RISC-V Architecture & Processor Design

Week 6

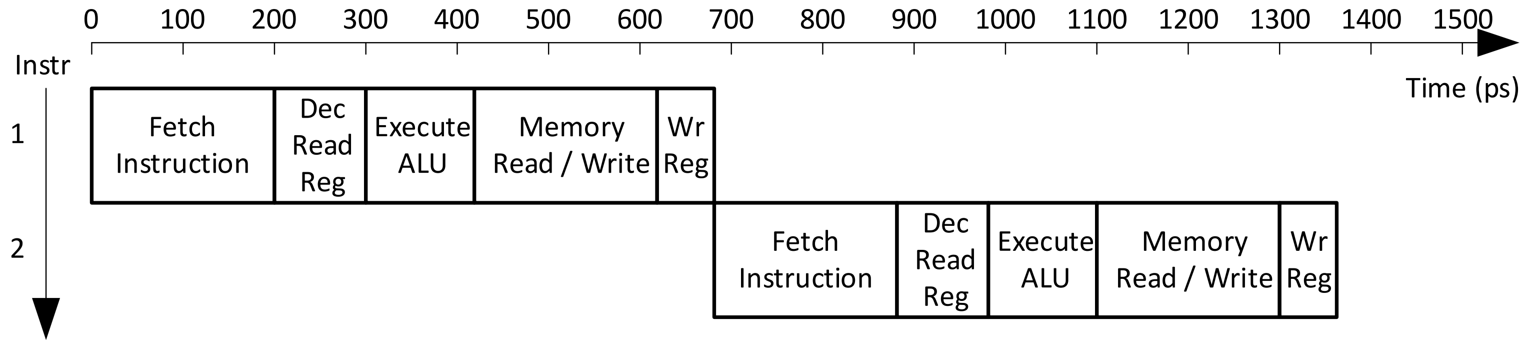
RISC-V uArchitecture: Pipelined Processor

Pipelined RISC-V Processor

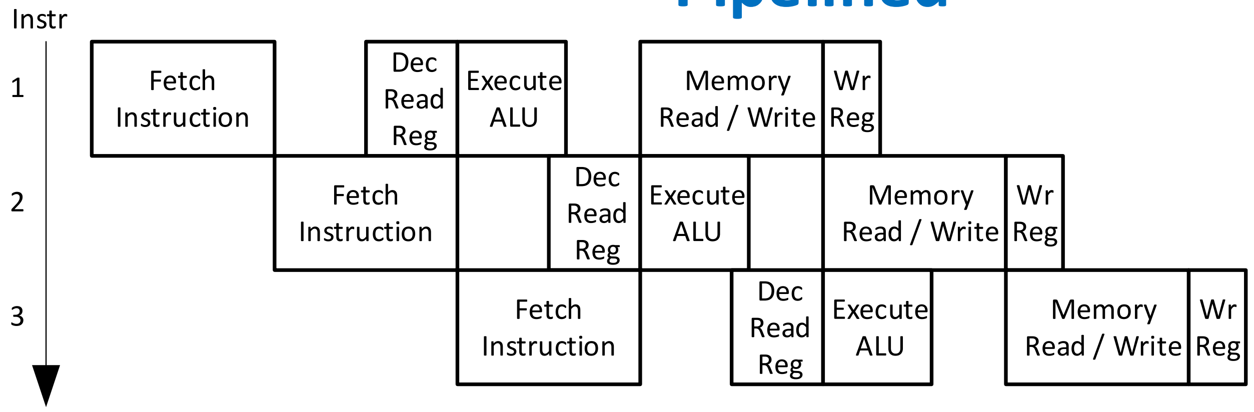
- **Temporal parallelism**
- Divide single-cycle processor into **5 stages**:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add **pipeline registers** between stages

