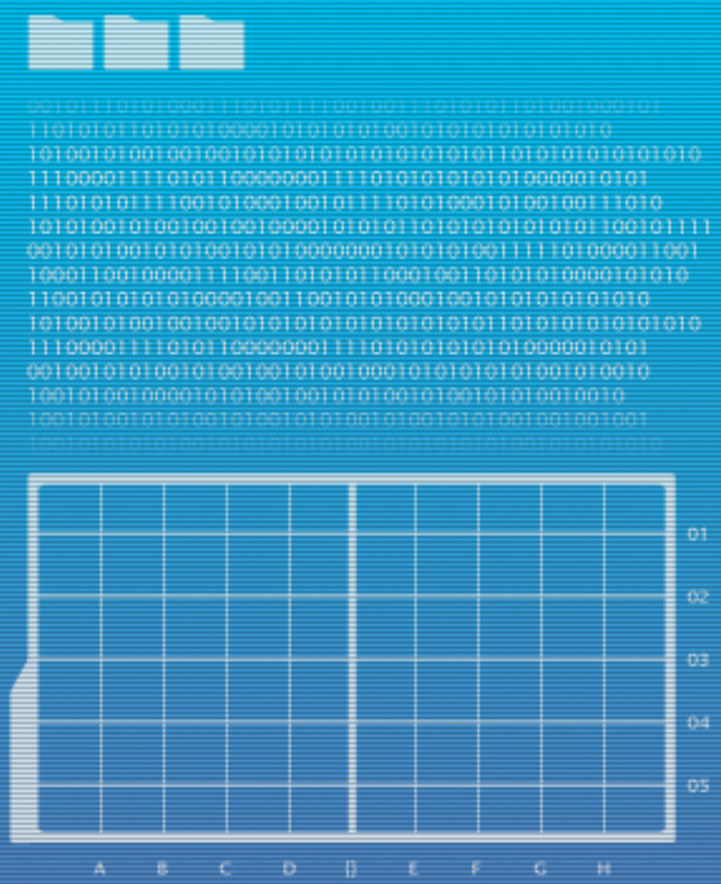
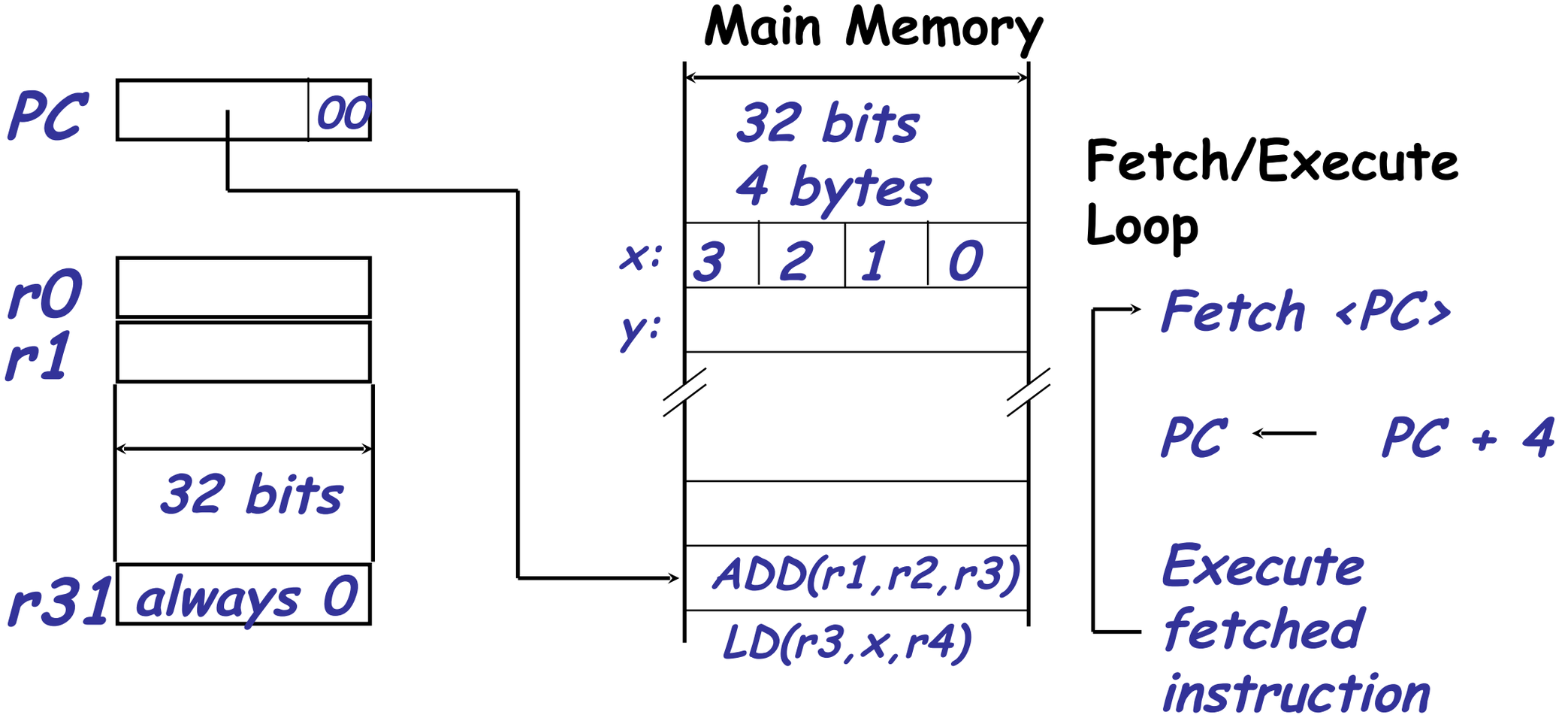
DEPARTMENT OF
INFORMATION SYSTEMS
AND COMPUTER SCIENCE



Implementing the Beta in Hardware

A Convenient Guide for the Final Lab Exercise

Recap: β Visible State & Computation Model



How do we implement this?

Approach: Incremental Design

- Divide instructions into various instruction classes based on **OPCODE** field:
  - Operate instructions: **ADD**, **ADDC**, etc.
  - Load and Store instructions: **LD**, **ST**
  - Jump and Branch instructions: **JMP**, **BEQ**, etc.
  - Exceptions: **LDR**
- Implement datapaths for each class individually.
- Merge datapaths.

