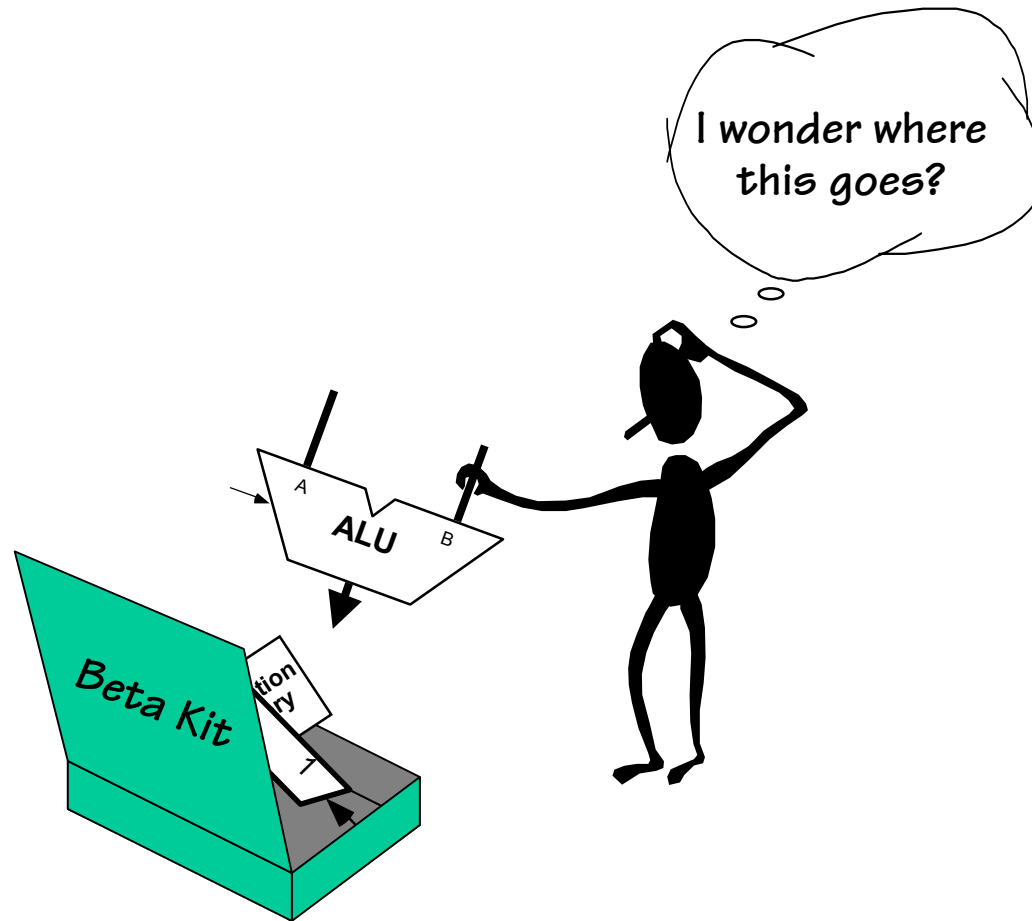


Building the Beta

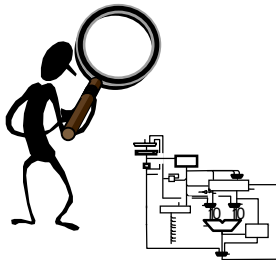


Handouts: Lecture Slides, PS 7, Lab 6

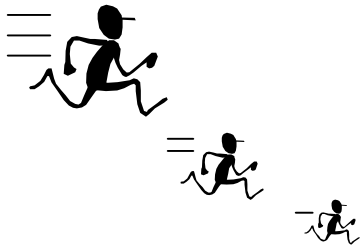
CPU Design Tradeoffs



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.

Performance Measure

Millions of Instructions per Second

$$\text{MIPS} = \frac{\text{Clock Frequency (MHz)}}{\text{C.P.I.}}$$

Clocks per instruction

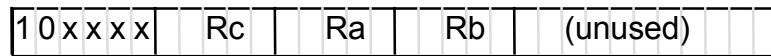
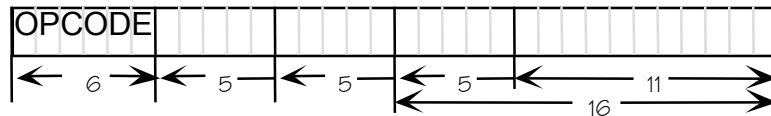
PUSHING PERFORMANCE ...

TODAY: 1 cycle/inst.

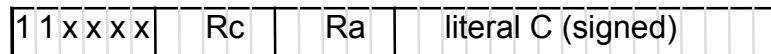
NEXT TIME: more MHz via pipelining

NEXT NEXT TIME: fixing various pipelining issues

The Beta ISA



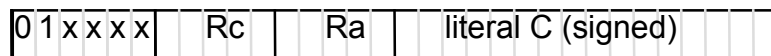
Operate class: $\text{Reg}[R_c] \leftarrow \text{Reg}[R_a] \text{ op } \text{Reg}[R_b]$



Operate class: $\text{Reg}[R_c] \leftarrow \text{Reg}[R_a] \text{ op } \text{SXT}(C)$

Opcodes, both formats:

ADD	SUB	MUL*	DIV*	*optional
CMPEQ	CMPL	CMPLT		
AND	OR	XOR		
SHL	SHR	SRA		



LD: $\text{Reg}[R_c] \leftarrow \text{Mem}[\text{Reg}[R_a] + \text{SXT}(C)]$
 ST: $\text{Mem}[\text{Reg}[R_a] + \text{SXT}(C)] \leftarrow \text{Reg}[R_c]$
 JMP: $\text{Reg}[R_c] \leftarrow \text{PC} + 4; \text{PC} \leftarrow \text{Reg}[R_a]$
 BEQ: $\text{Reg}[R_c] \leftarrow \text{PC} + 4; \text{if } \text{Reg}[R_a] = 0 \text{ then } \text{PC} \leftarrow \text{PC} + 4 + 4 * \text{SXT}(C)$
 BNE: $\text{Reg}[R_c] \leftarrow \text{PC} + 4; \text{if } \text{Reg}[R_a] \neq 0 \text{ then } \text{PC} \leftarrow \text{PC} + 4 + 4 * \text{SXT}(C)$
 LDR: $\text{Reg}[R_c] \leftarrow \text{Mem}[\text{PC} + 4 + 4 * \text{SXT}(C)]$

Instruction classes distinguished by OPCODE:

OP

OPC

MEM

Transfer of Control

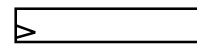
Approach: Incremental Featurism

Each instruction class can be implemented using a simple component repertoire. We'll try implementing data paths for each class individually, and merge them (using MUXes, etc).

