

# **CS 152 Computer Architecture and Engineering**

## **CS252 Graduate Computer Architecture**

### **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 Architecture >> ISAs and RTL
  - CS152 is about interaction of hardware and software, and design of appropriate abstraction layers
- 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.
- We use RISC-V as standard ISA in class ([www.riscv.org](http://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  $\Rightarrow$  hardwired, unpipelined

CISC  $\Rightarrow$  microcoded

RISC  $\Rightarrow$  hardwired, pipelined

VLIW  $\Rightarrow$  fixed-latency in-order parallel pipelines

JVM  $\Rightarrow$  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**

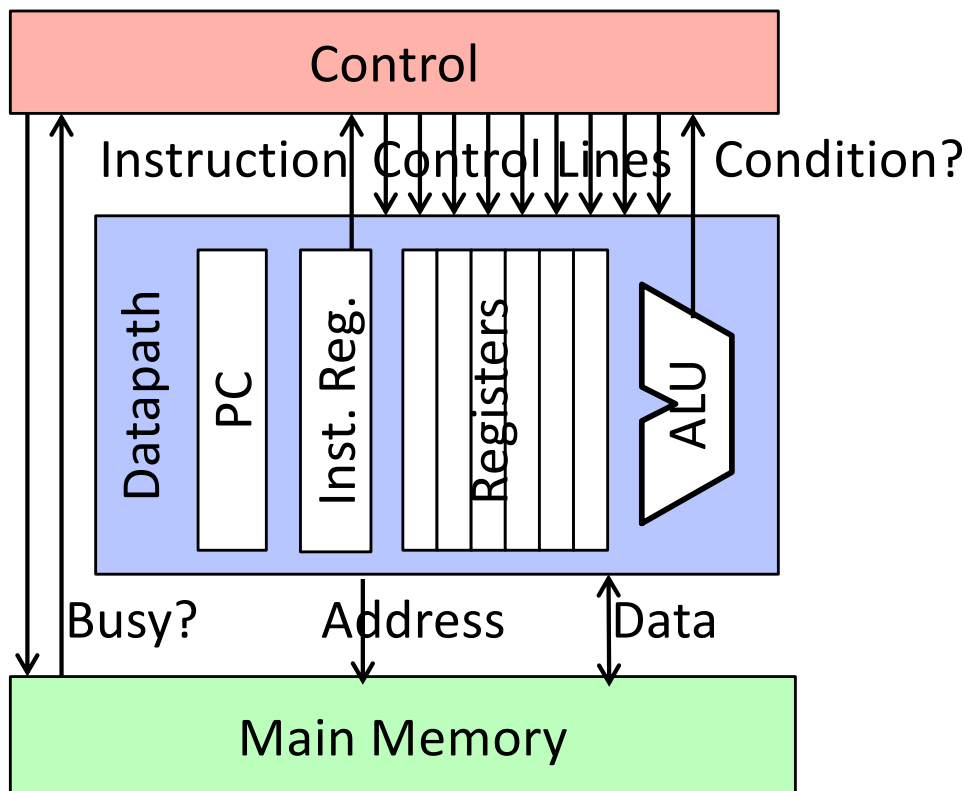
# Why Learn Microprogramming?

