TED ÜNİVERSİTESİ

CMPE361 Computer Organization

Department of Computer Engineering

TED University- Fall 2023

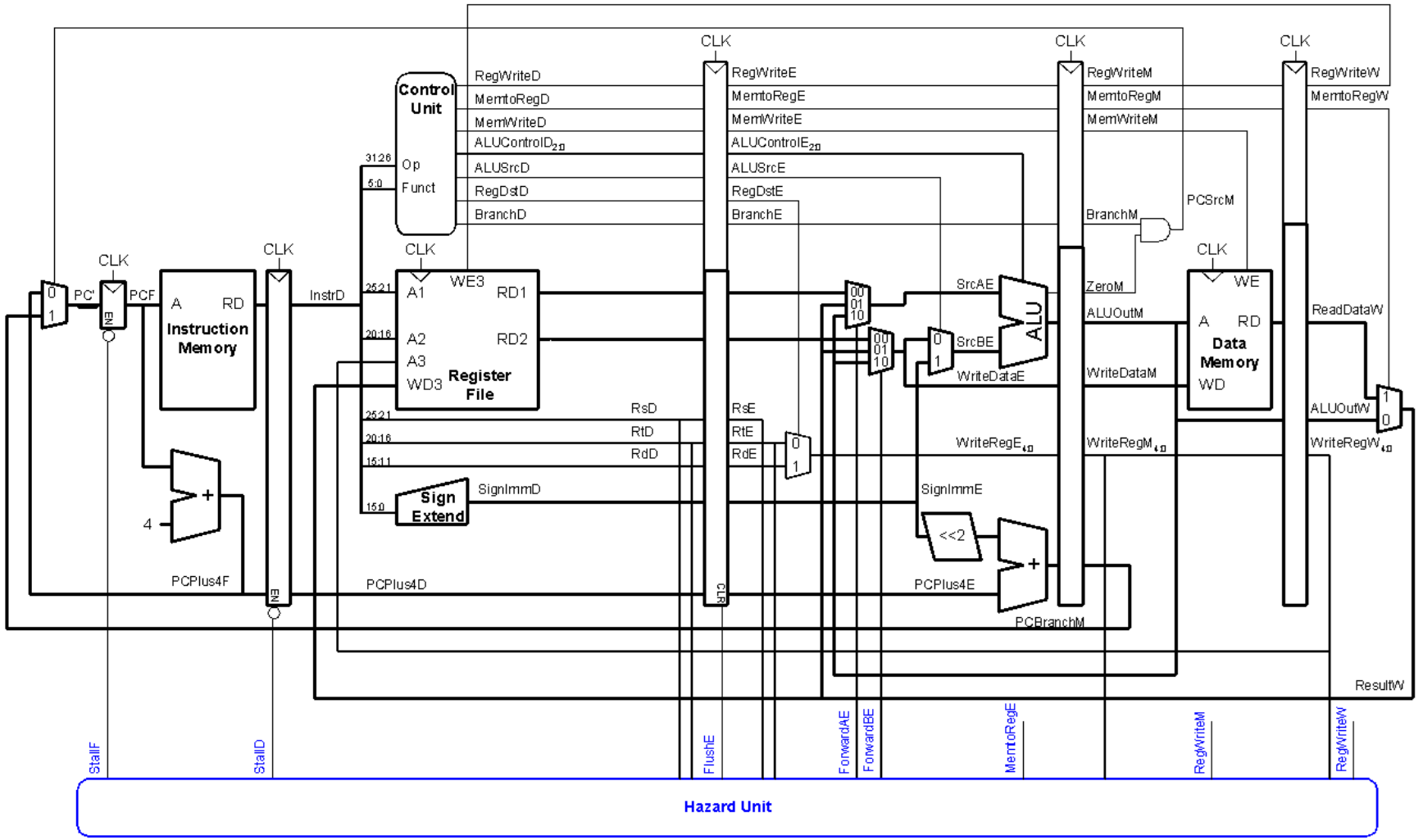
Pipelined MIPS processor – Control Hazards

These Slides are mainly based on slides of the text book (downloadable from the book’s website).

