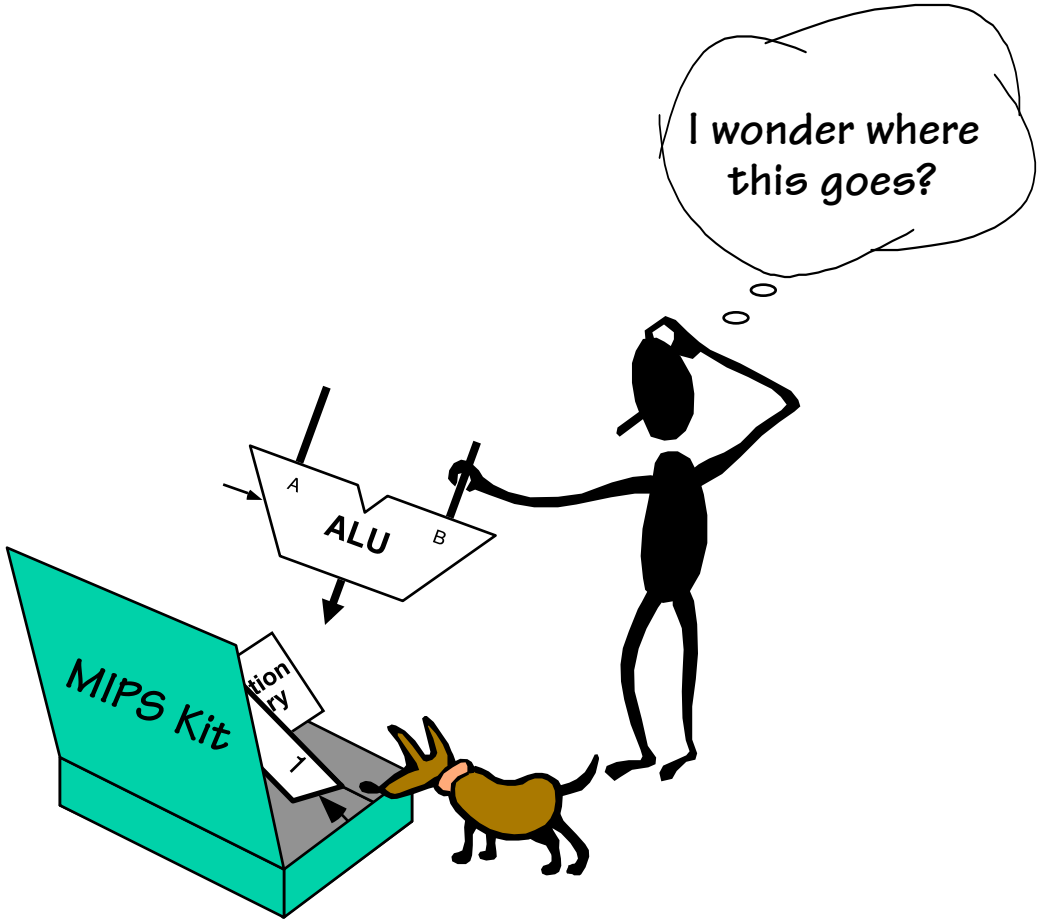
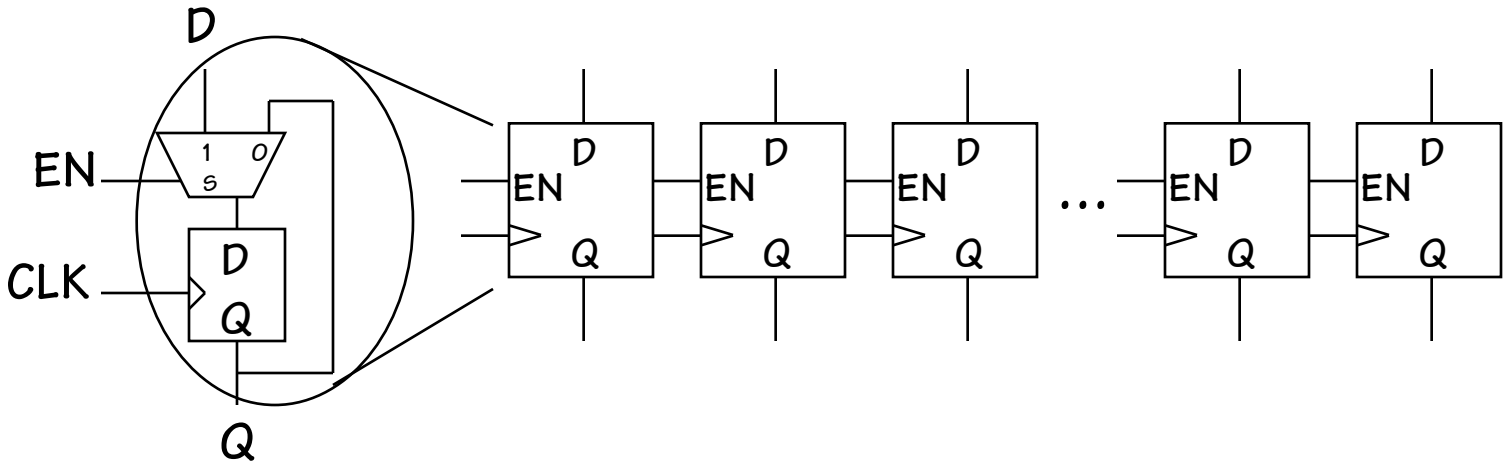
Building a Computer

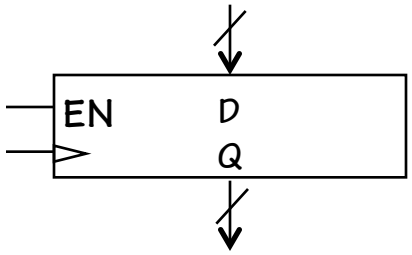


One More Functional Unit

Thus far, our processing blocks units have focused on logical and arithmetic functions. We'll also need functional units for storing intermediate results. By now, we are used to the notion of building wide registers.

