CS61c: Combinational Logic Blocks

J. Wawrzynek

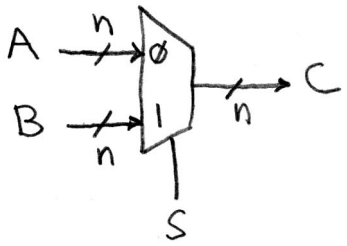
October 12, 2007

1 Introduction

Last time we saw how to represent and design combinational logic blocks. In this section we will study a few special logic blocks; data multiplexors, an arithmetic/logic unit, and a adder/subtractor circuit. These blocks will help later with implementation of the MIPS processor.

2 Data Multiplexors

A data multiplexor, commonly called a *mux*, is a circuit that selects its output value from a set of input values. For instance, consider the mux circuit shown below:



This mux has two n-bit inputs, A and B, and an n-bit output, C. Additionally, it has a special control signal labeled s, for *select*. The s signal is used to control which of the two input values is directed to the output. Specifically, the function of this mux can be described with these two rules:

when s=0, C=A
when s=1, C=B

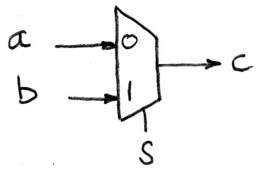
To remind us of which value of s corresponds to which input, within the mux symbol we commonly label each input with its corresponding s value.

This particular mux example is a called a *2-to-1, n-bit wide* multiplexor. It is *2-to-1* because it takes two data inputs and outputs one of them. It is *n-bit wide* because all data signals are n-bits in width. Notice, however, that the s signal is a single bit wide. As we will see in a later lecture, this particular mux finds common use within the design of a microprocessor, such as the MIPS. Whenever a circuit must choose data from multiple sources, a mux is used.

Let's take a look inside the n-bit wide 2-to-1 mux. The simplest way to understand it is to consider it to be a collection of *n instances of 1-bit wide 2-to-1* muxes. Each instance of the 1-bit wide mux is

responsible for generationg one bit of C from one bit of A and one bit of B. All n instances share the same control signal, s.

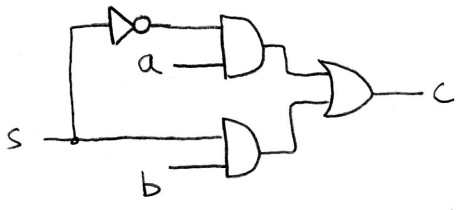
A 1-bit wide 2-to-1 mux is shown below along with its truth-table.



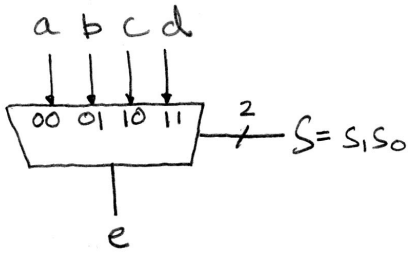
s	ab	c
0	00	0
0	01	0
0	10	1
0	11	1
1	00	0
1	01	1
1	10	0
1	11	1

To come up with the logic equation and the associated gate-level circuit diagram we can apply the technique that we studied last lecture. We write the sum-of-products canonical form and simplify through algebraic manipulation. The algebraic steps and final result is shown below. Intuitively this result makes sense; When the control input, s, is a 0, the right hand side of the equation reduces to a, and when it is a 1, the expression reduces to b.

$$\begin{aligned} c &= \overline{s}a\overline{b} + \overline{s}ab + s\overline{a}b + sab \\ &= \overline{s}(a\overline{b} + ab) + s(\overline{a}b + ab) \\ &= \overline{s}(a(\overline{b} + b)) + s((\overline{a} + a)b) \\ &= \overline{s}(a(1) + s((1)b) \\ &= \overline{s}a + sb \end{aligned}$$



Often times we find the need to extend the number of data inputs of a multiplexor. For instance consider a *4-to-1* multiplexor:

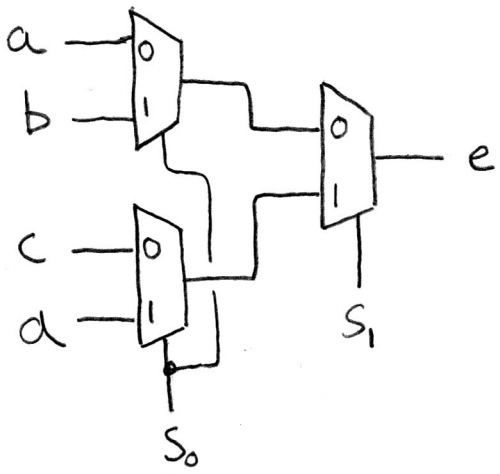


when S=00, e=a
when S=01, e=b
when S=10, e=c
when S=11, e=d

How would we come up with the circuit for this mux? We could start by enumerating the truth-table—in this case the function has 4 single bit data inputs and one 2-bit wide control input, for a total of 6 single bit inputs. The truth-table would have 2^6, or 64 rows. Certainly, a feasible approach. If we where to do this, we would end up with the following logic equation:

$$e = \overline{s_1} \cdot \overline{s_0}a + \overline{s_1}s_0b + s_1\overline{s_0}c + s_1s_0d$$

