Verilog Session Assignment [2 hours Every Thursday]

All assignments are behavioural unless specified otherwise.

Session-1: Structural ADDER.

Make half-adder, full adder using gate descriptions. Show waveform output and RTL design generated by Xilinx. Write test-bench for each and show output on waveform. Also try "$monitor" command to generate values of output from the testbench. Complete 4-bit ripple carry and show at the beginning of next lab.

Session-2: Ripple-carry – delay analysis

Implement a 4-bit ripple carry adder with gate delays:
output sum and cout are generated after 2ns each.
Thus answer for first bit comes after 2ns.
Answer for second bit comes after 4ns and so on.

A 4-bit ripple carry must give output after 8ns.
Try various testcases and use monitor stmt to check the timing of final output generation. Find the testcase which gives the worst case delay.

Ripple carry adders are slow due to the ripple effect. Therefore, Carry Look Ahead adders are better. Write the code for CLA adder for 4-bit and check the timings. Use equations for CLA logic that give out Pi and Gi for propagate and generate inputs. Each level of logic has 1ns delay. For example, if you have out= (a&b) | (c&d), then you have 2-level logic that uses AND gates followed by OR. So the delay for this will be 2ns.

Comment on which type of adder is good and when and why. Demonstrate your answer.

Session-3:

MUX

=====

Design a 4x1 MUx using following type of stmts:

Output = equations for AND-gates that are inside the mux. Inputs are i0,i1,i2,i3 and select=s1,s0.

Also write output = conditional assignment

Cond ? : stmt-1 : stmt-2

Synthesise both types are check the difference in RTL schematic. Also read the design summary generated and see if the final designs are different or not.

ALU

====

Design 4-bit ALU:

The 4-bit ALU has the following inputs:

A: 4-bit, B: 4-bit , Cin: 1-bit, Control: 3-bit control input

Output: ANS: 4-bit, Cout: 1-bit

Control Instruction Operation

000 ADD Output <= A + B + Cin; Cout contains the carry

001 SUB Output <= A – B - Cin; Cout contains the borrow

010 OR Output <= A or B

011 AND Output <= A and B

100 SHL Output <= A[2:0] & ‘0’

101 SHR Output <= ‘0’ & A[3:1]

110 ROL (Rotate left) Output <= A[2:0] & A[3]

111 ROR (Rotate right) Output <= A[0] & A [3:1]

Use 'case' statement. Design is purely combinational. Output should change as soon as any input or control combination changes.

Subtraction function must give the correct answer and indicate overflows.

Make sure there are no latches in the synthesised design (RTL schematic).

Try removing one case from the ‘case’ stmt and see the RTL. Compare the RTL with all case options. Identify the difference.

How many MUX are there in your RTL? 1 or 2? What is their purpose?

Session-4:

(1)

Design a 3-bit register using a serial input “a”. Input goes to first flip-flop (d0). Output of do which is q0 goes as input to second flip-flop d1-q1. Output of second flip-flop goes to third one d3-q3.

take inputs as d0,d1,d2 and outputs as q0,q1,q2. Don't take multibit registers.

Design using an always @clk statement.

(i) Change the order of your assignments within the always blocks.

(ii) Also implement the block statements using “<=” as assignment operator instead of “=”.

(ii) For both type of assignment operators and all possible reordering of stmts in always block check the RTL generated. Is it same or different?

(iv) Also check the “synthesis report” in the “design summary” and find out delay of our circuit under “HDL synthesis”.

(2)

Take multi-bit inputs and outputs d0-d3 and q0-q3. Design 4-bit register. that has LOAD and CLEAR as control inputs. CLEAR input has priority over LOAD.

Use parallel load feature.

Write appropriate testbench and show that CLEAR has priority.

Testbench mush generate CLK as a continuous waveform.

Session-5:

First make a 4-bit register with LOAD, CLEAR features.

Test it by loading and clearing at random time intervals.

Testcase will have CLK generated using for-loop.

Use the above register and change it to a up-counter. There is additional input INCR. If counter is enabled then it increments.

Write elaborate testcases to demonstrate all features.

Cascade the above counter with another one to generate a 8-bit up-counter. You have to use the 4-bit counter as a module for this design.

Make necessary changes in the 4-bit version if required.

Write proper testcases.

[BONUS]

BCD up-down counter

counter parameters:

Input = Enable, Load, Up, CLR (= async clear), Data-in

Carry-out, Data-out

counter works only when Enable=1.

LOAD=1, Data-in gets loaded in the counter

CLR=1, Counter is reset to 0, asynchronously

LOAD=0 & UP=1 : increment. when counter reaches 9, it makes carry-out=1 and value=0

LOAD=0 & UP=0 : decrement. when counter reaches 0, it makes carry-out=0

use clock period = 10 ns. Use for-loop in testbench to generate clock signal. Try different delay gaps between clk and load/clr and verify that the count responds only to the clock even when the load/clear are given earlier.

!!! MID-SEM !!!

