

Lecture 20

Arithmetic Logic Unit

- A major component of a processor.
- Arithmetic part does addition and subtraction of two numbers.
- Logical part calculates bitwise *AND* & *OR* of two values.

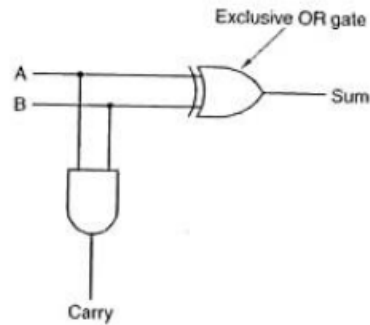
- How does it subtract? Uses adder.
 - $A - B = A + 2\text{'s complement of } B$
 $= A + ((1\text{'s comp. of } B) + 1)$
 - 1's complement of B is obtained using *NOT* gates.

- ALU has additional hardware for:
 - a) Overflow detection.
 - b) Zero detection.

- MIPS ALU has extra hardware for the `slt` (*set on less than*) instruction.

– **1-bit half adder:**

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

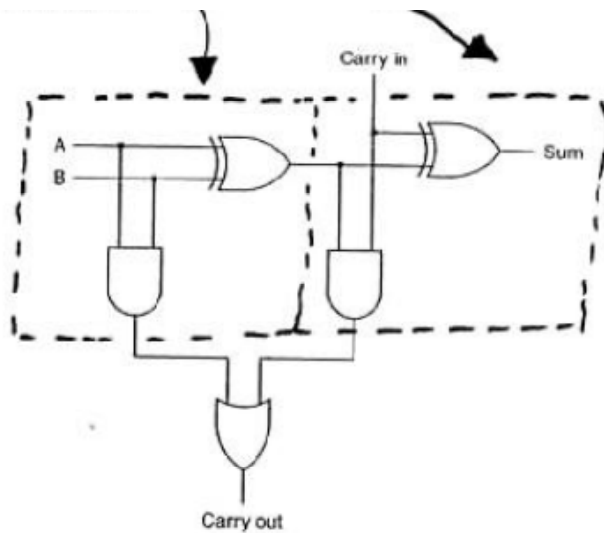


- Two inputs, two outputs
 - $\text{Sum} = A \oplus B = \bar{A} \cdot B + A \cdot \bar{B}$; $\text{carry} = A \cdot B$
- This is called a half adder because here is no carry-in.
- Carry-in is actually the carry-out of the previous addition
- We add the respective bits of two numbers, from right to left.

– **1-bit full adder:**

- Two half adders

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



- 3 inputs & 2 outputs.
- $\text{Sum} = A \oplus B \oplus C_{in}$ [carry-in]
- $C_{out} = \text{Majority}(A, B, C_{in})$.
 - When any two or more inputs are 1, we have C_{out} .
- Since we may generate C_{out} from either of the two half adders, we need an OR gate to combine these.

- The two units of circuitry below are combined to form a 1-bit ALU:

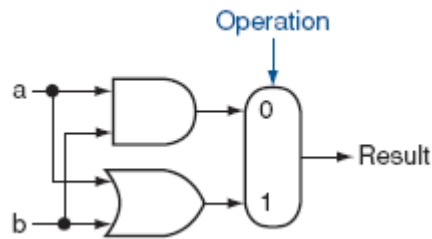


FIGURE C.5.1 The 1-bit logical unit for AND and OR.

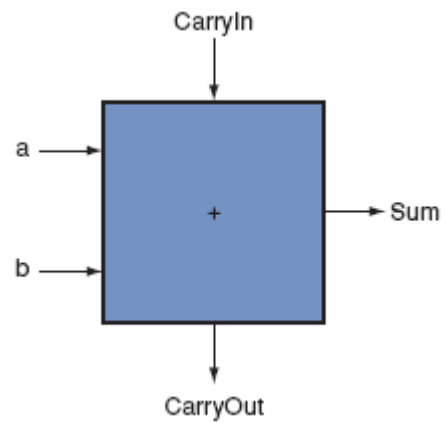


FIGURE C.5.2 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

- **1-bit ALU:** It does $a + b$, $a \text{ AND } b$ & $a \text{ OR } b$.

