CPU Organization (Design)

- **Datapath Design:** Components & their connections needed by ISA instructions
  - Capabilities & performance characteristics of principal Functional Units (FUs) needed by ISA instructions
  - (e.g., Registers, ALU, Shifters, Logic Units, ...) Components
  - Ways in which these components are interconnected (buses connections, multiplexors, etc.) Connections
  - How information flows between components.

- **Control Unit Design:** Control/sequencing of operations of datapath components to realize ISA instructions
  - Logic and means by which such information flow is controlled.
  - Control and coordination of FUs operation to realize the targeted Instruction Set Architecture to be implemented (can either be implemented using a finite state machine or a microprogram).

- **Hardware description with a suitable language, possibly using Register Transfer Notation (RTN).**
Major CPU Design Steps

1. **Analyze instruction set** to get datapath requirements:
   - Using independent RTN, write the micro-operations required for target ISA instructions.
     • This provides the required datapath components and how they are connected.

2. Select set of **datapath components** and establish clocking methodology (defines when storage or state elements can read and when they can be written, e.g. clock edge-triggered). **E.g. Flip-Flops**

3. **Assemble datapath** meeting the requirements.

4. Identify and define the function of all control lines, points or signals needed by the datapath.
   - Analyze implementation of each instruction to determine setting of control points that affects its operations.

5. **Control unit design**, based on micro-operation timing and control signals identified:
   - Combinational logic: For single cycle CPU. **E.g. Any instruction completed in one cycle**
   - Microprogrammed.
CPU Design & Implantation Process

- **Top-down Design:**
  - Specify component behavior from high-level requirements (ISA).

- **Bottom-up Design:**
  - Assemble components in target technology to establish critical timing (hardware delays, critical path timing).

- **Iterative refinement:**
  - Establish a partial solution, expand and improve.

Instruction Set Architecture (ISA): Provides Requirements

Target VLSI implementation Technology
Datapath Design Steps

• Write the micro-operation sequences required for a number of representative target ISA instructions using independent RTN.

• Independent RTN statements specify: the required datapath components and how they are connected.

• From the above, create an initial datapath by determining possible destinations for each data source (i.e. registers, ALU).
  – This establishes connectivity requirements (data paths, or connections) for datapath components.
  – Whenever multiple sources are connected to a single input, a multiplexor of appropriate size is added.

• Find the worst-time propagation delay (critical path) in the datapath to determine the datapath clock cycle (CPU clock cycle, C).

• Complete the micro-operation sequences for all remaining instructions adding datapath components + connections/multiplexors as needed.
MIPS 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><strong>R-Type</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</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>rs, rt, rd</td>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:11]</td>
<td>[10:6]</td>
<td>[5:0]</td>
<td></td>
</tr>
<tr>
<td>shamt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>I-Type: ALU</strong></th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>rs, rt</td>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
<td></td>
</tr>
<tr>
<td>Immediate (imm16)</td>
<td></td>
<td></td>
<td></td>
<td>Or address offset</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>J-Type: Jumps</strong></th>
<th>31</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6 bits</td>
<td>26 bits</td>
<td></td>
</tr>
<tr>
<td>target address</td>
<td>[31:26]</td>
<td>[25:0]</td>
<td></td>
</tr>
</tbody>
</table>

- **op**: Opcode, 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.
MIPS R-Type (ALU) Instruction Fields

R-Type: All ALU instructions that use three registers

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:11]</td>
<td>[10:6]</td>
<td>[5:0]</td>
</tr>
</tbody>
</table>

- **op**: Opcode, basic operation of the instruction.
  - For R-Type op = 0
- **rs**: The first register source operand.
- **rt**: The second register source operand.
- **rd**: The register destination operand.
- **shamt**: Shift amount used in constant shift operations.
- **funct**: Function, selects the specific variant of operation in the op field.

Examples:
- add $1,$2,$3
- sub $1,$2,$3
- add $1,$2,$3
- sub $1,$2,$3

R-Type = Register Type
Register Addressing used (Mode 1)

Function Field

Rs, rt, rd are register specifier fields

Independent RTN:

Instruction Word ← Mem[PC]
R[rd] ← R[rs] funct R[rt]
PC ← PC + 4

Funct field value examples:
Add = 32 Sub = 34 AND = 36 OR =37 NOR = 39

Operands register in rs

Destination register in rd

Operands register in rt

EECC550 - Shaaban
# MIPS ALU I-Type Instruction Fields

I-Type ALU instructions that use two registers and an immediate value
Loads/stores, conditional branches.