Pipeline Hazards

- When an instruction depends on the result from another instruction that hasn't completed yet, hazards (that can cause wrong results) occur.
- Hazard Types:
  - **Data hazards example:** a register value needed has not yet been written back to register file by the previous instruction
  - **Control hazards:** next instruction to be executed has not been decided yet (caused by branches)

Full Pipeline Design with Data Hazards

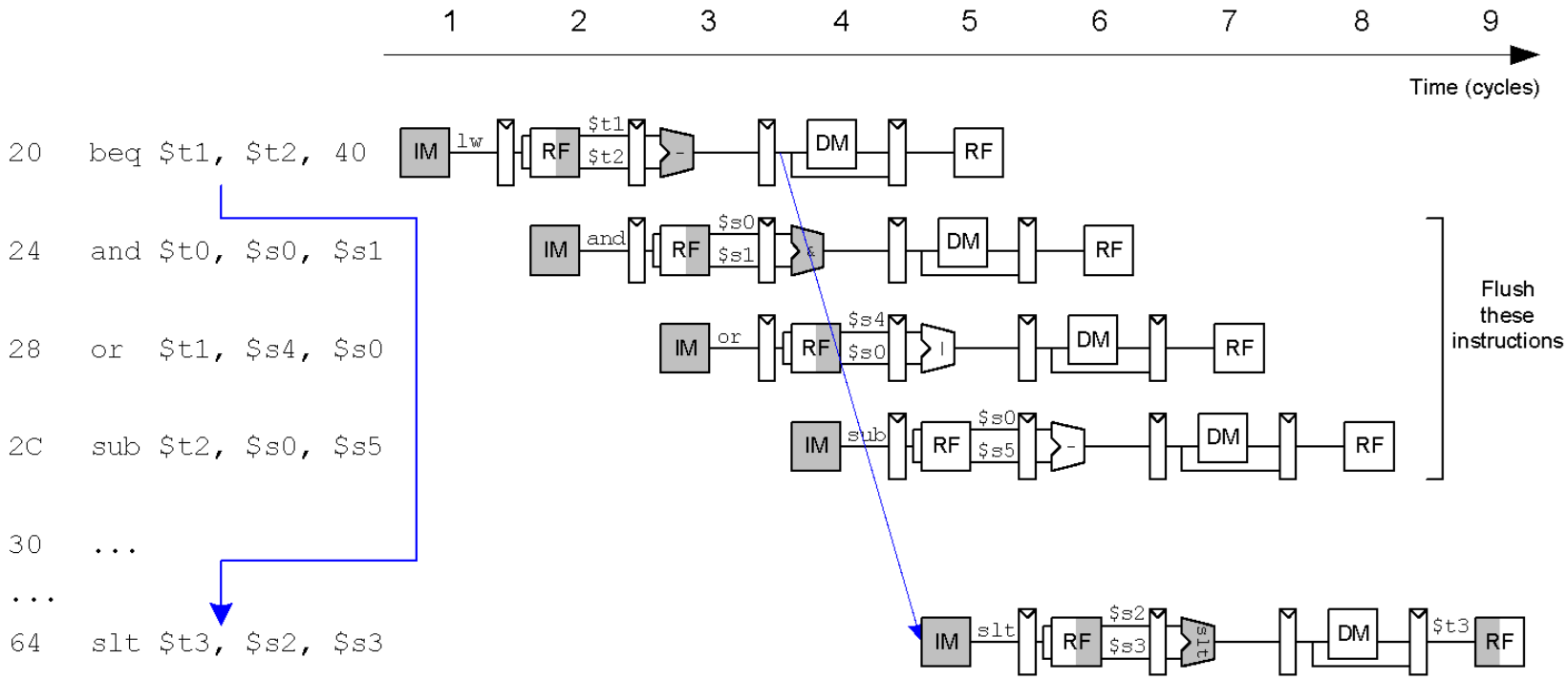


Control Hazards in Pipelined Processor Design

Control Hazards?

