COMP2611: Computer Organization

Arithmetic for Computers

Major Goals

- Revisit addition & subtraction for 2's complement numbers
- Explain the construction of a 32-bit arithmetic logic unit (ALU)
- Show algorithms and implementations of multiplication and division
- Demonstrate floating-point arithmetic operations

1. 2's Complement Arithmetic

Numbers

- Bits: the basis for binary number representation in digital computers

- We've learnt:
  - ✓ How to represent negative numbers
  - ✓ How to represent fractions and real numbers
  - ✓ What is the representable range

Signed and Unsigned Numbers

- **Signed numbers**
  - **negative** or **non-negative** integers, e.g. `int` in C/C++
- **Unsigned numbers**
  - **non-negative** integers, e.g. `unsigned int` in C/C++
- **Operations for unsigned numbers**
  - Comparison:

    `sltu (set on less than unsigned), sltiu`
  - Arithmetic:

    `addu, subu (add/subtract unsigned)`

    - Treat values of all registers as non-negative

    `addiu (add unsigned sign-extended immediate)`

    - The 16-bit immediate is sign-extended then addition as above

    `addi (add signed immediate)`
  - Load:

    `lbu (load byte unsigned), lhu (load half unsigned)`

Signed vs. Unsigned Comparison

- `$s0` $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$
- `$s1` $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$

- What are the values in registers `$t0` and `$t1` in the examples below?

```
slt $t0, $s0, $s1    # signed comparison
```

$\$s0 = -1_{10}$, $\$s1 = 1_{10}$, $\$t0 = 1$

```
sltu $t1, $s0, $s1   # unsigned comparison
```

$\$s0 = 4294967295_{10}$, $\$s1 = 1_{10}$, $\$t1 = 0$

Zero Extension and Sign Extension

- Operands in instruction may have mis-matched sizes

  `addi $t0, $s0, -5` # add 32-bit operands in $s0 with 16-bit immediate value in 2's complement

  `ori $t0, $s0, 0xFB32`

  `lb $t0, 0($s0)` # load a 8-bit signed number to 32-bit register

- Need conversion from **n**-bit binary numbers into **m**-bit numbers (**m** > **n**)

- **Sign extension:** Fill the leftmost bits (**n**-th ~ (**m**-1)-th) with the sign bit

  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

  `addi, lb`

- **Zero extension:** Pad the leftmost bits (**n**-th ~ (**m**-1)-th) with 0
  - `and, or`

Confusion on Unsigned Operation

- add immediate unsigned **`addiu $t0, $s1, 0xFB32`**
- Zero extended? **WRONG**

- MIPS unsigned arithmetic operations: **`addu`, `addiu`, `subu`, ~~`subiu`~~, `sltu`, `sltiu`**

- The immediate fields of **`addiu`**, **`sltiu`** are **sign-extended**!

The MIPS32 Instruction Set states that the word 'unsigned' as part of add and subtract instructions, is a misnomer. The difference between signed and unsigned versions of commands is not a sign extension of the operands, but controls whether a trap is executed on overflow (i.e. `add`) or an overflow is ignored (i.e. `addu`). An immediate operand CONST to these instructions is always sign-extended.

Addition and Subtraction

- **Addition**

  Bits are added bit by bit **from right to left**, with **carries** passed to the next bit position 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 (32 bits in MIPS)

Examples

- **Addition** (7 + 6 = 13):

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 & = \quad 7_{10} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_2 & = \quad 6_{10} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_2 & = \quad 13_{10} \end{array}$$

- **Subtraction** (7 - 6 = 1):

$$\begin{array}{lll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_2 & = \quad 7_{10} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_2 & = \quad -6_{10} \\ \hline = & \boxed{1}\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 & = \quad 1_{10} \end{array}$$

Examples

- **Addition** ($1073741824 + 1073741824 = 2147483648$):

$$
\begin{array}{rl}
 & 0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 1073741824_{10} \\
+ & 0100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 1073741824_{10} \\
\hline
= & 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 \neq 2147483648_{10}
\end{array}
$$

In 2's complement the MSb is a sign bit

then, it means -2147483648_{10}

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 condition**

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

MIPS detects overflow with an **exception** (also called an **interrupt**)

- Exceptions occur when unscheduled events disrupt program execution
- Some instructions are designed to cause exceptions on overflow

  e.g. **`add`**, **`addi`** and **`sub`** cause exceptions on overflow

  But, **`addu`**, **`addiu`** and **`subu`** do not cause exceptions on overflow; programmers are responsible for using them correctly

When an overflow exception occurs

- Control jumps to a **predefined address (code)** to **handle the exception**
- The interrupted address is saved to **EPC** for possible resumption

