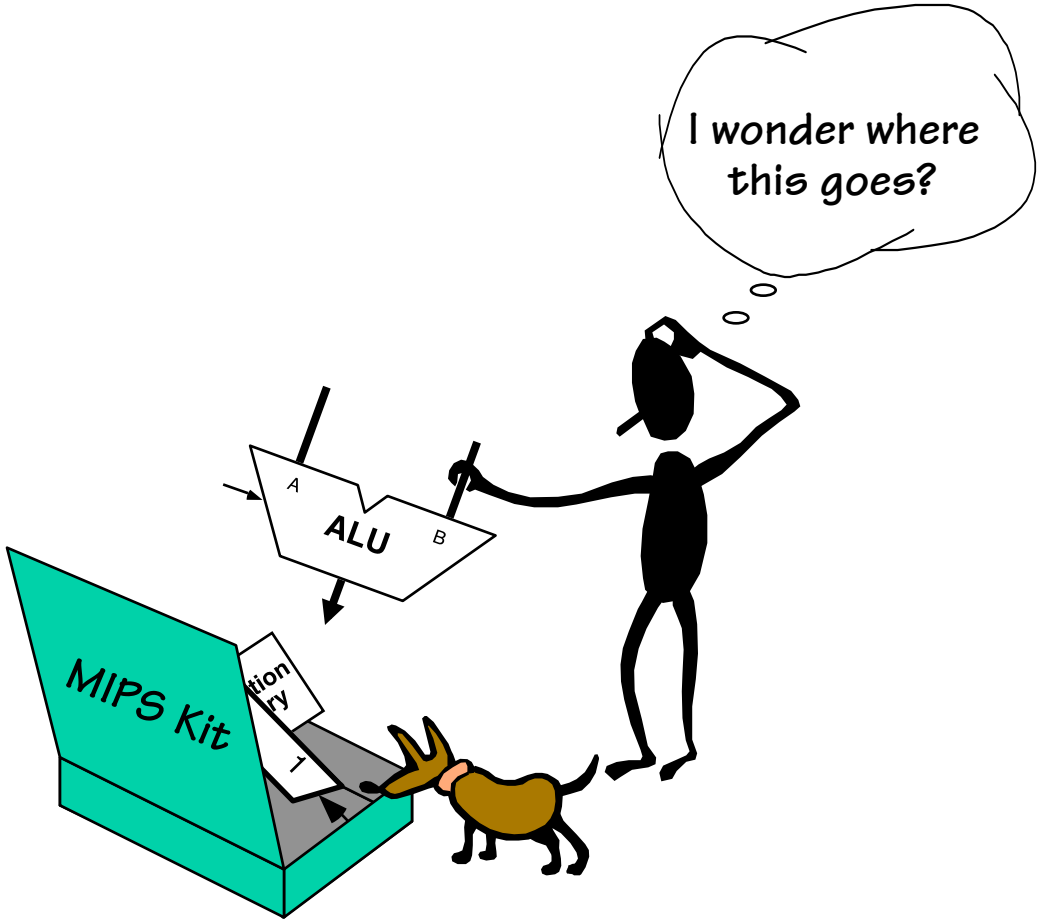
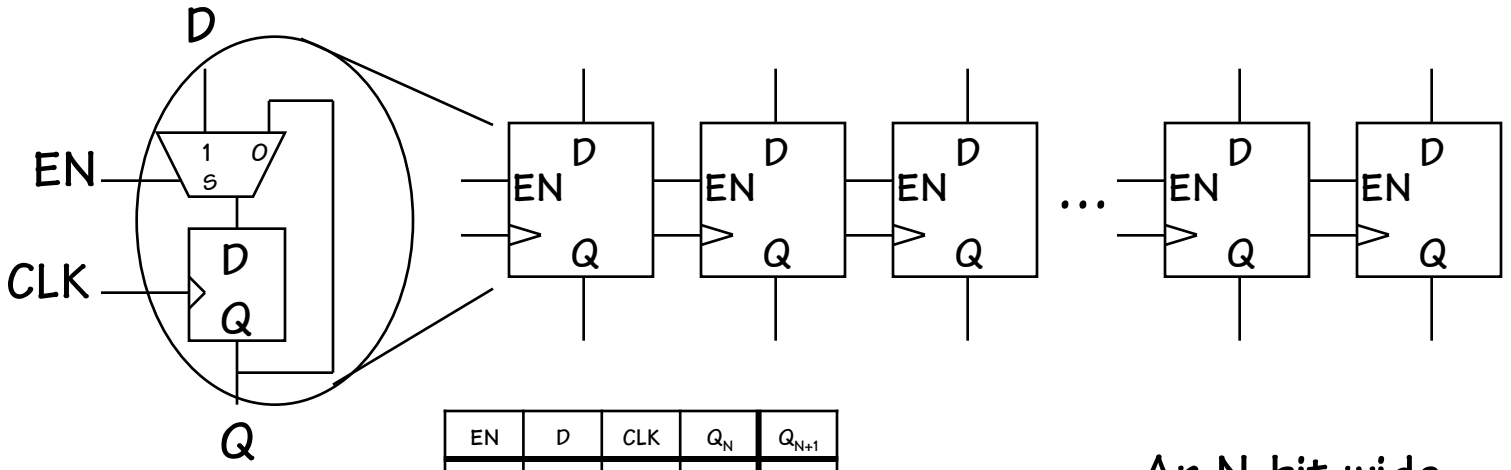
Building a Computer



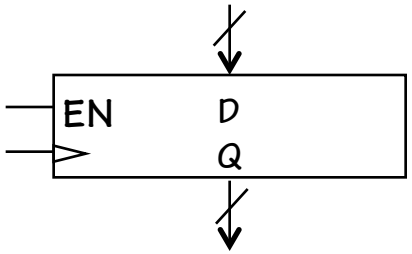
One More Functional Unit

Thus far, our building 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. Now we add a control that enables the loading of a register.



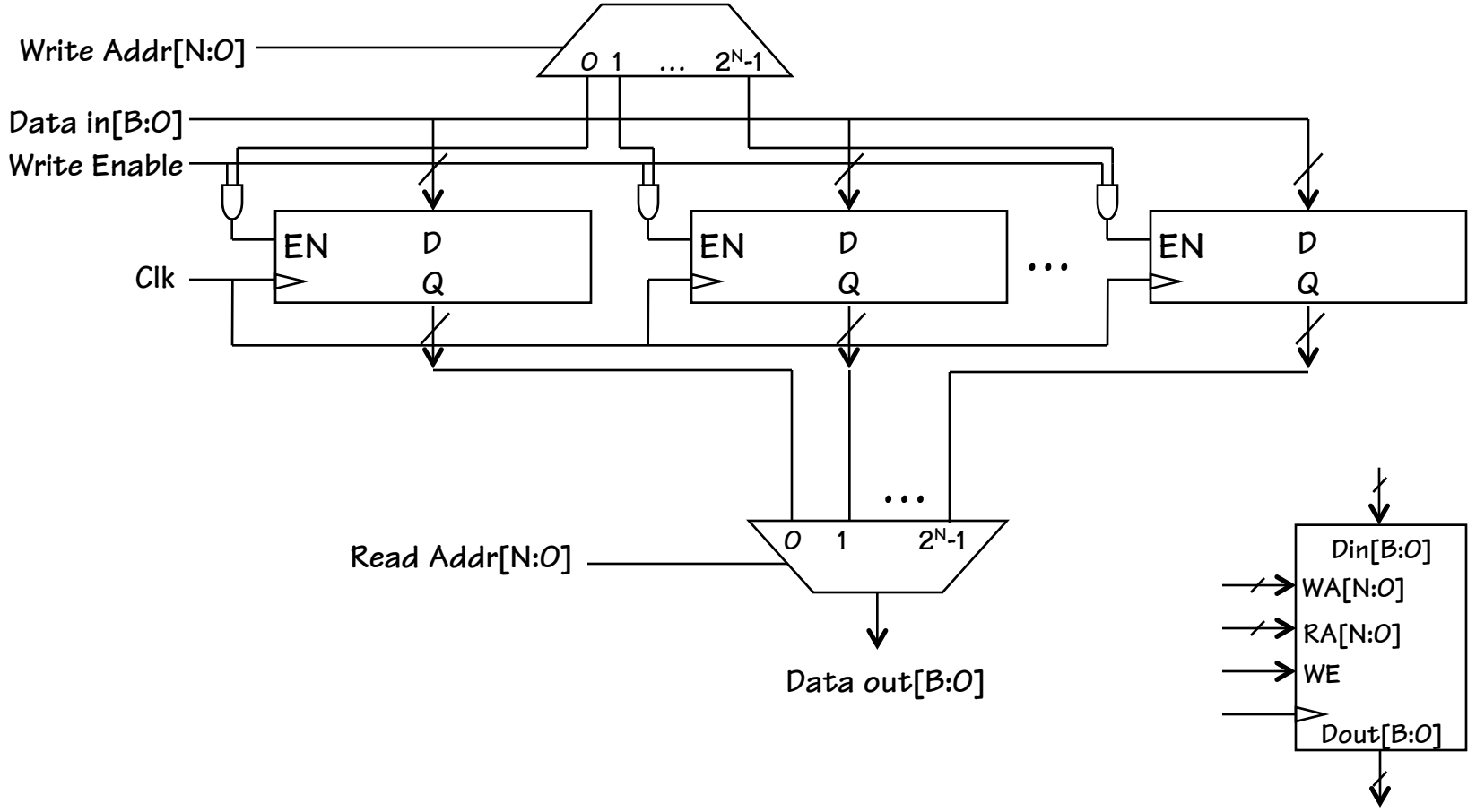
EN	D	CLK	Q_N	Q_{N+1}
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	1
0	X	↑	0	0
0	X	↑	1	1
1	0	↑	X	0
1	1	↑	X	1

