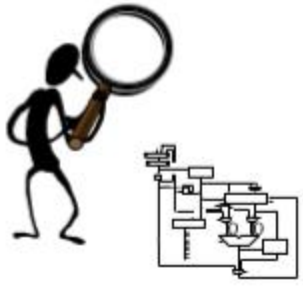
# CPU Design Tradeoffs

Maximum Performance: measured by the numbers of instructions executed per second



Minimum Cost : measured by the size of the circuit.

Best Performance/Price: measured by the ratio of MIPS to size. In power-sensitive applications MIPS/Watt is important too.

# Processor Performance

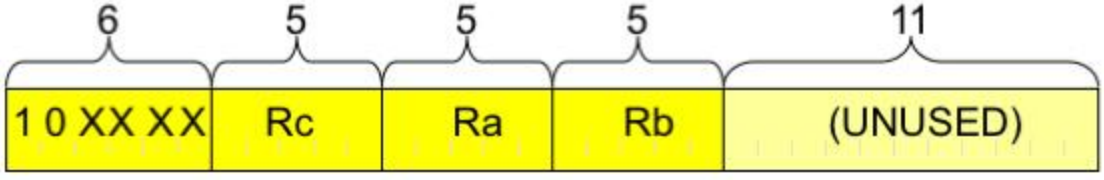
- "Iron Law" of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}} \qquad \text{Perf} = \frac{1}{\text{Time}}$$

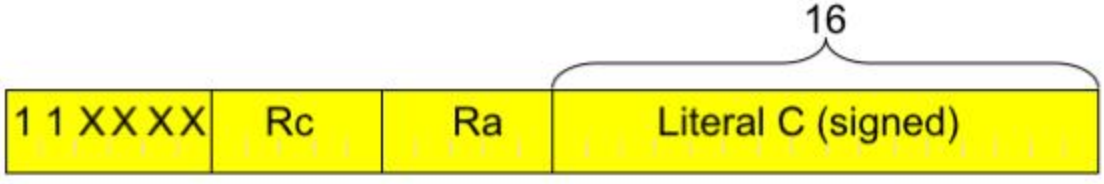
- Options to reduce execution time:
  - Executed instructions ↓ (work/instruction ↑)
  - Cycles per instruction (CPI) ↓
  - Cycle time ↓ (frequency ↑)

- Today: Simple, CPI=1 but low-frequency Beta
  - Later: Pipelining to increase frequency

# Reminder: Beta ISA




Operate class: `Reg[Rc] ← Reg[Ra] op Reg[Rb]`




Operate class: `Reg[Rc] ← Reg[Ra] op SXT(C)`

Opcodes, both formats:

| | | | | |
|---|---|---|---|---|
| ADD | SUB | MUL* | DIV* | *optional |
| CMPEQ | CMPLE | CMPLT | | |
| AND | OR | XOR | XNOR | |
| SHL | SHR | SRA | | |


```
LD:  Reg[Rc] ← Mem[Reg[Ra]+SXT(C)]
ST:  Mem[Reg[Ra]+SXT(C)] ← Reg[Rc]
LDR: Reg[Rc] ← Mem[PC + 4 + 4*SXT(C)]
BEQ: Reg[Rc] ← PC+4;  if Reg[Ra]=0  then  PC ← PC+4+4*SXT(C)
BNE: Reg[Rc] ← PC+4;  if Reg[Ra]≠0  then  PC ← PC+4+4*SXT(C)
JMP: Reg[Rc] ← PC+4;  PC ← Reg[Ra]
```

Instruction classes distinguished by OPCODE:

- OP
- OPC
- MEM
- Control Flow

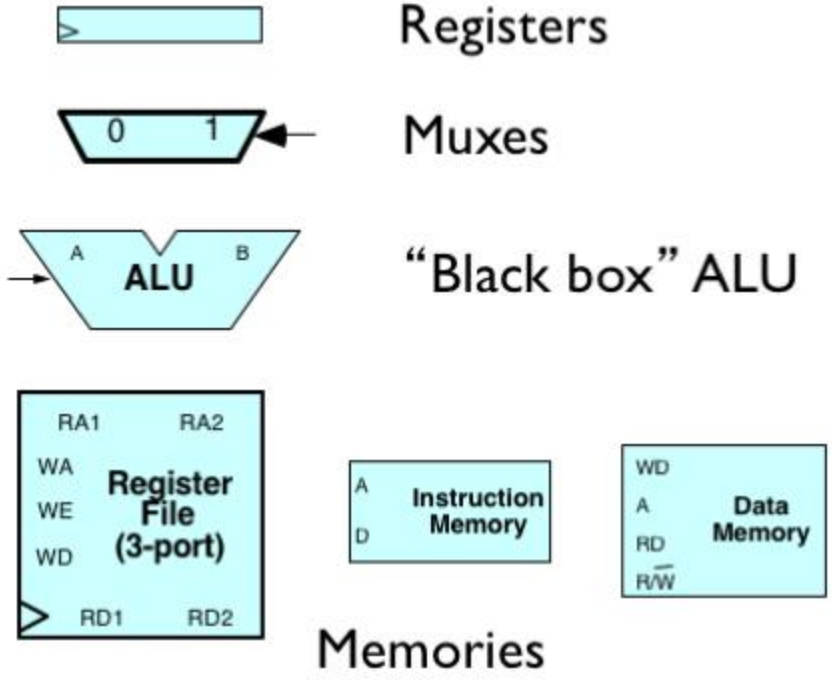
# Approach: Incremental Featurism

We'll implement datapaths for each instruction class individually, and merge them (using MUXes, etc)

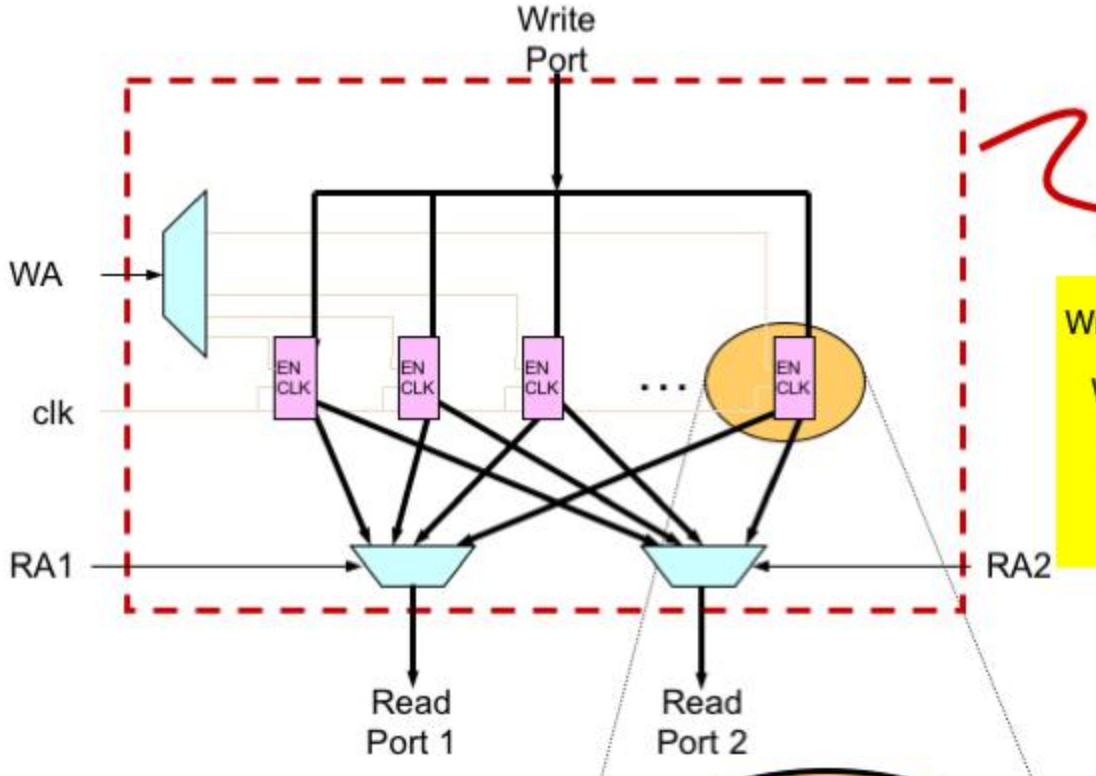
Steps:

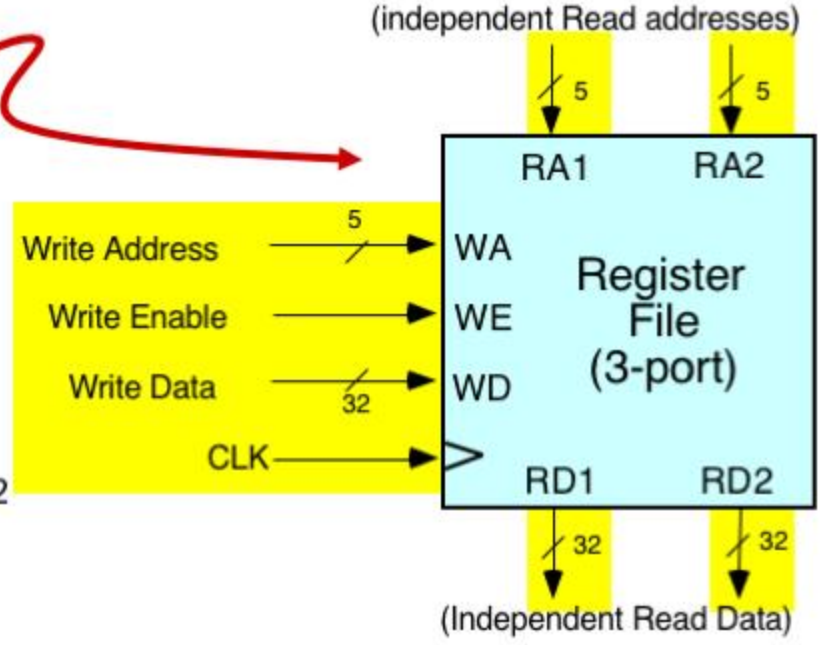
1. ALU instructions
2. Load & store instructions
3. Jump & branch instructions
4. Exceptions

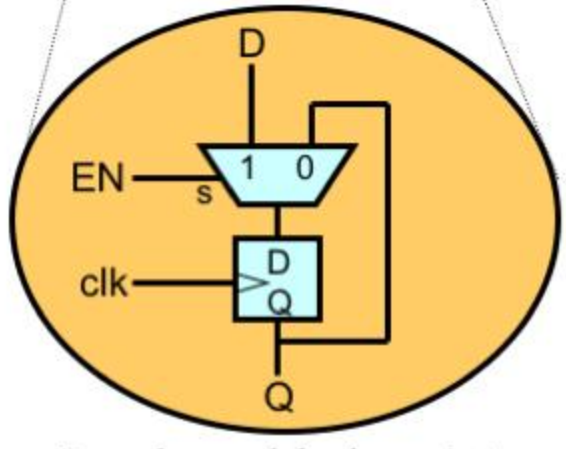
Component Repertoire:



# Multi-Ported Register File



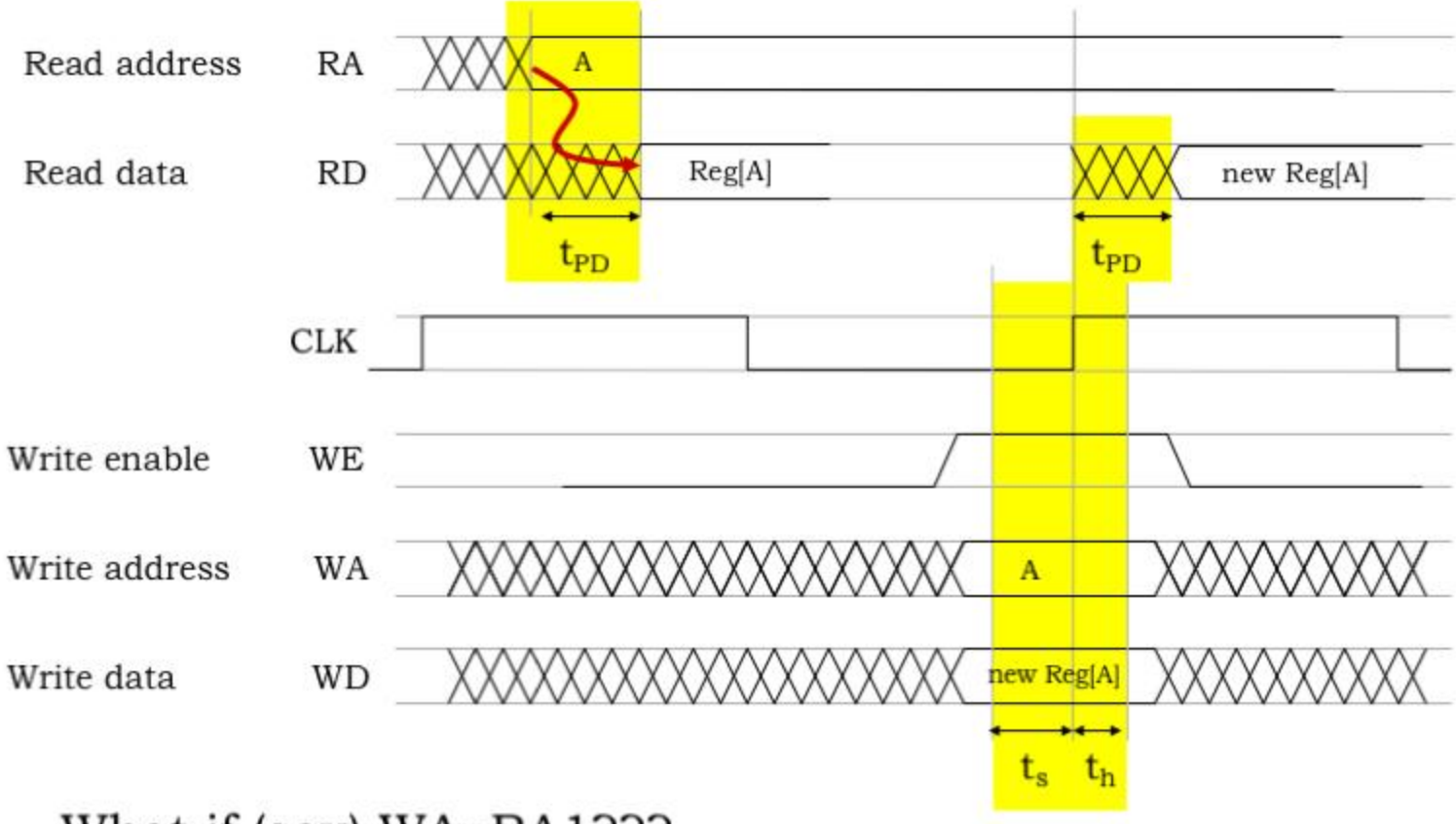

Load-enabled register

2 combinational READ ports*,
1 clocked WRITE port

*internal logic ensures Reg[31] reads as 0

# Register File Timing

2 combinational READ ports, 1 clocked WRITE port

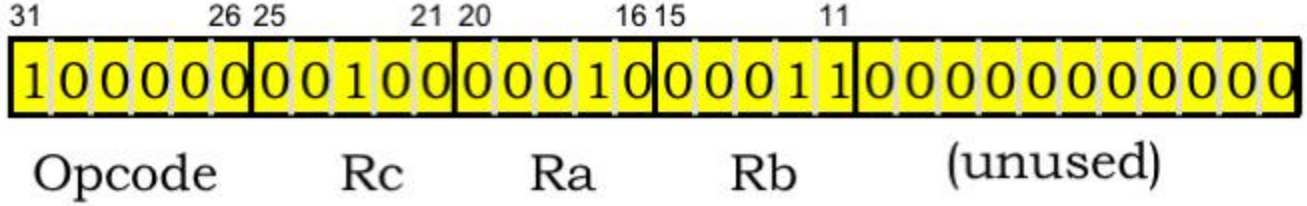



What if (say) WA=RA1???

RD1 reads "old" value of Reg[RA1] until next clock edge!

# ALU Instructions

32-bit (4-byte) ADD instruction:



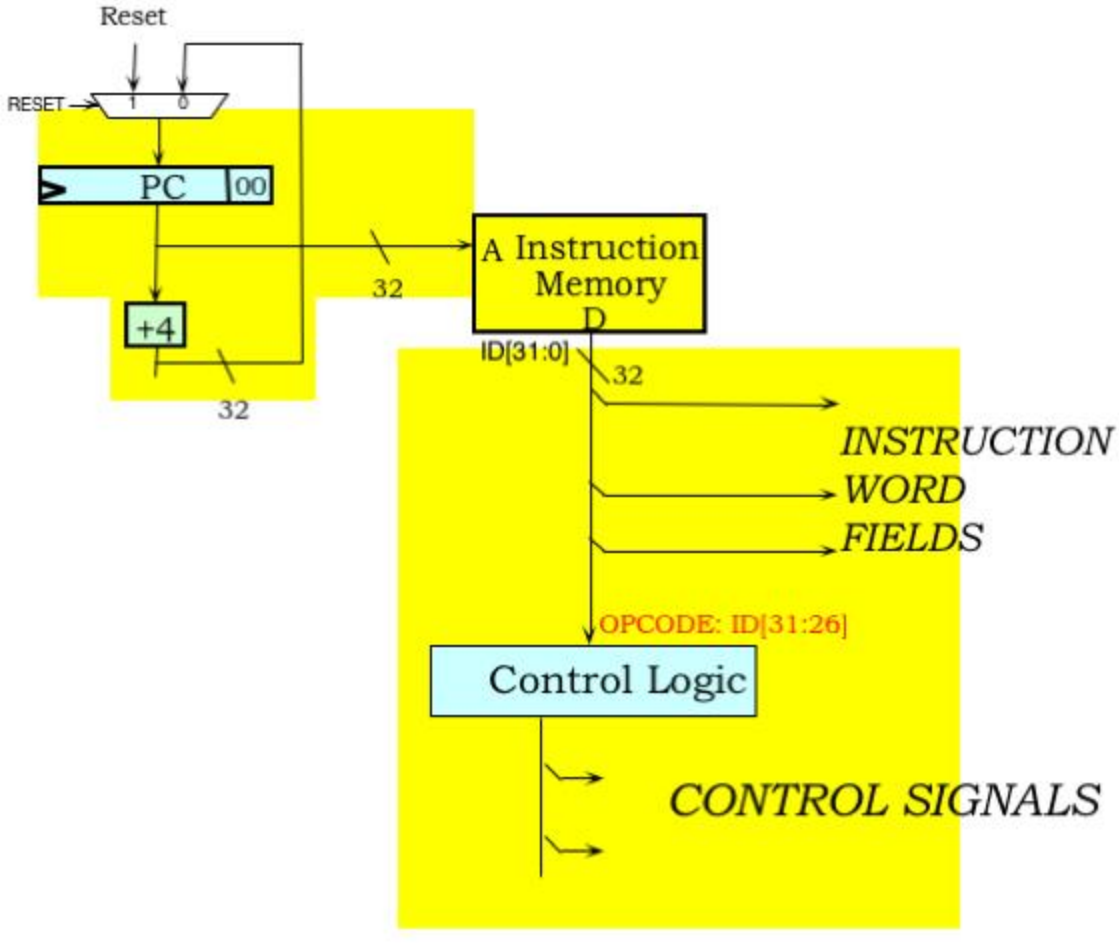

*Means, to Beta,* `Reg[R4] ← Reg[R2] + Reg[R3]`

Need hardware to:

- FETCH (read) 32-bit instruction for the current cycle
- DECODE instruction: ADD, SUB, XOR, etc
- READ operands (Ra, Rb) from Register File
- EXECUTE operation
- WRITE-BACK result into Register File (Rc)

# Instruction Fetch/Decode

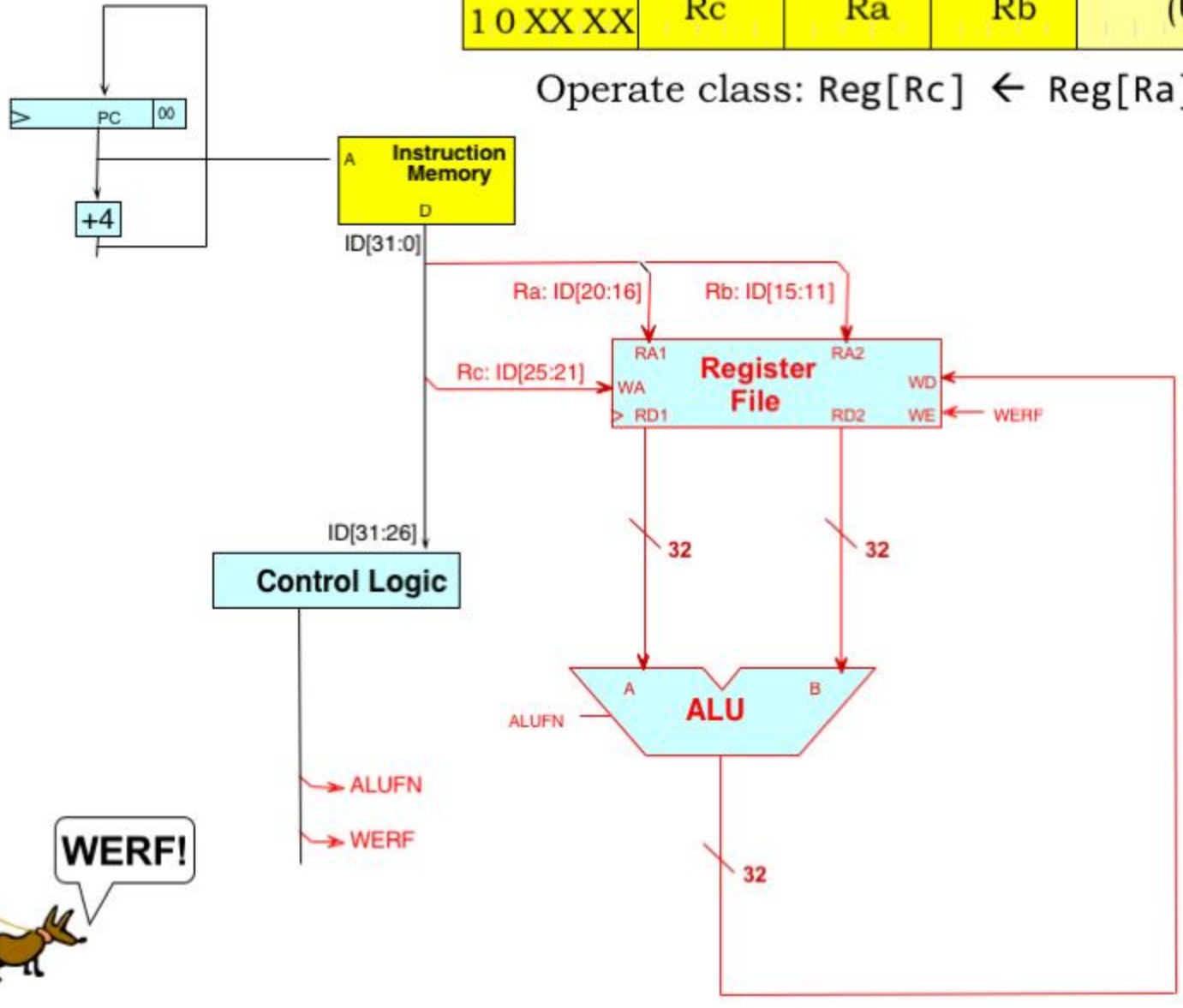
Use a counter to FETCH the next instruction: PROGRAM COUNTER (PC)



