CS 152 Computer Architecture and Engineering

Lecture 2 - Simple Machine Implementations

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

Last Time in Lecture 1

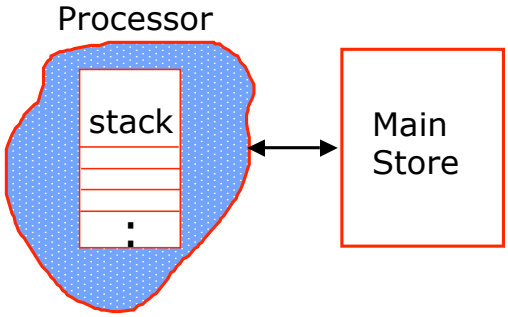
- Computer Science at crossroads from sequential to parallel computing
- Computer Architecture >> ISAs and RTL
  - CS152 is about interaction of hardware and software, and design of appropriate abstraction layers
- Comp. Arch. shaped by technology and applications
  - History provides lessons for the future
- Cost of software development a large constraint on architecture
  - Compatibility a key solution to software cost
- IBM 360 introduces notion of “family of machines” running same ISA but very different implementations
  - Within same generation of machines
  - “Future-proofing” for subsequent generations of machine

Burrough's B5000 Stack Architecture:

An ALGOL Machine, Robert Barton, 1960

- Machine implementation can be completely hidden if the programmer is provided only a high-level language interface.
- *Stack machine* organization because stacks are convenient for:
  1. expression evaluation;
  2. subroutine calls, recursion, nested interrupts;
  3. accessing variables in block-structured languages.
- B6700, a later model, had many more innovative features
  - tagged data
  - virtual memory
  - multiple processors and memories

A Stack Machine

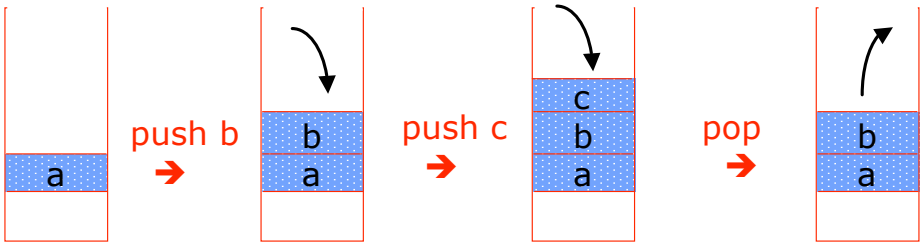


A Stack machine has a stack as a part of the processor state

typical operations:

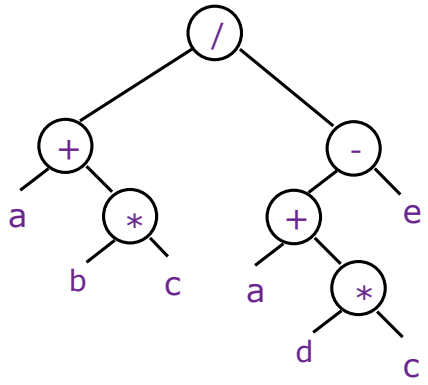
push, pop, +, *, ...

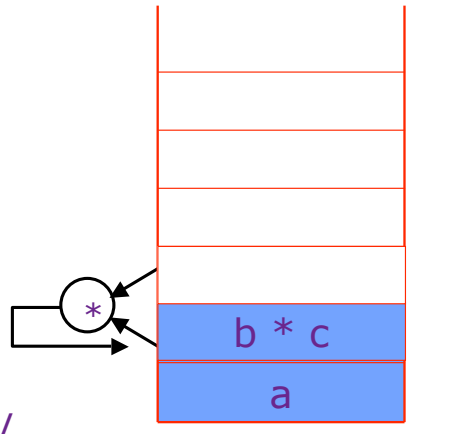
Instructions like + implicitly specify the top 2 elements of the stack as operands.



Evaluation of Expressions

(a + b * c) / (a + d * c - e)





Evaluation Stack

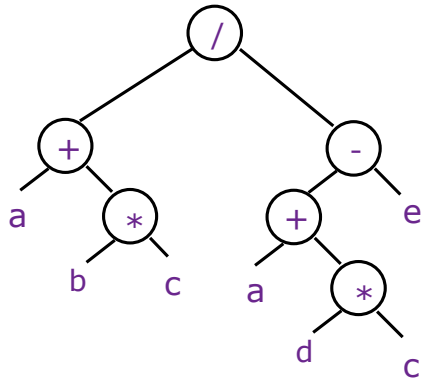
Reverse Polish

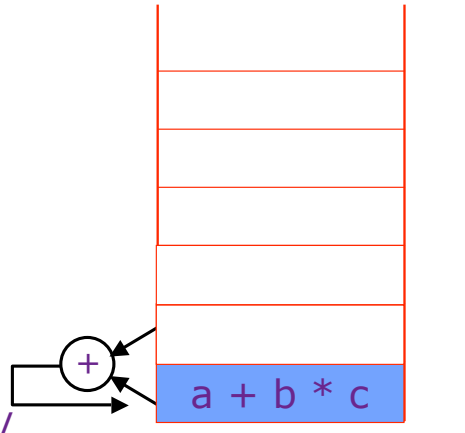
a b c * + a d c * + e - /

push push push multiply

Evaluation of Expressions

(a + b * c) / (a + d * c - e)





Evaluation Stack

Reverse Polish

a b c * + a d c * + e - /

add

Hardware organization of the stack

- Stack is part of the processor state
  ⇒ *stack must be bounded and small*
  ≈ number of Registers,
  not the size of main memory

- Conceptually stack is unbounded
  ⇒ *a part of the stack is included in the processor state; the rest is kept in the main memory*

Stack Operations and Implicit Memory References

- Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

Each *push* operation	⇒	1 memory reference
pop operation	⇒	1 memory reference

No Good!

- Better performance can be got if the top N elements are kept in registers and memory references are made only when register stack overflows or underflows.

Issue - when to Load/Unload registers ?