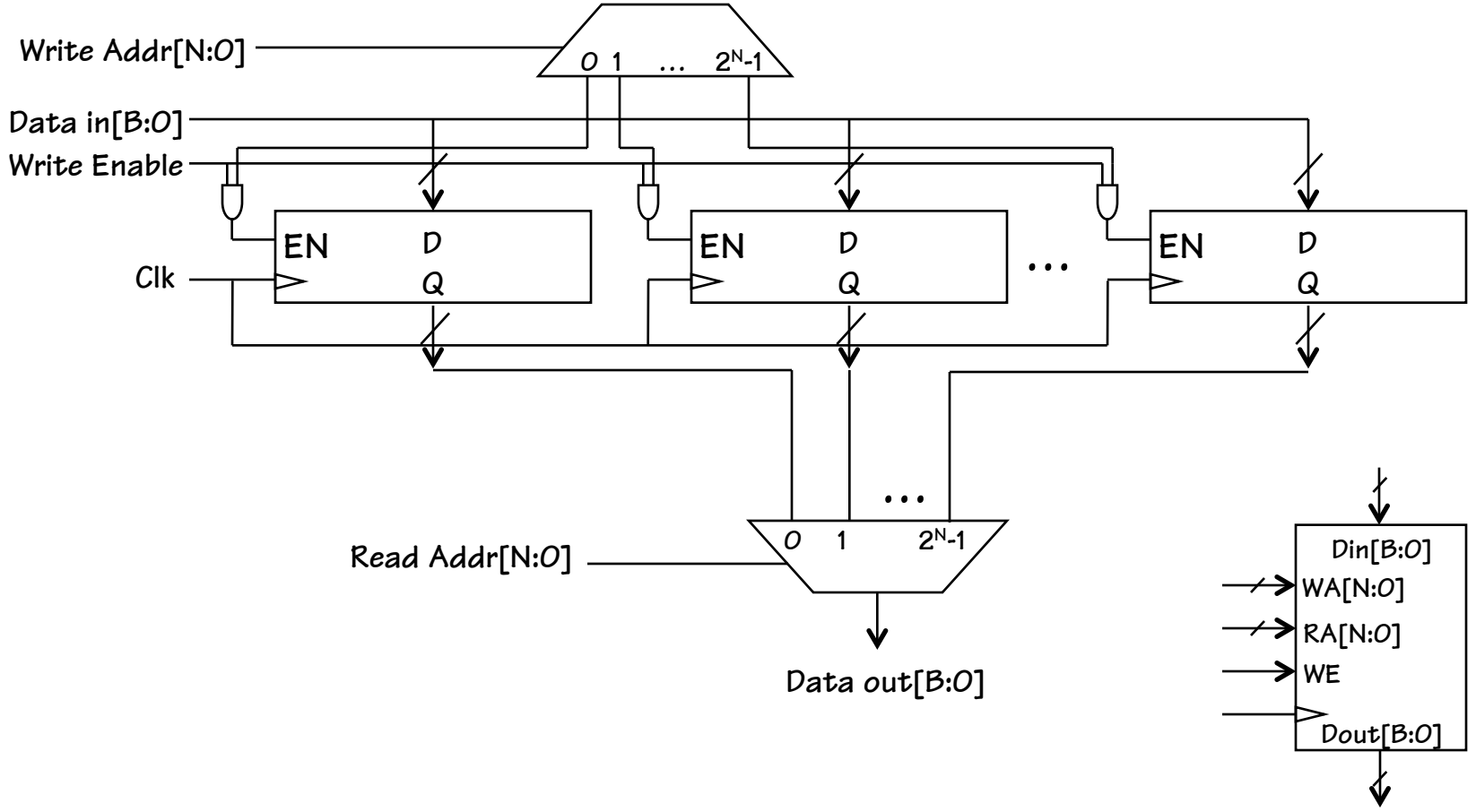


An N-bit wide
Register with enable



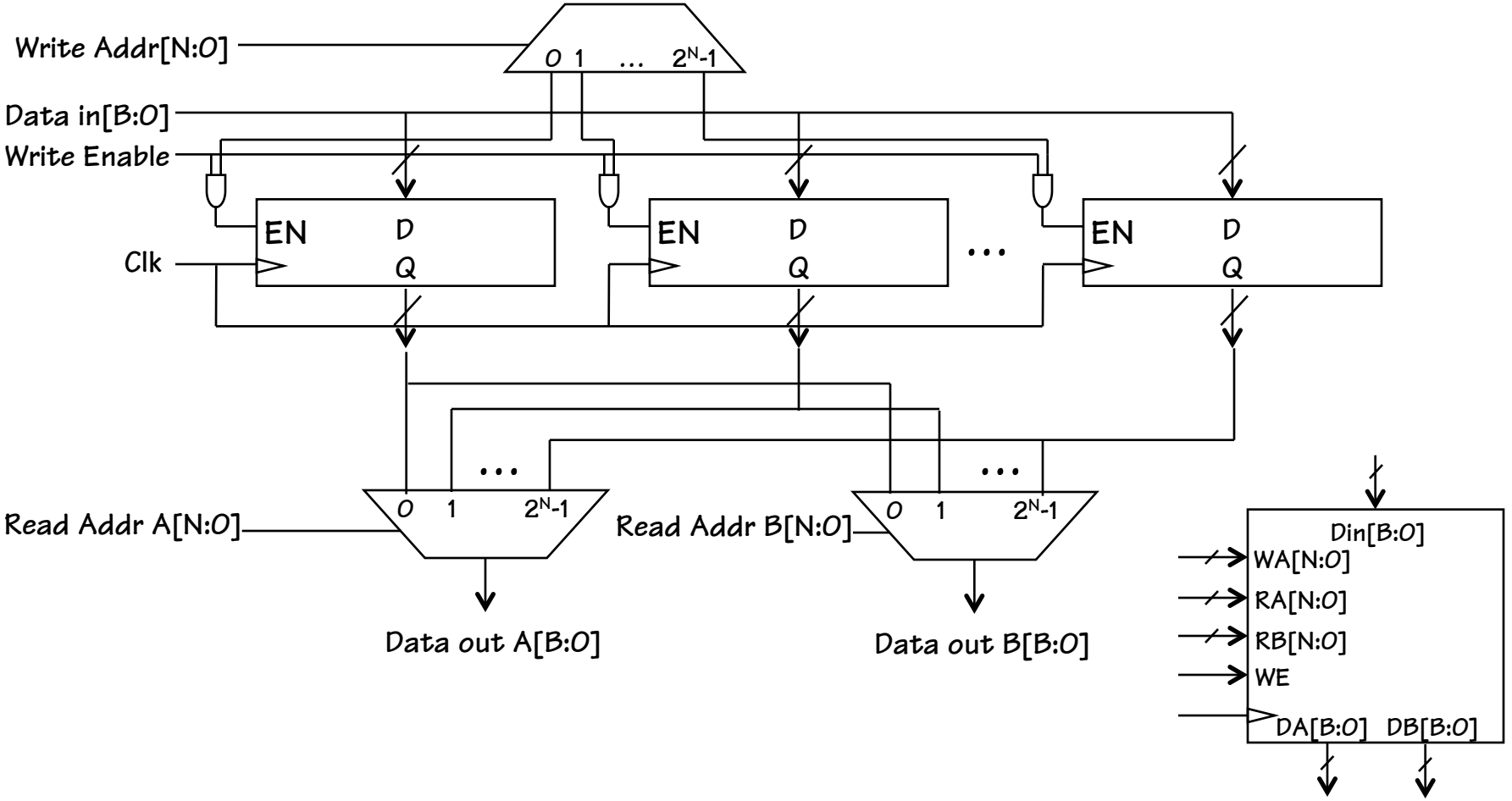
A Register File

We can also construct an addressable array of registers



A Multi-Ported Register File

Multiple read ports by simply adding more output MUXs



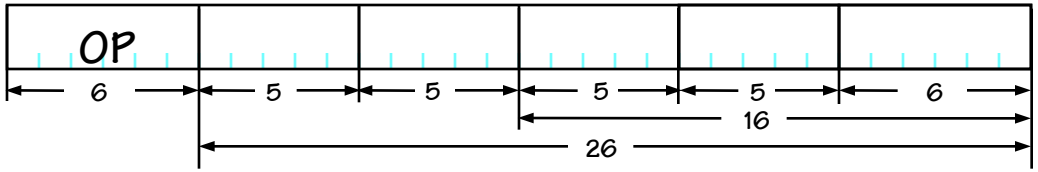
THIS IS IT!

"Motivating Force"
or
"Inciting Incident"

This is the point in the course where the PLOT actually begins. We are now ready to build a computer.

The ingredients are all in place, now it is time to build a legitimate computer. One that executes instructions, much the way any desktop, tablet, or other computer does.

The MIPS ISA



OP	r_s	r_t	r_d	shamt	func
000000					

R-type: ALU with Reguster operands
Reg[rd] ← Reg[rs] op Reg[rt]

OP	r_s	r_t	immediate
001XXX			

I-type: ALU with constant operand
Reg[rt] ← Reg[rs] op SEXT(immediate)

OP	r_s	r_t	immediate
10X011			

I-type: Load and Store
Reg[rt] ← Mem[Reg[rs] + SEXT(immediate)]
Mem[Reg[rs] + SEXT(immediate)] ← Reg[rt]

OP	r_s	r_t	immediate
10X011			

I-type: Branch Instructions
if (Reg[rs] == Reg[rt]) PC ← PC + 4 + 4*SEXT(immediate)
if (Reg[rs] != Reg[rt]) PC ← PC + 4 + 4*SEXT(immediate)

OP	26-bit constant
00001X	

J-type: jump
PC ← (PC & 0xf0000000) | 4*(immediate)

- The MIPS instruction set as seen from a Hardware Perspective

Instruction classes distinguished by types:
1) 3-operand ALU
2) ALU w/immediate
3) Loads/Stores
4) Branches
5) Jumps

Design Approach

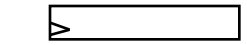
Incremental Featurism

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).

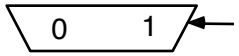
Steps:

1. 3-Operand ALU instructions
2. ALU w/immediate instructions
2. Load & Store Instructions
3. Jump & Branch instructions
4. Exceptions
5. Merge data paths

Our Bag of Components:

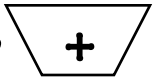


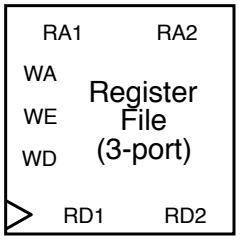
Registers

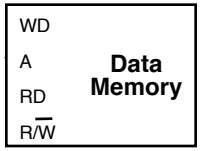


Muxes

ALU & adders



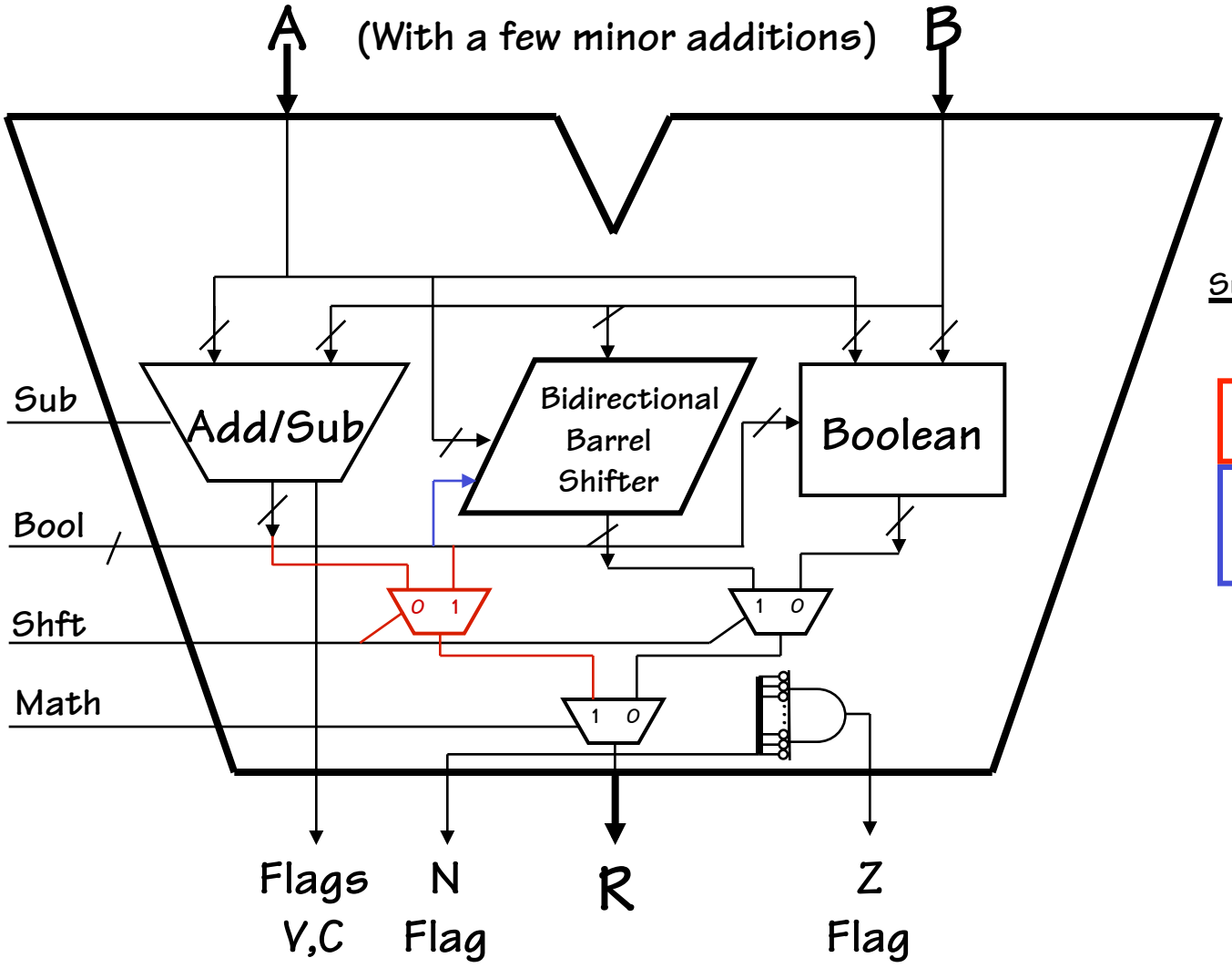




Memories

A Few ALU Tweaks

Let's review the ALU that we built a few lectures ago.

