Computer Architecture

Processor: Part 3

Pipelined microarchitecture

Agenda

- Parallelism
- Pipelined processors
- Hazards and the solutions
- Advanced microarchitecture

Several terms for describing the system speed

The speed of a system is characterized by the **latency** and **throughput** of information moving through it.

- **Token**: a group of inputs that are processed to produce a group of outputs. (i.e., a task)
- **Latency**: the time required for one token to pass through the system from the start to end.
- **Throughput**: the number of tokens that can be produced per unit time.

Example: cookie throughput and latency

Ben is making cookies for a party. It takes 5 minutes to roll cookies and place them on the tray. Then, it takes 15 minutes for the cookies to bake in the oven. Once the cookies are baked, he starts another tray. What is Ben's throughput and latency for a tray of cookies?

In this example,

- The token is to cook a tray of cookies.
- The latency is 5 minutes + 15 minutes = 20 minutes = ⅓ hours per tray.
- The throughput is 3 trays/hour.

Parallelism

Parallelism: processing several tokens at the same time. It has two forms:

- **Spatial parallelism**: It uses multiple copies of the hardware to process multiple tasks simultaneously.
- **Temporal parallelism** (**Pipelining**): It breaks down a task into stages, allowing different parts of the task to be executed simultaneously in a staggered manner.

Throughput in parallelism

Given a task with latency ***L***.

- In a system without parallelism, the throughput is ***1/L***.
- In a spatially parallel system with ***N*** copies of the hardware, the throughput is ***N/L***.
- In a temporally parallel system, the task is broken into ***N*** steps, and only one copy of the hardware. If the ***longest step*** has a latency $\boldsymbol{L_1}$, the pipelined throughput is $\boldsymbol{1/L_1}$.

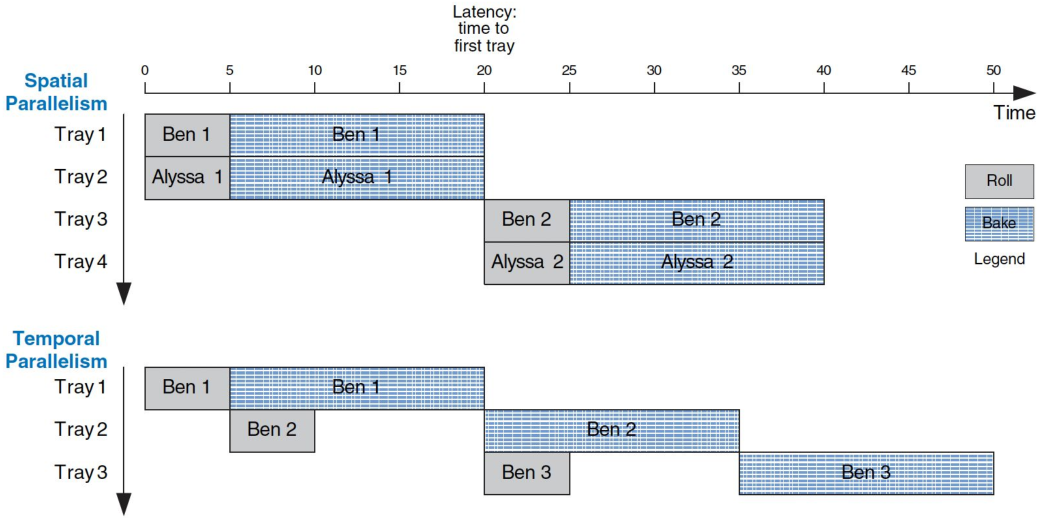
Example: Cookie parallelism

Ben is considering using spatial and/or temporal parallelism to bake cookies.

- Spatial parallelism: Ben asks Alyssa to help out. She has her own cookie tray and oven.
- Temporal parallelism: Ben gets a second cookie tray. Once he puts one cookie tray in the oven, he starts rolling cookies on the other tray rather than waiting for the first tray to bake.

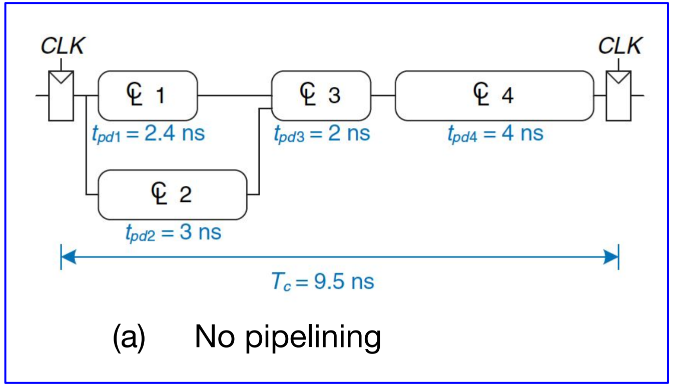
What is the throughput and latency using spatial parallelism? Using temporal parallelism? Using both?

Example: Cookie parallelism

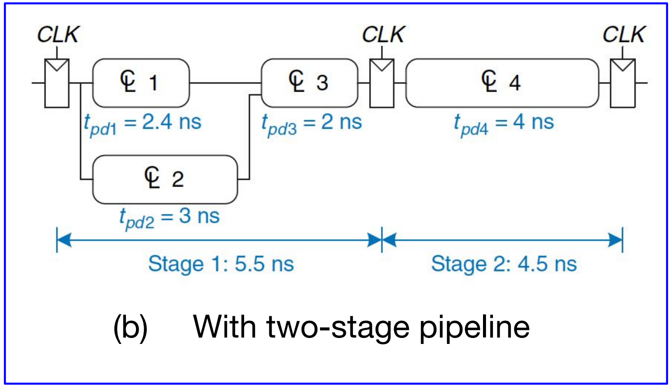


- Latency is the time to complete one task from start to finish. So, it is ⅓ hours in all cases.
- Throughput is the number of cookie trays per hour. So, with spatial parallelism, Ben and Alyssa each complete one tray every 20 minutes, and the throughput is 6 trays/hour. With temporal parallelism, Ben puts a new tray in the oven every 15 minutes, and the throughput is 4 tray/hour. (needs extra 5 minutes at the beginning for filling the pipeline)
- If Ben and Alyssa use both techniques, they can bake 8 trays/hour.

Example: Circuit parallelism



(a) No pipelining



(b) With two-stage pipeline

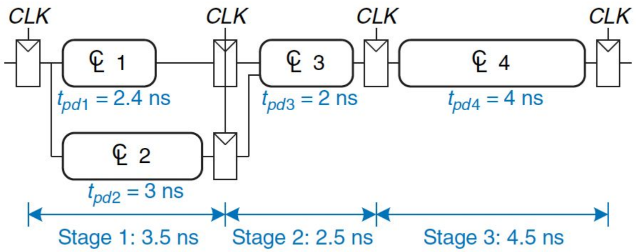
Assume the register has a *clock-to-Q* propagation delay of 0.3 ns and a *setup* time of 0.2 ns.

(a) The cycle time T_C = *0.3+0.3+2+4+0.2=9.5 ns*. The circuit latency is *9.5 ns* and a throughput is 1/9.5 ns = 106MHz

(b) The cycle time = *max* { T_{C1}, T_{C2} } = *5.5 ns*. The latency is T_C *x 2 = 11 ns* since a task should be completed by the two stages. The throughput is 1/5.5 ns = 182 MHz.

A two stage pipelining almost doubles the throughput and slightly increases the latency.

Example: Circuit parallelism



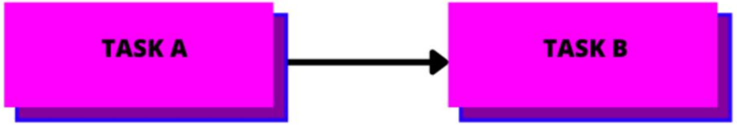
(c) circuit with three-stage pipeline

(c) The cycle time T_C = max { T_{C1}, T_{C2}, T_{C3} } = 4.5 ns. The latency = T_C x 3 = 13.5 ns. The throughput is 1/4.5 = 222 MHz

The bane of parallelism

Parallelism is one technique for designing high-performance digital system. But, it do not apply to all situations.

- The bane of parallelism is dependencies.
  - If a current task is dependent on the result of a prior task, rather than just prior steps in the current task, the task cannot start until the prior task has completed.

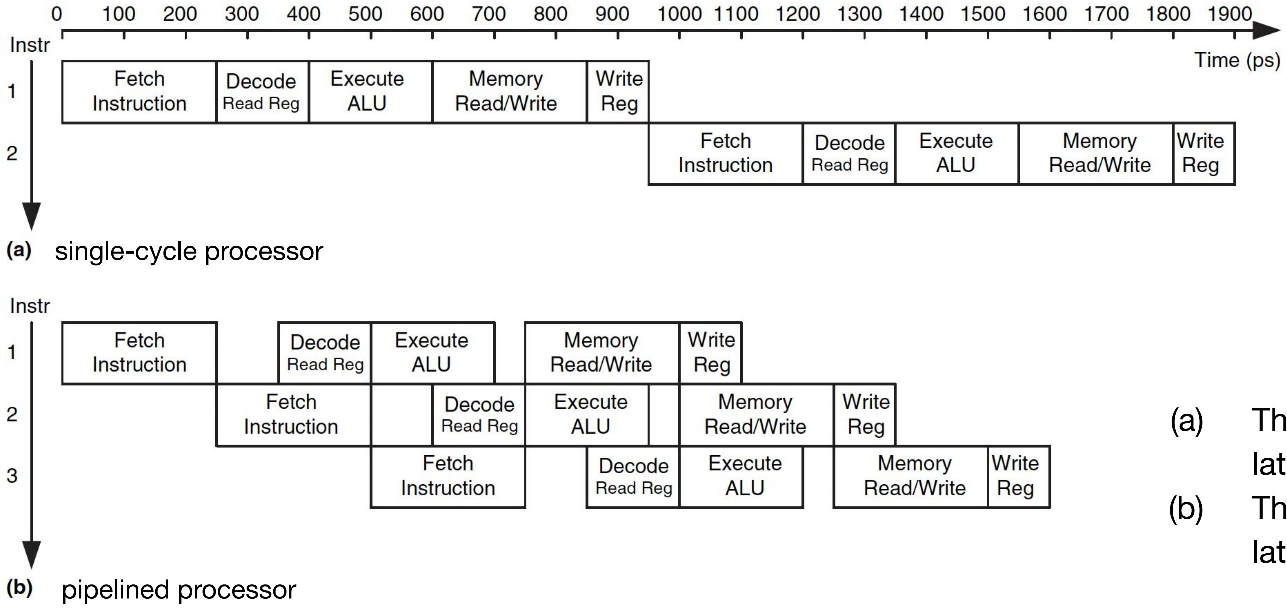


Pipelined MIPS processor (H&H Section 7.5)

To design a pipelined MIPS processor, we often subdivide the *single-cycle* MIPS processor into five pipeline stages.

