Computer Architecture

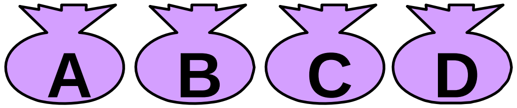
Ch. 6-2: Designing a Pipeline Processor

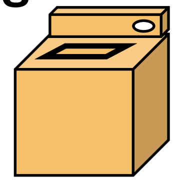
Spring, 2005

Sao-Jie Chen (csj@cc.ee.ntu.edu.tw)

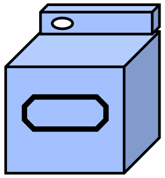
Pipelining: It’s Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold





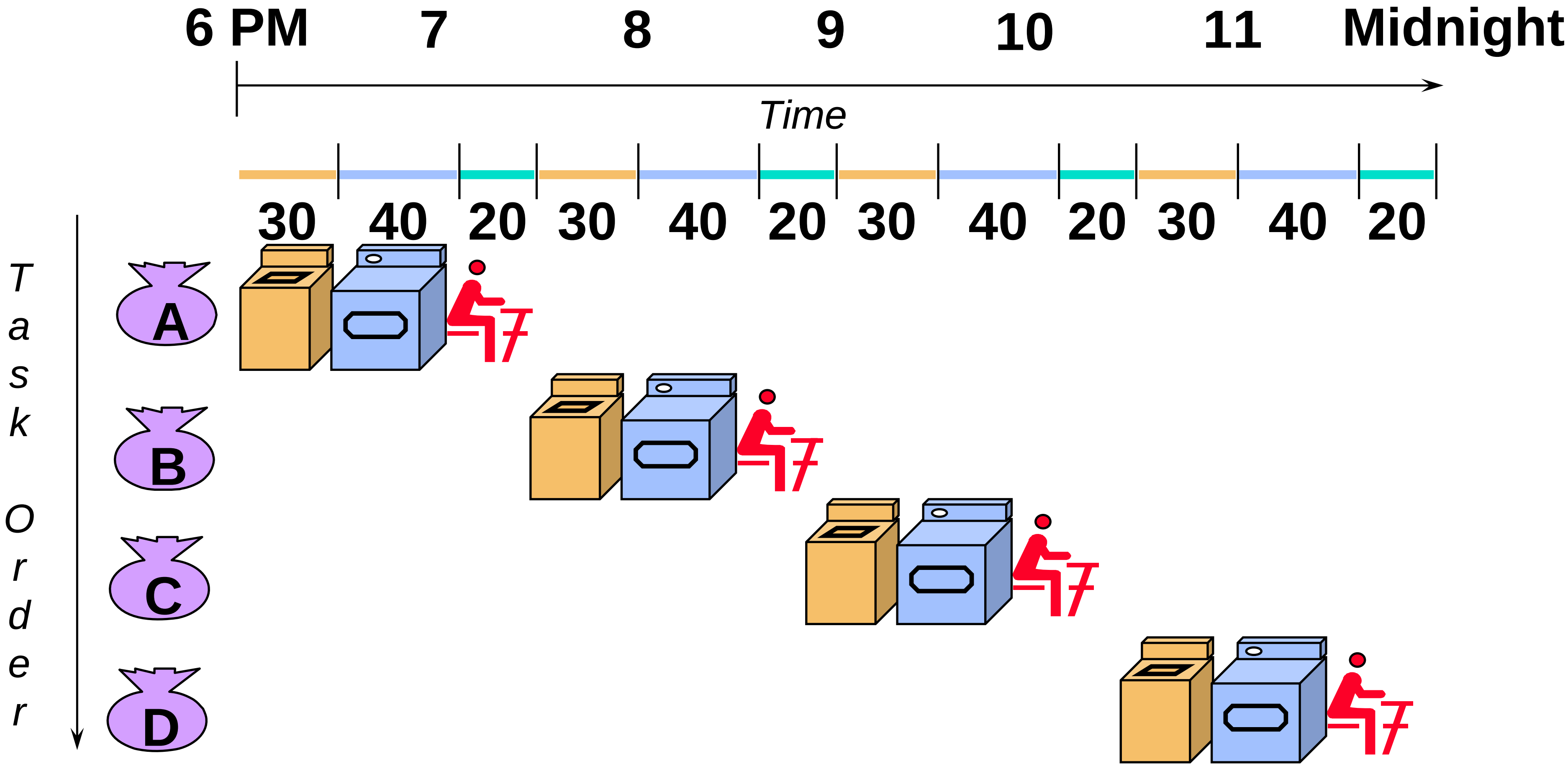
- Washer takes 30 minutes



- Dryer takes 40 minutes

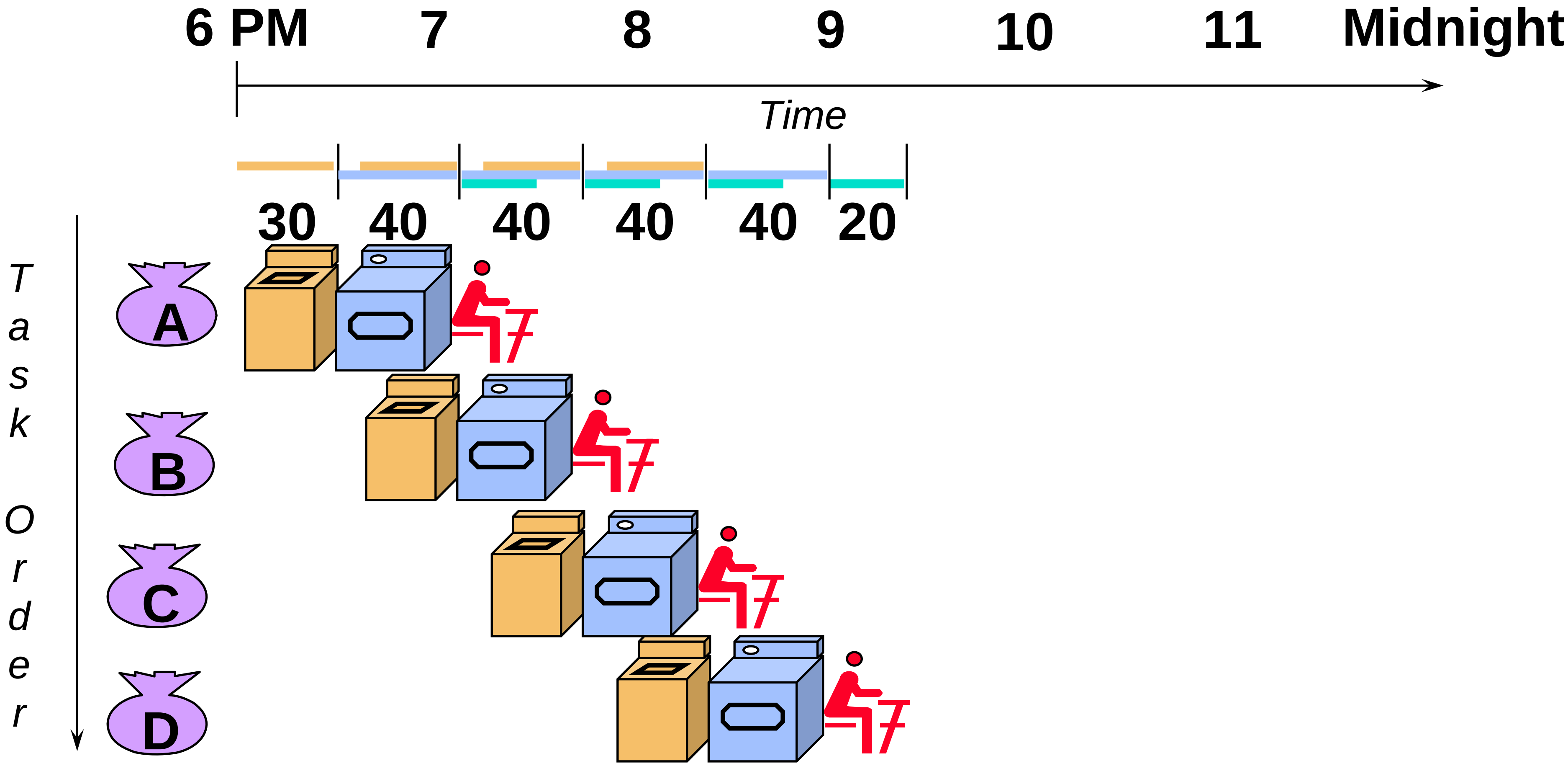
- “Folder” takes 20 minutes

Sequential Laundry



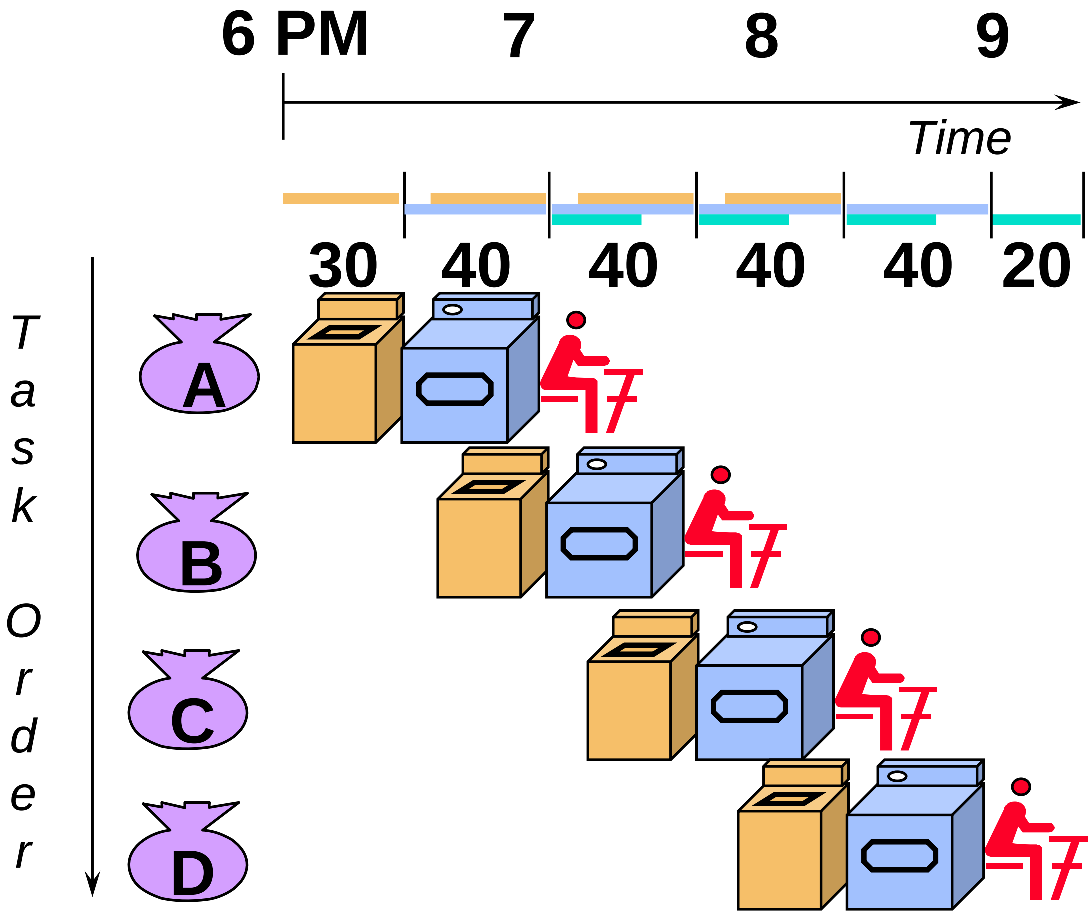
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



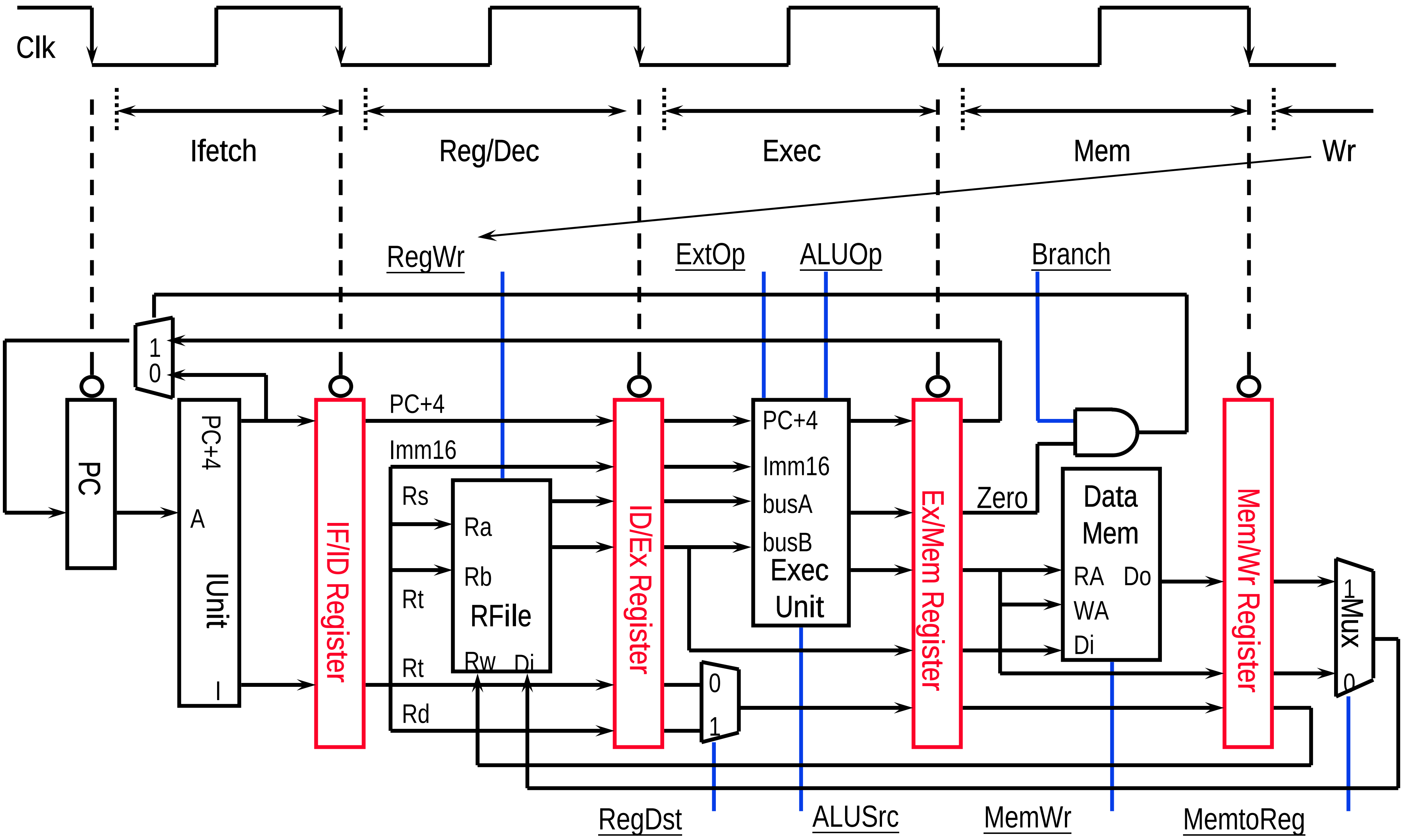
- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup

Review: A Pipelined Datapath



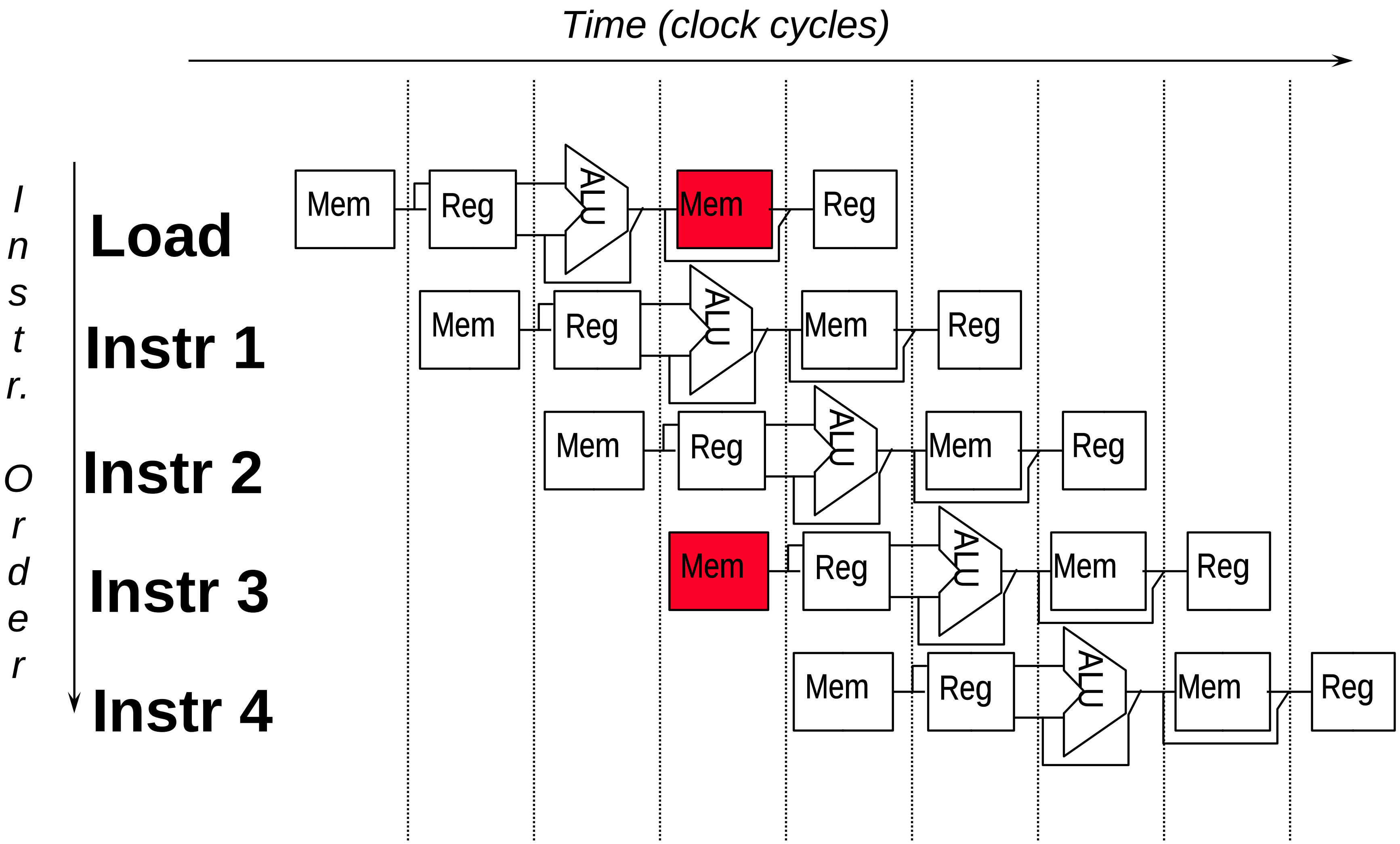
Review: Pipeline Summary

- **Pipeline Processor:**
  - **Natural enhancement of the multiple clock cycle processor**
  - **Each functional unit can only be used once per instruction**
  - **If an instruction is going to use a functional unit:**
    - ⇒ **it must use it at the same stage as all other instructions**
  - **Pipeline Control:**
    - ⇒ **Each stage's control signal depends ONLY on the instruction that is currently in that stage**

