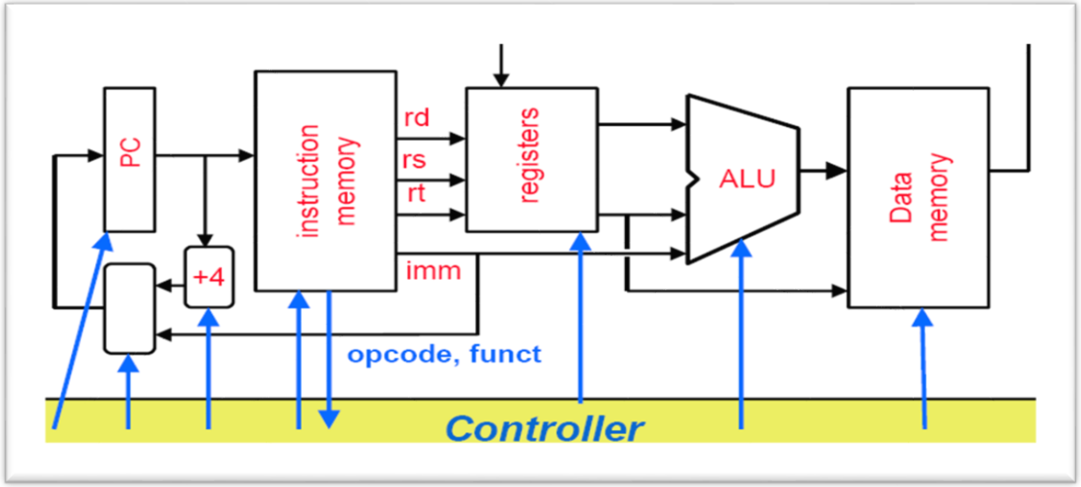
Lecture 12: Pipelining

For a program with **100 billion** instructions, Execution Time = ?

```
…
add $s0, $s2, $s3
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
…
```

Topics

- Introduction to pipelining
  - Performance
- Pipelined datapath
  - Design issues
- Hazards in pipeline
  - Types
  - Solutions

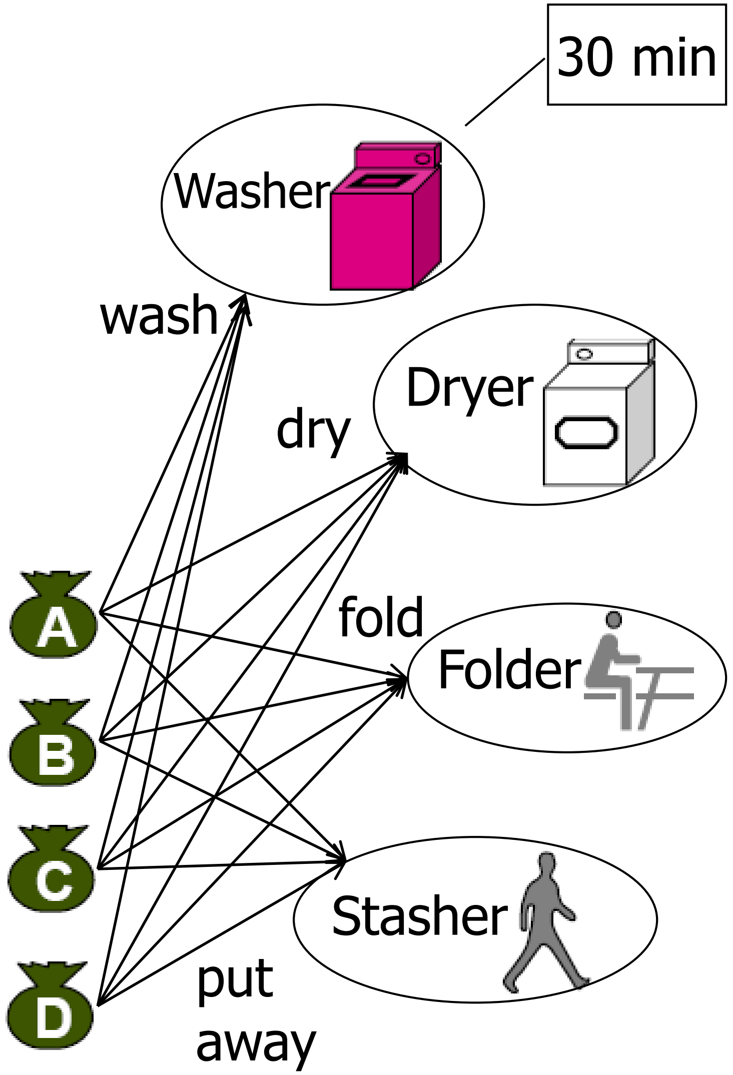


Pipelining is Natural! Laundry Example

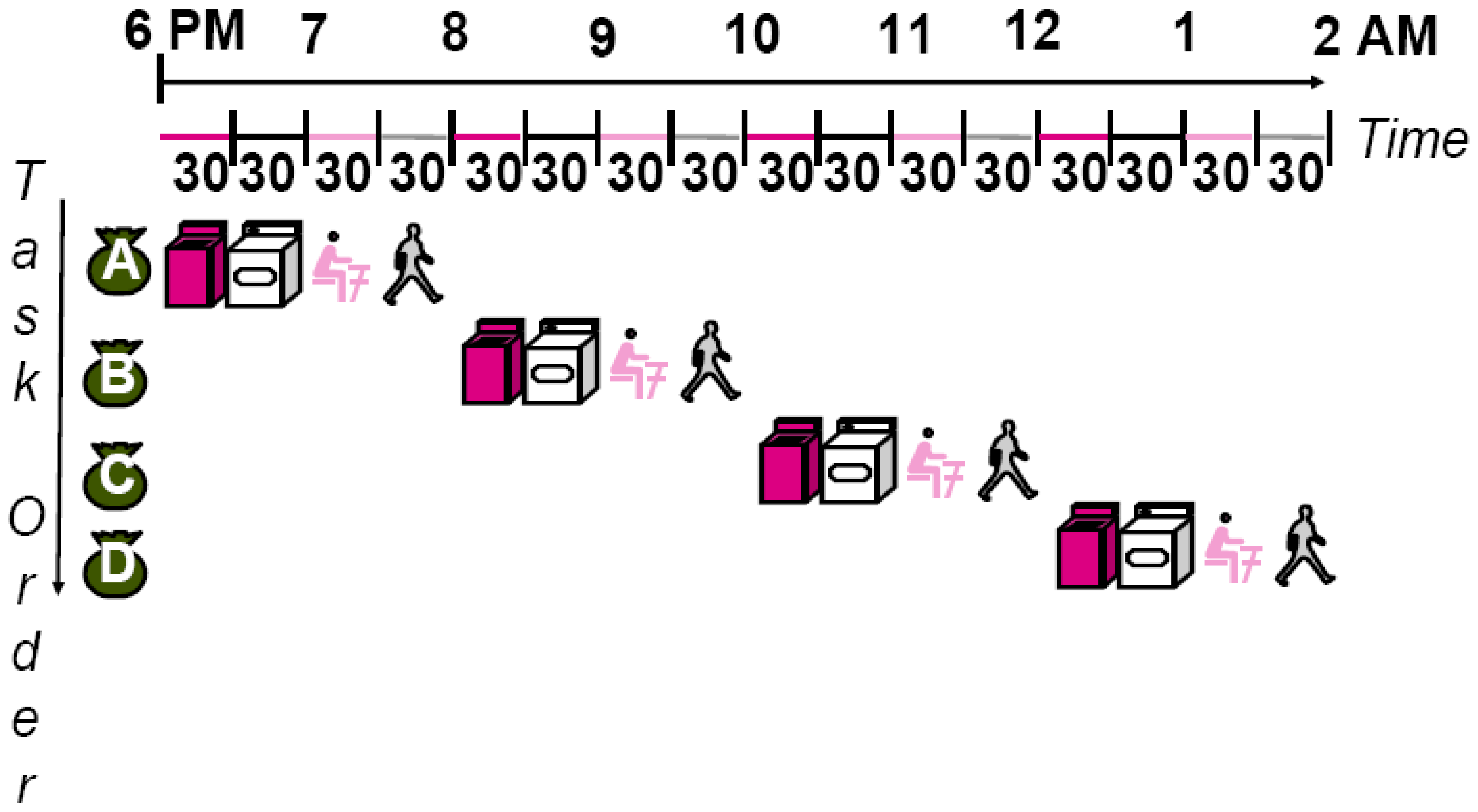
- Use case scenario
  - **Ann**, **Brian**, **Cathy**, **Dave**
  - each has one load of clothes to **wash**, **dry**, **fold**, and **put away**
- Load parameters

	Step	Device	Cycle
1	Wash	Washer	30 min
2	Dry	Dryer	30 min
3	Fold	Folder	30 min
4	Put away	Stasher (put clothes into drawers)	30 min

- Efficient organisation?

