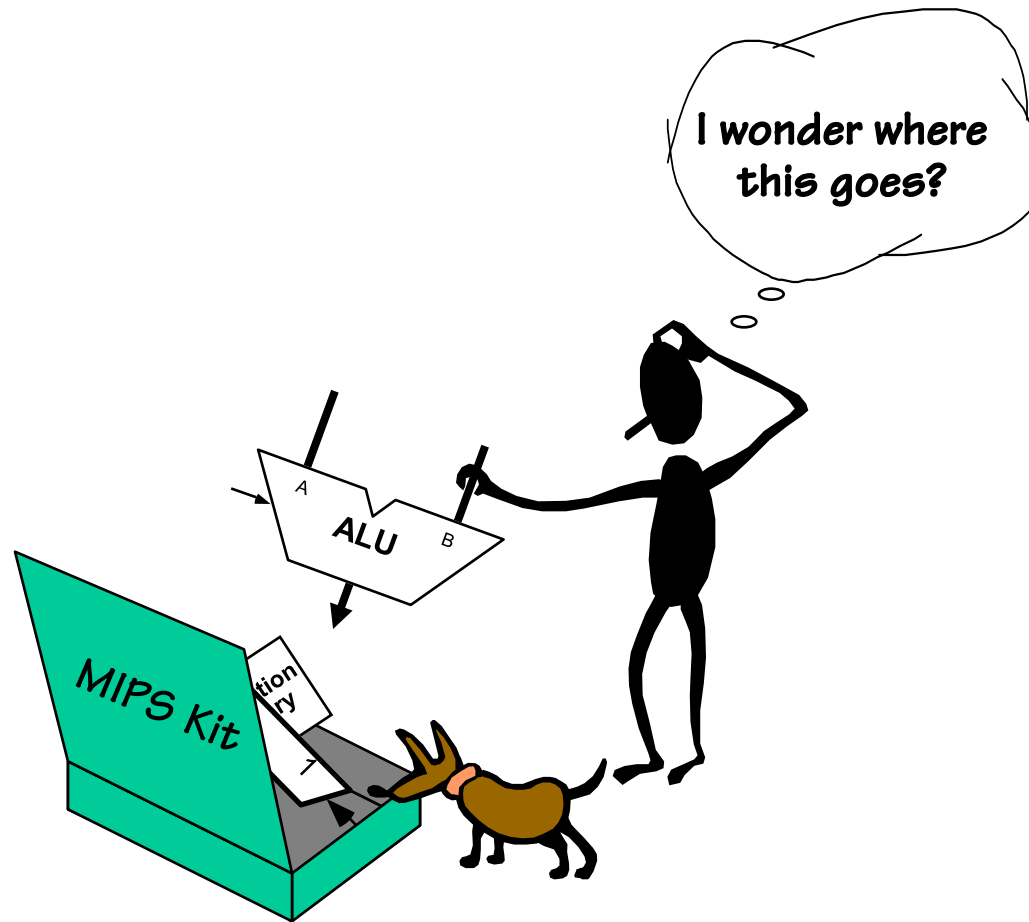


Building a Computer

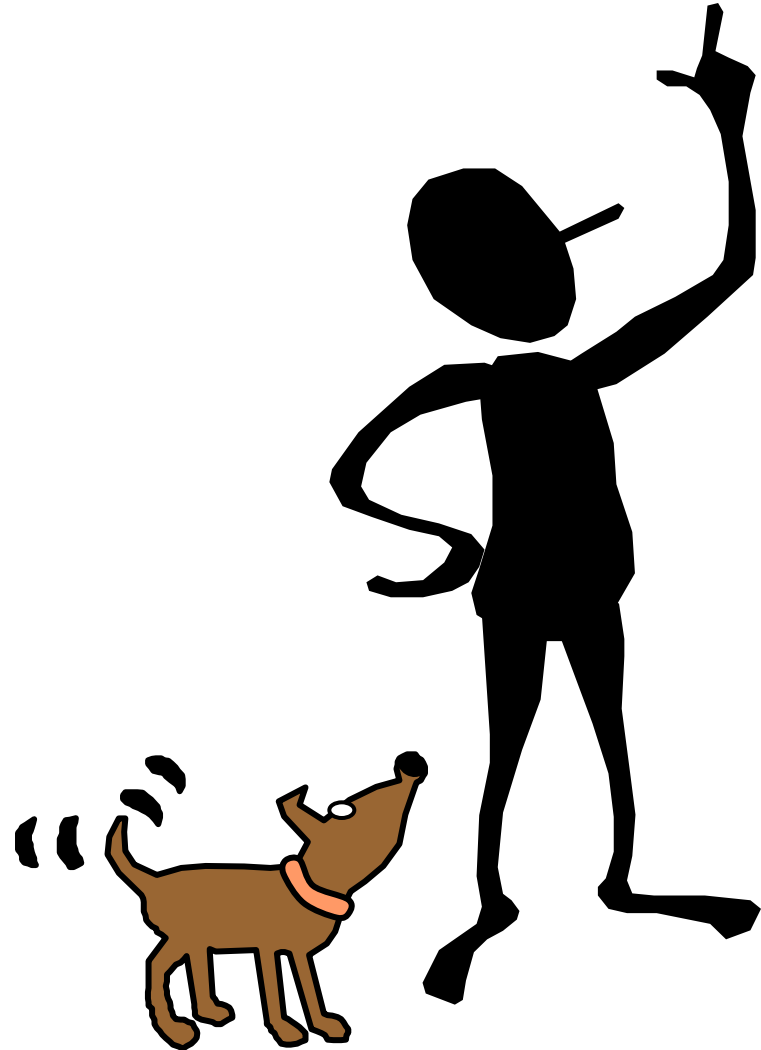


THIS IS IT!

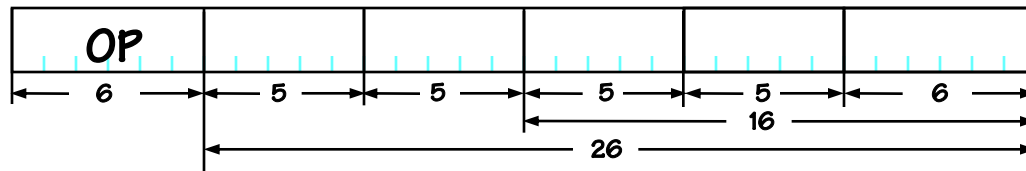
“Motivating Force” or “Inciting Incident”

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 other desktop, PDA, or other computer does.



Review: The MIPS ISA



000000	r_s	r_t	r_d	shamt	func
--------	-------	-------	-------	-------	------

R-type: ALU with Register operands
 $\text{Reg}[rd] \leftarrow \text{Reg}[rs] \text{ op } \text{Reg}[rt]$

001XXX	r_s	r_t	immediate
--------	-------	-------	-----------

I-type: ALU with constant operand
 $\text{Reg}[rt] \leftarrow \text{Reg}[rs] \text{ op } \text{SEXT}(\text{immediate})$

10XXXX	r_s	r_t	immediate
--------	-------	-------	-----------

I-type: Load and Store
 $\text{Reg}[rt] \leftarrow \text{Mem}[\text{Reg}[rs] + \text{SEXT}(\text{immediate})]$
 $\text{Mem}[\text{Reg}[rs] + \text{SEXT}(\text{immediate})] \leftarrow \text{Reg}[rt]$

0001XX	r_s	r_t	immediate
--------	-------	-------	-----------

I-type: Branch Instructions
 if ($\text{Reg}[rs] == \text{Reg}[rt]$) $\text{PC} \leftarrow \text{PC} + 4 + 4 * \text{SEXT}(\text{immediate})$
 if ($\text{Reg}[rs] != \text{Reg}[rt]$) $\text{PC} \leftarrow \text{PC} + 4 + 4 * \text{SEXT}(\text{immediate})$

00001X	26-bit constant
--------	-----------------

J-type: jump
 $\text{PC} \leftarrow (\text{PC} \& \text{0xf0000000}) \mid 4 * (\text{immediate})$

•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).

