Multi-Cycle Datapath

Lecture notes from MKP, H. H. Lee and S. Yalamanchili

Reading

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

1. Fetch Instruction
2. Decode Instruction
3. ALU Operation
4. Write Register

**Memory Read Instruction**

1. Fetch Instruction
2. Decode Instruction
3. Address Calculation
4. Memory Read
5. Write Register

**Memory Write Instruction**

1. Fetch Instruction
2. Decode Instruction
3. Address Calculation
4. Memory Write

**Branch Instruction**

1. Fetch Instruction
2. Decode Instruction
3. Branch Test & PC Update

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

Functional Behavior
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)
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

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

Instruction Fetch Control
IR = Memory[PC]; PC+=4

Instruction Fetch (IF)
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:

\[
\begin{align*}
A &= \text{Reg}[\text{IR}[25-21]]; \\
B &= \text{Reg}[\text{IR}[20-16]]; \\
\text{ALUOut} &= \text{PC} + (\text{sign-extend}(\text{IR}[15-0]) \ll 2); \\
\text{ALUSrcA} &= 0; \quad \text{ALUSrcB} = 11; \quad \text{ALUOp} = 00 \\
\text{(add)}; \quad \text{(for branch target)}
\end{align*}
\]
Execute: Memory Type
ALUOut = A + offset (address)

Execute: R- Type
ALUOut = A op B
- 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:**
    
    \[
    \begin{align*}
    \text{ALUOut} &= A + \text{sign-extend}(IR[15:0]); \\
    \text{ALUSrcA} &= 1; \text{ALUSrcB} = 10; \text{ALUop} = 00 \text{ (add)}
    \end{align*}
    \]

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

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

  - **Jump:**
    
    \[
    \begin{align*}
    \text{PC} &= (\text{PC}[31:28] \ || \ IR[25:0] \ || \ 2 \ 'b00); \\
    \text{PCSsource} &= 10; \text{PCWrite} = 1
    \end{align*}
    \]

(19)
MEM: Store

Memory Access or R-Type

- Loads and stores access memory

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

or

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

- 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 = Memory[ALUOut]; IorD=1; MemRead=1;

or

Memory[ALUOut] = B; IorD=1; MemWrite=1

R-type instructions finish

Reg[IR[15-11]] = ALUOut; RegDst=1; MemtoReg=0;

RegWrite =1
What about all the other instructions?

\[ Reg[IR[20-16]] = MDR; \text{ MemtoReg} = 1; \text{ RegWrite} = 1; \]
\[ \text{ RegDst} = 0; \]

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></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>A = Reg [IR[25-21]]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decode/register fetch</td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALUOut = PC + (sign-extend [IR[15-0]]) &lt;&lt; 2</td>
<td></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 = ALUOut</td>
<td>PC = PC [31-28] if (IR[25-0] &lt;&lt; 2)</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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:

---

### 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
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?

Example:

```
```

(29)

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

(30)
Graphical Specification of FSM

Control Signals: Truth Table

<table>
<thead>
<tr>
<th>Outputs</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWrite</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PWriteCond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IsD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemRead</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IRWts</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PScond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PSr0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUOp1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUOp0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ALUShiL</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUShiB0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUShiA</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RegWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RegOut</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
PLA Implementation

finite state machine for control

Alternatives for implementing the control logic?
ROM Implementation

- Very inefficient implementation
  - Mostly redundant entries

- Dominated by next state function

2^10 x 20 bit ROM

Inputs

Outputs

Instruction register opcode field

State register

PCWrite
PCWriteCond
Load
MemRead
MemWrite
IRWrite
MemtoReg
PCSource
ALUOp
ALUSubB
ALUSUBA
RegWrite
RegDest
N83
N82
N81
N80

Sequence 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>Opcode name</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 incremental table</td>
<td></td>
</tr>
</tbody>
</table>

Dispatch ROM 1

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>Rformat</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>jep</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>br</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>

Control Implementation

Instruction register opcode field

Instruction register opcode field

Instruction register opcode field

Instruction register opcode field
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 microprogramming
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 A.7 for details of MIPS 2000/3000 implementation

The OS Interactions

- The MIPS 32 Status and Cause Registers

<table>
<thead>
<tr>
<th>Status Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 15 8 4 0</td>
</tr>
<tr>
<td>Interrupt Mask</td>
</tr>
<tr>
<td>User mode</td>
</tr>
<tr>
<td>Exception Code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cause Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 15 8 6 2 0</td>
</tr>
<tr>
<td>Branch Delay</td>
</tr>
<tr>
<td>Pending Interns</td>
</tr>
<tr>
<td>Exception Code</td>
</tr>
</tbody>
</table>

- Operating System handlers interrogate these registers
- Manage all state saving requirements
- Read example in A.7
Exception Handling: State Machine

Exception Handling: Datapath
Implementing System Calls

- The machine operates in **user mode** or **kernel mode**.
- Transitions into kernel mode effected by executing system calls
- System calls implemented using the **trap instruction**
  - Vectors via a trap table to the appropriate trap handler
  - Table initialized at boot time by the OS
- More when we discuss operating systems......

Adding the trap instruction
Study Guide

• 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 e.g., state
• 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
• trap instruction
• ROM

(50)