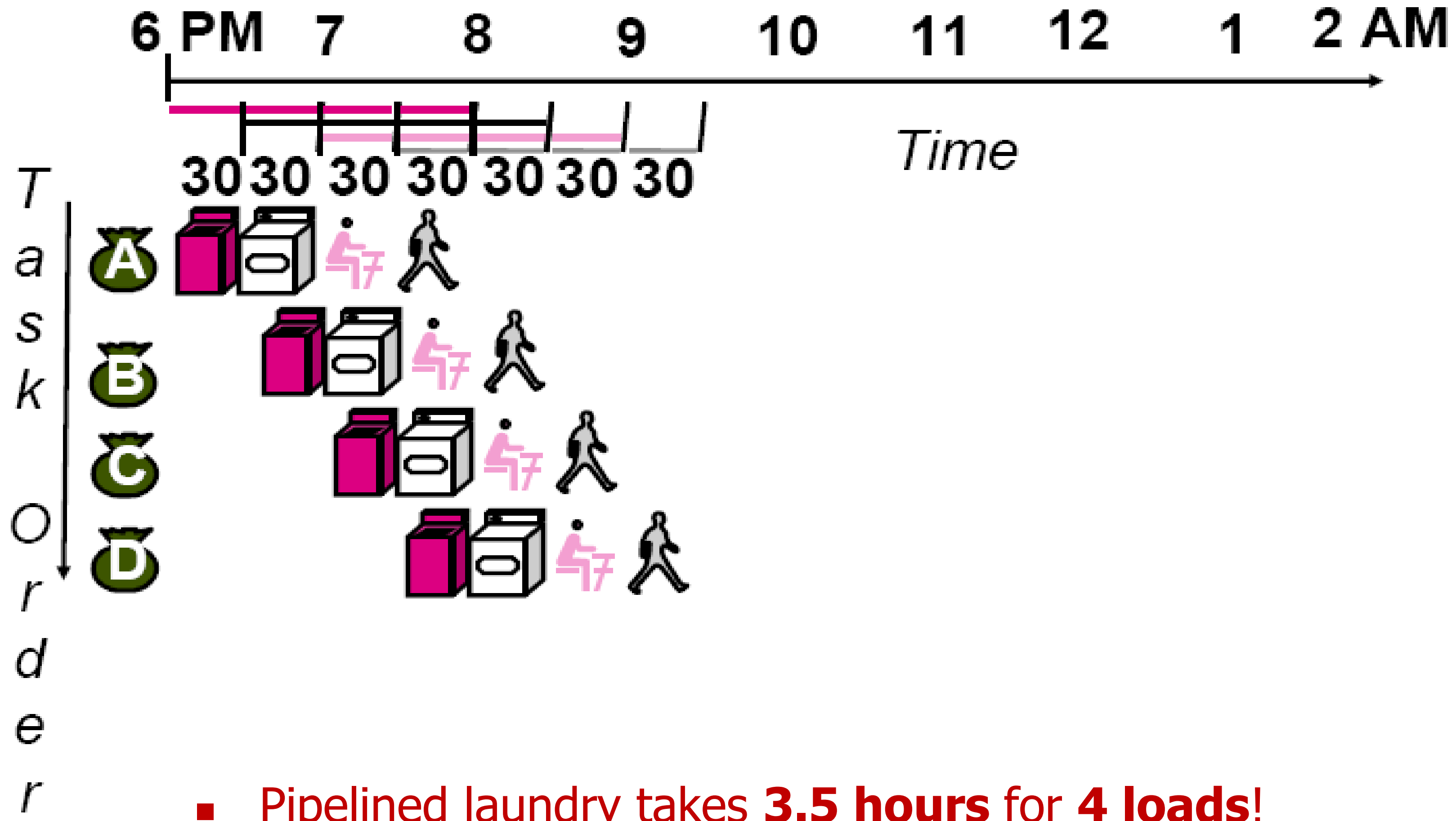


Sequential Laundry



- Sequential laundry takes **8 hours** for **4 loads**

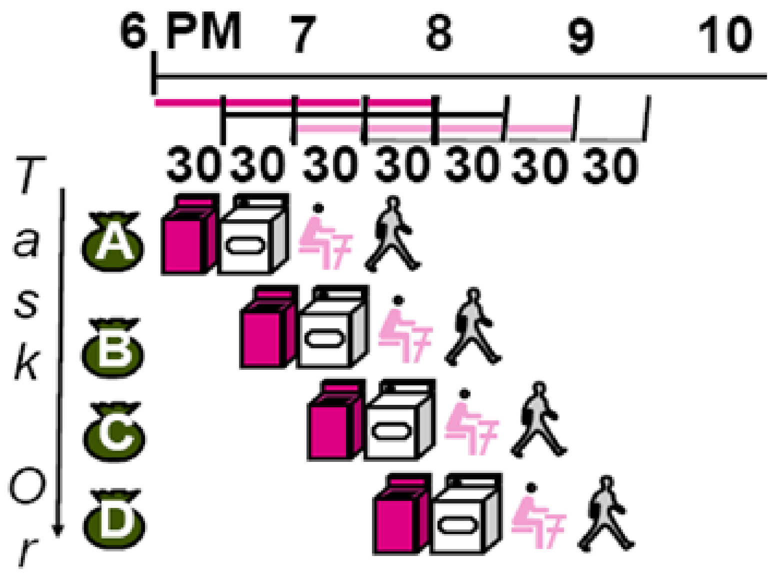
Pipelined Laundry: Start work ASAP



- Pipelined laundry takes **3.5 hours** for **4 loads**!

Pipelining Lessons (Task, Resource)

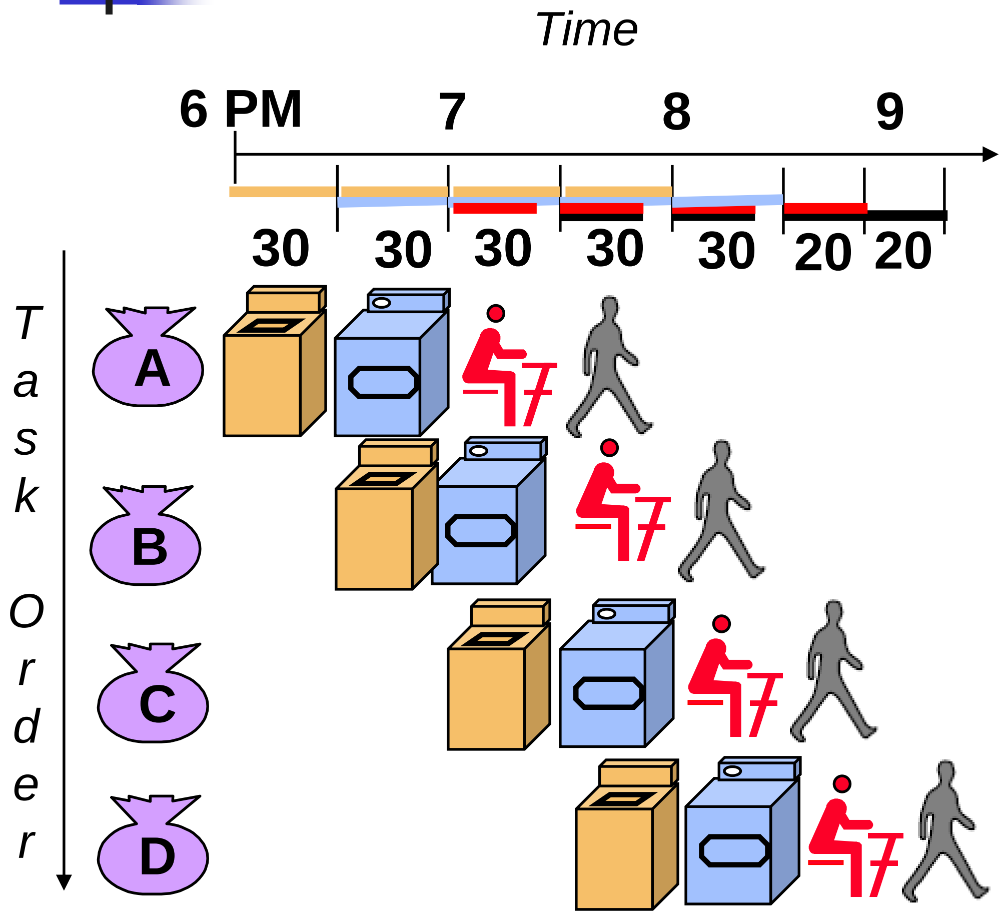
- A way to parallelize: a load consists of **multiple** sub-tasks operating simultaneously using **different** resources (hardware equipments)
- Pipelining in parallel: a load is processed over multiple steps (called pipeline "stages" or "segments") using corresponding tools in parallel
- Pipelining doesn't help **latency** of single task; it helps **throughput** of entire workloads



{Wash, Dry, Fold, Stash}

- Potential speedup = **Number of pipe-stages** (Pipeline depth)
  - Ideal could be: **8hrs/2hrs=4**
- Time to "**fill**" pipeline and time to "**drain**" reduce speedup:
  - Speedup in this example: **8hrs/3.5hrs=2.3**

Pipelining Lessons (Time cycles)



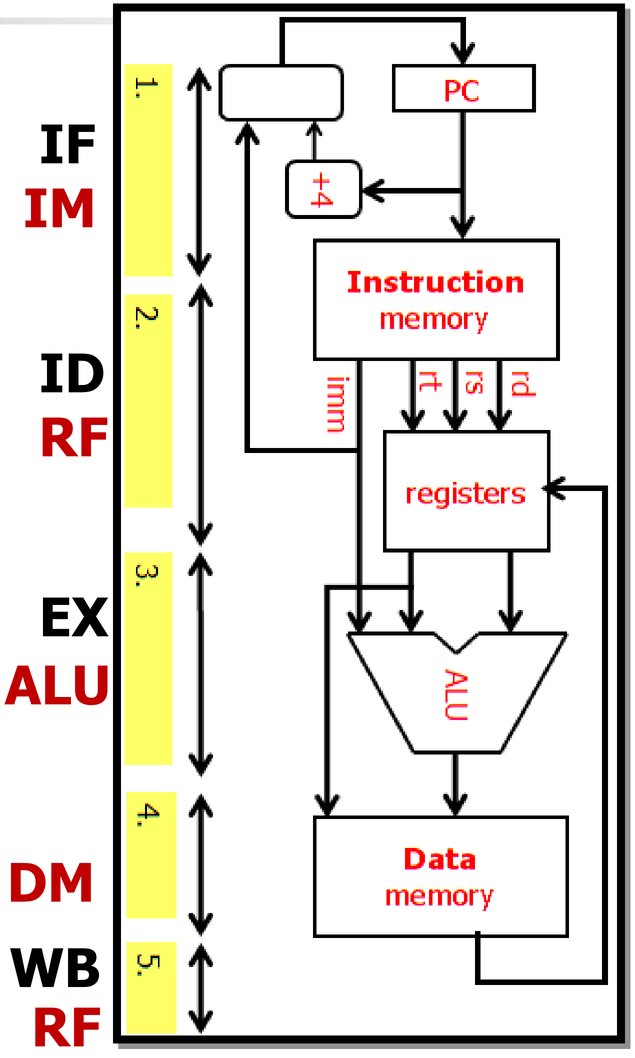
- Suppose new Folder takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

  /* see the last slide for answer */

- Pipeline rate is limited by the **slowest** pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

Steps in Executing MIPS instruction
instruction stages and hardware resources

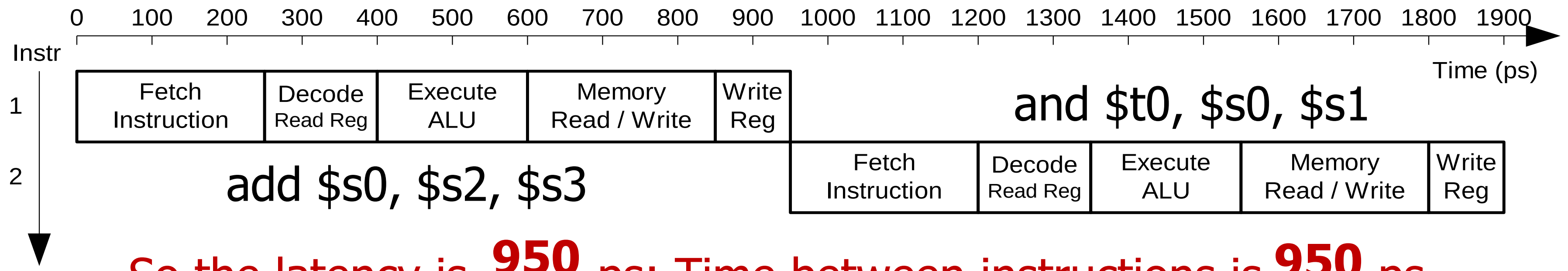
1) **IF**: Instruction Fetch (PC -> memory)
   Hardware: **I**nstruction **M**emory (**IM**)
2) **ID**: Instruction Decode (Read Registers)
   Hardware: **R**egister **F**ile [read] (**RF**)
3) **EX**: Execute (Perform Operation)
   Hardware: **ALU**
4) **MEM**: Memory Access (for data)
   Hardware: **D**ata **M**emory (**DM**)
   - Load: Read Data from Memory
   - Store: Write Data to Memory
5) **WB**: Write Back (Write Data to Register)
   Hardware: **R**egister **F**ile [write] (**RF**)



To simplify pipeline, every instruction takes the same pipeline depth
(even if not needed; unused stages are NOP.); every stage has the same length.

