**CS4100: 計算機結構**

**Designing a Multicycle Processor**

國立清華大學資訊工程學系
九十三學年度第一學期

Adapted from class notes of D. Patterson
Copyright 1998, 2000 UCB

---

**Outline**

- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions

---

**Recap: A Single-Cycle Processor**

---

**What's Wrong with Single-cycle?**

**Arithmetic & Logical**

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>ALU</th>
<th>MUX setup</th>
</tr>
</thead>
</table>

**Load**

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>ALU</th>
<th>Data Mem</th>
<th>MUX setup</th>
</tr>
</thead>
</table>

**Store**

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>ALU</th>
<th>Data Mem</th>
</tr>
</thead>
</table>

**Branch**

<table>
<thead>
<tr>
<th>PC</th>
<th>Inst Memory</th>
<th>Reg File</th>
<th>ALU</th>
<th>Data Mem</th>
</tr>
</thead>
</table>

- Long cycle time
- All instructions take same time as the slowest
- Real memory is not so ideal
  - cannot always get job done in one (short) cycle
- A FU can only be used once => higher cost
Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
  - Multicycle datapath
  - Multicycle execution steps
  - Multicycle control (Appendix C.3)
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions

Multicycle Implementation

- Reduce cycle time
- Diff. Inst. take diff. cycles
- Share functional units

Multicycle Approach

- 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

Partition Single-Cycle Datapath

- Add registers between smallest steps
- RF access
- ALU operation
- memory access
Multicycle Datapath

- 1 memory (instr. & data), 1 ALU (addr, PC+4, add,...), registers (IR, MDR, A, B, ALUOut)
- Storage for subsequent inst. (arch.-visible) vs. storage for same inst. but in a subsequent cycle

Fig. 5.25
Multicycle Design-8

Multicycle Datapath for Basic Instr.

- IR needs write control, but others don’t
- MUX to select 2 sources to memory; memory needs read signal
- PC and A to one ALU input; four sources to another input

Fig. 5.27

Adding Branch/Jump

- Three sources to PC
- Two PC write signals

Fig. 5.28
Multicycle Design-10

Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
  - Multicycle datapath
  - Multicycle execution steps
  - Multicycle control (Appendix C.3)
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions

