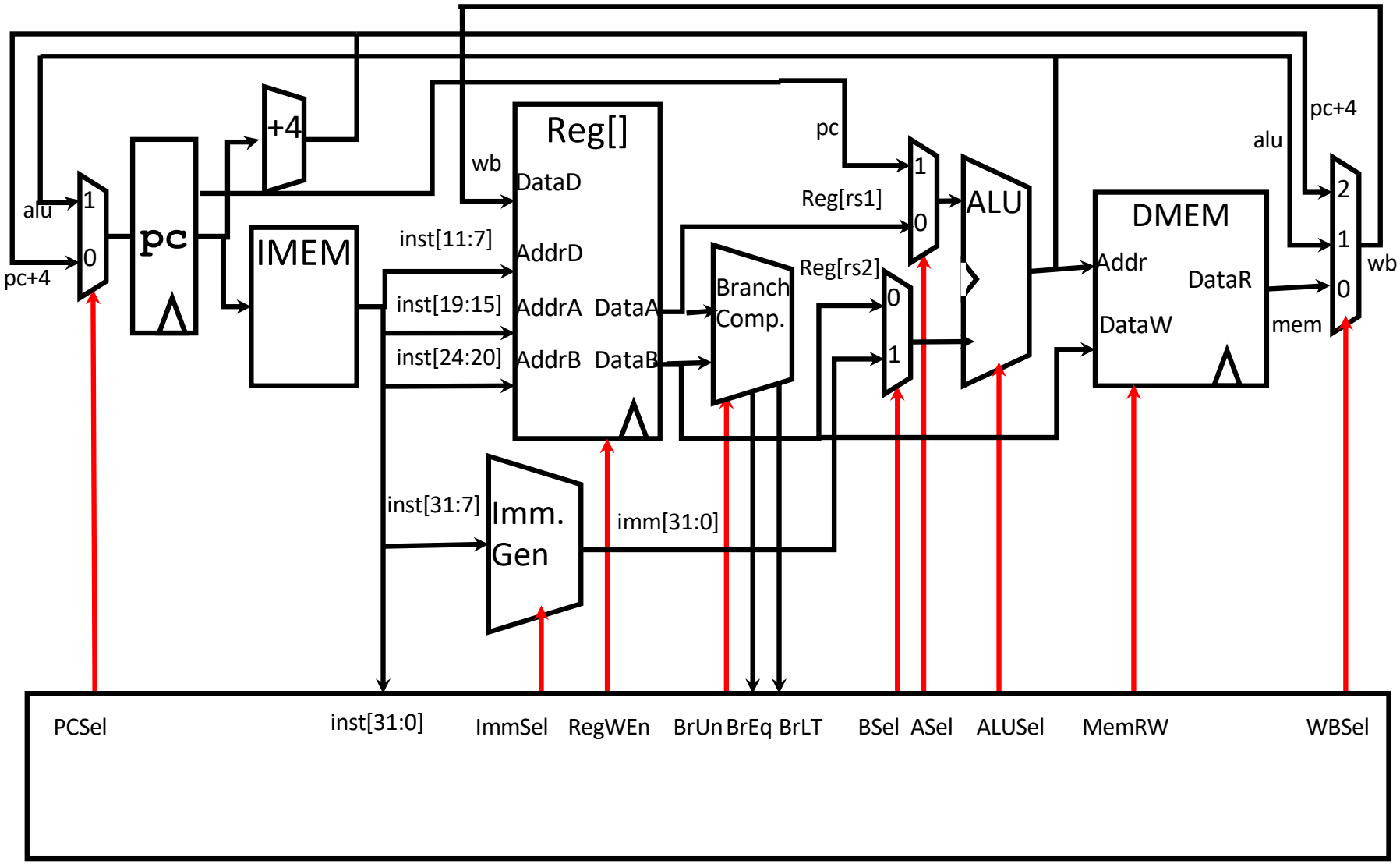
Computer Architecture, Fall 2019

RISC-V CPU Control

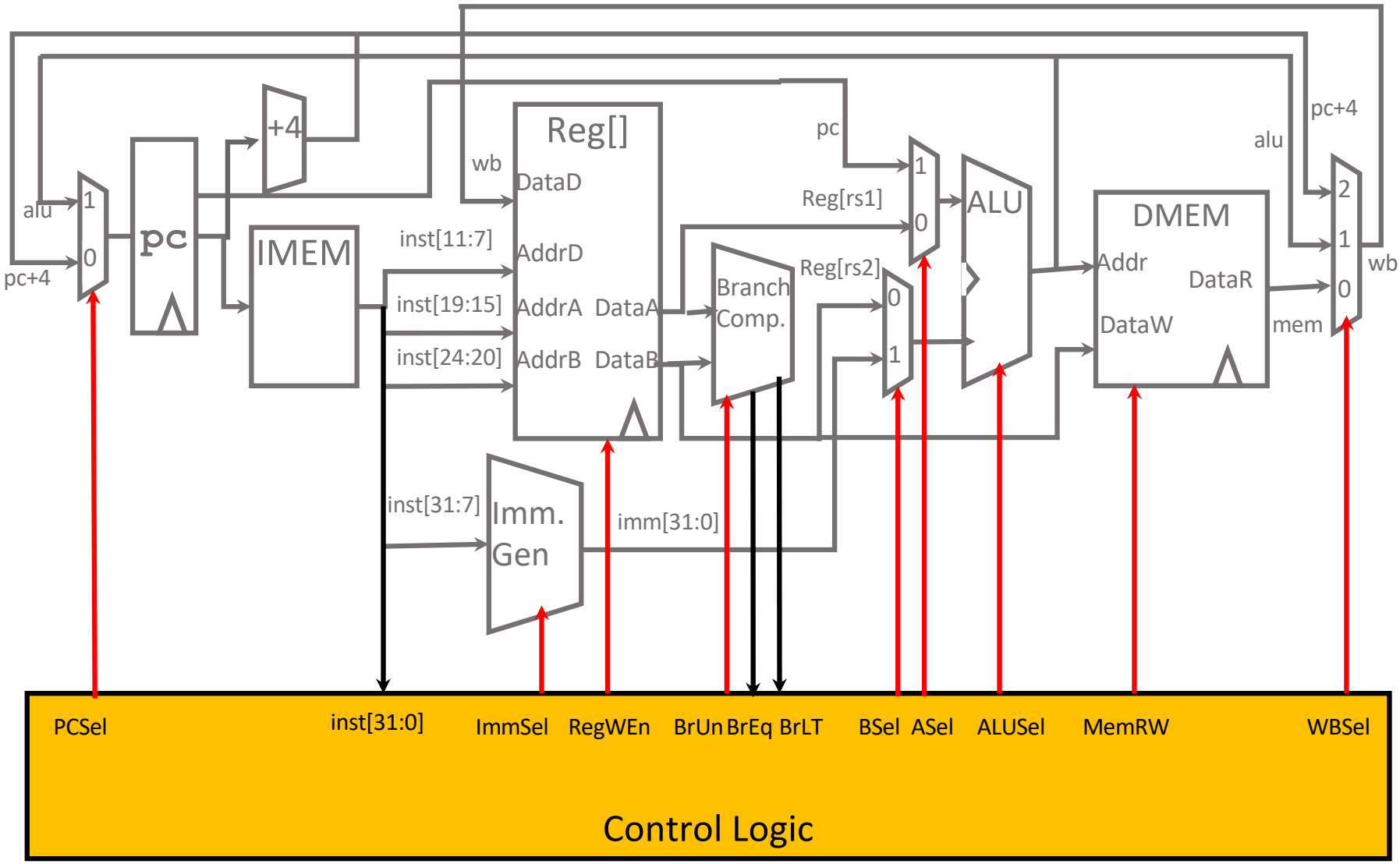
Agenda

- Datapath Review
- Control Implementation
- Performance Analysis

Single-Cycle RISC-V RV32I Datapath



Single-Cycle RISC-V RV32I Datapath



Agenda

- Quick Datapath Review
- Control Implementation
