Control Implementation Alternatives

- Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.

**Initial Representation**
- Finite State Diagram
- Microprogram

**Sequencing Control**
- Explicit Next State Function
- Microprogram counter + Dispatch ROMs

**Logic Representation**
- Logic Equations
- Truth Tables

**Implementation Technique**
- PLA
  - “hardwired control”
- ROM
  - “microprogrammed control”

Chapter 5.5, 5.6
## Operations In Each Cycle

<table>
<thead>
<tr>
<th>R-Type</th>
<th>Logic Immediate</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution</strong></td>
<td>ALUout ← A + B</td>
<td>ALUout ← A + ZeroExt(imm16)</td>
<td>ALUout ← A + SignEx(Im16)</td>
<td>ALUout ← A + SignEx(Im16)</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td>M ← Mem[ALUout]</td>
<td></td>
<td>Mem[ALUout] ← B</td>
</tr>
</tbody>
</table>
Finite State Machine (FSM) Specification

IR ← MEM[PC]  
PC ← PC + 4  
0000

“instruction fetch”

A ← R[rs]  
B ← R[rt]

ALUout ← PC + SX  
0001

“decode”

R-type

ALUout ← A fun B  
0100

ALUout ← A op ZX  
0110

ALUout ← A + SX  
1000

ALUout ← A + SX  
1011

BEQ

If A = B then  
PC ← ALUout  
0010

To instruction fetch

ORi

LW

ALUout ← A + SX  
1001

MEM[ALUout] ← M

MEM[ALUout] ← B  
1100

To instruction fetch

SW

M ← MEM[ALUout]  
1010

R[rd] ← ALUout  
0101

R[rt] ← ALUout  
0111

R[rt] ← M  
1010

To instruction fetch

To instruction fetch

To instruction fetch

To instruction fetch
Microprogrammed Control

• Finite state machine (FSM) control for a full set of instructions is very complex, and may involve a very large number of states:
  – Slight microoperation changes require new FSM controller.

• Microprogramming: Designing the control as a program that implements the machine instructions.

• A microprogram for a given machine instruction is a symbolic representation of the control involved in executing the instruction and is comprised of a sequence of microinstructions.

• Each microinstruction defines the set of datapath control signals that must asserted (active) in a given state or cycle.

• The format of the microinstructions is defined by a number of fields each responsible for asserting a set of control signals.

• Microarchitecture (or CPU organization):
  – Logical structure and functional capabilities of the hardware as seen by the microprogrammer.
A Typical Microcode Controller Implementation

ROM/PLA

Microcode storage

Outputs

Datapath control outputs

Input

Micro program counter

Address select logic

Micro PC

Sequencing control

1

Adder

Inputs from instruction register opcode field
"Macroinstruction" Interpretation

- Main Memory
- Execution Unit
- CPU
- Control Memory

Microprogram Storage

Microprogram

User program plus Data

- ADD
- SUB
- AND

one of these is mapped into one of these

AND microsequence

- e.g., Fetch
- Calc Operand Addr
- Fetch Operand(s)
- Calculate
- Save Answer(s)

e.g. MIPS code
Variations on Microprogram Formats

- “Horizontal” Microcode:
  - A control field for each control point in the machine.

- “Vertical” Microcode:
  - A Compact microinstruction format for each class of control points.
  - Local decode is used to generate all control points.

\[
\begin{array}{cccccc}
\mu\text{seq} & \mu\text{addr} & A\text{-mux} & B\text{-mux} & \text{bus enables} & \text{register enables} \\
\end{array}
\]
More Vertical Microprogram Formats

Multiformat Microcode:

1 3 6
0 cond next address
1 3 3 3
1 dst src alu

Branch Jump
Register Transfer Operation
Design of Microinstruction Format/Addressing

• Start with a list of all control signals needed.
• **Partition control signals with similar functions** into a number of signal sets that share a single microinstruction field.
• **A sequencing microinstruction field** is used to indicate the next microinstruction to execute.
• Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last).
• Since microinstructions are placed in a ROM or PLA, addresses must be assigned to microinstructions, usually sequentially.
• Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals.
• To minimize microinstruction width, operations that will never be used at the same time may be encoded.
Next Microinstruction Selection

• The next microinstruction to execute can be found by using the sequencing field value:

  0 Branch to a microinstruction that begins execution of the next MIPS instruction. “Fetch” is placed in the sequencing field.

