Lecture 10
The MIPS Datapath

The organization of a computer

Von Neumann Model:
- Stored-program machine instructions are represented as numbers
- Programs can be stored in memory to be read/written just like numbers.

Suggested Readings
- Readings
  - H&P: Chapter 4.1-4.4

Goal:
Describe the fundamental components required in a single core of a modern microprocessor as well as how they interact with each other, with main memory, and with external storage media.
Review: Functions of Each Component

- Datapath: performs data manipulation operations
  - arithmetic logic unit (ALU)
  - floating point unit (FPU)
- Control: directs operation of other components
  - finite state machines
  - micro-programming
- Memory: stores instructions and data
  - random access v.s. sequential access
  - volatile v.s. non-volatile
  - RAMs (SRAM, DRAM), ROMs (PROM, EEPROM), disk
  - tradeoff between speed and cost/bit
- Input/Output and I/O devices: interface to environment
  - mouse, keyboard, display, device drivers

The MIPS Subset

- To simplify things a bit we’ll just look at a few instructions:
  - memory-reference: lw, sw
  - arithmetic-logical: add, sub, and, or, slt
  - branching: beq, j

- Organizational overview:
  - fetch an instruction based on the content of PC
  - decode the instruction
  - fetch operands
    - (read one or two registers)
  - execute
    - (effective address calculation/arithmetic-logical operations/comparison)
  - store result
    - (write to memory / write to register / update PC)

With Von Neumann, RISC model do similar things for each instruction

Most common instructions

Board discussion:

- Let’s derive the MIPS datapath...

Implementation Overview

- Abstract / Simplified View:
  simplest view of Von Neumann, RISC μP

- 2 types of signals:
  - Data and control

- Clocking strategy:
  - Derived datapath is single cycle; did not talk about internal storage
Single Cycle Implementation

- Each instruction takes one cycle to complete.
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
- Cycle time determined by length of the longest path
**Review: Derivation of Single Cycle Datapath**

- Take bits from instruction encoding in IR and send to different parts of datapath
  - e.g. R-type, Add encoding:

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sham</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- During Decode...
  - Instruction format:
    - RTL:
      - Instruction fetch: mem[PC]
      - Go to next instruction: Pc <- PC+ 4
  - Ra, Rb and Rw are from instruction's rs, rt, rd fields → sort of like passing args into a function.
  - Actual ALU operation and register write should occur after decoding the instruction.

**Instruction Fetch Unit**

- Fetch the instruction: mem[PC] ,
- Update the program counter:
  - sequential code: PC <- PC+4
  - branch and jump: PC <- “something else”

**Let's say we want to fetch...**

...an R-type instruction (arithmetic)

- Instruction format:

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sham</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- RTL:
  - Instruction fetch: mem[PC]
  - Go to next instruction: Pc <- PC+ 4

- So IR ← Memory(PC)
**Datapath for R-Type Instructions**

- **Register timing:**
  - Register can always be read.
  - Register write only happens when RegWr is set to high and at the falling edge of the clock
- What does this say about CC time?

**Datapath for I-Type A/L Instructions**

- Destination registers are in different places in encoding so need a mux (rd[15-11] vs. rt[20-16])

**I-Type Arithmetic/Logic Instructions**

- **Instruction format:**

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>rs</td>
<td>rt</td>
<td>Address/Immediate value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- RTL for arithmetic operations: e.g., ADDI
  - Instruction fetch: mem[PC]
  - Go to next instruction: Pc <- PC+ 4

**I-Type Load/Store Instructions**

- **Instruction format:**

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>rs</td>
<td>rt</td>
<td>Address/Immediate value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- RTL for load/store operations: e.g., LW
  - Instruction fetch: mem[PC]
  - Compute memory address: Addr <- reg[rs] + SignExt(imm16)
  - Load data into register: reg[rt] <- mem[Addr]
  - Go to next instruction: Pc <- PC+ 4

- How about store? [same thing, just make 3rd step mem[addr] <- reg[rt]]
### Datapath for Load/Store Instructions

What happens here?