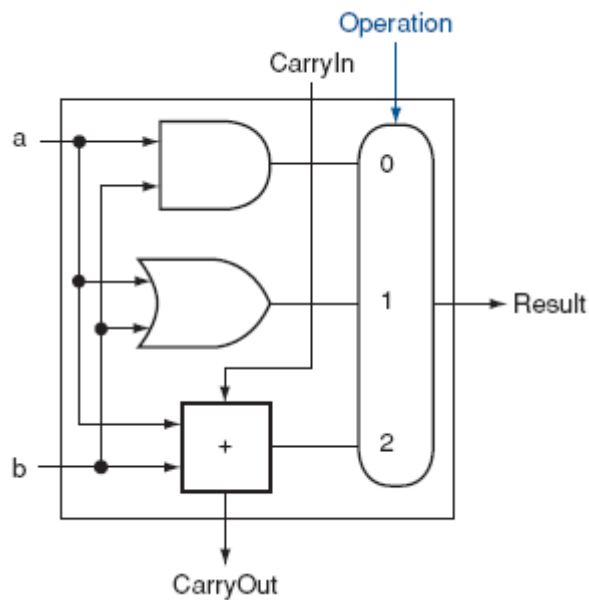


FIGURE C.5.6 A 1-bit ALU that performs AND, OR, and addition.

- This unit does not have hardware for subtraction ($a - b$) yet.

- A 32-bit ALU is build by connecting 32 of these 1-bit ALUs serially
 - This is the simplest way.
 - It is cheaper than the alternatives, but is slow.
 - The carry has to propagate from ALU0 to ALU31 – this takes time.
 - *AND/OR* operations are done in parallel. They are done quickly.

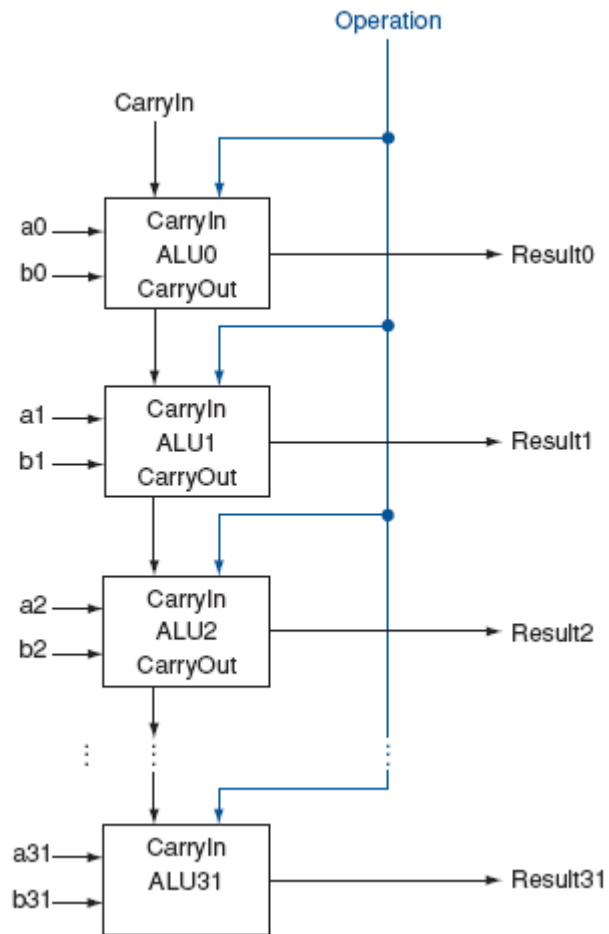


FIGURE C.5.7 A 32-bit ALU constructed from 32 1-bit ALUs. CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

- The 1-bit ALU below has an extra *NOT* gate for inverting *b*.

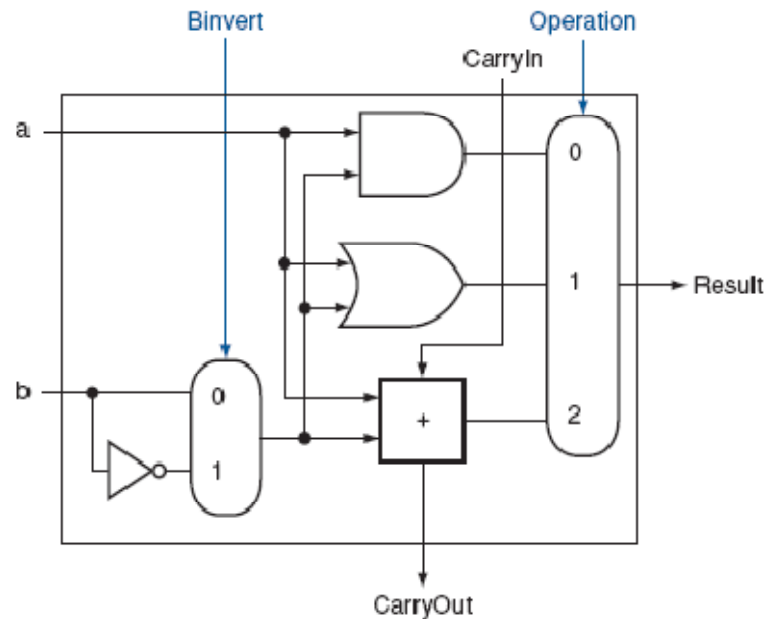


FIGURE C.5.8 A 1-bit ALU that performs AND, OR, and addition on *a* and *b* or *a* and $\neg b$. By selecting $\neg b$ ($B_{invert} = 1$) and setting $CarryIn$ to 1 in the least significant bit of the ALU, we get two's complement subtraction of *b* from *a* instead of addition of *b* to *a*.