Steps:

1. Operate instructions
2. Load & Store Instructions
3. Jump & Branch instructions
4. Exceptions
5. Merge data paths

Our Bag of Components:



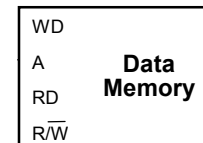
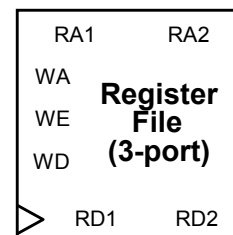
Registers



Muxes

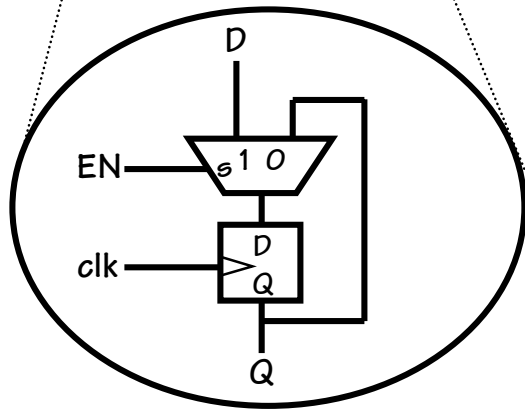
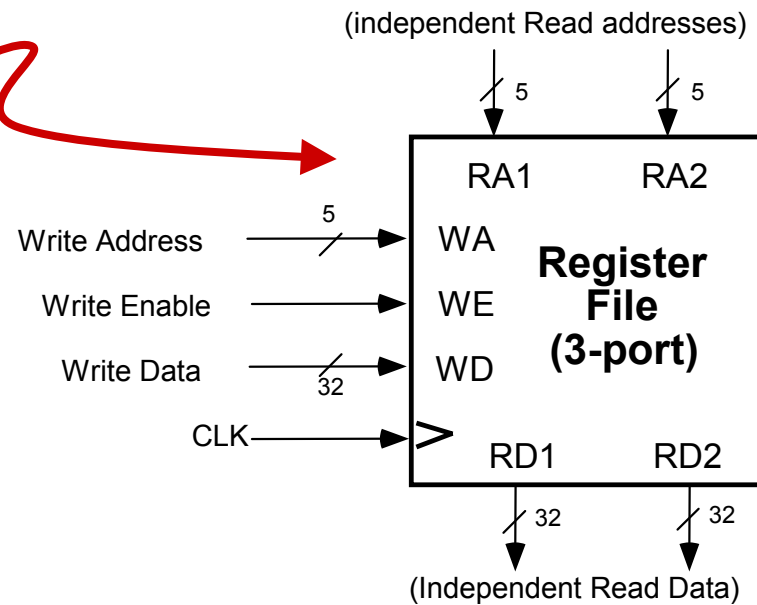
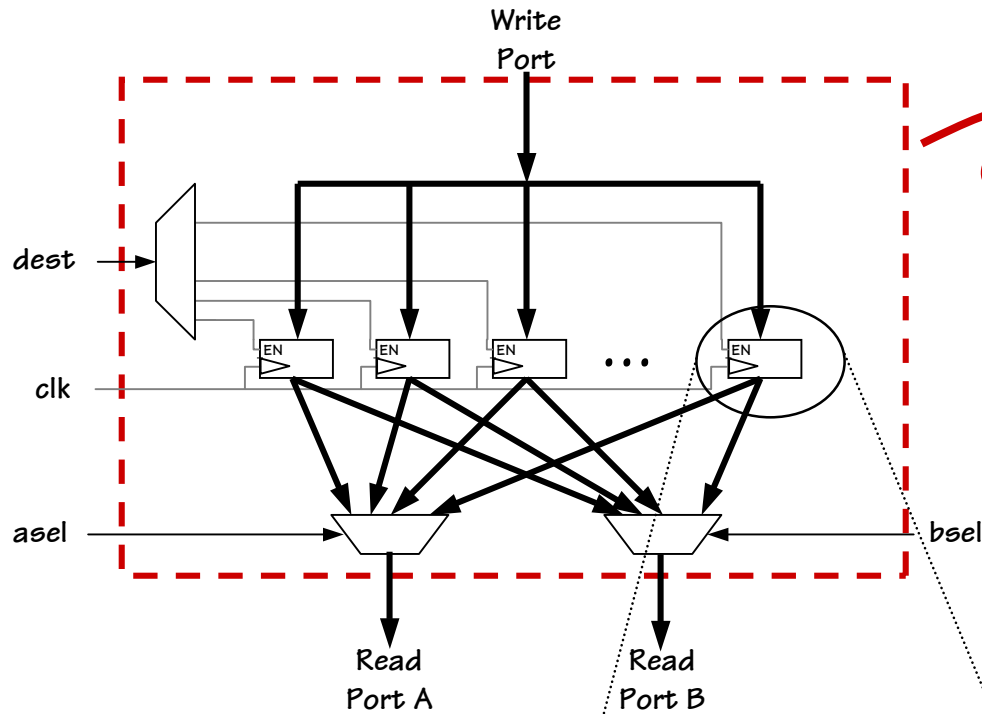


