The Pipelined CPU

Lecture for CPSC 5155

Edward Bosworth, Ph.D.
Computer Science Department
Columbus State University

Revised 9/22/2013

Pipelining: A Faster Approach

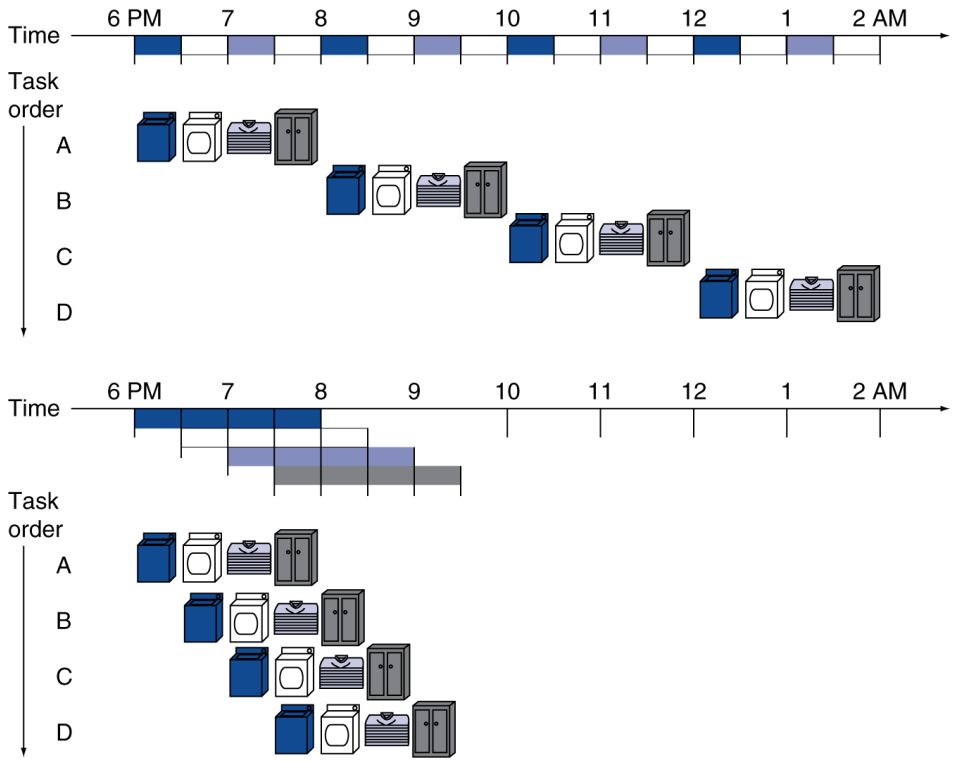
- There are two types of simple control unit design:

1. The single–cycle CPU with its slow clock, which executes one instruction per clock pulse.
2. The multi–cycle CPU with its faster clock. This divides the execution of an instruction into 3, 4, or 5 phases, but takes that number of clock pulses to execute a single instruction.

- We now move to the more sophisticated CPU design that allows the **apparent** execution of one instruction per clock cycle, even with the faster clock.
- This design technique is called **pipelining**, though it might better be considered as an assembly line.

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

The Ford Assembly Line in 1913

Mr. Ford's Idea

- Henry Ford began working on the assembly line concept about 1908 and had essentially perfected the idea by 1913. His motivations are worth study.
- In previous years, automobile manufacture was done by highly skilled technicians, each of whom assembled the whole car.
- It occurred to Mr. Ford that he could get more get more workers if he did not require such a high skill level. One way to do this was to have each worker perform only a few tasks.
- It soon became obvious that is was easier to bring the automobile to the worker than have the worker (and his tools) move to the automobile. The assembly line was born.

Productivity in an Assembly Line

- We may ask about how to measure the productivity of an assembly line.
- How long does it take for any individual car to be fabricated, from basic parts to finish? This might be a day or so.
- How many cars are produced per hour? This might be measured in the 100's.
- The key measure is the rate of production.

The Pipelined CPU

- The CPU pipeline is similar to an assembly line.

1. The execution of an instruction is broken into a number of simple steps, each of which can be handled by an efficient execution unit.
2. The CPU is designed so that it can execute a number of instructions simultaneously, each in its own distinct phase of execution.
3. The important number is the number of instructions completed per unit time, or equivalently the **instruction issue rate**.

A Constraint on Pipelining

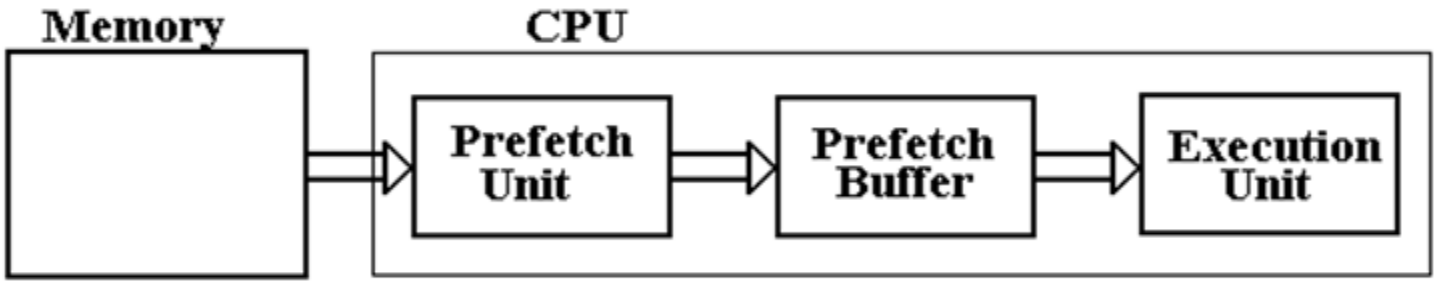
- The effect of an instruction sequence cannot be changed by the pipeline.
- Consider the following code fragment.
- **add $s0, $t0, $t1 # $s0 = $t0 + $t1**
- **sub $t2, $s0, $t3 # $t2 = $s0 - $t3**
- This does not have the same effect as it would if reordered.
- **sub $t2, $s0, $t3 # $t2 = $s0 - $t3**
- **add $s0, $t0, $t1 # $s0 = $t0 + $t1**

The Earliest Pipelines

- The first problem to be attacked in the development of pipelined architectures was the fetch–execute cycle. The instruction is fetched and then executed.
- How about fetching one instruction while a previous instruction is executing?
- It is here that we see one of the advantages of RISC designs, such as the MIPS. Each instruction has the same length (32 bits) as any other instruction, so that an instruction can be pre-fetched without taking time to identify it.

Instruction Pre-Fetch

- Break up fetch–execute and do the two in parallel. This dates to the IBM Stretch (1959).



- The prefetch buffer is implemented in the CPU with on–chip registers. Logically, it is a queue. The CDC–6600 buffer had a queue of length 8.

Flushing the Instruction Queue

- The CDC-6600 was word addressable.
- Suppose that an instruction at address X is executing. The instructions at addresses (X+1) through (X+7) are in the queue.
- Suppose that the instruction at address X causes a jump to address (X + 100).
- The instructions already in the queue are “stale” and will not be used. Flush the queue.

MIPS Pipeline

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

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

Pipeline Performance

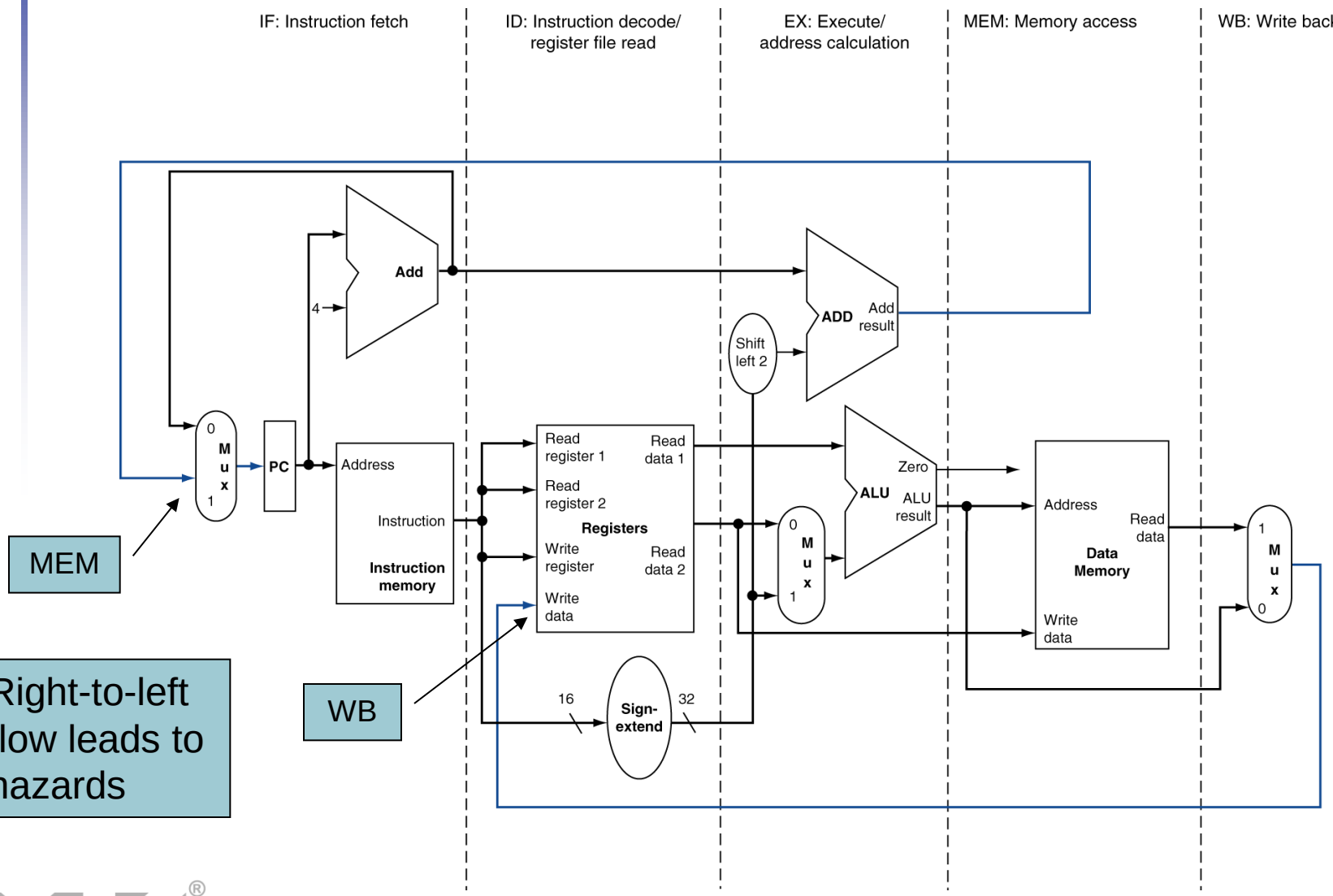
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

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

A Pipeline Needs Registers

- Suppose a CPU with five independent instructions in different stages of execution.
- Each instruction needs to execute within its own context, and have its own state (set of registers)
- Fortunately, Moore's law has allowed us to place increasingly larger numbers of transistors (hence registers) in a CPU.