Session-6: Sequence Detector – Parity Checker [Deadline – 13 April]

Draw the FSM for parity checker. Draw Moore as well as Mealy FSMs. Write Verilog code for the FSMs.

What is the difference between Moore and Mealy FSMs ?

Moore Code will have 3 always blocks:

Always block @clk to change state.

Always block @ input and state to compute next-state

Always block @ state to compute output.

Mealy code will have 2 always blocks:

Always block @clk to change state.

Always block @ input and state to compute next-state and output.

Synthesise the circuits. Show waveform outputs. Use forever-loop to generate clock with period=10ns. Change input at various time intervals using delay stmt in test-file.

Identify the difference in output of Moore vs Mealy.

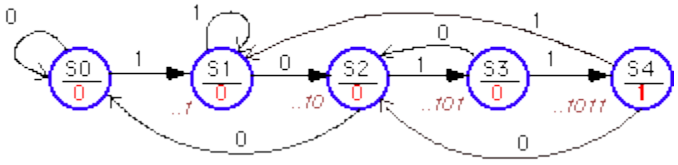
Note the time when the output is generated by both machines.

(insert a minor delay for Mealy machine output to observe changes in state and output timings.)

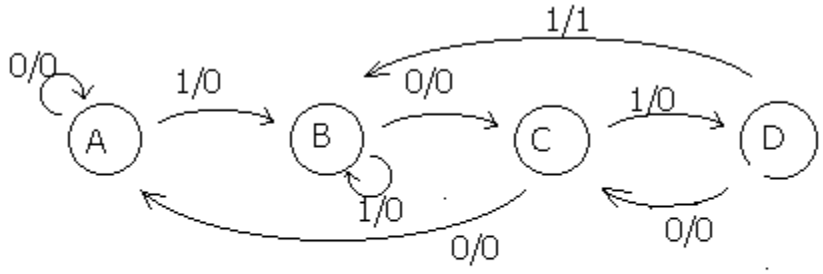
Session-7: Sequence Detector – for sequence 1011 [Deadline – 13 April]

Similar to the above problem, write a state machine based Verilog code to detect the sequence of 1011. Input comes as single bits one by one and the output is also one bit. The output becomes 1 when the sequence 1011 is seen at the input. In all other cases output remains 0. Design the machine to detect overlapping sequences. Design both Moore and Mealy machines.

Following is FSM for Moore machine:



Following is FSM for Mealy machine:



State diagram of 1011 sequence recognizer

Session-8: Booth Multiplier [Deadline – 20 April]

You have studied Booth Multiplication algorithm in the digital design theory course. Implement the same algorithm in Verilog on 4-bit input data.

Session-9: Sequential Multiplier using ASM charts [Deadline – 27 April]

In the chapter on ASM – Algorithmic State Machines, you must have studied the sequential multiplier. Implement the same in Verilog. Write separate modules for controller design, and datapath components.

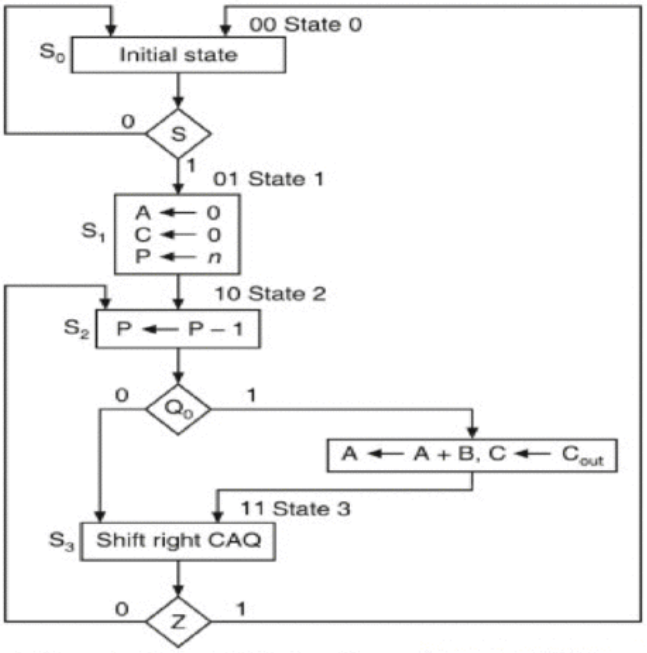


Figure 15.21 ASM chart for a binary multiplier.

Session-10: Single Port RAM module + Register-File [Deadline – 4 May]

RAM-module

inputs = clk, data_in, wr_en,rd_en, address

outputs = data_out

Make an array of registers to store the data which will represent the memory = RAM.

Make each location 2-byte wide and RAM will have total 16 locations.

You can use "initial" block inside the module to pre-load the RAM with data values.

Subsequently the feature of the RAM has to implemented.

At the clock when write enabled: write data in to the given address [clk related always block]

The RAM can be read anytime using read-enable, without the clock signal. [This will go in another always block]

-- you can try "assign" but that did not work for me !

Write an appropriate testbench.

testbench will try to write to few locations in the RAM and later read the same locations to cross check if the data got written properly.

