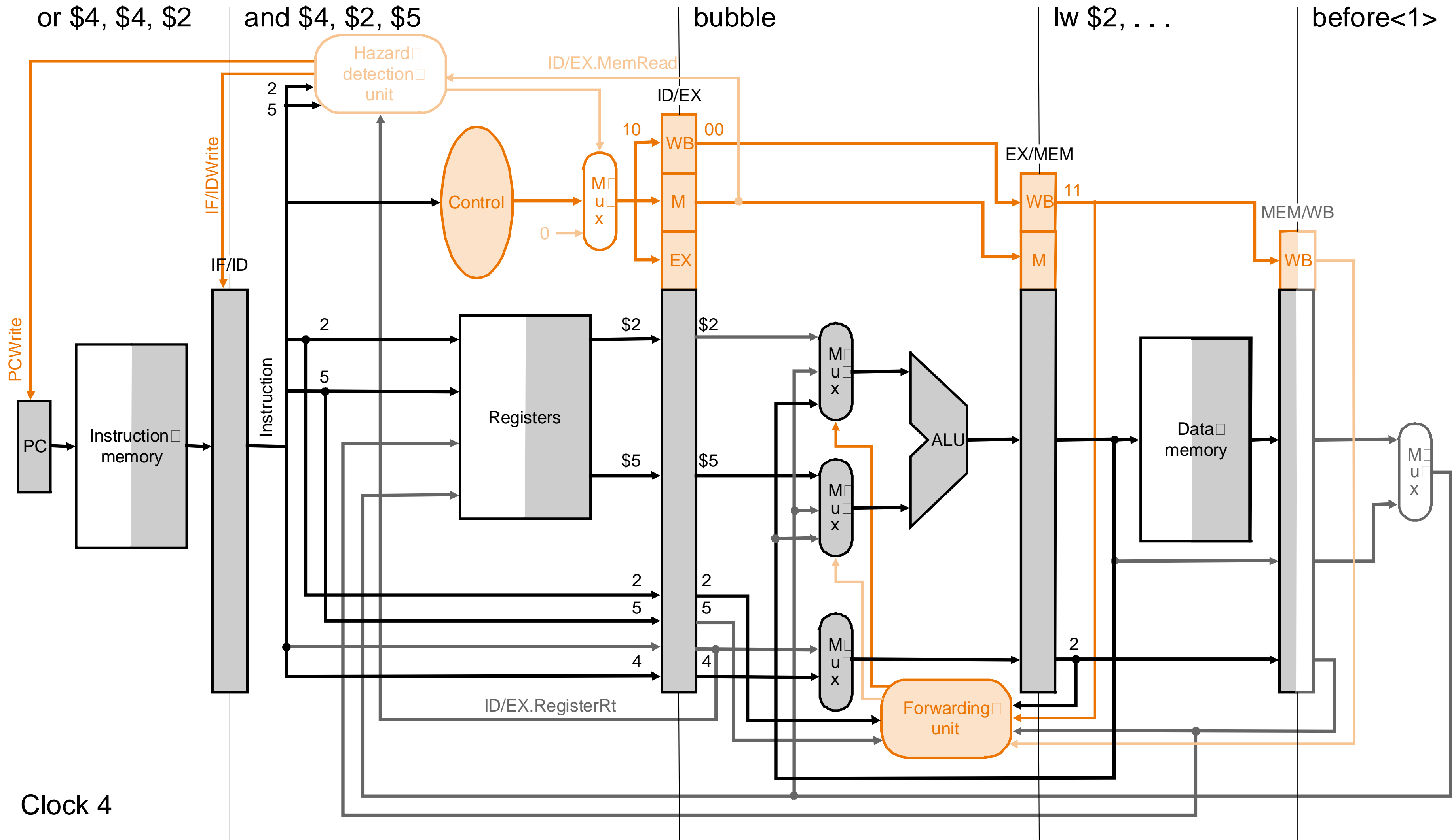
Example with stall



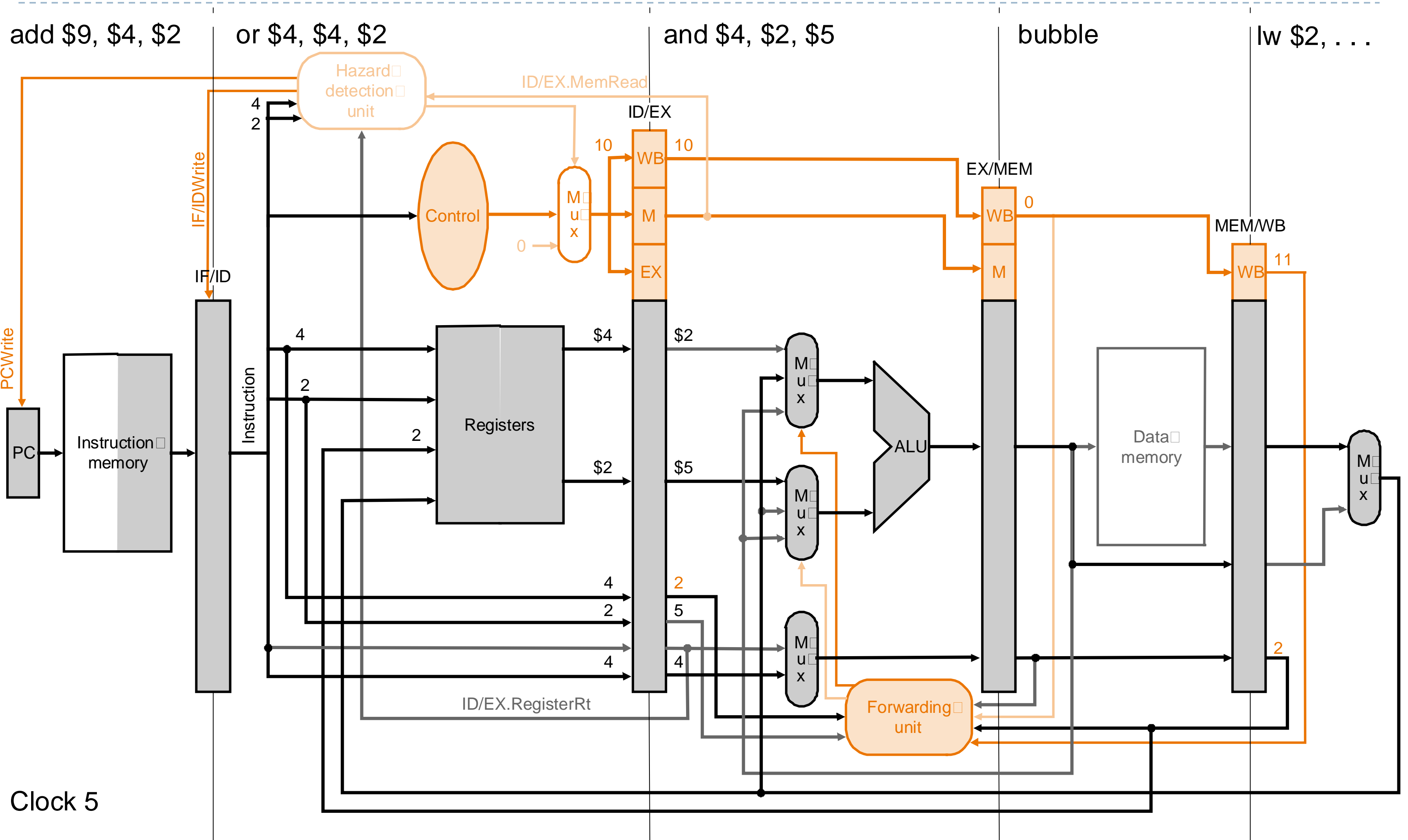
Example with stall



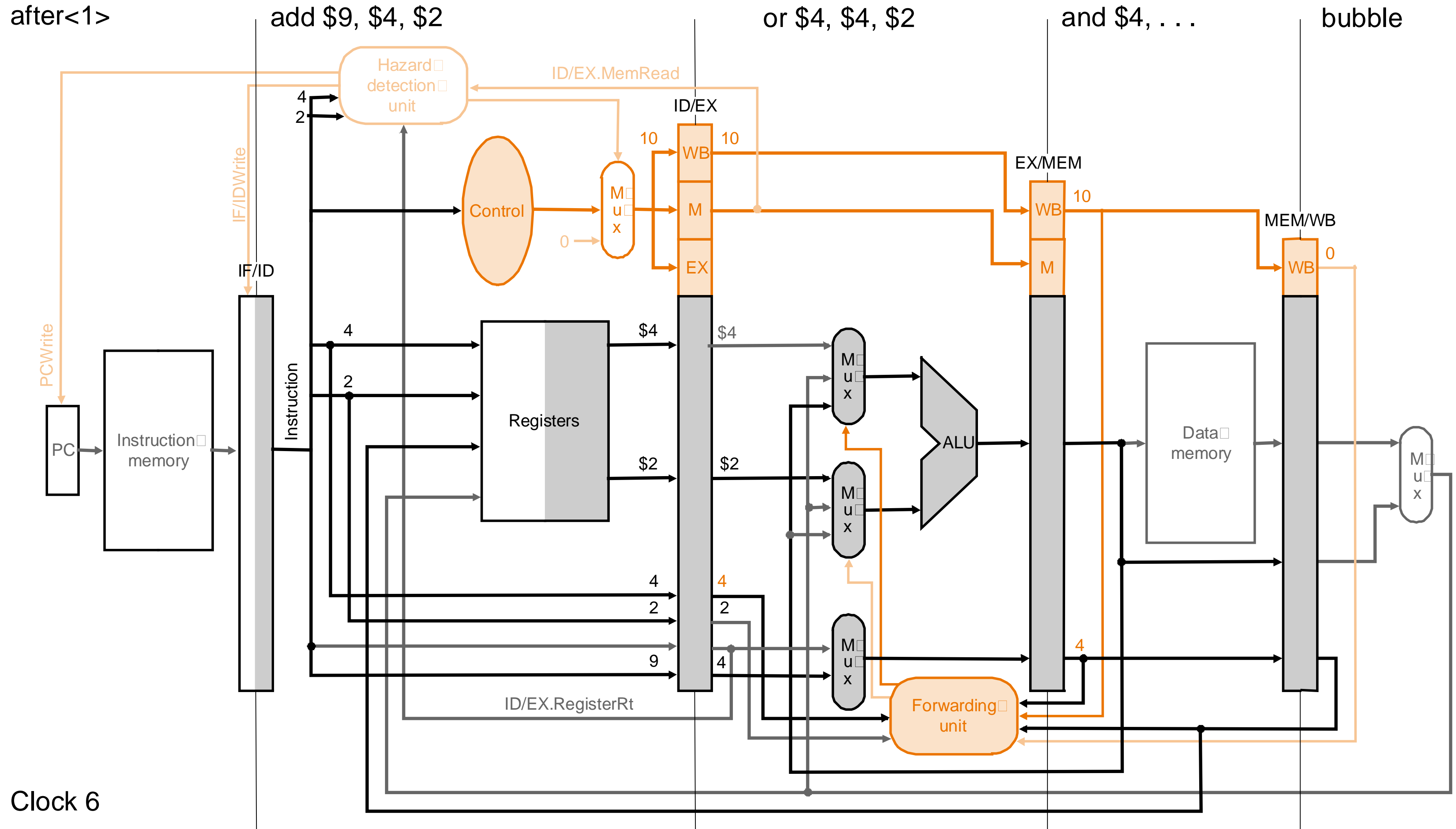
Example with stall



