Verilog Language and Application

Course Version 28.0

Lab Manual **Revision 1.0**

cadence®

Printed in the United States of America.

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

Table of Contents

Verilog Language and Application

Module 1: About This Course

There are no labs for this module.

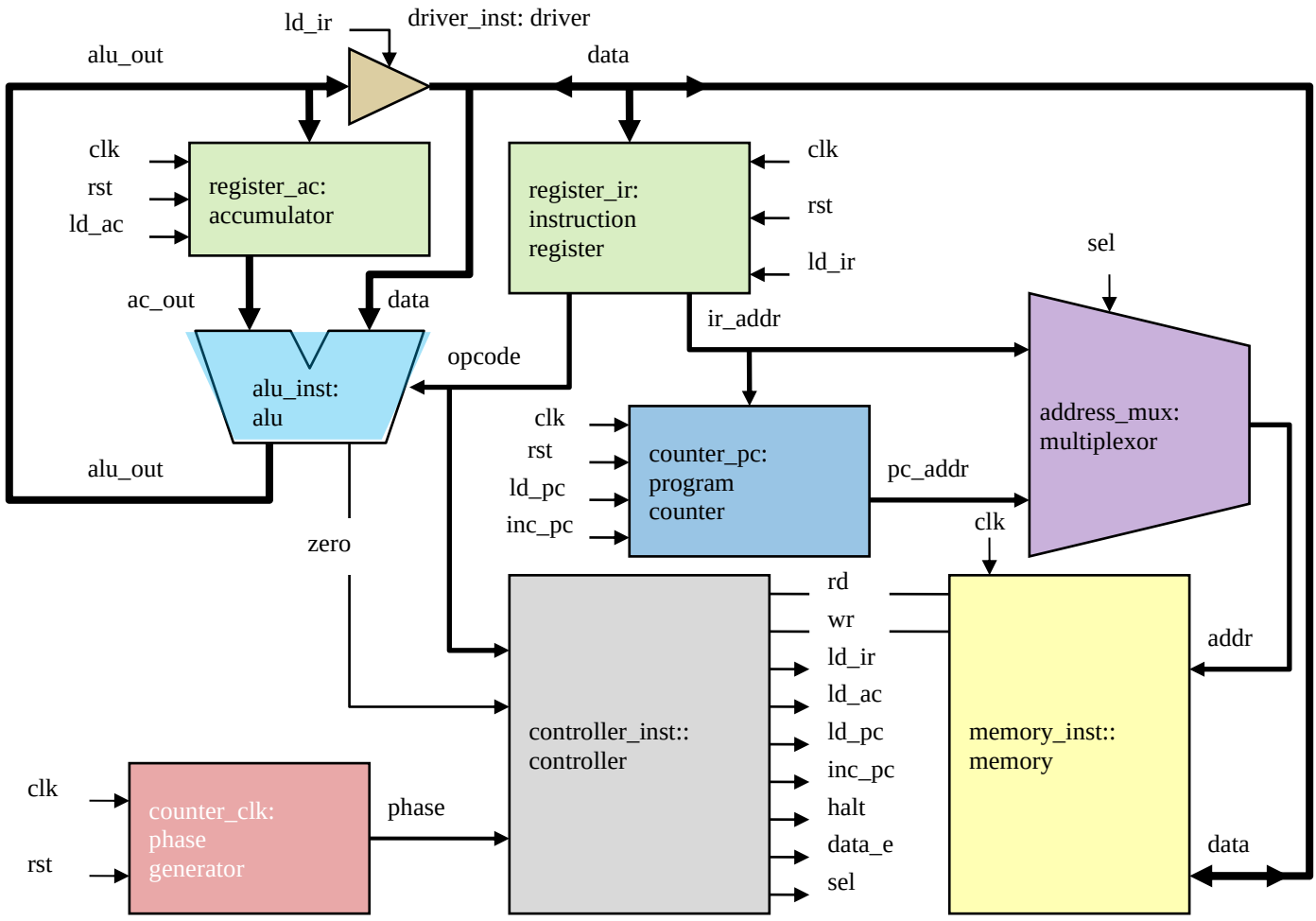
Module 2: Describing Verilog Applications

Lab 2-1 Exploring the VeriRISC CPU Design

Objective: To understand and become familiar with the VeriRISC lab model.

The VeriRISC model is a very-reduced-instruction-set processor coded in the Verilog HDL. Its instruction consists of a three-bit operation code and a five-bit operand. That restricts its instruction set to eight instructions and its address space to 32 locations. To simplify visualization and debug, it implements the fetch-and-execute cycle in eight phases.

As an exercise, you can easily reduce this to two phases, or with a dual-port memory, to one phase. Input to the model is a clock and a reset, and output from the model is a *halt* signal. Tests determine pass or fail status by the number of clocks the program consumes. **Spend some time and understand the architecture of VeriRISC in this lab. Do not worry about implementation and simulation at this point. You will be implementing and testing this architecture in further labs.**



The upcoming labs until Lab 12-1 are based on the above design. In each of these labs, you develop and verify individual components of the VeryRISC processor like Address Multiplexor, Counter, Controller, Register, ALU, etc. In Lab 12-1, you verify the complete design for a few basic instructions under consideration.

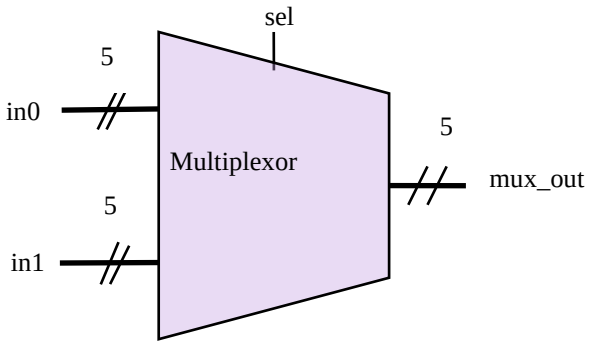
Module 3: Verilog Introduction

Lab 3-1 Modeling an Address Multiplexor

Objective: To use basic Verilog constructs to describe a simple Multiplexor design.

In this lab, you create a simple multiplexor design using Verilog constructs and verify it using the supplied testbench.

You will create the MUX design as per the specification and verify it using the provided testbench. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

- The address multiplexor selects between the instruction address during the instruction fetch phase and the operand address during the instruction execution phase.
- MUX width is parameterized with the default value of 5.
- If `sel` is `1'b0`, input `in0` is passed to the output `mux_out`.
- If `sel` is `1'b1`, input `in1` is passed to the output `mux_out`.

Designing the MUX

1. Change to the *lab3-mux* directory and examine the following files.

multiplexor_test.v	Multiplexor test

2. Create the *multiplexor.v* file, and using your favorite editor, describe the multiplexor module named "*multiplexor"*.

3. Parameterize the multiplexor input and output widths and assign a default value of 5.

Verifying the MUX design

1. Using the provided test module, check your multiplexor design using the following command with Xcelium™.

   ```
   xrun multiplexor.v multiplexor_test.v (Batch Mode)
   ```

   or

   ```
   xrun multiplexor.v multiplexor_test.v -gui -access +rwc ( GUI Mode)
   ```

2. 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
   ```

   You should see the following results.

   ```
   At time 1 sel=0 in0=10101 in1=00000, mux_out=10101
   At time 2 sel=0 in0=01010 in1=00000, mux_out=01010
   At time 3 sel=1 in0=00000 in1=10101, mux_out=10101
   At time 4 sel=1 in0=00000 in1=01010, mux_out=01010
   TEST PASSED
   ```

3. Correct your multiplexor design as needed.

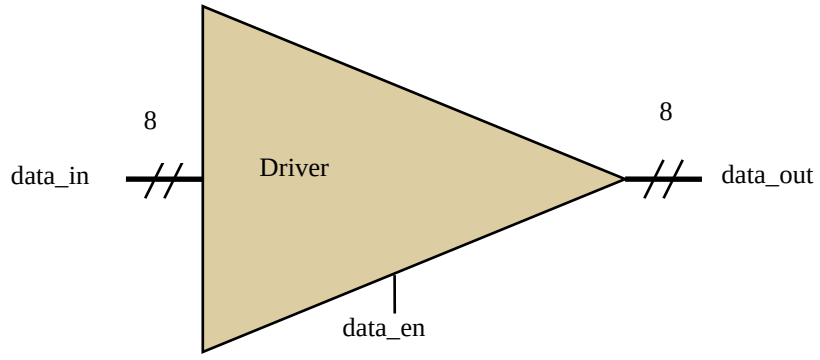
Module 4: Choosing Between Verilog Data Types

Lab 4-1 Modeling a Data Driver

Objective: To use a Verilog literal value to describe a parameterized-width bus driver.

The driver output is equal to the input value when enabled (*data_en* is true) and is high-impedance when not enabled (*data_en* is false).

In this lab, you create a data driver as per the specification and verify it using the provided testbench. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

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

Designing a Data Driver

1. Change to the *lab4-drvr* directory and examine the following files.

driver_test.v	Driver test

2. Create the *driver.v* file, and using your favorite editor, describe the driver module named "*driver*".

3. Parameterize the driver input and output width and assign a default value of 8.

Verifying the Driver Design

1. Using the provided testbench module, check your driver design using the following command with Xcelium™.

   ```
   xrun driver.v driver_test.v (Batch Mode)
   ```

   or

   ```
   xrun driver.v driver_test.v -gui -access +rwc ( GUI Mode)
   ```

2. 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
   ```

   You should see the following results.

   ```
   At time 1 data_en=0 data_in=xxxxxxxx data_out=zzzzzzzz
   At time 2 data_en=1 data_in=01010101 data_out=01010101
   At time 3 data_en=1 data_in=10101010 data_out=10101010
   TEST PASSED
   ```

3. Correct your driver design as needed.

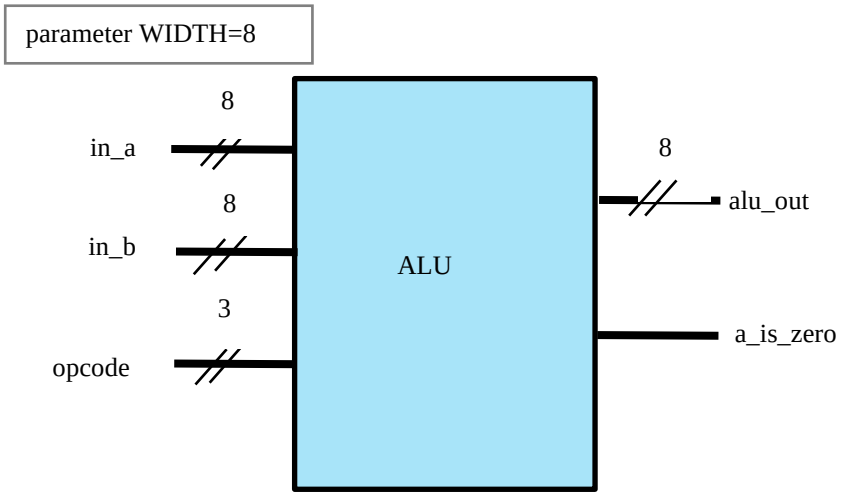
Module 5: Using Verilog Operators

Lab 5-1 Modeling the Arithmetic Logic Unit

Objective: To use Verilog operators to describe a parameterized-width arithmetic logic unit (ALU).

ALU performs arithmetic operations on numbers depending upon the operation encoded in the instruction. This ALU will perform 8 operations on the 8-bit inputs (see table in the S*pecification*) and generate an 8-bit output and single-bit output.

In this lab, you create an ALU design as per the specification and verify it using the provided testbench. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

- `in_a, in_b,` and `alu_out` are all 8-bit long. The `opcode` is a 3-bit value for the CPU operation code, as defined in the following table.
- `a_is_zero` is a single bit asynchronous output with a value of 1 when `in_a` equals 0. Otherwise, `a_is_zero` is 0.
- The output `alu_out` value will depend on the `opcode` value as per the following table.
- To select which of the 8 operations to perform, you will use `opcode` as the selection lines.
- The following table states the opcode/instruction, opcode encoding, operation, and output.

Opcode/ Instruction	Opcode Encoding	Operation	Output
HLT	000	PASS A	in_a => alu_out
SKZ	001	PASS A	in_a => alu_out
ADD	010	ADD	in_a + in_b => alu_out
AND	011	AND	in_a & in_b => alu_out
XOR	100	XOR	in_a ^ in_b => alu_out
LDA	101	PASS B	in_b => alu_out
STO	110	PASS A	in_a => alu_out
JMP	111	PASS A	in_a => alu_out

Designing an ALU

1. Change to the *lab5-alu* directory and examine the following files.

alu_test.v	ALU test
filelist.txt	File listing all modules to simulate

2. Use your favorite editor to create the *alu.v* file. Describe the ALU module named as *alu*.

   a. Parameterize the ALU input and output width so that the instantiating module can specify the width of each instance.

   b. Assign a default value to the parameter.

Verifying the ALU Design

1. Using the provided testbench module, check your ALU design using the following command with Xcelium™.

   ```
   xrun alu.v alu_test.v (Batch Mode)
   ```

   or

   ```
   xrun alu.v alu_test.v -gui -access +rwc ( GUI Mode)
   ```

2. 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
```

You should see the following results.

