Computer Structure

Unit 2: Arithmetic Logic Unit (ALU)

Escuela Politécnica Superior - UAM

Outline

- **Basic Structure of a Computer (adder)**
- Arithmetic and Logic Circuits
  - ✓ Logic operators
  - ✓ Adders and subtractors
  - ✓ Shifters and multipliers
- Design of an ALU

Introduction

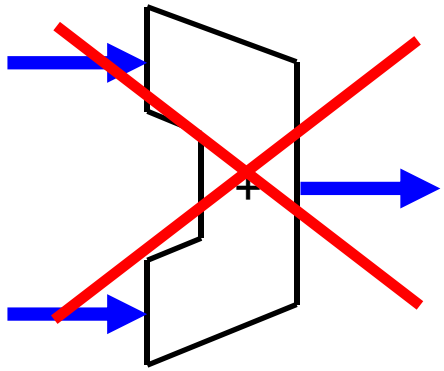
- Before designing a microprocessor, a hierarchical design can be done reusing blocks.

- Blocks that are used:
  - ✓ Multiplexers, decoders, registers and memories, arithmetic and logic circuits, etc…

- We are going to implement a simple processor which can make an addition with any number of operands.

Generic adder circuit

We are going to implement a simple processor which can make an addition with any number of operands, using operands of 32 bits (an adder with a fixed number operands cannot be used):

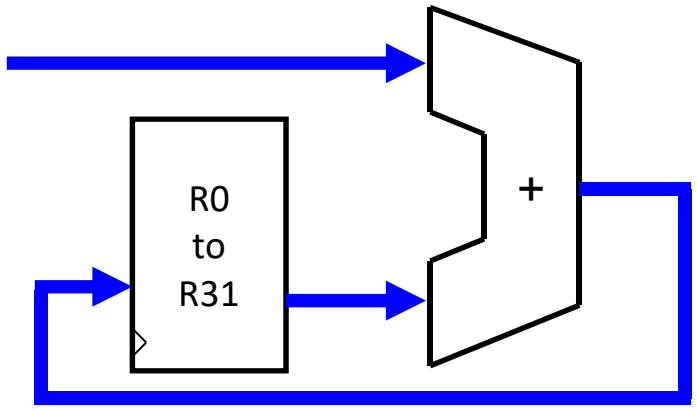


- ✓A + B;
- ✓C + D + E;
- ✓F + G + H + I;
- ✓…

A two-input adder can be used if the partial results are stored in registers => *Register File*

- ✓$R_1 = R_0 + F$;
- ✓$R_2 = R_1 + G$;
- ✓$R_3 = R_2 + H$;
- ✓$R_4 = R_3 + I$;

Equivalent to:
$R_4 = F + G + H + I$;
(Reg. R_0 is always 0)



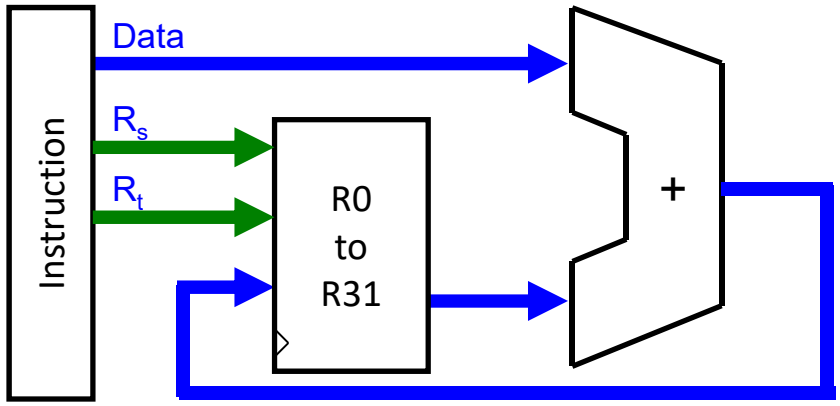
Generic adder circuit

Therefore, we need a circuit which implements the following “instruction” (it is a simplified microprocessor):

- ✓ R_t <= R_s + Data;

In every “instruction”, we need to provide the following information:

- ✓ Identifier of the destination register (R_t), from 0 to 31 => 5 bits
- ✓ Identifier of the source register (R_s), from 0 to 31 => 5 bits
- ✓ Data (immediate data) => 16 bits



Generic adder circuit

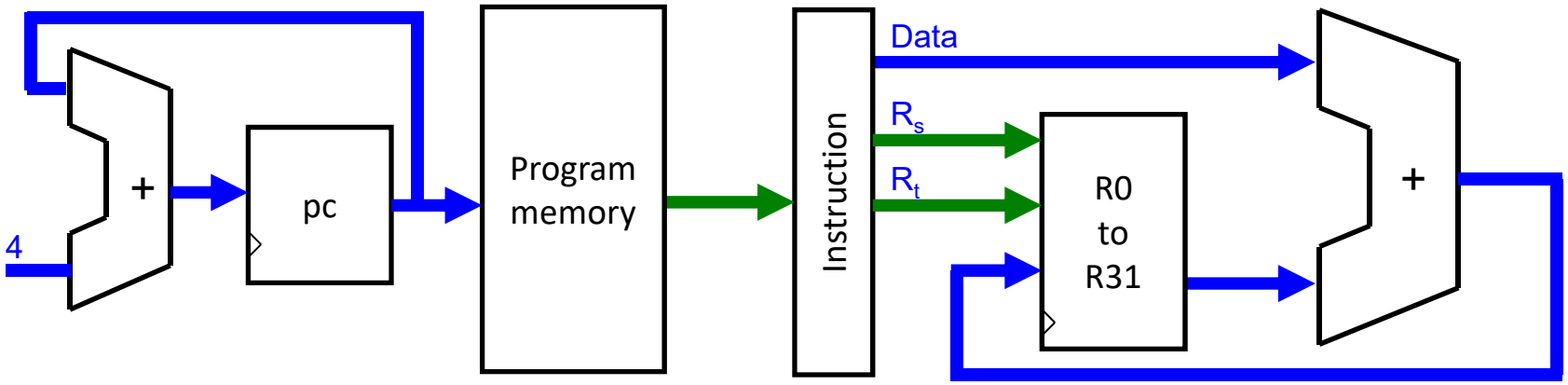
How does the microprocessor receive the instructions?

- ✓They are stored in a memory (program memory)
- ✓The microprocessor should read the memory, instruction by instruction

Every instruction need at least 5+5+16 = 26 bits

- ✓It is fitted to 32 bits (power of 2), to facilitate the design.
- ✓32 bits => 4 bytes.

Every instruction is stored in a memory address which is +4 bytes farther than the previous one.



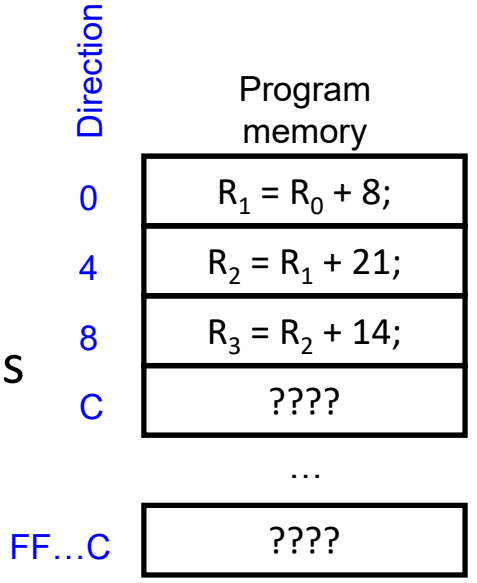
Example of a program

Addition of 8 + 21 + 14. It can be done executing the following program:

- ✓$R_1 = R_0 + 8;$
- ✓$R_2 = R_1 + 21;$
- ✓$R_3 = R_2 + 14;$

The *program counter* (pc) defines the memory address of the current instruction.

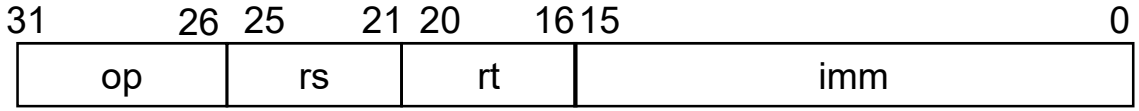
- ✓It starts at 0 and it is incremented by 4 in every instruction



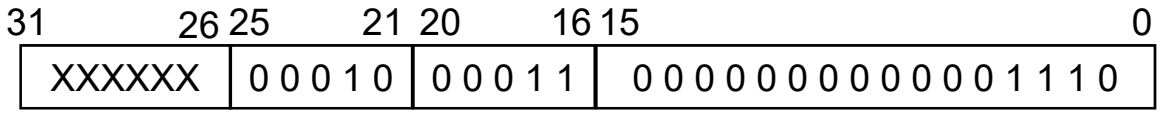
Program memory (instructions)

A 32-bit instruction is stored in every program memory address:

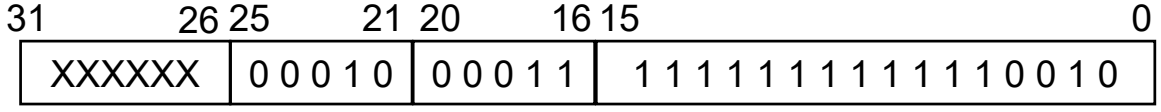
- ✓ 5 bits for the destination register, R_t
- ✓ 5 bits for the source register, R_s
- ✓ 16 bits for the immediate data
- ✓ The remaining bits are not used. In real microprocessors they are used to define the *operation code* (op), because they should execute more instructions apart from additions.



