EECS 478 Winter 2017 Project No. 2

The University of Michigan, EECS Department
EECS 478: Logic Synthesis and Optimization, Winter 2017
Professor John P. Hayes
Sixian Hong

Project 2: Logic and Datapath Synthesis

Distribution Date: Thursday February 23th, 2017
Due Date: Wednesday March 22th, 2017 at 23:55
Submit Online on Canvas → Assignments → Project 2

INSTRUCTIONS:

- You must abide by the Engineering Honor Code. All work submitted should be work done by you and you alone. If you are unsure about the level of collaboration, please consult Professor Hayes or GSI.
- For this project, you may exchange `.blif` files with other students. If you do, state which students you traded with in your report.
- When you are finished, collect the following final deliverables:
  - All `.cpp` and `.h` files that you use, including any extra files that you added.
  - `Makefile` - your program **must** compile.
  - `add8_{uniquename}.blif`, `sub16_{uniquename}.blif`, `mult5_{uniquename}.blif`, and `absmin9x12y_{uniquename}.blif`.
  - Report, either in `.doc` or `.pdf` format.
- Create a tar ball with all your deliverables. To do this:
  - Create an empty directory called `EECS478P2_{uniquename}` , where `{uniquename}` is your uniquename.
  - Copy all your deliverables into this directory.
  - Type `tar -cvf EECS478P2_{uniquename}.tar EECS478P2_{uniquename}`.
  - Type `gzip EECS478P2_{uniquename}.tar .`

1 Overview

This project illustrates the process of logic and data synthesis. Given an existing infrastructure in C++, you will instantiate logic modules, including adders, subtractors, multipliers, and a datapath module. All files generated should be BLIF (Sec. 2), and verified using the software package `abc` from UC Berkeley (Sec. 3).

2 Berkeley Library Interchange Format (BLIF)

Digital circuits can be represented in the Berkeley Library Interchange Format (BLIF). This format defines a truth table for each logic element (e.g., gate) as well as the connections between these elements.

The general format is as follows.

```
 1   .model <model name>
 2
 3  .inputs <in_1> <in_2> ... <in_n>
 4
 5  .outputs <out_1> <out_2> ... <out_m>
 6
 7  .names <in_1> <in_2> ... <in_k> <out>
 8  # truth table involving <in_1> ... <in_k> <out>
 9  # ...
10   # many truth tables can be specified
11
12  .end
```

Line 1 defines the model or circuit name. Line 3 specifies the primary inputs of the circuit. Line 5 specifies the primary outputs of the circuit. Lines 7-10 specify a truth table for a logic element with inputs in_1 ... in_k and output *out*. A '1' means the variable is uncomplemented, a '0' means the variable is complemented, and '-' means the variable is not set (don't care). By default, any input combinations **not** specified will mean the output result is 0. Therefore, it is sufficient to specify a *cover* for the output function. Multiple truth tables may be specified to define the full logic of the circuit. Line 12 defines the end of the model or circuit definition.

For example, given the following circuit definition (using Boolean logic), where the primary inputs are a, b, c, d, the primary output is f_3

$$f_1 = \overline{a} + b$$
$$f_2 = \overline{cd}$$
$$f_3 = f_1 \oplus f_2$$

A corresponding BLIF file would be:

```
 1   .model example
 2
 3   .inputs a b c d
 4
 5   .outputs f3
 6
 7   .names a b f1
 8   0- 1
 9   -1 1
10
11   .names c d f2
12   0- 1
13   -0 1
14
15   .names f1 f2 f3
16   01 1
17   10 1
18
19   .end
```

3 abc

`abc` (`http://www.eecs.berkeley.edu/~alanmi/abc/`) is a free software package from UC Berkeley that can validate the correctness of your circuit through equivalence checking.

Given a `.blif` file, you can check if the module you generated is equivalent to a trusted circuit. Consider two simple circuits.

```
 1   .model simple0
 2
 3   .inputs x y
 4
 5   .outputs z
 6
 7   .names x y z
 8   00 1
 9   01 1
10   10 1
11
12   .end
```

```
 1   .model simple1
 2
 3   .inputs x y
 4
 5   .outputs z
 6
 7   .names x y z
 8   0- 1
 9   -0 1
10
11   .end
```

In this case, `simple0` represents $z = \overline{xy}$ and `simple1` represents $z = \bar{x} + \bar{y}$. The two circuits are equivalent by de Morgan's Law.