```
At time 1 opcode=000 in_a=01000010 in_b=10000110 a_is_zero=0
alu_out=01000010
At time 2 opcode=001 in_a=01000010 in_b=10000110 a_is_zero=0
alu_out=01000010
At time 3 opcode=010 in_a=01000010 in_b=10000110 a_is_zero=0
alu_out=11001000
At time 4 opcode=011 in_a=01000010 in_b=10000110 a_is_zero=0
alu_out=00000010
At time 5 opcode=100 in_a=01000010 in_b=10000110 a_is_zero=0
alu_out=11000100
At time 6 opcode=101 in_a=01000010 in_b=10000110 a_is_zero=0
alu_out=10000110
At time 7 opcode=110 in_a=01000010 in_b=10000110 a_is_zero=0
alu_out=01000010
At time 8 opcode=111 in_a=01000010 in_b=10000110 a_is_zero=0
alu_out=01000010
At time 9 opcode=111 in_a=00000000 in_b=10000110 a_is_zero=1
alu_out=00000000
TEST PASSED
```

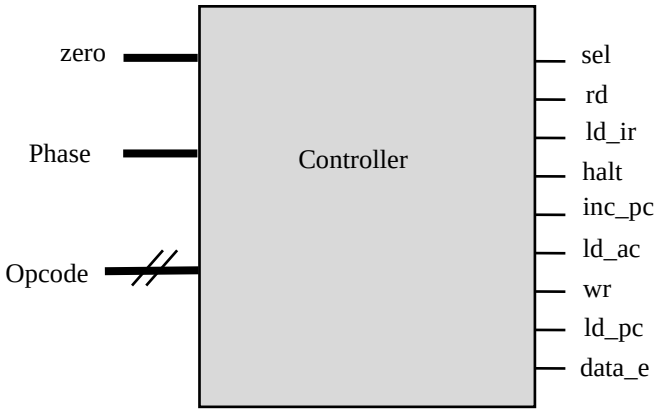
3. Correct your ALU design as needed.

Module 6: Making Procedural Statements

Lab 6-1 Modeling a Controller

Objective: To use the Verilog case statement to describe a controller.

The controller generates all control signals for the VeriRISC CPU. The operation code, fetch-and-execute phase, and whether the accumulator is zero determine the control signal levels.



Specifications

- The controller is clocked on the rising edge of `clk`.
- `rst` is synchronous and active high.
- `zero` is an **input** which is 1 when the CPU accumulator is zero and 0 otherwise.
- `opcode` is a 3-bit **input** for CPU operation, as shown in the following table.

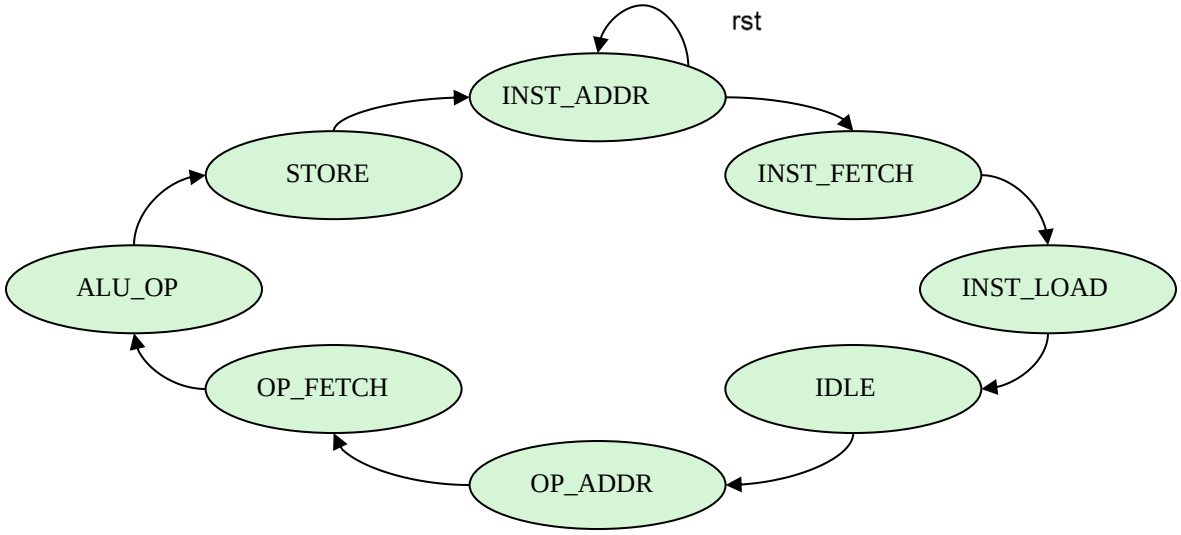
Opcode/ Instruction	Opcode Encoding	Operation	Output
HLT	000	PASS A	in_a => alu_out
SKZ	001	PASS A	in_a => alu_out
ADD	010	ADD	in_a + in_b => alu_out
AND	011	AND	in_a & in_b => alu_out
XOR	100	XOR	in_a ^ in_b => alu_out
LDA	101	PASS B	in_b => alu_out
STO	110	PASS A	in_a => alu_out
JMP	111	PASS A	in_a => alu_out

- There are 7 single-bit **outputs**, as shown in this table.

Output	Function
`sel`	select
`rd`	memory read
`ld_ir`	load instruction register
`halt`	halt
`inc_pc`	increment program counter
`ld_ac`	load accumulator
`ld_pc`	load program counter
`wr`	memory write
`data_e`	data enable

- The controller has a 3-bit phase input with a total of 8 phases processed. Phase transitions are unconditional which means `INST_ADDR, INST_ADDR.....STORE` any phase can come at any time according to the phase; all output signal values will be set according to the phase, which is shown in the table.

- The controller passes through the same 8-phase sequence, from `INST_ADDR` to `STORE`, every 8 `clk` cycles. The reset state is `INST_ADDR`.



- The controller outputs will be decoded w.r.t phase and opcode, as shown in this table.

Outputs	**Phase**						**Notes**		
	INST_ ADDR	INST_ FETCH	INST_ LOAD	IDLE	OP_ ADDR	OP_ FETCH	ALU_ OP	STORE	
`sel`	1	1	1	1	0	0	0	0	`ALU_OP = 1` if opcode is `ADD, AND, XOR` or `LDA`
`rd`	0	1	1	1	0	ALUOP	ALUOP	ALUOP	
`ld_ir`	0	0	1	1	0	0	0	0	
`halt`	0	0	0	0	HALT	0	0	0	
`inc_pc`	0	0	0	0	1	0	SKZ && zero	0	
`ld_ac`	0	0	0	0	0	0	0	ALUOP	
`ld_pc`	0	0	0	0	0	0	JMP	JMP	
`wr`	0	0	0	0	0	0	0	STO	
`data_e`	0	0	0	0	0	0	STO	STO	

Designing a Controller

1. Change to the *lab6-ctlr* directory and examine the following files.

controller_test.v	Controller test
filelist.txt	File listing all modules to simulate

2. Use your favorite editor to create the *controller.v* file and describe the controller module named "*control* ".

3. Declare a reg for each of these intermediate terms and establish their value immediately before entering the case statement.

Verifying the Controller Design

1. Using the provided test module, check your controller design using the following command with Xcelium™.

```
xrun controller.v controller_test.v (Batch Mode)
```

or

```
xrun controller.v controller_test.v -gui -access +rwc ( GUI Mode)
```

2. 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
```

You should see the following results.

```
Testing opcode HLT phase 0 1 2 3 4 5 6 7
Testing opcode SKZ phase 0 1 2 3 4 5 6 7
Testing opcode ADD phase 0 1 2 3 4 5 6 7
Testing opcode AND phase 0 1 2 3 4 5 6 7
Testing opcode XOR phase 0 1 2 3 4 5 6 7
Testing opcode LDA phase 0 1 2 3 4 5 6 7
Testing opcode STO phase 0 1 2 3 4 5 6 7
Testing opcode JMP phase 0 1 2 3 4 5 6 7
TEST PASSED
```

3. Correct your controller design as needed.

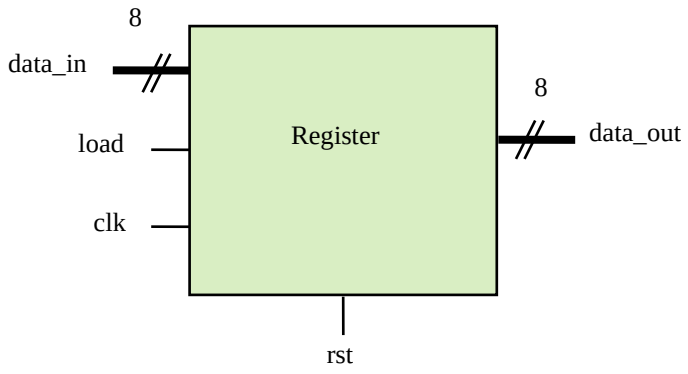
Module 7: Using Blocking and Nonblocking Assignments

Lab 7-1 Modeling a Generic Register

Objective: To use nonblocking assignments to describe a register.

The VeriRISC CPU contains an accumulator register and an instruction register. One generic register definition can serve both purposes.

In this lab, you create a simple register design as per the specification and verify it using the provided testbench. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

- `data_in` and `data_out` are both 8-bit signals.
- `rst` is synchronous and active high.
- The register is clocked on the rising edge of `clk`.
- If `load` is high, the input data is passed to the output `data_out`.
- Otherwise, the current value of `data_out` is retained in the register.

Designing a Register

1. Change to the *lab7-reg* directory and examine the following files.

register_test.v	Register test
filelist.txt	File listing all modules to simulate

2. Use your favorite editor to create the *register.v* file and to describe the register module named "*register* ".

   a. Parameterize the register data input and output width so that the instantiating module can specify the width of each instance.

   b. Assign a default value to the parameter as 8.

   c. Write the register model using the Verilog `always` procedural block.

Verifying the Register Design

1. Using the provided test module, check your register design using the following command with Xcelium™.

   ```
   xrun register.v register_test.v (Batch Mode)
   ```

   or

   ```
   xrun register.v register_test.v -gui -access +rwc ( GUI Mode)
   ```

2. 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
   ```

   You should see the following results.

   ```
   At time 20 rst=0 load=1 data_in=01010101 data_out=01010101
   At time 30 rst=0 load=1 data_in=10101010 data_out=10101010
   At time 40 rst=0 load=1 data_in=11111111 data_out=11111111
   At time 50 rst=1 load=1 data_in=11111111 data_out=00000000
   TEST PASSED
   ```

3. Correct your register design as needed.

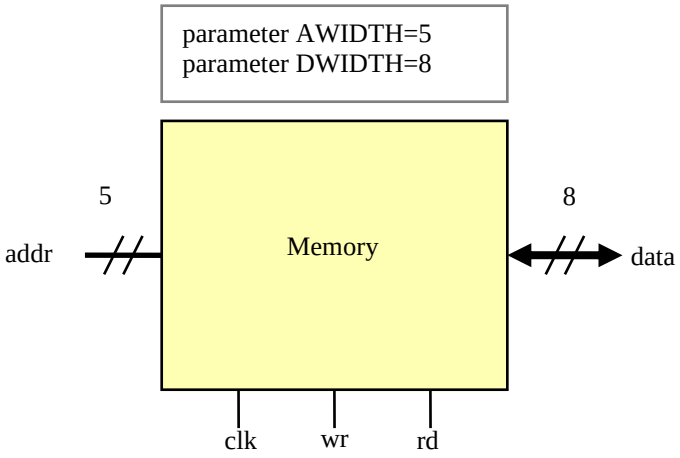
Module 8: Using Continuous and Procedural Assignments

Lab 8-1 Modeling a Single-Bidirectional-Port Memory

Objective: To use continuous and procedural assignments to describe a memory.

The VeriRISC CPU uses the same memory for instructions and data. The memory has a single bidirectional data port and separate write and read control inputs. It cannot perform simultaneous write and read operations.

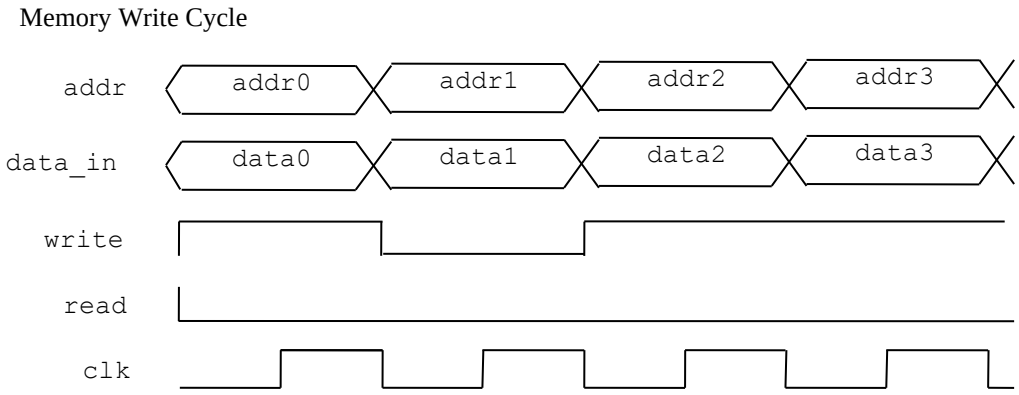
In this lab, you create a simple register design as per the specification and verify it using the provided testbench. Please make sure to use the same variable name as the ones shown in block diagrams.



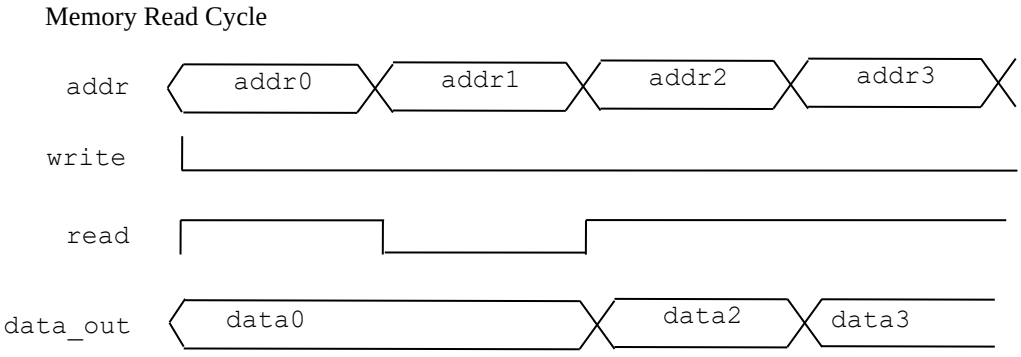
Specifications

- `addr` is parameterized to 5 and `data` is parameterized to 8.
- `wr` (write) and `rd` (read) are single-bit input signals.
- The memory is clocked on the rising edge of `clk`.
- Analyze the memory read and write operation timing diagram, as shown in the following graphic.

- **Memory write:** `data_in` is written to memory[addr] on the positive edge of `clk` when `wr` (write) =1.



- **Memory read**: `data_out` is assigned from memory[addr] when `rd` (read) =1.



Designing a Memory

1. Change to the *lab8-mem* directory and examine the following files.

memory_test.v	Memory test

2. Use your favorite editor to create the *memory.v* file and to describe the memory module named "*memory* ".

3. Parameterize the address and data widths so that the instantiating module can specify the width and depth of each instance.

4. Assign default values to the parameters.

5. Write data on the active clock edge when the *wr* input is true and drive data when the *rd* input is true.

6. Perform the write operation in a procedural block and perform the read operation as a continuous assignment.

Verifying the Memory Design

1. Using the provided test module, check your memory design using the following command with Xcelium™.

   ```
   xrun memory.v memory_test.v (Batch Mode)
   ```

   or

   ```
   xrun memory.v memory_test.v -gui -access +rwc ( GUI Mode)
   ```

2. 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
   ```

