Pipelined CPUs

Where are the registers?

Study Chapter 6 of Text

Amdahl's Law

\[ t_{\text{improved}} = \frac{t_{\text{affected}}}{r_{\text{speedup}}} + t_{\text{unaffected}} \]

Example:

"Suppose a program runs in 100 seconds on a machine, where
multiplies are responsible for 80 seconds of this time. How much do we have
to improve the speed of multiplication if we want the program to run 4 times
faster?"

\[ 25 = \frac{80}{r} + 20 \quad r = 16x \]

How about making it 5 times faster?

\[ 20 = \frac{80}{r} + 20 \quad r = ? \]

Principle: Make the common case fast
Example

Suppose we enhance a machine making all floating-point instructions run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if only half of the 10 seconds is spent executing floating-point instructions?

We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?

Remember

Performance is specific to a particular program.
Total execution time is a consistent summary of performance.

For a given architecture performance increases come from:
- increases in clock rate (without adverse CPI affects)
- improvements in processor organization that lower CPI
- compiler enhancements that lower CPI and/or instruction count

Pitfall: Expecting improvements in one aspect of a machine’s performance to affect the total performance.

You should not always believe everything you read! Read carefully!
Pipelined CPUs

Where are the registers?

Study Chapter 6 of Text

Review of CPU Performance

MIPS = Millions of Instructions/Second
Freq = Clock Frequency, MHz
CPI = Clocks per Instruction

To Increase MIPS:

1. DECREASE CPI.
   - RISC simplicity reduces CPI to 1.0.
   - CPI below 1.0? State-of-the-art multiple instruction issue

2. INCREASE Freq.
   - Freq limited by delay along longest combinational path; hence
   - PIPELINING is the key to improving performance.
miniMIPS Timing

The diagram on the left illustrates the Data Flow of miniMIPS

Wanted: longest path

Complications:

• some apparent paths aren't "possible"
• functional units have variable execution times (eg, ALU)
• time axis is not to scale (eg, $t_{PD,\text{MEM}}$ is very big!)

Where Are the Bottlenecks?

Pipelining goal:

Break LONG combinational paths

$\rightarrow$ memories, ALU in separate stages
Ultimate Goal: 5-Stage Pipeline

GOAL: Maintain (nearly) 1.0 CPI, but increase clock speed to barely include slowest components (mems, regfile, ALU)

APPROACH: structure processor as 5-stage pipeline:

- **IF** (Instruction Fetch): Maintains PC, fetches one instruction per cycle and passes it to
- **ID/RF** (Instruction Decode/Register File): Decode control lines and select source operands
- **ALU** (ALU stage): Performs specified operation, passes result to
- **MEM** (Memory stage): If it's a lw, use ALU result as an address, pass mem data (or ALU result if not lw) to
- **WB** (Write-Back stage): writes result back into register file.

miniMIPS Timing

Different instructions use various parts of the data path.

<table>
<thead>
<tr>
<th>Program execution order</th>
<th>Time</th>
<th>1 instr every 14 nS, 14 nS, 20 nS, 9 nS, 19 nS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLK</td>
<td></td>
</tr>
<tr>
<td>add $4, $5, $6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq $1, $2, 40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $3, 30($0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jal 20000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $2, 20($4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 nS</td>
<td></td>
<td>Instruction Fetch</td>
</tr>
<tr>
<td>2 nS</td>
<td></td>
<td>Instruction Decode</td>
</tr>
<tr>
<td>2 nS</td>
<td></td>
<td>Register Prop Delay</td>
</tr>
<tr>
<td>5 nS</td>
<td></td>
<td>ALU Operation</td>
</tr>
<tr>
<td>4 nS</td>
<td></td>
<td>Branch Target</td>
</tr>
<tr>
<td>6 nS</td>
<td></td>
<td>Data Access</td>
</tr>
<tr>
<td>1 nS</td>
<td></td>
<td>Register Setup</td>
</tr>
</tbody>
</table>

This is an example of a "Asynchronous Globally-Timed" control strategy (see Lecture 8). Such a system would vary the clock period based on the instruction being executed. This leads to complicated timing generation, and, in the end, slower systems, since it is not very compatible with pipelining!
Uniform miniMIPS Timing

With a fixed clock period, we have to allow for the worse case.

By accounting for the "worse case" path (i.e. allowing time for each possible combination of operations) we can implement a "Synchronous Globally-Timed" control strategy. This simplifies timing generation, enforces a uniform processing order, and allows for pipelining!

Step 1: A 2-Stage Pipeline

IF

EXE
2-Stage Pipe Timing

Improves performance by increasing instruction throughput.
Ideal speedup is number of pipeline stages in the pipeline.

2-Stage w/2-Cycle Loads & Stores

Further improves performance, with slight increase in control complexity.
Some 1st generation (pre-cache) RISC processors used this approach.

