Week 2 – WINTER

Topics:

Simplification of Functions
k-maps

Simplification of Boolean function using Boolean Algebra laws

When a Boolean function is implemented with logic gates, each literal represents an input to a gate and each term is represented with a gate. Minimizing a function might result in a smaller circuit (hardware).

Useful properties: $a + a' = 1$ for 2 variables: $x'y' + x'y + xy' + xy = 1$

Simplification of Boolean functions using K-map (The Map Method)
A Karnaugh map is a modified form of Truth-table in which the arrangement of the combinations is particularly convenient. Each n-variable map consists of 2^n cells representing all possible combinations of
the variables.

Karnaugh Map Method

By using a valid block of 2 squares (minterms) you can eliminate 1 variable
By using a valid block of 4 squares (minterms) you can eliminate 2 variables
By using a valid block of 8 squares (minterms) you can eliminate 3 variables

Three-Variable Map

Four – Variable Map

Combinational Circuits Design

we will go over some examples

Integrated Circuits:
Multiplexers
Decoders

Combinational circuit consists of logical gates whose outputs at any time are determined directly from the present combination of inputs without regard to previous inputs.

Digital circuits are constructed with **integrated circuits**. IC is a small silicon semiconductor crystal, called *chip* containing the electronic components for the digital gates. The chip is mounted in a ceramic or plastic container and connections are welded to external pins.

The number of logic gates in a single package categorizes digital ICs.

Small-scale integration (SSI): number of gates is usually < 10.

Medium-scale integration (MSI): 10 < number gates < 100
Examples: **decoders, adders, multiplexers**

Large-scale integration (LSI) 100 < number of gates < few thousand gates

Very large-scale integration (VLSI): contains thousands of gates

However with integrated circuit, it is not the count of gates that determines the cost, but the number and types of ICs employed and the number of interconnections needed to implement the given digital circuit.

There are several combinational circuits that are employed extensively in the design of digital systems. These circuits are classified as **MSI components** and **PLD components**.

MSI examples: **adders, comparators, decoders, encoders, multiplexers.**
These components are also used as standard modules within more complex LSI and VLSI circuits.

A programmable logic device (**PLD**) is an integrated circuit with internal logic gates that are connected through electronic fuses. The gates in a PLD are divided into an AND array and an OR array that are connected to provide an AND-OR sum of product implementation. The advantage of using PLDs is that they can be programmed to incorporate complex logic functions within one LSI circuit.
Examples of PLDs are: **ROMs, PLAs**.

+ 0 1 1 0

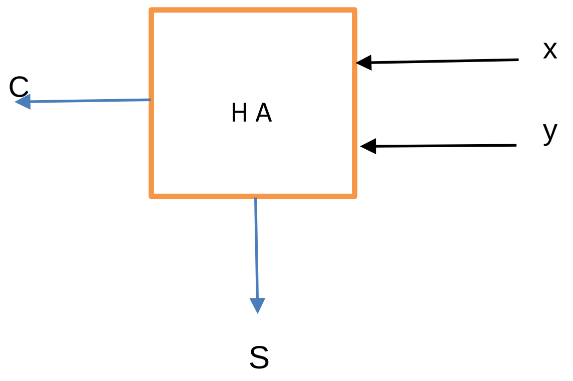
Adders

Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

C = xy
S = x'y + xy'

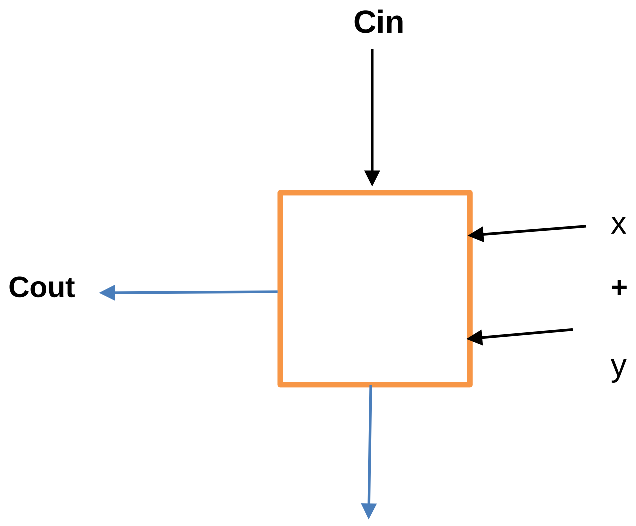
C = xy
S = x XOR y



Full Adder

1100 + 0110 →

```
  1 1 0 0
+ 0 1 1 0
```



S: sum (value of the least significant bit)

(COMPLETE) THE TRUTH TABLE

x	y	Cin	Cout	S
0	0	0		

USE K-MAPS TO SIMPLIFY THE OUTPUT

GIVE THE SIMPLIFIED EXPRESSION OF THE OUTPUT

S =

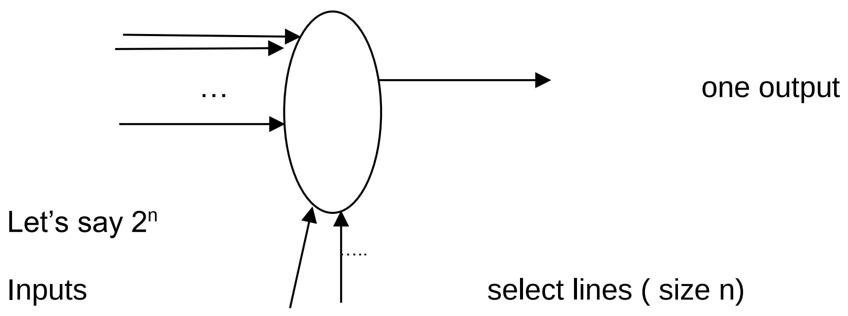
C =

Do you recognize the two know functions that S and C represent? Mention it in your submission.

