CS152 – Computer Architecture and Engineering
Lecture 12 – Control Wrap up: Microcode, Interrupts, RAW/WAR/WAW

2003-10-02

Dave Patterson
(www.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs152/

Pipelining Review

- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Data hazards must be handled carefully:
  - RAW data hazards handled by forwarding
  - WAW and WAR hazards don't exist in 5-stage pipeline
- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism

Outline

- RAW, WAR, WAW: 2nd Try
- Interrupts and Exceptions in MIPS
- How to handle them in multicycle control?
- What about pipelining and interrupts?
- Microcode: do it yourself microprogramming

3 Generic Data Hazards: RAW, WAR, WAW

- Read After Write (RAW)
  Instr$_J$ tries to read operand before Instr$_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.
- Forwarding handles many, but not all, RAW dependencies in 5 stage MIPS pipeline

3 Generic Data Hazards: RAW, WAR, WAW

- Write After Read (WAR)
  Instr$_J$ writes operand *before* Instr$_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "anti-dependence" by compiler writers. This results from "reuse" of the name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

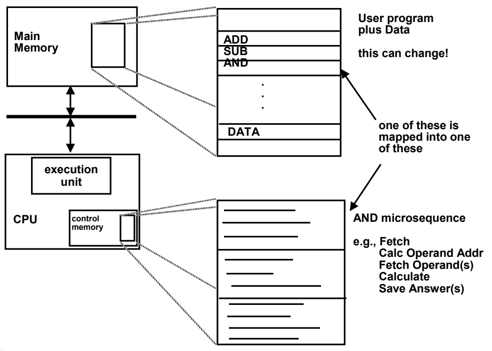
3 Generic Data Hazards: RAW, WAR, WAW

- Write After Write (WAW)
  Instr$_J$ writes operand *before* Instr$_I$ writes it.