- **Subtraction:** To do $a - b$, invert *b* using B_{invert} . This gives the 1's complement of *b*.
 - Add 1 using the carry-in of the LSB (least significant bit). Now we have the 2's complement of *b*.
 - Now *add* to obtain $a - b$.
 - Carry-in input of the 32-bit ALU is set to 1. This adds 1 at the least significant (*i.e.*, rightmost) position.
 - Thus, for subtraction, we set $C_{in} = 1$, B_{invert} to 1, and select the *add* operation.

- The following 1-bit ALU builds on the previous one by introducing a second *NOT* gate, for inverting *a* and thereby giving us *NOR* functionality.

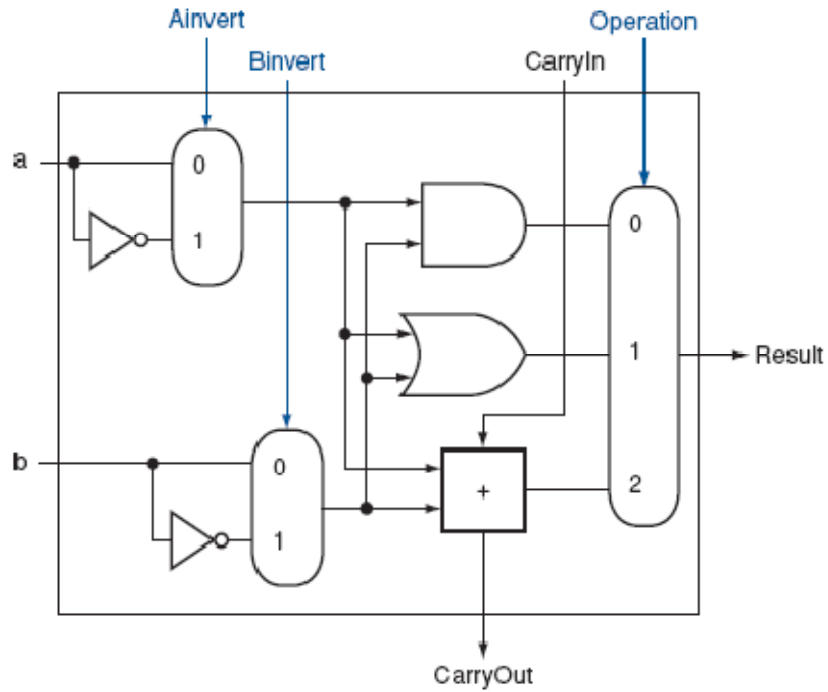


FIGURE C.5.9 A 1-bit ALU that performs AND, OR, and addition on *a* and *b* or $\neg a$ and $\neg b$. By selecting *a* (*Ainvert* = 1) and *b* (*Binvert* = 1), we get a NOR *b* instead of a AND *b*.

- **Set on less than:** `slt rd, rs, rt.`
 - Set destination register `rd` to 1 if `rs` is less than `rt`.
 - To set `rd` to 1, we must produce 1 at the output of the ALU.
 - Leftmost 31 bits must be all zeros.
 - Now `rs` is `a` and `rt` is `b`. If $a - b$ is negative then $rs < rt$.
 - Sign bit of the result will tell us if $rs < rt$.
 - We use a special 1-bit ALU at the MSB (most significant bit) – see right hand figure below.
 - Since we made a special 1-bit ALU for the MSB, it is a good idea to add to it circuitry for overflow detection.

