计算机组成与系统结构
Computer Organization & System Architecture

Huang Kejie (黄科杰) 百人计划研究员

Office: 玉泉校区老生仪楼 304

Email address: huangkejie@zju.edu.cn

HP: 17706443800

Review

- We have designed a complete datapath
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - New instructions can be added with just control?

Last Time in Lecture 5

- 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
- Complex instructions sets impede parallel and pipelined implementations
- Load/store, register-rich ISAs (pioneered by Cray, popularized by RISC) perform better in new VLSI technology

Analyzing Microcoded Machines

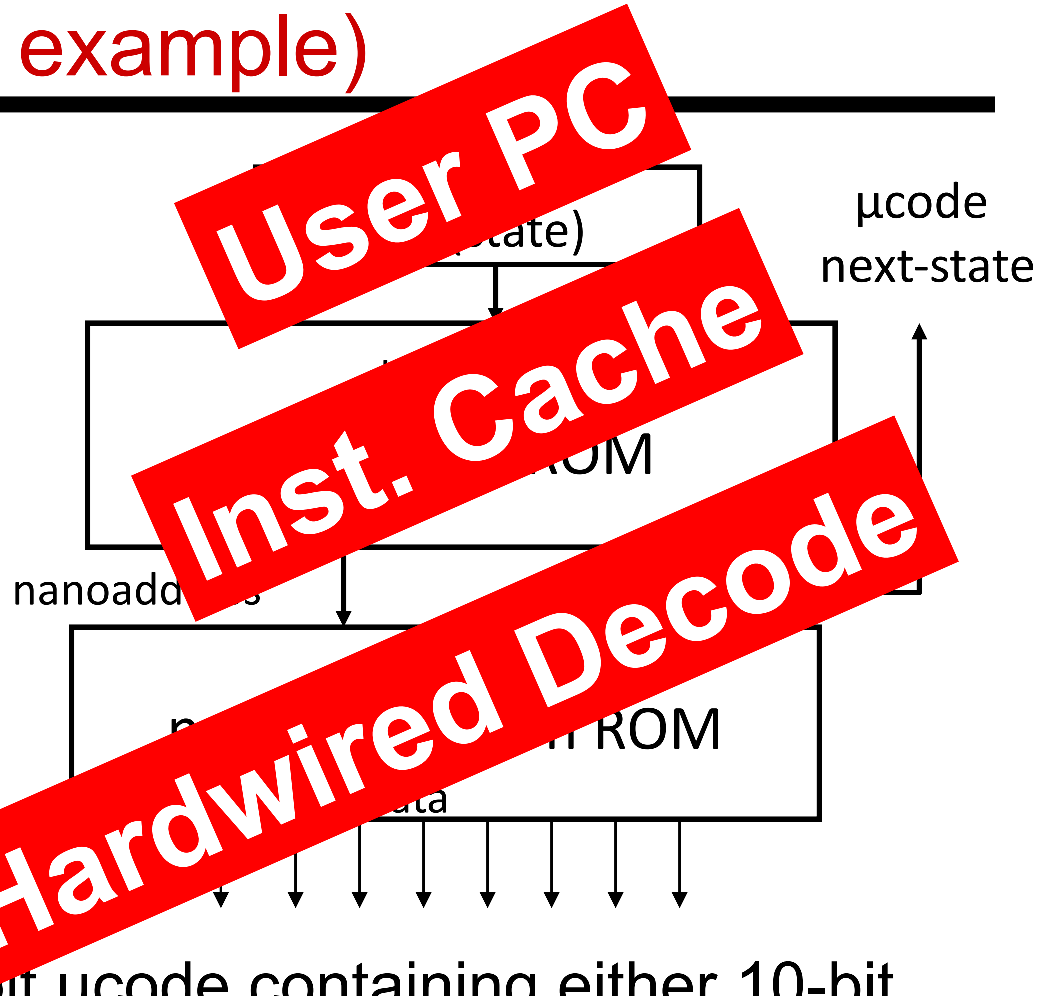
- John Cocke and group at IBM
  - Working on a simple pipelined processor, 801, and advanced compilers inside IBM
  - Ported experimental PL.8 compiler to IBM 370, and only used simple register-register and load/store instructions similar to 801
  - Code ran faster than other existing compilers that 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!

Reconsidering Microcode Machine (Nan 68000 example)

RISC!

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

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

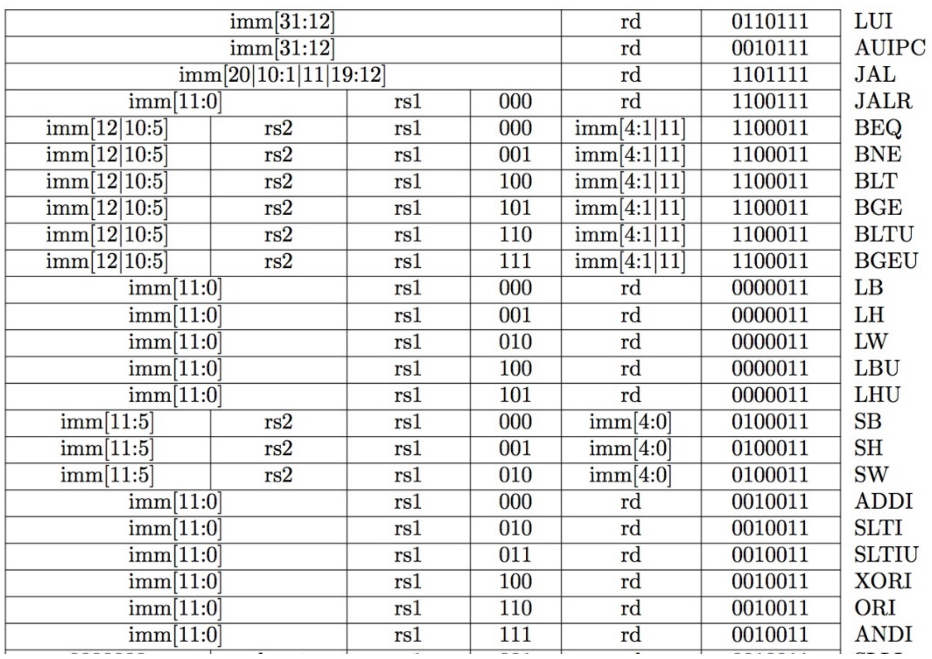


- Motorola 68000 had 17-bit µcode containing either 10-bit µjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