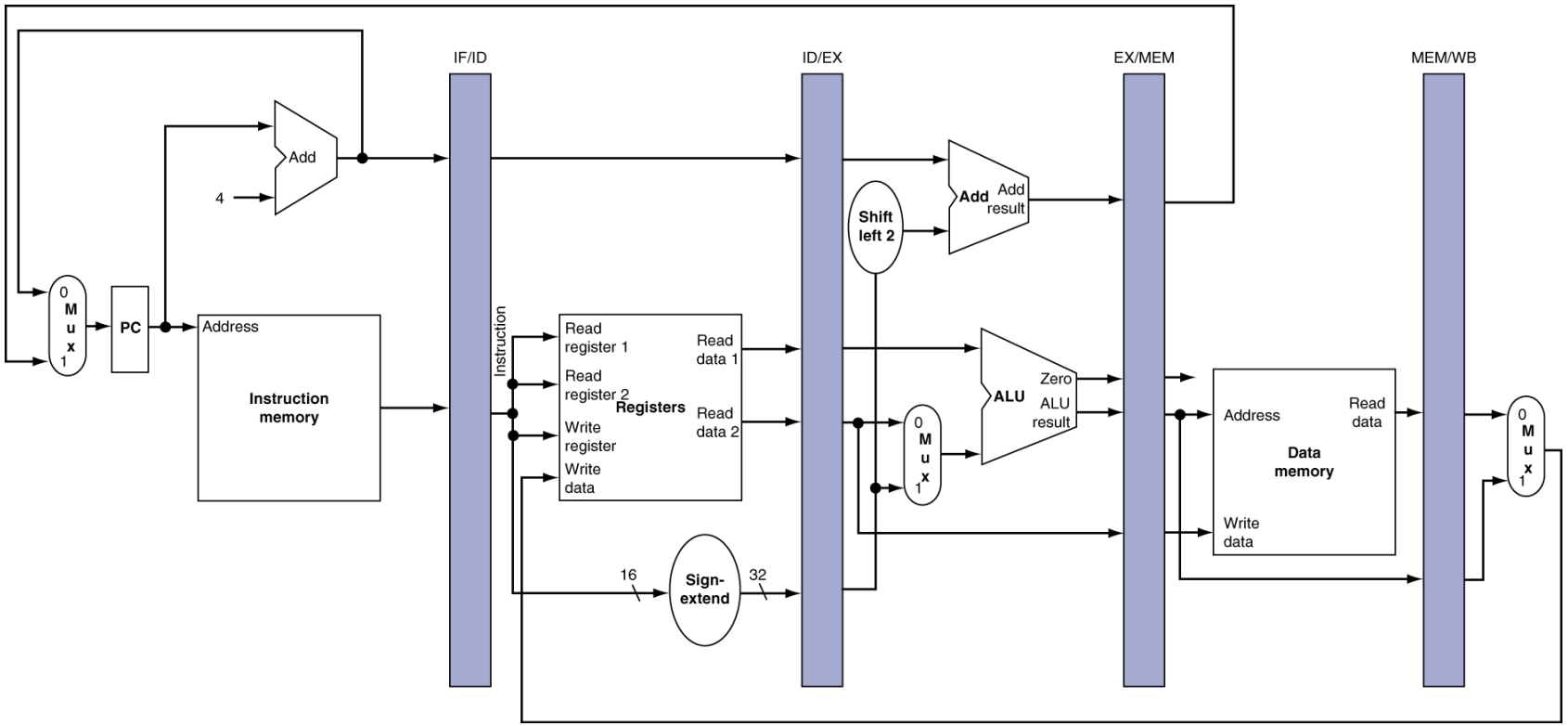
MIPS Pipelined Datapath



Right-to-left flow leads to hazards

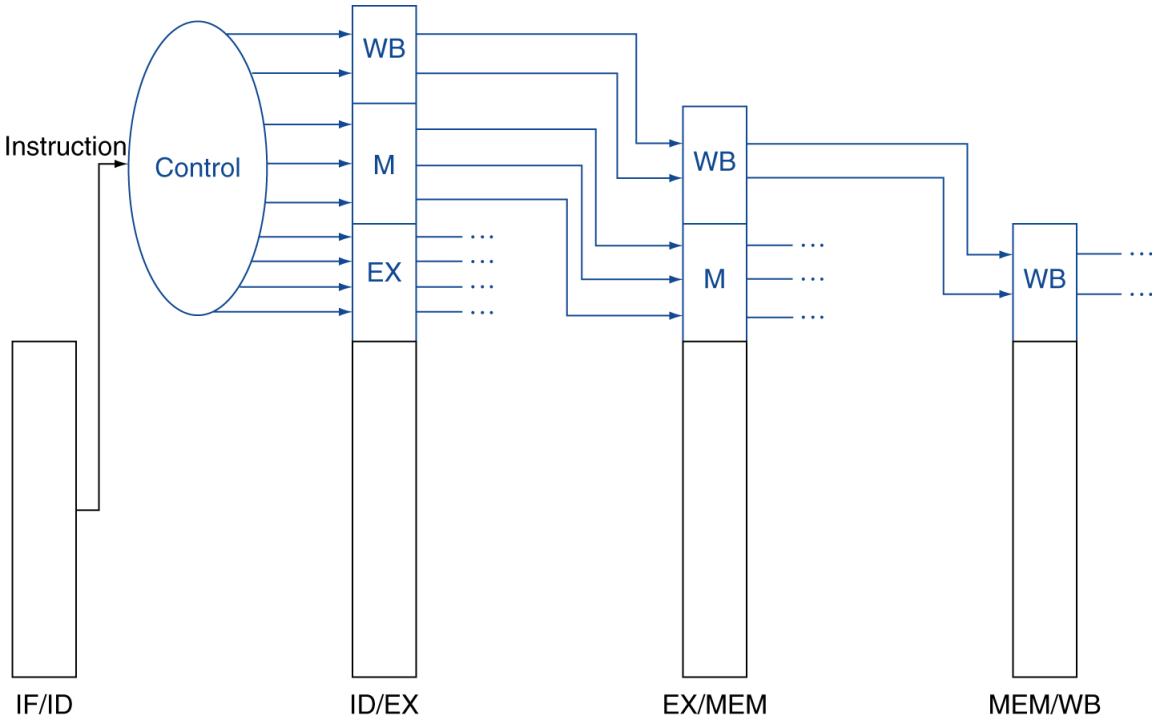
Pipeline registers

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

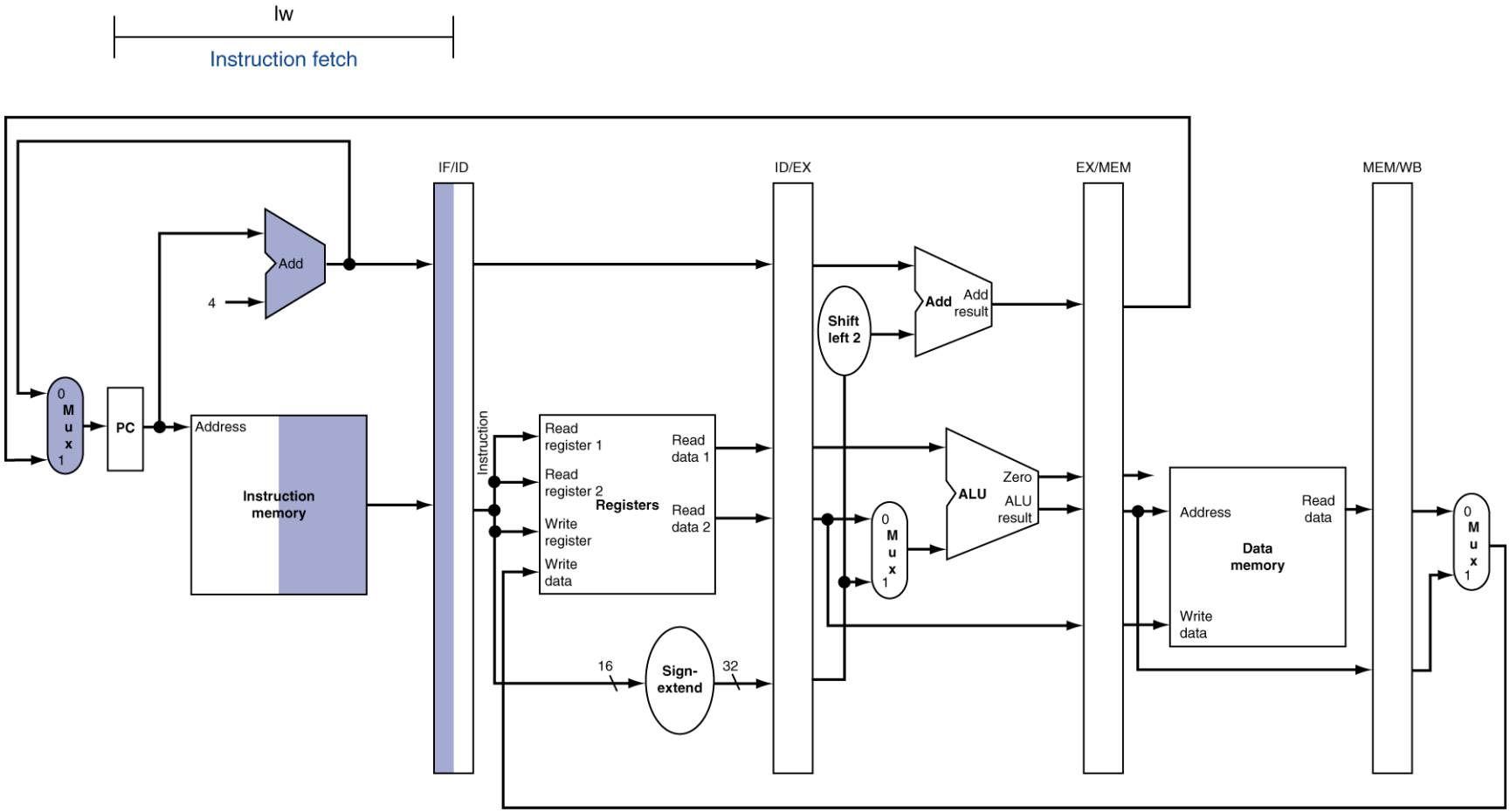


Pipelined Control

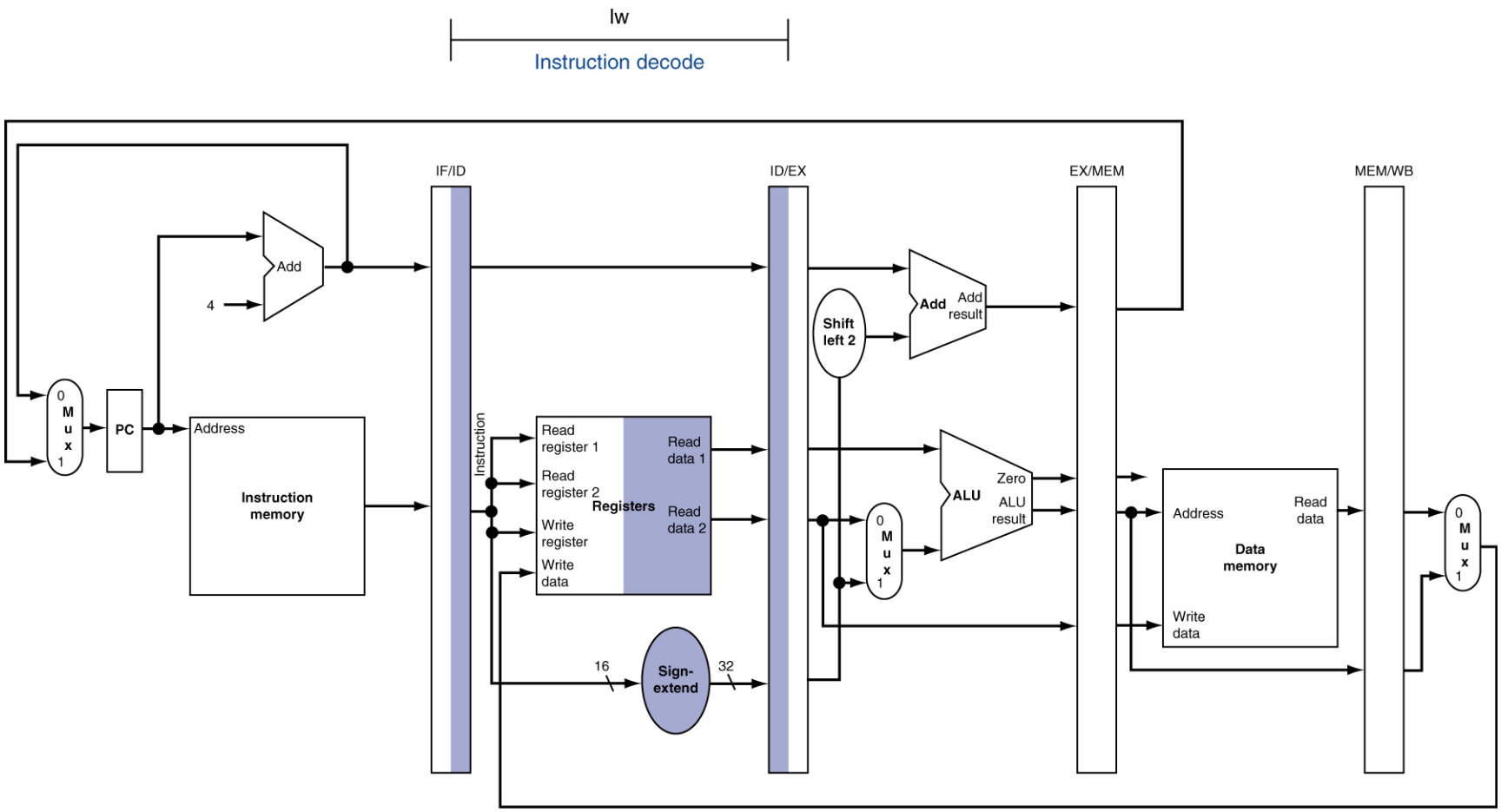
- Control signals derived from instruction
  - As in single-cycle implementation