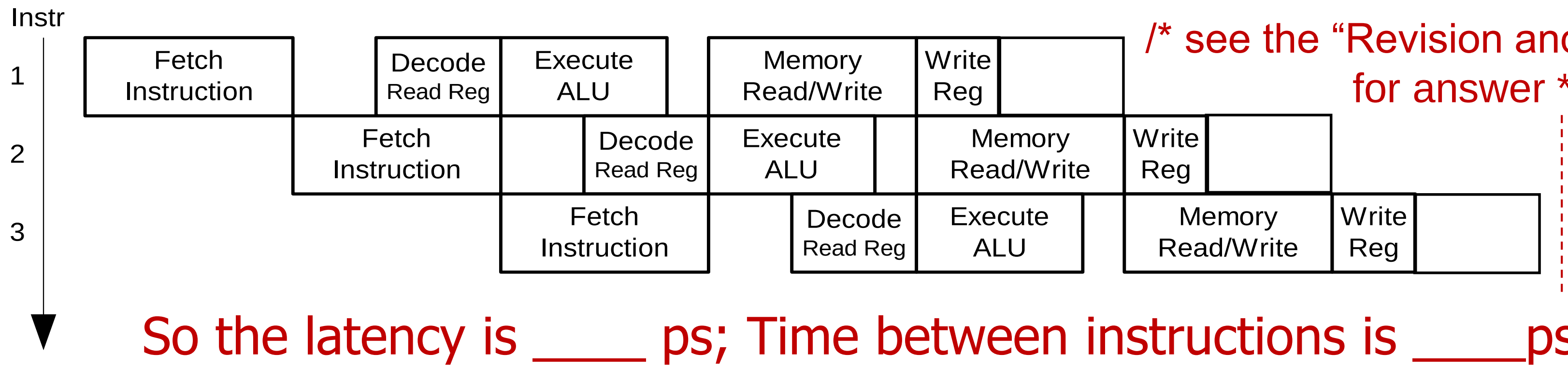
Pipelined Execution Representation

Sequential



So the latency is **950** ps; Time between instructions is **950** ps

Pipelined



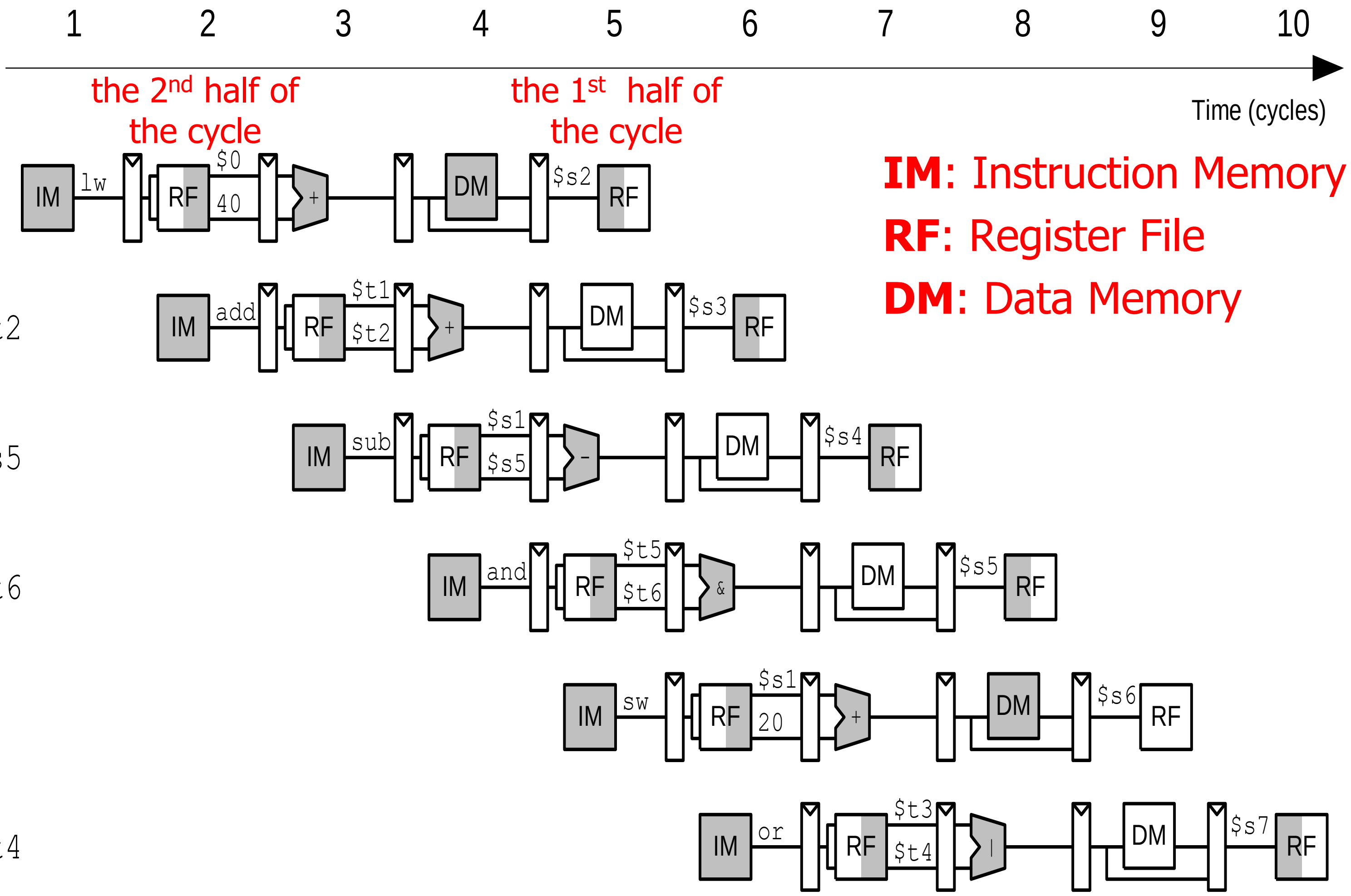
/* see the "Revision and quiz" slide for answer */

So the latency is _____ ps; Time between instructions is _____ps

- **Pipeline stages not balanced perfectly:**

  instructions take the same pipeline depth; some instructions have NOP stages; every stage has the same length as the longest.

Pipelining Abstraction
instruction stages and hardware resources



```
lw  $s2, 40($0)
add $s3, $t1, $t2
sub $s4, $s1, $s5
and $s5, $t5, $t6
sw  $s6, 20($s1)
or  $s7, $t3, $t4
```

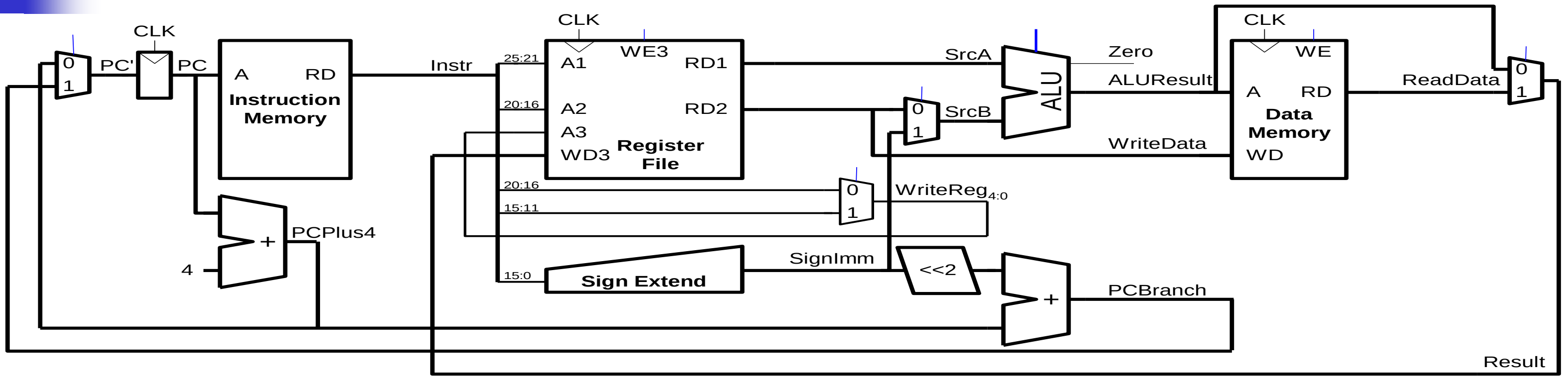
IM: Instruction Memory

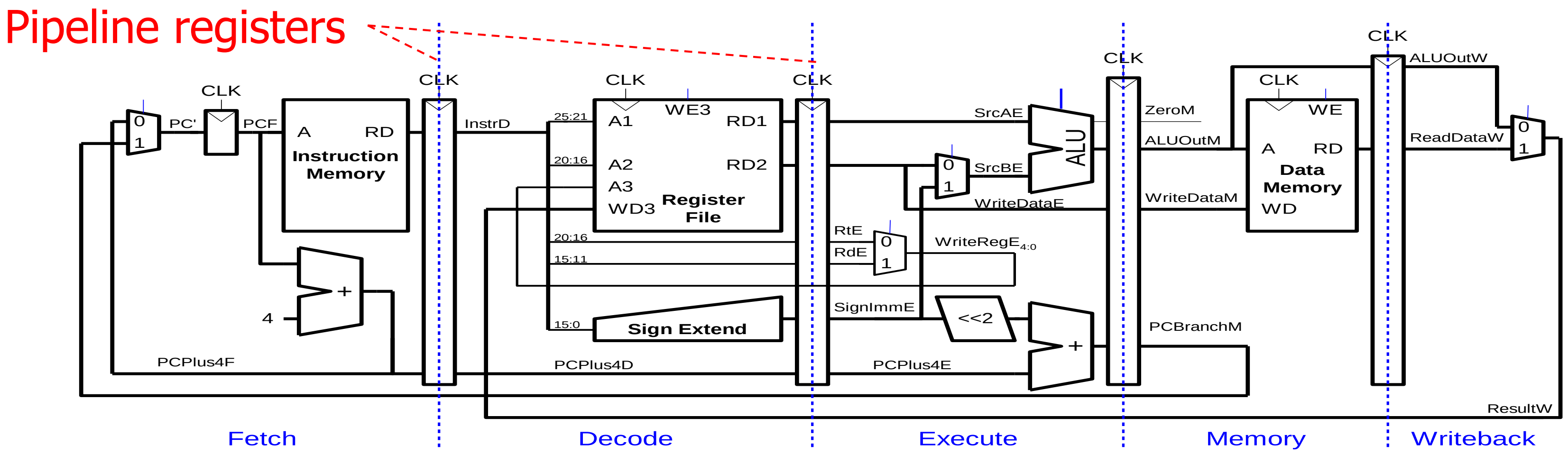
RF: Register File

DM: Data Memory

To simplify pipeline, every instruction takes the same pipeline depth (even if not needed; unused stages are NOP.); every stage has the same length.