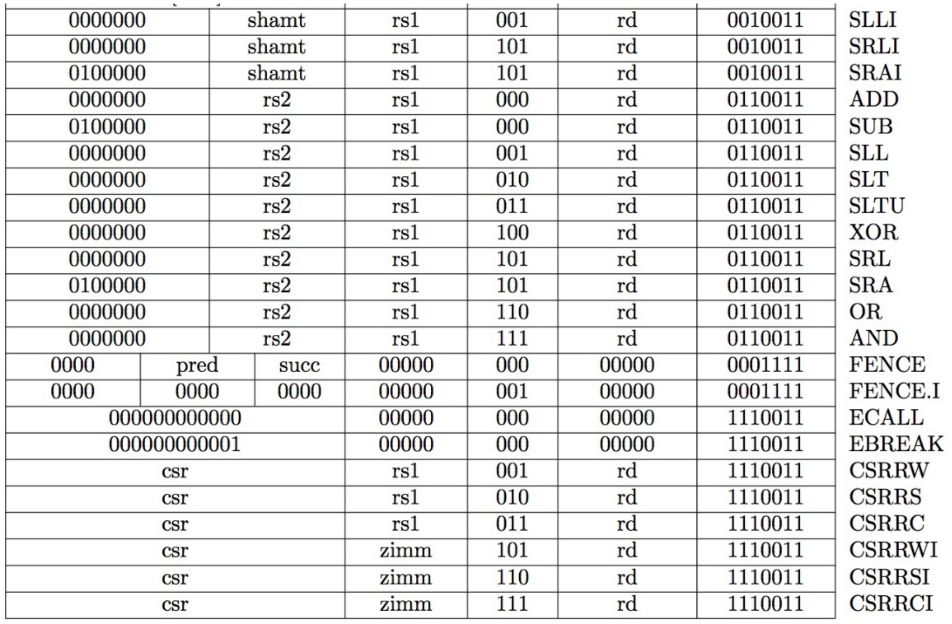
Microprogramming is far from extinct

- 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

Recap: Complete RV32I ISA



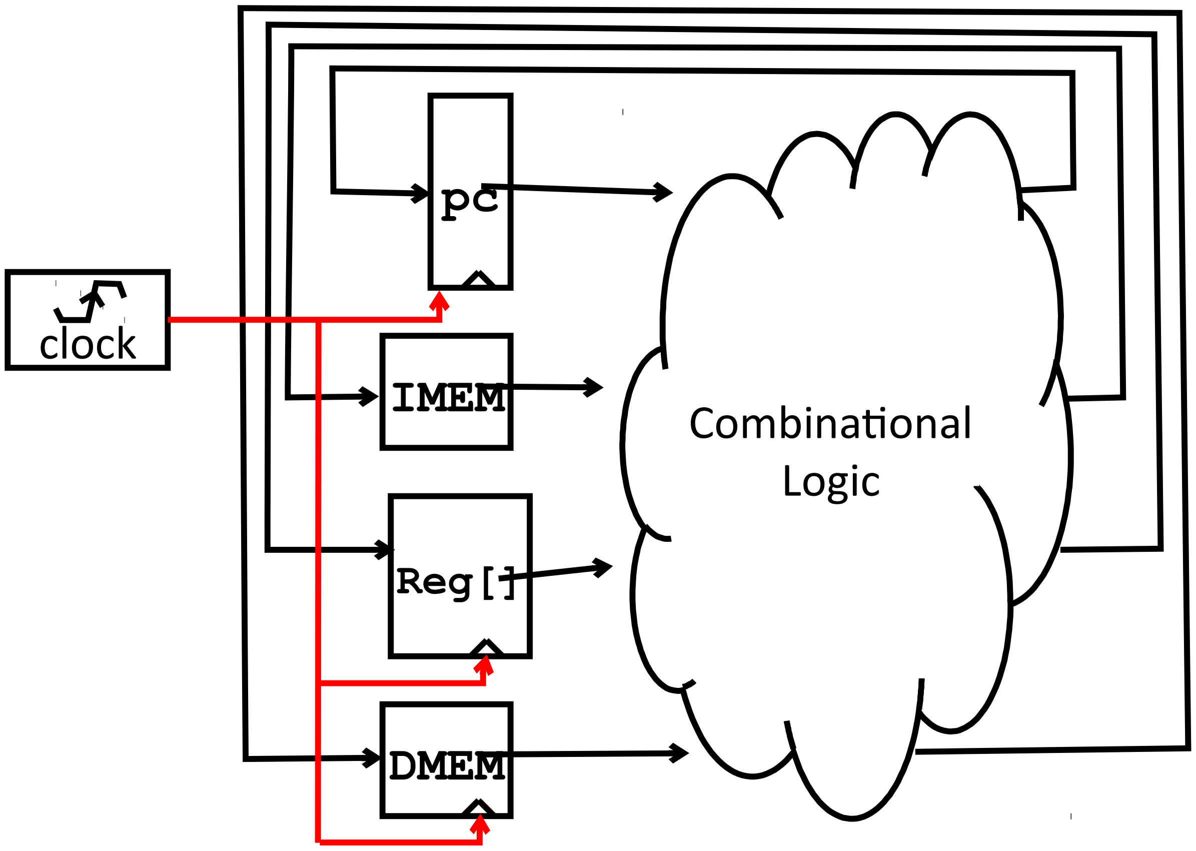
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI



0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 pred	succ	00000	000	00000	0001111	FENCE
0000 0000	0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

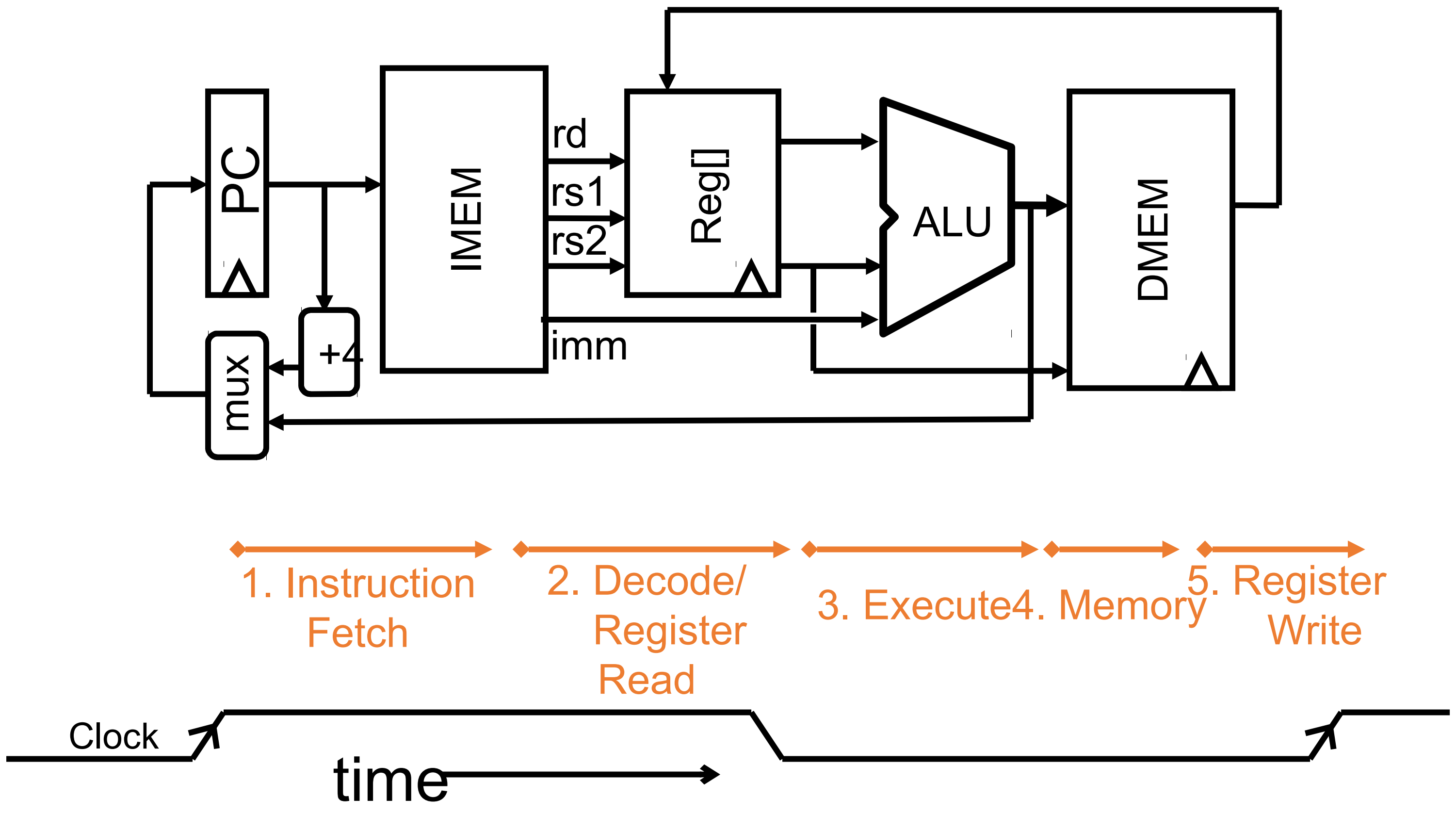
- RV32I has 47 instructions
- 37 instructions are enough to run any C program

