Single Cycle CPU

Jason Mars
The Big Picture: The Performance Perspective

Execute an entire instruction
The Big Picture: The Performance Perspective

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

Execute an entire instruction
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

- \[ ET = \text{Insts} \times CPI \times \text{Cyc Time} \]
Processor Datapath and 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`
Processor Datapath and 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 **program counter (PC)** to supply instruction address
  • get the **instruction** from memory
  • read registers
  • use the instruction to decide exactly what to do
Processor Datapath and 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 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
  • memory-reference? arithmetic? control flow?
Review: MIPS Instruction Formats

- All instructions 32-bits long

- 3 Formats:

<table>
<thead>
<tr>
<th>R-Type</th>
<th>I-Type</th>
<th>J-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode 6 bits</td>
<td>rs 5 bits</td>
<td>target 26 bits</td>
</tr>
<tr>
<td>rt 5 bits</td>
<td>rd 5 bits</td>
<td></td>
</tr>
<tr>
<td>shift amount 5 bits</td>
<td>immediate / offset 16 bits</td>
<td></td>
</tr>
<tr>
<td>funct 6 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The MIPS Subset

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift amount</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>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</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>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
The MIPS Subset

- R-Type
  - add rd, rs, rt
  - sub, and, or, slt

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shift</td>
<td>funct</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift amount</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>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</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>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</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

The MIPS instruction set includes three main types: R-Type, LOAD and STORE, and BRANCH. Each instruction type has a specific format for its components:

**R-Type:**
- opcode (6 bits)
- rs (5 bits)
- rt (5 bits)
- rd (5 bits)
- funct (6 bits)

**LOAD and STORE:**
- opcode (6 bits)
- rs (5 bits)
- rt (5 bits)
- immediate/offset (16 bits)

**BRANCH:**
- opcode (6 bits)
- rs (5 bits)
- rt (5 bits)
- immediate/offset (16 bits)
Basic Steps of Execution
Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?
Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?
- Decode
  - What’s the incoming instruction?
  - Where are the operands in an instruction?
Basic Steps of Execution

• Instruction Fetch
  • Where is the instruction?

• Decode
  • What’s the incoming instruction?
  • Where are the operands in an instruction?

• Execution: ALU
  • What is the function that ALU should perform?
Basic Steps of Execution

• Instruction Fetch
  • Where is the instruction?

• Decode
  • What’s the incoming instruction?
  • Where are the operands in an instruction?

• Execution: ALU
  • What is the function that ALU should perform?

• Memory access
  • Where is my data?
Basic Steps of Execution

• Instruction Fetch
  • Where is the instruction?

• Decode
  • What’s the incoming instruction?
  • Where are the operands in an instruction?

• Execution: ALU
  • What is the function that ALU should perform?

• Memory access
  • Where is my data?

• Write back results to registers
  • Where to write?
Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?
- Decode
  - What’s the incoming instruction?
  - Where are the operands in an instruction?
- Execution: ALU
  - What is the function that ALU should perform?
- Memory access
  - Where is my data?
- Write back results to registers
  - Where to write?
- Determine the next PC
Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction? *Instruction memory address: PC*
- Decode
  - What’s the incoming instruction?
  - Where are the operands in an instruction?
- Execution: ALU
  - What is the function that ALU should perform?
- Memory access
  - Where is my data?
- Write back results to registers
  - Where to write?
- Determine the next PC
Basic Steps of Execution

• Instruction Fetch
  • Where is the instruction?  
    Instruction memory address: PC

• Decode
  • What’s the incoming instruction?  
    register file
  • Where are the operands in an instruction?

• Execution: ALU
  • What is the function that ALU should perform?

• Memory access
  • Where is my data?

• Write back results to registers
  • Where to write?

• Determine the next PC
Basic Steps of Execution

• Instruction Fetch
  • Where is the instruction? Instruction memory address: PC

• Decode
  • What’s the incoming instruction? register file
  • Where are the operands in an instruction?

• Execution: ALU
  • What is the function that ALU should perform? ALU

• Memory access
  • Where is my data?

• Write back results to registers
  • Where to write?