Fetch	The processor reads the instruction from instruction memory.
Decode	The processor reads the source operands from the register file and decodes the instruction to produce the control signals.
Execute	The processor performs a computation with the ALU.
Memory	The processor reads or writes data memory.
Writeback	The processor writes the result to the register file, when applicable.

Timing diagrams for single-cycle and pipelined processors



(a) single-cycle processor

(b) pipelined processor

(a) The single-cycle processor has a latency 950 ps.

(b) The pipelined processor has a latency 250 x 5 = 1250 ps.

Exercise: calculate the throughput

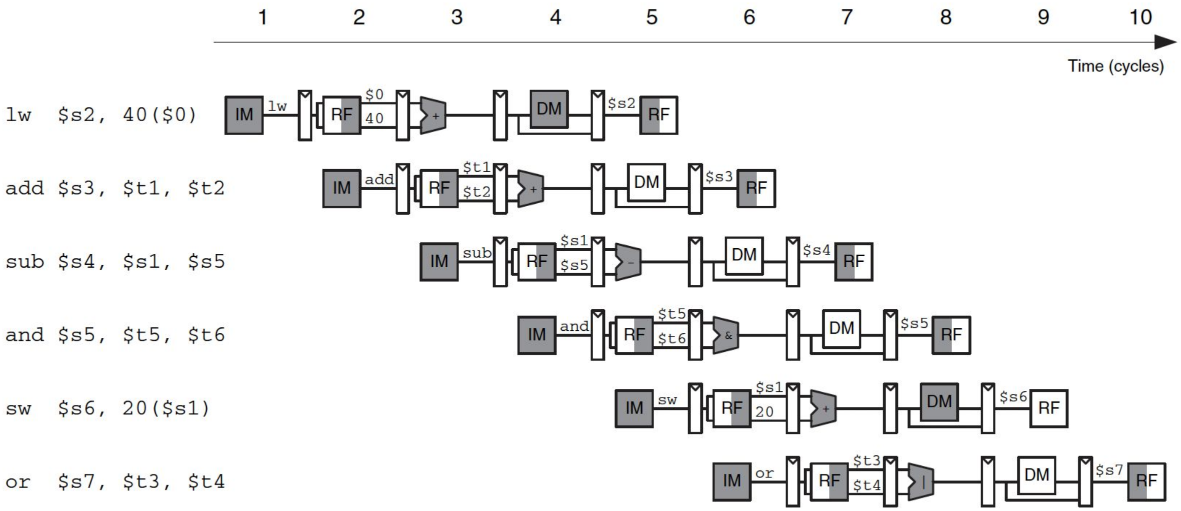
What are the throughputs of the single-cycle processor and the pipelined processor described in the previous slides?

Solution

What are the throughputs of the single-cycle processor and the pipelined processor described in the previous slides?

- Single-cycle processor:
  - Cycle time = 950 ps, throughput = 1/950 ps = 1.05 x 10^9 / second
- Pipelined processor:
  - Cycle time = 250 ps, throughput = 1/250 ps = 4 x 10^9 / second

Abstract view of pipeline in operation



Row: showing the clock cycles in which a particular instruction is in each stage;

Column: showing what the various pipeline stages are doing on a particular cycle.

The **shadow** in a device indicates the device is used.

The register file (**RF**) completes reading and writing in one clock cycle; it is applicable because write/read on RF is faster than on memory. A cycle is long enough for RF to do read and write. We shade the left half if the RF is being read and the right half when it is being written.

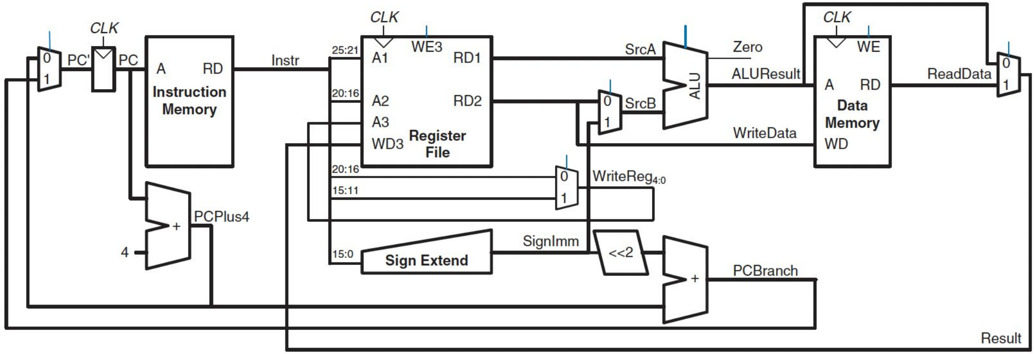
IM: instruction memory;
RF: register file; it is written in the first part of a cycle and read in the second part, as suggested by the shading.
DM: data memory.

Pipelined datapath

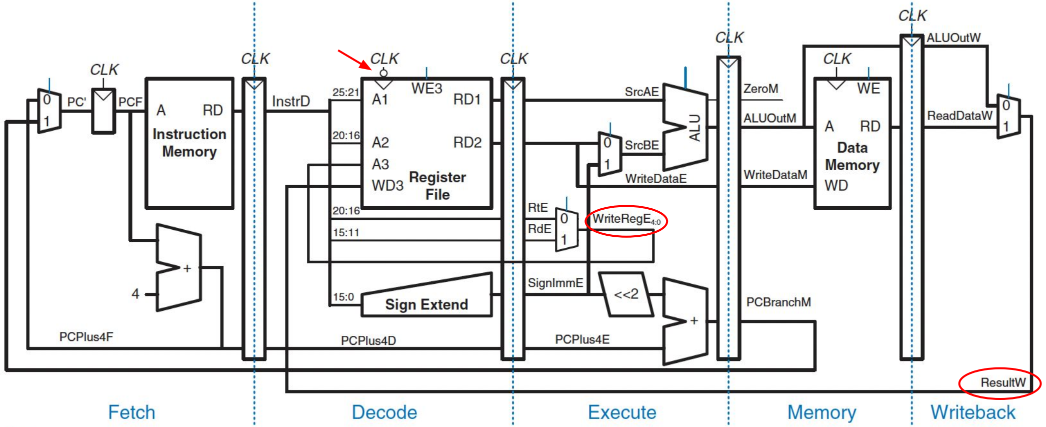
The pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by pipeline registers. In (b), there are four **pipeline registers** that separate the datapath into five stages.

<u>Pipeline registers store the instructions, data, and control signals generated in each stage, respectively</u>.

In the pipelined datapath, it should guarantee that <u>all signals</u> associated with a particular instruction must advance through the pipeline in unison. So, in (b) there is an error in datapath: the **ResultW** is from the pipeline register of Writeback stage, but the **WriteRegE** (indicating the address to write back the *ResultW*) is from the pipeline register of Decode stage.