  1 Choose the next microinstruction based on the control unit input (a dispatch).
      • Dispatches are implemented by a look-up table stored in a ROM containing addresses of target microinstruction.
      • The table is indexed by the control unit input (Opcode).
      • A dispatch operation is indicated by placing “Dispatch i” in the sequencing field; i is the dispatch table number.

  2 Increment the address of the current instruction. Indicated in the microinstruction by putting “Seq” in the sequencing field.
Types of "branching"
- Set state to 0 (fetch)
- Dispatch i (state 1)
- Use incremented address (seq) state number 2
Next State Function: Sequencing Field

- For next state function (next microinstruction address):

<table>
<thead>
<tr>
<th>Signal</th>
<th>Name</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequencing</td>
<td>Fetch</td>
<td>00</td>
<td>Next µaddress = 0</td>
</tr>
<tr>
<td></td>
<td>Dispatch</td>
<td>01</td>
<td>Next µaddress = dispatch ROM</td>
</tr>
<tr>
<td></td>
<td>Seq</td>
<td>10</td>
<td>Next µaddress = µaddress + 1</td>
</tr>
</tbody>
</table>

Microprogram Storage

1. Adder
2. Mux
3. ROM
4. Dispatch ROM (look-up table indexed by opcode)
### List of control Signals Grouped Into Fields

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted (=0)</th>
<th>Effect when asserted (=1)</th>
<th>(SRC1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSelA</td>
<td>1st ALU operand = PC</td>
<td>1st ALU operand = Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>Reg. is written</td>
<td></td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Reg. write data input = ALU</td>
<td>Reg. write data input = memory</td>
<td></td>
</tr>
<tr>
<td>RegDst</td>
<td>Reg. dest. no. = rt</td>
<td>Reg. dest. no. = rd</td>
<td></td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Memory at address is read,</td>
<td>Memory</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Memory at address is written</td>
<td></td>
</tr>
<tr>
<td>IorD</td>
<td>Memory address = PC</td>
<td>Memory address = S</td>
<td></td>
</tr>
<tr>
<td>IRWrite</td>
<td>None</td>
<td>IR ← Memory</td>
<td>PC</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>PC ← PCSource</td>
<td></td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>IF ALUzero then PC ← PCSource</td>
<td></td>
</tr>
<tr>
<td>PCSource</td>
<td>PCSource = ALU</td>
<td>PCSource = ALUout</td>
<td></td>
</tr>
</tbody>
</table>

### Single Bit Control

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Value</th>
<th>Effect</th>
<th>(ALU Control)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp</td>
<td>00</td>
<td>ALU adds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>ALU subtracts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>ALU does function code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>ALU does logical OR</td>
<td></td>
</tr>
<tr>
<td>ALUSelB</td>
<td>000</td>
<td>2nd ALU input = Reg[rt]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>001</td>
<td>2nd ALU input = 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>010</td>
<td>2nd ALU input = sign extended IR[15-0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>011</td>
<td>2nd ALU input = sign extended, shift left 2 IR[15-0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2nd ALU input = zero extended IR[15-0]</td>
<td></td>
</tr>
</tbody>
</table>

### Multiple Bit Control

- SRC1
- SRC2

**Legend:**
- (SRC1) Source of Control Signal 1
- (SRC2) Source of Control Signal 2
- (Memory) Memory
- (PC Control) PC
## Microinstruction Format

