Computer Arithmetic

Major Goals

- To introduce **two's complement** numbers and their addition and subtraction.
- To introduce **basic logic operations** (AND, OR, logical shift) and **hardware building blocks** (AND, OR, NOT gates and multiplexor).
- To explain the construction of a 32-bit **arithmetic logic unit** (**ALU**) that performs AND, OR, `add`, and `slt`.
- To show algorithms that perform **multiplication** and **division** and hardware that implements these algorithms.
- To demonstrate **floating-point** representation and arithmetic operations.

Numbers

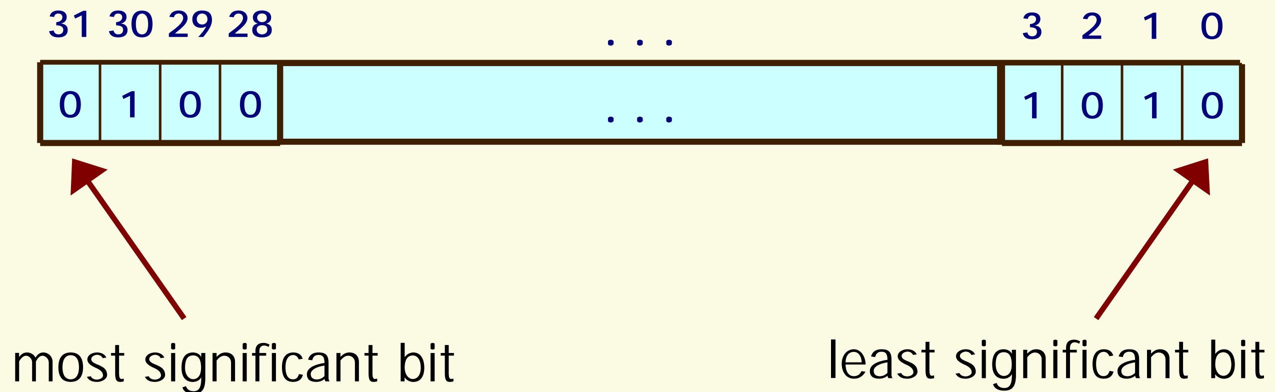
- **Bits** are the basis for **binary number representation** in digital computers.

- Conventions are needed to define relationships between **bit patterns** and **numbers**, e.g., conversion between non-negative (i.e., unsigned) **binary** and **decimal** numbers.

- However, things can get more complicated:
  - How to represent **negative numbers**
  - How to represent **fractions** and **real numbers**
  - How to handle numbers that go **beyond the representable range**

Words

- A MIPS **word** is a 32-bit pattern:



- A MIPS word can represent 2^{32} different 32-bit patterns.

Possible Representations

- Three possibilities:

Sign and Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues:
  - **Balance**
  - **Number of zeros**
  - **Ease of arithmetic operations**

- Which representation is the best? Why?

Two's Complement Representation

- **Bit 31** is called the **sign bit**:
  - **0** for **non-negative**
  - **1** for **negative**

- **Largest integer** represented by a MIPS word:
  $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$

- **Smallest integer** represented by a MIPS word:
  $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$

- All computers today use the two's complement representation for representing signed numbers.

Decimal to Binary Conversion

- Let -N be a negative decimal number.

- **Steps**:
  1. Convert N to 32-bit binary representation (sign bit is 0).
  2. Invert all 32 bits (i.e., 1 to 0 and 0 to 1).
  3. Add 1 to the inverted representation to obtain the two's complement representation of -N.

- **Property used**:
  - The sum of a binary number (x) and its inverted representation ($\overline{x}$) is $111...111_{two}$, which represents -1.

$$x + \overline{x} = -1 \quad \Rightarrow \quad x + \overline{x} + 1 = 0 \quad \Rightarrow \quad \overline{x} + 1 = -x$$

Binary to Decimal Conversion

- **Method 1**:
  - Let b_i denote the *i*th bit of a negative number in two's complement representation.
  - Decimal representation:

$$b_{31} \times (-2^{31}) + b_{30} \times 2^{30} + b_{29} \times 2^{29} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$$

- **Method 2**:
  - **Steps**:
    1. Invert all 32 bits.
    2. Add 1 to the inverted pattern.
    3. Compute decimal representation (let b_i denote the *i*th bit of the inverted number; b_{31} must be 0):

$$-(b_{30} \times 2^{30} + b_{29} \times 2^{29} + \ldots + b_1 \times 2^1 + b_0 \times 2^0)$$

Signed and Unsigned Numbers

- **Signed numbers** refer to integers that can be **negative or non-negative** (cf. `int` in C/C++).
- **Unsigned numbers** refer to **non-negative** integers (cf. `unsigned int` in C/C++).
- **Operations for unsigned numbers**:
  - Comparison: `sltu`, `sltiu`
  - Arithmetic: `addu`, `addiu`, `subu`
  - Load: `lbu`

Sign Extension

- The instruction `lb` copies a byte as a signed number into the 8 rightmost bits of a register and copies the sign bit repeatedly to fill the rest of the register.