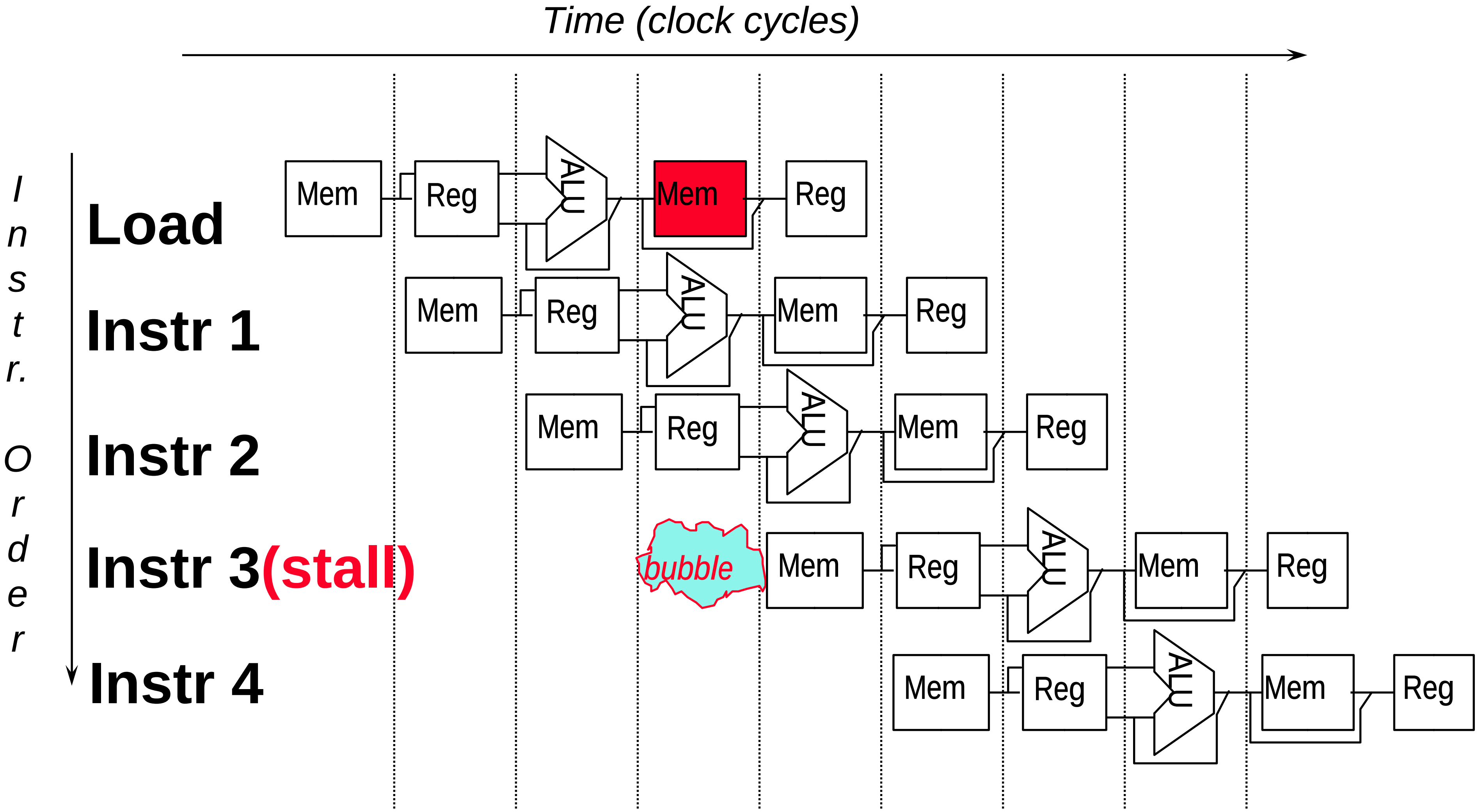
It’s not that easy for computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - structural hazards: HW cannot support this combination of instructions
  - data hazards: instruction depends on result of prior instruction still in the pipeline
  - control hazards: pipelining of branches & other instructions that change the PC
- Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline

Single Memory is a Structural Hazard

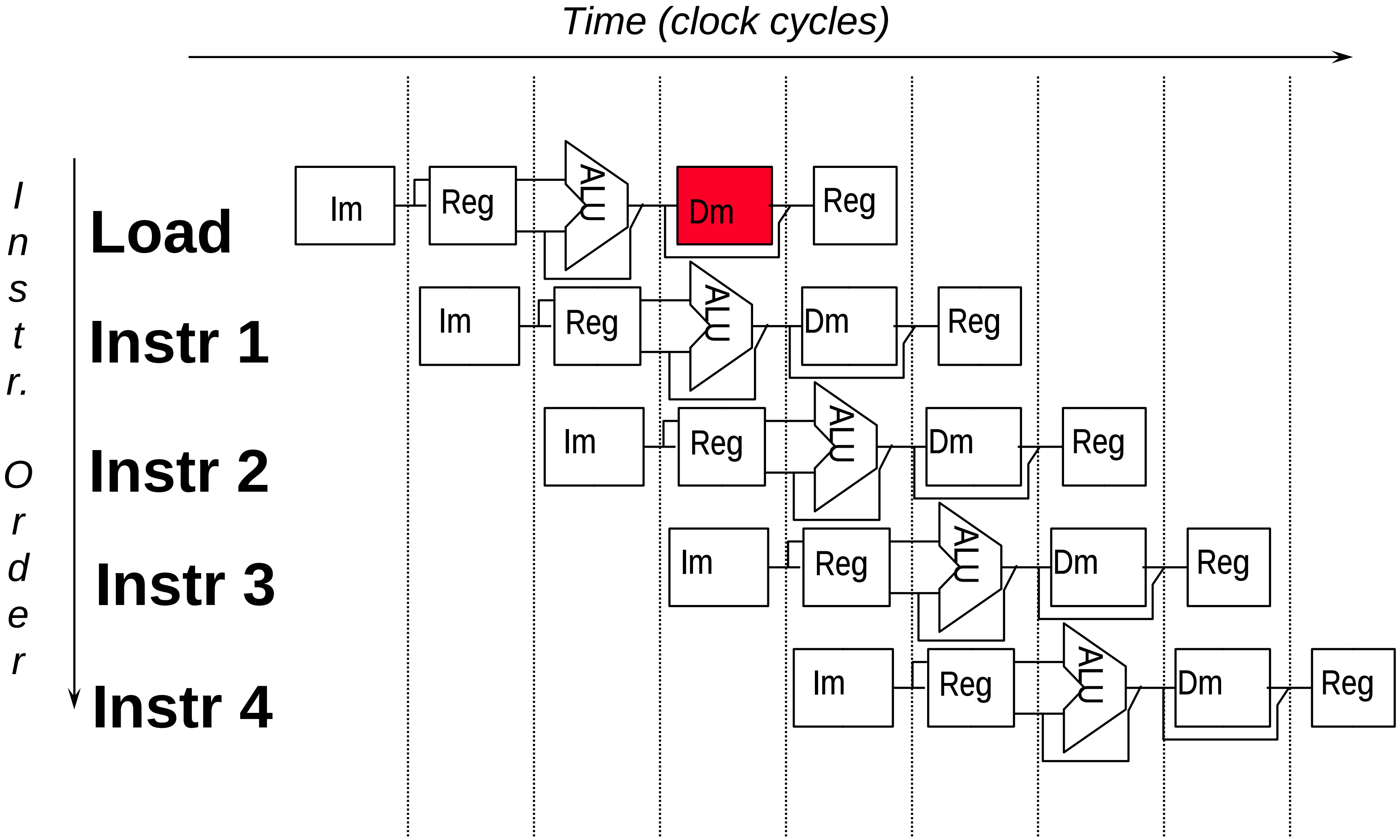


Option 1: Stall to resolve Memory Structural Hazard



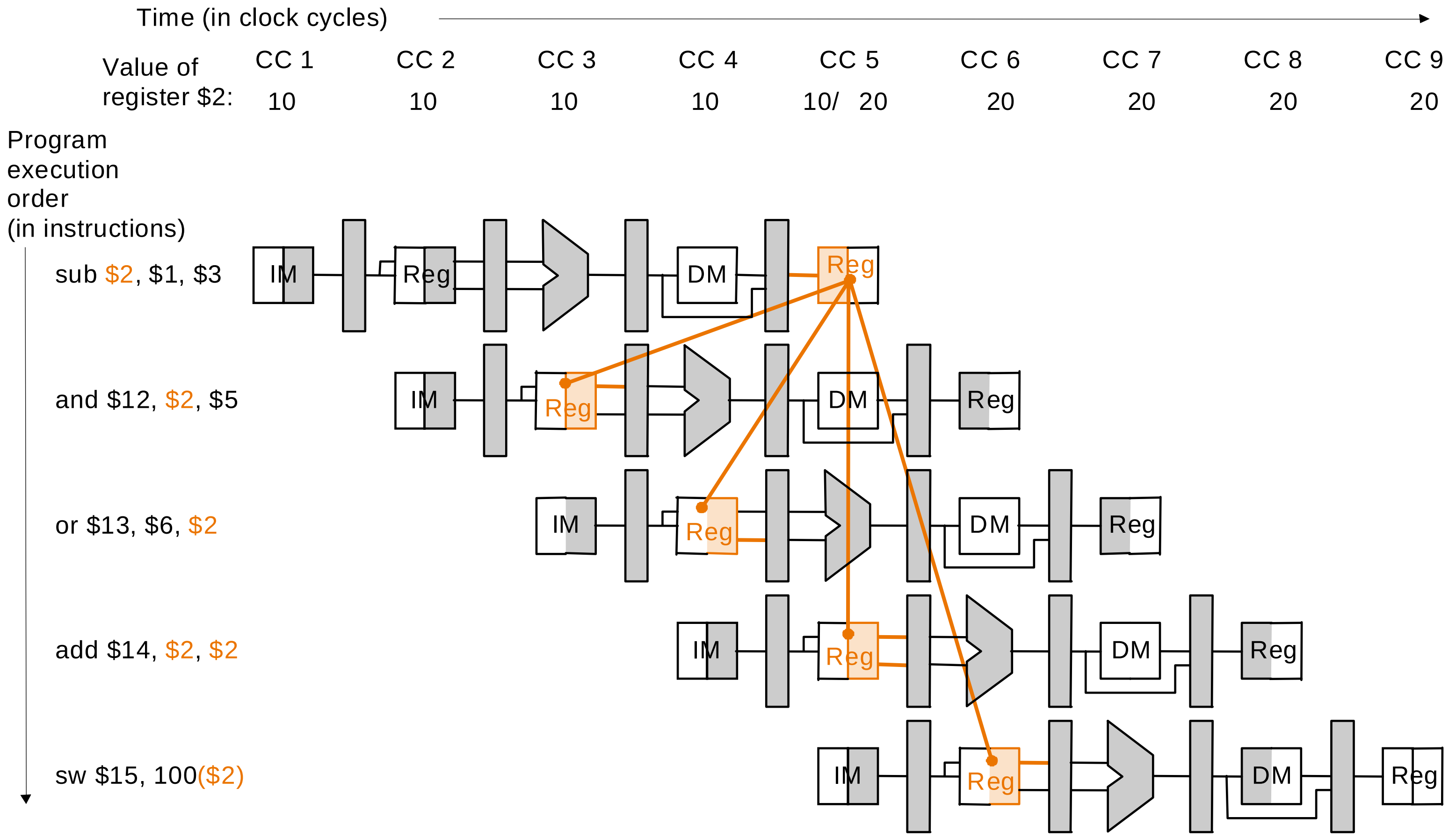
Option 2: Duplicate to Resolve Structural Hazard

- Separate Instruction Cache (Im) & Data Cache (Dm)



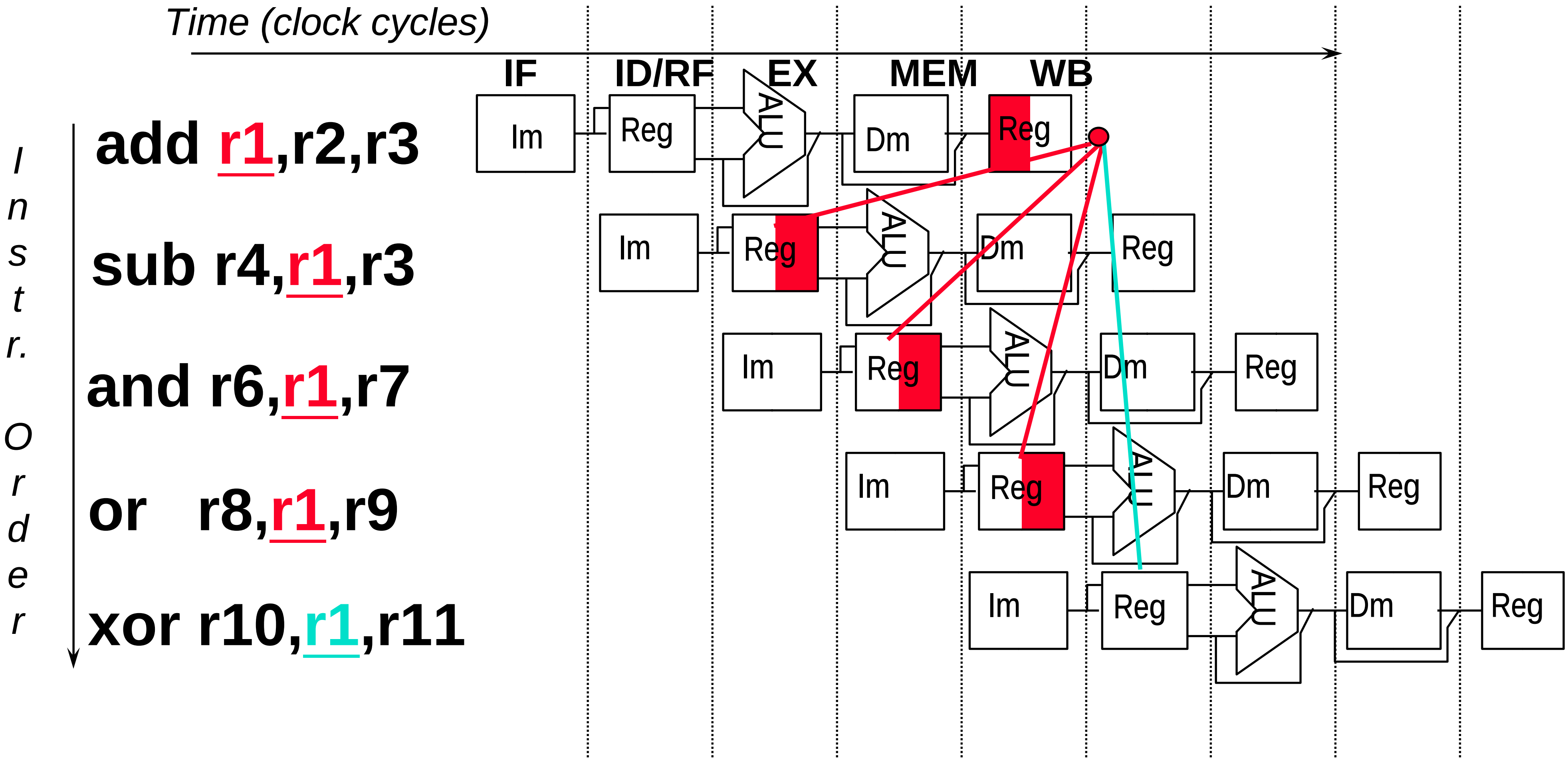
Dependencies

- Problem with starting next instruction before first is finished
  - dependencies that “go backward in time” are data hazards



Data Hazard on r1:

- Dependencies backwards in time are hazards



add r1,r2,r3

sub r4,r1,r3

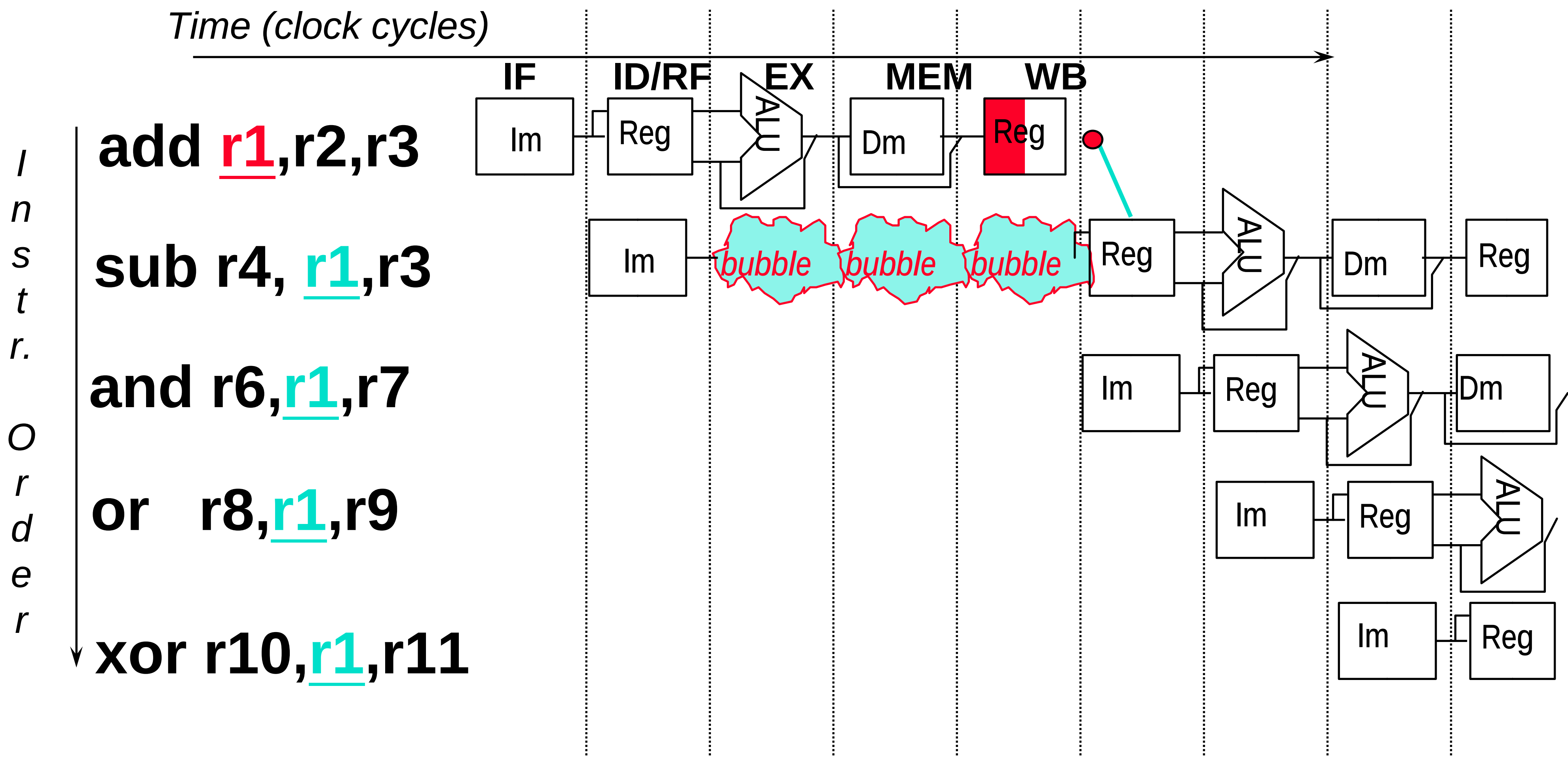
and r6,r1,r7

or r8,r1,r9

xor r10,r1,r11

Option1: HW Stalls to Resolve Data Hazard

- Dependencies backwards in time are hazards



Time (clock cycles)