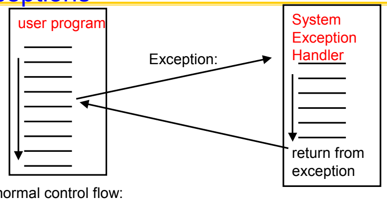
```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers This also results from the "reuse" of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Can see WAR and WAW in more complicated pipes

Recap:"Macroinstruction" Interpretation



Exceptions



normal control flow:
sequential, jumps, branches, calls, returns

- Exception = unprogrammed control transfer
  - system takes action to handle the exception
    - must record the address of the offending instruction
    - record any other information necessary to return afterwards
  - returns control to user
  - must save & restore user state
- Allows constuction of a "user virtual machine"

Two Types of Exceptions: Interrupts and Traps

- Interrupts
  - caused by external events:
    - Network, Keyboard, Disk I/O, Timer
  - asynchronous to program execution
    - Most interrupts can be disabled for brief periods of time
    - Some (like "Power Failing") are non-maskable (NMI)
  - may be handled between instructions
  - simply suspend and resume user program
- Traps
  - caused by internal events
    - exceptional conditions (overflow)
    - errors (parity)
    - faults (non-resident page)
  - synchronous to program execution
  - condition must be remedied by the handler
  - instruction may be retried or simulated and program continued or program may be aborted

Precise Exceptions

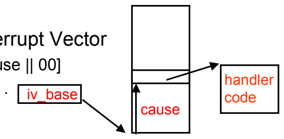
- Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-ot-order execution, ...
  - MIPS takes this position
- Imprecise ⇒ system software has to figure out what is where and put it all back together
- Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Big Picture: user / system modes

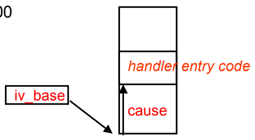
- Two modes of execution (user/system) :
  - operating system runs in privileged mode and has access to all of the resources of the computer
  - presents "virtual resources" to each user that are more convenient that the physical resources
    - files vs. disk sectors
    - virtual memory vs physical memory
  - protects each user program from others
  - protects system from malicious users.
  - OS is assumed to "know best", and is trusted code, so enter system mode on exception
- Exceptions allow the system to taken action in response to events that occur while user program is executing:
  - Might provide supplemental behavior (dealing with denormal floating-point numbers for instance).
  - "Unimplemented instruction" used to emulate instructions that were not included in hardware

Addressing the Exception Handler

- Traditional Approach: Interrupt Vector
  - PC <- MEM[IV_base + cause || 00]
  - 370, 68000, Vax, 80x86, . . .



- RISC Handler Table
  - PC <– IT_base + cause || 0000
  - saves state and jumps
  - Sparc, PA, M88K, . . .
- MIPS Approach: fixed entry
  - PC <– EXC_addr
  - Actually very small table
    - RESET entry
    - TLB
    - other



Saving State

- Push it onto the stack
  - Vax, 68k, 80x86
- Shadow Registers
  - M88k
  - Save state in a shadow of the internal pipeline registers
- Save it in special registers
  - MIPS EPC, BadVaddr, Status, Cause

Additions to MIPS ISA to support Exceptions?

- Exception state is kept in "coprocessor 0".
  - Use mfc0 read contents of these registers
  - Every register is 32 bits, but may be only partially defined

  BadVAddr (register 8)
  - register contained memory address at which memory reference occurred

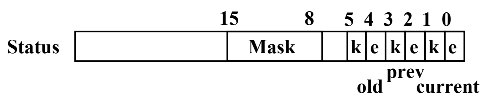
  Status (register 12)
  - interrupt mask and enable bits

  Cause (register 13)
  - the cause of the exception
  - Bits 6 to 2 of this register encodes the exception type (e.g undefined instruction=10 and arithmetic overflow=12)

  EPC (register 14)
  - address of the affected instruction (register 14 of coprocessor 0).

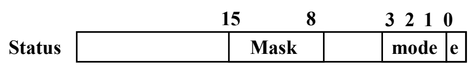
- Control signals to write BadVAddr, Status, Cause, and EPC
- Be able to write exception address into PC (8000 0180$_{hex}$)
- May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor): PC = PC - 4

Details of Status register: MIPS I



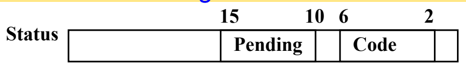
- Mask = 1 bit for each of 5 hardware and 3 software interrupt levels
  - 1 => enables interrupts
  - 0 => disables interrupts
- k = kernel/user
  - 0 => was in the kernel when interrupt occurred
  - 1 => was running user mode
- e = interrupt enable
  - 0 => interrupts were disabled
  - 1 => interrupts were enabled
- When interrupt occurs, 6 LSB shifted left 2 bits, setting 2 LSB to 0
  - run in kernel mode with interrupts disabled

Details of Status register: MIPS 32



- Mask = 1 bit for each of 5 hardware and 3 software interrupt levels
  - 1 => enables interrupts
  - 0 => disables interrupts
- mode = kernel/user
  - 0 => was in the kernel when interrupt occurred
  - 2 => was running user mode
  - (added 1 for "supervisor" state)
- e = interrupt enable
  - 0 => interrupts were disabled
  - 1 => interrupts were enabled

Details of Cause register



- Pending interrupt 5 hardware levels: bit set if interrupt occurs but not yet serviced
  - handles cases when more than one interrupt occurs at same time, or while records interrupt requests when interrupts disabled
- Exception Code encodes reasons for interrupt
  - 0 (INT) => external interrupt
  - 4 (ADDRL) => address error exception (load or instr fetch)
  - 5 (ADDRS) => address error exception (store)
  - 6 (IBUS) => bus error on instruction fetch
  - 7 (DBUS) => bus error on data fetch
  - 8 (Syscall) => Syscall exception
  - 9 (BKPT) => Breakpoint exception
  - 10 (RI) => Reserved Instruction exception
  - 12 (OVF) => Arithmetic overflow exception

Part of the handler in trap_handler.s

```
  .ktext 0x80000080
entry:                             ⇐ Exceptions/interrupts come here
  .set noat
     move  $k1 $at     # Save $at
  .set at
     sw    $v0 s1      # Not re-entrent and we can't trust $sp
     sw    $a0 s2
     mfc0  $k0 $13     # Cause     ⇐ Grab the cause register
     li    $v0 4       # syscall 4 (print_str)
     la    $a0 __m1_
     syscall
     li    $v0 1       # syscall 1 (print_int)
           srl          $a0 $k0 2   # shift Cause reg
     syscall