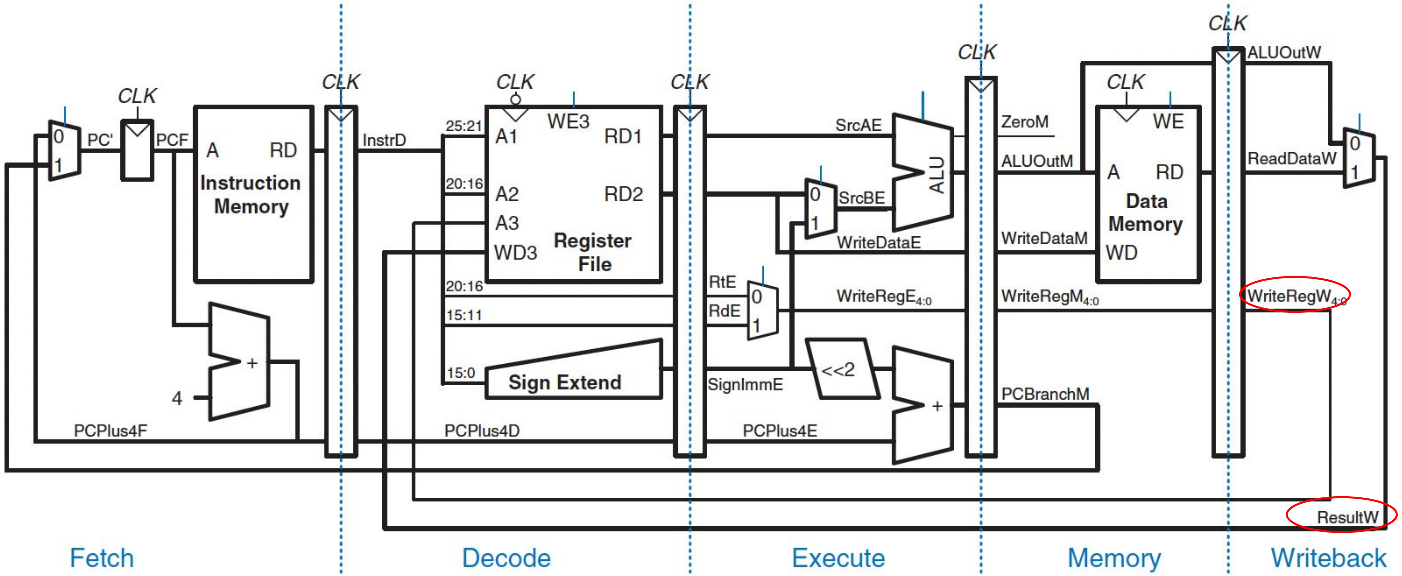


(a) Single-cycle datapath



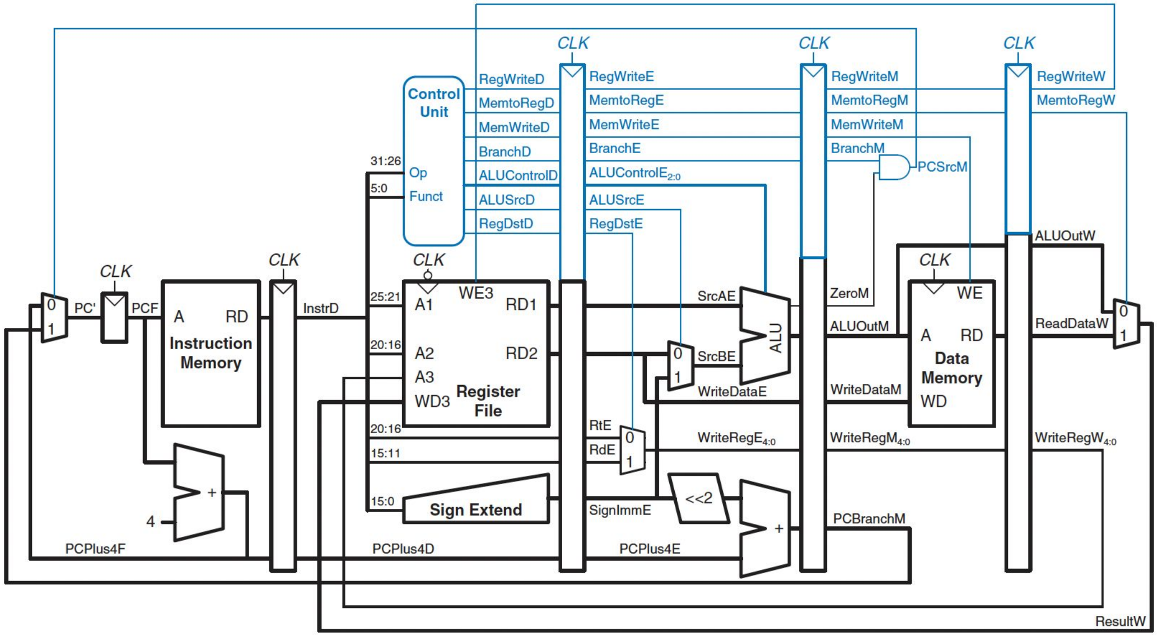
(b) Pipelined datapath. Note the inverter on the CLK in Register File; this ensures that the Register File writes on the rising edge the CLK signal. (Recall the D-flipflop)

Corrected pipelined datapath



The WriteReg signal is pipelined along through the Memory and Writeback stages so that remains in sync with the rest of the instruction.

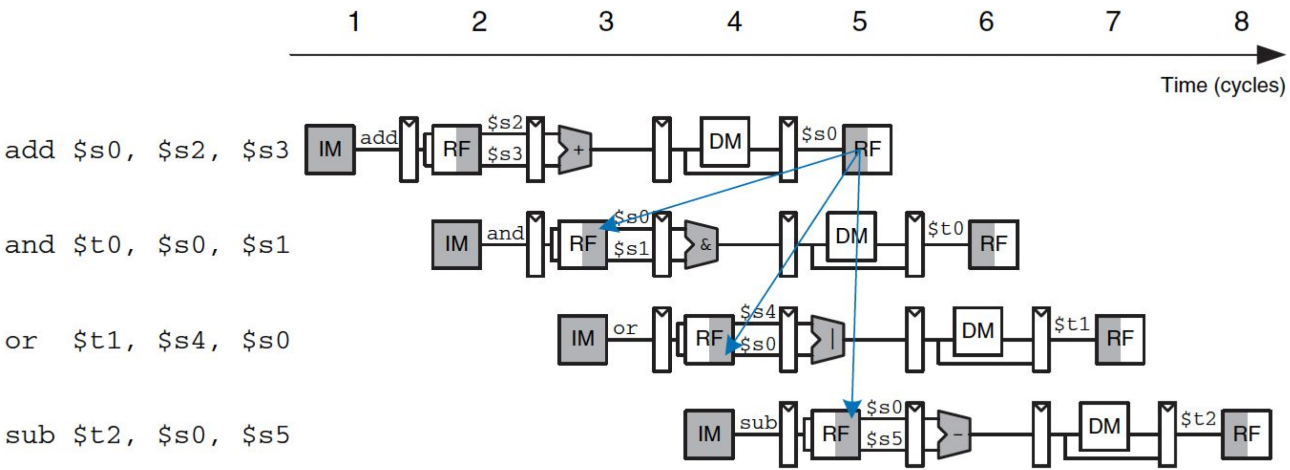
Pipelined processor with control



The control unit of a pipelined processor is same as the single-cycle processor.

The control unit examines the `opcode` and `funct` fields of the instruction in the Decode stage and produce the control signals.

Hazard



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

From the assembly code, the first instruction computes the value in $s0, then, the following instructions will use the value. But when we run the program using a pipelined processor, there are problems. add writes the result into $s0 in cycle 5, while the following and reads $s0 in cycle 3, and the or reads $s0 in cycle 4. They will obtain the wrong value.

Hazard

A **hazard** in a pipelined system refers to a situation that causes incorrect behavior or delays in instruction execution within a pipeline.

- Hazards occur when the concurrent execution of multiple instructions leads to conflicts or dependencies that the pipeline cannot handle efficiently.
- Three types: data hazard, control hazard, and structural hazard.

Data hazards

Cause: when an instruction depends on the result of a previous instruction that has not yet completed.

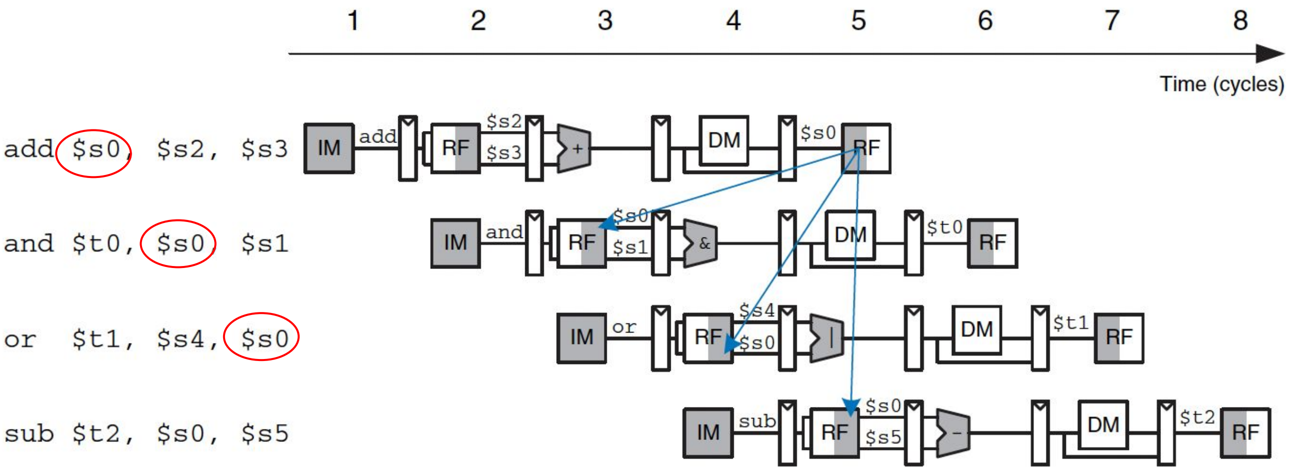
Types:

- Read After Write (RAW) hazard
- Write After Write (WAW) hazard
- Write After Read (WAR) hazard

Note: WAW and WAR do **not occur** in MIPS because MIPS avoids these hazards by adhering to strict in-order execution.

Read After Write Hazards

RAW: occurs when a later instruction tries to read a value that has not yet been written by an earlier instruction.



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

Write After Write Hazards

WAW: Occurs when two instructions write to the same location in an overlapping way, leading to potential overwrite errors.

WAW example (in other architectures that support out-of-order execution, not MIPS):

```
SUB R1, R2, R3  #writes to R1
ADD R1, R4, R5  #writes to R1 again
```

The above code is not a hazard in MIPS because the instructions are completed in order in MIPS. So, write to R1 always occurs after the first.

Write After Read Hazards

WAR: Occurs when a later instruction writes a value before an earlier instruction has read it.

WAR example (in other architectures that support out-of-order execution, not MIPS):

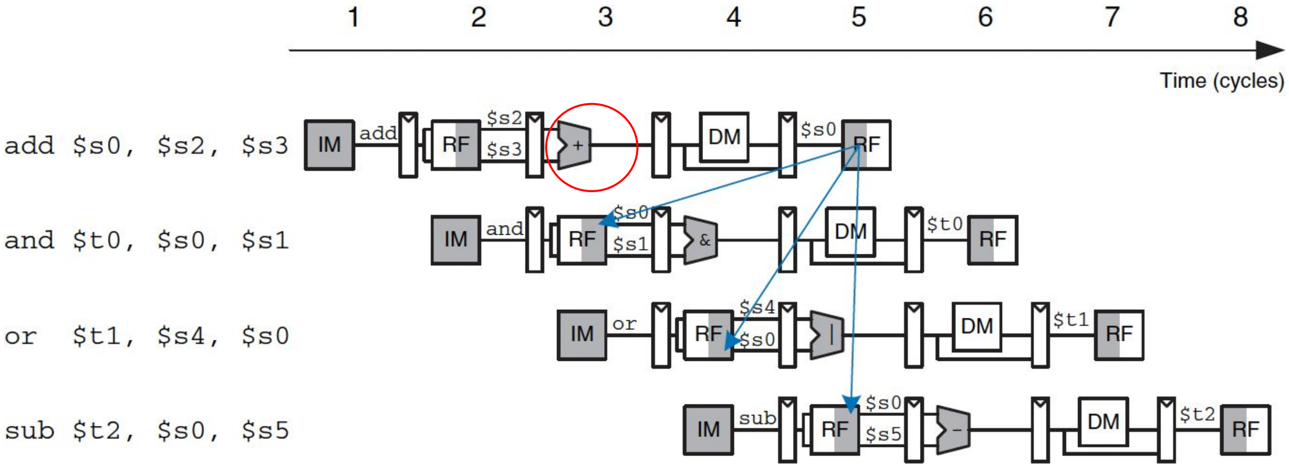
```
SUB R1, R2, R3 #reads R2 and R3
ADD R2, R4, R5 #writes to R2 before the first
instruction reads it
```

Note: The above code is not a hazard in MIPS, because reads occur in the decode stage, well before the writes in the writeback stage.

