CPS104
Computer Organization and programming
Lecture 13: Designing a Single Cycle Datapath

Oct 18, 1999

Dietolf (Dee) Ramm

http://www.cs.duke.edu/~dr/cps104.html
Outline of Today’s Lecture

- Where are we with respect to the BIG picture?
- The Steps of Designing a Processor
- Datapath components.
- Datapath and timing for Reg-Reg Operations
- Datapath for Logical Operations with Immediate
- Datapath for Load and Store Operations
- Datapath for Branch and Jump Operations
What is Computer Architecture?

- Coordination of levels of abstraction

Software

Interface Between HW and SW

Instruction Set Architecture, Memory, I/O

Hardware

Application

Operating System

Compiler

Firmware

CPU  Memory  I/O system

Digital Design

Circuit Design
The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

Today’s Topic: Datapath Design
# The MIPS Instruction Formats

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

<table>
<thead>
<tr>
<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>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

- **R-type**

- **I-type**

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

- **J-type**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
</tr>
</tbody>
</table>

- **The different fields are:**
  - **op**: operation of the instruction
  - **rs, rt, rd**: the source and destination register specifiers
  - **shamt**: shift amount
  - **funct**: selects the variant of the operation in the “op” field
  - **address / immediate**: address offset or immediate value
  - **target address**: target address of the jump instruction
# The MIPS Subset (We can’t do them all!)

## ADD and subtract
- **ADD**: $\text{add } rd, rs, rt$
- **SUB**: $\text{sub } rd, rs, rt$

## OR Immediate:
- **ORI**: $\text{ori } rt, rs, \text{imm16}$

## LOAD and STORE
- **LW**: $\text{lw } rt, rs, \text{imm16}$
- **SW**: $\text{sw } rt, rs, \text{imm16}$

## BRANCH:
- **BEQ**: $\text{beq } rs, rt, \text{imm16}$

## JUMP:
- **J**: $\text{j } \text{target}$

<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>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<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>

<table>
<thead>
<tr>
<th>op</th>
<th>target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
The Hardware “Program”

How does one build hardware that implements the MIPS instructions?
An Abstract View of the Implementation

Clk

PC

Instruction Address

Ideal Instruction Memory

Instruction

Rd 5
Rs 5
Rt 5
Imm 16

Rw Ra Rb

32 32-bit Registers

A

32

B

Data Address

Data Out

Data In

Clk

©GK&DR Fall 1999
An Abstract View of the Critical Path

- Register file and ideal memory:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after “access time.”

![Diagram of the processor components](image-url)
The Steps of Designing a Processor

° Instruction Set Architecture => Register Transfer Language

° Register Transfer Language =>
  • Datapath components
  • Datapath interconnect

° Datapath components => Control signals

° Control signals => Control logic
RTL: The ADD Instruction

° add rd, rs, rt

• mem[PC] Fetch the instruction from memory

• R[rd] <- R[rs] + R[rt] The ADD operation

• PC <- PC + 4 Calculate the next instruction’s address
RTL: The Load Instruction

\[ \text{l}w \quad \text{rt, rs, imm16} \]

- \( \text{mem[PC]} \) Fetch the instruction from memory
- \( \text{Addr} \leftarrow \text{R[rs]} + \text{SignExt(imm16)} \) Calculate the memory address
- \( \text{R[rt]} \leftarrow \text{Mem[Addr]} \) Load the data into the register
- \( \text{PC} \leftarrow \text{PC} + 4 \) Calculate the next instruction’s address
Combinational Logic Elements (Basic Building Blocks)

- **Adder**

- **MUX**

- **ALU**
**Master-Slave Data-Flip-Flop**

- On **C** ↓, D is transferred to the master stage and the slave is stable.
- On **C** ↑, the Master stage is transferred into the slave stage (output), and the master stage is stable.
DFF with Enable

Master

Data

Clock

Enable

Q

Q
Storage Element: Register (Basic Building Block)

- **Register**
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - Write Enable:
    - negated (0): Data Out will not change
    - asserted (1): Data Out will become the same as Data In.
Clocking Methodology

