CSC 631: High-Performance Computer Architecture

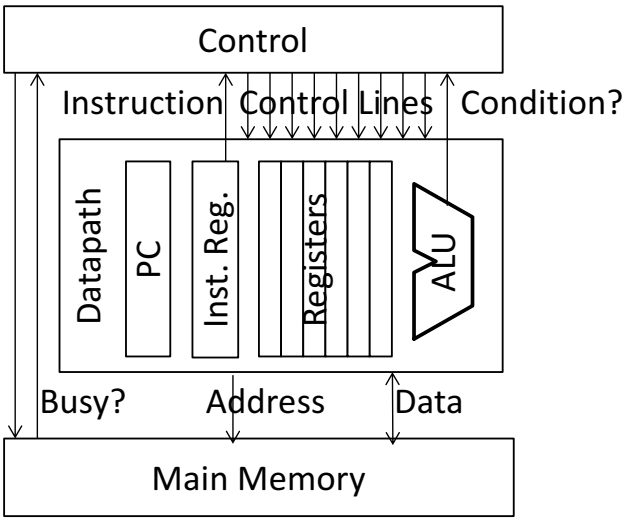
Spring 2017
Lecture 3: CISC versus RISC

Instruction Set Architecture (ISA)

- The contract between software and hardware
- Typically described by giving all the programmer-visible state (registers + memory) plus the semantics of the instructions that operate on that state
- IBM 360 was first line of machines to separate ISA from implementation (aka. microarchitecture)
- Many implementations possible for a given ISA
  - E.g., the Soviets build code-compatible clones of the IBM360, as did Amdahl after he left IBM.
  - E.g.2., today can buy AMD or Intel processors that run x86 ISA.
  - E.g.3: many cellphones use ARM ISA with implementations from many different companies including Apple, Qualcomm, Samsung, etc.
- We use Berkeley RISC-V 2.0 as standard ISA in this course
  - **`www.riscv.org`**

