CPE 315: Computer Architecture

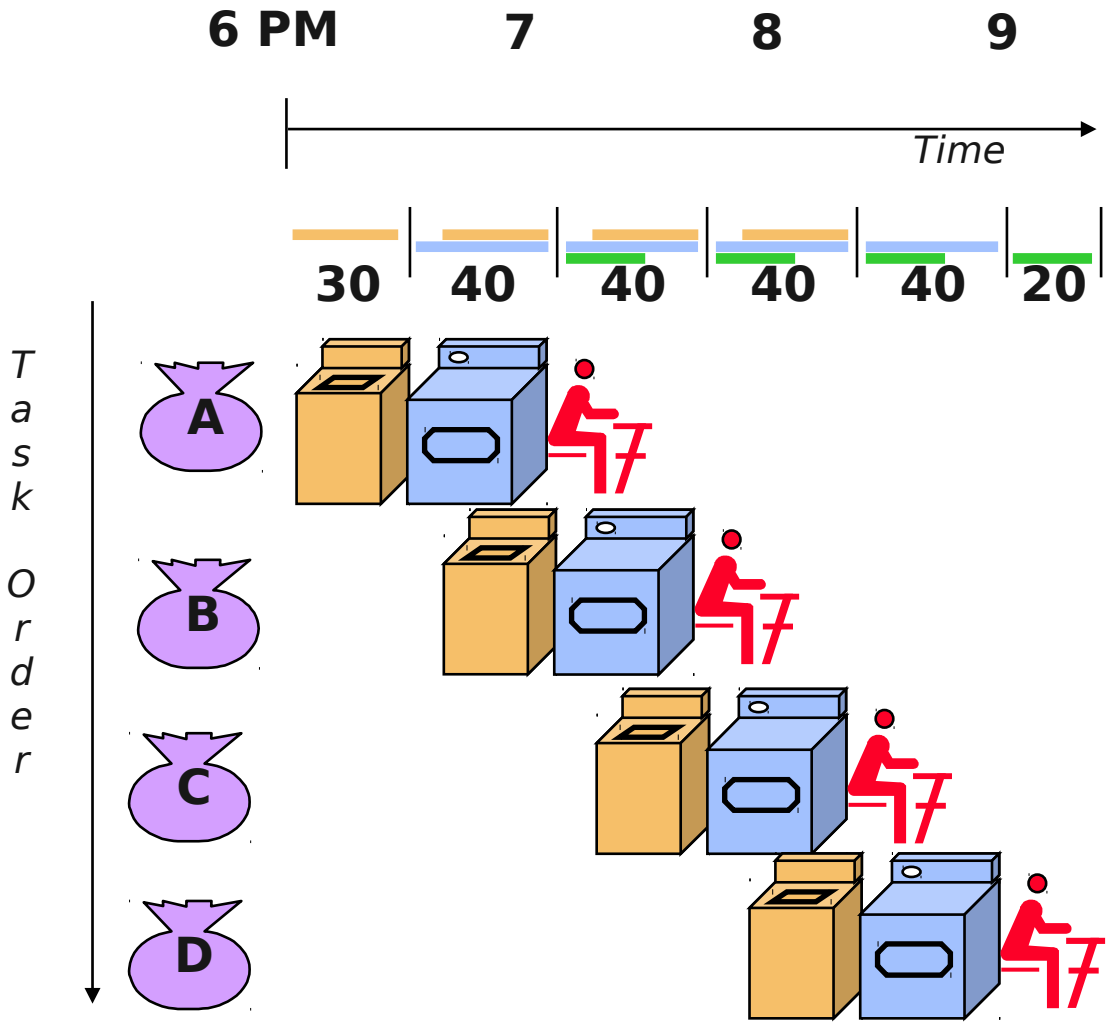
Pipelining Datapath

Chris Lupo

Cal Poly

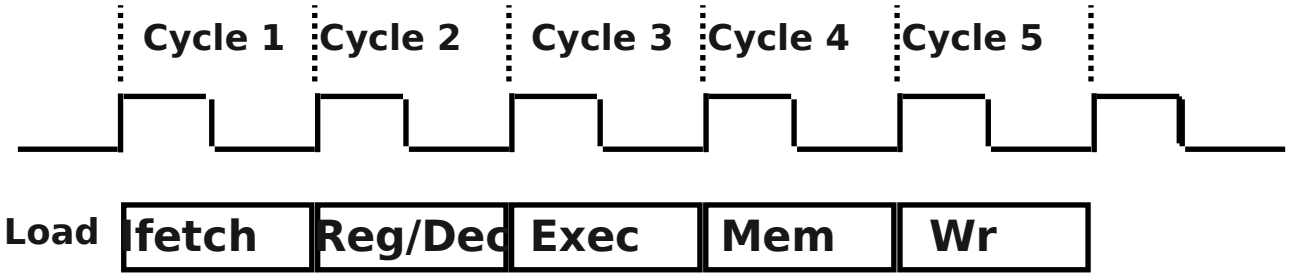
© Some slide material adapted from John Owens / UC Davis, *Computer Organization and Design*, Patterson & Hennessy, © 2005, © Mary Jane Irwin / Penn State 2005, © David Patterson / UCB 2003, © John Kubiatowicz / UCB 2002, © Krste Asinovic/Arvind / MIT 2002, © Morgan Kaufmann Publishers 1998.

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 using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependencies

The Five Stages of Load



Ifetch: Instruction Fetch

- Fetch the instruction from the Instruction Memory

Reg/Dec: Registers Fetch and Instruction Decode

Exec: Calculate the memory address

Mem: Read the data from the Data Memory

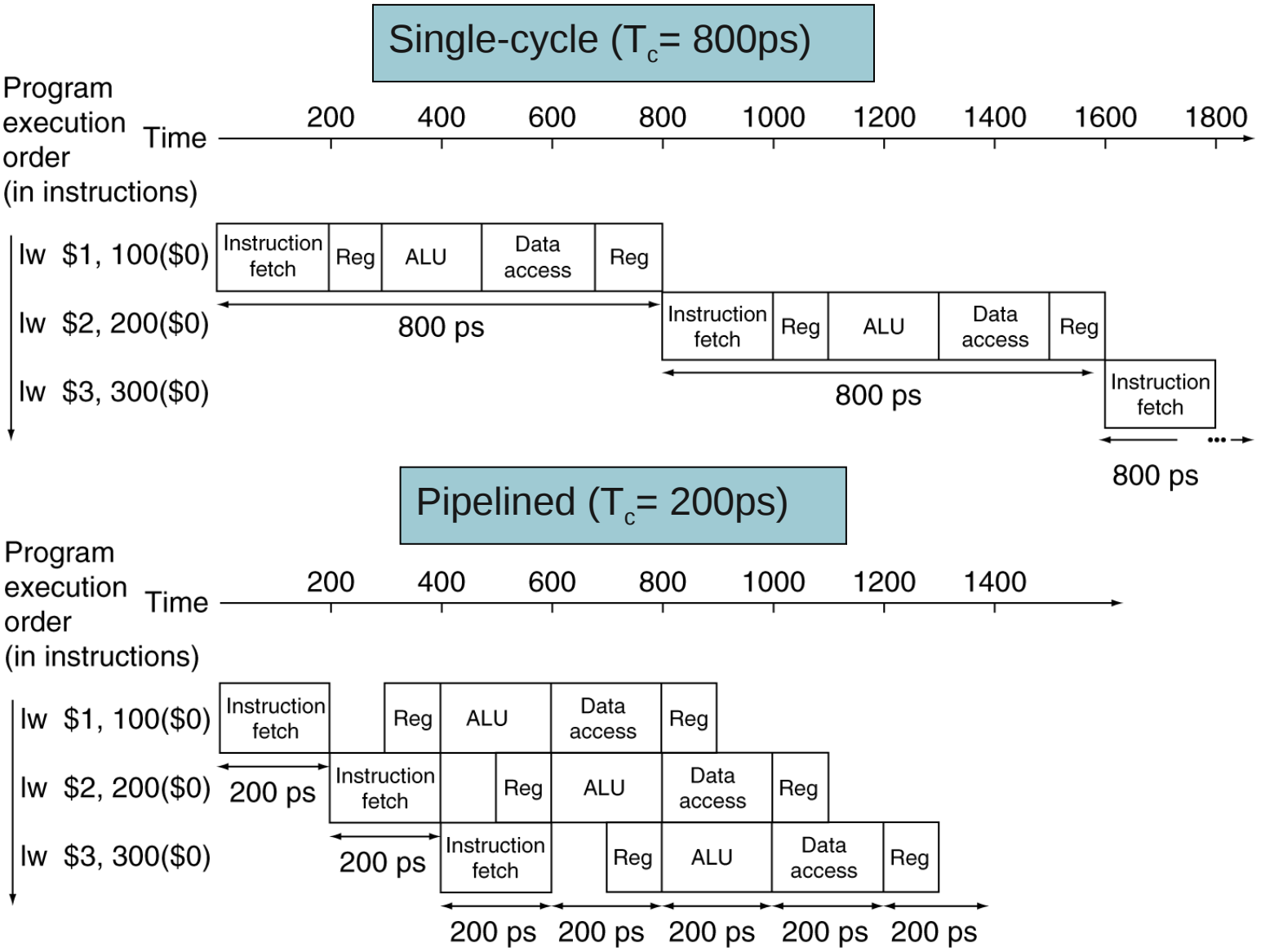
Wr: Write the data back to the register file

