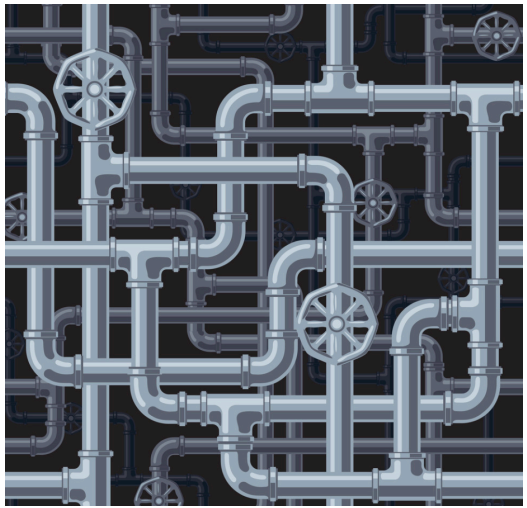
Great Ideas in Computer Architecture

RISC-V CPU Control, Pipelining

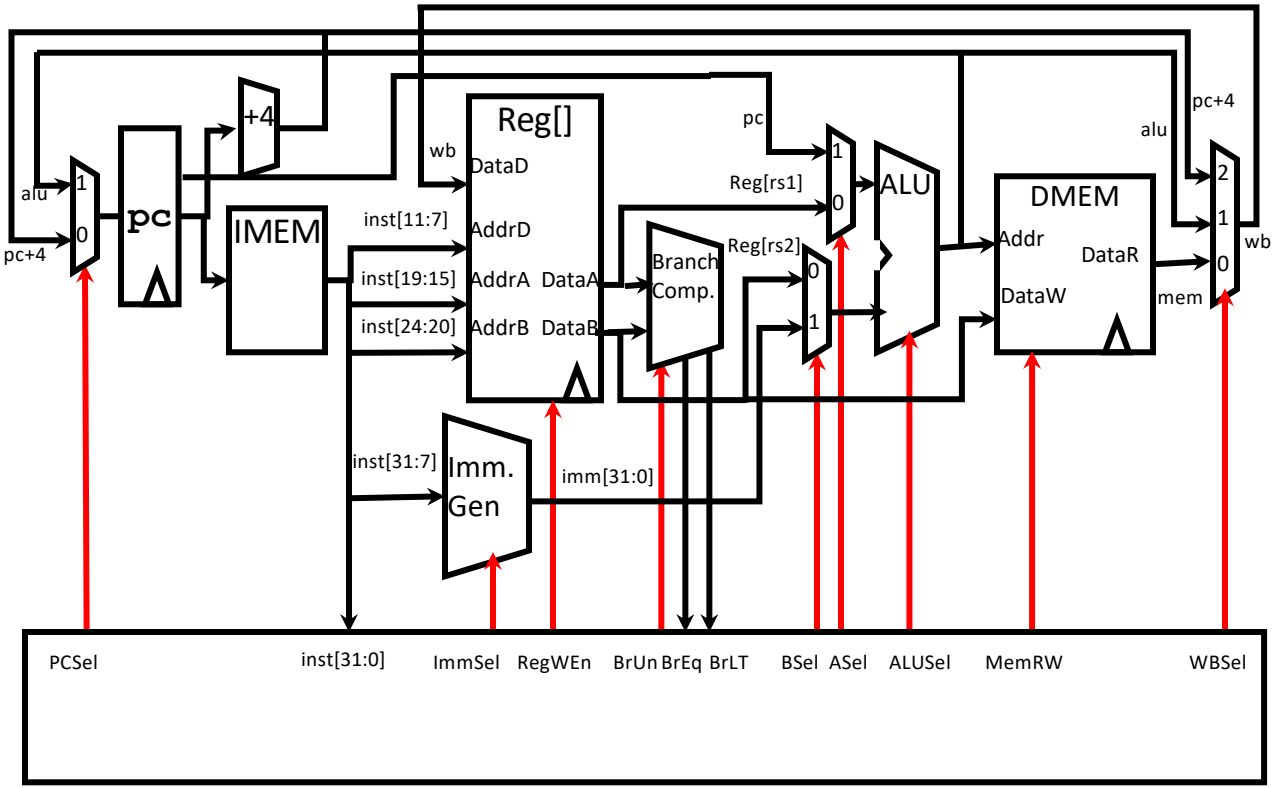
Instructor: Jenny Song



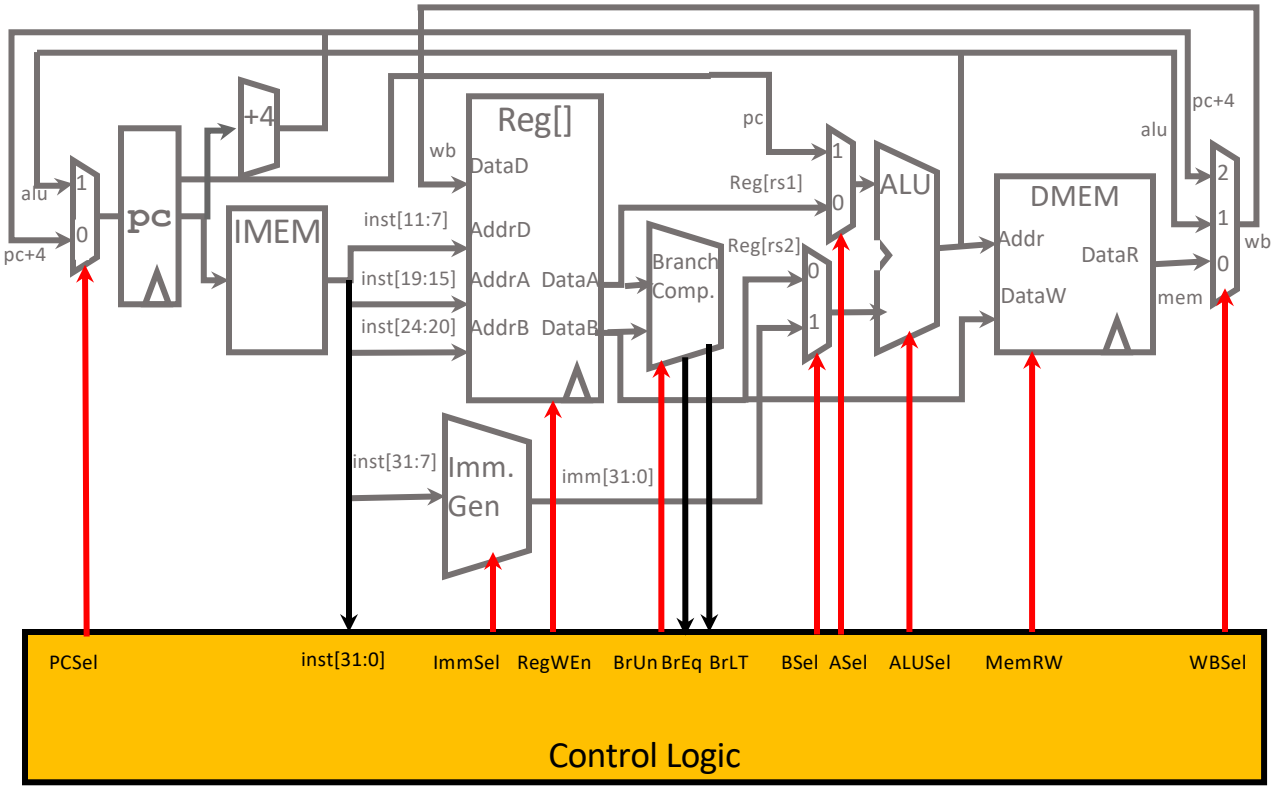
Agenda

- Datapath Review
- Control Implementation
- Administrivia
- Performance Analysis
- Pipelined Execution
- Pipelined Datapath