β Instruction Set Architecture

Form 1:

OPCODE	*rc*	*ra*	*rb*	unused
6	*5*	*5*	*5*	*11*

OPCODE = 10xxxx *ADD:* *rc ← <ra> + <rb>*

Form 2:

OPCODE	*rc*	*ra*	literal C (signed)
6	*5*	*5*	*16*

OPCODE = 11xxxx *ADDC:* *rc ← <ra> + C*

OPCODE = 01xxxx

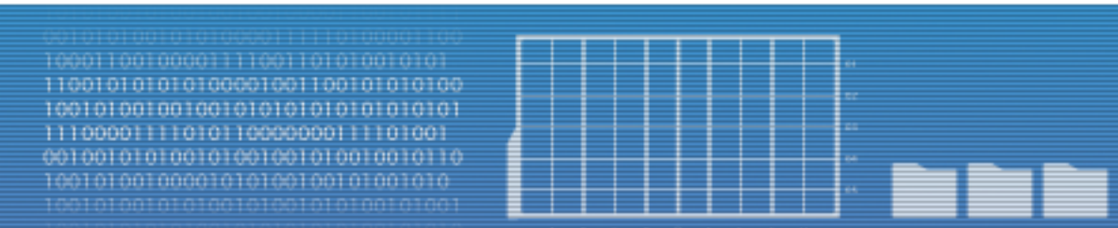
LD: *rc ← MEM[<ra> + C] *LD(ra, C, rc)*

ST: *MEM[<ra> + C] ← rc *ST(rc, C, ra)*

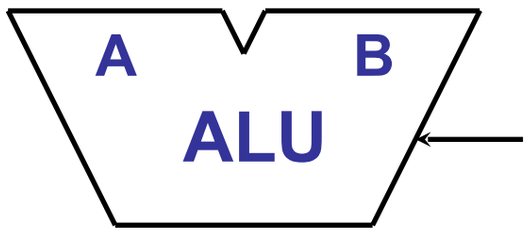
JMP: *rc ← <PC> + 4; PC ← <ra>*

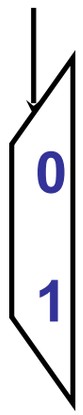
BEQ: *rc ← <PC> + 4; if <ra>=0 then PC ←(<PC> + 4) + C*4*

