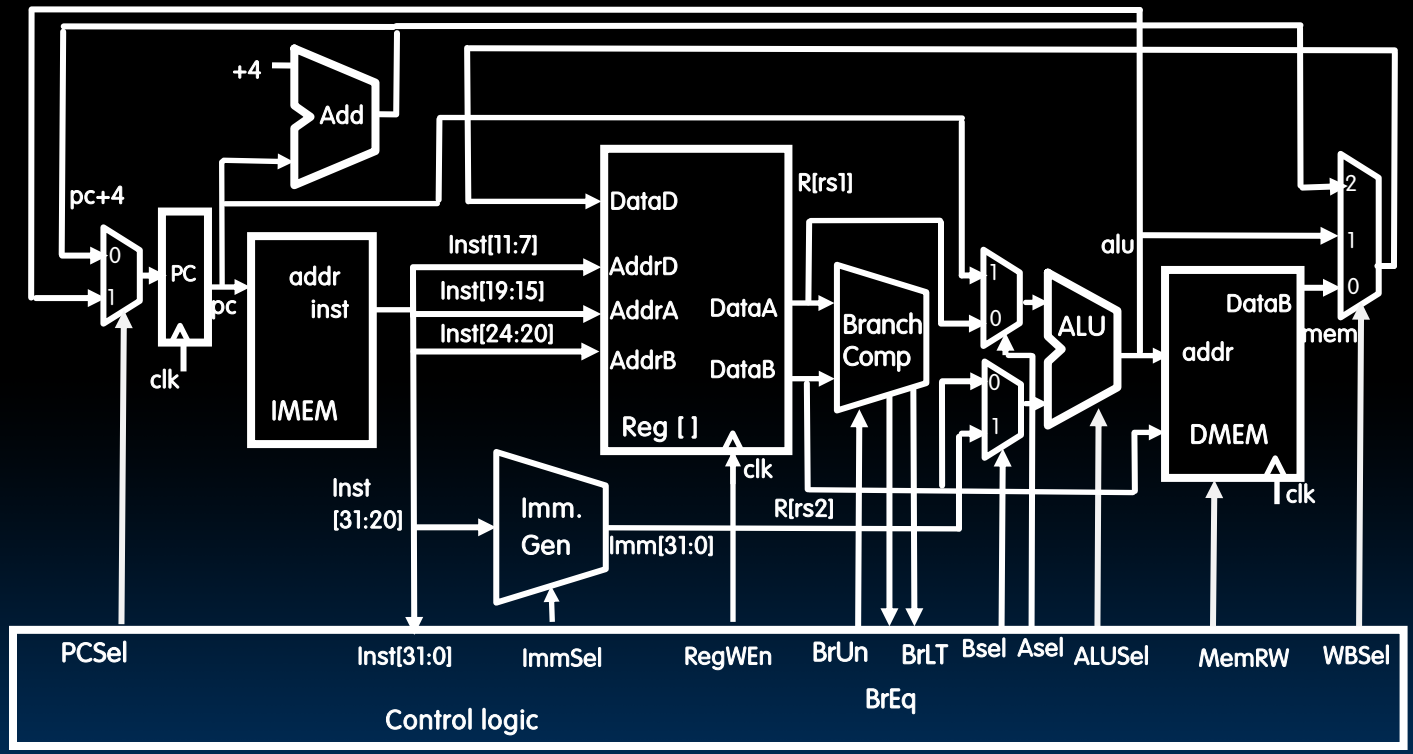
Control and Status Registers

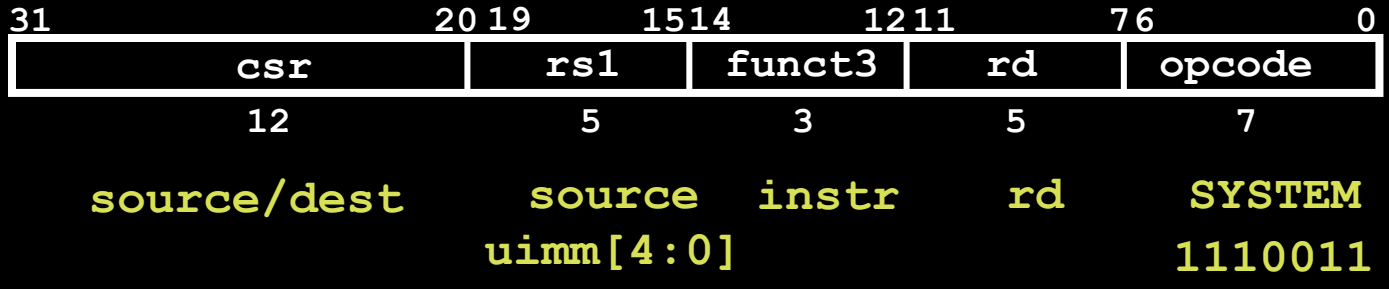
Complete Single-Cycle RV32I Datapath!



Control and Status Registers

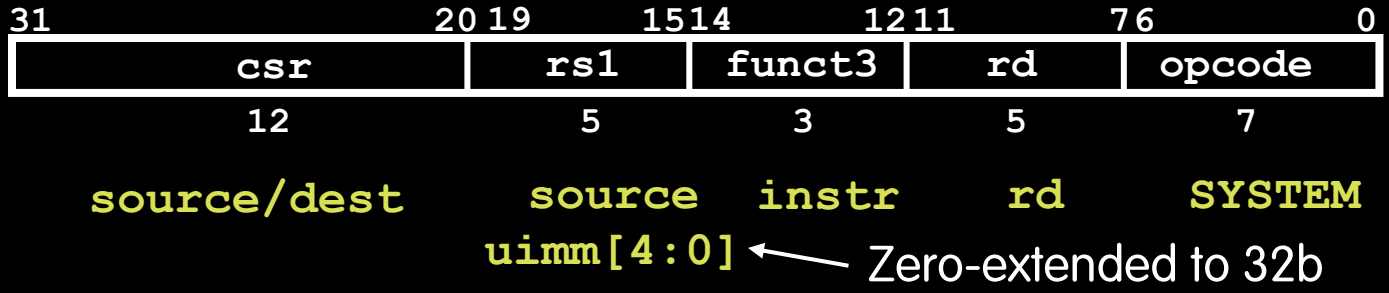
- Control and status registers (CSRs) are separate from the register file (**`x0`-`x31`**)
  - Used for monitoring the status and performance
  - There can be up to 4096 CSRs
- Not in the base ISA, but almost mandatory in every implementation
  - ISA is modular
  - Necessary for counters and timers, and communication with peripherals

CSR Instructions



Register operand				
Instr.	rd	rs	Read CSR?	Write CSR?
csrrw	**x0**	**–**	no	yes
csrrw	**!x0**	**–**	yes	yes
csrrs/c	**–**	**x0**	yes	no
csrrs/c	**–**	**!x0**	yes	yes

CSR Instructions



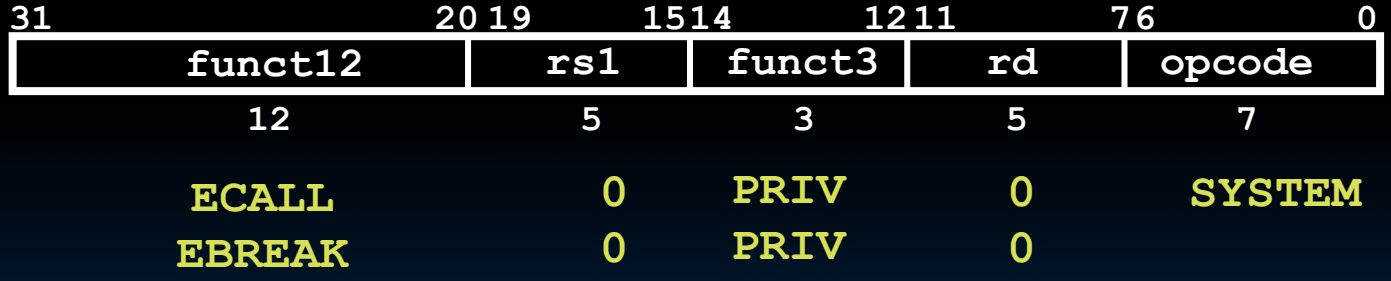
Immediate operand				
Instr.	rd	uimm	Read CSR?	Write CSR?
csrrwi	**x0**	**-**	no	yes
csrrwi	**!x0**	**-**	yes	yes
csrrs/ci	**-**	**0**	yes	no
csrrs/ci	**-**	**!0**	yes	yes