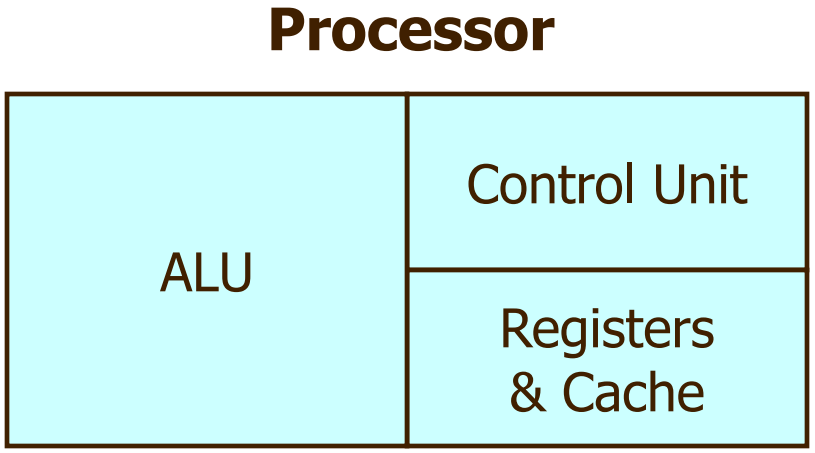
  EPC = **exception program counter**; a special register

  MIPS software return to the offending instruction via jump register

2. Arithmetic Logic Unit

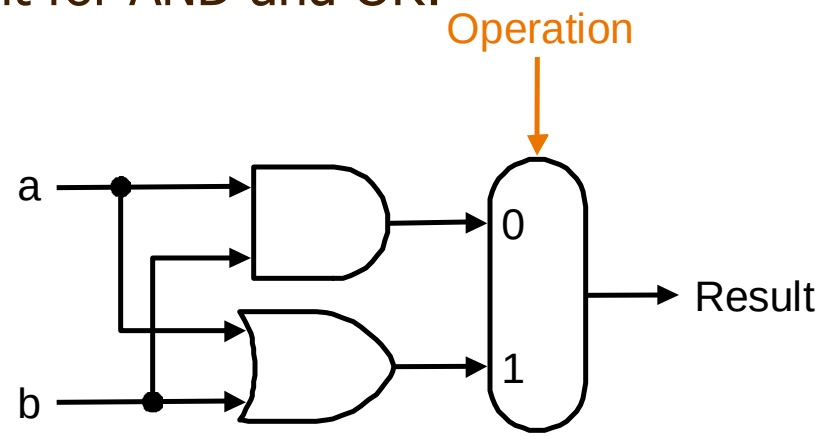
Constructing an Arithmetic Logic Unit

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



Constructing an Arithmetic Logic Unit (cont'd)

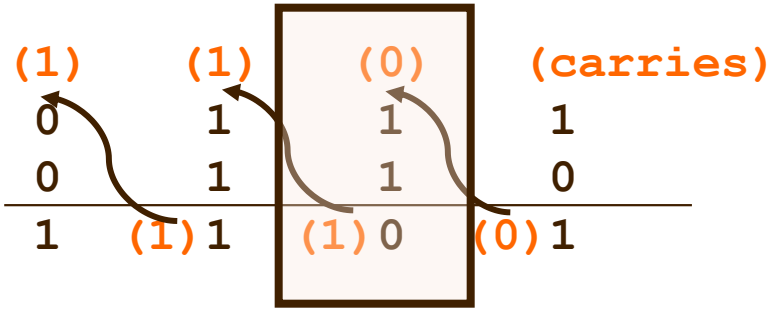
- Since a word in MIPS is 32 bits wide, we need a 32-bit ALU
- Ideally, we can build a 32-bit ALU by connecting 32 1-bit ALUs together (each of them takes care of the operation on one bit position)
- **1-bit** logical unit for AND and OR:

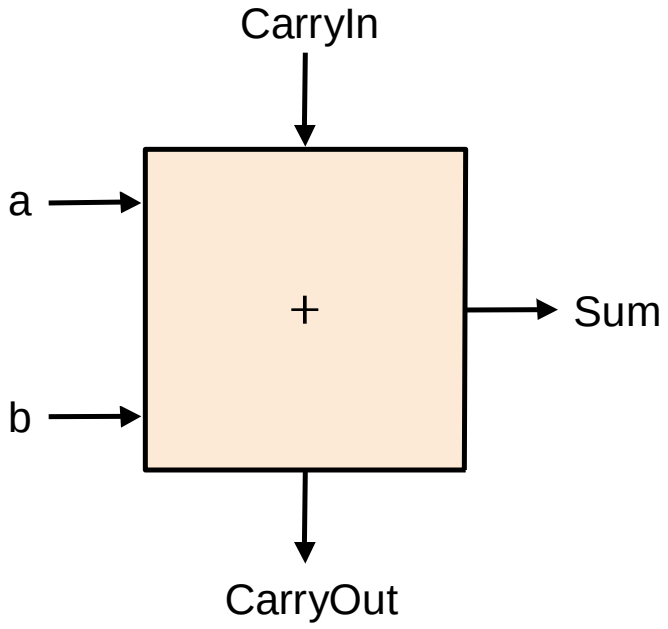


A multiplexor selects the appropriate result depending on the operation specified