Instr. Order

add r1,r2,r3

sub r4, r1,r3

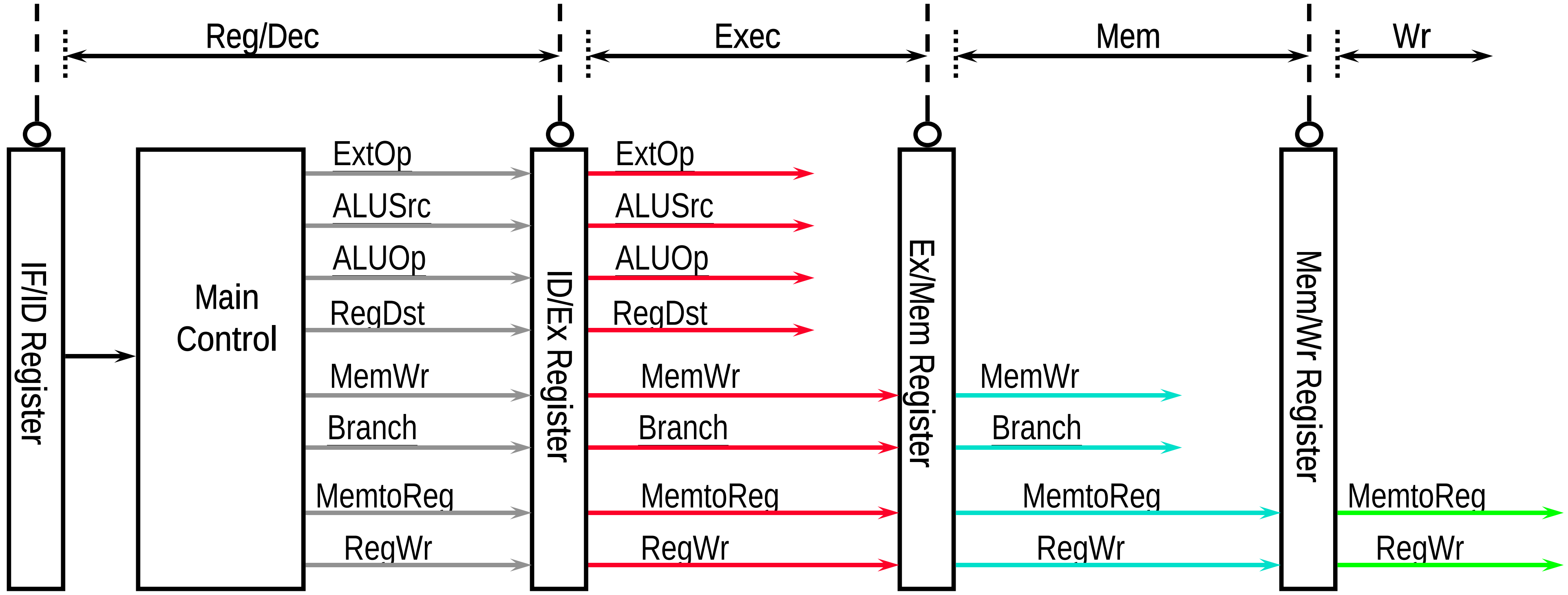
and r6,r1,r7

or   r8,r1,r9

xor r10,r1,r11

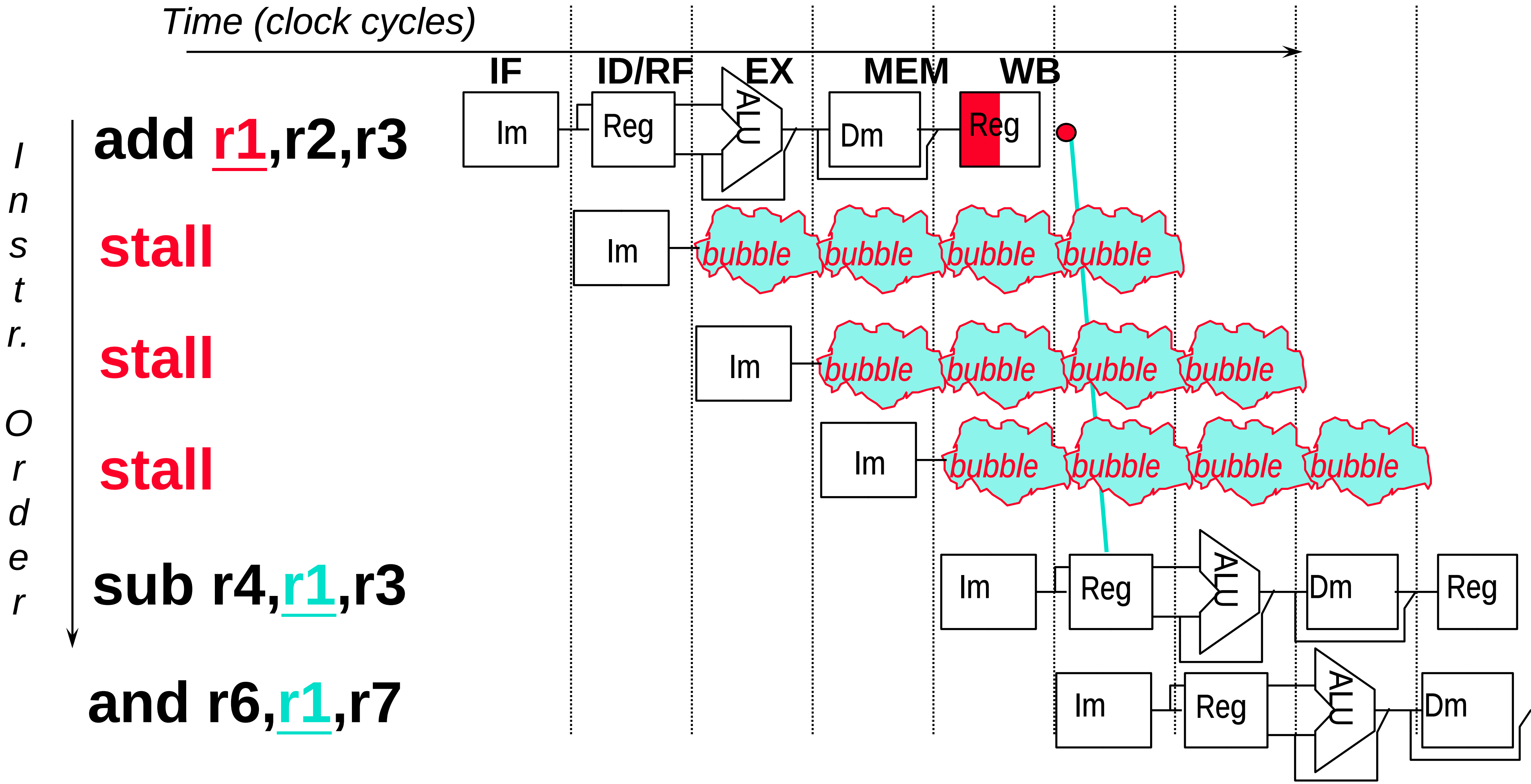
But recall use of “Data Stationary Control”

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Option 1: How HW really stalls pipeline

- HW doesn't change PC => keeps fetching same instruction & sets control signals to benign values (0)



add r1,r2,r3

stall

stall

stall

sub r4,r1,r3

and r6,r1,r7

Software Solution

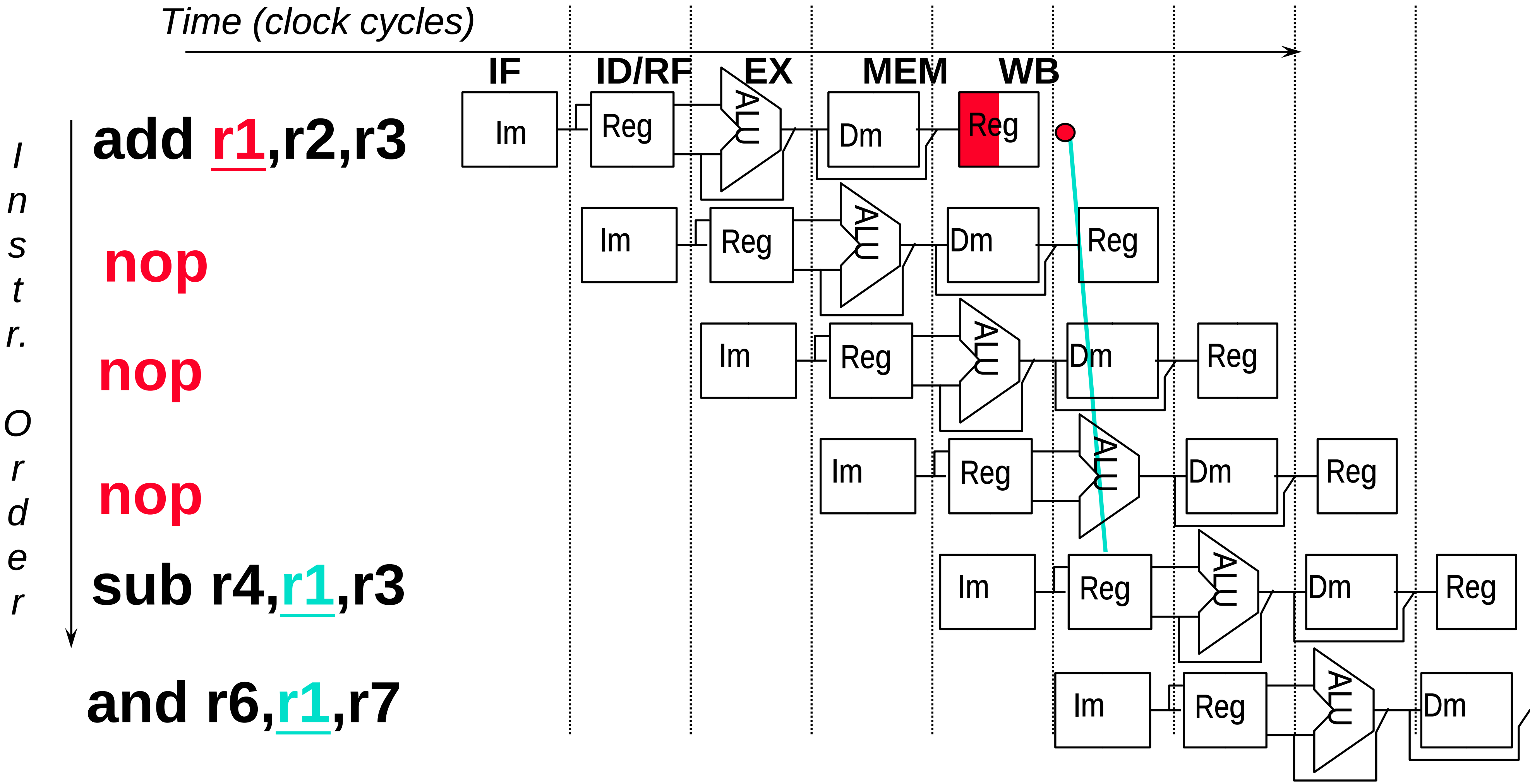
- Have compiler guarantee no hazards
- Where do we insert the “nops” ?

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

- Problem: this really slows us down!

Option 2: SW inserts independent instructions

- Worst case inserts NOP instructions



add r1,r2,r3

nop

nop

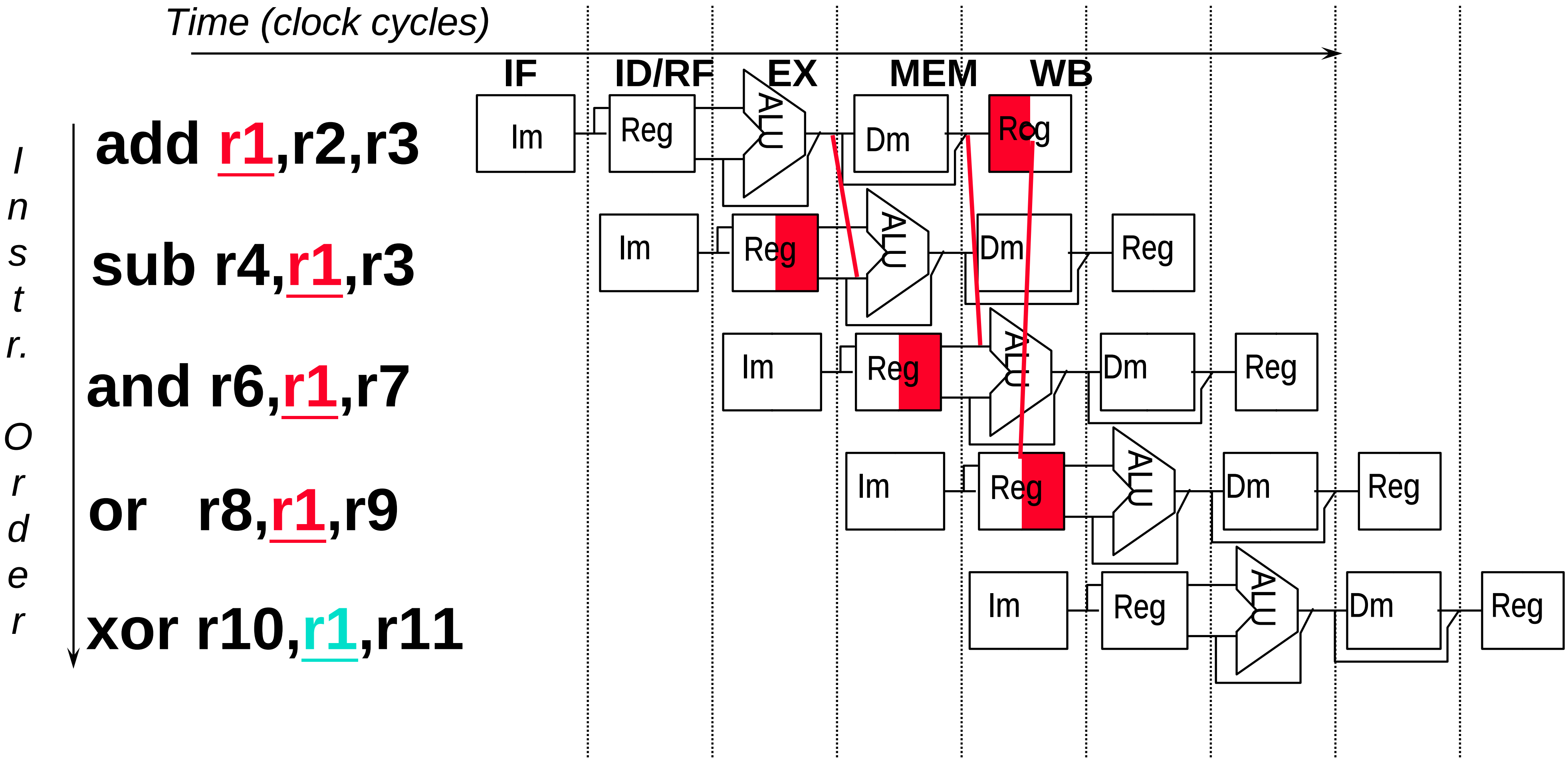
nop

sub r4,r1,r3

and r6,r1,r7

Option 3 Insight: Data is available!

- Pipeline registers already contain needed data



Time (clock cycles)

IF ID/RF EX MEM WB

Instr. Order

add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

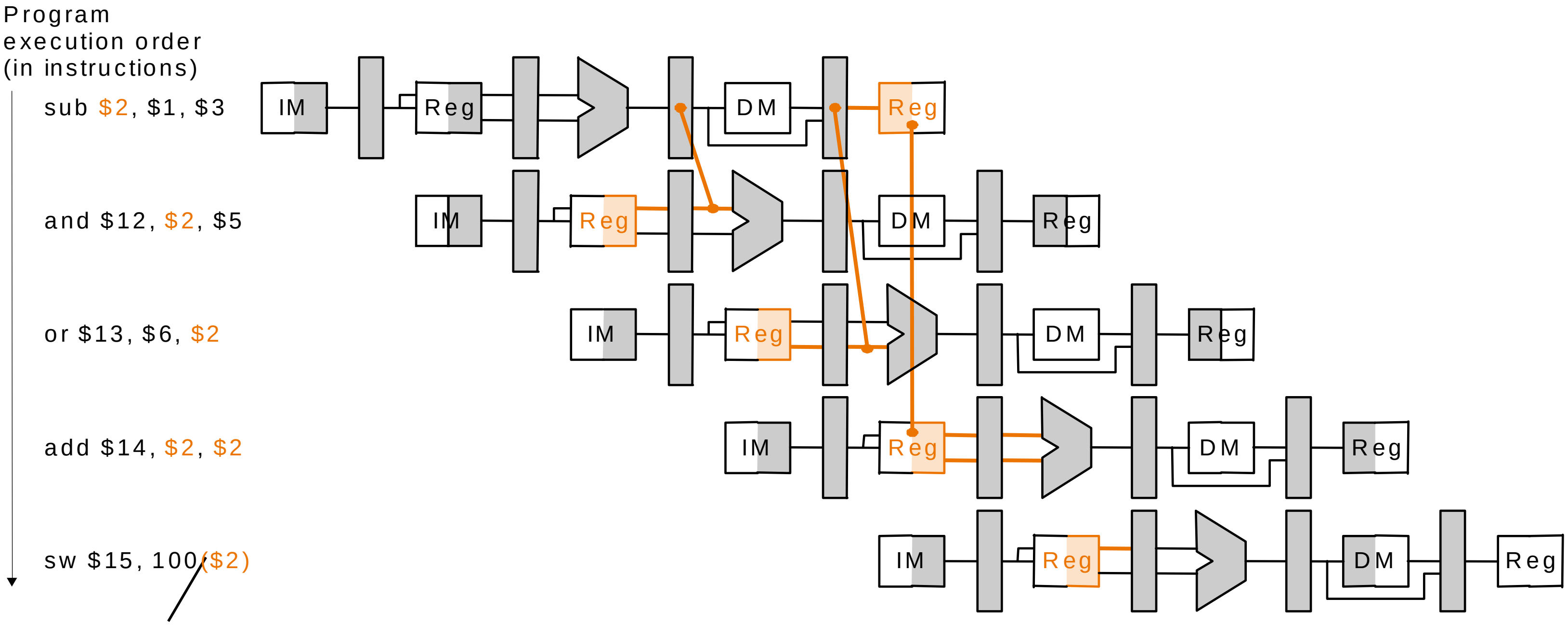
xor r10,r1,r11

Forwarding

- Use temporary results, don't wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding

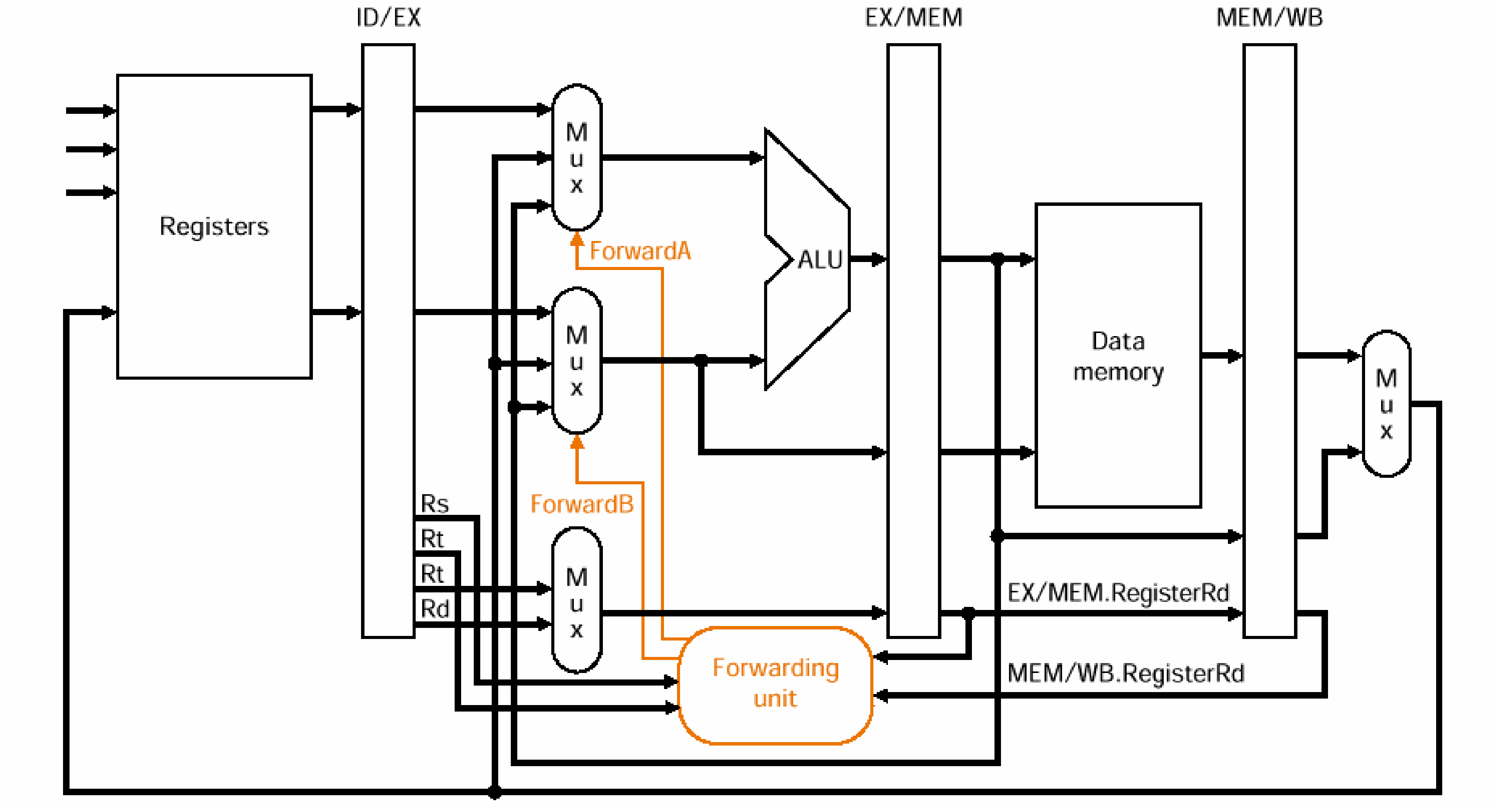
Time (in clock cycles) →

	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



what if this $2 was $13?