• Determine the next PC
Basic Steps of Execution

- **Instruction Fetch**
  - Where is the instruction?  
    - Instruction memory address: PC

- **Decode**
  - What’s the incoming instruction?  
    - register file
  - Where are the operands in an instruction?

- **Execution: ALU**
  - What is the function that ALU should perform?

- **Memory access**
  - Where is my data?  
    - Data memory address: effective address

- **Write back results to registers**
  - Where to write?

- **Determine the next PC**
Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?
  - Instruction memory address: PC

- Decode
  - What’s the incoming instruction?
  - Where are the operands in an instruction?
  - register file

- Execution: ALU
  - What is the function that ALU should perform?
  - ALU

- Memory access
  - Where is my data?
  - Data memory address: effective address

- Write back results to registers
  - Where to write?
  - register file

- Determine the next PC
Basic Steps of Execution

• Instruction Fetch
  • Where is the instruction? Instruction memory
    address: PC

• Decode
  • What’s the incoming instruction? register file
  • Where are the operands in an instruction?

• Execution: ALU
  • What is the function that ALU should perform? ALU

• Memory access
  • Where is my data? Data memory
    address: effective address

• Write back results to registers
  • Where to write? register file

• Determine the next PC program counter
Where We’re Going...
Where We’re Going...

Instruction memory address: PC
Where We’re Going...

Instruction memory address: PC

register file
Where We’re Going...

Instruction memory address: PC

register file

ALU

Data

Register #

Registers

Register #

Instruction memory

Address

Instruction

PC

register file

ALU

Data

Register #

Registers

Register #

Instruction memory

Address

Instruction

PC
Where We’re Going...

Instruction memory address: PC

register file

ALU

Data memory address: effective address
Where We’re Going...

Instruction memory
address: PC

Data memory
address: effective address

program counter

PC

Instruction memory

Address

Instruction

Register #

Registers

Register #

Register #

Data

ALU

Address

Data
Review: Two Type of Logical Components
Review: Two Type of Logical Components

\[ C = f(A,B) \]
Review: Two Type of Logical Components

A ——— Combinational Logic ——— B

C = f(A,B)

A ——— State Element ——— B

clk

C = f(A,B,state)
Clocking Methodology

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

32 32-bit Registers

- Write Data
- RR1
- RR2
- WR
- Clk

RegWrite

Read Data 1
Read Data 2
Storage Element: Register File

- Register File consists of (32) registers:
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- Register is selected by:
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- Register is selected by:
  - RR1 selects the register to put on bus “Read Data 1”
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- 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”
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- 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
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- 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
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- 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)
Inside the Register File

- The implementation of two read ports register file
  - n registers
  - done with a pair of n-to-1 multiplexors, each 32 bits wide.
Storage Element: Memory
Storage Element: Memory

- Memory

![Memory Diagram]

- MemWrite
- Address
- Write Data
  - 32
  - Clk
- MemRead
- Read Data
  - 32
Storage Element: Memory

- Memory
  - Two input buses: **WriteData**, **Address**
Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData
Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData
- Memory word is selected by:
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
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
  - If MemWrite = 1: address selects the memory word to be written via the WriteData bus
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
  - If MemWrite = 1: address selects the memory word to be written via the WriteData bus
- Clock input (CLK)
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
  - If 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
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
  - If 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:
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
  - If 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."
RTL: Register Transfer Language

- Describes the movement and manipulation of data between storage elements:

  \[ PC <- PC + 4 + R[5] \]
  \[ R[rd] <- R[rs] + R[rt] \]
  \[ R[rt] <- Mem[R[rs] + \text{immed}] \]
Instruction Fetch and Program Counter Management

a. Instruction memory

b. Program counter

c. Adder
Overview of the Instruction Fetch Unit

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

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift amount</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>

Diagram showing the datapath for register-register operations with steps including reading registers, performing an ALU operation, and writing data.
Datapath for Register-Register Operations

- \( R[rd] \leftarrow R[rs] \text{ op } R[rt] \)  
  Example: \textit{add} \quad rd, rs, rt
Datapath for Register-Register Operations

- $R[rd] \leftarrow R[rs] \text{ op } R[rt]$  Example: $add \quad rd, \quad rs, \quad rt$
- $RR1, \quad RR2, \quad$ and $WR$ comes from instruction’s $rs, \quad rt, \quad$ and $rd$ fields
Datapath for Register-Register Operations

