DEE 1053 Computer Organization
Lecture 5: The Processor: Datapath and Control

Dr. Tian-Sheuan Chang
tschang@twins.ee.nctu.edu.tw
Dept. Electronics Engineering
National Chiao-Tung University
Outline

- Logic Design Conventions
- Basic MIPS implementation
- Building a Datapath
- Simple Implementation
- Multicycle Implementation
- Exceptions
- Microprogramming for control design
Logic Design Conventions

• Two types of logic elements
  – Combinational logic
    • Output only depends on the current input
    • Uses for ALU, multiplier, and other datapath
  – Sequential logic
    • Output depends on current inputs and current states
    • State element to store the states
Logic Design Convention: State Elements

- **Two styles**
  - Unclocked vs. Clocked
  - Clocks used in synchronous logic
    - when should an element that contains state be updated?
    - Depends on the element type

- **Two synchronous state elements**
  - Latch
    - State changes on the valid level (level-triggered)
  - D Flip Flops
    - state changes only on a clock edge (edge-triggered methodology)
Clocking Methodology

- Clocking methodology
  - Defines when signals can be read and **when they can be written**
  - Mainstream: An **edge triggered** methodology
    - Determine when data is valid and stable **relative to the clock**

- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements

```
Clock cycle
```

Clock + write control signals
Read/Write at the Same Cycle

- Use the edge-triggered methodology
  - Read at the first half cycle and write at the second half cycle
• Built using D flip-flops
Abstraction

- Make sure you understand the abstractions!
- Sometimes it is easy to think you do, when you don’t
Register File

- Note: we still use the real clock to determine when to write
Outline

- Logic Design Conventions
- **Basic MIPS implementation**
- Building a Datapath
- Simple Implementation
- Multicycle Implementation
- Exceptions
- Microprogramming for control design
The Processor: Datapath & Control

- Simplified MIPS to contain only 3 classes of instructions:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`

- Key design principles
  - Make the common case fast
  - Simplicity favors regularity
Steps in Executing an Instruction

- **Instruction Fetch (IF)**
  - Fetch the next instruction from memory

- **Instruction Decode (ID)**
  - Examine instruction to determine:
    - What operation is performed by the instruction (e.g., addition)
    - What operands are required, and where the result goes

- **Operand Fetch**
  - Fetch the operands

- **Execution (EX)**
  - Perform the operation on the operands

- **Result Writeback (WB)**
  - Write the result to the specified location

- **Next Instruction**
  - Determine where to get next instruction
Overview of Implementations

• Generic Implementation (high similarity between instructions)
  – use the program counter (PC) to supply instruction address
  – get the instruction from memory
  – read registers
  – use the instruction to decide exactly what to do

• All instructions use the ALU after reading the registers
  – memory-reference
      lw $t1, 32($t2)
  – Arithmetic
      add $t1, $t2, $t3
  – control flow
      beq $t1, $t2, L1
Abstract View of MIPS Implementation

The diagram shows the abstract view of a MIPS processor. It includes the following components:

- **PC (Program Counter)**
- **Instruction memory**
- **Instruction**
  - Data
  - Register #
  - Registers
  - Register #
- **ALU**
- **Address memory**
  - Data
  - Address

The arrows indicate the flow of data and control signals between these components.
Next Instruction: PC Datapath

- NextPC = PC + 4
Next Instruction: PC Datapath

• \((PC + 4) + \text{branch offset}\)
Abstract View of Basic Implementation

- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)
Basic Implementation (+Instruction Decode)
Outline

- Logic Design Conventions
- Basic MIPS implementation
- **Building a Datapath**
- Simple Implementation
- Multicycle Implementation
- Exceptions
- Microprogramming for control design
Datapath Components

- Common to all instructions:
  - Instruction memory
  - PC and its update
- Datapath of R-R type instructions (e.g. ADD $t1, $t2, $t3)
  - ALU
  - Register set
- Datapath of memory-reference instructions (e.g. lw $t1, offset($2))
  - ALU (for address calculation)
  - Register set
  - Sign extension unit
  - Data memory
- Datapath for a branch inst. (e.g. beq $1, $2, offset)
  - Sign extension + 2bit shifter
  - Reg
  - Adder
  - ALU (zero output)
Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.
PC Datapath and Instruction Fetch