Single-Cycle vs. Pipelined Processor

Single-Cycle

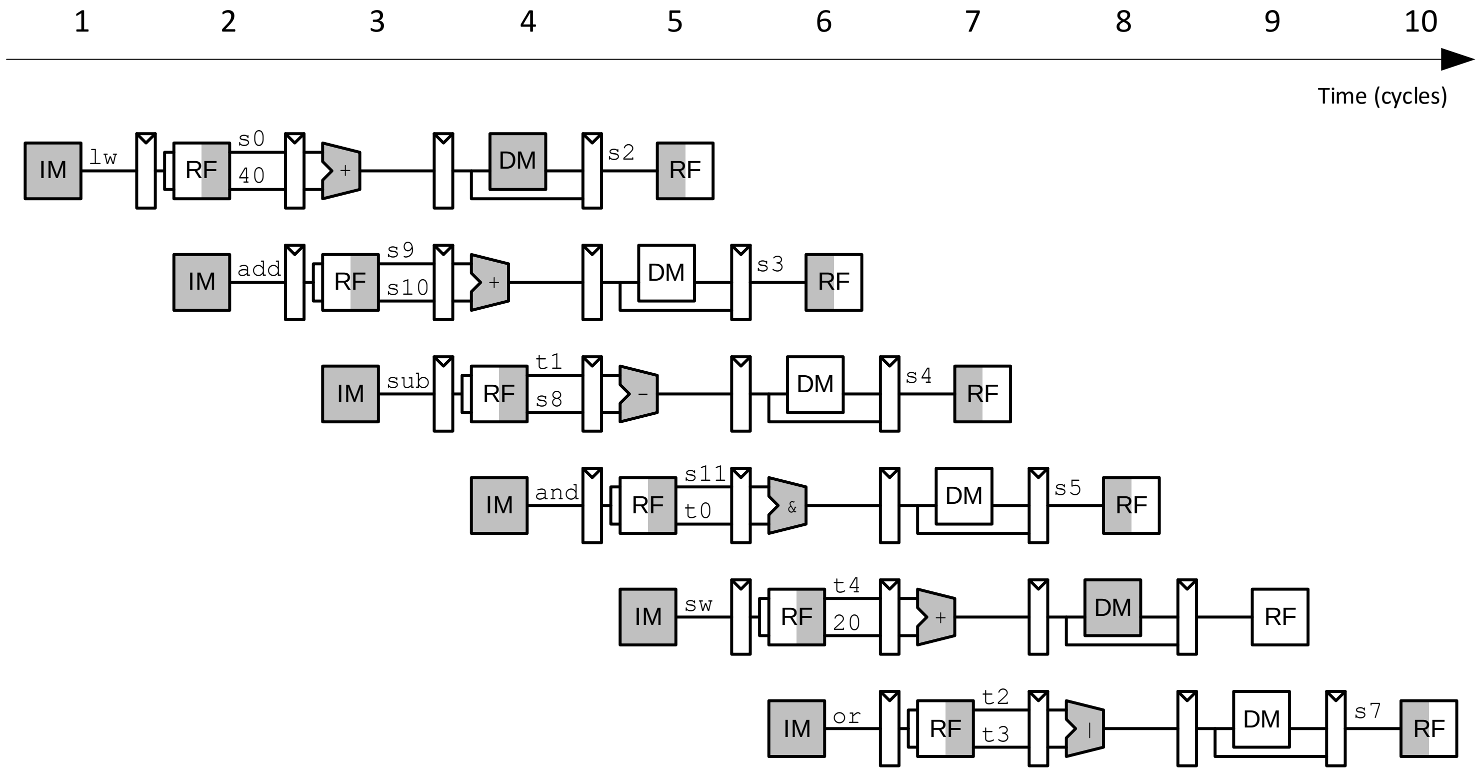


Pipelined

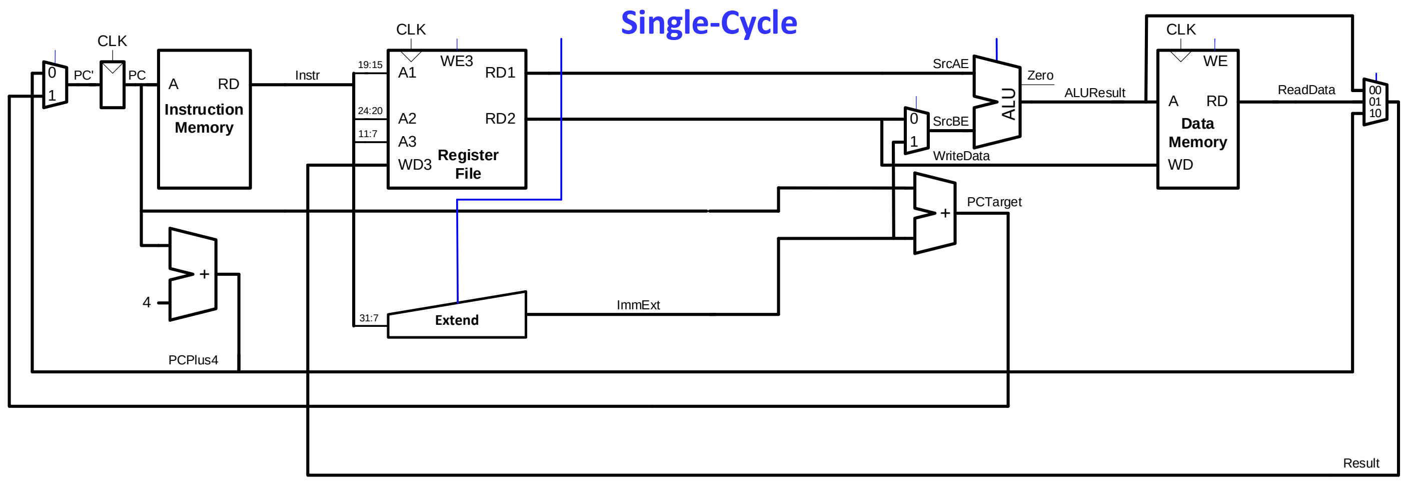


Pipelined Processor Abstraction

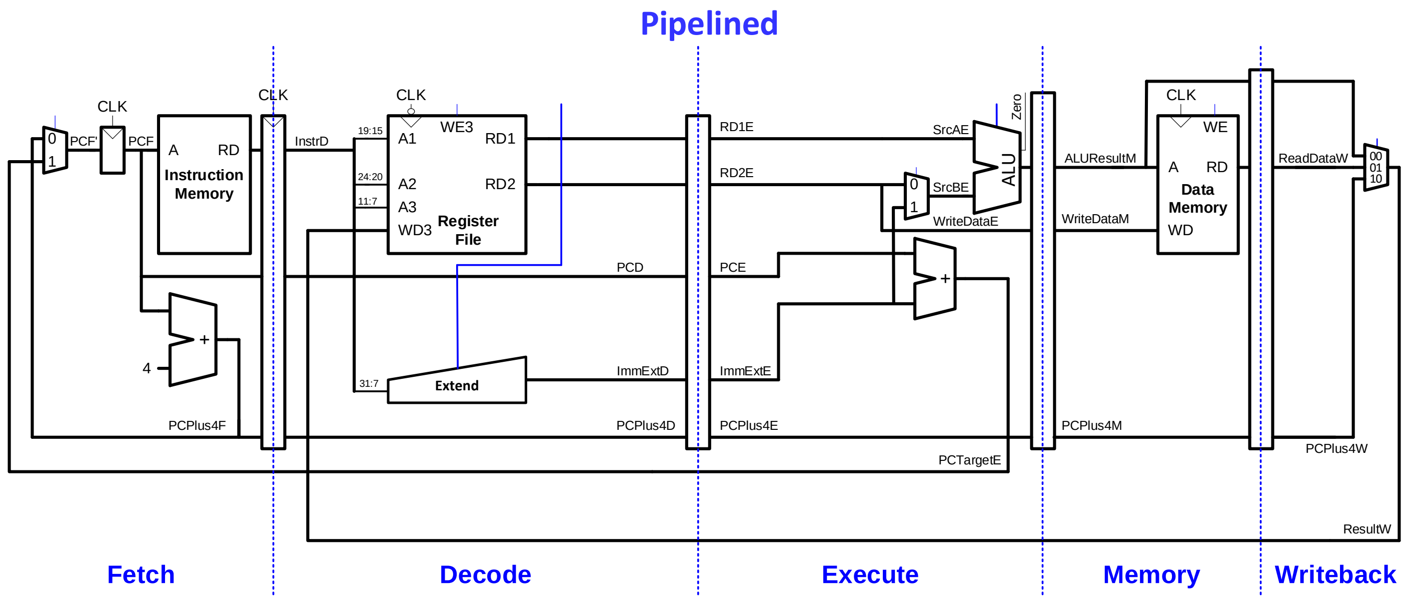
```
lw  s2, 40(s0)

add s3, s9, s10

sub s4, t1, s8

and s5, s11, t0

sw  s6, 20(t4)

or  s7, t2, t3
```



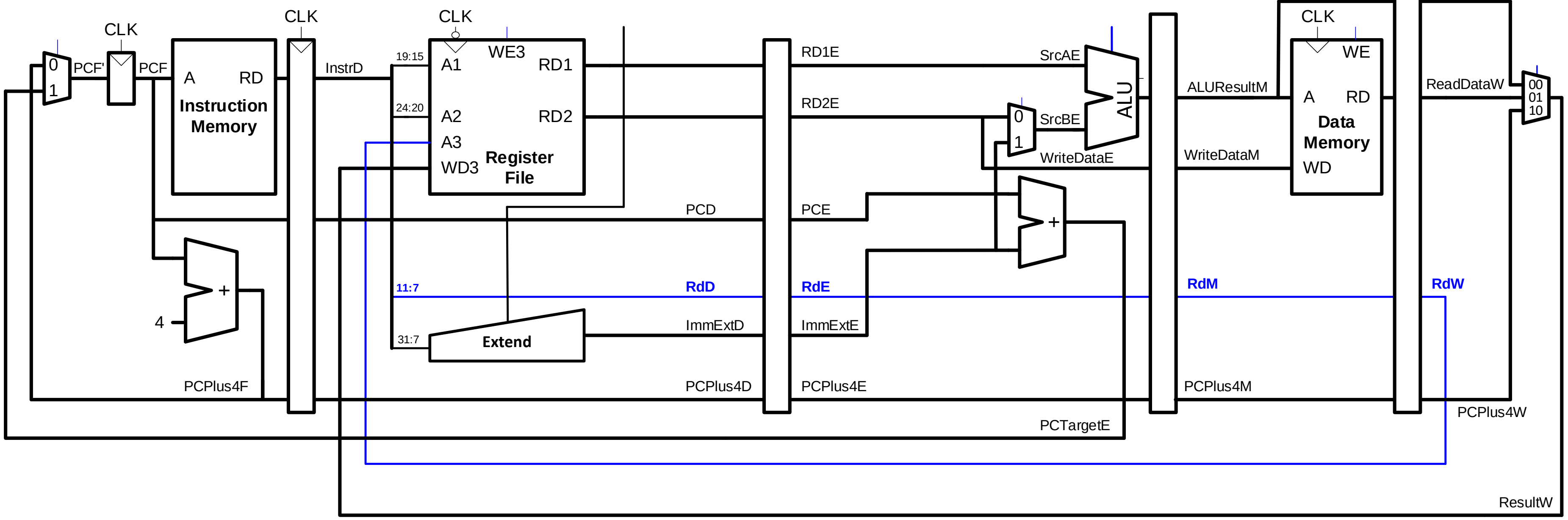
Single-Cycle & Pipelined Datapaths



Signals in Pipelined Processor are appended with first letter of stage (i.e., PCF, PCD, PCE).

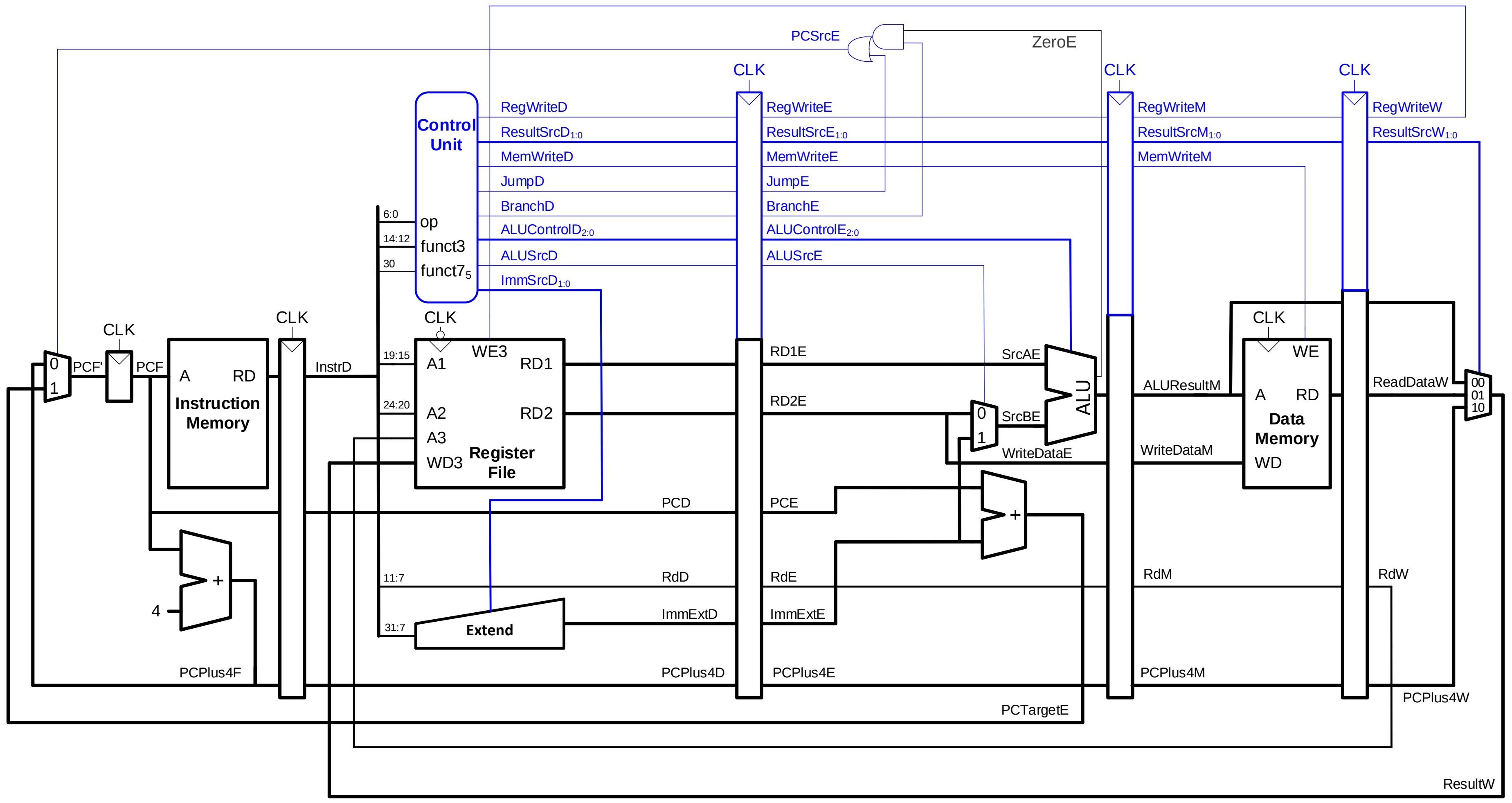


Corrected Pipelined Datapath



- ***Rd*** must arrive at same time as ***Result***
- Register file written on **falling edge** of *CLK*

Pipelined Processor with Control



- **Same control unit** as single-cycle processor
- **Control signals travel with** the instruction (drop off when used)