- \( R[rd] \leftarrow R[rs] \text{ op } R[rt] \)  Example: \( add \quad rd, \quad rs, \quad rt \)
- RR1, RR2, and WR comes from instruction’s rs, rt, and rd fields
- ALUoperation and RegWrite: control logic after decoding instruction
While easier to understand, this approach is not practical, since it would be slower than an implementation that allows different instruction classes to take different numbers of clock cycles, each of which could be much shorter. After designing the control for this simple machine, we will look at an implementation that uses multiple clock cycles for each instruction. This multicycle design is used to handle more complex instruction sequences efficiently.
While easier to understand, this approach is not practical, since it would be slower than an implementation that allows different instruction classes to take different numbers of clock cycles, each of which could be much shorter. After designing the control for this simple machine, we will look at an implementation that uses multiple clock cycles for each instruction. This multicycle design is used to implement the branch instruction.

**FIGURE 5.2** The basic implementation of the MIPS subset including the necessary multiplexors and control lines. The top multiplexor controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that "ands" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The multiplexor whose output returns to the register file is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for a nonimmediate arithmetic-logical instruction) or from the offset field of the instruction (for an immediate operation, a load or store, or a branch). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.
Datapath for Load Operations

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

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

![Diagram of datapath for load operations](image)
Datapath for Load Operations

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

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</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:
- Instruction
- Read register 1
- Read register 2
- Registers
- Write register
- Write data
- RegWrite
- 16
- Sign extend
- 32
- 3
- ALU operation
- Address
- Read data
- Write data
- Data memory
- MemRead
- MemWrite
Datapath for Load Operations

- $R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[imm16]]$  
  Example: $lw \quad rt, rs, imm16$

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Datapath for Load Operations

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

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

Diagram:

- Instruction
- Read register
- Write register
- RegWrite
- Sign extend
- ALU operation
- Address
- Data memory
- MemRead
- MemWrite
Datapath for Load Operations

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

- $R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[imm16]]$  
  Example: $lw \ rt, rs, imm16$

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

Diagram showing the datapath for load operations, including the instruction, read registers, ALU operation, address generation, and memory read/write processes.
Datapath for Load Operations

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

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

Instruction 

Read register 1
Read register 2
Write register
Write data

RegWrite

Sign extend

ALU operation

MemWrite

Read data

Data memory

Write data

MemRead
Datapath for Load Operations

- \( R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[imm16]] \)  
  Example: \( lw \quad rt, rs, \text{imm16} \)
Datapath for Load Operations

- \( R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]] \)  
  Example: \( \text{lw} \quad rt, rs, \text{imm16} \)
Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R(rt)  
  Example: sw   rt, rs, imm16
Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt]  
  Example: sw  rt, rs, imm16
Datapath for Store Operations

• Mem[R[rs] + SignExt[imm16]] <- R[rt]  
Example: sw   rt, rs, imm16
Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt]  
  Example: sw  rt, rs, imm16

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</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 showing the datapath with instructions and registers.
Datapath for Store Operations

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

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

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</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:
- Instruction
- Read register 1
- Read register 2
- Registers
- Write register
- Write data
- RegWrite
- Sign extend
- ALU operation
- Address
- Data memory
- MemWrite
- MemRead
- Read data
- Write data
Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt]  
  Example: sw  rt, rs, imm16

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

Example:

- sw  rt, rs, imm16

Diagram:

- Instruction
- Read register 1
- Read register 2
- Registers
- Write register
- Write data
- RegWrite
- Sign extend 16
- ALU operation
- ALU result
- Address
- Data memory
- Read data
- MemRead
- MemWrite
Datapath for Branch Operations

- \( Z \leftarrow (\text{rs} \equiv \text{rt}); \) if \( Z \), \( \text{PC} = \text{PC}+4+\text{imm16} \); else \( \text{PC} = \text{PC}+4 \)
- \( \text{beq} \quad \text{rs}, \text{rt}, \text{imm16} \)

