SystemVerilog for Design

Course Version 2.0

Lab Manual **Revision 1.0**

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Table of Contents

SystemVerilog for Design

Overview of Labs

In these labs, you use SystemVerilog language constructs to complete simple, common design tasks. You can use Verilog-2001 constructs to complete some of the design tasks described here, but that would obviously defeat the purpose of the labs. Where possible, try to use SystemVerilog constructs.

This lab book assumes you are familiar with the Cadence® simulator, which you use in this course. If this is not the case, please ask your instructor or contact Cadence for information on running the simulator.

The goal is not to complete all the exercises during the course, but to learn from each exercise at your own pace.

Software Releases

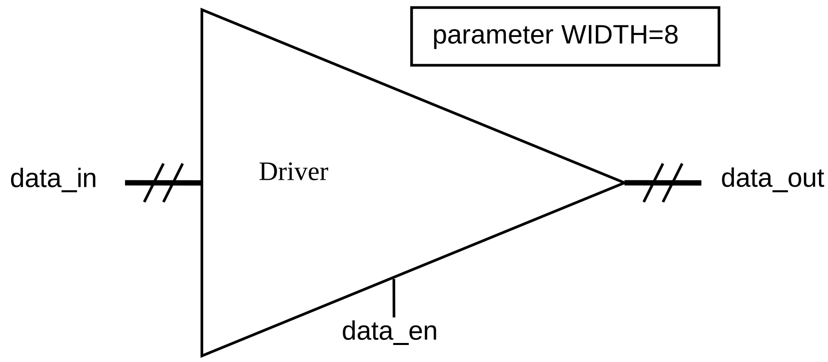
These exercises and code examples have been written and tested on the following releases:

- Xcelium™ 21.01

Lab 1 Modeling a Data Driver

Objective: To describe and instantiate a parameterized-width bus driver.

The driver output is the input value while enabled (*data_en* is true) and is high-impedance while not enabled.



Specification

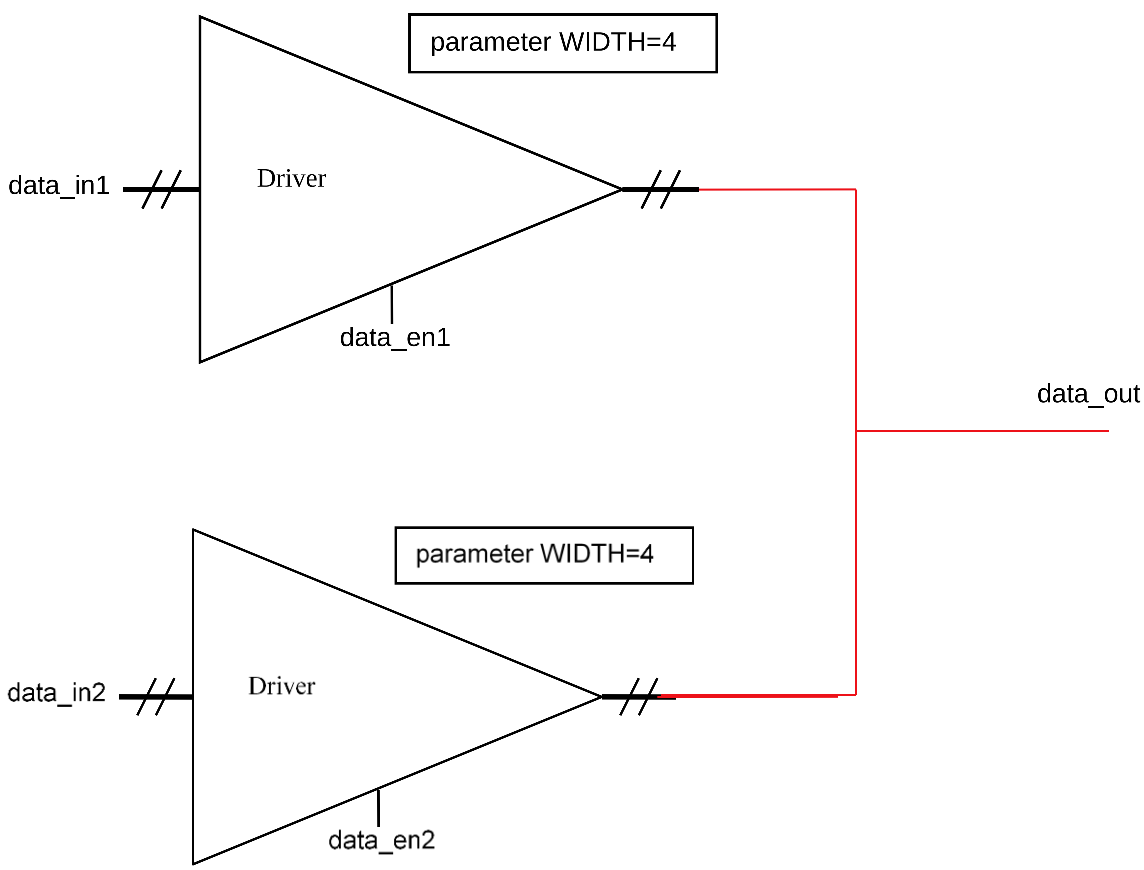
- `data_in` and `data_out` are both parameterized 8-bit `logic` vectors.
- If `data_en` is high, the input `data_in` is passed to the output `data_out`.
- Otherwise, `data_out` is high impedance.