Multicycle Design-11

Multicycle Design-9

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type Instruction Completion
- Memory Read Completion (Write-back)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register (IR)
- Increment the PC by 4 and put the result back in the PC
- Can be described succinctly using RTL (Register-Transfer Language)

```
IR = Memory[PC];
PC = PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case needed
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

We aren't setting any control lines based on the instruction type yet (we are busy "decoding" it in control logic)

Step 3: Execution

ALU is performing one of three functions, based on instruction type:

- Memory Reference:
  
  `ALUOut = A + sign-extend(IR[15-0]);`

- R-type:
  
  `ALUOut = A op B;`

- Branch:
  
  `if (A==B) PC = ALUOut;`
**Step 4: R-type or Memory-access**

- Loads and stores access memory
  
  \[
  MDR = \text{Memory[ALUOut]}; \]
  
  or
  
  \[
  \text{Memory[ALUOut]} = B; \]

- R-type instructions finish
  
  \[
  \text{Reg[IR[15-11]]} = \text{ALUOut}; \]

  *The write actually takes place at the end of the cycle on the edge*

---

**Step 5: Write-back**

- Loads write to register
  
  \[
  \text{Reg[IR[20-16]]} = \text{MDR}; \]

  *What about all the other instructions?*

---

**Summary of the Steps**

<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>\text{IR = Memory[PC]}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td>\text{A = Reg[IR[25-21]]}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\text{B = Reg[IR[20-16]]}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\text{ALUOut} = A \text{op B}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/ jump completion</td>
<td>\text{ALUOut} = A + \text{sign}-extend (IR[15-0]) \text{if (A \text{op} B)} \text{then} \text{PC} = \text{ALUOut}</td>
<td>\text{PC} = \text{PC}[31-28] II (IR[25-0]) \text{if (A \text{op} B)} \text{then}</td>
<td>\text{PC} = \text{PC}[31-28] II (IR[25-0]) \text{if (A \text{op} B)} \text{then}</td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>\text{Reg[IR[15-11]]} = ALUOut</td>
<td>Load: MDR = \text{Memory[ALUOut]} or Store: \text{Memory[ALUOut]} = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: \text{Reg[IR[20-16]]} = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Cycle 1 of add**

\[
\text{IR} = \text{Memory[PC]} ; \quad \text{PC} = \text{PC} + 4 ;
\]

![Diagram](https://via.placeholder.com/150)

*Fig. 5.30*
Cycle 2 of add

A = Reg[IR[25-21]]; B = Reg[IR[20-16]]; ALUOut = PC+(sign-ext(IR[15-0])<<2);

Cycle 3 of add

ALUOut = A op B;

Cycle 4 of add

Reg[IR[15-11]] = ALUOut;

Simple Question

♦ How many cycles will it take to execute this code?
  lw $t2, 0($t3)
  lw $t3, 4($t3)
  beq $t2, $t3, Label
  add $t5, $t2, $t3
  sw $t5, 8($t3)
  Label: ...

♦ What is going on during the 8th cycle of execution?
♦ In what cycle does the actual addition of $t2 and $t3 takes place?
Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
  - Multicycle datapath
  - Multicycle execution steps
  - Multicycle control (Appendix C.3)
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions

Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
    - Control must specify both the signals to be set in any step
      and the next step in the sequence
- Control specification
  - Use a finite state machine (graphically)
  - Use microprogramming
- Implementation can be derived from the specification and use gates, ROM, or PLA

Controller Design: An Overview

- Several possible initial representations, sequence control and logic representation, and control implementation => all may be determined indep.

Initial Rep.

Sequencing Control

Logic Rep.

Implementation

Finite State Diagram

Explicit Next State Function

Microprogram

Microprogram Counter + Dispatch ROMs

Logic Equations

Truth Tables

PLA

ROM

“hardwired control”

“microprogrammed control”

Review: Finite State Machines

- Finite state machines:
  - a set of states and
  - next state (set by current state and input)
  - output (set by current state and possibly input)

- We will use a Moore Machine (output based only on the current state)
## Our Control Model

- State specifies control points for RT
- Transfer at exiting state (same falling edge)
- One state takes one cycle

![Control Logic Diagram](image)

## Summary of the Steps

<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 = Memory[PC]</td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>A = Reg[IR[25:21]]</td>
<td>B = Reg[IR[20:16]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decode/register fetch</td>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15:0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15:0])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>computation, branch/</td>
<td></td>
<td>if (A — B) then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td>PC = ALUOut</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type</td>
<td></td>
<td>Load: MDR = Memory[ALUOut]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td>or Store: Memory[ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td></td>