CSR Instruction Example

- The CSRRW (Atomic Read/Write CSR) instruction 'atomically' swaps values in the CSRs and integer registers.
  - We will see more on 'atomics' later
- CSRRW reads the previous value of the CSR and writes it to integer register **`rd`**. Then writes **`rs1`** to CSR
- Pseudoinstruction **`csrw csr, rs1`** is **`csrrw x0, csr, rs1`**
  - **`rd=x0`**, just writes **`rs1`** to CSR
- Pseudoinstruction **`csrwi csr, uimm`** is **`csrrwi x0, csr, uimm`**
  - **`rd=x0`**, just writes **`uimm`** to CSR
- Hint: Use write enable and clock…

System Instructions

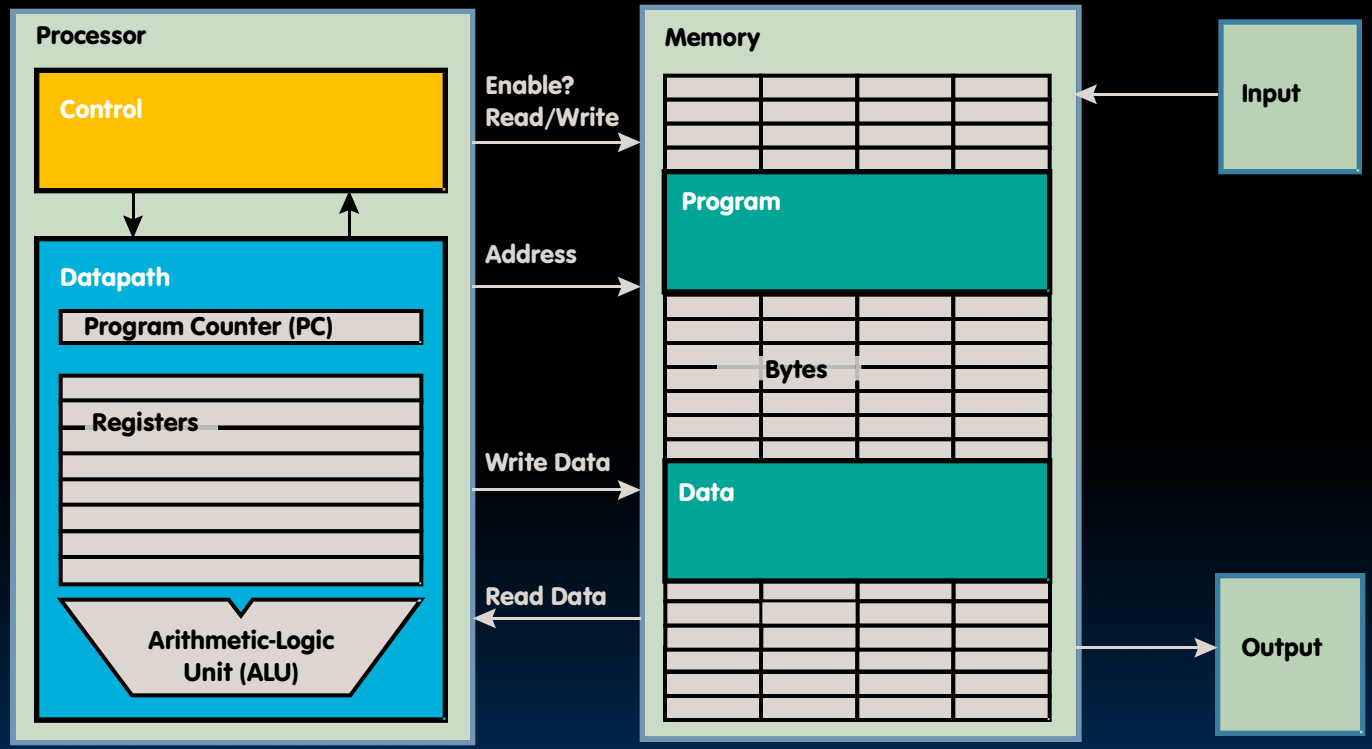
- **`ecall`** – (I-format) makes requests to supporting execution environment (OS), such as system calls (**`syscall`**s)
- **`ebreak`** – (I-format) used e.g. by debuggers to transfer control to a debugging environment



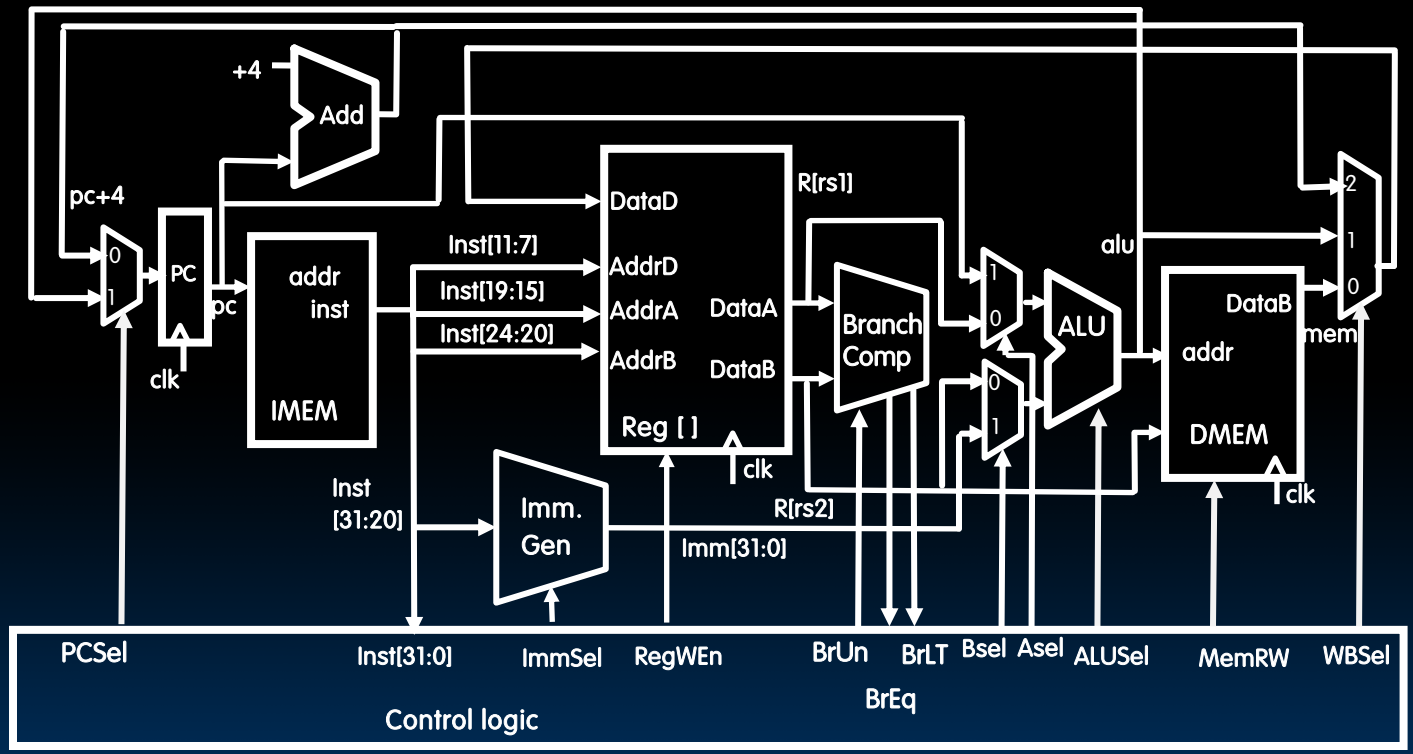
- **`fence`** – sequences memory (and I/O) accesses as viewed by other threads or co-processors