When verifying two circuits, the `.inputs` and `.outputs` **must be exactly the same**. Otherwise, `abc` will conclude that the two circuits are not equivalent, regardless of the logic. After installing the package, you should see the following after starting up `abc`.

```
UC Berkeley. ABC 1.01

abc 01>
```

For this project, you will mainly be concerned with the command `cec`. To check the equivalency of two `.blif` files, type the following after starting up `abc`.

```
cec file1.blif file2.blif
```

4 Provided Files

Log into Canvas and download the tar ball `eecs478p2.tar.gz`. After you untar the tar ball, you should see the following source files.

- `truthTable.h` and `truthTable.cpp`
- `node.h`
- `circuit.h` and `circuit.cpp`
- `library.cpp`, `modules.cpp`, and `datapaths.cpp`
- `main.cpp`
- `Makefile`
- `adder16.blif`

Feel free to modify any and all of these files to suit your needs. Please make sure that your final binary is called `project2`.

4.1 truthTable.h and truthTable.cpp

These two files store the underlying infrastructure for representing a truth table. Each entry stored represents a cover for the logic function.

4.2 node.h

This file defines a logical element (node) of a circuit. It includes a truth table, a list of fanin nodes, and a type (PRIMARY_INPUT, PRIMARY_OUTPUT, INTERNAL, ZERO_NODE, ONE_NODE).

4.3 circuit.h and circuit.cpp

These two files define the circuit or model, which is defined as a collection of nodes.

4.4 library.cpp, modules.cpp, and datapaths.cpp

These three files contain the implementation of different functions defined in `circuit.h`.

- `library.cpp` contains functions operating on nodes (mainly used for internal purposes)
- `modules.cpp` contains the implementations of the logic modules
- `datapaths.cpp` contains the implementations of the datapath module

4.5 main.cpp

This file contains the general wrapper on how to use the executable. There is a '-help' function built in. To see the usage, type `./project2 -help`.

4.6 Makefile

This file compiles and creates the final binary for this project. You are allowed to modify this file and add any optimization flags of your choice.

4.7 adder16.blif

This file is a reference `.blif` file that implements a 16-bit adder module.

5 Getting Started

Download and untar the package `abc70930.tar.gz` from Canvas by typing

```
tar -xvf abc70930.tar.gz
```

In the `abc70930` directory, you can find the `abc` binary. The binary runs in CAEN machines only.

Download the tar ball `eecs478p2.tar.gz` from Canvas. To decompress the tar ball, type

```
tar -xvf eecs478p2.tar.gz
```

into a directory of your choosing. This will create a directory named `eecs478p2`, and will include the necessary files for your program to compile. In that directory, compile the program by typing `make`. To run the program, type `./project2`. This will display the usage options of the binary.

You are allowed to do your development in any platform. However, **we will be testing and evaluating your code strictly on the CAEN Linux machines**.

6 Tasks (100 points)

This section details the different tasks you must complete. Be sure and test your code **thoroughly** before submitting. The more testing you do, the more confident you can be that your code is correct. The bulk of your modifications should be done in `library.cpp`, `modules.cpp`, `datapaths.cpp`, and `main.cpp`, but feel free to modify other files as needed. If you plan on making substantial changes to the code base, such as the underlying infrastructure, please consult with Professor Hayes or GSI first.

6.1 Creating Simple Gates (10 points)

You must implement the following three functions found in `library.cpp`.

6.1.1 2-Input AND (3 points)

Complete the implementation for the function `createAND2Node(...)`. This function should create the logical AND function in the output node `output` from the two input nodes `input1` and `input2`.

6.1.2 3-Input XOR (3 points)

Complete the implementation for the function `createXOR3Node(...)`. This function should create the logical XOR function in the output node `output` from the three input nodes `input1`, `input2`, and `input3`.

6.1.3 4-Input MUX With 2 Select Bits (4 points)

Complete the implementation for the function `createMUX4Node(...)`. This function should create a 4-input multiplexer with inputs `input1 ... input4` and two select bits `select1` and `select2` in the output node `output`. Assume that the most significant bit (MSB) of the select bits is `select2`, and the MSB of the input bits is `input4`. Likewise, the least significant bit (LSB) of the select bits is `select1`, and the LSB of the input bits is `input1`.