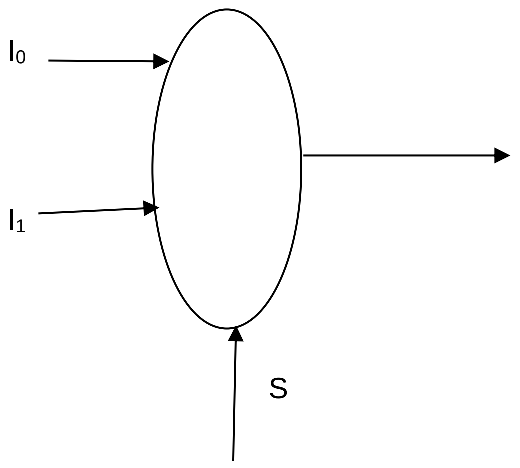
Multiplexers
A multiplexer connects multiple inputs to a single output.
It is a device intended to route data from one of several sources to a common destination; the source is determined by applying appropriate control (select) signals to the multiplexer.
Digital multiplexer selects binary information from one or many input lines and directs it to a single output line.

Multiplexers are used to control signals and data routing. For example, loading of the program counter. The value to be loaded into the program counter may come from one of several different sources.

The size of a multiplexer is specified by the number 2^n of its input lines and the single output line.



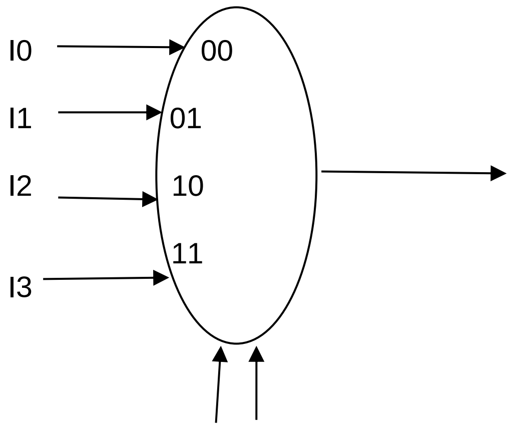
- 2 to 1 multiplexer representation and implementation
- n=1   2^1 x 1



O = S'I0 + SI1

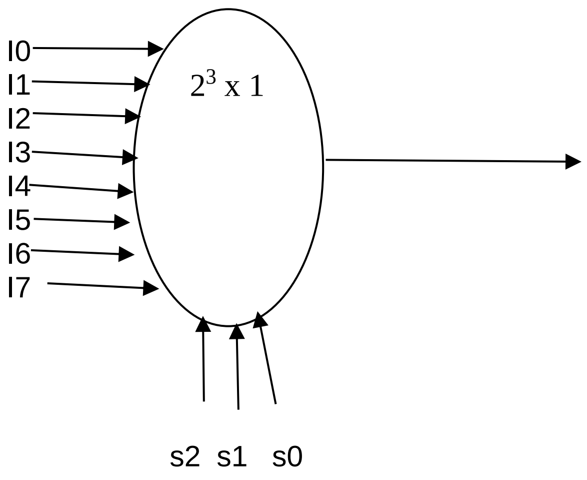
S = 0   O = I0
S = 1   O = I1

- 4 to 1 multiplexer representation and implementation



s1s0

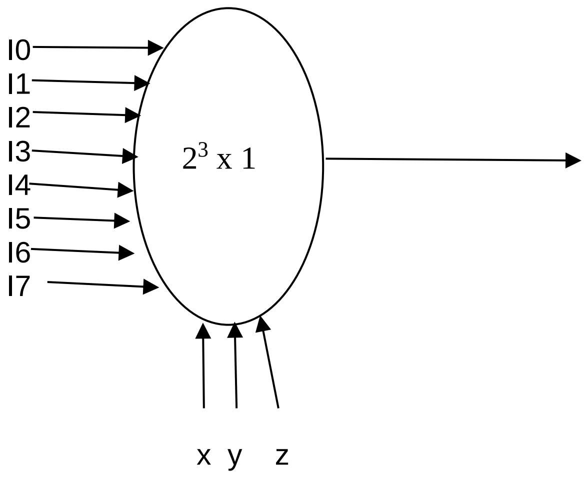
- 8 to 1 multiplexer representation and implementation



A multiplexer represents a functionally complete set.

Any Boolean function of n variables can be implemented by a 2^n to 1 multiplexer.
Any Boolean function of n variables can be implemented by a 2^{n-1} to 1 multiplexer.

For example:
Implement the **majority function** for three variables.   M(x,y,z) = ∑(3,5,6,7)



M(x,yz) = sum(3,5,6,7)

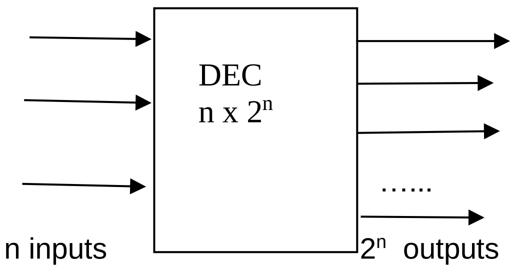
Decoders

Input size = n
Output size = at most 2^n (all possible input combinations), only one of which is asserted at any time.

