

# **18-447 Lecture 6: Microprogrammed Multi-Cycle Implementation**

James C. Hoe

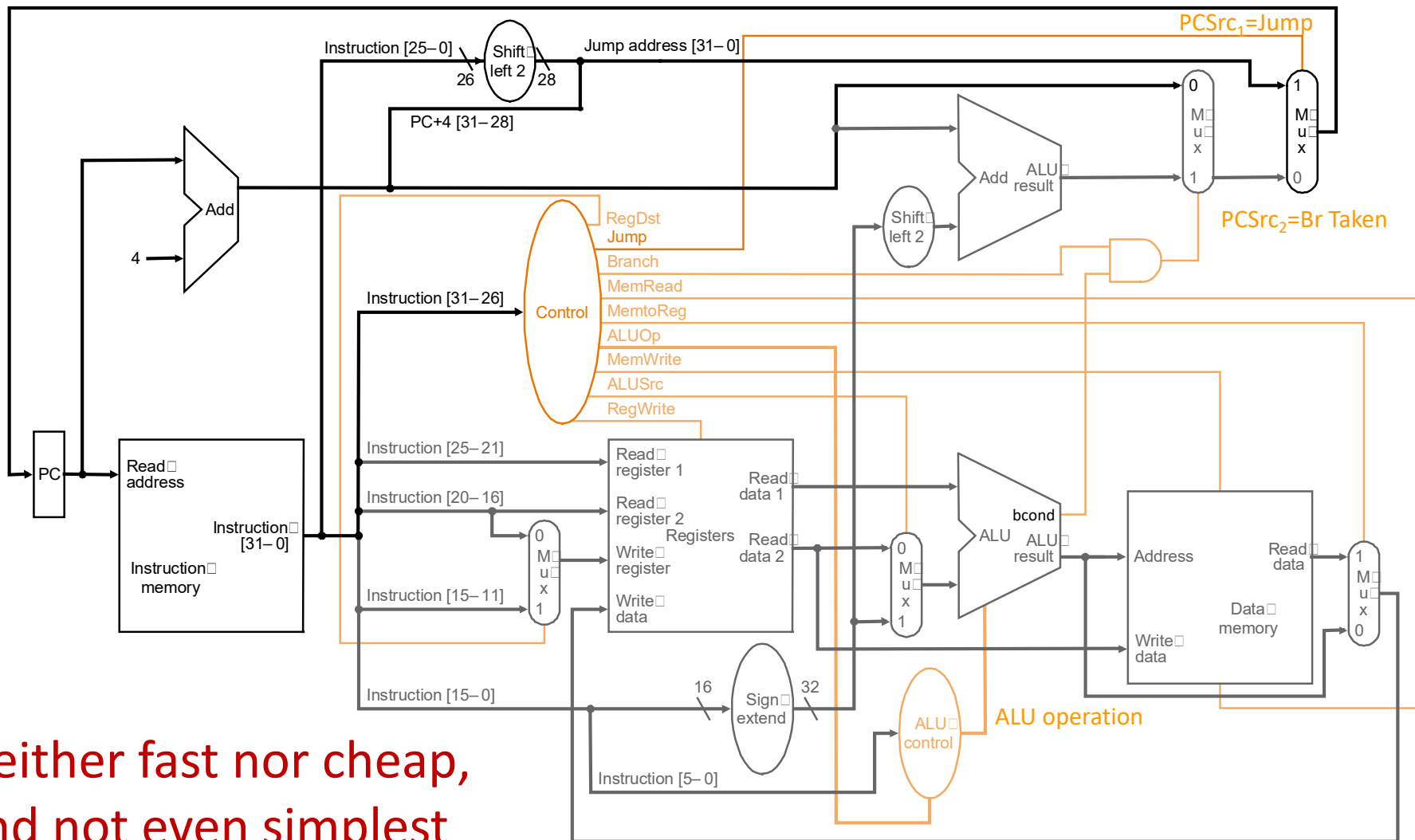
Department of ECE

Carnegie Mellon University

# Housekeeping

- Your goal today
  - understand why VAX was possible and reasonable
- Notices
  - Lab 1, Part B, **due this week**
  - HW1, **due Wed**
- Readings
  - P&H Appendix C
  - Start reading the rest of P&H Ch 4

# “Single-Cycle” Datapath: Is it any good?



Neither fast nor cheap,  
and not even simplest

**Go Fast(er)!!**

# Iron Law of Processor Performance

- wall clock time = (inst/program) (cyc/inst) (time/cyc)

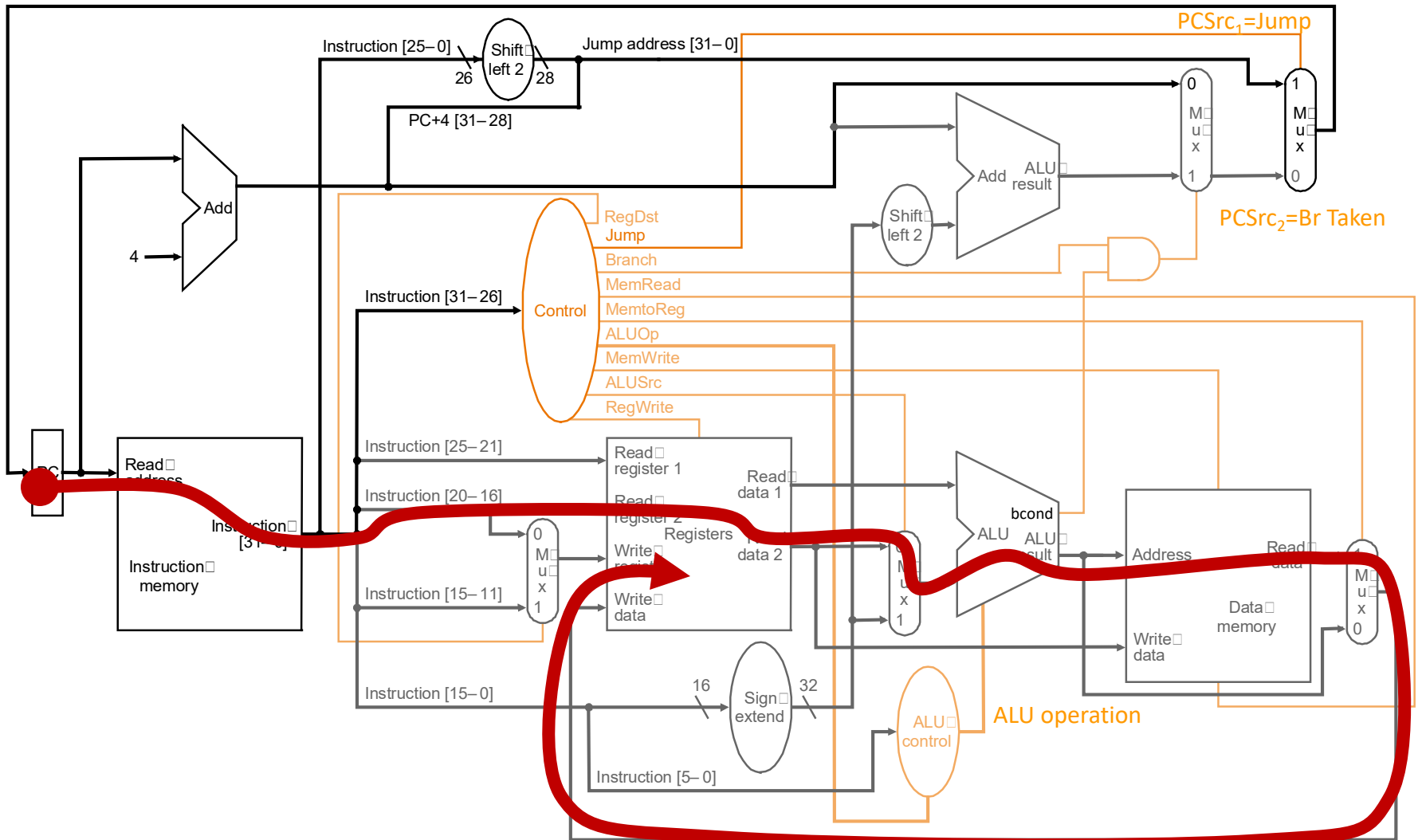


- Contributing factors
  - time/cyc: architecture and implementation
  - cyc/inst: architecture, implementation, instruction mix
  - inst/program: architecture, nature and quality of prgm
- **Note**: cyc/inst is a workload average

potentially large instantaneous variations  
due to instruction type and sequence

Recall

# Worst-Case Critical Path



# Single-Cycle Datapath Analysis

- Assume (numbers from P&H)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Bxx/JALR	200	50	100		<del>50</del>	350
JAL	200		100		<del>50</del>	300

# Single-Cycle Implementations

- Good match for the sequential and atomic semantics of ISAs
  - instantiate programmer-visible state one-for-one
  - map instructions to combinational next-state logic
- But, contrived and inefficient
  1. all instructions run as slow as slowest instruction
  2. must provide worst-case combinational resource in parallel as required by any one instruction
  3. what about CISC ISAs? polyf?

