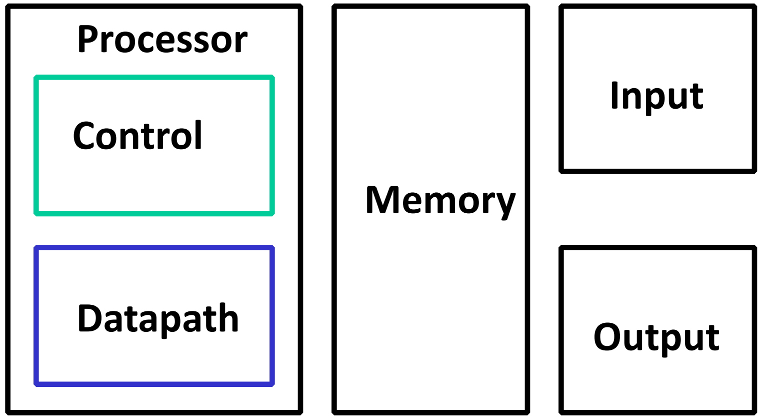
# RISC-V *CPU Datapath, Control Intro*

# 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



## 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?

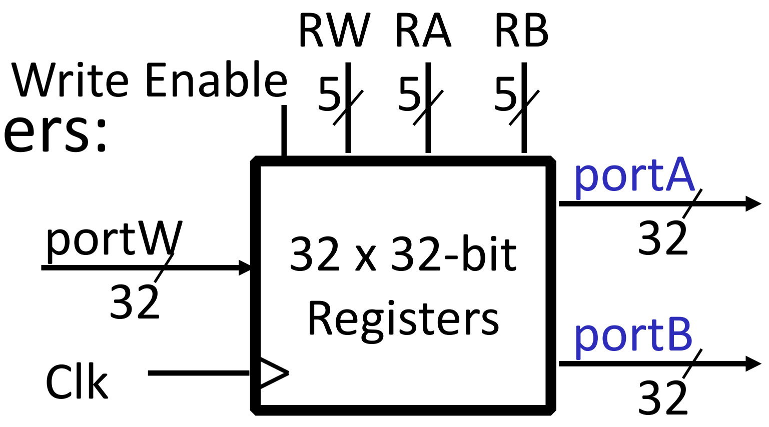
## Your CPU in two parts

- ***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!

## Design Principles

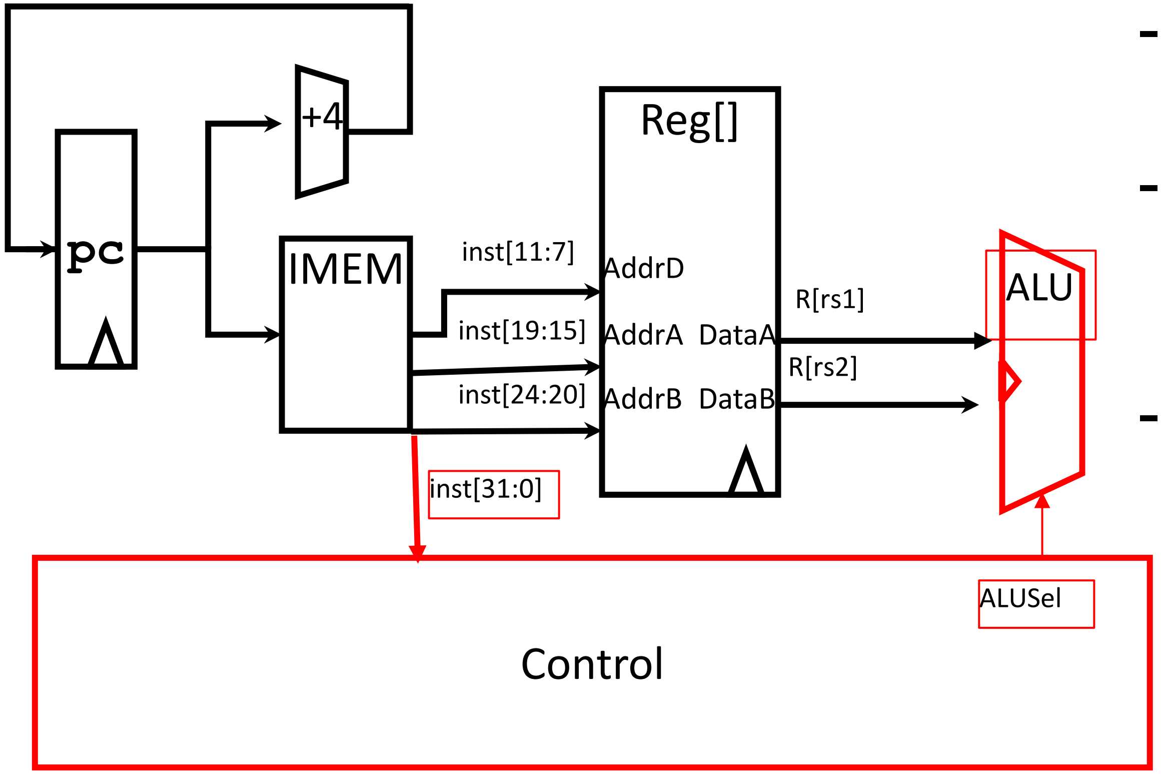
- Determining control signals
  - Any time a datapath element has an input that changes behavior, it requires a control signal (e.g. ALU operation, read/write)
  - Any time you need to pass a different input based on the instruction, add a **MUX** with a control signal as the selector (e.g. next PC,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

