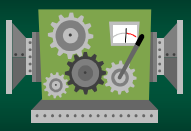
Part 10

Hardware



Instruction Execution

The Steps Involved

Instruction Execution

- When an instruction executes on a processor, a number of different tasks take place
- Typically, these tasks can be broken into 5 different steps

Five Steps in Execution

1. Fetch an instruction from memory
2. Decode it to determine what it is
3. Read the inputs from registers / memory
4. Execute for computations for instruction
5. Write the result into the registers / memory

1. Fetch the Instruction

- First, the processor fetches the instruction from the memory
- The result is stored in the Instruction Register
- Formally known as *Instruction Fetch (IF)*

2. Decode the Instruction

- Second, the instruction is *decoded* to determine what it is and its operands
- Signals are sent to the execution unit as input
- Formally known as *Instruction Decode (ID)*

3. Read Inputs

- The Execution Unit then reads the values of the instruction
- These can be in located in the instruction itself (immediate), register file, and from memory
- Formally known as *Memory Access (Mem)*

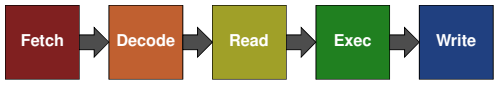
4. Execute the Instruction

- Decoded instruction and input values are passed to the ALU
- Depending on the complexity of the instruction, some computations require multiple clock cycles
- e.g. multiplication is requires more cycles and an add
- Formally: *Execute (EX)*

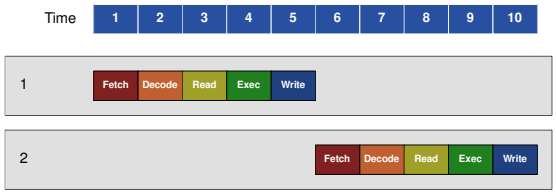
5. Write Result

- Finally, the result is written back into the register / memory
- Processor also updates flags and other state information such as the Program Counter
- Formally known as *Write Back (WB)*

Instruction Execution



Execution of 2 Instructions

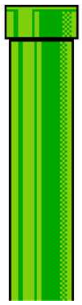


Pipelining

Do multiple things at once

Pipelining

- *Pipelining* is an technique where multiple instructions are executed at the same time
- This greatly improves the speed (and efficiency) of a system



Pipelining

- It is invisible to the programmer… implemented by the hardware
- Pipelining is different from multi-core processors!
- Pipeline happens on one core

The Laundry Metaphor

- To understand the concept, typically a "laundry metaphor" is used
- Based on how we all do our laundry:
  - put the clothes in the washing machine
  - put the wet clothes in the dryer
  - fold the clothes and put them away

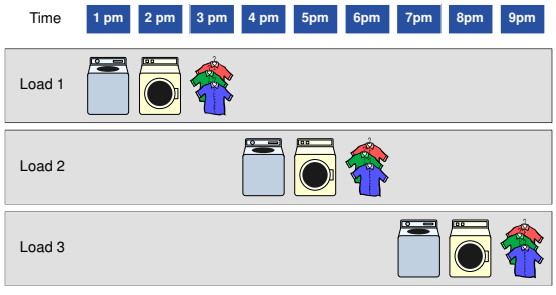
The Laundry Metaphor

Step 1. Put the clothes in the washer

Step 2. Put the wet clothes in the dryer

Step 3. Fold the clothes and put them away

Let's Do Our Laundry



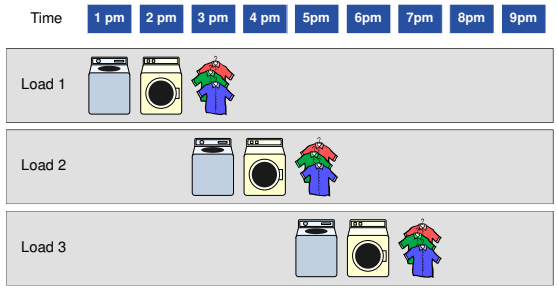
Looking At Our Task

- It took us until 10pm to finish our laundry!
- But…. some of our equipment was idle
  - when we were washing laundry, the dryer and folding counter was idle
  - when the dryer was being used, the washer and folding counter was idle
  - when the folding counter was being used, the washer and dryer were idle

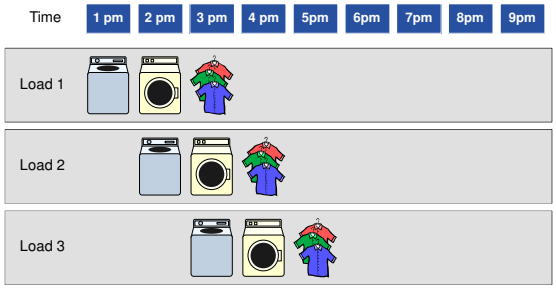
The Pipelined Approach

- Let's overlap these loads of laundry
- We don't have to wait until the first load is done before were start the next one
- Better approach
  - put the Load 1 in the washing machine
  - after Load 1 is washed, we place it in the dryer
  - washer is now available… start load 2

