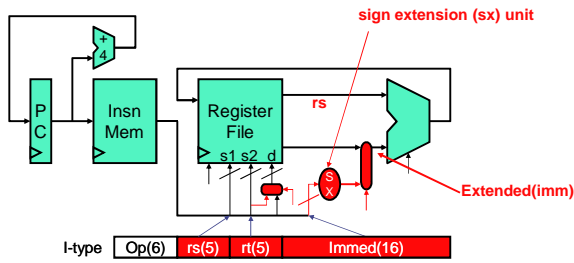
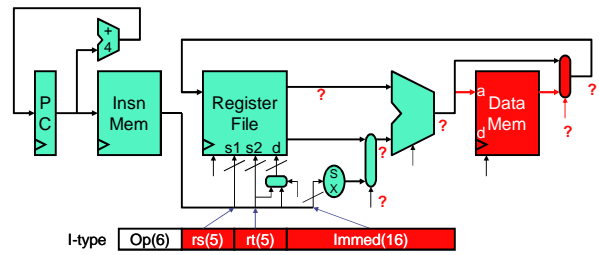


Second Instruction: addi \$rt, \$rs, imm



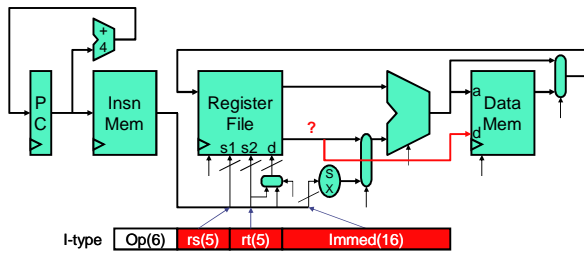
- Destination register can now be either rd or rt
- Add sign extension unit and mux into second ALU input

Third Instruction: lw \$rt, imm(\$rs)



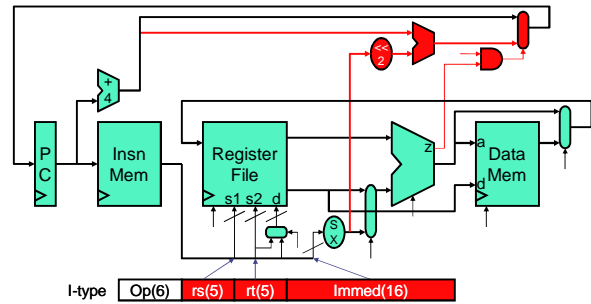
- Add data memory, address is ALU output (rs+imm)
- Add register write data mux to select memory output or ALU output

Fourth Instruction: sw \$rt, imm(\$rs)



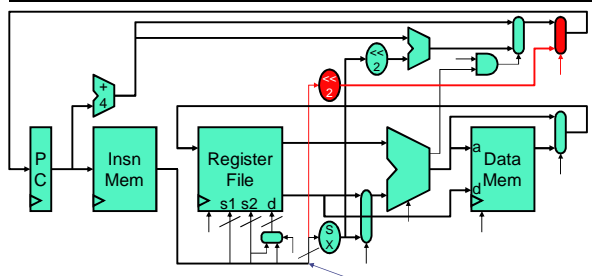
- Add path from second input register to data memory data input
- Disable RegFile's WE signal

Fifth Instruction: beq \$1, \$2, target



- Add left shift unit (why?) and adder to compute PC-relative branch target
- Add mux to do what?

Sixth Instruction: j



J-type **Op(6)** **Immed(26)**

- Add shifter to compute left shift of 26-bit immediate
- Add additional PC input mux for jump target

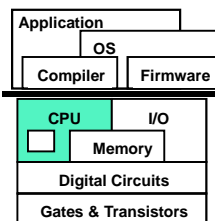
Seventh, Eight, Ninth Instructions

- Are these the paths we would need for all instructions?
 - sll** \$1,\$2,4 // shift left logical
 - Like an arithmetic operation, but need a shifter too
 - slt** \$1,\$2,\$3 // set less than (slt)
 - Like subtract, but need to write the condition bits, not the result
 - Need zero extension unit for condition bits
 - Need additional input to register write data mux
 - jal** absolute_target // jump and link
 - Like a jump, but also need to write PC+4 into \$ra (\$31)
 - Need path from PC+4 adder to register write data mux
 - Need to be able to specify \$31 as an implicit destination
 - jr** \$31 // jump register
 - Like a jump, but need path from register read to PC write mux

