Multi-cycle Implementation of MIPS-Lite CPU

CS 365

Multi-cycle Approach

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:
Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Unlike the single cycle implementation, our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
  - Need to specify not only instruction but also which cycle in instruction’s execution
- We’ll use a finite state machine for control

Review: finite state machines

- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)
- We’ll use a Moore machine (output based only on current state)
Review: finite state machines

• Example:

B. 21 A friend would like you to build an “electronic eye” for use as a fake security
device. The device consists of three lights lined up in a row, controlled by the outputs
Left, Middle, and Right, which, if asserted, indicate that a light should be on. Only one
light is on at a time, and the light “moves” from left to right and then from right to left,
thus scaring away thieves who believe that the device is monitoring their activity. Draw
the graphical representation for the finite state machine used to specify the electronic eye.
Note that the rate of the eye’s movement will be controlled by the clock speed (which
should not be too great) and that there are essentially no inputs.

Multicycle Approach

• 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
Changes to Datapath

- A single memory unit is used to store both instructions and data
  - need a new multiplexer (IorD) to control whether the memory is being accessed to read an instruction or read/write data (for lw/sw)
- New internal registers added to support multi-cycle operation
  - store data output by functional elements for use in subsequent cycles
  - Instruction register, Memory data register, A, B, ALUOut
    - only Instruction Register has write control input
- ALU used for all arithmetic (including computing branch target address and PC+4)
  - need a new multiplexer for top input to ALU (ALUSrcA)
  - bottom input multiplexer (ALUSrcB) to ALU extended to have four inputs (instead of two in single cycle datapath)

Multicycle Datapath with control signals
Multicycle Datapath & Control

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"

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

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

*What is the advantage of updating the PC now?*
Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

\[
\begin{align*}
A &= \text{Reg}[\text{IR}[25-21]]; \\
B &= \text{Reg}[\text{IR}[20-16]]; \\
ALUOut &= \text{PC} + (\text{sign-extend}(\text{IR}[15-0]) \ll 2); \\
\end{align*}
\]

- We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)

Instruction Fetch & Decode

<table>
<thead>
<tr>
<th>Instruction Fetch &amp; Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemRead</td>
</tr>
<tr>
<td>MemRead</td>
</tr>
<tr>
<td>ALUSrcA = 0</td>
</tr>
<tr>
<td>lorD = 0</td>
</tr>
<tr>
<td>IorD = 0</td>
</tr>
<tr>
<td>IRWrite</td>
</tr>
<tr>
<td>ALUSrcB = 01</td>
</tr>
<tr>
<td>ALUOp = 00</td>
</tr>
<tr>
<td>PCWrite</td>
</tr>
<tr>
<td>PCSource = 00</td>
</tr>
</tbody>
</table>

(Figure 5.33) Memory reference FSM
(Figure 5.34) R-type FSM
(Figure 5.35) Branch FSM
(Figure 5.36) Jump FSM
Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:
  \[ \text{ALUOut} = A + \text{sign-extend(IR}[15-0]) \]
- R-type:
  \[ \text{ALUOut} = A \text{ op B} \]
- Branch:
  \[ \text{if (A==B) PC = ALUOut} \]

Step 4 (R-type or memory-access)

- Loads and stores access memory
  \[ \text{MDR} = \text{Memory[ALUOut]} \]
  or
  \[ \text{Memory[ALUOut]} = B \]
- R-type instructions finish
  \[ \text{Reg[IR}[15-11]] = \text{ALUOut} \]

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

- `Reg[IR[20-16]] = MDR;

*What about all the other instructions?*

---

Finite state machine for memory access instructions

[Diagram showing the finite state machine for memory access instructions with states and transitions labeled as follows:]

- From state 1
- (Op = 'LW') or (Op = 'SW')
- Memory address computation
- ALUSrcA = 1
- ALUSrcB = 10
- ALUOp = 00
- MemRead, IoD = 1
- MemWrite, IoD = 1
- RegWrite, MemtoReg = 1
- RegDst = 0
- To state 0