Stack Size and Memory References

a b c * + a d c * + e - /

program	*stack (size = 2)*	*memory refs*
push a	R0	a
push b	R0 R1	b
push c	R0 R1 R2	c, ss(a)
*	R0 R1	sf(a)
+	R0	
push a	R0 R1	a
push d	R0 R1 R2	d, ss(a+b*c)
push c	R0 R1 R2 R3	c, ss(a)
*	R0 R1 R2	sf(a)
+	R0 R1	sf(a+b*c)
push e	R0 R1 R2	e,ss(a+b*c)
-	R0 R1	sf(a+b*c)
/	R0	

4 stores, 4 fetches (implicit)

Stack Size and Expression Evaluation

a b c * + a d c * + e - /

a and c are "loaded" twice

⇒

not the best use of registers!

program	*stack (size = 4)*
push a	R0
push b	R0 R1
push c	R0 R1 R2
*	R0 R1
+	R0
push a	R0 R1
push d	R0 R1 R2
push c	R0 R1 R2 R3
*	R0 R1 R2
+	R0 R1
push e	R0 R1 R2
-	R0 R1
/	R0

Register Usage in a GPR Machine

(a + b * c) / (a + d * c - e)

	Load	R0	a
	Load	R1	c
	Load	R2	b
Reuse R2	Mul	R2	R1
	Add	R2	R0
	Load	R3	d
Reuse R3	Mul	R3	R1
	Add	R3	R0
Reuse R0	Load	R0	e
	Sub	R3	R0
	Div	R2	R3

More control over register usage since registers can be named explicitly

Load	Ri m
Load	Ri (Rj)
Load	Ri (Rj) (Rk)

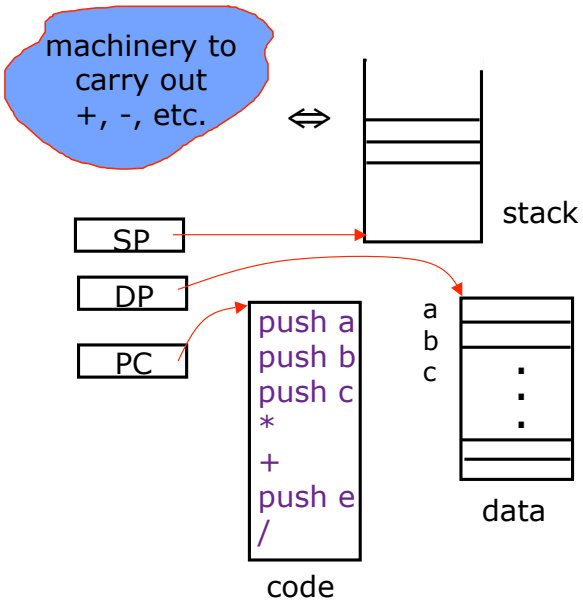
⇒

- *eliminates unnecessary Loads and Stores*
- *fewer Registers*

but instructions may be longer!

Stack Machines: Essential features

- In addition to push, pop, + etc., the instruction set must provide the capability to
  - *refer to any element in the data area*
  - *jump to any instruction in the code area*
  - *move any element in the stack frame to the top*



Stack versus GPR Organization

Amdahl, Blaauw and Brooks, 1964

1. The performance advantage of push down stack organization is derived from the presence of fast registers and not the way they are used.
2. "Surfacing" of data in stack which are "profitable" is approximately 50% because of constants and common subexpressions.
3. Advantage of instruction density because of implicit addresses is equaled if short addresses to specify registers are allowed.
4. Management of finite depth stack causes complexity.
5. Recursive subroutine advantage can be realized only with the help of an independent stack for addressing.
6. Fitting variable-length fields into fixed-width word is awkward.

Stack Machines (Mostly) Died by 1980

1. Stack programs are not smaller if short (Register) addresses are permitted.

2. Modern compilers can manage fast register space better than the stack discipline.

GPR's and caches are better than stack and displays

Early language-directed architectures often did not take into account the role of compilers!

B5000, B6700, HP 3000, ICL 2900, Symbolics 3600

Some would claim that an echo of this mistake is visible in the SPARC architecture register windows - more later…

Stacks post-1980

- Inmos Transputers (1985-2000)
  - Designed to support many parallel processes in Occam language
  - Fixed-height stack design simplified implementation
  - Stack trashed on context swap (fast context switches)
  - Inmos T800 was world's fastest microprocessor in late 80's
- Forth machines
  - Direct support for Forth execution in small embedded real-time environments
  - Several manufacturers (Rockwell, Patriot Scientific)
- Java Virtual Machine
  - Designed for software emulation, not direct hardware execution
  - Sun PicoJava implementation + others
- Intel x87 floating-point unit
  - Severely broken stack model for FP arithmetic
  - Deprecated in Pentium-4, replaced with SSE2 FP registers

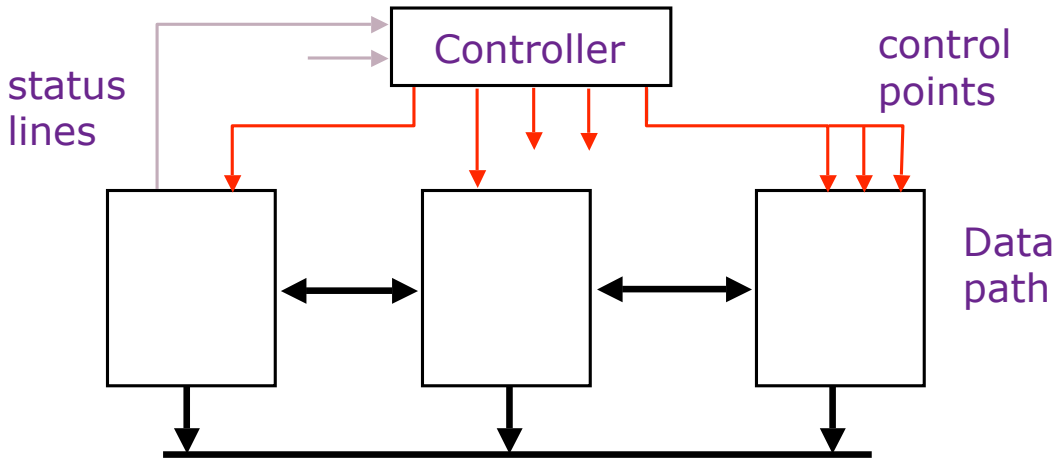
Microprogramming

- A brief look at microprogrammed machines
  - To show how to build very small processors with complex ISAs
  - To help you understand where CISC machines came from
  - Because it is still used in the most common machines (x86, PowerPC, IBM360)
  - As a gentle introduction into machine structures
  - To help understand how technology drove the move to RISC