Binary to Octal
Truth Table for a 3 to 8 decoder. $n \times 2^n$

Inputs			Outputs							
x	y	z	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

3 to 8 line Decoder: can be used for decoding 3-bit code to provide eight outputs, one for each element of code.
A decoder with an enable input is referred as a *decoder/demultiplexer*. Decoder/demultiplexer circuits can be connected together to form a larger decoder circuit.
Demultiplexer is a circuit that receives info on a single line and transmits this information on one of the 2^n lines.



2^n outputs (corresponding to all minterms can be created with the n inputs)

Encoders

perform inverse operation of decoders.

Octal to Binary Encoder

Inputs								Outputs		
D0	D1	D2	D3	D4	D5	D6	D7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

The encoder is implemented with 3 OR gates.
Limitation: if two inputs are active simultaneously, the output is undefined.
Ambiguity: an output of 0 is generated when input is D0 but also when all inputs are 0.
X = D4 + D5 + D6 + D7

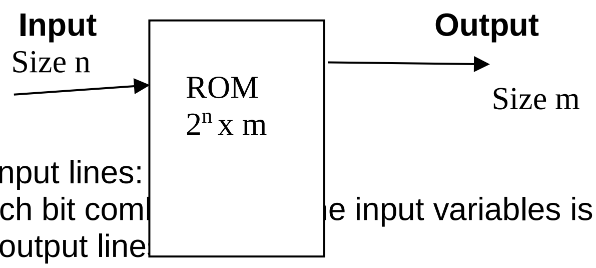
Very Large Size Integrated Circuits (VLSI)
ROM
PLA

Read-Only Memory (ROM)
ROM: device that includes a decoder and OR gates within a single IC package.
- used for permanent storage of binary information.

ROMs come with special internal electronic fuses that can be programmed for a specific configuration (storing data into the memory)

There is a difference between **storing permanent information** into the memory and **reading information from the ROM**.

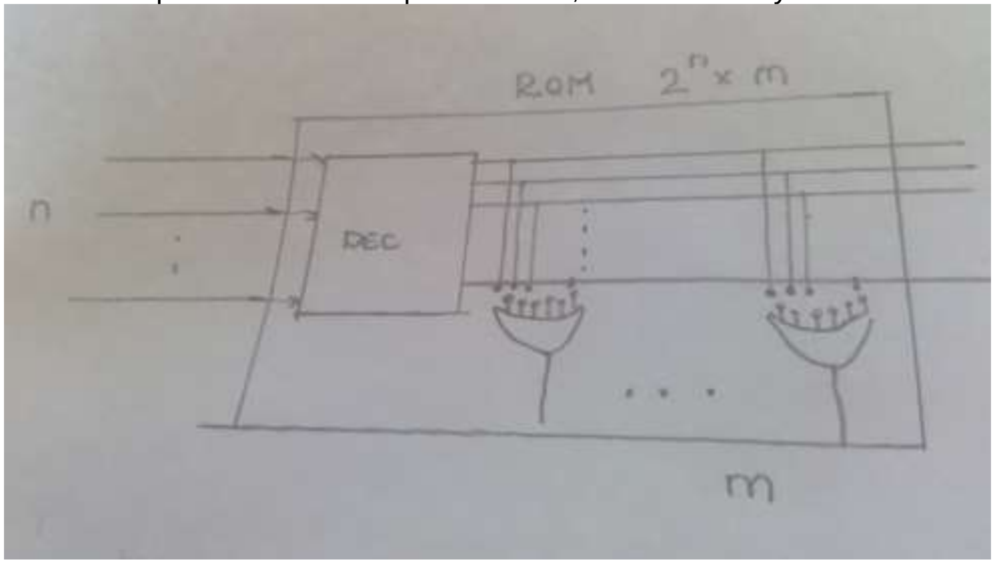
Reading data
ROM block diagram:



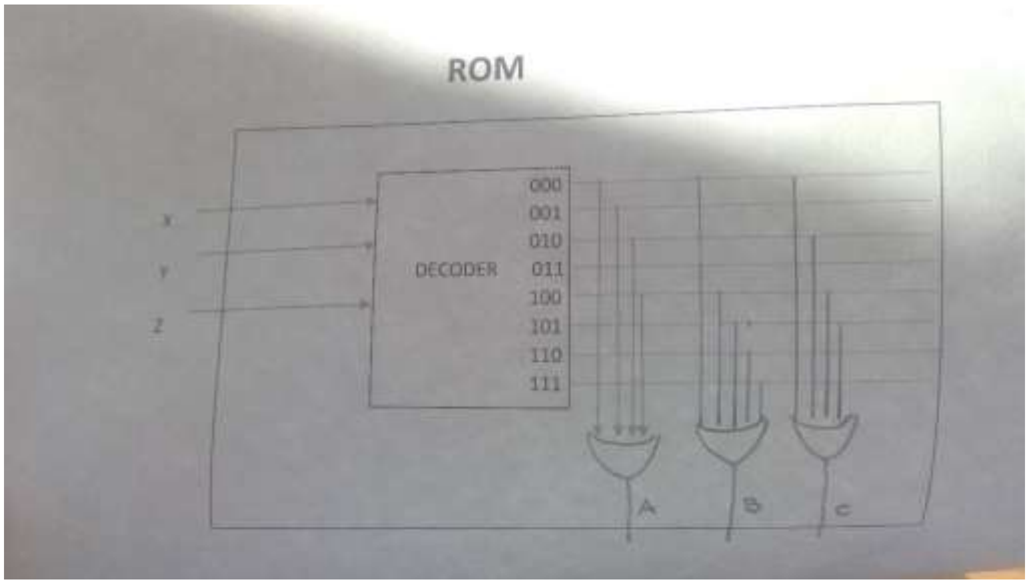
n input lines:
each bit com e input variables is called *address*
m output line

You go in with an address in order to retrieve (read) the word at that address.