Solving RAW hazards

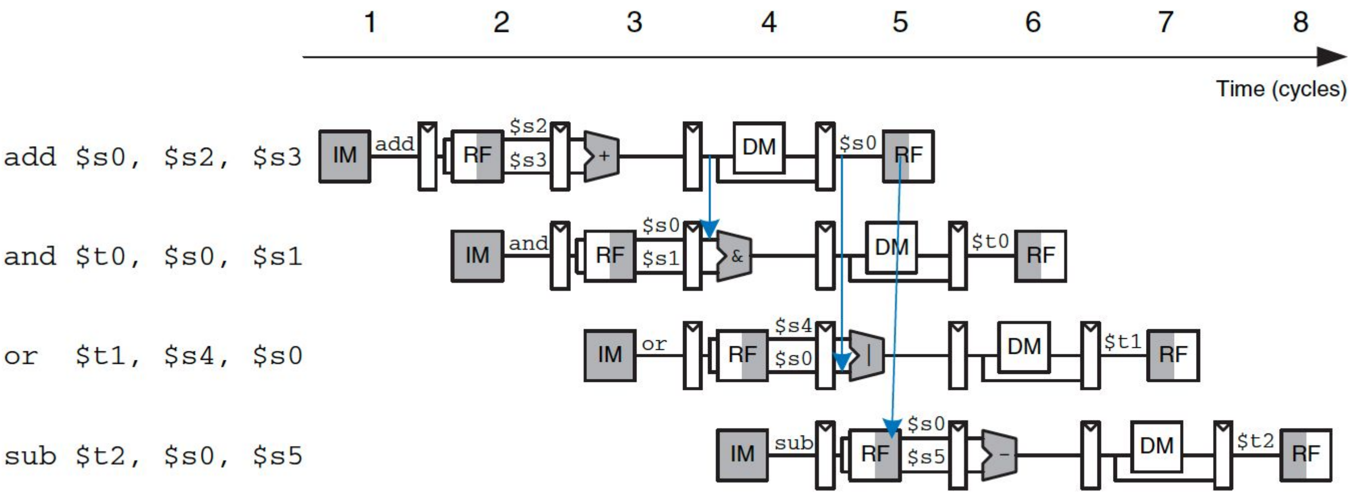
- Using forwarding (also called bypassing)
- Using stall

Solving data hazard with forwarding



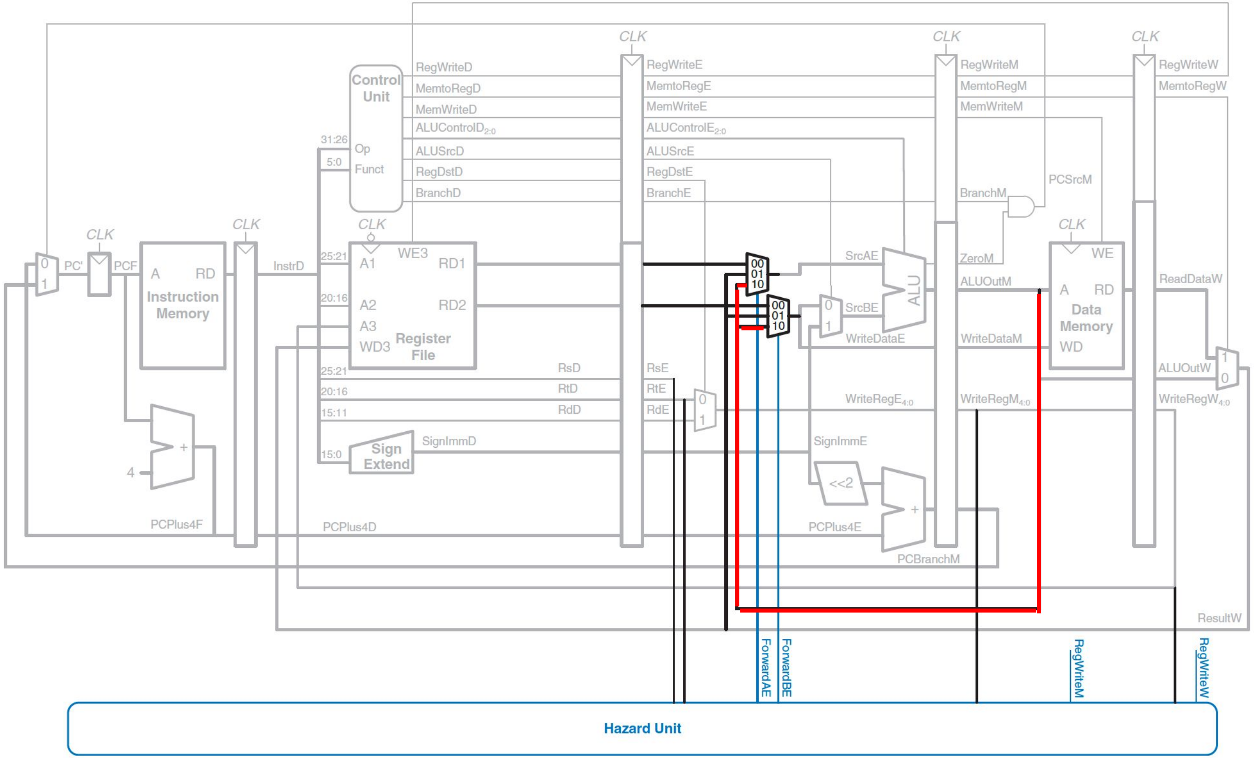
The value that will be written into `$s0` in cycle 5 has been calculated in cycle 3 already.

Solving data hazard with forwarding



The value that will be written into `$s0` in cycle 5 has been calculated in cycle 3 already. In cycle 4 (Memory stage of `add`; Execute stage of `and`), we can forward the value to the `and` instruction. In cycle 5 (Writeback stage of `add`; Execute stage of `or`), we can forward the value to the `or` instruction.

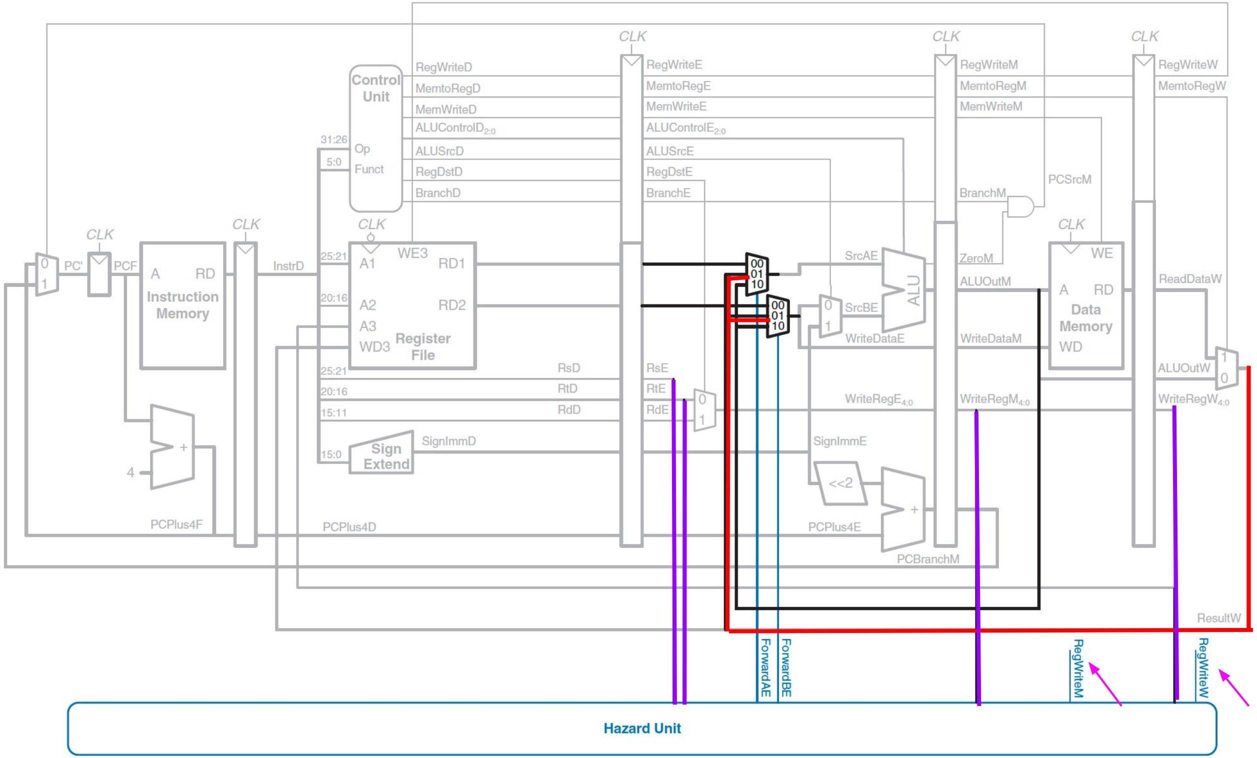
Datapath with forwarding: Memory ⇒ Execute



It adds the datapath (in red) and two multiplexers to control the inputs of the ALU.

Alaso, a hazard detection unit is added.

