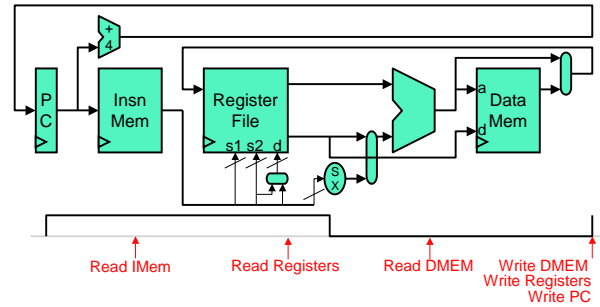


Clock Timing

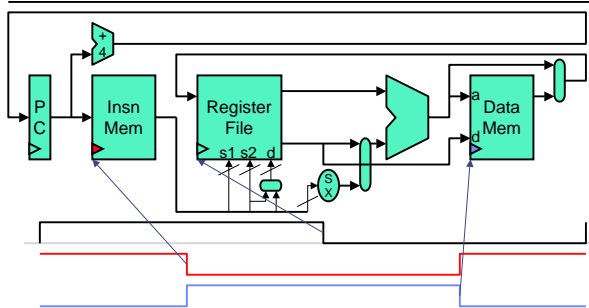
- Must deliver clock(s) to avoid races
- Can't write and read same value at same clock edge
 - Particularly a problem for RegFile and Memory
- May create multiple clock edges (from single input clock) by using buffers (to delay clock) and inverters

"Continuous Read" Datapath Timing

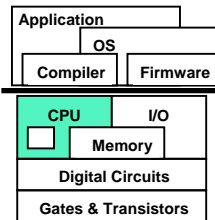


- This works because writes (PC, RegFile, DMEM) are independent
- And because no read logically follows any write

"Edge Read" Datapath Timing

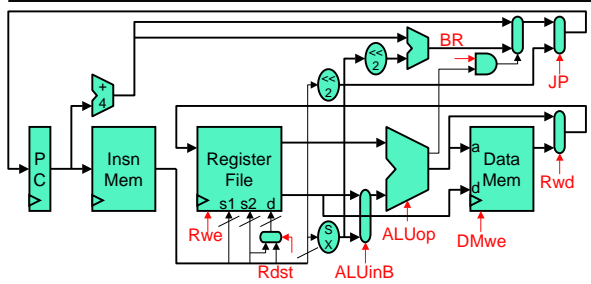


This Unit: Processor Design



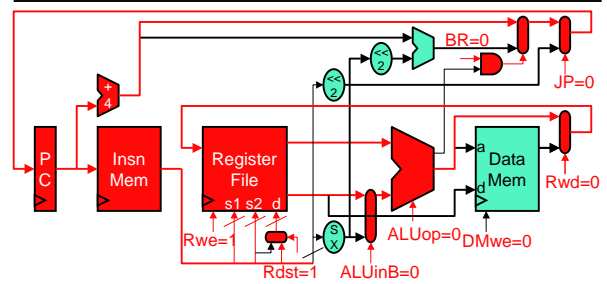
- Datapath components and timing
 - Registers and register files
 - Memories (RAMs)
 - Clocking strategies
- Mapping an ISA to a datapath
- Control
 - Single-cycle control
 - RAM/PLA
 - Multi-cycle control
 - RAM/PLA
 - Micro-programmed control
 - Implementing exceptions using control

What Is Control?

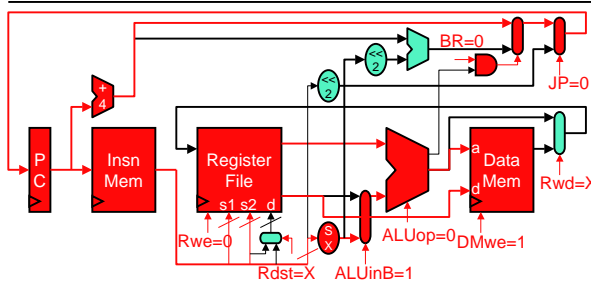


- 9 signals control flow of data through this datapath
- MUX selectors, or register/memory write enable signals
- Datapath of current microprocessor has 300-500 control signals