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

Design Approach

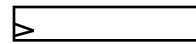
Incremental Featurism

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

Our Bag of Components:



Registers

Design Approach

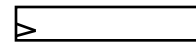
Incremental Featurism

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

Our Bag of Components:



Registers



Muxes

Design Approach

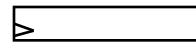
Incremental Featurism

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

Our Bag of Components:



Registers



Muxes



ALU & adders 

Design Approach

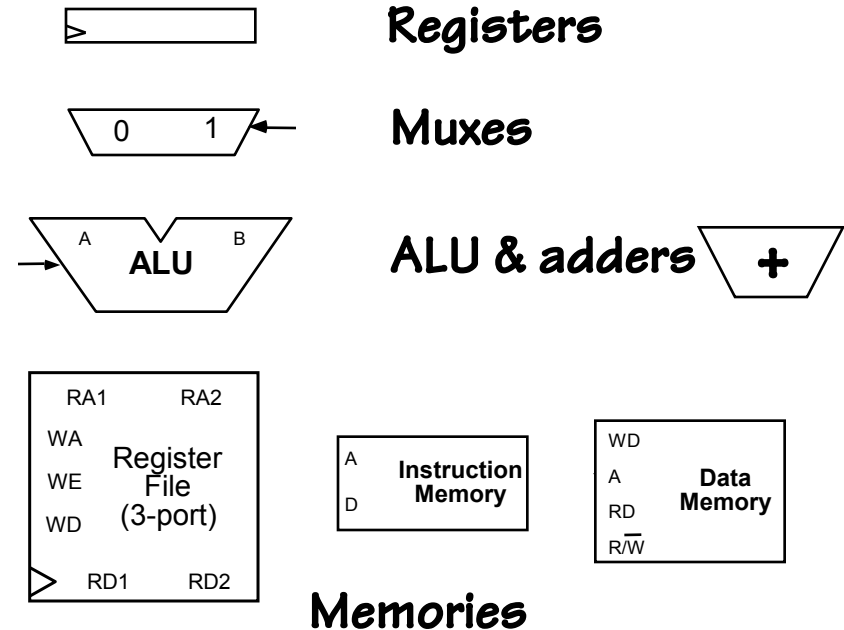
Incremental Featurism

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:

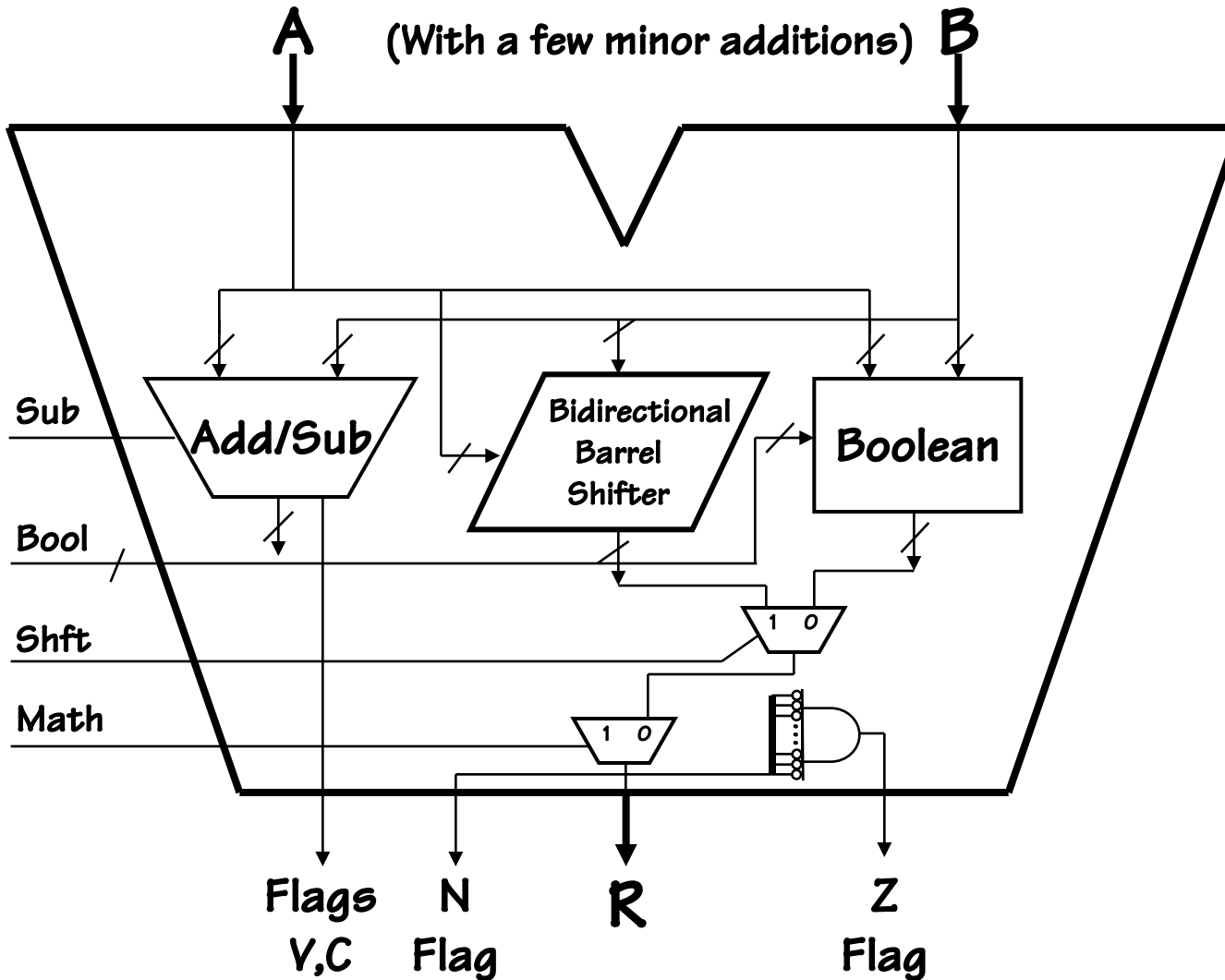
1. 3-Operand ALU instructions
2. ALU w/immediate instructions
2. Load & Store Instructions
3. Jump & Branch instructions
4. Exceptions

Our Bag of Components:



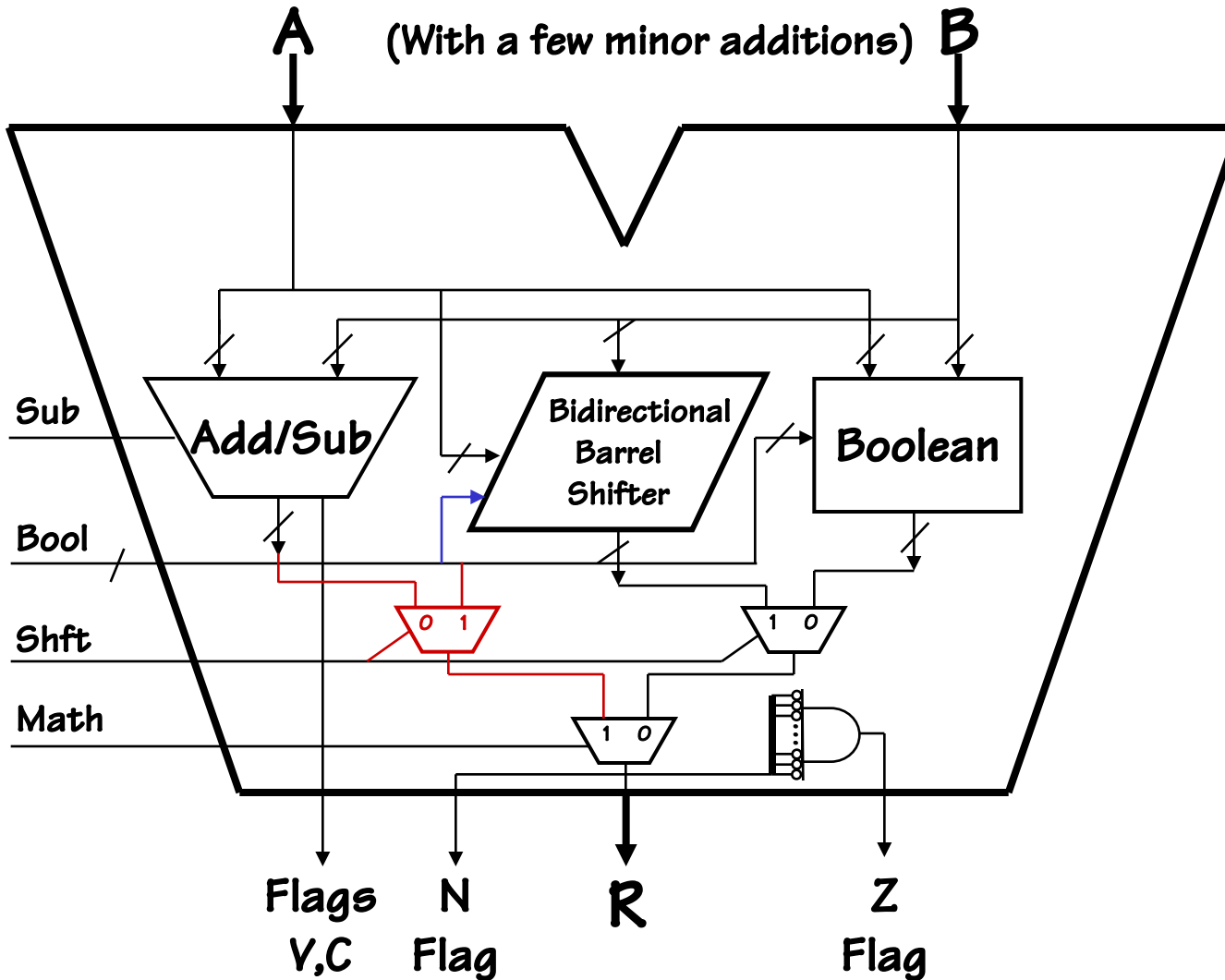
A Few ALU Tweaks

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



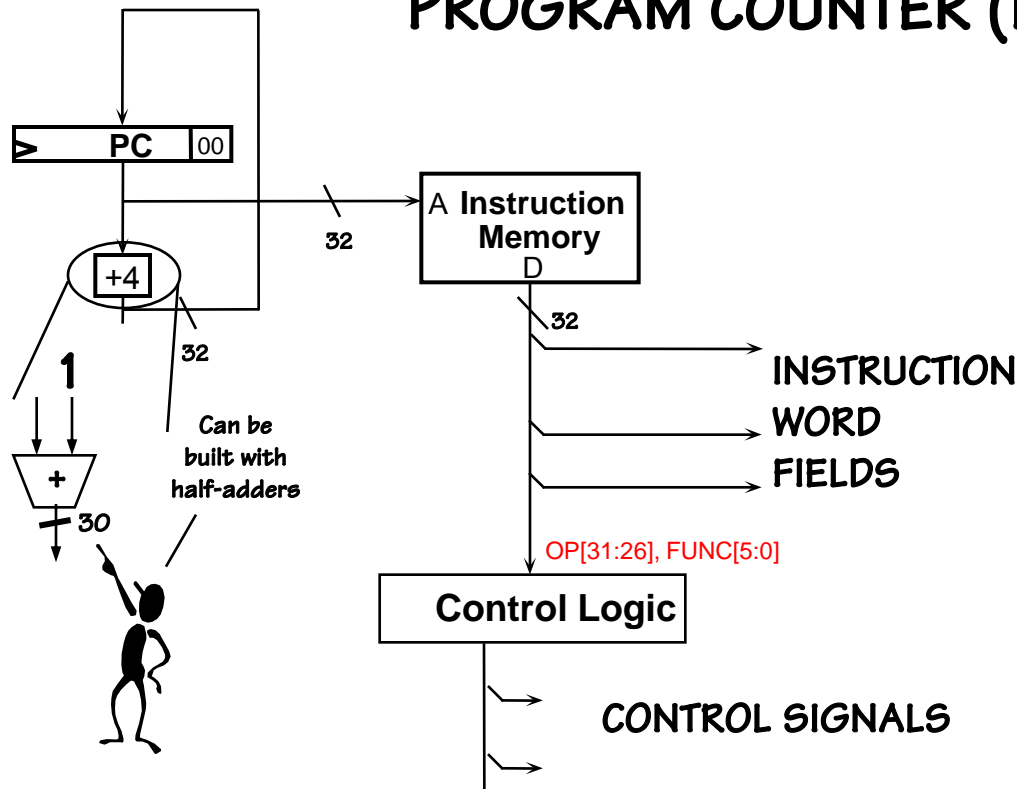
A Few ALU Tweaks

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



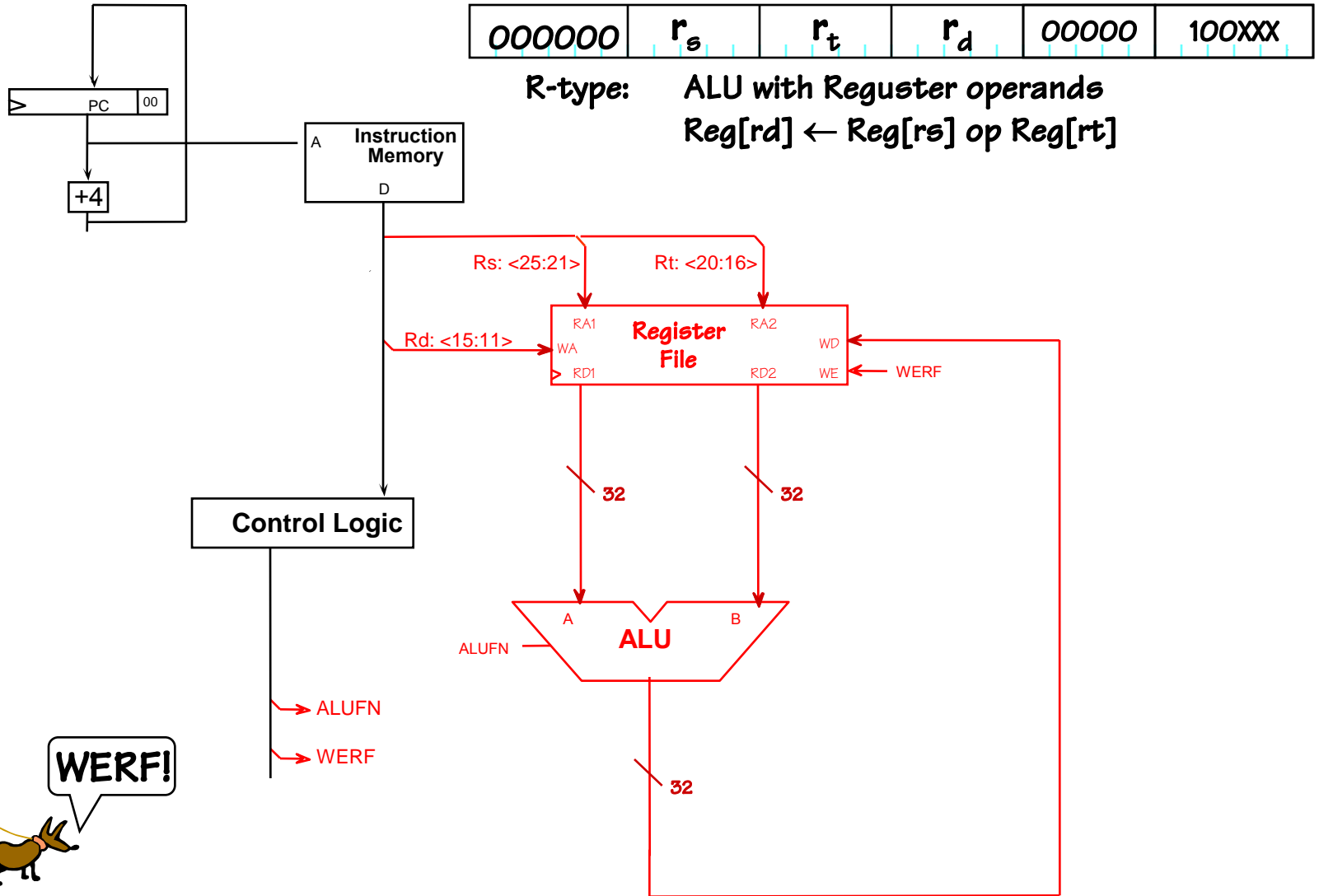
Instruction Fetch/Decode

- 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 $\langle 31:26 \rangle$ and $\langle 5:0 \rangle$ to generate controls

