CE2003 Laboratory 3

Pipelining

Many digital systems benefit significantly (in terms of data throughput) by using pipelining. In this lab, we will take an up-pipelined 2's complement array multiplier, implemented in Verilog, and add pipelining. The multiplier is parameterised, and so can be adapted to support different bit widths, with the constraint that the width of the multiplicand is the same as that for the multiplier. That is, the multiplier is N-bit x N-bit and produces a 2N-bit result.

Array multiplier implementation

As you will have seen in CS1105, 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:

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

$$= \sum_{i=0}^{N-2}\sum_{j=0}^{M-2} a_i b_j 2^{i+j} + a_{N-1}b_{M-1}2^{M+N-2} - \left(\sum_{i=0}^{N-2} a_i b_{M-1} 2^{i+M-1} + \sum_{j=0}^{M-2} a_{N-1} b_j 2^{j+N-1}\right)$$

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 Figure 1. This can be rearranged as shown in Figure 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

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

This is the standard Baugh-Wooley 2's complement multiplication algorithm which results in the array multiplier in Figure 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 Figure 3). The parameterised Verilog code for implementing N-bit x N-bit array multiplier is given in NTUlearn.

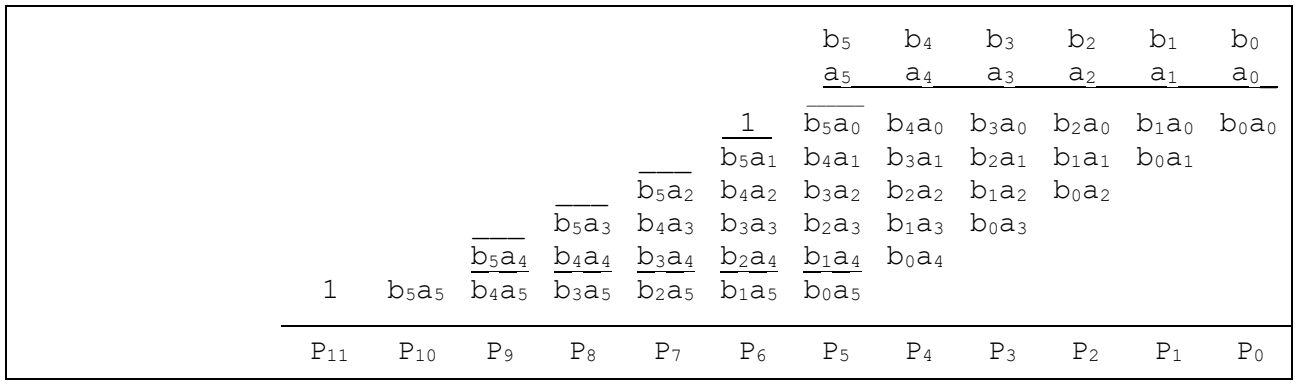

Figure 2. Two's complement multiplication (rearranged) with 6 partial products

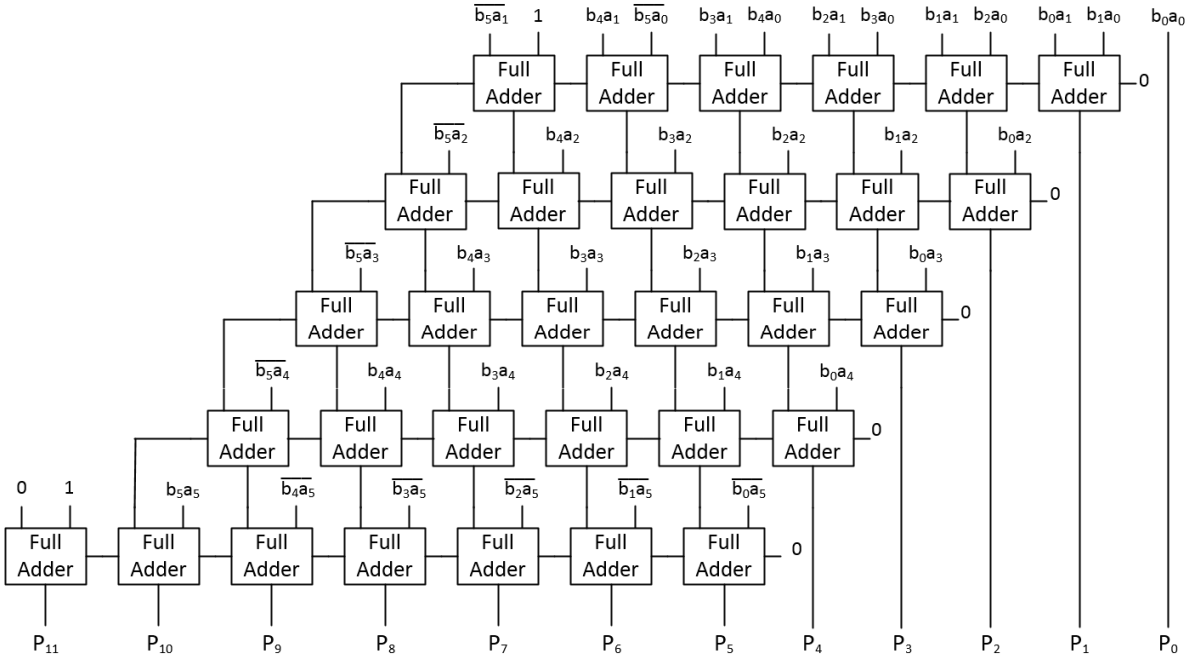

Figure 3. The 6-bit x 6-bit 2's complement array multiplier implementation

Task 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.

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));

Spot the generation of the partial products and then the intermediate sums and the final result. However, this code is a bit difficult to understand, and so we will use a non-parameterised 6-bit version for this lab. Next, open *array_mult_6Bit.v* in a text editor. This code directly matches to Figure 3. Compare the two implementations. Note, both use the same resources and produce identical hardware. Close the files.

Then 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). 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) and **synthesise the design**.

create_clock -add -name sys_clk_pin -period 9 -waveform {0 4.5} [get_ports clk]

Select **Reports** in the bottom window, then **o**pen the *synth_1_synth_ synthesis_report_0* 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).