LDR: *rc ← MEM[(<PC> + 4) + C*4]*

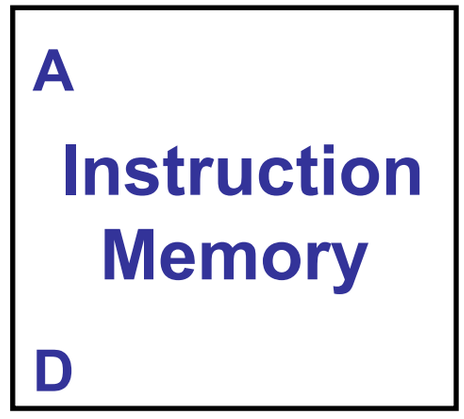


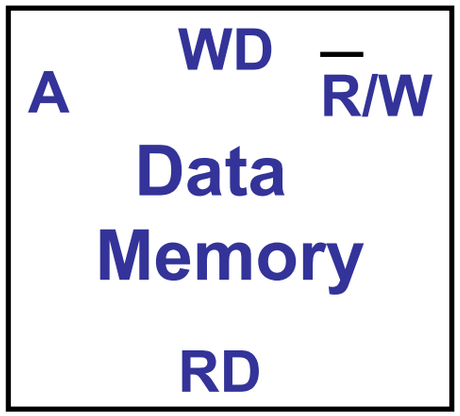
Datapath Components





MUX

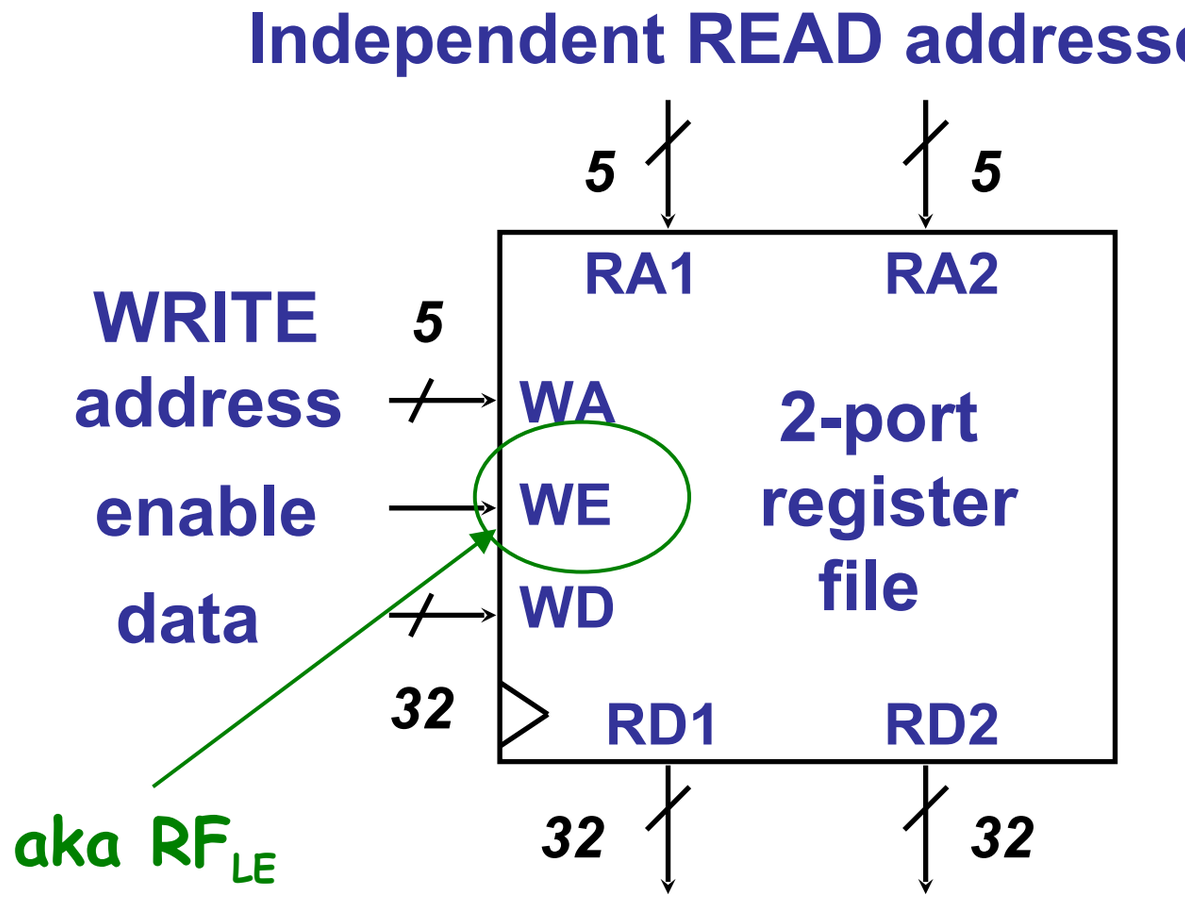




2-Port Register File

2 Combinational READ ports, 1 clocked WRITE port

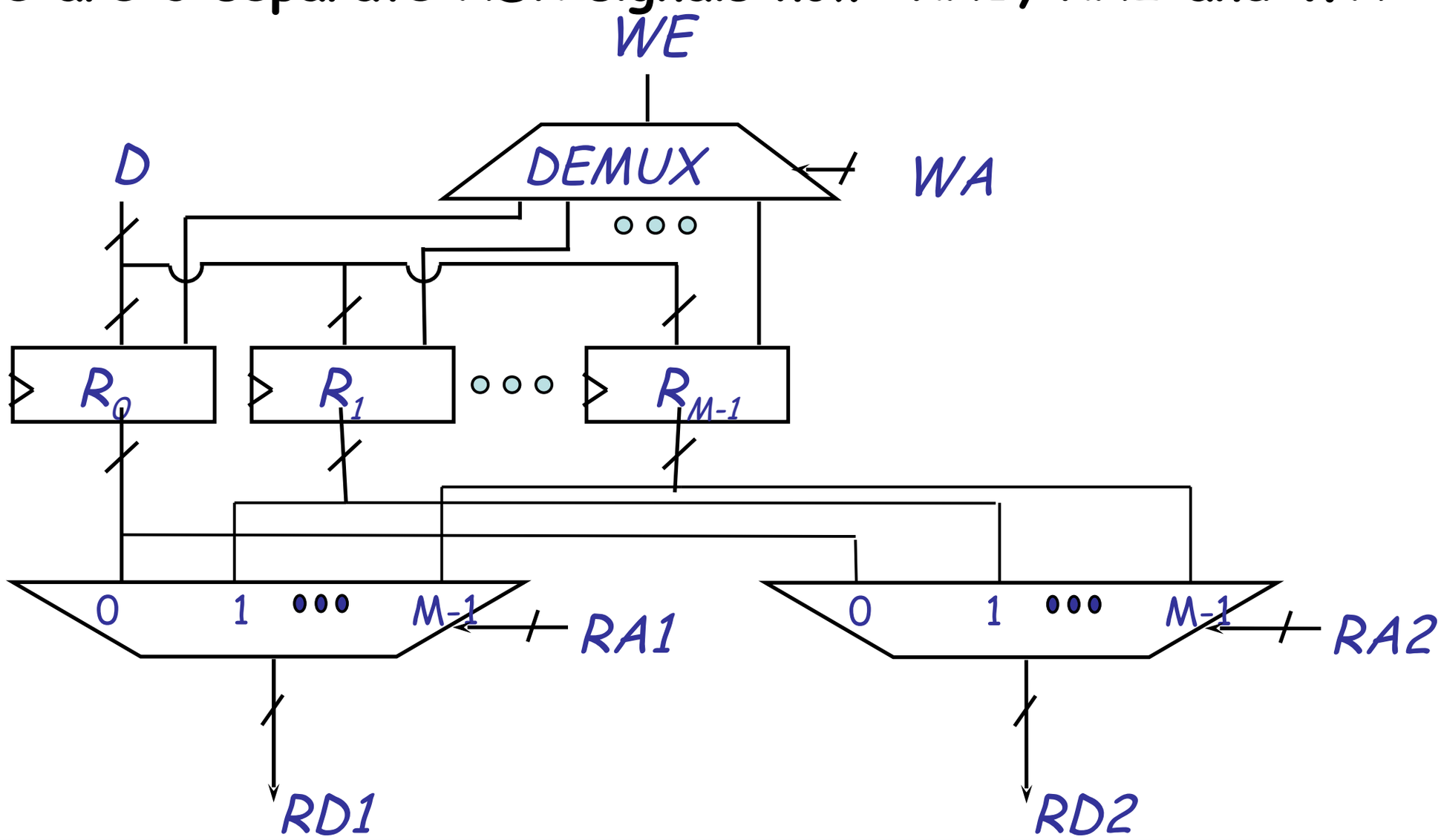
- One data input (WD), two data outputs (RD1, RD2)
- There are 3 separate ADR signals now: RA1, RA2 and WA



2-Port Register File

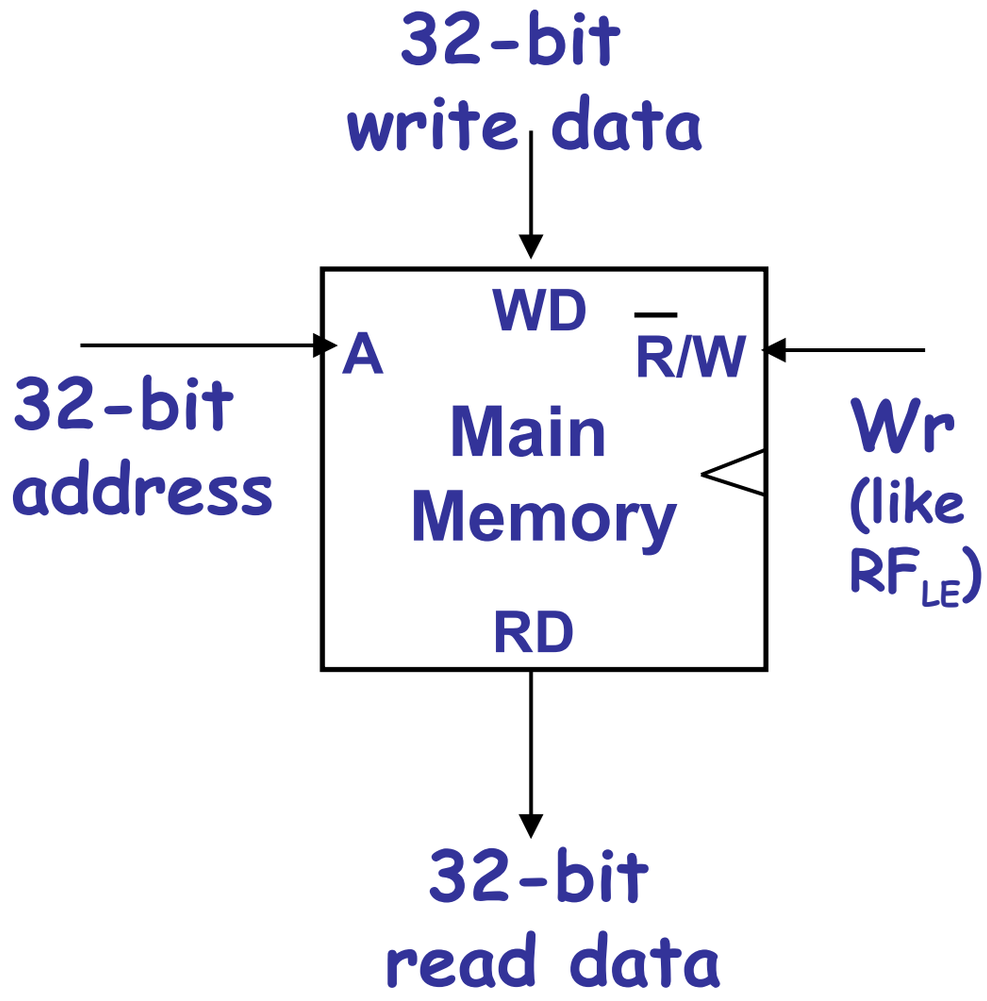
2 Combinational READ ports, 1 clocked WRITE port

- One data input (WD), two data outputs (RD1, RD2)
- There are 3 separate ADR signals now: RA1, RA2 and WA