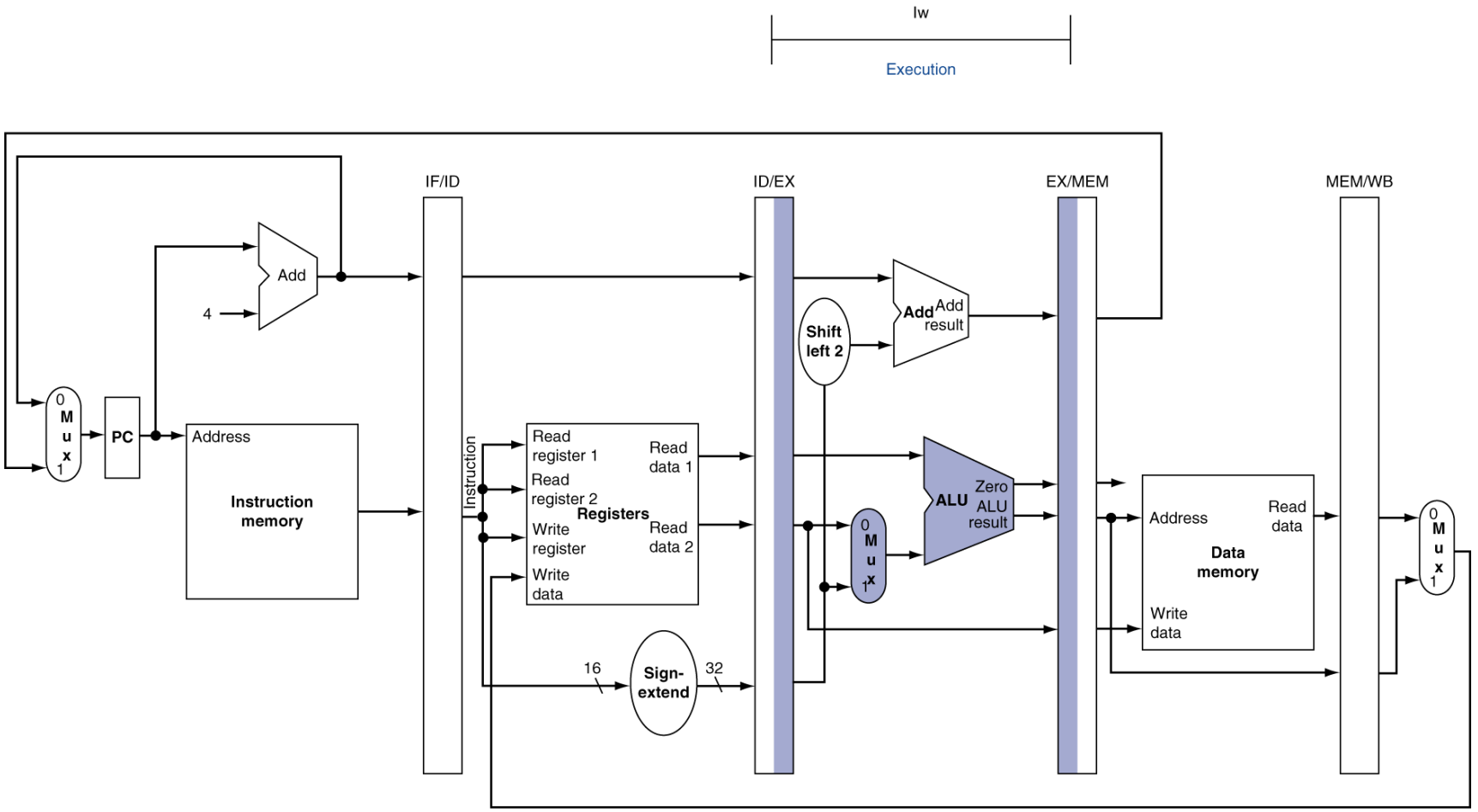
IF for Load, Store, …



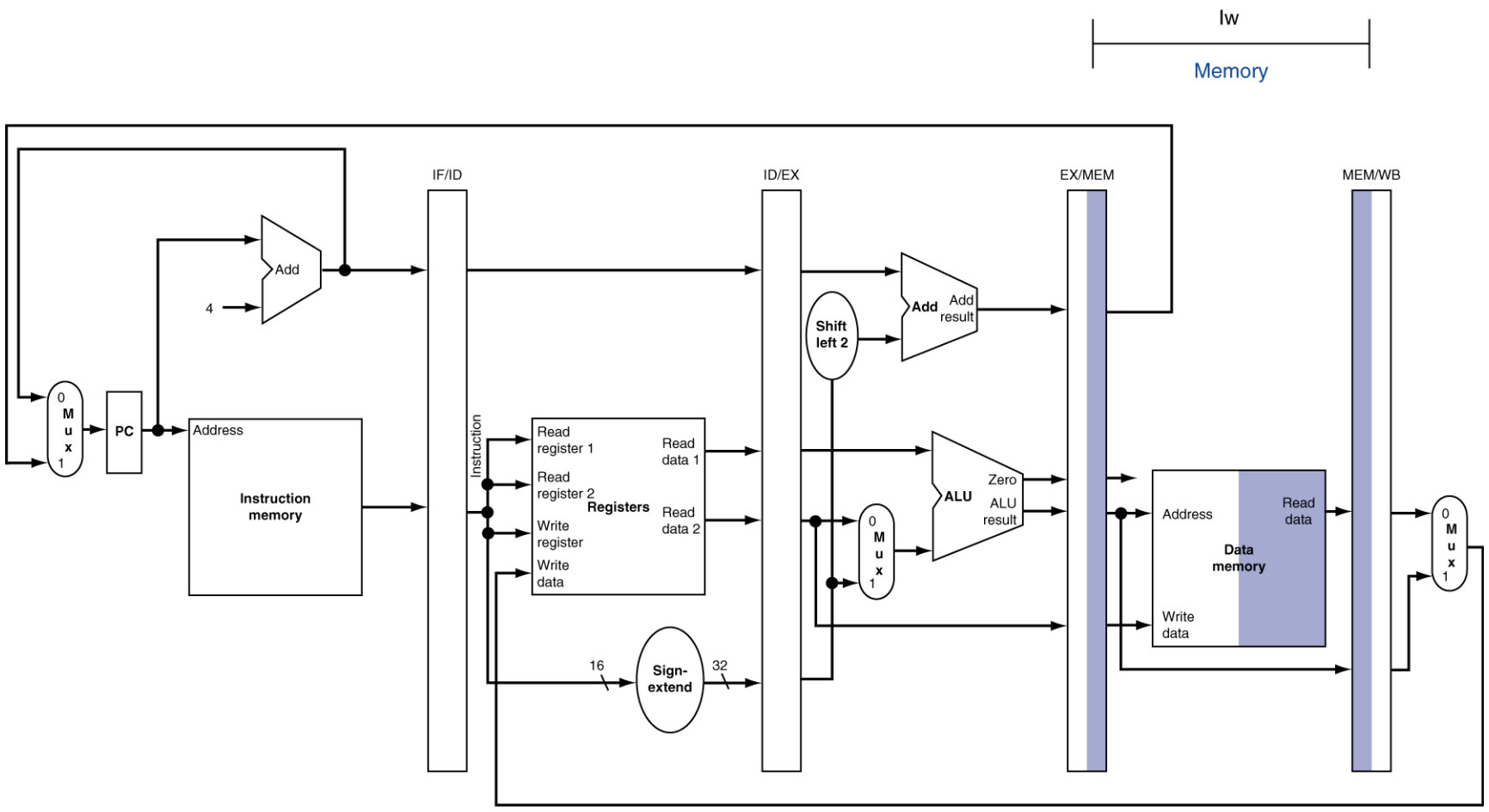
ID for Load, Store, …



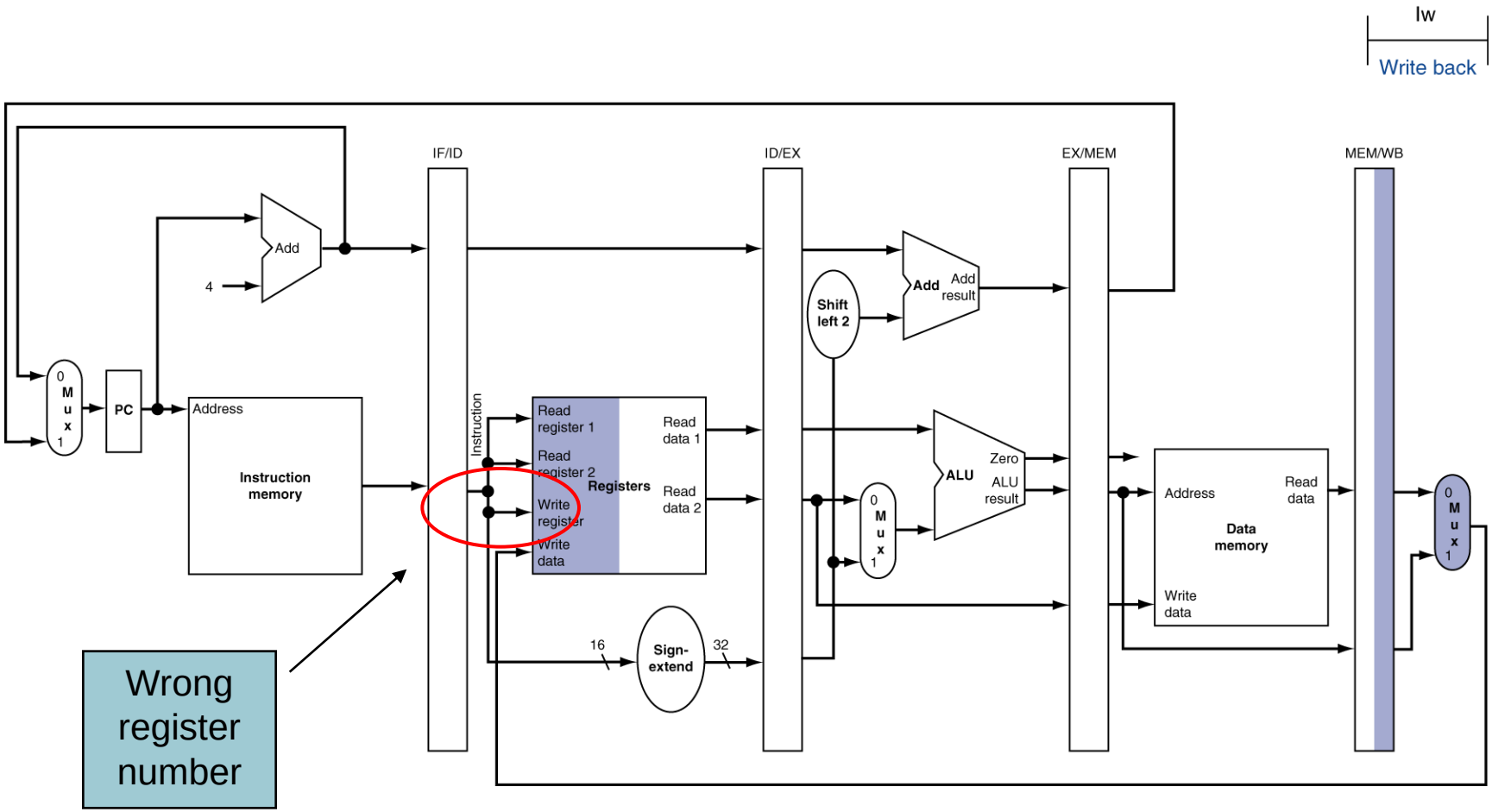
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

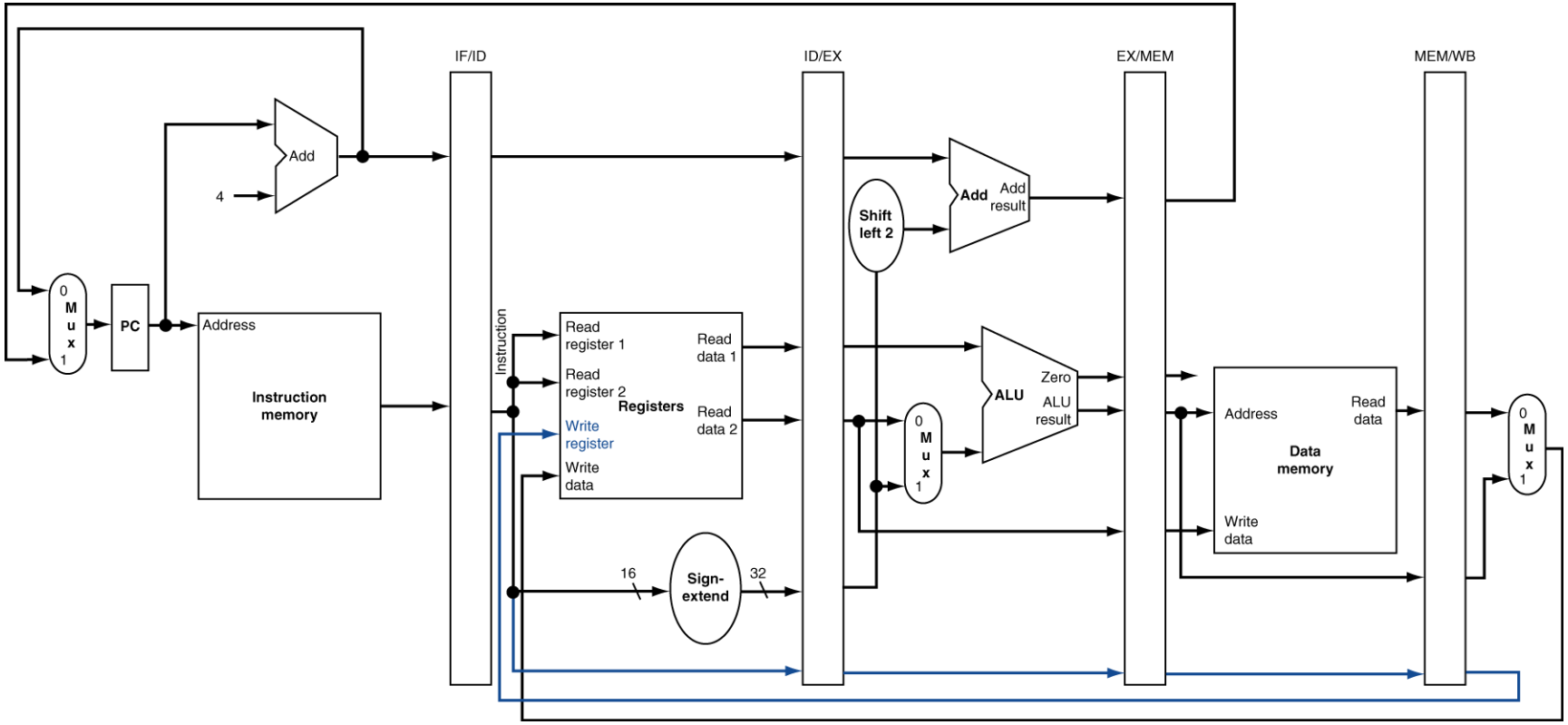
MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

