LABORATORY MANUAL

SC2103: Digital System Design

Experiment 3:

Pipelining

COLLEGE OF COMPUTING AND DATASCIENCE
NANYANG TECHNOLOGICAL UNIVERSITY

SC2103 Laboratory 3
Pipelining

Aim: To analyze the timing performance of an N-bit 2's complement array multiplier and then increase its operating frequency (data throughput) by strategically inserting pipeline registers and verifying the improvement through timing analysis and simulation.

Objectives:

Task	**Objective Description**
Task 1: Non-Pipelined Analysis	To synthesize the 6-bit x 6-bit 2's complement array multiplier, array_mult_6Bit.v and perform static timing analysis with a 9ns clock period.
Task 2: Non-Pipelined Simulation	To set up and run a behavioural simulation using the AM_top_tb.v testbench, verifying that the non-pipelined multiplier produces the correct 2's complement results after one clock cycle of delay.
Task 3: Pipelining & Optimization	To replace the single multiplier module array_mult_6Bit.v with two sub-modules multA.v and multB.v and insert appropriate pipeline registers to partition the critical path, achieving a shorter clock period of 5.5ns.

Array multiplier implementation

As you will have seen in SC1005, the multiplication of 2's complement binary numbers is more difficult than for unsigned numbers as some of the partial products are negative. In a 2's complement number, the most significant bit has a negative weight (which is why we do NOT refer to it as a sign bit, even though it does indicate if a number is negative or positive). In 2's complement, an N-bit number, *a*, can be specified as:

$$-a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i$$

The product of an N-bit 2's complement number, *a*, with an M-bit 2's complement number, *b*, is given by:

$$\begin{aligned}\mathrm{P} &= \left(-b_{M-1}2^{M-1} + \sum_{j=0}^{M-2} b_j 2^j\right).\left(-a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i\right)\\ &= \sum_{i=0}^{N-2}\sum_{j=0}^{M-2} a_i b_j 2^{i+j} + a_{N-1}b_{M-1}2^{M+N-2} - \left(\sum_{i=0}^{N-2} a_i b_{M-1} 2^{i+M-1} + \sum_{j=0}^{M-2} a_{N-1} b_j 2^{j+N-1}\right)\end{aligned}$$

For a 6-bit x 6-bit multiplier this reduces to:

$$\mathrm{P} = \sum_{i=0}^{4}\sum_{j=0}^{4} a_i b_j 2^{i+j} + a_5 b_5 2^{10} - \left(\sum_{i=0}^{4} a_i b_5 2^{i+5} + \sum_{j=0}^{4} b_j a_5 2^{j+5}\right)$$

This can be seen, in terms of the partial products, where the last two negative terms are represented as the 2's complement of the positive values, as shown in Fig. 1. This can be rearranged as shown in Fig. 2.

							b_5	b_4	b_3	b_2	b_1	b_0
							a_5	a_4	a_3	a_2	a_1	a_0
$\sum_{i=0}^{4}\sum_{j=0}^{4} a_i b_j 2^{i+j}$								b_4a_0	b_3a_0	b_2a_0	b_1a_0	b_0a_0
							b_4a_1	b_3a_1	b_2a_1	b_1a_1	b_0a_1	
						b_4a_2	b_3a_2	b_2a_2	b_1a_2	b_0a_2		
					b_4a_3	b_3a_3	b_2a_3	b_1a_3	b_0a_3			
				b_4a_4	b_3a_4	b_2a_4	b_1a_4	b_0a_4				
$a_5b_5 2^{10}$		b_5a_5										
$-\sum_{i=0}^{4} a_i b_5 2^{i+5}$	1	1	$\overline{a_4b_5}$	$\overline{a_3b_5}$	$\overline{a_2b_5}$	$\overline{a_1b_5}$	$\overline{a_0b_5}$	1	1	1	1	1
												1
$-\sum_{j=0}^{4} b_j a_5 2^{j+5}$	1	1	$\overline{b_4a_5}$	$\overline{b_3a_5}$	$\overline{b_2a_5}$	$\overline{b_1a_5}$	$\overline{b_0a_5}$	1	1	1	1	1
												1
	P_{11}	P_{10}	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Fig. 1: Two's complement multiplication (note: the overbars represent the logical complement)