ISA to Microarchitecture Mapping

- An ISA often designed for a particular microarchitectural style, e.g.,
  - CISC ⇒ microcoded
  - RISC ⇒ hardwired, pipelined
  - VLIW ⇒ fixed latency in-order pipelines
  - JVM ⇒ software interpretation
- But an ISA can be implemented in any microarchitectural style
  - Core 2 Duo: hardwired pipelined CISC (x86) machine (with some microcode support)
  - This lecture: a microcoded RISC (MIPS) machine
  - Intel might eventually have a dynamically scheduled out-of-order VLIW (IA-64) processor
  - PicoJava: A hardware JVM processor

Microarchitecture: *Implementation of an ISA*

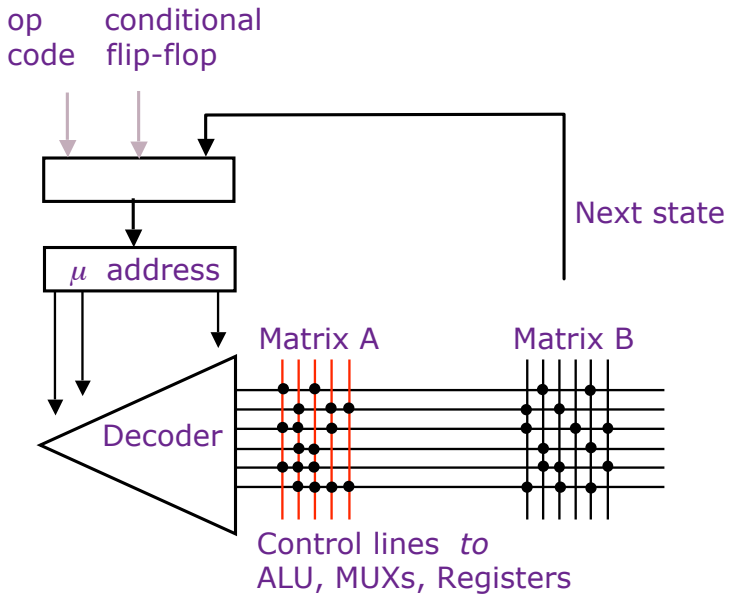


Structure: How components are connected.
Static

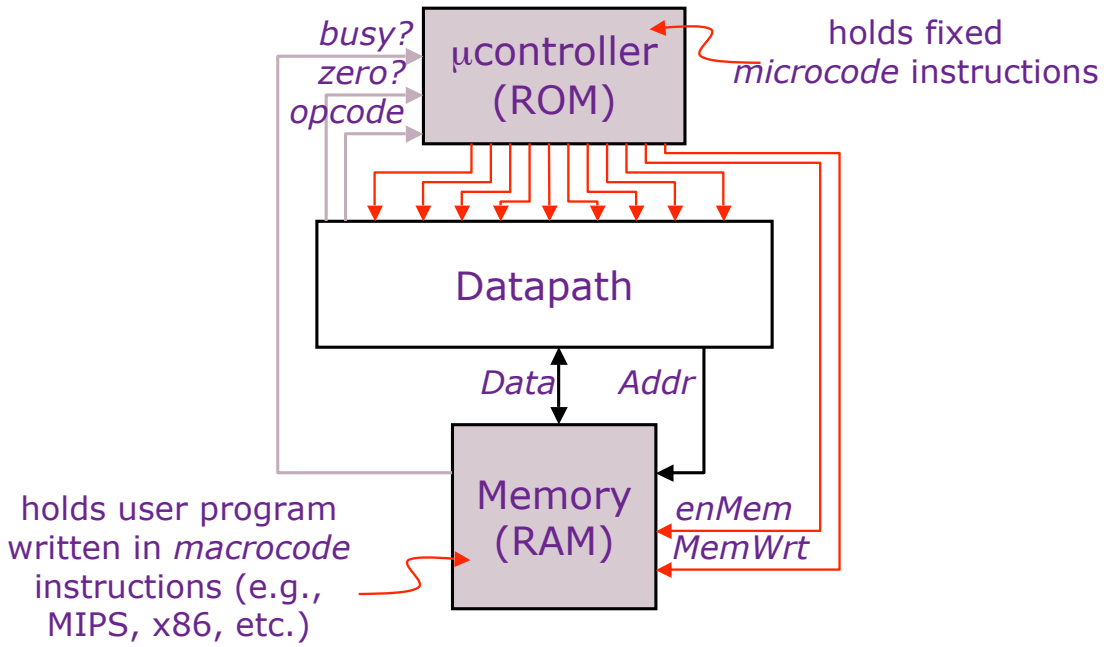
Behavior: How data moves between components
Dynamic

Microcontrol Unit *Maurice Wilkes, 1954*

Embed the control logic state table in a memory array



Microcoded Microarchitecture



The MIPS32 ISA

- Processor State
  - 32 32-bit GPRs, R0 always contains a 0
  - 16 double-precision/32 single-precision FPRs
  - FP status register, used for FP compares & exceptions
  - PC, the program counter
  - some other special registers

See H&P Appendix B for full description

- Data types
  - 8-bit byte, 16-bit half word
  - 32-bit word for integers
  - 32-bit word for single precision floating point
  - 64-bit word for double precision floating point

- Load/Store style instruction set
  - data addressing modes- immediate & indexed
  - branch addressing modes- PC relative & register indirect
  - Byte addressable memory- big-endian mode