# 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

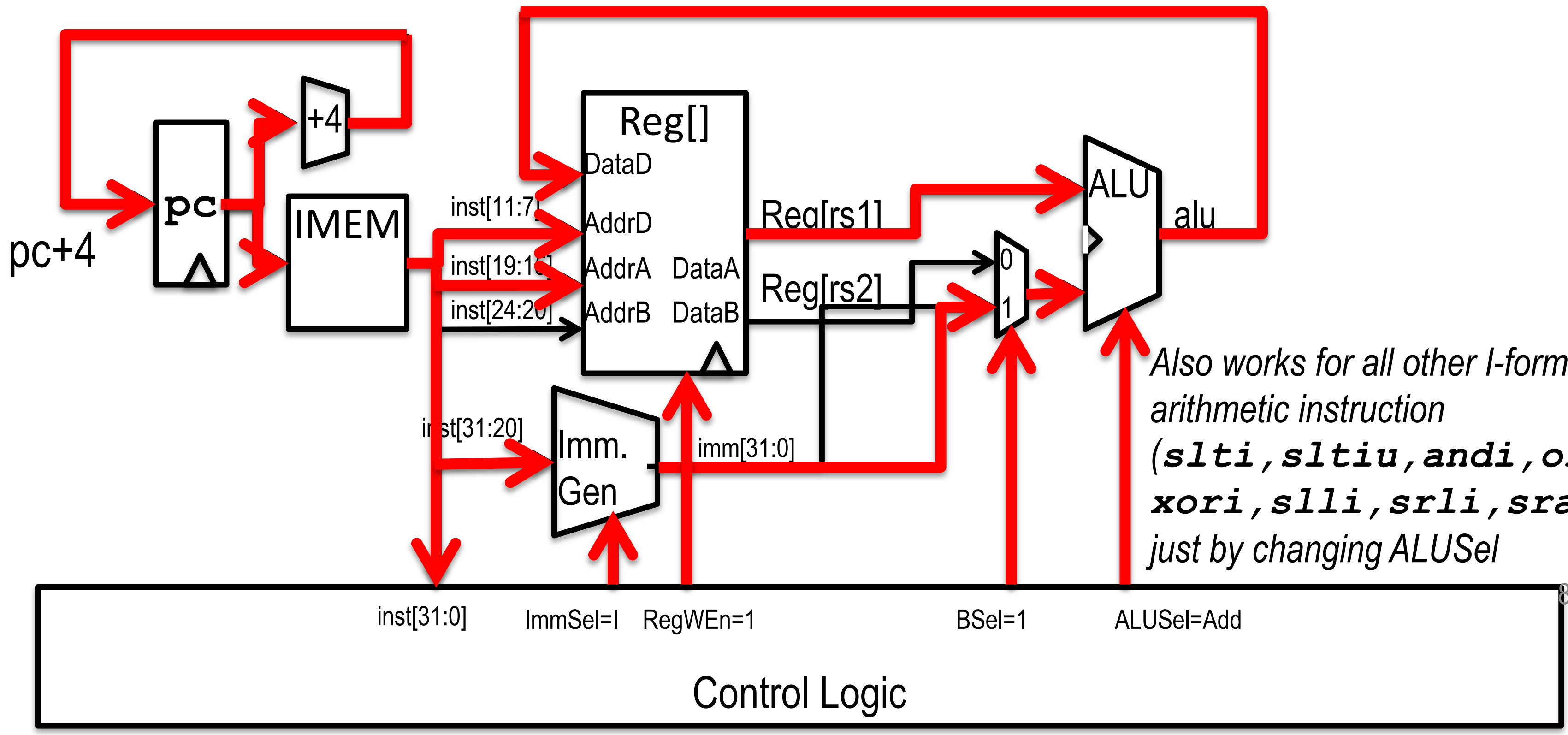
# Implementing R-Types



<u>(4) Perform operation</u>

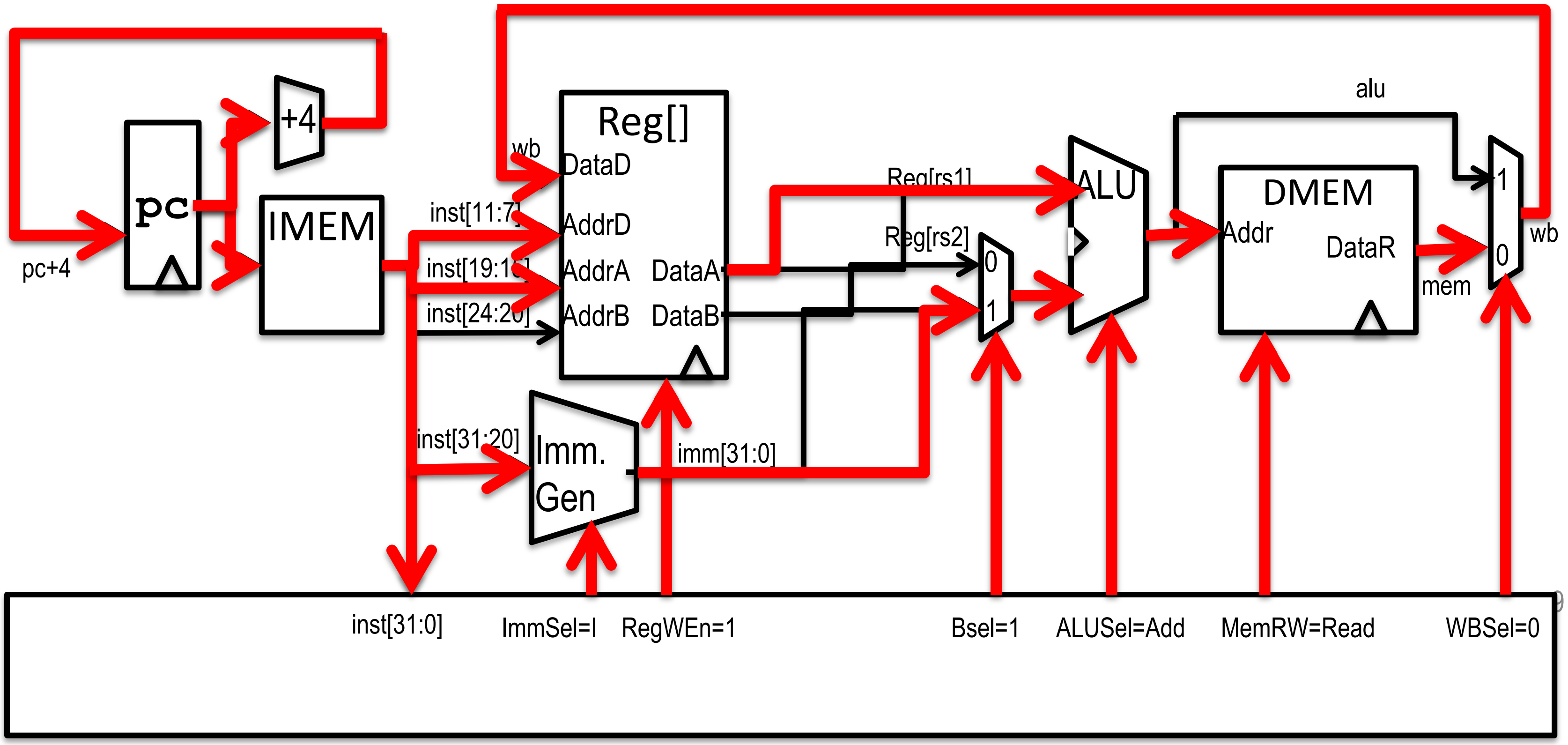
- New hardware: ALU (Arithmetic Logic Unit)
- Abstraction for adders, multipliers, dividers, etc.
