Control Implementation Alternatives For Multi-Cycle CPUs

- 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

Sequencing Control

Logic Representation

Implementation Technique

Finite State Diagram

Microprogram

Explicit Next State Function

Microprogram counter + Dispatch ROMs

Logic Equations

Truth Tables

Logic Equations Truth Tables

Implementation Technique

“hardwired control”

“AKA Control Program”

Programmable Logic Array

μPC

Read Only Memory

ROM

“microprogrammed control”

 PLA

μPC

“μPC”

Read Only Memory

ROM

“AKA Control Program”

3rd Edition - Microprogramming: Chapter 5.7, Appendix C (both posted online) Exception Handling: Chapter 5.6 (see handout) – Not in 4th Edition
Alternative Multiple Cycle Datapath With Control Lines
(Fig 5.28 In Textbook)

(Book version: ORI not supported, Jump supported)

3rd Edition Figure 5.28 page 323 – see handout
## The Effect of 1-bit 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>ALUSrcA</td>
<td>The first ALU operand is the PC</td>
<td>The First ALU operand is register A (i.e R[rs])</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Content of memory specified by the address input are put on the memory data output.</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Memory contents specified by the address input is 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 ALUOut register.</td>
<td>The value fed to the register write data input comes from data memory register (MDR).</td>
</tr>
<tr>
<td>IorD</td>
<td>The PC is used to supply the address to the memory unit.</td>
<td>The ALUOut register is used to supply the the address to the memory unit.</td>
</tr>
<tr>
<td>IRWrite</td>
<td>None</td>
<td>The output of the memory is written into Instruction Register (IR)</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>The PC is written; the source is controlled by PCSource</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>The PC is written if the Zero output of the ALU is also active.</td>
</tr>
</tbody>
</table>

3rd Edition Figure 5.29 page 324 – See handout
The Effect of 2-bit Control Signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Value (Binary)</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp</td>
<td>00</td>
<td>The ALU performs an add operation</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>The ALU performs a subtract operation</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>The funct field of the instruction determines the ALU operation (R-Type)</td>
</tr>
<tr>
<td>ALUSrcB</td>
<td>00</td>
<td>The second input of the ALU comes from register B (i.e R[rt])</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>The second input of the ALU is the constant 4</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>The second input of the ALU is the sign-extended 16-bit immediate field of the instruction in IR</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>The second input of the ALU is the sign-extended 16-bit immediate field of IR shifted left 2 bits</td>
</tr>
<tr>
<td>PCSrc</td>
<td>00</td>
<td>Output of the ALU (PC+4) is sent to the PC for writing</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>The content of ALUOut (the branch target address) is sent to the PC for writing</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC+4[31:28] is sent to the PC for writing i.e jump address</td>
</tr>
</tbody>
</table>

3rd Edition Figure 5.29 page 324 – See handout
## Operations (Dependant RTN) for Each Cycle

<table>
<thead>
<tr>
<th>Cycle</th>
<th>R-Type</th>
<th>Load</th>
<th>Store</th>
<th>Branch</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
<td>ALUout ← PC + (SignExt(imm16) x4)</td>
</tr>
<tr>
<td>EX</td>
<td>ALUout ← A funct B</td>
<td>ALUout ← A + SignEx(Imm16)</td>
<td>ALUout ← A + SignEx(Imm16)</td>
<td>Zero ← A - B</td>
<td>PC ← Jump Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Zero: PC ← ALUout</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MDR ← Mem[ALUout]</td>
<td>Mem[ALUout] ← B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>R[rd] ← ALUout</td>
<td>R[rt] ← MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Fetch (IF) & Instruction Decode (ID) cycles are common for all instructions

EECC550 - Shaaban
#5 Lec # 6 Winter 2011 1-17-2012
FSM State Transition Diagram (From Book)

Figure 5.37 page 338

See handout

More on FSM controller implementation in Appendix C
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 a new FSM controller design.

- **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 or design):
  - Logical structure and functional capabilities of the hardware as seen by the microprogrammer.