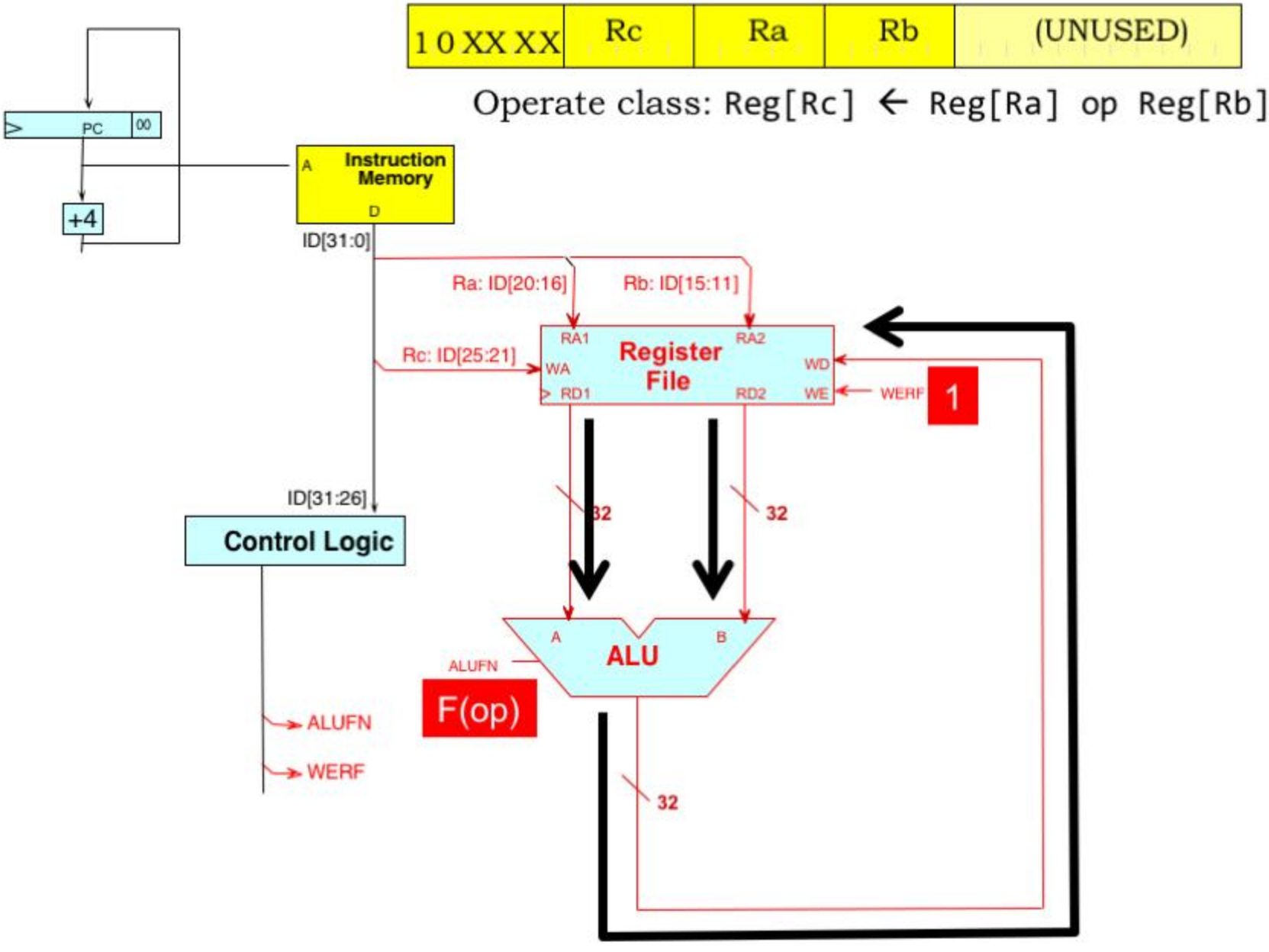

- Use PC as memory address
- Add 4 to PC, load new value at end of cycle
- Fetch instruction from memory
  - Use some instruction fields directly (register numbers, 16-bit constant)
  - Use bits [31:26] to generate control signals

# ALU Op Datapath

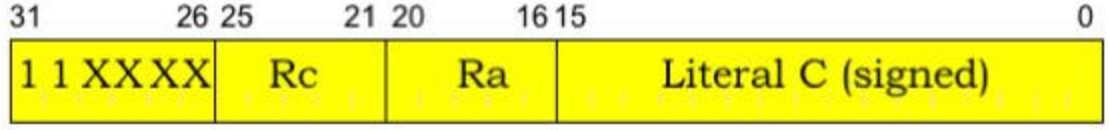

Operate class: Reg[Rc] ← Reg[Ra] op Reg[Rb]



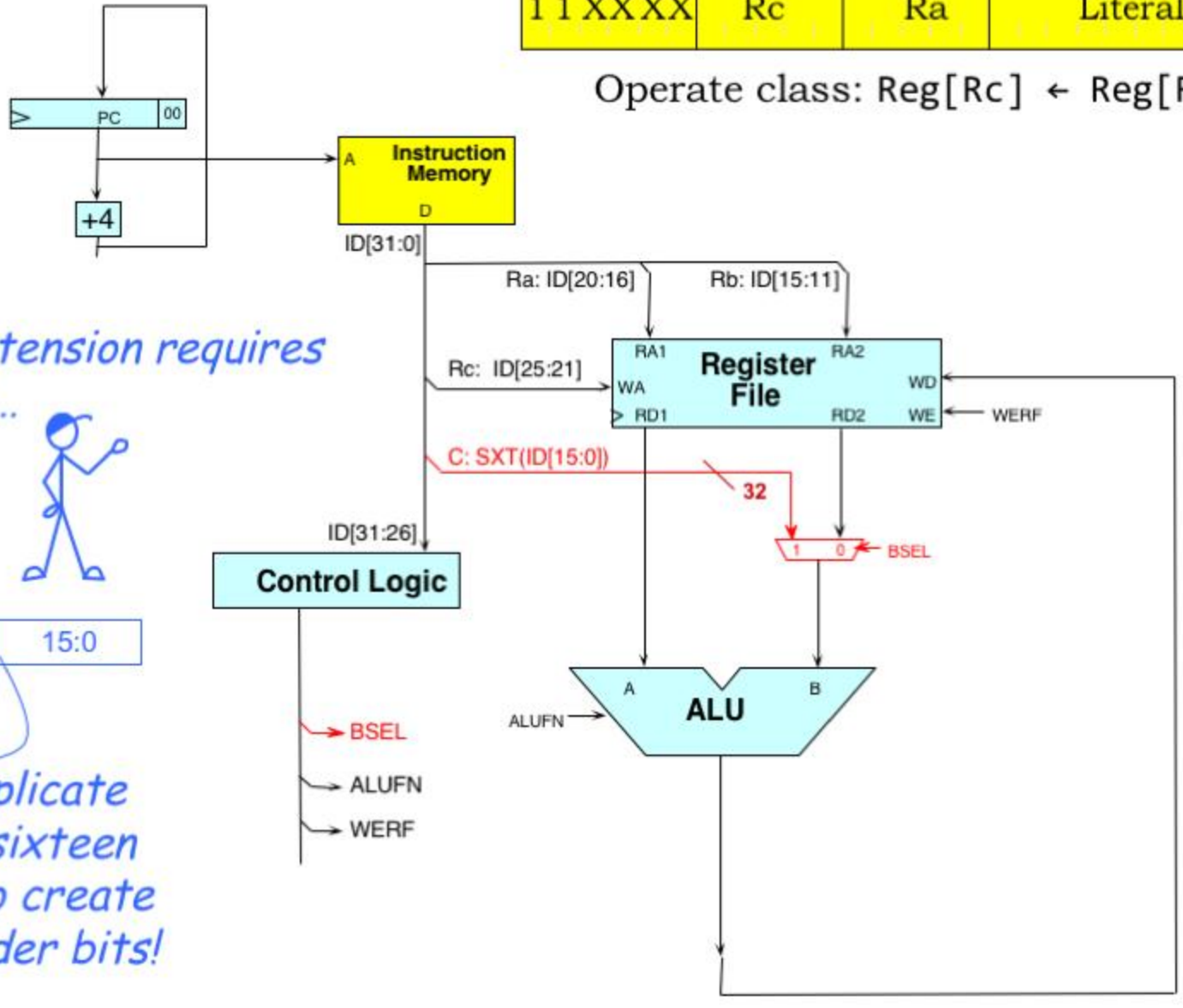
# ALU Op Datapath



# ALU Operations (with constant)




Operate class: Reg[Rc] ← Reg[Ra] op SXT(C)




