Multi-Cycle Datapath

Reading

- Appendices A.7, D.3, D.4, D.5
- Note: Appendices A-E in the hardcopy text correspond to chapters 7-11 in the online text.
- 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></td>
</tr>
<tr>
<td>R-format</td>
<td>2ns</td>
<td>1ns</td>
<td>2ns</td>
<td></td>
<td>1ns</td>
<td>7ns</td>
</tr>
<tr>
<td>Branch</td>
<td>2ns</td>
<td>1ns</td>
<td>2ns</td>
<td></td>
<td></td>
<td>6ns</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**
  - Fetch Instruction → Decode Instruction → ALU Operation → Write Register
- **Memory Read Instruction**
  - Fetch Instruction → Decode Instruction → Address Calculation → Memory Read → Write Register
- **Memory Write Instruction**
  - Fetch Instruction → Decode Instruction → Address Calculation → Memory Write
- **Branch Instruction**
  - Fetch Instruction → Decode Instruction → Branch Test & PC Update
  - Different instructions perform different operations at this step
**Functional Behavior**

- **Instruction Fetch**
  - MemInstr (Addr)
  - MemInstr (MRd)
  - MemInstr (MWr)
  - MemInstr (RWr)

- **Instruction Decode**
  - Control Flow Instructions
  - ALU Instructions

- **Instruction Execute**
  - ALU
  - Br

- **Memory**
  - Read/Write

- **Register Writeback**

---

**Multi-Cycle Datapath**

- **Fetch Instruction**
- **Decode Instruction**
- **Addr Calc**
- **ALU OP**
- **Branch Test**

- **New internal registers**

---

(Images and text from Georgia Tech diagrams)
When to write

When to write

What to write

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

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
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:

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

\[
\text{ALUSrcA} = 0; \; \text{ALUSrcB} = 11; \; \text{ALUOp} = 00 \quad \text{(add)}; \; \text{(for branch target)}
\]
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:**
    
    \[
    \text{ALUout} = A + \text{sign-extend}\{\text{IR}[15-0]\}; \\
    \text{ALUSrcA}=1; \text{ALUSrcB}=10; \text{ALUop}=00 \text{ (add)}
    \]

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

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

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

(MEM: Load)

---

(19)

---

(20)
MEM: Store

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

  \[ \text{Reg[IR[15-11]]} = \text{ALUOut}; \text{RegDst}=1; \text{MemtoReg}=0; \]
  
  \[ \text{RegWrite} = 1 \]
Memory Access or R-Type

- 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; RegDest=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</td>
<td>A = Reg [IR[25-21]]</td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decode/register fetch</td>
<td>ALUOut = PC + (sign-extend [IR[15-0]] &lt;&lt; 2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend</td>
<td></td>
<td></td>
</tr>
<tr>
<td>computation, branch/jump</td>
<td>(IR[15-0])</td>
<td>(IR[15-0])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>Store: Memory[ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td></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 $t3$ take place?

Example:

```


```

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

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>PWriteEnable</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>IsD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IRWrite</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>MemToReg</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>PCSource1</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>PCSource0</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>0</td>
</tr>
<tr>
<td>ALUOp1</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>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>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUShL</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>ALUShR</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>ALUShS</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>
<td>0</td>
</tr>
<tr>
<td>ALUShA</td>
<td>0</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>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>
Alternatives for implementing the control logic?

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

---

<table>
<thead>
<tr>
<th>State number</th>
<th>Control word bits 17-9</th>
<th>Control word bits 8-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000001000</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>00000000000100</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>00000000000010</td>
<td>00</td>
</tr>
<tr>
<td>3</td>
<td>00000000000001</td>
<td>00</td>
</tr>
<tr>
<td>4</td>
<td>00000000000000</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>00000000000011</td>
<td>00</td>
</tr>
<tr>
<td>6</td>
<td>00000000000100</td>
<td>00</td>
</tr>
<tr>
<td>7</td>
<td>00000000000101</td>
<td>00</td>
</tr>
</tbody>
</table>

(37)
### Microinstruction format

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Length</th>
<th>Comments</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>SRC1</td>
<td>ALUOp = 01</td>
<td>Use the PC as the ALU input.</td>
<td></td>
</tr>
<tr>
<td>SRC2</td>
<td>ALUOp = 10</td>
<td>Use the instruction function code to determine ALU control.</td>
<td></td>
</tr>
<tr>
<td>Bits</td>
<td>ALUOp = 11</td>
<td>Use the output of the sign extension unit as the second ALU input.</td>
<td></td>
</tr>
</tbody>
</table>

### Microprogramming

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

- Will two implementations of the same ISA architecture have the same microcode?
- What would a micro-assembler do?
**Microcode: Trade-offs**

- **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

---

**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 instructions
  - Extend the memory system where an access takes 2 cycles instead of 1 cycle

- **Draw the multicycle datapath from memory**
Glossary

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

(43)