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Width</th>
<th>Control Signals Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wide</td>
<td>narrow</td>
</tr>
<tr>
<td>ALU Control</td>
<td>4</td>
<td>2 AlUOp</td>
</tr>
<tr>
<td>SRC1</td>
<td>2</td>
<td>1 ALUSelA</td>
</tr>
<tr>
<td>SRC2</td>
<td>5</td>
<td>3 ALUSelB</td>
</tr>
<tr>
<td>Destination</td>
<td>3</td>
<td>2 RegWrite, MemtoReg, RegDst</td>
</tr>
<tr>
<td>Memory</td>
<td>4</td>
<td>3 MemRead, MemWrite, IorD</td>
</tr>
<tr>
<td>Memory Register</td>
<td>1</td>
<td>1 IRWrite</td>
</tr>
<tr>
<td>PCWrite Control</td>
<td>4</td>
<td>3 PCWrite, PCWriteCond, PCSource</td>
</tr>
<tr>
<td>Sequencing</td>
<td>3</td>
<td>2 AddrCtl</td>
</tr>
<tr>
<td>Total width</td>
<td>26</td>
<td>17 bits</td>
</tr>
</tbody>
</table>
# Microinstruction Field Values

<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</td>
<td>Add</td>
<td>ALU adds</td>
</tr>
<tr>
<td></td>
<td>Subt.</td>
<td>ALU subtracts</td>
</tr>
<tr>
<td></td>
<td>Func code</td>
<td>ALU does function code</td>
</tr>
<tr>
<td></td>
<td>Or</td>
<td>ALU does logical OR</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>1st ALU input = PC</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>1st ALU input = Reg[rs]</td>
</tr>
<tr>
<td>SRC2</td>
<td>4</td>
<td>2nd ALU input = 4</td>
</tr>
<tr>
<td></td>
<td>Extend</td>
<td>2nd ALU input = sign ext. IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>Extend0</td>
<td>2nd ALU input = zero ext. IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>Extshft</td>
<td>2nd ALU input = sign ex., sl IR[15-0]</td>
</tr>
<tr>
<td></td>
<td>rt</td>
<td>2nd ALU input = Reg[rt]</td>
</tr>
<tr>
<td>destination</td>
<td>rd ALU</td>
<td>Reg[rd] ← ALUout</td>
</tr>
<tr>
<td></td>
<td>rt ALU</td>
<td>Reg[rt] ← ALUout</td>
</tr>
<tr>
<td></td>
<td>rt Mem</td>
<td>Reg[rt] ← Mem</td>
</tr>
<tr>
<td>Memory</td>
<td>Read PC</td>
<td>Read memory using PC</td>
</tr>
<tr>
<td></td>
<td>Read ALU</td>
<td>Read memory using ALU output</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>Write memory using ALU output, value B</td>
</tr>
<tr>
<td>Memory register</td>
<td>IR</td>
<td>IR ← Mem</td>
</tr>
<tr>
<td>PC write</td>
<td>ALU</td>
<td>PC ← ALU</td>
</tr>
<tr>
<td></td>
<td>ALUoutCond</td>
<td>IF ALU Zero then PC ← ALUout</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 i</td>
<td>Dispatch using ROM.</td>
</tr>
</tbody>
</table>
Instruction Fetch/decode Microcode Sequence

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Dest.</th>
<th>Memory</th>
<th>Mem. Reg.</th>
<th>PC Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch:</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td></td>
<td>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
</tr>
<tr>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dispatch</td>
</tr>
</tbody>
</table>

First microinstruction: Fetch, increment PC

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Add</td>
<td>ALU adds</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>1st ALU input = PC</td>
</tr>
<tr>
<td>SRC2</td>
<td>4</td>
<td>2nd ALU input = 4</td>
</tr>
<tr>
<td>Memory</td>
<td>Read PC</td>
<td>Read memory using PC</td>
</tr>
<tr>
<td>Memory register</td>
<td>IR</td>
<td>IR ← Mem</td>
</tr>
<tr>
<td>PC write</td>
<td>ALU</td>
<td>PC ← ALU</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>Go to sequential µinstruction</td>
</tr>
</tbody>
</table>

Second microinstruction: Decode, calculate branch address

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Add</td>
<td>ALU adds result in ALUout</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>1st ALU input = PC</td>
</tr>
<tr>
<td>SRC2</td>
<td>Extshft</td>
<td>2nd ALU input = sign ex., sl IR[15-0]</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Dispatch</td>
<td>Dispatch using ROM according to opcode</td>
</tr>
</tbody>
</table>
LW Completion Microcode Sequence