- **For lw/sw send address from ALU to data memory**
- **For lw, need to send data to register file**
- **For sw, need to send data to memory**

### I-Type Branch Instructions

- **Instruction format:**
  - Instruction fetch: \( \text{mem[PC]} \)
  - Compute conditon: \( \text{Cond} \leftarrow \text{reg[rs]} - \text{reg[rt]} \)
  - Calculate the next instruction's address:
    
    \[
    \text{if (Cond eq 0)} \quad \text{then} \\
    \quad \text{PC} \leftarrow \text{PC} + 4 + \text{SignExd(imm16)} \times 4 \\
    \text{else ?}
    \]

- **RTL for branch operations: e.g., BEQ**
  - Instruction fetch: \( \text{mem[PC]} \)
  - Compute condition: \( \text{Cond} \leftarrow \text{reg[rs]} - \text{reg[rt]} \)
  - Calculate the next instruction's address:
    
    \[
    \text{if (Cond eq 0)} \quad \text{then} \\
    \quad \text{PC} \leftarrow \text{PC} + 4 + \text{SignExd(imm16)} \times 4 \\
    \text{else ?}
    \]

### Datapath for Branch Instructions

- **Next Address Logic**
  - \( \text{ADD} \quad \text{MUX} \quad \text{SignExt} \quad \text{Instruction Memory} \)
  - \( \text{imm16} \quad \text{Branch} \quad \text{Zero} \)
  - \( \text{PC} \) contains \( \text{PC} + 4 \)
  - May not want to change \( \text{PC} \) if BEQ condition not met
  - (implicitly says: “this stuff happens anyway so we have to be sure we don’t change things we don’t want to change”)
  - If branch instruction \( \text{AND 0} \), can automatically generate control signal
A Single Cycle Datapath

Let’s trace a few instructions:

- **Add** $5, $6, $7
- **SW** 0($9), $10
- **Sub** $1, $2, $3
- **LW** $11, 0($12)
Single cycle versus multi-cycle

### Single-Cycle Implementation

- **Single-cycle, fixed-length clock:**
  - \(\text{CPI} = 1\)
  - Clock cycle = propagation delay of the longest datapath operations among all instruction types
  - Easy to implement

- **How to determine cycle length?**
- **Calculate cycle time assuming negligible delays except:**
  - memory (2ns), ALU and adders (2ns), register file access (1ns)

  - R-type: \(\text{max}\{\text{mem} + \text{RF} + \text{ALU} + \text{RF}, \text{Add}\}\) = 6ns
  - LW: \(\text{max}\{\text{mem} + \text{RF} + \text{ALU} + \text{mem} + \text{RF}, \text{Add}\}\) = 8ns
  - SW: \(\text{max}\{\text{mem} + \text{RF} + \text{ALU} + \text{mem}, \text{Add}\}\) = 7ns
  - BEQ: \(\text{max}\{\text{mem} + \text{RF} + \text{ALU}, \text{max}\{\text{Add}, \text{mem} + \text{Add}\}\}\) = 5ns

### Multiple Cycle Alternative

- **Break an instruction into smaller steps**
- **Execute each step in one cycle.**
- **Execution sequence:**
  - Balance amount of work to be done
  - Restrict each cycle to use only one major functional unit
  - At the end of a cycle
    - Store values for use in later cycles
    - Introduce additional “internal” registers
- **The advantages:**
  - Cycle time much shorter
  - Diff. inst. take different # of cycles to complete
  - Functional unit used more than once per instruction

But before, datapath was “multi-cycle”….
A Multiple Cycle MIPS Datapath

Five Step Execution

1. Instruction Fetch (IFetch):
   - Fetch instruction at address ($PC$)
   - Store the instruction in register $IR$  
   - Increment PC

2. Instruction Decode and Register Read (Decode):
   - Decode the instruction type and read register
   - Store the register contents in registers $A$ and $B$
   - Compute new PC address and store it in $ALUOut$

3. Execution, Memory Address Computation, or Branch Completion (Execute):
   - Compute memory address (for LW and SW), or
   - Perform R-type operation (for R-type instruction), or
   - Update PC (for Branch and Jump)
   - Store memory address or register operation result in $ALUOut$

4. Memory Access or R-type instruction completion (MemRead/RegWrite/MemWrite):
   - Read memory at address $ALUOut$ and store it in $MDR$
   - Write $ALUOut$ content into register file, or
   - Write memory at address $ALUOut$ with the value in $B$

