Reconfigurable Computing

Single Cycle RISC Processor

Indeed. RISC architecture is gonna change everything
- Angelina Jolie as Kate in Hackers

Philip Leong (philip.leong@sydney.edu.au)
School of Electrical and Information Engineering
http://www.ee.usyd.edu.au/~phwl

lecture slides adapted from Patterson and Hennesy,
“Computer Organization and Design”: http://inst.eecs.berkeley.edu/~cs152/
Discuss the design of a reduced instruction set (RISC) processor for the MIPS instruction set

The design methodology for any field-programmable custom computing machine (FCCM) is similar
- Those without programmability are a special case of processors

The are several different design methodologies, we cover single cycle here

Slides come from an excellent book, Patterson and Hennessy, “Computer Organization and Design: The Hardware/Software Interface”
The Big Picture: Where are We Now?

° The Five Classic Components of a Computer

° Today’s Topic: Design a Single Cycle Processor

inst. set design (L1-2)  machine design  Arithmetic (L4-6)

technology (L3)
The Big Picture: The Performance Perspective

- Performance of a machine is determined by:
  - Instruction count
  - Clock cycle time
  - Clock cycles per instruction

- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction

- Today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time
How to Design a Processor: step-by-step

1. Analyze instruction set => datapath requirements
   - the meaning of each instruction is given by the register transfers
   - datapath must include storage element for ISA registers
     - possibly more
   - datapath must support each register transfer

2. Select set of datapath components and establish clocking methodology

3. Assemble datapath meeting the requirements

4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.

5. Assemble the control logic
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>R-type</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
<tr>
<td></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>I-type</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-type</td>
<td>op</td>
<td>target address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
<td></td>
<td></td>
<td></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
Step 1a: The MIPS-lite Subset for today

° ADD and SUB
  • addU rd, rs, rt
  • subU rd, rs, rt

° OR Immediate:
  • ori rt, rs, imm16

° LOAD and STORE Word
  • lw rt, rs, imm16
  • sw rt, rs, imm16

° BRANCH:
  • beq rs, rt, imm16
Logical Register Transfers

° RTL gives the meaning of the instructions

° All start by fetching the instruction

\[ \text{op} \mid \text{rs} \mid \text{rt} \mid \text{rd} \mid \text{shamt} \mid \text{funct} = \text{MEM}[\ PC \ ] \]

\[ \text{op} \mid \text{rs} \mid \text{rt} \mid \text{Imm16} = \text{MEM}[\ PC \ ] \]

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU</td>
<td>( \text{R[rd]} \leftarrow \text{R[rs]} + \text{R[rt]}; ) PC &lt;- PC + 4</td>
</tr>
<tr>
<td>SUBU</td>
<td>( \text{R[rd]} \leftarrow \text{R[rs]} - \text{R[rt]}; ) PC &lt;- PC + 4</td>
</tr>
<tr>
<td>ORi</td>
<td>( \text{R[rt]} \leftarrow \text{R[rs]} \mid \text{zero_ext(Imm16)}; ) PC &lt;- PC + 4</td>
</tr>
<tr>
<td>LOAD</td>
<td>( \text{R[rt]} \leftarrow \text{MEM}[\text{R[rs]} + \text{sign_ext(Imm16)}]; ) PC &lt;- PC + 4</td>
</tr>
<tr>
<td>STORE</td>
<td>( \text{MEM}[\text{R[rs]} + \text{sign_ext(Imm16)}] \leftarrow \text{R[rt]}; ) PC &lt;- PC + 4</td>
</tr>
<tr>
<td>BEQ</td>
<td>if ( \text{R[rs]} == \text{R[rt]} ) then PC &lt;- PC + 4 + ( \text{sign_ext(Imm16)} \mid \mid 00 ) \else PC &lt;- PC + 4</td>
</tr>
</tbody>
</table>
Step 1: Requirements of the Instruction Set

° Memory
  • instruction & data

° Registers (32 x 32)
  • read RS
  • read RT
  • Write RT or RD

° PC

° Extender

° Add and Sub register or extended immediate

° Add 4 or extended immediate to PC
Step 2: Components of the Datapath

° Combinational Elements

° Storage Elements
  • Clocking methodology
Combinational Logic Elements (Basic Building Blocks)

° Adder

° MUX

° ALU
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 Data In
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 (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) 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:
  • Address selects the word to put on Data Out
  • 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.”
Recall: Clocking Methodology

° All storage elements are clocked by the same clock edge

° **Cycle Time** = CLK-to-Q + Longest Delay Path + Setup + Clock Skew
Step 3: Assemble DataPath meeting our requirements

° Register Transfer **Requirements**  
  ⇒ Datapath **Assembly**

° Instruction Fetch

° Read Operands and Execute Operation
3a: Overview of the Instruction Fetch Unit

° The common RTL operations
  • Fetch the Instruction: mem[PC]
  • Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- “something else”
3b: Add & Subtract

