CSC 631: High-Performance Computer Architecture

Fall 2022
Lecture 2: Instruction Set Architectures

Last Time in Lecture 1

- Technology and Applications shape Computer Architecture
  - History provides lessons for the future
- First 130 years of CompArch, from Babbage to IBM 360
  - Move from calculators (no conditionals) to fully programmable machines
  - Rapid change started in WWII (mid-1940s), move from electro-mechanical to pure electronic processors
- Cost of software development becomes a large constraint on architecture (need compatibility)
- IBM 360 introduces notion of "family of machines" running same ISA but very different implementations
  - Six different machines released on same day (April 7, 1964)
  - "Future-proofing" for subsequent generations of machine

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., Soviets built code-compatible clones of the IBM360, as did Amdahl after he left IBM.
  - E.g.2., AMD, Intel, VIA processors run the AMD64 ISA
  - E.g.3: many cellphones use the ARM ISA with implementations from many different companies including Apple, Qualcomm, Samsung, Huawei, etc.
- This course uses RISC-V as standard ISA (www.riscv.org)
  - Many companies and open-source projects build RISC-V implementations

ISA to Microarchitecture Mapping

- ISA often designed with particular microarchitectural style in mind, e.g.,

  Accumulator ⇒ hardwired, unpipelined
  CISC ⇒ microcoded
  RISC ⇒ hardwired, pipelined
  VLIW ⇒ fixed-latency in-order parallel pipelines
  JVM ⇒ software interpretation
- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Apple M1 (native ARM ISA, emulates x86 in software)
  - Spike: Software-interpreted RISC-V machine
  - ARM Jazelle: A hardware JVM processor
  - **This lecture: a microcoded RISC-V machine**

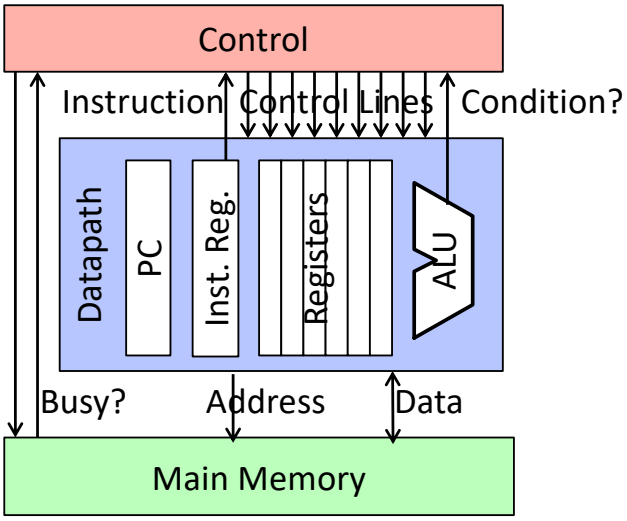
Why Learn Microprogramming?

