Multi-Cycle Datapath

Reading

- Appendices A.7, D.3, D.4, D.5
- Practice Problems: 15, 16, 22
What are the performance limitations of this datapath?

Single Cycle Datapath

Why Multi-Cycle?

<table>
<thead>
<tr>
<th>Instruction Class</th>
<th>Instruction Fetch</th>
<th>Register Read</th>
<th>ALU Operation</th>
<th>Data Access</th>
<th>Register Write</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load word</td>
<td>2ns</td>
<td>1ns</td>
<td>2ns</td>
<td>2ns</td>
<td>1ns</td>
<td>8ns</td>
</tr>
<tr>
<td>Store word</td>
<td>2ns</td>
<td>1ns</td>
<td>2ns</td>
<td>2ns</td>
<td></td>
<td>7ns</td>
</tr>
<tr>
<td>R-format</td>
<td>2ns</td>
<td>1ns</td>
<td>2ns</td>
<td></td>
<td>1ns</td>
<td>6ns</td>
</tr>
<tr>
<td>Branch</td>
<td>2ns</td>
<td>1ns</td>
<td>2ns</td>
<td></td>
<td></td>
<td>5ns</td>
</tr>
</tbody>
</table>

- Single cycle datapath
  - Design for the worst case!
  - Need to make the cycle time = 8ns per cycle
- Multi-cycle datapath
  - Design for each individual instruction class
  - For the above example: cycle time = 2ns
  - Lw=10ns (5 cycles), sw=8ns (4 cycles), R-format=8ns(4 cycles), beq=6ns (3 cycles)
Multi-Cycle Approach

- Break up the instructions into steps, each step takes a clock 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

Instruction Execution Steps

Register-Register Instruction

Memory Read Instruction

Memory Write Instruction

Branch Instruction

Different instructions perform different operations at this step
Multi-Cycle Datapath

Instruction Fetch

Instruction Decode

Instruction Execute

Memory

Register Writeback

Functional Behavior
Complete Control Lines

Five Execution Steps

- Instruction Fetch (IF)
- Instruction Decode and Register Fetch (ID)
- Execution, Memory Address Computation, or Branch Completion (EX)
- Memory Access or R-type instruction completion (MEM)
- Write-back step (WB)
Instruction Fetch Control
IR = Memory[PC]; PC +=4

Instruction Fetch (IF)

- 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"

IR = Memory[PC];
PC = PC + 4;

Can we figure out the values of the control signals?

- IR = Memory[PC]; MemRead=1; IRWrite=1; IorD=0;
- PC = PC + 4; ALUSrcA=0; ALUSrcB=01; ALUOp=00 (add);
- PCSource=00; PCWrite=1

---

(add)
ID Stage: Assign A and B; Calculate Branch Address

Instruction Decode and Register Fetch (ID)

- Still do not have any idea what instruction it is.
- Read registers rs and rt in case we need them.
- Compute the branch address (used in next cycle in case the instruction is a branch)
- RTL:

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

\[
\text{ALUSrcA} = 0; \quad \text{ALUSrcB} = 11; \quad \text{ALUOp} = 00 \quad \text{(add)}; \quad \text{(for branch target)}
\]
Execute: Memory Type
ALUOut = A + offset (address)

Execute: R-Type
ALUOut = A op B
Execute: Branch Type
if (A==B) PC=ALUOut

Execute: Jump Type (New PC)
Execute, memory or branch (EX: instruction dependent)

- The first cycle, the operation is determined by the instruction class
- ALU is performing one of the following functions, based on instruction type

**Memory Reference:**
\[ ALUOut = A + \text{sign-extend(IR[15-0])}; \]  
\[ ALUSrcA=1; ALUSrcB=10; ALUop=00 \text{ (add)} \]

**R-type:**
\[ ALUOut = A \text{ op B}; \]  
\[ ALUSrcA=1; ALUSrcB=00; ALUop=10 \text{ (funct, inst[5:0], decides op)} \]

**Branch:**
\[ \text{if } (A==B) \text{ PC = ALUOut;} \]  
\[ PCSource=01; ALUSrcA=1; ALUSrcB=00; ALUop=01 \text{ (sub)}; \]  
\[ PCWriteCond=1; PCWrite=0; \]

**Jump:**
\[ \text{PC = } [\text{PC[31:28]} \text{ || IR[25:0]} \text{ || 2`b00}]; \]  
\[ PCSource=10; PCWrite=1; \]

MEM: Load

(19)

(20)
Memory Access or R-Type

- Loads and stores access memory

\[ \text{MDR} = \text{Memory[ALUOut]}; \text{IorD}=1; \text{MemRead}=1; \]

or

\[ \text{Memory[ALUOut]} = B; \text{IorD}=1; \text{MemWrite}=1 \]