   You should see the following results.

   ```
   At time 370 addr=11111 data=00000000
   At time 380 addr=11110 data=00000001
   At time 390 addr=11101 data=00000010
   At time 400 addr=11100 data=00000011
   At time 410 addr=11011 data=00000100
   At time 420 addr=11010 data=00000101
   At time 430 addr=11001 data=00000110
   At time 440 addr=11000 data=00000111
   At time 450 addr=10111 data=00001000
   At time 460 addr=10110 data=00001001
   At time 470 addr=10101 data=00001010
   At time 480 addr=10100 data=00001011
   At time 490 addr=10011 data=00001100
   At time 500 addr=10010 data=00001101
   At time 510 addr=10001 data=00001110
   At time 520 addr=10000 data=00001111
   At time 530 addr=01111 data=00010000
   At time 540 addr=01110 data=00010001
   At time 550 addr=01101 data=00010010
   At time 560 addr=01100 data=00010011
   At time 570 addr=01011 data=00010100
   ```



```
At time 580 addr=01010 data=00010101
At time 590 addr=01001 data=00010110
At time 600 addr=01000 data=00010111
At time 610 addr=00111 data=00011000
At time 620 addr=00110 data=00011001
At time 630 addr=00101 data=00011010
At time 640 addr=00100 data=00011011
At time 650 addr=00011 data=00011100
At time 660 addr=00010 data=00011101
At time 670 addr=00001 data=00011110
TEST PASSED
```

3. Correct your memory design as needed.

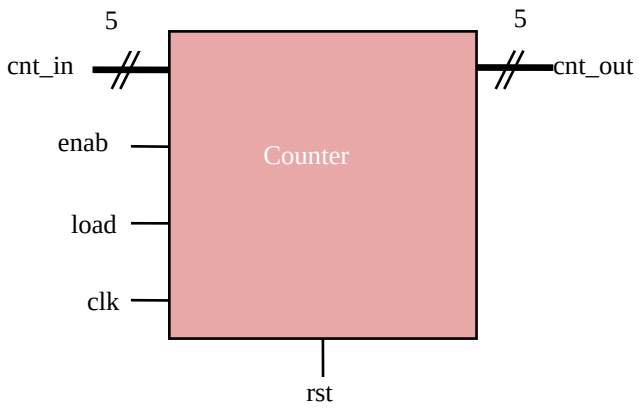
Module 9: Understanding the Simulation Cycle

Lab 9-1 Modeling a Generic Counter

Objective: To use blocking and nonblocking assignments to describe a counter.

The VeriRISC CPU contains a program counter and a phase counter. One generic counter definition can serve both purposes.

In this lab, you create a simple register design as per the specification and verify it using the provided testbench. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

- The counter is clocked on the rising edge of `clk`.
- `rst` is active high.
- `cnt_in` and `cnt_out` are both 5-bit signals.
- If `rst` is high, output will become *zero*.
- If `load` is high, the counter is loaded from the input `cnt_in`.
- Otherwise, if `enab` is high, `cnt_out` is incremented, and `cnt_out` is unchanged.

Designing a Generic Counter

1. Change to the *lab9-cntr* directory and examine the following files.

counter_test.v	Counter test

2. Use your favorite editor to create the *counter.v* file and to describe the counter module named as "*counter* ".

3. Parameterize the counter data input and output width so that the instantiating module can specify the width of each instance. Assign a default value to the parameter.

4. Describe the counter behavior in separate combinational and sequential procedures.

Verifying the Counter Design

1. Using the provided test module, check your counter design using the following command with Xcelium™.

```
xrun counter.v counter_test.v (Batch Mode)
```

or

```
xrun counter.v counter_test.v -gui -access +rwc ( GUI Mode)
```

2. 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
```

You should see the following results.

```
At time 20 rst=0 load=1 enab=1 cnt_in=10101 cnt_out=10101
At time 30 rst=0 load=1 enab=1 cnt_in=01010 cnt_out=01010
At time 40 rst=0 load=1 enab=1 cnt_in=11111 cnt_out=11111
At time 50 rst=1 load=1 enab=1 cnt_in=11111 cnt_out=00000
At time 60 rst=0 load=1 enab=1 cnt_in=11111 cnt_out=11111
At time 70 rst=0 load=0 enab=1 cnt_in=11111 cnt_out=00000
TEST PASSED
```

3. Correct your counter description as needed.

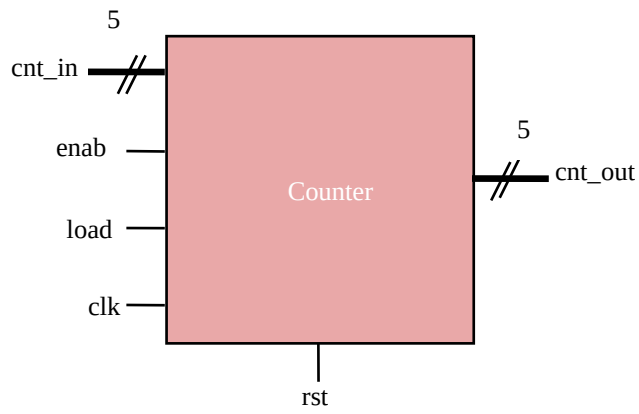
Module 10: Using Functions and Tasks

Lab 10-1 Modeling the Counter Using Functions

Objective: To encapsulate counter design combinational behaviors in a function.

You typically use a function to encapsulate an operation performed multiple times with multiple different sets of operands. Encapsulating functionality reduces code “bloat” to make it more understandable and thus more reusable.

In this lab, you create a counter design as per the specification using functions to practice the construct and verify it using the provided testbench. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

- The counter is clocked on the rising edge of `clk`.
- `rst` is active high.
- `cnt_in` and `cnt_out` are both 5-bit signals.
- If `rst` is high, output will become *zero*.
- If `load` is high, the counter is loaded from the input `cnt_in`.
- Otherwise, if `enab` is high, `cnt_out` is incremented and `cnt_out` is unchanged.

Designing a Counter

1. Change to the *lab10-func* directory and examine the following file.

counter_test.v	Counter test
counter.v	Counter module (incomplete)

2. Use your favorite editor to modify the *counter.v* file name it as "*counter*".

3. Describe the combinational logic of the counter using functional block. You can use an 'if' construct to model the counter combinational block.

Verifying the Counter Design

1. Using the test module, to test your counter design using the following command with Xcelium™.

```
xrun counter.v counter_test.v (Batch Mode)
```

or

```
xrun counter.v counter_test.v -gui -access +rwc ( GUI Mode)
```

2. 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
```

You should see the following results.

```
At time 20 rst=0 load=1 enab=1 cnt_in=10101 cnt_out=10101
At time 30 rst=0 load=1 enab=1 cnt_in=01010 cnt_out=01010
At time 40 rst=0 load=1 enab=1 cnt_in=11111 cnt_out=11111
At time 50 rst=1 load=1 enab=1 cnt_in=11111 cnt_out=00000
At time 60 rst=0 load=1 enab=1 cnt_in=11111 cnt_out=11111
At time 70 rst=0 load=0 enab=1 cnt_in=11111 cnt_out=00000
TEST PASSED
```

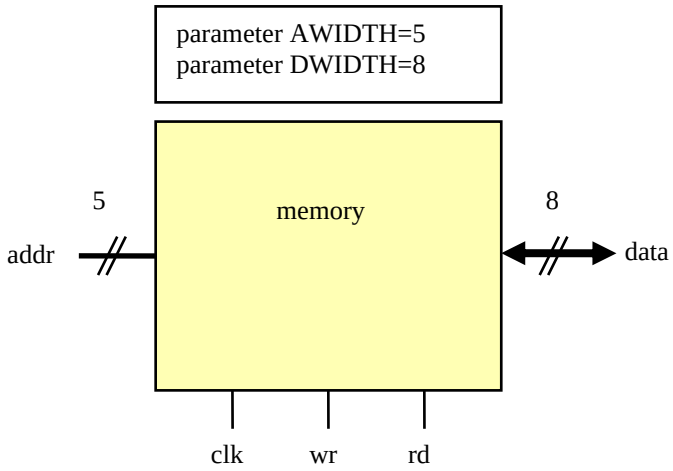
3. Correct your counter description as needed.

Lab 10-2 Modeling the Memory Test Block Using Tasks

Objective: To encapsulate procedural behaviors in memory test using tasks.

You typically use a function to encapsulate an operation performed multiple times with multiple different sets of operands. Encapsulating functionality reduces code “bloat” to make it more understandable and thus more reusable.

In this lab, you create a memory design as per the specification using tasks to practice the construct and verify it using the provided testbench. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

- `addr` is parameterized to 5 and `data` is parameterized to 8.
- `wr` (write) and `rd` (read) are single-bit input signals.
- The memory is clocked on the rising edge of `clk`.
- Analyze memory read and write operation timing diagram given in *Lab 9-1*.

Designing Memory Using Tasks

1. Change to the *lab10-task* directory and examine the following files.

memory.v	Memory module
memory_test.v	Memory test (incomplete)

2. Use your favorite editor to modify the *memory_test.v* file to code the write procedure and read procedure using *tasks*.

3. In the memory test block, call the appropriate read and write tasks to verify the memory block.

   Note: The statement sets that you code read and write procedures given the following task calls.

