Lecture 8

Today
- Finish single-cycle datapath/control path
- Look at its performance and how to improve it.
The final datapath
The control unit is responsible for setting all the control signals so that each instruction is executed properly.

- The control unit’s input is the 32-bit instruction word.
- The outputs are values for the blue control signals in the datapath.

Most of the signals can be generated from the instruction opcode alone, and not the entire 32-bit word.

To illustrate the relevant control signals, we will show the route that is taken through the datapath by R-type, lw, sw and beq instructions.
- The R-type instructions include `add`, `sub`, `and`, `or`, and `slt`.
- The `ALUOp` is determined by the instruction’s “func” field.
lw instruction path

- An example load instruction is `lw $t0, -4($sp)`.
- The **ALUOp** must be 010 (add), to compute the effective address.
sw instruction path

- An example store instruction is `sw $a0, 16($sp)`.
- The **ALUOp** must be 010 (add), again to compute the effective address.
• One sample branch instruction is \texttt{beq $at, $0, offset}.
• The \texttt{ALUOp} is 110 (subtract), to test for equality.
### Control signal table

<table>
<thead>
<tr>
<th>Operation</th>
<th>RegDst</th>
<th>RegWrite</th>
<th>ALUSrc</th>
<th>ALUOp</th>
<th>MemWrite</th>
<th>MemRead</th>
<th>MemToReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>and</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>or</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>slt</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>010</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

- sw and beq are the only instructions that do not write any registers.
- lw and sw are the only instructions that use the constant field. They also depend on the ALU to compute the effective memory address.
- ALUOp for R-type instructions depends on the instructions’ func field.
- The PCSrc control signal (not listed) should be set if the instruction is beq and the ALU’s Zero output is true.
Generating control signals

- The control unit needs 13 bits of inputs.
  - Six bits make up the instruction’s opcode.
  - Six bits come from the instruction’s func field.
  - It also needs the Zero output of the ALU.

- The control unit generates 10 bits of output, corresponding to the signals mentioned on the previous page.

- You can build the actual circuit by using big K-maps, big Boolean algebra, or big circuit design programs.

- The textbook presents a slightly different control unit.
Summary of Single-Cycle Implementation

- A **datapath** contains all the functional units and connections necessary to implement an instruction set architecture.
  - For our **single-cycle implementation**, we use two separate memories, an ALU, some extra adders, and lots of multiplexers.
  - MIPS is a 32-bit machine, so most of the buses are 32-bits wide.
- The **control unit** tells the datapath what to do, based on the instruction that’s currently being executed.
  - Our processor has ten **control signals** that regulate the datapath.
  - The control signals can be generated by a combinational circuit with the instruction’s 32-bit binary encoding as input.
- Next, we’ll see the performance limitations of this single-cycle machine and try to improve upon it.
Last time we saw a MIPS single-cycle datapath and control unit.

Today, we’ll explore factors that contribute to a processor’s execution time, and specifically at the performance of the single-cycle machine.

Next time, we’ll explore how to improve on the single cycle machine’s performance using pipelining.
Three Components of CPU Performance

\[
\text{CPU time}_{X,p} = \text{Instructions executed}_p \times \text{CPI}_{X,p} \times \text{Clock cycle time}_X
\]
Instructions Executed

- Instructions executed:
  - We are not interested in the **static instruction count**, or how many lines of code are in a program.
  - Instead we care about the **dynamic instruction count**, or how many instructions are actually executed when the program runs.

- There are three lines of code below, but the number of instructions executed would be 2001.

```
li $a0, 1000
Ostrich: sub $a0, $a0, 1
bne $a0, $0, Ostrich
```
The average number of clock cycles per instruction, or CPI, is a function of the machine and program.

- The CPI depends on the actual instructions appearing in the program— a floating-point intensive application might have a higher CPI than an integer-based program.
- It also depends on the CPU implementation. For example, a Pentium can execute the same instructions as an older 80486, but faster.

In CS231, we assumed each instruction took one cycle, so we had $\text{CPI} = 1$.
- The CPI can be $>1$ due to memory stalls and slow instructions.
- The CPI can be $<1$ on machines that execute more than 1 instruction per cycle (superscalar).
One “cycle” is the minimum time it takes the CPU to do any work.
  – The clock cycle time or clock period is just the length of a cycle.
  – The clock rate, or frequency, is the reciprocal of the cycle time.