Creating the Data Driver Design

Work in the `lab01-driver` directory:

1. Create the *tri_driver.sv* file, and using your favorite editor, describe the driver module. Parameterize the driver input and output width so that the instantiating module can specify the width of each instance. Assign a default value of 8 to the parameter.

2. Create the *multi_driver.sv* file and instantiate two instances of *tri_driver* module with the same `data_out` and override the default parameter value with 4, as shown below.



Testing the Driver Design

1. A testbench is provided in the file `driver_test.sv`. Simulate the testbench and register a design with the following run command.

   ```
   xrun driver_test.sv multi_driver.sv tri_driver.sv
   ```

   or

   ```
   xrun -f lab1.f
   ```

   Note: The *lab1.f* file lists all the files that need to be simulated (provided in the same directory).

   You should see the following results:

   ```
   At time 1 data_en1=0 data_in1=0000 data_en2=0 data_in2=1111 data_out=zzzz
   At time 7 data_en1=1 data_in1=0000 data_en2=0 data_in2=1111 data_out=0000
   At time 13 data_en1=0 data_in1=0000 data_en2=1 data_in2=1111 data_out=1111
   At time 19 data_en1=1 data_in1=0000 data_en2=1 data_in2=1111 data_out=xxxx
   TEST PASSED
   ```

   Debug your design as required.

Run Command

xrun -f lab1.f (Batch Mode)

xrun -f lab1.f -gui -access +rwc (GUI Mode)

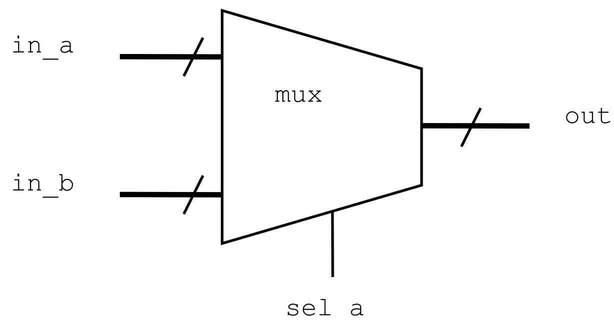
End of Lab

Lab 2 Modeling a Simple Multiplexor

Objective: To use SystemVerilog procedural constructs to model a simple multiplexor.

Create a simple multiplexor design using SystemVerilog constructs and test it using the supplied testbench.