ret: lw    $v0 s1
     lw    $a0 s2
     mfc0  $k0 $14     # EPC       ⇐ Get the return address (EPC)
  .set noat
     move  $at $k1     # Restore $at
  .set at
     rfe               # Return from exception handler
     addiu             $k0 $k0 4   # Return to next instruction
     jr    $k0
```

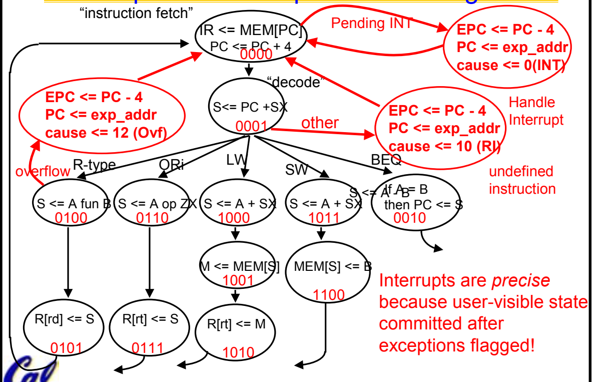
Administrivia

- Lab 4 demo Mon 10/13, write up Tue 10/14
- Reading Ch 5: 5.1 to 5.8, Ch 6: 6.1 to 6.7
- Midterm Wed Oct 8 5:30 - 8:30 in 1 LeConte
  - Midterm review Sunday Oct 4, 5 PM in 306 Soda
  - Bring 1 page, handwritten notes, both sides
  - Meet at LaVal's Northside afterwards for Pizza
  - No lecture Thursday Oct 9
- Office hours
  - Mon 4 – 5:30 Jack, Tue 3:30-5 Kurt, Wed 3 – 4:30 John, Thu 3:30-5 Ben
  - Dave's office hours Tue 3:30 – 5

Example: How Control Handles Traps in our FSD

- Undefined Instruction–detected when no next state is defined from state 1 for the op value.
  - We handle this exception by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, beq, and ori as new state 12.
  - Shown symbolically using "other" to indicate that the op field does not match any of the opcodes that label arcs out of state 1.
- Arithmetic overflow–detected on ALU ops such as signed add
  - Used to save PC and enter exception handler
- External Interrupt – flagged by asserted interrupt line
  - Again, must save PC and enter exception handler
- Note: Challenge in designing control of a real machine is to handle different interactions between instructions and other exception-causing events such that control logic remains small and fast.
  - Complex interactions makes the control unit the most challenging aspect of hardware design

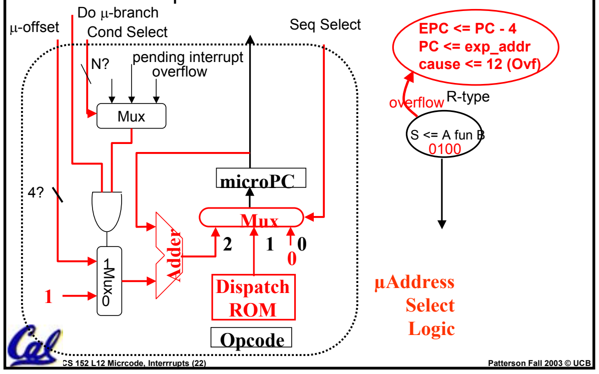
How add traps and interrupts to state diagram?



Interrupts are *precise* because user-visible state committed after exceptions flagged!

But: What has to change in our μ-sequencer?

- Need concept of *branch* at micro-code level



Exception/Interrupts and Pipelining

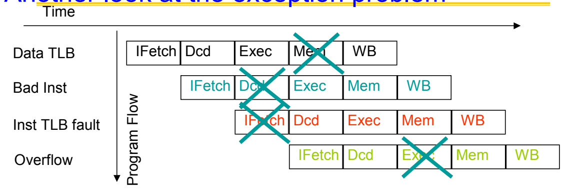
5 instructions, executing in 5 different pipeline stages!

- Who caused the interrupt?

Stage	*Problem interrupts/Exceptions occurring*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined (or illegal) opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation; memory error

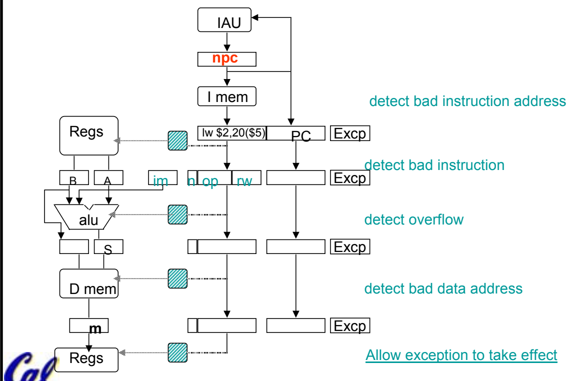
- How do we stop the pipeline? How do we restart it?
- Do we interrupt immediately or wait?
- How do we sort all of this out to maintain preciseness?

