The Multicycle Implementation

Note: Some of the material in this lecture are
COPYRIGHT 1998 MORGAN KAUFMANN PUBLISHERS, INC.
ALL RIGHTS RESERVED.
Figures may be reproduced only for classroom or personal education
use in conjunction with our text and only when the above line is included.
The Five Cycles

- Five execution steps (some instructions use fewer)
  - **IF**: Instruction Fetch
  - **ID**: Instruction Decode (& register fetch & add PC+immed)
  - **EX**: Execute
  - **Mem**: Memory access
  - **WB**: Write-Back into registers

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R-type</strong></td>
<td>I cache</td>
<td>Decode, R-Read</td>
<td>ALU</td>
<td>PC update</td>
<td>D cache</td>
</tr>
<tr>
<td><strong>Load</strong></td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td><strong>Store</strong></td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td><strong>beq</strong></td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td>-</td>
</tr>
</tbody>
</table>
Complete Multicycle Datapath
# Summary of execution steps

<table>
<thead>
<tr>
<th>Step</th>
<th>R-type</th>
<th>Memory</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>IR = Mem[PC]</td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Instruction Decode/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>register fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address</td>
<td>ALUout = A op B</td>
<td>ALUout = A + sign-extend(IR[15-0])</td>
<td>if (A==B) then PC=ALUout</td>
</tr>
<tr>
<td>computation, branch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type</td>
<td>Reg[IR[15-11]] = ALUout</td>
<td>memory-data = Mem[ALUout] or</td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td>Mem[ALUout]=B</td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td></td>
<td>Reg[IR[20-16]] = memory-data</td>
<td></td>
</tr>
</tbody>
</table>

This is **Register Transfer Language (RTL)**

“High level” description of changes to state elements

We’ll go through these in exacting detail

And translate them to “low level” control signal settings

Modern design tools do this automatically
Cycle 1: Instruction Fetch

Datapath: IR = Memory[PC], PC = PC + 4 (may be revised later)
Control: IorD=0, MemRead=1, MemWr=0, IRwrite=1, ALUsrcA=0, etc
Control for IF cycle

MemRead
ALUsrcA = 0
IorD = 0
IRwrite
ALUsrcB = 01
ALUop = 00
PCwrite
PCsource = 00
Cycle 2: Instruction Decode (and register fetch)

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

We compute target address even though we don’t know if it will be used
- Operation may not be branch
- Even if it is, branch may not be taken

Why?
Everything up to this point must be instruction-independent, because we haven’t decoded the instruction.
The ALU, the (incremented) PC, and the immed field are now all available
Cycle 2: Instruction Decode cycle

A = Register[IR[25-21]]
B = Register[IR[20-16]]
ALUout = PC + (sign-extend (IR[15-0]) << 2)
Control for first two cycles

Instruction Fetch, *state 0*
- MemRead
- ALUsrcA = 0
- IorD = 0
- IRWrite
- ALUsrcB = 01
- ALUOp = 00
- PCWrite
- PCSource = 00

Instruction Decode, *state 1*

After cycle two, we can treat different instructions separately

This is beginning of a **Finite State Machine (FSM)** specification of control.
Cycle 3 for beq: EXecute

- In cycle 1, PC was incremented by 4
- In cycle 2, ALUout was set to branch target
- This cycle, we conditionally reset PC: if (A==B) PC=ALUout
FSM state for cycle 3 of beq

from state 1

ALUsrcA = 1
ALUsrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01

To state 0
R-type instructions

- **Cycle 3 (EXecute)**
  \[ \text{ALUout} = A \text{ op } B \]

- **Cycle 4 (WriteBack)**
  \[ \text{Reg}[\text{IR}[15-11]] = \text{ALUout} \]

R-type instruction is finished
Cycle 3: \( \text{ALUout} = A \text{ op B} \)

Cycle 4: \( \text{Reg}[\text{IR}[15-11]] = \text{ALUout} \)
R-type EXecution and WriteBack

**Cycle 3:** \( ALUout = A \ op \ B \)

**Cycle 4:** \( \text{Reg}[\text{IR}[15-11]] = ALUout \)
FSM states for R-type Instructions

from state 1

ALUsrcA = 1
ALUsrcB = 00
ALUop = 10

Execution

WriteBack

To state 0
Load and Store

- **EXecute** (cycle 3): compute address
  \[
  \text{ALUout} = A + \text{sign-extend}(\text{IR}[15-0])
  \]

- **Mem** (cycle 4): access memory
  \[
  \text{Store: Mem[ALUout]} = B \quad \text{(store is finished)}
  \]
  \[
  \text{Load: MDR = Mem[ALUout]}
  \]
  \[
  \text{("MDR" is Memory Data Register)}
  \]

- **WB** (cycle 5, only for load): write register
  \[
  \text{Reg[IR[20-16]]} = \text{MDR}
  \]

Use next 5 slides to figure out control signals!
Cycle 3 for lw and sw: Address Computation

\[ \text{ALUout} = A + \text{sign-extend}(IR[15-0]) \]
Cycle 4 for Store: Memory Access

Memory[ALUout] = B
Cycle 4 for Load: Memory Access

Memory Data Register = Memory[ALUout]
Cycle 5 for load: WriteBack

Reg[IR[20-16]] = memory-data
Memory Instruction states

from state 1

Address Computation

MemRead IorD = 1

Memory Access

MemWrite IorD = 1

write-back

RegWrite MemtoReg = 1
RegDst = 0

To state 0
Conditional branches and jumps

• Almost all conditional branches (loops, if-then) go a short distances from the current program counter.
  - we can specify a relative address in many fewer bits than an absolute address
  - e.g., beq $1, $2, 100  => if ($1 == $2) PC = PC + 100 * 4

• But we need to be able to jump to an absolute address
  - jump:  j 10000  means “set PC = 10000*4”
  - jump and link: jal 1000 means “$31 = PC + 4; PC = 1000*4”
    • used for procedure calls
  - jump register: jr $31 means “PC = $31” (used for return)
Branch and Jump Addressing Modes

• Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, target is PC-displacement mode.
  