An N-bit wide Register with enable



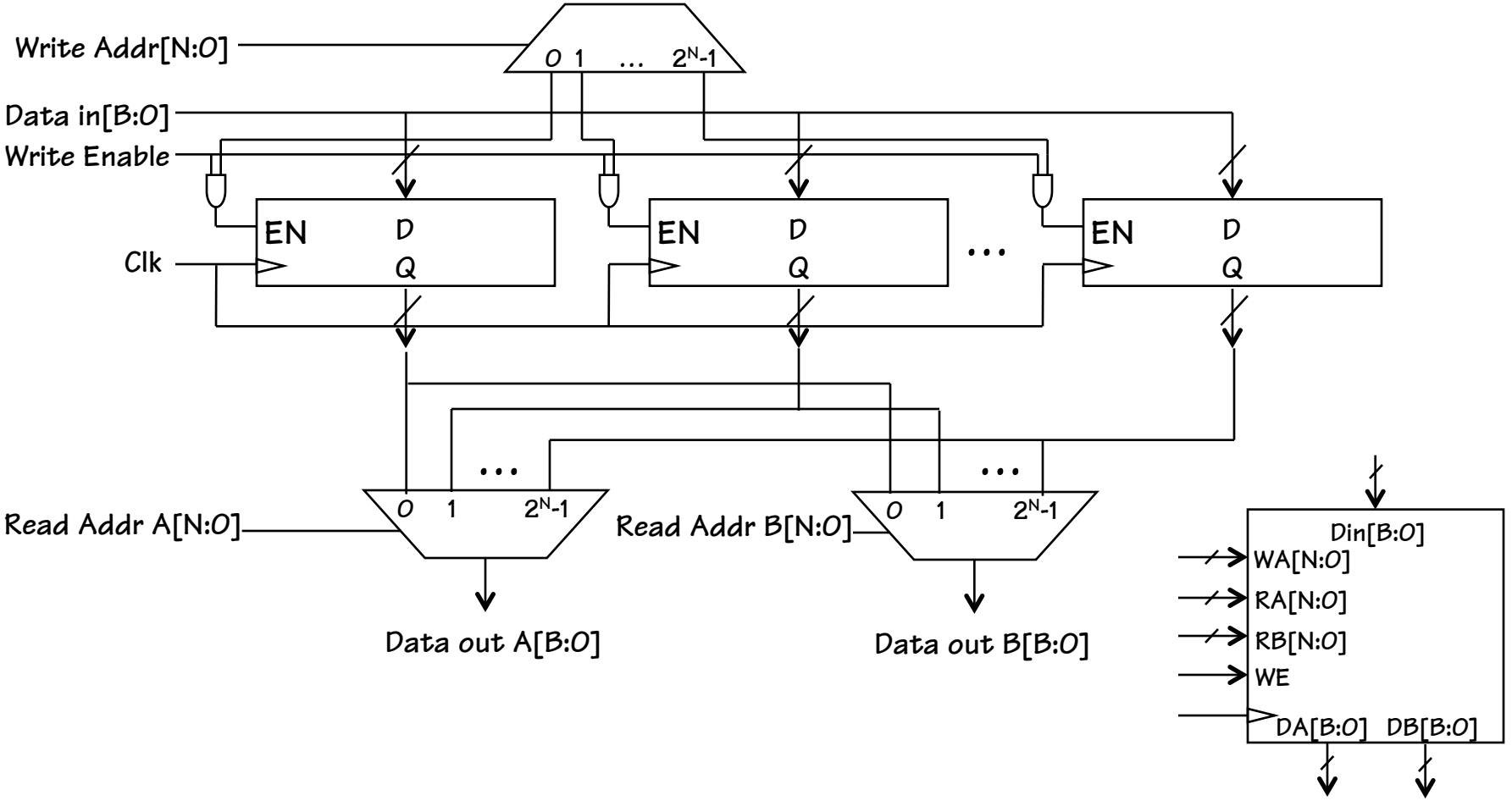
A Register File

We can also construct an addressable array of registers



A Multi-Ported Register File

Multiple read ports by simply adding more output MUXs



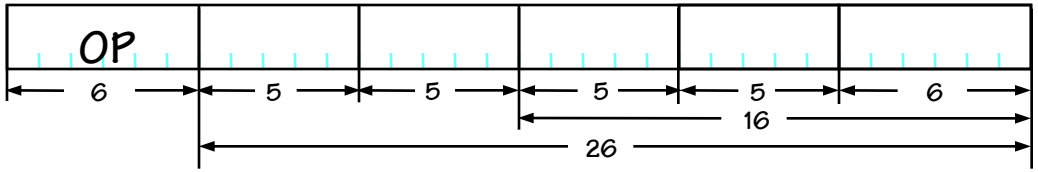
THIS IS IT!

"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, smart phone, 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

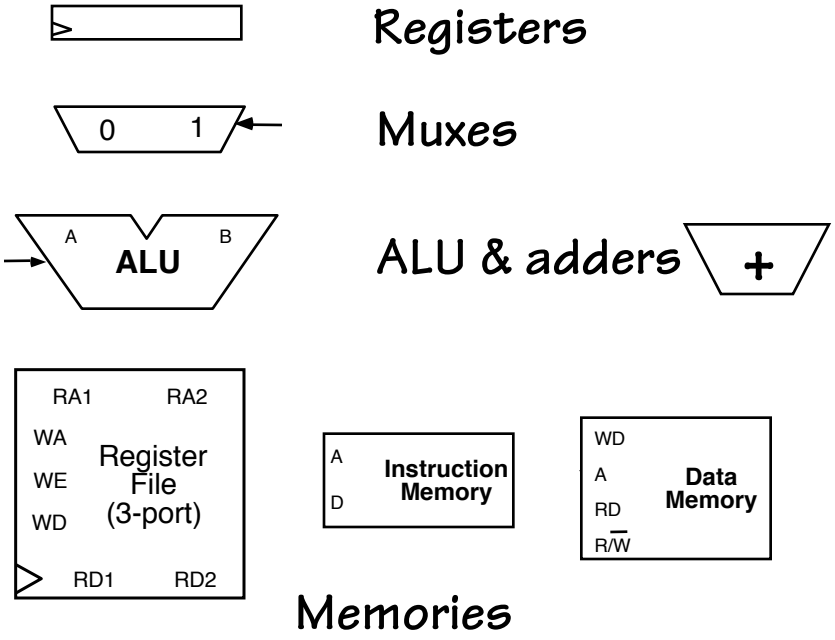
Incremental Featurism

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

Steps:

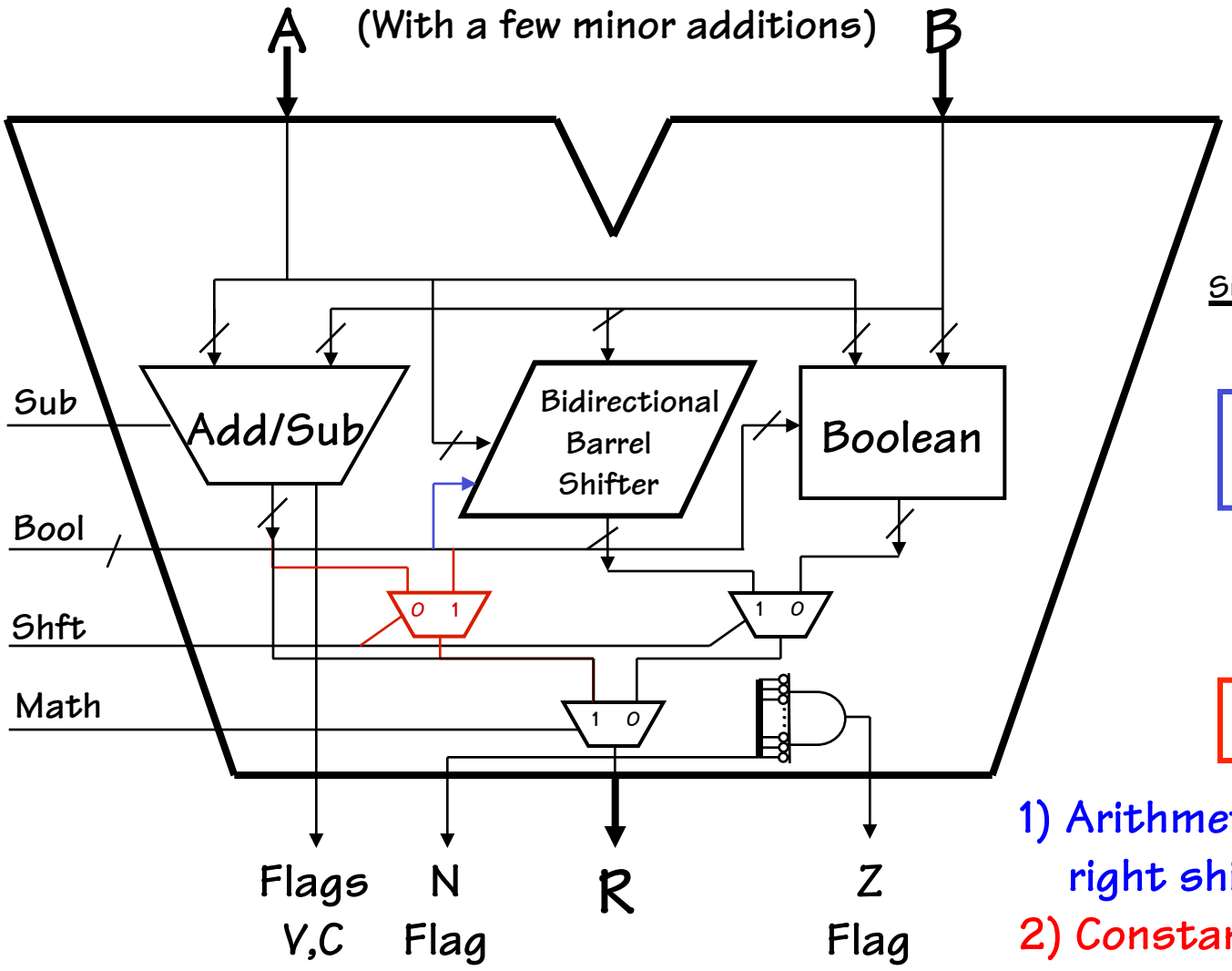
1. 3-Operand ALU instructions
2. ALU w/immediate instructions
2. Load & Store Instructions
3. Jump & Branch instructions
4. Leftovers
5. Reset & Exceptions

Our Bag of Components:



A Few ALU Tweaks

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



5-bit ALUFN

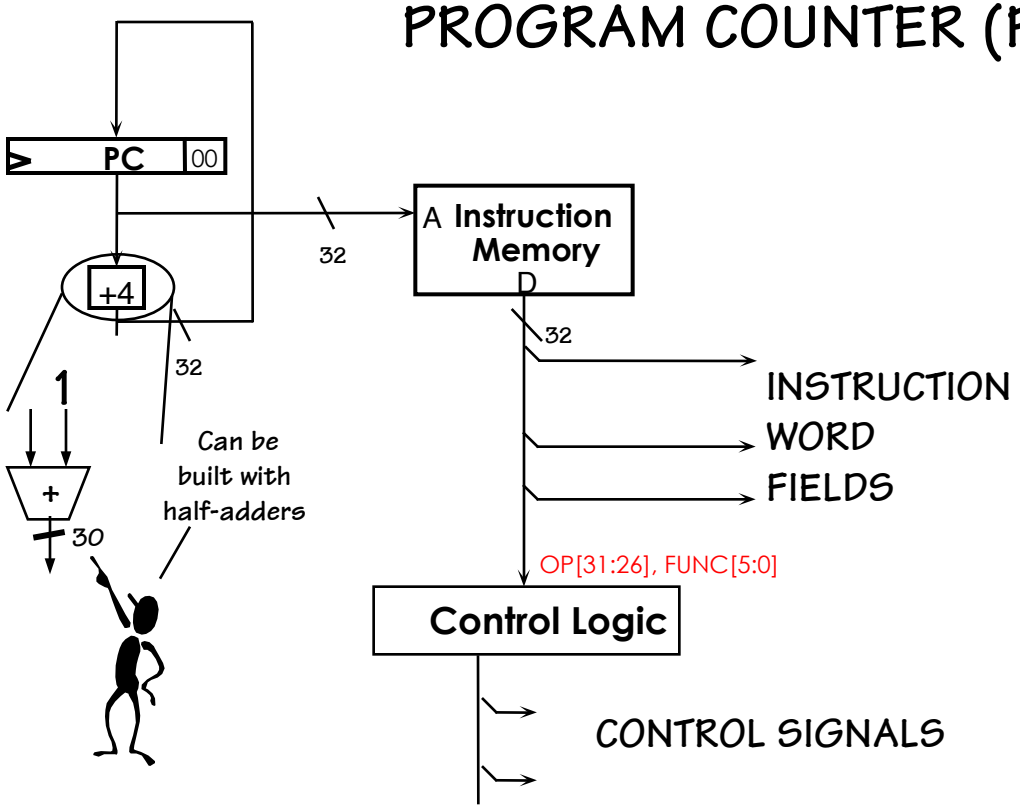
Sub	Bool	Shft	Math	OP
0	XX	0	1	A+B
1	XX	0	1	A-B
X	00	1	0	B<<A
X	10	1	0	B>>A
X	11	1	0	B>>>A
X	00	0	0	A & B
X	01	0	0	A | B
X	10	0	0	A ^ B
X	11	0	0	$\overline{A \mid B}$

1) Arithmetic/Logical right shifts

2) Constants 0 and 1

Instruction Fetch/Decode

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



- use PC as memory address
- add 4 to current PC, and update on the next rising clock
- fetch instruction from memory
  - use some instruction fields directly (register numbers, 16-bit constant)
  - use bits <31:26> and <5:0> to generate controls

A "counter" is an FSM where the "next state" is just the "current state" plus some constant

MIPS Instruction Decoding Ring

OP	000	001	010	011	100	101	110	111
000	ALU		j	jal	beq	bne		
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010								
011								
100				lw				
101				sw				
110								
111								

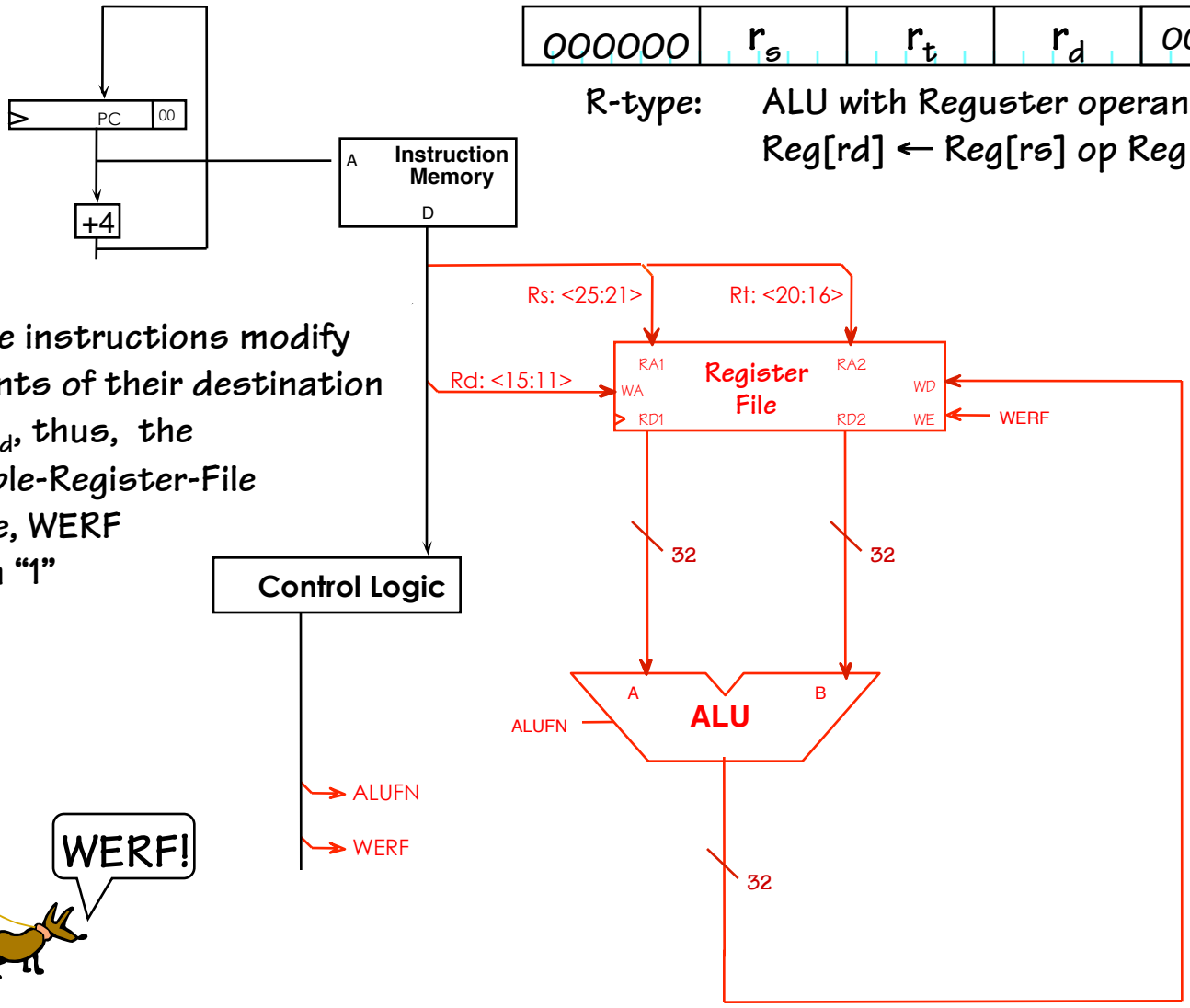
ALU	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010								
011	mul		div					
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

3-Operand ALU Data Path

R-type: ALU with Reguster operands

Reg[rd] ← Reg[rs] op Reg[rt]

All of these instructions modify the contents of their destination register, r_d, thus, the Write-Enable-Register-File control line, WERF is always a "1"