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

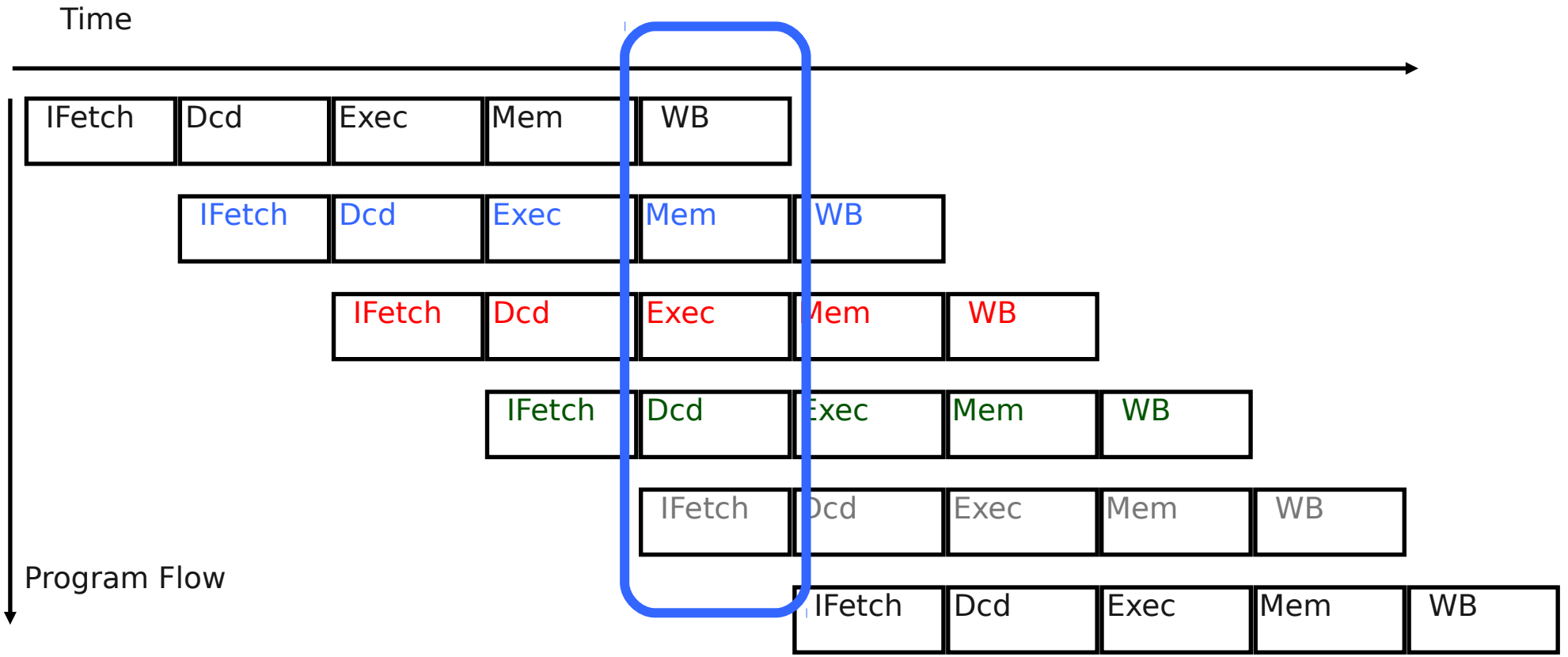
Pipeline Performance



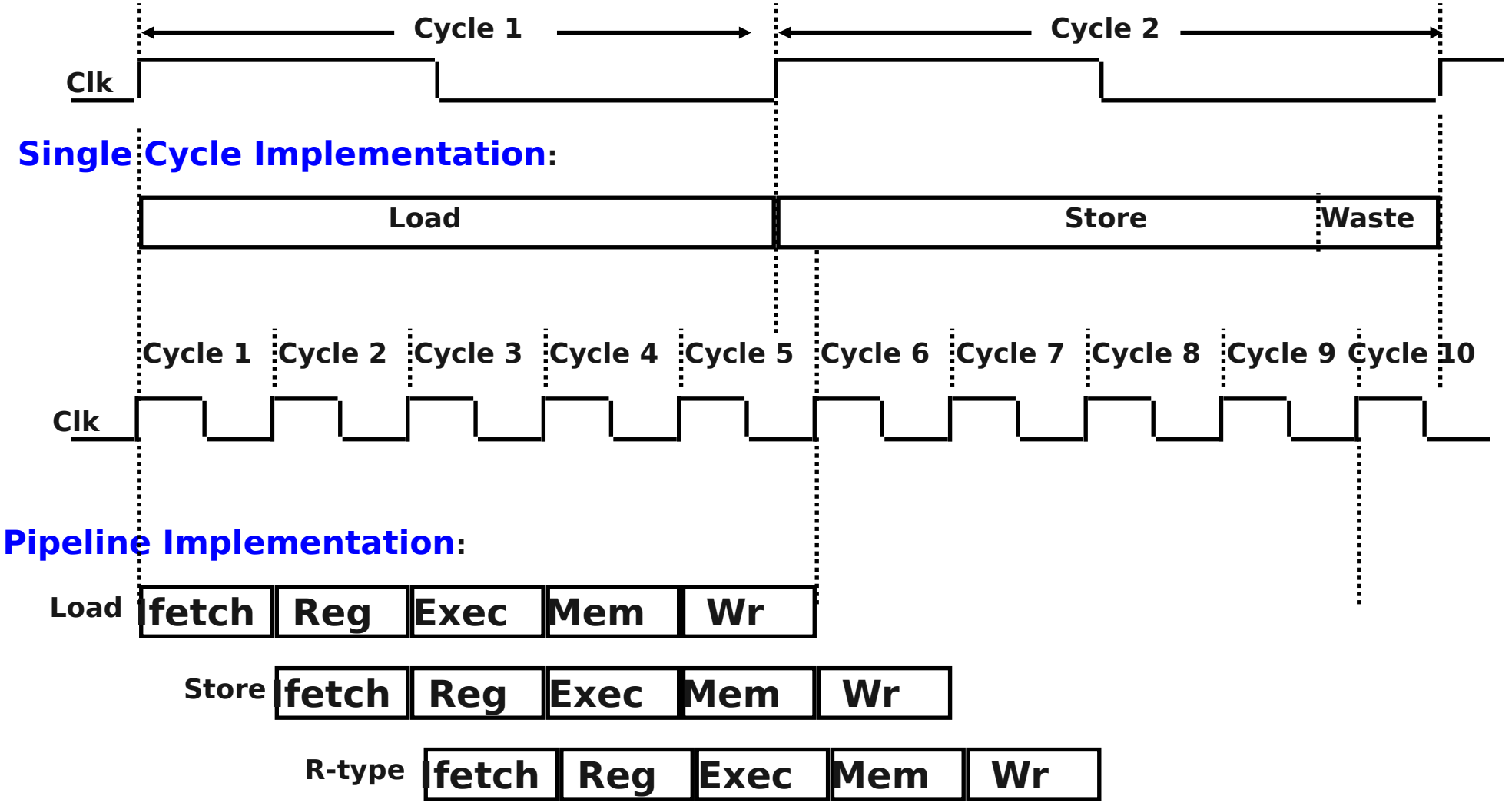
Pipelining and ISA Design

- 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

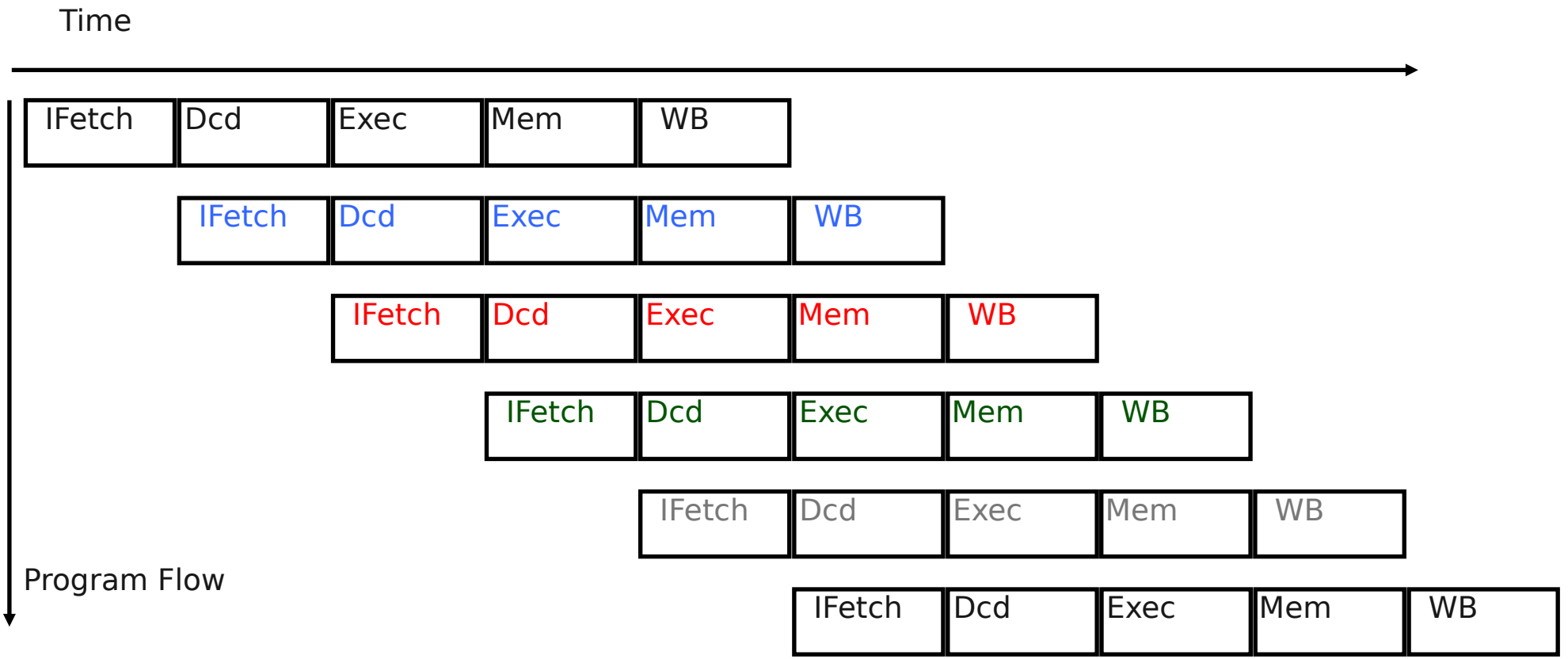
Conventional Pipelined Execution Representation



Single Cycle vs. Pipeline



Pipelining Overhead

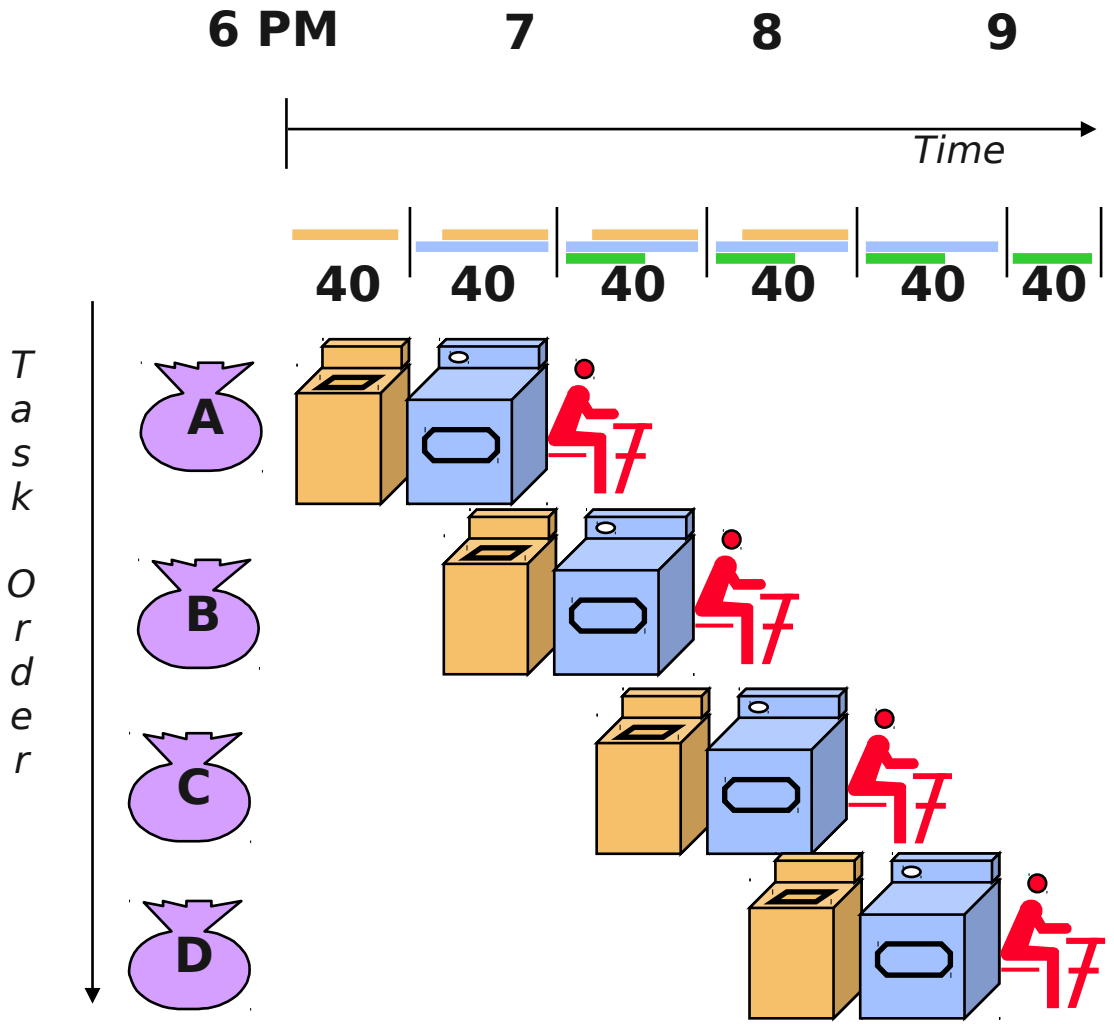


How many instructions?

How many cycles?

Why are they not the same?

Remember the Laundry Example



Total runtime is NOT 40 minutes * number of loads

- This example: 40 minutes per load (4 loads), plus 2*40 minutes extra

Why?

- At start of pipeline, not all hw is busy
- At end of pipeline, not all hw is busy
- You only need to account for one of these

Why Pipeline?

Suppose we execute 100 instructions. How long on each architecture?

Single Cycle Machine

- 4.5 ns/cycle, CPI=1

Ideal pipelined machine

- 1.0 ns/cycle, CPI=1 (but remember fill cost!)

Why Pipeline?

Suppose we execute 100 instructions

Single Cycle Machine

- 4.5 ns/cycle x 1 CPI x 100 inst = 450 ns

Ideal pipelined machine

- 1.0 ns/cycle x (1 CPI x 100 inst + 4 cycle fill) = 104 ns

Fill Costs

Suppose we pipeline different numbers of instructions. What is the overhead of pipelining?

Ideal pipelined machine

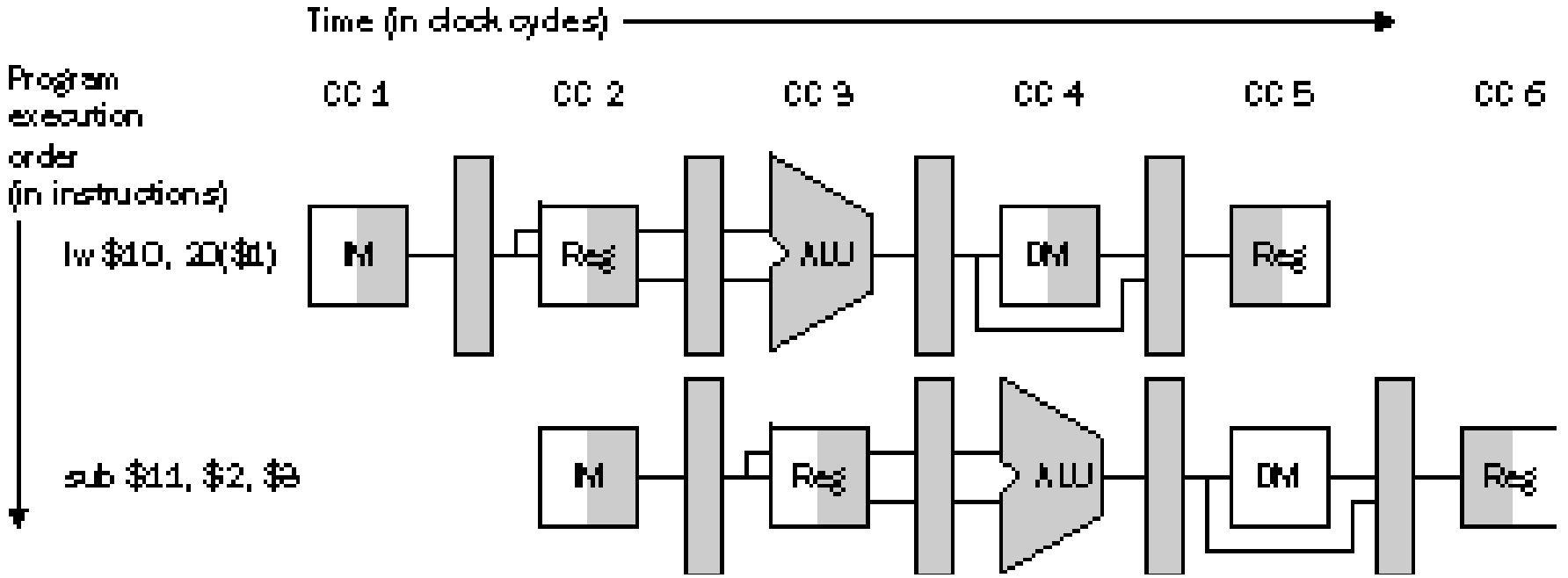
- 1.0 ns/cycle, CPI=1, 10 instructions
- 1.0 ns/cycle, CPI=1, 1000 instructions
- 1.0 ns/cycle, CPI=1, 100,000 instructions

Fill Costs

Overhead: Ideal pipelined machine

- 1.0 ns/cycle, CPI=1, 10 instructions
- 10 ns runtime + 4 ns overhead: 40% overhead
- 1.0 ns/cycle, CPI=1, 1000 instructions
- 1000 ns runtime + 4 ns overhead: 0.4% overhead
- 1.0 ns/cycle, CPI=1, 100,000 instructions
- 100,000 ns runtime + 4 ns overhead: 0.004% overhead

Graphically Representing Pipelines



Can help with answering questions like:

- how many cycles does it take to execute this code?
- what is the ALU doing during cycle 4?
- use this representation to help understand datapaths