One-Instruction-Per-Cycle RISC-V Machine

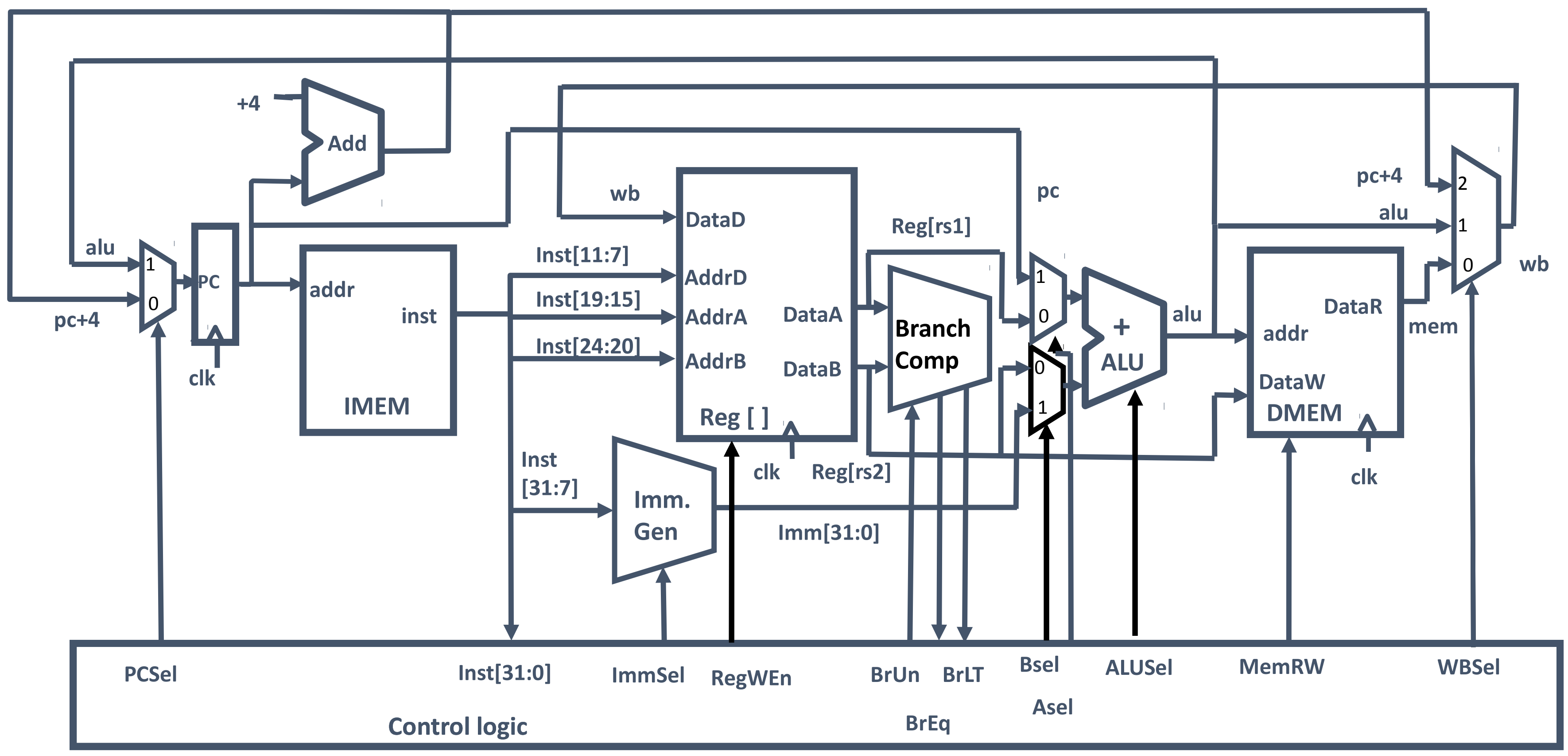


- On every tick of the clock, the computer executes one instruction
- Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge
- At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle

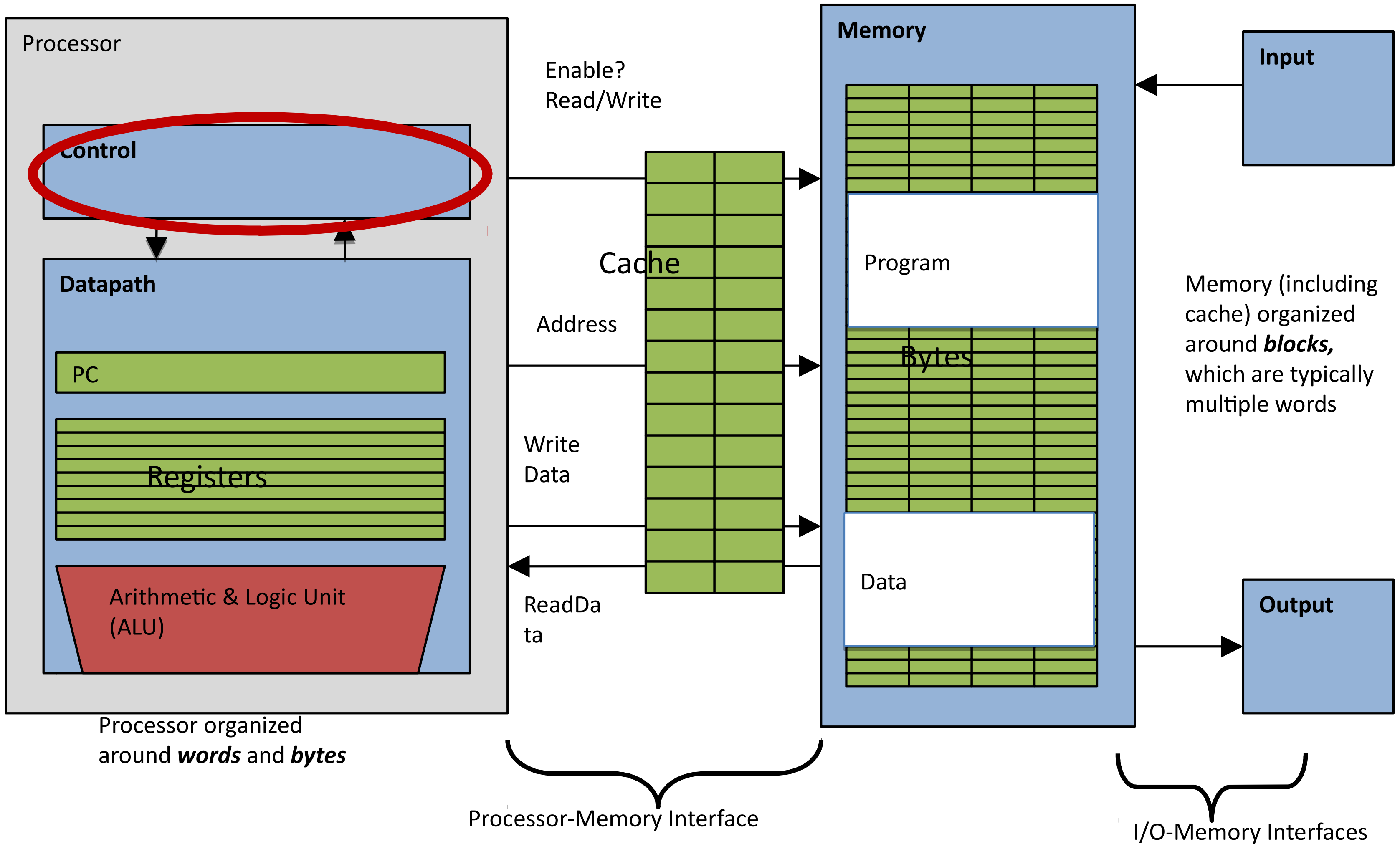
Basic Phases of Instruction Execution



Complete Single-Cycle RV32I Datapath!



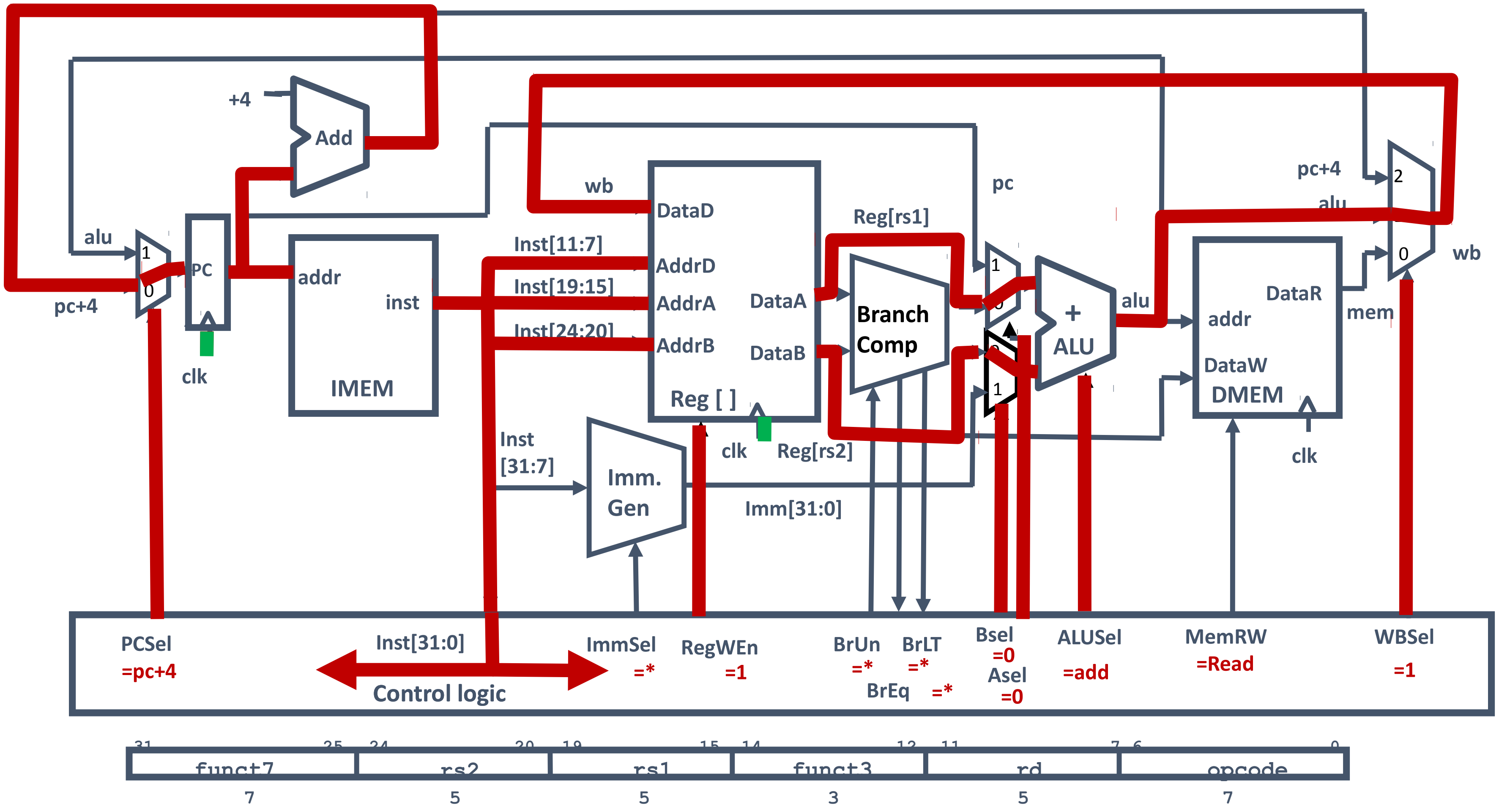
Our Single-Core Computer (w/o FPU, SIMD)