- NextPC = PC + 4
Datapath for R-format

• R type instructions (e.g. \texttt{ADD }\$t1, \$t2, \$t3) 
  – steps:  
    • Read two registers  
      – Register file  
    • Perform an ALU operations  
      – ALU  
    • Write the result into a register  
      – Register file

• Datapath component  
  – Register file  
    • A collection of registers in which any register can be read or written by specifying the number of register (register address) in the file  
    • Needs a write control signal “RegWrite”  
    • How many ports are required?  
  – ALU
The two elements needed to implement R-format ALU operation are the register file and the ALU.

**Datapath for R-format (1)**

- **32 regs**
- **2 read ports and one write port**

The diagram illustrates the following:

- **Register numbers**
  - 5 for reading registers 1 and 2
  - 5 for writing registers

- **Data**
  - 5 for reading data 1
  - 5 for reading data 2

- **RegWrite**
  - ALU operation
  - Zero
  - ALU result

- **2 read ports and one write port**

**Only write control**

- a. Registers
- b. ALU
Datapath for R-format (2)

The datapath for R-type instruction
Datapath of Memory-Reference Instructions

- memory-reference instructions (e.g. `lw $t1, offset($t2)`)
  - Steps:
    - Read one or two registers
      - Register file
    - Memory address calculation
      - ALU + sign extension unit
    - Memory read/write
      - Data memory
    - Write the result into a register
      - Register file

- Datapath components
  - ALU (for address calculation)
  - Register set
  - Sign extension unit
    - Can you figure out why this?
  - data memory
The two units needed to implement loads and stores are the data memory unit and the sign-extension unit, in addition to the register file and ALU.
Datapath of Memory-Reference Instructions (2)

The datapath for a load or store that does a register access
Combine R-type and Memory-Reference
Datapath for Branch

- **Branch inst. (e.g. beq $1, $2, offset)**
  - Base address: PC+4
    - Add at the moment of instruction fetch
  - Offset
    - Shift left by 2-bits
      - Word alignment
  - Steps
    - Read one or two registers
      - Register file
    - Branch/Jump address calculation
      - **Adder + sign extension unit + 2-bit shift**
    - Compare the register contents to check the condition is true or not
      - ALU (zero output)
    - Write the result into PC
      - If branch, New PC = branch address
      - If jump, replacing the lower 28-bit of the PC with the 26-bit immediate values from the instructions shifted by 2-bits
The datapath for a branch uses an ALU for evaluation of the branch condition and a separate adder for computing the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the I instruction (the branch displacement) shifted left 2 bits.
Creating a Single Datapath
Outline

• Logic Design Conventions
• Basic MIPS implementation
• Building a Datapath
• **Simple Implementation**
  – ALU control
  – MUX control
• Multicycle Implementation
• Exceptions
• Microprogramming for control design
Control – The hardest Part of Design

• Purpose
  – Selecting the **operations to perform** (ALU, read/write, etc.)
  – Controlling the **flow of data** (multiplexor inputs)

• How you get these control signals:
  – Information comes from the 32 bits of the instruction
  – Example:

    add $8, $17, $18  Instruction Format:

    | op  | rs  | rt  | rd  | shamt | funct |
    |-----|-----|-----|-----|-------|-------|
    | 00000 | 10010 | 10000 | 01000 | 00000 | 100000 |

  – ALU's operation based on **instruction type** and **function code**
What Control Signals Do We Need?

- For mux
- For ALU
Design Method for Control

- Multi-level control (decoding)

- Instruction opcode: main control unit (first level)
  - ALU control
    - Sub-control for arithmetic
  - MUX control
    - Which source registers and destination registers
    - ALU input source
    - Input source of destination register
    - Input source of PC
  - Result for first level
    - Seven 1-bit control lines
    - 2-bit ALUOP control signals
    - The above control signals can be set based solely on the opcode field of the instruction
      - Exception: PCSrc (depends on the beq result)
ALU Control (1)

- Instructions using ALU
  - Load/store
    - address calculation – add
      \[ \text{lw } t1, \text{ offset}(t2) \]
  - Branch eq
    - Subtract for comparison
    - add/subtract for address calculation
      \[ \text{beq } t1, t2, \text{ offset} \]
  - R-type
    - and/or
    - set-on-less-than
ALU Control (2)

