CS2100 Computer Organisation

The Processor: Control
(AY2016/2017) Semester 2

Ack: Some slides are adapted from Dr Tulika Mitra’s CS1104 notes and Mr. Aaron Tan’s CS2100 notes
Road Map: Part II

- Processor: Control
  - The control unit
  - Control Signals
  - ALU Control Signal

- Performance
- Assembly Language
- Processor: Datapath
- Processor: Control
- Pipelining
- Cache
## Identified Control Signals

<table>
<thead>
<tr>
<th>Control Signal</th>
<th>Execution Stage</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RegDst</strong></td>
<td>Decode / Operand Fetch</td>
<td>Select the destination register number</td>
</tr>
<tr>
<td><strong>RegWrite</strong></td>
<td>Decode/Operand Fetch Result Write</td>
<td>Enable writing of register</td>
</tr>
<tr>
<td><strong>ALUSrc</strong></td>
<td>ALU</td>
<td>Select the 2\textsuperscript{nd} operand for ALU</td>
</tr>
<tr>
<td><strong>ALUControl</strong></td>
<td>ALU</td>
<td>Select the operation to be performed</td>
</tr>
<tr>
<td><strong>MemRead / MemWrite</strong></td>
<td>Memory</td>
<td>Enable reading/writing of data memory</td>
</tr>
<tr>
<td><strong>MemToReg</strong></td>
<td>Result Write</td>
<td>Select the result to be written back to register file</td>
</tr>
<tr>
<td><strong>PCSrc</strong></td>
<td>Memory / Result Write</td>
<td>Select the next PC value</td>
</tr>
</tbody>
</table>
Generating Control Signals: Idea

- The control signals are generated based on the instruction to be executed:
  - Opcode ➔ Instruction Format
  - Example:
    - R-Format instruction ➔ RegDst = 1 (use Inst[15:11])
  - R-Type instruction has additional information:
    - The 6-bit "funct" (function code, Inst[5:0]) field

- Idea:
  - Design a combinatorial circuit to generate these signals based on Opcode and possibly Function code
  - A control unit is needed (a draft design is shown next)
The Control Unit (draft)
Let's Implement the Control Unit!

- **Approach:**
  - Take note of the instruction subset to be implemented:
    - Opcode and Function Code (if applicable)
  - Go through each signal:
    - Observe how the signal is generated based on the instruction opcode and/or function code
  - Construct truth table
  - Design the control unit using logic gates
### Review: MIPS Instruction Subset

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>0_{16}</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>20_{16}</td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>0_{16}</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>22_{16}</td>
</tr>
<tr>
<td><strong>and</strong></td>
<td>0_{16}</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>24_{16}</td>
</tr>
<tr>
<td><strong>or</strong></td>
<td>0_{16}</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>25_{16}</td>
</tr>
<tr>
<td><strong>slt</strong></td>
<td>0_{16}</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>2A_{16}</td>
</tr>
</tbody>
</table>

#### R-type

- **lw**
  - 23_{16} rs rd offset
- **sw**
  - 2B_{16} rs rd offset

#### I-type

- **beq**
  - 4_{16} rs rd offset
Control Signal: RegDst

- **False (0):** Write register = $\text{Inst}[20:16]$
- **True (1):** Write register = $\text{Inst}[15:11]$
Control Signal: **RegWrite**

- **False (0):** No register write
- **True (1):** New value will be written
Control Signal: **ALUSrc**

- **False (0):** Operand2 = Register Read Data 2
- **True (1):** Operand2 = SignExt(\textbf{Inst}[15:0])
Control Signal: **MemRead**

- **False (0):** Not performing memory read access
- **True (1):** Read memory using `Address`
Control Signal: **MemWrite**

- **False (0):** Not performing memory write operation
- **True (1):** memory\[**Address**\] ← Register Read Data 2
Control Signal: **MemToReg**

- **True (1):** Register write data = Memory read data
- **False (0):** Register write data = ALU Result

**IMPORTANT:**
The input of MUX is swapped in this case
Control Signal: **PCSrc**

- The "isZero?" signal from the ALU gives us the actual branch outcome (taken / not taken)
- **Idea:** "If instruction is a branch AND taken, then…"
Control Signal: **PCS\text{Src}**