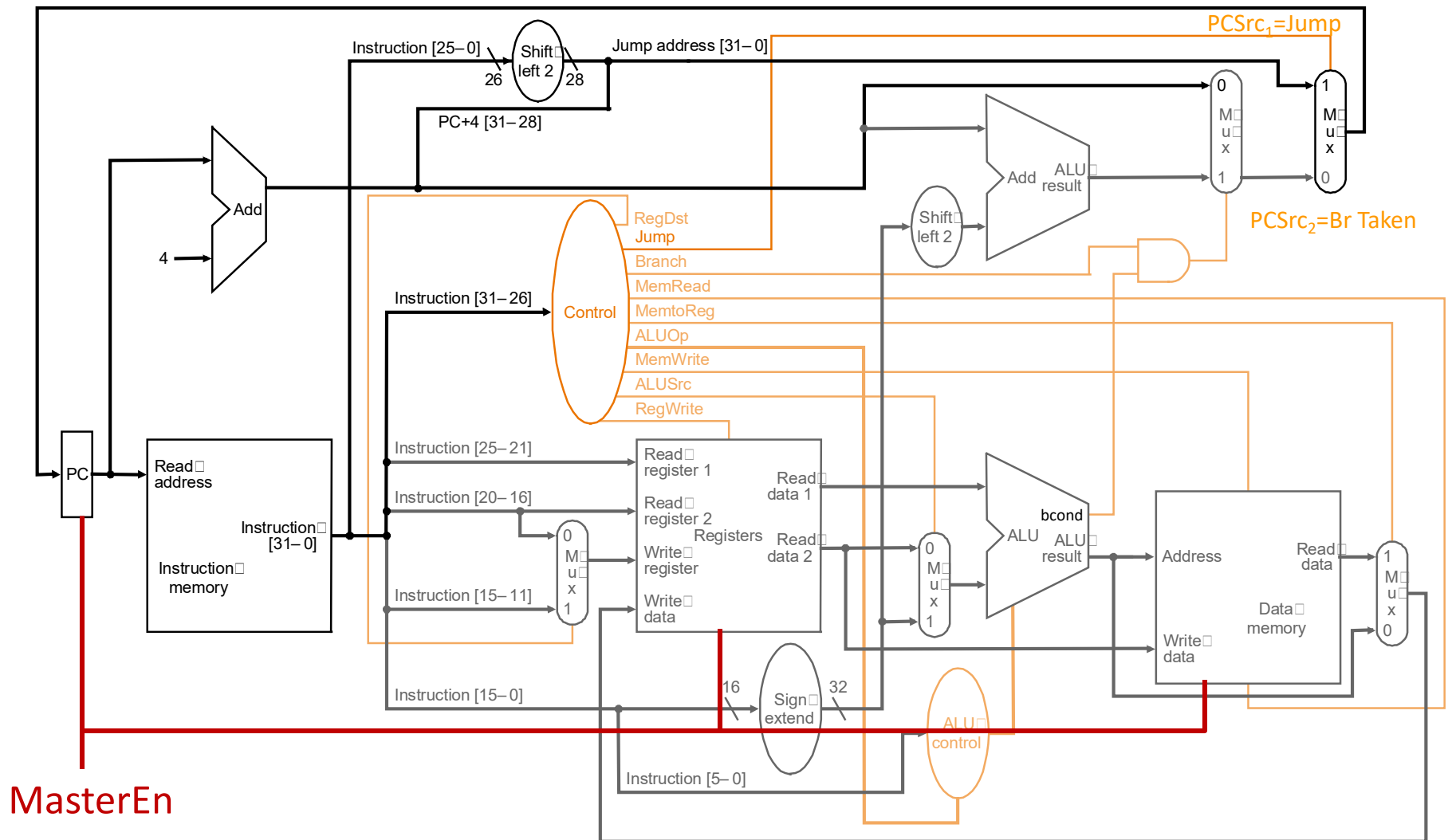
**Not the fastest, cheapest or even the simplest way**



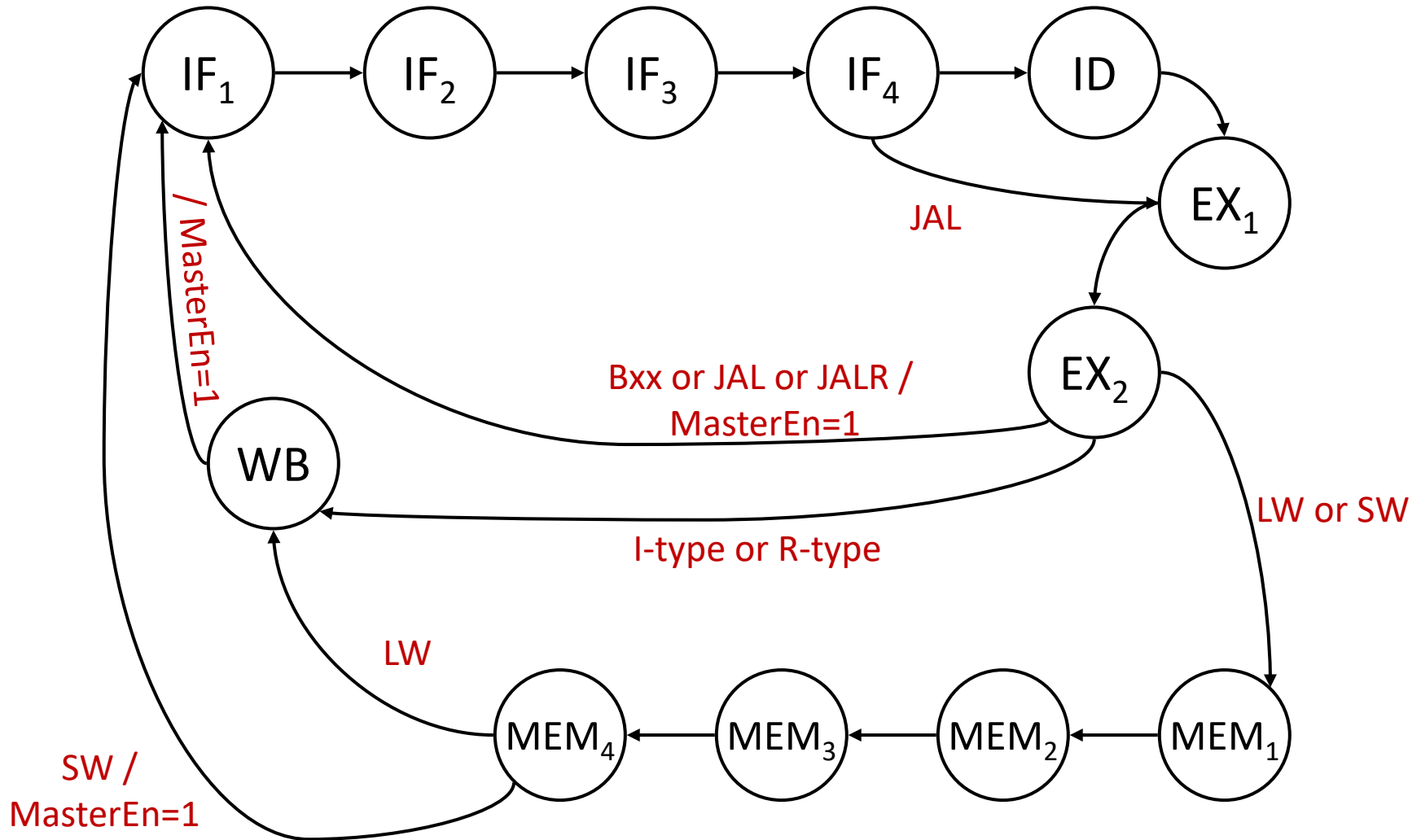
# Multi-cycle Implementation: Ver 1.0

- Each instruction type take only as much time as needed
  - run a 50 psec clock
  - each instruction type take as many 50-psec clock cycles as needed
- Add “MasterEnable” signal so architectural state ignores clock edges until after enough time
  - an instruction’s effect is still purely combinational from state to state
  - all other control signal unaffected

# Multi-Cycle Datapath: Ver 1.0



# Sequential Control: Ver 1.0



# Performance Analysis

- Iron Law:

$$\text{wall clock time} = (\text{inst/program}) (\text{cyc/inst}) (\text{time/cyc})$$

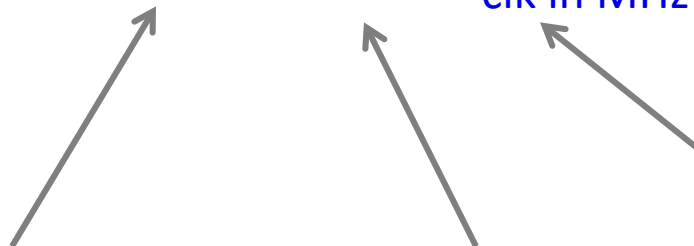
- For same ISA, inst/program is the same; okay to compare

$$\text{MIPS} = \text{IPC} \times f_{\text{clk in MHz}}$$

million instructions  
per second

instructions per cycle

frequency in MHz



# Performance Analysis

- Single-Cycle Implementation

$$1 \times 1,667\text{MHz} = 1667 \text{ MIPS}$$

- Multi-Cycle Implementation

$$\text{IPC}_{\text{avg}} \times 20,000 \text{ MHz} = 2178 \text{ MIPS}$$

what is  $\text{IPC}_{\text{average}}$ ?

