CS-UY 2214 — Homework 2

Adapted from Jeff Epstein, Modified by Ratan Dey

Introduction

Complete the following exercises. When you finish, submit your solutions on Gradescope.

Submit your code files exactly as named in the questions.

Problems

1. If you haven't yet done so, complete the Verilog tutorial, which you can find on Brightspace. You don't have to submit anything: this is just to familiarize you with the tools.

   For these exercises, if you use Anubis, please connect to the playgroud named "CompArch / PL with Webtop." You'll have to use GKTWave, which will not run in the non-Webtop playground.

   You may want to look at the Verilog cheat sheet and reference manual, also available on Brightspace.

2. Download and extract the enclosed files from `hw2a.zip`:

   - `full_adder_nodelay.v` – a full one-bit adder module
   - `four_bit_adder_subtractor.v` – a (broken) four-bit ripple-carry adder/subtractor, which uses the adder in the above file
   - `adder_subtractor_test.v` – a test bench that will verify the behavior of the above module

   To run the test bench, compile `adder_subtractor_test.v` and run the resulting binary in the simulator. View the resulting waveform in a waveform viewer. The following commands will do it:

   ```
   iverilog -o adder_subtractor_test.vvp adder_subtractor_test.v
   vvp adder_subtractor_test.vvp
   ```

   The resulting file `adder_subtractor_test.vcd` is your waveform. To view the waveform, use one of these options:

   - If you're using Vital or Anubis with Webtop, you can open it using the command `gtkwave adder_subtractor_test.vcd`.
   - Alternatively, you can open the file on this website.

   The `four_bit_adder_subtractor` module *should* be able to add *and* subtract four bit numbers. As provided, however, it can currently only add.

   The `four_bit_adder_subtractor` module takes three inputs:

   - A – the first input (four bit)
   - B – the second input (four bit)
   - Op – the operation (one bit); this will be 0 for addition and 1 for subtraction. Initially, it is ignored within the module.

And it yields two outputs:

- S – the sum (four bit)
- Cout – the carry out (one bit)

When you run the Verilog simulator, the test bench will notify you if the adder produces unexpected results. Initially, you should see the following output:

```
When subtracting  1 and  1, got  2, but expected  0
When subtracting  2 and  5, got  7, but expected -3
When subtracting  4 and -4, got  0, but expected -8
When subtracting  7 and  7, got -2, but expected  0
```

(The numbers might seem a little funny: the test bench claims that 4 minus -4 should yield -8. This is because in a 4-bit 2's complement number, it's impossible to represent 8, the mathemtically correct solution. The bit pattern 1000 will be interpreted as -8.)

By modifying the `four_bit_adder_subtractor.v` file, make it support both addition and subtraction. **Do not** modify the other files. Your solution must be consistent with the addition algorithm we discussed in class. You may not use Verilog's built-in arithmetic operators. You may use the conditional operator (`?:`) as a multiplexer. You can use all the basic gates (`&`, `|`, `^`, $\sim$). If you've done it correctly, the test bench shouldn't print out any unexpected results.

Submit your corrected `four_bit_adder_subtractor.v` file, as well as a screenshot (named `four_bit_adder_subtractor.png` or `four_bit_adder_subtractor.jpg`) showing the output of your waveform viewer when viewing the waveform of ports on the corrected `four_bit_adder_subtractor` module. Make sure that your `four_bit_adder_subtractor` module's waveform doesn't contain any red (unknown) values. Unknown values are okay in the test bench.

3. Write a combinatorial Verilog module `binary_to_gray`. Its input, B, is an unsigned 4-bit binary number. Its 4-bit output, G, is the corresponding gray code. Use only Verilog's low-level logical operations (operators `|`, `&`, `^`, $\sim$) in your answer.

   Your module will start like this:

   ```
   module binary_to_gray(B, G);
   ```

   Be sure to declare your ports as `input` or `output`; declare any additional `wire`s you may need; and use `assign` statements to connect ports to gates.

   One way to convert from binary to gray code is to observe the following rules:

   - The MSB (most significant bit) of the output is equal to the MSB of the input.
   - Bit i of the output is obtained by XOR-ing together bits i and $i+1$ of the input (i.e. that bit, and the one to its left).

   Put your code in a file named `binary_to_gray.v`. Test your code using the `binary_to_gray_tb.v` test bench file provided in `hw2b.zip`. Note that the test bench does not automatically verify the test cases; you'll need to look at the waveform in your waveform viewer in order to determine if your output is correct. The correct gray codes are given in binary on the Wikipedia page linked above.

   Your solution must be combinatorial. You may not use `always`, `reg`, or sequential assignment.