- Performance Analysis

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

- 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!

I recognize this is hard :(

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.

PCSel: Branch Instructions

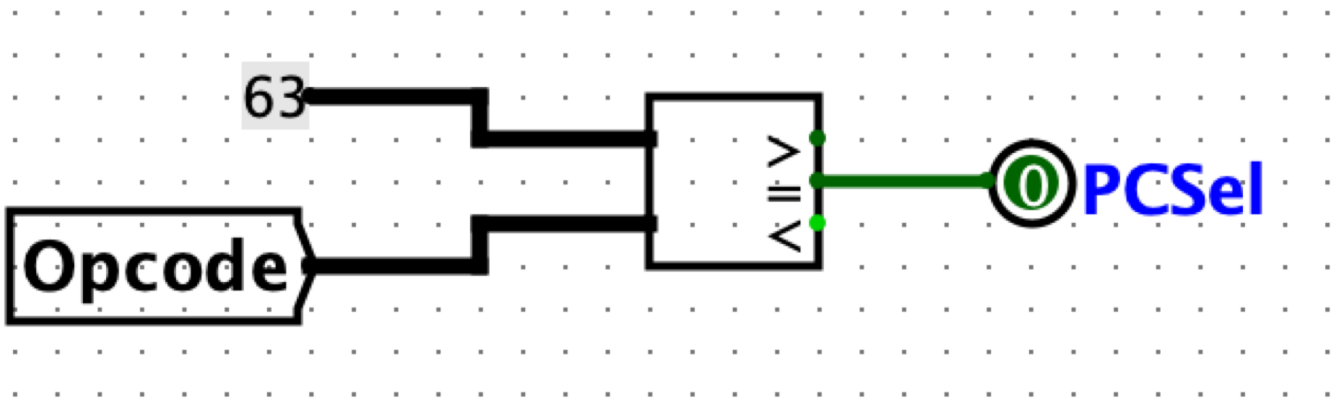
- How do we know if an instruction is a branch?
- Check the green sheet!