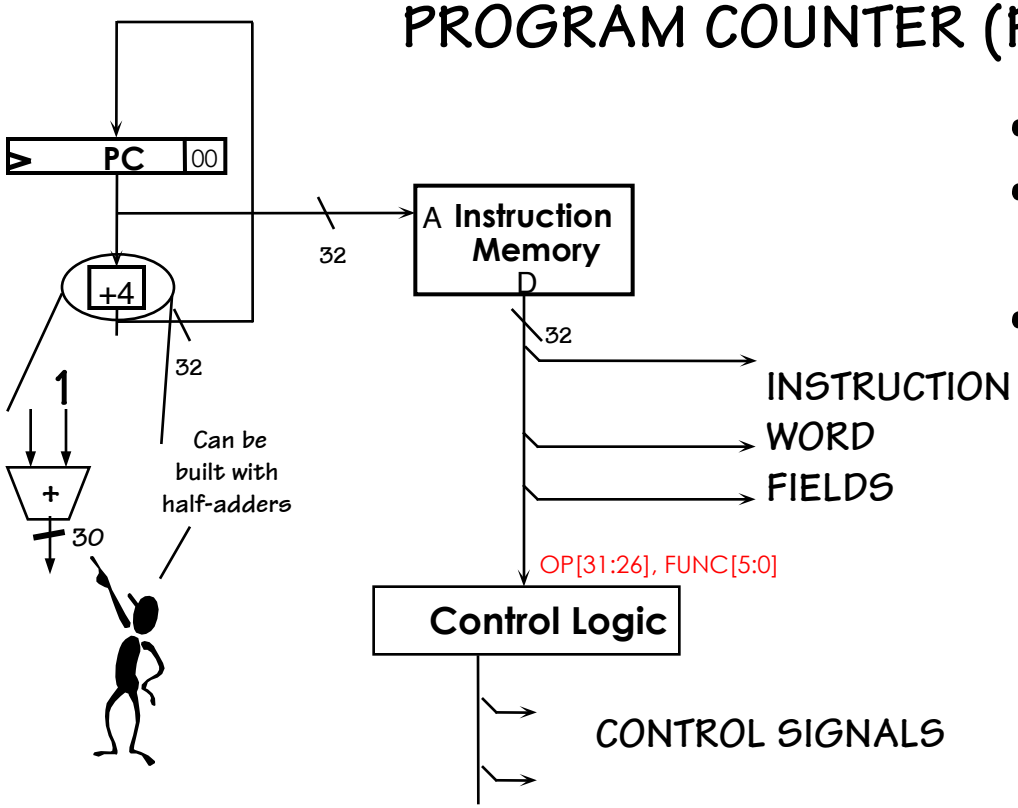


5-bit ALUFN

Sub	Bool	Shft	Math	OP
0	XX	0	1	A+B
1	XX	0	1	A-B
X	X0	1	1	0
X	X1	1	1	1
X	00	1	0	B<<A
X	10	1	0	B>>A
X	11	1	0	B>>>A
X	00	0	0	A & B
X	01	0	0	A | B
X	10	0	0	A ^ B
X	11	0	0	$\overline{A \mid B}$