<td>Load: Reg[IR[20:16]] = MDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Flowchart](image)

## Control Specification for Multicycle

- **Instruction fetch**
  - IR = MEM[PC]
  - PC = PC + 4

- **Decode/register fetch**
  - A = R[r]
  - B = R[r]
  - S = PC + sx(imm16)/60

- **Execute**
  - If zero: PC = S
  - lw: PC = IR ...
  - sw: PC = S
  - beq: PC = S
  - Jump

- **Memory access**
  - R[rd] = S
  - M = MEM[S]
  - MEM[S] = B

- **Memory read**
  - R[rt] = M

![Control Flowchart](image)

## Organization of Multicycle Processor

![Processor Diagram](image)
Control Signals

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted</th>
<th>Effect when asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSrcA</td>
<td>1st ALU operand = PC</td>
<td>1st ALU operand = Reg[rs]</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>Reg file is written</td>
</tr>
<tr>
<td>MemToReg</td>
<td>Reg. data input = ALU</td>
<td>Reg. write data input = MDR</td>
</tr>
<tr>
<td>MemDst</td>
<td>Reg. write dest. no. = rt</td>
<td>Reg. write dest. no. = rd</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Memory at address is read</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Memory at address is written</td>
</tr>
<tr>
<td>Ird</td>
<td>Memory address = PC</td>
<td>Memory address = ALUout</td>
</tr>
<tr>
<td>IRWrite</td>
<td>None</td>
<td>IR = Memory</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>PC = PC+</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>If zero then PC = PCSource</td>
</tr>
</tbody>
</table>

Mapping RT to Control Signals

- Instruction fetch and decode portion of every instruction is identical:

- Instruction decode/ Register fetch

- Memory reference FSM (Figure 5.38)
- R-type FSM (Figure 5.39)
- Branch FSM (Figure 5.40)
- Jump FSM (Figure 5.41)

Complete FSM

- State number assignment
From FSM to Truth Table

- Please reference the logic equations in Fig. C.3.3 and the truth table in Fig. C.3.6

<table>
<thead>
<tr>
<th>Output</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCWrite</td>
<td>state0 + state9</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>state8</td>
</tr>
<tr>
<td>lORD</td>
<td>state3 + state5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NextState0</th>
<th>Output</th>
<th>Current states</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>NextState1</td>
<td>PCWrite</td>
<td>1 0 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>NextState2</td>
<td>PCWriteCond</td>
<td>0 0 0 0 0 0 0 1 0 0</td>
</tr>
<tr>
<td>NextState3</td>
<td>lORD</td>
<td>0 0 0 1 0 1 0 0 0 0</td>
</tr>
</tbody>
</table>

Designing FSM Controller

Truth Table

Control signals

The Control Unit

ROM Implementation

- Need a ROM of 10-bit address, 20-bit word
  (16-bit datapath control, 4-bit next state)

Address: ROM content

Lower 4 bits of address | Bits 19-4 of word
---|---
0000 | 1001 1100 0001 0000 0000 0000 1100
0001 | 0000 0000 0000 0000 0000 0000 0000
0010 | 0000 0000 0000 0000 0000 0000 0000
0011 | 0000 0000 0000 0000 0000 0000 0000
0100 | 0000 0000 0000 0000 0000 0000 0000
0101 | 0000 0000 0000 0000 0000 0000 0000
0110 | 0000 0000 0000 0000 0000 0000 0000
0111 | 0000 0000 0000 0000 0000 0000 0000
1000 | 0000 0000 0000 0000 0000 0000 0000
1001 | 0000 0000 0000 0000 0000 0000 0000
1010 | 0000 0000 0000 0000 0000 0000 0000
1011 | 0000 0000 0000 0000 0000 0000 0000
1100 | 0000 0000 0000 0000 0000 0000 0000
1101 | 0000 0000 0000 0000 0000 0000 0000
1110 | 0000 0000 0000 0000 0000 0000 0000
1111 | 0000 0000 0000 0000 0000 0000 0000
**ROM Implementation (cont.)**

- Rather wasteful, since for lots of entries, outputs are same or are don’t-care
- Could break up into two smaller ROMs (Fig. C.3.7, C.3.8)

**ROM vs PLA**

- ROM: use two smaller ROMs (Fig. C.3.7, C.3.8)
  - 4 state bits give the 16 outputs, $2^4 \times 16$ bits of ROM
  - 10 bits give 4 next state bits, $2^{10} \times 4$ bits of ROM
  - Total = 4.3K bits of ROM (compared to $2^{10} \times 20$ bits of single ROM implementation)
- PLA is much smaller
  - can share product terms
  - only need entries that produce an active output
  - can take into account don’t-cares
  - Size is ($\#inputs \times \#product-terms) + (\#outputs \times \#product-terms)$
    For this example = $(10 \times 17) + (20 \times 17) = 460$ PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

**Complete FSM**

- State number assignment
Use Counter for Sequence Control

Address Select Unit

Control Contents

Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
  - A multicycle implementation
    - Multicycle datapath
    - Multicycle execution steps
    - Multicycle control
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions
**Microprogrammed Controller**

- Control is the hard part of processor design
  - Datapath is fairly regular and well-organized
  - Memory is highly regular
  - Control is irregular and global
- But, the state diagrams that define the controller for an instruction set processor are highly structured
  - Use this structure to construct a simple “microsequencer”
  - Control reduces to programming this simple device