(As opposed to ISA which is visible to assembly programmer)
A Typical Microcode Controller Implementation

ROM/PLA

Microcode storage

Read Only Memory (ROM)

(Microinstruction Address)

Outputs

Datapath control outputs

To Datapath

Micro PC

μPC

Sequencing control

Adder

Address select logic

Inputs from instruction register opcode field

 Opcode

State in FSM Control = Microinstruction in microprogrammed control

EECC550 - Shaaban
"Macroinstruction" Interpretation

i.e ISA Instruction

Main Memory

ISA Instructions

ADD
SUB
AND
DATA

User program plus Data

e.g. MIPS code

one of these is mapped into one of these

Microprogram

AND microsequence
(e.g. a sequence of microinstructions)

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

CPU

c控制 memory

execution unit

Microprogram Storage
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:
  
  **Fetch (field value =0)**
  Branch to a microinstruction that begins execution of the next MIPS instruction. “Fetch” is placed in the sequencing field.

  **Dispatch i (in this case: i = 1 or 2, field value 1 or 2)**
  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.

  **Seq (field value =3)**
  Increment the address of the current instruction. Indicated in the microinstruction by putting “Seq” in the sequencing field.

Sequencing Field Values = 0, 1, 2, or 3

Field Size = 2 bits

i.e. Jump based on Opcode

i.e Next sequential microinstruction

ROM = Read Only Memory
Types of “branching”
- Set state to 0 (fetch)
- Dispatch 1 (state 1)
- Dispatch 2 (state 2)
- Use incremented address (seq)

State in FSM Control = Microinstruction in microprogrammed control
**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 1</td>
<td>01</td>
<td>Next µaddress = dispatch ROM 1</td>
</tr>
<tr>
<td></td>
<td>Dispatch 2</td>
<td>10</td>
<td>Next µaddress = dispatch ROM 2</td>
</tr>
<tr>
<td></td>
<td>Seq</td>
<td>11</td>
<td>Next µaddress = µaddress + 1</td>
</tr>
</tbody>
</table>

Field Size = 2 bits

### Microprogram Storage

- Dispatch ROMs (look-up table indexed by opcode)

- More details in 3rd Edition Appendix C (posted online)

### Dispatch ROM 1

<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>0110</td>
</tr>
<tr>
<td>000010</td>
<td>jmp</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>beq</td>
<td>1000</td>
</tr>
<tr>
<td>100011</td>
<td>lw</td>
<td>0010</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0010</td>
</tr>
</tbody>
</table>

### Dispatch ROM 2

<table>
<thead>
<tr>
<th>Op</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>lw</td>
<td>0011</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0101</td>
</tr>
</tbody>
</table>

The dispatch ROMs each have $2^6 = 64$ entries that are 4 bits wide, since that is the number of bits in the state encoding.
<table>
<thead>
<tr>
<th>State number</th>
<th>Address-control action</th>
<th>Value of AddrCtl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Use dispatch ROM 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Use dispatch ROM 2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
</tbody>
</table>

Micro instruction Sequencing Field

Value for Each State Transition

<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>0</td>
<td>Next µaddress = 0</td>
</tr>
<tr>
<td>Dispatch 1</td>
<td>dispatch ROM 1</td>
<td>1</td>
<td>Next µaddress = dispatch ROM 1</td>
</tr>
<tr>
<td>Dispatch 2</td>
<td>dispatch ROM 2</td>
<td>2</td>
<td>Next µaddress = dispatch ROM 2</td>
</tr>
<tr>
<td>Seq</td>
<td>Seq</td>
<td>3</td>
<td>Next µaddress = µaddress + 1</td>
</tr>
</tbody>
</table>

**Micro instruction Sequencing Field**

**Values for Each State Transition**

State in FSM Control = Microinstruction in microprogrammed control
## Microinstruction Format

