CS 152/252A Computer Architecture and Engineering

Sophia Shao

Lecture 3 – Microcoding

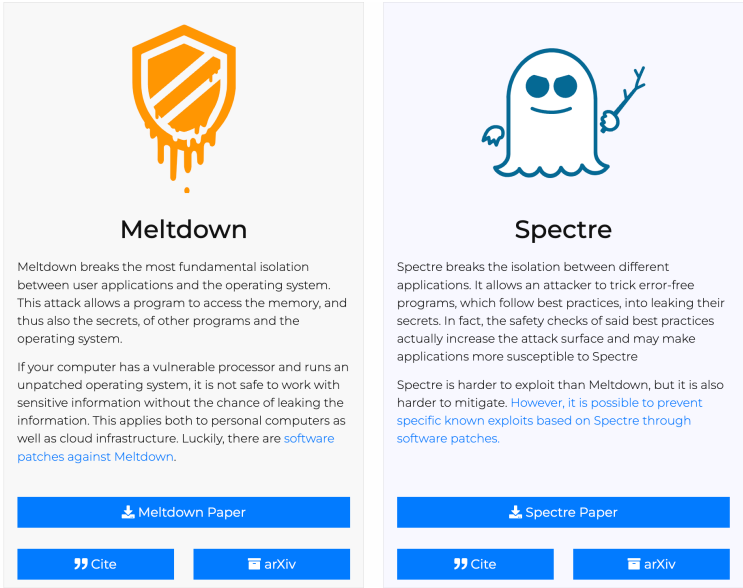
Intel's new Spectre fix: Skylake, Kaby Lake, Coffee Lake chips get stable microcode

Intel makes progress on reissuing stable microcode updates against the Spectre attack.

Customers running machines with newer Intel chips can expect to receive stable firmware updates for the Spectre CPU attack Variant 2 soon.

Intel says it has given PC makers a new set of microcode updates that mitigate the branch target injection Spectre attack on its 6th, 7th, and 8th generation Intel Core chips.

https://www.zdnet.com/article/intels-new-spectre-fix-skylake-kaby-lake-coffee-lake-chips-get-stable-microcode/