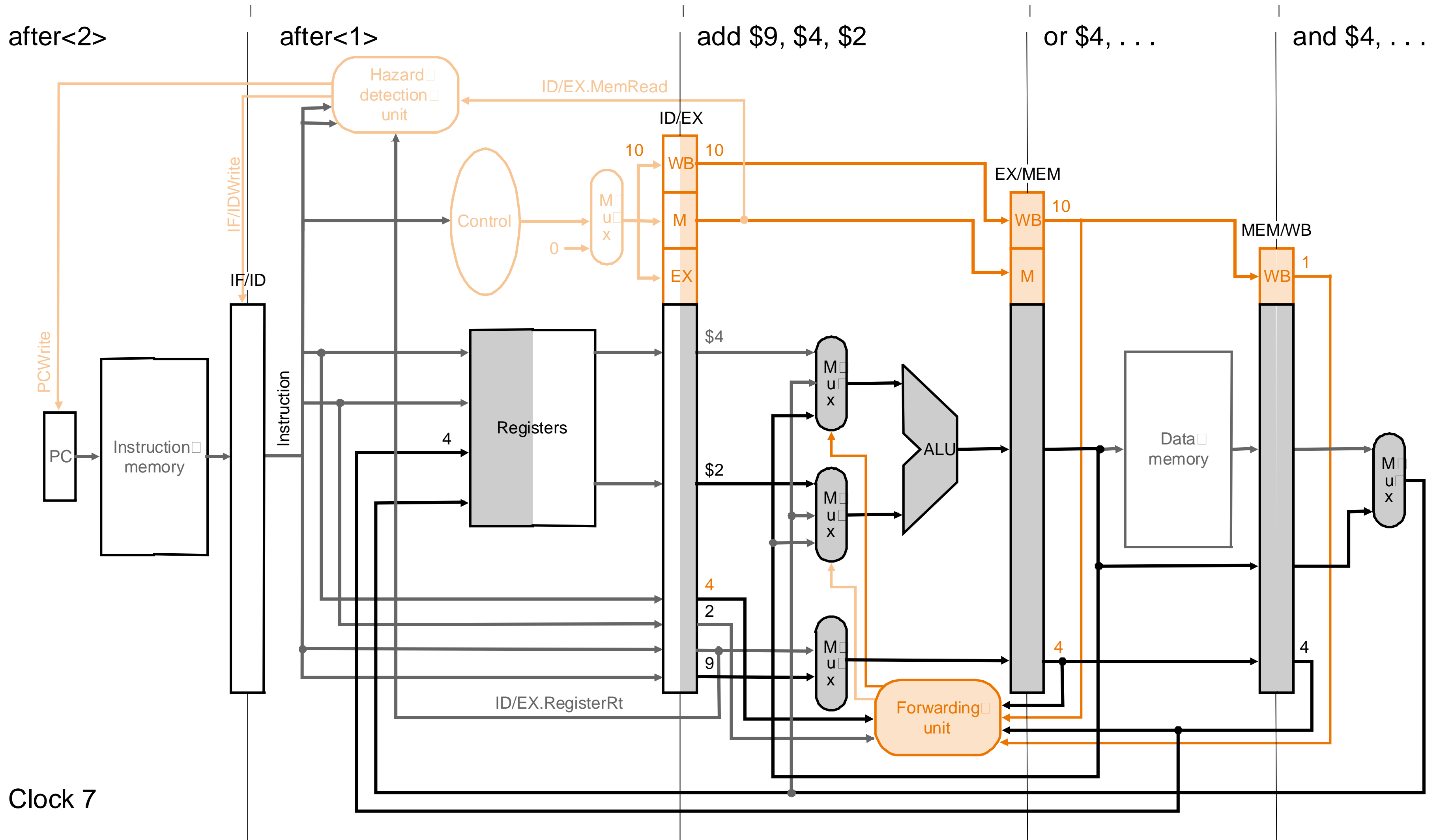
Example with stall



Example with stall



Example with stall



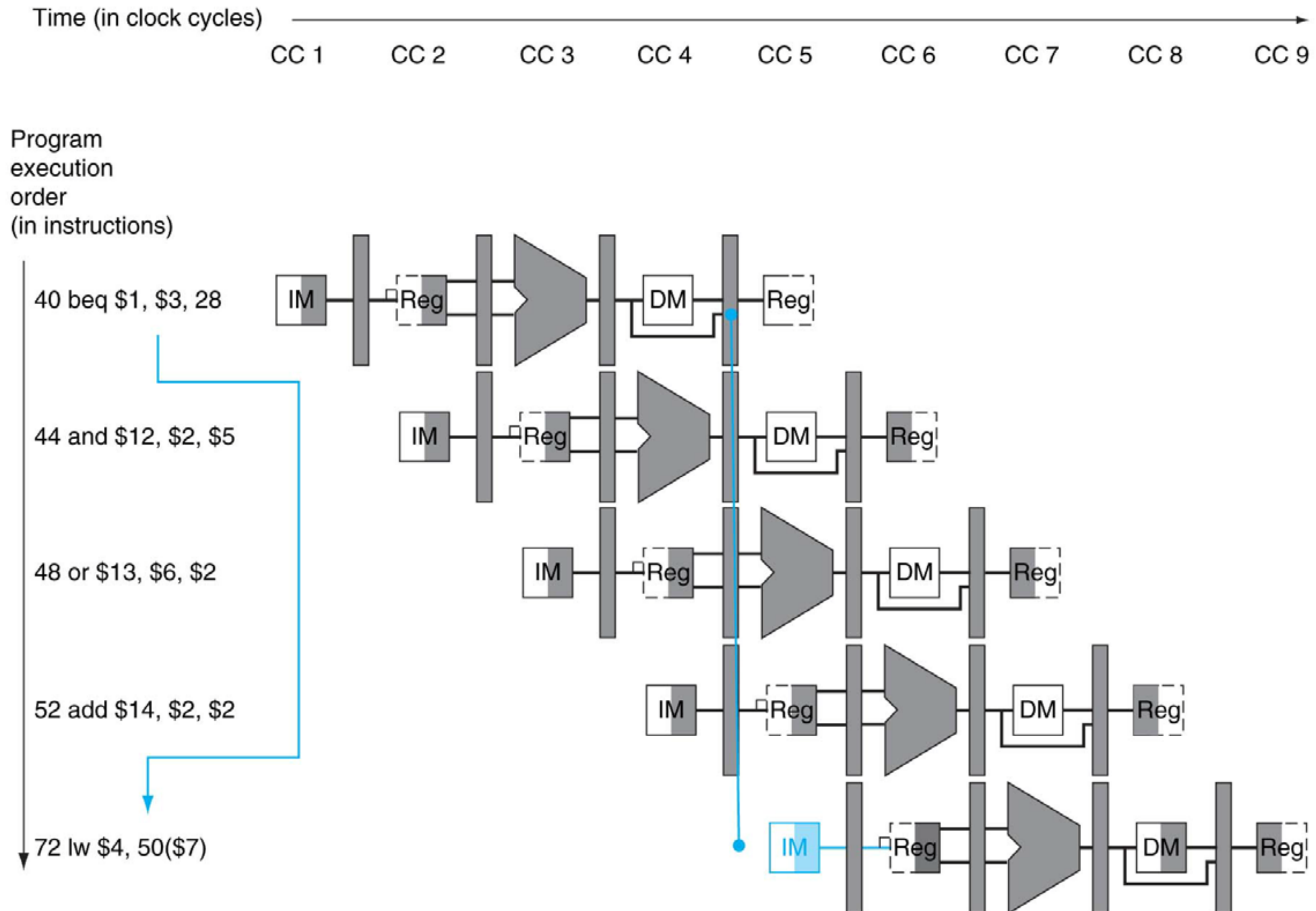
Branch hazards

Current design:

- Decision occurs in MEM stage
- Branch not taken
  - Continuously fetch instructions
  - Simply continue
- Branch taken
  - Continuously fetch instructions
  - On decision: discard three instructions
  - Set controls to '0'
  - Clear instructions in IF, ID and EX stage
  - No register changed because no instruction has reached the write back stage

Branch hazard

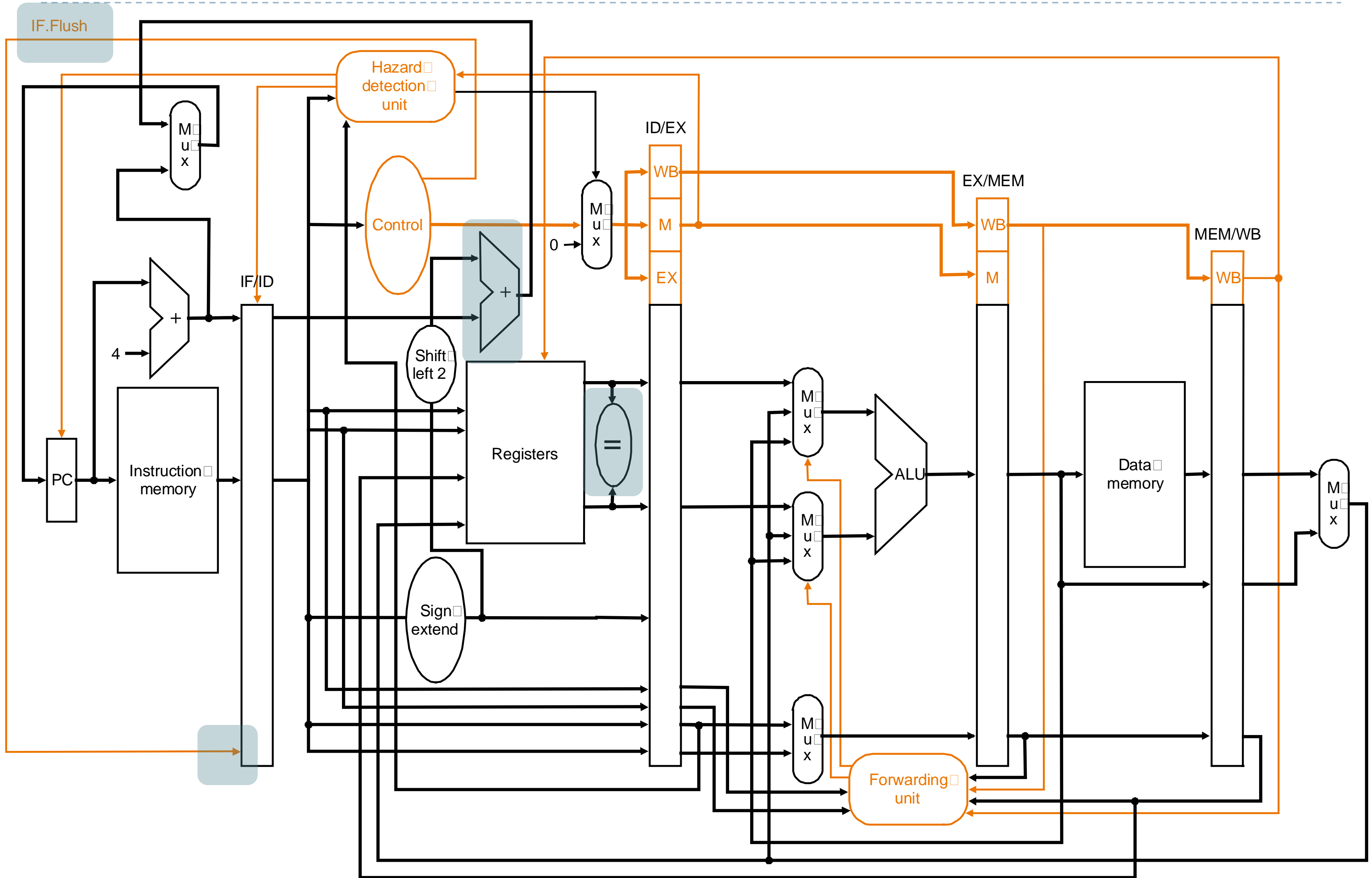
- Continuation of the program @ 72



Reduction of branch costs

- Move branch decision to an earlier stage
- Select branch address at
  - End of EX stage -> two cycle penalty
  - End of ID stage -> one cycle penalty
- Move the branch address adder to ID stage
- Branch detection in ID stage
  - Exclusive-or of the bits
  - AND of the results
- Clear instruction field in IF/ID pipeline -> creates a NOP

