Imperial College London

Lecture 8

Pipelined Processor

Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

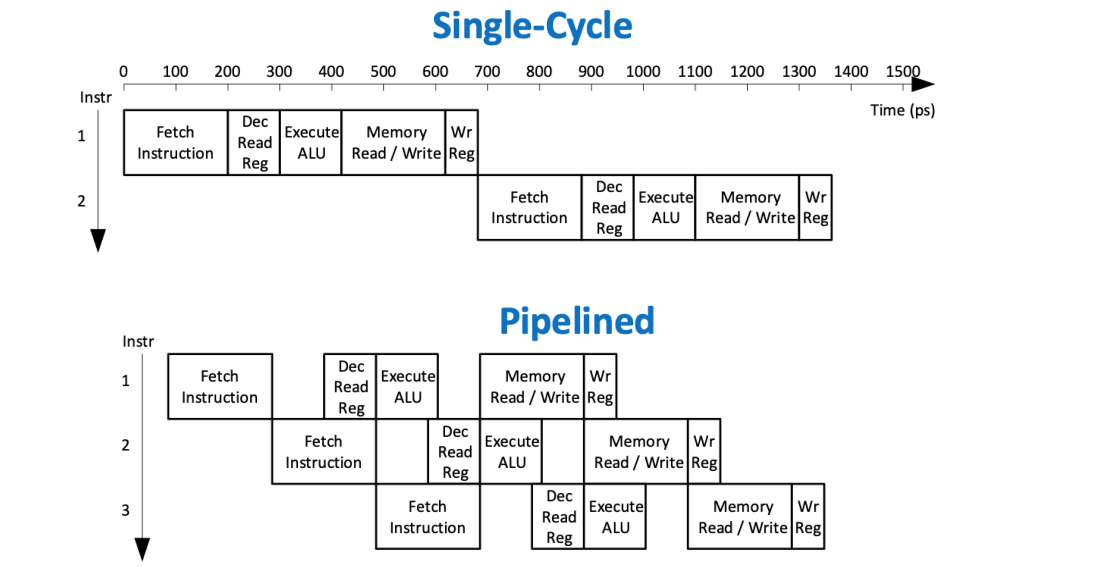
In this lecture, we consider how to improve the performance of a processor using a technique known as pipelining.

The idea here is to exploit temporal parallelism. Executing an instruction require various steps. In the single-cycle processor, these are performed one step after another. Hence the total time taken for the processor to complete one instruction cycle is the sum of the time taken by each of these steps or stages.

The idea of pipelining is to divide the single-cycle instruction cycles into 5 separate stages. Then add a register between each stage so that different stages can happen in parallel. In a 5 stage pipeline, the process can be executing 5 instructions simultaneously. Each instruction is in different stage of completion.

This is rather like making a car in a production line. Instead of waiting for a car to be completely built from start to finish, the manufacturing process is divided into many stages. In this way, the factory is building many cars in a single production line. The register is like the storage area where each stage puts its output for the next stage to add their contributions.

Single-cyle vs Pipelined Processor



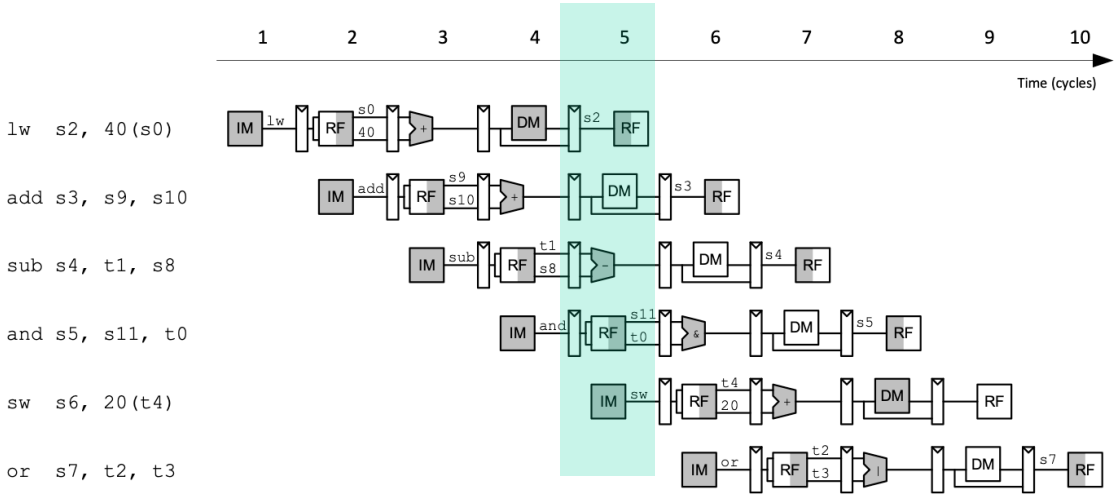
Based on: *"Digital Design and Computer Architecture (RISC-V Edition)"* by Sarah Harris and David Harris (H&H),