WB for Load


lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32
Wrong register number

Corrected Datapath for Load



Pipelines and Heat

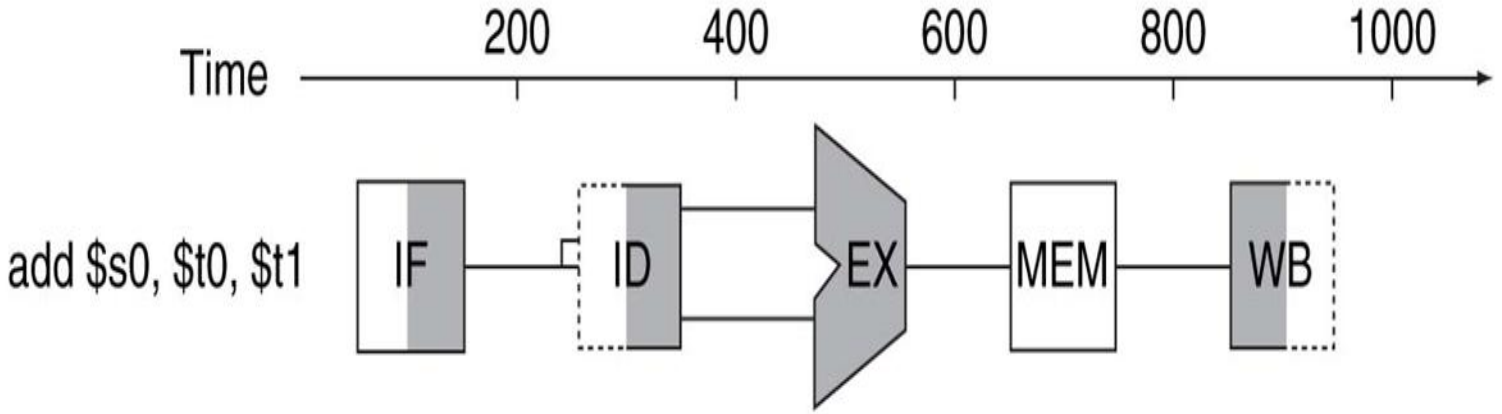
- The number of stages in a pipeline is often called the **“depth”** of the pipeline.
- The MIPS design has depth = 5.
- The greater the depth of the pipeline,

➢ the faster the CPU, and

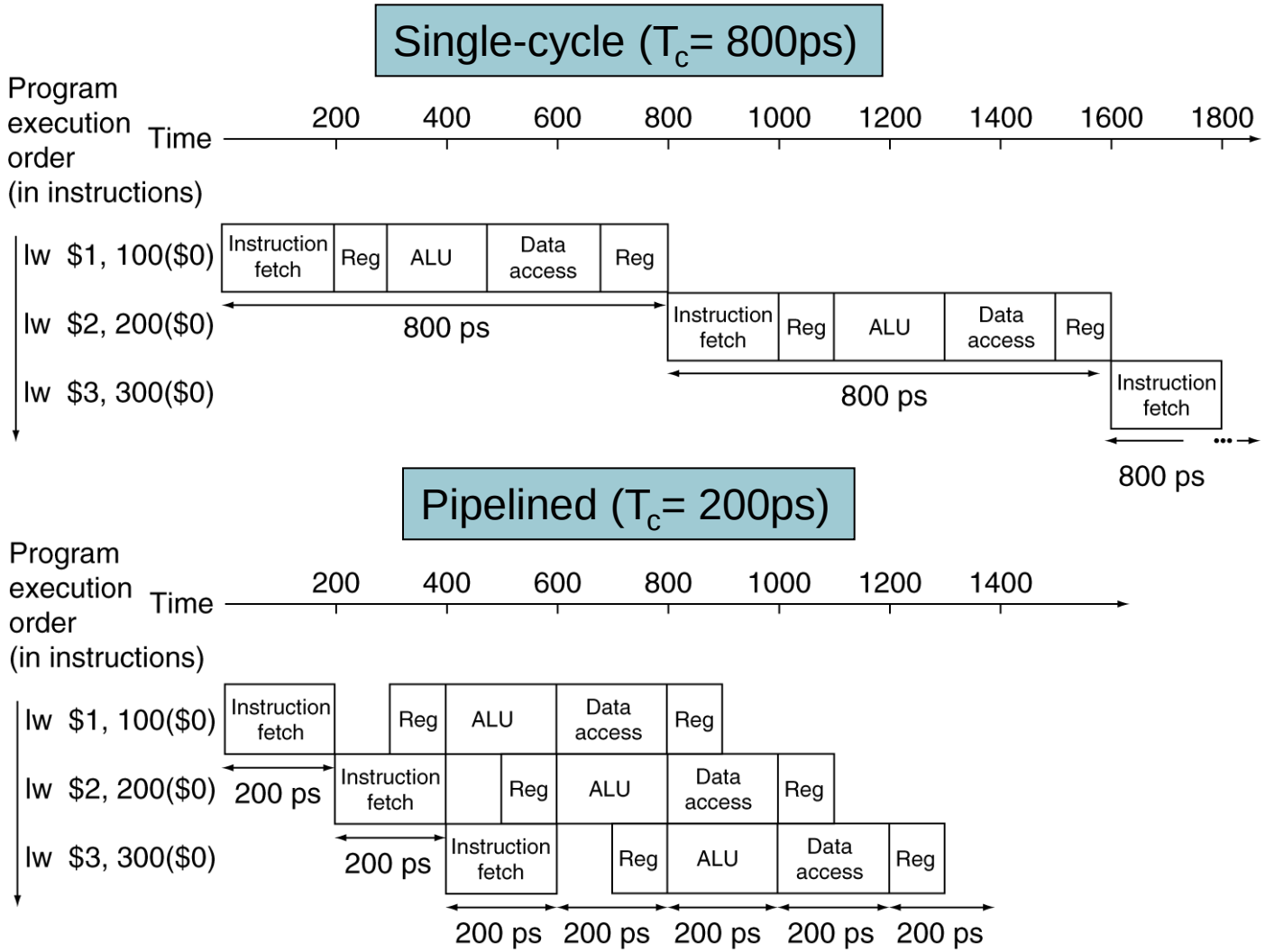
➢ the more heat the CPU generates, as it has more registers and circuits to support the pipeline.

Graphical Depiction of the Pipeline

- The data elements in IF and ID are read during the second half of the clock cycle.
- The data elements in WB are written during the first half of the clock cycle.



Pipeline Performance



Pipeline Speedup

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

Pipelining and ISA Design

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

The Ideal Pipeline

- Each instruction is **issued** and enters the pipeline. This step is IF (Instruction Fetch)
- As it progresses through the pipeline it does not depend on the results of any other instruction now in the pipeline.
- The speedup of an N-stage pipeline under these conditions is about N.
- The older **vector machines**, such as the Cray, structured code that would meet this ideal.

Pipeline Realities

1. It is not possible for any instruction to depend on the results of instructions that will execute in the future. This is a logical impossibility.
2. There are no issues associated with dependence on instructions that have completed execution and exited the pipeline.
3. Hazards occur due to instructions that have started execution, but are still in the pipeline.
4. It is possible, and practical, to design a compiler that will minimize hazards in the pipeline. This is a desirable result of the joint design of compile and ISA.
5. It is not possible for the compiler to eliminate **all** pipelining problems without reducing the CPU to a non–pipelined datapath, which is unacceptably slow.

One Simplistic Solution

- As we shall see in a moment, the following code causes a data hazard in the pipeline.

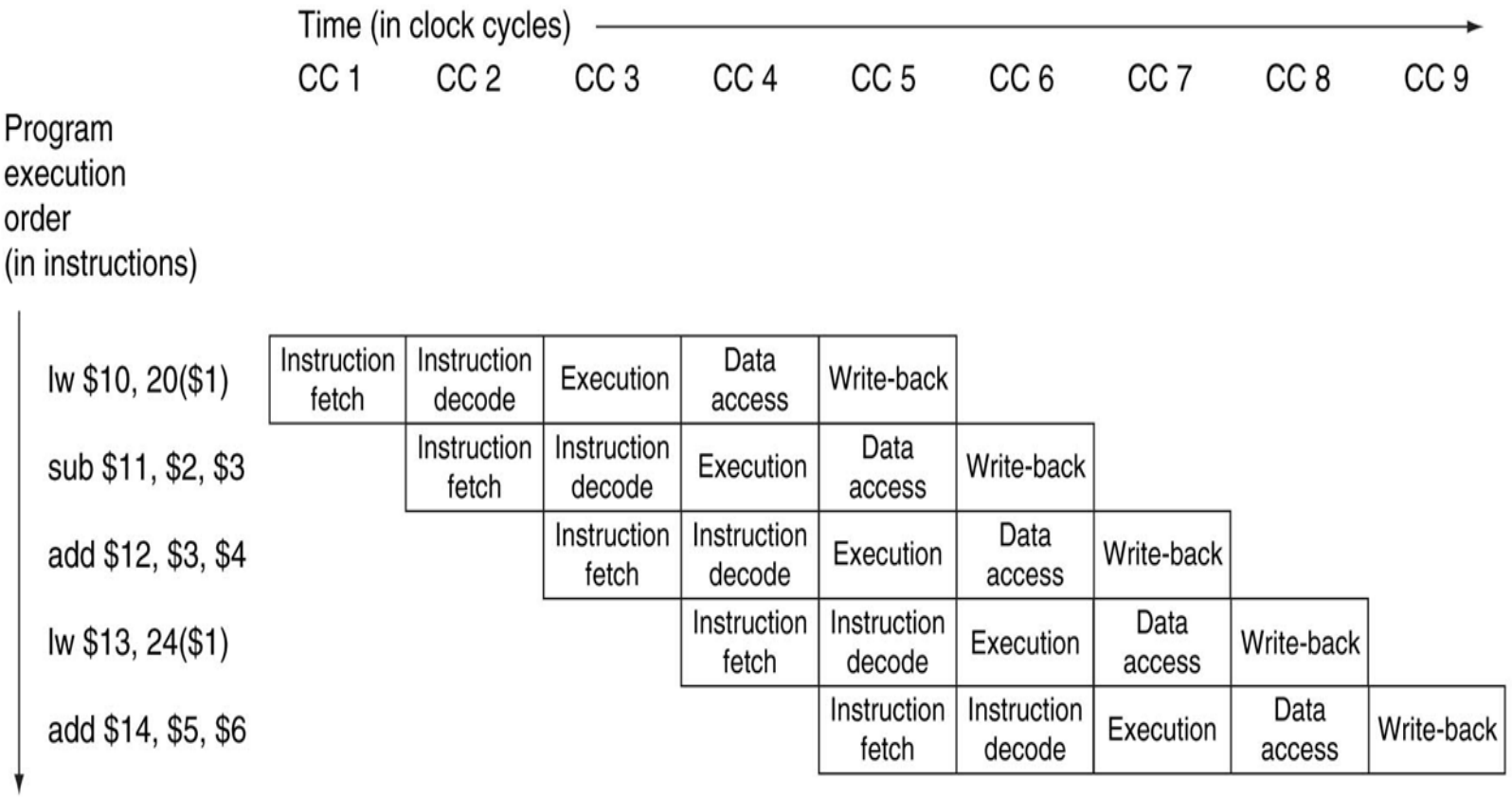
```
add     $s0, $t0, $t1
sub     $t2, $s0, $t3
```

- Here is a simplistic way to avoid the hazard.

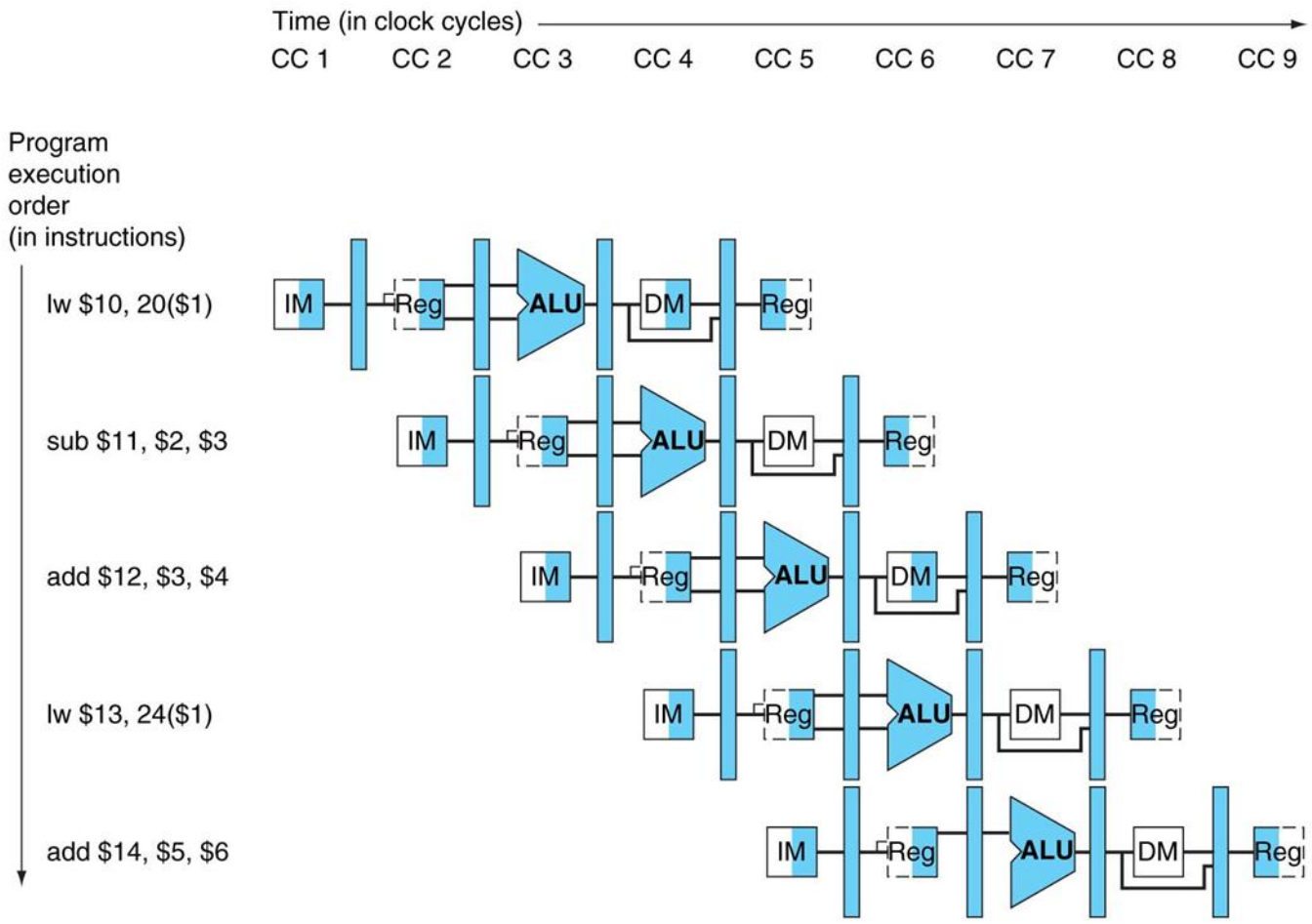
```
add     $s0, $t0, $t1
nop     # Do nothing for two
nop     # clock cycles.
sub     $t2, $s0, $t3
```

One View of the Pipeline Process

This is the ideal case with no stalls.



