Control Overview

Single-cycle implementation

- Datapath: combinational logic + I-mem + regs + D-mem + PC
- Last three written at end of cycle
- Need control - just combinational logic!
- Inputs: Instruction (I-mem out) + Zero (for beq)
- Outputs: control lines for muxes, ALUop, Write-enables

Control Overview

Fast control

- divide up work on “need to know basis”
- logic with fewer input is faster

E.g.,

Global control need not know which ALUop
ALU Control

Assume ALU uses

000 and
001 or
010 add
110 sub
111 slt (set less than)

others don’t care

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add</td>
<td>000000</td>
<td>100000</td>
</tr>
<tr>
<td>sub</td>
<td>sub</td>
<td>000000</td>
<td>100010</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>000000</td>
<td>100100</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td>000000</td>
<td>100101</td>
</tr>
<tr>
<td>slt</td>
<td>slt</td>
<td>000000</td>
<td>101010</td>
</tr>
</tbody>
</table>

ALU-ctrl = f(opcode, function)
But ..

don’t forget

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>add</td>
<td>100011</td>
<td>xxxxxx</td>
</tr>
<tr>
<td>sw</td>
<td>add</td>
<td>101011</td>
<td>xxxxxx</td>
</tr>
<tr>
<td>beq</td>
<td>sub</td>
<td>000100</td>
<td>10010</td>
</tr>
</tbody>
</table>

To simplify ALU-ctrl

ALUop = f(opcode)

2 bits       6 bits

ALU Control

10  add, sub, and, . . .
00  lw, sw
01  beq

ALU-ctrl = f(ALUop, function)

3 bits       2 bits, 6 bits

Requires only five gates plus inverters
Global Control

R-format: opcode rs rt rd shamt funct
6 5 5 5 5 6

I-format: opcode rs rt address/immediate
6 5 5 16

J-format: opcode addr
6 26
Global Control

Route instruction(25-21) read reg 1 spec
Route instruction(20-16) read reg 2 spec
Route instruction(20-16) (store) and instruction(15-11) (others)
  • write reg mux

Call instruction(31-26) op(5-0)

Global Control

Global control outputs
  • ALU-ctrl - see above
  • ALU src - R-format, beq vs. ld/st
  • MemRead - lw
  • MemWrite - sw
  • MemtoReg - sw
  • RegDst - sw dst in bits 20-16, not 15-11
  • RegWrite - all but beq and lw
  • PCSrc - beq taken
Global Control

global control outputs
  • Replace PCsrc with
  • Branch beq
  • PCSrc = Branch*Zero

what are the inputs needed to determine above signals?

Just Op(5-0)!

Global Control (Fig. 5.20)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>RegDst</th>
<th>ALUSrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrr</td>
<td>000000</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100111</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>???</td>
<td>others</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

RegDst = Op(0)  ALUSrc = OP(0)  RegWrite = Op(3) * Op(2)
Global Control

More complex with entire MIPS ISA
    • need more systematic structure
    • want to share gates between control signals

Common solution: PLA
(FYI, MIPS opcodes designed minimize PLA inputs, minterms, & outputs
    See MIPS Opcode Map (Figure A.19, p. A-54)

PLA

In AND-plane, AND selected inputs to get minterms

In OR-plane, OR selected minterms to get outputs

E.g.
Control Signals Reprise; Add Jumps?

Control Signals With Jumps (Fig. 5.29)
What’s wrong with single cycle

time/prog = instrs/prog * CPI * cycle time

P  * 1 * ?

Critical path probably lw:
i-mem, reg-read, alu, d-mem, reg-write (not to mention muxes, etc)

Other instructions faster
e. g., rrr: skip d-mem

instruction variation much worse for full ISA and real implementation
  • floating point divide
  • cache misses (what the heck is this? - chapter 7)

Single cycle implementation

Solution
  • variable clock
    • too hard to control
  • fixed short clock
    • variable cycles per instruction
Multi-cycle implementation

clock cycle = Max (i-mem, reg-read+reg-write, ALU, d-mem)

reuse combinational logic on different cycles!
  • one memory
  • ALU without other adders

But
  • control more complex (later)
  • need new registers to save values - e.g., Instr Register (IR)
    • used on later cycles
    • logic that computes them is reused

High-Level Multi-Cycle Datapath

Note

Instruction register & memory data register
One memory with address bus
One ALU w/ ALUOut
Comment on Buses

Share the buses to reduce #signals

Multiple sources driving one bus
  • ensure only one driver at a time

Like a distributed multiplexor

Multicycle Ctrl Signals Needed (Fig. 5.32)
Multi-cycle Steps

Instruction Fetch (IF): \( IR = MEM(pc); PC = PC + 4 \)

Instruction Decode (ID): \( A = \text{Reg}(IR(25-21)) \quad B = \text{Reg}(IR(20-16)) \)
   - Target = \( PC + \text{Sign-extend}(IR915-0) \ll 2 \)

Execute (EX): \( \text{ALUoutput} = A + \text{SE}(IR(15-0)) \quad # \text{lw/sw} \)
   - \( \text{ALUoutput} \ A \text{ op } B \quad # \text{rrr} \)
   - if (\( A == B \)) \( \text{Pc} = \text{target} \quad # \text{beq} \)

Memory (Mem): \( \text{Mem}(\text{ALUoutput}) = B \quad # \text{sw} \)
   - mem-data = \( \text{Mem}(\text{ALUoutput}) \quad # \text{lw} \)
   - \( \text{Reg}(IR(15-11)) = \text{ALUoutput} \quad # \text{rrr} \)

Write Back (WB): \( \text{Reg}(IR(20-16)) = \text{memory-data} \quad # \text{lw} \)
Multi-cycle Control

Function of Op(5-0) and which step

Defined as Finite State Machine (FSM) or microprogram

FSM - Appendix B

- State is combination of step and which path

![FSM Diagram]

FSM

Each state define

- control signals for datapath this cycle
- control signals to determine next state

All instructions start in same IF state

Instructions terminate by making next state IF

- after PC update, of course
Multi-cycle Example

Look at datapath of Figure 5.33
Walk and $1, $2, $3 through datapath
Look at FSM of Figure 5.42
Walk and $1, $2, $3 through FSM
Repeat for lw, sw, beq taken and not taken
(This could take half a lecture.)
Finite State Machine Figure 5.42