- How do we know what operation to execute?
    - Our first control bit! ALUSel(ect)

# Adding `addi` to datapath

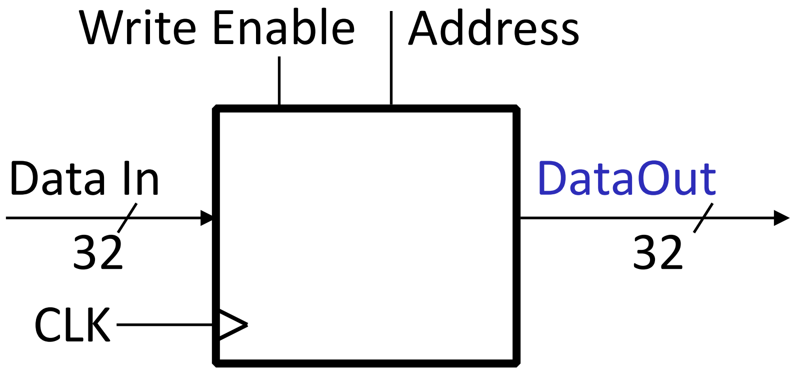


*Also works for all other I-format arithmetic instruction (**`slti,sltiu,andi,ori,xori,slli,srli,srai`**) just by changing ALUSel*

