Making some assumptions regarding the operation times for some of the basic hardware units in our datapath, we have the following timings:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction fetch</th>
<th>Register read</th>
<th>ALU operation</th>
<th>Data access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>100 ps</td>
<td>800 ps</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>700 ps</td>
<td></td>
</tr>
<tr>
<td>R-format</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>100 ps</td>
<td>600 ps</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>500 ps</td>
<td></td>
</tr>
</tbody>
</table>

How long would it take to execute the following sequence of instructions?

\[
\begin{align*}
lw & \quad \$1, \ 100(\$0) \\
lw & \quad \$2, \ 200(\$0) \\
lw & \quad \$3, \ 300(\$0)
\end{align*}
\]

But, maybe there’s a way we can cheat and complete the sequence faster.

What if we think of the datapath as a linear sequence of stages?

We have 5 stages, which will mean that on any given cycle up to 5 different instructions will be in various points of execution.

Can we operate the stages independently, using an earlier one to begin the next instruction before the previous one has completed?
Pipelining

We’ve only considered unimaginative execution; consider our longest instruction:

<table>
<thead>
<tr>
<th>Program execution order</th>
<th>Time (in instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, 100d(3)</td>
<td>200 ps</td>
</tr>
<tr>
<td>lw $2, 200d(60)</td>
<td>500 ps</td>
</tr>
<tr>
<td>lw $3, 300d(60)</td>
<td>800 ps</td>
</tr>
</tbody>
</table>

Ideal speedup is number of stages in the pipeline. Do we achieve this?

Improve performance by increasing instruction throughput:

<table>
<thead>
<tr>
<th>Program execution order</th>
<th>Time (in instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, 100d(3)</td>
<td>200 ps</td>
</tr>
<tr>
<td>lw $2, 200d(60)</td>
<td>500 ps</td>
</tr>
<tr>
<td>lw $3, 300d(60)</td>
<td>800 ps</td>
</tr>
</tbody>
</table>

The average time between initiating instructions has dropped to 200 ps.

Why do we have idle “gaps”?

Assume:
- register file write occurs in first half of a cycle
- register file read occurs in second half of a cycle

Total time here is 1400 ps versus 2400 ps for the original version...

...but consider how this would look if we were had 1,000,000 more lw instructions in our sequence...
Pipelining

What makes it easy
- all instructions are the same length
- just a few instruction formats
- memory operands appear only in loads and stores

What makes it hard?
- structural hazards: suppose we had only one memory
- control hazards: need to worry about branch instructions
- data hazards: an instruction depends on a previous instruction

We’ll build a simple pipeline and look at these issues

We’ll talk about modern processors and what really makes it hard:
- exception handling
- trying to improve performance with out-of-order execution, etc.
In some cases, the next instruction cannot execute in the following clock cycle. We will introduce some of the potential issues in the next few slides.

**structural hazard**
- hardware cannot support the necessary combination of operations at once
- reconsider the earlier example with a single memory unit and a fourth `lw` instruction

**data hazard**
- data that is necessary to execute the instruction is not yet available
- consider:

  ```assembly
  add $s0, $t0, $t1
  sub $t2, $s0, $s3
  ```

- *load-use* hazard occurs when data imported by a load instruction is not available when it is requested

Here, the second instruction needs the final result from the first instruction during the register fetch portion of the instruction decode phase:

Obviously the value will not be available in register `$s0` until the first instruction has completed.

However, the computed value IS actually available after the first instruction finishes its third stage, just in time to satisfy the need of the ALU when the second instruction reaches its third stage. This is indicated above by a *forwarding* link.

In principle, the hazard here could be detected and handled.

But, what if the "forwarding" link actually went backwards?
Data Hazard Example: Stalling

Here the first instruction is a load, and its result simply won't be available in time:

As indicated, this can be resolved by stalling the pipeline, delaying the initiation of the second instruction for 1 cycle.

Again, if we can detect this situation, we can in principle impose the solution shown above.

A pipeline stall is often referred to as a bubble.

Control Hazards: Stall on Branch

A control hazard occurs when the instruction that was fetched is not the one that is needed.

Note that our pipeline discussion so far assumes sequential execution.

When the current instruction is a conditional branch, this may be incorrect.

