170 MHz FIFOs Using the Virtex Block SelectRAM+

XAPP131 December 10, 1998 (Version 1.1)

Application Note by Nick Camilleri

Summary

The Virtex FPGA Series provides dedicated on-chip blocks of 4096 bit dual-port synchronous RAM, which are ideal for use in FIFO applications. This application note describes a way to create a common-clock (synchronous) version and an independent-clock (asynchronous) version of a 512x8 FIFO, with the depth and width being adjustable within the Verilog code. A hand-placed version of the design runs at 170MHz in the -6 speed grade.

Xilinx Families

Virtex Series

Introduction

The Virtex Series of devices includes Block SelectRAM+™, which are fully synchronous dual-ported RAMs with 4096 memory cells. These blocks are ideal for FIFO applications, and each port can be configured independently as 4Kx1, 2Kx2, 1Kx4, 512x8 or 256x16.

This application note describes a 512x8 FIFO, but each port structure can be changed if the control logic is changed accordingly. First the design for a 512x8 FIFO with common Read and Write clocks is described, and then the design changes required for the more difficult case of independent Read and Write clocks are presented. Signal names in parenthesis are a reference to the name in the Verilog code.

Port Definitions - Common-Clock Design

Signal Name	Port Direction	Port Width
clock_in	input	1
fifo_gsr_in	input	1
write_enable_in	input	1
write_data_in	input	8
read_enable_in	input	1
read_data_out	output	8
full_out	output	1
empty_out	output	1
fifocount_out	output	4

Synchronous Design Using Common Clocks

When both the Read and Write clocks originate from the same source, it simplifies the operation and arbitration of the FIFO, and the Empty and Full flags can be generated more easily.

Linear Feedback Shift Registers (LFSRs) are used for both the read (read_addr) and write (write_addr) address counters. Because they are addressing a RAM, a binary counting sequence is not necessary, and the pseudo-random sequence of the LFSRs is acceptable. They use very little logic, and are therefore much faster than a standard binary implementation. The only drawback is that the FIFO size is reduced by one, to 511x8.

To perform a read, Read Enable (read_enable) is driven High prior to a rising clock edge, and the Read Data (read_data) will be presented on the outputs during the next clock cycle. To do a Burst Read, simply leave Read Enable High for as many clock cycles as desired, but if Empty goes active after reading, then the last word has been read, and the next Read Data would be invalid.

To perform a write, the Write Data (write_data) must be present on the inputs, and Write Enable (write_enable) is driven High prior to a rising clock edge. As long as the Full flag is not set, the Write will be executed. To do a Burst Write, the Write Enable is left High, and new Write Data must be available every cycle.

The Empty flag is set when the Next Read Address (next_read_addr) is equal to the current Write Address, and only a Read is being performed. This early decoding allows Empty to be set immediately after the last Read. It is cleared after a Write operation (with no simultaneous Read). Similarly, the Full flag is set when the Next Write Address (next_write_addr) is equal to the current Read Address, and only a Write is being performed. It is cleared after a Read operation (with no simultaneous Write). If both a Read and Write are done in the same clock cycle, there is no change to the status flags. During global reset (fifo_gsr), both these signals are driven High, to prevent any external logic from interfacing with the FIFO during this time.

A FIFO count (fifocount) is added for convenience, to determine when the FIFO is 1/2 full, 3/4 full, etc. It is a binary count of the number of words currently stored in the FIFO.

It is incremented on Writes, decremented on Reads, and stays the same if both operations are performed within the same clock cycle. In this application, only the upper 4 bits are sent to I/O, but that can easily be modified.

Port Definitions - Independent-Clock Design

Signal Name	Port Direction	Port Width
write_clock_in	input	1
read_clock_in	input	1
fifo_gsr_in	input	1
write_enable_in	input	1
write_data_in	input	8
read_enable_in	input	1
read_data_out	output	8
full_out	output	1
empty_out	output	1
fifostatus_out	output	5

Changes Required for Independent Clocks

In order to operate a FIFO with independent Read and Write clocks, some asynchronous arbitration logic is needed to determine the status flags. The previous Empty/Full generation logic and associated flip-flops are no longer reliable, because they are now asynchronous with respect to one another, since Empty is clocked by the Read Clock, and Full is clocked by the Write Clock.