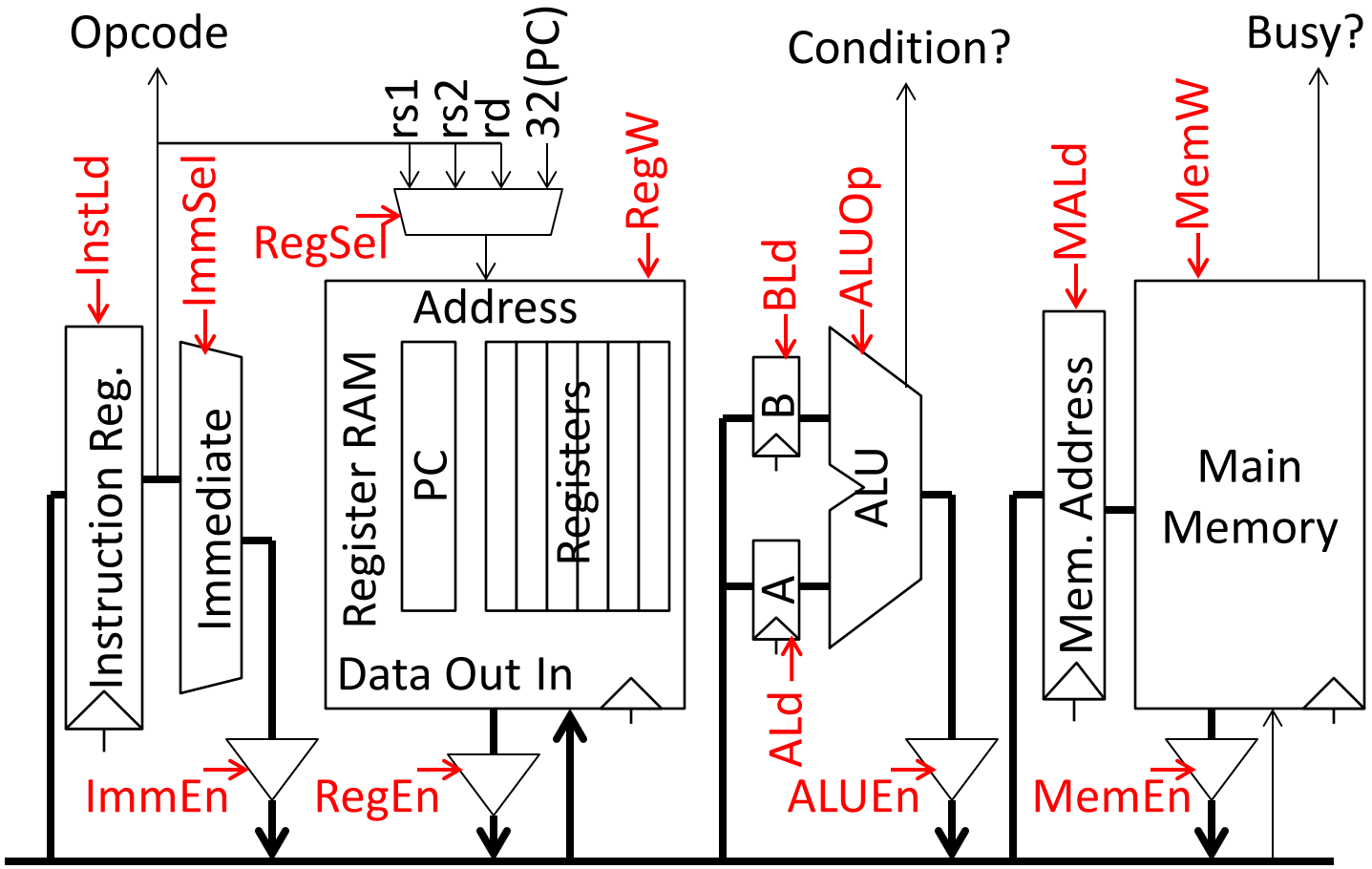


https://meltdownattack.com/

Last Time in Lecture 2

- Microcoding, an effective technique to manage control unit complexity, invented in era when logic (tubes), main memory (magnetic core), and ROM (diodes) used different technologies
- Difference between ROM and RAM speed motivated additional complex instructions
- Technology advances leading to fast SRAM made technology assumptions invalid

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

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

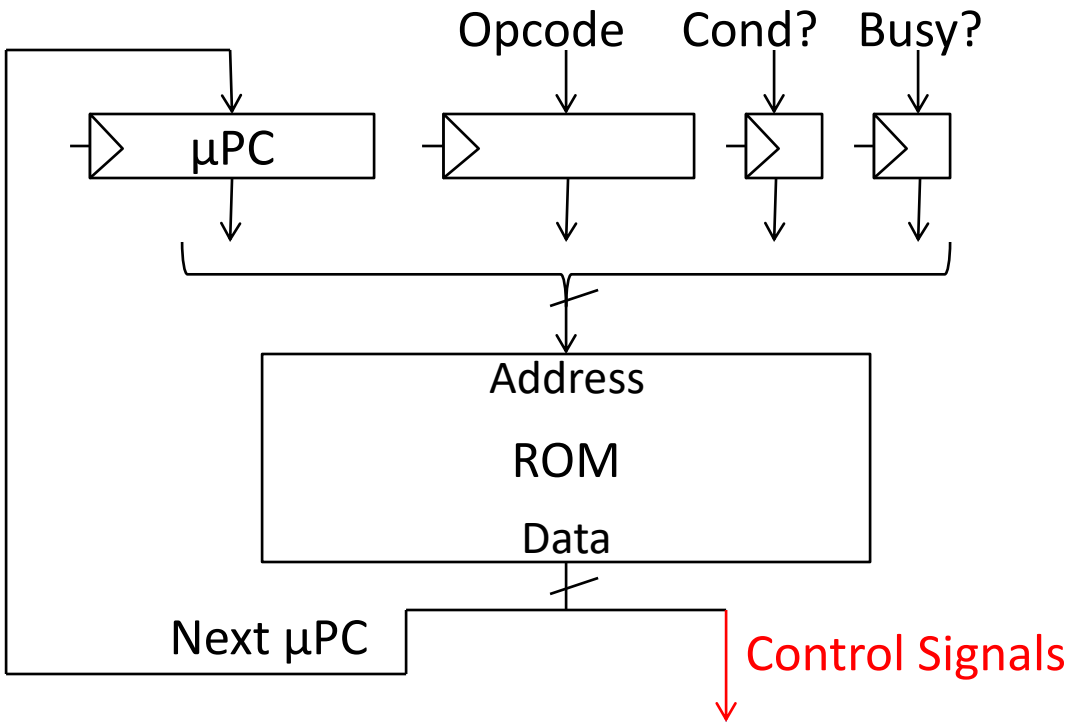
JAL:

Reg[rd]:=A // Store return address (A:=PC from fetch)
A:=A-4 // Recover PC (PC incremented in fetch)
B:=ImmJ // Jump-style immediate
PC:=A+B // (Alternative: PC:=A+B-4)
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? (index to the ROM)

  $|\mu address| = |\mu PC| + |opcode| + 1 + 1$

- How many data bits? (width of the ROM)

  $|data| = |\mu PC| + |control\ signals| = |\mu PC| + 18$

- Total ROM size = $2^{|\mu address|} \times |data|$

Pure ROM Contents (Truth Table)

Address (Inputs)				Data (Outputs)	
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

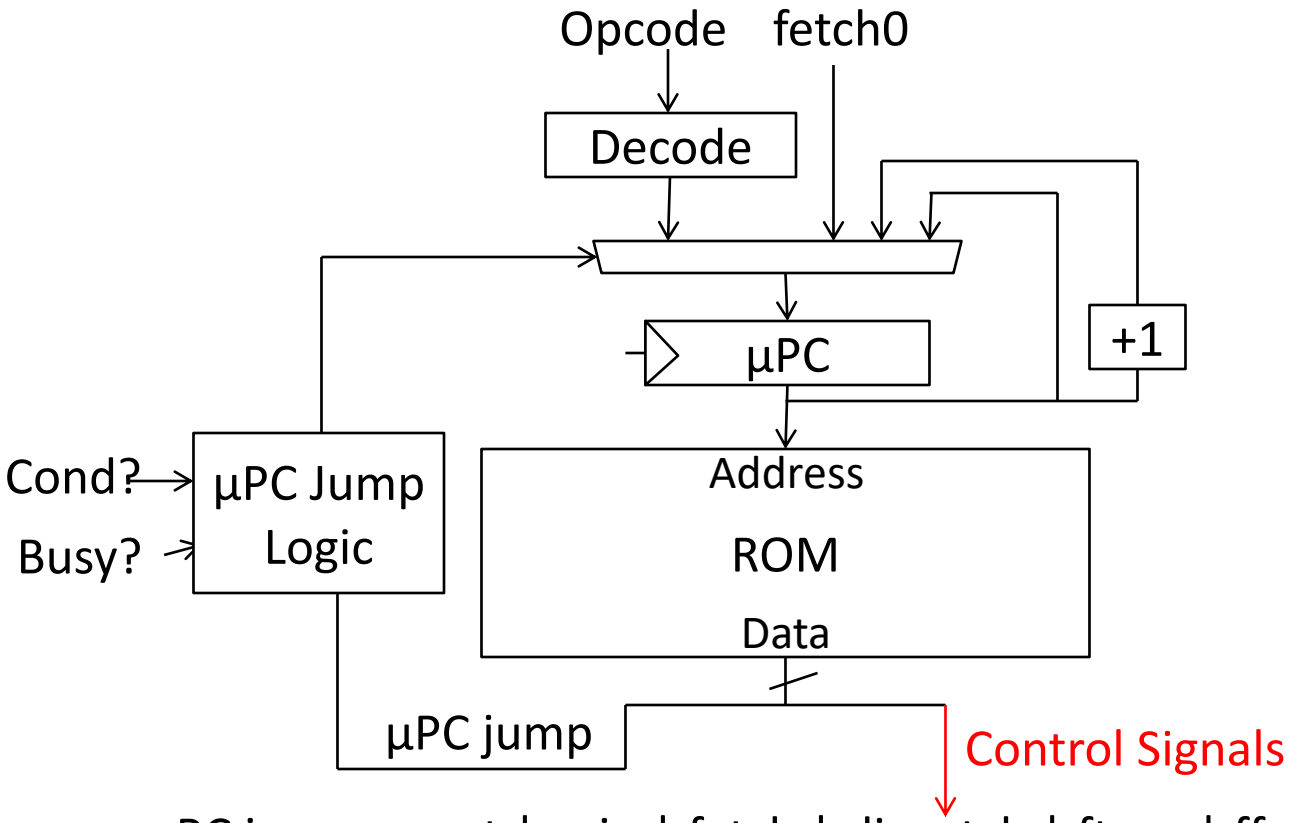
- Opcode is 5 bits, ~18 control signals

- 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

- *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	µPC Jump
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 (not taken)
Branch3	A:=A-4	next (taken)
Branch4	B:=ImmB	next
Branch5	PC:=A+B	fetch