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:

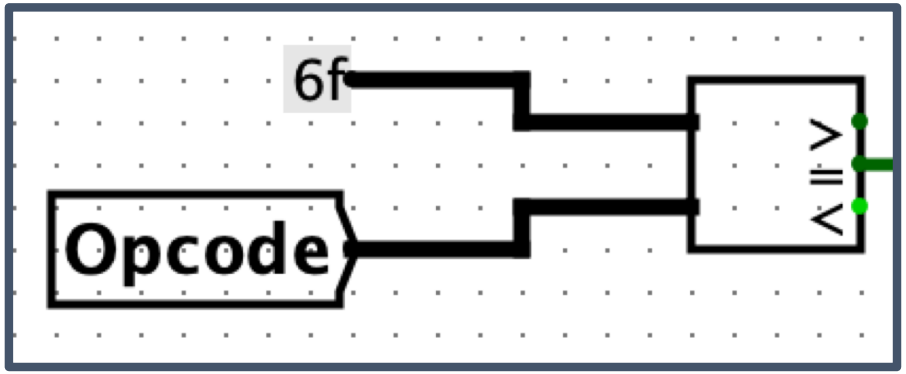


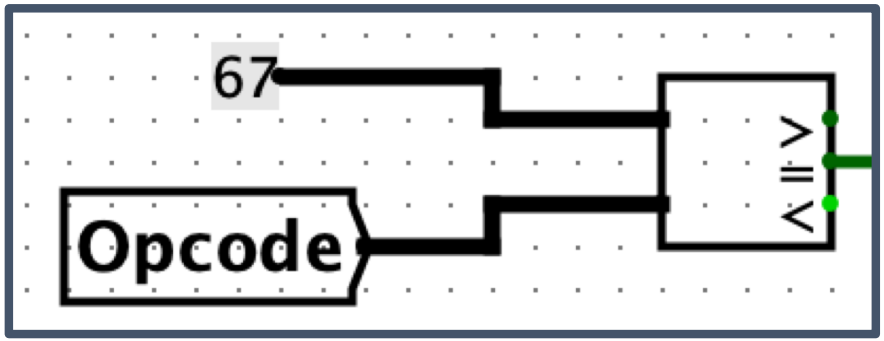
PCSel: Jumps

- 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 generalization 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.