Another View of the Pipeline

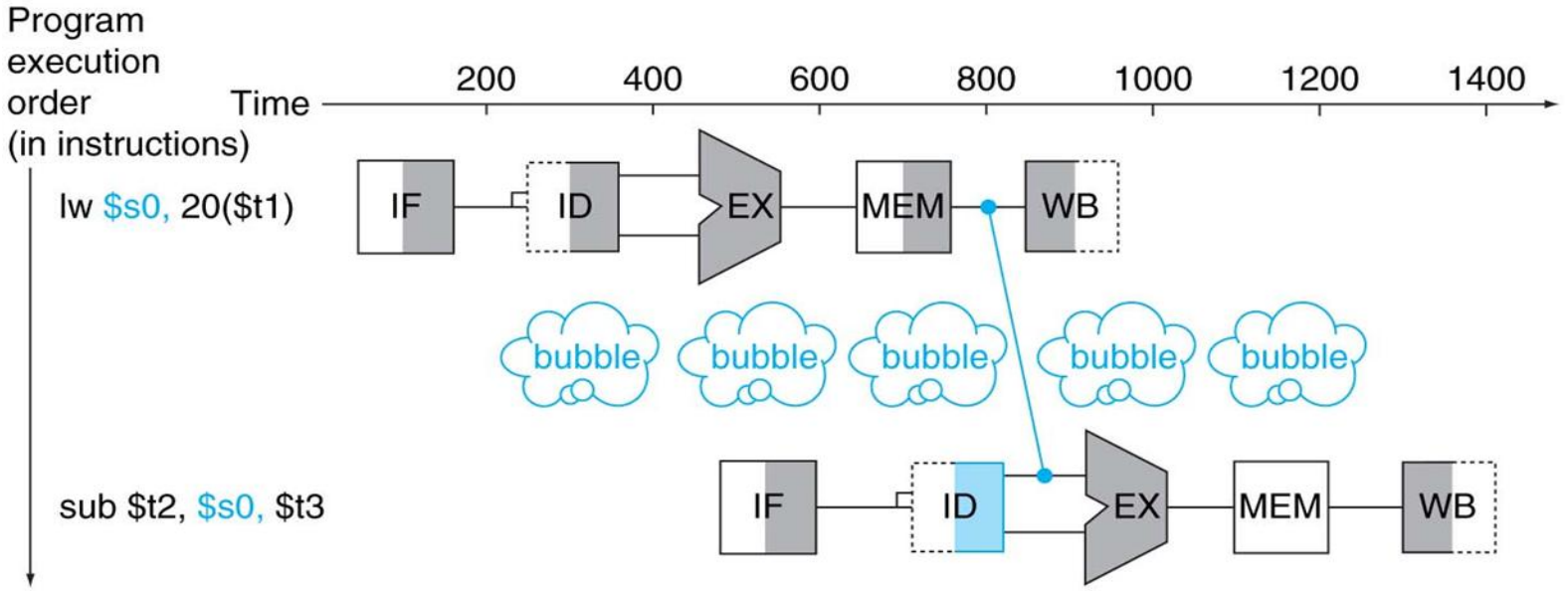


Hazards

- 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

Bubbles in the Pipeline

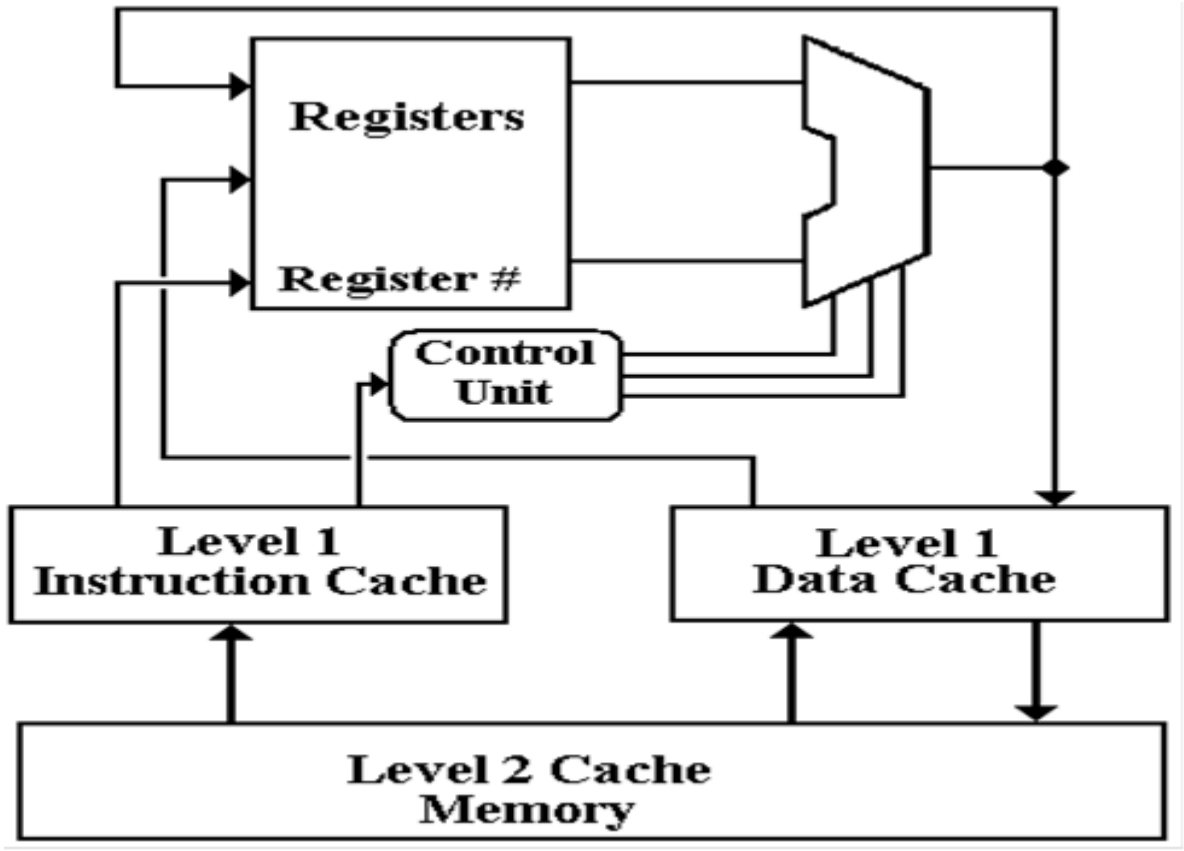
- A **“bubble”**, also called a **“pipeline stall”** occurs when a hazard must be resolved prior to continuation of instruction execution.
- The control unit will insert a **`nop`** instruction into the pipeline in order to stall it.



Structure Hazards

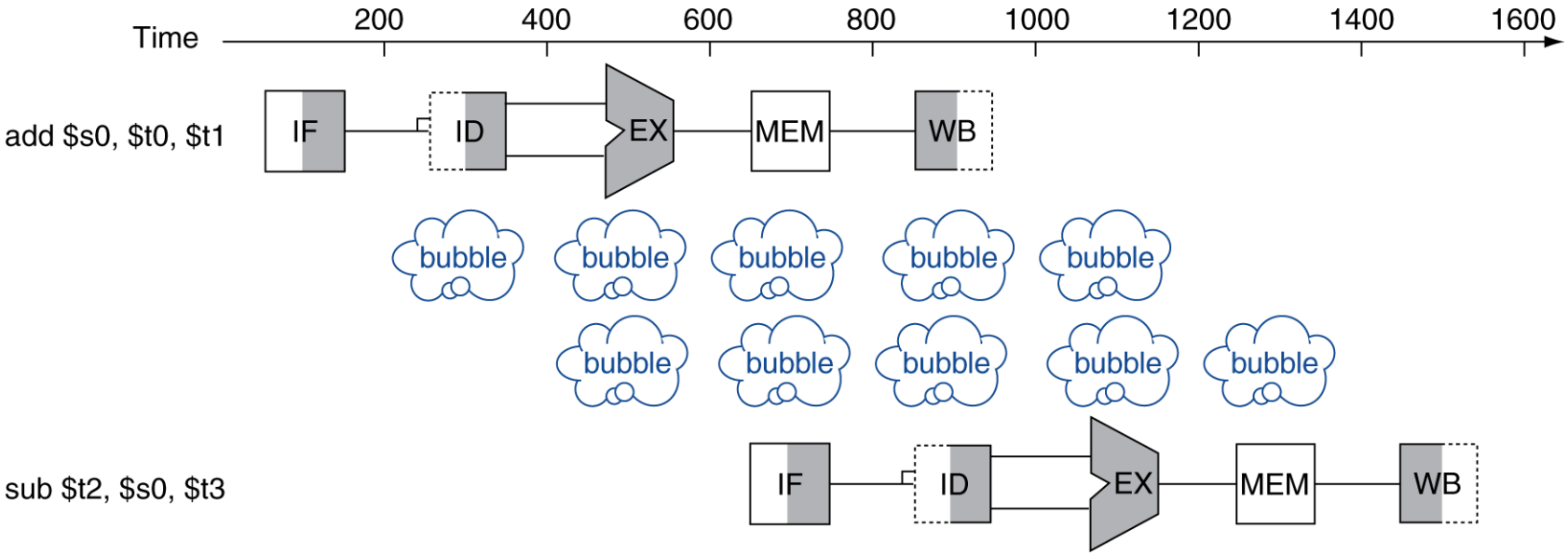
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data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

Avoiding the Structure Hazard



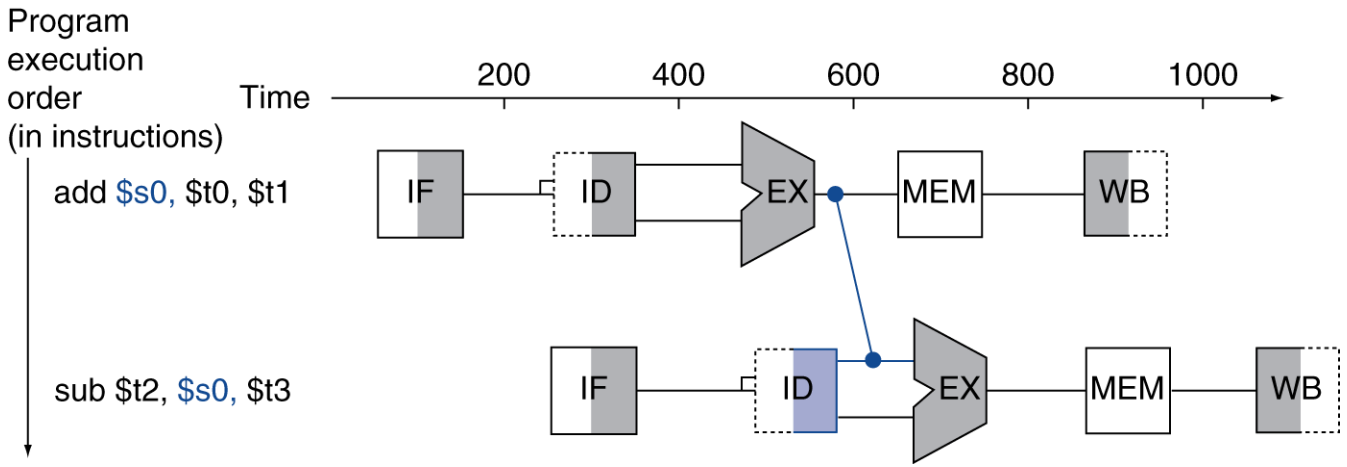
Data Hazards

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



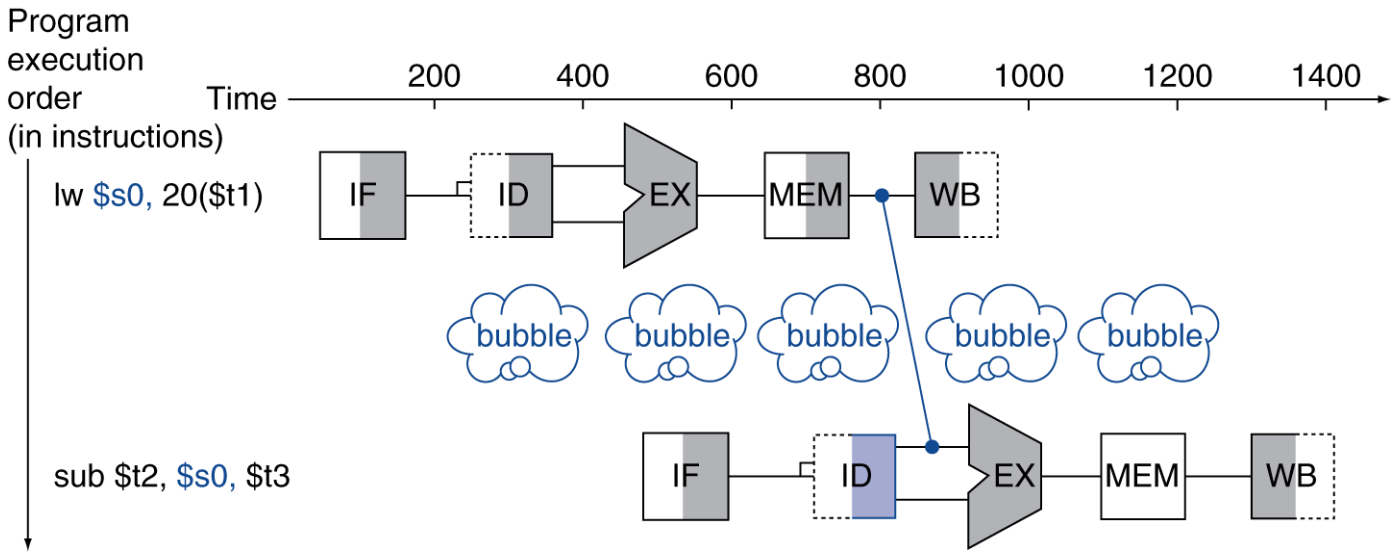
Forwarding (aka Bypassing)