One approach would be to stall when a branch instruction is discovered, until the necessary computations are completed and then fetch the correct instruction next.
Control Hazards: Branch Prediction

A second approach is to predict whether the branch will be taken and load the corresponding instruction into the pipeline.

If we guess that the branch will NOT be taken, we just increment the PC, fetch and proceed:

This worked out perfectly. There was no delay… however, what if the branch HAD been taken?

More sophisticated variants actually retain information (history) about individual branch instructions and use that history to predict future behavior.

Control Hazards: Delayed Branch

A third approach is to delay the time at which the branch takes effect by always executing the next sequential instruction following a branch instruction, and then making the branch (if necessary) immediately after one-instruction delay.

To do this, the assembler will automatically accomplish this by placing an instruction immediately after the branch instruction that is not affected by the branch.

This approach is used in the MIPS architecture.

Of course, it’s not always that simple. What would we do if the add instruction had stored its result in one of the registers used by the beq instruction?
Have assembler guarantee no hazards. One approach would be to rearrange statements; another would be to insert no-op (no operation) statements, to induce the necessary stalls.

Where do we insert the “no-ops”?

\[
\begin{align*}
\text{sub} &: \$2, \$1, \$3 \\
\text{and} &: \$12, \$2, \$5 \\
\text{or} &: \$13, \$6, \$2 \\
\text{add} &: \$14, \$2, \$2 \\
\text{sw} &: \$15, 100(\$2)
\end{align*}
\]

Problem: this really slows us down!

---

What difficulties can you identify in the following code sequences?

1. \[
\begin{align*}
\text{lw} &: \$t0, 0(\$t0) \\
\text{add} &: \$t1, \$t0, \$t0
\end{align*}
\]

   The result of the lw is needed by the add during its second cycle; a stall is needed.

2. \[
\begin{align*}
\text{add} &: \$t1, \$t0, \$t0 \\
\text{addi} &: \$t2, \$t0, 5 \\
\text{addi} &: \$t4, \$t1, 5
\end{align*}
\]

   The result of the addi is needed by second addi during its second cycle, but isn’t written to the register file until the next cycle; however, we can forward the value since it’s been computed two cycles before it’s written.

3. \[
\begin{align*}
\text{addi} &: \$t1, \$t0, 1 \\
\text{addi} &: \$t2, \$t0, 2 \\
\text{addi} &: \$t3, \$t0, 3 \\
\text{addi} &: \$t4, \$t0, 4 \\
\text{addi} &: \$t5, \$t0, 5
\end{align*}
\]

   No problems here.
Basic Idea Redux

What do we need to add/modify to actually split the datapath into stages?

Instructions and data generally move from left to right.

Two exceptions:
- write-back of data to register file
- selecting the next value for the PC (incremented PC versus branch address)

The cases where data flows right to left do not affect the current instruction, but rather they affect later instructions.

The first case can lead to a data hazard; the second can lead to a control hazard.

Analysis

Consider a timeline showing overlapping pipeline logic for a set of instructions:

Problem: the original contents of the IR will be lost when the next instruction is fetched, but those original contents are needed at a later cycle as well. (Why?)

So, how do we fix this?

Basically, we need the ability to preserve results generated in each stage until they are actually needed.

So, we can add a bank of storage locations between each pair of stages.
Datapath with Pipeline Registers

Here's a first attempt; we just add an unspecified amount of storage between datapath stages:

- Incremented PC value is passed forward
- Original IR contents are passed forward... for later use
- IR is embedded here

How large must the IF/ID register storage be?

The next order of business is to examine the other inter-stage registers.

Boundary Analysis

Let's consider the "boundaries":

- What about the PC? In effect it IS a pipeline register, feeding the IF stage.
- No pipeline register is needed after the WB stage. Why?

The next order of business is to examine the other inter-stage registers.
Load Instruction Analysis: IF

PC is incremented by 4. Result is written back into PC, but also into IF/ID pipeline register in case it is needed later... ...don't know what the instruction actually is yet.

Instruction is fetched into the pipeline register.

Register shading indicates whether a write (left half) or a read (right half) is occurring.

Load Instruction Analysis: ID

Incremented PC value is also passed forward to next-state pipeline register.

Values read from register file, and extended immediate field are stored in the next pipeline register.