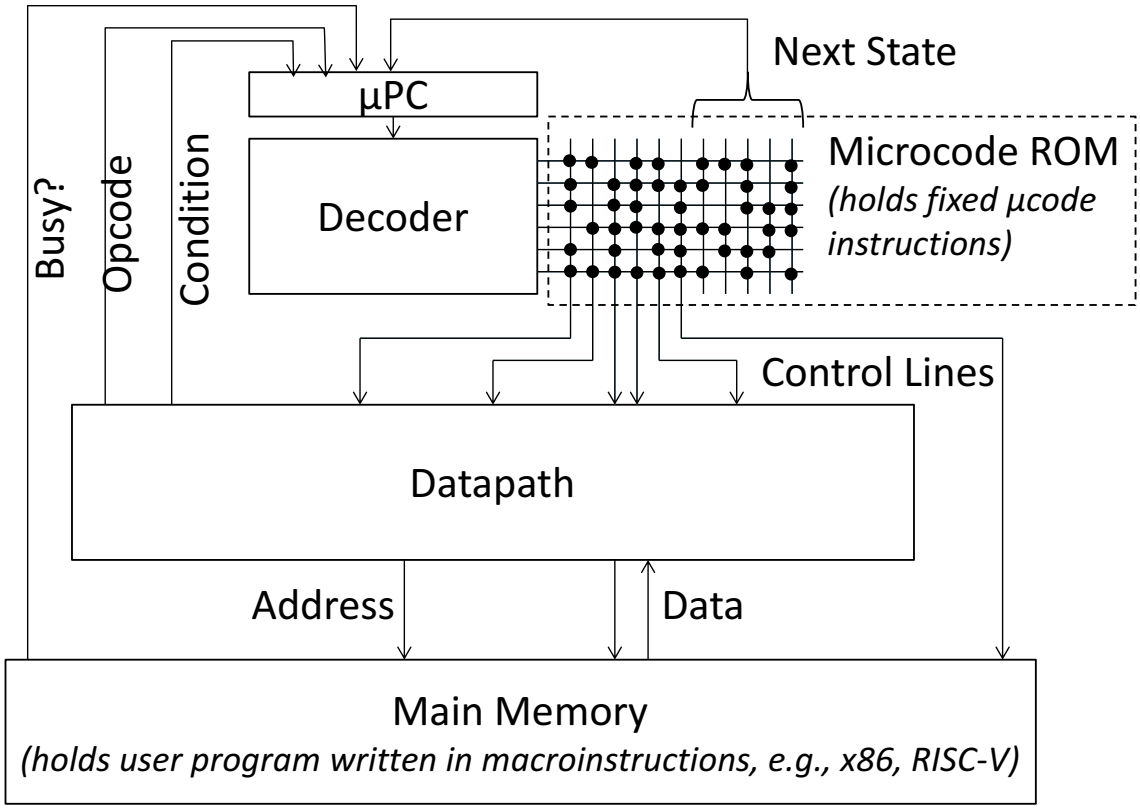
Control versus Datapath

- Processor designs can be split between *datapath*, where numbers are stored and arithmetic operations computed, and *control*, which sequences operations on datapath



- Biggest challenge for early computer designers was getting control circuitry correct
- Maurice Wilkes invented the idea of microprogramming to design the control unit of a processor for EDSAC-II, 1958
  - Foreshadowed by Babbage's "Barrel" and mechanisms in earlier programmable calculators

