Chip name: SUNCHILD
Date: 12/06/2010
Author Contact:
  Ryan Child: (513) 629-0800 (cell); ryan.child@gmail.com
  Xinyu Sun: (203) 243-6298 (cell); sunxinyu19841114@hotmail.com
**Table of Contents**

**Chapter 1 - First Progress Report** ................................................................. 1-5
  Pin-Out Diagram ......................................................................................... 1-6
  Explanation of Chip Function and Test Mode ........................................... 1-8
    Comments – Ryan Child ........................................................................... 1-8
    Comments – Xinyu Sun ........................................................................... 1-10
  Major Design Decisions ............................................................................. 1-10
    Comments – Ryan Child ........................................................................... 1-10
    Comments – Xinyu Sun ........................................................................... 1-11
  Behavioral VHDL model without Test Mode ............................................. 1-12
  Behavioral VHDL model including Test Mode ......................................... 1-15
  Top-level Block Diagram of the SUNCHILD1049 Architecture .................. 1-18
  Bit-slicing Scheme ..................................................................................... 1-19
  Division of Work ......................................................................................... 1-21

**Chapter 2 - Second Progress Report** ............................................................. 2-1
  2. Revised First Progress Report ............................................................... 2-2
    Explanation of Chip Function and Test Mode ........................................ 2-2
  3. Logic Hierarchy .................................................................................... 2-3
    Top-level ................................................................................................. 2-3
    Bitslice .................................................................................................. 2-4
    Controller ............................................................................................... 2-5
  4. Layouts & Layout Simulations ............................................................... 2-5
    Bitslice .................................................................................................. 2-5
    Layout .................................................................................................... 2-5
    SPICE Simulation .................................................................................. 2-5
    Controller ............................................................................................... 2-10
    Layout .................................................................................................... 2-10
    IRSIM Simulation .................................................................................. 2-10
  5. VHDL Code & Simulations .................................................................... 2-11
    Notes on delay time ............................................................................... 2-11
    Code: Chip ............................................................................................. 2-12
    Model ...................................................................................................... 2-12
    Testbench ............................................................................................... 2-15
    Code: Bitslice ......................................................................................... 2-16
    Model ...................................................................................................... 2-16
    Testbench ............................................................................................... 2-19
    Code: Controller ..................................................................................... 2-24
    Model ...................................................................................................... 2-24
    Testbench ............................................................................................... 2-27
    Code: Leaf-level (gates) ......................................................................... 2-28
    DFFPOSX1 ............................................................................................ 2-28
    MUX2X1 .................................................................................................. 2-28
    NAND2X1 ............................................................................................... 2-29
    NAND3X1 ............................................................................................... 2-29
    NOR2X1 ................................................................................................. 2-30
    NOR3X1 ................................................................................................. 2-30
    OR2X1 .................................................................................................... 2-30
    XOR2X1 .................................................................................................. 2-31
    INVX1 ..................................................................................................... 2-32
    FAX1 ....................................................................................................... 2-32
    Comparison with VHDL of First Progress Report .................................. 2-32
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation: Chip</td>
<td>2-32</td>
</tr>
<tr>
<td>Simulation: Bitslice</td>
<td>2-33</td>
</tr>
<tr>
<td>6. Floor Plan</td>
<td>2-34</td>
</tr>
<tr>
<td>7. Major Design Decisions</td>
<td>2-35</td>
</tr>
<tr>
<td>8. Division of Work</td>
<td>2-36</td>
</tr>
<tr>
<td>Chapter 3 - Third Progress Report</td>
<td>3-1</td>
</tr>
<tr>
<td>Revised First and Second Progress Reports</td>
<td>3-2</td>
</tr>
<tr>
<td>Pinout</td>
<td>3-2</td>
</tr>
<tr>
<td>Bit-slice Layout</td>
<td>3-4</td>
</tr>
<tr>
<td>VHDL</td>
<td>3-5</td>
</tr>
<tr>
<td>Layout</td>
<td>3-8</td>
</tr>
<tr>
<td>IRSIM/SPICE Simulations</td>
<td>3-10</td>
</tr>
<tr>
<td>IRSIM .cmd file:</td>
<td>3-10</td>
</tr>
<tr>
<td>Simulation of normal mode</td>
<td>3-13</td>
</tr>
<tr>
<td>Simulation of test mode</td>
<td>3-14</td>
</tr>
<tr>
<td>Simulation strategy explanation</td>
<td>3-14</td>
</tr>
<tr>
<td>Chip clock frequency</td>
<td>3-15</td>
</tr>
<tr>
<td>VHDL</td>
<td>3-15</td>
</tr>
<tr>
<td>Final chip simulation</td>
<td>3-18</td>
</tr>
<tr>
<td>Test Strategy</td>
<td>3-18</td>
</tr>
<tr>
<td>Standard Cells</td>
<td>3-19</td>
</tr>
<tr>
<td>Bit-slice</td>
<td>3-19</td>
</tr>
<tr>
<td>Test mode / normal mode</td>
<td>3-19</td>
</tr>
<tr>
<td>Internal signals</td>
<td>3-20</td>
</tr>
<tr>
<td>User Guide</td>
<td>3-20</td>
</tr>
<tr>
<td>Major Design Decisions</td>
<td>3-21</td>
</tr>
<tr>
<td>Division of Work</td>
<td>3-21</td>
</tr>
</tbody>
</table>
Lists of Figures and Tables

Figure 1: Pin-out diagram .................................................................................................................. 1-6
Figure 2: Writing $AB = \{0101\} \{0111\}$ to a N=4 chip ................................................................. 1-9
Figure 3: Product of multiplication $P = 5 \times 7 = 35 (00011111)$ ............................................. 1-9
Figure 4: Waveform for the function simulation ................................................................................. 1-15
Figure 5: waveform for the test mode simulation ............................................................................. 1-18
Figure 6: Top-level diagram of SUNCHILD1049 architecture. Non-scan-chain test inputs and outputs not shown .................................................................................................................. 1-19
Figure 7: Bitslice circuit ..................................................................................................................... 1-20
Figure 8: Writing $AB = \{0101\} \{0111\}$ to a N=4 chip ................................................................. 2-2
Figure 9: Product of multiplication $P = 5 \times 7 = 35 (00011111)$ ............................................. 2-2
Figure 10: Top-level schematic of SUNCHILD1049 ....................................................................... 2-3
Figure 11: Bitslice schematic ............................................................................................................ 2-4
Figure 12: Updated controller schematic .......................................................................................... 2-5
Figure 13: Bitslice Layout .................................................................................................................. 2-5
Figure 14: Bitslice SPICE test 0 ....................................................................................................... 2-8
Figure 15: Bitslice SPICE test 1 ....................................................................................................... 2-8
Figure 16: Bitslice SPICE test 2 ....................................................................................................... 2-9
Figure 17: Bitslice SPICE test 3 ....................................................................................................... 2-9
Figure 18: Bitslice SPICE test 4 ....................................................................................................... 2-10
Figure 19: Controller Layout ........................................................................................................... 2-10
Figure 20: IRSIM Simulation of controller. Timestep = 10ns ............................................................ 2-11
Figure 21: VHDL simulation of bitslice with delays from cell library datasheets .............................. 2-34
Figure 22: Floorplan .......................................................................................................................... 2-35
Figure 23: Revised pin-out diagram of SUNCHILD multiplier .......................................................... 3-2
Figure 24: Revised bitslice layout. Dimensions: 1152 x 1082 ......................................................... 3-5
Figure 25: Hierarchical Layout showing placement and ordering of slices and controller .............. 3-9
Figure 26: Chip layout including pad frame ...................................................................................... 3-10
Figure 27: normal mode irsim simulation ......................................................................................... 3-13
Figure 28: test mode irsim simulation ............................................................................................... 3-14
Figure 29: chip simulation ................................................................................................................ 3-18
Figure 30: Timing diagram for SUNCHILD multiplier ...................................................................... 3-21

Table 1: Description of Pins .............................................................................................................. 1-8
Table 2: Division of Work .................................................................................................................. 1-21
Table 3: Controller component delays ............................................................................................. 2-12
Table 4: Bislice component delays .................................................................................................. 2-12
Table 5: Description of task allocation changes ............................................................................... 2-37
Table 6: Division of Work ................................................................................................................ 2-37
Table 7: Pin numberings and description ......................................................................................... 3-4
Table 8: Pins for testing bit-slice ..................................................................................................... 3-19
Chapter 1 - First Progress Report
Pin-Out Diagram

The pin-out for the SUNCHILD1049 is shown as Figure 1 and pin definitions are given in Table 1.

### Figure 1: Pin-out diagram

<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLK</td>
<td>I</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>RESET</td>
<td>I</td>
<td>Synchronous Reset. Toggle CLK while driving</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>I</td>
<td>Factor A Serial Input</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>I</td>
<td>Factor B Serial Input</td>
</tr>
<tr>
<td>5</td>
<td>WR</td>
<td>I</td>
<td>Write. Drive to 0V for N clock cycles to simultaneously read factors into shift registers.</td>
</tr>
<tr>
<td>6</td>
<td>BUSY</td>
<td>O</td>
<td>High during multiplication. Transitions low when multiplication is complete and latched into internal shift register.</td>
</tr>
<tr>
<td>7</td>
<td>P</td>
<td>O</td>
<td>Product Serial Output</td>
</tr>
<tr>
<td>8</td>
<td>SI</td>
<td>I</td>
<td>Scan Chain Input</td>
</tr>
<tr>
<td>9</td>
<td>SE</td>
<td>I</td>
<td>Scan Chain Input Enable</td>
</tr>
<tr>
<td>10</td>
<td>SO</td>
<td>O</td>
<td>Scan Chain Output</td>
</tr>
<tr>
<td>11</td>
<td>SSI</td>
<td>I</td>
<td>Test Slice Scan In</td>
</tr>
<tr>
<td>12</td>
<td>SSE</td>
<td>I</td>
<td>Test Slice Scan Enable</td>
</tr>
<tr>
<td>13</td>
<td>SSO</td>
<td>O</td>
<td>Test Slice Scan Out</td>
</tr>
<tr>
<td>14</td>
<td>SR</td>
<td>I</td>
<td>Test Slice Reset</td>
</tr>
<tr>
<td>15</td>
<td>SWR</td>
<td>I</td>
<td>Test Slice Write</td>
</tr>
<tr>
<td>16</td>
<td>GND</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>17</td>
<td>SSR</td>
<td>I</td>
<td>Test Slice Shift Result</td>
</tr>
<tr>
<td>18</td>
<td>SAN</td>
<td>I</td>
<td>Test Slice A&lt;sub&gt;0&lt;/sub&gt;. Connects to Aout of last slice in chain.</td>
</tr>
<tr>
<td>19</td>
<td>SAI</td>
<td>I</td>
<td>Test Slice Ain</td>
</tr>
<tr>
<td>20</td>
<td>SAO</td>
<td>O</td>
<td>Test Slice Aout</td>
</tr>
<tr>
<td>21</td>
<td>SBL</td>
<td>I</td>
<td>Test Slice B Left. Connects to Bout of slice to the left.</td>
</tr>
<tr>
<td>22</td>
<td>SBR</td>
<td>I</td>
<td>Test Slice B Right. Connects to Bout of slice to the right.</td>
</tr>
<tr>
<td>23</td>
<td>SBO</td>
<td>O</td>
<td>Test Slice Bout</td>
</tr>
<tr>
<td>24</td>
<td>SC</td>
<td>I</td>
<td>Test Slice clock</td>
</tr>
<tr>
<td>25</td>
<td>SUI</td>
<td>I</td>
<td>Test Slice Uin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Description of Pins</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---------------------</td>
</tr>
<tr>
<td>26</td>
<td>SUO</td>
<td>O</td>
<td>Test Slice Uout</td>
</tr>
<tr>
<td>27</td>
<td>SVI</td>
<td>I</td>
<td>Test Slice Vin</td>
</tr>
<tr>
<td>28</td>
<td>SVO</td>
<td>O</td>
<td>Test Slice Vout</td>
</tr>
<tr>
<td>29</td>
<td>SVDD</td>
<td>---</td>
<td>Test Slice VDD</td>
</tr>
<tr>
<td>30</td>
<td>SVGD</td>
<td>---</td>
<td>Test Slice GND</td>
</tr>
<tr>
<td>31</td>
<td>U2</td>
<td>O</td>
<td>U of third-to-last slice in chain (bit 2)</td>
</tr>
<tr>
<td>32</td>
<td>V2</td>
<td>O</td>
<td>V of third-to-last slice in chain (bit 2)</td>
</tr>
<tr>
<td>33</td>
<td>A0</td>
<td>O</td>
<td>A of last slice in chain (bit 0, LSB)</td>
</tr>
<tr>
<td>34</td>
<td>B0</td>
<td>O</td>
<td>B of last slice in chain (bit 0, LSB)</td>
</tr>
<tr>
<td>35</td>
<td>U0</td>
<td>O</td>
<td>U of last slice in chain (bit 0, LSB)</td>
</tr>
<tr>
<td>36</td>
<td>VDD</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>37</td>
<td>A1</td>
<td>O</td>
<td>A of second-to-last slice in chain (bit 1)</td>
</tr>
<tr>
<td>38</td>
<td>B1</td>
<td>O</td>
<td>B of second-to-last slice in chain (bit 1)</td>
</tr>
<tr>
<td>39</td>
<td>U1</td>
<td>O</td>
<td>U of second-to-last slice in chain (bit 1)</td>
</tr>
<tr>
<td>40</td>
<td>V1</td>
<td>O</td>
<td>V of second-to-last slice in chain (bit 1)</td>
</tr>
</tbody>
</table>

**Table 1: Description of Pins**

**Explanation of Chip Function and Test Mode**

**Comments – Ryan Child**
To load the two factors A and B into the chip and begin multiplication, the user must simultaneously shift A and B into the internal shift registers while driving the WR pin to GND. Both N-bit serial inputs must be padded with exactly N zeros to the left, so that the write time is exactly 2N clock cycles. The timing diagram for multiplying \( A = 5 \) (0101) and \( B = 7 \) (0111) is shown below.
Figure 2: Writing AB = \{0101\} \{0111\} to a N=4 chip

When the \(WR\) signal is returned to VDD, the multiplication is initiated and the BUSY signal goes high. The BUSY signal returns to 0V when the multiplication is complete and can be used to latch the outgoing product. The BUSY signal can also be connected in series with the \(WR\) pin of another SUNCHILD1049 to perform multiplications with more than two factors, although it would need to be gated with a counter such that the \(WR\) input of the other chip would return to 5V after 2N clock cycles. We are still contemplating whether to include a 2N-bit counter in our design. Whether we do will depend on the size of the slice layout and the value of N.