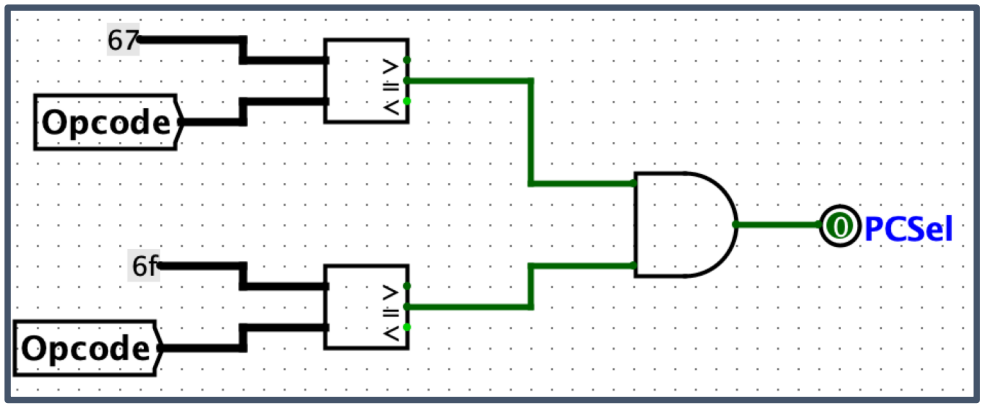




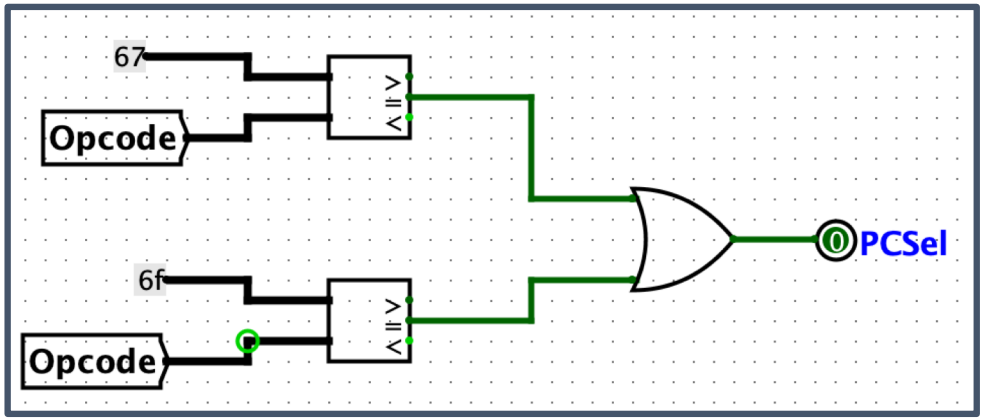
Question

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

A



B

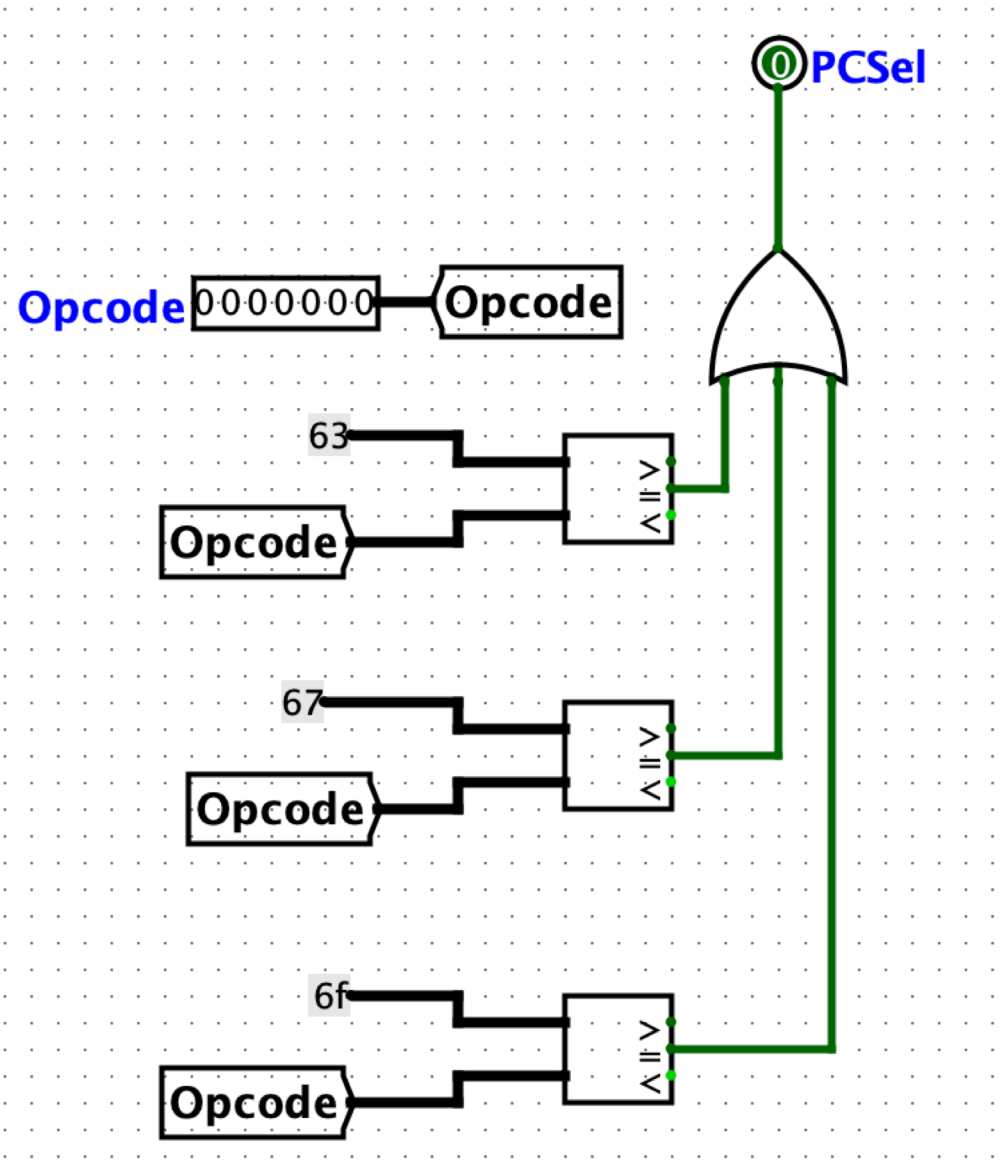


C, D, E: I don't understand how iClicker questions work

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 :)

