COMP2611: Computer Organization

Arithmetic for Computers

Major Goals

- ❑ Review 2's complement numbers and introduce their addition & subtraction
- ❑ 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

- Questions to answer next:
  - ✓ How to represent negative numbers
  - ✓ How to represent fractions and real numbers
  - ❍ How to handle numbers that go beyond the representable range
    - What is a representable range?

2's Complement Representation

- All computers use 2's complement representation for signed numbers

- Bit 31 is called the sign bit: **(**0 for non-negative, 1 for negative**)**
  - The positive half uses the same representation as before
  - The negative half uses conversion illustrated below:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_2 = 6_{10}$

$\downarrow$ i) Invert bits to get 1's complement

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1001_2 = -7_{10}$

$\downarrow$ ii) Add 1 to get 2's complement

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_2 = -6_{10}$

- Largest integer represented by a MIPS word:

  $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = (2^{31} - 1)_{10} = 2{,}147{,}483{,}647_{10}$
- Smallest integer represented by a MIPS word:

  $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2^{31}{}_{10} = -2{,}147{,}483{,}648_{10}$

- Immediate part for **lw**, **sw** & **addi** are represented in 2's complement

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`

Sign Extension

- e.g. `lb $t0, 0($s0)` # load a 8-bit signed number to 32-bit register
  - Bits 0~7 of `$t0` will contain the **byte** value stored at `0($s0)`
  - If the **byte** is a negative number, what happens to bits 8~24 of `$t0`?

Conversion of **n**-bit binary signed numbers into **m**-bit numbers (**m** > **n**)

- Done by filling the leftmost bits (**n**-th ~ (**m**-1)-th) with the sign bit
- For example:
  - 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
- If the immediate in the `addi` instruction = 1111 1111 1111 1110, the sign is extended as shown above before the ALU starts addition

- For unsigned addition operation `addiu`, **sign is also extended** as shown above before the ALU starts addition

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)

Examples

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

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

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

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

Examples

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

```
     0100 0000 0000 0000 0000 0000 0000 0000₂  = 1073741824₁₀
+    0100 0000 0000 0000 0000 0000 0000 0000₂  = 1073741824₁₀
   ------------------------------------------------------------
=    1000 0000 0000 0000 0000 0000 0000 0000₂  ≠ 2147483648₁₀
```

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

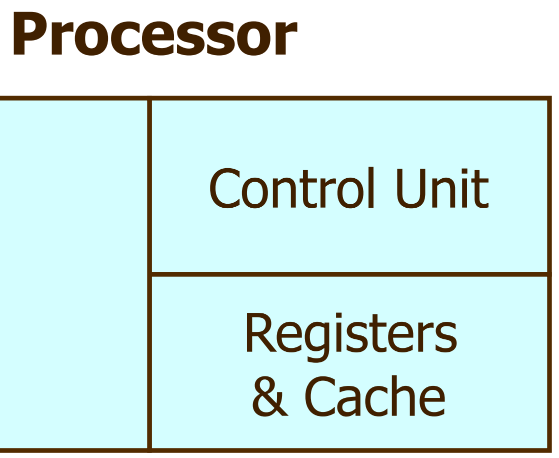
When an overflow exception occurs

- 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

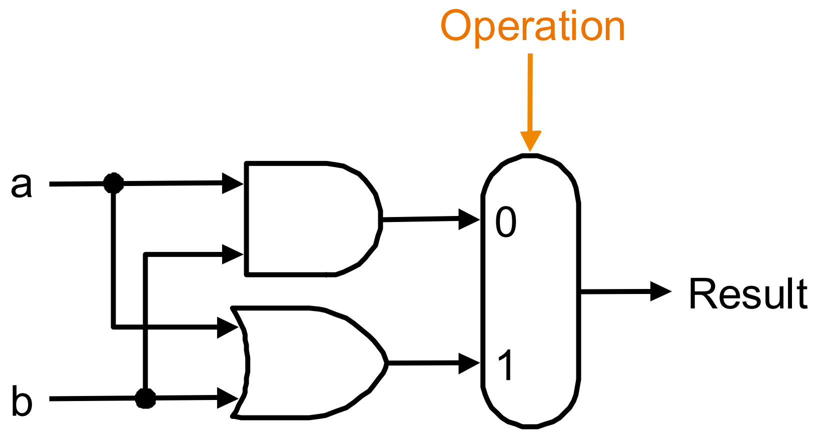
Constructing an Arithmetic Logic Unit

- 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)