Example: R3 = R2 + 14;



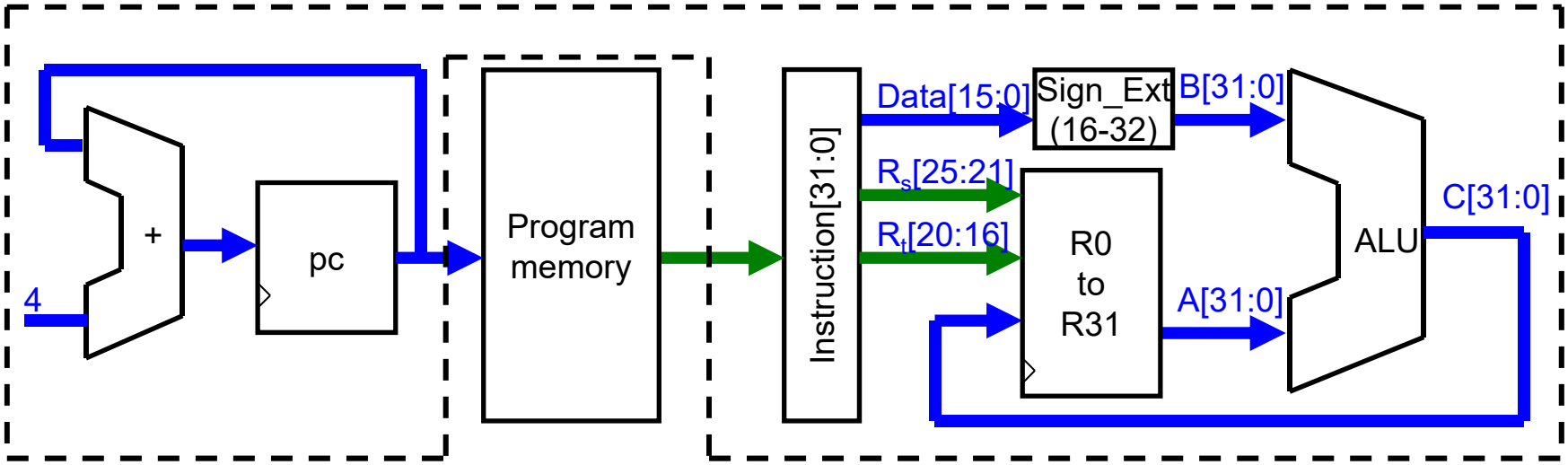
Example: R3 = R2 - 14;



Word width

The microprocessor uses 32-bit data:

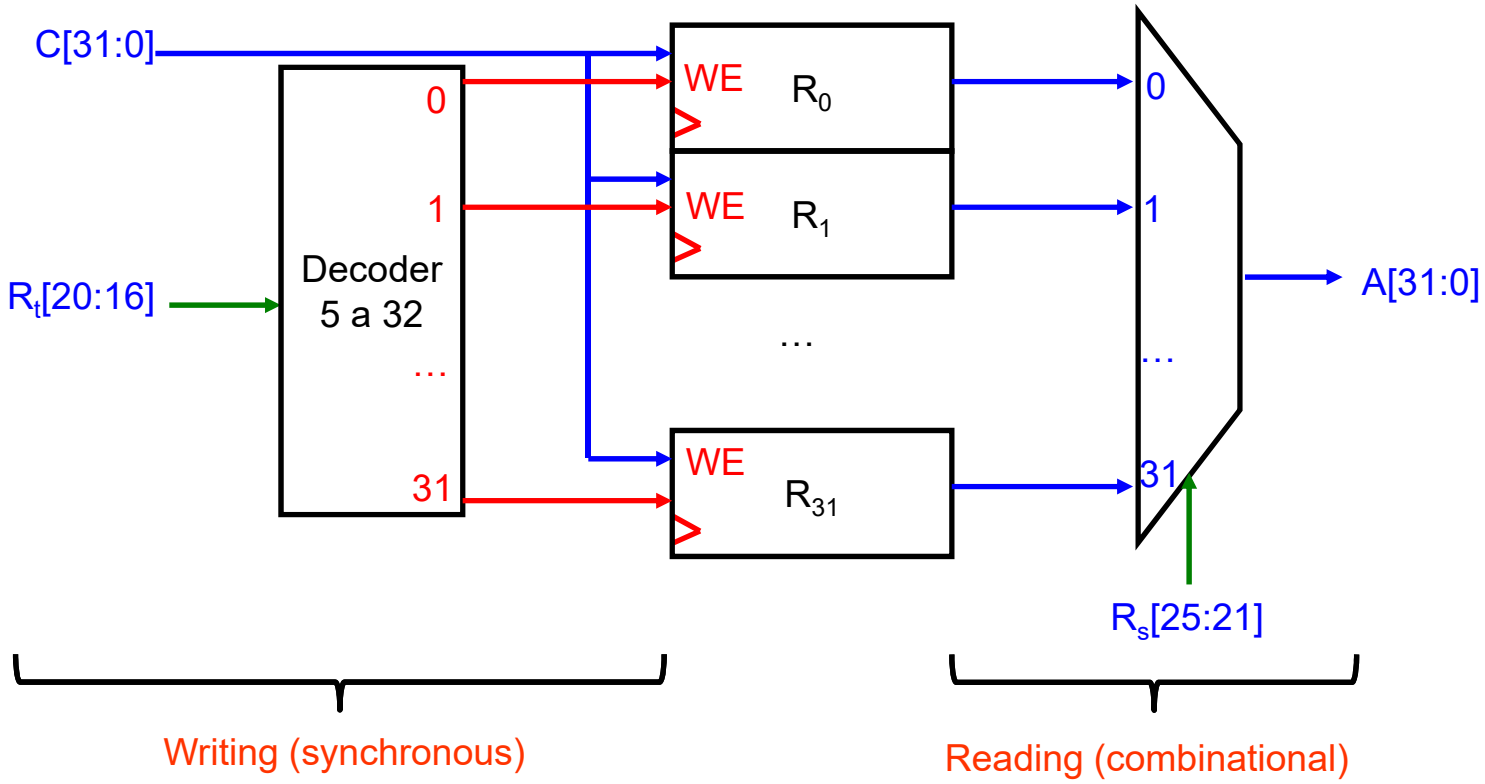
- ✓The inputs and output of the adder (ALU) have 32 bits
- ✓The registers of the register file have 32 bits
- ✓As the immediate data has 16 bits, it is sign-extended to 32 bits



Register File (GPR, General Purpose Registers)

How is the Register File implemented?

- ✓Multiplexing 32 registers (every register has 32 bits)

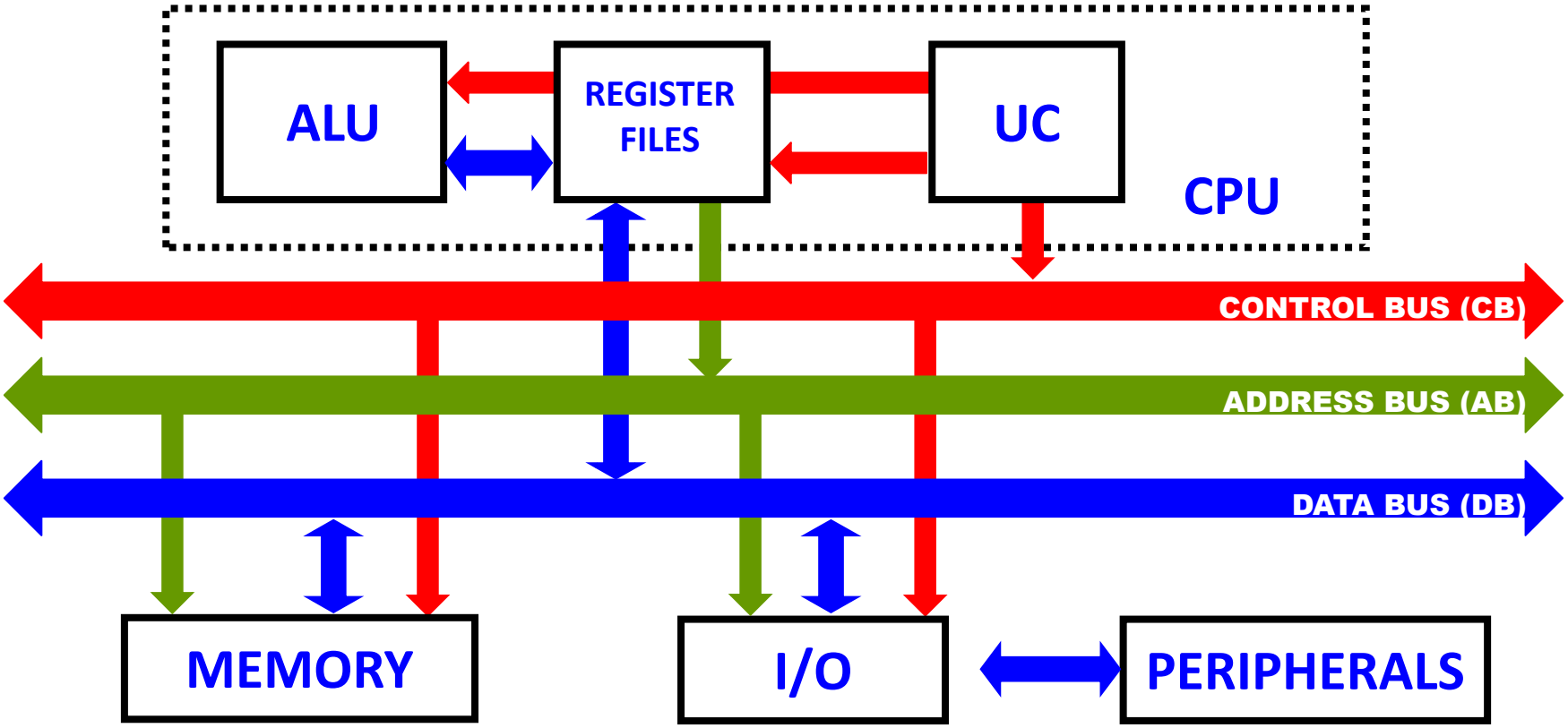


Real microprocessor

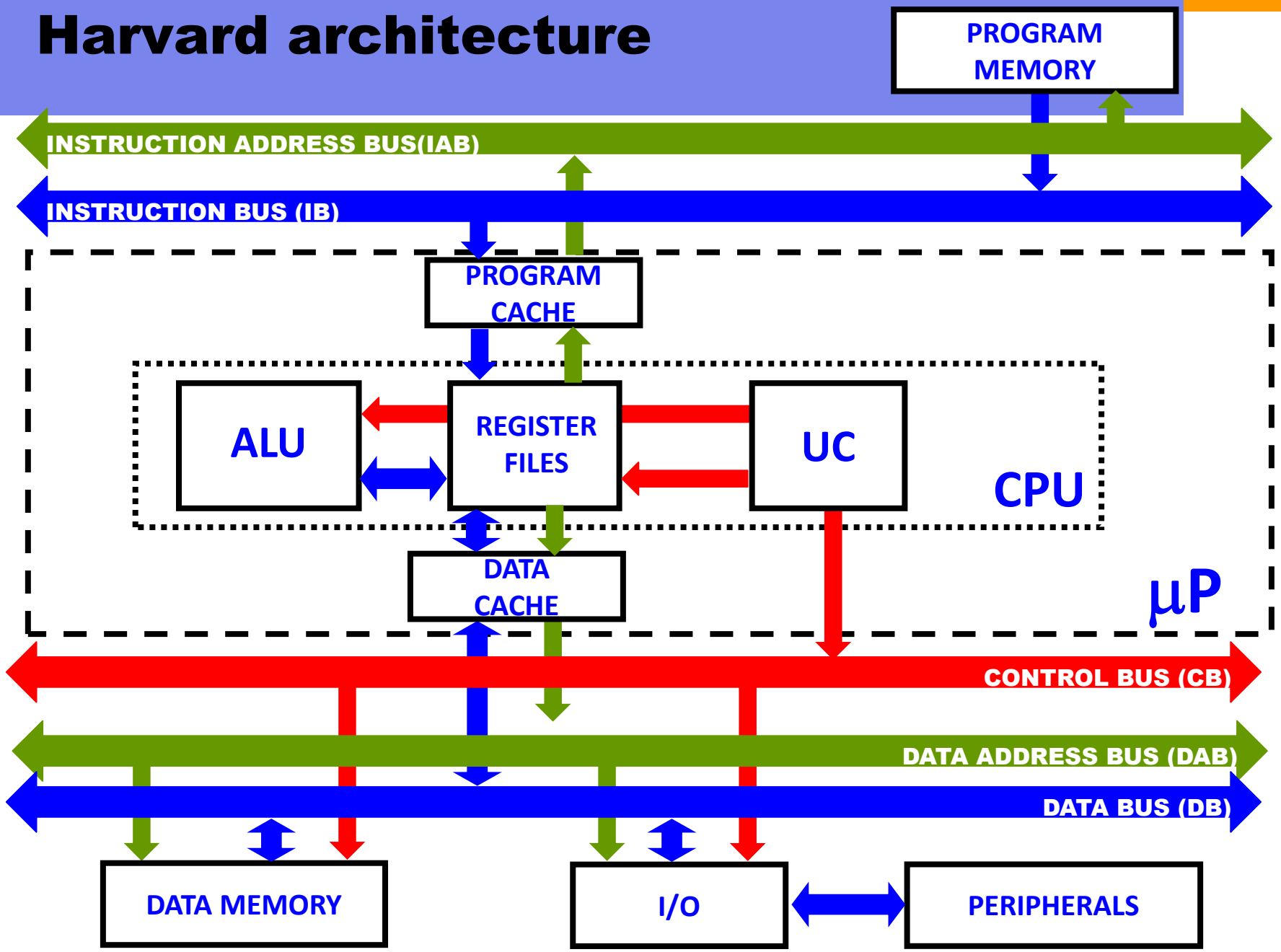
What should a real microprocessor have?