- Control hazards occur when the decision of what instruction to fetch **has not** been made by the time the next instruction must be fetched
- For example, the beq instruction presents a control hazard:
  - the processor may not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction needs to be fetched.

Control Hazards Example: beq



The next instruction to be executed could be the one at address 64 (=24+40) or 24 based on the execution of beq

Control Hazards Example: beq

- **When is branch determined?**
  - Branch is determined in the 4th stage
  - The next instruction to be executed will not be known until that point.
  - Simple solution:
    - stall all following instructions 3 stages…
      - This causes extreme degrading!

Alternate solutions to beq Control Hazard

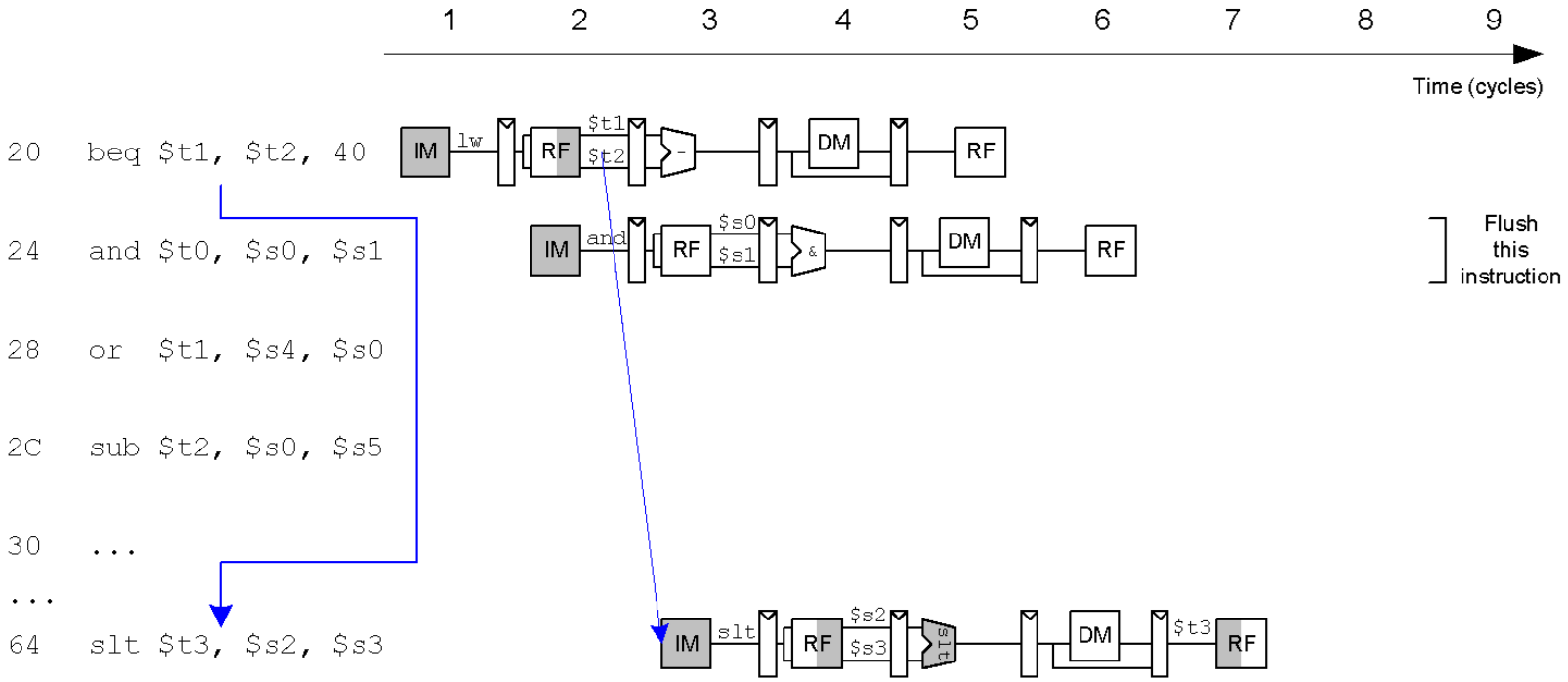
- First alternative solution:
  - Predict the next instruction to be executed.
    - If the prediction comes out to be wrong, it will cause a penalty (known as **branch misprediction penalty)**