3-Operand ALU Data Path

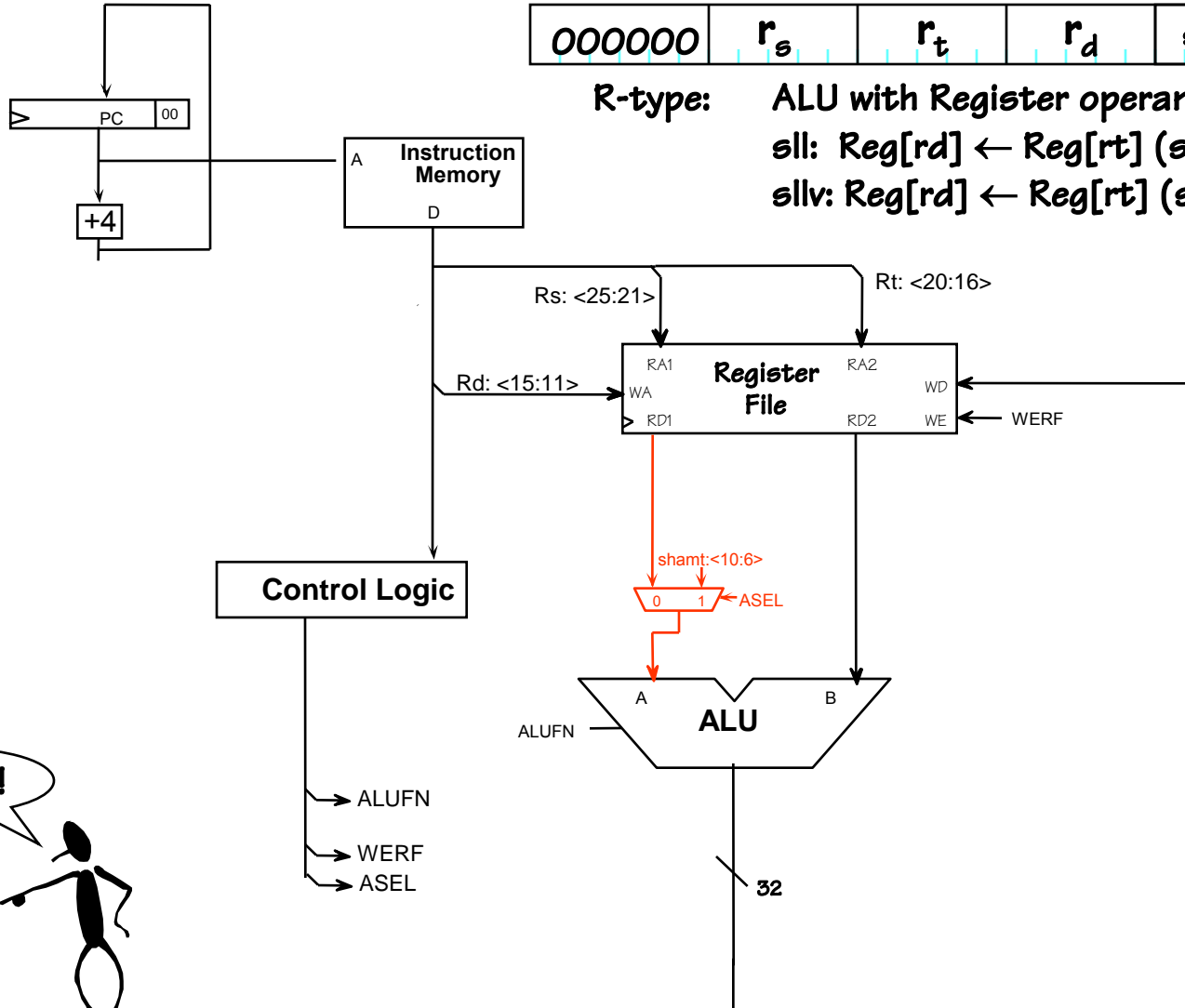


Shift Instructions

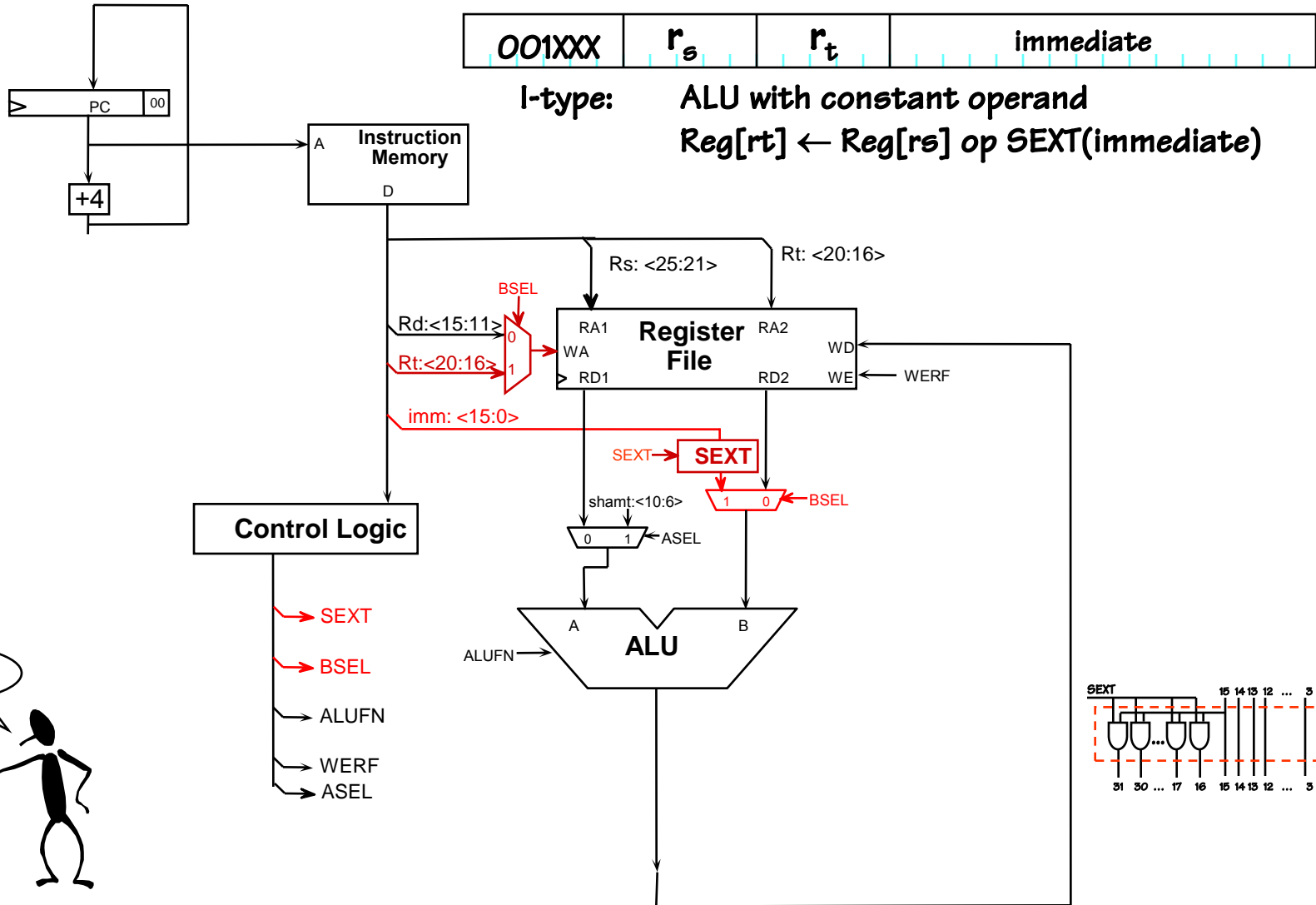


R-type: ALU with Register operands

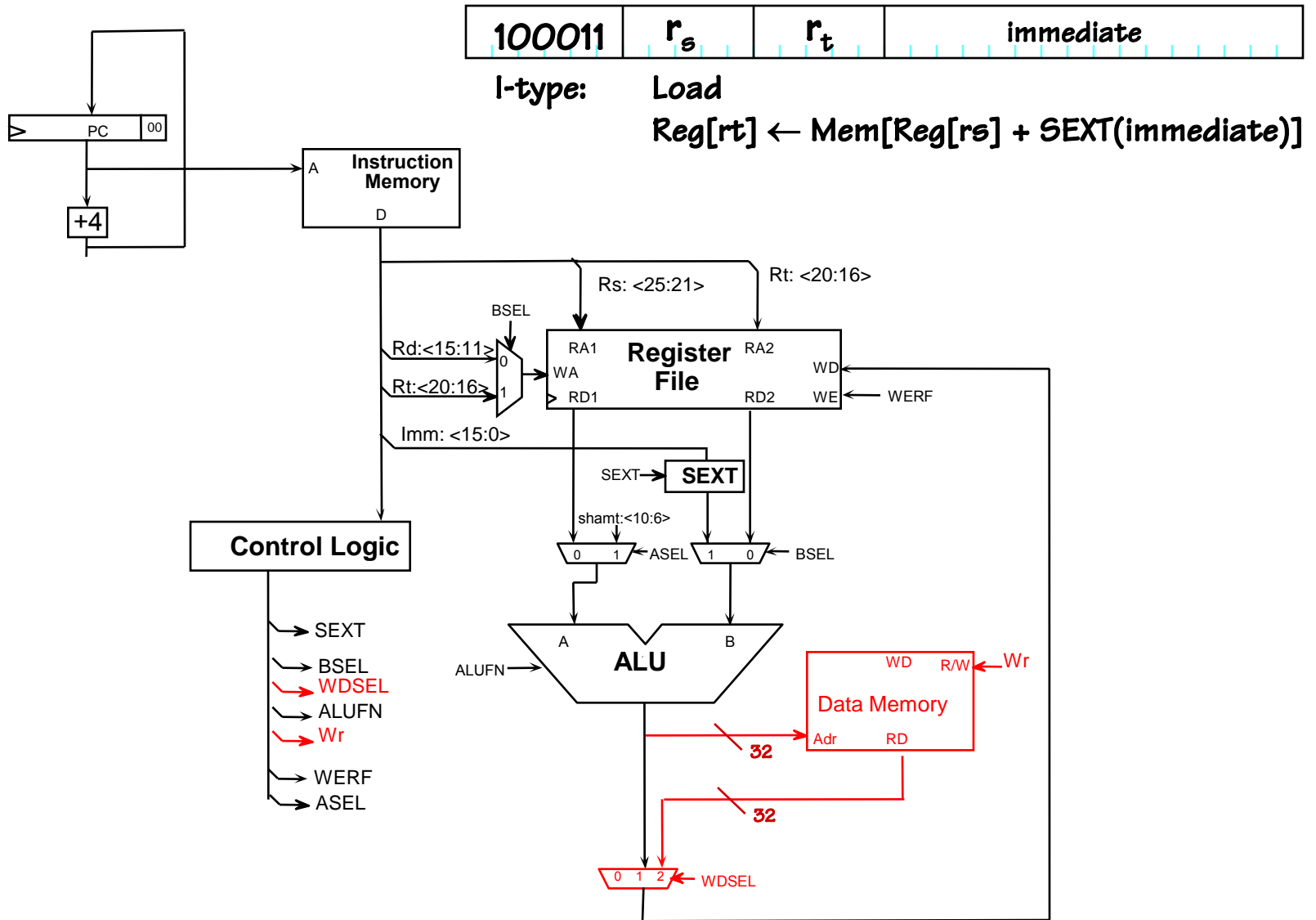
sll: $\text{Reg}[rd] \leftarrow \text{Reg}[rt] \text{ (shift) shamt}$
 sllv: $\text{Reg}[rd] \leftarrow \text{Reg}[rt] \text{ (shift) Reg}[rs]$