Read the specification first and then follow the instructions in *Creating the MUX Design*.



Specification

- `in_a`, `in_b` and `out` are all `logic` vectors.
- The MUX width is parameterized with a default value of 1.
- If `sel_a` is `1'b1`, input `in_a` is passed to the output.
- If `sel_a` is `1'b0`, input `in_b` is passed to the output.

Creating the MUX Design

1. Work in the `lab02-mux` directory.
2. Create a new file called `scale_mux.sv`, containing a module named `scale_mux`.
3. Write the MUX model using the following SystemVerilog constructs:
   - Verilog2001 ANSI-C port declarations
     - Parameterize the MUX width and give it a default value of 1
   - `always_comb` procedural block
   - timeunit and timeprecision

- `unique case` construct
  - Include a `default` match that sets the output to unknown.

Testing the MUX Design

1. A testbench is provided in the `scale_mux_test.sv` file. Simulate the testbench and MUX design.

   You should see the following results:

   ```
   0ns in_a=00 in_b=00 sel_a=0 out=00
   1ns in_a=00 in_b=00 sel_a=1 out=00
   2ns in_a=00 in_b=ff sel_a=0 out=ff
   3ns in_a=00 in_b=ff sel_a=1 out=00
   4ns in_a=ff in_b=00 sel_a=0 out=00
   5ns in_a=ff in_b=00 sel_a=1 out=ff
   6ns in_a=ff in_b=ff sel_a=0 out=ff
   7ns in_a=ff in_b=ff sel_a=1 out=ff
   MUX TEST PASSED
   ```

   Debug your MUX as required.

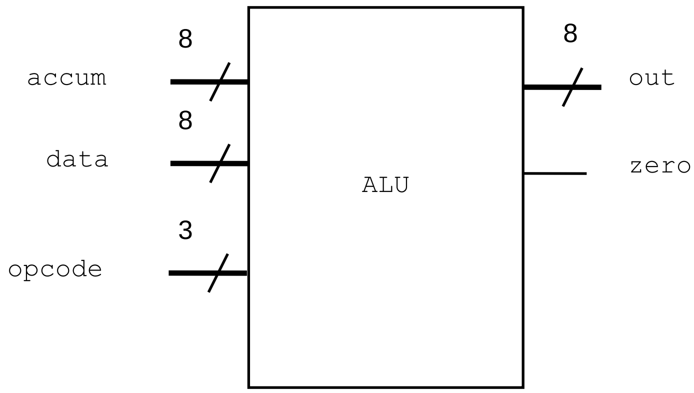
End of Lab

Lab 3 Modeling an Arithmetic Logic Unit (ALU)

Objective: To use SystemVerilog procedural constructs to model an ALU.

Create an ALU design using SystemVerilog constructs and test it using the supplied testbench.

Read the specification first and then follow the instructions in the Creating the ALU Design section of this lab.



Specification

- `accum`, `data` and `out` are all 8-bit `logic` vectors. `opcode` is a 3-bit `logic` vector for the CPU operation code.
- `zero` is a single-bit, asynchronous output with the value of 1 when `accum` equals 0. Otherwise, `zero` is 0.
- `out` takes the following values depending on `opcode`.

Opcode	Encoding	Output
HLT	000	accum
SKZ	001	accum
ADD	010	data + accum
AND	011	data & accum
XOR	100	data ^ accum
LDA	101	data
STO	110	accum
JMP	111	accum

Creating the ALU Design

Work in the `lab03-operators` directory:

1. Find `alu_test.sv` file already existing in the directory, which is used to verify the file you create.

2. Create a new file called `alu.sv`, containing a module named `alu`.

3. Write the ALU model using the following SystemVerilog constructs:
   - Verilog2001 ANSI-C port declarations
   - `timeunit` and `timeprecision`
   - `always_comb` procedural block to generate `zero`
   - `always_comb` procedural block to generate `out`

