Multicycle Approach (No Pipelining)

- 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
  - introduce additional "internal" registers

Multicycle Approach

- We will be using functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a "subtract" instruction?
- We'll use a finite state machine for control

Outline

- Multicycle Design (5.5)
- Implementing Control & Microprogramming (5.7)
- Exceptions (5.6)
- Exceptions in a Pipeline (6.8)
### What Instructions Need to Do

<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 decode/register fetch</td>
<td>A = Reg [R[25-21]]</td>
<td>B = Reg [R[20-16]]</td>
<td>ALUOut = PC = [sign-extend] (IR[15-0]) &lt;&lt; 2</td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/ jump completion</td>
<td>ALUOut = A + B</td>
<td>ALUOut = A + [sign-extend] (IR[15-0]) if (A == B) then PC = PC + ALUOut</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [R[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: Reg[R[20-16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Multicycle Datapath

- Minimizes Hardware: 1 memory, 1 adder

### Outline

- Multicycle Design (5.5)
- Implementing Control & Microprogramming (5.7)
- Exceptions (5.6)
- Exceptions in a Pipeline (6.8)
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

Alternative: Microprogramming

- Sequence of RTL steps
  - program using microinstructions

Finite State Machine for Multicycle Control

- Implementation
- State bits
  - D-flipflops
- Control Logic
  - Comb. Block
  - Use PLA or ROM

Macroinstruction interpretation

User program plus Data
this can change!
one of these is mapped into one of these
AND microsequence
  e.g., Fetch Operands, Add, Fetch Operand(s), Calculate, Save Answer(s)
Microprogramming

• A specification methodology (alternate to FSM)
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions
  - Microassembler?

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU control</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Register control</th>
<th>Memory</th>
<th>PCKWrite 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>Byte</td>
<td>Fetch</td>
</tr>
<tr>
<td>Men1</td>
<td>Add</td>
<td>PC</td>
<td>2</td>
<td>Read ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td>Seq</td>
</tr>
<tr>
<td>LW0</td>
<td>Add</td>
<td>PC</td>
<td>2</td>
<td>Read ALU</td>
<td>Seq</td>
<td>Fetch</td>
<td>Seq</td>
</tr>
<tr>
<td>SW2</td>
<td>Add</td>
<td>PC</td>
<td>2</td>
<td>Write ALU</td>
<td>Patch</td>
<td>Fetch</td>
<td>Patch</td>
</tr>
<tr>
<td>Format1</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td>Write MDR</td>
<td>Patch</td>
<td>Fetch</td>
<td>Patch</td>
</tr>
<tr>
<td>ADD1</td>
<td>Sub</td>
<td>A</td>
<td>B</td>
<td>ALUOut-cond</td>
<td>Patch</td>
<td>Fetch</td>
<td>Patch</td>
</tr>
</tbody>
</table>

Microprogramming Pros and Cons

• 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 control memory)
• Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
• Compatibility
  - Many organizations, same instruction set
• Costly to implement
• Slow

Next time: memory

Control: Summary

Control is the hard part

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
  ROM
  “hardwired control”
  “microprogrammed control”
Outline

• Multicycle Design (5.5)

• Implementing Control & Microprogramming (5.7)

• Exceptions (5.6)

• Exceptions in a Pipeline (6.8)

Exceptions

• Exception = unprogrammed control transfer
  - system takes action to handle the exception
  - must record the address of the offending instruction
  - returns control to user
  - must save & restore user state

• Allows construction of a "user virtual machine"

Interrupt, Exception, Trap?

• Interrupts
  - caused by external events
  - asynchronous to program execution
  - may be handled between instructions
  - simply suspend and resume user program

• Traps
  - caused by internal events
  - exceptional conditions (overflow)
  - errors (parity)
  - faults (non-resident page)
  - synchronous to program execution
  - condition must be remedied by the handler
  - instruction may be retried or simulated and program continued or program may be aborted

• MIPS convention:
  - External : Interrupts
  - Internal : Exception

Exception Semantics

• MIPS architecture defines the instruction as having no effect if the instruction causes an exception.

• When get to virtual memory we will see that certain classes of exceptions must prevent the instruction from changing the machine state.

• This aspect of handling exceptions becomes complex and potentially limits performance => why it is hard
  - Precise interrupts vs Imprecise interrupts
MIPS Exceptions

- All exceptions jump to same handler code
  - "Cause" register
- We consider
  - Illegal instructions
  - Arithmetic overflows
- Handler behavior
  - Save PC of offending instruction (How? PC+4 has already been written to PC)
  - Use special register EPC (why not use $31 like jal?)
  - Set cause register appropriately (0=ILL; 1=OVF)
  - Jump to handler at fixed address

Other issues

- Vectored exceptions
  - "cause" folded into handler address
  - Different causes jump to different handlers
- User vs kernel mode
- Software issues
  - Disabling exceptions in handler
- Returning from interrupt

Outline

- Multicycle Design (5.5)
- Implementing Control & Microprogramming (5.7)
- Exceptions (5.6)
- Exceptions in a Pipeline (6.8)
Exceptions

• Semantics
  - No instruction after the exception causing instruction may execute
  - Every instruction preceding the exception causing instruction must complete execution
  - Set cause register
  - Jump to exception handler address
• Multiple instructions (exceptions) in a cycle!

Datapath modifications

• Pipeline complications
• What stage is exception detected?
  - Overflow?
    - In EX stage. Also squash (convert to nop) EX stage
  - Illegal Instruction?
    - In ID stage. squash (convert to nop) ID stage
    - Similar to RAW hazard
  - What about external interrupts?
• Overflow in instruction i, illegal instruction in instruction i+1
  - Simultaneous exceptions
  - Hardware sorting

Walkthrough (1 of 2)

• All three instructions converted to nop

Walkthrough (2 of 2)

• Fetch next instruction from handler PC (MIPS)
Pipelined Processor