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>Immediate (imm16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
</tr>
</tbody>
</table>

- **op**: Opcode, operation of the instruction.
- **rs**: The register source operand.
- **rt**: The result destination register.
- **immediate**: Constant second operand for ALU instruction.

Examples:

- **add immediate**: `addi $1,$2,100`
- **and immediate**: `andi $1,$2,10`

Independent RTN for addi:

- Instruction Word ← Mem[PC]
- R[rt] ← R[rs] + imm16
- PC ← PC + 4

**imm16**

I-Type = Immediate Type
Immediate Addressing used (Mode 2)
MIPS Load/Store I-Type Instruction Fields

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>address (e.g. offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0]</td>
</tr>
</tbody>
</table>

- **op**: Opcode, operation of the instruction.
  - For load word op = 35, for store word op = 43.
- **rs**: The register containing memory base address.
- **rt**: For loads, the destination register. For stores, the source register of value to be stored.
- **address**: 16-bit memory address offset in bytes added to base register.

Examples:

- **Store word**: `sw $3, 500($4)`
  - Instruction Word ← Mem[PC]
  - PC ← PC + 4

- **Load word**: `lw $1, 32($2)`
  - Instruction Word ← Mem[PC]
  - R[rt] ← Mem[R[rs] + imm16]
  - PC ← PC + 4

Signed address offset in bytes

Base or Displacement Addressing used (Mode 3)
MIPS Branch I-Type Instruction Fields

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>address (e.g. offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits imm16</td>
</tr>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:0] Signed address offset in words</td>
</tr>
</tbody>
</table>

- **op**: Opcode, operation of the instruction.
- **rs**: The first register being compared.
- **rt**: The second register being compared.
- **address**: 16-bit memory address branch target offset in words added to PC to form branch address.

Examples:
- **Branch on equal**: beq $1,$2,100
- **Branch on not equal**: bne $1,$2,100

Independent RTN for beq:

Instruction Word ← Mem[PC]
R[rs] = R[rt] : PC ← PC + 4 + imm16 x 4
R[rs] ≠ R[rt] : PC ← PC + 4

PC-Relative Addressing used (Mode 4)

imm16 = 16 bit immediate field

Word = 4 bytes

imm16 = 16 bit immediate field

Imm16 x 4

imm16 = 16 bit immediate field

Imm16 x 4
MIPS J-Type Instruction Fields

J-Type: Include jump j, jump and link jal

<table>
<thead>
<tr>
<th>OP</th>
<th>jump target</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
<tr>
<td>[31:26]</td>
<td>[25:0]</td>
</tr>
</tbody>
</table>

- op: Opcode, operation of the instruction.
  - Jump j   op = 2
  - Jump and link jal op = 3
- jump target: jump memory address in words.

Examples:
- Jump j 10000
- Jump and link jal 10000

Effective 32-bit jump address: PC(31-28),jump_target,00

From PC+4

Jump memory address in bytes equal to instruction field jump target x 4

Independent RTN for j:
Instruction Word ← Mem[PC]
PC ← PC + 4
PC ← PC(31-28),jump_target,00

J-Type = Jump Type Pseudodirect Addressing used (Mode 5)
## A Subset of MIPS Instructions

### ADD and SUB:

- **add rd, rs, rt**
- **sub 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**

#### Details:

- **add rd, rs, rt**
- **sub rd, rs, rt**

#### Table Formats:

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6</td>
<td>Operation</td>
</tr>
<tr>
<td>rs</td>
<td>5</td>
<td>Source Register</td>
</tr>
<tr>
<td>rt</td>
<td>5</td>
<td>Target Register</td>
</tr>
<tr>
<td>rd</td>
<td>5</td>
<td>Destination Register</td>
</tr>
<tr>
<td>shamt</td>
<td>5</td>
<td>Shift Amount</td>
</tr>
<tr>
<td>funct</td>
<td>6</td>
<td>Function</td>
</tr>
</tbody>
</table>

#### Immediate Fields:

- **lw** 35 = 16 bits Offset in bytes
- **sw** 43 = 16 bits Offset in bytes
- **beq** 4 = 16 bits Offset in words
Basic MIPS Instruction Processing Steps

- **Instruction Fetch**
  - Obtain instruction from program storage
  - Instruction ← Mem[PC]

- **Next Instruction**
  - Update program counter to address of next instruction
  - PC ← PC + 4

- **Instruction Decode**
  - Determine instruction type
  - Obtain operands from registers

- **Execute**
  - Compute result value or status

- **Result Store**
  - Store result in register/memory if needed
    (usually called Write Back).