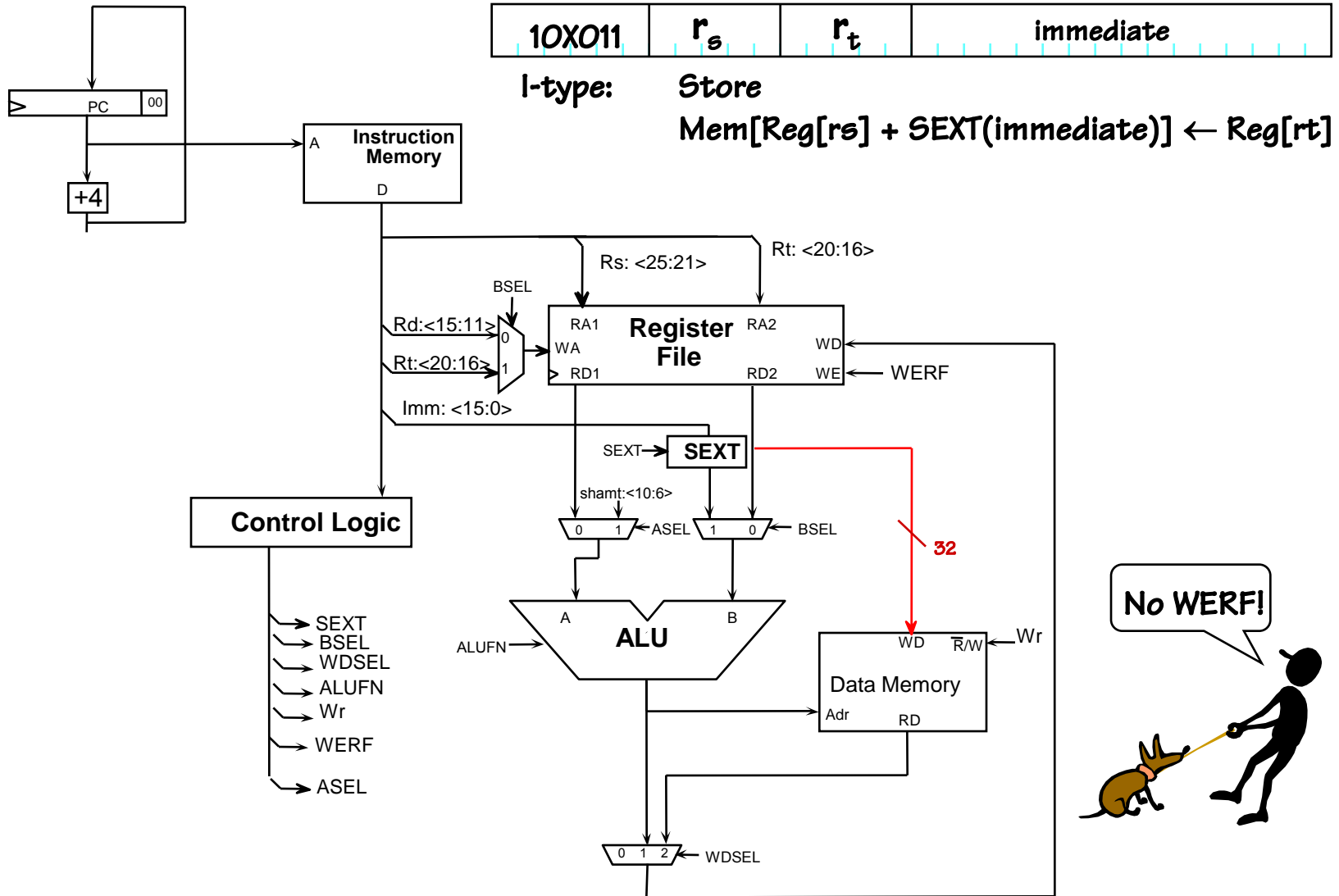
ALU with Immediate



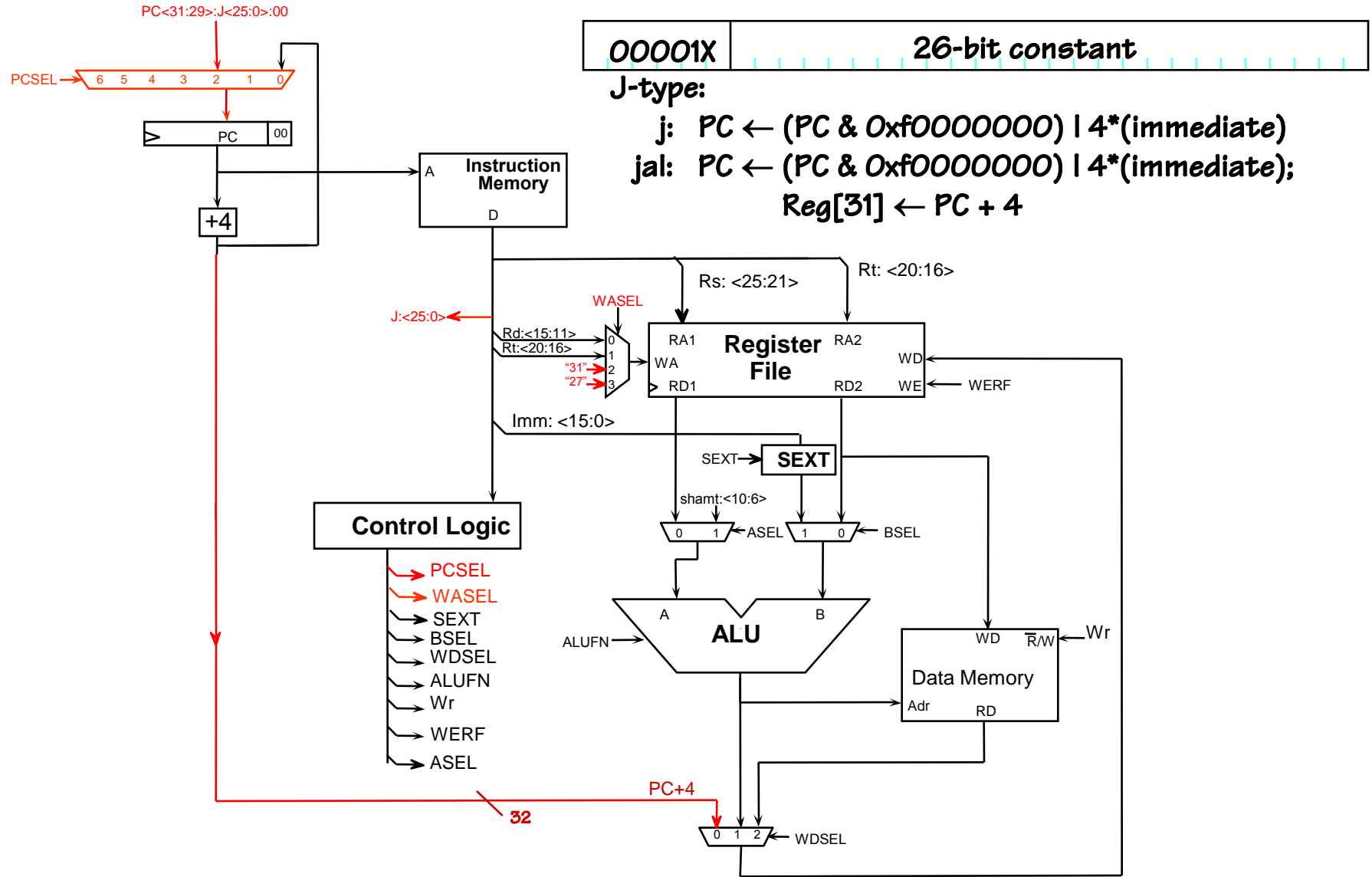
Load Instruction



Store Instruction



JMP Instructions



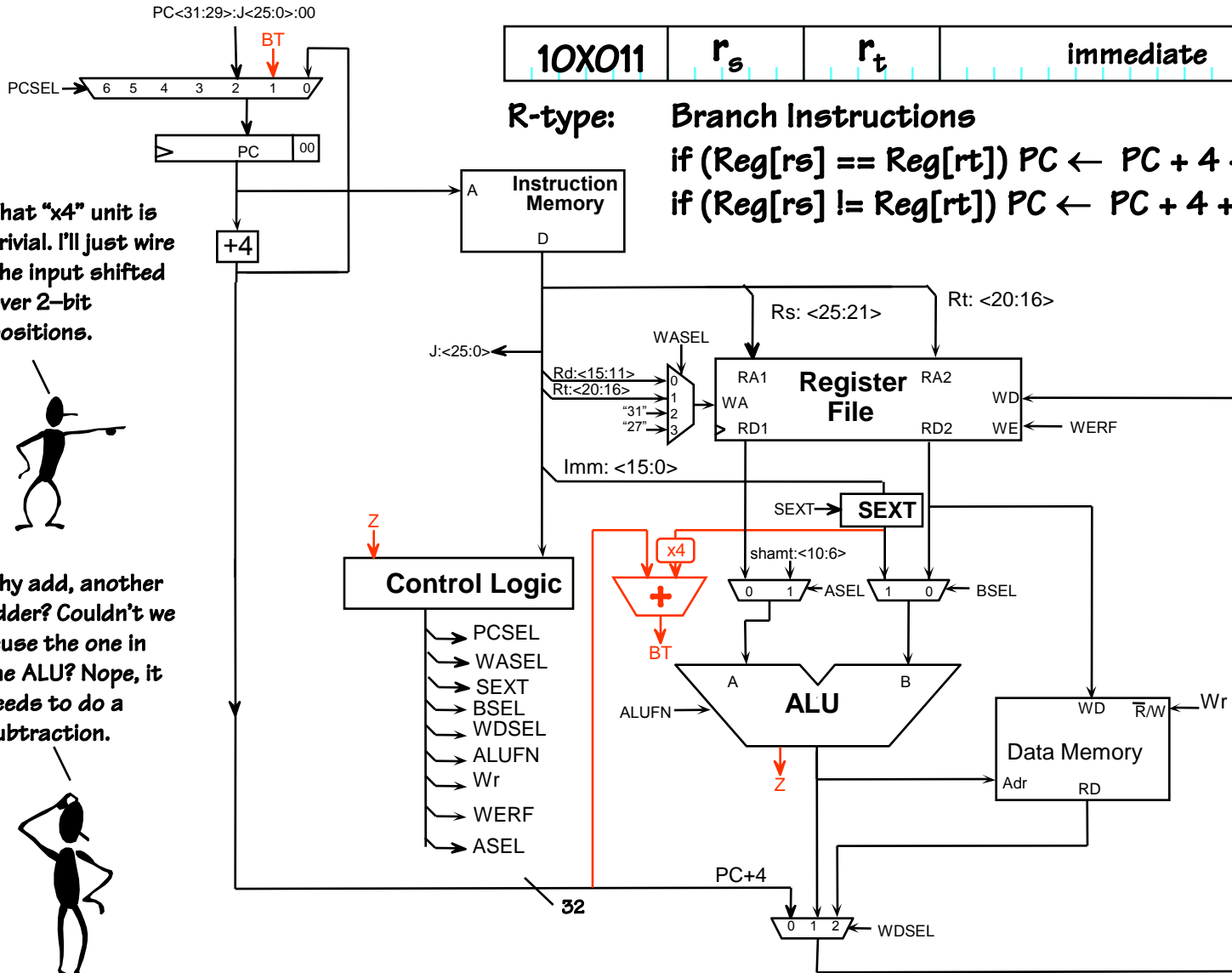