*Sign-extension requires no logic...*

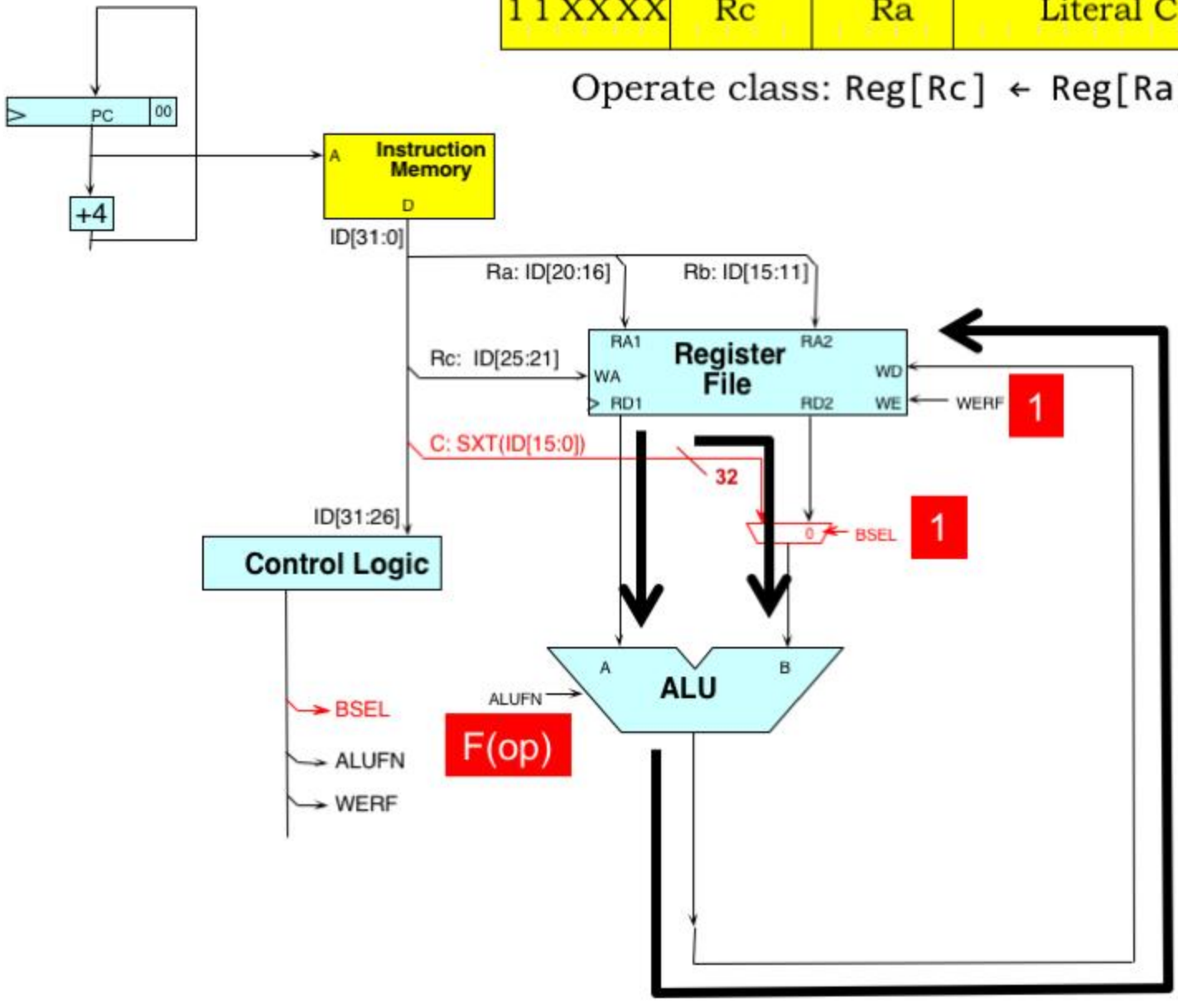



*Just replicate ID[15] sixteen times to create high-order bits!*

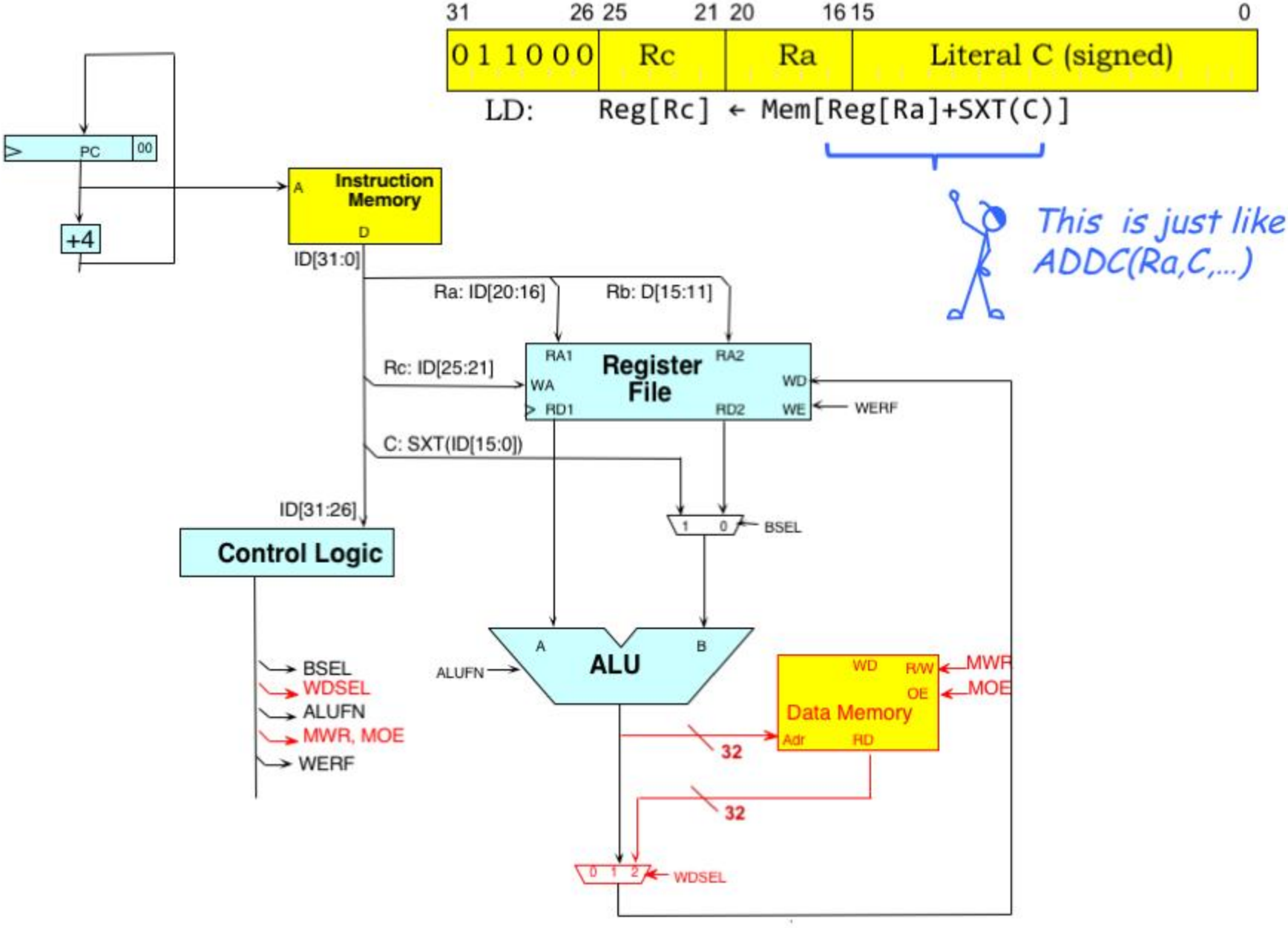
# ALU Operations (with constant)

| 1 1 XXXX | Rc | Ra | Literal C (signed) |
|---|---|---|---|

Operate class: Reg[Rc] ← Reg[Ra] op SXT(C)



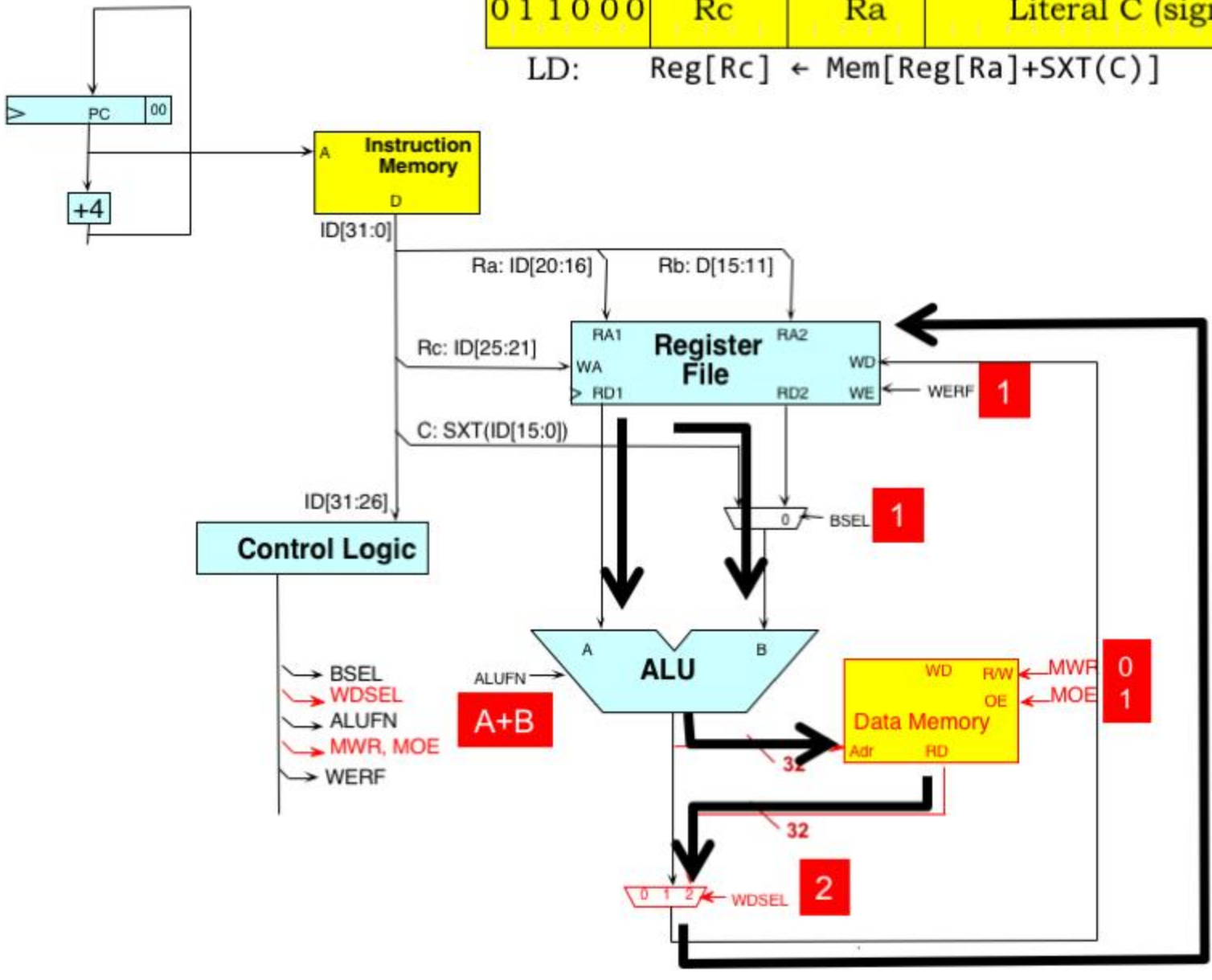
# Load Instruction



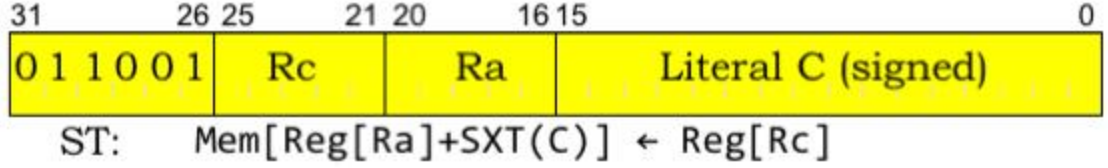
# Load Instruction

| 0 1 1 0 0 0 | Rc | Ra | Literal C (signed) |
|---|---|---|---|

LD: Reg[Rc] ← Mem[Reg[Ra]+SXT(C)]