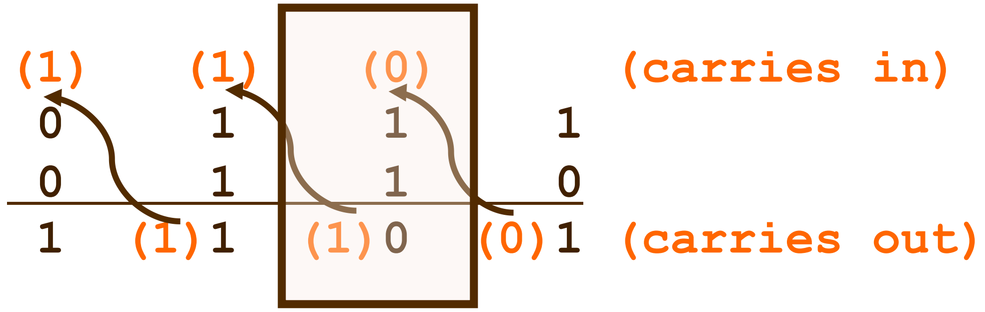
- Since a word in MIPS is 32 bits wide, we need a 32-bit ALU
- Ideally, we can build a 32-bit ALU by connecting 32 1-bit ALUs together (each of them takes care of the operation on one bit position)
- **1-bit** logical unit for AND and OR:

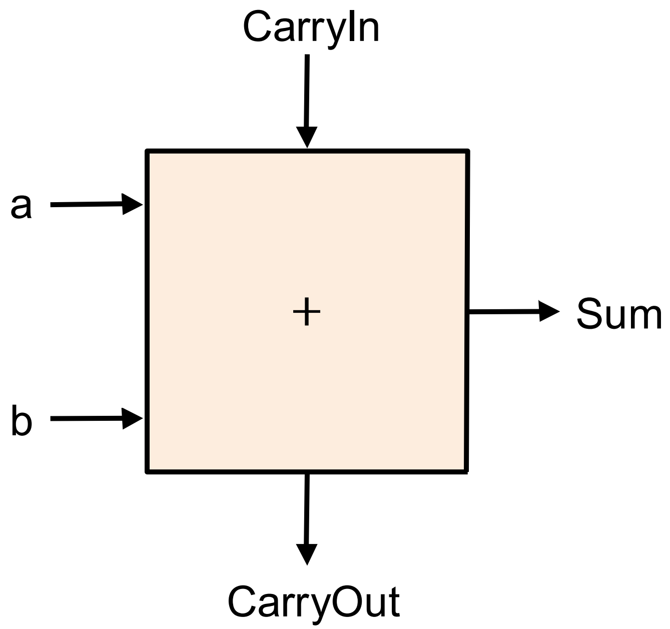


  - A multiplexor selects the appropriate result depending on the operation specified

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	$1 + 1 + 1 = 11_2$

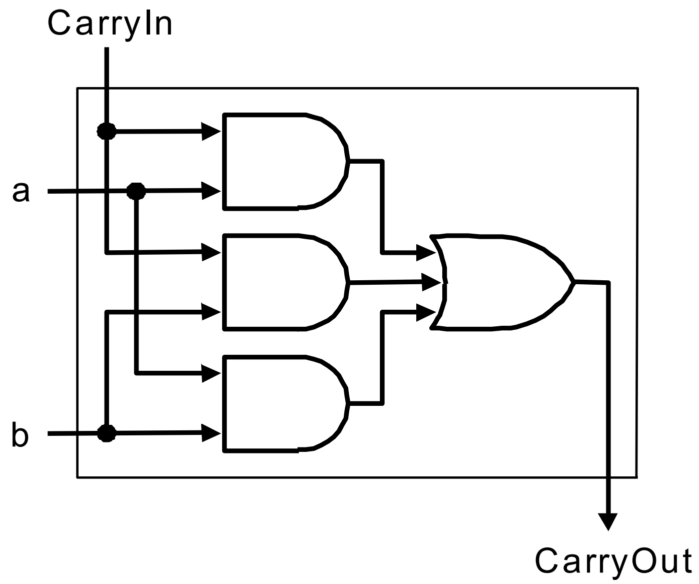
- Logic equations:

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

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

Hardware Implementation of 1-Bit Adder

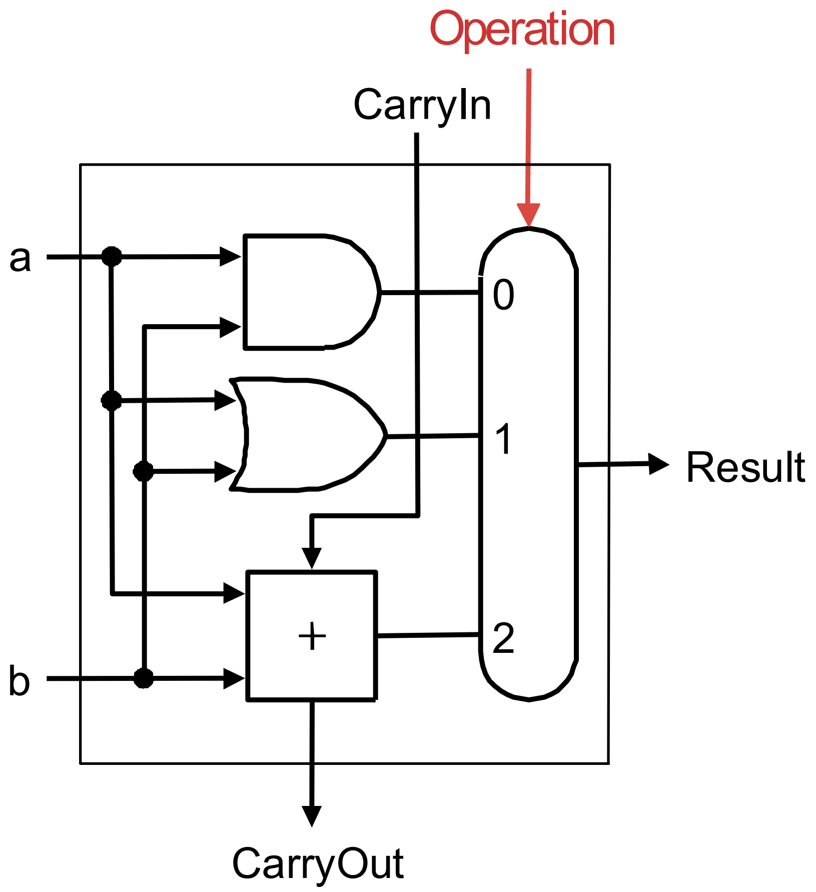
- $\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$
  $= (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$



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

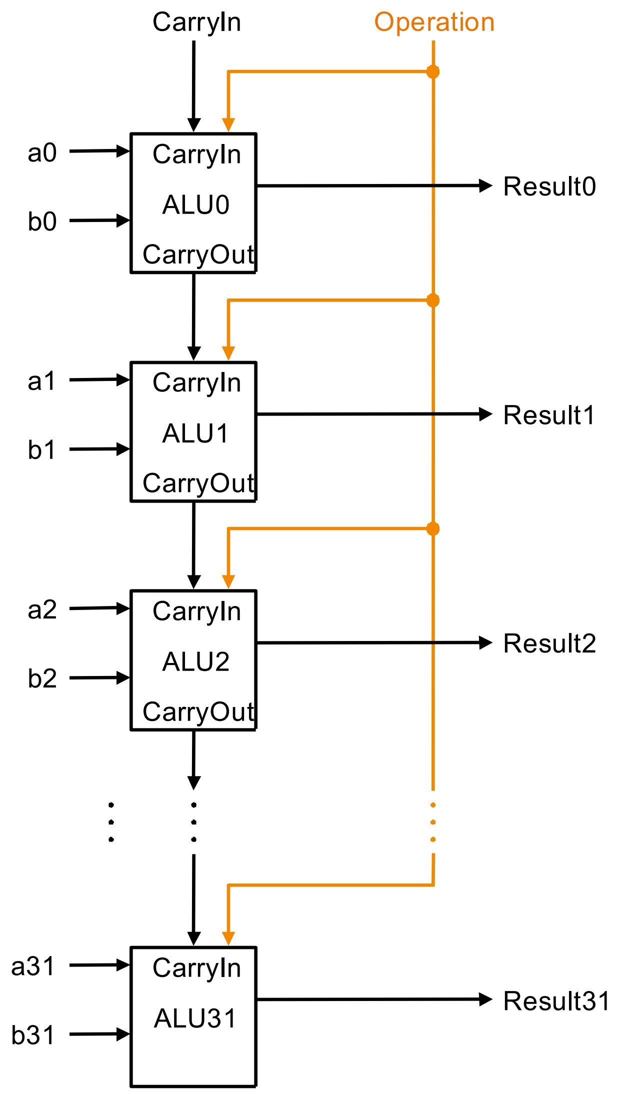
1-Bit ALU (AND, OR, and Addition)

- 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)



