50.002 Computation Structures
Procedure Compilation Recap & Building the Beta

Oliver Weeger

2018 Term 3, Week 9, Session 1

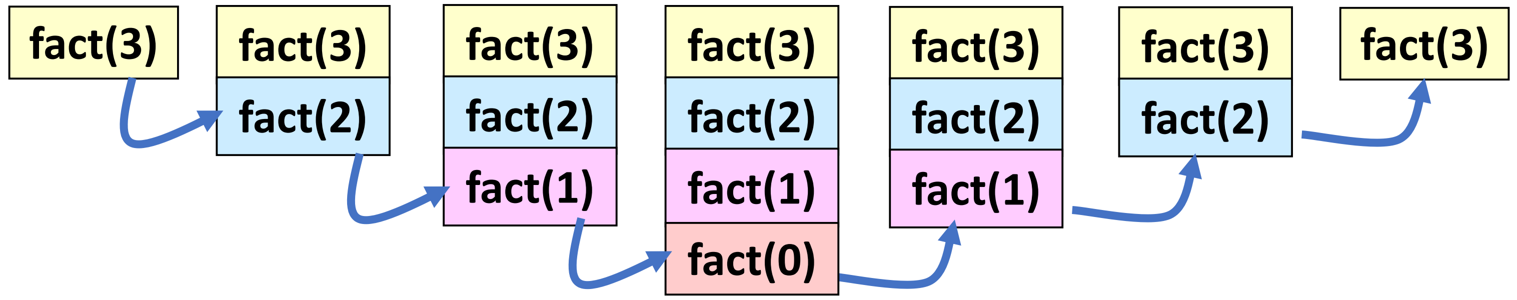
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Compiling procedures

- Call sequence: jump/branch into procedure, pass arguments, allocate local variables, return value, jump back to caller statement

```
int fact(int n) {
   if (n > 0)
      return n*fact(n-1);
   else
      return 1;
}
```



- Need memory locations for overhead (**activation record**) of procedure: **stack**
  (arguments, values in registers, local variables, return address, return value)
- Dedicated registers with pointers into stack for procedure call overhead:
  - `R29=SP` **Stack pointer**: first unused memory location in stack
  - `R28=LP` **Linkage pointer**: return address to caller
  - `R27=BP` **Base pointer**: points into stack to local variables of callee
- Stack management macros: `PUSH(Rc), POP(Rc), ALLOCATE(k), DEALLOCATE(k)`

Procedure compiling procedure & stack frames

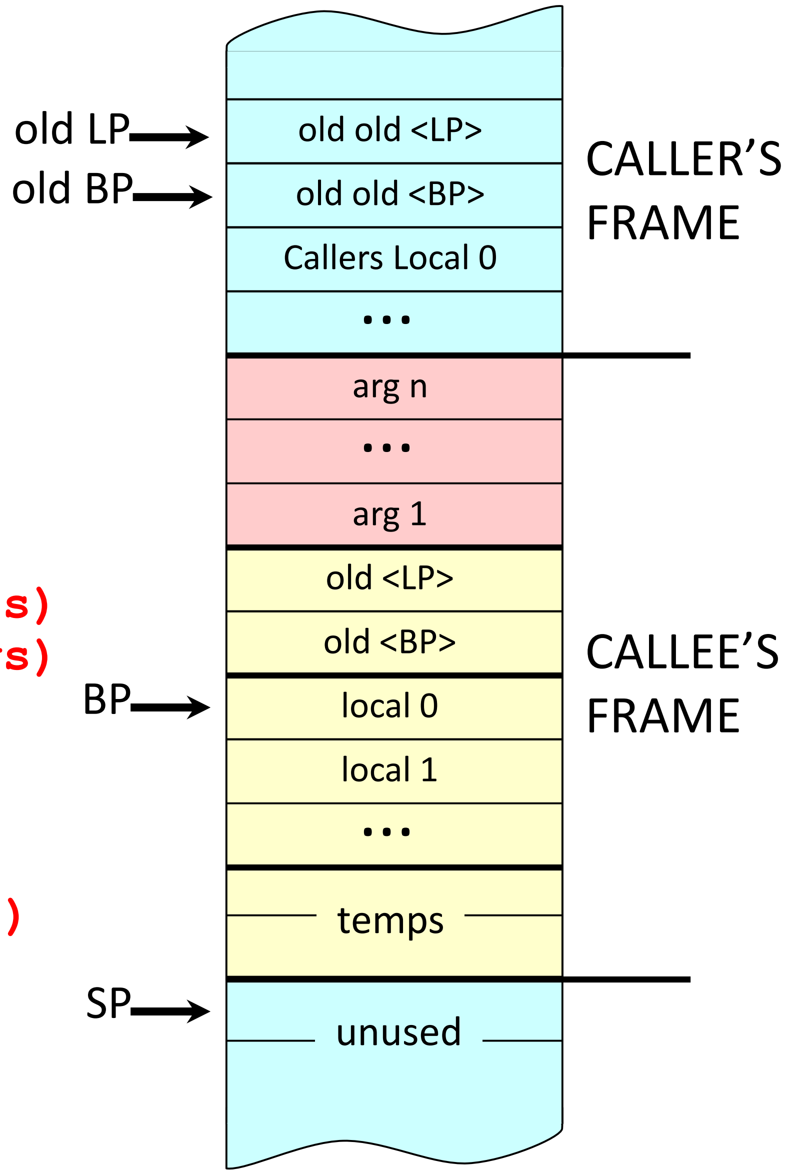
1. Caller pushes arguments onto stack, in reverse order
2. Caller branches to callee, putting return address (current PC) into LP
3. Callee pushes LP & BP, sets BP=SP, allocates local vars, pushes any used registers
4. Callee performes computation, leaving return value in R0
5. Callee pops registers, (deallocates local vars,) sets SP=BP, pops BP & LP
6. Callee branches to return address (LP)
7. Caller pops/deallocates arguments from stack

caller

```
    PUSH(arg_n)
    . . .
    PUSH(arg_1)
    BR(f,LP)
    DEALLOCATE(n)
    . . .
```

callee

```
f:  PUSH(LP)
    PUSH(BP)
    MOVE(SP,BP)
    ALLOCATE(nlocals)
    (push other regs)

    . . .
    MOVE(val,R0)

    (pop other regs)
    MOVE(BP,SP)
    POP(BP)
    POP(LP)
    JMP(LP,R31)
```



Quiz 2: Compilation

C code of function f:

```
int f(int a, int b)
{
   if(a<b)
      return [6];
   else
      return f(a-b,b);
}
```

Compiled beta assembly code:

```
F:      PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        PUSH(R1)
        PUSH(R2)
        LD(BP,-12,R0)       | #1
        LD(BP,-16,R1)
        CMPLT([4a],[4b],R2)
LL01:   BNE(R2,LL02)        | #2
        SUB(R0,R1,R0)
        PUSH(R1)
        PUSH(R0)
        BR(F,LP)            | #3
        DEALLOCATE(2)
LL02:   POP([5a])
        POP([5b])
        MOVE(BP,SP)
        POP(BP)
        POP(LP)
        JMP(LP)
```

```
OPCODE Rc=R0 Ra=BP=R27     CONST=-12
0110000000011011 1111111111110100

OPCODE Rc=R31 Ra=R2        CONST=7
0111101111100010 0000000000000111

BEQ Rc=LP=R28 Ra=R31       CONST=-19
0111011110000000 1111111111101101
```

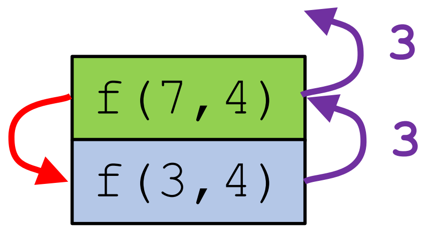
Quiz 2: Compilation

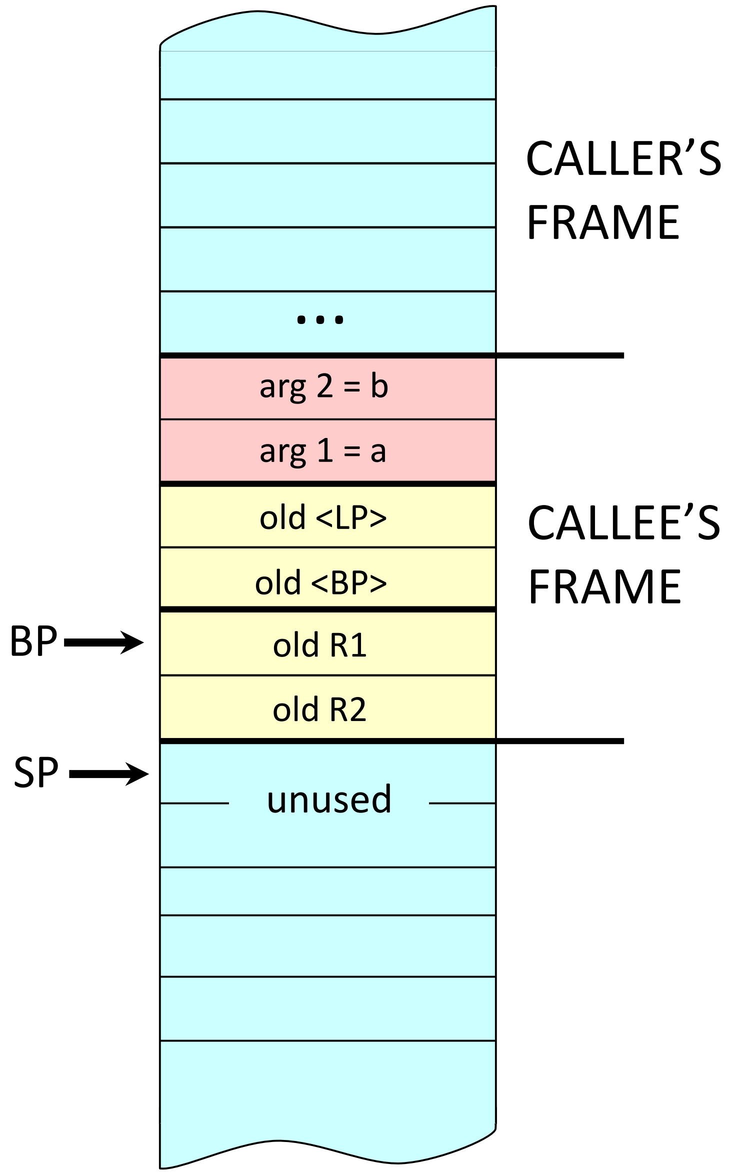
C code of function f:

```
int f(int a, int b)
{
   if(a<b)
      return [6];
   else                 a = R0
      return f(a-b,b);
}
```

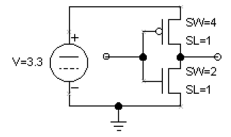
Compiled beta assembly code:

```
F:     PUSH(LP)
       PUSH(BP)
       MOVE(SP,BP)
       PUSH(R1)
       PUSH(R2)
       LD(BP,-12,R0)       a → R0
       LD(BP,-16,R1)       b → R1
       CMPLT([4a],[4b],R2)
                                R1
LL01:  BNE(R2,LL02)
       SUB(R0,R1,R0)          R0
       PUSH(R1)
       PUSH(R0)
       BR(F,LP)
       DEALLOCATE(2)

LL02:  POP([5a])            R2
       POP([5b])            R1
       MOVE(BP,SP)
       POP(BP)
       POP(LP)
       JMP(LP)
```

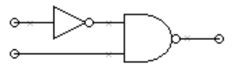




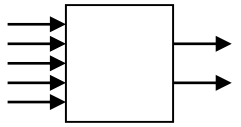
Where are we? The 50.002 roadmap



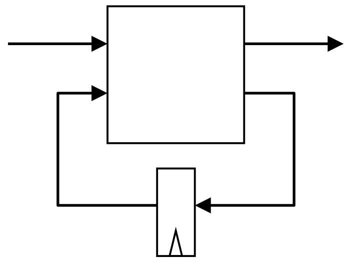
Digital abstraction, Fets & CMOS, Static discipline



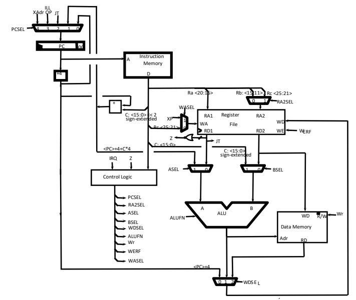
Logic gates & Boolean Algebra (AND, OR, NAND, NOR, etc.)