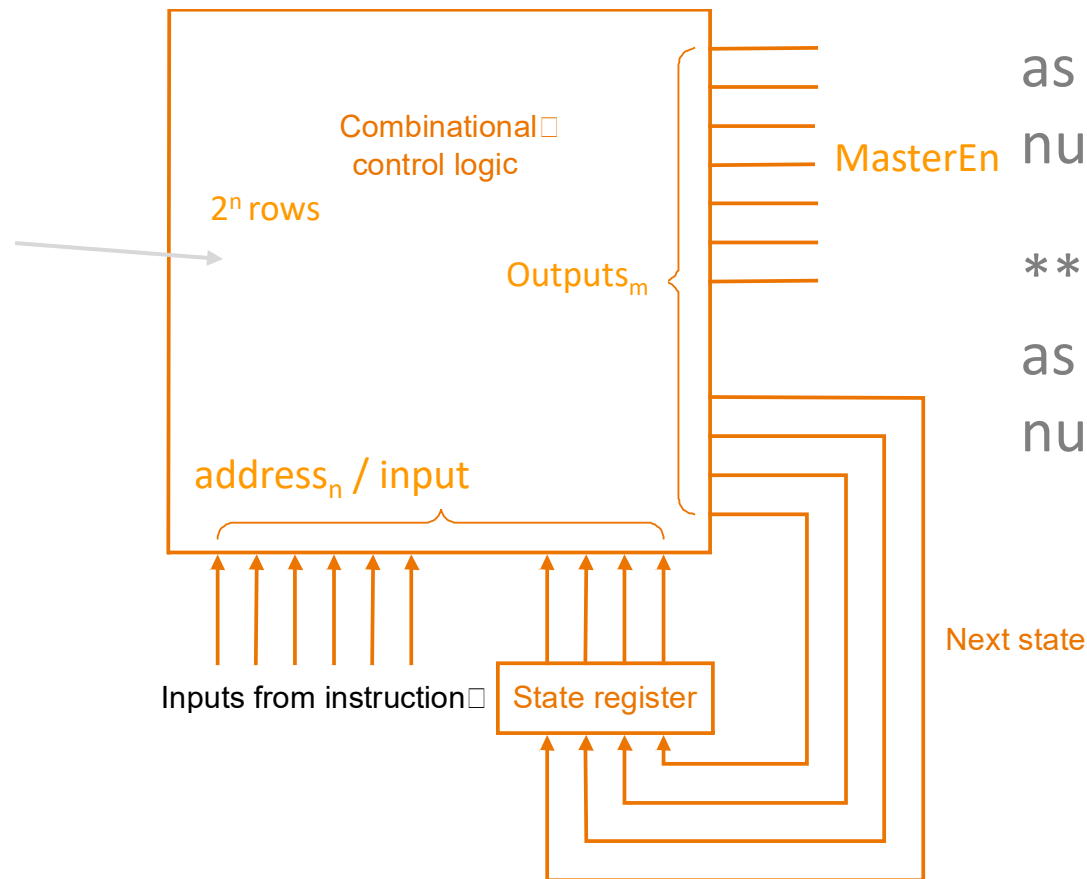
- Assume: 25% LW, 15% SW, 40% ALU, 13.3% Branch, 6.7% Jumps [Agerwala and Cocke, 1987]
  - weighted arithmetic mean of CPI  $\Rightarrow 9.18$
  - weighted harmonic mean of IPC  $\Rightarrow 0.109$
  - ~~weighted arithmetic mean of IPC  $\Rightarrow 0.115$~~

$$\text{MIPS} = \text{IPC} \times f_{\text{clk}}$$

# Microsequencer: Ver 1.0

- ROM as a combinational logic lookup table

literally  
holds the  
truth table



\*\* ROM size grows  
as  $O(2^n)$  as the  
number of inputs

\*\* ROM size grows  
as  $O(m)$  as the  
number of outputs

# Microcoding: Ver 0

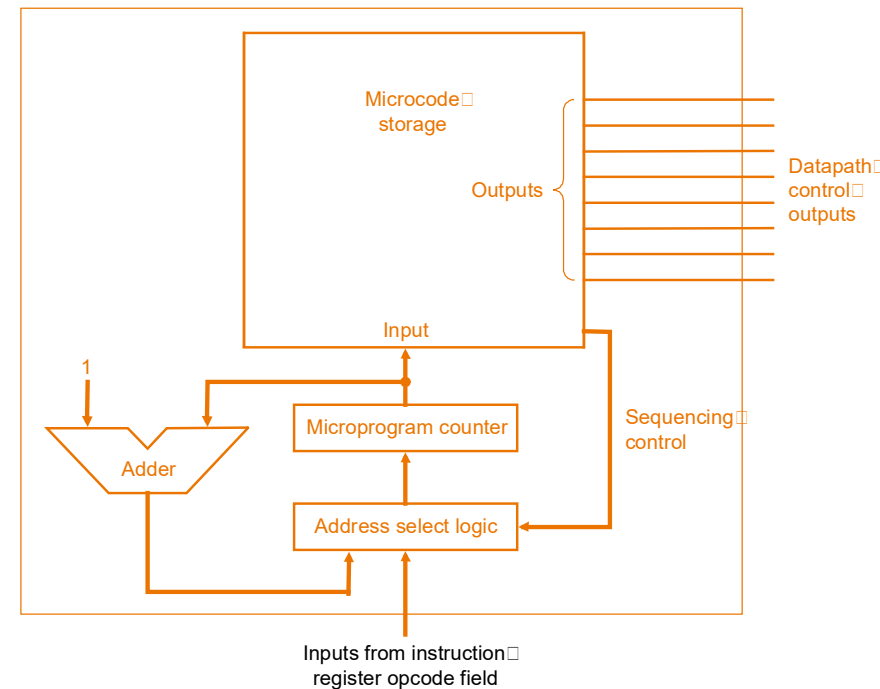
(note: this is only about counting clock ticks)

state label	cntrl flow	conditional targets					
		R/I-type	LW	SW	Bxx	JALR	JAL
IF <sub>1</sub>	next	-	-	-	-	-	-
IF <sub>2</sub>	next	-	-	-	-	-	-
IF <sub>3</sub>	next	-	-	-	-	-	-
IF <sub>4</sub>	goto	ID	ID	ID	ID	ID	EX <sub>1</sub>
ID	next	-	-	-	-	-	-
EX <sub>1</sub>	next	-	-	-	-	-	-
EX <sub>2</sub>	goto	WB	MEM <sub>1</sub>	MEM <sub>1</sub>	IF <sub>1</sub>	IF <sub>1</sub>	IF <sub>1</sub>
MEM <sub>1</sub>	next	-	-	-	-	-	-
MEM <sub>2</sub>	next	-	-	-	-	-	-
MEM <sub>3</sub>	next	-	-	-	-	-	-
MEM <sub>4</sub>	goto	-	WB	IF <sub>1</sub>	-	-	-
WB	goto	IF <sub>1</sub>	IF <sub>1</sub>	-	-	-	-
<b>CPI</b>		<b>8</b>	<b>12</b>	<b>11</b>	<b>7</b>	<b>7</b>	<b>6</b>

A systematic approach to FSM sequencing/control

# Microcontroller/Microsequencer

- A stripped-down “processor” for sequencing and control
  - control states are like  $\mu$ PC
  - $\mu$ PC indexed into a  $\mu$ program ROM to select an  $\mu$ instruction
  - $\mu$ program state and well-formed control-flow support (branch, jump)
  - fields in the  $\mu$ instruction maps to control signals
- Very elaborate  $\mu$ controllers have been built