=> microprogramming

**Microinstruction**

- Control signals:
  - Think of the set of control signals that must be asserted in a state as an instruction
  - Executing a microinstruction has the effect of asserting the control signal specified by the microinstruction
- Sequencing:
  - What microinstruction should be executed next?
    - Execute sequentially (next state unconditionally)
    - Branch (next state also depends on inputs)
- A microprogram is a sequence of microinstructions executing a program flow chart (finite state machine)

**Designing a Microinstruction Set**

1) Start with a list of control signals
2) Group signals together that make sense (vs. random): called fields
3) Places fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
4) Create a symbolic legend for the microinstruction format, showing name of field values and how they set control signals
  - Use computers to design computers
5) To minimize the width, encode operations that will never be used at the same time

**1-3) Control Signals and Fields**

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted</th>
<th>Effect when asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSrcA</td>
<td>1st ALU operand = PC</td>
<td>1st ALU operand = Reg[rs]</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>Reg file is written</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Reg. write data input = ALU Reg. write data input = MDR RegDest</td>
<td>Reg. write dest. no. = rt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reg. write dest. no. = rd</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Memory at address is read</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Memory at address is written</td>
</tr>
<tr>
<td>IrdD</td>
<td>Memory address = PC</td>
<td>Memory address = ALUOut</td>
</tr>
<tr>
<td>IrWrite</td>
<td>None</td>
<td>IR = Memory</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>PC = PCSource</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>If zero then PC = PCSource</td>
</tr>
</tbody>
</table>

**Multiple Bit Control**

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUop</td>
<td>00</td>
<td>ALU adds</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>ALU subtracts</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>ALU operates according to func code</td>
</tr>
<tr>
<td>ALUSrcB</td>
<td>00</td>
<td>2nd ALU input = B</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>2nd ALU input = 4</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2nd ALU input = sign extended IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>2nd ALU input = sign extended, shift left 2 IR[15-0]</td>
</tr>
<tr>
<td>PCSrc</td>
<td>00</td>
<td>PC = ALU (PC + 4)</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>PC = ALUOut (branch target address)</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>PC = PC+4[31-28] : IR[25-0] &lt;&lt; 2</td>
</tr>
</tbody>
</table>
4) Fields and Legend

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field with Specific Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU control</td>
<td>Add</td>
<td>ALU adds</td>
</tr>
<tr>
<td></td>
<td>Subt</td>
<td>ALU subtracts</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>ALU does function code</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>1st ALU input = A (Reg[r])</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2nd ALU input = B (Reg[r])</td>
</tr>
<tr>
<td></td>
<td>Ext</td>
<td>2nd ALU input = sign ext</td>
</tr>
<tr>
<td></td>
<td>Extshft</td>
<td>2nd ALU input = sign ex., sl</td>
</tr>
<tr>
<td>Register control</td>
<td>Read A = Reg[r]; B = Reg[r];</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>Reg[rd] = ALUOut</td>
</tr>
<tr>
<td></td>
<td>Write MDR</td>
<td>Reg[r] = MDR</td>
</tr>
<tr>
<td>Memory</td>
<td>Read PC</td>
<td>IR (MDR) = mem[PC]</td>
</tr>
<tr>
<td></td>
<td>Read ALU</td>
<td>MDR = mem[ALUout]</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>mem[ALUout] = B</td>
</tr>
<tr>
<td>PC write</td>
<td>ALU</td>
<td>PC = ALU output</td>
</tr>
<tr>
<td></td>
<td>ALUout-cond.</td>
<td>if ALU zero then PC = ALUout</td>
</tr>
<tr>
<td></td>
<td>Jump addr.</td>
<td>PC = PCsrc</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>Go to sequential microinstruction</td>
</tr>
<tr>
<td></td>
<td>Fetch</td>
<td>Go to the first microinstruction</td>
</tr>
<tr>
<td></td>
<td>Dispatch 1</td>
<td>Dispatch using ROM1</td>
</tr>
<tr>
<td></td>
<td>Dispatch 2</td>
<td>Dispatch using ROM2</td>
</tr>
</tbody>
</table>

Control Signals

<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></td>
<td>Subt</td>
<td>ALUOp = 01</td>
<td>Cause the ALU to subtract; this implements the compare for branches</td>
</tr>
<tr>
<td>Func code</td>
<td>ALU</td>
<td>ALUOp = 10</td>
<td>Use the instruction function code to determine ALU control.</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>ALUInstrw = 0</td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>ALUInstrw = 1</td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>ALUInstrw = 01</td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>Ext</td>
<td>ALUInstrw = 01</td>