Implementing Complex Instructions

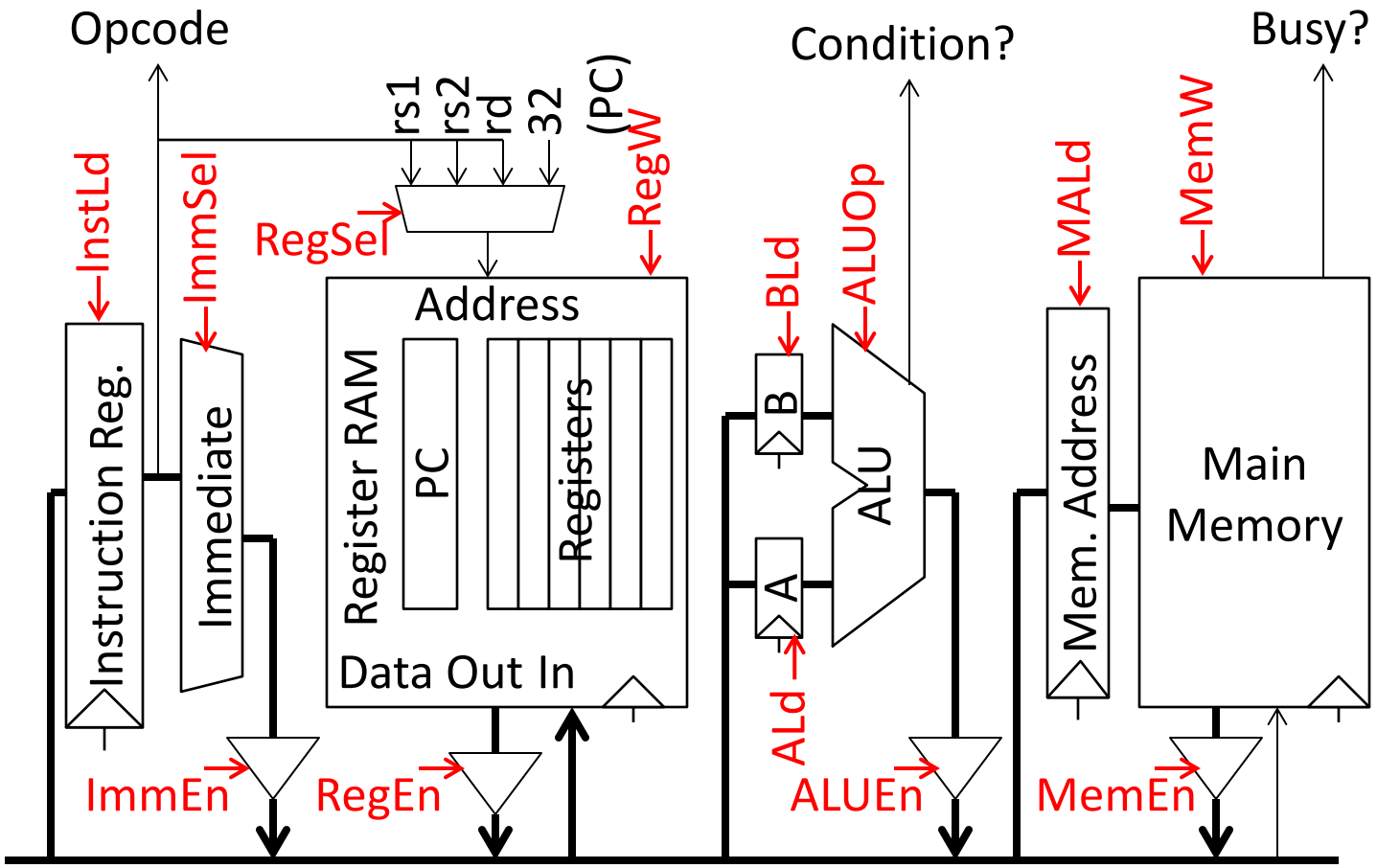
Memory-memory add: M[rd] = M[rs1] + M[rs2]