1-Bit Full Adder

- An adder must have
  - Two inputs (bits) for the operands
  - A single-bit output for the sum
- Also, must have a second output to pass on the carry, called carry-out
  - Carry-out becomes the carry-in to the neighbouring adder
- 1-bit full adder is also called a **(3, 2) adder** (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	$0 + 0 + 0 = 00_2$
0	0	1	0	1	$0 + 0 + 1 = 01_2$
0	1	0	0	1	$0 + 1 + 0 = 01_2$
0	1	1	1	0	$0 + 1 + 1 = 10_2$
1	0	0	0	1	$1 + 0 + 0 = 01_2$
1	0	1	1	0	$1 + 0 + 1 = 10_2$
1	1	0	1	0	$1 + 1 + 0 = 10_2$
1	1	1	1	1	$1 + 1 + 1 = 11_2$

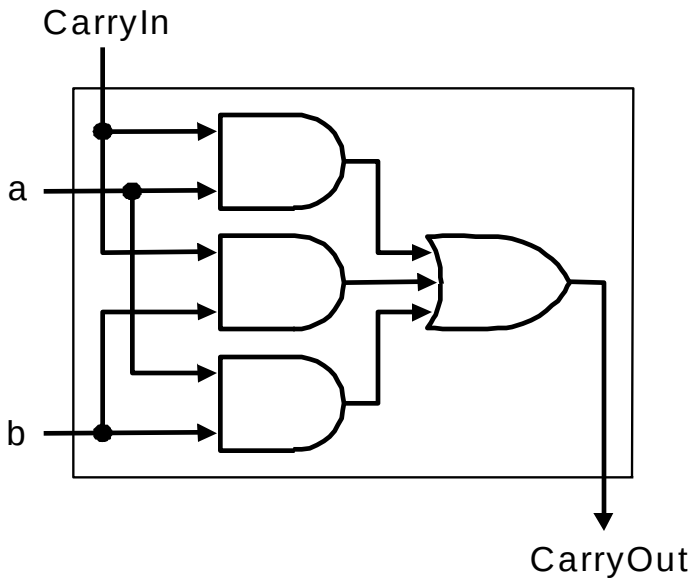
- Logic equations:

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

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

Hardware Implementation of 1-Bit Adder

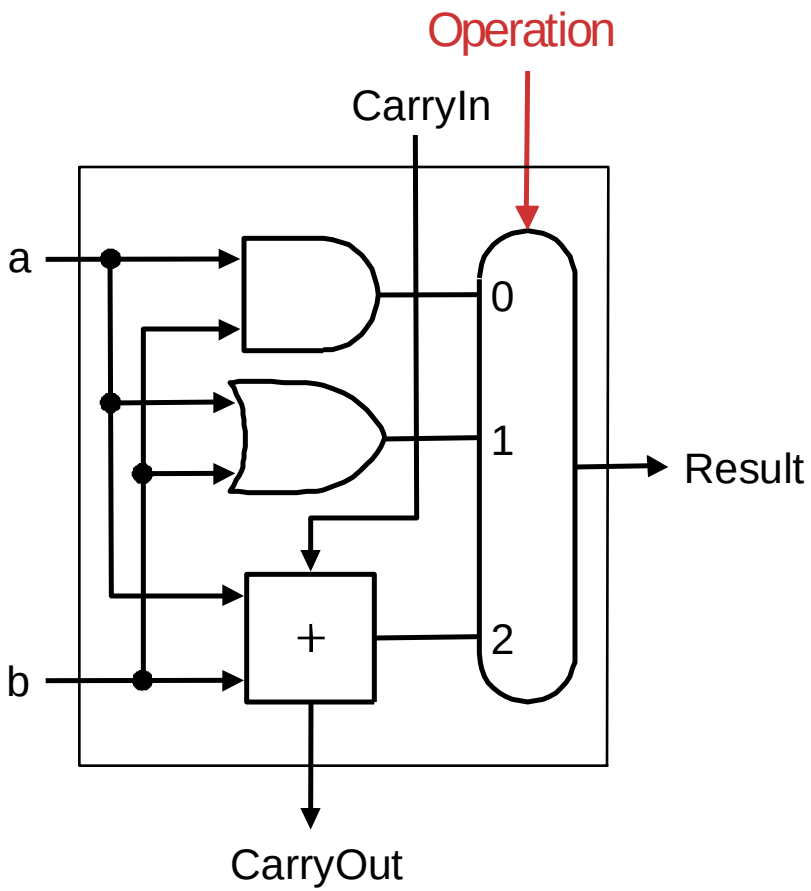
- $\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$
  $= (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$



- SumOut bit: (It is left as an exercise)

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

- 3 in 1 building block
  - Use the Operation bits to decide what result to push out
  - Operation = 0, do AND
  - Operation = 1, do OR
  - Operation = 2, do addition

