Designing MIPS Processor (Single-Cycle)

Dr. Arjan Durresi  
Louisiana State University  
Baton Rouge, LA 70810  
Durresi@Csc.LSU.Edu

These slides are available at:  
http://www.csc.lsu.edu/~durresi/CSC3501_07/

Overview

- Datapath  
- Control Unit  
- Problems with single cycle Datapath
Performance

- Instruction Count
- Clock cycle time
- Clock cycle per Instruction

Implementation of Processor

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, j
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers

More Implementation Details

- Abstract / Simplified View:

- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

Building the Datapath
Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements at the beginning of the clock cycle,
  - send values through some combinational logic,
  - write results to one or more state elements at the end of the clock cycle.

- An edge triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could to indeterminate data.

---

Single Cycle Design

- We shall first design a simpler processor that executes each instruction in only one clock cycle time.
- This is not efficient from performance point of view, since:
  - a clock cycle time (i.e. clock rate) must be chosen such that the longest instruction can be executed in one clock cycle
  - makes shorter instructions execute in one unnecessary long cycle.
- Additionally, no resource in the design may be used more than once per instruction, thus some resources will be duplicated.
- Because of that, the single cycle design will require:
  - two memories (instruction and data),
  - two additional adders.
Elements for Datapath Design

Include the functional units we need for each instruction

- Instruction memory
- Program counter
- Adder

Why do we need this stuff?

Abstract /Simplified View (1st look)

Generic implementation:
- use the program counter (PC) to supply instruction address,
- get the instruction from memory,
- read registers,
- use the instruction to decide exactly what to do.
Abstract /Simplified View (2nd look)

- PC is incremented by 4, by most instructions, and by $4 + 4 \times \text{offset}$, by branch instructions.
- Jump instructions change PC differently (not shown).

Incrementing PC & Fetching Instruction

Figure 5.6 with addition in red
Two elements needed to implement R-formant ALU operations

```
add $t1,$t2,$t3
```

Complete Datapath for R-type Instructions

Based on contents of op-code and funct fields, Control Unit sets ALU control appropriately and asserts RegWrite, i.e. RegWrite = 1.
Two elements needed to implement loads and stores

```
lw $t1, offset_value($t2)
Sw $t1, offset_value($t2)
```

Compute a memory address by adding the base register ($t2) to the 16-bit signed offset field contained in the instruction

Datapath for LW and SW Instructions

Control Unit sets:
- ALU control = 0010 (add) for address calculation for both lw and sw
- MemRead=0, MemWrite=1 and RegWrite=0 for sw
- MemRead=1, MemWrite=0 and RegWrite=1 for lw
Datapath for R-type, LW & SW Instructions

Let us determine setting of control lines for R-type, lw & sw instructions.

The datapath for a branch

beq $t1, $t2, offset

Compute the branch target address by adding the sign-extended offset of the instruction to PC
Generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field ALUOp
Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction

Example:

```
add $8, $17, $18
```

Instruction Format:

```
000000 10001 10010 01000 00000 100000
```

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

ALU’s operation based on instruction type and function code

Load, store and branch instructions

Load or store instruction

```
35 or 43  rs  rt  address
```

31:26 25:21 20:16 15:0

Branch instruction

```
4  rs  rt  address
```

31:26 25:21 20:16 15:0
Control

- Must describe hardware to compute 4-bit ALU control input
  - given instruction type
    - 00 = lw, sw
    - 01 = beq,
    - 10 = arithmetic
  - function code for arithmetic
- Describe it using a truth table (can turn into gates):

<table>
<thead>
<tr>
<th>ALUType</th>
<th>Function Field</th>
<th>Control</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x x x x</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x x x x</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x x x x</td>
<td>0 1 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x x x x</td>
<td>0 0 1 0 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>x x x x</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x x x x</td>
<td>1 0 0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x x x x</td>
<td>0 1 0 1 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x x x x</td>
<td>0 0 1 1 1</td>
</tr>
</tbody>
</table>

FIGURE 5.13 The truth table for the main ALU control bits (called Operation). The inputs are the ALUs operation code field plus the inputs for which the ALU control is asserted are shown. Note don’t care entries have been added. For example, the 8000 from the floating-point floating
the truth table contains entries 11 and 11, rather than 8 and 8. Also, when the function field is used, the
and have two bits 000 and 001 of these instructions are shown 11 so that the don’t care entries are replaced
with 11 in the table.