- ✓Implement other instructions (arithmetic and logic operations)
- ✓Use more than 32 values, so it should use the data memory (with instructions to access the data memory)
- ✓Be able to change the execution flow so loops and if-then sentences can be done, like *for*, *if*, etc… (conditional and unconditional jumps)

Von Neumann classic architecture



Harvard architecture

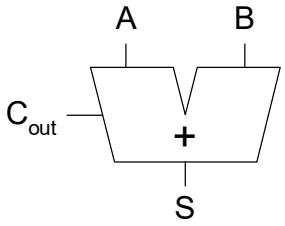


Outline

- Basic Structure of a Computer (adder)
- **Arithmetic and Logic Circuits**
  - ✓ Logic operators
  - ✓ Adders and subtractors
  - ✓ Shifters and multipliers
- Design of an ALU

1-bit Adder

Half Adder

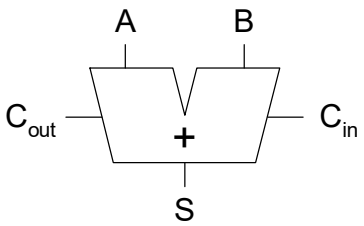


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

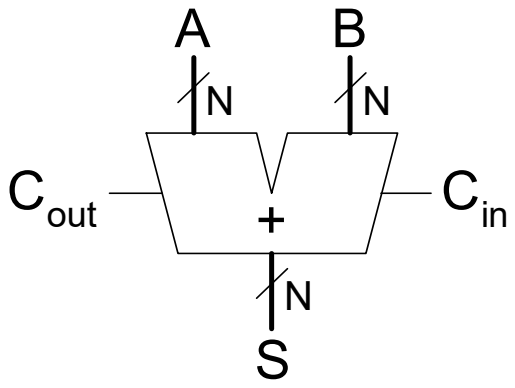
$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + BC_{in} + AC_{in}$

Multibit Adders

- Multibit C*arry Propagate Adders (CPA*):
  - Ripple-Carry Adders, RCA (slow)
  - Carry-Lookahead Adders, CLA (fast)
  - Parallel Prefix Adders, PPA (faster)
- CLA and PPA are faster but they require more hardware resources.

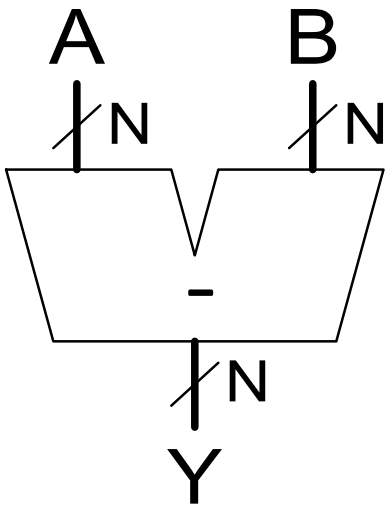
Symbol



Other operators

Subtractor: Y = A - B

Symbol



Remember

(*Computer Basics*)

$Y = A-B = A +(-B)$

How to calculate the additive inverse of a two's-complement number? using a **NOT** gate and then **adding 1**.

eg: $Y=8_{10}-5_{10}=3_{10}= 01000_2 - 00101_2=?$

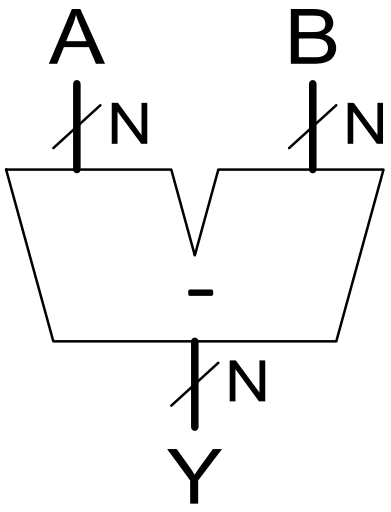
$Inverse(00101_2)= NOT(00101)+1 =$
$11010+1 = 11011$

$Y=A+(-B) = 01000_2+11011_2 = 00011_2$

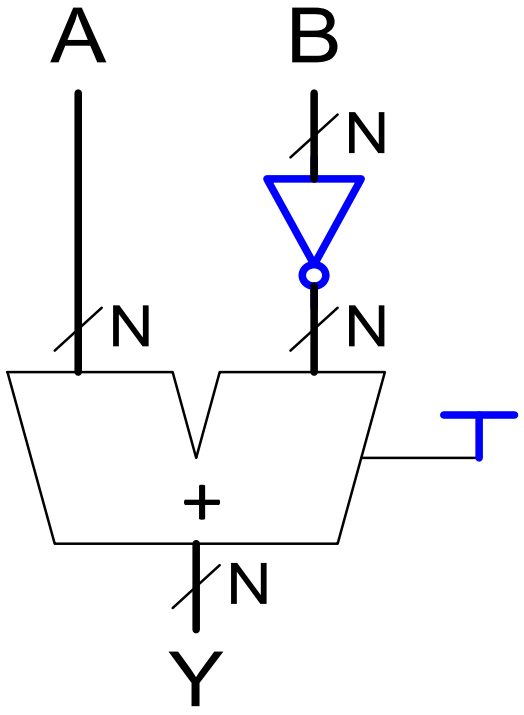
Other operators

Subtractor: Y = A - B

Symbol



Implementation

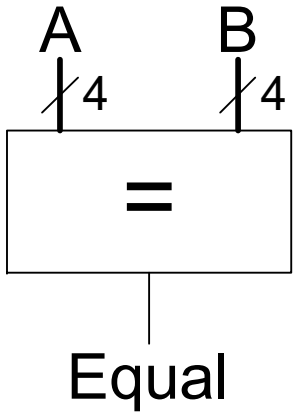


2.1, 2.2, 2.3

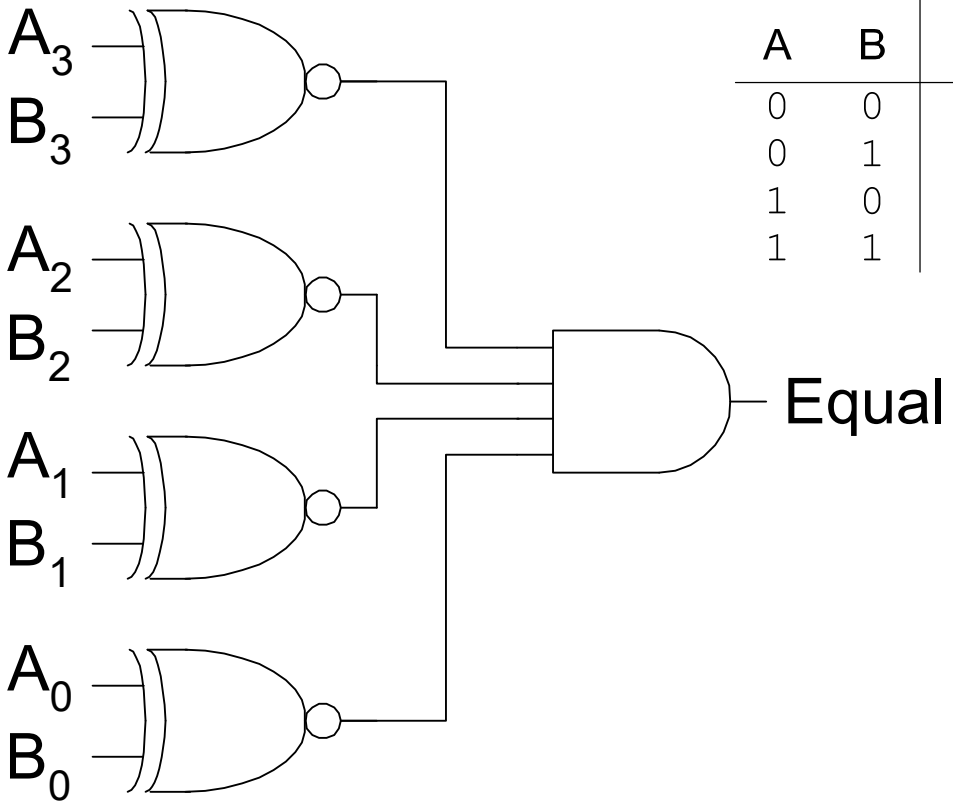
Other operators

Equal operator: A = B ?