Data path with branch

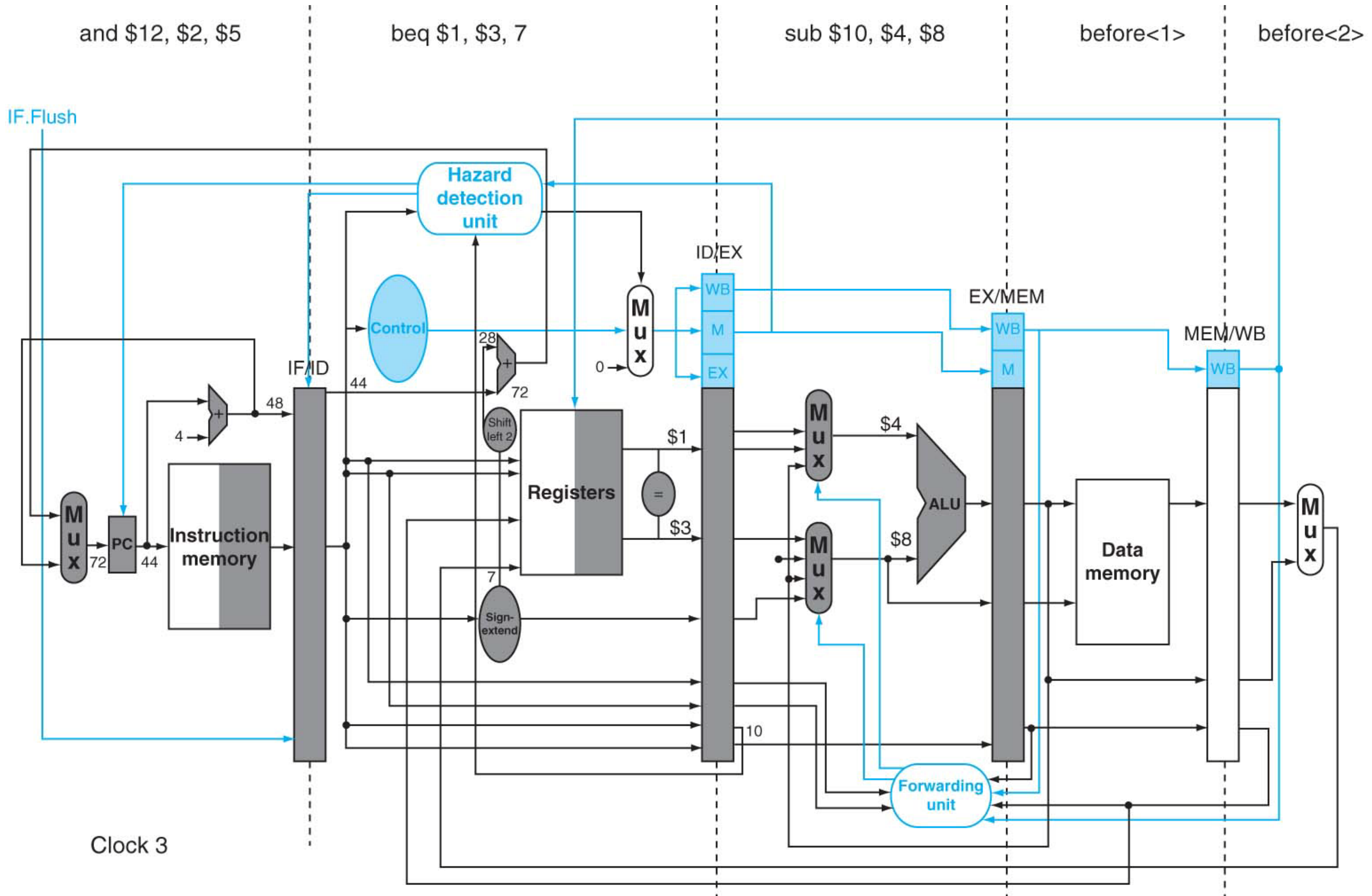


Example

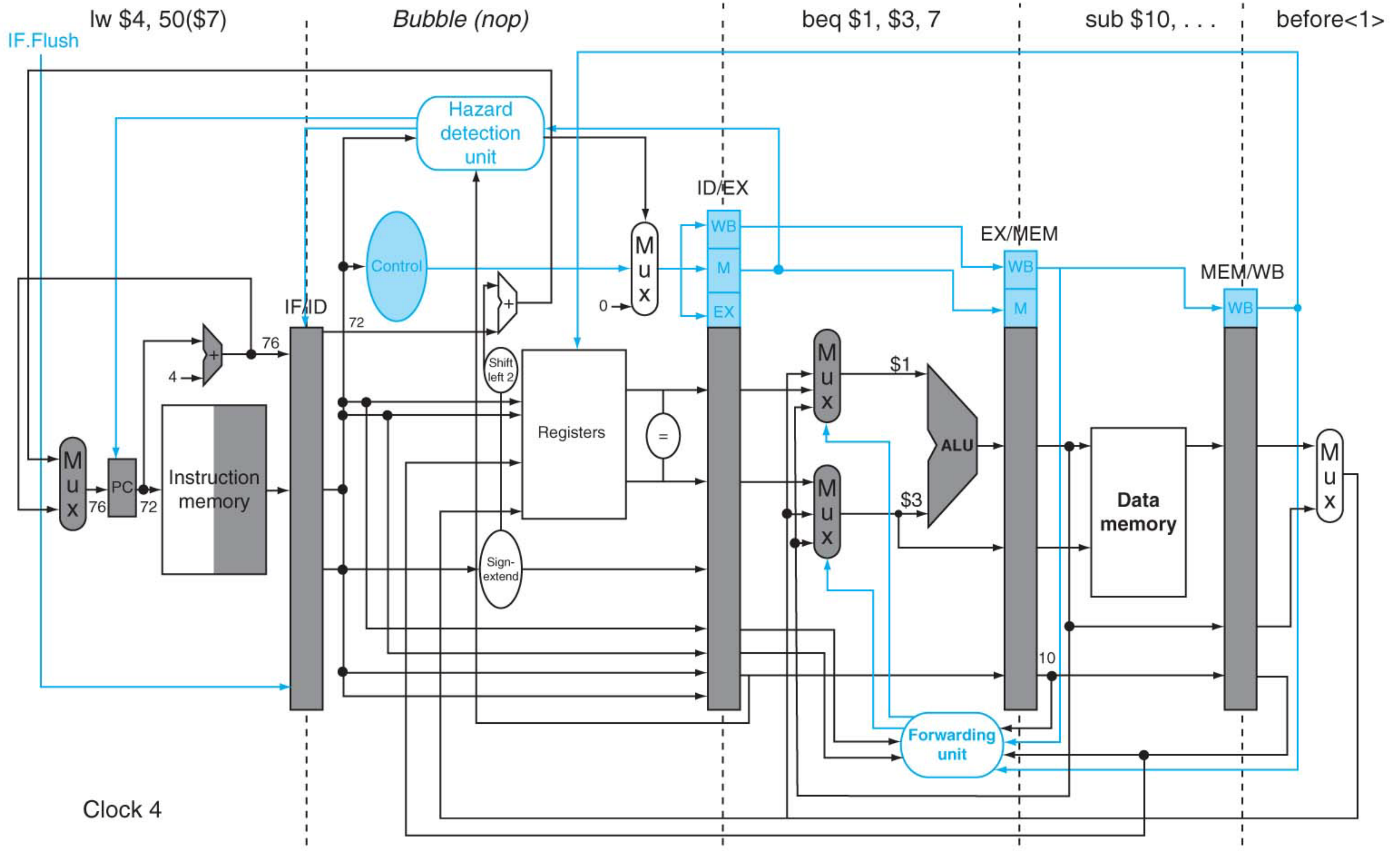
- Branch is taken

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

Example



Example

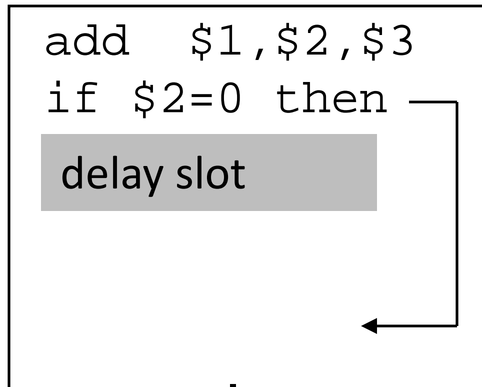


Delayed branch

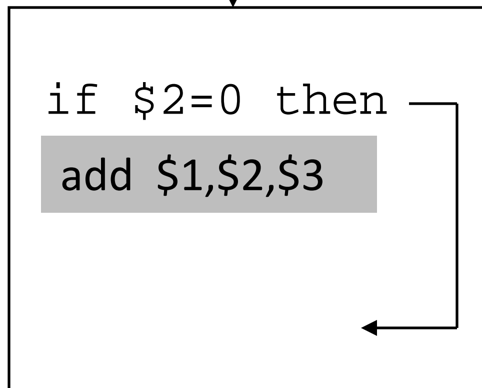
- If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with delayed branches which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction
  - MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay
- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper

Scheduling Branch Delay Slots

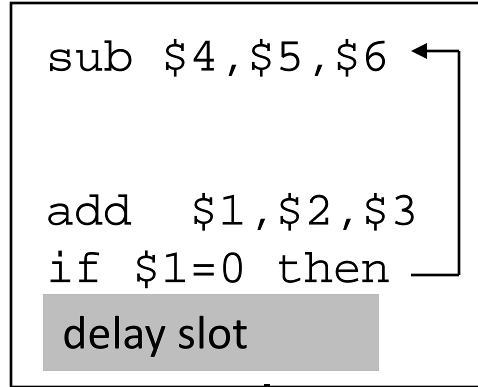
A. From before branch



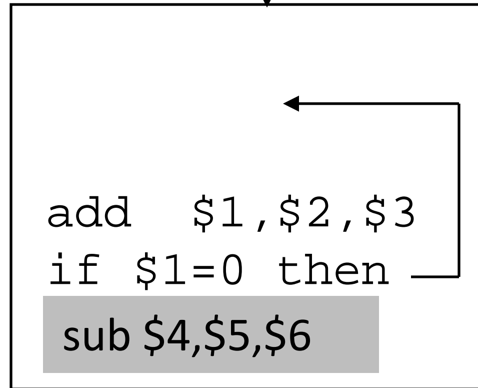
becomes



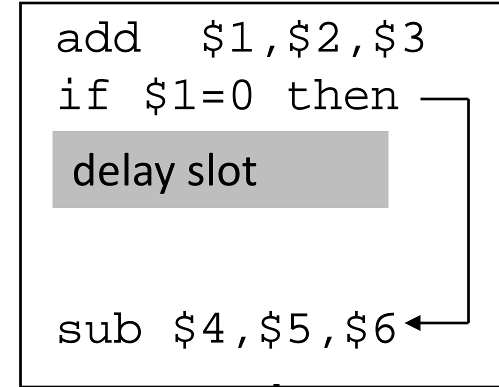
B. From branch target



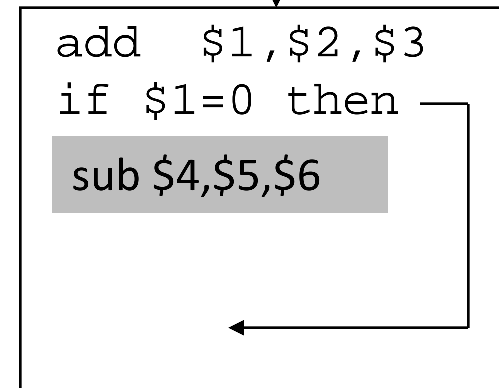
becomes



C. From fall through



becomes



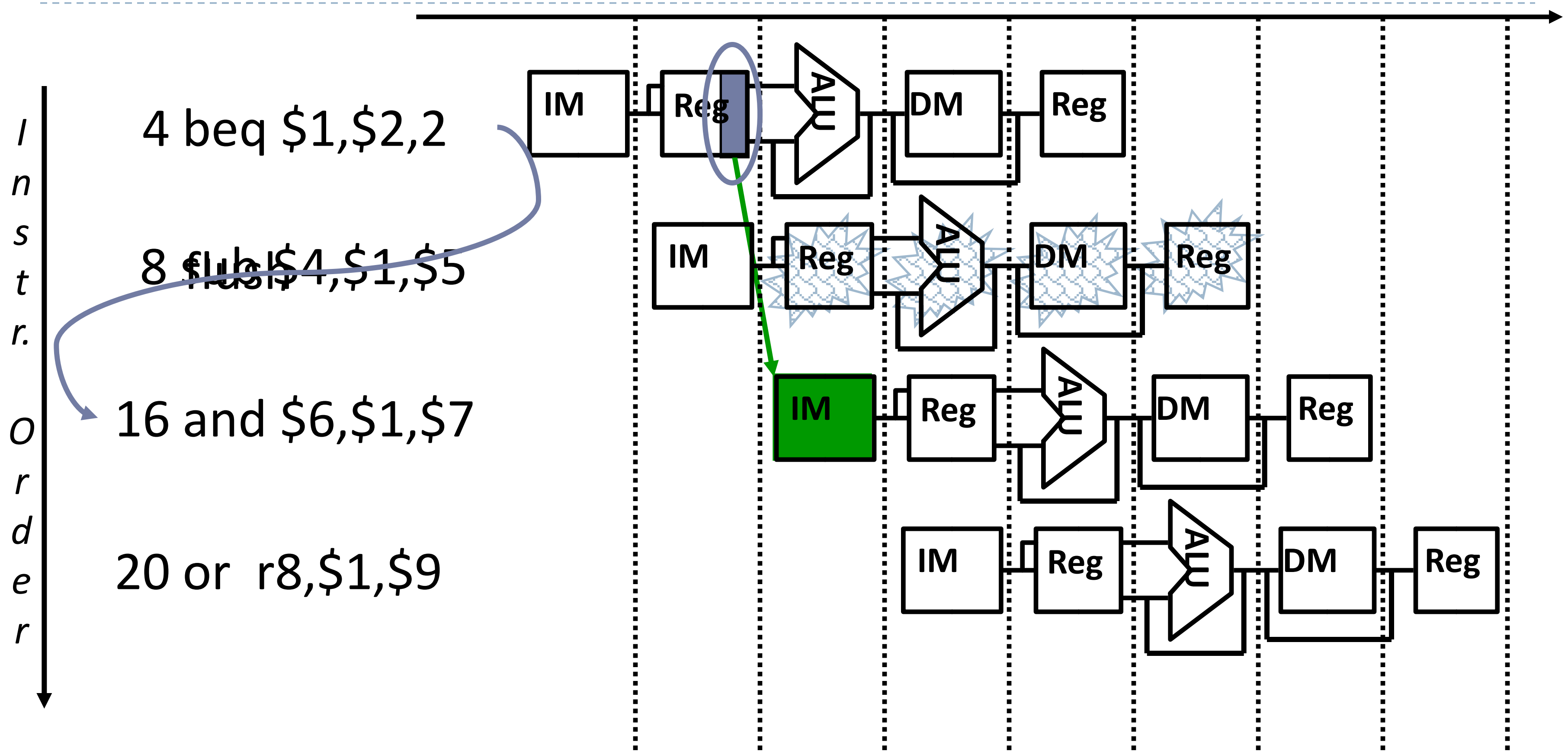
- A is the best choice, fills delay slot and reduces IC
- In B and C, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Static Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

1. Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
   - If taken, flush instructions after the branch (earlier in the pipeline)
     - in IF, ID, and EX stages if branch logic in MEM – three stalls
     - In IF and ID stages if branch logic in EX – two stalls
     - in IF stage if branch logic in ID – one stall
   - ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
   - restart the pipeline at the branch destination

Flushing with Misprediction (Not Taken)



- To flush the IF stage instruction, assert `IF.Flush` to zero the instruction field of the IF/ID pipeline register (transforming it into a `noop`)

Branching Structures

- Predict not taken works well for "top of the loop" branching structures
  - But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead

```
Loop: beq $1,$2,Out
      1nd loop instr
           .
           .
           .
      last loop instr
      j  Loop
Out:  fall out instr
```

- Predict not taken doesn't work well for "bottom of the loop" branching structures

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

Static Branch Prediction, con't

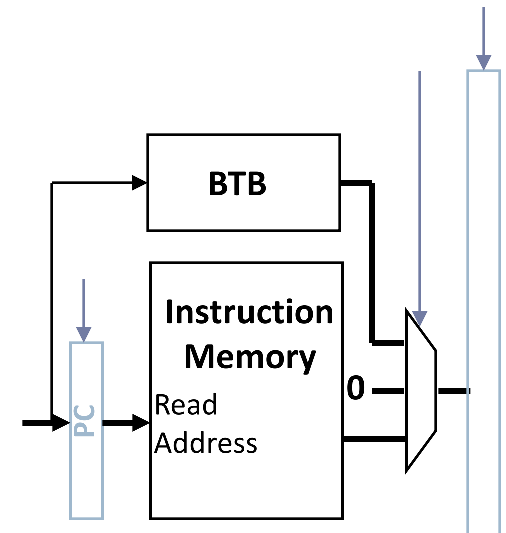
- Resolve branch hazards by assuming a given outcome and proceeding
  - Predict taken – predict branches will always be taken
    - Predict taken always incurs one stall cycle (if branch destination hardware has been moved to the ID stage)
    - Is there a way to "cache" the address of the branch target instruction ??
- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution
  - Dynamic branch prediction – predict branches at run-time using run-time information

Dynamic Branch Prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was execute
  - Prediction bit may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but the doesn't affect correctness, just performance
    - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s)
  - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit(s)
    - A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)

Branch Target Buffer

- The BHT predicts *when* a branch is taken, but does not tell *where* its taken to!
  - A branch target buffer (BTB) in the IF stage caches the branch target address, but we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which "next" instruction will be loaded into IF/ID at the next clock edge
    - Would need a two read port instruction memory
    - Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction



- If the prediction is correct, stalls can be avoided no matter which direction they go

1-bit Prediction Accuracy

- 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

  1. 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)
  2. As long as branch is taken (looping), prediction is correct
  3. 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)

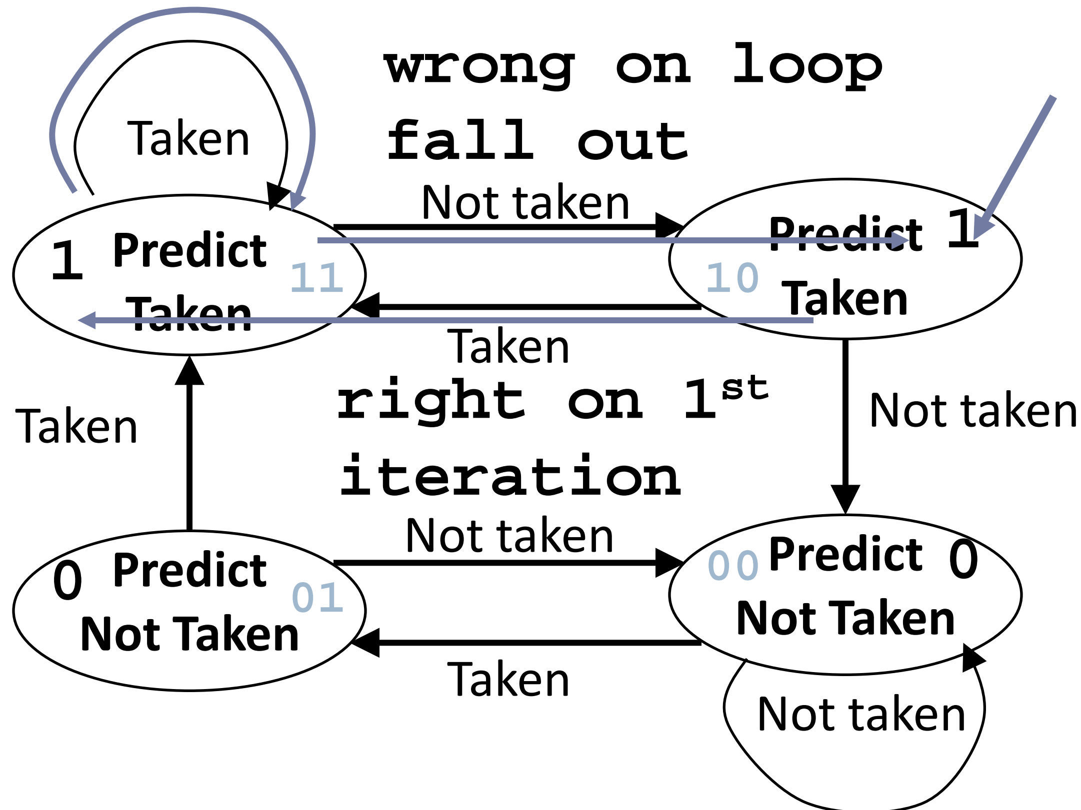
```
Loop: 1st loop instr
      2nd loop instr
          .
          .
          .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

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

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed

right 9 times




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

- BHT also stores the initial FSM state

Extracting Yet *More* Performance

- Increase the depth of the pipeline to increase the clock rate – superpipelining
  - The more stages in the pipeline, the more forwarding/hazard hardware needed and the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time)
- Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – multiple-issue
  - The instruction execution rate, CPI, will be less than 1, so instead we use IPC: instructions per clock cycle
    - E.g., a 6 GHz, four-way multiple-issue processor can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4
  - If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?

Types of Parallelism

- Instruction-level parallelism (ILP) of a program – a measure of the average number of instructions in a program that a processor *might* be able to execute at the same time
  - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions
- Data-level parallelism (DLP)

