Building an ALU

2-26-2003

Opening Discussion

- What did we talk about last class?
- Have you seen anything interesting in the news?

What is an ALU

- An ALU, or Arithmetic Logic Unit is one of the most basic types of structures that we like to put on silicon when building a chip. It basically handles the processing of integer type expressions.
- Note that it doesn't control the processing, it just does it when the right signals come in. The nature of having signals come in is probably worth taking a second to discuss.

A 1-bit ALU

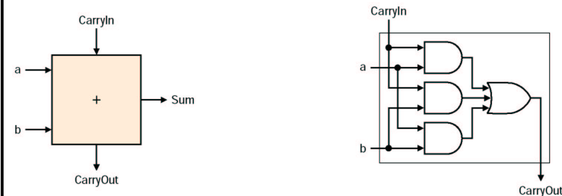
- For a MIPS machine, we really want a full 32-bit ALU, but if possible, we want to try to build it from many roughly identical pieces.
- Each of these pieces should be able to perform the types of operations we want on two single bit inputs.
- If all we needed was AND and OR then the simple circuit we saw last time would suffice.

Boolean Algebra

- As we saw last time, we are going to build all of our circuits from basic logic gates. Sometimes it is helpful to be able to write our logic expressions in a shorter format just to work through things.
- Boolean algebra helps for this. Basically you put your predicates (inputs) in as variables then use + for OR and * for AND. Note that it works fairly well for C-style boolean logic.

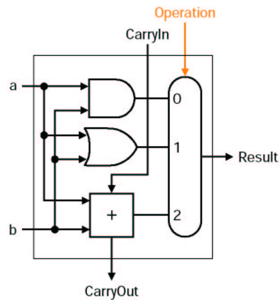
A 1-bit Adder

- We also want to have the operation of addition. Unlike AND and OR the bits aren't completely independent so we have a CarryIn and a CarryOut.



The Schematic 1-bit ALU

- This now gives us a 1-bit ALU that can do +, AND, and OR operations.
- Note that we have to add a new option to our multiplexor to get it to work.

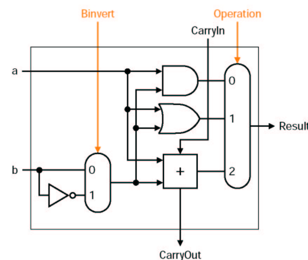


Making a 32-bit ALU

- The ideal for us would be that we could build a 32-bit ALU from 32 1-bit ALUs. For the most part, that is what we are going to do.
- In a simplistic way, chaining them together gives us what we want.
- What operations are we still lacking? What commands have we discussed that aren't here?

Including Subtraction

- Allow the second value to be the bitwise inverse and then add 1 with the first carry-in.
- This is why we use two's complement.



Implementing SLT

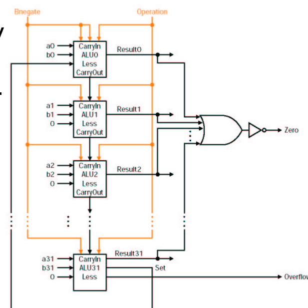
- Another operation that we are lacking is set on less than. We can implement this with a simple subtraction. If the first operand is less than the second, we get a negative difference which can be quickly detected by looking at the highest bit.
- Putting this in the chip requires just a bit of extra logic outside of the set of 1-bit ALUs. And a "LESS" input for zeros that are used for higher bits.

Checking Equality

- Similarly, we need to be able to check for equality for the beq and bne commands.
- The easiest way to do this is once again with subtraction. This time, we can OR together all the output bits. If the result is 0 then they were equal, otherwise they were not equal.

The 32-bit ALU

- This shows a very trimmed down version of the 32-bit ALU that we have just designed.



Minute Essay

- We skipped the part of the adder that actually did a+b for the 1-bit. This is equivalent to a logical XOR. Draw the diagram for an XOR built from AND, OR, and inverters.