Identify the clock cycles required to read the data and to complete a write.

For write, you have to send address and wr-enable. See if data gets written same cycle or next.

To read, you have to send the address, the data will come on the output. Find out it the data comes in same clock cycle or next.

Register File

inputs = clk, reset, read_select, write-select, write-enable, data-in

output = data-out

[read and write select are the index of the register ID.]

A register file is a collection of registers used inside the processor.

Make an array of registers. Each register 8-bit wide and make 8 such registers.

At each clock if write-enabled, the input data is written to the appropriate register.

The register-id to which data is to be written comes as input.

[use one always block for this]

You can use a read-enable to read or just read anytime (even when clock is absent).

[use another always block for this]

when reset make all registers contents = 0;

Write appropriate testbench.

Test bench should try to reset all registers and then write and read from random reg-id.

Try to mix read and write tests.

Check if read happens in same clock or next.

Check if write is effective in same clock or next.

Session-11: Fetch + Execute instruction [Deadline – 11 May]

We are not making a “real” processor. But will try to get a feel of the components.

Write a top-level module “mycontroller” and include the RAM and Register file from the above assignment in this module.

In the RAM module using initial block and add some content to first 16 locations of the RAM.

“mycontroller.v”

inputs = clk, reset

output = data_out // used as test output to verify if program is working

output = curState // current state is output to verify if program is in correct state

Build the code slowly. Follow steps given below. After each "Step" write extra states to verify what you have done is working or not.

Step-1

Read addresses 0 to 5 of RAM and store the data-read into Reg-file Registers 0 to 5.

Do this by building a state machine. read one location in one or more states. and send the memory data to the corresponding register. You have to assign the memory data to appropriate internal variables. These internal variables will be arguments to the RAM and Reg-File module. In particular, the output of RAM will come in a “wire”-type internal variable and you copy this into another “reg”-type internal variable. This reg-type internal variable is data-input to the Reg-file instance.

To test if this is working: add one more state after this and try to read any of these registers and output the content on data_out to verify.

Step-2

You will require ALU to execute the instructions. Use the ALU you have designed in the earlier assignment.

ALU :

inputs = a (8-bits) , b (8-bits), func(3-bits)

output = out (8-bits) // ignore any overflows and carry-outs

func = a+b, a-b, a&b, a^b, ~a, a<<b, a>>b

include ALU module in the "mycontroller" file.

Step-3

Assume that RAM locations 6 onwards has some sample instructions to execute.

You can design your own instruction formats once you are confident about the procedure. I am giving a sample format which you can start with.

Reg-Reg type instruction: ex. `ADD R0, R1, R6` ==> R6 = R1+R2

Reg-Imm type instruction: ex. `ADD R0, R6, #4` ==> R6 = R1+4

ALU function needs 3 bits and type of instruction needs 1-bit to identify one of the above types of instruction.

Therefore, opcode for our two types of instructions needs 4-bits.

source and destination register identification will need 3-bits each (as we have 8-register in the Reg-File) [if you have more registers in Reg-file, this field will be larger]

For convenience the immediate-data field is also kept to 3-bits. [i.e. any instruction can add immediate values ranging from 0-15]

For reg-reg type the instruction will have following format:

13-bits required. Use MSB side 3-bits as don't care (because RAM will give 16 bits output as instruction)

```
xxx          oooo      sss      sss       ddd
don't care   opcode    src-1    src-2     dest
```

Similarly the Reg-Imm type instruction is as follows:

```
xxx          oooo      sss      ddd       ccc
don't care   opcode    src-1    dest      const
```

This way your ADD R0, R1, R6 instruction = `xxx 0000 000 001 110`

You can pre-load RAM with some sample instructions like the above. Load such instruction starting from address 6 onwards.

Step-4

In the "mycontroller" file from the state machine position after your register filling state (end of Step-1 above) do the following.

In each new state read the RAM. This time you get an instruction-out from the RAM.

Using the above format, identify the src-1 and src-2 registers.

Read the Reg-File for src-1 and send the content of the register to the ALU as the first operand.

Next read src-2 and send the register value to ALU as second operand.

Assign the 'func' input of ALU using the opcode field of the instruction.

After doing this, the ALU will compute the output and in the next state you can read the output.

Read the alu-out and assign it to data_out register. Observe this output and verify if it has done the correct ALU operation.

Step-X

The test-bench for "mycontroller" is very simple. Because we have done all the scheduling inside the verilog file itself.

In the test-bench write the code for clock generation. and just reset the components. Then you have to do nothing in the test-bench.

```
always #5 clk = ~clk;


initial

begin

#100;  clk=1; reset=1;

 #65; reset=0;

 end
```

The real processor will have separate data-path and control path. The “mycontroller” will become the controller which will send only signals to all data-path modules. You will also have a program counter “PC” which will point to the location of RAM to read the instruction. PC will increment after each instruction fetch. You will have a “decode” module which will take the instruction and send parameters to the “ALU”, etc. More types of instruction can be implemented.

If you are interested you can try the full-fledged processor design.

END-SEM