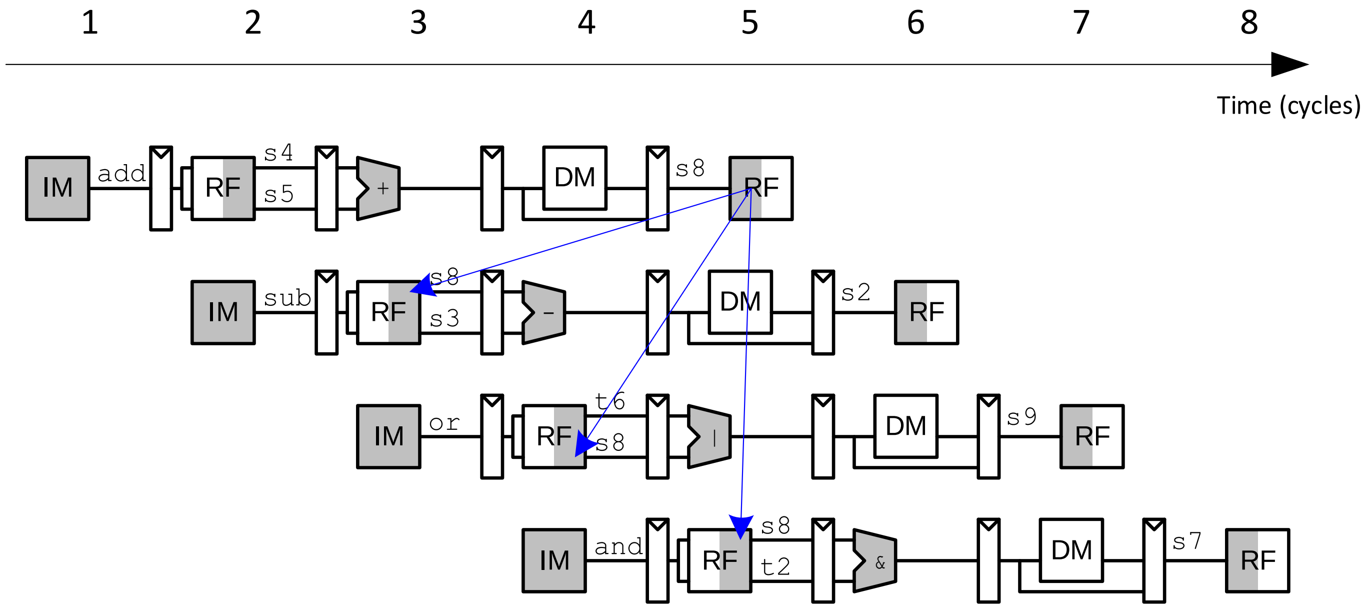
Pipelined Processor Hazards

Pipelined Hazards

- When an instruction depends on result from instruction that hasn’t completed
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branch)

Data Hazard

```
add s8, s4, s5

sub s2, s8, s3

or  s9, t6, s8

and s7, s8, t2
```

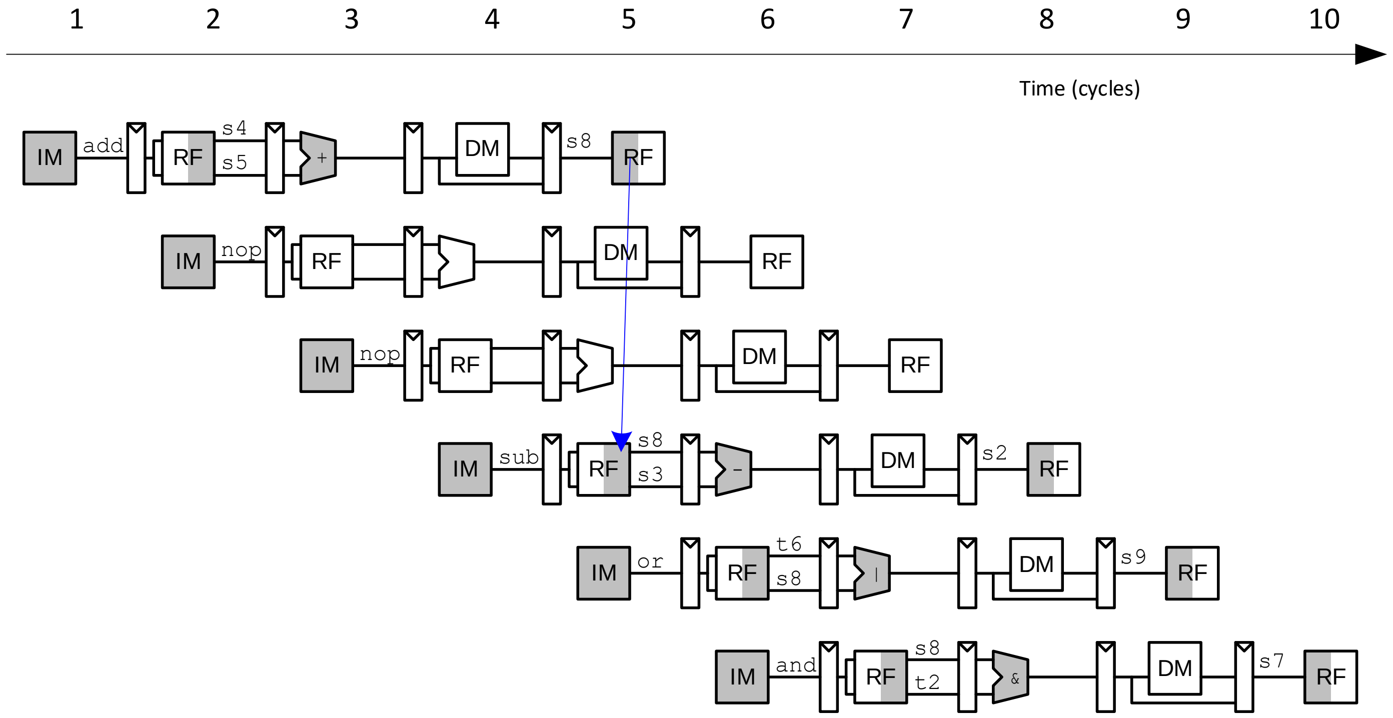


Handling Data Hazards

Handling Data Hazards

- **Insert** enough **nops** for result to be ready
- Or move independent useful instructions forward

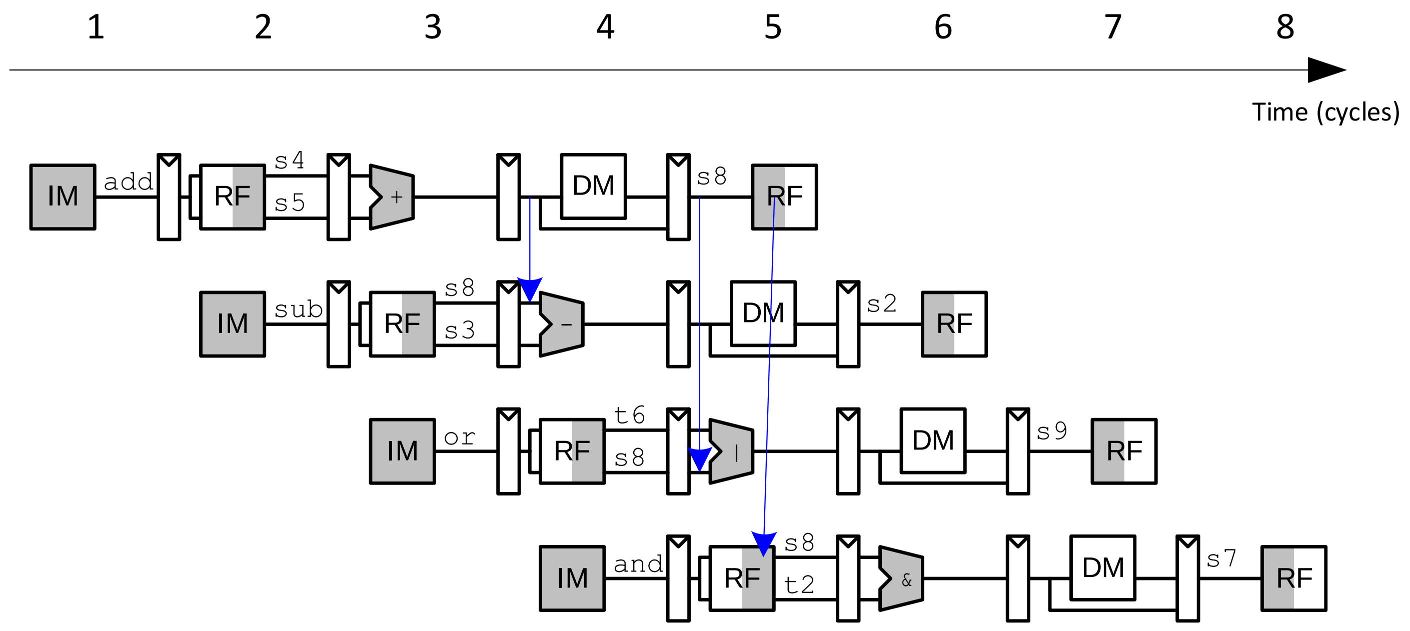
```
add s8, s4, s5
nop
nop
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



Data Forwarding

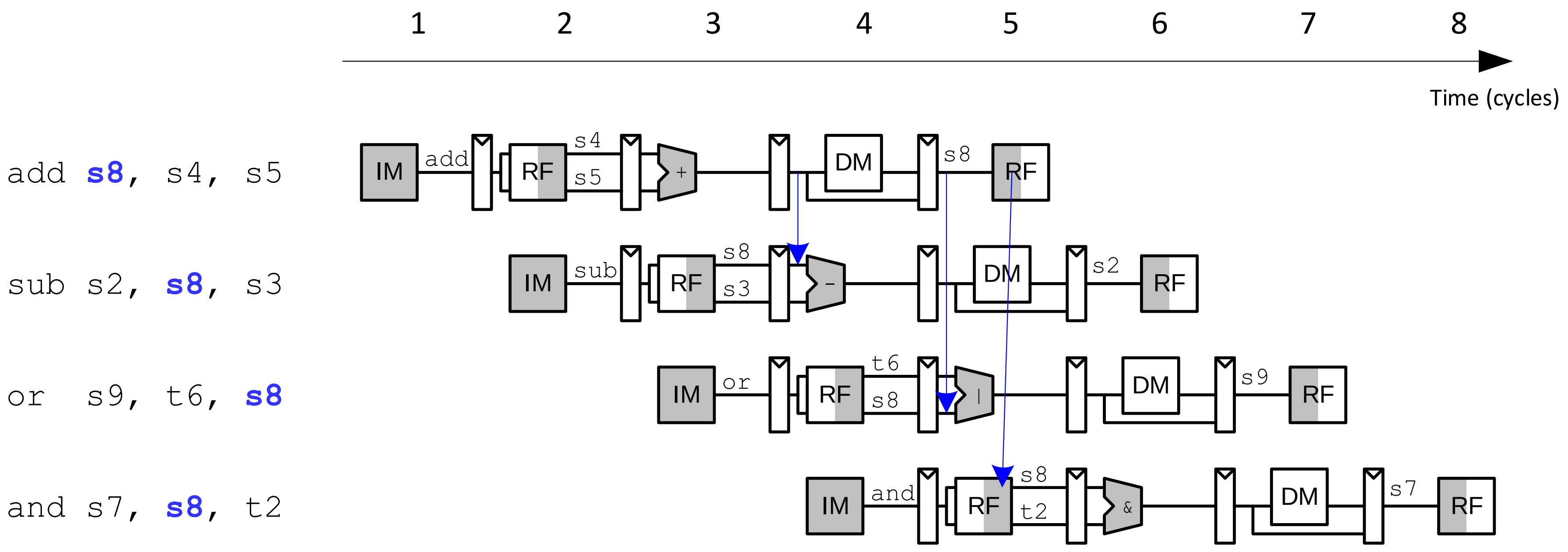
- Data is **available on internal busses** before it is written back to the register file (RF).
- **Forward data** from internal busses **to Execute stage**.