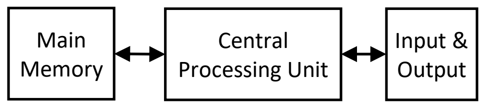
Combinational logic circuits: Truth tables, Multiplexers, ROMs



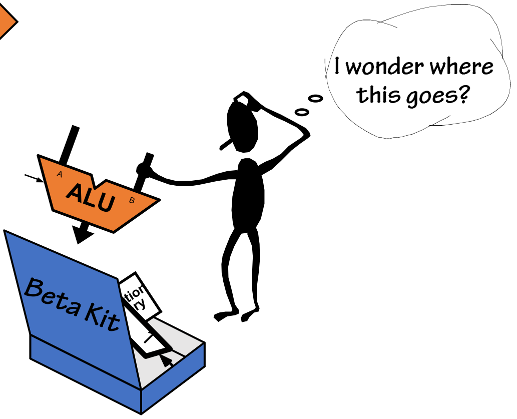
Sequential logic & Finite State Machines: Dynamic Discipline, Registers, State Transition Diagrams

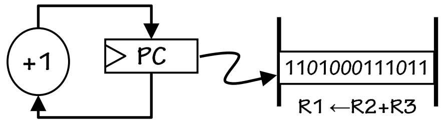


β CPU Architecture

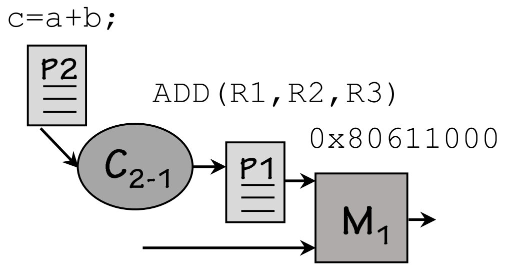


Programmability & von Neumann model, General-Purpose Computer



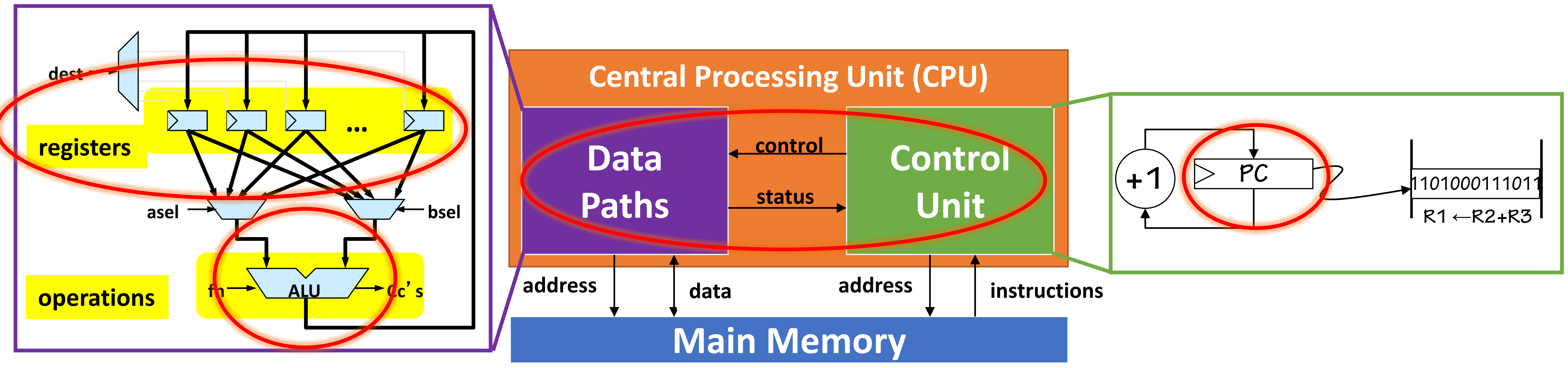


Instruction Set Architecture: β RISC, ALU (lab)

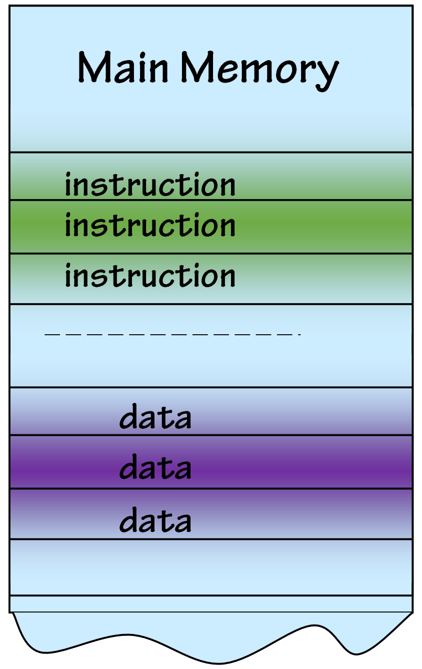


Software Abstraction, Assembler Language, C Compilation

Anatomy of von Neumann computer



- **PC**:
  Address of current instruction in memory
- **Registers**:
  Store operands and results
- **ALU**:
  Perform arithmetic & logic operations on data in registers
- **CU:**
  Interprete instructions into control signals and wire everything together into data path

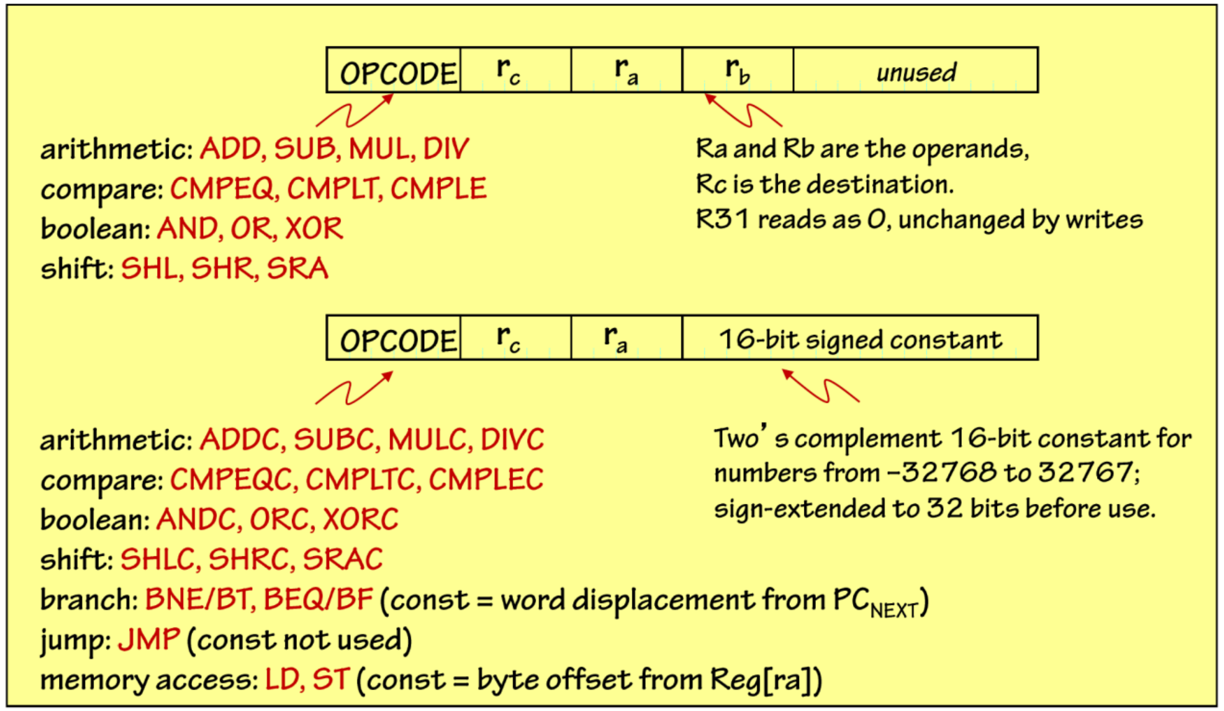


Pseudo code of FSM:

```
Reset     PC ← 0x0
Repeat    - CU reads word of instruction
            from mem[PC] & interprets it
            (generate control signals)
          - PC ← PC+1
          - ALU executes instruction
```

Building the Beta: Implementing the data paths for the ISA

Instruction Set Architecture:



OPCODE	r_c	r_a	r_b	*unused*

arithmetic: ADD, SUB, MUL, DIV
compare: CMPEQ, CMPLT, CMPLE
boolean: AND, OR, XOR
shift: SHL, SHR, SRA

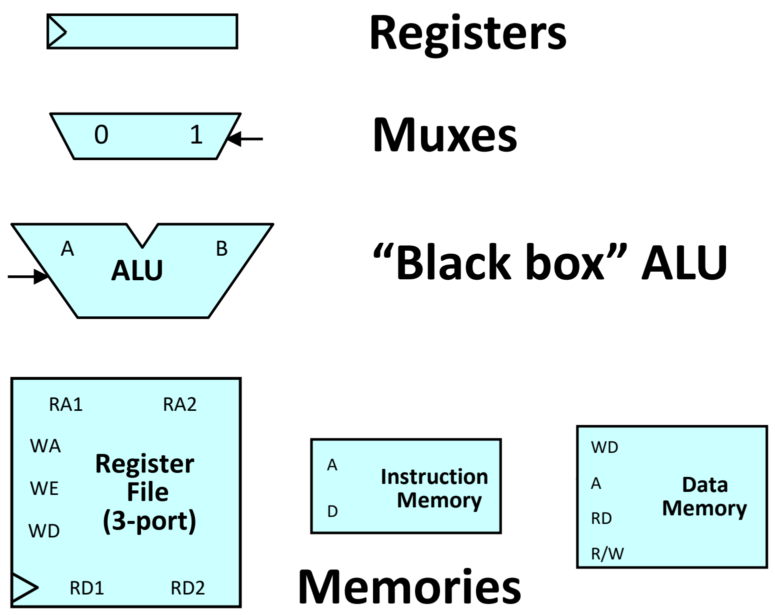
Ra and Rb are the operands,
Rc is the destination.
R31 reads as 0, unchanged by writes

OPCODE	r_c	r_a	16-bit signed constant

arithmetic: ADDC, SUBC, MULC, DIVC
compare: CMPEQC, CMPLTC, CMPLEC
boolean: ANDC, ORC, XORC
shift: SHLC, SHRC, SRAC
branch: BNE/BT, BEQ/BF (const = word displacement from PC_{NEXT})
jump: JMP (const not used)
memory access: LD, ST (const = byte offset from Reg[ra])

Two's complement 16-bit constant for numbers from −32768 to 32767; sign-extended to 32 bits before use.

