Computer Architecture, Fall 2019

RISC-V CPU Datapath, Control Intro

Great Idea #1: Levels of Representation & Interpretation

Higher-Level Language Program (e.g. C)

Compiler

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

Assembler

Machine Language Program (RISC-V)

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

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

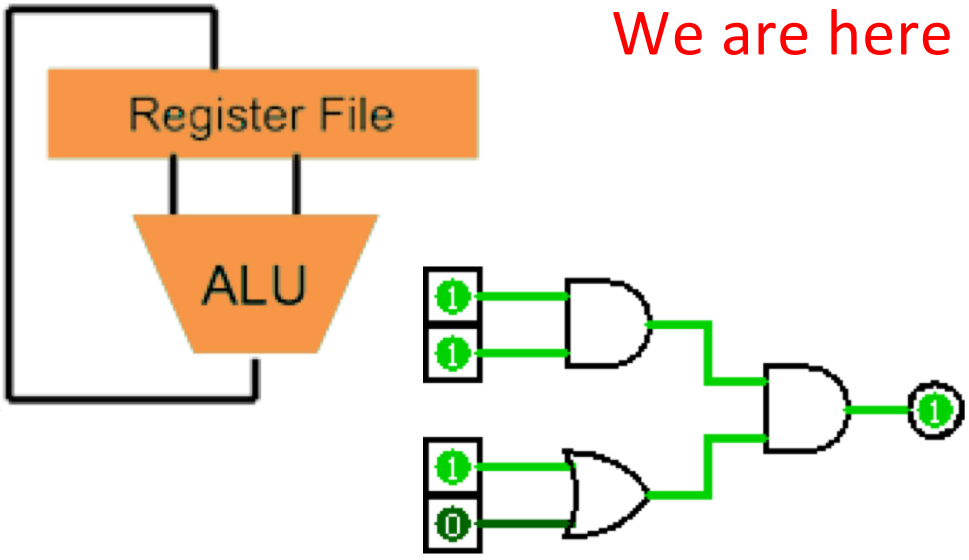
Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

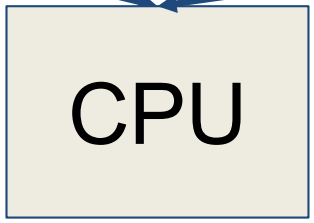
We are here



Putting it together!

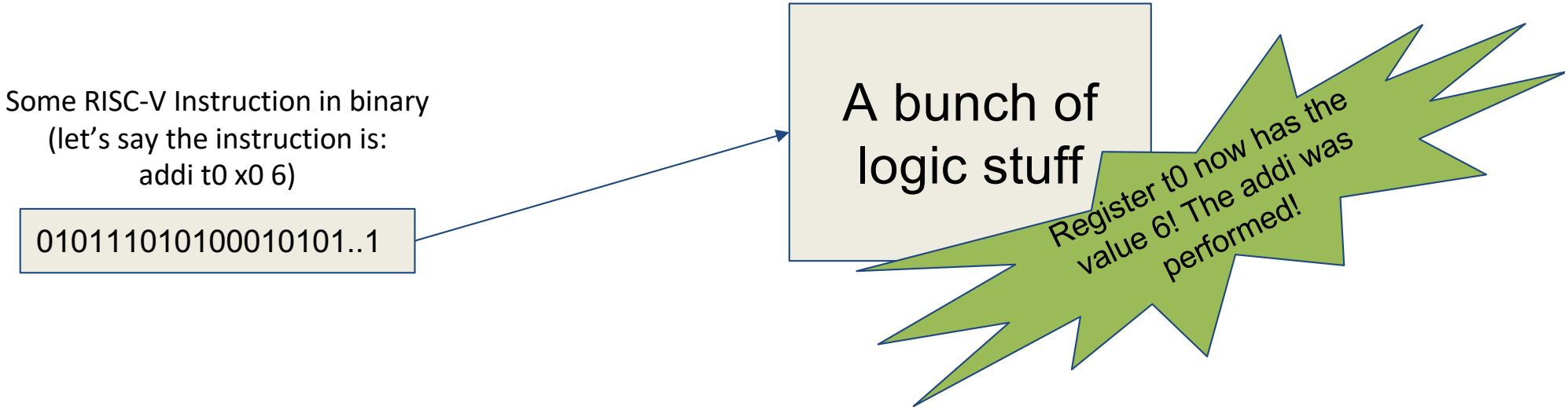
High level languages (ex. C) become machine language through compilation, assembly, and linking.

Registers, clock circuits, gates, and other logic devices are the fundamental building blocks of digital decision-making



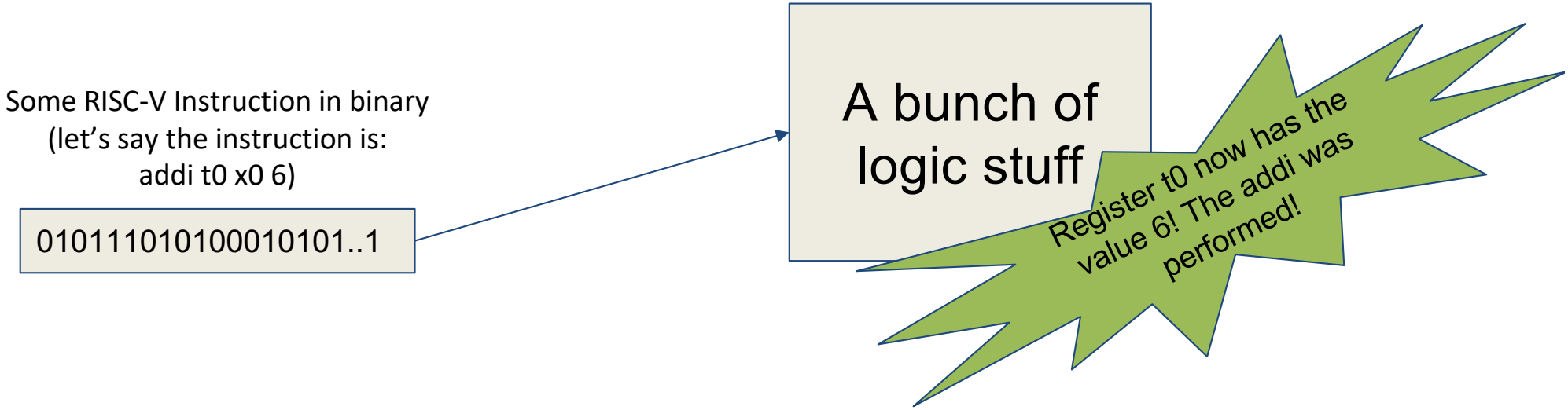
Today's goal:

Create a "circuit" of logic elements that, when given an assembly instruction, perform the action the instruction describes



Today's goal:

Create a "circuit" of logic elements that, when given an assembly instruction, perform the action the instruction describes



We'll call the logic stuff your CPU: Central Processing Unit

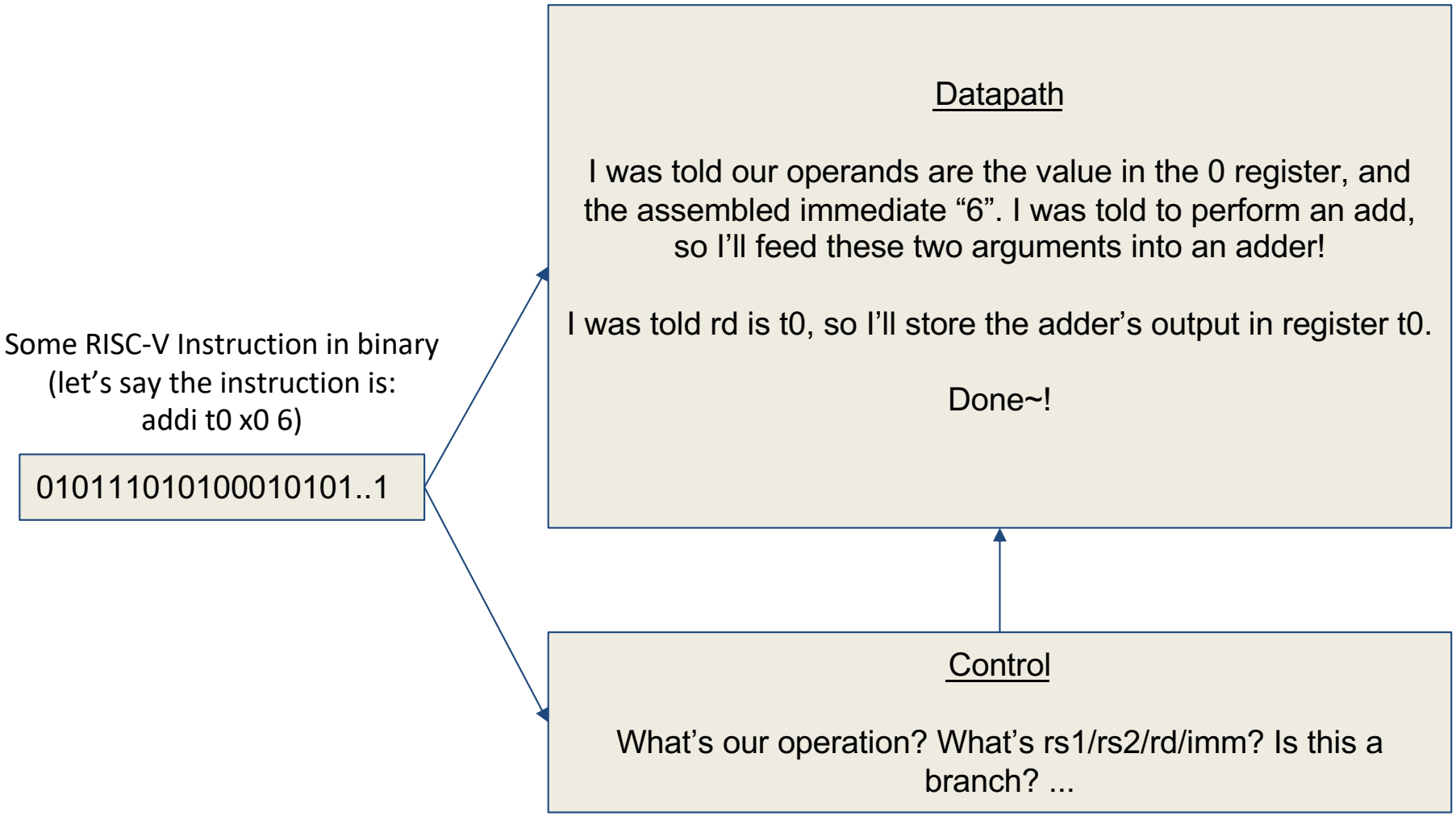
Agenda

- What's a CPU?
- Building from what we know
- Administrivia
- Our CPU
- Processor Design Principles

Your CPU in two parts

- ***Central Processing Unit (CPU):***
  - *Datapath:* contains the hardware necessary to perform operations required by the processor
    - Reacts to what the controller tells it! (ie. “I was told to do an add, so I”ll feed these arguments through an adder)
  - *Control:* decides what each piece of the datapath should do
    - What operation am I performing? Do I need to get info from memory? Should I write to a register? Which register?
    - Has to make decisions based on the input instruction only!

Your CPU in two parts



Designing our Datapath: Where to start?

- Let's start with a broad question:
  - What operations does our datapath need to be capable of performing?

- And also maybe a more specific one:
  - How can we ensure, when we build this, that all RISC-V instructions will be supported?

- Talk with your neighbours! … and then I wanna hear from you!

Designing our Datapath: Where to start?

- 6 different formats: R, I , S, SB, U, UJ
  - Arithmetic, Immediate, Store, Branch, Upper-immediate, JAL
- Instructions are classified into these formats based on their behaviors, meaning each type does something a little different!
  - If we're building a CPU to run /all/ instructions, we'll need to figure out what functionalities each type needs → support them all!

Agenda

- What's a CPU?
- Building from what we know
- Our CPU
- Processor Design Principles

Working with an ‘R-type’…

Work with the people around you. What needs to happen for an ‘R-type’ inst to execute?