To solve this problem, and to maximize the speed of the control logic, additional logic complexity is accepted for increased performance. There are primary 9-bit Read and Write binary address counters, which drive the address inputs to the Block RAM. The binary addresses are converted to Gray-code, and pipelined for a few stages to create several address pointers (read_addrgray, read_nextgray, read_lastgray, write_addrgray, write_nextgray) which are used to generate the Full and Empty flags as quickly as possible. Gray-code addresses are used so that the registered Full and Empty flags are always clean, and never in an unknown state due to the asynchronous relationship of the Read and Write clocks. In the worst case scenario, Full and Empty would simply stay active one cycle longer, but this would not generate an error.

When the Read and Write Gray-code pointers are equal, the FIFO is empty. When the Write Gray-code pointer is equal to the next Read Gray-code pointer, the FIFO is full, having 511 words stored. Additional comparators are used to determine when the FIFO is Almost Empty and Almost Full, so that Empty and Full can be generated on the same clock edge as the last operation. (Traditional control logic uses an asynchronous signal to set the flags, but this is much slower and limits the overall performance).

Because, different from the common-clock version, it is not possible to keep a reliable count of the number of words in the FIFO, a FIFO status output is used instead. It is 5 bits wide, with the signals representing various ranges of fullness, as seen in the following table.

FIFO Status Bit	Description
fifostatus[0]	FIFO is between Empty and 1/4 Full
fifostatus[1]	FIFO is between 1 word and 1/2 Full
fifostatus[2]	FIFO is between 1/4 full and 3/4 Full
fifostatus[3]	FIFO is between 1/2 Full and Full
fifostatus[4]	FIFO is between 3/4 Full and Full

The FIFOstatus outputs are mutually exclusive, meaning only one will be High at any one time, but the ranges that they cover overlap. They are based on the Gray-code pointers, and the quadrant deltas that exist between the Read and Write addresses. Because of the nature of Gray-code counting, more precision (such as one based on octants) can be easily added, because the upper bits of a Gray-code address are themselves Gray-coded, so there will not be any incorrect status registered.

The overall worst-case path is the generation of Full and Empty, and the delays for these paths are reported in the following table. Delays shown for hand-placed versions are near-optimal but not necessarily absolutely optimal.

Design Version	Speed Grade	
	-6	-5
Independent Clocks (automatic placement)	6.6 ns (150 MHz)	7.6 ns (130 MHz)
Independent Clocks (hand-placed)	5.9 ns (170 MHz)	6.7 ns (150 MHz)
Common Clocks (automatic placement)	6.0 ns (167 MHz)	6.9 ns (145 MHz)

The Independent Clock design has been bench-tested, and simulation test benches for each of the FIFO designs are provided, and have been simulated using the Model Tech Simulator.

The design for Independent Read and Write clocks is fifoctlr_ic.v (Figure 1), and the design for Common Read and Write clocks is fifoctlr_cc.v (Figure 2), on WebLINX.