Processor organized around ***words*** and ***bytes***

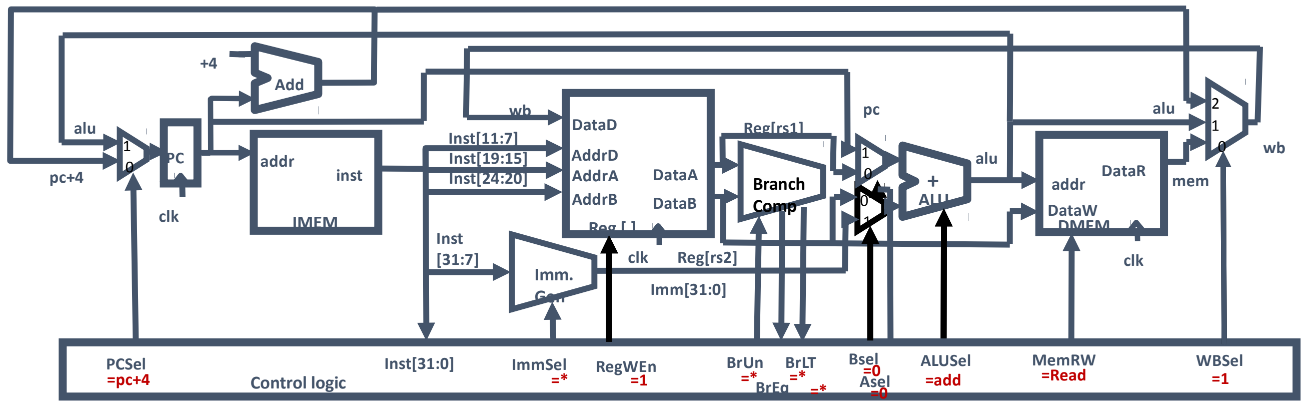
Memory (including cache) organized around ***blocks,*** which are typically multiple words

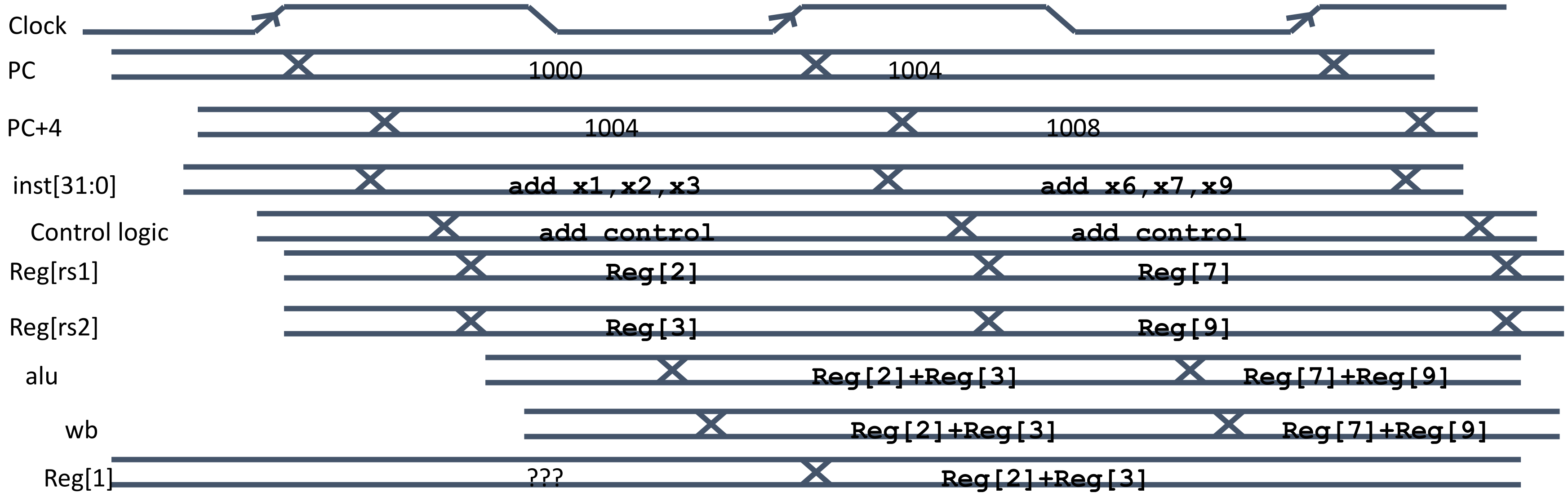
Example: add



+4
Add
alu
pc+4
1
0
PC
clk
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
wb
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
Reg[rs1]
Reg[rs2]
Branch Comp
pc
1
0
0
1
ALU
+
alu
Inst [31:7]
Imm. Gen
Imm[31:0]
addr
DataR
DataW
DMEM
clk
mem
pc+4
alu
2
0
wb
PCSel
=pc+4
Inst[31:0]
Control logic
ImmSel
=*
RegWEn
=1
BrUn
=*
BrLT
=*
BrEq
=*
Bsel
=0
Asel
=0
ALUSel
=add
MemRW
=Read
WBSel
=1
31
25
24
20
19
15
14
12
11
7
6
0
funct7
rs2
rs1
funct3
rd
opcode
7
5
5
3
5
7