- **Conversion of 16-bit binary signed numbers into 32-bit numbers** (in general n bits to >n bits) is done by filling the leftmost bits with the sign bit.

- **Examples**:
  - 2 (16 bits -> 32 bits):

  0000 0000 0000 0010 -> 0000 0000 0000 0000 0000 0000 0000 0010

  - -2 (16 bits -> 32 bits):

  1111 1111 1111 1110 -> 1111 1111 1111 1111 1111 1111 1111 1110

Hexadecimal Numbers

- To avoid reading and writing long binary numbers, a higher base than binary that can be converted easily into binary is desirable.

- Since almost all computer data sizes are multiples of 4, **hexadecimal (base 16)** numbers are commonly used. Since base 16 is a power of 2, we can simply convert by replacing **each group of four bits** by a single hexadecimal digit, and vice versa.

- **Hexadecimal-to-binary conversion**:
  - 0_{hex} - 9_{hex} for 0000_{two} - 1001_{two}
  - a_{hex} - f_{hex} for 1010_{two} - 1111_{two}

Addition and Subtraction

- **Addition**:
  - Digits are added bit by bit **from right to left**, with **carries** passed to the next digit to the left.

- **Subtraction**:
  - **Subtraction uses addition.**
  - The appropriate operand is negated before being added to the other operand.

- **Overflow**:
  - The result is too large to fit into a word.

Examples

- Addition (7 + 6 = 13):

```
        0000 0000 0000 0000 0000 0000 0000 0111two  =   7ten
+       0000 0000 0000 0000 0000 0000 0000 0110two  =   6ten
        -------------------------------------------------------
=       0000 0000 0000 0000 0000 0000 0000 1101two  =  13ten
```

- Subtraction (7 - 6 = 1):

```
        0000 0000 0000 0000 0000 0000 0000 0111two  =   7ten
+       1111 1111 1111 1111 1111 1111 1111 1010two  =  -6ten
        -------------------------------------------------------
=       0000 0000 0000 0000 0000 0000 0000 0001two  =   1ten
```

Detecting Overflow

Addition (X + Y)

- No overflow occurs when:
  - X and Y are of different signs.
- Overflow occurs when:
  - X and Y are of the same sign, but X + Y is represented in a different sign.

Subtraction (X - Y)

- No overflow occurs when:
  - X and Y are of the same sign.
- Overflow occurs when:
  - X and Y are of different signs, but X - Y is represented in a different sign from X.

Overflow Conditions

Operation	Sign Bit of X	Sign Bit of Y	Sign Bit of Result
X + Y	0	0	1
X + Y	1	1	0
X – Y	0	1	1
X – Y	1	0	0

Effects of Overflow

- When an overflow **exception** (or called **interrupt**) occurs:
  - Control jumps to a **predefined address** for **handling the exception**.
  - The interrupted address is saved for **possible resumption**.

- Details of exception handling depend on software system and language:
  - E.g., flight control system vs. homework assignment

- Some instructions are designed in the way that they do not cause exceptions on overflow (for efficiency), requiring the programmer to be responsible for using them correctly:
  - E.g., **`addu`**, **`addiu`**, **`subu`**

Logical Operations

- Some programs need to operate on fields of bits or even individual bits within a word.

- One class of such operations is called **shifts**. They move all the bits in a word to the left or to the right by a **specified number of bits**, filling the emptied bits with 0s.

- **Example**:
  - Original value:

    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two}$

  - Shifted value (shift left by 8; equivalent to multiplying it by 256):

    $0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{two}$

Shift Instructions

- `sll` ('shift left logical') and `srl` ('shift right logical'):

- Example:

```
sll  $t2, $s0, 8      # $t2 gets $s0 << 8 bits
```

op	rs	rt	rd	shamt	funct
0	0	16	10	8	0

  - The encoding of `sll` is `0` in both the **op** and **funct** fields.
  - **rd** contains `$t2`; **rt** contains `$s0`; **shamt** contains `8`.
  - **rs** is not used, and thus is set to `0`.

Logical AND Operation

- It is a **bit-by-bit operation** that leaves a 1 in the result only if both bits of the two operands are 1. A bit pattern, called **mask**, can be used in conjunction with AND to force 0s where there is a 0 in the mask.

- **Example**:

```
and  $t0, $t1, $t2    # $t0 gets $t1 AND $t2
```

  - **Source operands**:

    Content of **$t1**: 0000 0000 0000 0000 0011 1100 0000 0000_{two}

    Content of **$t2**: 0000 0000 0000 0000 0000 1101 0000 0000_{two}

  - **Destination operand**:

    Content of **$t0**: 0000 0000 0000 0000 0000 1100 0000 0000_{two}

- **andi** ('and immediate'): one operand is a constant

Logical OR Operation

- Similar to AND, it is a **bit-by-bit operation** that leaves a 1 in the result if either operand bit is a 1.