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Function of field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU control</td>
<td>Specify the operation being done by the ALU during this clock; the result is</td>
</tr>
<tr>
<td></td>
<td>always written in ALUOut.</td>
</tr>
<tr>
<td>SRC1</td>
<td>Specify the source for the first ALU operand.</td>
</tr>
<tr>
<td>SRC2</td>
<td>Specify the source for the second ALU operand.</td>
</tr>
<tr>
<td>Register control</td>
<td>Specify read or write for the register file, and the source of the value for a</td>
</tr>
<tr>
<td></td>
<td>write.</td>
</tr>
<tr>
<td>Memory</td>
<td>Specify read or write, and the source for the memory. For a read, specify the</td>
</tr>
<tr>
<td></td>
<td>destination register.</td>
</tr>
<tr>
<td>PCWrite control</td>
<td>Specify the writing of the PC.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Specify how to choose the next microinstruction to be executed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Width (bits)</th>
<th>Control Signals Set in Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU Control</td>
<td>2</td>
<td>ALUOp</td>
</tr>
<tr>
<td>SRC1</td>
<td>1</td>
<td>ALUSrcA</td>
</tr>
<tr>
<td>SRC2</td>
<td>2</td>
<td>ALUSrcB</td>
</tr>
<tr>
<td>Register Control</td>
<td>3</td>
<td>RegWrite, MemtoReg, RegDst</td>
</tr>
<tr>
<td>Memory</td>
<td>4</td>
<td>MemRead, MemWrite, IorD, IRWrite</td>
</tr>
<tr>
<td>PCWrite Control</td>
<td>4</td>
<td>PCWrite, PCWriteCond, PCSource</td>
</tr>
<tr>
<td>Sequencing</td>
<td>2</td>
<td>AddrCtl</td>
</tr>
</tbody>
</table>

Total width: 18 bits

**Partitioning Control Lines to Fields:**

- Seven Fields

---

*EECC550 - Shaaban*

#15 Lec # 6 Winter 2011 1-17-2012
## Microinstruction Field Values/Names

<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>Label</td>
<td>Any string</td>
<td>Used to specify labels to control microcode sequencing. Labels that end in a 1 or 2 are used for dispatching with a jump table that is indexed based on the opcode. Other labels are used as direct targets in the microinstruction sequencing. Labels do not generate control signals directly but are used to define the contents of dispatch tables and generate control for the Sequencing field.</td>
</tr>
<tr>
<td>ALU control</td>
<td>Add</td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td></td>
<td>Subt</td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
</tr>
<tr>
<td></td>
<td>Func code</td>
<td>Use the instruction’s func field to determine ALU control.</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td>SRC2</td>
<td>B</td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Use 4 for the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>Extend</td>
<td>Use output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>Extshft</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</td>
<td>Read two registers using the rs and rt fields of the IR as the register numbers, putting the data into registers A and B.</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>Write the register file using the rd field of the IR as the register number and the contents of ALUOut as the data.</td>
</tr>
<tr>
<td></td>
<td>Write MDR</td>
<td>Write the register file 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>Read memory using the PC as address; write result into IR (and the MDR).</td>
</tr>
<tr>
<td></td>
<td>Read ALU</td>
<td>Read memory using ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>Write memory using the ALUOut as address; contents of B as the data.</td>
</tr>
<tr>
<td>PCWrite control</td>
<td>ALU</td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<td></td>
<td>ALUOut-cond</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 address</td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td></td>
<td>Fetch</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td></td>
<td>Dispatch i</td>
<td>Dispatch using the ROM specified by i (1 or 2).</td>
</tr>
</tbody>
</table>
## Active Control Lines for Microinstruction Field Values + Names