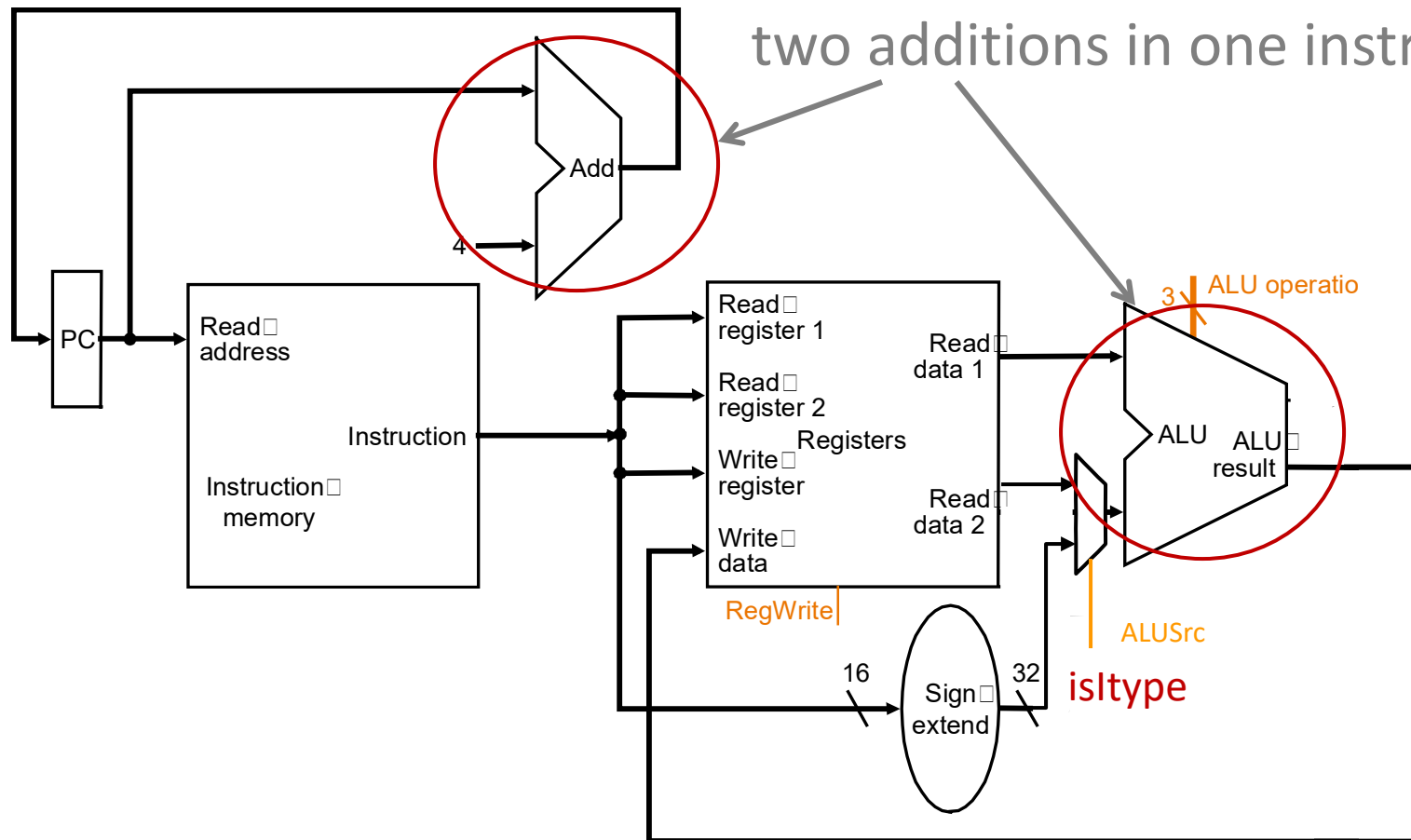




**Go Cheap!!  
(And More Capable)**

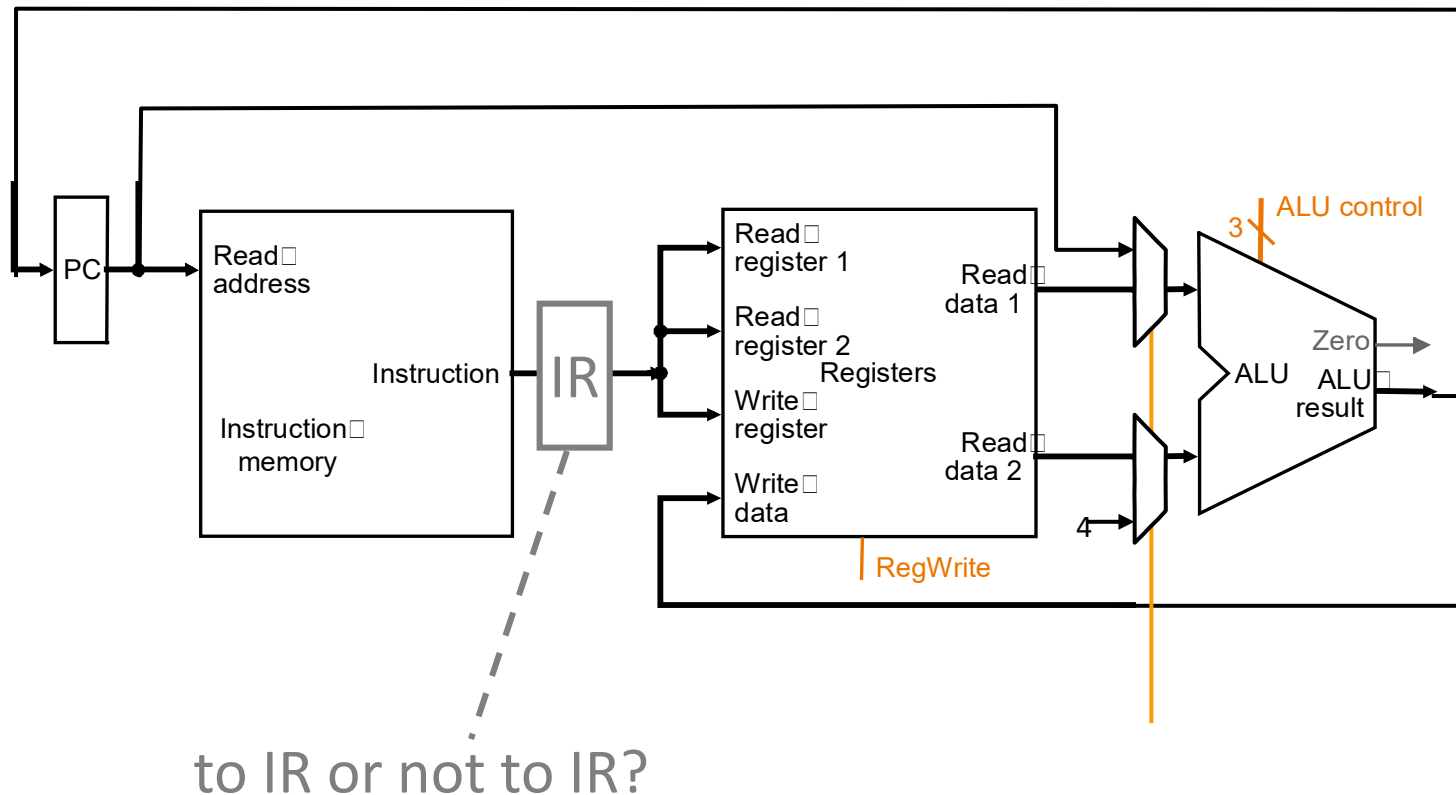
# Reducing Datapath by Resource Reuse

How to reuse same adder for two additions in one instruction

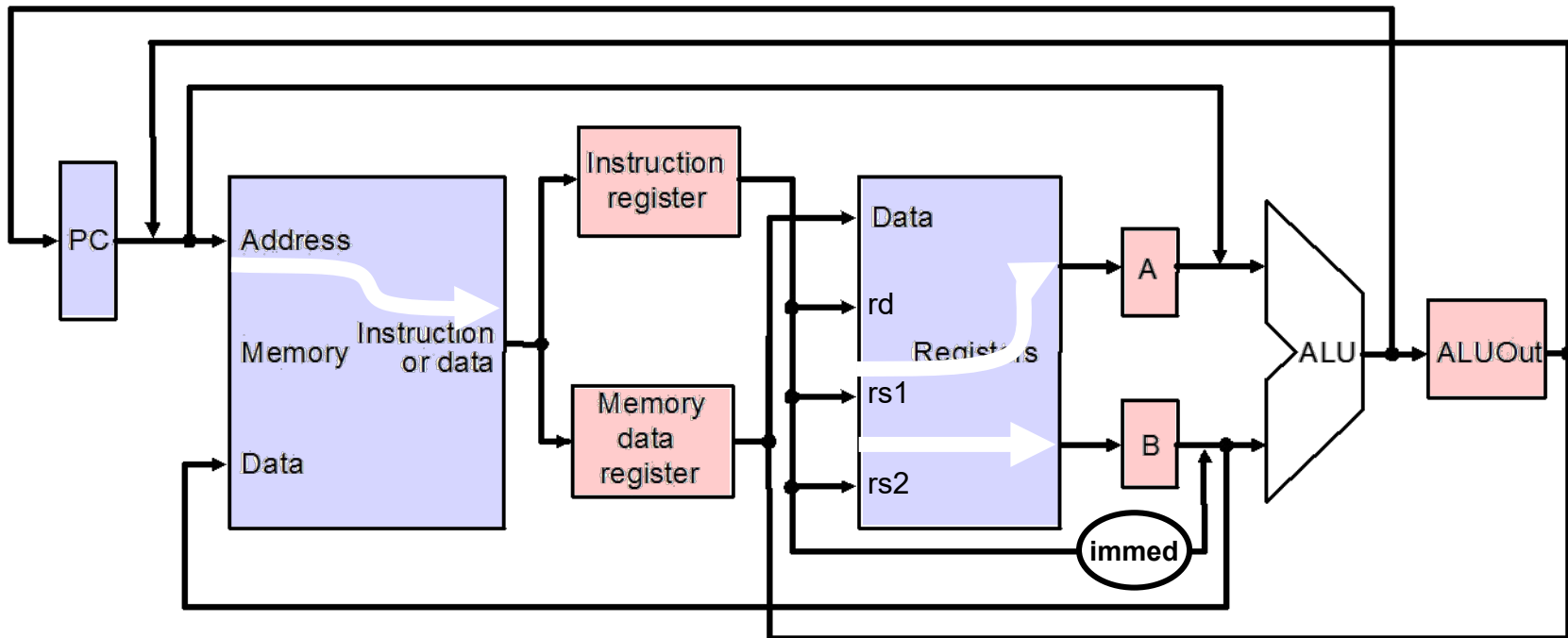


“Single-cycle” reused same adder for different instructions

# Reducing Datapath by Sequential Reuse



# Removing Redundancies



- Latch Enables: PC, IR, MDR, A, B, ALUOut, RegWr, MemWr
- Steering:  $ALUSrc1\{RF, PC\}$ ,  $ALUSrc2\{RF, immed\}$ ,  
 $MAddrSrc\{PC, ALUOut\}$ ,  $RFDDataSrc\{ALUOut, MDR\}$

Could also reduce down to a single register read-write port!