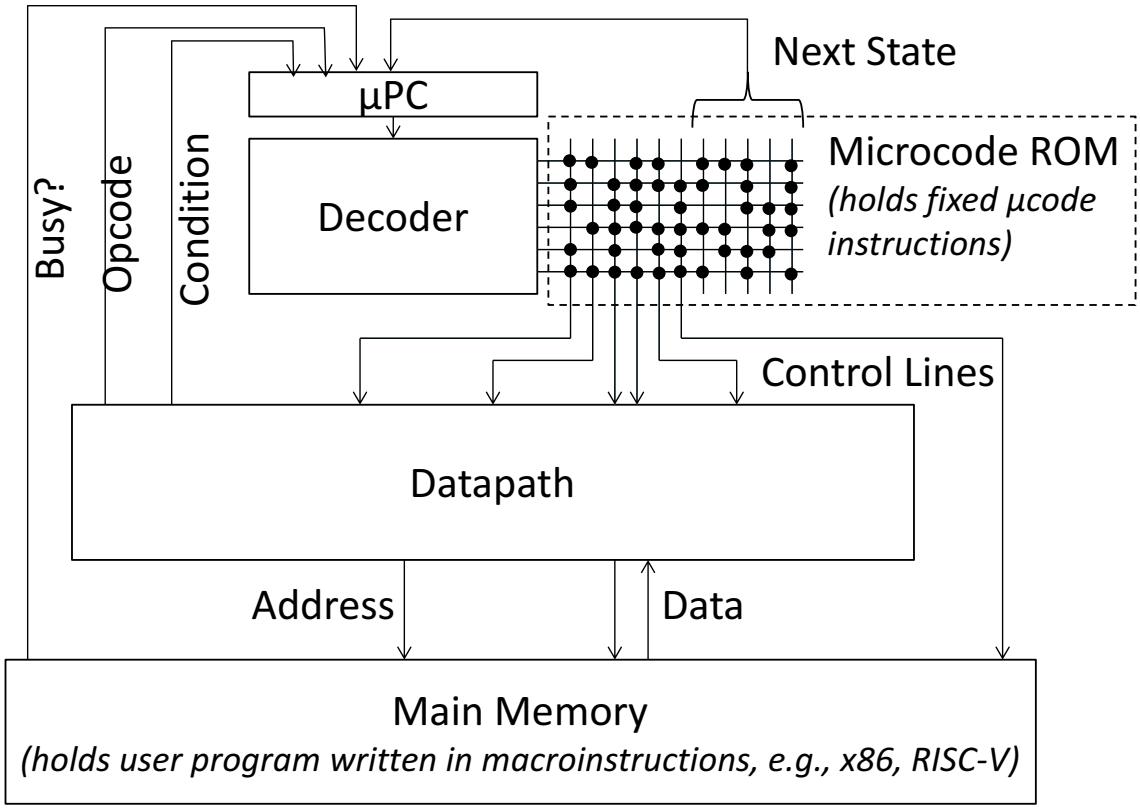
Microcoded CPU



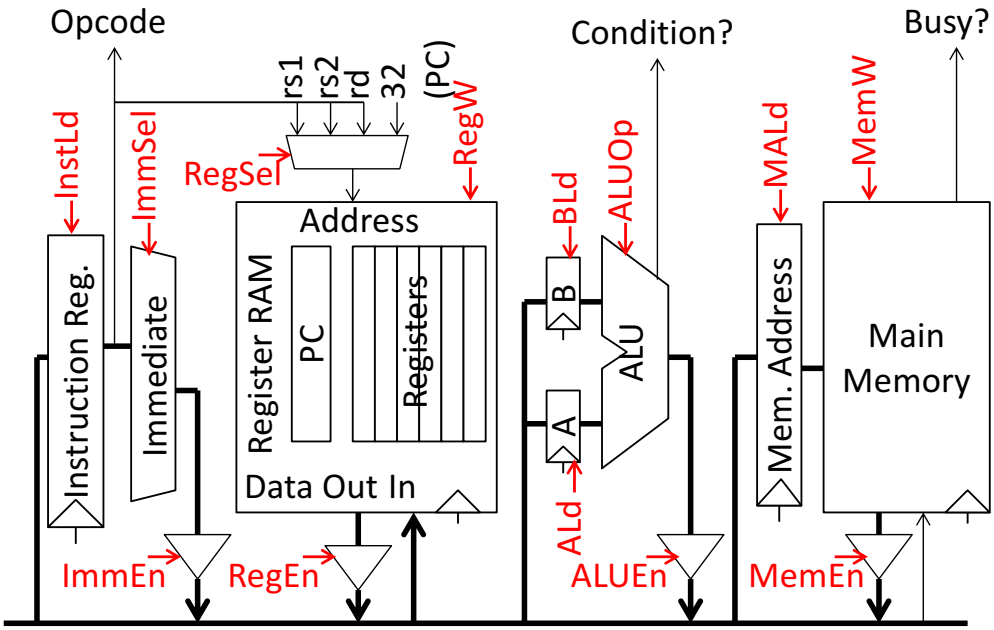
Technology Influence

- When microcode appeared in 50s, different technologies for:
  - Logic: Vacuum Tubes
  - Main Memory: Magnetic cores
  - Read-Only Memory: Diode matrix, punched metal cards,…
- Logic very expensive compared to ROM or RAM
- ROM cheaper than RAM
- ROM much faster than RAM

Microcoded CPU



Single Bus Datapath for Microcoded RISC-V



Microinstructions written as register transfers:

- MA:=PC means RegSel=PC; RegW=0; RegEn=1; MALd=1
- B:=Reg[rs2] means RegSel=rs2; RegW=0; RegEn=1; BLd=1
- Reg[rd]:=A+B means ALUop=Add; ALUEn=1; RegSel=rd; RegW=1

RISC-V Instruction Execution Phases

- Instruction Fetch
- Instruction Decode
- Register Fetch
- ALU Operations
- Optional Memory Operations
- Optional Register Writeback
- Calculate Next Instruction Address

Microcode Sketches (1)

Instruction Fetch:
MA,A:=PC
PC:=A+4
wait for memory
IR:=Mem
dispatch on opcode

ALU:
A:=Reg[rs1]
B:=Reg[rs2]
Reg[rd]:=ALUOp(A,B)
goto instruction fetch

ALUI:
A:=Reg[rs1]
B:=ImmI //Sign-extend 12b immediate
Reg[rd]:=ALUOp(A,B)
goto instruction fetch

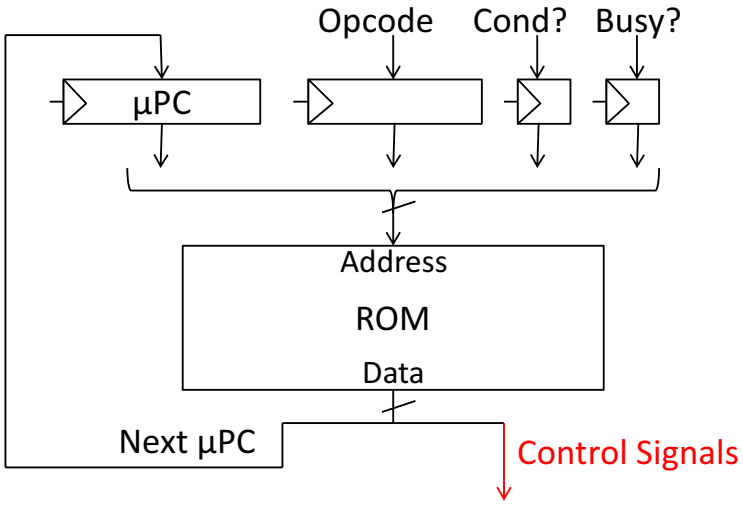
Microcode Sketches (2)