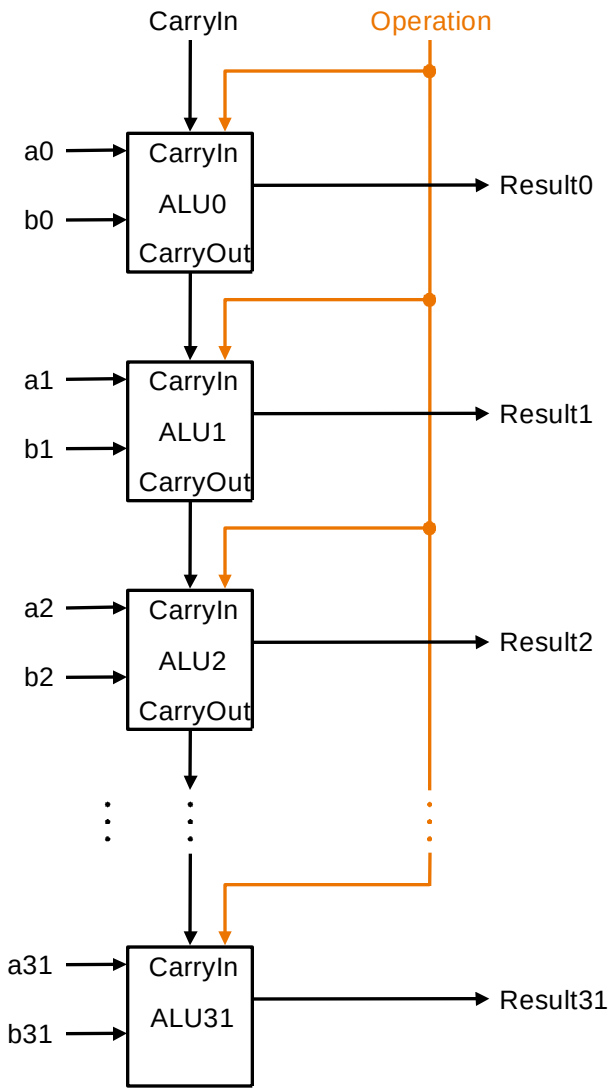


32-Bit ALU

- **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 adders, causing a carry out of the most significant bit (Result31)

  There exist more efficient implementations (based on the **carry lookahead** idea to be explained later)



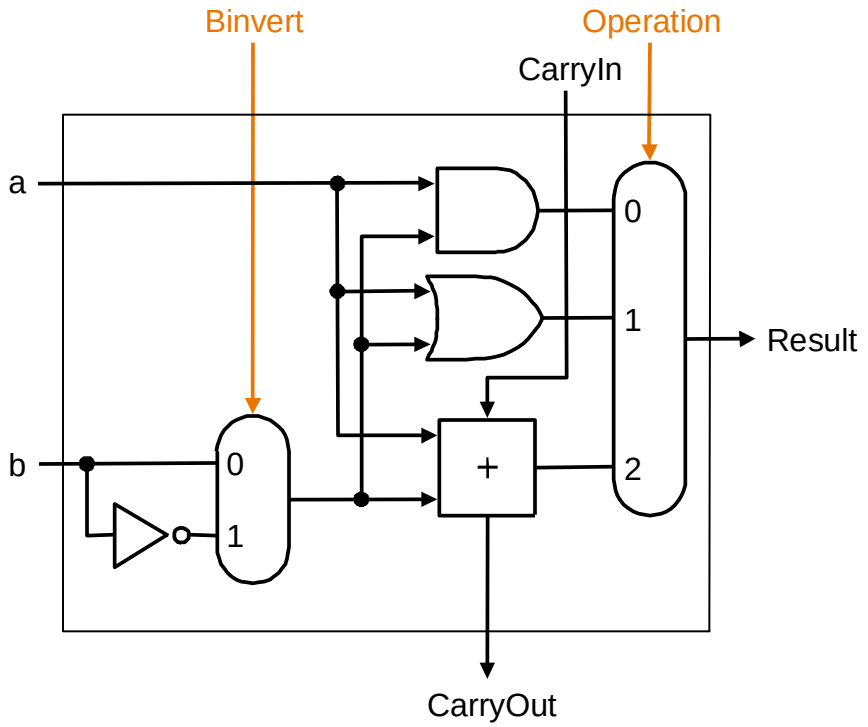
Subtraction

- **Subtraction** is the same as adding the negated operand
- By doing so, 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**)

- **Shortcut** for negating a 2's complement number:
  - Invert each bit (to get the 1's complement representation)
  - Add 1: Obtained by setting the ALU0's carry bit to 1

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

- To execute a – b we can execute a + (-b)
- Binvert: the selector input of a multiplexor to choose between addition and subtraction



- To form a 32-bit ALU, connect 32 of these 1-bit ALUs
- To negate b we must invert it and add 1 (2's complement), so we must Set CarryIn input of the least significant bit (ALU0) to 1 for subtraction