add Execution

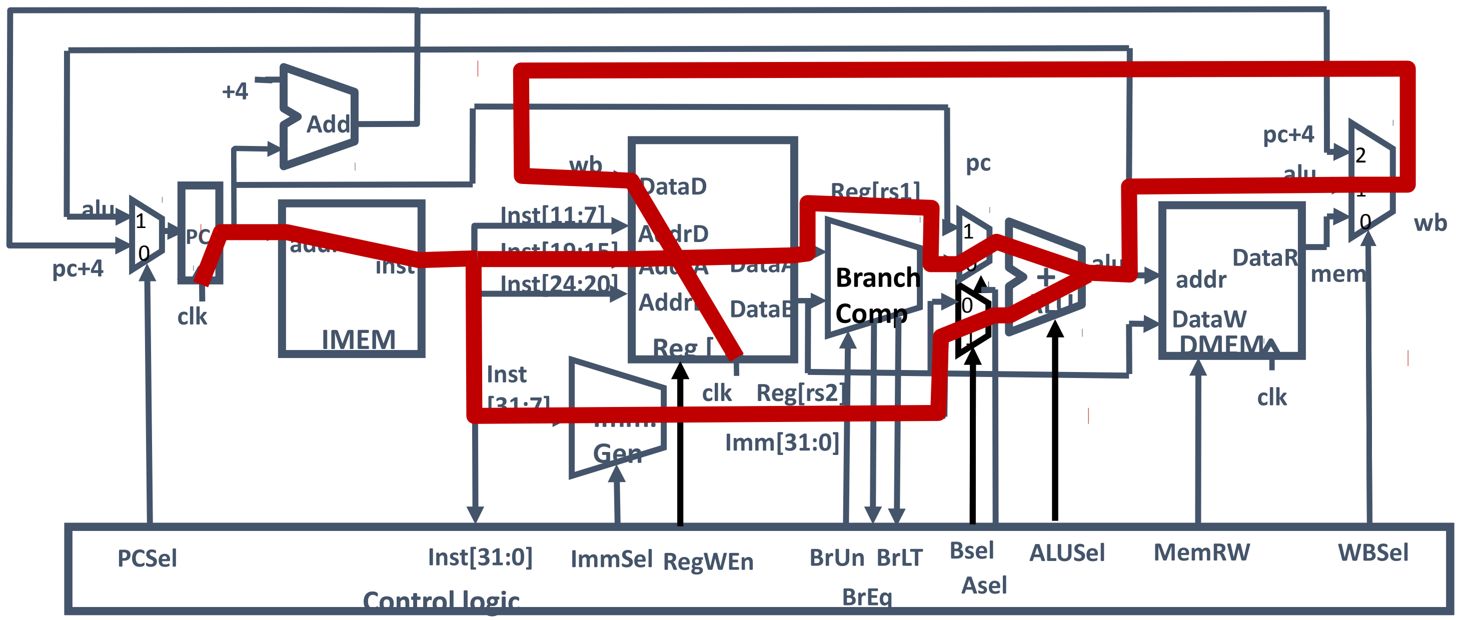




Peer Instruction(s): Critical Path

Critical path for a addi

R[rd] = R[rs1]+imm



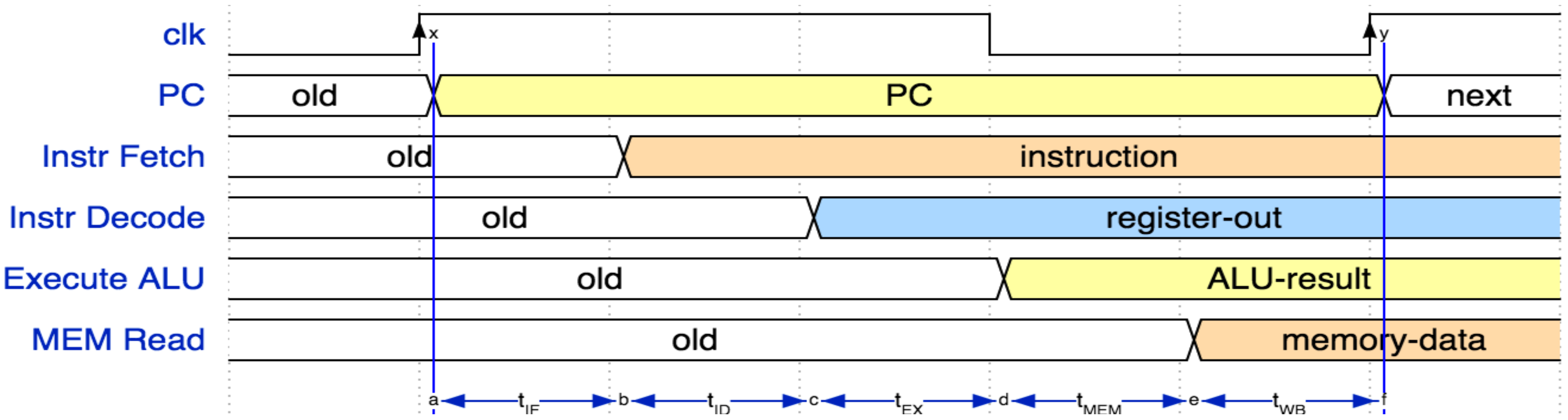
1) $t_{clk\text{-}q} + t_{Add} + t_{IMEM} + t_{Reg} + t_{BComp} + t_{ALU} + t_{DMEM} + t_{mux} + t_{Setup}$

[✓ 2) $t_{clk\text{-}q} + t_{IMEM} + \max\{t_{Reg}, t_{Imm}\} + t_{ALU} + 2t_{mux} + t_{Setup}$]

3) $t_{clk\text{-}q} + t_{IMEM} + \max\{t_{Reg}, t_{Imm}\} + t_{ALU} + 3t_{mux} + t_{DMEM} + t_{Setup}$

4) None of the above

Instruction Timing



IF	ID	EX	MEM	WB	Total
I-MEM	Reg Read	ALU	D-MEM	Reg W	
200 ps	100 ps	200 ps	200 ps	100 ps	800 ps

Instruction Timing

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X			500ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

- Maximum clock frequency
  - f_{max} = 1/800ps = 1.25 GHz

- Most blocks idle most of the time
  - E.g. $f_{max,ALU}$ = 1/200ps = 5 GHz!
  - How can we keep ALU busy all the time?
  - 5 billion adds/sec, rather than just 1.25 billion?
  - Idea: Factories use three employee shifts - equipment is always busy!

Performance Measures

- “Our” RISC-V executes instructions at 1.25 GHz
  - 1 instruction every 800 ps
- Can we improve its performance?
  - What do we mean with this statement?
  - Not so obvious:
    - Quicker response time, so one job finishes faster?
    - More jobs per unit time (e.g. web server returning pages)?
    - Longer battery life?

Transportation Analogy

	Sports Car	Bus
Passenger Capacity	2	50
Travel Speed	200 mph	50 mph
Gas Mileage	5 mpg	2 mpg

50 Mile trip:

	Sports Car	Bus
Travel Time	15 min	60 min
Time for 100 passengers	750 min	120 min
Gallons per passenger	5 gallons	0.5 gallons

Computer Analogy

Transportation	Computer
Trip Time	Program execution time: e.g. time to update display
Time for 100 passengers	Throughput: e.g. number of server requests handled per hour
Gallons per passenger	Energy per task*: e.g. how many movies you can watch per battery charge or energy bill for datacenter

* Note: power is not a good measure, since low-power CPU might run for a long time to complete one task consuming more energy than faster computer running at higher power for a shorter time

"Iron Law" of Processor Performance

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

Instructions per Program

- Determined by
  - Task
  - Algorithm, e.g. $O(N^2)$ vs $O(N)$
  - Programming language
  - Compiler
  - Instruction Set Architecture (ISA)

(Average) Clock cycles per Instruction

- Determined by
  - ISA
  - Processor implementation (or *microarchitecture*)
  - E.g. for “our” single-cycle RISC-V design, CPI = 1
  - Complex instructions (e.g. **`strcpy`**), CPI >> 1
  - Superscalar processors, CPI < 1 (next lecture)

Time per Cycle (1/Frequency)

- Determined by
  - Processor microarchitecture (determines critical path through logic gates)
  - Technology (e.g. 14nm versus 28nm)
  - Power budget (lower voltages reduce transistor speed)

Speed Tradeoff Example

- For some task (e.g. image compression) …

	Processor A	Processor B
# Instructions	1 Million	1.5 Million
Average CPI	2.5	1
Clock rate *f*	2.5 GHz	2 GHz
Execution time	1 ms	0.75 ms

Processor B is faster for this task, despite executing more instructions and having a lower clock rate!

Energy per Task

$$\frac{\text{Energy}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Energy}}{\text{Instruction}}$$

$$\frac{\text{Energy}}{\text{Program}} \alpha \frac{\text{Instructions}}{\text{Program}} * C \quad V^2$$

"Capacitance" depends on technology, processor features e.g. # of cores

Supply voltage, e.g. 1V

Want to reduce capacitance and voltage to reduce energy/task

Energy Tradeoff Example

- "Next-generation" processor
  - C (Moore's Law): -15 %
  - Supply voltage, V_{sup}: -15 %
  - Energy consumption: $1 - (1-0.85)^3$ = -39 %

- Significantly improved energy efficiency thanks to
  - Moore's Law AND
  - Reduced supply voltage

Energy “Iron Law”

Performance =	Power	*	Energy Efficiency
(Tasks/Second)	*(Joules/Second)*		*(Tasks/Joule)*

- Energy efficiency (e.g., instructions/Joule) is key metric in all computing devices

- For power-constrained systems (e.g., 20MW datacenter), need better energy efficiency to get more performance at same power

- For energy-constrained systems (e.g., 1W phone), need better energy efficiency to prolong battery life

End of Scaling

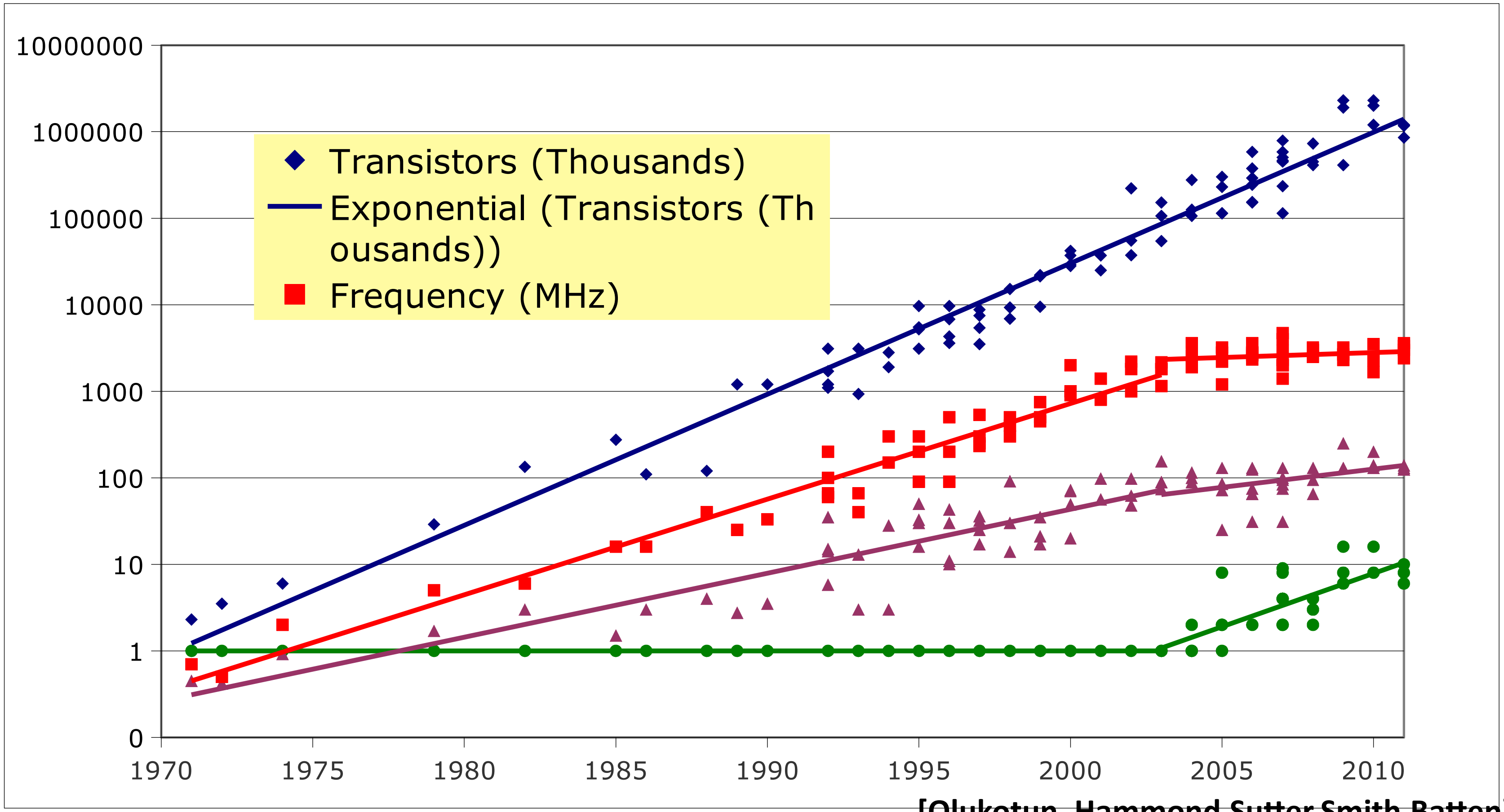
- In recent years, industry has not been able to reduce supply voltage much, as reducing it further would mean increasing "leakage power" where transistor switches don't fully turn off (more like dimmer switch than on-off switch)