Example: Control for add

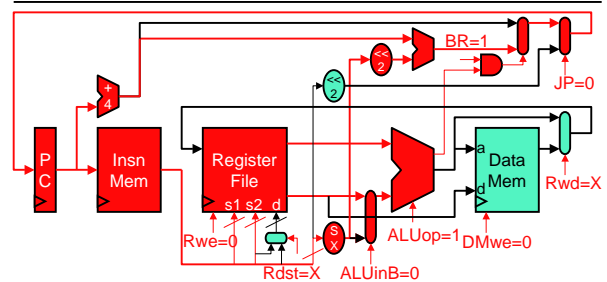


Example: Control for sw



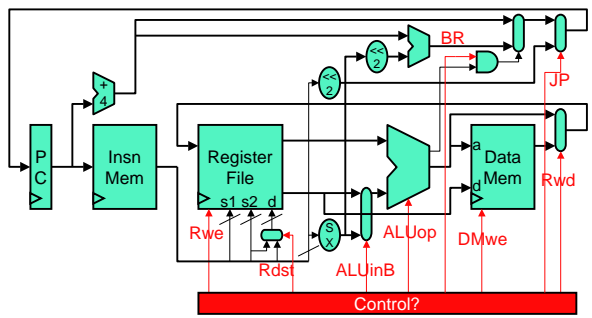
- Difference between a sw and an add is 5 signals
- 3 if you don't count the X ("don't care") signals

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



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

How Is Control Implemented?



© 2008 Daniel J. Sorin
from Roth

ECE152

37

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

© 2008 Daniel J. Sorin
from Roth

ECE152

38

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

opcode	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

© 2008 Daniel J. Sorin
from Roth

ECE152

39

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

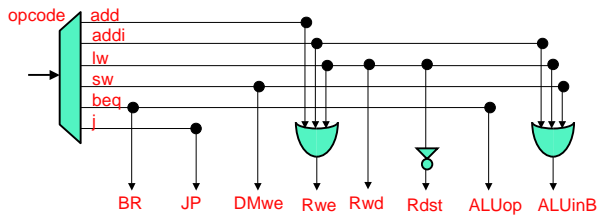
© 2008 Daniel J. Sorin
from Roth

ECE152

40

Control Implementation: Combinational Logic

- Example: combinational logic control for our simple datapath

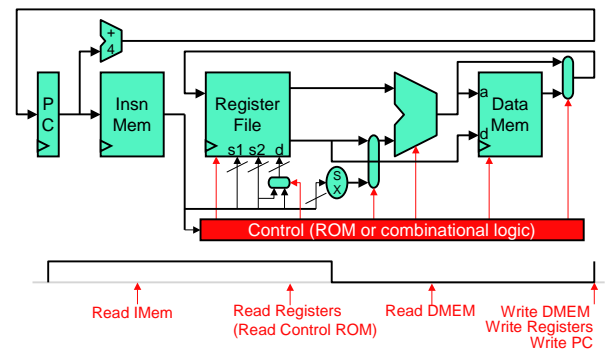


© 2008 Daniel J. Sorin
from Roth

ECE152

41

Datapath and Control Timing



© 2008 Daniel J. Sorin
from Roth

ECE152

42

"Single-Cycle" Performance

- Useful metric: cycles per instruction (CPI)
- + Easy to calculate: $CPI = 1$
 - Seconds/program = (insns/program) * 1 CPI * (N 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!
- Hard to implement long operations (e.g., multiply)

© 2008 Daniel J. Sorin
from Roth

ECE152

43

A Multi-Cycle Design

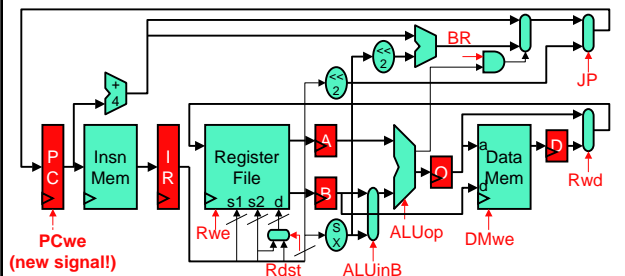
- Idea: **multi-cycle design**
 - + Speed up clock speed
 - Each insn takes multiple cycles to traverse datapath
 - + Shorter insns take fewer cycles
 - Long insns (e.g., multiply) take as many cycles as they need
 - All modern microprocessors are multi-cycle (for pipelining)
- Datapath: can use basically the same one
 - Can also optimize: insn can use a structure more than once
 - E.g.: use single memory for insns and data
 - E.g.: use single ALU for branch condition, target calculation
 - Book does it this way, we won't in the slides
 - **Need to isolate combinational stages from one another**
- Control: signals must be "staged"