<table>
<thead>
<tr>
<th>Field Size</th>
<th>Field name</th>
<th>Value</th>
<th>Signals active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2 bits</strong></td>
<td>ALU control</td>
<td>Add</td>
<td>ALUOp = 00</td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td></td>
<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></td>
<td></td>
<td>Func code</td>
<td>ALUOp = 10</td>
<td>Use the instruction's function code to determine ALU control.</td>
</tr>
<tr>
<td><strong>1 bit</strong></td>
<td>SRC1</td>
<td>PC</td>
<td>ALUSrcA = 0</td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>ALUSrcA = 1</td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td><strong>2 bits</strong></td>
<td>SRC2</td>
<td>B</td>
<td>ALUSrcB = 00</td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>ALUSrcB = 01</td>
<td>Use 4 as the second ALU input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extend</td>
<td>ALUSrcB = 10</td>
<td>Use output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extshft</td>
<td>ALUSrcB = 11</td>
<td>Use the output of the shift-by-two unit as the second ALU input.</td>
</tr>
<tr>
<td><strong>3 bits</strong></td>
<td>Register control</td>
<td>Read</td>
<td></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>
</tr>
<tr>
<td></td>
<td></td>
<td>Write ALU</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 ALUOut as the data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write MDR</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>
<tr>
<td><strong>4 bits</strong></td>
<td>Memory</td>
<td>Read PC</td>
<td>MemRead, IorD = 0, IRWrite</td>
<td>Read memory using the PC as address; write result into IR (and the MDR).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read ALU</td>
<td>MemWrite, IorD = 1</td>
<td>Read memory using ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write ALU</td>
<td>MemWrite, IorD = 1</td>
<td>Write memory using the ALUOut as address, contents of B as the data.</td>
</tr>
<tr>
<td><strong>4 bits</strong></td>
<td>PC write control</td>
<td>ALU</td>
<td>PCSource = 00, PCWrite</td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUOut-cond</td>
<td>PCSource = 01, PCWriteCond</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></td>
<td>jump address</td>
<td>PCSource = 10, PCWrite</td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td><strong>2 bits</strong></td>
<td>Sequencing</td>
<td>Seq</td>
<td>AddrCtl = 11</td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td></td>
<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></td>
<td>Dispatch 1</td>
<td>AddrCtl = 01</td>
<td>Dispatch using the ROM 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dispatch 2</td>
<td>AddrCtl = 10</td>
<td>Dispatch using the ROM 2.</td>
</tr>
</tbody>
</table>

*Figure C.5.1 (3rd Edition Appendix C – posted online)*
Instruction Fetch/decode Microcode Sequence

<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</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
<td>Dispatch 1</td>
</tr>
<tr>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td></td>
<td>Read</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First microinstruction: *(Corresponds to State 0)* Fetch, increment PC

IR ← Mem[PC]  ;  PC ← PC + 4

<table>
<thead>
<tr>
<th>Fields</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU control, SRC1, SRC2</td>
<td>Compute PC + 4. (The value is also written into ALUOut, though it will never be read from there.)</td>
</tr>
<tr>
<td>Memory</td>
<td>Fetch instruction into IR.</td>
</tr>
<tr>
<td>PCWrite control</td>
<td>Causes the output of the ALU to be written into the PC.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Go to the next microinstruction.</td>
</tr>
</tbody>
</table>

Second microinstruction: *(Corresponds to State 1)* Decode, calculate branch target

A ← R[rs]  ;  B ← R[rt]  ;  ALUout ← PC + (SignExt(imm16) x 4)

<table>
<thead>
<tr>
<th>Fields</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU control, SRC1, SRC2</td>
<td>Store PC + sign extension (IR[15:0]) &lt;&lt; 2 into ALUOut.</td>
</tr>
<tr>
<td>Register control</td>
<td>Use the rs and rt fields to read the registers placing the data in A and B.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Use dispatch table 1 to choose the next microinstruction address.</td>
</tr>
</tbody>
</table>
**LW/SW Completion Microcode Sequence**

<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>2</td>
<td>Mem1</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td></td>
<td></td>
<td>Dispatch 2</td>
</tr>
<tr>
<td>3</td>
<td>LW2</td>
<td></td>
<td></td>
<td></td>
<td>Read ALU</td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>Write MDR</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SW2</td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
</tbody>
</table>

1. **First microinstruction:** *(Corresponds to State 2)* Execute, effective memory address calculation
   \[ \text{ALUout} \leftarrow A + \text{SignEx(Im16)} \]

2. **Fields**
   - **Effect**: Compute the memory address: Register (rs) + sign-extend (IR[15:0]), writing the result into ALUOut.
   - **Sequencing**: Use the second dispatch table to jump to the microinstruction labeled either LW2 or SW2.

3. **Second microinstruction:** *(Corresponds to State 3)* LW Memory, read using ALUout
   \[ \text{MDR} \leftarrow \text{Mem[ALUout]} \]

4. **Fields**
   - **Effect**: Read memory using the ALUOut as the address and writing the data into the MDR.
   - **Sequencing**: Go to the next microinstruction.

