Kevin Gomes
Dr. Baudet
CSC 411
21 November 2017
Assignment #7

Reference point:

| F1 | F0 | Operation |
|---|---|---|
| 0 | 0 | X OR Y = Z |
| 0 | 1 | X AND Y = Z |
| 1 | 0 | NOT Y = Z |
| 1 | 1 | X XOR Y = Z |

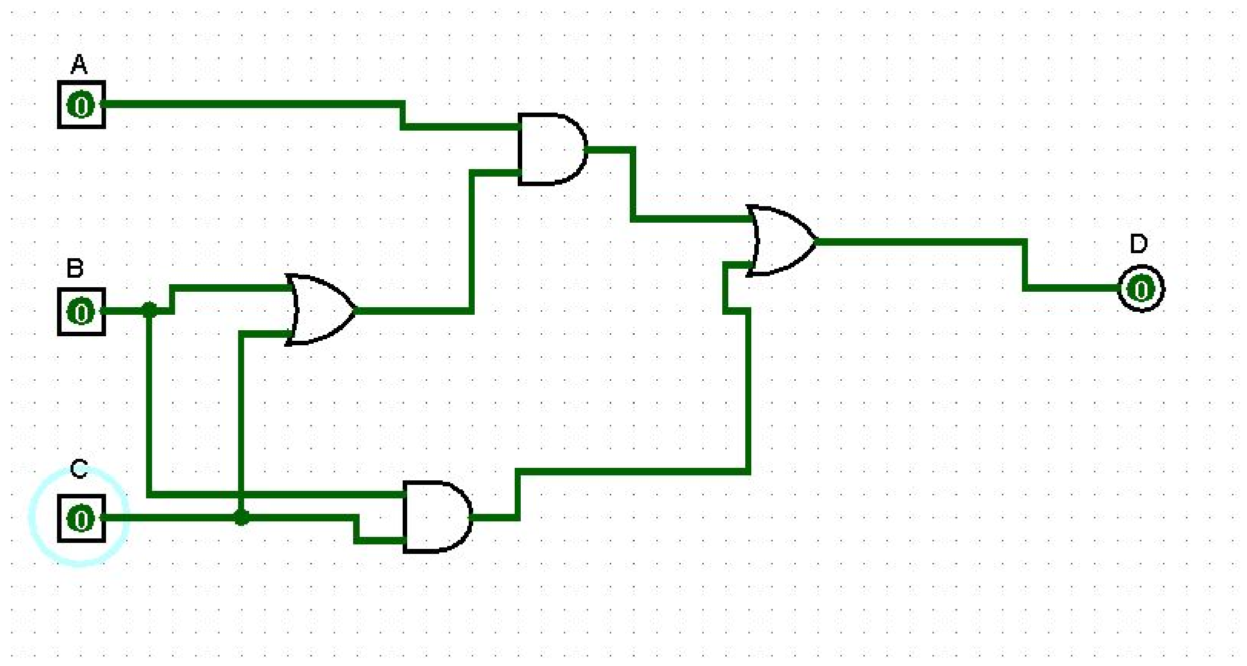
Memory Addresses:

| X1 | X0 | Address points to... |
|---|---|---|
| 0 | 0 | A |
| 0 | 1 | B |
| 1 | 0 | C |
| 1 | 1 | D |

Order of instruction:
F1F0 X1X0 Y1Y0 Z1Z0
NOTE: All comments will be denoted by a **semi-colon ;** at the start of the line. If we use a block comment, we will use a square bracket to denote the start after the semi-colon, and end it with a square bracket followed by semi-colon. The last comment will contain the complete code to make it easier to read. The following are examples of how I will comment.

```
;[ this
        Is a block comment.
Still in the comment!
];
;This is a regular comment. It is outside of the block comment.
;       Another regular comment.
```

Step 1: Implement, using basic 2-input gates, a 3-bit majority function.
**NOTE:** I realize we could instead make it simply AB + ( (A XOR B) AND C). But I wanted to use basic logic gates for this one. I've grown fond of them :)



Step 2: Implement that same 3-bit majority function using binary operations with the Puny Computer. Comment it. Can destroy contents of any register.
00 01 10 11

;[^B + C = D    This makes register D store the value of B OR C, which is that  first OR gate in the logic gate solution.
];
01 01 10 01