Consider how a single-cycle processor executes an instruction. The steps are dividied into five consecutive stages. The horizontal axis is time in ps, and is roughly correct for a typical processor in 2020 technology.

Fetching instruction from memory takes around 200ps. This is mostly taken by the memory access delay from address to data. The register address fields of the instructions AD1, AD2 are presented to the Register File. Register File is small and therefore has fast access time. Register operands takes, say, 100ps to be available. The ALU then takes another 120ps to perform the ALU operation. The ALU result could be used to access data memory (e.g. as address pointer), which takes another 200ps. Finally the memory data could be written back to the Register File, which takes around 60ps. The total time taken to complete this five stage execution of the instruction could be around 680ps.

In the case of a pipeline processor, each stage is now a pipeline stage taking one clock cycle. Multiple instructions are executed at the same time, but progressing at different stages as shown in the diagram. The cycle time is now the longer time for any one of the stage to complete. In this case, it is 200ps. Therefore in theory, we could be executing instructions over 3 times faster. The latency is longer, i.e. the time it takes for an instruction to complete is 5 x 200ps = 1ns. However, 5 instructions are completed within that time overall. Therefore pipeline processors are always faster than single-cycle processors.

Pipelined Processor Abstraction

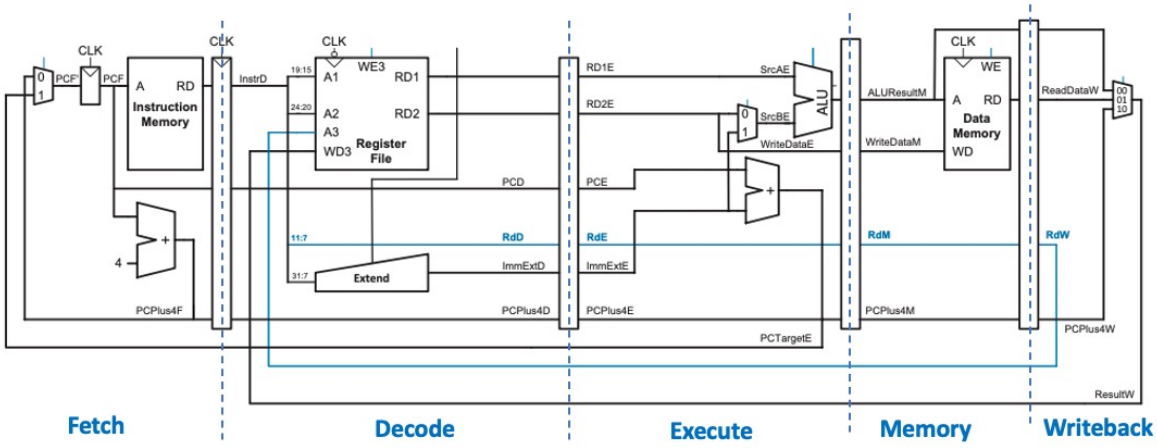


Based on: *"Digital Design and Computer Architecture (RISC-V Edition)"* by Sarah Harris and David Harris (H&H),

Here is an abstract diagram showing what happens when a 5-stage pipeline processing executing a series of instructions as shown. A pipeline register is inserted between stages.