						b_5	b_4	b_3	b_2	b_1	b_0
						a_5	a_4	a_3	a_2	a_1	a_0
					1	$\overline{b_5a_0}$	b_4a_0	b_3a_0	b_2a_0	b_1a_0	b_0a_0
					$\overline{b_5a_1}$	b_4a_1	b_3a_1	b_2a_1	b_1a_1	b_0a_1	
				$\overline{b_5a_2}$	b_4a_2	b_3a_2	b_2a_2	b_1a_2	b_0a_2		
			$\overline{b_5a_3}$	b_4a_3	b_3a_3	b_2a_3	b_1a_3	b_0a_3			
		$\overline{b_5a_4}$	b_4a_4	b_3a_4	b_2a_4	b_1a_4	b_0a_4				
1	b_5a_5	$\overline{b_4a_5}$	$\overline{b_3a_5}$	$\overline{b_2a_5}$	$\overline{b_1a_5}$	$\overline{b_0a_5}$					
P_{11}	P_{10}	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Figure 2. Two's complement multiplication (rearranged) with 6 partial products

This is the standard Baugh-Wooley 2's complement multiplication algorithm which results in the array multiplier in Fig. 3. Note that there is no need for sign extension of the partial products. The Baugh-Wooley array multiplier usually is implemented using carry-save adder modules as this reduces the delay through the array. However, in FPGA, ripple carry adders are fast because of the fast carry chain logic within a logic slice, so it is easier to use standard adders (as shown in Fig. 3).

Task 1

1. Create a new project called Lab3 in Xilinx Vivado, targeting the Basys 3 board (Artix-7 xc7a35tcpg236-1 FPGA). **Remember to use a local drive location NOT a network drive.** Download the lab3.zip file from NTUlearn and unzip it into the Lab3 directory.
2. Add the Verilog design files (*AM_top.v* and *array_mult_6Bit.v*) to the design (select **+** OR **Add Sources** from **PROJECT MANAGER** in the **Flow Navigator** window). Note *array_mult_6Bit.v* is a non-parameterised 6-bit version of the multiplier that directly matches to Fig. 3.
3. Also add the constraints file (*Lab3.xdc*). The *Lab3.xdc* file contains the pin constraints for the design. Change the clock constraint to a 9ns clock (as shown below).
   create_clock -add -name sys_clk_pin -period 9 -waveform {0 4.5} [get_ports clk]
4. Run **SYNTHESIS** in the **Flow Navigator** window for the 6 bit multiplier. Select **Reports** in the bottom window, then open the *synthesis report* and verify that only the resource expected has been used (2 x 6-bit registers and 1 x 12-bit register and 5 x 7-bit adders).

5. From **SYNTHESIS** in the **Flow Navigator** window, select **Report Timing Summary.** Click **OK** at the pop-up window. Then in the bottom window, select "Timing" and verify that all the **user specified timing constraints are met**. Note that the *Worst Negative Slack* (WNS) and *Worst hold Slack* (WHS) should be fractions of a ns, meaning that we could not operate the circuit (the clock) much faster. That is, not much below 8.5ns (9ns is the current value).
6. Note that WNS is positive means, even the *slowest* (worst-case) data path meets timing with margin. With a 9 ns clock, Longest data path delay (shown by the red line in Fig.3) $\approx$9ns- WNS. We still have WNS of safety margin.
7. **IMPORTANT: Use a sniping tool to keep a copy of the "Design Timing Summary" window for assessment purposes.**