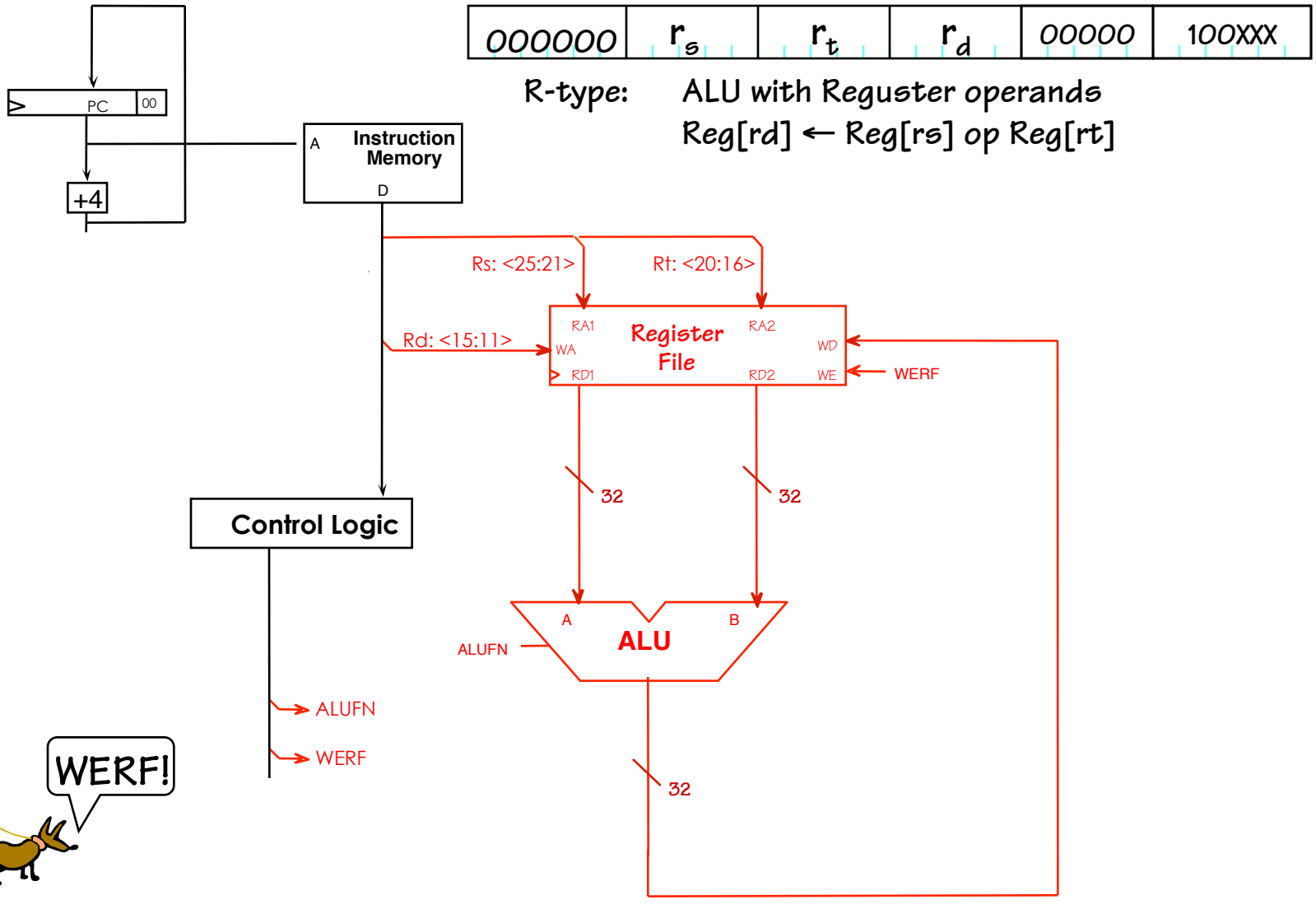
Instruction Fetch/Decode

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

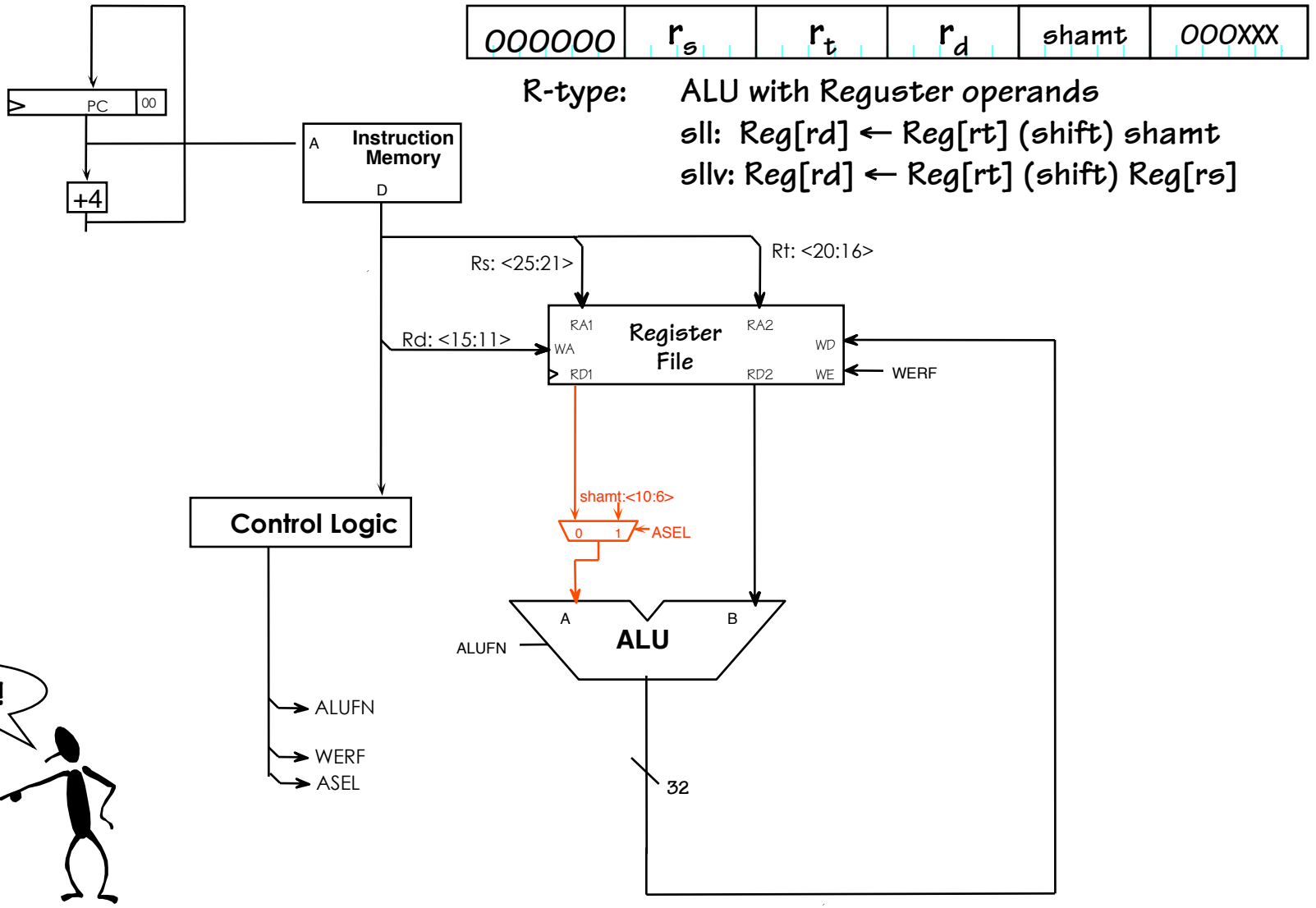


- use PC as memory address
- add 4 to PC, load new value at end of cycle
- fetch instruction from memory
  - use some instruction fields directly (register numbers, 16-bit constant)
  - use bits <31:26> and <5:0> to generate controls