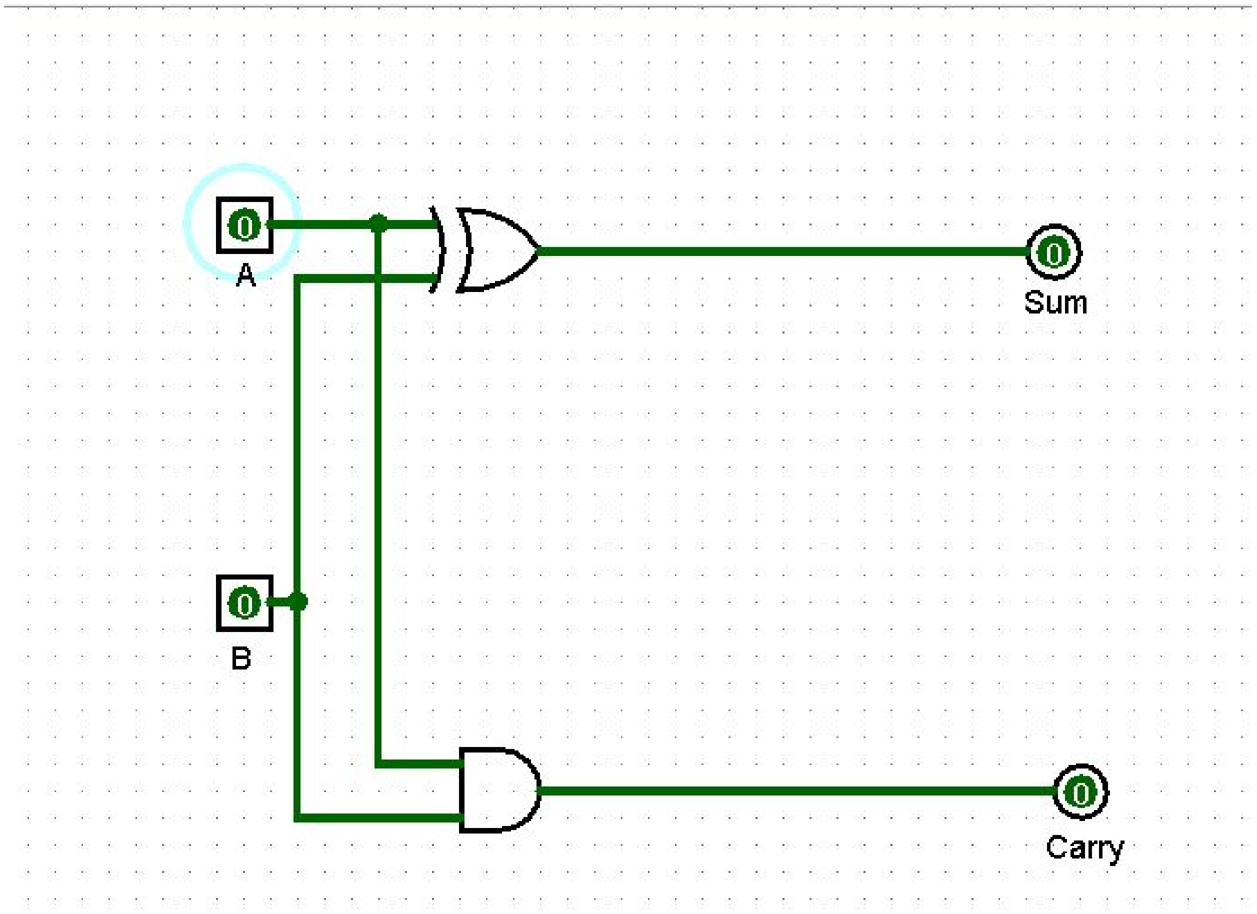
;[^BC = B        This stores the result of B AND C into register B, which is the first AND gate in the logic gate solution (bottom one)
];
01 00 11 10

;[^AD = C        This stores the result of A AND D into register C. Note that D is = B OR C (original value of B), so by substituting D with B OR C, we get A AND (B OR C). This is the top AND gate in the logic gate solution.
];

00 10 01 11
;[^C + B = D    Finally, with our registers holding the result of gates, use those results together with an OR gate. If we look above, B is = to the original values of B AND C, and C is = to the

original values of A AND D (which is actually A AND (B OR C)). Therefore, with substitution again, C is actually A AND (B OR C), and B is actually B AND C. Knowing this, D is equal to  (A AND (B OR C)) OR (B AND C). This is reflected in our logic gate solution. Alone, the program looks as follows:
00 01 10 11
01 01 10 01
01 00 11 10
00 10 01 11
];