5. Write-back step (WrBack):
   - Write the memory content read into register file

Execution Sequence Summary

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>$IR = M[PC]$, $PC = PC + 4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td>$A = RF[R[25:21]]$, $B = RF[R[20:16]]$, $ALUOut = PC + (sign-extend(R[1:0]) &lt;&lt; 2)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>$ALUOut = A \text{ op } B$</td>
<td>$ALUOut = A + \text{ sign-extend } (R[15:0])$</td>
<td>if $A = B$ then $PC = ALUOut$</td>
<td>$PC = PC[31:28]</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>$RF[R[15:11]] = ALUOut$</td>
<td>Load: $MDR = M[ALUOut]$ or Store: $M[ALUOut] = B$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: $RF[R[20:16]] = MDR$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Number of cycles for an instruction:
  - R-type: 4
  - lw: 5
  - sw: 4
  - Branch or Jump: 3
Some Simple Questions

- How many cycles will it take to execute this code?
  
  lw $t2, 0($t3)
  lw $t3, 4($t3)
  beq $t2, $t3, Label #assume branch is not taken
  add $t5, $t2, $t3
  sw $t5, 8($t3)

  Label: ...

  5+5+3+4+4=21

- What is being done during the 8th cycle of execution?
  Compute memory address: 4+$t3

- In what cycle does the actual addition of $t2 and $t3 takes place? 16

  What if multi-cycle clock period is 2 ns vs. 8 ns for a single cycle?

Multiple Cycle Design

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit

- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers
Control Logic

(i.e. now, we need to make the HW do what we want it to do - add, subtract, etc. - when we want it to...)

Implementing the Control (Part 1)

- Implementation Steps:
  - Identify control inputs and control output (control words)
  - Make a control signal table for each cycle
  - Derive control logic from the control table

- Do we need a FSM here?

This logic can take on many forms: combinational logic, ROMs, microcode, or combinations...

Control inputs:
- Opcode (5 bits)
- Func (6 bits)

Control outputs:
- RegDst
- MempToReg
- RegWrite
- MemRead
- MemWrite
- ALUSrc
- ALUctrl
- Branch
- Jump

Implementing Control

- Implementation Steps:
  1. Identify control inputs and control outputs
  2. Make a control signal table for each cycle
  3. Derive control logic from the control table

- This logic can take on many forms: combinational logic, ROMs, microcode, or combinations...
**Single Cycle Control Input/Output**

- **Control Inputs:**
  - Opcode (6 bits)
  - How about R-type instructions?

- **Control Outputs:**
  - RegDst
  - ALUSrc
  - MemtoReg
  - RegWrite
  - MemWrite
  - Branch
  - Jump
  - ALUctr

---

**Control Signal Table**

<table>
<thead>
<tr>
<th>Add</th>
<th>Sub</th>
<th>LW</th>
<th>SW</th>
<th>BEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>100010</td>
<td>xxxxxx</td>
<td>xxxxxx</td>
<td>xxxxxx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op (input)</th>
<th>Add</th>
<th>Sub</th>
<th>LW</th>
<th>SW</th>
<th>BEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Mem-to-Reg</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Reg. Write</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mem. Read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Mem. Write</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUOp</td>
<td>Add</td>
<td>Sub</td>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
</tbody>
</table>

**Main control, ALU control**

- Use **OP** field to generate **ALUOp** (encoding)
  - Control signal fed to ALU control block
- Use **Func** field and **ALUOp** to generate **ALUctr** (decoding)
  - Specifically sets 3 ALU control signals
    - B-Invert, Carry-in, operation
Main control, ALU control

- We want these outputs:
  - ALU Operation and or add sub sht
  - ALUctr<2:0> 000 001 101 111

  - ALUctr<2> = B-negate (C-in & B-invert)
  - ALUctr<1> = Select ALU Output
  - ALUctr<0> = Select ALU Output

  - Invert B and C-in must be a 1 for subtract

- We have these inputs...

  - func<5:0>
  - ALUOp<1:0> 10 00 00 01

  - We have 8 bits of input to our ALU control block; we need 3 bits of output...

Generating ALUctr

The Logic

- This table is used to generate the actual boolean logic gates that produce ALUctr.