After the BUSY signal transitions low to signal completion, the result is shifted out immediately. The result is provided LSB first, as shown in Figure 3 below.

Test Mode simply connects all DFFs in the design together. This is accomplished with one MUX for each DFF, controlled by a global scan chain input enable (SE) signal. To verify the correct functioning of all DFFs in the chip, the user may drive SE to VDD, serially input a scan chain of length 4N into the scan chain input (SI) pin, and read the output at the scan chain output (SO) pin. To test for a stuck-at fault, any scan chain except for a chain of all 0s or all 1s may be chosen, and a fault is identified somewhere along the chain if the chain output is all 0s or all 1s. The faulty DFF cannot be identified using this method alone, however. To find the faulty DFF, one must completely scan in the chain and then return SE to GND, setting the chip into its normal mode of operation. In this way, the internal state of the chip can be set and the outputs are read using the product (P) pin and extra test pins.

Only 12 pins are needed for the functional, scan, VDD and GND pins. The other 28 pins are allocated to allow access to an isolated bitslice, and the registers of the last few slices in the chain. This will allow us to fully test the slice independently and verify the combinational logic of the last few bits in the chain.
Comments – Xinyu Sun
In order to make sure all the DFFs in our design are working normally, we have to design a test mode to inspect the behaviors of all the DFFs. In this design a scan chain technique is used for test. The objective of the scan chain is to make testing easier by providing a simple way to set and observe every DFFs in an integrated circuit. A special signal called enable is added to a design. When this signal is asserted, every flip-flop in the design is connected into a long shift register. one input pin provides the data to this chain, and one output pin is connected to the output of the chain. Then using the chip's clock signal, an arbitrary pattern can be entered into the chain of flips flops, and the state of every flip flop can be read out.

Thus in our design, we choose test_en (VHDL) / SE (pinout) as an enable signal. The DFFs work in test mode, and the input signal which is Test_in (VHDL) / SI (pinout) will be sent to the A_reg, and then the signal will be passed to B_reg, and then can be passed to all the DFFs we need in our design. After that the input signal will come out from the last DFF of the chain. We can compare the output signal with the input signal to see whether the DFFs are working correct. When the test_en is set to low, the circuit will work in normal function mode.

Major Design Decisions

Comments – Ryan Child
We decided to implement the fast-array version of the Russian Peasant Algorithm for its speed. Of course, this comes at the cost of more space for the extra two DFFs and MUXs per bitslice. We hope to minimize the bitslice anyway through an efficient layout and achieve an N that approaches what would be possible with the slow array.

A bitsliced design was chosen for its simplicity; Once we have designed and simulated the bitslice we will have completed most of the layout work for the project. By using the MAGIC getcell and array commands we will be able to complete the chain of slices with just a few keystrokes. After laying out the slice chain, we will need only to layout the controller and route global signals.

To initiate the shifting-out of the result, the controller will generate a signal, shift_res, when the U register contains the correct product. This signal can be obtained two ways: using a (2N-1)-bit counter or with the expression (A==0x00 & & V==0x00 & & BUSY==1). The counter would occupy less area but would slow down the operation as the product will often arrive in the U register much sooner than 2N-1 time (for example, A=2 and B=15). The other method would shift the result out as soon as it is available, but would occupy more area due to the large number of and AND and NOR gates. The controller will be implemented over the next two weeks as we prepare the second progress report. A decision will be made as to which method to use for the generation of the shift_res signal after we have a better idea of the layout area available and the size of N. As of now we are leaning towards the AND/NOR gate method for its speed.
Comments – Xinyu Sun

A multiplier can be implemented based on lots of algorithms. In our design we select Russian Peasant’s Algorithm as the design idea for our multiplier. The general idea for this Algorithm is that: we make two columns and put one input number in each, and then halve first number and store as the next entry in first column, double second number and store as the next entry in second column, and redo this procedure, till the first number reduces to 1. In this case, for all odd entries in first column, we add corresponding entries in second column. The sum is the product of the two numbers. In binary number, we can shift the number left to double it, and shift the number right to half it.

In fact the RPA can be implemented by two methods, one is Slow linear-array implementation, and another one is Fast linear-array implementation. The main procedure of this two ways are similar as I talked above, the difference of this two is the way how to store the internal numbers.

For the slow version, we have to use 2n cells. And there are 3 registers which are A_REG, B_REG, PRODUCT_REG, in each cell. And If we assign A to shift right, and B shift right every clock cycle. Then we can get the product due to the following equations:

\[
\begin{align*}
\text{if } a(j) \text{ is odd, } p(j+1) &= p(j) + b(j); \\
\text{if } a(j) \text{ is even, } p(j+1) &= p(j);
\end{align*}
\]

In here, the P_reg is used to store sums, and get the final answer.

Note that each right/left shift takes a unit time. Thus, each step takes Θ(n) worst case time, since the implementation uses a ripple carry adder with depth Θ(n) i.e. the duration of clock period must be at least Θ(n). Now, there are 2n cells here, and so the entire complexity of this implementation becomes Θ(n^2).

For the fast version, we still use 2n cells. However each cell has 4 registers: one each for A and B, and one each for U and V. in addition, This algorithm eliminates the need of a ripple carry approach by implementing carry-save addition. each step takes a unit time here, the product is from the following equations:

\[
\begin{align*}
\text{If } a(j) \text{ is odd, } \\
&ui(j+1) = \text{parity}(bi(j), ui(j), vi(j)) \text{; for } i = 0,1…2n-1 \\
&vi(j+1) = \text{majority}(bi-1(j), ui-1(j), vi-1(j)) \text{; for } 1 \leq i \leq 2n-1 \\
&= 0 \text{; for } i = 0
\end{align*}
\]

\[
\begin{align*}
\text{If } a(j) \text{ is even, } \\
&ui(j+1) = \text{parity}(0, ui(j), vi(j)) \text{; for } i = 0,1…2n-1 \\
&vi(j+1) = \text{majority}(0, ui-1(j), vi-1(j)) \text{; for } 1 \leq i \leq 2n-1 \\
&= 0 \text{; for } i = 0
\end{align*}
\]

In here, ‘a’ shifts right by one bit and ‘b’ shifts left. This implementation requires 2n-1 steps. Once all of ‘a’ has been shifted right, we are left with only to add ‘u’ and ‘v’ with carrysave addition. Thus at the end of 2n-1 steps, U holds the product value. Thus it is seen that each step
takes $\Theta(1)$ time and there are $2n-1$ steps in total. Hence, the complexity of this implementation is $\Theta(n)$ overall.

Because of the small complexity of the fast linear-array implementation, we choose it to implement our multiplier design.

**Behavioral VHDL model without Test Mode**

**VHDL module:**

```vhdl
-- file: RPA_BEHAV.vhd
-- Author: Xinyu Sun (University of Cincinnati)
-- Created: 11/12/2010
-- Updates: None
-- Note: in the code, the signal 'out_en' is same as the signal 'busy' in the bit slice design.
--******************************************************************************
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
entity rpa_behav is
generic (    width : integer :=3);
port (    A, B, clk, rdwt, reset : in std_logic;
output, out_en : out std_logic);
end rpa_behav;
architecture multiplier_behav of rpa_behav is
signal a_reg, b_reg: std_logic_vector ((2*width-1) downto 0); --store the input signal
signal p_reg : std_logic_vector ((2*width-1) downto 0); --store the output signal
signal a_int, b_int: std_logic_vector((2*width-1) downto 0);
signal pdt_int: std_logic_vector ((2*width-1) downto 0);
signal count1 : integer :=(2*width); --counter
signal count2 : integer :=(width-1); --counter
signal count3 : integer :=(width-1); --counter
signal count_en:std_logic;
begin
-- multiplier_behav
multiplier_process: process (clk)
begin
if (reset='1') then
for j in 0 to (2*width-1) loop---reset the value of product
    pdt_int(j)<'0';
end loop;
for i in 0 to (2*width-1) loop -- use for loop reset input
    a_reg(i) <= '0';
    b_reg(i) <= '0';
end loop;
end if;
end process;
end rpa_behav;
```

1-12
end if;
if (clk'EVENT and clk='1') then
  if (reset='0' and rdwt='0') then
    out_en <= '0';
    if (count2>=0) then
      count_en<='0';
      a_reg (width-1) <= A;
      b_reg (width-1) <= B;
      count2 <= count2-1;
    else
      count_en<='1';
    end if;
    a_int <= a_reg; --multiplication
    b_int <= b_reg;
    if (count_en='1') then
      if (count3>=0) then
        a_int((2*width-1) downto 0) <= a_int((2*width-1) downto 1);
        b_int(0) <= '0';
        b_int((2*width-1) downto 1) <= b_int((2*width-2) downto 0);
        if (a_int(0)='1') then
          pdt_int<=pdt_int+b_int;
        else
          pdt_int<=pdt_int;
        end if;
        count3<=count3-1;
      end if;
      p_reg <= std_logic_vector (pdt_int);
    end if;
    if (reset='0' and rdwt='1') then --send the output signal
      if (count1>=0) then
        out_en <= '1';
        output <= p_reg(0);
        p_reg ((2*width-2) downto 0) <= p_reg((2*width-1) downto 1);
        count1 <= count1-1;
      else
        out_en <= '0';
      end if;
    end if;
  end if;
end process multiplier_process;
end multiplier_behav;

VHDL Test bench:
The test bench is as follows.
--file: RPA_BEHAV_testbench.vhd
library ieee;
use ieee.std_logic_1164.all;
entity tb_rpt_behave is
end tb_rpt_behave;
architecture tb_behav of tb_rpt_behave is
component rpa_behav
    generic ( width: integer);
    port ( A, B, clk, rdwt, reset : in std_logic;
          output, out_en : out std_logic);
end component;
constant bit_width : integer := 3;
signal t_A, t_B, t_clk, t_rdwt, t_reset, t_output, t_out_en : std_logic;
begin
U1 : rpa_behav
    generic map (bit_width)
    port map ( t_A, t_B, t_clk, t_rdwt, t_reset,
               t_output, t_out_en);
process
    begin t_clk <= '1';
    wait for 10 ns;
    t_clk <= '0';
    wait for 10 ns;
end process;
test_process: process
    begin t_reset <= '1';
    wait for 20 ns;
    t_reset <= '0';
    t_rdwt <= '0';
    t_A <= '1';
    t_B <= '1';
    wait for 20 ns;
    t_A <= '1';
    t_B <= '1';
    wait for 20 ns;
    t_A <= '1';
    t_B <= '1';
    wait for 120 ns;
    t_A <= '0';
    t_B <= '0';
    t_rdwt <= '1';
    wait for 120 ns;
    wait;
end process;
end tb_behav;
configuration CFG_TB_top of tb_rpt_behave is
    for tb_behav
    end for;
end CFG_TB_top;
Simulation result:

Figure 4: Waveform for the function simulation

In this test bench, A is equal to 7(111), and B is equal to 5 (101). So the result from the output is 35 (100011), when the out_en signal is in logic ‘1’.