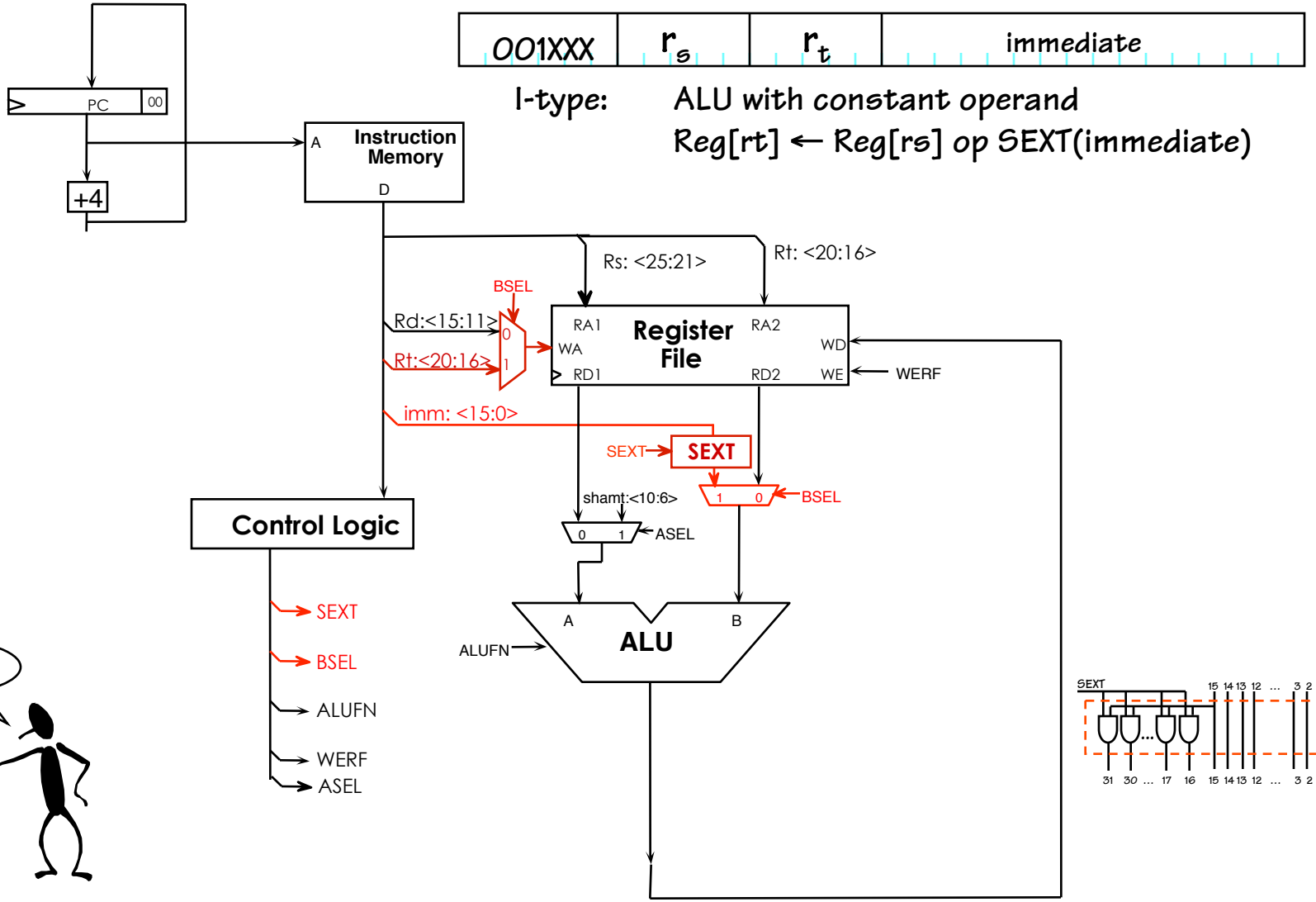
3-Operand ALU Data Path



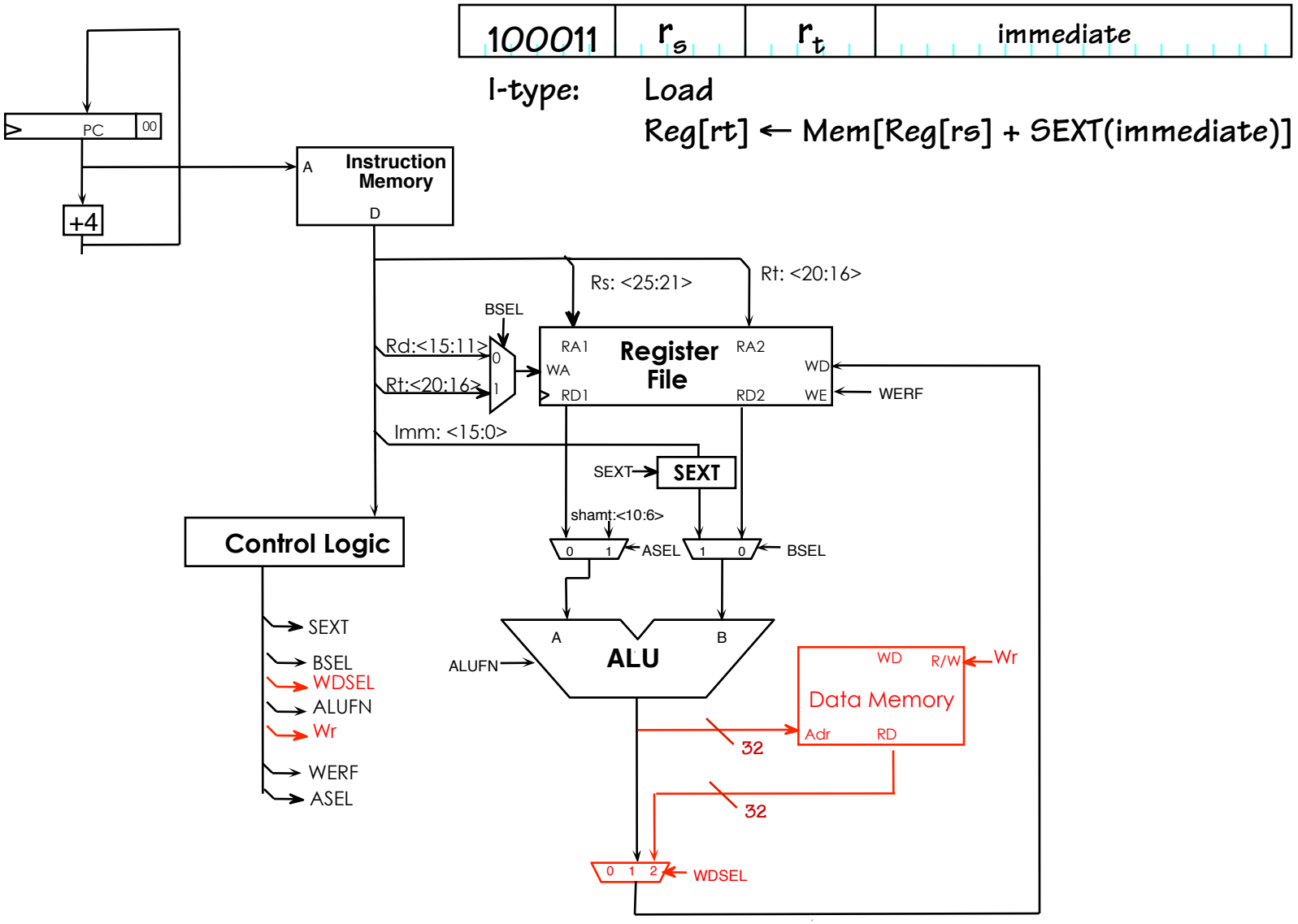
Shift Instructions



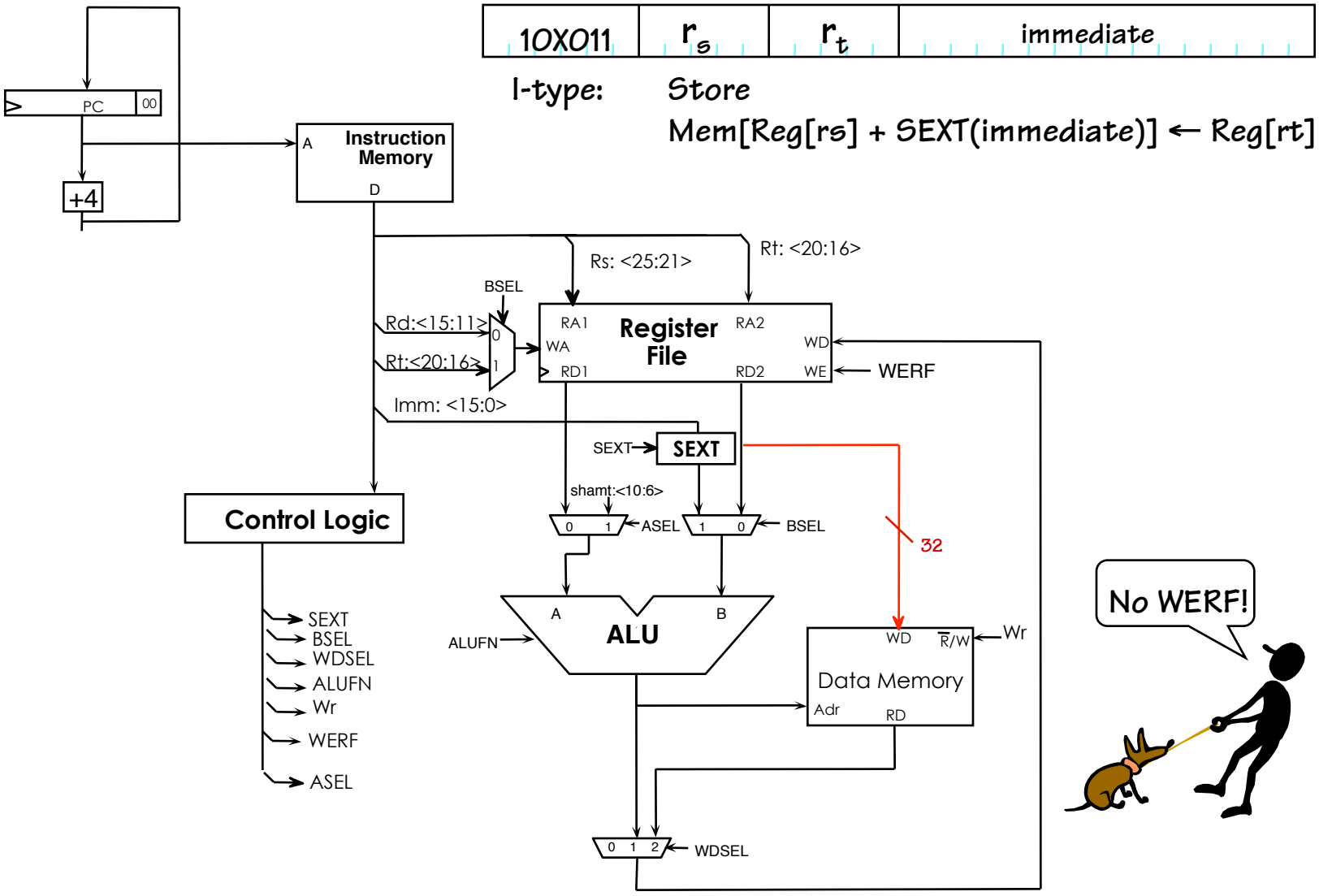
ALU with Immediate



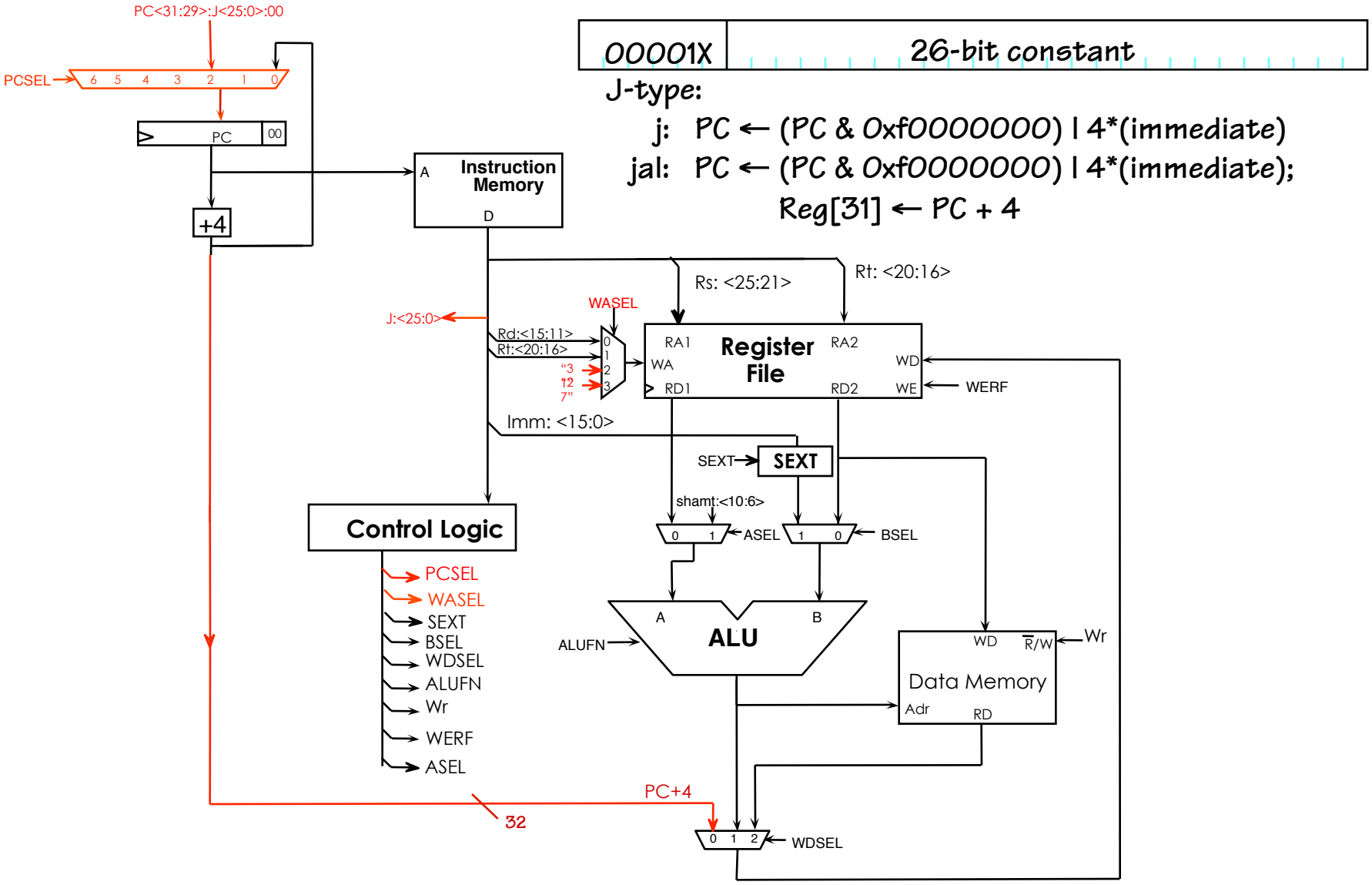
Load Instruction



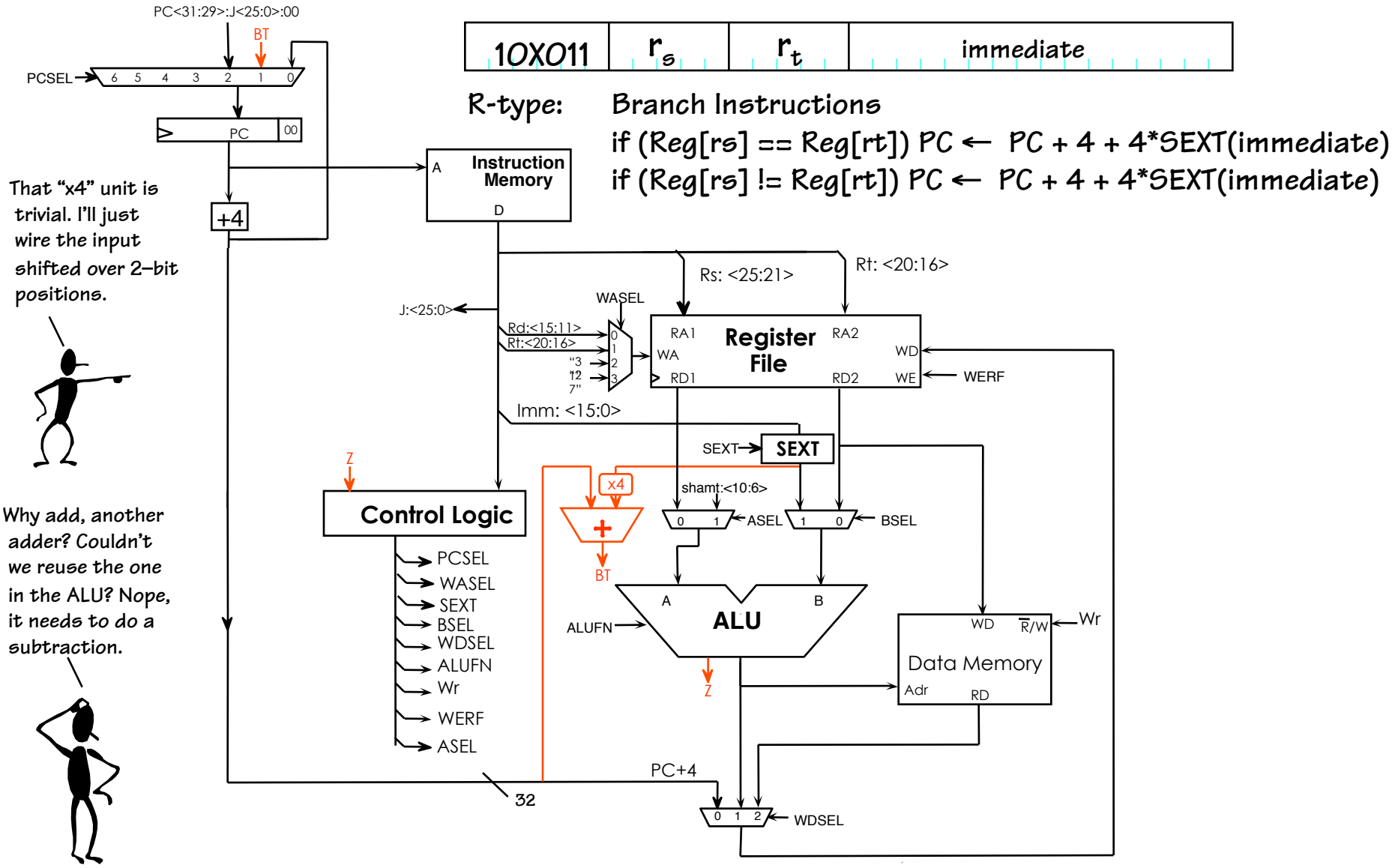
Store Instruction



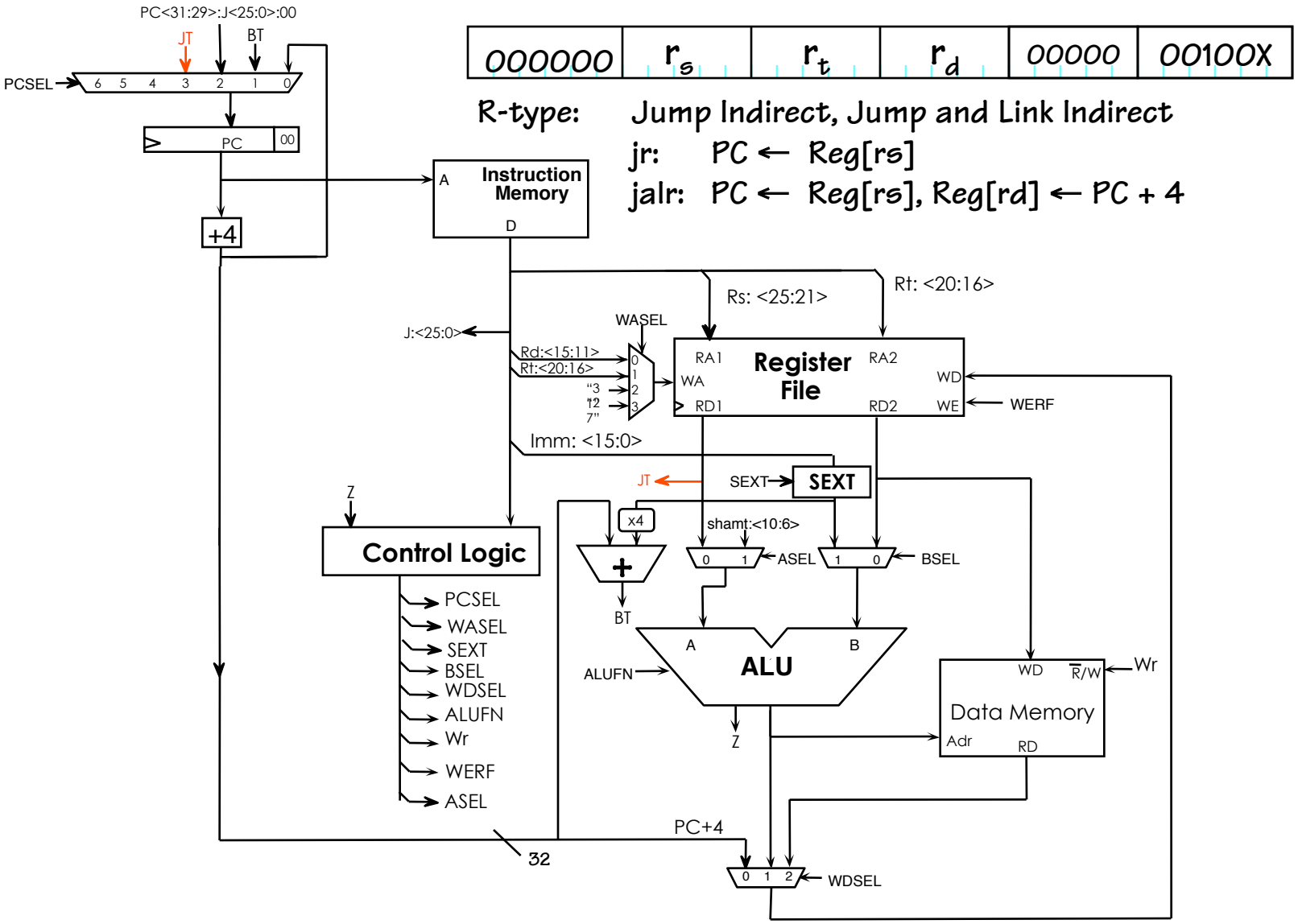
JMP Instructions



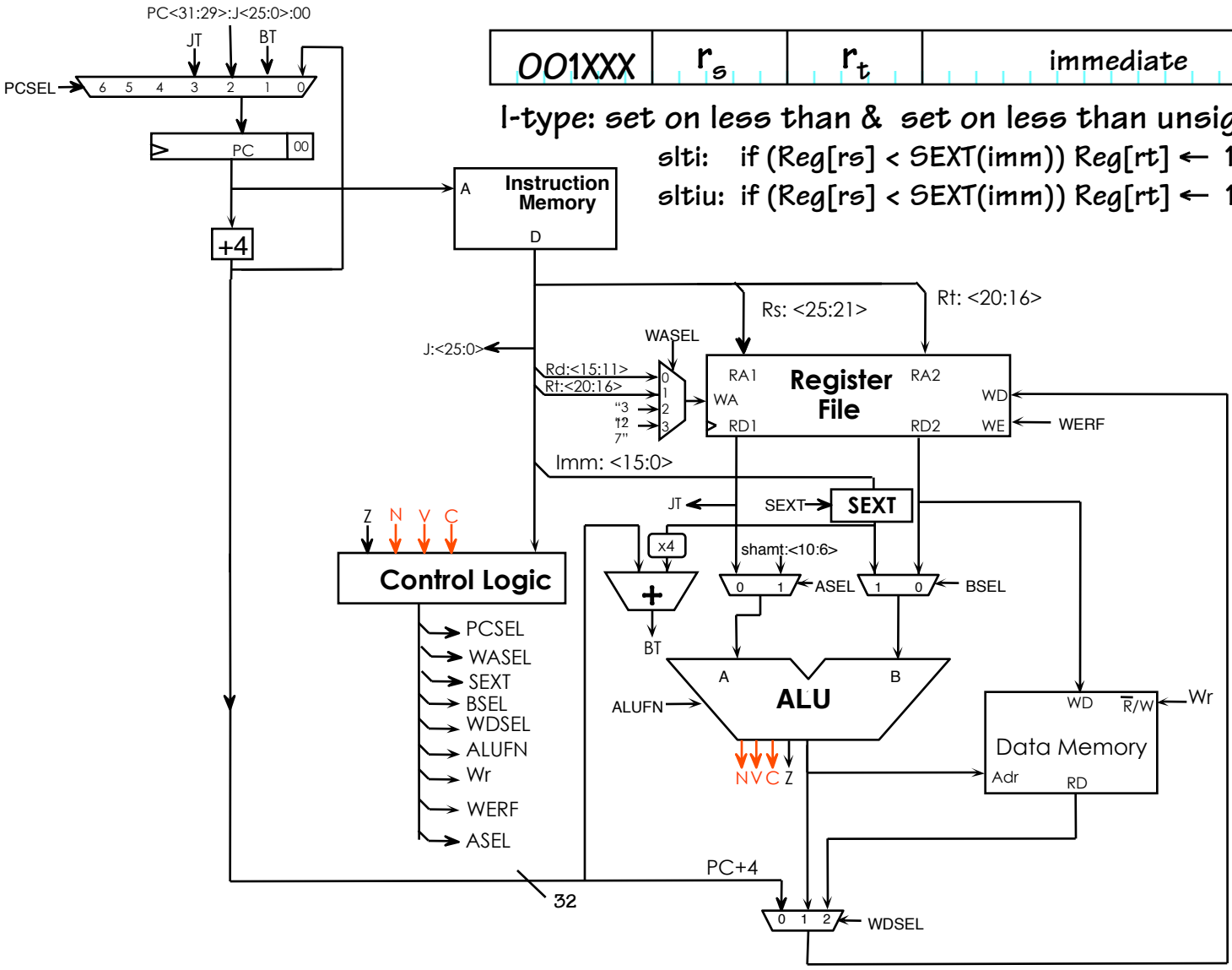
BEQ/BNE Instructions



Jump Indirect Instructions



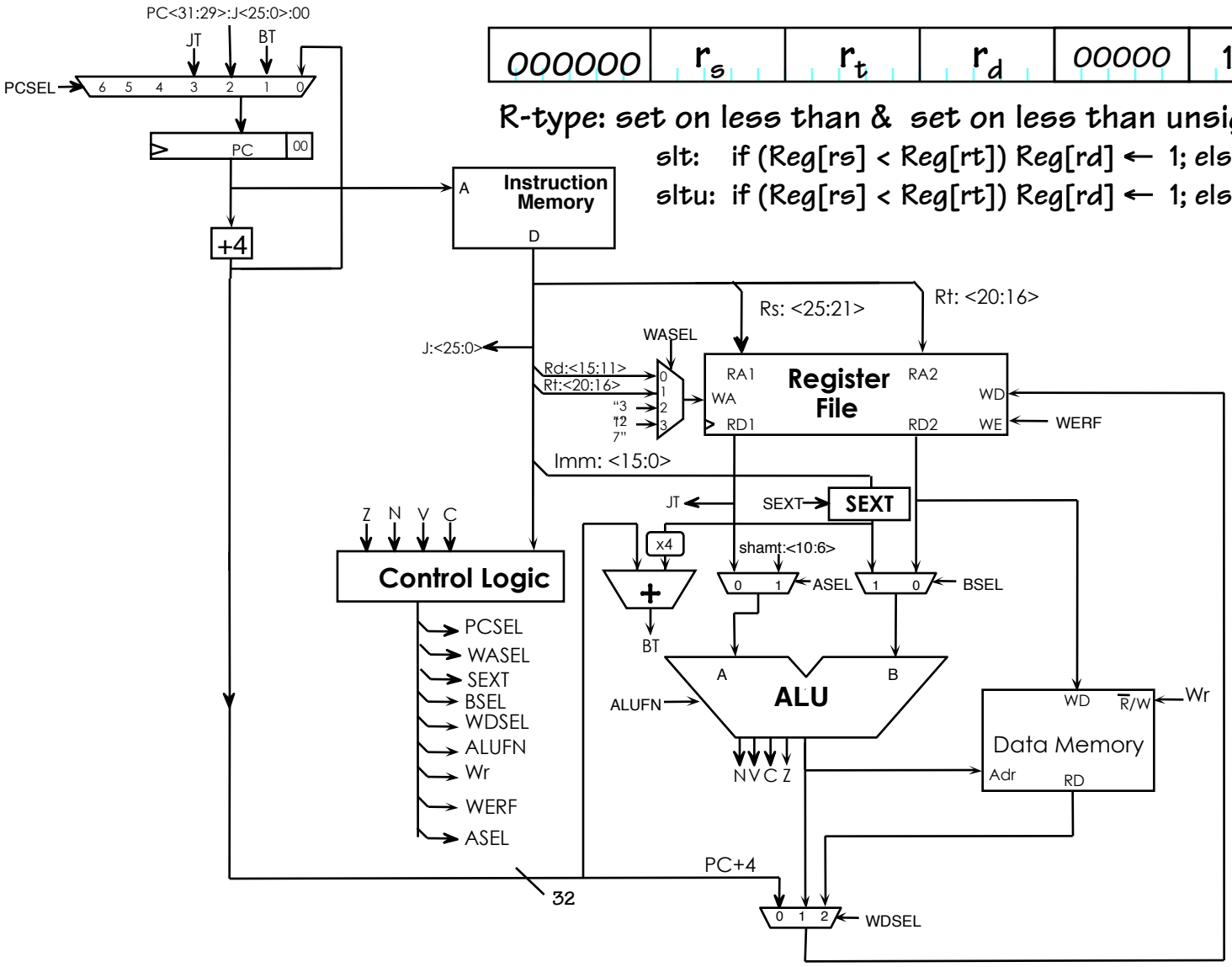
Loose Ends



Reminder:
To evaluate (A < B) we first compute A-B and look at the flags.