Another way to design the circuit is to base it on the hierarchical nature of multiplexing. We can build a 4-to-1 mux from three 2-to-1 muxes as shown below:



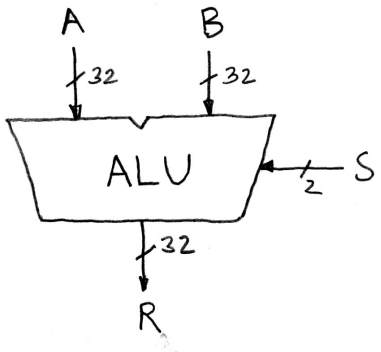
The first layer of muxes uses the s_0 input to narrow the four inputs down to two, then the second layer uses s_1 to choose the final output.

3 An Arithmetic and Logic Unit (ALU)

Most processor implementations include a special combinational logic block called an arithmetic and logic unit (ALU). In the MIPS, the ALU is used to compute the result in the R-type instructions, such as, add, sub, and, or addi, ori, etc.

We are going to consider the design of a simpler version of the ALU than the one in the MIPS. Ours will include only for basic functions, ADD, SUB, bitwise AND, and bitwise OR. The ALU is organized

as a CL block with two 32-bit wide data inputs, A and B, a 32-bit wide data output, R, and a 2-bit wide control input, S. S is used to indicate which of the four possible operations the ALU is to perform. Typically, the S input will be controlled by the processor based on the opcode (actually the "func" field on the MIPS) of the instruction currently executing on the processor.



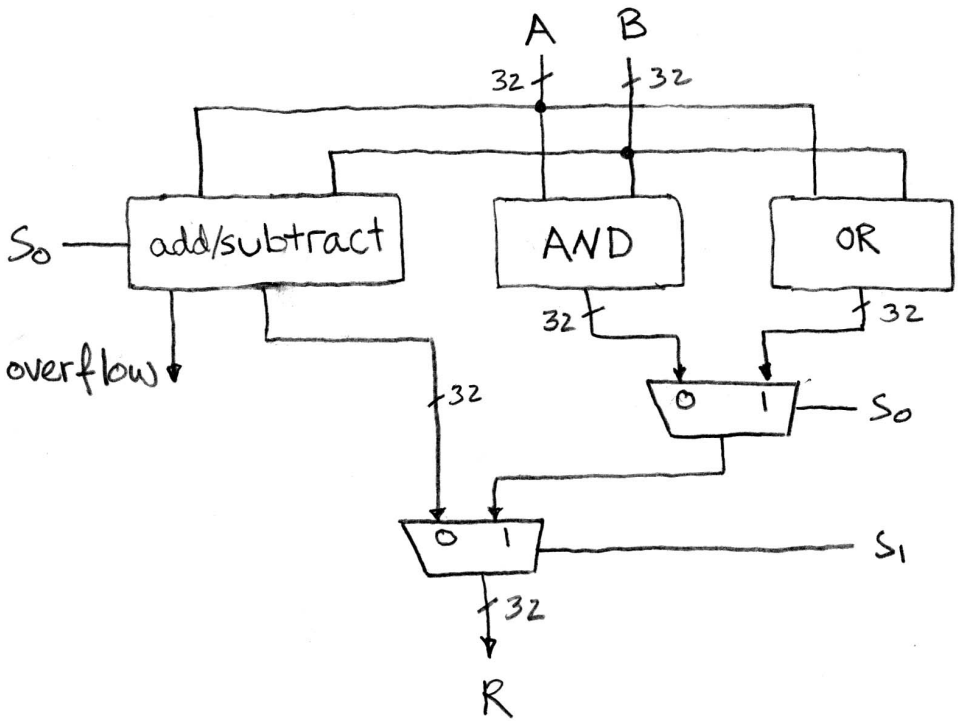
when S=00, R=A+B
when S=01, R=A-B
when S=10, R=A AND B
when S=11, R=A OR B

A common way to implement an ALU is to provide an instance of a CL block for each of the possible ALU functions. The inputs, A and B, get distributed to all the blocks, and the output of the proper block is selected with a mux. In this configuration, every function of the ALU is computed internally to the ALU on every cycle, but only one of the results is sent to the output.