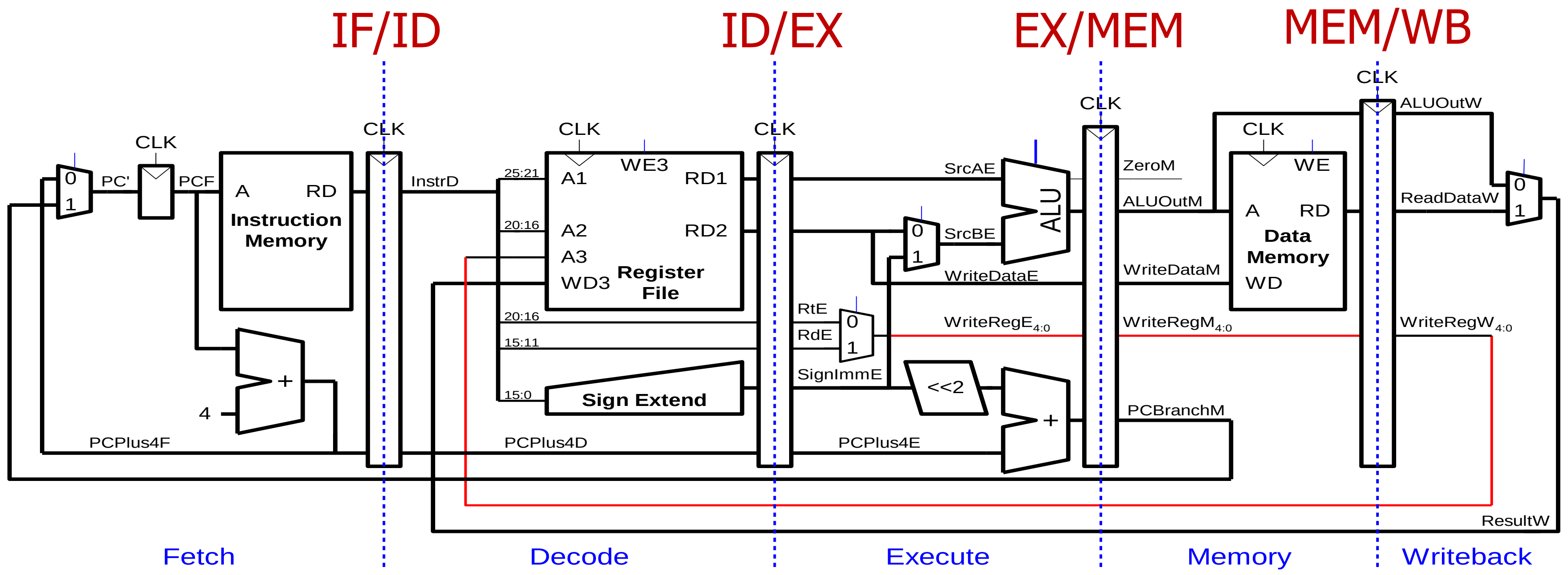
Pipelined Datapath





- State transition: need for propagating state at the end of every **stage** to push the instruction through the pipeline.
- Pipeline registers: add registers between every **two stages** (used as state elements)

Pipeline registers

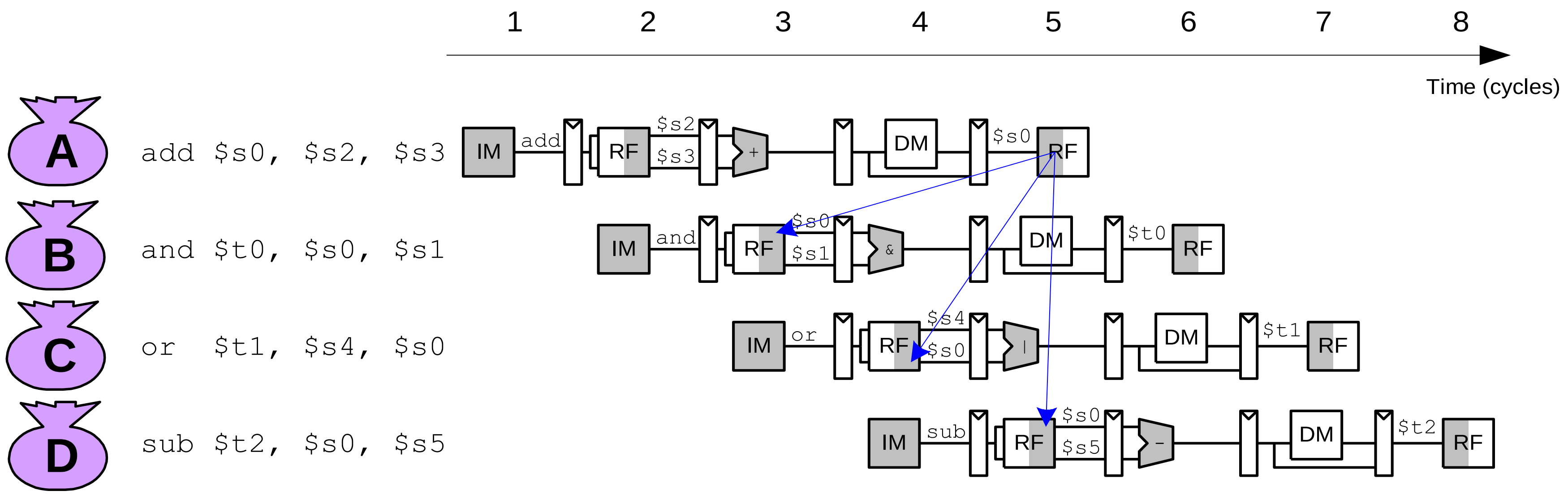


- Pipeline registers separate pipeline stages
- Names of registers: **names of two stages they separate**

  IF/ID, ID/EX, EX/MEM, MEM/WB. Why no register WB/IF?
- Wide enough to store all **data** (and **control** signals) passed between stages

Problems for Pipelining

- **Limits to pipelining**: hazards prevent next instruction from executing during its designated clock cycle



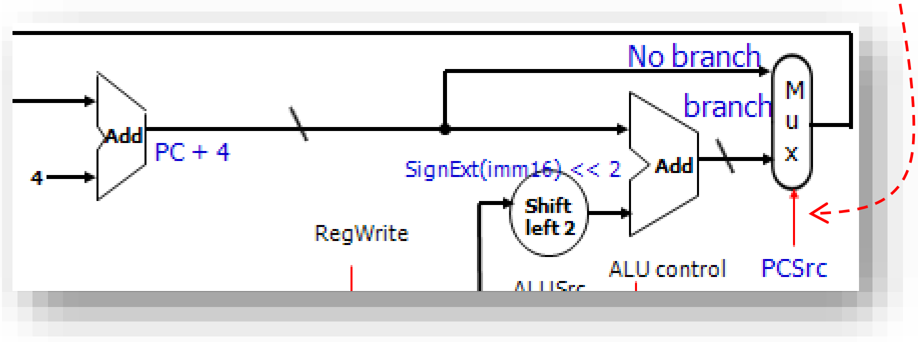
- Data hazard: **B, C, D** depends on the result of **A.**
- Possible solutions to data hazard :

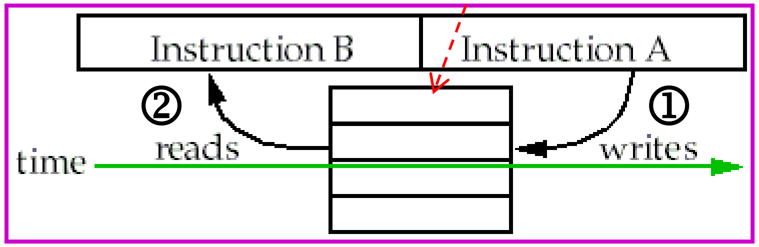
Static	Dynamic
Insert nops in code at compile time Rearrange code at compile time	Forward data at run time Stall the processor/pipeline at run time

Pipeline Hazards (conflicts)

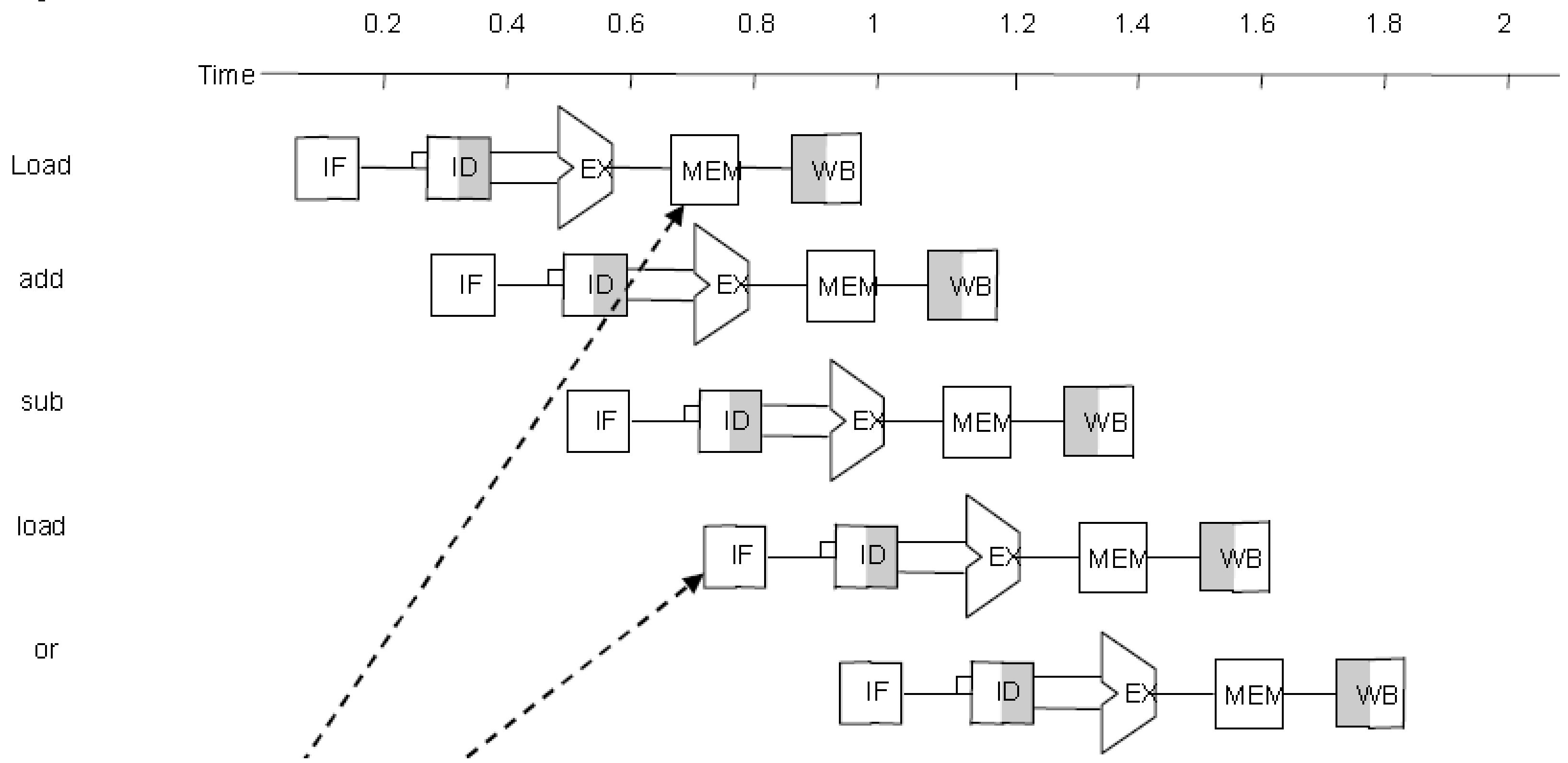
- **Limits to pipelining**: hazards prevent next instruction from executing during its designated clock cycle