# Synchronous Register Transfers

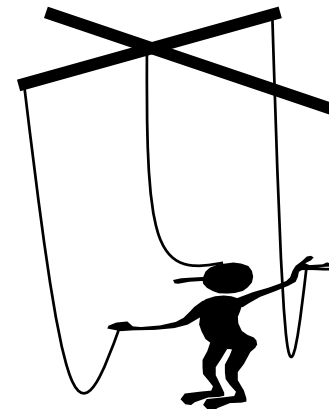
- Synchronous state with latch enables
    - PC, IR, RF, MEM, A, B, ALUOut, MDR
  - One can enumerate all possible “register transfers”
  - For example starting from PC
    - $IR \leftarrow MEM[PC]$
    - $MDR \leftarrow MEM[PC]$
    - $PC \leftarrow PC \oplus 4$
    - $PC \leftarrow PC \oplus B$
    - $PC \leftarrow PC \oplus \text{immediate}(IR)$
    - $ALUOut \leftarrow PC \oplus 4$
    - $ALUOut \leftarrow PC \oplus \text{immediate}(IR)$
    - $ALUOut \leftarrow PC \oplus B$
- Not all feasible RTs are meaningful

# Useful Register Transfers (by dest)

- $PC \leftarrow PC + 4$
- $PC \leftarrow PC + \text{immediate}_{\text{SB-type, U-type}}(\text{IR})$
- $PC \leftarrow A + \text{immediate}_{\text{SB-type}}(\text{IR})$
- $IR \leftarrow \text{MEM}[ PC ]$
- $A \leftarrow \text{RF}[ rs1(\text{IR}) ]$
- $B \leftarrow \text{RF}[ rs2(\text{IR}) ]$
- $\text{ALUOut} \leftarrow A + B$
- $\text{ALUOut} \leftarrow A + \text{immediate}_{\text{I-type, S-type}}(\text{IR})$
- $\text{ALUOut} \leftarrow PC + 4$
- $\text{MDR} \leftarrow \text{MEM}[ \text{ALUOut} ]$
- $\text{MEM}[ \text{ALUOut} ] \leftarrow B$
- $\text{RF}[ rd(\text{IR}) ] \leftarrow \text{ALUOut},$
- $\text{RF}[ rd(\text{IR}) ] \leftarrow \text{MDR}$

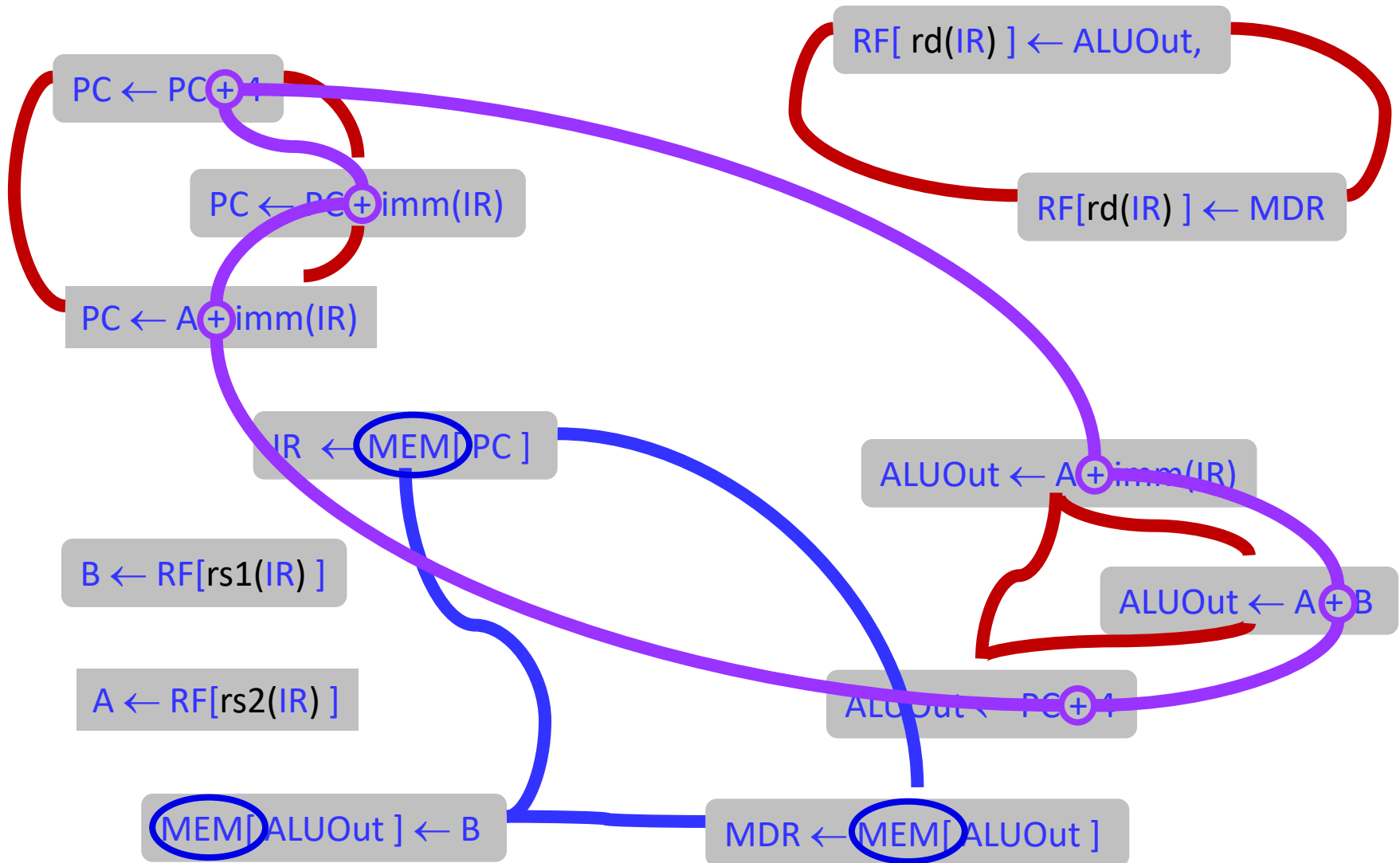
# RT Sequencing: R-Type ALU

- **IF**  
 $IR \leftarrow MEM[PC]$  step 1
- **ID**  
 $A \leftarrow RF[rs1(IR)]$  step 2  
 $B \leftarrow RF[rs2(IR)]$  step 3
- **EX**  
 $ALUOut \leftarrow A + B$  step 4
- **MEM**
- **WB**  
 $RF[rd(IR)] \leftarrow ALUOut$  step 5  
 $PC \leftarrow PC + 4$  step 6



if MEM[PC] == ADD rd rs1 rs2  
 $GPR[rd] \leftarrow GPR[rs1] + GPR[rs2]$   
 $PC \leftarrow PC + 4$

