CS 152 Computer Architecture and Engineering

Lecture 11: Multicycle Controller Design

October 3, 1997

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

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

Recap

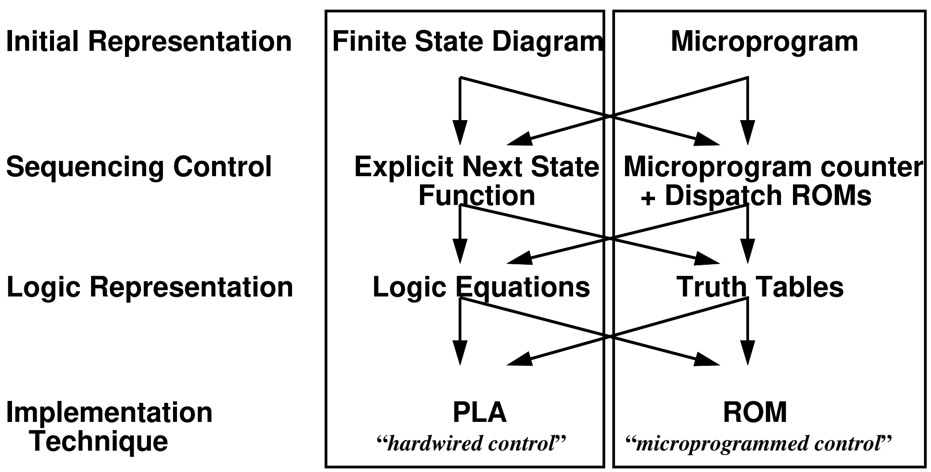
° Partition datapath into equal size chunks to minimize cycle time
  - ~10 levels of logic between latches

° Follow same 5-step method for designing “real” processor

° Control is specified by finite state digram

Overview of Control

° Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.

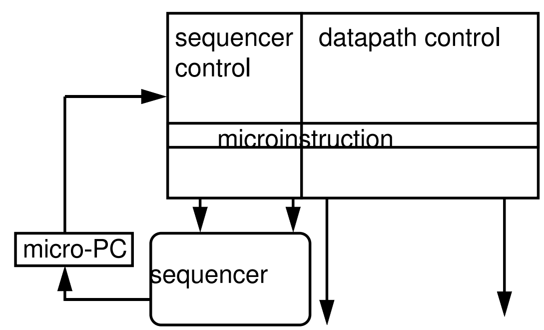


Recap: Controller Design

° The state digrams that arise define the controller for an instruction set processor are highly structured

° Use this structure to construct a simple “microsequencer”

° Control reduces to programming this very simple device
  - microprogramming

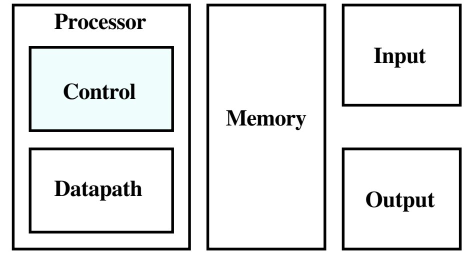


Recap: Microprogram Control Specification

	µPC	Taken	Next	IR	PC en sel	Ops A B	Exec Ex Sr ALU S	Mem R W M	Write-Back M-R Wr Dst
	0000	?	inc	1					
	0001	0	load						
	0001	1	inc						
	0010	x	zero		1 1				
BEQ	0011	x	zero		1 0				
R:	0100	x	inc				0 1 fun 1		
	0101	x	zero		1 0				0 1 1
ORi:	0110	x	inc				0 0 or 1		
	0111	x	zero		1 0				0 1 0
LW:	1000	x	inc				1 0 add 1		
	1001	x	inc					1 0 1	
	1010	x	zero		1 0				1 1 0
SW:	1011	x	inc				1 0 add 1		
	1100	x	zero		1 0			0 1	