Address	Data	
µPC	Control Lines	µPC Jump
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

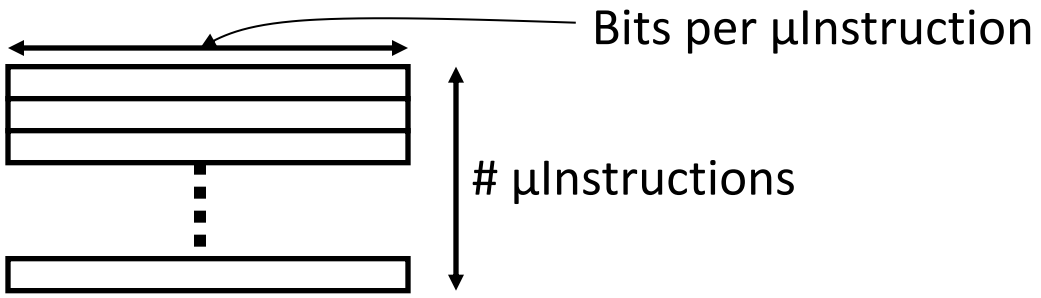
Very difficult to implement these instructions using a hardwired controller without substantial datapath modifications

Single-Bus Datapath for Microcoded RISC-V



Datapath unchanged for complex instructions!

Horizontal vs Vertical µCode

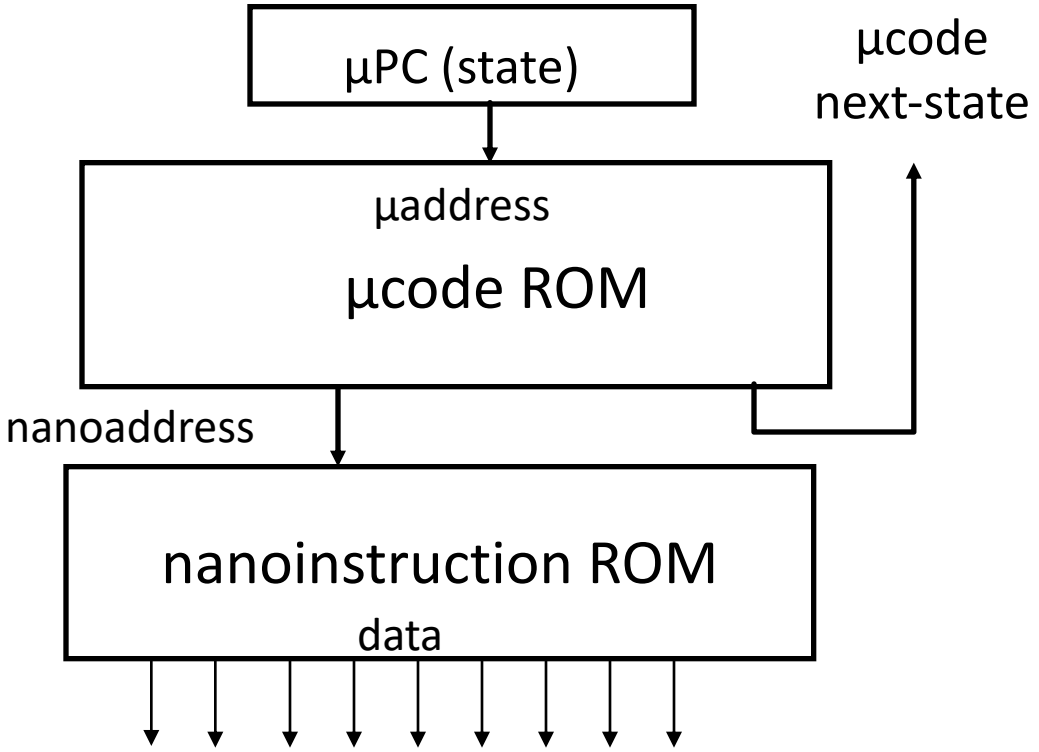


- Horizontal µcode has wider µinstructions
  - Multiple parallel operations per µinstruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical µcode has narrower µinstructions
  - Typically a single datapath operation per µinstruction
    - separate µinstruction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