**T = I x CPI x C**
Overview of MIPS Instruction Micro-operations

- All instructions go through these **common steps:**
  - Send program counter to instruction memory and **fetch** the instruction.
    
    *(fetch) Instruction ← Mem[PC]*
  - **Update the program counter** to point to next instruction  
    
    PC ← PC + 4
  - **Read one or two registers**, using instruction fields. *(decode)*
    
    • Load reads one register only.

- **Additional instruction execution actions** *(execution)* depend on the instruction in question, but similarities exist:
  - **All instruction classes** *(except J type)* use the **ALU** after reading the registers:
    
    • Memory reference instructions use it for **effective address calculation**.
    
    • Arithmetic and logic instructions *(R-Type)*, use it for the **specified operation**.
    
    • Branches use it for comparison.

- **Additional execution steps** where instruction classes differ:
  - **Memory reference instructions**: Access memory for a load or store.
  - **Arithmetic and logic instructions**: Write ALU result back in register.
  - **Branch instructions**: Possibly change next instruction address *(update PC)* based on comparison *(i.e if branch is taken).*
Design target: A single-cycle per instruction MIPS CPU design

All micro-operations of an instruction are to be carried out in a single CPU clock cycle. Cycles Per Instruction = CPI = 1

CPU Performance Equation:

\[ T = I \times CPI \times C \]

Abstract view of single cycle MIPS CPU showing major functional units (components) and major connections between them.
R-Type Example: Micro-Operation Sequence For ADD

**add rd, rs, rt**

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>[31:26]</td>
<td>[25:21]</td>
<td>[20:16]</td>
<td>[15:11]</td>
<td>[10:6]</td>
<td>[5:0]</td>
</tr>
</tbody>
</table>

Instruction Word ← Mem[PC]

PC ← PC + 4

R[rd] ← R[rs] + R[rt]

Fetch the instruction

Increment PC

Add register rs to register rt result in register rd

Independent RTN ?

i.e Funct = add = 32

32 = add
34 = sub

Common Steps

Program Memory

EECC550 - Shaaban

#15 Lec # 4 Winter 2012 12-11-2012
Initial Datapath Components

Three components needed by:

- **Instruction Fetch:**
  - Instruction \( \leftarrow \text{Mem}[PC] \)

- **Program Counter Update:**
  - \( PC \leftarrow PC + 4 \)

- **Adder:**
  - 32-bit

Two state elements (memory) needed to store and access instructions:

1. **Instruction memory:**
   - Only read access (by user code). No read control signal needed.

2. **Program counter (PC):** 32-bit register.
   - Written at end of every clock cycle (edge-triggered): No write control signal needed.

3. **32-bit Adder:** To compute the the next instruction address (PC + 4).

---

Basics of logic design/logic building blocks review in Appendix C in Book CD (4th Edition Appendix B)

4th Edition Figure 4.5, page 308 - 3rd Edition Figure 5.5, page 293
Building The Datapath

Instruction Fetch & PC Update:

Instruction ← Mem[PC]
PC ← PC + 4

Portion of the datapath used for fetching instructions and incrementing the program counter (PC).

Clock input to PC, memory not shown

PC write or update is edge triggered at the end of the cycle

4th Edition Figure 4.6 page 309 - 3rd Edition Figure 5.6 page 293
More Datapath Components

ISA Register File

- Contains all ISA registers.
- Two read ports and one write port.
- Register writes by asserting write control signal

Clocking Methodology: Writes are edge-triggered.
- Thus can read and write to the same register in the same clock cycle.

32-bit Arithmetic and Logic Unit (ALU)

Zero = Zero flag = 1
When ALU result equals zero

e.g add = 0010

Register File:

- Contains all ISA registers.
- Two read ports and one write port.
- Register writes by asserting write control signal

Clocking Methodology: Writes are edge-triggered.
- Thus can read and write to the same register in the same clock cycle.
Register File Details

- 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):
    \[ \text{busA} = R[\text{RA}] \]
  - RB (number) selects the register to put on busB (data):
    \[ \text{busB} = R[\text{RB}] \]
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
    \[ \text{Write Enable: } R[\text{RW}] \leftarrow \text{busW} \]