Symbol



Implementation



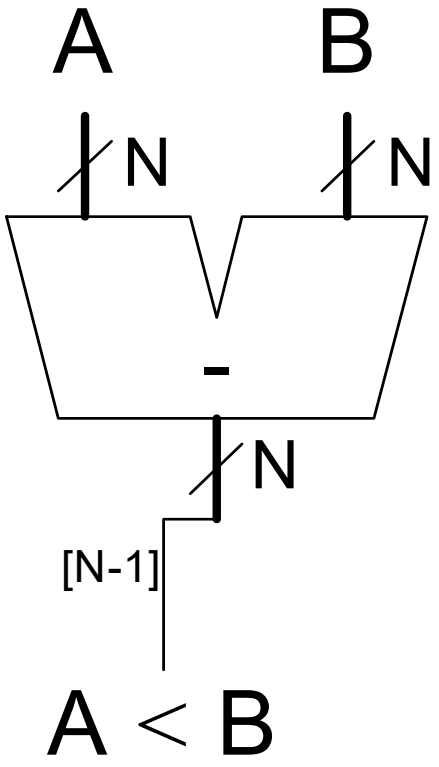
XNOR

A	B	S
0	0	1
0	1	0
1	0	0
1	1	1

Two numbers are equal when all their bits are equal

Other operators

***Less Than* operator: A < B ?**



To check if a number is greater/lower than another one, subtract them and check the sign of the result (MSB)

Shifters

- **Logical shifter:** It shifts the value to the left or right and fill with 0's.
  - ✓ Ex: 11001 >> 2 = 00110
  - ✓ Ex: 11001 << 2 = 00100
- **Arithmetic shifter:** It is similar to the Logic shifter but it fills with the sign bit (MSB) when there is a right shifting.
  - ✓ Ex: 11001 >>> 2 = **11**110
  - ✓ Ex: 11001 <<< 2 = 00100
- **Rotator:** It rotates to the left or right the bits like in a circle, so the bits that are missed from one side appear in the other side.
  - ✓ Ex: 11001 ROR 2 = 01110
  - ✓ Ex: 11001 ROL 2 = 00111

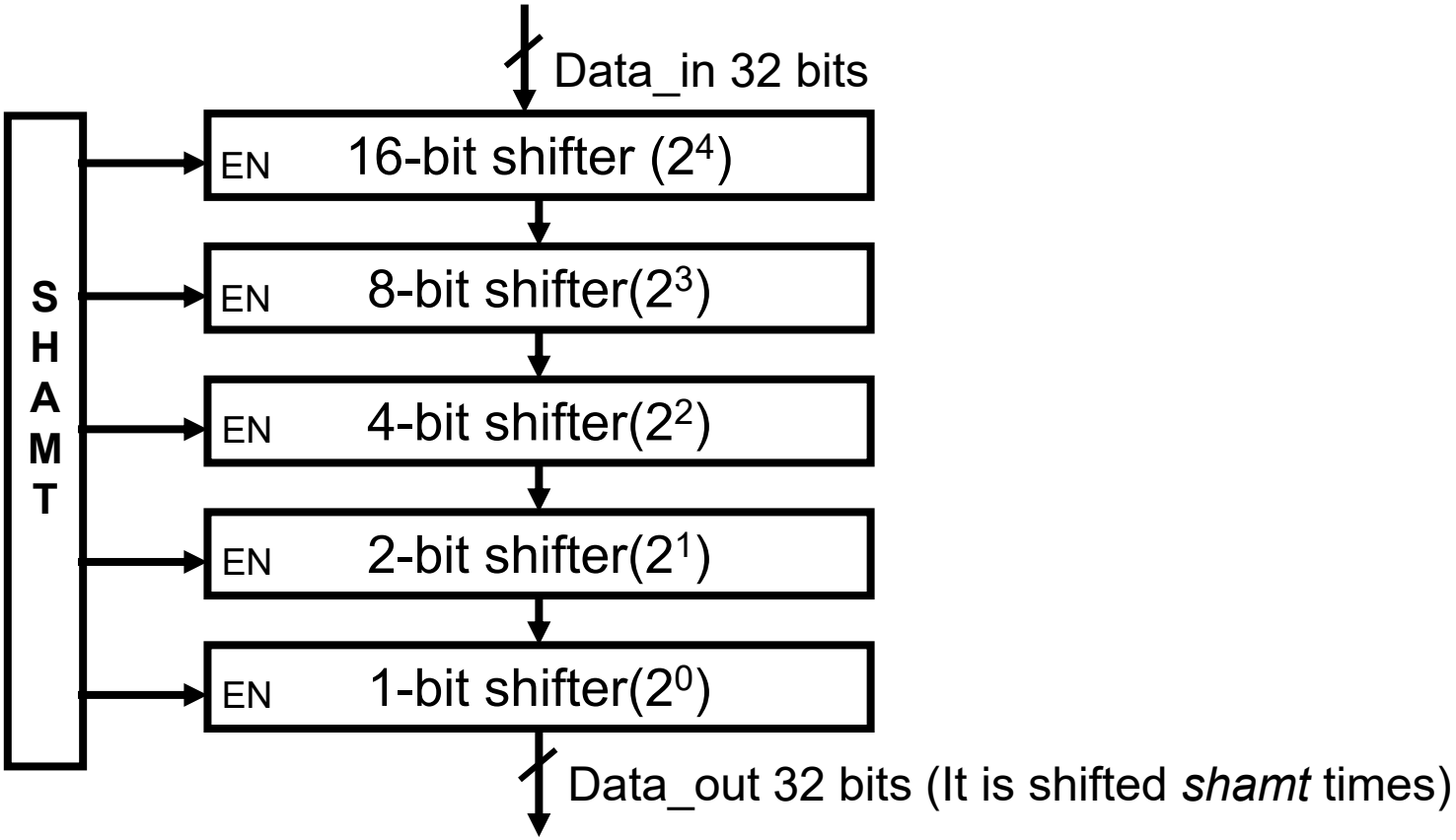
Using shifters as multipliers and dividers

- A left shifting of *N* bits is equivalent to multiply by 2^N
  - ✓Eg: 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - ✓Eg: 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)

- An arithmetic right shifting is of *N* bits is equivalent to divide by 2^N
  - ✓Eg: 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - ✓Eg: 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)

Barrel Shifter

- How to shift an undefined number of bits, from 0 to 31?
  - ✓ In MIPS, that amount of shifting is stored in $shamt_{4:0}$
- Using fixed shifters 2^N in series
  - ✓ Every shifter is turned on/off depending on a bit of $shamt_{4:0}$



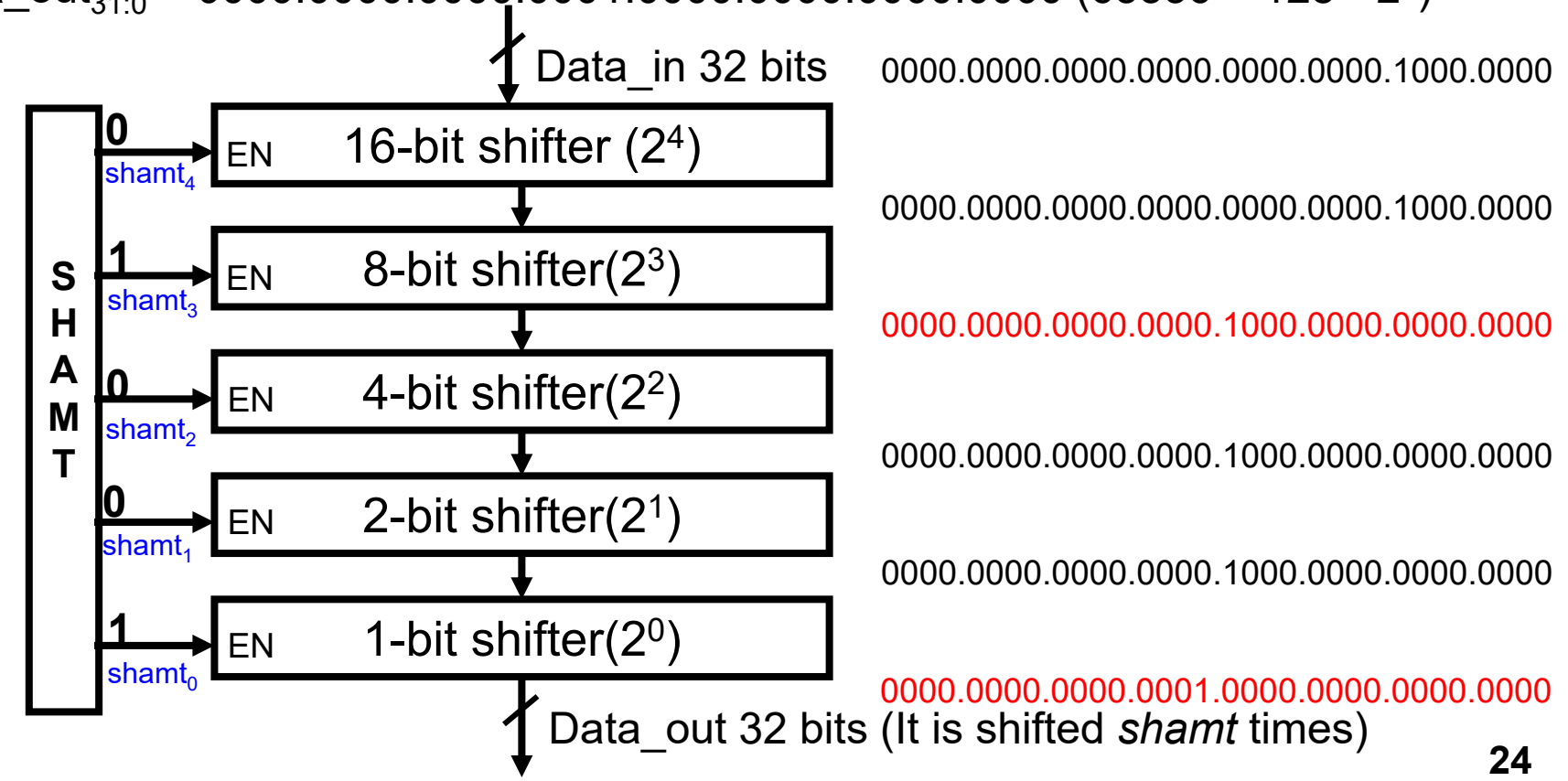
Barrel Shifter

Example:

$shamt_{4:0}$ = 01001 (9)

$Data\_in_{31:0}$ = 0000.0000.0000.0000.0000.0000.1000.0000 (128)

$Data\_out_{31:0}$ = 0000.0000.0000.0001.0000.0000.0000.0000 (65536 = 128 * 2^9)



Multipliers

- Steps to multiply (in decimal or binary):
  - The partial products are calculated multiplying a digit of the multiplier by the whole multiplicand
  - The partial products are shifted and they are added to get the final result

Decimal

	230	multiplicand
x	42	multiplier
	460	partial products
+	920	
	9660	result

230 x 42 = 9660

Multiplication, unsigned

Unsigned

```
      1 1 0 0 0 1      (49)
    x   1 1 0 1 0      (26)
-------------------
      0 0 0 0 0 0
    1 1 0 0 0 1
  0 0 0 0 0 0
1 1 0 0 0 1
1 1 0 0 0 1
-------------------
1 0 0 1 1 1 1 1 0 1 0  (1274)
```

Unsigned

```
      0 0 1 0 0 1      ( 9)
    x     0 1 1 0      (6)
-------------------
      0 0 0 0 0 0
    0 0 1 0 0 1
  0 0 1 0 0 1
0 0 0 0 0 0
-------------------
0 0 0 1 1 0 1 1 0      (54)
```

Multiplication, signed

- **Using unsigned arithmetic:**
  - Calculate the sign of the result:
    - ✓ Pos*Pos = Pos, Pos*Neg = Neg , Neg*Pos = Neg, Neg*Neg = Pos.
  - If any operand is negative:
    - ✓ Do the two's complement to make it positive.
  - Multiply like with unsigned operands
  - If the result should be negative:
    - ✓ Do the two's complement to the previous result.