Tailoring the ALU for MIPS

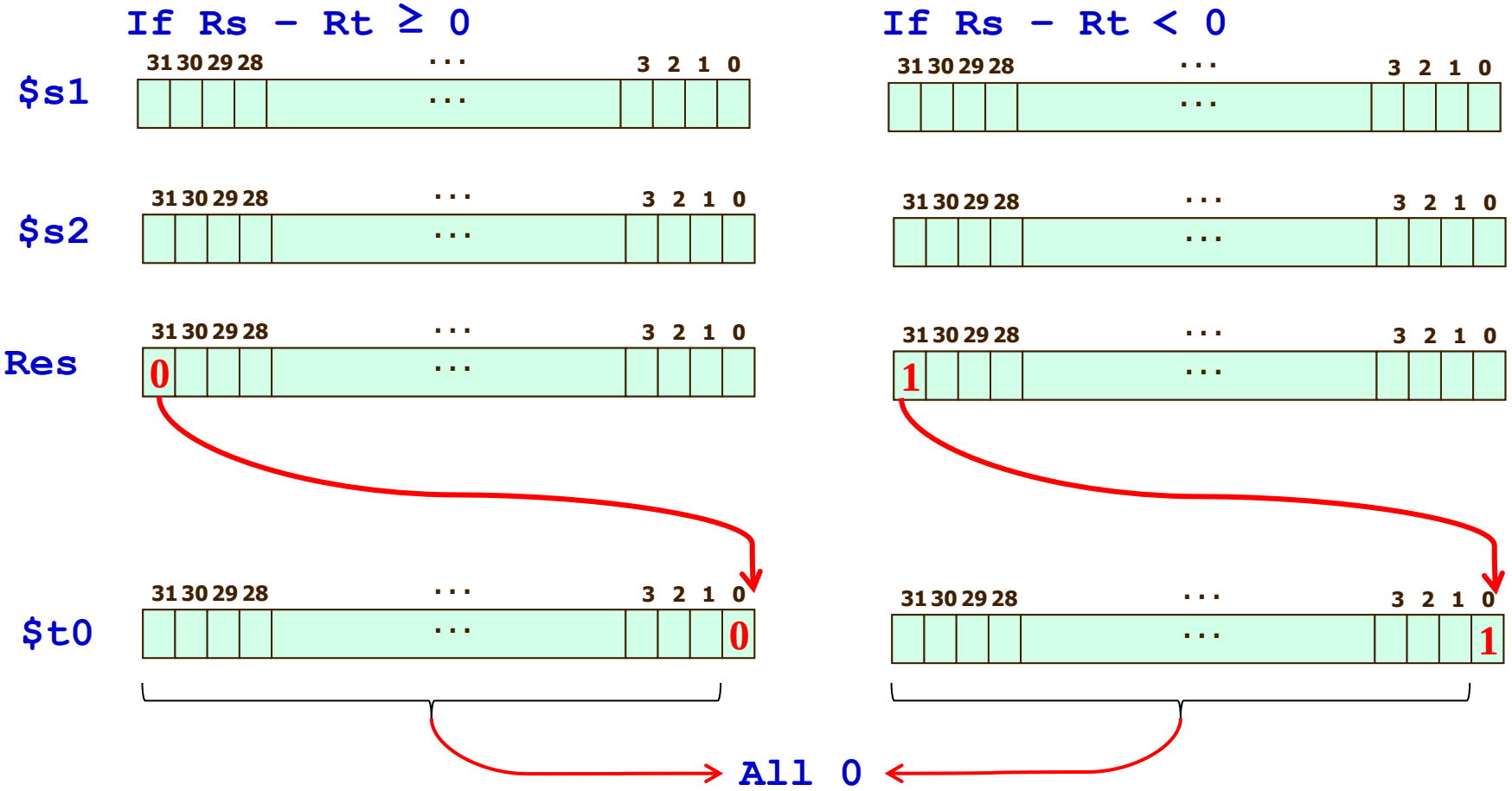
- The 32-bit ALU being designed so far can perform **`add`**, **`sub`**, **`and`**, **`or`** operations which constitute a large portion of MIPS' instruction set
- Two instructions not yet supported are: **`slt`** and **`beq`**

- When we need to compare **`Rs`** to **`Rt`**
  - By definition of **`slt`**, if **`Rs`** < **`Rt`**
    - LSb of the output is set to 1
    - Otherwise, it is reset to 0
- How to implement it?
  - The comparison is equivalent to testing if **`(Rs – Rt) < 0`**
  - If (**`Rs – Rt`**) is smaller than **`0`**
    - MSb of the subtraction (**`Rs – Rt`**) equals to 1 (means negative)
    - Otherwise, MSb of the subtraction equals to 0
  - Notice that the outcome of MSb is similar to the result of **`slt`**
  - ☑ Idea: **copy the MSb of the subtraction result to the LSb of `slt`'s output. All other bits of the output are 0**
  - ☑ **`slt`** can be done using two types of 1-bit ALUs