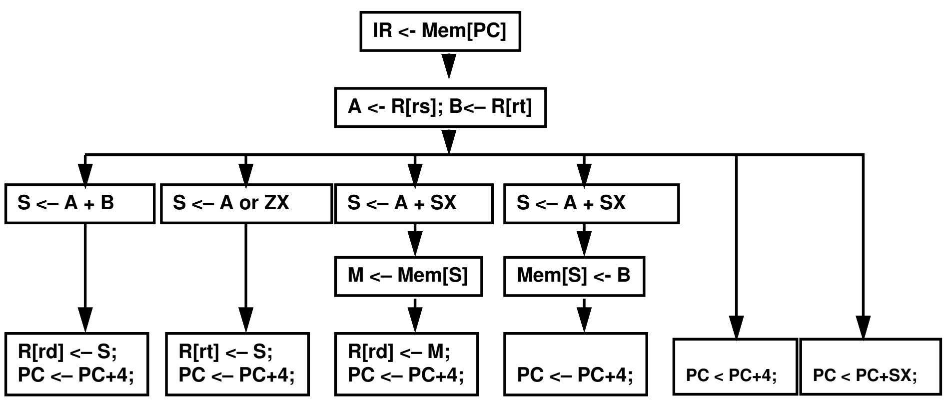
The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



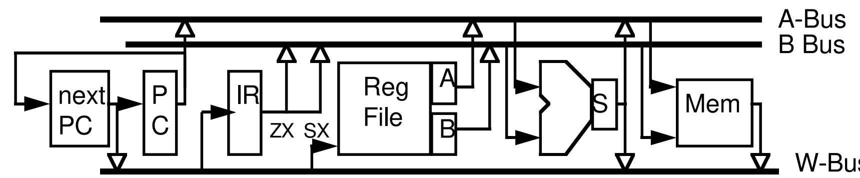
- Today's Topics:
  - Microprogramed control
  - Administrivia
  - Microprogram it yourself
  - Exceptions

How Effectively are we utilizing our hardware?



- Example: memory is used twice, at different times
  - Ave mem access per inst = 1 + Flw + Fsw ~ 1.3
  - if CPI is 4.8, imem utilization = 1/4.8, dmem =0.3/4.8
- We could reduce HW without hurting performance
  - extra control