Multiplication, signed

$0010_2 \times 1100_2 = 2_{10} \times -4_{10}$

P N

Result sign: Negative (PxN)
C2 of $(1100_2) = 0100_2$

```
P →   0 0 1 0        (2)
P → x 0 1 0 0        (4)
      -------
      0 0 0 0
    0 0 0 0
  0 0 1 0
0 0 0 0
-------------
0 0 0 1 0 0 0        (8)
-------------
1 1 1 1 0 0 0        (-8)
```

Multiplication, signed

- **Using signed arithmetic:**
  - Do the partial multiplications like with unsigned operands and sign-extend the results.
  - If the multiplier MSB is -1, the multiplier is negative, so the last partial result will not be the multiplicand but the two's complement of it.
  - The final product is the addition of all the partial products.

Multiplication, signed

$0010_2 \times 1100_2 = 2_{10} \times -4_{10}$

	0 0 1 0	(2)
	x 1 1 0 0	(-4)
	0 0 0 0 0 0 0 0	(2*(0*2^0)
	0 0 0 0 0 0 0	(2*(0*2^1)
	0 0 0 0 1 0	(2*(1*2^2)
C2(0010) →	1 1 1 1 0	(2*(-1*2^3)
	1 1 1 1 1 0 0 0	(-8)

$1011_2 \times 0010_2 = -5_{10} \times 2_{10}$

	1 0 1 1	(-5)
	x 0 0 1 0	(2)
	0 0 0 0 0 0 0 0	(-5*(0*2^0)
	1 1 1 1 0 1 1	(-5*(1*2^1)
	0 0 0 0 0 0	(-5*(0*2^2)
	0 0 0 0 0	(-5*(0*2^3)
	1 1 1 1 0 1 1 0	(-10)

Multiplication, signed

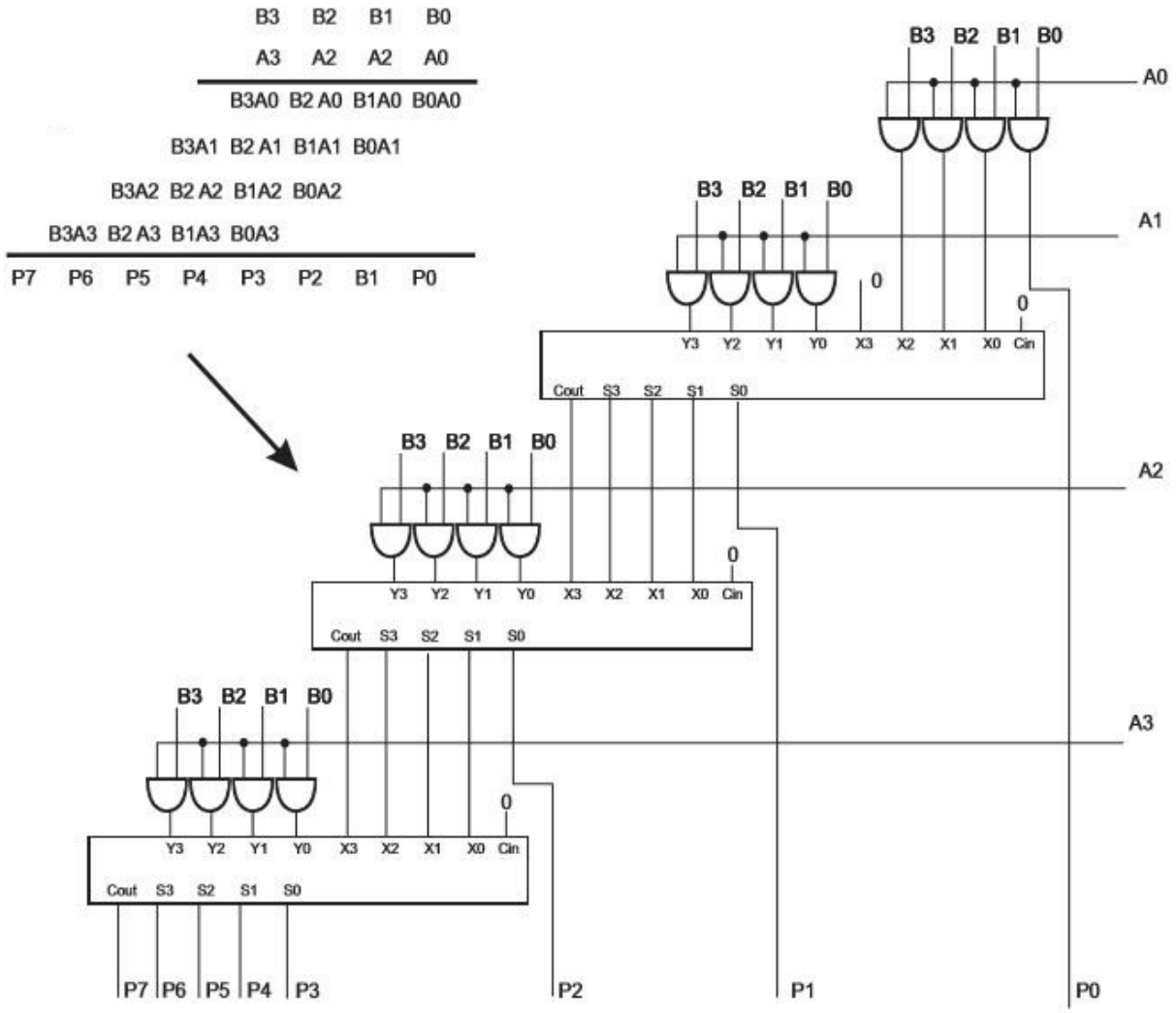
Unsigned

```
      1 1 0 0 0 1      (49)
    x   1 1 0 1 0      (26)
-------------------
      0 0 0 0 0 0
    1 1 0 0 0 1
  0 0 0 0 0 0
1 1 0 0 0 1
1 1 0 0 0 1
-------------------
1 0 0 1 1 1 1 1 0 1 0  (1274)
```

Signed

```
      1 1 0 0 0 1      (- 15)
    x   1 1 0 1 0      (- 6)
-------------------
0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 1
0 0 0 1 1 1 1
-------------------
0 0 0 0 1 0 1 1 0 1 0  (+ 90)
```

4 x 4 multiplier (unsigned)



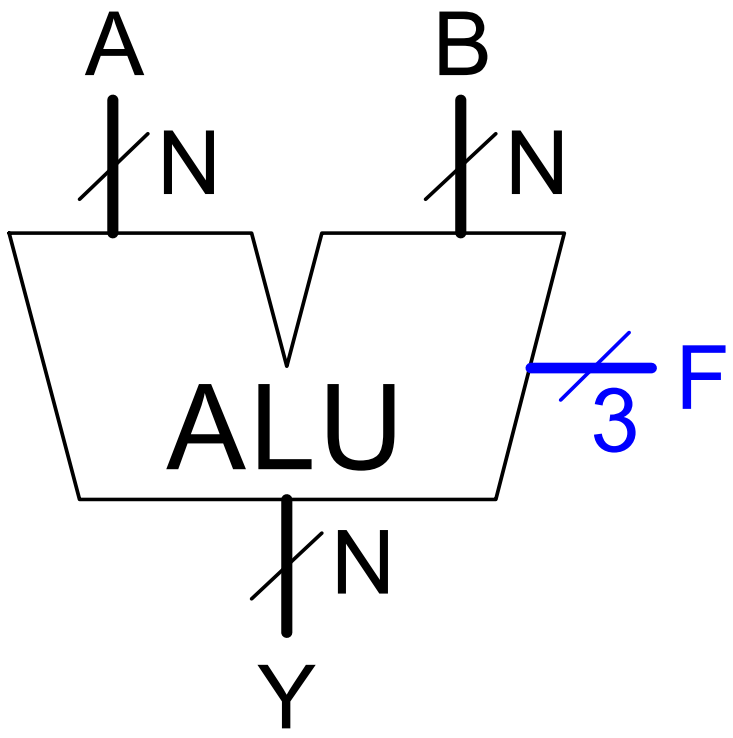
2.4

Outline

- Basic Structure of a Computer (adder)
- Arithmetic and Logic Circuits
  - ✓ Logic operators
  - ✓ Adders and subtractors
  - ✓ Shifters and multipliers
- **Design of an ALU**

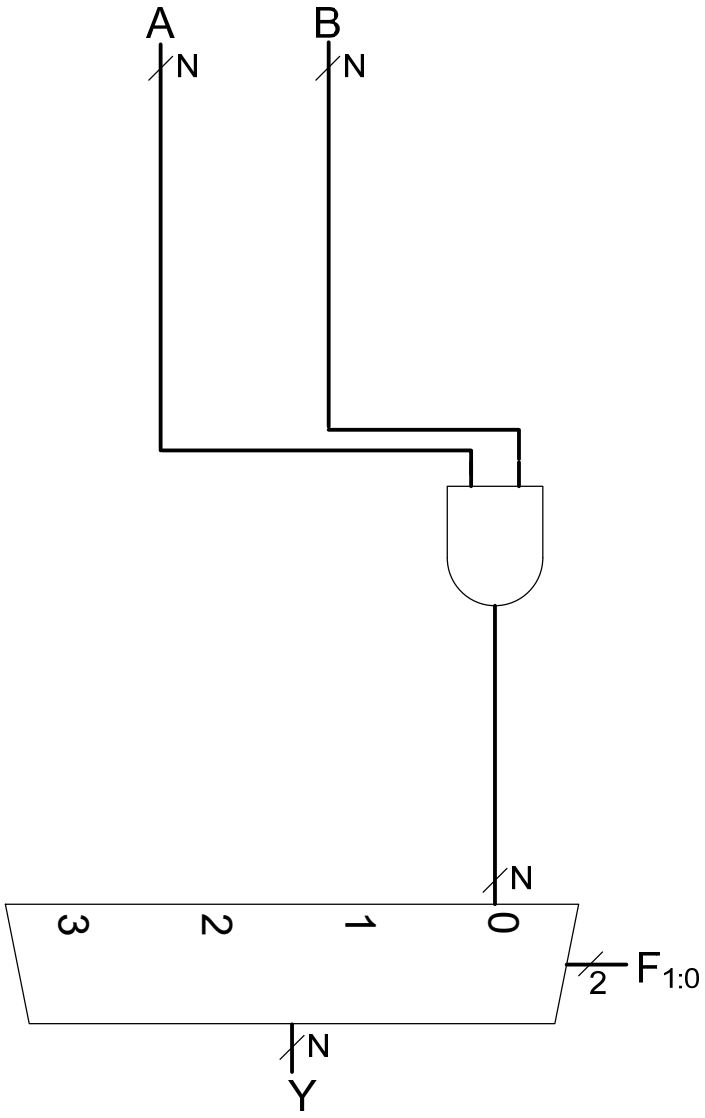
Arithmetic Logic Unit (ALU)

