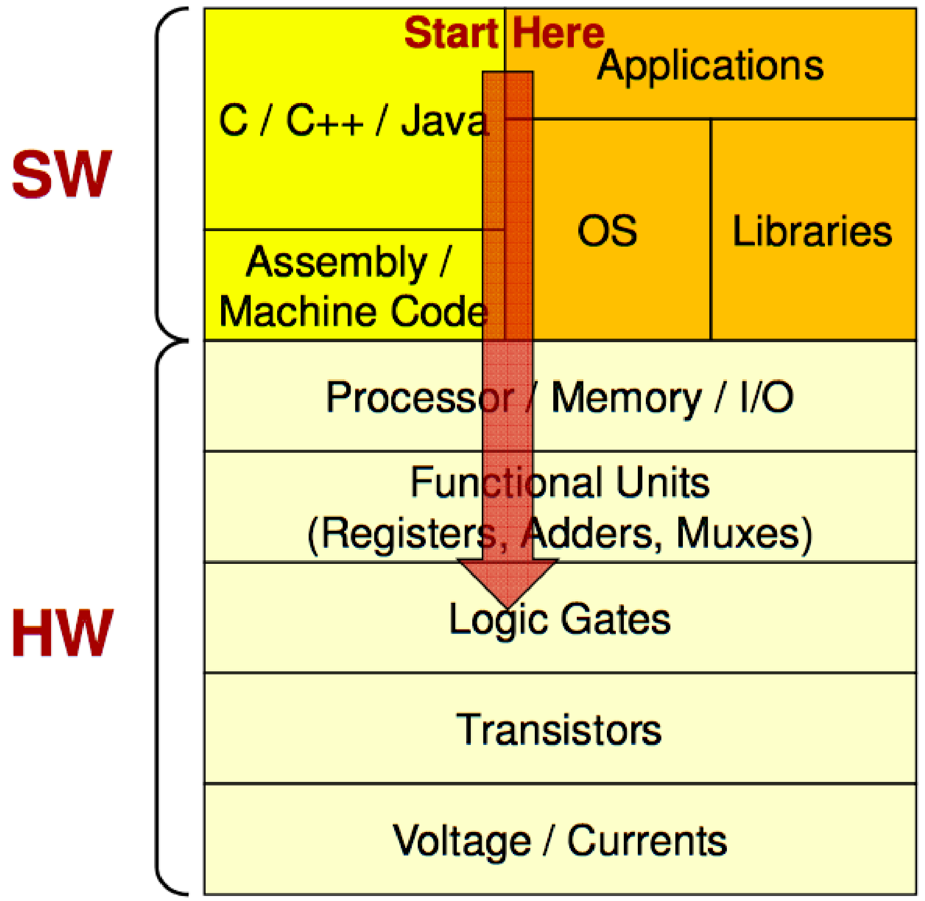
CS6L047: Advanced Computer Architecture

Pipelining [Appendix C]

PIPELINING REVIEW

Software-Hardware stack



Credit:: Mark Redekopp, USC

ISA example - MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)
- Used only for ALU instructions