   Submit your source file (named `binary_to_gray.v`) and a screenshot of your waveform viewer showing the waveforms of your module's ports (named `binary_to_gray.png` or `binary_to_gray.jpg`). Make sure that your module's waveform doesn't contain any red (unknown) values. Unknown values are okay in the test bench.

4. Download the file `hw2c.zip`.

   Using the ALU circuit diagram that we discussed in class as a guide, implement a combinatorial 4-bit Arithmetic Logic Unit in Verilog. Your module must be functionally identical to the unit design presented in class: it must support a 3-bit operation input that selects one of the following operations: `A & B`, `A | B`, `A + B`, `A & ~B`, `A | ~B`, `A - B`, `A < B`. The inputs and output of the unit should be 4-bit values.

   A circuit diagram of the ALU is available in the zip file, as well as in the class slides.

   In your implementation, do *not* use the high-level Verilog operators `+`, `-`, `<`, etc. Instead, implement the ALU's logic as given in the diagram, using low-level gate logic (operators `|`, `&`, `^`, ~) and an adder. Use the included `four_bit_adder` module to provide the adder functionality. You should not need more than one instance of this adder. You may use the conditional operator (`?:`) as a multiplexer. You can nest several instances of the conditional operator to simulate a larger multiplexer. Your solution must be combinatorial: you may not use `always`, `reg`, or sequential assignment.

   Use the included `alu.v` to get started. Test your code using the included Verilog testbench `alu_tb.v`. Please note that the test bench will emit an error message for every incorrect result; if you don't see any error messages, it means that your implementation is correct. Submit your completed `alu.v`.

   Hint: to create an instance of `four_bit_adder` inside your module, you can use syntax similar to the following. Below, we're creating an instance of `four_bit_adder` named `the_adder`; the name does not matter. We create a 4-bit wire named `adder_sum` and a 1-bit wire named `adder_Cout` for the sum and carry out of the adder, respectively, which we connect to the adder. You are responsible for providing reasonable values for `A`, `B`, and carry in.

```
wire [3:0] adder_sum;
wire adder_Cout;
four_bit_adder the_adder(A, B, 0, adder_sum, adder_Cout);
```

   Hint: you can use the `?:` syntax to select from one of several options. For example:

```
// X will be 3 when Y is 0, and 4 otherwise.
assign X = (Y==0) ? 3 : 4;

// Z will be 3 when Y is 0, and 4 when Y is 2, and 5 otherwise.
assign Z = (Y==0) ? 3 : (Y==2) ? 4 : 5;
```

You already know the OR, AND, and XOR operations as applied to logical (i.e. true/false) inputs. For example, `True and False == False`.

These operations can also apply to pairs of n-bit binary numbers a and b. In this case, the numbers will be treated as a sequence of bits $a_{(n-1)\ldots0}$ and $b_{(n-1)\ldots0}$. The result of a **op** b will be the sequence of bits arising from applying operation **op** to pairs of bits from each operand, i.e. a_{n-1} **op** b_{n-1}, a_{n-2} **op** b_{n-2}, ..., a_0 **op** b_0. When these operations are applied to sequences of bits, they are called *bitwise operations*.

In C and C++, the `&` operator is bitwise AND, `|` is bitwise OR, `^` is bitwise XOR, and ~ is bitwise NOT. (Distinguish these bitwise operators from their logical/boolean counterparts: `&&` for logical AND; `||` for logical OR; and `!` for logical NOT.)

For example, let's say we want to calculate $25_{10} \mid 5_{10}$, i.e. the bitwise OR of 25 and 5. That is, in decimal:

$$\begin{array}{rr} & 2\ \ 5 \\ | & \ \ 5 \\ \hline & \ \ ? \end{array}$$

First, let's convert both numbers to binary. Here, we write 8 bits for each number:

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & = 25_{10} \\ | & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & = 5_{10} \\ \hline & & & & & & & & ? & \end{array}$$

To calculate the answer, we apply the OR operation to the two bits in each column, yielding:

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & = 25_{10} \\ | & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & = 5_{10} \\ \hline & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & \end{array}$$

That value, 00011101_2, can be converted into decimal:

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & = 25_{10} \\ | & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & = 5_{10} \\ \hline & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & = 29_{10} \end{array}$$

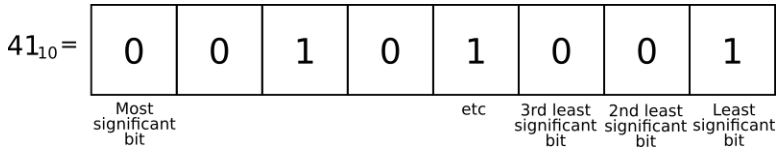
Therefore $25_{10} \mid 5_{10} = 29_{10}$.