© 2008 Daniel J. Sorin
from Roth

ECE152

44

Multi-Cycle Design



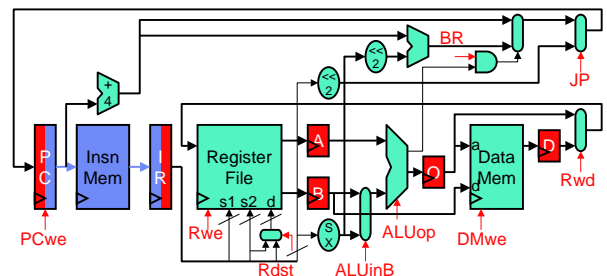
- First order of business: break datapath into stages (5 here)
 - Make each stage roughly same number of gate levels (ICQ: why?)
 - Separate stages using **registers** (ICQ: why?)

© 2008 Daniel J. Sorin
from Roth

ECE152

45

MC Stage I: Fetch



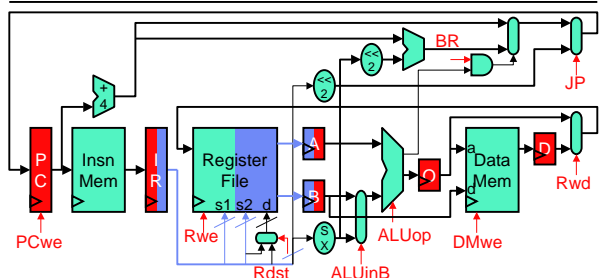
- Access I-mem using PC, write insn into insn register (IR)
- How do we prevent things from happening in later stages?

© 2008 Daniel J. Sorin
from Roth

ECE152

46

MC Stage II: Decode/RegRead



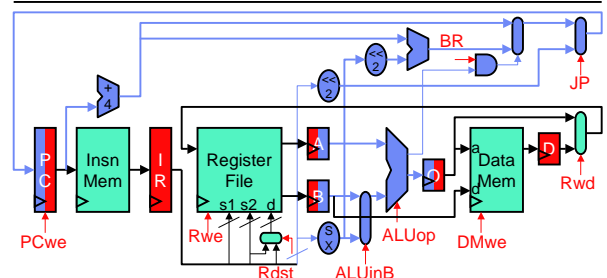
- Read registers RS1/RS2 from register file, write into A/B

© 2008 Daniel J. Sorin
from Roth

ECE152

47

MC Stage III: ALU/BeqJComplete

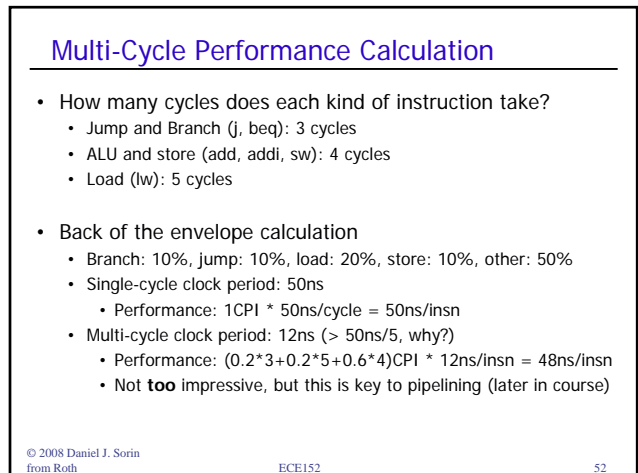
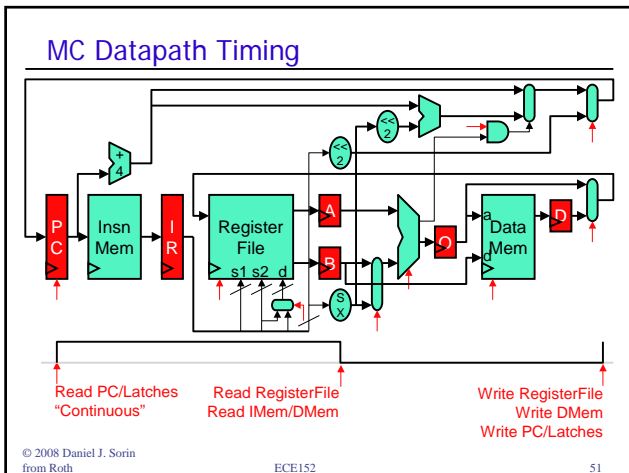
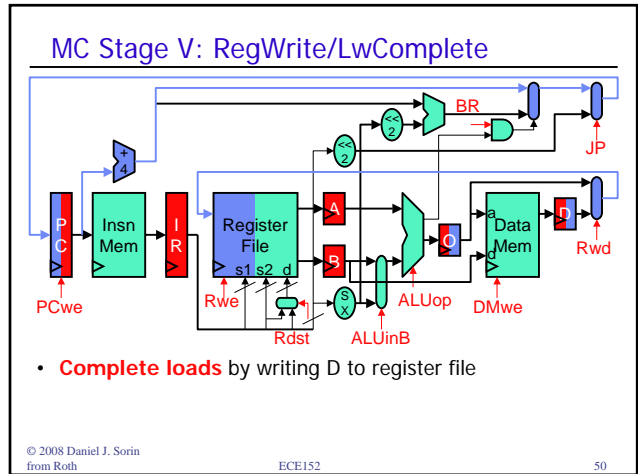
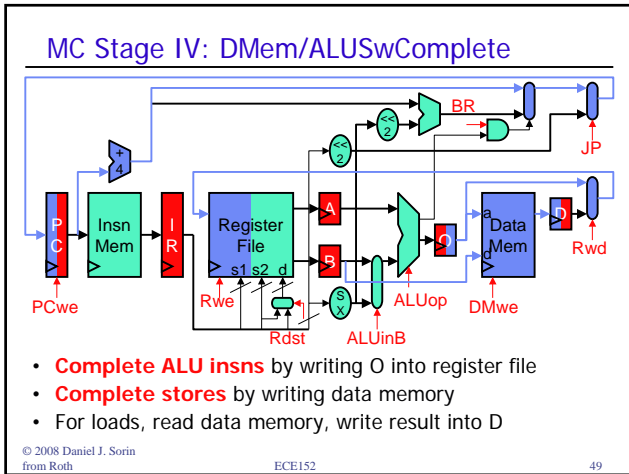