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $r8, $r17, $r18
```

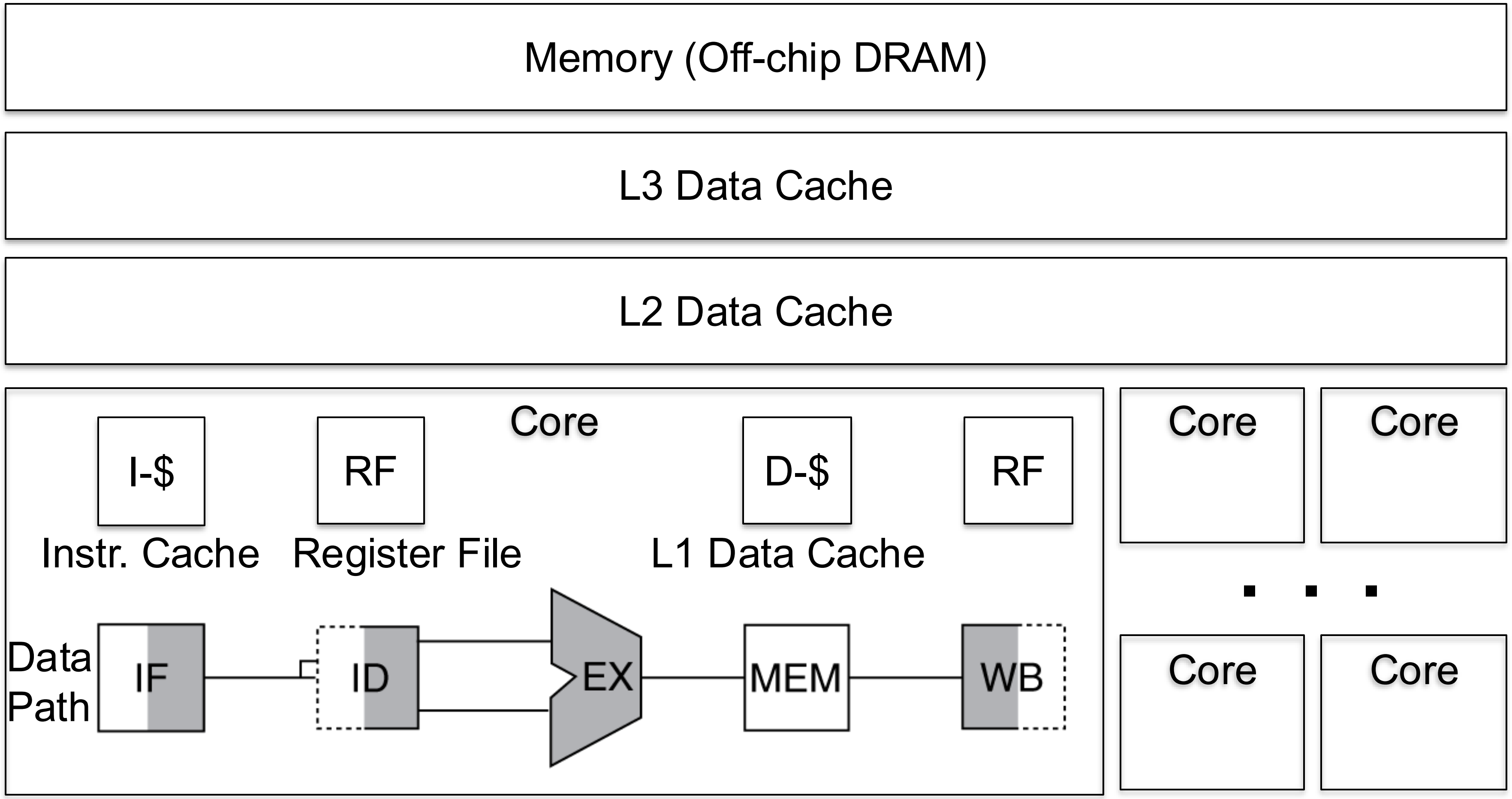
special	$r17	$r18	$r8	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

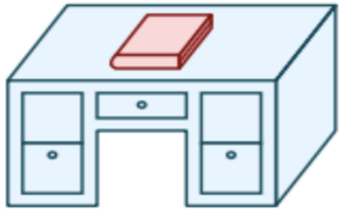
$00000010001100100100000000100000_2 = 02324020_{16}$

Abstracted CPU



Cache/Bookshelf Analogy

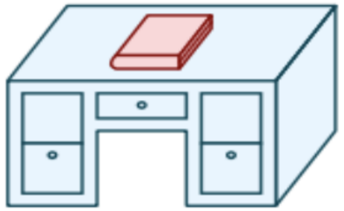
Desk
(can read one book)



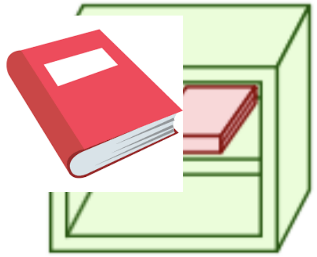
Library
(can hold many books)

Problem: Far, Long Access time

Desk
(can read one book)



Book Shelf
(can hold few books)

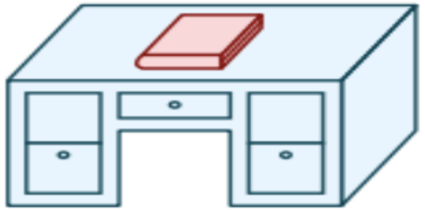


Library
(can hold many books)

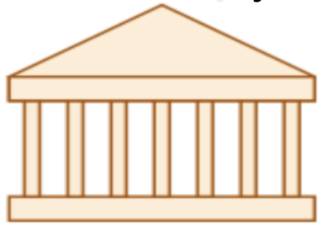
Solution: Keep few frequently accessed items close to desk.

Cache/Bookshelf Analogy

Desk
(can read one book)



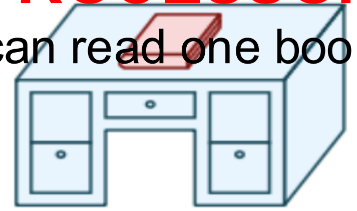
Library
(can hold many books)



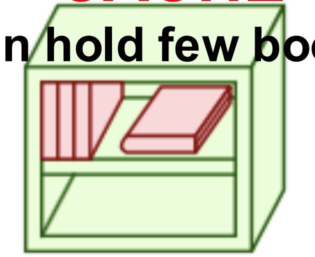
Processor = Desk
Cache = Book Shelf
Mem = Library

Problem: Far, Long Access time

Desk = **PROCESSOR**
(can read one book)

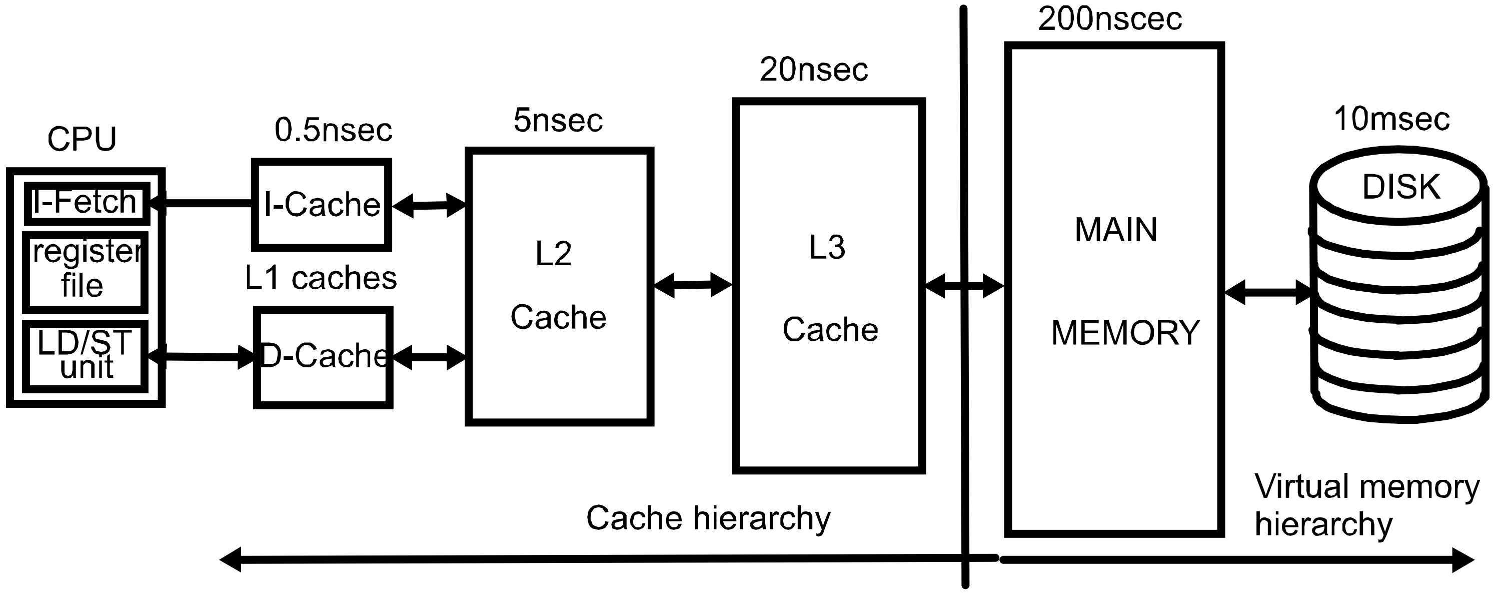


Book Shelf = CACHE
(can hold few books)



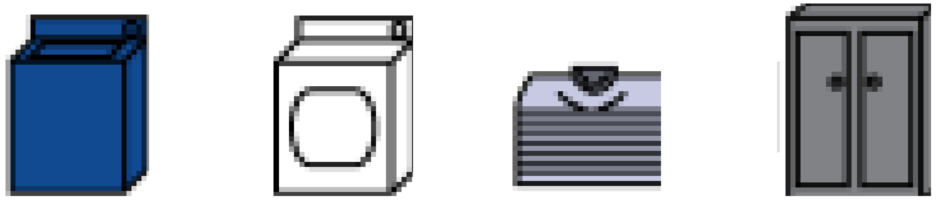
Library = **MEMORY**
(can hold many books)

Memory Hierarchy

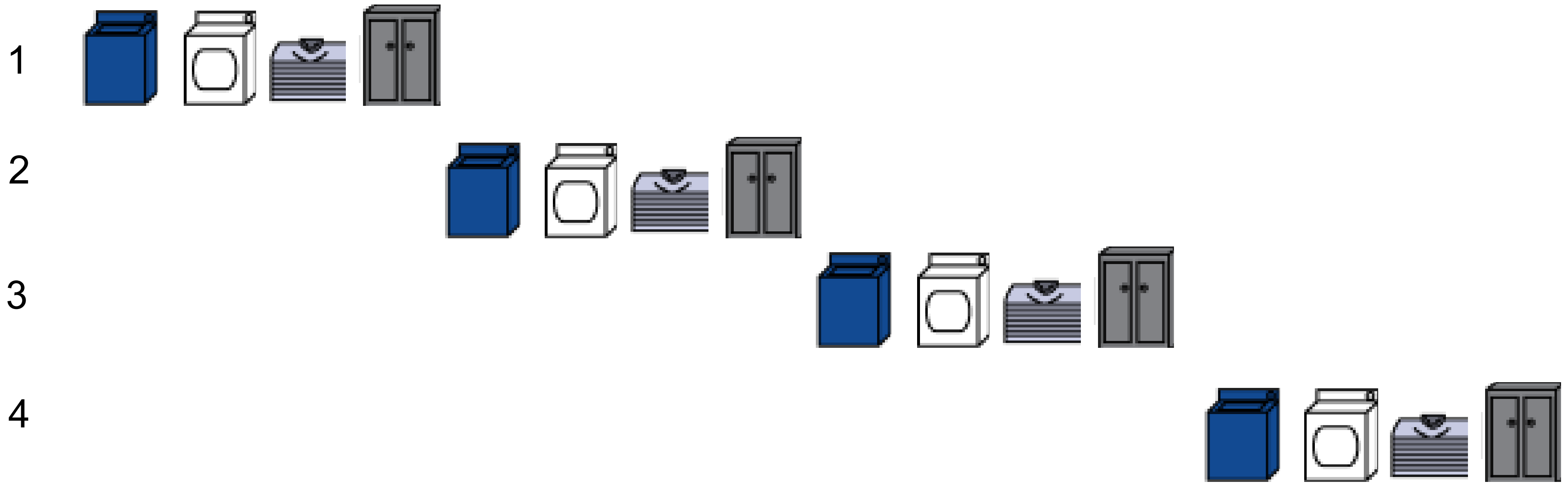


Pipelining Analogy

- Laundry steps:
  - Wash
  - Dry
  - Fold
  - Put it away – Closet / Dresser / Neat pile on floor



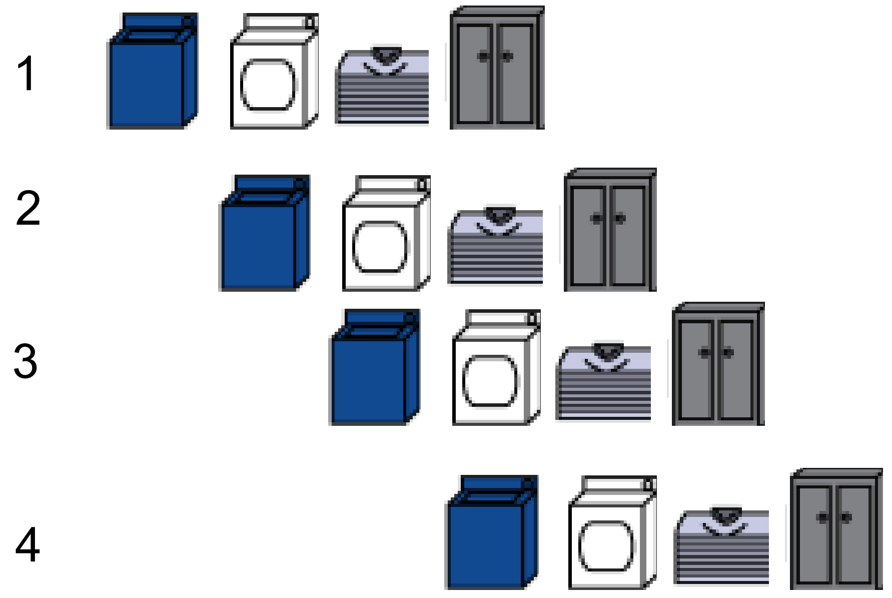
Pipelining Analogy



Assuming each step take 1 hour,
4 loads would take 16 hours!

Pipelining Analogy

- To speed things up, overlap steps



- 4 loads of laundry now only takes 7 hours!

Speedup of Pipelining

- *k* stages pipeline, *t* time per stage, *n* jobs
- Non-pipelined time = n*k*t
- Pipelined time = (k+n-1)*t

$$Speedup = \frac{n \times k \times t}{(k+n-1) \times t} \xrightarrow[n \to \infty]{} k$$

This is an ideal case:

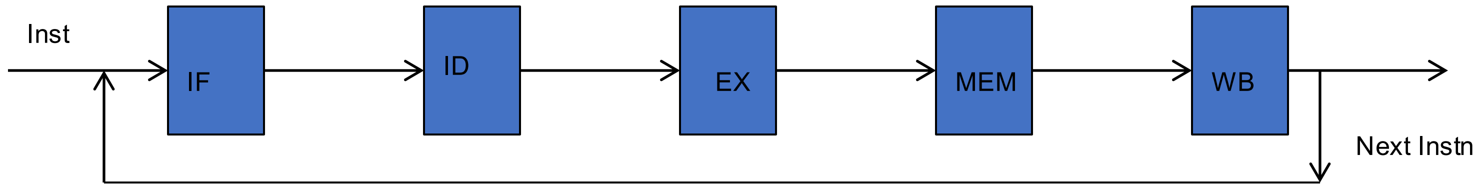
- No job depends on a previous job
- All jobs behave exactly the same

Not realistic.

SIMPLE 5 STAGE PIPELINE

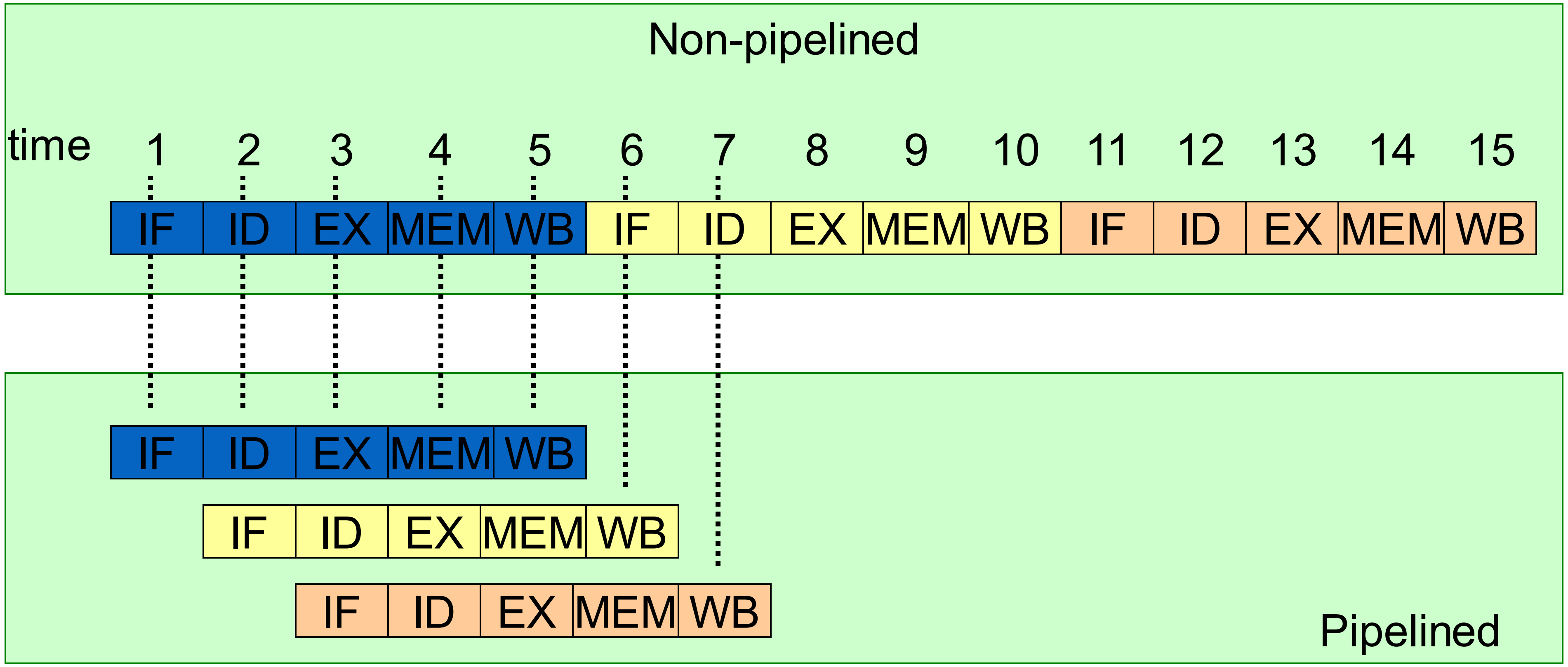
Classic 5-stage Execution of a RISC Procesor

- 5 canonical stage “RISC” load-store architecture
  1. Instruction fetch (IF):
     - get instruction from memory/cache
  2. Instruction decode, Register read (ID):
     - translate opcode into control signals and read regs
  3. Execute (EX):
     - perform ALU operation, load/store address, branch outcomes
  4. Memory (MEM):
     - access memory if load/store, everyone else idle
  5. Writeback/retire (WB):
     - write results to register file



Pipeline Execution

- Overlap execution of instructions
  - Start instruction on **every** cycle, e.g. the new instruction can be fetched while the previous one is decoded – ***pipeline***. Each cycle performing a specific task; number of stages is called pipeline depth (5 here)



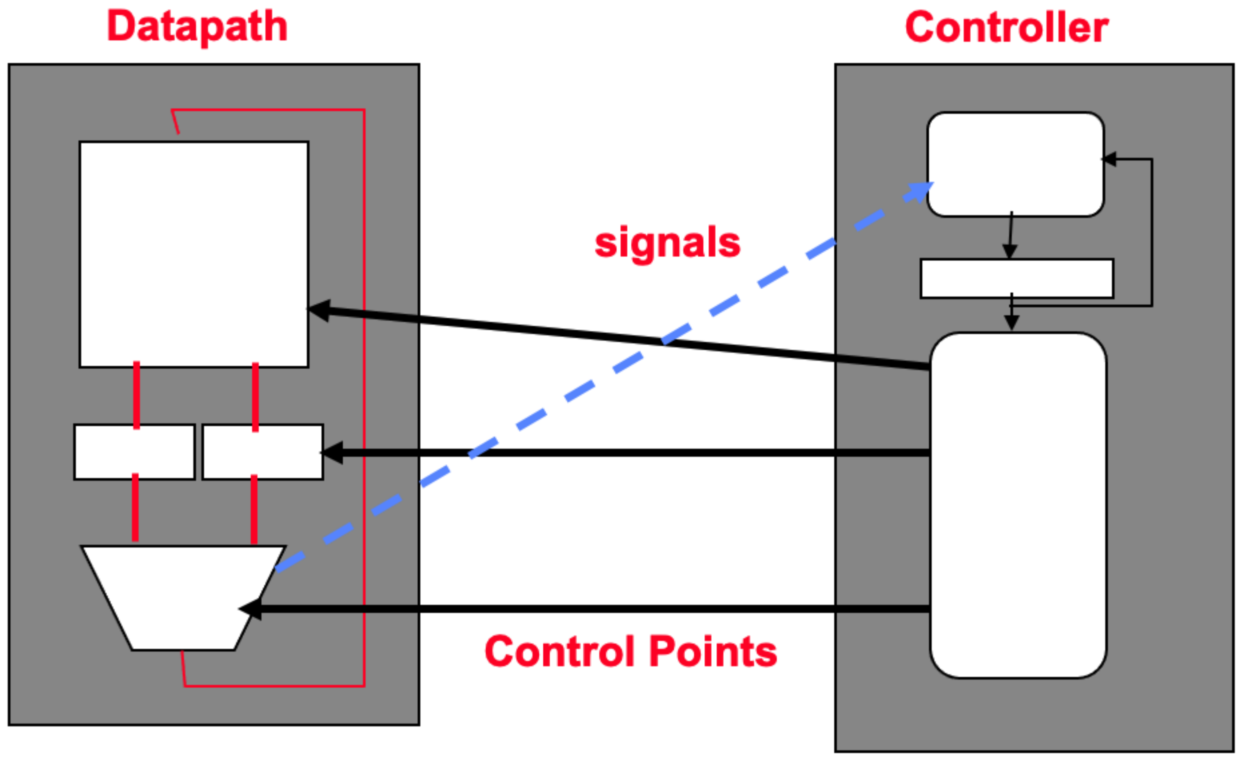
ALU Functions by Instruction Type

Instruction Type	ALU Function
Memory Reference	Adds the **base register** and **offset** to compute the **effective address**.
Register-Register ALU	Executes the operation defined by the **ALU opcode** on two values from the **register file**.
Register-Immediate ALU	Performs the operation using one **register value** and a **sign-extended immediate** value.
Conditional Branch	Evaluates the condition to decide whether to **branch** or continue sequential execution.

The ALU operates on operands that were prepared in the **previous clock cycle**.

In a **load-store architecture**, the **effective address calculation** and **execution** can occur in the **same cycle**, since no instruction needs to compute a data address and manipulate data simultaneously.

Datapath vs Control

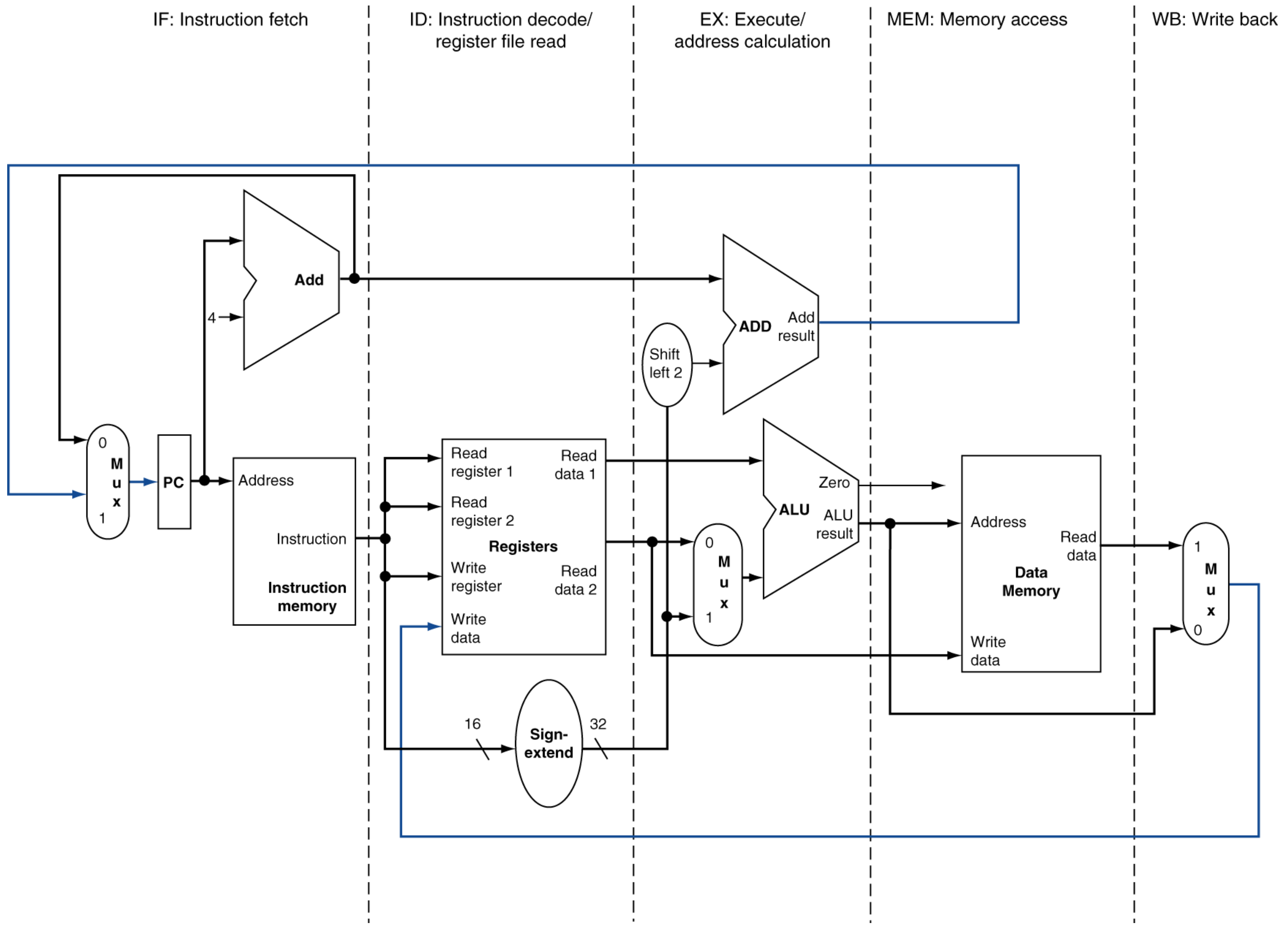


Component	Role	Inputs	Outputs
Datapath	Executes operations	Control signals	Computation results
Controller	Directs datapath behavior	Instruction & status	Control signals

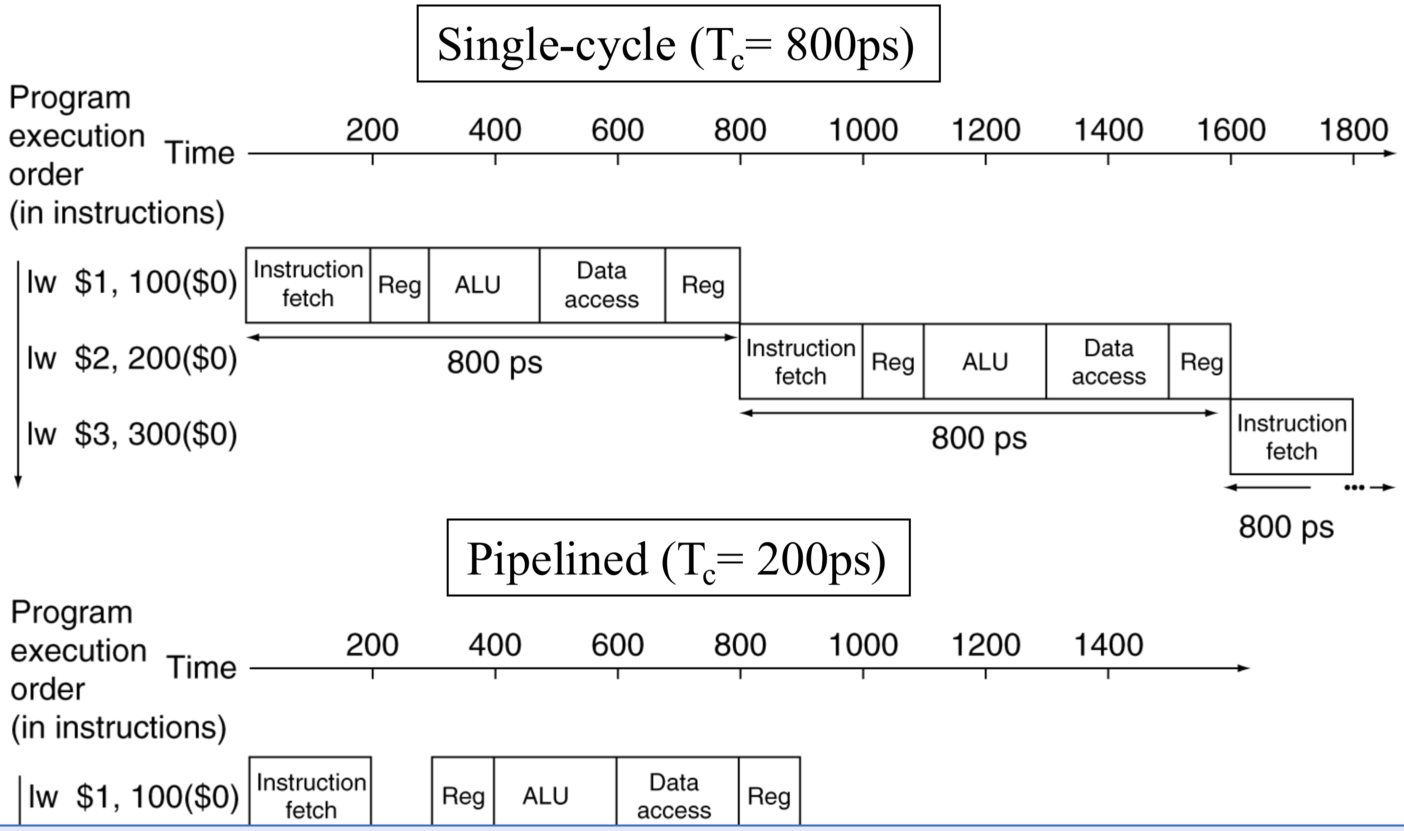
Assignment: Data and control Path recap of

