# ECE-111: Advanced Digital Design Project: Homework 5

| | |
|---|---|
| **Designs** | Carry Look Ahead Adder, Booth Multiplier |
| **Deadline** | November 6, 2024 at 11:59pm |
| **Max. late days** | 2 (20% grade reduction per day) |

## Overview

There will be two parts for this homework. In **homework-5a** you will design a synthesizable SystemVerilog model of a Carry Look Ahead Adder. In **homework-5b**, you will develop a synthesizable SystemVerilog code for a Booth Multiplier.

We have provided a folder called **Lab5.zip** which contains the following:

**Homework-5a:**

1. carry_lookahead_adder.sv partial design template code.
2. carry_lookahead_adder_testbench.sv full code.
3. full_adder.sv design code.

**Note:**

1. For learning purposes, students can change the stimulus in the initial block in the testbench file.
2. Full adder is required when developing the carry lookahead adder design module. Add fulladder.sv file in Quartus and Modelsim for compilation.

**Homework-5b:**

1. booth_multiplier.sv partial design template code.
2. booth_multiplier_testbench.sv full code.

## Assignment Tasks

This assignment requires you to complete the following tasks:

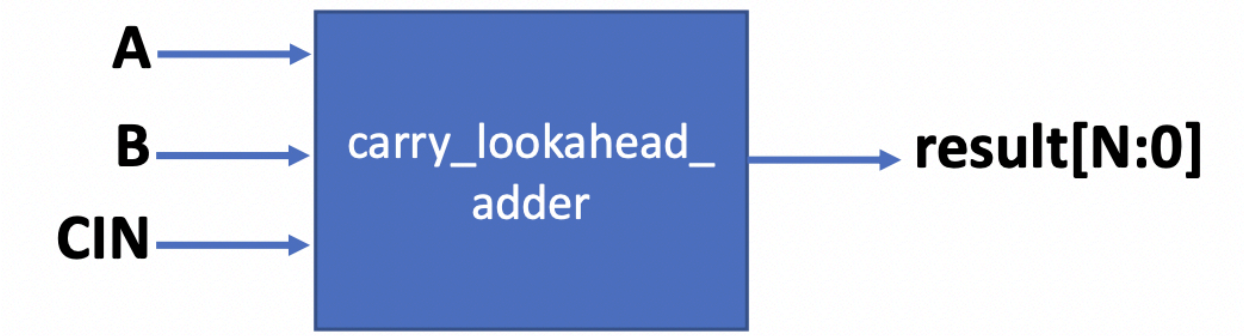
**Recommended task:**

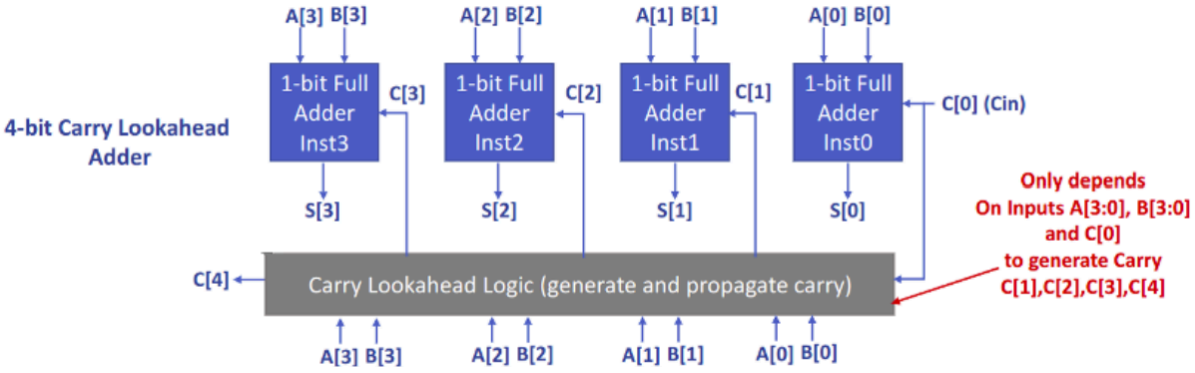
- Watch the discussion video that goes over homework 5

**For Homework-5a:**

- **Develop SystemVerilog RTL model for N-bit Carry Lookahead Adder**
  - Synthesize carry look ahead adder design and run simulation using testbench provided with N=4.
  - Review synthesis results (resource usage and RTL netlist/schematic).
  - Review input and output signals in simulation waveform and transcript.
  - Assume below mentioned primary port names and SystemVerilog RTL module name carry_lookahead_adder.