Consider what happens during cycle 5. The first instruction (lw) is finishing by writing to s2 of the Register File. The 2nd instruction (add) is not really doing much but is passing the ALU data to the next pipeline register bypassing data memory. The 3rd instruction (sub) is performing the ALU subtraction operation. The 4th instruction (and) is fetching the operand from the Register File. Finally, the 5th instruction (sw) is being fetch from the instruction memory.

Adding Pipeline to Single-Cycle Processor



- Signals in Pipelined Processor are appended with first letter of stage (i.e., PC**F**, PC**D**, PC**E**).
- Register file written on **falling edge** of *CLK*

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

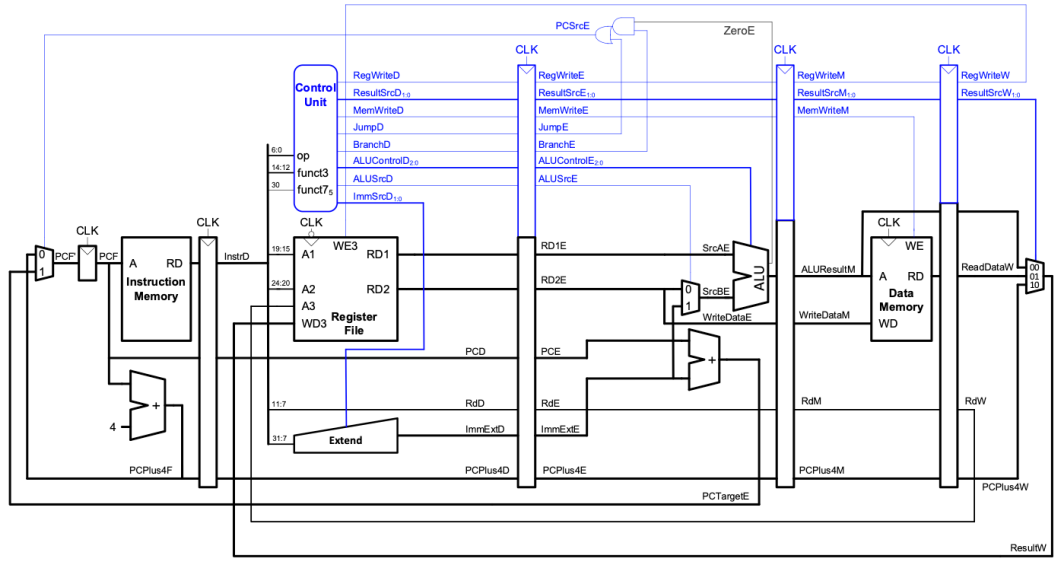
Adding pipelining to the single-cycle microarchitecture is easy. We simply insert registers between each of the five stages. The five stages are given the names: Fetch, Decode, Execute, Memory and Writeback.

Introducing pipeline registers also create signals with the same purpose but for different instructions. For example, the Program Counter value PC is moving from stage to stage on each clock cycle. The PC value at the Execute stage is for a different instruction to the PC value at the Fetch stage.

To distinguish the PC signal at different stage of the pipeline, we append the letter F, D, E, M, and W to indicate which stage the signal has reached.

Note that the writeback to Register File now happens on the FALL EDGE of the clock instead of the rising edge. In this way, data can be written in the first half cycle and read back in the second half of the cycle for use in a subsequent instruction.

Adding Pipeline to Single-Cycle Processor



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

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

The Control Unit for a pipelined processor is the same as that for the single-cycle processor excepted that all the control signals MUST also be pipelined so that they arrive in synchrony to the datapath. In other words, the control signals must travel with the data so that each stage is being controlled by the correct signal.