- **False** (0): Next PC = PC + 4
- **True** (1): Next PC = SignExt(\text{Inst}[15:0]) \ll 2 + (PC + 4)

\[\text{PCS\text{Src}} = (\text{Branch AND isZero})\]
Midpoint Check

- We have gone through almost all of the signals:
  - Left with the more challenging ALUControl signal

- Observation so far:
  - The signals discussed so far can be generated by opcode directly
    - Function code is not needed up to this point
  - A major part of the controller can be built based on opcode alone
Closer Look at ALU Unit

- The ALU Unit is a combinatorial circuit:
  - Capable of performing several arithmetic operations
- In Lecture #14:
  - We noted the required operations for the MIPS subset

**Question:**
- How is the **ALUcontrol** signal designed?

<table>
<thead>
<tr>
<th>ALUcontrol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>slt</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>
One Bit At A Time

- A simplified 1-bit MIPS ALU can be implemented as follows:

- 4 control bits are needed:
  - Ainvert:
    - 1 to invert input A
  - Binvert:
    - 1 to invert input B
  - Operation (2-bit)
    - To select one of the 3 results
One Bit At A Time (A-HA!)

- Can you see how the ALUcontrol (4-bits) signal controls the ALU?
  - Note: implementation for slt not shown

<table>
<thead>
<tr>
<th>ALUcontrol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ainvert</td>
<td>Binvert</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Acknowledgement: Image taken from NYU Course CSCI-UA.0436
Multilevel Decoding

Now we can start to design for ALU control signal, which depends on:

- Opcode (6-bit) and Function Code Field (6-bit)

**Brute Force approach:**

- Use Opcode and Function Code directly, i.e. finding expressions with 12 variables

**Multilevel Decoding approach:**

- Use some of the input to reduce the cases, then generate the full output
  + Simplify the design process, reduce the size of the main controller, potentially speedup the circuit
Intermediate Signal: \texttt{ALUop}

- **Basic Idea:**
  1. Use Opcode to generate a 2-bit \texttt{ALUop} signal
     - Represents classification of the instructions:
     
     \[
     \begin{array}{|c|c|}
     \hline
     \text{Instruction Type} & \text{ALUop} \\
     \hline
     \text{lw / sw} & 00 \\
     \text{beq} & 01 \\
     \text{R-type} & 10 \\
     \hline
     \end{array}
     \]
  2. Use \texttt{ALUop} signal and Function Code field (for R-type instructions) to generate the 4-bit \texttt{ALUcontrol} signal
2-Level Implementation

Step 1. Generate ALUop signal from 6-bit opcode.

Step 2. Generate ALUctrl signal from ALUop and optionally 6-bit Funct field.

### Control

**opcode**: 31:26

**rs**: 25:21

**rt**: 20:16

**rd**: 15:11

**shamt**: 10:6

**funct**: 5:0

**ALUop**

00: lw, sw  
01: beq  
10: add, sub, and, or, slt

**ALUcontrol**

0000: and  
0001: or  
0010: add  
0110: sub  
0111: set on less than

### ALU

### Step 1.
Generate ALUop signal from 6-bit opcode.

### Step 2.
Generate ALUctrl signal from ALUop and optionally 6-bit Funct field.
## Generating ALUControl Signal

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ALUop</th>
<th>Instruction Operation</th>
<th>Funct field</th>
<th>ALU action</th>
<th>ALU control</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td>load word</td>
<td></td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td>store word</td>
<td></td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td>branch equal</td>
<td></td>
<td>subtract</td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td></td>
<td>add</td>
<td></td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td></td>
<td>subtract</td>
<td></td>
<td>subtract</td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td></td>
<td>AND</td>
<td></td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td></td>
<td>OR</td>
<td></td>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td></td>
<td>set on less than</td>
<td></td>
<td>set on less than</td>
<td></td>
</tr>
</tbody>
</table>

**ALUcontrol Function**

<table>
<thead>
<tr>
<th>ALUcontrol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>slt</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>