```
DO   I = 1   TO   100
   A[I] = A[I] + 1
CONTINUE
```

  - Machine parallelism of a processor – a measure of the ability of the processor to take advantage of the ILP of the program
    - Determined by the number of instructions that can be fetched and executed at the same time
  - To achieve high performance, need *both* ILP and machine parallelism

Multiple-Issue Processor Styles

- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)
    - 128-bit “bundles” containing three instructions, each 41-bits plus a 5-bit template field (which specifies which FU each instruction needs)
    - Five functional units (IntALU, Mmedia, Dmem, FPALU, Branch)
    - Extensive support for speculation and predication
- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8) are being made dynamically (at run time by the hardware)
  - E.g., IBM Power series, Pentium 4, MIPS R10K, AMD Barcelona

Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - How many instructions to issue in one clock cycle – issue slots
  - Storage (data) dependencies – aka data hazards
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

Speculation

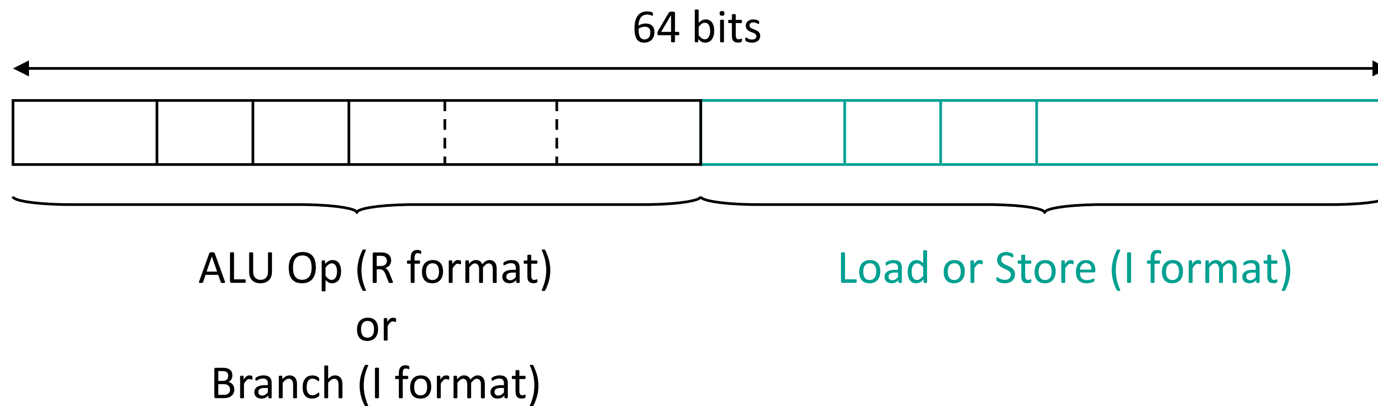
- Speculation is used to allow execution of future instr's that (may) depend on the speculated instruction
  - Speculate on the outcome of a conditional branch (branch prediction)
  - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (load speculation)
- Must have (hardware and/or software) mechanisms for
  - Checking to see if the guess was correct
  - Recovering from the effects of the instructions that were executed speculatively if the guess was incorrect
- Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur

Static Multiple Issue Machines (VLIW)

- Static multiple-issue processors (aka VLIW) use the compiler (at compile-time) to statically decide which instructions to issue and execute simultaneously
  - Issue packet – the set of instructions that are bundled together and issued in one clock cycle – think of it as one large instruction with multiple operations
  - The mix of instructions in the packet (bundle) is usually restricted – a single "instruction" with several predefined fields
  - The compiler does static branch prediction and code scheduling to reduce (control) or eliminate (data) hazards
- VLIW's have
  - Multiple functional units
  - Multi-ported register files
  - Wide program bus

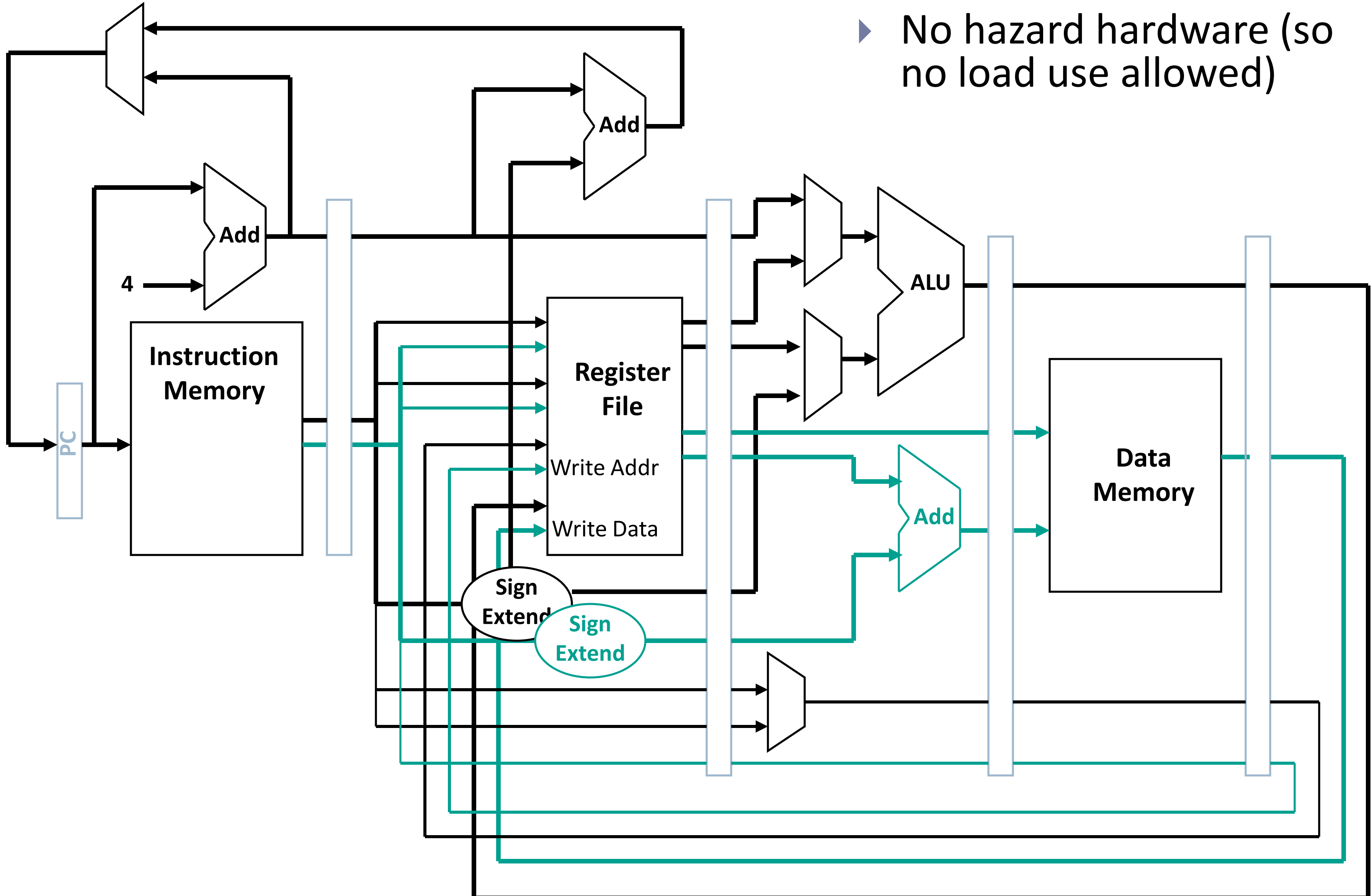
An Example: A VLIW MIPS

- Consider a 2-issue MIPS with a 2 instr bundle



- Instructions are always fetched, decoded, and issued in pairs
  - If one instr of the pair can not be used, it is replaced with a noop
- Need 4 read ports and 2 write ports and a separate memory address adder

A MIPS VLIW (2-issue) Datapath



- No hazard hardware (so no load use allowed)

Code Scheduling Example

- Consider the following loop code

```
lp:   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,$0,lp      # branch if $s1 != 0
```

- Must “schedule” the instructions to avoid pipeline stalls
  - Instructions in one bundle must be independent
  - Must separate load use instructions from their loads by one cycle
  - Notice that the first two instructions have a load use dependency, the next two and last two have data dependencies
  - Assume branches are perfectly predicted by the hardware

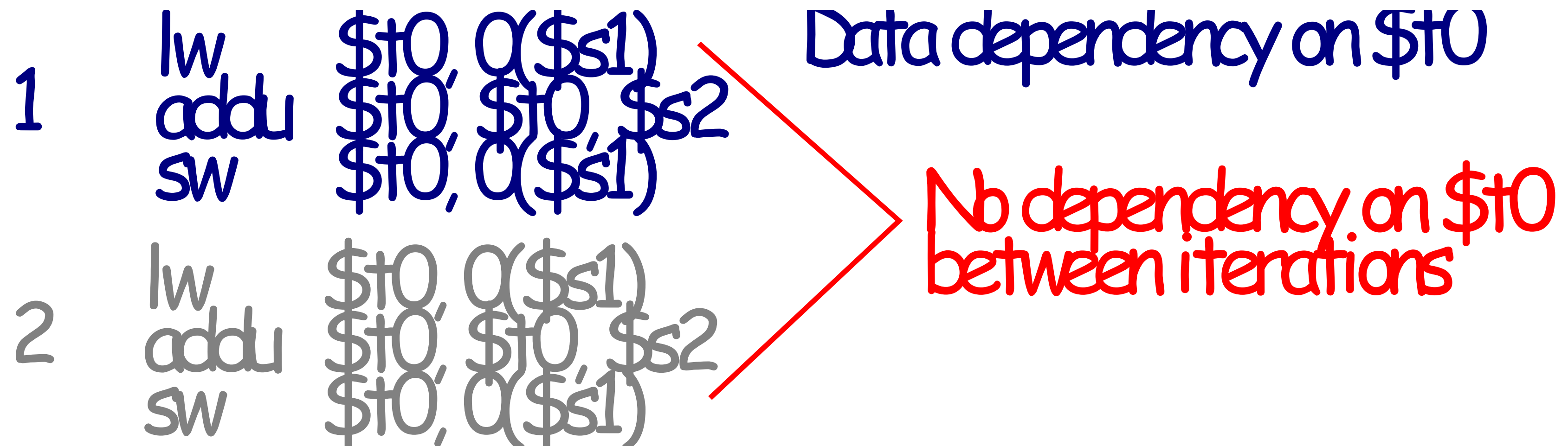
The Scheduled Code (Not Unrolled)

	ALU or branch	Data transfer	CC
lp:		`lw $t0,0($s1)`	1
	`addi $s1,$s1,-4`		2
	`addu $t0,$t0,$s2`		3
	`bne $s1,$0,lp`	`sw $t0,4($s1)`	4
			5

- Four clock cycles to execute 5 instructions for a
  - CPI of 0.8 (versus the best case of 0.5)
  - IPC of 1.25 (versus the best case of 2.0)
  - noops don't count towards performance !!

Loop Unrolling

- Loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP
- Apply loop unrolling (4 times for our example) and then schedule the resulting code
  - Eliminate unnecessary loop overhead instructions
  - Schedule so as to avoid load use hazards
- During unrolling the 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,$0,lp     # branch if $s1 != 0
```