- 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.*

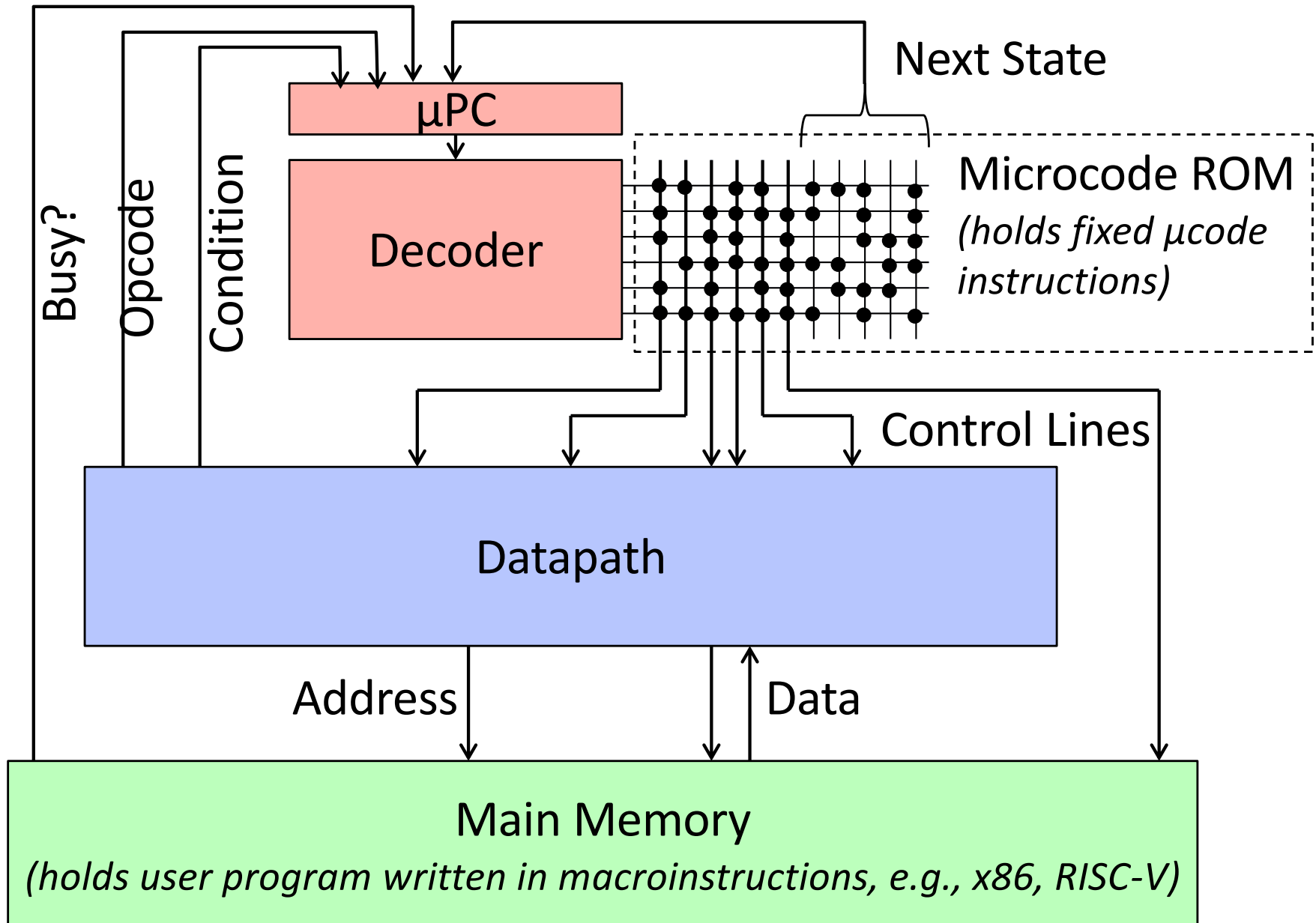
# Control versus Datapath

- 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
  - Foreshadowed by Babbage's "Barrel" and mechanisms in earlier programmable calculators

# 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
- 
- Please see CS61C Fall 2017, Lectures 5-7 for RISC-V ISA review:  
**<http://inst.eecs.berkeley.edu/~cs61c/fa17/>**

# RV32I Processor State

Program counter (**pc**)

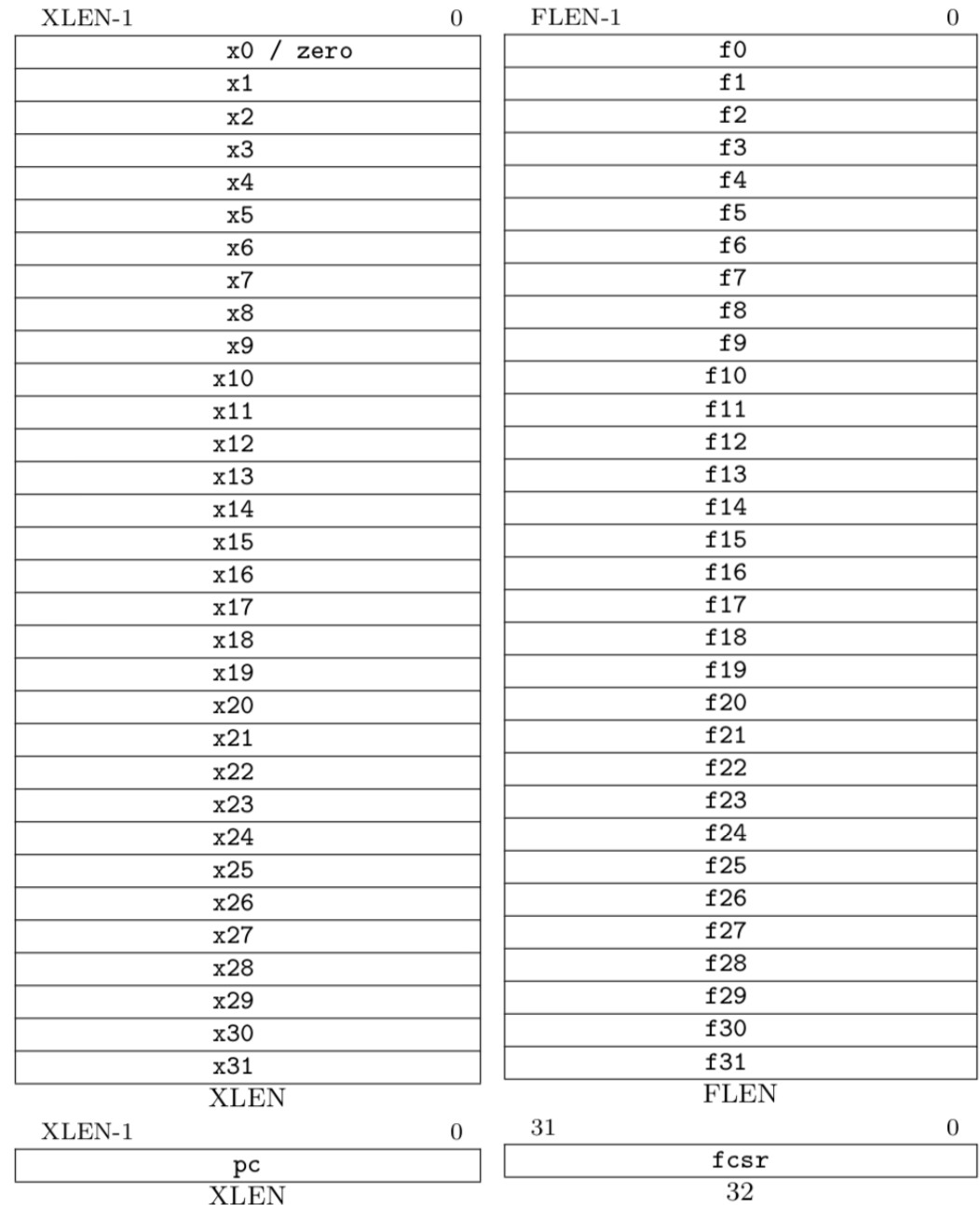
32x32-bit integer registers (**x0-x31**)