BEQ/BNE Instructions



R-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)



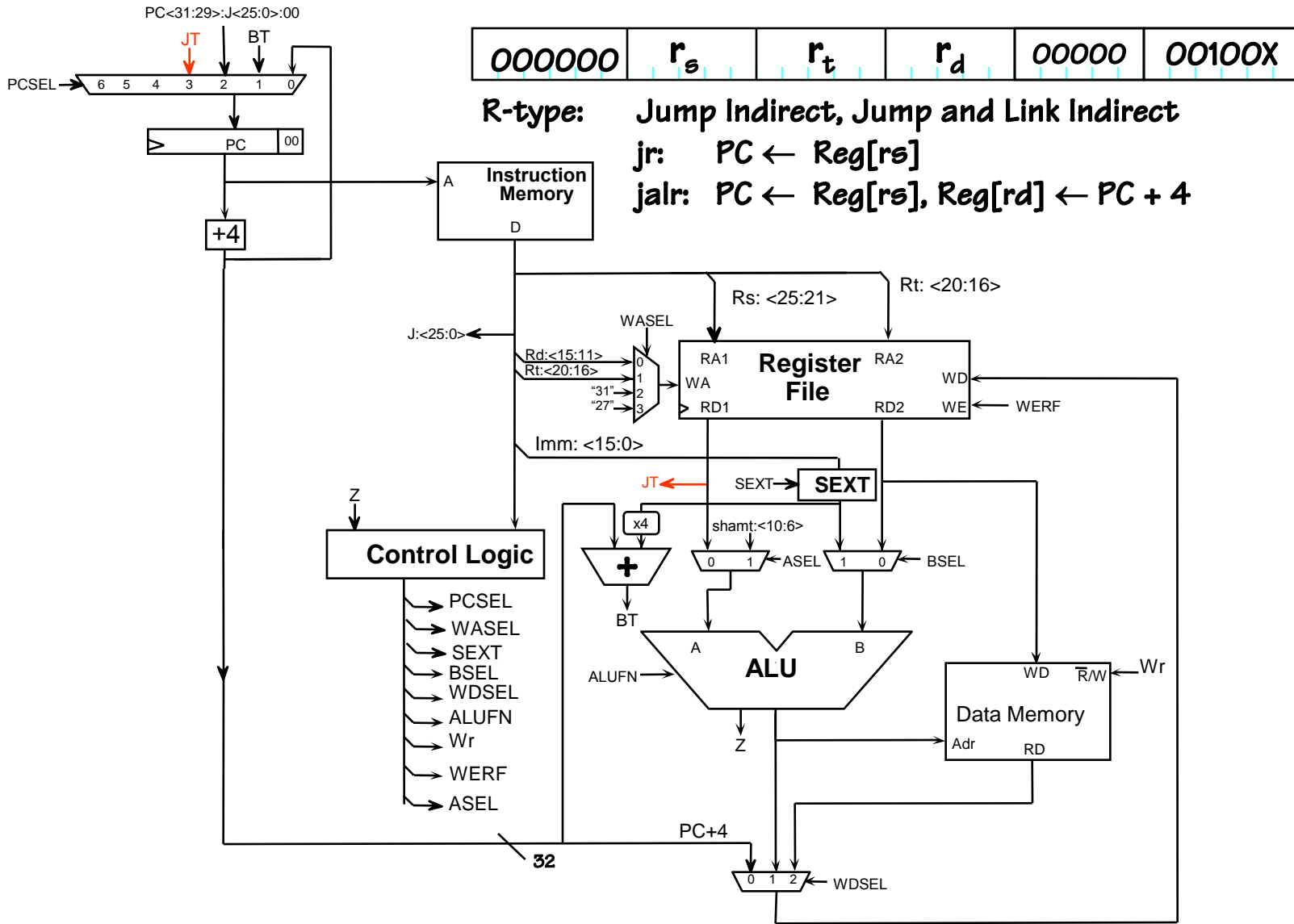
That "x4" unit is trivial. I'll just wire the input shifted over 2-bit positions.