- **Primary Ports for carry_lookahead_adder module**
  - Input [N-1:0] A, B : These are values to be added using carry lookahead adder logic).
  - Input CIN : carryin to first stage Full Adder Instance in carry lookahead adder.
  - Output result : final output addition result and final stage carryout).
    - i.e. result of A[N-1:0] + B[N-1:0] + CIN)
    - **Note:** Output signal result MSB bit includes final stage full adder carryout.



| inputs | | | outputs | |
|---|---|---|---|---|
| A | B | Cin | Cout | Sum |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

| inputs | | | outputs | |
|---|---|---|---|---|
| A | B | Cin | Cout | Sum |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

- Carry Propagate - When either of inputs is 1 and Cin is 1
  - P[i] = (A[i] | B[i])
- Carry Generate - When both inputs are 1 irrespective of Cin value
  - G[i] = A[i] & B[i]
- Carry Out from Full Adder = Carry Propagate | Carry Generate
  - Cin[i+1] = Cout[i]

= (P[i] & Cin[i]) | G[i]

= ((A[i] | B[i]) & Cin[i]) | (A[i] & B[i])

❑ **G[i] and P[i] can be further replaced with A[i].B[i] and A[i]+B[i] below in each of the C[1], C[2], C[3], C[4] carry equations**

C[1] = G[0] + P[0].Cin

= (A[0].B[0]) + (A[0]+B[0]).Cin

C[2] = G[1] + P[1].G[0] + P[1].P[0].Cin

= (A[1].B[1]) + (A[1]+B[1]). (A[0].B[0]) + (A[1]+B[1]). (A[0]+B[0]).Cin

C[3] = G[2] + P[2].G[1] + P[2].P[1].G[0] + P[2].P[1].P[0].Cin

= (A[2].B[2]) + ((A[2]+B[2]).(A[1].B[1])) + ((A[2]+B[2]). (A[1]+B[1]).(A[0].B[0])) + ((A[2]+B[2]).(A[1]+B[1]).(A[0]+B[0]).Cin)

C[4] = G[3] + P[3].G[2] + P[3].P[2].G[1] + P[3].P[2].P[1].G[0] + P[3].P[2].P[1].P[0].Cin

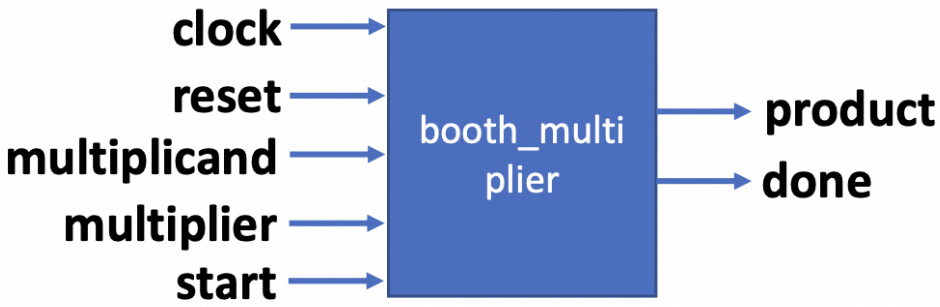
= (A[3].B[3]) + ((A[3]+B[3]).(A[2].B[2])) + ((A[3]+B[3]). (A[2]+B[2]).(A[1].B[1])) + ((A[3]+B[3]). (A[2]+B[2]).(A[1]+B[1]).(A[0].B[0])) + ((A[3]+B[3]). (A[2]+B[2]).(A[1]+B[1]).(A[0]+B[0]).Cin)

C[1],C[2],C[3],C[4] depends on inputs A[0], A[1], A[2], A[3], B[0], B[1], B[2], B[3] and Cin, whose values are known upfront and hence Carry inputs C[1] to C[4] can be computed right away and hence all 4 full adders can perform operation simultaneously to allow fast add operation !

**For homework-5b:**

- **Develop SystemVerilog RTL model for N-bit Booth Multiplier**
  - Develop a Finite state machine and a state transition diagram for the Booth multiplier algorithm.
  - FSM coding style recommended : Single always block with non-blocking assignment statements within always block.
  - Synthesize booth multiplier design for parameter N=4 and run simulation using the testbench provided.
  - Review synthesis results (resource usage and RTL netlist/schematic).
  - Review input and output signals in the simulation waveform and transcripts.
  - Assume the below mentioned primary port names and SystemVerilog RTL module name as booth_multiplier.
  - **Note : FSM code framework is provided in the Lab folder with comments to help develop the code.**
  - **Testbench provided has a built-in checker to ensure design output is expected. See messages in Modelsim transcript window when performing simulation.**

- **Primary Ports for booth_multiplier module**
  - Input clock, reset : posedge clock and asynchronous posedge reset.
  - Input start : 1 cycle pulse generated. Indicates to FSM to start multiplication operation.
  - Input logic[N-1:0] multiplicand, multiplier : Multiplicand and Multiplier inputs to Integer Multiplier.
  - Output logic[(2*N)-1:0] product : Result of multiplication includes carry bit as MSB bit.
  - Output logic done : Indicates that product is available. This is one cycle pulse generated by FSM.