The clock rate of this variant is more than twice that of our original design. Does that mean it is twice as fast? Not likely. In practice, as many as 30% of instructions access memory. Thus, the effective speed up is:

\[
\text{speed up} = \frac{\text{old clock period}}{\text{new clock period}}
\]

= \frac{20}{8(1.3)} = 1.923
2-Stage Pipelined Operation

Consider a sequence of instructions:

\[
\begin{align*}
... & \quad \text{addi } \$t2,\$t1,1 \\
& \quad \text{xor } \$t2,\$t1,\$t2 \\
& \quad \text{sltiu } \$t3,\$t2,1 \\
& \quad \text{srl } \$t2,\$t2,1 \\
... & 
\end{align*}
\]

Executed on our 2-stage pipeline:

\[
\begin{array}{cccccccc}
& & i & i+1 & i+2 & i+3 & i+4 & i+5 & i+6 \\
\text{IF} & \text{addi} & \text{xor} & \text{sltiu} & \text{srl} & ... \\
\text{EXE} & \text{addi} & \text{xor} & \text{sltiu} & \text{srl} & ...
\end{array}
\]

Recall "Pipeline Diagrams" from Lecture 8.

It can't be this easy!

Pipeline Control Hazards

BUT consider instead:

\[
\begin{align*}
\text{loop: } & \quad \text{add } \$t1,\$t1,\$t0 \\
& \quad \text{srl } \$t2,\$t2,1 \\
& \quad \text{bne } \$t2,\$0,\text{loop} \\
& \quad \text{andi } \$t0,\$0,\$t2,1 \\
... & 
\end{align*}
\]

\[
\begin{array}{cccccccc}
& & i & i+1 & i+2 & i+3 & i+4 & i+5 & i+6 \\
\text{IF} & \text{add} & \text{srl} & \text{bne} & \text{andi} & ... \\
\text{EXE} & \text{add} & \text{cmp} & \text{bne} & ? & ... \\
\end{array}
\]

This is the cycle where the branch decision is made... but we've already fetched the following instruction which should be executed only if branch is not taken!

Pipelining HAZARDS are situations where the next instruction cannot execute in the next clock cycle. There are two forms of hazards, CONTROL and STRUCTURAL.
**Branch Delay Slots**

**PROBLEM:** One (or more) following instructions are fetched before the branch decision is made (to take, or not to take).

**POSSIBLE SOLUTIONS:**

1. Make hardware “annul” instructions following taken branches, e.g., by disabling WERF and WR.

2. “Program around it”. Either
   a) Follow each BNE/BEQ with a NOP instruction; or
   b) Make compiler clever enough to move USEFUL instructions into the branch delay slots
      i. Always execute instructions in delay slots
      ii. Conditionally execute instructions in delay slots

*Delay slots also apply to the jump instructions, j, jal, and jr*
Branch Solution 1

Make the hardware annul instructions in the branch delay slots of a taken branch.

Pros: Programs run identically on both unpipelined and pipelined hardware
Cons: in SPEC benchmarks 14% of instructions are taken branches → 14/114 = 12% of total cycles are annulled

Branch Annulment Hardware

We need to annul on taken branches, which is basically the same decoding logic as selecting PCSEL = 1.
Branch Alternative 2a

Always fill branch delay slots with NOP instructions (i.e., the software equivalent of alternative 1)

Branch taken

Pros: Does not require H/W modifications, only compiler changes
Cons: NOPs make code longer; 12% of cycles spent executing NOPs

Branch Alternative 2b(i)

Put USEFUL instructions in the branch delay slots; remember they will be executed whether the branch is taken or not

Pros: only two “extra” instructions are executed (on last iteration)
Cons: finding “useful” instructions that should always be executed is difficult; clever rewrite may be required. Program executes differently on unpipelined implementation.

This is the standard approach for pipelined MIPS implementations
### Branch Alternative 2b(ii)

Put USEFUL instructions in the branch delay slots; annul them if branch doesn’t behave as predicted.

**Pros:**
- Only one instruction is annulled (on last iteration); about 70% of branch delay slots can be filled with useful instructions.

**Cons:**
- Program executes differently on naïve unpipelined implementation;
- Difficult to utilize with more than one delay slot.

<table>
<thead>
<tr>
<th>i</th>
<th>i+1</th>
<th>i+2</th>
<th>i+3</th>
<th>i+4</th>
<th>i+5</th>
<th>i+6</th>
<th>i+7</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>add</td>
<td>srl</td>
<td>bne.t</td>
<td>add</td>
<td>srl</td>
<td>bne.t</td>
<td>add</td>
</tr>
<tr>
<td>EXE</td>
<td>add</td>
<td>srl</td>
<td>bne.t</td>
<td>add</td>
<td>srl</td>
<td>bne.t</td>
<td>andi</td>
</tr>
</tbody>
</table>

#### Architectural Issue:

**Branch Decision Timing**

The number of branch delay slots is determined by where in the pipeline the branch decision is made. Consider the 5-stage miniMIPS pipeline shown on the right.

Where is the branch decision resolved?