Control Hazards Example: beq

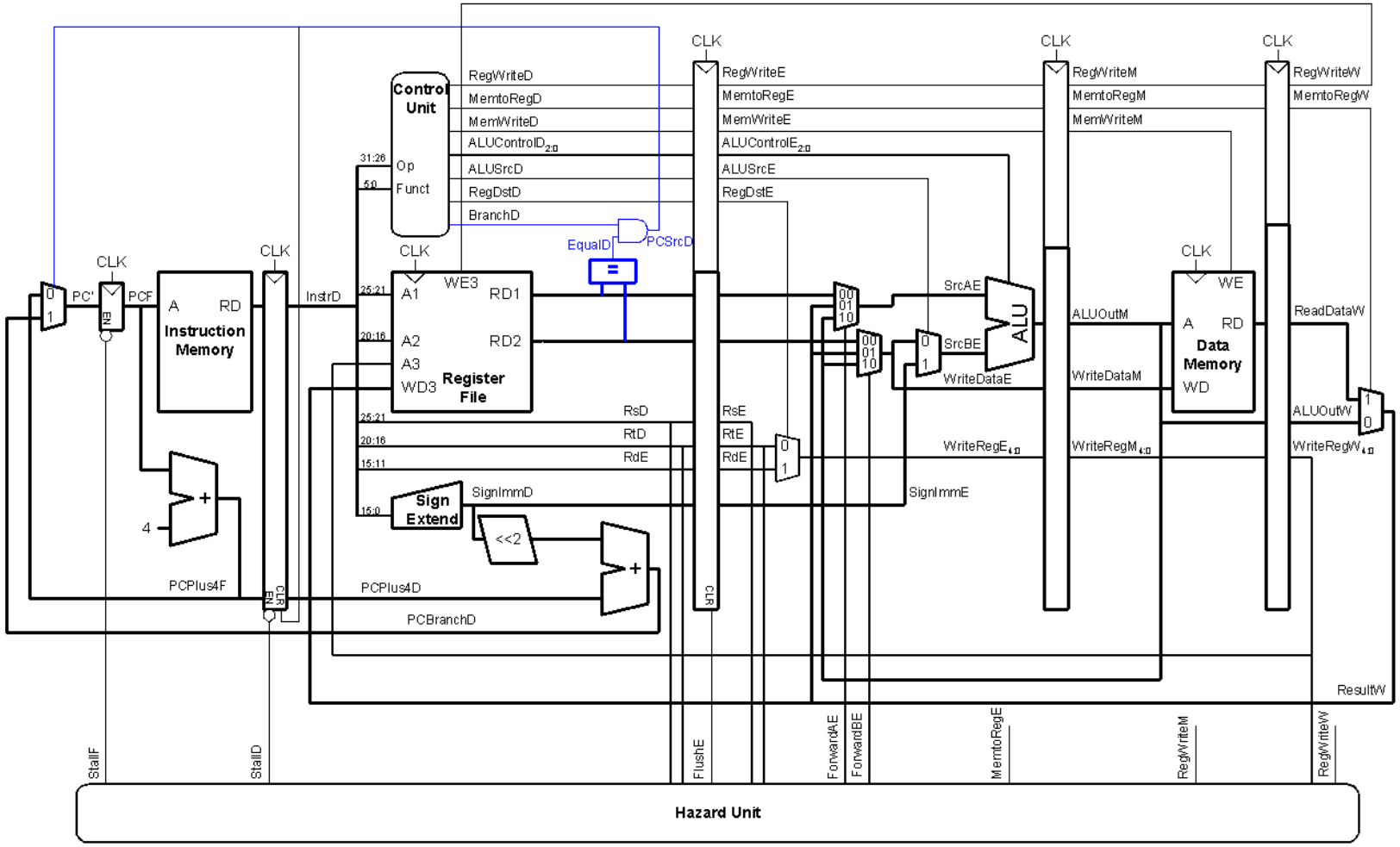
Second alternative solution: Modify the design

- Move the branch decision earlier, to the decode stage.
- Add a dedicated **equality comparator** and **PCBranch adder** to the Decode stage to determine PCSrc earlier.
- The degrading would be limited to one instruction.
- The instruction after beq needs to be flushed out, as it should have not been executed if the branch is taken.
- The synchronous clear input (CLR) connected to PCSrcD so that the incorrectly fetched instruction just after beq can be flushed (cleared), if the result is address 64.