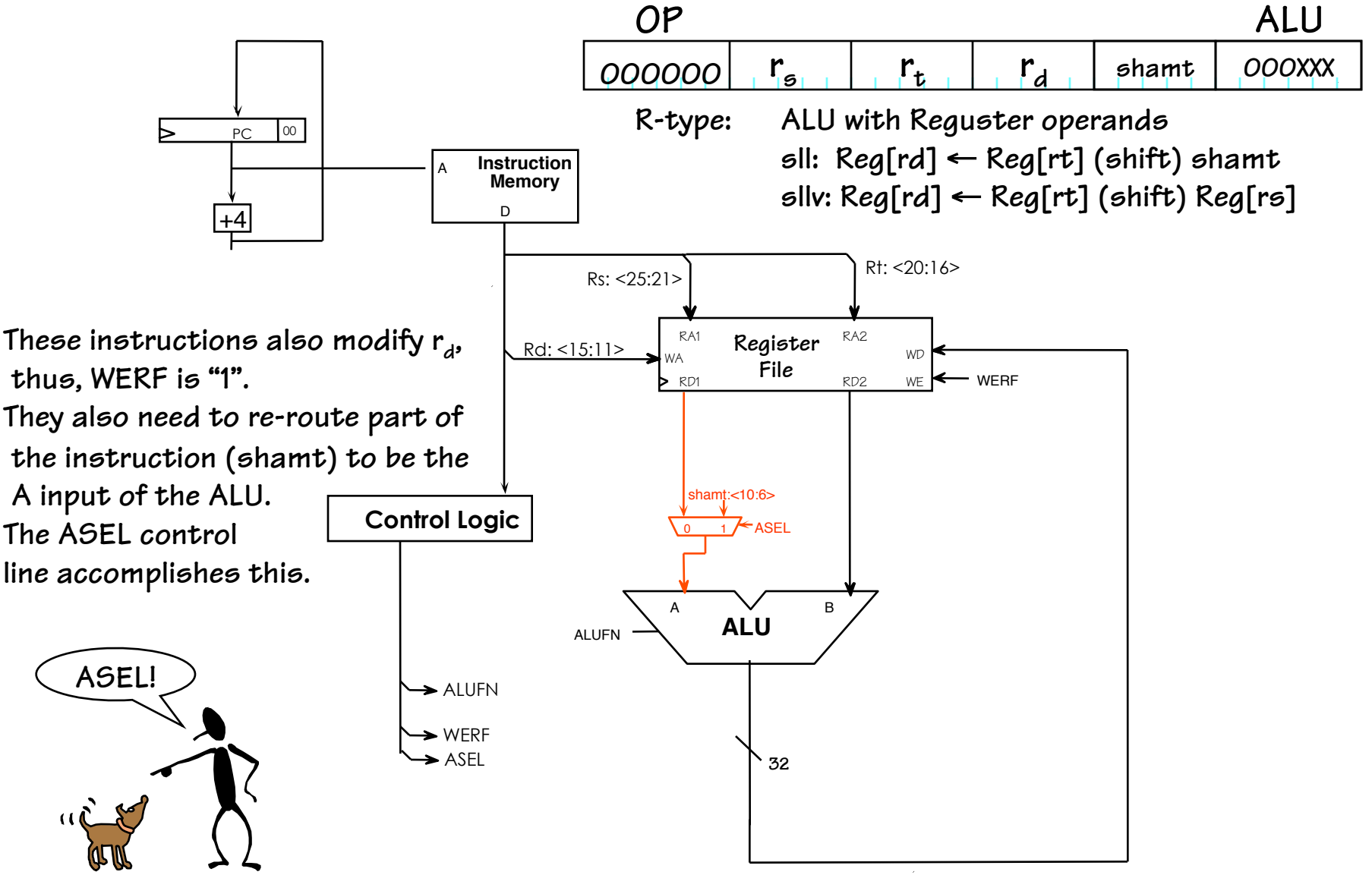


MIPS Instruction Decoding Ring

OP	000	001	010	011	100	101	110	111
000	ALU		j	jal	beq	bne		
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010								
011								
100				lw				
101				sw				
110								
111								

ALU	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010								
011	mul		div					
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

Shift Instructions

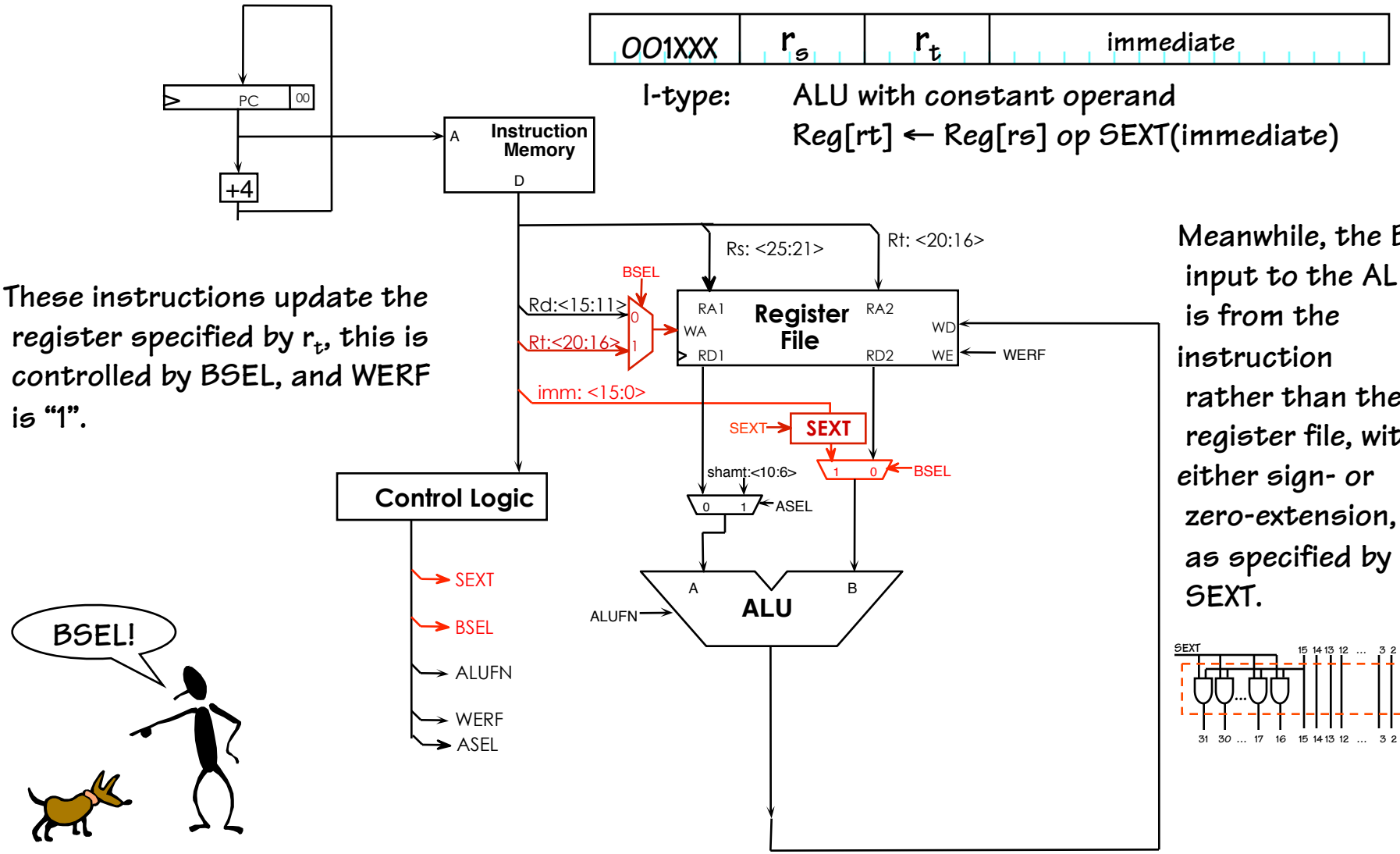


MIPS Instruction Decoding Ring

OP	000	001	010	011	100	101	110	111
000	ALU		j	jal	beq	bne		
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010								
011								
100				lw				
101				sw				
110								
111								

ALU	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010								
011	mul		div					
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

ALU with Immediate



MIPS Instruction Decoding Ring

OP	000	001	010	011	100	101	110	111
000	ALU		j	jal	beq	bne		
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010								
011								
100				lw				
101				sw				
110								
111								

ALU	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010								
011	mul		div					
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