- All storage elements are clocked by the same clock edge
- Cycle Time $\geq$ CLK-to-Q + Longest Delay Path + Setup + Clock Skew
Storage Element: Register File

- Register File consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW

- Register is selected by:
  - RA selects the register to put on busA
  - RB selects the register to put on busB
  - RW selects the register to be written via busW when Write Enable is 1

- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after “access time.”
Storage Element: Idealized Memory

- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out

- Memory word is selected by:
  - Write Enable = 0: Address selects the word to put on the Data Out bus
  - Write Enable = 1: Address selects the memory word to be written via the Data In 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 => Data Out valid after “access time.”
Overview of the Instruction Fetch Unit

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

- **add**  \( \text{rd, rs, rt} \)

<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><strong>op</strong></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
<tr>
<td><strong>rs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rd</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>shamt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>funct</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **mem[PC]**
  - Fetch the instruction from memory
- **R[rd] <- R[rs] + R[rt]**
  - The actual operation
- **PC <- PC + 4**
  - Calculate the next instruction’s address
RTL: The Subtract Instruction

<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>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

° sub  rd, rs, rt

- **mem[PC]**
  - Fetch the instruction from memory

- **R[rd] <- R[rs] - R[rt]**
  - The actual operation

- **PC <- PC + 4**
  - Calculate the next instruction's address
Datapath for Register-Register Operations

° R[rd] <- R[rs] op R[rt]
  • Ra, Rb, and Rw comes from instruction’s rs, rt, and rd fields
  • ALUctr and RegWr: control logic after decoding the instruction fields: op and funct

```
<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>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>
```

Example: add rd, rs, rt
Register-Register Timing

- **Clk**
  - Clk-to-Q
- **PC**
  - Instruction Memory Access Time
  - Old Value
  - New Value
- **Rs, Rt, Rd, Op, Func**
  - Delay through Control Logic
  - Old Value
  - New Value
- **ALUctr**
  - Old Value
  - Register Write Occurs Here
- **RegWr**
  - Register File Access Time
  - Old Value
  - New Value
- **busA, B**
  - ALU Delay
  - Old Value
  - New Value
- **busW**
  - Old Value
  - New Value

- **ALU**
  - 32 32-bit Registers

- **Result**
  - 32

**Register Write**

**Instruction Memory Access Time**

**Delay through Control Logic**

**Register File Access Time**

**ALU Delay**

**Register Write Occurs Here**

**32 32-bit Registers**

**Inst fetch**

**Decode**

**Opr. fetch**

**Execute**

**Write Back**

**Rw**

**Ra**

**Rb**

**busA**

**busB**

**busW**

**Clk**

©GK&DR Fall 1999
RTL: The OR Immediate Instruction

- ori rt, rs, imm16

  - mem[PC] Fetch the instruction from memory
  - R[rt] <- R[rs] or ZeroExt(imm16) The OR operation
  - PC <- PC + 4 Calculate the next instruction’s address
Datapath for Logical Operations with Immediate

\[ R[rt] \leftarrow R[rs] \text{ op ZeroExt}[\text{imm16}] \]

Example: ori \( rt, rs, \text{imm16} \)

<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>

- \( R^r_t \leftarrow R^r_s \text{ op ZeroExt}^{imm16} \)
RTL: The Load Instruction

- **lw**  
  rt, rs, imm16

- **mem[PC]**  
  Fetch the instruction from memory

- **Addr <- R[rs] + SignExt(imm16)**  
  Calculate the memory address

- **R[rt] <- Mem[Addr]**  
  Load the data into the register

- **PC <- PC + 4**  
  Calculate the next instruction’s address
Datapath for Load Operations

\[ R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[imm16]] \]

Example: \textit{lw} \hspace{1em} rt, rs, imm16

---

**Functional Unit Diagram:**

- **32 32-bit Registers**
- **Extender**
- **ALU**
- **Mem**

---

**Instruction Format:**

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

---

**Control Signals:**

- **RegDst**
- **Rd**
- **Rt**
- **RegWr**
- **Don’t Care**
- **(Rt)**
- **ALUctr**
- **MemtoReg**