Early Branch Resolution



Design with Early Branch Resolution



PcBranch adder is moved to decode stage.

Clear signal is added to pipeline register bewteen Fetch and Decode Stage.

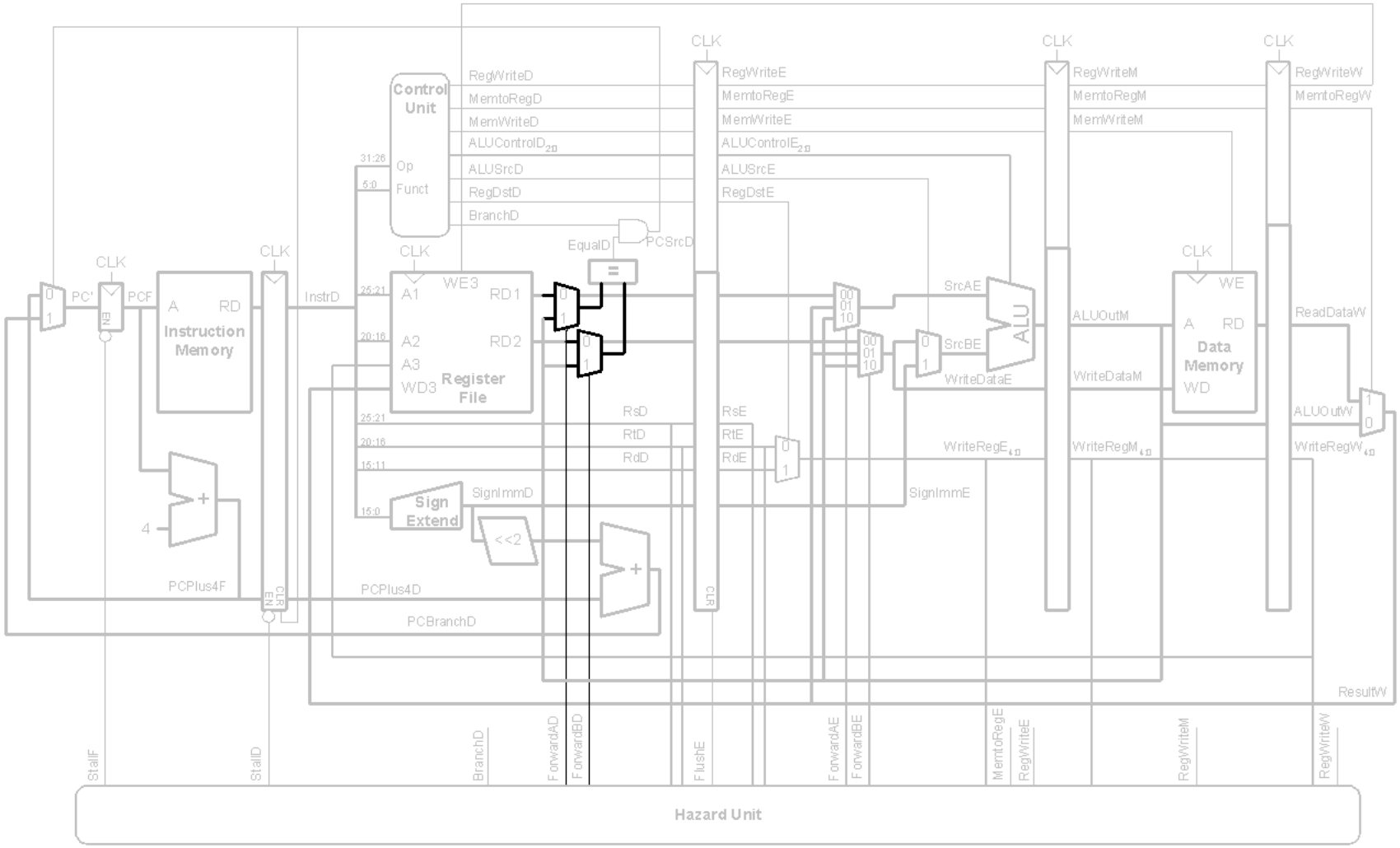
beq may be dependent on previous instructions