Testing the ALU Design

1. Simulate the testbench and controller design.

   You might find it easier to list all the files and simulation options in a text file and pass the file into the simulator using the `-f` `xrun` option:

   ```
   xrun -f filelist.txt -access rwc
   ```

   Debug your ALU as required, until you see the following message:

   ```
   ALU TEST PASSED
   ```

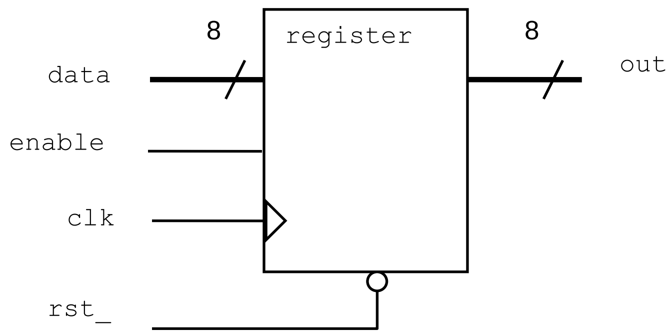
End of Lab

Lab 4 Modeling a Simple Register

Objective: To use procedural constructs to model a simple register.

Create a simple register design using SystemVerilog and Verilog-2001 constructs and test it using the supplied testbench.

Read the specification first and then follow the instructions in the Creating the Register Design section.



Specification

- `data` and `out` are both 8-bit `logic` vectors.
- `rst_` is asynchronous and active low.
- The register is clocked on the rising edge of `clk`.
- If `enable` is high, the input `data` is passed to the output `out`.
- Otherwise, the current value of `out` is retained in the register.

Creating the Register Design

Work in the `lab04-reg` directory:

1. Create a new file called `register.sv`, containing a module named `register`.
2. Write the register model using the following SystemVerilog constructs:
   a. `always_ff` procedural block
   b. `timeunit` and `timeprecision`

c. Verilog2001 ANSI-C port declarations

Testing the Register Design

3. A testbench is provided in the file `register_test.sv`. Simulate the testbench and register design.

   You should see the following results:

```
time=   0.0 ns enable=x rst_=1 data=xx out=xx
time=  15.0 ns enable=x rst_=0 data=xx out=00
time=  25.0 ns enable=0 rst_=1 data=xx out=00
time=  35.0 ns enable=1 rst_=1 data=aa out=aa
time=  45.0 ns enable=0 rst_=1 data=55 out=aa
time=  55.0 ns enable=x rst_=0 data=xx out=00
time=  65.0 ns enable=0 rst_=1 data=xx out=00
time=  75.0 ns enable=1 rst_=1 data=55 out=55
time=  85.0 ns enable=0 rst_=1 data=aa out=55
REGISTER TEST PASSED
```

   Debug your register as required.

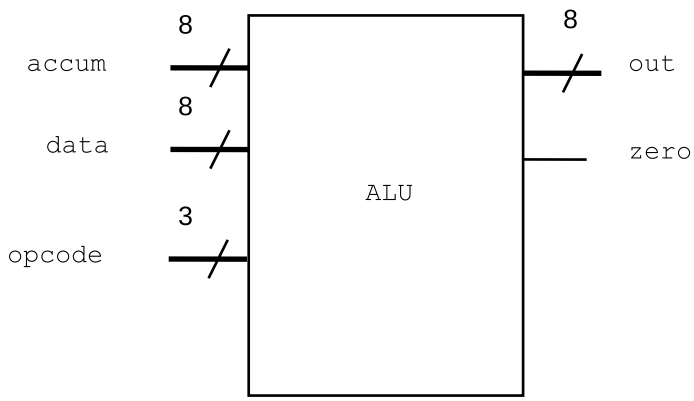
End of Lab

Lab 5 Remodeling the Arithmetic Logic Unit (ALU)

Objective: To remodel the ALU using enumerated types and packages.

Create an ALU design using SystemVerilog constructs and test it using the supplied testbench.