All instructions are 32 bits

MIPS Instruction Formats

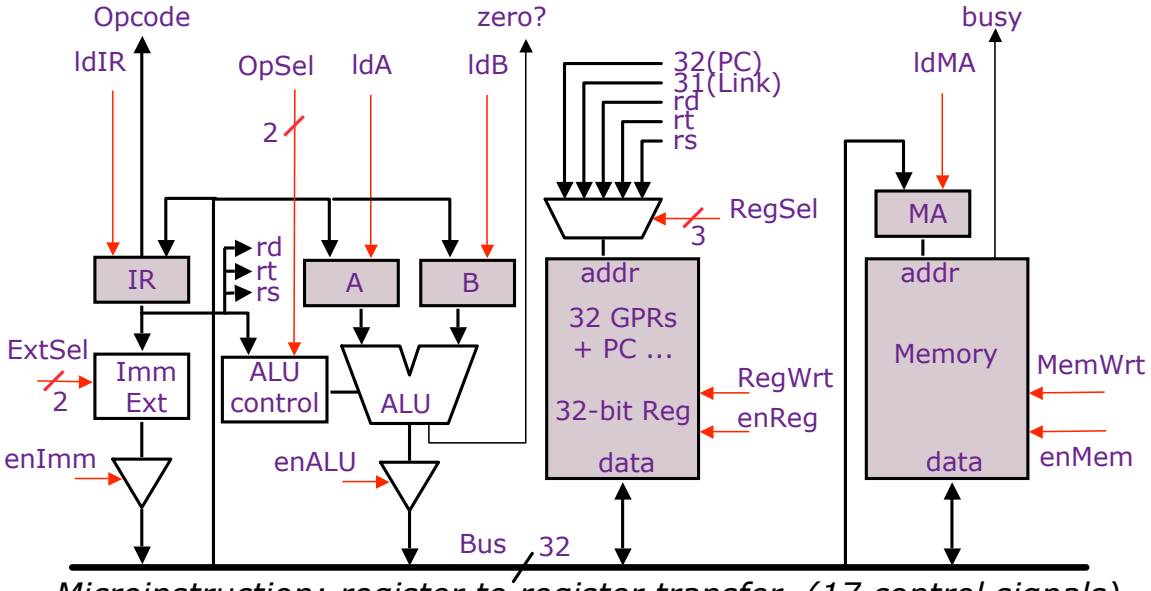
	6	5	5	5	5	6	
ALU	0	rs	rt	rd	0	func	rd ← (rs) func (rt)
ALUi	opcode	rs	rt	immediate			rt ← (rs) op immediate

	6	5	5	16	
Mem	opcode	rs	rt	displacement	M[(rs) + displacement]

6	5	5	16	
opcode	rs		offset	BEQZ, BNEZ
opcode	rs			JR, JALR

6	26	
opcode	offset	J, JAL

A Bus-based Datapath for MIPS

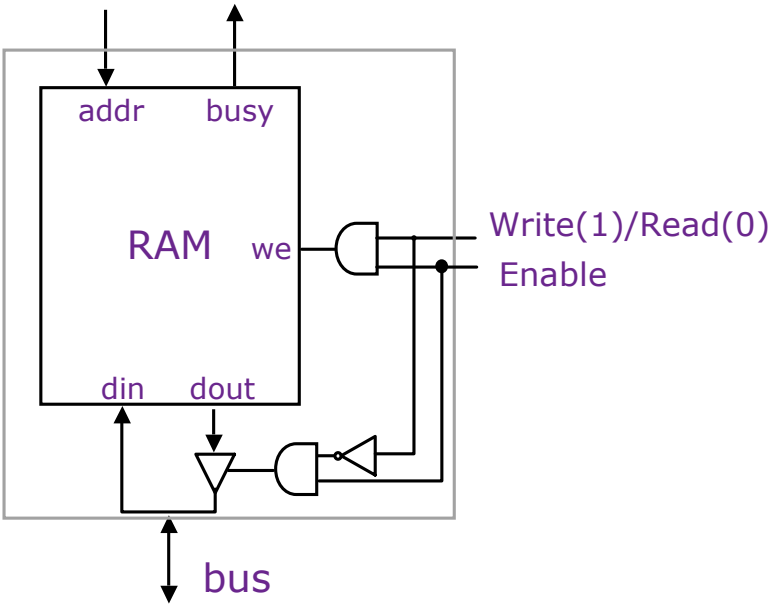


Microinstruction: register to register transfer (17 control signals)

MA ← PC *means* RegSel = PC; enReg=yes; ldMA= yes

B ← Reg[rt] *means* RegSel = rt; enReg=yes; ldB = yes

Memory Module



Assumption: Memory operates independently and is slow as compared to Reg-to-Reg transfers (multiple CPU clock cycles per access)

Instruction Execution

Execution of a MIPS instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back to register file (optional)

   + the computation of the *next instruction* address

Microprogram Fragments

instr fetch:

MA ← PC
A ← PC
IR ← Memory
PC ← A + 4
dispatch on OPcode

can be treated as a macro

ALU:

A ← Reg[rs]
B ← Reg[rt]
Reg[rd] ← func(A,B)
do instruction fetch

ALUi:

A ← Reg[rs]
B ← Imm *sign extension ...*
Reg[rt] ← Opcode(A,B)
do instruction fetch