First microinstruction: Execute, effective memory address calculation

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Add</td>
<td>ALU adds, result in ALUout</td>
</tr>
<tr>
<td>SRC1</td>
<td>rs</td>
<td>1st ALU input = Reg[rs]</td>
</tr>
<tr>
<td>SRC2</td>
<td>Extend</td>
<td>2nd ALU input = sign ext. IR[15-0]</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>Go to sequential µinstruction</td>
</tr>
</tbody>
</table>

Second microinstruction: Memory, read using ALUout

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Read ALU</td>
<td>Read memory using ALU output</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>Go to sequential µinstruction</td>
</tr>
</tbody>
</table>

Third microinstruction: Write Back, from memory to register rt

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination</td>
<td>rt Mem</td>
<td>Reg[rt] ← Mem</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Fetch</td>
<td>Go to the first microinstruction (fetch)</td>
</tr>
</tbody>
</table>
SW Completion Microcode Sequence

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Dest.</th>
<th>Memory</th>
<th>Mem. Reg.</th>
<th>PC Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
</tbody>
</table>

First microinstruction: Execute, effective memory address calculation

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Add</td>
<td>ALU adds result in ALUout</td>
</tr>
<tr>
<td>SRC1</td>
<td>rs</td>
<td>1st ALU input = Reg[rs]</td>
</tr>
<tr>
<td>SRC2</td>
<td>Extend</td>
<td>2nd ALU input = sign ext. IR[15-0]</td>
</tr>
</tbody>
</table>

Second microinstruction: Memory, write to memory

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Write ALU</td>
<td>Write memory using ALU output, value B</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Fetch</td>
<td>Go to the first microinstruction (fetch)</td>
</tr>
</tbody>
</table>
### R-Type Completion Microcode Sequence

#### Field Name | Value for Field | Function of Field
--- | --- | ---
ALU | Func code | ALU does function code
SRC1 | rs | 1st ALU input = Reg[rs]
SRC2 | rt | 2nd ALU input = Reg[rt]
Sequencing | Seq | Go to sequential µinstruction

**First microinstruction:** Execute, perform ALU function

**Second microinstruction:** Write Back, ALU result in register rd

#### Field Name | Values for Field | Function of Field
--- | --- | ---
destination | rd ALU | Reg[rd] ← ALUout
Sequencing | Fetch | Go to the first microinstruction (fetch)
# BEQ Completion Microcode Sequence

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Dest.</th>
<th>Memory</th>
<th>Mem. Reg.</th>
<th>PC Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beq:</td>
<td>Subt.</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ALUoutCond.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
</tbody>
</table>

First microinstruction: Execute, compute condition, update PC

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Subt.</td>
<td>ALU subtracts</td>
</tr>
<tr>
<td>SRC1</td>
<td>rs</td>
<td>1st ALU input = Reg[rs]</td>
</tr>
<tr>
<td>SRC2</td>
<td>rt</td>
<td>2nd ALU input = Reg[rt]</td>
</tr>
<tr>
<td>PC write</td>
<td>ALUoutCond</td>
<td>IF ALU Zero then PC ← ALUout</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Fetch</td>
<td>Go to the first microinstruction (fetch)</td>
</tr>
</tbody>
</table>
## ORI Completion Microcode Sequence

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Dest.</th>
<th>Memory</th>
<th>Mem. Reg.</th>
<th>PC Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ori:</td>
<td>Or</td>
<td>rs</td>
<td>Extend0</td>
<td></td>
<td>ALU</td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Mem</td>
<td>Reg</td>
<td>PC Write</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### First microinstruction: Execute, rs OR immediate

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Or</td>
<td>ALU does logical OR result in ALUout</td>
</tr>
<tr>
<td>SRC1</td>
<td>rs</td>
<td>1st ALU input = Reg[rs]</td>
</tr>
<tr>
<td>SRC2</td>
<td>Extend0</td>
<td>2nd ALU input = zero ext. IR[15-0]</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>Go to sequential µinstruction</td>
</tr>
</tbody>
</table>