LW:
A:=Reg[rs1]
B:=ImmI //Sign-extend 12b immediate
MA:=A+B
wait for memory
Reg[rd]:=Mem
goto instruction fetch

JAL:
Reg[rd]:=A // Store return address
A:=A-4 // Recover original PC
B:=ImmJ // Jump-style immediate
PC:=A+B
goto instruction fetch

Branch:
A:=Reg[rs1]
B:=Reg[rs2]
if (!ALUOp(A,B)) *goto instruction fetch* //Not taken
A:=PC //Microcode fall through if branch taken
A:=A-4
B:=ImmB// Branch-style immediate
PC:=A+B
goto instruction fetch

Pure ROM Implementation



- How many address bits?
  |µaddress| = |µPC|+|opcode|+ 1 + 1
- How many data bits?
  |data| = |µPC|+|control signals| = |µPC| + 18
- Total ROM size = $2^{|\mu address|}$x|data|

Pure ROM Contents

	Address			Data	
µPC	Opcode	Cond?	Busy?	Control Lines	Next µPC
fetch0	X	X	X	MA,A:=PC	fetch1
fetch1	X	X	1		fetch1
fetch1	X	X	0	IR:=Mem	fetch2
fetch2	ALU	X	X	PC:=A+4	ALU0
fetch2	ALUI	X	X	PC:=A+4	ALUI0
fetch2	LW	X	X	PC:=A+4	LW0
….					
ALU0	X	X	X	A:=Reg[rs1]	ALU1
ALU1	X	X	X	B:=Reg[rs2]	ALU2
ALU2	X	X	X	Reg[rd]:=ALUOp(A,B)	fetch0

Single-Bus Microcode RISC-V ROM Size

- Instruction fetch sequence 3 common steps
- ~12 instruction groups
- Each group takes ~5 steps (1 for dispatch)
- Total steps 3+12*5 = 63, needs 6 bits for µPC

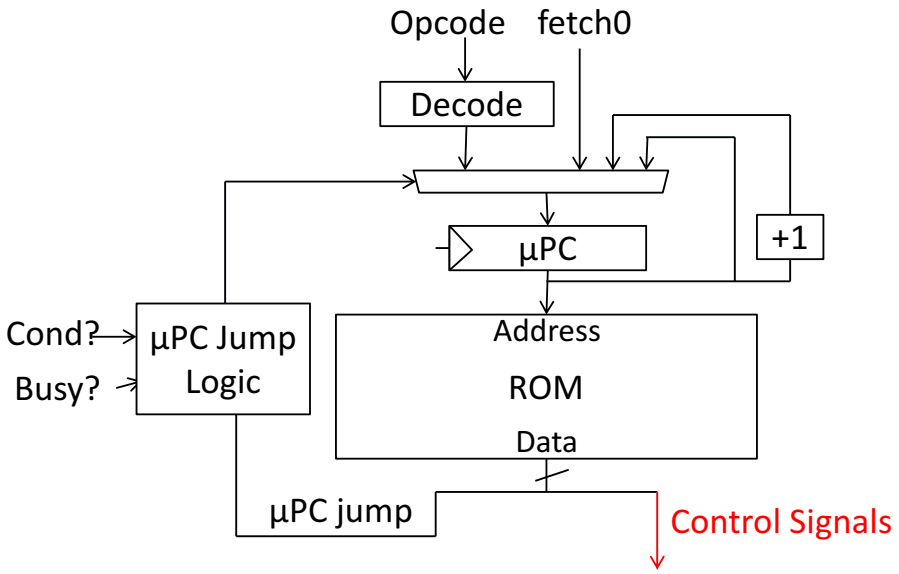
- Opcode is 5 bits, ~18 control signals

- Total size = $2^{(6+5+2)}$x(6+18)=2^{13}x24 = ~25KB!