Pipeline Hazards

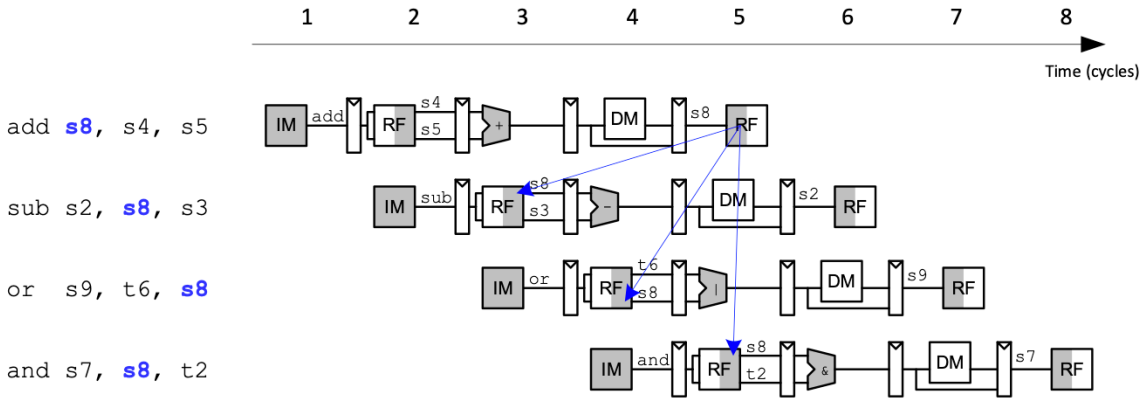
- When an instruction depends on result from instruction that hasn't completed
- Two types of hazards:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branch)

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

Running several instruction in a pipelined manner has potential of wrong data being used. A later instruction could depend on result from an earlier instruction that is yet to finish.

Such program introduced by pipelining is known as "Hazards". There are two types of hazard: Data and Control.

Data Hazard



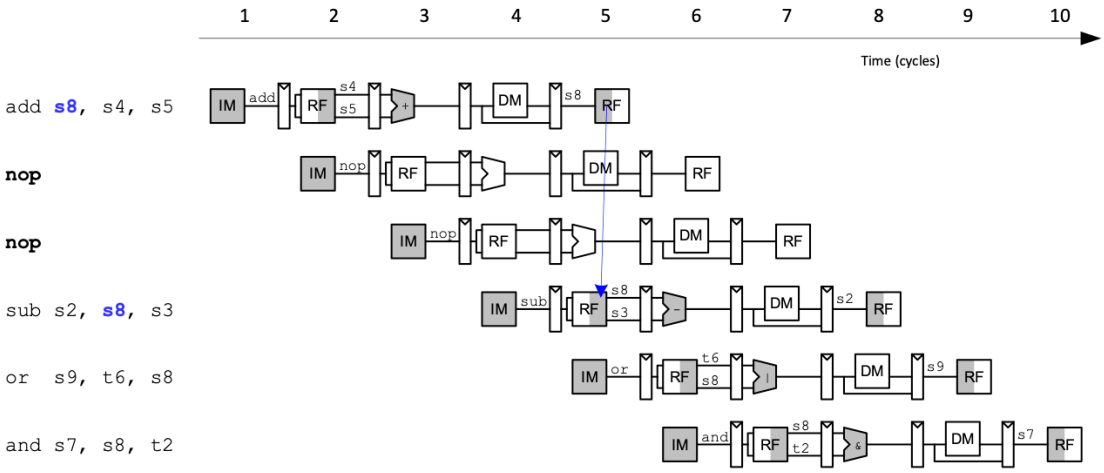
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

In the code sequence shown here, instruction 1 (add) does not produce the correct result in s8 until cycle 5. Yet, instruction 2 (sub) uses the s8 result in cycle 3, instruction 3 (or) uses in cycle 4 and so on. The blue arrow show when s8 receives its correct results as compared to when it is needed by subsequent instructions.

This is called a Read After Write (**RAW**) hazard. The sub instruction tries to **read** s8 **after** the add instruction is suppose to have **written** it in the instruction code sequence. However, pipelining results in the wrong data being read.