"Black box" ALU



Memories

Multi-Port Register Files

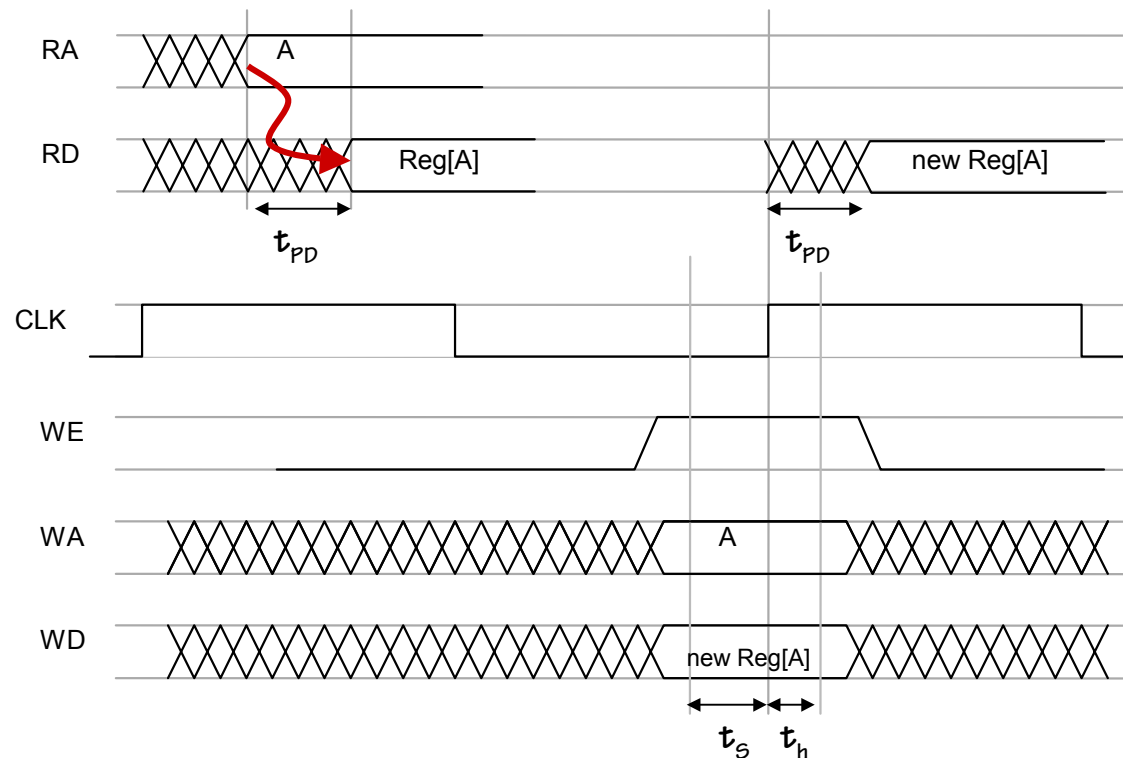


2 combinational READ ports*,
1 clocked WRITE port

**internal logic ensures Reg[31] reads as 0*

Register File Timing

2 combinational READ ports, 1 clocked WRITE port

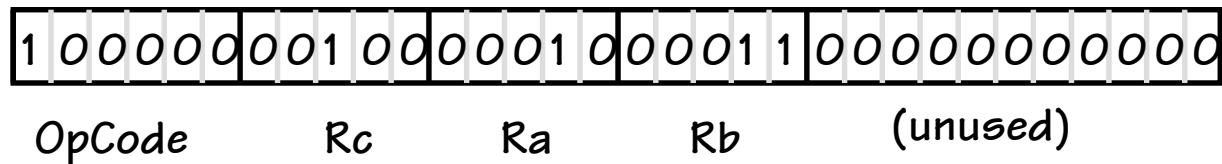


What if (say) $WA=RA1$???

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

Starting point: ALU Ops

32-bit (4-byte) ADD instruction:



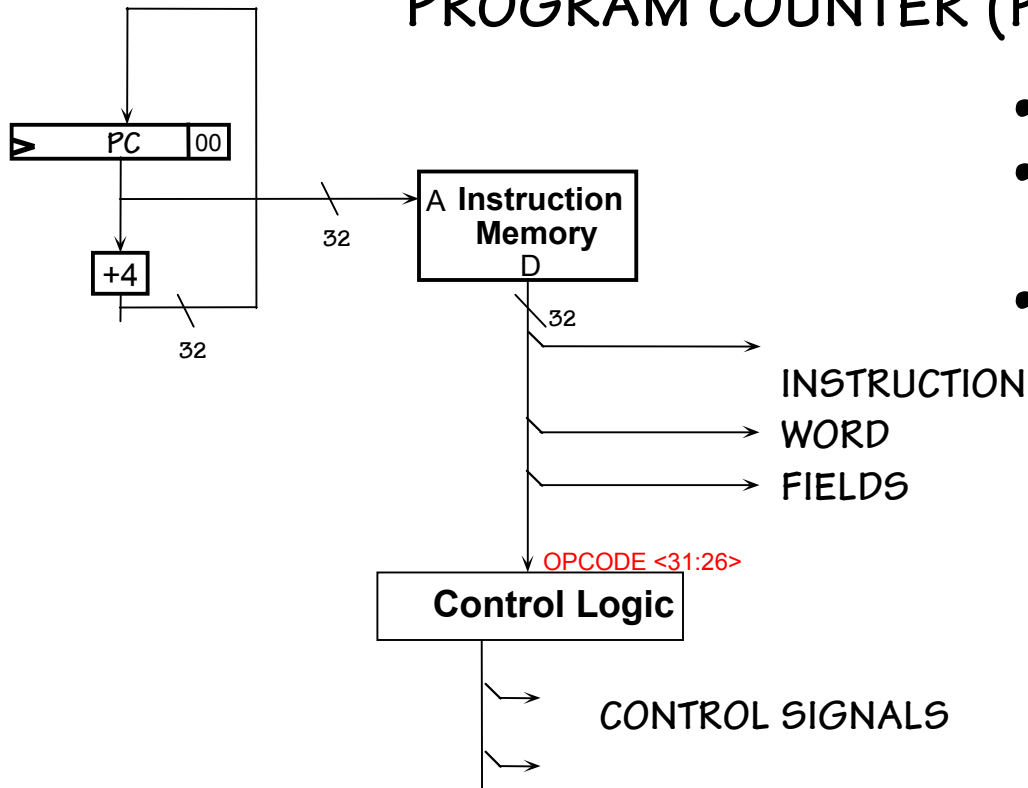
Means, to BETA, $Reg[R4] \leftarrow Reg[R2] + Reg[R3]$

First, we'll need hardware to:

- Read next 32-bit instruction
- DECODE instruction: ADD, SUB, XOR, etc
- READ operands (Ra, Rb) from Register File;
- PERFORM indicated operation;
- WRITE result back into Register File (Rc).

