Status and Control Semaphore Registers Using Partial Reconfiguration

XAPP 153 June 7, 1999 (Version 1.0)

Application Note by Nick Camilleri

Summary

The Virtex FPGA Series supports partial reconfiguration of a cross-section of data while the rest of the circuit is still in operation. This enables a system to read and write specific bits within a LUT configured as RAM, through the configuration port. This application note demonstrates how to lock the LUT SelectRAM to specific locations, determine the corresponding frame of data in the RBT (Rawbits) file, modify the LUT memory as desired, and re-write this frame into the chip. This provides a microprocessor/FPGA interface through the configuration port with a minimum of IOs.

Xilinx Families:

Virtex Series

Introduction

Advanced systems employing multiple FPGAs often use an embedded CPU for system-level operating tasks, including FPGA configuration: the CPU monitors operation of FPGA subsystems or provides real-time control; Networking systems may use FPGAs on communications ports; advanced systems designs exploit the soft FPGA logic to give field upgradability of hardware; ports can be independently configured based on network protocol requirements. In many systems, monitoring of real-time operation of FPGAs is needed. In an ATM switch, for example, cells passing through a port can be tabulated in an FPGA and read by a CPU at minute intervals to monitor network ATM cell traffic.

The Virtex SelectMAP™ interface enables high-speed (400 MB/s) configuration or partial configuration of the FPGA. Through a "Semaphore" mechanism, the SelectMAP port can read and write to the configured logic, which is the equivalent of microprocessor peripheral status and control registers. Semaphores rely on Virtex partial-configuration technology. When combined with re-configuration of FPGA logic, they let a single Status/Control register set be visible to the CPU, even as different designs are loaded into the FPGA.

Semaphore Methodology Overview

A Virtex SelectMAP configuration port is an 8-bit bi-directional port providing for:

- FPGA configuration
- FPGA configuration readback
- FPGA reconfiguration
- FPGA partial reconfiguration/Semaphore Write
- FPGA partial readback/Semaphore Read

Semaphores, special logic entities instantiatable in Verilog/VHDL designs, aid communications between FPGA logic and the Virtex SelectMAP ports. Via SelectMAP, an embedded CPU can write values to the Semaphore (Control) and read values from the Semaphore (Status). Semaphore units, connected to logic signals in the HDL design, then connect via normal FPGA routing. (See Figure 1).

This application note describes a 16-bit Semaphore module that uses RAM16X1D cells and the procedure to read and write to this module through the configuration port. For more information, see XAPP151, "Virtex Configuration Architecture Advanced User's Guide".

This concept can be used to modify or read status for a chip, but for simplicity, the Semaphore Module is limited to a 16-bit data word in this example. Semaphore Module data width is easily expanded.

Semaphore Module Definition

The Verilog source for instantiating a Semaphore Module into a simple counter design is shown in Figure 3 and Figure 4, and the Semaphore Module itself is listed in the "Appendix" on page 4. It is a column of 16 RAM16X1Ds configured as 1x1s, providing an interface between internal logic and an external host through the configuration port (see Table 1).

Table 1: Semaphore Module Ports

Signal Name	Type	Description
wclk	input	Write Clock
we	input	Write Enable
data[15:0]	input	Write Data
rout[15:0]	output	Read Data