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

PC + 4 from instruction datapath

Instruction

1. Read register 1
2. Read register 2
3. Read data 1
4. Write register
5. Write data
6. RegWrite

16 Sign extend
32

Add Sum

Branch target

Shift left 2

ALU operation

ALU Zero

To branch control logic
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt); \) if \( Z \), \( PC = PC+4+imm16; \) else \( PC = PC+4 \)
- beq  \( rs, rt, imm16 \)

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

PC + 4 from instruction datapath

Add  Sum  Branch target

Shift left 2

3  ALU operation

ALU Zero  To branch control logic

Instruction

Read register 1
Read register 2
Write register
Write data

RegWrite

16 Sign extend 32
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt) \); if \( Z \), \( PC = PC + 4 + \text{imm}_16 \); else \( PC = PC + 4 \)
- \( \text{beq } rs, rt, \text{imm}_16 \)

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</th>
</tr>
</thead>
</table>

6 bits 5 bits 5 bits 16 bits

PC + 4 from instruction datapath

Add Sum Branch target

Shift left 2

3 ALU operation

To branch control logic

Instruction

Read register 1

Read register 2

Write register

Read data 1

Read data 2

RegWrite

16

32

Sign extend
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt); \) if \( Z \), \( PC = PC + 4 + \text{imm16} \); else \( PC = PC + 4 \)
- \textit{beq} \hspace{1em} rs, rt, \text{imm16}
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt); \) if \( Z \), \( PC = PC+4+imm16 \); else \( PC = PC+4 \)
- \( \text{beq } rs, rt, imm16 \)

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

\( PC + 4 \) from instruction datapath

Add Sum → Branch target

Shift left 2

ALU operation

To branch control logic

Instruction

Read register 1
Read register 2
Write register
Write data
RegWrite

Sign extend

16

32
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt); \) if \( Z \), \( PC = PC + 4 + \text{imm16} \); else \( PC = PC + 4 \)
- \textit{beq rs, rt, imm16}

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</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 of the datapath for branch operations](image)
Datapath for Branch Operations

- $Z \leftarrow (rs == rt)$; if $Z$, PC = PC+4+imm16; else PC = PC+4
- $beq \; rs, \; rt, \; imm16$

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

PC + 4 from instruction datapath

Add Sum → Branch target

Shift left 2 → 3

ALU operation

To branch control logic
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt); \) if \( Z \), PC = PC+4+imm16; else PC = PC+4
- ```
  beq    rs, rt, imm16
  ```

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

PC + 4 from instruction datapath → Add Sum → Branch target

- Instruction
- Read register
- Read register 2
- Write register
- Write data
- ALU operation 3
- ALU Zero
- To branch control logic
- Sign extend
- RegWrite
- Shift left 2
- 16
- 32
Datapath for Branch Operations

- $Z \leftarrow (rs == rt)$; if $Z$, $PC = PC+4+imm16$; else $PC = PC+4$
- $beq \quad rs, rt, imm16$

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

PC + 4 from instruction datapath

Add Sum  
Branch target

Shift left 2

3  
ALU operation

ALU Zero  
To branch control logic

Instruction  
Read register 1
Read register 2
Write register
Write data

RegWrite

Sign extend

16

32
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt) \); if \( Z \), \( PC = PC + 4 + \text{imm16} \); else \( PC = PC + 4 \)
- \textit{beq} \quad rs, rt, \text{imm16}

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

PC + 4 from instruction datapath

Add Sum → Branch target

Shift left 2

3 → ALU operation

ALU Zero → Control logic

Read register 1
Read register 2
Write register
Write data
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt); \) if \( Z \), \( PC = PC+4+imm16 \); else \( PC = PC+4 \)
- `beq rs, rt, imm16`

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

PC + 4 from instruction datapath

Add Sum \( \rightarrow \) Branch target

Shift left 2

3 \( \uparrow \) ALU operation

To branch control logic

Instruction

Read register 1
Read register 2
Write register
Write data
RegWrite

16 \( \rightarrow \) Sign extend

32
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt); \) if \( Z \), PC = PC+4+imm16; else PC = PC+4
- \textit{beq} \quad rs, rt, imm16