Storing information
ROM is a functionally complete set. It can implement any switching function. In fact ROM is a VLSI (Very-large-scale integration) combinational circuit.
The main components of a ROM (size 2^n x m) are an n x 2^n Decoder and m OR gates.
Blowing of the fuses is referred as programming the ROM.
Note: for implementation of a specific circuit, all is necessary is the **truth table of the circuit**.



A(x,y,z) = x’z’ + xy’z’+x’y’z
B(x,y,z) = x’y’z’ + xy’ + xy
C(x,y,z) = ∑(0,2,4,5)



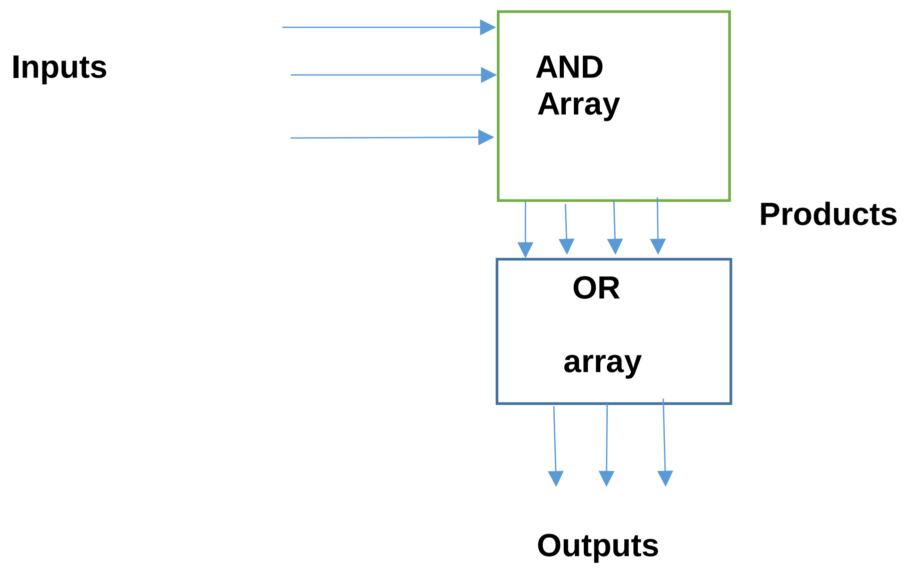
Programmable Logic Array (PLA)

Not all the bit patterns available in the ROM are used; this may be considered a waste of available equipment.

For cases where the number of don't care conditions is excessive, it is more economical to use a second type of (V)LSI component named *programmable logic array*.

PLA consists of regular arrangement of NOT, AND, OR gates on a chip.

- Each chip input is passed through a NOT gate so that each input and its complement are available to each AND gate.
- The output of each AND gate is available to each OR and the output of the OR gate is a chip output.



For designing combinational circuit using PLA all information is needed is the
PLA program table:

1st column:	product term
2nd column:	input: path from input to AND gates block
3rd column :	output: path from AND block & OR gates

We must minimize the number of AND gates (product terms; number of literals inside the product doesn't matter). The simplified form of the functions complements are analyzed as well in order to choose the smallest set of product terms.

A typical commercial PLA would have over 10 inputs and about 50 product terms.

Using PLA, implement the 3 functions (A, B, C)

Using K-Maps simplify the functions
PLA Table
Draw the PLA circuit that will implement A, B, C