Clock Timing

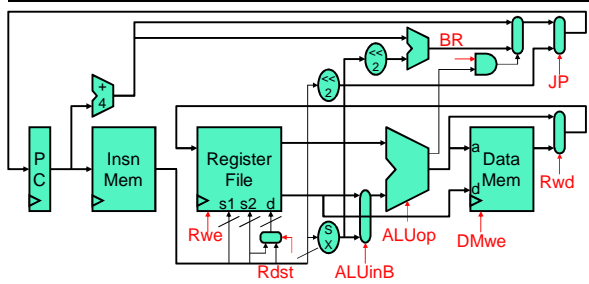
- Must deliver clock(s) to avoid races
- Can't write and read same value at same clock edge
 - Particularly a problem for RegFile and Memory
- May create multiple clock edges (from single input clock) by using buffers (to delay clock) and inverters

This Unit: Processor Design



- Datapath components and timing
 - Registers and register files
 - Memories (RAMs)
 - Clocking strategies
- Mapping an ISA to a datapath
- Control
- Exceptions

What Is Control?



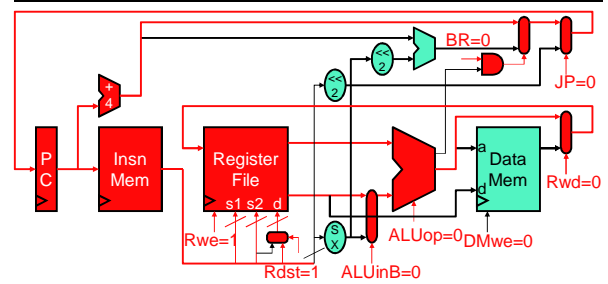
- 9 signals control flow of data through this datapath
 - MUX selectors, or register/memory write enable signals
 - Datapath of current microprocessor has 100s of control signals

© 2012 Daniel J. Sorin
from Roth

ECE152

30

Example: Control for add

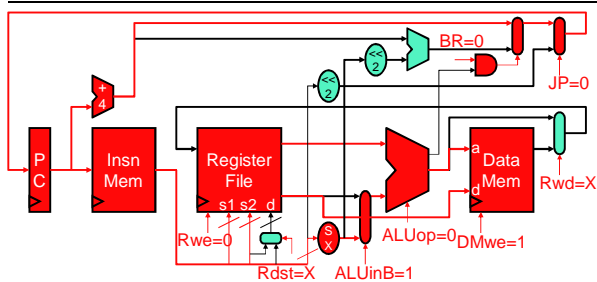


© 2012 Daniel J. Sorin
from Roth

ECE152

31

Example: Control for sw



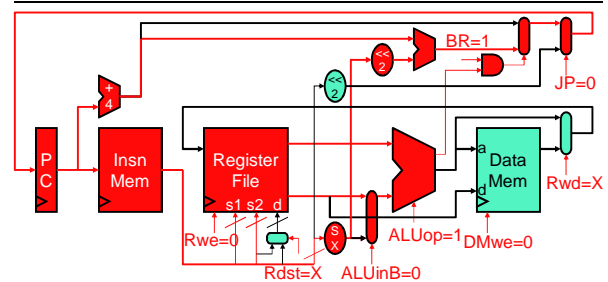
- Difference between a sw and an add is 5 signals
 - 3 if you don't count the X ("don't care") signals

© 2012 Daniel J. Sorin
from Roth

ECE152

32

Example: Control for beq \$1, \$2, target



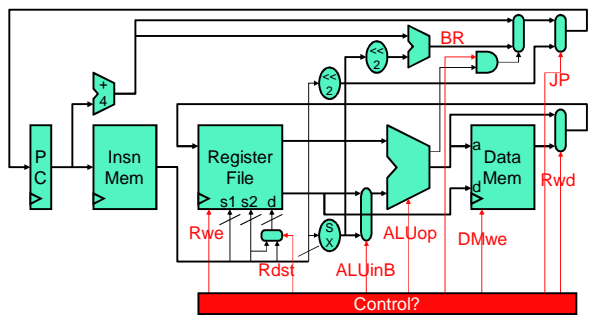
- Difference between a store and a branch is only 4 signals

© 2012 Daniel J. Sorin
from Roth

ECE152

33

How Is Control Implemented?