$F_{2:0}$ = Function selector



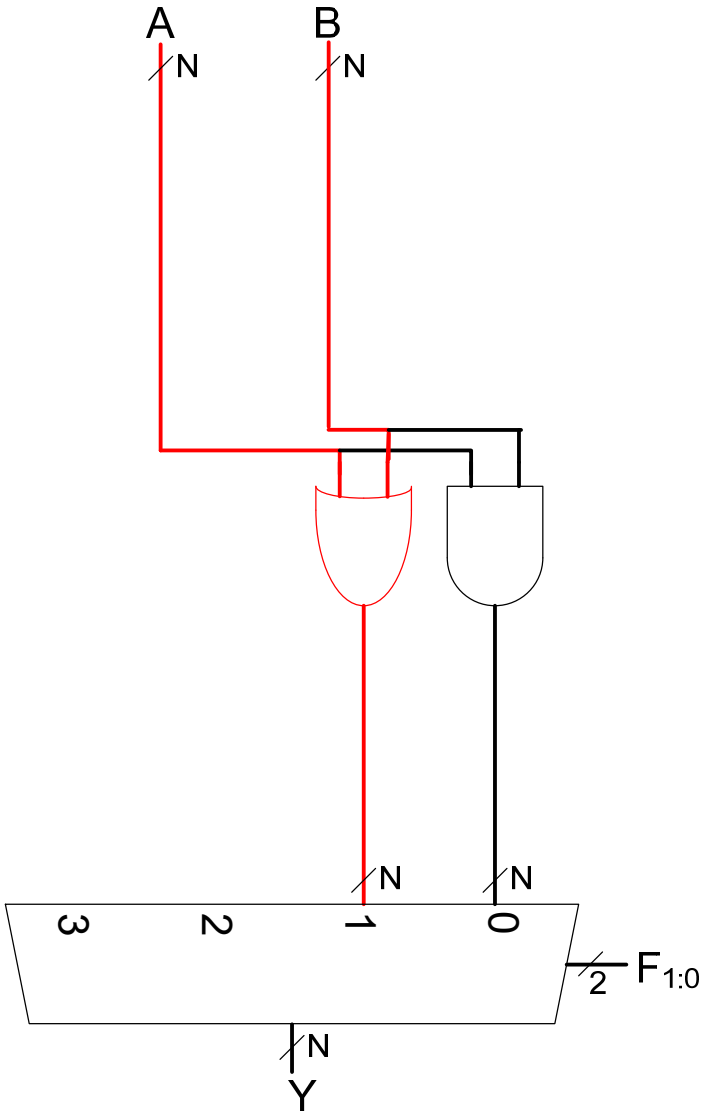
$F_{2:0}$	Function
000	A and B
001	A or B
010	A + B
011	Unused
100	A and /B
101	A or /B
110	A – B
111	SLT

ALU design



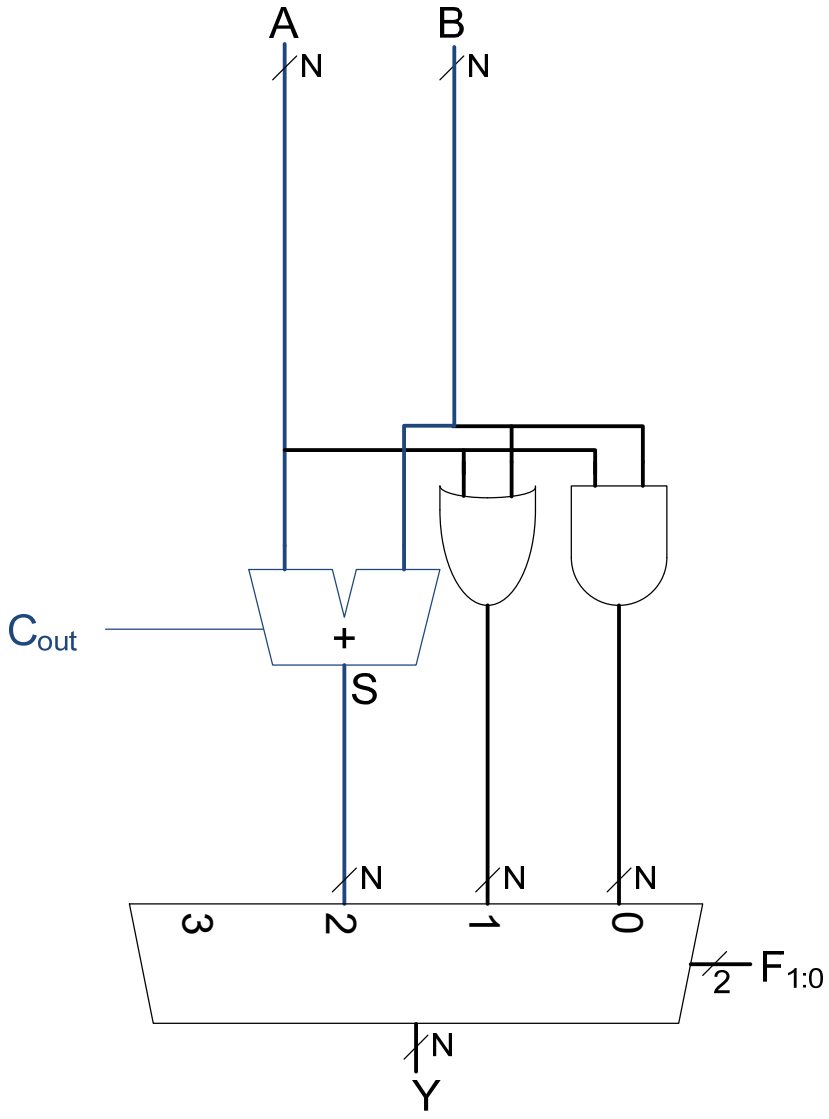
$F_{2:0}$	Function
000	A and B
001	A or B
010	A + B
011	Unused
100	A and /B
101	A or /B
110	A – B
111	SLT

ALU design



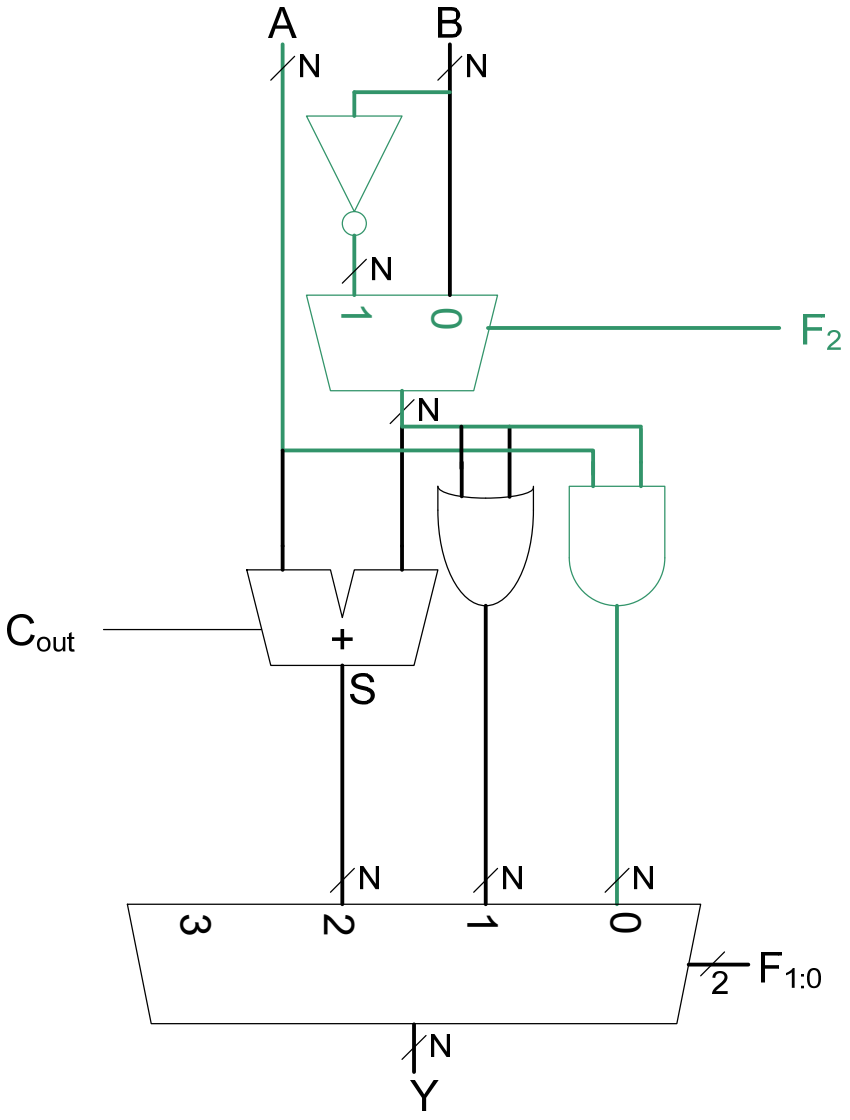
$F_{2:0}$	Function
000	A and B
001	A or B
010	A + B
011	Unused
100	A and /B
101	A or /B
110	A – B
111	SLT

ALU design



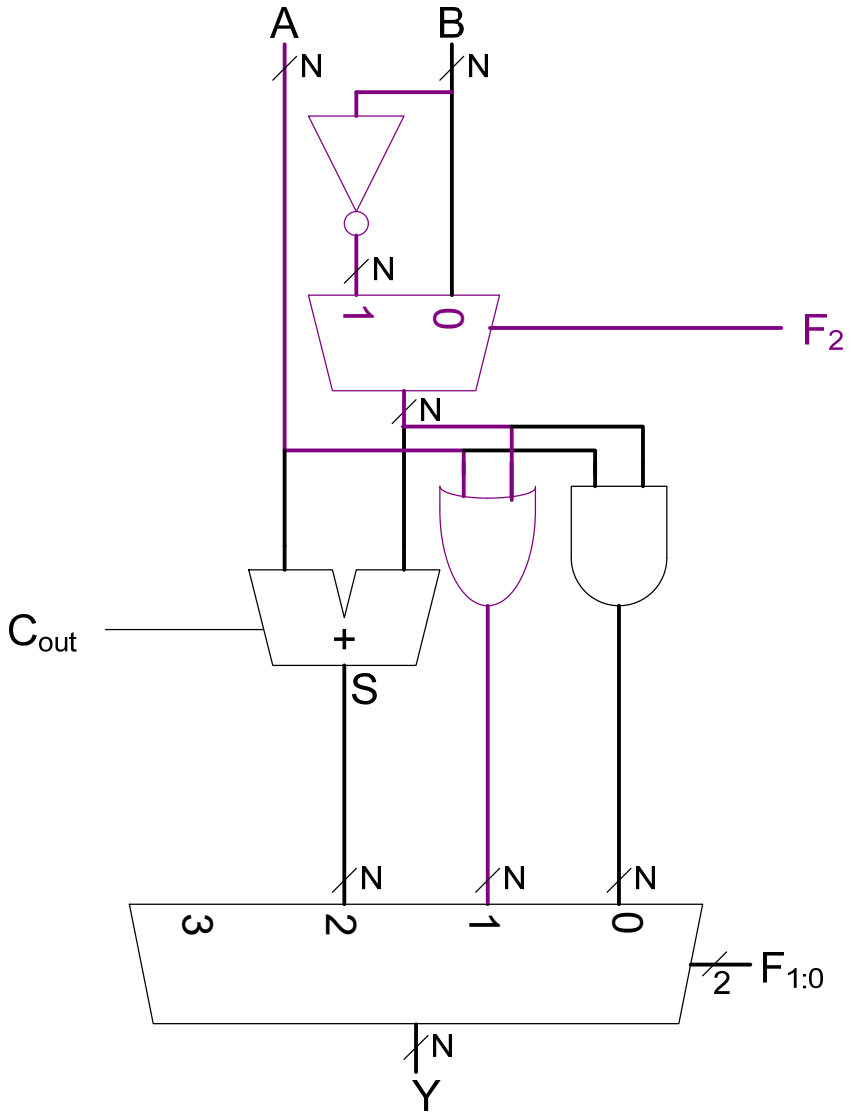
$F_{2:0}$	Function
000	A and B
001	A or B
010	A + B
011	Unused
100	A and /B
101	A or /B
110	A – B
111	SLT