° \( R[rd] \leftarrow R[rs] \text{ op } R[rt] \)

- \( Ra, Rb, \text{ and } Rw \) come from instruction’s \( rs, rt, \text{ and } rd \) fields
- ALUctr and RegWr: control logic after decoding the instruction

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>
</tr>
</tbody>
</table>

```
Rd  Rs  Rt
5    5    5
```

```
ALUctr
```

```
RegWr
```

```
busW
```

```
busA
```

```
busB
```

```
Result
```

```
Clk
```

```
32  32-bit Registers
```

Example: \( \text{addU } \) rd, rs, rt
Register/Register Timing: One complete cycle

- **Clk:** Clock signal.
- **PC:** Program Counter.
- **Rs, Rt, Rd, Op, Func:** Register file access.
- **ALUctr:** ALU control register.
- **RegWr:** Register file write.
- **busA, B:** Bus signals.
- **busW:** Bus write signal.

**Delay Components:**
- **Clk-to-Q:** Clock-to-Q delay.
- **Instruction Memory Access Time:** Time to access instruction memory.
- **Delay through Control Logic:** Delay through control logic.
- **Register File Access Time:** Time to access register file.
- **ALU Delay:** Delay through ALU.
- **Register Write:** Time for register write operation.

**Register File:**
- **32 32-bit Registers:**
- **Rw, Ra, Rb:** Register inputs.

**ALU:**
- **32:** ALU 32-bit output.

**Result:** Output of ALU.

**Register Write Occurs Here:**

©UCB Spring 2003
### 3c: Logical Operations with Immediate

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

<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>

\[ \text{rd?} \]

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### ALU Circuit Diagram

- **RegDst**
- **Rd**
- **Rt**
- **Mux**
- **RegWr**
- **Rs**
- **Rt?**
- **busA**
- **ALUctr**
- **Result**