### Second microinstruction: Write Back, ALU result in register rt

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination</td>
<td>rt ALU</td>
<td>Reg[rt] ← ALUout</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Fetch</td>
<td>Go to the first microinstruction (fetch)</td>
</tr>
</tbody>
</table>
# Microprogram for The Control Unit

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Dest.</th>
<th>Memory</th>
<th>Mem. Reg.</th>
<th>PC Write</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch:</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>IR</td>
<td>ALU</td>
<td>Seq</td>
<td>Dispatch</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Read ALU</td>
<td>rt MEM</td>
<td></td>
<td>Seq</td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sw:</td>
<td>Add</td>
<td>rs</td>
<td>Extend</td>
<td>Write ALU</td>
<td></td>
<td></td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtype:</td>
<td>Func</td>
<td>rs</td>
<td>rt</td>
<td>rd ALU</td>
<td></td>
<td></td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beq:</td>
<td>Subt.</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>ALUoutCond.</td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>Ori:</td>
<td>Or</td>
<td>rs</td>
<td>Extend0</td>
<td></td>
<td></td>
<td>rt ALU</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
</tbody>
</table>
Microprogramming Pros and Cons

Pros:
- **Ease of design.**
- **Flexibility:**
  - Easy to adapt to changes in organization, timing, technology.
  - Can make changes late in design cycle, or even in the field.
- Can implement very powerful instruction sets (just more microprogram control memory is needed).
- **Generality:**
  - Can implement multiple instruction sets on the same machine.
  - Can tailor instruction set to application.
- **Compatibility:**
  - Many organizations, same instruction set.

Cons:
- Possibly more costly to implement than FSM control.
- Usually slower than FSM control.
Exceptions Handling in MIPS

- **Exceptions**: Events Other than branches or jumps that change the normal flow of instruction execution.

- Two main types: **Interrupts, Traps**.
  - *An interrupt* usually comes from outside the processor (I/O devices) to get the CPU’s attention to start a service routine.
  - *A trap* usually originates from an event within the CPU (Arithmetic overflow, undefined instruction) and initiates an exception handling routine usually by the operating system.

- The current MIPS implementation being considered can be extended to handle exceptions by adding two additional registers and the associated control lines:
  - **EPC**: A 32 bit register to hold the address of the affected instruction
  - **Cause**: A register used to record the cause of the exception.

    In this implementation only the low-order bit is used to encode the two handled exceptions:
    - undefined instruction = 0
    - overflow = 1

- Two additional states are added to the control finite state machine to handle these exceptions.
Two Types of Exceptions

- **Interrupts:**
  - Caused by external events (e.g. I/O device requests).
  - Asynchronous to program execution.
  - May be handled between instructions.
  - Simply suspend and resume user program.

- **Traps:**
  - Caused by internal events:
    - Exceptional conditions (e.g. overflow).
    - Errors (e.g. memory parity error).
    - Faults (e.g. Page fault, non-resident page).
  - Synchronous to program execution.
  - Condition must be remedied by the system exception handler.
  - Instruction may be executed again and program continued or program may be aborted.
Exception Handling

- Exception = an unprogrammed control transfer
  - System takes action to handle the exception which include:
    - Recording the address of the offending instruction.
    - Saving & restoring user program state.
    - Returning control to user (unless user program is aborted).
Addressing The Exception Handler

- **Traditional Approach, Interrupt Vector:**
  - $PC \leftarrow MEM[IV\_base + cause || 00]$
  - Used in: 370, 68000, Vax, 80x86, . . .

- **RISC Handler Table:**
  - $PC \leftarrow IT\_base + cause || 0000$
  - saves state and jumps
  - Used in: Sparc, HP-PA, . . .