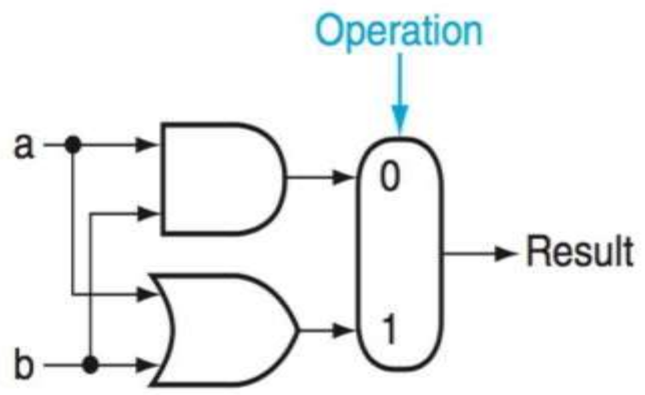
Combinational circuits

- **1-bit ALU**
- **32-bit ALU**

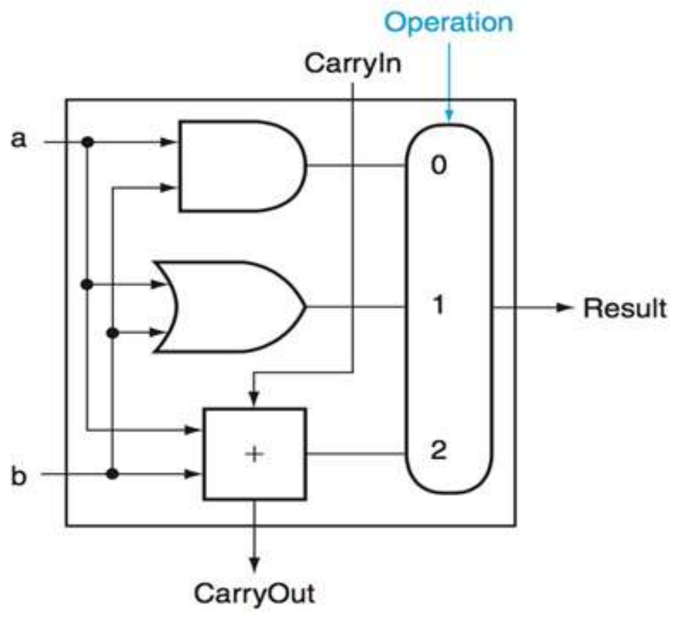
READING: APPENDIX B (POSTED ON COURSE'S SITE) B5-26 TO B5-36

Addition (and many other operations that computer performs) are bitwise operations.

<u>1-bit ALU</u> (AND, OR)



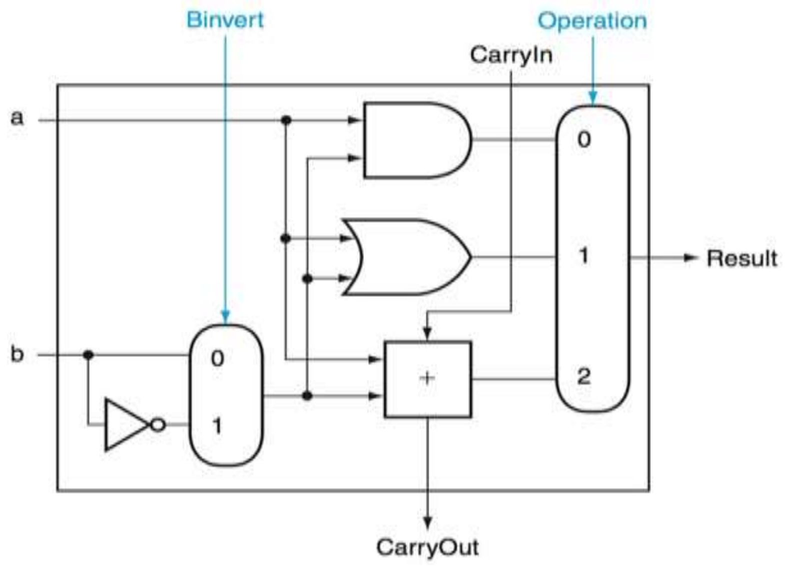
<u>1-bit ALU</u> (AND, OR, Addition)



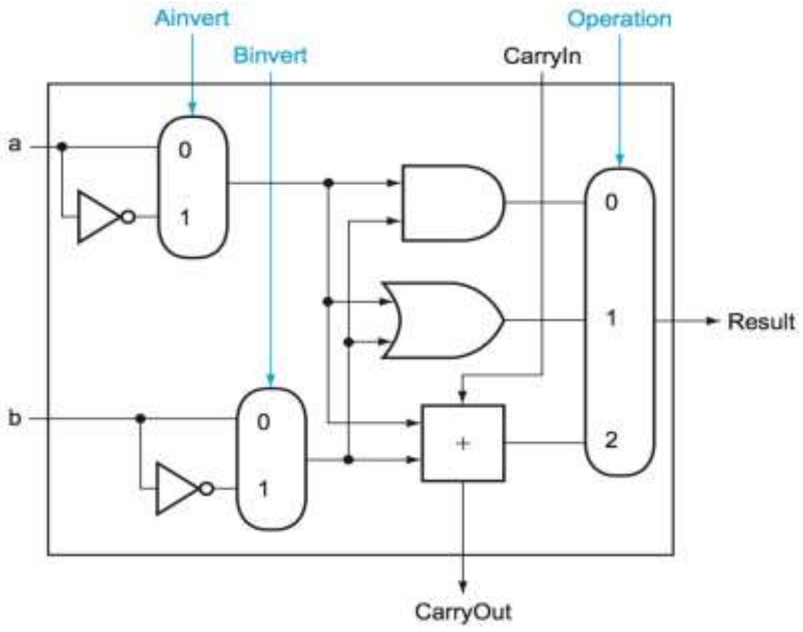
<u>1-bit ALU</u> (AND, OR, addition, subtraction)

$$a + b = a + b + 0$$

Subtraction is a special addition: a – b = a + (b' + 1)



