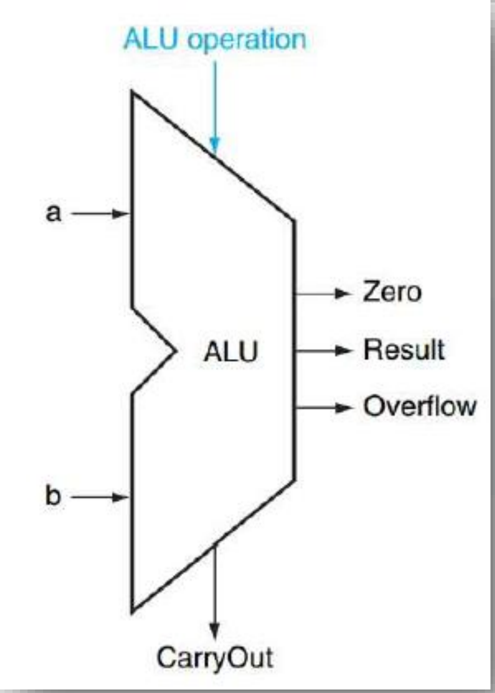


Arithmetic and Logic Unit (ALU) Design

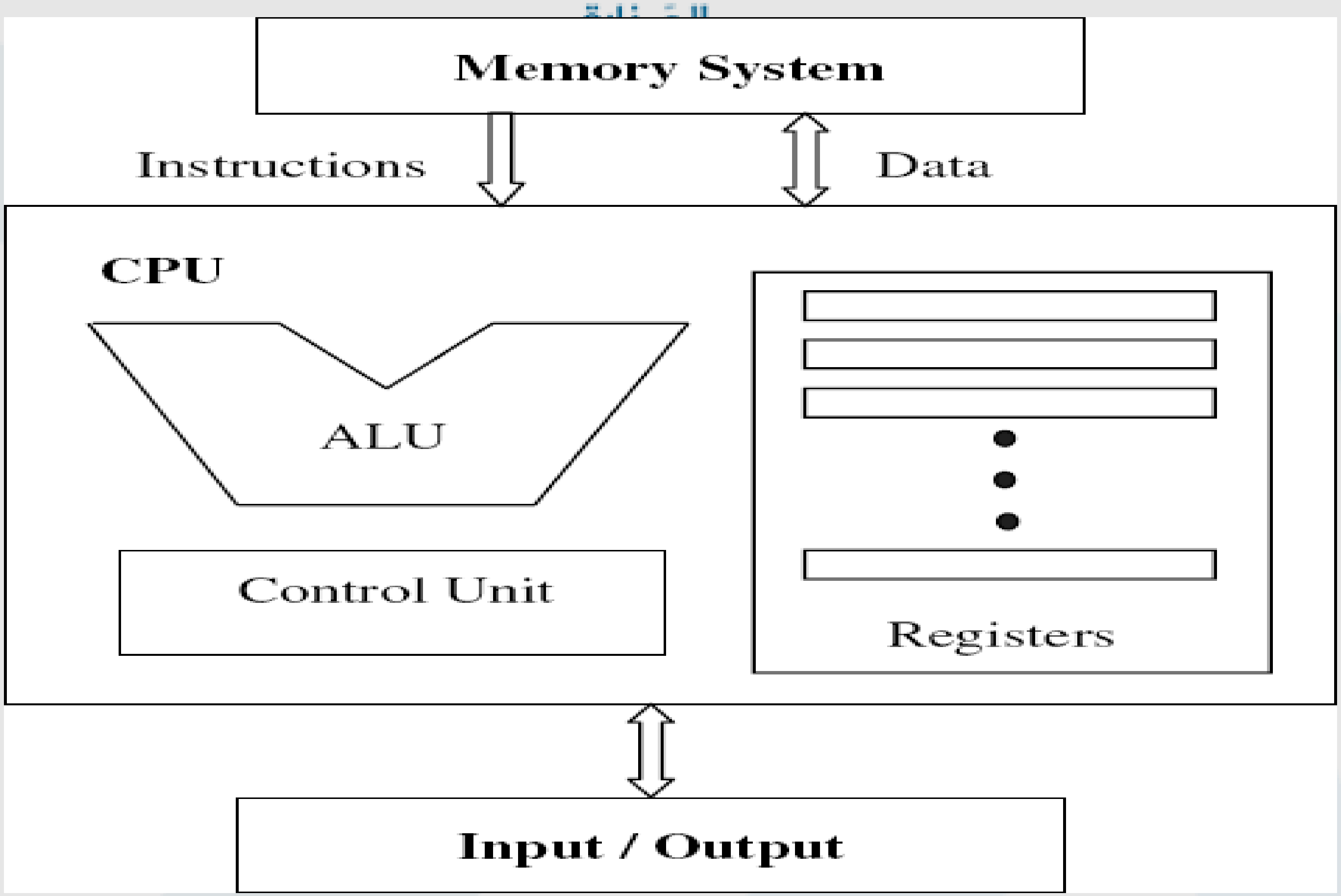
Computer Architecture

Lectures 5-6

Mechatronics Engineering

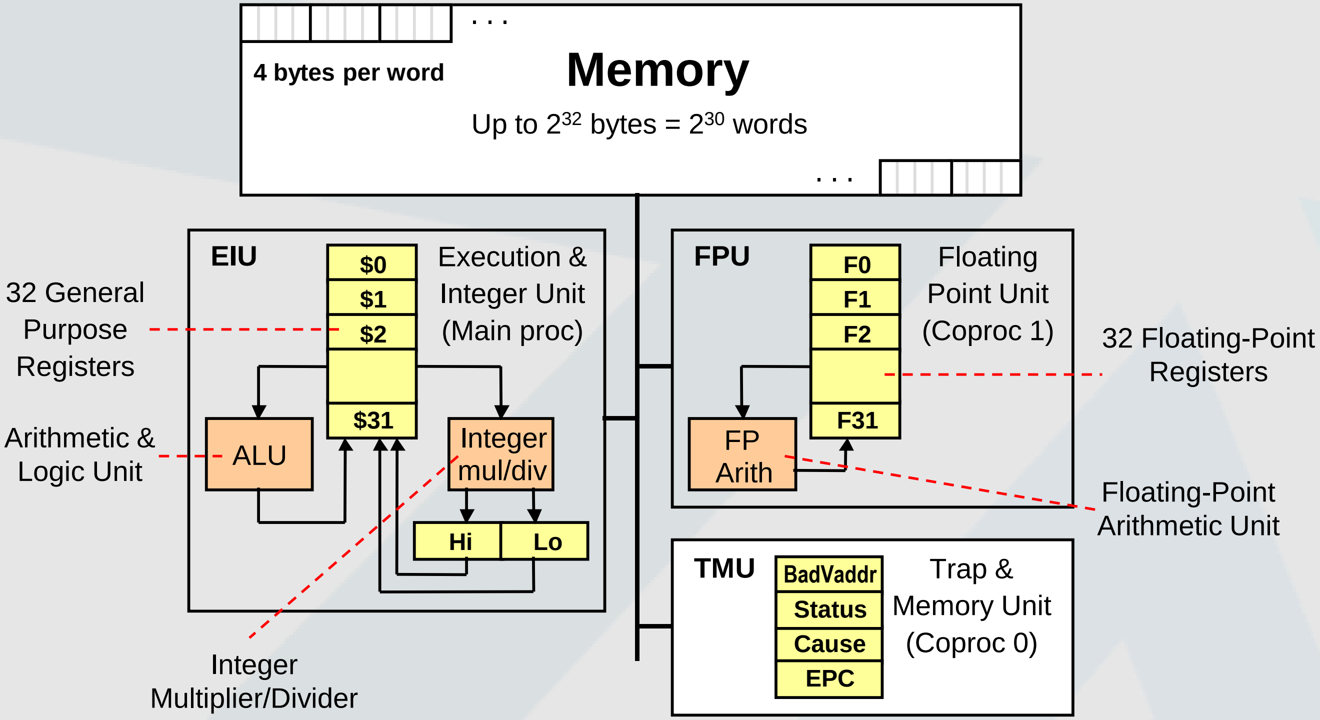
Assistant Professor: Isam M. Asaad

Computer Architecture



Central processing unit main components and interactions with the memory and I/O

Overview of the MIPS Architecture


. . .
4 bytes per word
Memory
Up to 2^{32} bytes = 2^{30} words
. . .
EIU
$0
$1
$2
$31
Execution & Integer Unit (Main proc)
32 General Purpose Registers
Arithmetic & Logic Unit
ALU
Integer mul/div
Hi
Lo
Integer Multiplier/Divider
FPU
F0
F1
F2
F31
Floating Point Unit (Coproc 1)
32 Floating-Point Registers
FP Arith
Floating-Point Arithmetic Unit
TMU
BadVaddr
Status
Cause
EPC
Trap & Memory Unit (Coproc 0)

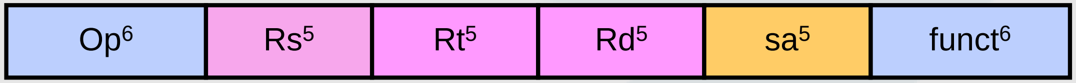
MIPS Register Conventions

- Assembler can refer to registers by name or by number
  - It is easier for you to remember registers by name
  - Assembler converts register name to its corresponding number

Name	Register	Usage	
`$zero`	`$0`	Always 0	(forced by hardware)
`$at`	`$1`	Reserved for assembler use	
`$v0 – $v1`	`$2 – $3`	Result values of a function	
`$a0 – $a3`	`$4 – $7`	Arguments of a function	
`$t0 – $t7`	`$8 – $15`	Temporary Values	
`$s0 – $s7`	`$16 – $23`	Saved registers	(preserved across call)
`$t8 – $t9`	`$24 – $25`	More temporaries	
`$k0 – $k1`	`$26 – $27`	Reserved for OS kernel	
`$gp`	`$28`	Global pointer	(points to global data)
`$sp`	`$29`	Stack pointer	(points to top of stack)
`$fp`	`$30`	Frame pointer	(points to stack frame)
`$ra`	`$31`	Return address	(used by jal for function call)

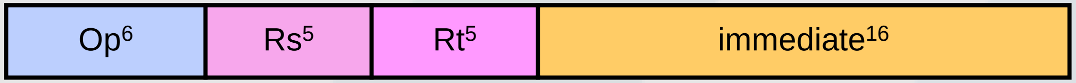
Instruction Formats

- All instructions are 32-bit wide, Three instruction formats:
- Register (R-Type)
  - – Register-to-register instructions
  - – Op: operation code specifies the format of the instruction



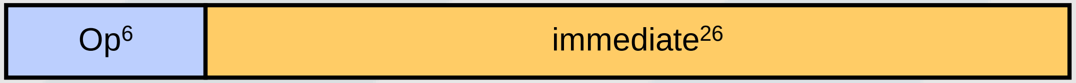
Op^6	Rs^5	Rt^5	Rd^5	sa^5	funct^6

- Immediate (I-Type)
  - – 16-bit immediate constant is part in the instruction



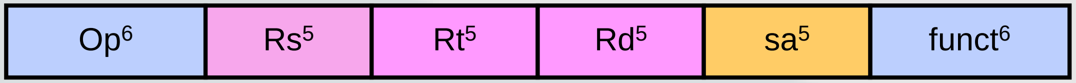
Op^6	Rs^5	Rt^5	immediate^{16}

- Jump (J-Type)
  - – Used by jump instructions



Op^6	immediate^{26}

R-Type Format



Op^6	Rs^5	Rt^5	Rd^5	sa^5	funct^6

- **Op**: operation code (opcode)
  - Specifies the operation of the instruction
  - Also specifies the format of the instruction
- **funct**: function code – extends the opcode
  - Up to $2^6 = 64$ functions can be defined for the same opcode
  - MIPS uses opcode 0 to define R-type instructions
- Three Register Operands (common to many instructions)
  - **Rs**, **Rt**: first and second source operands
  - **Rd**: destination operand
  - **sa**: the shift amount used by shift instructions