- Do ALU operation, store result into O
- **Complete jumps** by writing PC using jump target
- **Complete branches** by writing PC using ALU condition

© 2008 Daniel J. Sorin
from Roth

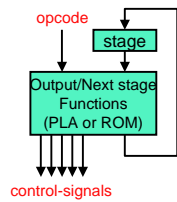
ECE152

48



Multi-Cycle Control

- Single-cycle: opcode → control-signals
 - Implementation: ROM (or PLA) or combinational logic
- Multi-cycle: opcode+stage → control-signals+next-stage
 - Implementation: FSM (ECE 52 returns yet again)



© 2008 Daniel J. Sorin
from Roth

ECE152

53

Multi-Cycle Control FSM

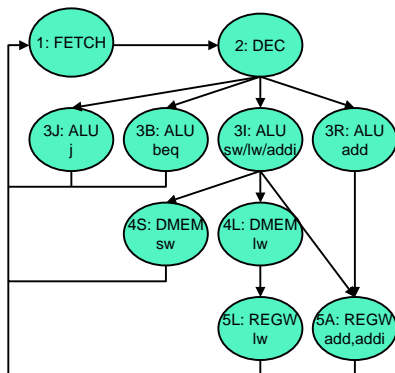
1. **Fetch**: one state for all insns (next state: 2)
2. **Decode**: one state for all insns (next state: 3R,3I,3B, or 3J)
3. **ALU**
 - **R**: add (next state: 5A)
 - **I**: lw,sw,addi (next state: 4L, 4S, or 5A)
 - **B**: beq (complete, next state: 1)
 - **J**: j (complete, next state: 1)
4. **DataMemory**
 - **S**: sw (complete, next state: 1)
 - **L**: lw (next state: 5L)
5. **RegisterWrite**
 - **A**: add, addi (complete, next state: 1)
 - **L**: lw (complete, next state: 1)

© 2008 Daniel J. Sorin
from Roth

ECE152

54

MC Control FSM: Graphically



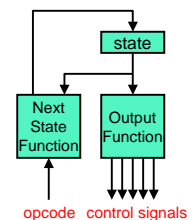
© 2008 Daniel J. Sorin
from Roth

ECE152

55

MC FSM Implementation Strategy

- One FSM state per opcode/stage pair
 - Not one state per stage
- Split **Output** function from **Next State** function
 - Output function: Output(state) → control signals
 - Next state function: NextState(state,opcode) → state



© 2008 Daniel J. Sorin
from Roth

ECE152

56

MC FSM Output Function

- **Output function (control signals):** ROM or PLA
 - Why do we need separate states 1, 2, 3R and 4L?