Avoid Data Hazard in code

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



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

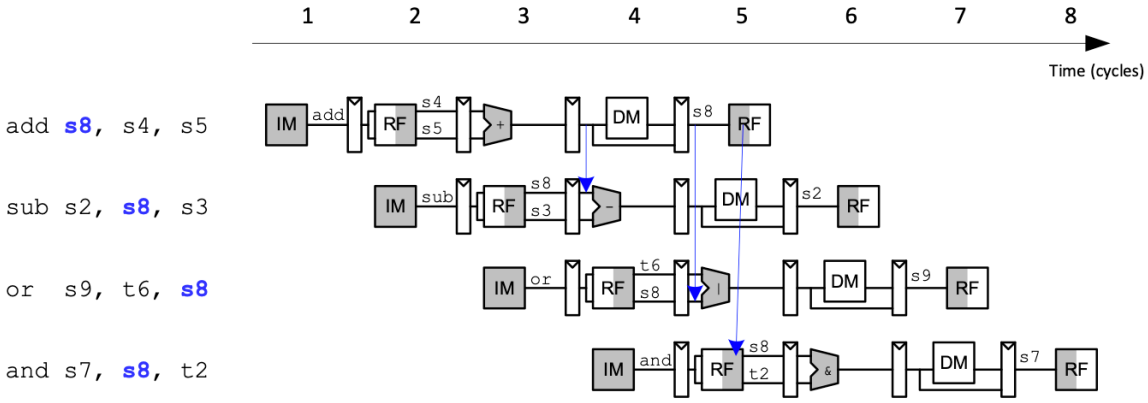
A way to solve the RAW hazard problem is to avoid using the result until it is ready to be used. This can be achieved by inserting NOP instructions (which is a pseduoinstruction in RISC-V: addi zero, zero, 0).

In this example, inserting two NOPs after the add instruction means that by cycle 5, s8 is updated on the first half of the cycle, and the sub instruction now reads the correct value of s8 in the second half of the cycle. Thereafter, s8 is also correct for the remaining instructions.

Inserting NOP instructions is a waste of cycles. So another method used in compilers is to swap other instructions after "add" which do not rely on s8 results.

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



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

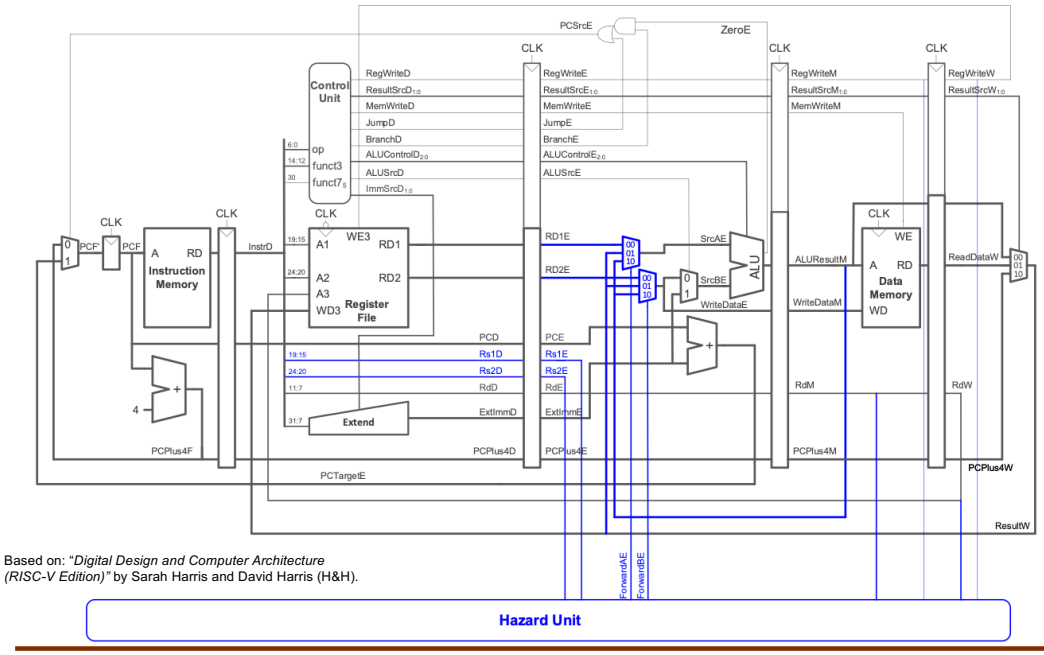
Another method to overcome the data hazard problem is to add hardware to "forward" required data internally from earlier stages of the pipeline.