  - If opcode is 6 bits, how many bits are available for displacement? How far can you jump?

• Jump uses pseudo-direct addressing mode. 26 bits of the address is in the instruction, the rest is taken from the PC.
Cycle 3 for Jump

\[ PC = PC[31-28] \ | \ (IR[25-0] \ll 2) \]
Cycle 3 JMP FSM state

from state 1

PCWrite
PCSource = 10

To state 0
Control of Multicycle Implementation

• We’ve been building up a Finite State Machine (FSM)

• FSM shows a succession of states, transitions between states (based on inputs), and outputs for each state.

• First two states are the same for every instruction, later states depend on opcode
The complete (but fuzzy) FSM

Instruction fetch

- Start
- MemRead
  - ALUSrcA = 0
  - IorD = 0
  - IRWrite
  - ALUSrcB = 01
  - ALUOp = 00
  - PCWrite
  - PCSource = 00

Instruction decode/register fetch

- ALUSrcA = 0
- ALUSrcB = 11
- ALUOp = 00

Execution

- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 10

Memory address computation

- Op = LW
- Op = SW

Memory access

- MemRead
  - IorD = 1

Memory access

- MemWrite
  - IorD = 1

R-type completion

- RegDst = 1
- MemtoReg = 0

Write-back step

- MemWrite
  - IorD = 1

- RegDst = 0
- RegWrite
- MemtoReg = 1

Branch completion

- (Op = R-type)

Jump completion

- (Op = BEQ)
- (Op = J)

Memory access

- MemRead
  - IorD = 1

Branch completion

- (Op = R-type)

Jump completion

- (Op = BEQ)
- (Op = J)

Write-back step

- MemWrite
  - IorD = 1

- RegDst = 0
- RegWrite
- MemtoReg = 1
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 taken
  add $t5, $t2, $t3
  sw $t5, 8($t3)
  Label:  ...

• What goes on during the 8th cycle of execution?

• In what cycle does the actual addition of $t2 and $t3 takes place?
Implementing the FSM in hardware

<table>
<thead>
<tr>
<th>Opcode</th>
<th>State Reg</th>
<th>Next State</th>
<th>Datapath control</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw, sw</td>
<td>0001</td>
<td>0010</td>
<td>10011...</td>
</tr>
<tr>
<td>lw, sw</td>
<td>0001</td>
<td>0010</td>
<td>10011...</td>
</tr>
<tr>
<td>lw, sw</td>
<td>0001</td>
<td>1000</td>
<td>10011...</td>
</tr>
<tr>
<td>lw, sw</td>
<td>0110</td>
<td>0111</td>
<td>R-type cycle 3</td>
</tr>
<tr>
<td>lw, sw</td>
<td>0111</td>
<td>0000</td>
<td>R-type cycle 4</td>
</tr>
<tr>
<td>lw, sw</td>
<td>1000</td>
<td>0000</td>
<td>lw or sw cycle 3</td>
</tr>
<tr>
<td>lw, sw</td>
<td>0010</td>
<td>0101</td>
<td>lw or sw cycle 3</td>
</tr>
</tbody>
</table>

**Inputs**
- Inputs from instruction register opcode field
- Inputs from instruction register opcode field

**Outputs**
- Outputs of combinational control logic
- Outputs of combinational control logic

**Datapath control outputs**
- PCWrite
- PCWriteCond
- MemAddr
- MemWrite
- IRWrite
- IRWrite
- MemRead
- MemWrite
- PCSource
- ALUOp
- ALUSrc
- ALUSrcA
- RegDest
- RegWrite
- Next State
- NS3
- NS2
- NS1
- NS0

**Combinational control logic**
- State register
- Instruction register opcode field
- Outputs
- Inputs

**Instruction register**
- opcode field

**Outputs**
- Outputs
- Next state
- Inputs

**State register**
- State register
- Instruction register
- Opcode
- Templates

**Control logic**
- Control logic
- Outputs
- Inputs
Implementing the FSM

- Each state is given a number (4 bits in our design)

- Datapath control signals depend only on state
  - Thus, control signals don’t change during ID stage

- Next state may depend on opcode ...
  … whenever the FSM diagram shows a choice

- Last state for each opcode returns to start state

- Can be implemented via random logic, PLA, or ROM.
ROM Implementation

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

• How many output bits are there?
  16 datapath-control, 4 next state bits = 20

• ROM is $2^{10} \times 20 = 20$K bits
  - Or, use a $2^4 \times 16$ for datapath, $2^{10} \times 4$ for next state

• Can be wasteful; for many entries, outputs are the same
  - in particular, opcode is often ignored
Later, we’ll see other implementations of the control

- **Microprograms** are an alternate to FSM’s for specifying control signals
  - FSM is like a flow chart;
  - Microprogram is a program (written in microinstructions).

- State sequencing can be done with a counter instead of an explicit “next state” function.
Multicycle CPU Key Points

- Performance can be improved by using variable-length instructions
- Saves hardware but needs additional state elements
- Control is more complex than single-cycle
- "High level" description can be in RTL
- "Low level" settings of control signals can be given by FSM (later, we’ll see microprograms)
- Can implement control in combinational logic or ROM
- Sequencing can use explicit "next state" function