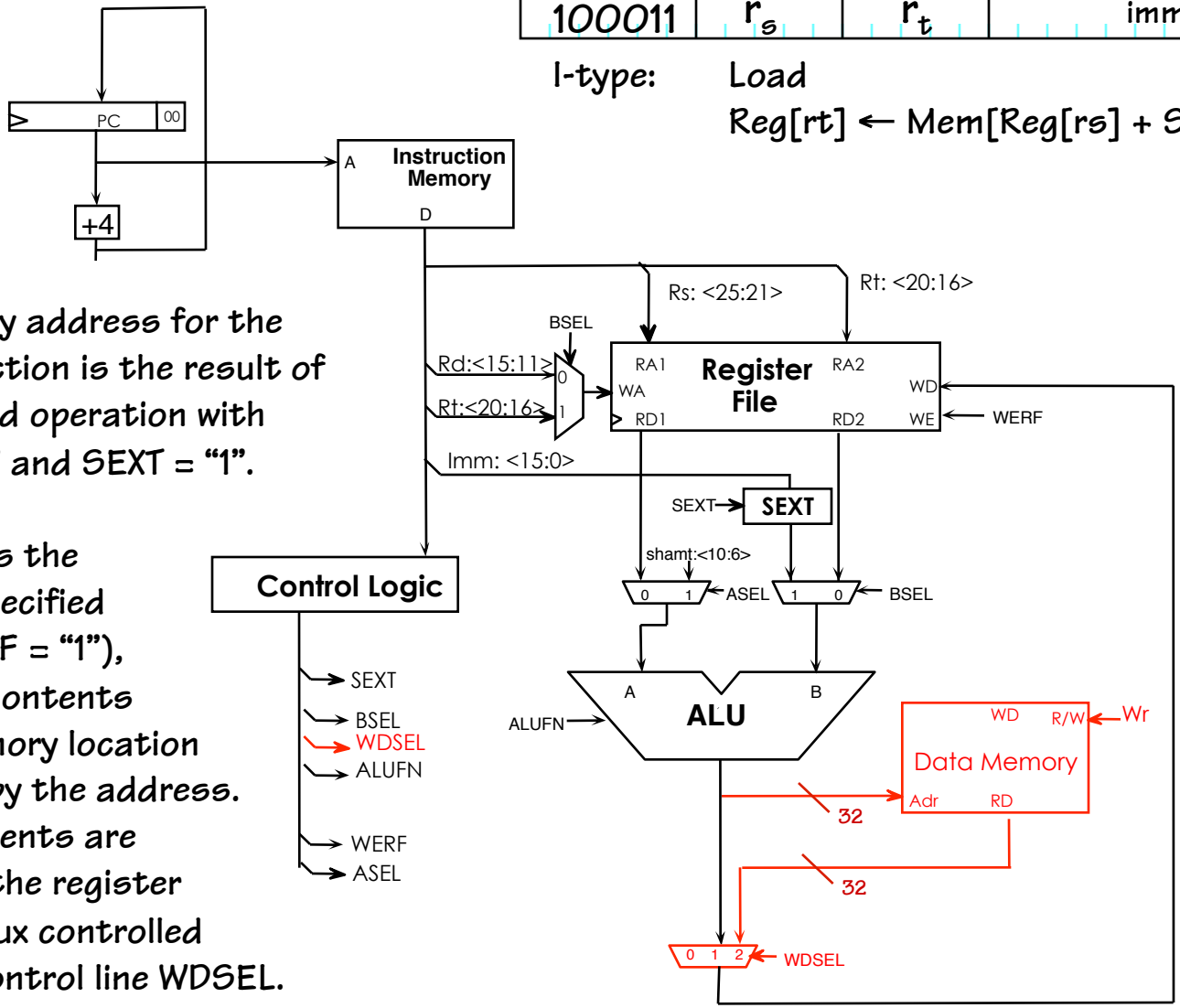
Load Instruction

I-type: Load

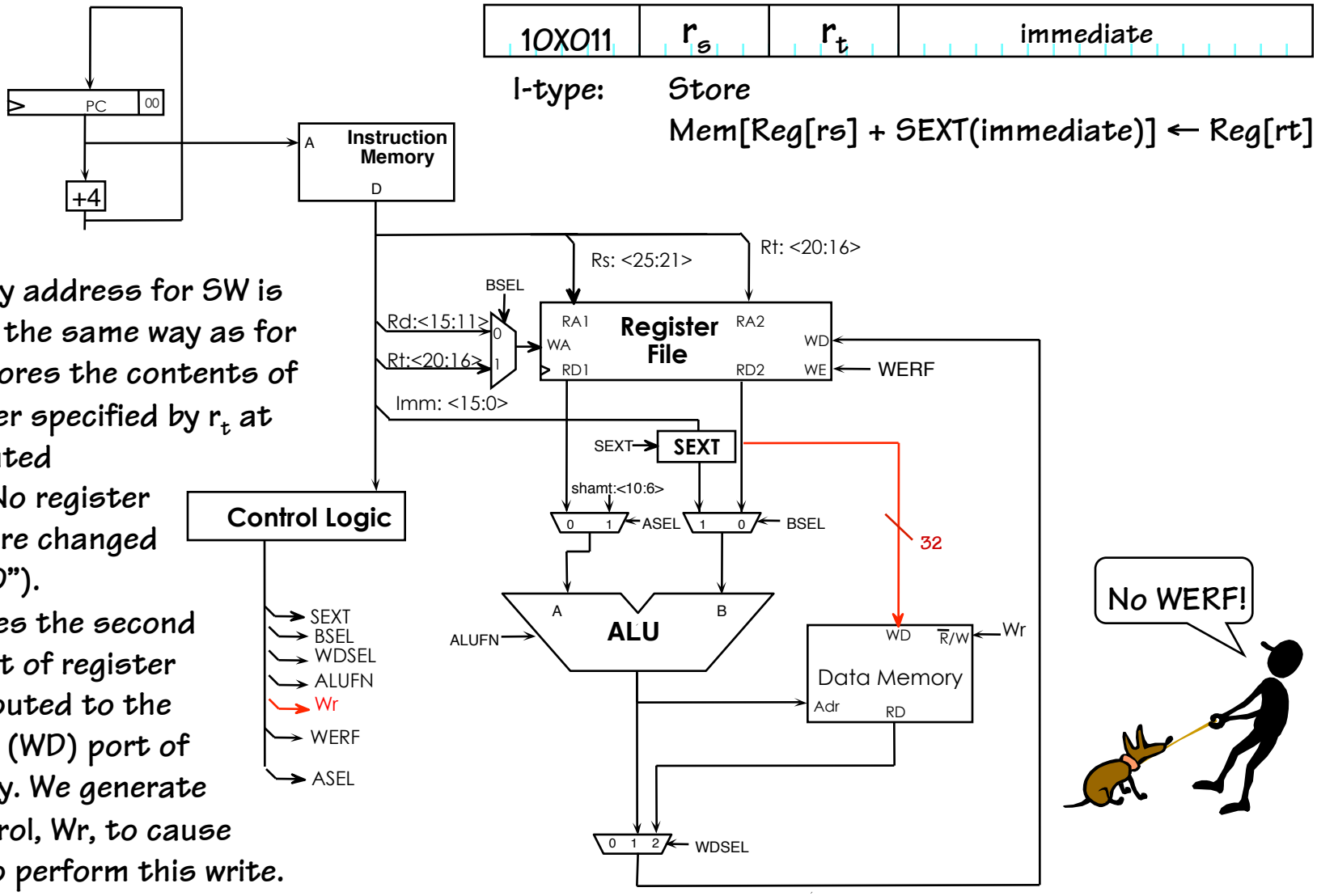
Reg[rt] ← Mem[Reg[rs] + SEXT(immediate)]

The memory address for the LW instruction is the result of an ALU add operation with BSEL = "1" and SEXT = "1".

LW updates the register specified by r_t, (WERF = "1"), using the contents of the memory location specified by the address. These contents are routed to the register file via a mux controlled by a new control line WDSEL.



Store Instruction



The memory address for SW is computed the same way as for LW. SW stores the contents of the register specified by r_t at the computed address. No register contents are changed (WERF = "0").

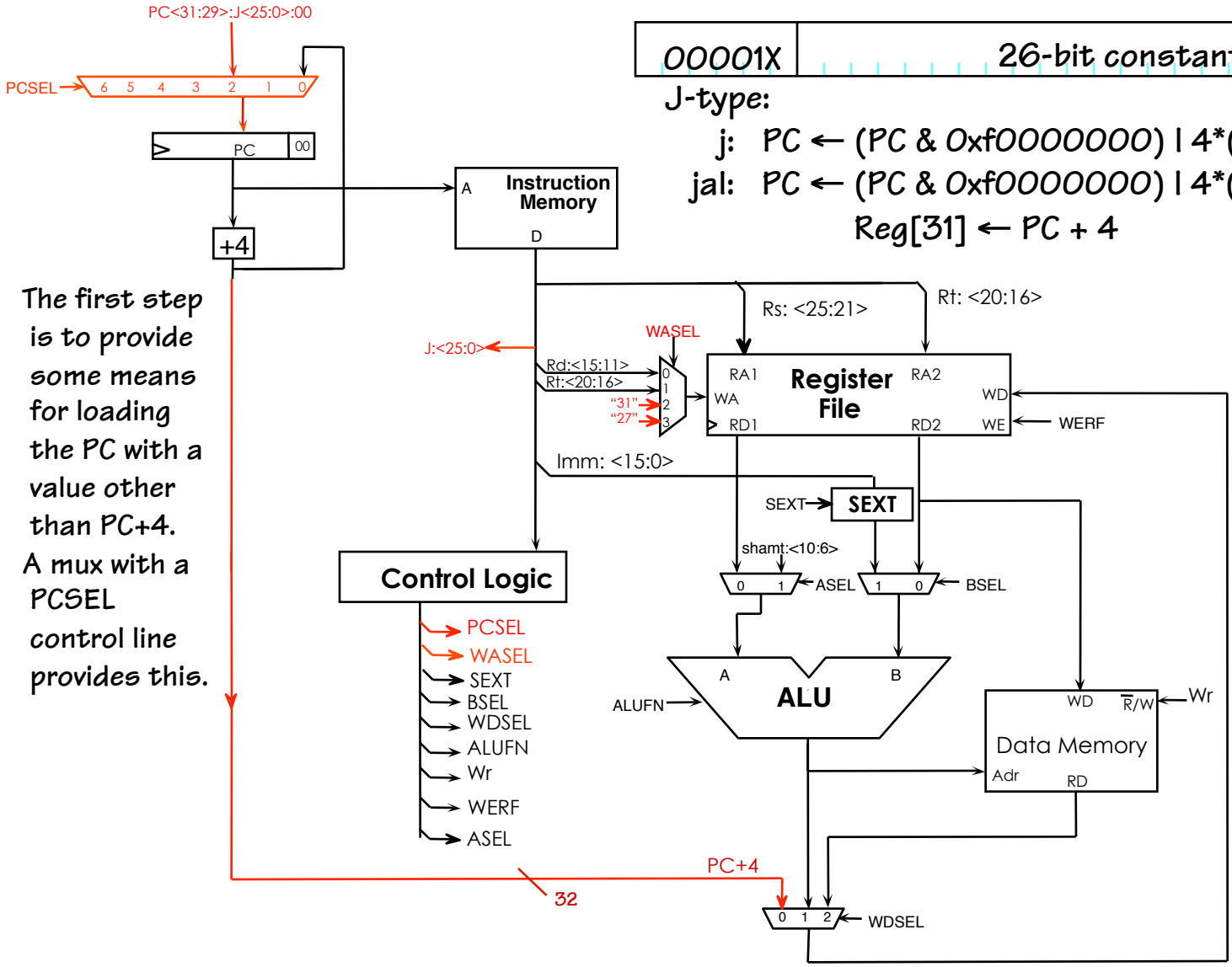
This requires the second output port of register file to be routed to the Write Data (WD) port of the memory. We generate a new control, Wr, to cause memory to perform this write.

MIPS Instruction Decoding Ring

OP	000	001	010	011	100	101	110	111
000	ALU		j	jal	beq	bne		
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010								
011								
100				lw				
101				sw				
110								
111								

ALU	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010								
011	mul		div					
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

JMP Instructions



00001X	26-bit constant

J-type:

j: PC ← (PC & 0xf0000000) | 4*(immediate)

jal: PC ← (PC & 0xf0000000) | 4*(immediate);
Reg[31] ← PC + 4

The first step is to provide some means for loading the PC with a value other than PC+4. A mux with a PCSEL control line provides this.