   ```
   For write: wr=1; rd=0; memory_test.addr=addr; rdata=data; @(negedge
      clk);

   For read:  wr=0; rd=1; memory_test.addr=addr; rdata='bz; @(negedge
      clk) expect(data);
   ```

Verifying the Counter Design

1. Using the provided test module, check your counter design using the following command with Xcelium™.

   ```
   xrun memory.v memory_test.v (Batch Mode)
   ```

   or

   ```
   xrun memory.v memory_test.v -gui -access +rwc ( GUI Mode)
   ```

2. 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
   ```

   You should see the following results.

   ```
   330 addr=11111, exp_data= 00000000, data=00000000
   340 addr=11110, exp_data= 00000001, data=00000001
   350 addr=11101, exp_data= 00000010, data=00000010
   360 addr=11100, exp_data= 00000011, data=00000011
   370 addr=11011, exp_data= 00000100, data=00000100
   380 addr=11010, exp_data= 00000101, data=00000101
   390 addr=11001, exp_data= 00000110, data=00000110
   400 addr=11000, exp_data= 00000111, data=00000111
   410 addr=10111, exp_data= 00001000, data=00001000
   420 addr=10110, exp_data= 00001001, data=00001001
   430 addr=10101, exp_data= 00001010, data=00001010
   440 addr=10100, exp_data= 00001011, data=00001011
   450 addr=10011, exp_data= 00001100, data=00001100
   460 addr=10010, exp_data= 00001101, data=00001101
   470 addr=10001, exp_data= 00001110, data=00001110
   ```

```
480 addr=10000, exp_data= 00001111, data=00001111
490 addr=01111, exp_data= 00010000, data=00010000
500 addr=01110, exp_data= 00010001, data=00010001
510 addr=01101, exp_data= 00010010, data=00010010
520 addr=01100, exp_data= 00010011, data=00010011
530 addr=01011, exp_data= 00010100, data=00010100
540 addr=01010, exp_data= 00010101, data=00010101
550 addr=01001, exp_data= 00010110, data=00010110
560 addr=01000, exp_data= 00010111, data=00010111
570 addr=00111, exp_data= 00011000, data=00011000
580 addr=00110, exp_data= 00011001, data=00011001
590 addr=00101, exp_data= 00011010, data=00011010
600 addr=00100, exp_data= 00011011, data=00011011
610 addr=00011, exp_data= 00011100, data=00011100
620 addr=00010, exp_data= 00011101, data=00011101
630 addr=00001, exp_data= 00011110, data=00011110
TEST PASSED
```

3. Correct your memory test as needed.

Module 11: Directing the Compiler

Lab 11-1 Verifying the VeriRISC CPU Design

Objective: To test or verify the VeriRISC CPU.

At the *behavioral* level of abstraction, you code behavior with no regard for an actual hardware implementation, so you can utilize any Verilog construct. The behavioral level of abstraction is useful for exploring design architecture and especially useful for developing test benches. As both uses are beyond the scope of this training module, it only briefly introduces testbench concepts.

Read the specification first and then follow the instructions provided.

Specifications

The CPU architecture is as follows:

- The Program Counter (`counter`) provides the program address.
- The MUX (`mux`) selects between the program address or the address field of the instruction.
- The Memory (`memory`) accepts data and provides instructions and data.
- The Instruction Register (`register`) accepts instructions from the memory.
- The Accumulator Register (`register`) accepts data from the ALU.
- The ALU (`alu`) accepts memory and accumulator data, and the opcode field of the instruction, and provides new data to the accumulator and memory.

If all components of the CPU are working properly, it will:

1. Fetch an instruction from the memory.
2. Decode the instruction.
3. Fetch a data operand from memory if required by the instruction.
4. Execute the instruction, processing mathematical operations, if required.
5. Store results back into either the memory or the accumulator. This process is repeated for every instruction in a program until an `HLT` instruction is found.

Verifying the VeriRISC CPU Design

1. Change to the *lab11-risc* directory and examine the following files.

risc.v	RISC model
risc_test.v	RISC test (incomplete)
files.txt	List of RISC sources
CPUtest1.txt CPUtest2.txt CPUtest3.txt	Diagnostic Test programs
restore.tcl	Simulation setup files

2. Review the supplied testbench in the file `risc_test.sv`. The testbench verifies your CPU design using three diagnostic programs as follows.

   CPUtest1.txt (Basic Test) – This diagnostic program tests the basic instruction set of the VeriRisc system. If the system executes each instruction correctly, then it should halt when the HLT instruction at address 17(hex) is executed. If the system halts at any other location, then an instruction did not execute properly. Refer to the comments in this file to see which instruction failed.

   CPUtest2.txt (Advanced Test) – This diagnostic program tests the advanced instruction set of the VeriRisc system. If the system executes each instruction correctly, then it should halt when the HLT instruction at address 10(hex) is executed. If the system halts at any other location, then an instruction did not execute properly. Refer to the comments in this file to see which instruction failed.

   CPUtest3.txt (Fibonacci Calculator) – This is an actual program that calculates the Fibonacci number sequence from 0 to 144. The Fibonacci number sequence is a series of numbers in which each number in the sequence is the sum of the preceding two numbers (i.e., 0, 1, 1, 2, 3, 5, 8, 13 ...). If all the instructions execute correctly, the CPU will encounter an HLT instruction at Program Counter address 0x0C. If the CPU halts at some other address, then an instruction did not execute properly. Refer to the comments in this file to see which instruction failed.

3. Use your favorite editor to modify the *risc_test.v* file to complete the RISC verification environment to verify the *JMP* instructions only. In the file comments are mentioned where code needs to be added; also, you can refer to the SKZ instruction.

   Please follow the comments in the *risc_test.v* file.

4. Using the following command with Xcelium, simulate the design and testbench.

   ```
   xrun -f files.txt (Batch Mode)
   ```

   You should see the following results:

   ```
   Testing reset
   Testing HLT instruction
   Testing JMP instruction
   Testing SKZ instruction
   Testing LDA instruction
   Testing STO instruction
   Testing AND instruction
   Testing XOR instruction
   Testing ADD instruction
   Doing test CPUtest1.txt
   Doing test CPUtest2.txt
   Doing test CPUtest3.txt
   TEST PASSED
   ```

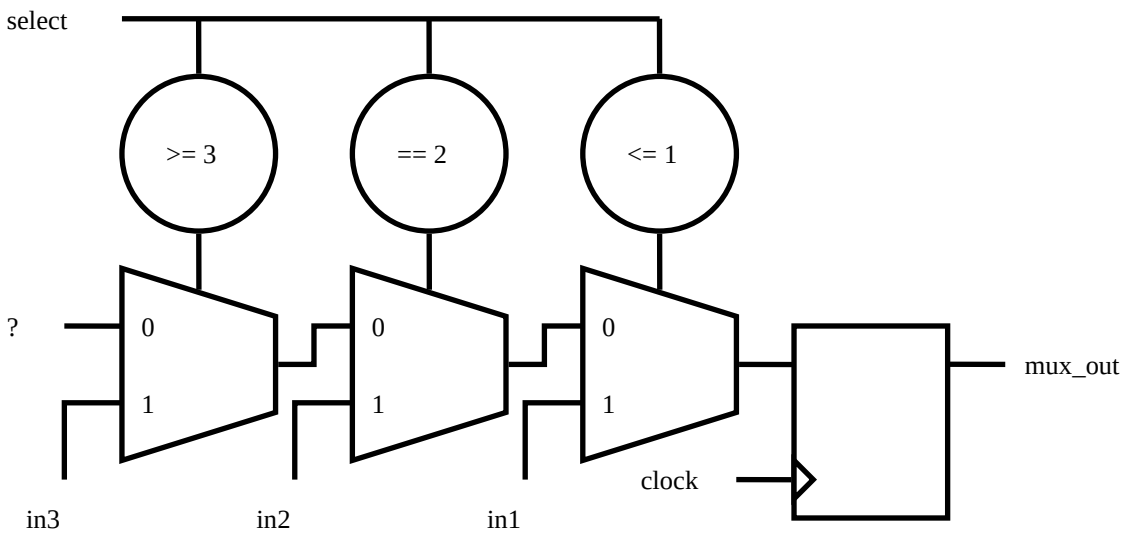
5. Correct your RISC test modifications as needed.

Module 12: Introducing the Process of Synthesis

Lab 12-1 Exploring the Synthesis Process

Objective: To briefly observe the synthesis process and examine the results.

In this lab, you synthesize a small multiplexor model and examine the synthesis results



Important Information

- The multiplexor model conditionally uses either a *case* statement or an *if* statement.
- If you specify neither of them, then it uses an *if* statement. If you inadvertently specify both, it uses a *case* statement.

Synthesizing a Generic Counter Design

1. Change to the *lab12-muxsynth* directory and examine the following files.

mux.v	Multiplexor model
mux_test.v	Multiplexor test
genus_shell.tcl	Synthesis script

2. Verify the RTL model using the the commands with Xcelium™.

   ```
   xrun mux.v mux_test.v (Batch Mode)
   ```

   You should see the below results.

   ```
   time=15 select=00 in1=0 in2=x in3=x mux_out=0
   time=25 select=00 in1=1 in2=x in3=x mux_out=1
   time=35 select=01 in1=1 in2=x in3=x mux_out=1
   time=45 select=01 in1=0 in2=x in3=x mux_out=0
   time=55 select=10 in1=x in2=0 in3=x mux_out=0
   time=65 select=10 in1=x in2=1 in3=x mux_out=1
   time=75 select=11 in1=x in2=x in3=1 mux_out=1
   time=85 select=11 in1=x in2=x in3=0 mux_out=0
   TEST PASSED
   ```

3. Synthesize the RTL model by using the Genus™ Synthesis Solution tool.

   ```
   genus -f genus_shell.tcl
   ```

   Check to see that the *synthesis succeeded*.

   The script writes a pre-synthesis netlist and a post-synthesis netlist.

4. In your favorite editor, examine the *mux.vs* pre-synthesis netlist and attempt to correlate its contents with the multiplexor behavioral description.

   How many multiplexors does it contain?

   Answer: _____

   How many operators does it contain?

   Answer: _____

   How many latches does it contain?

   Answer: _____

5. In your favorite editor, examine the *mux.vg* post-synthesis netlist and attempt to correlate its contents with the multiplexor behavioral description.

   How many multiplexors does it contain?

   Answer: _____

   How many operators does it contain?

   Answer: _____

   How many latches does it contain?

   Answer: _____

6. Verify the gate-level model using the following commands with Xcelium.

   ```
   xrun mux.vg mux_test.v -v ../tutorial.v -vlogext vg
   ```

   You should see TEST PASSED with similar results as above.

7. Save the *mux.vs* and *mux.vg* as *mux_ifdef.vs* and *mux_ifdef.vg* for future comparison with the USE_CASE version.

8. Again, verify the RTL model but this time use the following commands with Xcelium.

   ```
   xrun mux.v mux_test.v -define USE_CASE
   ```

   You should see TEST PASSED with similar results as above.

9. Synthesize the design again and examine the pre-synthesis and post-synthesis netlists.

   *For this model, does coding style significantly affect the **pre**-synthesis netlist?*

   Answer: _____

   *For this model, does coding style significantly affect the **post**-synthesis netlist?*

   Answer: _____

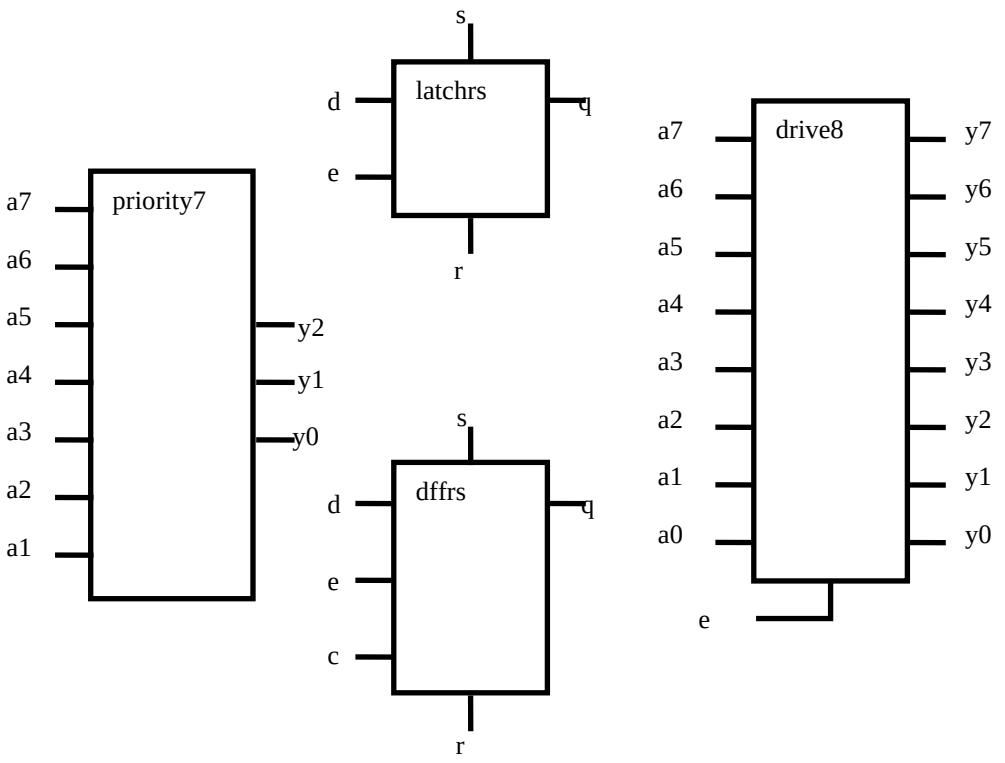
10. Optionally verify the gate-level model again.

Module 13: Coding RTL for Synthesis

Lab 13-1 Using a Component Library

Objective: To demonstrate mastery of basic coding styles and to code and synthesize some representative models.

In this lab, you will code and synthesize some representative models.



Synthesizing a Generic Modules

1. Change to the *lab13-lib* directory and examine the following files.

lib.v	Model library (incomplete)
lib_test.v	Model library tests
genus_shell.tcl	Synthesis script

Note: To facilitate your development effort, the library and tests can conditionally compile and test individual models. You specify the models and tests to compile by defining some combination of the *priority7*, *latchrs*, *dffrs*, and *drive8* macros on the invocation command line, as described in the previous lab. If you do not specify any, then all are compiled and tested.

2. In your favorite editor, modify the *lib.v* file to complete the model definitions. Set and reset are asynchronous and active low. You can elect to individually complete and test each model.

   Note: From the latch template, the synthesis tool cannot infer an asynchronous set or reset. The latch model includes the *async_set_reset* pragma to specify the signals to directly connect to the component's asynchronous set and reset pins.

3. Verify the RTL model(s) using the following commands with Xcelium™.

   a. Fill in the macro name (or omit the option to test all models).

   ```
   xrun lib.v lib_test.v -define macrolib (Batch Mode)
   ```

   You should see the below:

   ```
   TEST PASSED - DRIVER
   TEST PASSED - PRIORITY
   TEST PASSED - LATCH
   TEST PASSED - DFF
   ```

4. Correct your models until all pass their tests.

5. When all models pass their test, synthesize the RTL models by using the Genus™ Synthesis Solution tool.

   ```
   genus -f genus_shell.tcl
   ```

   a. The script writes a separate post-synthesis netlist for each model.

   b. Check to see that the *synthesis succeeded*.

   c. Correct your models until all can synthesize.

6. Verify the gate-level models using the following commands with Xcelium.

   ```
   xrun *.vg lib_test.v -v ../tutorial.v -vlogext vg -define macrolib
   ```

   a. Fill in the macro name (or omit the option to test all models).

b. You should see the following:

```
TEST PASSED - DRIVER
TEST PASSED - PRIORITY
TEST PASSED - LATCH
TEST PASSED - DFF
```

c. If any model fails its test, then you can ask the instructor for help or try again after you study the lecture module *Avoiding Simulation Mismatches*.

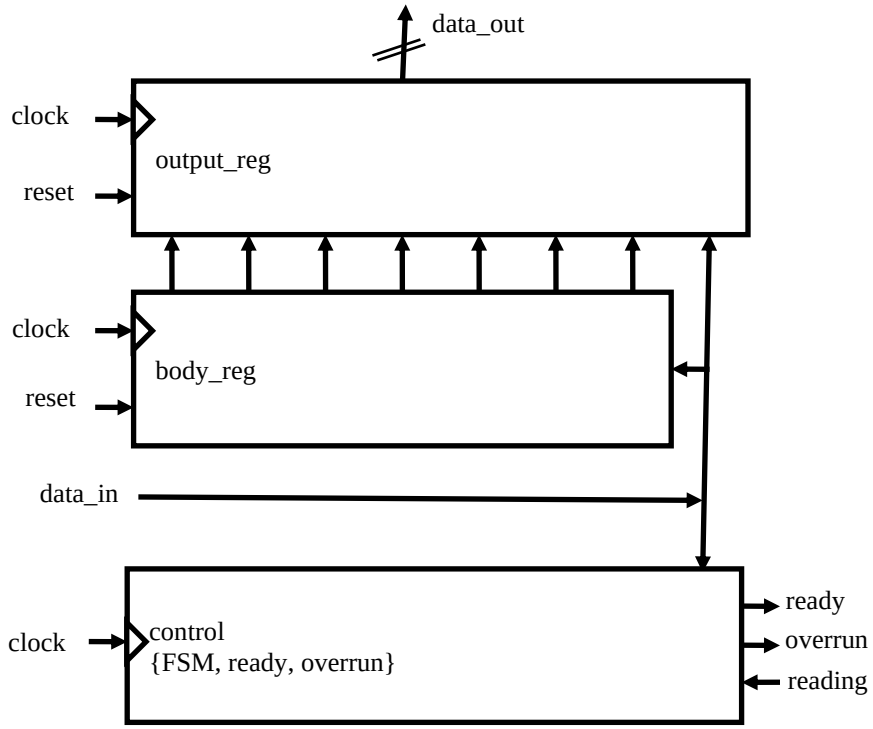
Module 14: Designing Finite State Machines

Lab 14-1 Coding State Machines in Multiple Styles

Objective: To code state machines in different styles for synthesis.

In this lab, you code the serial-to-parallel interface receiver as an FSM in different styles. Instead of shifting the matching character, the FSM steps to the next state upon receiving a correct match bit and returns to a previous state while receiving an incorrect match bit.

Read the specification first and then follow the instructions in this lab. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

- `data_in`, `reading`, `clk`, and `reset` are single-bit input signals.
- `data_out` is an 8-bit output signal, whereas `ready` and `overrun` are 1-bit output.
- The design is clocked on the rising edge of `clk`.
- Every serial bit transmission has a preamble that needs to be sent before sending the character. For this lab, the preamble value is fixed to 8'hA5.
- After the preamble, the `ready` signal will be high.

Designing a Finite State Machine

1. Change to the *lab14-fsm* directory and examine the files.

2. Write down the following three FSM styles.

   a. A single sequential block (as the *rcvr.v* file partially codes).

   b. Move the “next state” encoding to a separate combinational block.

   c. Also, move all non-FSM registers to a separate sequential block.

3. In your favorite editor, modify the *rcvr.v* file to comply with the coding style.

Verifying the Finite State Machine Design

1. Verify the RTL model using the following commands with Xcelium™.

   ```
   xrun rcvr.v rcvr_test.v (Batch Mode)
   ```

2. Verify that you see the following message.

   ```
   Message sending: I Love Verilog
   At time 35: Put character "I"
   At time 39: Got character "I"
   At time 77: Put character " "
   At time 95: Got character " "
   At time 117: Put character "L"
   At time 127: Got character "L"
   At time 153: Put character "o"
   At time 175: Got character "o"
   At time 193: Put character "v"
   At time 207: Got character "v"
   At time 231: Put character "e"
   At time 257: Got character "e"
   At time 275: Put character " "
   At time 297: Got character " "
   At time 309: Put character "V"
   At time 313: Got character "V"
   At time 349: Put character "e"
   At time 365: Got character "e"
   At time 391: Put character "r"
   ```

```
At time 423: Got character "r"
At time 425: Put character "i"
At time 433: Got character "i"
At time 459: Put character "l"
At time 487: Got character "l"
At time 495: Put character "o"
At time 507: Got character "o"
At time 537: Put character "g"
At time 563: Got character "g"
Message received: I Love Verilog
TEST DONE
```

3. Correct your model until it passes the test.

4. When all models pass their test, synthesize the RTL model using the Genus™ Synthesis Solution.

   ```
   genus -f genus_shell.tcl
   ```

5. Check to see that the *synthesis succeeded*. Correct your model until it is synthesized.

6. Verify the gate-level model using the following commands with Xcelium.

   ```
   xrun rcvr.vg rcvr_test.v -v ../tutorial.v -vlogext vg
   ```

7. Check to see the messages I Love Verilog and TEST PASSED as above. If the model fails its test, then you can ask the instructor for help or try again after you study the lecture module *Avoiding Simulation Mismatches*.

8. Correct your design as needed.

Module 15: Avoiding Simulation Mismatches

There are no labs for this module.

Module 16: Managing the RTL Coding Process

There are no labs for this module.

Module 17: Managing the Logic Synthesis Process

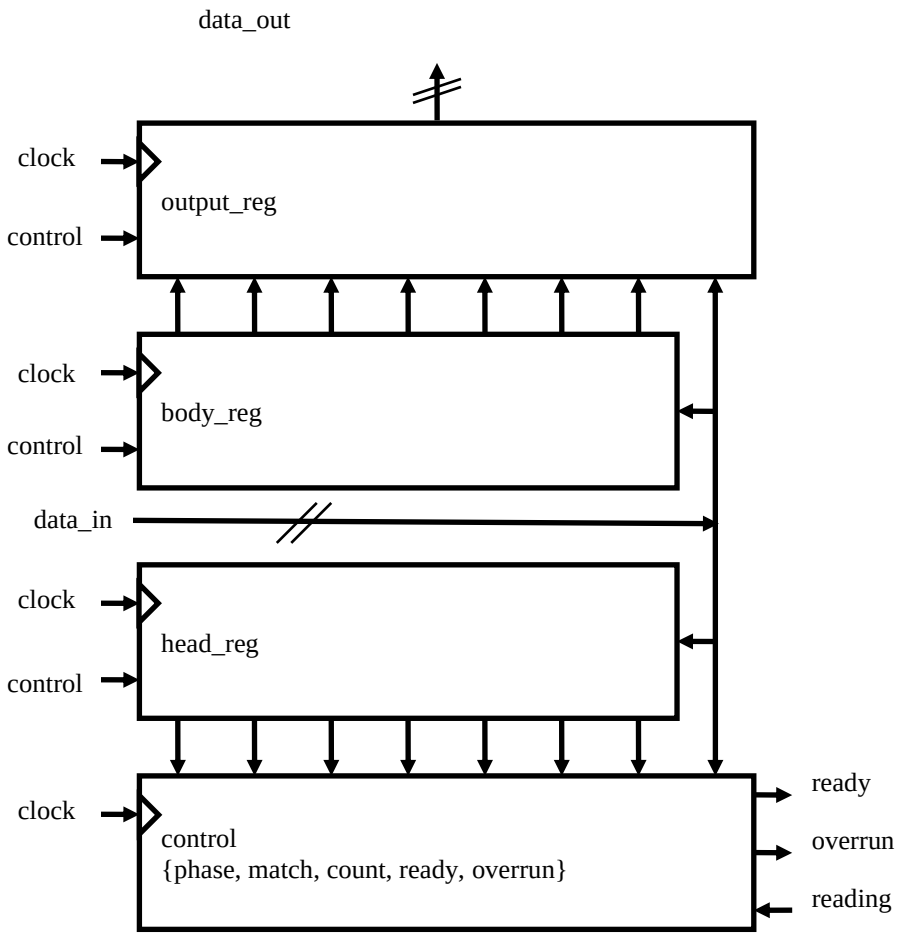
There are no labs in this module.

Module 18: Coding and Synthesizing an Example Verilog Design

Lab 18-1 Coding a Serial-to-Parallel Interface Receiver

Objective: To code a moderately difficult Verilog RTL design for synthesis.

In this lab, you code a moderately difficult Verilog design for synthesis. Read the specification first and then follow the instructions in the lab section. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

- `data_in`, `reading`, `clk`, and `reset` are single-bit input signals.
- `data_out` is an 8-bit output signal, whereas `ready` and `overrun` are 1-bit output.
- The design is clocked on the rising edge of `clk`.
- Every serial bit transmission has a preamble that needs to be sent before sending the character. For this lab, the preamble value is fixed to 8'hA5.

- After the preamble, the `ready` signal will be high.

Designing a Serial-to-Parallel Interface

1. Change to the *lab18-rcvr* directory and examine the following files.

rcvr.v	Receiver model (incomplete)
rcvr_test.v	Receiver test
genus_shell.tcl	Synthesis script

   a. The receiver searches the serial input stream for a match character, and when found, loads the next serial input character to the output buffer, asserts a “ready” flag, and immediately searches for the next match. Modeling the individual parts of the receiver is like modeling individual components.

2. In your favorite editor, modify the *rcvr.v* file to complete the receiver definition.

Verifying the Serial-to-Parallel Interface

1. Verify the RTL model using the following commands with Xcelium™.

```
xrun rcvr.v rcvr_test.v
```

2. You should see the following.