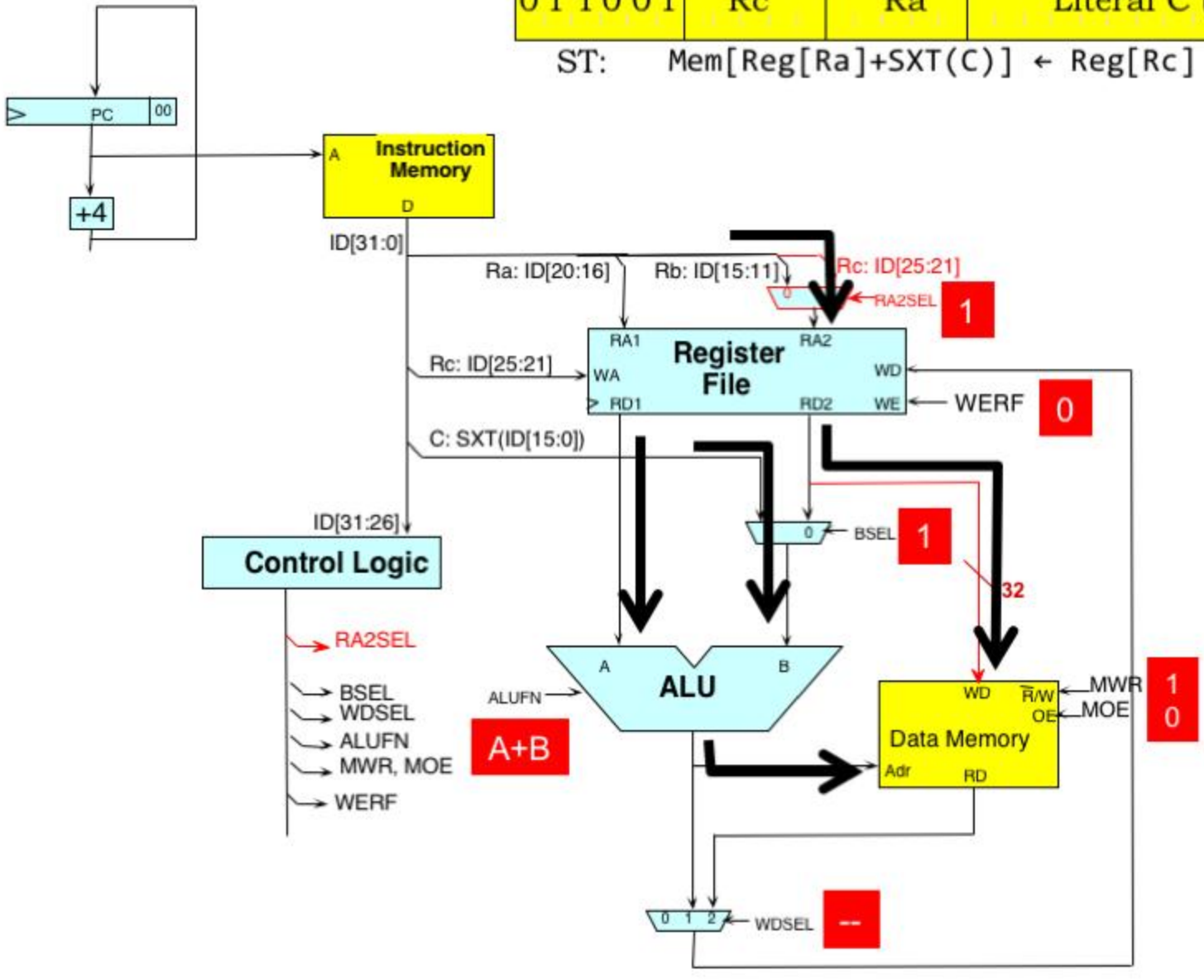
# Store Instruction



# Store Instruction

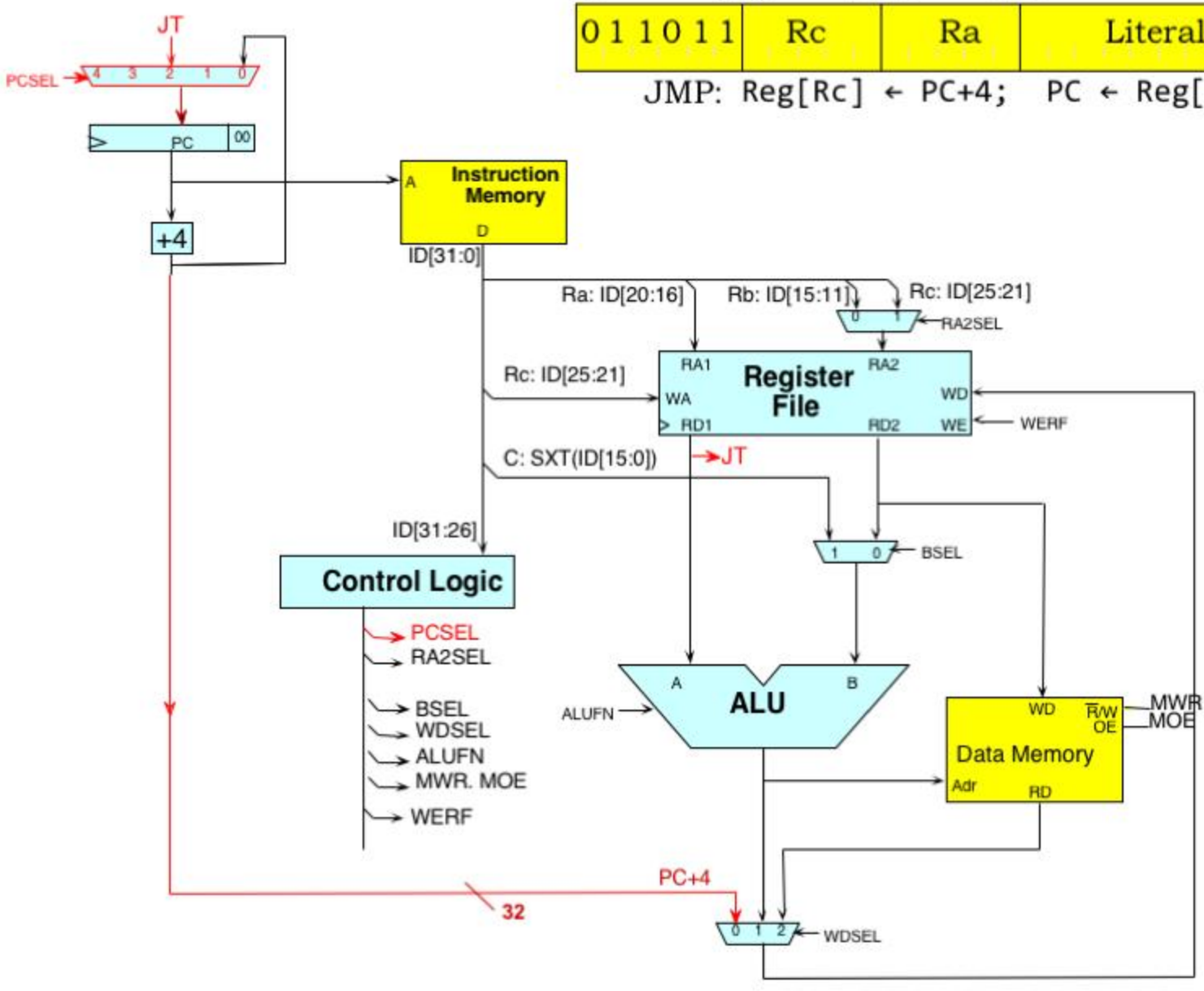
| 011001 | Rc | Ra | Literal C (signed) |
|---|---|---|---|

ST: Mem[Reg[Ra]+SXT(C)] ← Reg[Rc]