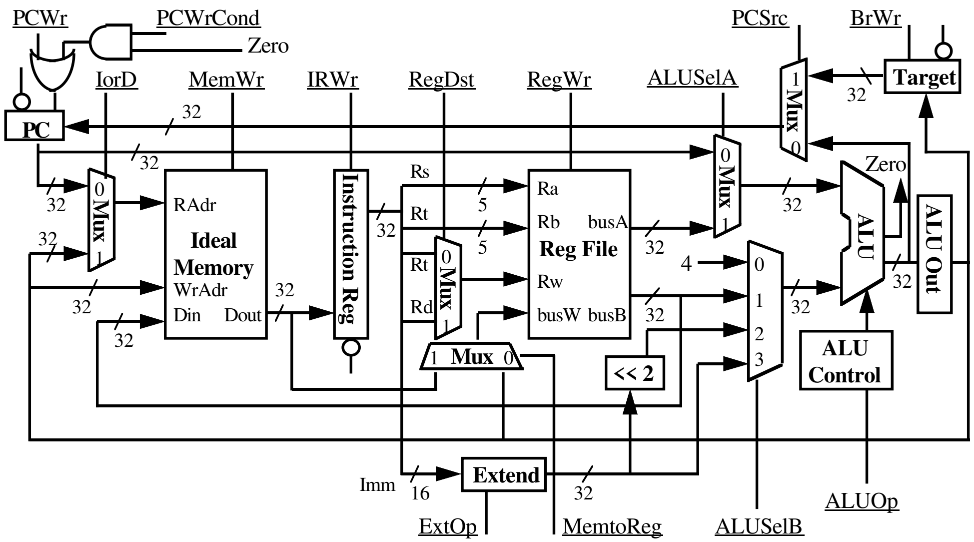
"Princeton" Organization



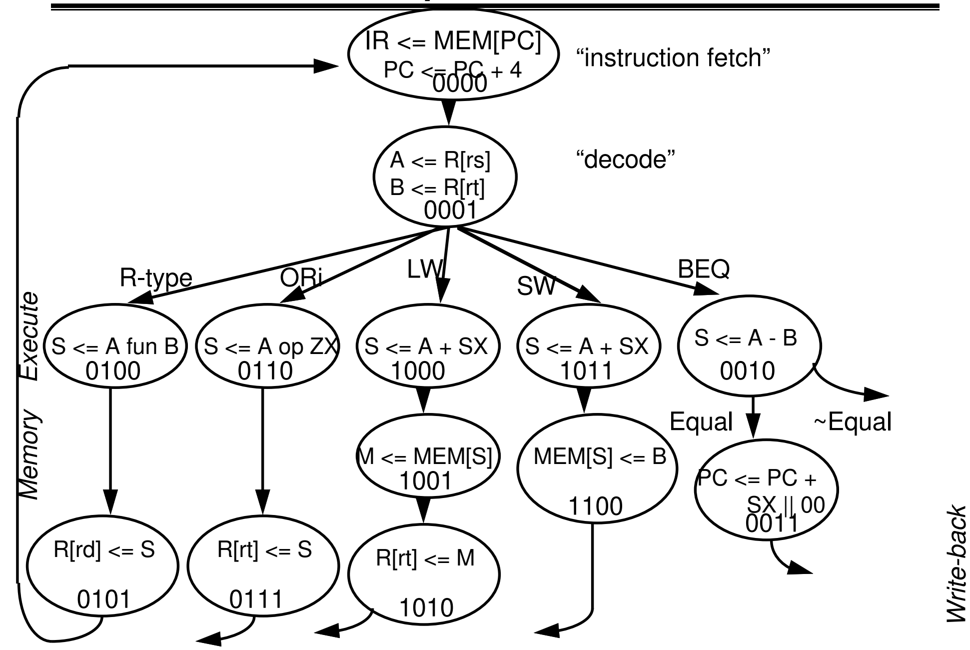
- Single memory for instruction and data access
  - memory utilization -> 1.3/4.8
- In this case our state diagram does not change
  - several additional control signals
  - must ensure each bus is only driven by one source on each cycle

Alternative datapath (book): Multiple Cycle Datapath

° **Miminizes Hardware: 1 memory, 1 adder**



Our Controller FSM Spec



Microprogramming

° **Control is the hard part of processor design**

- ° **Datapath is fairly regular and well-organized**
- ° **Memory is highly regular**
- ° **Control is irregular and global**

Microprogramming:

-- A Particular Strategy for Implementing the Control Unit of a processor by "programming" at the level of register transfer operations

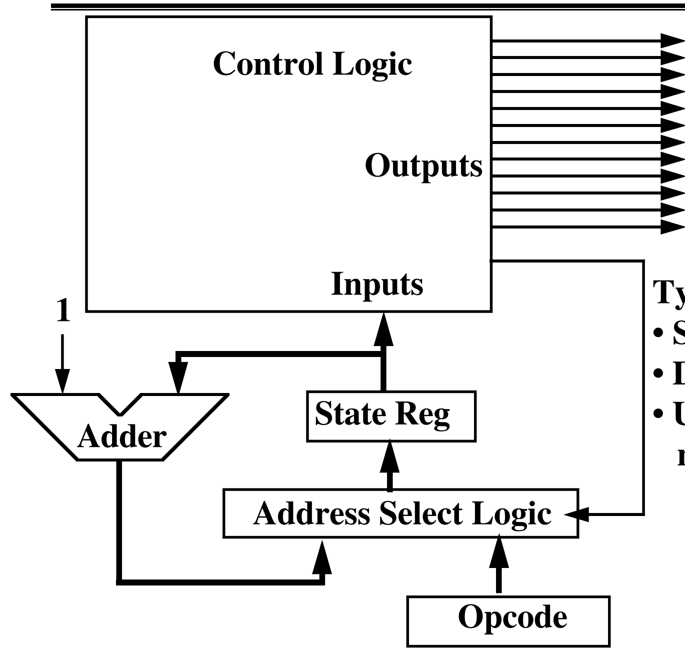
Microarchitecture:

-- Logical structure and functional capabilities of the hardware as seen by the microprogrammer

Historical Note:

IBM 360 Series first to distinguish between architecture & organization
Same instruction set across wide range of implementations, each with different cost/performance