Single-Cycle RISC-V RV32I Datapath



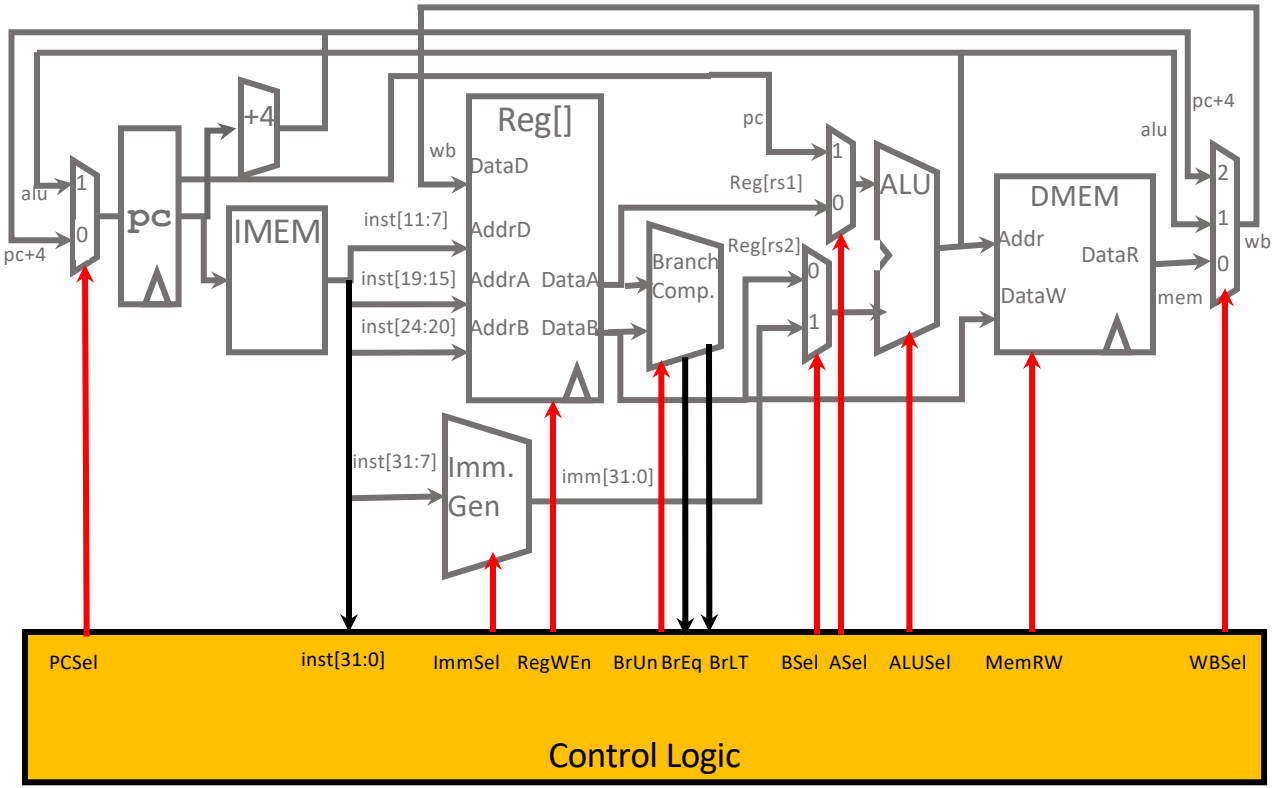
Single-Cycle RISC-V RV32I Datapath



Agenda

- Quick Datapath Review
- Control Implementation
- Performance Analysis
- Pipelined Execution
- Pipelined Datapath

Control bits



Our Control Bits

- PCSel
  - Does this instruction change my control flow?
  - What is the address of my next instruction?
- ImmSel
  - Does this instruction have/use an immediate?
    - If yes, what type of instruction is it? How is the immediate stored?
- RegWEn
  - Does this instruction write to the destination register rd?
- BrUn
  - Does this instruction do a branch? If so, is it unsigned or signed?

Our Control Bits

- BSel
  - Does this instruction operate on R[rs2] or an immediate?
- ASel
  - Does this instruction operate on R[rs1] or PC?
- ALUSel
  - What operation should we perform on the selected operands?
- MemRW
  - Do we want to write to memory? Do we want to read from memory?
    - If we don't care about the memory output, what should we do?
- WBSel
  - Which value do we want to write back to rd?
    - If we aren't writing back (RegWEn = 0) does this value matter?

Designing Control Signals

- The majority of project 3!
  - So we won't give the solutions out in lecture ;)
- Questions you should ask:
  - Is this control signal the same for every instruction of the same type? (I, R, S, SB, etc.) If so, can you use a combination of opcode/funct3/funct7 to encode the value?
  - Is this control signal dependent on *other* controls?
    - ie. PCSel and BrEq, BrLT
  - Does the value of this control signal alter the execution of the instruction?
    - Some cases: yes! (MemRW, for example)
    - Some cases: no! (ImmSel in R-type inst, for example)

Let’s try an example!

Design PCSel yourself!

Might help to split it into three cases:

- Regular (non-control) instructions
- Branches
- Jumps

You may assume PCSel = 0 → PC = PC + 4, and PCSel = 1 → PC = ALUout

PCSel: Regular Instructions

- Assumption: PCSel = 0 → PC = PC + 4, and PCSel = 1 → PC = ALUout
  - This isn't the case in every datapath! How can you check? Look at what the PC-input MUX maps 0 and 1 to. Its controlled by PCSel!
  - For regular instructions, PCSel is always 0, so our circuit looks like this (pretty boring, huh?)

PCSel: Branch Instructions

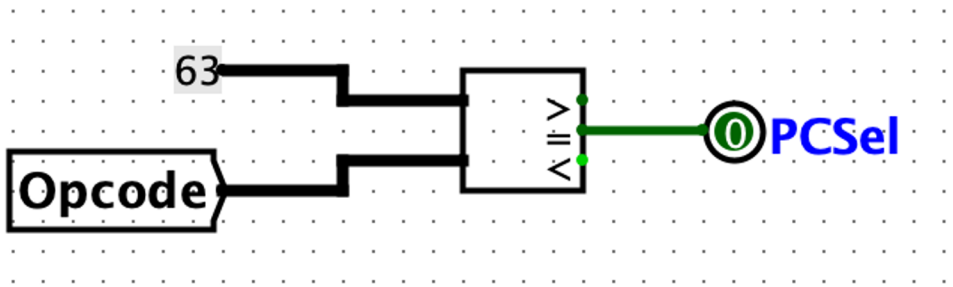
- How do we know if an instruction is a branch?
- Intuition: check the green sheet!

beq	SB	1100011	000	63/0
bne	SB	1100011	001	63/1
blt	SB	1100011	100	63/4
bge	SB	1100011	101	63/5
bltu	SB	1100011	110	63/6
bgeu	SB	1100011	111	63/7

- In order: opcode, func3 → same fields but in hex
- Look at that! they all have the same opcode! We should also check to make sure no other instructions have the same one!
  - spoiler: they don't, but you should check!

PCSel: Branch Instructions

- Let's describe our desired behaviour in words:
  - If we are a regular instruction, choose PC+4. If we are a branch instruction, choose ALUout
  - If we are a regular instruction, set PCSel = 0. If we are a branch instruction, set PCSel = 1
- We can identify a branch instruction by doing an equality check on the opcode. Here's our sub circuit:

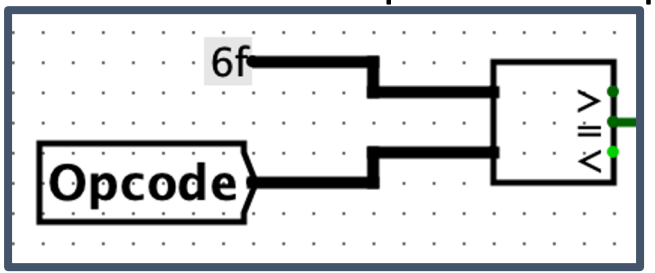


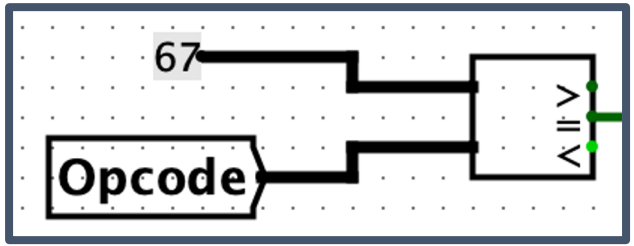
PCSel: Jumps

- Which instructions are our jump instructions?

jalr	I	1100111	000	67/0
jal	UJ	1101111		6F

- In order, opcode, func3 (none for jal) → hex
  - Oh no! These are different, so no easy generalisation here.
- Same as with branching, though, we can do an equality check on the opcode using a comparator. We'll have to do two separate comparisons here.

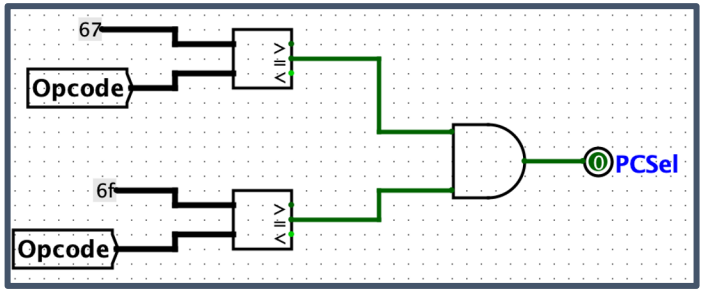




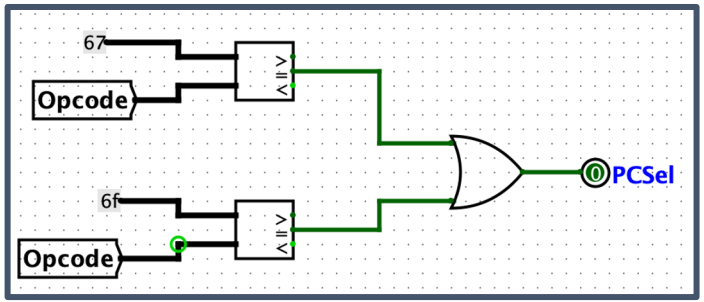
iClicker!

Which of the following circuits is the correct PCSel for jumps?