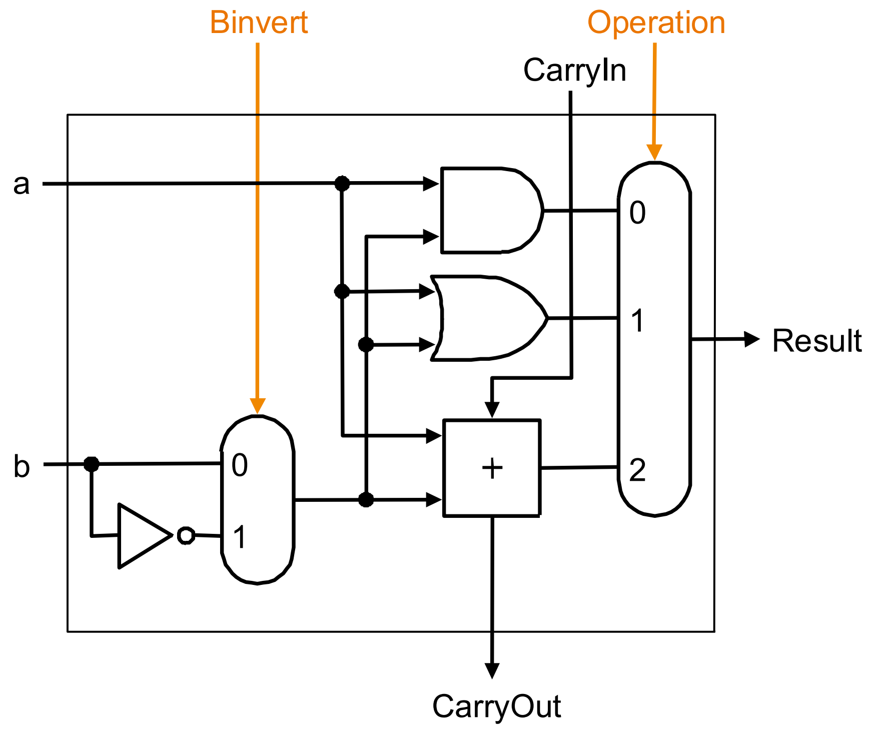
Subtraction

- **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`

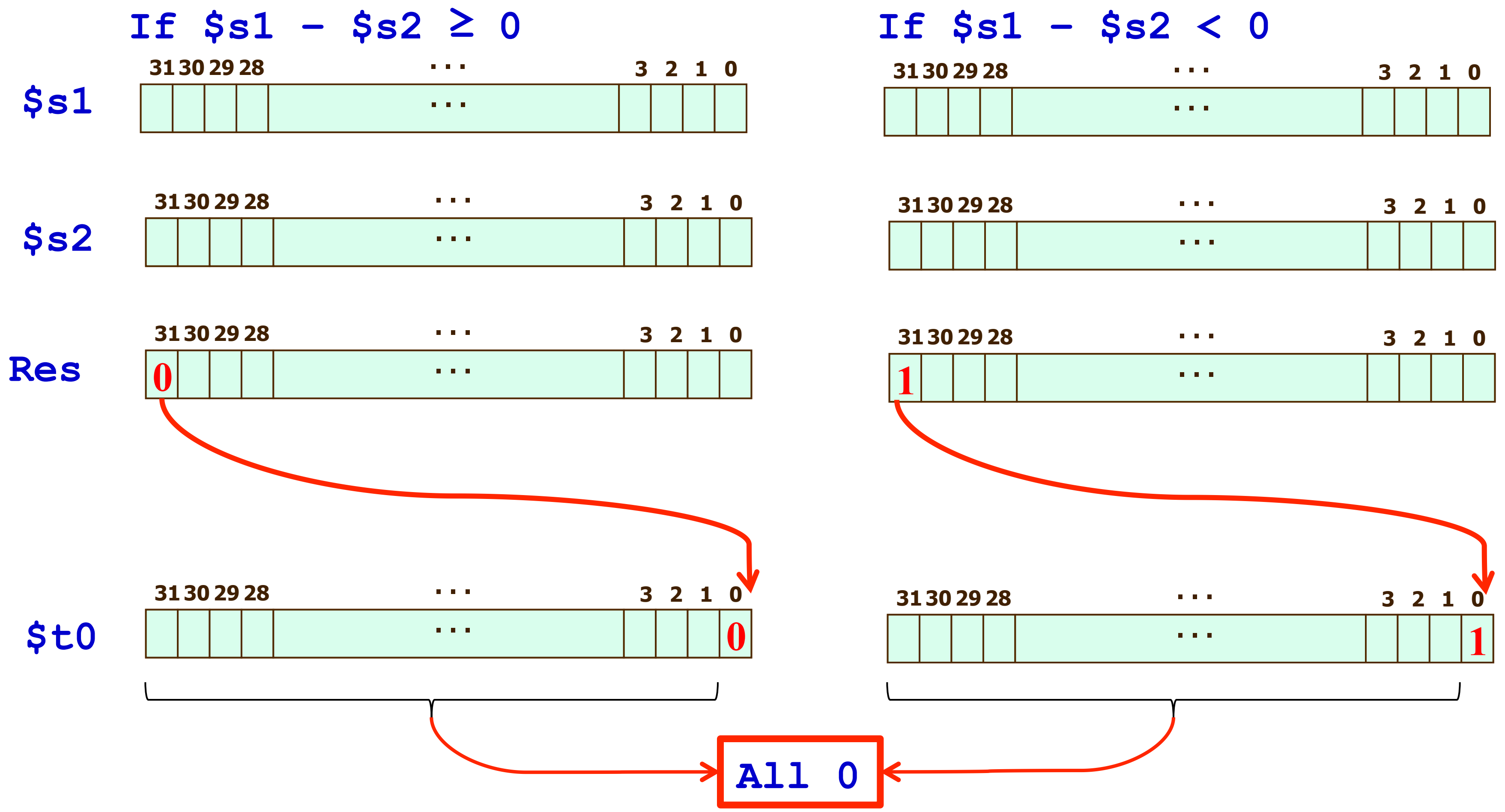
- ❑ When we need to compare `Rs` to `Rt`
  - ❍ By definition of `slt`, if `Rs` < `Rt`
    - LSb of the output is set to 1
    - Otherwise, it is reset to 0
- ❑ How to implement it?
  - ❍ `If(Rs – Rt) < 0`
    - MSb of the result of (`Rs – Rt`) equals to 1 (means <u>negative</u>)
    - Otherwise, MSb of the result equals to 0
  - ❍ Notice that the outcome of MSb is similar to the result of `slt`

