EECS478 Assignment 2 Report

Yongjoo Park (UMID:4562 2850)
pyongjoo@umich.edu

April 2, 2013

1 Creating Simple Gates

1.1 Implementation

Mostly I copied the code from createOR2Node, and changed the truth table entries.

1.2 Verification

The logic was too simple, and I did not perform any special verification. However, createXOR3Node has been used to implement a function createFullAdder which is used to implement n-bit adder, and it is verified naturally as verifying bigger circuits, in a black box manner.

2 Creating Logic Modules

2.1 Implementation

The implementation of the function createADDModule basically follows the structure of another function createSHIFTModule which is given as an example. It first creates input and output nodes, and carry nodes used internally. The name of carry nodes are assigned by concatenating c_ to input strings to make the node name unique throughout the circuit. This strategy is helpful later when we designed a bigger circuit (Datapath).

The n-bit adder implemented by the function createADDModule is ripple-carry adder, and it is not an optimal implementation in terms of speed. The better approach would be carry-lookahead adder, but I did not bother myself to increase the efficiency, since it is apparently not what this assignment was asking. The second module createSUBModule uses the above function createADDModule internally (which might be well expected). In order to convert the addition operation to subtraction operation, we need a light wrapper to invert the second input, which will be subtracted, and set the initial carry to the adder to one.

2.2 Verification

I tested the adder for 4, 8, and 16 bits. Four bits and eight bits implementations were tested quite comprehensively by creating equivalent blif files with python code Yes, it was a very brute-force truth table generation. However, the strategy was no longer tractable when it comes to higher bits implementations due to the computational inefficiency; generation two 16 bits input combination with corresponding output seem to not stop forever. The 16 bits implementation was simply compared against the provided blif sample file with abc software following the tutorial given in the assignment manual.

The subtractor does not require much verification as much as the adder since its already using the adder module internally. I tested a very simple two-bit subtractor with several input combinations with the provided simulator, and observed satisfactory answers.

3 Creating a Datapath

3.1 Implementation

The function 'createABSMIN5X3YModule' which implements this module relies on several other modules desinged in the previous sections or newly for this task. The major module introduced newly would be 'min' and 'abs' operations. Before discussing these modules, we go step by step.

First the multiplications by five and three have been implemented with the n-bit shifter provided and the adder designed above. 'min' operation is wired by another function 'createMINModule'. The function contains a subtractor internally, and use the msb of the output from the subtraction operation to MUX either of inputs. The MUX is performed bit by bit using a newly created function 'createMUX2Node' in library.cpp. The 'abs' module also uses a substractor internally to generate two's-complemented value. The msb of the input is used to MUX either of the original input or the complemented value.

3.2 Verification

I tested the module with several sample inputs randomly generated by hand-made python code. The result is matched by running the simulator program provided. The python code used is attached below. I tested with 10 random samples, and all produced exactly the same results.

```
'''
Test pattern generator for the function 'createABSMIN5X3YModule'.
'''
import random
from bitstring import BitArray

def operation(input1, input2):
    return abs(min(5*input1, 3*input2))

def run():
    num_test = 10
    for i in range(num_test):
        input1 = random.randint(-2**15, 2**15-1)
        input2 = random.randint(-2**15, 2**15-1)
        output = operation(input1, input2)

        ib1 = BitArray(int=input1, length=16)
        ib2 = BitArray(int=input2, length=16)
        out = BitArray(int=output, length=19)

        print ib1.bin
        print ib2.bin
        print out.bin
        print

run()
```