A



B

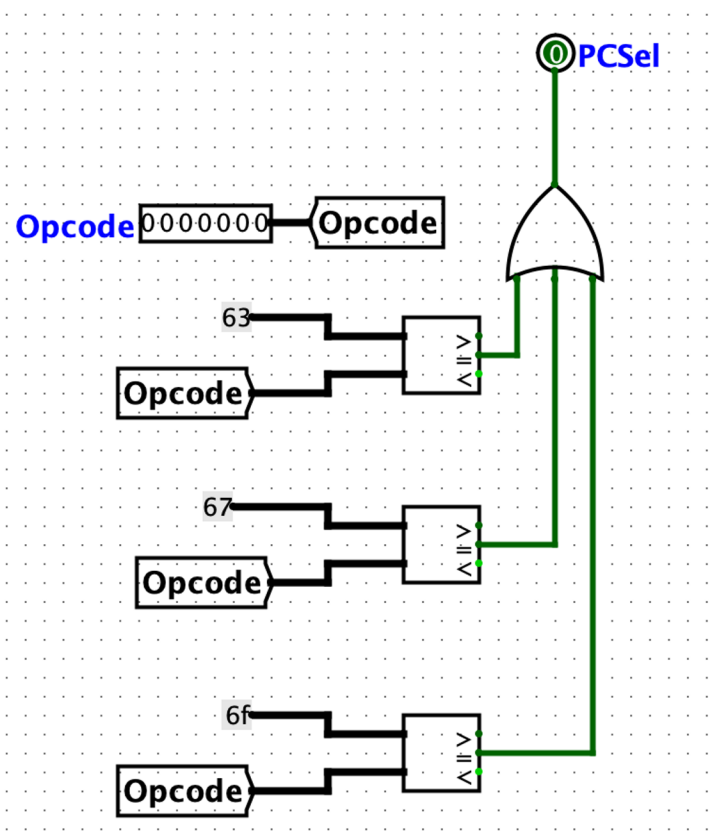


C, D, E: I don't understand how iClicker questions work

Putting them all together

- We can have a regular instruction OR a branch instruction OR a jump instruction. To combine all our signals together and retain the functionality of each individual piece, we'll OR them!
  - Describing your circuit aloud, and keying in on the words you use, might be a helpful design/debugging strategy!
- If any of the sub-circuits are true, PCSel will become (1)
  - Otherwise, it'll be 0
- Because we only have sub-circuits for the branch and jump cases, all normal instructions will have PCSel = 0, while branch, jump will have PCSel = 1 as desired :)

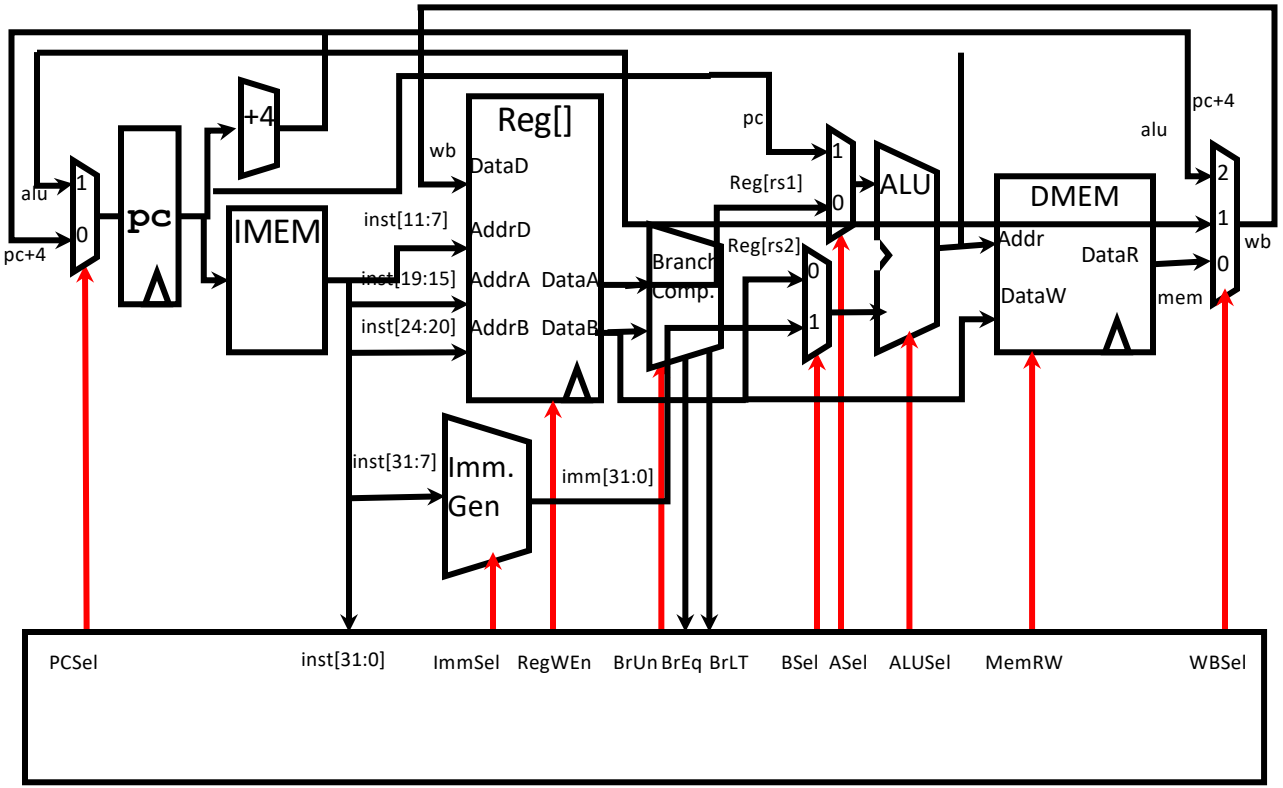
PCSel: Final Circuit



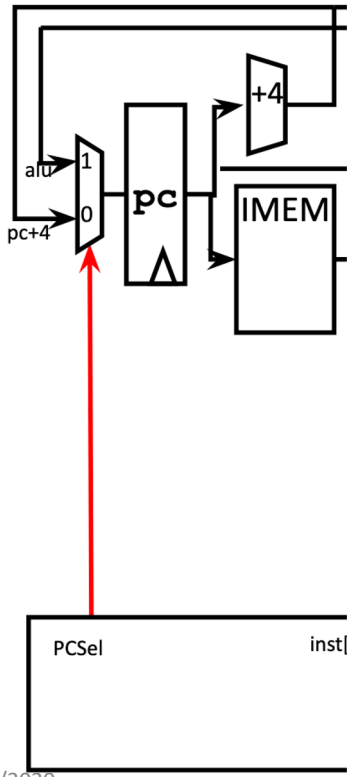
Control Signals: Big picture!

- Control signals are how we get the same hardware to behave differently and produce different instructions
- For every instruction, all control signals are set to one of their possible values (Not always 0 or 1!) or an indeterminate (*) value indicating the control signal doesn't affect the instruction's execution
- Each control signal has a sub-circuit based on ~nine bits from the instruction format:
  - Upper 5 func7 bits (lower 2 are the same for all 61C instructions)
  - All func3 bits
  - "2nd" upper opcode bit (others are the same for all 61C instructions)

Control Signals: ADD

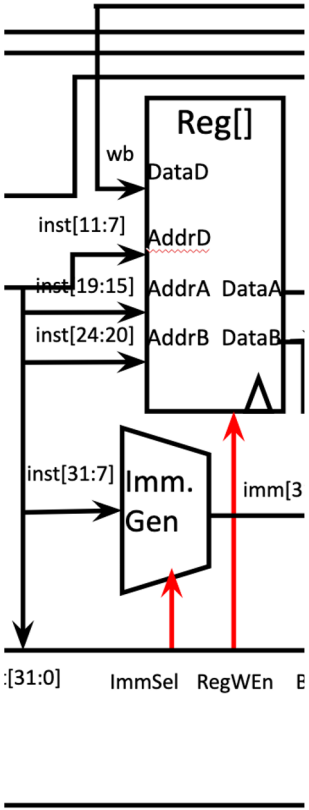


ADD: PCSel



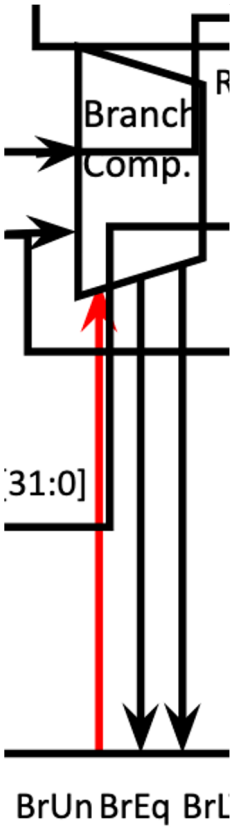
- Should we execute the next instruction (0), or jump control flow to the address given by our ALU output (1)?
- We aren't a branch or jump!
- PCSel = 0

ADD: ImmSel, RegWEn



- How do we want to assemble our immediate?
  - Wait… we don't ? have one?
  - We DON'T CARE about this signal
- ImmSel = *
- Do we want to write (1) to our destination register rd, or not (0)?
  - Add should write!
- RegWEn = 1