Generally, a higher frequency is better.

Some examples illustrate some typical frequencies.
  – A 500MHz processor has a cycle time of 2ns.
  – A 2GHz (2000MHz) CPU has a cycle time of just 0.5ns (500ps).
Execution time, again

\[ CPU \text{ time}_{x,p} = \text{Instructions executed}_p \times CPI_{x,p} \times \text{Clock cycle time}_x \]

- The easiest way to remember this is match up the units:

\[
\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycle}}
\]

- Make things faster by making any component smaller!!

<table>
<thead>
<tr>
<th></th>
<th>Program</th>
<th>Compiler</th>
<th>ISA</th>
<th>Organization</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Executed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock Cycle Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Often easy to reduce one component by increasing another
Example 1: ISA-compatible processors

- Let’s compare the performances of two x86-based processors.
  - An 800MHz AMD Duron, with a CPI of 1.2 for an MP3 compressor.
  - A 1GHz Pentium III with a CPI of 1.5 for the same program.
- Compatible processors implement identical instruction sets and will use the same executable files, with the same number of instructions.
- But they implement the ISA differently, which leads to different CPIs.

\[
\text{CPU time}_{\text{AMD},P} = \text{Instructions}_P \times \text{CPI}_{\text{AMD},P} \times \text{Cycle time}_{\text{AMD}} \\
= \\
= \\

\text{CPU time}_{\text{P3},P} = \text{Instructions}_P \times \text{CPI}_{\text{P3},P} \times \text{Cycle time}_{\text{P3}} \\
= \\
= \]
How the add goes through the datapath

PC + 4

Add

Shift left 2

MemWrite

MemToReg

PCSrc

0 Mux 1

0 Mux

1 Mux

MemRead

Data memory

ALU Zero

ALUOp

ALUSrc

Result

RegWrite

ALU

Registers

Write data

Write register

Read data 2

Read register 2

Read data 1

Read register 1

RegWrite

RegDst

I [15 - 0]

10100

I [15 - 11]

0 Mux 1

I [20 - 16]

01010

I [25 - 21]

01001

Instruction memory

Read Instruction address [31-0]

Instruction memory

1 Mux

00...11

00...10

00...01

0 Mux

PC

4
Performance of Single-cycle Design

\[
\text{CPU time}_{X,p} = \text{Instructions executed}_p \times \text{CPI}_{X,p} \times \text{Clock cycle time}_X
\]
Edge-triggered state elements

- In an instruction like `add $t1, $t1, $t2`, how do we know $t1$ is not updated until *after* its original value is read?
- We’ll assume that our state elements are **positive edge triggered**, and are updated only on the positive edge of a clock signal.
  - The register file and data memory have explicit write control signals, `RegWrite` and `MemWrite`. These units can be written to only if the control signal is asserted *and* there is a positive clock edge.
  - In a single-cycle machine the PC is updated on each clock cycle, so we don’t bother to give it an explicit write control signal.
The datapath and the clock

1. On a positive clock edge, the PC is updated with a new address.
2. A new instruction can then be loaded from memory. The control unit sets the datapath signals appropriately so that
   – registers are read,
   – ALU output is generated,
   – data memory is read or written, and
   – branch target addresses are computed.
3. Several things happen on the next positive clock edge.
   – The register file is updated for arithmetic or lw instructions.
   – Data memory is written for a sw instruction.
   – The PC is updated to point to the next instruction.

- In a **single-cycle datapath** everything in Step 2 must complete within one clock cycle, before the next positive clock edge.

*How long is that clock cycle?*
Compute the longest path in the add instruction
The slowest instruction...

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the slowest instruction.
- For example, `lw $t0, -4($sp)` is the slowest instruction needing ___ns.
  - Assuming the circuit latencies below.
The slowest instruction...