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

** "CISC"/"RISC" names much newer than style of machines they refer to.*

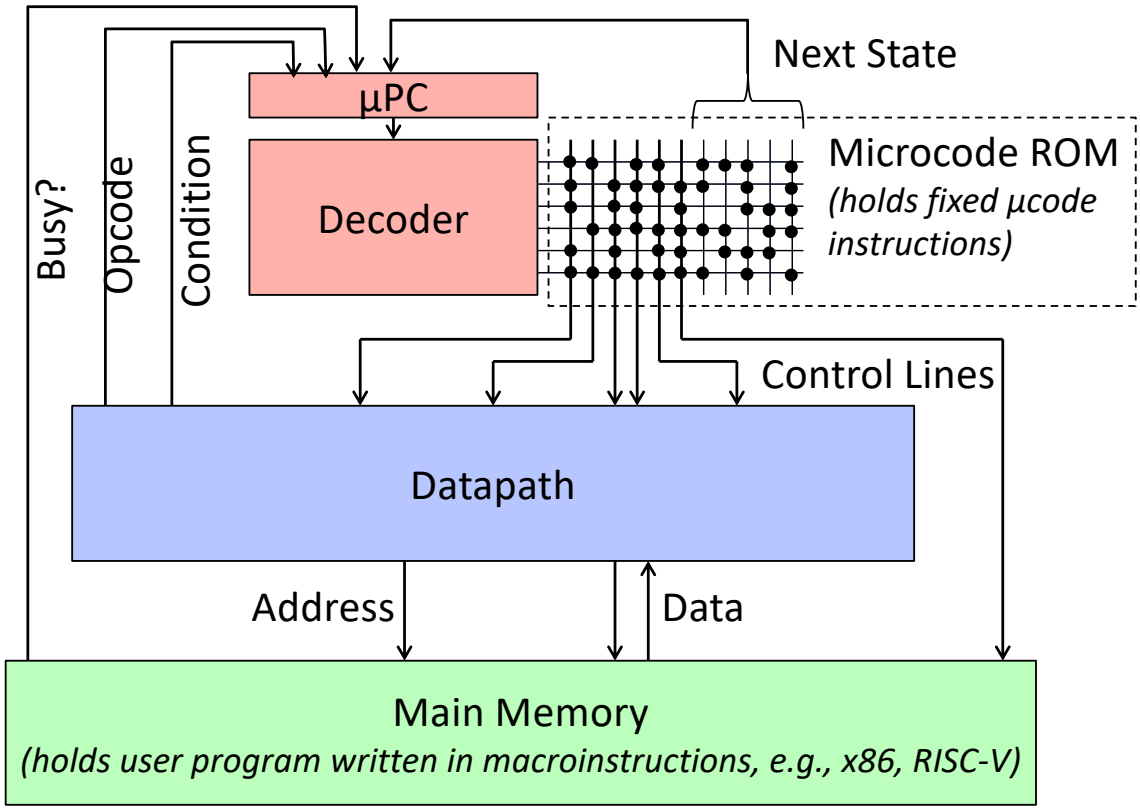
Control versus Datapath

- 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

Microcoded CPU



Technology Influence

- When microcode appeared in 1950s, 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

RISC-V ISA

- New fifth-generation RISC design from UC Berkeley
- Realistic & complete ISA, but open & small
- Not over-architected for a certain implementation style
- Both 32-bit (RV32) and 64-bit (RV64) address-space variants
- Designed for multiprocessing
- Efficient instruction encoding
- Easy to subset/extend for education/research
- RISC-V spec available on Foundation website and github
- Increasing momentum with industry adoption

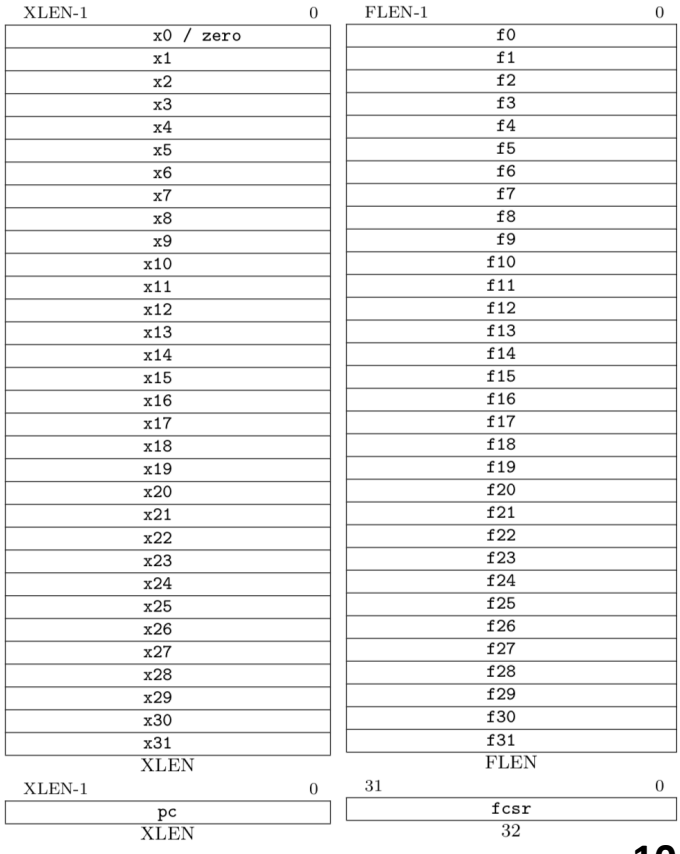
RV32I Processor State

Program counter (**pc**)