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



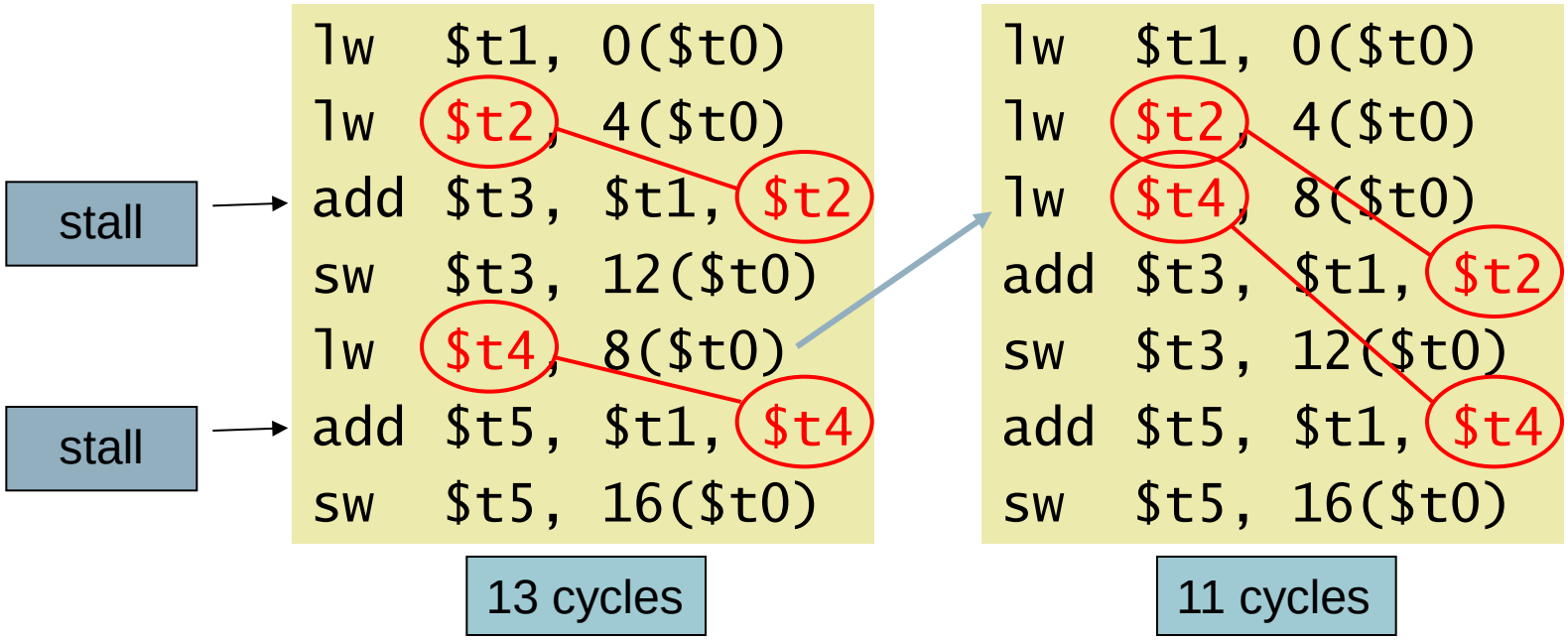
Load-Use Data Hazard

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

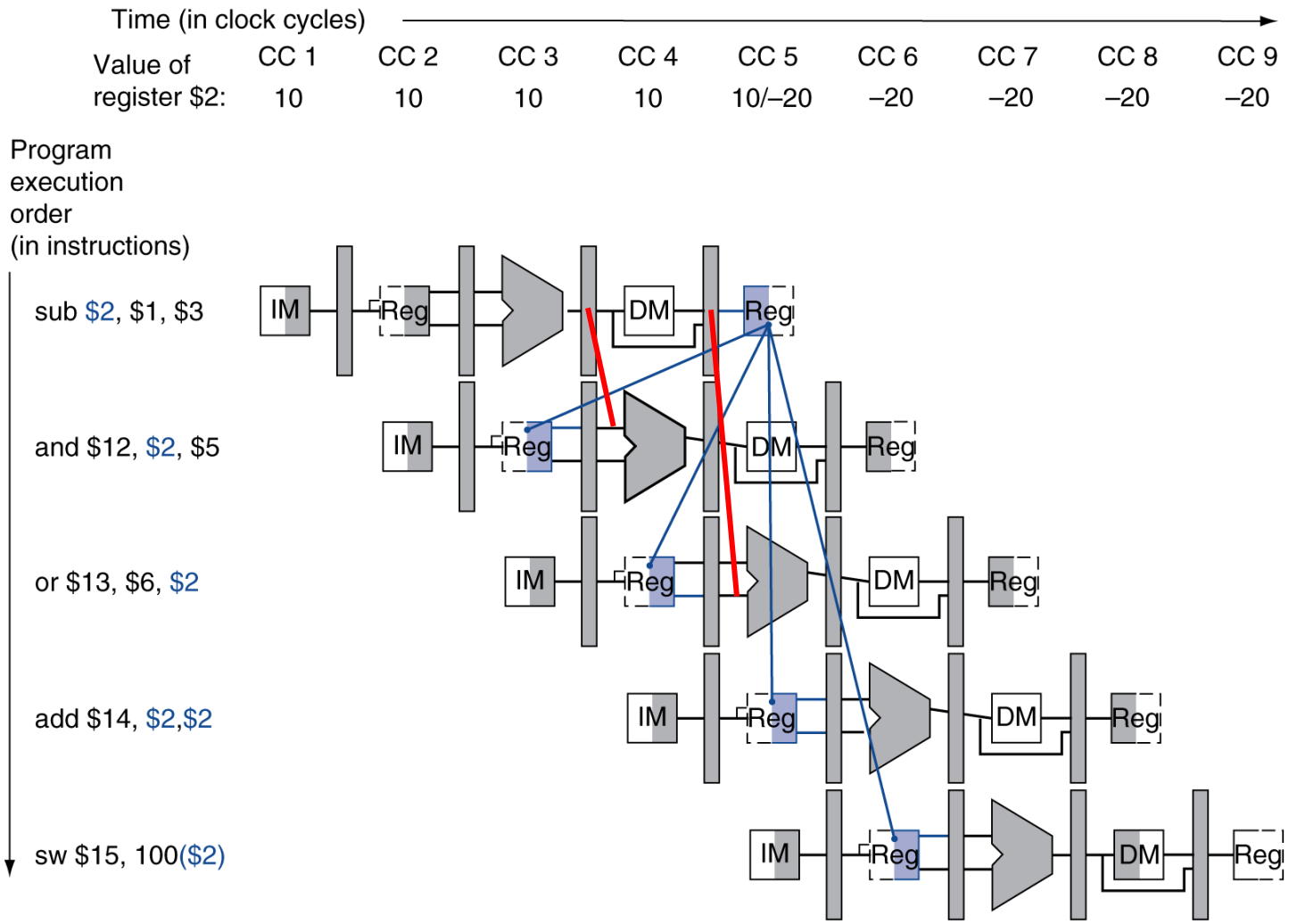


Code Scheduling to Avoid Stalls

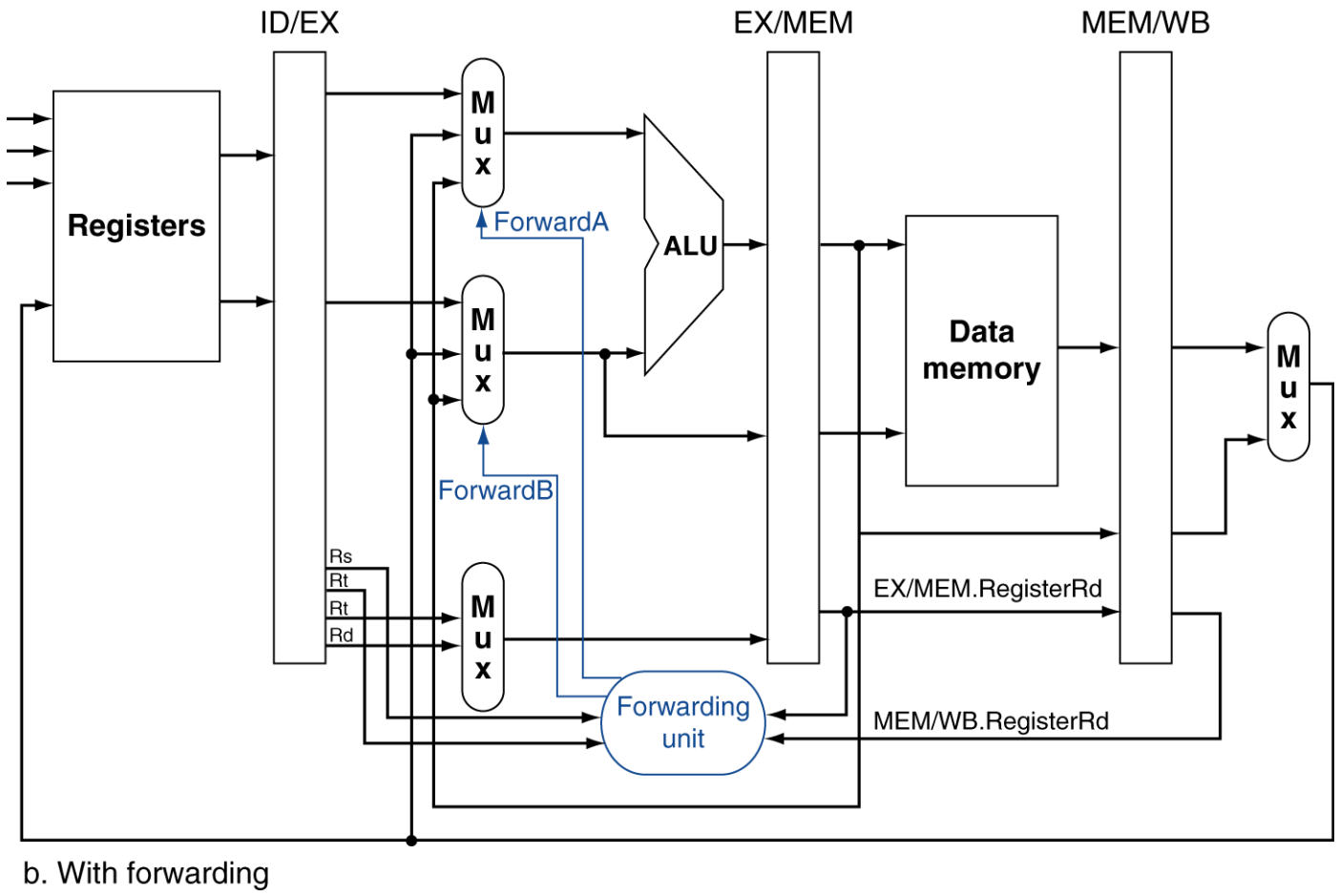
- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`