- **MIPS Approach: Fixed entry**
  - $PC \leftarrow EXC\_addr$
  - Actually a very small table:
    - RESET entry
    - TLB
    - other
Exception Handling: Saving The State

• Push it onto the stack:
  – Vax, 68k, x86

• Save it in special registers:
  – **MIPS**: EPC, BadVaddr, Status, Cause

• Shadow Registers:
  – M88k.
    • Save state in a shadow (a copy) of the internal CPU registers.
Additions to MIPS to Support Exceptions

- **EPC:** A 32-bit register used to hold the address of the affected instruction (in reality register 14 of coprocessor 0).
- **Cause:** A register used to record the cause of the exception. In the MIPS architecture this register is 32 bits, though some bits are currently unused. Assume that bits 5 to 2 of this register encode the two possible exception sources mentioned above:
  - Undefined instruction = 0
  - Arithmetic overflow = 1 (in reality, register 13 of coprocessor 0).
- **BadVAddr:** Register contains memory address at which memory reference occurred (register 8 of coprocessor 0).
- **Status:** Interrupt mask and enable bits (register 12 of coprocessor 0).
- Control signals to write EPC, Cause, BadVAddr, and Status.
- Be able to write exception address into PC, increase mux to add as input 01000000 00000000 00000000 01000000, two (8000 0080\text{hex}).
- Must undo PC = PC + 4, since we want EPC to point to offending instruction (not its successor); PC = PC - 4
Details of MIPS Status Register

- Mask = 1 bit for each of 5 hardware and 3 software interrupt levels
  - 1 → enables interrupts
  - 0 → disables interrupts
- k = kernel/user
  - 0 → was in the kernel when interrupt occurred
  - 1 → was running user mode
- e = interrupt enable
  - 0 → interrupts were disabled
  - 1 → interrupts were enabled
Details of MIPS Cause register

Status

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>10</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pending</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Pending interrupt**: 5 hardware levels: bit set if interrupt occurs but not yet serviced:
  - Handles cases when more than one interrupt occurs at same time, or while records interrupt requests when interrupts disabled.

- **Exception Code**: Encodes reasons for interrupt:
  0 (INT) → external interrupt
  4 (ADDRL) → Address error exception (load or instr fetch).
  5 (ADDRS) → Address error exception (store).
  6 (IBUS) → Bus error on instruction fetch.
  7 (DBUS) → Bus error on data fetch.
  8 (Syscall) → Syscall exception.
  9 (BKPT) → Breakpoint exception.
  10 (RI) → Reserved Instruction exception.
  12 (OVF) → Arithmetic overflow exception.
The MIPS Multicycle Datapath With Exception Handling Added
Finite State Machine (FSM) Specification

“instruction fetch”

“decode”

A ← R[rs]
B ← R[rt]

ALUout ← PC + SX

0001

R-type

ALUout ← A fun B
0100

ALUout ← A op ZX
0110

ALUout ← A + SX
1000

ALUout ← A + SX
1011

If A = B then
PC ← ALUout
0010

M ← MEM[ALUout]
1001

MEM[ALUout] ← B
1100

R[rd] ← ALUout
0101

R[rt] ← ALUout
0111

R[rt] ← M
1010

LW

SW

BEQ

To instruction fetch

To instruction fetch

To instruction fetch

To instruction fetch

To instruction fetch

To instruction fetch
FSM Control Specification To Handle Exceptions

IR ← MEM[PC]
PC ← PC + 4
0000

“instruction fetch”

EPC ← PC - 4
PC ← exp_addr
cause ← 1

overflow

ALUout ← PC + SX
0001

“decode”

EPC ← PC - 4
PC ← exp_addr
cause ← 0

undefined instruction

ALUout ← A fun B
0100

R-type

ALUout ← A op ZX
0110

ALUout ← A + SX
1000

ALUout ← A + SX
1011

if A = B then
PC ← ALUout
0010

To instruction fetch

Write-back

Memory

Execute

R[rd] ← ALUout
0101

R[rs] ← A

ORi

M ← MEM[ALUout]
1001

MEM[ALUout] ← B
1100

R[rt] ← ALUout
0111

ALUout ← A + SX
1010

To instruction fetch

To instruction fetch

To instruction fetch
Control Finite State Machine
With Exception Detection

Version In Textbook
Figure 5.50

EECC550 - Shaaban

Version In Textbook
Figure 5.50