6.2 Creating Logic Modules (60 points)

You must implement the following three logic modules found in `modules.cpp`. The adder module is worth 20 points, the subtractor module is worth 10 points, and the multiplier module is worth 30 points. For all cases, let the MSB be the bit with the highest index, and let the LSB be the bit with the lowest index. For instance, given inputs $a[3]a[2]a[1]a[0]$, $a[3]$ is the MSB, and $a[0]$ is the LSB. Please make sure the name/format of the inputs/outputs are consistent with the name/format given in the instructions.

6.2.1 Adder (20 points)

Complete the implementation for the function `createADDModule(...)`. This function should create an *numBits*-bit adder that computes $a+b+cin$ and stores the result in s and *cout*. Assume the inputs are *unsigned*.

Primary Inputs: $a[numBits-1]...a[0]$, $b[numBits-1]...b[0]$, cin
Primary Outputs: $s[numBits-1]...s[0]$, *cout* (*cout* is considered the MSB or equivalently $s[numBits]$)

6.2.2 Subtractor (10 points)

Complete the implementation for the function `createSUBModule(...)`. This function should create an *numBits*-bit subtractor that computes $a-b$ and stores the result in s. Assume the inputs are *signed* (2's complement).

Primary Inputs: $a[numBits-1]...a[0]$, $b[numBits-1]...b[0]$
Primary Outputs: $s[numBits-1]...s[0]$

6.2.3 Multiplier (30 points)

Complete the implementation for the function `createMULTModule(...)`. This function should create an *numBits*-bit multiplier that computes $a \times b$ and stores the result in s. Assume the inputs are *unsigned*.

Primary Inputs: $a[numBits-1]...a[0]$, $b[numBits-1]...b[0]$
Primary Outputs: $s[2 \times numBits-1]...s[0]$

6.2.4 Verification

It is your job to define a verification strategy and follow it throughout the project. You can start by writing 1-bit and 2-bit modules in BLIF and then increase your bit-size to check the limits of your program. We have provided a 16-bit adder `adder16.blif` to help verify the correctness of your generated module.

You can use the BLIF simulator available in the `sim.tar.gz` file. This simulator runs in CAEN machines only. You may also swap BLIF files with other students. If you do, state clearly who you have collaborated with in your report. In any case, you should include your verification strategy in your final report.

6.3 Creating a Datapath (20 points)

Complete the implementation function `createABSMIN9X12YModule(...)` in `datapaths.cpp`. This function gets two 16-bit inputs x and y and returns a 20-bit output $z = abs(min(9x, 12y))$ (the absolute value of the minimum of $9 \times x$ and $12 \times y$). x and y are signed (2's complement) numbers. Note that the inputs are fixed to 16 bits and the output has 20 bits, because $12 \times y$ can use up to 20 bits. As with the logic modules, the msb, which is also the sign-bit, is the bit with the highest index, and the lsb is the with the lowest index.

Primary Inputs: $x[15]...x[0], y[15]...y[0]$
Outputs: $z[19]...z[0]$

You may use previously designed modules such as multiplexer, adder, subtractor, and so on. You may also want to design new modules to simplify your code.

6.4 Code Readability (2 points)

A small portion of points will be for coding style and readability. Your code does not need to be perfectly clean, but should be well-structured and clear. The following are some helpful guidelines to writing readable code.

- Use proper indentation when declaring `while` and `for` loops, and when declaring `if` statements.
- Avoid going over 80 characters per line.
- Write a 1-2 line comment for explaining non-standard coding syntax.
- Write a short comment for explaining what each function (or sub-function) does.

6.5 Report (8 points)

You will need to write a (minimum) 2-3 page report using 12-point font in either `.pdf` or `.doc` format. On the front page, print your name and uniquename. For each Task, write a brief description of your implementation. Explain any specific issues you faced and any optimizations you performed (or could have performed). In addition, include anything of interest that you think may be helpful when evaluating your submission.

6.6 Additional Files To Submit

In addition to all your code, you must submit the following `.blif` files.

- `add8_{uniquename}.blif` (8-bit adder module).
- `sub16_{uniquename}.blif` (16-bit subtractor module).
- `mult5_{uniquename}.blif` (5-bit multiplier module).
- `absmin9x12y_{uniquename}.blif` (abs(min(9x, 12y)) module).

Be sure to replace `{uniquename}` with your own uniquename. For submission instructions, refer to the front page.