© 2012 Daniel J. Sorin
from Roth

ECE152

34

Implementing Control

- Each instruction has a unique set of control signals
 - Most signals are function of opcode
 - Some may be encoded in the instruction itself
 - E.g., the ALUop signal is some portion of the MIPS Func field
 - + Simplifies controller implementation
 - Requires careful ISA design
- Options for implementing control
 1. Use instruction type to look up control signals in a table
 2. Design FSM whose outputs are control signals
 - Either way, goal is same: turn instruction into control signals

© 2012 Daniel J. Sorin
from Roth

ECE152

35

Control Implementation: ROM

- **ROM (read only memory):** like a RAM but unwritable
 - Bits in data words are control signals
 - Lines indexed by opcode
- Example: ROM control for our simple datapath

	BR	JP	ALUinB	ALUop	DMwe	Rwe	Rdst	Rwd
add	0	0	0	0	0	1	1	0
addi	0	0	1	0	0	1	1	0
lw	0	0	1	0	0	1	0	1
sw	0	0	1	0	1	0	0	0
beq	1	0	0	1	0	0	0	0
j	0	1	0	0	0	0	0	0

© 2012 Daniel J. Sorin
from Roth

ECE152

36

ROM vs. Combinational Logic

- A control ROM is fine for 6 insns and 9 control signals
- A real machine has 100+ insns and 300+ control signals
 - Even "RISC"s have lots of instructions
 - 30,000+ control bits (~4KB)
 - Not huge, but hard to make fast
 - Control must be faster than datapath
- Alternative: **combinational logic**
 - ECE 52 strikes back!
 - Exploits observation: many signals have few 1s or few 0s

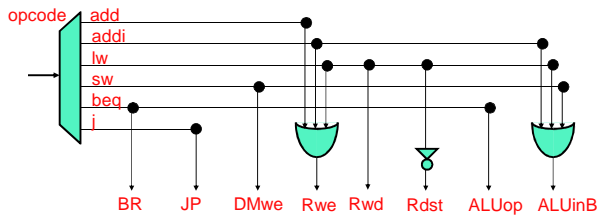
© 2012 Daniel J. Sorin
from Roth

ECE152

37

Control Implementation: Combinational Logic

- Example: combinational logic control for our simple datapath

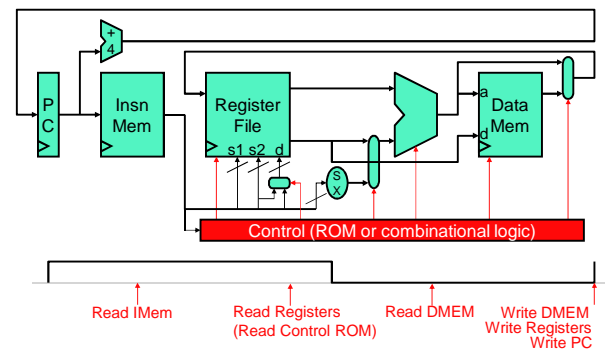


© 2012 Daniel J. Sorin
from Roth

ECE152

38

Datapath and Control Timing

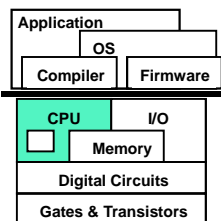


© 2012 Daniel J. Sorin
from Roth

ECE152

39

This Unit: Processor Design



- Datapath components and timing
 - Registers and register files
 - Memories (RAMs)
 - Clocking strategies
- Mapping an ISA to a datapath
- Control
- Exceptions