```
add $r8, $r17, $r18
```

5-stage pipeline



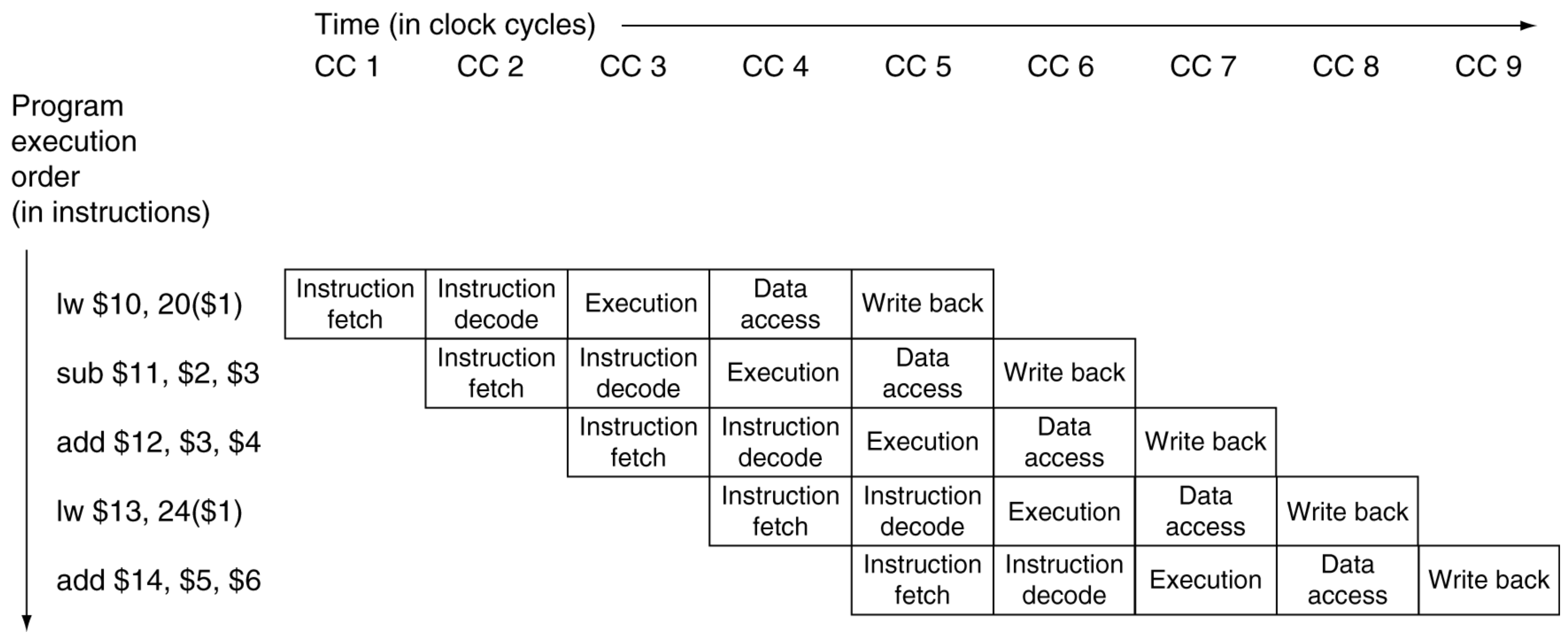
Pipeline Performance



What is the Pipeline Width here and how does it impact the overall pipeline performance?

Multi-Cycle Pipeline Diagram

- A Sequence of MIPS instructions

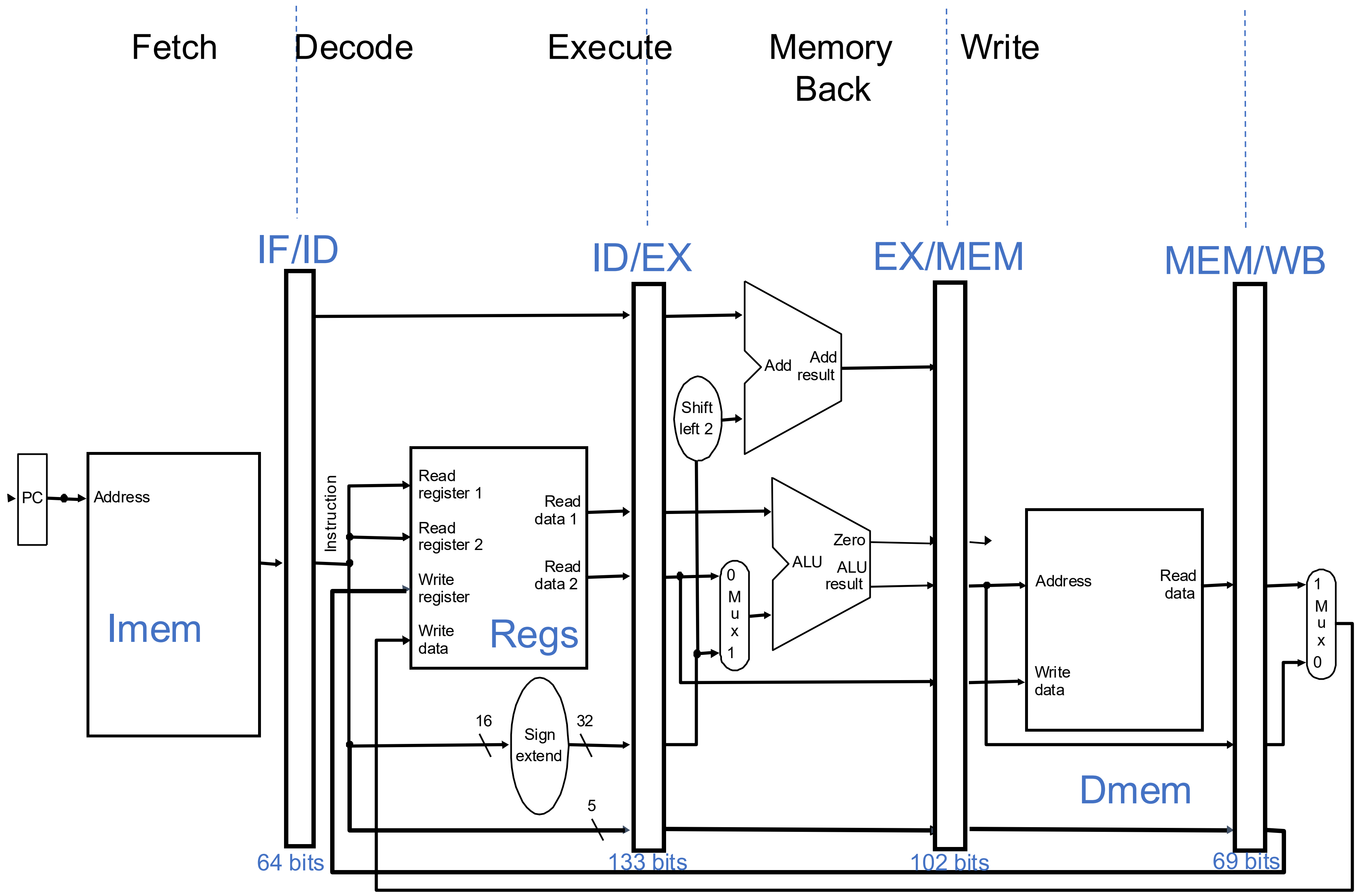


Time (in clock cycles) →

Program execution order (in instructions) ↓

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

Pipelined Datapath



Performance issue with pipelining

- **Goal**: Increase **instruction throughput**—more instructions completed per unit time.
- **Myth**: It doesn’t make **individual instructions** faster.
- **Fact**: Each instruction may take slightly longer due to pipeline control overhead.
- Example: 5 instructions in 5 stage pipeline. Each stage takes 1 cc

Metric	Without Pipelining	With Pipelining

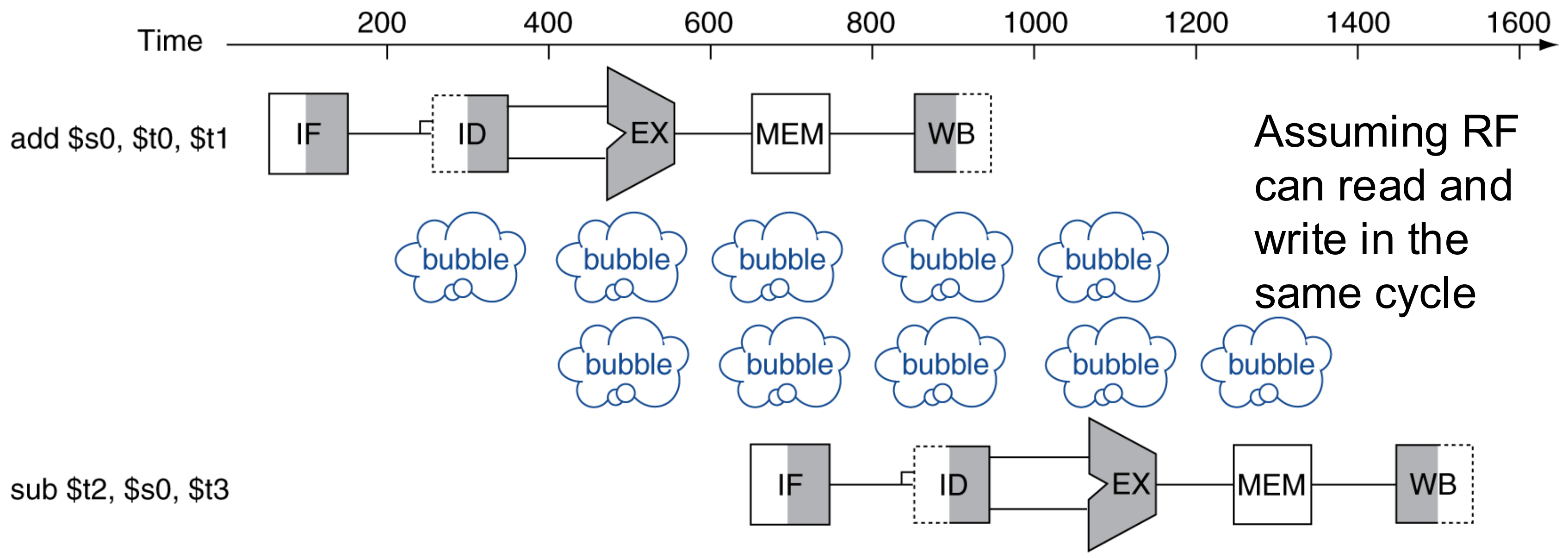
no single instruction runs faster, but **more instructions finish per unit time**, making the **whole program run faster**.

DATA HAZARDS

Data Hazards

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

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



Types of data hazards

- Read After Write (RAW), true, or dataflow, dependence

```
i1: add r1, r2, r3
i2: add r4, r1, r5
```

the most common, these occur when a read of register x by instruction j occurs before the write of register x by instruc- tion i. If this hazard were not prevented instruction j would use the wrong value of x.

Why it matters: This is the most common data hazard in pipelined processors. Without proper handling, your program could behave unpredictably

— like reading yesterday’s news thinking it’s today’s headline.

Types of data hazards

- Write After Read (WAR), anti dependence

```
i1: add r1, r2, r3
i2: add r2, r4, r5
```

this hazard occurs when read of register x by instruction i occurs after a

Why it matters:
- WAR hazards don’t happen in a basic 5-stage integer pipeline.
- But once instructions get shuffled for performance, these hazards can creep in.

Analogy: Reading a recipe *after* someone rewrote the ingredients — your dish won’t taste right!

Types of data hazards

- Write After Write (WAW), output dependence

  i1: add r1, r2, r3

  i2: add r1, r4, r5

  this hazard occurs when write of register x by instruction i occurs after a write of register x by instruction j. When this occurs, register x will have the

Why it matters:
- WAW hazards don't occur in a simple 5-stage integer pipeline.
- But they *do* show up when instructions are reordered or have varying execution times.

Analogy: It's like saving a document, then someone else edits/saves over it — your changes are gone!

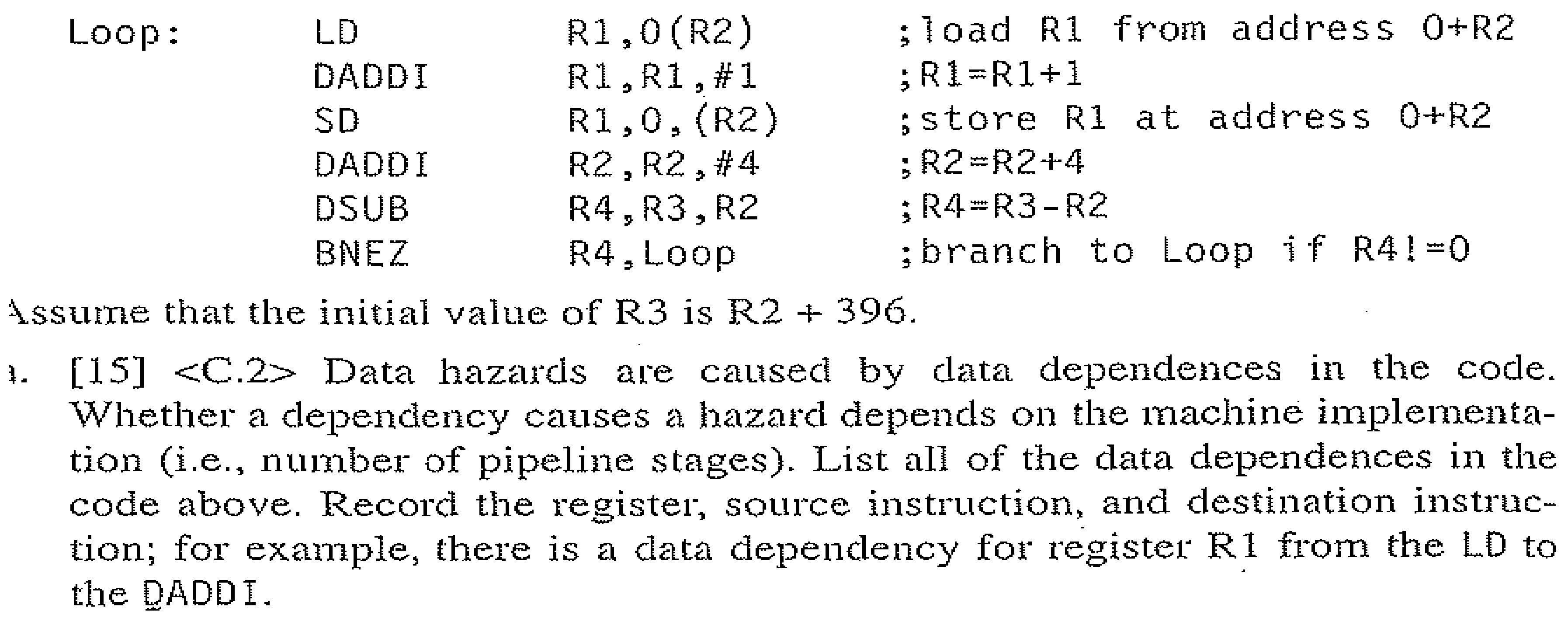
WAR & WAW

- WAR & WAW are name dependencies
  - **Dependence is on the register name**, not the actual value inside.
- Can be **eliminated by renaming**:
  - **Static** renaming: done in software (compiler level)
  - **Dynamic** renaming: handled by hardware (during execution)

1. No WAW or WAR in 5-Stage MIPS Pipeline, **in-order issue**

Analogy: Two people trying to write to the same folder — if you give each a unique folder name, no mix-ups happen!

Appendix C – Question C.1