- **Example**:

```
or   $t0, $t1, $t2        # $t0 gets $t1 OR $t2
```

  - **Source operands**:

    Content of **`$t1`**: 0000 0000 0000 0000 0011 1100 0000 0000_{two}

    Content of **`$t2`**: 0000 0000 0000 0000 0000 1101 0000 0000_{two}

  - **Destination operand**:

    Content of **`$t0`**: 0000 0000 0000 0000 0011 1101 0000 0000_{two}

- **`ori`** ('or immediate'): one operand is a constant

Constructing an Arithmetic Logic Unit

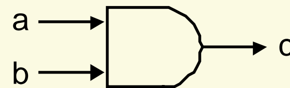
- The **arithmetic logic unit** (**ALU**) of a computer is the hardware component that performs:
  - **Arithmetic operations** (like addition and subtraction)
  - **Logical operations** (likes AND and OR)

Processor

Hardware Building Blocks for ALU

- AND gate:

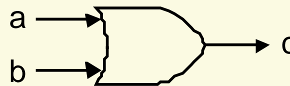
  $c = a \cdot b$



a	b	c = a . b
0	0	0
0	1	0
1	0	0
1	1	1

- OR gate:

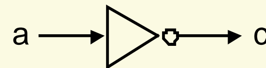
  $c = a + b$



a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

- Inverter:

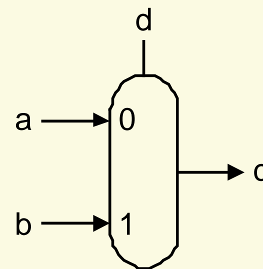
  $c = \overline{a}$



a	$c = \overline{a}$
0	1
1	0

- Multiplexor:

  $$c = \begin{cases} a & \text{if } d = 0 \\ b & \text{if } d = 1 \end{cases}$$



d	c
0	a
1	b

Some Basics of Logic Design

- **Goal**: To give a brief overview of logic design so that we can design the components of a computer (i.e., processor, memory, etc.).
- Reference: Appendix B (B.2-B.3) of textbook
- *For those who are already familiar with logic design, this is an opportunity for you to refresh your memory.*

Digital Logic Circuits

- The electronics inside modern computers are **digital**. They operate with only two voltage levels of interest - hence the use of **binary** numbers.

- Two types of logic circuits:
  - **Combinational logic circuits**:
    - Logic circuits that do not have memory.
    - The output depends only on the current input.
  - **Sequential logic circuits**:
    - Logic circuits that have memory.
    - The output depends on both the current input and the value stored in memory (called **state**).

- *We will focus on combinational circuits in this topic. Sequential logic circuits will be needed later.*

Truth Tables

- Since a combinational logic circuit contains **no memory**, it can be **completely specified** by defining the values of the outputs for each possible set of input values. Such a description is called a **truth table**.
- For a logic circuit with N inputs, there are **2^N entries** in the truth table.
- Truth tables can completely describe any combinational logic function. However, truth tables grow in size quickly (**exponential to N**) and may not be easy to understand.

Example

- Consider a logic function with three inputs (A, B, C) and three outputs (D, E, F). Give the truth table that corresponds to the function with the following properties: D is true if at least one input is true; E is true if exactly two inputs are true; F is true only if all three inputs are true.

- **Answer**:

Inputs			Outputs		
A	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Boolean Algebra

- Truth tables have **scale-up problem**. Another approach is to express the logic function using **Boolean algebra**.

- All variables have the values 0 or 1.

- **Three basic operators**:
  - **AND** operator: denoted as ' · ' as in $A \cdot B$
  - **OR** operator: denoted as '+' as in $A + B$
  - **NOT** operator: denoted as ' ¯ ' as in $\overline{A}$

- Any logic function can be implemented using only AND, OR, and NOT operations. AND, OR, and NOT are said to form a **complete set**.

Logic Equations

- Any set of logic functions can be written as a series of **logic equations**, each of which has an output on the left-hand side and a formula consisting of variables and the three operators (AND, OR, NOT) on the right-hand side.

- **Examples**:

$$D = (A + \overline{B}) \cdot C$$

$$E = (\overline{A \cdot C}) + (\overline{B} \cdot C)$$

Several Laws in Boolean Algebra

- **Identity laws**:

  $A + 0 = A \qquad A \cdot 1 = A$

- **Zero and one laws**:

  $A + 1 = 1 \qquad A \cdot 0 = 0$

- **Inverse laws**:

  $A + \overline{A} = 1 \qquad A \cdot \overline{A} = 0$

- **Commutative laws**:

  $A + B = B + A \qquad A \cdot B = B \cdot A$

- **Associative laws**:

  $A + (B + C) = (A + B) + C \qquad A \cdot (B \cdot C) = (A \cdot B) \cdot C$

- **Distributive laws**:

  $A \cdot (B + C) = (A \cdot B) + (A \cdot C) \qquad A + (B \cdot C) = (A + B) \cdot (A + C)$

Example

- Consider a logic function with three inputs (A, B, C) and three outputs (D, E, F). Show the logic equations for the logic function that has the following properties: D is true if at least one input is true; E is true if exactly two inputs are true; F is true only if all three inputs are true.