- Multi-level control (decoding)
  - Instruction opcode: main control unit – first level
    00 = lw, sw
    01 = beq,
    10 = arithmetic
  - 2nd level: function code for arithmetic: sub control
    - Reduce the size of main control but may increase the delay

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>Function Code</th>
<th>Desired ALU action</th>
<th>ALU control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>00</td>
<td>Load word</td>
<td>xxxxxxx</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>SW</td>
<td>00</td>
<td>store word</td>
<td>xxxxxxx</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>Branch equal</td>
<td>01</td>
<td>branch equal</td>
<td>xxxxxxx</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>AND</td>
<td>100100</td>
<td>and</td>
<td>0000</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>OR</td>
<td>100101</td>
<td>or</td>
<td>0001</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>set-on-less-than</td>
<td>101010</td>
<td>set-on-less-than</td>
<td>0111</td>
</tr>
</tbody>
</table>
### ALU Control (3) Truth Table For Gate Implementation

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>Function Code</th>
<th>Desired ALU action</th>
<th>ALU control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>00</td>
<td>Load word</td>
<td>xxxxxxx</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>SW</td>
<td>00</td>
<td>store word</td>
<td>xxxxxxx</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>Branch equal</td>
<td>01</td>
<td>branch equal</td>
<td>xxxxxxx</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>AND</td>
<td>100100</td>
<td>and</td>
<td>0000</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>OR</td>
<td>100101</td>
<td>or</td>
<td>0001</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>set-on-less-than</td>
<td>101010</td>
<td>set-on-less-than</td>
<td>0111</td>
</tr>
</tbody>
</table>

### FIGURE 5.13 The truth table for the three ALU control bits (called Operation). The inputs
Design the Main Control Unit

- The op field, also called the **opcode**
  - **always** contained in bits **31-26**. We will refer to this field as **Op[5-0]**.
- The two **registers** to be read
  - **always** specified by the **rs** and **rt** fields, at positions **25-21** and **20-16**.
    - This is true for the **R-type instructions**, branch equal, and for store.

### R-Format

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>15:11</td>
<td>10:6</td>
<td>5:0</td>
</tr>
<tr>
<td>31:26</td>
<td>25:21</td>
<td>20:16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### I-Format: load/store

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 or 43</td>
<td>25:21</td>
<td>20:16</td>
<td></td>
</tr>
<tr>
<td>31:26</td>
<td></td>
<td></td>
<td>15:0</td>
</tr>
</tbody>
</table>

### I-Format: branch

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>25:21</td>
<td>20:16</td>
<td>15:0</td>
</tr>
</tbody>
</table>
Design the Main Control Unit

- The base register for load and store instructions – always in bit positions 25-21 (rs).
- The 16-bit offset for branch equal, load, and store – always in positions 15-0.

<table>
<thead>
<tr>
<th>I-Format: load/store</th>
<th>35 or 43</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/store</td>
<td>31:26</td>
<td>25:21</td>
<td>20:16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I-Format: branch</th>
<th>4</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31:26</td>
<td>25:21</td>
<td>20:16</td>
<td>15:0</td>
</tr>
</tbody>
</table>
Design the Main Control Unit

- The destination register is in **one of two places**
  - For a load it is in bit positions 20-16 (rt),
  - For an R-type instruction it is in bit positions 15-11 (rd).
  - Need to **add a multiplex** or to select which field of the instruction is used to indicate the register number to be written.

<table>
<thead>
<tr>
<th>R-Format</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I-Format: load/store</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 or 43</td>
</tr>
<tr>
<td>Load/store</td>
</tr>
<tr>
<td>31:26</td>
</tr>
<tr>
<td>31:26</td>
</tr>
</tbody>
</table>
Design the Main Control Unit

What remained?
Seven 1-bit Control
The function of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes.