Integer Add /Subtract Instructions

Instruction	Meaning	R-Type Format					
add $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x20
addu $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x21
sub $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x22
subu $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x23

- ❖ add & sub: overflow causes an arithmetic exception
  - ✧ In case of overflow, result is not written to destination register
- ❖ addu & subu: same operation as add & sub
  - ✧ However, no arithmetic exception can occur
  - ✧ **Overflow is ignored**
- ❖ Many programming languages ignore overflow
  - ✧ The + operator is translated into **addu**
  - ✧ The – operator is translated into **subu**

Logical Bitwise Instructions

Instruction	Meaning	R-Type Format					
and $s1, $s2, $s3	$s1 = $s2 & $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x24
or $s1, $s2, $s3	$s1 = $s2 | $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x25
xor $s1, $s2, $s3	$s1 = $s2 ^ $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x26
nor $s1, $s2, $s3	$s1 = ~($s2|$s3)	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x27

- Examples:

Assume `$s1 = 0xabcd1234` and `$s2 = 0xffff0000`

```
and $s0,$s1,$s2      # $s0 = 0xabcd0000
or  $s0,$s1,$s2      # $s0 = 0xffff1234
xor $s0,$s1,$s2      # $s0 = 0x54321234
nor $s0,$s1,$s2      # $s0 = 0x0000edcb
```

I-Type Format

- ❖ Constants are used quite frequently in programs
  - ✧ The R-type shift instructions have a 5-bit shift amount constant
  - ✧ What about other instructions that need a constant?
- ❖ I-Type: Instructions with Immediate Operands

Op^6	Rs^5	Rt^5	immediate^{16}

- ❖ 16-bit immediate constant is stored inside the instruction
  - ✧ Rs is the source register number
  - ✧ Rt is now the destination register number (for R-type it was Rd)
- ❖ Examples of I-Type ALU Instructions:
  - ✧ Add immediate: `addi  $s1, $s2, 5        # $s1 = $s2 + 5`
  - ✧ OR immediate: `ori   $s1, $s2, 5        # $s1 = $s2 | 5`

I-Type ALU Instructions

Instruction	Meaning	I-Type Format			
addi $s1, $s2, 10	$s1 = $s2 + 10	op = 0x8	rs = $s2	rt = $s1	imm^{16} = 10
addiu $s1, $s2, 10	$s1 = $s2 + 10	op = 0x9	rs = $s2	rt = $s1	imm^{16} = 10
andi $s1, $s2, 10	$s1 = $s2 & 10	op = 0xc	rs = $s2	rt = $s1	imm^{16} = 10
ori $s1, $s2, 10	$s1 = $s2 | 10	op = 0xd	rs = $s2	rt = $s1	imm^{16} = 10
xori $s1, $s2, 10	$s1 = $s2 ^ 10	op = 0xe	rs = $s2	rt = $s1	imm^{16} = 10
lui $s1, 10	$s1 = 10 << 16	op = 0xf	0	rt = $s1	imm^{16} = 10

- addi: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register
- addiu: same operation as addi but overflow is ignored
- Immediate constant for addi and addiu is signed
  - No need for **subi** or **subiu** instructions
- Immediate constant for andi, ori, xori is unsigned

Examples: I-Type ALU Instructions

- Examples: assume A, B, C are allocated $s0, $s1, $s2

`A = B+5;`	translated as	`addiu $s0,$s1,5`
`C = B–1;`	translated as	`addiu $s2,$s1,-1`

op=001001	rs=$s1=10001	rt=$s2=10010	imm = -1 = 1111111111111111

`A = B&0xf;`	translated as	`andi $s0,$s1,0xf`
`C = B|0xf;`	translated as	`ori $s2,$s1,0xf`
`C = 5;`	translated as	`ori $s2,$zero,5`
`A = B;`	translated as	`ori $s0,$s1,0`

Immediate value is the last value in the instruction

- No need for `subi`, because `addi` has signed immediate
- Register 0 (`$zero`) has always the value 0

Instruction Categories

- Integer Arithmetic
  - Arithmetic, logical, and shift instructions
- Data Transfer
  - Load and store instructions that access memory
  - Data movement and conversions
- Jump and Branch
  - Flow-control instructions that alter the sequential sequence
- Floating Point Arithmetic
  - Instructions that operate on floating-point registers
- Miscellaneous
  - Instructions that transfer control to/from exception handlers
  - Memory management instructions (e.g. stack)

Logic Design

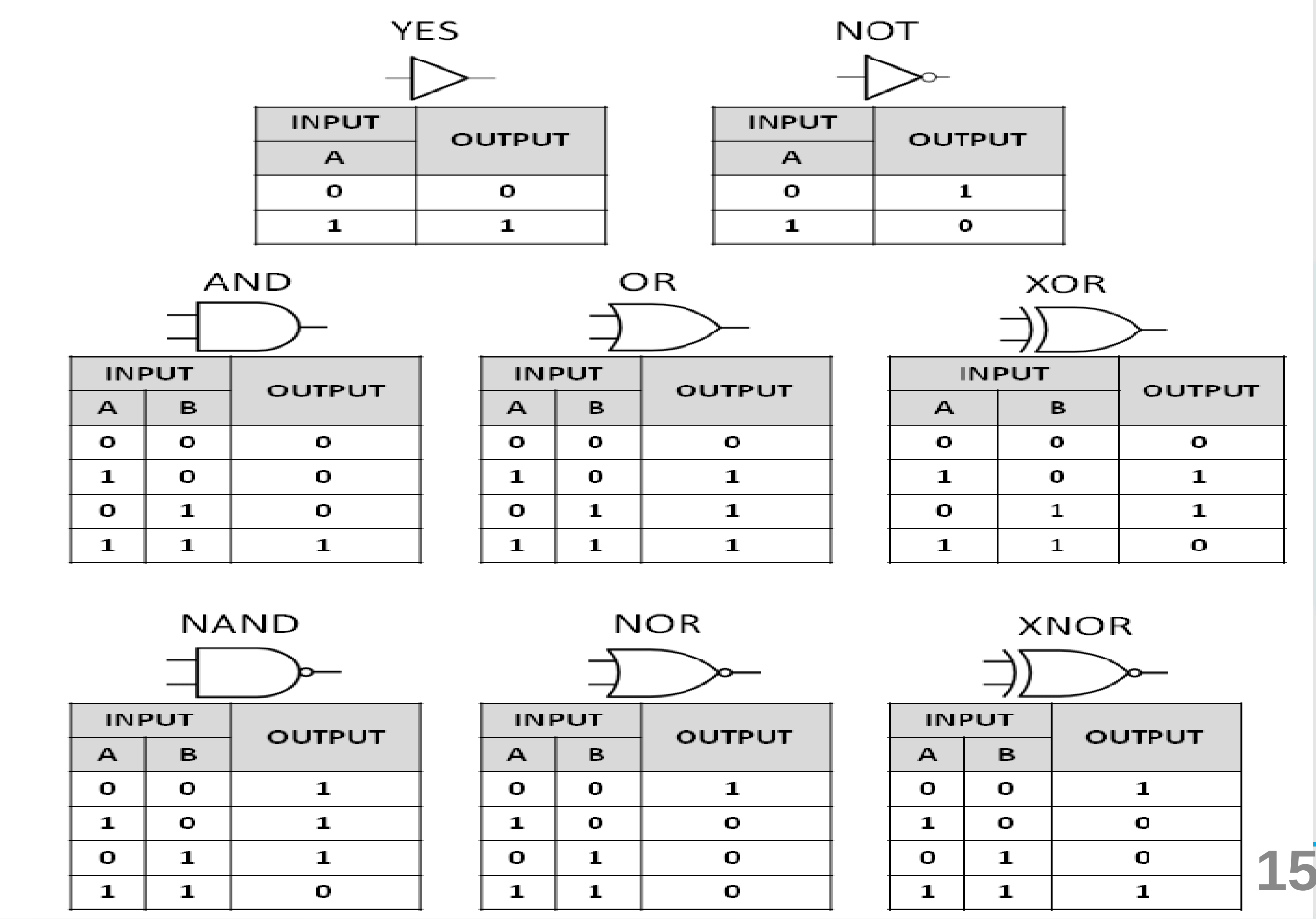
The datapath elements in the MIPS implementation consist of two different types of logic elements: elements that operate on data values and elements that contain state.

- The elements that operate on data values are all **combinational**, which means that their outputs depend only on the current inputs.
- Decoder and Selector/Multiplexor are examples of **combinational elements.**
- Given the same input, a combinational element always produces the same output.
- The ALU is an example of a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

Logic Design

- Other elements in the design are not combinational, but instead contain *state*.
- An element contains state if it has some internal storage. We call these elements **state elements**.
- A state element has at least two inputs and one output. The required inputs are the **data** value to be written into the element and the **clock**, which determines when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle.
- For example, one of the logically simplest state elements is a D-type flip-flop
- In addition to flip-flops, our MIPS implementation also uses two other types of state elements: **memories** and **registers**.
- **The clock is used to determine when the state element should be written; a state element can be read at any time.**

Logic Design (Logic gates)



YES

INPUT	OUTPUT
A	
0	0
1	1

NOT

INPUT	OUTPUT
A	
0	1
1	0

AND

INPUT		OUTPUT
A	B	
0	0	0
1	0	0
0	1	0
1	1	1

OR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

XOR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	0

NAND

INPUT		OUTPUT
A	B	
0	0	1
1	0	1
0	1	1
1	1	0

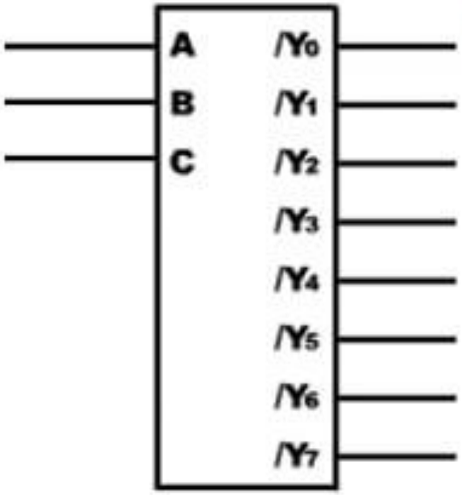
NOR

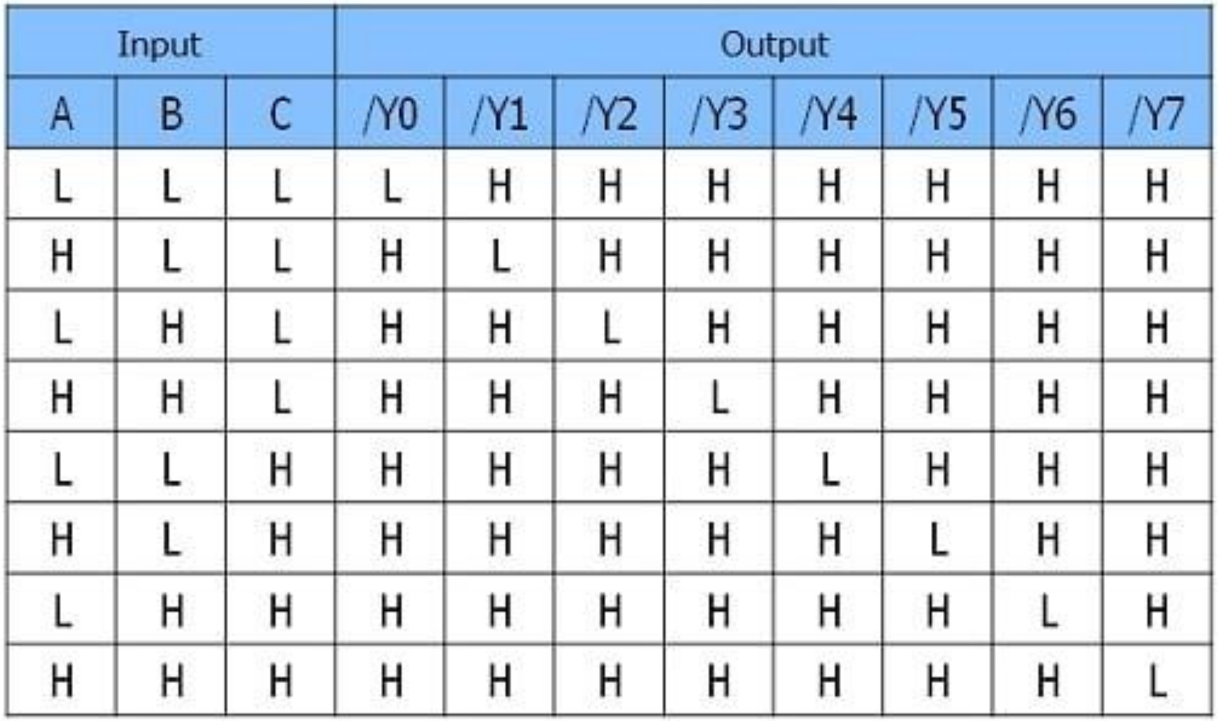
INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	0