**Behavioral VHDL model including Test Mode**

**VHDL model:**

```vhdl
-- library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
entity rpa_test is
  generic (width : integer:=3);
  port (A, B, clk, rdwt, reset,test_en,test_in : in std_logic;
        output, out_en,test_out : out std_logic);
end rpa_test;
architecture multiplier_behav of rpa_test is
  signal a_reg, b_reg: std_logic_vector ((2*width-1) downto 0); --store the input signal
  signal p_reg : std_logic_vector ((2*width-1) downto 0); --store the output signal
  signal a_int, b_int: std_logic_vector((2*width-1) downto 0);
  signal pdt_int: std_logic_vector((2*width-1) downto 0);
  signal count1 : integer :=(2*width); --counter
  signal count2 : integer :=(width-1); --counter
  signal count3 : integer :=(width-1); --counter
```
signal count_en:std_logic;
begin -- multiplier_behav
multiplier_process: process (clk)
begin
  if (reset='1') then
    for j in 0 to (2*width-1) loop
      pdt_int(j)<'0';
    end loop;
    for i in 0 to (2*width-1) loop
      a_reg(i) <= '0';
      b_reg(i) <= '0';
    end loop;
  end if;
  if (clk'EVENT and clk='1') then
    if(test_en='1') then -- test mode
      a_reg((2*width-2) downto 0) <= a_reg ((2*width-1) downto 1);
      b_reg((2*width-2) downto 0) <= b_reg ((2*width-1) downto 1);
      p_reg((2*width-2) downto 0) <= p_reg ((2*width-1) downto 1);
      test_out<= p_reg(0);
    else
      if (reset='0' and rdwt='0') then
        out_en <= '0';
        if (count2>=0) then
          count_en<='0';
          a_reg (width-1) <= A; --read the input
          a_reg ((width-2) downto 0) <= a_reg ((width-1) downto 1);
          b_reg (width-1) <= B;
          b_reg ((width-2) downto 0) <= b_reg ((width-1) downto 1); --read the input
          count2 <= count2-1;
        else
          count_en<='1';
        end if;
        a_int <= a_reg; --multiplication
        b_int <= b_reg;
        if (count_en='1') then
          if (count3>=0) then
            a_int((2*width-1)<'0';
            a_int ((2*width-2) downto 0) <= a_int ((2*width-1) downto 1);
            b_int(0)<'0';
            b_int((2*width-1) downto 1) <= b_int((2*width-2) downto 0);
          end if;
          pdt_int<=pdt_int+b_int;
        else
          pdt_int<=pdt_int;
        end if;
      count3<=count3-1;
      end if;
      p_reg <= std_logic_vector (pdt_int);
    end if;
  if (reset='0' and rdwt='1') then --send the output signal
    if (count1>=0) then

out_en <= '1';
output <= p_reg(0);
p_reg ((2*width-2) downto 0) <= p_reg((2*width-1) downto 1);
count1 <= count1-1;
else
  out_en <= '0';
end if;
end if;
end if;
end if;
end process multiplier_process;
end multiplier_behav;

VHDL Test bench:

library ieee;
use ieee.std_logic_1164.all;
entity tb_rpa_test is
end tb_rpa_test;
architecture tb_behav of tb_rpa_test is
component rpa_test
  generic (width: integer);
  port (A, B, clk, rdwt, reset, test_en, test_in : in std_logic;
        output, out_en, test_out: out std_logic);
end component;
constant bit_width : integer := 3;
signal t_A, t_B, t_clk, t_rdwt, t_reset, t_test_en, t_test_in, t_output,
t_out_en, t_test_out: std_logic;
begin
  U1 : rpa_test
      generic map (bit_width)
      port map (t_A, t_B, t_clk, t_rdwt, t_reset, t_test_en, t_test_in, t_output,
t_out_en, t_test_out);
  process
    begin t_clk <= '1';
    wait for 10 ns;
    t_clk <= '0';
    wait for 10 ns;
  end process;
  test_process: process
    begin t_reset <= '1';
    t_test_en<= '1';
    wait for 20 ns;
    t_reset<= '0';
    t_test_en<= '1';
    wait for 20 ns;
    t_test_en<= '0';
    wait for 20 ns;
  end process;
Simulation result:

From the waveform we can see the value of the test_in is (1001). When the test_en set to ‘1’, the circuit works in test mode, and after 18 clock cycles, the output comes out. We found the value of the output is (1001) which is the same as the input. In addition, the reason why the test_out needs 18 clock cycles to come out is that we have 18 REGs, which is because the width is 3, which means we have 6 REGs for A, B, and Product. In this way, the waveform of test_in signal and test_out signal can check whether all the D-FFs can work well inside the chip. Here, it works well.

**Top-level Block Diagram of the SUNCHILD1049 Architecture**

Figure 6 below shows a top-level view of our architecture. The only test signals shown are SI, SO, SE, and A0. All other test signals, and VDD not shown for clarity.
Figure 6: Top-level diagram of SUNCHILD1049 architecture. Non-scan-chain test inputs and outputs not shown.

Bit-slicing Scheme
The bitslice contains DFFs, a full adder, some simple combinational logic, and routing for \( WR \), shift_res (internal), \( \text{a0} \) (internal), SE, \( \text{RESET} \), and CLK signals. These signals are shown with buffers in Figure 7, but buffers may not be necessary depending on the number of slices we are able to fit inside the pad frame.

When connected together as shown in Figure 6, a chain of 2N bitslices will form 2 \( 2^N \)-bit shift-right registers (A, U), a shift-right/left register (B), a register V and a \( 2^N \)-bit carry-save adder. The dual function right/left-shift register B consists of a MUX and DFF, and is controlled by the global \( WR \) signal. When the \( WR \) signal is externally driven to 0V, B functions as a shift-right register, loading the B factor into place, with the LSB at the end of the bitslice chain. When \( WR \) is removed (returned to 5V), B functions as a shift-left register and the multiplication begins.

The shift_res signal is driven by the controller and initiates the shifting of the product out of the U register. When the shift_res signal is low, U is not a shift register but stores the sum of the carry-save adder. The V register stores the carry of the carry-save adder and does not provide any alternate functions.
Division of Work

Sections of this document were divided according to Table 2 below. Each team member was responsible for the design work involved as well as figures and documentation for all of his sections.

<table>
<thead>
<tr>
<th>Ryan Child</th>
<th>Pin-Out Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Explanation of Chip Function and Test Mode</td>
</tr>
<tr>
<td></td>
<td>Major Design Decisions</td>
</tr>
<tr>
<td></td>
<td>Top-level Block Diagram</td>
</tr>
<tr>
<td></td>
<td>Bit-slicing Scheme</td>
</tr>
<tr>
<td></td>
<td>Division of Work</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Xinyu Sun</th>
<th>Explanation of Chip Function and Test Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Major Design Decisions</td>
</tr>
<tr>
<td></td>
<td>Behavioral VHDL model without Test Mode</td>
</tr>
<tr>
<td></td>
<td>Behavioral VHDL model with Test Mode</td>
</tr>
</tbody>
</table>

Table 2: Division of Work
Chapter 2 - Second Progress Report
2. Revised First Progress Report

Explanation of Chip Function and Test Mode

To load the two factors A and B into the chip and begin multiplication, the user must simultaneously shift A and B into the internal shift registers while driving the WR pin to GND. A and B must be less than or equal to N. Factors less than N will automatically be padded to length N by the controller’s FSM. The timing diagram for multiplying A = 5 (0101) and B = 7 (0111) is shown below.

![Figure 8: Writing AB = {0101}{0111} to a N=4 chip](image)

After N clock cycles, the multiplication is initiated and the BUSY signal goes high. The BUSY signal returns to 0V when the multiplication is complete and can be used to latch the outgoing product. The BUSY signal can also be connected in series with the WR pin of another SUNCHILD1049 to perform multiplications with more than two factors, although it would need to be gated with a counter such that the WR input of the other chip would return to 5V after at most N clock cycles.

After the BUSY signal transitions low to signal completion, the result is shifted out immediately. The result is provided LSB first, as shown in Figure 3 below.

![Figure 9: Product of multiplication P = 5 * 7 = 35 (00011111)](image)

Test Mode simply connects all DFFs in a scan chain. This is accomplished with one MUX for each DFF, controlled by a global scan chain input enable (SE) signal. To verify the correct functioning of all DFFs in the chip, the user may drive SE to VDD, serially input a scan chain of length 4N + 7 into the scan chain input (SI) pin, and read the output at the scan chain output (SO) pin. To test for a stuck-at fault, any scan chain except for a chain of all 0s or all 1s may be chosen, and a fault is identified somewhere along the chain if the chain output is all 0s or all 1s. The faulty DFF cannot be identified using this method alone, however. To find the faulty DFF, one must completely scan in the chain and then return SE to GND, setting the chip into its normal mode of operation. In this
way, the internal state of the chip can be set and the outputs are read using the product (P) pin and extra test pins.

Only 12 pins are needed for the functional, scan, VDD and GND pins. The other 28 pins are allocated to allow access to an isolated bitslice, and the registers of the last few slices in the chain. This will allow us to fully test the slice independently and verify the combinational logic of the last few bits in the chain.

3. Logic Hierarchy

Top-level

![Diagram of Top-level schematic of SUNCHILD1049]

Figure 10: Top-level schematic of SUNCHILD1049
Bitslice

Figure 11: Bitslice schematic
Controller

Figure 12: Updated controller schematic.

4. Layouts & Layout Simulations

Bitslice

Layout

Figure 13: Bitslice Layout

SPICE Simulation

The sentences which should be added into the spice program are as follows.

** hspice subcircuit dictionary
.include model_t36s.sp
**** Constant Voltage Sources ****
VDD vdd Gnd 5.0v
VIN0 CKLI Gnd PULSE (0 5 0.5us 0.01us 0.01us 0.5us 1us)

**Test 0:**
*** test 0:
**** inputs/outputs tested: ain, aout, sei, seo
****
***
<table>
<thead>
<tr>
<th>0T</th>
<th>1T</th>
<th>2T</th>
<th>3T</th>
</tr>
</thead>
<tbody>
<tr>
<td>ain</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sei</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

VIN1 ain Gnd PWL (0 5 1u 5 1.01u 5 2u 5 2.01u 0 3u 0)
VIN2 sei Gnd PWL (0 0 1u 0 1.01u 5 2u 5 2.01u 5 3u 5)
VIN3 sin Gnd PWL (0 0 1u 0 1.01u 0 2u 0 2.01u 0 3u 0)

**Test 1:**
*** inputs/outputs tested: bleft, bright, wri, sei
****
***
<table>
<thead>
<tr>
<th>3T</th>
<th>4T</th>
<th>5T</th>
<th>6T</th>
<th>7T</th>
</tr>
</thead>
<tbody>
<tr>
<td>sei</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>wri</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bleft</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

VIN1 wri Gnd PWL (0 5 1u 5 1.01u 5 2u 5 2.01u 0 3u 0)
VIN2 sei Gnd PWL (0 5 1u 5 1.01u 5 2u 5 2.01u 5 3u 5)
VIN3 bleft Gnd PWL (0 0 1u 0 1.01u 5 2u 5 2.01u 5 3u 5)
VIN4 bright Gnd PWL (0 5 1u 5 1.01u 0 2u 0 2.01u 0 3u 0)

**Test 2:**
*** inputs/outputs tested: bleft, bright, wri, sei
****
***
<table>
<thead>
<tr>
<th>3T</th>
<th>4T</th>
<th>5T</th>
<th>6T</th>
<th>7T</th>
</tr>
</thead>
<tbody>
<tr>
<td>sei</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sri</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>uin</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

VIN1 sri Gnd PWL (0 5 1u 5 1.01u 5 2u 5 2.01u 5 3u 5)
VIN2 sei Gnd PWL (0 5 1u 5 1.01u 5 2u 5 2.01u 5 3u 5)
VIN3 uin Gnd PWL (0 0 1u 0 1.01u 5 2u 5 2.01u 5 3u 5)
VIN4 bright Gnd PWL (0 5 1u 5 1.01u 0 2u 0 2.01u 0 3u 0)

**Test 3:**
*** inputs/outputs tested: bleft, bright, wri, sei
****
***
<table>
<thead>
<tr>
<th>3T</th>
<th>4T</th>
<th>5T</th>
<th>6T</th>
<th>7T</th>
</tr>
</thead>
</table>

VIN1 bleft Gnd PWL (0 5 1u 5 1.01u 5 2u 5 2.01u 0 3u 0)
VIN2 bright Gnd PWL (0 5 1u 5 1.01u 0 2u 0 2.01u 0 3u 0)
VIN3 bleft Gnd PWL (0 0 1u 0 1.01u 5 2u 5 2.01u 5 3u 5)
VIN4 bright Gnd PWL (0 5 1u 5 1.01u 0 2u 0 2.01u 0 3u 0)

---

2-6
Simulation result:
I use each of the block above one time to run the spice to get the following 5 test result. Because in the layout the output signal clko, wro, seo, sro, reseto, a0o are same as the input clki, wri, sei, sri, reseti, a0i, which are a same metal. Thus the output simulation does not include this output signal.