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted</th>
<th>Effect when asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Data memory contents at the read address are put on read data output</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Data memory contents at address given by write address is replaced by value on write data input.</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>The second ALU operand comes from the second Register file output.</td>
<td>The second ALU operand is the sign-extended lower 16-bits of the instruction.</td>
</tr>
<tr>
<td>RegDst</td>
<td>The register destination number for the Write register comes from the rt field</td>
<td>The register destination number for the Write register comes from the rd field.</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>The register on the Write register input is written into with the value on the write data input.</td>
</tr>
<tr>
<td>PCSrc</td>
<td>The PC is replaced by the output of the adder That computes the value of PC + 4.</td>
<td>The PC is replaced by the output of the adder that computes the branch target.</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>The value fed to the register write data input comes from the ALU</td>
<td>The value fed to the register write data input comes from the data memory.</td>
</tr>
<tr>
<td>Instruction</td>
<td>RegDst</td>
<td>ALUSrc</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>br</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>
Control Unit Design

Control

0p5

0p0

RegDst
ALUSrc
ALU0p1
ALU0p0
Final Control – Truth Table

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>R-format</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td>0p5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0p4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inputs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0p3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0p2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0p1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0p0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Outputs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemRead</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUOp1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUPp0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The control function for the simple one-clock implementation is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes that determine the control output setting. (Remember that Op (5-0) corresponds to bits 31-26 of the instruction, which is the opcode field.) The bottom portion of the table gives the outputs.
Datapath for R-type
Datapath for a “load” Operation
Datapath for “beq”
Design with “jump” Instruction (1)

- Implement “jump” by concatenating
  - Upper 4-bits of “PC+4”: NextPC[31:28]
  - 26-bit immediate field from instruction
  - Bits 00

{NextPC[31:28], Instruction[25:0], 2’b00}

J-Format

```
000010 | address
31:26   25:0
```
Design with “jump” Instruction (2)
Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

We are ignoring some details like setup and hold times
Performance of Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (200ps), ALU and adders (100ps), register file access (50ps)
# Critical Path for Different Instructions

<table>
<thead>
<tr>
<th>Class</th>
<th>Function units</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td></td>
<td>Register access</td>
</tr>
<tr>
<td></td>
<td>ALU</td>
</tr>
<tr>
<td></td>
<td>Register access</td>
</tr>
<tr>
<td>Load word</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td></td>
<td>Register access</td>
</tr>
<tr>
<td></td>
<td>ALU</td>
</tr>
<tr>
<td></td>
<td>Memory access</td>
</tr>
<tr>
<td></td>
<td>Register access</td>
</tr>
<tr>
<td>Store word</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td></td>
<td>Register access</td>
</tr>
<tr>
<td></td>
<td>ALU</td>
</tr>
<tr>
<td></td>
<td>Memory access</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td></td>
<td>Register access</td>
</tr>
<tr>
<td></td>
<td>ALU</td>
</tr>
<tr>
<td>Jump</td>
<td>Instruction fetch</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>Instruction memory</th>
<th>Register read</th>
<th>ALU operation</th>
<th>Data memory</th>
<th>Register write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>0</td>
<td>50</td>
<td>400</td>
</tr>
<tr>
<td>Load word</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>600</td>
</tr>
<tr>
<td>Store word</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td></td>
<td>550</td>
</tr>
<tr>
<td>Branch</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>0</td>
<td></td>
<td>350</td>
</tr>
<tr>
<td>Jump</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>
Performance of Single Cycle Machines

- Two implementation types
  - 1 cycle per instruction with fixed length cycle
  - 1 cycle per instruction with variable length cycle
  - Which one is faster? Assume 25% load, 10% stores, 45% ALU, 15% branch, 5% jump

CPU cycle for variable length

\[
\text{CPU cycle for variable length} = 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\%
\]

\[
= 447.5\text{ps}
\]

\[
\frac{\text{CPU performance}_{\text{variable}}}{\text{CPU performance}_{\text{fixed}}} \times \frac{\text{CPU Time}_{\text{fixed}}}{\text{CPU Time}_{\text{variable}}} = \frac{600}{447.5} = 1.34
\]
Single Cycle Design