- Clock input (CLK)
  - The CLK input is a factor ONLY during write operations.
  - During read operation, it behaves as a combinational logic block:
    - RA or RB valid \(\Rightarrow\) busA or busB valid after “access time.”
A Possible Register File Implementation

Each Register contains 32 edge triggered D-Flip Flops

Also see Appendix C in Book CD (3rd Edition Appendix B) - The Basics of Logic Design

EECC550 - Shaaban

#20 Lec # 4 Winter 2012 12-11-2012
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 bus.
  - Write Enable = 1: address selects the memory word to be written via the Data In bus.
- Clock input (CLK):
  - The CLK input is a factor ONLY during write operation,
  - During read operation, this memory behaves as a combinational logic block:
    - Address valid => Data Out valid after “access time.”
- Ideal Memory = Short access time.
Clocking Methodology Used: Edge Triggered Writes

- All storage element (e.g. Flip-Flops, Registers, Data Memory) writes are triggered by the same clock edge.
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew

Here writes are triggered on the rising edge of the clock
Simplified Datapath For MIPS
R-Type Instructions

Components and connections as specified by RTN statement:

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

i.e. Funct = function = add

Destination register R[rd] write or update is edge triggered at the end of the cycle.
More Detailed Datapath
For R-Type Instructions
With Control Points Identified

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

i.e. Funct = function = add
R-Type Register-Register Timing

Clk

PC

Rs, Rt, Rd, Op, Func

ALUctrl

RegWr

busA, B

busW

RegWr

busW

Rw, Ra, Rb

32 32-bit Registers

All register writes occur on falling edge of clock (clocking methodology)
Logical Operations with Immediate Example:
Micro-Operation Sequence For ORI

ori $rt$, $rs$, $imm_{16}$

- Instruction Word $\leftarrow$ Mem[PC]
- $PC \leftarrow PC + 4$
- $R[rt] \leftarrow R[rs] \text{ OR } \text{ZeroExt}[imm_{16}]$

Common Steps
- Fetch the instruction
- Increment PC
- OR register $rs$ with immediate field zero extended to 32 bits, result in register $rt$

Done by Main ALU

Not in book version
Datapath For Logical Instructions With Immediate

\[ R[rt] \leftarrow R[rs] \text{ OR } \text{ZeroExt}[imm16] \]
Load Operations Example:
Micro-Operation Sequence For LW

lw rt, rs, imm16

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>Immediate (imm16)</th>
</tr>
</thead>
</table>

Instruction Word ← Mem[PC]
P ← PC + 4
R[rt] ← Mem[R[rs] + SignExt[imm16]]

Common Steps
Fetch the instruction
Increment PC
Immediate field sign extended to 32 bits and added to register rs to form memory load address, write word at load effective address to register rt
### Additional Datapath Components For Loads & Stores

**Inputs:**
- for address and write (store) data

**Output**
- for read (load) data

Data memory write or update is edge triggered at the end of the cycle (clocking methodology)

**For SignExt[imm16]**
- 16-bit input sign-extended into a 32-bit value at the output

#### Diagram
- **Address**
- **Data memory**
- **Read data**
- **Write data**
- **MemWrite**
- **MemRead**

4th Edition Figure 4.8, page 311
3rd Edition Figure 5.8, page 296

**EECC550 - Shaaban**

#29  Lec # 4  Winter 2012  12-11-2012
**Datapath For Loads**

\[ R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[imm16]] \]
Store Operations Example:
**Micro-Operation Sequence For SW**

sw rt, rs, imm16

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>Immediate (imm16)</th>
</tr>
</thead>
</table>

Instruction Word ← Mem[PC]

PC ← PC + 4

Mem[R[rs] + SignExt[imm16]] ← R[rt]

Fetch the instruction

Increment PC

Immediate field sign extended to 32 bits and added to register rs to form memory store effective address, register rt written to memory at store effective address.

Common Steps

Effective Address

To store at

Data Memory
Datapath For Stores

```
Mem[R[rs] + SignExt[imm16]] ← R[rt]
```
Conditional Branch Example:
Micro-Operation Sequence For BEQ

beq  rs, rt, imm16

Instruction Word ← Mem[PC]
P ← P + 4
Zero ← R[rs] - R[rt]

Condition

Action

Zero : P ← P + (SignExt(imm16) x 4)

“Zero” is zero flag of main ALU
Main ALU evaluates branch condition
New adder to compute branch target:
• Sum of incremented PC and
  sign-extended lower 16-bits on the
  instruction.