<td>Use the output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>Extshft</td>
<td>ALUInstrw = 11</td>
<td>Use the output of the shift by two unit as the second ALU input.</td>
</tr>
<tr>
<td>Register control</td>
<td>Read RegWrite, Reg[rd] = MDR</td>
<td>Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>Reg[rd] = MDR</td>
<td>Write a register using the rt field of the IR as the register number and the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td></td>
<td>Write MDR</td>
<td>Reg[rd] = 0, Mem=Reg[rd] = 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>
<tr>
<td>Memory</td>
<td>Read PC</td>
<td>MemRead, xrd = 0</td>
<td>Read memory using the PC as address; write result into IR (and the MDR).</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>MemWrite, xrd = 1</td>
<td>Write memory using the ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td></td>
<td>Write MDR</td>
<td>MemWrite, xrd = 1</td>
<td>Write memory using the ALUOut as address; contents of B as the data.</td>
</tr>
<tr>
<td>PC write control</td>
<td>ALU ALUOut-cond.</td>
<td>PCSource + 01, PCWrite</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
</tr>
<tr>
<td></td>
<td>Jump addr.</td>
<td>PCSource + 15, PCWrite</td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>AddrCtl = 11</td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td></td>
<td>Fetch</td>
<td>AddrCtl = 00</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td></td>
<td>Dispatch 1</td>
<td>AddrCtl = 01</td>
<td>Dispatch using the ROM 1.</td>
</tr>
<tr>
<td></td>
<td>Dispatch 2</td>
<td>AddrCtl = 10</td>
<td>Dispatch using the ROM 2.</td>
</tr>
</tbody>
</table>
**The Dispatch ROMs**

<table>
<thead>
<tr>
<th>Op</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>R-format</td>
<td>Rformat1</td>
</tr>
<tr>
<td>000010</td>
<td>jmp</td>
<td>JUMP1</td>
</tr>
<tr>
<td>000100</td>
<td>beq</td>
<td>BEQ1</td>
</tr>
<tr>
<td>100011</td>
<td>lw</td>
<td>Mem1</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>Mem1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>lw</td>
<td>LW2</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>SW2</td>
</tr>
</tbody>
</table>

**Our Plan: Using ROM**

**Microinstruction Interpretation**

- Main Memory
- CPU
- Control Memory
- Execution Unit
- Data

**Microprogramming Using ROM: Pros and Cons**

- **Ease of design**
- **Flexibility**
  - Each to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- **Generality**
  - Implement multiple instr. sets on same machine
  - Can tailor instruction set to application
  - Can implement very powerful instruction sets (just more control memory)
- **Compatibility**
  - Many organizations, same instruction set
- **Costly to implement and Slow**
5) Microinstruction Encoding

<table>
<thead>
<tr>
<th>State number</th>
<th>Control bits 17-2</th>
<th>Control bits 1-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1001010000001000</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>000000000011000</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>0000000000011000</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>0011000000000000</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>0000001000000000</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>0010100000000000</td>
<td>00</td>
</tr>
<tr>
<td>6</td>
<td>0000000010000000</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>0000000000110000</td>
<td>00</td>
</tr>
<tr>
<td>8</td>
<td>0100000100000000</td>
<td>00</td>
</tr>
<tr>
<td>9</td>
<td>1000001000000000</td>
<td>00</td>
</tr>
</tbody>
</table>

Fig. C.4.5

- Bits 7-13 can be encoded to 3 bits because only one bit of the 7 bits of the control word is ever active.

---

Minimal vs. Maximal Encoding

- **Minimal (Horizontal):**
  + more control over the potential parallelism of operations in the datapath
  - uses up lots of control store

- **Maximal (Vertical):**
  + uses less number of control store
  - extra level of decoding may slow the machine down

---

Summary of Control

- Control is specified by a finite state diagram
- Specialized state-diagrams easily captured by microsequencer
  - simple increment and "branch" fields
  - datapath control fields
- Control can also be specified by microprogramming
- Control is more complicated with:
  - complex instruction sets
  - restricted datapaths
- Simple instruction set and powerful datapath => simple control
  - could reduce hardware
  - Or go for speed => many instructions at once!

---

Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
  - Multicycle datapath
  - Multicycle execution steps
  - Multicycle control
- Microprogramming: simplifying control
- Exceptions
Exceptions

- Normal control flow: sequential, jumps, branches, calls, returns
- Exception = unprogrammed control transfer
  - system takes action to handle the exception
    - must record address of the offending instruction
    - should know cause and transfer to proper handler
  - if it returns to user, must save & restore user state

User/System Modes

- By providing two modes of execution (user/system), computer may manage itself