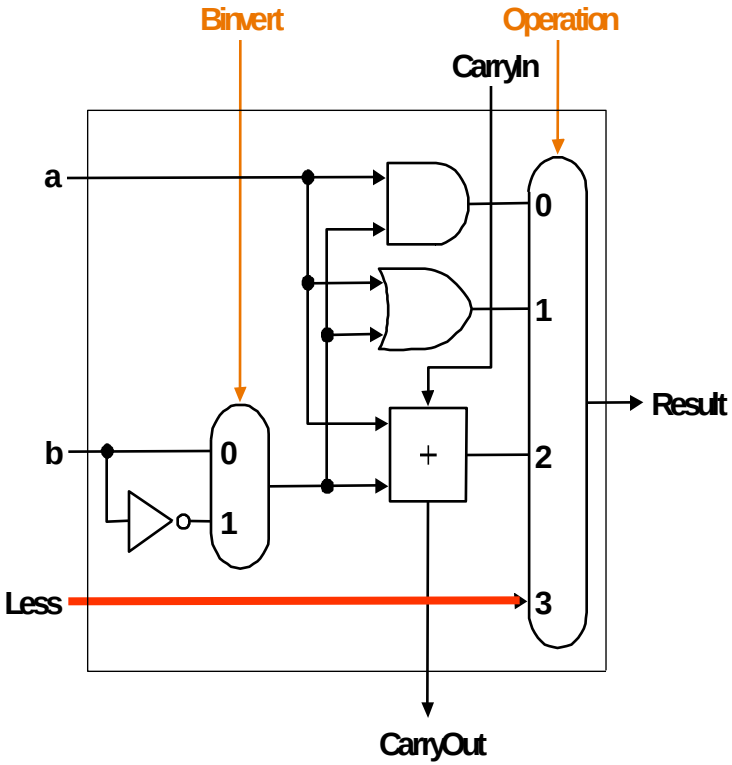
Tailoring the ALU for MIPS

- How to implement it?
  - The comparison is equivalent to testing if `(Rs – Rt) < 0`

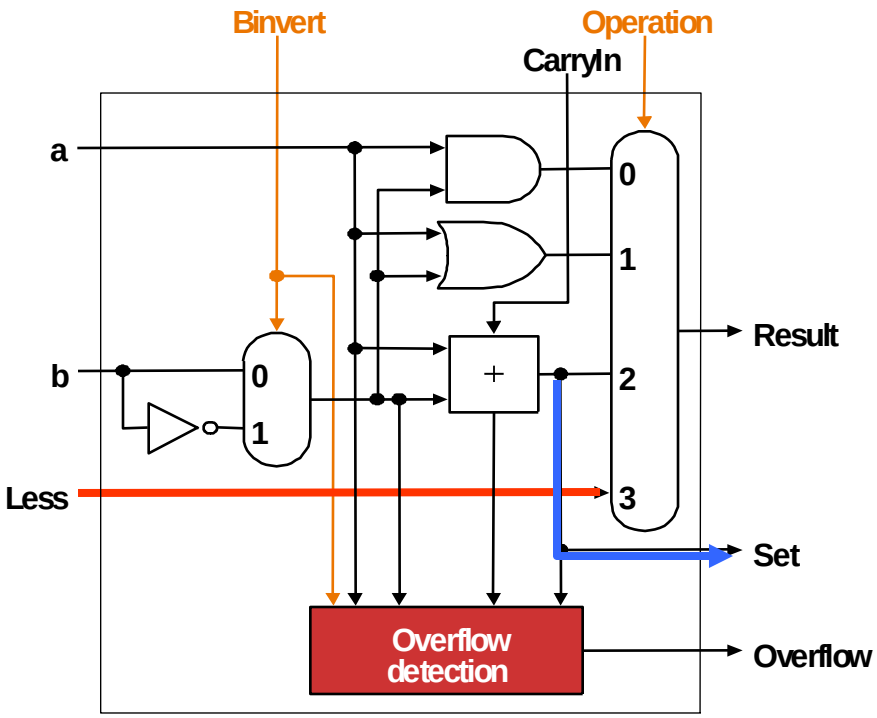
```
slt $t0, $s1, $s2
```



Tailoring the ALU for MIPS (cont'd)