ALU design



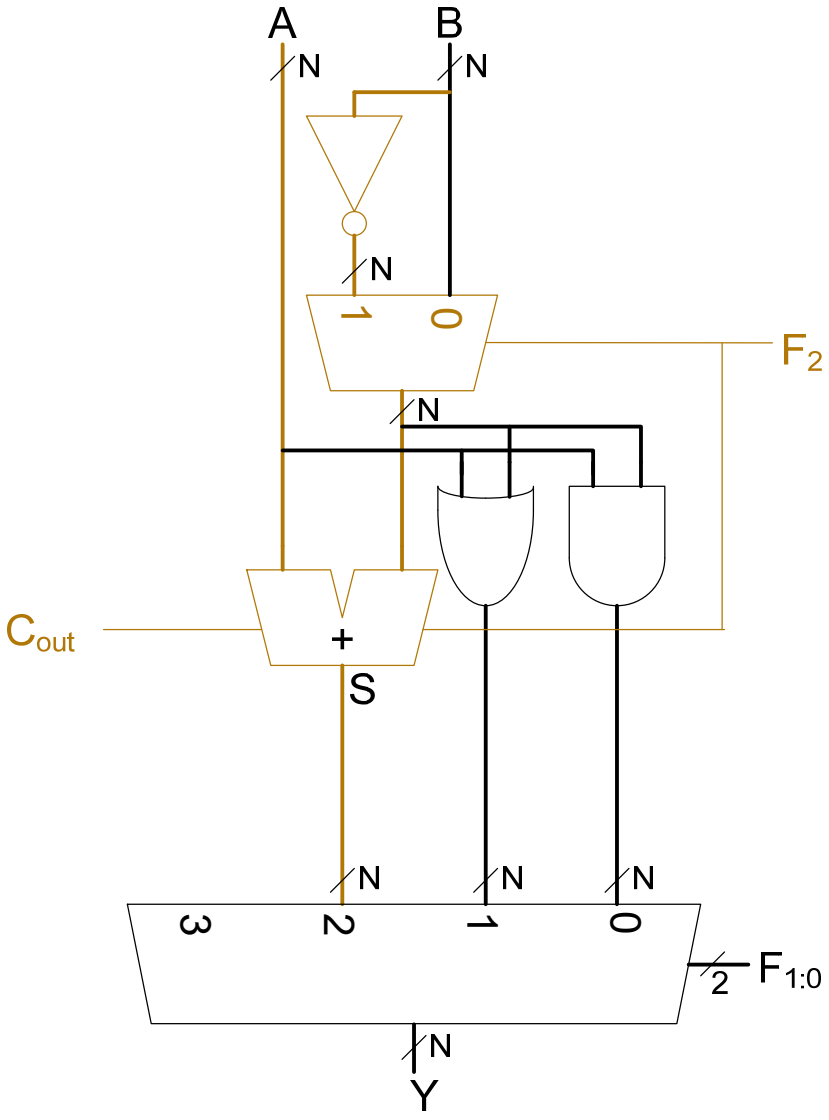
$F_{2:0}$	Function
000	A and B
001	A or B
010	A + B
011	Unused
100	A and /B
101	A or /B
110	A – B
111	SLT

ALU design



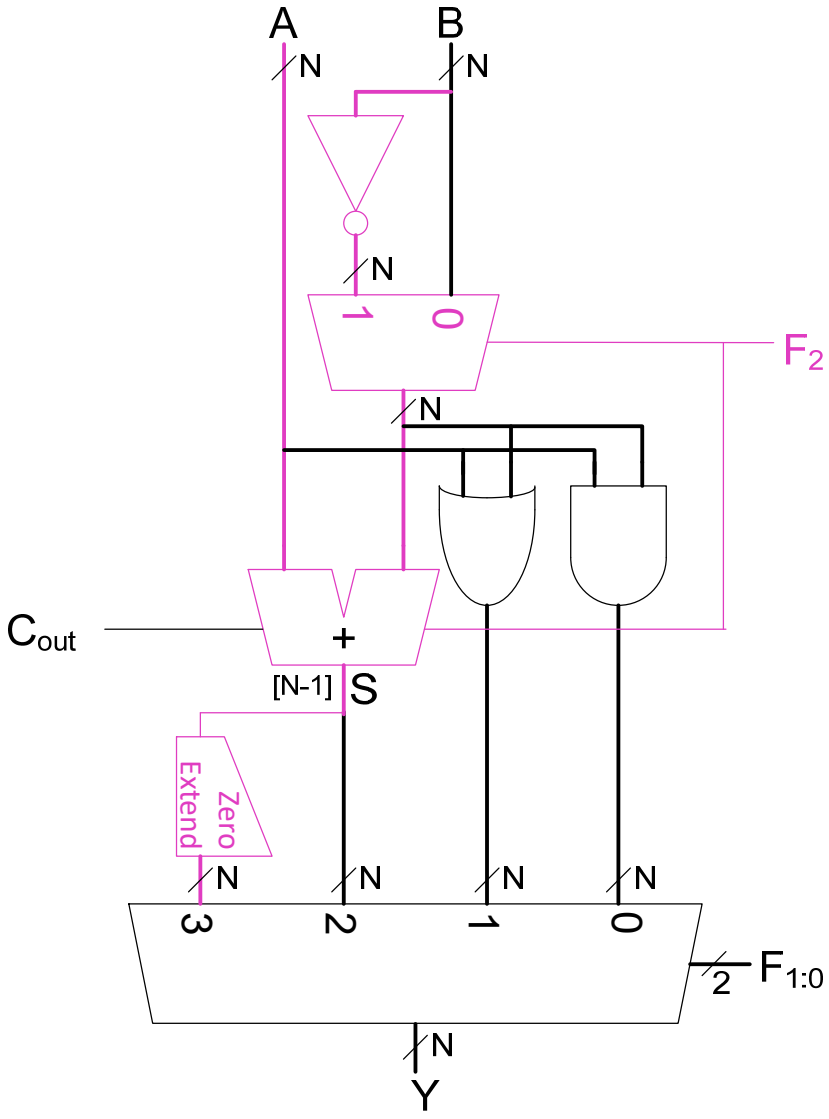
$F_{2:0}$	Function
000	A and B
001	A or B
010	A + B
011	Unused
100	A and /B
101	A or /B
110	A – B
111	SLT

ALU design



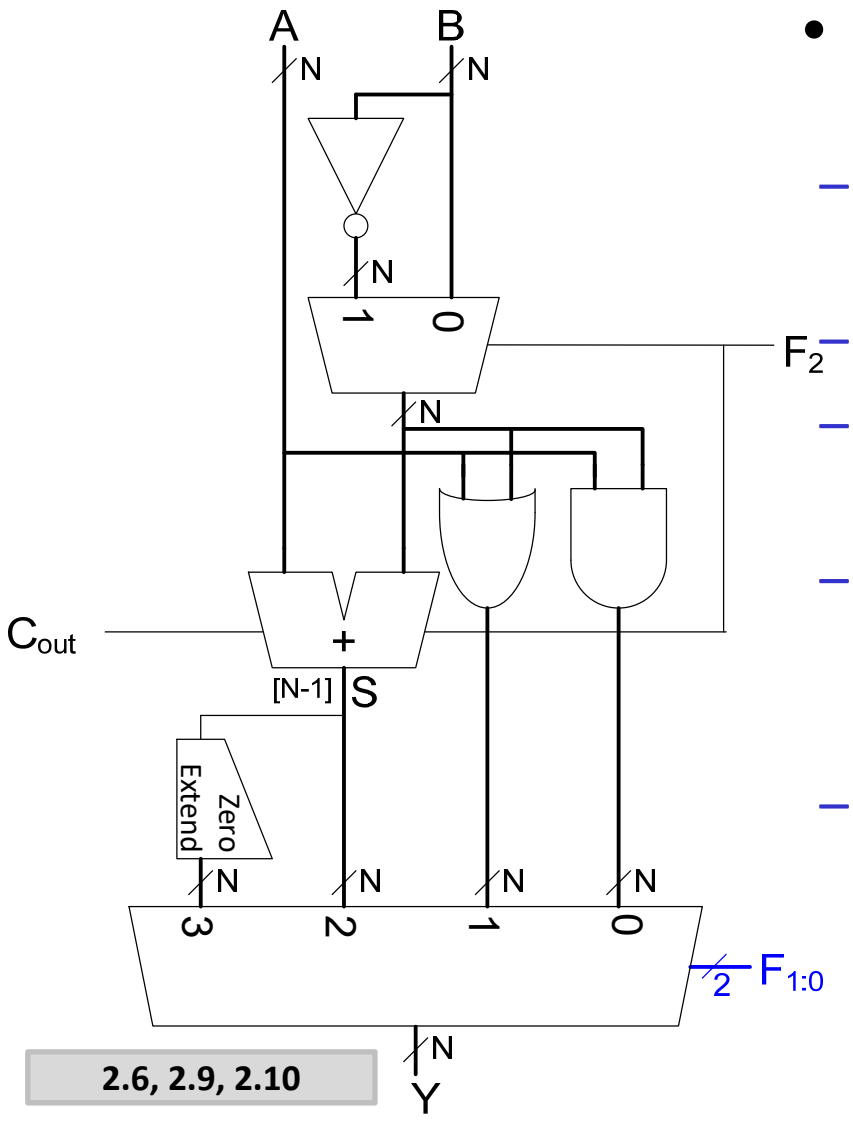
$F_{2:0}$	Function
000	A and B
001	A or B
010	A + B
011	Unused
100	A and /B
101	A or /B
110	A – B
111	SLT

ALU design



$F_{2:0}$	Function
000	A and B
001	A or B
010	A + B
011	Unused
100	A and /B
101	A or /B
110	A – B
111	SLT

Example of *Set Less Than* (SLT)



2.6, 2.9, 2.10

- The result should be 1 if A<B and 0 otherwise. E.g.: *A* = 25 y *B* = 32.
  - A is lower than B, so Y should be 1 written with 32 bits (0x00000001).
  - To choose SLT, $F_{2:0}$ = 111.
  - F_2 = 1 lets the adder to do a subtraction. Hence, 25 - 32 = -7.
  - The C2 representation of -7 has 1 in the *most significant bit* (MSB), so S_{31} = 1.
  - $F_{1:0}$ = 11, so the multiplexer chooses $Y = S_{31}$ (zero extended) = 0x00000001.