Test0:

Explanation: when the ain is high, and the sei is high which select the ain, at this time the aout is high, when the clk at the rising edge. When the ain goes to low, and the sei still select the ain the aout goes to low too. However, when the sei goes to low, which select the sin, which is ‘0’, so the aout is still ‘0’, after the sei goes to ‘0’

Test1:
Explanation: wri- is used to select b_left or b_right. When the sei is high, it means the chip is in the working function. Then the bout will get the value either from b_right or b_left. From the waveform we can see it works correct.

Test2:

![Graph](image1)

**Figure 16: Bitslice SPICE test 2**

Explanation: when the sri is high and the sei is high, the uin will be selected to uout. The value of the uout will be changed at the rising edge of the clk. From the waveform we can see it works correct.

Test3:

![Graph](image2)

**Figure 17: Bitslice SPICE test 3**
Explanation: The value of \( so \) comes from the value of \( vin \) and \( reset \). When \( vin \) and \( reset \) are both 1 the output to the Reg will be high, then the \( so \) will be high. It means the chip is reset- low trigger. From the waveform we can see it works correct.

Test4:

![Figure 18: Bitslice SPICE test 4](image)

Explanation: here we want to test the function of the full adder. The inputs of the adder are \( uout \) and \( so \), the carry in of the adder is the number of \( bout \) && \( a0i \). the carry out is \( vout \). According to the spice simulation, we get the conclusion, the outputs are correct and the logic of the bit slice layout is correct.

Controller

**Layout**

![Figure 19: Controller Layout](image)

**IRSIM Simulation**

A command file was written to reflect the VHDL testbench and verify the correctness of the controller layout. The command file is not provided here as the clock generation code is extremely redundant and would waste many pages of paper. The IRSIM output window does not include a time axis, but the timestep was set to 10ns for a clock period of 20ns. First, the scan input was tested by asserting \( SE \) and sending a pattern of 0000011 through
the scan chain. The simulation shows that the output at the end of the scan chain, SO, is indeed 0000011 after 7 clock cycles. Secondly, the simulation shows that the counter stops counting and the BUSY (so) signal goes low after 3N clock cycles. We estimated our final N to be about 22, so the controller was laid out with a 20-bit counter. This can easily be modified after we layout the bitslices inside the pad frame.

Figure 20: IRSIM Simulation of controller. Timestep = 10ns

5. VHDL Code & Simulations

Notes on delay time
A structural architecture was created directly from the logic-level design, and delays were found from the datasheets of the cell library used in layout. The datasheets contain tables which show the delays of gates given their load capacitances and input rise times. The load capacitances were determined by adding the pin capacitances apparent on each output, and the input rise times were found by multiplying the drive strength (sec/F) of the gate driving the input by the capacitance on the input. Appropriate delay times were then assigned to each gate instance in the bitslice VHDL model through the use of generic parameters. The resulting delays are shown in Table 3 and Table 4.

<table>
<thead>
<tr>
<th>Component</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>And2x1</td>
<td>0.185ns</td>
</tr>
<tr>
<td>Xor2x1</td>
<td>0.08 ns</td>
</tr>
<tr>
<td>Or2x1(1)</td>
<td>0.167 ns</td>
</tr>
<tr>
<td>Or2x1(2)</td>
<td>0.226</td>
</tr>
<tr>
<td>Or2x1(3)</td>
<td>0.549</td>
</tr>
<tr>
<td>Dff(1,3,5)</td>
<td>0.34 ns</td>
</tr>
<tr>
<td>Dff(2,4)</td>
<td>0.338ns</td>
</tr>
<tr>
<td>Dff(6)</td>
<td>0.415ns</td>
</tr>
<tr>
<td>Dff(7)</td>
<td>0.45ns</td>
</tr>
<tr>
<td>Mux(1,2,3,4,5,6,7,)</td>
<td>0.0832 ns</td>
</tr>
<tr>
<td>gate #</td>
<td>gate type</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
</tr>
<tr>
<td>1</td>
<td>mux</td>
</tr>
<tr>
<td>2</td>
<td>inv</td>
</tr>
<tr>
<td>3</td>
<td>dff</td>
</tr>
<tr>
<td>4</td>
<td>fa (ys)</td>
</tr>
<tr>
<td></td>
<td>fa (yc)</td>
</tr>
<tr>
<td>5</td>
<td>mux</td>
</tr>
<tr>
<td>6</td>
<td>inv</td>
</tr>
<tr>
<td>7</td>
<td>mux</td>
</tr>
<tr>
<td>8</td>
<td>inv</td>
</tr>
<tr>
<td>9</td>
<td>dff</td>
</tr>
<tr>
<td>10</td>
<td>and</td>
</tr>
<tr>
<td>11</td>
<td>mux</td>
</tr>
<tr>
<td>12</td>
<td>inv</td>
</tr>
<tr>
<td>13</td>
<td>and</td>
</tr>
<tr>
<td>14</td>
<td>mux</td>
</tr>
<tr>
<td>15</td>
<td>inv</td>
</tr>
<tr>
<td>16</td>
<td>dff</td>
</tr>
<tr>
<td>17</td>
<td>and</td>
</tr>
<tr>
<td>18</td>
<td>mux</td>
</tr>
<tr>
<td>19</td>
<td>inv</td>
</tr>
<tr>
<td>20</td>
<td>dff</td>
</tr>
</tbody>
</table>

Table 3: Controller component delays

<table>
<thead>
<tr>
<th>Code: Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
</tr>
</tbody>
</table>

```vhdl
--************************************************************
-- file: rpa_chip.vhd
-- Author: Xinyu Sun (University of Cincinnati)
-- Created: 11/26/2010
-- Updates: None
-- Note:
--************************************************************
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```
ENTITY rpa_chip is
    generic ( width : integer:=23 );
    port ( A,B, WR, SI, SE, CLK, RESET : in std_logic ;
           P,SO,BUSY : out std_logic );
end rpa_chip;

architecture structure of rpa_chip is
    component bitslice
        port (ain, bleft, bright, vin, uin, wri, a0i, sri, sei, si, clki, reseti: in std_logic;
              aout, boutr,boutl, vout, uout, wro, a0o, sro, seo, so, clko, reseto: out std_logic);
    end component;

    component and2x1
        port(a, b: in std_logic;
             y: out std_logic);
    end component;

    component controller
        port (wr, sei, si, clki, reset: in std_logic;
              so, sbr,pab: out std_logic);
    end component;

    signal s1, s2, s3, s4, s5, s6 ,s7,s8,s9,s10,s11,s12, w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12:
        std_logic_vector((width+1) downto 0);
    signal s_so,s_sbr,s_wr: std_logic;

begin
    s_wr<=not WR;
    chip0:controller port map ( wr=>WR, sei=>SE, si=>SI, clki=>CLK,
                                reset=>RESET,sbr=>s_sbr, so=>s_so);
    and1:and2x1 port map( a=>B, b=>s_wr, y=>s2(width+1));
    and2:and2x1 port map( a=>s_wr, b=>A, y=>s1(width+1));

    chip_gen0 : for i in 1 to (width) generate
        begin
            C0: bitslice port map ( ain => s1(i+1),
                                    aout => s2(i+1),
                                    bleft => s2(i+1),
                                    boutr => s2(i),
                                    bright => s3(i-1),
                                    boutl => s3(i),
                                    vin => s4(i-1),
                                    vout => s4(i),
                                    uin => s5(i+1),
                                    uout => s5(i),
                                    --SBR
                                    wri => s6(i+1),
                                    wro => s6(i),
                                    a0i =>s7(i-1),
                                    a0o =>s7(i),
                                    sri =>s8(i+1),
                                )
        end
    end
\begin{verbatim}
sro  => s8(i),
sei  => s9(i+1),
seo  => s9(i),
si   => s10(i+1),
so   => s10(i),
ciki => s11(i+1),
ciko => s11(i),
reseti => s12(i+1),
reseto => s12(i));
end generate chip_gen0;
chip_gen1 : for j in 1 to (width) generate
begin
    C1: bitslice port map (
        ain       => w1(j+1),
        aout      => w1(j),
        bleft     => w2(j+1),
        boutr     => w2(j),
        bright    => w3(j-1),
        boutl     => w3(j),
        vin       => w4(j-1),
        vout      => w4(j),
        uin       => w5(j+1),
        uout      => w5(j),
        --SBR
        wri       => w6(j+1),
        wro       => w6(j),
        a0i       => w7(j-1),
        a0o       => w7(j),
        sri       => w8(j+1),
        sro       => w8(j),
        sei       => w9(j+1),
        seo       => w9(j),
        si        => w10(j+1),
        so        => w10(j),
        clki      => w11(j+1),
        clko      => w11(j),
        reseti    => w12(j+1),
        reseto    => w12(j));
end generate chip_gen1;
s7(0)<= s1(1);
P<= s5(1);
S0<= s10(1);
BUSY<= s_so;
\end{verbatim}
```
s11(width+1)<=w11(1); --clk
s6(width+1)<=w6(1); --sbr
s9(width+1)<=w9(1); --se
s12(width+1)<=w12(1);--reset
s8(width+1)<=w8(1);--sro-sri
--connect to inputs
w11(width+1)<= CLK;
w8(width+1)<=s_so;
w10(width+1)<=s_so;
w6(width+1)<=s_sbr;
w9(width+1)<=SE;
w12(width+1)<=RESET;
--initial 0:
w2(width+1)=='0';
w5(width+1)=='0';
w1(width+1)=='0';
--for slice0
s4(0)=='0';
s3(0)=='0';
end structure;

Testbench

--*****************************************************************
-- file: test_bench_rpa_chip.vhd
-- Author: Xinyu Sun (University of Cincinnati)
-- Created: 11/26/2010
-- Updates: None
-- Note:
--*****************************************************************
library ieee;
use ieee.std_logic_1164.all;
entity tb_rpa_chip is
end tb_rpa_chip;
architecture tb_arch of tb_rpa_chip is
  component rpa_chip
    generic ( width : integer );
    port ( A,B, WR, SI, SE, CLK, RESET : in std_logic ;
P,SO,BUSY : out std_logic );
  end component;
signal t_A, t_B, t_WR, t_SI,t_SE, t_CLK, t_RESET, t_P, t_SO,t_BUSY :
  std_logic;
constant bit_width : integer := 23;
begin
  U1 : rpa_chip
    generic map ( bit_width )
    port map ( t_A, t_B, t_WR, t_SI,t_SE, t_CLK, t_RESET, t_P, t_SO,t_BUSY );