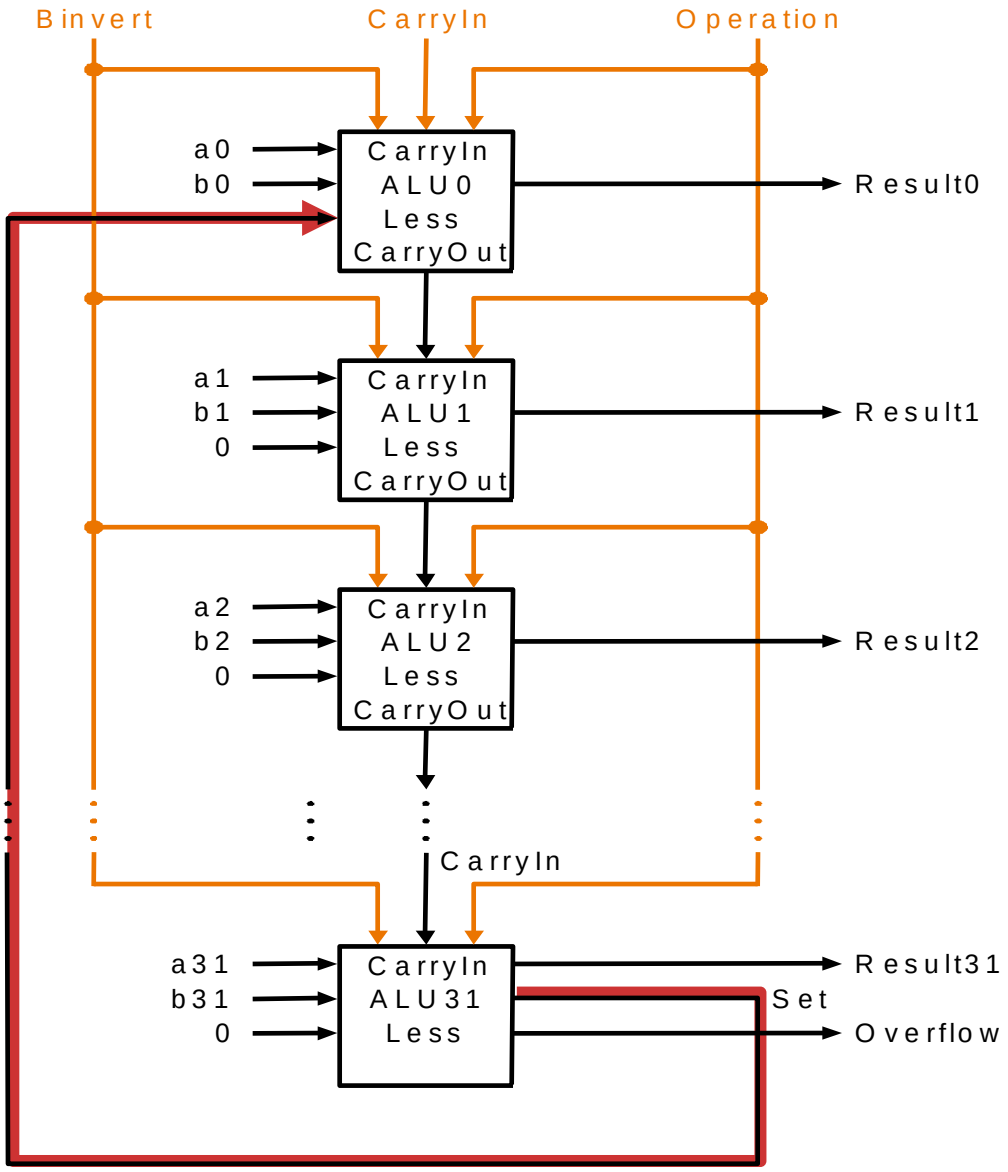
1-bit ALU for bits 0 to 30



1-bit ALU for the MSb (bit 31)

32-Bit ALU with (add, sub, AND, OR, slt)

- The "set" signal is the MSb of the result of the subtraction, A – B
- It is passed to LSB
- Result0 will equal to this "set" signal when operation = 3 (which means `slt` instruction is being executed)