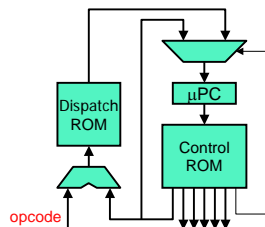
	PCwe	BR	JP	ALUinB	ALUop	DMwe	Rwe	Rdst	Rwd
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3R	0	0	0	0	0	0	0	0	0
3I	0	0	0	1	0	0	0	0	0
3B	1	1	0	1	1	0	0	0	0
3J	1	0	1	0	0	0	0	0	0
4L	0	0	0	0	0	0	0	0	0
4S	1	0	0	0	0	1	0	0	0
5A	1	0	0	0	0	0	1	1	0
5L	1	0	0	0	0	0	1	0	1

MC FSM Next State Function

- Hard part: multi-way "branch" in one cycle
- Option I: **ROM** with #states*#opcodes entries
 - Huge, many wasted entries
- Option II: **combinational logic/PLA**
 - Not intuitive, could also be huge
- Option III: **micro-program**
 - Notice that control FSM looks like a small processor
 - Control-signals are "micro" instructions (μ insn)
 - **States are "micro" program counters (μ PC)**
 - Sometimes go to next sequential μ PC
 - Sometimes unconditionally jump to a non-sequential μ PC
 - Sometimes branch to another μ PC based on opcode

MC FSM: Micro-programmed Design

- Output function ROM (**control ROM**) with #states entries
 - Two extra fields: action, dest
 - action==jump? next_ μ PC = dest
 - action==branch? next_ μ PC = **dispatch_ROM**[dest+opcode]
- + Only multi-way branches require storage proportional to #opcodes



MC FSM Control ROM

- Two multi-way branch points: states 2 and 3I
 - State 2 branch table begins at dispatch ROM index 0
 - State 3I branch table begins at dispatch ROM index 6 (6 insns)

μ PC	PCwe	BR	JP	ALUinB	ALUop	DMwe	Rwe	Rdst	Rwd	action	dest
0(1)	0	0	0	0	0	0	0	0	0	jump	1
1(2)	0	0	0	0	0	0	0	0	0	branch	0
2(3R)	0	0	0	0	0	0	0	0	0	jump	3
3(5A)	1	0	0	0	0	0	1	1	0	jump	0
4(3I)	0	0	0	1	0	0	0	0	0	branch	6
5(3B)	1	1	0	1	1	0	0	0	0	jump	0
6(3J)	1	0	1	0	0	0	0	0	0	jump	0
7(4S)	1	0	0	0	0	1	0	0	0	jump	0
8(4L)	0	0	0	0	0	0	0	0	0	jump	9
9(5L)	1	0	0	0	0	0	1	0	1	jump	0

MC FSM Dispatch ROM

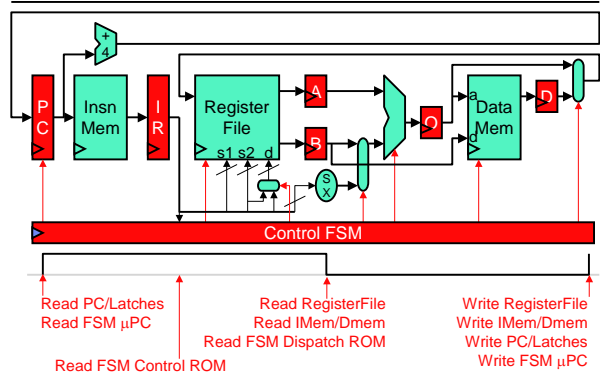
	next_μPC
0 (2+add)	2(3R)
1 (2+addl)	4(3I)
2 (2+lw)	4(3I)
3 (2+sw)	4(3I)
4 (2+beq)	5(3B)
5 (2+j)	6(3J)
6 (3l+add)	-
7 (3l+addl)	3(5A)
8 (3l+lw)	8(4L)
9 (3l+sw)	7(4S)
10 (3l+beq)	-
11 (3l+j)	-

- 2 multi-way branch points
 - State 2
 - State 3I
- 6 opcodes
- $2 * 6 = 12$ total entries
- Analogous to a "jump table" used to implement switches in C/C++/Java