  - Could generate gates by hand, often done w/SW.

- R-type: lw/sw beq

Recall...
Well, here’s what we did...

We came up with the information to generate this logic which would fit here in the datapath.

Implementing the Control (Part 2)

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

- How to represent all the information?
  - finite state diagram
  - microprogramming

- Realization of a control unit is independent of the representation used
  - Control outputs: random logic, ROM, PLA
  - Next-state function: same as above or an explicit sequencer

Microprogramming as an Alternative

- Control unit can easily reach thousands of states with hundreds of different sequences.
  - A large set of instructions and/or instruction classes (x86)
  - Different implementations with different cycles per instruction

- Flexibility may be needed in the early design phase

- An alternative: Microcode.
  - Treat the set of control signals to be asserted in a state as an instruction to be executed (referred to as microinstructions)
  - Treat state transitions as an instruction sequence
Microprogramming as an Alternative (cont’d)

- Each state \(\rightarrow\) one microinstruction
- State transitions \(\rightarrow\) microinstruction sequencing
- Setting up control signals \(\rightarrow\) executing microinstructions
- To specify control, we just need to write microprograms (or microcode)

Microinstruction Format (1)

- Group the control signals according to how they are used
- For the 5-cycle MIPS organization:
  - Memory: IorD, MemRead, MemWrite
  - Instruction Register: IRWrite
  - PC: PCWrite, PCWriteCond, PCSource
  - Register File: RegWrite, MemtoReg, RegDst
  - ALU: ALUSrcA, ALUSrcB, ALUOp
- Group them as follows:
  - Memory (for both Memory and Instruction Register)
  - PC write control (for PC)
  - Register control (for Register File)
  - ALU control
  - SRC1
  - SRC2
  - Sequencing

Microinstruction Format (2)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Value</th>
<th>Signals active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU control</td>
<td>Add</td>
<td>ALUOp = 00</td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td>Sub</td>
<td>ALUOp = 01</td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
<td></td>
</tr>
<tr>
<td>Func code</td>
<td>ALUOp = 10</td>
<td>Use the instruction's func to determine ALU control.</td>
<td></td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>ALUSrcA = 0</td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td>A</td>
<td>ALUSrcA = 1</td>
<td>Register A is the first ALU input.</td>
<td></td>
</tr>
<tr>
<td>SRC2</td>
<td>B</td>
<td>ALUSrcB = 0</td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td>4</td>
<td>ALUSrcB = 01</td>
<td>Use 4 as the second ALU input.</td>
<td></td>
</tr>
<tr>
<td>Extshift</td>
<td>ALUSrcB = 10</td>
<td>Use output of the sign ext unit as the 2nd ALU input.</td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>Read</td>
<td>RegWrite, RegDst = 1, MemtoReg=0</td>
<td>Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td>MDR</td>
<td>Write</td>
<td>RegWrite, RegDst = 0, MemtoReg=1</td>
<td>Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.</td>
</tr>
</tbody>
</table>
### Microinstruction Format (3)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Value</th>
<th>Signals active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read PC</td>
<td>MemRead, lorD = 0</td>
<td>Read memory using the PC as address; write result into IR (and the MDR).</td>
<td></td>
</tr>
<tr>
<td>Read ALU</td>
<td>MemRead, lorD = 1</td>
<td>Read memory using the ALUOut as address; write result into MDR.</td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td>MemWrite, lorD = 1</td>
<td>Write memory using the ALUOut as address, contents of B as the data.</td>
<td></td>
</tr>
<tr>
<td>ALU write control</td>
<td>PCSource 00 PCWrite</td>
<td>Write the output of the ALU into the PC.</td>
<td></td>
</tr>
<tr>
<td>ALUOut cond</td>
<td>PCSource=01, If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump address</td>
<td>PCSource=10, Write the PC with the jump address from the instruction.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequencing</td>
<td>AddrCtl = 11</td>
<td>Choose the next microinstruction sequentially.</td>
<td></td>
</tr>
<tr>
<td>Fetch</td>
<td>AddrCtl = 00</td>
<td>Go to the first microinstruction to a new instruction.</td>
<td></td>
</tr>
<tr>
<td>Dispatch 1</td>
<td>AddrCtl = 01</td>
<td>Dispatch using the ROM 1.</td>
<td></td>
</tr>
<tr>
<td>Dispatch 2</td>
<td>AddrCtl = 10</td>
<td>Dispatch using the ROM 2.</td>
<td></td>
</tr>
</tbody>
</table>

