The Processor: Datapath and Control

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Introduction

- **CPU performance factors**
  - Instruction count – Determined by ISA and compiler
  - CPI and cycle time – Determined by CPU hardware

- **We will examine two MIPS implementations**
  - A simplified version
  - A more realistic pipelined version

- **Simple subset, shows most aspects**
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`
Logic Design Basics

- **Information encoded in binary**
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses

- **Combinational elements**
  - Operate on data
  - Output is a function of input

- **State (sequential) elements**
  - Store information
Combinational Elements

- **AND-gate**
  - Y = A & B
  - ![AND-gate diagram]

- **Adder**
  - Y = A + B
  - ![Adder diagram]

- **Multiplexer**
  - Y = S? I1 : I0
  - ![Multiplexer diagram]

- **Arithmetic/Logic Unit**
  - Y = F (A, B)
  - ![ALU diagram]
Sequential Elements (1)

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

![D flip-flop diagram]

![Timing diagram for register]

Clk: 1 0 1
D: 1 0 1
Q: 1 0 1
Sequential Elements (2)

- **Register with write control**
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

![Diagram of register with write control](image-url)
Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period
Instruction Execution

- **Fetch instruction**
  - PC $\rightarrow$ instruction memory

- **Read registers**
  - Register numbers $\rightarrow$ register file

- **Depending on instruction class**
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC $\leftarrow$ target address or PC + 4
CPU Overview
Multiplexers

Can’t just join wires together → Use multiplexers
Control

[Diagram of computer architecture showing control flow with components including PC, Mux, Add, Registers, ALU, MemRead, MemWrite, Data memory, and Control.]
Building a Datapath

- **Datapath**
  - Elements that process data and addresses in the CPU
  - Registers, ALUs, MUX’s, memories, ...

- **We will build a MIPS datapath incrementally**
  - Refining the overview design
Instruction Fetch

- 32-bit register
- Increment by 4 for next instruction
- Instruction memory
- Instruction
- Add
- 4
- PC
- Read address
R-Format Instructions

- **Operations**
  - Read two register operands
  - Perform arithmetic/logical operation
  - Write register result

![Diagram of R-Format Instructions](image)
**Load/Store Instructions**

- **Operations**
  - Read register operand
  - Calculate address using 16-bit offset
    - Use ALU, but sign-extend offset
  - Load: Read memory and update register
  - Store: Write register value to memory

![Diagram of Load/Store Instructions](image)
Branch Instructions (1)

- **Operations**
  - Read register operands
  - Compare operands
    - Use ALU, subtract and check Zero output
  - Calculate target address
    - Sign-extend displacement
    - Shift left 2 places (word displacement)
    - Add to PC + 4
      (Already calculated by instruction fetch)
Branch Instructions (2)

Just re-routes wires

PC + 4 from instruction datapath

Instruction

Read register 1
Read register 2
Write register
Write data

Registers

Read data 1
Read data 2

Shift left 2

Add Sum
Branch target

ALU Zero
To branch control logic

4 ALU operation

Sign-extend

16 32

Sign-bit wire replicated
Composing the Elements

- **Building a simple datapath**
  - Executes all instructions in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
  - Use multiplexers where alternate data sources are used for different instructions
R-Type/Load/Store Datapath

Diagram showing the R-Type/Load/Store Datapath with components such as Instruction, Registers, ALU, ALU operation, Zero, ALU result, Address, Data memory, Write data, MemRead, MemWrite, MemtoReg, RegWrite, Read data, Read data 1, Read data 2, ALUSrc, and Sign-extend.
Full Datapath
The Main Control Unit

- **Control signals derived from instruction**

<table>
<thead>
<tr>
<th>Instruction Type</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>R-type</strong></td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct 5:0</td>
</tr>
<tr>
<td><strong>Load/Store</strong></td>
<td>4</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>35:26</td>
<td>25:21</td>
<td>20:16</td>
<td></td>
<td>15:0</td>
<td></td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td>35:26</td>
<td>25:21</td>
<td>20:16</td>
<td></td>
<td>15:0</td>
<td></td>
</tr>
</tbody>
</table>

- **opcode**
- **always read**
- **read, except for load**
- **write for R-type and load**
- **sign-extend and add**
ALU Control (1)

- ALU used for
  - Load/Store: \( F = \text{add} \)
  - Branch: \( F = \text{subtract} \)
  - R-type: \( F \) depends on funct field

<table>
<thead>
<tr>
<th>ALU control</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>set-on-less-than</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>
### ALU Control (2)

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

<table>
<thead>
<tr>
<th>opcode</th>
<th>ALUOp</th>
<th>Operation</th>
<th>funct</th>
<th>ALU function</th>
<th>ALU control</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>00</td>
<td>load word</td>
<td>XXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>sw</td>
<td>00</td>
<td>store word</td>
<td>XXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>branch equal</td>
<td>XXXXXX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AND</td>
<td>100100</td>
<td>AND</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
<td>100101</td>
<td>OR</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set-on-less-than</td>
<td>101010</td>
<td>set-on-less-than</td>
<td>0111</td>
</tr>
</tbody>
</table>
R-Type Instruction
Load Instruction
Branch-on-Equal Instruction
Implementing Jumps

- Jump instruction

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode
Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file

- Not feasible to vary period for different instructions

- Violates design principle
  - Making the common case fast

- We will improve performance by pipelining