- **Answer**:

$$D = A + B + C$$

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot (\overline{A \cdot B \cdot C})$$

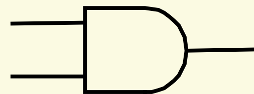
$$F = A \cdot B \cdot C$$

Gates

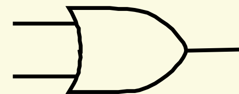
- **Gates** are basic building blocks for constructing **logic circuits**.

- **Three basic gates** that correspond to the three basic logic operations:
  - **AND** gate
  - **OR** gate
  - **NOT** gate

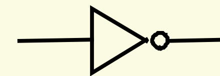
- Standard representation:



AND



OR



NOT

Combinational Logic

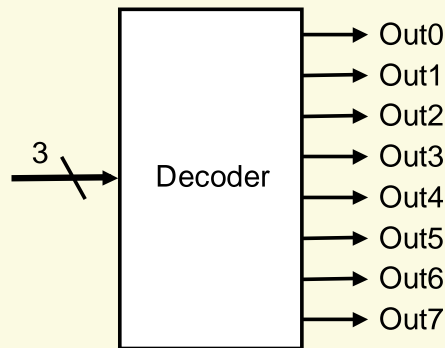
- Other than logic gates that are the most basic building blocks, there also exist some **higher-level basic building blocks** that are also commonly used:
  - **Decoders/encoders**
  - **Multiplexors**
  - **Two-level logic** and **PLAs**

- These building blocks can be implemented using AND, OR, and NOT gates.

Decoders

- A **decoder** (**N-to-2^N decoder**) is a logical block with an N-bit input and 2^N 1-bit outputs. The output that corresponds to the input bit pattern is true while all other outputs are false.

- **Example** (3-to-8 decoder):



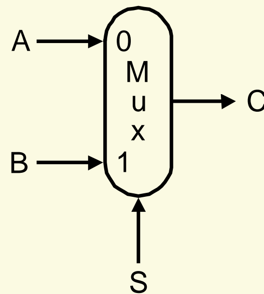
- An **encoder** performs the inverse function of a decoder, taking 2^N inputs and producing an N-bit output.

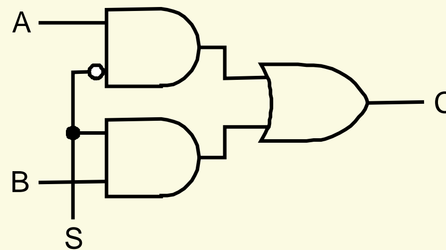
Multiplexors

- A **multiplexor** (or **selector**) selects one of the data inputs as output by a control input value.

- A multiplexor can have an arbitrary number of data inputs:
  - Two data inputs require one selector input.
  - N data inputs require $\lceil \log_2 N \rceil$ selector inputs.

- **Example** (2-input multiplexor):





$$C = (A \cdot \overline{S}) + (B \cdot S)$$

Two-Level Logic

- Any logic function can be expressed in a canonical form as a **two-level representation**:
  - Every input is either a variable or its negated form.
  - One level consists of **AND** gates only.
  - The other level consists of **OR** gates only.

- **Sum-of-products representation**:
  - E.g., $E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$
  - More commonly used than product-of-sums representation.

- **Product-of-sums representation**:
  - E.g., $E = (\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{C} + B) \cdot (\overline{B} + \overline{C} + A)$

Example

- Show the sum-of-products representation for the following truth table:

Inputs			Output
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- **Answer**:

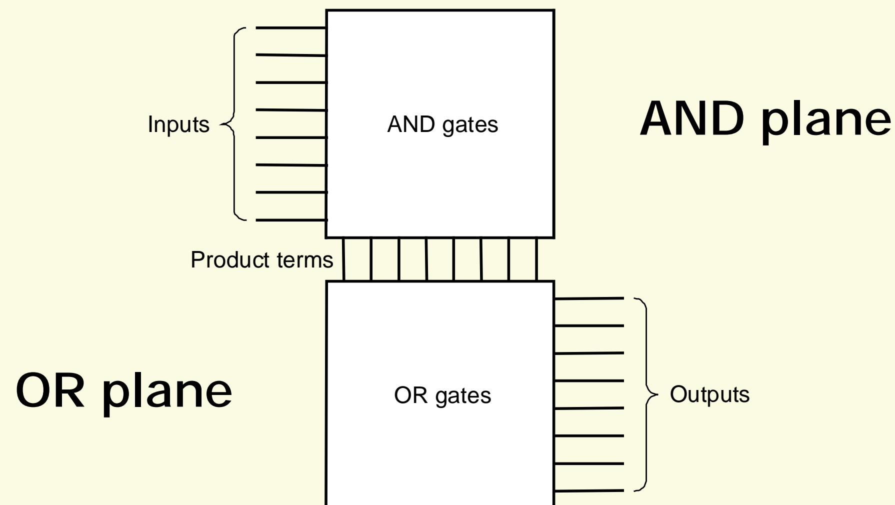
$$D = (\overline{A} \cdot \overline{B} \cdot C) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot B \cdot C)$$

  - Only those table entries for which the output is 1 generate corresponding terms in the equation.

