Designing a Single Cycle Datapath

or

The Do-It-Yourself CPU Kit

The Big Picture: The Performance Perspective

- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Starting today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time

\[ ET = \text{Insts} \times \text{CPI} \times \text{Cycle Time} \]

The Processor: Datapath & Control

- We're ready to look at an implementation of the MIPS simplified to contain only:
  - memory-reference instructions: lw, sw
  - arithmetic-logical instructions: add, sub, and, or, slt
  - control flow instructions: beq
- Generic Implementation:
  - use the to supply instruction address
  - get the from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers
  memory-reference? arithmetic? control flow?

Review: The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shamt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I-type

<table>
<thead>
<tr>
<th>op</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

J-type

<table>
<thead>
<tr>
<th>op</th>
<th>target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>26</td>
</tr>
</tbody>
</table>
The MIPS Subset

- **R-type**
  - `add rd, rs, rt`
  - `sub, and, or, slt`

- **LOAD and STORE**
  - `lw rt, rs, imm16`
  - `sw rt, rs, imm16`

- **BRANCH:**
  - `beq rs, rt, imm16`

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td><strong>op</strong></td>
<td><strong>rs</strong></td>
<td><strong>rt</strong></td>
<td><strong>rd</strong></td>
<td><strong>shamt</strong></td>
<td><strong>funct</strong></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>op</strong></td>
<td><strong>rs</strong></td>
<td><strong>rt</strong></td>
<td><strong>immediate</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>op</strong></td>
<td><strong>rs</strong></td>
<td><strong>rt</strong></td>
<td><strong>displacement</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where We’re Going – The High-level View

Clocking Methodology

Storage Element: Register

- Register
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - Write Enable:
    - 0: Data Out will not change
    - 1: Data Out will become Data In (on the clock edge)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Clk</strong></td>
<td><strong>Setup</strong></td>
<td><strong>Hold</strong></td>
<td><strong>Don’t Care</strong></td>
<td><strong>Setup</strong></td>
<td><strong>Hold</strong></td>
</tr>
</tbody>
</table>

- All storage elements are clocked by the same clock edge
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses:
  - One 32-bit input bus: busW
- Register is selected by:
  - RR1 selects the register to put on bus “Read Data 1”
  - RR2 selects the register to put on bus “Read Data 2”
  - WR selects the register to be written via WriteData when RegWrite is 1
- Clock input (CLK)

Register Transfer Language (RTL)

- is a mechanism for describing the movement and manipulation of data between storage elements:

\[
\begin{align*}
PC & \leftarrow PC + 4 + R[5] \\
R[rd] & \leftarrow R[rs] + R[rt] \\
R[rt] & \leftarrow Mem[R[rs] + \text{immed}] 
\end{align*}
\]

Instruction Fetch and Program Counter Management

Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData
- Memory word is selected by:
  - Address selects the word to put on ReadData bus
  - MemWrite = 1: address selects the memory word to be written via the WriteData bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => ReadData valid after “access time.”
Overview of the Instruction Fetch Unit

- The common RTL operations
  - Fetch the Instruction: \( \text{inst} \leftarrow \text{mem}[\text{PC}] \)
  - Update the program counter:
    - Sequential Code: \( \text{PC} \leftarrow \text{PC} + 4 \)
    - Branch and Jump: \( \text{PC} \leftarrow \text{“something else”} \)

Datapath for Register-Register Operations

- \( R[rd] \leftarrow R[rs] \text{ op } R[rt] \)
  - Example: \( \text{add rd, rs, rt} \)
  - RR1, RR2, and WR come from instruction’s rs, rt, and rd fields
  - \( \text{ALU operation and RegWrite} \): control logic after decoding instruction

Datapath for Load Operations

- \( R[rt] \leftarrow \text{Mem[R[rs] + SignExt[imm16]]} \)
  - Example: \( \text{l w rt, rs, imm16} \)

Datapath for Store Operations

- \( \text{Mem[R[rs] + SignExt[imm16]]} \leftarrow R[rt] \)
  - Example: \( \text{s w rt, rs, imm16} \)
Datapath for Branch Operations

Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4

```
beq rs, rt, imm16
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Binary Arithmetic for the Next Address

- In theory, the PC is a 32-bit byte address into the instruction memory:
  - operation: PC<31:0> = PC<31:0> + 4
  - operation: PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] * 4
- The magic number “4” always comes up because:
  - The 32-bit PC is a byte address
  - And all our instructions are 4 bytes (32 bits) long
  - The 2 LSBs of the 32-bit PC are always zeros
  - There is no reason to have hardware to keep the 2 LSBs
- In practice, we can simplify the hardware by using a 30-bit PC<31:2>:
  - Sequential operation: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - In either case: Instruction Memory Address = PC<31:2> concat “00”

Putting it All Together: A Single Cycle Datapath

- We have everything except control signals

The R-Format (e.g. add) Datapath
CSE 141 Dean Tullsen

The Load Datapath

MemtoReg
MemRead
MemWrite
ALUOp
ALUSrc
RegDst
PC
Instruction
memory
Read
address
Instruction
[31–0]
Instruction
[20–16]
Instruction
[25–21]
Add
Instruction
[5–0]
RegWrite
4
16
32
Instruction
[15–0]
Registers
Write
register
Write
data
Read
data
Read
register
1
Read
register
2
ALU
result
ALU
control
Shift
left
2
PCSrc
ALU
Add
ALU
result
Zero
Data
memory
Address
Read
data
Mux
Mux
Mux
Mux
Mux

CSE 141 Dean Tullsen

The store Datapath

MemtoReg
MemRead
MemWrite
ALUOp
ALUSrc
RegDst
PC
Instruction
memory
Read
address
Instruction
[31–0]
Instruction
[20–16]
Instruction
[25–21]
Add
Instruction
[5–0]
RegWrite
4
16
32
Instruction
[15–0]
Registers
Write
register
Write
data
Read
data
Read
register
1
Read
register
2
ALU
result
ALU
control
Shift
left
2
PCSrc
ALU
Add
ALU
result
Zero
Data
memory
Address
Read
data
Mux
Mux
Mux
Mux
Mux

CSE 141 Dean Tullsen

The beq Datapath

MemtoReg
MemRead
MemWrite
ALUOp
ALUSrc
RegDst
PC
Instruction
memory
Read
address
Instruction
[31–0]
Instruction
[20–16]
Instruction
[25–21]
Add
Instruction
[5–0]
RegWrite
4
16
32
Instruction
[15–0]
Registers
Write
register
Write
data
Read
data
Read
register
1
Read
register
2
ALU
result
ALU
control
Shift
left
2
PCSrc
ALU
Add
ALU
result
Zero
Data
memory
Address
Read
data
Mux
Mux
Mux
Mux
Mux

CSE 141 Dean Tullsen

Key Points

• CPU is just a collection of state and combinational logic
• We just designed a very rich processor, at least in terms of functionality
• ET = IC * CPI * Cycle Time
  – where does the single-cycle machine fit in?