- Timing problems
  - Fixed clock cycle time
    - **Significant penalty**
    - Clock cycle equal to the worst case
      - what if we had a more complicated instruction like floating point?
      - Violates “make the common case fast”
    - Acceptable only for the small instruction set
  - Variable clock cycle time
    - **Hard to implement**
    - Asynchronous design style?
      - AMULET: asynchronous ARM
      - [http://www.cs.man.ac.uk/apt/projects/processors/amulet/AMULET3i.html](http://www.cs.man.ac.uk/apt/projects/processors/amulet/AMULET3i.html)

- Area problems
  - Wasteful of area: duplicate resources
Where we are headed

- **One Solution:**
  - Shorter clock cycle time and use “multiple clock cycle”
    - Different instructions take different numbers of cycles
    - Multicycle datapath

- **Another solution**
  - Pipelining (Ch. 6)
    - Overlapping the execution of multiple instructions
Outline

• Basic MIPS implementation
• Logic Design Conventions
• Building a Datapath
• Simple Implementation
• Multicycle Implementation
• Exceptions
• Microprogramming for control design
Multicycle Implementation

• Adv.
  – Allow instructions to take **different number of cycles**
  – **Share function** units within the execution of a single instructions

Differences compared to 1-cycle design

• Only single memory unit, single ALU
  – Add **mux** for sharing

• More registers added
  – Hold value for subsequent clock cycles
  – **Extra registers**
    • Instruction register (IR) and memory data register
      – IR shall hold data until the end of execution
      – Other just hold data between clock cycles
    • A and B registers from register file
    • ALUOut register
Single Cycle Revisited
Multicycle Implementation (High level view)
Multicycle Datapath
MultiCycle Flow
Control Signals

- Control for programmer-visible state units
  - Write control for PC, memory, register, IR
  - Read control for memory
- ALU control
  - Same as the one used in single cycle datapath
- MUX control
Multicycle datapath with Control
Including Jump and Branch

• 3 sources of the PC value
  – Output of ALU value
    • PC+4 during instruction fetch
    • Directly stored into PC
  – Register ALUout
    • Branch target after computed
  – Jump address
    • \{NextPC[31:28], Instruction[25:0], 2’b00\}

• Control signals for PC
  – Unconditional: PCWrite
    • Normal increment and Jump
  – Conditional: PCWriteCond, with “zero” test
    • Branch
Complete Multicycle Datapath
Multicycle Approach (1) Implementation

- Datapath: We will be **reusing** functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data

- Control: Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a “subtract” instruction?

- What and when to assert the control signals
  - **finite state machine**
  - microprogramming
Multicycle Approach (2) Operations

- 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

---

[Diagram of instruction processing]
Instructions from ISA perspective

• Consider each instruction from perspective of ISA.
• Example:
  – The add instruction changes a register.
  – Destination register specified by bits 15:11 of instruction.
  – Instruction address specified by the PC.
  – New value is the sum (“op”) of two registers.
  – Source registers specified by bits 25:21 and 20:16 of the instruction

\[
\text{Reg[Memory[PC][15:11]]} \leq \text{Reg[Memory[PC][25:21]] \ op} \\
\text{Reg[Memory[PC][20:16]]}
\]

  – In order to accomplish this we must break up the instruction. 
    (kind of like introducing variables when programming)
Breaking down an instruction

• ISA definition of arithmetic:

\[
\text{Reg[Memory[PC][15:11]]} \leftarrow \text{Reg[Memory[PC][25:21]] op Reg[Memory[PC][20:16]]}
\]

• Could break down to:
  - IR \leftarrow \text{Memory[PC]}
  - A \leftarrow \text{Reg[IR[25:21]]}
  - B \leftarrow \text{Reg[IR[20:16]]}
  - ALUOut \leftarrow A \text{ op } B
  - \text{Reg[IR[20:16]]} \leftarrow \text{ALUOut}

• We forgot an important part of the definition of arithmetic!
  - PC \leftarrow PC + 4
Idea behind multicycle approach

• We define each instruction from the ISA perspective (do this!)

• Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)

• Introduce new registers as needed (e.g., A, B, ALUOut, MDR, etc.)