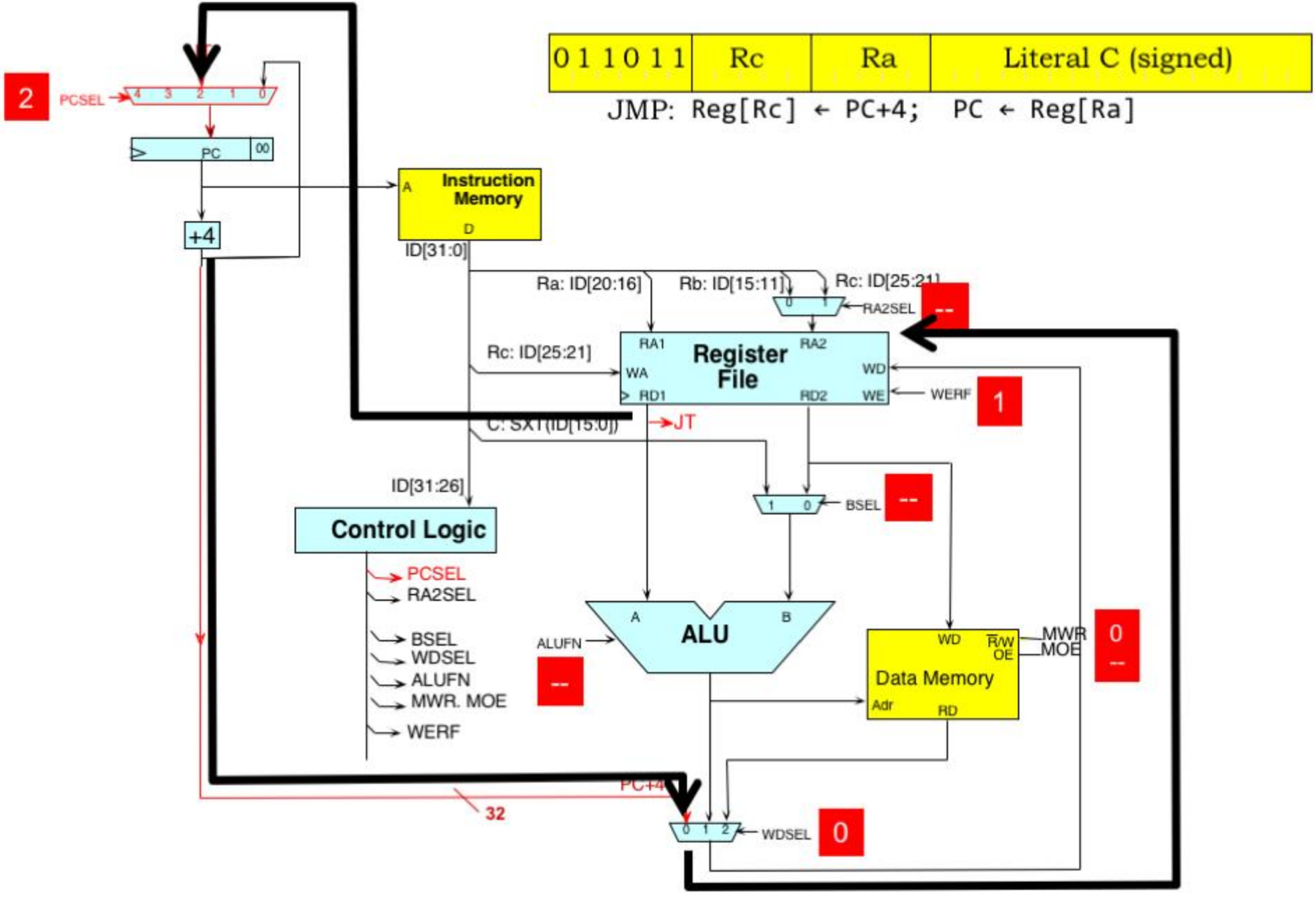


# JMP Instruction

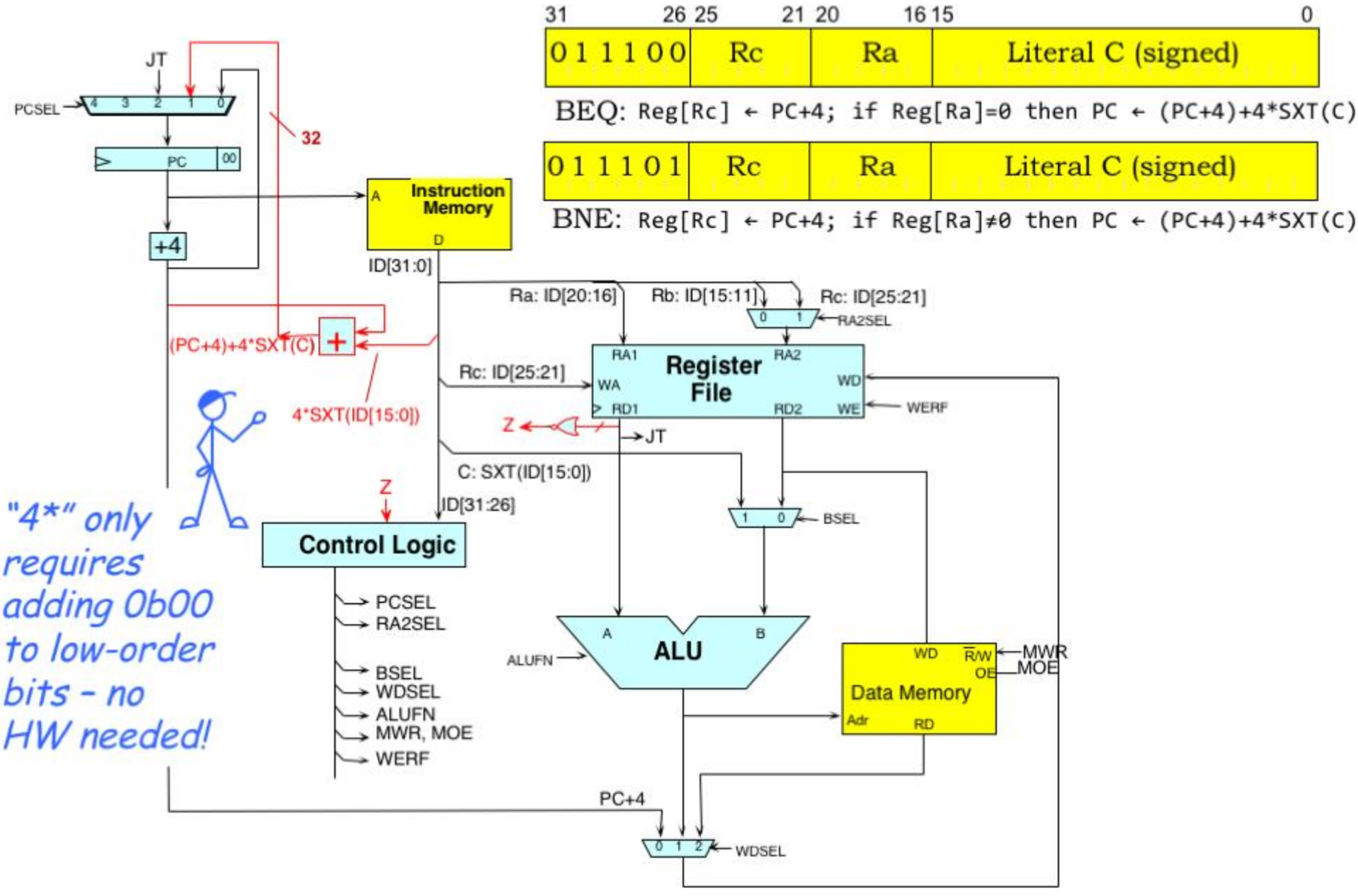

JMP: Reg[Rc] ← PC+4; PC ← Reg[Ra]



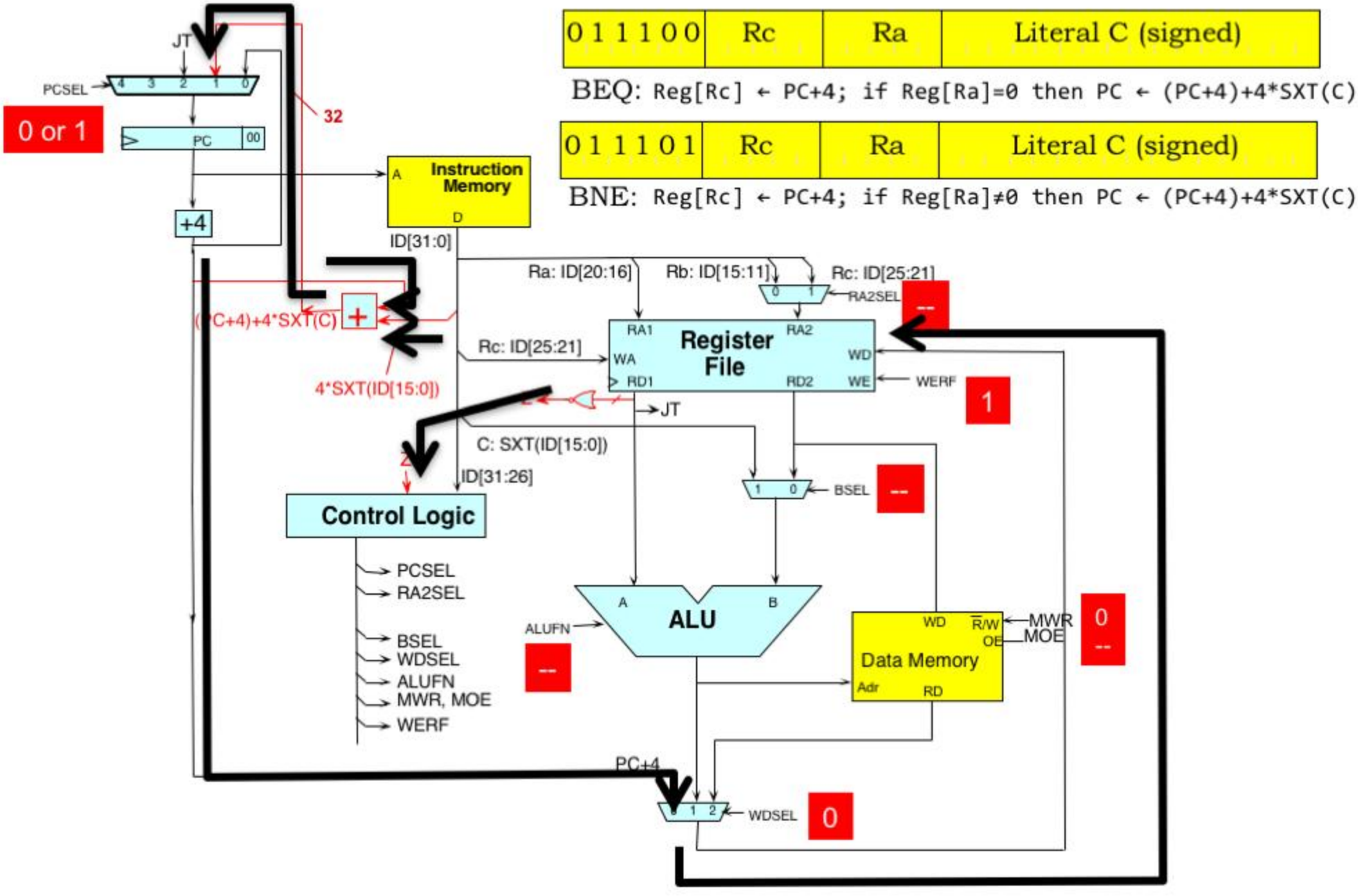
# JMP Instruction



# BEQ/BNE Instructions



# BEQ/BNE Instructions



# Load Relative Instruction


LDR: Reg[Rc] ← Mem[PC + 4 + 4*SXT(C)]

What's Load Relative good for anyway??? I thought

- Code is "PURE", i.e. READ-ONLY; and stored in a "PROGRAM" region of memory;
- Data is READ-WRITE, and stored either
  - On the STACK (local); or
  - In some GLOBAL VARIABLE region; or
  - In a global storage HEAP.

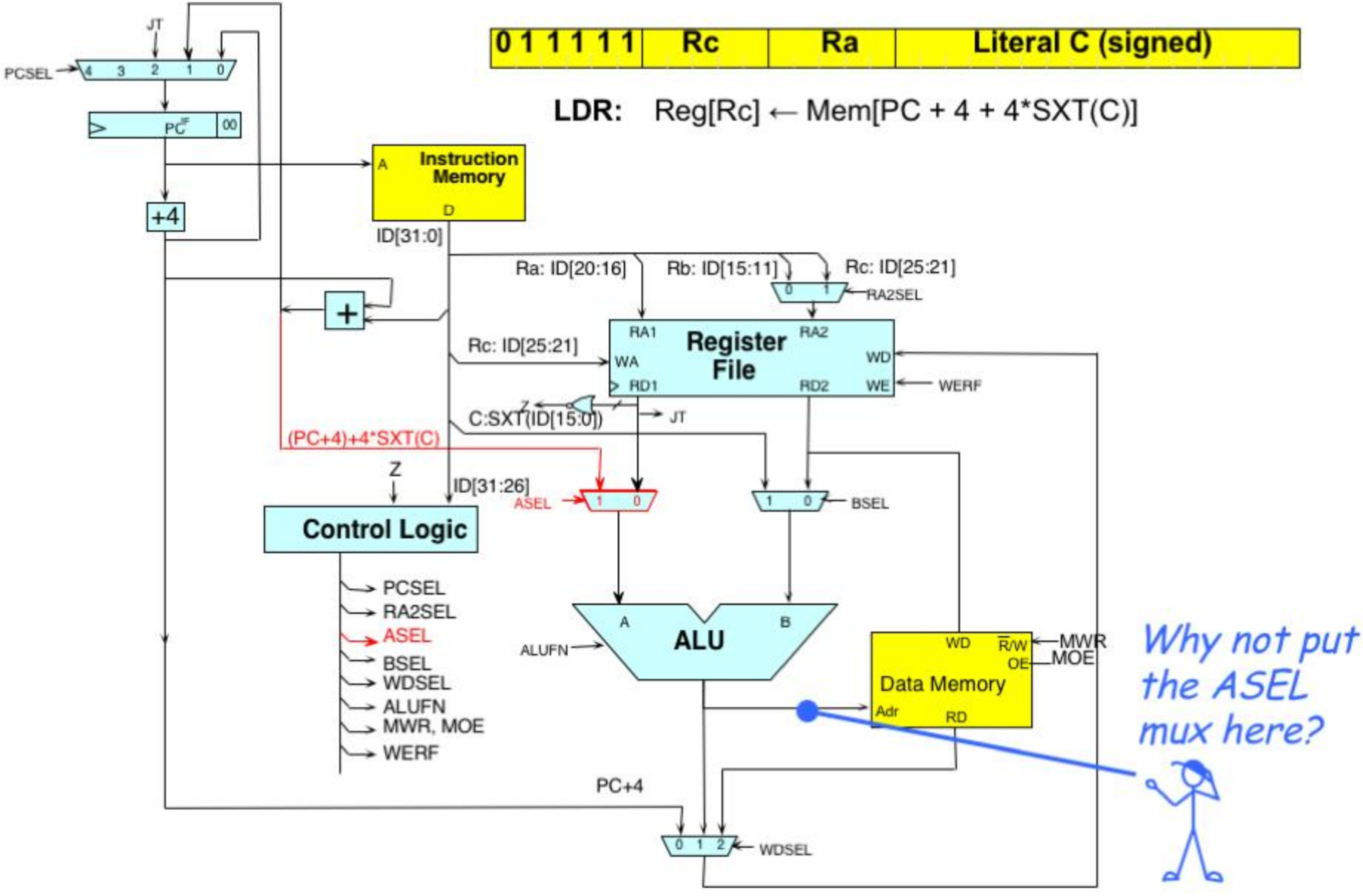
So why have an instruction designed to load data that's "near" the instruction???