5. **Third microinstruction:** *(Corresponds to State 4)* LW Write Back, from memory to register rt
   \[ \text{R[rt]} \leftarrow \text{MDR} \]

6. **Fields**
   - **Effect**: Write the contents of the MDR into the register file entry specified by rt.
   - **Sequencing**: Go to the microinstruction labeled Fetch.

---

State in FSM Control = Microinstruction in microprogrammed control
### LW/SW Completion Microcode Sequence

<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>Mem1</td>
<td>Add</td>
<td>A</td>
<td></td>
<td>Extend</td>
<td></td>
<td></td>
<td>Dispatch 2</td>
</tr>
<tr>
<td>LW2</td>
<td></td>
<td></td>
<td></td>
<td>Read ALU</td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write MDR</td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>SW2</td>
<td></td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
</tbody>
</table>

**Fourth microinstruction:** *(Corresponds to State 5)* SW Memory cycle, write to memory  
Mem[ALUout] ← B

<table>
<thead>
<tr>
<th>Fields</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Write memory using contents of ALUOut as the address and the contents of B as the value.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Go to the microinstruction labeled Fetch.</td>
</tr>
</tbody>
</table>
R-Type Completion Microcode Sequence

<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>6</td>
<td>Rformat1</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td></td>
<td>Fetch</td>
</tr>
</tbody>
</table>

First microinstruction: *(Corresponds to State 6)* Execute, perform ALU function

\[ \text{ALUout} \leftarrow A \text{ funct } B \]

<table>
<thead>
<tr>
<th>Fields</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU control, SRC1, SRC2</td>
<td>The ALU operates on the contents of the A and B registers, using the function field to specify the ALU operation.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Go to the next microinstruction.</td>
</tr>
</tbody>
</table>

Second microinstruction: *(Corresponds to State 7)* Write Back, ALU result in register rd

\[ R[rd] \leftarrow \text{ALUout} \]

<table>
<thead>
<tr>
<th>Fields</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register control</td>
<td>The value in \text{ALUOut} is written into the register file entry specified by the rd field.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Go to the microinstruction labeled Fetch.</td>
</tr>
</tbody>
</table>
BEQ Completion Microcode Sequence

<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>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td>ALUOut-cond</td>
<td>Fetch</td>
</tr>
</tbody>
</table>

Microinstruction: (Corresponds to State 8) Execute, compute condition, update PC

\[ \text{Zero} \leftarrow A - B \]

\[ \text{Zero} : \text{PC} \leftarrow \text{ALUout} \]

<table>
<thead>
<tr>
<th>Fields</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU control, SRC1, SRC2</td>
<td>The ALU subtracts the operands in A and B to generate the Zero output.</td>
</tr>
<tr>
<td>PCWrite control</td>
<td>Causes the PC to be written using the value already in ALUOut, if the Zero output of the ALU is true.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Go to the microinstruction labeled Fetch.</td>
</tr>
</tbody>
</table>

State in FSM Control = Microinstruction in microprogrammed control
## Jump Completion Microcode Sequence

<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>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump address</td>
<td>Fetch</td>
</tr>
</tbody>
</table>

*MicroinSTRUCTION: (Corresponds to State 9) Execute, update PC with Jump Address
PC ← Jump Address*

<table>
<thead>
<tr>
<th>Fields</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCWrite control</td>
<td>Causes the PC to be written using the jump target address.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Go to the microinstruction labeled Fetch.</td>
</tr>
</tbody>
</table>
## Microprogram for the Control Unit

<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>0</td>
<td>Fetch</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read</td>
<td></td>
<td>Dispatch 1</td>
</tr>
<tr>
<td>2</td>
<td>Mem1</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td></td>
<td></td>
<td>Dispatch 2</td>
</tr>
<tr>
<td>3</td>
<td>LW2</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td>Read ALU</td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write MDR</td>
<td></td>
<td>Fetch</td>
</tr>
<tr>
<td>5</td>
<td>SW2</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td>Write ALU</td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Rformat1</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td>Write ALU</td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td>8</td>
<td>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>ALUOut- cond</td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td>Jump address</td>
<td>Fetch</td>
<td></td>
</tr>
</tbody>
</table>

State in FSM Control = Microinstruction in microprogrammed control
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 (ISAs) on the same machine.
  - Can tailor instruction set to application.
- Compatibility:
  - Many organizations, same instruction set.