• Finally try and pack as much work into each step (avoid unnecessary cycles) while also trying to share steps where possible (minimizes control, helps to simplify solution)

• Result: Our book’s multicycle Implementation!
Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!
Step 1: Instruction Fetch

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

\[
\begin{align*}
IR &= \text{Memory}[PC]; \\
PC &= PC + 4;
\end{align*}
\]

*Can we figure out the values of the control signals?*

*What is the advantage of updating the PC now?*

- Control signals
  - IorD = 0, ALUSrcA = 0, ALUSrcB = 1, ALUOP = 00 (add), PCSource = 0,
  - Assert PCWrite
Step 2: Instruction Decode and Register Fetch

- **Read registers** rs and rt in case we need them
  - Parallel decoding due to regular instruction format
- **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
  (we are busy "decoding" it in our control logic)
- **Control:** ALUSrcA = 0, ALUSrcB = 0, ALUOp = 00 (add)
Step 3 (instruction dependent)

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

- Memory Reference:
  \[ \text{ALUOut} \leq A + \text{sign-extend}(\text{IR}[15:0]) \]
  - Control: ALUSrcA = 1, ALUSrcB = 10, ALUOp = 00 (add)

- R-type:
  \[ \text{ALUOut} \leq A \text{ op } B \]
  - Control: ALUSrcA = 1, ALUSrcB = 10, ALUOp = 10 (function field)

- Branch:
  \[ \text{if (A==B) } \text{PC} \leq \text{ALUOut} \]
  - Control: ALUSrcA = 1, ALUSrcB = 10, ALUOp = 01 (sub)
  - Assert PCCondWrite, PCSource = 01

- Jump
  - PC = \{PC[31:28], Instruction[25:0], 2'b00\}
  - Assert PCWrite
Step 4 (R-type or memory-access)

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

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

*The write actually takes place at the end of the cycle on the edge*
Write-back step

- $\text{Reg}[\text{IR}[20:16]] \leftarrow \text{MDR}$

*Which instruction needs this?*
### Summary:

<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 &lt;= Memory[PC]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC &lt;= PC + 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td>A &lt;= Reg [IR[25:21]]</td>
<td>B &lt;= Reg [IR[20:16]]</td>
<td>ALUOut &lt;= PC + (sign-extend (IR[15:0]) &lt;&lt; 2)</td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>ALUOut &lt;= A op B</td>
<td>ALUOut &lt;= A + sign-extend (IR[15:0])</td>
<td>If (A == B)</td>
<td>PC &lt;= ALUOut</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [IR[15:11]] &lt;= ALUOut</td>
<td>Load: MDR &lt;= Memory[ALUOut]</td>
<td>or Store: Memory [ALUOut] &lt;= B</td>
<td>PC &lt;= (PC [31:28], (IR[25:0]), 2'b00)</td>
</tr>
<tr>
<td>Memory read completion</td>
<td></td>
<td>Load: Reg[IR[20:16]] &lt;= MDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 5.30 Summary of the steps taken to execute any instruction class.** Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.
Simple Questions