Data hazards [intermediate input]	Instruction **depends on the result** of prior instruction that is still in the pipeline
Structural hazards	Attempt to use the **same resource** in two different ways at the same time; hardware cannot support a particular combination of instructions.
Control hazards [decision making]	Pipelining of **branches** & other **control instructions** stall the pipeline until the next value of PC is known





Single Memory is a Structural Hazard



- Each instruction is fetched from memory
- 30% of instructions are load or store
- Not possible to access same memory twice in same clock pulse

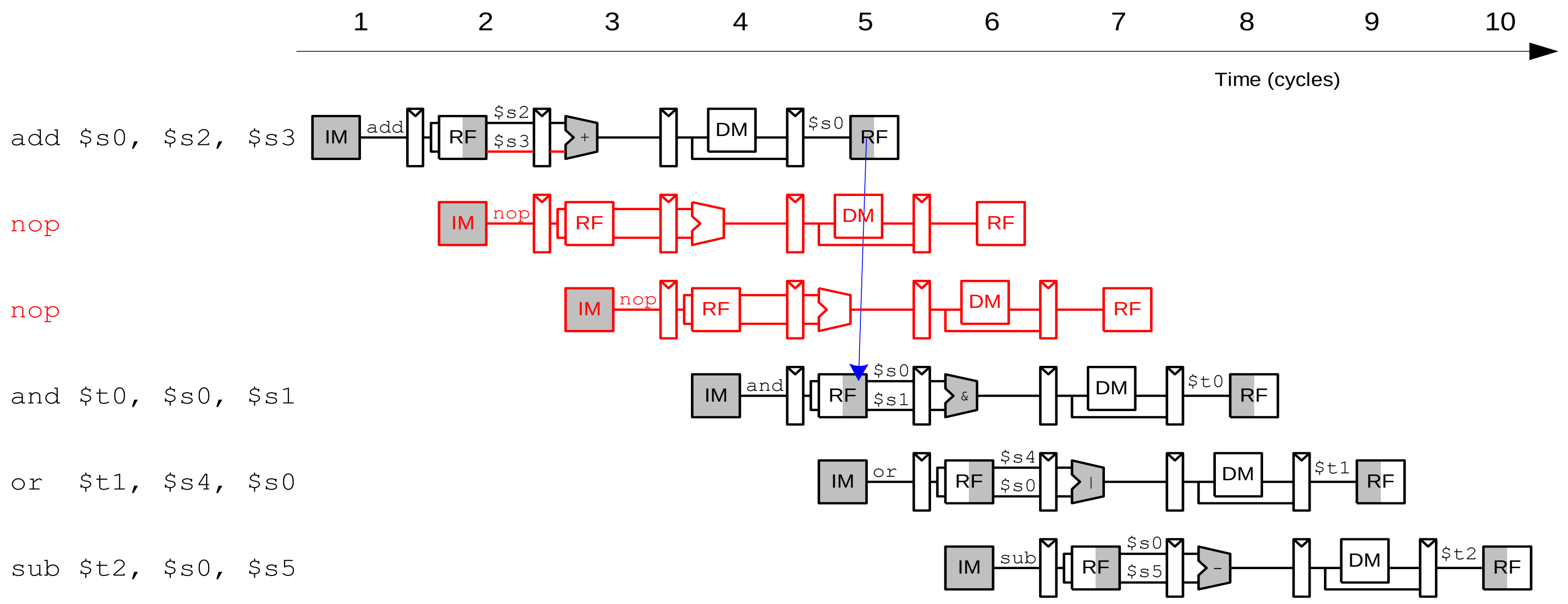
Structural Hazards and Solutions

/* see the “Revision and quiz” slide for answer */

- Example:
  - Each instruction is fetched from memory = _______ memory access
  - 30% of instructions are load or store = _______ memory access
  - Total no. of accesses calculated per cycle = _____
  - Total no. of accesses expected per cycle = ___ no duplicated rsc
- Solution:
  - level 1 cache is split into:
    - instruction cache: **I-Cache**, and
      each instruction needs one read from I-cache
    - data cache: **D-Cache**
      30% instructions need access to D-cache
  - we can maintain CPI of 1

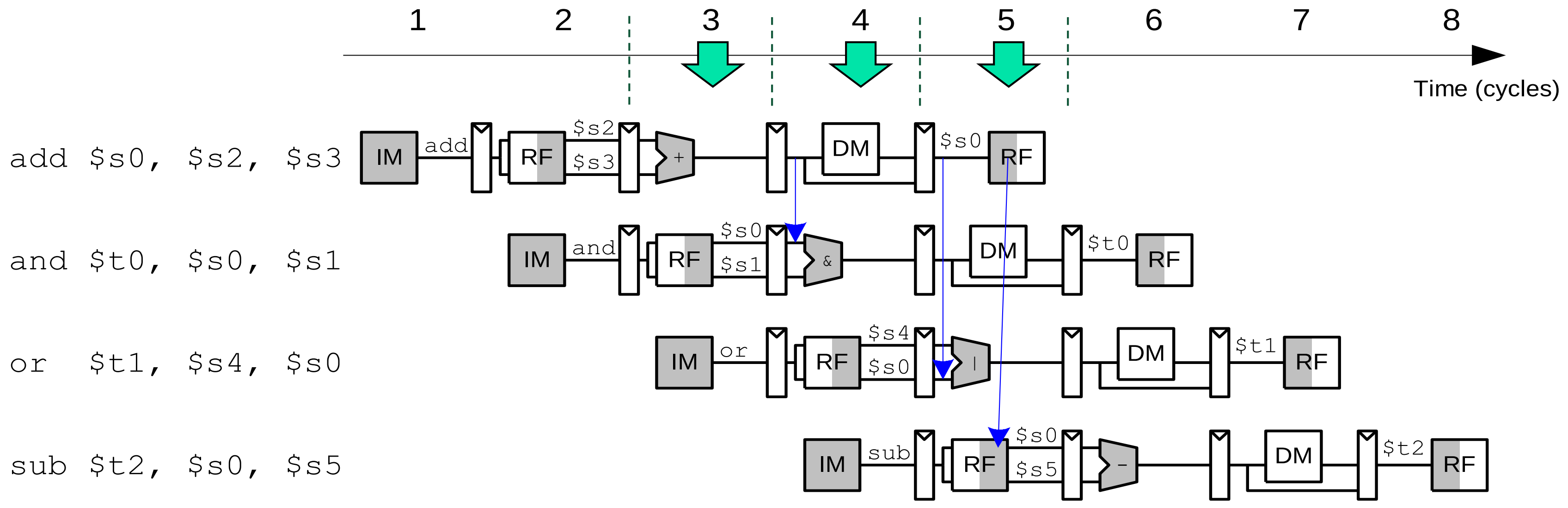
Data hazards

- Later instructions use the results of the earlier instructions (attempt to use an item before it is ready)
- Solution: Compile-Time Hazard Elimination
  - Insert enough nops for result to be ready
  - Or find other instructions which will logically fit between add and subsequent instructions (move independent instructions forward)



Data hazards

- Solution: Dynamic Hazard Elimination
  - Data forwarding: the result can be passed directly to the functional unit which needs it.

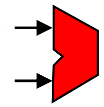


/* see the "Revision and quiz" slide for answer */

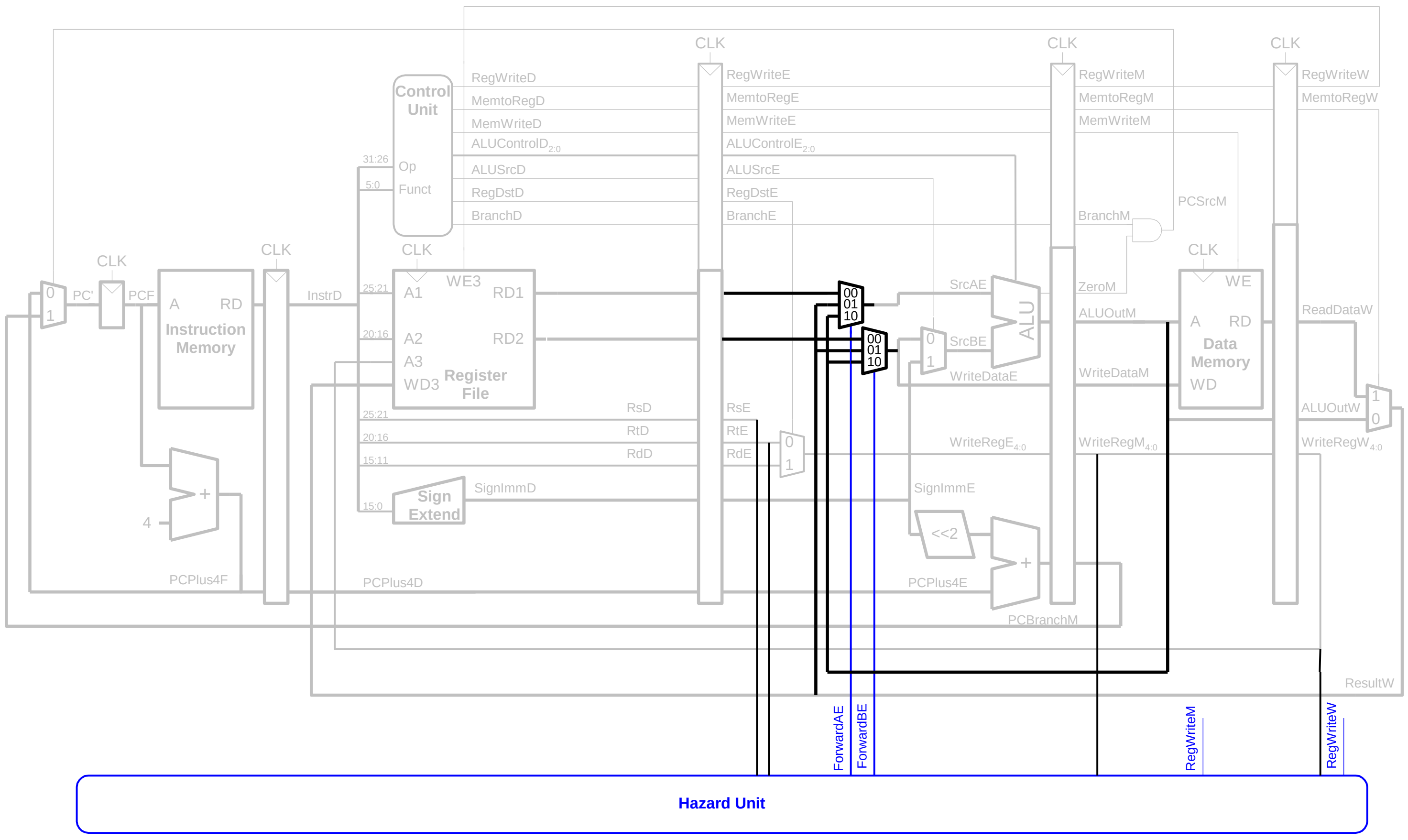
- Issue: The subsequent instructions need this value $s0 earlier
- ADD gets value of $s0 after which stage? ____________________
- AND needs before its stage ______; OR in stage ______; SUB in stage ______