XNOR

INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	1

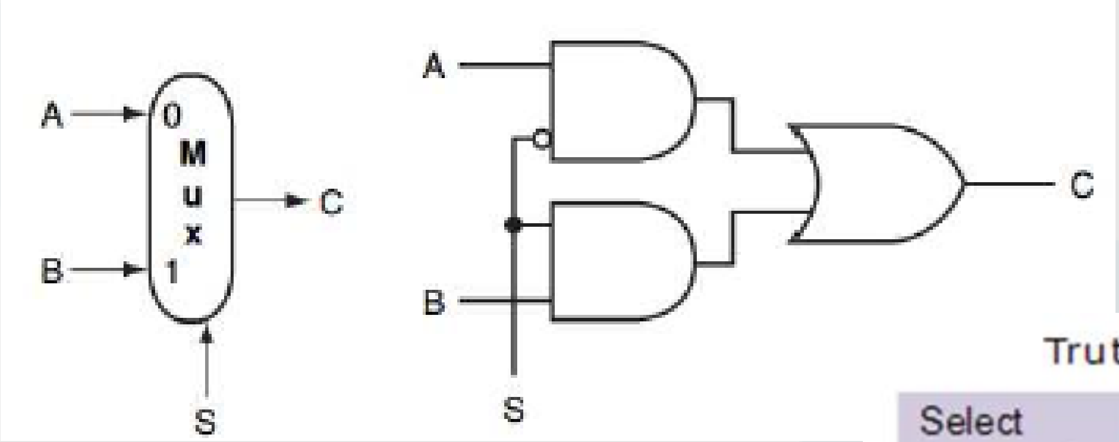
Logic Design (Decoder)





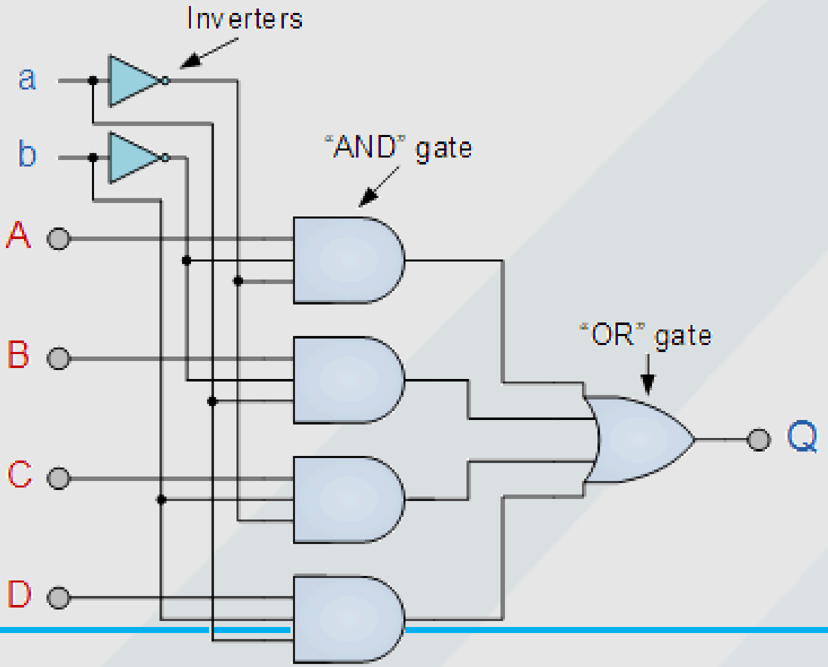
Input			Output							
A	B	C	/Y0	/Y1	/Y2	/Y3	/Y4	/Y5	/Y6	/Y7
L	L	L	L	H	H	H	H	H	H	H
H	L	L	H	L	H	H	H	H	H	H
L	H	L	H	H	L	H	H	H	H	H
H	H	L	H	H	H	L	H	H	H	H
L	L	H	H	H	H	H	L	H	H	H
H	L	H	H	H	H	H	H	L	H	H
L	H	H	H	H	H	H	H	H	L	H
H	H	H	H	H	H	H	H	H	H	L

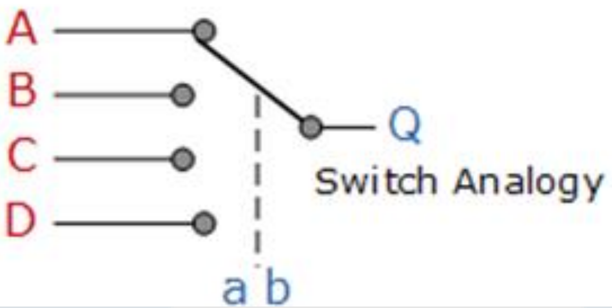
Logic Design (Multiplexor)



Truth Table

Select		Inputs				
b	a	D	C	B	A	Q
0	0	x	x	x	1	1
0	1	x	x	1	x	1
1	0	x	1	x	x	1
1	1	1	x	x	x	1



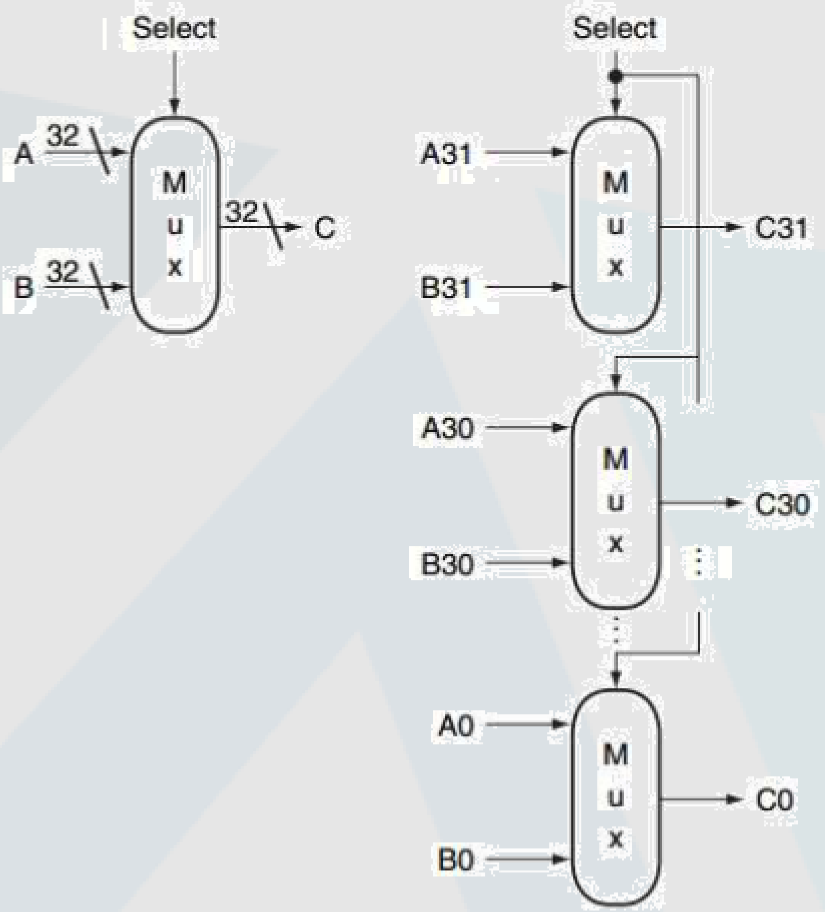


Switch Analogy

Use of multiplexors to choose values on busses

Bus:

- **Definition**: A group of lines (data) treated as a single logical signal.
  - Example: Lines that transfer a value from a register to memory (32-bit wide)
- We'll use the multiplexor to select the source of the values that appears on the bus:
  - Bus width: 32 bits
  - Number of possible sources: 2
  - Number of selection lines needed: 1
  - We need 32 simple multiplexor to form an array of multiplexors.

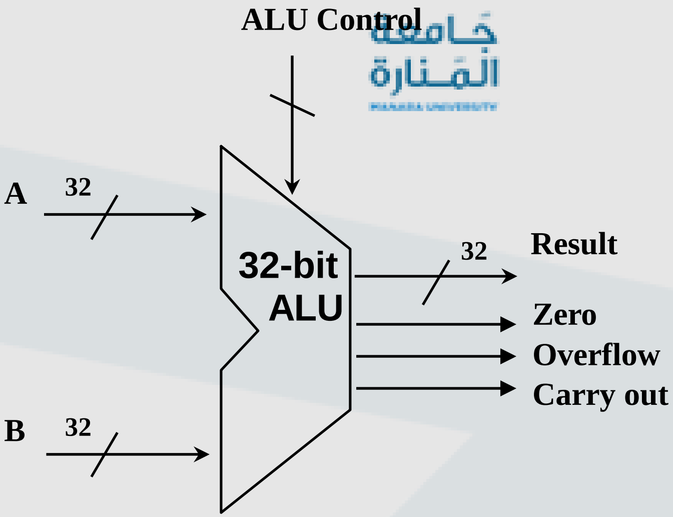


Design of an Arithmetic and logic Unit (ALU)

Steps:

1. Design of a 1-bit ALU that can achieve:
    1. AND
    2. OR
2. Building a 32-bit ALU.
3. Adding the ability of achieving summation (with carry).
4. Adding the ability of achieving some MIPS processor’s instructions:
    1. slt (set on less than)
    2. Conditional branch (beq, bne)
5. Adding the ability of achieving NOR.

32-bit ALU



- Our ALU should be able to perform functions:
  - – logical and function
  - – logical or function
  - – arithmetic add function
  - – arithmetic subtract function
  - – arithmetic slt (set-less-than) function
  - – logical nor function
- ALU control lines define a function to be performed on A and B.