Datapath For
Branch Instructions

Main ALU evaluates Branch Condition
(subtract)

Zero ← R[rs] - R[rt]

To branch control logic

Zero flag = 1
if R[rs] - R[rt] = 0
(i.e R[rs] = R[rt])

4th Edition Figure 4.9, page 312 - 3rd Edition Figure 5.9, page 297
More Detailed Datapath For Branch Operations

Branch Zero

Instruction Address

PC+4

Adder

Branch Target

imm16

PC Ext

Sign extend shift left 2

Branch Target ALU

New 32-bit ALU (adder)

New 2X1 32-bit MUX to select next PC value

Main ALU (subtract)

Zero ← R[rs] - R[rt]

Zero

Equal?

Rw Ra Rb

32 32-bit Registers

RegWr

busA

Rs

5

5

Rt

5

R[rs]

busB

32

R[rt]

32

busW

Clk

PC

0

1

Mux

Adder

Adder

4

32
Combining The Datapaths For Memory Instructions and R-Type Instructions

Highlighted multiplexors and connections added to combine the datapaths of memory and R-Type instructions into one datapath

This is book version ORI not supported

4th Edition Figure 4.10 Page 314 - 3rd Edition Figure 5.10 Page 299
Instruction Fetch Datapath Added to
ALU R-Type and Memory Instructions Datapath

This is book version ORI not supported, no zero extend of immediate needed

Combination of Figure 4.10 (p. 314) and Figure 4.6 (p. 309)
[3rd Edition Figure 5.10 (p. 299) and Figure 5.6 (p. 293)]
A Simple Datapath For The MIPS Architecture

Datapath of branches and a program counter multiplexor are added. Resulting datapath can execute in a single cycle the basic MIPS instruction:

- load/store word
- ALU operations
- Branches

This is book version ORI not supported, no zero extend of immediate needed

4th Edition Figure 4.11 page 315 - 3rd Edition Figure 5.11 page 300
Main ALU Control

- The main ALU has four control lines (detailed design in Appendix B) with the following functions:

<table>
<thead>
<tr>
<th>ALU Control Lines</th>
<th>ALU Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>Set-on-less-than</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>

- For our current subset of MIPS instructions only the top five functions will be used (thus only three control lines will be used).
- For R-type instruction the ALU function depends on both the opcode and the 6-bit “funct” function field.
- For other instructions the ALU function depends on the opcode only.
- A local ALU control unit can be designed to accept 2-bit ALUop control lines (from main control unit) and the 6-bit function field and generate the correct 4-bit ALU control lines.
### Local ALU Decoding of “func” Field

#### Instruction Opcode

<table>
<thead>
<tr>
<th>Instruction Opcode</th>
<th>Instruction Operation</th>
<th>ALUOp</th>
<th>Funct Field</th>
<th>Desired ALU Action</th>
<th>ALU Control Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>Load word</td>
<td>00</td>
<td>XXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>SW</td>
<td>Store word</td>
<td>00</td>
<td>XXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>Branch Equal</td>
<td>branch equal</td>
<td>01</td>
<td>XXXXXXX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-Type</td>
<td>add</td>
<td>10</td>
<td>1000000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>R-Type</td>
<td>subtract</td>
<td>10</td>
<td>1000100</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-Type</td>
<td>AND</td>
<td>10</td>
<td>1001000</td>
<td>and</td>
<td>0000</td>
</tr>
<tr>
<td>R-Type</td>
<td>OR</td>
<td>10</td>
<td>1001010</td>
<td>or</td>
<td>0001</td>
</tr>
<tr>
<td>R-Type</td>
<td>set on less than</td>
<td>10</td>
<td>1010100</td>
<td>set on less than</td>
<td>0111</td>
</tr>
</tbody>
</table>

#### Diagram

![Diagram of Local ALU Decoding of “func” Field]

- **Opcode**
- **Main Control**
- **ALU Control (Local)**
- **ALU**
- **Desired ALU Action**
- **ALU Control Lines**

**R-Type** = 10
Local 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>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>111</td>
</tr>
</tbody>
</table>

Add = 00
Subtract = 01
R-type = 10

(2 lines From main control unit)

FIGURE C.2.1  The truth table for the three ALU control bits (called Operation) as a function of the ALUOp and function code field. This table is the same as that shown Figure 5.13.