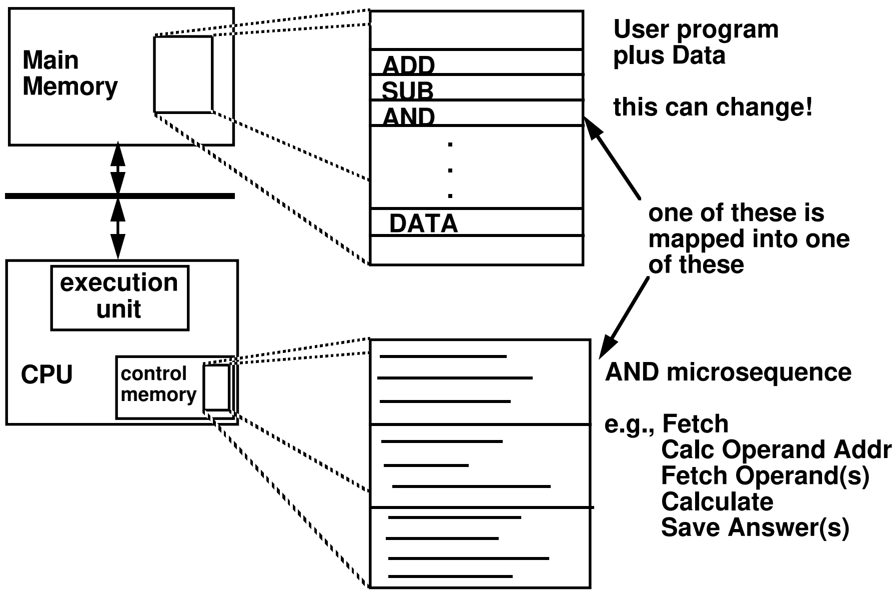
Sequencer-based control unit



Types of "branching"

- **Set state to 0**
- **Dispatch (state 1)**
- **Use incremented state number**

"Macroinstruction" Interpretation



Variations on Microprogramming

° "Horizontal" Microcode

– control field for each control point in the machine

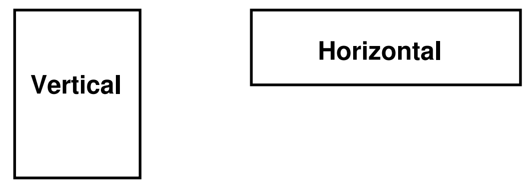
µseq	µaddr	A-mux	B-mux	bus enables	register enables	

° "Vertical" Microcode

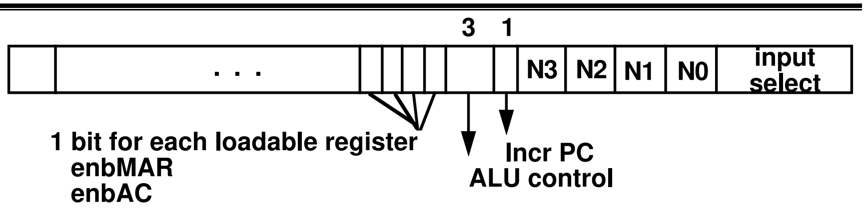
– compact microinstruction format for each class of microoperation

– local decode to generate all control points

branch:	µseq-op	µadd
execute:	ALU-op	A,B,R
memory:	mem-op	S, D



Extreme Horizontal



Depending on bus organization, many potential control combinations simply wrong, i.e., implies transfers that can never happen at the same time.

Makes sense to encode fields to save ROM space

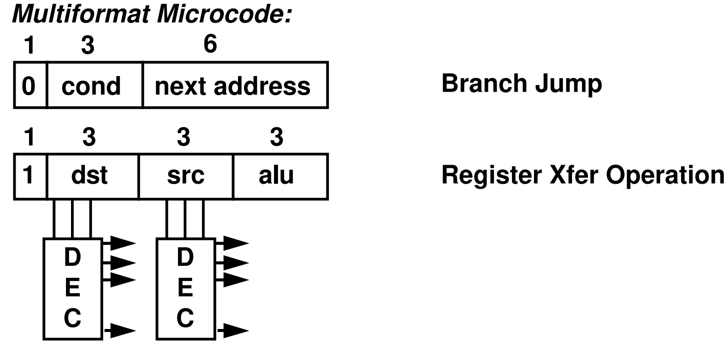
Example: mem_to_reg and ALU_to_reg should never happen simultenously; => encode in single bit which is decoded rather than two separate bits

NOTE: <u>encoding should be just sufficient that parallel actions that the datapath supports should still be specifiable in a single microinstruction</u>

More Vertical Format

Some of these may have nothing to do with registers!