Pentium 4 Pipeline

1–2: Trace cache next instruction pointer

3–4: Trace cache fetch

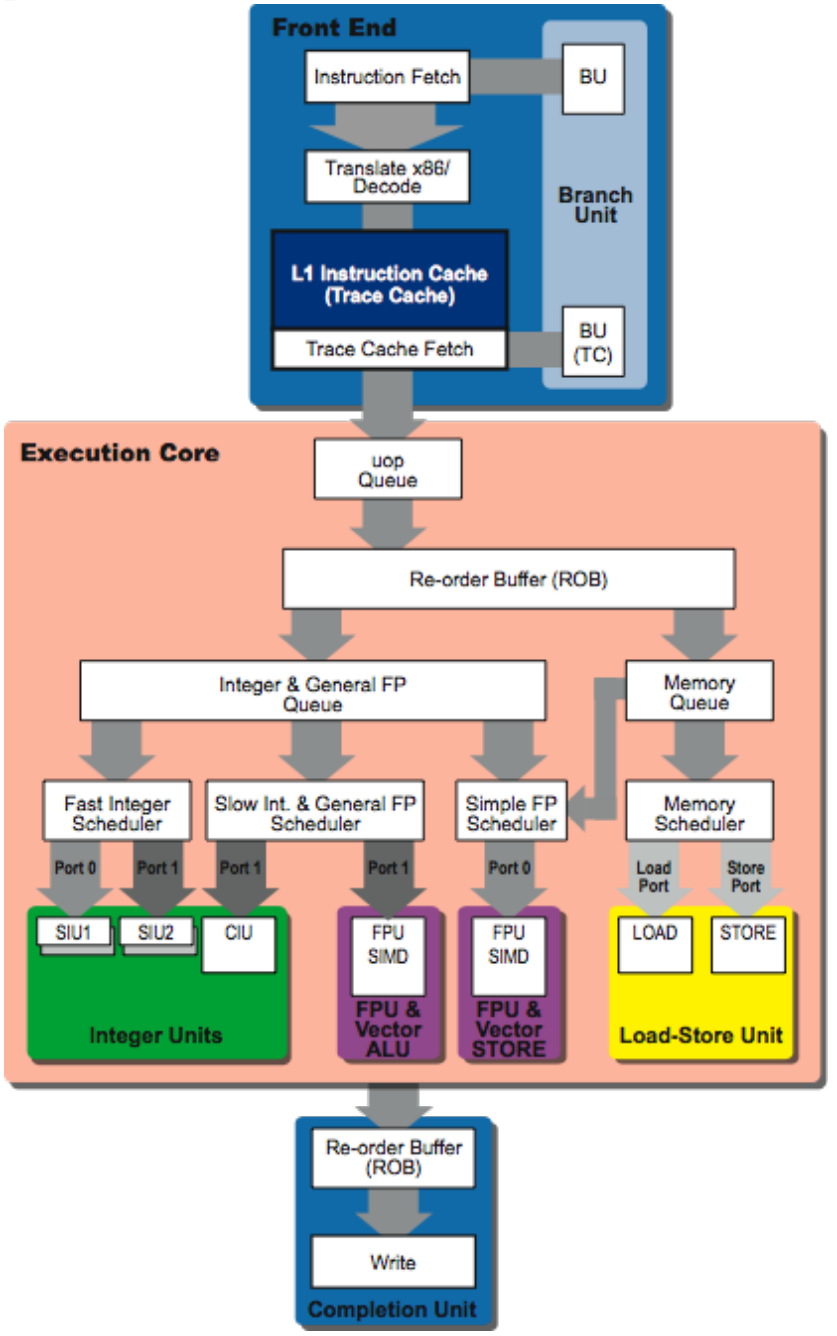
5: Drive

- Up to 3 µops now sent into µop queue

6–8: Allocate and rename

- Each µop enters 1 queue, up to 3 µops into queue

9: Queue (in-order queue within queue)



Pentium 4 Pipeline

10–12: Schedule

- 8–12 entry mini-µop queue, arbitrates for one of 4 issue ports

13–14: Issue

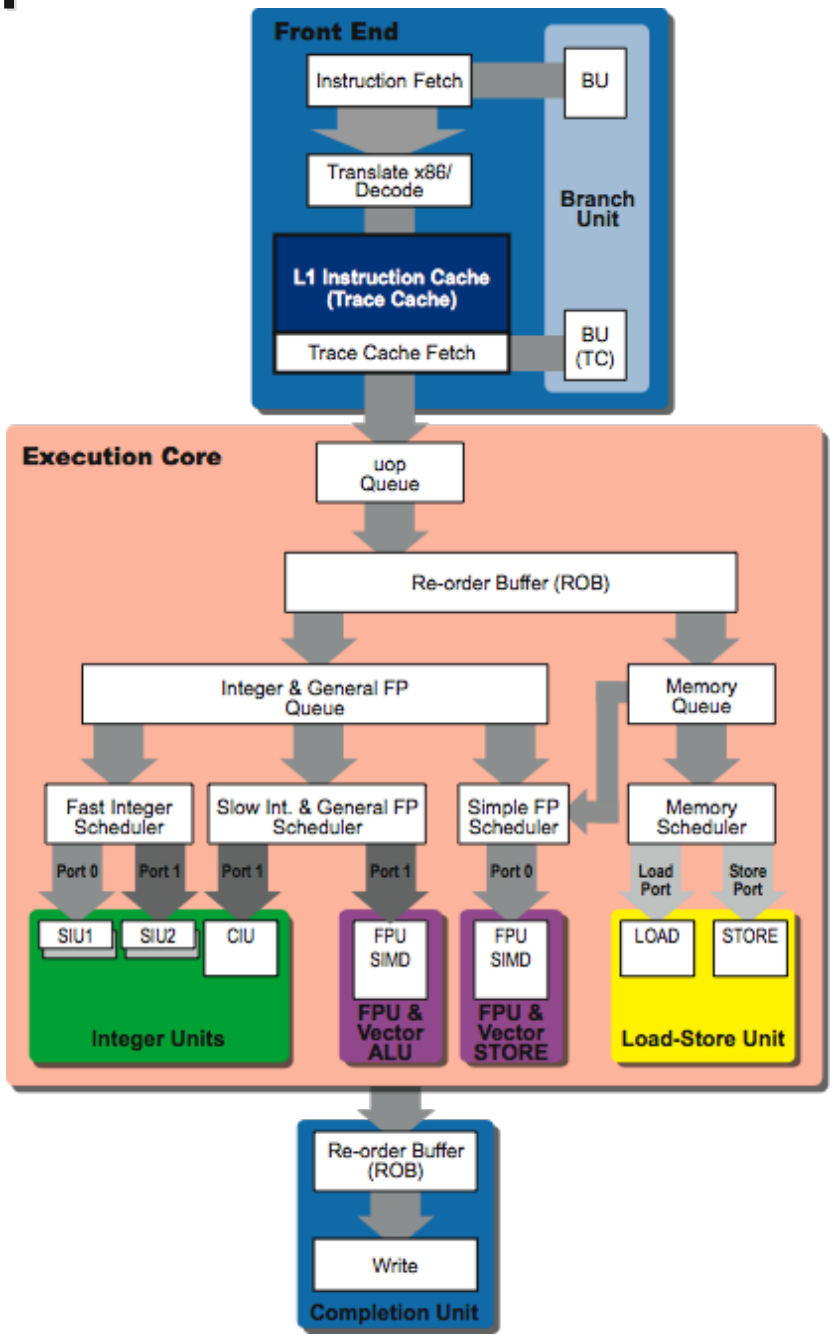
- Up to 6 µops/cycle thru 4 ports
- 2 execution ports are double-clocked

15–16: Register files

17: Execute 18: Flags

19: Branch check 20: Drive

21+: Complete and commit



Can pipelining get us into trouble?

Yes: Pipeline Hazards

- structural hazards: attempt to use the same resource two different ways at the same time
- control hazards: attempt to make a decision before condition is evaluated
- data hazards: attempt to use item before it is ready

Can always resolve hazards by waiting

- pipeline control must detect the hazard
- take action (or delay action) to resolve hazards

Data Hazards

Data hazards: attempt to use item before it is ready

- E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
- instruction depends on result of prior instruction still in the pipeline

Data Hazard Terminology

Could the wrong thing happen?

RAW: Read After Write

- "True Dependence" or "Data Dependence"
- add $t0, $s0, $s1
  sub $t2, $t0, $s3

WAW: Write After Write

- "Output Dependence"
- sub $t2, $t0, $s3
  or $t2, $t7, $s2

WAR: Write After Read

- "Name dependence" or "Antidependence"
- sub $t2, $t0, $s3
  or $s3, $t7, $s4

RAR: Read After Read?

Know this terminology!

Identify the Hazards!

ADD R1, R2, R3

DIV R8, R1, R3

ADD R3, R4, R5

ADD R3, R6, R7

Identify the Hazards!

ADD R1, R2, R3

R1: RAW or Data Dependence

DIV R8, R1, R3

R3: WAR or Anti-dependence

ADD R3, R4, R5

R3: WAW or Output Dependence

ADD R3, R6, R7

MIPS and Data Hazards

Can these hazards ever happen in MIPS 5-stage pipeline?

RAW: Read After Write

- Definitely. Next few slides.

WAW: Write After Write

- Not possible - why?
- All writes happen in same stage.

WAR: Write After Read

- Not possible - why?
- All writes happen after all reads.

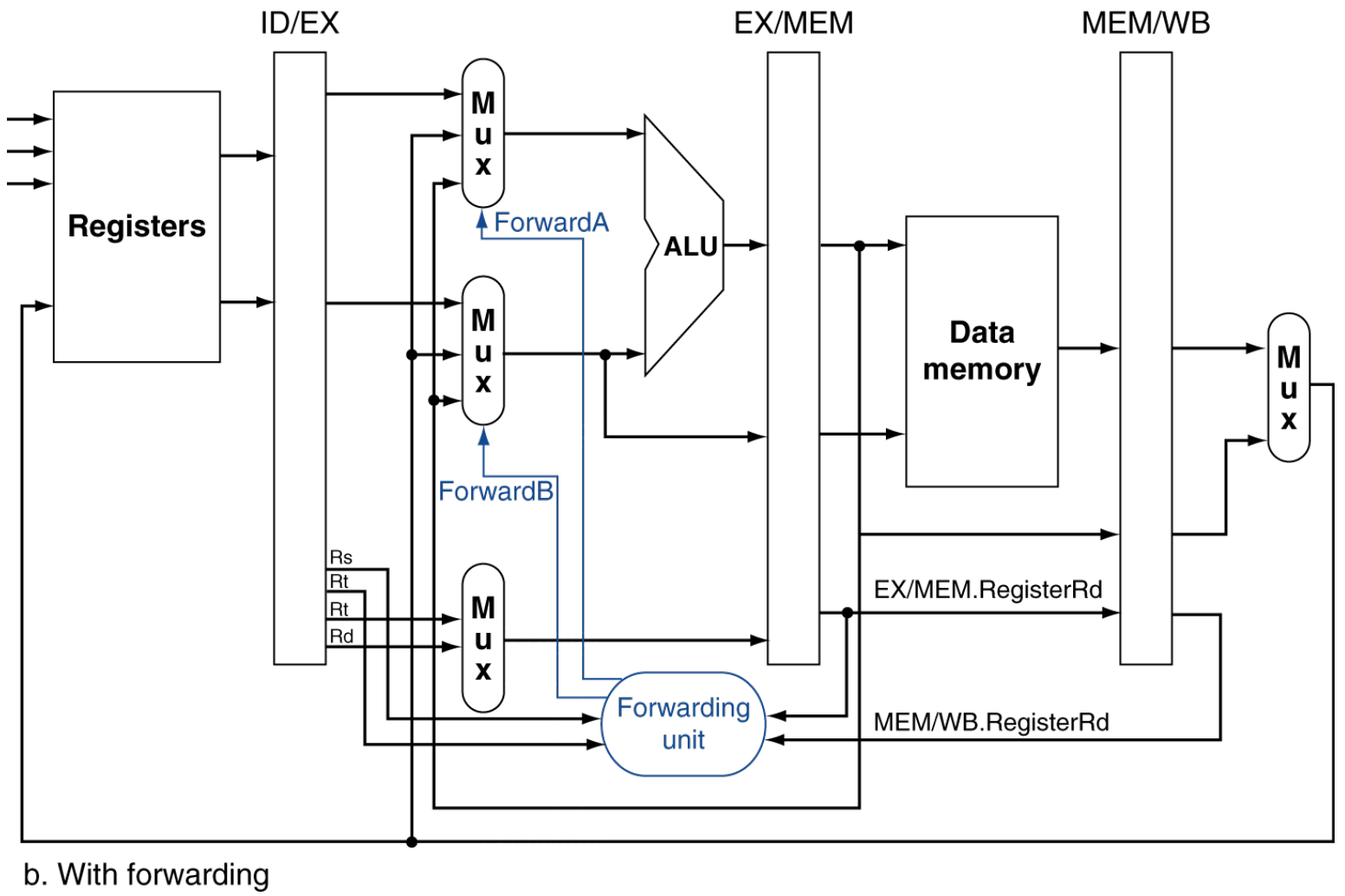
Data Hazards in ALU Instructions

- Consider this sequence:

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

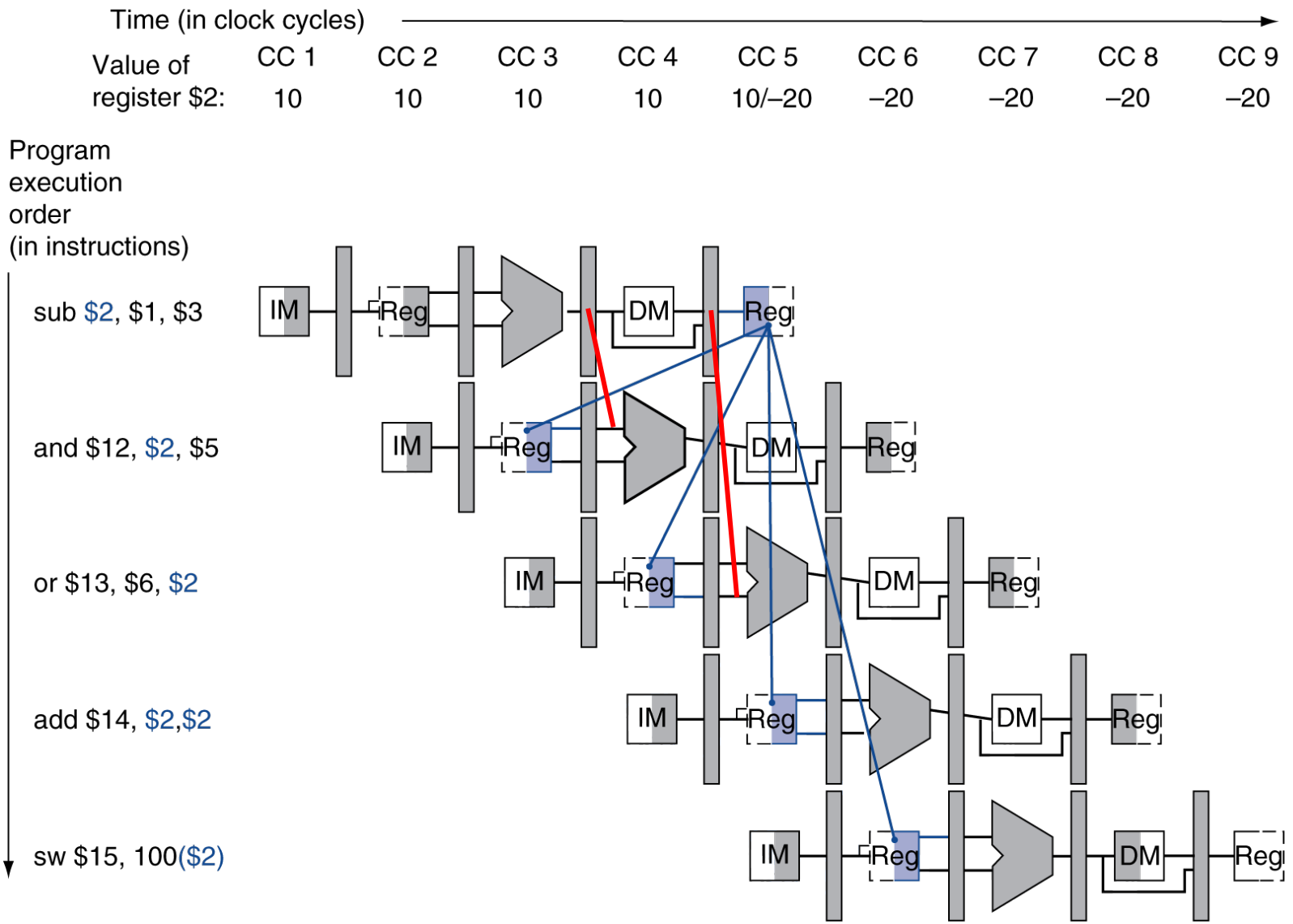
- We can resolve hazards with forwarding
  - How do we detect when to forward?