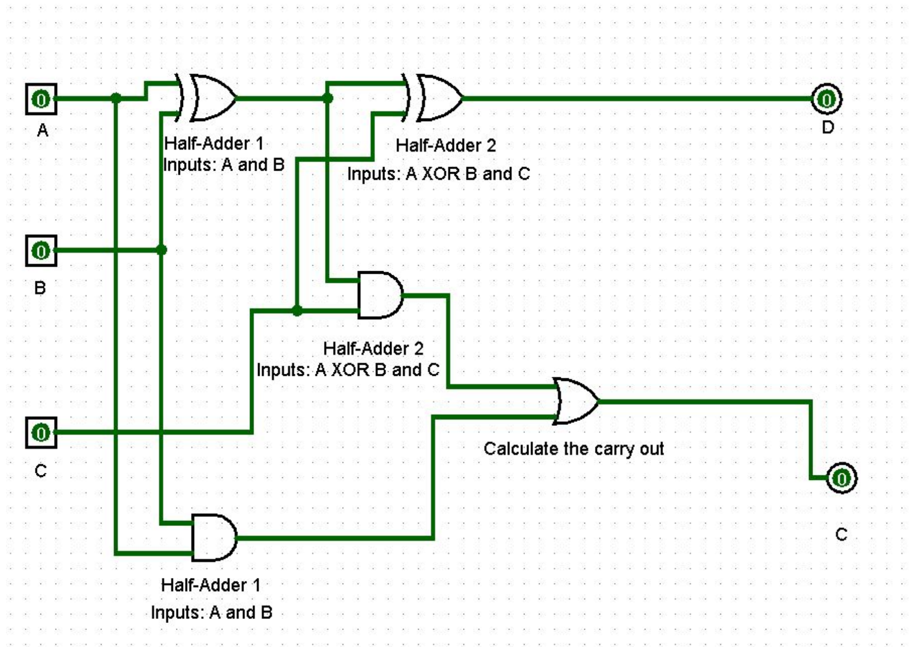
Step 3: Write Half-Adder with Puny Computer. Comment it. Can destroy contents of A and B.
;[First, we will make it in logic gates so it is easier to create in Puny Computer. This practice is actually JUST like creating higher-level code, then converting it to low-level MicMac assembly code. Interesting **coincidence**…….



Now to actually make the Puny Computer code, which seems very simple...
];
11 00 01 11

[;^ A XOR B = D Store the result of A XOR B (the inputs) into register D. This is the sum. Not much to say here.
];
01 00 01 10
[;^ AB = C Store the result of A AND B into register C, which is the carry. Because 1 + 1 = 2, but we only have 1 and 0, we must carry a 1 to the next spot, which is used in the full adder.
Our final program, in pretty form:
11 00 01 11
01 00 01 10
];

Step 4: Write Full-Adder with Puny Computer. Comment it. Can change contents of original C, A and B.
[; Obviously, the Full-Adder uses Half-Adders in its implementation, so when we make the logic gates, we will use the module Half-Adder not as a box, but with its implementation taken out. This is to make it easier to see what's going on. Note that due to having 4 registers, we will simply change what C is to replicate the output C (it is at first whatever the input is, but we change it to be what it should be).



Now to code it. ];

11 00 01 11
[;^ A XOR B = D so now D (which eventually will be sum) has contents of first XOR (A XOR B)];

01 00 01 01
[;^AB = B So B has the contents of A AND B, which is the 1st AND from the left (the original value of B is used here, and now B has a new value). Note that in doing this, we now realize we no longer need the original values of A and B. We used A for both of its uses (A XOR B and A AND B), and B for both of its uses (the same uses). By doing this, A is now a freed register with which we can use in any way, and B is now A AND B (original value of A)];

00 11 11 00
[;^D + D = A For organization, we put the value of D into A. It is impossible to simply say X = Y, so we say X OR X = Y because X OR X just reduces to X. Now we have A being A XOR B, and B being A AND B];

11 00 10 11
[;^ A XOR C = D Basically, this forms as (A XOR B) XOR C = D. Remember before we set A to be the previous value of D, which was A XOR B. This is the final sum (the 2nd XOR from the left)! We are done with the sum.];

01 00 10 10
[;^ AC = C So now we have (A XOR B) AND C, which is that AND in the middle of the diagram, 2nd from the left. We changed the carry-in to be this, because just like the case of A and B, we no longer need the original value for any operations, as we already used it for the sum! ];

00 01 10 10
[;^ B + C = C With B being A AND B, and C being (A XOR B) AND C, we now test for if either is true. This basically means if A and B are 1, or one of them is 1 AND C is 1, then make that the carry out. It’s basically a better majority function than mine in step 1, as any 2 that are on results in a 1. After this, the carry-out and sum (C and D, respectively) are set, and we are finished
Our final code, in pretty form:

```
11 00 01 11
01 00 01 01
00 11 11 00
11 00 10 11
01 00 10 10
00 01 10 10
];
```