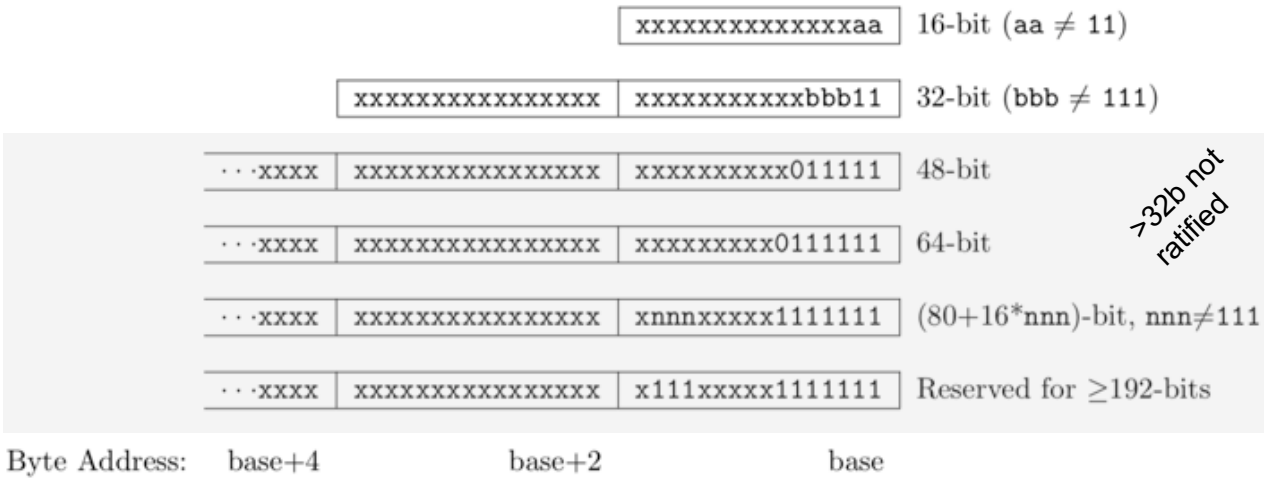
32x32-bit integer registers (**x0-x31**)
• **x0** always contains a 0

32 floating-point (FP) registers (**f0-f31**)
• each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