- Multiplier : 4'b1101 (-3), Multiplicand : 4'b1011 (-5), Expected Product = -3 x -5 = 15 (8'b000_1111)
- N-bit Multiplier has N stages of SHIFT and ADD round of computation. Extra bit always initialized with 1'b0.
- If Shift_register[1:0] = 01 Perform ADD of Accumulator + Multiplicand and store back to Accumulator and then perform Arithmetic Right Shift by 1
- If Shift_register[1:0] = 10 Perform ADD of Accumulator + (– Multiplicand) and store back to Accumulator and then perform Arithmetic Right Shift by 1
- If Shift_register[1:0] = 00 or 11 Perform Right Shift by 1

Shift Register — Shift -> Arithmetic Shift

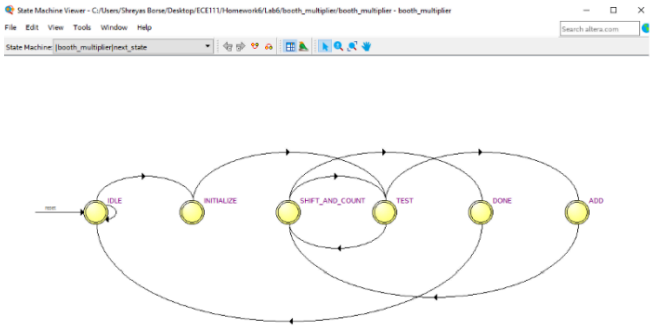
acc + (– multiplicand)
0_0000 + 0_0101 = 0_0101

acc + (multiplicand)
0_0010 + 1_1011 = 1_1101

acc + (– multiplicand)
1_1110+ 0_0101 = 0_0011

| Stage | Carry | Accumulator | Multiplier | Extra Bit | Operation Performed |
|---|---|---|---|---|---|
| 0 | 0 | 0 0 0 0 | 1 1 0 **1** | **0** | INITIALIZE |
| 1 | 0 | 0 1 0 1 | 1 1 0 1 | 0 | ADD |
| | 0 | 0 0 1 0 | 1 1 1 **0** | **1** | SHIFT>>>1 |
| 2 | 1 | 1 1 0 1 | 1 1 1 0 | 1 | ADD |
| | 1 | 1 1 1 0 | 1 1 1 **1** | **0** | SHIFT>>>1 |
| 3 | 0 | 0 0 1 1 | 1 1 1 1 | 0 | ADD |
| | 0 | 0 0 0 1 | 1 1 1 **1** | **1** | SHIFT>>>1 |
| 4 | 0 | **0 0 0 0** | **1 1 1 1** | 1 | DONE |

Product




| | Source State | Destination State | Condition |
|---|---|---|---|
| 1 | ADD | SHIFT_AND_COUNT | |
| 2 | DONE | IDLE | |
| 3 | IDLE | INITIALIZE | (start) |
| 4 | IDLE | IDLE | (!start) |
| 5 | INITIALIZE | TEST | |
| 6 | SHIFT_AND_COUNT | TEST | (!count[0]) + (count[0]).(count[1]) |
| 7 | SHIFT_AND_COUNT | DONE | (count[0]).(count[1]) |
| 8 | TEST | SHIFT_AND_COUNT | (!shift_reg[0]).(!shift_reg[1]) + (shift_reg[0]).(shift_reg[1]) |
| 9 | TEST | ADD | (!shift_reg[0]).(shift_reg[1]) + (shift_reg[0]).(!shift_reg[1]) |

**IDLE**: Wait in this state until Start=1. Then move to INITIALIZE state if input signal Start==1

INITIALIZE: Multiplicand, Multiplicand and Multiplier are loaded into a positive load register, negative load register and a shift register, respectively

TEST: The LSB in the shift register which contains the multiplier is tested to decide the next state. If shift register LSB[1:0] is '01' or '10', then next state is to ADD otherwise next sate is to SHIFT_AND_COUNT

ADD: If LSB[1:0] is '01', the adder adds previous stage add result with Multiplicand, if LSB[1:0] is '10', the Adder adds previous stage add result with negated Multiplicand. The result is stored to the accumulation result, back to shift register and then the state machine transits to SHIFT_AND_COUNT state.

SHIFT_AND_COUNT: If shift register content is right shifted by 1 bit position. MSB of shift register is sign extended (Read about and operators in previous pages of this document)

DONE: Done signal is asserted to '1' when count reaches N and specific bits of shift register content is sent to 'product' output signal . Otherwise to TEST state.

## Submission Requirements

Submit a report on Gradescope in PDF format which includes the following :

**For homework-5a:**

- SystemVerilog design code.
- Synthesis resource usage and schematic generated from RTL netlist viewer.
- Simulation waveform and transcript snapshots and explain the simulation result to confirm that the RTL model developed works as a carry lookahead adder.
- Post-Mapping schematic is optional to submit.
- Explanation of FPGA resource usage in the report is not required.

**For Homework-5b:**

- SystemVerilog FSM design code and State transition diagram (snapshot of hand drawn diagram or Quartus auto-generated diagram, either is acceptable).
- Synthesis resource usage and schematic generated from RTL netlist viewer.
- Simulation waveform and transcript snapshots and explain the simulation result to confirm that the RTL model developed works as a booth multiplier.

**Here is a reference output snapshot:**

multiplier

❑ **Reference Output Snapshot**