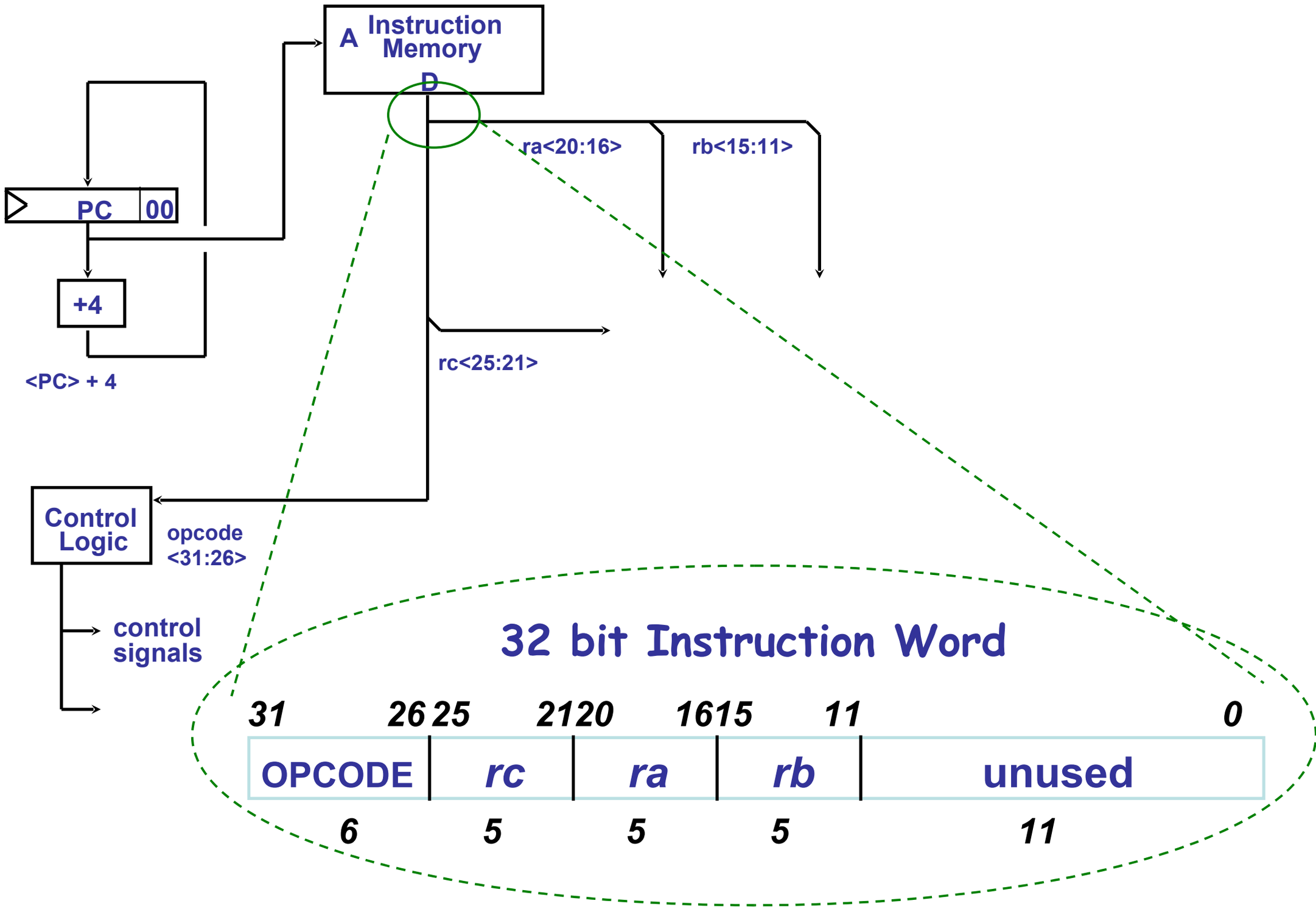


Main Memory

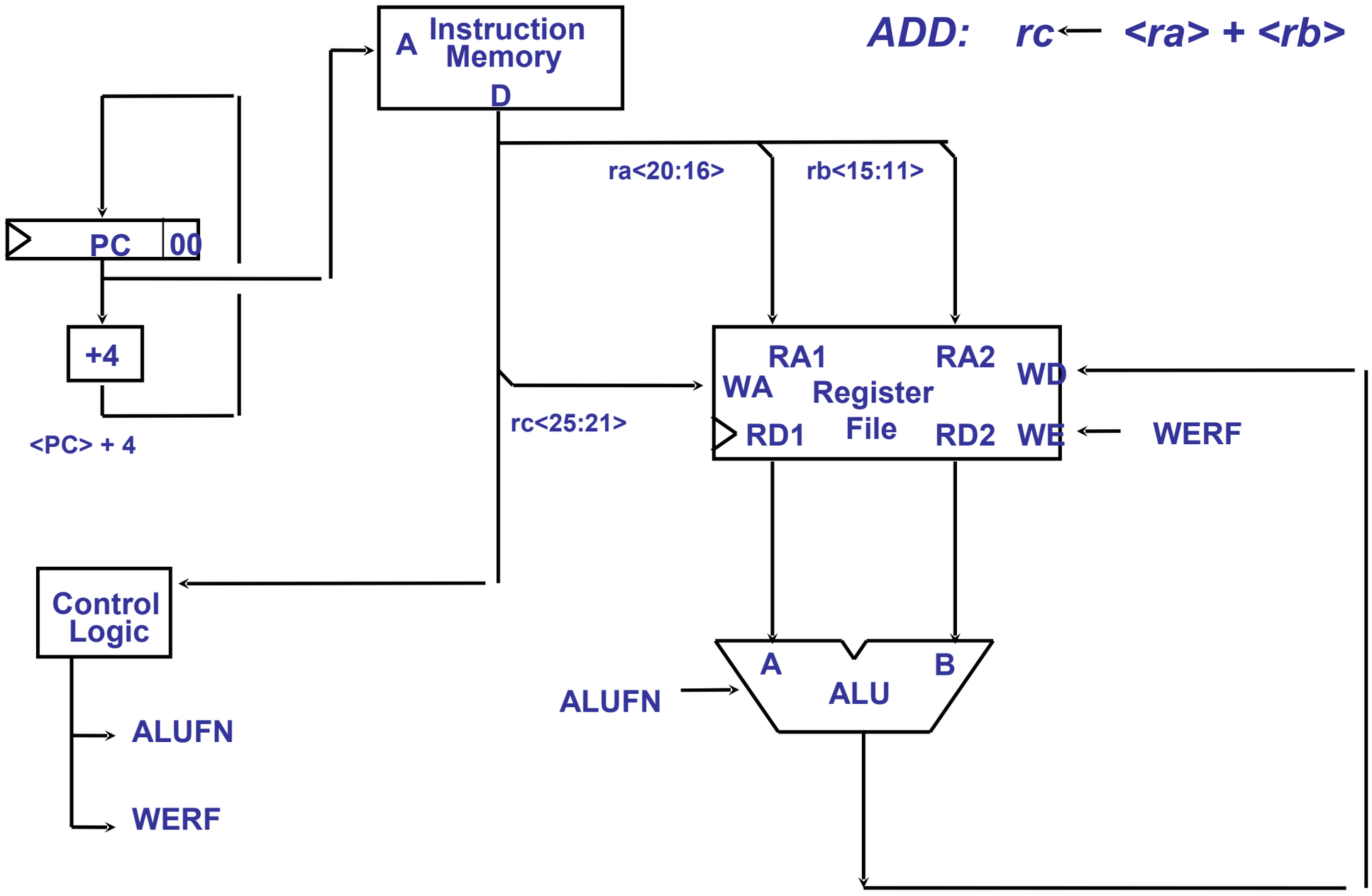
- Signals are similar to original Reg File.
  - ADR (A) signal is bigger but is shared by read and write.
  - Combinational read.
  - Clocked write.
- Memory has two "copies":
  - Instruction memory.
  - Data memory.
- aka "Harvard Architecture" after the Harvard Mark-1.
- In reality, we have only 1 main memory, but use it for both instructions and data with separate caches.
  - So looks like Harvard anyway.