Read the specification first and then follow the instructions in the Creating the ALU Design section of this lab.



Specification

- `accum`, `data` and `out` are all 8-bit `logic` vectors. `opcode` is a 3-bit `logic` vector for the CPU operation code.
- `zero` is a single bit, asynchronous output with the value of 1 when `accum` equals 0. Otherwise, `zero` is 0.
- `out` takes the following values depending on `opcode`.

Opcode	Encoding	Output
HLT	000	accum
SKZ	001	accum
ADD	010	data + accum
AND	011	data & accum
XOR	100	data ^ accum
LDA	101	data
STO	110	accum
JMP	111	accum

Creating the ALU Design

Work in the `lab05-operators_enum` directory:

1. Copy just the alu.sv file from lab03-operators.

2. Write the package for the `opcode` enum declaration in `typedefs.sv` package file.

3. Import the package in the ALU and `alu.sv`, and modify the ALU to use the enumerate type.

Testing the ALU Design

1. Check that your package containing the opcode type declarations is imported into `alu_enum_test.sv`.

2. Simulate the testbench and controller design.

   Debug your ALU as required until you see the following message:

   ```
   ALU TEST PASSED
   ```

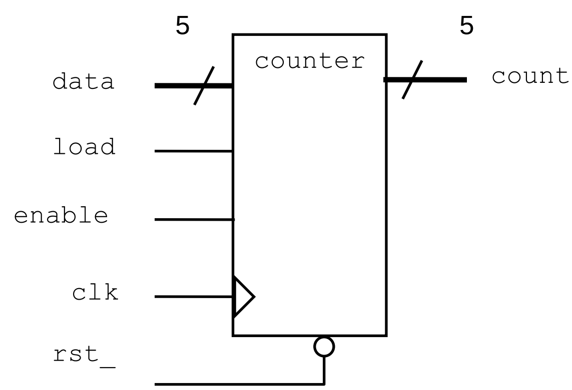
End of Lab

Lab 6 Modeling a Simple Counter

Objective: To use sequential procedural blocks correctly to model a simple counter.

Create a simple loadable, enabled counter design using SystemVerilog and Verilog-2001 constructs and test it using the supplied testbench.

Read the specification first and then follow the instructions in the Creating the Counter Design section in this lab.



Specification

- `data` and `count` are both 5-bit `logic` vectors.
- `rst_` is asynchronous and active low.
- The counter is clocked on the rising edge of `clk`.
  - If `load` is high, the counter is loaded from the input `data`.
  - Otherwise, if `enable` is high, `count` is incremented.
  - Otherwise, `count` is unchanged.

Creating the Counter Design

Work in the `lab06-counter` directory:

1. Create a new file called `counter.sv`, containing a module named `counter`.

2. Write the counter model using the following SystemVerilog and Verilog constructs:
   - Verilog2001 ANSI-C port declarations
   - `always_ff` procedural block
   - `timeunit` and `timeprecision`

Testing the Memory

3. A testbench is provided in the file `counter_test.sv`. Simulate the testbench and counter design.

   You see the following results:

```
time=   0ns clk=1 rst_=x load=x enable=x data=xx count=xx
time=   5ns clk=0 rst_=0 load=x enable=x data=xx count=00
time=  10ns clk=1 rst_=0 load=x enable=x data=xx count=00
time=  15ns clk=0 rst_=1 load=0 enable=1 data=xx count=00
time=  20ns clk=1 rst_=1 load=0 enable=1 data=xx count=01
time=  25ns clk=0 rst_=1 load=0 enable=1 data=xx count=01
time=  30ns clk=1 rst_=1 load=0 enable=1 data=xx count=02
...
time= 105ns clk=0 rst_=1 load=0 enable=1 data=xx count=1e
time= 110ns clk=1 rst_=1 load=0 enable=1 data=xx count=1f
time= 115ns clk=0 rst_=1 load=0 enable=1 data=xx count=1f
time= 120ns clk=1 rst_=1 load=0 enable=1 data=xx count=00
time= 125ns clk=0 rst_=1 load=0 enable=1 data=xx count=00
time= 130ns clk=1 rst_=1 load=0 enable=1 data=xx count=01
COUNTER TEST PASSED
```