  - OS is a special program that runs in the privileged system mode and has access to all of the resources of the computer
  - Presents “virtual resources” to each user that are more convenient than the physical resources
    - files vs. disk sectors
    - virtual memory vs. physical memory
  - protects each user program from others
- Exceptions allow the system to take action in response to events that occur while user program is executing
  - OS begins at the handler

Two Types of Exceptions

- Interrupts:
  - caused by external events and asynchronous to execution
  - may be handled between instructions
  - simply suspend and resume user program
- Exceptions:
  - caused by internal events and synchronous to execution, e.g., exceptional conditions (overflow), errors (parity), faults
  - instruction may be retried or simulated and program continued or program may be aborted

MIPS Convention of Exceptions

- MIPS convention:
  - exception means any unexpected change in control flow, without distinguishing internal or external
  - use interrupt only when the event is externally caused

<table>
<thead>
<tr>
<th>Type of event</th>
<th>From where?</th>
<th>MIPS terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>External</td>
<td>Interrupt</td>
</tr>
<tr>
<td>Invoke OS from user program</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Hardware malfunctions</td>
<td>Either</td>
<td>Exception or</td>
</tr>
<tr>
<td>Arithmetic overflow</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Using an undefined inst.</td>
<td>Internal</td>
<td>Exception</td>
</tr>
</tbody>
</table>
Precise Interrupts
- Precise: machine state is preserved as if program executed up to the offending instruction.
  - Same system code will work on different implementations of the architecture.
  - Position clearly established by IBM, and taken by MIPS.
  - Difficult in the presence of pipelining, out-of-order execution, ...
- Imprecise: system software has to figure out what is where and put it all back together.
- Performance goals often lead designers to forsake precise interrupts.
  - System software developers, user, markets etc., usually wish they had not done this.

Handling Exceptions in Our Design
- Consider two types of exceptions: undefined instruction & arithmetic overflow.
- Basic actions on exception:
  - Save state: save the address of the offending instruction in the exception program counter (EPC).
  - Transfer control to OS at some specified address: => need to know the cause for the exception.
  - Then know the address of exception handler.
  - After service, OS can terminate the program or continue its execution, using EPC to return.

Saving State: General Approaches
- Push it onto the stack:
  - Vax, 68k, 80x86.
- Save it in special registers:
  - MIPS EPC, BadVaddr, Status, Cause.
- Shadow Registers:
  - M88k.
  - Save state in a shadow of the internal pipeline registers.

Addressing the Exception Handler
- Traditional approach: interrupt vector
  - The cause of an exception is a vector giving the address of the handler.
  - PC <- MEM[IV_base + cause || 00]
  - 68000, Vax, 80x86, ...
- RISC Handler Table
  - PC <- IV_base + cause || 0000
  - Saves state and jumps.
  - Sparc, PA, M88K, ...
- MIPS approach: fixed entry
  - Use a status register (cause register) to hold a field to indicate the cause.
  - PC <- EXC_addr.
Additions for Our Design

- **EPC**: reg. to hold address of affected inst.
- **Cause**: reg. to record cause of exception
  - Assume LSB encodes the two possible exception sources: undefined instruction=0 and arithmetic overflow=1
- Two control signals to write EPC (EPCWrite) and Cause (CauseWrite), and one control signal (IntCause) to set LSB of Cause register
- Be able to write exception address into PC, assuming at C000 0000hex
  => needs a 4-way MUX to PC
- May undo PC = PC + 4, since want EPC to point to offending inst. (not its successor)

Exception Detection

- **Undefined instruction**: detected when no next state is defined from state 1 for the op value
  - Handle this by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, and beq as a new state, “other”
- **Arithmetic overflow**: detected with the Overflow signal out of the ALU
  - This signal is used in the modified FSM to specify an additional possible next state

Note: challenge in designing control of a real machine is to handle different interactions between instructions and other exception-causing events such that control logic remains small and fast
- Complex interactions makes the control unit the most challenging aspect of hardware design
Summary

♦ Specialize state diagrams easily captured by microsequencer
  ● simple increment and branch fields
  ● datapath control fields
♦ Control design reduces to microprogramming
♦ Exceptions are the hard part of control
  ● Need to find convenient place to detect exceptions and to branch to state or microinstruction that saves PC and invokes OS
  ● Harder with pipelined CPUs that support page faults on memory accesses, i.e., the instruction cannot complete AND you must restart program at exactly the instruction with the exception