Microprogram Fragments *(cont.)*

LW:
A ← Reg[rs]
B ← Imm
MA ← A + B
Reg[rt] ← Memory
do instruction fetch

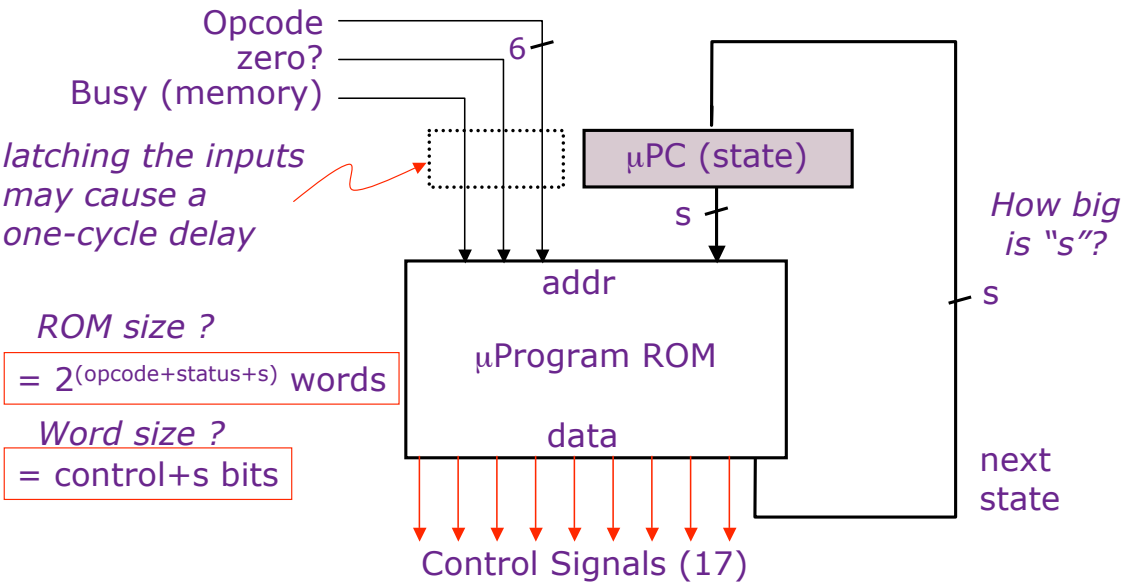
J:
A ← PC
B ← IR
PC ← JumpTarg(A,B)
do instruction fetch

JumpTarg(A,B) = {A[31:28],B[25:0],00}

beqz:
A ← Reg[rs]
If zero?(A) *then go to* bz-taken
do instruction fetch

bz-taken:
A ← PC
B ← Imm << 2
PC ← A + B
do instruction fetch

MIPS Microcontroller: *first attempt*



Microprogram in the ROM *worksheet*

State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	*	*	*	PC ← A + 4	?
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
ALU_0	*	*	*	A ← Reg[rs]	ALU_1
ALU_1	*	*	*	B ← Reg[rt]	ALU_2
ALU_2	*	*	*	Reg[rd] ← func(A,B)	$fetch_0$

Microprogram in the ROM

State	Op	zero?	busy	Control points	next-state
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
$fetch_3$	ALUi	*	*	PC ← A + 4	$ALUi_0$
$fetch_3$	LW	*	*	PC ← A + 4	LW_0
$fetch_3$	SW	*	*	PC ← A + 4	SW_0
$fetch_3$	J	*	*	PC ← A + 4	J_0
$fetch_3$	JAL	*	*	PC ← A + 4	JAL_0
$fetch_3$	JR	*	*	PC ← A + 4	JR_0
$fetch_3$	JALR	*	*	PC ← A + 4	$JALR_0$
$fetch_3$	beqz	*	*	PC ← A + 4	$beqz_0$
...					
ALU_0	*	*	*	A ← Reg[rs]	ALU_1
ALU_1	*	*	*	B ← Reg[rt]	ALU_2
ALU_2	*	*	*	Reg[rd] ← func(A,B)	$fetch_0$

Microprogram in the ROM *Cont.*

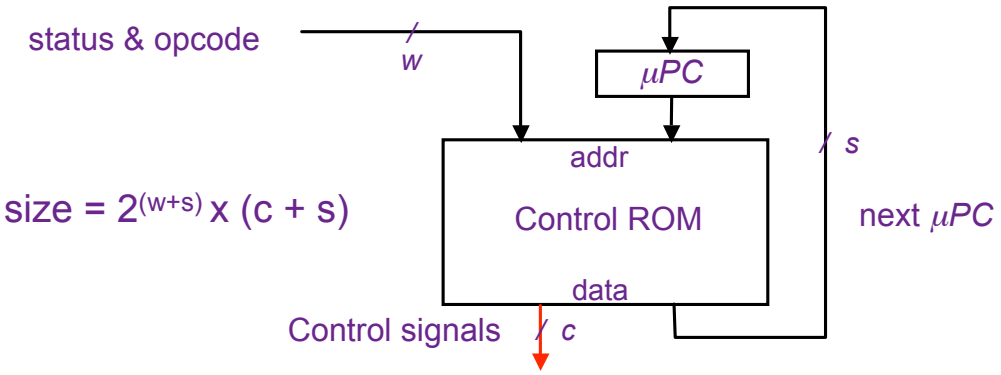
State	Op	zero?	busy	Control points	next-state
$ALUi_0$	*	*	*	A ← Reg[rs]	$ALUi_1$
$ALUi_1$	sExt	*	*	B ← $sExt_{16}$(Imm)	$ALUi_2$
$ALUi_1$	uExt	*	*	B ← $uExt_{16}$(Imm)	$ALUi_2$
$ALUi_2$	*	*	*	Reg[rd]← Op(A,B)	$fetch_0$
…					
J_0	*	*	*	A ← PC	J_1
J_1	*	*	*	B ← IR	J_2
J_2	*	*	*	PC ← JumpTarg(A,B)	$fetch_0$
…					
$beqz_0$	*	*	*	A ← Reg[rs]	$beqz_1$
$beqz_1$	*	yes	*	A ← PC	$beqz_2$
$beqz_1$	*	no	*	….	$fetch_0$
$beqz_2$	*	*	*	B ← $sExt_{16}$(Imm)	$beqz_3$
$beqz_3$	*	*	*	PC ← A+B	$fetch_0$
…					

JumpTarg(A,B) = {A[31:28],B[25:0],00}

CS152 Administrivia

- Tuesday's lecture 1/29 will be a special section to cover the lab infrastructure
- Krste, no office hours Monday - email for alternate time

Size of Control Store



size = $2^{(w+s)}$ x (c + s)

MIPS: w = 6+2 c = 17 s = ?