Dependencies & Forwarding



Forwarding Paths



b. With forwarding

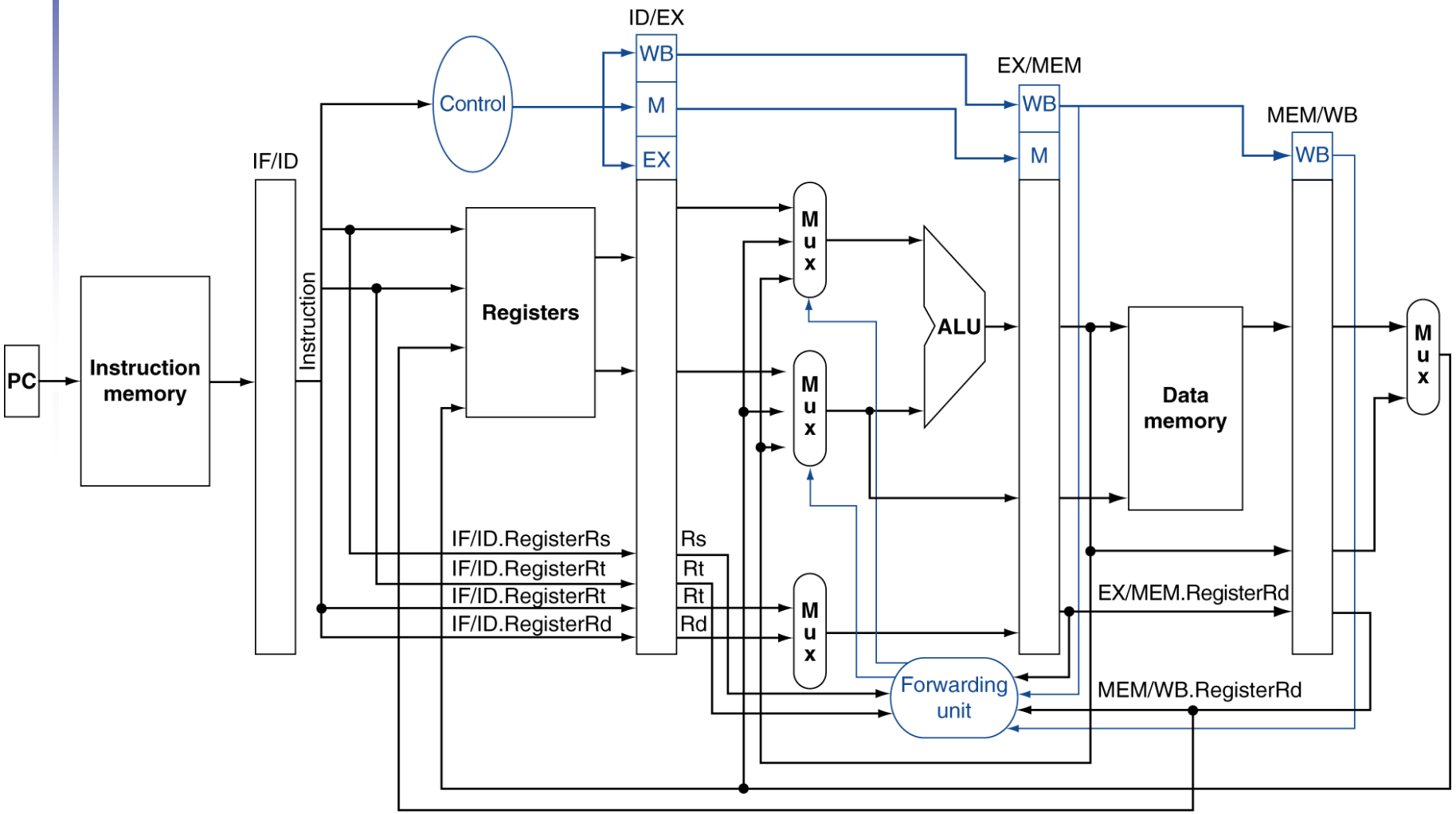
Detecting the Need to Forward

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

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

Datapath with Forwarding



Control Hazards

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

HLL Control Hazard Examples

- **# Branch based on a loop structure.**

Python semantics: xx[10] is the last one processed.

- for k in range (0, 11):
  xx[k] = 0

- **# Branch based on a conditional statement.**
- if (x < y):
  z = y – x
  else:
  z = x – y

Branch for Loop

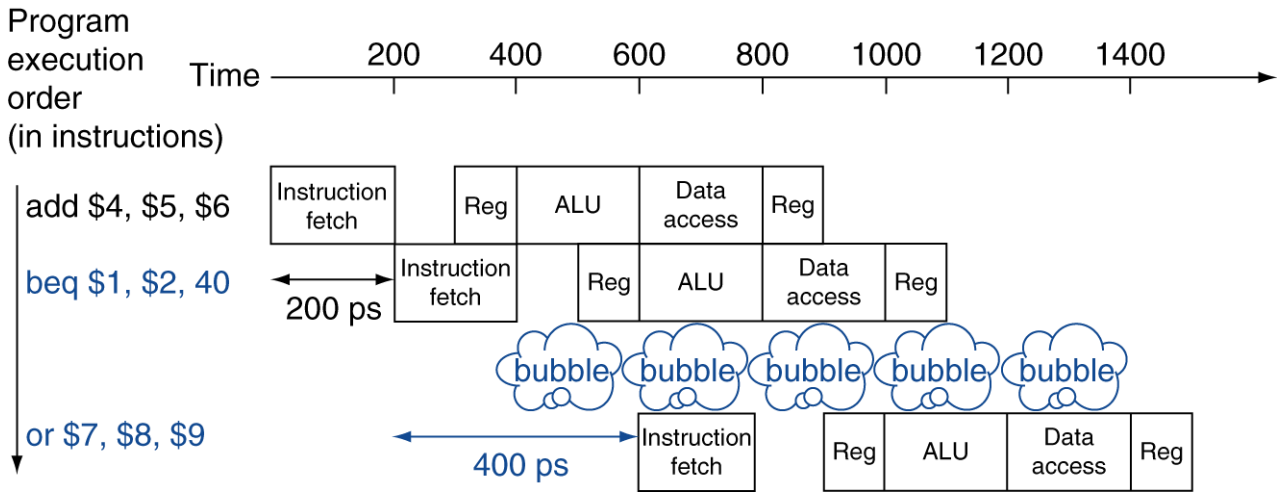
```
100   la    $t0, xx          #Get base address of xx.  Pseudo-
                             #instruction taking 8 bytes.
108   li    $t1, 0           #Initialize counter
112   slti  $t2, $t1,11      #$t2 set to 1 of $t1 < 11
116   beq   $t2, $0, 4       # 120 + 4*4 = 136
120   sw    $0, ($t0)        #Clear word at address in $t0
124   addi  $t0, 4           #Move address to next word
128   addi  $t1, 1           #Increment counter
132   beq   $0,$0, -6        #Branch back to 112
                             #End of loop
136   li    $t0, 0           # 136 - 6*4 = 112
```

Branch for Conditional

```
200   lw    $t1, x          #Get value of x into $t1
204   lw    $t2, y          #Get value of y into $t2
208   slt   $t0, $t1, $t2   #Set $t0 = 1 iff $t1 < $t2
212   beq   $t0, $0, 3      #Branch if $t0 == 0 ($t1 >= $t2)
216   sub   $t3, $t2, $t1   #Form y - x
220   beq   $0, $0, 1       #Go to 228
224   sub   $t3, $t1, $t2   #Form x - y
228   sw    $t3, z          #Store result
```

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

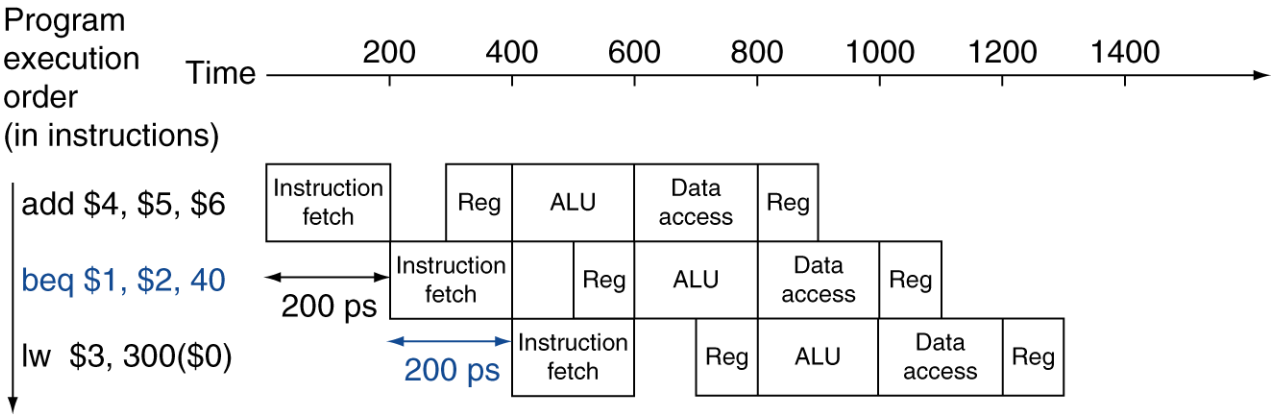


Branch Prediction

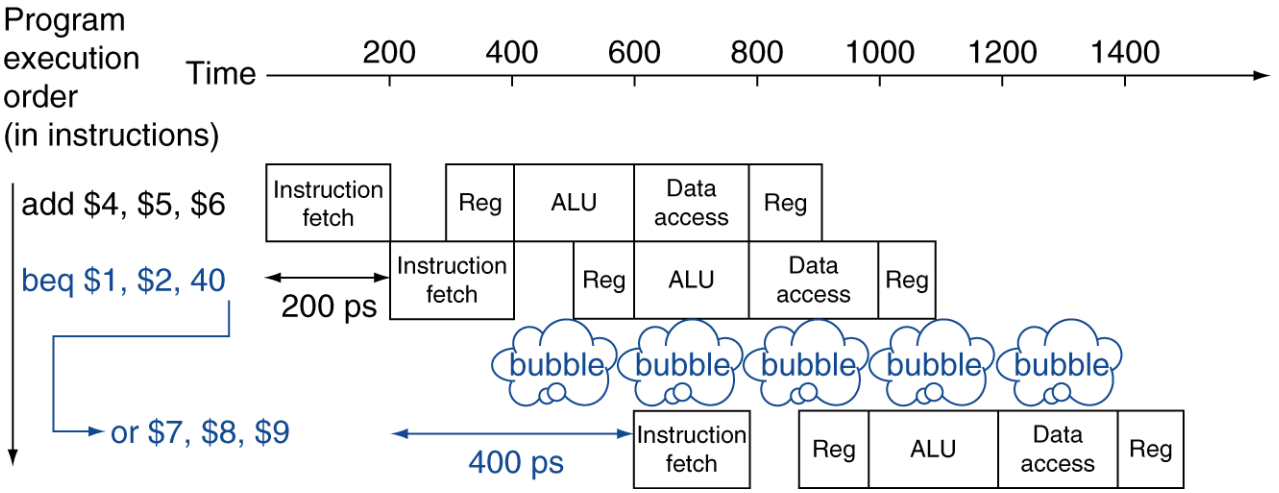
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

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

Unrolling Loops

- Consider the following loop code

```
for ( k = 0; k < 3, k++)
     a[k] = b[k] + c[k];
```

- This is logically equivalent to the following:

```
        k = 0 ;
loop:   a[k] = b[k] + c[k] ;
        k = k + 1;
        if (k < 3) go to loop
        # Here is the branch
```

The Unrolled Loop

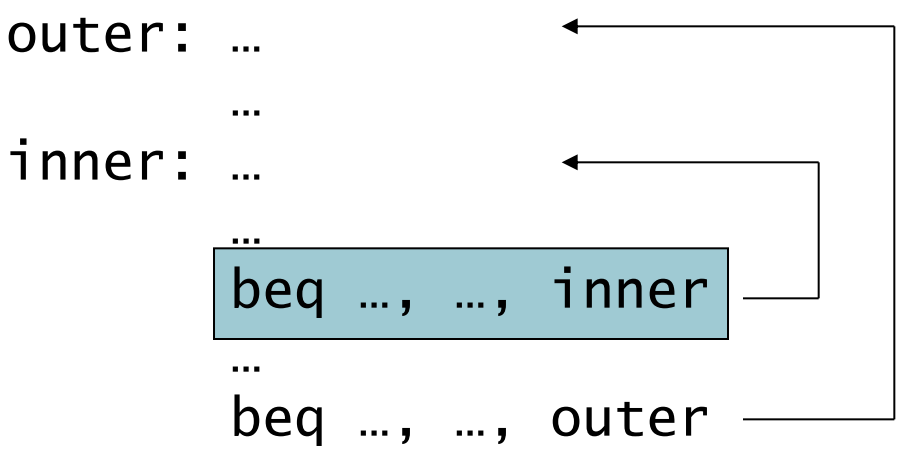
- In this case, unrolling the loop removes the necessity for branch prediction by removing the branch instruction.
- Here is the unrolled code

```
a[0] = b[0] + c[0] ;
a[1] = b[1] + c[1] ;
a[2] = b[2] + c[2] ;
```

- Note: There are no data hazards here. This is the trick used by vector computers.

1-Bit Predictor: Shortcoming

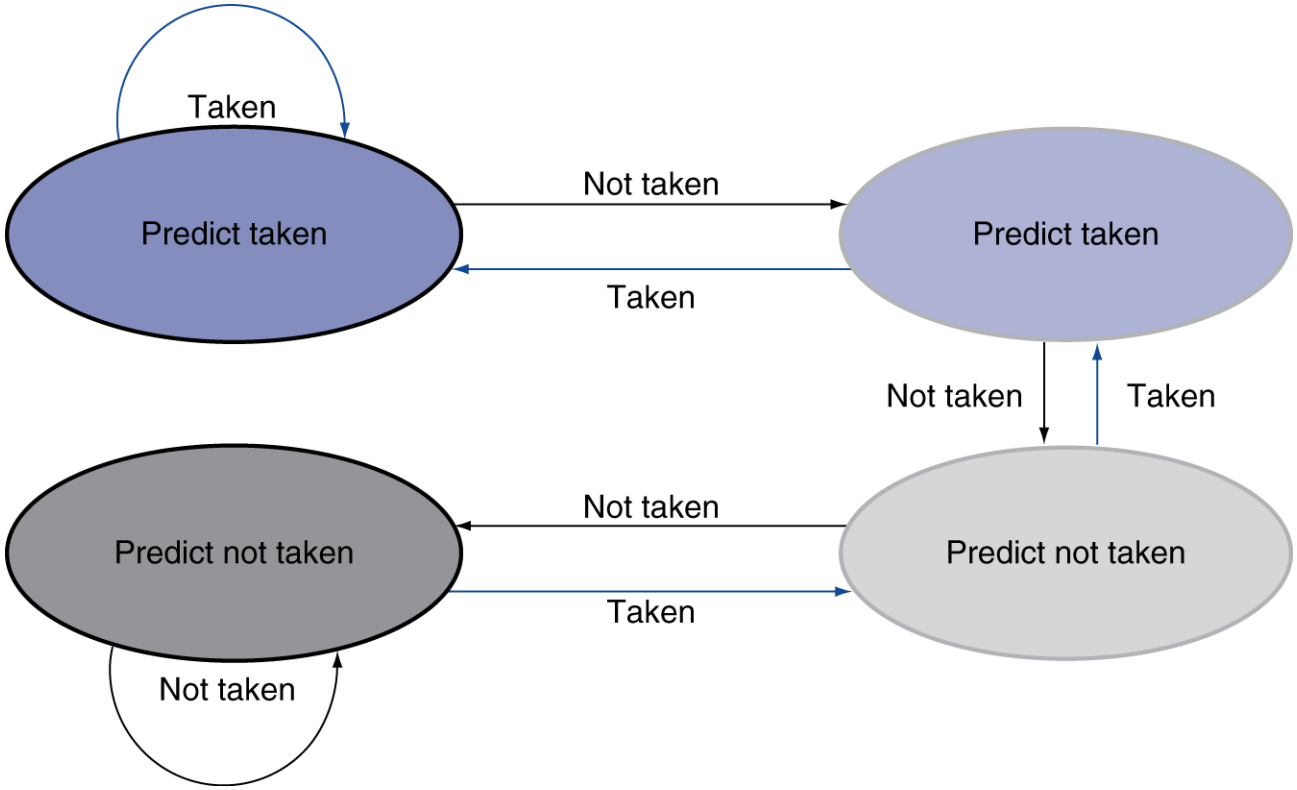
- Inner loop branches mispredicted twice!