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

PC + 4 from instruction datapath

Add Sum
Branch target

Shift left 2

3
ALU operation

ALU Zero
To branch control logic

Instruction

Read register 1
Read register 2
Write register
Write data
RegWrite

16 Sign extend
32
Datapath for Branch Operations

- \( Z \leftarrow (rs == rt); \) if \( Z \), \( PC = PC+4+imm16 \); else \( PC = PC+4 \)
- \( \text{beq } rs, rt, imm16 \)

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

PC + 4 from instruction datapath

Add Sum \( \rightarrow \) Branch target

Shift left 2

ALU operation

ALU Zero \( \rightarrow \) To branch control logic

Instruction \( \rightarrow \) Read register 1

Read register 2 \( \rightarrow \) Read data 1

Write register \( \rightarrow \) Read data 2

RegWrite \( \rightarrow \) 16 Sign extend

32
While easier to understand, this approach is not practical, since it would be slower than an implementation that allows different instruction classes to take different numbers of clock cycles, each of which could be much shorter. After designing the control for this simple machine, we will look at an implementation that uses multiple clock cycles for each instruction. This multicycle design is used to determine whether the second ALU input is from the registers (for a nonimmediate arithmetic-logical instruction) or from the offset field of the instruction (for an immediate operation, a load or store, or a branch). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.
While easier to understand, this approach is not practical, since it would be slower than an implementation that allows different instruction classes to take different numbers of clock cycles, each of which could be much shorter. After designing the control for this simple machine, we will look at an implementation that uses multiple clock cycles for each instruction. This multicycle design is used to handle more complex instructions and improve performance.

FIGURE 5.2 The basic implementation of the MIPS subset including the necessary multiplexors and control lines. The top multiplexor controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that “ands” together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The multiplexor whose output returns to the register file is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for a nonimmediate arithmetic-logical instruction) or from the offset field of the instruction (for an immediate operation, a load or store, or a branch). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.
Binary Arithmetic for the Next Address
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: $PC_{31:0} = PC_{31:0} + 4 + \text{SignExt}[\text{Imm16}] \times 4$
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
  • The 2 LSBs of the 32-bit PC are always zeros
  • There is no reason to have hardware to keep the 2 LSBs
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 + 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
GAME: GUESS THE FUNCTION!!

- We have everything except control signals
GAME: GUESS THE FUNCTION!!

- We have everything except control signals
GAME: GUESS THE FUNCTION!!

- We have everything except control signals
GAME: GUESS THE FUNCTION!!

- We have everything except control signals
GAME: GUESS THE FUNCTION!!

• We have everything except control signals
GAME: GUESS THE FUNCTION!!

- We have everything except control signals
GAME: GUESS THE FUNCTION!!

- We have everything except control signals
GAME: GUESS THE FUNCTION!!

- We have everything except control signals
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath

[Diagram of the R-Format Datapath]
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath
The R-Format (e.g. add) Datapath
The Load Datapath
The Load Datapath
The Load Datapath
The Load Datapath
The Load Datapath
The Load Datapath
The Load Datapath
The Load Datapath
The Load Datapath
The Load Datapath
The Load Datapath
The Store Datapath
The Store Datapath
The Store Datapath
The Store Datapath
The Store Datapath
The Store Datapath
The Store Datapath
The Store Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath

[Diagram of the Branch Datapath]

- **PC**: Read address
- **Instruction memory**: [31 0]
- **Instruction [31 0]**:
  - [25 21]: Read register 1
  - [20 16]: Read register 2
  - [15 11]: Write register
  - [15 0]: RegDst
- **Instruction [15 0]**:
  - [5 0]: ALU control
- **ALUOp**: 16
- **32**: Sign extend
- **ALU result**: Zero
- **Shift left 2**: Add
- **Add result**: MUX 0
- **PCSr0**: MUX 0
- **MemRead**: Mux Write data
- **Data memory**: Read data
- **MemWrite**: RegWrite
- **RegWrite**: MUX 0

- **ALUSrc**: MUX 0

---

Tuesday, February 5, 13
The Branch (beq) Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath
The Branch (beq) Datapath
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals
GAME: GUESS THE FUNCTION!! (Review)

• We have everything except control signals
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals
Key Points
Key Points

• CPU is just a collection of state and combinational logic
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
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

• $ET = IC \times CPI \times \text{Cycle Time}$
  
  • where does the single-cycle machine fit in?
The Control Unit
Putting it All Together: A Single Cycle Datapath