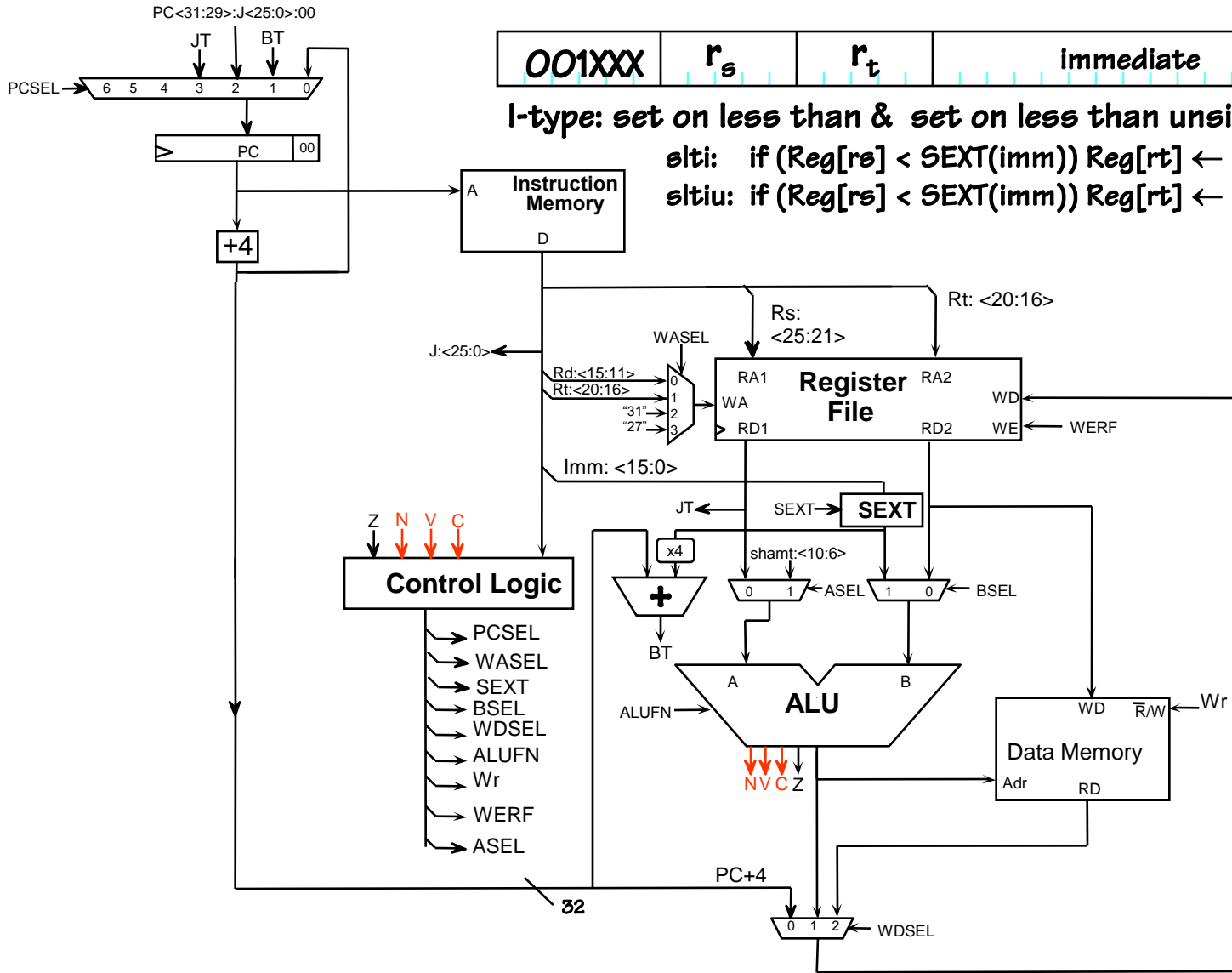
Why add, another adder? Couldn't we reuse the one in the ALU? Nope, it needs to do a subtraction.



Jump Indirect Instructions



Loose Ends



I-type: set on less than & set on less than unsigned immediate

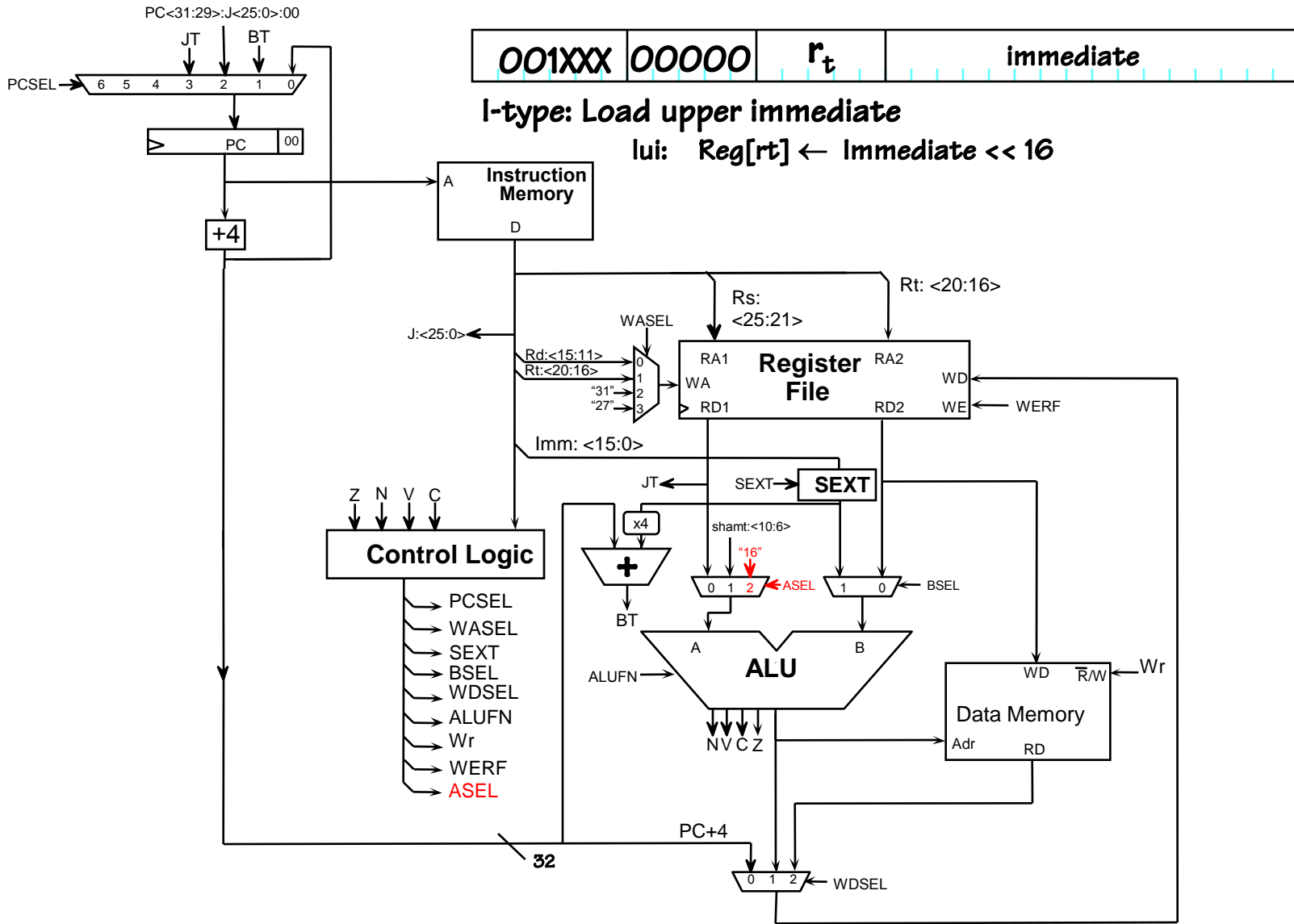
slt: if ($Reg[r_s] < SEXT(imm)$) $Reg[r_t] \leftarrow 1$; else $Reg[r_t] \leftarrow 0$

sltiu: if ($Reg[r_s] < SEXT(imm)$) $Reg[r_t] \leftarrow 1$; else $Reg[r_t] \leftarrow 0$