```
add s8, s4, s5

sub s2, s8, s3

or  s9, t6, s8

and s7, s8, t2
```

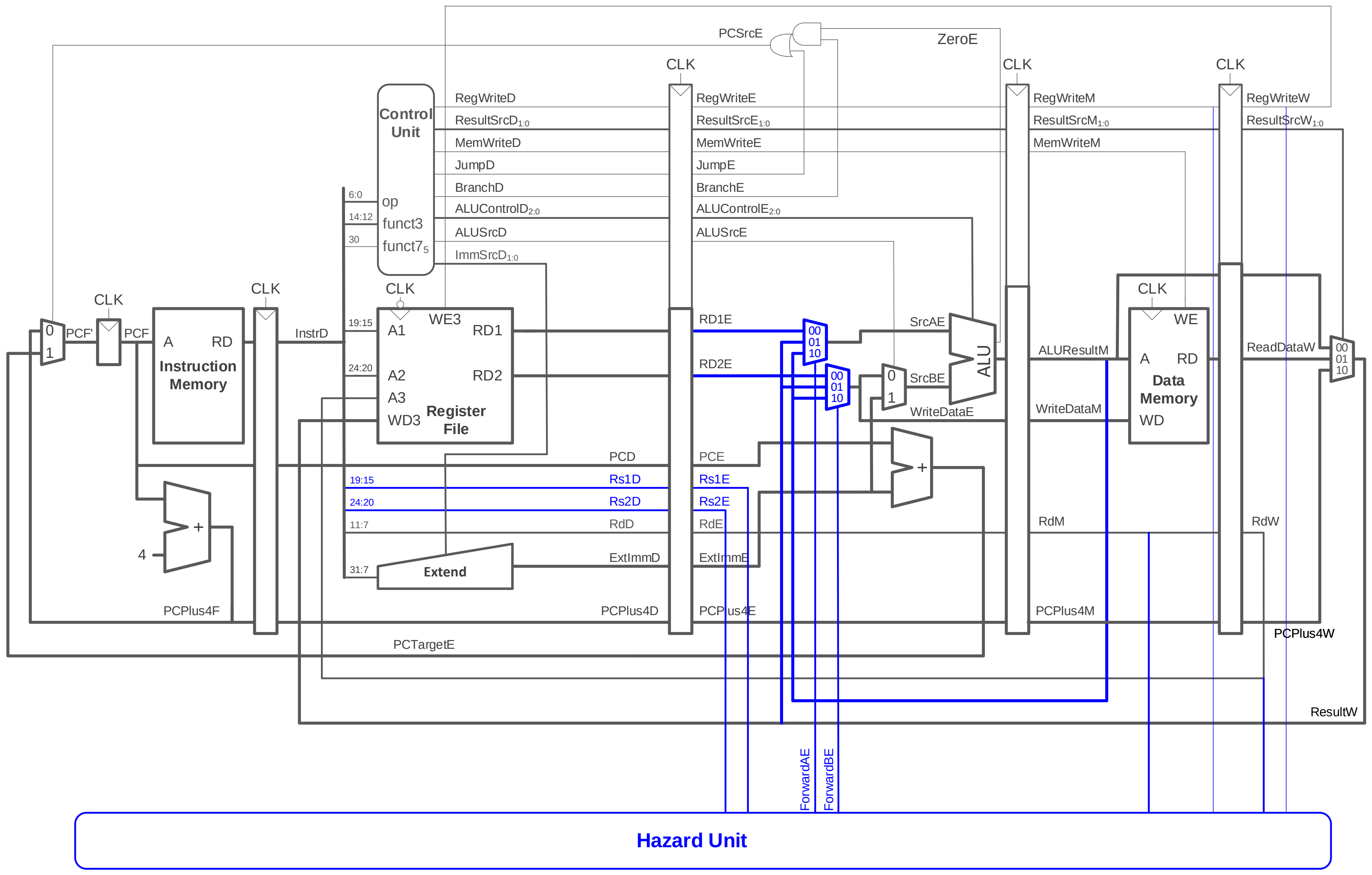


Data Forwarding

- Check if source register **in Execute stage** **matches** destination register of instruction **in Memory or Writeback stage**.
- If so, forward result.



Data Forwarding: Hazard Unit



Data Forwarding

- **Case 1:** **Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2:** **Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for *Rs1*:

```
if       ((Rs1E == RdM) AND RegWriteM)       // Case 1
                  ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW)        // Case 2
                  ForwardAE = 01
else              ForwardAE = 00              // Case 3
```

***ForwardBE* equations are similar (replace Rs1E with Rs2E)**

Data Forwarding

- **Case 1: Execute** stage *Rs1* or *Rs2* matches **Memory** stage *Rd*?
  Forward from Memory stage
- **Case 2: Execute** stage *Rs1* or *Rs2* matches **Writeback** stage *Rd*?
  Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

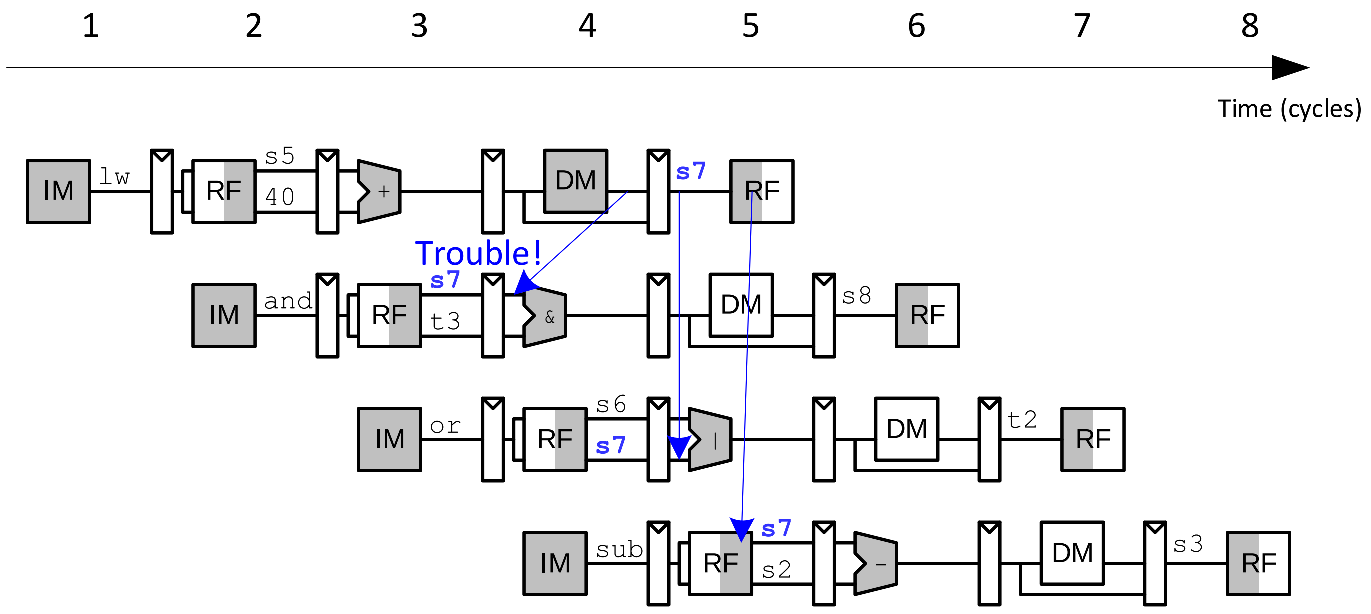
Equations for *Rs1*:

```
if       ((Rs1E == RdM) AND RegWriteM) AND (Rs1E != 0) // Case 1
                   ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW) AND (Rs1E != 0) // Case 2
                   ForwardAE = 01
else               ForwardAE = 00                      // Case 3
```

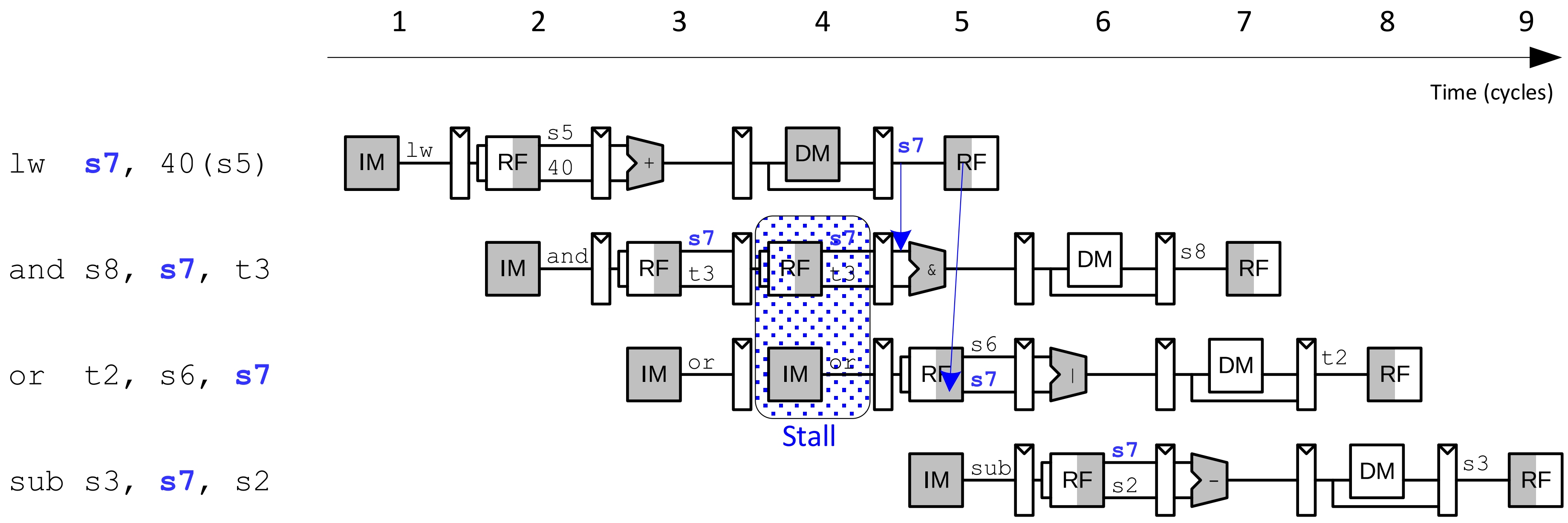
***ForwardBE* equations are similar (replace Rs1E with Rs2E)**

Data Hazard due to `lw` Dependency