Datapath Control

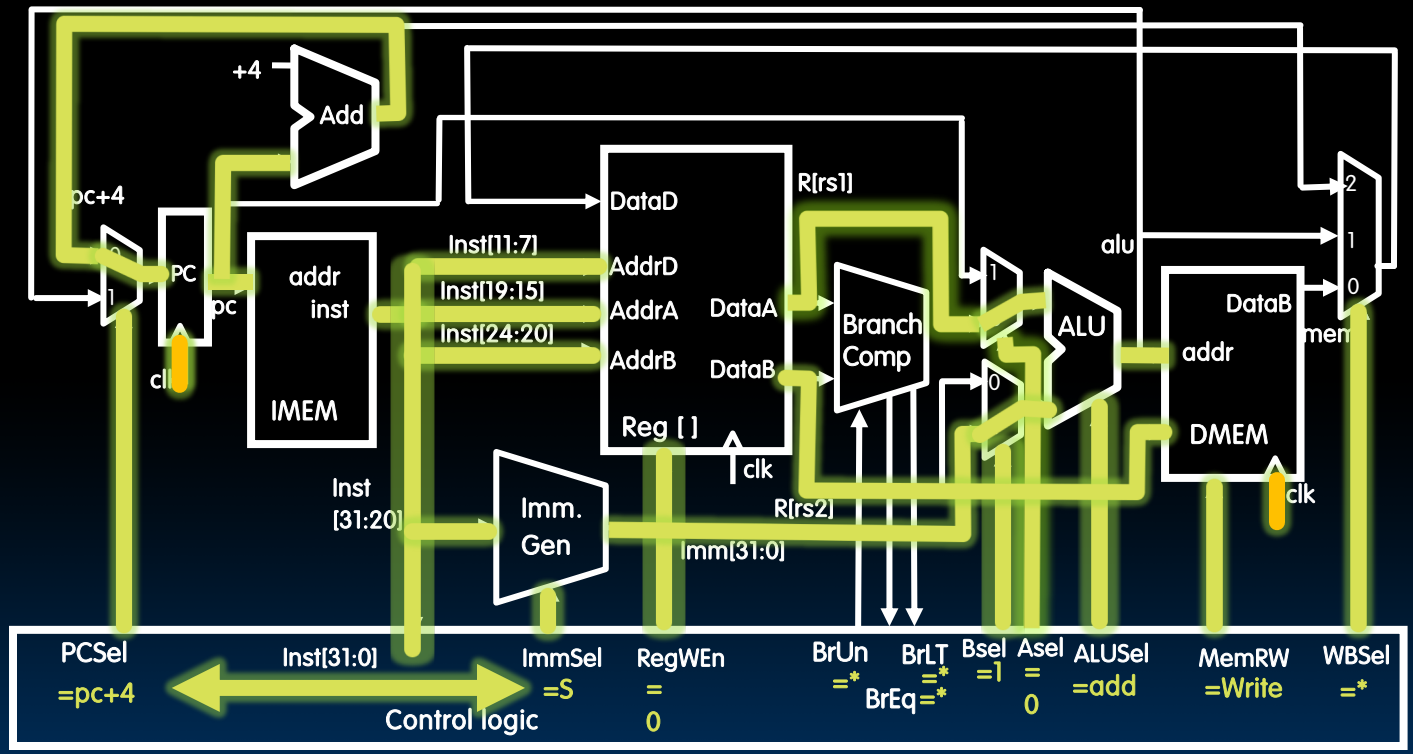
Our Single-Core Processor



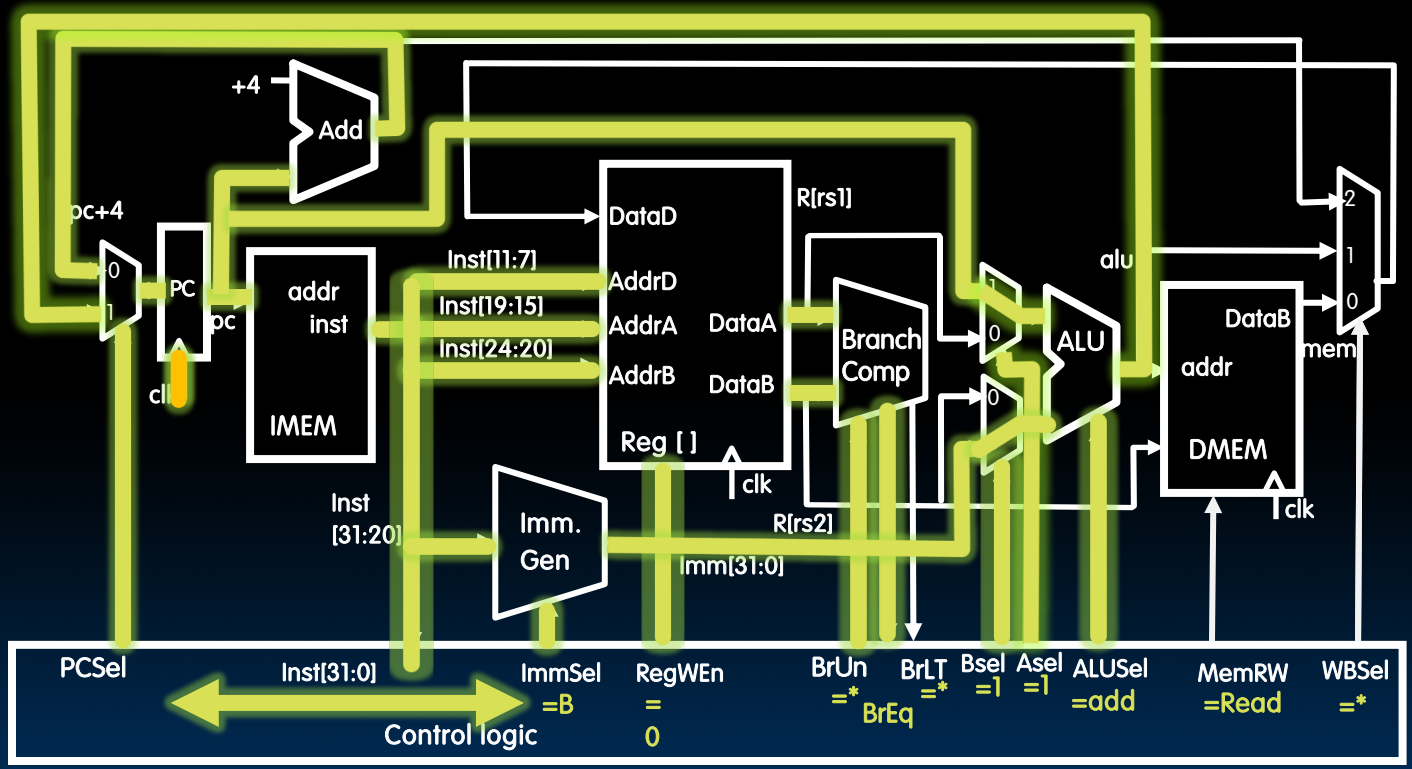
Single-Cycle RV32I Datapath and Control



Example: sw

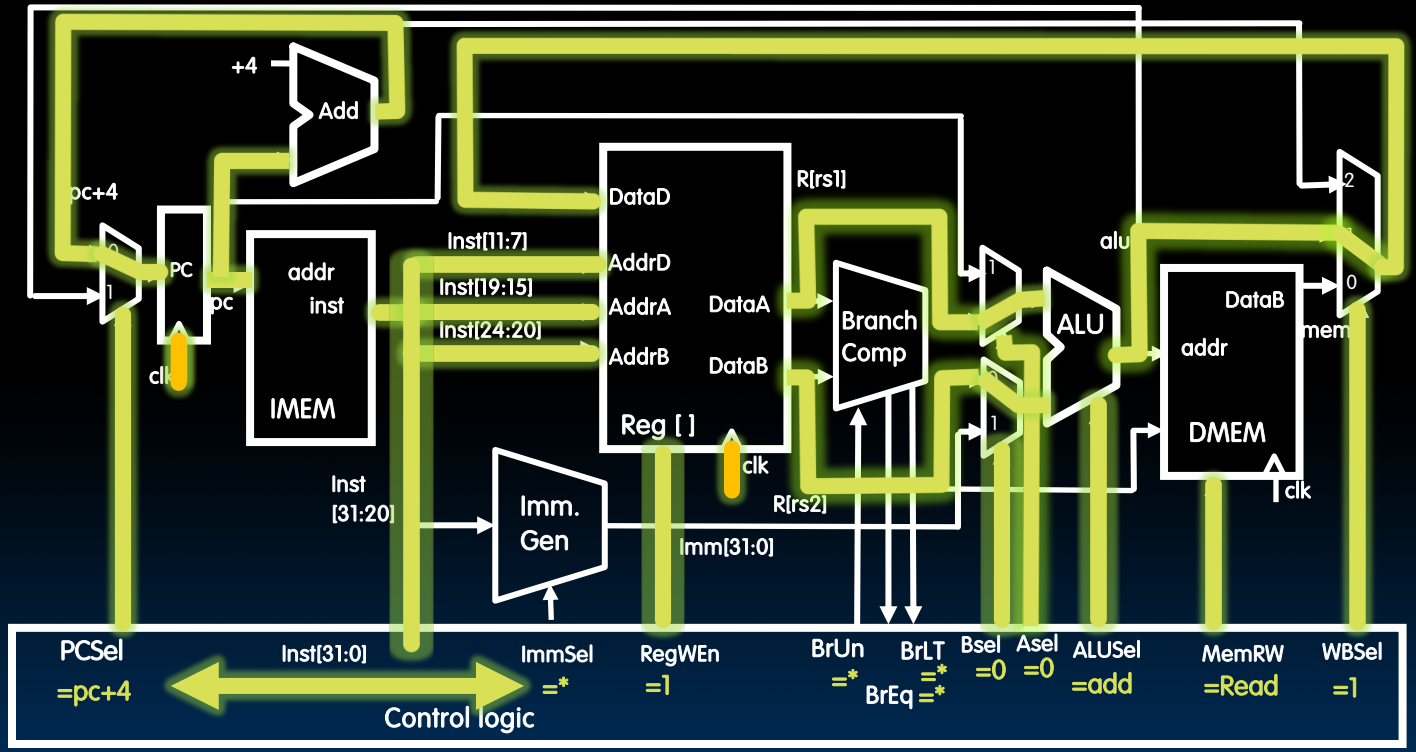


Example: beq

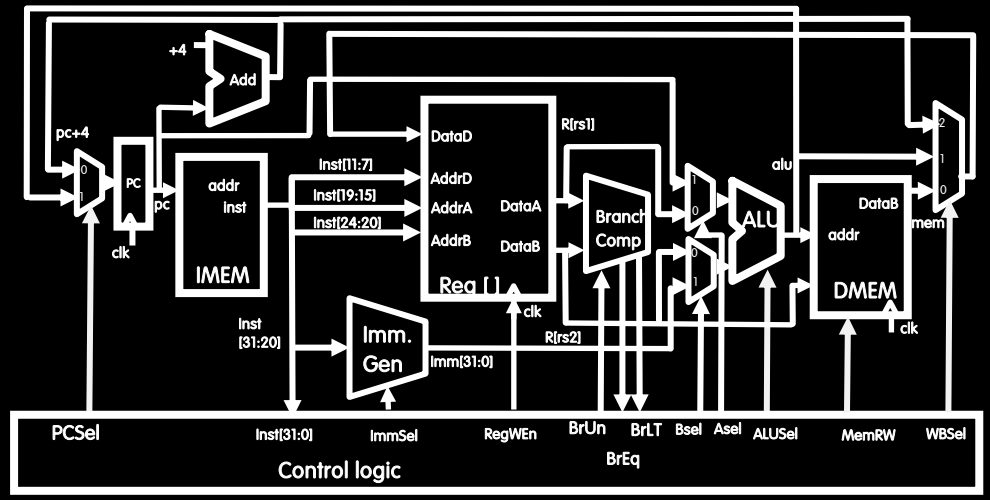


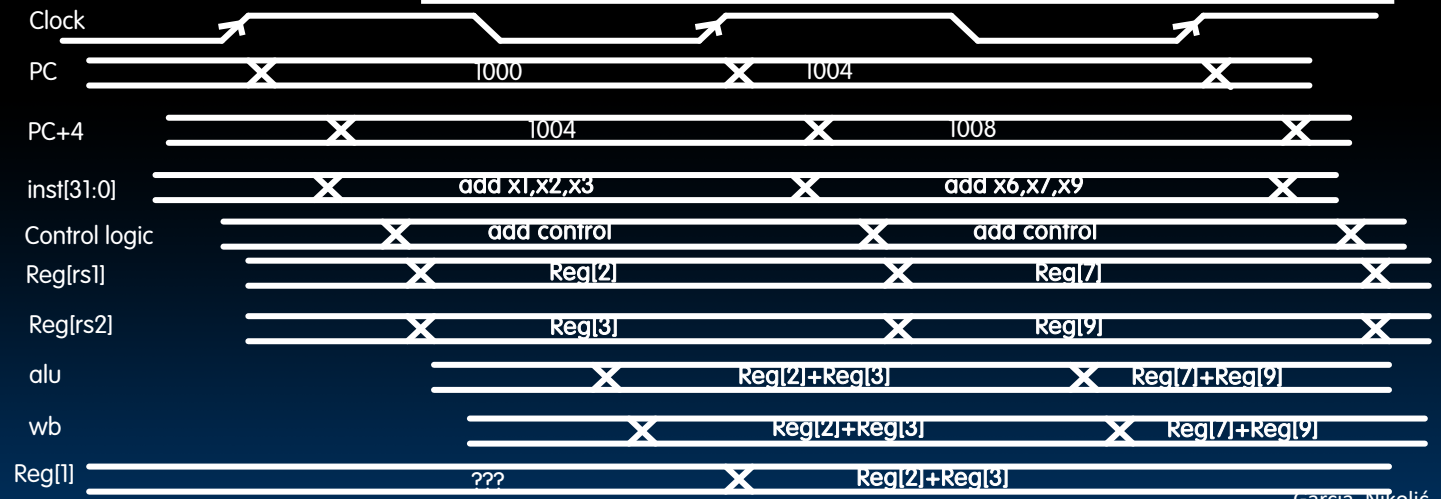
Instruction Timing

Example: add



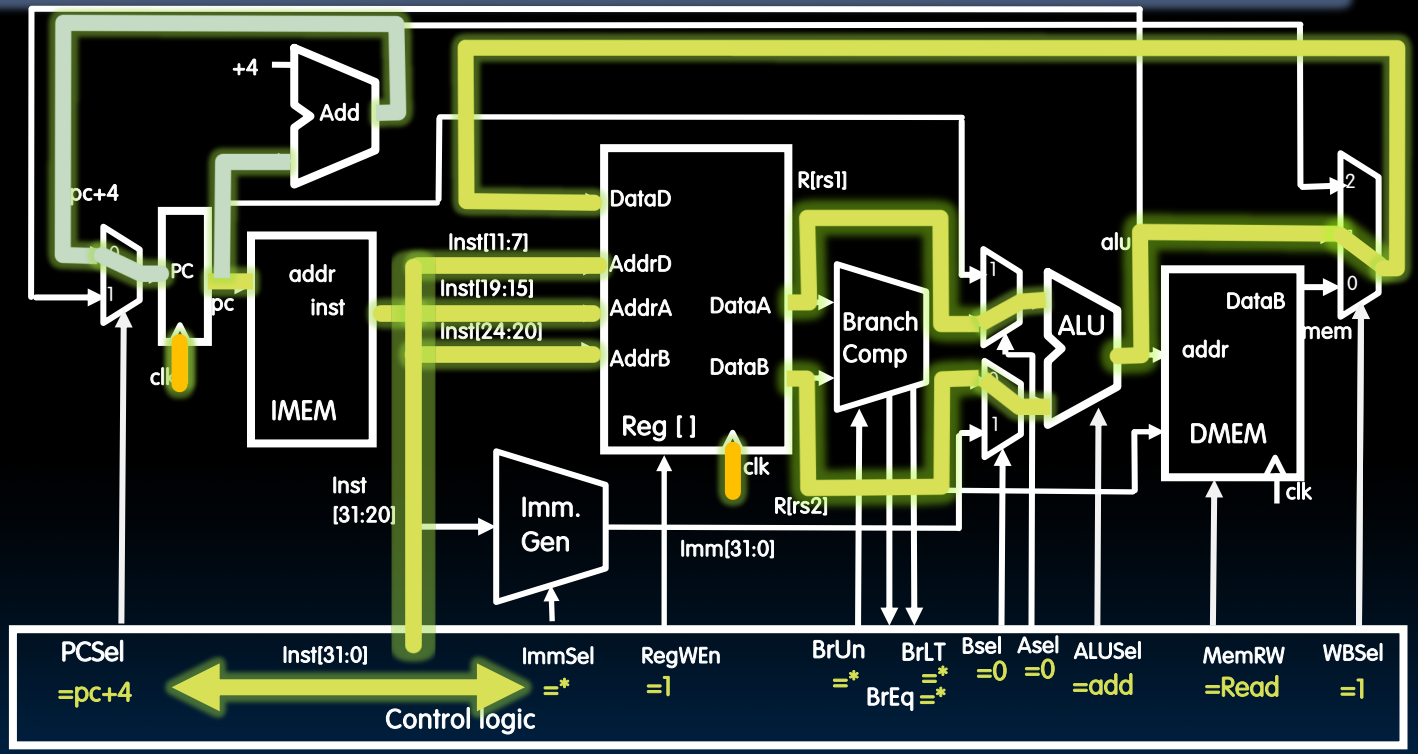
Add Execution





García, Nikolić

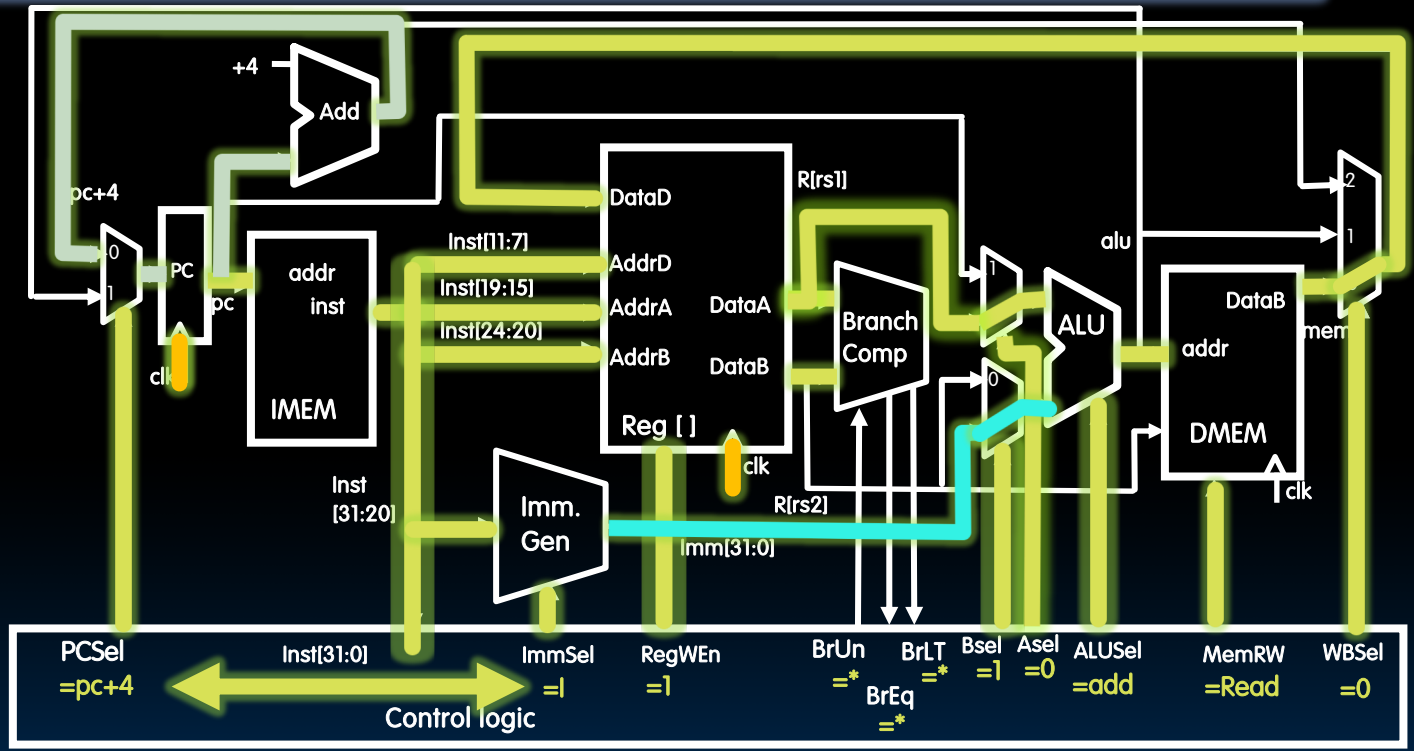
Example: add timing



Critical path = $t_{clk\text{-}q} + \max\{t_{Add}+t_{mux}, t_{IMEM}+t_{Reg}+t_{mux}+t_{ALU}+t_{mux}\}+t_{setup}$

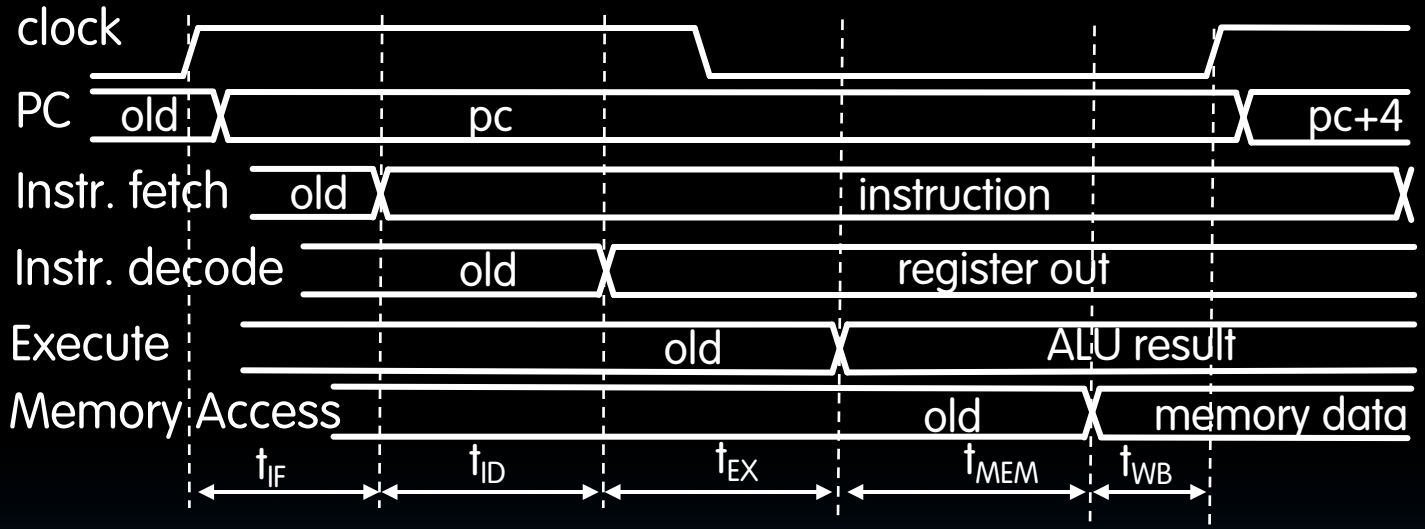
$= t_{clk\text{-}q} + t_{IMEM}+t_{Reg}+t_{mux}+t_{ALU}+t_{mux}+t_{setup}$

Example: lw



Critical path = $t_{clk\text{-}q} + \max\{t_{Add}+t_{mux},\ t_{IMEM}+t_{Imm}+t_{mux}+t_{ALU}+t_{DMEM}+t_{mux},\ t_{IMEM}+t_{Reg}+t_{mux}+t_{ALU}+t_{DMEM}+t_{mux}\}+t_{setup}$