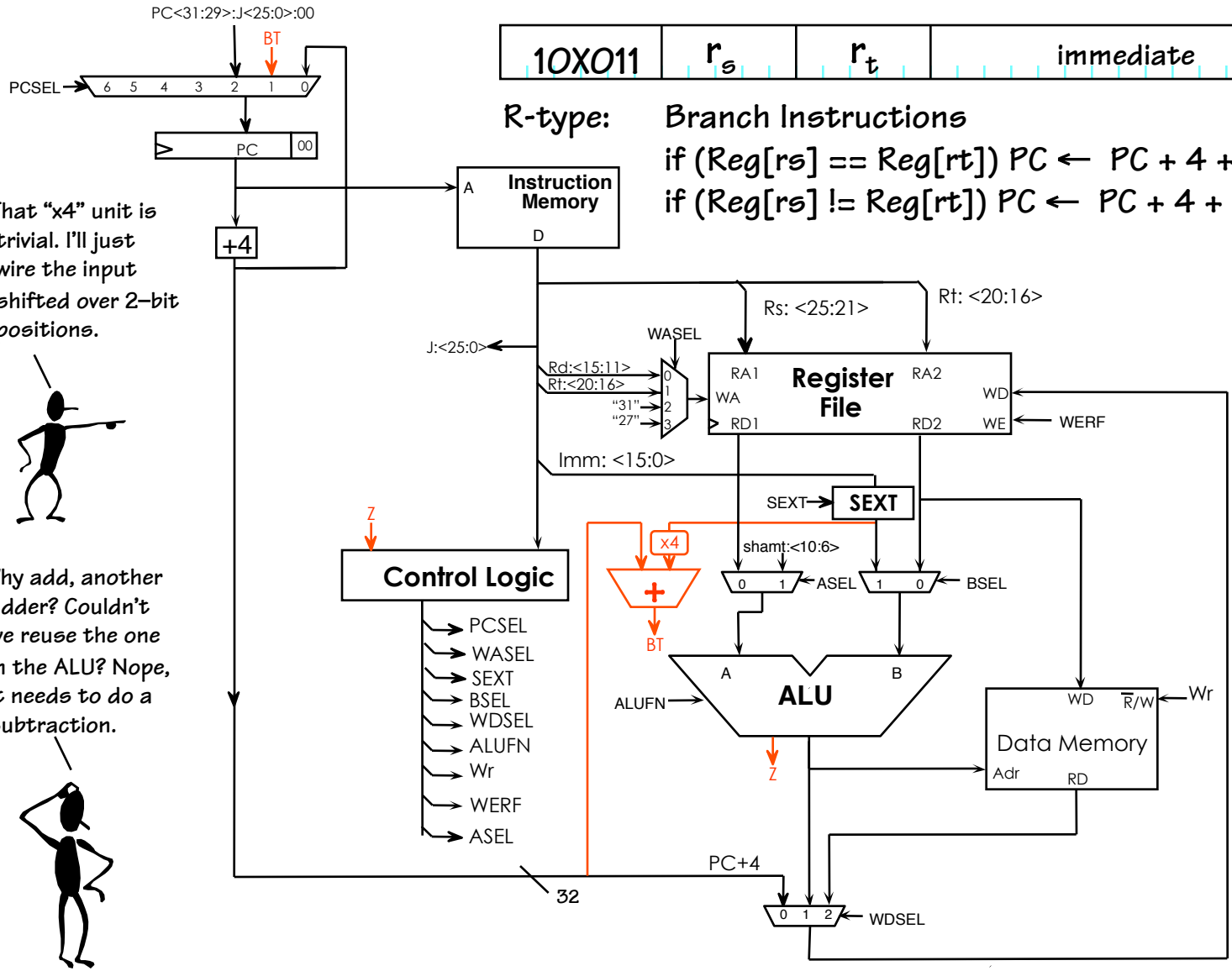
To support the JAL instruction we need to save PC+4 into register "31". This requires that we rework the mux on the WA input of the register file to allow the constant "31" to be selected and that we provide a path so that PC+4 can be written into the register file as controlled by WDSEL.

MIPS Instruction Decoding Ring

OP	000	001	010	011	100	101	110	111
000	ALU		j	jal	beq	bne		
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010								
011								
100				lw				
101				sw				
110								
111								

ALU	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010								
011	mul		div					
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

BEQ/BNE Instructions



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.

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)

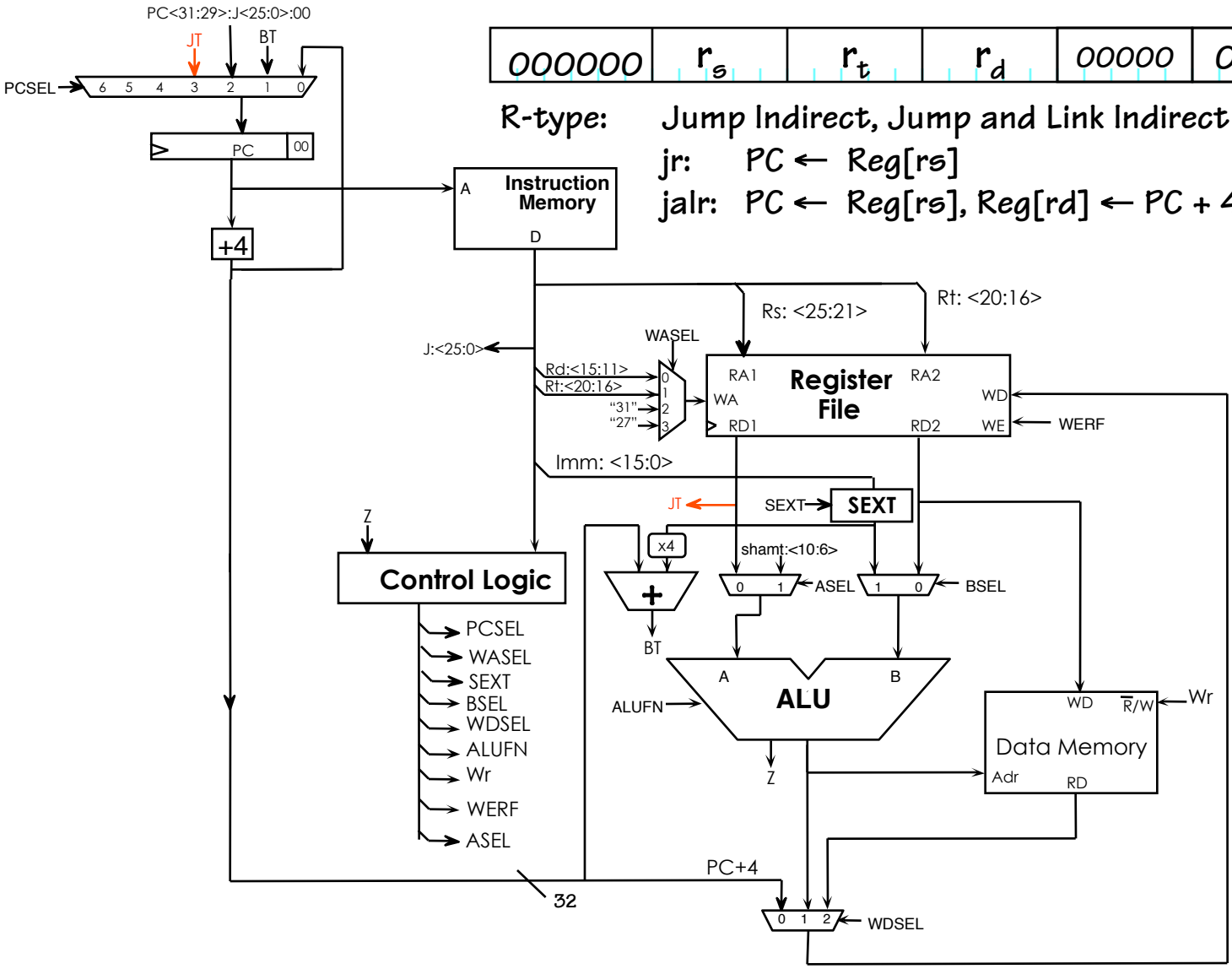
The branch instructions, BEQ and BNE, need an adder to compute the Branch Target (BT) address. They also configure the ALU to subtract the contents of r_t from r_s (ASEL=0, BSEL=0, WERF=0). Depending on the instruction and whether the result is zero, it will set PCSEL to select either BT or PC+4.

MIPS Instruction Decoding Ring

OP	000	001	010	011	100	101	110	111
000	ALU		j	jal	beq	bne		
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010								
011								
100				lw				
101				sw				
110								
111								

ALU	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010								
011	mul		div					
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

Jump Indirect Instructions



000000	r_s	r_t	r_d	00000	00100X

R-type: Jump Indirect, Jump and Link Indirect

jr: PC ← Reg[rs]

jalr: PC ← Reg[rs], Reg[rd] ← PC + 4

The instructions, JR and JALR, set the next PC using the contents of the register specified by r_s. This requires the the output of the first register port be routed to the mux used to load the next PC.

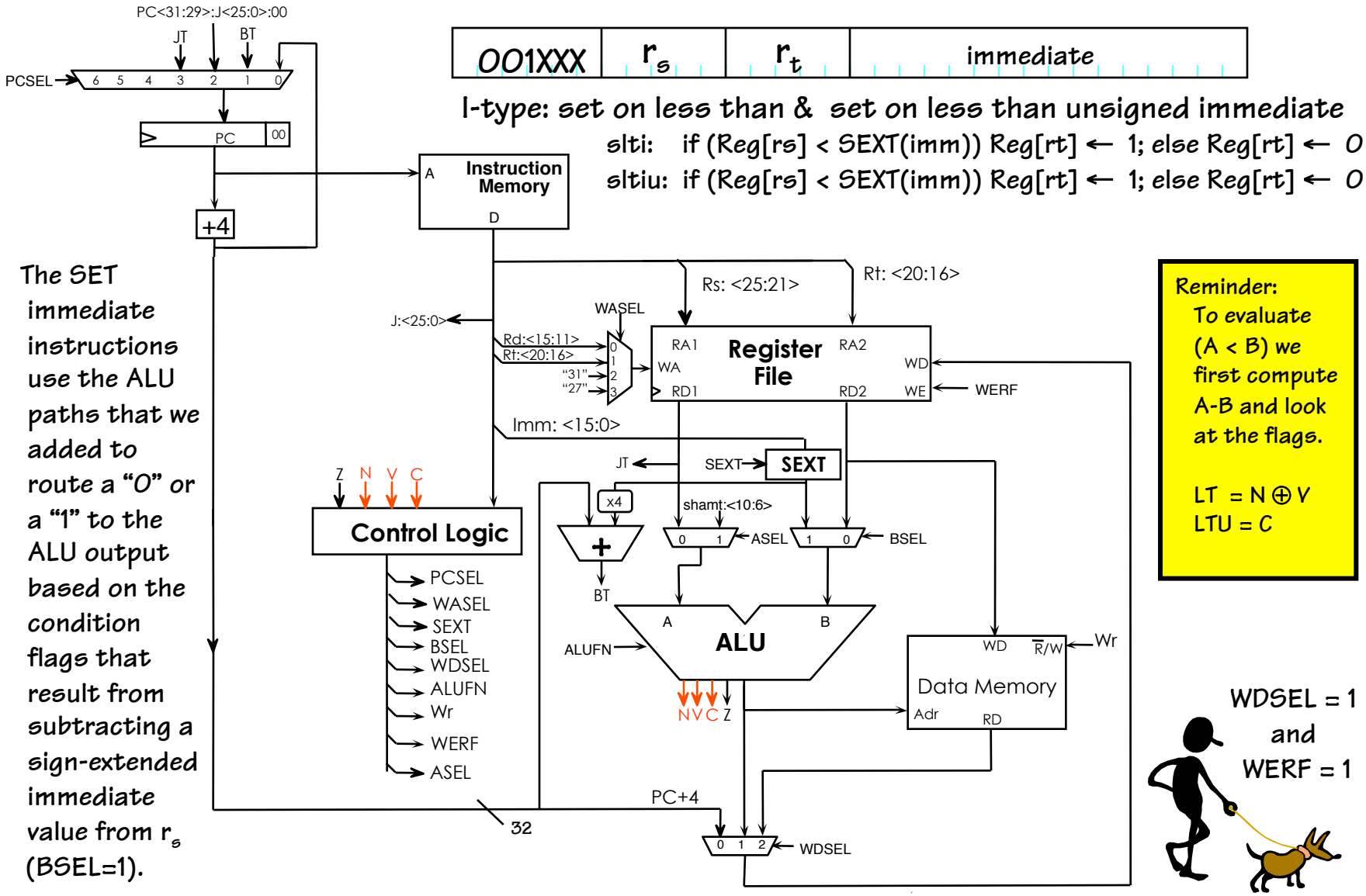
The JALR instruction reuses the connections from JAL that allow PC+4 to be stored in "31".