Truth Table for (Main) Control Unit

- ALUOp[1-0] = 00 → signal to ALU Control unit for ALU to perform add function, i.e. set Ainvert = 0, Binvert=0 and Operation=10
- ALUOp[1-0] = 01 → signal to ALU Control unit for ALU to perform subtract function, i.e. set Ainvert = 0, Binvert=1 and Operation=10
- ALUOp[1-0] = 10 → signal to ALU Control unit to look at bits \(i_{[5:4]}\) and based on its pattern to set Ainvert, Binvert and Operation so that ALU performs appropriate function, i.e. add, sub, slt, and, or & nor

<table>
<thead>
<tr>
<th>Op-code</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>Reg</th>
<th>Mem</th>
<th>Write</th>
<th>Mem</th>
<th>Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>100011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>101011</td>
<td>d</td>
<td>1</td>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>000100</td>
<td>d</td>
<td>0</td>
<td>d</td>
<td>0</td>
<td>d</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Truth Table of ALU Control Unit

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>ALUOp0</th>
<th>F5</th>
<th>F4</th>
<th>F3</th>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>ALU Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>0.10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>0.10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Aimvert Bivert Operation

Design of (Main) Control Unit

<table>
<thead>
<tr>
<th>Op-code Bits</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1000111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1010111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>


Figure C.2.5
Design of Control Unit (J included)

<table>
<thead>
<tr>
<th>Op-code bits</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>BrCond</th>
<th>ALUOp</th>
<th>ALUOp</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101011</td>
<td>d</td>
<td>1</td>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000100</td>
<td>d</td>
<td>0</td>
<td>d</td>
<td>0</td>
<td>d</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000010</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>0</td>
<td>0</td>
<td>d</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Jump = \text{Op}_9 \text{Op}_8 \text{Op}_7 \text{Op}_6 \text{Op}_5 \text{Op}_4 \text{Op}_3 \text{Op}_2 \text{Op}_1 \text{Op}_0

No changes in ALU Control unit

Design of 7-Function ALU Control Unit

![Diagram of 7-Function ALU Control Unit](attachment:image.png)

Figure C.2.3
with improvements

ALU Control Lines
(Bitwise & Operation)

Louisiana State University
8- Single Cycle Datapath - 27
CSC3501 S07
Cycle Time Calculation

- Let us assume that the only delays introduced are by the following tasks:
  - Memory access (read and write time = 3 nsec)
  - Register file access (read and write time = 1 nsec)
  - ALU to perform function (= 2 nsec)
- Under those assumption here are instruction execution times:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Reg</th>
<th>ALU</th>
<th>Data</th>
<th>Reg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>7 nsec</td>
</tr>
<tr>
<td>lw</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>10 nsec</td>
</tr>
<tr>
<td>sw</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>9 nsec</td>
</tr>
<tr>
<td>branch</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td></td>
<td>6 nsec</td>
</tr>
<tr>
<td>jump</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3 nsec</td>
</tr>
</tbody>
</table>

- Thus a clock cycle time has to be 10 nsec, and clock rate = 1/10 nsec = 100MHz

Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (200ps),
  - ALU and adders (100ps),
  - register file access (50ps)
Performance of Single cycle machines

- Memory (200ps), ALU and adders (100ps), register file access (50ps)
- Instructions mix: 25% loads, 10% stores, 45% ALU, 15% branches, 5% jumps.
- Which of the following implementations would be faster?
  - Every instruction operates in a 1 clock cycle of a fixed length
  - Every instruction operates in a 1 clock cycle of a variable length
- CPU execution time = Instruction count x CPI x Clock cycle time
- Since CPI = 1
- CPU execution time = Instruction count x Clock cycle time
- Using the critical paths we can compute the required length for each class:
  - R-type 400ps, Load word 600ps, Store word 550ps, Branch 350ps, jump 200ps
- In case 1 the clock has to be 600ps depending on the longest instruction
- A machine with a variable clock will have a clock cycle that varies between 200ps and 600ps.
- The average CPU clock cycle: 600×25% + 550×10% + 400×45% + 350×15%+200×5% = 447.5ps
- So the variable clock machine is faster 1.34 times

Example

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Functional units used by the instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>Load word</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>Store word</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>Branch</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>Jump</td>
<td>Instruction fetch</td>
</tr>
</tbody>
</table>
**Example**

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Instruciton memory</th>
<th>Register read</th>
<th>ALU</th>
<th>Data memory</th>
<th>Register write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>0</td>
<td>50</td>
<td>400ps</td>
</tr>
<tr>
<td>Load word</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>600ps</td>
</tr>
<tr>
<td>Store word</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>0</td>
<td>550ps</td>
</tr>
<tr>
<td>Branch</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>0</td>
<td></td>
<td>350ps</td>
</tr>
<tr>
<td>Jump</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200ps</td>
</tr>
</tbody>
</table>

**Abstract View of our single cycle process:**

- Looks like a FSM with PC as state.
What's wrong with our CPI=1 processor?

- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not as nice as our idealized memory
  - cannot always get the job done in one (short) cycle

Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a "smaller" cycle time
  - have different instructions take different numbers of cycles
  - a "multicycle" datapath:
Single Cycle Processor: Conclusion

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - a clock cycle would be much longer,
  - thus for shorter and more often used instructions, such as add & lw, wasteful of time.

- One Solution:
  - use a "smaller" cycle time, and
  - have different instructions take different numbers of cycles.

- And that is a "multi-cycle" processor.