CSE220 – Adders & Arithmetic Logic Unit (ALU)

– 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 \textbf{slt} (set on less than) instruction.

1-bit half adder:

- Two inputs, two outputs
  - Sum = \( A \oplus B = \overline{A} \cdot B + A \cdot \overline{B} \); 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:

- One way to build a full adder is to connect two half adders

- 3 inputs & 2 outputs.
- 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.
Building ALUs
1-bit ALU: It does \( a + b \), \( a \text{ AND } b \) & \( a \text{ OR } b \).

- This unit does not have hardware for subtraction \((a - b)\) yet.
- **Note** that this unit, orders the operation control bits differently than done in lecture. The order is decided by the designer and the unit can be built to that specification.

- 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.

<table>
<thead>
<tr>
<th>Operation Control Bits {C1, C0}</th>
<th>ALU Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ( \text{AND} )</td>
<td>0 ( \text{AND} )</td>
</tr>
<tr>
<td>0 ( \text{OR} )</td>
<td>1 ( \text{OR} )</td>
</tr>
<tr>
<td>1 ( \text{ADD} )</td>
<td>0 ( \text{ADD} )</td>
</tr>
</tbody>
</table>
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 $b$.

A 1-bit ALU that performs AND, OR, and addition on $a$ and $b$ or $a$ and $\neg b$. By selecting $\neg b$ (Binvert = 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 $B_{\text{inv}}$.
  - 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 \textit{add} operation.

The following 1-bit ALU builds on the previous one by introducing a second \textit{NOT} gate, for inverting \( a \) and thereby giving us \textit{NOR} functionality.

A 1-bit ALU that performs \textit{AND}, \textit{OR}, and addition on \( a \) and \( b \) or \( \neg a \) and \( \neg b \). By selecting \( a \) (\( A_{invert} = 1 \)) and \( b \) (\( B_{invert} = 1 \)), we get a \textit{NOR} \( b \) instead of a \textit{AND} \( b \).

\( (B_{invert} = 1), \) we get a \textit{NOR} \( b \) instead of a \textit{AND} \( b \).

\textbf{Set on less than:} \( \text{slt \ rd, rs, rt} \).

The set on less than instruction performs the following operation:

- Set destination register \( \text{rd} \) to 1 if \( \text{rs} \) is less than \( \text{rt} \).
- To set \( \text{rd} \) to 1, we must produce 1 at the output of the ALU.
- Leftmost 31 bits must be all zeros.
- Now \( \text{rs} \) is \( a \) and \( \text{rt} \) is \( b \). If \( a - b \) is negative then \( \text{rs}<\text{rt} \).
- Sign bit of the result will tell us if \( \text{rs}<\text{rt} \).

The Harris/Harris book builds an n-bit ALU with this instruction in the following way:

When the ALU operations are performed on N-bits, you are essentially using gates which handle N-bits at a time. In the case of SLT, when when A-B is performed, the sign bit can be routed to the LSB of output Y and all higher bits set to 0. In this way, the Y value is 1 if \( A < B \) and 0 if \( A \geq B \).
The same concept can be applied when building ALU from 1-bit ALUs (or any ALU unit of < N bits).

- 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.

![Two figures of ALUs](image)

**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*. The *Less* bit is set to 0 for the upper (or leftmost) 31 bits.
- When executing `slt`, the *Result* outputs from the upper 31 bits of the adder are selected from the *Less* inputs (these *Less* inputs all have value 0, as noted above).
- The *Result* output from the rightmost bit (the LSB) is also selected from the *Less* input of that bit, but this value actually comes from the *Set* output of the special 1-bit ALU in the MSB. This *Set* value of the leftmost bit is fed back as the *Less* input of the rightmost bit.
FIGURE C.5.12 A 32-bit ALU constructed from the 31 copies of the 1-bit ALU in the left half of Figure C.5.10 and one 1-bit ALU in the right half of that figure. A zero detector has been added. The Less inputs are connected to 0 except for the least significant bit, which is connected to the Set output of the most significant bit. If the ALU performs $a - b$ and we select the input 3 in the multiplexor in Figure C.5.10, then Result = $0 \ldots 001$ if $a < b$, and Result = $0 \ldots 000$ otherwise.

- Note that the $B_{\text{negate}}$ signal is fed in as the $B_{\text{invert}}$ to each of the 32 1-bit ALUs.
  - It is also fed in as the $C_{\text{in}}$ 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 $\text{beq}$ instruction (and the pseudoinstruction $\text{beqz}$, which translates to $\text{beq}$).
Symbol for the ALU:

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

**ALU control bits:**

<table>
<thead>
<tr>
<th>ALU control bits</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td><em>AND</em></td>
</tr>
<tr>
<td>0 01</td>
<td><em>OR</em></td>
</tr>
<tr>
<td>0 10</td>
<td>add</td>
</tr>
<tr>
<td>1 10</td>
<td>subtract</td>
</tr>
<tr>
<td>1 11</td>
<td>set on less than</td>
</tr>
</tbody>
</table>

- We saw how *NOR* can be implemented using an 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 \(B_{\text{negate}} / B_{\text{invert}}\) 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.