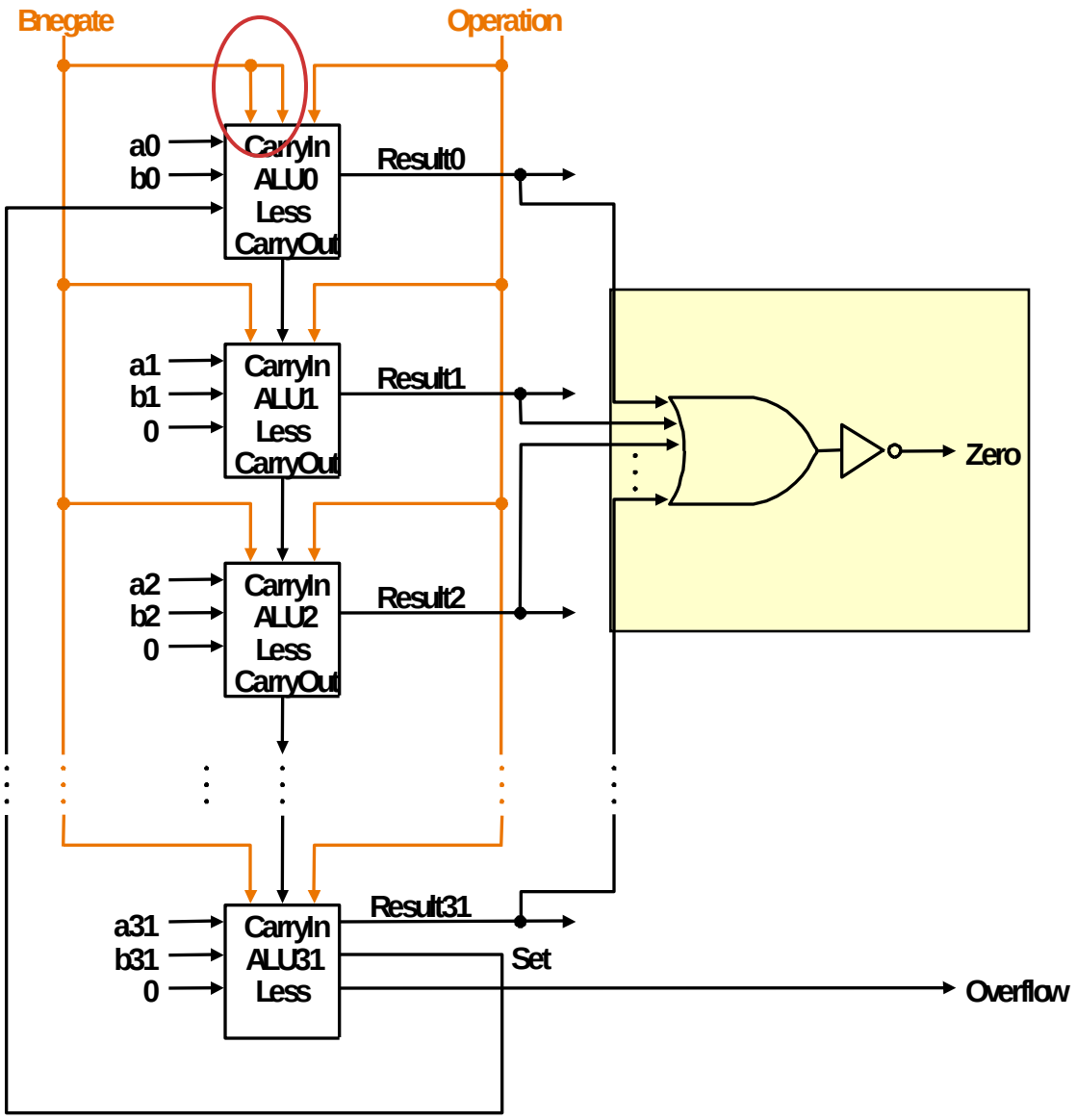


Tailoring the ALU for MIPS

- To support `beq`
- We need to compare `Rs` to `Rt`
  - The comparison is equivalent to testing if `(Rs - Rt) == 0`
  - If (`Rs - Rt`) is equal to `0`
    - All bits of the output are 0
    - Otherwise, at least one of them is non 0

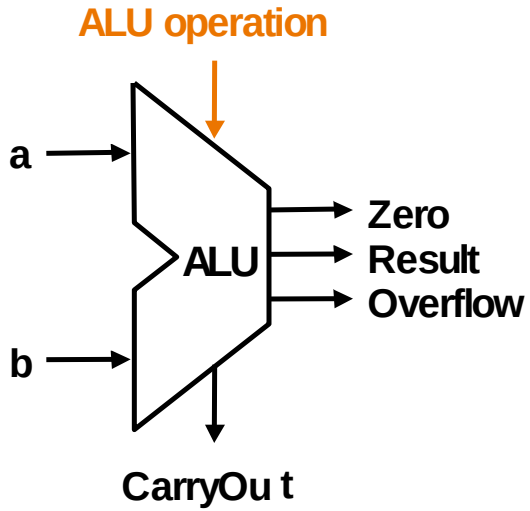
32-Bit ALU with (add, sub, AND, OR, slt)

- Finally, this adds a zero detector
- For addition and AND/OR operations both `Bnegate` and `CarryIn` are 0 and for subtract, they are both 1 so we combine them into a single line



Universal Representation

- Knowing what is exactly inside a 32-bits ALU, from now on we will use the universal symbol for a complete ALU as follows:



ALU Control lines	Operation
000	AND
001	OR
010	ADD
110	SUB
111	SLT

Carry Lookahead

- Using the ripple carry adder, the carry has to propagate from the LSb to the MSb in a sequential manner, passing through all the 32 1-bit adders one at a time. 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:

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

All other CarryIn bits can also be expressed using CarryIn0

Carry Lookahead

- A Bit position generates a Carry iff both inputs are 1: $G_i = a_i \cdot b_i$
- A Bit position propagates a Carry if exactly one input is 1: $P_i = a_i + b_i$
- Carryout at bit i can be expressed as:

$$C_i = G_i + P_i \cdot C_{i-1}$$

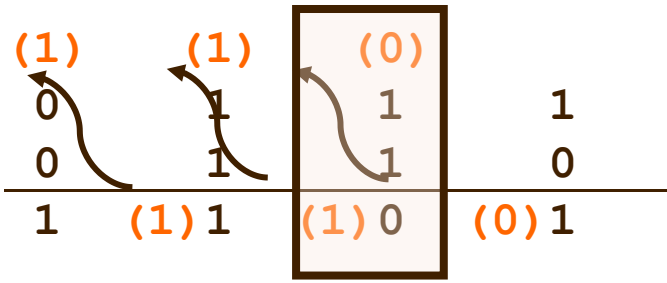
- After substitution we have

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + G_0 P_1 + C_0 P_0 P_1$$

$$C_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2$$

$$C_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3$$



- WE can build a circuit to predict all Carries at the same time and do the additions in parallel
- Possible because electronic chips becoming cheaper and denser

Carry Lookahead Adder