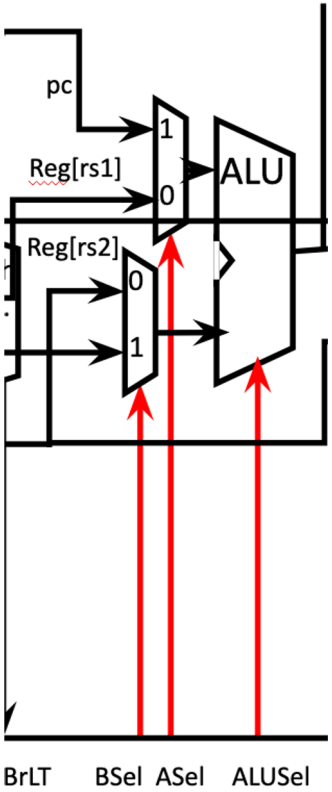
ADD: BrUn



- When we compare R[rs1] and R[rs2], should the comparison be signed (0), or unsigned (1)?
  - We aren't doing a branch !
  - This value doesn't matter

- BrUn = *

ADD: ASel, BSel, ALUSel



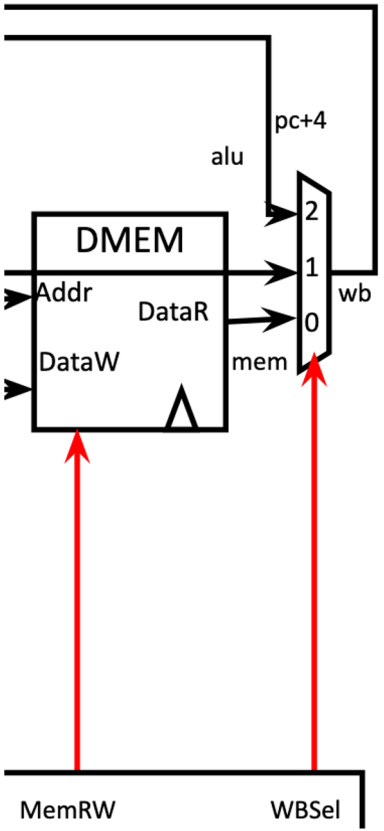
- Which operands do we want to operate on?
  - ADD requires rs1 and rs2
- ASel = 0 (rs1)
- BSel = 0 (rs2)

- What operation do we want to perform?
  - ADD == uh… add ?
- ALUSel = “Add”
  - but wait… that’s not binary, how does that work?

ALUSel

- For diagramming purposes, we set ALUSel on examples and exam questions to an english value (add, sub, or, etc.)
- In your CPU, it'll have a binary value (and so will all other signals!)
- The mapping between english words and binary values depends on how you build your ALU!
  - These mappings are arbitrary! As long as you're consistent (all add-based instructions have the same ALUSel) things will work just fine

ADD: MemRW, WBSel



- Are we reading (0) or writing (1) memory?
  - Wait, we're not doing anything with memory. Can this be a "don't care" value?
    - NO NO NO NO NO ! :(
    - We never want to "accidentally" write memory! This has to be a "passive read".
- MemRW = 0
- What value do we want to write back to rd?
  - ALU Out!
- WBSel = 1

ADD: Control Signals

Here are the signals and values we've compiled for our ADD instruction:

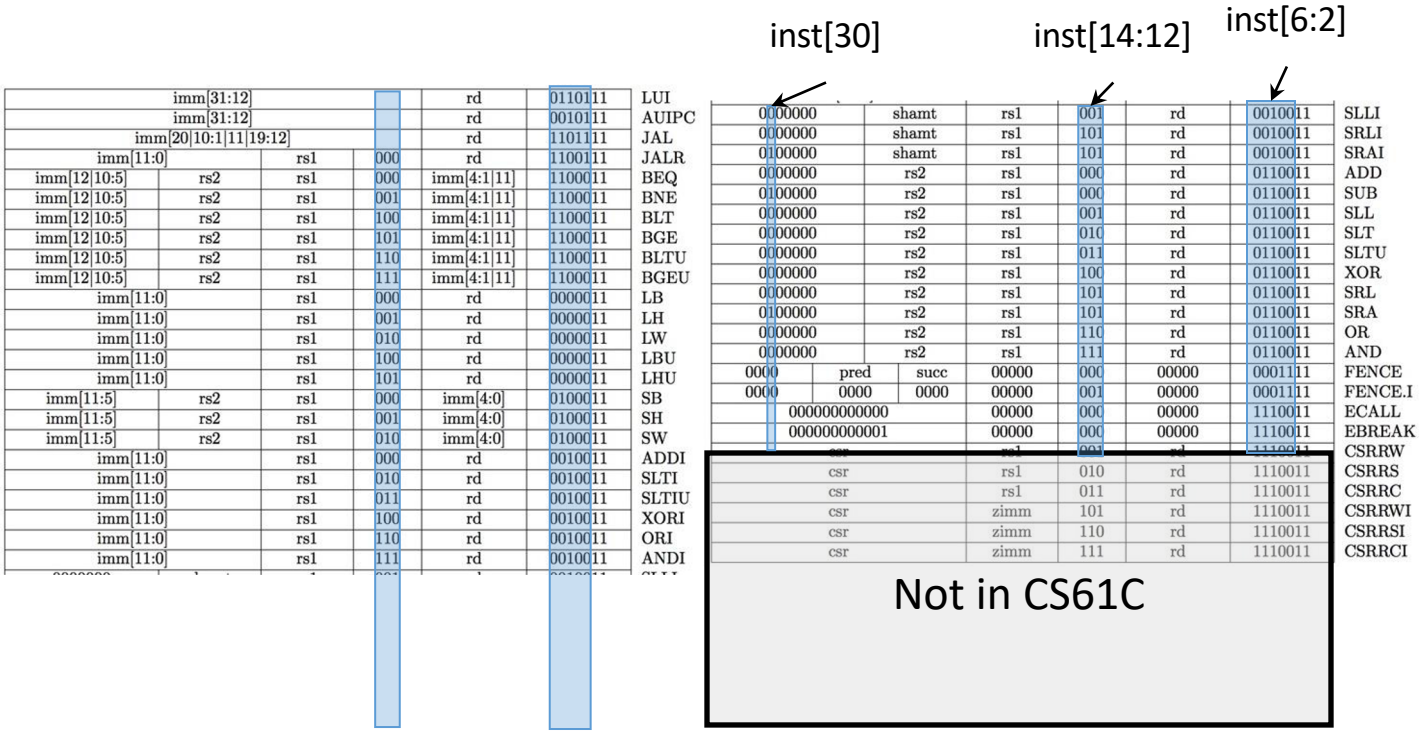
Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
`add`	*	*	+4	*	*	Reg	Reg	Add	Read	1 (Y)	ALU

(green = left 3 cols = control INPUTS)

(orange = right 9 cols = control OUTPUTS)

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1 (Y)	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 (N)	*
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

RV32I, a nine-bit ISA!

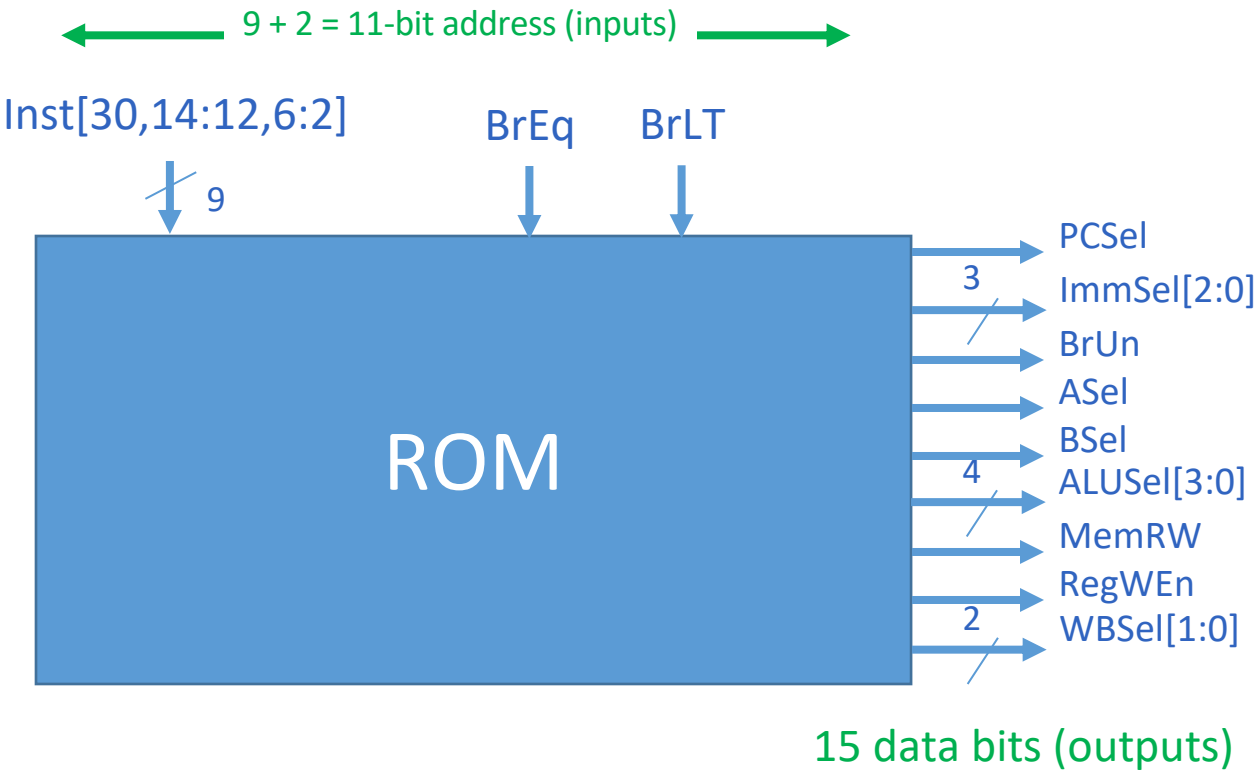


Instruction type encoded using only 9 bits
inst[30],inst[14:12], inst[6:2]