MIPS Instruction Decoding Ring

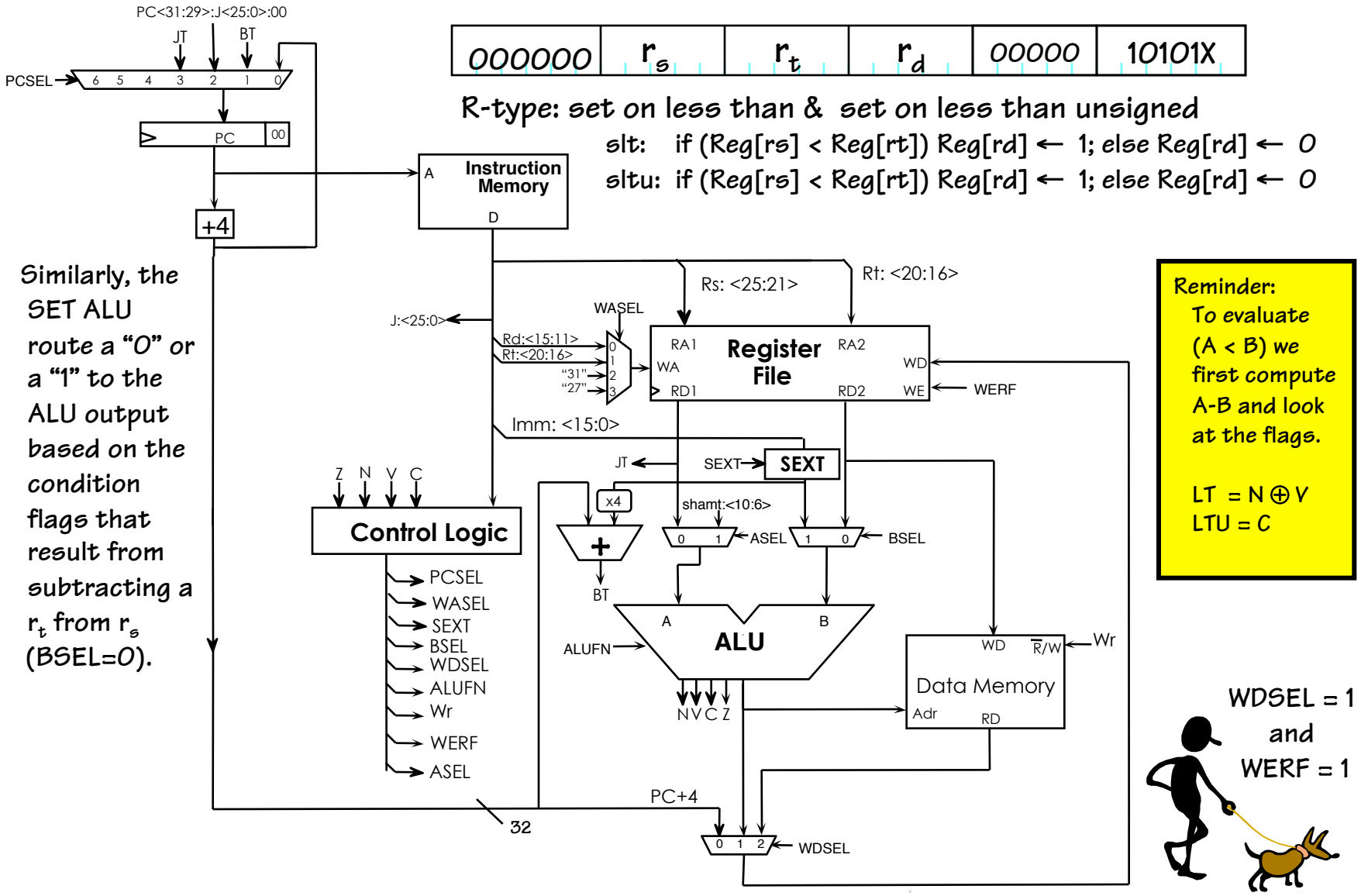
OP	000	001	010	011	100	101	110	111
000	ALU		j	jal	beq	bne		
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010								
011								
100				lw				
101				sw				
110								
111								

ALU	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010								
011	mul		div					
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

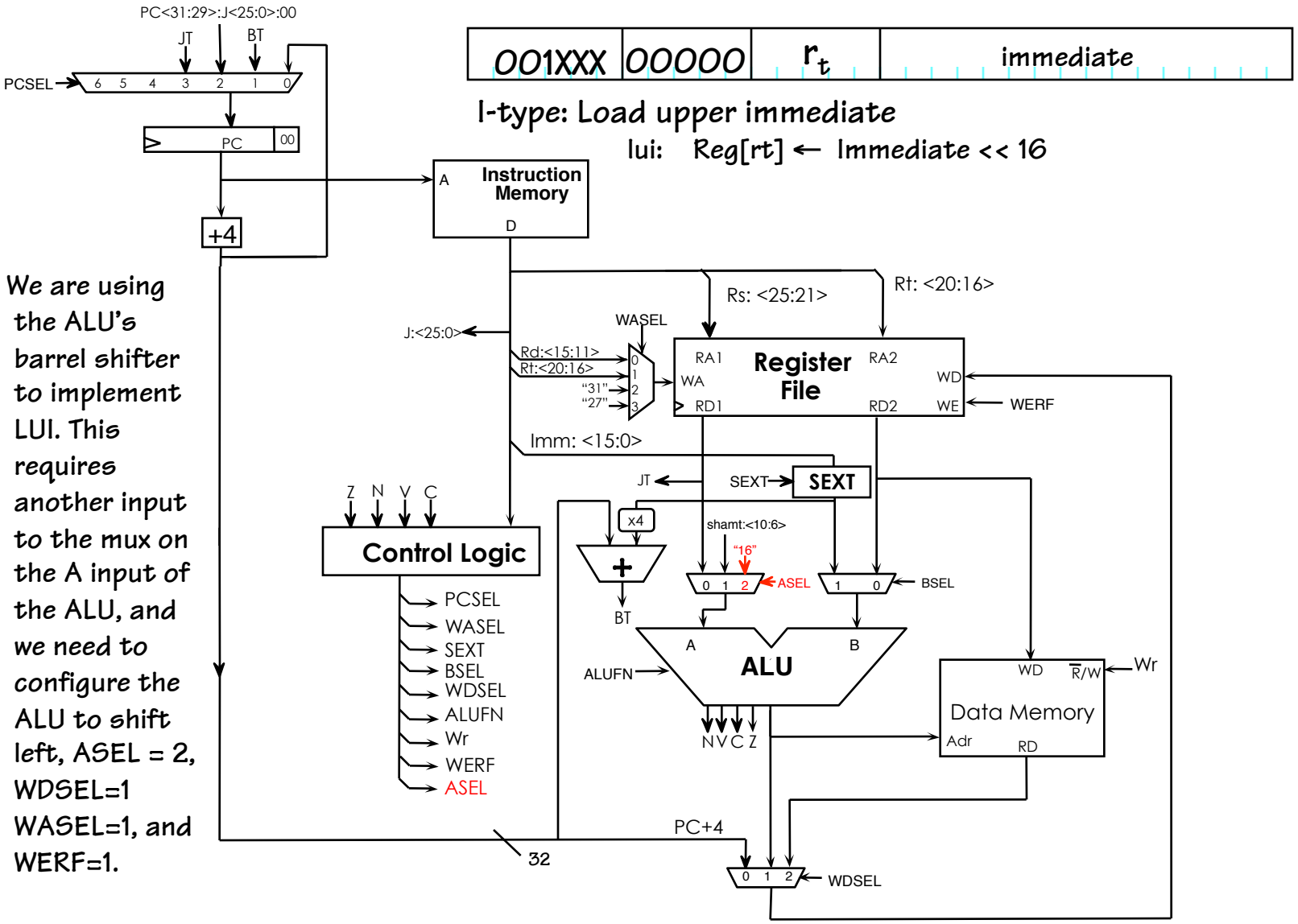
Loose Ends


PC<31:29>:J<25:0>:00
JT
BT
PCSEL
6 5 4 3 2 1 0
PC
00
+4
A
Instruction Memory
D
OO1XXX
r_s
r_t
immediate
I-type: set on less than & set on less than unsigned immediate
slti: if (Reg[rs] < SEXT(imm)) Reg[rt] ← 1; else Reg[rt] ← 0
sltiu: if (Reg[rs] < SEXT(imm)) Reg[rt] ← 1; else Reg[rt] ← 0
The SET immediate instructions use the ALU paths that we added to route a "0" or a "1" to the ALU output based on the condition flags that result from subtracting a sign-extended immediate value from r_s (BSEL=1).
Rs: <25:21>
Rt: <20:16>
J:<25:0>
WASEL
Rd:<15:11>
Rt:<20:16>
"31"
"27"
RA1
WA
RD1
Register File
RA2
WD
RD2
WE
WERF
Imm: <15:0>
JT
SEXT
SEXT
Z N V C
Control Logic
x4
shamt:<10:6>
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
Reminder:
To evaluate (A < B) we first compute A-B and look at the flags.
LT = N ⊕ V
LTU = C
WDSEL = 1
and
WERF = 1