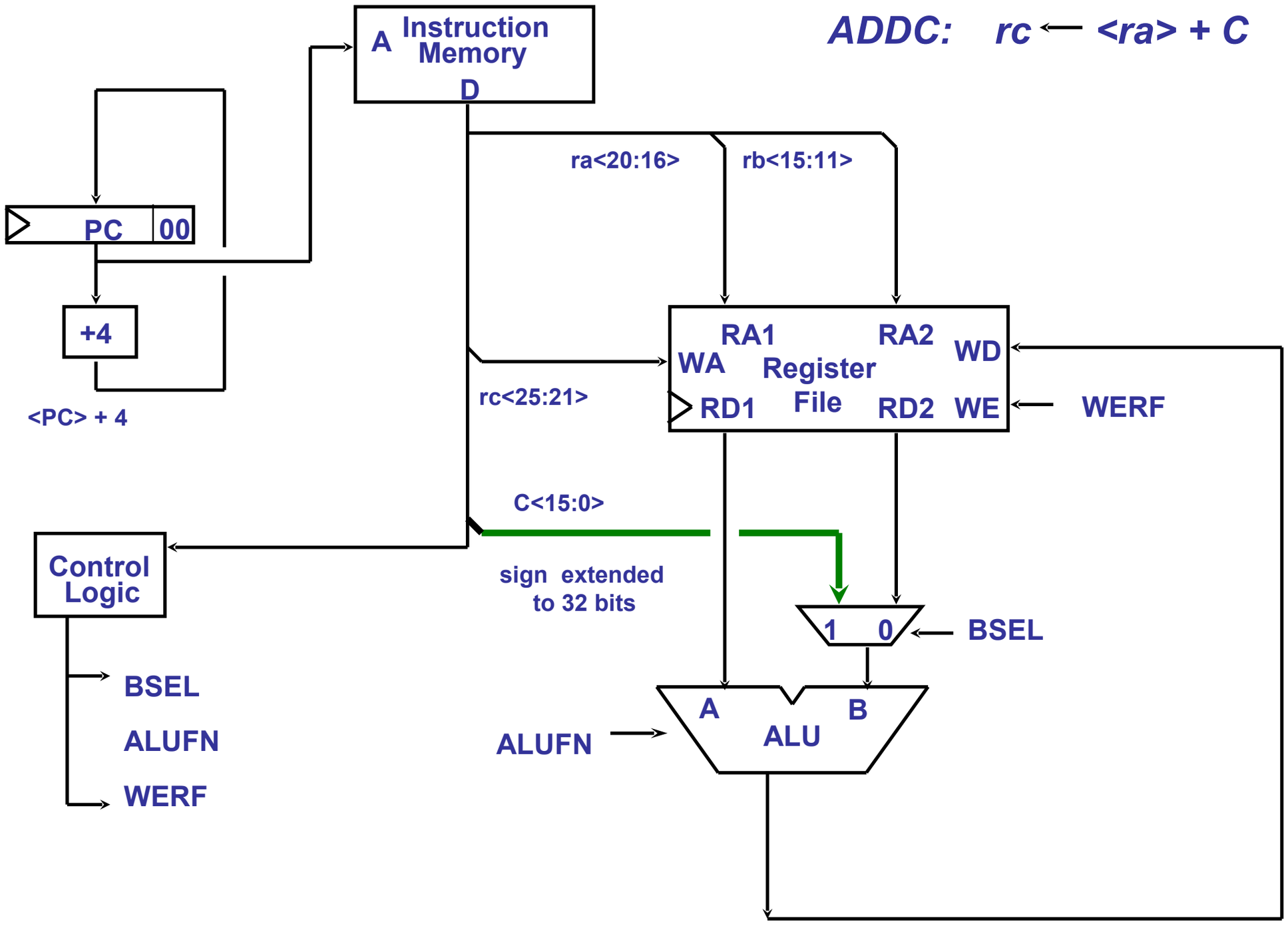
Instruction Fetch



ALU Register–Register Operations

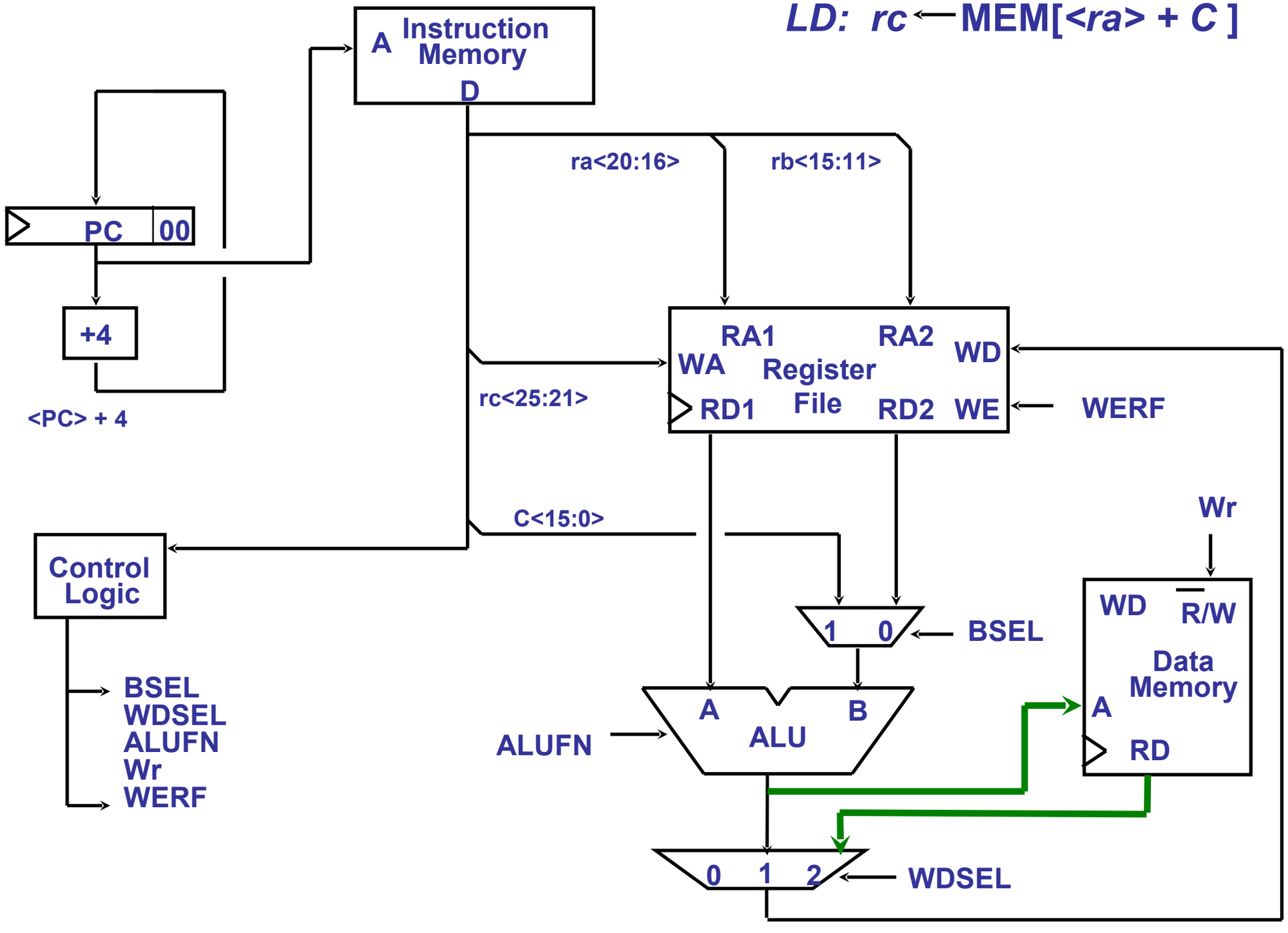