Another look at the exception problem

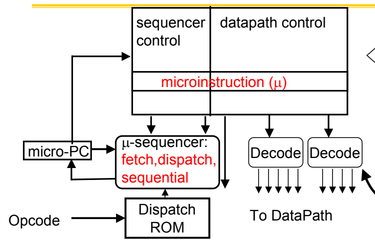


- Use pipeline to sort this out!
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don't act on exception until it reache WB stage
- Handle interrupts through "faulting noop" in IF stage
- When instruction reaches end of MEM stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all (partially-executed) succeeding instructions into noops!

Exception Handling: Add to pipe. reg to record



Recap: Microprogramming



µ-Code ROM

Decoders implement our µ-code language:

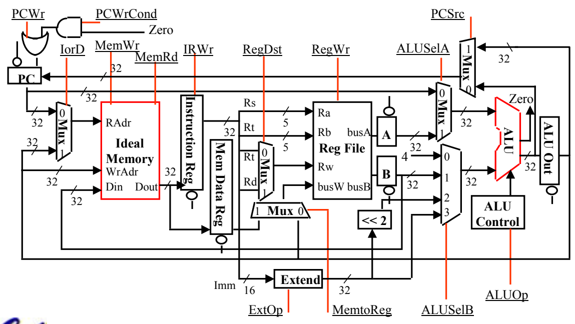
For instance:
- rt-ALU
- rd-ALU
- mem-ALU

- Microprogramming is a fundamental concept
  - implement an instruction set by building a very simple processor and interpreting the instructions
  - essential for very complex instructions and when few register transfers are possible
  - overkill when ISA matches datapath 1-1

Recap: Microprogramming

- Microprogramming is a convenient method for implementing *structured* control state diagrams:
  - Random logic replaced by microPC sequencer and ROM
  - Each line of ROM called a µinstruction:
    contains sequencer control + values for control points
  - limited state transitions:
    branch to zero, next sequential,
    branch to µinstruction address from displatch ROM
- Design of a Microprogramming language
  1. Start with list of control signals
  2. Group signals together that make sense (vs. random): called “fields”
  3. Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
  4. To minimize the width, encode operations that will never be used at the same time
  5. Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals

Recap: Multicycle datapath (book)



Recap: List of control signals

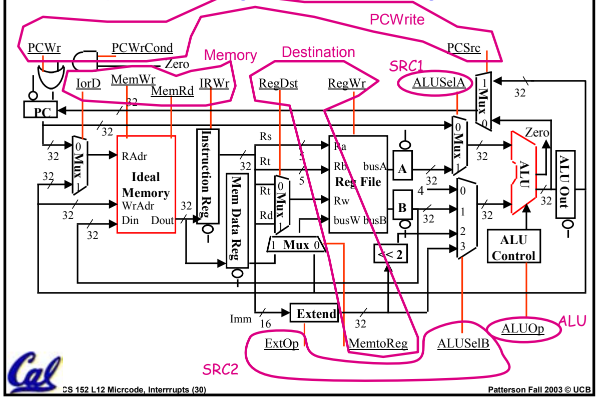
Single Bit Control

Signal name	*Effect when deasserted*	*Effect when asserted*
ALUSelA	1st ALU operand = PC	1st ALU operand = Reg[rs]
RegWrite	None	Reg. is written
MemtoReg	Reg. write data input = ALU	Reg. write data input = memory
RegDst	Reg. dest. no. = rt	Reg. dest. no. = rd
MemRead	None	Memory at address is read, MDR <= Mem[addr]
MemWrite	None	Memory at address is written
IorD	Memory address = PC	Memory address = S
IRWrite	None	IR <= Memory
PCWrite	None	PC <= PCSource
PCWriteCond	None	IF ALUzero then PC <= PCSource
PCSource	PCSource = ALU	PCSource = ALUout
ExtOp	Zero Extended	Sign Extended

Multiple Bit Control

Signal name	***Value***	***Effect***
ALUOp	**00**	**ALU adds**
	01	**ALU subtracts**