More details found in Appendix D in Book CD – (3rd Edition Appendix C)
**Single Cycle MIPS Datapath**

Necessary multiplexors and control lines are identified here and local ALU control added:

This is book version ORI not supported, no zero extend of immediate needed

4th Edition Figure 4.15 page 320 - 3rd Edition Figure 5.15 page 305
Putting It All Together: A Single Cycle Datapath

Instruction Memory

<table>
<thead>
<tr>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>Imm16</th>
</tr>
</thead>
<tbody>
<tr>
<td>21:25</td>
<td>19:20</td>
<td>11:15</td>
<td>0:15</td>
</tr>
</tbody>
</table>

ALUop (2-bits)

00 = add
01 = subtract
10 = R-Type

RegDst

<table>
<thead>
<tr>
<th>Rd</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td></td>
</tr>
</tbody>
</table>

MemWr
MemtoReg

ALU Control

Function Field

Zero

ALU

Extender

32 32-bit Registers

<table>
<thead>
<tr>
<th>Rw</th>
<th>Ra</th>
<th>Rb</th>
</tr>
</thead>
</table>

Branch Target

PC+4

Adder

Mux

Clk

PC Src

RegWr

<table>
<thead>
<tr>
<th>Rs</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 5</td>
<td></td>
</tr>
</tbody>
</table>

Data Memory

<table>
<thead>
<tr>
<th>Adr</th>
<th>Data</th>
</tr>
</thead>
</table>

Data In

MemRd

WrEn

Main ALU

Data Memory

<table>
<thead>
<tr>
<th>Adder</th>
<th>PC Ext</th>
</tr>
</thead>
</table>

Branch Zero

e.g. Sign Extend + Shift Left 2

(Includes ORI not in book version)
Instruction Memory

Instruction<31:0>

Op  Fun  Rt  Rs  Rd  Imm16  Jump_target

Control Unit

Control Lines

RegDst  ALUSrc  MemtoReg  RegWrite  MemRead  MemWrite  Branch  ALOp (2-bits)

DATA PATH
## The Effect of The Control Signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Effect when deasserted (=0)</th>
<th>Effect when asserted (=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>The register destination number for the write register comes from the rt field (instruction bits 20:16).</td>
<td>The register destination number for the write register comes from the rd field (instruction bits 15:11).</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>The register on the write register input is written with the value on the Write data input.</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>The second main ALU operand comes from the second register file output (Read data 2) R[rt]</td>
<td>The second main ALU operand is the sign-extended lower 16 bits on the instruction (imm16)</td>
</tr>
<tr>
<td>Branch (BEQ)</td>
<td>The PC is replaced by the output of the adder that computes PC + 4</td>
<td>If Zero =1 The PC is replaced by the output of the adder that computes the branch target.</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Data memory contents designated by the address input are put on the Read data output.</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Data memory contents designated by the address input are replaced by the value on the Write data input.</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>The value fed to the register write data input comes from the main ALU.</td>
<td>The value fed to the register write data input comes from data memory.</td>
</tr>
</tbody>
</table>
### Control Line Settings

<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>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

4th Edition Figure 4.18 page 323
3rd Edition Figure 5.18 page 308
# The Truth Table For The Main Control

<table>
<thead>
<tr>
<th>Control</th>
<th>Signal name</th>
<th>R-format</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op5</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Op4</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Op3</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Op2</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Op1</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Op0</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RegDst</td>
<td></td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ALUSrc</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MemtoReg</td>
<td></td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RegWrite</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemRead</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUOp1</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUOp0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**FIGURE C.2.4** The control function for the simple one-clock implementation is completely specified by this truth table. This table is the same as that shown in Figure 5.22.
PLA Implementation of the Main Control

Figure D.2.5 in Appendix D
(3rd Edition Figure C.2.5 in Appendix C)

PLA = Programmable Logic Array - Appendix C (3rd Edition Appendix B)
In this book version, ORI is not supported—no zero extend of immediate needed.
Adding Support For Jump:
Micro-Operation Sequence For Jump: J

\[ j \text{ jump}_\text{target} \]

**OP**  |  **Jump_target**
--- | ---
2  | 6 bits [31:26]  | 26 bits [25:0]  | Jump address in words

Instruction Word ← Mem[PC]