- **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

xxxxxxxxxxxxxxxxaa 16-bit (aa  $\neq$  11)

xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxbbb11 32-bit (bbb  $\neq$  111)

· · ·xxxx xxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxx011111 48-bit

· · ·xxxx xxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxx011111 64-bit

· · ·xxxx xxxxxxxxxxxxxxxxxxxxxx xnnnxxxxx111111 (80+16\*nnn)-bit, nnn  $\neq$  111

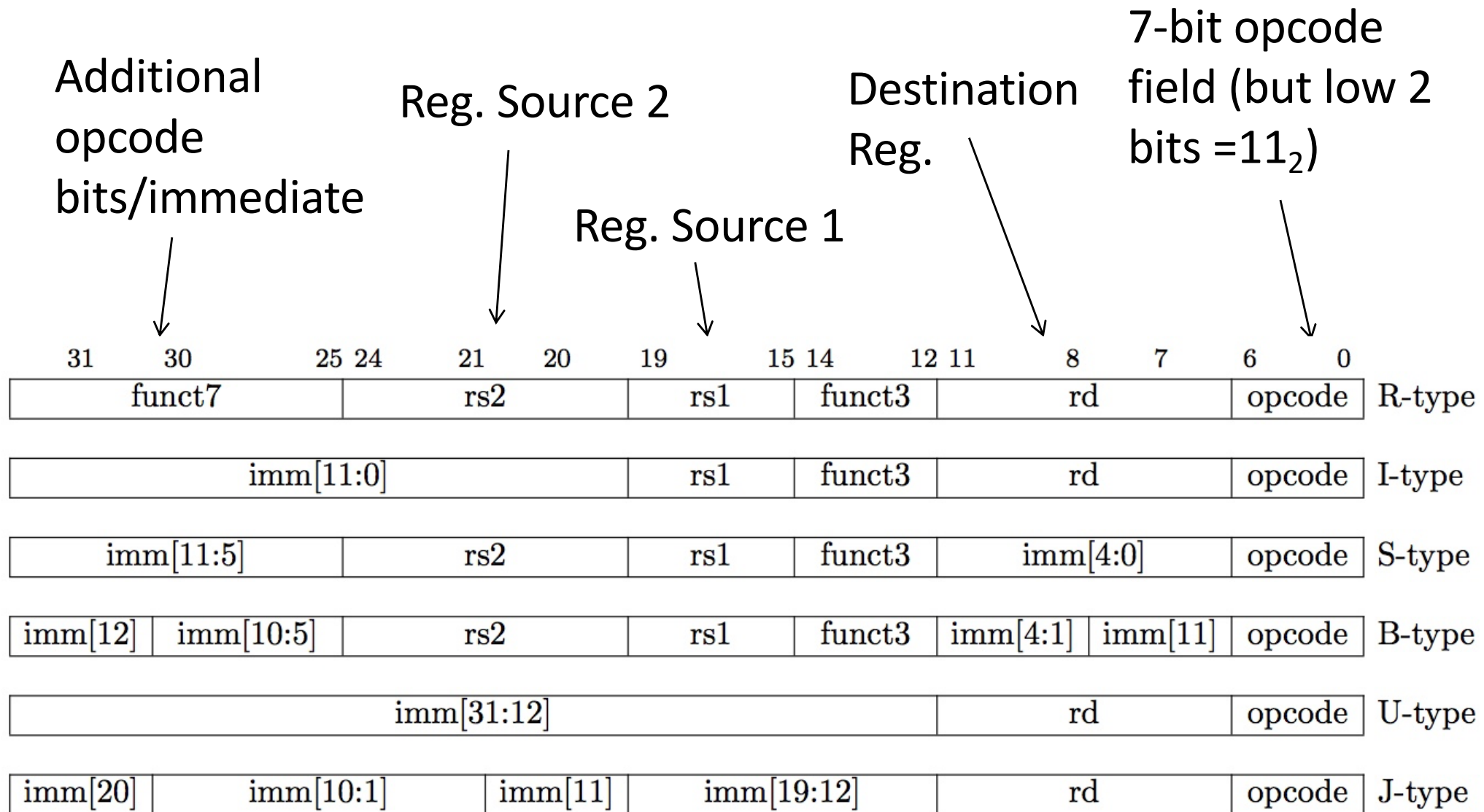
· · ·xxxx xxxxxxxxxxxxxxxxxxxxxx x111xxxxx111111 Reserved for  $\geq 192$ -bits