More Loose Ends



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
We are using the ALU's barrel shifter to implement LUI. This requires another input to the mux on the A input of the ALU, and we need to configure the ALU to shift left, ASEL = 2, WDSEL=1 WASEL=1, and WERF=1.
Rs: <25:21>
Rt: <20:16>
J:<25:0>
WASEL
Rd:<15:11>
Rt:<20:16>
"31"
"27"
0
1
2
3
RA1
WA
RD1
Register File
RA2
WD
RD2
WE
WERF
Imm: <15:0>
JT
SEXT
SEXT
Z N V C
Control Logic
PCSEL
WASEL
SEXT
BSEL
WDSEL
ALUFN
Wr
WERF
ASEL
x4
+
BT
shamt:<10:6>
"16"
0 1 2
ASEL
1 0
BSEL
A
B
ALU
ALUFN
N V C Z
WD
R/W
Wr
Data Memory
Adr
RD
PC+4
32
0 1 2
WDSEL

Reset, Interrupts, and Exceptions

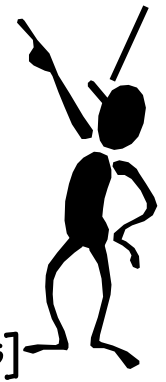
FIRST, we need some way to get our machine into a known initial state. This doesn't mean that all registers will be initialized, just that we'll know where to fetch the first instruction. We'll call this control input, RESET

We'd also like RECOVERABLE INTERRUPTS for

- FAULTS (eg, Illegal or unimplemented Instruction)
  - CPU or SYSTEM generated [synchronous]
- TRAPS & system calls (eg, read-a-character)
  - CPU generated [synchronous, caused by an instruction]

  (Implemented as an "agreed" upon Illegal instruction)
- I/O events (eg, key press)
  - externally generated [asynchronous]

These are "Software" notions of synchrony and asynchrony.



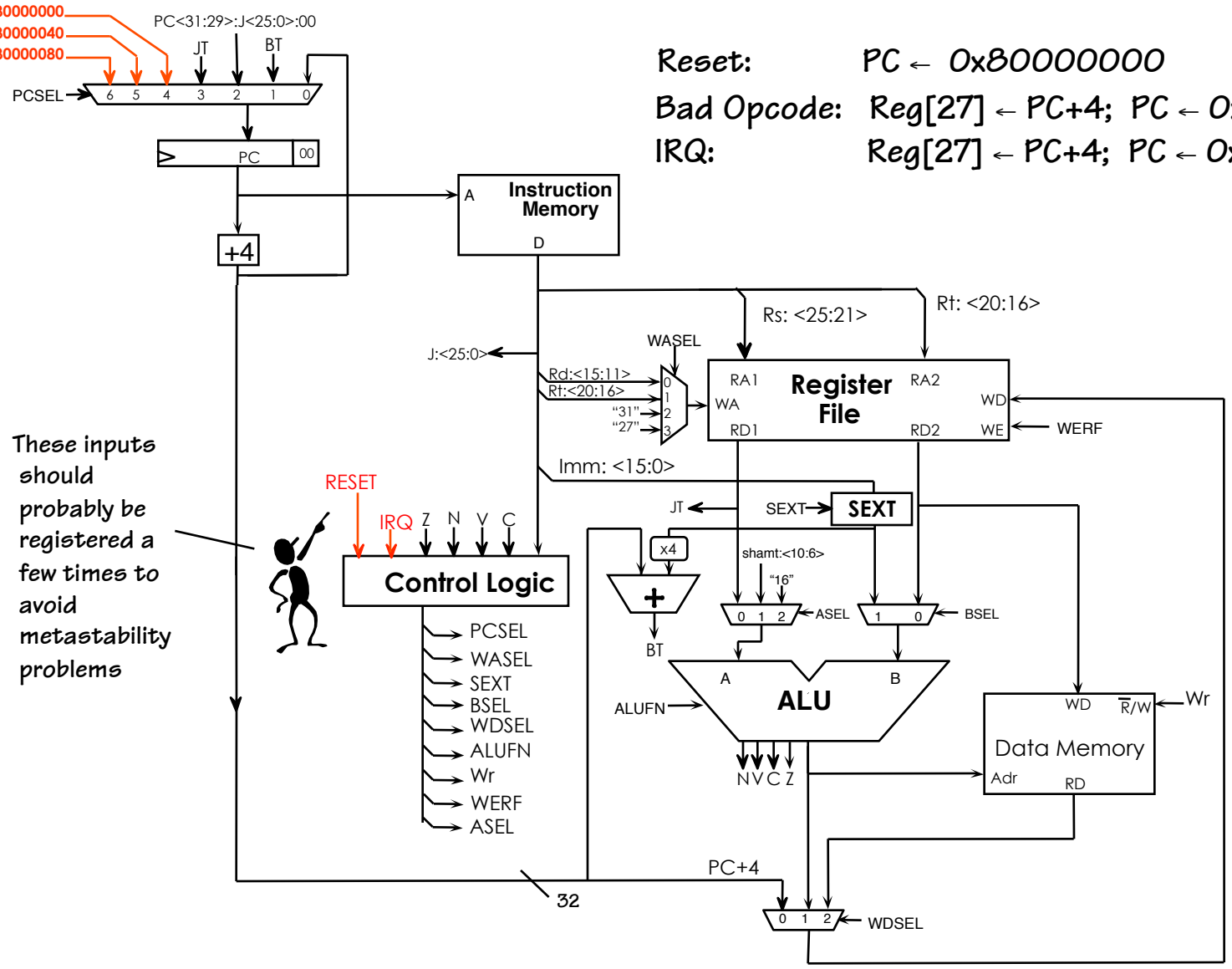
EXCEPTION GOAL: Interrupt running program, invoke exception handler, return to continue execution.

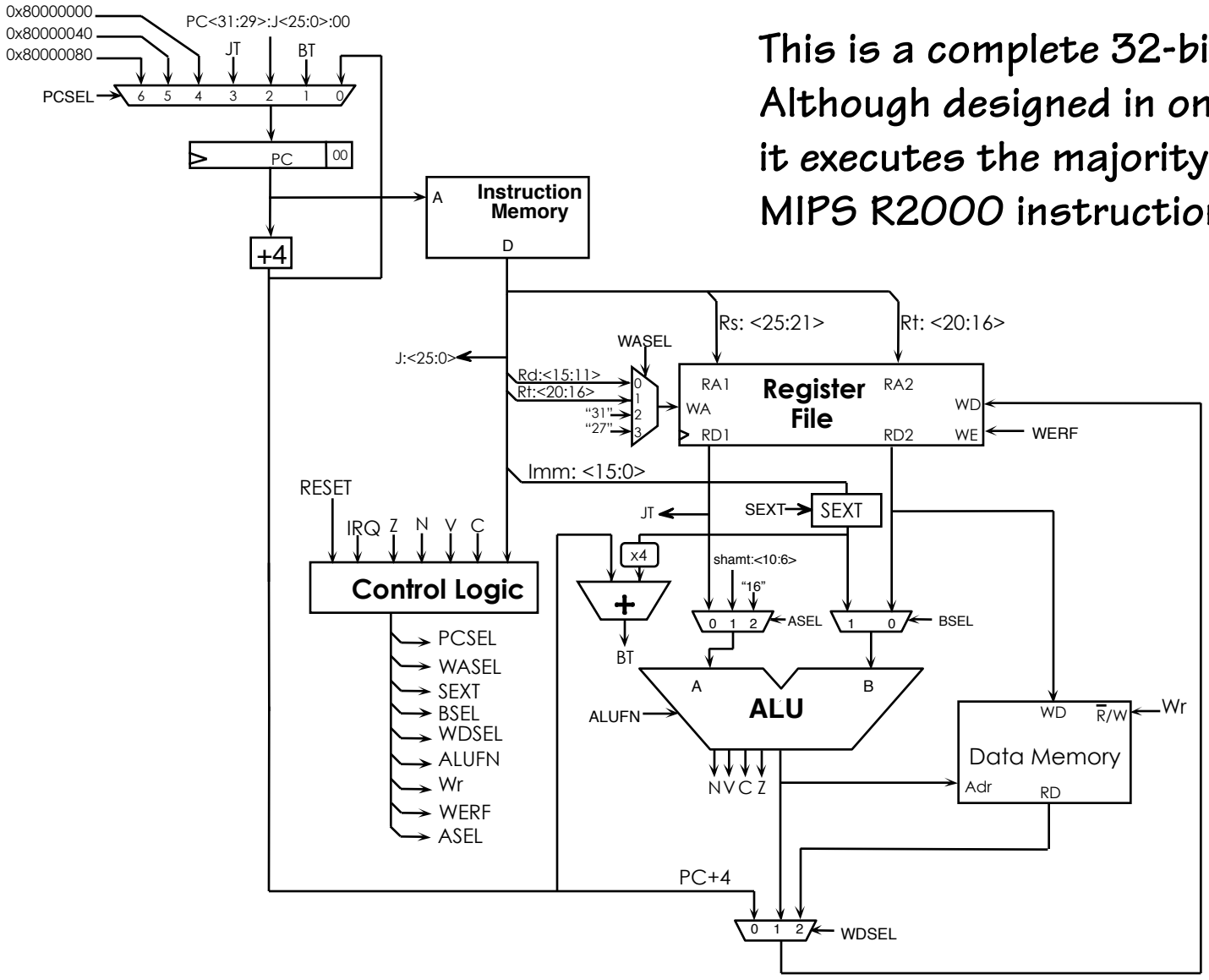
Exceptions

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				
---	1	X	X	X	X	X	4	0	0	0	0	00	0	0	0	0	0	0
---	0	1	X	X	X	X	6	0	3	0	0	00	0	0	0	0	0	0
add	0	0	X	X	0	X	0	0	0	1	0	00	0	1	0	1	0	0
sll																		
andi																		
lw	0	0	X	X	X	X	0	1	1	2	0	XX	0	1	0	1	0	1
sw	0	0	X	X	X	X	0	1	X	X	0	XX	0	1	1	0	0	1
beq	0	0	0	X	X	X	0	1	X	X	1	XX	0	1	0	0	0	0
beq	0	0	1	X	X	X	1	1	X	X	1	XX	0	1	0	0	0	0
lui																		

UNC miniMIPS