Lab 2: Processor Components

1 Introduction

1.1 Purpose

The purpose of this assignment is to familiarize you with constructing hierarchical RTL implementations. This assignment asks you to create simple low-level modules, similarly to Lab 1, and then compose them into a more complex circuit.

2 P1: Hierarchical 16-bit adder

Implement a 16-bit adder by composing four 4-bit adders together.

2.1 Design

Implement both a top module and submodule. The submodule should have the following declaration:

```
module rtl_4bit_adder(
  input  logic [3:0] a,    // Input A
  input  logic [3:0] b,    // Input B
  input  logic       cin, // Carry in
  output logic [3:0] s,    // Sum
  output logic       cout // Carry out
);
```

`rtl_4bit_adder` should compute $a + b + c_{in}$ and output the corresponding value to s and c_{out}.

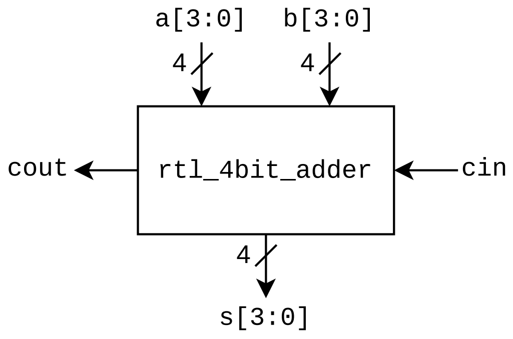

Figure 1: `rtl_4bit_adder` block diagram

The top module should be named `rtl_16bit_adder` and have the following declaration:

```
module rtl_16bit_adder(
  input  logic [15:0] a,    // Input A
  input  logic [15:0] b,    // Input B
  input  logic        cin, // Carry in
  output logic [15:0] s,    // Sum
  output logic        cout // Carry out
);
```

`rtl_4bit_adder` should compute $a + b + c_{in}$ and output the corresponding value to s and c_{out}. The module **must** instantiate instances of `rtl_4bit_adder` in order to perform the operation, as shown in Figure 2.

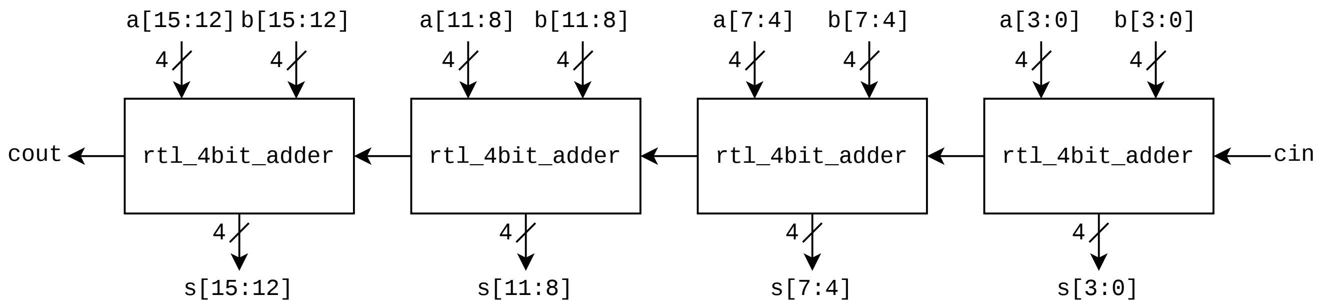

Figure 2: `rtl_16bit_adder` block diagram

2.2 Testbench

Implement a testbench which verifies that your top level design in Section 2.1 conforms to the specification. You will note that unlike Lab 1, it is not feasible to exhaustively check every input combination for the 33 inputs (2^{33} possibilities!). Instead, you should use directed testing to test for common inputs as well as edge cases.

Although you will not need to submit it to the autograder, you may find it helpful to first write a testbench for `rtl_4bit_adder` and test that the module works individually, before testing the entire `rtl_16bit_adder` module. This is known as unit testing.

3 P2: Processor Design

Implement a processor design using the 16-bit adder from Section 2.1, along with a mux, status register, and 8-to-3 priority encoder.

3.1 Design

3.1.1 32-bit 2-to-1 Mux

Construct a 32-bit, 2-to-1 Multiplexer. Recall the functionality of a multiplexer from Lab 1. Use the following declaration:

```
module mux32(
  input  logic [31:0] a,
  input  logic [31:0] b,
  input  logic        sel,
  output logic [31:0] y
);
endmodule
```

`mux32` should pass `a` to the output if `sel` is 1 and `b` if `sel` is 0.

3.1.2 Status Register

Construct an 8-bit status register. This module stores various flags and bits in a status register based on the input conditions. The status register is synchronous with the positive edge of `clk`. If reset is active (`rstN` == 0), then the register value should be synchronously set to 0. Otherwise, the register bits should be set with the corresponding flag based on the flag name. Unused states should be statically set to 1. The output of the module should be the output of the register. Use the following declaration:

```
module status_reg(
  input  logic       clk,
  input  logic       rstN,   // active low reset
  input  logic       int_en,
  input  logic       zero,
  input  logic       carry,
  input  logic       neg,
  input  logic [1:0] parity,
  output logic [7:0] status
);
endmodule
```

You should use the following mapping for the status bits:

status[7]	status[6]	status[5]	status[4]	status[3]	status[2]	status[1:0]
int_en	unused	unused	zero	carry	neg	parity

3.1.3 8-to-3 Priority Encoder

Construct an 8-to-3 priority encoder. Recall the functionality of a priority encoder from Lab 1. Use the following declaration:

```
module Pri_En(
  input  logic [7:0] D,
  output logic [2:0] Q
);
endmodule
```

3.1.4 Processor

The processor module is the top-level module which integrates the multiplexer, status register, and 8-to-3 priority encoder submodules. Use the following declaration:

```
module processor(
  input  logic        clk,         // clock
  input  logic        rstN,        // active low reset signal
  input  logic [31:0] data_in,
  input  logic [31:0] i_data,
  input  logic        data_select,
  input  logic [15:0] status_flags,
  output logic [31:0] data_out,
  output logic [ 7:0] status,
  output logic [ 2:0] Q
);
```

This module must instantiate and connect the 3 submodules (mux32, status_reg, Pri_En) as shown in Figure 3.