The unary NOT operation can also be applied bitwise, in which case each of the bits of the input will be inverted. For example, to calculate $\sim 25_{10}$, we convert the value into binary, yielding 00011001_2. Note that we've written the number with 8 bits. Now we invert each bit, giving us 11100110_2, which is 230_{10}.

For the following questions, use *only* the C/C++ bit-twiddling operators `&`, `|`, `^`, and $\sim$, as well as the shift operators `<<` and `>>`.

You don't need a compiler to verify your work. You should be able to complete the functions based on your own calculations.

The following diagram, showing the terminology used to name individual bits in an 8-bit representation of the decimal number 41, may help:



5. Complete the following C function, which will return a value equal to its parameter, with the second least significant bit toggled, i.e. if the second least significant bit is on, turn it off, and if it is off, turn it on. That is, `toggleBit2(90)` should return 88, but `toggleBit2(45)` should return 47.

```
unsigned int toggleBit2(unsigned int x) {
    return YOUR CODE HERE;
}
```

Your solution should be one line. Submit your answer in a plain text file named `hw2.txt`. Number your answer with the question number. Use *only* the C/C++ bit-twiddling operators `&`, `|`, `^`, and $\sim$, as well as the shift operators `<<` and `>>`.

Hint: for this question, you probably want to use the xor operator (`^`).

6. Complete the following C function, which will return a value equal to its parameter, with the fifth least significant bit flipped *off*. That is, `flipOffBit5(90)` should return 74, but `flipOffBit5(45)` should return 45, since the fifth least significant bit is already off in 45.

Do not make any assumptions about the size of an `unsigned int`. That is, your solution should work equally well running on an architecture where `unsigned int`s are 16-bit, 32-bit, or 64-bit.

```
unsigned int flipOffBit5(unsigned int x) {
    return YOUR CODE HERE;
}
```

Your solution should be one line. Submit your answer in a plain text file named `hw2.txt`. Number your answer with the question number. Use *only* the C/C++ bit-twiddling operators `&`, `|`, `^`, and $\sim$, as well as the shift operators `<<` and `>>`.

7. Complete the following C function, which will a return value equal to the three least significant bits of its parameter repeated three times. That is, given an integer parameter whose value expressed in binary is *abc*, the function will return an integer whose value expressed in binary is *abcabcabc*.

   For example, `repeatBitsThrice(5)` should return 365, because $5_{10} = 101_2$ and $365_{10} = 101101101_2$, that is, the bit pattern of 5 repeated three times. For another example, `repeatBitsThrice(12)` should return 292, because the three least significant bits of 12 are 100_2, and $292_{10} = 100100100_2$.

   ```
   unsigned int repeatBitsThrice(unsigned int x) {
       return YOUR CODE HERE;
   }
   ```

   Your solution should be one line. Submit your answer in a plain text file named `hw2.txt`. Number your answer with the question number. Use *only* the C/C++ bit-twiddling operators `&`, `|`, `^`, and $\sim$, as well as the shift operators `<<` and `>>`.

8. Complete the following C function, which will return true (1) when its parameter's 6th, 7th, and 9th least significant bits are 1, 0, and 1, respectively; and will return false (0) otherwise. That is, given an integer parameter whose value expressed in binary is $1x01xxxxx$, the function will return return true regardless of the value of the bits in positions marked with an x.

   For example, `detectBitPattern(943)` should return true, because $943_{10} = 1110101111_2$, which has a 1 in the 6th least significant bit, a 0 in the 7th least significant bit, and a 1 in the 9th least significant bit. For another example, `detectBitPattern(47)` should return false, because $47_{10} = 101111_2$ and even though there is a 1 in the 6th least significant bit, the other conditions are not met.

   ```
   unsigned int detectBitPattern(unsigned int x) {
       return YOUR CODE HERE;
   }
   ```

   Your solution should be one line. Submit your answer in a plain text file named `hw2.txt`. Number your answer with the question number. Use *only* the C/C++ bit-twiddling operators `&`, `|`, `^`, and $\sim$, as well as the shift operators `<<` and `>>`. For this question, in addition to the bit-twiddling and shift operators, you may also use the comparison operators `==` and `!=`. Note that these operators return 1 for true and 0 for false.

   Hint: to solve this, you need only one `&` and one `==`.

9. Your friend Rupert often confuses the C/C++ logical AND operator (`&&`) with the bitwise AND operator (`&`). In frustration, he decides to abandon the logical version, and use the bitwise form exclusively. Because C and C++ use zero to represent false, and any nonzero value to represent true, he opines, using the bitwise operator in place of the logical operator will produce equivalent results.

   Is Rupert correct to make this substitution when applied to integer operands? Justify your answer with examples.