>32b not  
ratified

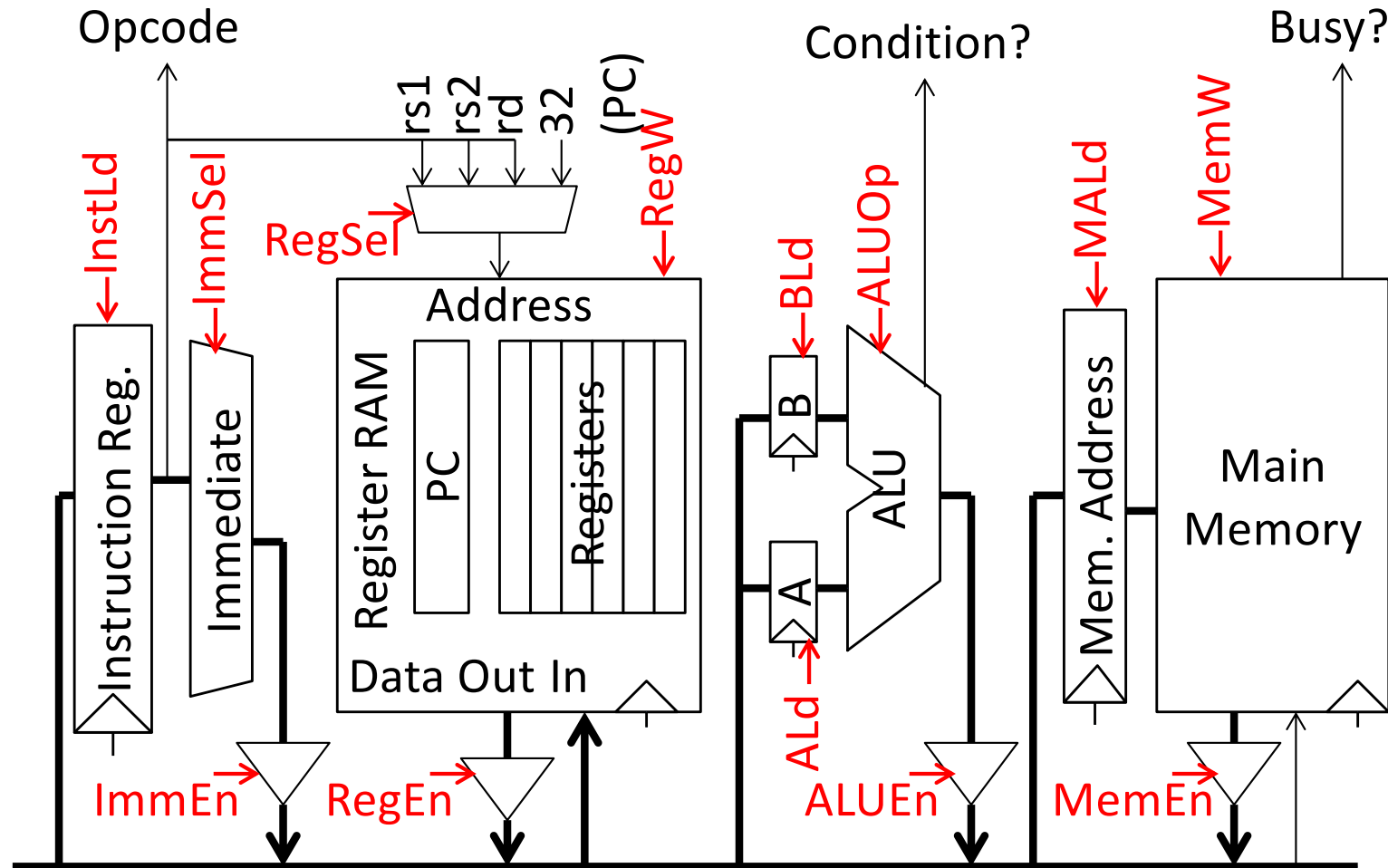
Byte Address:      base+4                                  base+2                                  base

- 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)

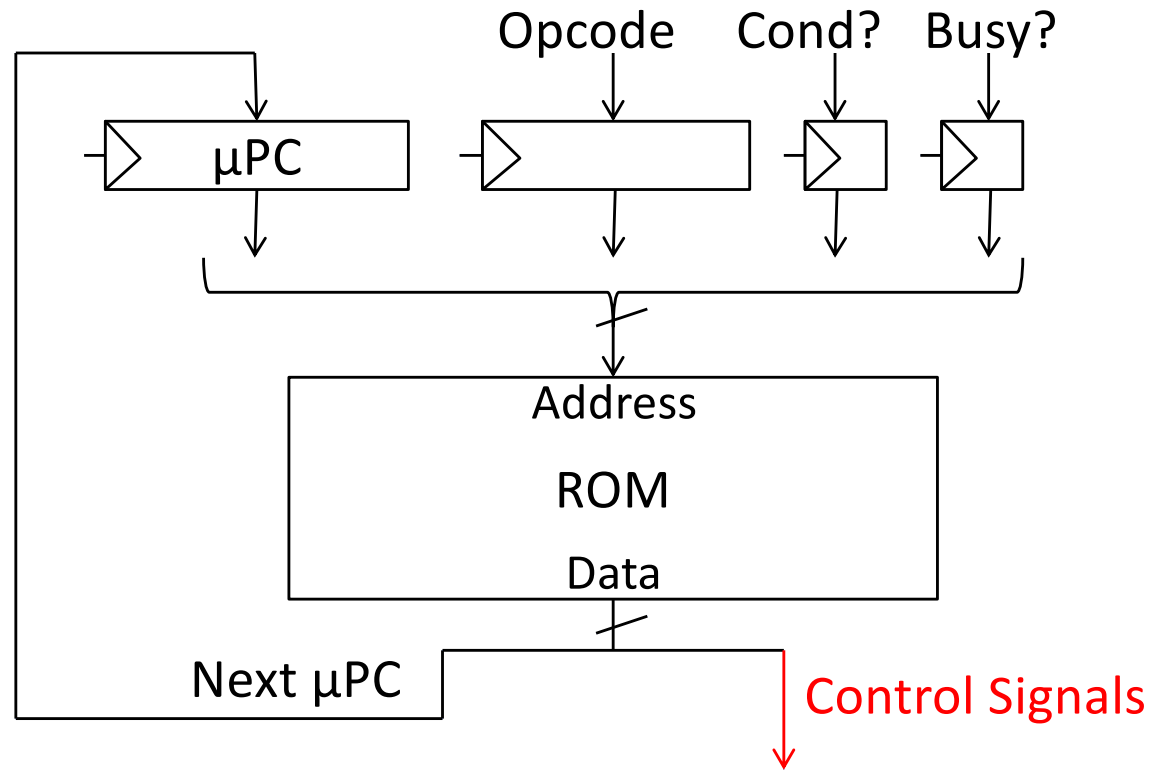
LW:                   A:=Reg[rs1]  
                      B:=ImmI //Sign-extend 12b immediate  
                      MA:=A+B  
                      *wait for memory*  
                      Reg[rd]:=Mem  
                      *goto instruction fetch*