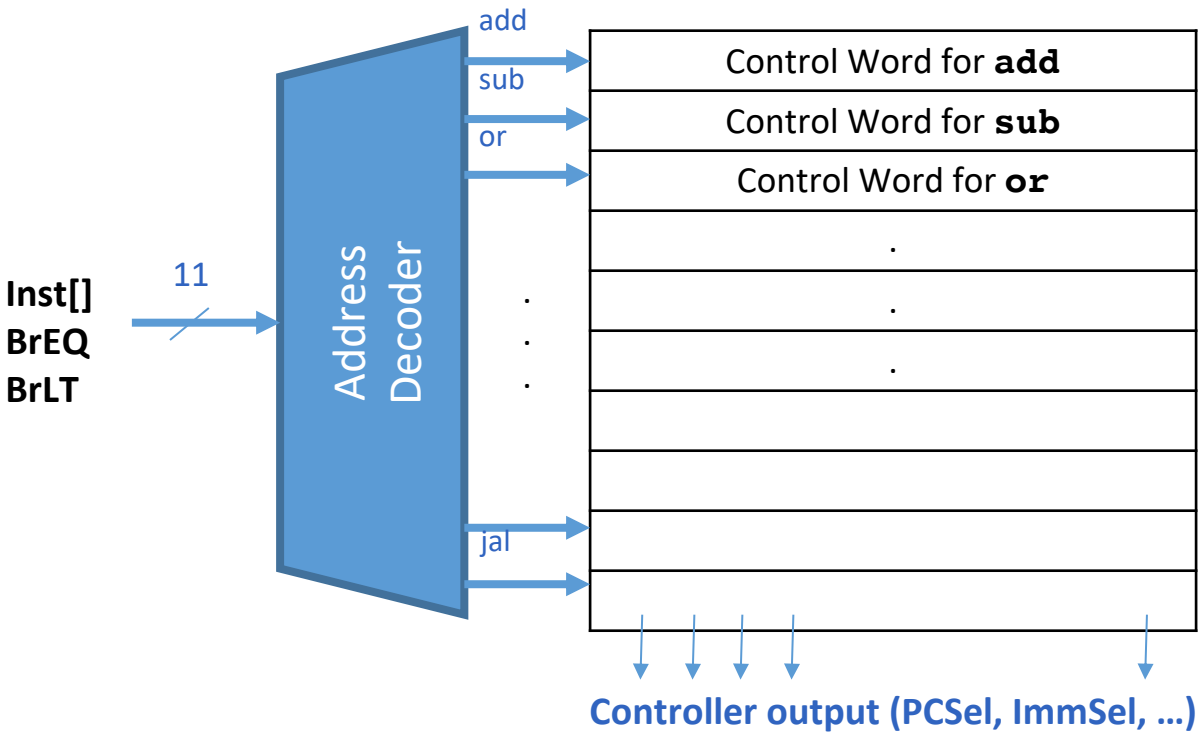
Control Construction Options

- ROM
  - "Read-Only Memory"
  - Regular structure (like previous slide's table)
  - 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
  - Not easily changeable/re-programmable because requires modifying hardware
    - But! Likely less expensive, more complex

ROM-based Control



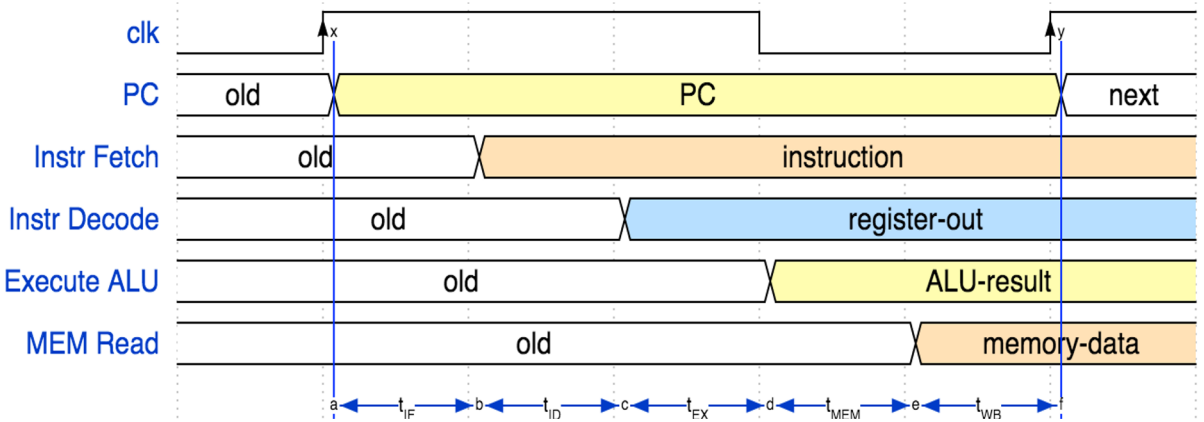
ROM Controller Implementation



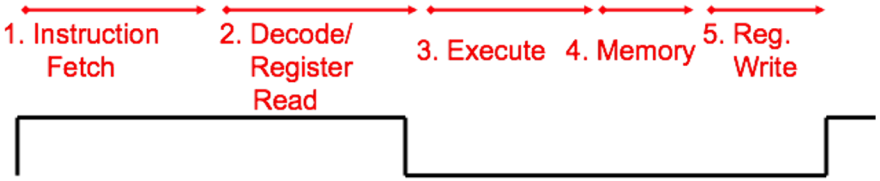
Agenda

- Quick Datapath Review
- Control Implementation
- Performance Analysis
- Pipelined Execution
- Pipelined Datapath

Instruction Timing



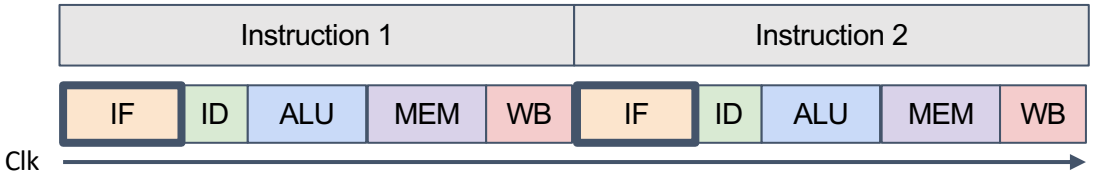
IF	ID	EX	MEM	WB	Total
IMEM	Reg Read	ALU	DMEM	Reg W	
200 ps	100 ps	200 ps	200 ps	100 ps	800 ps



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! ex. “IF” active every 600ps



Performance Measures

- In our example, 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)?
    - 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: **Latency** 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

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

Instructions per Program

Determined by

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

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

(Average) Clock cycles per Instruction, or CPI

Determined by

- ISA and processor implementation (or *microarchitecture*)
  - E.g. for "our" single-cycle RISC-V design, CPI = 1
- Complex instructions (e.g. **`strcpy`**), CPI >> 1
  - True for most CISC languages
- Superscalar processors, CPI < 1 (next lecture)

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

Time per Cycle (1/Frequency)

Determined by

- Processor microarchitecture (processor critical path)
- Technology (e.g. transistor size)
- Power budget (lower voltages reduce transistor speed)

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

Speed Trade-off 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! Why? Each instruction is less complex! (~2.5 B instructions = 1 A instruction)

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 Trade-off 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

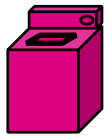
Agenda

- Quick Datapath Review
- Control Implementation
- Performance Analysis
- Pipelined Execution
- Pipelined Datapath

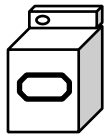
Pipeline Analogy: Doing Laundry

- Dan, Stephan, Sean and Jenny each have one load of clothes to wash, dry, fold, and put away