no. of steps per opcode = 4 to 6 + fetch-sequence

no. of states ≈ (4 steps per op-group) x op-groups + common sequences

= 4 x 8 + 10 states = 42 states ⇒ s = 6

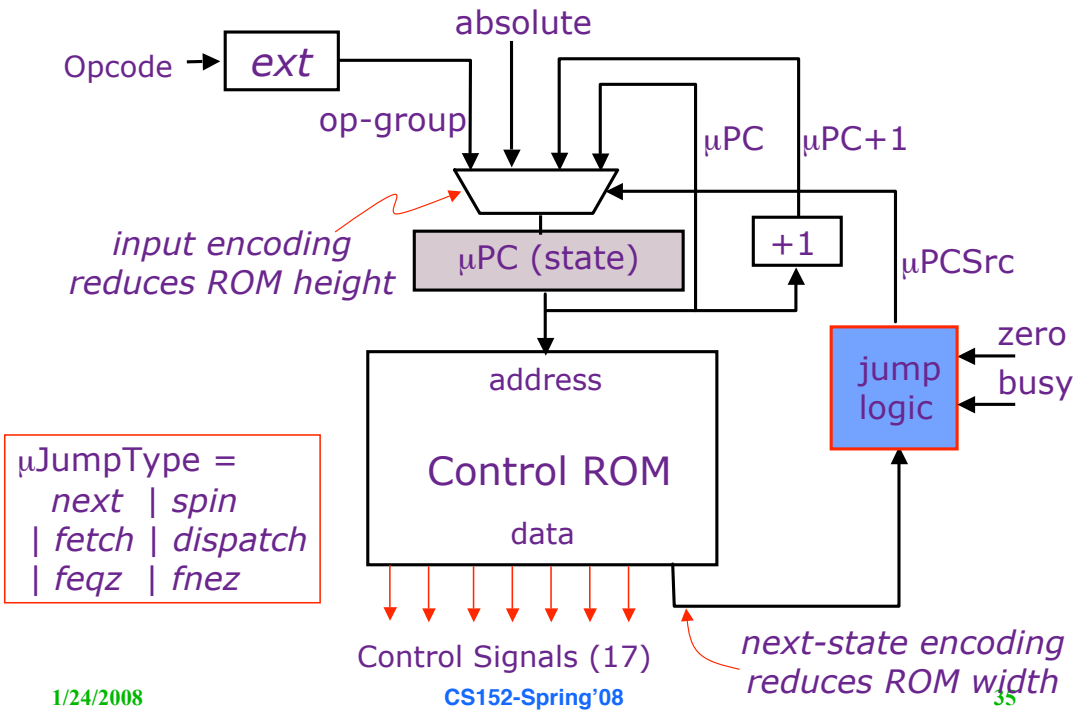
Control ROM = $2^{(8+6)}$ x 23 bits ≈ 48 Kbytes

Reducing Control Store Size

Control store has to be *fast* ⇒ *expensive*

- Reduce the ROM height (= address bits)
  - *reduce inputs by extra external logic*
    each input bit doubles the size of the control store
  - *reduce states by grouping opcodes*
    find common sequences of actions
  - *condense input status bits*
    combine all exceptions into one, i.e., exception/no-exception
- Reduce the ROM width
  - *restrict the next-state encoding*
    Next, Dispatch on opcode, Wait for memory, ...
  - *encode control signals (vertical microcode)*

MIPS Controller V2



Jump Logic

μPCSrc = *Case* μJumpTypes

next	⇒	μPC+1
spin	⇒	if (busy) then μPC else μPC+1
fetch	⇒	absolute
dispatch	⇒	op-group
feqz	⇒	if (zero) then absolute else μPC+1
fnez	⇒	if (zero) then μPC+1 else absolute

Instruction Fetch & ALU: *MIPS-Controller-2*

State	Control points	next-state
$fetch_0$	MA ← PC	next
$fetch_1$	IR ← Memory	spin
$fetch_2$	A ← PC	next
$fetch_3$	PC ← A + 4	dispatch
...		
ALU_0	A ← Reg[rs]	next
ALU_1	B ← Reg[rt]	next
ALU_2	Reg[rd]←func(A,B)	fetch
$ALUi_0$	A ← Reg[rs]	next
$ALUi_1$	B ← $sExt_{16}$(Imm)	next
$ALUi_2$	Reg[rd]← Op(A,B)	fetch

Load & Store: *MIPS-Controller-2*

State	Control points	next-state
LW_0	A ← Reg[rs]	next
LW_1	B ← $sExt_{16}$(Imm)	next
LW_2	MA ← A+B	next
LW_3	Reg[rt] ← Memory	spin
LW_4		fetch
SW_0	A ← Reg[rs]	next
SW_1	B ← $sExt_{16}$(Imm)	next
SW_2	MA ← A+B	next
SW_3	Memory ← Reg[rt]	spin
SW_4		fetch

Branches: *MIPS-Controller-2*

State	Control points	next-state
$BEQZ_0$	A ← Reg[rs]	next
$BEQZ_1$		fnez
$BEQZ_2$	A ← PC	next
$BEQZ_3$	B ← $sExt_{16}$(Imm<<2)	next
$BEQZ_4$	PC ← A+B	fetch
$BNEZ_0$	A ← Reg[rs]	next
$BNEZ_1$		feqz
$BNEZ_2$	A ← PC	next
$BNEZ_3$	B ← $sExt_{16}$(Imm<<2)	next
$BNEZ_4$	PC ← A+B	fetch