Datapath with forwarding: Writeback ⇒ Execute



The hazard unit receives:

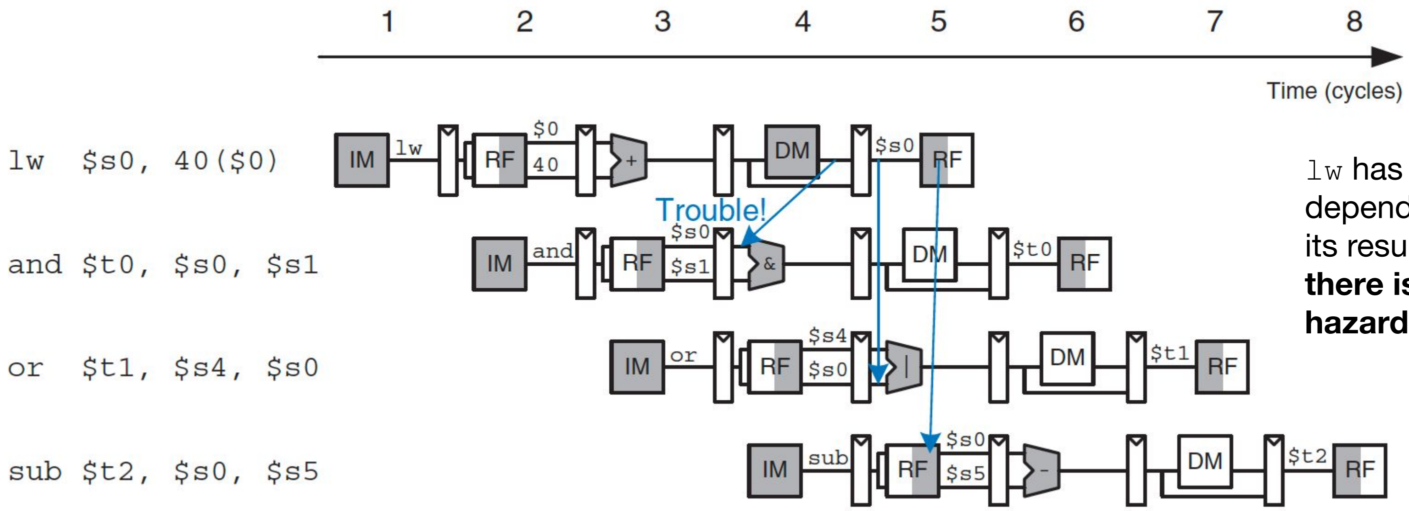
- the **two source registers** in the Execute stage, and
- the **destination registers** in Memory and Writeback stages.
- **RegWriteM** and **RegWriteW** signals from the Memory and Writeback stages.

It should forward from a stage:

- If that stage will write a destination register and the destination register matches the source register.

Load-Use data hazards

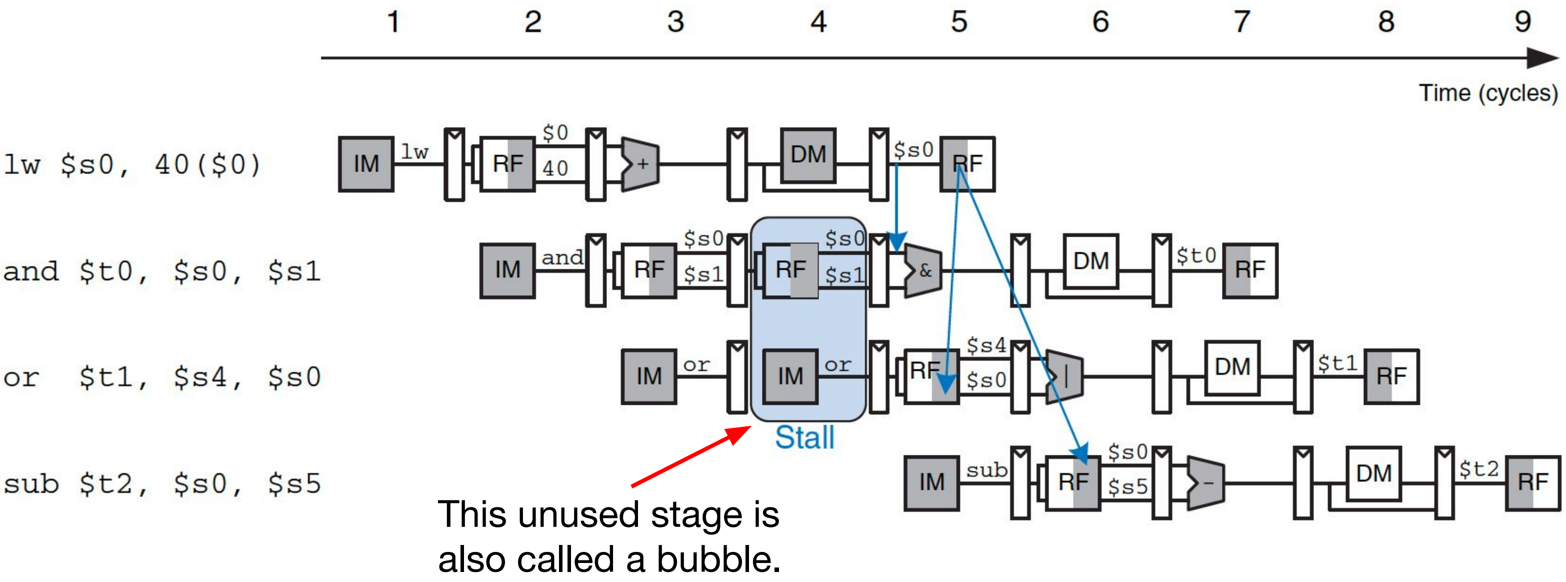
Load-use data hazard: occurs when an instruction requires data loaded from memory by a preceding load instruction before the load instruction has completed accessing memory and updating the register file.



`lw` has a two-cycle latency; a dependent instruction cannot use its result until two cycles later. So, **there is no way to solve this hazard with forwarding.**

Solving data hazards with stalls

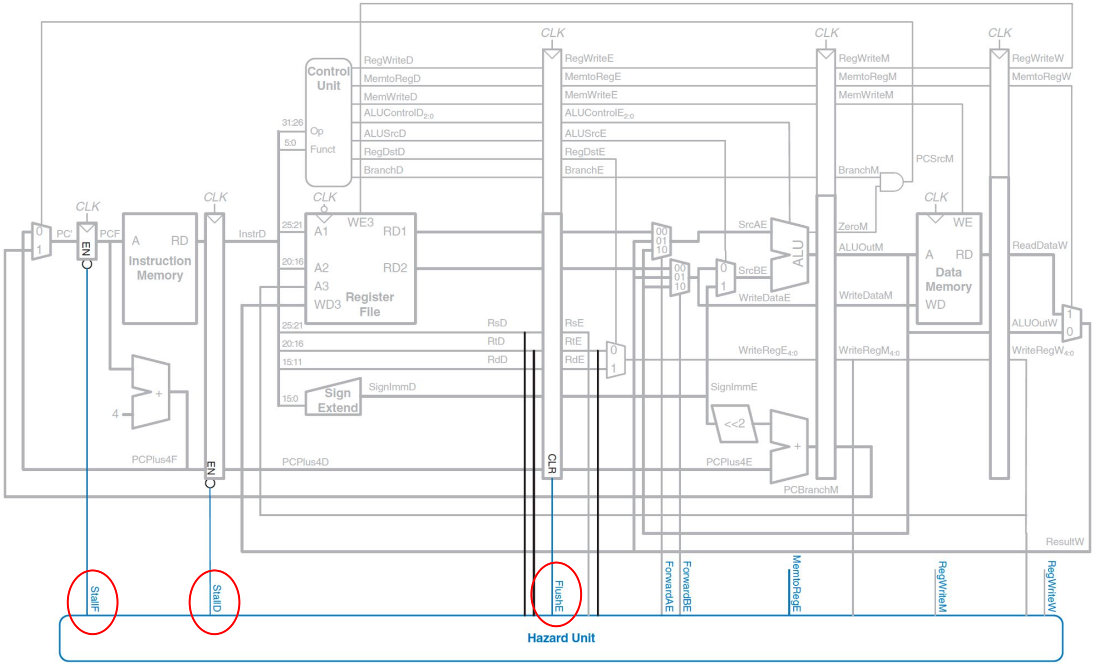
An alternative solution to RAW hazards is to **stall** the pipeline, holding up operation until the data is available.



```
lw $s0, 40($0)
and $t0, $s0, $s1
or  $t1, $s4, $s0
sub $t2, $s0, $s5
```

This unused stage is also called a bubble.

Datapath with stall



It adds enable inputs (EN) to the Fetch and Decode pipeline registers and a synchronous reset/clear (CLR) input to the Execute pipeline register.

When an `lw` stall occurs, ***StallD*** and ***StallF*** are asserted to force the Decode and Fetch stage pipeline registers to hold their old values. ***FlushE*** is asserted to clear the contents of the Execute stage pipeline register, introducing a bubble.

Control hazards

Cause: arise from the need to make decisions based on branch or jump instructions before knowing the target address or branch condition.

Example: the `beq` instruction in MIPS.