For our simple ALU will need an adder block, a subtractor block, an AND block, and an OR block. Each of these blocks will take two 32-bit inputs and produce a 32-bit output. As you might suspect, a subtractor circuit is very similar to an adder circuit. Therefore, instead of providing a separate adder and subtractor, we are going to provide a single circuit that is capable of either operation. We will see in the next section how to design such a circuit. For now, assume that we have an add/subtract circuit with a special control input, labeled SUB, which when set to 1 forces the circuit to perform the subtraction A–B. When SUB=0 the circuit performs addition.

Our add/subtract block has a special output, labeled "overflow". Upon performing an addition or subtraction, this output will be a 1 if the result is too large to fit in 32 bits. Overflow can occur when adding a pair of negative numbers or when adding a pair of positive numbers.

The internal design of our simple ALU is shown below. The high bit of S, s_1, is used to choose between the add/subtract unit and the output of the mux that chooses between the AND and OR blocks. The choice of AND and OR is controlled by the low bit of S, s_0. The low bit of S is also used to specify the operation of the add/subtract block—1 for sub, and 0 for add. As you can see in the figure, A and B are distributed to all three blocks.



3.1 Implementing the Internal Blocks

The logical operations as defined by the MIPS instruction set are *bitwise* operations. That means that in the case of the AND, the resultant bit r_i is generated as a_i AND b_i. The circuit to perform this operation is simply a collection of 32 AND gates. Each AND gate is responsible for one of the 32 resultant bits. Similarly, the OR block is a collection of 32 OR gates.

The add/subtract block is a significantly more complex block than the AND or OR block. Its design is the subject of the next section.

4 Adder/Subtractor Design

We will start out by studying the design of an adder circuit, then later augment it to also perform subtraction.

One obvious method for arriving at the detailed gate-level circuit for the adder would be to follow the procedure we learned in the previous lecture. We would start with a truth-table, then write the canonical Boolean equation, then simplify. Unfortunately, that technique is only effective for very narrow adders, because the size of the truth-table is too large for wider adders. Therefore, we need to find a way to break the design up into smaller more manageable pieces. We will design the smaller pieces individually, then wire them together to create the entire wide adder.

Long-hand addition gives us a good guide on how to break up the adder into pieces. When we add by hand, we begin by adding together the two least significant bits of A and B, a_0 and b_0, respectively. From that addition, we generate a result bit, s_0, and possibly a carry bit. Then we move over to the next

more significant column, adding the carry from the previous stage along with the next two bits of A and B, a_1 and b_1, respectively. We then continue the process by moving left one column at a time until we have finished all n columns.

The truth-table representing the value of least significant sum bit, s_0, and the carry out of the least significant stage, c_1, is shown below. (We name a carry bit according to the stage to which it is an input. Therefore the carry into stage 1 is called c_1.) By inspection of the true-table we can simply write the logic equations for this stage.

$$\begin{array}{cccc|c} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline & s_3 & s_2 & s_1 & s_0 \end{array}$$

a_0	b_0	s_0	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$s_0 = a_0 \text{ XOR } b_0$$
$$c_1 = a_0 \text{ AND } b_0$$

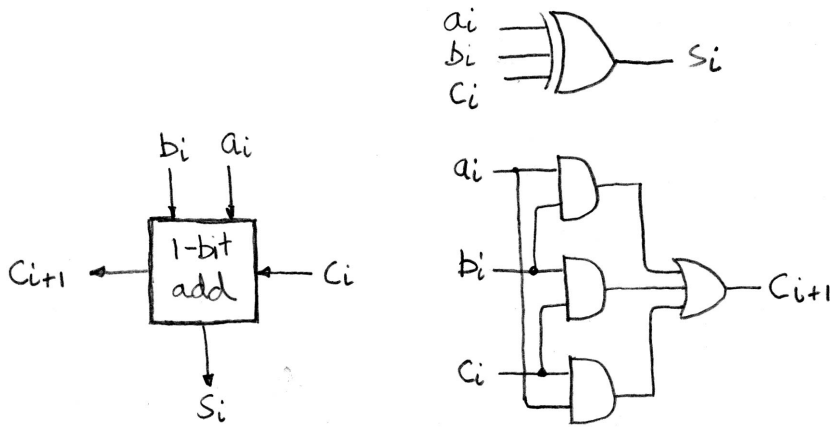
Here are the details for the next significant stage and all subsequent stages. The truth-table now has 8 rows, because there are three inputs into these stages, a bit from each of A and B, along with the carry-out from the previous stage. Even though the true-table is larger, the logic equations are still simple to write. By carefully inspecting the truth-table you will notice that the sum output is the exclusive-or of the three inputs—it is a 1 when the number of 1's in the input is odd. The carry-out function is the majority function—it is a 1 when the number of 1's in its input is greater than the number of 0's.