LT = N ⊕ V
LTU = C

More Loose Ends



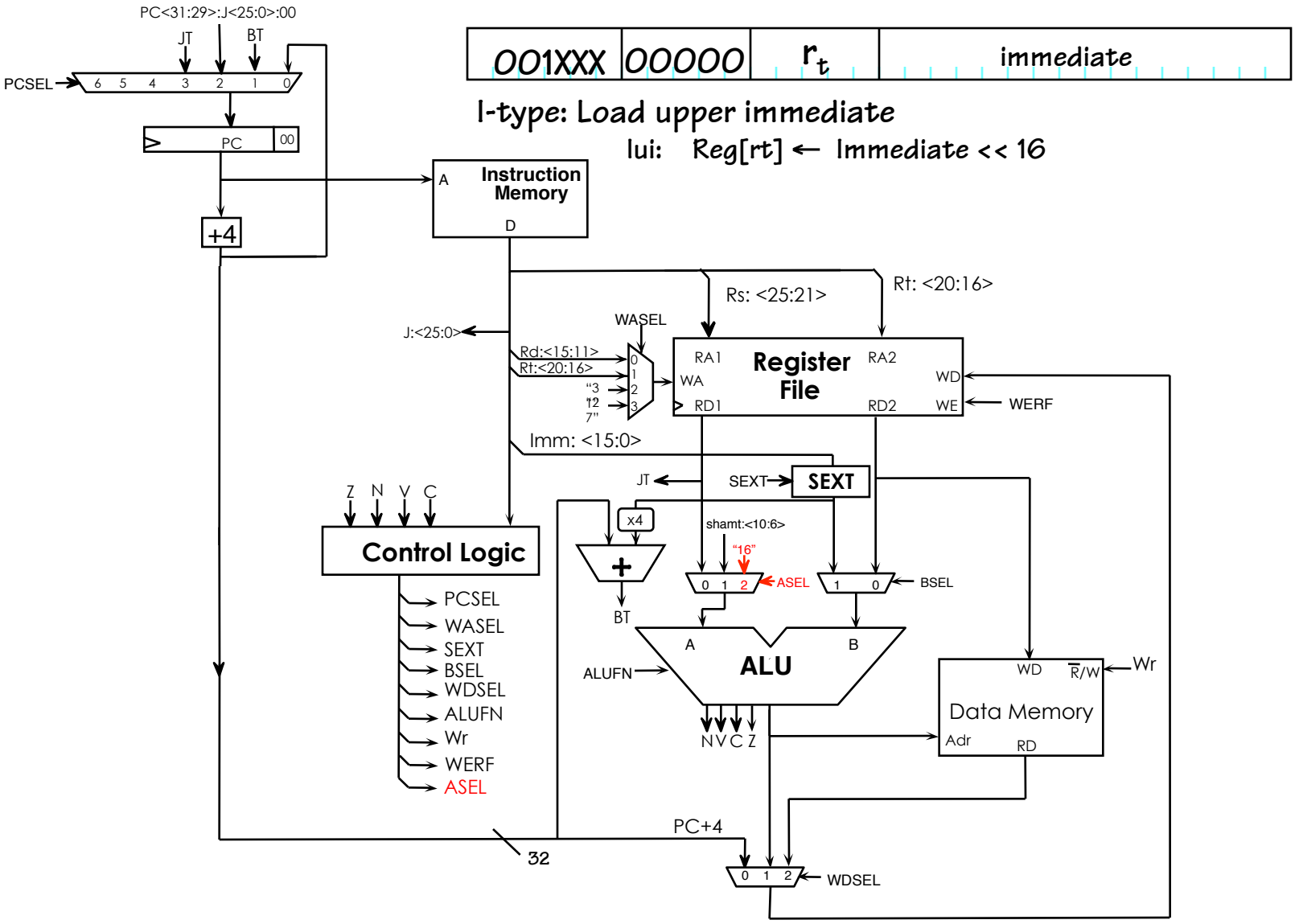
000000	r_s	r_t	r_d	00000	10101X

R-type: set on less than & set on less than unsigned

slt: if (Reg[rs] < Reg[rt]) Reg[rd] ← 1; else Reg[rd] ← 0

sltu: if (Reg[rs] < Reg[rt]) Reg[rd] ← 1; else Reg[rd] ← 0

LUI Ends


PC<31:29>:J<25:0>:00
JT
BT
PCSEL
6 5 4 3 2 1 0
PC
00
+4
A
Instruction Memory
D
001XXX
00000
r_t
immediate
I-type: Load upper immediate
lui: Reg[rt] ← Immediate << 16
Rs: <25:21>
Rt: <20:16>
J:<25:0>
WASEL
Rd:<15:11>
Rt:<20:16>
"31"
"7"
RA1
RA2
WA
WD
RD1
RD2
WE
Register File
WERF
Imm: <15:0>
JT
SEXT
SEXT
Z N V C
Control Logic
x4
shamt:<10:6>
"16"
ASEL
BSEL
BT
A
B
ALU
ALUFN
N V C Z
WD
R/W
Wr
Data Memory
Adr
RD
PCSEL
WASEL
SEXT
BSEL
WDSEL
ALUFN
Wr
WERF
ASEL
PC+4
32
WDSEL

Reset, Interrupts, and Exceptions

FIRST, we need some way to get our machine into a known initial state. This doesn't mean that all registers will be initialized, just that we'll know where to fetch the first instruction. We'll call this control input, RESET

We'd also like RECOVERABLE INTERRUPTS for

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

These are "Software" notions of synchrony and asynchrony.