(Figure 5.32)
Finite state machine for R-format instructions

From state 1
(Op = R-type)

Execution

6

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

R-type completion

7

RegDst = 1
RegWrite
MemtoReg = 0

To state 0
(Figure 5.32)

Finite state machine for branch instruction

From state 1
(Op = 'BEQ')

Branch completion

8

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01

To state 0
(Figure 5.32)
Finite State Machine for jump

From state 1
(Op = 'J')

Jump completion

9

PCWrite
PCSource = 10

To state 0
(Figure 5.32)

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 = Memory[PC]</td>
<td></td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td>A = Reg [IR[25-21]]</td>
<td></td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/ jump completion</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A == B) then PC = ALUOut</td>
<td>PC = PC + 4 (or 0)</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [IR[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[IR[20-16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Complete Finite State Machine

Simple Questions

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

```
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?
Additional Topics

Exceptions

- Hardest part of control is to implement exceptions & interrupts

<table>
<thead>
<tr>
<th>Type of event</th>
<th>From where?</th>
<th>MIPS terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>External</td>
<td>Interrupt</td>
</tr>
<tr>
<td>Invoke the operating system from user program</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Arithmetic overflow</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Using an undefined instruction</td>
<td>Internal</td>
<td>Exception</td>
</tr>
<tr>
<td>Hardware malfunctions</td>
<td>Internal or External</td>
<td>Exception or interrupt</td>
</tr>
</tbody>
</table>
How are exceptions handled?

- In our design, we will consider two types of exceptions
  - Arithmetic overflow
  - Execution of an undefined instruction
- Actions on exception
  - Save address of offending instruction in the Exception Program Counter (EPC)
  - Transfer control to the operating system at a pre-specified address (exception handler)
    - OS then takes appropriate action

Exception handling

- For the OS to take appropriate action, it must know the reason for the exception
- Two ways to communicate reason to OS
  - Have a Status register which holds a field that indicates the reason for the exception
  - Vectored interrupts
    - Address to which control is transferred depends upon the cause of the exception
- MIPS uses first method above; it has a register called Cause (in addition to the EPC register)
Datapath & Control with support for exceptions

Exception Handling

- Datapath additions
  - EPC, Cause (for undefined instruction, Cause = 0, arithmetic overflow Cause = 1)
- Control Signals
  - EPCWrite, CauseWrite
  - IntCause (sets the Cause register)
  - PCSrc has to be modified so that one of its sources is the OS entry point
- Three steps
  1. Write Cause
  2. EPC = PC – 4 (Have to use ALU, so need to expand multiplexors for ALUSrcA and ALUSrcB)
  3. Write PC
Datapath & Control with support for exceptions

States for handling exceptions
Complete FSM including support for exceptions

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
Graphical Specification of FSM

Finite State Machine for Control

• Implementation:

Instruction register
opcode field

State register

Outputs

Inputs

Control logic

PCWrite
PCWriteCond
IorD
MemRead
MemWrite
RegWrite
MemReg
RegDst
RegWriteCond
IRWrite
IRWriteCond

O0 O1 O2 O3 O4 O5 O6 O7

36
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.

- 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

ROM vs PLA

- 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 $(#inputs \times #product-terms) + (#outputs \times #product-terms)$
  For this example = $(10 \times 17) + (20 \times 17) = 460$ PLA cells

- PLA cells usually about the size of a ROM cell (slightly bigger)
### The Big Picture

<table>
<thead>
<tr>
<th>Initial representation</th>
<th>Finite state diagram</th>
<th>Microprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequencing control</td>
<td>Explicit next state function</td>
<td>Microprogram counter + dispatch ROMS</td>
</tr>
<tr>
<td>Logic representation</td>
<td>Logic equations</td>
<td>Truth tables</td>
</tr>
<tr>
<td>Implementation technique</td>
<td>Programmable logic array</td>
<td>Read only memory</td>
</tr>
</tbody>
</table>