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

 $LT = N \oplus V$
 $LTU = C$

LUI



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
- TRAPS & system calls (eg, read-a-character)
 - CPU generated

(Implemented as an "agreed upon" Illegal instruction)

- I/O events (eg, key struck)
 - externally generated

[synchronous]

[synchronous]

[asynchronous]

These are "Software" notions of synchrony and asynchrony

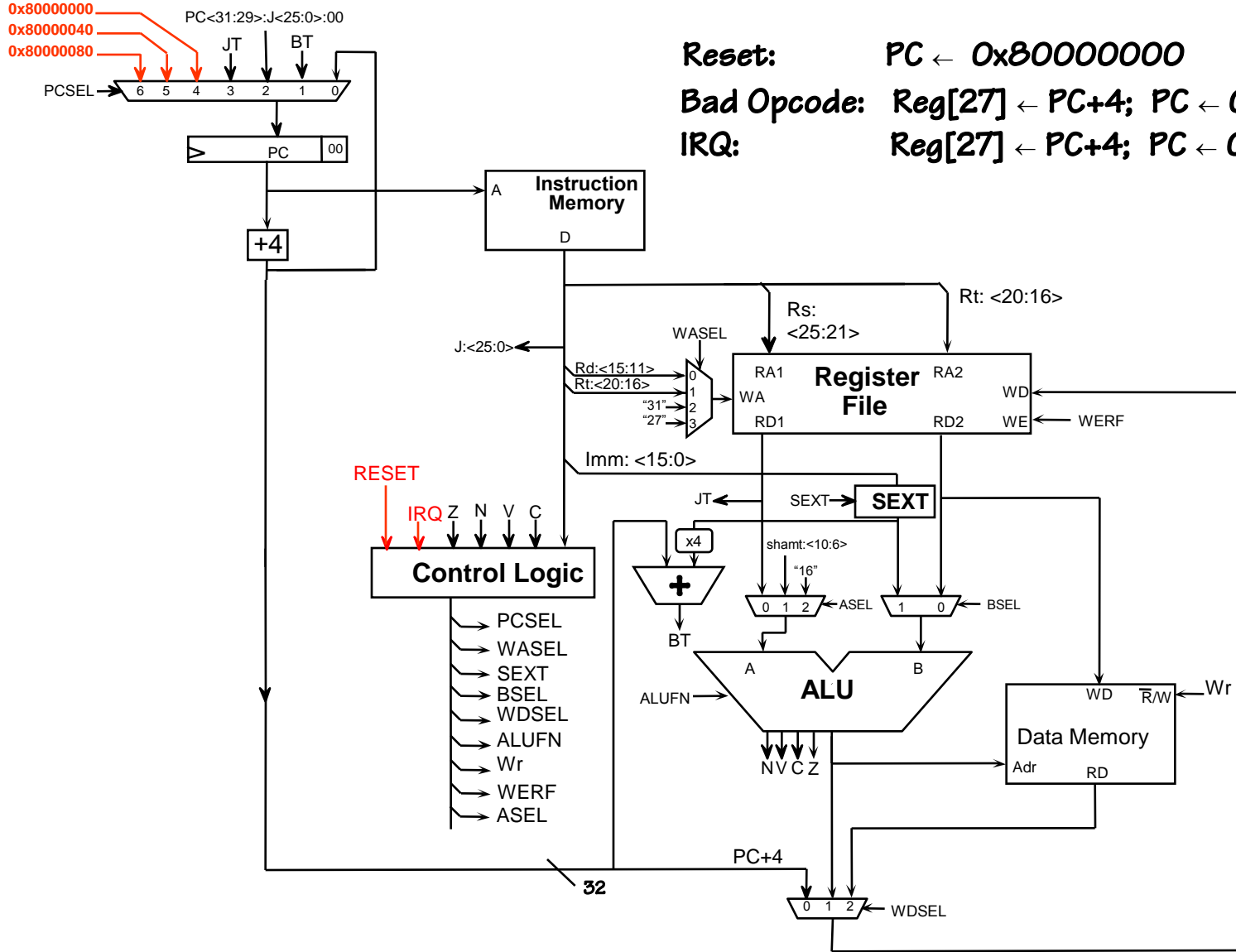


EXCEPTION GOAL: Interrupt running program, invoke exception handler, return to continue execution.

Exceptions

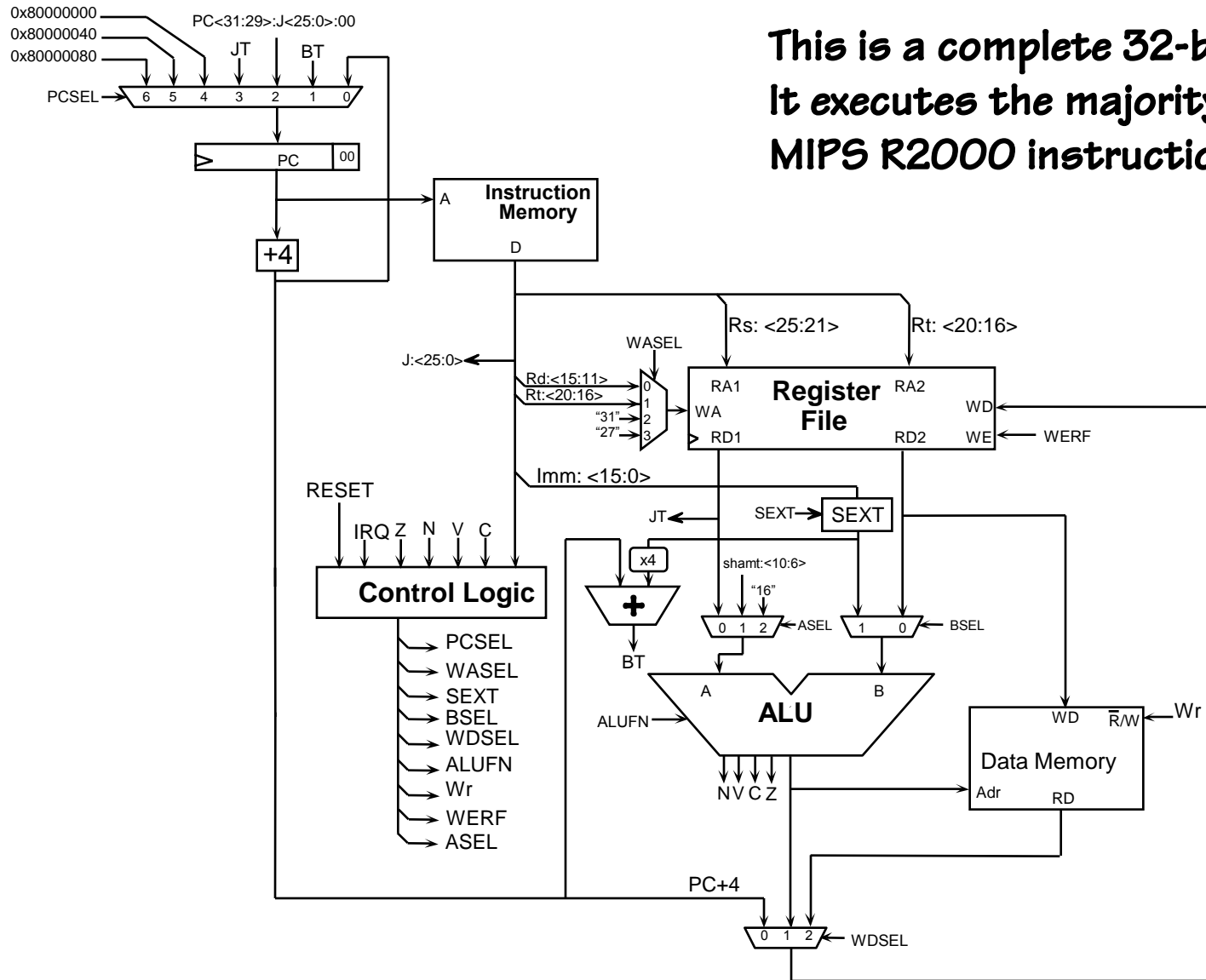
0x80000000
 0x80000040
 0x80000080

Reset: $PC \leftarrow 0x80000000$
Bad Opcode: $Reg[27] \leftarrow PC+4; PC \leftarrow 0x80000040$
IRQ: $Reg[27] \leftarrow PC+4; PC \leftarrow 0x80000080$



MIPS: Our Final Version

This is a complete 32-bit processor. 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 built as a large ROM

Instruction	R E S E T	I R Q	Z	N	V	C	P C S E L	S E X T	W A S E L	W D S E L	ALUFN Sub Bool Shft Math				W R	W E R F	A S E L	B S E L
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																		