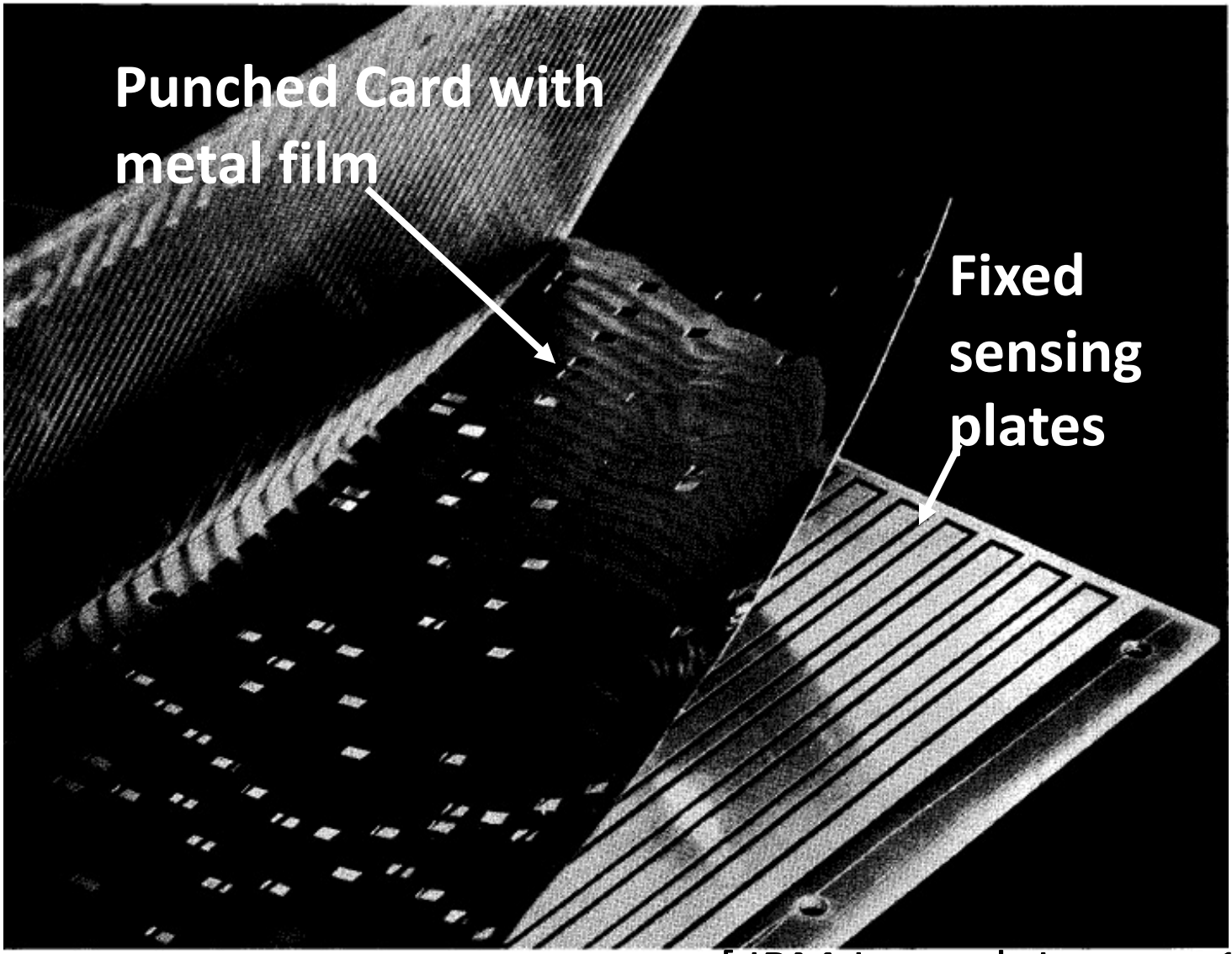
- Motorola 68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

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]

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

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

CS152 Administrivia

- HW1 released
  - Due Feb 02
- Lab group matching
  - https://docs.google.com/forms/d/e/1FAIpQLSfxXVEwM6a-pR2-RU_ntjD1zfskipOWf4e-8eCIjxfOHtTiaA/viewform?usp=sf_link
- Grading clarifications
  - You must complete 3/5 labs or get an automatic F regardless of other grades
- Slip days
  - 7 slip days for labs and PS
  - Submit extension requests if needed.
- Discussions and OHs start this week.
  - Check the course calendar for details.
  - Wednesday 10am-12pm discussion dropped.
  - Discussions will be recorded.