- If one of the source registers of beq is computed by the previous instruction and has not yet been written to the RF, the wrong value will be read.
  - How to take care of this?

Dependency of beq on previous instruction

- There are beq dependency possibilities that may cause wrong data flow:
  - If beq depends on
    - writeback stage: no problem.
    - memory stage: forward to the equality comparator through 2 new multiplexers.
    - ALU in execute stage or lw instruction in memory stage stall the pipeline at decode stage.
- Enhance the decode stage with **two multiplexer** to allow dependency handled

Handling data dependencies for branch



Control Forwarding & Stalling Logic

- **Forwarding logic for Decoding Stage:**

```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM AND (~MemToRegM)
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM AND (~MemToRegM)
```

- **Stalling logic: Branch need to make decision in decode stage. If either of the source registers depends on an ALU instruction in the Execute stage or on a lw in the Memory stage, the processor must stall until the sources are ready.**

```
branchstall = BranchD AND
 [RegWriteE AND ((WriteRegE == rsD) OR (WriteRegE == rtD))]//stall when branch in
decode stage depends on lw in memory stage or an ALU instruction in execute stage
                OR
 [MemtoRegM AND ((WriteRegM == rsD) OR (WriteRegM == rtD))]//stall when branch in
decode stage depends on lw instruction in memory stage
```

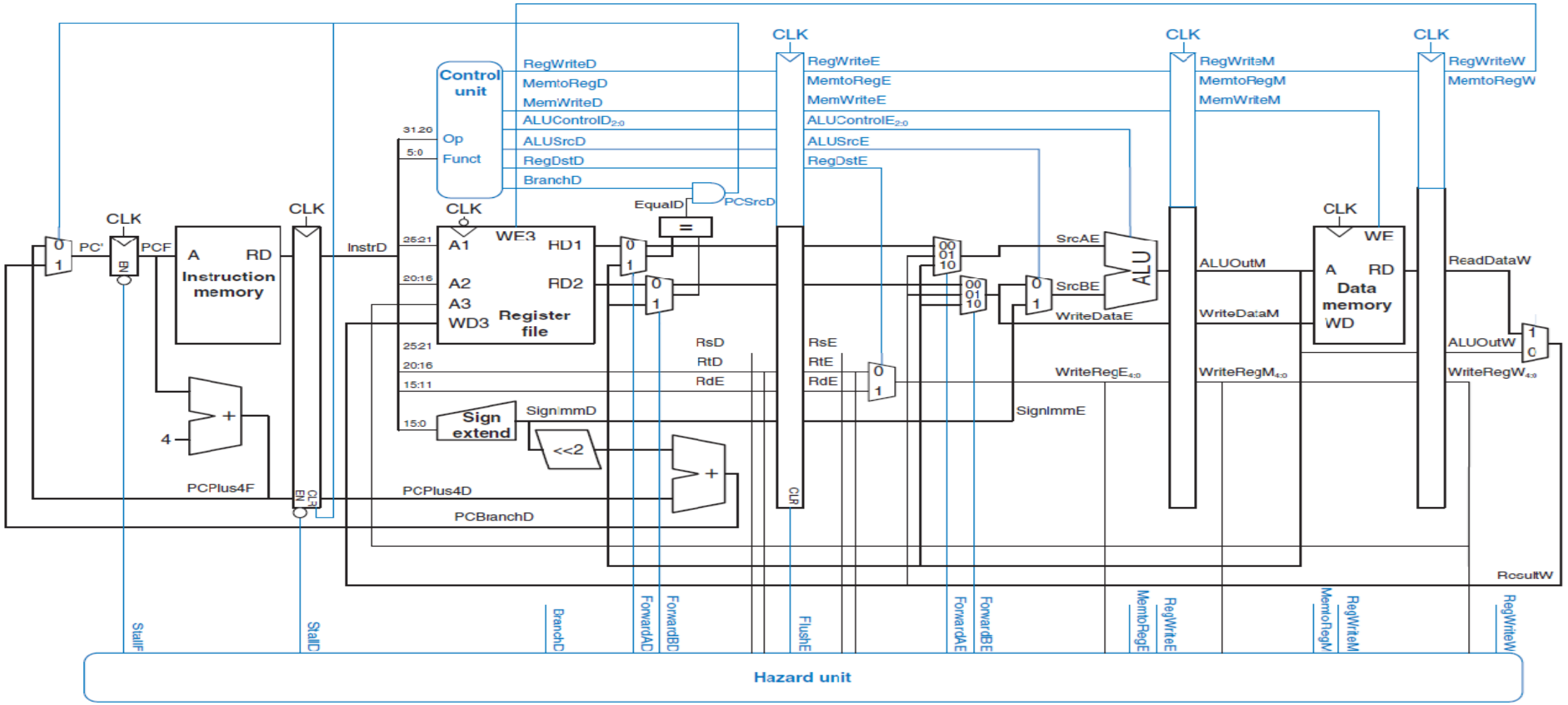
```
StallF = StallD = FlushE = lwstall OR branchstall
```