FP status register (**fcsr**), used for FP rounding mode & exception reporting

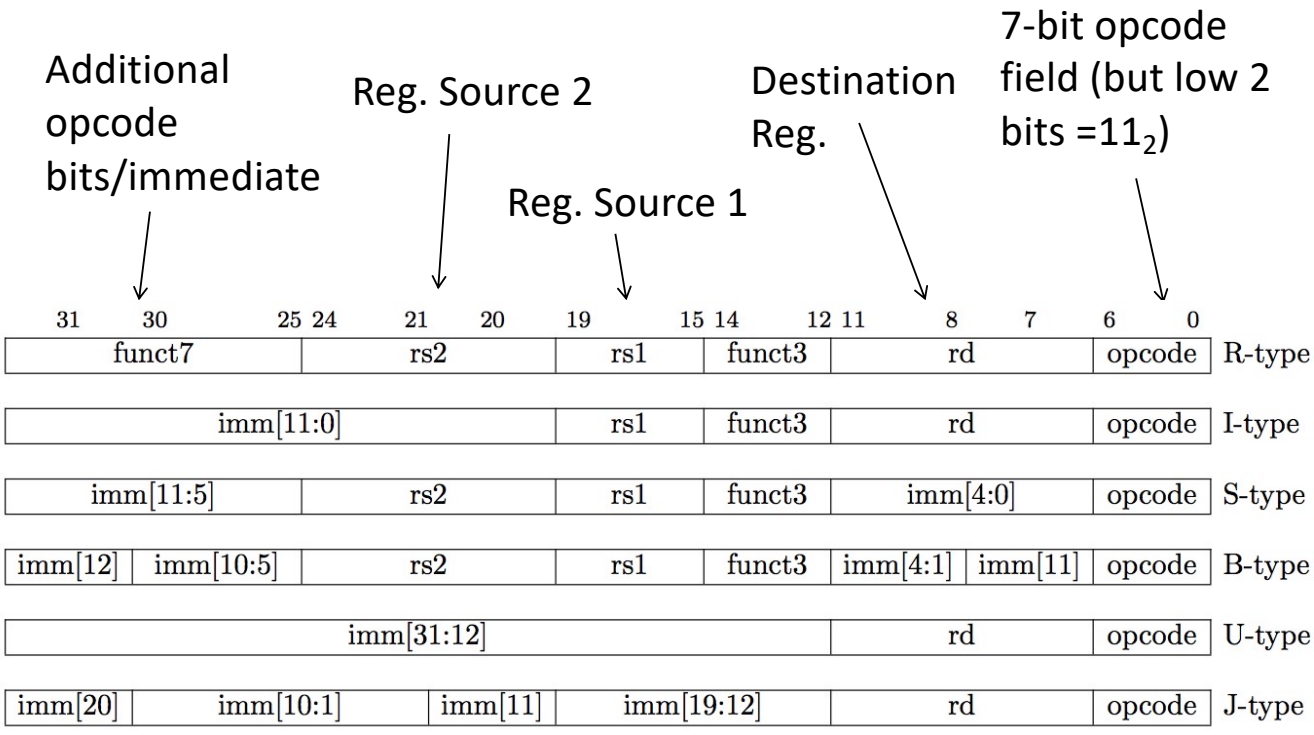


RISC-V Instruction Encoding

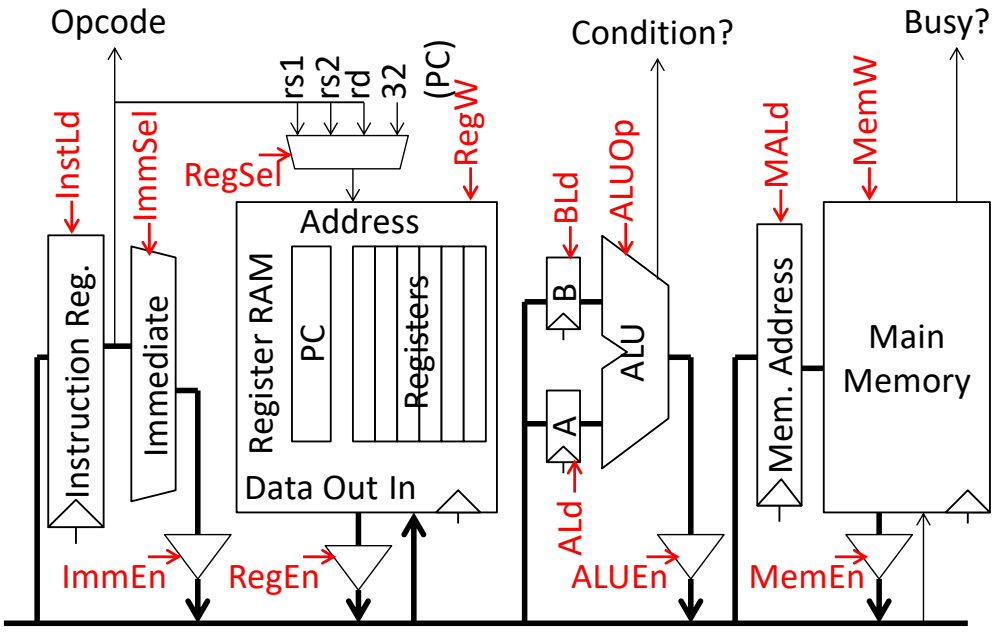


- Can support variable-length instructions.
- Base instruction set (RV32) always has fixed 32-bit instructions lowest two bits = 11_2
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)

RISC-V Instruction Formats



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

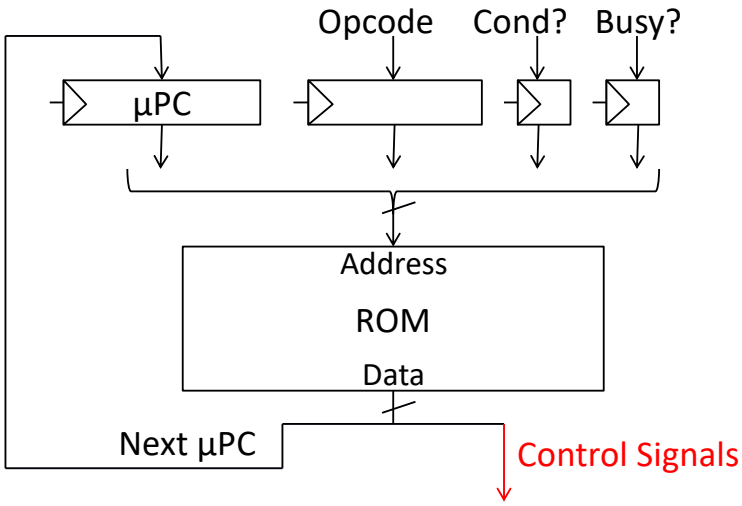
Microcode Sketches (2)

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

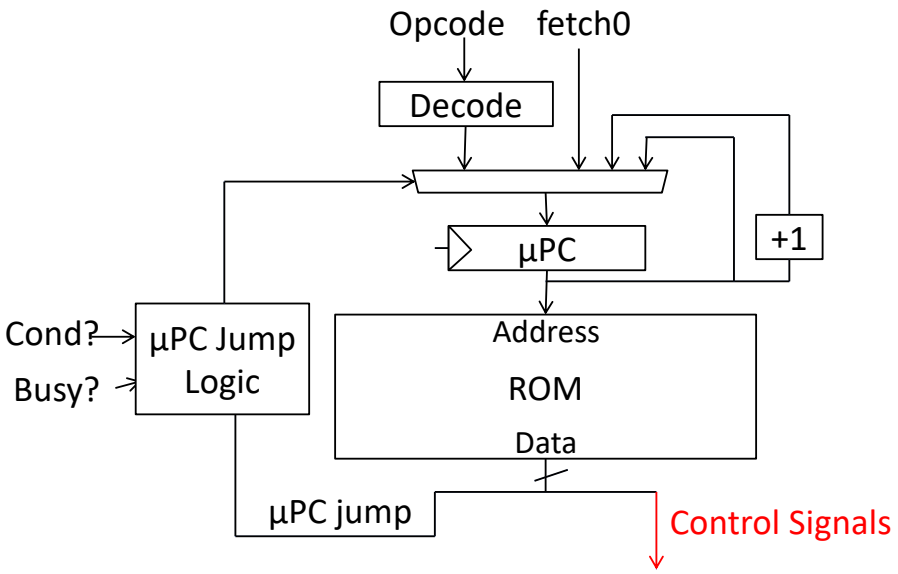
- Opcode is 5 bits, ~18 control signals

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

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
- *ftrue/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