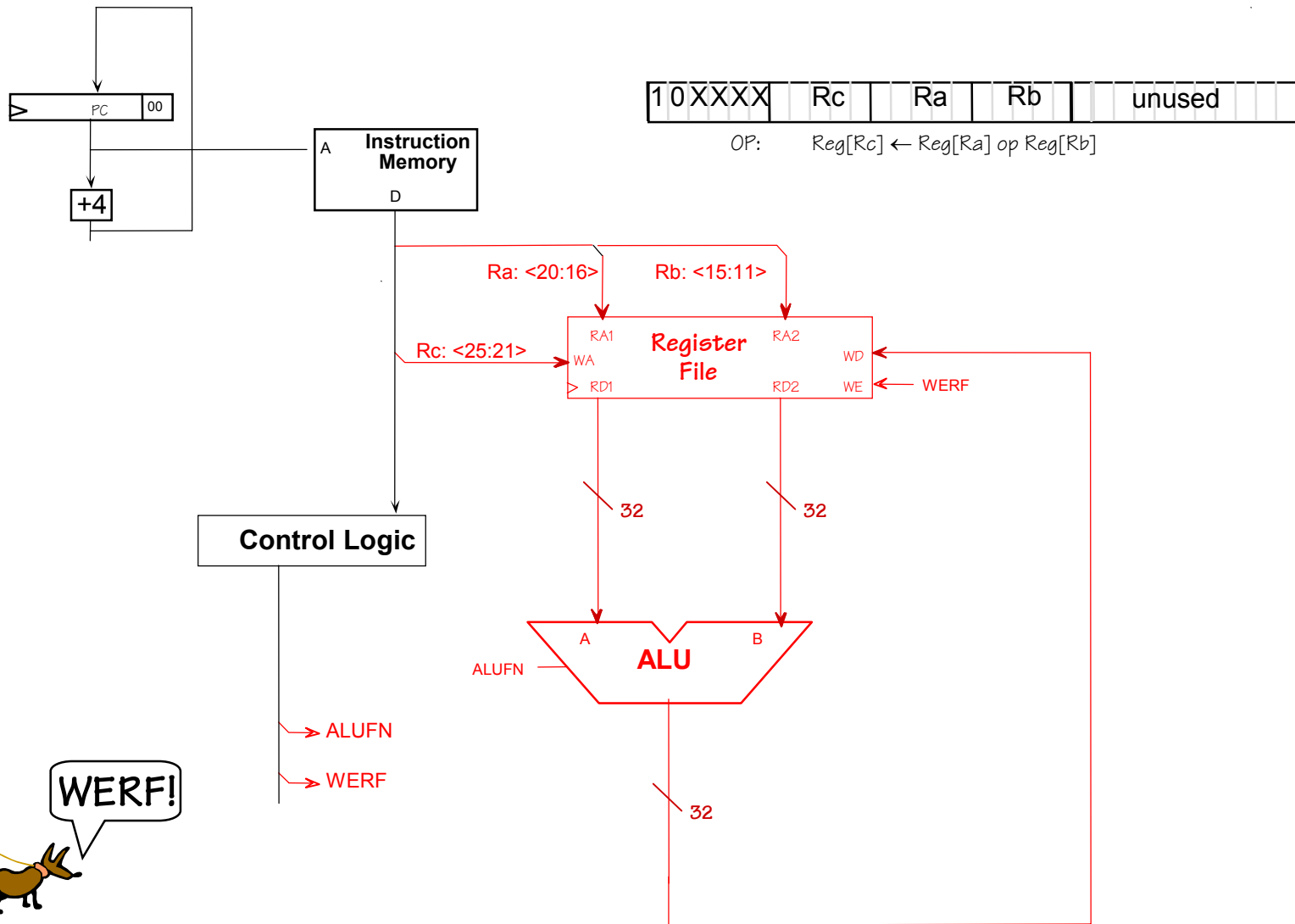
Instruction Fetch/Decode

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

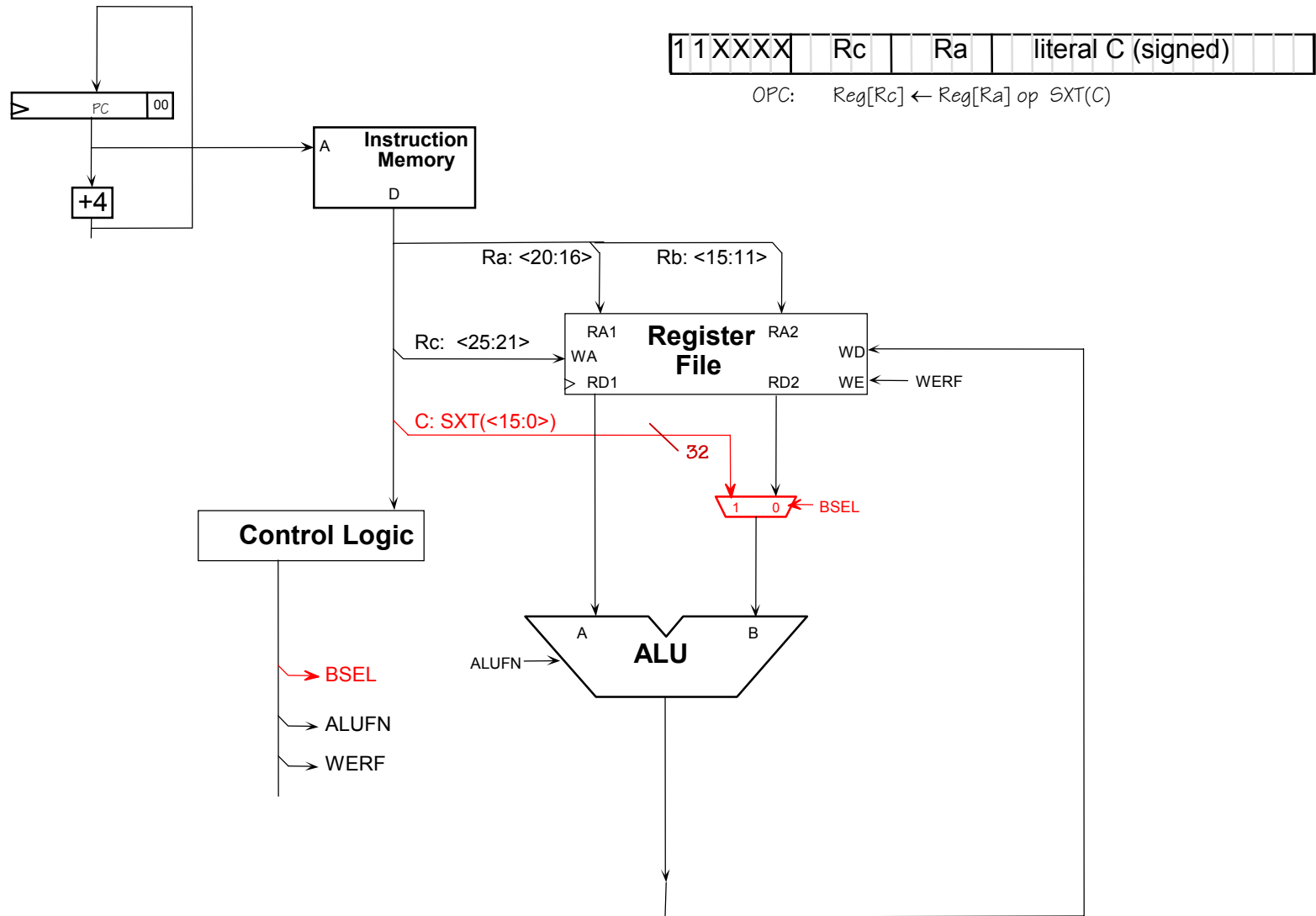


- use PC as memory address
- add 4 to PC, load new value at end of cycle
- fetch instruction from memory
 - use some instruction fields directly (register numbers, 16-bit constant)
 - use bits <31:26> to generate controls