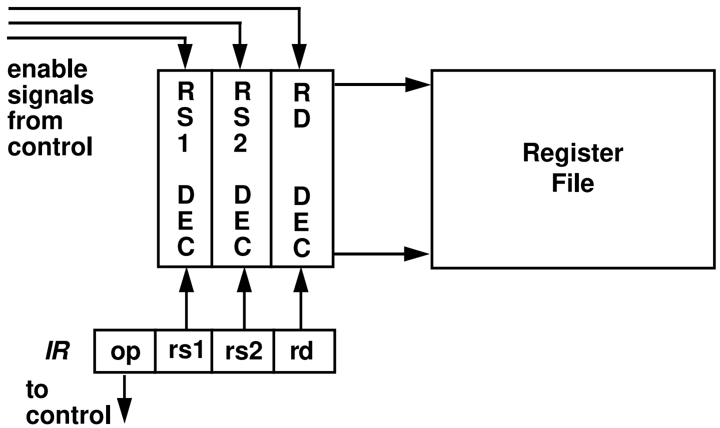
Multiformat Microcode:



Hybrid Control

Not all critical control information is derived from control logic

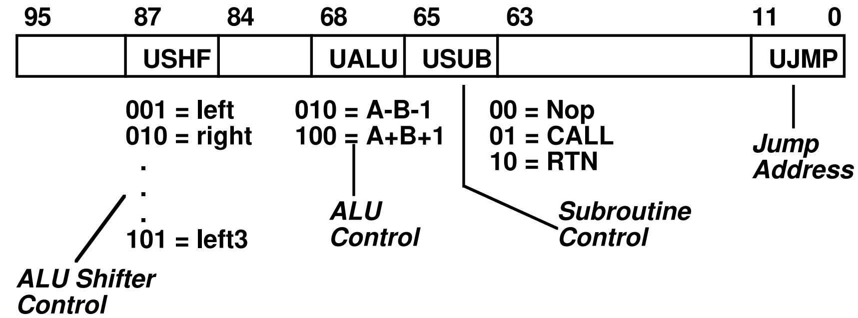
E.g., Instruction Register (IR) contains useful control information, such as register sources, destinations, opcodes, etc.



Vax Microinstructions

VAX Microarchitecture:

96 bit control store, 30 fields, 4096 μinstructions for VAX ISA
encodes concurrently executable "microoperations"



Horizontal vs. Vertical Microprogramming

NOTE: previous organization is not TRUE horizontal microprogramming; register decoders give flavor of *encoded* microoperations

Most microprogramming-based controllers vary between:

horizontal organization (1 control bit per control point)

vertical organization (fields encoded in the control memory and must be decoded to control something)

Horizontal

+ more control over the potential parallelism of operations in the datapath
- uses up lots of control store

Vertical

+ easier to program, not very different from programming a RISC machine in assembly language
- extra level of decoding may slow the machine down

Administration

- Schedule change: Delay Lab 4 until Tuesday after midterm (10/14)?
  - => Delay Lab 5 until 10/28
    => Delay Lab 6 until 11/11
    => Delay Midterm II until 11/18
- Midterm Wed 10/8 at 5:30 in 306 Soda; no lecture that day
- Review Session 1-3 Sunday 10/5 in 306 Soda

Designing a Microinstruction Set

1) Start with list of control signals

2) Group signals together that make sense (vs. random): called "fields"

3) Places fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)

4) Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals

- Use computers to design computers

5) To minimize the width, encode operations that will never be used at the same time

1&2) Start with list of control signals, grouped into fields

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
TargetWrite	None	Target reg. = ALU
MemRead	None	Memory at address is read
MemWrite	None	Memory at address is written
IorD	Memory address = PC	Memory address = ALU
IRWrite	None	IR = Memory
PCWrite	None	PC = PCSource
PCWriteCond	None	IF ALUzero then PC = PCSource

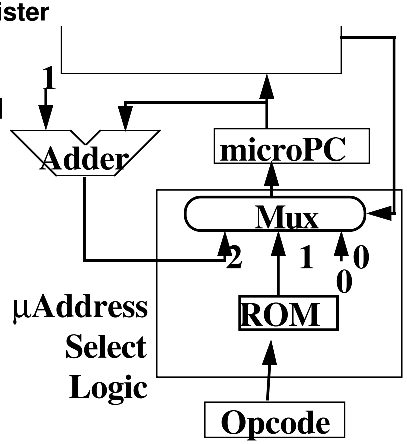
Multiple Bit Control

Signal name	***Value***	***Effect***
ALUOp	00	ALU adds
	01	ALU subtracts
	10	ALU does function code
	11	ALU does logical OR
ALUSelB	000	2nd ALU input = Reg[rt]
	001	2nd ALU input = 4
	010	2nd ALU input = sign extended IR[15-0]
	011	2nd ALU input = sign extended, shift left 2 IR[15-0]
	100	2nd ALU input = zero extended IR[15-0]
PCSource	00	PC = ALU
	01	PC = Target
	10	PC = PC+4[29-26] : IR[25–0] << 2