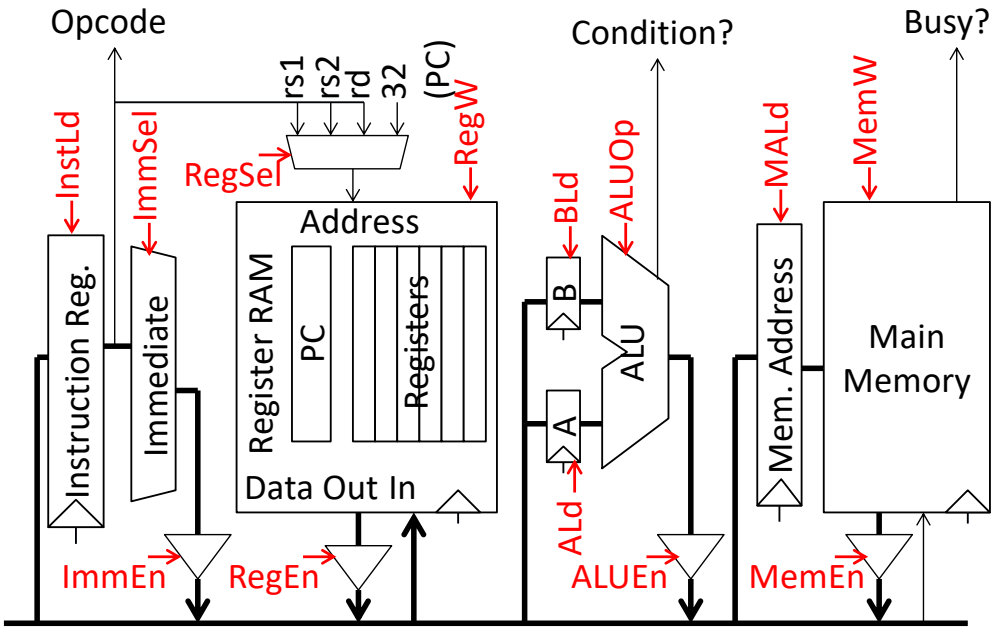
Memory-memory add: M[rd] = M[rs1] + M[rs2]

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

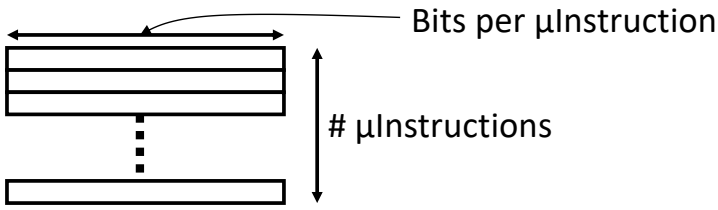
Very difficult to implement these instructions using a hardwired controller without substantial datapath modifications

Single-Bus Datapath for Microcoded RISC-V



Datapath unchanged for complex instructions!

Horizontal vs Vertical µCode

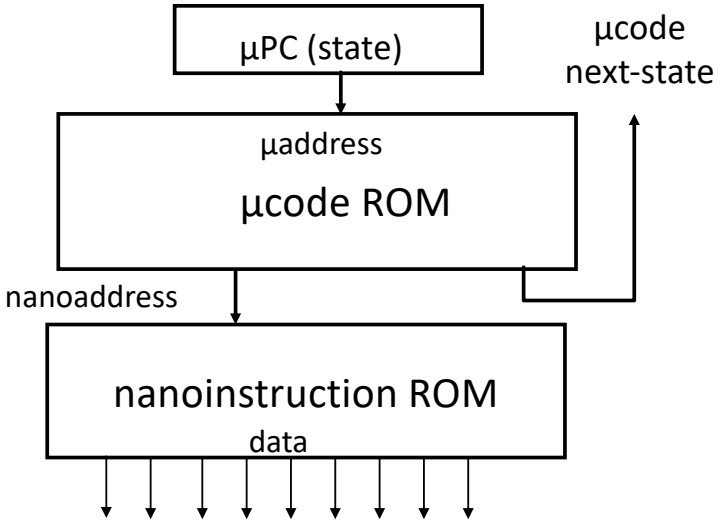


- Horizontal µcode has wider µinstructions
  - Multiple parallel operations per µinstruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical µcode has narrower µinstructions
  - Typically a single datapath operation per µinstruction
    - separate µinstruction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