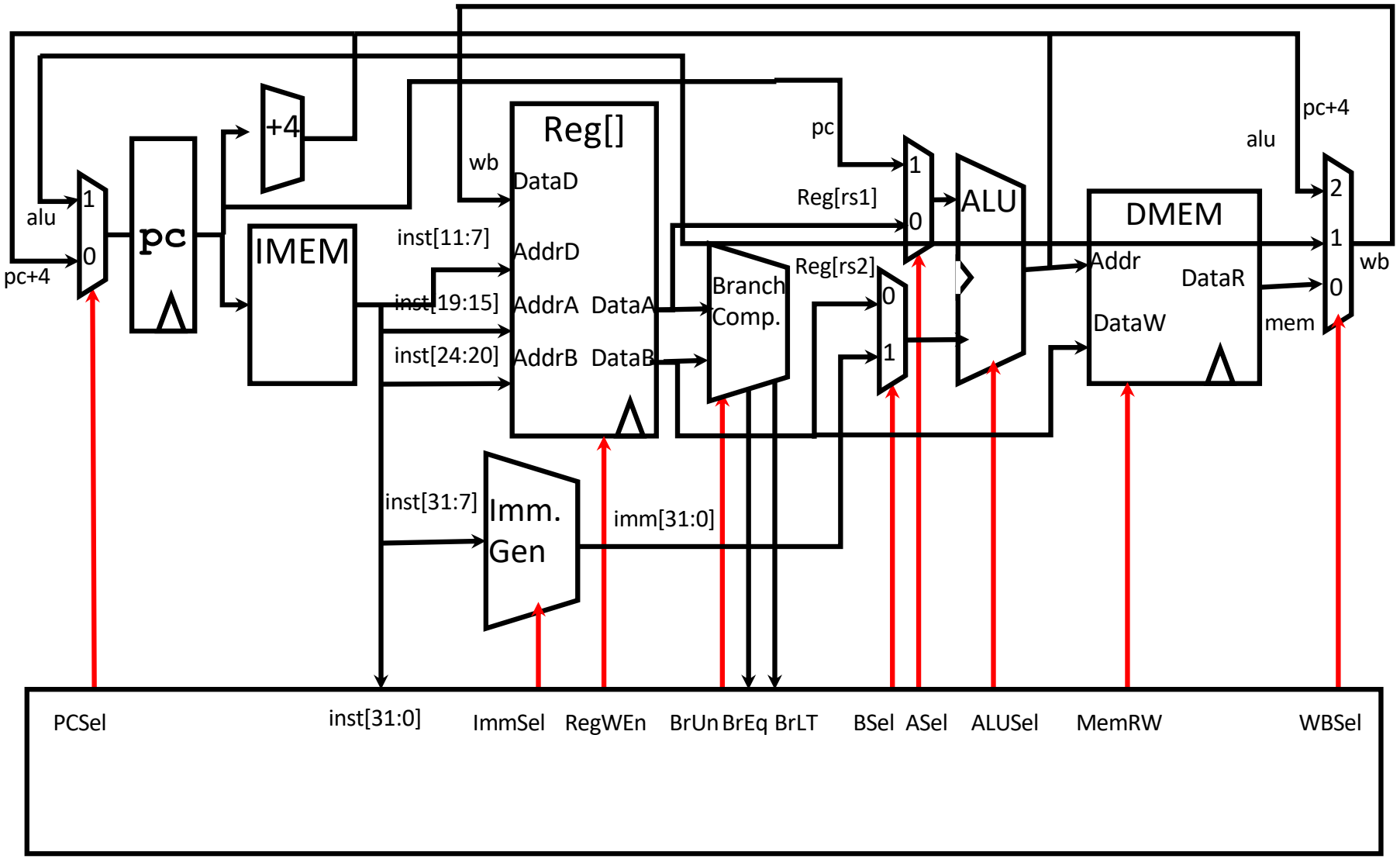
PCSel: Final Circuit



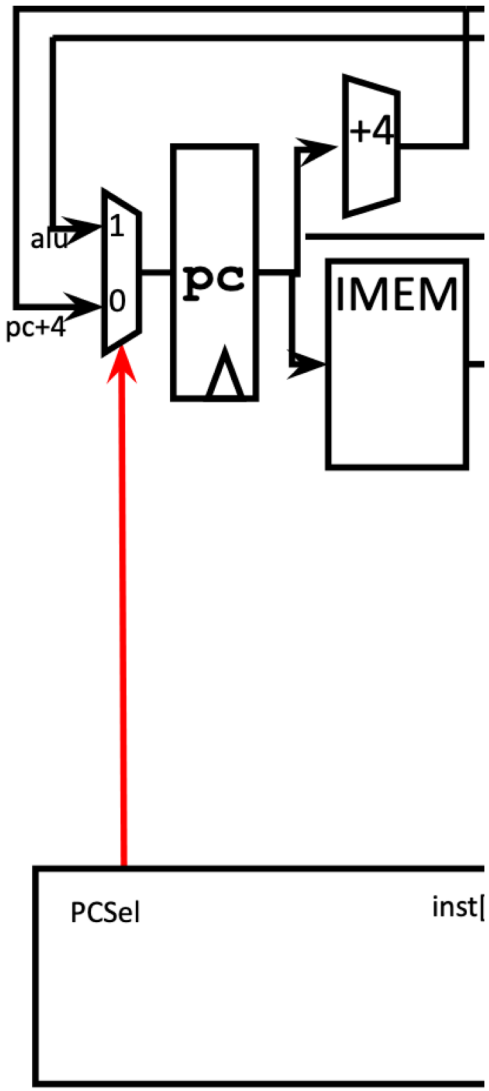
Control Signals: Big picture!

- 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 instructions)
  - All func3 bits
  - "2nd" upper opcode bit (others are the same for all instructions)