Data Forwarding

add **$s0**, $s2, $s3
and $t0, **$s0**, $s1
… (rd) (rs) (rt)

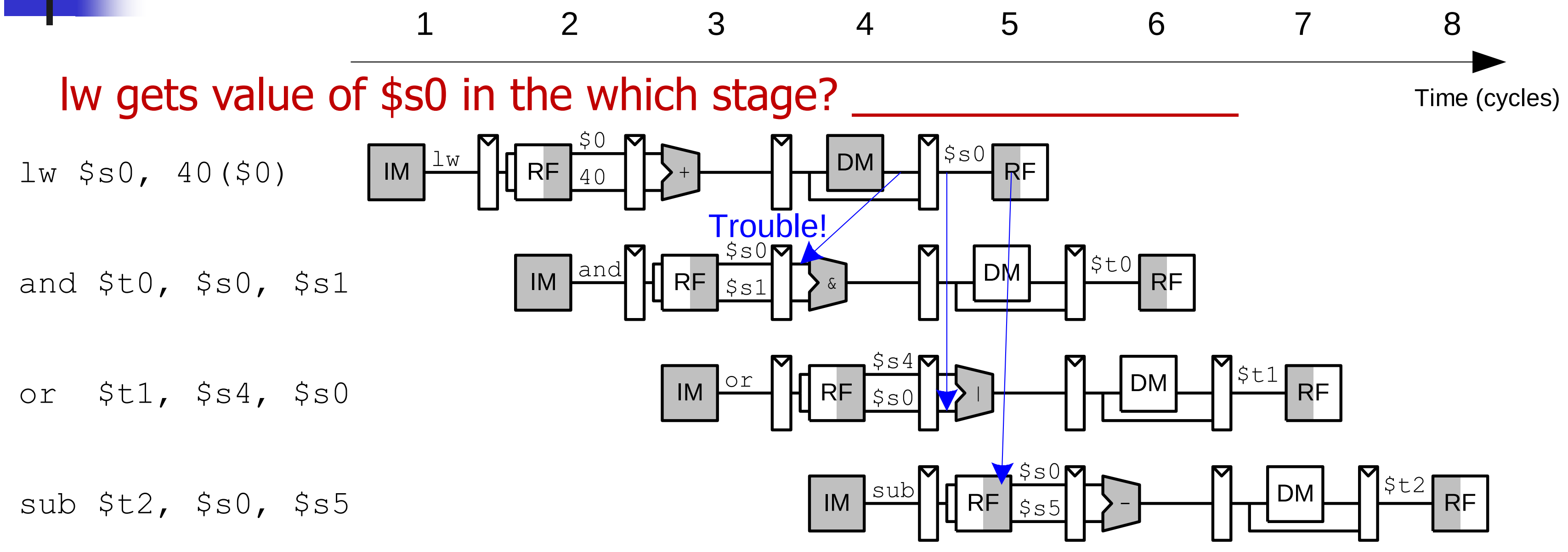


- Forward result to EX stage of subsequent instructions from either:
  - Memory stage or Writeback stage of ADD

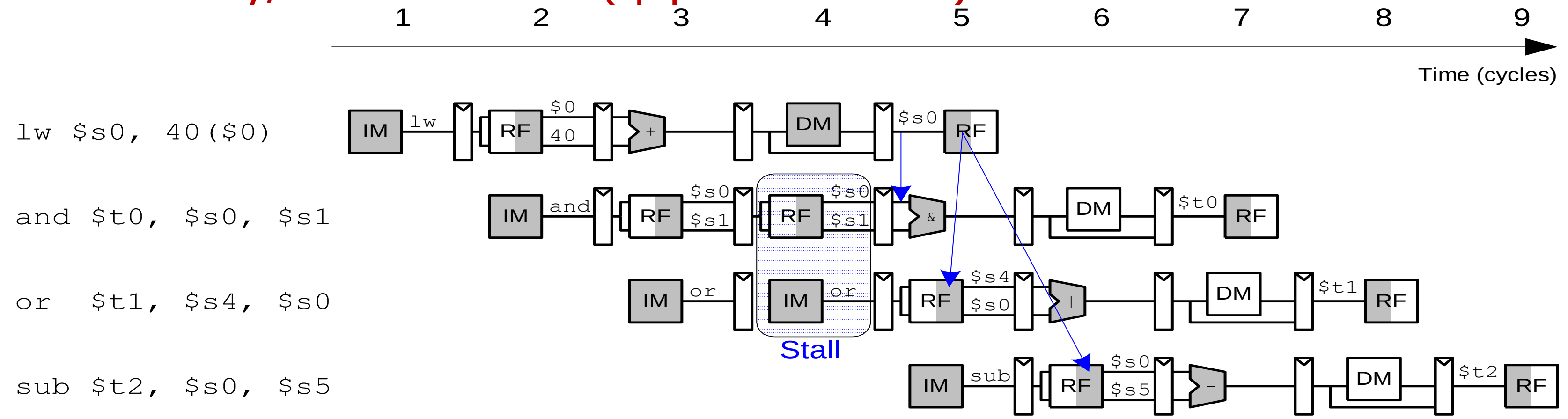


Forwarding can fail where **result** is not ready

lw gets value of $s0 in the which stage? ________________



Must delay/stall instruction ("pipeline bubble")



Data hazards in Memory

- So far we considered data hazards limited to registers
- There may also be data hazards in memory
  - A cache miss may delay load or store instruction
  - Store and load instructions on the same address too close in time
    - Name dependency: only happens to use the same name
- Three categories: Classification depending on order of reads and writes:
  - RAW read after write
  - WAR write after read
  - WAW write after write

What about RAR read after read?

- Solutions
  - some machines enforce memory accesses strictly in program order
    - this affects pipeline performance
  - some allow a different order
    - in general more hardware is required for control

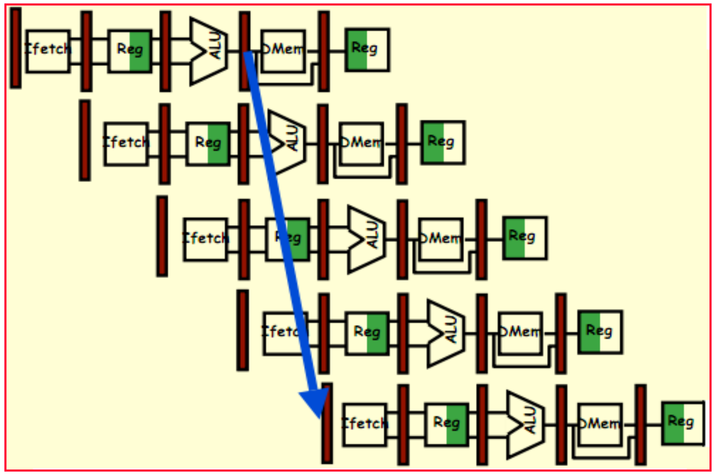
Data hazards in Memory: RAW, WAR

- **RAW**: Write may not complete before read is attempted
- This is the most common type of data hazard
  - It is created by the logic of the program
  - Obviously a value must be computed before it is used.
- Forwarding can be used to overcome this hazard

- **WAR**: the location is overwritten before read is completed
- This type of data hazard is rare (Antidependences)
  - This can only occur if results are written early in the pipeline and operands are read late in the pipeline. (only complex addressing modes require operands to be read late in the pipeline.)
- Register renaming can be used to overcome this hazard
  - A large no of architecture registers reduces name dependencies.

Control hazards

- Instructions in the pipeline are fetched in order of the program
  - This flow would change when there is a **branch**, or a **procedure call**
- **Branch penalty**: The pipeline has to stall until the address of the next instruction is determined
  - Resolve the condition of the branch
    - conditional branch only
  - Calculate branch address
    - conditional branch
    - unconditional branch
    - return from procedure



- In our pipeline
  - When is branch address calculated? __ID stage (stage **2**)__
  - When is condition evaluated? __EX stage (stage **3**)__
  - What resources are used? __[Adder], ALU__

Control hazards

- Solutions
  - [**Stall when branch decoded**]: 3 cycles lost per branch! (too much waste as there are ~30% branch instructions)
  - **Delayed branches**: execute instructions not dependent on branch direction before the address is resolved
  - **Predict Branch Not Taken (e.g. for selections)**:
    - 47% branches not taken on average
    - Execution continued down the sequential path
    - If branch taken, "squash/cancel" instructions in pipeline
  - **Predict Branch Taken (e.g. for loops)**:
    - 53% MIPS branches taken on average
    - Can't proceed until we know the branch address (**ID** stage)
  - **Sophisticated branch prediction**: execute the "more likely" stream of instructions, and undo if guessed wrongly

Control hazards

- Delayed branches
  - Compilers move instruction that has no conflict with branch into delay slot
    - Fills about 60% of branch delay slots
    - About 80% of instructions executed in branch delay slots useful in computation
    - 48% (60% x 80%) of slots usefully filled
  - Where to get branch delay slot instructions?
    - Before branch instruction
    - From fall through

      only valuable when branch-not-taken
    - From the target address

      only valuable when branch-taken

```
Original sequence
        add    $4 $5 $6
        beq    $1 $2 40
Reordered to this
        beq    $1 $2 40
        add    $4 $5 $6
```

Control hazards

- Predict-not-taken branch
  - Execute successor instructions in sequence
    - PC+4 already calculated, so use it to get next instruction
    - This scheme is simplest to implement
  - "Squash" instructions in pipeline if branch taken
    - the instructions in IF, ID and EX stage are discarded
    - nothing has been written so far