	10	**ALU does function code**
	11	**ALU does logical OR**
ALUSelB	**00**	**2nd ALU input = 4**
	01	**2nd ALU input = Reg[rt]**
	10	**2nd ALU input = extended,shift left 2**
	11	**2nd ALU input = extended**

Recap: Group together related signals



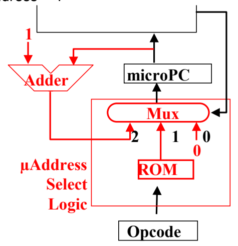
Recap: Specific Sequencer from before

Sequencer-based control unit from last lecture
- Called "microPC" or "µPC" vs. state register

Code	Name	Effect
00	fetch	Next µaddress = 0
01	dispatch	Next µaddress = dispatch ROM
10	seq	Next µaddress = µaddress + 1

ROM:

Opcode: Dispatch state	
000000: Rtype	(0100)
●	
000100: BEQ	(0010)
●	
001101: ORI	(0110)
●	
100011: LW	(1000)
●	
101011: SW	(1011)
●	

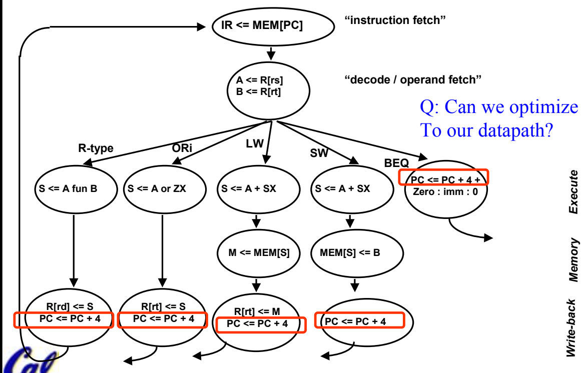


Recap: Group into Fields, Order and Assign Names

ALU	SRC1	SRC2	Dest	Mem	Memreg	PCwrite	Seq

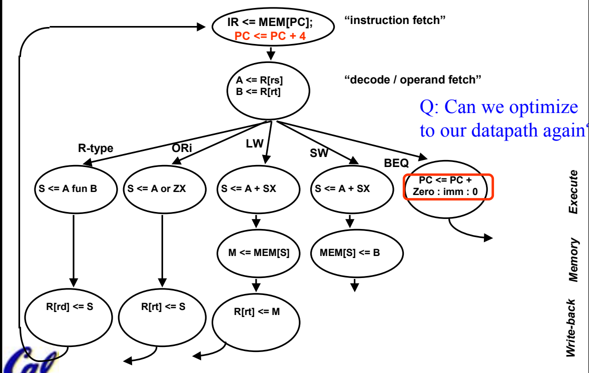
Field Name	*Values for Field*	*Function of Field with Specific Value*
ALU	Add	ALU adds
	Subt.	ALU subtracts
	Func	ALU does function code
	Or	ALU does logical OR
SRC1	PC	1st ALU input <= PC
	rs	1st ALU input <= Reg[rs]
SRC2	4	2nd ALU input <= 4
	Extend	2nd ALU input <= sign ext. IR[15-0]
	Extend0	2nd ALU input <= zero ext. IR[15-0]
	Extshft	2nd ALU input <= sign ex., {IR[15-0],2b00}
	rt	2nd ALU input <= Reg[rt]
Dest(ination)	rd ALU	Reg[rd] <= ALUout
	rt ALU	Reg[rt] <= ALUout
	rt Mem	Reg[rt] <= Mem
Mem(ory)	Read PC	Read memory using PC
	Read ALU	Read memory using ALUout for addr
	Write ALU	Write memory using ALUout for addr
Memreg	IR	IR <= Mem
PCwrite	ALU	PC <= ALU
	ALUoutCond	IF (Zero) PC <= ALUout
Seq(uencing)	Seq	Go to next sequential µinstruction
	Fetch	Go to the first microinstruction
	Dispatch	Dispatch using ROM.

Recap: Multicycle FSM



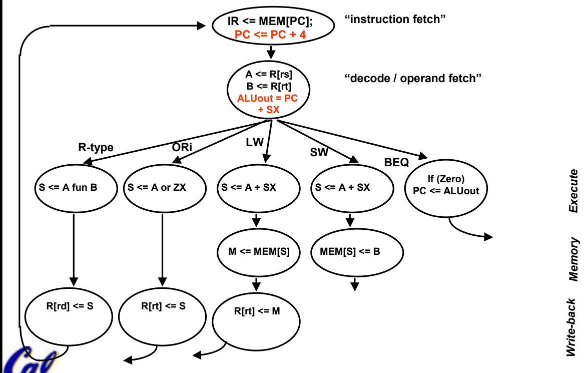
Q: Can we optimize
To our datapath?

Revised Multicycle FSM



Q: Can we optimize
to our datapath again?

Revised Multicycle FSM 2



Recap: 1st Microinstruction (1/10)

Addr	*ALU*	*SRC1*	*SRC2*	*Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequencing*
Fetch:								
0000:	Add	PC	4		Read PC	IR	ALU	Seq
0001:								
BEQ:								
0010:								
Rtype:								
0100:								
0101:								
ORI:								
0110:								
0111:								
LW:								
1000:								
1001:								
1010:								
SW:								
1011:								
1100:								

Microprogram it yourself! (2/10)

Addr	ALU	SRC1	SRC2	Dest.	Memory	Mem. Reg.	PC Write	Sequencing
Fetch:								
0000:	Add	PC	4		Read PC	IR	ALU	Seq
0001:	Q1?							
BEQ:								
0010:	Q2?							
Rtype:								
0100:	Q3?							
0101:	Q4?							
