EECS 478 Fall 2014 Project 2 Report

Pengxiang Jiang
Uniquename: jpxx
UMID: 63245826

Project Implementation

1. **Creating Simple Gates**
   Completed the implementation for the function *createAND2Node()*, *createXOR3Node(),* and *createMUX4Node()* to simulate the logical 2-input AND, 3-input XOR, and 4-input MUX with 2 selected bits, separately. This task is done by modifying codes in library.cpp file.
   Because there are some given codes like *createOR2Node ()* function, so we could build other functions by referring the given codes. Although this is a simple task, there is one thing need to be attention. That is, the bit order is important for *createMUX4Node ()* function.

2. **Creating Logic Modules**
   This task is to implement two logic modules: adder module and multiplier module. For completing this task, both module.cpp file and main.cpp file are modified.
   (a) Adder
   The adder module was implemented by coding *createADDModule()* function. The basic idea for designing this n-bit adder is by recursively using the following two expressions for each bit:

$$S = A \oplus B \oplus Cin$$

$$Cout = A*B + cin*(A+B)$$

   Notice that we need not only build nodes for inputs and outputs, also some internal nodes should be created to store the internal value.
   In my implementation, I built many different nodes, like *outputAB[]* and *outputAorB[]*, to store the internal value for computing the internal *Cin* and the final *Cout*.
   After finishing *createADDModule()*, I implemented the corresponding part for adder module in main.cpp file. The mainly work here is to set the name of the circuits (*setName*), set primary inputs and outputs (*setPI, setPIs, setPO, setPOs*), and set to write the circuit into BLIF format (*writeBLIF*).
   (b) Multiplier
   The multiplier module was implemented by coding *createMULTModule()* function. The basic idea implementing this n-bit multiplier is to find all of the partial products and then add them together to get the final product. Each partial products are generated by ANDing each bit of input2 with each bit of shifted input1. Shift function is implemented by the given *createSHIFTModule()*.
   In my implementation, I used three different cases to calculate the sum of partial products:
   (1). 1-bit multiplication
   This case is the simplest one because it is actually a 1-bit adder. Notice that the output should be a 2-bit number. Therefore, zero is assigned to the MSB of the output.
   (2). 2-bit multiplication
   For this case, there are two partial products be generated: *Partial_Products0* and

Partial_Products1. *Partial_Products1* is generated by ANDing the MSB of input2 with left-shifted-one-bit input1. Therefore, *Partial_Products1* is a 3-bit number.

For adding these two partial products, *Partial_Products0* should be extended to 3-bit. Notice that the final product should be a 4-bit number. The MSB of the final product is the cout generated by *creatADDModule()*.

(3). n-bit multiplication (n > 2)

For n-bit multiplication, the basic method is similar to 2-bit multiplication. The main difference is that n-bit multiplication will generate n-level partial products. For example, *Partial_Products0* is the 0^{th} level, *Partial_Products1* is the 1^{st} level. The method to add all of these partial products together is complex than the previous two cases.

In my implementation, I used *if statement* to judge three different situations. When $i = 1$, the first two levels of partial products will be added. This situation is actually the same with 2-bit multiplication. The difference is that the output here is not the final result. It will be stored and then be used in the next step to add with next level of partial products. When $i \mathrel{!}= numBits - 1$, the function will add all of the levels of partial products except the first two and the last partial products. When $i = numBits – 1$, the sum of preceding partial products will add with the last level of partial products to obtain the final result.

The corresponding part of multiplier module in main.cpp is also completed after creating the multiplier.

3. **Creating a Datapath**

   This task is to implement function *createSUMABMULTPLYSUMCDModule()* to calculate $z = (a+b) * (c+d)$.

   For completing this task, both datapaths.cpp file and main.cpp file are modified.

4. **Create BLIF Files and Verification**

   For this project, I generated add8.blif, mult16.blif, and sumab_multiply_sumcd.blif files, which represents 8-bit adder, 16-bit multiplier, and $(a+b) * (c+d)$ datapath respectively.

   There are two approaches to implement verification for these different modules. The first approach is by using given BLIF simulator. By manually assign inputs for tested module, the simulator will generate corresponding output. Then we can check the output to verify the correctness of the module. For my implementation, all of the outputs for each module are correct.

   The second approach is by using ABC. I checked my add16.blif with the given adder16.blif by typing:

   cec add16_jpxx.blif adder16.blif

   Then the terminal showed: *Networks are equivalent.*

   Therefore, the module I generated is equivalent to a trusted circuit.