```
Loop:     LD       R1,0(R2)      ;load R1 from address 0+R2
          DADDI    R1,R1,#1      ;R1=R1+1
          SD       R1,0,(R2)     ;store R1 at address 0+R2
          DADDI    R2,R2,#4      ;R2=R2+4
          DSUB     R4,R3,R2      ;R4=R3-R2
          BNEZ     R4,Loop       ;branch to Loop if R4!=0
```

Assume that the initial value of R3 is R2 + 396.

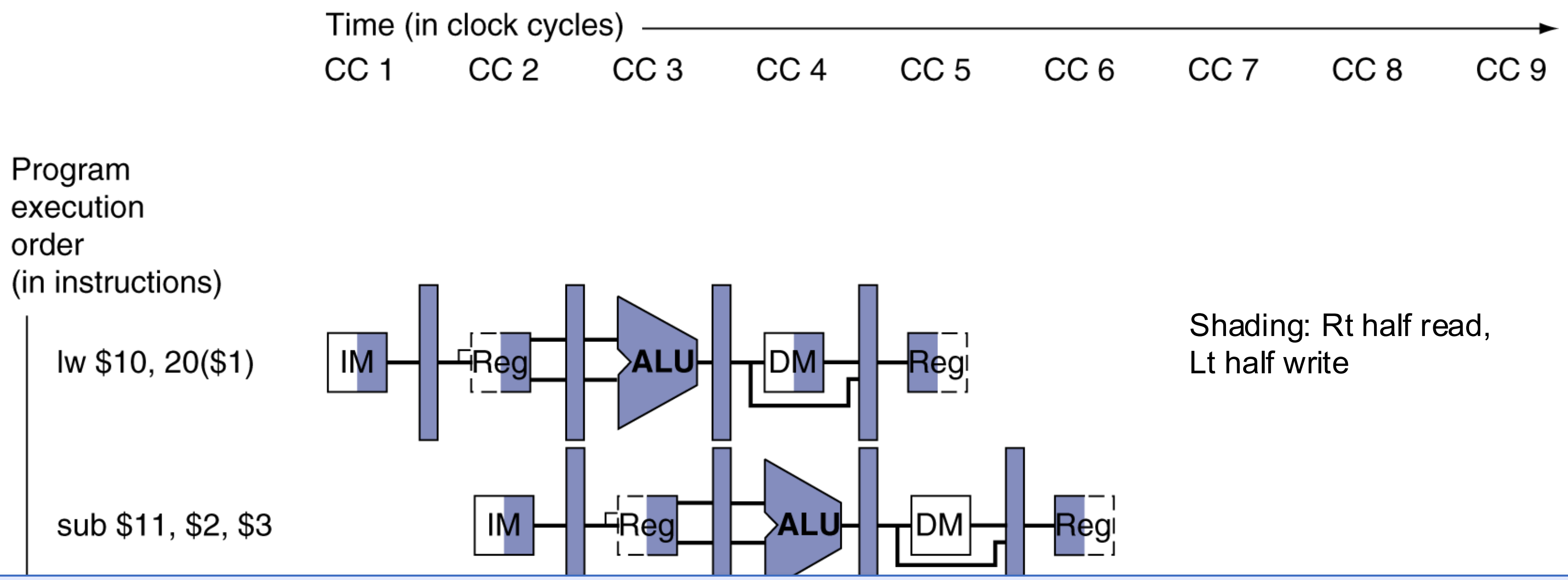
a. [15] <C.2> Data hazards are caused by data dependences in the code. Whether a dependency causes a hazard depends on the machine implementation (i.e., number of pipeline stages). List all of the data dependences in the code above. Record the register, source instruction, and destination instruction; for example, there is a data dependency for register R1 from the LD to the DADDI.

Solution C.1 a.

R1: LD -> DADDI - RAW
R1: DADDI -> SD - RAW
R1: LD -> SD - RAW
R1: LD -> DADDI - WAW
R2: LD -> DADDI - WAR
R2: SD -> DADDI - WAR
R2: DADDI -> DSUB - RAW
R4: DSUB -> BNEZ - RAW

What if it is a simple 5-stage linear pipeline?

Introduce Buffers to separate stages

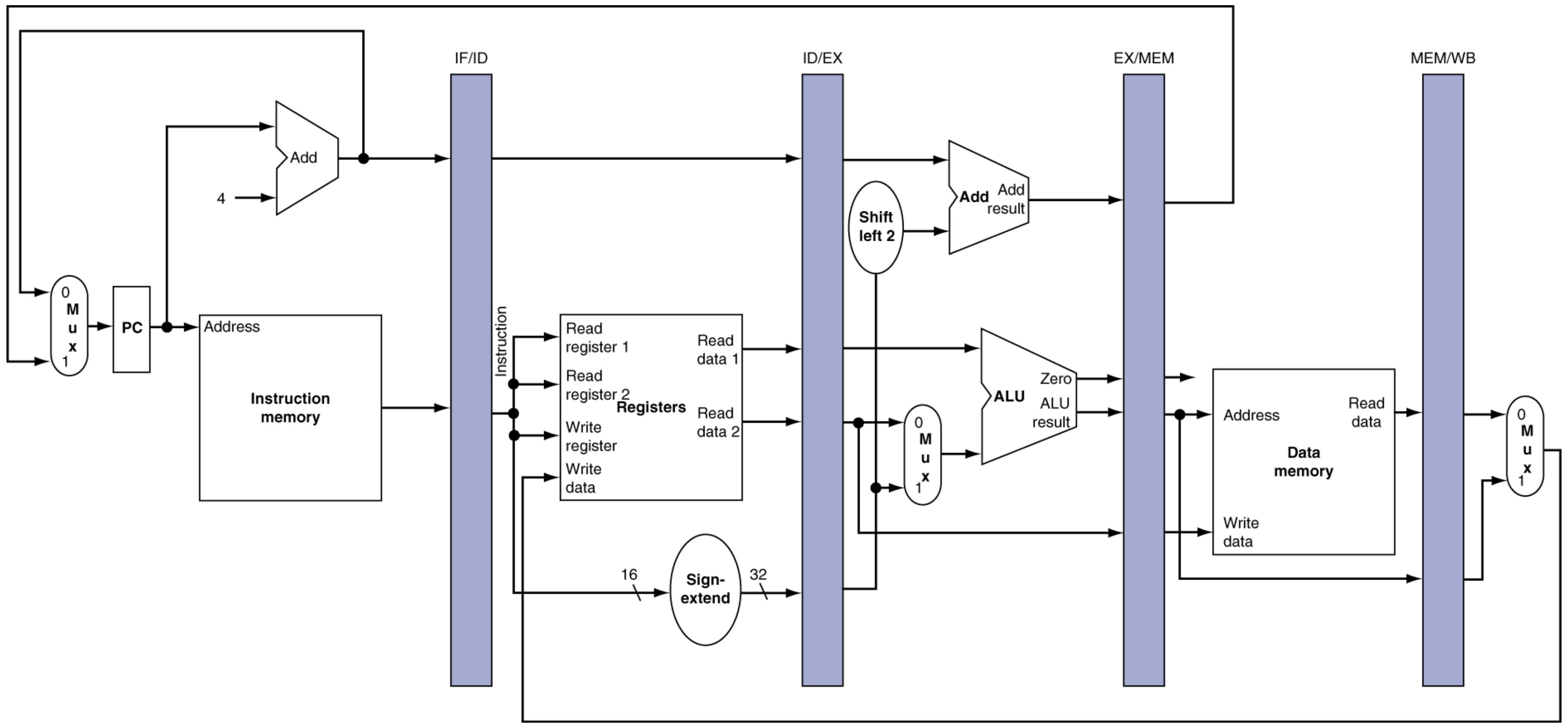


Shading: Rt half read,
Lt half write

1. What is the role of Pipeline buffers?
2. Why is a portion of the IM/DM and registers shaded?
3. How to decide the sequence if both read and write operations are happening in the same cc?

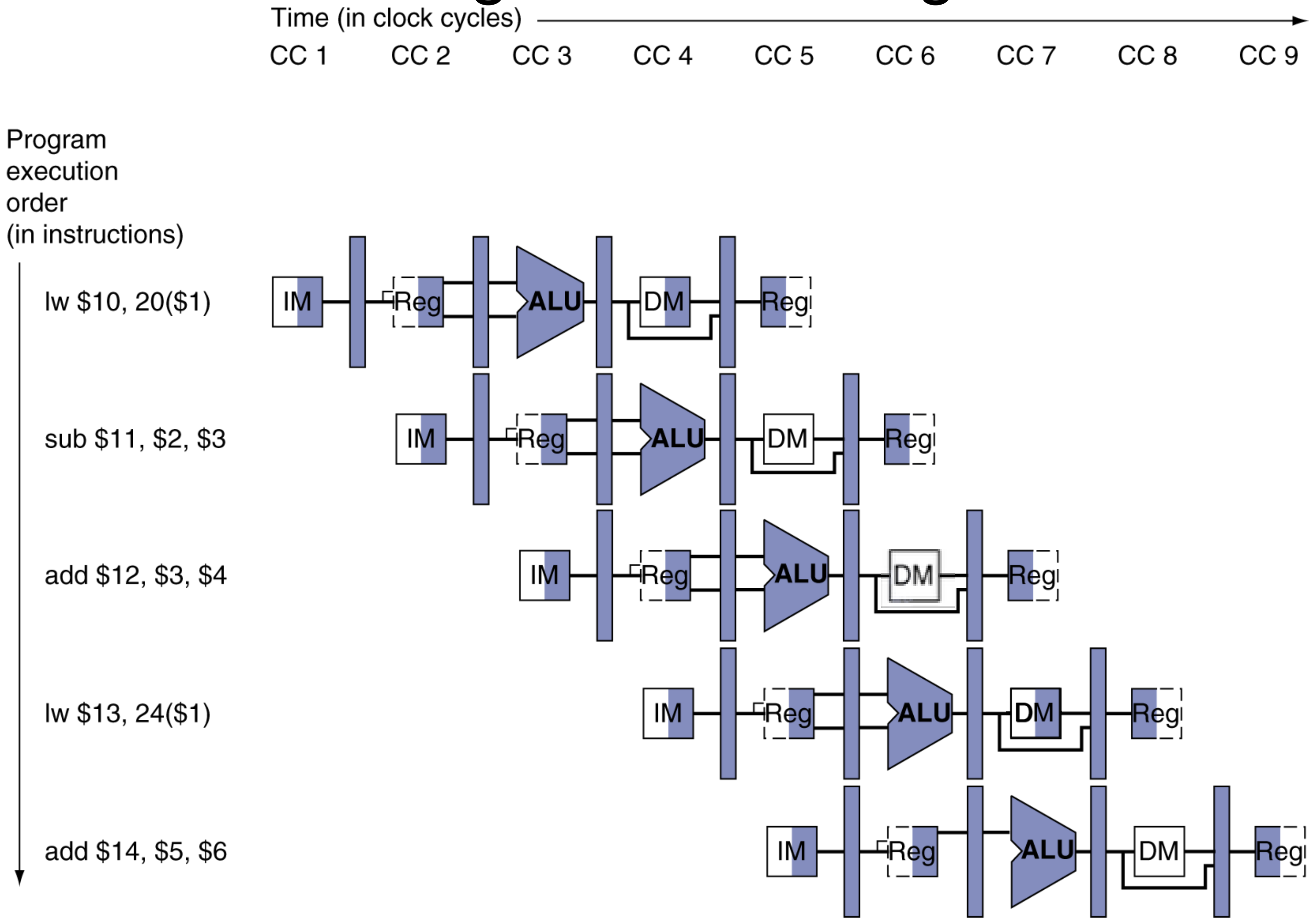
Pipeline registers

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



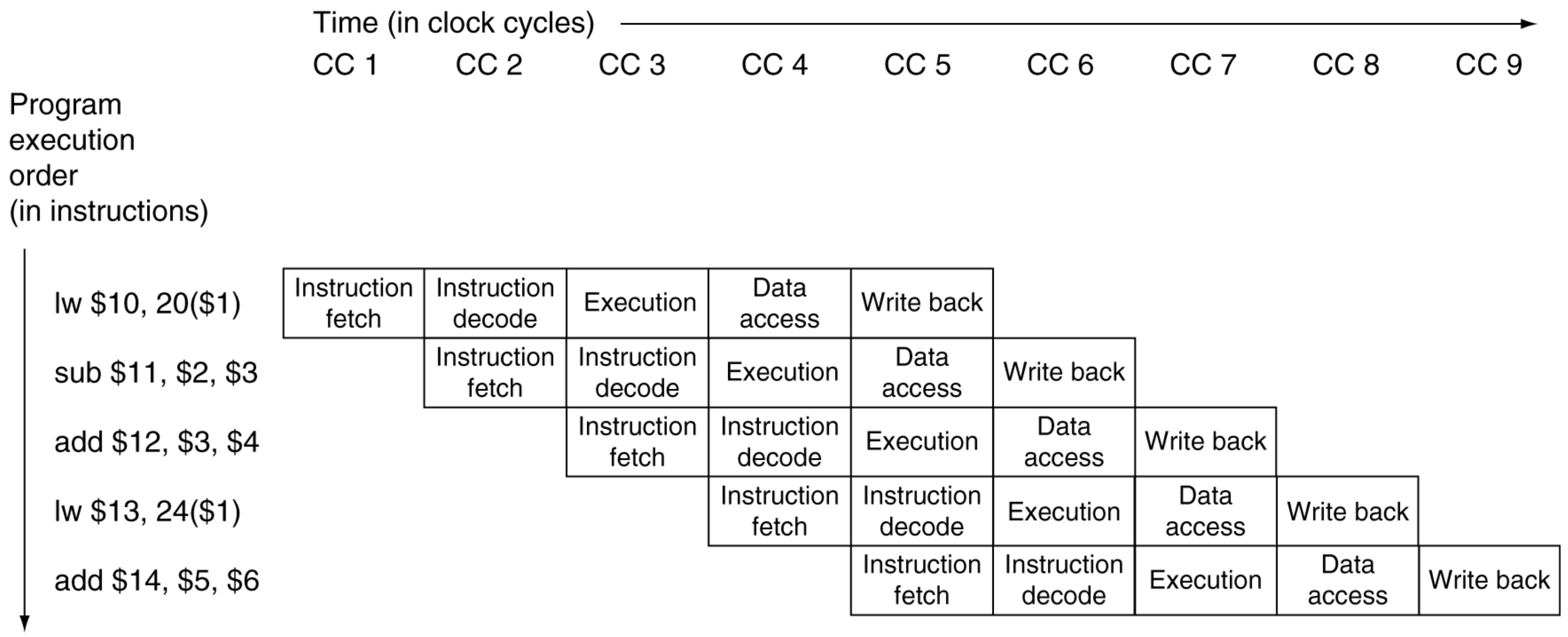
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

- Traditional form



Key Observations

1. Separate instruction and data memories

Key Observations

1. Separate instruction and data memories

- We use **separate instruction and data memories**—think of them as two specialized libraries:
- 📘 **Instruction Cache**: Stores the program's instructions.
- 📗 **Data Cache**: Holds the actual data the program works with.
  - This separation avoids a traffic jam: if both instruction fetch and data access tried to use the same memory at once, chaos would ensue. With two caches, each task gets its own lane.

Key Observations

1. Separate instruction and data memories

2. Register File Used twice (ID -> Read Values, WB -> Write Results)

Key Observations

1. Separate instruction and data memories

2. Register File Used twice (ID -> Read Values, WB -> Write Results)

- Since these actions happen at different times, we show the register file in **two places** in diagrams. Every clock cycle, we:
  - Perform **two reads** (for operands).
  - Do **one write** (to store the result).
- To avoid read/write conflicts, we split the clock:
- 🕐 **First half**: Write to the register.
- 🕒 **Second half**: Read from the register.

Key Observations

1. Separate instruction and data memories

2. Register File Used twice (ID -> Read Values, WB -> Write Results)

3. Branch Instructions
IF -> Update PC,
ID -> adder for branch target address,
EX-> Compare reg values and decide on Branch Taken/not taken

Key Observations

1. Separate instruction and data memories

2. Register File Used twice (ID -> Read Values, WB -> Write Results)

3. Branch Instructions (IF, ID, EX used stages; M, WB unused stages)

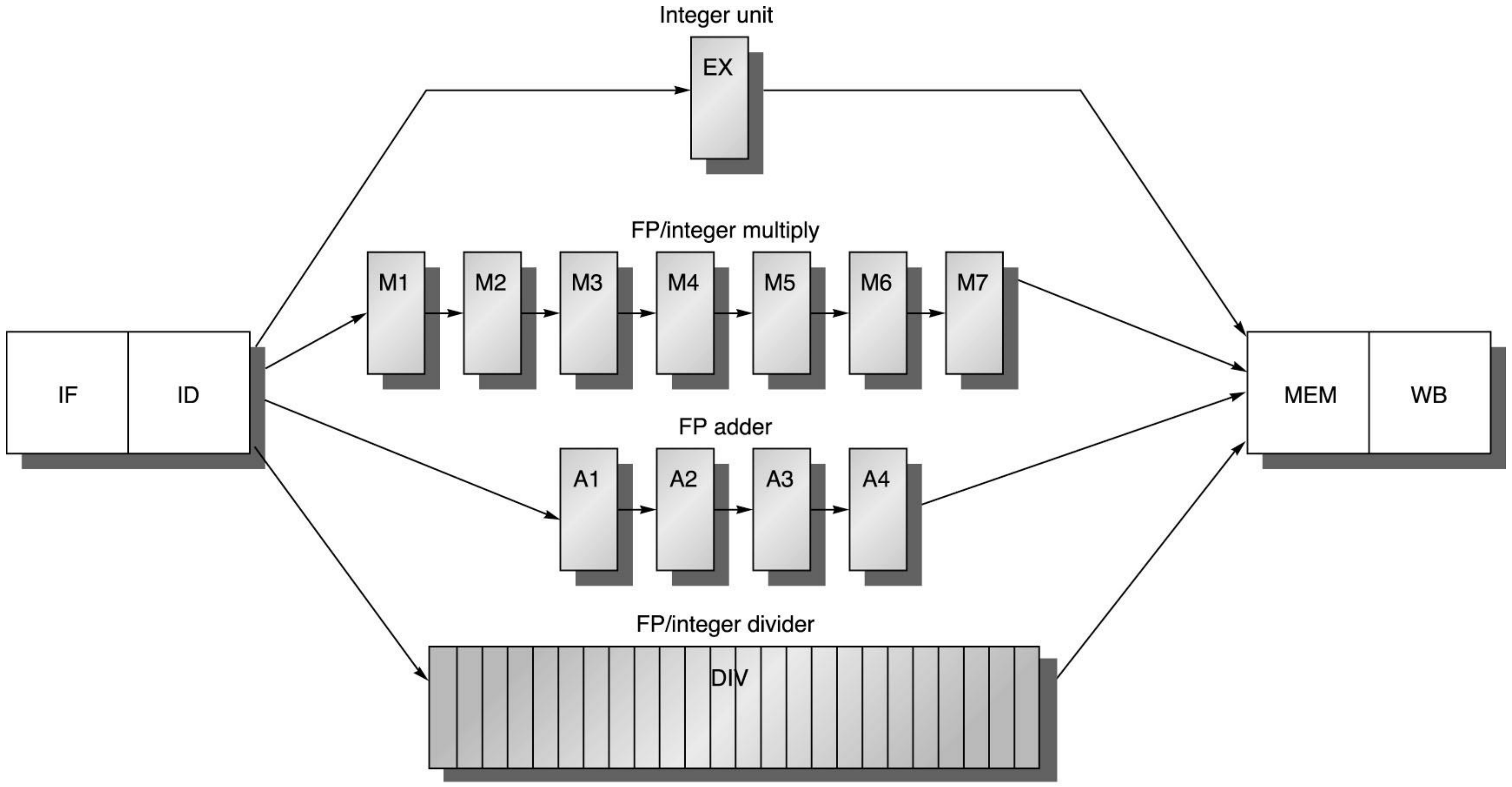
4. Pipeline Registers between stages=> **batons in a relay race**

Key Observations

1. Separate instruction and data memories
2. Register File Used twice (ID -> Read Values, WB -> Write Results)
3. Branch Instructions (IF, ID, EX used stages; M, WB unused stages)
4. Pipeline Registers between stages=> **batons in a relay race**

•At the end of each clock cycle, results from one stage are stored.
•On the next cycle, the next stage picks up the baton and continues.
This ensures:
• ✅ No resource conflicts.
• ✅ Smooth handoff between stages.
• ✅ Maximum instruction throughput.

A pipeline with multi-cycle FP operations (Ex. MIPS R4000)



Multi-Cycle FP Operations in an Pipeline

- Unlike simple integer operations, **floating-point operations** (like multiplication, division, and square root) are **computationally intensive** and require **multiple clock cycles** to complete. This is because:
  - They involve more complex arithmetic.
  - Precision requirements are higher.
  - Hardware units (like FP multipliers/dividers) are slower and more resource-heavy.

Multi-Cycle FP Pipeline Work

- Unlike integer pipelines (which often complete in 1 cycle per stage), FP pipelines:
- Allow **multiple instructions to be in-flight** across different stages.
- Use **reservation stations** or **scoreboarding** to track instruction progress.
- Require **stalling or forwarding** to handle **data hazards** (e.g., when a later instruction depends on an unfinished FP result).

Multi-Cycle FP Pipeline: Analogy

- Think of the FP pipeline like a **multi-lane toll booth**:
  - Each lane handles a different type of vehicle (operation).
  - Some vehicles (like trucks = divide) take longer to pass.
  - If too many trucks arrive, cars (add/multiply) must wait their turn.
- Let’s say we issue a **FP divide** followed by a **FP add**:
  - Divide takes ~30 cycles.
  - Add takes ~5 cycles.
  - If the add is issued too soon (e.g., between cycle 28–33), it may **stall** because the divide hasn’t released the hardware or produced its result

Challenges in Multi-Cycle FP Pipelines

Structural Hazards

- Limited hardware means only one instruction can use a unit at a time. If two instruct. need the same unit, one must wait.

Data Hazards (RAW)

- If an inst depends on the result of a prev. inst, it must wait until the result is ready.

Control Hazards

- Branch decisions based on FP compare may be delayed, affecting instruct. flow