Register read numbers are supplied to register file.

16-bit immediate field is supplied to the sign-extender.
Contents of first read register and sign-extended immediate are sent to the ALU from the pipeline register. Resulting sum is then placed into next-stage pipeline register.

Incremented PC value is NOT carried forward... why?

Computed address is passed from pipeline register to memory unit. Retrieved data is written into next-stage pipeline register.
Load Instruction Analysis: WB

So, what have we learned?

One key point: a logical component of the datapath, like the ALU, must be used only in a single pipeline stage… otherwise, we have a structural hazard.

If you were paying very close attention, we've uncovered a bug in the proposed handling of a load instruction.

Take another look at what happens in the final stage… where does the number of the write register come from?

Alas, we will no longer have the original instruction in the IF/ID pipeline register, and so we won't have the information we need.

Solution: pass the write register number forward to the MEM/WB pipeline register, so it is still available during the final stage.
Further, similar analysis of other instructions leads to a corrected, but incomplete, pipeline design:

An important question is just how much storage must each pipeline register provide?

That is left to the reader.

We have 5 stages. What needs to be controlled in each stage?
- Instruction Fetch and PC Increment
- Instruction Decode / Register Fetch
- Execution
- Memory Stage
- Write Back

How would control be handled in an automobile plant?
- a fancy control center telling everyone what to do?
- should we use a finite state machine?
Pipelining

Identify the necessary control signals:

Pass control signals along just like the data:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg</td>
<td>Dist</td>
<td>Op1</td>
</tr>
<tr>
<td>R-format</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Pipelining

Use temporary results, don’t wait for them to be written
Forwarding

The main idea (some details not shown)

---

Inadequacy of Forwarding

Load word can still cause a hazard:
- an instruction tries to read a register following a load instruction that writes to the same register.

Thus, we need a hazard detection unit to “stall” the load instruction
We can resolve the problem by keeping the dependent instruction in the same stage:

Thus, we need a hazard detection unit to “stall” the load instruction.

Hazard Detection Unit

Stall by letting an instruction that won’t write anything go forward.
Branch Hazards

When we decide to branch, other instructions are in the pipeline!

We are predicting “branch not taken”
- need to add hardware for flushing instructions if we are wrong

Flushing Instructions

Note: we’ve also moved branch decision to ID stage
Branches

Pipelining

If the branch is taken, we have a penalty of one cycle
For our simple design, this is reasonable
With deeper pipelines, penalty increases and static branch prediction drastically hurts performance
Solution: dynamic branch prediction

A 2-bit prediction scheme

Branch Prediction

Pipelining

Sophisticated Techniques:
- A “branch target buffer” to help us look up the destination
- Correlating predictors that base prediction on global behavior and recently executed branches (e.g., prediction for a specific branch instruction based on what happened in previous branches)
- Tournament predictors that use different types of prediction strategies and keep track of which one is performing best.
- A “branch delay slot” which the compiler tries to fill with a useful instruction (make the one cycle delay part of the ISA)

Branch prediction is especially important because it enables other more advanced pipelining techniques to be effective!
Modern processors predict correctly 95% of the time!
Improving Performance

Try and avoid stalls! E.g., reorder these instructions:

lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)

Dynamic Pipeline Scheduling
- Hardware chooses which instructions to execute next
- Will execute instructions out of order (e.g., doesn’t wait for a dependency to be resolved, but rather keeps going!)
- Speculates on branches and keeps the pipeline full (may need to rollback if prediction incorrect)

Trying to exploit instruction-level parallelism

Advanced Pipelining

Increase the depth of the pipeline
Start more than one instruction each cycle (multiple issue)
Loop unrolling to expose more ILP (better scheduling)
“Superscalar” processors
- DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
All modern processors are superscalar and issue multiple instructions usually with some limitations (e.g., different “pipes”)
VLIW: very long instruction word, static multiple issue (relies more on compiler technology)

This class has given you the background you need to learn more!
Pipelining does not improve latency, but does improve throughput

- Slow
  - Single-cycle (Section 5.4)
  - Multicycle (Section 5.5)
- Faster
  - Pipelined
  - Multiple-issue (Section 6.9, 6.10)

Instructions per clock (IPC = 1/CPI)