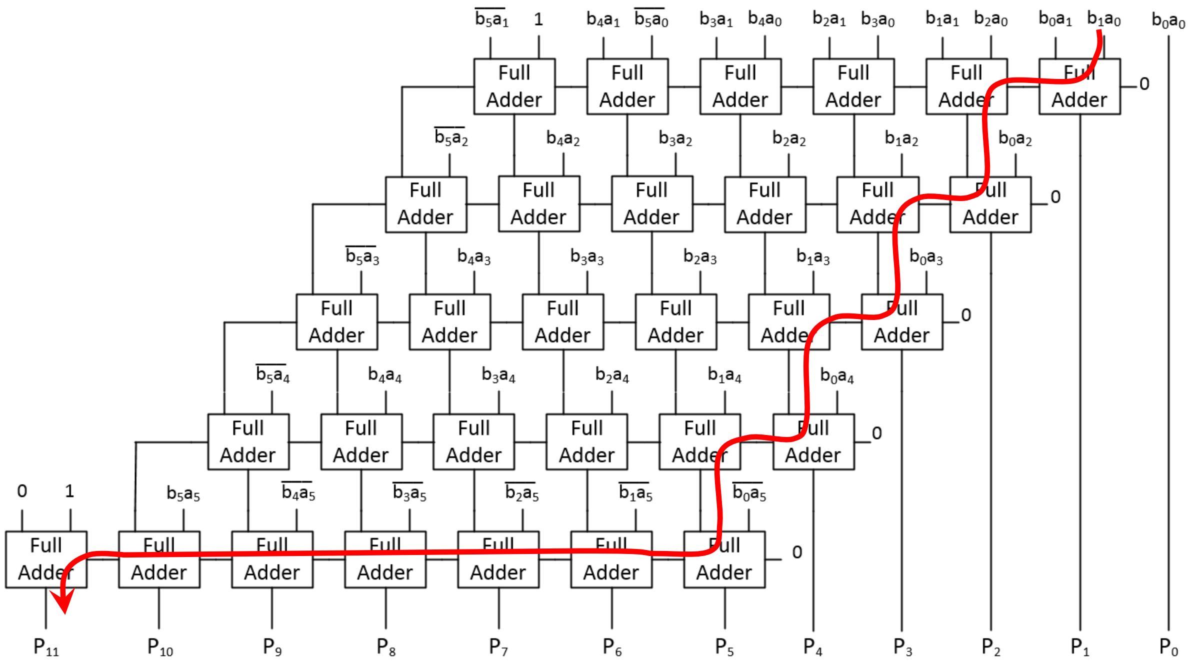

Fig. 3: The 6-bit x 6-bit 2's complement array multiplier implementation

Task 2 (Simulation of multiplier)

1. **Close the SYNTHESIDED DESIGN window** and then select add Sources for simulation. Select **Add or create simulation sources**, then **Next**. Select **Add files**, and then select *AM_top_tb.v (testbench)* and click **OK**. Click **Finish**. The test bench instantiates the top module and provides a 10ns clock and selected multiplier inputs.
2. Run the Behavioural Simulation after making sure that *AM_top_tb.v* is the **top module** under simulation sources. In the waveform you cannot see a_r and b_r. To select them, select *uut* from **Scope** in the **SIMULATION** window. In the **Objects** window select *a_r* and *b_r* and move them to the **Name** section of the waveform window (the black part). This will add the registered multiplier inputs (e.g. aligned to the clock) to the simulation. Move *result[11:0]* to the bottom (below the other signals). Then ***Restart*** and ***Run All*** or **Relaunch** to redo the simulation with the new signals. Change the radix to signed decimal and check the multiplier gives the correct results.
3. The waveform window should look like Fig 4. Note the registered signals are aligned to the clock and the result is delayed by 1 clock cycle.
4. Understand the working and the correctness of the circuit. Once you finish the same, close the SIMULATION window. We have finished understanding the nonpipelined multiplier.

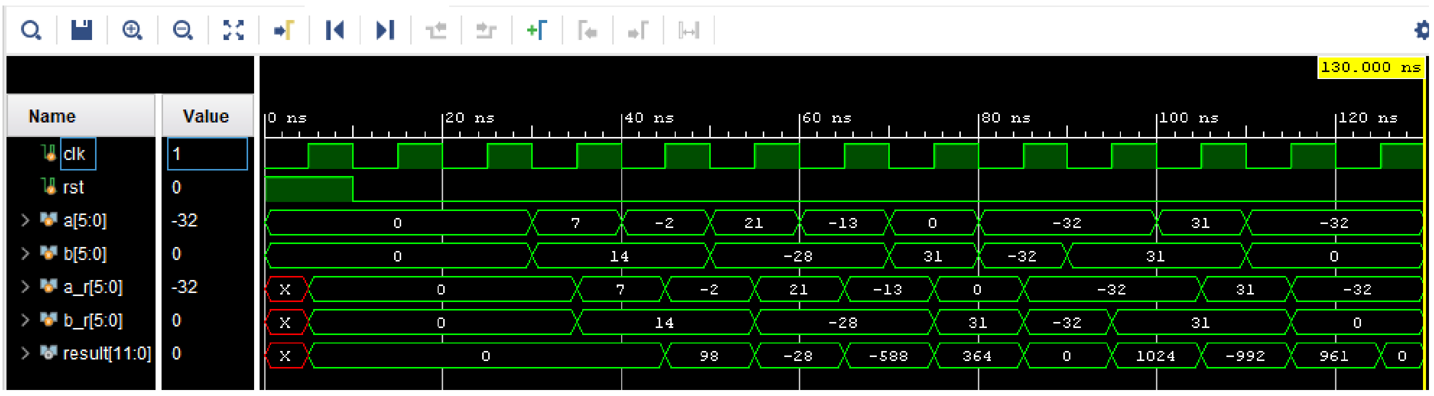
Name
Value
clk 1
rst 0
a[5:0] -32
b[5:0] 0
a_r[5:0] -32
b_r[5:0] 0
result[11:0] 0
130.000 ns
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
a: 0, 7, -2, 21, -13, 0, -32, 31, -32
b: 0, 14, -28, 31, -32, 31, 0
a_r: X, 0, 7, -2, 21, -13, 0, -32, 31, -32
b_r: X, 0, 14, -28, 31, -32, 31, 0
result: X, 0, 98, -28, -588, 364, 0, 1024, -992, 961, 0