Imprecise Exceptions

- Because FP instructions finish out of order, exceptions (like divide-by-zero) can be hard to trace.
- Some CPUs use **precise mode** (slower but easier to debug) or **buffer results** until safe to commit.

Pipeline Hazards

- Hazards are caused by conflicts between instructions. Will lead to incorrect behavior if not fixed.
- Three types:
  - Structural: two instructions use same h/w in the same cycle – resource conflicts (e.g. one memory port, unpipelined divider etc).
  - Data: two instructions use same data storage (register/memory) – dependent instructions.
  - Control: one instruction affects which instruction is next – PC modifying instruction, changes control flow of program.

1. Force Stalls or Insert Bubbles

- **What it means:**
  - A **stall** pauses the pipeline at a specific stage.
  - A **bubble** is like a placeholder (NOP) that moves through the pipeline without doing anything.
- **Why it's needed:**
  - To wait for **multi-cycle FP operations** to finish→ data hazard
- **Implementation:**
  - **De-assert write-enable signals** on pipeline registers to freeze the current stage.
  - This prevents new data from overwriting the stalled instruction.

2. Stop Younger Instructions

- **What it means:**
  - "Younger" instructions (issued later) are paused if they depend on "older" ones.
- **Why it's needed:**
  - Prevents incorrect execution due to **RAW (Read After Write)** hazards.
- **Implementation:**
  - Use **hazard detection units** to monitor dependencies.
  - Stall the decode or execute stage until the older instruction completes.

3. Make Younger Instructions Wait

- **What it means:**
  - Instructions wait their turn to avoid resource conflicts or data hazards.
- **Why it's needed:**
  - FP units (like dividers) are **multi-cycle and shared**, so only one instruction can use them at a time. --- Structural Hazard
- **Implementation:**
  - Use **scoreboarding** or **reservation stations** to track availability.
  - Stall or reorder instructions dynamically.

4. Flush the Pipeline

- **What it means:**
  - Remove all instructions currently in the pipeline.
- **Why it's needed:**
  - To handle **control hazards**, especially after a mispredicted branch.
- **Implementation:**
  - **Assert clear signals** on pipeline registers.
  - This wipes out the instructions and resets the pipeline stages.

5. Blow Instructions Out of the Pipeline

- **What it means:**
  - Similar to flushing, but more aggressive—used when a branch or exception invalidates current instructions.
- **Why it's needed:**
  - Ensures no incorrect instructions are executed. ---- Branch Misprediction
- **Implementation:**
  - Replace instructions with NOPs or reset control signals.

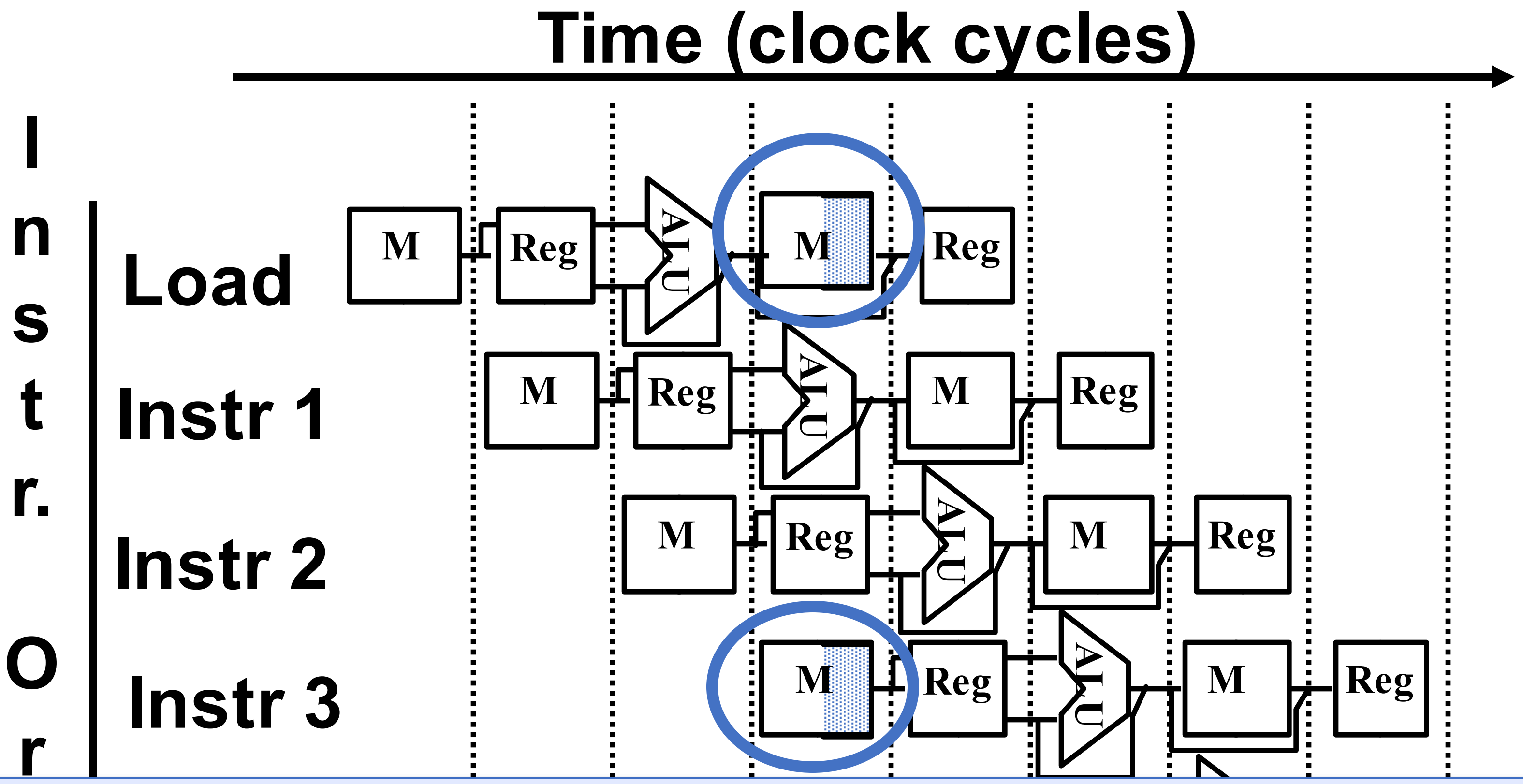
6. Re-fetch New Instructions

- **What it means:**
  - After flushing, the processor fetches the correct instructions from memory.
- **Why it's needed:**
  - To recover from control hazards like branches or jumps.
- **Implementation:**
  - Update the **Program Counter (PC)** with the correct target.
  - Restart fetch from the new address.

Handling Hazards --- Summary

Mechanism	Purpose	Triggered By	Implementation
Stall/Bubble	Pause pipeline	Data hazard	De-assert/deactivate write-enable
Stop Younger Instructions	Prevent premature execution	RAW hazard	Hazard detection logic
Wait for Older Ones	Avoid resource conflicts	Structural hazard	Scoreboarding / reservation stations
Flush Pipeline	Clear invalid instructions	Control hazard	Assert clear signals
Blow Instructions	Remove corrupted instructions	Branch misprediction	Insert NOPs or reset control
Re-fetch Instructions	Resume correct execution	Branch resolution	Update PC and restart fetch

Single Memory is a Structural Hazard



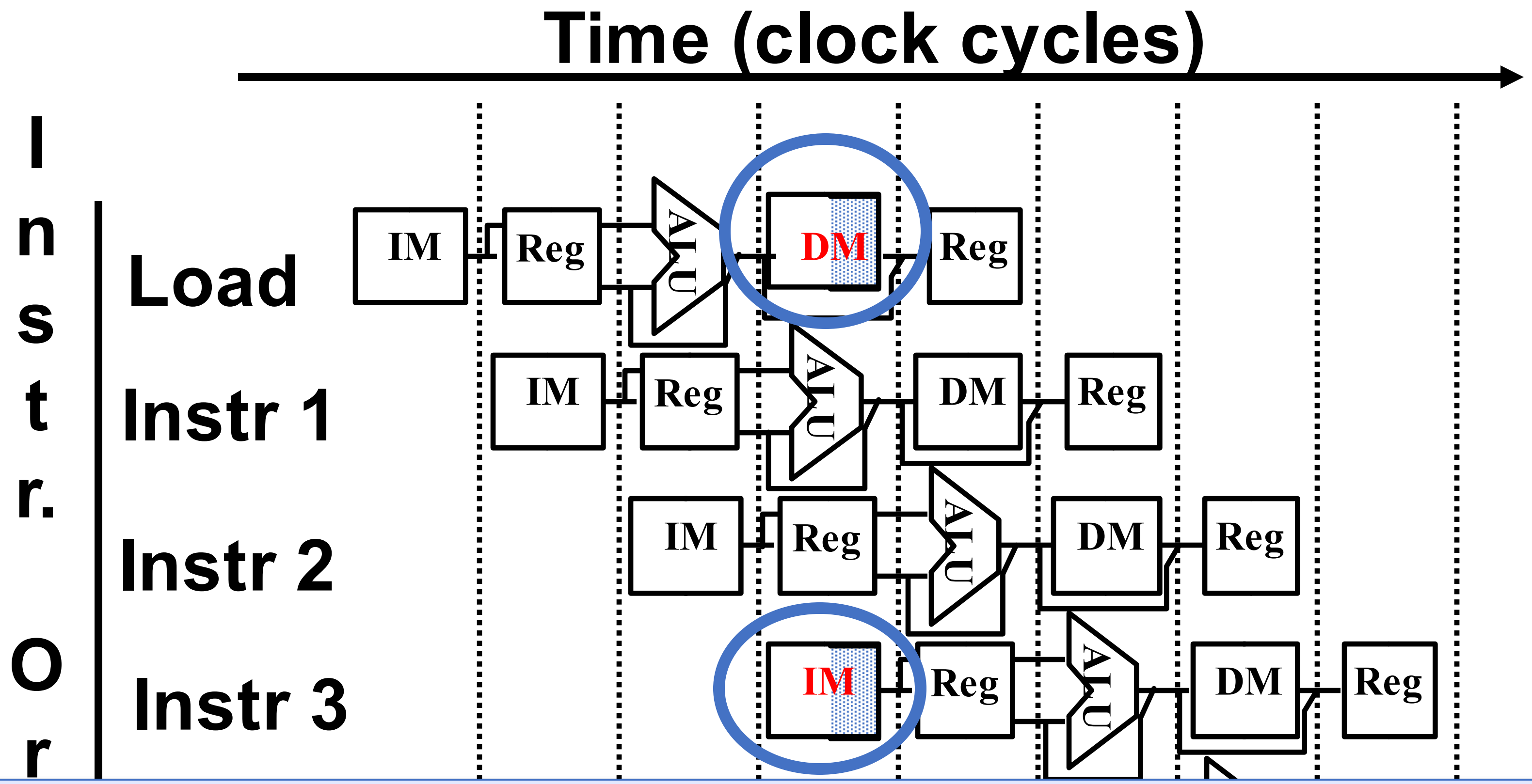
1. In the Figure, ALU and MEM units are bypassed.

Is there any typo in the figure?

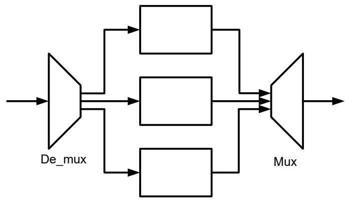
2. Mem Read done by two different instructions in same cycle.

Will it result in any issues in the datapath?

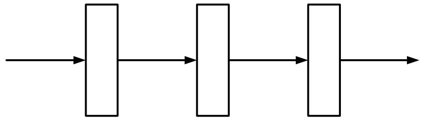
Harvard Architecture: Dual Memory



How about two separate memories IM for instruction and DM for data?
Almost all commercial architectures provide L1 I-cache and D-cache
Called Harvard Architecture



Dealing with Structural Hazards



Strategy	Performance Boost	Hardware Cost	Complexity	Best For
Stall	Low	Low	Low	Cheap, rarely used resources
Replicate	High	High	Medium	Frequently used simple units
Pipeline	High	Medium	High	Multi-cycle or complex resources

Analogy -------------
Stall: Like waiting at a red light—simple, but slows you down.
Replicate: Like adding more checkout counters at a store—faster, but costs more.
Pipeline: Like a car wash with multiple stages—efficient, but needs coordination.

Speed Up Equation for Pipelining

$CPI_{pipelined}$ = Ideal CPI + Pipeline stall clock cycles per instn

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI/Ideal CPI}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

Pipelining Summary

- 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
- What makes it really hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

HAZARDS

Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

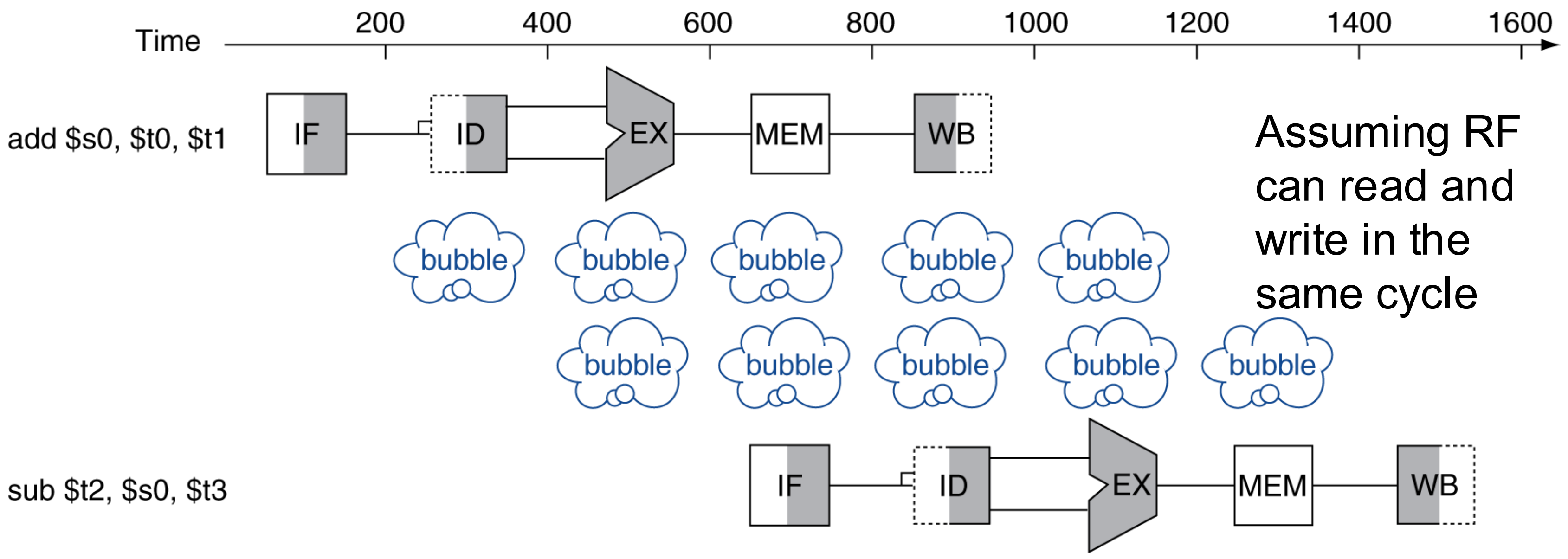
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data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

DATA HAZARD PREVENTION: FORWARDING / BYPASSING / SHORT-CIRCUITING

How can we speed this up??

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

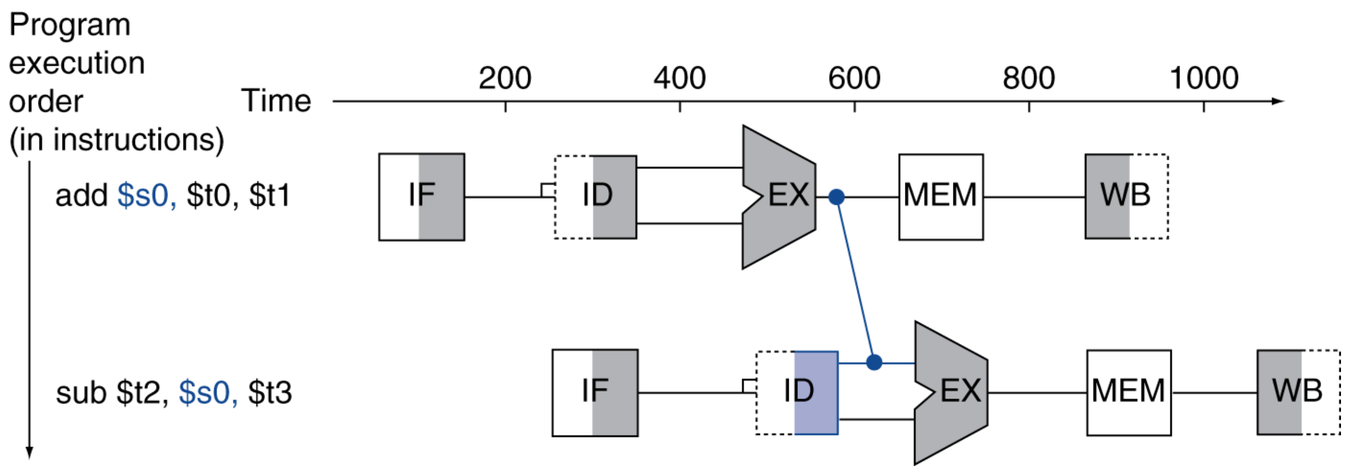
  -
    ```
    add   $s0, $t0, $t1
    sub   $t2, $s0, $t3
    ```