A 1-bit ALU that performs AND, OR, and addition on a and b or not a and not b

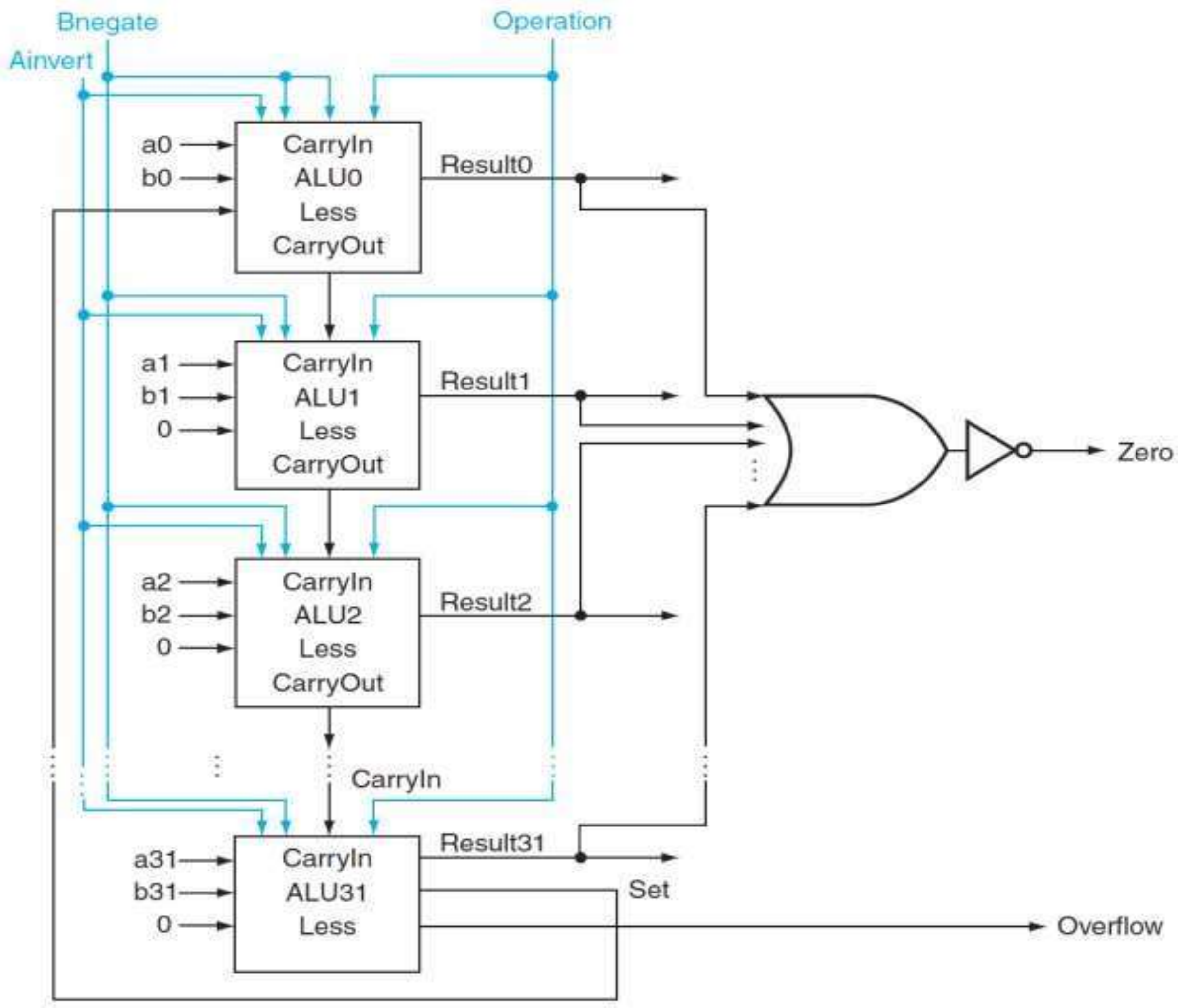


a nand b = (a and b)' = a' + b'

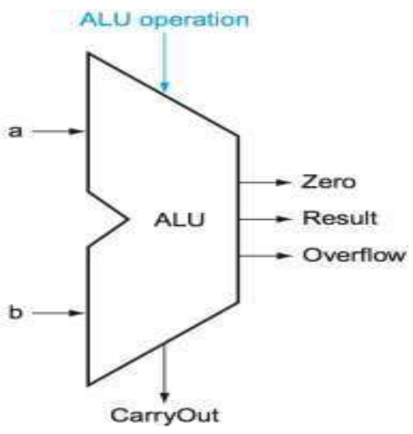
a nor b = (a or b)' = a'*b'

32-bit ALU constructed from 32 1-bit ALUs

Implementation of slt
Implementation of check for zero

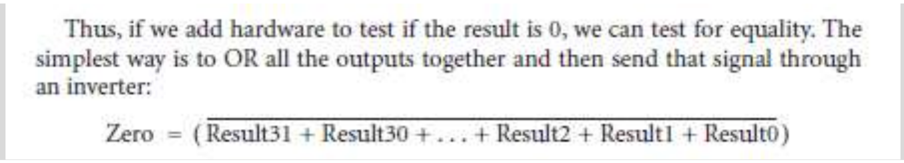


The symbol commonly used to represent an ALU
This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.



CHECK FOR ZERO IMPLEMENTATION

A-B = zero → check for zero bit will be set to 1.



Thus, if we add hardware to test if the result is 0, we can test for equality. The simplest way is to OR all the outputs together and then send that signal through an inverter:

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0})}$$

SLT IMPLEMENTATION

slt: set on less than	slt \$t1, \$t2, \$t3	\$t1 ←1 if \$t2 < \$t3 0 otherwise

MIPS assembly language - Introduction

Go to **http://spimsimulator.sourceforge.net/**

Learn about Getting started with PCSpim by reading
http://www.cs.wisc.edu/~larus/PCSpim.pdf

It is recommended to also download a free text editor specifically designed for MIPS (mars for MIPS 32-bit architecture).