• How many cycles will it take to execute this code?

```assembly
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label #assume not
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?
CPI in a Multicycle CPU

- **Instruction mix**
  - Assume 25% load, 10% stores, 52% ALU, 11% branch, 2% jump
  - Load: 5, stores: 4, ALU: 4, branch: 4, jump: 3

- **CPI**
  - $0.25 \times 5 + 0.1 \times 4 + 0.52 \times 4 + 0.11 \times 3 + 0.02 \times 3 = 4.12$
  - Worst case: 5 if all instructions use the same number of cycles
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
• Each state in FSM will take 1 cycle
Instruction Decoder and Register Fetch

• Fig. 5.32
Memory-Reference FSM

• Fig. 5.33
Other FSM

- R-type

Branch FSM

Jump FSM
Complete FSM

• Note:
  – don’t care if not mentioned
  – asserted if name only
  – otherwise exact value

• How many state bits will we need?
Implementation

- Implementation:

  ![Diagram of processor implementation]

  - Inputs:
    - $NS_3$, $NS_2$, $NS_1$, $NS_0$

  - Outputs:
    - $PCWrite$, $PCWriteCond$
    - $IorD$, $MemRead$, $MemWrite$
    - $IRWrite$, $MemtoReg$
    - $PCSource$, $ALUOp$
    - $ALUSrcB$, $ALUSrcA$
    - $RegWrite$, $RegDst$
    - $S_3$, $S_2$, $S_1$, $S_0$

  - Control logic
  - Instruction register (opcode field)
  - State register

  ROM or PLA
PLA Implementation

- If I picked a horizontal or vertical line could you explain it?
ROM Implementation

• ROM = "Read Only Memory"
  – values of memory locations are fixed ahead of time

• A ROM can be used to implement a truth table
  – if the address is m-bits, we can address $2^m$ entries in the ROM.
  – our outputs are the bits of data that the address points to.

\[ \begin{array}{c}
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 
\end{array}
\end{array} \]

m is the "height", and n is the "width"
ROM Implementation

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., $2^{10} = 1024$ different addresses)

- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs

- ROM is $2^{10} \times 20 = 20K$ bits  (and a rather unusual size)

- Rather wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored
• Break up the table into two parts
  — 4 state bits tell you the 16 outputs, \( 2^4 \times 16 \) bits of ROM
  — 10 bits tell you the 4 next state bits, \( 2^{10} \times 4 \) bits of ROM
  — Total: 4.3K bits of ROM

• 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 \((\#\text{inputs} \times \#\text{product-terms}) + (\#\text{outputs} \times \#\text{product-terms})\)
  For this example \( = (10 \times 17) + (20 \times 17) = 510\) PLA cells

• PLA cells usually about the size of a ROM cell (slightly bigger)
Another Implementation Style

- Complex instructions: the "next state" is often current state + 1
## 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>

### State number | Address-control action | Value of AddrCtl
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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>

---

DEE 1050 Lecture 5: Processor
Microprogramming: Simplifying the Control

- FSM for above multicycle implementation
  - Too complex if instructions no. is getting larger and more complex
  - Thousand of states are required

- Use the similar concept from programming
  - Designing the control as a program (microprogram) that implement the machine instructions in terms of microinstructions
  - Each microinstructions defines the set of datapath control signals to be asserted in a given state

- Microinstruction format
  - Partition into fields
  - Each control signals correspond to 1 bit in microinstructions
  - Each field of the microinstructions responsible for specifying a nonoverlapping set of control signals
Microprogramming

Control unit

Microcode memory

Outputs

Datapath

1

Adder

Microprogram counter

Address select logic

Instruction register opcode field

Input

Op[5:0]

PCWrite

PCWriteCond

IorD

MemRead

MemWrite

IRWrite

BWrite

MemtoReg

PCSource

ALUOp

ALUSrcB

ALUSrcA

RegWrite

RegDst

AddrCtl

Outputs

Datapath

PCWrite

PCWriteCond

IorD

MemRead

MemWrite

IRWrite

BWrite

MemtoReg

PCSource

ALUOp

ALUSrcB

ALUSrcA

RegWrite

RegDst

AddrCtl

Control unit

Microcode memory

Datapath

Instruction register

opcode field

Adder

Microprogram counter

Address select logic

Input

Op[5:0]
Microprogramming

- A specification methodology
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions

<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>ALU</td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read</td>
<td></td>
<td>Dispatch 1</td>
<td></td>
</tr>
<tr>
<td>Mem1</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td>Read ALU</td>
<td></td>
<td>Dispatch 2</td>
<td></td>
</tr>
<tr>
<td>LW2</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td>Write MDR</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>SW2</td>
<td>Rformat1</td>
<td>Func code</td>
<td>A</td>
<td>Write ALU</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>ALUOut-cond</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td>Jump address</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
</tbody>
</table>

- Will two implementations of the same architecture have the same microcode?
- What would a microassembler do?
### Microinstruction format

<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></td>
<td>Func code</td>
<td>ALUOp = 10</td>
<td>Use the instruction's function code to determine ALU control.</td>
</tr>
<tr>
<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>A</td>
<td>ALUSrcA = 1</td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td>SRC2</td>
<td>B</td>