- Wanna work with an example? Use:

```
add t0 t2 t3
```

Come up with a list of actions, then I’ll take a volunteer :)

A: Okay, I will do this
B: I don’t understand what you want
C, D, E: I want to press another letter

Working with an ‘R-type’

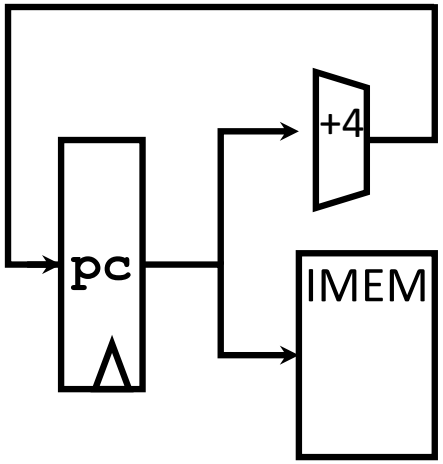
What needs to be done before our R type is executed or evaluated?

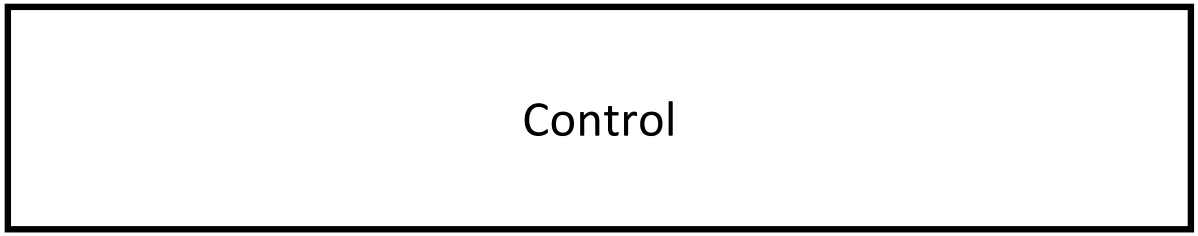
- (1) Get the instruction
- (2) Parse instruction fields (rd, rs1, rs2, operation…)
- (3) Read data based on parsed operands
- (4) Perform operation
- (5) Write result to our destination register

Working with an ‘R-type’

- (1) Get the instruction
  - add t0 t2 t3
- (2) Parse instruction fields (rd, rs1, rs2, operation…)
  - rd = t0 rs1 = t2 rs2 = t3
- (3) Read data based on parsed operands
  - R[t2] R[t3]
- (4) Perform operation
  - R[t2] + R[t3]
- (5) Write result to our destination register
  - R[t0] = R[t2] + R[t3]

Implementing R-Types

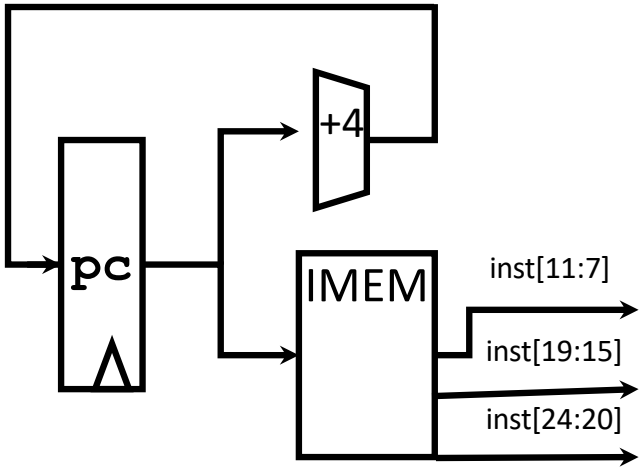


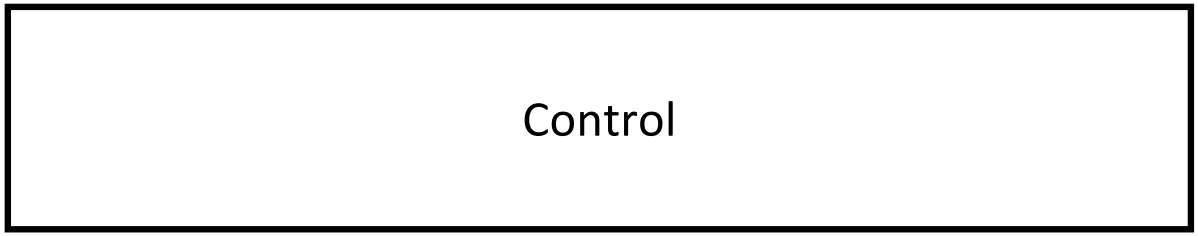


(1) Get the instructio

- PC holds the address of our current instruction
- Where are the bits making up our inst stored?
- How does PC change after an R-Type is executed?

Implementing R-Types

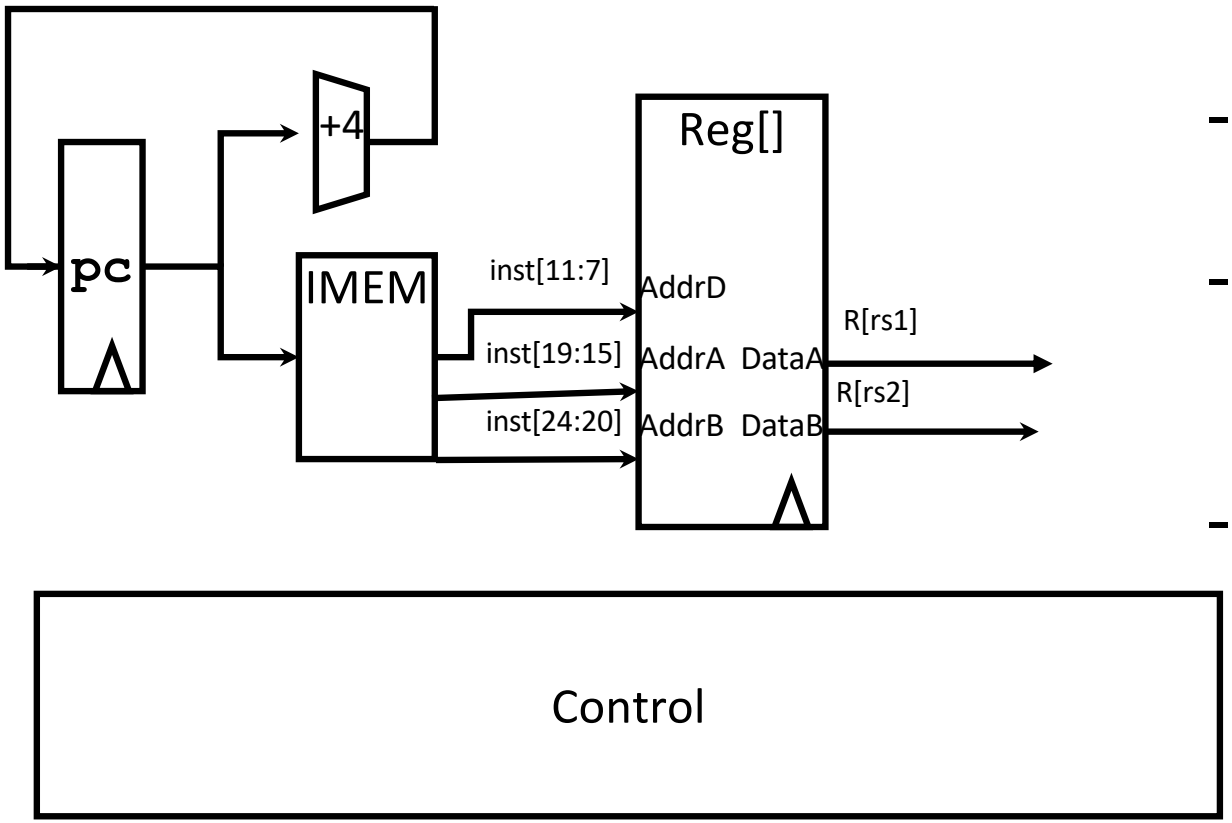




(2) Parse instruction fields (rd, rs1, rs2, operation…)

- What registers are we operating on?
- Where do they lie in our instruction format?
- How big is each field?

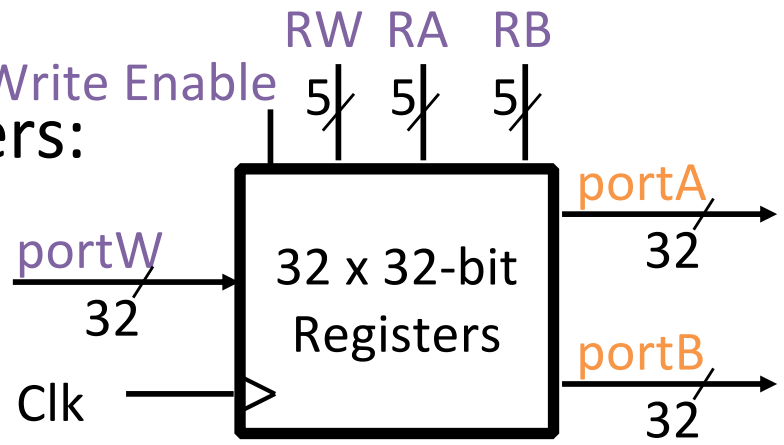
Implementing R-Types



(3) Read data based on parsed operands

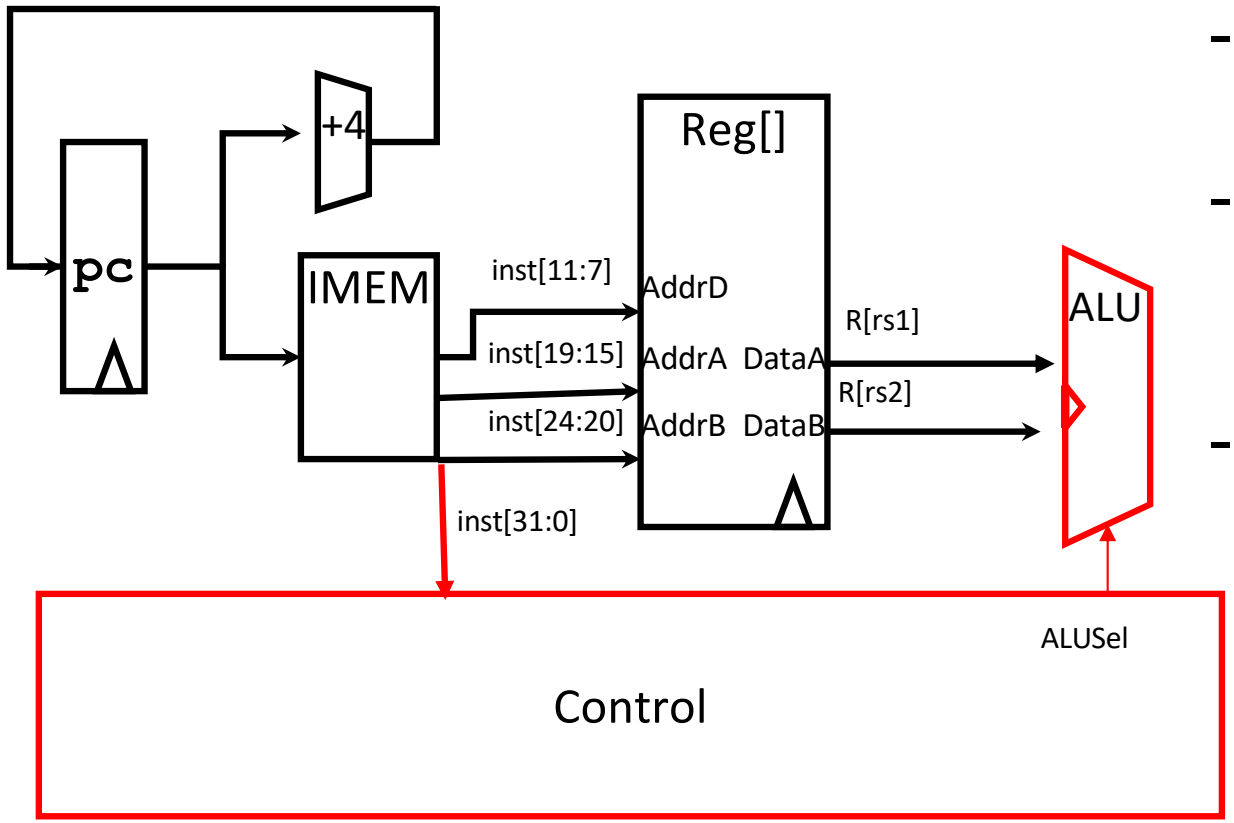
- New hardware: Register File
- Abstraction for all our registers (x0..x31 minus PC) and some mux'ing
- Reading, Writing happens here

Storage Element: Register File



- *Register File* consists of 31 registers:
  - Output ports portA and portB
  - Input port portW
- Register selection
  - Place data of register RA (number) onto portA
  - Place data of register RB (number) onto portB
  - Store data on portW into register RW (number) when Write Enable is 1
- Clock input (CLK)
  - CLK is passed to all internal registers so they can be written to if they match RW and Write Enable is 1

Implementing R-Types



(4) Perform operation

- New hardware: ALU (Arithmetic Logic Unit)
- Abstraction for adders, multipliers, dividers, etc.
- How do we know what operation to execute?
  - Our first control bit! ALUSel(ect)

But wait! There are many R-Type operations!

