Summary of First Multiple Cycle CPU Design

Instruction fetch

Decode and Register Fetch

Memory instructions
R-type instructions
Branch instructions
Jump instruction

The Problem with FSMs as control sequencers

- They get unmanageable quickly as they grow.
  - hard to specify
  - hard to manipulate
  - error prone
  - hard to visualize
Implementing a control FSM

Control Logic

Inputs

Opcode State Reg

Outputs

Outputs

Out

Implementing a control FSM with ROM

Each line in the ROM contains control signal outputs (an operation), and next-state outputs (branch destination)

ROM

Address

 Opcode State Reg

Outputs

Outputs

Out

Implementing a control FSM with ???

ROM

Address

 Opcode State Reg

Outputs

Outputs

Out

Implementing a control FSM with a microprogram

Each line in the ROM is now a microprogram instruction, corresponding to a FSM state, with an operation (control signals) and branch destination (next state info).

µprogram in ROM

Address

 Opcode

µPC + next µPC logic

Outputs

Outputs

Out

Outputs

Outputs

Out

Outputs

Out

Outputs

Outputs

Out
Microprogram Implementation

- **Microprogram counter**
- **Address select logic**
- **Adder**
- **Datapath control outputs**

Microcode storage

Inputs from instruction register opcode field

Microprogramming

- If a microprogram is fundamentally the same as the FSM, what’s the big deal?
  - Easier to specify (program), visualize, and manipulate.
  - Allows us to think about the control symbolically

- Each microinstruction typically specifies information and information.

- Control signals are grouped into “fields” of mutually exclusive signals or operations.

### Microinstruction Fields

<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 = 01</td>
<td>Cause the ALU to add</td>
</tr>
<tr>
<td>SFC1</td>
<td>SRC1</td>
<td>PC = ALUOut, RegDst = 0</td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td>SRC2</td>
<td>SRC2</td>
<td>PC = ALUOut, MemRead = 1, MemWrite = 0</td>
<td>Read two registers using the IR as the register numbers and putting the data into registers A and B.</td>
</tr>
<tr>
<td>Register control</td>
<td>Write</td>
<td>RegWrite, RegDst = 1, MemRead = 0, MemWrite = 0</td>
<td>Write a register using the IR as the register number and the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td>Memory</td>
<td>Read</td>
<td>MemRead = 0, MemWrite = 0</td>
<td>Read memory using the PC as address; write result into IR (and the MDR).</td>
</tr>
<tr>
<td>Memory</td>
<td>Read</td>
<td>MemRead = 0, MemWrite = 1</td>
<td>Read memory using the ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td>Memory</td>
<td>Write</td>
<td>MemWrite = 0</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</td>
<td>PCSrc = 0, PCWrite = 0</td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<td>PC write control</td>
<td>ALUOut-cond</td>
<td>PCSrc = 0, PCWriteCond = 0</td>
<td>If the ALU output of the ALU is active, write the PC with the contents of the regular ALUOut.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>ALDst = 11, Addr = 10</td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td>Datum</td>
<td>Addr</td>
<td>Addr = 10, Addr = 0</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td>Datum</td>
<td>Dispatch</td>
<td>Addr = 0, Addr = 10</td>
<td>Dispatch using the ROM 1.</td>
</tr>
<tr>
<td>Datum</td>
<td>Dispatch</td>
<td>Addr = 10, Addr = 10</td>
<td>Dispatch using the ROM 2.</td>
</tr>
</tbody>
</table>

### A Microprogram

**Label**

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>Mem1</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td>Read PC</td>
<td>ALU</td>
<td>Dispatch 1</td>
</tr>
<tr>
<td>LW2</td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read ALU</td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td>SW2</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td>Write MDR</td>
<td>Write ALU</td>
<td>Seq</td>
</tr>
<tr>
<td>Rformat1</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td>Write ALU</td>
<td></td>
<td>Seq</td>
</tr>
<tr>
<td>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>Write ALU</td>
<td>ALUOut-cond</td>
<td>Fetch</td>
</tr>
<tr>
<td>JUMP1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>Jump address</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
</tbody>
</table>
Exceptions

or

Oops!

Exceptions

• There are two sources of non-sequential control flow in a processor
  – explicit branch and jump instructions
  – exceptions

• Branches are and

• Exceptions are typically and non-deterministic

• Guess which is more difficult to handle?

(control flow refers to the movement of the program counter through memory)

Exceptions and Interrupts

• the terminology is not consistent, but we’ll refer to
• exceptions as any unexpected change in control flow
• interrupts as any externally-caused exception

So then, what is:
  – arithmetic overflow
  – divide by zero
  – I/O device signals completion to CPU
  – user program invokes the OS
  – memory parity error
  – illegal instruction
  – timer signal

For now...

• The machine we’ve been designing in class can generate
  two types of exceptions.
  –
  –

• On an exception, we need to
  – save the PC (invisible to user code)
  – record the nature of the exception/interrupt
  – transfer control to OS
Handling exceptions

- PC saved in EPC (exception program counter), which the OS may read and store in kernel memory
- A status register, and a single exception handler may be used to record the exception and transfer control, or
- A vectored interrupt transfers control to a different location for each possible type of interrupt/exception

Supporting exceptions

- For our MIPS-subset architecture, we will add two registers:
  - EPC: a 32-bit register to hold the user’s PC
  - Cause: A register to record the cause of the exception
    - we’ll assume undefined inst = 0, overflow = 1
- We will also add three control signals:
  - EPCWrite (will need to be able to subtract 4 from PC)
  - CauseWrite
  - IntCause
- We will extend PCSource multiplexor to be able to latch the interrupt handler address into the PC.

Supporting exceptions in our DataPath

Supporting exceptions in our FSM
Supporting exceptions in our FSM

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
from state 1

RegDst = 1
RegWrite
MemtoReg = 0

overflow
To state 11

To state 0

Supporting exceptions in our FSM

ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite

IntCause=0
CauseWrite

state 10

Illegal instruction

state 11

IntCause=1
CauseWrite

state 12

ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource=11

fetch

Key Points

• microprogramming can simplify (conceptually) CPU control generation
• a microprogram is a small program inside the CPU that executes the individual instructions of the “real” program.
• Exception-handling is difficult in the CPU, because the interactions between the executing instructions and the interrupt are complex and unpredictable.