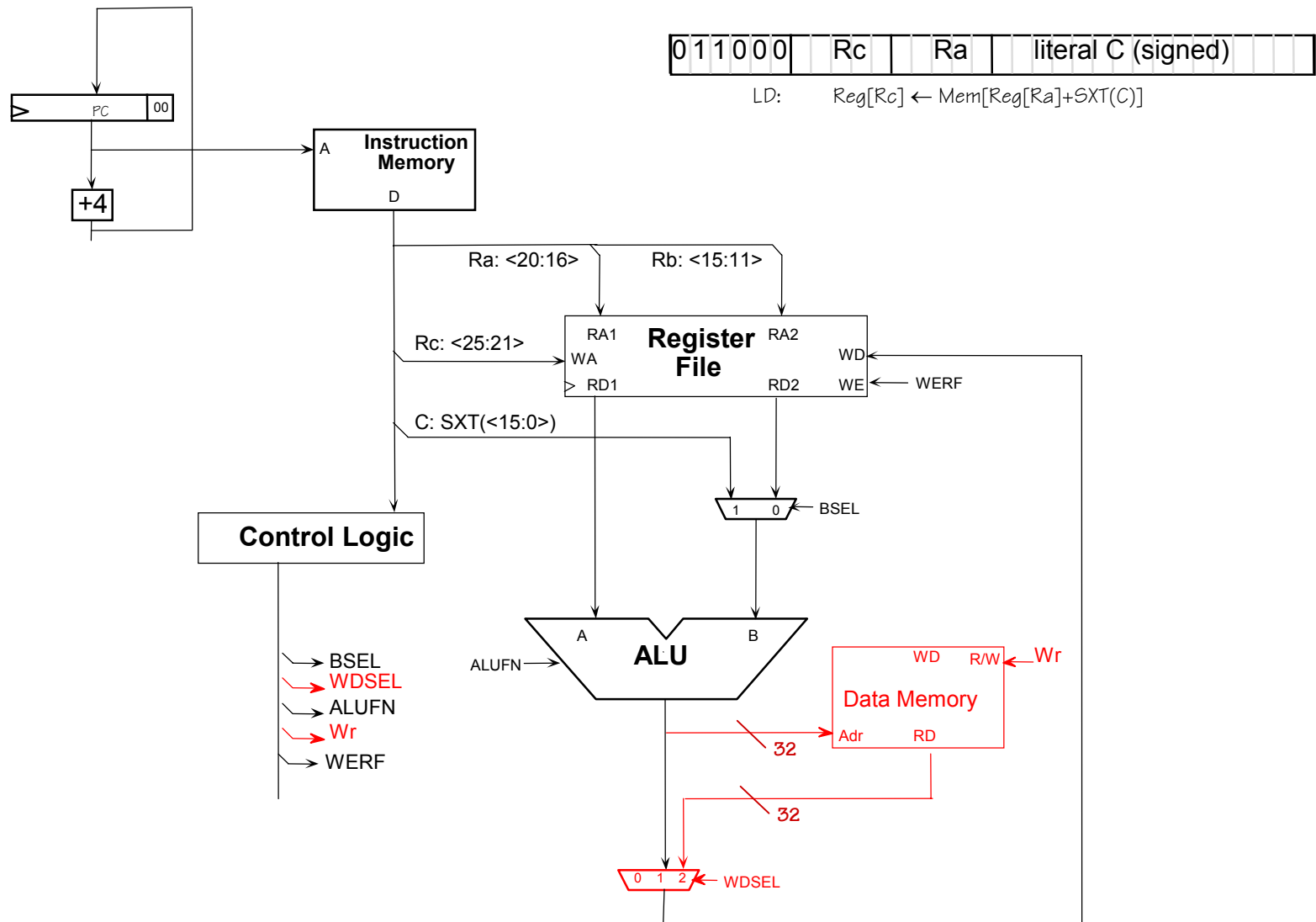
ALU Op Data Path



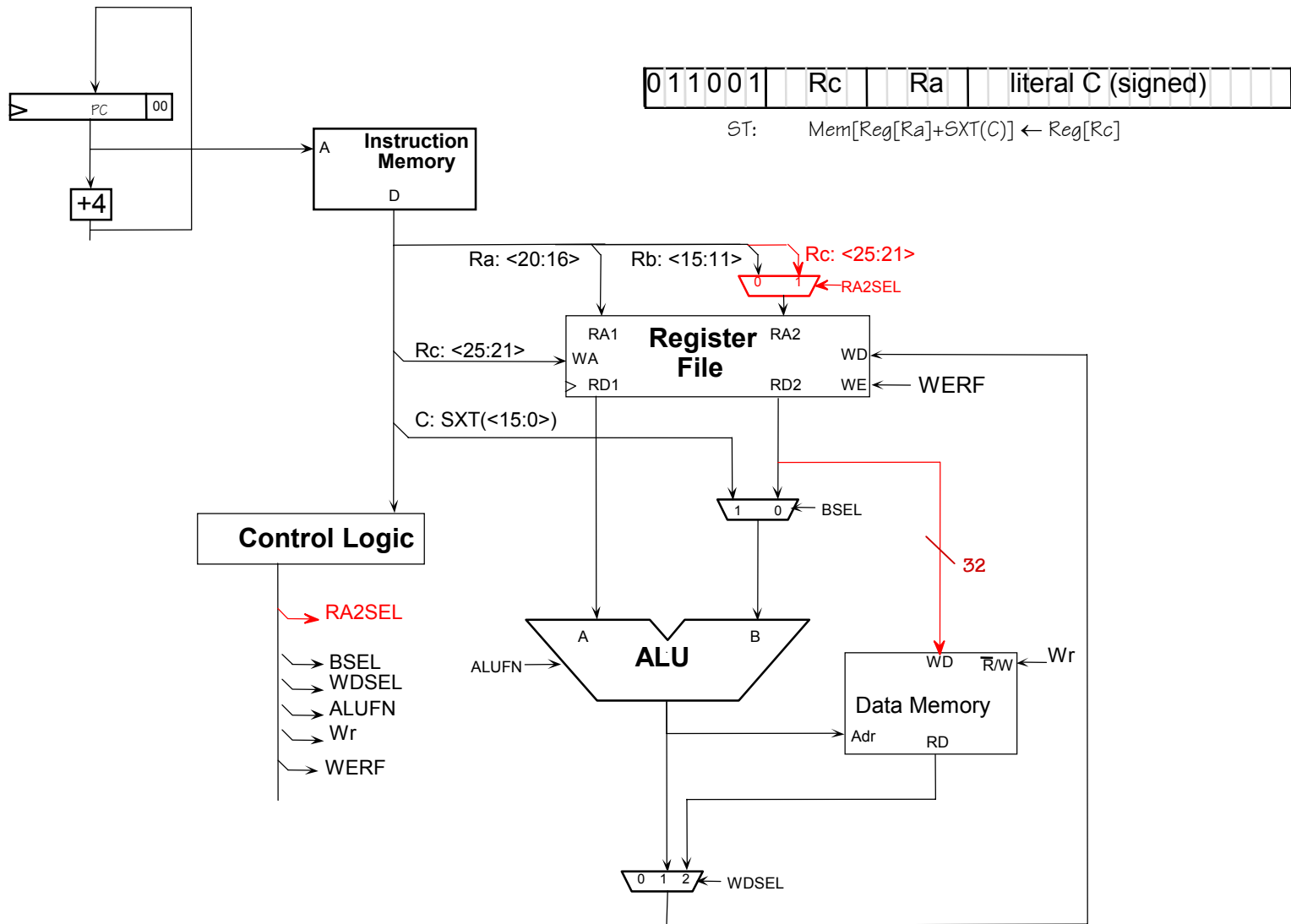
ALU Operations (w/constant)