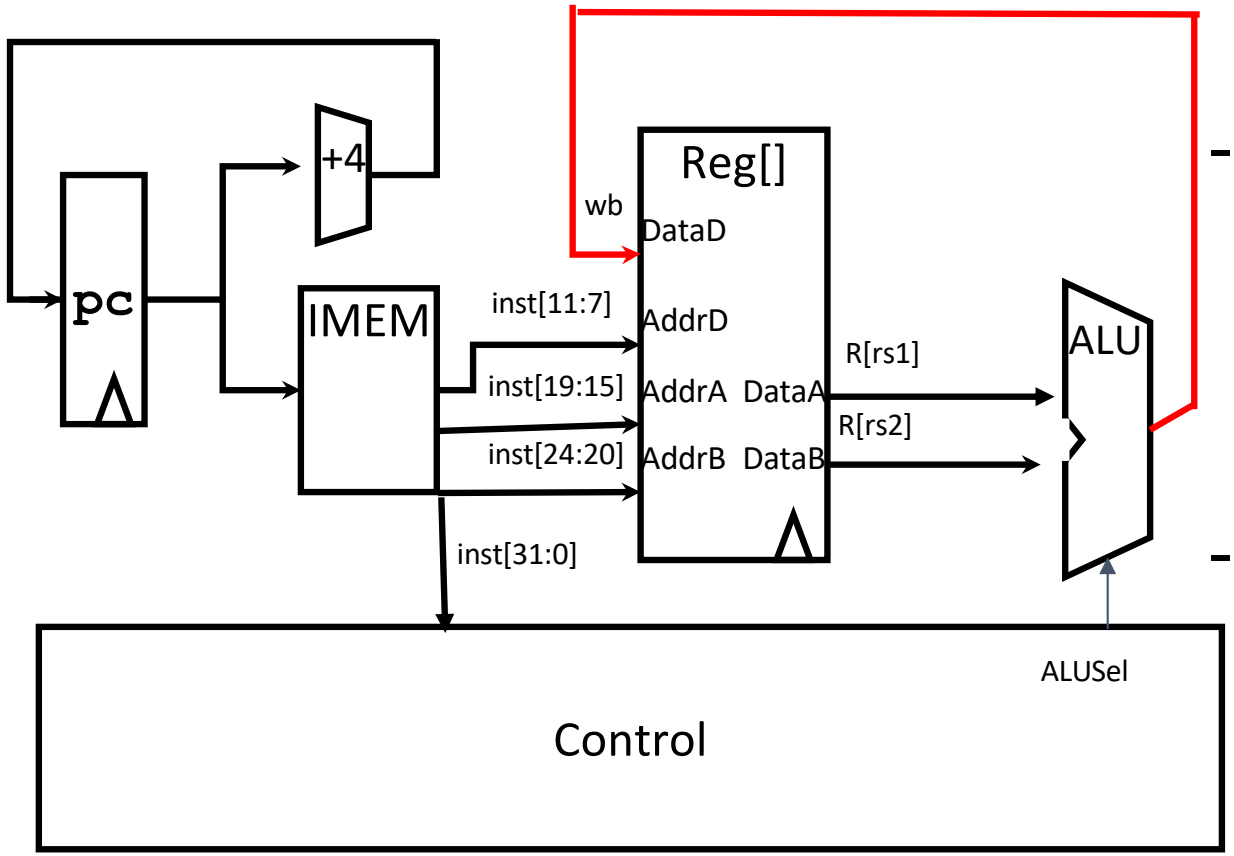
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

ALU == “Arithmetic Logic Unit”

ALUSel is a control bit which encodes the operation we should perform on the given operands

- The value of ALUSel is a mapping from `func3` and `func7` values to operations
- “if `func3` == 000 and `func7` == 0000000, perform an `add`”
- Multiple `func3`, `func7` combinations might lead to the same operation! (ie. `add` and `addi`)

Implementing R-Types



(5) Write result to our destination register

- The data we want to write is the result of computing operation on operands, ie. the output from our ALU
- Send it back to the regfile for writing

What changes with an arithmetic 'I-type'?

- A: Get the instruction
- B: Parse instruction fields (rd, rs1, rs2)
- C: Read data based on parsed operands
- D: Perform operation
- E: Write result to our destination register

Think: Do we need to add more hardware, more control, or more of both?

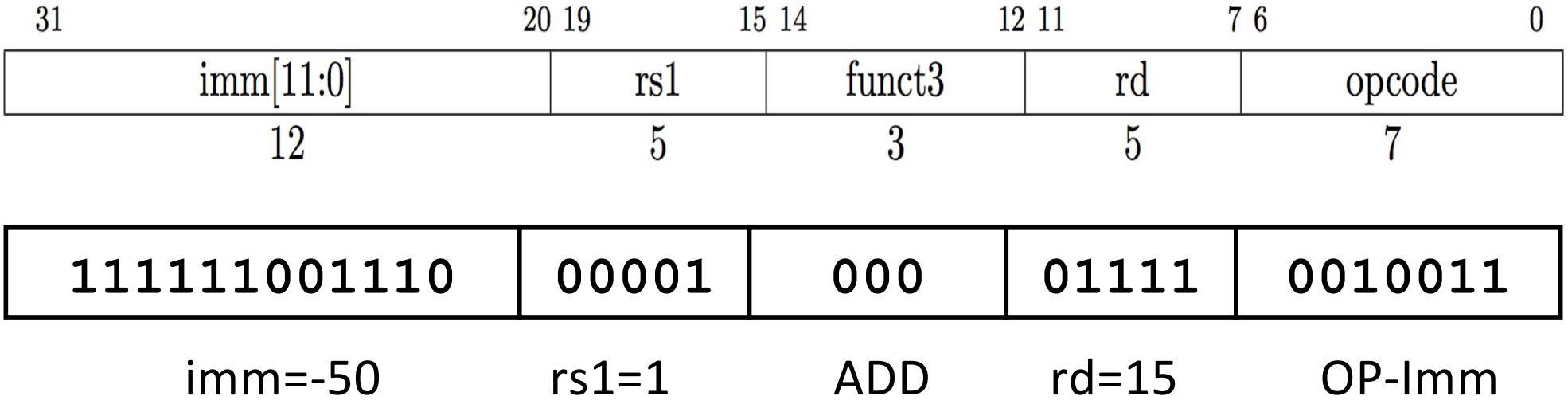
What changes with an arithmetic 'I-type'?

- A: Get the instruction
- B: Parse instruction fields (rd, rs1, rs2, operation…)
  - We also need to parse (and reassemble) our immediate!
- C: Read data based on parsed operands
  - Also an okay answer!
- D: Perform operation
- E: Write result to our destination register