Jumps: *MIPS-Controller-2*

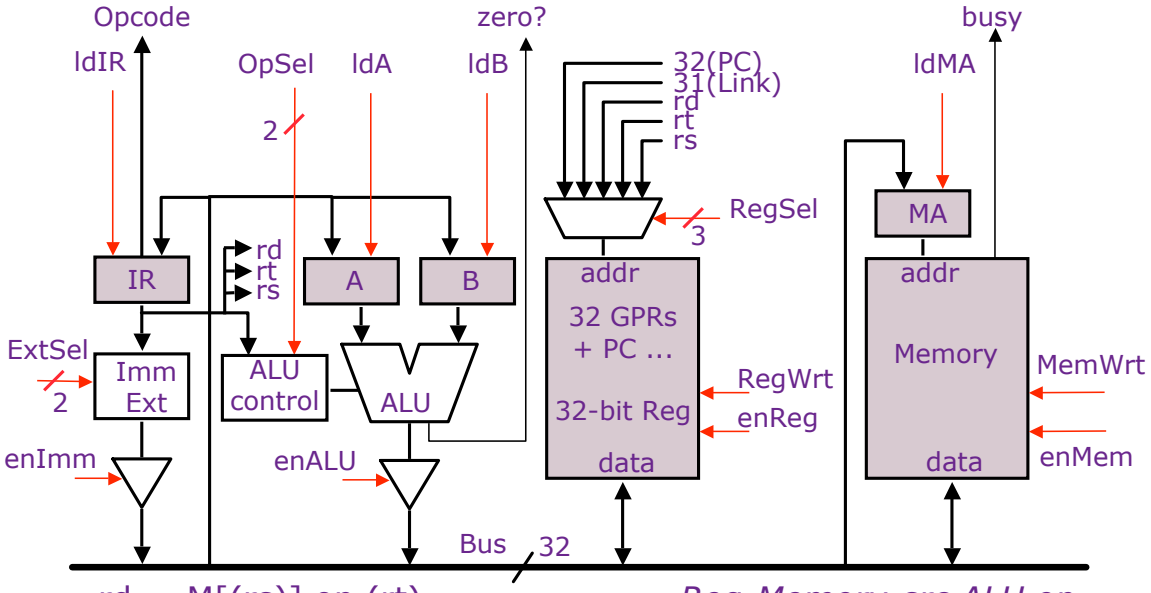
State	Control points	next-state
J_0	A ← PC	next
J_1	B ← IR	next
J_2	PC ← JumpTarg(A,B)	fetch
JR_0	A ← Reg[rs]	next
JR_1	PC ← A	fetch
JAL_0	A ← PC	next
JAL_1	Reg[31] ← A	next
JAL_2	B ← IR	next
JAL_3	PC ← JumpTarg(A,B)	fetch
$JALR_0$	A ← PC	next
$JALR_1$	B ← Reg[rs]	next
$JALR_2$	Reg[31] ← A	next
$JALR_3$	PC ← B	fetch

VAX 11-780 Microcode

```
; P1WFUD,!    [600,1205]      MICRO2  1F(12)     26-May-81  14:58:1      VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122        Page  771
; CALL2 ,MIC  [600,1205]      Procedure call          : CALLG, CALLS

                                                ;29744  ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
                                                ;29745
                                                ;29746  W0      ;-------------------------------;CALL SITE FOR MPUSH
                                                ;29747  CALL,7: D_0,AND,RC[T2],                 ;STRIP MASK TO BITS 11:0
6557K     0  U 11F4, 0811,2035,0180,F910,0000,0CD8  ;29748          CALL,J/MPUSH                          ;PUSH REGISTERS
                                                ;29749
                                                ;29750          ;-------------------------------;RETURN FROM MPUSH
                                                ;29751          CACHE_D[LONG],                  ;PUSH PC
6557K 7763K U 11F5, 0000,003C,0180,3270,0000,134A  ;29752          LAR_R[SP]                             ; BY SP
                                                ;29753
                                                ;29754          ;-------------------------------;
6856K     0  U 134A, 0018,0000,0180,FAF0,0200,134C  ;29755  CALL,8: R[SP]&VA_LA-K[,8]                ;UPDATE SP FOR PUSH OF PC &
                                                ;29756
                                                ;29757          ;-------------------------------;
6856K     0  U 134C, 0800,003C,0180,FA68,0000,11F8  ;29758          D_R[FP]                               ;READY TO PUSH FRAME POINTER
                                                ;29759
                                                ;29760  W0      ;-------------------------------;CALL SITE FOR PSHSP
                                                ;29761          CACHE_D[LONG],                  ;STORE FP,
                                                ;29762          LAR_R[SP],                      ; GET SP AGAIN
                                                ;29763          SC_K[,FFF0],                    ;-16 TO SC
6856K    21M U 11F8, 0000,003D,6D80,3270,0084,6CD9  ;29764          CALL,J/PSHSP
                                                ;29765
                                                ;29766          ;-------------------------------;
                                                ;29767          D_R[AP],                        ;READY TO PUSH AP
6856K     0  U 11F9, 0800,003C,3DF0,2E60,0000,134D  ;29768          O_ID[PSL]                             ; AND GET PSW FOR COMBINATIO
                                                ;29769
                                                ;29770          ;-------------------------------;
                                                ;29771          CACHE_D[LONG],                  ;STORE OLD AP
                                                ;29772          O_0,ANDNOT,K[,1F],              ;CLEAR PSW<T,N,Z,V,C>
6856K    21M U 134D, 0019,2024,8DC0,3270,0000,134E  ;29773          LAR_R[SP]                             ;GET SP INTO LATCHES AGAIN
                                                ;29774
                                                ;29775          ;-------------------------------;
6856K     0  U 134E, 2010,0038,0180,F909,4200,1350  ;29776          PC&VA_RC[T1], FLUSH,IB                ; LOAD NEW PC AND CLEAR OUT
                                                ;29777
                                                ;29778          ;-------------------------------;
                                                ;29779          D_DAL,SC,                       ;PSW TO D<31:16>
                                                ;29780          O_RC[T2],                       ;RECOVER MASK
                                                ;29781          SC_SC+K[,3],                    ;PUT -13 IN SC
6856K     0  U 1350, 0D10,0038,0DC0,6114,0084,9351  ;29782          LOAD,IB, PC_PC+1                      ;START FETCHING SUBROUTINE I
                                                ;29783
                                                ;29784          ;-------------------------------;
                                                ;29785          D_DAL,SC,                       ;MASK AND PSW IN D<31:03>
                                                ;29786          O_PC[T4],                       ;GET LOW BITS OF OLD SP TO Q<1:0>
6856K     0  U 1351, 0D10,0038,F5C0,F920,0084,9352  ;29787          SC_SC+K[,A]                           ;PUT -3 IN SC
                                                ;29788
```


Implementing Complex Instructions



rd ← M[(rs)] op (rt) *Reg-Memory-src ALU op*

M[(rd)] ← (rs) op (rt) *Reg-Memory-dst ALU op*

M[(rd)] ← M[(rs)] op M[(rt)] *Mem-Mem ALU op*

Mem-Mem ALU Instructions:

MIPS-Controller-2

Mem-Mem ALU op M[(rd)] ← M[(rs)] op M[(rt)]