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		X	600ps
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!

Control Logic Design

Control Logic Truth Table

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
`add`	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
`sub`	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
`(R-R Op)`	*	*	+4	*	*	Reg	Reg	*(Op)*	Read	1	ALU
`addi`	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
`lw`	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
`sw`	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
`beq`	0	*	+4	B	*	PC	Imm	Add	Read	0	*
`beq`	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
`bne`	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
`bne`	1	*	+4	B	*	PC	Imm	Add	Read	0	*
`blt`	*	1	ALU	B	0	PC	Imm	Add	Read	0	*
`bltu`	*	1	ALU	B	1	PC	Imm	Add	Read	0	*
`jalr`	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
`jal`	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
`auipc`	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU

Control Realization Options

- ROM
  - "Read-Only Memory"
  - Regular structure
  - Can be easily reprogrammed
    - fix errors
    - add instructions
  - Popular when designing control logic manually
- Combinatorial Logic
  - Today, chip designers use logic synthesis tools to convert truth tables to networks of gates

RV32I, A Nine-Bit 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
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

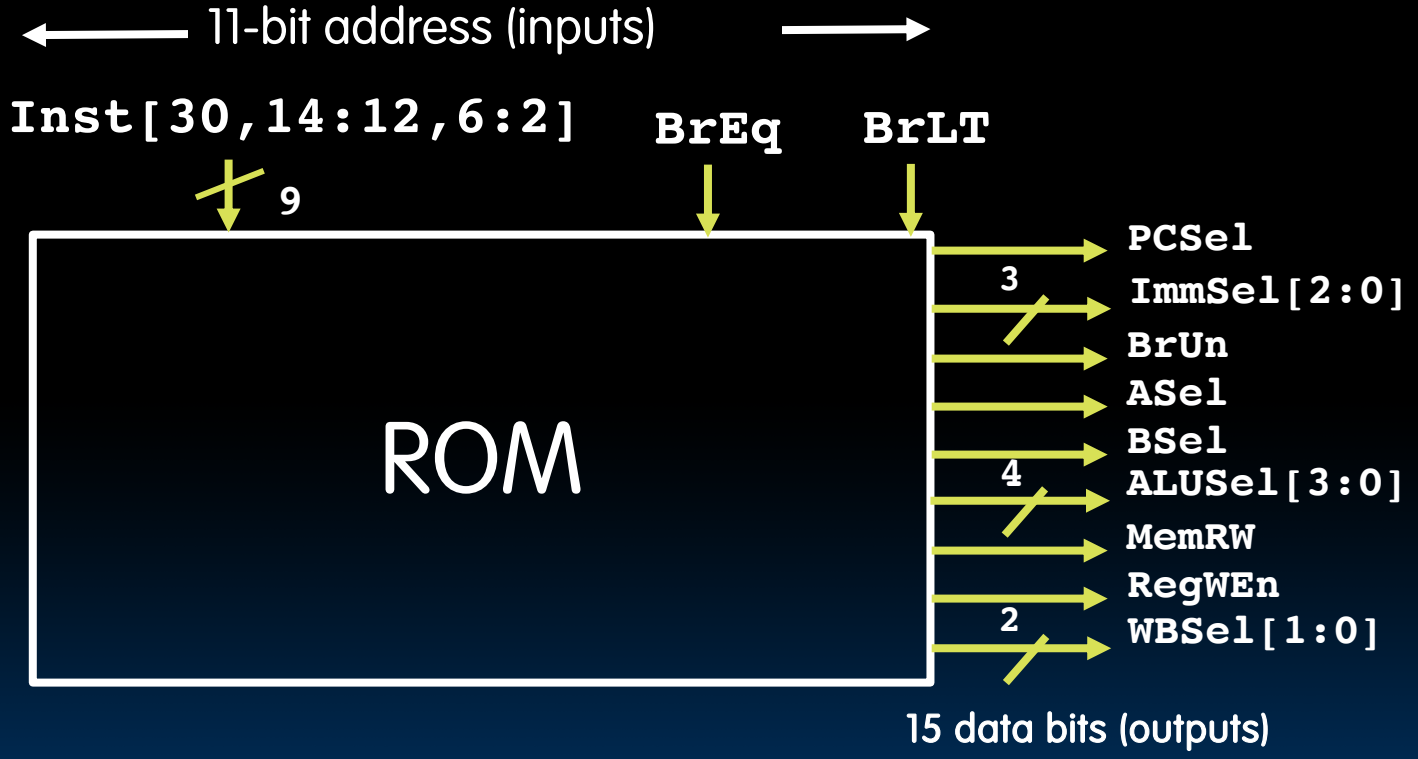
- Instruction type encoded using only 9 bits:
- **`inst[30]`, `inst[14:12]`, `inst[6:2]`**