Then **Open Synthesized Design** from **SYNTHESIS** in the **Flow Navigator** window. Then select **Report Timing Summary.** Click **OK** at the pop-up window. Then in the bottom window, select "Timing" and verify that all of 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). **IMPORTANT: Use a sniping tool to keep a copy of the "Design Timing Summary" window for assessment purposes.**

Task 2

Next we will simulate the design. **Close the SYNTHESIDED DESIGN window** and then select **Add Sources**. Select **Add or create simulation sources**, then **Next**. Select **Add files**, and then select *AM_top_tb.v* and click **OK**. Click **Finish**. The test bench instantiates the top module and provides a 10ns clock and selected multiplier inputs.

Run the Behavioural Simulation. 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*** to redo the simulation with the new signals. Change the radix to signed decimal and check the multiplier gives the correct results. The waveform window should look like Figure 4. Note the registered signals are aligned to the clock and the result is delayed by 1 clock cycle.

Task 3

Now we should pipeline the multiplier design. First, examine the block diagram of the multiplier shown in Figure 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 11 FA delays. So we need to partition the adder somewhere such that the critical path in the two partitions stays roughly the same. From Figure 3, it can be seen that inserting a pipeline register between the 3rd and 4th adder blocks, we get a critical path of 8 FA delays for both partitions. The pipelined design should look like (and will use the same signals) as shown in Figure 5.

Close the SIMULATION window. Remove *array_mult_6Bit.v* from the project (right click on it and then select "Remove File from project"). Then 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). 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 Figure 5.

Then modify AM_top to remove the *array_mult_6Bit* instantiation and instantiate the two new multiplier modules (remember that *multB.v* has an extra input (*Pin*)). 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. Next modify the always @ (posedge clk) block.

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]

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.

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. **IMPORTANT: Again, use a sniping tool to keep a copy of the "Design Timing Summary" window for assessment purposes.**

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*** to redo the simulation with the new signals. Verify the results are correct. Why is *result* just after reset non-zero?