```
lw   s7, 40(s5)

and s8, s7, t3

or  t2, s6, s7

sub s3, s7, s2
```



Stalling to solve `lw` Data Dependency



Stalling Logic

- Is either **source register in the Decode stage** the same as the **destination register in the Execute stage**?

  AND

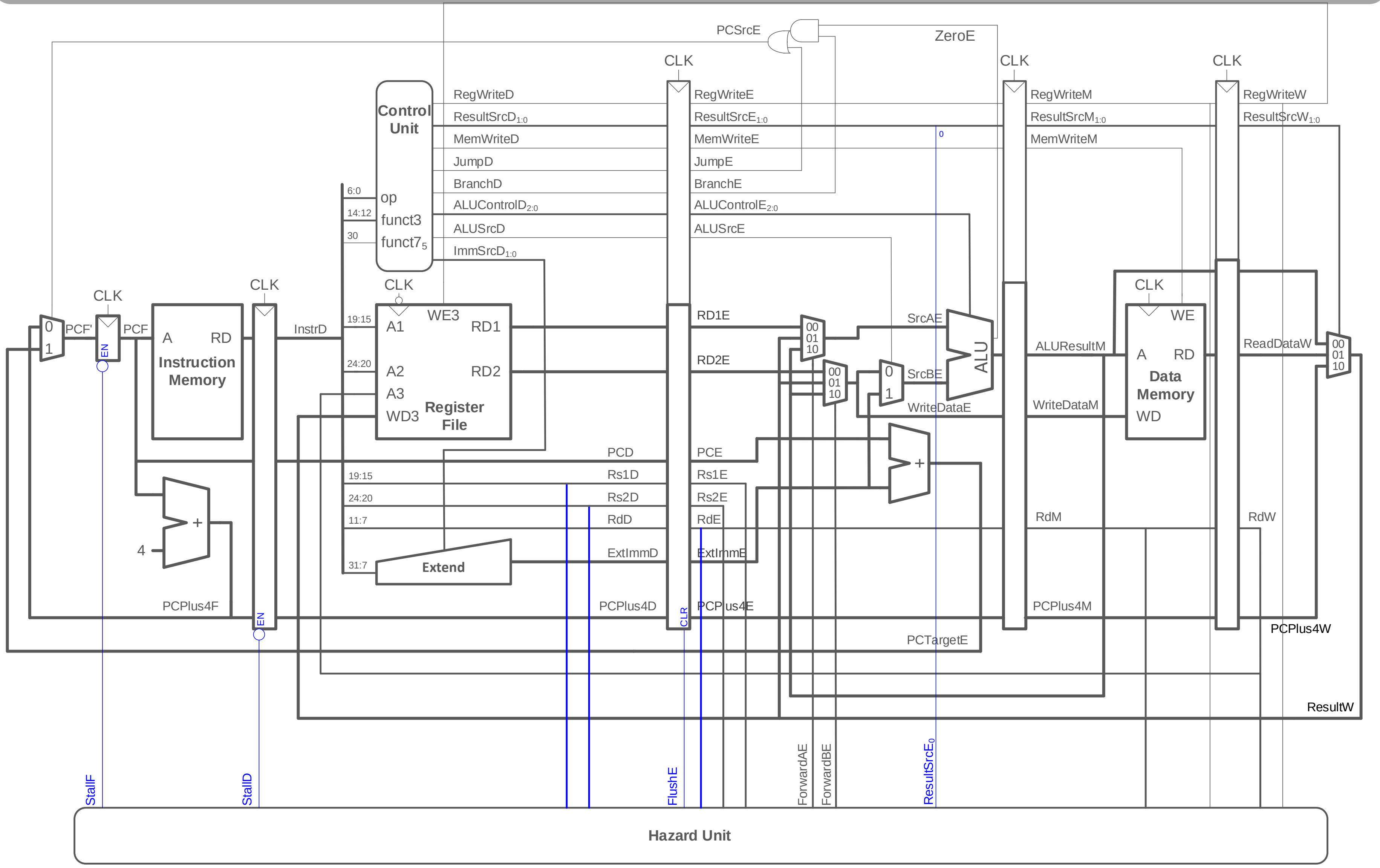
- Is the instruction in the **Execute stage a `lw`**?

lwStall = ((*Rs1D* == *RdE*) OR (*Rs2D* == *RdE*)) **AND** $\boldsymbol{ResultSrcE_1}$

StallF = ***StallD*** = ***FlushE*** = ***lwStall***

(Stall the Fetch and Decode stages, and flush the Execute stage.)

Stalling Hardware



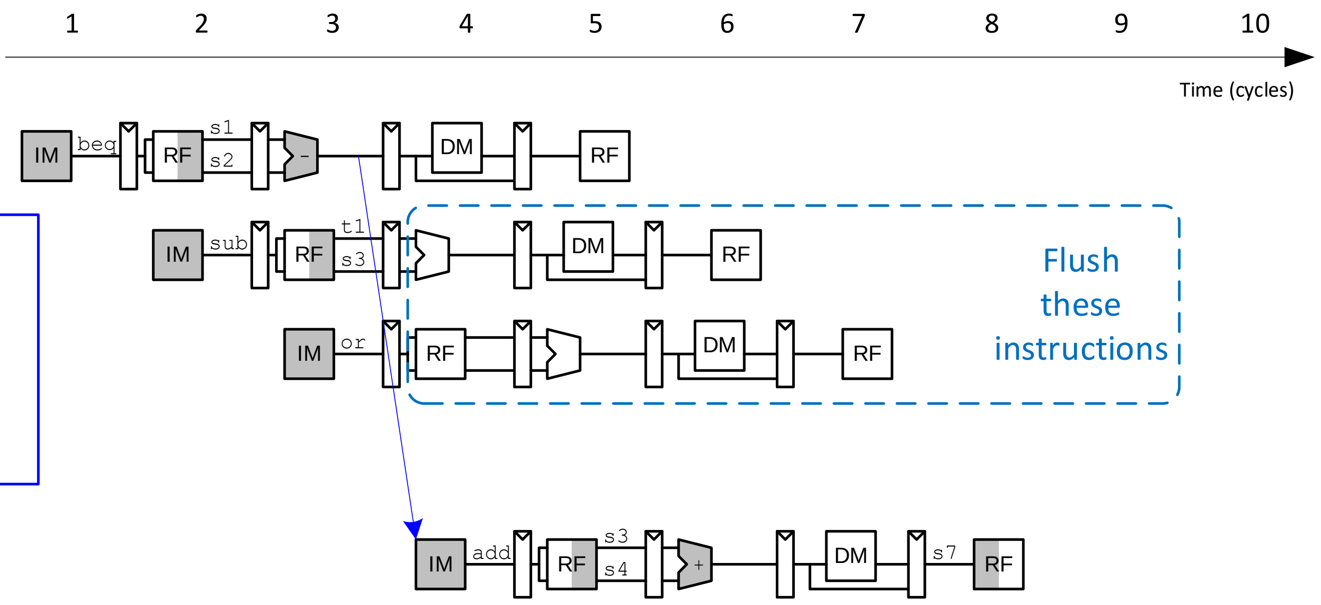
Pipelined Processor Control Hazards

Control Hazards

- **`beq`:**
  - Branch **not determined until the Execute stage** of pipeline
  - **Instructions** after branch **fetched** before branch occurs
  - These **2 instructions must be flushed** if branch happens

Control Hazards

```
20        beq s1, s2, L1

24        sub s8, t1, s3

28        or  s9, t6, s5

2C        ...

...       ...