You don't need to have them both installed. One is enough. Mars allows some sugar code.
http://courses.missouristate.edu/kenvollmar/mars/ you can download Mars 4.5

SPIM is more rigid and exact.

In order to write your program, you can use a regular text editor such as Notepad or Wordpad.

Assembly language is the symbolic representation of a computer's binary encoding—**machine language**
Assembly language is more readable than machine language because it uses symbols instead of bits.

A tool called an assembler translates assembly language into binary instructions. Assemblers provide a friendlier representation than a computer's 0s and 1s that simplifies writing and reading programs. An assembler reads a single assembly language *source file* and produces an *object file* containing machine instructions and bookkeeping information that helps combine several object files into a program.

Assembly language is a programming language. Its principal difference from high-level languages such as BASIC, Java, and C is that assembly language provides only a few, simple types of data and control flow. Assembly language programs do not specify the type of value held in a variable. Instead, a programmer must apply the appropriate operations (e.g., integer or floating-point addition) to a value. In addition, in assembly language, programs must implement all control flow with *goto*s. Both factors make assembly language programming for any machine—MIPS or 80x86—more difficult and error-prone than writing in a high-level language.

The primary reason to program in assembly language, as opposed to an available high-level language, is when the speed or size of a program is critically important. For example, consider a computer that controls a piece of machinery, such as a car's brakes.

SYNTAX FOR MIPS ASSEMBLY LANGUAGE INSTRUCTIONS

[label:] Op-Code [operand], [operand], [operand] [#comment]

Label field is optional. Anything following the # sign is a comment.

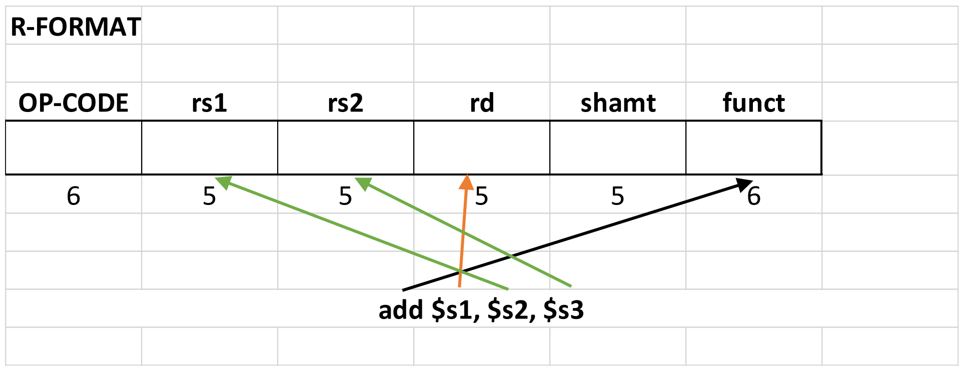
MIPS ASSEMBLY LANGUAGE

Instructions are divided into three (binary) formats (R, I, and J format)

R FORMAT
Register Format (binary) Instruction

Op-code	rs1(rs)	rs2(rt)	rd	shamt	funct
6bits	5bits	5bits	5bits	5bits	6bits

Common arithmetic and logic instructions (R format)
Three operands: result + two sources



add	add $s1, $s2, $s3	$s1 = $s2 + $s3
subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3
and	and $s1, $s2, $s3	$s1 ←$s2 and $s3
shift left logical	sll $t0, $t1, 3	$t0 ←$t1 shifted 3 bits to left ($t1 | 000) if $t1 contains a number it's the same as multiplying that number by 8 (2^3)

Addition in assembly
Example: add $s0, $s1, $s2 (in MIPS)
Equivalent to a = b + c (in C)
where MIPS registers $s0, $s1, $s2 are associated with C variables a, b, c

Subtraction in assembly
Example: sub $s3, $s4, $s5 (in MIPS)
Equivalent to d = e - f (in C)
where MIPS registers $s3, $s4, $s5 are associated with C variables d, e, f

Multiplication
In C: a = b * c

In MIPS: b → $s2, c → $s3, and a → $s0 and $s1
```
mult $s2, $s3                # b*c
mfhi $s0                     # upper half of product into $s0; move from hi
mflo $s1                     # lower half of product into $s1; move from low
```

Often, we only care about the lower half of the result
NOTE: if the result doesn't need two registers, you will be allowed to use

mul $t2, $t1, $t3

Division (signed)
```
 div register 1, register 2          #Divides 32 bits register 1 by 32 bits register 2
```
Put reminder of division in hi, quotient in lo

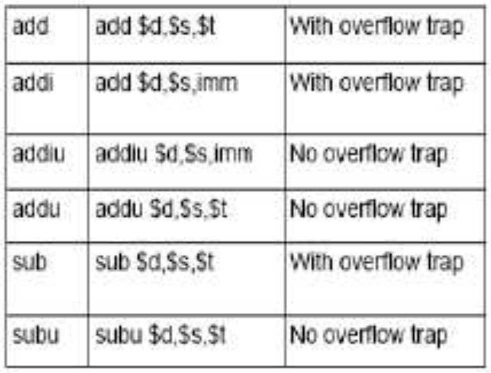
Example in C: a = c/d; b = c % d;
In MIPS: a -> s0, b -> s1, c -> s2, d -> s3
```
div $s2, $s3          # lo = c/d, hi = c%d
mflo $s0              # Get quotient
mfhi $s1              # Get remainder
```

NOTE: we will probably NOT use div in our programs.

Overflow in Arithmetic
Some languages detect overflow (Ada), some don't (C).