Digital Circuit Components:

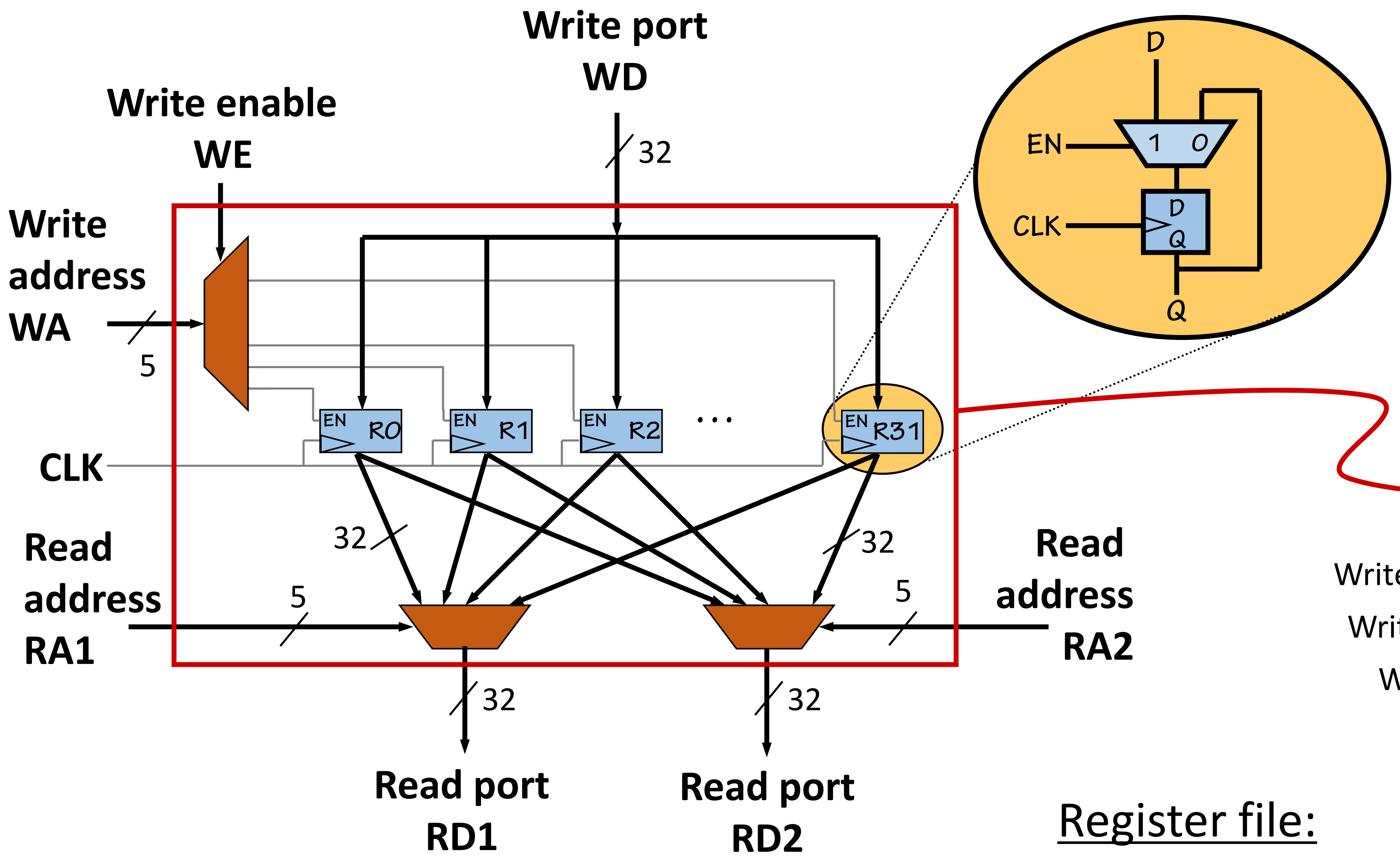


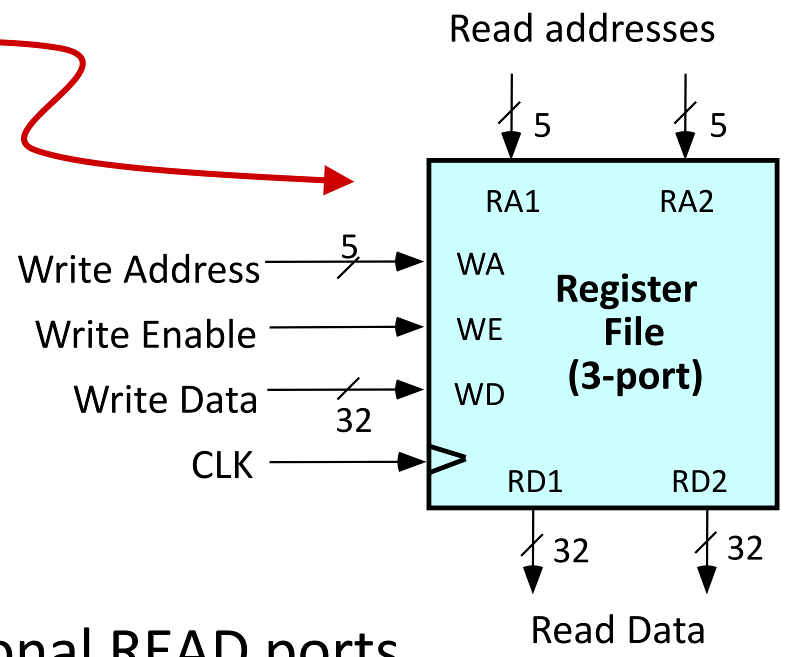
Classes of instructions:

1. ALU: OP & OPC
2. LD & ST
3. Branches
4. Exceptions

Hardware implementation of registers, PC, control logic & data paths

Register file



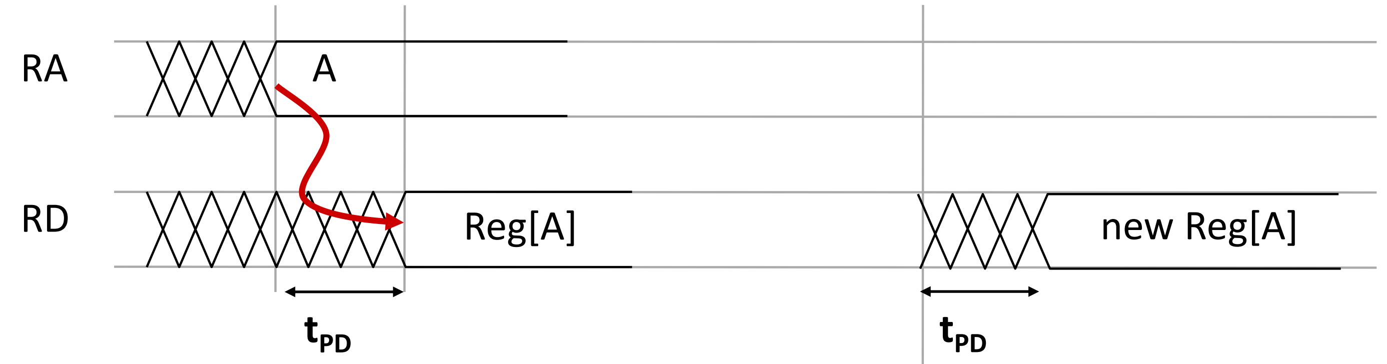


Multiplexers (combinational logic)
Registers (sequential logic)

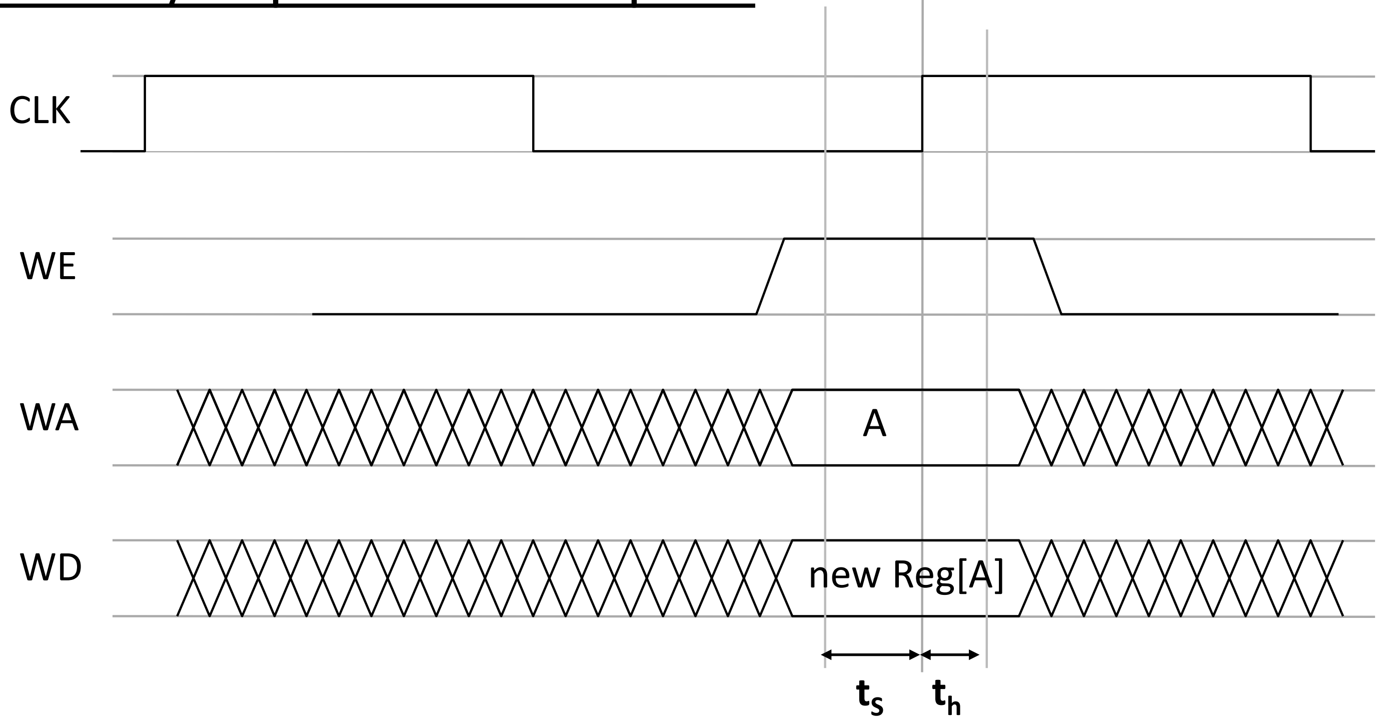
Register file:
2 combinational READ ports
1 clocked WRITE port