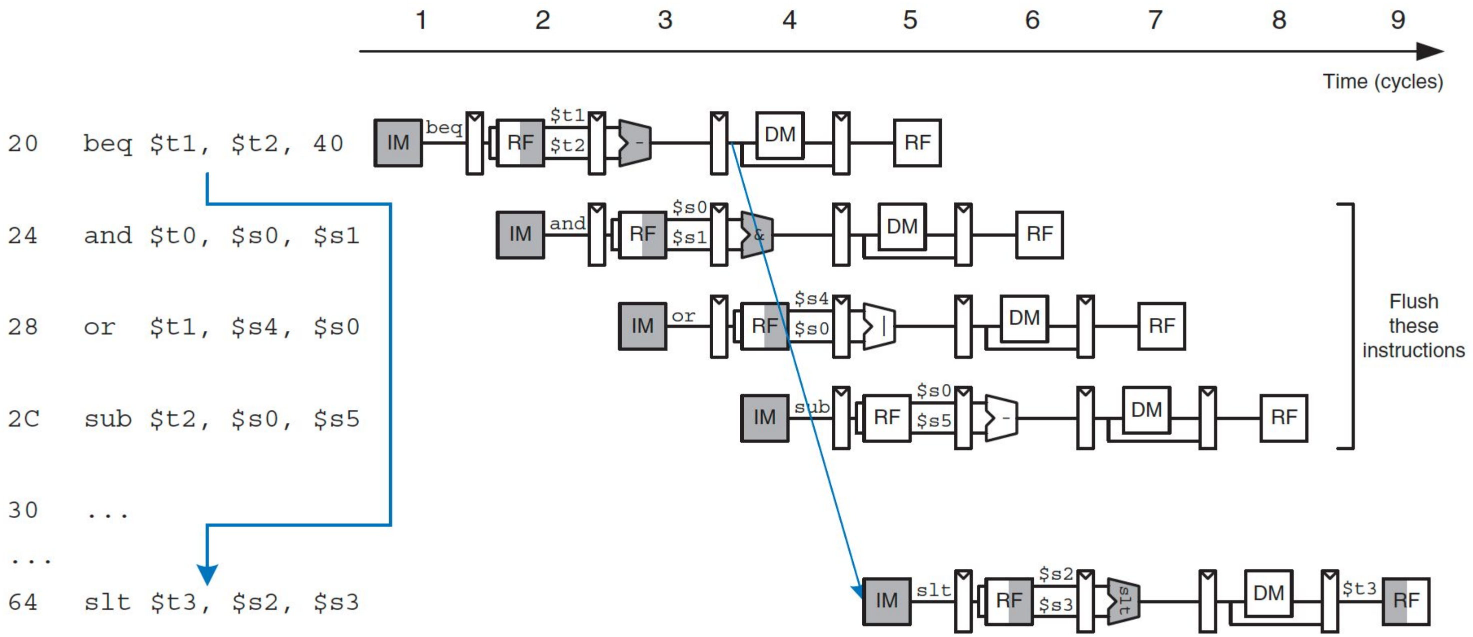
```
beq  r1, r2, LABEL     # Branch if R1 == R2
add  r3, r4, r5        # Next sequential instruction
LABEL: sub  r6, r7, r8 # Target of the branch
```

- If the branch is taken, the pipeline must start fetching from `LABEL`
- If the branch is not taken, the pipeline continues with `add`
- It is until in the Execute stage, the pipeline can know which path to take.

Solving control hazards

- Stall the pipeline until the decision is made
  - Since the decision is made in the Memory stage, the pipeline would have to be stalled for three cycles at every branch.
  - This would severely degrade the system performance.
- Predict whether the branch will be taken and begin executing instructions based on the prediction.
  - Once the branch prediction is available, the processor can throw out the instructions if the prediction was wrong.
  - **Branch misprediction penalty**: when the processor guess wrong, it must flush the instructions fetched based on the incorrect prediction and restart the pipeline with the correct instructions. This causes wasted cycles and reduces overall performance.

Flushing when a branch is taken

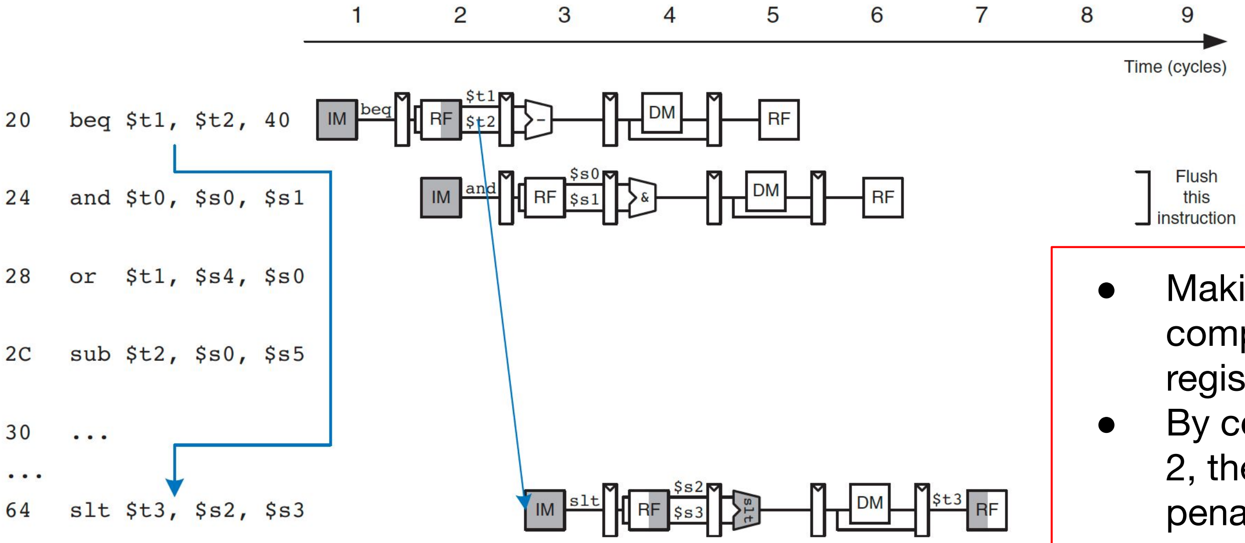


```
20   beq $t1, $t2, 40

24   and $t0, $s0, $s1

28   or  $t1, $s4, $s0

2C   sub $t2, $s0, $s5

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

A branch from address 20 to address 64 is taken. The branch decision is not made until cycle 4, by which point the `and`, `or`, and `sub` instructions at addresses 24, 28, and 2C have already been fetched. These three instructions must be flushed, and the `slt` instruction is fetched from address 64 in cycle 5.

Earlier branch decision

We can reduce the branch misprediction penalty by making the branch decision earlier.

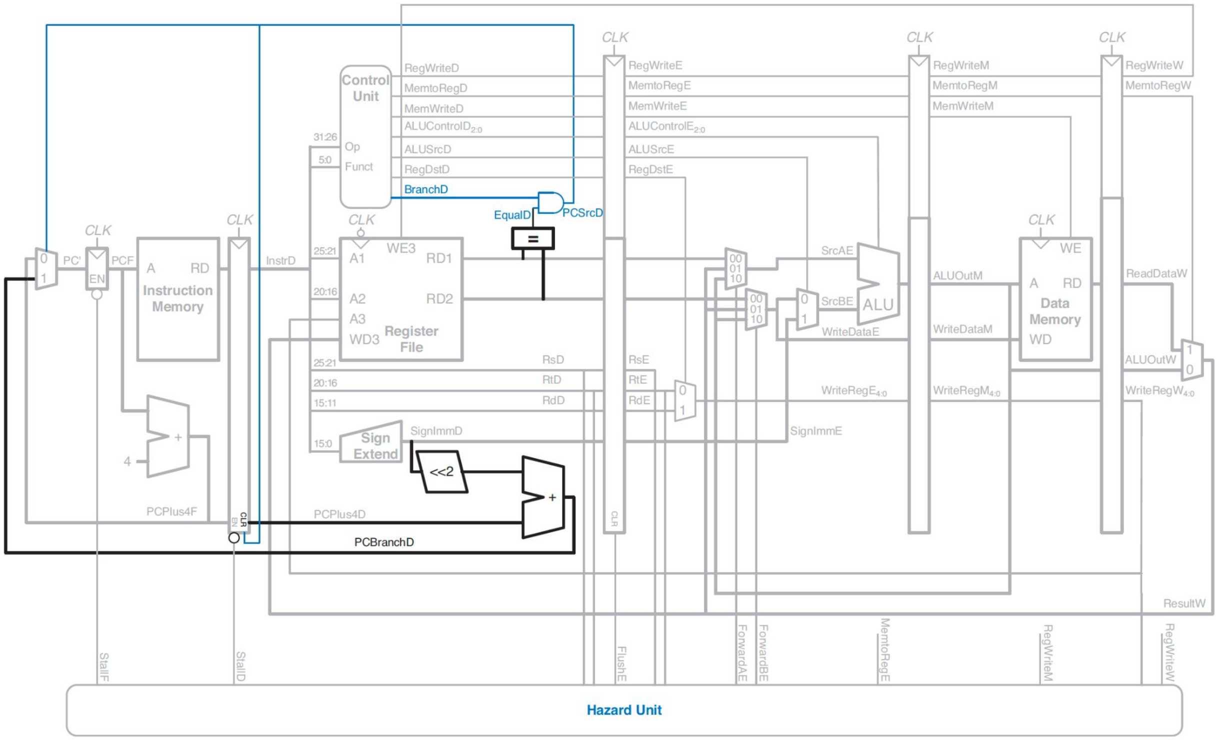


```
20    beq $t1, $t2, 40

24    and $t0, $s0, $s1

28    or  $t1, $s4, $s0

2C    sub $t2, $s0, $s5

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

Flush this instruction

- Making the decision requires comparing the values of two registers.
- By comparing the values in cycle 2, the branch misprediction penalty is reduced to only one instruction.

Datapath for handling branch control hazard



It uses a dedicated equality comparator, which is much faster than performing a subtraction and zero detection.

The PCBranch adder is moved into the Decode stage to compute the destination address in time.

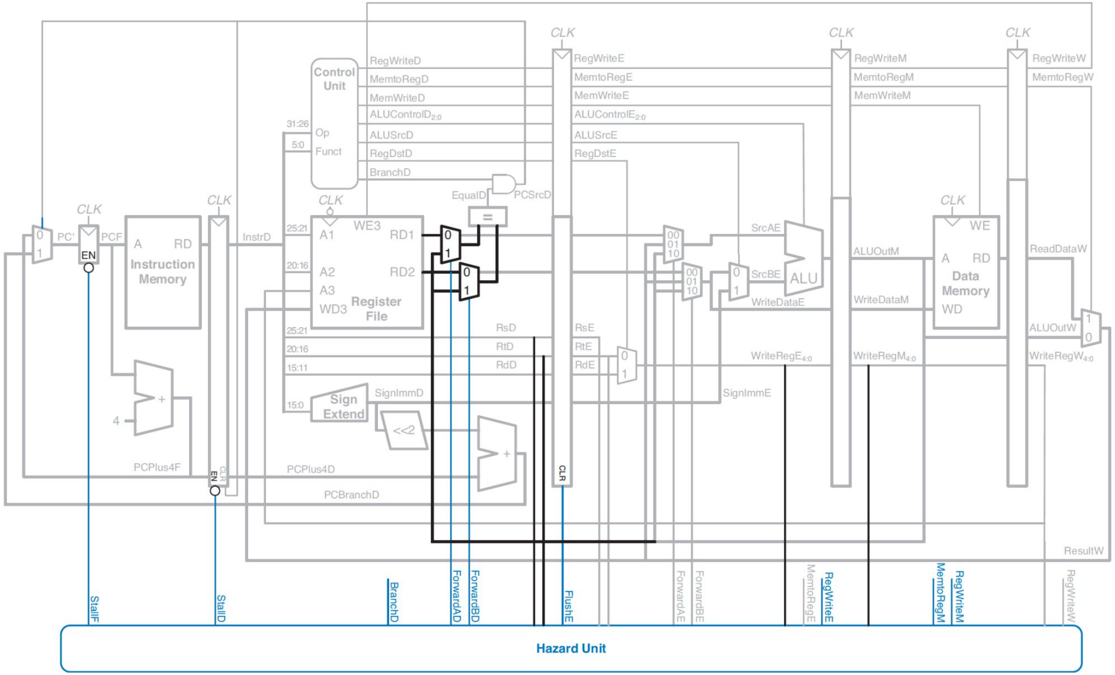
PCSrc is determined in the Decode stage. It is connected to the PC multiplexer and the CLR of the Decode stage pipeline register. So, the incorrectly fetched instruction can be flushed when a branch is taken.