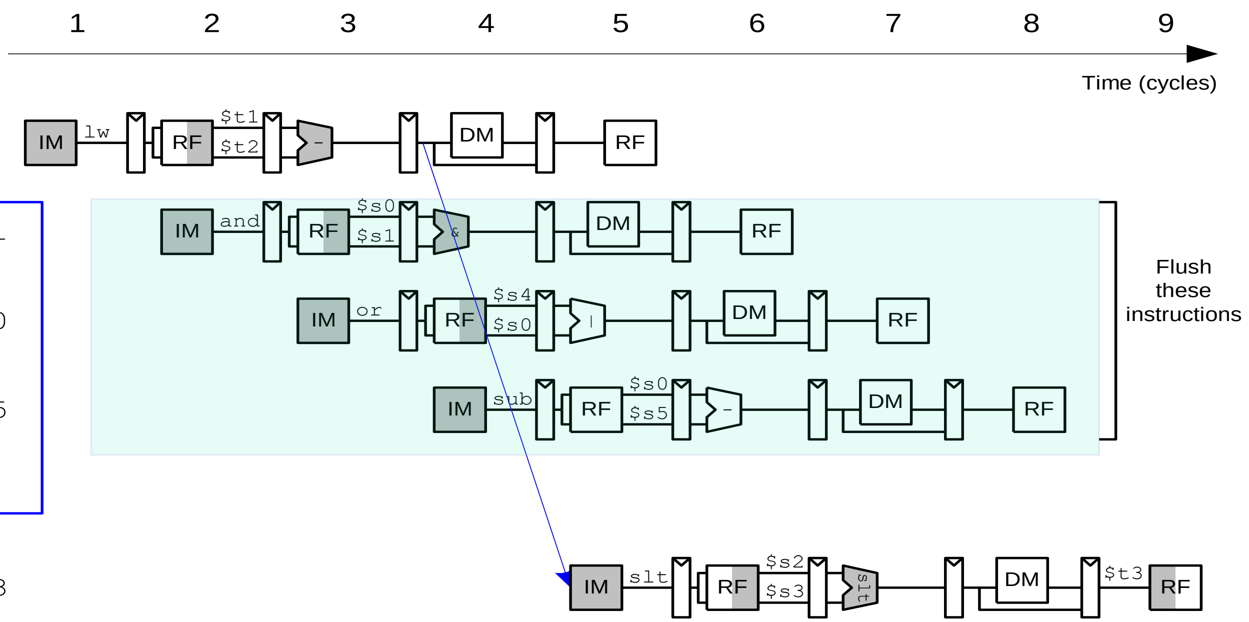
```
20   beq $t1, $t2, 40

24   and $t0, $s0, $s1

28   or  $t1, $s4, $s0

2C   sub $t2, $s0, $s5

30   ...
...
64   slt $t3, $s2, $s3
```



Control hazards

- Predict-taken branch (Early Branch Resolution)
  - Can’t proceed until we know the branch address (**ID** stage)
    - MIPS still incurs one cycle branch penalty
      [Penalty now is only one lost cycle]
  - Makes sense if
    - branch conditions are more complex and take longer to calculate

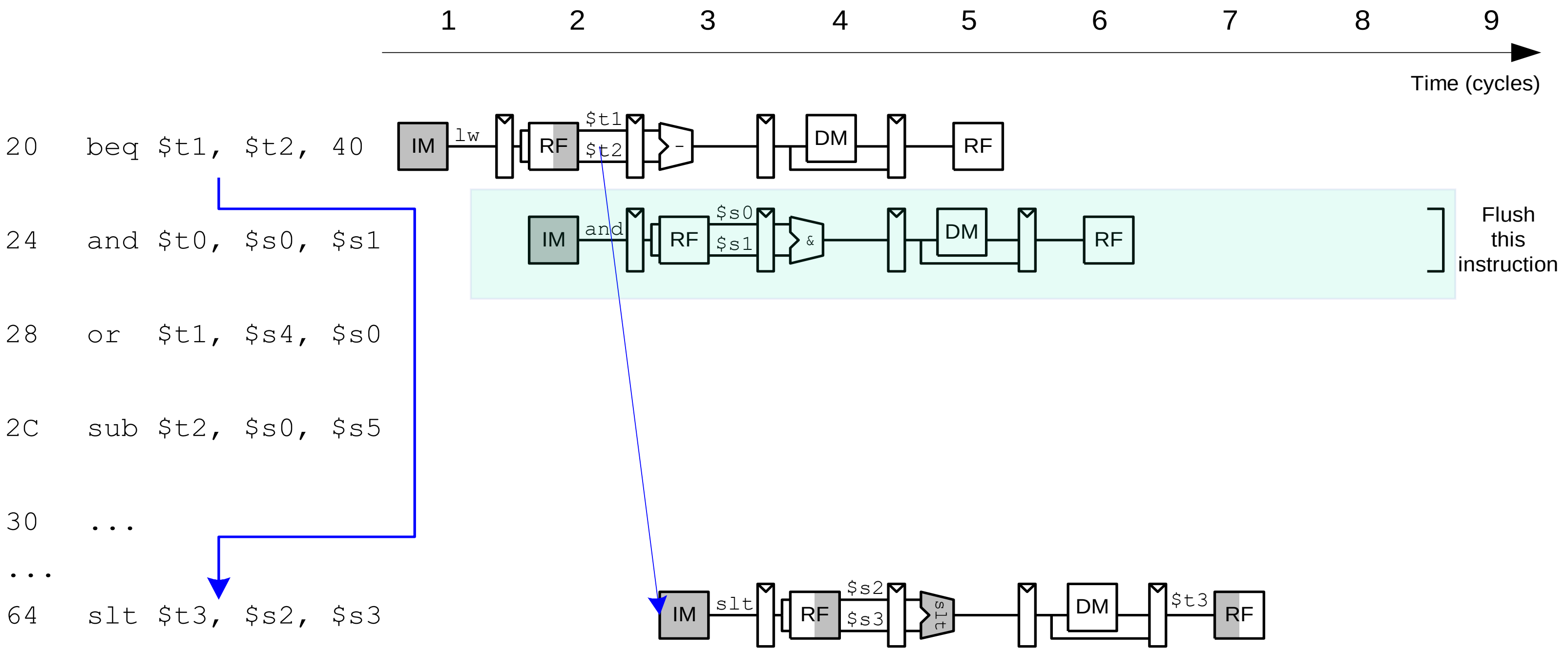
```
20   beq $t1, $t2, 40

24   and $t0, $s0, $s1

28   or  $t1, $s4, $s0

2C   sub $t2, $s0, $s5

30   ...
...
64   slt $t3, $s2, $s3
```

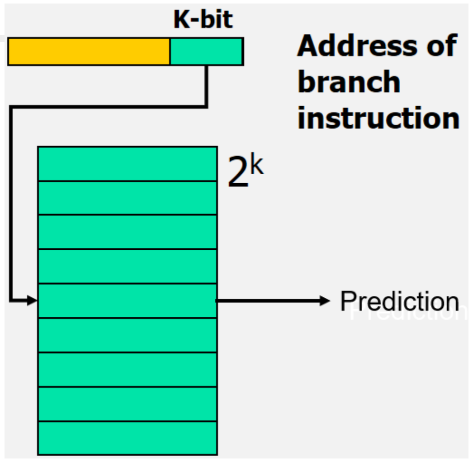


Control hazards

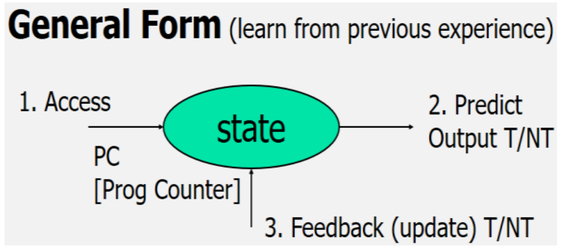
- **Static branch prediction**
  - can be performed by the compiler by hinting
    - all forward branches not-taken
    - all backward branches (loops) as taken
    - unrolling loops to decrease the frequency of branches
  - can be based on program traces
    - for frequently executed programs we can determine which branches are more likely to be taken, and which more likely to be not taken
- **Dynamic branch prediction**
  - Prediction is based on the run-time behaviour of the program
    - Behaviour of the branch
    - Behaviour of the preceding branches
  - Requires additional hardware to store this information
    - Branch Prediction Buffer
    - Branch History Table

`[0x00400028] beq $t1,$t2,L`

Branch Prediction Buffer



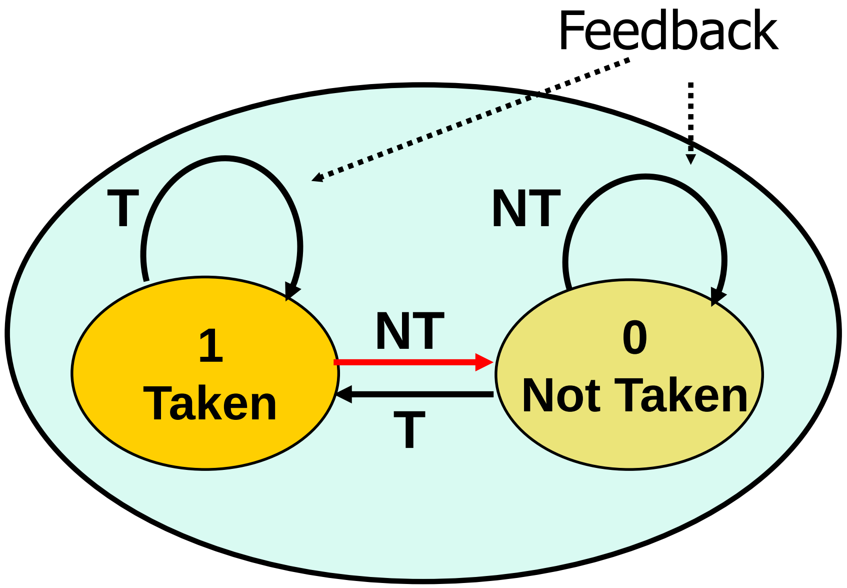
- A table indexed by the lower portion of the address of the branch instruction
  - Contains one bit per entry:
    - recently taken or not taken
    - like a cache but every access is a hit
  - The prediction MAY NOT be for the branch currently executed
    - it depends how many entries in the table
    - more entries -> more hardware -> slower
  - start branch execution according to the prediction
    - if prediction correct – continue
    - if prediction incorrect
      - invert the prediction bit and store in the table
      - cancel partially executed instructions
      - start the proper sequence of instructions


General Form (learn from previous experience)
1. Access
PC
[Prog Counter]
state
2. Predict
Output T/NT
3. Feedback (update) T/NT