Start with list of control signals, cont'd

- For next state function (next microinstruction address), use Sequencer-based control unit from last lecture
  - Called "microPC" or "µPC" vs. state register

Signal	***Value***	***Effect***
Sequen-cing	00	Next µaddress = 0
	01	Next µaddress = dispatch ROM
	10	Next µaddress = µaddress + 1



3) Microinstruction Format: unencoded vs. encoded fields

Field Name	***Width*** wide	narrow	***Control Signals Set***
ALU Control	4	2	ALUOp
SRC1	2	1	ALUSelA
SRC2	5	3	ALUSelB
ALU Destination	6	4	RegWrite, MemtoReg, RegDst, TargetWr.
Memory	4	3	MemRead, MemWrite, IorD
Memory Register	1	1	IRWrite
PCWrite Control	5	4	PCWrite, PCWriteCond, PCSource
Sequencing	3	2	AddrCtl
Total width	30	20	bits

4) Legend of Fields and Symbolic Names

Field Name	*Values for Field*	*Function of Field with Specific Value*
ALU	Add	ALU adds
	Subt.	ALU subtracts
	Func code	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., sl IR[15-0]
	rt	2nd ALU input = Reg[rt]
ALU destination	Target	Target = ALUout
	rd	Reg[rd] = ALUout
Memory	Read PC	Read memory using PC
	Read ALU	Read memory using ALU output
	Write ALU	Write memory using ALU output
Memory register	IR	IR = Mem
	Write rt	Reg[rt] = Mem
	Read rt	Mem = Reg[rt]
PC write	ALU	PC = ALU output
	Target-cond.	IF ALU Zero then PC = Target
	jump addr.	PC = PCSource
Sequencing	Seq	Go to sequential μinstruction
	Fetch	Go to the first microinstruction
	Dispatch	Dispatch using ROM.

Microprogram it yourself!

Label	*ALU*	*SRC1*	*SRC2*	*ALU Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequencing*
Fetch	Add	PC	4		Read PC	IR	ALU	Seq

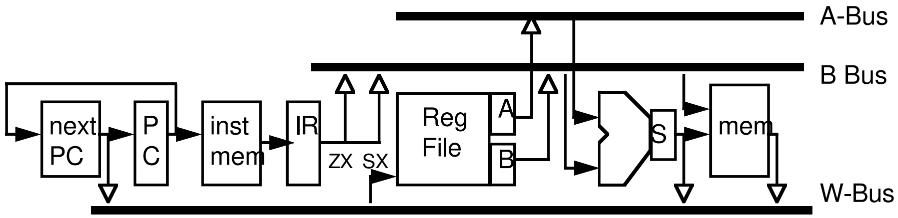
Legacy Software and Microprogramming

- 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 compatability
- 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

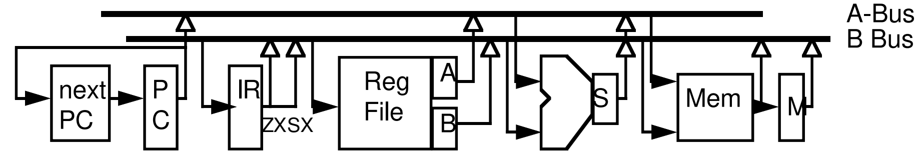
An Alternative MultiCycle DataPath



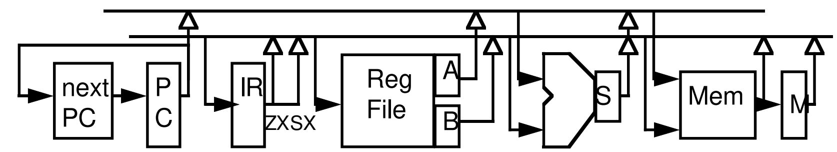
- In each clock cycle, each Bus can be used to transfer from one source
- μ-instruction can simply contain B-Bus and W-Dst fields

What about a 2-Bus Microarchitecture (datapath)?