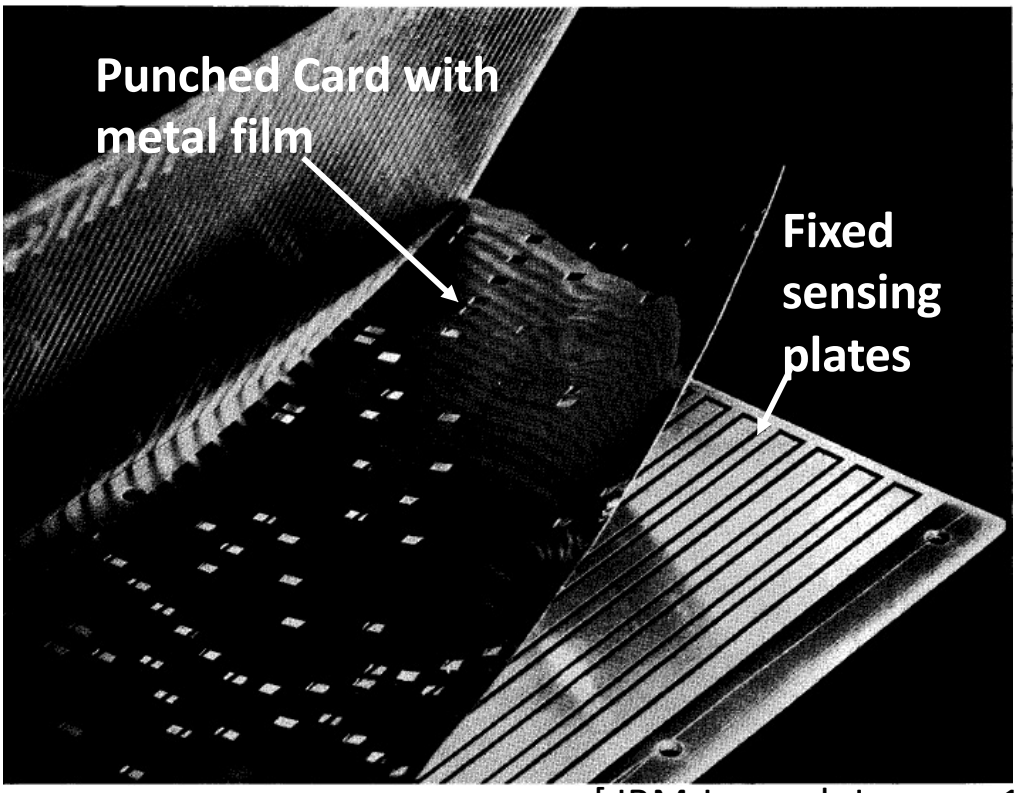
- Motorola 68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

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

IBM Card-Capacitor Read-Only Storage



[IBM Journal, January 1961]

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
  - i.e., 650 simulated on 1401 emulated on 360