  - Washer takes 30 minutes



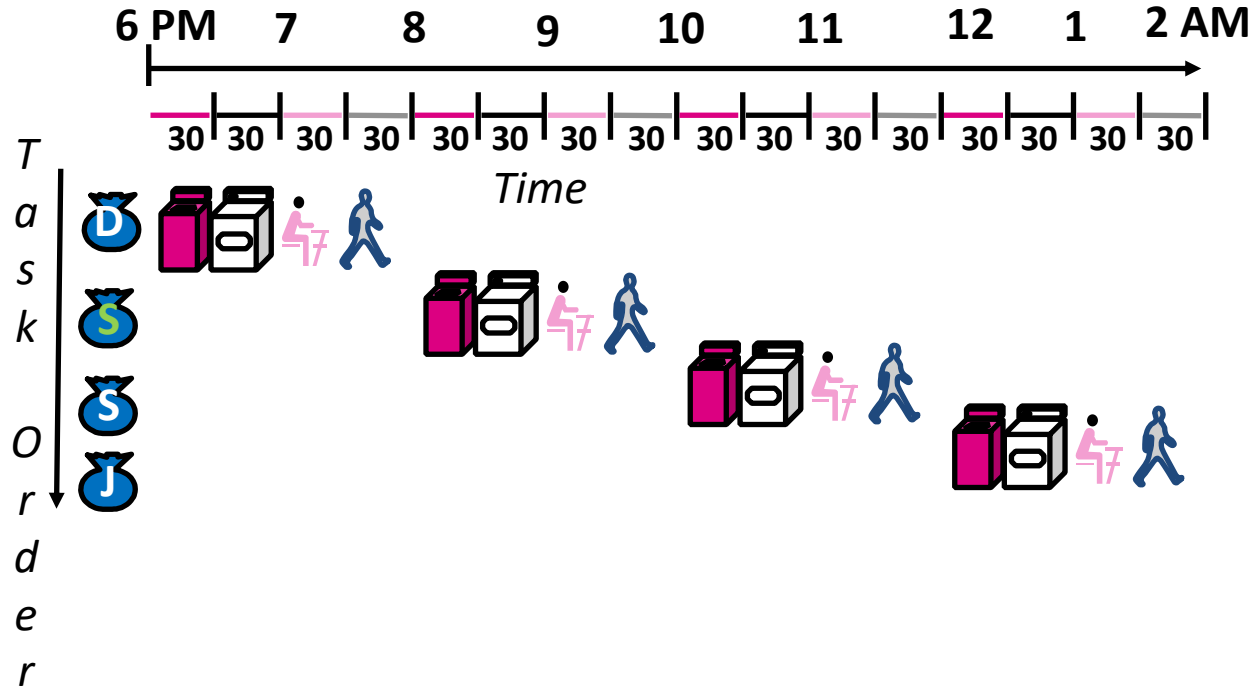
  - Dryer takes 30 minutes



  - "Folder" takes 30 minutes

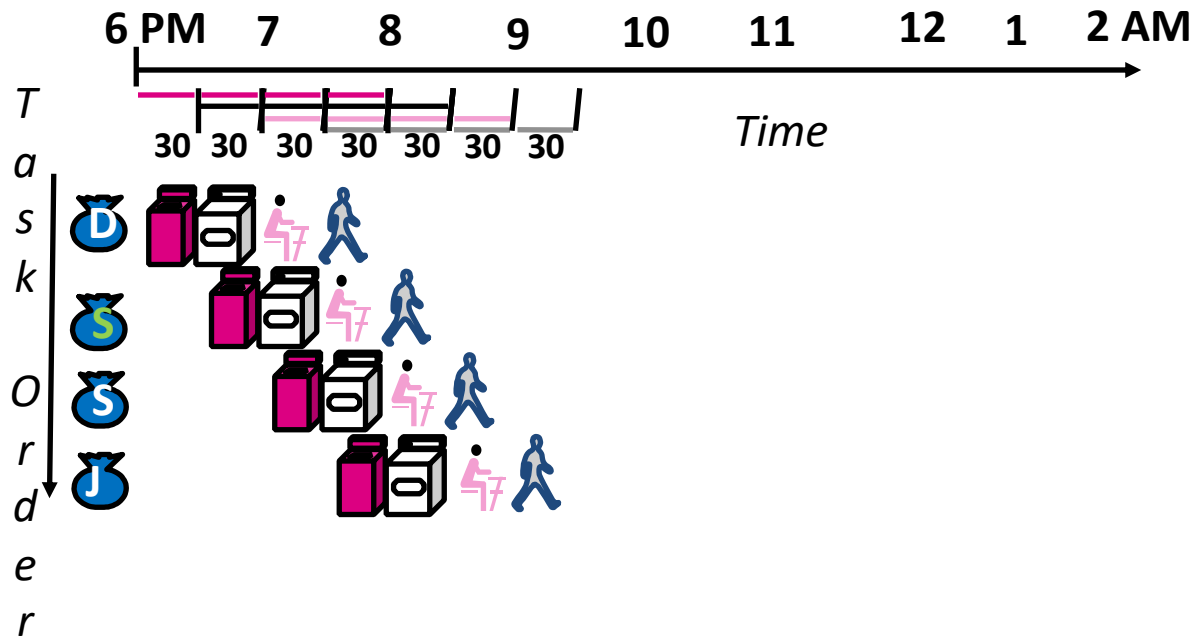
  - "Stasher" takes 30 minutes to put clothes into drawers

Sequential Laundry



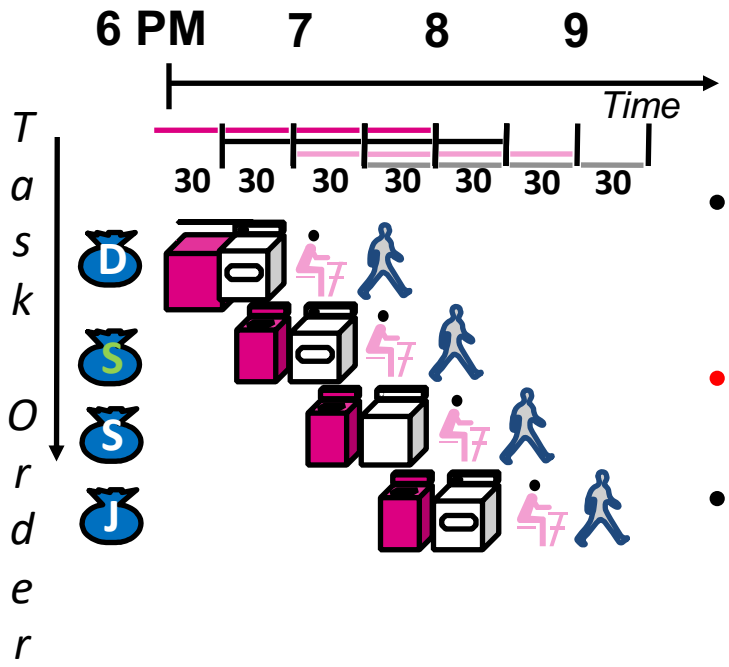
- Sequential laundry takes 8 hours for 4 loads
- 1 load finishes every 2 hours, and Jenny is up til 2AM...

Pipelined Laundry



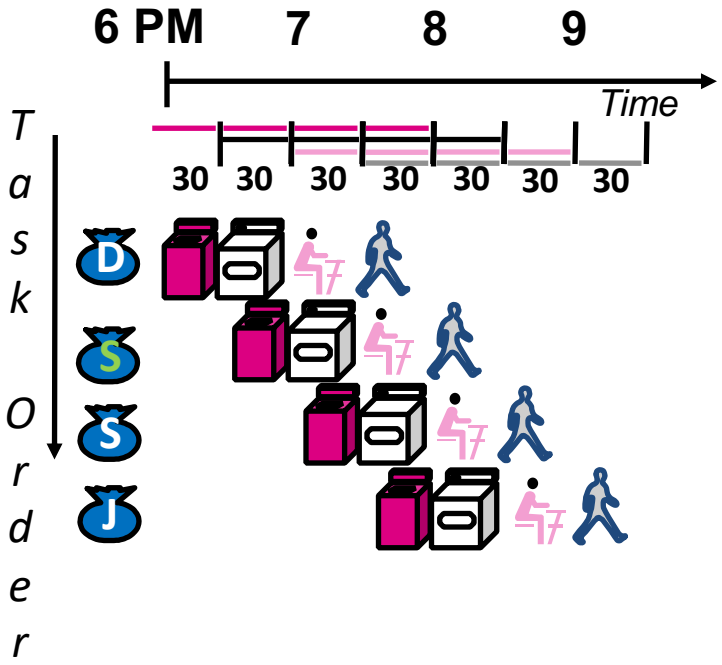
- Pipelined laundry takes 3.5 hours for 4 loads!
- 1 load finishes every half hour (after the first load, which takes 2 hours)

Pipelining Lessons (1/2)



- Pipelining doesn't decrease *latency* of single task; it increases *throughput* of entire workload
- *Multiple* tasks operating simultaneously using different resources
- Potential speedup ~ number of pipeline stages
- Speedup reduced by time to *fill* and *drain* the pipeline:
  8 hours/3.5 hours which gives 2.3X speedup v. potential 4X in this example

Pipelining Lessons (2/2)