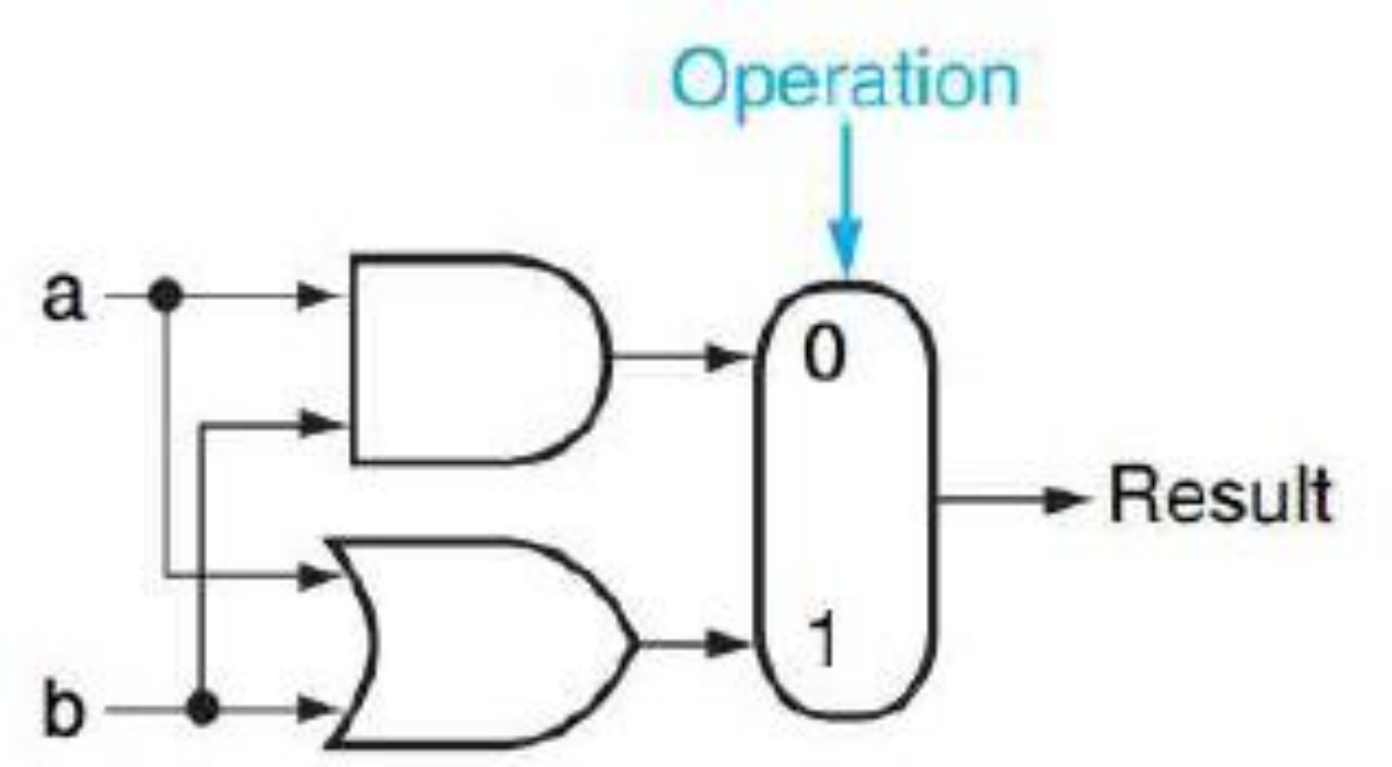
Design of an Arithmetic and logic Unit (ALU)

Steps:

1. Design of a 1-bit ALU that can achieve:
    1. AND
    2. OR
2. Building a 32-bit ALU.
3. Adding the ability of achieving summation (with carry).
4. Adding the ability of achieving some MIPS processor’s instructions:
    1. slt (set on less than)
    2. Conditional branch (beq, bne)
5. Adding the ability of achieving NOR.

1-Bit ALU for AND and OR

- We will use a selection line (Operation) to select which result to get.



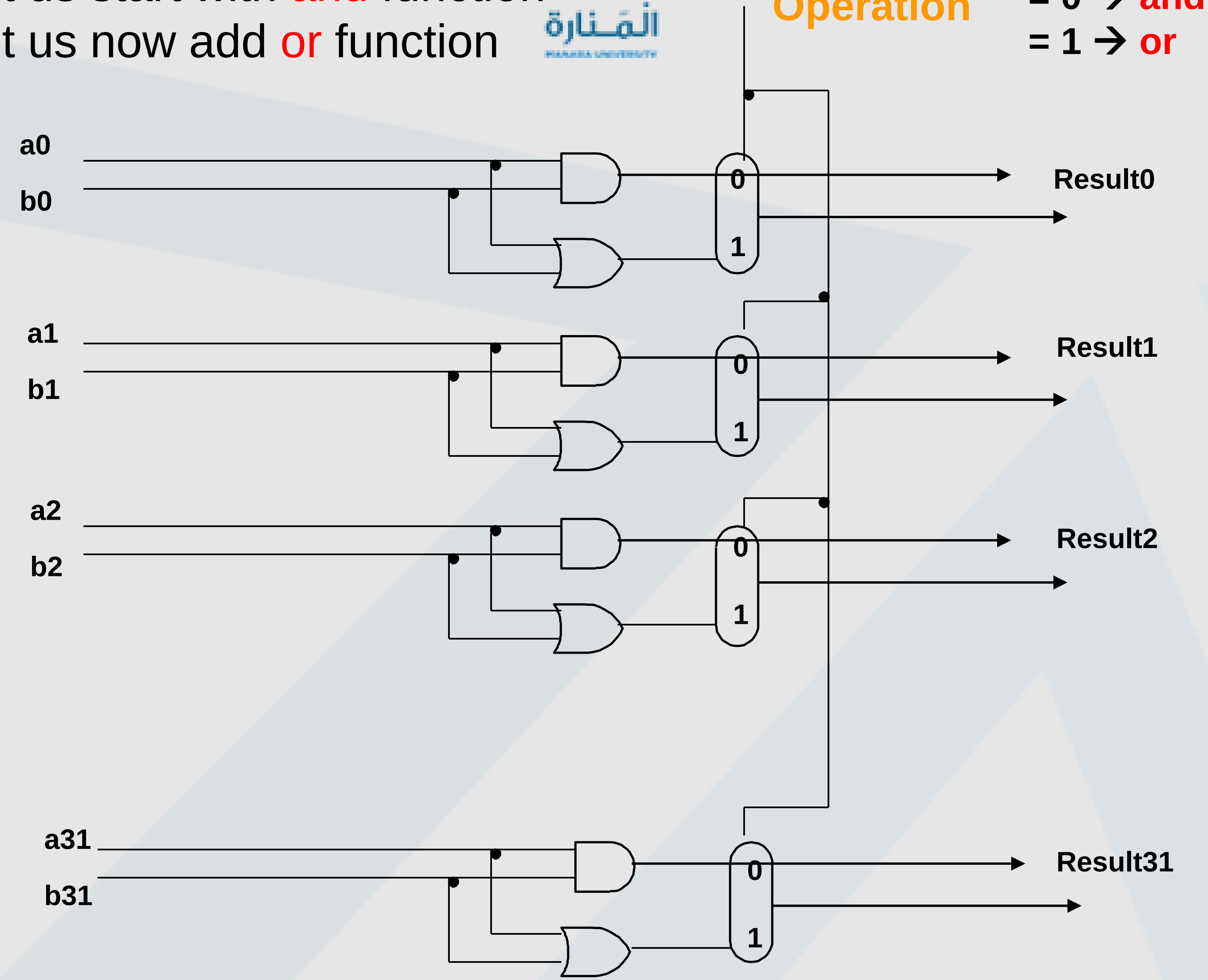
Design of an Arithmetic and logic Unit (ALU)

Steps:

1. Design of a 1-bit ALU that can achieve:
    1. AND
    2. OR
2. Building a 32-bit ALU.
3. Adding the ability of achieving summation (with carry).
4. Adding the ability of achieving some MIPS processor’s instructions:
    1. slt (set on less than)
    2. Conditional branch (beq, bne)
5. Adding the ability of achieving NOR.

Designing a 32-bit ALU: Beginning

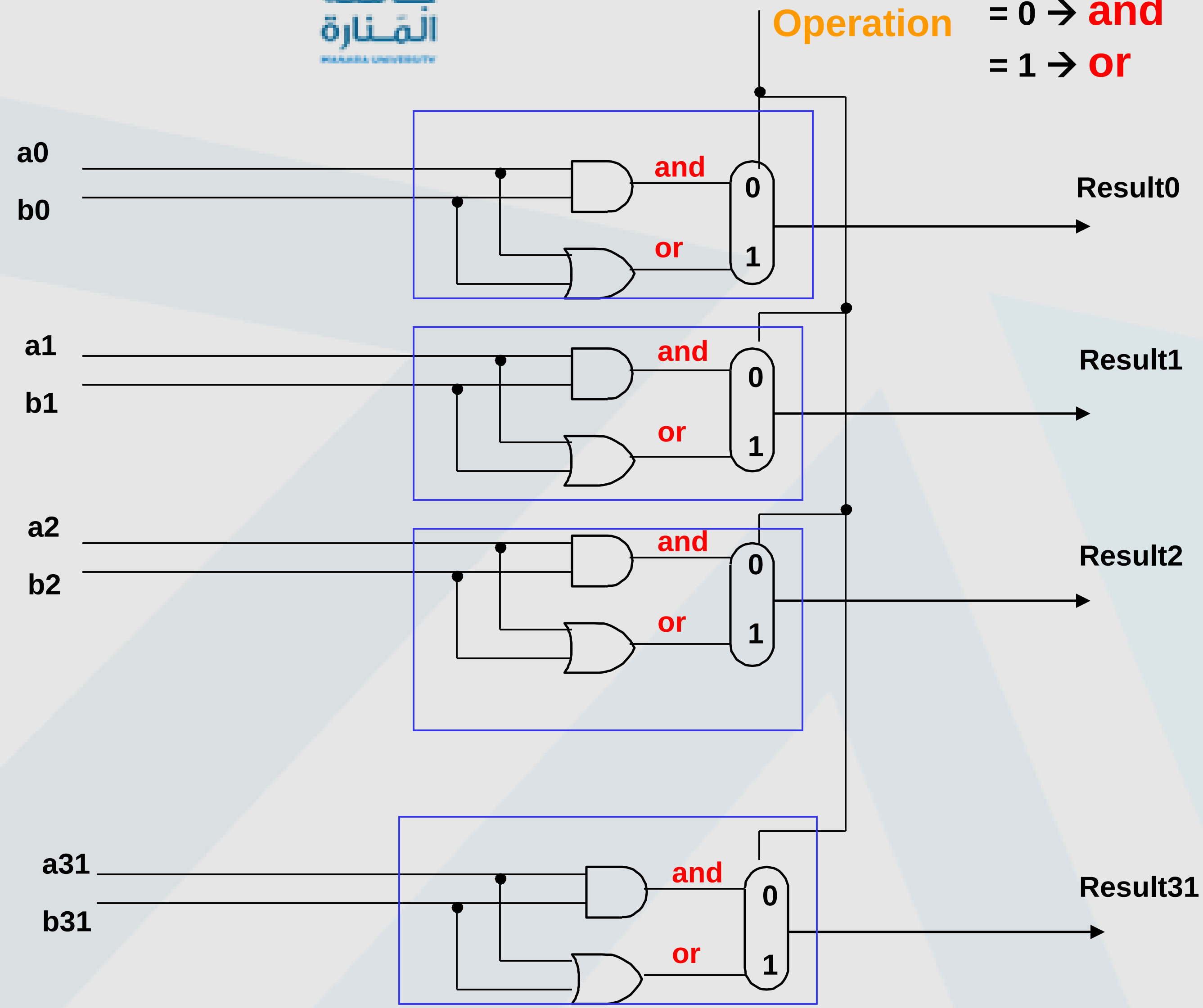
1. Let us start with and function
2. Let us now add or function



Designing 32-bit ALU: Principles

- Number of functions are performed internally, but only one result is chosen for the output of ALU

- 32-bit ALU is built out of 32 identical 1-bit ALU's

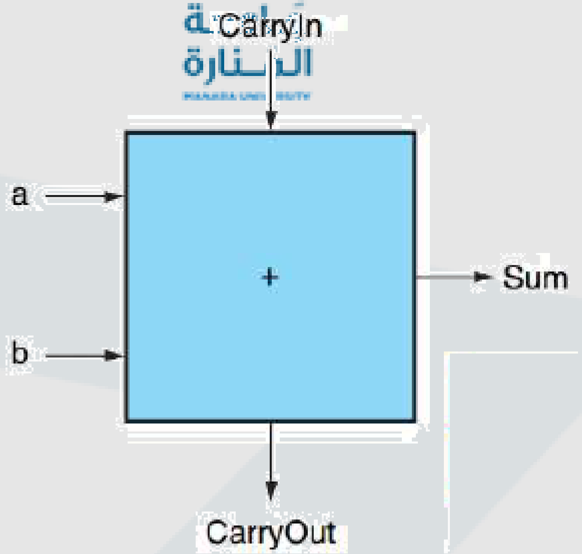


Design of an Arithmetic and logic Unit (ALU)

Steps:

1. Design of a 1-bit ALU that can achieve:
    1. AND
    2. OR
2. Building a 32-bit ALU.
3. Adding the ability of achieving summation (with carry).
4. Adding the ability of achieving some MIPS processor’s instructions:
    1. slt (set on less than)
    2. Conditional branch (beq, bne)
5. Adding the ability of achieving NOR.