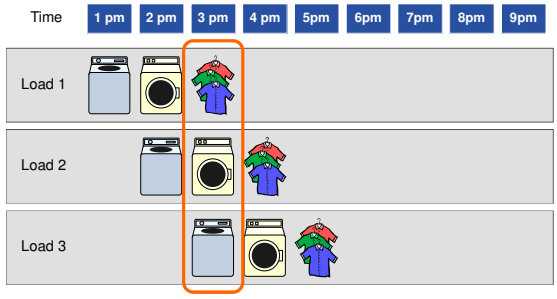
Pipelined: Fold & Wash Together



Fully Pipelined



All Three Components in Use



So, is it faster?

- Note: the actual time required to finish a load of laundry has *not* changed
- It still requires the same 3 steps and each takes one hour
- The speedup occurs because the different loads are executed in *parallel*

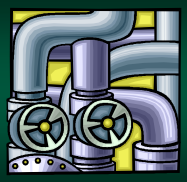
Analyzing Speedup

- So, how have we improved efficiency?
- In pipelining, the speedup is directly proportional to the number of steps (that we can do in parallel)

$$\text{Speedup}_{\text{pipelined}} = \text{Number of Steps}$$

Start-up and Wind-down

- At the beginning of the work load, the pipeline wasn't completely full
- So, the pipeline has to fill before we get our full speedup
- As the number of Loads increases…
  - pipeline is full for a larger fraction of the time
  - so the start-up and wind-down (finishing the last load) becomes meaningless

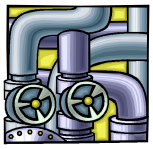


Pipelined Instructions

Modern Processors

Pipelined Instructions

- Just like the laundry metaphor, processors have different components that can used at the same time
- On modern processors, practically all the hardware (all those transistors) are in continuous use

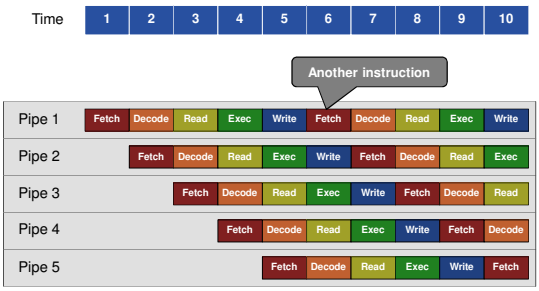


Pipelined Instructions

- Ideally, *nothing is ever idle!*
- Different stages of execution are pipelined
  - fetch
  - decode
  - read
  - execution
  - write back

Unpipelined Instructions

5 Pipelines

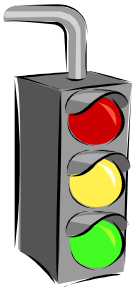


Problems Arise

- The problem arises that different stages take different amounts of time
- This is not being measured in clock cycles – but in nanoseconds
- So, pipelining requires a mechanism to keep everything in sync

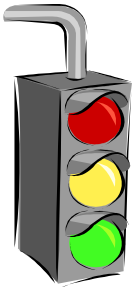
Latches

- To implement pipelining, a execution is divided into stages
- These often exist, physically, on different sections of the processor



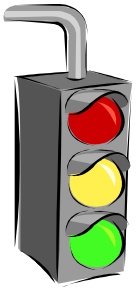
Latches

- Designer places *latches* (aka *buffers*) between each section
- Instruction will not advance onto the next stage until the latch is lifted



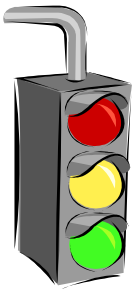
How Latches Work

- The latches read their inputs at the start of each cycle
- Store the data to be passed from them - remain constant throughout the rest of the cycle

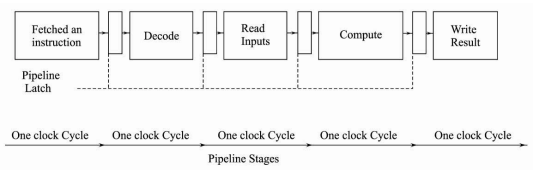


How Latches Work

- When a stage is ready…
  - it signals the prior stage
  - prior state sends the data
  - and becomes available



Latched Instructions



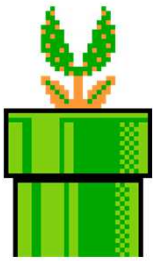
Latched Instructions

Pipeline Read / Write Hazards

Call the plumber!