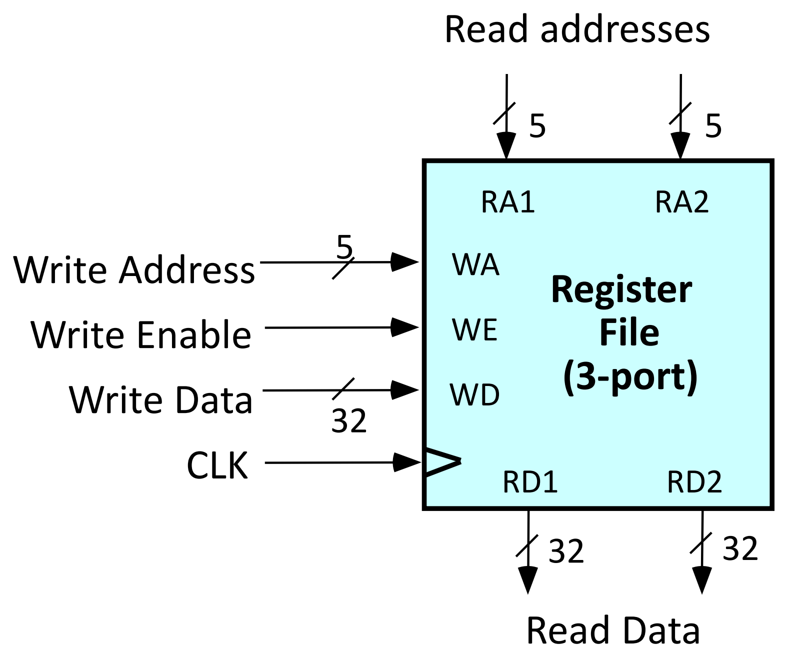
Register file: static & dynamic discipline

<u>2 combinational read ports:</u>

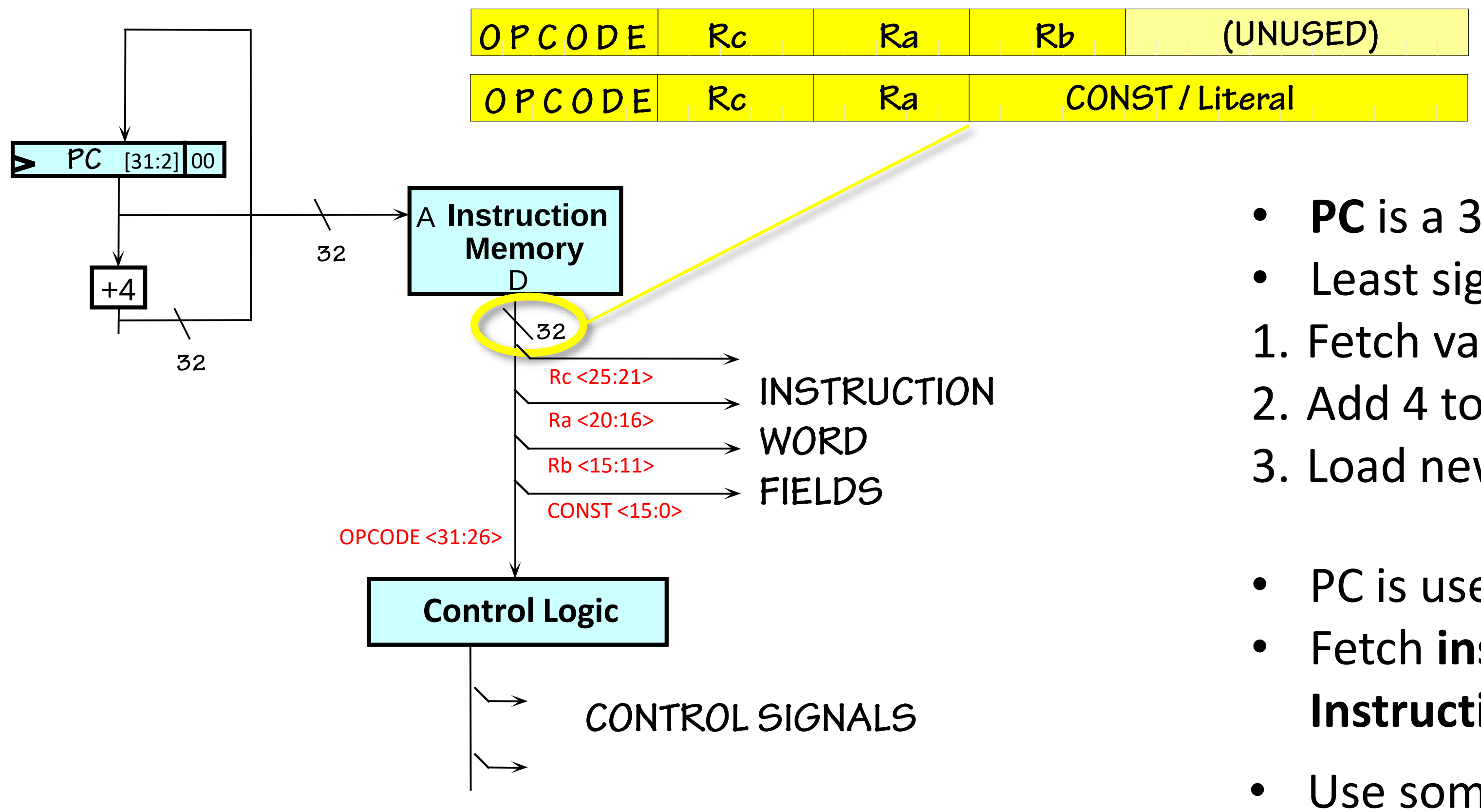


<u>1 clocked/sequential write port:</u>





Building the Beta: PC & control



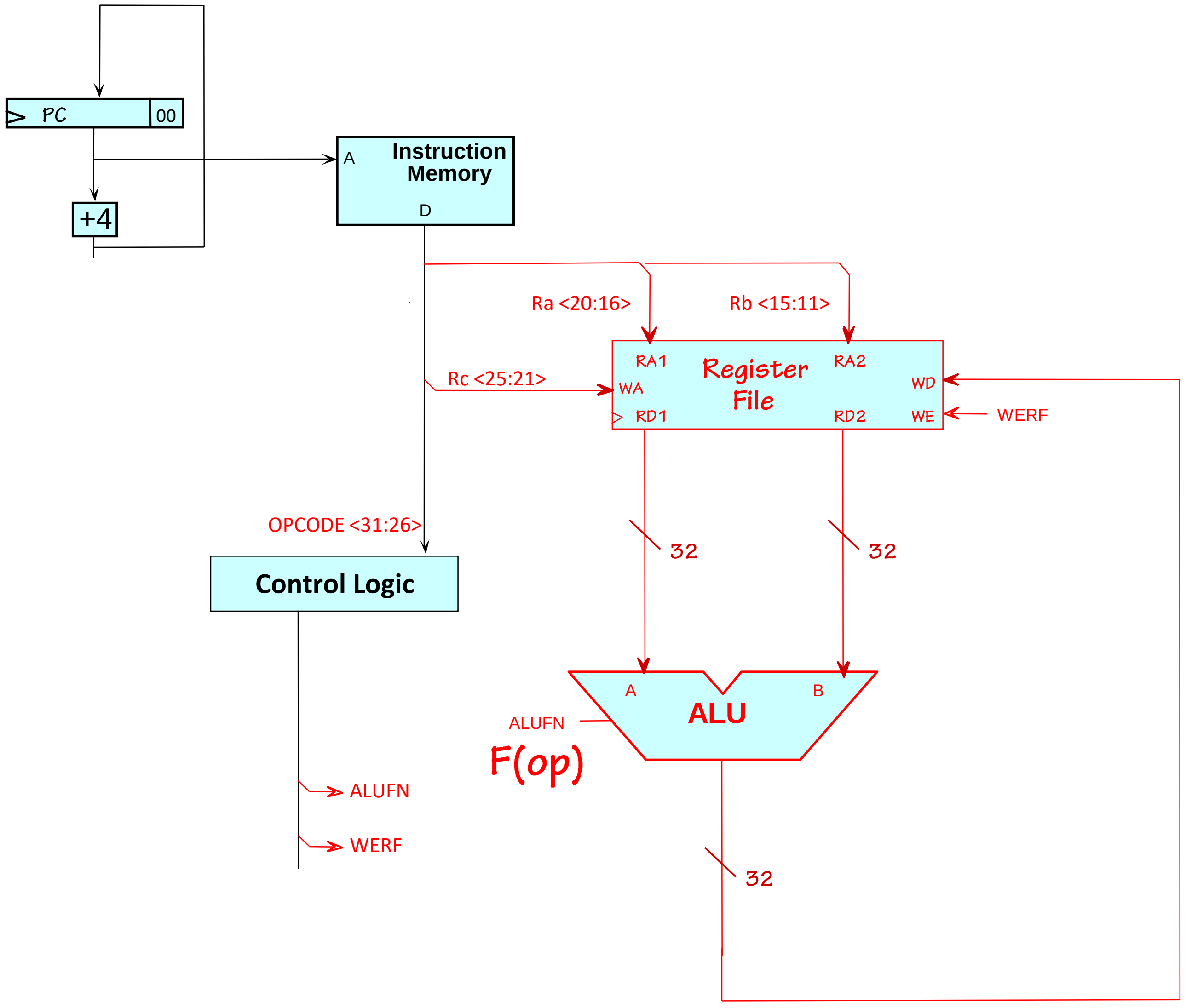
- **PC** is a 32-bit register
- Least significant 2 bits are always 0

1. Fetch value of PC
2. Add 4 to value of PC
3. Load new value into PC at end of clock cycle

- PC is used as a memory address
- Fetch **instruction** (32-bit word) from **Instruction Memory**
- Use some fields/bits directly (registers Rc, Ra, Rb; 16-bit constant)
- OPCODE (6 most significant bits) is used by **Control Logic** to creat control signals for ALU and other parts of β architecture