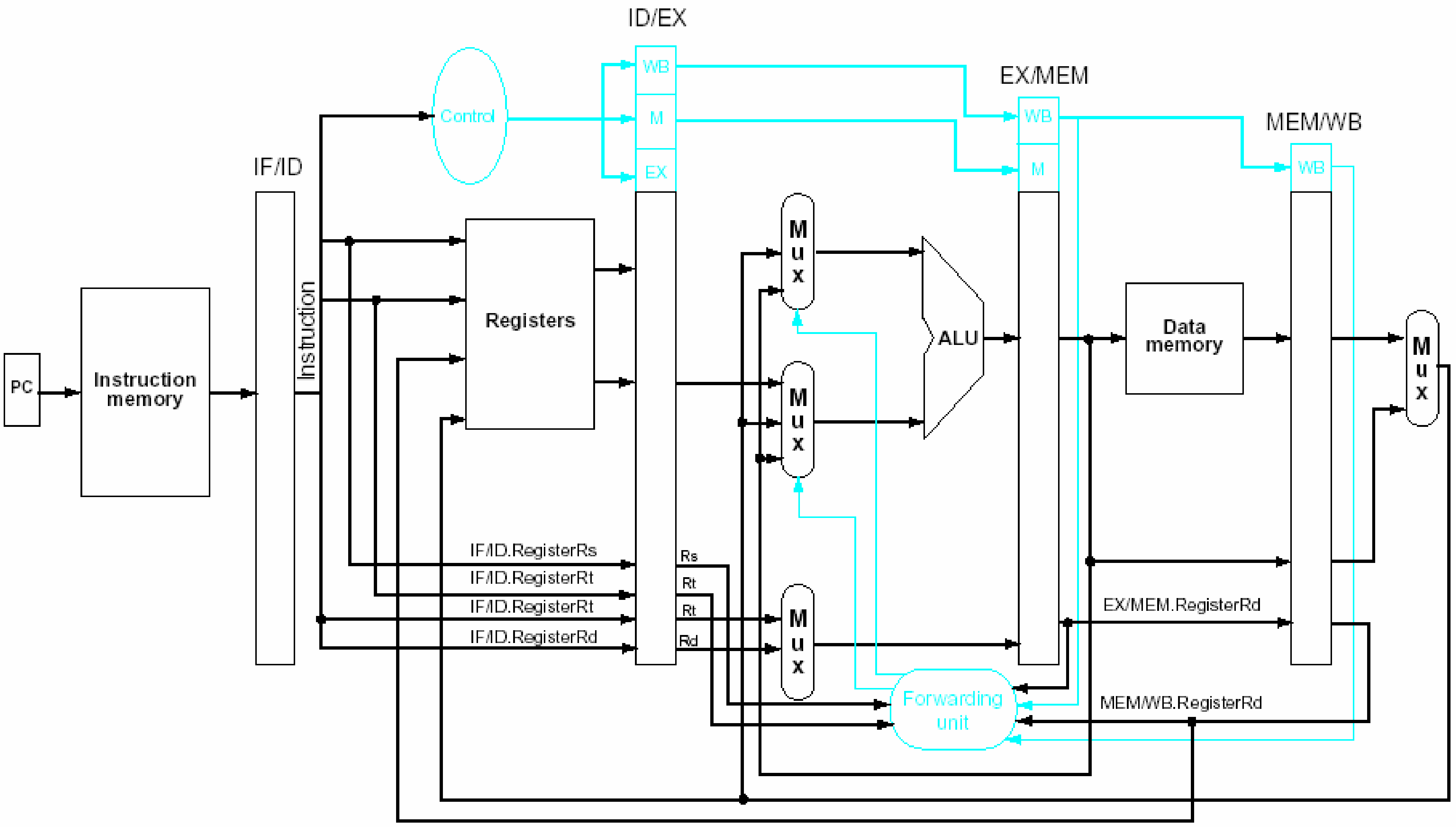
HW Change for “Forwarding” (Bypassing):



Conditions for Detecting Hazards

- EX/MEM 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/WB 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

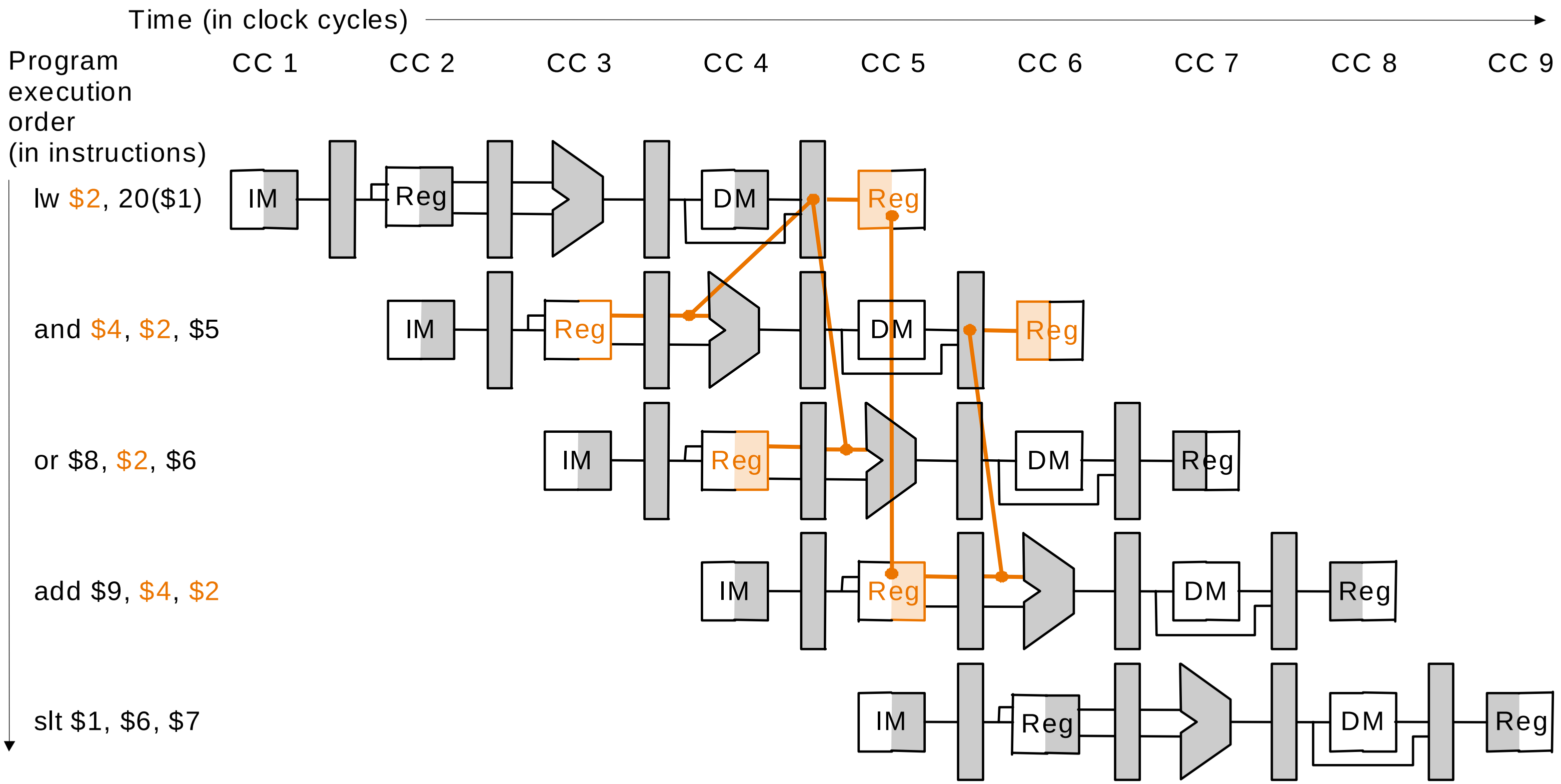
Forwarding



ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
WB
M
WB
PC
Instruction memory
Instruction
Registers
M u x
M u x
M u x
ALU
Data memory
M u x
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
Rs
Rt
Rt
Rd
EX/MEM.RegisterRd
MEM/WB.RegisterRd
Forwarding unit

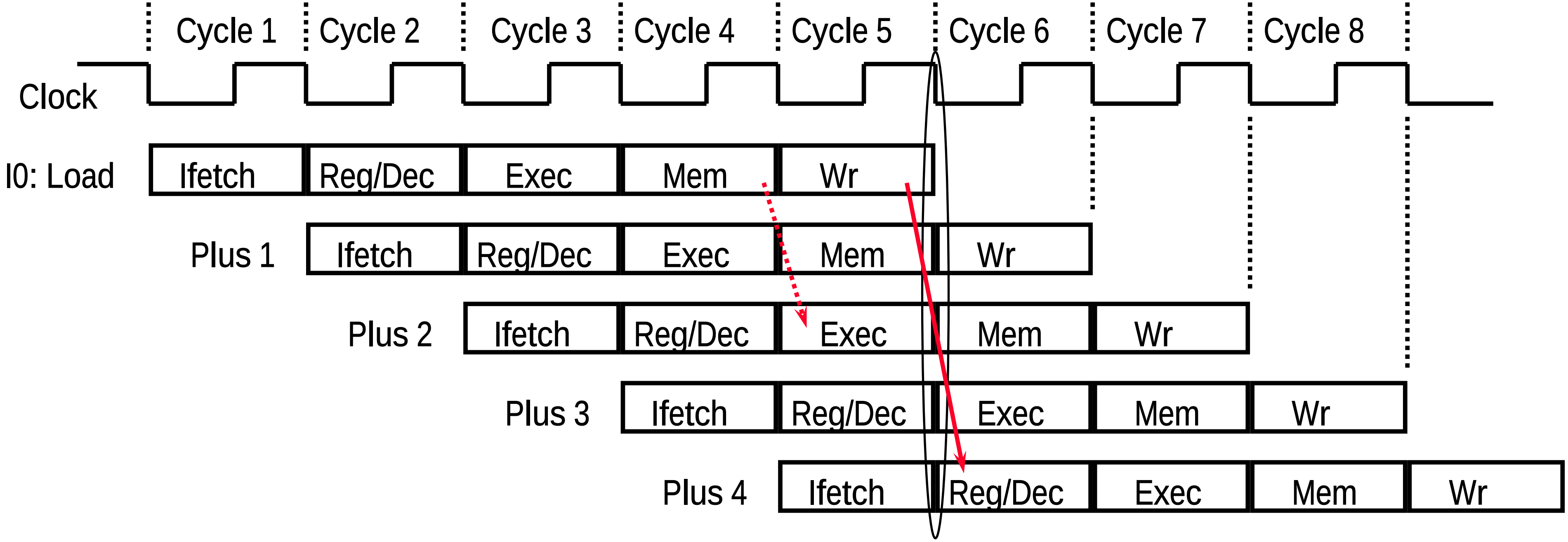
Can't always forward

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.



- Thus, we need a hazard detection unit to “stall” the load instruction

From Last Lecture: The Delay Load Phenomenon



- Although Load is fetched during Cycle 1:
  - The data is NOT written into the Reg File until the end of Cycle 5
  - We cannot read this value from the Reg File until Cycle 6
  - 3-instruction delay before the load take effect

Forwarding reduces Data Hazard to 1 cycle:

Time (clock cycles)

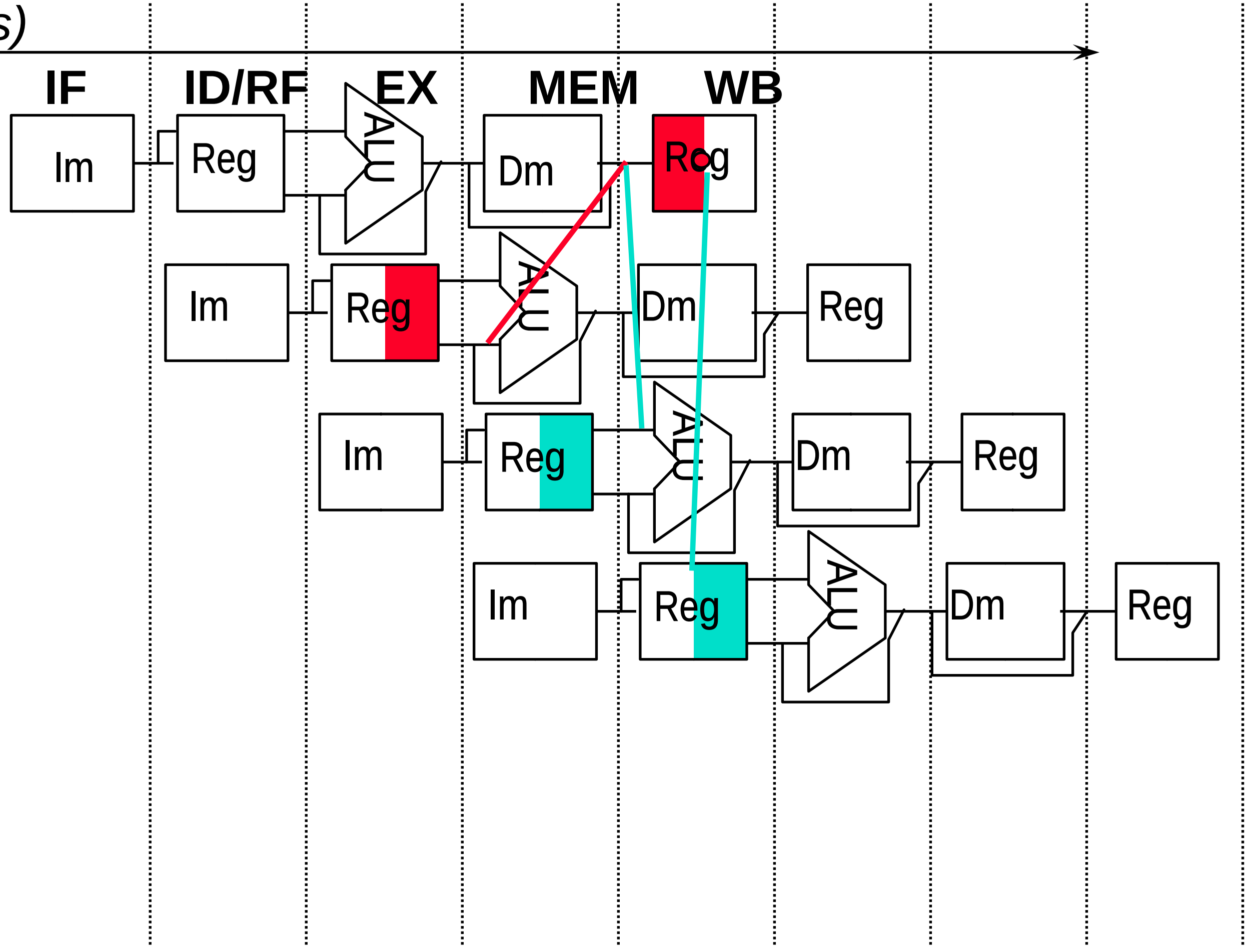
Instr. Order

lw r1, 0(r2)