© 2012 Daniel J. Sorin
from Roth

ECE152

40

Exceptions

- **Exceptions and interrupts**
 - Infrequent (exceptional!) events
 - I/O, divide-by-0, illegal instruction, page fault, protection fault, ctrl-C, ctrl-Z, timer
 - Handling requires intervention from operating system
 - End program: divide-by-0, protection fault, illegal insn, ^C
 - Fix and restart program: I/O, page fault, ^Z, timer
 - Handling should be transparent to application code
 - Don't want to (can't) constantly check for these using insns
 - Want "Fix and restart" equivalent to "never happened"

© 2012 Daniel J. Sorin
from Roth

ECE152

41

Exception Handling

- What does exception handling look like to software?
 - When exception happens...
 - Control transfers to OS at pre-specified exception handler address
 - OS has privileged access to registers user processes do not see
 - These registers hold information about exception
 - Cause of exception (e.g., page fault, arithmetic overflow)
 - Other exception info (e.g., address that caused page fault)
 - PC of application insn to return to after exception is fixed
 - OS uses privileged (and non-privileged) registers to do its "thing"
 - OS returns control to user application
- Same mechanism available programmatically via SYSCALL

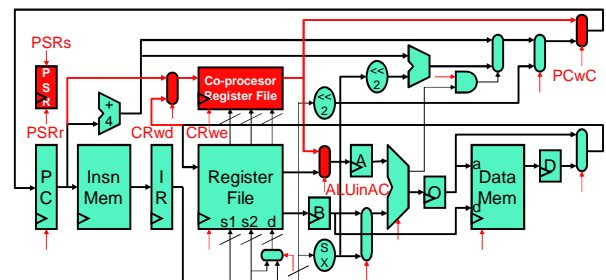
MIPS Exception Handling

- MIPS uses registers to hold state during exception handling
 - These registers live on "coprocessor 0"
 - \$14: EPC (holds PC of user program during exception handling)
 - \$13: exception type (SYSCALL, overflow, etc.)
 - \$8: virtual address (that produced page/protection fault)
 - \$12: exception mask (which exceptions trigger OS)
- Exception registers accessed using two **privileged** instructions `mfc0`, `mtc0`
 - Privileged = user process can't execute them
 - `mfc0`: move (register) from coprocessor 0 (to user reg)
 - `mtc0`: move (register) to coprocessor 0 (from user reg)
- Privileged instruction `ret` restores user mode
 - Kernel executes this instruction to restore user program

Implementing Exceptions

- Why do architects care about exceptions?
 - Because we use datapath and control to implement them
 - More precisely... to implement aspects of exception handling
 - Recognition of exceptions
 - Transfer of control to OS
 - Privileged OS mode

Datapath with Support for Exceptions



- Co-processor register (CR) file needn't be implemented as RF
 - Independent registers connected directly to pertinent muxes
- **PSR (processor status register):** in privileged mode?

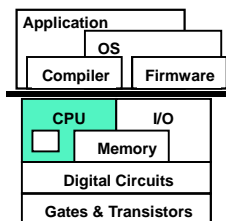
Summary

- We now know how to build a fully functional processor
- But ...
 - We're still treating memory as a black box (actually two green boxes, to be precise)
 - Our fully functional processor is slow. Really, really slow.

"Single-Cycle" Performance

- Useful metric: cycles per instruction (CPI)
- + Easy to calculate for single-cycle processor: $CPI = 1$
 - $\text{Seconds/program} = (\text{insns/program}) * 1 \text{ CPI} * (N \text{ seconds/cycle})$
 - **ICQ: How many cycles/second in 3.8 GHz processor?**
- Slow!
 - Clock period must be elongated to accommodate longest operation
 - In our datapath: lw
 - Goes through five structures in series: insn mem, register file (read), ALU, data mem, register file again (write)
 - No one will buy a machine with a slow clock
 - Not even your grandparents!

This Unit: Processor Design



- Datapath components and timing
 - Registers and register files
 - Memories (RAMs)
 - Clocking strategies
- Mapping an ISA to a datapath
- Control

Next up: Pipelining