Data dependencies in earlier branch decision

The earlier decision hardware introduces a new RAW hazard:

- If one of the source operands for the branch was computed by a previous instruction and has not yet been written into the register file, the branch will read the wrong operand value from the register file.
- We can solve this data hazard by forwarding the correct value if it is available or by stalling the pipeline until the data is ready.

Handling RAW for the earlier branch decision

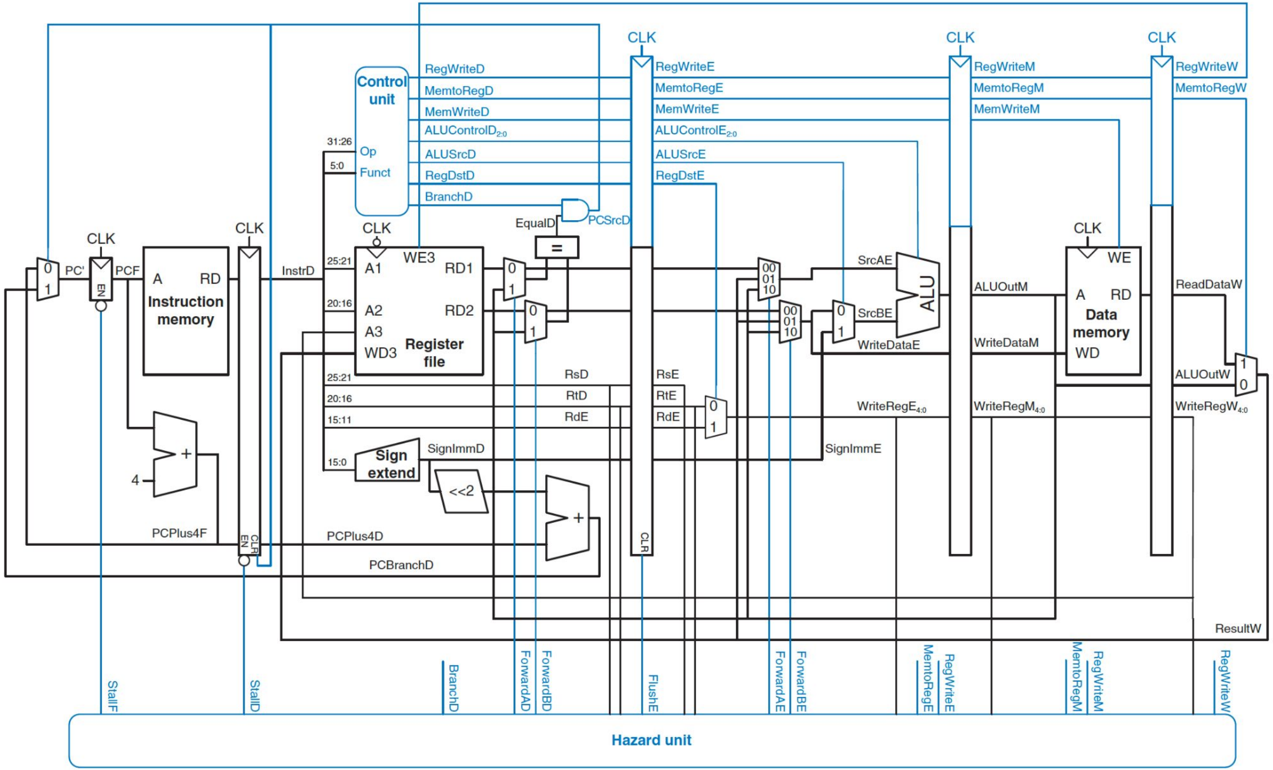


If a result is in the Writeback stage, it will be written in the first half of the cycle and read during the second half. So, no hazard exists.

If the result of an ALU instruction is in the Memory stage, it can be forward to the equality comparator through the two multiplexers.

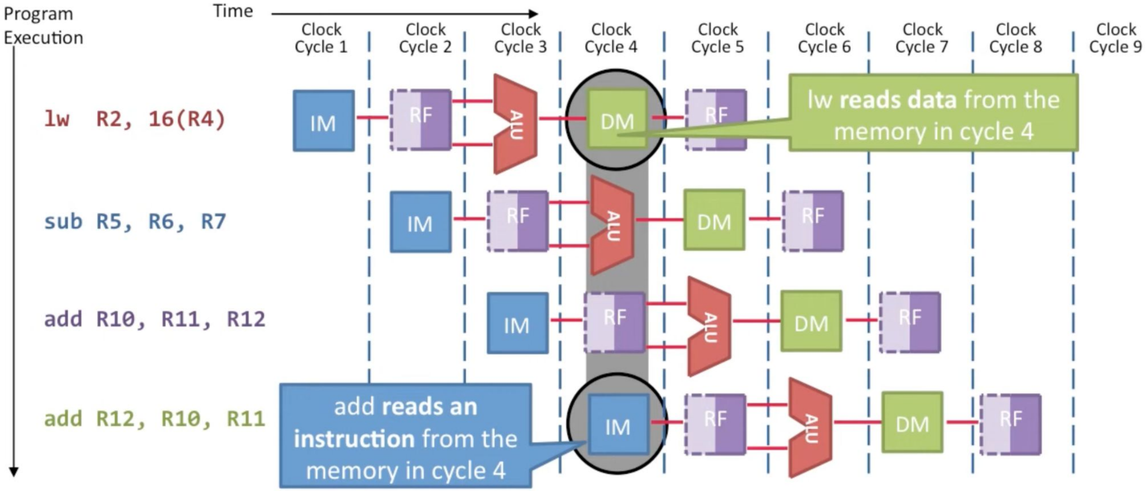
If the result of an ALU instruction is in the Execute stage or the result of a lw instruction is in the Memory stage, the pipeline must be stalled at the Decode stage until the result is ready.

Pipelined processor with full hazard handling



Structural hazard

Cause: attempt to use the same resource from different institutions in the same cycle.



lw **reads data** from the memory in cycle 4

add **reads an instruction** from the memory in cycle 4

- **Two instructions access memory in cycle 4**
  - We use **separate instruction** and **data memories** to support this
  - (Access instruction memory on every cycle!)

Example: when a system uses a single memory for instructions and data, it will result in structural hazards.

Resolution: adding additional hardware resources (e.g., using instruction and data memories /caches) or stalling the pipeline.

Minimizing the use of stalls

Stalls can solve all kinds of hazards in a pipelined processor. But they are the last-resort solution because they come with significant performance costs:

- **Reduced Performance**: Stalls waste pipeline cycles, decreasing the throughput and efficiency of the processor.
- **Hardware Underutilization**: Other stages of the pipeline might remain idle during a stall, wasting resources.
- **Scalability Issues**: In deeply pipelined or superscalar architectures, stalls cause larger delays due to the increased complexity and number of instructions in-flight.

Performance analysis

Ideally, the pipelined processor would have a CPI of 1, but in practice, the CPI is slightly higher than 1 because of a stall or a flush in the program being executed.

We can determine the cycle time by considering the critical path in each of the five pipeline stages:

$$T_c = max\left(\left. \begin{array}{c} t_{pcq}+t_{\text{mem}}+t_{\text{setup}} \\ 2(t_{RF\text{read}}+t_{\text{mux}}+t_{\text{eq}}+t_{\text{AND}}+T_{\text{mux}}+t_{\text{setup}}) \\ t_{pcq}+t_{\text{mux}}+t_{\text{mux}}+t_{\text{ALU}}+t_{\text{setup}} \\ t_{pcq}+t_{\text{memwrite}}+t_{\text{setup}} \\ 2(t_{pcq}+t_{\text{mux}}+t_{RF\text{write}}) \end{array} \right\} \begin{array}{c} \text{Fetch} \\ \text{Decode} \\ \text{Execute} \\ \text{Memory} \\ \text{Writeback} \end{array} \right)$$

Pipelined processor CPI (H&H Example 7.9)

The SPECINT2000 benchmark considered in Example 7.7 (in H&H) consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions.

Assume that 40% of the loads are immediately followed by an instruction that uses the result, requiring a stall, and that one quarter of the branches are mispredicted, requiring a flush. Assume that jumps always flush the subsequent instruction. Ignore other hazards. Compute the average CPI of the pipelined processor.

Pipelined processor CPI (H&H Example 7.9)

The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of time that instruction is used.

- **Loads** take one clock cycle when there is no dependency and two cycles when the processor must stall for a dependency. So, they have a CPI of 0.6 x 1 + 0.4 x 2 = 1.4
- **Branches** take one clock cycle when they are predicted correctly and two when they are not. So, the CPI is 0.75 x 1 + 0.25 x 2 = 1.25.
- **Jumps** always have a CPI of 2.
- **All other instructions** have a CPI of 1 because a new instruction is issued every cycle.