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?

$$|\mu\text{address}| = |\mu\text{PC}| + |\text{opcode}| + 1 + 1$$

- How many data bits?

$$|\text{data}| = |\mu\text{PC}| + |\text{control signals}| = |\mu\text{PC}| + 18$$

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

## 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

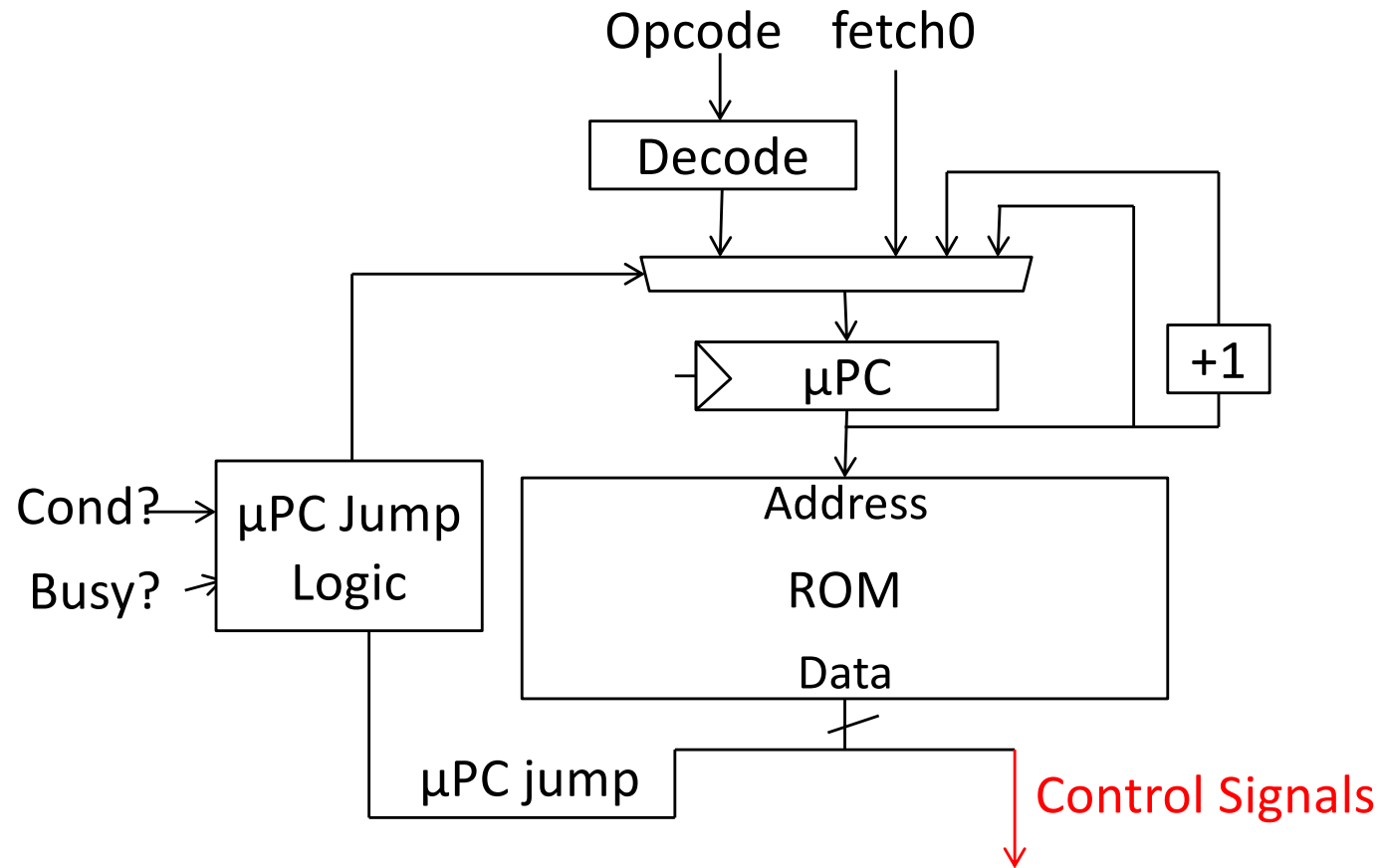
# Single-Bus Microcode RISC-V ROM Size

- 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  $\mu$ PC
- Opcode is 5 bits, ~18 control signals
- Total size =  $2^{(6+5+2)} \times (6+18) = 2^{13} \times 24 = \sim 25\text{KiB!}$

# 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  $\mu$ PC encoding (next,dispatch,wait on memory,...)
  - Encode control signals (vertical  $\mu$ coding, nanocoding)

# Single-Bus RISC-V Microcode Engine



$\mu\text{PC jump} = \text{next} \mid \text{spin} \mid \text{fetch} \mid \text{dispatch} \mid \text{ftrue} \mid \text{ffalse}$