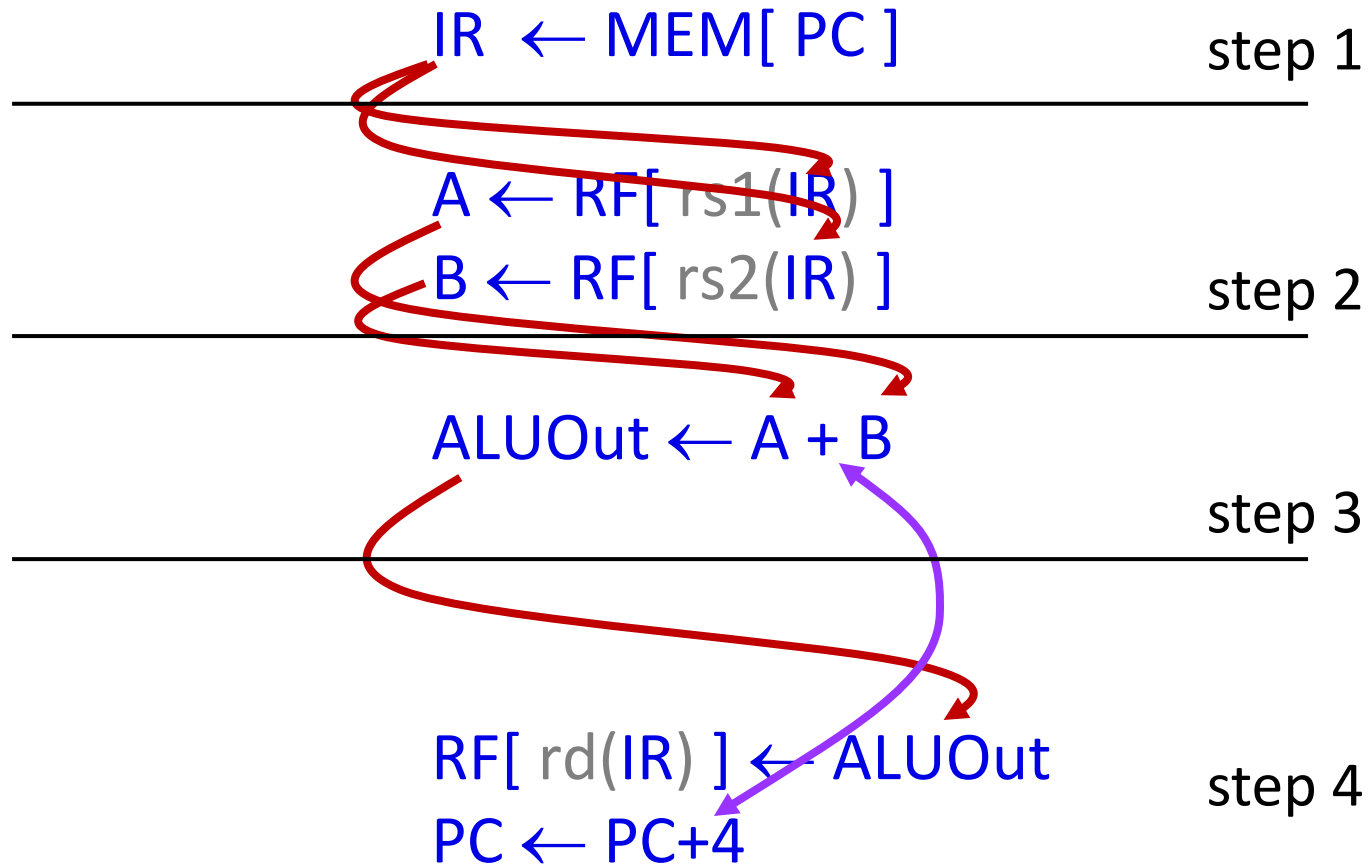
# RT Datapath Conflicts



Can utilize each resource only once per control step (cycle)



# RT Sequencing: R-Type ALU

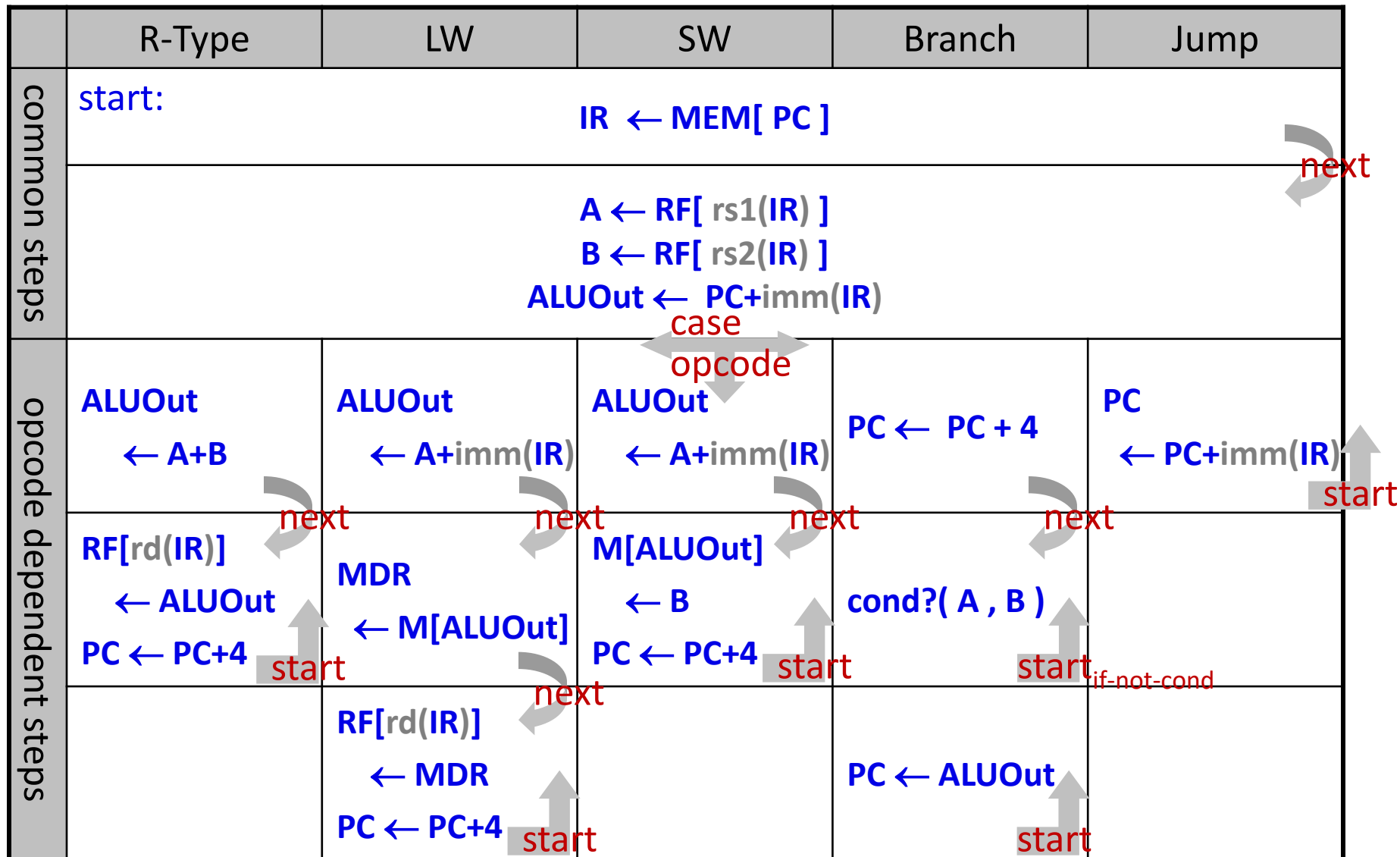


# RT Sequencing: LW

- **IF**  
 $IR \leftarrow \text{MEM}[PC]$
- **ID**  
 $A \leftarrow \text{RF}[rs1(IR)]$   
 $B \leftarrow \text{RF}[rs2(IR)]$
- **EX**  
 $\text{ALUOut} \leftarrow A + \text{imm}_{I\text{-type}}(IR)$
- **MEM**  
 $\text{MDR} \leftarrow \text{MEM}[\text{ALUOut}]$
- **WB**  
 $\text{RF}[rd(IR)] \leftarrow \text{MDR}$   
 $PC \leftarrow PC + 4$

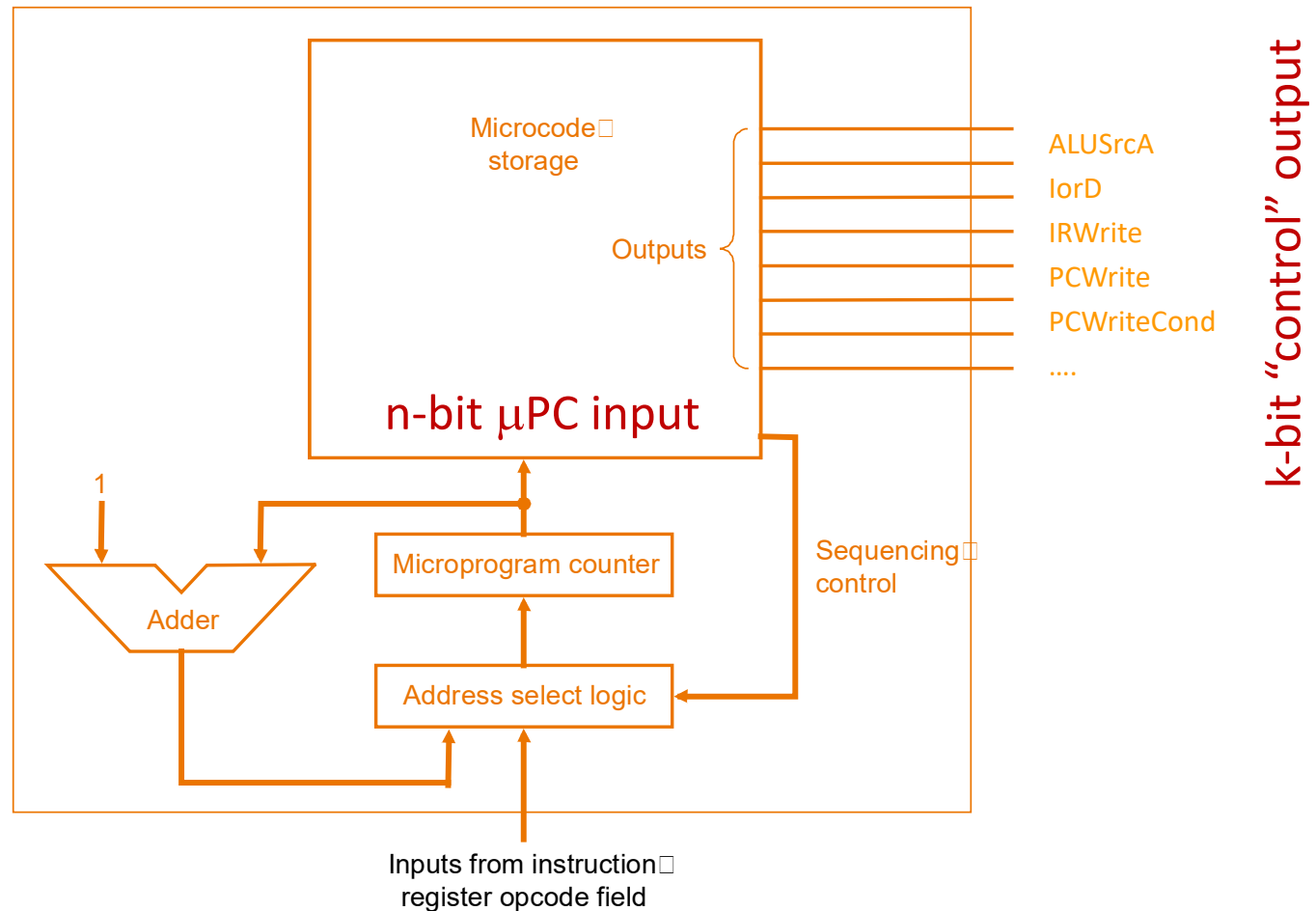
if MEM[PC]==LW rd offset(base)  
EA = sign-extend(offset) + GPR[base]  
GPR[rd] ← MEM[ EA ]  
PC ← PC + 4