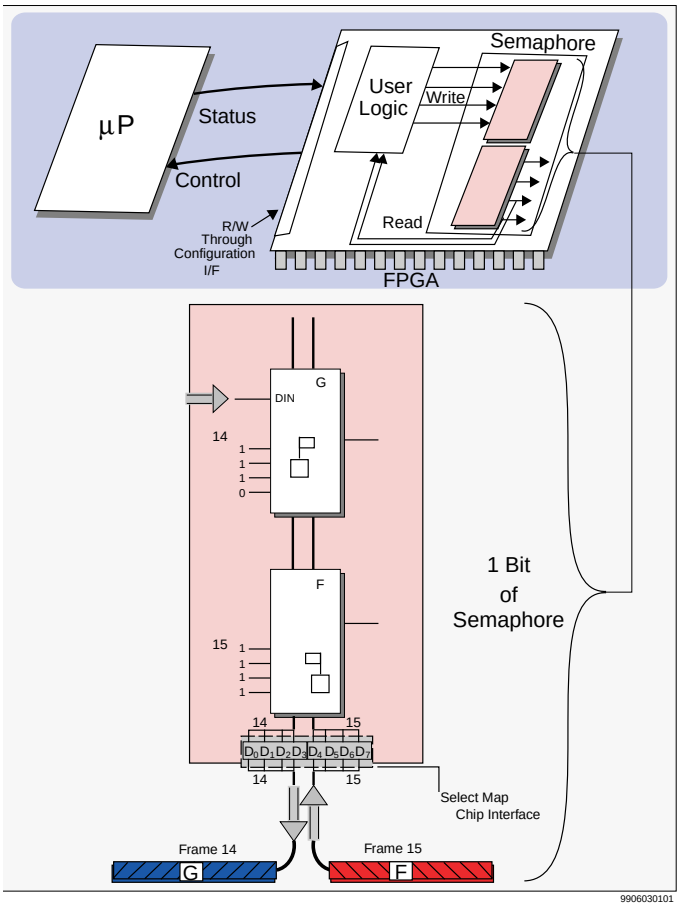


Figure 1: Semaphore Methodology

Internal logic uses the ports shown to write to and read from the Semaphore. The external host writes by executing a partial reconfiguration of one frame and changing SelectRAM bits as needed. A read is made by doing a partial readback and examining the desired bits.

The dual-port SelectRAM write port address is fixed to 1111, and the read port address is set to 1110. This effectively creates two individual 1x1 SelectRAMs, where each is used to send data back and forth between the internal logic and the external host in one of two directions. While it is not necessary to limit the address space to 1, it simplifies the design and makes communications quicker.

An “Updated” flag can be included to let internal logic know when the external host has reconfigured the SelectRAM cells. For example, this flag could be set to 1 when data is written to the FPGA through the configuration port, then reset after the data is read from the design logic.

By instantiating this module at the top-level of a Verilog design, the individual SelectRAM cells will have instantiation names of semaphore/rambit15, semaphore/rambit14, semaphore/rambit00. Physical placement of these SelectRAMs can then be locked within a constraints file (see Table 2). Alternatively, constraints could be entered in the source HDL, and embedded within the EDIF file. Constraints shown in Table 2 lock the 16-bit Semaphore module into CLB Column 5, Slice 0, spanning from Row 9 to Row 16. The bus order was chosen from MSB to LSB, which is the most convenient.

Determining Bit Locations in the RBT file

Once the design is compiled, placed and routed, the next step is to determine the bit locations within the RBT file that correspond to the SelectRAM bits. Please reference XAPP151, “Virtex Configuration Architecture Advanced User’s Guide” for detailed information.

A Semaphore module can be placed anywhere in the device, when no other LUT SelectRAMs or Shift Register LUTs are placed in the same CLB slice column. Otherwise the RAM/SRL contents would be overwritten. Multiple Semaphores can be placed within the same design, provided that they are not in the same CLB slice column.

The example (Figure 2) uses a XCV100 device. Frame size for an XCV100 is fourteen 32-bit words. The first 396 bits are memory cells, the next 20 are padding bits. The last 32 bits are a “dummy word” at the end of the frame.

The Major Address (internal CLB column reference address) is 22, which corresponds to CLB column 5, and the Minor Address is 47, which corresponds to the frame that contains LUT bit 15 (the fixed write address of the LUT SelectRAM).

Table 2: Sample Constraints (.UCF): Semaphore Module

Sample Constraints for Semaphore Module	
INST semaphore/rambit15	LOC = R5C5.S0;
INST semaphore/rambit14	LOC = R6C5.S0;
INST semaphore/rambit13	LOC = R7C5.S0;
INST semaphore/rambit12	LOC = R8C5.S0;
INST semaphore/rambit11	LOC = R9C5.S0;
INST semaphore/rambit10	LOC = R10C5.S0;
INST semaphore/rambit09	LOC = R11C5.S0;
INST semaphore/rambit08	LOC = R12C5.S0;
INST semaphore/rambit07	LOC = R13C5.S0;
INST semaphore/rambit06	LOC = R14C5.S0;
INST semaphore/rambit05	LOC = R15C5.S0;
INST semaphore/rambit04	LOC = R16C5.S0;
INST semaphore/rambit03	LOC = R17C5.S0;
INST semaphore/rambit02	LOC = R18C5.S0;
INST semaphore/rambit01	LOC = R19C5.S0;
INST semaphore/rambit00	LOC = R20C5.S0;

After this frame is located in the RBT file, LUT bits need to be determined within the frame. The bit position for address 1111 of rambit15 locked to Row 5 is 92 (with the bit index starting from 0). Every other bit is located 18 cells away, i.e. rambit14 is at bit 110, rambit13 is at bit 128, et cetera. LUT data bits are stored in their inverted sense. A bit value of 1 as seen by the internal logic is read and written as a 0 in the configuration memory.