Instruction Fetch

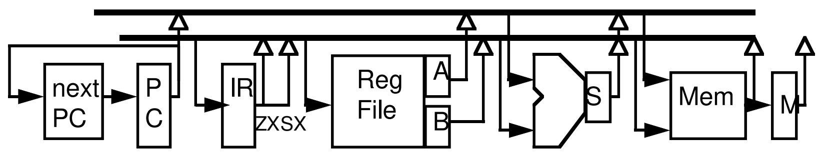


Decode / Operand Fetch

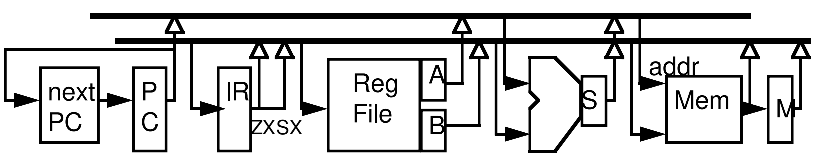


Load

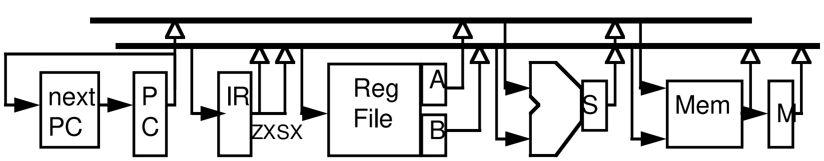
Execute



Mem

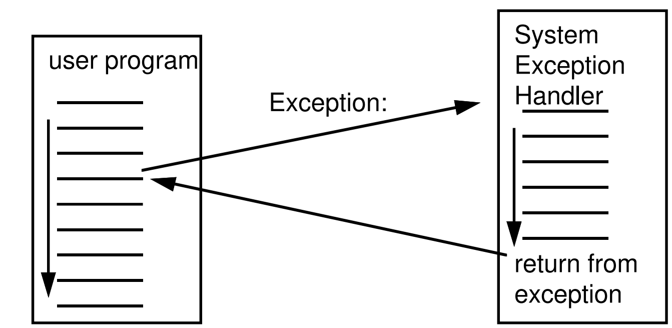


Write-back



- What about 1 bus ? 1 adder? 1 Register port?

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
  - returns control to user
  - must save & restore user state
- **Allows constuction of a “user virtual machine”**

What happens to Instruction with Exception?

- MIPS architecture defines the instruction as having <u>no effect</u> if the instruction causes an exception.
- When get to virtual memory we will see that certain classes of exceptions must prevent the instruction from changing the machine state.
- This aspect of handling exceptions becomes complex and potentially limits performance => why it is hard

Two Types of Exceptions

- Interrupts
  - caused by external events
  - asynchronous to program execution
  - 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

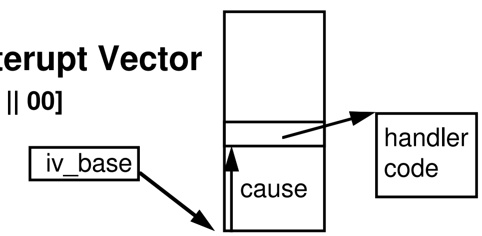
MIPS convention:

- exception means any unexpected change in control flow, without distinguishing internal or external; use the term interrupt only when the event is externally caused.

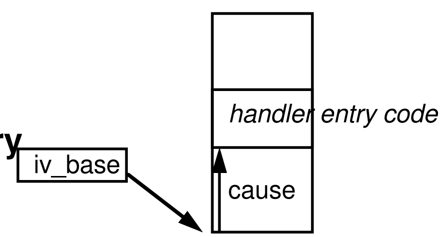
Type of event	*From where?*	*MIPS terminology*
I/O device request	External	Interrupt
Invoke OS from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or Interrupt

Addressing the Exception Handler

- Traditional Approach: Interupt 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



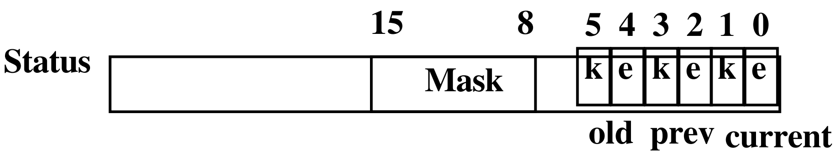
Saving State

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

Additions to MIPS ISA to support Exceptions?