  - ❍ Idea: **Do `(Rs – Rt):`**
    - ☑**copy the <u>MSb</u> of the result to the <u>LSb</u> of `slt` output;**
    - ☑**Set all other bits of `slt` output are 0**

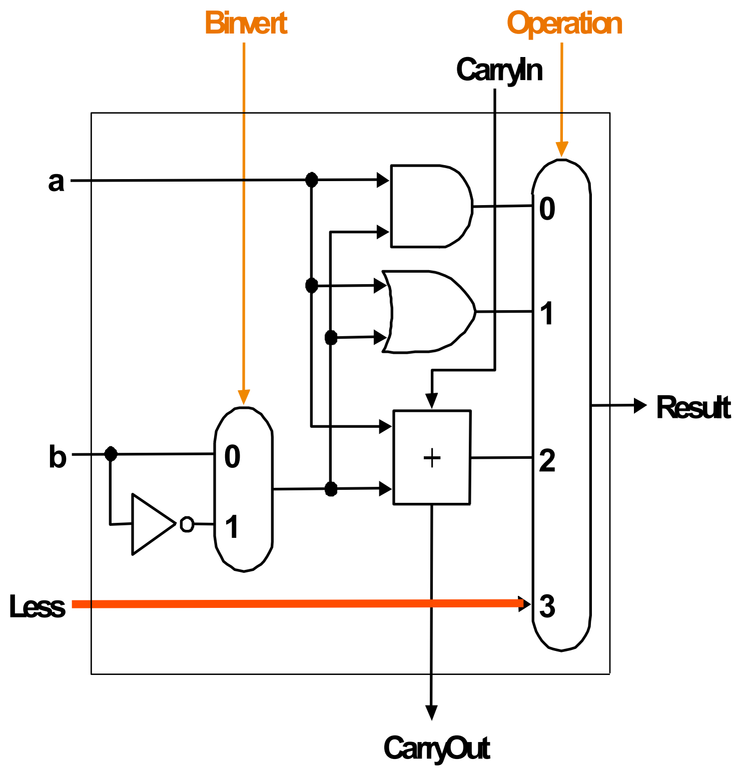
Tailoring the ALU for MIPS

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

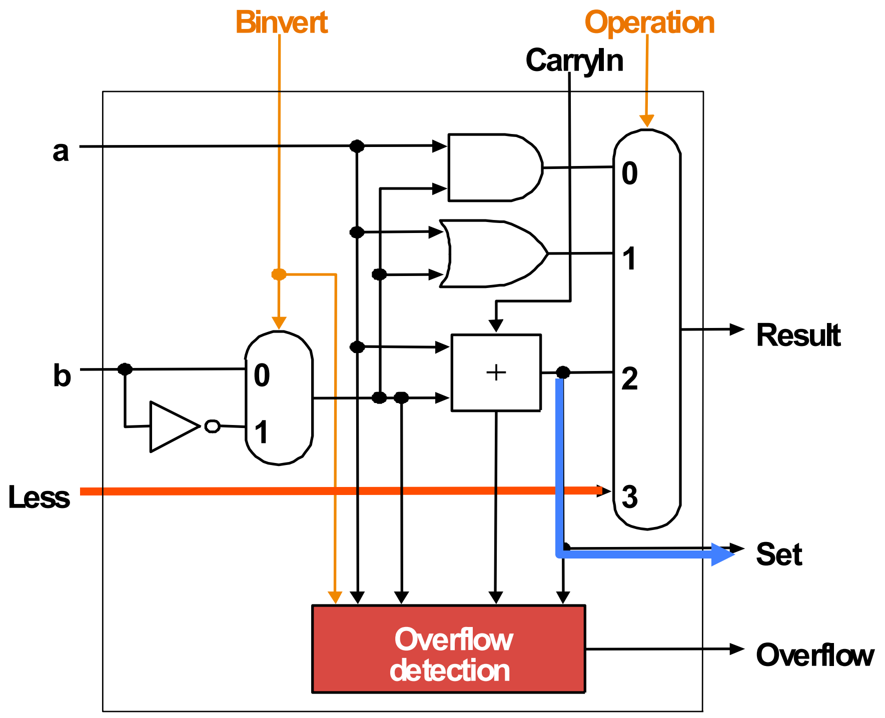
    `slt $t0, $s1, $s2`



Tailoring the ALU for MIPS (cont'd)