- R-type instructions finish

\[ \text{Reg[IR[15-11]]} = \text{ALUOut}; \text{RegDst}=1; \text{MemtoReg}=0; \]
\[ \text{RegWrite} = 1 \]
• Loads and stores access memory

\[
MDR = \text{Memory[ALUOut]}; \text{IorD}=1; \text{MemRead}=1;
\]

or

\[
\text{Memory[ALUOut]} = B; \text{IorD}=1; \text{MemWrite}=1
\]

• R-type instructions finish

\[
\text{Reg[IR[15-11]} = \text{ALUOut}; \text{RegDst}=1; \text{MemtoReg}=0;
\]

\[
\text{RegWrite} = 1
\]
What about all the other instructions?

• Reg[IR[20-16]] = MDR; MemtoReg = 1; RegWrite = 1; RegDst = 0;

What about all the other instructions?
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>B = Reg [IR[20-16]]</td>
<td></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 = PC[31-28] II (IR[25-0] &lt; 2)</td>
<td></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>

Example:

(27)

Multi-Cycle Datapath

- Programs take only as long as they need to
  - Variable timing per instruction
  - Pick a base cycle time
- Re-use hardware
  - Avoid unnecessary duplication of hardware
- Revisit control
  - Combinational vs. state machine design

(28)
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?

---

Implementing the Control (D.3)

- Value of control signals is dependent upon:
  - What instruction is being executed
  - Which step is being performed

- Use the information we have 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

Alternatives for implementing the control logic?
PLA Implementation

control signal and next state truth tables

(33)

ROM Implementation

- Very inefficient implementation
  - Mostly redundant entries
- Dominated by next state function
  - Adding pipelined of FP instructions

(34)
Sequencer Implementation (D.4)

- Sequence through the state machine
  - What are my options?
- Control branches in the state machine via dispatch tables

Control Flow

Sequencing Options

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Op</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Set state to 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Dispatch with ROM 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Dispatch with ROM 2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Use the incremented state</td>
<td></td>
</tr>
</tbody>
</table>

 Dispatch ROM 2

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Op</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>0051</td>
</tr>
</tbody>
</table>

Dispatch ROM 1

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Op</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>lw</td>
<td>0110</td>
</tr>
<tr>
<td>000110</td>
<td>jep</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>bcd</td>
<td>1000</td>
</tr>
<tr>
<td>100011</td>
<td>lw</td>
<td>0010</td>
</tr>
<tr>
<td>101011</td>
<td>lw</td>
<td>0010</td>
</tr>
</tbody>
</table>
Control Implementation

Separation of control flow and data

Final Contents of Control ROM

Separation of control flow and data
Specifying Control

- Specifying all of the control bits for each state is tedious and error prone
- Use a symbolic language
  - Each state is a microinstruction
    - Organize the control bits into fields → microinstruction
- Microprograms, microcode, and micropgramming

Exceptions (A.7)

- Distinguish between internal and external events that cause an unexpected change in flow of control
  - Exceptions vs. interrupts
  - I/O, service, OS traps, errors
- Updates to the data path
  - Recording the cause of the exception and transferring control to the OS
  - Consider the impact of hardware modifications on the critical path
Exception Handling: Control

- Record the cause of the exception
  - MIPS uses the a status register referred to as the cause register
  - Record the return address in the exception program counter (EPC)
    - Of the offending instruction or the following instruction

- Transfer control
  - To a fixed address → exception handler decodes the cause
  - Vectored exceptions → target address encodes the cause

Exception Handling: Operations

- Add two registers to the datapath
  - EPC and Cause registers

- Add a state for each exception condition
  - Use the ALU to compute the EPC contents
  - Write the Cause register with exception condition
  - Update the PC with OS handler address
  - Generate control signals for each operation

- See Appendix B.7 for details of MIPS 2000/3000 implementation
The OS Interactions

- The MIPS 32 Status and Cause Registers
  - Read example in B.7
  - Manage all state saving requirements
  - Operating System handlers interrogate these registers

Exception Handling: State Machine

(43)
Given a MIPS 32 instruction sequence
- Reproduce the sequence of states generated by the controller
- Compute the execution time in cycles
- Determine all control signal values if the system is frozen at an arbitrary clock cycle

Modify the datapath and state machine to
- Add new exception conditions
- New instructions
- Extend the memory system where an access takes 2 cycles instead of 1 cycle

Modify microcode to add instructions
- Draw the multicycle datapath from memory
### Glossary

- Dispatch table
- Exception
- Finite State Machine
- Instruction decode
- Instruction fetch
- Microinstruction
- Microprogramming
- Microcode
- Multi-Cycle datapath
- Sequencer
- ROM
- ROM