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the slowest instruction.
- For example, `lw $t0, -4($sp)` needs 8ns, assuming the delays shown here.

  reading the instruction memory       2ns
  reading the base register $sp        1ns
  computing memory address $sp-4       2ns
  reading the data memory              2ns
  storing data back to $t0             1ns

\[
\begin{align*}
\text{I [25-21]} & \quad \text{I [20-16]} & \quad \text{I [15-11]} & \quad \text{I [15-0]} \\
\text{Write register 1} & \quad \text{Write register 2} & \quad \text{Write data} & \quad \text{Read Instruction address [31-0]} \\
\text{Instruction memory} & \quad \text{Instruction memory} & \quad \text{Instruction memory} & \quad \text{Instruction memory} \\
2 \text{ ns} & \quad 2 \text{ ns} & \quad 1 \text{ ns} & \quad 0 \text{ ns}
\end{align*}
\]

\[
\begin{align*}
\text{0 Mux 1} & \quad \text{0 Mux 1} & \quad \text{0 Mux 1} & \quad \text{0 Mux 1} \\
\text{Read data 1} & \quad \text{Read data 2} & \quad \text{Read data 1} & \quad \text{Read data 1} \\
\text{ALU} & \quad \text{ALU} & \quad \text{ALU} & \quad \text{ALU} \\
\text{Zero} & \quad \text{Zero} & \quad \text{Zero} & \quad \text{Zero} \\
\text{Result} & \quad \text{Result} & \quad \text{Result} & \quad \text{Result} \\
\text{2 ns} & \quad \text{2 ns} & \quad \text{2 ns} & \quad \text{2 ns} \\
\text{Read address} & \quad \text{Read address} & \quad \text{Read address} & \quad \text{Read address} \\
\text{Write address} & \quad \text{Write address} & \quad \text{Write address} & \quad \text{Write address} \\
\text{Data memory} & \quad \text{Data memory} & \quad \text{Data memory} & \quad \text{Data memory} \\
1 \text{ ns} & \quad 0 \text{ ns} & \quad 0 \text{ ns} & \quad 0 \text{ ns} \\
\end{align*}
\]
...determines the clock cycle time

- If we make the cycle time 8ns then every instruction will take 8ns, even if they don’t need that much time.
- For example, the instruction `add $s4, $t1, $t2` really needs just 6ns.

reading the instruction memory \(2\) ns
reading registers \$t1 and \$t2 \(1\) ns
computing \$t1 + \$t2 \(2\) ns
storing the result into \$s0 \(1\) ns
How bad is this?

- With these same component delays, a `sw` instruction would need 7ns, and `beq` would need just 5ns.
- Let’s consider the `gcc` instruction mix from p. 189 of the textbook.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>48%</td>
</tr>
<tr>
<td>Loads</td>
<td>22%</td>
</tr>
<tr>
<td>Stores</td>
<td>11%</td>
</tr>
<tr>
<td>Branches</td>
<td>19%</td>
</tr>
</tbody>
</table>

- With a single-cycle datapath, each instruction would require 8ns.
- But if we could execute instructions as fast as possible, the average time per instruction for gcc would be:

  \[
  (48\% \times 6\text{ns}) + (22\% \times 8\text{ns}) + (11\% \times 7\text{ns}) + (19\% \times 5\text{ns}) = 6.36\text{ns}
  \]

- The single-cycle datapath is about 1.26 times slower!
It gets worse...

- We’ve made **very** optimistic assumptions about memory latency:
  - Main memory accesses on modern machines is >50ns.
    - For comparison, an ALU on an AMD Opteron takes ~0.3ns.
- Our worst case cycle (loads/stores) includes 2 memory accesses
  - A modern single cycle implementation would be stuck at <10Mhz.
  - Caches will improve common case access time, not worst case.
- Tying frequency to worst case path violates first law of performance!!
  - “Make the common case fast” (we’ll revisit this often)
Summary

- **Performance** is one of the most important criteria in judging systems.
  - Here we’ll focus on **Execution time**.

- Our main performance equation explains how performance depends on several factors related to both hardware and software.

\[
\text{CPU time}_{x,p} = \text{Instructions executed}_p \times \text{CPI}_{x,p} \times \text{Clock cycle time}\_x
\]

- It can be hard to measure these factors in real life, but this is a useful guide for comparing systems and designs.

- A single-cycle CPU has two main disadvantages.
  - The cycle time is limited by the worst case latency.
  - It isn’t efficiently using its hardware.

- Next time, we’ll see how this can be rectified with pipelining.