ALU Register–Constant Operations

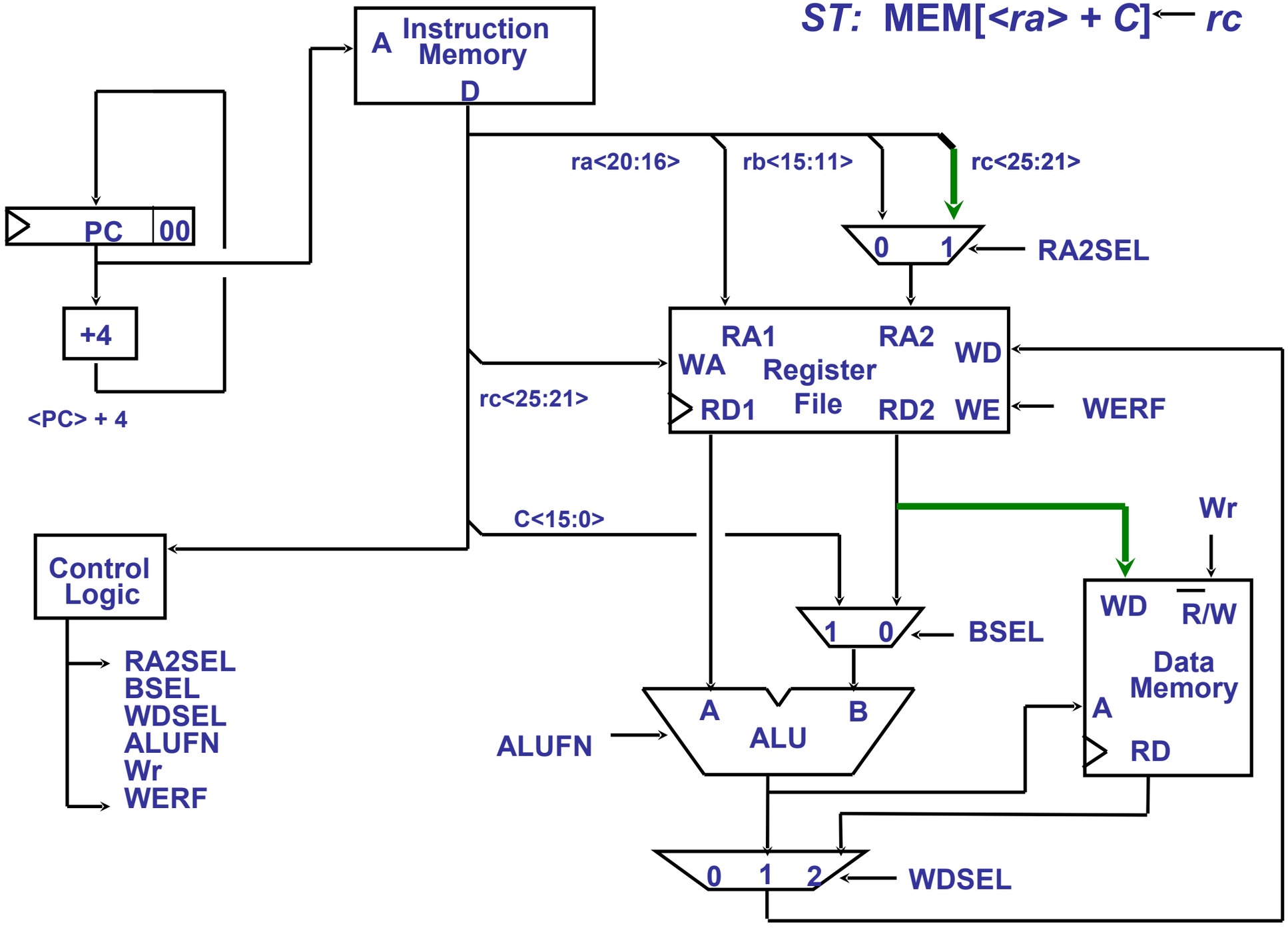


Load Instruction

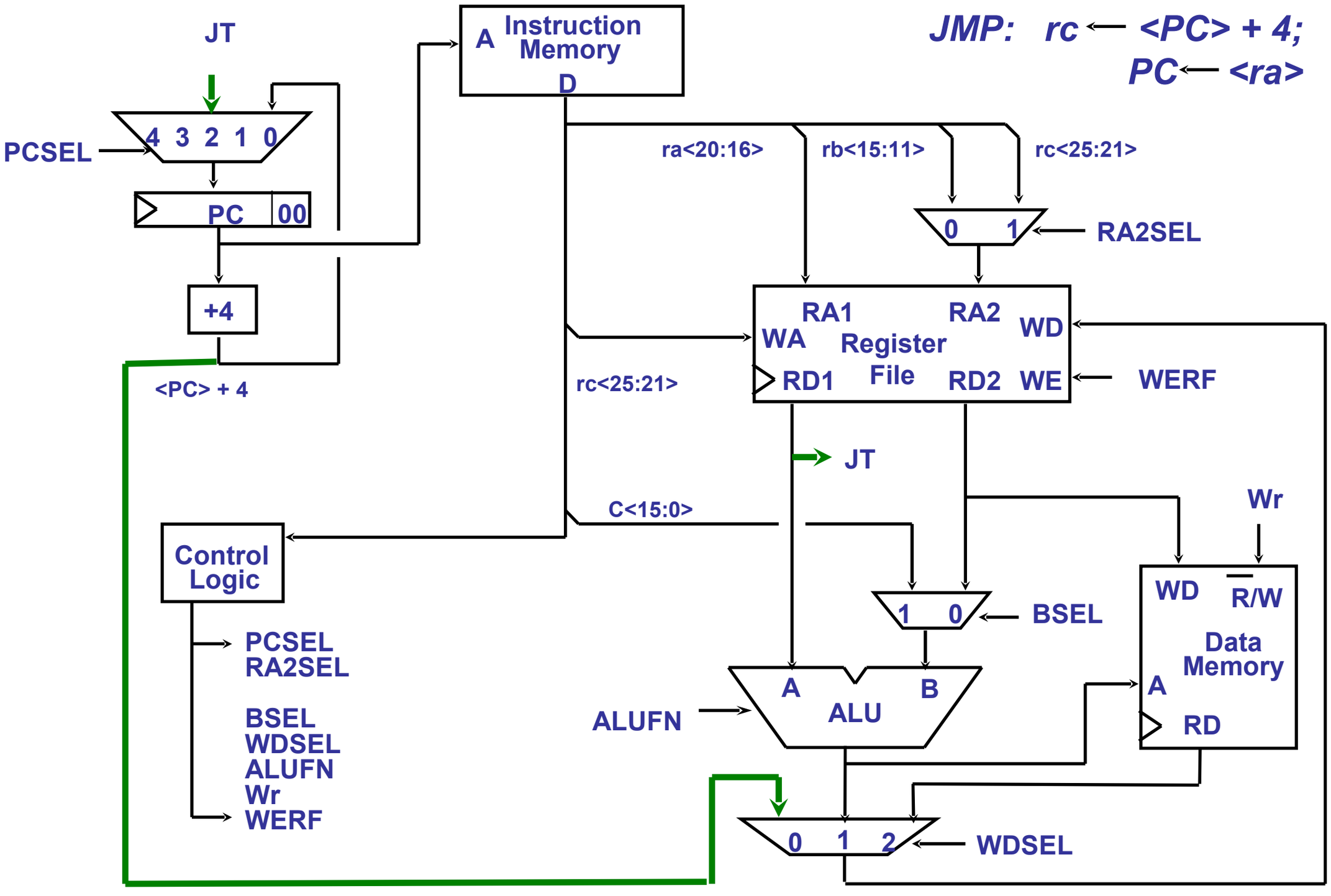
LD: rc ← MEM[<*ra*> + *C*]