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

2-Bit Predictor

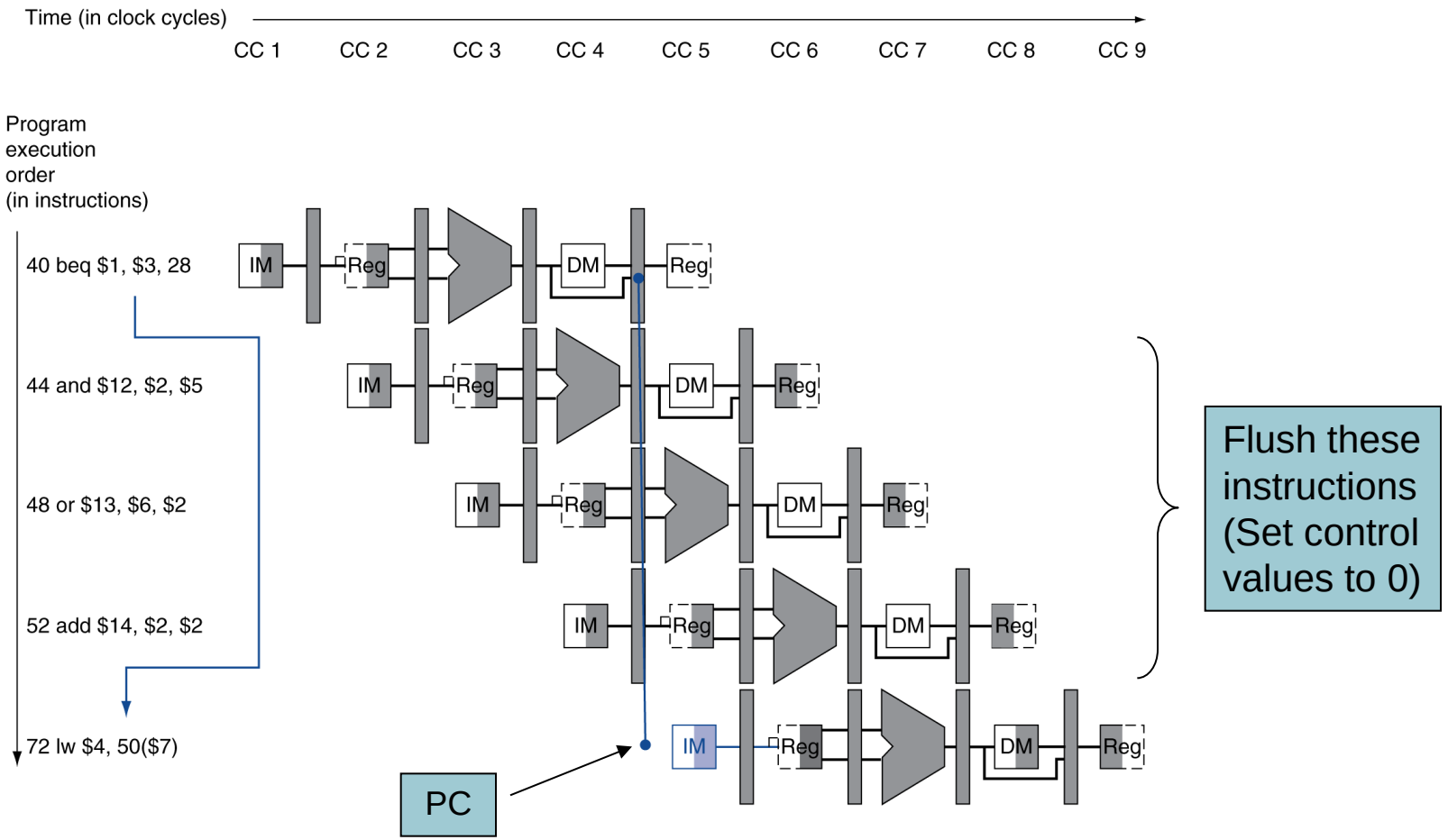
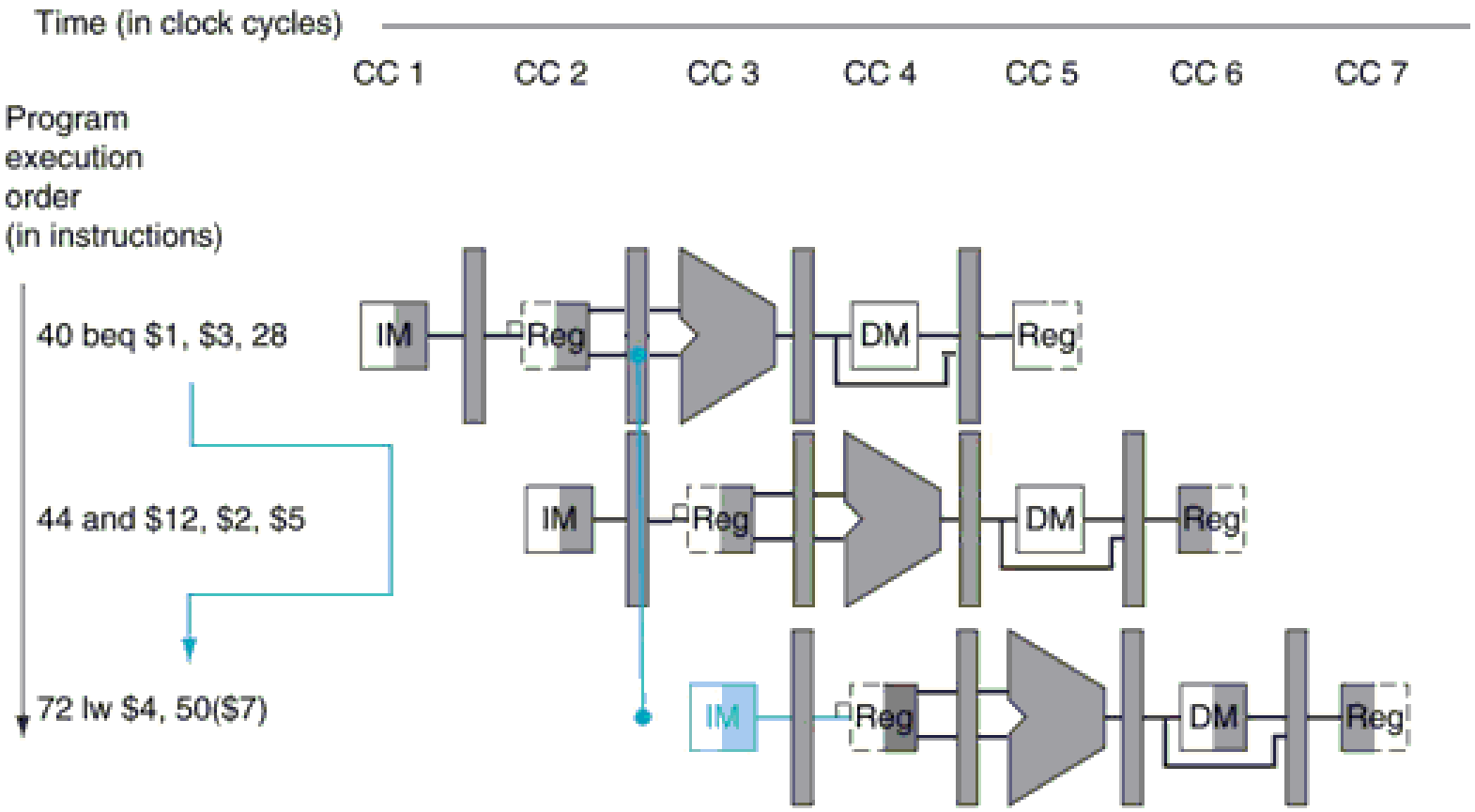
- Only change prediction on two successive mispredictions



Calculating the Branch Target

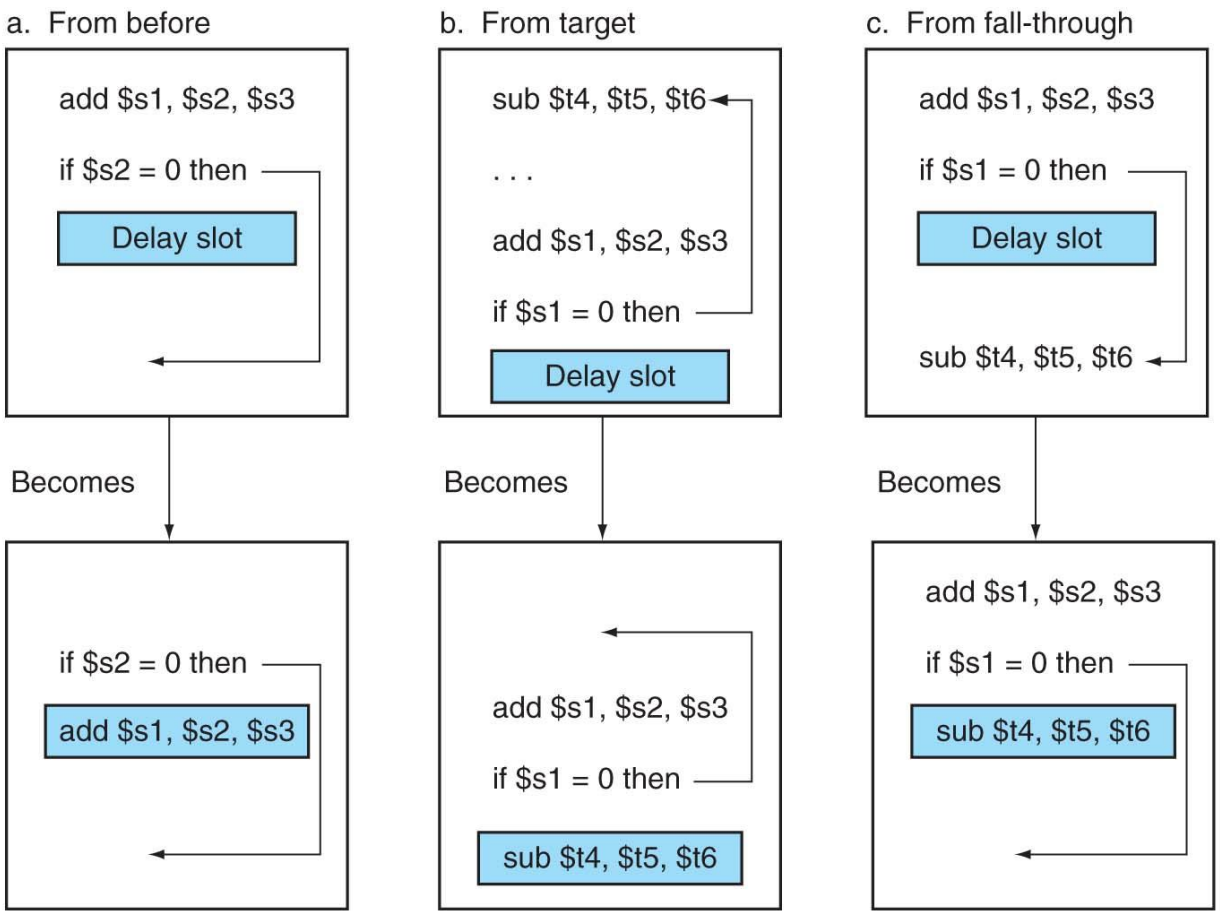
- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

The Branch Penalty

- Consider the following program fragment in which the branch has been taken.
- ```
  36  sub $10, $4, $8
  40  beq $1, $3, 7  #40 + 4 + 7*4
  44  and $12, $2, $5
  48  or $13, $2, $6
  52  add $14, $4, $7
  56  slt $15, $6, $7
      and so on
  72  lw $2, 50($7)
  ```
  ```

Branch Hazards

- If branch outcome determined in MEM



Reducing Branch Delay

- Move the hardware to determine the branch outcome to the ID stage.
- This involves minimal additional hardware.
  - 1. A register comparison unit, comprising XOR gates to handle the branch on equal.
  - 2. An additional adder to determine the target address.
- Now only the instruction in the IF state must be flushed if the branch is taken.

If Branch Outcome Determined in ID



Reorder the Code

- This illustrates the idea of a delay slot.
  The instruction after the **beq** is always executed.
- ```
  36  beq $1, $3, 8  #36 + 4 + 8*4
  40  sub $10, $4, $8 #Always
  44  and $12, $2, $5
  48  or $13, $2, $6
  52  add $14, $4, $7
  56  slt $15, $6, $7
      and so on
  72  lw $2, 50($7)
  ```
  ```

Scheduling the Delay Slot

- The placement depends on the code construct.



Pipeline Summary

The BIG Picture

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