Assuming RF can read and write in the same cycle

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



Forwarding Logic in Pipelined CPUs

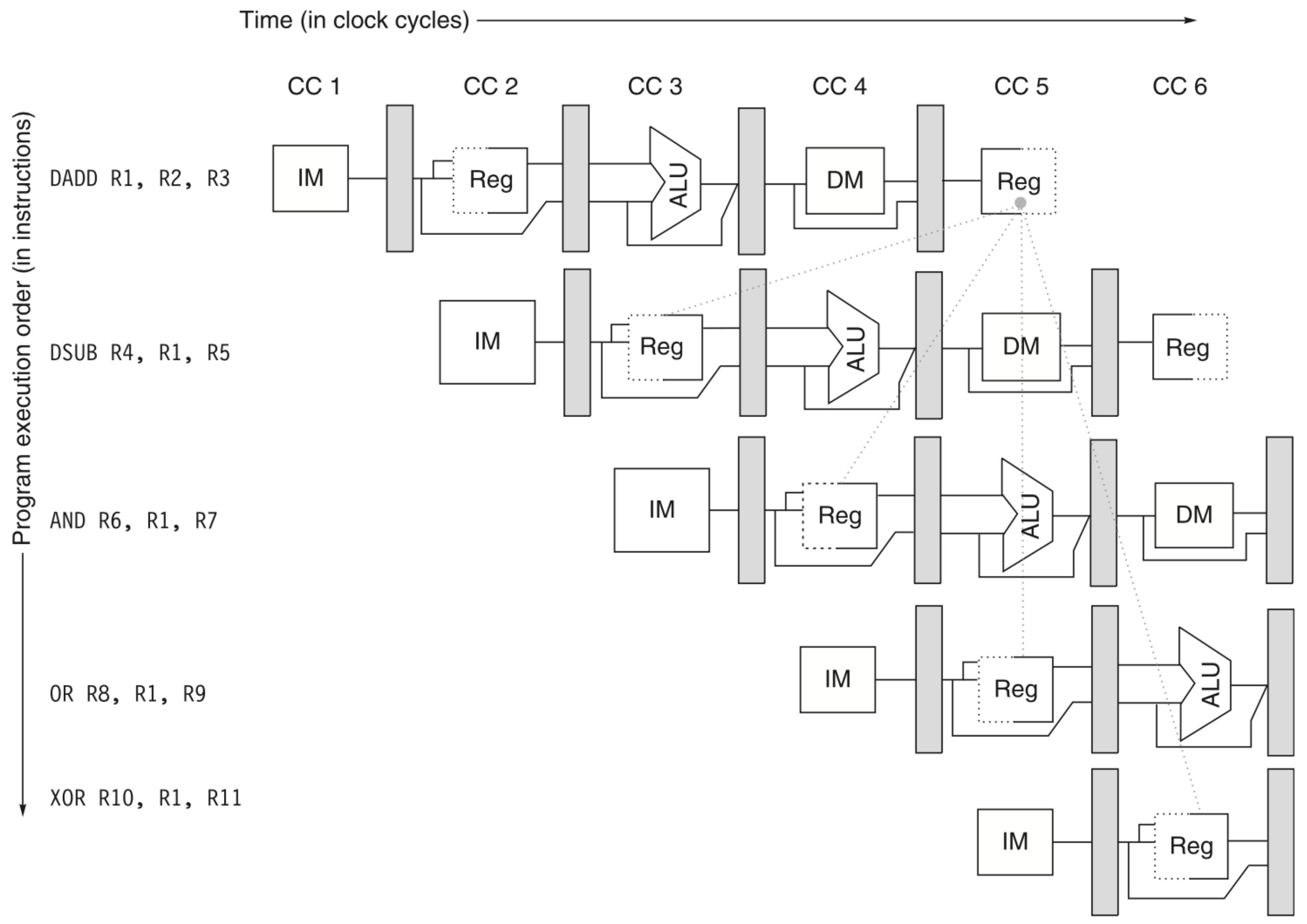


Figure C.4 The use of the result of the add **instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.**

Forwarding Logic in Pipelined CPUs

1. Always Ready to Forward

- The **ALU result** from both EX/MEM and MEM/WB pipeline registers → **continuously fed back** to ALU inputs.

2. Smart Detection & Selection

- **Forwarding hardware** checks if:
  - A previous ALU instruction **writes to a register**
  - That register is a **source operand** for the current ALU instruction
- If true:
  - **Control logic** selects the **forwarded result** → instead of the stale value from the **register file**

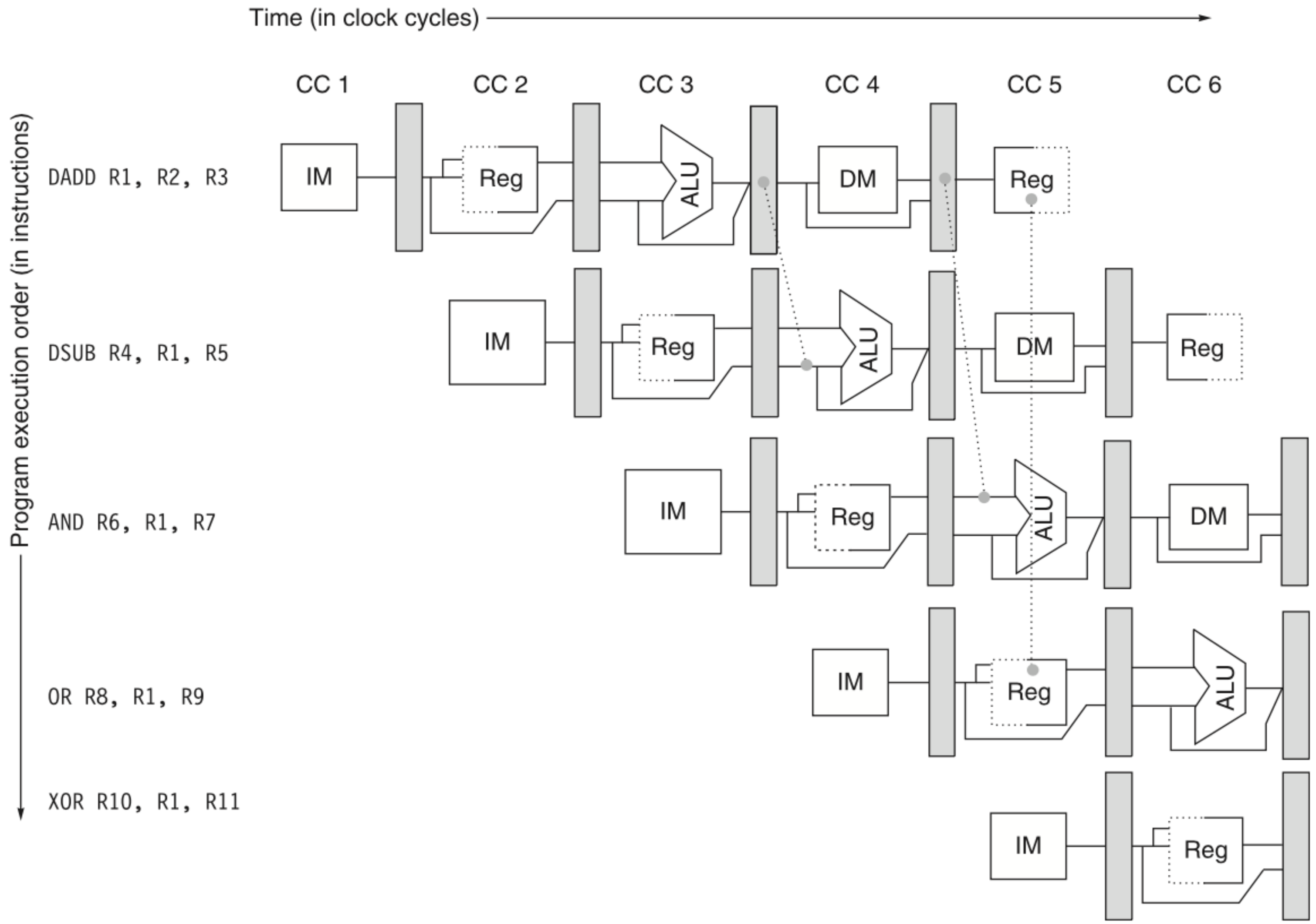


Figure C.5 A set of instructions that depends on the `add` **result uses forwarding paths to avoid the data hazard.** The inputs for the `sub` and `and` instructions forward from the pipeline registers to the first ALU input. The `or` receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the `and` instruction was `and x6,x1,x4`.

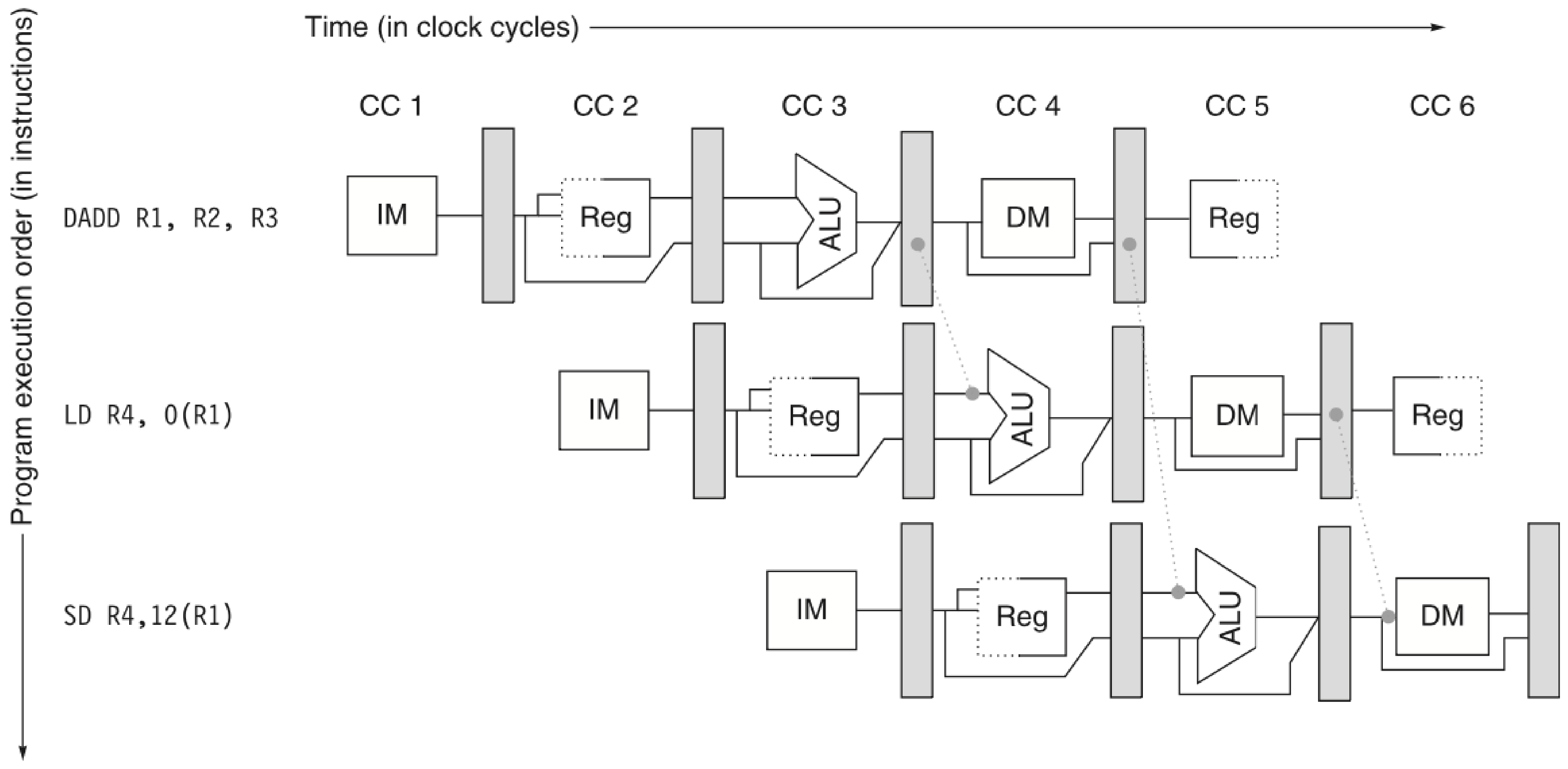


Figure C.6 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown herein), the result would need to be forwarded to prevent a stall.

DATA HAZARD PREVENTION: STALLS

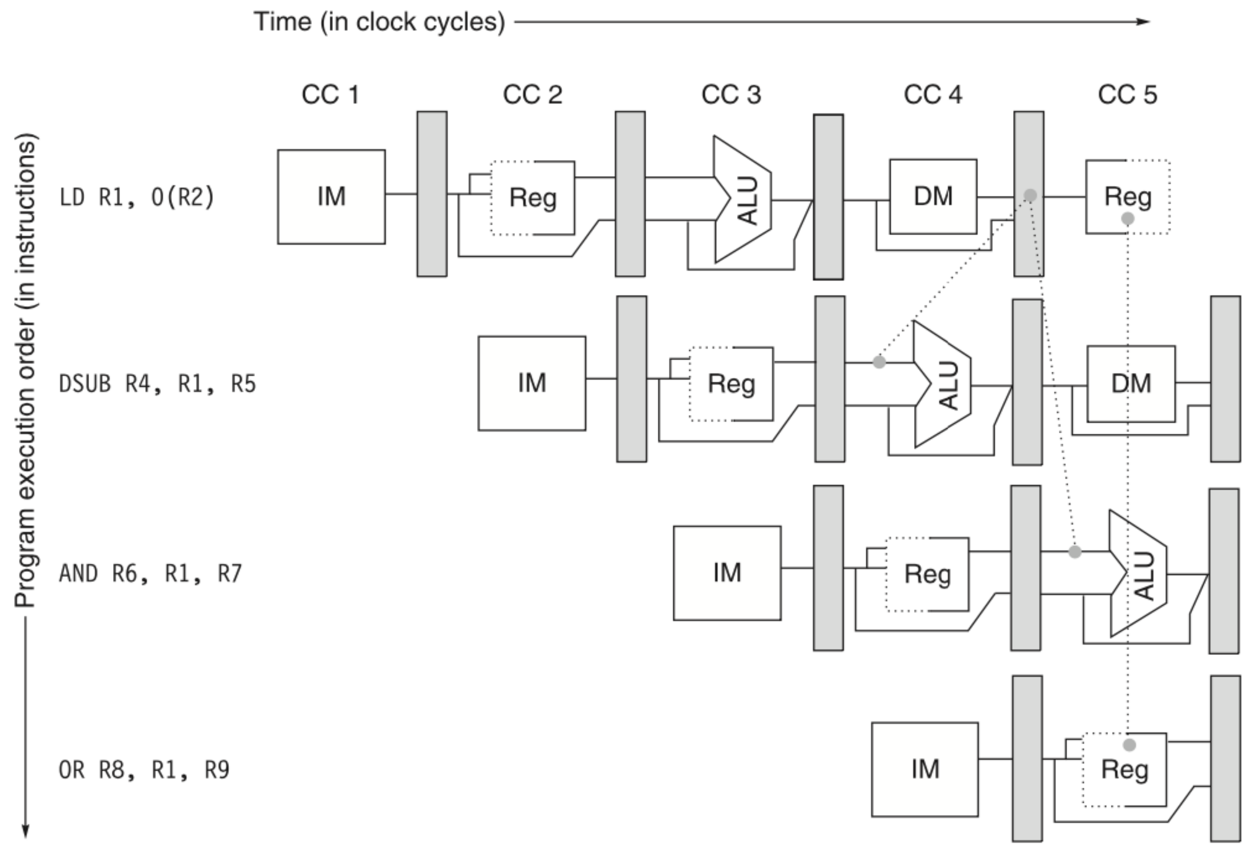


Figure C.7 The load instruction can bypass its results to the `and` and `or` instructions, but not to the `sub`, because that would mean forwarding the result in "negative time."