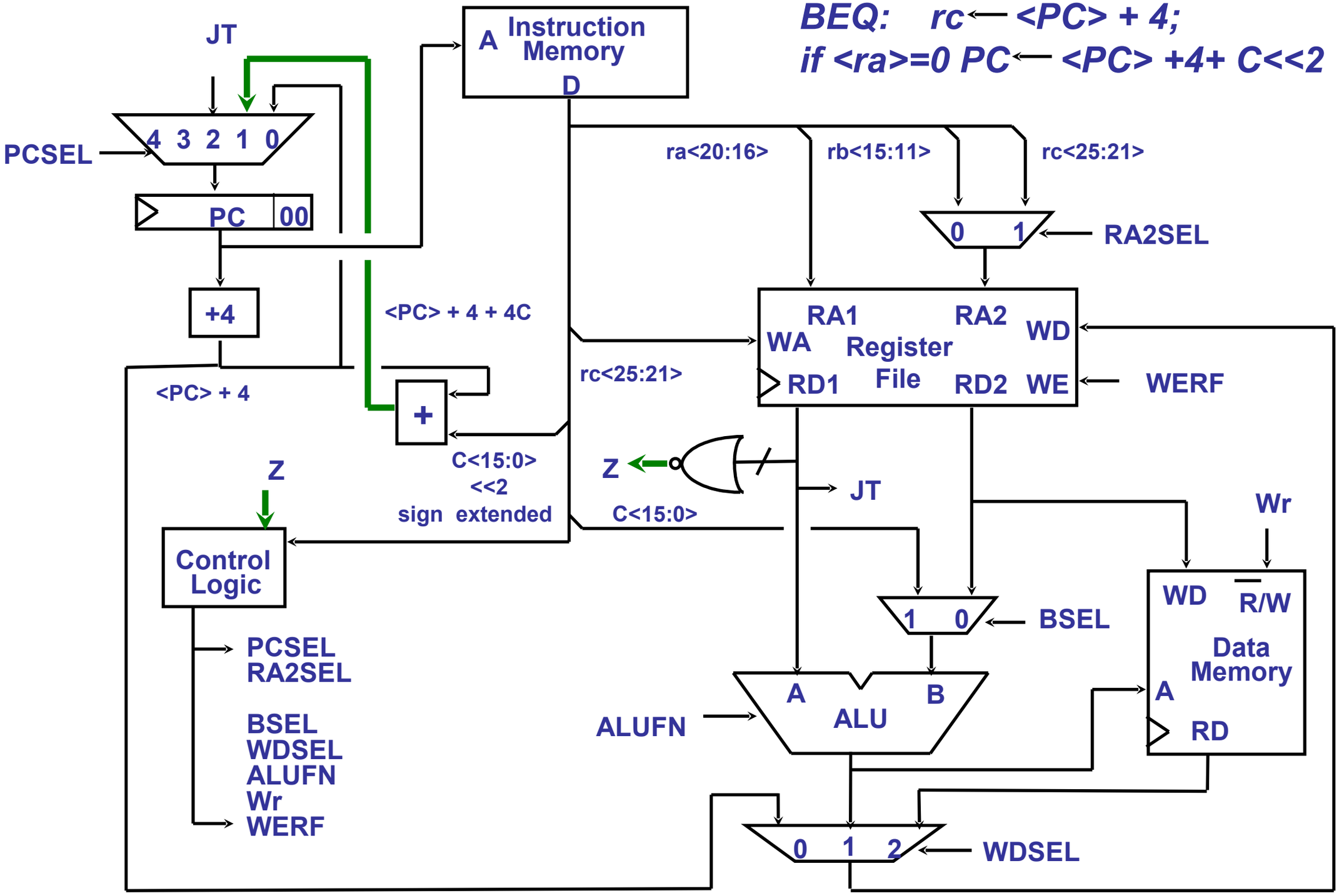
Store Instruction



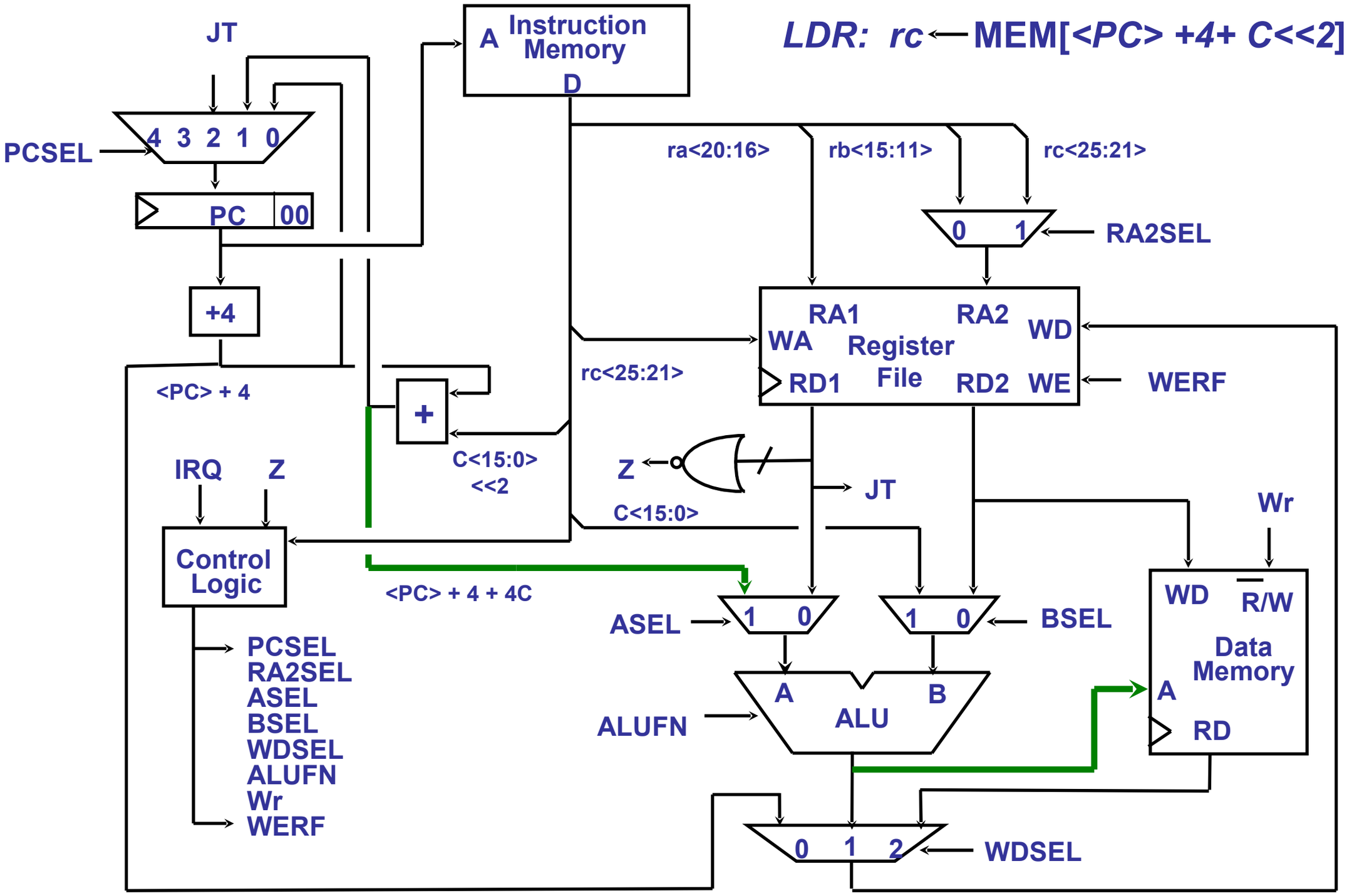
Jump Instruction



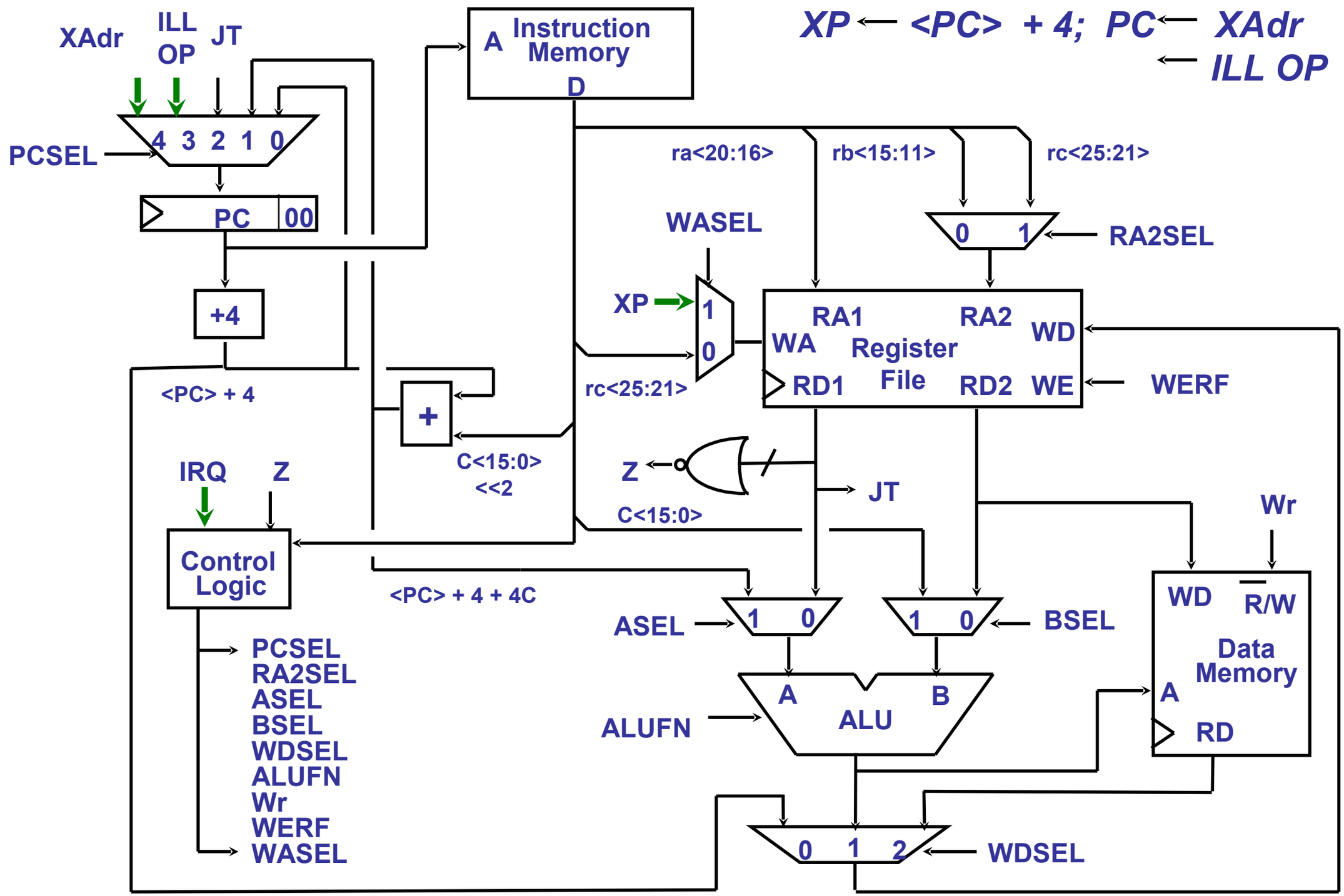
Branch Instructions (Final Lab)



Load Relative Instruction



Exceptions: Traps, Faults and Interrupts



Exceptions and Interrupts

- What's XP for? Handling "INTERRUPTS" aka "Exceptions"
  - FAULTS
    - CPU or SYSTEM generated (synchronous)
    - e.g., Illegal Instruction
  - TRAPS & system calls
    - CPU generated (synchronous)
    - e.g., read-a-character
  - I/O events (IRQ)
    - externally generated (asynchronous)
    - e.g., key struck
- How Interrupts Work:
  - Interrupt running program.
  - Invoke exception handler (like a procedure call).
  - Return to continue execution.
- NOT IMPORTANT in CS152 (not implemented in lab either)

KEY:
TRANSPARENCY
to interrupted
program.
(Most difficult
for asynchronous
interrupts.)

Exceptions and Interrupts

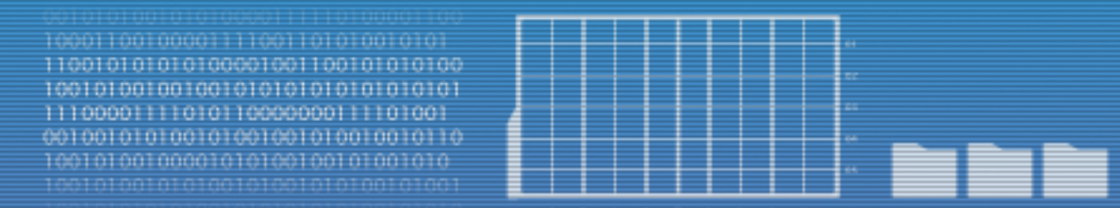
- **IMPLEMENTATION:**