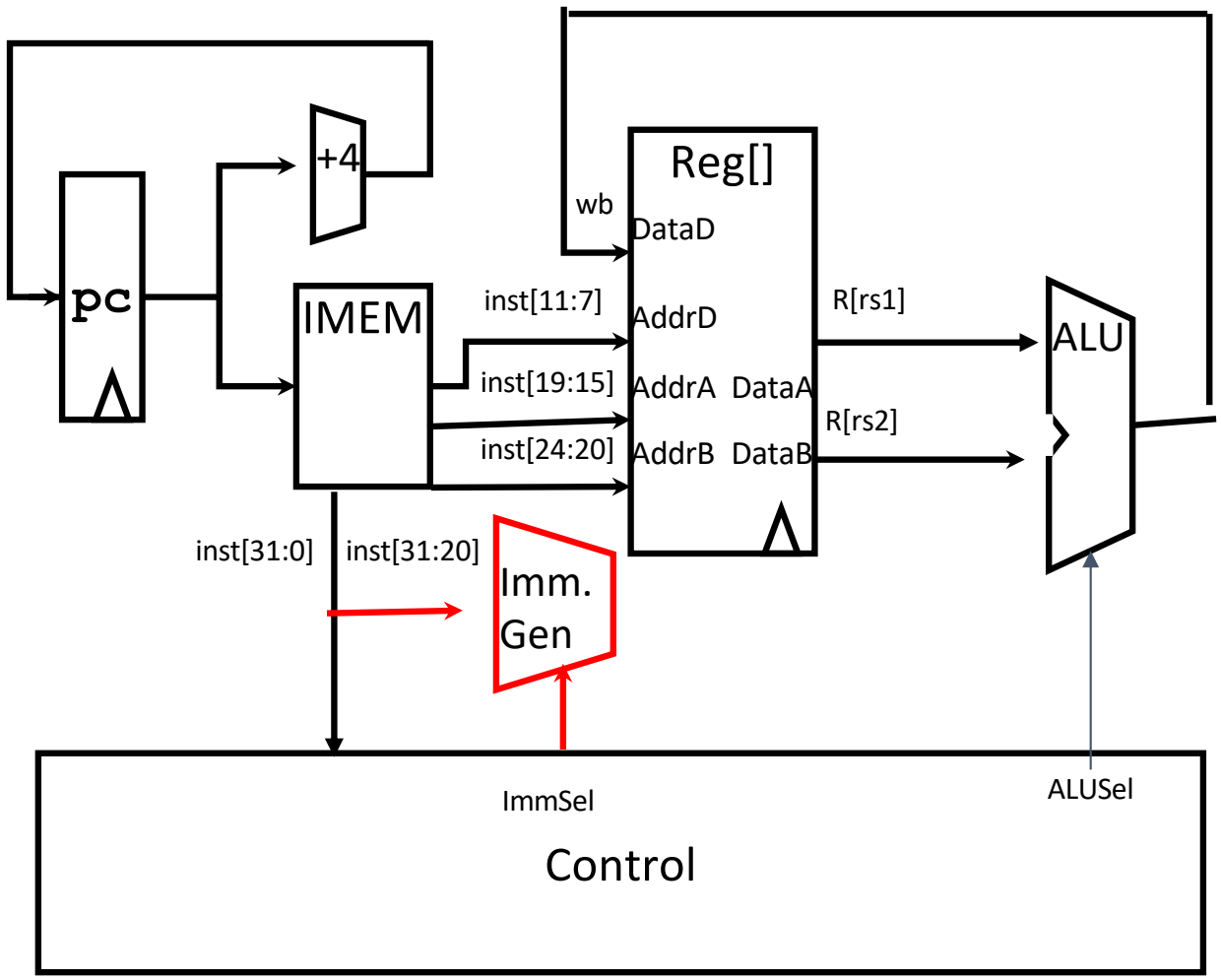
Implementing the `addi` instruction

- RISC-V Assembly Instruction:

```
addi   x15,x1,-50
```



Implementing arithmetic I-Types



(1) Get the instruction

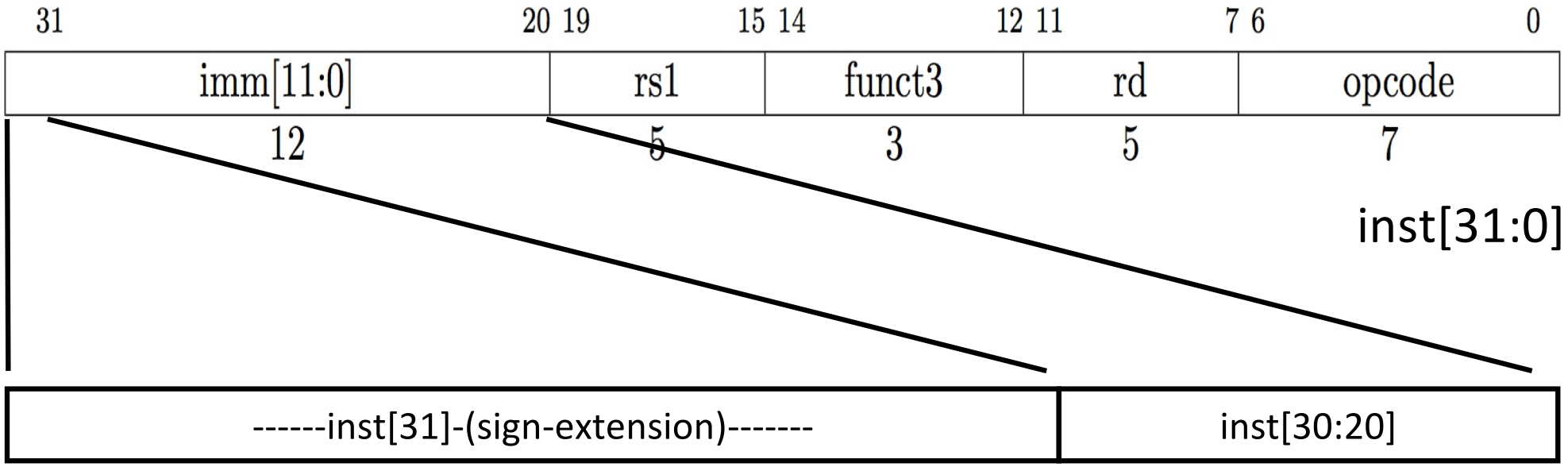
(2) Parse instruction fields (rd, rs1, rs2, operation, imm…)

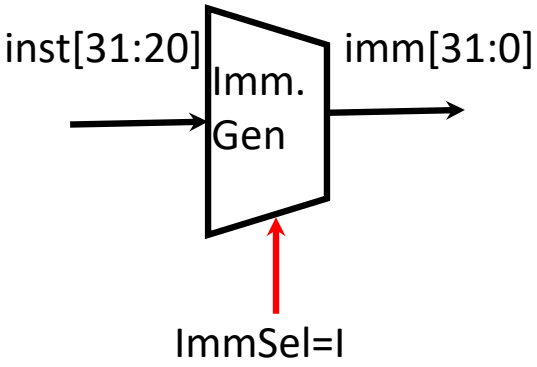
(3) Read data based on parsed operands

(4) Perform operation

(5) Write result to our destination register

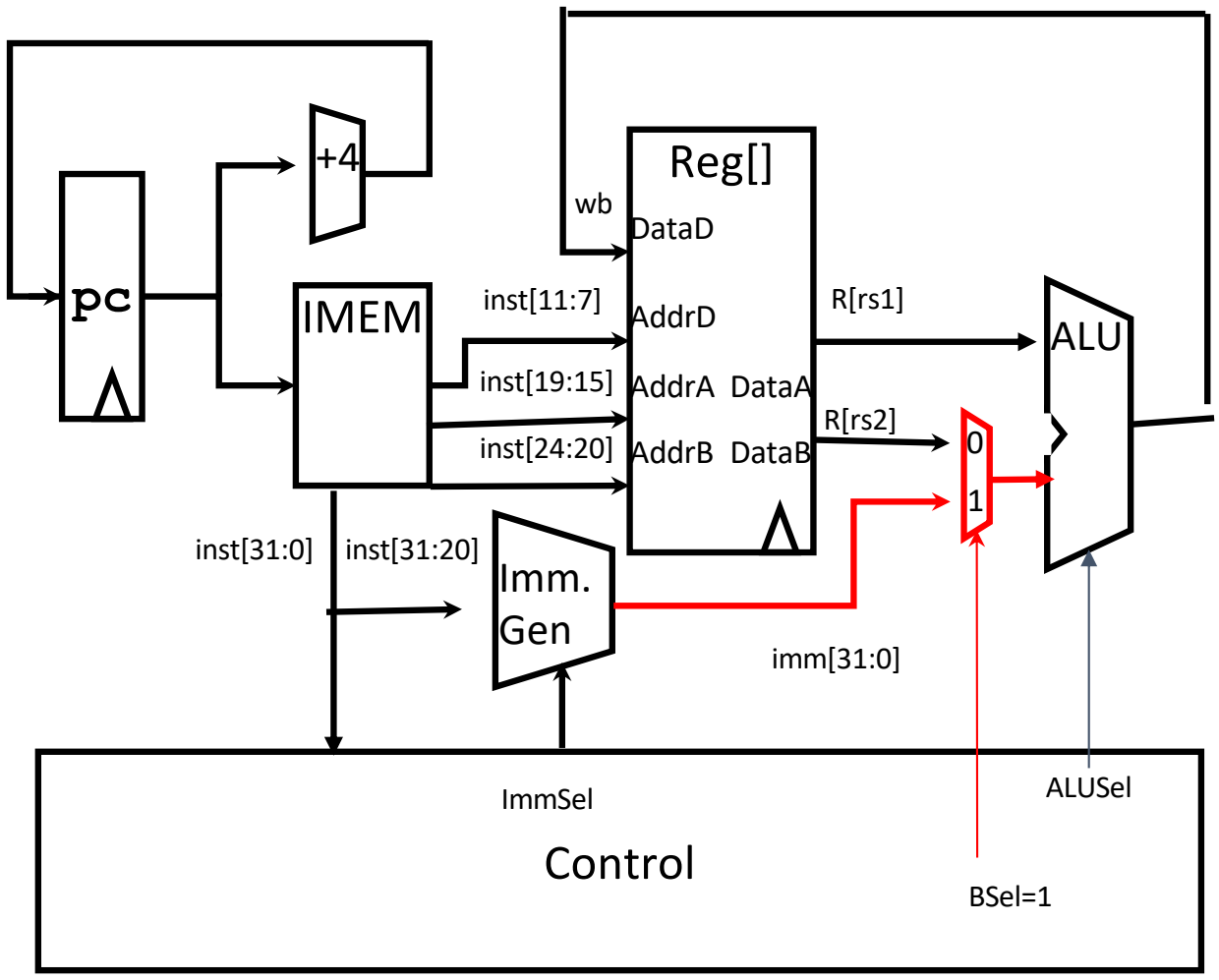
I-Format immediates





- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

Implementing arithmetic I-Types



(1) Get the instruction

(2) Parse instruction fields (rd, rs1, rs2, operation, imm…)

(3) Read data based on parsed operands

(4) Perform operation

(5) Write result to our destination register

But wait… Loads are ‘I-type’s also?

We know we can parse the immediate in the load-word format, but…

- What do we do with the immediate?
  - What operation should we perform?
- Maybe a better question: How do we know what the instruction does?

Any ideas?

But wait… Loads are ‘I-type’s also?

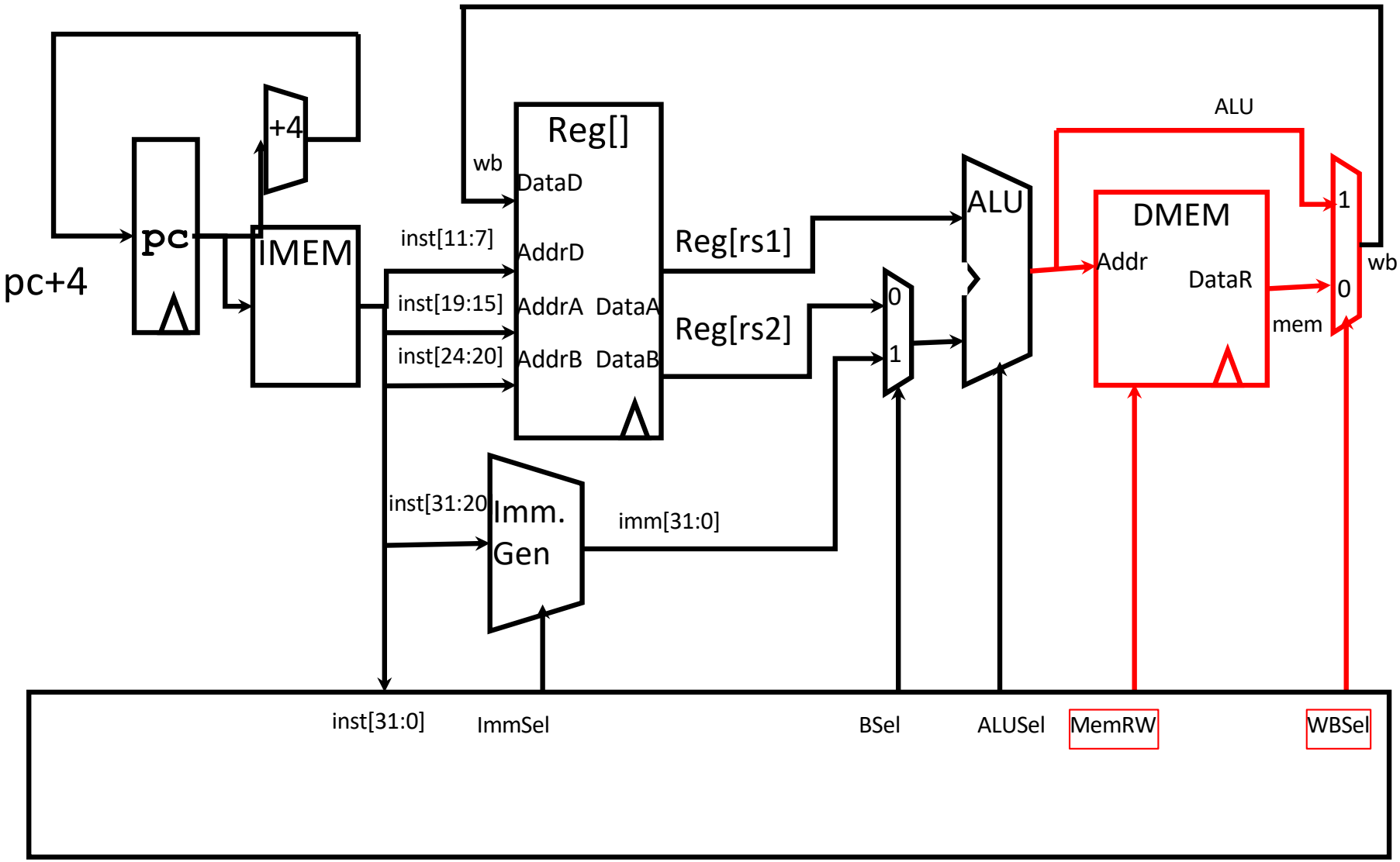
We know we can parse the immediate in the load-word format, but…

- What do we do with the immediate?
  - What operation should we perform?

R[rd] = (...)M[R[rs1] + imm](...)