# Combined RT Sequencing



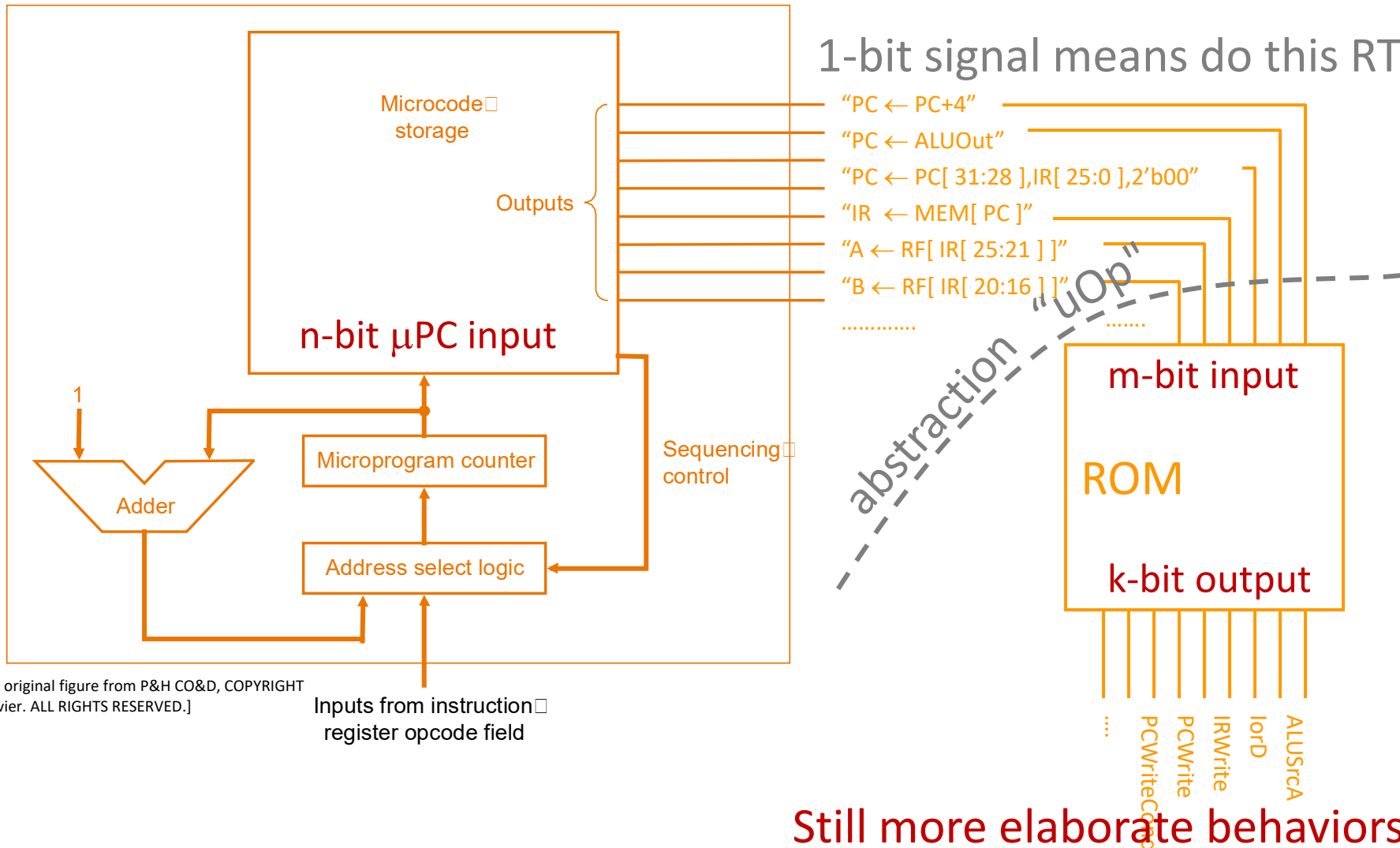
RTs in each state corresponds to some setting of the control signals

# Horizontal Microcode



Control Store:  $2^n \times k$  bit (not including sequencing)

# Vertical Microcode

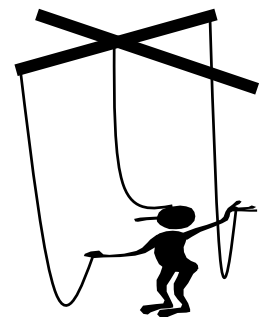
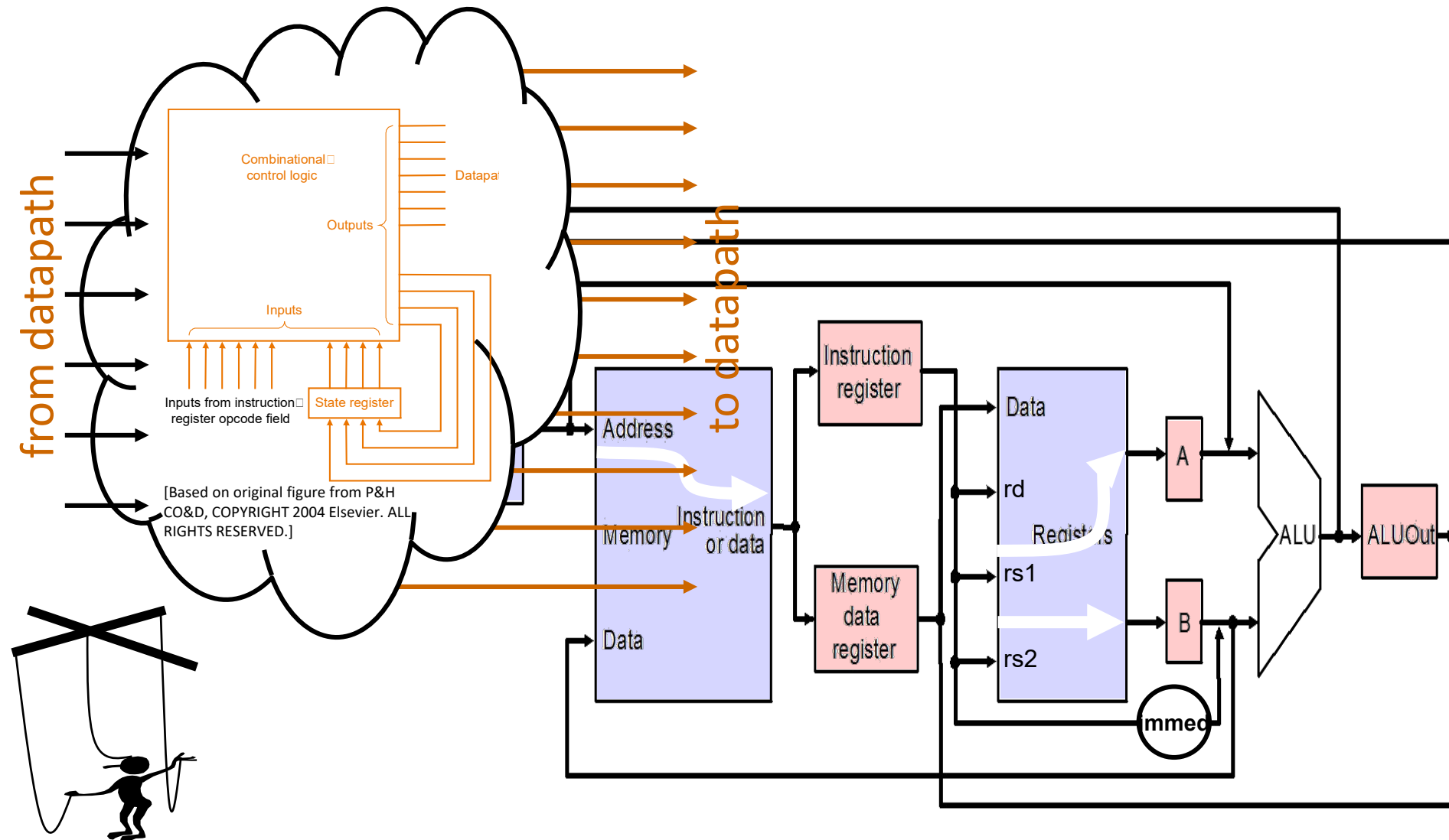


Still more elaborate behaviors  
can be sequenced as  $\mu$ subroutines

[Based on original figure from P&H CO&D, COPYRIGHT  
2004 Elsevier. ALL RIGHTS RESERVED.]