Addresses & other large constants

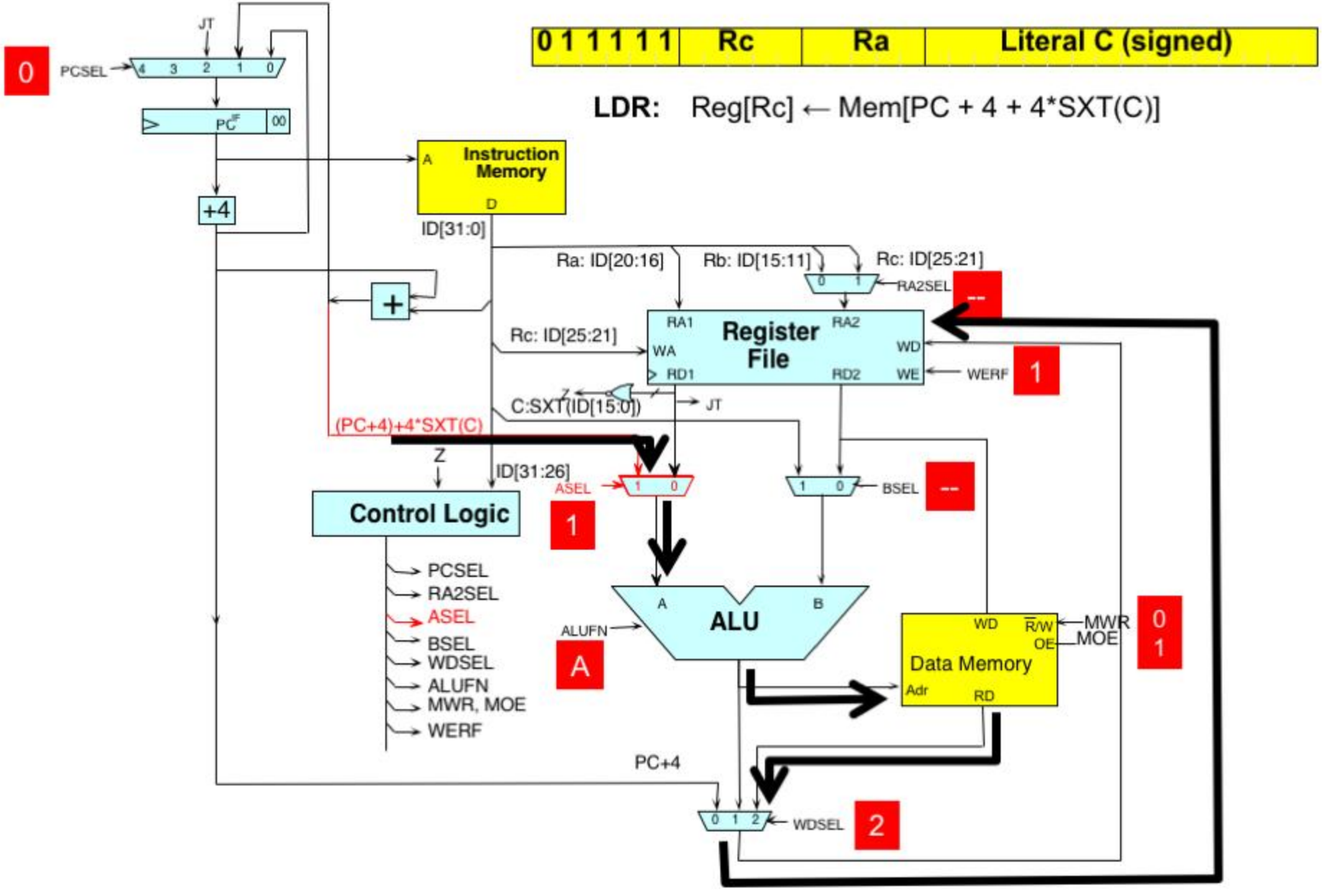
```
C:      X = X * 123456;

BETA:
        LD(X, r0)
        LDR(c1, r1)
        MUL(r0, r1, r0)
        ST(r0, X)
        ...
c1:     LONG(123456)
```

# LDR Instruction



# LDR Instruction



# Exceptions

- What if something bad happens?
  - Execution of an illegal opcode
  - Reference to non-existent memory
  - Divide by zero
- Or maybe just something unanticipated
  - User hits a key
  - A packet comes in via the network
- Exceptions let us handle these cases in software:
  - Treat each case as an (implicit) procedure call
  - Procedure handles problem, returns to interrupted program
  - Transparent to interrupted program!
  - Important added capability: handlers for certain errors (illegal opcodes), can extend ISA using software

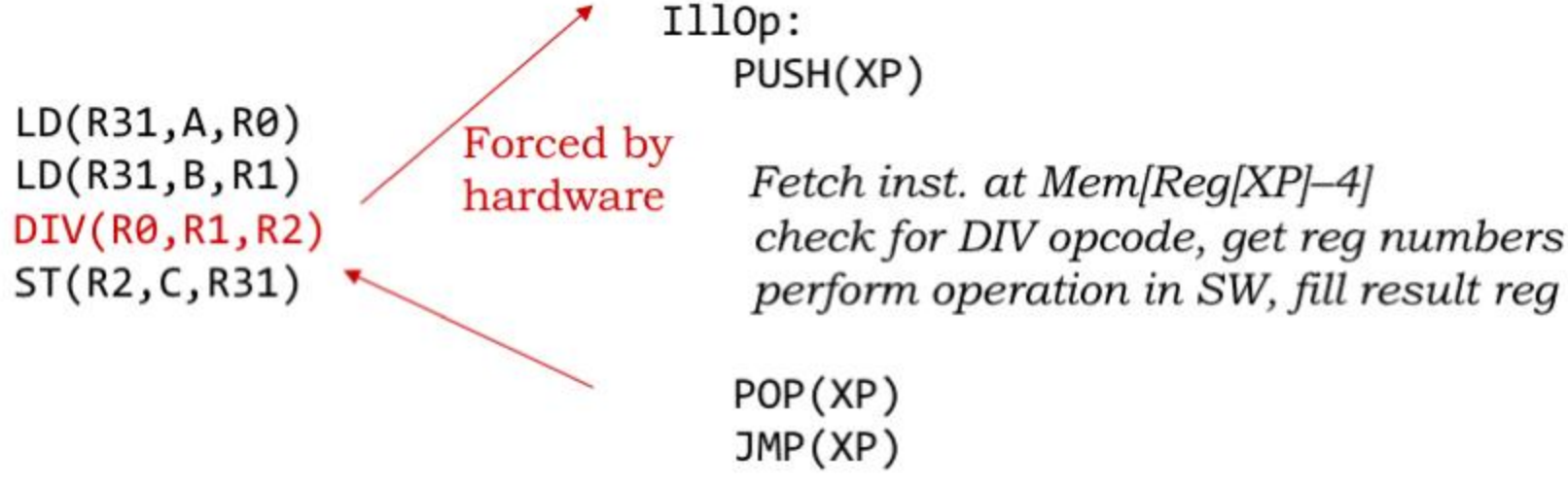
# Exception Processing

- Plan:
  - Interrupt running program
  - Invoke exception handler (like a procedure call)
  - Return to continue execution

- Exception and interrupt terms often used interchangeably, with minor distinctions:
  - Exceptions usually refer to synchronous events, generated by program (e.g., illegal instruction, divide-by-0, illegal address)
  - Interrupts usually refer to asynchronous events, generated by I/O devices (e.g., keystroke, packet received, disk transfer complete)

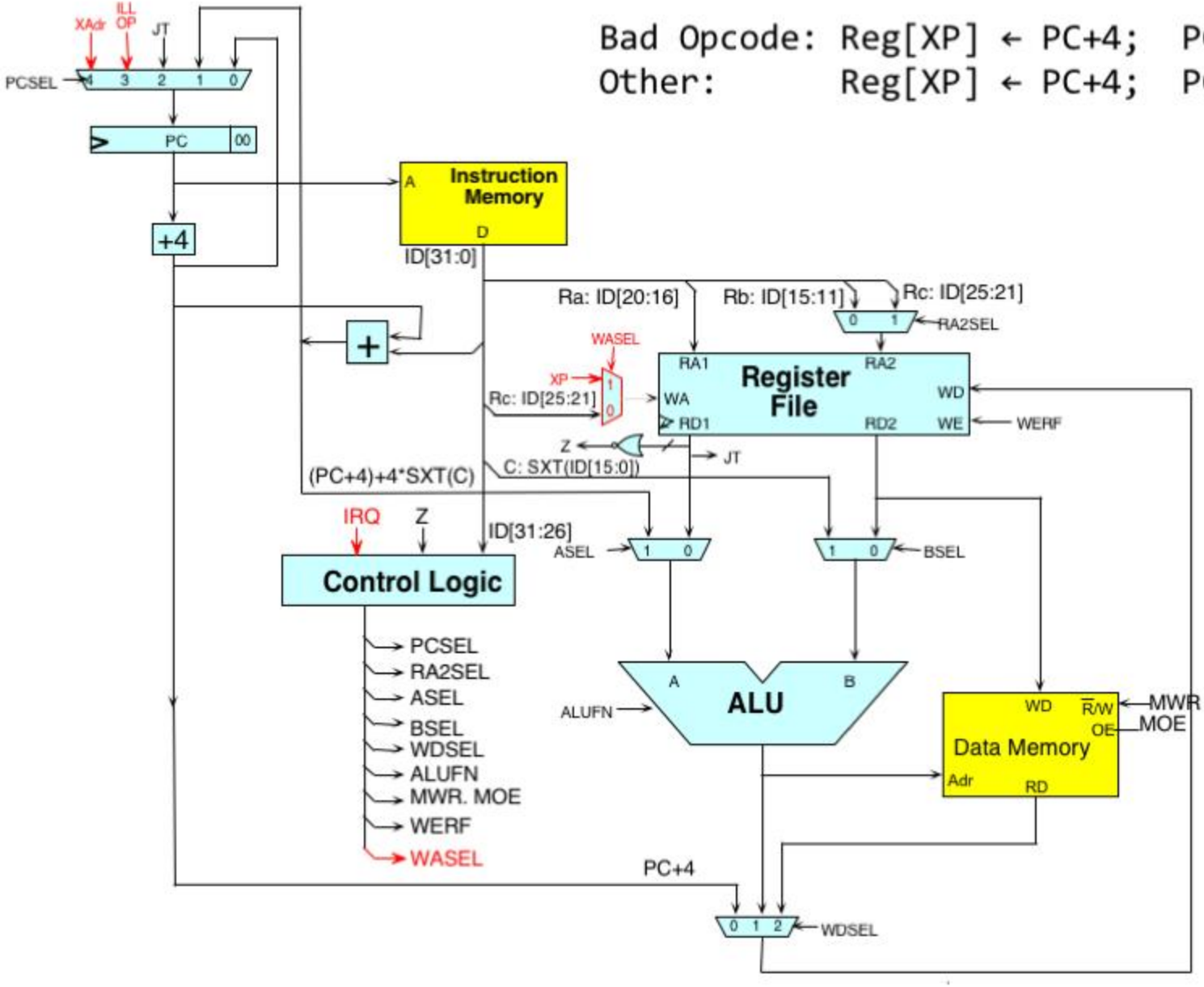
# Exception Implementation

- Instead of executing instruction, fake a procedure call
  - Save current PC+4 (as branches do)
  - Load PC with exception vector: 0x4 for synchronous events, 0x8 for asynchronous events
- We save PC+4 in register R30 (which we call XP)
  - … and prohibit programs from using XP (why?)
- Example: DIV unimplemented