# Adding `lw` to datapath

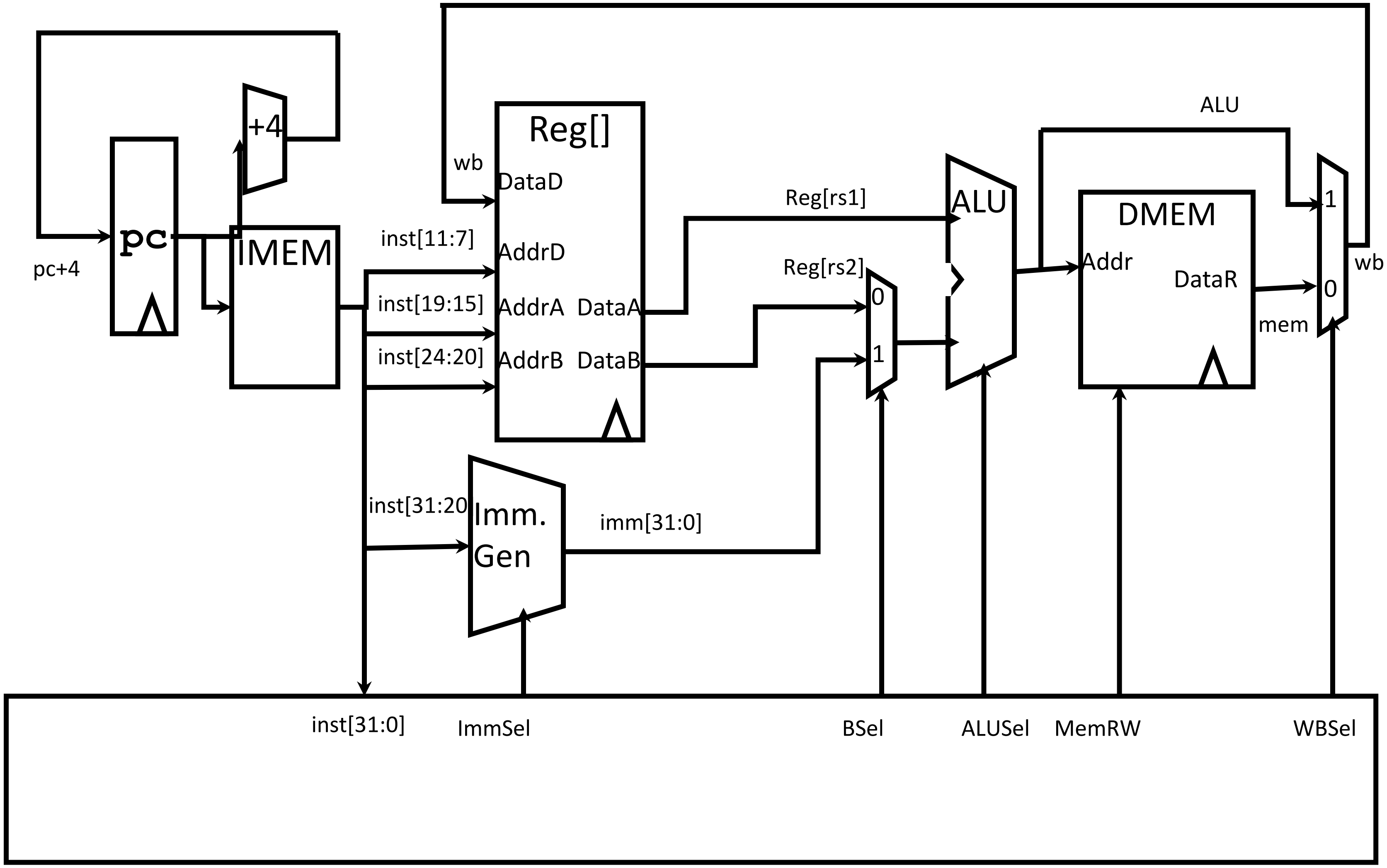


## Storage Element: Idealized Memory

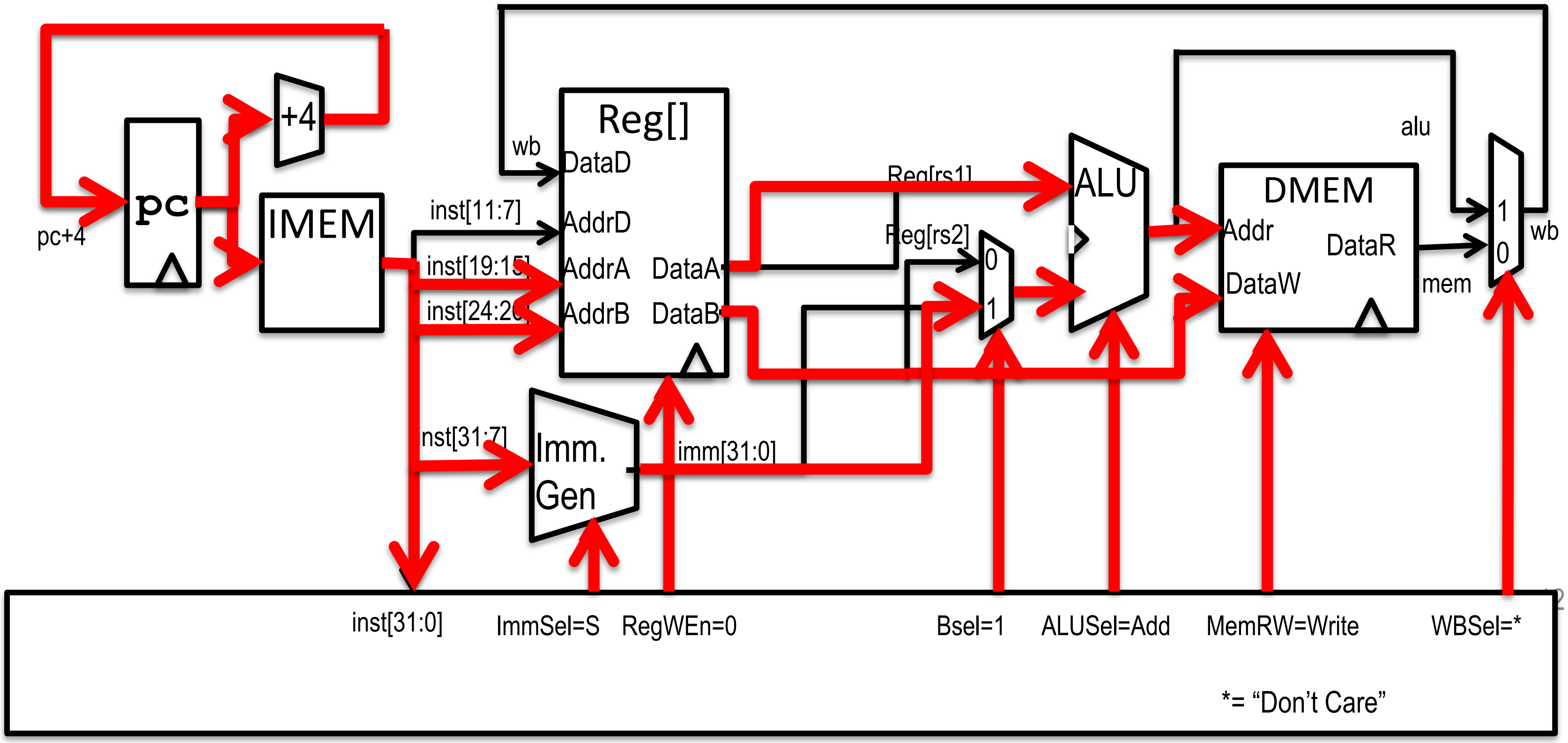


- Memory (idealized)
  - One input port: Data In
  - One output port: Data Out
- Memory access:
  - Read: Write Enable = 0, data at Address is placed on Data Out
  - Write: Write Enable = 1, Data In written to Address
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During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