Cons:
- Possibly more costly (more hardware) 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**.
  1. An interrupt usually comes from outside the processor (I/O devices) to get the CPU’s attention to start a service routine.
  2. A trap usually originates from an event within the CPU (Arithmetic overflow, undefined instruction, system calls, etc.) 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

  i.e lowest order bit of Cause Register

• Two additional states are added to the control finite state machine to handle these exceptions.

---

3rd Edition Chapter 5.6 – See Handout

EECC550 - Shaaban
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).
    • System calls.
  – Synchronous to program execution.
  – Condition must be remedied by the system exception handler.
  – Instruction may be executed again and program continued (resuming exception) 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 OS Exception Handler

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

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

- **MIPS Approach: Fixed entry**
  - $PC \leftarrow \text{EXP}_\text{addr}$
  - Actually a very small table:
    - RESET entry
    - TLB
    - other

$\text{EXP}_\text{addr}$ = Address of OS exception handler
$\text{IV} =$ Interrupt Vector

\[\text{iv}_\text{base} \quad \text{cause} \quad \text{handler code}\]

\[\text{iv}_\text{base} \quad \text{cause} \quad \text{handler entry code}\]
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 (Exception PC):** 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 (in slide 26):
  - 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.

We need to:

1. Be able to write exception handler address (EXP_addr) into PC, increase mux to add as input 01000000 00000000 00000000 01000000 two (8000 0080 hex).

2. Must undo PC = PC + 4, since we want EPC to point to offending instruction (not its successor); PC = PC – 4

3. Write cause of exception to Cause register.
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

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

3rd Edition Figure 5.39, page 344 – See Handout
Normal Finite State Machine (FSM) Specification

```
IR ← MEM[PC]
PC ← PC + 4
0000

A ← R[rs]
B ← R[rt]

ALUout ← PC + SX
0001

```

```
“instruction fetch”
```

```
ALUout ← A fun B
0100
```

```
ALUout ← A op ZX
0110
```

```
ALUout ← A + SX
1000
```

```
ALUout ← A + SX
1011
```

```
M ← MEM[ALUout]
1001
```

```
MEM[ALUout] ← B
1100
```

```
If A = B then
PC ← ALUout
0010
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
FSM Control Specification To Handle Exceptions

IR ← MEM[PC]
PC ← PC + 4
0000

“A instruction fetch”

A ← R[rs]  B ← R[rt]
ALUout ← PC + SX
0001

“decode”

undefined instruction

EPC ← PC - 4
PC ← exp_addr
cause ← 1

overflow

R-type

ALUout ← A fun B
0100

ORj

ALUout ← A op ZX
0110

LW

ALUout ← A + SX
1000

SW

ALUout ← A + SX
1011

BEQ

If A = B then
PC ← ALUout
0010

To instruction fetch

M ← MEM[ALUout]
1001

MEM[ALUout] ← B
1100

R[rd] ← ALUout
0101

R[rt] ← ALUout
0111

R[rt] ← M
1010

To instruction fetch

To instruction fetch

To instruction fetch

EPC ← PC - 4
PC ← exp_addr
cause ← 0

undefined instruction

EPC ← PC - 4
PC ← exp_addr
cause ← 1

overflow

R-type

ALUout ← A fun B
0100

ORj

ALUout ← A op ZX
0110

LW

ALUout ← A + SX
1000

SW

ALUout ← A + SX
1011

BEQ

If A = B then
PC ← ALUout
0010

To instruction fetch

M ← MEM[ALUout]
1001

MEM[ALUout] ← B
1100

R[rd] ← ALUout
0101

R[rt] ← ALUout
0111

R[rt] ← M
1010

To instruction fetch

To instruction fetch

To instruction fetch
Control Finite State Machine
With Exception Detection
Version In Textbook
Figure 5.40

3rd Edition Figure 5.40 page 345 – See Handout