Pipeline Issues

- When you execute instructions in parallel, there are inherit dangers that must be handled
- Often, one instruction can interfere with another and cause strange side effects

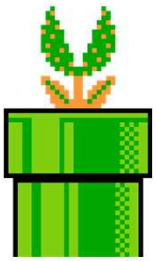


Pipeline Issues

- There are 3 read/write dangers in pipelining
  - Read-After-Write (RAW)
  - Write-After-Write (WAW)
  - Write-After-Read (WAR)
- Each involves the writing of data to memory and / or registers
- These are known as *data dependencies*

The Result of the Hazards

- These hazards will always results in the program malfunctioning – *even though it might be written correctly*
- The hardware fails… not the software



Read-After-Write (RAW)

- A *Read-After-Write Hazard* is caused when one instruction reads data before the previous one has written it
- Instruction ***B*** reads an operand before Instruction ***A*** writes it

Read-After-Write (RAW)

- The result is that the instruction ***B*** has read old data – before ***A*** changed it
- It is though instruction ***A*** doesn't exist!

Read-After-Write (RAW) Example

```
MOV $12, %rax
ADD $5, %rax
SUB %rax, value    #Read output of add
```

If Sub reads rax before Add writes, it will subtract 12 rather than 17

Write-After-Read (WAR)

- A *Write-After-Read Hazard* is caused when one instruction writes data before the previous one has read it
- Instruction ***B*** writes an operand before Instruction ***A*** reads from it

Write-After-Read (WAR)

- The first instruction gets the wrong operand
- It should contain the value before Instruction ***A*** executes
- Instead, contains the value after instruction ***B*** executes

Write-After-Read (WAR) Example

```
MOV $6, %rax
SUB %rax, value
ADD $1, %rax
```

If Add writes rax before Sub reads, it will subtract 7 rather than 6

Write-After-Write (WAW)

- A *Write-After-Write Hazard* is caused two instructions attempt to write data, and the later one writes it first
- Instruction ***A*** and ***B*** attempt to write the same operand, but ***A*** writes it last

Write-After-Write

- The operand contains the wrong result
- Should contain the value of instruction ***B*** but it contains the result of Instruction ***A***
- It is though ***B*** never executed!

Write-After-Write (WAW)Example

```
MOV $42, %rax
MOV $31, %rax
```

If first MOV writes after the second MOV, rax will contain 42 rather than 31

Score Boarding

Solution to Data Dependency

Score Boarding

- *Score boarding* is a solution for register read stage structure hazards
- Processors track:
  - which registers will be written to by instructions in the pipeline
  - subsequent instructions determine whether their input registers are available

Score Boarding Bits

- In score boarding, a *"presence"* bit (aka p-bit), is added to each register
- The presence bit records:
  - if the register *full* – i.e. it is available for reading
  - or if the register is *empty* – i.e. waiting for an instruction to write its output

P-Bit Score Boarding

P	Register Bits

P	Register Bits

- P = 1, when data available for read
- After an instruction is going to write a register, P is set to 0 until it completes write-back

Operation: Input is Full (Ready)

- When an instruction enters the Read stage, the hardware checks to see if all of its input registers are *full (i.e. ready)*
- If so, the hardware reads the values of all the input registers

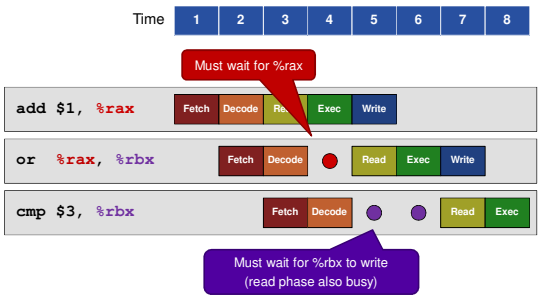
Operation: Input is Full (Ready)

- The hardware then marks all the output registers as *empty (busy)* – so no other instruction can read it until they are written
- The instruction then proceeds to the execute stage on the next cycle

Operation: Input is Empty

- If, in the Read Stage, the registers are *empty (busy)*, the instruction must wait
- The hardware holds the instruction, in the register read stage, until its input values become full
- In the meantime it *inserts bubbles* into the execute stage – it basically *stalls* it

Bubbles



Bypass Forwarding

Heads Up, the Data is Being Thrown!