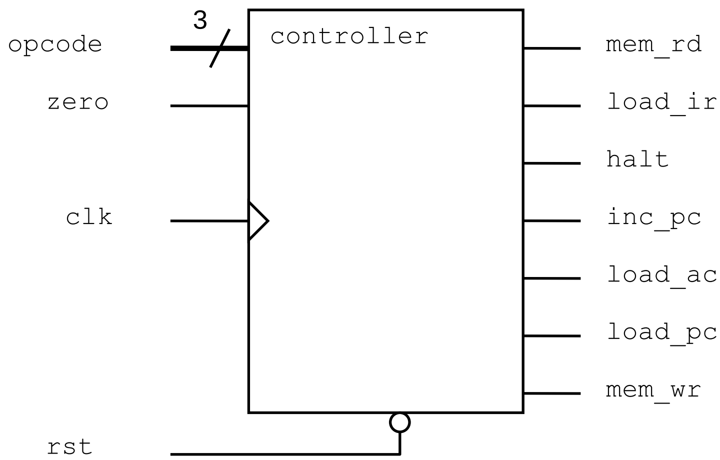
   Debug your counter as required.

End of Lab

Lab 7 Modeling a Sequence Controller

Objective: To use enumerate types, procedural statements, and operators to model a state machine.

Create an FSM Sequence Controller design using SystemVerilog constructs and test it using the supplied testbench. Read the specification first and then follow the instructions in the lab section Creating the Controller Design.



Specification

Work in the `lab07-controller` directory:

- The controller is clocked on the rising edge of `clk`.
- `rst_` is asynchronous and active low.
- `opcode` is a 3-bit `logic` **input** for the CPU operation code as follows:

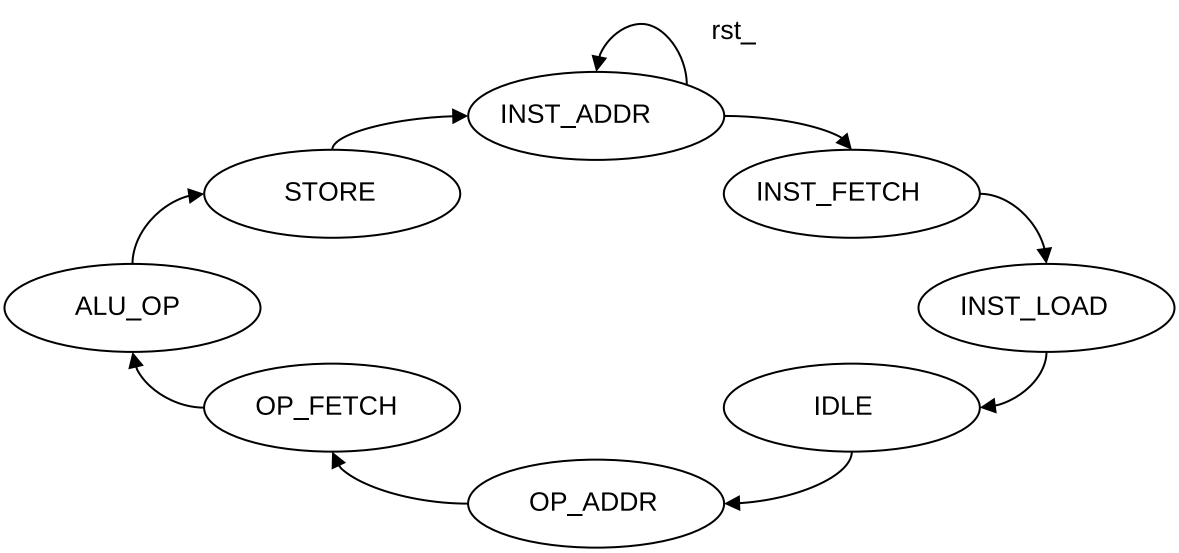
Opcode	Encoding	CPU Operation
HLT	000	Halt
SKZ	001	Skip if zero==1
ADD	010	data + accumulator
AND	011	data & accumulator
XOR	100	data ^ accumulator
LDA	101	Load accumulator
STO	110	Store accumulator
JMP	111	Jump to address