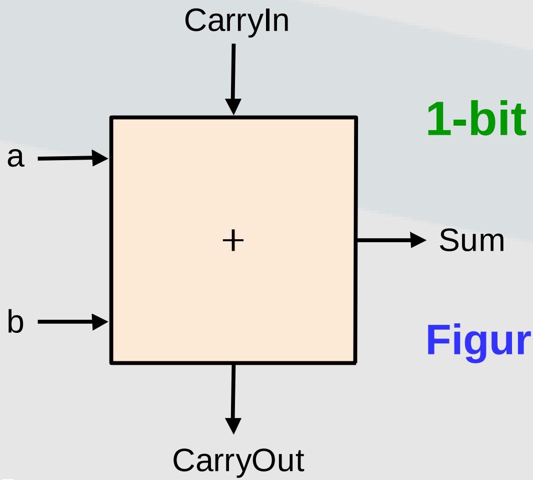
Full Adder



Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

Designing Adder

- 32-bit adder is built out of 32 1-bit adders



1-bit Adder

Figure B.5.2

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

$$\Downarrow$$

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

From the truth table and after minimization, we can have this design for CarryOut

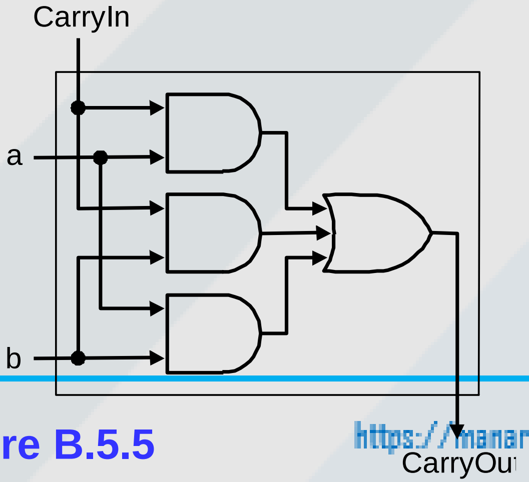


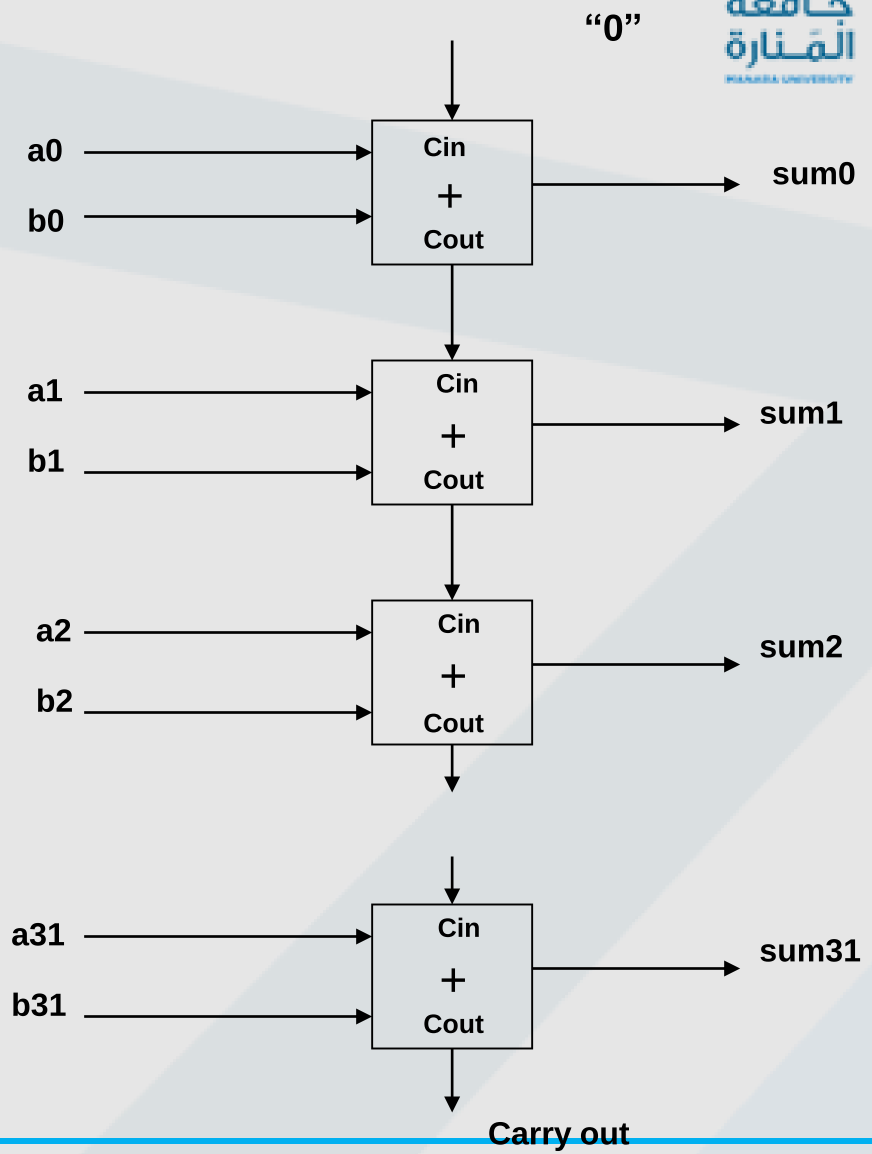
Figure B.5.5

1-bit Adder Truth Table

Input			Output	
a	b	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

32-bit Adder



This is a ripple carry adder.

32-bit ALU With 3 Functions

1-bit ALU

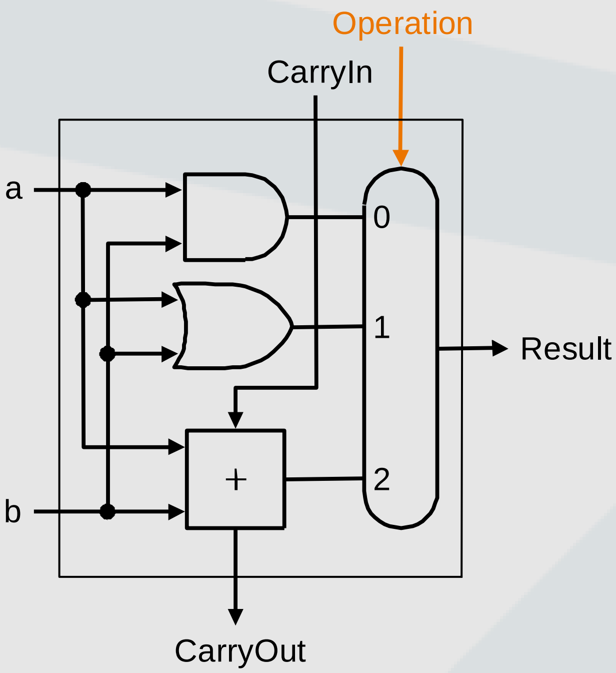
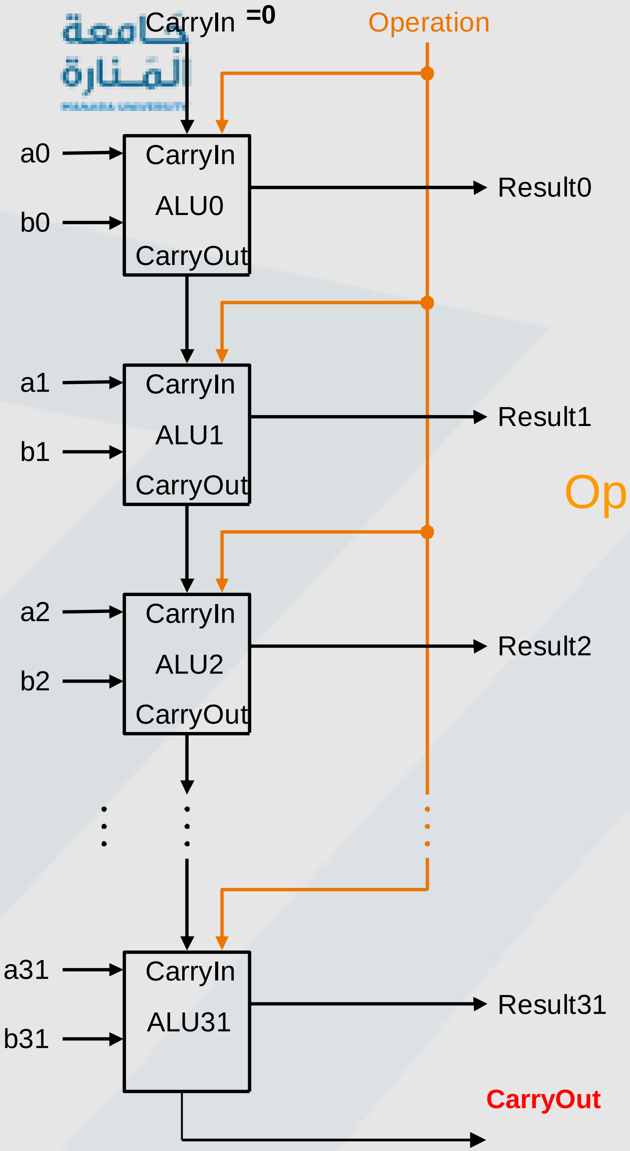