58    L1: add s7, s3, s4
```



Branch misprediction penalty:

The number of instructions flushed when a branch is taken (in this case, 2 instructions)

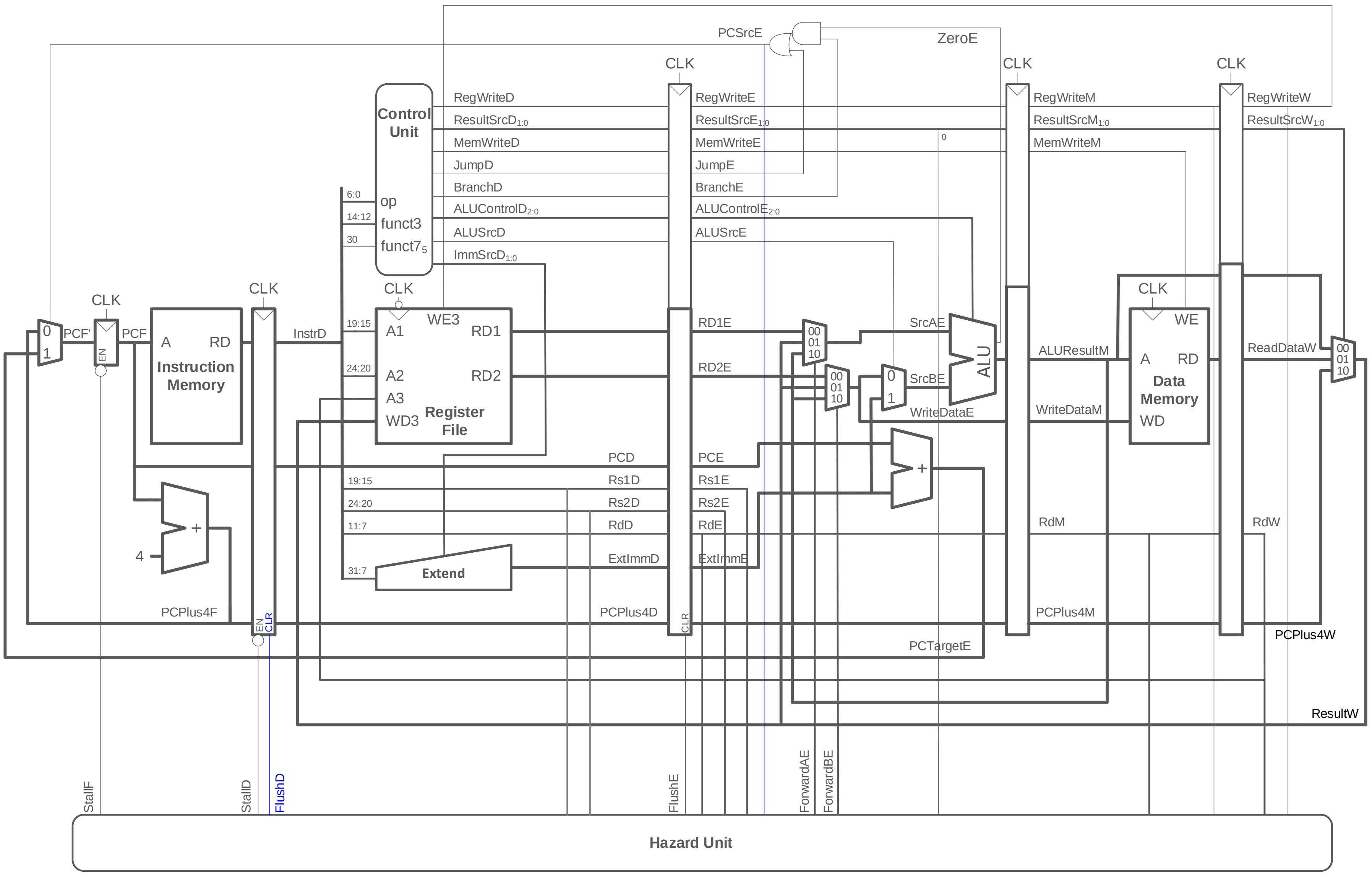
Control Hazards: Flushing Logic

- If branch is taken in execute stage, need to flush the instructions in the Fetch and Decode stages
  - Do this by clearing Decode and Execute Pipeline registers using *FlushD* and *FlushE*
- **Equations:**

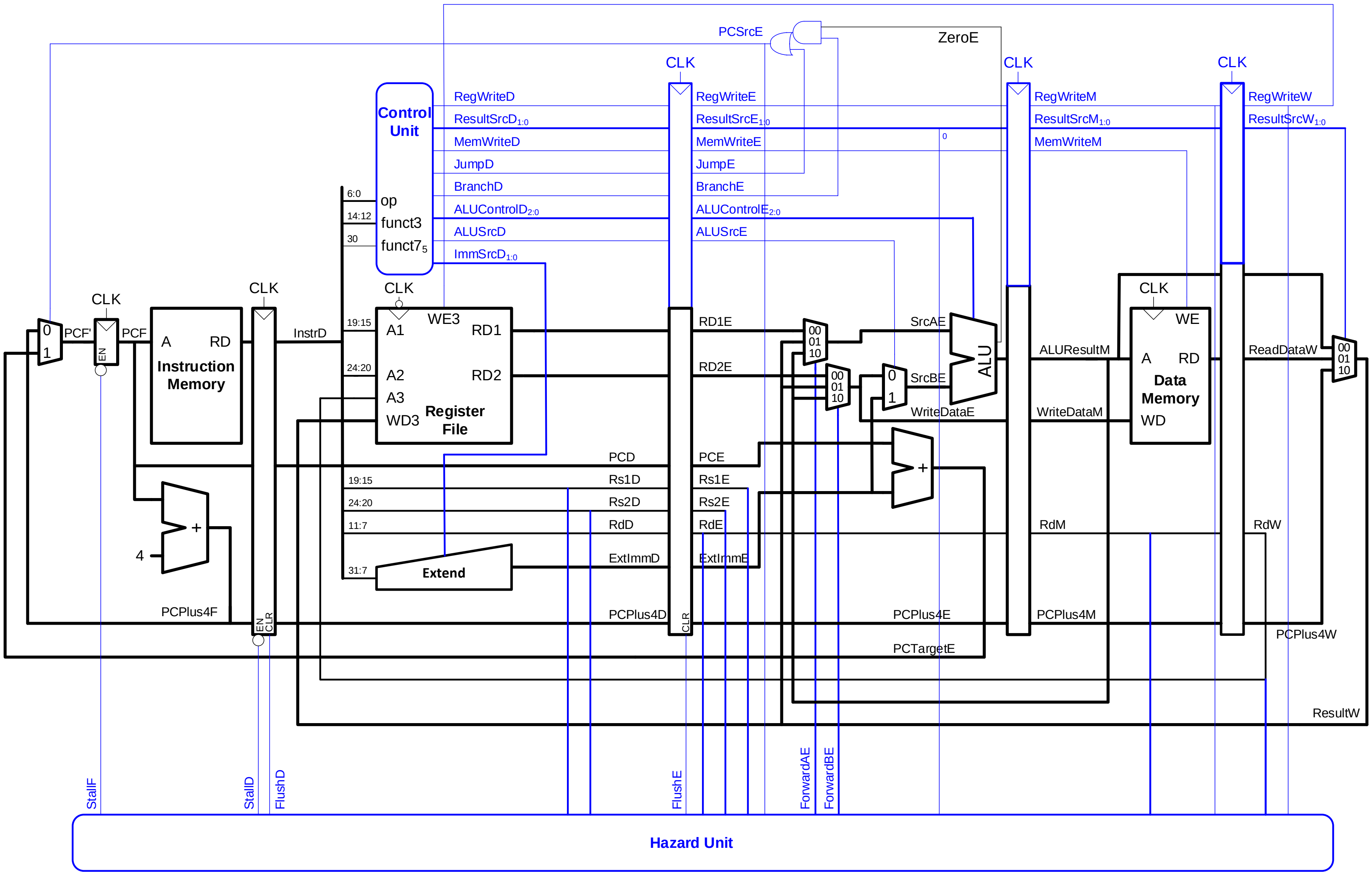
$$FlushD = PCSrcE$$

$$FlushE = lwStall \text{ OR } PCSrcE$$

Control Hazards: Flushing Hardware



RISC-V Pipelined Processor with Hazard Unit



Pipelined Performance

Pipelined Processor Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?** (Ideally it’s 1, but…)

Pipelined Processor Performance Example

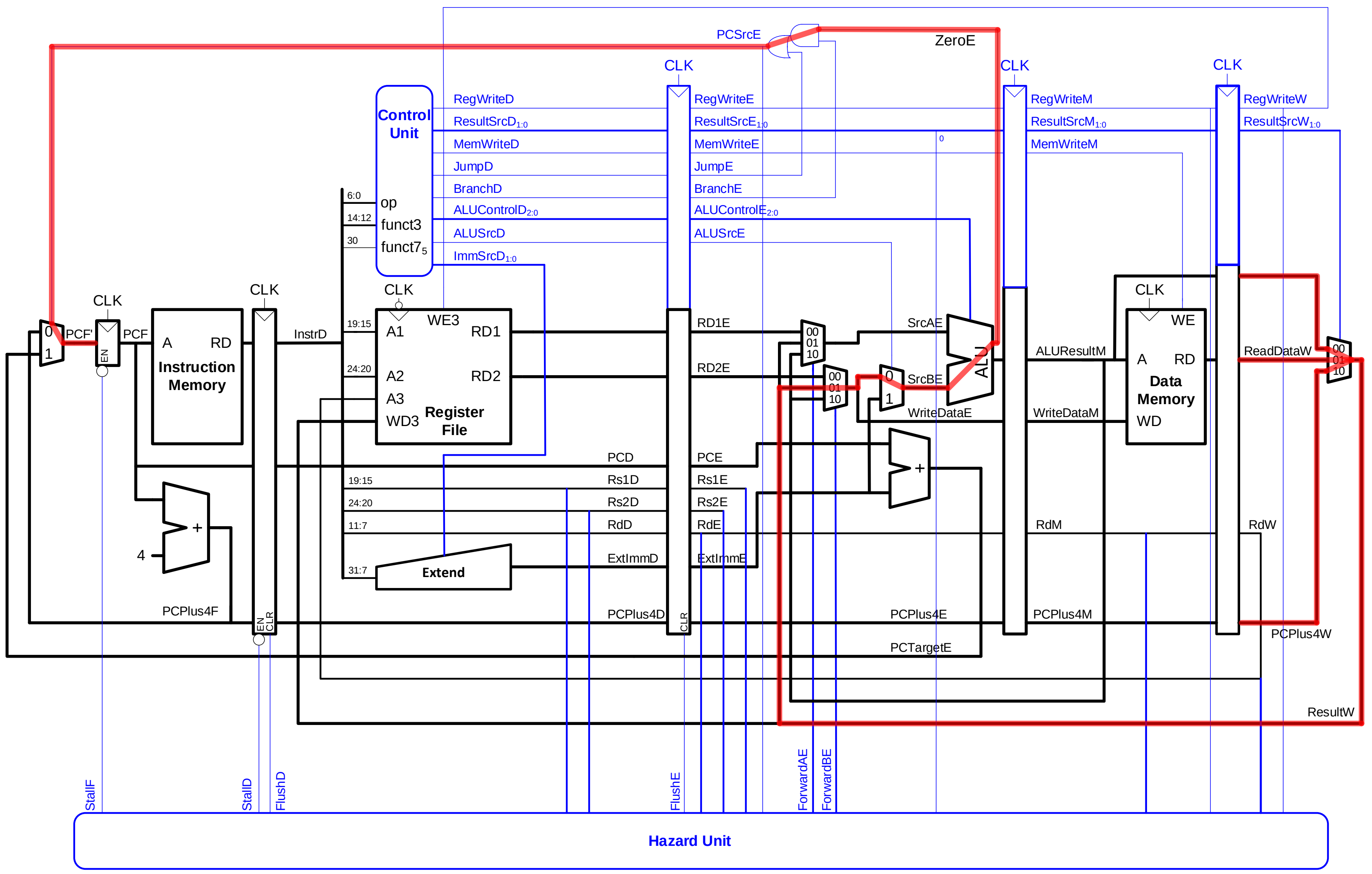
Pipelined processor critical path:

$T_{c\_pipelined}$ = max of

$t_{pcq} + t_{\text{mem}} + t_{\text{setup}}$	**Fetch**
$2(t_{\text{RFread}} + t_{\text{setup}})$	**Decode**
$t_{pcq} + 4t_{\text{mux}} + t_{\text{ALU}} + t_{\text{AND-OR}} + t_{\text{setup}}$	**Execute**
$t_{pcq} + t_{\text{mem}} + t_{\text{setup}}$	**Memory**
$2(t_{pcq} + t_{\text{mux}} + t_{\text{RFwrite}})$	**Writeback**

- Decode and Writeback stages **both use the register file** in each cycle
- So each stage gets half of the cycle time (**T_c/2**) to do their work
- Or, stated a different way, **2x of their work** must fit in a cycle (T_c)

Pipelined Critical Path: Execute Stage



Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	t_{AND-OR}	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{dec}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$$T_{c\_pipelined} = t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$$
$$=$$

Pipelined Performance Example