Inputs from instruction  
register opcode field

# μProgrammed Implementation



# Microcoding for CISC

- Can we extend last slide
  - to support a new instruction?
  - to support a complex instruction, e.g. polyf?
- Yes, very simple datapath do very complicated things easily but with a slowdown
  - if I can sequence an arbitrary RISC instruction then I can sequence an arbitrary “RISC program” as a  $\mu$ program sequence
  - will need some  $\mu$ ISA state (e.g. loop counters) for more elaborate  $\mu$ programs
  - more elaborate  $\mu$ ISA features also make life easier

# Single-Bus Microarchitecture

## [8086 Family User's Manual]

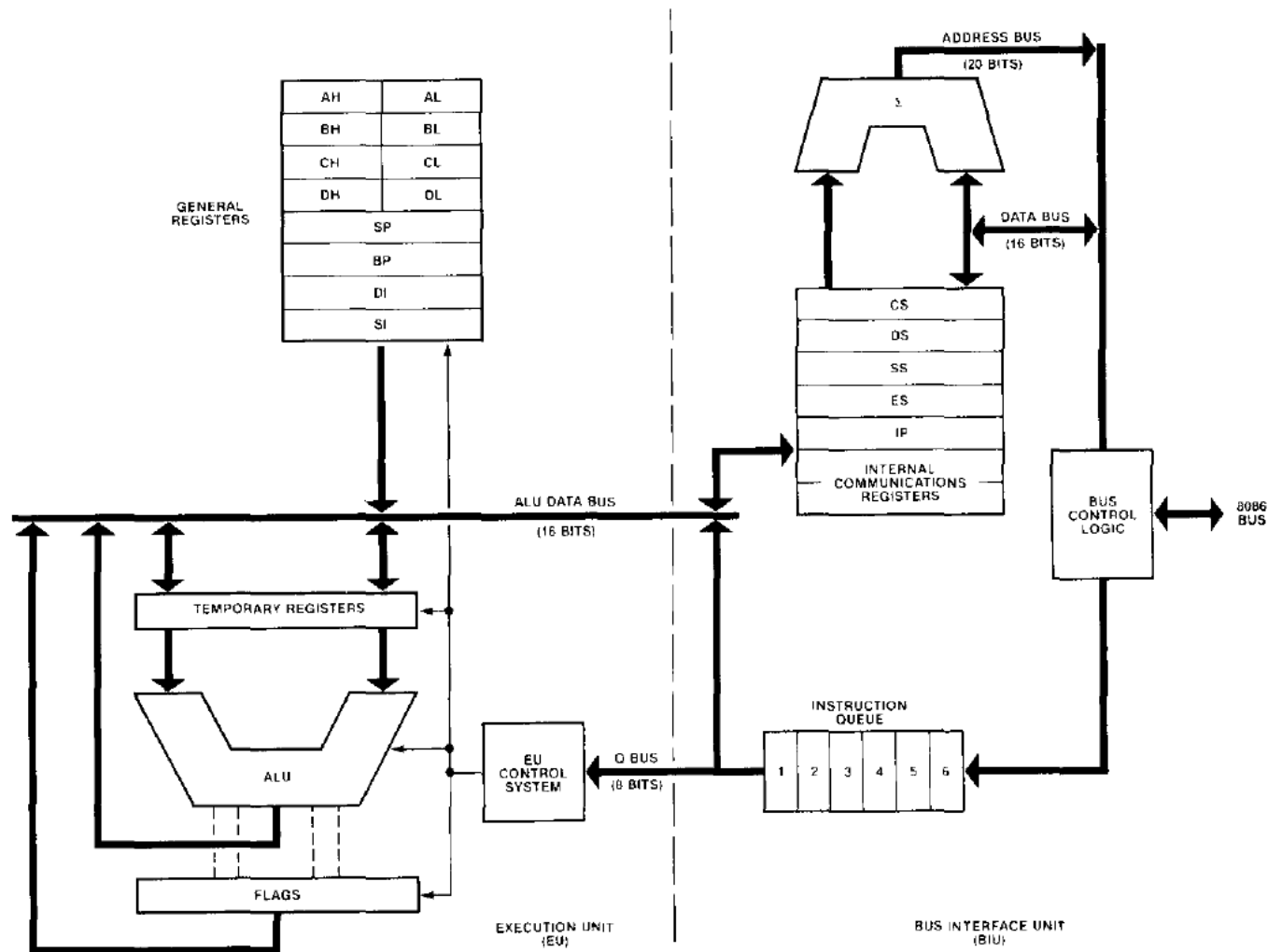


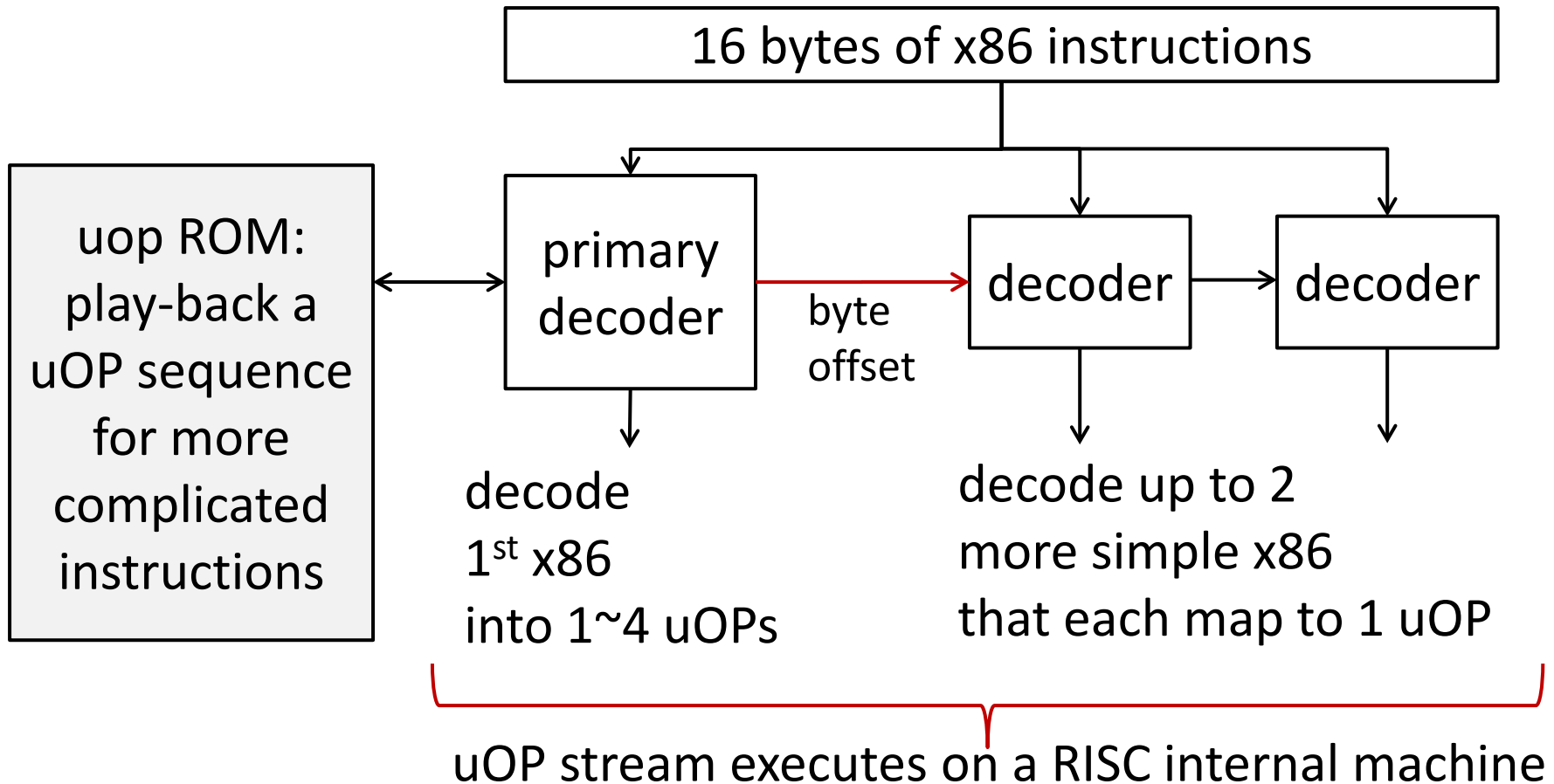
Figure 4-3. 8086 Elementary Block Diagram

You get a try on HW2



# High Performance CISC Today

- High-perf x86s translate CISC inst's to RISC uOPs
- Pentium-Pro decoding example:



# Evolution of ISAs

- Why were the earlier ISAs so simple? e.g., EDSAC
  - technology
  - precedence
- Why did it get so complicated later? e.g., VAX11
  - assembly programming
  - lack of memory size and performance
  - microprogrammed implementation
- Why did it become simple again? e.g., RISC
  - memory size and speed (cache!)
  - compilers
- Why is x86 still so popular?
  - technical merit vs. {SW base, psychology, deep pocket}
- Why has ARM thrived while other RISC ISAs vanished

CISC  
Complex Instruction  
Set Architecture

Reduced Instruction  
Set Architecture

Recall

Why RISC-V now?