# 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

# CS152 Administrivia

- Grading clarifications

- You must complete 3/5 labs or get an automatic F regardless of other grades

- Slip days

- Problem sets have no slip days
- Labs have two free extensions (max one per lab) until next class after due date
- No other extensions without documented issue



# CS252 Administrivia

## ■ CS252 Readings on Website

- Must use Piazza to send private note on each per paper thread to instructors before midnight Wednesday before Thursday discussion containing paper report:
  - Write one paragraph on main content of paper including good/bad points of paper
  - Also, 1-3 questions about paper for discussion
  - First two “360 Architecture”, “B5000 Architecture”

## ■ CS252 Project Timeline

- Proposal due start of class Wed Feb 24th
- 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

# Implementing Complex Instructions

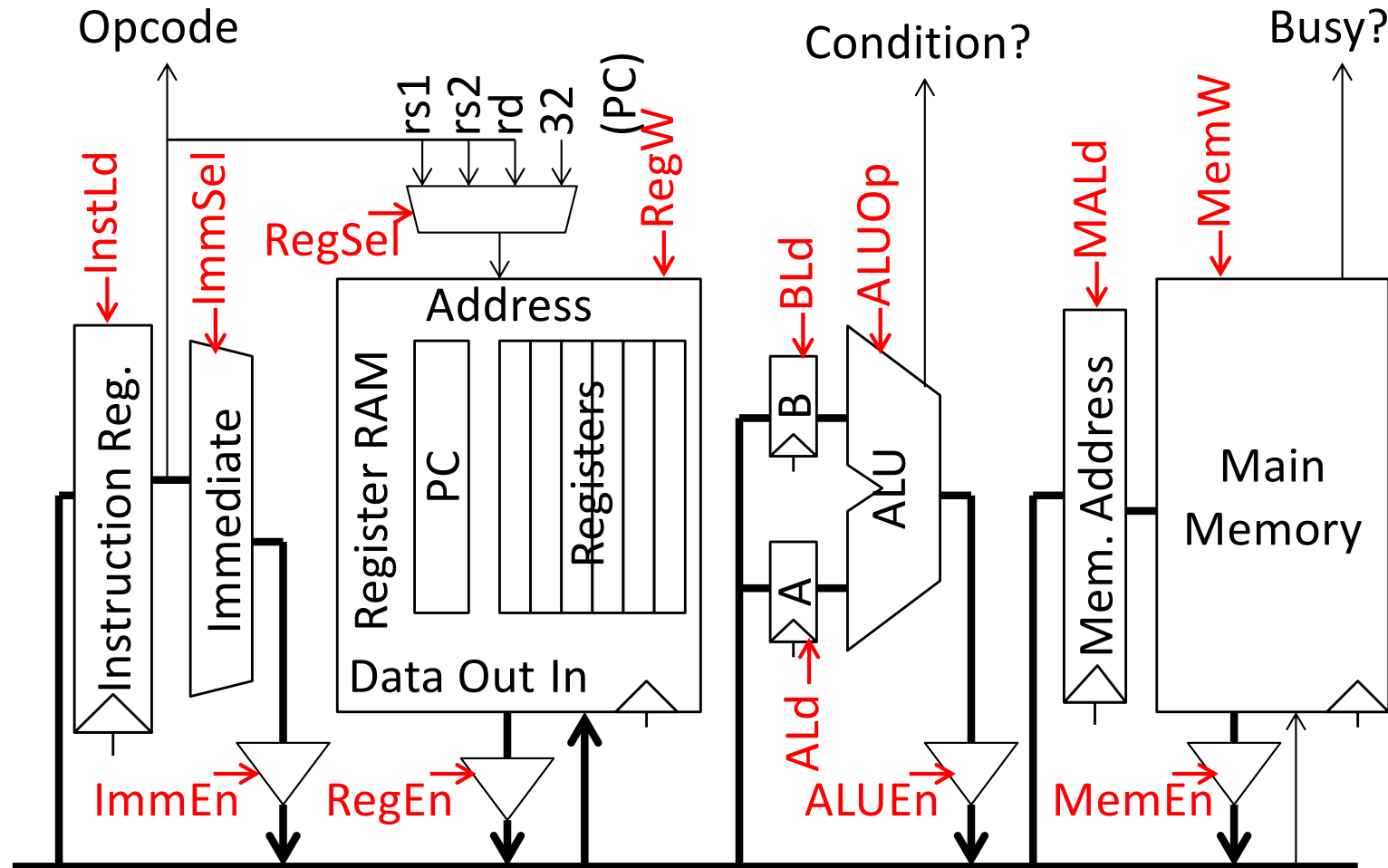
Memory-memory add:  $M[rd] = M[rs1] + M[rs2]$

Address	Data	
$\mu PC$	Control Lines	Next $\mu PC$
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

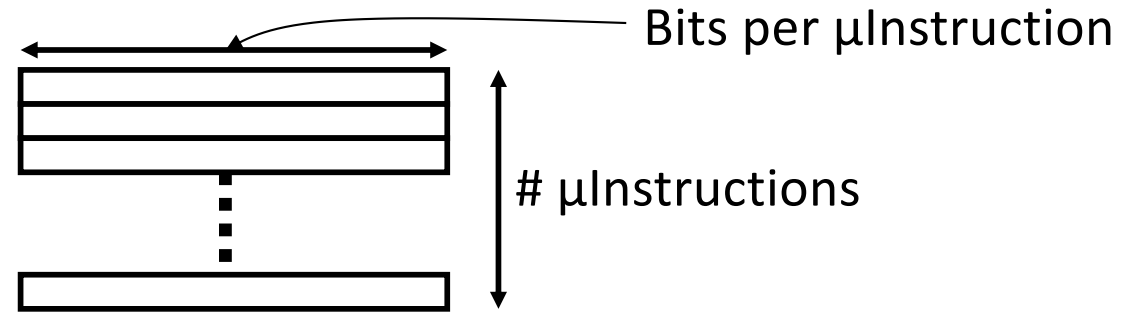
Very difficult to implement these instructions using a hardwired controller without substantial datapath modifications

# Single-Bus Datapath for Microcoded RISC-V



Datapath unchanged for complex instructions!