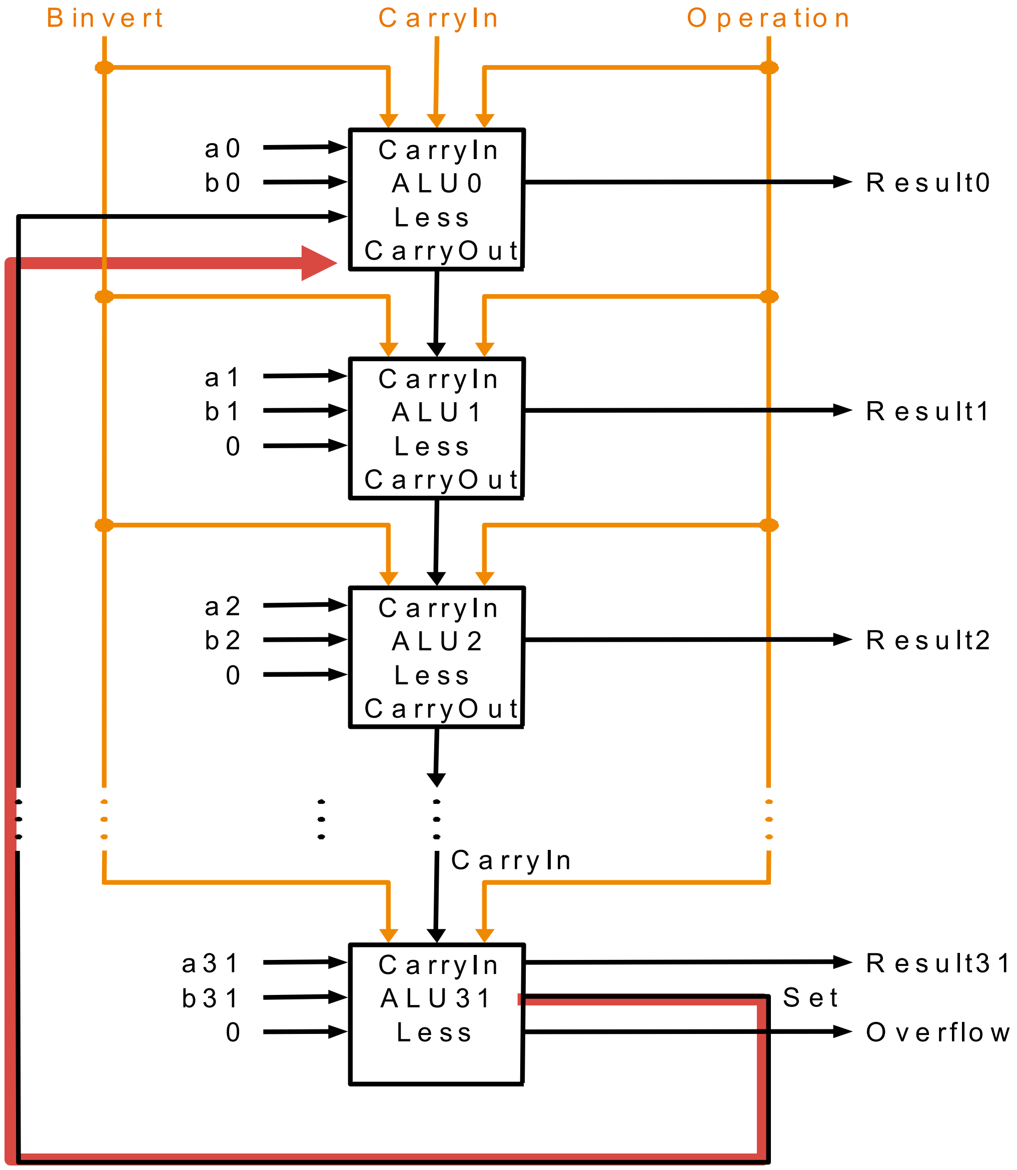
1-bit ALU for bits 0 to 30



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)

