Carrying, Multiplication, and Division

2-28-2003

Opening Discussion

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

Abstracting Carrying

- We saw last time that just waiting for the CarryOut to propagate through all our 1-bit adders was probably too slow.
- In order to find a faster way of doing it, we need to try to abstract the ideas of carrying. We do this by trying to pull apart what actually causes us to carry to see if we can put it in a simpler format.

First Level

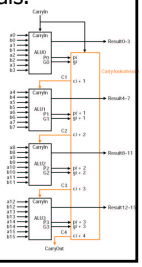
- We can recognize that there are basically two situations that cause a bit to carry over. If both bits are on then it "generates" a carry and the carry will always happen. Alternately, if one bit is on we will "propagate" a carry if a carry comes in.

$$g_i = a_i \cdot b_i$$
$$p_i = a_i + b_i$$
$$c_{i+1} = g_i + p_i \cdot c_i$$
$$c_1 = g_0 + (p_0 \cdot c_0)$$
$$c_2 = g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot c_0)$$
$$c_3 = g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot c_0)$$
$$c_4 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) + (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0)$$

Second Level

- We can further optimize this by grouping the ALU into 4 bit groups that use the formula on the previous slide and defining "super" signals.

$$P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$
$$P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4$$
$$P_2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8$$
$$P_3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}$$
$$G_0 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0)$$
$$G_1 = g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4)$$
$$G_2 = g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8)$$
$$G_3 = g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12})$$
$$C_1 = G_0 + (P_0 \cdot c_0)$$
$$C_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot c_0)$$
$$C_3 = G_2 + (P_2 \cdot G_1) + (P_2 \cdot P_1 \cdot G_0) + (P_2 \cdot P_1 \cdot P_0 \cdot c_0)$$
$$C_4 = G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0)$$

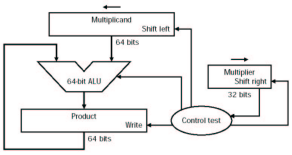


Multiplication

- Just as with addition, our best path for understanding binary multiplication is to make an analogy with "long" decimal multiplication.
- The two are done identically, but the binary format is easier because each digit can only be a 1 or a 0, two things that are very easy to multiply by.

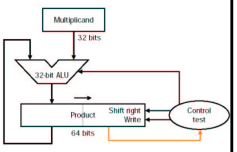
Simple Algorithm

- We can implement our basic idea in an algorithm to get the following circuit.
- Note the lengths of both the product and the multiplicand registers and the ALU.



Refinements

- We can optimize this by noticing that we can limit ourselves to mostly 32-bit components by adding in the the upper half of the product register and downshifting at each step.
- A third version can be created by putting the multiplier in the 64-bit Product register.



Booth's Algorithm

- Does additions and subtractions, but only at the "edges" of groups of ones.
- The product now must store a signed value and the right shift has to preserve the sign.
- The shifts that preserve the sign a arithmetic shifts instead of logical shifts.
- This algorithm works equally well with negative numbers.

Multiply Instructions

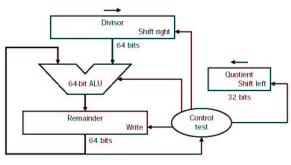
- mult and multu perform multiplication and store the result in the special hi and low. You can then use mflo and mfhi to move those values into general purpose registers.
- To find out if there was an overflow from the 32-bit low register you have to check the high register.

Division

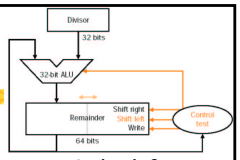
- Let's review the way that we do long division, and try to apply it to binary numbers.
- As with multiplication, we will use this as a foundation for developing our approach to building the circuit. We will then look at that approach and see how it can be optimized.

Simple Algorithm

- Again we can put our simple approach into a circuit and again we face the problem of needing a 64-bit ALU and registers.



Refinements



- We improve this by shifting the remainder left instead of the divisor right and now can use a 32-bit divisor and ALU. Also we shift before subtracting to remove one step from the arithmetic.
- Shifting quotient bits into the remainder register removes the need for a quotient register. Ends by shifting back the remainder half.

Minute Essay

- How well are you understanding the way the ALU works?
- Remember that the midterm is a week from today.

To Nick from Renuka:

Clue #1: There is no desk in the back of coates.

Your Mission: To collect all the clues and have a wonderful anniversary.