Data path for ALU OP instruction

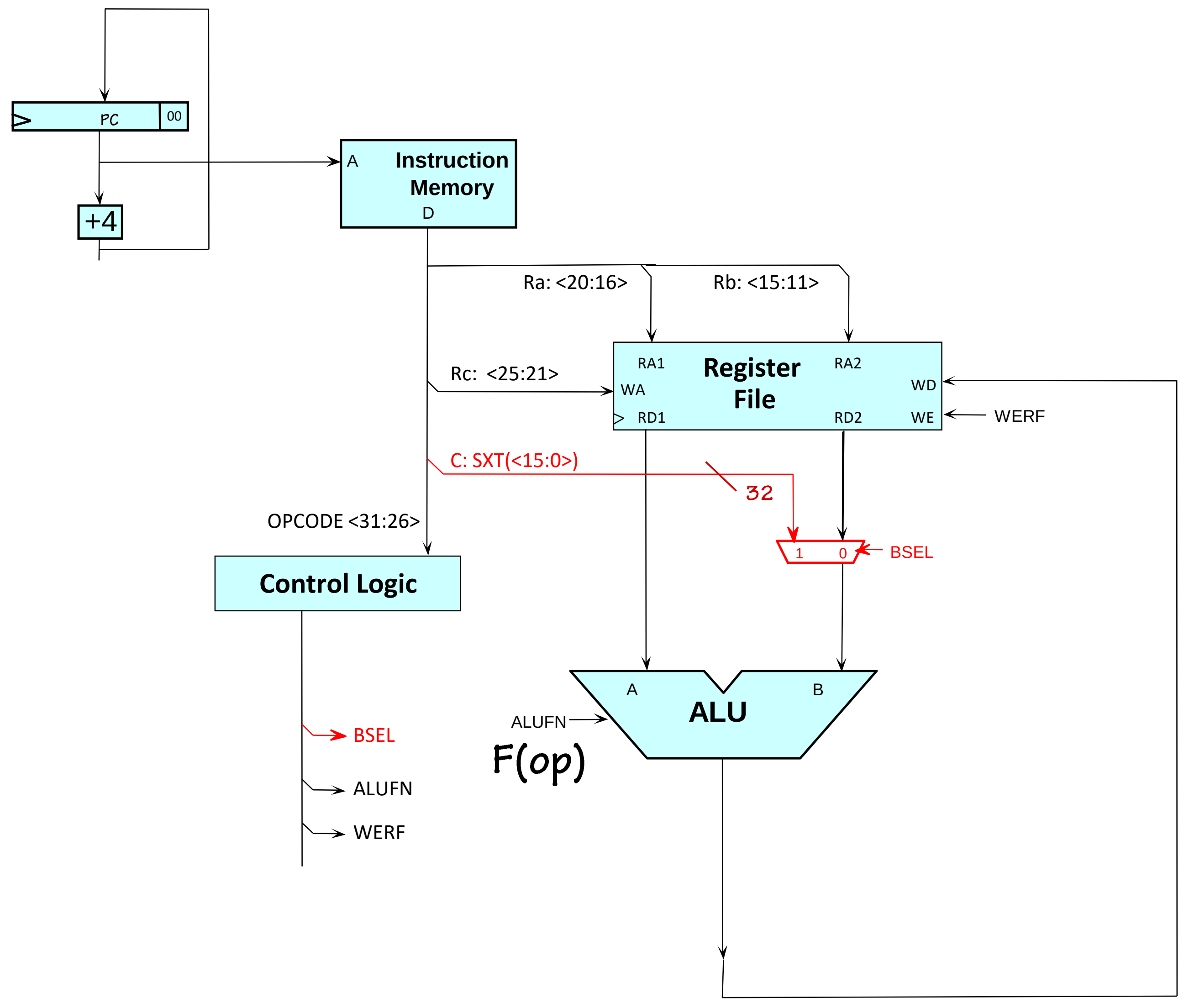
	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)									
WERF	1									



1 0 X X X X	Rc	Ra	Rb	(UNUSED)

Operate class: Reg[Rc] ← Reg[Ra] op Reg[Rb]

Data path for ALU OPC instruction

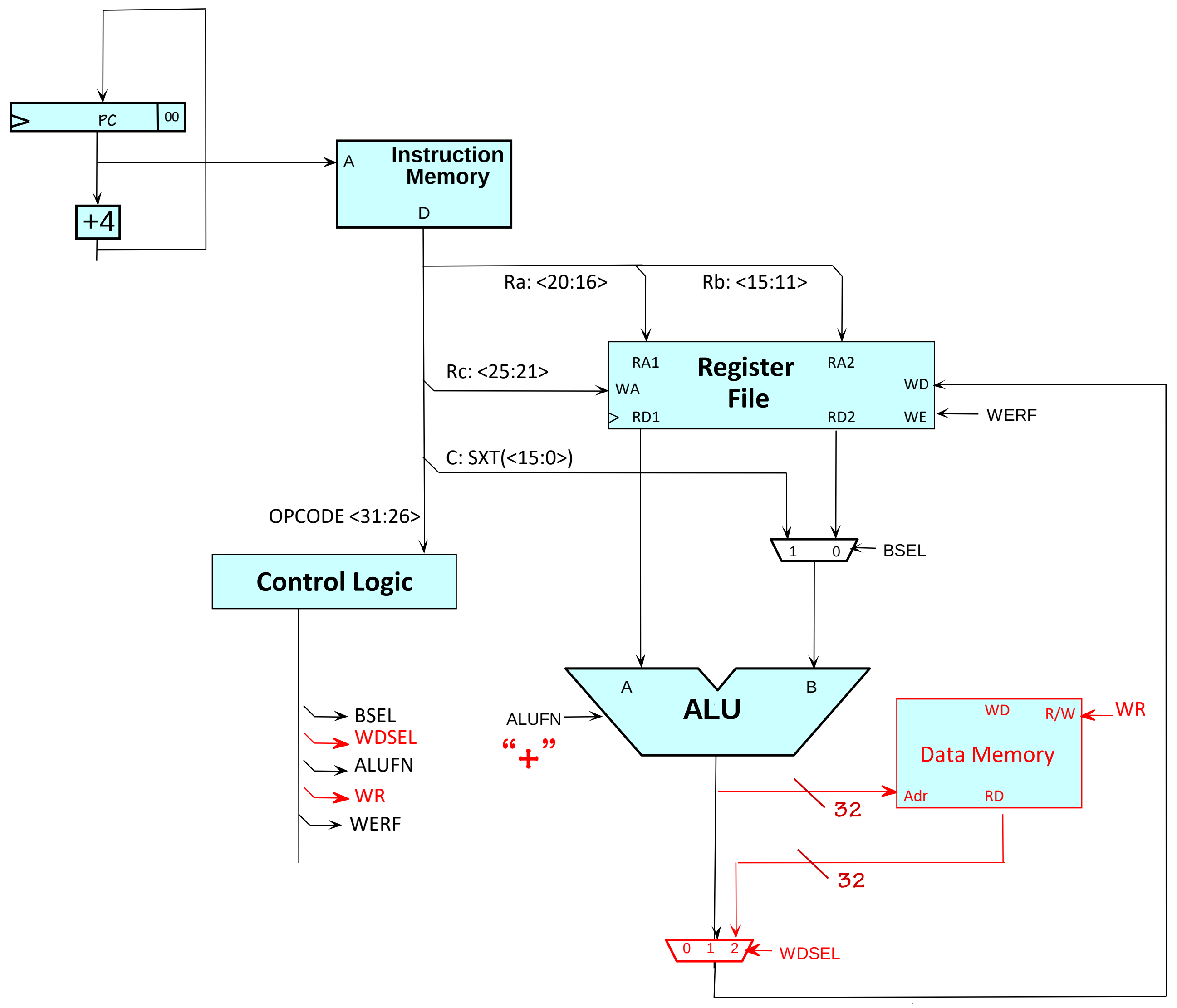


	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)	F(op)								
WERF	1	1								
BSEL	0	1								

1 1 X X X X	Rc	Ra	Literal C (signed)

Operate class: Reg[Rc] ← Reg[Ra] op SXT(C)

Data path for LD instruction

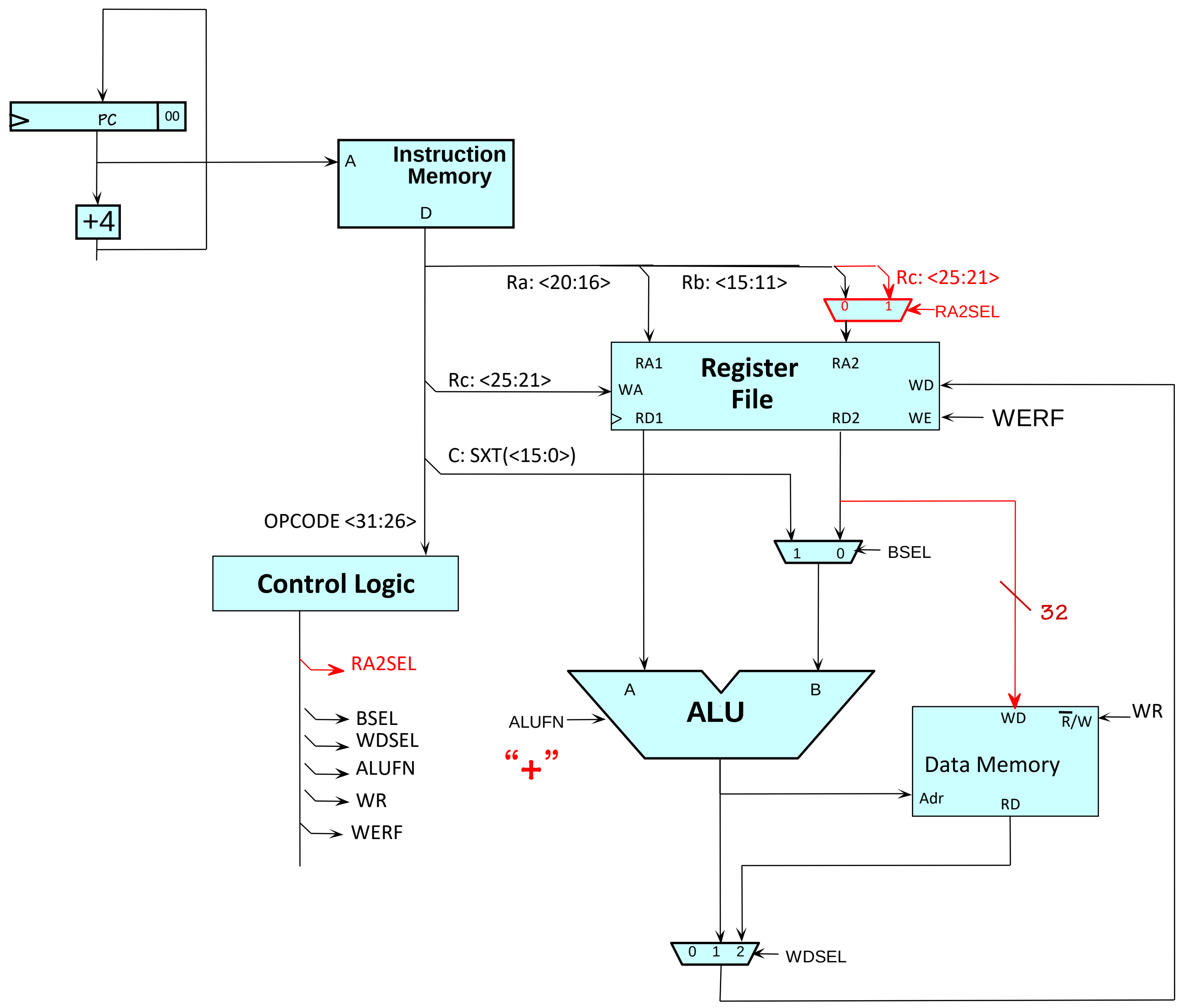


	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)	F(op)	"+"							
WERF	1	1	1							
BSEL	0	1	1							
WDSEL	1	1	2							
WR	0	0	0							