  - **“FORCED” procedure call: new PC, save old PC for return.**
  - **New PC comes from:**
    - **IllOp handler address, in case of traps and faults.**
    - **XAdr, in case of interrupts (IRQ=1).**
  - **Save old PC in XP (R30).**
  - **Don’t execute current instruction in hardware.**

Example: DIV unimplemented

```
LD(R31,A,R0)
LD(R31,B,R1)
DIV(R0,R1,R2)
ST(R2,C,R31)
```

Forced by hardware

```
IllOp:
PUSH(XP)
 Fetch inst. at Mem[Reg[XP]-4],
 check for DIV opcode, get reg numbers,
 perform operation in software, fill result reg
POP(XP)
JMP(XP)
```



DISCS

Specifying Control Logic Signals

Control logic inputs

OPCODE
Z

Signals may be taken from part of OPCODE

conditional

INSTR.	ADD	ADDC	LD	ST	BEQ	BNE	JMP	...
PCSEL	0				Z			
RA2SEL	0				*			
BSEL	0				*			
WDSEL	1				0			
ALUFN	op				*			
Wr	0				0			
WERF	1				1			

Control logic outputs

- Looking at the datapath you should be able to fill in the entire table!
- (See Final Lab for actual implementation.)

Cheat Sheet #2: Unpipelined β Datapath