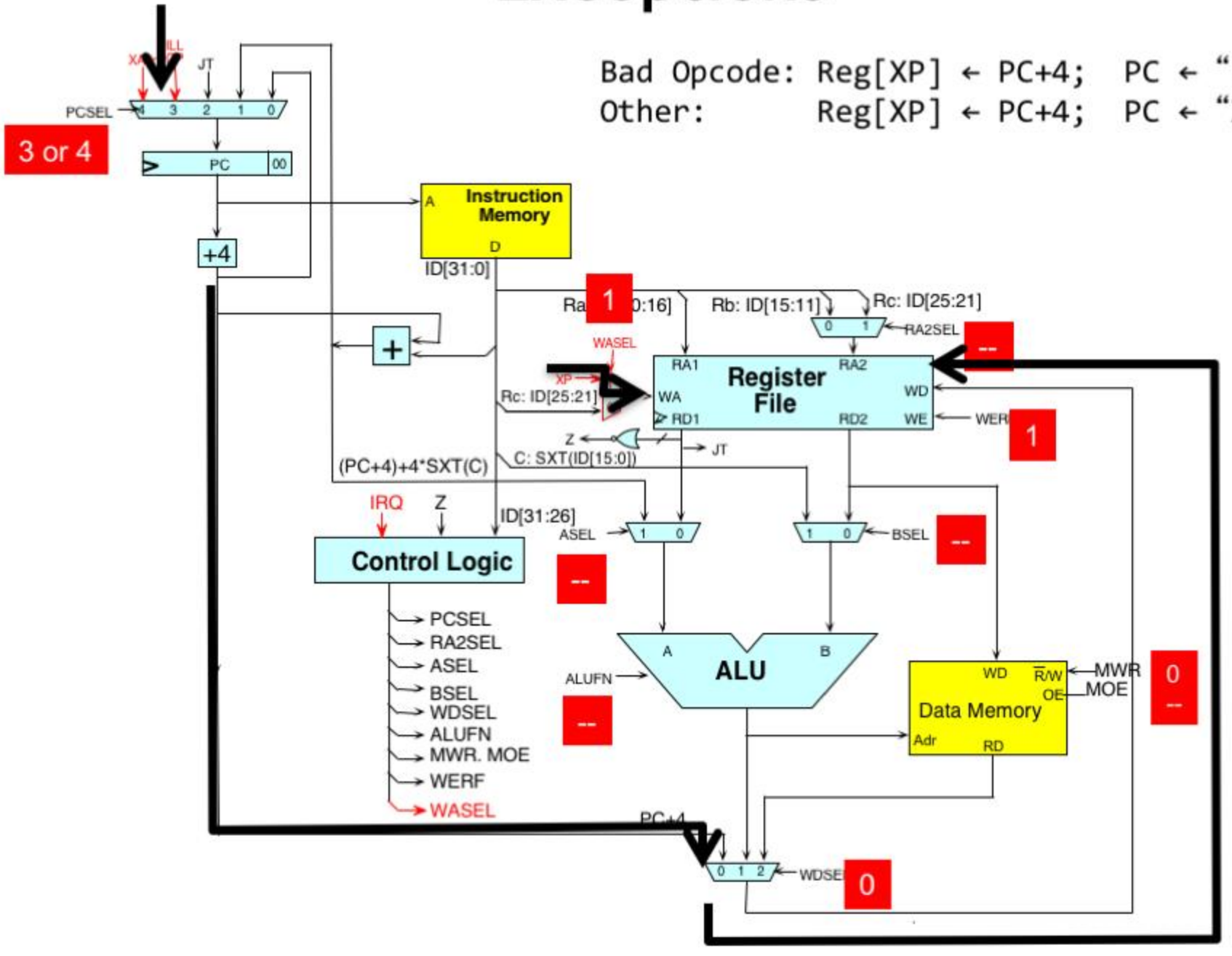
# Exceptions

```
Bad Opcode: Reg[XP] ← PC+4;   PC ← "IllOp"
Other:      Reg[XP] ← PC+4;   PC ← "Xadr"
```

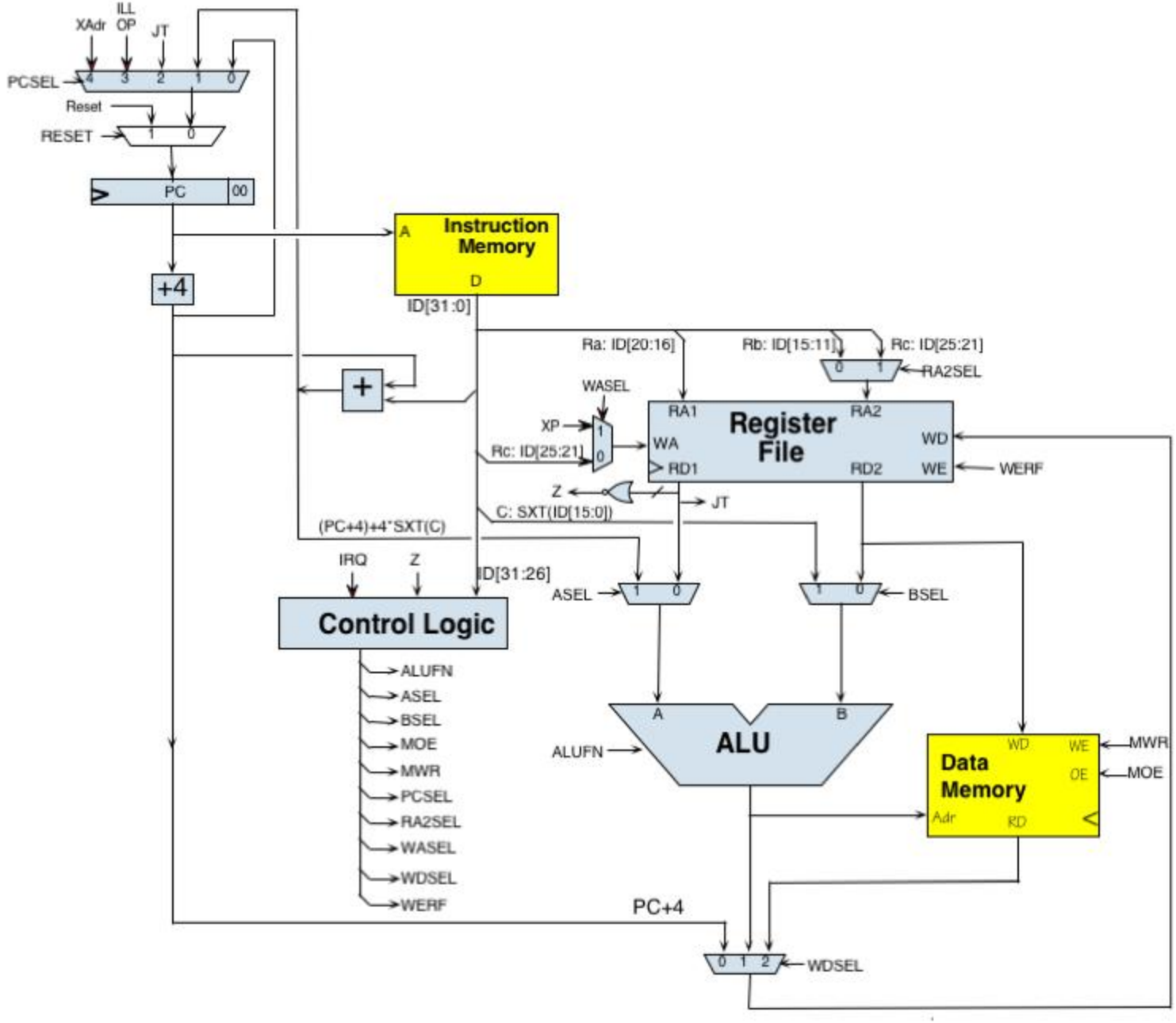


# Exceptions

Bad Opcode: Reg[XP] ← PC+4; PC ← "IllOp"
Other: Reg[XP] ← PC+4; PC ← "Xadr"



# Beta: Our "Final Answer"

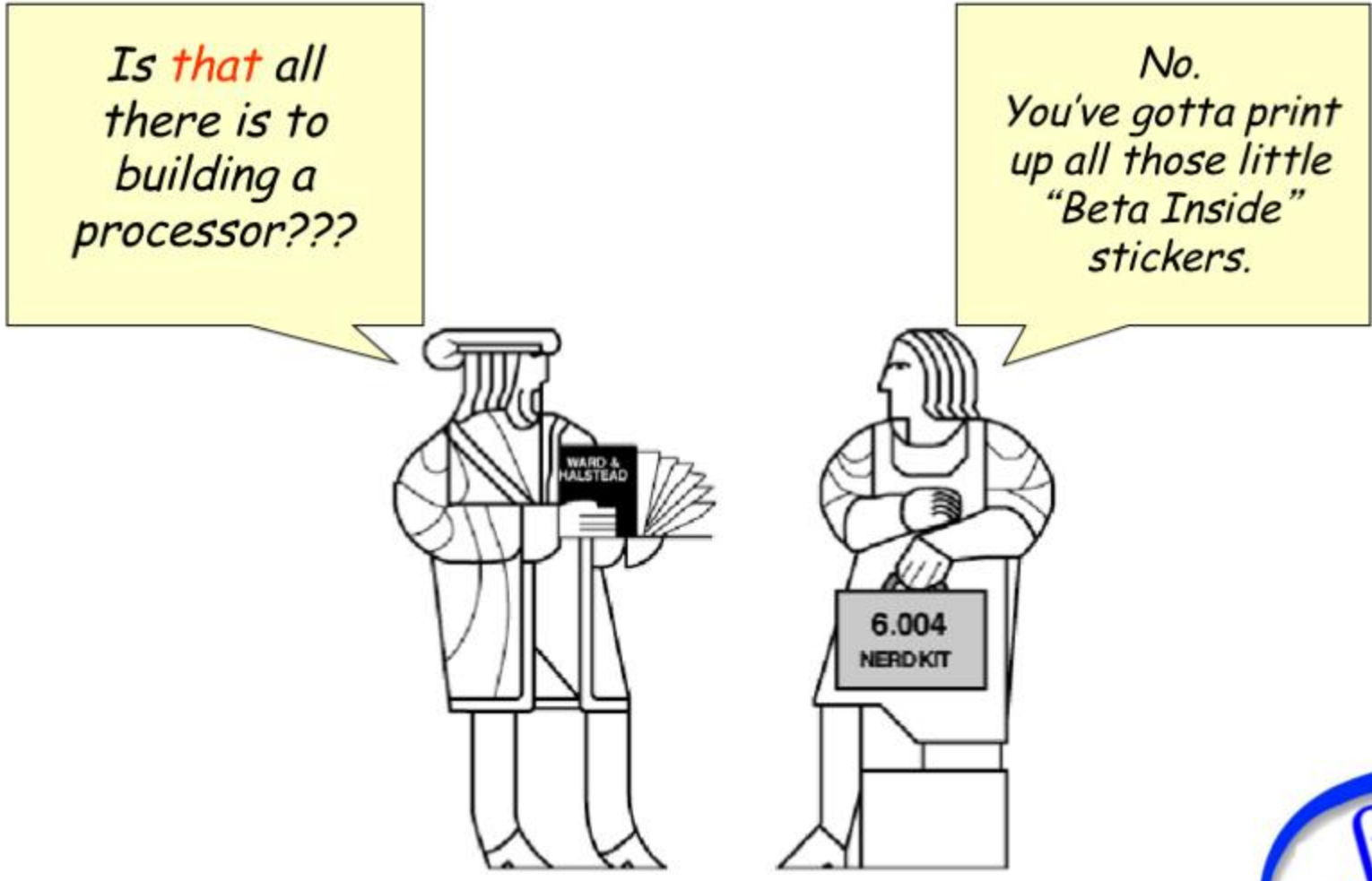


# Control Logic

| | RESET | IRQ | OP | OPC | LD | LDR | ST | JMP | BEQ | BNE | ILLOP |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ALUFN | -- | -- | F(op) | F(op) | "+" | "A" | "+" | -- | -- | -- | -- |
| ASEL | -- | -- | 0 | 0 | 0 | 1 | 0 | -- | -- | -- | -- |
| BSEL | -- | -- | 0 | 1 | 1 | -- | 1 | -- | -- | -- | -- |
| MOE | -- | -- | -- | -- | 1 | 1 | 0 | -- | -- | -- | -- |
| MWR | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| PCSEL | -- | 4 | 0 | 0 | 0 | 0 | 0 | 2 | Z ? 1 : 0 | Z ? 0 : 1 | 3 |
| RA2SEL | -- | -- | 0 | -- | -- | -- | 1 | -- | -- | -- | -- |
| WASEL | -- | 1 | 0 | 0 | 0 | 0 | -- | 0 | 0 | 0 | 1 |
| WDSEL | -- | 0 | 1 | 1 | 2 | 2 | -- | 0 | 0 | 0 | 0 |
| WERF | -- | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |

Implementation choices:

- 64-location ROM indexed by opcode with external logic to handle changes due to Z and IRQ inputs
- Entirely combinational logic (faster, but much more work!)


Is that all there is to building a processor???
No. You've gotta print up all those little "Beta Inside" stickers.
WARD & HALSTEAD
6.004
NERD KIT
β inside