Programmable Logic Arrays

- A **programmable logic array** (**PLA**) is a gate-level implementation of the **two-level representation** for any set of logic functions, which corresponds to a truth table with multiple output columns.

- A PLA corresponds to the **sum-of-products representation**.



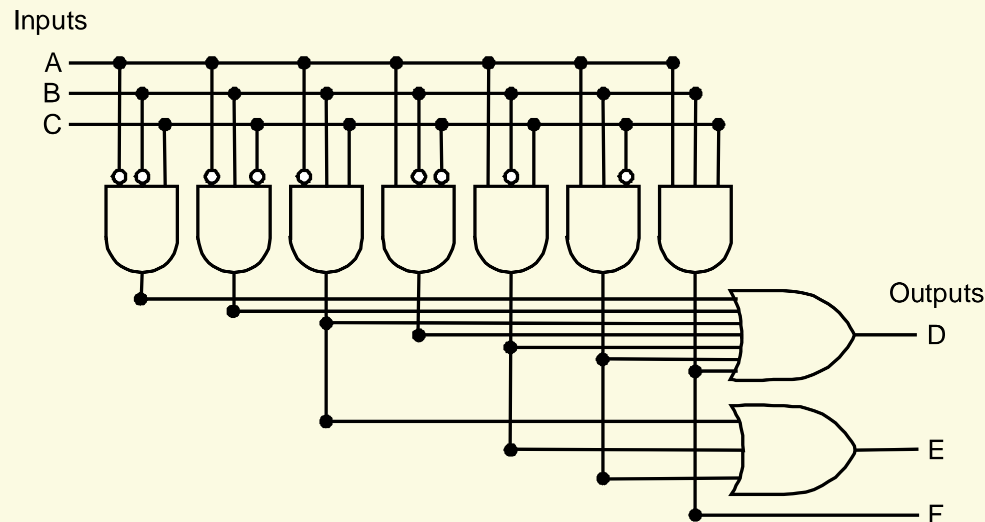
Example - Problem

- Show a PLA implementation of this example:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Example - Answer

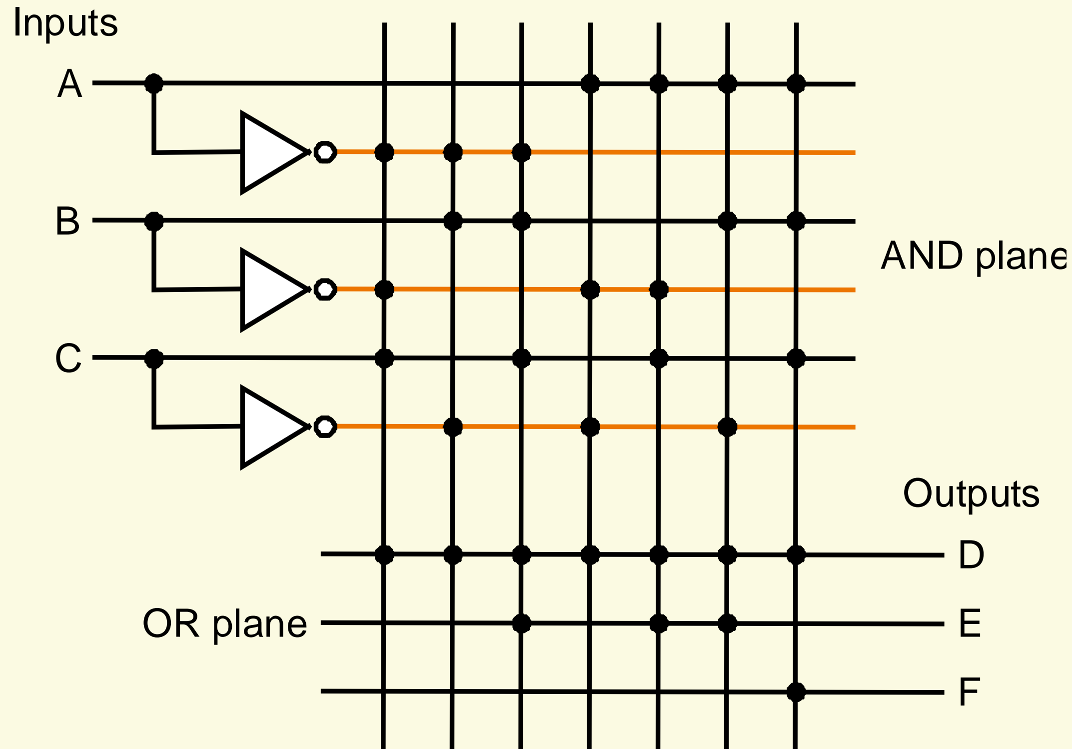
- There are seven unique product terms with at least one true value in the output section, and hence there are seven columns in the AND plane. There are three inputs and hence the number of rows in the AND plane is three.