CS252 Administrivia

- CS252 Readings on Website
  - Use hotcrp to upload reviews before Wednesday:
    - Write one paragraph on main content of paper including good/bad points of paper
    - Also, answer/ask 1-3 questions about paper for discussion
    - First two “360 Architecture”, “VAX11-780”
  - 2-3pm Wednesday, Soda 606/Zoom
- CS252 Project Timeline
  - Proposal Wed Feb 22
  - One page in PDF format including:
    - project title
    - team members (2 per project)
    - what problem are you trying to solve?
    - what is your approach?
    - infrastructure to be used
    - timeline/milestones

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!

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!

User PC

Inst. Cache

Hardwired Decode

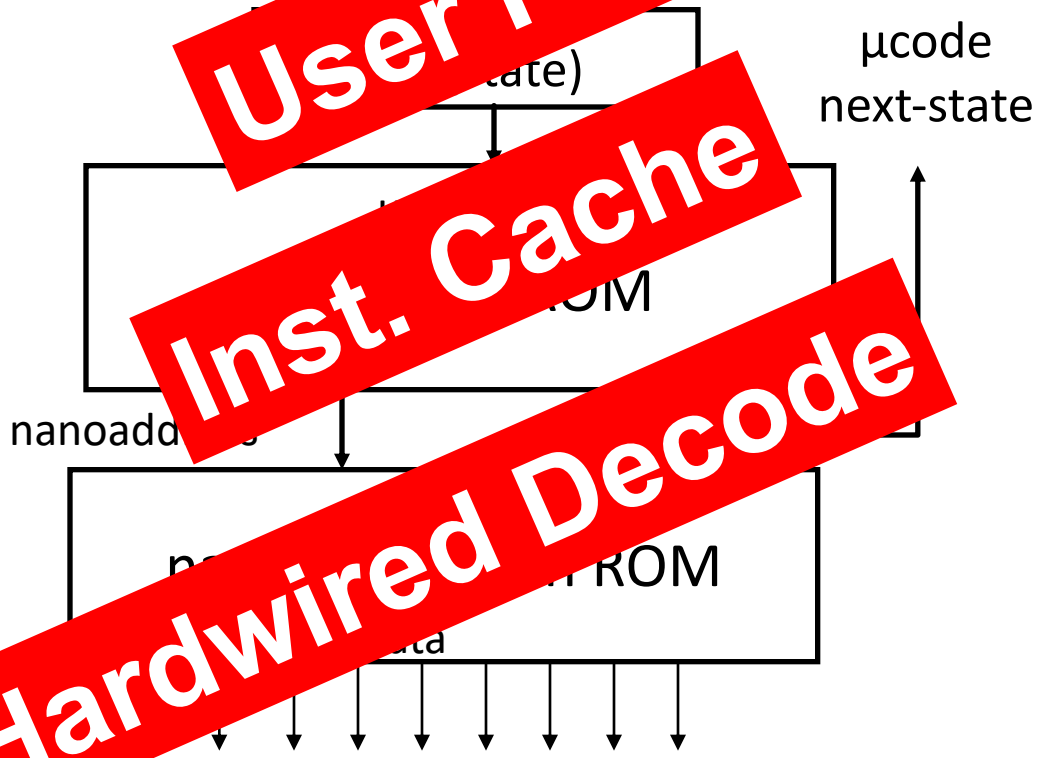
Exploits recurring control signal patterns in µcode, e.g.,

ALU0 A ← Reg[rs1]

...

ALUI0 A ← Reg[rs1]

...