sub r4,r1,r6

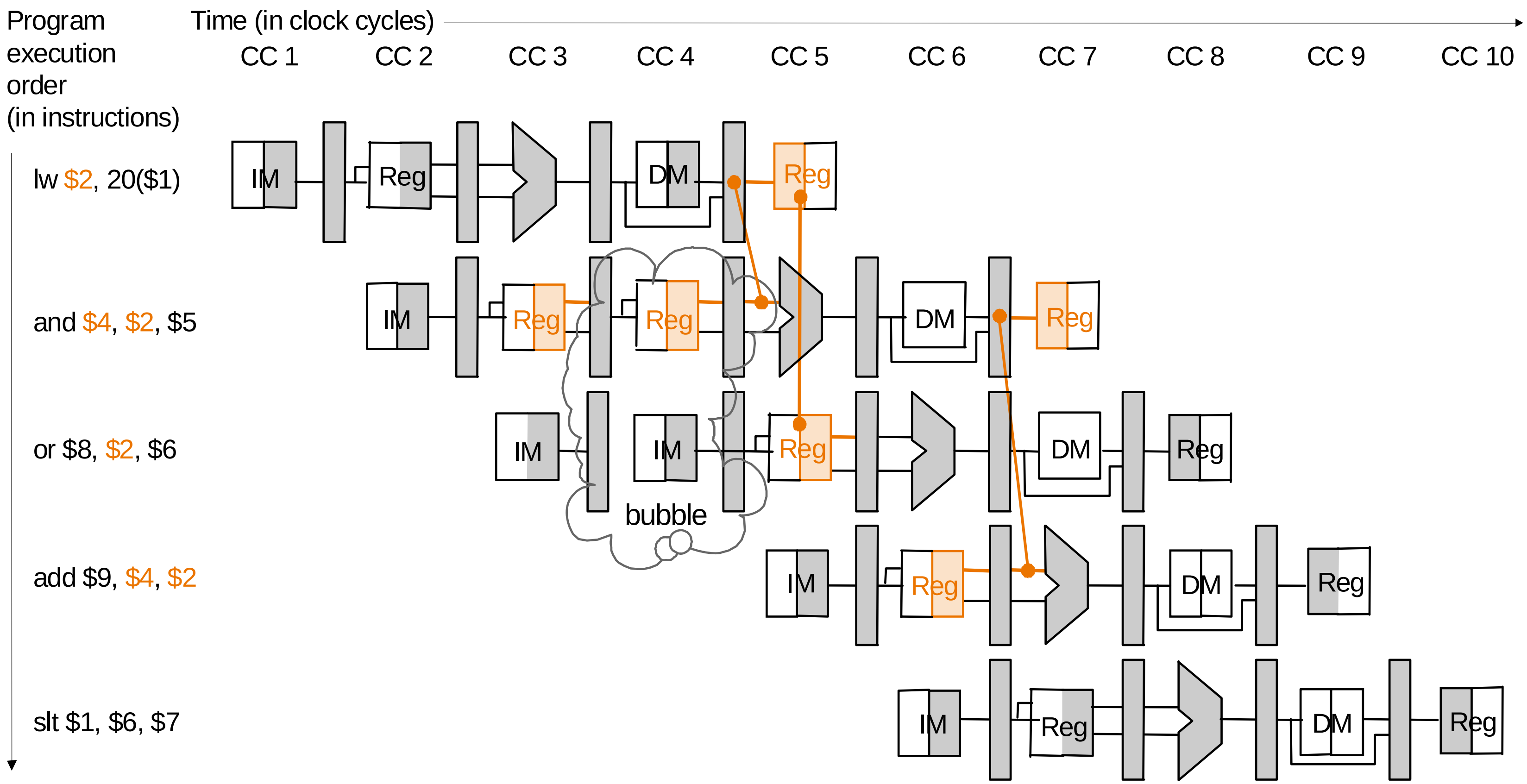
and r6,r1,r7

or   r8,r1,r9



Stalling

- We can stall the pipeline by keeping an instruction in the same stage



Condition for Load-Use Hazard Detection

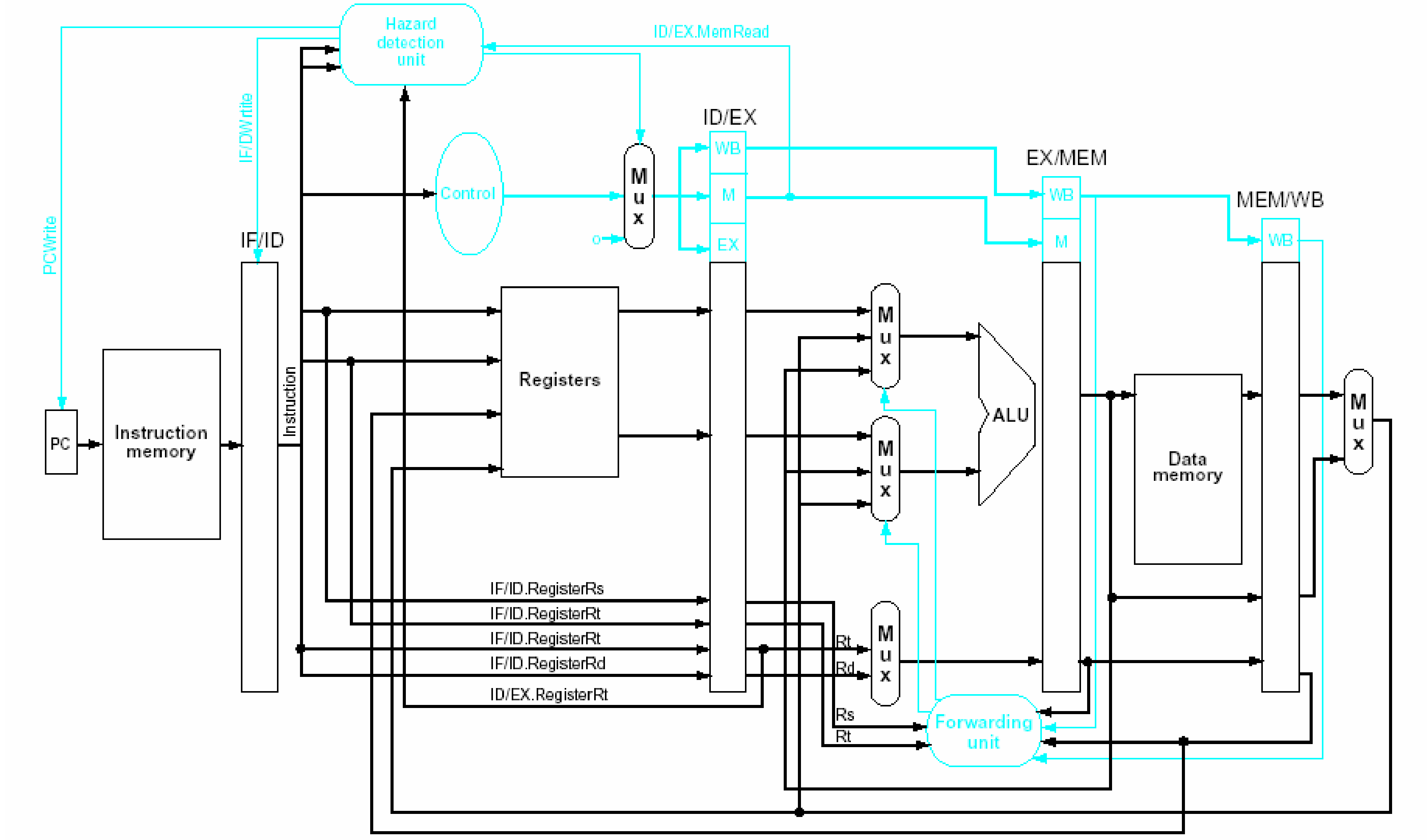
- Checking for load instructions, the control for the load-use hazard condition is as follows:
  - if (ID / EX.MemRead

    and ((ID / EX.RegisterRt = IF / ID.RegisterRs) or

    (ID / EX.RegisterRt = IF / ID.RegisterRt)) stall the pipeline

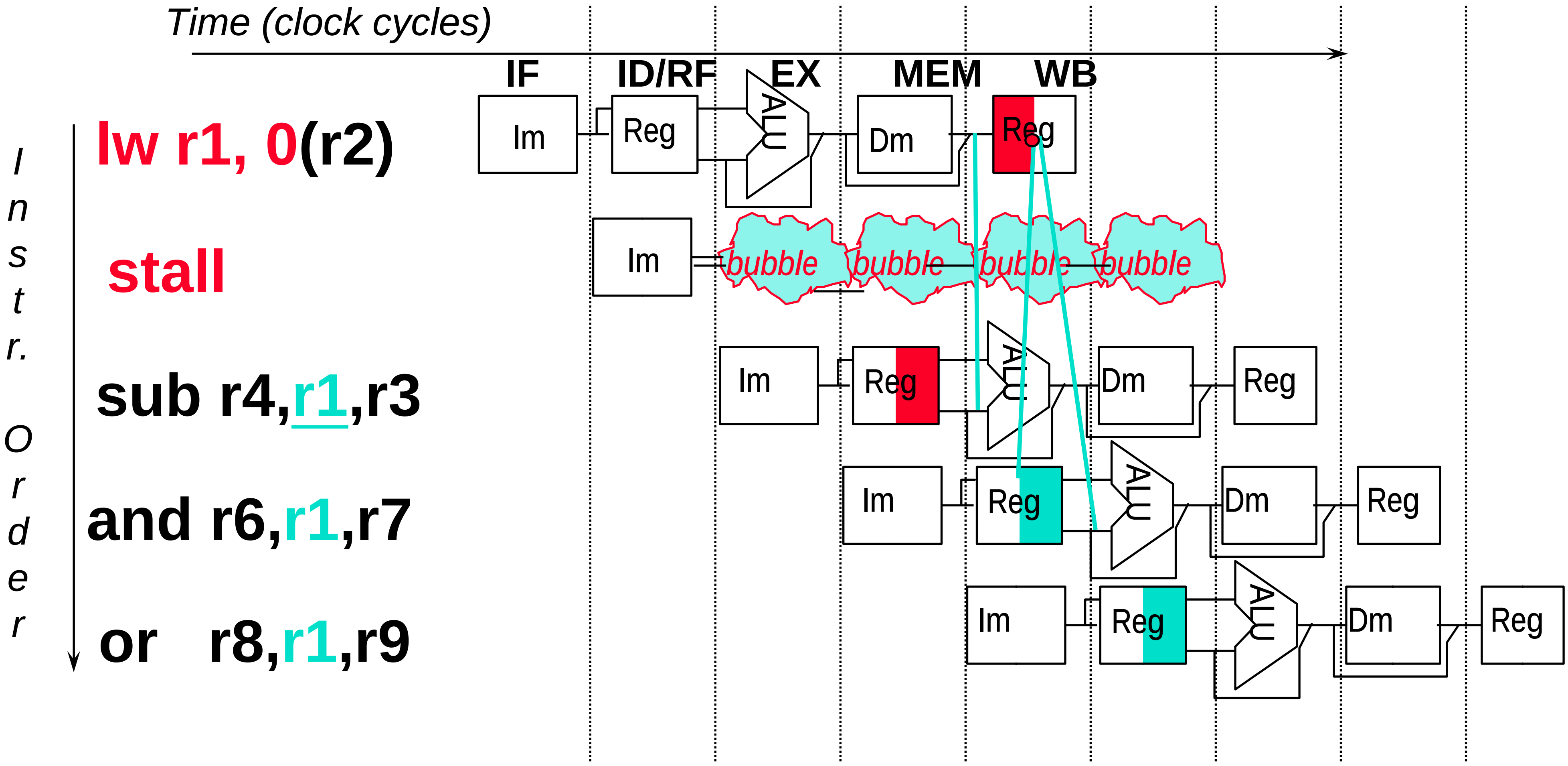
Hazard Detection Unit

- Stall by letting an instruction that won’t write anything go forward



Option1: HW Stalls to Resolve Data Hazard

- “Interlock” checks for hazard & stalls



Time (clock cycles)

Instr. Order

lw r1, 0(r2)

stall

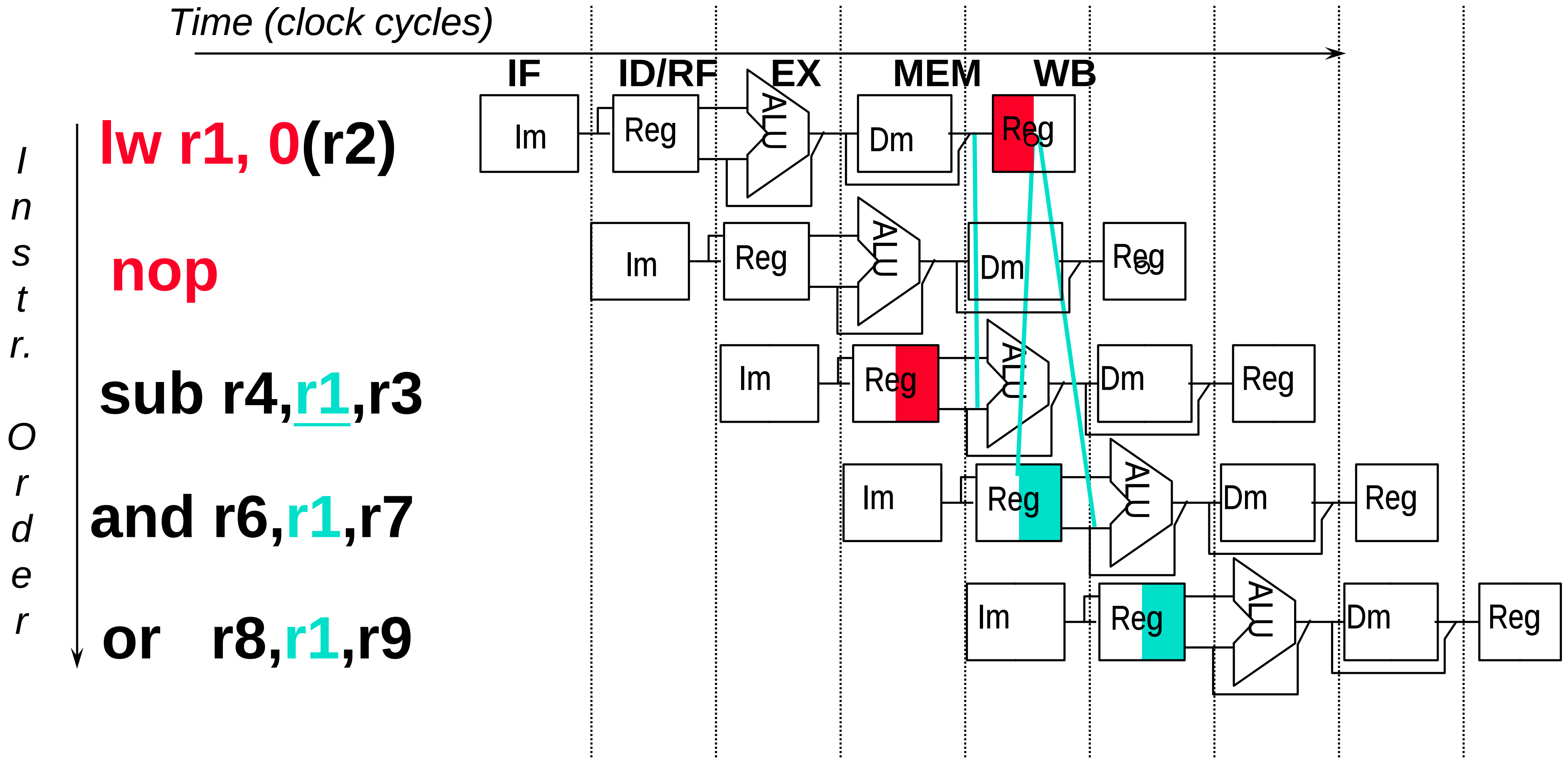
sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

Option 2: SW inserts independent instructions

- Worst case inserts NOP instructions
- MIPS I solution: No HW checking



lw r1, 0(r2)

nop

sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

Software Scheduling to Avoid Load Hazards

Try producing fast code for

```
a = b + c;
d = e - f;
```

assuming a, b, c, d ,e, and f in memory.

Slow code:

```
LW      Rb,b
LW      Rc,c
ADD     Ra,Rb,Rc
SW      a,Ra
LW      Re,e
LW      Rf,f
SUB     Rd,Re,Rf
SW      d,Rd
```

Software Scheduling to Avoid Load Hazards

Try producing fast code for

```
a = b + c;
d = e - f;
```

assuming a, b, c, d ,e, and f in memory.

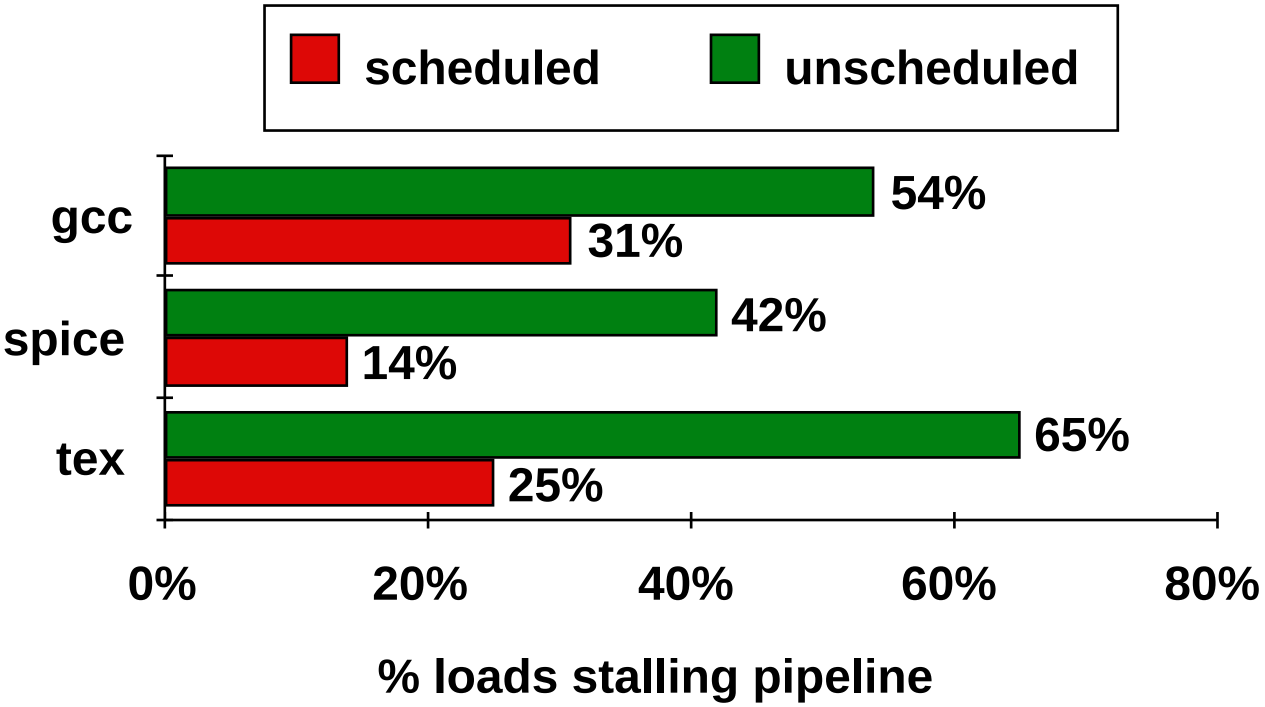
Slow code:

```
LW      Rb,b
LW      Rc,c
ADD     Ra,Rb,Rc
SW      a,Ra
LW      Re,e
LW      Rf,f
SUB     Rd,Re,Rf
SW      d,Rd
```

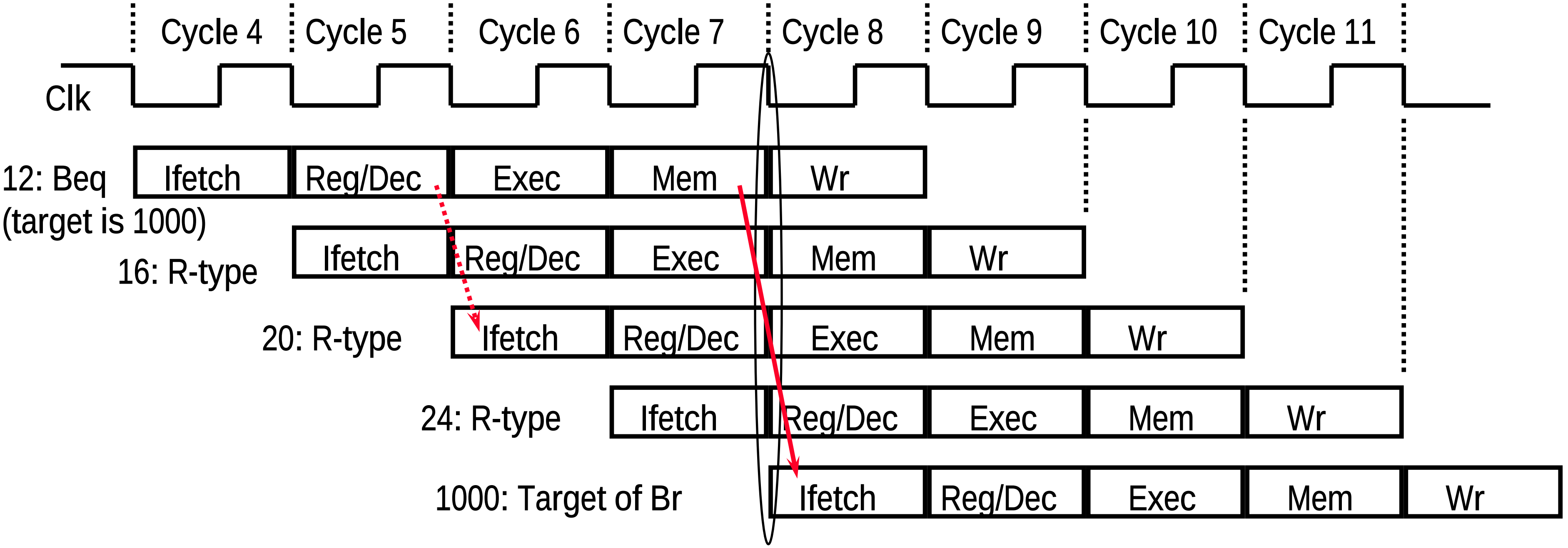
Fast code:

```
LW      Rb,b
LW      Rc,c
LW      Re,e
ADD     Ra,Rb,Rc
LW      Rf,f
SW      a,Ra
SUB     Rd,Re,Rf
SW      d,Rd
```