Program with 100 billion instructions

Execution Time $= (\# \text{ instructions}) \times \text{CPI} \times T_c$

$= (100 \times 10^9)(1.23)(350 \times 10^{-12})$

= 43 seconds

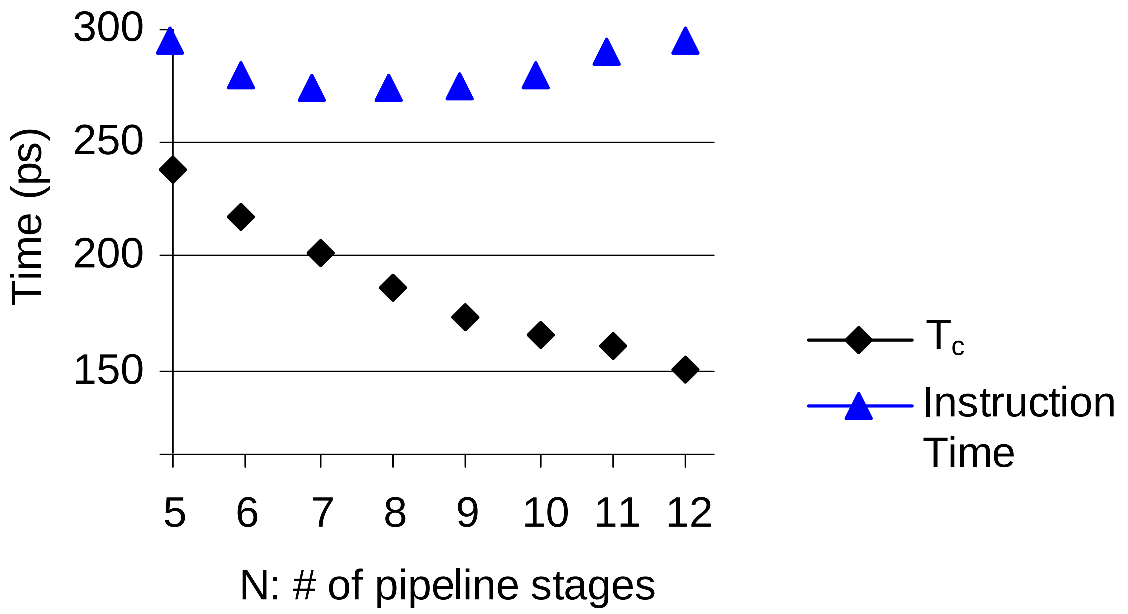
Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	75	1
Multicycle	155	0.5
Pipelined	43	1.7

Branch Prediction

Deep Pipelining

- **10-20 stages typical**
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost



Micro-operations

- Decompose complex instructions into series of simple instructions called ***micro-operations*** (*micro-ops* or *μ-ops*)
- **At run-time**, complex instructions are decoded into one or more micro-ops
- Used heavily in **CISC** (complex instruction set computer) architectures (e.g., x86)

Complex Op	**Micro-op Sequence**
`lw s1, 0(s2), postincr 4`	`lw   s1, 0(s2)`
	`addi s2, s2, 4`

Without μ-ops, would need 2nd write port on the register file

Branch Prediction

- **Guess** whether branch will be taken
  - Backward branches are usually taken (loops)
  - Consider history to improve guess
- Good prediction **reduces fraction of branches requiring a flush**

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- **Static branch prediction:**
  - Check direction of branch (forward or backward)
  - If backward, predict taken
  - Else, predict not taken
- **Dynamic branch prediction:**
  - Keep **history** of last several hundred (or thousand) branches in *branch target buffer*, record:
    - Branch destination
    - Whether branch was taken

Dynamic Branch Prediction

- 1-bit branch predictor
- 2-bit branch predictor

Branch Prediction Example

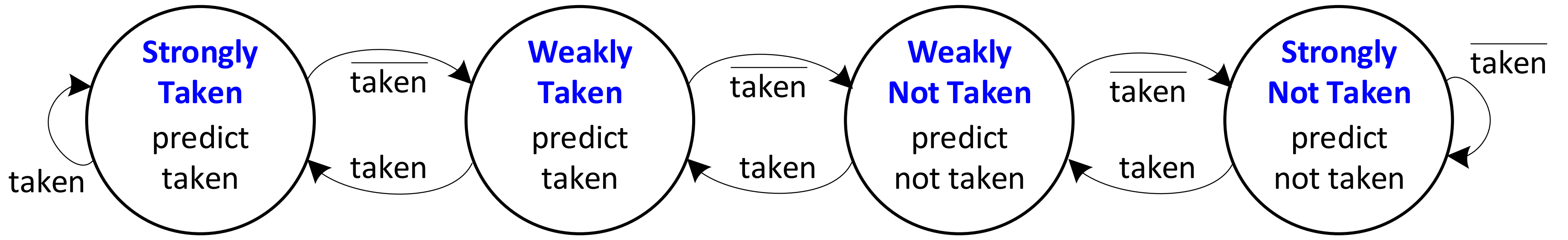
```
  addi s1, zero, 0      # s1 = sum
  addi s0, zero, 0      # s0 = i
  addi t0, zero, 10     # t0 = 10