Microprogramming thrived in ‘60s and ‘70s

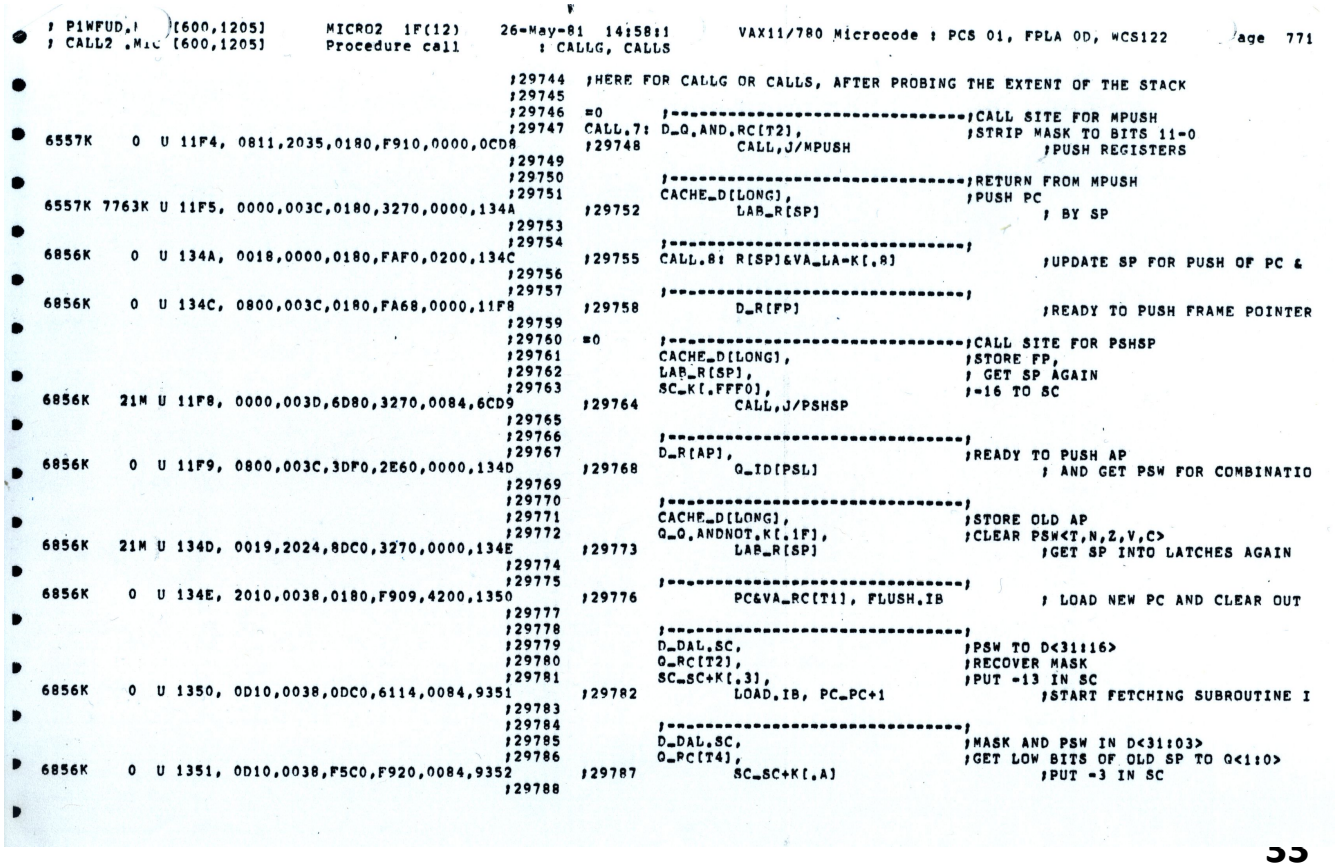
- Significantly faster ROMs than DRAMs were available
- For complex instruction sets, datapath and controller were cheaper and simpler
- New instructions , e.g., floating point, could be supported without datapath modifications
- Fixing bugs in the controller was easier
- ISA compatibility across various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