PC ← PC + 4

PC ← PC(31-28), jump_target, 00

Fetch the instruction

Increment PC

Update PC with jump address

Jump address in words

Jump target

4 bits

26 bits

2 bits

4 highest bits from PC + 4

Jump Address

Common Steps
Datapath For Jump

Instruction(15-0)  
imm16  
PC Ext  
e.g. Sign Extend + Shift Left 2

Instruction(25-0)  
jump_target  
Shift left 2

PC(31-28)  
Jump target  
0 0

Jump Address

PC+4(31-28)  
Branch Target

Mux

0 1

JUMP

Next Instruction Address

Shift left 2

Jump target

4 bits  26 bits  2 bits

PC(31-28)  
jump_target  
0 0

C550 - Shaaban
In this book version, ORI is not supported—no zero extend of immediate needed.
Control Line Settings
(with jump instruction, j added)

<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>
<th>Jump</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>
<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>
<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>
<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>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.18 page 323 (3rd Edition Figure 5.18 page 308) modified to include j
**Worst Case Timing (Load)**

- **Clk-to-Q**
- **PC Old Value**
- **Rs, Rt, Rd, Op, Func**
- **ALUctr**
- **ExtOp**
- **ALUSrc**
- **MemtoReg**
- **RegWr**
- **busA**
- **busB**
- **busW**
- **Instruction Memory Access Time**
- **Delay through Control Logic**
- **Register File Access Time**
- **Delay through Extender & Mux**
- **ALU Delay**
- **Data Memory Access Time**

**Register Write Occurs**
Instruction Timing Comparison

Arithmetic & Logical

- PC  Inst Memory  Reg File  mux  ALU  mux  setup

Load

- PC  Inst Memory  Reg File  mux  ALU  Data Mem  mux  setup

Store

- Critical Path

Branch

- PC  Inst Memory  Reg File  cmp  mux

Jump

- PC  Inst Memory  mux
Simplified Single Cycle Datapath Timing

• Assuming the following datapath/control hardware components delays:
  - Memory Units: 2 ns
  - ALU and adders: 2 ns
  - Register File: 1 ns
  - Control Unit < 1 ns

• Ignoring Mux and clk-to-Q delays, critical path analysis:

  Obtained from low-level target VLSI implementation technology of components

\[ ns = \text{nanosecond} = 10^{-9} \text{ second} \]
Performance of Single-Cycle (CPI=1) CPU

- Assuming the following datapath hardware components delays:
  - Memory Units: 2 ns
  - ALU and adders: 2 ns
  - Register File: 1 ns

- The delays needed for each instruction type can be found:

<table>
<thead>
<tr>
<th>Instruction Class</th>
<th>Instruction Memory</th>
<th>Register Read</th>
<th>ALU Operation</th>
<th>Data Memory</th>
<th>Register Write</th>
<th>Total Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td></td>
<td>1 ns</td>
<td>6 ns</td>
</tr>
<tr>
<td>Load</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td>2 ns</td>
<td>1 ns</td>
<td>8 ns</td>
</tr>
<tr>
<td>Store</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td>2 ns</td>
<td></td>
<td>7 ns</td>
</tr>
<tr>
<td>Branch</td>
<td>2 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td></td>
<td></td>
<td>5 ns</td>
</tr>
<tr>
<td>Jump</td>
<td>2 ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 ns</td>
</tr>
</tbody>
</table>

Load has longest delay of 8 ns thus determining the clock cycle of the CPU to be 8ns

\[ T = I \times CPI \times C \]

\[ C = 8 \text{ ns} \]

- The clock cycle is determined by the instruction with longest delay: The load in this case which is 8 ns. Clock rate = \( 1 / 8 \text{ ns} = 125 \text{ MHz} \)

- A program with \( I = 1,000,000 \) instructions executed takes:
  Execution Time = \( T = I \times CPI \times C = 10^6 \times 1 \times 8 \times 10^{-9} = 0.008 \text{ s} = 8 \text{ msec} \)
Adding Support for jal to Single Cycle Datapath

• The MIPS jump and link instruction, jal is used to support procedure calls by jumping to jump address (similar to j) and saving the address of the following instruction PC+4 in register $ra ($31)

jal Address

• jal uses the j instruction format:

<table>
<thead>
<tr>
<th>op (6 bits)</th>
<th>Target address (26 bits)</th>
</tr>
</thead>
</table>

• We wish to add jal to the single cycle datapath in Figure 4.24 page 329 (3rd Edition Figure 5.24 page 314). Add any necessary datapaths and control signals to the single-clock datapath and justify the need for the modifications, if any.