  process
  begin
    t_CLK <= '0';
  end process;
end tb_arch;
```
wait for 10 ns;
t_CLK <= '1';
wait for 10 ns;
end process;

test_process : process
begin
  t_WR <= '0';
  -- test mode
  -- t_SE <= '1';
  t_SE<='0';-- WORKING MODE (when u wanna use test mode please delet this)
  t_RESET<= '0';-- reset signal

  wait for 20 ns;
  t_RESET<='1';
  t_WR <='0';
  t_A <= '0';
  t_B <= '1';
  -- scan in
  -- t_SI<='1';
  wait for 20 ns;
  -- t_SI<='0';
  t_A <= '1';
  t_B <= '0';
  wait for 20 ns;
  -- t_SI<='1';
  t_A <= '1';
  t_B <= '1';
  wait for 20 ns;
  -- t_SI<='0';
  t_A <= '0';
  t_B <= '1';
  wait for 20 ns;
  t_WR <= '1';
  wait for 4000 ns;
  wait;
end process;
end tb_arch;

configuration CFG_TB_top of tb_rpa_chip is
for tb_arch
end for;
end CFG_TB_top;

---****************************

--- file: bitslice.vhd
--- Author: Ryan Child (University of Cincinnati)
--- Created: 11/28/2010
--- Updates: None
library ieee;
use ieee.std_logic_1164.all;

entity bitslice is
  port (ain, bleft, bright, vin, uin, sbri, a0i, sri, sei, si, clki, reseti: in std_logic;
        aout, bout, vout, uout, sbro, a0o, sro, seo, so, clko, reseto: out std_logic);
end bitslice;

-- ::input definitions::
--
-- ain: input to A flip-flop
-- bleft: input to B flip-flop; connects to bout of slice to the left
-- bright: input to B flip-flop; connects to bout of slice to the right
-- vin: input to V flip-flop
-- uin: input to V flip-flop
-- sbri: shift b right
-- a0i: connects to aout of first slice (LSB)
-- sri: shift result
-- sei: scan enable
-- si: scan in
-- clki: clock
-- reseti: reset (active low)
--
-- ::output definitions::
--
-- aout: output of A flip-flop
-- bout: output of B flip-flop
-- vout: carry output of full adder
-- uout: output of U flip-flop
-- sbro: pass-through shift b right to next slice
-- a0o: pass-through aout of first slice to next slice
-- sro: pass-through shift result to next slice
-- seo: pass-through scan enable to next slice
-- so: scan out; also output of V flip-flop
-- clko: pass-through clock signal to next slice
-- reseto: pass-through reset signal to next slice

-- define structural architecture
architecture structure of bitslice is

component dffpos1
  generic(delay: time);
  port(d, clk: in std_logic;
       q: out std_logic);
end component;

component mux2x1
  generic(delay: time);
  port(a, b, s: in std_logic;
       y: out std_logic);
end component;

component and2x1 is
generic(delay: time);
    port(a, b: in std_logic;
        y: out std_logic
    );
end component;

cOMPONENT invx1 IS
    GENERIC (delay: time);
    PORT (a: in std_logic;
        y: out std_logic)
END COMPONENT;

COMPONENT fax1 IS
    GENERIC (delays, delayc: time);
    PORT (a, b, c: in std_logic;
        yC, yS: out std_logic)
END COMPONENT;

-- general purpose wires
signal w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14, w15, w16: std_logic;

-- needed to access outputs
signal waout, wbout, wuout, wso: std_logic;

BEGIN

-- pass-through signals
    sez <= sei;
    a06 <= a0i;
    s06 <= sri;
    clko <= clki;
    reseto <= reseti;
    sb06 <= sbri;

-- connect wires to outputs
    aout <= waout;
    bout <= wbout;
    uout <= wuout;
    so <= wso;

-- DFFs
    g03: dffposx1 generic map(delay => 0.338 ns) port map(d => w1, clk => clki, q => waout);
    g09: dffposx1 generic map(delay => 0.397 ns) port map(d => w3, clk => clki, q => wbout);
    g16: dffposx1 generic map(delay => 0.338 ns) port map(d => w6, clk => clki, q => wuout);
    g20: dffposx1 generic map(delay => 0.397 ns) port map(d => w8, clk => clki, q => wso);

-- MUXs
    g01: mux2x1 generic map(delay => 0.0832 ns) port map(a => s, b => a08, s => sei, y => w11);
    g05: mux2x1 generic map(delay => 0.0832 ns) port map(a => bleft, b => bright, s => sbri, y => w12);
    g07: mux2x1 generic map(delay => 0.0832 ns) port map(a => waout, b => w2, s => sei, y => w13);
library ieee;
use ieee.std_logic_1164.all;

entity tb_bitslice is
end tb_bitslice;

architecture arch of tb_bitslice is
component bitslice

 Testbench
The testbench was written to test all inputs. Test patterns and expected outputs for each test are shown as comments at the top of the testbench file.

--************************************************************
-- file: tb_bitslice.vhd
-- Author: Ryan Child (University of Cincinnati)
-- Created: 11/28/2010
-- Updates: None
--************************************************************
port (ain, bleft, bright, vin, uin, sbri, a0i, sri, sei, si, clki, reseti: in std_logic;
    aout, bout, vout, uout, sbro, a0o, sro, seo, so, clko, reseto: out std_logic);
end component;

-- ::input definitions::
--
-- ain:     input to A flip-flop
-- bleft:   input to B flip-flop; connects to bout of slice to the left
-- bright:  input to B flip-flop; connects to bout of slice to the right
-- vin:     input to V flip-flop
-- uin:     input to V flip-flop
-- sbri:    shift b right
-- a0i:     connects to aout of first slice (LSB)
-- sri:     shift result
-- sei:     scan enable
-- si:      scan in
-- clki:    clock
-- reseti:  reset (active low)
--
-- ::output definitions::
--
-- aout:    output of A flip-flop
-- bout:    output of B flip-flop
-- vout:    carry output of full adder
-- uout:    output of U flip-flop
-- sbro:    pass through shift b right to next slice
-- a0o:     pass through aout of first slice to next slice
-- sro:     pass through shift result to next slice
-- seo:     pass through scan enable to next slice
-- so:      scan out; also output of V flip-flop
-- clko:    pass through clock signal to next slice
-- reseto:  pass through reset signal to next slice

signal ain, bleft, bright, vin, uin, sbri, a0i, sri, sei, si, clki, reseti,
    aout, bout, vout, uout, sbro, a0o, sro, seo, so, clko, reseto: std_logic;

-- ::clock speed::
--
-- T = 20ns -> 50MHz clock
constant period: time:= 20 ns;

-- ::test definitions::
--
-- test 0:
--    inputs/outputs tested: ain, aout, sei, seo
--
--          | 0T | 1T | 2T | 3T
-- ain     | 0   | 1  | 1  |
-- sei     | 0   | 0  | 1  |
-- si      | 0   | 0  | 0  |
-- expected aout | X  | 0  | 1  | 0
-- expected seo | 0  | 0  | 1  | X
-- test 1:
-- inputs/outputs tested: bleft, bright, wri, sei
--
--
<table>
<thead>
<tr>
<th></th>
<th>3T</th>
<th>4T</th>
<th>5T</th>
<th>6T</th>
<th>7T</th>
<th>8T</th>
</tr>
</thead>
<tbody>
<tr>
<td>sei</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sbri</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>bleft</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>bright</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>expected bout</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>expected seo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>expected sbro</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>
--

-- test 2:
-- inputs/outputs tested: uin, uout, sei, seo
--
--
<table>
<thead>
<tr>
<th></th>
<th>8T</th>
<th>9T</th>
<th>10T</th>
<th>11T</th>
</tr>
</thead>
<tbody>
<tr>
<td>uin</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sei</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sri</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>expected uout</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>expected seo</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>
--

-- test 3:
-- inputs/outputs tested: vin, sei, seo, so
--
--
<table>
<thead>
<tr>
<th></th>
<th>11T</th>
<th>12T</th>
<th>13T</th>
<th>14T</th>
</tr>
</thead>
<tbody>
<tr>
<td>vin</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sei</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>expected so</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>expected seo</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>
--

-- test 4:
-- inputs/outputs tested: sri, a0i, wri, uin, vin, bright,
-- sro, a0o, wro, bout, uout, so, vout
--
--
<table>
<thead>
<tr>
<th></th>
<th>14T</th>
<th>15T</th>
<th>16T</th>
<th>17T</th>
<th>18T</th>
<th>19T</th>
<th>20T</th>
</tr>
</thead>
<tbody>
<tr>
<td>sri</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a0i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sbri</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>uin</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>vin</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>bright</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>expected sro</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>expected a0o</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>expected sbro</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>expected bout</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>expected uout</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>expected so</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>expected vout</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
--

-- test 5:
-- inputs/outputs tested: reset
--
--
<table>
<thead>
<tr>
<th></th>
<th>20T</th>
<th>21T</th>
<th>22T</th>
</tr>
</thead>
<tbody>
<tr>
<td>reseti</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>expected reseto</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>expected uout</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>expected so</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
begin

  clock:
  process
  begin
    clki <= '1';
    wait for period/2;
    clki <= '0';
    wait for period/2;
  end process;

  uut: bitslice
  port map(ain, bleft, bright, vin, uin, sbri, a0i, sri, sei, si, clki,
  reseti,
            aout, bout, vout, uout, sbro, a0o, sro, seo, so, clko,
  reseto);

  testbench:
  process(clki)
  variable runtime: time:= 0 ns;
  begin
    if (clki'event and clki='1') then
      if (runtime = 0*period) then
        -- test 0 begin
        ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
        sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
        '1';
      elsif (runtime = 1*period) then
        ain<='1'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
        sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
        '1';
      elsif (runtime = 2*period) then
        ain<='1'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
        sbri<='1'; a0i<='0'; sri<='0'; sei<='1'; si<='0'; reseti <=
        '1';
      elsif (runtime = 3*period) then
        -- test 1 begin
        ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
        sbri<='0'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
        '1';
      elsif (runtime = 4*period) then
        ain<='0'; bleft<='1'; bright<='1'; vin<='0'; uin<='0';
        sbri<='0'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
        '1';
      elsif (runtime = 5*period) then
        ain<='0'; bleft<='1'; bright<='0'; vin<='0'; uin<='0';
        sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
        '1';
      elsif (runtime = 6*period) then
        ain<='0'; bleft<='1'; bright<='0'; vin<='0'; uin<='0';
        sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
        '1';
      elsif (runtime = 7*period) then
        ain<='0'; bleft<='1'; bright<='0'; vin<='0'; uin<='0';
        sbri<='1'; a0i<='0'; sri<='0'; sei<='1'; si<='0'; reseti <=
        '1';
    else
      begin
        ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
        sbri<='0'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
        '1';
elsif (runtime = 8*period) then
  -- test 2 begin
  ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
  sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 9*period) then
  ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='1';
  sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 10*period) then
  ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='1';
  sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 11*period) then
  -- test 3 begin
  ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
  sbri<='0'; a0i<='0'; sri<='1'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 12*period) then
  ain<='0'; bleft<='0'; bright<='0'; vin<='1'; uin<='0';
  sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 13*period) then
  ain<='0'; bleft<='0'; bright<='0'; vin<='1'; uin<='0';
  sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 14*period) then
  -- test 4 begin
  ain<='0'; bleft<='0'; bright<='0'; vin<='1'; uin<='1';
  sbri<='0'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 15*period) then
  ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
  sbri<='0'; a0i<='0'; sri<='1'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 16*period) then
  ain<='0'; bleft<='0'; bright<='0'; vin<='1'; uin<='0';
  sbri<='0'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 17*period) then
  ain<='0'; bleft<='0'; bright<='0'; vin<='1'; uin<='0';
  sbri<='0'; a0i<='0'; sri<='1'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 18*period) then
  ain<='0'; bleft<='0'; bright<='0'; vin<='1'; uin<='0';
  sbri<='0'; a0i<='0'; sri<='1'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 19*period) then
  ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
  sbri<='0'; a0i<='1'; sri<='0'; sei<='0'; si<='0'; reseti <=
  '1';
elsif (runtime = 20*period) then
  -- test 5 begin
  ain<='0'; bleft<='0'; bright<='0'; vin<='0'; uin<='0';
  sbri<='1'; a0i<='0'; sri<='0'; sei<='0'; si<='0'; reseti <=
  '0';
elsif (runtime = 21*period) then
ain='0'; bleft='0'; bright='0'; vin='0'; uin='0';
  sbri='1'; a0i='0'; sri='0'; sei='0'; si='0'; reseti <= '0';
end if;
runtime:= runtime + period;
end if;
end process;
end arch;

Code: Controller

Model

library ieee;
use ieee.std_logic_1164.all;

entity controller is
  port (wr, sei, si, clki, reset: in std_logic;
        so, sbr,pab: out std_logic);
end controller;

-- define structural architecture
architecture structure of controller is
  component dffposx1
    generic(delay: time);
    port(d, clk: in std_logic;
         q: out std_logic);
  end component;
  component mux2x1
    generic(delay: time);
    port(a, b, s: in std_logic;
         y: out std_logic);
  end component;
  component and2x1
    generic(delay: time);
    port(a, b: in std_logic;
         y: out std_logic);
  end component;
  component nand2x1
    generic(delay: time);
    port(a, b: in std_logic;
         y: out std_logic);
); end component;

component invx1 is
  generic(delay: time);
  port(a: in std_logic;
       y: out std_logic);
end component;

component xor2X1 is
  generic(delay: time);
  port(a, b: in std_logic;
       y: out std_logic);
end component;

component or2X1 is
  generic(delay: time);
  port(a, b: in std_logic;
       y: out std_logic);
end component;

component nor2X1 is
  generic(delay: time);
  port(a, b: in std_logic;
       y: out std_logic);
end component;

component nor3X1 is
  generic(delay: time);
  port(a, b, c: in std_logic;
       y: out std_logic);
end component;

component nand3X1 is
  generic(delay: time);
  port(a, b, c: in std_logic;
       y: out std_logic);
end component;

-- general purpose wires
signal scnt, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13,
s14, s15, s16,s17, s18, s19, s20, s21, s22, s23, s24, s25, s26, s27,
s28, s29, s30, s31, s32, s33, s34, s35, s36, s37, s38, s39, s40,s41: std_logic;

begin
  s1<=wr;
  s2<= reset;
  -- connect wires to outputs
  pab <= wr;
  so <= s7;
  sbr<= not s7;
  -- DFFs
  dff1: dffposx1 generic map(delay => 0.34 ns)port map(d => s19, clk => clki, q => s20);
  dff2: dffposx1 generic map(delay => 0.338 ns)port map(d => s22,
clk => clki, q => s23);
  dff3: dffposx1 generic map(delay => 0.34 ns)port map(d => s27, clk => clki, q => s28);
  dff4: dffposx1 generic map(delay => 0.338 ns)port map(d => s32, clk => clki, q => s33);
  dff5: dffposx1 generic map(delay => 0.34 ns)port map(d => s37, clk => clki, q => s16);
  dff6: dffposx1 generic map(delay => 0.415 ns)port map(d => s32, clk => clki, q => s8);
  dff7: dffposx1 generic map(delay => 0.45 ns)port map(d => s12, clk => clki, q => s7);

  -- MUXs
  mux1: mux2x1 generic map(delay => 0.0832 ns)port map(a => s6, b => s18, s => sei, y => s19);
  mux2: mux2x1 generic map(delay => 0.0832 ns)port map(a => s20, b => s21, s => sei, y => s22);
  mux3: mux2x1 generic map(delay => 0.0832 ns)port map(a => s23, b => s26, s => sei, y => s27);
  mux4: mux2x1 generic map(delay => 0.0832 ns)port map(a => s28, b => s31, s => sei, y => s32);
  mux5: mux2x1 generic map(delay => 0.0832 ns)port map(a => s33, b => s36, s => sei, y => s37);
  mux6: mux2x1 generic map(delay => 0.0832 ns)port map(a => s16, b => s15, s => sei, y => s17);
  mux7: mux2x1 generic map(delay => 0.0832 ns)port map(a => s8, b => s11, s => sei, y => s12);

  -- INVs
  inv1: invx1 generic map(delay => 0.3354 ns)port map(a => s4, y => s41);
  inv2: invx1 generic map(delay => 0.0909 ns)port map(a => si, y => s6);

  -- Nands
  nand1: nand3x1 generic map(delay => 0.15 ns)port map(a => s41, b => s2, c => s2, y => s18);
  nand2: nand3x1 generic map(delay => 0.15 ns)port map(a => s41, b => s38, c => s2, y => s21);
  nand3: nand3x1 generic map(delay => 0.15 ns)port map(a => s41, b => s25, c => s2, y => s26);
  nand4: nand3x1 generic map(delay => 0.15 ns)port map(a => s41, b => s30, c => s2, y => s31);
  nand5: nand3x1 generic map(delay => 0.15 ns)port map(a => s41, b => s35, c => s2, y => s36);
  nand6: nand2x1 generic map(delay => 0.104 ns)port map(a => s2, b => s14, y => s15);
  nand7: nand2x1 generic map(delay => 0.104 ns)port map(a => s2, b => s10, y => s11);
  nand8: nand3x1 generic map(delay => 0.15 ns)port map(a => s20, b => s28, c => s16, y => s39);

  -- NORs
  nor1: nor2x1 generic map(delay => 0.15 ns)port map(a => s39, b => s40, y => s4);
  nor2: nor3x1 generic map(delay => 0.15 ns)port map(a => s8, b => s7, c => s1, y => s5);

  -- ORs
or1: or2x1 generic map(delay => 0.167 ns) port map (a=>s23, b=>s33, y=>s40);
or2: or2x1 generic map(delay => 0.226 ns) port map (a=>s4, b=>s5, y=>s13);
or3: or2x1 generic map(delay => 0.549 ns) port map (a=>s7, b=>s8, y=>s1);

-- XORs
xor1: xor2x1 generic map(delay => 0.08 ns) port map (a=>scnt, b=>s20, y=>s3);
xor2: xor2x1 generic map(delay => 0.08 ns) port map (a=>s20, b=>s23, y=>s38);
xor3: xor2x1 generic map(delay => 0.08 ns) port map (a=>s24, b=>s28, y=>s25);
xor4: xor2x1 generic map(delay => 0.08 ns) port map (a=>s33, b=>s29, y=>s30);
xor5: xor2x1 generic map(delay => 0.08 ns) port map (a=>s34, b=>s16, y=>s35);
xor6: xor2x1 generic map(delay => 0.08 ns) port map (a=>s13, b=>s8, y=>s14);
xor7: xor2x1 generic map(delay => 0.08 ns) port map (a=>s9, b=>s7, y=>s10);

-- ANDs
and1: and2x1 generic map(delay => 0.185 ns) port map (a=>s23, b=>s20, y=>s24);
and2: and2x1 generic map(delay => 0.185 ns) port map (a=>s24, b=>s28, y=>s29);
and3: and2x1 generic map(delay => 0.185 ns) port map (a=>s29, b=>s33, y=>s34);
and4: and2x1 generic map(delay => 0.185 ns) port map (a=>s8, b=>s4, y=>s9);

end structure;

Testbench

--****************************************************************************************
-- file: RPA_controller_testbench.vhd
-- Author: Xinyu Sun (University of Cincinnati)
-- Created: 11/26/2010
-- Updates: None
-- Note:
--****************************************************************************************
library ieee;
use ieee.std_logic_1164.all;

entity tb_controller is
end tb_controller;

architecture arch of tb_controller is
component controller
  port (wr, sei, si, clki, reset: in std_logic;
       so, sbr, pab: out std_logic);
end component;
Code: Leaf-level (gates)
Both team members contributed to the behavioral VHDL modules of cells in the library.

**DFFPOSX1**