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

Microprogramming: early 1980s

- 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

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  =0      ;-------------------------------;CALL SITE FOR MPUSH
                                                ;29747  CALL,7: D_Q,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        LAB_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  =0      ;-------------------------------;CALL SITE FOR PSHSP
                                                ;29761          CACHE_D[LONG],                  ;STORE FP,
                                                ;29762          LAB_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        Q_ID[PSL]                               ; AND GET PSW FOR COMBINATIO
                                                ;29769
                                                ;29770          ;-------------------------------;
                                                ;29771          CACHE_D[LONG],                  ;STORE OLD AP
                                                ;29772          Q_Q,ANDNOT,K[,1F],              ;CLEAR PSW<T,N,Z,V,C>
6856K   21M U 134D, 0019,2024,8DC0,3270,0000,134E ;29773        LAB_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          Q_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          Q_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
```

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 is far from extinct

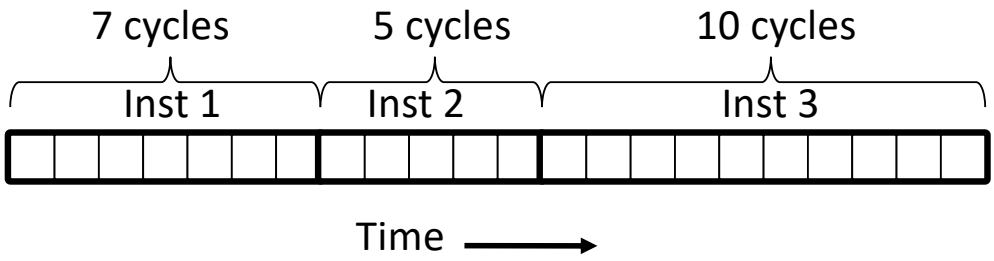
- Played a crucial role in micros of the Eighties
  - DEC uVAX, Motorola 68K series, Intel 286/386
- Plays an assisting role in most modern micros
  - e.g., AMD Zen, Intel Sky Lake, Intel Atom, IBM PowerPC, …
  - Most instructions executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke microcode
- Patchable microcode common for post-fabrication bug fixes, e.g. Intel processors load µcode patches at bootup
  - Intel had to scramble to resurrect microcode tools and find original microcode engineers to patch Meltdown/Spectre security vulnerabilites

"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

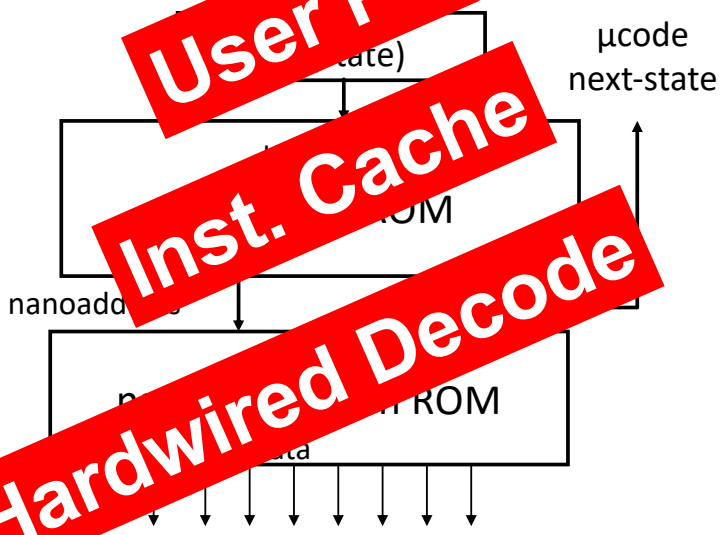
- Logic, RAM, ROM all implemented using MOS transistors
- Semiconductor RAM ~ same speed as ROM

Reconsidering Microcode Machine (Nanocoded 68000 example)

RISC!

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

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



User PC

Inst. Cache

Hardwired Decode

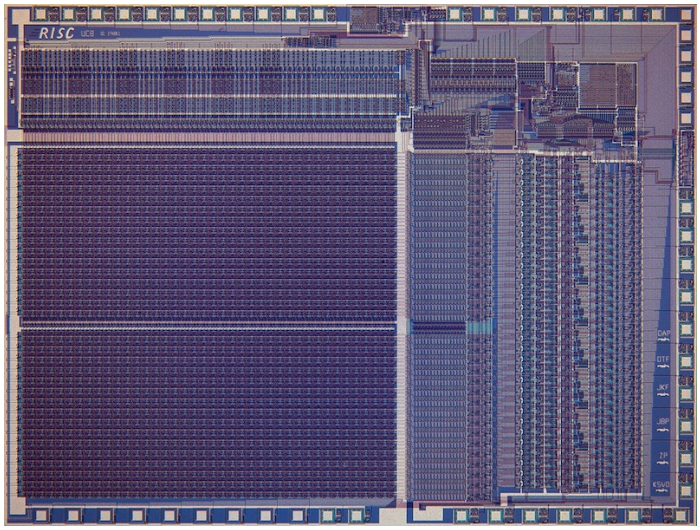
- Motorola 68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

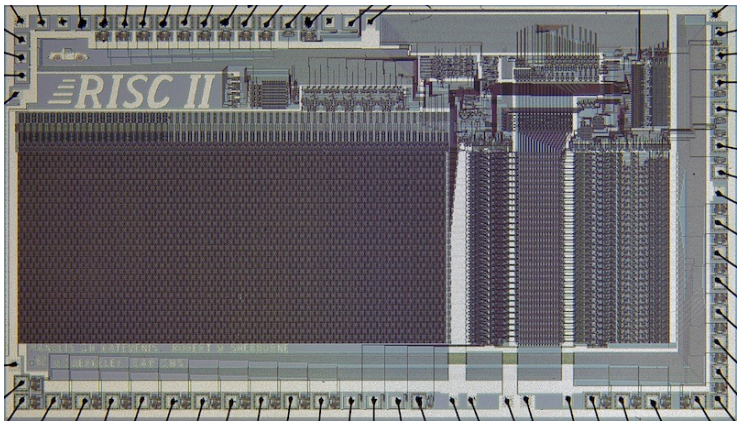
From CISC to RISC

- Use fast RAM to build fast instruction *cache* of user-visible instructions, not fixed hardware microroutines
  - Contents of fast instruction memory change to fit application needs
- Use simple ISA to enable hardwired pipelined implementation
  - Most compiled code only used few CISC instructions
  - Simpler encoding allowed pipelined implementations
  - RISC ISA comparable to vertical microcode
- Further benefit with integration
  - In early ‘80s, finally fit 32-bit datapath + small caches on 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…

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)
  - Krste Asanovic