011000	Rc	Ra	Literal C (signed)

LD: Reg[Rc] ← Mem[Reg[Ra]+SXT(C)]

Data path for ST instruction

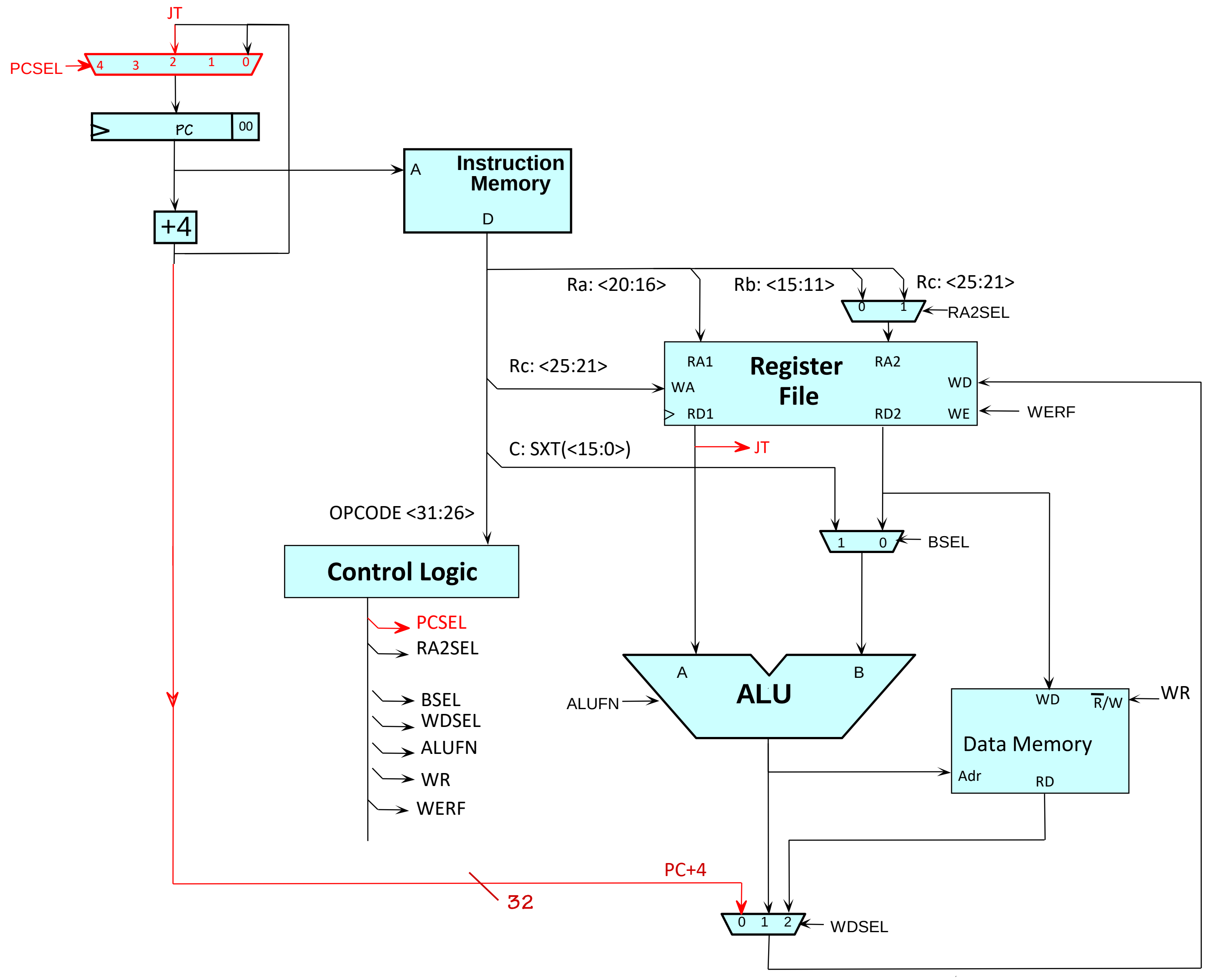


	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)	F(op)	"+"	"+"						
WERF	1	1	1	0						
BSEL	0	1	1	1						
WDSEL	1	1	2	--						
WR	0	0	0	1						
RA2SEL	0	--	--	1						

0 1 1 0 0 1	Rc	Ra	Literal C (signed)

ST: Mem[Reg[Ra]+SXT(C)] $\leftarrow$ Reg[Rc]

Data path for JMP instruction

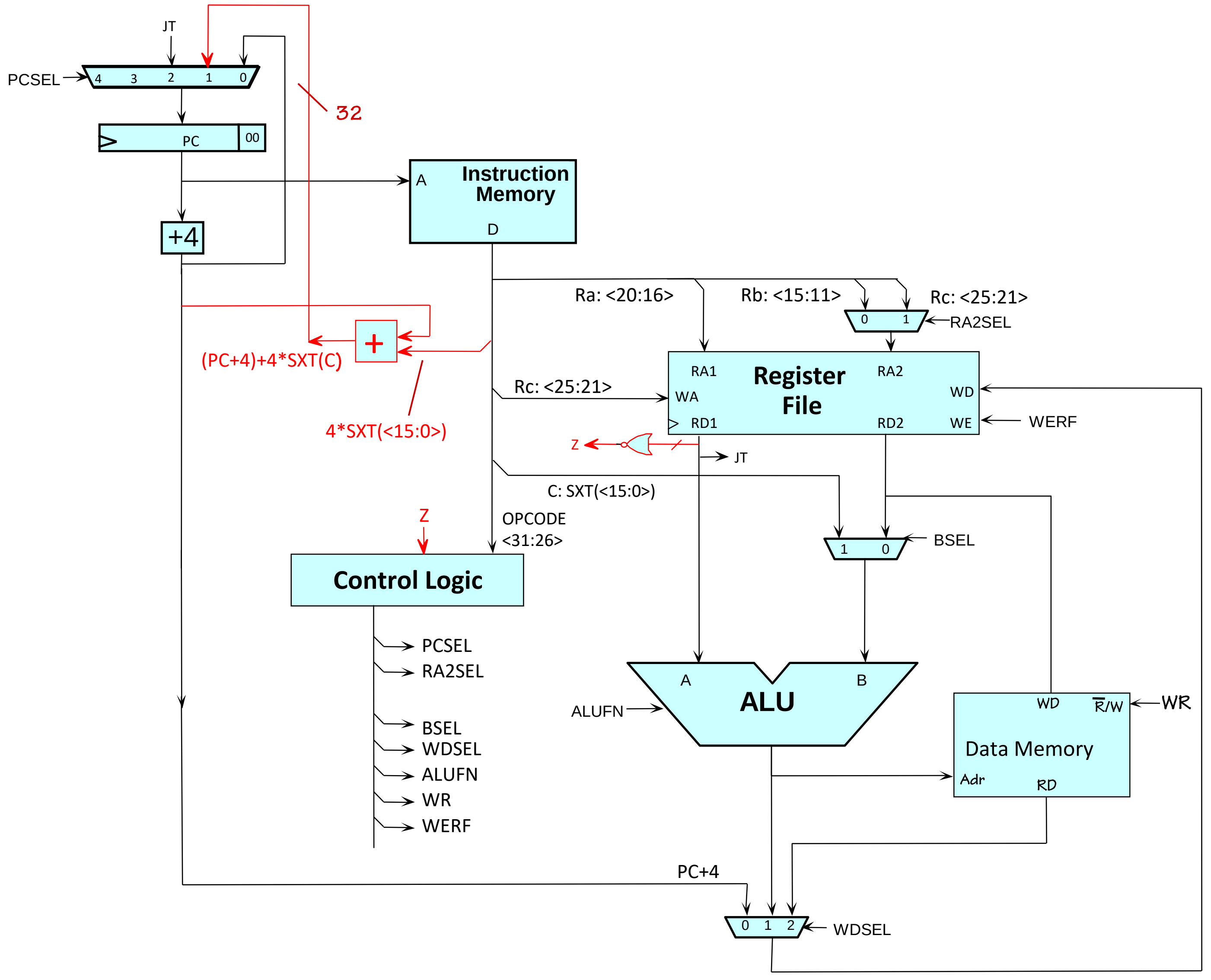


	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)	F(op)	"+"	"+"	--					
WERF	1	1	1	0	1					
BSEL	0	1	1	1	--					
WDSEL	1	1	2	--	0					
WR	0	0	0	1	0					
RA2SEL	0	--	--	1	--					
PCSEL	0	0	0	0	2					

011011	Rc	Ra	Literal C (signed)

JMP: Reg[Rc] ← PC+4;
PC ← Reg[Ra]

Data path for BEQ & BNE instruction



	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)	F(op)	"+"	"+"	--	--	--			
WERF	1	1	1	0	1	1	1			
BSEL	0	1	1	1	--	--	--			
WDSEL	1	1	2	--	0	0	0			
WR	0	0	0	1	0	0	0			
RA2SEL	0	--	--	1	--	--	--			
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1			

0 1 1 1 0 1	Rc	Ra	Literal C (signed)