```
Message sending:  I Love Verilog
At time 35: Put character "I"
At time 39: Got character "I"
At time 77: Put character " "
At time 95: Got character " "
At time 117: Put character "L"
At time 127: Got character "L"
At time 153: Put character "o"
At time 175: Got character "o"
At time 193: Put character "v"
At time 207: Got character "v"
At time 231: Put character "e"
At time 257: Got character "e"
At time 275: Put character " "
At time 297: Got character " "
```

```
At time 309: Put character "V"
At time 313: Got character "V"
At time 349: Put character "e"
At time 365: Got character "e"
At time 391: Put character "r"
At time 423: Got character "r"
At time 425: Put character "i"
At time 433: Got character "i"
At time 459: Put character "l"
At time 487: Got character "l"
At time 495: Put character "o"
At time 507: Got character "o"
At time 537: Put character "g"
At time 563: Got character "g"
Message received: I Love Verilog
TEST DONE
```

Correct your model until it passes the test.

3. When all models pass their test, synthesize the RTL model by entering:

   ```
   genus -f genus_shell.tcl
   ```

   Check to see that the *synthesis succeeded*.

   Correct your model until it can synthesize.

4. Verify the gate-level model using the following commands with Xcelium.

   ```
   xrun rcvr.vg rcvr_test.v -v ../tutorial.v -vlogext vg
   ```

   Check to see the messages *I Love Verilog* and *TEST PASSED* similar to, as shown above.

   If the model fails its test, then you can ask the instructor for help or try again after you study the lecture module *Avoiding Simulation Mismatches*.

Module 19: Using Verification Constructs

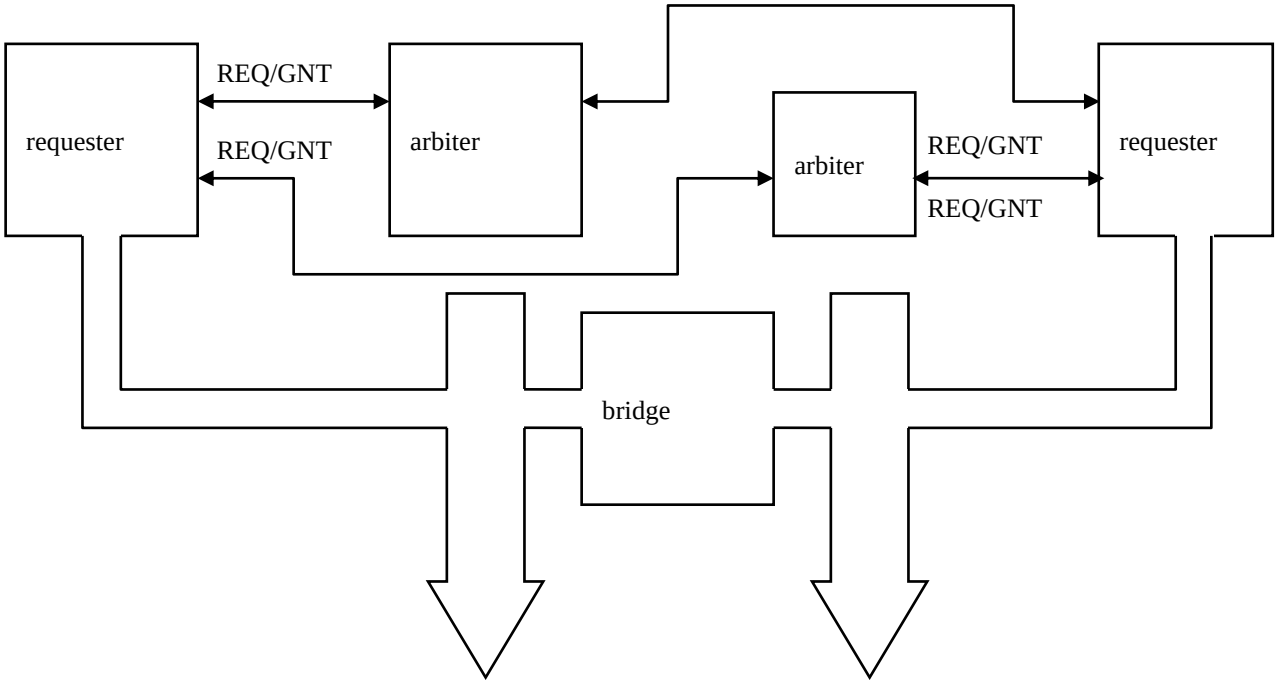
Lab 19-1 Resolving a Deadlocked System

Objective: To resolve a deadlock between two devices competing for two resources.

You will frequently encounter the situation where the test environment waits an undetermined time for the system being tested to respond to some stimulus. The system might malfunction so that it does not ever respond. The test must anticipate this failure mode and report it.

The lab model is an arbiter that arbitrates between two users of a resource. The arbiter prioritizes the requests for the resource. The lab does not model the resource.

The test environment is two instances of the arbiter and two instances of the device making the requests. The device at random intervals requires one or both resources. It requests the first resource, and upon being granted the first resource, sometimes also requests a second resource. Due to the random nature of the requests, the simulation will invariably eventually come to a point at which both devices have one resource and cannot continue until they have the second resource – hence both become “deadlocked.”



Creating a Design to Resolve Deadlock in a System

1. Change to the *lab19-lock* directory and examine the files.

2. Simulate the test case using the following commands with Xcelium™.

   ```
   xrun test.v
   ```

   The simulation should finish with no error indication.

3. Examine the *outfile.txt* file.

   You should expect the last line to indicate a deadlock situation with both requesters requesting both devices and both requesters having one of the devices as in this example:

   ```
   time  r1 r2  g1 g2
      ...
        27  11 11  01 10
   ```

   Note: The *r* column is the requester 1 and requester 2 requests and the *g* column is the requester 1 and requester 2 grants. The test is limited to 99ns, which is probably sufficient to develop a deadlock.

4. Modify the test case requester definition as follows.

   a. Define a watchdog task that after a reasonable amount of time (the solution uses 17ns) drops both request signals and disables the request loop (the request loop will immediately automatically restart).

   b. Each place where the requester waits for a request to be granted, replace the wait statement with a block that does two things in parallel:

      - Enables the watchdog task
      - Waits for the grant, and when the grant occurs, disable the watchdog task

5. Simulate the test case as before and correct any reported errors.

6. Examine the *outfile.txt* file as before.

   You are likely to see multiple deadlock situations occur and be resolved before the test runs out of time at 99ns.

Module 20: Coding Design Behavior Algorithmically

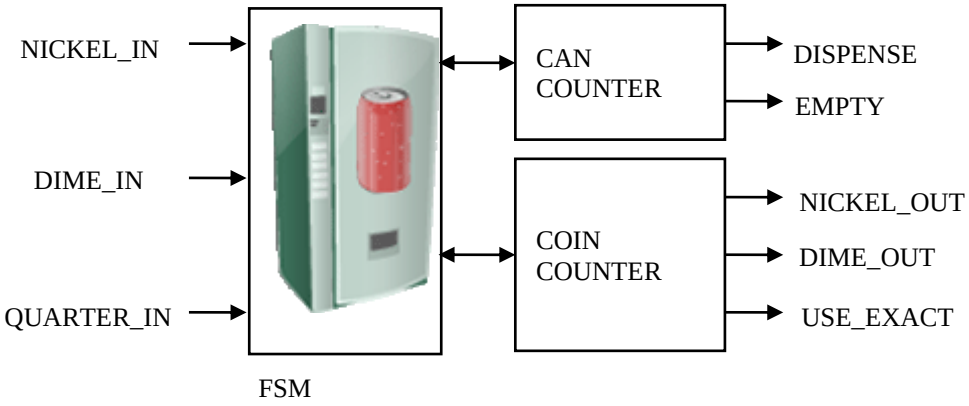
There are no labs for this module.

Module 21: Using System Tasks and System Functions

Lab 21-1 Adding System Tasks and System Functions to a Beverage Dispenser Design

Objective: To add system tasks and system functions to support debugging efforts.

The lab model is a beverage dispenser (drink machine). Read the specification first and then follow the instructions in the lab section. Please make sure to use the same variable name as the ones shown in block diagrams.



Specifications

1. The drink machine operates on the positive clock edge and has a synchronous high-active reset.

2. After resetting the machine, you load it with coins and cans.

3. Then, as you insert `NICKELS`, `DIMES`, and `QUARTERS`, at 50 cents or above, it dispenses a can and attempts to issue your change (if any).

4. This drink machine has no coin return feature and inserted coins are not available later as change.

5. If the `EMPTY` signal is true, then you lose any coins you insert.

6. If the `USE_EXACT` signal is true, then you might find yourself short-changed.

7. As the machine is currently defined, it does not attempt to issue lower-denomination coins if higher-denomination coins are not available.

8. **Drink Machine State Table**: Models the behavior for accepting coins and dispensing drinks:
   - Amount of money that the user has deposited so far defines the current state.
   - The type of coin that the user deposits determines the machine's next state.

Current State	Transition Value	Next State (Change)
idle 4'd0	nickel_in dime_in quarter_in	five ten twenty_five
five 4'd1	nickel_in dime_in quarter_in	ten fifteen thirty
ten 4'd2	nickel_in dime_in quarter_in	fifteen twenty thirty_five
fifteen 4'd3	nickel_in dime_in quarter_in	twenty twenty_five forty
twenty 4'd4	nickel_in dime_in quarter_in	twenty_five thirty forty_five
twenty_five 4'd5	nickel_in dime_in quarter_in	thirty thirty_five idle
thirty 4'd6	nickel_in dime_in quarter_in	thirty_five forty idle (nickel_out)
thirty_five 4'd7	nickel_in dime_in quarter_in	forty forty_five idle (dime_out)
forty 4'd8	nickel_in dime_in quarter_in	forty_five idle idle (nickel_out, dime_out)
forty_five 4'd9	nickel_in dime_in quarter_in	idle idle (nickel_out) idle (two_dime_out)

Designing a Beverage Dispenser FSM Design

1. Change to the *lab21-dkm* directory and examine the files provided.

dkm.v	DUT
test.v	Test(incomplete)

Note: The test developer developed the test in a top-down manner, determining and coding top-level tasks first, then subtasks, and then more subtasks. Only the lowest-level tasks actually interact with the machine. You can also adopt this approach to test development.

2. Design and test are given; you have to modify the test file as instructed.

   a. Improve the *expect()* task so that it displays an error message and displays each machine output that is erroneous.

   b. Display the expected and actual values in *expect()* task.

   c. Display the simulation time and terminate the simulation.

   d. Add a monitor code in the test file with the following functionality.

      i. Monitors drink machine inputs and outputs to a disk file. The monitor should print the simulation time of each signal change and should use the %t formatter so that the printed time value takes a fixed number of columns.

      ii. Dumps all top-level drink machine nets and variables into a VCD file.

      iii. Dumps all drink machine ports to an extended VCD file.

   e. Improve the test procedure so that if it completes, then it displays a happy message and terminates the simulation.

Verifying the Serial-to-Parallel

1. Simulate the design and test using the following commands with Xcelium.

   ```
   xrun dkm.v test.v (Batch Mode)
   ```

   or

   ```
   xrun dkm.v test.v -gui -access +rwc ( GUI Mode)
   ```

2. On successful simulation, you will see the following display.

   ```
   CANS 1 ;  COINS 0,0 ;  INSERT 0,0,2
   CANS 1 ;  COINS 1,2 ;  INSERT 0,3,1
   CANS 1 ;  COINS 1,2 ;  INSERT 0,1,2
   CANS 1 ;  COINS 2,3 ;  INSERT 1,1,2
   CANS 1 ;  COINS 1,4 ;  INSERT 1,1,2
   TEST PASSED
   ```


3. If you have sufficient remaining lab time, add the interactive test by adding the ***+INTERACTIVE=1*** invocation option. The interactive test should add some cans and coins and then, in a loop, repeatedly request the user to insert a coin. Your code can read an input character, discard the remainder of the input line, and, depending upon the character, insert a nickel, dime, or quarter. Your code should check and report the machine outputs after each can is dispensed. You may find the following information useful.

   a. stdin is file descriptor 32'h8000_0000

   b. stdout is file descriptor 32'h8000_0001

   c. reg_8_bit = $fgetc (file_descriptor); // gets the one next 8-bit character

   d. error_integer = $fgets (reg_vector, file_descriptor); // gets remaining line

      (Ensure that reg_vector is sufficiently wide enough to get the entire remaining line.)

      Note: You must increase the clock count to accommodate this additional test. Now that your test procedure gracefully terminates the test, you can alternatively replace the *repeat (n)* loop with a *forever* loop.

4. Simulate the design and test and correct the test as needed.

5. Load the VCD file into a graphical simulation analysis environment by entering:

   ```
   simvision your_dumpfile_name
   ```

6. Verify that the drink machine operates correctly.

Module 22: Generating Test Stimulus

Lab 22-1 Verifying a Serial Interface Receiver

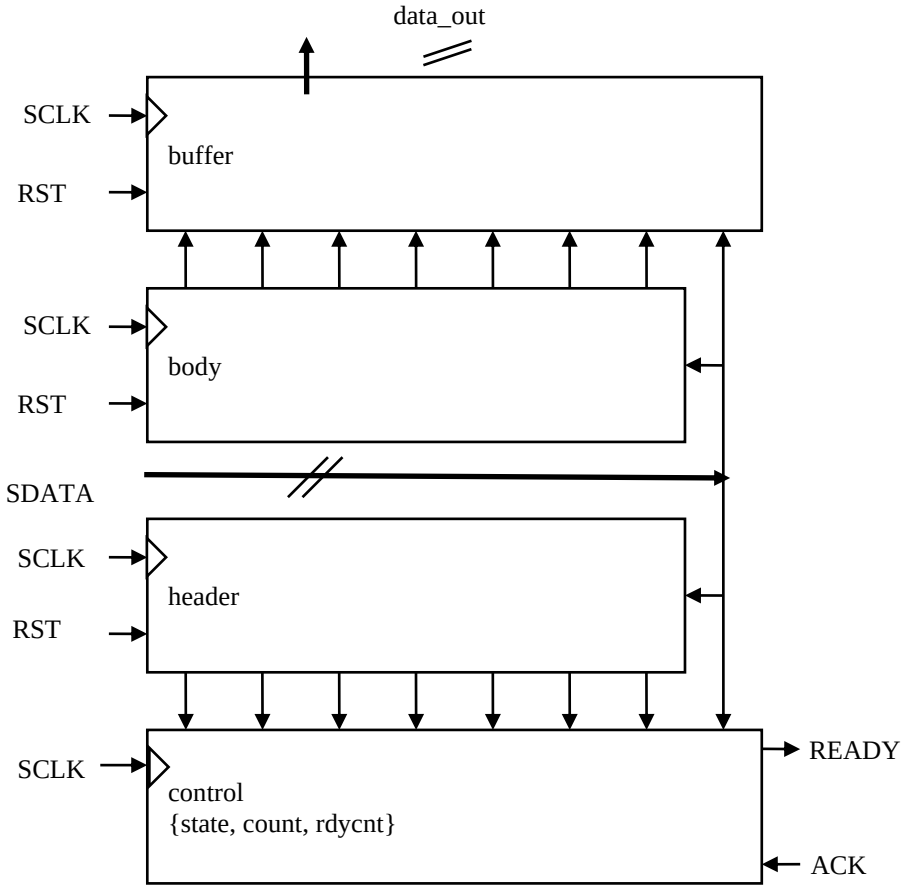
Objective: To test a serial interface receiver.

The lab model is a serial interface receiver. The receiver interfaces between a serial input stream and a parallel output stream.

In this lab, read the specification first and then follow the instructions. Please make sure to use the same variable name as the ones shown in block diagrams.

Specifications

- The receiver operates on the positive edge of the serial clock **SCLK** and has a synchronous active-high reset **RST**.
- A 24-bit packet represents the data.
- The packet consists of an 8-bit header with a value of 0xa5 followed by two 8-bit data bytes.
- The receiver initially shifts input data left into the header register until it detects the header value.
- For this reason, the header register must be initialized to a value opposite the leftmost header value bit.
- Upon detecting the packet header, the receiver clears the header register and shifts input data left into the body register while counting to 16.
- Upon the 16th count, the receiver moves the data to the output buffer, clears the counter, asserts the ready output, and again shifts input data left into the header register. The receiver can thus move a packet every 24 clocks.



Designing the Finite State Machine

1. The test environment reads the leftmost buffer byte while acknowledging the ready signal. The receiver shifts the output buffer left by one byte upon each such acknowledgment. The receiver counts acknowledge and drop the ready signal upon the last acknowledge. The test environment can delay the last acknowledge up to and including the clock that loads the next data into the output buffer. Failure of the test environment to retrieve data within that interval results in lost data.

2. In this lab, you have to generate a simple test of the serial interface receiver, which generates random data for four packets.

3. You statically construct a 256-bit stream containing four valid packets that are surrounded by values other than the header value.

4. Define HEADER_SIZE, BODY_SIZE and HEADER_VALUE as parameters with values 8, 16, and 8`ha5, respectively.

5. You reset the receiver and give this stream as an input to the receiver inputs. While adhering to the output protocol, you retrieve the receiver output data and verify that all packets are received and correct.

Verifying the Finite State Machine Design

1. Change to the *lab22-rcvr* directory and examine the following files.

rcvr.v	DUT

2. Simulate the design and test using the following commands with Xcelium.