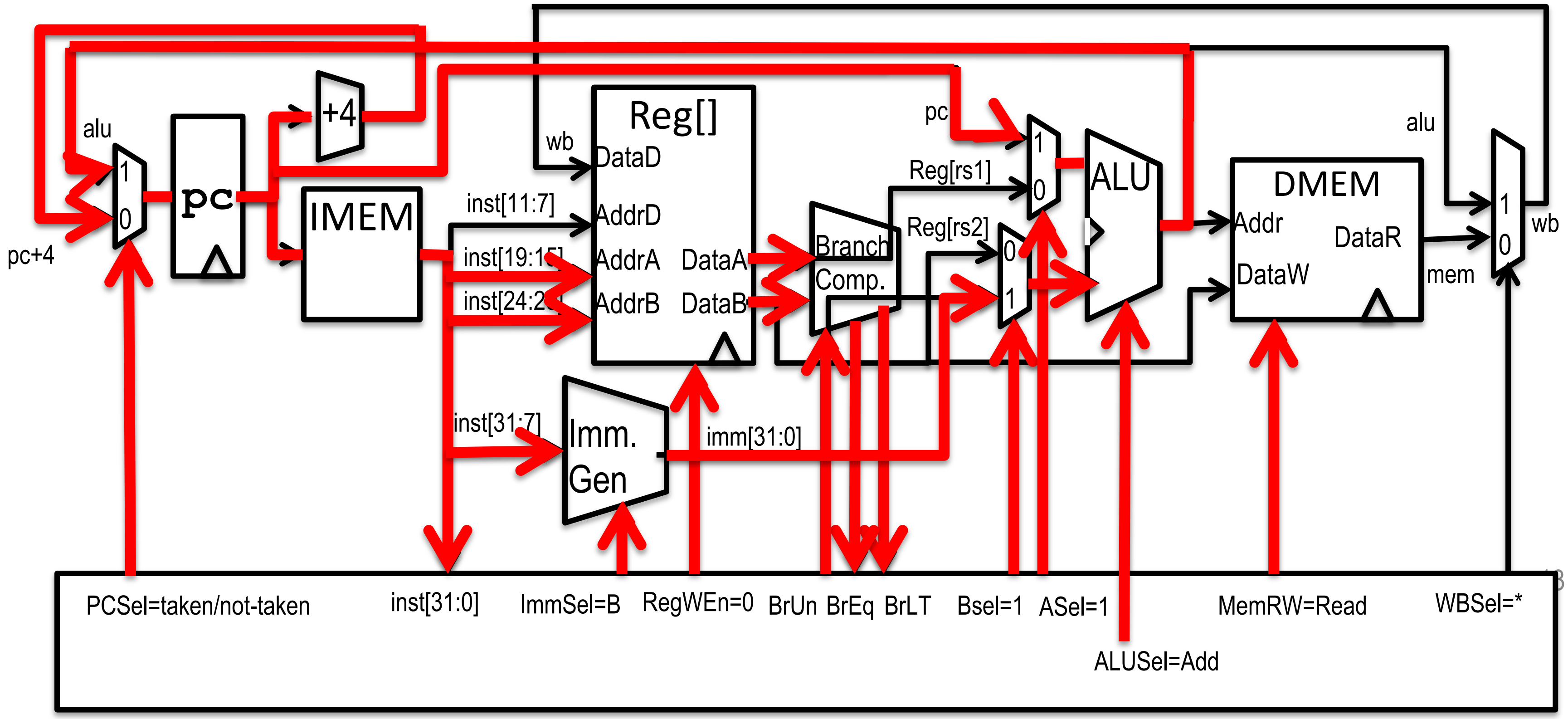
# Current Datapath



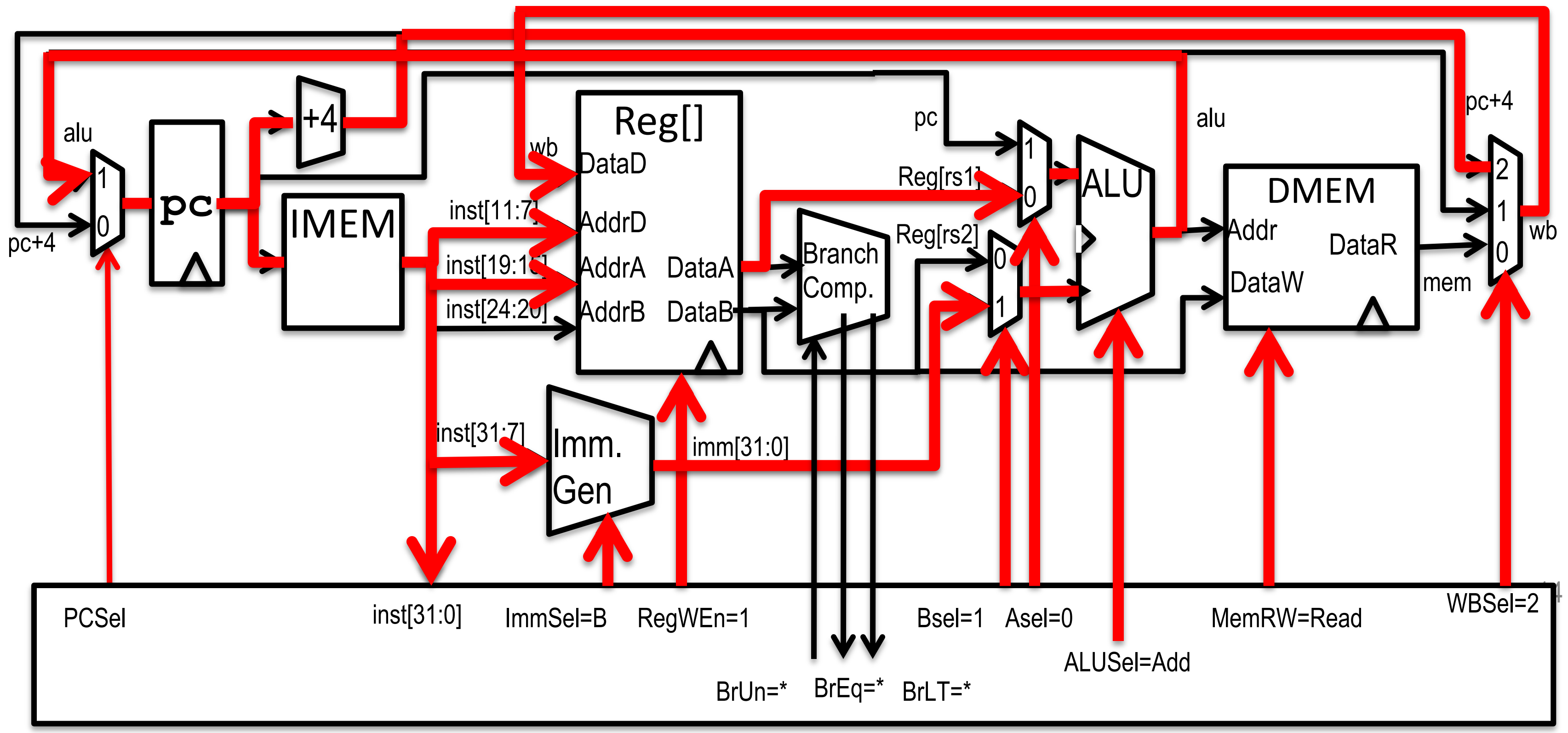
# Adding sw to datapath



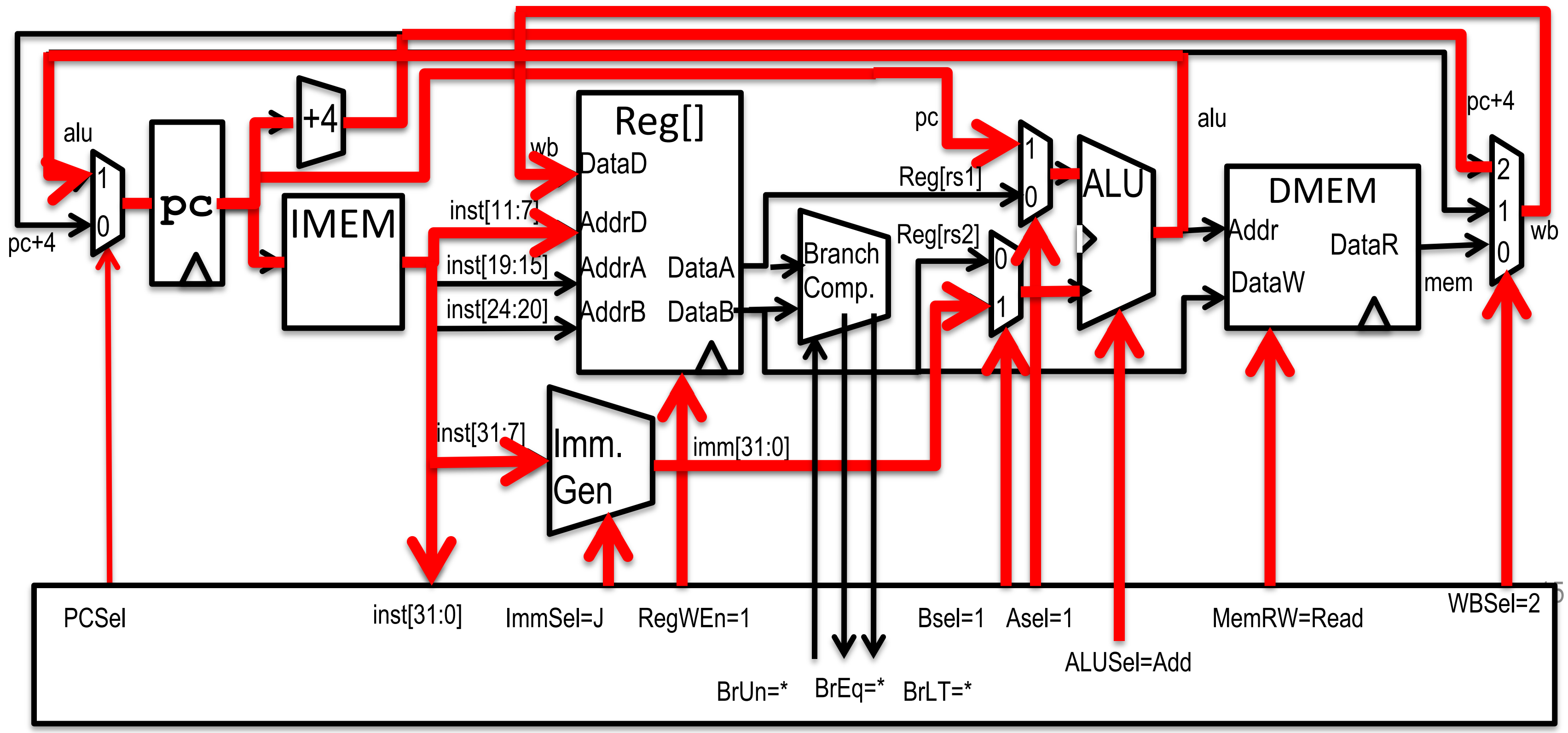