Now the processor might stall due to either a load or a branch hazard:

Remember the generation of lwstall signal:

lwstall = ((rsD = = rtE) OR (rtD = = rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall



Pipelined processor with full hazard handling

Performance

- The pipelined processor, clock cycle is determined by the longest phase.
- Ideally pipelined processor would have a CPI of 1
  - a new instruction can be issued every cycle, if the instructions are independent of each other.
- However, stall or flush situations exists.
- So the average CPI is normally higher, depending on the program.

Example case

- SPECINT2000 benchmark related to instruction mix:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches miss predicted
  - All jumps flush next instruction
- **Compute the average CPI:**

i.e. 40% of load have CPI of 2 cycles and 60% of loads have CPI of 1 cycle.

75% of branches have CPI of 1 cycle and 25% of them have CPI of 2 cycles

All jumps have CPI of 2 cycles.

Pipelined Performance Example

- SPECINT2000 benchmark: Standard performance Evaluation Corporation INT2000
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches miss-predicted
  - All jumps flush next instruction
- **What is the average CPI?**
  - Load/Branch CPI = 1 when no stalling, 2 when stalling
  - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
  - $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

Average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1)

= 1.15

Pipelined Performance

- Pipelined processor critical path:

$$T_c = \max \{$$

Fetch Stage $\{ t_{pcq} + t_{mem} + t_{setup}$

Decode Stage $\{ 2(t_{pcq} + t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$

Execute Stage $\{ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}$

Memory Stage $\{ t_{pcq} + t_{memwrite} + t_{setup}$

Writeback Stage $\{ 2(t_{pcq} + t_{mux} + t_{RFwrite}) \}$

the read operation must happen in half cycle. Write operation takes another half cycle. So, the clock period must be TWICE of time taken, for decode and writeback stages.

Pipelined vs. Single-Cycle Performance Comparison

Element	Parameter	Delay (ps)
Register clock-to-Q	T_{pcq}	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

Pipelined: $T_c = 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$

$= 2[150 + 25 + 40 + 15 + 25 + 20]$ ps **= 550 ps**

Pipelined Performance Example

Program with 100 billion instructions

Execution Time = (# instructions) × CPI × T_c

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

= 63 seconds

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	92.5	1
Pipelined	63	1.47

Execution pipeline processor is not 5 times faster than SCP, because:-

1. Stalls due to hazards (data and control) have a some overhead.
2. Clock to Q (T_{pcd}) and register setup (T_{setup}) time required for all of the pipeline registers is main culprit for not achieving the ideal performance for pipelined processor.