`ld` instruction gets its data **at the end of clock cycle 4** (MEM stage).

`dsub` instruction needs that data **at the beginning of clock cycle 4** — too early!

🚫 **Why forwarding fails here:** To fix this, the data would need to travel *backward in time* — something hardware just can't do (yet!).

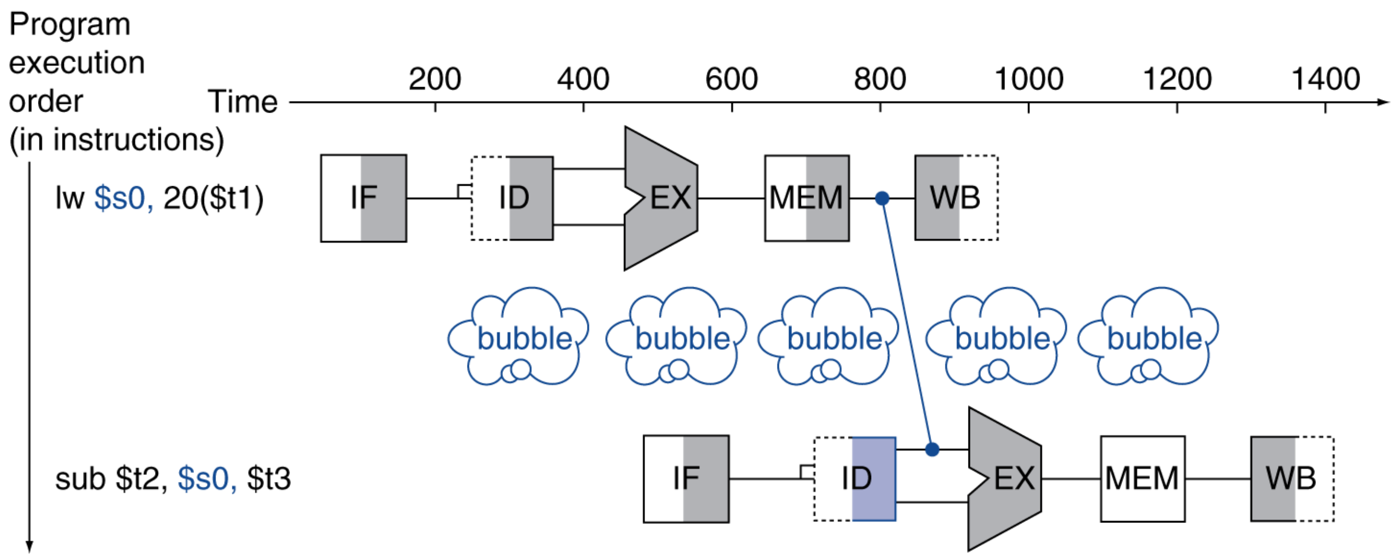
`and` instruction: starts **2 cycles later** -------------------------→ gets data via forwarding ✔

`or` instruction: reads from register file-------------------------→ no hazard ✔

`dsub` instruction: needs data too soon -------------------------→ forwarding arrives **too late** ❌

Load-Use Data Hazard

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



Pipeline Interlock: Handling Load-Use Hazards

1. How is Pipeline Interlock Stall different than the stalls we have been studying so far?

MEM stage

- Forwarding can't help if the next instruction needs the data **earlier**

Enter the Pipeline Interlock (hardware)

- **Detects hazards** (like load-use)
- **Stalls** the pipeline until the data is ready

- Inserts a **bubble** (NOP) to delay dependent instruction

Pipeline Interlock: Handling Load-Use Hazards

1. How is Pipeline Interlock Stall different than the stalls we have been studying so far?

Type	**What It Is**	**When It's Used**	**Who Controls It**
Interlock Stall	Automatic hardware mechanism that detects hazards and stalls dependent instructions	Used for **data hazards** (especially load-use hazards) when forwarding isn’t enough	Controlled by **hardware**
Regular Stall	Manual or compiler-inserted delay (e.g., inserting NOPs or reordering instructions)	Used for **structural hazards**, **control hazards**, or when hardware lacks interlock support	Controlled by **software/compiler** or **hardware**

Pipeline Interlock: Handling Load-Use Hazards

System	Description	Use of Interlock Stalls
MIPS Architecture (Classic 5-stage pipeline)	Educational and early RISC processor design	Uses interlock stalls for **load-use hazards** when forwarding isn't sufficient
ARM Cortex-M Series	Microcontrollers used in embedded systems	Implements interlocks to manage **data hazards** in simple pipelines
SPARC (Scalable Processor Architecture)	RISC architecture used in enterprise systems	Employs interlocks for **RAW hazards** in its pipeline stages
RISC-V (Basic implementations)	Open-source ISA used in academia and industry	Some implementations use interlocks for **hazard resolution**, especially in simpler cores
Intel 4004 / 8086 (early x86)	Early microprocessors	Relied on hardware interlocks due to lack of advanced compiler scheduling

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;
  - Reordering is commonly done by modern compilers

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

?? cycles

Code Scheduling to Avoid Stalls

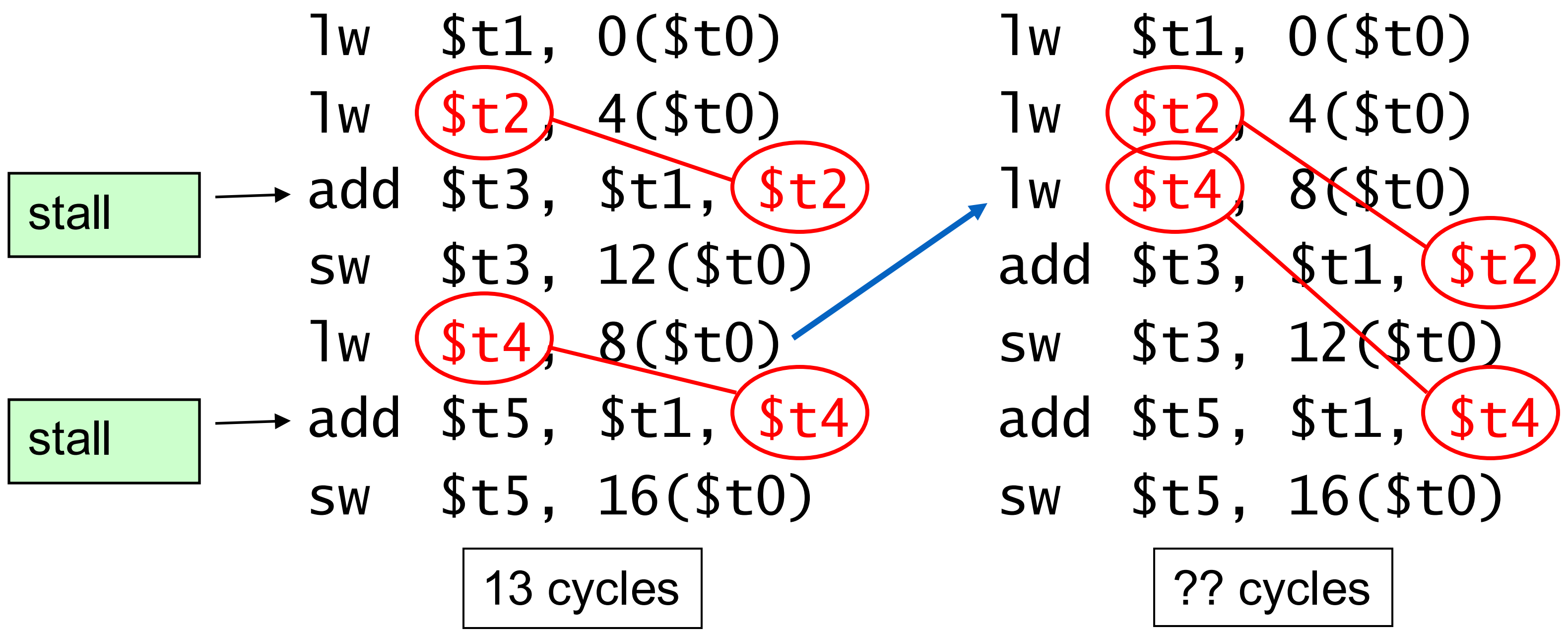
- Reorder code to avoid use of load result in the next instruction
  - C code for A = B + E; C = B + F;
  - Reordering is commonly done by modern compilers

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

13 cycles

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;
  - Reordering is commonly done by modern compilers

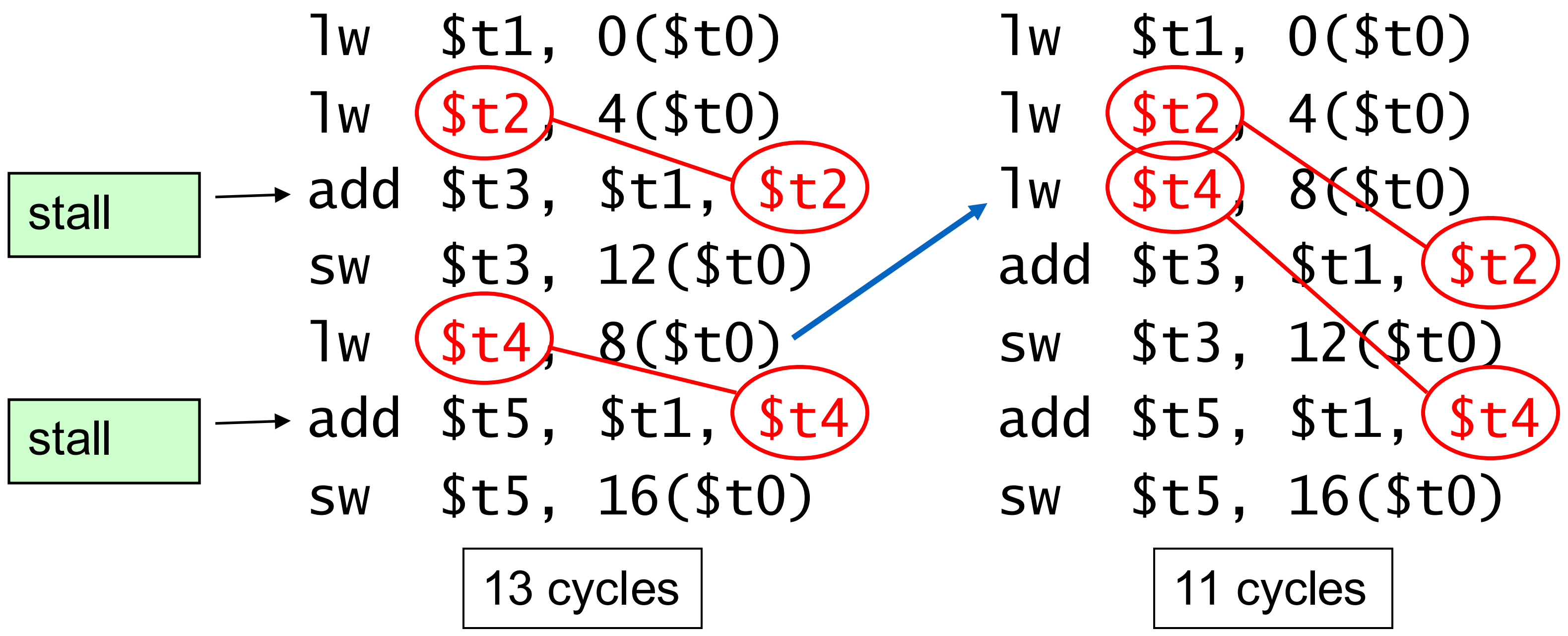


```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

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

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;
  - Reordering is commonly done by modern compilers



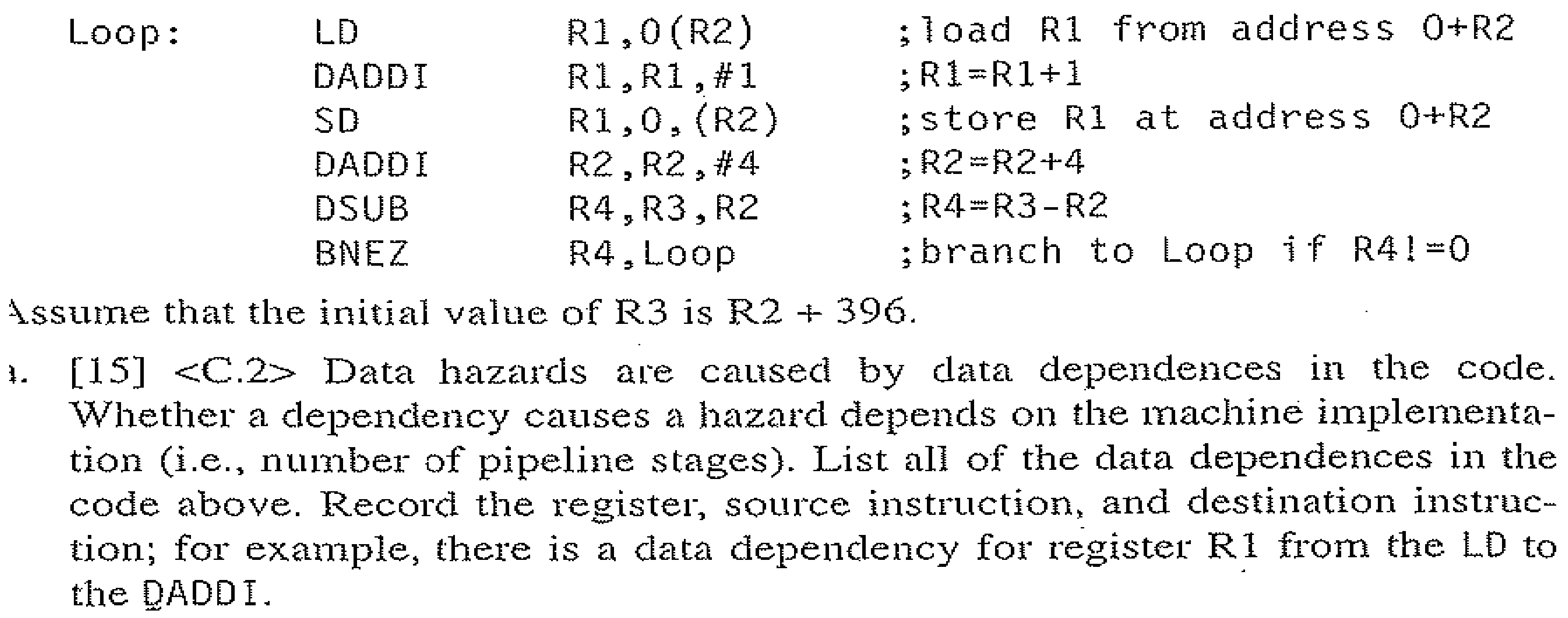
```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

13 cycles

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

11 cycles

Appendix C – Question C.1



```
Loop:      LD       R1,0(R2)      ;load R1 from address 0+R2
           DADDI    R1,R1,#1      ;R1=R1+1
           SD       R1,0,(R2)     ;store R1 at address 0+R2
           DADDI    R2,R2,#4      ;R2=R2+4
           DSUB     R4,R3,R2      ;R4=R3-R2
           BNEZ     R4,Loop       ;branch to Loop if R4!=0
```

Assume that the initial value of R3 is R2 + 396.

a. [15] <C.2> Data hazards are caused by data dependences in the code. Whether a dependency causes a hazard depends on the machine implementation (i.e., number of pipeline stages). List all of the data dependences in the code above. Record the register, source instruction, and destination instruction; for example, there is a data dependency for register R1 from the LD to the DADDI.

Solution C.1 a.