Reducing Control Store Size

- Reduce ROM height (#address bits)
  - Use external logic to combine input signals
  - Reduce #states by grouping opcodes
- Reduce ROM width (#data bits)
  - Restrict µPC encoding (next,dispatch,wait on memory,…)
  - Encode control signals (vertical µcoding, nanocoding)

Single-Bus RISC-V Microcode Engine



µPC jump = next | spin | fetch | dispatch | ftrue | ffalse

µPC Jump Types

- *next* increments µPC
- *spin* waits for memory
- *fetch* jumps to start of instruction fetch
- *dispatch* jumps to start of decoded opcode group
- *fture/ffalse* jumps to fetch if Cond? true/false

Encoded ROM Contents

Address	Data	
µPC	Control Lines	Next µPC
fetch0	MA,A:=PC	next
fetch1	IR:=Mem	spin
fetch2	PC:=A+4	dispatch
ALU0	A:=Reg[rs1]	next
ALU1	B:=Reg[rs2]	next
ALU2	Reg[rd]:=ALUOp(A,B)	fetch
Branch0	A:=Reg[rs1]	next
Branch1	B:=Reg[rs2]	next
Branch2	A:=PC	ffalse
Branch3	A:=A-4	next
Branch4	B:=ImmB	next
Branch5	PC:=A+B	fetch

Implementing Complex Instructions

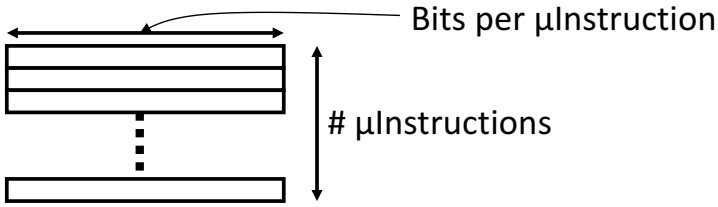
Memory-memory add: M[rd] = M[rs1] + M[rs2]

Address	Data	
µPC	Control Lines	Next µPC
MMA0	MA:=Reg[rs1]	next
MMA1	A:=Mem	spin
MMA2	MA:=Reg[rs2]	next
MMA3	B:=Mem	spin
MMA4	MA:=Reg[rd]	next
MMA5	Mem:=ALUOp(A,B)	spin
MMA6		fetch

Complex instructions usually do not require datapath modifications, only extra space for control program

Very difficult to implement these instructions using a hardwired controller without substantial datapath modifications

Horizontal vs Vertical µCode

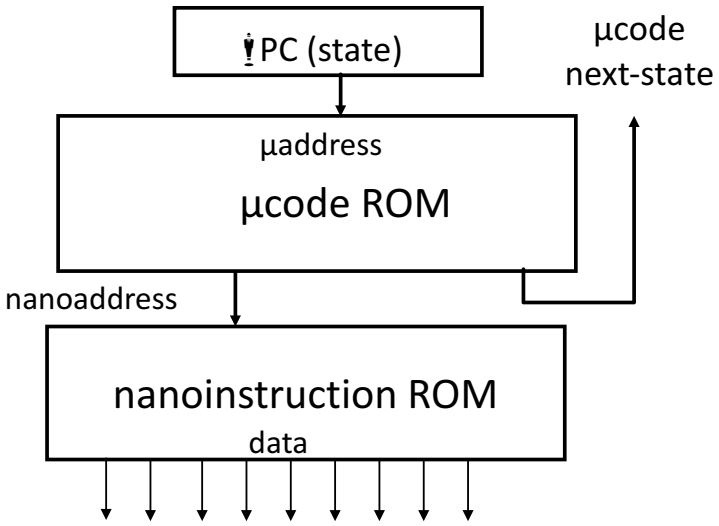


- Horizontal µcode has wider µinstructions
  - Multiple parallel operations per µinstruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical µcode has narrower µinstructions
  - Typically a single datapath operation per µinstruction
    - separate µinstruction for branches
  - More microcode steps 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.,

ALU0 A ← Reg[rs1]
...
ALUI0 A ← Reg[rs1]
...