For:                    # for (i=0; i<10; i=i+1)
  bge  s0, t0, Done
  add  s1, s1, s0       # sum = sum + i
  addi s0, s0, 1        # i = i + 1
  j    For

Done:
```

1-Bit Branch Predictor

- **Remembers** whether branch was taken the last time and **does the same thing**
- Mispredicts first and last branch of loop

```
  addi s1, zero, 0      # s1 = sum
  addi s0, zero, 0      # s0 = i
  addi t0, zero, 10     # t0 = 10

For:                    # for (i=0; i<10; i=i+1)
  bge  s0, t0, Done
  add  s1, s1, s0       # sum = sum + i
  addi s0, s0, 1        # i = i + 1
  j    For

Done:
```

2-Bit Branch Predictor



```
  addi s1, zero, 0      # s1 = sum
  addi s0, zero, 0      # s0 = i
  addi t0, zero, 10     # t0 = 10

For:                    # for (i=0; i<10; i=i+1)
  bge  s0, t0, Done
  add  s1, s1, s0       # sum = sum + i
  addi s0, s0, 1        # i = i + 1
  j    For

Done:
```

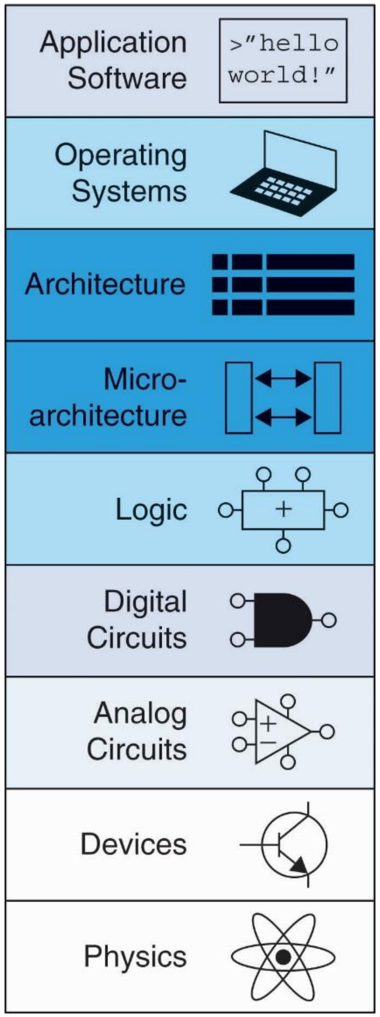
Only mispredicts **last branch** of loop

RISC-V Architecture & Processor Design

Week 6

Memory Systems

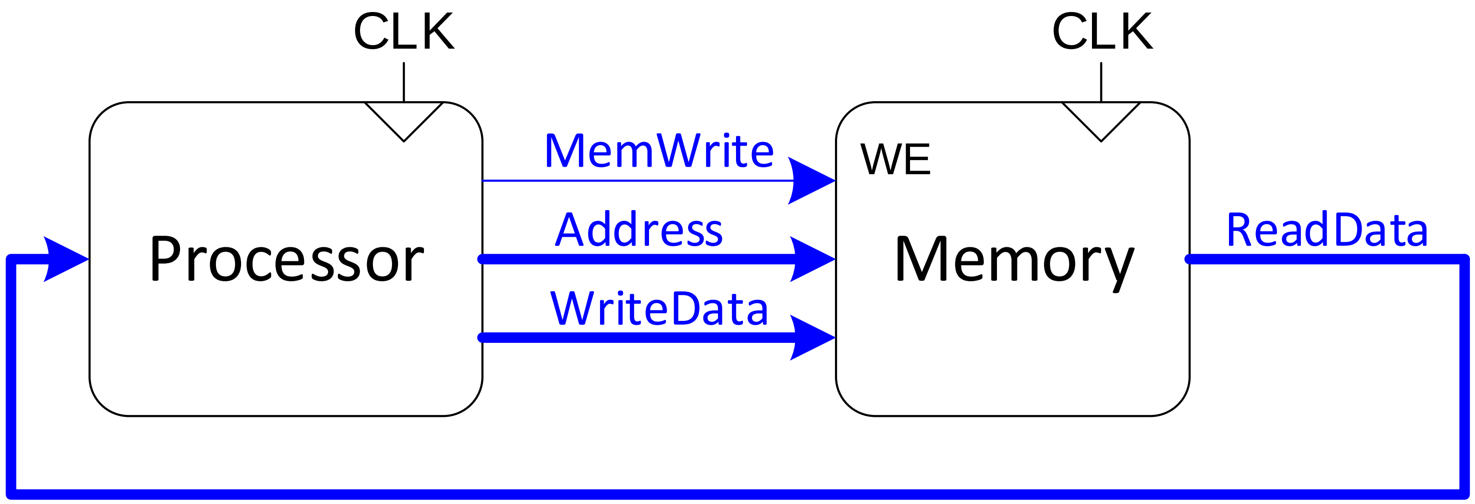
- **Introduction**
- **Memory System Performance Analysis**
- **Caches**



Introduction

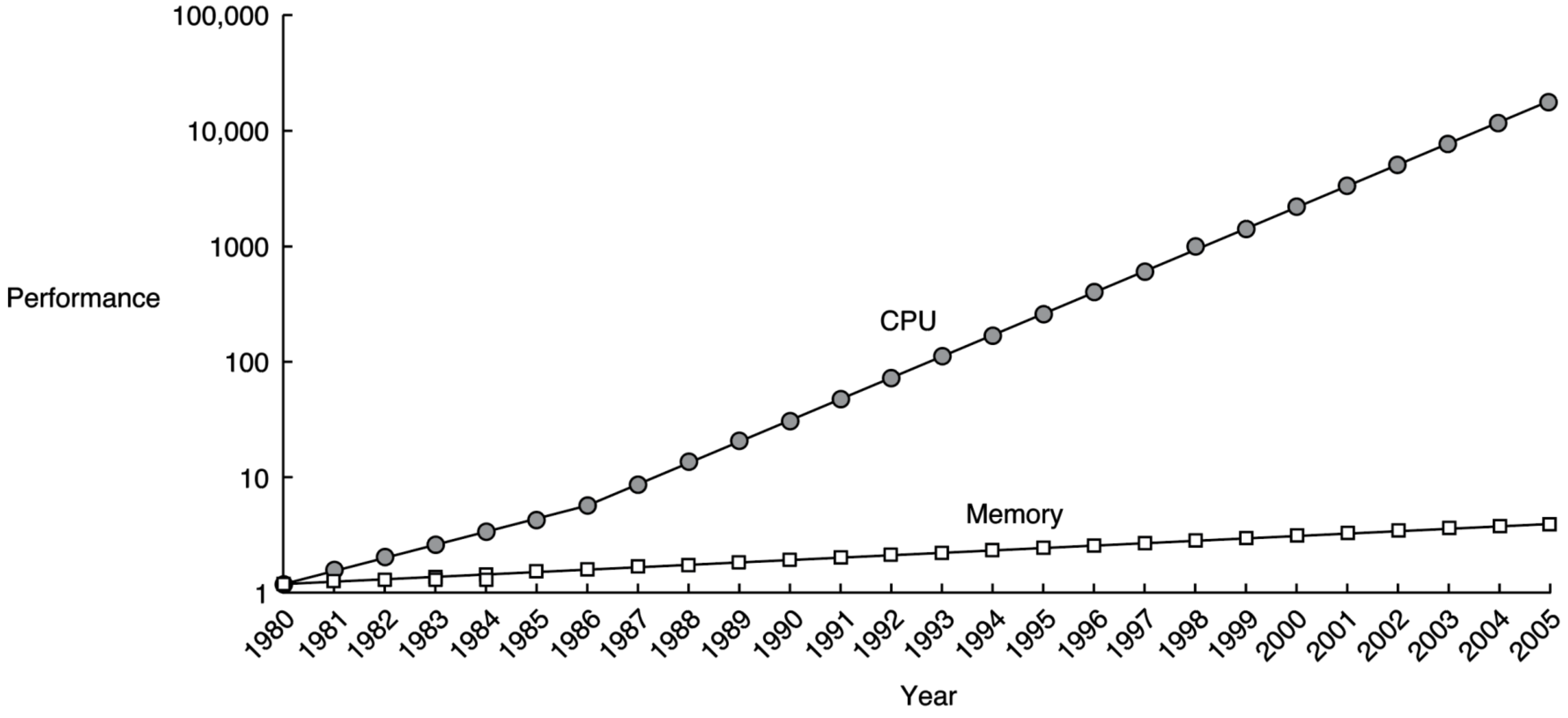
- **Computer performance depends on:**
  - **Processor** performance
  - **Memory system** performance

Processor / Memory Interface:



Processor-Memory Gap

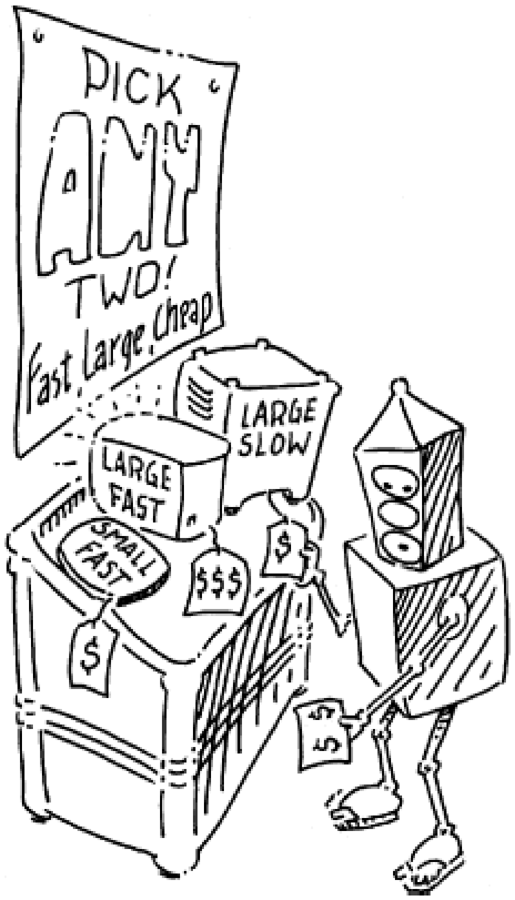
- In prior chapters, assumed access memory in 1 clock cycle – but hasn't been true since the 1980's.



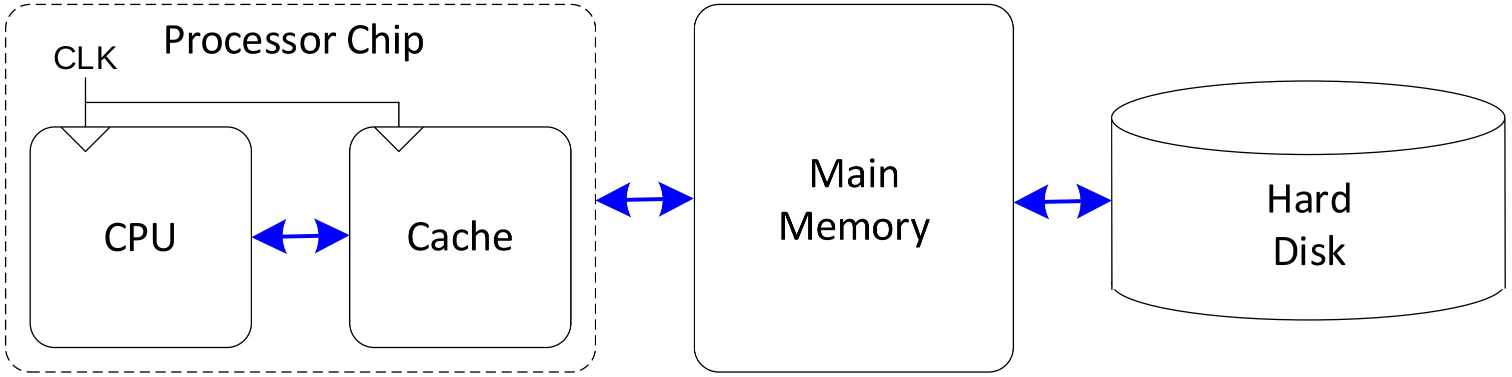
Memory System Challenge

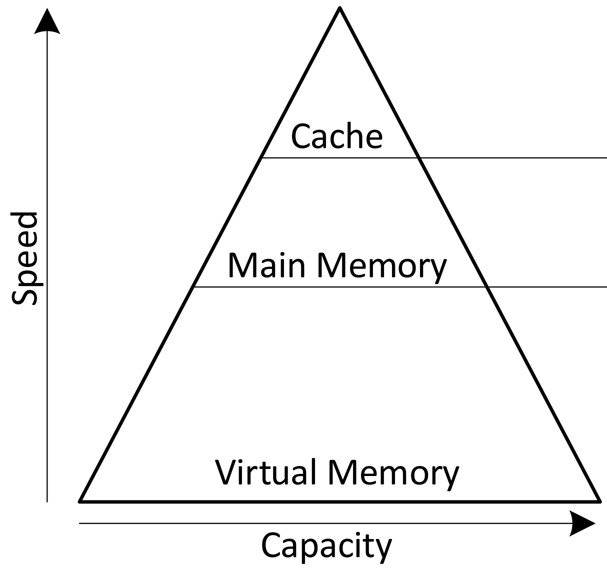
- Make memory system appear as fast as processor
- Use hierarchy of memories
- Ideal memory:
  - **Fast**
  - **Cheap** (inexpensive)
  - **Large** (capacity)

- **But can only choose two!**



Memory Hierarchy





Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$100	0.2 - 3	100+
DRAM	$3	10 - 50	30
SSD	$0.10	20,000	0.05 - 3
HDD	$0.03	5,000,000	0.001 - 0.1

Locality

Exploit locality to make memory accesses fast:

- **Temporal Locality:**
  - Locality in time
  - If data used recently, likely to use it again soon
  - **How to exploit:** keep recently accessed data in higher levels of memory hierarchy
- **Spatial Locality:**
  - Locality in space
  - If data used recently, likely to use nearby data soon
  - **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too

Memory Performance

Memory Performance

- **Hit:** data found in that level of memory hierarchy
- **Miss:** data not found (must go to next level)

Hit Rate	= # hits / # memory accesses
	= 1 – Miss Rate
Miss Rate	= # misses / # memory accesses
	= 1 – Hit Rate

- **Average memory access time (AMAT):** average time for processor to access data

AMAT $= t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$

Memory Performance Example 1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- What are the **cache hit and miss rates?**

Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- t_{cache} = 1 cycle, t_{MM} = 100 cycles
- What is the **AMAT** (average memory access time) of the program from Example 1**?**