Forwarding Paths



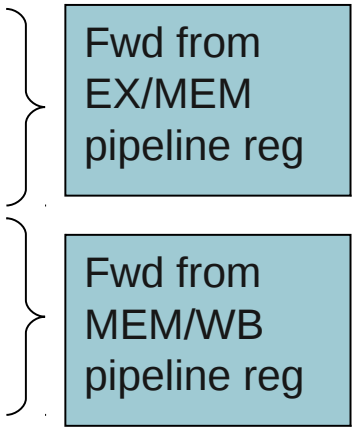
b. With forwarding

Dependencies & Forwarding



Detecting the Need to Forward

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



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 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:

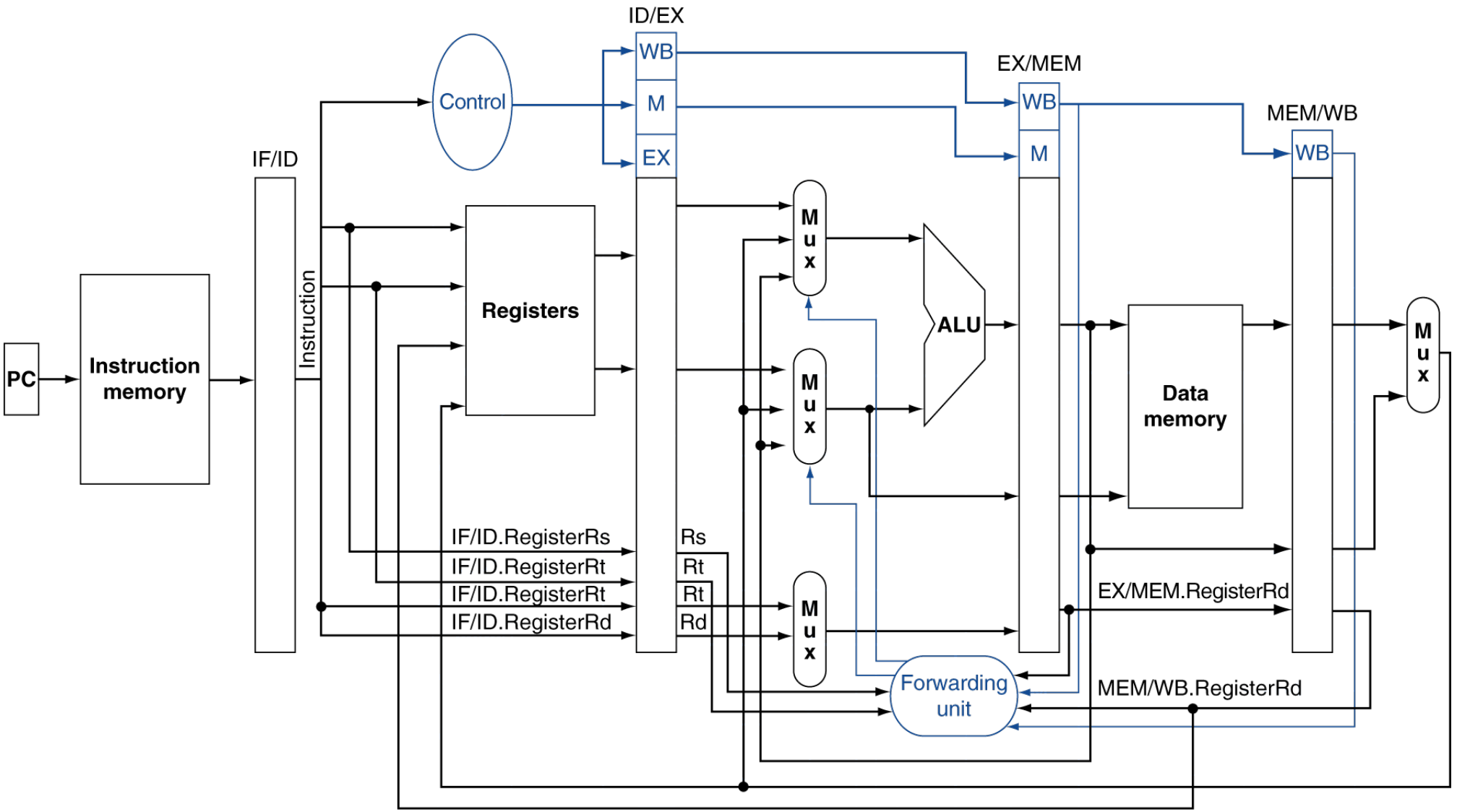
```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

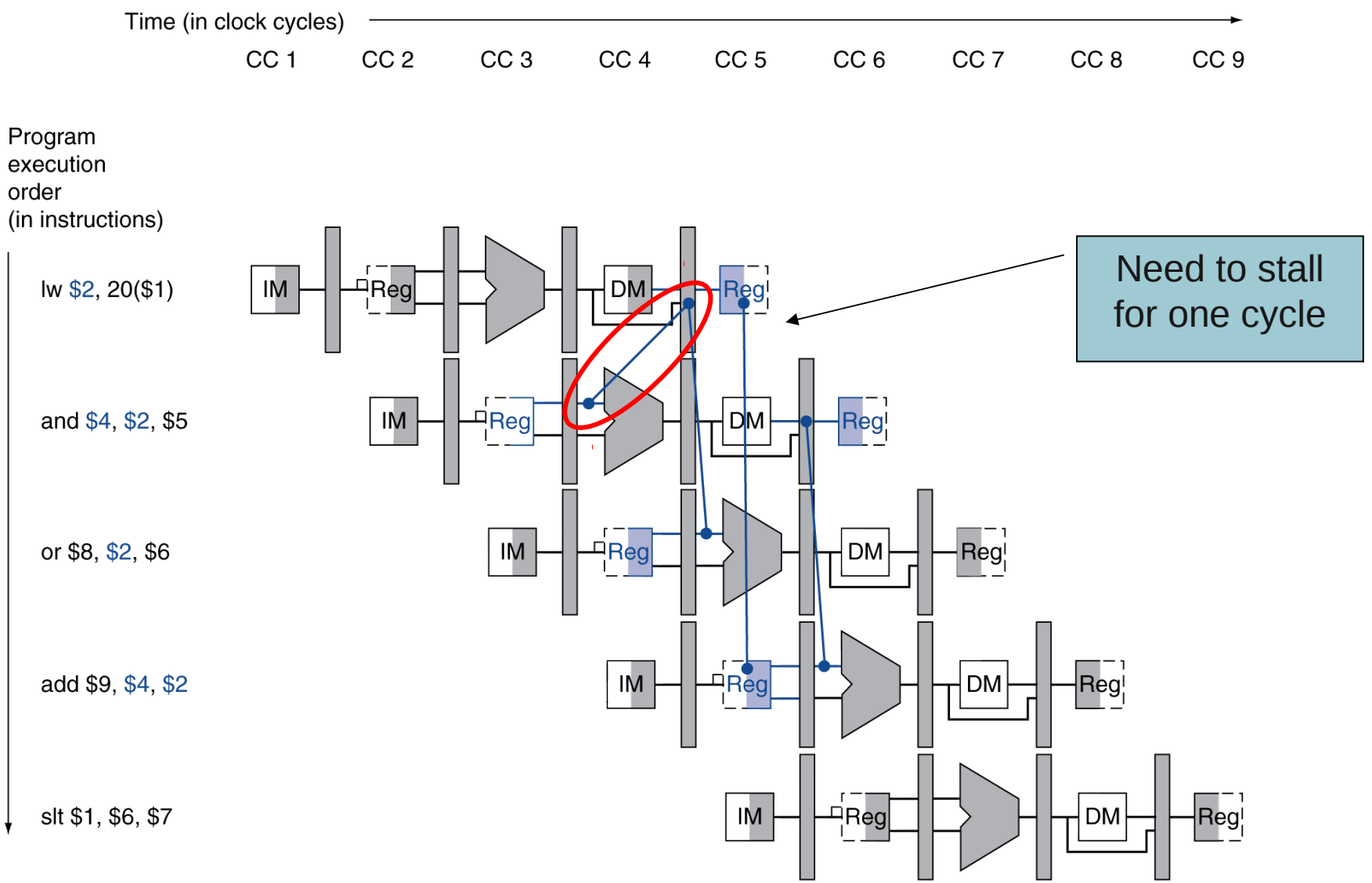
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 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



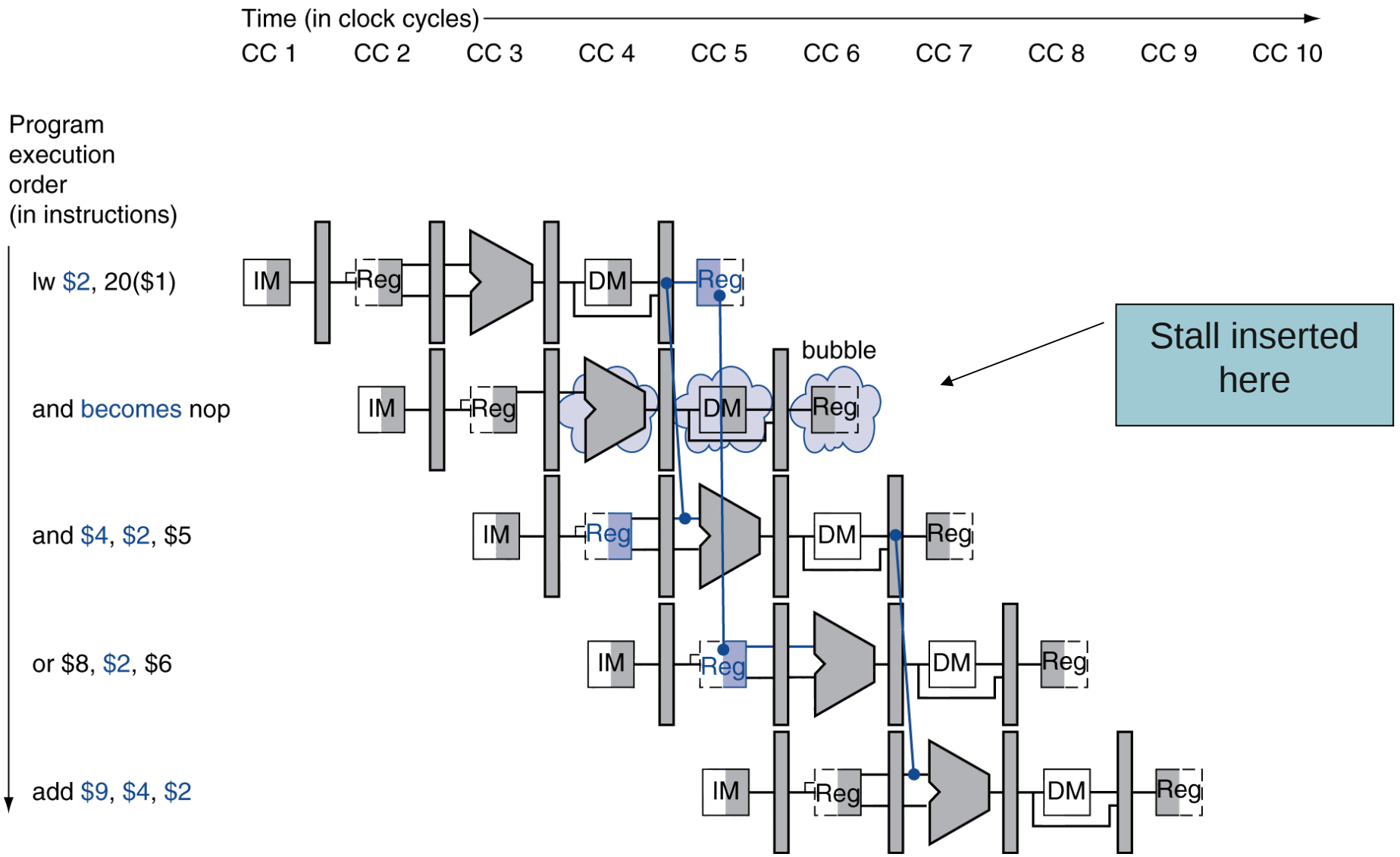
Load-Use Hazard Detection

- 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

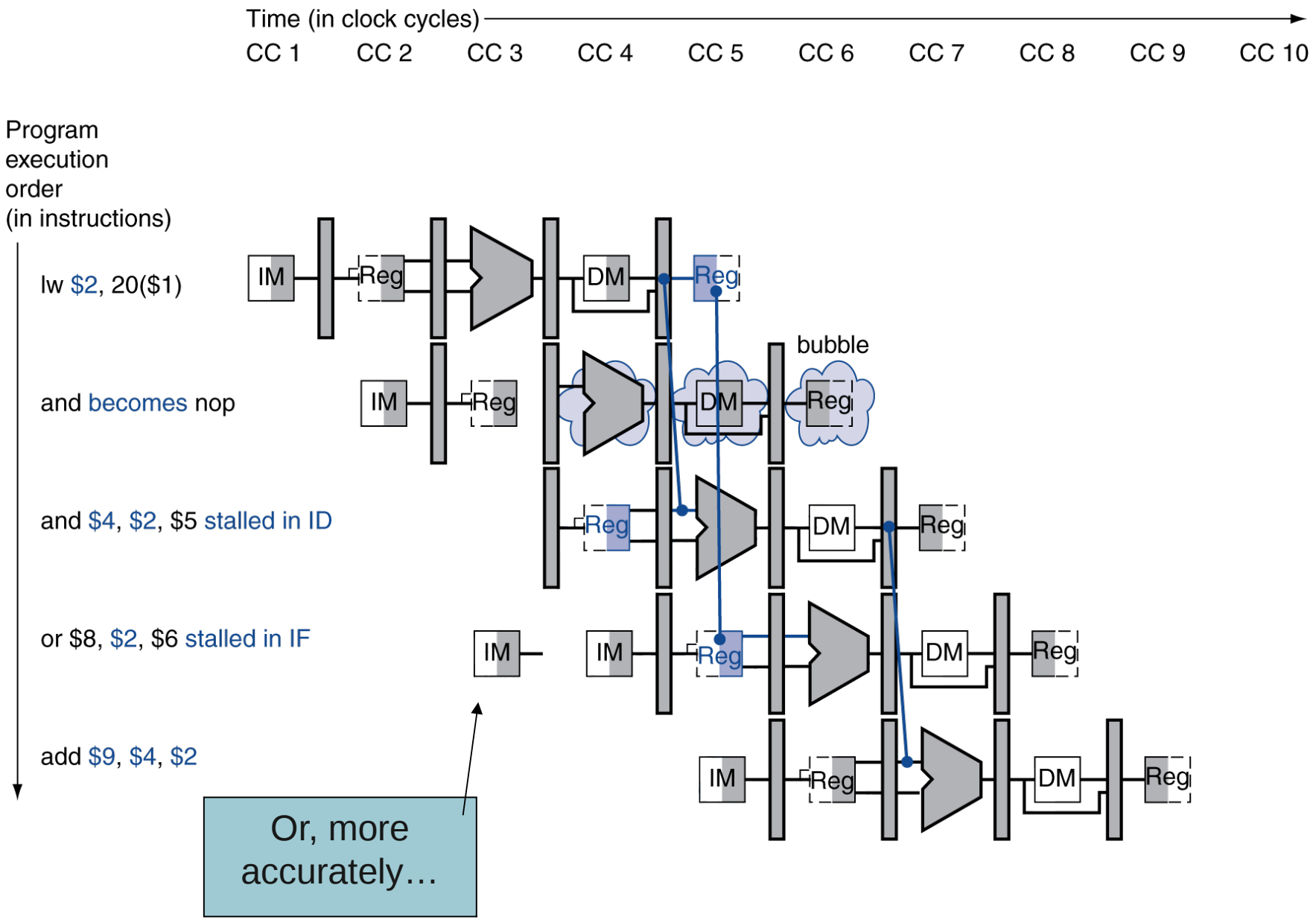
How to Stall the Pipeline

- 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

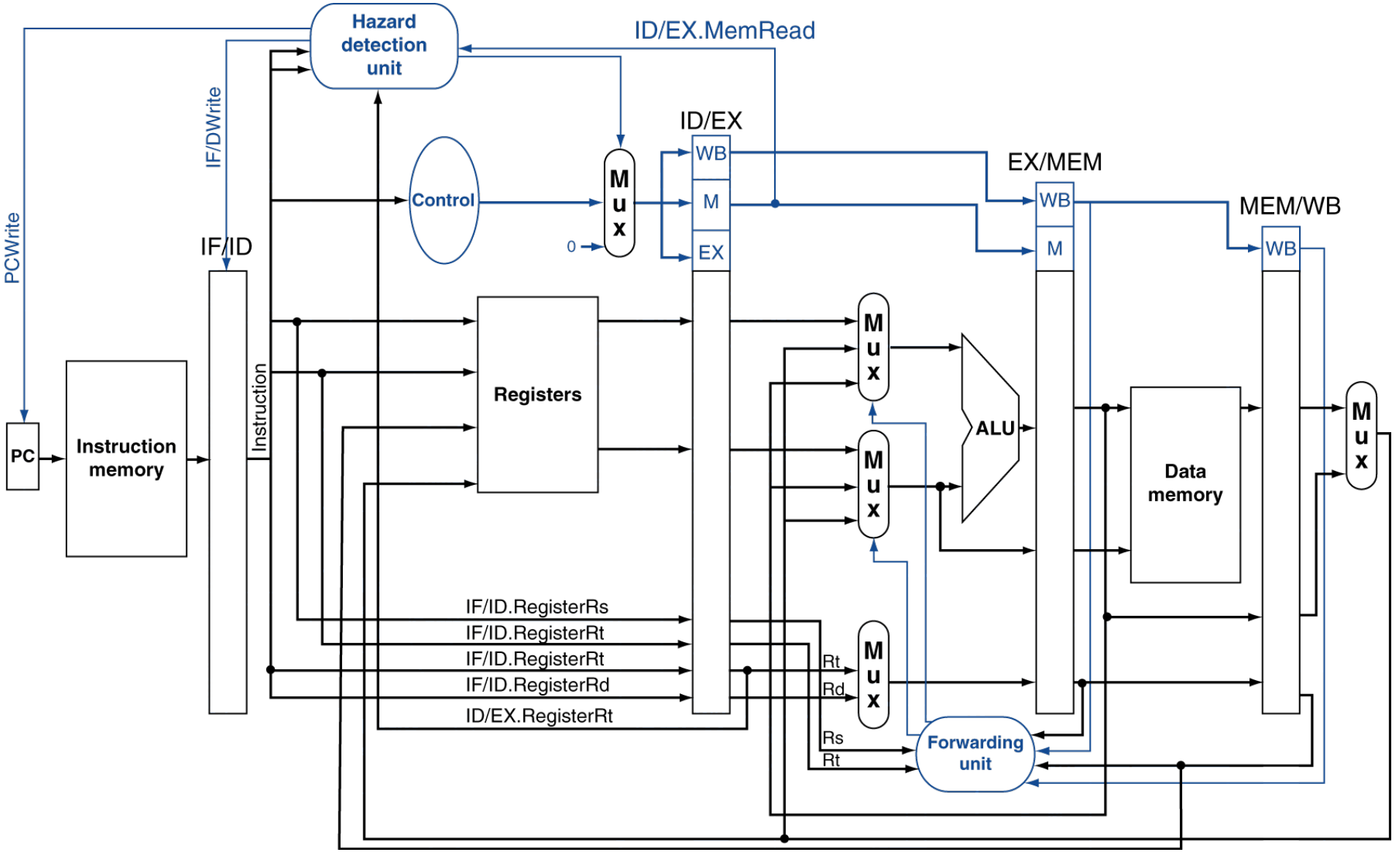
Stall/Bubble in the Pipeline



Stall/Bubble in the Pipeline



Datapath with Hazard Detection



Stalls and Performance

The BIG Picture

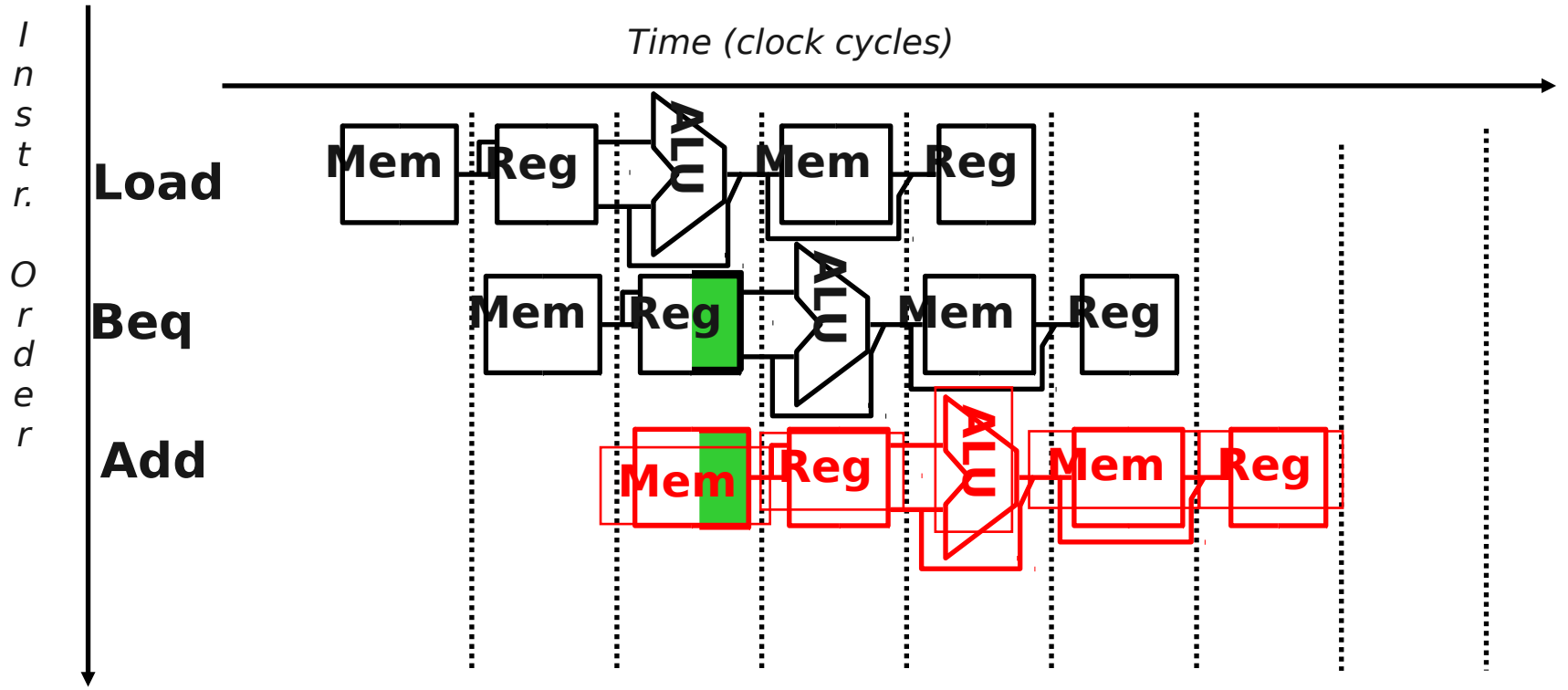
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Control Hazards

Control hazards: attempt to make a decision before condition is evaluated

- E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
- branch instructions

Control Hazards

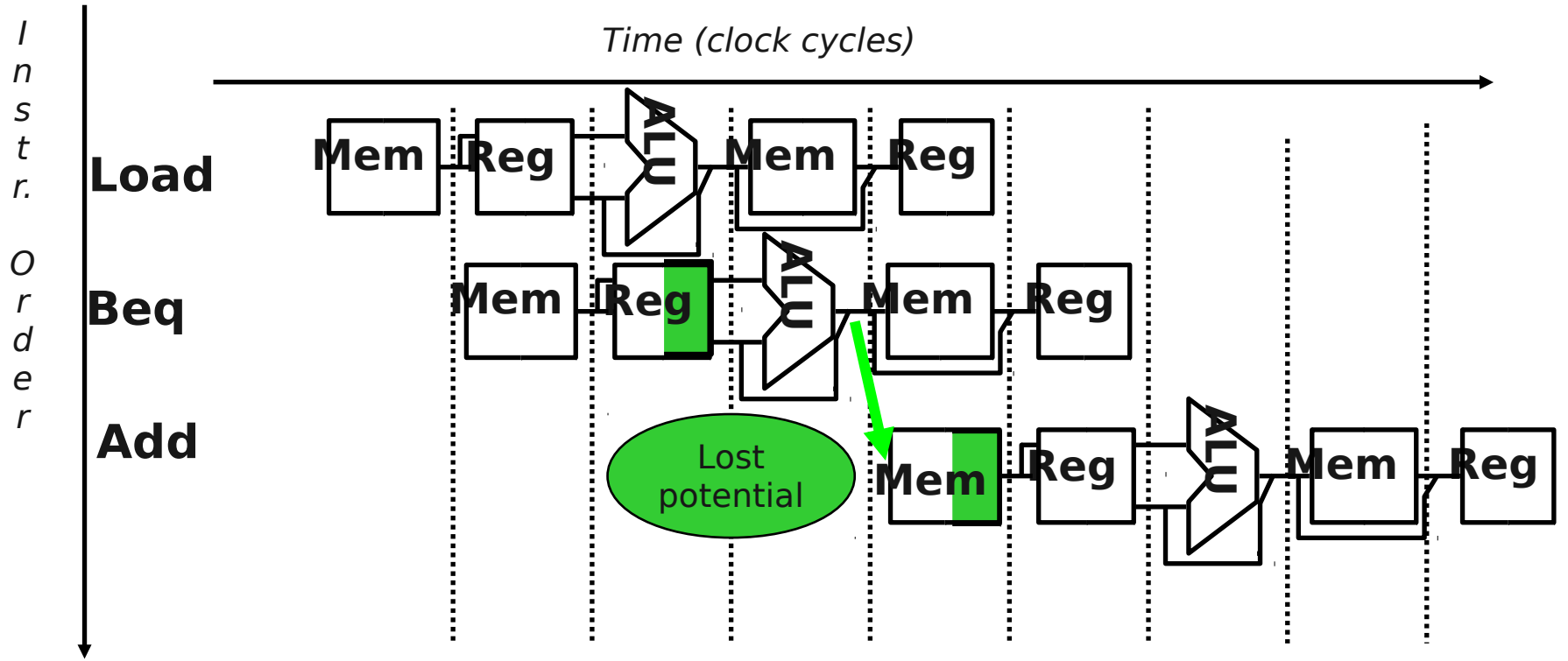


BEQ instruction: Decides which instruction to run next

We don't know the outcome of the BEQ until ALU is done

What do we fetch?

Control Hazard Solution #1: Stall



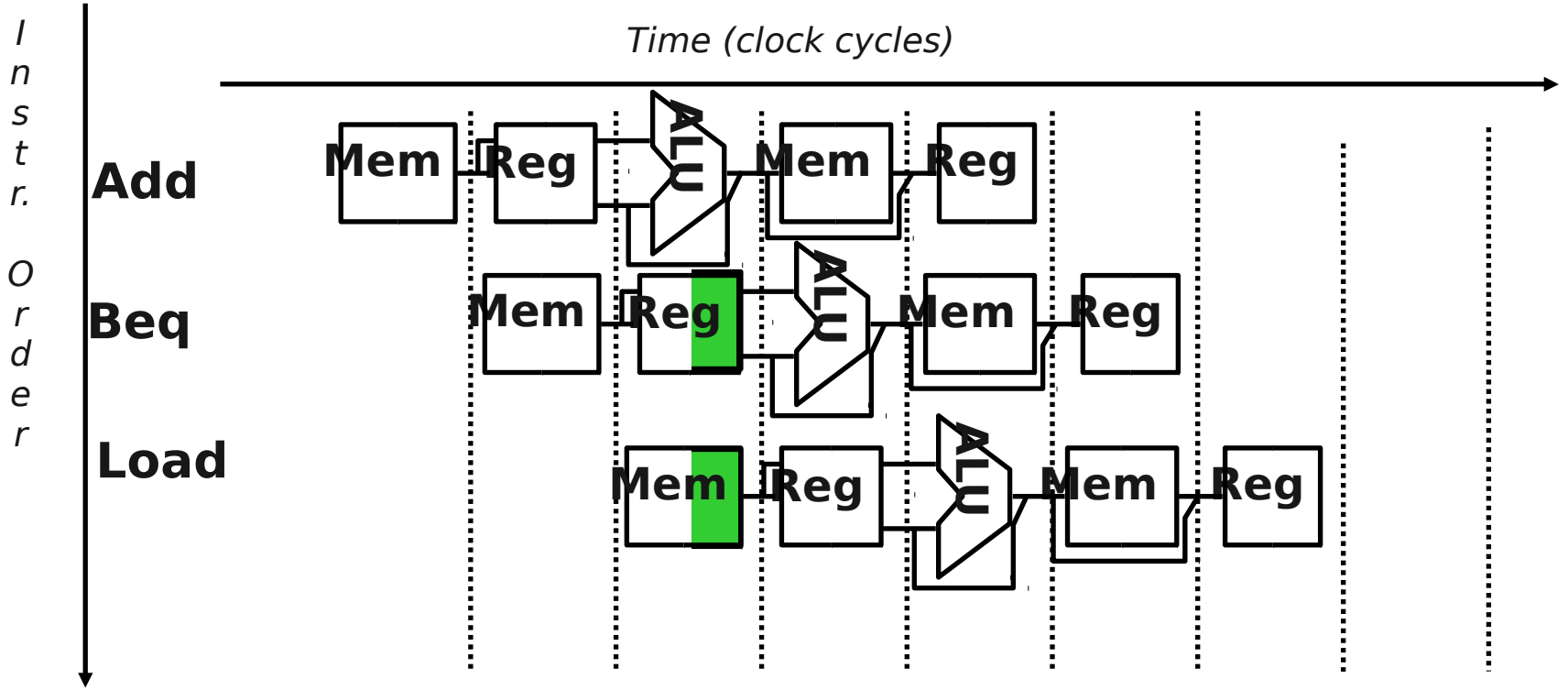
Stall: wait until decision is clear