```
xrun rcvr.v your_test_file_name -access +rwc
```

You should see the following result if simulation was successful.

```
121ns: Rcvd data 3524
201ns: Rcvd data 5e81
377ns: Rcvd data d609
489ns: Rcvd data 5663
489ns: Response process complete
517ns: Stimulus process complete
517ns: frames_sent=4, frames_rcvd=4
TEST PASSED
```

3. Correct your design as needed.

Module 23: Developing a Testbench

Lab 23-1 Testing the VeriRISC CPU Model

Objective: To interactively select and download test microcode and run it.

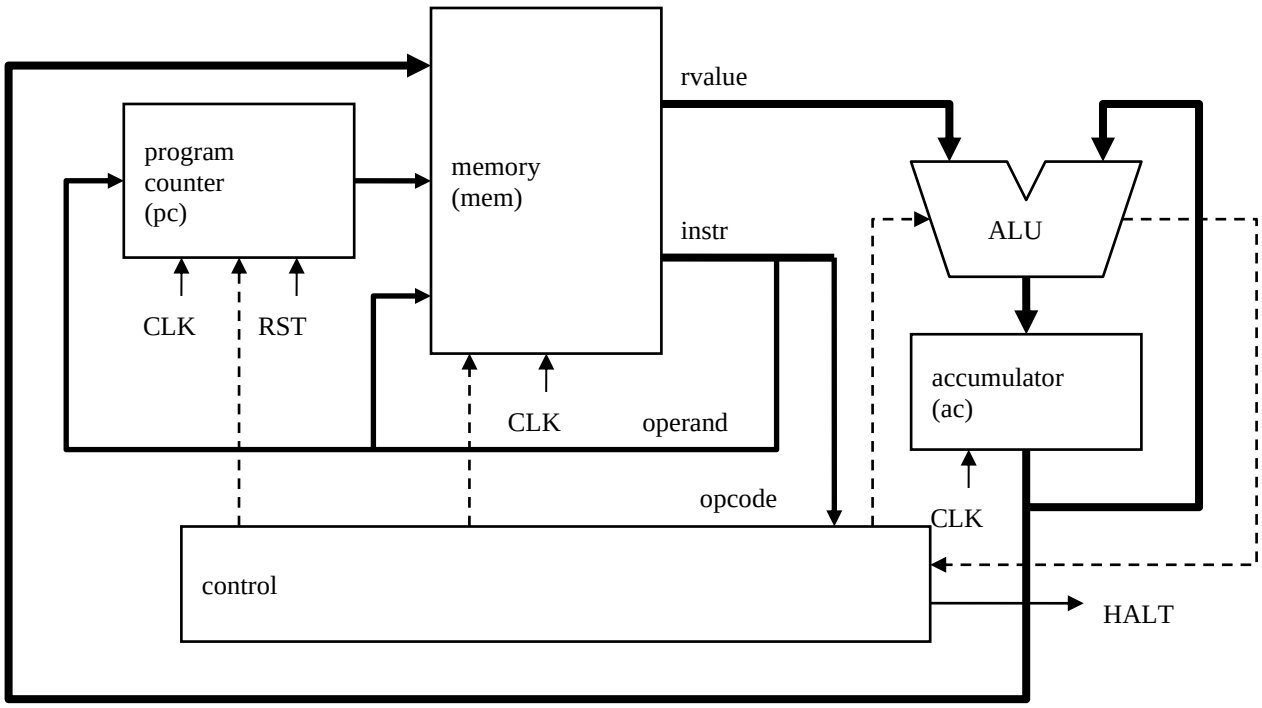
The lab model is a VeriRISC CPU. In the Lab-12 specification, the CPU architecture and operation is explained. In this specification, a few more operations are explained.

Read the specification first and then follow the instructions in the lab.

Specifications

The CPU architecture is as follows:

- The CPU operates on the positive clock edge.
- It has a synchronous high-active reset.
- The CPU uses a two-port memory so that it can fetch and execute an instruction on each clock cycle.
- The 8-bit CPU instruction consists of a 3-bit leftmost operation code encoding eight instructions.
- And a 5-bit rightmost operand addressing up to 32 words.



Designing the Testbench for VeriRISC CPU Verification

1. In the previous block diagram, you can observe the CPU has only HALT as output.

2. The testbench environment can detect program failure only by detecting that the CPU halted at an incorrect address.

3. This lab provides three programs with incremental levels of complexity that halt (when operating correctly) at different program counter values:

   For program 1 halt value is – 0x17

   For program 2 halt value is – 0x10

   For program 3 halt value is – 0x0c

4. You have to code the specified testbench, which can have well-defined tasks for the smoother execution of the programs.

   a. First, code a task displaying an initial message informing the user what commands to use to operate the test. You can write the task by using the following interactive command in which you have to place a value in a test register.

deposit *object_name value*	Deposit a value onto a simulation object
task *task_name*	Schedule a Verilog task for immediate execution
run *time_spec*	Continue the simulation to the next breakpoint, interrupt, or (optional) time point

   Note: For example: Deposit a value to a register/variable inside a task like{deposit *modiule_name.task_name.variable_name value*} **deposit test.run.number 1** where “number” is the register/variable.

5. Code a task to run a program in which the task creates a filename (which is a string) by concatenating the value of a register/variable that the user sets by using the *deposit* command. In the lab directory, you can see the txt filename convention that you use to name the programs such as *PROGn.txt*.

   Note: The ASCII value of the *0* character is 0x30 so your concatenation would be something like {"PROG", 8 'h30 + *test_number_reg*, ".txt "}.

6. Load that program file into the CPU memory. Reset the CPU. To avoid a clock/data race, you can move the reset signal on the opposite clock edge in which the CPU works (use **negedge**). After reset, the CPU automatically starts executing at location 0.

7. Code a task displaying a final message informing the user of the program counter address where the CPU halted and whether that is correct.

8. Code a procedural block generating a free-running clock.

9. Code a procedural block that upon startup:

   a. Displays the initial message.

   b. Pauses the simulation (use **$stop**).

10. Code a procedural block that upon every assertion of the HALT signal:

    a. Displays the final message.

    b. Again, displays the initial message.

    c. Again, pauses the simulation.

Verifying VeriRISC CPU Design

1. Change to the *lab23-cpu* directory and examine the following files.

cpu.v	CPU module
PROG1.txt PROG2.txt PROG3.txt	Program files

2. Code the specified test with well-defined tasks for the smoother execution of the programs.

3. Simulate the design and test using the following commands with Xcelium.

```
xrun cpu.v your_test_file_name -access +rwc
```

4. Execute the following commands and compare the outputs after each run.

```
xcelium> deposit test.run.number 1; task test.run; run
Halted at address = 17
Expected  address = 17
TEST PASSED
```



```
xcelium> deposit test.run.number 2; task test.run; run
Halted at address = 10
Expected  address = 10
TEST PASSED
```

5. Correct your test and repeat as needed.

Module 24: Example Verilog Testbench

Lab 24-1 Developing a Script-Driven Testbench Using Verilog 1995

Objective: To develop a script-driven testbench using Verilog 1995.

It is possible to define a set of instructions by which the operation of a testbench can be controlled. This collection of user-defined instructions is often called “Pseudo-code” and a testbench usually reads these instructions from a text file. This type of testbench is also called a script-driven testbench. The testbench is written to interpret the text of each instruction and then perform that operation. You will see an example in this lab.

One issue that we have to remember is that the user-defined instructions are more likely to be written by hand, and therefore contain errors than a file of stimulus data created by, for example, a graphics package. Therefore, our testbench needs to carefully check the instruction data it is trying to read. In this lab, you write a Verilog 1995 *vreadmem_test.v* that uses the system task $*readmemh* for getting the required commands from a *data.txt* into the procedure call within the testbench.

Creating a Script-Driven Test Bench

1. Change to the *script-driven-tests-95* directory and examine the following files.

data	Commands required
vreadmem_test95.v	Module (incomplete)

2. In the incomplete module, comments are given to write a procedure using the system task **$readmemh** to get the commands from *data.txt* into the cmdarray “cmdarr” defined.

3. Create a loop (using the **if** statement) wherein 8 commands are read, corresponding to each bit of the cmdarray, and then when encountered, the 16'bx or Fh displays “end of commands” and then stops reading using **$stop**.

4. In the next part of the loop, check for the SEND, ADDR, NEXT commands (using the **case** statement) and create respective tasks. Have the default display as “unknown command” for any other command encountered.

5. Simulate the design and test using the following commands with Xcelium.

```
xrun vreadmem_test95.v
```

You should see the following result on successful simulation.

```
do_send with addr = 11, data = a
do_addr with addr = 00
do_send with addr = 10, data = f
```