# Horizontal vs Vertical $\mu$ Code



- Horizontal  $\mu$ code has wider  $\mu$ instructions
  - Multiple parallel operations per  $\mu$ instruction
  - Fewer microcode steps per macroinstruction
  - Sparser encoding  $\Rightarrow$  more bits
- Vertical  $\mu$ code has narrower  $\mu$ instructions
  - Typically a single datapath operation per  $\mu$ instruction
    - separate  $\mu$ instruction for branches
  - More microcode steps per macroinstruction
  - More compact  $\Rightarrow$  less bits
- Nanocoding
  - Tries to combine best of horizontal and vertical  $\mu$ code

# Nanocoding

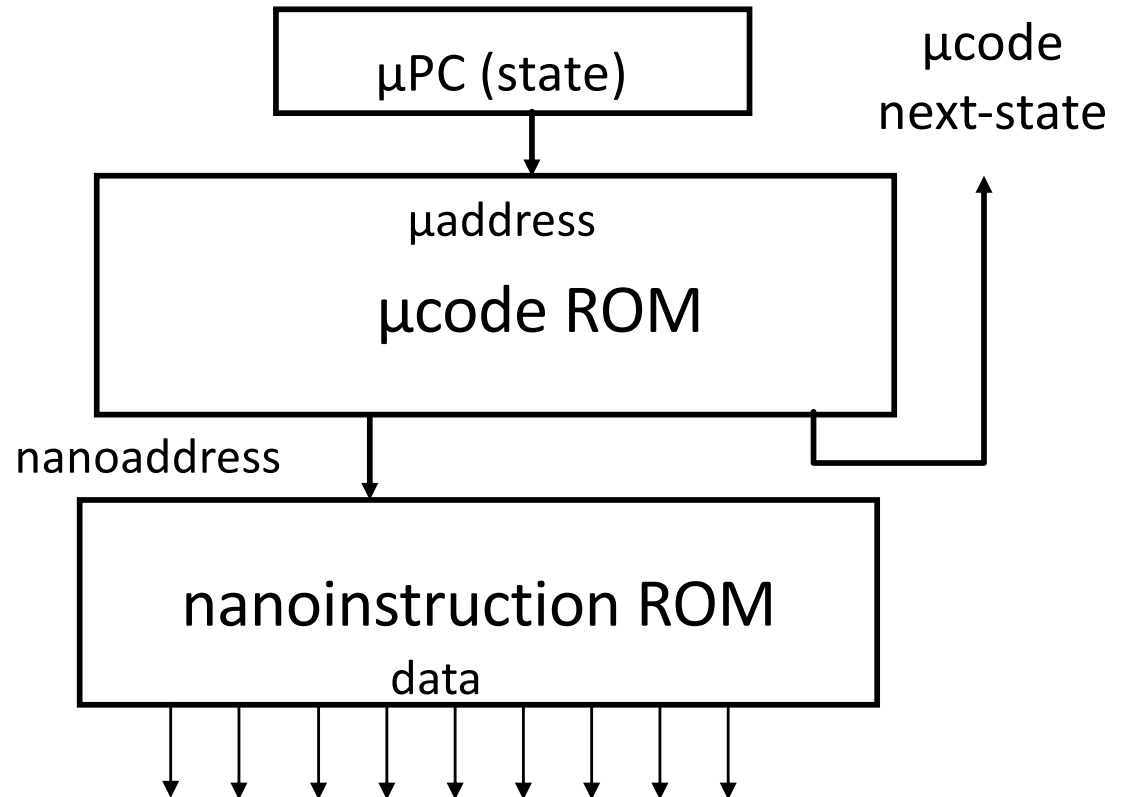
Exploits recurring control signal patterns in  $\mu$ code, e.g.,

ALU0  $A \leftarrow \text{Reg}[\text{rs1}]$

...

ALUI0  $A \leftarrow \text{Reg}[\text{rs1}]$

...