Impact: 2 lost cycles (i.e. 3 clock cycles for Beq instruction above) => slow

Move decision to end of decode

- save 1 cycle per branch, may stretch clock cycle

Control Hazard Solution #2: Predict



Predict: guess one direction then back up if wrong

Impact: 0 lost cycles per branch instruction if guess right, 1 if wrong (right ~ 50% of time)

- Need to “Squash” and restart following instruction if wrong
- Produce CPI on branch of (1 *.5 + 2 * .5) = 1.5
- Total CPI might then be: 1.5 * .2 + 1 * .8 = 1.1 (20% branch)

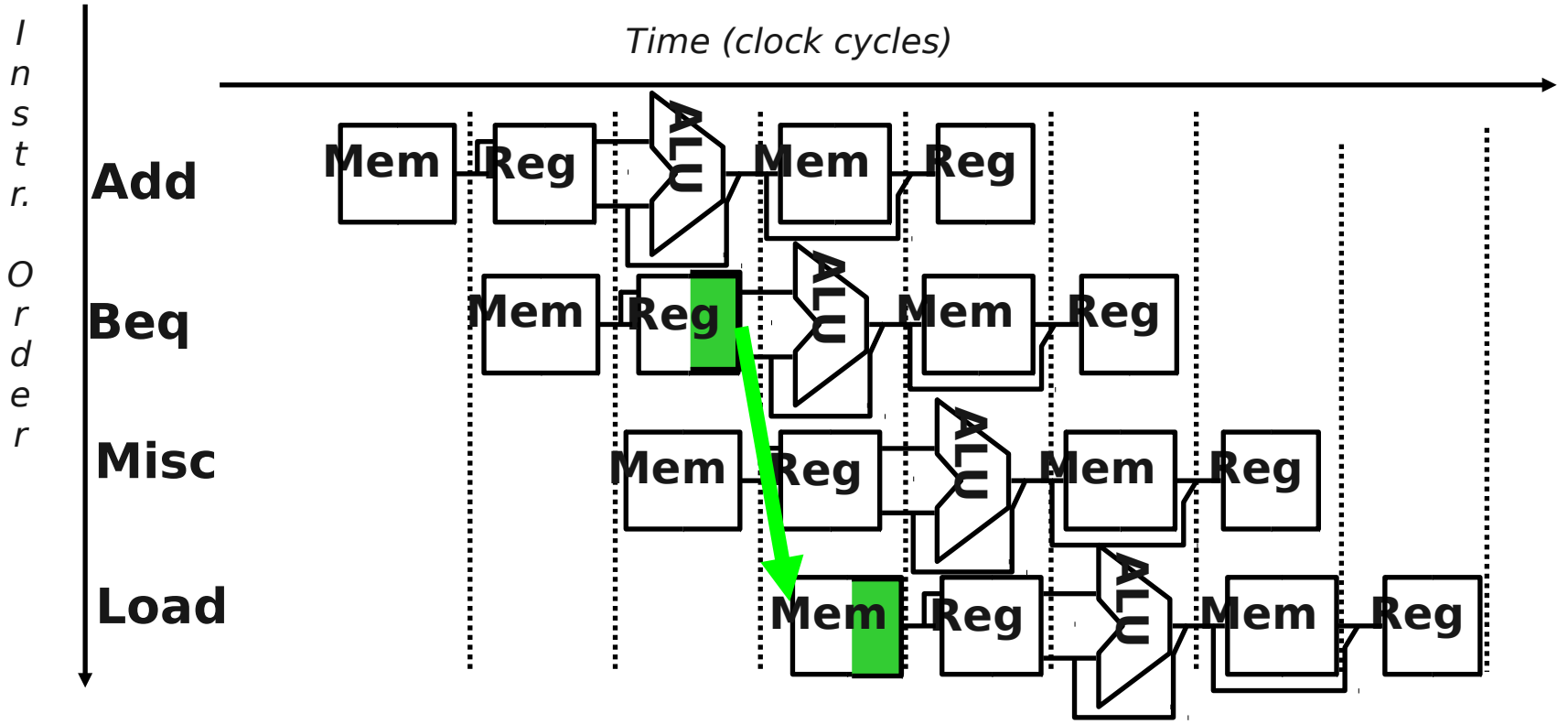
More dynamic schemes: history of branch behavior (~90-99%)

Branch Prediction

Goal: Predict which way the branch will go

- Easiest: Forward branches not taken, backward branches taken (why?)
  - How does hardware know?
- More complex: Keep track of previous behavior, predict based on that (> 90% accuracy)
- Alternate: Let programmer (compiler) specify

Control Hazard Solution #3: Delayed Branch



Delayed Branch: Redefine branch behavior (takes place after next instruction)

Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (~50% of time)

As launch more instruction per clock cycle, less useful

MIPS and Delayed Branch

BEQ (Branch If Equal):

Description: A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs and the contents of general register rt are compared. If the two registers are equal, then the program branches to the target address, *with a delay of one instruction.*

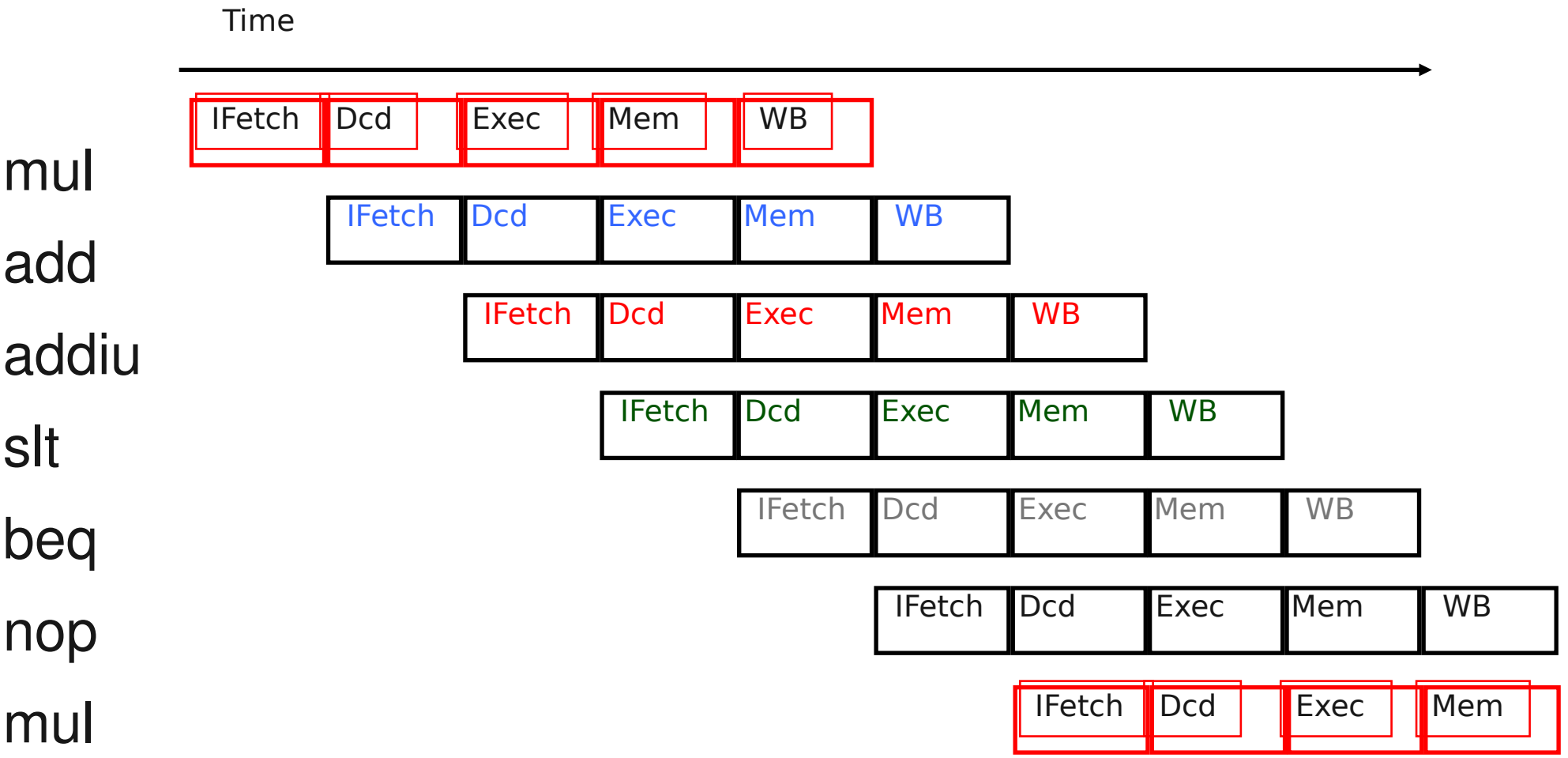
Consider following code:

```
for (i=0, s=0 ; i < 100 ; i++) {
    s += i*i;
  }
```

```
top: mul temp, i, i  # pretend OK
     add s, s, temp
     addiu i, i, 1
     sltiu b, i, 100
     beq b, 0, top
      nop          # doesn't count as an instr
```

How many cycles to run this? CPI? How to make it faster?

Unmodified Code



6 cycles/iter * 100 iter = 600 cycles (+drain)

CPI = 600 cycles / 500 instrs = 1.2

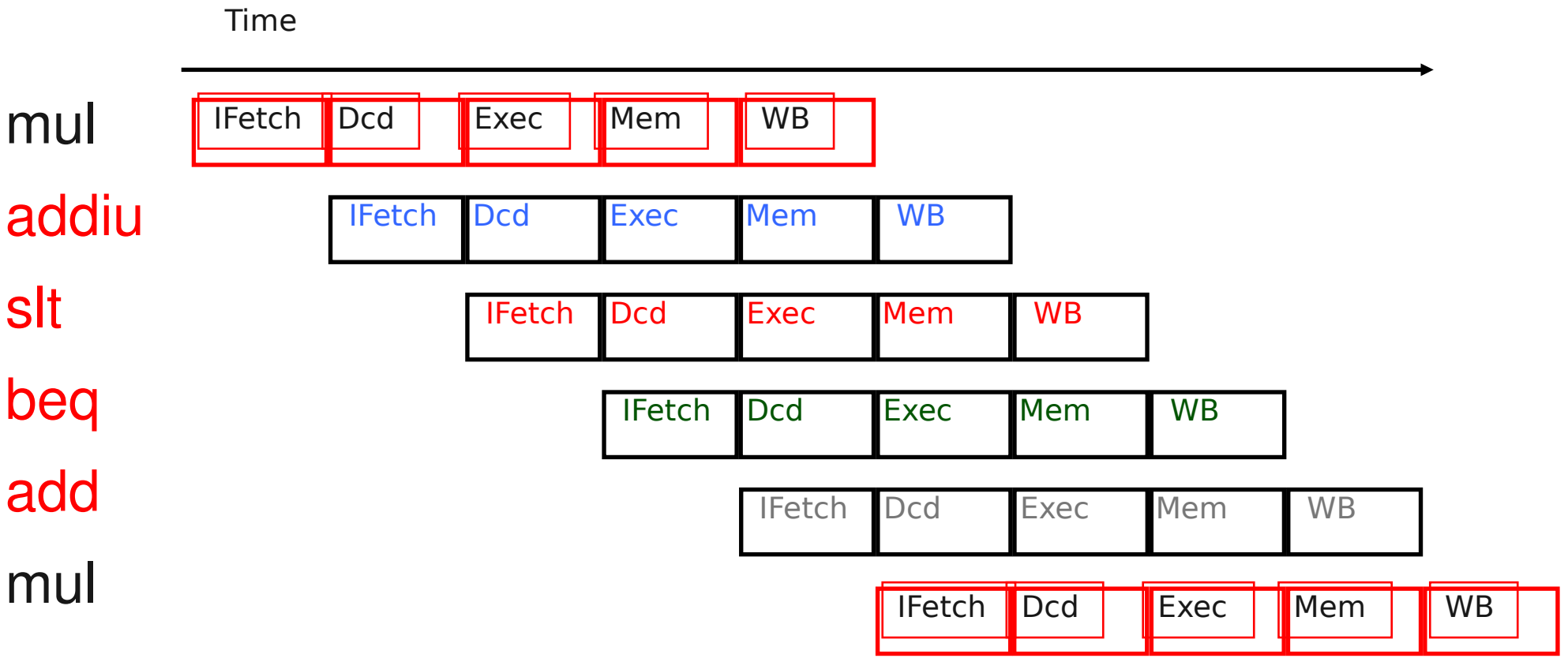
Rewrite:

```
for (i = 0, s = 0 ; i < 100 ; i++) {
    s += i*i;
  }
```

```
top:    mul temp, i, i  # pretend OK
        addiu i, i, 1
        sltiu b, i, 100
        beq b, 0, top
        add s, s, temp
```

How many cycles to run this? How to make it faster?

After Rewrite



5 cycles/iter * 100 iter = 500 cycles

CPI = 500 cycles / 500 instrs = 1.0

Loop Unrolling

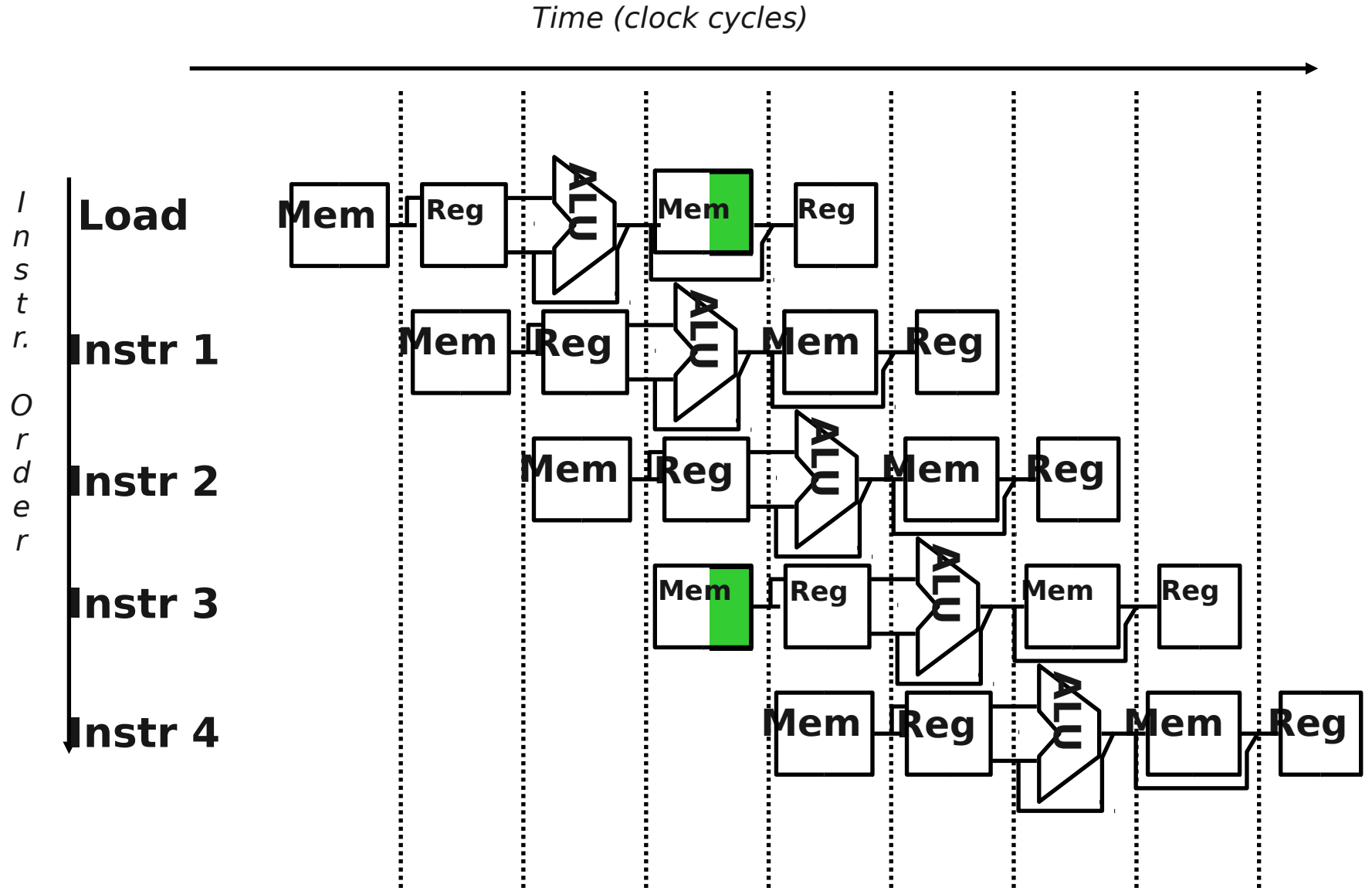
```
for (i=0, s=0 ; i < 100 ; i++) {
     s += i*i; i++;
     s += i*i;
  }
```