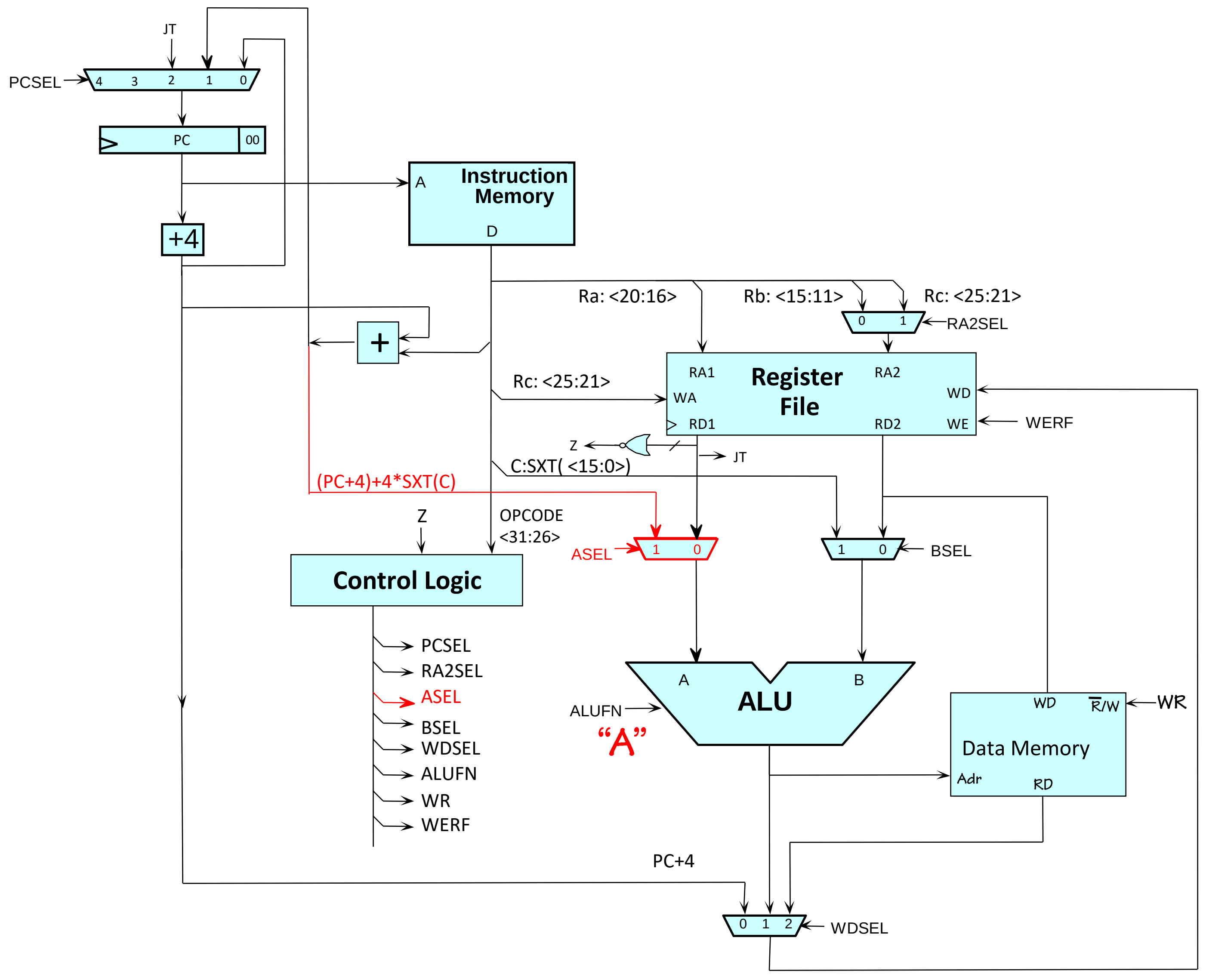
BEQ: Reg[Rc] ← PC+4;
if (Reg[Ra]==0)
PC ← (PC+4)+4*SXT(C)

0 1 1 1 1 0	Rc	Ra	Literal C (signed)

BNE: Reg[Rc] ← PC+4;
if (Reg[Ra]≠0)
PC ← (PC+4)+4*SXT(C)

Data path for LDR instruction

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN



	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)	F(op)	"+"	"+"	--	--	--	"A"		
WERF	1	1	1	0	1	1	1	1		
BSEL	0	1	1	1	--	--	--	--		
WDSEL	1	1	2	--	0	0	0	2		
WR	0	0	0	1	0	0	0	0		
RA2SEL	0	--	--	1	--	--	--	--		
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0		
ASEL	0	0	0	0	--	--	--	1		

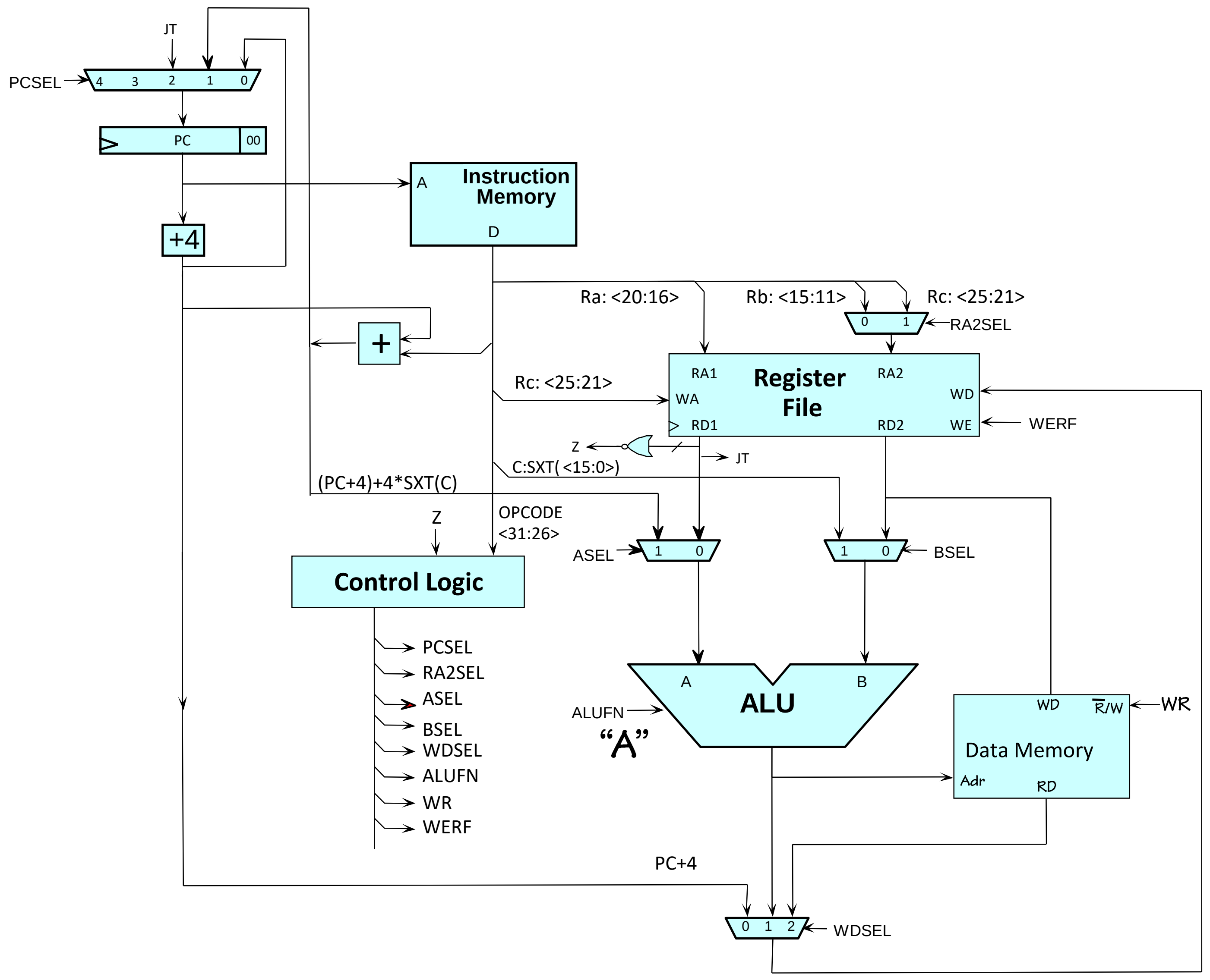
0 1 1 1 1 1	Rc	Ra	Literal C (signed)

LDR: Reg[Rc] ← Mem[(PC + 4) + 4*SXT(C)]

Used to reference addresses & other large constants

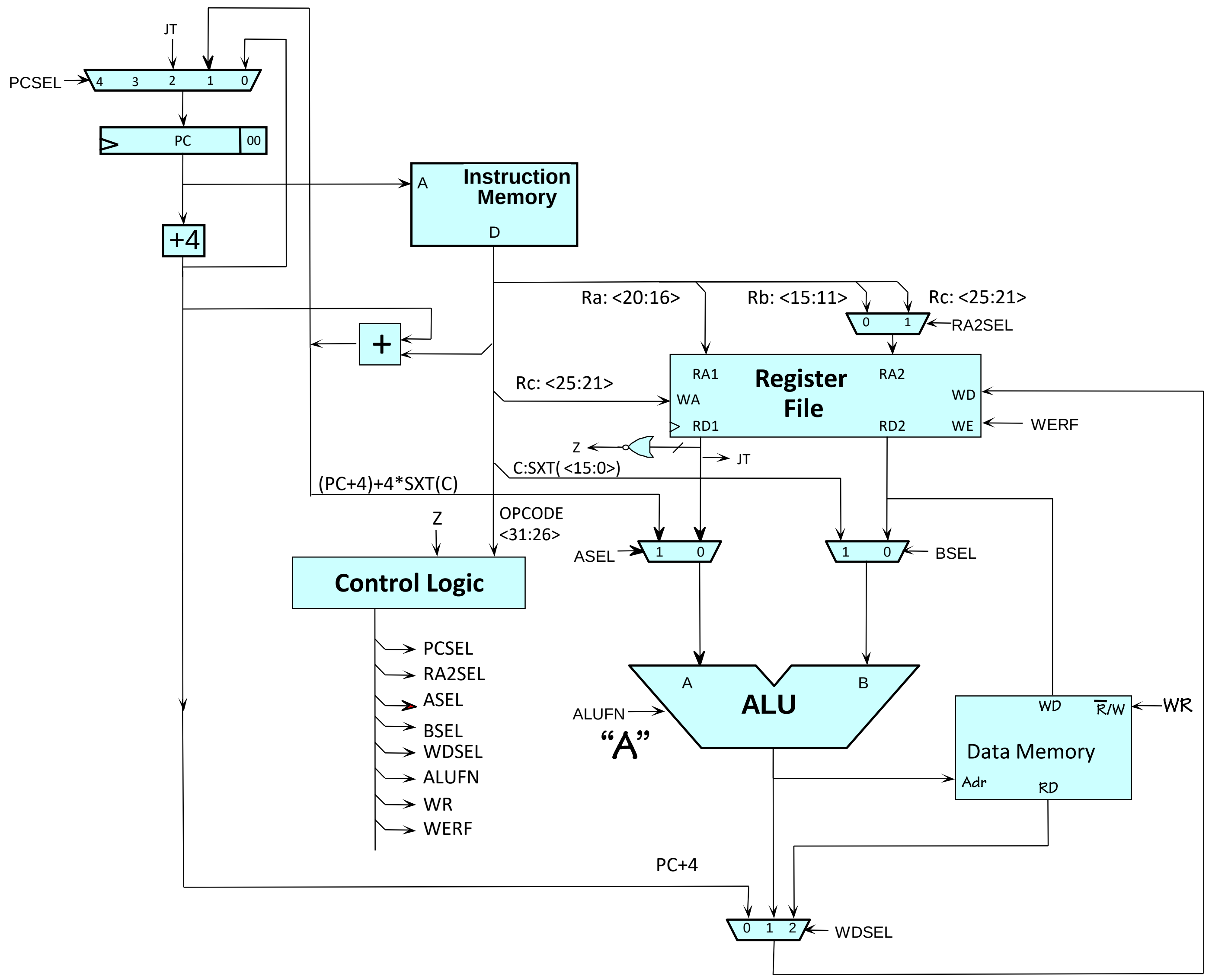
```
C:      X = X * 123456;

BETA:
        LD(X, r0)
        LDR(c1, r1)
        MUL(r0, r1, r0)
        ST(r0, X)
        ...
c1:     LONG(123456)
```