$ALUMM_0$	MA ← Reg[rs]	next
$ALUMM_1$	A ← Memory	spin
$ALUMM_2$	MA ← Reg[rt]	next
$ALUMM_3$	B ← Memory	spin
$ALUMM_4$	MA ←Reg[rd]	next
$ALUMM_5$	Memory ← func(A,B)	spin
$ALUMM_6$		fetch

Complex instructions usually do not require datapath modifications in a microprogrammed implementation
-- only extra space for the control program

Implementing these instructions using a hardwired controller is difficult without datapath modifications

Performance Issues

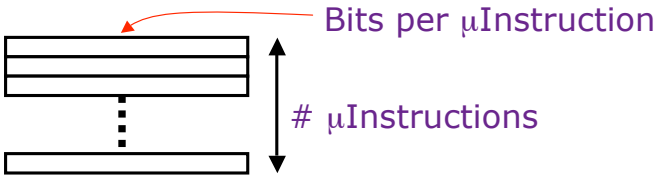
Microprogrammed control
⇒ multiple cycles per instruction

Cycle time ?
$t_C > \max(t_{reg\text{-}reg}, t_{ALU}, t_{\mu ROM})$

Suppose $10 * t_{\mu ROM} < t_{RAM}$

Good performance, relative to a single-cycle hardwired implementation, can be achieved even with a CPI of 10

Horizontal vs Vertical μCode

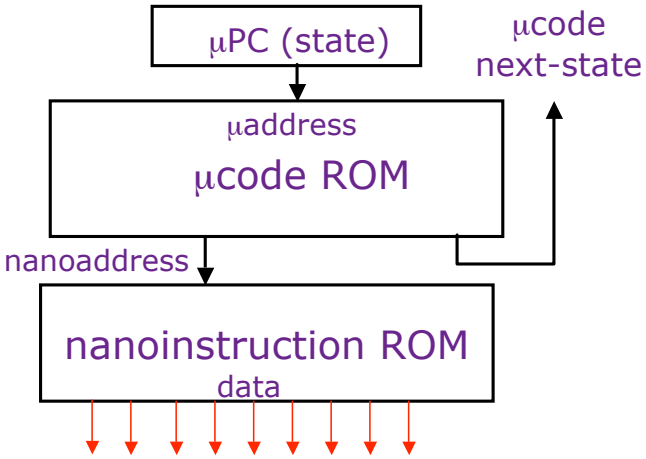


- Horizontal μcode has wider μinstructions
  - Multiple parallel operations per μinstruction
  - Fewer steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical μcode has narrower μinstructions
  - Typically a single datapath operation per μinstruction
    - separate μinstruction for branches
  - More steps to per macroinstruction
  - More compact ⇒ less bits
- Nanocoding
  - Tries to combine best of horizontal and vertical μcode

Nanocoding

Exploits recurring control signal patterns in μcode, e.g.,

ALU_0 A ← Reg[rs]
...
$ALUi_0$ A ← Reg[rs]
...



- MC68000 had 17-bit μcode containing either 10-bit μjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

Microprogramming in IBM 360

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
μinst width (bits)	50	52	85	87
μcode size (K minsts)	4	4	2.75	2.75
μstore technology	CCROS	TCROS	BCROS	BCROS
μstore cycle (ns)	750	625	500	200
memory cycle (ns)	1500	2500	2000	750
Rental fee ($K/month)	4	7	15	35

Only the fastest models (75 and 95) were hardwired

Microcode Emulation

- IBM initially miscalculated the importance of software compatibility with earlier models when introducing the 360 series
- Honeywell stole some IBM 1401 customers by offering translation software ("Liberator") for Honeywell H200 series machine
- IBM retaliated with optional additional microcode for 360 series that could emulate IBM 1401 ISA, later extended for IBM 7000 series
  - one popular program on 1401 was a 650 simulator, so some customers ran many 650 programs on emulated 1401s
    - (*650 simulated on 1401 emulated on 360)*

Microprogramming thrived in the Seventies

- Significantly faster ROMs than DRAMs were available
- For complex instruction sets, datapath and controller were *cheaper and simpler*
- *New instructions* , e.g., floating point, could be supported without datapath modifications
- *Fixing bugs* in the controller was easier
- ISA compatibility across various models could be achieved easily and cheaply

Except for the cheapest and fastest machines, all computers were microprogrammed

Writable Control Store (WCS)

- Implement control store in RAM not ROM
  - MOS SRAM memories now almost as fast as control store (core memories/DRAMs were 2-10x slower)
  - Bug-free microprograms difficult to write
- User-WCS provided as option on several minicomputers
  - Allowed users to change microcode for each processor
- User-WCS *failed*
  - Little or no programming tools support
  - Difficult to fit software into small space
  - Microcode control tailored to original ISA, less useful for others
  - Large WCS part of processor state - expensive context switches
  - Protection difficult if user can change microcode
  - Virtual memory required *restartable* microcode

Microprogramming: early Eighties

- Evolution bred more complex micro-machines
  - Complex instruction sets led to need for subroutine and call stacks in µcode
  - Need for fixing bugs in control programs was in conflict with read-only nature of µROM
  - --> WCS (B1700, QMachine, Intel i432, …)
- With the advent of VLSI technology assumptions about ROM & RAM speed became invalid -> more complexity
- Better compilers made complex instructions less important.
- Use of numerous micro-architectural innovations, e.g., pipelining, caches and buffers, made multiple-cycle execution of reg-reg instructions unattractive
- Looking ahead to RISC next time
  - Use chip area to build fast instruction cache of user-visible vertical microinstructions - use software subroutine not hardware microroutines
  - Use simple ISA to enable hardwired pipelined implementation

Modern Usage

- *Microprogramming is far from extinct*
- Played a crucial role in micros of the Eighties
  DEC uVAX, Motorola 68K series, Intel 386 and 486
- Microcode pays an assisting role in most modern micros *(AMD Athlon, Intel Core 2 Duo, IBM PowerPC)*
  - Most instructions are executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke the microcode engine
- *Patchable* microcode common for post-fabrication bug fixes, e.g. Intel Pentiums load µcode patches at bootup

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252