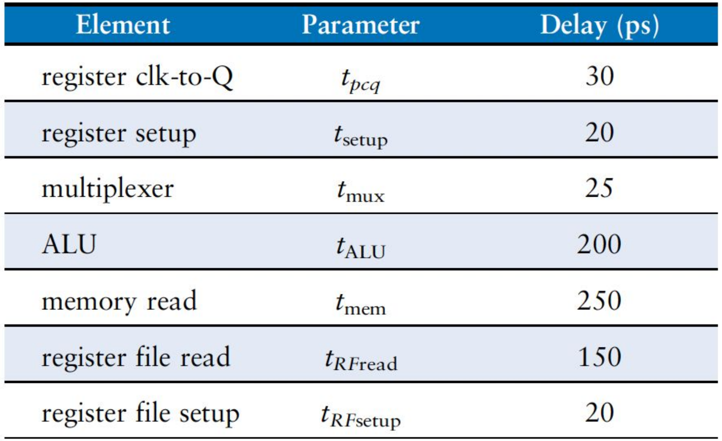
Therefore, the Average CPI = 0.25 x 1.4 + 0.1x1 + 0.11x1.25 + 0.02x2 + 0.52x1 = 1.15

Processor performance comparison (H&H Example 7.10)

Ben needs to compare the pipelined processor performance to that of the single-cycle and multicycle processors considered in Example 7.8 (in H&H). Most of the logic delays were given in Table 7.6.

The other element delays are 40 ps for an equality comparator, 15 ps for an AND gate, 100 ps for a register file write, and 220 ps for a memory write. Help Ben compare the execution time of 100 billion instructions from the SPECINT2000 benchmark for each processor.

Table 7.6 Delays of circuit elements



Element	Parameter	Delay (ps)
register clk-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	$t_{RF\text{read}}$	150
register file setup	$t_{RF\text{setup}}$	20

Processor performance comparison (H&H Example 7.10)

The cycle time of the pipelined processor is

$$
\begin{aligned}
T_{C3} = max\{\ & 30 + 250 + 20, \\
& 2(150 + 25 + 40 + 15 + 25 + 20), \\
& 30 + 25 + 25 + 200 + 20, \\
& 30 + 220 + 20, \\
& 2(30 + 25 + 100) \\
\} &= 550\ ps
\end{aligned}
$$

The execution time is

$T_3 = (100 \text{ x } 10^9 \text{ instructions})(1.15 \text{ cycles/instruction})(550 \text{ x } 10^{-12} \text{ s/cycle}) =$ *63.3 seconds.*

Unbalance problem

Durations of each step of different stage cuts (balanced and unbalanced)								
Fetch	Pipeline register	Decode	Pipeline register	Execute	Pipeline register	Memory	Pipeline register	Writeback
200 ps	50 ps	200 ps	50 ps	200 ps	50 ps	200 ps	50 ps	200 ps
150 ps	50 ps	150 ps	50 ps	150 ps	50 ps	400 ps	50 ps	150 ps

- Single-cycle processor: 1000 ps/instruction
- If all stages are balanced (i.e., all take the same time), 250 ps/instruction; 4 times faster than a single-cycle processor
- If stages are unbalanced, 450 ps/instruction; 2.22 times faster

Balanced stages has higher performance.

Deep pipelines

We may speed up the pipelined processor by increasing the number of stages. By splitting the long stages into smaller sub-stages,

- Less logic in each stage, so less duration (smaller CPI)
- The durations can be much equal

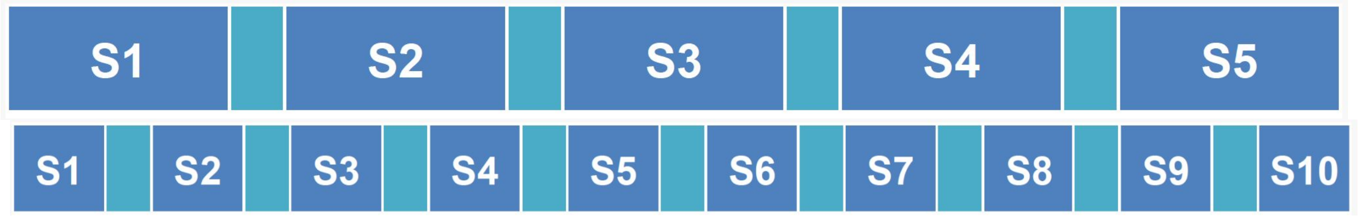
It is common to use 10 to 20 stages in modern CPUs.

Is deeper better?

The answer is no, because the more stages, the more delays on pipeline registers.

5-stage pipeline

10-stage pipeline



5-stage pipeline:

- Cycle time: 200 ps + 50 ps = 250 ps
- Latency: 250 ps x 5 = 1250 ps
- Delay of pipeline register/cycle time = 50 ps / 250 ps = 20%

10-stage pipeline:

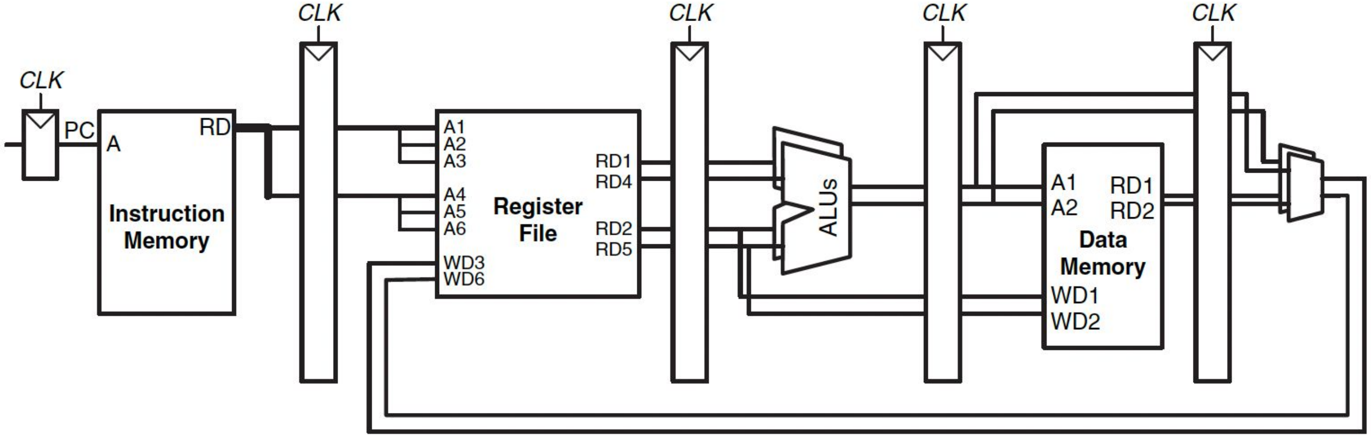
- Cycle time: 100 ps + 50 ps = 150 ps
- Latency: 150 ps x 10 = 1500 ps
- Delay of pipeline register/cycle time = 50 ps / 150 ps = 33%

The changes of depth in pipelined processors

1960s -1980s	Pipeline depth increased modestly (2~5 stages)	MIPS R2000 (5 stages)
1990s	Depth increased significantly with RISC and superscalar designs (7~10 stages)	Intel Pentium Pro (10 stages)
2000s	Pushed to extremes (~30 stages) for high clock speeds but faced diminishing returns.	Intel Pentium 4 Prescott (31 stages!!!), very deep pipelines. Severe penalties for branch mispredictions and pipeline flushing. Power consumption and heat dissipation increased significantly.
2010s -2020s	Moderate depths (~14-20 stages) with a focus on balanced designs, energy efficiency, and specialization.	Intel Core series, AMD Ryzen, Apple Silicon series.

Superscalar processor

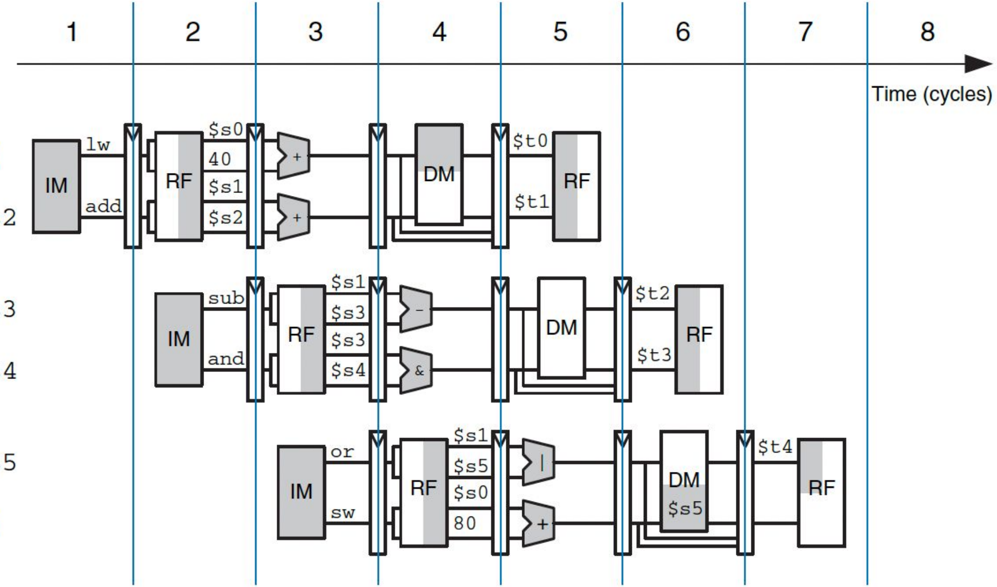
A superscalar processor contains multiple copies of the datapath hardware to execute multiple instructions simultaneously.



Superscalar pipeline

Superscalar processors exploit both temporal and spatial parallelism to squeeze out performance far exceeding that of the single-cycle and multicycle processors.

```
lw   $t0, 40($s0)
add  $t1, $s1, $s2

sub  $t2, $s1, $s3
and  $t3, $s3, $s4

or   $t4, $s1, $s5
sw   $s5, 80($s0)
```


1
2
3
4
5
6
7
8
Time (cycles)
IM
lw
add
RF
$s0
40
$s1
$s2
+
+
DM
$t0
$t1
RF
IM
sub
and
RF
$s1
$s3
$s3
$s4
-
&
DM
$t2
$t3
RF
IM
or
sw
RF
$s1
$s5
$s0
80
|
+
DM
$s5
$t4
RF