Once the correct bits are modified, a partial reconfiguration RBT file needs to be created that can be downloaded to the chip after the original design is loaded.

```
Xilinx ASCII Bitstream
Created by reconp V1.00
Design name:     Sema4
Architecture:    virtex
Part:            xcv100pq240
Date:            Tue May 4 11:37:47 PST 1999
Bits:            1280
11111111111111111111111111111111 (dummy word #1)
11111111111111111111111111111111 (dummy word #2)
10100101001100101010101100110 (synchronization word)
00110000000000001000000000000001 (write to CMD register)
00000000000000000000000000000001 (register value for WCFG)
00110000000000000010000000000001 (write to FAR register)
00000000010110001011110000000000 (value for Major Address)
00110000000000000100000000000000 (write to FDRI register)
01010000000000000000000000011100 (value for FDRI wordcount)
...
...(Frame data)
...
11111111111111111111111111111111 (dummy frame,14 words total)
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
00110000000000001000000000000001 (write to CMD register)
00000000000000000000000000000000 (value for "NOP")
11111111111111111111111111111111 (dummy word #3)
```

9905240302

Figure 2: Sample .RBT File After Modification

Programming Semaphore Script

A PERL script called RECONP assists in programming the Semaphore Module. RECONP reads in an RBT file for the full user design (by running BITGEN with the -b option on <design>.ncd), queries the user for information about the final placement location of the Top-Left corner of the Semaphore (in the example from Table 2, this would be R5C5.S0) and the value to be written to the Semaphore, and proceeds to write out two Rawbits files. The first, <design>.rbt (see example in Figure 2), is used when the external host wants to write to the Semaphore, and the second (<design>.rbr) is used for doing a partial readback of information contained in the Semaphore.

Write function of RECONP has been tested in hardware.

References

Data files for this application note are available at: ftp://ftp.xilinx.com/pub/applications/xapp/xapp153.zip.

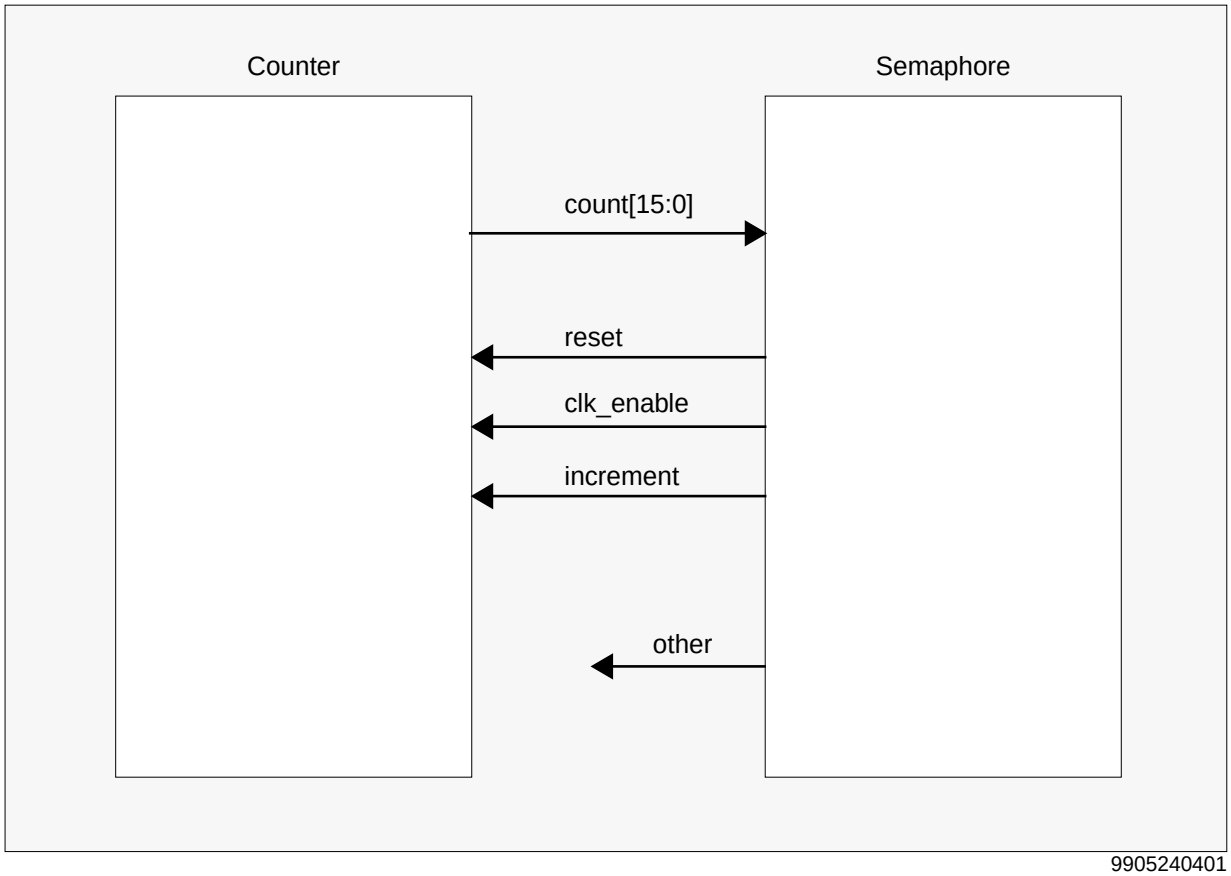
```
module sema4 (clk, we, sout);
input clk, we;
output [15:0] sout
reg [15:0] count;
wire [15:0] semout;
wire reset, clk_enable, increment;
assign reset = semout[0];
assign clk_enable = semout[1];
assign increment = semout[2];


always @(posedge clk)
  if (reset) count <= 'h0;
  else if (clk_enable)
    begin
      if (increment) count <= count +1;
      else count <= count -1;
    end


semaphore semaphore (.wclk(clk), .we(we), .data(count), .rout(semout) );
assign sout = semout;
endmodule
```