$$\begin{array}{cccc|c} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline & s_3 & s_2 & s_1 & s_0 \end{array}$$

a_i	b_i	c_i	s_i	c_{i+1}
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

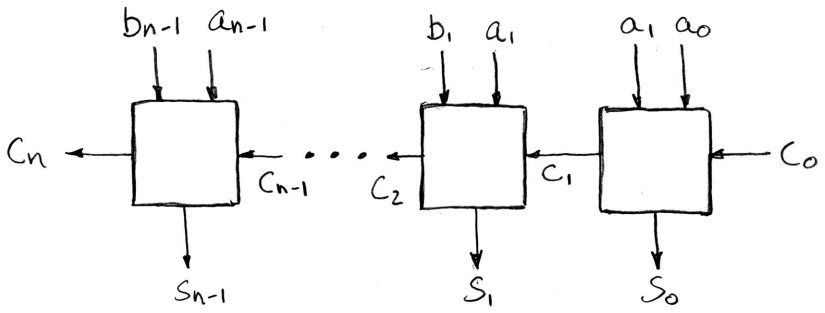
$$\begin{aligned} s_i &= \text{XOR}(a_i, b_i, c_i) \\ c_{i+1} &= \text{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i \end{aligned}$$

We will encapsulate the operations of one stage, or column, of the add operation into a small block. You can think of this block as a *1-bit adder*. It's official name is a *full-adder cell*, but most people confuse this with an n-bit adder, which it certainly is not. The symbol we will use for this 1-bit adder, along with the gate-level circuit diagrams for its internals are shown below:



We will not bother creating a special block for the first column of the adder, because anything it can do the 1-bit adder block can do as well. We just need to figure out where to attach the carry-in to the first column. More on this later.

The next step in the design of our adder circuit is to wire together a collection of our 1-bit adders to create an n-bit adder. All we need to do is to wire the carry-out output of one stage into the carry-in input of the next, from least significant to most, as shown below:



Inputs are applied at the top and after some delay, associated with the delay through the logic gates for the blocks, and the delay of the carry from stage to stage, the final result appears at the bottom.

The carry-out from the most significant stage, c_n, can be used as the n^{th}+1 bit of the result (remember, adding two n-bit numbers could result in a (n+1)-bit result. However, in most uses for an adder, we must generate an n bit result. For instance, in the MIPS, because the registers can only store 32 bits, the result of the add instruction must be a 32-bit number.

As discussed earlier, an *overflow* output must be generated to indicate when the result cannot fit into n bits. We can make the following observations about adds in this circuit. Remember, the most significant stage is the stage associated with the sign bit. If there was a carry into the most significant stage, but no carry out of that stage, then A and B where both positive and the result of the addition overflowed, erroneously generating a 1 in the sign bit position. If there was a carry out of the most significant stage and no carry into that stage, then A and B where both negative and the result of addition overflowed. In all other cases the value of the carry in to the most significant stage matches the

carry out, then there was no overflow. Based on these observations, we can design a simple circuit that generates the overflow output signal by comparing the carry-in and carry-out of stage n-1. A simple circuit that indicates when two signals are different in value is the XOR gate. Therefore:

overflow $= c_n$ XOR c_{n-1}

Next we need to decide what to do with the carry in to the least significant stage, c_0. Clearly for the adder to produce the correct result this input must be connected to a source of logic 0 (GND in the circuit). Now we will find something more interesting to do with c_0.

4.1 Subtractor Design

We mentioned earlier that addition and subtraction are closely related, and therefore we would expect that their respective circuits are similar and could serve dual both purposes. We know that if we want to compute $A - B$ we can compute $A + (-B)$ instead. $-B$ is the 2's complement of B. We also know that the 2's complement of B, is defined as $\overline{B} + 1$, where $\overline{B}$ is the 1's complement, or the inversion, of B. Therefore, $A - B = A + \overline{B} + 1$. This is a nice result as it gives us a very simply way to augment our adder circuit to be a combined adder and subtractor. We don't really need a second adder to perform the $+1$, because we have the unused input c_0. In the case of subtraction, if we connect c_0 to 1 instead of 0, the circuit will add an extra 1 in with the least significant column, achieving the extra $+1$.

The other augmentation needed for subtraction is to invert all the bits of B before feeding them into the top of the adder. Of course, we want the inversion to be conditional on which operation we wish the circuit to perform. Once again, we turn to our old friend the XOR gate. If looked at the right way, an XOR gate is an *conditional inverter*. If one of the inputs to an XOR gate is a 0, then the output takes on the value of the other input. On the other hand, if one of the inputs to an XOR gate is a 1, then the other input passes through inverted. We can conditionally invert B, by passing each of its bits through an XOR gate on the way to the input of the adder. The augmented adder design is shown below. When the input SUB is 1 the block performs subtraction, when SUB=0 the block performs addition. Also shown is the extra XOR gate for generating the overflow signal.