- `zero` is a `logic` **input** which is 1 when the CPU accumulator is zero and 0 otherwise.

- There are 7 `logic` **outputs** as follows:

Output	Function
`mem_rd`	memory read
`load_ir`	load instruction register
`halt`	halt
`inc_pc`	increment program counter
`load_ac`	load accumulator
`load_pc`	load program counter
`mem_wr`	memory write

- The controller has 8 **states**. State transitions are unconditional, i.e., the controller passes through the same 8-state sequence, from `INST_ADDR` to `STORE`, every 8 `clk` cycles. The reset state is `INST_ADDR`.



- The output decode for the controller is as follows:

Outputs	States								Notes
	IST_ADDR	IST_FETCH	IST_LOAD	LE	_ADDR	_FETCH	U_OP	ORE	
m_rd	0	1	1	1	0	ALUOP	ALUOP	ALUOP	UOP = 1 if opcode is ADD, AND, XOR or LDA
ad_ir	0	0	1	1	0	0	0	0	
lt	0	0	0	0	HLT	0	0	0	
c_pc	0	0	0	0	1	0	KZ && zero	JMP	
ad_ac	0	0	0	0	0	0	ALUOP	ALUOP	
ad_pc	0	0	0	0	0	0	JMP	JMP	
m_wr	0	0	0	0	0	0	0	STO	

The controller is a Mealy state machine, so the outputs are a function of the current state and also of the `opcode` and `zero` inputs.

For example, if the controller is in state `ALU_OP`, then the output `inc_pc` is high if `opcode` is `SKZ` *and* `zero` is high.

Creating the Controller Design

Work in the `lab07-controller` directory:

1. Create a new file called typedefs.sv containing a package named typedefs.

2. In the package, declare an enumerated type for the opcode controller input named opcode_t. Declare opcode_t with an explicit logic vector base type and make sure each value has the right encoding.

3. In the same package, declare an enumerated type, named state_t, for the controller state. Use an explicit base type and make sure the encoding is correct. We will need these values in the testbench to help verify the design.

4. Complete the controller definition in the file `control.sv` using SystemVerilog constructs where possible:

   a. Import the package and use your enumerated type declarations for the input `opcode` and state variable(s) of the controller input.

   b. Complete the state generation procedure using enumeration methods.

   c. Generate outputs based on the current phase using the table above. Use `always_comb` and either `unique case` or `unique if` constructs. Be sure to include a default match in the case statement.

Testing the Controller Design

5. Check that your package containing the enumerated type declarations is imported into `control_test.sv`. If you did not name your enumerated types `opcode_t` and `state_t`, then you will need to modify the testbench to use your own type names.

6. Simulate the testbench and controller design. Make sure you compile your package file before compiling any modules which import the package. You do **not** need to compile the `*.pat` files – these are read by the testbench.

   If there is a problem with your design, then you should see something similar to the following output:

   ```
   CONTROLLER TEST FAILED
   {mem_rd,load_ir,halt,inc_pc,load_ac,load_pc,mem_wr}
   is        0000000
   should be 1000000
   state: INST_FETCH   opcode: HLT   zero: 0
   ```

   This tells you that the `mem_rd` output is `0` when it should be `1` in state `INST_FETCH` when the `opcode` input is `HLT` and `zero` input is 0.

   Debug your controller as required until you see the message:

   ```
   CONTROLLER TEST PASSED
   ```

End of Lab