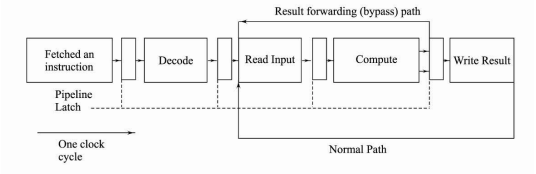
Bypass Forwarding

- Another approach to handling data hazards is to, in some cases, skip the register file
- The result is *forwarded* to the next register's read stage

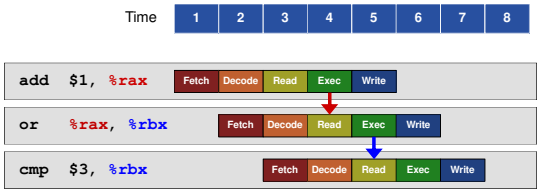
Bypass Forwarding

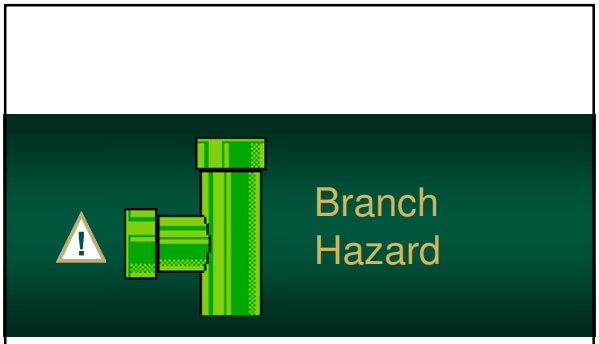
- Bypassing the write-back stage, we can eliminate data hazards
- Essentially,
  - we pass the ALU output from the first instruction directly to the second instruction
  - we skip the register file

Paths for Result Forwarding



Pipelined Instructions

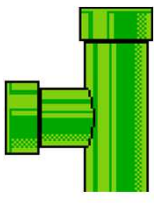




Jumps and Pipelines are not friends

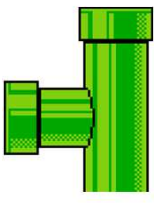
Branch Hazard

- So far, we covered register related hazards
- These are, unfortunately, not the only hazards that exist in pipelining
- Conditional branches cause problems

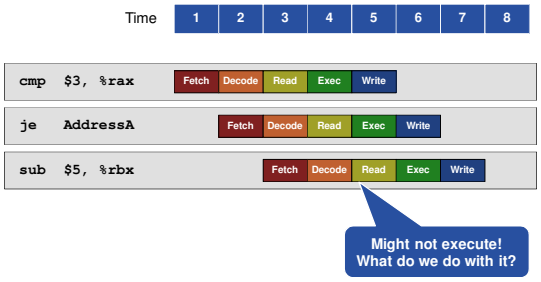


Branch Hazard

- For example, what happens if several instructions are in the pipeline
- One instruction contains a conditional jump
- Do we just execute the instructions after it? They may never run!



Branch Hazard



Conditional Branches

- Unconditional and conditional branches, create data dependencies
- These are between…
  - branch instruction and the instruction fetch stage of the pipeline
  - branch instruction computes the address of the next instruction - that the instruction fetch stage should fetch

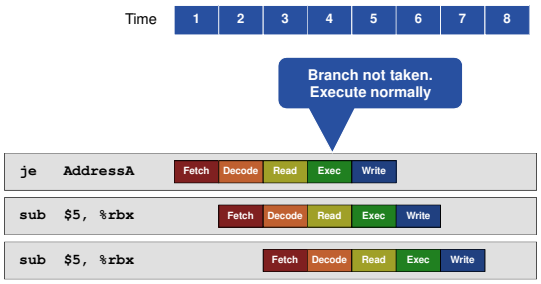
Flushing

- After the conditional branch instruction is analyzed, the hardware changes the program counter
- All the instructions in the pipeline are invalid (since they will not execute)
- These pipelines are flushed – data discarded and restarted

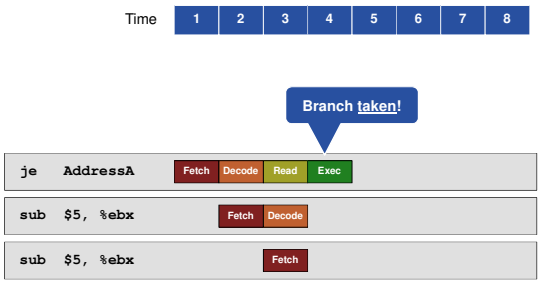
Flushing

- Naturally, this causes a huge delay for the new pipeline's startup latency
- So, *conditional branch penalty* has a huge effect on processor performance

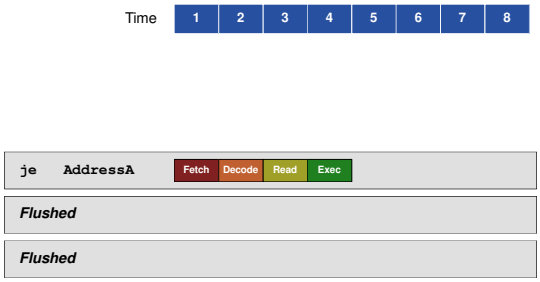
Conditional Jump Not Taken



Flush Required – 1 of 3



Flush Required – 2 of 3



Flush Required – 3 of 3