Fig. 4: Simulation waveform for non-pipelined multiplier

Task 3 (Pipelining multiplier design)

1. First, examine the block diagram of the multiplier shown in Fig. 3. Note that the critical path is from the inputs to the full adder (FA) block at the top right (b_0a_1 and b_1a_0) to the output from the FA block at the bottom left (P_{11}) and is 15 FA delays.
2. So we need to partition the adder somewhere such that the critical path in the two partitions stays roughly the same. From Fig. 3, inserting a pipeline register between the 3rd and 4th adder blocks, gives a critical path of 10 FA delays for the top partition (9 for the bottom). The pipelined design is as shown in Fig 5 (and should use the same signals).
3. Create a new project in Vivado with all the initial settings.

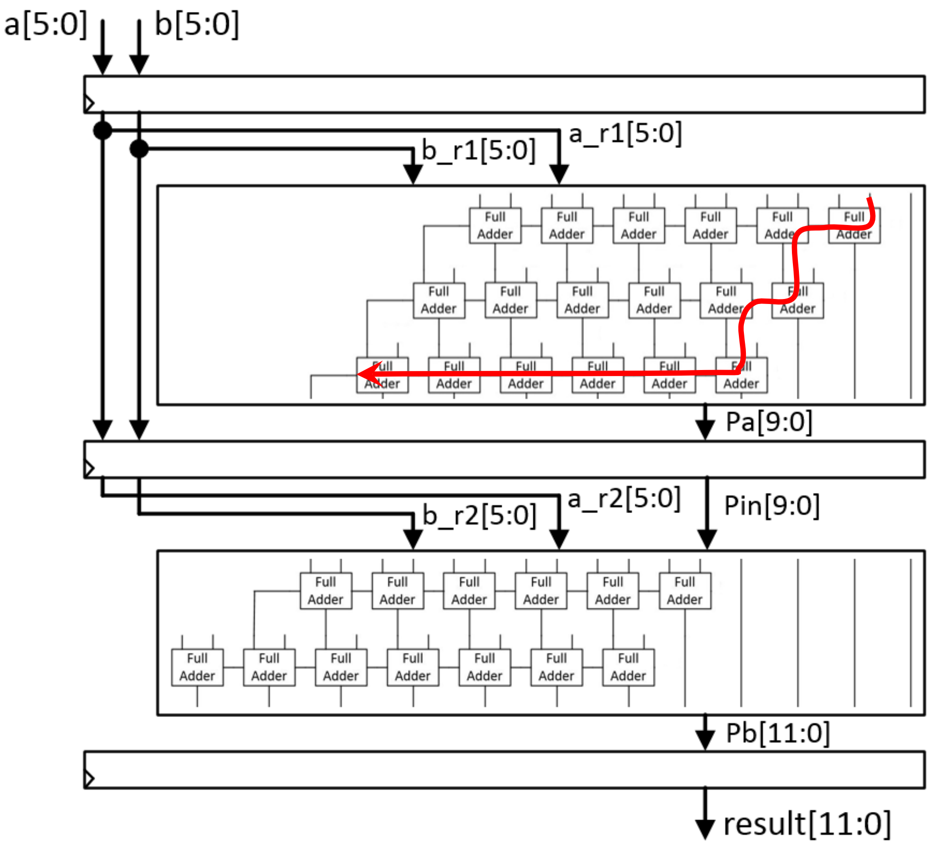

Fig. 5: The block diagram of the pipelined multiplier

4. Add *multA.v* and *multB.v* to the design. Open *multA.v* and check that it only uses the first three partial products and that the 10-bit *P* is assigned as the sum of *PP3* and *Isum2* (plus some lower terms).
5. Next open *multB.v*. It uses the original *a* and *b* inputs as well as the 10-bit product (*Pin*) generated in *multA.v* and produces a 12-bit result (*P*). Verify that the two new modules match those shown in Fig. 5.
6. Then modify *AM_top.v*
    a. Remove the *array_mult_6Bit* instantiation and instantiate the two new multiplier modules
    b. Check the input and output of *multA.v and multB.v* and add wires accordingly.
    c. Do remember that *multB.v* has an extra input (*Pin*).
    d. You will also need to add appropriate pipeline registers of the correct bit widths. The pipeline register outputs (inputs to the two modules) should be declared as type reg while the module outputs will be declared as wires.
7. Next modify the always @ (posedge clk) block.
    a. In the if (rst) block, set all pipeline regs and result to 0.
    b. In the else block, implement the 2-stage pipeline shift.
    c. Load stage-1 regs from inputs and shift stage-1 regs into stage-2 regs.
8. Once done verify the design by making sure that both *multA.v and multB.v* are instantiated under *AM_top.v* . DO synthesis for *AM_top.v* to understand if there is any programming errors. Fix them.
9. Open the constraints file (*Lab.xdc*) and edit the clock constraint, and change to:
   create_clock -add -name sys_clk_pin -period 5.5 -waveform {0 2.75} [get_ports clk]