```
top:      mul temp, i, i  # pretend OK
          add s, s, temp
          addiu i, i, 1
          mul temp, i, i
          addiu i, i, 1
          sltiu b, i, 100
          beq b, 0, top
          add s, s, temp
```

How many cycles to run this? Where did we save?

8 cycles/iteration * 50 iterations = 400 cycles; CPI = 1

Single Memory is a Structural Hazard



Detection is easy in this case! **(right half highlight means read, left half write)**

Structural Hazards limit performance

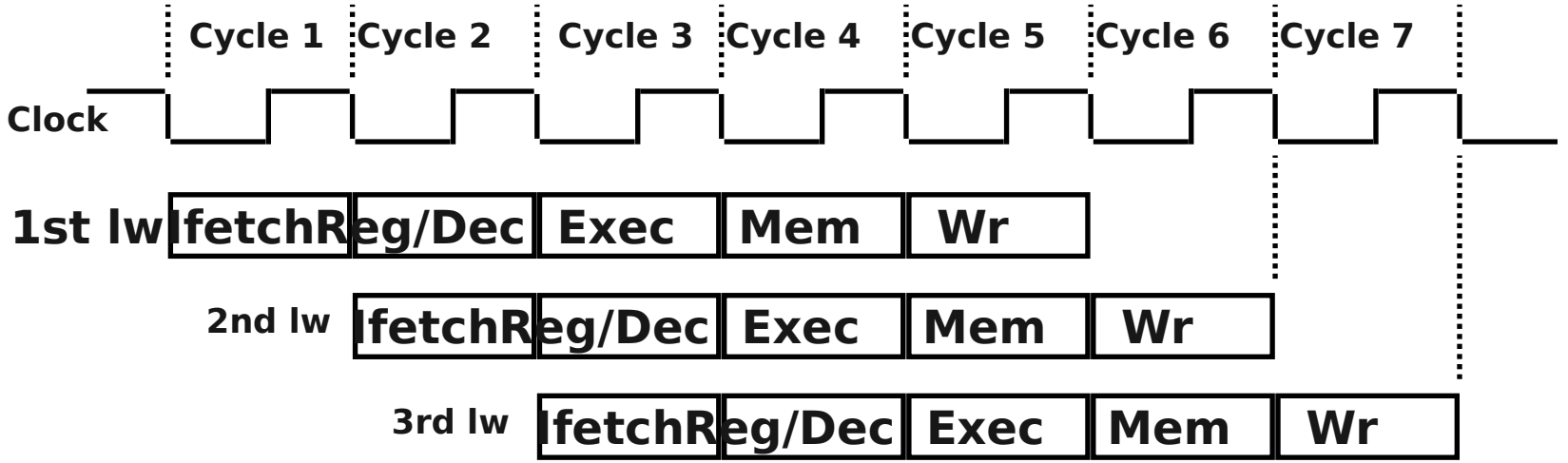
Example: if 1.3 memory accesses per instruction and only one memory access per cycle then

- average CPI ≥ 1.3
- otherwise resource is more than 100% utilized

One Structural Hazard solution: more resources

- Instruction cache and Data cache

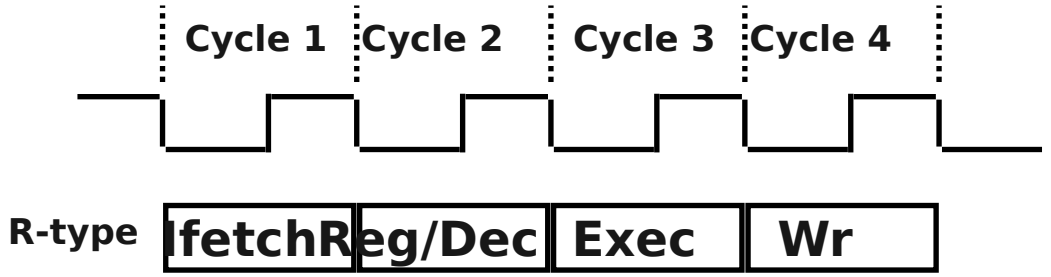
Pipelining the Load Instruction



The five independent functional units in the pipeline datapath are:

- Instruction Memory for the Ifetch stage
- Register File's Read ports (bus A and busB) for the Reg/Dec stage
- ALU for the Exec stage
- Data Memory for the Mem stage
- Register File's Write port (bus W) for the Wr stage

The Four Stages of R-type



Ifetch: Instruction Fetch

- Fetch the instruction from the Instruction Memory

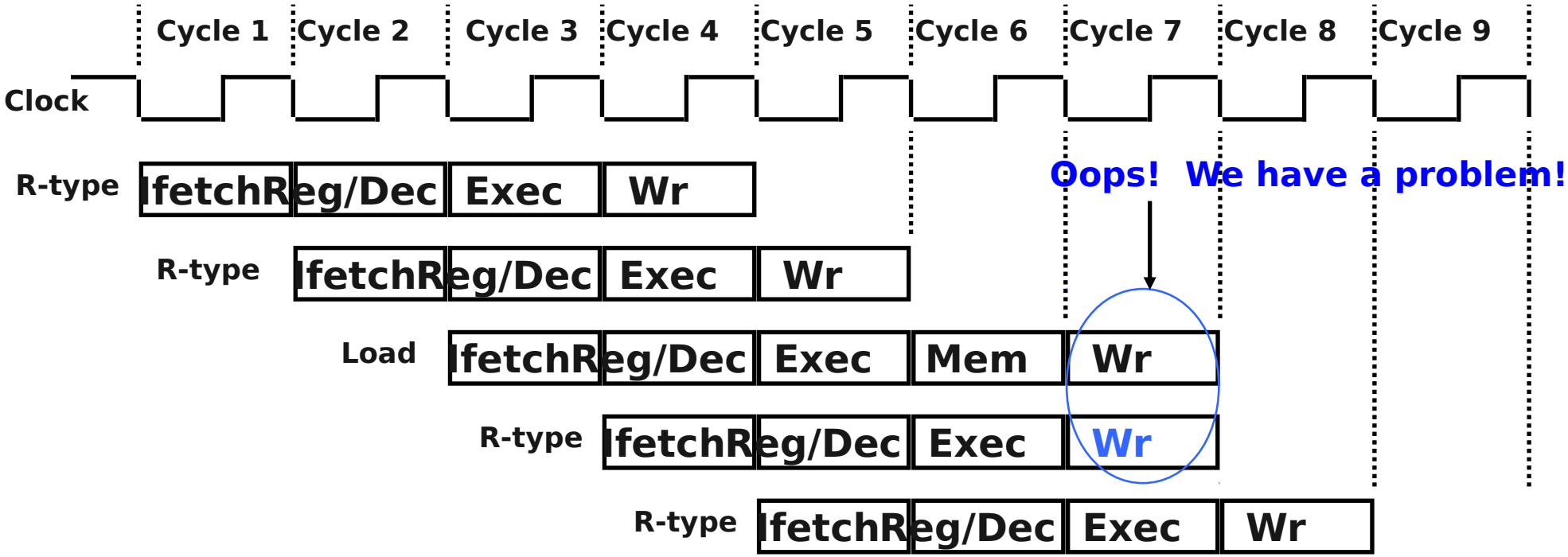
Reg/Dec: Registers Fetch and Instruction Decode

Exec:

- ALU operates on the two register operands
- Update PC

Wr: Write the ALU output back to the register file

Pipelining the R-type and Load Instruction



We have pipeline conflict or structural hazard:

- Two instructions try to write to the register file at the same time!
- Only one write port

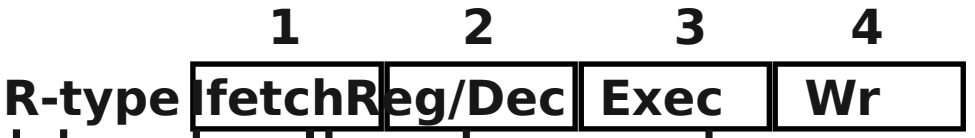
Important Observation

Each functional unit can only be used once per instruction

Each functional unit must be used at the same stage for all instructions:

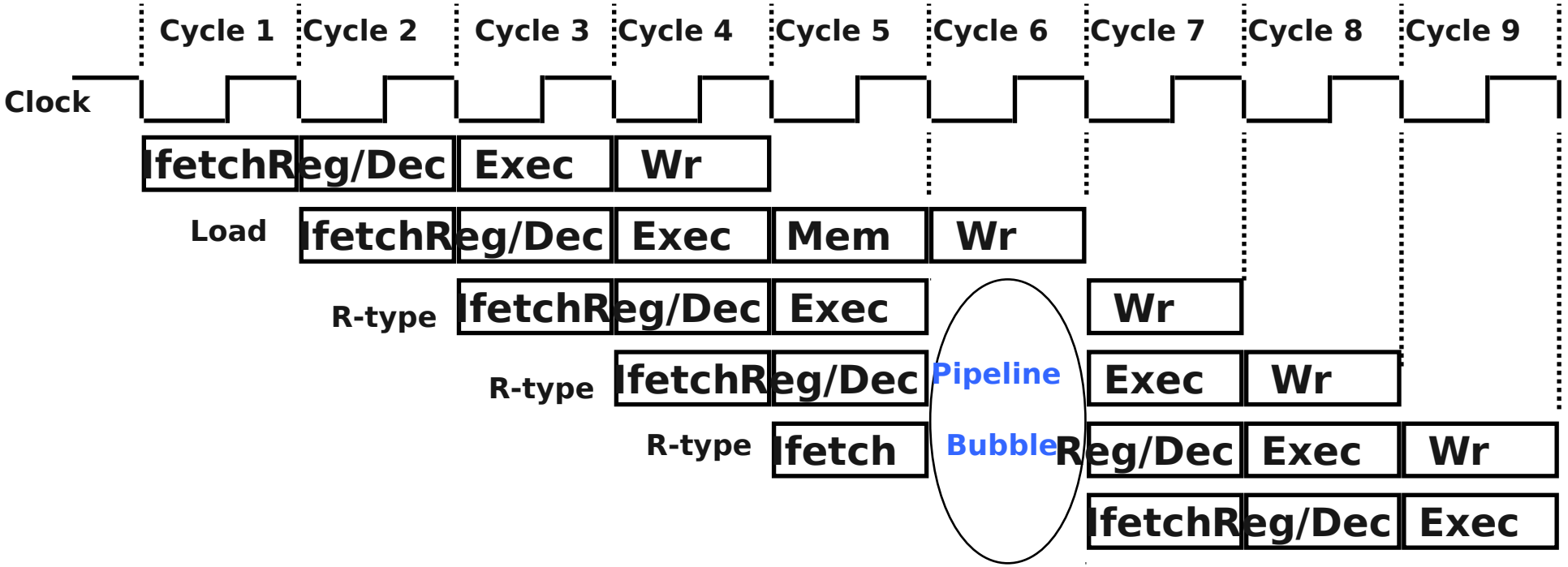
- Load uses Register File's Write Port during its 5th stage

- R-type uses Register File's Write Port during its 4th stage



2 ways to solve this pipeline hazard.

Solution 1: Insert “Bubble” into the Pipeline



Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle

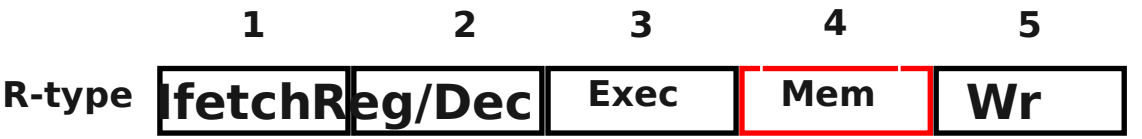
- The control logic can be complex.
- Lose instruction fetch and issue opportunity.

No instruction is started in Cycle 6!

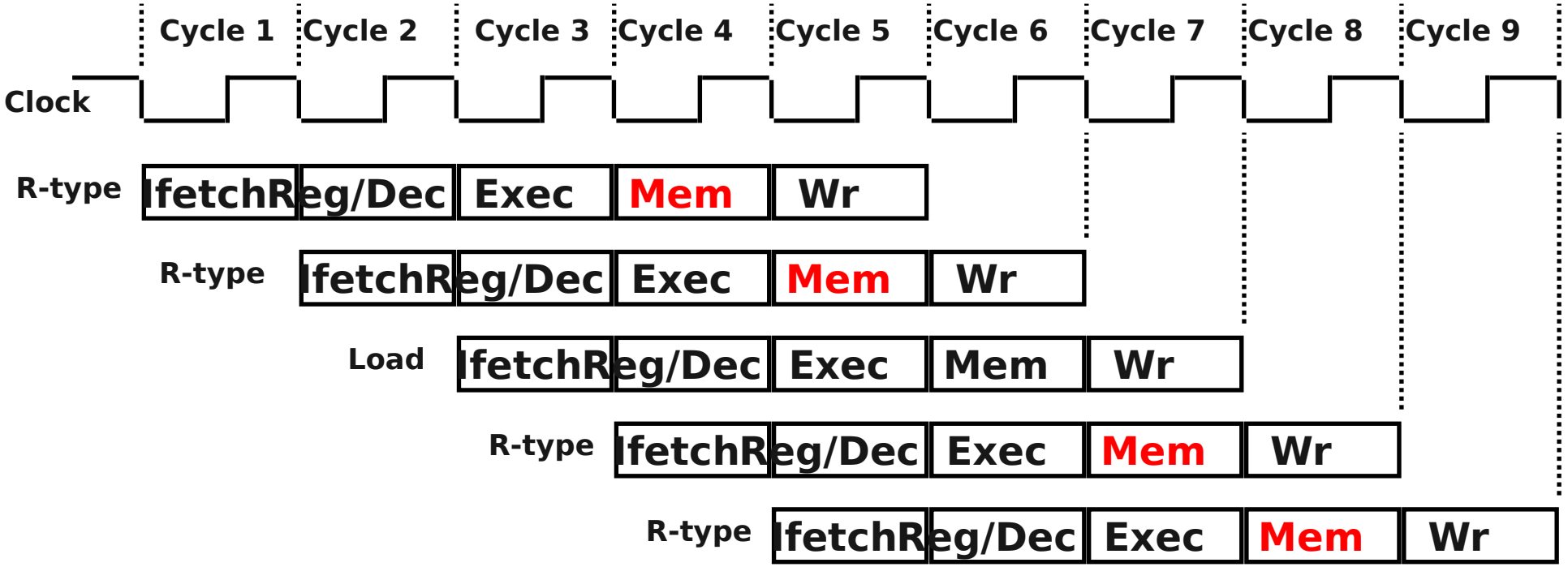
Solution 2: Delay R-type's Write by One Cycle

Delay R-type's register write by one cycle:

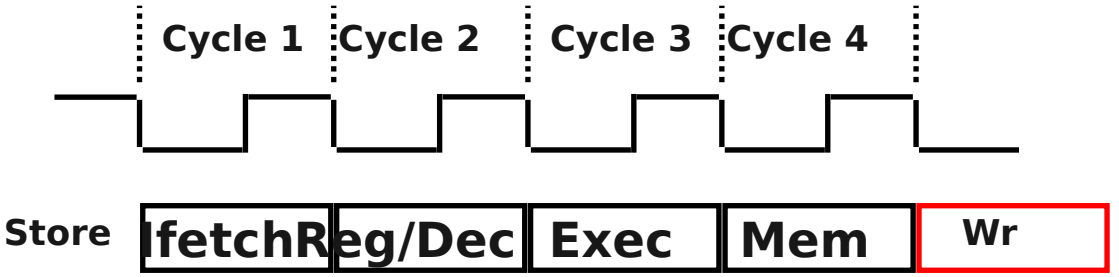
- Now R-type instructions also use Reg File's write port at Stage 5



- Mem stage is a NOP stage: nothing is being done.



The Four Stages of Store => 5 stages



Ifetch: Instruction Fetch

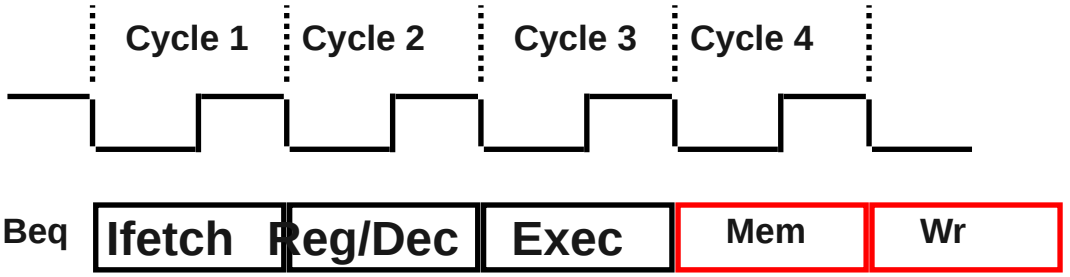
- Fetch the instruction from the Instruction Memory

Reg/Dec: Register Fetch and Instruction Decode

Exec: Calculate the memory address

Mem: Write the data into the Data Memory

The Three Stages of Beq => 5 stages



Ifetch: Instruction Fetch

- Fetch the instruction from the Instruction Memory

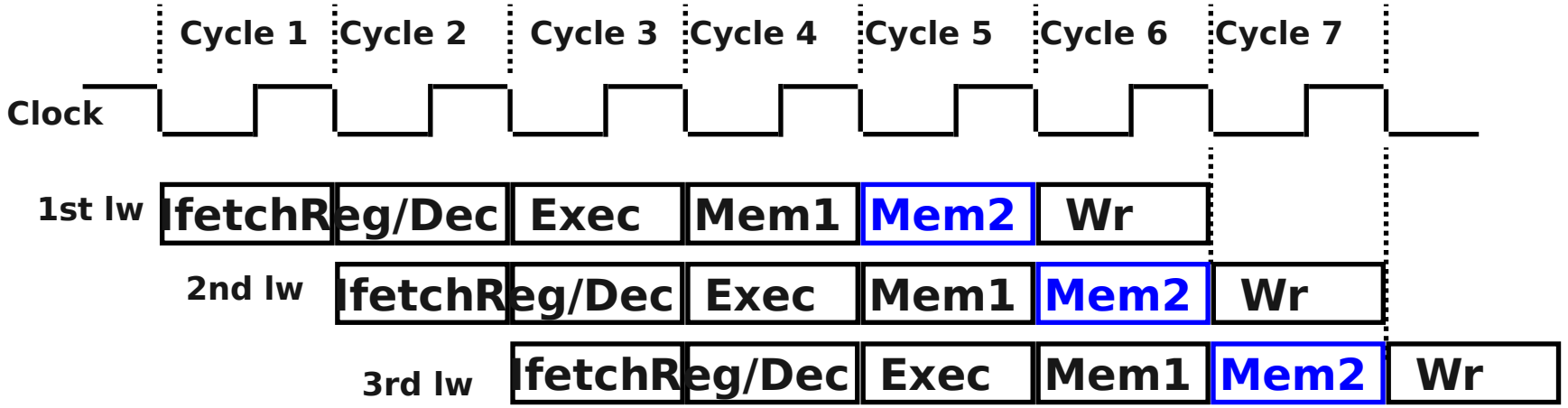
Reg/Dec:

- Registers Fetch and Instruction Decode

Exec:

- compares the two register operand,
- select correct branch target address
- latch into PC

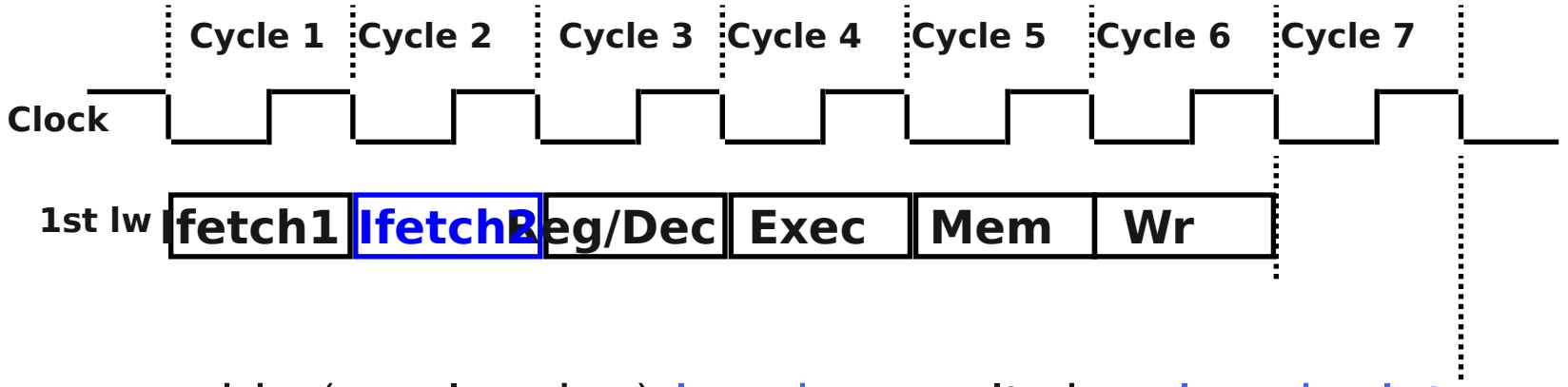
Problem #1



Suppose a big (overlapping) data cache results in a data cache latency of 2 clock cycles and a 6-stage pipeline. What is the impact?

1. Instruction bandwidth is now 5/6-ths of the 5-stage pipeline
2. Instruction bandwidth is now 1/2 of the 5-stage pipeline
3. The branch delay slot is now 2 instructions
4. The load-use hazard can be with 2 instructions following load
5. Both 3 and 4: branch delay and load-use now 2 instructions
6. None of the above

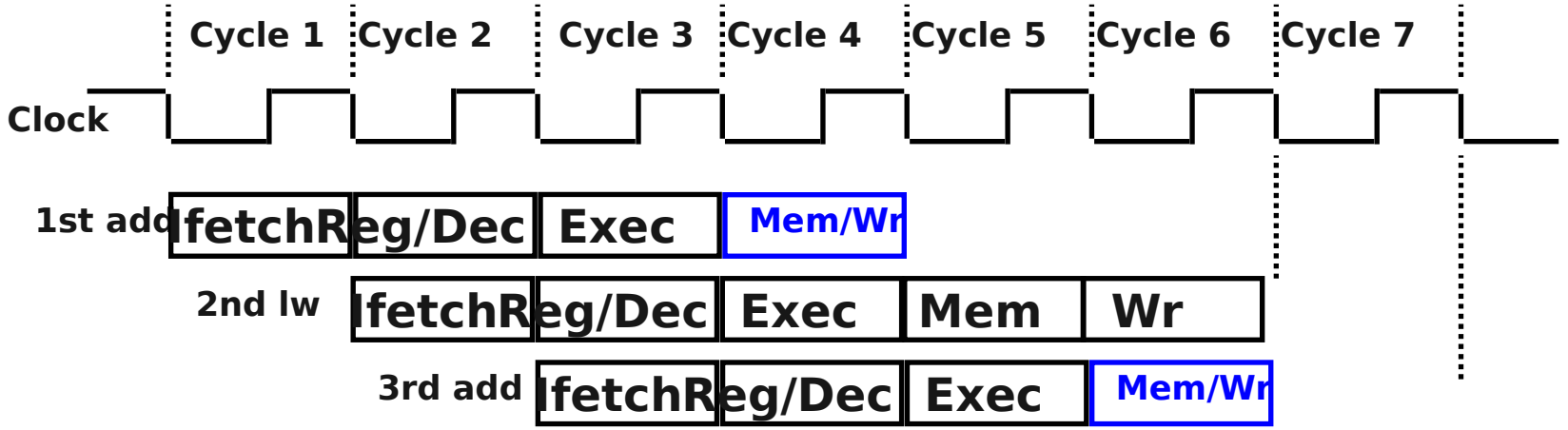
Problem #2



Suppose a big (overlapping) I cache results in a I cache latency of 2 clock cycles and a 6-stage pipeline. What is the impact?

1. Instruction bandwidth is now 5/6-ths of the 5-stage pipeline
2. Instruction bandwidth is now 1/2 of the 5-stage pipeline
3. The branch delay slot is now 2 instructions
4. The load-use hazard can be with 2 instructions following load
5. Both 3 and 4: branch delay and load-use now 2 instructions
6. None of the above

Problem #3



Suppose we use with a 4 stage pipeline that combines memory access and write back stages for all instructions but load, stalling when there are structural hazards. Impact?

1. The branch delay slot is now 0 instructions
2. Instr after every load stalls since it has a structural hazard
3. Every store stalls since it has a structural hazard
4. Both 2 & 3: instr-after-load & stores stall due to structural hazards
5. Every instr-after-load stalls, but there is no load-use hazard anymore
6. Both 2 & 3, but there is no load-use hazard anymore
7. None of the above

Designing a Pipelined Processor

- Go back and examine your datapath and control diagram
- Associate resources with states
- Ensure that backwards flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

Summary: Pipelining

Reduce CPI by overlapping many instructions

- Average throughput of approximately 1 CPI with fast clock

Utilize capabilities of the Datapath

- start next instruction while working on the current one
- limited by length of longest stage (plus fill/flush)
- detect and resolve hazards

What makes it easy

- all instructions are the same length
- just a few instruction formats
- memory operands appear only in loads and stores

What makes it hard?

- structural hazards: suppose we had only one memory
- control hazards: need to worry about branch instructions
- data hazards: an instruction depends on a previous instruction