- Also, size of transistors and hence capacitance, not shrinking as much as before between transistor generations

- Power becomes a growing concern – the "power wall"

- Cost-effective air-cooled chip limit around ~150W

End of Scaling



[Olukotun, Hammond,Sutter,Smith,Batten]

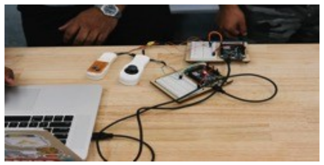
Pipelining

- A familiar example:
  - Getting a university degree

Year 1

Year 2

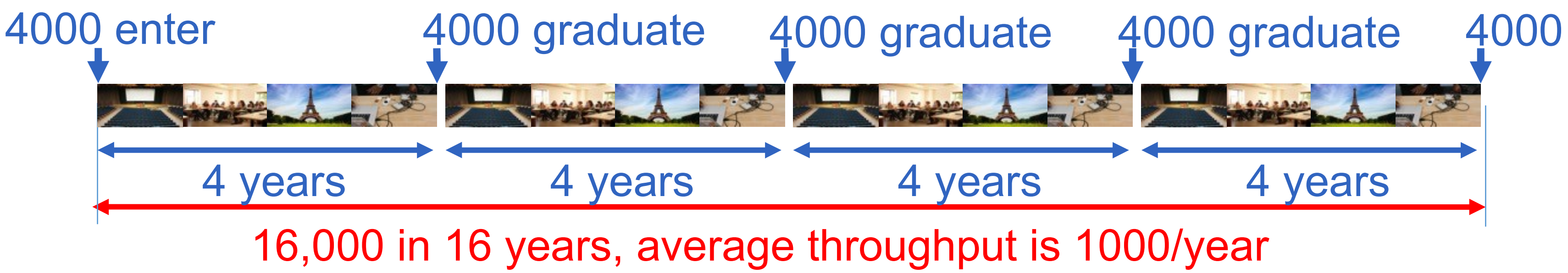
Year 3


Year 4

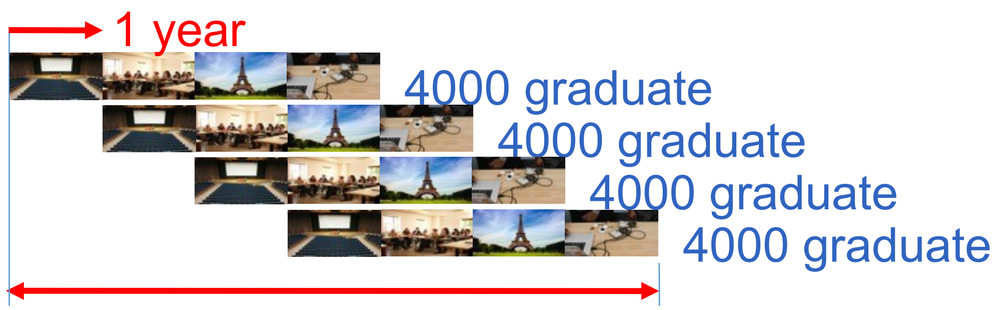
- Shortage of Computer scientists (your startup is growing):
  - How long does it take to educate 16,000?

Computer Scientist Education

- Option 1: *serial*



- Option 2: *pipelining*



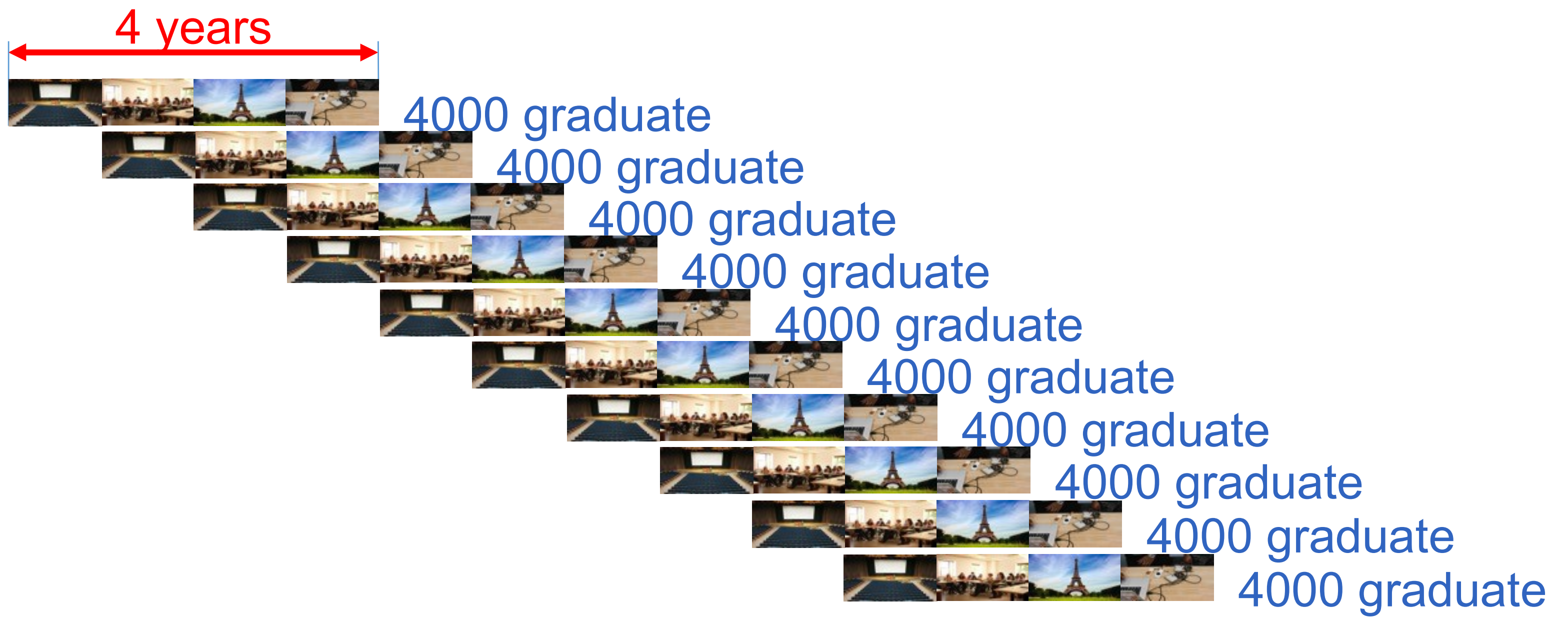
  - 16,000 in 7 years
  - Steady state throughput is 4000/year
  - Resources used efficiently
  - **4-fold improvement over serial education**

Latency versus Throughput

- Latency
  - Time from entering college to graduation
  - Serial 4 years
  - Pipelining 4 years
- Throughput
  - Average number of students graduating each year
  - Serial 1000
  - Pipelining 4000
- Pipelining
  - Increases throughput (4x in this example)
  - But does nothing to latency
    - sometimes worse (additional overhead e.g. for shift transition)

Simultaneous versus Sequential

- What happens *sequentially*?
- What happens *simultaneously*?