```
do_next with addr = 01
do_addr with addr = 11
end of commands
```

6. Correct your test and repeat as needed.

Lab 24-2 Developing a Script-Driven Testbench Using Verilog 2001

Objective: To develop a script-driven testbench using Verilog 2001.

For this testbench, write a Verilog 2001 *vfopen_test.v* that uses the system task $*fopen* to get the file containing the *cmds* to be opened by the procedure call within the testbench.

The declarations of necessary variables and a set of tasks have been created in the testbench file. Follow the instructions in the comments in the *vfopen_test2001.v* and *vreadmem_test95.v* tests and create the procedures using *readmemh* and *fopen* to get the *data.txt* and *cmd.txt* into the procedures within the testbenches, respectively.

Creating a Script-Driven Test Bench

1. Change to the *script-driven-tests-2001* directory and examine the following files.

cmd	Commands required
vfopen_test2001	Incomplete module

2. Given the test file is incomplete, follow the comments inside to write a procedure using the system task **$fopen** to get the commands from given *cmd.txt*.

   Note: Use variable fid = fopen the cmd.txt.

3. Use the **while** condition with **$feof** to check that it is not the end-of-file of *fid*, then scanf the cmd and addr using **$fscanf**.

4. In the second part of the loop, for cmd SEND, ADDR, NEXT, use the above tasks created. If any other cmd is encountered, have a default display "unknown command".

5. Simulate the design and test using the following commands with Xcelium.

   ```
   xrun vfopen_test2001
   ```

   You should see the following result, if simulation was successful.

   ```
   do_send with addr = 01, data = a
   do_addr with addr = 00
   do_send with addr = 00, data = f
   do_next with addr = 01
   do_addr with addr = 01
   ```

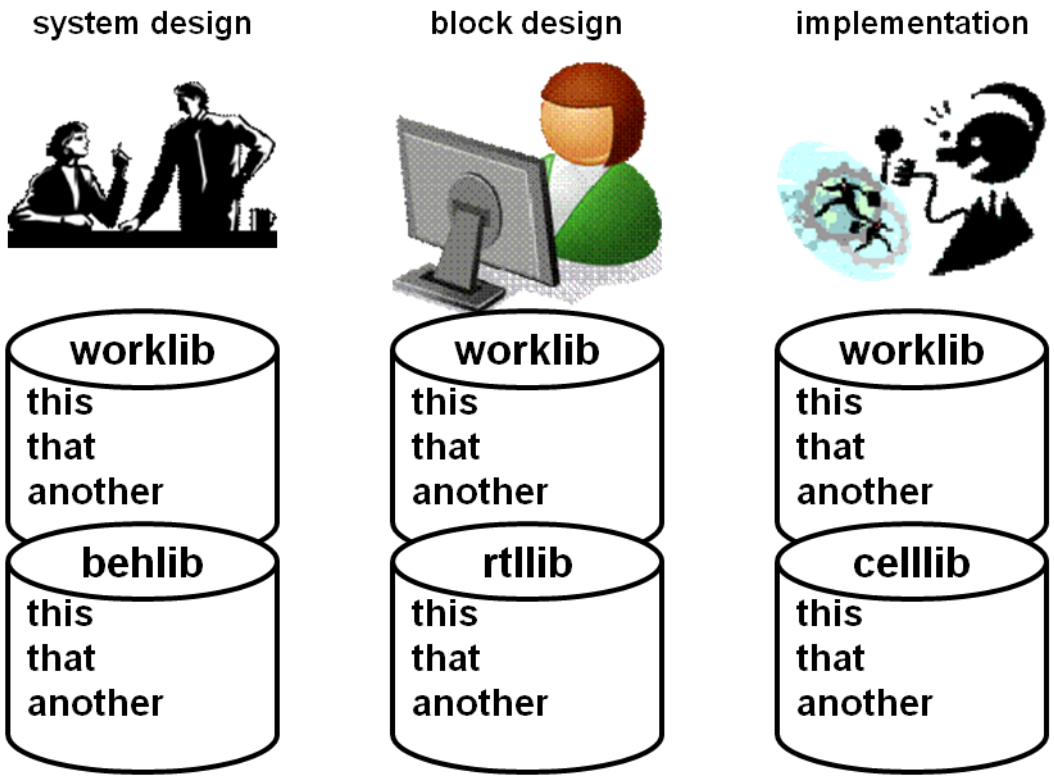
6. Correct your test and repeat as needed.

Appendix A: Configurations

Lab A-1 Configuring a Simulation

Objective: To use a library map file to define a configuration that includes at least one gate-level component.

Verilog configurations provide a standard portable way to select among multiple implementations of components stored in multiple libraries to assemble a simulation. The multiple implementations can be, for example, different levels of abstraction or different internal functionality, or different speeds. This lab provides a library of components duplicating all components you have previously developed for this training. The library components are coded using accelerated Verilog primitives. Your configuration can specify to use any of these components except the memory, as the test does not know how to download a program into a gate-level memory.



Creating VeriRISC CPU Design

1. Change to the *lab-appendixA-conf* directory and examine the following files.

cell_lib.v	Library of cells
proj_lib.v	Library of components
risc.v	RISC model
risc_test.v	RISC test (incomplete)
files.txt	List of RISC sources
CPUtest1.txt CPUtest2.txt CPUtest3.txt	Test programs
libmap.txt	Library mapping file

2. Create the *libmap.txt* file and use your favorite editor to define the library mapping and configuration. This file is partially provided below to help you get started.

```
library celllib cell_lib.v;
   library projlib proj_lib.v;
   library worklib "../.../*";
   config risc_test;
     design worklib.risc_test;
     default liblist worklib projlib celllib;
   // TO DO: FOR SOME SET OF RISC CELLS OR INSTANCES USE THE PROJECT
   LIBRARY.
   //        FOR THE MEMORY YOU MUST STILL USE THE DEFAULT WORK LIBRARY.
   endconfig
```

3. Test your configuration using the following commands with Xcelium.

```
xrun cell_lib.v proj_lib.v -f files.txt -libmap libmap.txt \
   -libverbose -top risc_test:config
```

   The *-libverbose* option logs verbose elaboration information. The *-top* option causes the *risc_test configuration* to elaborate instead of the *risc_test* unit.

4. You should see the following.

```
Testing reset
Testing HLT instruction
Testing JMP instruction
Testing SKZ instruction
```

```
Testing LDA instruction
Testing STO instruction
Testing AND instruction
Testing XOR instruction
Testing ADD instruction
Doing test CPUtest1.txt
Doing test CPUtest2.txt
Doing test CPUtest3.txt
TEST PASSED
```

5. Examine the log file to verify that the elaborator bound the specified gate-level units.

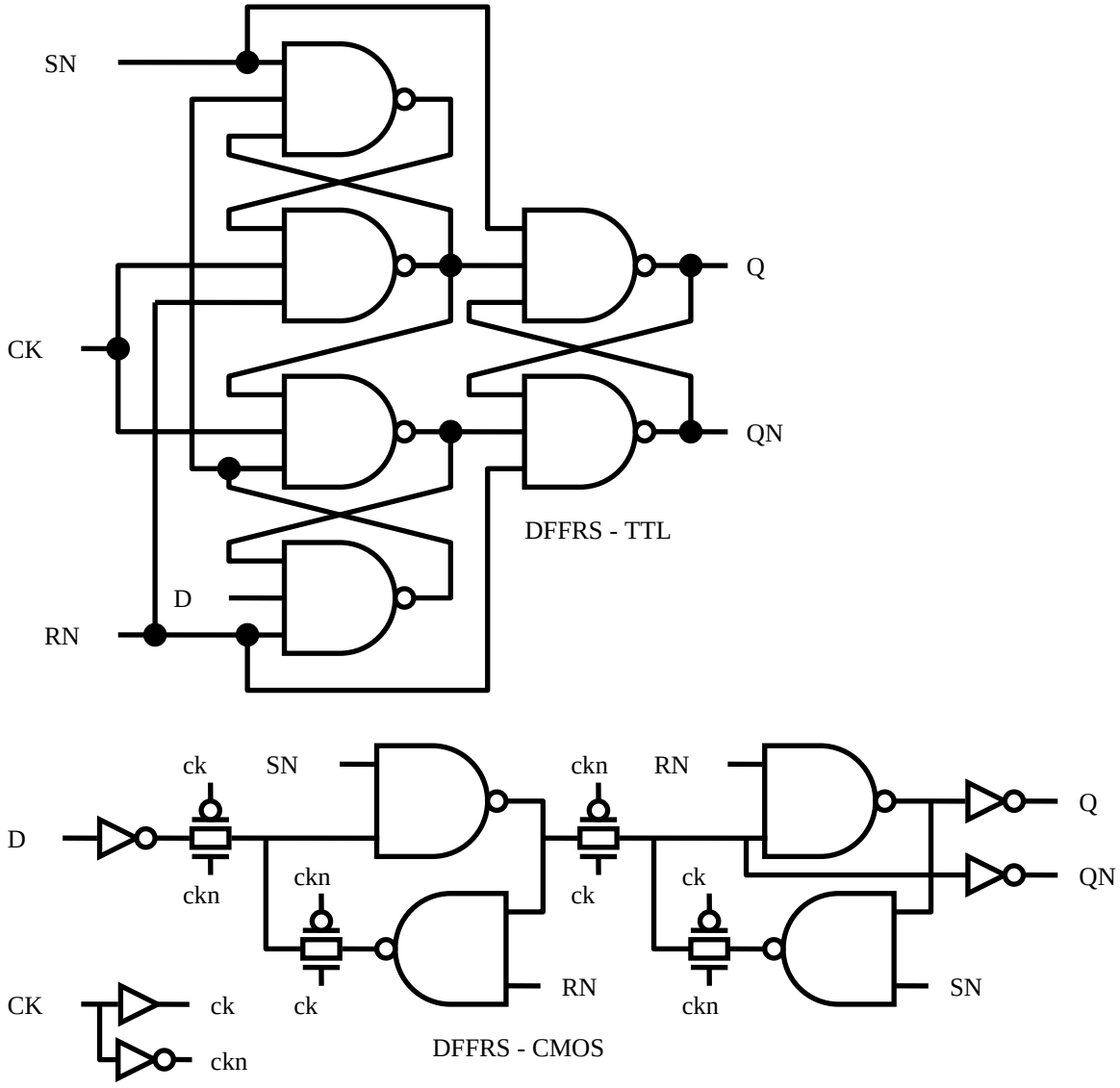
6. Correct your library map file as needed.

Appendix B: Modeling with Verilog Primitives and UDPs

Lab B-1 Using Built-In Verilog Primitives with a Macro Library

Objective: To use built-in primitives to model logic.

For this lab, you modify a provided macro library file to replace the RTL descriptions with descriptions based upon the Verilog built-in primitives.



Creating Verilog Primitives with a Macro Library

1. Change to the *lab-appendixB-primitives* directory and examine the following files.

README.txt	Lab instructions
techlib.v	Macro library
testdir/	Test cases

2. Verify the RTL technology library for each test using the following commands with Xcelium.

   ```
   xrun testdir/test_name.v -v techlib.v
   ```

   The *-v* option provides a module library file.

   You can create a small script to make this task less tedious.

   Each simulation finishes and indicates *TEST PASSED*.

3. Modify the macro library file to replace the RTL descriptions with descriptions based on the Verilog built-in primitives. Refer as needed to the flip-flop diagrams on the first page of this lab. The first diagram is of a positive edge-triggered TTL flip-flop. Replace the **nand** primitives with **nor** primitives to create a negative edge-triggered TTL flip-flop. The second diagram is of a positive pulse-triggered CMOS master-slave flip-flop. Do not overly concern yourself with whether it is the preset or the reset that “wins” because no synthesis tool will activate them both simultaneously.

4. Verify your modifications and correct them as needed.

Lab B-2 Using User-Defined Verilog Primitives with a Macro Library

Objective: To define a sequential primitive.

A user-defined Verilog primitive (UDP) is essentially a look-up table (LUT). The LUT requires substantial memory, so you generally define a UDP only to replace several built-in primitives, especially if the module is instantiated multiple times.

For this lab, you modify the macro library file further to define a sequential flip-flop UDP and to replace the primitive-based flip-flops with UDP-based flip-flops.

Creating UDP with a Macro Library

1. Change to the *lab-appendixB-udp* directory and examine the following files.

techlib.v	Macro library (incomplete file)
testdir/	Test cases

2. Modify your technology library file as follows.

   a. Define a sequential UDP to represent a flip-flop having a positive edge-triggered clock and asynchronous preset and reset. Do not overly concern yourself with whether it is the preset or the reset that “wins” when simultaneously activated.

   b. Modify the flip-flop descriptions to instantiate the UDP instead of the built-in primitives. Invert the clock where needed. Tie off unused preset and/or reset inputs. Buffer the Q and QN outputs.

3. Verify the RTL technology library for each test using the following commands with Xcelium.

```
xrun testdir/test_name.v -v techlib.v
```

4. Verify your modifications and correct them as needed.

Appendix C: SDF Annotation Review

Lab C-1 Annotating an SDF with Timing

Objective: To annotate timing information.

In this lab, you annotate timing data to a design. The design is a serial interface receiver. It receives characters serially and transmits them in parallel.

Annotating SDF with Timing Information

1. Change to the *lab-appendixC-sdf* directory and examine the following files.

rcvr.v	DUT (RTL)
rcvr_test.v	Test
rcvr.vg	DUT (gates)
rcvr.sdf	Timing
techlib.v	Component Library

2. Simulate the RTL design and test using the following commands with Xcelium.

   ```
   xrun rcvr.v rcvr_test.v
   ```

   The test displays the message *I Love Verilog*.

   Check to see the message *TEST DONE*.

3. Copy your modified technology library from the previous lab. If you have not completed the lab, then copy the technology library from its solutions directory.

4. Simulate the gate-level design and test using the following commands with Xcelium.

   ```
   xrun rcvr.vg rcvr_test.v -timescale 1ns/10ps -v techlib.v -vlogext vg
   ```

   The *-timescale* option provides a default timescale for the gate-level description.

   The *-vlogext* option registers an additional Verilog file extension.

   Check that you have multiple hold violations and that you see *TEST TIMEOUT*.

5. Use the graphical simulation analysis environment (SimVision™) to verify the *body_reg_reg[0]* propagation delay.

   a. Use the following command with Xcelium.

   ```
   xrun rcvr.vg rcvr_test.v -access r -gui -maxdelays \
      -timescale 1ns/10ps -v techlib.v -vlogext vg
   ```


The *-maxdelays* option selects to use the maximum delays.

b. Probe *body_reg_reg[0]* to a waveform display and run the simulation.

c. Observe and record the Q output rise and fall delays.

d. Verify that these are the maximum delays the library specifies for this type.

6. In an initial block at time 0, modify the test to annotate the timing in the *rcvr.sdf* file to the receiver instance.

7. Use the graphical simulation analysis environment (as before) to verify the annotated *body_reg_reg[0]* propagation delay.

8. Verify that these are the maximum delays that the *rcvr.sdf* file specifies for this instance.