**Arithmetic Operations**

- We will review the arithmetic building blocks we have previously used, and look at some new ones.
  - Addition
  - incrementer
  - Addition/subtraction
  - decremener
  - Comparison

---

**Binary Adder**

\[
F(A,B,C) = A \ XOR \ B \ XOR \ C \quad G = AB + AC + BC
\]

These equations look familiar. These define a **Binary Full Adder**:

\[
\begin{align*}
\text{Sum} &= A \ XOR \ B \ XOR \ Cin \\
\text{Cout} &= AB + Cin \times (A + B)
\end{align*}
\]

**Full Adder (FA)**

---

**4 Bit Ripple Carry Adder**

---
How did we get the Incrementer equations?

Full Adder equations:
- Sum = A xor B xor Cin
- Cout = AB or Cin A or Cin B
  = AB or Cin (A or B)

Let B = 0, Cin = 1 so that Sum = A + 1. Then equations simplify to:
- Sum = A xor 1 xor 0 = A xor 1 = A'
- Cout = 0 or 1 (A or 0) = A.

If we want an “En” input, then we want SUM = A if En=0, else SUM = A+1 if En = ‘1’. Filling in the above equations:
- SUM = A En' or A' En = A xor En
  Cout = A En

The “Cout” of one bit becomes the “En” signal for the next bit!!!!

A Subtractor

What is subtraction?

A - B = A + (-B)

How do you take the negative of a number? Depends on the sign representation (signed magnitude, 1s complement, 2s complement). Let’s assume 2’s complement since it is most common).

(-B) = B' + 1
So:
A - B = A + (-B) = A + B' + 1
Subtractor using an Adder

A[3:0] + B'[3:0] + Cin
SUM[3:0] = A - B

What if we want a block that can do both addition and subtraction?

Adder/Subtractor

Y[3:0] = (A-B) when (Sub = '1') else A+B;

Recall what a Comparator is...

Equality comparator.

AeqB = (A(0) xnor B(0)) and (A(1) xnor B(1)) and … etc.

A=B if A(0) = B(0) and A(1) = B(1) … and A(n-1)=B(n-1)

Recall that “xnor” function is ‘1’ if A=0, B=0 or A=1, B=1!

So AeqB is:

AeqB = (A(0) xnor B(0)) and (A(1) xnor B(1)) and … etc.
What is logic structure for equality comparator?

A(N-1)  B(N-1)  A(N-2)  B(N-2)  A0  B0

AeqB

AND Tree (will be multiple AND gates in tree arrangement)

Is there another Logic structure possible?

Compare “iteratively” from LSB to MSB

If (A(0) = B(0) then
if (A(1) = B(1) then
...
If (A(N-1) = B(N-1) then
AeqB = ‘1’ ; !!!!!!

A(N-1)  B(N-1)  A(1)  B(1)  A(0)  B(0)

AeqB

Signal from one bit block to next is “enable” for that block.

Iterative Comparator Structure

A(N-1)  B(N-1)  A(1)  B(1)  A(0)  B(0)

AeqB

An advantage to this structure is that the design for each bit is the same same, and we can extend it indefinitely. But it will be slow.
Two ways to do a Large AND function

Multi-level. # of levels depends on total number of inputs, number of inputs on each gate. A tree arrangement like this will take more gates, but will be fast.

Serial arrangement. Will take less gates, but will be slow.

What about “<” (less than), “>” (greater than)?

Full comparator.

The logic for AltB, AgtB depends on whether we are comparing signed numbers or not. We will assume unsigned numbers for now.

Logic for “AgtB” (unsigned)

Consider A > B, both N bit numbers, A[N-1:0], B[N-1:0]

If (A(N-1) = '1' and B(N-1) = '0') then
  AgtB = '1';
elsif ((A(N-1) = B(N-1)) and (A(N-2) = '1' and B(N-2)'0')) then
  AgtB = '1';
else...
  A=1xxx... B=00xxxx
  A=01xx... B=00xxxx
  A=11xx... B=10xxxx
Look at “bit(i)”. The enable signal from previous bit is A= B up until now. If this is ‘1’, then we need to do a comparison.

However, if “AgtB” is already true, then we don’t need to do comparison and can skip this comparison!
Iterative Implementation of AgtB

\[ en_o = (A \oplus B) \text{ and } en_i; \]

If \( \text{skip}_i = '1' \) then
\[ \text{skip}_o = '1'; \]
else
\[ \text{skip}_o = en_i \text{ and } (A \text{ and } B'); \]
end if;

Logic Implementation

\[ en_o = (A \oplus B) \text{ and } en_i; \]

If \( \text{skip}_i = '1' \) then
\[ \text{skip}_o = '1'; \]
else
\[ \text{skip}_o = en_i \text{ and } (A \text{ and } B'); \]
end if;

Can use a K-map to simplify this logic.

The \( \text{skip}_o \) of the LAST bit is the AgtB signal!
The \( en_o \) of the LAST bit is the AeqB signal!
What about AltB???
\[ AltB = \text{AgtB}' \text{ and } \text{AeqB}'; \]

Final Comparator

\[ AltB \Leftarrow \text{not (AeqB) and not (AgtB)}; \]
For $i$'th bit:

- $e_{n}(i) = (A(i) \text{ xor } B(i)) \text{ and } e_{n}(i+1)$
- If $(\text{skip}(i+1) = '1')$ then
  - $\text{skip}(i) = '1'$
  - else
    - $\text{skip}(i) = e_{n}(i) \text{ and } (A(i) \text{ and } B'(i))$

8 Bit Comparator

architecture a of comp is

begin
  sgn_en, skip : std_logic_vector(8 downto 0);

  begin
    aeqb <= en(0);
    agtb <= skip(0);
    altb <= ('1' when (a = b) else '0');
  process (a,b)
  begin
    en(8) <= '1';
    skip(8) <= '0';
    for i in 7 downto 0 loop
      en(i) <= '1';
      skip(i) <= '0';
    end loop;
    if (skip(i+1) == '1') then
      skip(i) <= '1';
    else
      skip(i) <= '0';
    end if;
    end process;
    end a;

Alternate VHDL specification

architecture a of compa is

begin
  aeqb <= '1' when (a = b) else '0';
  agtb <= '1' when (a > b) else '0';
  altb <= '1' when (a < b) else '0';

end a;

Synthesis tool will pick a logic implementation for implementation of "$\approx$, ">", "><" based on user constraints such as propagation delay.
Glitches in synthesized logic. Ok as long as we are using FFs to latch result.