- `beq rs,rt,offset`
  - if `(Reg[rs] == Reg[rt])`
  - `PC ← PC + 4 + 4*SEXT(offset)`

The decision is based on the ALU’s Z-flag, which is determined at the very end of the ALU stage, nearly 2 clocks after the instruction fetch. Therefore, a naïve miniMIPS implementation has at least TWO branch delay slots.

![MiniMIPS Pipeline Diagram](image)
Recall that the Instruction Decode and Register Access stage was already one of our fastest paths. The logic for testing for the equality of two inputs is called a comparator.

Step 2: 4-Stage miniMIPS

Treat register file as two separate devices: combinational READ, clocked WRITE at end of pipe.

What other information do we have to pass down pipeline? PC (return addresses) instruction fields (decoding)

What sort of improvement should we expect in cycle time?
4-Stage miniMIPS Operation

Consider a sequence of instructions:

```
addi $t0, $t0, 1
sll $t1, $t1, 2
andi $t2, $t2, 15
sub $t3, $0, $t3
```

Executed on our 4-stage pipeline:

<table>
<thead>
<tr>
<th>TIME (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>IF</td>
</tr>
<tr>
<td>RF</td>
</tr>
<tr>
<td>ALU</td>
</tr>
<tr>
<td>WB</td>
</tr>
</tbody>
</table>

Pipeline “Structural Hazard”

BUT consider instead:

```
addi $t0, $t0, 1
sll $t1, $t0, 2
andi $t2, $t2, 15
sub $t3, $0, $t3
```

```
Oops! sll is trying to read Reg[8] during cycle i+2 but addi doesn’t write its result into Reg[8] until the end of cycle i+3!
```

Stuff like this never happened when we did pipelining before. Why now?

Before, we forbade feedback. Can’t do that with a useful CPU. How can we fix this one?
Data Hazard Solution 1

“Program around it”
... document weirdo semantics, declare it a software problem.
- Breaks sequential semantics!
  (Order of instruction execution is not obvious)
- Costs code efficiency.

EXAMPLE: Rewrite

\[\begin{align*}
\text{addi} & \ t0, \ t0, 1 \\
\text{sll} & \ t1, \ t0, 2 \\
\text{andi} & \ t2, \ t2, 15 \\
\text{sub} & \ t3, \ 0, \ t3 \\
\text{addi} & \ t0, \ t0, 1 \\
\text{andi} & \ t2, \ t2, 15 \\
\text{sub} & \ t3, \ 0, \ t3 \\
\text{sll} & \ t1, \ t0, 2
\end{align*}\]

How often can we do this?

Not Very.

Data Hazard Solution 2

Stall the pipeline:
Freeze IF, RF stages for 2 cycles, inserting NOPs into ALU-stage instruction register

\[\begin{align*}
\text{IF} & \ \text{addi} & \ \text{sll} & \ \text{andi} & \ \text{andi} \\
\text{RF} & \ \text{addi} & \ \text{sll} & \ \text{sll} & \ \text{andi} \\
\text{ALU} & \ \text{addi} & \ \text{NOP} & \ \text{NOP} & \ \text{sll} & \ \text{andi} \\
\text{WB} & \ \text{addi} & \ \text{NOP} & \ \text{NOP} & \ \text{sll}
\end{align*}\]

Drawback: Added NOPs mean “wasted” cycles. Lot’s of wasted cycles.
(A large percentage of instructions depend on results from the immediately preceding instruction)
Data Hazard Solution 3

Bypass (aka forwarding) Paths:
Add extra data paths & control logic to re-route data in problem cases.

<table>
<thead>
<tr>
<th>i</th>
<th>i+1</th>
<th>i+2</th>
<th>i+3</th>
<th>i+4</th>
<th>i+5</th>
<th>i+6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>sll</td>
<td>andi</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RF</td>
<td>addi</td>
<td>sll</td>
<td>andi</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>addi</td>
<td>sll</td>
<td>andi</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>sll</td>
<td>andi</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Idea: The result from the addi, which will be written into the register file at the end of cycle i+3, is actually available at output of the ALU during cycle i+2 – just in time for it to be used by sll in the RF stage!

Bypass Paths (I)

SELECT this Bypass path if

\[ Op_{RF} = \text{read } Rs \text{ and } \]

\[ \begin{cases} 
\text{Op}_{ALU} = \text{R-type and } Rs_{RF} = Rs_{ALU} \\
\text{Op}_{ALU} = \text{I-type and } Rs_{RF} = Rs_{ALU} 
\end{cases} \]

i.e., instructions which use ALU to compute result

\[ \text{and } Rs^{IF} \neq O \]
Bypass Paths (II)

SELECT this Bypass path if

\[ Op^{RF} = \text{reads Ra} \]
\[ \text{and } Ra^{RF} = 0 \]
\[ \text{and not using ALU bypass} \]
\[ \text{and WERF = 1} \]
\[ \text{and } Rs^{RF} = \text{WA} \]

Why can't we get it from the register file? It's being written this cycle!

Next Time

Many More Bypasses Ahead