### Register Block

- **Rw**
- **Ra**
- **Rb**
- **32 32-bit Registers**

### ALU Inputs

- **busB**
- **imm16**
- **ZeroExt**
- **ALUSrc**

### ALU Outputs

- **busW**
- **Clk**
3d: Load Operations

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

**Example: lw rt, rs, 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>
```

**Diagram:**
- Registers
- ALU
- Memory
- Muxes
- ALU control
- Data input
- WrEn
- Addr
- Data memory
- Extender
- ExtOp

2/19/03

©UCB Spring 2003

CS152 / Kubiatowicz
Lec7.22

Example: sw rt, rs, 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>

3e: Store Operations
3f: The Branch Instruction

beq rs, rt, imm16

- mem[PC]  Fetch the instruction from memory
- Equal <- R[rs] == R[rt]  Calculate the branch condition
- if (Equal)  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

Datapath generates condition (equal)

<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>

Inst Address

Equal?

Cond

PC Ext

imm16

4

Adder

nPC_sel

Mux

Clk

RegWr

Rs

Rt

busA

busB

32 32-bit Registers

Rw Ra Rb

32

PC

32

busW

Clk

©UCB Spring 2003

CS152 / Kubiatowicz
Lec7.25
Putting it all together

- Rs, Rt, Rd and Imed16 hardwired into datapath from Fetch Unit
- We have everything except control signals (underline)
  - Today’s lecture will show you how to generate the control signals
Recap: Meaning of the Control Signals

° nPC_MUX_sel:  
  0 ⇒ PC ← PC + 4  
  1 ⇒ PC ← PC + 4 + SignExt(Im16) || 00

° Later in lecture: higher-level connection between mux and branch cond
Recap: Meaning of the Control Signals

- **ExtOp**: “zero”, “sign”
- **ALUsrc**: 0 ⇒ regB; 1 ⇒ immed
- **ALUctr**: “add”, “sub”, “or”
- **MemWr**: 1 ⇒ write memory
- **MemtoReg**: 0 ⇒ ALU; 1 ⇒ Mem
- **RegDst**: 0 ⇒ “rt”; 1 ⇒ “rd”
- **RegWr**: 1 ⇒ write register
Step 4: An Abstract View of the Implementation

Datapath

Control

Instruction Address

Ideal Instruction Memory

32 32-bit Registers

Instruction

Control Signals

Conditions

Data Out

Data Memory

Next Address

PC

Data In

32 bit

ALU

A

32

B

32

Data Address

Clk

Rw  Ra  Rb

5  5  5

Rd Rs Rt

32 32 32
RTL: The **Add** Instruction

<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>

° *add 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
Instruction Fetch Unit at the Beginning of Add

- Fetch the instruction from Instruction memory: Instruction <- mem[PC]
  - This is the same for all instructions
The Single Cycle Datapath during \texttt{Add}

\[ R[rd] \leftarrow R[rs] + R[rt] \]

- \texttt{RegDst} = 1
- \texttt{RegWr} = 1
- \texttt{busW} = 32
- \texttt{Clk} = 32
- \texttt{imm16} = 16
- \texttt{ExtOp} = x
- \texttt{nPC\_sel} = +4
- \texttt{ALUctr} = \texttt{Add}
- \texttt{ALUSrc} = 0
- \texttt{MemtoReg} = 0
- \texttt{Zero} = 0
- \texttt{MemWr} = 0
- \texttt{Instruction<31:0>}
- \texttt{Rt} \to \texttt{Rs} \to \texttt{Rd} \to \texttt{Imm16}
Instruction Fetch Unit at the End of Add

° PC <- PC + 4
  • This is the same for all instructions except: Branch and Jump
° R[rt] <- R[rs] or ZeroExt[Imm16]

The Single Cycle Datapath during Or Immediate

<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>
</tbody>
</table>

° R[rt] <- R[rs] or ZeroExt[Imm16]
The Single Cycle Datapath during Or Immediate

\[ R[rt] \leftarrow R[rs] \text{ or } \text{ZeroExt}[\text{Imm16}] \]
The Single Cycle Datapath during Load

- **R[rt] <- Data Memory \{R[rs] + SignExt[imm16]\}**

![Diagram of the Single Cycle Datapath during Load](image-url)
The Single Cycle Datapath during Store

° Data Memory \{R[rs] + SignExt[imm16]\} \leftarrow R[rt]

\[
\begin{array}{c|cccc|c}
31 & 26 & 21 & 16 & \text{immediate} \\
\hline
\text{op} & \text{rs} & \text{rt} & & \\
\end{array}
\]

©UCB Spring 2003
The Single Cycle Datapath during Store

- Data Memory \{R[rs] + \text{SignExt}[imm16]\} \leftarrow R[rt]
The Single Cycle Datapath during Branch

- if (R[rs] - R[rt] == 0) then Zero <- 1; else Zero <- 0

RegDst = x
RegWr = 0

ALUctr = Sub

nPC_sel = "Br"

Instruction Fetch Unit

Instruction<31:0>

MemWr = 0
MemtoReg = x

ALUSrc = 0
ExtOp = x

Extender

Data Memory

WrEn Adr

nPC = nPC + 1 (Branch)

if (R[rs] - R[rt] == 0) then Zero <- 1; else Zero <- 0
Instruction Fetch Unit at the End of Branch

\[
\text{if (Zero == 1) then } \text{PC} = \text{PC} + 4 + \text{SignExt[imm16]} \times 4 \; \text{; else } \text{PC} = \text{PC} + 4
\]

- What is encoding of nPC_sel?
  - Direct MUX select?
  - Branch / not branch

- Let’s choose second option

<table>
<thead>
<tr>
<th>nPC_sel</th>
<th>zero?</th>
<th>MUX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2/19/03
A Summary of Control Signals: The Decode Process

This is the process of “decoding” instructions!

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R[rd] ← R[rs] + R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>SUB</td>
<td>R[rd] ← R[rs] – R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = RegB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>ORi</td>
<td>R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = Im, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>LOAD</td>
<td>R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>STORE</td>
<td>MEM[R[rs] + sign_ext(Imm16)] ← R[rs]; PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td>BEQ</td>
<td>if ( R[rs] == R[rt] ) then PC ← PC + sign_ext(Imm16)</td>
</tr>
<tr>
<td></td>
<td>nPC_sel = “Br”, ALUctr = “sub”</td>
</tr>
</tbody>
</table>
Step 5: Assemble Control

° This is a hardware implementation issue
  • We will complete design next lecture
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.”

Critical Path (Load Operation) =
PC’s Clk-to-Q +
Instruction Memory’s Access Time +
Register File’s Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew
Worst Case Timing (Load)

- **Clk**
  - Clk-to-Q

- **PC**
  - Old Value
  - New Value

- **Rs, Rt, Rd, Op, Func**
  - Old Value
  - New Value

- **ALUctr**
  - Old Value
  - New Value

- **ExtOp**
  - Old Value
  - New Value

- **ALUSrc**
  - Old Value
  - New Value

- **MemtoReg**
  - Old Value
  - New Value

- **RegWr**
  - Old Value
  - New Value

- **busA**
  - Old Value
  - New Value

- **busB**
  - Old Value
  - New Value

- **Address**
  - Old Value
  - New Value

- **busW**
  - Old Value
  - New Value

- **Instruction Memory Access Time**
- **Delay through Control Logic**
- **Register File Access Time**
- **Delay through Extender & Mux**
- **ALU Delay**
- **Data Memory Access Time**
5 steps to design a processor

• 1. Analyze instruction set => datapath requirements
• 2. Select set of datapath components & establish clock methodology
• 3. Assemble datapath meeting the requirements
• 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
• 5. Assemble the control logic

MIPS makes it easier

• Instructions same size
• Source registers always in same place
• Immediates same size, location
• Operations always on registers/immediates

Single cycle datapath => CPI=1, CCT => long

Next time: implementing control