- We have everything except control signals
Putting it All Together: A Single Cycle Datapath
Putting it All Together: A Single Cycle Datapath
Putting it All Together: A Single Cycle Datapath
ALU Control Bits

• 5-Function ALU

<table>
<thead>
<tr>
<th>ALU control input</th>
<th>Function</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>And</td>
<td>and</td>
</tr>
<tr>
<td>001</td>
<td>Or</td>
<td>or</td>
</tr>
<tr>
<td>010</td>
<td>Add</td>
<td>add, lw, sw</td>
</tr>
<tr>
<td>110</td>
<td>Subtract</td>
<td>sub, beq</td>
</tr>
<tr>
<td>111</td>
<td>Slt</td>
<td>slt</td>
</tr>
</tbody>
</table>

• Note: book also has NOR, not used - and a forth bit, not used
what signals accomplish:

Binvert CIn Oper
add?
sub?
and?
or?
beq?
slt?
what signals accomplish:

- Binvert
- CIn
- Oper

add? sub? and? or? beq? slt?
Full ALU

what signals accomplish:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIGURE C.5.14 The symbol commonly used to represent an ALU, as shown in Figure C.5.12. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

module MIPSALU (ALUctl, A, B, ALUOut, Zero);
input [3:0] ALUctl;
input [31:0] A, B;
output reg [31:0] ALUOut;
output Zero;
assign Zero = (ALUOut == 0); // Zero is true if ALUOut is 0
always @(ALUctl, A, B) begin // reevaluate if these change
  case (ALUctl)
    0: ALUOut <= A & B;
    1: ALUOut <= A | B;
    2: ALUOut <= A + B;
    6: ALUOut <= A - B;
    7: ALUOut <= A < B ? 1 : 0;
    12: ALUOut <= ~(A | B); // result is nor
    default: ALUOut <= 0;
  endcase
end
endmodule

FIGURE C.5.15 A Verilog behavioral definition of a MIPS ALU.
Full ALU

what signals accomplish:

<table>
<thead>
<tr>
<th>Binvert</th>
<th>CIn</th>
<th>Oper</th>
</tr>
</thead>
<tbody>
<tr>
<td>add?</td>
<td>0</td>
<td>0 10</td>
</tr>
<tr>
<td>sub?</td>
<td>1</td>
<td>1 10</td>
</tr>
<tr>
<td>and?</td>
<td>0</td>
<td>0 0 10</td>
</tr>
<tr>
<td>or?</td>
<td>0</td>
<td>0 0 00</td>
</tr>
<tr>
<td>beq?</td>
<td>0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>slt?</td>
<td>0</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

FIGURE C.5.15 A Verilog behavioral definition of a MIPS ALU.
Full ALU

what signals accomplish:

<table>
<thead>
<tr>
<th>Binvert</th>
<th>CIn</th>
<th>Oper</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>sub?</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>and?</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>or?</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>beq?</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>slt?</td>
<td>0</td>
<td>01</td>
</tr>
</tbody>
</table>

FIGURE C.5.14 The symbol commonly used to represent an ALU, as shown in Figure C.5.12.

FIGURE C.5.15 A Verilog behavioral definition of a MIPS ALU.

C.5 Constructing a Basic Arithmetic Logic Unit
Full ALU

what signals accomplish:

<table>
<thead>
<tr>
<th>Binvert</th>
<th>CIn</th>
<th>Oper</th>
</tr>
</thead>
<tbody>
<tr>
<td>add?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sub?</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>and?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>or?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq?</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>slt?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
what signals accomplish:

<table>
<thead>
<tr>
<th>Binvert</th>
<th>CIn</th>
<th>Oper</th>
</tr>
</thead>
<tbody>
<tr>
<td>add?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sub?</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>and?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>or?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq?</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>slt?</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
ALU Control Bits

- 5-Function ALU

<table>
<thead>
<tr>
<th>ALU control input</th>
<th>Function</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>And</td>
<td>and</td>
</tr>
<tr>
<td>001</td>
<td>Or</td>
<td>or</td>
</tr>
<tr>
<td>010</td>
<td>Add</td>
<td>add, lw, sw</td>
</tr>
<tr>
<td>110</td>
<td>Subtract</td>
<td>sub, beq</td>
</tr>
<tr>
<td>111</td>
<td>Slt</td>
<td>slt</td>
</tr>
</tbody>
</table>