10. Now, synthesize the pipelined design. There are several additional warnings in the design relating to unconnected ports. These can be ignored as a quick check of the *multA.v* and *multB.v* Verilog code (in the "generate partial products" section) will show that *multA* does not use *a[5:4]* and *multB* does not use *a[3:0]*. Again "Open (the) Synthesised Design" and select "Report Timing Summary". Then OK.
11. Then select "Timing" and verify that the **user specified timing constraints** are met. Note in the unpipelined design we used a 9ns clock (111.1MHz), whereas here we are now using a 5.5ns clock (181.8MHz). This represents a 64% increase in clock frequency due to pipelining. A significant increase.
12. **IMPORTANT: Use a sniping tool to keep a copy of the "Design Timing Summary" window for assessment purposes.**
13. Simulate the design. You should be able to use the same test bench from Task 2. At the simulation window, add the 5 pipeline register outputs (*a_r1*, *a_r2*, *b_r1*, *b_r2* and *Pin*) to the simulation window. Move *result[11:0]* to the bottom. Then add a divider and add *Pa* and *Pb* at the bottom. ***Restart*** and ***Run All*** or ***Relaunch*** to redo the simulation with the new signals. Verify the results are correct.
14. Why is *result* just after reset non-zero?

Inform your lab supervisor once you reach this point. You need to show the simulation for the pipelined code.

Optional exercise:

1. The multiplier given to you is a non-parametrizable one for easier understanding. Open the file *array_mult_parameterised.v* in a text editor. This code will generate any arbitrary N-bit x N-bit multiplier. It is currently set to an 8-bit x 8-bit multiplier. But remember Verilog allows you to override the parameters in an upper level module. To generate a 6-bit multiplier, we could use:
   ```
   array_mult #(.SIZE(6)) uut (.a(a), .b(b), .P(P));
   ```
2. Spot the generation of the partial products and then the intermediate sums and the final result.
3. Compare the two implementations (parametric and non-parametric for 6 bit multiplier). Note, both use the same resources and produce identical hardware.

Note: The following warnings can be safely ignored

Synthesis (Task 1):

```
WARNING: [Constraints 18-5210] No constraints selected for write.
```

Synthesis (Task 3):

```
WARNING: [Synth 8-3331] design multB has unconnected port a[3] (5 more like this)
WARNING: [Constraints 18-5210] No constraints selected for write.
```

The simulation for Task 3