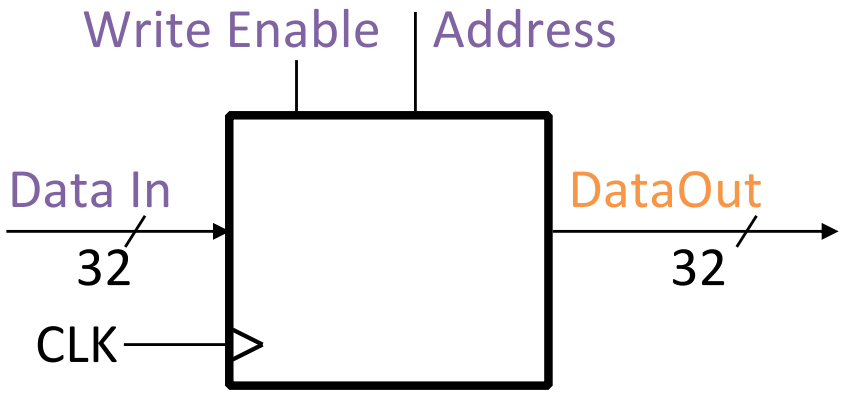
We need a (data) memory component!

Adding `lw` to datapath



Storage Element: Idealized Memory

- Memory (idealized)
  - One input port: Data In
  - One output port: Data Out
- Memory access:
  - Read: Write Enable = 0, data at Address is placed on Data Out
  - Write: Write Enable = 1, Data In written to Address
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational logic block: Address valid → Data Out valid after "access time"



A few notes on our new datapath…

- We have a lot of different components!
  - IMEM, Register file, ALU, DMEM
- Does every instruction need every component?
  - No! We got through all of the R-Types (and some of the I-Types) without DMEM
- Does any instruction need every component?
  - Yep! Loads!
  - This is the instruction which exercises our “critical path”

A few notes on our new datapath…

- Control refresher:
  - ALUSel: What operation are we performing?
  - RWEn: Should we write to our destination register?
  - BSel: (R[rs1] and R[rs2]) or (R[rs1] and imm)?
  - ImmSel: How should we reassemble the immediate?
  - MemRW: Should we read from memory or write to it?
  - WBSel: Do we want to write back to our destination register the ALU output or DMEM output?
- Often these values are encoded as binary values, but we may represent some (ie. ImmSel, ALUSel) with words instead (add, I-type, etc.)
- Are any of these dependent?
  - Do I care what value I'll write back (WBSel) if I'm not writing back (RWEn = 0)?

All RV32 Load Instructions

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

funct3 field encodes size and signedness of load data

- Supporting the narrower loads requires additional circuits to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.
- We'll assume these are implemented in the DMEM module (not our datapath) and won't add them to our schematic

Agenda

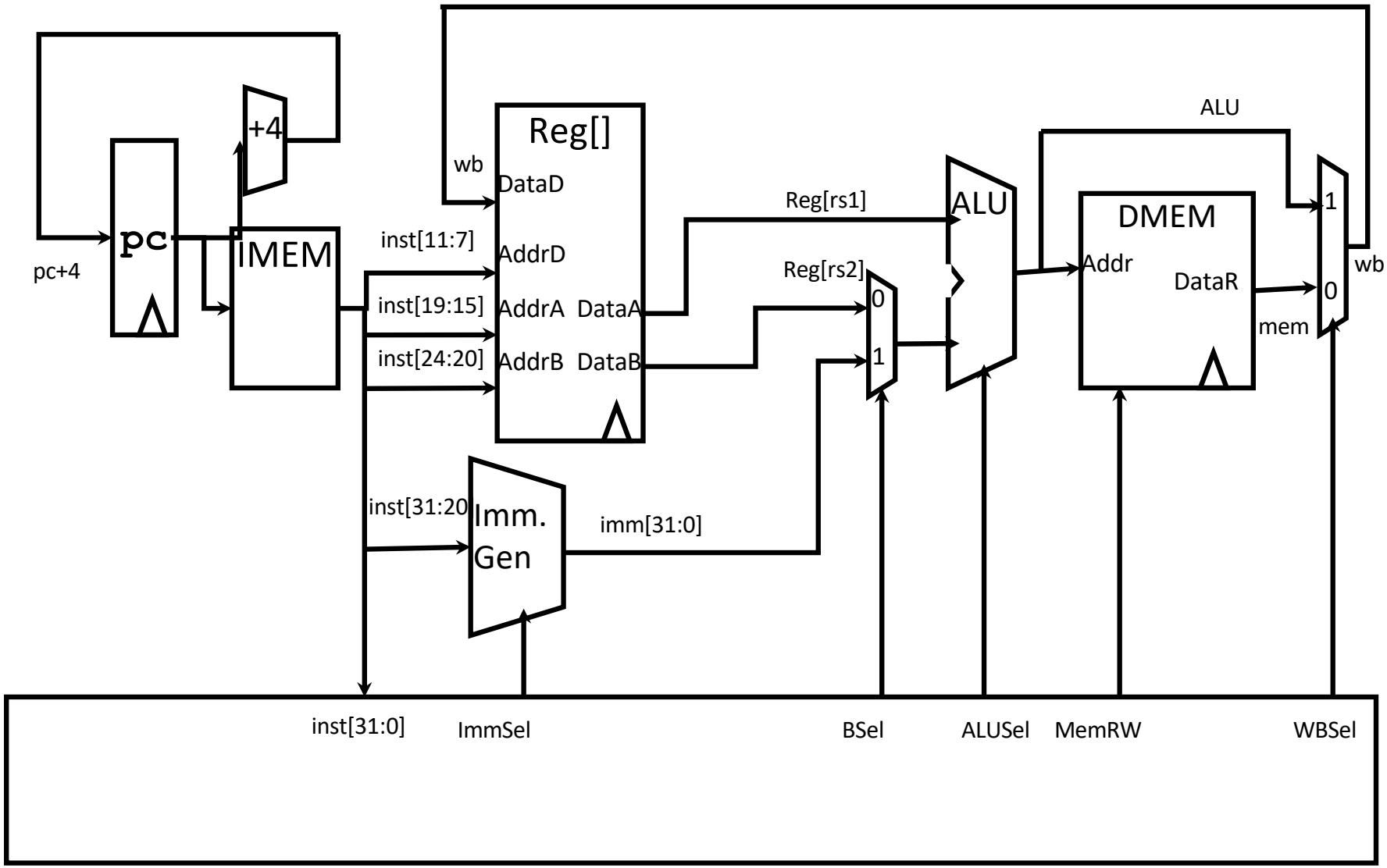
- Building from what we know
- Our CPU
- Processor Design Principles

This lecture is long… what have we done again?

- Added ‘R-type’ and ‘I-type’ instructions to our data path!
- We still need to figure out how to do S, SB, U, and UJ types
  - Don’t worry, we actually have all the big pieces we need!
- We talked about control bits and how they instruct our hardware to deal with many different kinds of instructions

- Let’s keep going!