- Based on opcode (bits 31-26) and function code (bits 5-0) from instruction
- ALU doesn’t need to know all opcodes--we will summarize opcode with ALUOp (2 bits):
  - 00 - lw, sw  01 - beq  10 - R-format
## Generating ALU Control

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>Function code</th>
<th>Desired ALU action</th>
<th>ALU control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>00</td>
<td>load word</td>
<td>xxxxxxx</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>sw</td>
<td>00</td>
<td>store word</td>
<td>xxxxxxx</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>branch eq</td>
<td>xxxxxxx</td>
<td>subtract</td>
<td>110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>AND</td>
<td>100100</td>
<td>and</td>
<td>000</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>OR</td>
<td>100101</td>
<td>or</td>
<td>001</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>slt</td>
<td>101010</td>
<td>slt</td>
<td>111</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Mem to Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tuesday, February 5, 13
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Controlling the CPU

### R-format Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

**Table: Instruction Formats**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Tuesday, February 5, 13
Controlling the CPU

**R-format**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</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>sw</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Mmto-Reg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</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>sw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</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>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Tuesday, February 5, 13
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Mem to Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Tuesday, February 5, 13
Controlling the CPU

### Instruction Format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</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>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Mem-to-Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Tuesday, February 5, 13
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</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>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Tuesday, February 5, 13
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

### Instruction Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Mem-to-Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</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>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

PC = 4

Add

RegDst Branch MemRead MemtoReg ALUOp MemWrite ALUSrc RegWrite

PCSrc

Shift left 2

Add ALU result

ALUOp

Mem

Write

Branch

ALUOp1

ALUOp0

ALUOp

Add

RegDst

Control

Instruction [31-26]

Instruction [25-21]

Instruction [20-16]

Instruction [15-11]

Instruction [15-0]

Instruction memory

Read address

Write data

Read register 1

Read data 1

Read register 2

Read data 2

Write register

Write data

RegDst branch

MemRead

MemtoReg

ALUOp

MemWrite

ALUSrc

RegWrite

ALU control

Address

Read data

Data memory

Write data
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Mem to Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

### R-format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</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>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

PCSrc

Add

Shift left 2

ALU result

RegDst

Branch

MemRead

MemtoReg

ALUOp

MemWrite

ALUSrc

RegWrite

Add

ALUOp

Mux

ALU control

Address

Write data

Data memory

Write data

Mux

ALU result

0

1

32

16

align extend

Instruction [31-26]

Instruction [25-21]

Instruction [20-16]

Instruction [15-11]

Instruction [15-0]

Instruction [5-0]
Controlling the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp</th>
<th>ALUp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</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>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Outputs</td>
<td>R-format</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemRead</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUOp1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUOp0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control

• Simple Combinational Logic (truth tables)
Single Cycle CPU Summary
Single Cycle CPU Summary

- Easy, particularly the control
Single Cycle CPU Summary

• Easy, particularly the control

• Which instruction takes the longest?  By how much?  Why is that a problem?
  • $ET = IC \times CPI \times CT$
Single Cycle CPU Summary

• Easy, particularly the control

• Which instruction takes the longest? By how much? Why is that a problem?
  
  • \( ET = IC \times CPI \times CT \)

• What else can we do?
Single Cycle CPU Summary

• Easy, particularly the control

• Which instruction takes the longest? By how much? Why is that a problem?
  • \( ET = IC \times CPI \times CT \)

• What else can we do?

• When does a multi-cycle implementation make sense?
  • e.g., 70% of instructions take 75 ns, 30% take 200 ns?
  • suppose 20% overhead for extra latches
Single Cycle CPU Summary

• Easy, particularly the control

• Which instruction takes the longest? By how much? Why is that a problem?
  • \( ET = IC \times CPI \times CT \)

• What else can we do?

• When does a multi-cycle implementation make sense?
  • e.g., 70% of instructions take 75 ns, 30% take 200 ns?
  • suppose 20% overhead for extra latches

• Real machines have much more variable instruction latencies than this.