# Adding branches to datapath



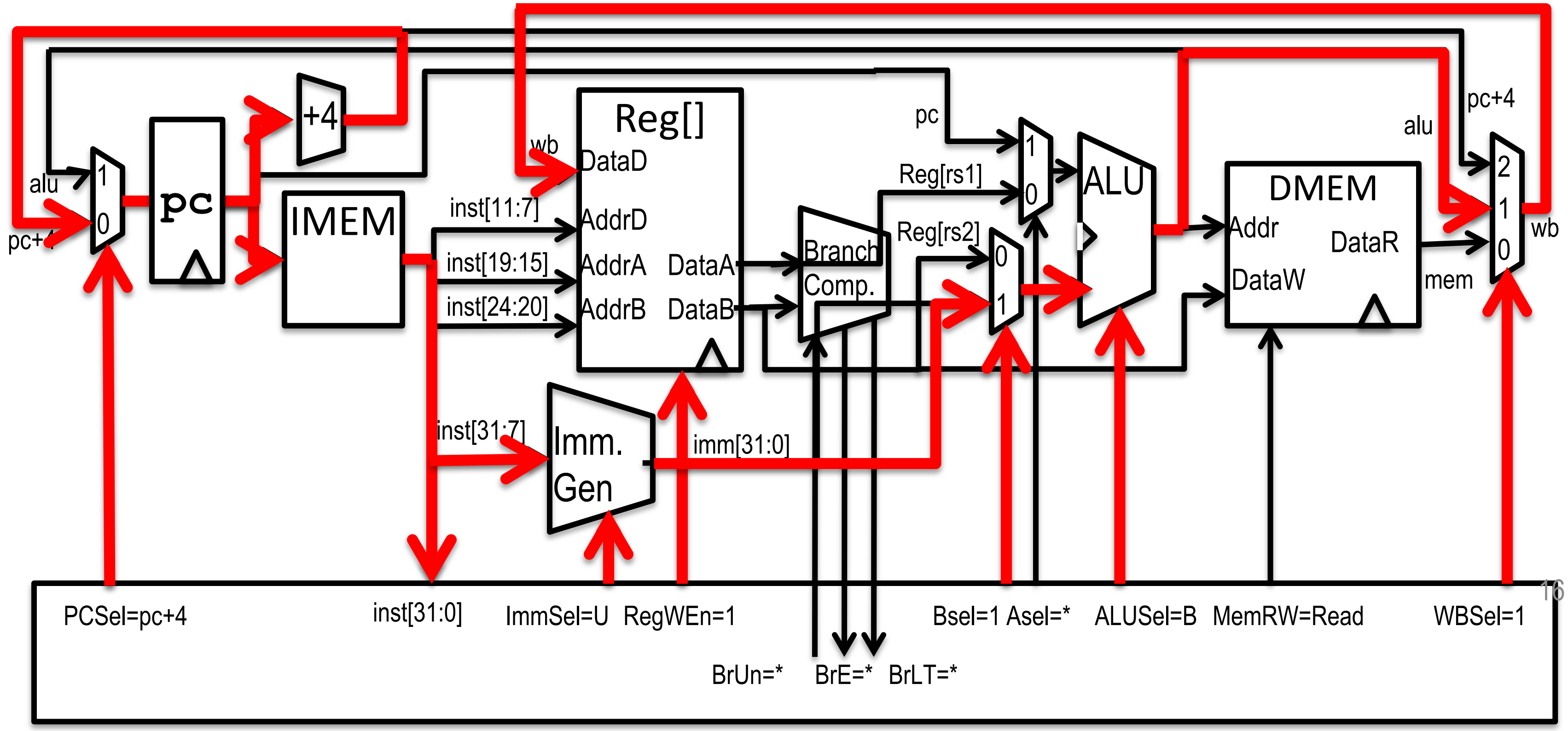
# Adding `jalr` to datapath



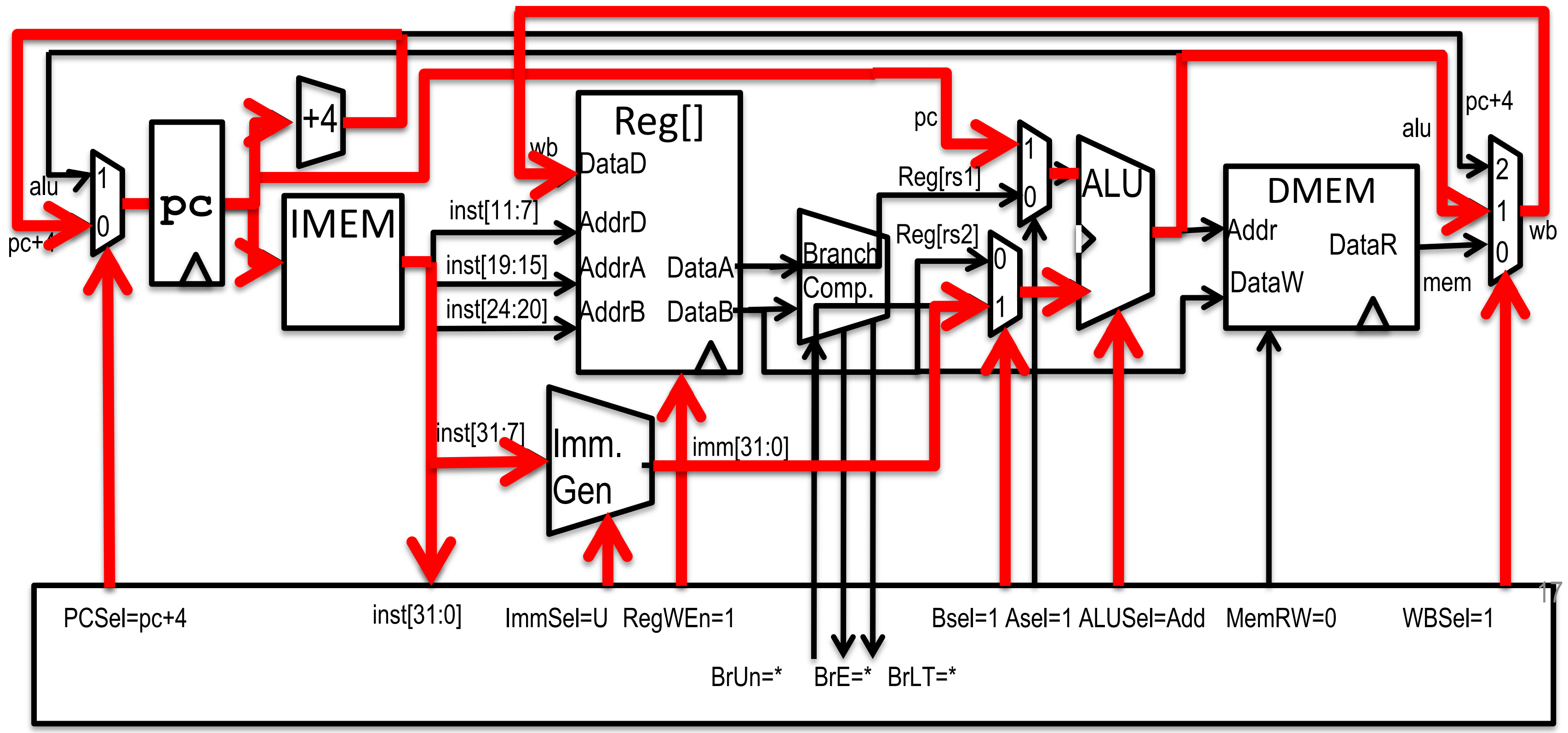
# Adding `jal` to datapath



# Implementing `lui`



# Implementing `auipc`


alu
pc+4
1
0
pc
+4
IMEM
wb
Reg[]
DataD
inst[11:7]
AddrD
inst[19:15]
AddrA
DataA
inst[24:20]
AddrB
DataB
inst[31:7]
Imm. Gen
imm[31:0]
Branch Comp.
pc
Reg[rs1]
Reg[rs2]
ALU
DMEM
Addr
DataR
DataW
alu
pc+4
2
mem
wb
PCSel=pc+4
inst[31:0]
ImmSel=U
RegWEn=1
BrUn=*
BrE=*
BrLT=*
Bsel=1
Asel=1
ALUSel=Add
MemRW=0
WBSel=1