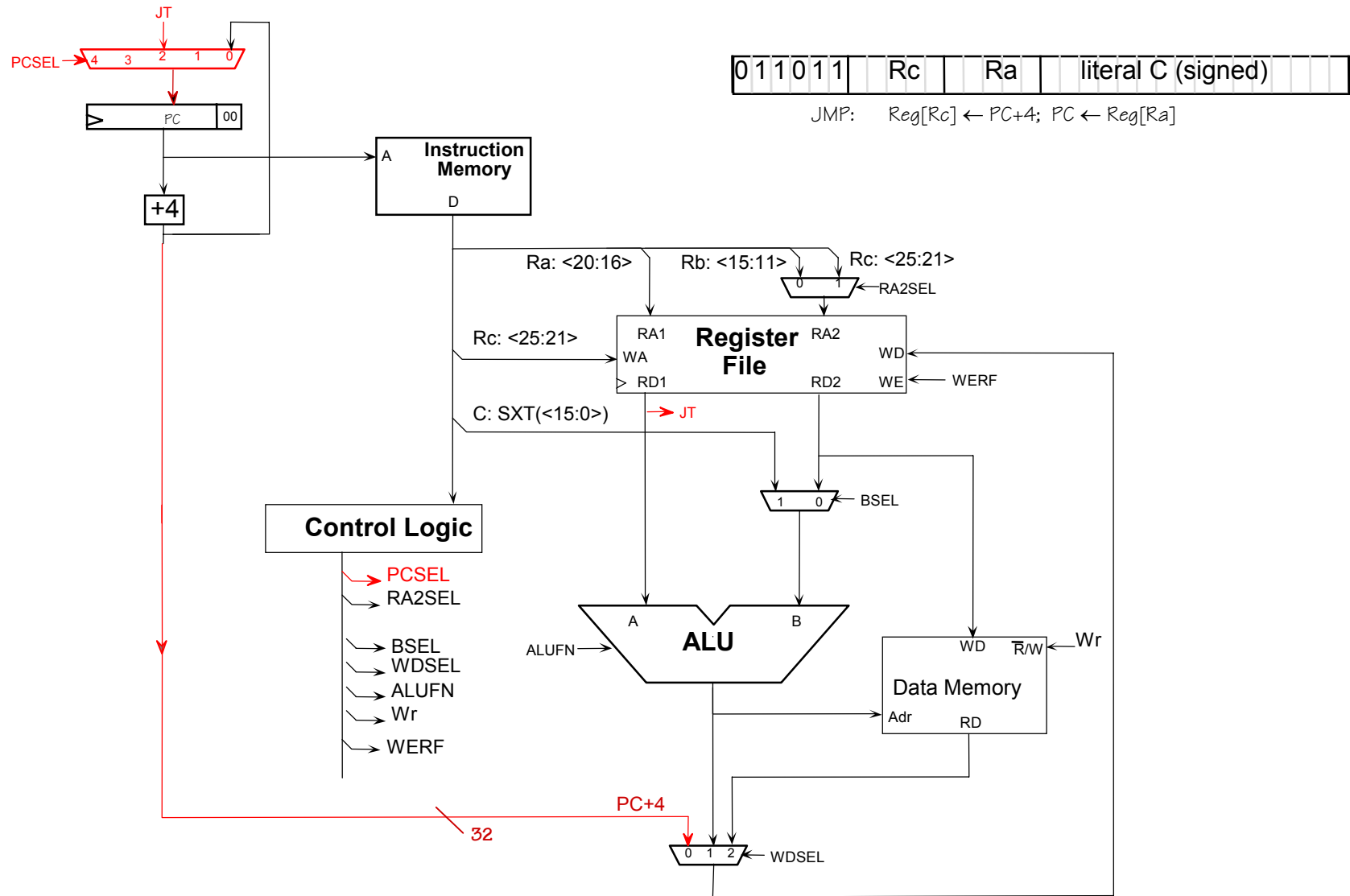
Load Instruction



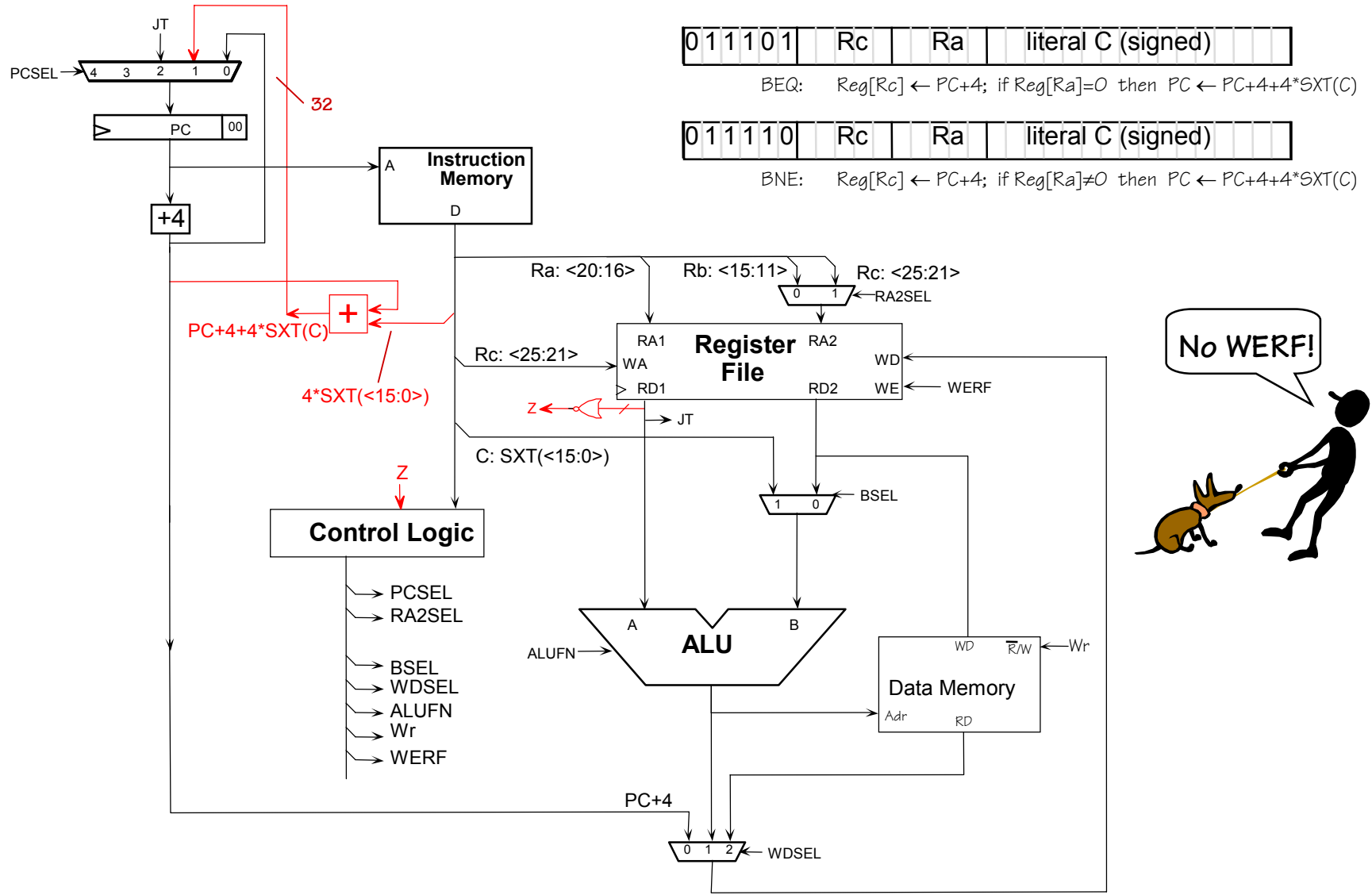
Store Instruction



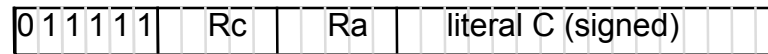
JMP Instruction



BEQ/BNE Instructions



Load Relative Instruction



LDR: $Reg[Rc] \leftarrow Mem[PC + 4 + 4 * SXT(C)]$

Hey, WAIT A MINUTE. 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.

