Lab 5: Make it Fast!

- Tasks
  - Turn your processor into a 5 stage pipeline.
  - Measure the performance improvement it provides.
- Three weeks
One Possible Pipeline

I did this in 30 seconds. No guarantee that it’s right. You will need to think through it yourself.
One Possible Pipeline

I did this in 30 seconds. No guarantee that it’s right. You will need to think through it yourself.
One Possible Pipeline

I did this in 30 seconds. No guarantee that it’s right. You will need to think through it yourself.
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)

```plaintext
add $s0, $t0, $t1

sub $t2, $s0, $t3

add $t3, $s0, $t4

add $t3, $t2, $t4
```
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
  - Register values
  - Also memory accesses (more on this later)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Data Dependences

A data dependence occurs whenever one instruction needs a value produced by another.

- Register values
- Also memory accesses (more on this later)

```assembly
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Data Dependences

• A data dependence occurs whenever one instruction needs a value produced by another.
  • Register values
  • Also memory accesses (more on this later)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Data Dependences

• A data dependence occurs whenever one instruction needs a value produced by another.
  • Register values
  • Also memory accesses (more on this later)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
add $t3, $t2, $t4
```
Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```
Solution: Stall

• When you need a value that is not ready, “stall”
  • Suspend the execution of the executing instruction
  • and those that follow.
• This introduces a pipeline “bubble.”
• A bubble is a lack of work to do, it propagates through the pipeline like nop instructions
Solution: Stall

- When you need a value that is not ready, “stall”
  - Suspend the execution of the executing instruction
  - and those that follow.
  - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions

Both of these instructions are stalled

NOPs in the bubble
Solution: Stall

- When you need a value that is not ready, “stall”
  - Suspend the execution of the executing instruction
  - and those that follow.
  - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions

Both of these instructions are stalled

One instruction or nop completes each cycle
Stalling the pipeline

• Freeze all pipeline stages before the stage where the hazard occurred.
  • Disable the PC update
  • Disable the pipeline registers
• This is equivalent to inserting into the pipeline when a hazard exists
  • Insert nop control bits at stalled stage (decode in our example)
  • How is this solution still potentially “better” than relying on the compiler?
Hardware for Stalling

- Turn off the enables on the earlier pipeline stages
  - The earlier stages will keep processing the same instruction over and over.
  - No new instructions get fetched.
- Insert control and data values corresponding to a nop into the “downstream” pipeline register.
  - This will create the bubble.
  - The nops will flow downstream, doing nothing.
- When the stall is over, re-enable the pipeline registers
  - The instructions in the “upstream” stages will start moving again.
  - New instructions will start entering the pipeline again.

injected_nop_decode_execute

\[\text{stall\_pc}\quad\text{stall\_fetch\_decode}\]

\[\text{Fetch} \rightarrow \text{Decode/ reg read} \rightarrow \text{EX} \rightarrow \text{Mem} \rightarrow \text{Write back}\]
One Possible Pipeline

Do not trust this schematic. Design your own.
One Possible Pipeline

Do not trust this schematic. Design your own.
One Possible Pipeline

Do not trust this schematic. Design your own.
Details

• Suggestions
  • Do one pipeline stage at a time. Start at fetch.
  • Test thoroughly.
    • Create the test case before you implement the functionality
    • Run all your tests repeatedly.
    • Find a way to automate your test.

• Things to watch out for
  • Pipelining control
  • You need to implement a branch delay slot.
  • You don’t need to implement a load delay slot.
  • You will need to stall for “data hazards.
    • You should create a “hazard detection module” to detect hazards and stall the pipeline accordingly.
One Testing Strategy

```assembly
or v0, zero, zero
// Test 1 (sets s0)
add v0, v0, s0
// Test 2 (sets s0)
add v0, v0, s0
// Test 3 (sets s0)
add v0, v0, s0
// Test 4 (sets s0)
add v0, v0, s0
// Test 5 (sets s0)
add v0, v0, s0
// Test 6 (sets s0)
add v0, v0, s0
// Check final value of v0.
```