### Sample Microinstruction (1)

**IFetch:** IR = Mem[PC], PC = PC+4

- **PCWrite:** 1
- **PCWriteCond:** lorD = 0
- **MemRead:** 1
- **IRWrite:** 1
- **MementoReg:**
- **PCSource:**
  - **ALUOp:** 00
  - **ALUSrcB:** 01
  - **ALUSrcA:** 0
- **RegWrite:**
- **RegDst:**
- **AddrCtl:** 11

**Microinstruction:**

<table>
<thead>
<tr>
<th>ALUctrl</th>
<th>SRC1</th>
<th>SRC2</th>
<th>RegCtrl</th>
<th>Memory</th>
<th>PCWrite</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>--</td>
<td>ReadPC</td>
<td>ALU</td>
<td>Seq</td>
</tr>
<tr>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read</td>
<td></td>
<td></td>
<td>Dispatch 1</td>
</tr>
<tr>
<td>Mem1</td>
<td>Add A</td>
<td>Extend</td>
<td></td>
<td></td>
<td></td>
<td>Dispatch 2</td>
</tr>
<tr>
<td>LW2</td>
<td></td>
<td></td>
<td>Read ALU</td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td>SW2</td>
<td></td>
<td></td>
<td>Write MDR</td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>Rformat1</td>
<td>Func code A B</td>
<td>Write ALU</td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>BEQ1</td>
<td>Sub A B</td>
<td>ALUOut-cond</td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>JUMP1</td>
<td></td>
<td></td>
<td>Jump address</td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
</tbody>
</table>

### Sample Microinstruction (2)

Decode: A = RF[IR[25:21]], B = RF[IR[20:16]], ALUOut = PC + Sign_Ext(IR[15:0]) << 2;

**Microinstruction:**

<table>
<thead>
<tr>
<th>ALUctrl</th>
<th>SRC1</th>
<th>SRC2</th>
<th>RegCtrl</th>
<th>Memory</th>
<th>PCWrite</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>PC</td>
<td>ExtShft</td>
<td>Read</td>
<td></td>
<td></td>
<td>Disp 1</td>
</tr>
</tbody>
</table>

### Put It All Together

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU control</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Register control</th>
<th>Memory</th>
<th>PCWrite control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Add PC Extshft</td>
<td>Read</td>
<td>ALU</td>
<td>Seq</td>
<td>Dispatch 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem1</td>
<td>Add A Extend</td>
<td>Read</td>
<td>ALU</td>
<td>Seq</td>
<td>Dispatch 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW2</td>
<td></td>
<td>Read</td>
<td>ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW2</td>
<td></td>
<td>Write MDR</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rformat1</td>
<td>Func code A B</td>
<td>Write ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ1</td>
<td>Sub A B ALUOut-cond</td>
<td>ALUOut-cond</td>
<td>Seq</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP1</td>
<td></td>
<td>Jump address</td>
<td>Fetch</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control Implementations

Could use a programmable ROM

Exceptions

- Exceptions: unexpected events from within the processor
  - arithmetic overflow
  - undefined instruction
  - switching from user program to OS
- Interrupts: unexpected events from outside of the processor
  - I/O request
- Consequence: alter the normal flow of instruction execution
- Key issues:
  - detection
  - action
    - save the address of the offending instruction in the EPC
    - transfer control to OS at some specified address
- Exception type indication:
  - status register
  - interrupt vector

Exception Handling

- Types of exceptions considered:
  - undefined instruction
  - arithmetic overflow
- MIPS implementation:
  - EPC: 32-bit register, EPCWrite
  - Cause register: 32-bit register, CauseWrite
    - undefined instruction: Cause register = 0
    - arithmetic overflow: Cause register = 1
  - IntCause: 1 bit control
  - Exception Address: C0000000 (hex)
- Detection:
  - undefined instruction: op value with no next state
  - arithmetic overflow: overflow from ALU
- Action:
  - set EPC and Cause register
  - set PC to Exception Address

FSM with Exception Handling