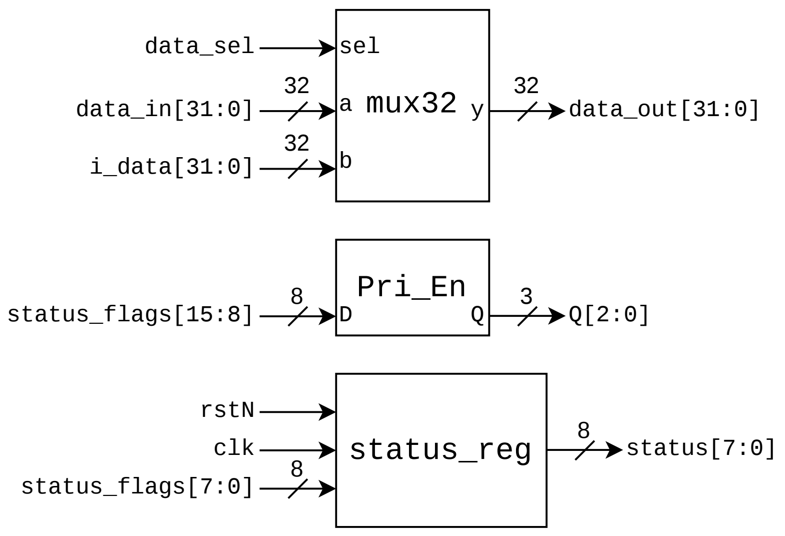

Figure 3: `processor` block diagram

3.2 Testbench

Implement a testbench which verifies that your design in Section 3.1 conforms to the specification. You can use the following template:

```
module processor_tb();

  // Declare inputs (drive these in your test sequence)
  logic        clk,         // clock
  logic        rstN,        // active low reset signal
  logic [31:0] data_in,
  logic [31:0] i_data,
  logic        data_select,
  logic [15:0] status_flags,

  // Declare outputs (check these match expected)
  logic [31:0] data_out;
  logic [ 7:0] status;
  logic [ 2:0] Q;

  // Instantiate DUT (ports match names, so wildcard is OK)
  processor DUT(.*);

  // Simple clock generator (edit period if you want)
  initial clk = 0;
  always #5 clk = ~clk;

  initial begin
    // Write your test sequence here

  end

endmodule
```

4 Testing

Testing is performed the same as Lab 1. The details are repeated here for your convenience.

4.1 Running Your Tests

After writing your designs and testbenches, you should run them using VCS. You can use other RTL simulation tools if you are familiar with them; however, it is strongly recommended to use VCS as 1) your assignment will be graded using VCS as the difinitive source of truth, and 2) it provides experience with the most widely used RTL simulator in industry.

As an example, you can compile Problem 1 using the following command in the terminal:

```
vcs -full64 -kdb -sverilog adder_tb.sv adder.sv -lca -debug_access+all
```

Then, you can run the simulation with: `./simv`

For more details, please see the lab recordings on how to use VCS and Verdi for testing and debugging.

4.2 Testbench Output

Testbenches must report their status by printing either:

- `@@@PASS` if the design behavior is correct
- `@@@FAIL` if the design behavior is incorrect

These strings must be printed **verbatim** (case-sensitive, and including the `@` symbol). Each testbench should print **exactly one** status message. If a testbench prints multiple status strings, or both `@@@PASS` and `@@@FAIL`, it will be ignored by the autograder. It is therefore recommended that you:

- Use `$display("@@@PASS")` only at the end of your testbench, after all checks pass, followed by `$finish()`.
- Call `$display("@@@FAIL")` immediately when a failure is detected, followed by `$finish()`.

You are free to display other strings for debugging (e.g., `"pass"`, `"TEST FAILED"`), but they will be ignored by the autograder.

5 Submission

For each problem:

1. Write SystemVerilog code to implement the design. For Problem 1, name the file `adder.sv`. For Problem 2, name the file `processor.sv`. Put all modules for the respective problem into the same file.
2. Write your own testbench to verify your implementation on a sequence of inputs. For Problems 1/2, name the testbench files as adder_tb.sv / processor_tb.sv and ensure the testbench module name is `adder_tb` / `processor_tb`, respectively.
3. Once you are convinced your implementation is correct, submit the design files to the autograder. See the autograder tutorial on BrightSpace for instructions on how to submit to the autograder as well as tips for interpreting feedback.

Do not change the module declarations from the ones provided. If there are any mismatches, the design will fail to compile.

Your testbenches must be **black-box**, meaning they cannot access internal signals of the module being tested. Your testbenches are only allowed to view the inputs and outputs of the design.

6 Grading

Both your design **and** your testbench(es) will be graded by the autograder. Your design will be checked for correctness by running it against several hidden instructor test cases. You will receive points for each test that you pass. Your testbench(es) will be checked for coverage and thoroughness by running it/them against several hidden instructor buggy designs. You will receive points for each buggy design you catch, meaning at least 1 testbench correctly reports the design as failing. Testbenches will be ignored if they report a correct design as buggy or fail to compile. Your final score is determined by the score of your **best** submission at the time of the deadline.

This assignment has a firm deadline (check BrightSpace). No late submissions will be accepted.