Generation of **ALUop** signal will be discussed later.
Design of ALU Control Unit (1/2)

- **Input:** 6-bit *Funct* field and 2-bit *ALUop*
- **Output:** 4-bit *ALUcontrol*
- Find the simplified expressions

<table>
<thead>
<tr>
<th>ALUop</th>
<th><strong>Funct Field</strong> (F[5:0] == Inst[5:0])</th>
<th><strong>ALUcontrol</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F5</td>
<td>F4</td>
</tr>
<tr>
<td><strong>MSB</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LSB</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>slt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design of ALU Control Unit (2/2)

- Simple combinational logic

\[ \text{ALUcontrol}_2 = \text{ALUOp}_0 + \text{ALUOp}_1 \cdot F_1 \]
Finale: Control Design

- We have now considered all individual signals and their expected values
  - Ready to design the controller itself

- Typical digital design steps:
  - Fill in truth table
    - **Input**: Opcode
    - **Output**: Various control signals as discussed
  - Derive simplified expression for each signal
# Control Design: Outputs

<table>
<thead>
<tr>
<th></th>
<th>RegDest</th>
<th>ALUSrc</th>
<th>MemTo Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUop</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>op1</td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>op0</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Control Design Diagram]
Control Design: Inputs

With the input (opcode) and output (control signals), let's design the circuit

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value in Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op[5:0] == Inst[31:26]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R-type</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op5</td>
<td>Op4</td>
<td>Op3</td>
<td>Op2</td>
</tr>
<tr>
<td>Op1</td>
<td>Op0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- LW
- SW
- BEQ
Combinational Circuit Implementation

Control Logic

**Control Signals**
- RegDst
- ALUSrc
- MemtoReg
- RegWrite
- MemRead
- MemWrite
- Branch
- ALUOp1
- ALUOp0

**Opcode**
- Op5
- Op4
- Op3
- Op2
- Op1
- Op0

**Inputs**
- R-format
- lw
- sw
- beq

**Outputs**
- Control Logic
- Signals
Big Picture: Instruction Execution

- Instruction Execution =
  1. Read contents of one or more storage elements (register/memory)
  2. Perform computation through some combinational logic
  3. Write results to one or more storage elements (register/memory)

- All these performed **within a clock period**

Don’t want to read a storage element when it is being written
Single Cycle Implementation: Shortcoming

- Calculate cycle time assuming negligible delays except: memory (2ns), ALU/adders (2ns), register file access (1ns)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Inst Mem</th>
<th>Reg read</th>
<th>ALU</th>
<th>Data Mem</th>
<th>Reg write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>lw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>sw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>beq</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

- All instructions take as much time as the slowest one (i.e., load)

→ Long cycle time
Solution 1: Multicycle Implementation

- Break up the instructions into execution steps:
  1. Instruction fetch
  2. Instruction decode and register read
  3. ALU operation
  4. Memory read/write
  5. Register write

- Each execution **step takes one clock cycle**
  - Cycle time is much shorter, i.e., clock frequency is much higher

- Instructions take variable number of clock cycles to complete execution

- Not covered in class:
  - See Section 5.5 if interested
Solution 2: Pipelining

- Break up the instructions into execution steps one per clock cycle

- Allow different instructions to be in different execution steps simultaneously

- Covered in lecture set #16
Summary

- A very simple implementation of MIPS datapath and control for a subset of its instructions

Concepts:

- An instruction executes in a single clock cycle
- Read storage elements, compute, write to storage elements
- Datapath is shared among different instructions types using MUXs and control signals
- Control signals are generated from the machine language encoding of instructions
Reading Assignments

- **The Processor: Datapath and Control**
  - 3rd edition: Chapter 5 Section 5.4
  - 4th edition: Chapter 4 Section 4.4

- **Exploration:**
  - ALU design and implementation:
    - 4th edition (MIPS): Appendix C
    - [http://cs.nyu.edu/courses/fall11/CSCI-UA.0436-001/class-notes.html](http://cs.nyu.edu/courses/fall11/CSCI-UA.0436-001/class-notes.html)
End