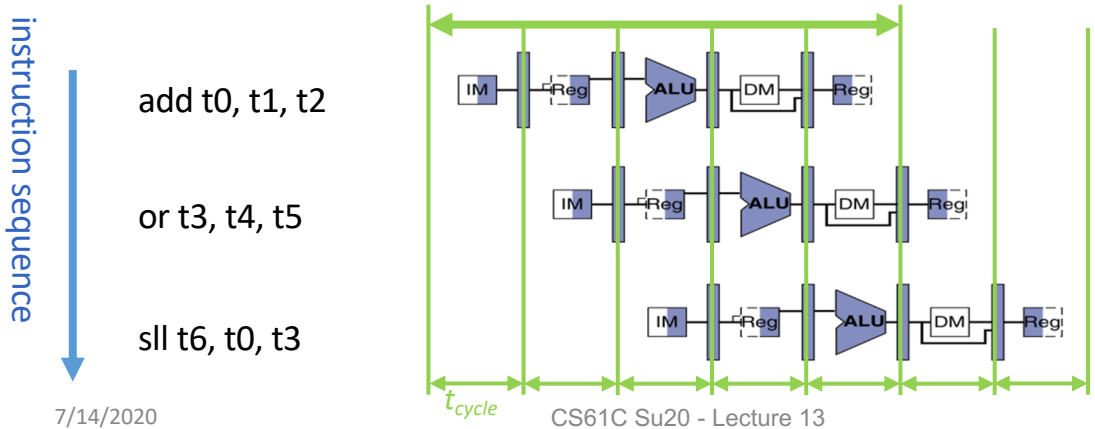
- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
  - Pipeline rate limited by *slowest* pipeline stage
  - Unbalanced lengths of pipeline stages reduces speedup

Agenda

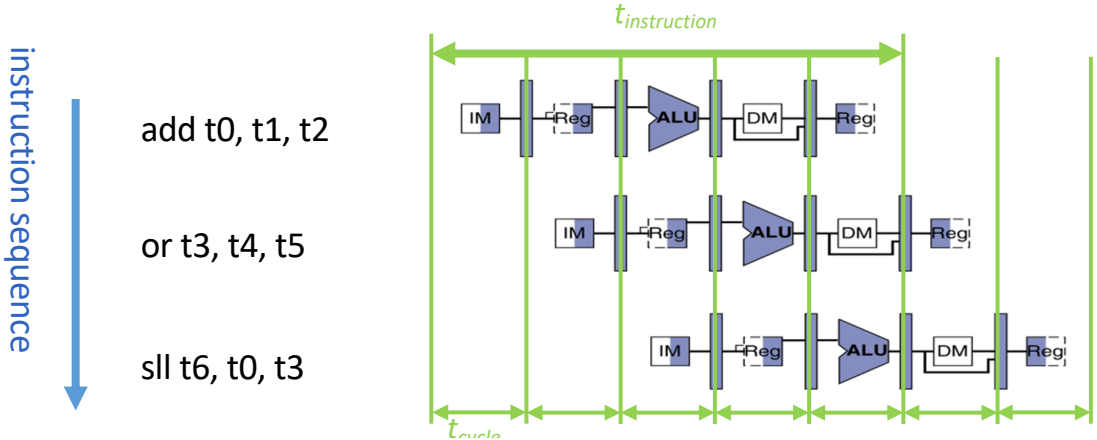
- Quick Datapath Review
- Control Implementation
- Administrivia
- Performance Analysis
- Pipelined Execution
- Pipelined Datapath

Pipelining with RISC-V

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



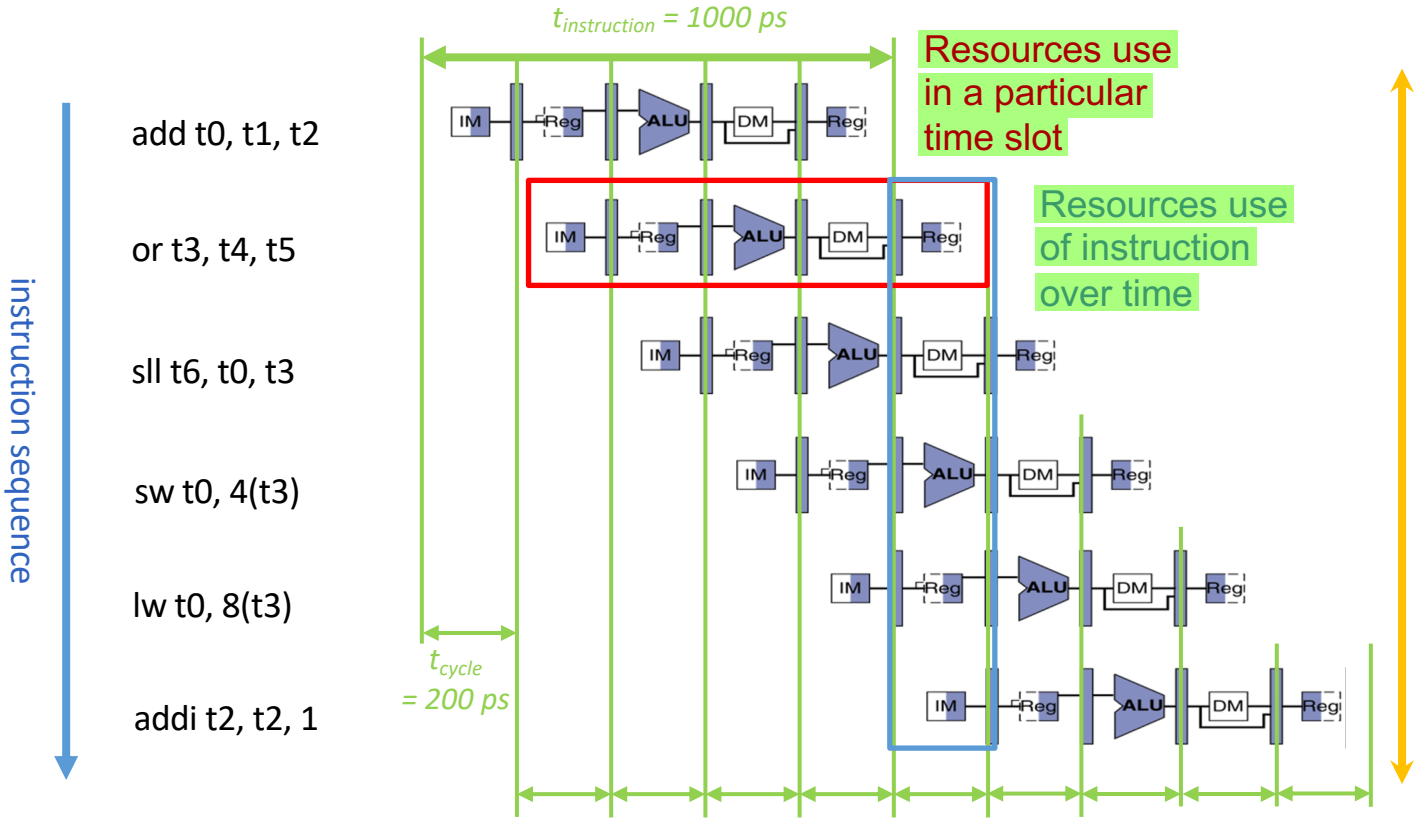
Pipelining with RISC-V



	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
CPI (Cycles Per Instruction)	~ 1 (ideal)	> 1 (actual)
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

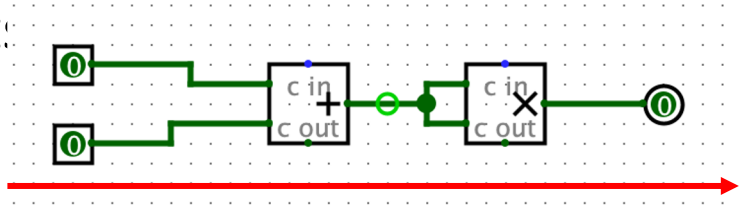
Sequential vs Simultaneous

What happens sequentially, what happens simultaneously?

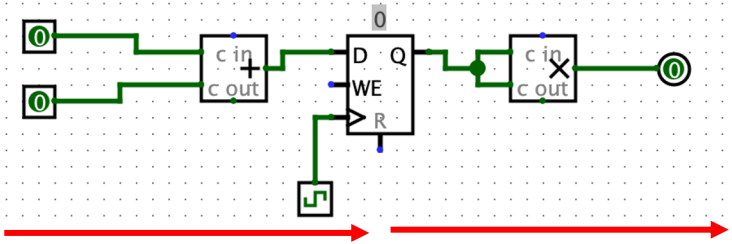


Quick review: Circuit pipelining!