Compiler Avoiding Load Stalls:



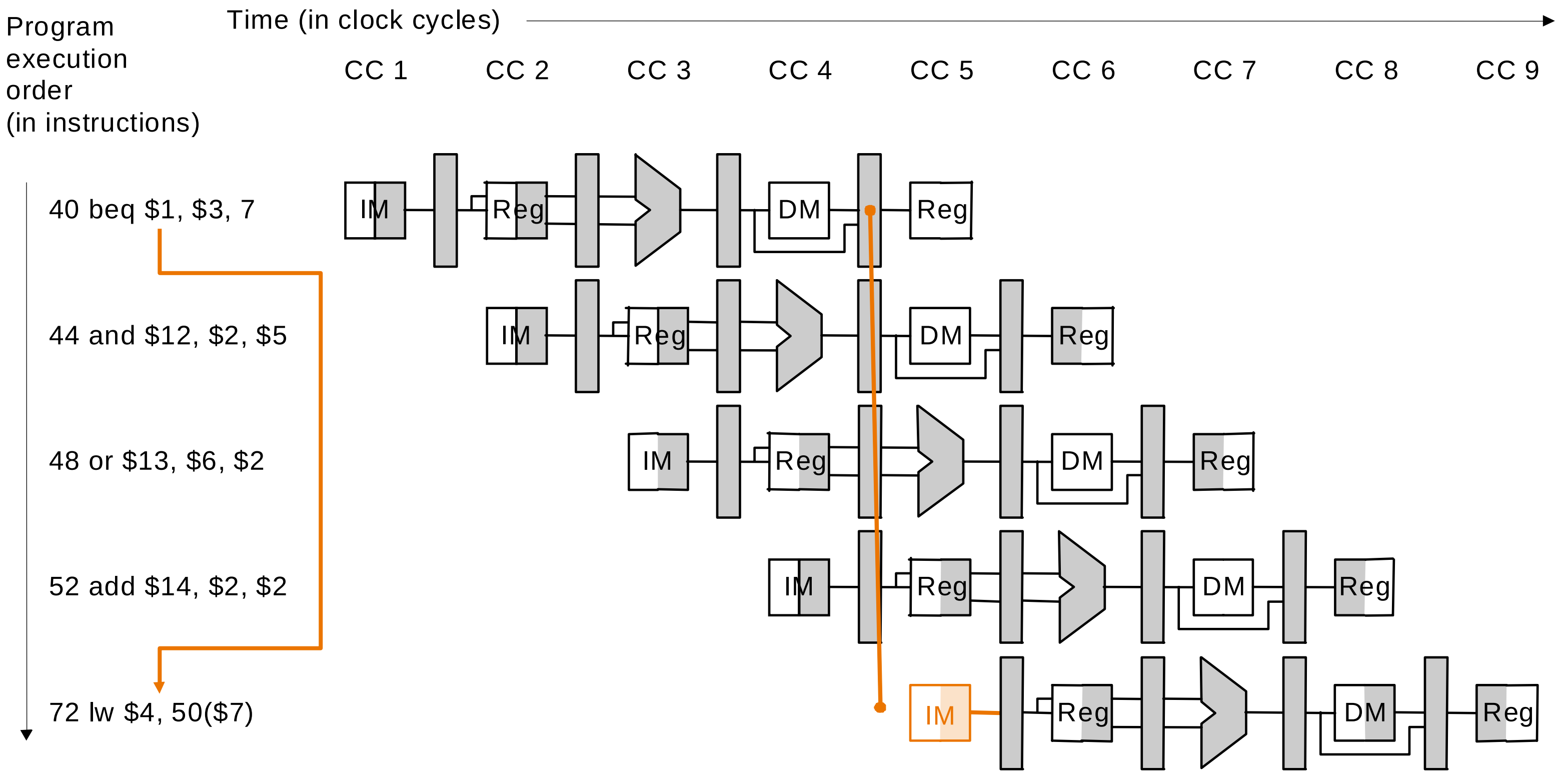
From Last Lecture: The Delay Branch Phenomenon



	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11	
12: Beq (target is 1000)	Ifetch	Reg/Dec	Exec	Mem	Wr				
16: R-type		Ifetch	Reg/Dec	Exec	Mem	Wr			
20: R-type			Ifetch	Reg/Dec	Exec	Mem	Wr		
24: R-type				Ifetch	Reg/Dec	Exec	Mem	Wr	
1000: Target of Br					Ifetch	Reg/Dec	Exec	Mem	Wr

- **Although Beq is fetched during Cycle 4:**
  - **Target address is NOT written into the PC until the end of Cycle 7**
  - **Branch's target is NOT fetched until Cycle 8**
  - **3-instruction delay before the branch take effect**

Control Hazard on Branches: 3 stage stall



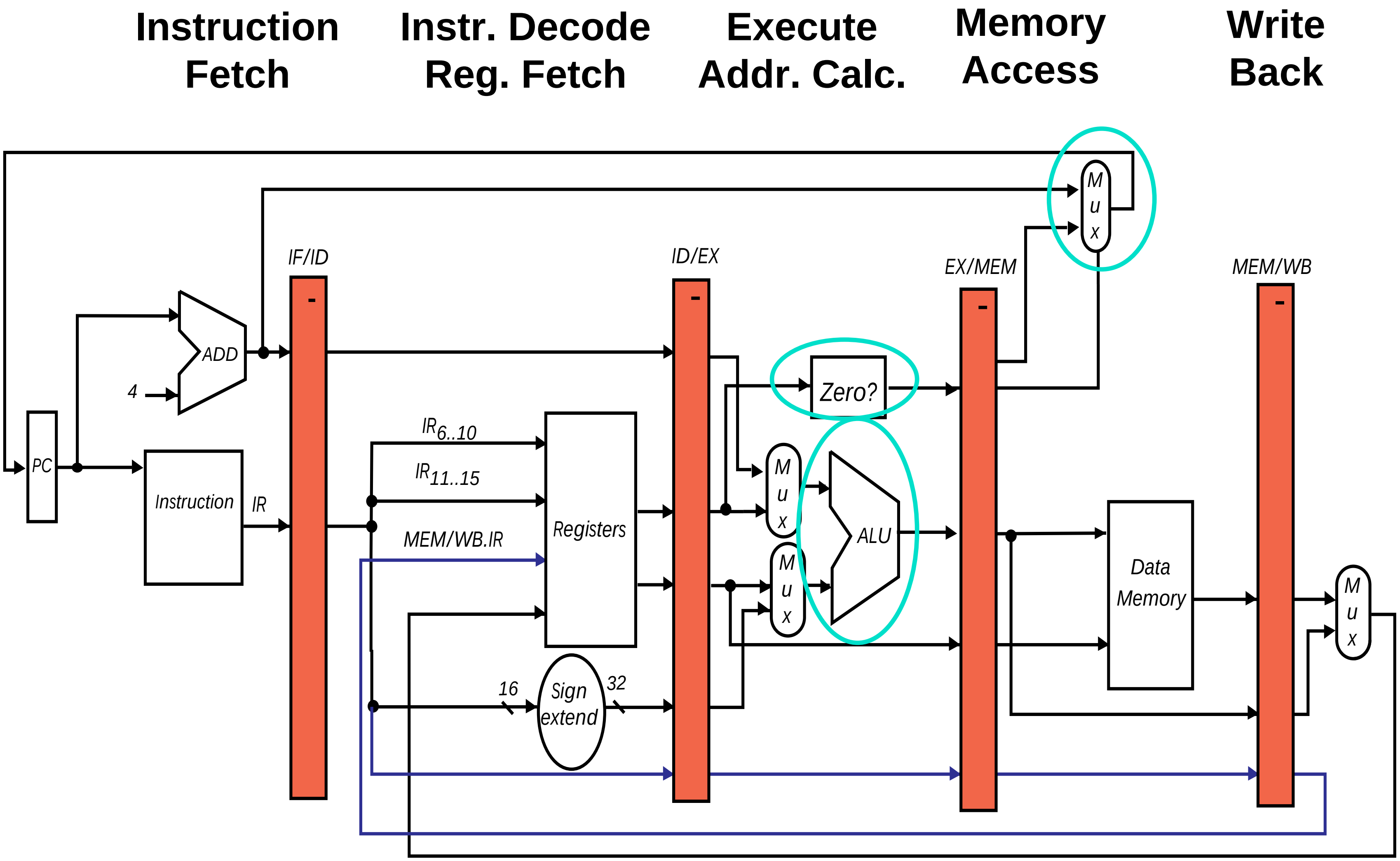
Lec 15 hazards.36

© MKP 2004

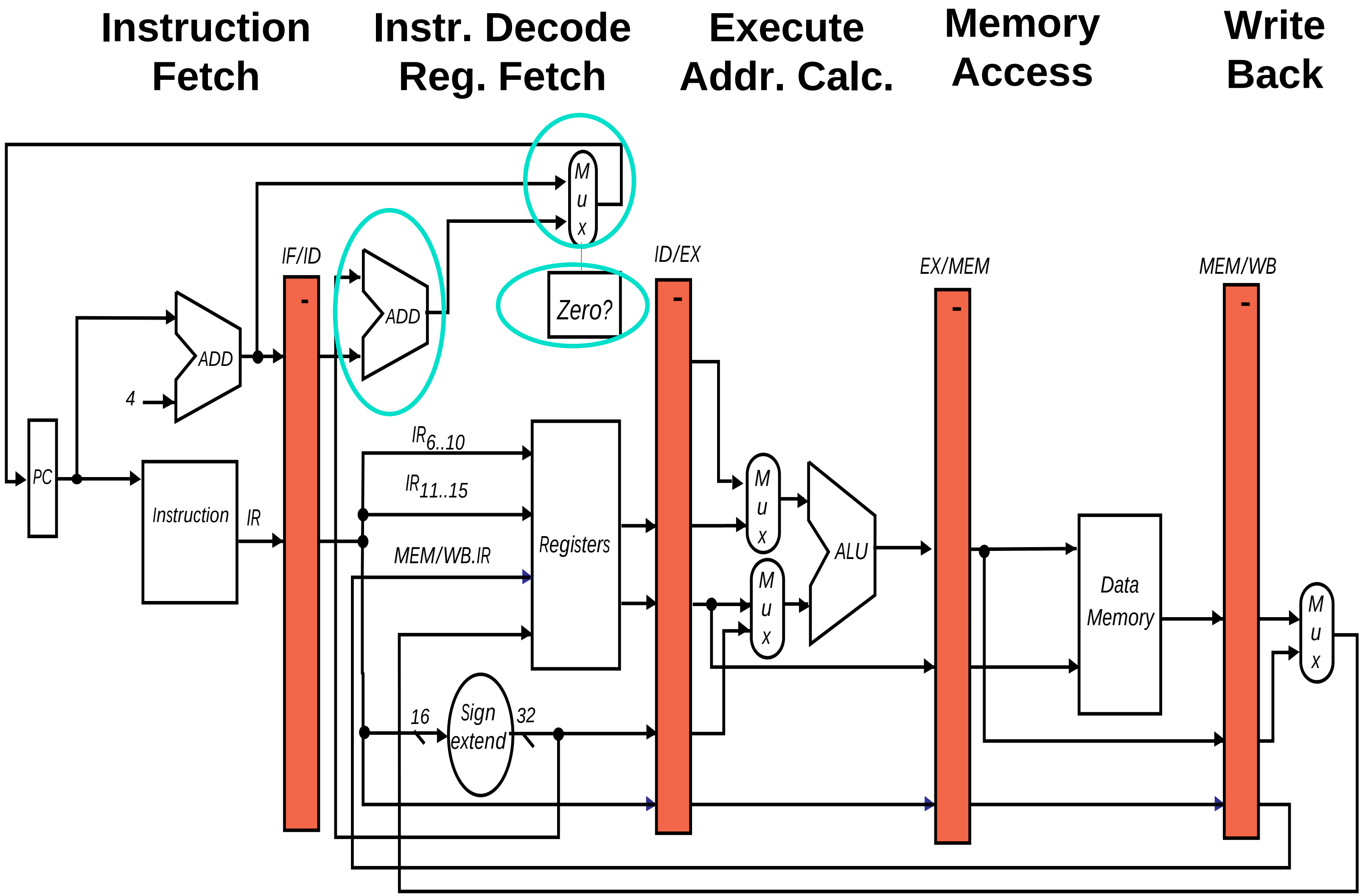
Branch Stall Impact

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

Option 1: move HW forward to reduce branch delay



Branch Delay now 1 clock cycle



Option 2: Define Branch as Delayed

- Worst case, SW inserts NOP into branch delay
- Where get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch
  - From fall through: only valuable when don’t branch
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - about 50% (60% x 80%) of slots usefully filled

Moving Branch Decisions Earlier in Pipe

- **Move the branch decision hardware back to the EX stage**
  - **Reduces the number of stall cycles to two**
  - **Adds an and gate and a 2x1 mux to the EX timing path**
- **Add hardware to compute the branch target address and evaluate the branch decision to the ID stage**
  - **Reduces the number of stall cycles to one (like with jumps)**
  - **Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)**
  - **Comparing the registers can't be done until after RegFile read, so comparing and updating the PC adds a comparator, an and gate, and a 3x1 mux to the ID timing path**
  - **Need forwarding hardware in ID stage**
- **For longer pipelines, decision points are later in the pipeline, incurring more stalls, so we need a better solution**

Early Branch Forwarding Issues

- Bypass of source operands from the EX/MEM

```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == IF/ID.RegisterRs))
      ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == IF/ID.RegisterRt))
      ForwardD = 1
```

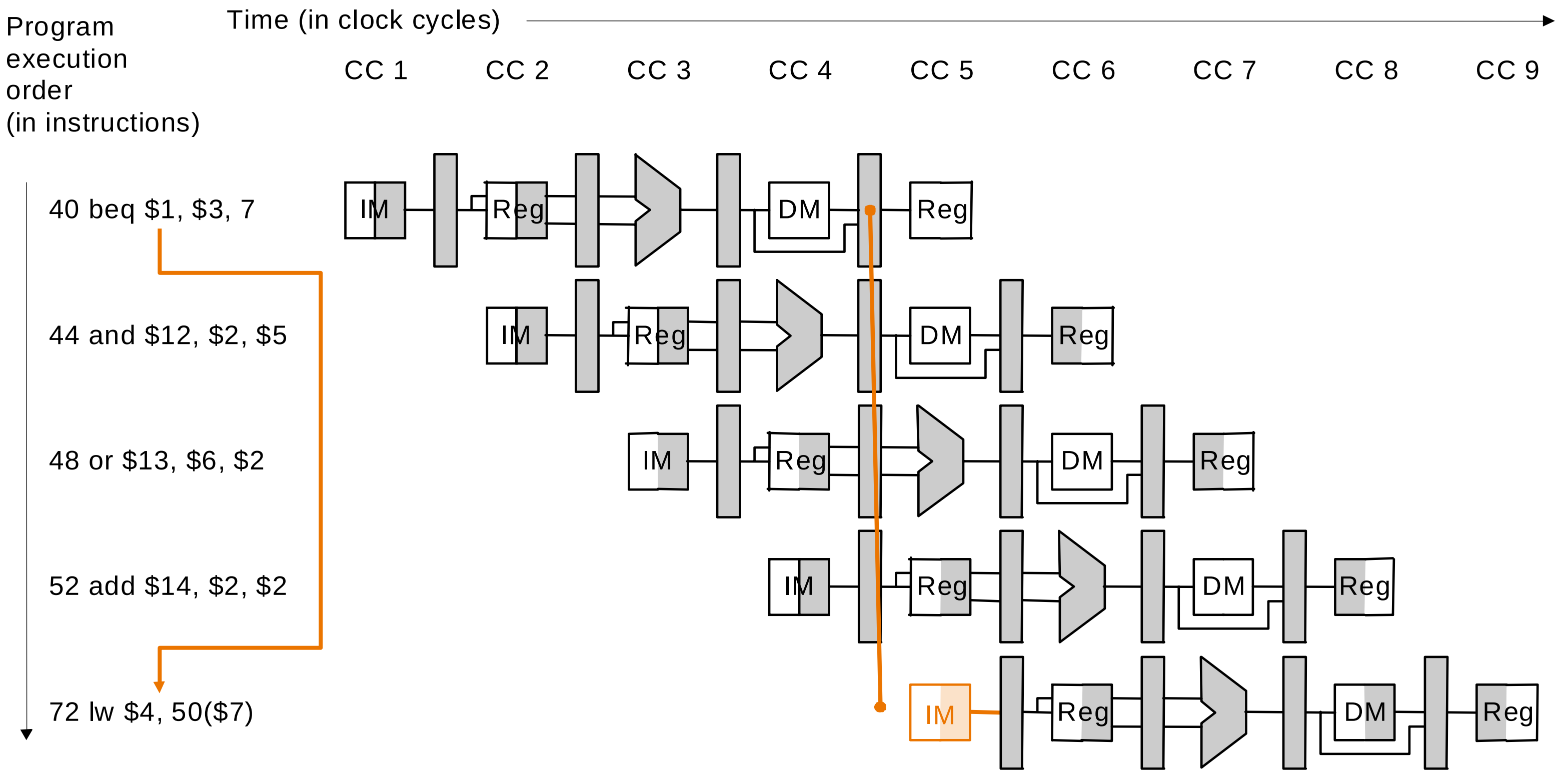
Forwards the result from the second previous instr. to either input of the Compare

- MEM/WB dependency also needs to be forwarded

- If the instruction 2 before the branch is a load, then a stall will be required since the MEM stage memory access is occurring at the same time as the ID stage branch compare operation

Branch Hazards

- When we decide to branch, other instructions are in the pipeline!