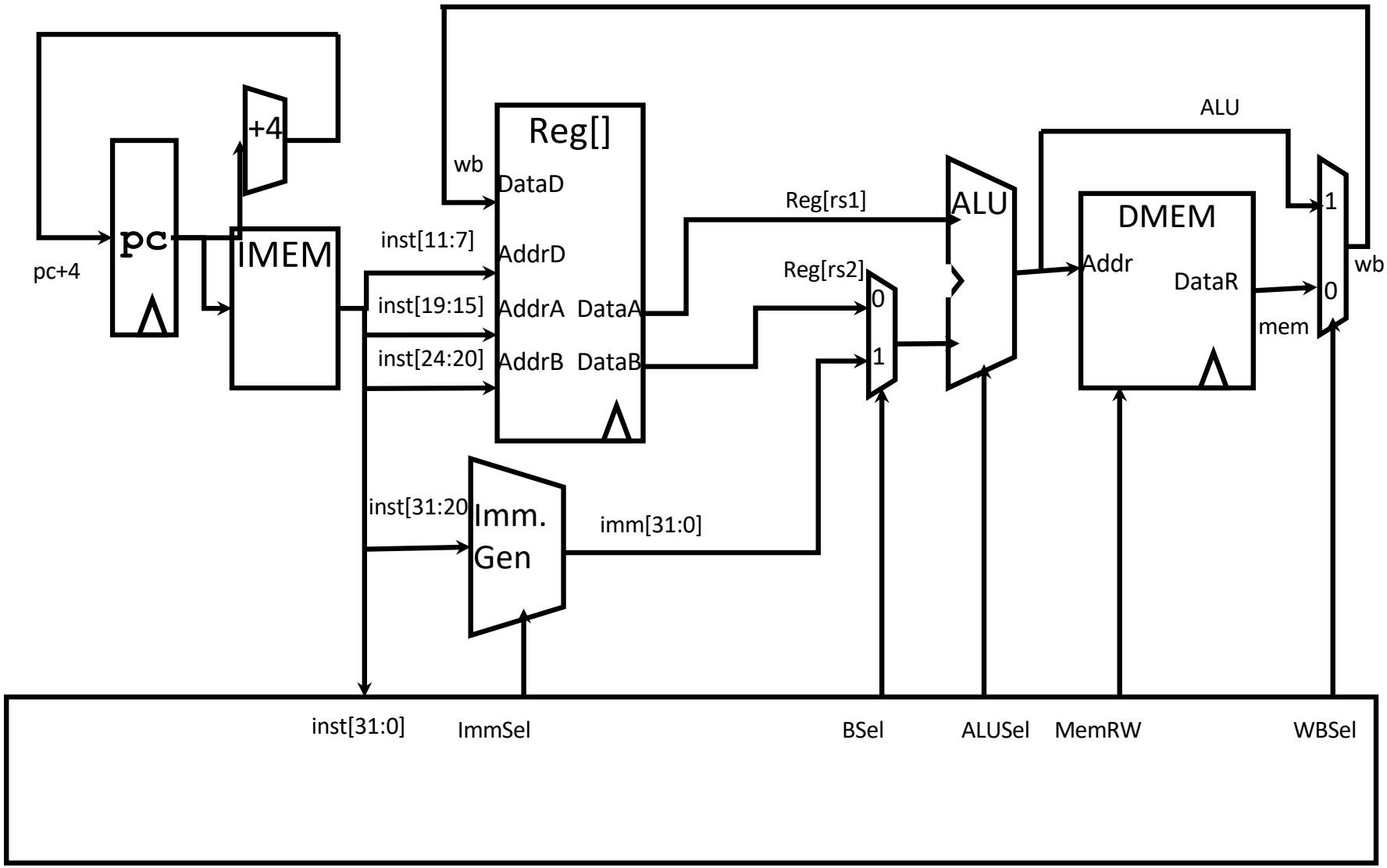
Current Datapath



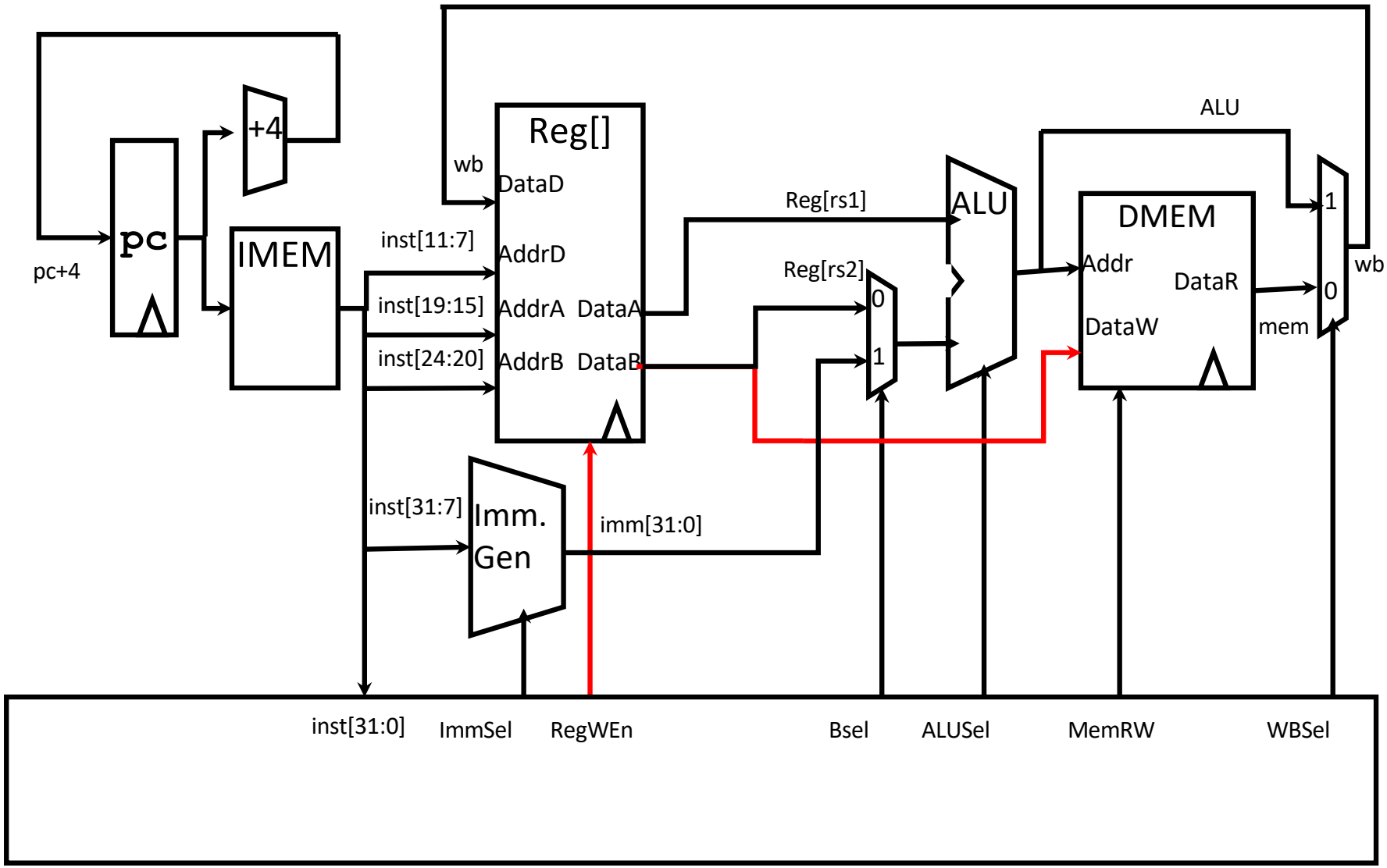
How can we support ‘S-type’ instructions?

- Do we need to change hardware, control, or both?
  - Can we fetch our instruction?
    - YES!
  - Can we decode our instruction?
    - YES! (Change ImmSel to ‘S-type’)
  - Can we select the operands we need?
    - YES! (still R[rs1] and immediate)
  - Do we have support for the operation we’re performing?
    - YES! (ALUSel = Add)
  - Can we access memory correctly?
    - YES! (MemRW should be 1 for ‘write’)
    - But… what data are we writing? We need to pass R[rs2] to DMEM
  - Do we need to update R[rd]?
    - No! Stores read from R[rs2] but don’t write, so we’ll need a new control bit.
- Both hardware and control!

Current Datapath



Adding **sw** to datapath

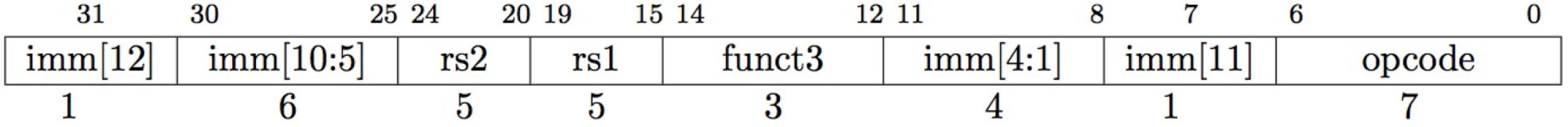


Modifying Program Execution (branching, jumping)

- So far we've been working with instructions that modify the contents of registers in our reg file or instructions that modify memory
- We also have instructions which modify how the program executes!
  - These change the value of our PC register from the "next" instruction (PC + 4) to an instruction at a label or register address
    - These instructions are either PC-relative (add an immediate to PC) or absolute (encode an actual address that PC is changed to)

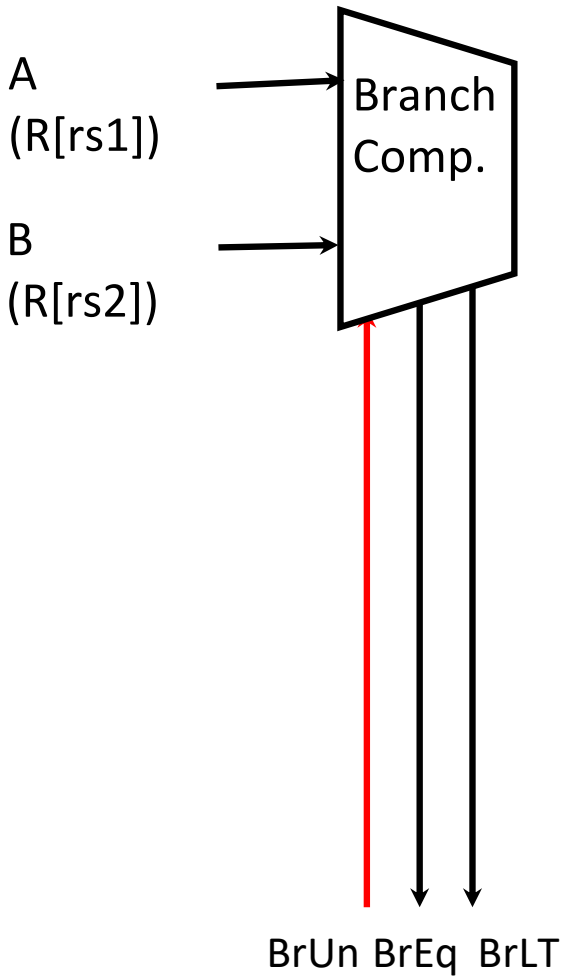
Let's start with branching!

Implementing Branches



- SB-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- Branching involves two operations:
  - Comparison (R[rs1] == R[rs2], possibly <, >, etc.)
  - Addition (PC = PC + Imm)
- Our ALU can do one of these, but not both! We need to add more hardware!

Branch Comparator

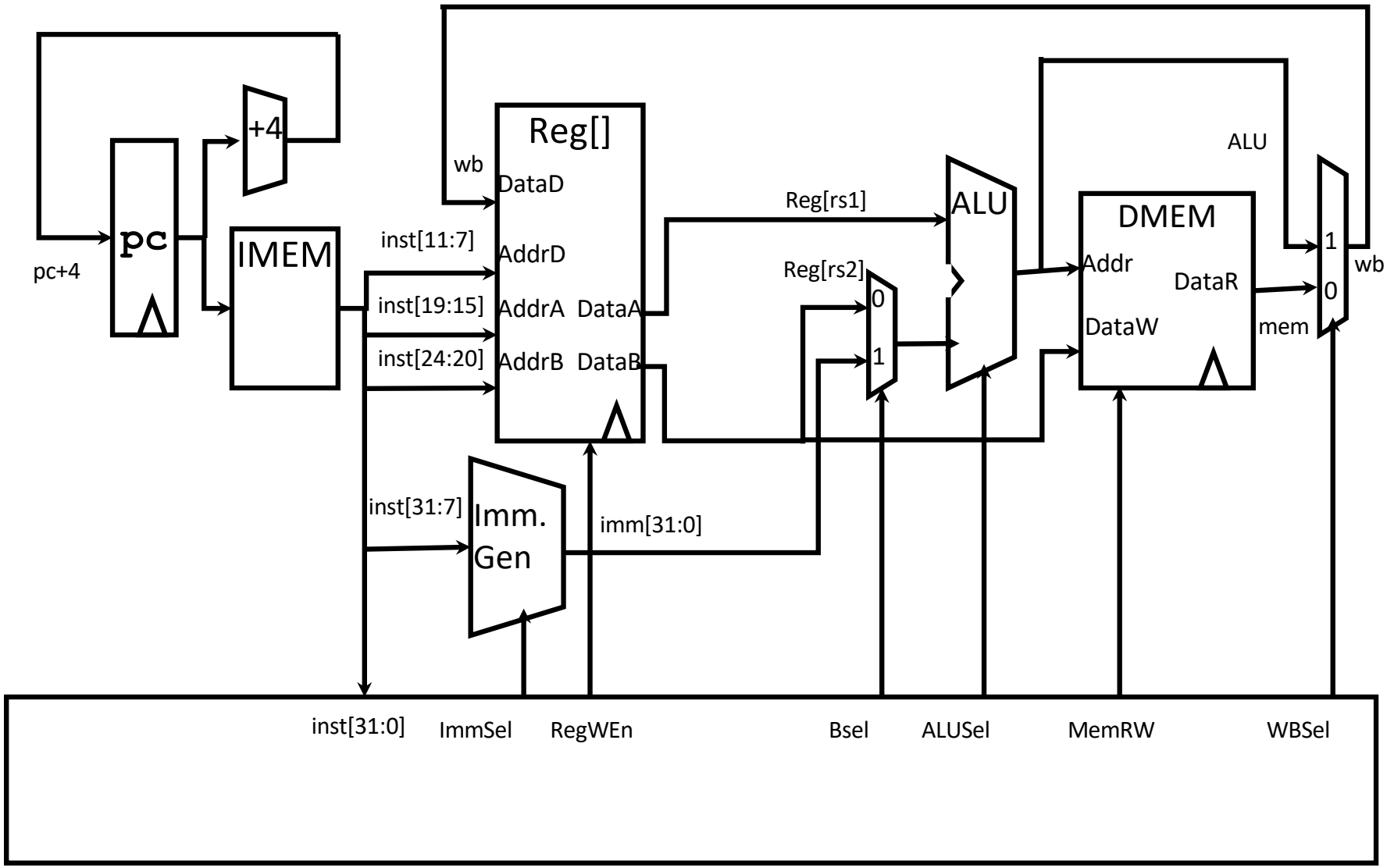


- BrEq = 1, if A=B
- BrLT = 1, if A < B
- BrUn =1 selects unsigned comparison for BrLT, 0=signed

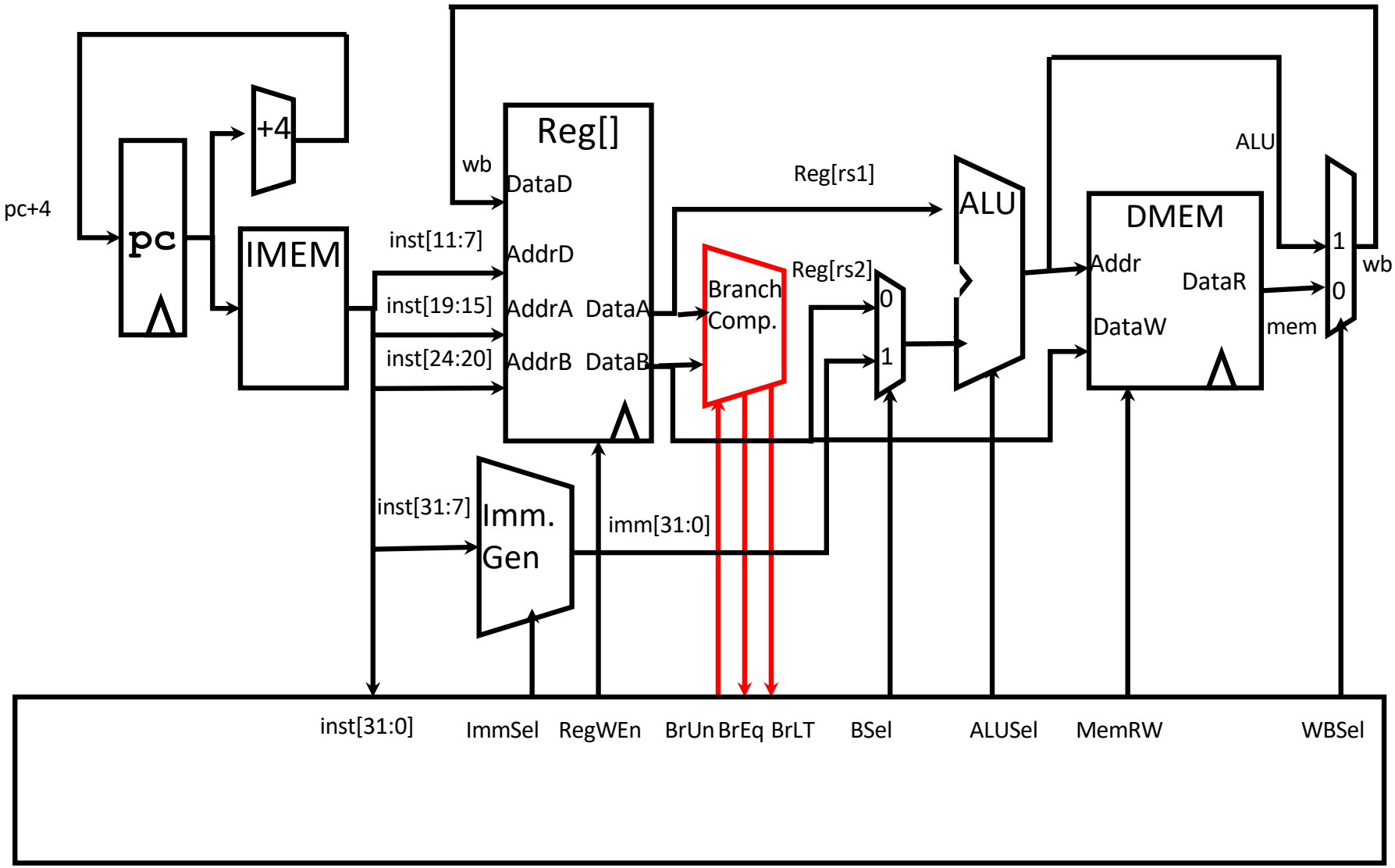
- BGE branch: A >= B, if !(A<B)