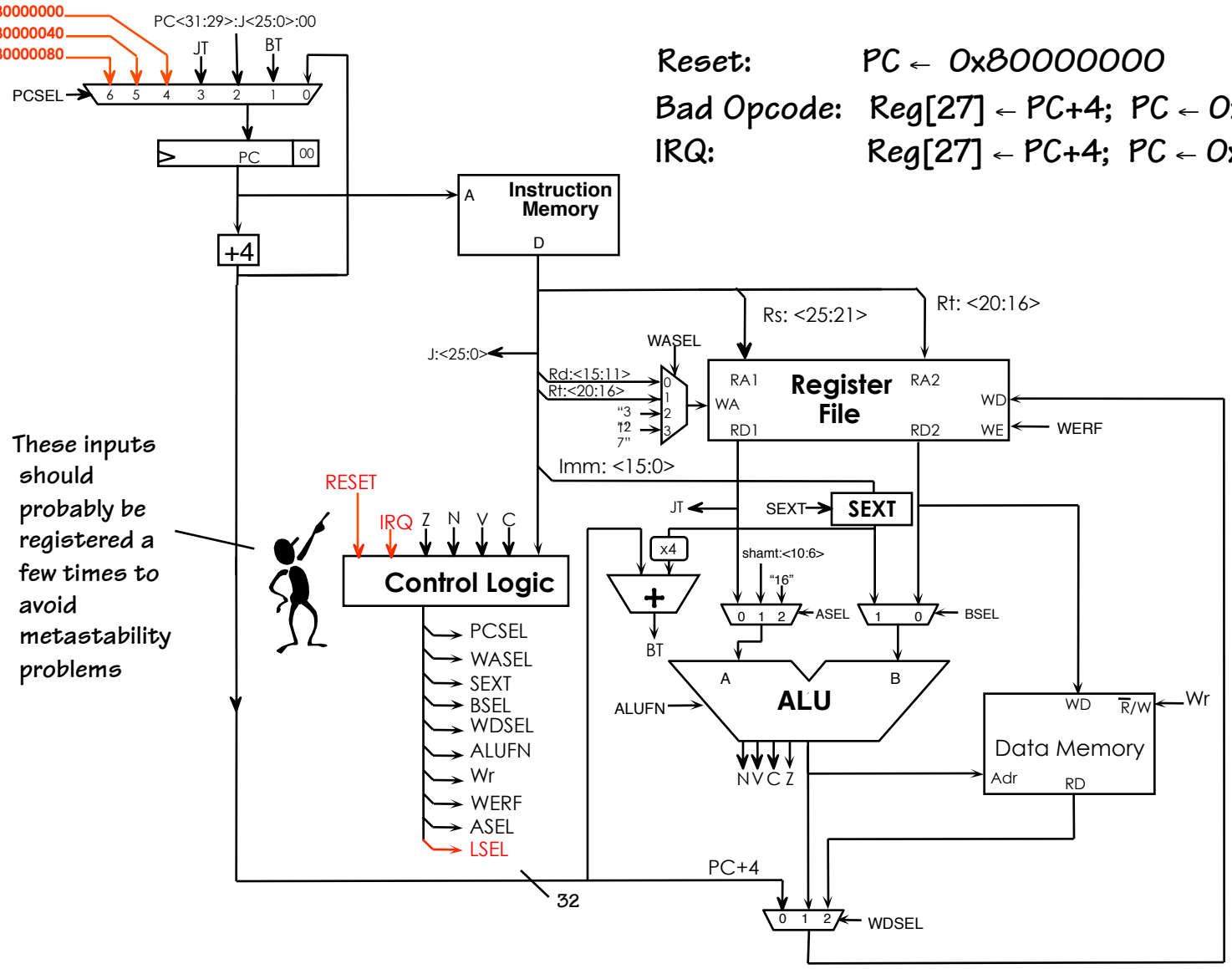
EXCEPTION GOAL: Interrupt running program, invoke exception handler, return to continue execution.

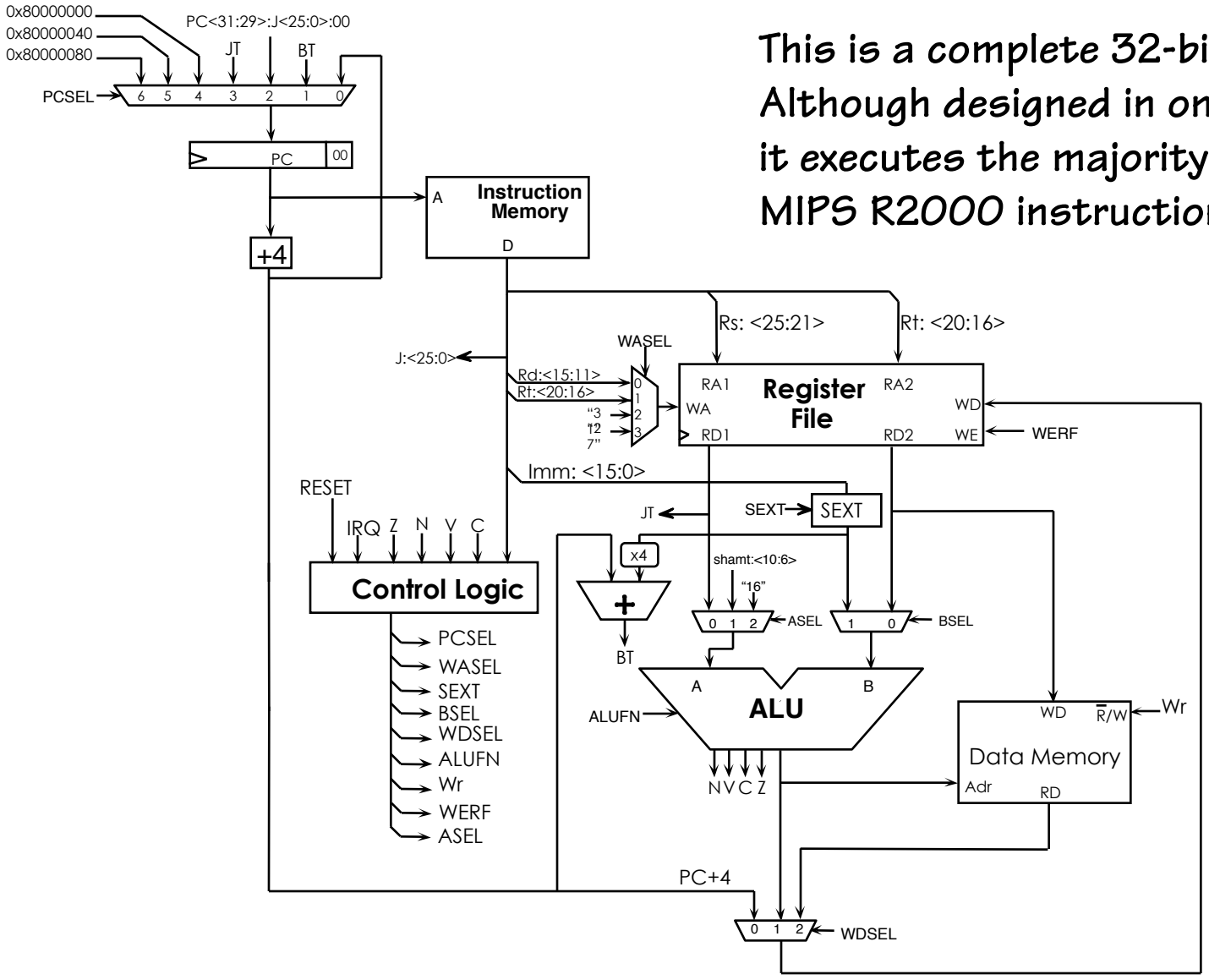
Exceptions

Reset: PC ← 0x80000000

Bad Opcode: Reg[27] ← PC+4; PC ← 0x80000040

IRQ: Reg[27] ← PC+4; PC ← 0x80000080





This is a complete 32-bit processor. Although designed in one class lecture, it executes the majority of the MIPS R2000 instruction set.

- Executes one instruction per clock
- All that's left is the control logic design

MIPS Control

The control unit can be implemented using a ROM

Instruction	RESET	IRQ	Z	N	V	C	PCSEL	SEXT	WASEL	WDSEL	ALUFN				WR	WERF	ASEL	BSEL
											Sub	Bool	Shft	Math				
X	1	X	X	X	X	X	4	0	0	0	0	00	0	0	0	0	0	0
X	0	1	X	X	X	X	6	0	3	0	0	00	0	0	0	0	0	0
add	0	0	X	X	X	X	0	0	0	1	0	00	0	1	0	1	0	0
sll																		
andi																		
lw																		
sw																		
beq																		
j																		
lui																		

UNC miniMIPS