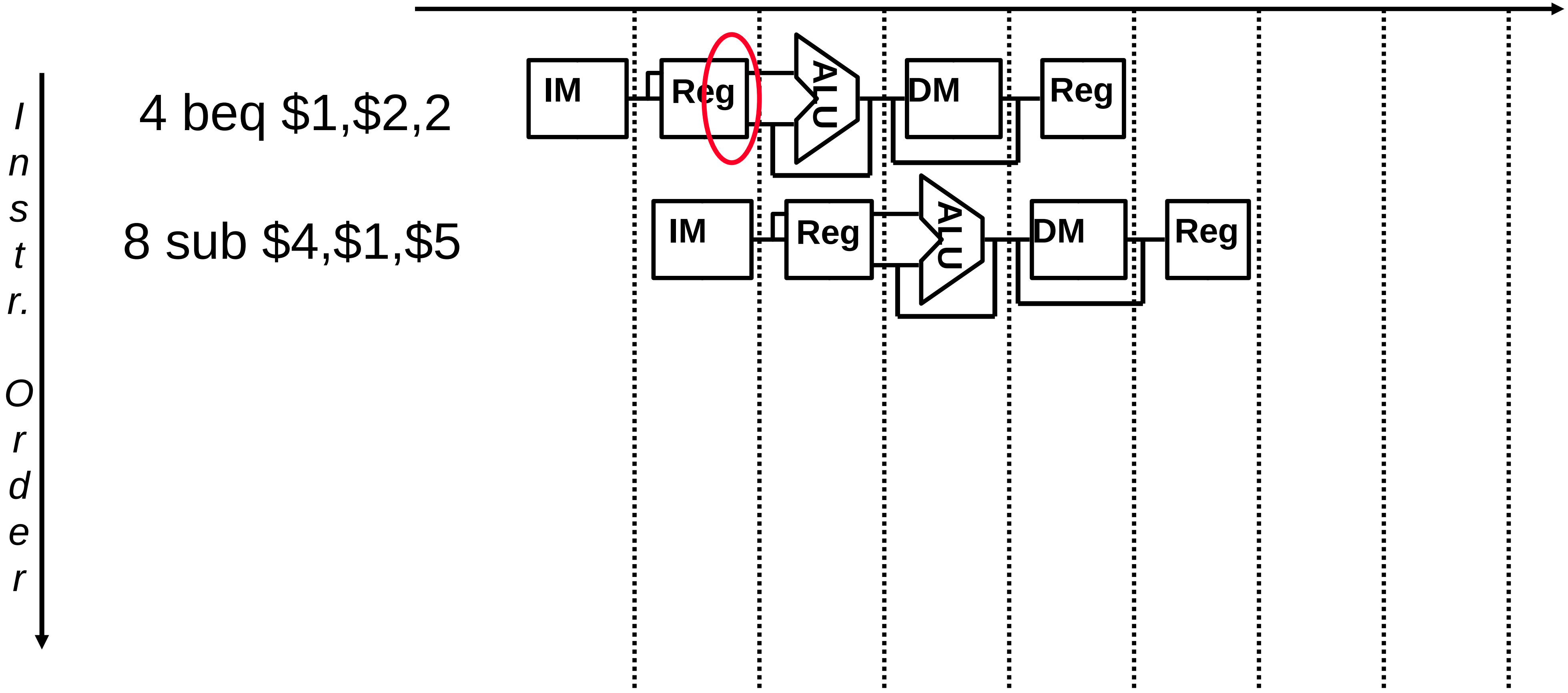
- We are predicting “branch not taken”
  - need to add hardware for flushing instructions if we are wrong

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 in the pipeline after the branch
     - ⇒ in IF, ID, and EX if branch logic in MEM – three stalls
     - ⇒ in IF if branch logic in ID – one stall
   - ensure that those flushed instructions haven't changed machine state– automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite or RegWrite)
   - restart the pipeline at the branch destination

Flushing with Misprediction (Not Taken)



- To flush the IF stage instruction, add a IF.Flush control line that zeros the instruction field of the IF/ID pipeline register (transforming it into a noop)

Flushing with Misprediction (Not Taken)

I n s t r.

O r d e r

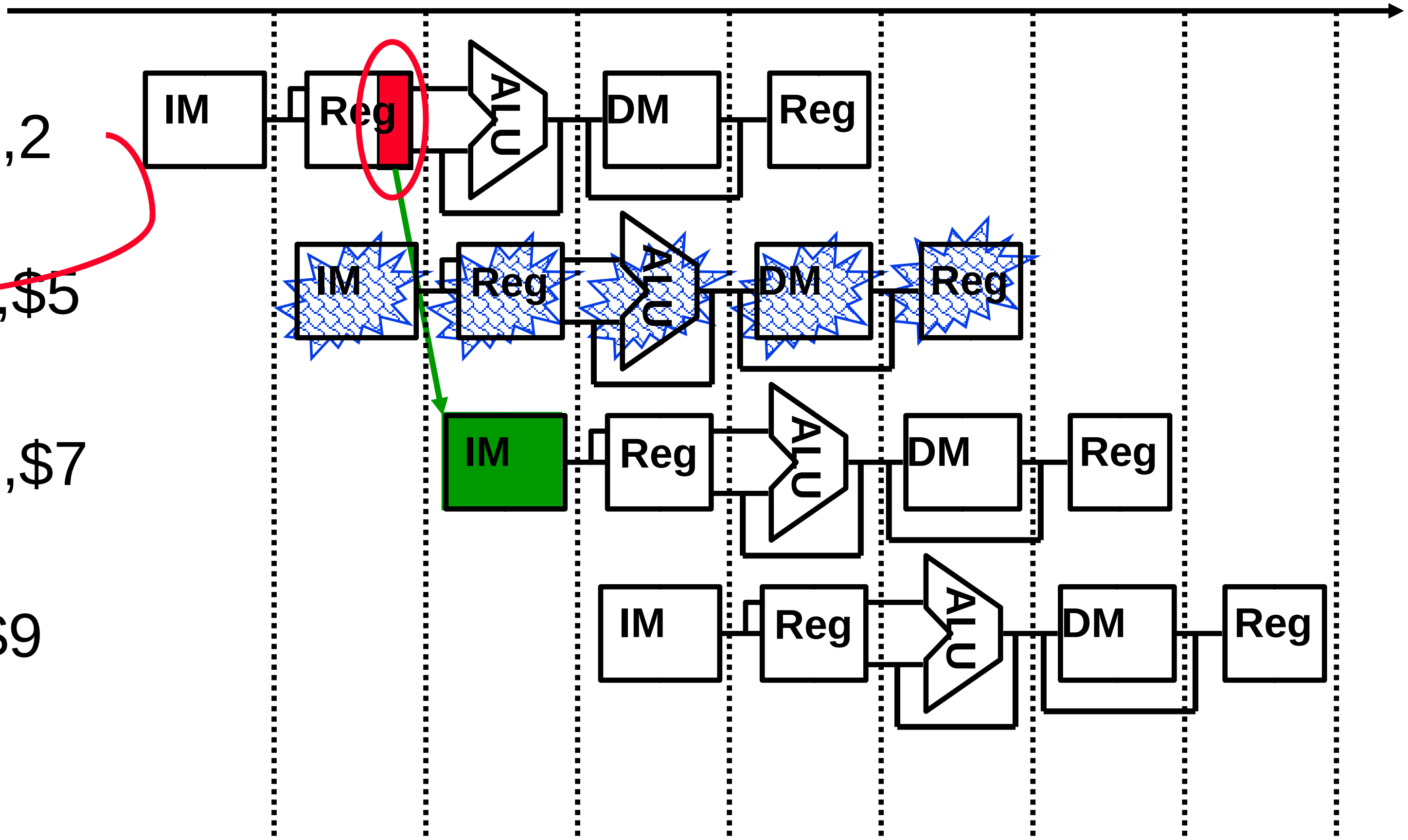
4 beq $1,$2,2

8 sub $4,$1,$5

flush

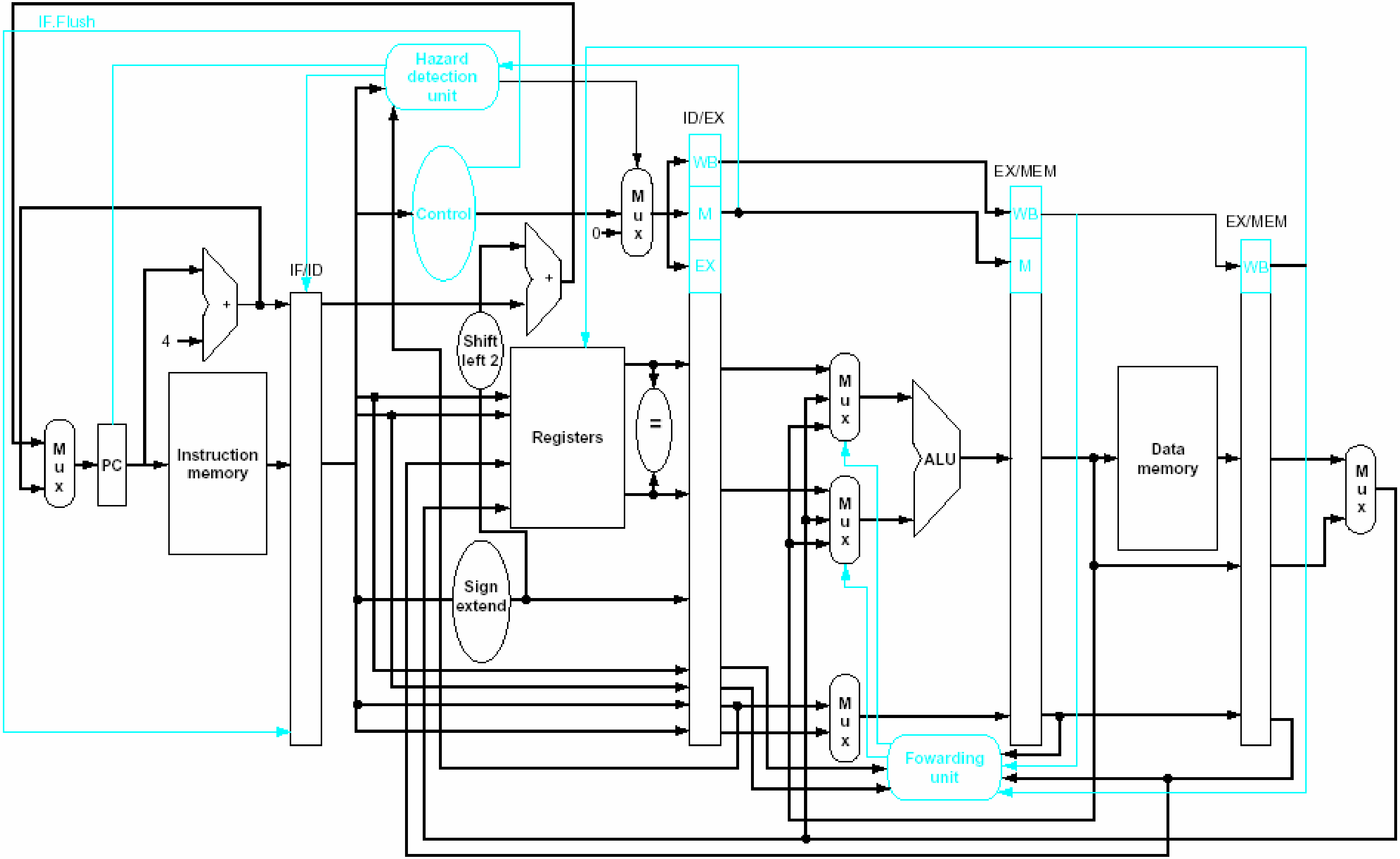
16 and $6,$1,$7

20 or r8,$1,$9



- To flush the IF stage instruction, add a IF.Flush control line that zeros the instruction field of the IF/ID pipeline register (transforming it into a noop)

Flushing Instructions (Fig. 6.41)



Branch Prediction, con’t

- Resolve branch hazards by statically assuming a given outcome and proceeding

2. Predict taken – always predict branches will be taken
   - Predict taken always incurs a stall (if branch destination hardware has been moved to the ID stage)

- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance

- With more hardware, possible to try to predict branch behavior dynamically during program execution

3. 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 a bit that tells whether the branch was taken the last time it was execute
  - Bit may predict incorrectly (may be from a different branch with the same low order PC bits, or may be a wrong prediction for this branch) but the doesn't affect correctness, just performance
  - If the prediction is wrong, flush the incorrect instructions in pipeline, restart the pipeline with the right instructions, and invert the prediction bit
- 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 can cache the branch target address (or "even" the branch target instruction) so that a stall can be avoided

1-bit Prediction Accuracy

- 1-bit predictor in loop is 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

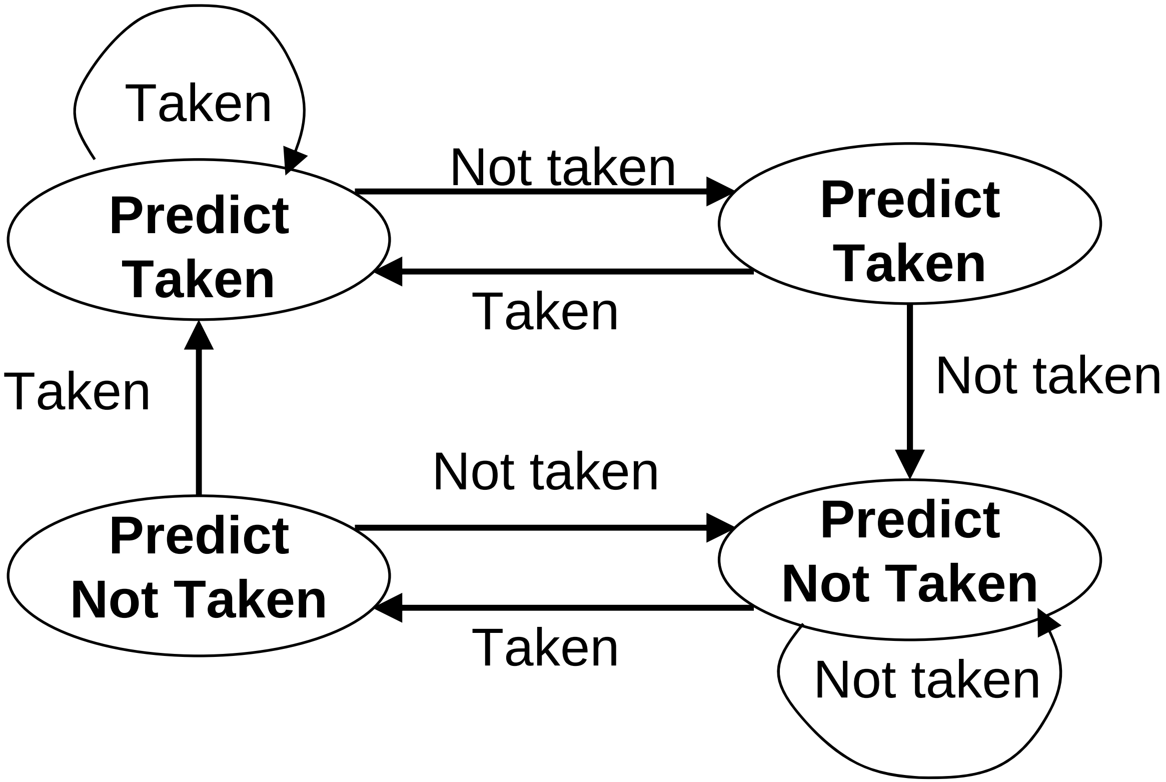
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  2. As long as branch is taken (looping), prediction is correct
  3. Exiting the loop, the predictor again mispredicts the branch since 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.**

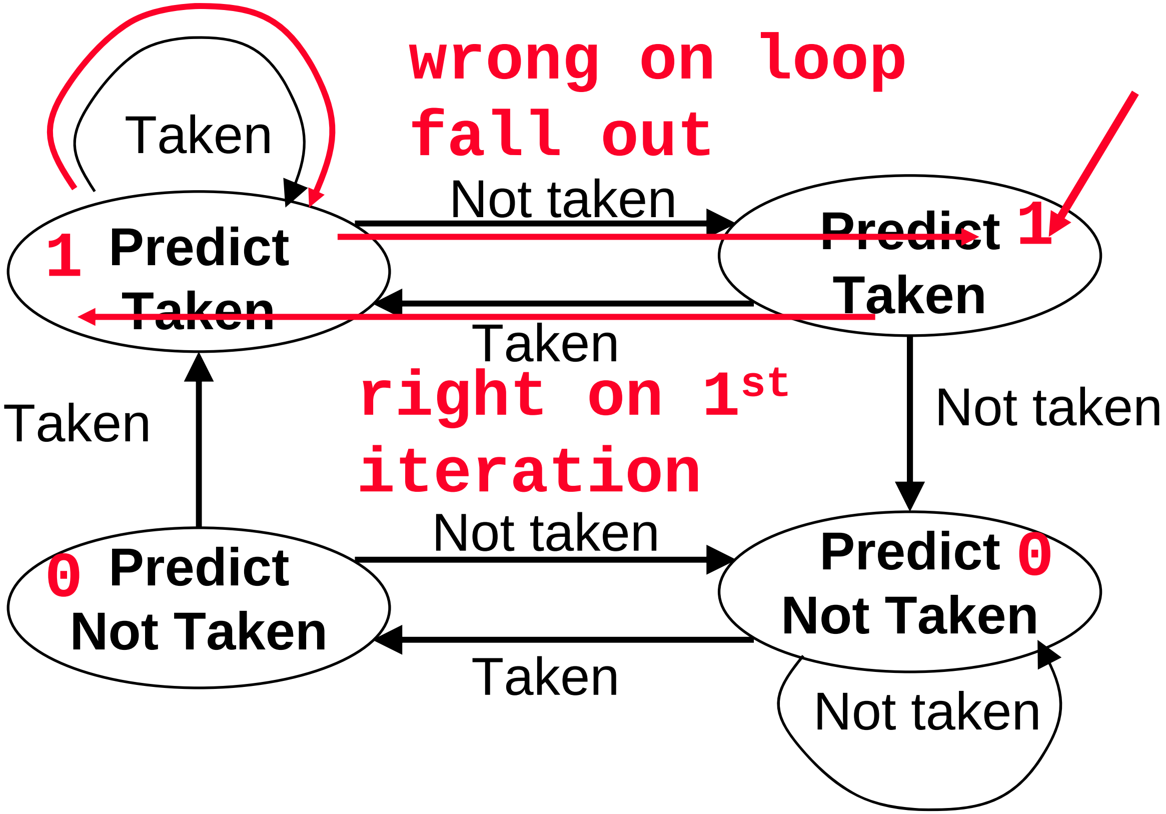


```
Loop: 1st loop instr
      2nd loop instr
           .
           .
           .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

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



wrong on loop fall out

right on 1st iteration

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

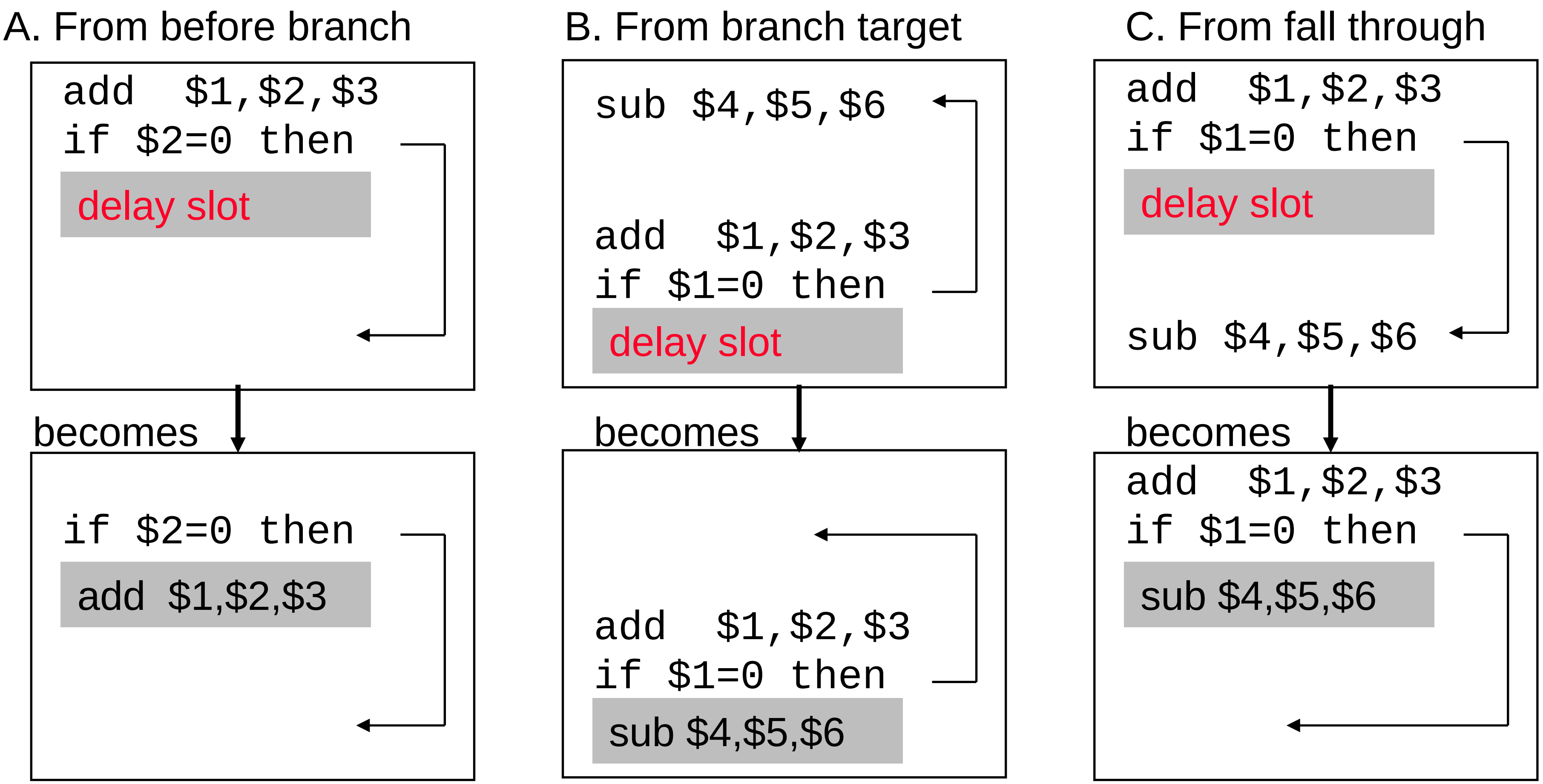
Delayed Decision

- First, move the branch decision hardware and target address calculation to the ID pipeline stage

- A delayed branch always executes the next sequential instruction – the branch takes effect after that next instruction
  - MIPS software moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay

- As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Scheduling Branch Delay Slots



A. From before branch