<td>ALUSrcB = 00</td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>ALUSrcB = 01</td>
<td>Use 4 as the second ALU input.</td>
</tr>
<tr>
<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>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>Register control</td>
<td>Read</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 the ALUOut as the data.</td>
</tr>
<tr>
<td></td>
<td>Write ALU</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>Memory</td>
<td>Read PC</td>
<td>MemRead, lorD = 0</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>MemRead, lorD = 1</td>
<td>Read memory using the ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>MemWrite, lorD = 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</td>
<td>PCSource = 00, PCWrite</td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<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>jump address</td>
<td>PCSource = 10, 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>
Maximally vs. Minimally Encoded

• No encoding:
  – 1 bit for each datapath operation
  – faster, requires more memory (logic)
  – used for Vax 780 — an astonishing 400K of memory!

• Lots of encoding:
  – send the microinstructions through logic to get control signals
  – uses less memory, slower

• Historical context of CISC:
  – Too much logic to put on a single chip with everything else
  – Use a ROM (or even RAM) to hold the microcode
  – It’s easy to add new instructions
Microcode: Trade-offs

- Distinction between specification and implementation is sometimes blurred

- Specification Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel

- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers

- Implementation Disadvantages, **SLOWER now** that:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes
Historical Perspective

- In the ‘60s and ‘70s microprogramming was very important for implementing machines
- This led to more sophisticated ISAs and the VAX
- In the ‘80s RISC processors based on pipelining became popular
- Pipelining the microinstructions is also possible!
- Implementations of IA-32 architecture processors since 486 use:
  - “hardwired control” for simpler instructions (few cycles, FSM control implemented using PLA or random logic)
  - “microcoded control” for more complex instructions (large numbers of cycles, central control store)
- The IA-64 architecture uses a RISC-style ISA and can be implemented without a large central control store
Pentium 4

- Pipelining is important (last IA-32 without it was 80386 in 1985)

“Simply put, a high performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize the complex, less frequently used, instructions”
Pentium 4

- Somewhere in all that “control we must handle complex instructions

- Processor executes simple microinstructions, 70 bits wide (hardwired)
- 120 control lines for integer datapath (400 for floating point)
- If an instruction requires more than 4 microinstructions to implement, control from microcode ROM (8000 microinstructions)
- Its complicated!
Exception

• Control is the most challenging part of processor design
  – Exceptions and interrupts are one of the hardest part of control

• Exception
  – Unexpected event from within the processor
  – E.g. arithmetic flow, undefined instructions, invoke the OS from user program

• Interrupt
  – An except that comes from outside of the processor
  – E.g. I/O devices, hardware malfunctions

• These two terms are mixed in different processors

• Why are they so hard
  – Detect exceptional conditions and take the action is often on the critical timing path of a machine
How Exceptions are Handled

• Current design
  – Undefined instruction and arithmetic overflow

• Basic action to handle exceptions
  – Similar to interrupt service in X86
  – 1. Save the address of the offending instructions in EPC (exception program counter)
  – 2.1 Transfer control to the OS, take action on the causes
    • Service to the user program, or
    • Predefined instruction to an overflow, or
    • Stop program execution and report error
  – 2.2 OS terminate the program or continue its execution using the EPC
How OS Knows the Exception Reasons

• By status register
  – Used in MIPS
  – Register fields indicates the reasons of the exception

• By vector interrupts
  – Each exception vector address corresponds to its reasons
    Undefined instruction C000 0000
    Arithmetic overflow C000 0020
  – Single entry point + status register for non-vectored interrupt
Multicycle Datapath for Exception
How to Implement Exception

- **Datapath**
  - **EPC registers**
    - 32-bit registers to hold the address of the affected instructions
    - \( \text{EPC} = (\text{PC}+4) - 4 \) (use ALU)
  - **Cause registers**
    - Register to record the cause of the exception, use the low order two bits for undefined instruction and arithmetic overflow
- **Control signals:**
  - Register write control: EPCWrite, CauseWrite,
  - Interrupt/exception cause selection: IntCause
- **Exception address**
  - Constant value: 8000 0180
How Control Checks for Exceptions

- Undefined instructions
  - Opcode ≠ value of lw, sw, R-type, J and beq
- Arithmetic overflow
  - Overflow signals from ALU is used
FSM with Exceptions
Chapter 5 Summary

• If we understand the instructions… We can build a simple processor!
• If instructions take different amounts of time, multi-cycle is better
• Datapath implemented using:
  – Combinational logic for arithmetic
  – State holding elements to remember bits
• Control implemented using:
  – Combinational logic for single-cycle implementation
  – Finite state machine for multi-cycle implementation