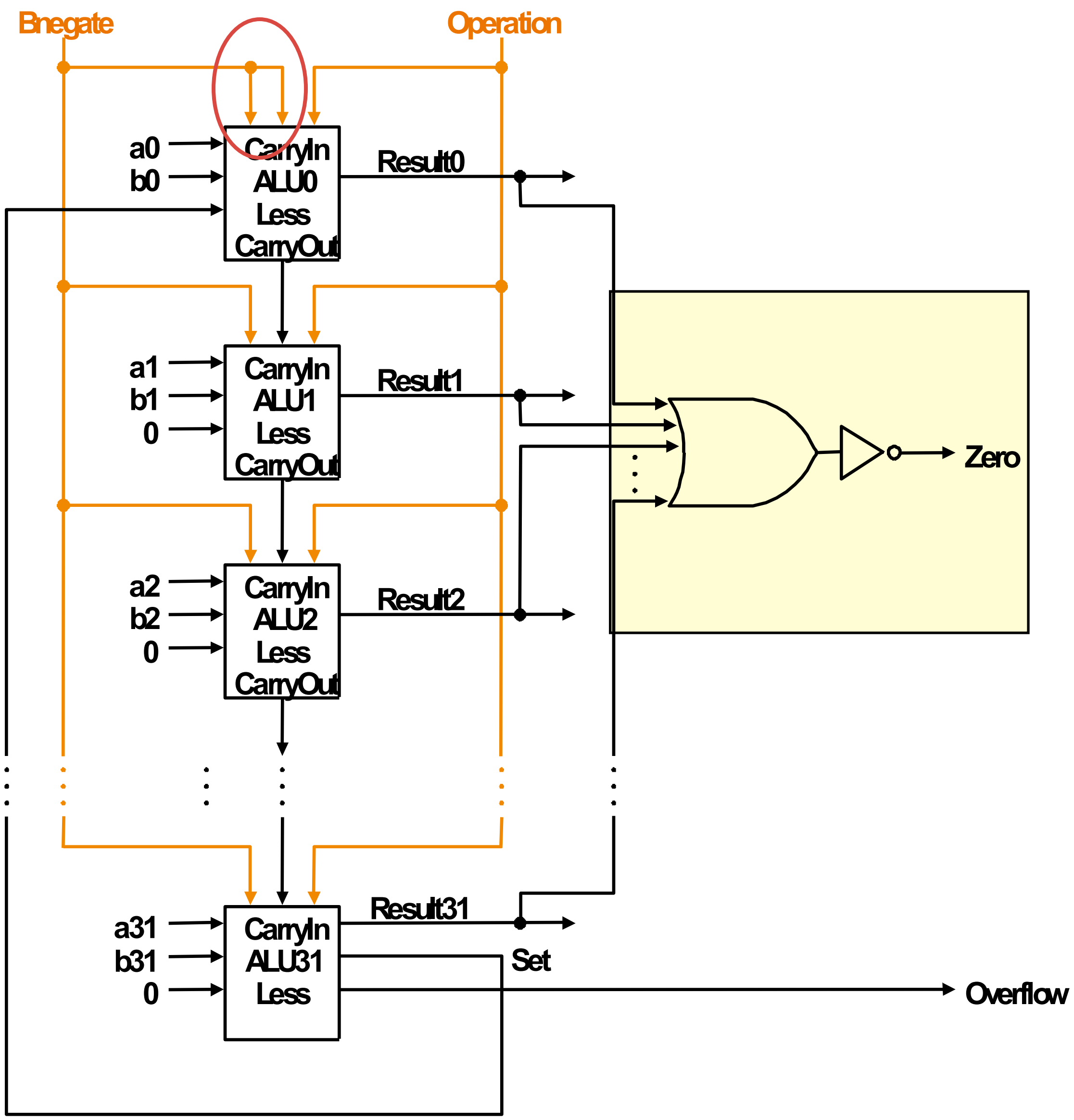


Tailoring the ALU for MIPS

- 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

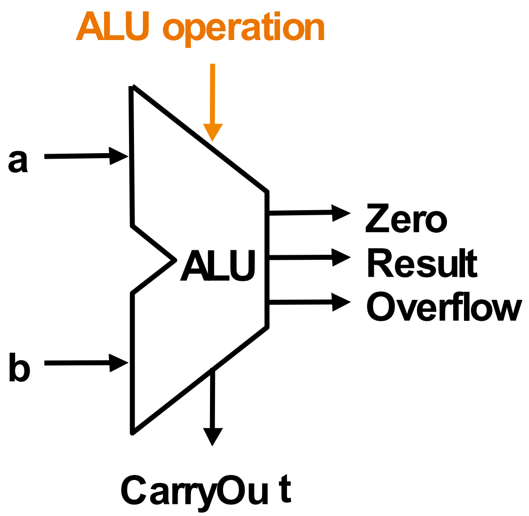
32-Bit ALU with (add, sub, AND, OR, slt)

- 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**:

$$\mathrm{CarryIn2} = (b1 \cdot \mathrm{CarryIn1}) + (a1 \cdot \mathrm{CarryIn1}) + (a1 \cdot b1)$$
$$\mathrm{CarryIn1} = (b0 \cdot \mathrm{CarryIn0}) + (a0 \cdot \mathrm{CarryIn0}) + (a0 \cdot b0)$$