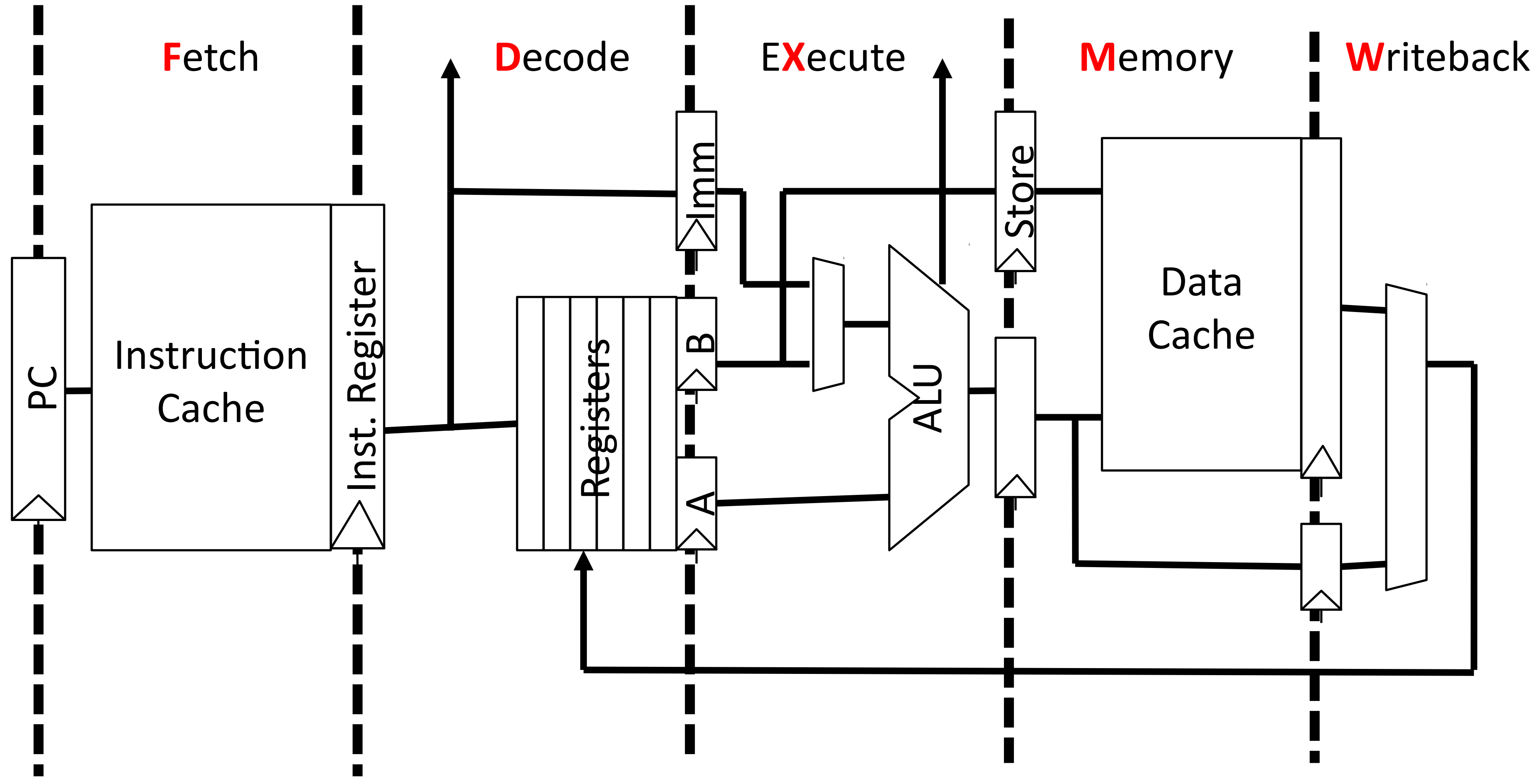
“Iron Law” of Processor Performance

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

- 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

Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

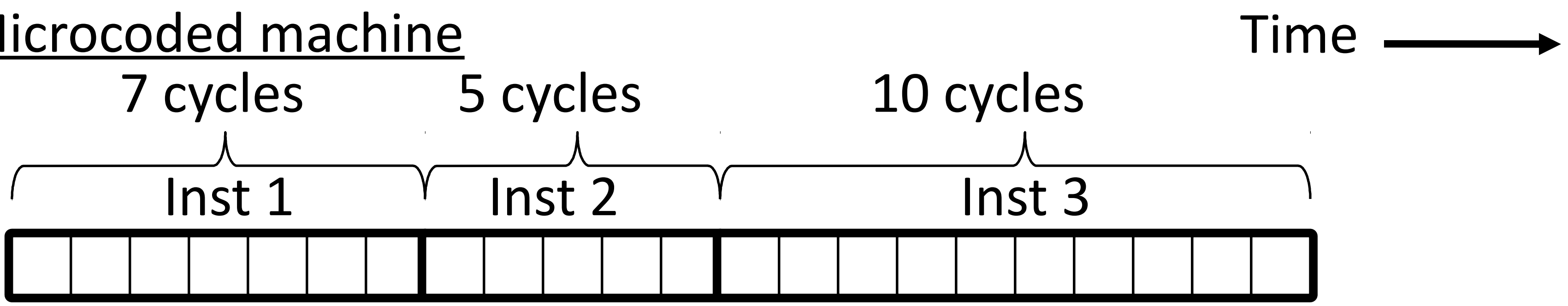
Classic 5-Stage RISC Pipeline



This version designed for regfiles/memories with synchronous reads and writes.

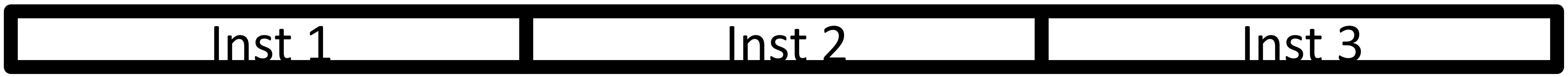
CPI Examples

Microcoded machine



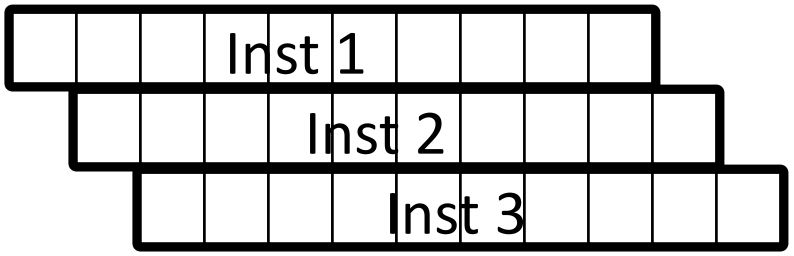
3 instructions, 22 cycles, CPI=7.33

Unpipelined machine



3 instructions, 3 cycles, CPI=1

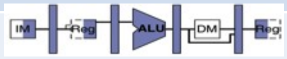
Pipelined machine



3 instructions, 3 cycles, CPI=1

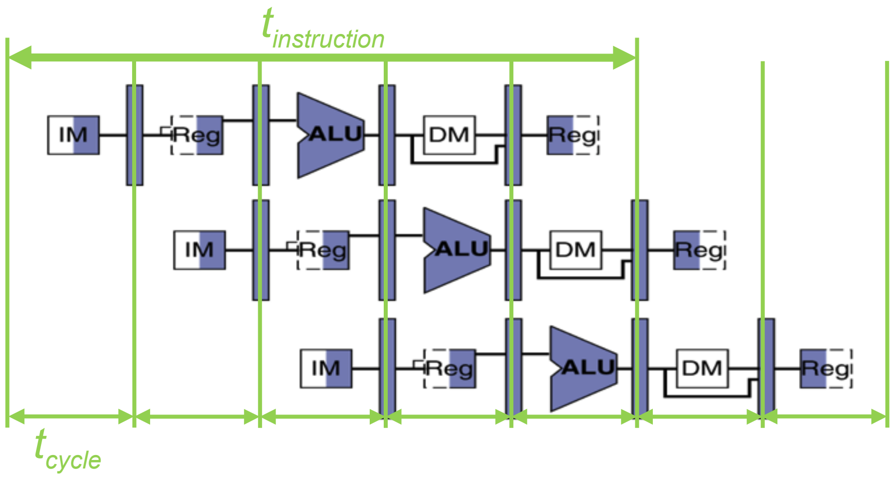
5-stage pipeline CPI≠5!!!

Pipelining with RISC-V

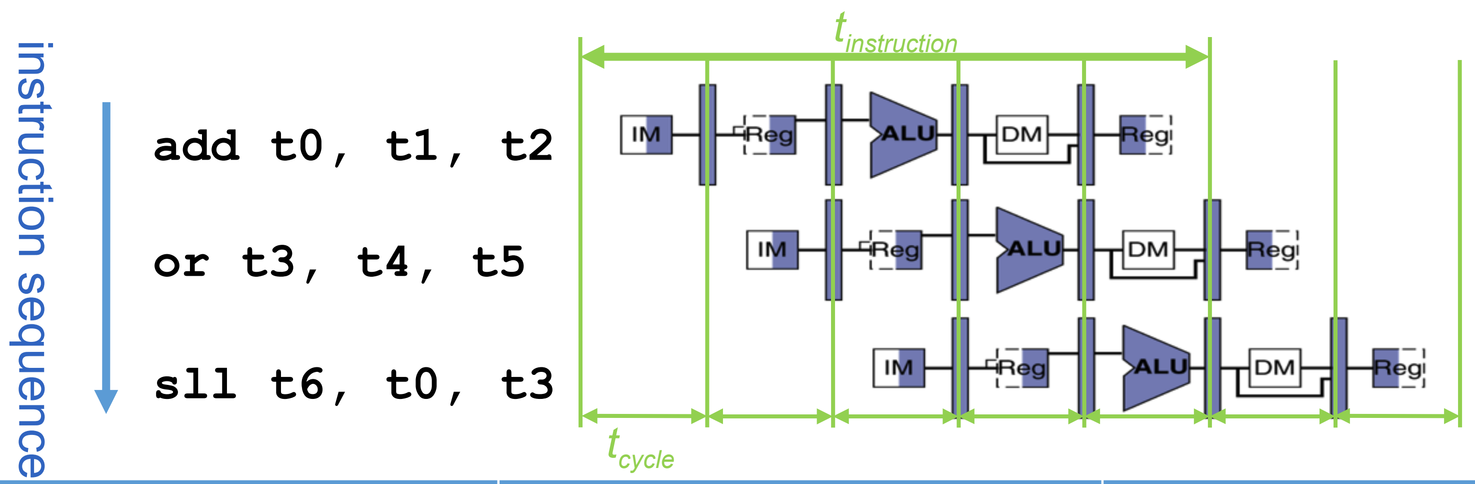
Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch	IM	200 ps	200 ps
Reg Read	Reg	100 ps	200 ps
ALU	ALU	200 ps	200 ps
Memory	DM	200 ps	200 ps
Register Write	Reg	100 ps	200 ps
$t_{instruction}$		**800 ps**	**1000 ps**