Control Signals: ADD

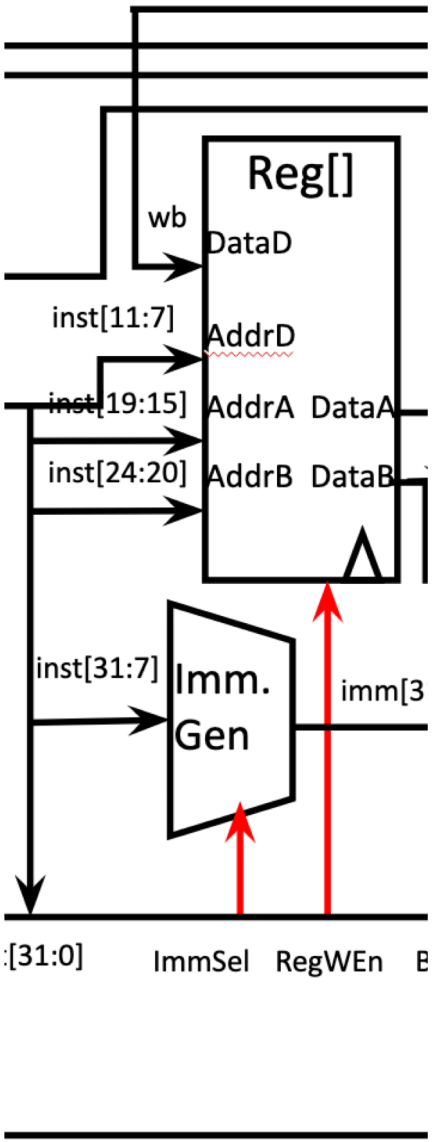


ADD: PCSel



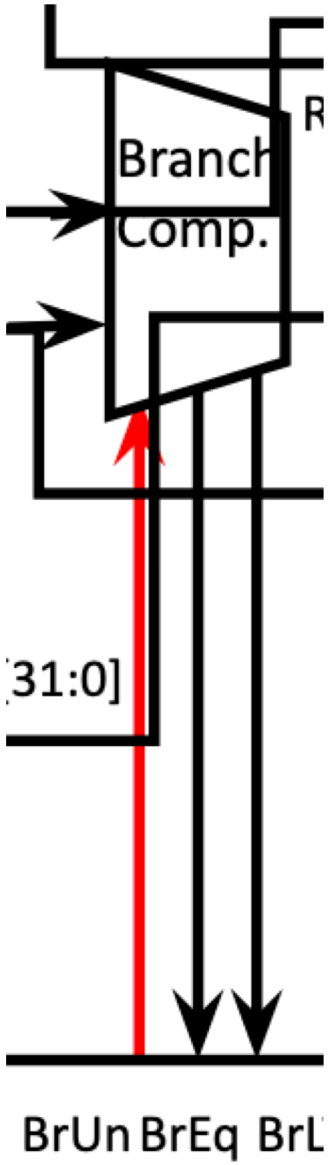
- Should we execute the next instruction (0), or jump control flow to the address given by our ALU output (1)?
- We aren't a branch or jump!
- PCSel = 0

ADD: ImmSel, RegWEn



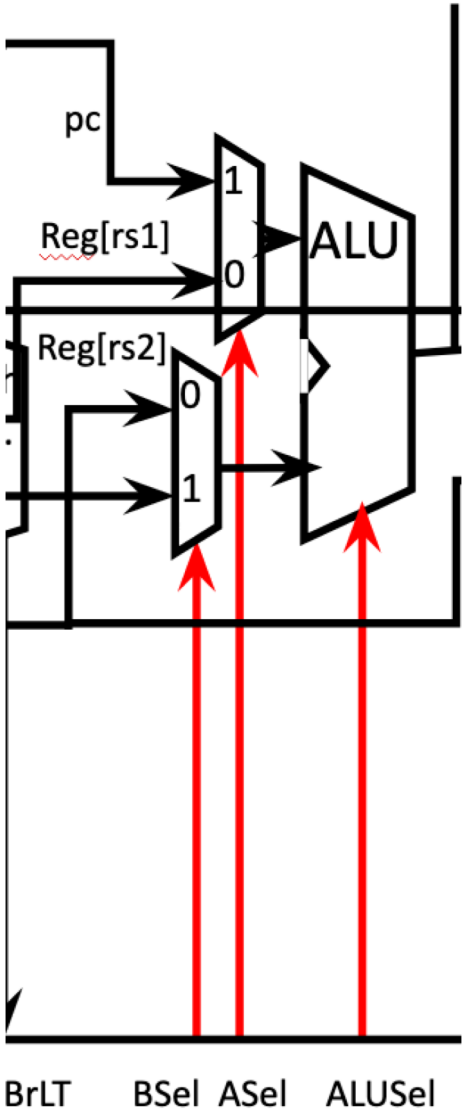
- How do we want to assemble our immediate?
  - We DON'T CARE about this signal
- ImmSel = *
- Do we want to write (1) to our destination register rd, or not (0)?
  - ADD should write!
- RegWEn = 1

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



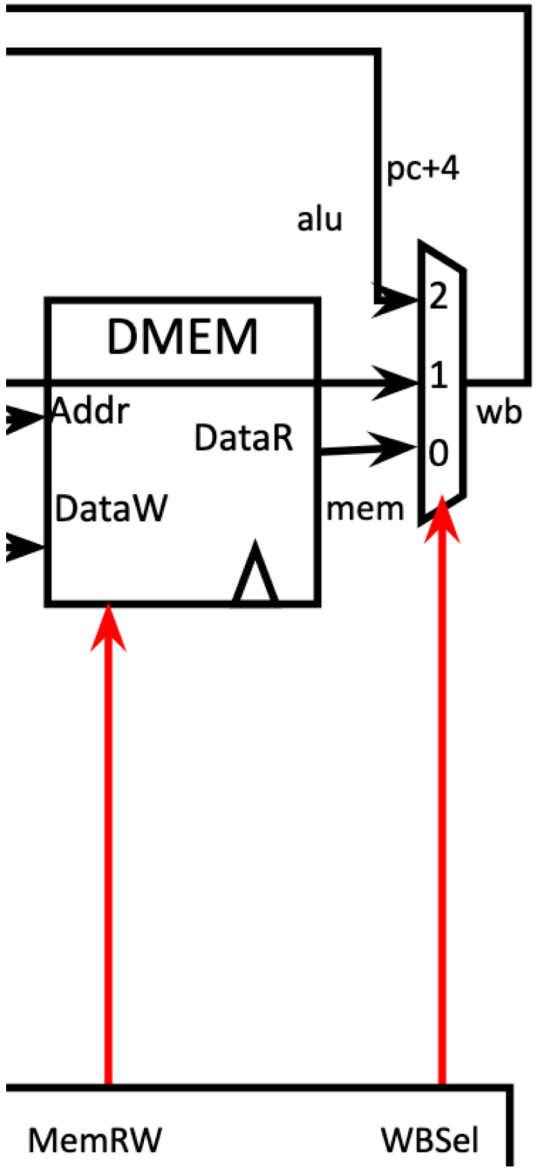
- Which operands do we want to operate on?
  - ADD requires rs1 and rs2
- ASel = 0 (rs1)
- BSel = 0 (rs2)