- Motorola 68000 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

IBM 360: Initial Implementations

	Model 30	. . .	*Model 70*
Storage	8K - 64 KB		256K - 512 KB
Datapath	8-bit		64-bit
Circuit Delay	30 nsec/level		5 nsec/level
Local Store	Main Store		Transistor Registers
Control Store	Read only 1 µsec		Conventional circuits

IBM 360 instruction set architecture (ISA) completely hid the underlying technological differences between various models.

Milestone: The first true ISA designed as portable hardware-software interface!

With minor modifications it still survives today!

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 µinsts)	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 '60s and '70s

- 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

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
  - →Writable Control Store (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

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

Analyzing Microcoded Machines

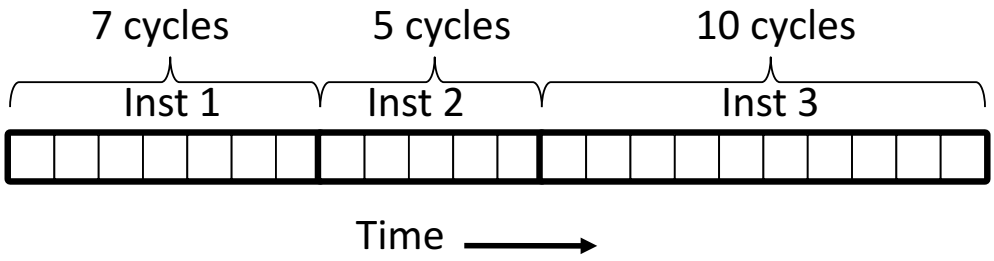
- John Cocke and group at IBM
  - Working on a simple pipelined processor, 801, and advanced compilers inside IBM
  - Ported experimental PL.8 compiler to IBM 370, and only used simple register-register and load/store instructions similar to 801
  - Code ran faster than other existing compilers that used all 370 instructions! (up to 6MIPS whereas 2MIPS considered good before)
- Emer, Clark, at DEC
  - Measured VAX-11/780 using external hardware
  - Found it was actually a 0.5MIPS machine, although usually assumed to be a 1MIPS machine
  - Found 20% of VAX instructions responsible for 60% of microcode, but only account for 0.2% of execution time!
- VAX8800
  - Control Store: 16K*147b RAM, Unified Cache: 64K*8b RAM
  - 4.5x more microstore RAM than cache RAM!

"Iron Law" of Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends on ISA and µarchitecture
- Time per cycle depends upon the µarchitecture and base technology

CPI for Microcoded Machine



Total clock cycles = 7+5+10 = 22
Total instructions = 3
CPI = 22/3 = 7.33
CPI is always an average over a large number of instructions.

IC Technology Changes Tradeoffs

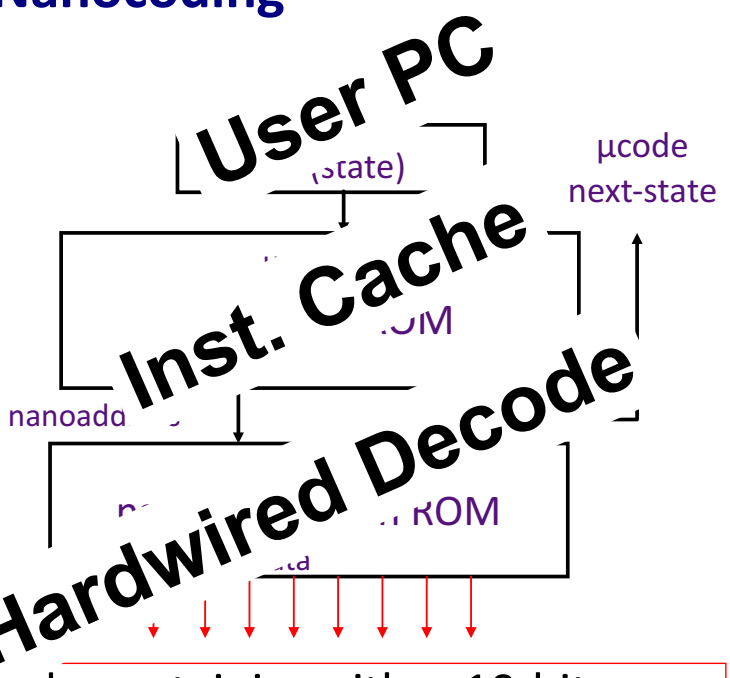
- Logic, RAM, ROM all implemented using MOS transistors
- Semiconductor RAM ~ same speed as ROM

Nanocoding

RISC

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

ALU_0 A ← Reg[rs1]
...
$ALUi_0$ A ← Reg[rs1]
...