- There are three outputs and hence the number of rows in the OR plane is three.

Example - Answer (cont'd)

- An equivalent PLA representation:



Read-Only Memories

- A **read-only memory** (**ROM**) is a memory which has a set of locations that can be read.

- Like PLA, **a ROM can implement any set of logic functions**:
  - **Inputs**: **address** lines that specify a memory location
  - **Outputs**: bits that hold the **content** of the specified memory location

- A ROM can represent a set of logic functions directly from the truth table. For M inputs and N logic functions, we need a ROM with:
  - **Height** (i.e., number of addressable entries) = 2^M
  - **Width** (i.e., number of output bits) = N

Read-Only Memories

- Typically, the contents of the memory locations in a ROM are fixed at the time the ROM is created, and hence it is read-only storage.

- **Programmable ROMs** (**PROMs**): ROMs that can be programmed electronically when a designer knows their contents.

- **Erasable PROMs** (**EPROMs**): PROMs that require a relatively slow erasure process (e.g., using UV light).

- **ROMs vs. PLAs**:
  - ROMs are fully decoded: they contain a full output word for every possible input combination. PLAs are only partially decoded.
  - PLAs are generally more efficient for implementing logic functions because they have fewer terms.
  - ROMs are better if the logic functions change over time.

Don't Cares

- In implementing combinational logic functions, there are situations when we do not care what the value of some input or output is. Such situations are referred to as **don't cares** and are represented by 'X' in the truth table.
- Two types: **input don't cares** and **output don't cares**
- Don't cares are important because they make it easier to optimize the implementation of a logic function.

Example - Problem

- Suppose a logic function is represented by the following truth table with output don't cares, which has seven product terms:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

Simplify the truth table to minimize the number of product terms.

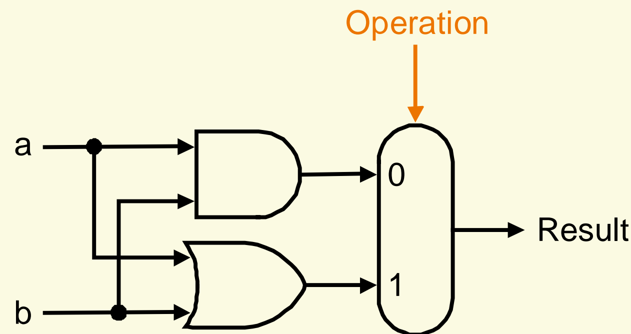
Problem - Answer

- The following simplified truth table requires a PLA with only four product terms.

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

Constructing an Arithmetic Logic Unit

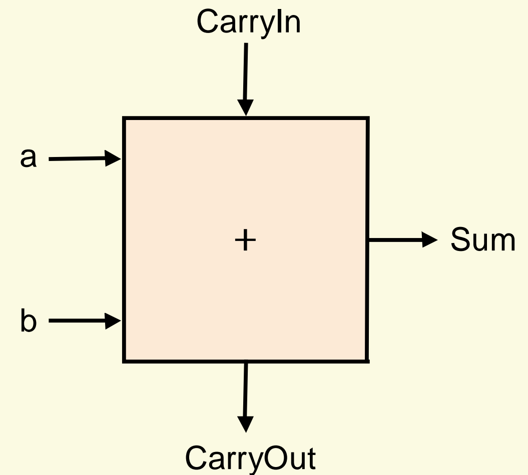
- Since a word in MIPS is 32 bits wide, we need a 32-bit ALU.
- **Assumption**: we can build a 32-bit ALU by connecting 32 1-bit ALUs together.
- **1-bit logical unit for AND and OR**:



  - A multiplexor selects the appropriate result depending on the operation specified.

1-Bit Adder

- **Inputs**:
  - Two inputs for operands
  - One input for carry from right neighbor
- **Outputs**:
  - One output for sum
  - One output for carry to left neighbor



- This is also called a **full adder** or a **(3, 2) adder** (because it has 3 inputs and 2 outputs).