"Output" control signals are used to set other controls! (ex. we only want to write to PC if our branch succeeded)

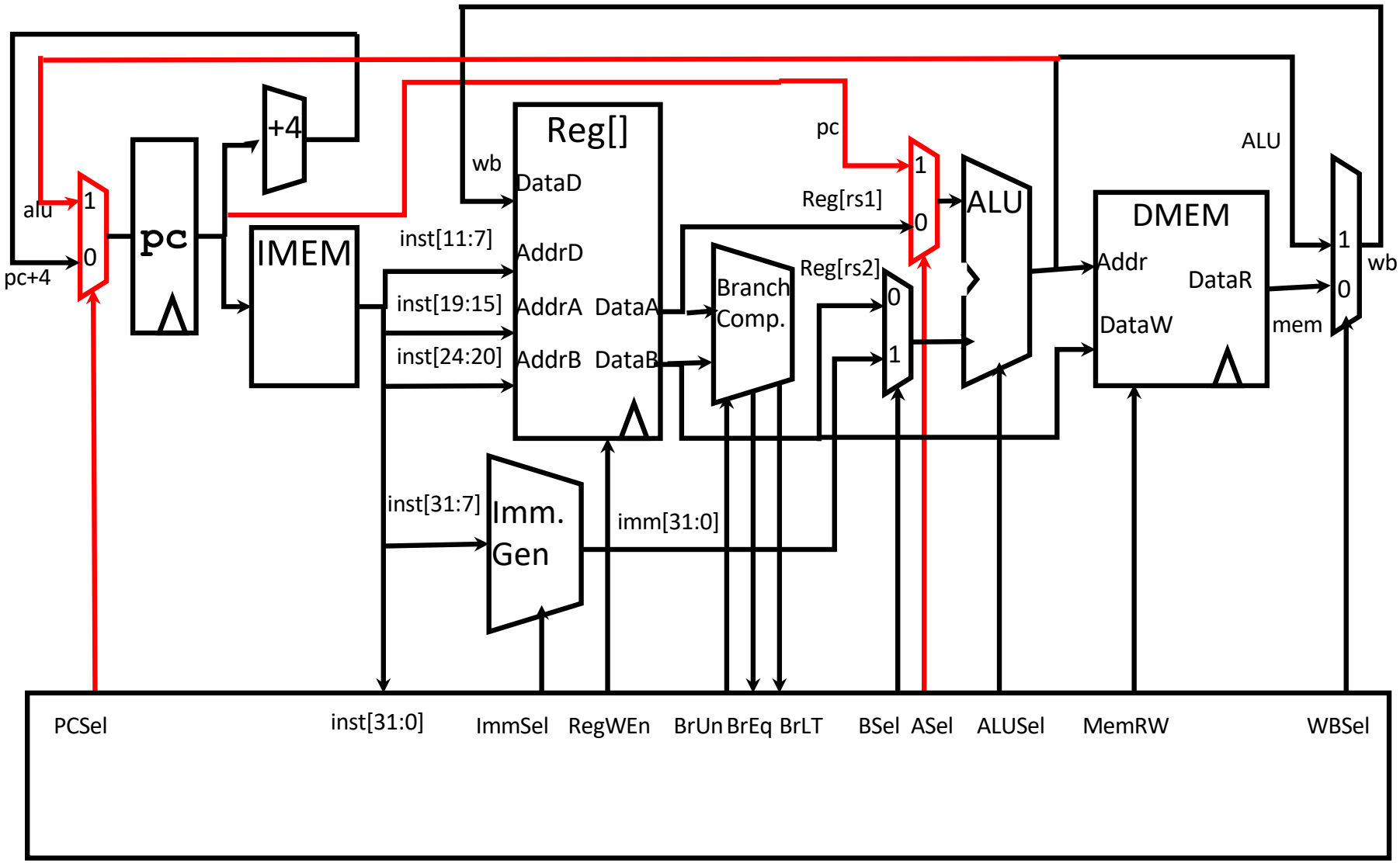
Current Datapath



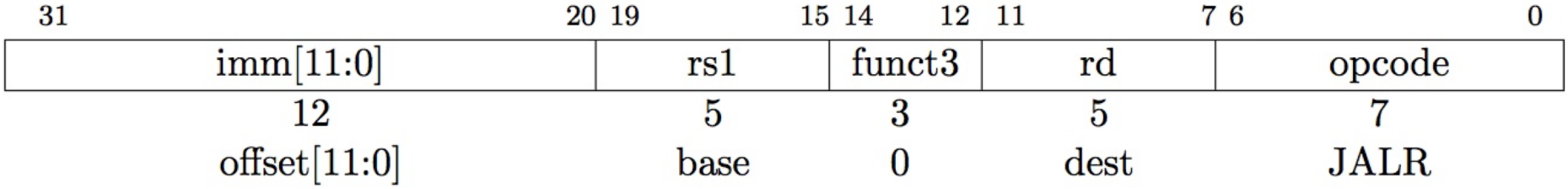
Adding branches to datapath



Adding branches to datapath

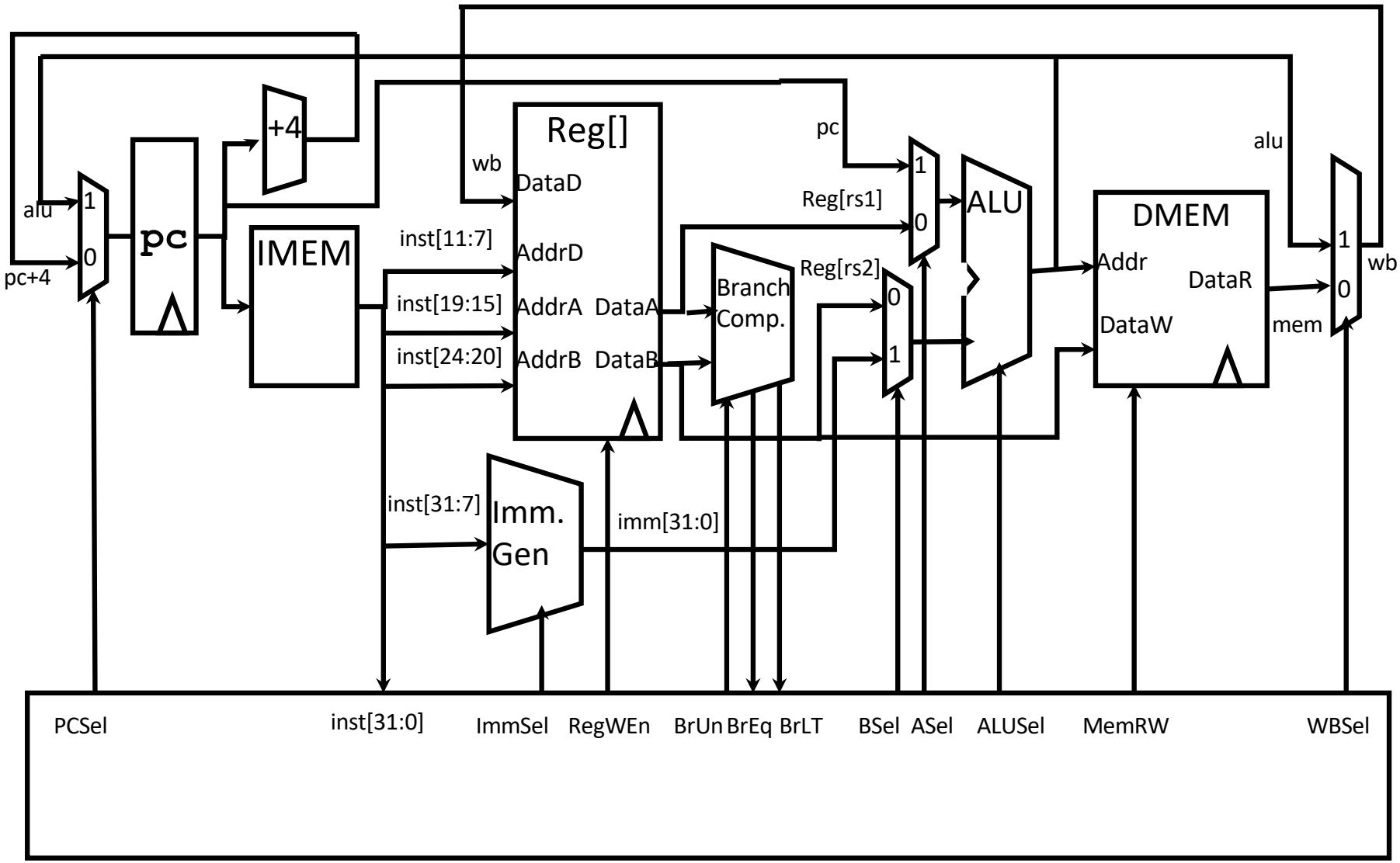


Implementing **JALR** Instruction (I-Format)