```vhdl
-- dffposx1.vhd
library ieee;
use ieee.std_logic_1164.all;
entity dffposx1 is
generic(delay: time:=0 ps);
```
DFFPOSX

-- dffposx.vhd
library std;
use std_logic_1164.all;

entity dffposx is
  port(d,clk: in std_logic;
       q: out std_logic);
end dffposx;
architecture behv of dffposx is
begin
  process(clk) begin
    if(clk'event and clk='1') then
      q <= d after delay;
    end if;
  end process;
end behv;

MUX2X1

-- mux2x1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity mux2x1 is
  generic(delay: time:=0 ps);
  port(a, b, s :in std_logic;
       y :out std_logic);
end mux2x1;
architecture behv of mux2x1 is
begin
  process (a, b, s)
  begin
    if s = '0' then
      y <= not b after delay;
    elsif s = '1' then
      y <= not a after delay;
    end if;
  end process;
end behv;

NAND2X1

-- nand2x1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity nand2x1 is
  generic(delay: time);
  port(a,b : in std_logic;
       y: out std_logic);
end entity nand2x1;
architecture behav of nand2x1 is
begin
  process (a, b)
begin
    y <= a nand b after delay;
end process;
end behav;

**NAND3X1**

begin
    y <= not (a and b and c) after delay;
end process;
end behav;

**NOR2X1**

begin
    y <= a nor b after delay;
end behav;

**NOR3X1**

begin
    y <= a nor b after delay;
end behav;
end entity nor3x1;
architecture behav of nor3x1 is
begin
    process (A, B, C)
    begin
        Y <= not (A or B or C) after delay;
    end process;
end behav;

---

**OR2X1**

```vhdl
-- or2x1.vhd
library ieee;
use ieee.std_logic_1164.all;
entity or2x1 is
    generic(delay: time:=0 ps);
    port(a, b : in std_logic;
         y: out std_logic);
end entity or2x1;
architecture behav of or2x1 is
begin
    process (a, b)
    begin
        y<= a or b after delay;
    end process;
end behav;
```

---

**XOR2X1**

```vhdl
-- xor2x1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity xor2x1 is
    generic(delay: time:=0 ps);
    port(a, b : in std_logic;
         y: out std_logic);
end entity xor2x1;
architecture behav of xor2x1 is
begin
    process (a, b)
    begin
        y <= a xor b after delay;
    end process;
end behav;
```
INVX1

-- invx1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity invx1 is
  generic(delay: time:=0 ps);
  port(a :in std_logic;
       y :out std_logic);
end invx1;

architecture behv of invx1 is
begin
  y <= not a after delay;
end behv;

FAX1

-- fax1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity fax1 is
  generic(delays, delayc: time:=0 ps);
  port(a, b, c:in std_logic;
       yc, ys :out std_logic);
end fax1;

architecture behv of fax1 is
begin
  ys <= a xor b xor c after delays;
  yc <= (a and b) or (b and c) or (a and c) after delayc;
end behv;

Comparison with VHDL of First Progress Report
The results of both VHDL programs are correct. Because the VHDL program in the first report is just on the behavioral level, some control signals are different, such as the rdwr signal in behavioral level which is not quite the same as wr in the structure level. Besides, in the first report because it is the behavioral level, there are some delays in signal passing inside. But it cannot happen in the structure level. In addition, the names of some signals are changed in the structure level. Such as we use ‘BUSY’ instead of ‘out_en’, use ‘SO’ to replace ‘test_out’, use ‘P’ to replace ‘output’.

Simulation: Chip
the simulation result of the test mode:
Explanation: the clock period is 20 ns. From the wave form we can see the when the chip is working in the test mode the SE signal is set to 1. And the scan in is 1010, the scan out (SO) goes out after 191 clock cycle. That’s because the chip is 23 bits multiplier, which need 46 bit slices. And each bits lice has 4 regs, thus we have 4*46=184 regs so far. In addition, the chip has one controller, which has 7 Regs. Thus totally is 191 regs. So we can get the conclusion the test mode works correct.

The simulation result of multiplication

Explanation:
The input for the A is 110, and for the B is 1101. They are input serially with the LSB first. When the chip is loading number the WR is set to ‘0’. When the load is done the WR goes back to ‘1’. From the wave form we can see, the BUSY is set to 0 at the first 23 clock cycle, that’s because the chip is loading the 23 bits number, which is controlled by a counter. Then BUSY goes to ‘1’, which means the chip is doing the multiplication, which need 23*2 clock cycles. When the BUSY set to ‘0’, which means the multiplication is done. Then signal P out put products LSB first serially. So in this simulation the product is 1001110 which is: 78=6(110)*13(1101). Thus we can get the conclusion the multiplier works correct and match the design requirement. The clock is 50MHZ, so the overall time to do this multiplication expect the loading time is 20ns*23*2=960ns

Simulation: Bitslice
The output of the testbench is shown in Figure 21. The waveforms match the
expected output (shown in the testbench file), indicating that the model is functionally correct.

Figure 21: VHDL simulation of bitslice with delays from cell library datasheets

6. Floor Plan

The diagrams of floor plan are as follows.
7. Major Design Decisions
The controller design was revised so that the user would not need to pad their N-bit numbers with N extra 0s, which would have been an impractical requirement. At the core of the new counter unit is an N-bit counter that counts from 0 to N-2 three times during each operation. The first time is to load A and B, and pad with 0s if necessary. By 0-
padding in this way, the N-bit chip can be used with any shift register of width less than or equal to N. The second two times are to control the BUSY pin, which signals data ready and controls the shifting behavior of the U register.

The decision to use a counter-based control unit rather than a zero-detector was made to eliminate the need for a 2N-bit bus and NOR/NAND arrays that a zero-detector would require. This seemed like an appropriate decision given that the primary goal of this project is to maximize capacity. A disadvantage of this design is an increased average number of computation cycles: The multiplication will always take 3N clock cycles to complete, whereas a zero-detector-based design may only take slightly longer than N for some inputs. It is our hope that the absence of the 2N-bit bus and lengthy interconnects will allow us to increase our clock speed and partially offset this problem.

The bitslice and controller were laid out in a way that will allow us to optimize our use of available space in the floorplan. The bitslice can be flipped upside-down successively to overlap the vdd and gnd rails, which also creates the desired alternating vdd and gnd pattern.

8. Division of Work
The task allocation needed to be modified due to design decisions and circumstances unknown to us at the time of original task allocation. Some changes lead to other changes so that the workload would remain balanced. These changes were unavoidable. Descriptions and justification of changes are given in Table 5.

<table>
<thead>
<tr>
<th>Task</th>
<th>Original Allocation</th>
<th>New Allocation</th>
<th>Reason for Re-allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revised First</td>
<td>Sun</td>
<td>Sun / Child</td>
<td>We decided that each of us should be responsible for updating our own sections from the first progress report.</td>
</tr>
<tr>
<td>Progress Report</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logic Hierarchy</td>
<td>Sun</td>
<td>Child</td>
<td>We were unable to install the logic design and simulation software on Mr. Sun’s computer, and Mr. Child had already designed most of the logic for the previous report.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Layouts and VHDL
Sun: SPICE, VHDL entities
Child: Layout, IRSIM, VHDL testbench

Sun: Controller VHDL, bitslice layout & SPICE
Child: Bitslice VHDL, controller layout & IRSIM

We did not have an idea of what our components would be when we first wrote the task allocation document. With the controller and bitslice designed, we decided that we should each layout and simulate one component, and do the VHDL for the other team member’s component. This way, we both fully understand the structure and behavior of both major modules.

Floorplan
Child
Sun
Re-allocated to balance with logic hierarchy changes.

| Layouts and VHDL | Sun: SPICE, VHDL entities | Sun: Controller VHDL, bitslice layout & SPICE | We did not have an idea of what our components would be when we first wrote the task allocation document. With the controller and bitslice designed, we decided that we should each layout and simulate one component, and do the VHDL for the other team member’s component. This way, we both fully understand the structure and behavior of both major modules. |
| Floorplan | Child | Sun | Re-allocated to balance with logic hierarchy changes. |

Table 5: Description of task allocation changes

The final task allocation for this second progress report is given in Table 6.

| Ryan Child | Revised First Progress Report Logic Hierarchy Layouts – Controller VHDL & Simulations – Bitslice Major Design Decisions |
| Xinyu Sun | Revised First Progress Report Layouts – Bitslice VHDL – Controller VHDL & Simulations – Chip Floor Plan |

Table 6: Division of Work
Chapter 3 - Third Progress Report
Revised First and Second Progress Reports

Pinout

Figure 23: Revised pin-out diagram of SUNCHILD multiplier.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLK</td>
<td>I</td>
<td>Clock</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>---</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>2</td>
<td>RESET</td>
<td>I</td>
<td>Synchronous Reset</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>I</td>
<td>Factor A Serial Input</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>I</td>
<td>Factor B Serial Input</td>
</tr>
<tr>
<td>5</td>
<td>WR</td>
<td>I</td>
<td>Write. Drive to 0V for N clock cycles to simultaneously read factors into shift registers.</td>
</tr>
<tr>
<td>6</td>
<td>SE</td>
<td>I</td>
<td>Scan Chain Input Enable</td>
</tr>
<tr>
<td>7</td>
<td>BUSY</td>
<td>O</td>
<td>High during multiplication. Transitions low when multiplication is complete and latched into internal shift register.</td>
</tr>
<tr>
<td>8</td>
<td>SI</td>
<td>I</td>
<td>Scan Chain Input</td>
</tr>
<tr>
<td>9</td>
<td>SO</td>
<td>O</td>
<td>Scan Chain Output</td>
</tr>
<tr>
<td>10</td>
<td>P / TUO0</td>
<td>O</td>
<td>Product Serial Output / Test Uout of Slice 0</td>
</tr>
<tr>
<td>11</td>
<td>NC</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>TAO</td>
<td>O</td>
<td>Test Aout of Slice 0</td>
</tr>
<tr>
<td>13</td>
<td>TBO</td>
<td>O</td>
<td>Test Bout of Slice 0</td>
</tr>
<tr>
<td>14</td>
<td>TUO1</td>
<td>Test Uout of Slice 1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>NC</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>GND</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>17</td>
<td>NC</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>TSBO</td>
<td>O</td>
<td>Test Slice Bout</td>
</tr>
<tr>
<td>19</td>
<td>TSSBR</td>
<td>I</td>
<td>Test Slice Shift B Right</td>
</tr>
<tr>
<td>20</td>
<td>TSUO</td>
<td>O</td>
<td>Test Slice Uout</td>
</tr>
<tr>
<td>21</td>
<td>TSBR</td>
<td>I</td>
<td>Test Slice B Right. Connects to Bout of slice to the right.</td>
</tr>
<tr>
<td>22</td>
<td>TSBL</td>
<td>I</td>
<td>Test Slice B Left. Connects to Bout of slice to the left.</td>
</tr>
<tr>
<td>23</td>
<td>TSSE</td>
<td>I</td>
<td>Test Slice Scan Enable</td>
</tr>
<tr>
<td>24</td>
<td>TSAO</td>
<td>O</td>
<td>Test Slice Aout</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>25</td>
<td>TSAI</td>
<td>I</td>
<td>Test Slice Ain</td>
</tr>
<tr>
<td>26</td>
<td>TSSO</td>
<td>O</td>
<td>Test Slice Scan Out</td>
</tr>
<tr>
<td>27</td>
<td>TSSI</td>
<td>I</td>
<td>Test Slice Scan In</td>
</tr>
<tr>
<td>28</td>
<td>TSA0</td>
<td>I</td>
<td>Test Slice A0. Connects to Aout of last slice in chain.</td>
</tr>
<tr>
<td>29</td>
<td>TSSR</td>
<td>I</td>
<td>Test Slice Shift Result</td>
</tr>
<tr>
<td>30</td>
<td>TDQ</td>
<td>O</td>
<td>Test positive edge DFF Q</td>
</tr>
<tr>
<td>31</td>
<td>TSVO</td>
<td>O</td>
<td>Test Slice Vout</td>
</tr>
<tr>
<td>32</td>
<td>TSUI</td>
<td>I</td>
<td>Test Slice Uin</td>
</tr>
<tr>
<td>33</td>
<td>TSVI</td>
<td>I</td>
<td>Test Slice Vin</td>
</tr>
<tr>
<td>34</td>
<td>TSCLK</td>
<td>I</td>
<td>Test Slice clock</td>
</tr>
<tr>
<td>35</td>
<td>TSR</td>
<td>I</td>
<td>Test Slice Reset</td>
</tr>
<tr>
<td>36</td>
<td>VDD</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>37</td>
<td>TDD</td>
<td>I</td>
<td>Test positive edge DFF D</td>
</tr>
<tr>
<td>38</td>
<td>TDCLK</td>
<td>I</td>
<td>Test positive edge DFF clock</td>
</tr>
<tr>
<td>39</td>
<td>TIA</td>
<td>I</td>
<td>Test Inverter A</td>
</tr>
<tr>
<td>40</td>
<td>TIY</td>
<td>O</td>
<td>Test Inverter Y</td>
</tr>
</tbody>
</table>

**Table 7: Pin numberings and descriptsions**

**Bit-slice Layout**

The bit-slice was revised to include buffers for global signals SE and CLK. Without these buffers, the load capacitance of these signals at the top of the slice stack, not accounting for wire capacitance, would be $0.032\text{pF} \times 4 \times 20 = 2.56\text{pF}$ for SE, and $0.05\text{pF} \times 4 \times 20 = 4\text{pF}$ for CLK, both of which are greater than the maximum capacitance of the strongest buffer cell from the library (BUFX4 Driving Strength Limit = 1.85pF). Also, signals were re-routed such that the slices would be able to be connected with minimal space (overlapping power rails). The revised bit-slice layout is shown in Figure 24.
VHDL

In this report due to the space limits of the pad frame, we changed the controller to control 20x20 multiplication, rather than 23x23 we did in the second report. The VHDL for the revised controller is as follows:

```vhdl
-- file: RPA controller.vhd
-- Author: Xinyu Sun (University of Cincinnati)
-- Created: 12/2/2010
-- Updates: None
-- Note:
*******************************************************************************
library ieee;
use ieee.std_logic_1164.all;

entity controller is
    port (wr, sei, si, clki, reset: in std_logic;
          so, sbr, pab: out std_logic);
end controller;

-- define structural architecture
architecture structure of controller is

component dffposx1
    generic(delay: time);
    port(d, clk: in std_logic;
         q: out std_logic);
end component;

component mux2x1
    generic(delay: time);
    port(a, b, s: in std_logic;
         y: out std_logic);
end component;

component and2x1
    generic(delay: time);
    port(a, b: in std_logic;
         y: out std_logic);
end component;

component nand2x1
    generic(delay: time);
    port(a, b: in std_logic;
         y: out std_logic);
end component;
```
"end component;

component invx1 is
  generic(delay: time);
  port(a: in std_logic;
       y: out std_logic);
end component;

component xor2X1 is
  generic(delay: time);
  port(a, b: in std_logic;
       y: out std_logic);
end component;

component or2X1 is
  generic(delay: time);
  port(a, b: in std_logic;
       y: out std_logic);
end component;

component nor2X1 is
  generic(delay: time);
  port(a, b: in std_logic;
       y: out std_logic);
end component;

component nor3X1 is
  generic(delay: time);
  port(a, b, c: in std_logic;
       y: out std_logic);
end component;

component nand3X1 is
  generic(delay: time);
  port(a, b, c: in std_logic;
       y: out std_logic);
end component;

-- general purpose wires
signal scnt, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15, s16, s17, s18, s19, s20, s21, s22, s23, s24, s25, s26, s27, s28, s29, s30, s31, s32, s33, s34, s35, s36, s37, s38, s39, s40, s41: std_logic;

begin
  s1 <= wr;
  s2 <= reset;
  -- connect wires to outputs
  pab <= wr;
  so <= s7;
  sbr <= not s7;

  -- DFFs
  dffl: dffposx1 generic map(delay => 0.34 ns) port map(d => s19, clk => clki, q => s20);"
dff2: dffposx1 generic map(delay => 0.338 ns) port map(d => s22, clk => clki, q => s23);
    dff3: dffposx1 generic map(delay => 0.34 ns) port map(d => s27, clk => clki, q => s28);
    dff4: dffposx1 generic map(delay => 0.338 ns) port map(d => s32, clk => clki, q => s33);
    dff5: dffposx1 generic map(delay => 0.34 ns) port map(d => s37, clk => clki, q => s38);
    dff6: dffposx1 generic map(delay => 0.415 ns) port map(d => s17, clk => clki, q => s18);
    dff7: dffposx1 generic map(delay => 0.45 ns) port map(d => s12, clk => clki, q => s19);

    --MUXs
    mux1: mux2x1 generic map(delay => 0.0832 ns) port map(a => s6, b => s18, s => sei, y => s19);
    mux2: mux2x1 generic map(delay => 0.0832 ns) port map(a => s20, b => s21, s => sei, y => s22);
    mux3: mux2x1 generic map(delay => 0.0832 ns) port map(a => s23, b => s26, s => sei, y => s27);
    mux4: mux2x1 generic map(delay => 0.0832 ns) port map(a => s28, b => s31, s => sei, y => s32);
    mux5: mux2x1 generic map(delay => 0.0832 ns) port map(a => s33, b => s36, s => sei, y => s37);
    mux6: mux2x1 generic map(delay => 0.0832 ns) port map(a => s16, b => s15, s => sei, y => s17);
    mux7: mux2x1 generic map(delay => 0.0832 ns) port map(a => s8, b => s11, s => sei, y => s12);

    --INVs
    inv1: invx1 generic map(delay => 0.3354 ns) port map(a => s4, y => s41);
    inv2: invx1 generic map(delay => 0.0909 ns) port map(a => si, y => s6);

    --Nands
    nand1: nand3x1 generic map(delay => 0.15 ns) port map(a => s41, b => s3, c => s2, y => s18);
    nand2: nand3x1 generic map(delay => 0.15 ns) port map(a => s41, b => s38, c => s2, y => s21);
    nand3: nand3x1 generic map(delay => 0.15 ns) port map(a => s41, b => s25, c => s2, y => s26);
    nand4: nand3x1 generic map(delay => 0.15 ns) port map(a => s41, b => s30, c => s2, y => s31);
    nand5: nand3x1 generic map(delay => 0.15 ns) port map(a => s41, b => s35, c => s2, y => s36);
    nand6: nand2x1 generic map(delay => 0.104 ns) port map(a => s2, b => s14, y => s15);
    nand7: nand2x1 generic map(delay => 0.104 ns) port map(a => s2, b => s10, y => s11);
    nand8: and2x1 generic map(delay => 0.15 ns) port map(a => s16, b => s23, y => s39);

    --NORs
    nor1: and2x1 generic map(delay => 0.15 ns) port map(a => s39, b => s40, y => s4);
    nor2: nor3x1 generic map(delay => 0.15 ns) port map(a => s8, b => s7, c => s1, y => s5);

    --ORs
```vhdl
-- ORs
or1: or3x1 generic map(delay => 0.167 ns) port map (a=>s20, b=>s33, c=>s28, y=>s40);
or2: or2x1 generic map(delay => 0.226 ns) port map (a=>s4, b=>s5, y=>s13);
or3: or2x1 generic map(delay => 0.549 ns) port map (a=>s7, b=>s8, y=>scnt);
-- XORs
xor1: xor2x1 generic map(delay => 0.08 ns) port map (a=>scnt, b=>s20, y=>s3);
xor2: xor2x1 generic map(delay => 0.08 ns) port map (a=>s20, b=>s23, y=>s38);
xor3: xor2x1 generic map(delay => 0.08 ns) port map (a=>s24, b=>s28, y=>s25);
xor4: xor2x1 generic map(delay => 0.08 ns) port map (a=>s33, b=>s29, y=>s30);
xor5: xor2x1 generic map(delay => 0.08 ns) port map (a=>s34, b=>s16, y=>s35);
xor6: xor2x1 generic map(delay => 0.08 ns) port map (a=>s13, b=>s8, y=>s14);
xor7: xor2x1 generic map(delay => 0.08 ns) port map (a=>s9, b=>s7, y=>s10);
-- ANDs
and1: and2x1 generic map(delay => 0.185 ns) port map (a=>s23, b=>s20, y=>s24);
and2: and2x1 generic map(delay => 0.185 ns) port map (a=>s24, b=>s28, y=>s29);
and3: and2x1 generic map(delay => 0.185 ns) port map (a=>s29, b=>s33, y=>s34);
and4: and2x1 generic map(delay => 0.185 ns) port map (a=>s8, b=>s4, y=>s9);
end structure;
```

### Layout

The hierarchy of the top-level logic layout is shown in Figure 25. Rationale of placement and routing is given in the Major Design Decisions section on page 3-21.
Figure 25: Hierarchical Layout showing placement and ordering of slices and controller

The chip layout including pad frame is shown below as Figure 26.
Figure 26: Chip layout including pad frame

**IRSIM/SPICE Simulations**

**IRSIM .cmd file:**