R1: LD -> DADDI - RAW
R1: DADDI -> SD - RAW
R1: LD -> SD - RAW
R1: LD -> DADDI - WAW
R2: LD -> DADDI - WAR
R2: SD -> DADDI - WAR
R2: DADDI -> DSUB - RAW
R4: DSUB -> BNEZ - RAW

What if it is a simple 5-stage linear pipeline?

```
Loop:      LD         R1,0(R2)      ;load R1 from address 0+R2
           DADDI      R1,R1,#1      ;R1=R1+1
           SD         R1,0,(R2)     ;store R1 at address 0+R2
           DADDI      R2,R2,#4      ;R2=R2+4
           DSUB       R4,R3,R2      ;R4=R3-R2
           BNEZ       R4,Loop       ;branch to Loop if R4!=0
```

Assume that the initial value of R3 is R2 + 396.

b. [15] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline without any forwarding or bypassing hardware but assuming that a register read and a write in the same clock cycle "forwards" through the register file, as shown in Figure C.6. Use a pipeline timing chart like that in Figure C.5. Assume that the branch is handled by flushing the pipeline. If all memory references take 1 cycle, how many cycles does this loop take to execute?

b. Forwarding is performed only via the register file. Branch outcomes and targets are not known until the end of the execute stage. All instructions introduced to the pipeline prior to this point are flushed.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
LD R1, 0(R2)	F	D	X	M	W													
DADDI R1, R1, #1		F	s	s	D	X	M	W										
SD 0(R2), R1					F	s	s	D	X	M	W							
DADDI R2, R2, #4								F	D	X	M	W						
DSUB R4, R3, R2									F	s	s	D	X	M	W			
BNEZ R4, Loop												F	s	s	D	X	M	W
LD R1, 0(R2)																	F	D

Since the initial value of R3 is R2 + 396 and equal instance of the loop adds 4 to R2, the total number of iterations is 99. Notice that there are 8 cycles lost to RAW hazards including the branch instruction. Two cycles are lost after the branch because of the instruction flushing. It takes 16 cycles between loop instances; the total number of cycles is 98 × 16 + 18 = 1584. The last loop takes two addition cycles since this latency cannot be overlapped with additional loop instances.

```
Loop:     LD        R1,0(R2)     ;load R1 from address 0+R2
          DADDI     R1,R1,#1     ;R1=R1+1
          SD        R1,0,(R2)    ;store R1 at address 0+R2
          DADDI     R2,R2,#4     ;R2=R2+4
          DSUB      R4,R3,R2     ;R4=R3-R2
          BNEZ      R4,Loop      ;branch to Loop if R4!=0
```

ssume that the initial value of R3 is R2 + 396.

c. [15] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline with full forwarding and bypassing hardware. Use a pipeline timing chart like that shown in Figure C.5. Assume that the branch is handled by predicting it as not taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?

c. Now we are allowed normal bypassing and forwarding circuitry. Branch outcomes and targets are known now at the end of decode.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
LD R1, 0(R2)	F	D	X	M	W													
DADDI R1, R1, #1		F	D	s	X	M	W											
SD R1, 0(R2)			F	s	D	X	M	W										
DADDI R2, R2, #4					F	D	X	M	W									
DSUB R4, R3, R2						F	D	X	M	W								
BNEZ R4, Loop							F	s	D	X	M	W						
(incorrect instruction)									F	s	s	s	s					
LD R1, 0(R2)										F	D	X	M	W				

Again we have 99 iterations. There are two RAW stalls and a flush after the branch since the branch is taken. The total number of cycles is $9 \times 98 + 12 = 894$. The last loop takes three addition cycles since this latency cannot be overlapped with additional loop instances.

CONTROL HAZARDS

MIPS Branch Instruction [Assembly]

```
BEQ $t0, $t1, LABEL # Branch if $t0 == $t1
ADD $t2, $t2, $t3    # Next instruction(speculatively fetched)
LABEL: SUB $t4, $t4, $t5
```

1. Do you observe any stalls or delay introduced to the pipeline?

2. At **which stage** in the 5-stage MIPS pipeline, branch target address is calculated and branch decision is made?

```
BEQ $t0, $t1, LABEL
```

Stage	**Action**
IF	Fetch the instruction
ID	Decode and read registers \$t0 and \$t1
EX	Compare \$t0 and \$t1, and compute branch target address

2. At **which stage** in the 5-stage MIPS pipeline, branch target address is calculated and branch decision is made?

- The comparison between reg happens in the ALU during the EX stage.
- The **branch target address** is also calculated here using the PC and offset.
- Only after **EX** can the processor decide whether the branch is taken or not.

`BEQ $t0, $t1, LABEL`

1. Do you **observe any stalls** or delay introduced to the pipeline?

- **Consequence: Control Hazard**
  - Since the branch decision is delayed until EX, the pipeline may have already fetched the next instruction.
  - If the branch is taken, that instruction is wrong and must be **flushed**.
  - This introduces a **stall of 2 cycles** (minimum) in the pipeline.

Branch Handling in MIPS Pipeline

- **Why Branches Matter**
  - **Branches control program flow** → They decide which instruction executes next
  - **Next instruction depends on branch outcome** → Taken or not taken?
- **Pipeline Challenges**
  - **Instruction Fetch (IF) stage can't always predict correctly** → Branch decision isn't ready yet
  - **Branch resolution is done made in EX stage**

Branch Handling in MIPS Pipeline

- **Goal**
  - Avoid fetching wrong instructions
  - Minimize pipeline stalls
  - Improve instruction throughput
  - Enable smoother control flow

3. Do you **observe any stalls** or delay introduced to the pipeline with **Early** Branch Resolution (**at ID stage**)?

  - **Compare registers and compute branch target early** → Do it in **Instruction Decode (ID)** stage
  - **Add hardware support** → Enables faster branch resolution → Reduces control hazards

MIPS Branch Instruction [Assembly]

```
BEQ $t0, $t1, LABEL # Branch if $t0 == $t1
ADD $t2, $t2, $t3    # Next instruction(speculatively fetched)
LABEL: SUB $t4, $t4, $t5
```

3. Do you **observe any stalls** or delay introduced to the pipeline with **Early** Branch Resolution (**at ID stage**)?

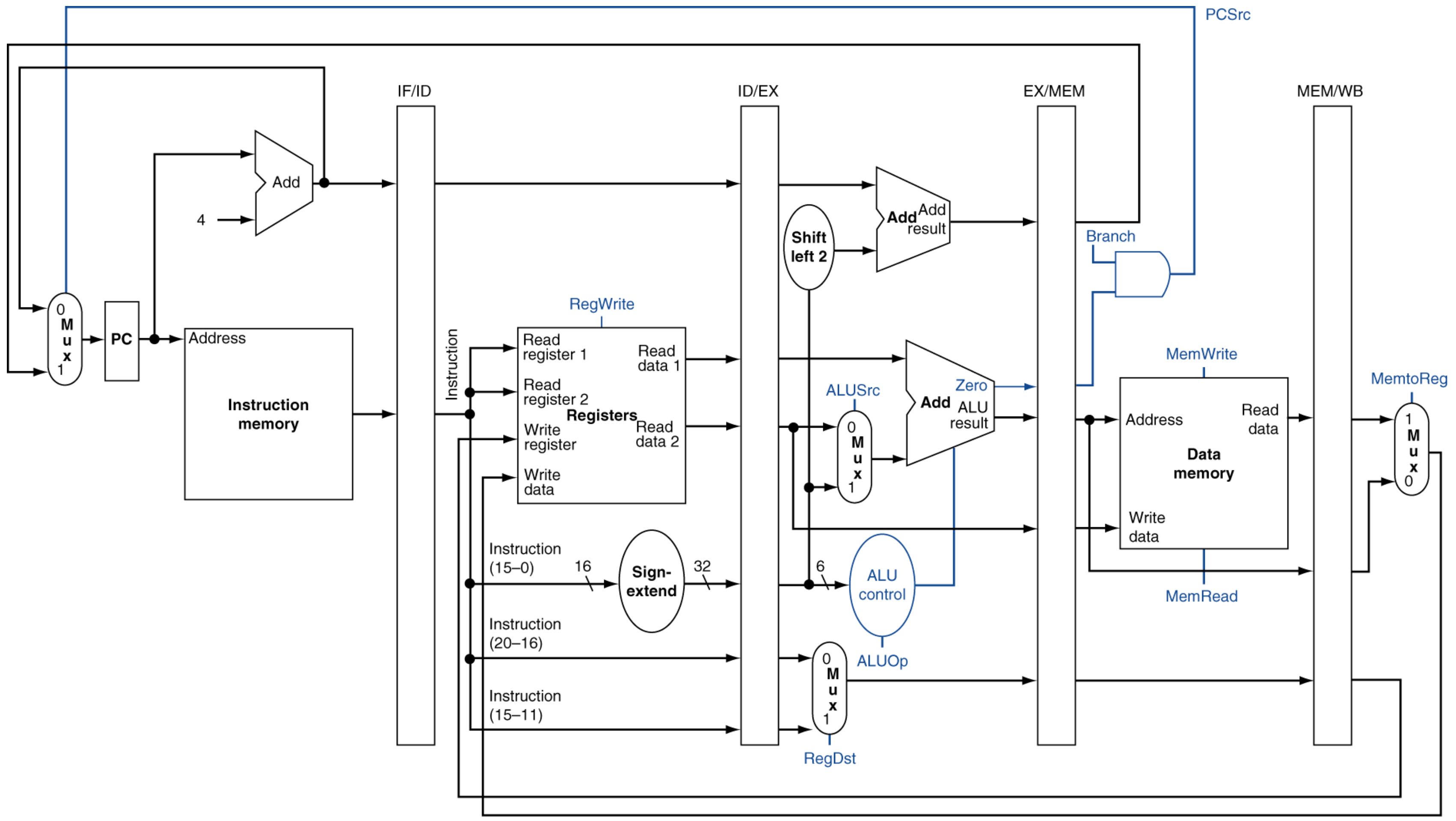
Ans: This introduces a **stall of 1 cycles** (minimum) in the pipeline.

- This comparison happens in the **ID stage**, but by then, the pipeline has

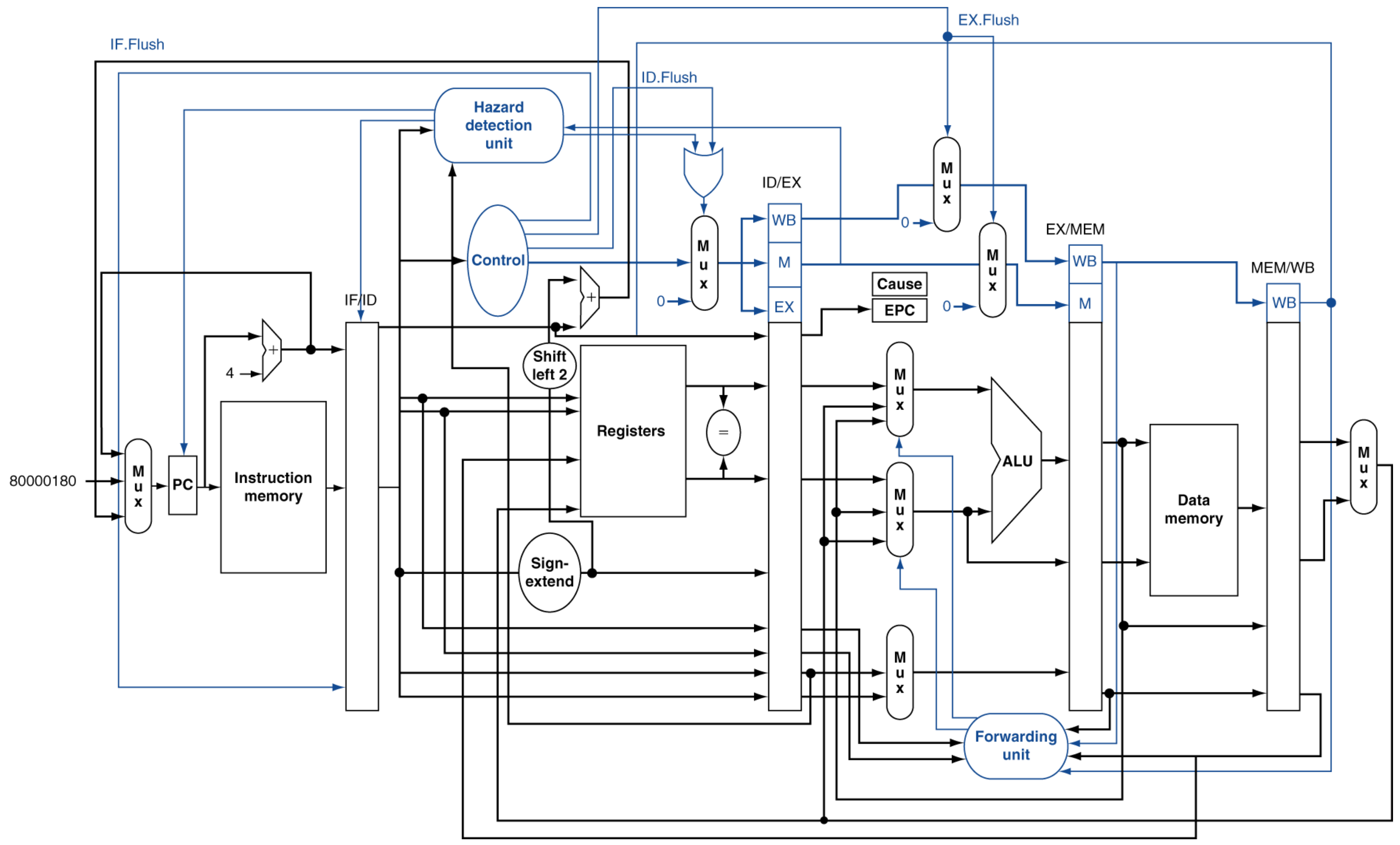
4. Can I reduce the stalls due to branch instruction even further i.e. eliminate it completely?

Ans: **1. Branch Delay Slot or 2. Branch Prediction** to keep pipeline busy

Pipeline (Simplified)



Pipeline (w/ early branch)



Branch Problem/Control Hazard in MIPS Pipeline

- **Branches resolved in EX stage ➡ 2-cycle penalty** for all branches [Taken/Not Taken]
- **Impact on CPI (Cycles Per Instruction)**
  - **Ideal CPI** = 1
  - **Branch instructions** = 32% of all instructions
  - **Penalty** = 2 cycles per branch
  - **New CPI** =

$$1+0.32\times 2=1.64 \text{ (OR) } 1 \times 0.68 + 0.32 \times 3$$

Solutions to Reduce Branch Penalty

- **Change Datapath**
  - Add hardware (e.g., adder) to compute branch target in **ID stage**
  - Resolve branch earlier → fewer stalls
- **Branch Delay Slot – Code reordering**
  - Fill delay slot with a **useful instruction**
  - Keeps pipeline busy while branch resolves
- **Branch Prediction**
  - **Guess** branch outcome to avoid stalling
  - **Static Prediction:** Same prediction every time (e.g., always taken)
  - **Dynamic Prediction:** Uses **runtime history** to predict based on past behavior

Control Hazards – branch delay slots

- In pipelined processors like MIPS, when a **branch instruction** is executed, the next instruction is fetched before the branch decision is resolved. This creates a **delay slot**—a one-cycle gap where the processor could be idle.

- To avoid wasting that cycle, compilers try to **fill the delay slot** with a **useful instruction that will execute regardless of whether the branch is taken or not.**

Example: Branch Delay Slot Filling

```
ADD R3, R4, R5
BEQ R1, R2, LABEL    ; Branch if R1 == R2
LABEL: SUB R6, R7, R8
```

Without Delay Slot Filling: Do you see any wasted cycle?

Example: Branch Delay Slot Filling

```
ADD R3, R4, R5
BEQ R1, R2, LABEL    ; Branch if R1 == R2
LABEL: SUB R6, R7, R8
```

Without Delay Slot Filling: Do you see any wasted cycle?

Ans: Just by re-ordering the instructions….
If the branch is taken, the ADD instruction is fetched and then discarded. --→. Wasted cycle.

Example: Branch Delay Slot Filling

```
ADD R3, R4, R5
BEQ R1, R2, LABEL    ; Branch if R1 == R2
LABEL: SUB R6, R7, R8
```

With Delay Slot Filling: Do you see any wasted cycle?

Delay Slot Introduced:

```
BEQ R1, R2, LABEL
NOP                  ; Delay slot (wasted if not filled)
LABEL: SUB R6, R7, R8
```

Example: Branch Delay Slot Filling

```
ADD R3, R4, R5
BEQ R1, R2, LABEL    ; Branch if R1 == R2
LABEL: SUB R6, R7, R8
```

With Delay Slot Filling: Do you see any wasted cycle?

Delay Slot Filled [Pipeline Optimization]:

```
BEQ R1, R2, LABEL
< Which Instruction can I fill in here? >
LABEL: SUB R6, R7, R8
```

Control Hazards – branch delay slots

- **Branch delay slot filling:**

move an instruction into the slot right after the branch, hoping that its execution is necessary.

Limitations: restrictions on which instructions can be rescheduled, compile time prediction of taken or untaken branches.

Feature	Non-Delayed Branch	Delayed Branch
What is means?	The processor waits to **resolve the branch before executing the next instruction**. This can cause pipeline stalls and slow down performance.	The instruction immediately after the branch (in the ***delay slot***) is always executed, regardless of the branch outcome. This avoids stalls and improves efficiency if the slot is filled wisely.
Execution Model	Stall until branch resolved	Execute next instruction regardless
Pipeline Impact	Causes stalls	Avoids stalls with delay slot
Compiler Involvement	Minimal	Required for scheduling
Performance	Lower due to stalls	Higher if delay slot is used well
Used in	Common in modern CPUs	Rare in modern CPUs, used in older RISC. Modern CPUs rarely use delay slots. Instead, they rely on: **- Branch prediction** **- Speculative execution** **- Out-of-order execution**

Example Nondelayed vs. Delayed Branch

Nondelayed Branch

```
or    M8, M9 ,M10

add M1 ,M2,M3

sub M4, M5,M6

beq M1, M4, Exit

xor M10, M1,M11

Exit:
```

Delayed Branch

```
add M1 ,M2,M3

sub M4, M5,M6

beq M1, M4, Exit

or    M8, M9 ,M10

xor M10, M1,M11

Exit:
```

Branch Delay Slot