In this example s8 is written back to Register File only in cycle 5. However, the value is available in cycle 4 after the ALU stage. Therefore it is possible to bypass the last pipeline register, and present the ALU result directly as the input operand for the sub instruction as shown in the diagram.

Similarly, or instruction can take the s8 results from the last pipeline register before writeback happens.

As for and instruction, no bypassing is required because by the time "and" needs s8, the correct data is already stored in the Register File.

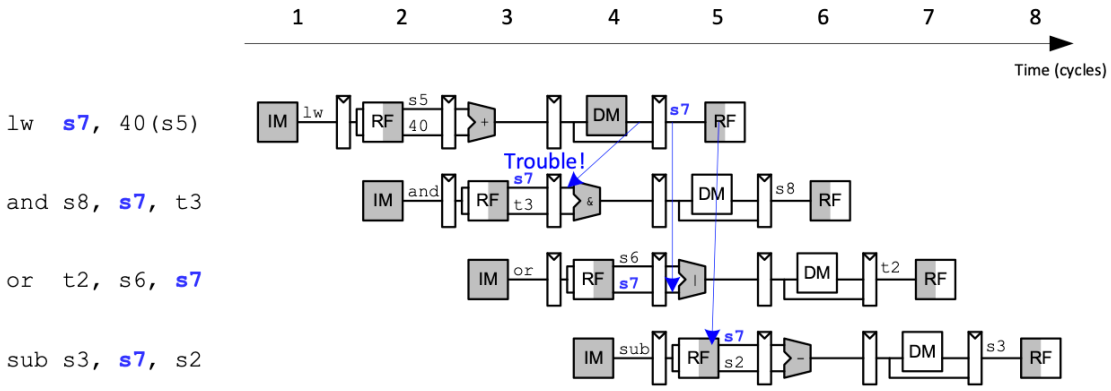
Adding Hazard Unit



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

To manage all these bypassing, one needs to add multiplexers a show in the diagram above, and a Hazard Unit that detects data depency from the instructions and determine when to and not to bypass the pipeline stages.

Data Hazard due to lw Data Dependency

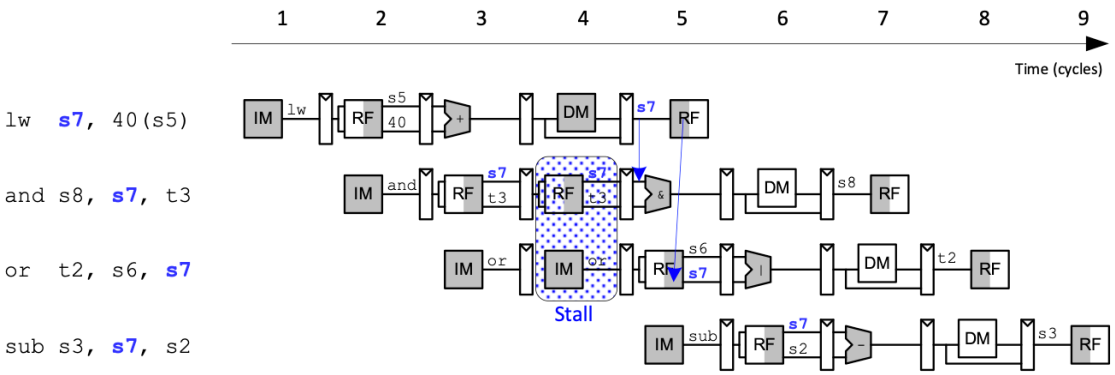


Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

Forwarding (or bypassing) only works if the data required is already in the pipeline. Unfortunately for the lw instruction in the example above, the correct value of s7 is not anywhere until cycle 5. Therefore it is NOT possible to perform forwarding to the 2nd instruction (AND).

.