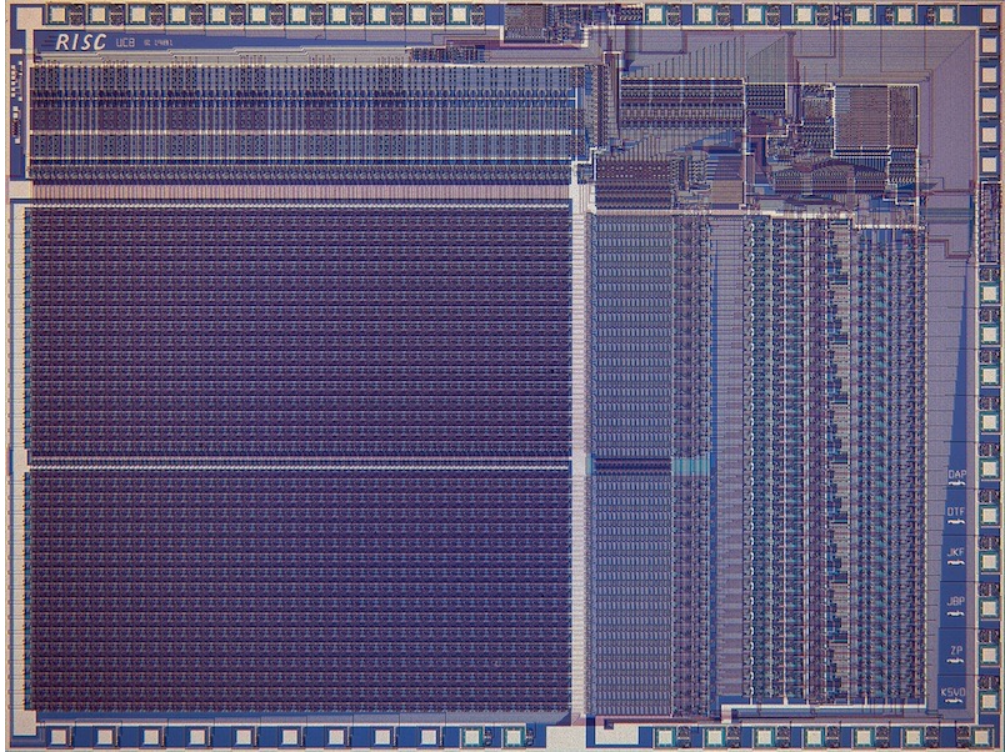
- Motorola 68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

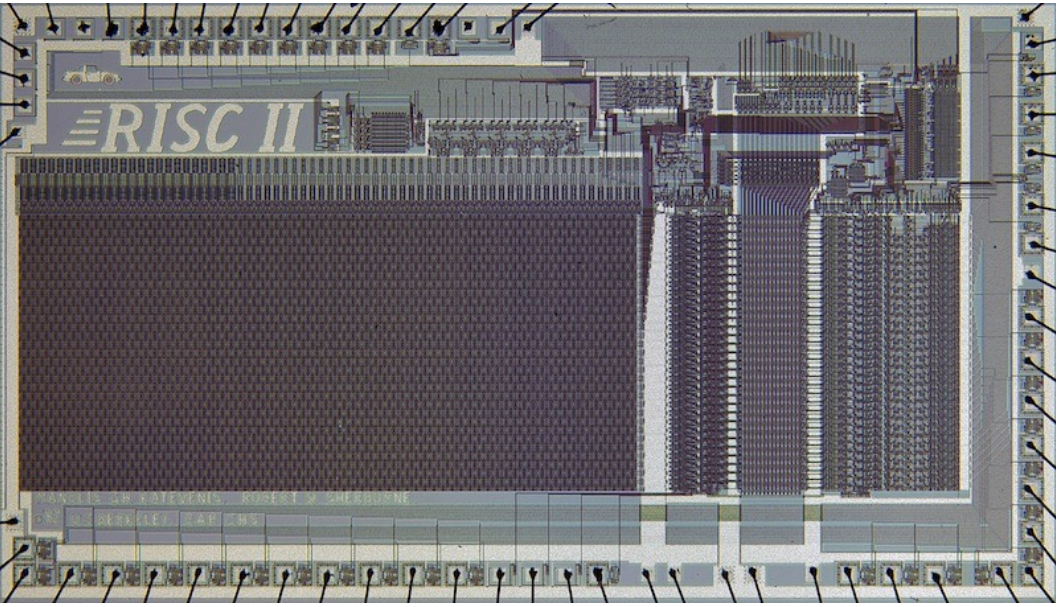
From CISC to RISC

- 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 mm^2, 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 mm^2.

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)

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