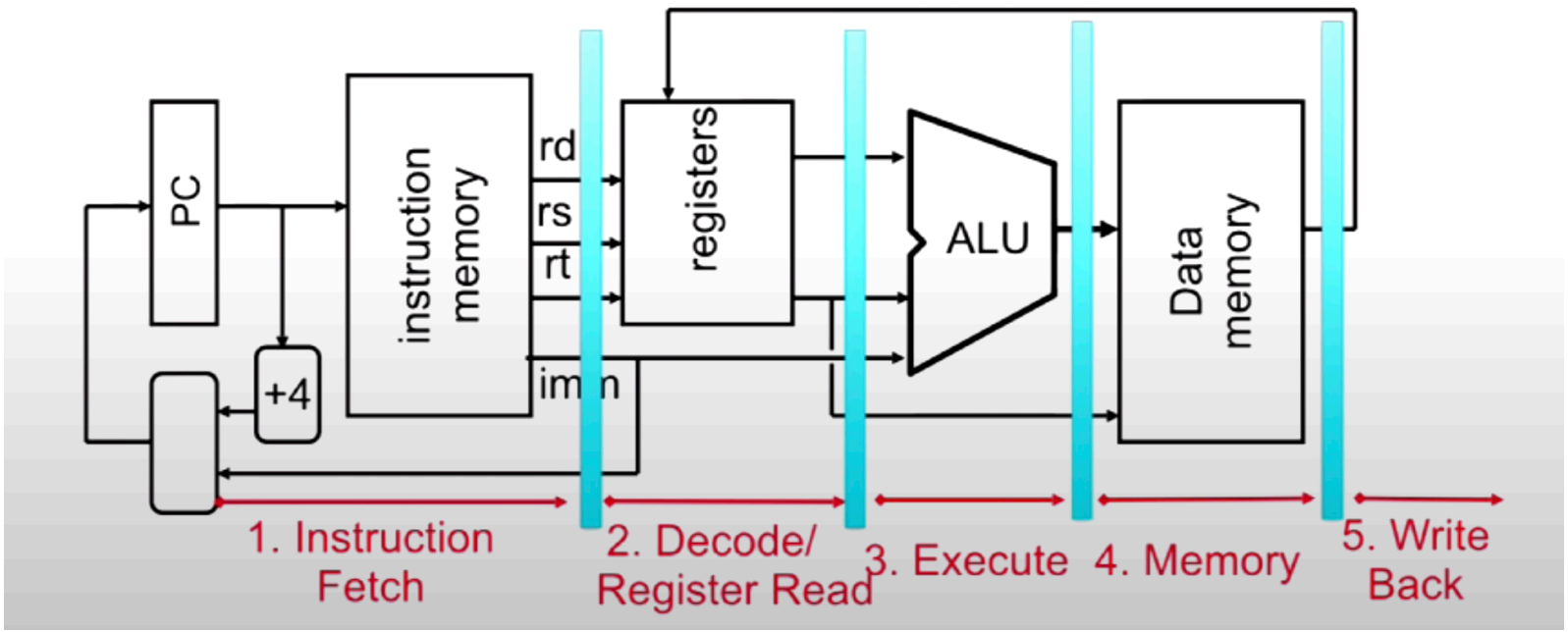
- When we calculate cycle time, or critical path, we do so between state elements, inputs, and outputs



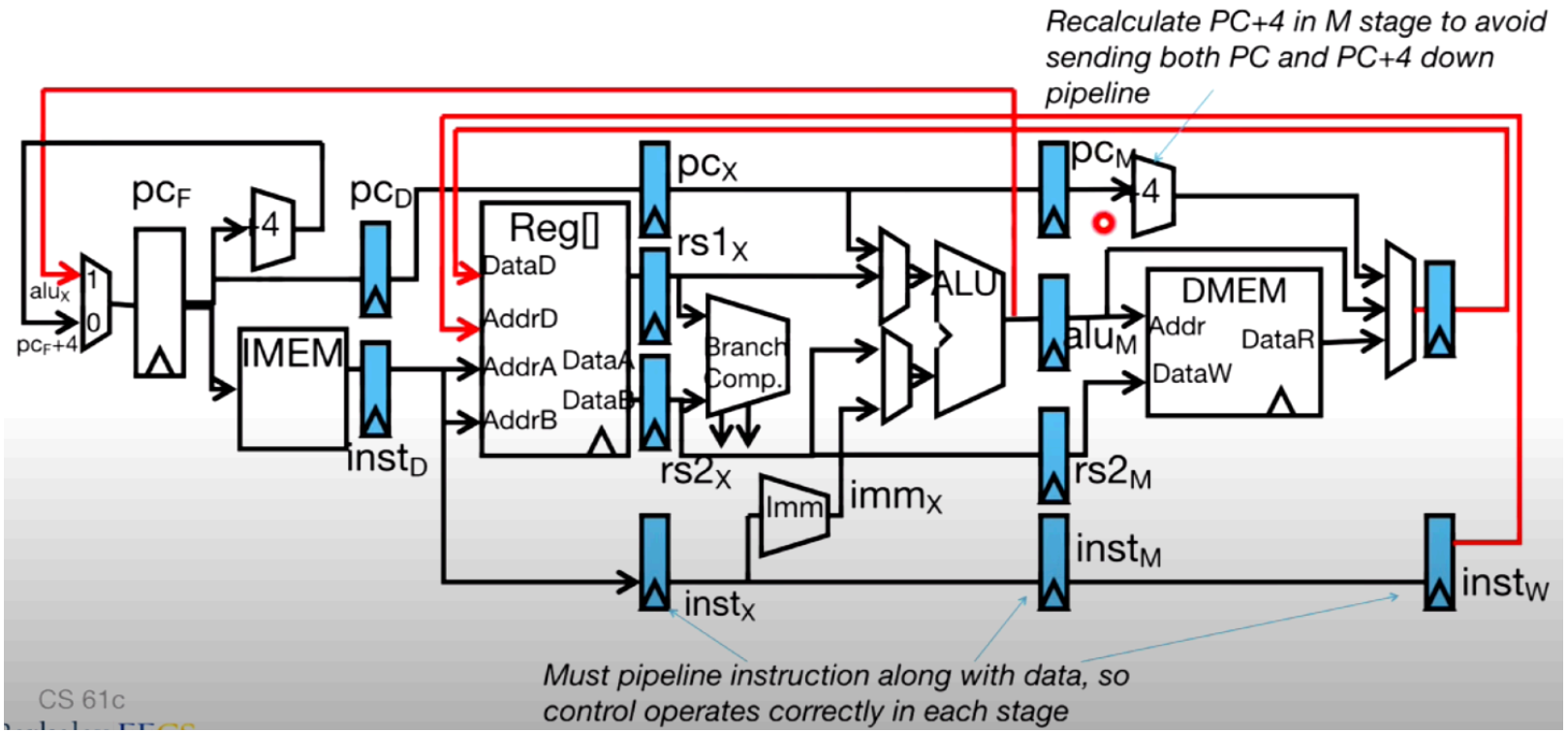
- Adding registers between circuit components *decreases* our critical path and *increases* our frequency



Pipelining with RISC-V

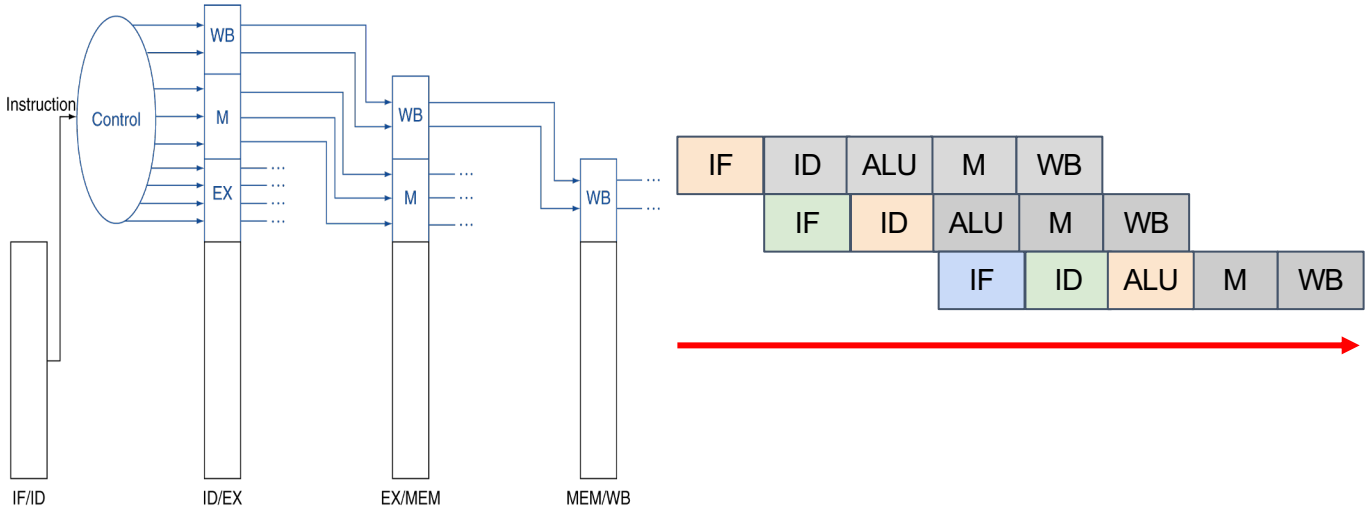


Pipelining with RISC-V



Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages
- At any given point, there are up to 5 different instructions in the datapath! We must keep track of 5 different sets of control bits!



Summary

- Implementing controller for your datapath
  - Ask yourself the questions on the beginning slides!
  - Work in stages, put everything together at the end!
- Pipelining improves performance by exploiting Instruction Level Parallelism
  - 5-stage pipeline for RV32I: IF, ID, EX, MEM, WB
  - Executes multiple instructions in parallel
  - Each instruction has the same latency, but there's better throughput
  - Think: what problems does pipelining introduce? (*more on this next lecture*)