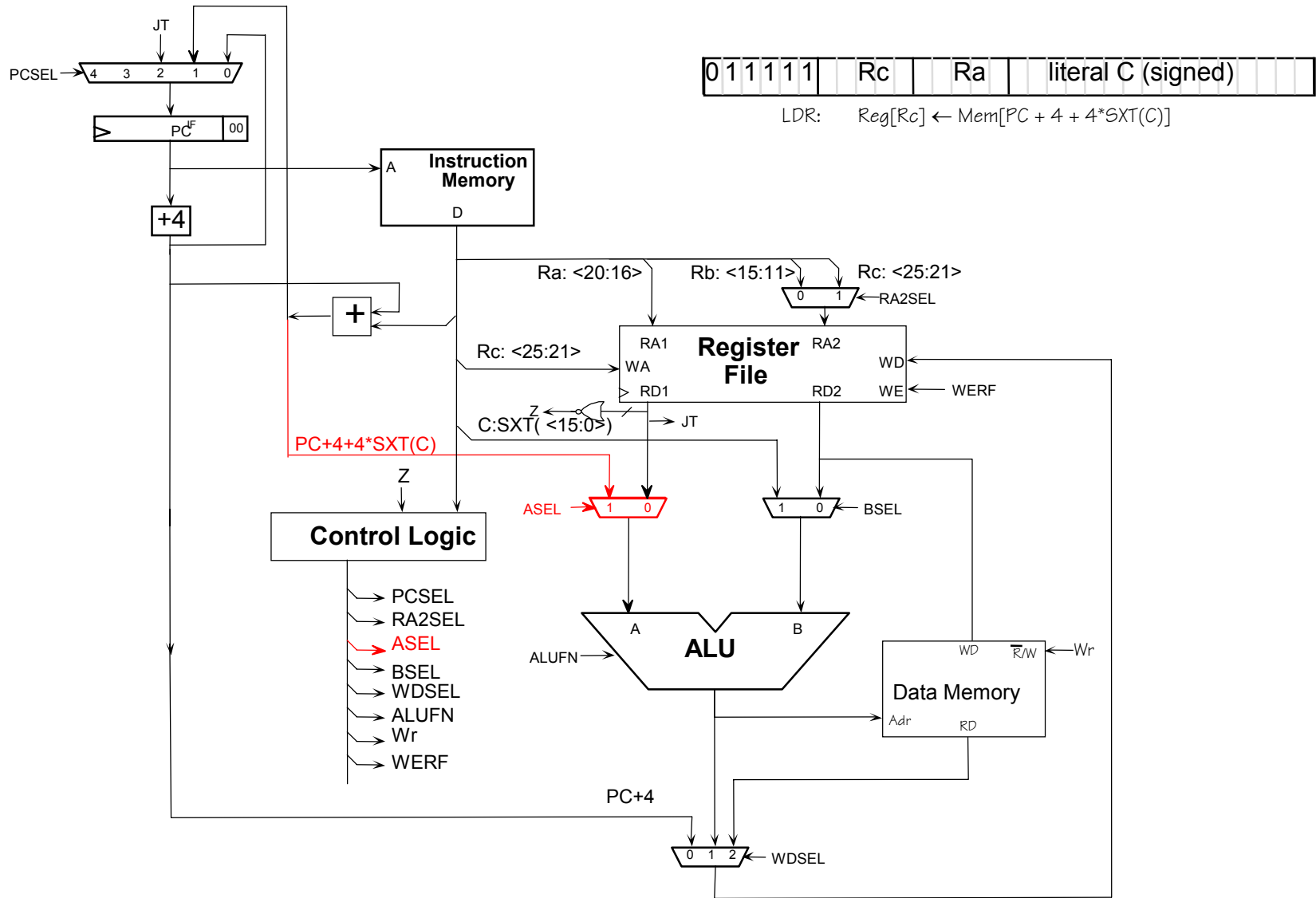
So why an instruction designed to load data that's “near” the instruction???

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



Exception Processing

We'd like **RECOVERABLE INTERRUPTS** for

- FAULTS (eg, Illegal Instruction)
 - CPU or SYSTEM generated [synchronous]
- TRAPS & system calls (eg, read-a-character)
 - CPU generated [synchronous]
- I/O events (eg, key struck)
 - externally generated [asynchronous]

GOAL: Interrupt running program, invoke exception handler (like a procedure call), return to continue execution.

KEY: TRANSPARENCY to interrupted program.

- Most difficult for asynchronous interrupts

Implementation...

IMPLEMENTATION:

- “FORCED” procedure call: new PC, save old PC for return.
- ... but WHERE to save old PC? Use LP?
- ... don't execute current instruction

APPROACH: Another reserved register, XP (aka R30).

Why can't interrupted program use XP?

Example: DIV unimplemented

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

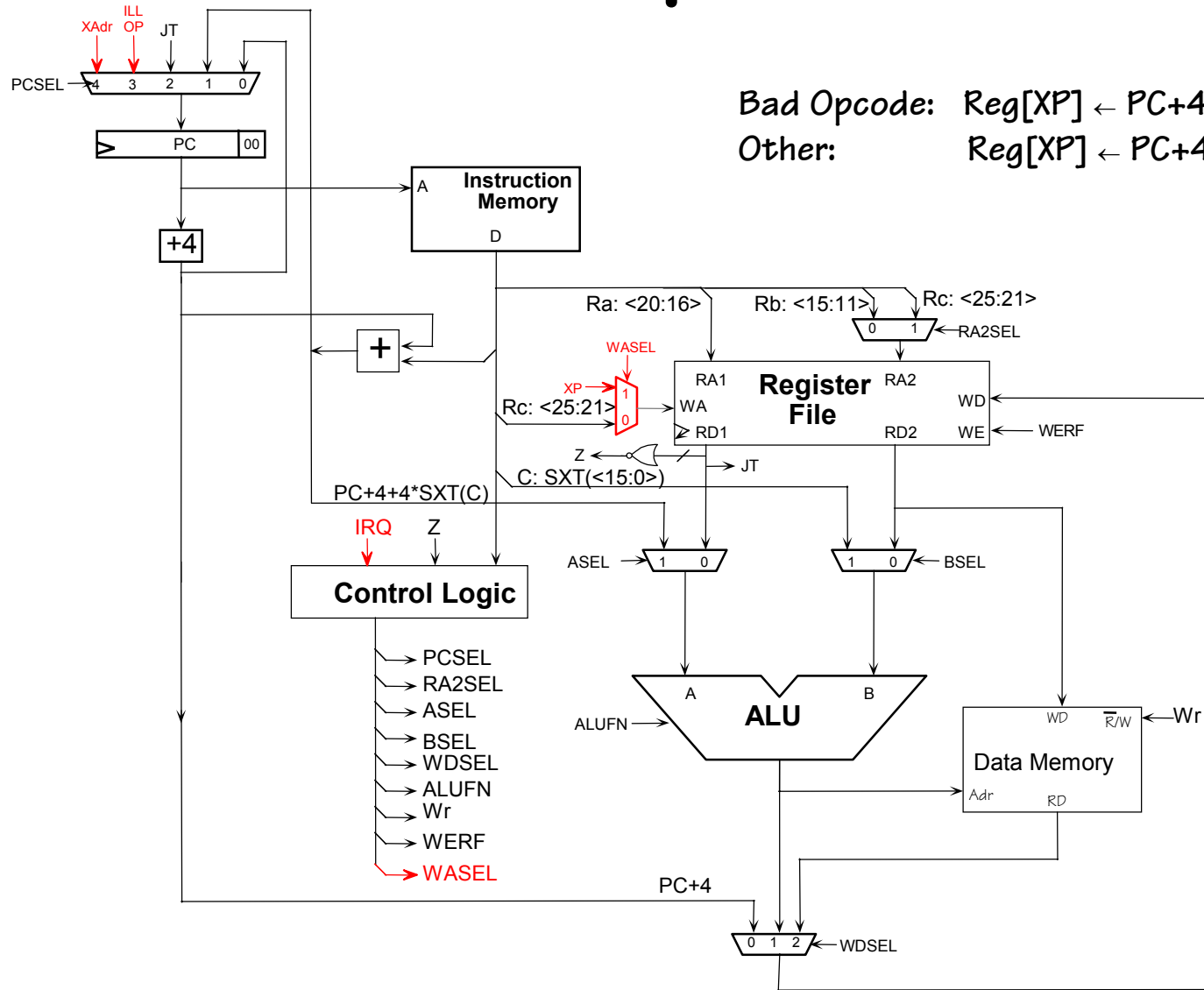
Forced by
hardware

IllOp:
PUSH (XP)