The Scheduled Code (Unrolled)

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

- Eight clock cycles to execute 14 instructions for a
  - CPI of 0.57 (versus the best case of 0.5)
  - IPC of 1.8 (versus the best case of 2.0)

VLIW Advantages & Disadvantages

- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
    - Allow more instr’s per VLIW bundle and add more FUs
- Disadvantages
  - Programmer/compiler complexity and longer compilation times
    - Deep pipelines and long latencies can be confusing (making peak performance elusive)
  - Lock step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
  - Object (binary) code incompatibility
  - Needs lots of program memory bandwidth
  - Code bloat
    - Noops are a waste of program memory space
    - Loop unrolling to expose more ILP uses more program memory space

Dynamic Multiple Issue Machines (SS)

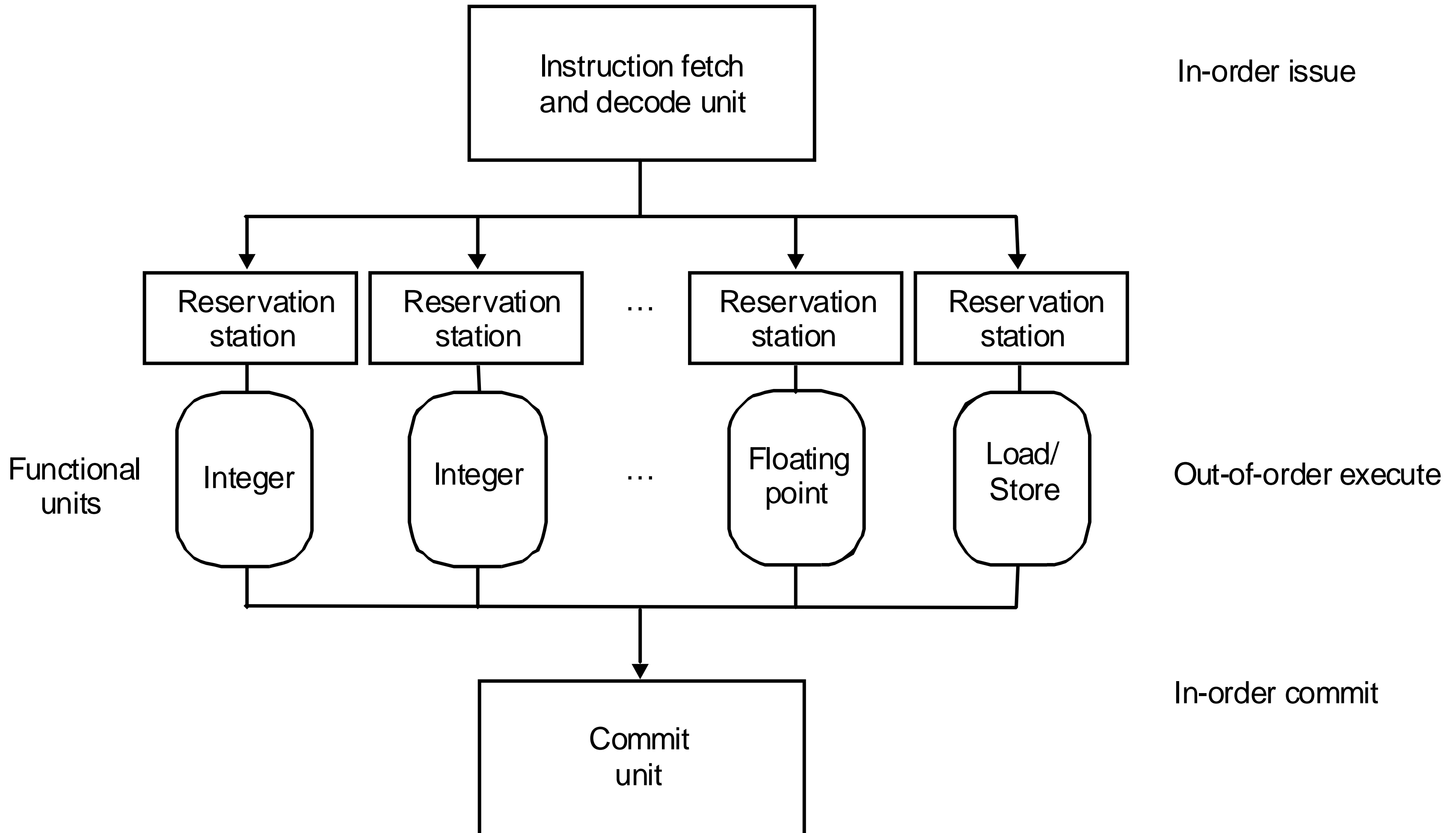
- Dynamic multiple-issue processors (aka SuperScalar) use hardware at run-time to dynamically decide which instructions to issue and execute simultaneously
- Instruction-fetch and issue – fetch instructions, decode them, and *issue* them to a FU to await execution
  - Defines the Instruction lookahead capability – fetch, decode and issue instructions beyond the current instruction
- Instruction-execution – as soon as the source operands and the FU are ready, the result can be calculated
  - Defines the processor lookahead capability – complete execution of issued instructions beyond the current instruction
- Instruction-commit – when it is safe to, write back results to the RegFile or D$ (i.e., change the machine state)

In-Order vs Out-of-Order

- Instruction fetch and decode units are required to issue instructions in-order so that dependencies can be tracked
- The commit unit is required to write results to registers and memory in program fetch order so that
  - if exceptions occur the only registers updated will be those written by instructions before the one causing the exception
  - if branches are mispredicted, those instructions executed after the mispredicted branch don't change the machine state (i.e., we use the commit unit to correct incorrect speculation)
- Although the front end (fetch, decode, and issue) and back end (commit) of the pipeline run in-order, the FUs are free to initiate execution whenever the data they need is available – out-of-(program) order execution
  - Allowing out-of-order execution increases the amount of ILP