ORI:								
0110:	Q5?							
0111:	Q4?							
LW:								
1000:	Q1?							
1001:	Q4?							
1010:	Q4?							
SW:								
1011:	Q1?							
1100:	Q4?							

1.	Q1:Add	Q2:Subt	Q3:Func	Q4: Or	Q5:blank
2.	Q1:Add	Q2:Subt	Q3:Func	Q4: blank	Q5:Or
3.	Q1:blank	Q2:Subt	Q3:Func	Q4: Add	Q5:Or
4.	Q1:blank	Q2:Add	Q3:Func	Q4: Or	Q5:blank
5.	Q1:Func	Q2:Add	Q3:Func	Q4: blank	Q5:Or
6.	None of the above				

Add	ALU adds
Subt	ALU subtracts
Func	ALU does function code
Or	ALU does logical OR
(blank)	(do nothing)

Legacy Software and Microprogramming

- IBM bet company on 360 Instruction Set Architecture (ISA): single instruction set for many classes of machines
  - (8-bit to 64-bit)
- Stewart Tucker stuck with job of what to do about software compatibility
  - If microprogramming could easily do same instruction set on many different microarchitectures, then why couldn't multiple microprograms do multiple instruction sets on the same microarchitecture?
  - Coined term "emulation": instruction set interpreter in microcode for non-native instruction set
  - Very successful: in early years of IBM 360 it was hard to know whether old instruction set or new instruction set was more frequently used

Microprogramming Pros and Cons

- Ease of design
- Flexibility
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- Can implement very powerful instruction sets (just more control memory)
- Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Costly to implement
- Slow

Thought: Microprogramming one inspiration for RISC

- If simple instruction could execute at very high clock rate…
- If you could even write compilers to produce microinstructions…
- If most programs use simple instructions and addressing modes…
- If microcode is kept in RAM instead of ROM so as to fix bugs …
- If same memory used for control memory could be used instead as cache for "macroinstructions"…
- Then why not skip instruction interpretation by a microprogram and simply compile directly into lowest language of machine? (microprogramming is overkill when ISA matches datapath 1-1)

Summary

- Exceptions, Interrupts handled as unplanned procedure calls
- Control adds arcs to check for exceptions, new states to adjust PC, set CPU status
- OS implements interrupt/exception policy (priority levels) using Interrupt Mask
- For pipelining, interrupts need to be precise (like multicycle)
- Control design can reduces to Microprogramming
- Control is more complicated with:
  - complex instruction sets
  - restricted datapaths (see the book)