Fetch inst. at Mem[Reg[XP]-4]
check for DIV opcode, get reg numbers
perform operation in SW, fill result reg

POP (XP)
JMP (XP)

Exceptions



Bad Opcode: $Reg[XP] \leftarrow PC+4; PC \leftarrow \text{"IllOp"}$
 Other: $Reg[XP] \leftarrow PC+4; PC \leftarrow \text{"Xadr"}$

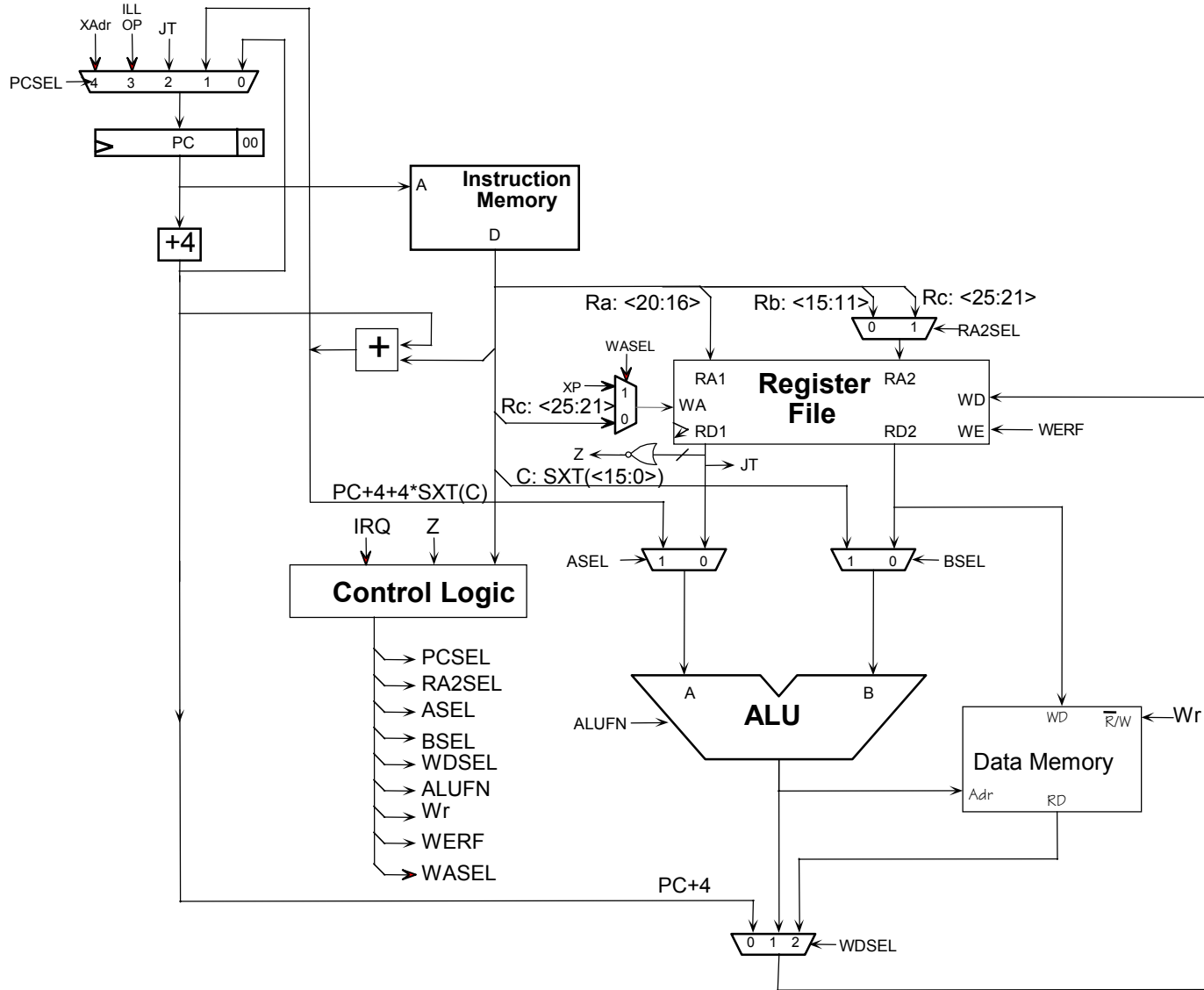
Control Logic

	<i>OP</i>	<i>OPC</i>	<i>LD</i>	<i>ST</i>	<i>JMP</i>	<i>BEQ</i>	<i>BNE</i>	<i>LDR</i>	<i>Illop</i>	<i>trap</i>
<i>ALUFN</i>	F(op)	F(op)	"+"	"+"	—	—	—	"A"	—	—
<i>WERF</i>	1	1	1	0	1	1	1	1	1	1
<i>BSEL</i>	0	1	1	1	—	—	—	—	—	—
<i>WDSEL</i>	1	1	2	—	0	0	0	2	0	0
<i>WR</i>	0	0	0	1	0	0	0	0	0	0
<i>RA2SEL</i>	0	—	—	1	—	—	—	—	—	—
<i>PCSEL</i>	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0	3	4
<i>ASEL</i>	0	0	0	0	—	—	—	1	—	—
<i>WASEL</i>	0	0	0	—	0	0	0	0	1	1

Implementation choices:

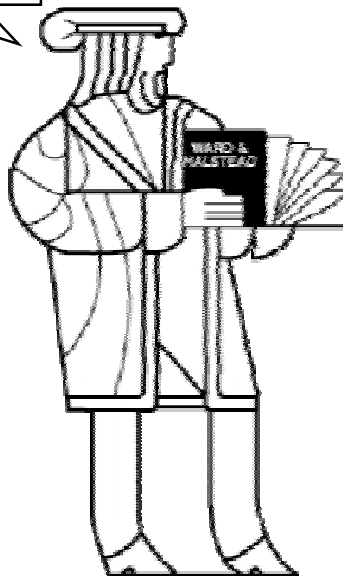
- ROM indexed by opcode, external branch & trap logic
- PLA
- "random" logic (eg, standard cell gates)

Beta: Our "Final Answer"



Next Time: Pipelined Betas

Hey, building a computer is not really all that difficult



So let's run down to the 6.004 lab and put Intel out of business!