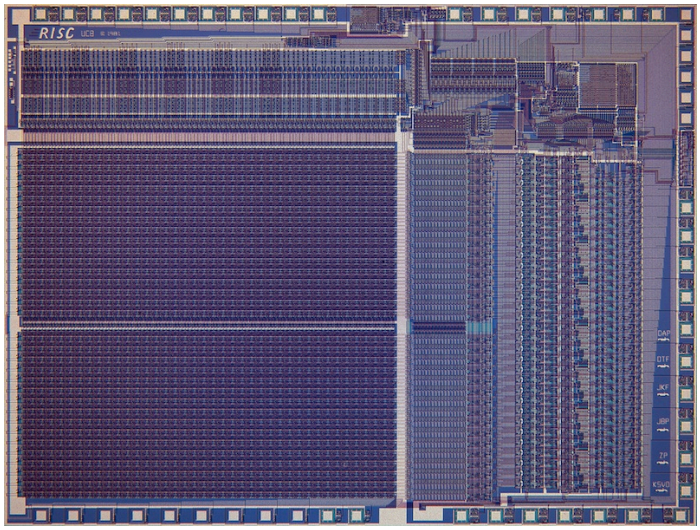
- MC68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

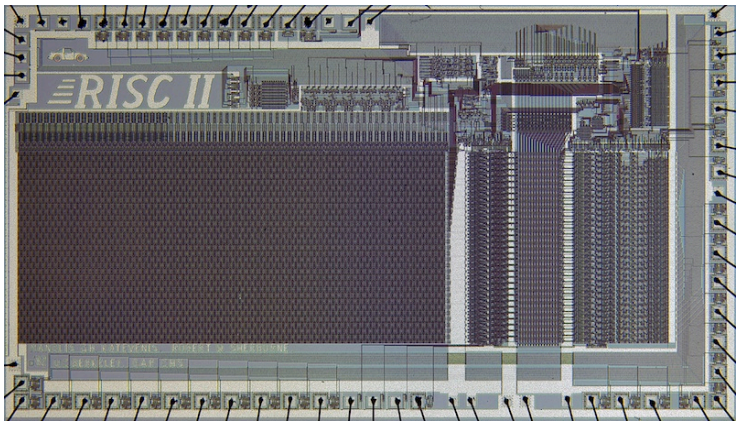
From CISC to RISC

- Use fast RAM to build fast instruction *cache* of user-visible instructions, not fixed hardware microroutines
  - Contents of fast instruction memory change to fit what application needs right now
- Use simple ISA to enable hardwired pipelined implementation
  - Most compiled code only used a few of the available CISC instructions
  - Simpler encoding allowed pipelined implementations
- Further benefit with integration
  - In early '80s, could finally fit 32-bit datapath + small caches on a single chip
  - No chip crossings in common case allows faster operation

Berkeley RISC Chips

RISC-I (1982) Contains 44,420 transistors, fabbed in 5 µm NMOS, with a die area of 77 mm², ran at 1 MHz. This chip is probably the first VLSI RISC.





RISC-II (1983) contains 40,760 transistors, was fabbed in 3 µm NMOS, ran at 3 MHz, and the size is 60 mm².

Stanford built some too…

Microprogramming is far from extinct

- Played a crucial role in micros of the Eighties
  - DEC uVAX, Motorola 68K series, Intel 286/386
- Plays an assisting role in most modern micros
  - e.g., AMD Bulldozer, Intel Ivy Bridge, Intel Atom, IBM PowerPC, …
  - Most instructions executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke microcode
- Patchable microcode common for post-fabrication bug fixes, e.g. Intel processors load µcode patches at bootup

Acknowledgements

- These course notes were developed by:
  - Krste Asanovic (UCB)
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)