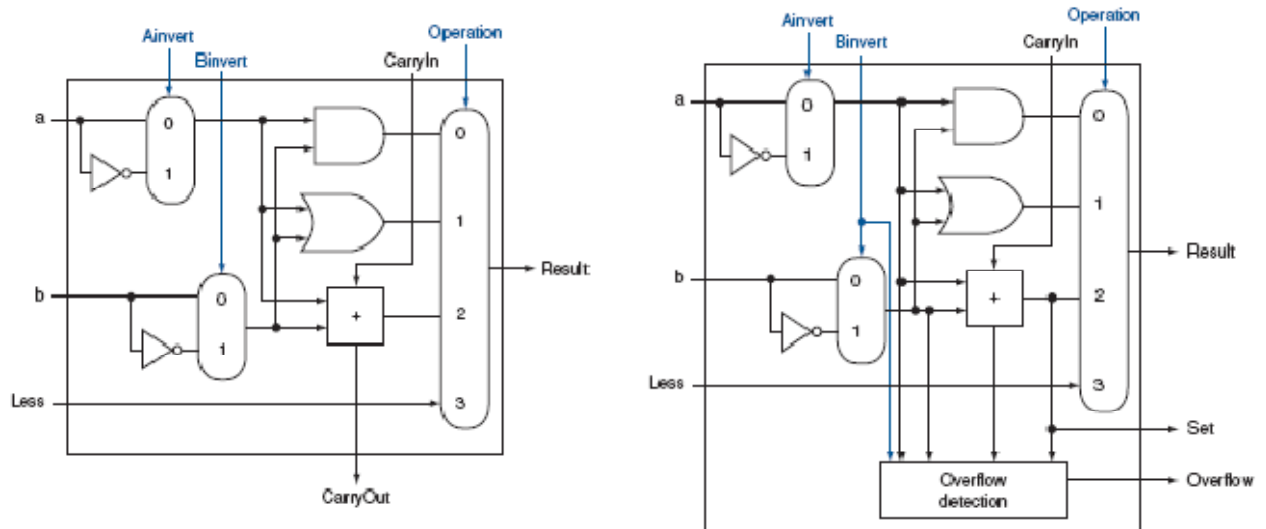


FIGURE C.5.10 (Left) A 1-bit ALU that performs AND, OR, and addition on `a` and `b` or `b`, and (right) a 1-bit ALU for the most significant bit. The left drawing includes a direct input that is connected to perform the set on less than operation (see Figure C.5.12 below); the right figure has a direct output from the adder for the less than comparison called Set.

- One extra input is added for `slt`.
- This is called *Less*.

- Note that the B_{negate} signal is fed in as the B_{invert} to each of the 32 1-bit ALUs.
 - It is also fed in as the C_{in} input to the LSB ALU. This gives us the extra 1 value to add to the 1's complement of b in order to obtain its 2's complement when doing subtraction $a - b$.
- This ALU has hardware for zero detection. If the output of the ALU is zero, then the output of the *NOT* gate will be 1.
 - This is used in the `beq` instruction (and the pseudoinstruction `beqz`, which translates to `beq`).

- Symbol for the ALU:

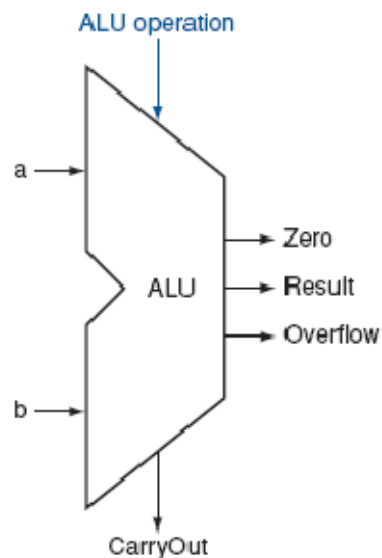
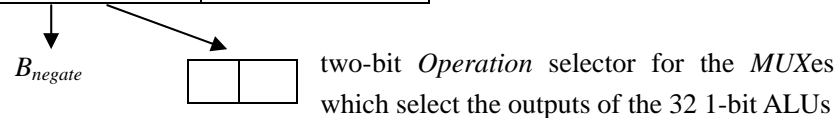


FIGURE C.5.14 The symbol commonly used to represent an ALU. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

- The same symbol is used for *adder* as well.
- *Less* is not shown because it is internal to the ALU.

- ALU control bits:

ALU control bits	Operation
0 00	<i>AND</i>
0 01	<i>OR</i>
0 10	add
1 10	subtract
1 11	set on less than



- We saw how *NOR* can be implemented using *a* inversion in the 1-bit ALUs. However, for simplification, we have not included this in the ALU control bits above.
 - Note that with *NOR* provided, we can use it to implement *NOT* as a pseudo-instruction. How?
- Each 1-bit ALU takes the two rightmost bits of our simplified 3-bit control code as the *Operation* selector to the $4 \rightarrow 1$ *MUX* which selects the output. The leftmost control code bit feeds in to the 1-bit ALUs as the B_{negate} / B_{invert} signal.
- The ALU we studied does not include additional logic functions:
 - *XOR*, for example, is provided in MIPS as an operation.
 - We would need extra gates for this in our ALU.
 - Instead of a $4 \rightarrow 1$ *MUX*, we would now need a $5 \rightarrow 1$ *MUX* in the 1-bit ALUs to select the output.