Hardwired vs. Micro-programmed

- Distinction is fuzzy, somewhat arbitrary, mostly historical
- **Micro-programmed control**
 - Implemented using ROMs/RAMs
 - Indirect next_state function: "here's how to compute next state"
 - Slower ... but can do complex instructions
 - Multi-cycle execution (of control)
- **Hardwired control**
 - Implemented using random logic ("hardwired" can't re-program)
 - Direct next_state function: "here is the next state"
 - Faster ... for simple instructions (speed is function of complexity)
 - Single-cycle execution (of control)
- Machines today (e.g., Pentiums) use combination of both
 - Hardwired for simple insns, micro-programming for complex ones

Microcoded Datapath and Control Timing



Adding Multi-Cycle Operations

- How do we add a 4-stage multiplier to the datapath?
 - Not too tough given what we already know
 - Add multiplier to datapath
 - Add appropriate buses, muxes, and control signals
 - Add appropriate states and transitions to control FSM
 - How many new states in this case?
 - Any multi-way branches?
- Try to work this out for yourself

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"

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

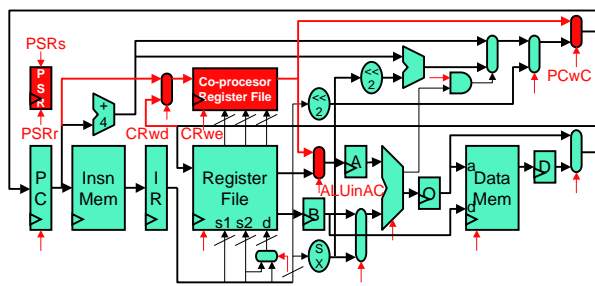
MIPS Exception Handling

- 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 **rfe** 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 file need not be implemented as RF
 - Independent registers connected directly to pertinent muxes
- PSR (processor status register): in privileged mode?

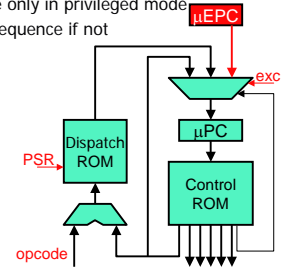
© 2008 Daniel J. Sorin
from Roth

ECE152

69

Control Support for Exceptions

- Hardwired exception μ PC (μ EPC) input to μ PC MUX
 - Allows control μ program to jump to exception handling sequence
- PSR input to dispatch ROM
 - Allows certain insns to execute only in privileged mode
 - Jumps to exception handling sequence if not
- In hardwired control?



© 2008 Daniel J. Sorin
from Roth

ECE152

70

New Control ROM Entries

- Handle exception micro-program sequence
 - E1: write PC into cop_regfile[EPC]
 - E2: write cause into cop_regfile[ECAUSE]
 - E3: write exception handler address to PC, setting PSR to 1
 - Start fetching at exception handler PC (by jumping to state 0)
- When exception occurs, μ program jumps to this sequence
 - In the middle of whatever instruction it is currently executing!!

μ PC	PCwe	...	PCwC	CRwe	Rdst	Rwd	PSRs	PSRr	action	dest
10(E1)	0		0	1	14	0	0	0	jump	11
11(E2)	0		0	1	12	2	0	0	jump	12
12(E3)	1		1	0	0	0	1	0	jump	0

© 2008 Daniel J. Sorin
from Roth

ECE152

71

Supporting Privileged Instructions

- Implemented using a new μ PC sequencing directive
 - $action == branch_psr?$ $\mu PC = dispatch_ROM[dest + (op < 1) + PSR]$
- Example: assume new privileged insn **mfc0**
 - PSR=1? μ program executes mfc0 as normal instruction
 - PSR=0? μ program jumps to exception handling code
- How would you support SYSCALL?

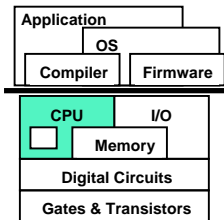
μ PC	...	action	dest	next_ μ PC
0(1)		jump	1	0 (2+add+0) 2(3R)
1(2)		branch_psr	0	1 (2+add+1) 2(3R)
2(3R)		jump	3	...
3(5A)		jump	0	...
4(3I)		branch	14	12 (2+mcf0+0) 10(E1)
5(3B)		jump	0	13 (2+mcf0+1) 2(3R)
...				14 (31+add) 3(5A)

© 2008 Daniel J. Sorin
from Roth

ECE152

72

This Unit: Processor Design



- Datapath components and timing
 - Registers and register files
 - Memories (RAMs)
 - Clocking strategies
- Mapping an ISA to a datapath
- Control
 - Single-cycle control
 - RAM/PLA
 - Multi-cycle control
 - RAM/PLA
 - Micro-programmed control
 - Implementing exceptions using control

Next up: Pipelining