`inst[6:2]`

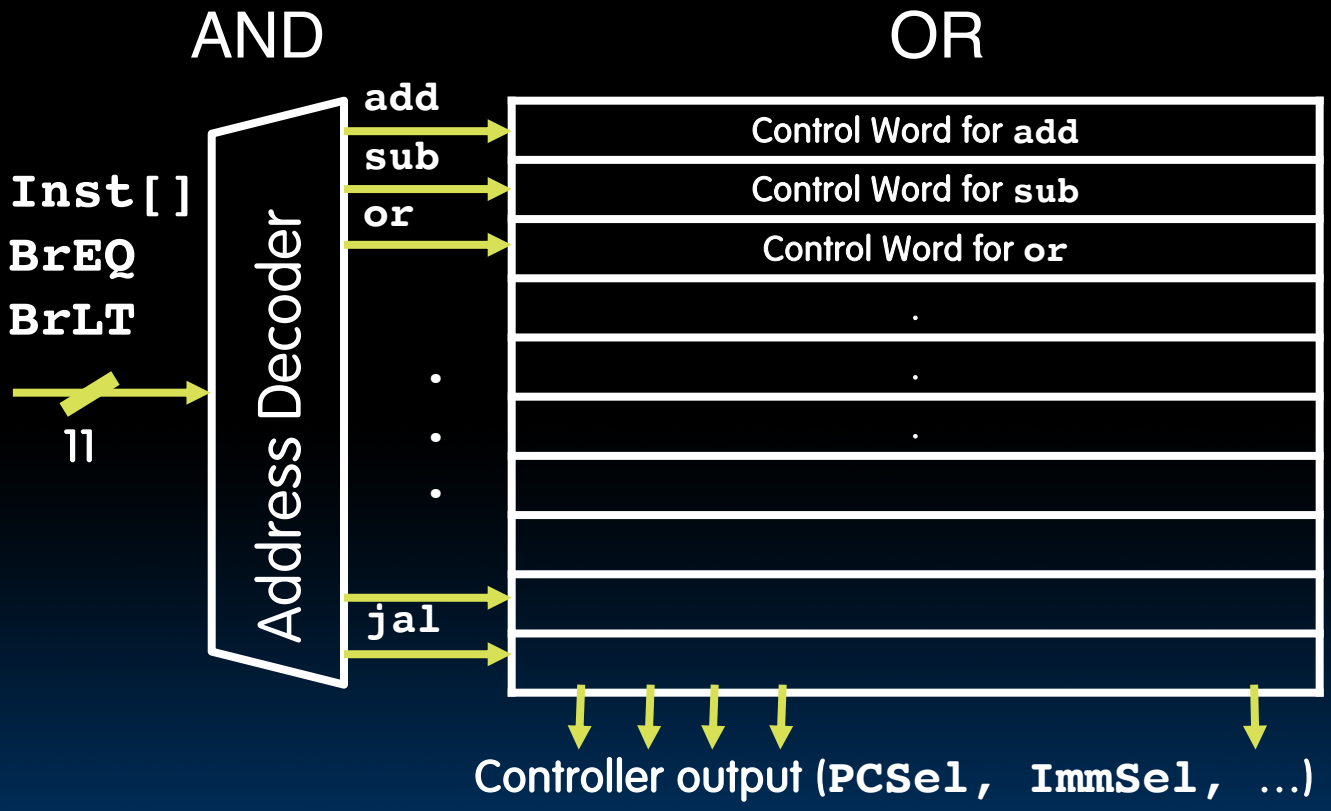
`inst[14:12]`

`inst[30]`

ROM-based Control

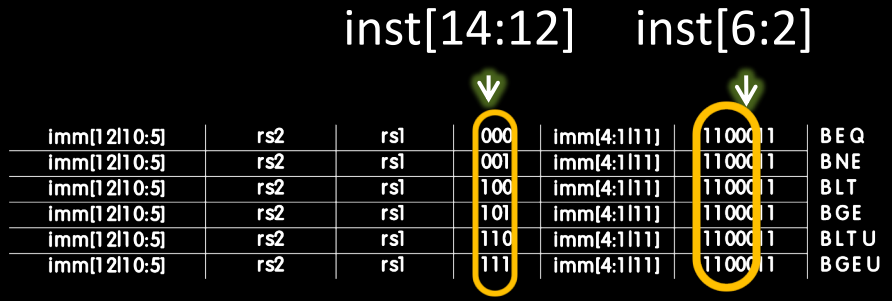


ROM Controller Implementation



Combinational Logic Control

- Simplest example: **`BrUn`**



- How to decode whether BrUn is 1?

`BrUn = Inst [13] • Branch`

Control Logic to Decode add

inst[30] inst[14:12] inst[6:2]

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

$$\text{add} = \overline{i[30]} \cdot \overline{i[14]} \cdot \overline{i[13]} \cdot \overline{i[12]} \cdot \text{R-type}$$

$$\text{R-type} = \overline{i[6]} \cdot i[5] \cdot i[4] \cdot \overline{i[3]} \cdot \overline{i[2]} \cdot \text{RV32I}$$

$$\text{RV32I} = i[1] \cdot i[0]$$

“And In Conclusion…”

Call home, we've made HW/SW contact!