```
add  $1,$2,$3
if $2=0 then
```

delay slot

becomes

```
if $2=0 then
```

add $1,$2,$3

B. From branch target

```
sub $4,$5,$6

add  $1,$2,$3
if $1=0 then
```

delay slot

becomes

```
add  $1,$2,$3
if $1=0 then
```

sub $4,$5,$6

C. From fall through

```
add  $1,$2,$3
if $1=0 then
```

delay slot

```
sub $4,$5,$6
```

becomes

```
add  $1,$2,$3
if $1=0 then
```

sub $4,$5,$6

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

When is pipelining hard?

- Interrupts: 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restart?
  - Who caused the interrupt?

Stage	*Problem interrupts occurring*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation

- Load with data page fault, Add with instruction page fault?
- *Solution 1*: interrupt vector/instruction, check last stage
- Solution 2: interrupt ASAP, restart everything incomplete

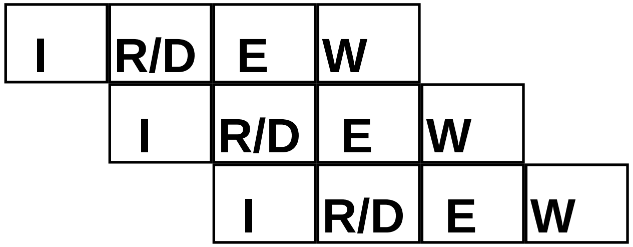
When is pipelining hard?

- Complex Addressing Modes and Instructions
- Address modes: Autoincrement causes register change during instruction execution
  - Interrupts?
  - Now worry about write hazards since write no longer last stage
    - ⇒Write After Read (WAR): Write occurs before independent read
    - ⇒Write After Write (WAW): Writes occur in wrong order, leaving wrong result in registers
    - ⇒(Previous data hazard called RAW, for Read After Write)
- Memory-memory Move instructions
  - Multiple page faults
  - make progress?

Avoiding Data Hazards by Design

- Suppose instructions are executed in a pipelined fashion such that Instructions are initiated in order.
- *WAW avoidance:* if writes to a particular resource (e.g., reg) are performed in the same stage for all instructions, then no WAW hazards occur.

  proof: writes are in the same time sequence as instructions.



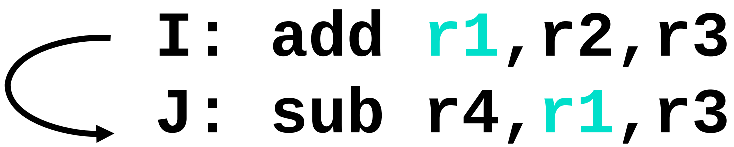
I	R/D	E	W		
	I	R/D	E	W	
		I	R/D	E	W

- *WAR avoidance*: if in all instructions reads of a resource occur at an earlier stage than writes to that resource occur in any instruction, then no WAR hazards occur.

  proof: A successor instruction must issue later, hence it will perform writes only after all reads for the current instruction.

Generic Data Hazards: RAW, WAR, WAW

- **Read After Write (RAW)**
  Instr$_J$ tries to read operand before Instr$_I$ writes it



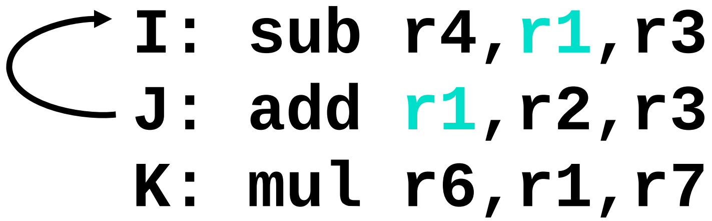
```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- **Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.**

- **Forwarding handles many, but not all, RAW dependencies in 5 stage MIPS pipeline**

Generic Data Hazards: RAW, WAR, WAW

- **Write After Read (WAR)**
  Instr$_J$ writes operand *before* Instr$_I$ reads it



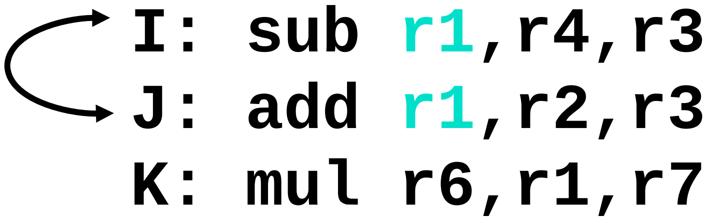
```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- **Called an “anti-dependence” by compiler writers.**
  This results from “reuse” of the name “r1”.

- **Can’t happen in MIPS 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Reads are always in stage 2, and**
  - **Register Writes must be in stage 5**

Generic Data Hazards: RAW, WAR, WAW

- **Write After Write (WAW)**
  Instr$_J$ writes operand *before* Instr$_I$ writes it.



```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- **Called an "output dependence" by compiler writers**
  This also results from the "reuse" of name "r1".

- **Can't happen in MIPS 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Register Writes must be in stage 5**

- **Can see WAR and WAW in more complicated pipes**

When is pipelining hard?

- Floating Point: long execution time
- Also, may pipeline FP execution unit so that can initiate new instructions without waiting full latency

FP Instruction	*Latency*	*Initiation Rate*	*(MIPS R4000)*
Add, Subtract	4	3	
Multiply	8	4	
Divide	36	35	
Square root	112	111	
Negate	2	1	
Absolute value	2	1	
FP compare	3	2	

- Divide, Square Root take -10X to -30X longer than Add
  - Exceptions?
  - Adds WAR and WAW hazards since pipelines are no longer same length

First Generation RISC Pipelines

- All instructions follow same pipeline order (“static schedule”).
- Register write in last stage
  - Avoid WAW hazards
- All register reads performed in first stage after issue.
  - Avoid WAR hazards
- Memory access in stage 4
  - Avoid all memory hazards
- Control hazards resolved by delayed branch (with fast path)
- RAW hazards resolved by bypass, except on load results which are resolved by fiat (delayed load).

Substantial pipelining with very little cost or complexity.

Machine organization is (slightly) exposed!

Relies very heavily on "hit assumption" of memory accesses in cache

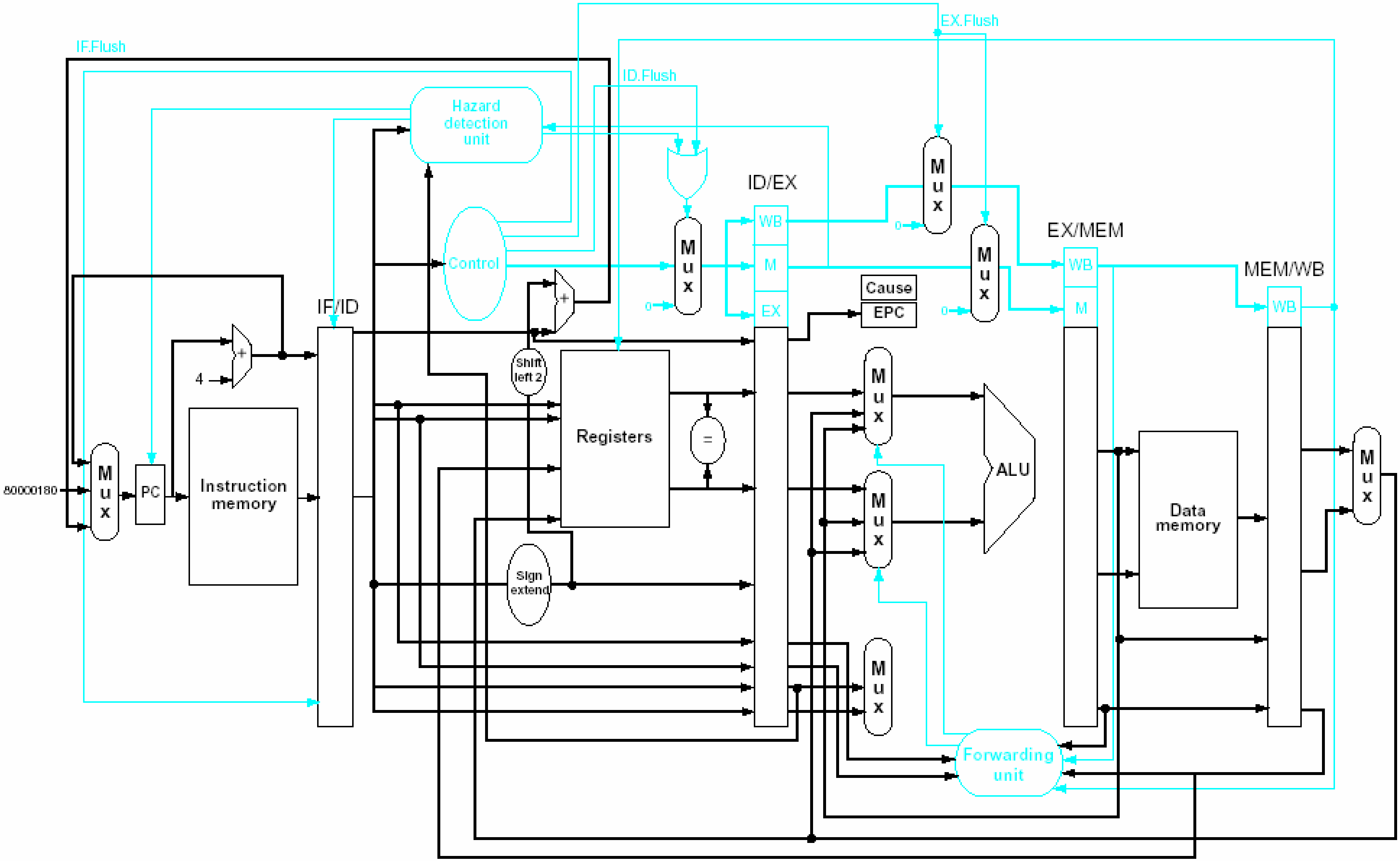
Review: Summary of Pipelining Basics

- Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

- Hazards limit performance on computers:
  - structural: need more HW resources
  - data: need forwarding, compiler scheduling
  - control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards since pipelining helps instruction bandwidth, not latency
- Compilers key to reducing cost of data and control hazards
  - load delay slots
  - branch delay slots
- Exceptions, Instruction Set, FP makes pipelining harder
- Longer pipelines => Branch prediction, more instruction parallelism?

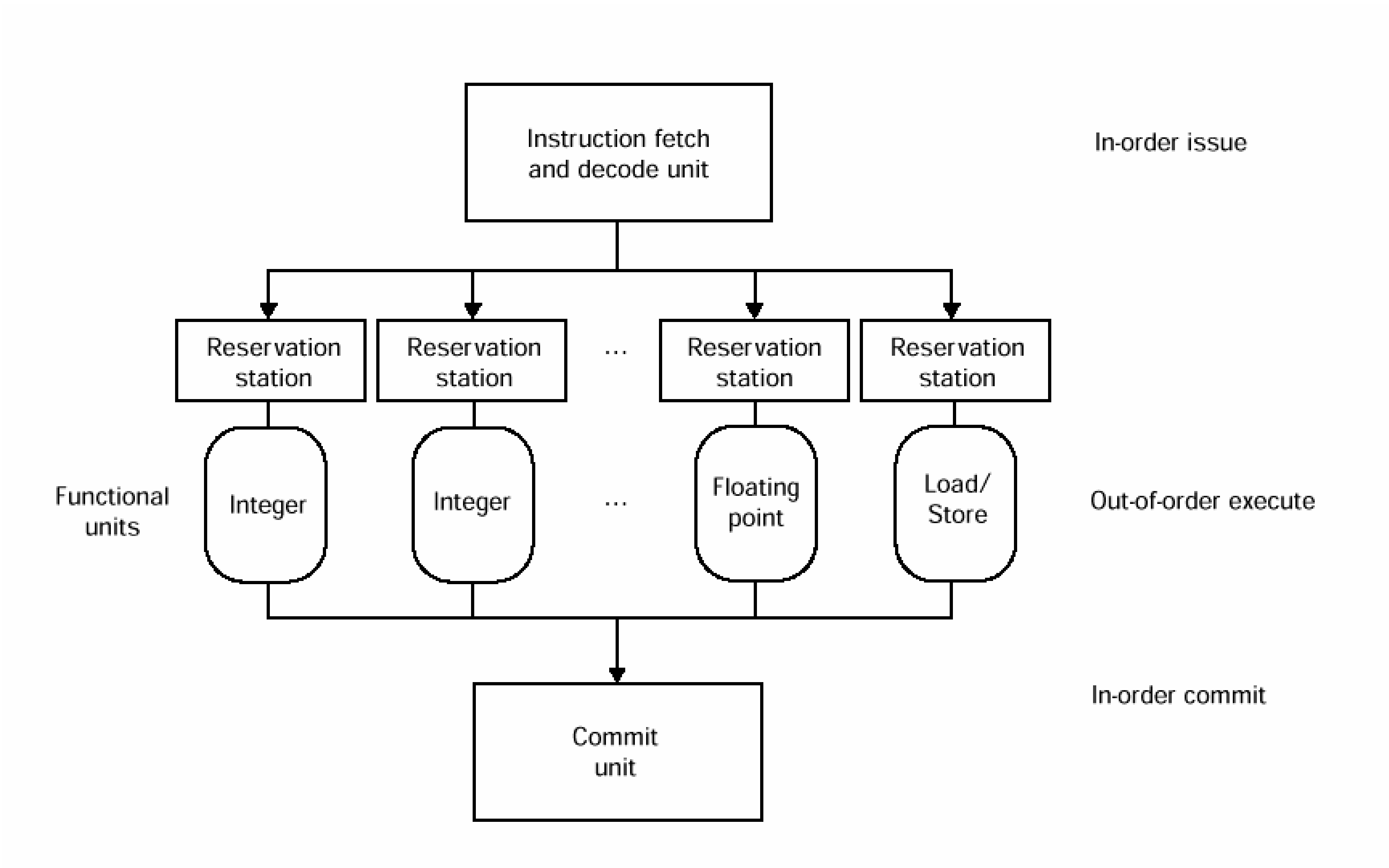
Datapath with Controls to Handle Exceptions (Fig. 6.42)



Dynamic Scheduling

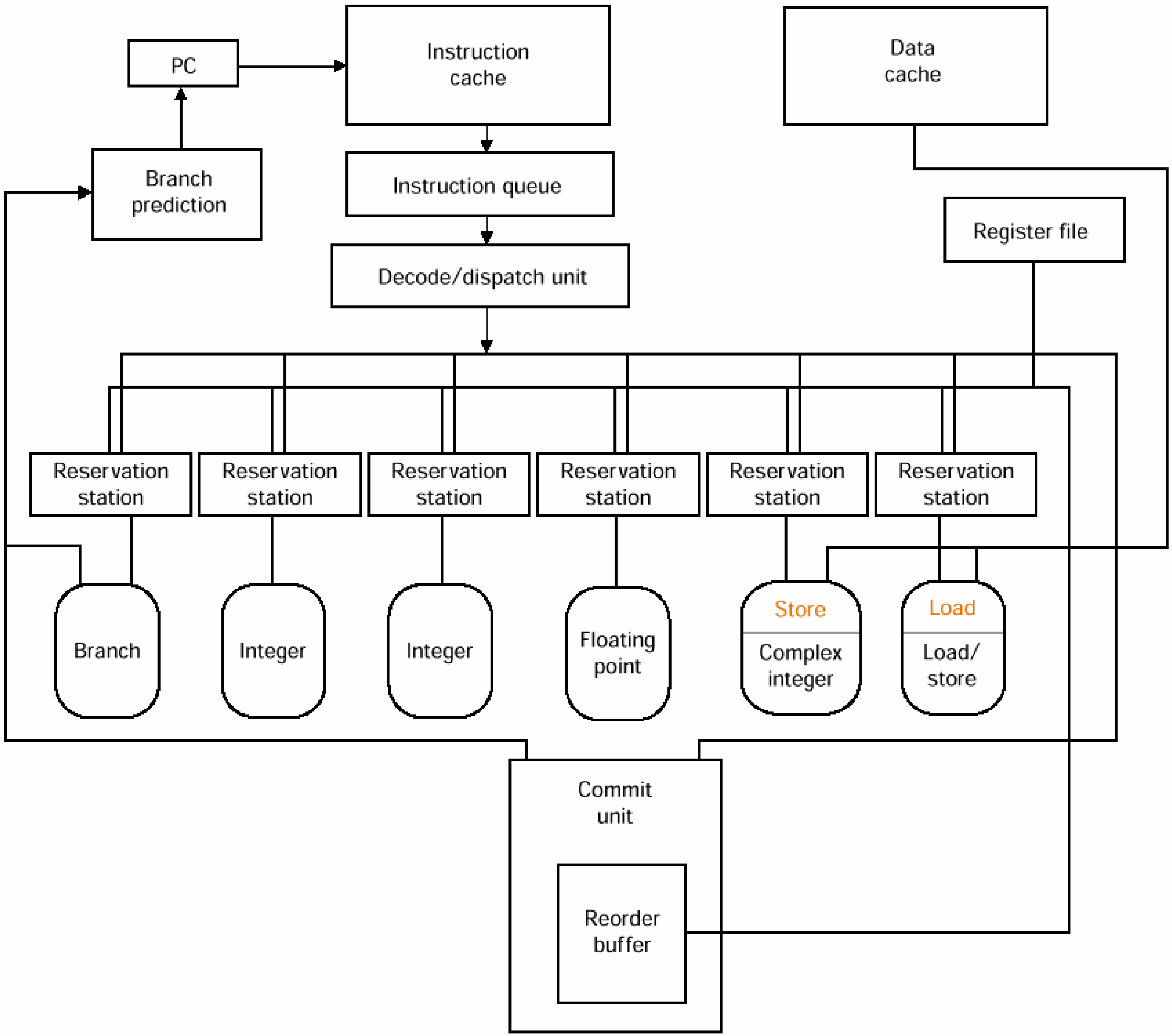
- The hardware performs the “scheduling”
  - hardware tries to find instructions to execute
  - out of order execution is possible
  - speculative execution and dynamic branch prediction
- All modern processors are very complicated
  - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
  - PowerPC and Pentium: branch history table
  - Compiler technology important
- This class has given you the background you need to learn more

Primary Units of a Dynamic Pipeline Scheduling



© MKP 2004

PowerPC 604 and Pentium Pro Pipelines


PC
Instruction cache
Data cache
Branch prediction
Instruction queue
Register file
Decode/dispatch unit
Reservation station
Reservation station
Reservation station
Reservation station
Reservation station
Reservation station
Branch
Integer
Integer
Floating point
Store
Complex integer
Load
Load/ store
Commit unit
Reorder buffer