Figure B.5.6

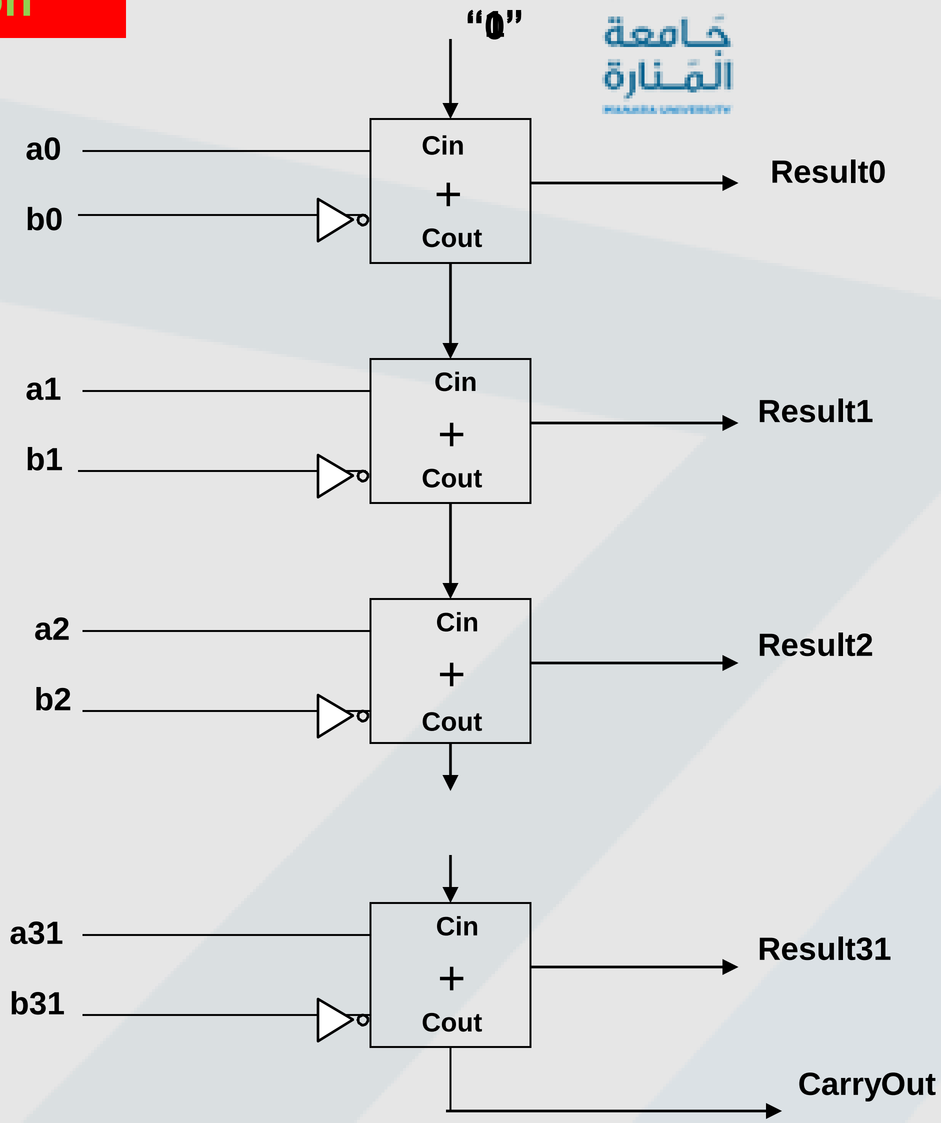


Operation = 00 → **and**
= 01 → **or**
= 10 → **add**

Figure B.5.7
+ carry out

Some parts are moving in the presentation

32-bit Subtractor

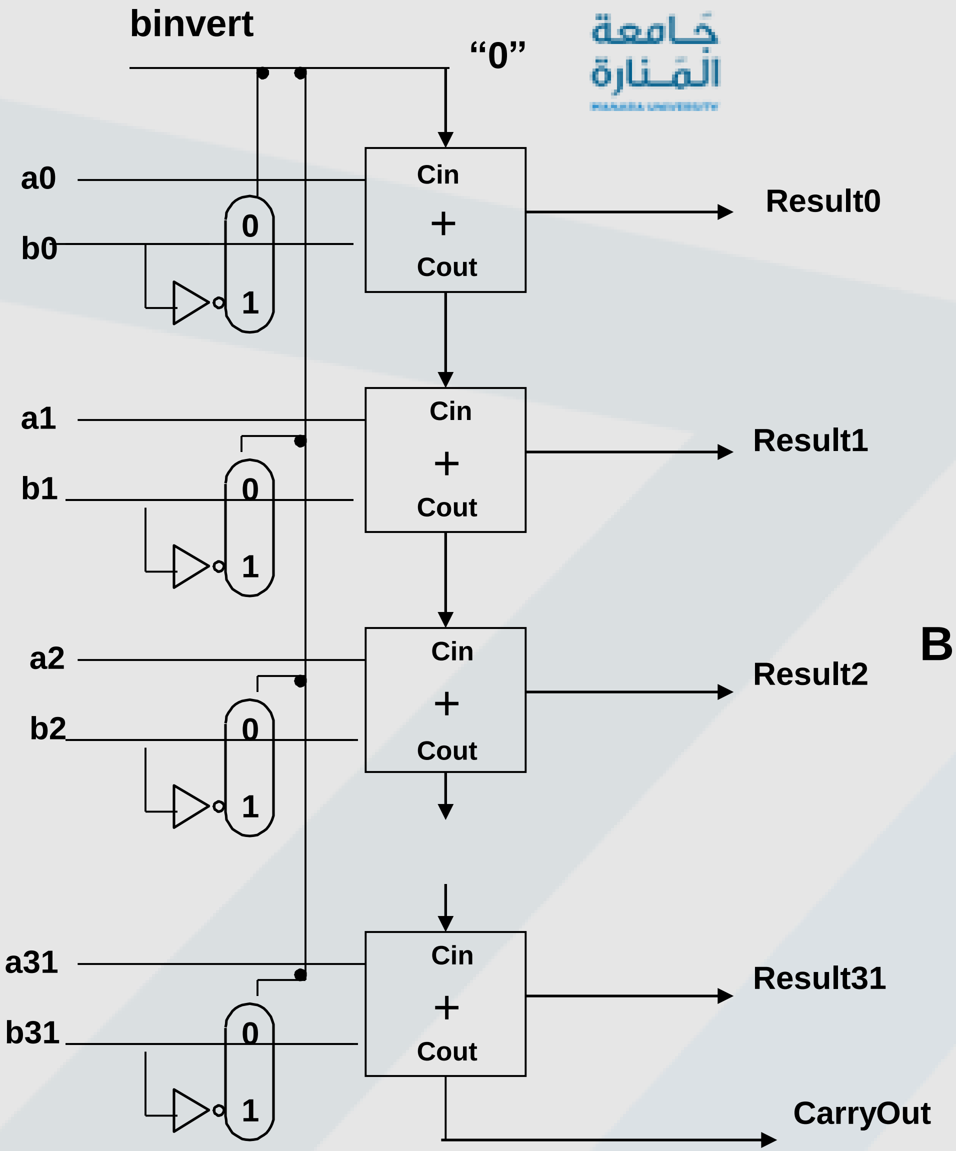


- Subtraction is equivalent to the addition of the negative value.
- We can get the negative value by calculating the two’s complement.
- We need to add:
  - The ability of negating of one input (b).
  - Least significant CarryIn=1.

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

$$= A + \overline{B} + 1$$

32-bit Adder / Subtractor



Binvert = 0 → addition
= 1 → subtraction

and least significant CarryIn=1

32-bit ALU With 4 Functions

1-bit ALU

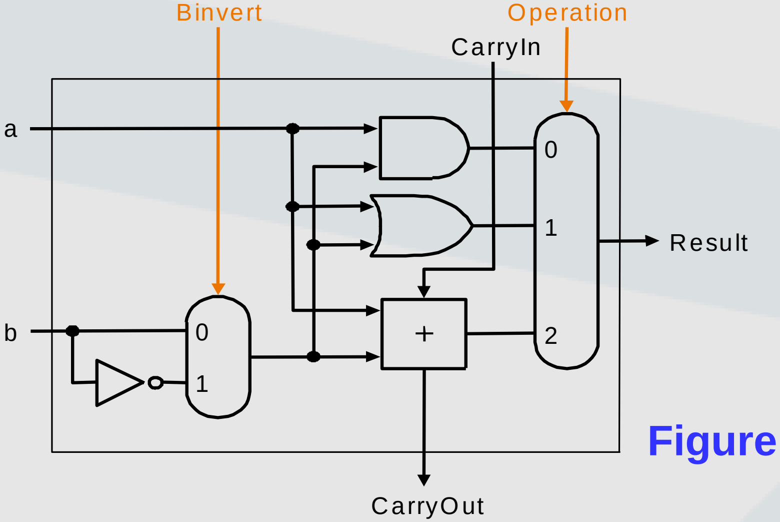
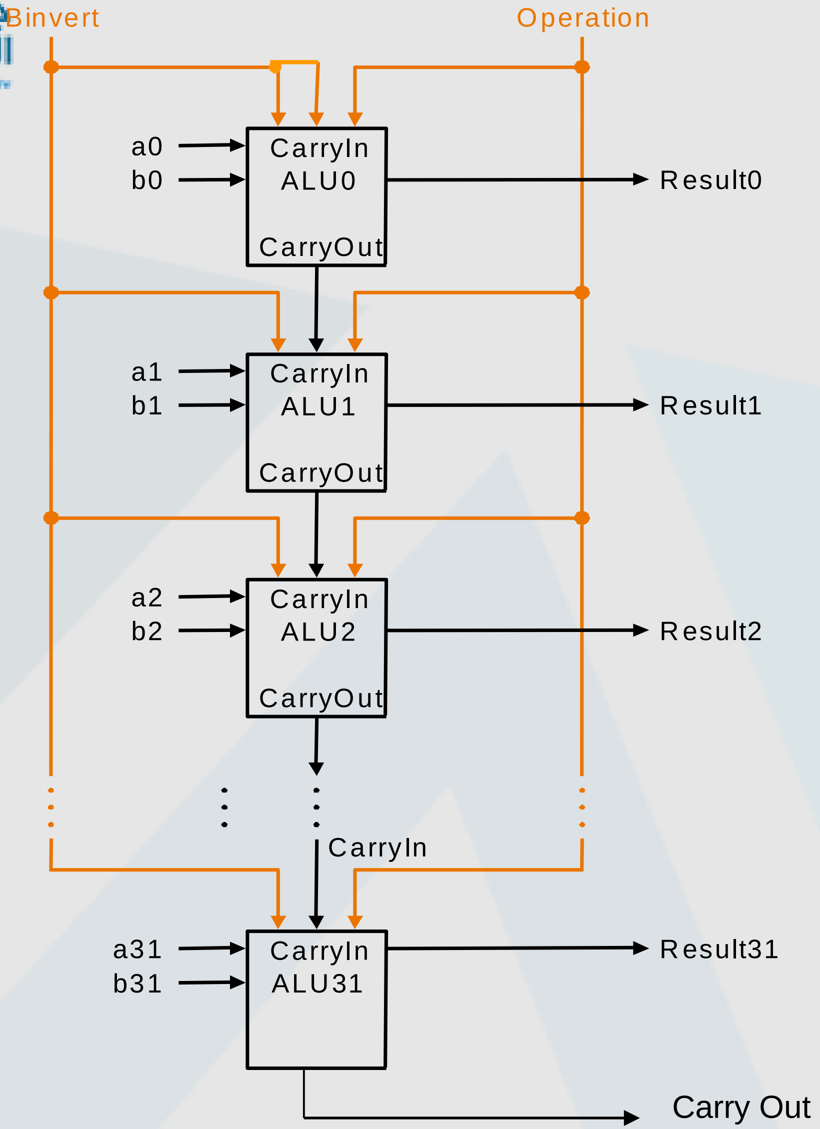


Figure B.5.8



	Control lines	
Function	**Binvert** (1 line)	**Operation** (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10

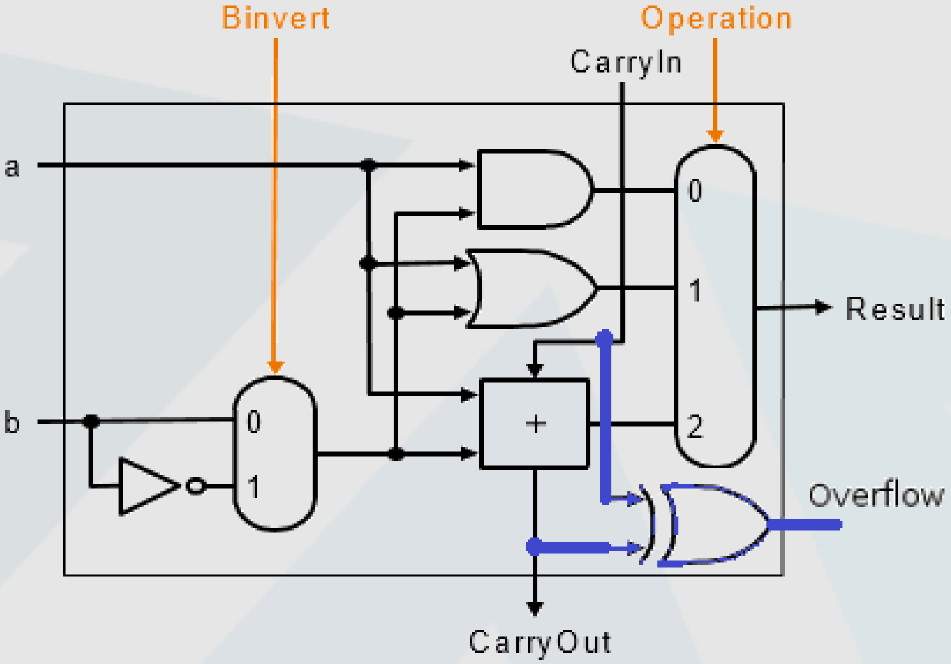
2’s Complement Overflow

1-bit ALU for the most significant bit

2’s complement overflow happens:

- **if sum of two positive numbers results in a negative number**
- **if sum of two negative numbers results in a positive number**

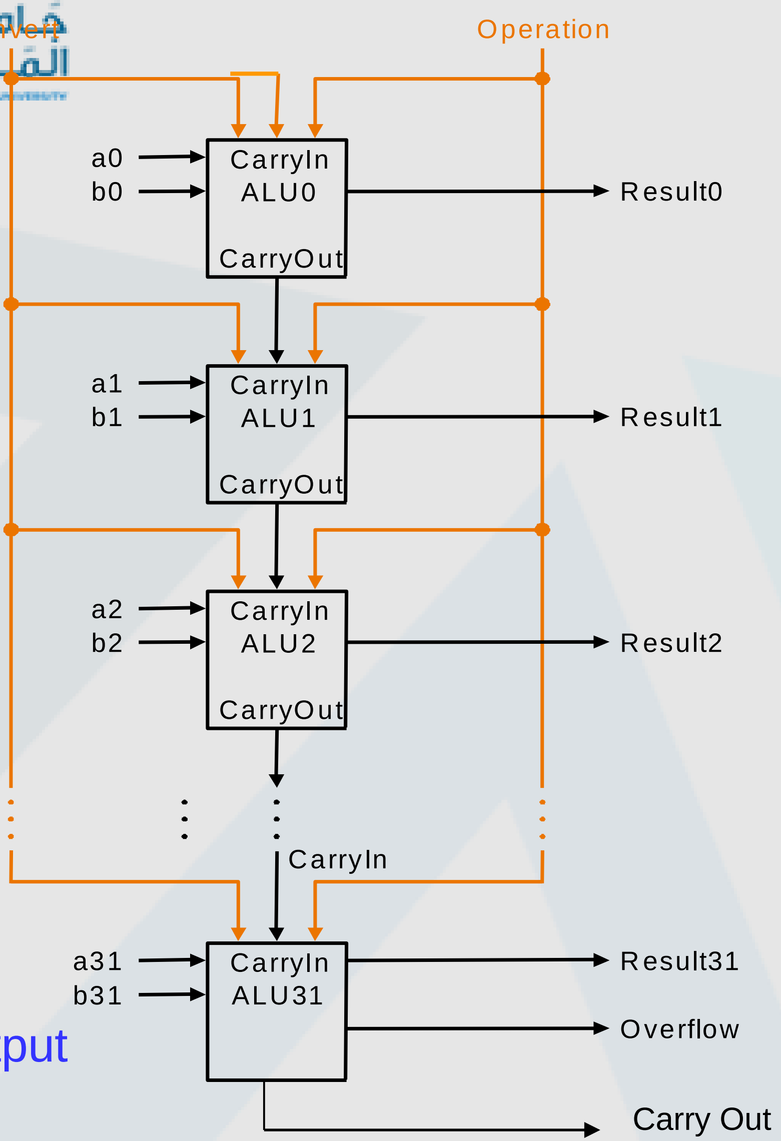
2’s complement overflow happens if the most significant full adder CarryIn doesn’t equal to the most significant full adder CarryOut.



Other 1-bit ALUs, i.e. non-most significant bit ALUs, are not affected.

32-bit ALU With 4 Functions and Overflow

Function	Control lines Binvert (1 line)	 Operation (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10



Missing: slt & nor functions and Zero output

Design of an Arithmetic and logic Unit (ALU)

Steps:

1. Design of a 1-bit ALU that can achieve:
    1. AND
    2. OR
2. Building a 32-bit ALU.
3. Adding the ability of achieving summation (with carry).
4. Adding the ability of achieving some MIPS processor’s instructions:
    1. slt (set on less than)
    2. Conditional branch (beq, bne)
5. Adding the ability of achieving NOR.

Conditional Branch Instructions

- MIPS compare and branch instructions:

`beq Rs,Rt,label`	branch to **`label`** if (**`Rs == Rt`**)
`bne Rs,Rt,label`	branch to **`label`** if (**`Rs != Rt`**)

- MIPS compare to zero & branch instructions

Compare to zero is used frequently and implemented efficiently

`bltz Rs,label`	branch to **`label`** if (**`Rs < 0`**)
`bgtz Rs,label`	branch to **`label`** if (**`Rs > 0`**)
`blez Rs,label`	branch to **`label`** if (**`Rs <= 0`**)
`bgez Rs,label`	branch to **`label`** if (**`Rs >= 0`**)

- No need for **`beqz`** and **`bnez`** instructions. Why?

Set on Less Than Instructions

- ❖ MIPS also provides set on less than instructions

`slt   rd,rs,rt`	if (rs < rt) rd = 1 else rd = 0
`sltu  rd,rs,rt`	unsigned <
`slti  rt,rs,`im^{16}	if (rs < im^{16}) rt = 1 else rt = 0
`sltiu rt,rs,`im^{16}	unsigned <

- ❖ Signed / Unsigned Comparisons

Can produce different results

Assume `$s0 = 1` and `$s1 = -1 = 0xffffffff`

`slt  $t0,$s0,$s1`	results in	`$t0 = 0`
`sltu $t0,$s0,$s1`	results in	`$t0 = 1`

More on Branch Instructions

- MIPS hardware does NOT provide instructions for …

`blt, bltu`	branch if less than	(signed/unsigned)
`ble, bleu`	branch if less or equal	(signed/unsigned)
`bgt, bgtu`	branch if greater than	(signed/unsigned)
`bge, bgeu`	branch if greater or equal	(signed/unsigned)

Can be achieved with a sequence of 2 instructions

- How to implement:
- Solution:

```
blt $s0,$s1,label
slt $at,$s0,$s1
bne $at,$zero,label
```

- How to implement:
- Solution:

```
ble $s2,$s3,label #s2≤s3 ⇔ s3≥s2
slt $at,$s3,$s2 #(lower means smaller and not equal)
beq $at,$zero,label
```

Set Less Than (slt) Function

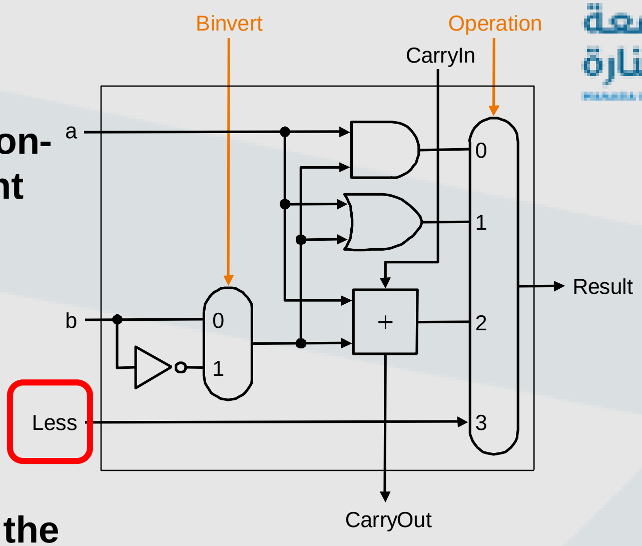
- slt function is defined as:

$$A\ \text{slt}\ B = \begin{cases} 000 \ldots 001 & \text{if } A < B, \text{ i.e. if } A - B < 0 \\ 000 \ldots 000 & \text{if } A \geq B, \text{ i.e. if } A - B \geq 0 \end{cases}$$

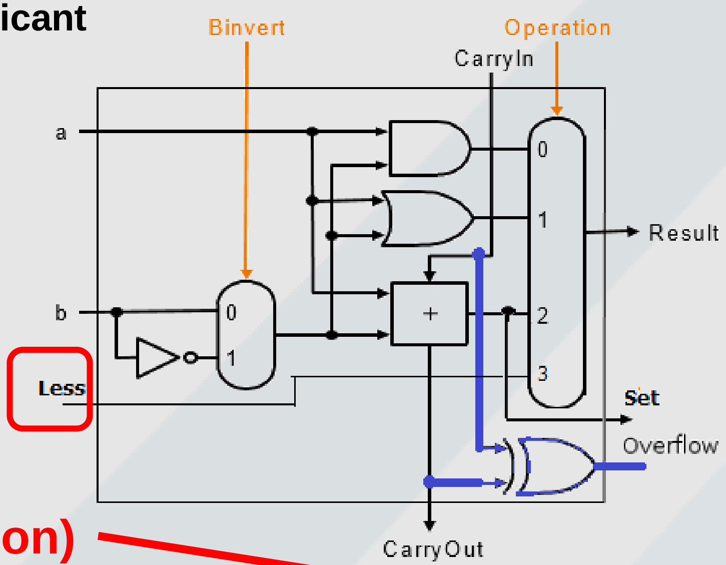
- Thus each 1-bit ALU should have an additional input (called "Less"), that will provide results for slt function. This input has value 0 for all but 1-bit ALU for the least significant bit.
- For the least significant bit Less value should be sign of A – B

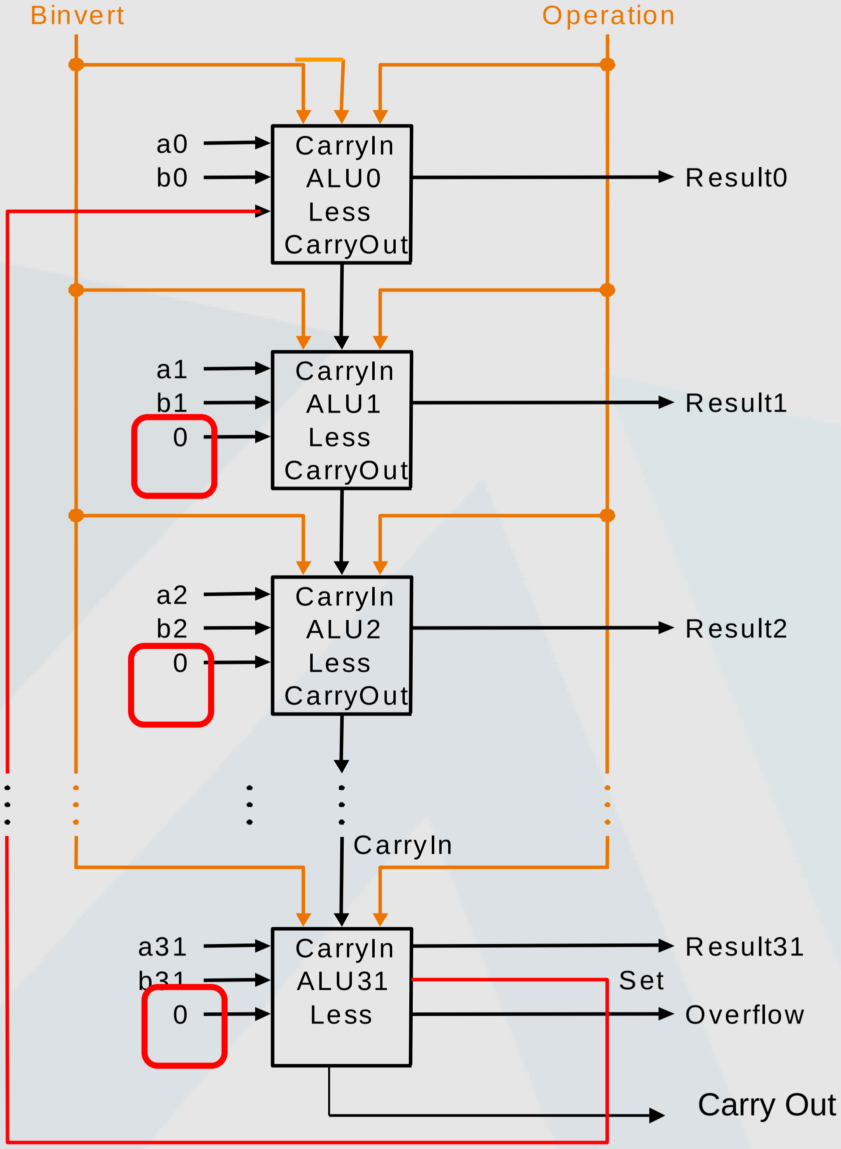
32-bit ALU With 5 Functions

1-bit ALU for non-most significant bits



1-bit ALU for the most significant bits





(Subtraction)