MIPS solution:
Add (add), add immediate (addi) and subtract (sub) cause overflow to be detected.

Add unsigned (addu), add immediate unsigned (addiu) and subtract unsigned (subu) do not cause overflow.



add	add $d,$s,$t	With overflow trap
addi	add $d,$s,imm	With overflow trap
addiu	addiu $d,$s,imm	No overflow trap
addu	addu $d,$s,$t	No overflow trap
sub	sub $d,$s,$t	With overflow trap
subu	subu $d,$s,$t	No overflow trap

Compiler selects appropriate arithmetic.

I FORMAT

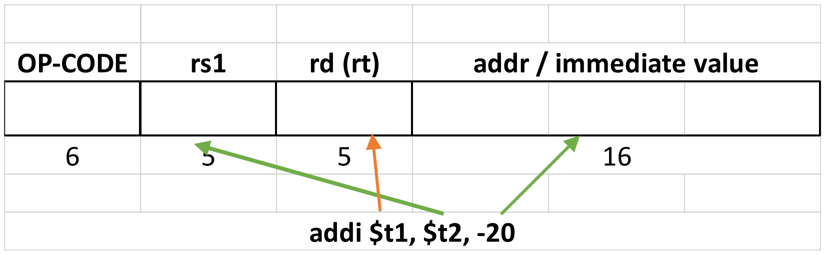
I(mmediate) Format (binary) Instruction

Op-code	rs(rs1)	rt(rs2)	constant or address
6 bits	5 bits	5 bits	16 bits

OP-CODE	rs1	rs2 (rt)	addr / immediate value
6	5	5	16

Immediate arithmetic / logical instruction

addi \$t1, \$t2, -20 $\quad$ \$t1 ← \$t2 - 20



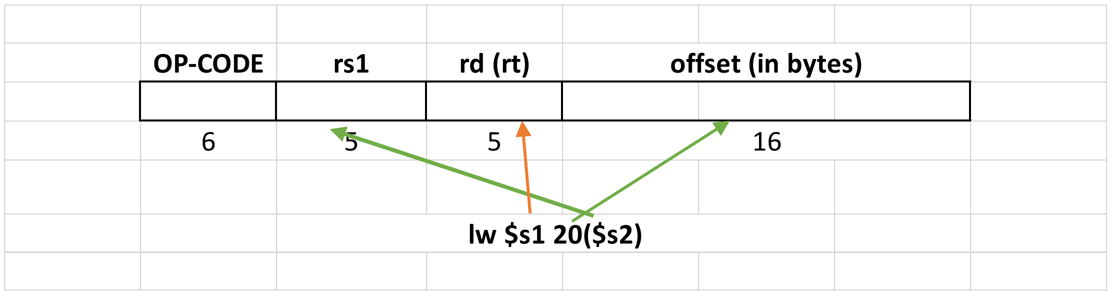
Data Transfer - Memory access

Load – transfer from memory to register
Store – transfer from register to memory

For the address we need the base register (address) and the offset (displacement)

Load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]
Store word	sw \$s1, 20(\$s2)	Memory[\$s2 + 20] = \$s1

\$s2: base register; 20: offset (number of bytes)
Target address: base address + offset



MIPS uses byte addressing; this means that the offset needs to be **expressed in bytes**; for both lw and sw, the sum of the base address and the offset must be a multiple of 4.
Alignment: objects must start on an address that is a multiple of 4.

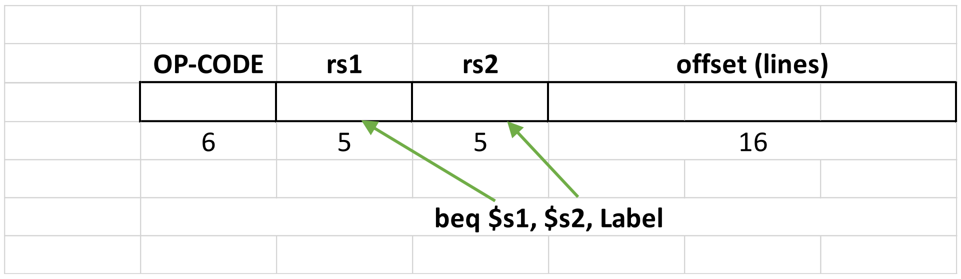
Conditional Branches

Branch on equal beq $s1, $s2, Label

Bne: branch not equal

Branches where the value of a register is compared against 0.
(less 0, <=0, >0, >=0)

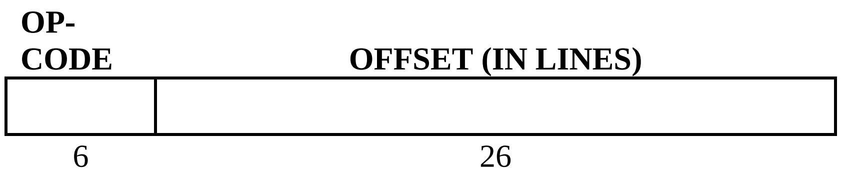
IT IS NOT ALLOWED TO USE: BLT OR BGT
USE ONLY BRANCHES POSTED ON THE INTEGER INSTRUCTION SET



Target address: address of the next instruction + offset (bytes)

J – type Instruction Format

op-code	address (offset in lines)
6 bits	26 bits



MIPS – Assembly Instruction

j done

Target Address : $[PC+4]^{31-28}$ | offset | 00