Inform your lab supervisor once you reach this point

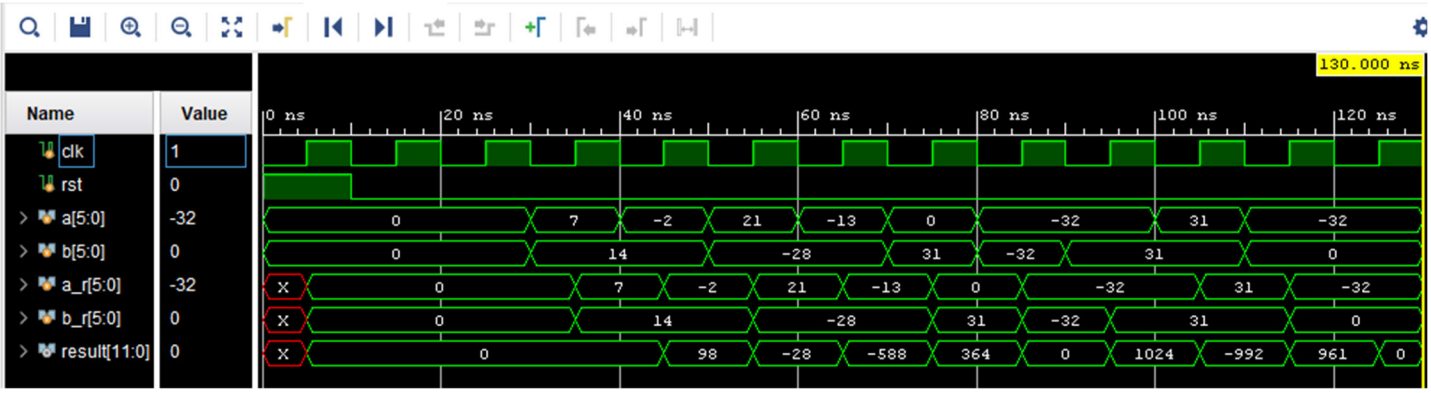

Figure 4. Simulation waveform for non-pipelined multiplier

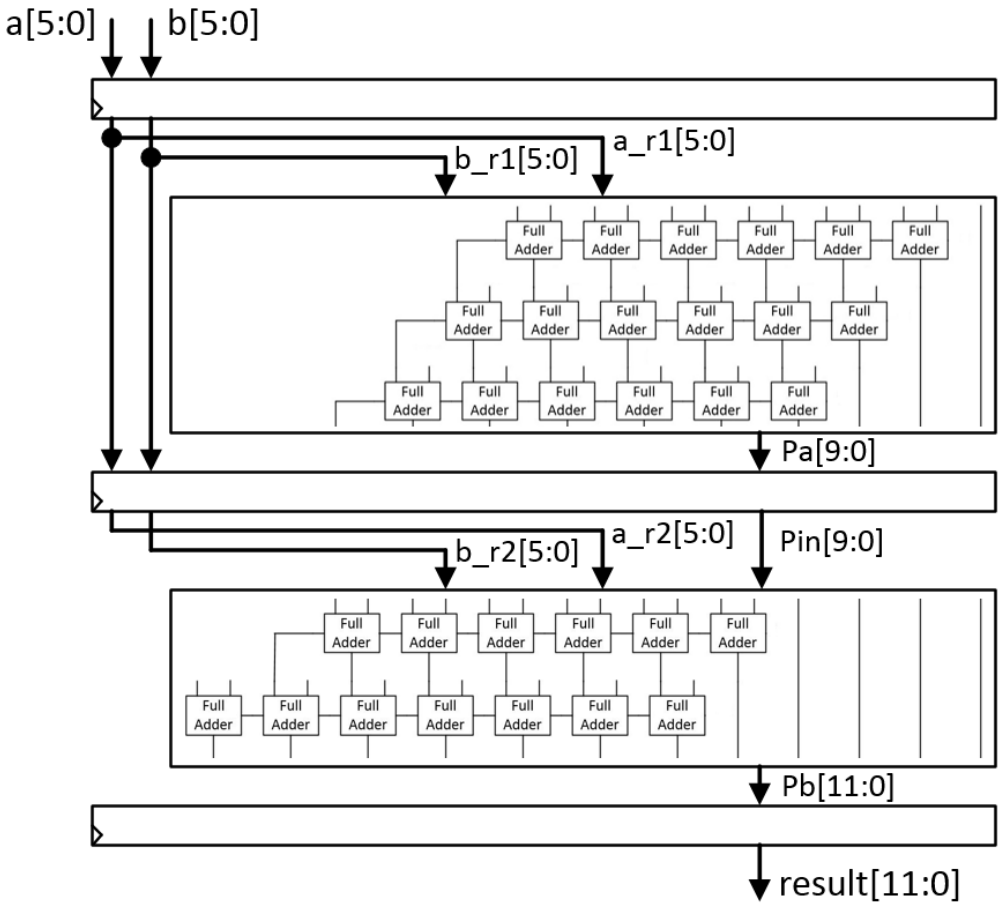

Figure 5. The block diagram of the pipelined multiplier