---

**Data Path:**

- **busW**
- **Clk**
- **imm16**
- **ExtOp**
- **ALUSrc**
- **Data In**
- **WrEn**
- **Adr**
- **MemWr**

---

**Notes:**

- ALU input sources: \( \text{ExtOp} \) and \( \text{ALUSrc} \)
- Memory input: \( \text{Data In} \)
- Memory output: \( \text{MemtoReg} \)
**RTL: The Store Instruction**

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

° `sw rt, rs, imm16`

- `mem[PC]` Fetch the instruction from memory
- `Addr <- R[rs] + SignExt(imm16)` Calculate the memory address
- `Mem[Addr] <- R[rt]` Store the register into memory
- `PC <- PC + 4` Calculate the next instruction’s address
Datapath for Store Operations

\[ \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}] \leftarrow R[rt]] \quad \text{Example: sw rt, rs, imm16} \]
RTL: The Branch Instruction

° beq rs, rt, imm16

• mem[PC] Fetch the instruction from memory

• Cond <- R[rs] - R[rt] Calculate the branch condition

• if (COND eq 0) Calculate the next instruction’s address
  - PC <- PC + 4 + ( SignExt(imm16) x 4 )

• else
  - PC <- PC + 4
Datapath for Branch Operations

° beq rs, rt, imm16

We need to compare Rs and Rt!
Binary Arithmetic for the Next Address

° In theory, the PC is a 32-bit byte address into the instruction memory:
  • Sequential operation: $PC^{31:0} = PC^{31:0} + 4$
  • Branch operation: $PC^{31:0} = PC^{31:0} + 4 + \text{SignExt}[\text{Imm16}] \times 4$

° 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

° In other words:
  • 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: $PC^{31:2} = PC^{31:2} + 1 + \text{SignExt}[\text{Imm16}]$
  • In either case: Instruction-Memory-Address = $PC^{31:2}$ concat “00”
Next Address Logic: Expensive and Fast Solution

° Using a 30-bit PC:
  • Sequential operation: $PC_{31:2} = PC_{31:2} + 1$
  • Branch operation: $PC_{31:2} = PC_{31:2} + 1 + \text{SignExt}[\text{Imm16}]$
  • In either case: Instruction-Memory-Address = $PC_{31:2}$ concat “00”
Next Address Logic

Diagram:
- **PC** (Program Counter) feeds into a 30-bit signal.
- **Clk** (Clock) input.
- **imm16** (Immediate 16-bit) input.
- **Instruction<15:0>** (Instruction bits 15 to 0) input.
- **SignExt** (Sign Extension) input.
- **Branch Zero** output.
- **Mux** (Multiplexer) with inputs 0 and 1, controlled by **Branch Zero**.
- **Adder** with **Carry In** input.
- **Addr<31:2>** (Address bits 31 to 2) output.
- **Addr<1:0>** (Address bits 1 to 0) output.
- **Instruction Memory** with outputs **Addr<31:2>** and **Addr<1:0>**.
- **Instruction<31:0>** (Instruction bits 31 to 0) output.
RTL: The Jump Instruction

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
</tr>
</tbody>
</table>
```

° j target

- **mem[PC]**
  - Fetch the instruction from memory

- **PC<31:2> ← PC<31:28> concat target<26:0> concat <00>**
  - Calculate the next instruction’s address
Instruction Fetch Unit

- $j \rightarrow \text{target}$
  - $\text{PC}<31:2> \leftarrow \text{PC}<31:28> \text{ concat target}<25:0>$

Diagram:
- $\text{PC}<31:28>$
- $\text{Target}$
- $\text{Instruction}<25:0>$
- $\text{Jump}$
- $\text{Instruction}<31:0>$
- $\text{PC}<31:2>$
- $\text{Addr}<1:0>$
Putting it All Together: A Single Cycle Datapath

We have everything except control signals.