Operation = 3 and Binvert =1 for slt function

Overflow may affect the real value of slt

32-bit ALU beq/bne

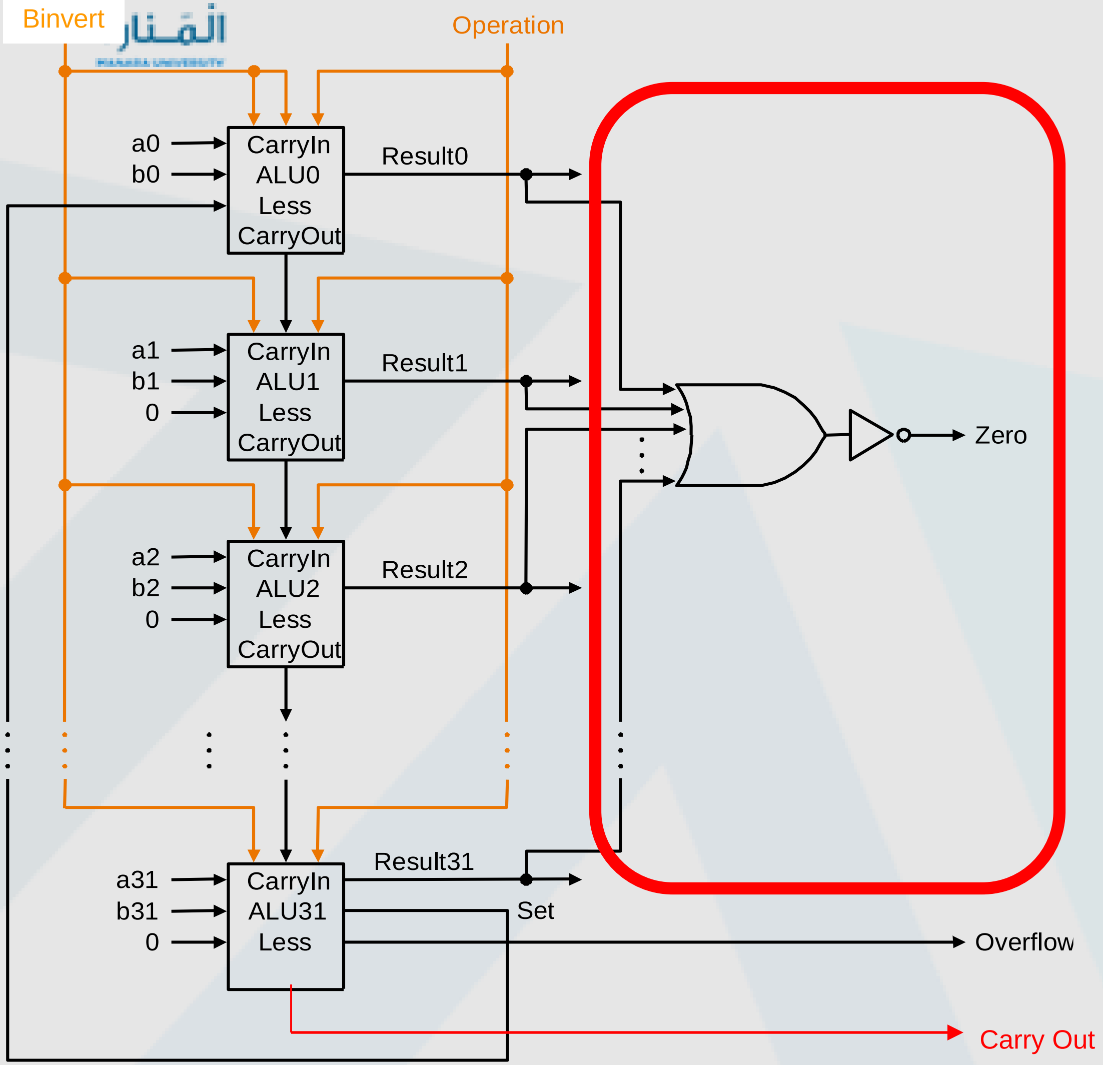
- The part related to ALU is a test of whether two values are equal or not.
- Branch may or may not be done according to the result of test.
- ALU is not the responsible part for achieving the branch.
- How can two values be **tested** for **equality**?
    - By Subtraction.

      (a-b=0)➔a=b
- The possibility in the ALU is to test whether the subtraction result is equal to zero.
- The test needs special circuits to achieve:

$$\mathbf{Zero}=\overline{(\boldsymbol{Result31}+\boldsymbol{Result30}+\cdots+\boldsymbol{Result2}+\boldsymbol{Result1}+\boldsymbol{Result0})}$$

32-bit ALU with 5 Functions and Zero

	Control lines	
Function	**Binvert** (1 line)	**Operation** (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10
slt	1	11



Design of an Arithmetic and logic Unit (ALU)

Steps:

1. Design of a 1-bit ALU that can achieve:
    1. AND
    2. OR
2. Building a 32-bit ALU.
3. Adding the ability of achieving summation (with carry).
4. Adding the ability of achieving some MIPS processor’s instructions:
    1. slt (set on less than)
    2. Conditional branch (beq, bne)
5. Adding the ability of achieving NOR.

32-bit ALU with 6 Functions

A nor B = $\overline{A}$ and $\overline{B}$

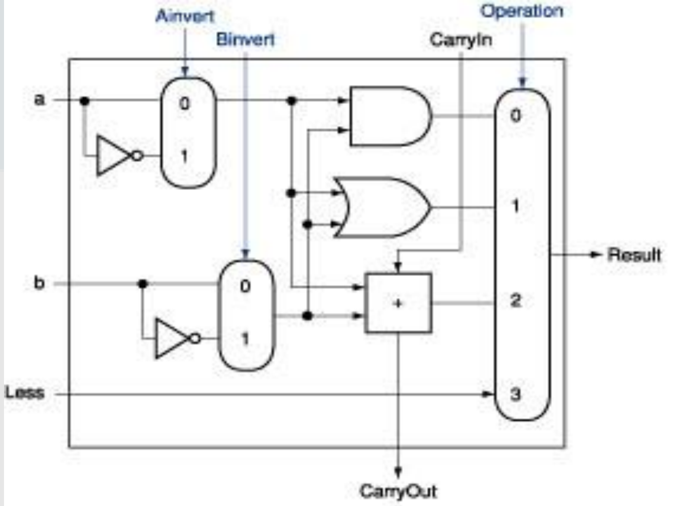


Figure B.5.10 (Top)

Function	Ainvert	Binvert	Operation
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
slt	0	1	11
nor	1	1	00

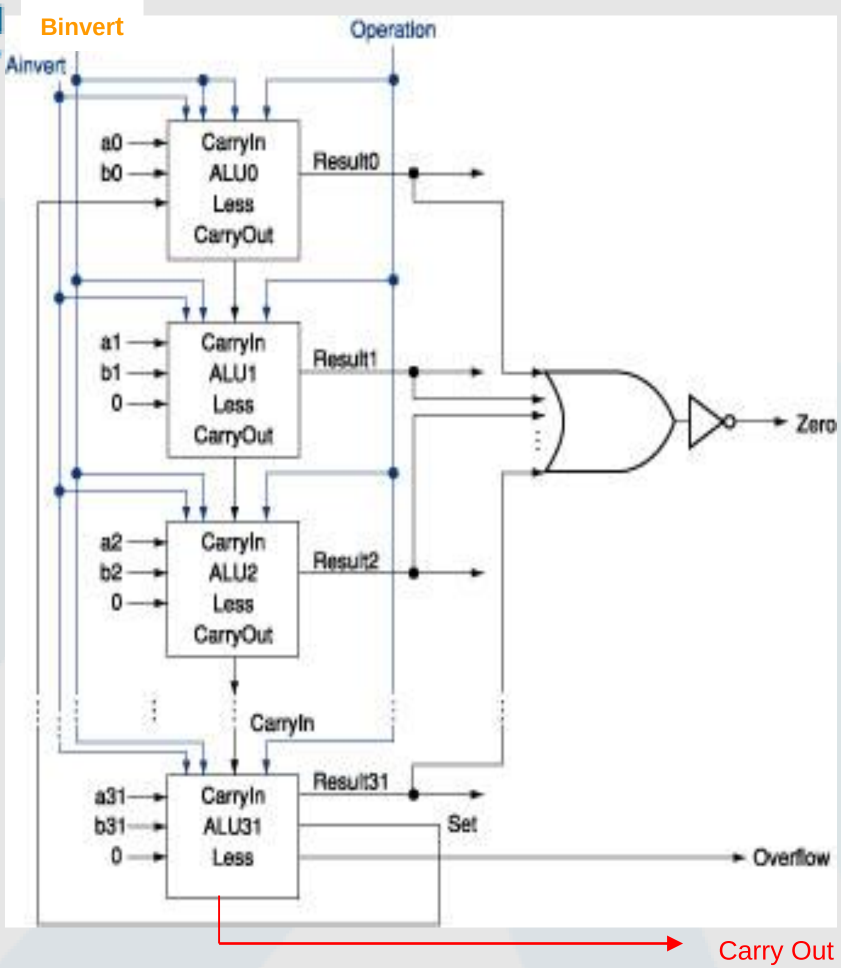


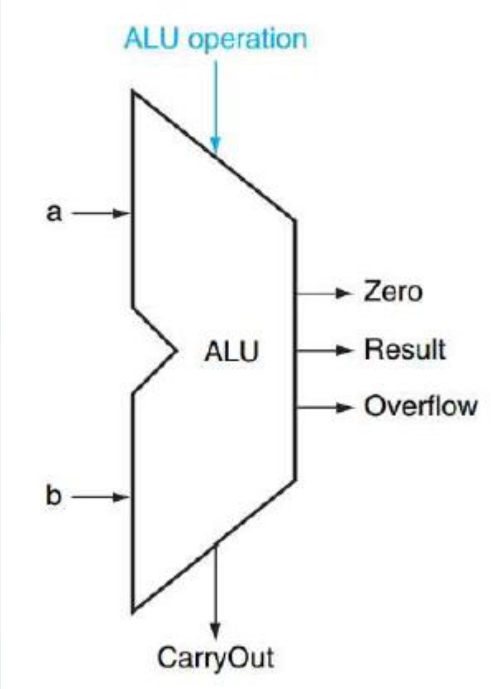
Figure B.5.12

Binvert+CarryIn=Bnegate

32-bit ALU with 6 Functions

- ALU control
  - **Ainvert** (1 bit)
  - **Binvert** (1 bit)
  - **Operation** (2 bits)

4-bit control lines for the ALU



Function	Ainvert	Binvert	Operation
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
slt	0	1	11
nor	1	1	00

ALU operation	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	Set on less than
1100	NOR

References

- David A. Patterson, John L. Hennessy, “Computer Organization and Design MIPS Edition: The Hardware/Software Interface”, Morgan Kaufmann, 2020.
- CSE 675.02: Introduction to Computer Architecture, Presentation F, Slides by Gojko Babić.