The Beta CPU data path (so far)



	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)	F(op)	"+"	"+"	--	--	--	"A"		
WERF	1	1	1	0	1	1	1	1		
BSEL	0	1	1	1	--	--	--	--		
WDSEL	1	1	2	--	0	0	0	2		
WR	0	0	0	1	0	0	0	0		
RA2SEL	0	--	--	1	--	--	--	--		
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0		
ASEL	0	0	0	0	--	--	--	1		

Exercise: extending the Beta (1)



	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	LDX
ALUFN	F(op)	F(op)	"+"	"+"	--	--	--	"A"	"+"
WERF	1	1	1	0	1	1	1	1	1
BSEL	0	1	1	1	--	--	--	--	0
WDSEL	1	1	2	--	0	0	0	2	2
WR	0	0	0	1	0	0	0	0	0
RA2SEL	0	--	--	1	--	--	--	--	0
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0	0
ASEL	0	0	0	0	--	--	--	1	0

How could we add an instruction

`LDX(R0,R1,R2)`

as a short-cut for

`ADD(R1,R0,R0)`

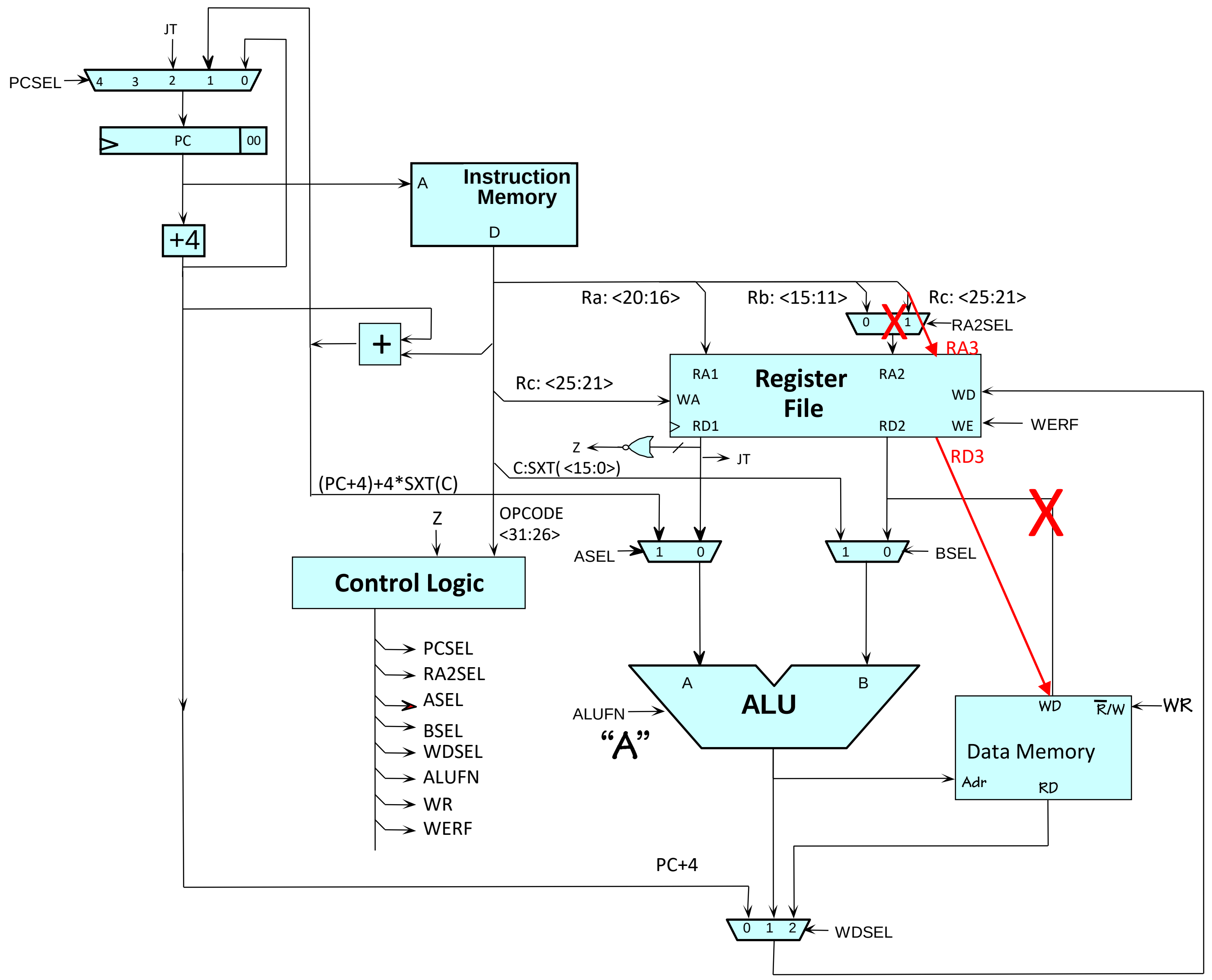
`LD(R0,0,R2)` ?

Register-transfer language expression:

Reg[Rc] ← Mem[Reg[Ra] + Reg[Rb]]

`LDX(Ra,Rb,Rc)`

Exercise: extending the Beta (2)



	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	STX
ALUFN	F(op)	F(op)	"+"	"+"	--	--	--	"A"	"+"
WERF	1	1	1	0	1	1	1	1	0
BSEL	0	1	1	1	--	--	--	--	0
WDSEL	1	1	2	--	0	0	0	2	0
WR	0	0	0	1	0	0	0	0	1
RA2SEL	0	--	--	1	--	--	--	--	0
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0	0
ASEL	0	0	0	0	--	--	--	1	0

How could we add an instruction

`STX(R2,R0,R1)`

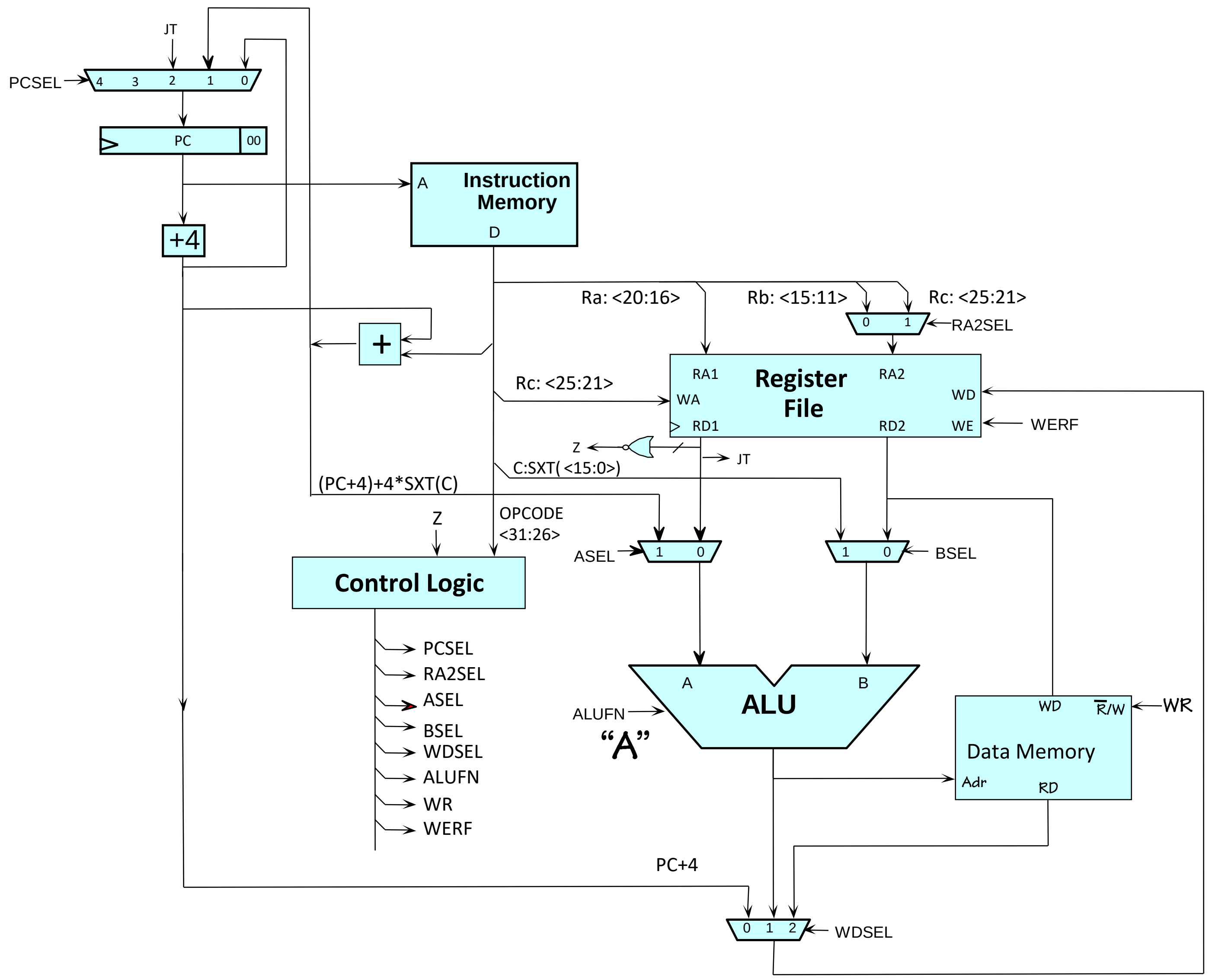
as a short-cut for

`ADD(R1,R0,R0)`
`ST(R2,0,R0)` ?

Register-transfer language expression:
Mem[Reg[Ra] + Reg[Rb]] ← Reg[Rc]
`STX(Rc,Rb,Ra)`

Must amend data path & register file!
Register file needs another RA/RD port!
Could eliminate RA2SEL mux!

Summary: The β CPU data path (so far)



	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)	F(op)	"+"	"+"	--	--	--	"A"		
WERF	1	1	1	0	1	1	1	1		
BSEL	0	1	1	1	--	--	--	--		
WDSEL	1	1	2	--	0	0	0	2		
WR	0	0	0	1	0	0	0	0		
RA2SEL	0	--	--	1	--	--	--	--		
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0		
ASEL	0	0	0	0	--	--	--	1		

<u>Summary:</u>

Hardware implementation of β CPU

- ✓ ALU, PC
- ➢ Register File
- ➢ Control Logic
- ➢ Data paths

To be continued:

Exceptions & Completion of β data path