  - Substituting the latter into the former, we have:

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

  - All other CarryIn bits can also be expressed using CarryIn0

Carry Lookahead

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

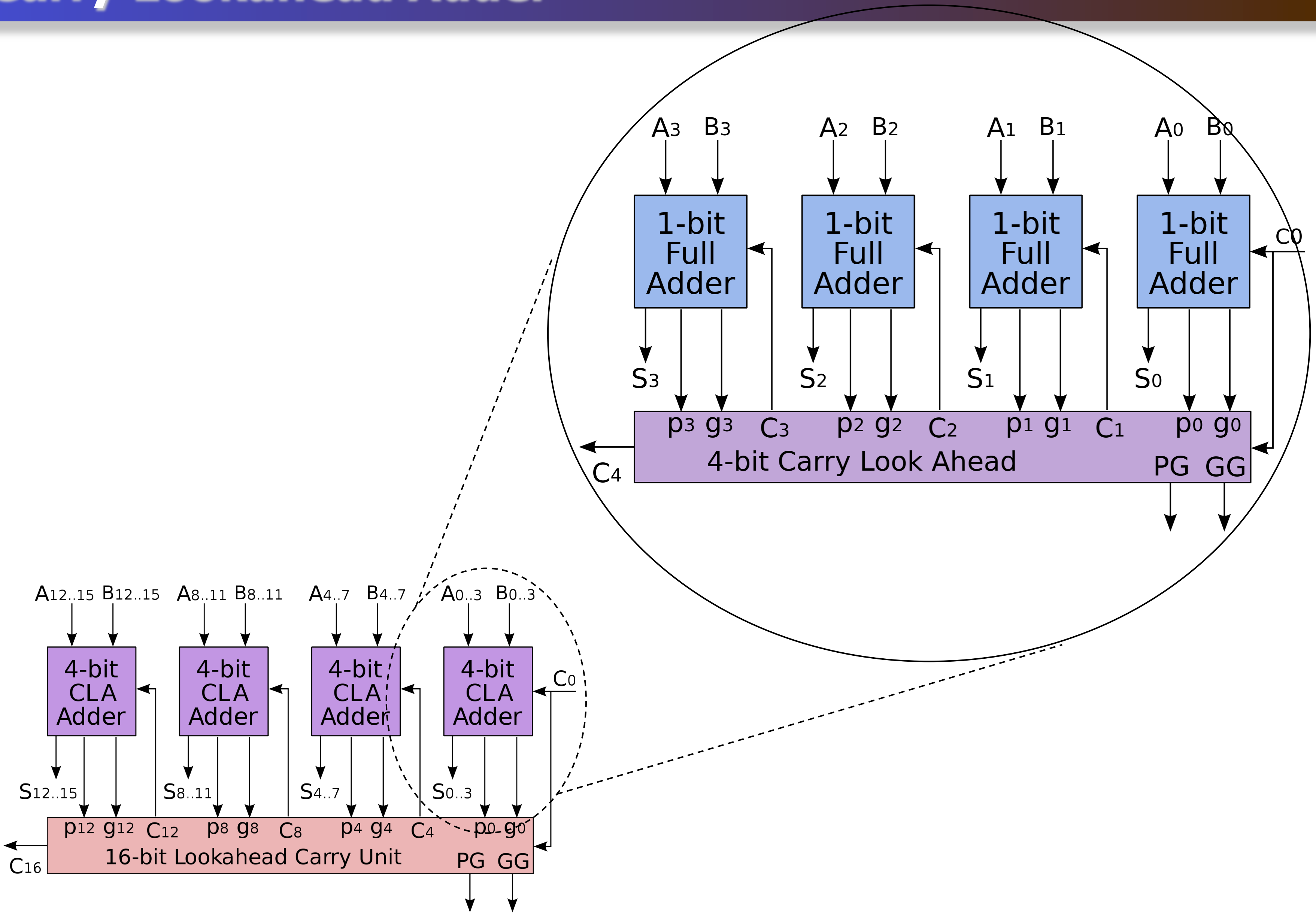
$$C_{i+1} = G_i + P_i \bullet C_i$$

- 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


A3 B3 A2 B2 A1 B1 A0 B0
1-bit Full Adder
C0
S3 S2 S1 S0
p3 g3 C3 p2 g2 C2 p1 g1 C1 p0 g0
C4
4-bit Carry Look Ahead
PG GG
A12..15 B12..15 A8..11 B8..11 A4..7 B4..7 A0..3 B0..3
4-bit CLA Adder
C0
S12..15 S8..11 S4..7 S0..3
p12 g12 C12 p8 g8 C8 p4 g4 C4 p0 g0
C16
16-bit Lookahead Carry Unit
PG GG