• Specify control line values for this instruction.
jump and link, jal support to Single Cycle Datapath

Instruction Word ← Mem[PC]
R[31] ← PC + 4
PC ← Jump Address

1. Expand the multiplexor controlled by RegDst to include the value 31 as a new input.
2. Expand the multiplexor controlled by MemtoReg to have PC+4 as a new input.
### Adding Control Lines Settings for jal

(For Textbook Single Cycle Datapath including Jump)

#### Instruction Word

- **Instruction Word**: \( \text{Mem}[PC] \)
- **R[31]**: \( \text{PC + 4} \)
- **PC**: \( \text{Jump Address} \)

#### Table

<table>
<thead>
<tr>
<th></th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>01</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>00</td>
<td>1</td>
<td>01</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>xx</td>
<td>1</td>
<td>xx</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>beq</td>
<td>xx</td>
<td>0</td>
<td>xx</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>xx</td>
<td>x</td>
<td>xx</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>JAL</td>
<td>10</td>
<td>x</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Notes

- **RegDst**: 2 bits
- **MemtoReg**: 2 bits
- **R[31]**: \( PC + 4 \)
- **PC**: \( \text{Jump Address} \)
Adding Support for LWR to Single Cycle Datapath

• We wish to add a variant of lw (load word) let’s call it LWR to the single cycle datapath in Figure 4.24 page 329 (3rd Edition Figure 5.24 page 314).

  LWR  $rd, $rs,  $rt

• The LWR instruction is similar to lw but it sums two registers (specified by $rs, $rt) to obtain the effective load address and uses the R-Type format.

• Add any necessary datapaths and control signals to the single cycle datapath and justify the need for the modifications, if any.

• Specify control line values for this instruction.
LWR (R-format LW) support to Single Cycle Datapath

Instruction Word $\leftarrow$ Mem[PC]
PC $\leftarrow$ PC + 4
R[rd] $\leftarrow$ Mem[ R[rs] + R[rt] ]

No new components or connections are needed for the datapath just the proper control line settings

Adding Control Lines Settings for LWR
(For Textbook Single Cycle Datapath including Jump)

<table>
<thead>
<tr>
<th></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>
<th>Jump</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>
<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>
<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>
<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>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>LWR</td>
<td>1</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>
</tbody>
</table>

rd R[rt]
Adding Support for jm to Single Cycle Datapath

- We wish to add a new instruction jm (jump memory) to the single cycle datapath in Figure 4.24 page 329 (3rd Edition Figure 5.24 page 314).

  \[ \text{jm \ offset($rs$)} \]

- The jm instruction loads a word from effective address (R[rs] + offset), this is similar to lw except the loaded word is put in the PC instead of register $rt$.
- Jm used the I-format with field rt not used.

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>address (imm16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>Not Used 16 bits</td>
</tr>
</tbody>
</table>

- Add any necessary datapaths and control signals to the single cycle datapath and justify the need for the modifications, if any.
- Specify control line values for this instruction.
Adding jump memory, jm support to Single Cycle Datapath

Instruction Word ← Mem[PC]
PC ← Mem[R[rs] + SignExt[imm16]]

1. Expand the multiplexor controlled by Jump to include the Read Data (data memory output) as new input 2. The Jump control signal is now 2 bits

![Diagram of the datapath with jump memory support.]
## Adding jm support to Single Cycle Datapath

### Adding Control Lines Settings for jm

(For Textbook Single Cycle Datapath including Jump)

<table>
<thead>
<tr>
<th></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>
<th>Jump</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>
<td>00</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>
<td>00</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>
<td>00</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>
<td>00</td>
</tr>
<tr>
<td>J</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>01</td>
</tr>
<tr>
<td>Jm</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Jump is now 2 bits

PC $\leftarrow$ Mem[R[rs] + SignExt[imm16]]
Drawbacks of Single Cycle Processor

1. Long cycle time:
   - All instructions must take as much time as the slowest
     • Here, cycle time for load is longer than needed for all other instructions.
     - Cycle time must be long enough for the load instruction:
       PC’s Clock -to-Q + Instruction Memory Access Time +
       Register File Access Time + ALU Delay (address calculation) +
       Data Memory Access Time + Register File Setup Time + Clock Skew

   - Real memory is not as well-behaved as idealized memory
     • Cannot always complete data access in one (short) cycle.

2. Impossible to implement complex, variable-length instructions and complex addressing modes in a single cycle.
   - e.g indirect memory addressing.  

3. High and duplicate hardware resource requirements
   - Any hardware functional unit cannot be used more than once in a single cycle (e.g. ALUs).

4. Does not allow overlap of instruction processing (instruction pipelining, chapter 6).