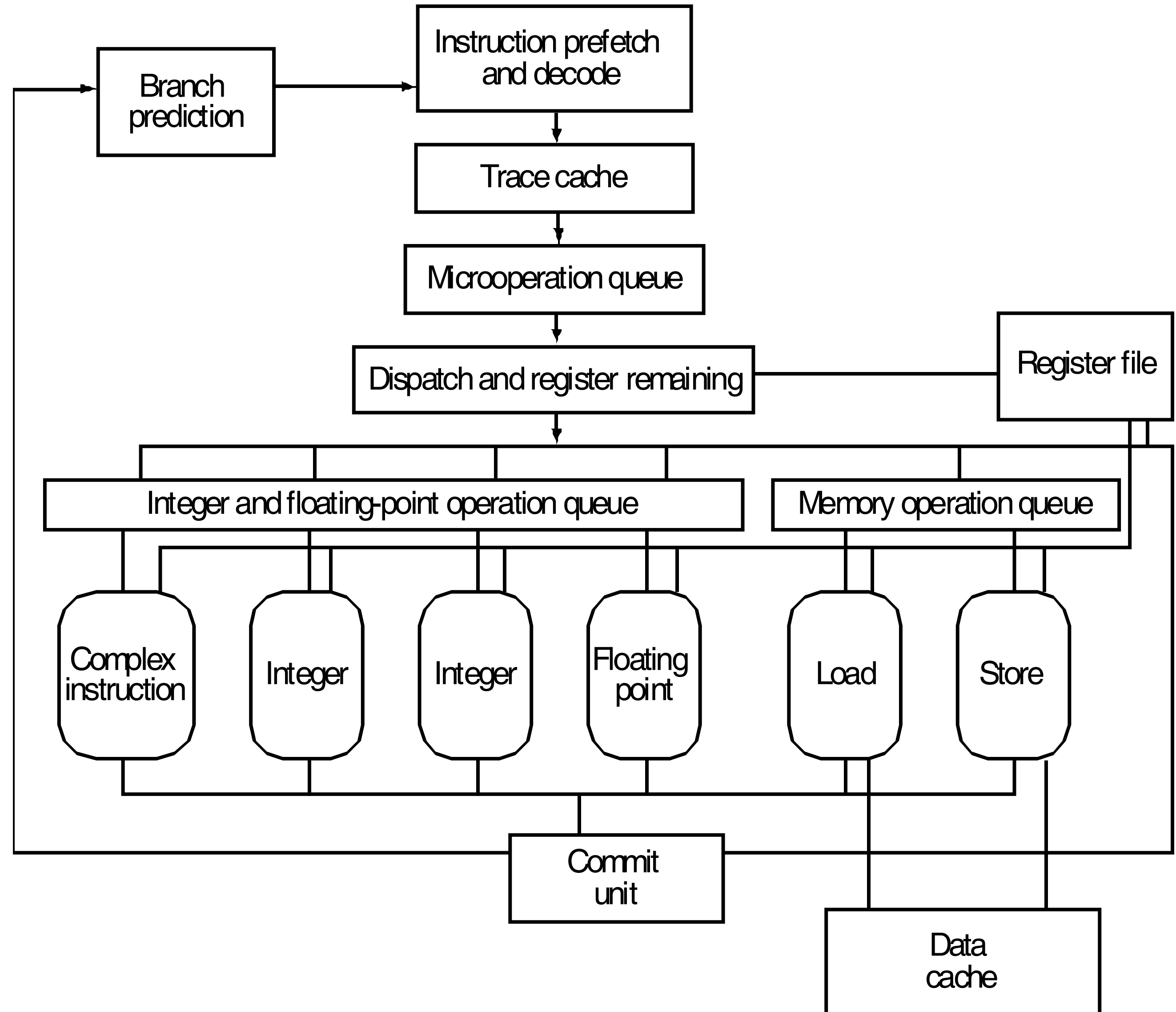
Dynamic Pipeline Scheduling

- Three core element

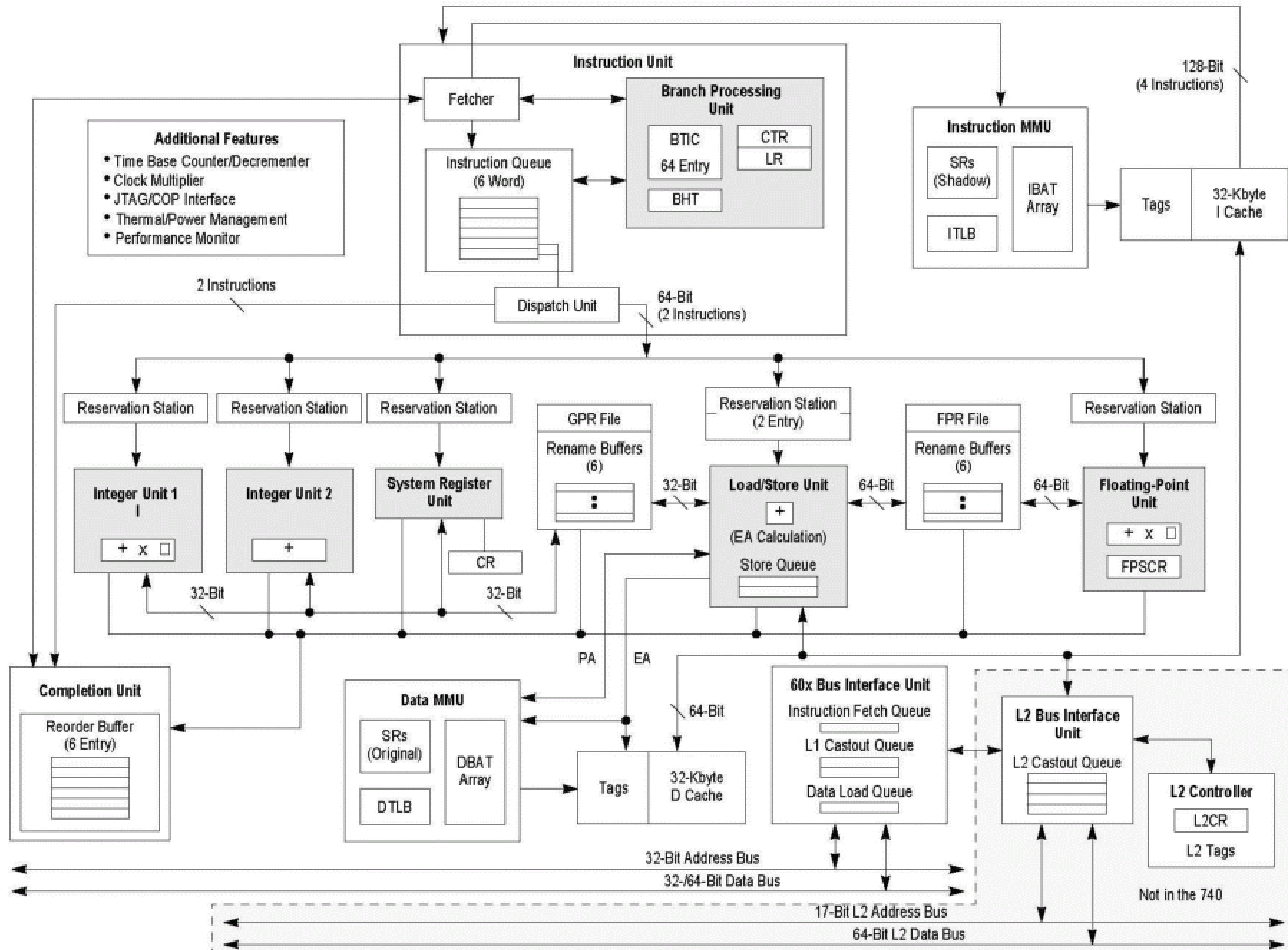


Pentium P4

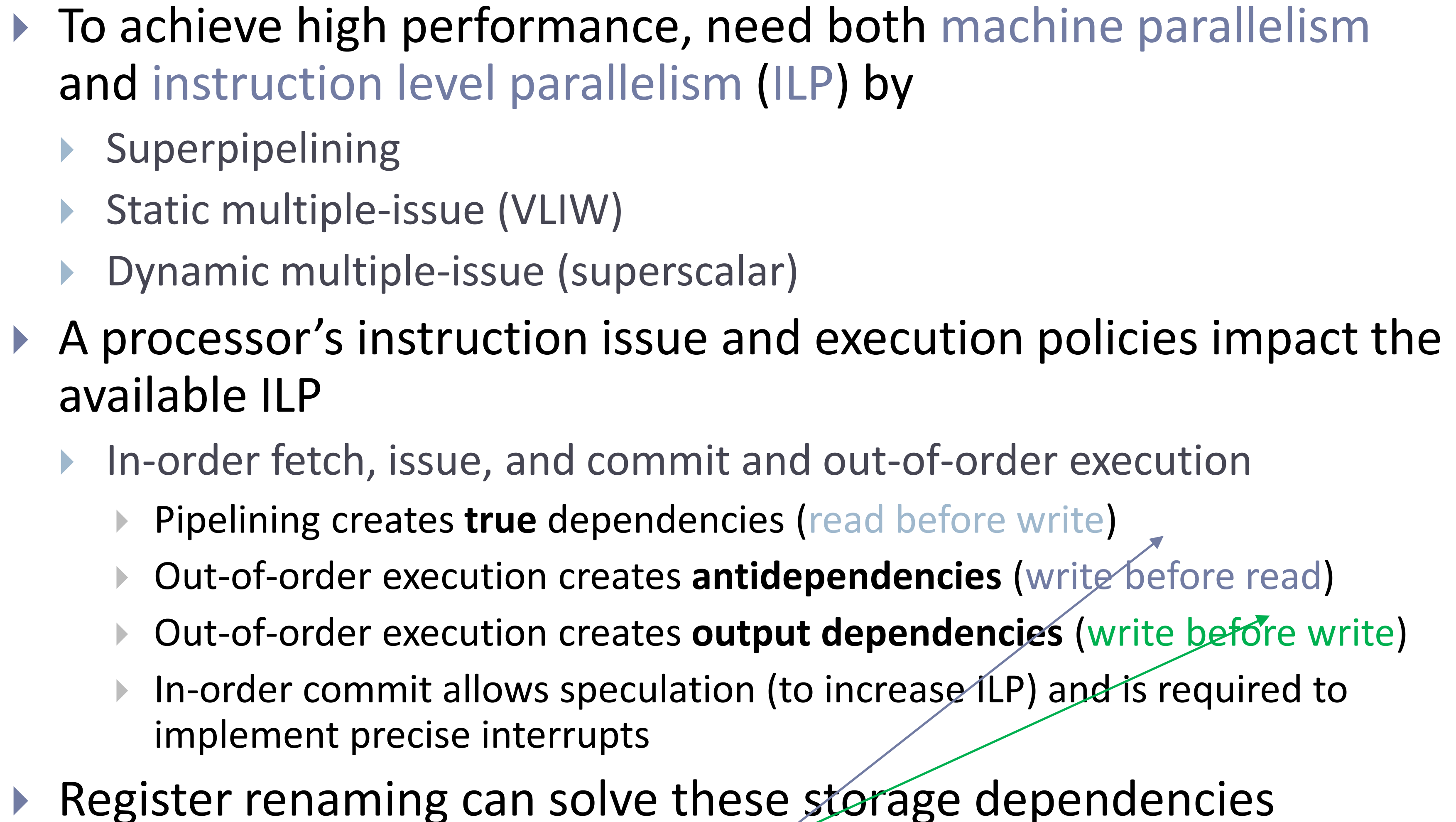
- 20 stages
- 7 FU
- Trace cache



PowerPC 750



Summary: Extracting More Performance



- To achieve high performance, need both machine parallelism and instruction level parallelism (ILP) by
  - Superpipelining
  - Static multiple-issue (VLIW)
  - Dynamic multiple-issue (superscalar)
- A processor's instruction issue and execution policies impact the available ILP
  - In-order fetch, issue, and commit and out-of-order execution
    - Pipelining creates **true** dependencies (read before write)
    - Out-of-order execution creates **antidependencies** (write before read)
    - Out-of-order execution creates **output dependencies** (write before write)
    - In-order commit allows speculation (to increase ILP) and is required to implement precise interrupts
- Register renaming can solve these storage dependencies

CISC vs RISC vs SS vs VLIW

	CISC	RISC	Superscalar	VLIW
Instr size	variable size	fixed size	fixed size	fixed size (but large)
Instr format	variable format	fixed format	fixed format	fixed format
Registers	few, some special Limited # of ports	Many GP Limited # of ports	GP and rename (RUU) Many ports	many, many GP Many ports
Memory reference	embedded in many instr's	load/store	load/store	load/store
Key Issues	decode complexity	data forwarding, hazards	hardware dependency resolution	(compiler) code scheduling

Evolution of Pipelined, SS Processors

	Year	Clock Rate	# Pipe Stages	Issue Width	OOO?	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	2930 MHz	14	4	Yes	2	75 W
Sun USPARC III	2003	1950 MHz	14	4	No	1	90 W
Sun T1 (Niagara)	2005	1200 MHz	6	1	No	8	70 W