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

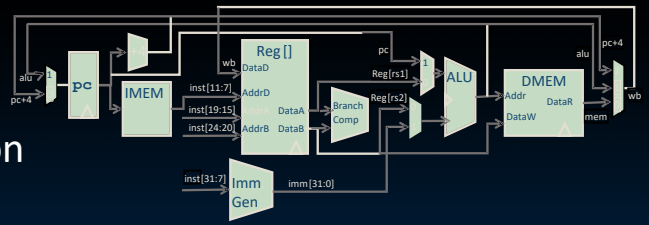
```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Assembler

Machine Language Program (RISC-V)

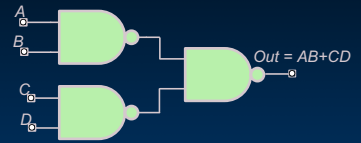
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



"And In conclusion..."

- We have built a processor!
  - Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions
  - Critical path changes
- 5 Phases of execution
  - IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases
- Controller specifies how to execute instructions
  - Implemented as ROM or logic

UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture (a.k.a. Machine Structures)

UC Berkeley
Professor
Bora Nikolić

Pipelining

New-School Machine Structures

Software

Parallel Requests
Assigned to computer
e.g., Search "Cats"

Parallel Threads
Assigned to core e.g., Lookup, Ads

Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions

Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words

Hardware descriptions
All gates work in parallel at same time

Harness Parallelism & Achieve High Performance

Hardware

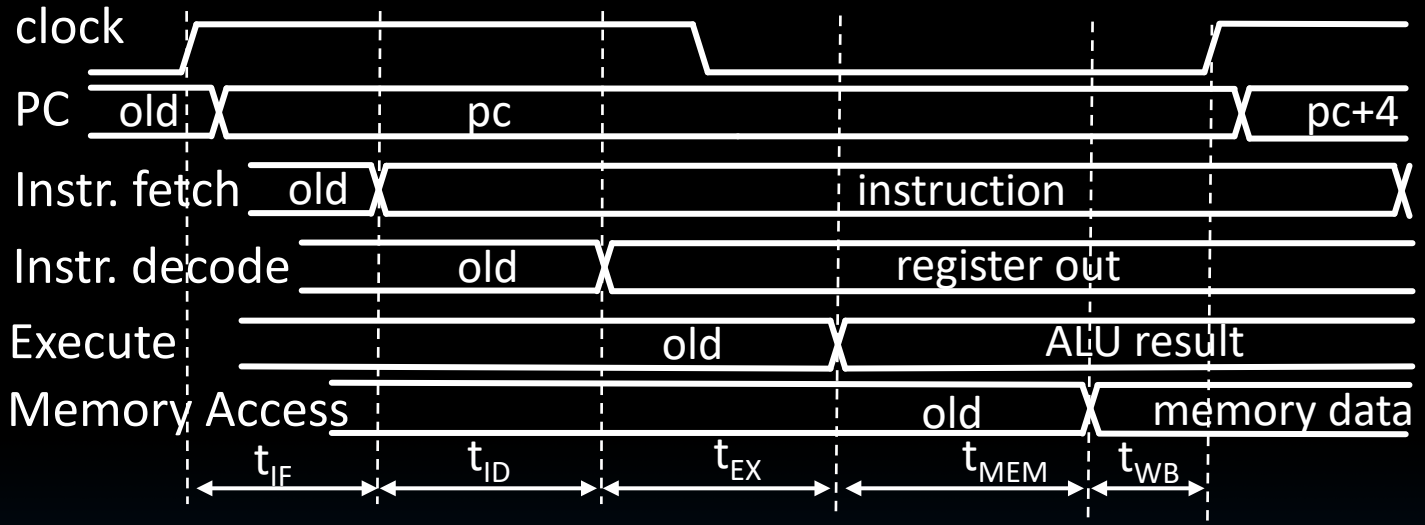
Warehouse Scale Computer

Smart Phone

6 Great Ideas in Computer Architecture

1. Abstraction (Layers of Representation/Interpretation)
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy

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		X	600ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

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

Performance Measures

- "Our" Single-cycle RISC-V CPU 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, spoken words recognized)?
    - 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 (assume they return instantaneously)

	Sports Car	Bus
Travel Time	15 min	60 min
Time for 100 passengers	750 min (50 2-person trips)	120 min (two 50-person trips)
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

Processor Performance Iron Law

"Iron Law" of Processor Performance

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

CPI = Cycles Per Instruction

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 (CPI)

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 lectures)

Time per Cycle (1/Frequency)

Determined by

- Processor microarchitecture (determines critical path through logic gates)