```
 analyzer CLK  RESET  SE  SI  WR  B  A  SO  BUSY  P
 stepsize 10
 l  CLK
 h  VDD
 l  GND
 l  RESET
 l  SE
 h  WR
```
Simulation of normal mode

Figure 27: normal mode irsim simulation
**Explanation:**

This is the worst case pattern for our chip. The input for the A is 11111111111111111111, and the B is 10101010101010101010. After the BUSY signal turns to 0 the output P signal get the product which is 101010101010101010101010101010101010110. So the normal mode works correct. In addition, due to the worst case path, we get the minimum clock period is 2.2ns. Thus the maximum clock for the normal mode is \( \frac{1}{2.2\text{ns}} = 454\text{Mhz} \).

**Simulation of test mode**

Due the space of the report we did not attach the test mode irsim cmd file here. However it is similar as the normal mode except using the SI rather than the A and B.

![Figure 28: test mode irsim simulation](image)

**Explanation:**

When we switch the SE to high, the mode will go to the test mode. SI is the input for the scan chain, and the SO is the output for the scan chain. The scan chain has totally \( 7+40 \times 4 \) = 167 DFFs. Thus the signal from SI to SO should take 167 clock cycles. From the waveform we can see the test mode works correct. However the minimum clock period for the test mode is 2.4 ns which is longer than the normal mode.

**Simulation strategy explanation**

There are five work processes in chip. They are the reset process, the data input process, the computing process, the data output process, and the test mode process. The simulation test includes all these processes.

Reset process: the RESET signal in our design is negative active. In the first clock cycle, the reset signal and the SE signal should both be ‘0’, and the WR signal is ‘1’. Besides A and B can be ‘0’ at the same time. Keep for one cycle, and then the value in the controller and the U, V register in bits-lice will be reset to be ‘0’.

Data input process: The signal WR and SE should be set as ‘0’, and the reset signal goes to ‘1’. Data A and data B, which should be 20 bit binary number, are put in
separately from PIN A and PIN B. Then system needs 20 cycles time, to load the inputs.

Computing process: The WR signal should go back to ‘1’. At the same time the SE is still set at ‘0’, which means it works at the normal mode.

Data output process: This process will be carried out when the controller finished all the shifting, when all the A registers are 0s. At this time BUSY signal turns to ‘0’.

Test mode process: When we want to test the scan chain, the SE signal should be ‘1’. Then the scan in signal SI can pass the value into the first register of the scan chain, then goes out from the SO signal which is in the last register of the scan chain. Totally we have 167 DFFs, so, in theoretic the So should comes out the value from the SI after 167 clock cycle.

**Chip clock frequency**

As we discussed in the wave form simulation the minimum clock cycle for the chip is the max of the two mode which is 2.4ns. then we get the maximum clock frequency is 417Mhz.

**VHDL**

Vhdl code for test bench of the final chip:

```vhdl
--
***************************************************************
-- file: sun-child testbench.vhd
-- Author: Xinyu Sun (University of Cincinnati)
-- Created: 12/2/2010
-- Updates: None
-- Note:
--
***************************************************************
library ieee;
use ieee.std_logic_1164.all;

entity tb_rpa_chip is
end tb_rpa_chip;

architecture tb_arch of tb_rpa_chip is

  component rpa_chip
    generic ( width : integer );
    port ( A,B, WR, SI, SE, CLK, RESET : in std_logic ;
          P,SO,BUSY : out std_logic );
  end component;

signal t_A, t_B, t_WR, t_SI,t_SE, t_CLK, t_RESET, t_P, t_SO,t_BUSY : std_logic;
constant bit_width : integer := 20;
begin
  U1 : rpa_chip
    generic map ( bit_width )
    port map ( t_A, t_B, t_WR, t_SI,t_SE, t_CLK, t_RESET, t_P, t_SO,t_BUSY );
```
process
begin
  t_CLK <= '0';
  wait for 1 ns;
  t_CLK <= '1';
  wait for 1 ns;
end process;

test_process : process
begin
  t_WR <= '1';
  --test mode
  t_SE <= '1';
  t_RESET<= '0'; --WORKING MODE( when u wanna use test mode please delet
this)
  t_RESET<='0';-- reset signal
  wait for 2 ns;
  t_RESET<='1';
  t_WR <= '0';
  t_A <= '1';
  t_B <= '0';
  --scan in
  --t_SI<='1';
  wait for 2 ns;
  --t_SI<='0';
  t_A <= '1';
  t_B <= '1';
  wait for 2 ns;
  --t_SI<='1';
  t_A <= '1';
  t_B <= '0';
  wait for 2 ns;
  t_A <= '1';
  t_B <= '1';
  --scan in
  --t_SI<='1';
  wait for 2 ns;
  --t_SI<='0';
  t_A <= '1';
  t_B <= '1';
  wait for 2 ns;
  t_A <= '1';
  t_B <= '1';
  --scan in
  --t_SI<='1';
  wait for 2 ns;
  --t_SI<='0';
  t_A <= '1';
  t_B <= '1';
  --scan in
  --t_SI<='1';
  wait for 2 ns;
  --t_SI<='0';
  t_A <= '1';
  t_B <= '1';
wait for 2 ns;
--t_SI<='1';
t_A <= '1';
t_B <= '0';
wait for 2 ns;
t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

--t_SI<='1';
t_A <= '1';
t_B <= '0';
wait for 2 ns;
t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

--t_SI<='1';
t_A <= '1';
t_B <= '0';
wait for 2 ns;
t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

--t_SI<='1';
t_A <= '1';
t_B <= '0';
wait for 2 ns;
t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

--t_SI<='1';
t_A <= '1';
t_B <= '0';
wait for 2 ns;
t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

t_A <= '1';
t_B <= '1';
--scan in
--t_SI<='1';
wait for 2 ns;

t_WR <= '1';
wait for 400 ns;
Final chip simulation

Figure 29: chip simulation

Explanation:
The input the A is 0xFFFFF (1048575), and the input for B is 0xAAAAA (699050), which are both inserted as LSB first serially. When we load the inputs the WR signal is hold on low, until the loading procedure is done. Then relief the WR, which goes back to 1. At this time the multiplication begins, which shifts 40 clk cycles, in which time the BUSY signal is hold on 1. When the multiplication procedure is done the BUSY signal goes back to low and the P get the answer, and output it LSB first serially, which is 0xAAAA955556 (733006353750). Form the wave form we can see the output is correct. Because this is a worst case pattern, the minimum clock period for it is 2.0 ns. Compare with the Irsim simulation, which is 2.4 ns, it is a little less. That is because we add the delay in to the leaf level of the chip in VHDL and it may ignore some internal Capacitances between the wires. However the Irsim consider more accurate than the VHDL. The result matches the requirement.

Test Strategy

In this chip design there are 4 parts we have to test from the PINs, which are as follows: first is the two standard cells. They are placed independently of our design. Second is the stand-alone bit-slice. Third is the test mode and normal mode. The forth part is test the signals inside the chip. The detailed description is as follows:
Standard Cells

The two standard cells in our chip are an inverter and a D-FF. They are placed completely isolated from the rest of our chip. There are 5 Pins connected to them to test whether they have no structural flow and make sure they are working. The 5 PINs are: TSA for the input of inv, TSY for the output of the inv; TSDCLK for the clk of the DFF, TSD for the input of the DFF and TSQ for the output of the DFF.

Bit-slice

Because our design is in case of bit-slice design, thus to make sure the bit slice works correct is very important for our design. We place a isolated bit slice for testing. The PINs which are connected to the bit slice are as follows:

<table>
<thead>
<tr>
<th>Test Pin Name</th>
<th>Pin Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSCLK</td>
<td>test slice clock</td>
</tr>
<tr>
<td>TSR</td>
<td>test slice reset</td>
</tr>
<tr>
<td>TSVO</td>
<td>test slice vout</td>
</tr>
<tr>
<td>TSVI</td>
<td>test slice vin</td>
</tr>
<tr>
<td>TSUI</td>
<td>test slice uin</td>
</tr>
<tr>
<td>TSUO</td>
<td>test slice uout</td>
</tr>
<tr>
<td>TSSR</td>
<td>test slice sr</td>
</tr>
<tr>
<td>TSA0</td>
<td>test slice a0</td>
</tr>
<tr>
<td>TSSI</td>
<td>test slice si</td>
</tr>
<tr>
<td>TSSO</td>
<td>test slice so</td>
</tr>
<tr>
<td>TSAI</td>
<td>test slice ain</td>
</tr>
<tr>
<td>TSAO</td>
<td>test slice aout</td>
</tr>
<tr>
<td>TSSE</td>
<td>test slice se</td>
</tr>
<tr>
<td>TSBO</td>
<td>test slice bout</td>
</tr>
<tr>
<td>TSBL</td>
<td>test slice bleft</td>
</tr>
<tr>
<td>TSBR</td>
<td>test slice bright</td>
</tr>
<tr>
<td>TSSBR</td>
<td>test slice sbr</td>
</tr>
</tbody>
</table>

Table 8: Pins for testing bit-slice

The user can get the information from the bit slice design based on this PINs.

Test mode / normal mode

The scan chain technique is used in this test mode, by which every flip-flop in the chip can be observed. When this signal is asserted, every flip-flop in the design is connected into a long shift register, the input pin provides the data to this chain, and the output pin is connected to the output of the chain. Then using the chip's clock signal, we can observe whether the data from output can match the data into the input in serial.

As for our chip, the Scan in (SI) goes from the first DDF in controller then goes out from the last DFF in the last bit slice, which is shown as SO in the Pin. It totally should take
(20x2x4+7) clock cycles. When the scan out does not follow the pattern, then the user should test the internal signal in the chip as well.

For testing the normal mode, the user can test the Pins which are connected to the inputs and outputs of the whole chip, and to see whether the chip realize the function of the RPA multiplication. The Pins are as follows: CLK, RESET, A, B, BUSY, SE, WR, SO, P.

**Internal signals**

There are 3 extra Pins to connect to the internal signal of the chip. They are pins TAO, TBO, and TUO1. The TAO is connected to the Aout in bit slice 0, which is used to see whether we load the input A correctly; the TBO is connected to the Bout in bit slice 0, which is used to see whether we load the input B correctly. And the TUO1 is connected to the Uout in bit slice0, which is used to test the result in the U register.

**User Guide**

See Table 7 on page 3-4 for a complete list of pins and descriptions.

Before using the chip it is necessary to reset by applying a zero at the **RESET** pin and a positive edge at the CLK pin, as shown in Figure 30. A and B can then be provided serially, LSB first, while simultaneously applying a zero to **WR**. **WR** must be held down for the exact length of the two factors for the chip to function correctly. Although the chip is designed to multiply 20-bit factors, it can also multiply factors of any smaller bit-length. As soon as the **WR** pin is returned to the high state, the higher-significance bits will be zero-padded automatically.

The BUSY output pin goes high after A and B are fully loaded, indicating that the multiplication has begun. The negative edge of the BUSY pin signals multiplication completion and can be used to latch the output on the P pin. The value of P when BUSY transitions low is the LSB of the product.

As an example, refer to Figure 30. First, the RESET pin is held low for one clock cycle. Next, **WR** is held low while A and B are serially loaded, LSB first. Here, A = 0x9D0B3 and B = 0x92656. The BUSY signal goes high immediately after the rising edge in which the MSB of the two factors is loaded. It transitions back to zero as the product is shifted out. Here, P = 0x59CE91AE22.
Figure 30: Timing diagram for SUNCHILD multiplier

Major Design Decisions

Our strategy for routing signals vertically through the bitslices proved to be very successful. By ordering the slices as shown in the hierarchical layout as shown in Figure 25 on page 3-9, we were able to connect all slices in a manner of minutes using the copy command of MAGIC. This also shows the advantage of the bitslice method of design in which the internal logic of the chip has regularity. The ordering of the slices was also chosen so that we would be able to “inject” a and B into the least significant half of the $2^N$-long shift register. This eliminates the need to shift through all $2^N$ bits and reduces the effective computation time by approximately 25%.

Buffers were added to the slices and to the top of the slice stack according to pin capacitances and estimated wire capacitances of the signals through the stack. Pin capacitances and output driving strengths of the cells in the library were obtained from the cell datasheets.

No logic design revisions were made since the second progress report.

Division of Work

According to the original task allocation statement, the layout was to be done by Mr. Sun. We revised this such that the work would be shared between setting up the internal logic and the connections to the pad frame, since the layout is so labor-intensive and time-consuming. The responsibilities for description of architecture and design decisions, and description of test mode were swapped as Mr. Child had written all of the design decisions sections to that point. Mr. Child did not do SPICE simulations because they were not called for. Mr. Sun did the VHDL simulations instead of Mr. Child because it was an extension of his work in the second progress report (also unforeseen at the time of task allocation).
| Ryan Child | • Revised First & Second Progress Reports - pinout  
|           | • Layout – top-level logic and pad frame connections, extraction and cif generation  
|           | • User Guide  
|           | • Major Design Decisions  
| Xinyu Sun | • Revised First & Second Progress Reports - VHDL  
|          | • Layout – connections to pad frame  
|          | • IRSIM Simulation  
|          | • VHDL  
|          | • Test Strategy |