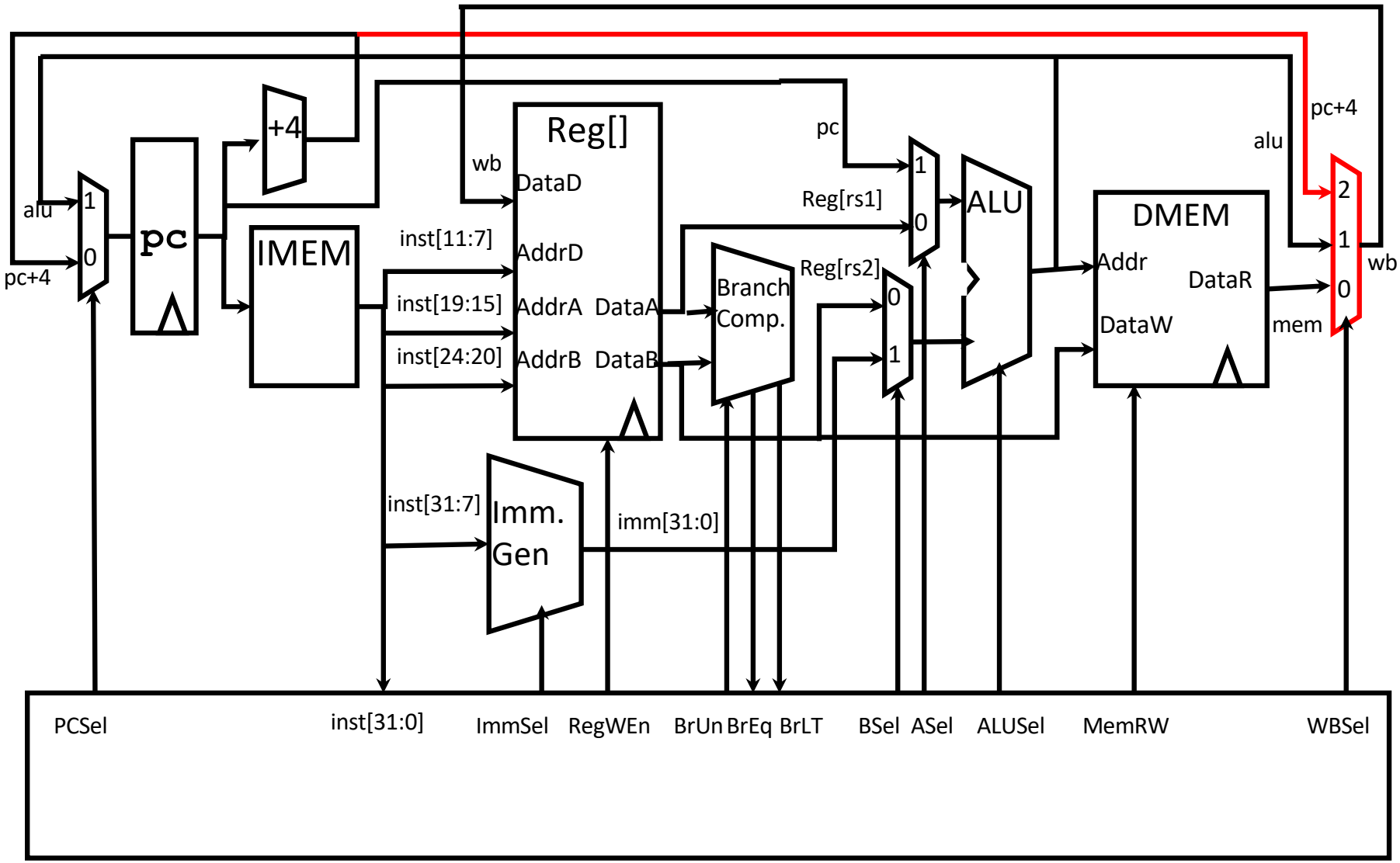


- JALR rd, rs, immediate
  - Writes PC+4 to Reg[rd] (return address)
    - We need to change our MUX to allow this as a write back value!
  - Sets PC = Reg[rs1] + immediate
    - We can already feed these arguments into our ALU, so no change here
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes (different from branch!)

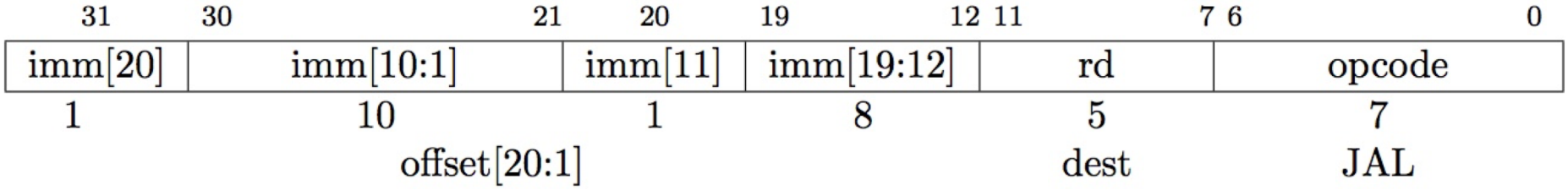
Current Datapath



Adding `jalr` to datapath



Lastly: Implementing `jal` Instruction

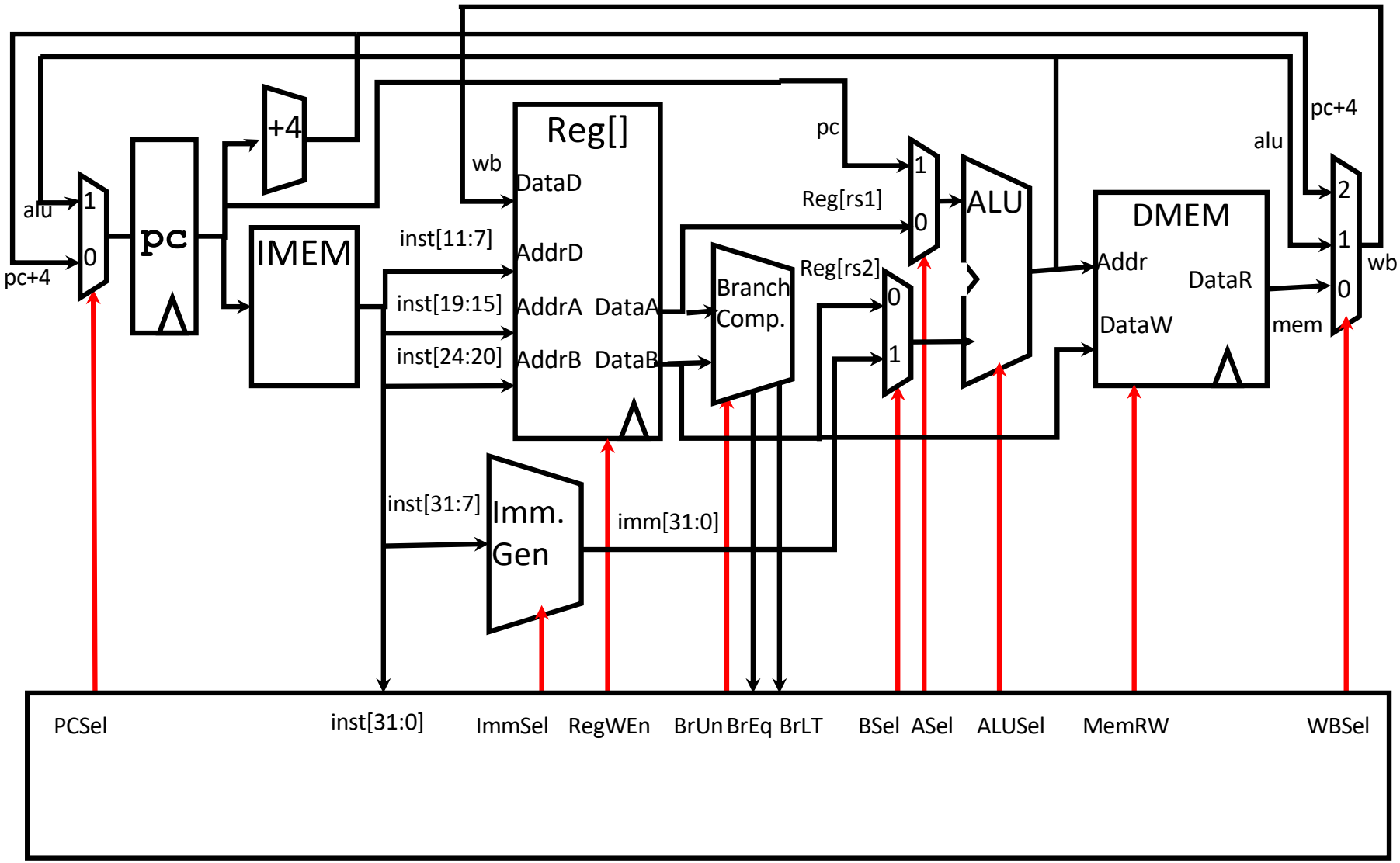


- JAL saves PC+4 in Reg[rd] (the return address)
  - Supported!
- Set PC = PC + offset (PC-relative jump)
  - Supported!
  - Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
    - $\pm 2^{18}$ 32-bit instructions
  - Immediate encoding optimized similarly to branch instruction to reduce hardware cost
- No changes needed! :)

Agenda

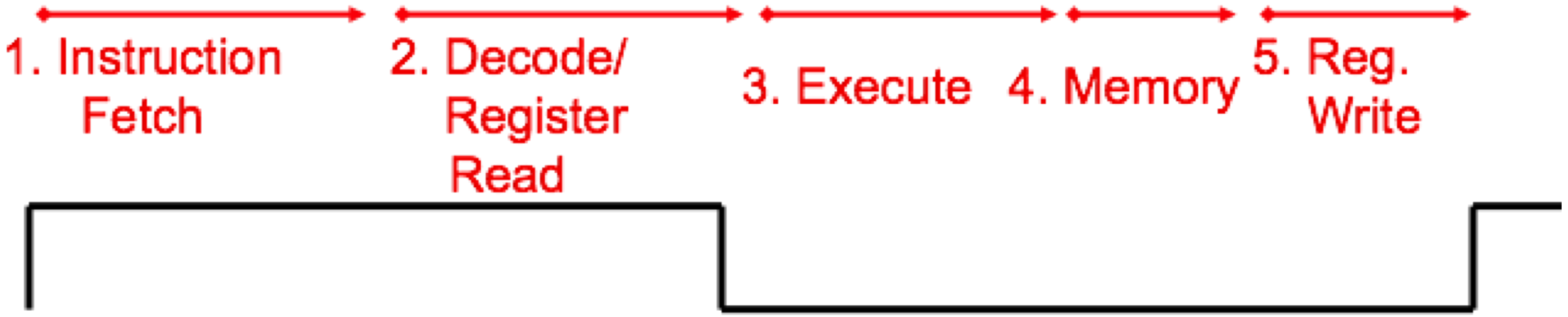
- What's a CPU?
- Building from what we know
- Our CPU
- Processor Design Principles

Single-Cycle RISC-V RV32I Datapath



CPU Clocking

- For each instruction, how do we control the flow of information through the datapath?
- Single Cycle CPU: All stages of an instruction completed within one long clock cycle
  - Clock cycle sufficiently long to allow each instruction to complete all stages without interruption within one cycle

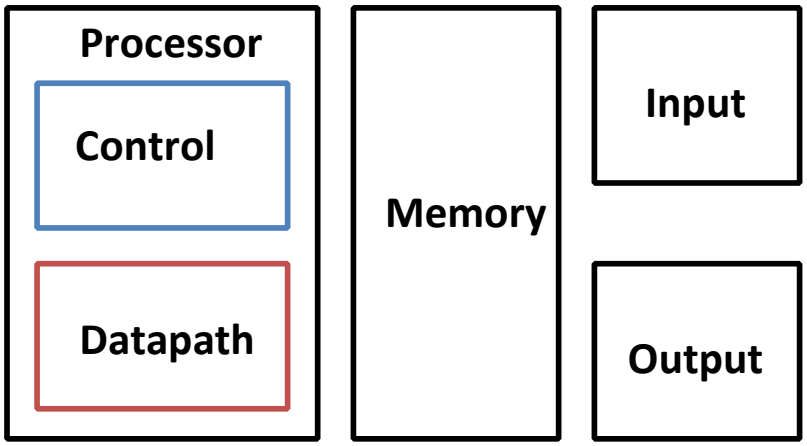


Agenda

- What's a CPU?
- Building from what we know
- Our CPU
- Processor Design Principles

Design Principles

- Five steps to design a processor:
  1) Analyze instruction set → datapath requirements
  2) Select set of datapath components & establish clock methodology
  3) Assemble datapath meeting the requirements



  4) Analyze implementation of each instruction to determine setting of control points that effects the register transfer
  5) Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits

Design Principles

- Determining control signals
  - Any time a datapath element has an input that changes behavior, it requires a control signal (e.g. ALU operation, read/write)
  - Any time you need to pass a different input based on the instruction, add a **MUX** with a control signal as the selector (e.g. next PC, ALU input, register to write to)
- Your control signals will change based on your exact datapath
- Your datapath will change based on your ISA

Summary !

- Universal 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 (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory), WB (Write Back)
  - Not all instructions are active in all phases (except for loads!)
- Controller specifies how to execute instructions
  - Worth thinking about: what new instructions can be added with just most control?