Stalling to solve lw Data Dependency

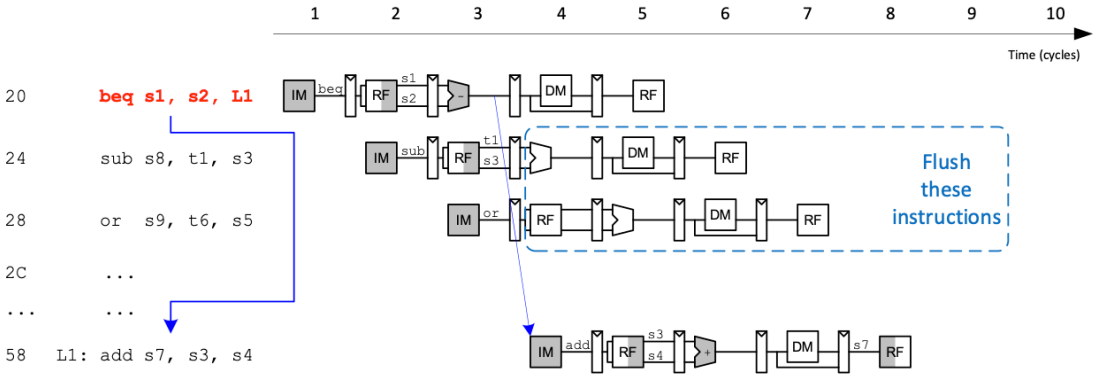


Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

This problem can be solved by pausing or stalling the execution of AND and OR instruction as shown above. In cycle 4, the Decode stage for the AND instruction is stalled for an extra cycle, and so is the Fetch stage of the OR instruction. Now s7 can be forwarded for the AND instruction and the OR instruction.

Control Hazard

- **beq, bne:**
  - 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



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

Control Hazard happens when we execute branch instructions. In the example shown here, the BEQ instruction can change the control flow of the program (to another address L1).

By the time the branch instruction is taken, two instructions following BEQ is already in various stages of execution in the pipeline. The pipeline register therefore contains results for SUB and OR instructions, which should not be executed. This situation is known as Control Hazard. To overcome this hazard, we must **FLUSH** or discard the data stored in the pipeline.

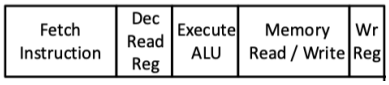
Single-Cycle Processor Performance

Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

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



Program with 100 billion instructions:

Exec Time = # instructions x CPI x T_C

= $(100 \times 10^9)(1)(750 \times 10^{-12}\,s)$

= **75 seconds**

750ps

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

$$T_{c\_single} = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}}$$

The performance of a processor is determined by the time it takes to execute N instructions where N can be millions or billions.

The total elapsed time is shown by the simple equation above. CPI is the number of clock cycles per instruction. In the case of single-cycle process, this is 1 by definition. Tc is the cycle time of the clock, which is the time it takes for one instruction to be completed.

The table above shows the typical delay incurred by different stages of the processor. The ones marked in red dot are the delays that are most significant and cannot be absorbed in anyway. For example, memory read/write operation and ALU operation tends to dominate the delay time.

Therefore, the estimate cycle time for single-cycle processor is the sum of these significant delay. In our example here, this amount to 750ps.

So executing 100 billion instructions using this single-cycle processor takes 75 seconds.

Pipelined Processor Performance

Pipelined processor critical path:

$T_{c\_pipelined}$ = max of

$t_{pcq} + t_{mem} + t_{setup}$	Fetch	
$2(t_{RFread} + t_{setup})$	Decode	
$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND\text{-}OR} + t_{setup}$	Execute	350ps
$t_{pcq} + t_{mem} + t_{setup}$	Memory	
$2(t_{pcq} + t_{mux} + t_{RFwrite})$	Writeback	

Program with 100 billion instructions

Execution Time = (# instructions) × CPI × T_c

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

= **43 seconds**

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H).

For the pipelined processor, the cycle time Tc is determined by the delay of the longest (slowest) pipeline stage. In our case, it the execute stage, which takes 350ps.

Also, to overcome hazards caused by pipelining, we often need to stall the processor. Therefore the Cycle per Instruction (CPI) value for pipeline processor is always larger than 1 (i.e. takes more than one cycle to complete an instruction). In this case, we assume CPI = 1.23.

The total time taken to complete 100 billion instructions is 43 seconds. This is therefore 1.7X faster than the single-cycle processor.