Unidad 2: Arithmetic Logic Unit (ALU)

Escuela Politécnica Superior - UAM

Unit 2 exercises

2.1.- Do a truth table to show that the logic expression Cn□Cn-1 generates the flag named overflow, V, that is active when there is an error when adding two signed integer numbers into other integer number with the same number of bits (the result does not fit into the same size).

2.2. Apart from the main result, an ALU generates a set of bits (flags) to control the arithmetic or logic operations executed. The most used flags are the sign flag (N, N = '1' for negative results), the zero flag (Z, Z = '1' if the result is zero), the carry flag (C, C = '1' when there is carry when adding two positive numbers), and the overflow flag (V, V = '1' when the result exceeds the representation capabilities). Using 8 bits signed operands (two's complement representation), do the following operations (decimal notation in the statement) including both the result and the four flags, N, Z, C and V, indicating the meaning of the flags.

a) 46 + 67 **b)** 112 - 89 **c)** 75 + 95 **d)** -34 – 97

2.3.- Using binary 8 bits signed numbers in two's complement, do the operations in the order stated by the parentheses. For each partial result, check the overflow, and then also check the final result after all the operations. Analyse the results taking into account the overflow flag. Please, be aware that incorrect partial results do not always mean wrong final results.

a) (((((32 +100) + 70) + 24) – 62) – 50). **b)** ((((43 - 12) + 34) + 75) – 47). **c)** ((((15 – 77) – 43) – 38) + 32)

T

Unit 2 exercises

2.4. Multiply the following numbers in two's complement:

a) 010111_2 x 110110_2 **b)** 110011_2 x 101100_2 **c)** 0011111_2 x 0011111_2

d) 56_{10} x $(-67)_{10}$ **e)** -43_{10} x $(-113)_{10}$ **f)** $A4_{16}$ x $B8_{16}$.

Note: Use 8 bits for each operand (sign extend if needed), and the necessary number of bits in the result for avoiding possible overflows.

T

Unit 2 exercises

2.6.- Please, consider the ALU and registers of the figure. Answer the questions writing the necessary control word(s). Each control word must be in the format $C_4C_3C_2C_1C_0$. For instance, the operation "A+B->A" would have the control word 10001.

a) Two methods for putting 0 into A.

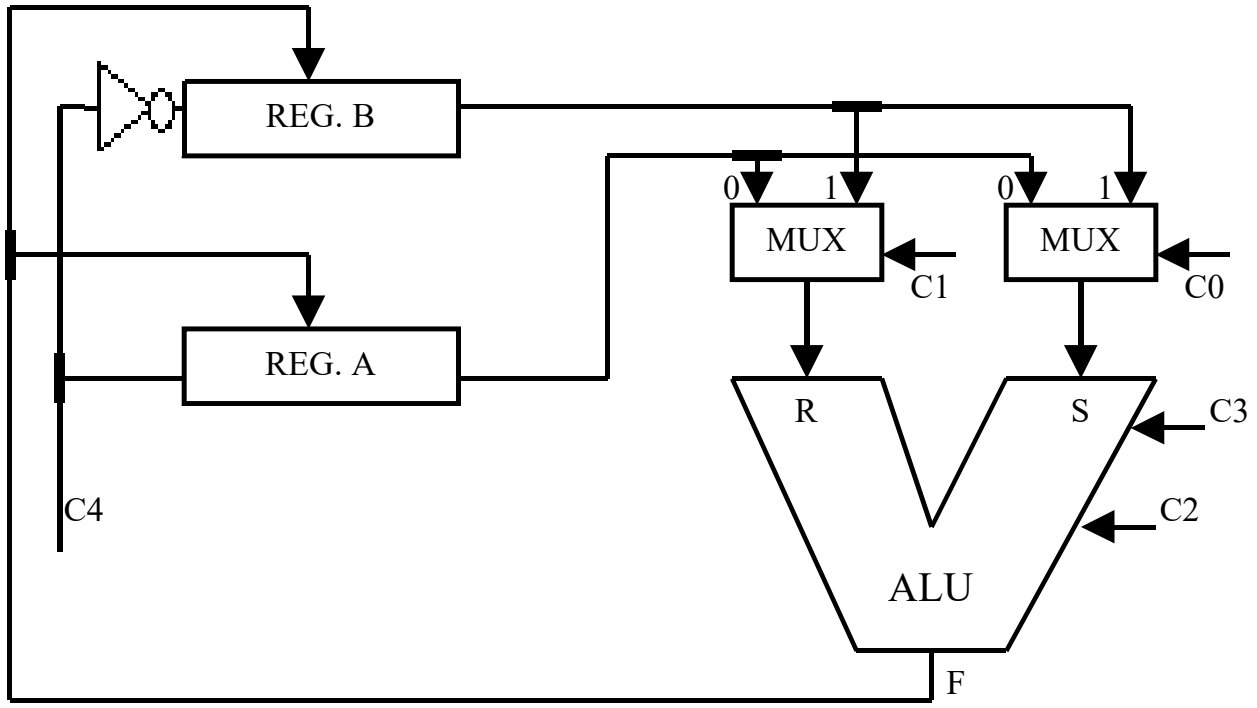
b) A sequence of operations for swapping the contents of registers A and B.

The behaviour depending on the control bits is the following.

C_1C_0	→ R	→ S
00	A	A
01	A	B
10	B	A
11	B	B

C_3C_2	F
00	R + S
01	R – S
10	R **AND** S
11	R **XOR** S

C_4	ACCION
0	F → B
1	F → A



Unit 2 exercises

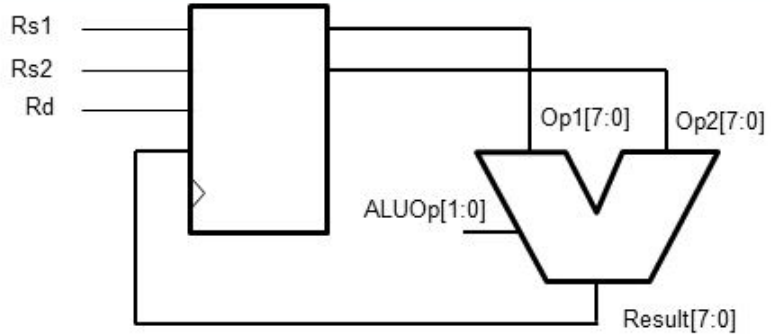
2.9. The figure shows an architecture with two registers (R0 and R1) of 8 bits and an 8-bits ALU which can do four operations (see table). The control signal ALUOp, of 2 bits, chooses the operation. The control signals Rs1, Rs2 and Rd choose the 2 registers used as ALU inputs (Rs1 for Op1 and Rs2 for Op2) and the register that receives the result (Rd for Result). In these control signals, '0' means R0 and '1' means R1. The SLT operation puts the output to 1 (integer number 1) if Op1 < Op2 and to 0 (integer number 0) otherwise, considering that the operands are signed numbers in two's complement.
The 5 bits control word is (ALUOp,Rs1,Rs2,Rd). For example, when the control word is ¨00011¨ the executed operation is R1 <= R0 + R1;

a) Choose a control word for setting R0 to 0

b) Choose a control word for setting R1 to NOT(R0)

c) Given the initial values R0=0x08 y R1=0xFF, what is the final value of R1 after executing the sequence of instructions: 1. "00010" y 2. "11101"

d) Given the initial values R0=0xFE y R1=0x05, what is the final value of R1 after executing the sequence of instructions: 1. "11011" y 2. "10011"

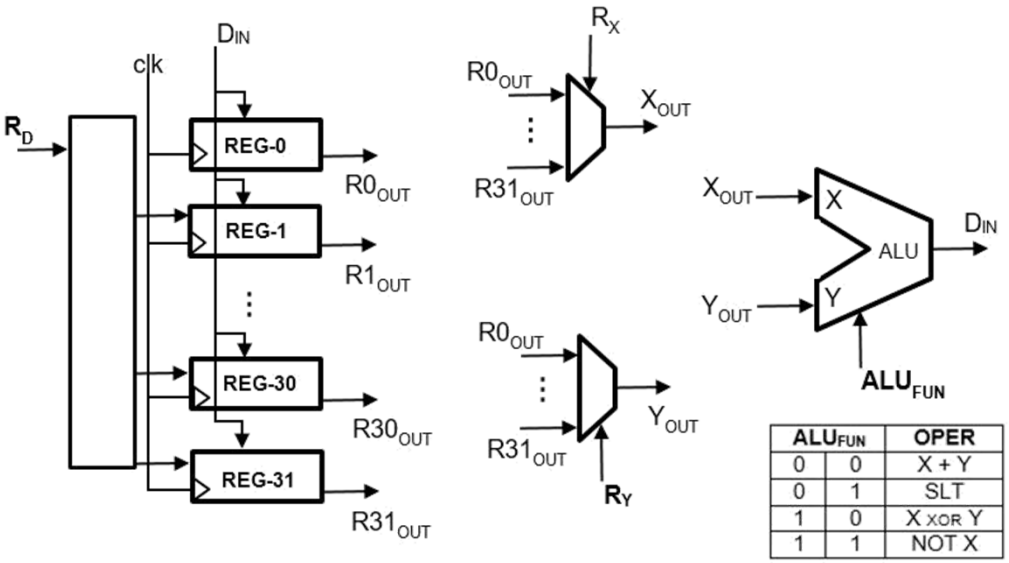


ALUOp	Operación
00	Result=Op1+Op2
01	Result=NAND(Op1,Op2)
10	Result=XOR(Op1,Op2)
11	Result=SLT(Op1,Op2)

Unit 2 exercises

2.10. The figure shows an architecture using some combinational circuits, such as multiplexers or decoders and an ALU, and sequential circuits, which are 32 registers. As usual, register R0 is an only read register stuck at 0. All the registers are 32 bits wide and with unknown values, except for R0. There are four control signals with different widths, R_D, R_X, R_Y and ALU_{FUN}.

The SLT operation puts a 1 in the destination register when the X < Y, otherwise it puts a 0.



ALU_{FUN}		OPER
0	0	X + Y
0	1	SLT
1	0	X XOR Y
1	1	NOT X

a) Indicate the number of bits and functionality of the control signal "R_D". Reason your answer.

b) Indicate the number of bits and functionality of the control signal "R_X". Reason your answer.

c) Write a code, with its control words, in order to obtain R5 <= 0xFFFFFFFF.

d) Write a code, with its control words, in order to obtain R4 <= Two's complement of R3.

NOTE: Control words must written in the order RD, RX, RY, ALUFUN. Use commas for separating each part.

T