States in 1-bit prediction scheme

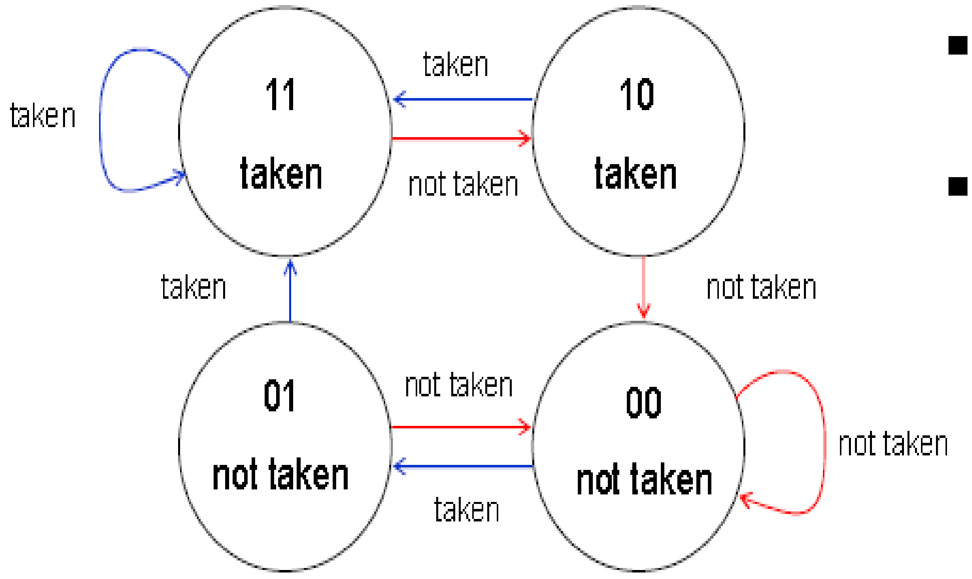
A loop of 10 iterations before exit:

```
        // … …
    for (i=0; i<10; i++)
          a[i] = a[i] * 2.0;
```



- Performance for loops
  - one misprediction on the last execution of the loop
  - one misprediction on the first execution of the loop (previous execution of this loop ended with branch not taken)
  - for example if the loop executed 10 times: 80% accuracy
- For less regular branches
  - much lower accuracy
  - may even be zero if the branch is alternately: not taken - taken

States in 2-bit prediction scheme



- Finite state automaton
  - the 2-bit value is a saturating "counter"
  - increment on "taken"
  - decrement on "not taken"

- Use 2 bits per entry instead of one
- Prediction must be wrong twice before it is changed
  - if predicted taken

    start execution at target as soon as target is known
  - if predicted not-taken

    continue sequential execution until direction
- Change prediction only if getting misprediction twice

Pipelined Performance Example

The rest is for self-study

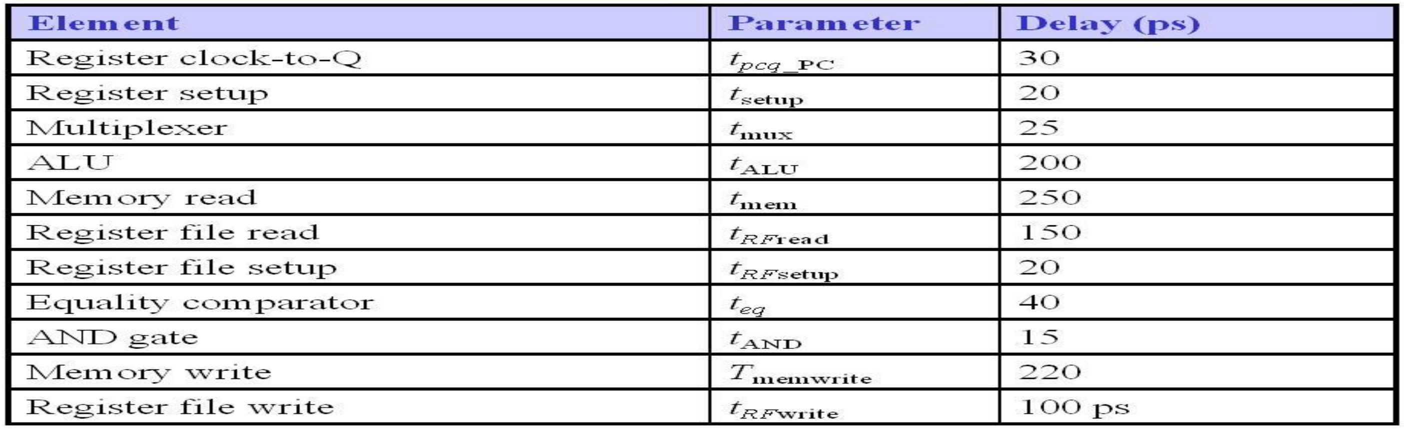
For a program with **100 billion** instructions, Execution Time = ?

- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
  - All jumps flush next instruction
- What is the average CPI?
  - Load/Branch CPI = 1 when no stalling, 2 when stalling. Thus,
    - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
    - $CPIb_{eq} = 1(0.75) + 2(0.25) = 1.25$
    - Average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1) = 1.15

Pipelined Performance Example

- Pipelined processor critical path:
  - $T_c = 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}) =$
    $2[150 + 25 + 40 + 15 + 25 + 20]$ ps $= \mathbf{550}$ **ps**

Pipelined processor critical path			
	1) Fetch	$t_{pcq} + t_{mem} + t_{setup}$	
	2) Decode	$2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$	2*half cycle
T_c = max	3) Execute	$t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}$	
	4) Memory	$t_{pcq} + t_{memwrite} + t_{setup}$	
	5) Writeback	$2(t_{pcq} + t_{mux} + t_{RFwrite})$	2*half cycle



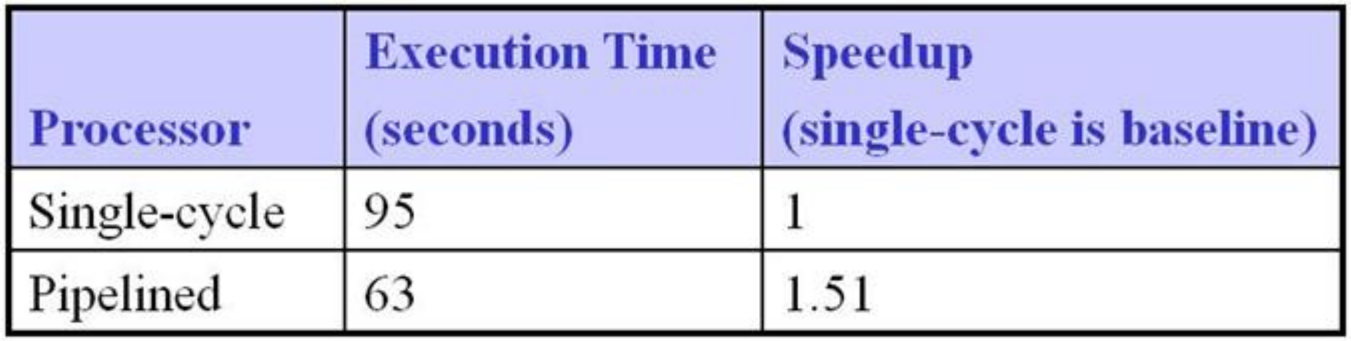
Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	$T_{memwrite}$	220
Register file write	$t_{RFwrite}$	100 ps

Pipelined Performance Example

- For a program with 100 billion instructions executing on a pipelined MIPS processor,
  - CPI = 1.15
  - T_c = 550 ps
- Execution Time = (# instructions) × CPI × Tc

$$= (100 \times 10^9)(1.15)(550 \times 10^{-12})$$
$$= 63 \text{ seconds}$$

- What is speedup?



Processor	Execution Time (seconds)	Speedup (single-cycle is baseline)
Single-cycle	95	1
Pipelined	63	1.51

Summary

- Pipelining is a fundamental concept
  - Multiple steps using distinct resources
  - Exploiting parallelism in instructions
- What makes it easy? (MIPS vs. 80x86)
  - All instructions are the same length
    ⇒ simple instruction fetch
  - Just a few instruction formats
    ⇒ read registers before decode instruction
  - Memory operands only in loads and stores
    ⇒ fewer pipeline stages
  - Data aligned ⇒ 1 memory access / load; 1 memory access / store
- What makes it hard?
  - Structural hazards: suppose we had only one cache?
    ⇒ Need more HW resources
  - Control hazards: need to worry about branch instructions?
    ⇒ Branch prediction, delayed branch
  - Data hazards: an instruction depends on results of a previous instruction?
    ⇒ need forwarding, compiler scheduling

Revision and quiz

- … How much faster is pipeline (slide 6)? Answer: 20min
- So the latency is 1250 ps; Time between instructions is 250 ps
- List the steps for executing MIPS instructions through pipelining datapath. In addition, what functional hardware components are used?
- There are three types of pipeline hazards, what they are?
- … answer (slide 15):

  memory = 100% memory access
  store = 30% memory access
  per cycle = 1.3
  er cycle = 1.0 no duplicated rsc

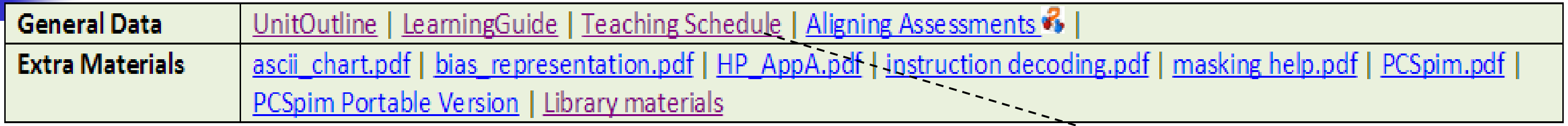
- … answer (slide 17):

  Issue: The subsequent instructions need this value $s0 earlier
  ADD gets value of $s0 after its stage: EX stage (cycle 3)
  AND before its stage EX stage (cycle 4); OR in stage EX stage (cycle 5); SUB in stage RF stage (cycle 5)

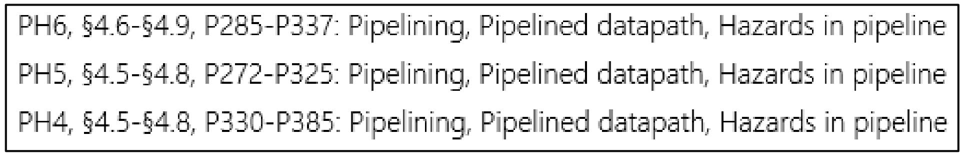
- With the delayed branch strategy, getting instructions from 'fall through' is one solution. This solution is valuable only when branch-not-taken.

  1) True 2) False

Recommended readings



General Data	UnitOutline | LearningGuide | Teaching Schedule | Aligning Assessments |
Extra Materials	ascii_chart.pdf | bias_representation.pdf | HP_AppA.pdf | instruction decoding.pdf | masking help.pdf | PCSpim.pdf | PCSpim Portable Version | Library materials



PH6, §4.6-§4.9, P285-P337: Pipelining, Pipelined datapath, Hazards in pipeline

PH5, §4.5-§4.8, P272-P325: Pipelining, Pipelined datapath, Hazards in pipeline

PH4, §4.5-§4.8, P330-P385: Pipelining, Pipelined datapath, Hazards in pipeline

Text readings are listed in Teaching Schedule and Learning Guide

PH6 (PH5 & PH4 also suitable): check whether eBook available on library site

PH6: companion materials (e.g. online sections for further readings)

https://www.elsevier.com/books-and-journals/book-companion/9780128201091

PH5: companion materials (e.g. online sections for further readings) http://booksite.elsevier.com/9780124077263/?ISBN=9780124077263