- EPC–a 32-bit register used to hold the address of the affected instruction (register 14 of coprocessor 0).
- Cause–a register used to record the cause of the exception. In the MIPS architecture this register is 32 bits, though some bits are currently unused. Assume that bits 5 to 2 of this register encodes the two possible exception sources mentioned above: undefined instruction=0 and arithmetic overflow=1 (register 13 of coprocessor 0).
- BadVAddr - register contained memory address at which memory reference occurred (register 8 of coprocessor 0)
- Status - interrupt mask and enable bits (register 12 of coprocessor 0)
- Control signals to write EPC , Cause, BadVAddr, and Status
- Be able to write exception address into PC, increase mux to add as input 01000000 00000000 00000000 01000000_{two} (8000 0080_{hex})
- May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor); PC = PC - 4

Recap: Details of Status register

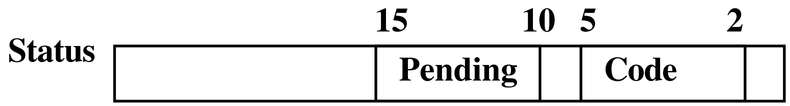


- Mask = 1 bit for each of 5 hardware and 3 software interrupt levels
  - 1 => enables interrupts
  - 0 => disables interrupts
- k = kernel/user
  - 0 => was in the kernel when interrupt occurred
  - 1 => was running user mode
- e = interrupt enable
  - 0 => interrupts were disabled
  - 1 => interrupts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

Big Picture: user / system modes

- By providing two modes of execution (user/system) it is possible for the computer to manage itself
  - operating system is a special program that runs in the priviledged 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 resurces
    - files vs. disk sectors
    - virtual memory vs physical memory
  - protects each user program from others
- Exceptions allow the system to taken action in response to events that occur while user program is executing
  - O/S begins at the handler

Recap: 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

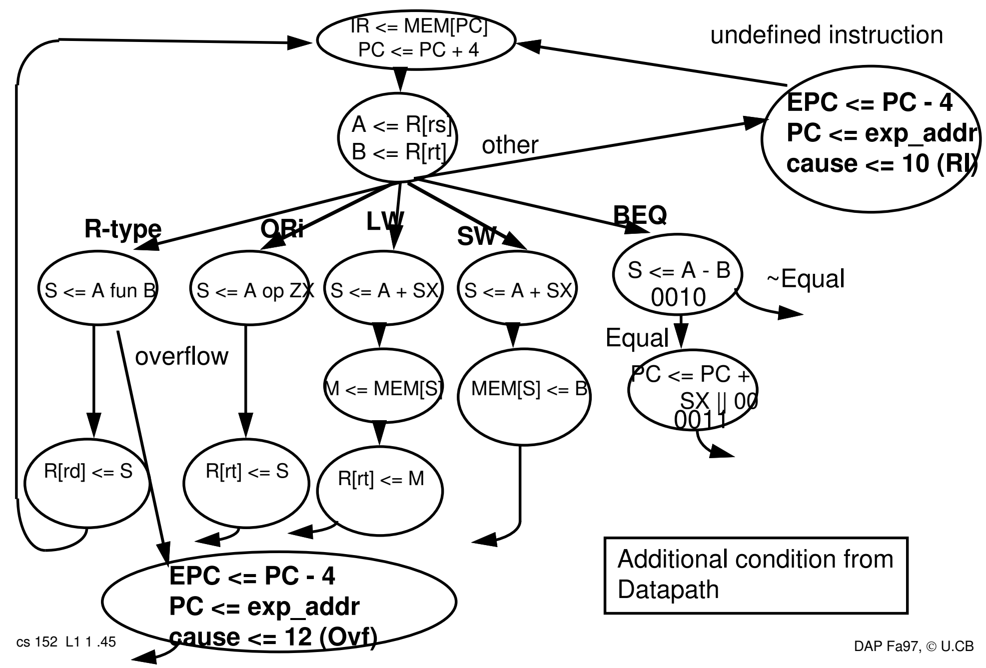
Precise Interrupts

- Precise => state of the machine is preserved as if program executed upto the offending instruction
  - Same system code will work on different implementations of the architecture
  - Position clearly established by IBM
  - 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

How Control Detects Exceptions 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–Chapter 4 included logic in the ALU to detect overflow, and a signal called Overflow is provided as an output from the ALU. This signal is used in the modified finite state machine to specify an additional possible next state
- 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

Modification to the Control Specification



Summary

- Specialize state-diagrams easily captured by microsequencer
  - simple increment & “branch” fields
  - datapath control fields
- Control design reduces to Microprogramming
- Exceptions are the hard part of control
- Need to find convenient place to detect exceptions and to branch to state or microinstruction that saves PC and invokes the operating system
- As we get pipelined CPUs that support page faults on memory accesses which means that the instruction cannot complete AND you must be able to restart the program at exactly the instruction with the exception, it gets even harder

Summary: 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?