instruction sequence

```
add t0, t1, t2

or t3, t4, t5

sll t6, t0, t3
```



Pipelining with RISC-V

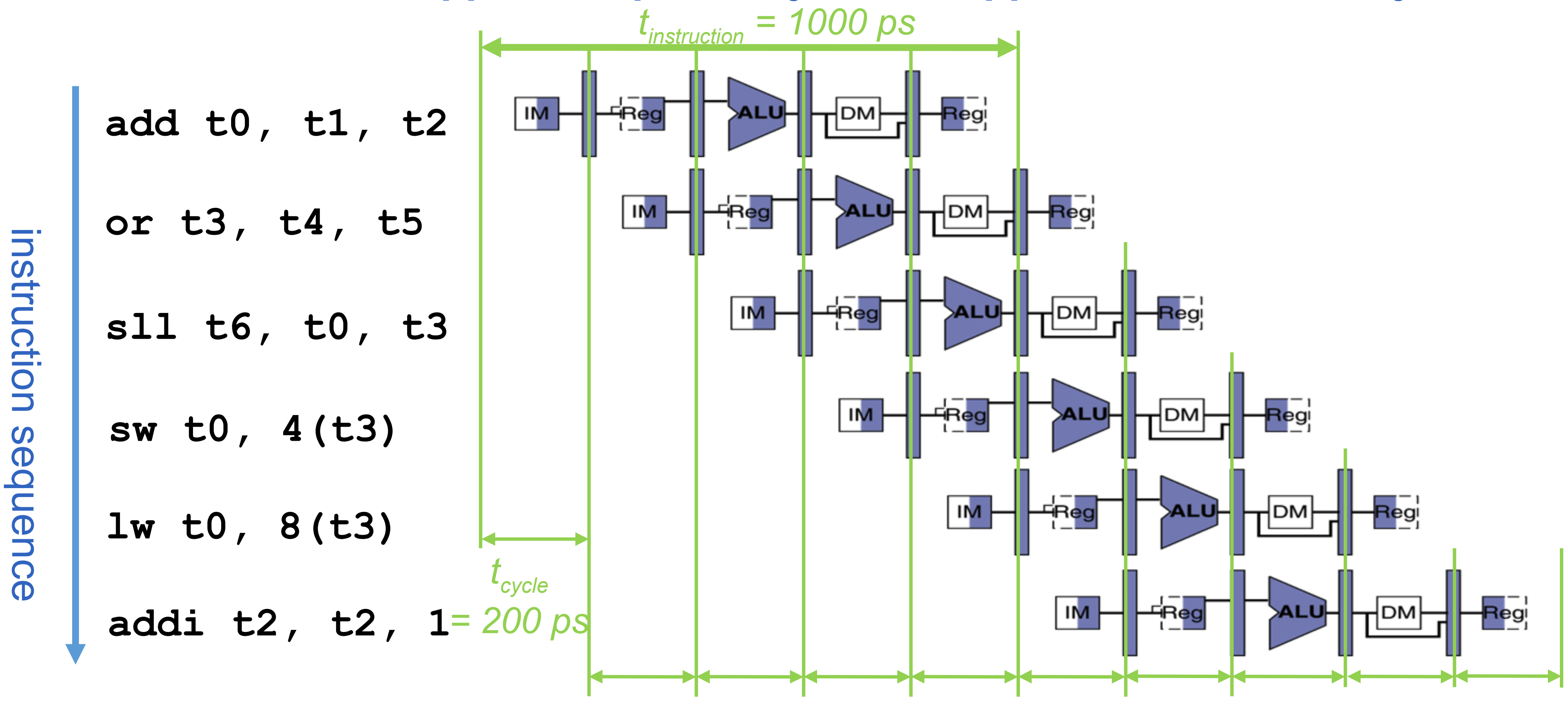


	Single Cycle	Pipelining
Timing	t_{step} = 100 … 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

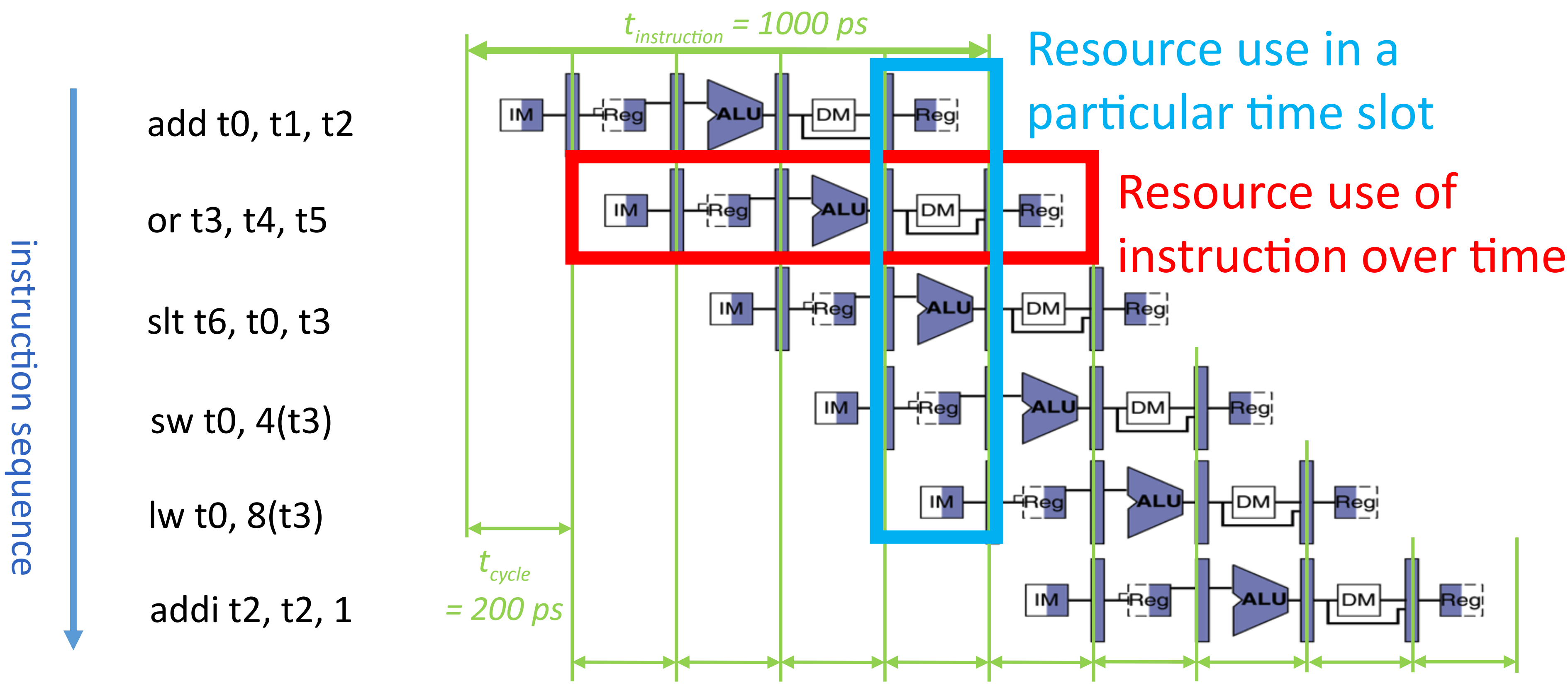
Sequential vs Simultaneous

What happens sequentially, what happens simultaneously?

```
add t0, t1, t2

or t3, t4, t5

sll t6, t0, t3

sw t0, 4(t3)

lw t0, 8(t3)

addi t2, t2, 1
```



RISC-V Pipeline



And in Conclusion, …

- Controller
  - Tells universal datapath how to execute each instruction
- Instruction timing
  - Set by instruction complexity, architecture, technology
  - Pipelining increases clock frequency, “instructions per second”
    - But does not reduce time to complete instruction
- Performance measures
  - Different measures depending on objective
    - Response time
    - Jobs / second
    - Energy per task