Truth Table and Logic Equations for 1-Bit Adder

- Truth table:

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	SumOut	
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	$1 + 1 + 1 = 11_{two}$

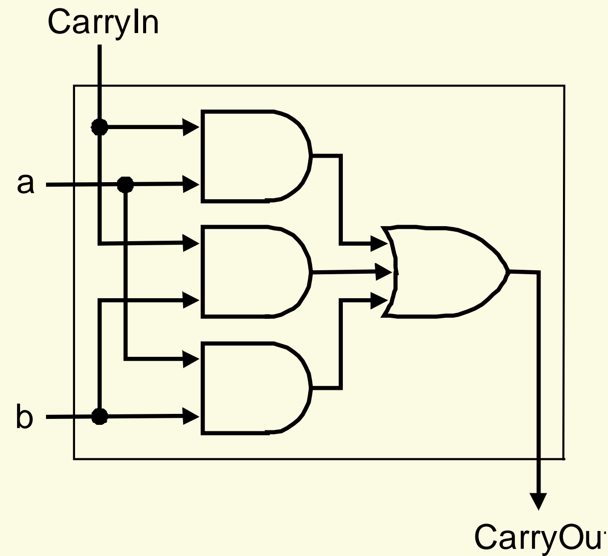
- Logic equations:

$$\begin{aligned}\text{CarryOut} &= (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn}) \\ &= (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)\end{aligned}$$

$$\begin{aligned}\text{SumOut} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) \\ + (a \cdot b \cdot \text{CarryIn})\end{aligned}$$

Hardware Implementation of 1-Bit Adder

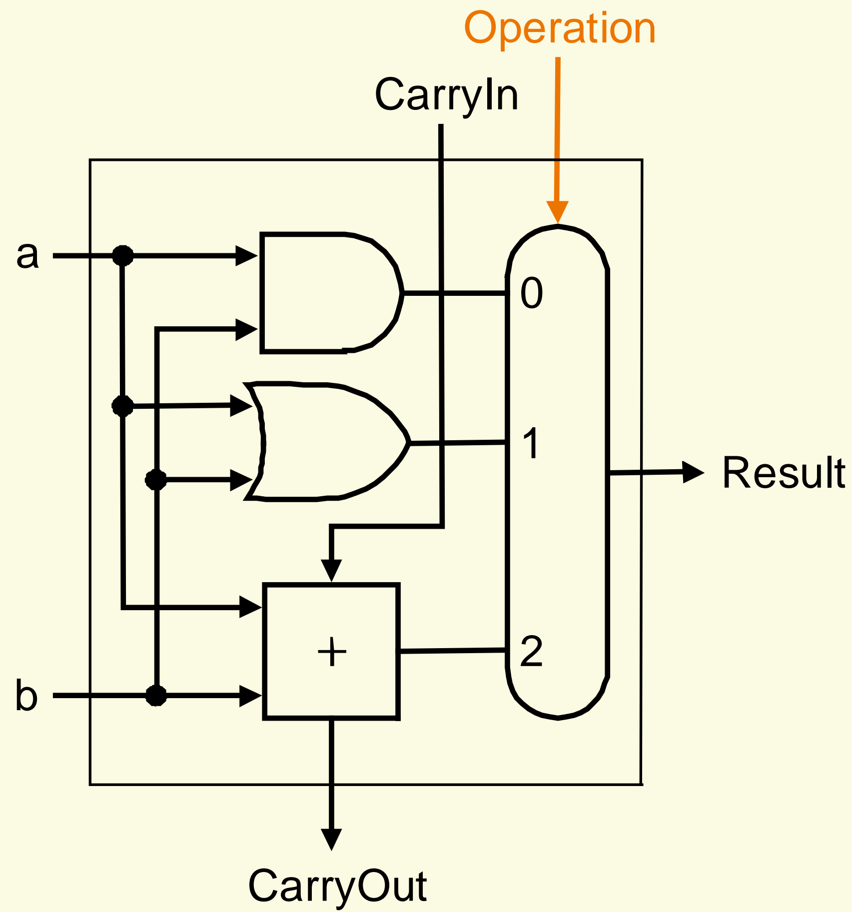
- CarryOut bit:



- SumOut bit:

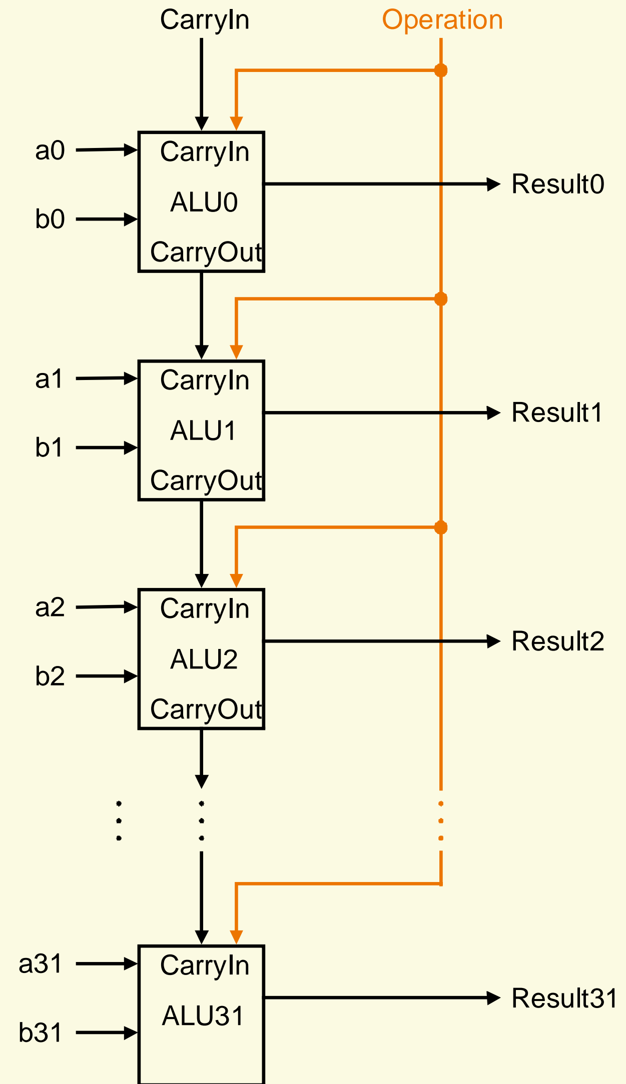
??

1-Bit ALU (AND, OR, and Addition)



32-Bit ALU

- **Ripple carry** organization of a 32-bit ALU constructed from 32 1-bit ALUs:
  - A single carry out of the least significant bit (Result0) could ripple all the way through the adder, causing a carry out of the most significant bit (Result31).
  - There exist more efficient implementations (based on the **carry lookahead** idea).



Subtraction

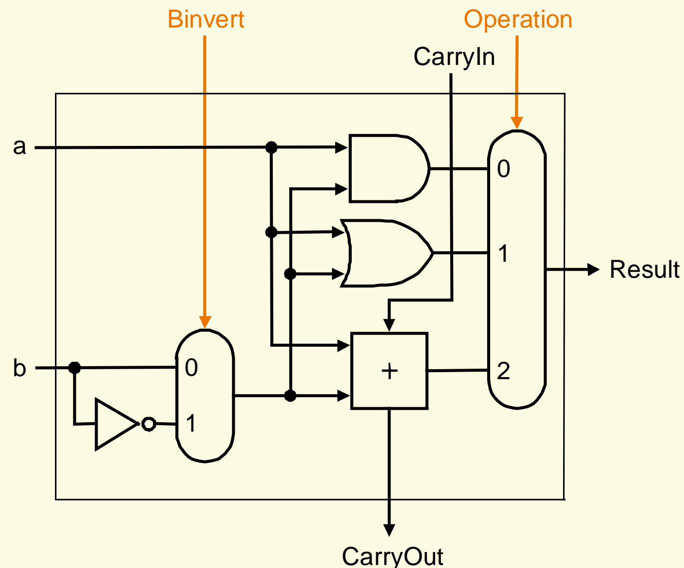
- **Subtraction** is the same as adding the negative version of an operand.

- **Shortcut** for negating a two's complement number:
  - Invert each bit (to get the one's complement representation).
  - Add 1.

- An adder can be used for **both** addition and subtraction. A 2:1 multiplexor is used to choose between an operand (for **addition**) and its negative version (for **subtraction**).

1-Bit ALU (AND, OR, Addition, and Subtraction)

- Binvert is the selector input of a multiplexor that chooses between addition and subtraction.



- To connect 32 of these 1-bit ALUs to form a 32-bit ALU, the CarryIn input of the least significant bit is set to 1 when subtraction is performed (needed for the second step in computing a two's complement number).

Carry Lookahead

- Using the ripple carry adder, the carry has to propagate from the least significant bit to the most significant bit in a sequential manner, passing through all the 32 1-bit adders one at a time. This is too slow for time-critical hardware.

- Key idea behind fast carry schemes **without the ripple effect**:

$$\text{CarryIn2} = (b1 \cdot \text{CarryIn1}) + (a1 \cdot \text{CarryIn1}) + (a1 \cdot b1)$$

$$\text{CarryIn1} = (b0 \cdot \text{CarryIn0}) + (a0 \cdot \text{CarryIn0}) + (a0 \cdot b0)$$

  - Substituting the latter into the former, we have:

$$\begin{aligned}\text{CarryIn2} = {} & (a1 \cdot a0 \cdot b0) + (a1 \cdot a0 \cdot \text{CarryIn0}) + (a1 \cdot b0 \cdot \text{CarryIn0}) \\ & + (b1 \cdot a0 \cdot b0) + (b1 \cdot a0 \cdot \text{CarryIn0}) + (b1 \cdot b0 \cdot \text{CarryIn0}) \\ & + (a1 \cdot b1)\end{aligned}$$

  - Other CarryIn bits can also be expressed using CarryIn0.