9905240201

Figure 3: Verilog Source Example: Instantiated Semaphore Module



9905240401

Figure 4: Block Diagram of Instantiated Semaphore Module

Appendix

Code Example 1 Verilog Source for Semaphore module

```
module semaphore (wclk, we, data, rout);
input wclk, we;
input [15:0] data;
output [15:0] rout;
wire logic0 = 0;
wire logic1 = 1;
RAM16X1D rambit15 (.WCLK(wclk), .WE(we), .D(data[15]), .DPO(rout[15]), .A3(logic1),
    .A2(logic1), .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), DPRA1(logic1),
    .DPRA0(logic0));
RAM16X1D rambit14 (.WCLK(wclk), .WE(we), .D(data[14]), .DPO(rout[14]), .A3(logic1),
    .A2(logic1), .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1),
    .DPRA0(logic0));
RAM16X1D rambit13 (.WCLK(wclk), .WE(we), .D(data[13]), .DPO(rout[13]), .A3(logic1),
    .A2(logic1), .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1),
    .DPRA0(logic0));
RAM16X1D rambit14 (.WCLK(wclk), .WE(we), .D(data[14]), .DPO(rout[14]), .A3(logic1),
    .A2(logic1), .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1),
    .DPRA0(logic0));
RAM16X1D rambit13 (.WCLK(wclk), .WE(we), .D(data[13]), .DPO(rout[13]), .A3(logic1),
    .A2(logic1), .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1),
    .DPRA0(logic0));
RAM16X1D rambit12 (.WCLK(wclk), .WE(we), .D(data[12]), .DPO(rout[12]), .A3(logic1),
    .A2(logic1), .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1),
    .DPRA0(logic0));
RAM16X1D rambit11 (.WCLK(wclk), .WE(we), .D(data[11]), .DPO(rout[11]), .A3(logic1),
    .A2(logic1), .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1),
    .DPRA0(logic0));
RAM16X1D rambit10 (.WCLK(wclk), .WE(we), .D(data[10]), .DPO(rout[10]), .A3(logic1),
    .A2(logic1), .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1),
    .DPRA0(logic0));
RAM16X1D rambit09 (.WCLK(wclk), .WE(we), .D(data[9]), .DPO(rout[9]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
RAM16X1D rambit08 (.WCLK(wclk), .WE(we), .D(data[8]), .DPO(rout[8]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
RAM16X1D rambit07 (.WCLK(wclk), .WE(we), .D(data[7]), .DPO(rout[7]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
RAM16X1D rambit06 (.WCLK(wclk), .WE(we), .D(data[6]), .DPO(rout[6]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
RAM16X1D rambit05 (.WCLK(wclk), .WE(we), .D(data[5]), .DPO(rout[5]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
RAM16X1D rambit04 (.WCLK(wclk), .WE(we), .D(data[4]), .DPO(rout[4]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
RAM16X1D rambit03 (.WCLK(wclk), .WE(we), .D(data[3]), .DPO(rout[3]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
RAM16X1D rambit02 (.WCLK(wclk), .WE(we), .D(data[2]), .DPO(rout[2]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
RAM16X1D rambit01 (.WCLK(wclk), .WE(we), .D(data[1]), .DPO(rout[1]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
RAM16X1D rambit00 (.WCLK(wclk), .WE(we), .D(data[0]), .DPO(rout[0]), .A3(logic1), .A2(logic1),
    .A1(logic1), .A0(logic1), .DPRA3(logic1), .DPRA2(logic1), .DPRA1(logic1), .DPRA0(logic0));
endmodule
```