- What operation do we want to perform?
  - addition
- ALUSel = “Add”
  - 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?
  - 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 = 2

ADD: Control Signals

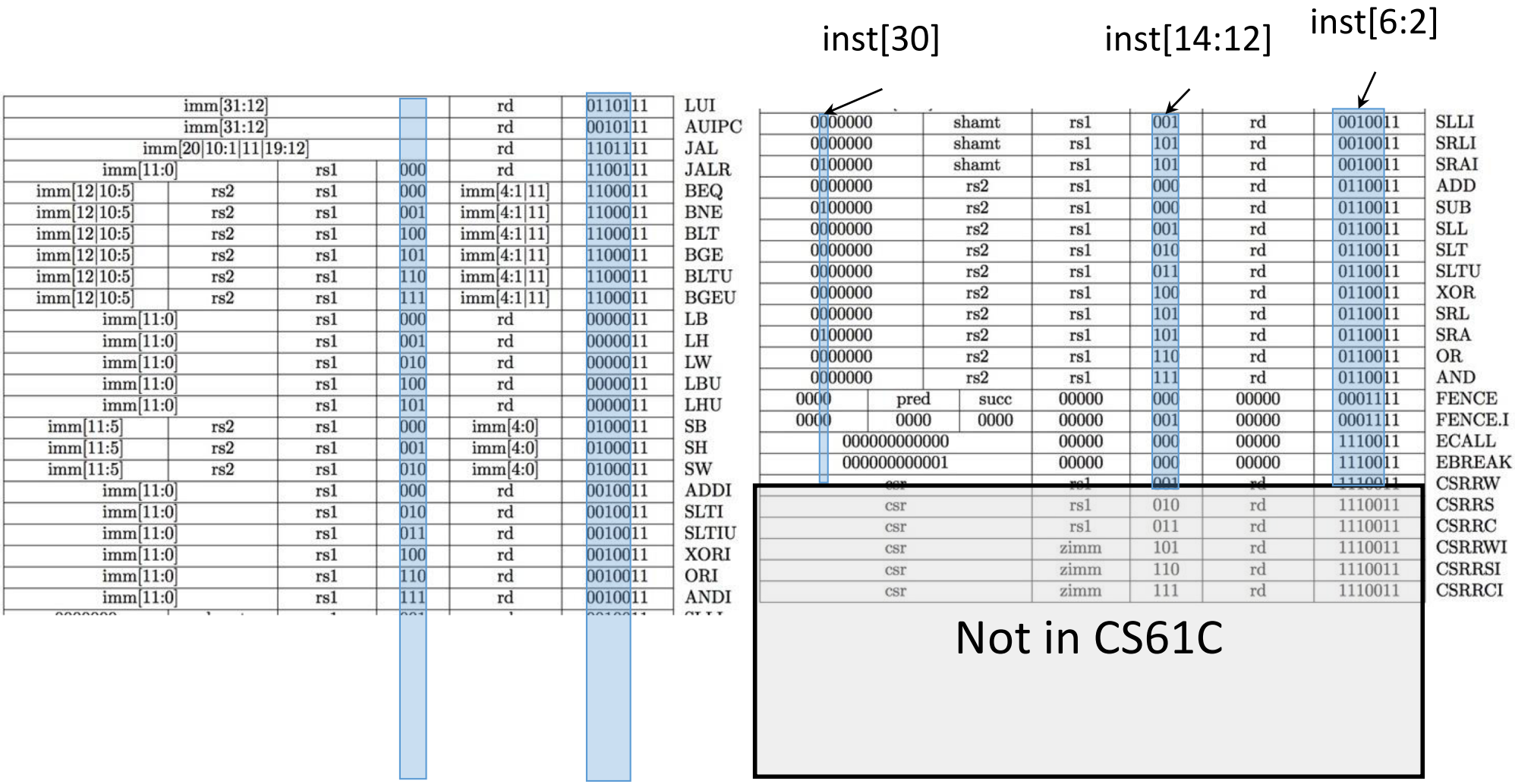
Here are the signals and values we've compiled for our ADD instruction:

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 columns = control INPUTS)

(orange = right 9 columns = control OUTPUTS)

RV32I, a nine-bit ISA!