- Technology (e.g. 5nm versus 28nm)
- Power budget (lower voltages reduce transistor speed)

Speed Tradeoff Example

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

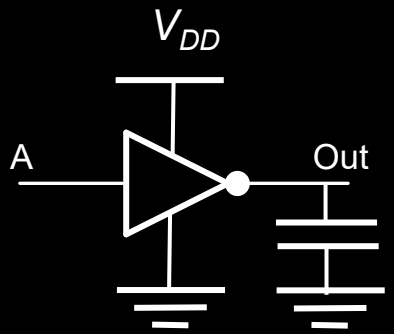
	Processor A	Processor B
# Instructions	1 Million	1.5 Million
Average CPI	2.5	1
Clock rate *f*	2.5 GHz	2 GHz
Execution time	1 ms	0.75 ms

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

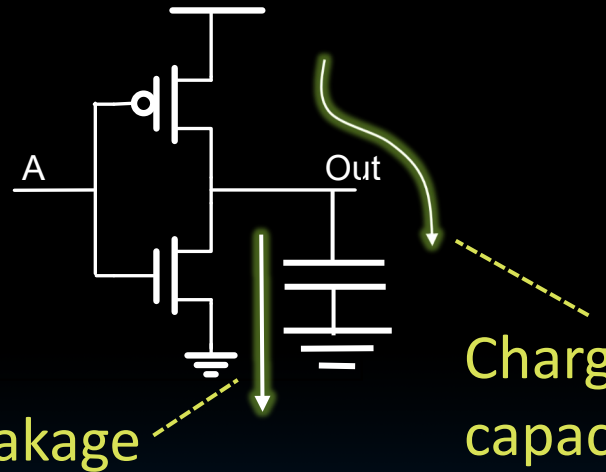
Energy Efficiency

Where Does Energy Go in CMOS?

Symbol (INV)



Schematic



(30%)

(70%)

Energy per Task

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

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

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

Supply voltage, e.g. 1V

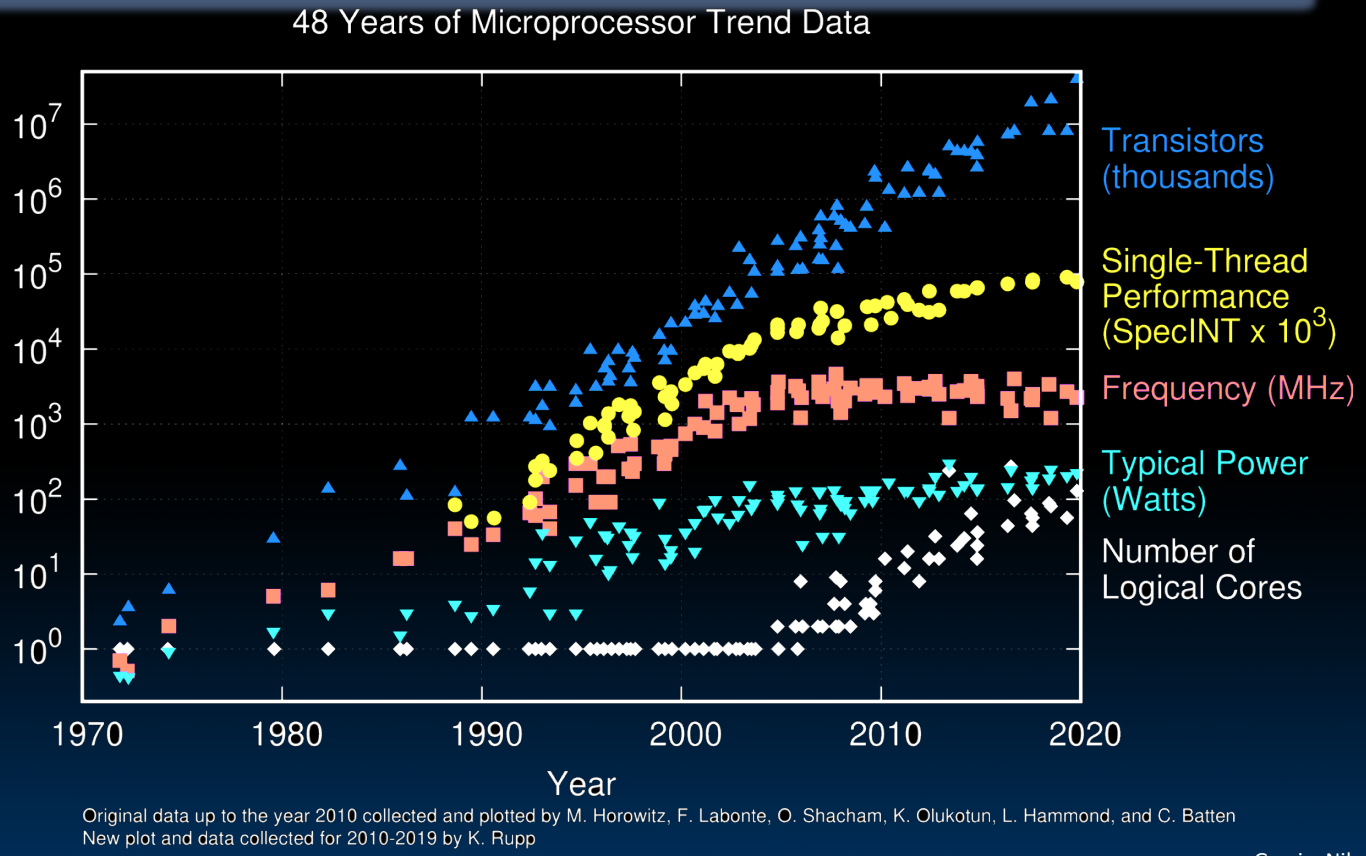
Want to reduce capacitance and voltage to reduce energy/task

Energy Tradeoff Example

- "Next-generation" processor
  - C (Moore's Law): -15 %
  - Supply voltage, V_{sup}: -15 %
  - Energy consumption: $0 - (1-0.85^3) = -39\ \%$

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

Performance/Power Trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

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
  - Need to go to 3D
- Power becomes a growing concern – the “power wall”

Energy "Iron Law"

Performance = Power * Energy Efficiency

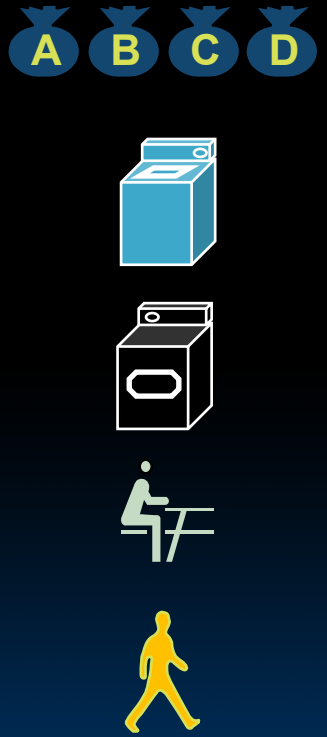
(Tasks/Second) (Joules/Second) (Tasks/Joule)

- 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

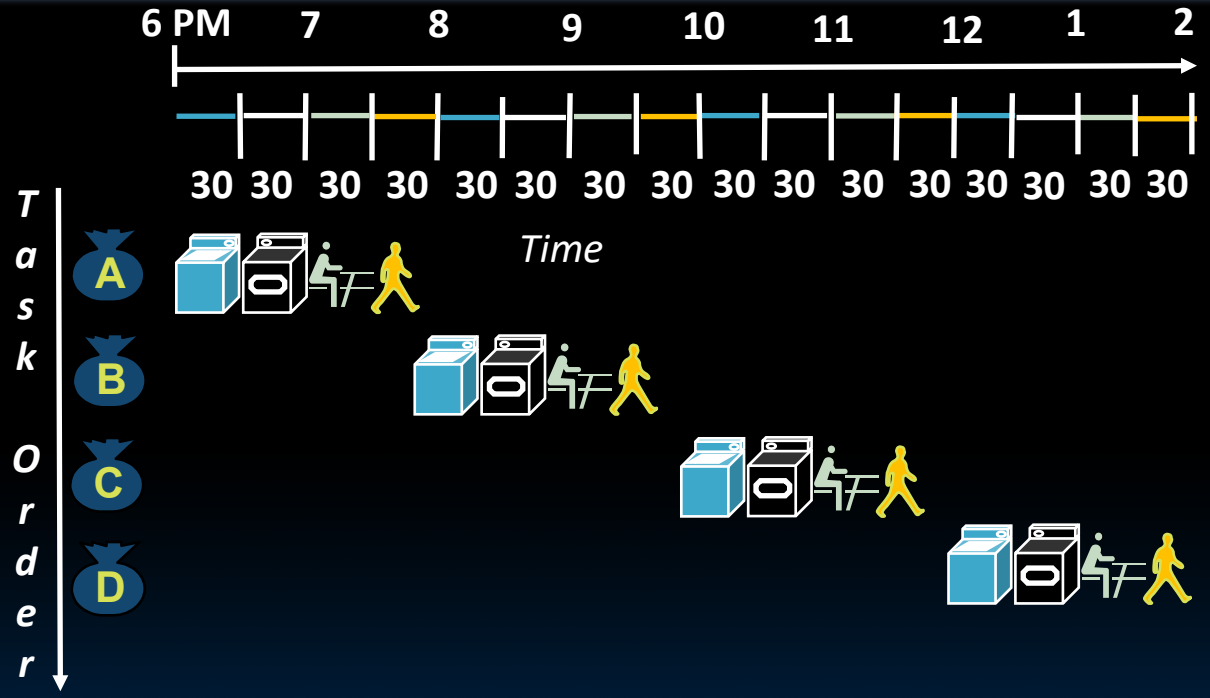
Introduction to Pipelining

Gotta Do Laundry

- Avi, Bora, Caroline, Dan each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers

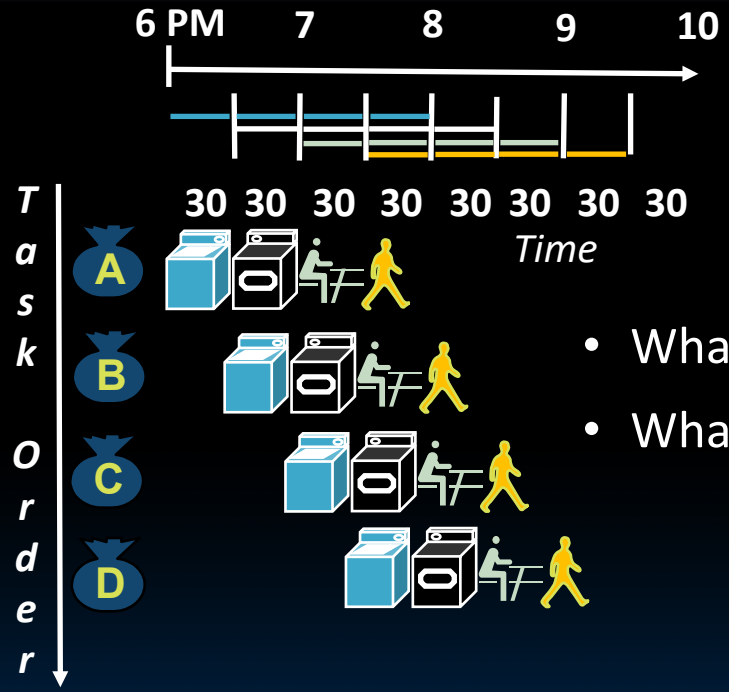


Sequential Laundry



Sequential laundry takes 8 hours for 4 loads!

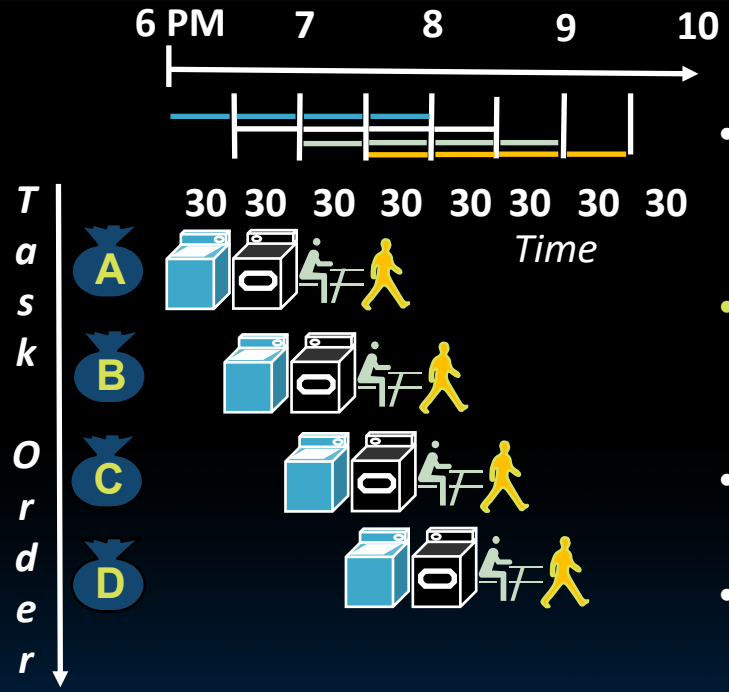
Pipelined Laundry



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

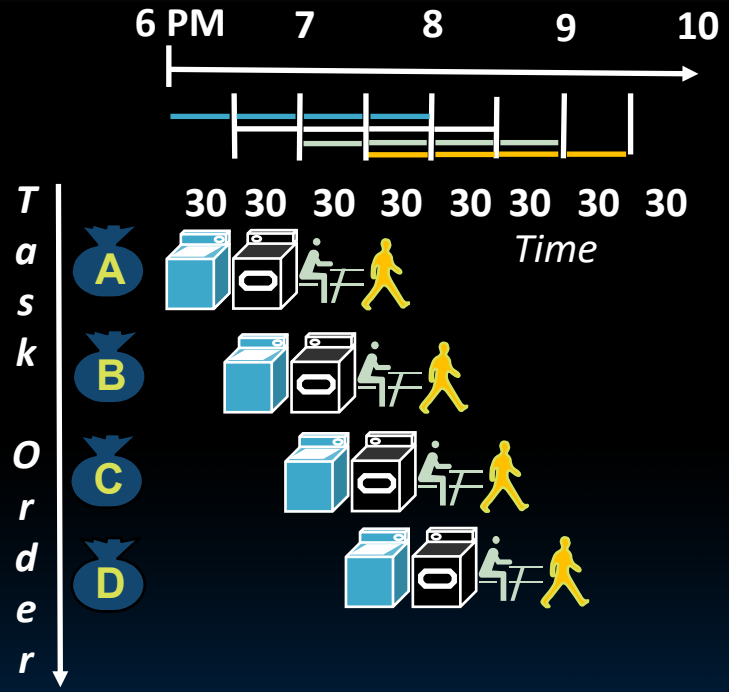
Pipelined laundry takes 3.5 hours for 4 loads!

Sequential Laundry



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number of pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup:
  2.3X v. 4X in this example

Sequential Laundry



Suppose:
- new Washer takes 20 minutes
- new Stasher takes 20 minutes.

How much faster is pipeline?

Pipeline rate limited by slowest pipeline stage

Unbalanced lengths of pipe stages reduce speedup