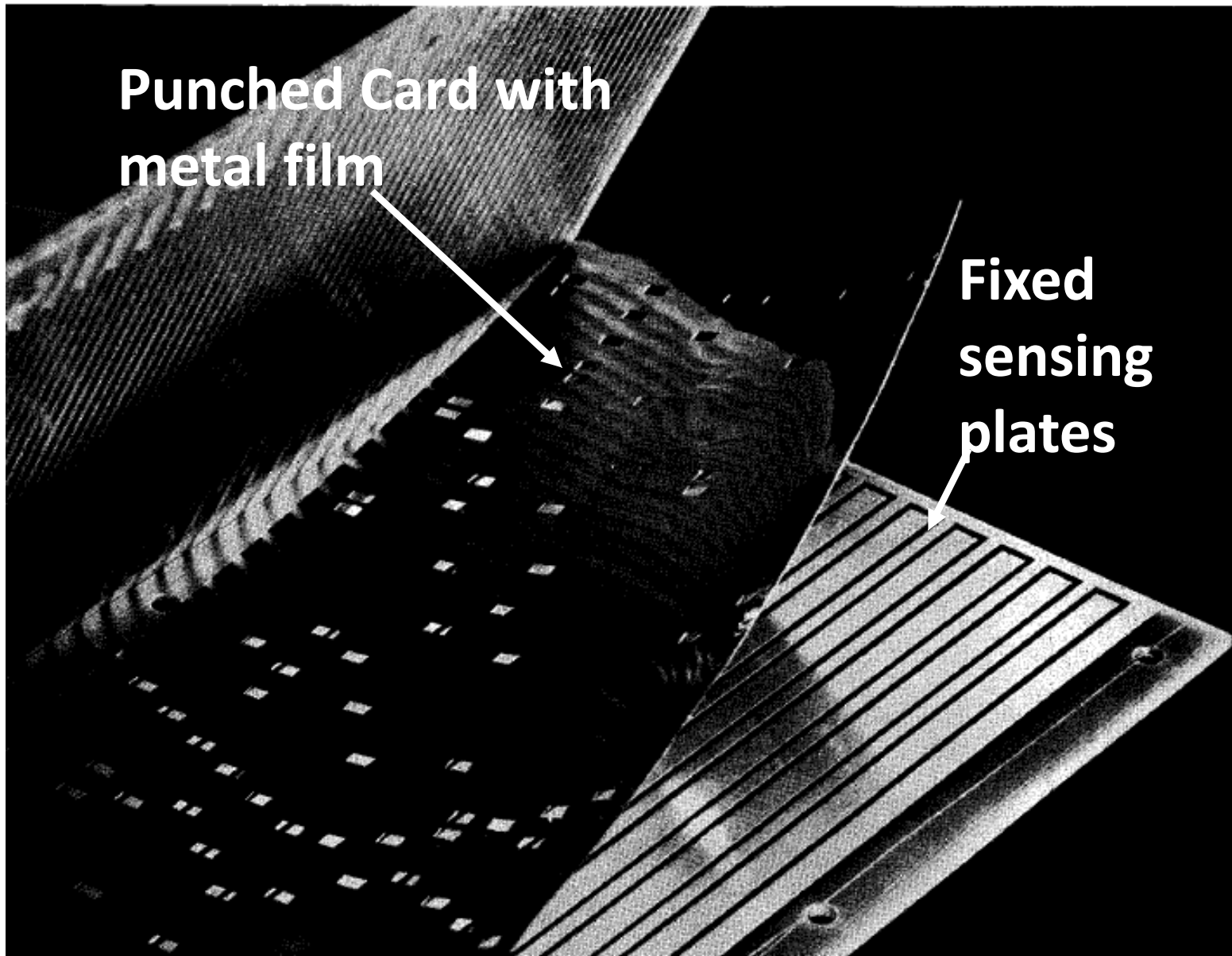
- Motorola 68000 had 17-bit  $\mu$ code containing either 10-bit  $\mu\text{jump}$  or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

## Microprogramming in IBM 360

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
$\mu$ inst width (bits)	50	52	85	87
$\mu$ code size (K $\mu$ insts)	4	4	2.75	2.75
$\mu$ store technology	CCROS	TCROS	BCROS	BCROS
$\mu$ store 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  $\mu$ code
  - Need for fixing bugs in control programs was in conflict with read-only nature of  $\mu$ ROM
  - ➔ 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,1 [600,1205]  
; CALL2 .MIC [600,1205]

MICRO2 1F(12)  
Procedure call

26-May-81 14:58:1  
; CALLG, CALLS

VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122

Page 771

```

;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
;29748 CALL,J/MPUSH ;PUSH REGISTERS
;29749
;29750 ;-----;RETURN FROM MPUSH
;29751 CACHE_D[LONG], ;PUSH PC
;29752 LAB_R[SP] ; BY SP
;29753
;29754 ;-----;
;29755 CALL.8: R[SP]&VA_LA-K[.8] ;UPDATE SP FOR PUSH OF PC &
;29756
;29757 ;-----;
;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
;29764 CALL,J/PSHSP
;29765
;29766 ;-----;
;29767 D_R[AP], ;READY TO PUSH AP
;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>
;29773 LAB_R[SP] ;GET SP INTO LATCHES AGAIN
;29774
;29775 ;-----;
;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
;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>
;29787 SC_SC+K[.A] ;PUT -3 IN SC
;29788

```

6557K 0 U 11F4, 0811,2035,0180,F910,0000,0CD8

6557K 7763K U 11F5, 0000,003C,0180,3270,0000,134A

6856K 0 U 134A, 0018,0000,0180,FAF0,0200,134C

6856K 0 U 134C, 0800,003C,0180,FA68,0000,11F8

6856K 21M U 11F8, 0000,003D,6D80,3270,0084,6CD9

6856K 0 U 11F9, 0800,003C,3DF0,2E60,0000,134D

6856K 21M U 134D, 0019,2024,8DC0,3270,0000,134E

6856K 0 U 134E, 2010,0038,0180,F909,4200,1350

6856K 0 U 1350, 0D10,0038,0DC0,6114,0084,9351

6856K 0 U 1351, 0D10,0038,F5C0,F920,0084,9352

# 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  $\mu$ code patches at bootup
  - Intel had to scramble to resurrect microcode tools and find original microcode engineers to patch Meltdown/Spectre security vulnerabilities

# 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