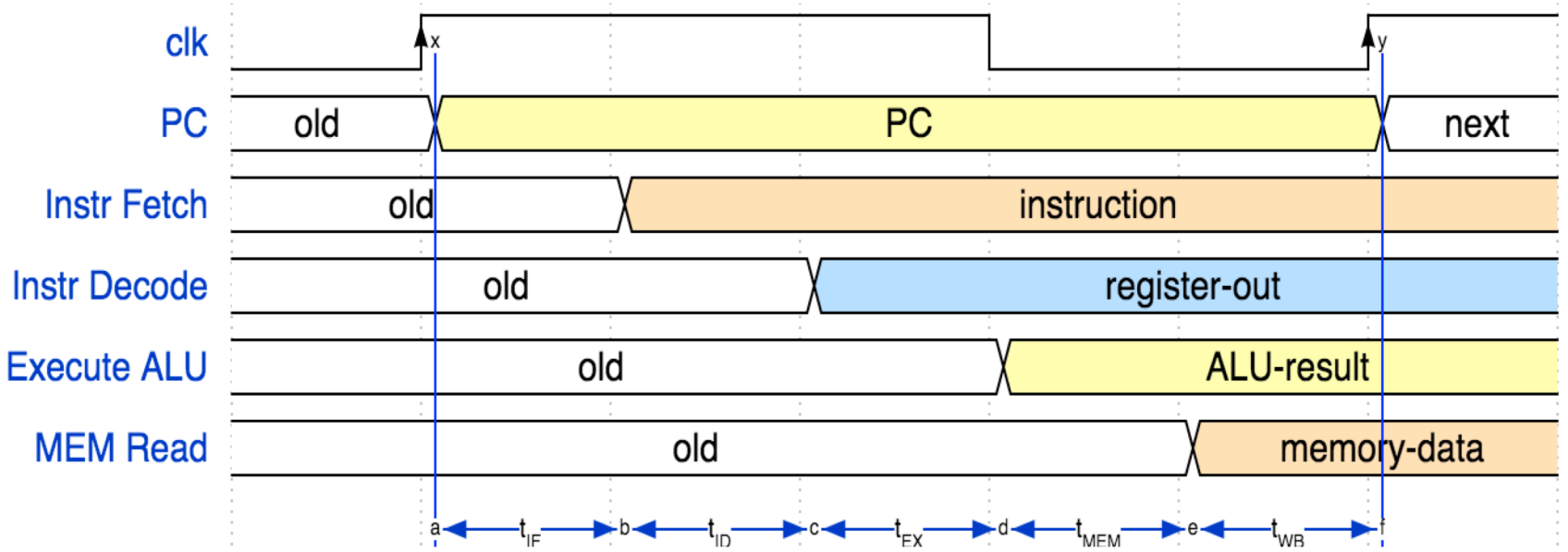
Instruction type encoded using only 9 bits
inst[30],inst[14:12], inst[6:2]

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

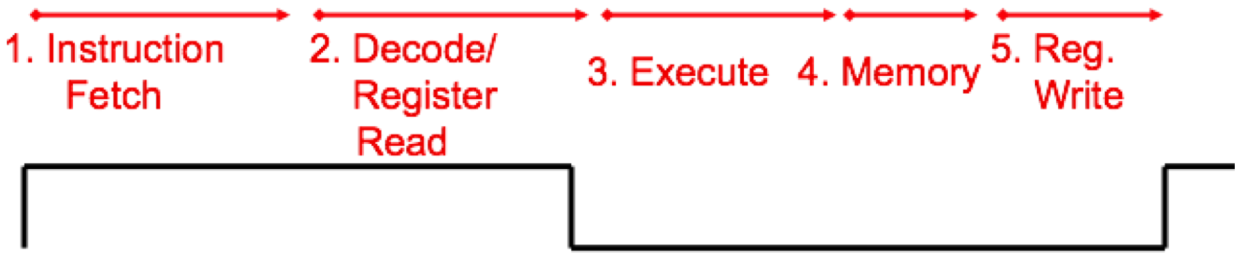
Agenda

- Quick Datapath Review
- Control Implementation
- Performance Analysis

Instruction Timing



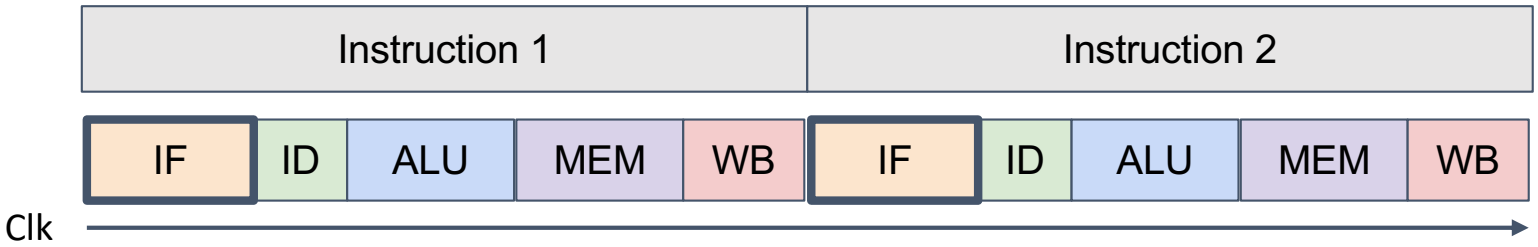
IF	ID	EX	MEM	WB	Total
IMEM	Reg Read	ALU	DMEM	Reg W	
200 ps	100 ps	200 ps	200 ps	100 ps	800 ps



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?

“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)

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

(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

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

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)

Question

If we reduce the time a program takes to execute by pipelining our CPU, which factor is likely to shrink?

A) Instructions per program
B) Cycles per instruction
C) Time per cycle
D) I appreciate Morgan's podcast recs
E) I do not appreciate Morgan's podcast recs

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

Question

If we reduce the time a program takes to execute by pipelining our CPU, which factor is likely to shrink?

A) Instructions per program
B) Cycles per instruction (This will increase!)
C) Time per cycle
D) I appreciate Morgan's podcast recs
E) I do not appreciate Morgan's podcast recs

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

Summary

- Implementing controller for your datapath
  - Ask yourself the questions on the beginning slides!
  - Work in stages, put everything together at the end!