Figure 1: Verilog Source Code for *fifoctlr_ic.v*

```
/*****************************************************************\
*                                                                 *
*  Module      : fifoctlr_ic.v                  Last Update: 10/19/98  *
*                                                                 *
*  Description : FIFO controller top level.                       *
*                Implements a 511x8 FIFO with independent read/write    *
*                clocks.                                          *
*                                                                 *
*  The following Verilog code implements a 511x8 FIFO in a Virtex       *
*  device.  The inputs are a Read Clock and Read Enable, a Write Clock  *
*  and Write Enable, Write Data, and a FIFO_gsr signal as an initial    *
*  reset.  The outputs are Read Data, Full, Empty, and the FIFOstatus   *
*  outputs, which indicate roughly how full the FIFO is.              *
*                                                                 *
\*****************************************************************/

‘timescale 1ns / 10ps

‘define DATA_WIDTH 7:0
‘define ADDR_WIDTH 8:0

module fifoctlr_ic (read_clock_in, write_clock_in, read_enable_in,
                    write_enable_in, fifo_gsr_in, write_data_in,
                    read_data_out, fifostatus_out, full_out,
                    empty_out);

input read_clock_in, write_clock_in;
input read_enable_in, write_enable_in;
input fifo_gsr_in;
input  [‘DATA_WIDTH] write_data_in;
output [‘DATA_WIDTH] read_data_out;
output [4:0] fifostatus_out;
output full_out, empty_out;

wire read_enable = read_enable_in;
wire write_enable = write_enable_in;
wire fifo_gsr = fifo_gsr_in;
wire [‘DATA_WIDTH] write_data = write_data_in;
wire [‘DATA_WIDTH] read_data;
assign read_data_out = read_data;
reg [4:0] fifostatus;
assign fifostatus_out = fifostatus;
reg full, empty;
assign full_out = full;
assign empty_out = empty;

reg [‘ADDR_WIDTH] read_addr, write_addr;
reg [‘ADDR_WIDTH] write_addrgray, write_nextgray;
reg [‘ADDR_WIDTH] read_addrgray, read_nextgray, read_lastgray;

wire read_allow, write_allow;

wire [‘ADDR_WIDTH] ecomp, aecomp, fcomp, afcomp;
wire [‘ADDR_WIDTH] emuxcyo, aemuxcyo, fmuxcyo, afmuxcyo;
```

```
wire emptyg, almostemptyg, fullg, almostfullg;
wire tempo1, tempo2, tempo3, tempo4;
wire ecin, fcin, aecin, afcin;

wire gnd = 0;
wire pwr = 1;

/*****************************************************************\
*                                                                 *
* Global input clock buffers are instantiated for both the read_clock *
* and the write_clock, to avoid skew problems.                    *
*                                                                 *
\*****************************************************************/

BUFGP gclkread (.I(read_clock_in), .O(read_clock));
BUFGP gclkwrite (.I(write_clock_in), .O(write_clock));

/*****************************************************************\
*                                                                 *
* BLOCK RAM instantiation for FIFO.  Module is 512x8, of which one     *
* address location is sacrificed for the overall speed of the design.  *
*                                                                 *
\*****************************************************************/

RAMB4_S8_S8 bram1 ( .ADDRA(read_addr), .ADDRB(write_addr), .DIB(write_data),
                    .DIA({gnd, gnd, gnd, gnd, gnd, gnd, gnd, gnd}),
                    .WEA(gnd), .WEB(write_allow), .CLKA(read_clock),
                    .CLKB(write_clock), .RSTA(gnd), .RSTB(gnd),
                    .ENA(read_allow), .ENB(pwr), .DOA(read_data) );

/*********************************************************\
*                                                         *
*  Empty flag is set on fifo_gsr (initial), or when gray    *
*  code counters are equal, or when there is one word in    *
*  the FIFO, and a Read operation is about to be performed  *
*                                                         *
\*********************************************************/

always @(posedge read_clock or posedge fifo_gsr)
   if (fifo_gsr) empty <= 'b1;
   else empty <= (emptyg || (almostemptyg && read_enable && ! empty));

/*********************************************************\
*                                                         *
*  Full flag is set on fifo_gsr (initial, but it is cleared *
*  on the first valid write_clock edge after fifo_gsr is    *
*  de-asserted), or when Gray-code counters are one away    *
*  from being equal (the Write Gray-code address is equal   *
*  to the Last Read Gray-code address), or when the Next    *
*  Write Gray-code address is equal to the Last Read Gray-  *
*  code address, and a Write operation is about to be       *
*  performed.                                             *
*                                                         *
\*********************************************************/
```

```
always @(posedge write_clock or posedge fifo_gsr)
   if (fifo_gsr) full <= 'b1;
   else full <= (fullg || (almostfullg && write_enable && ! full));

/*********************************************************\
*                                                         *
*  Generation of Read address pointers.  The primary one is  *
*  binary (read_addr), and the Gray-code derivatives are     *
*  generated via pipelining the binary-to-Gray-code result.  *
*  The initial values are important, so they're in sequence. *
*  Grey-code addresses are used so that the registered       *
*  Full and Empty flags are always clean, and never in an    *
*  unknown state due to the asynchronous relationship of the *
*  Read and Write clocks.  In the worst case scenario, Full  *
*  and Empty would simply stay active one cycle longer, but  *
*  it would not generate an error or give false values.      *
*                                                         *
\*********************************************************/

always @(posedge read_clock or posedge fifo_gsr)
   if (fifo_gsr) read_addr <= 'h0;
   else if (read_allow) read_addr <= read_addr + 1;

always @(posedge read_clock or posedge fifo_gsr)
   if (fifo_gsr) read_nextgray <= 9'b100000000;
   else if (read_allow)
      read_nextgray <= { read_addr[8], (read_addr[8] ^ read_addr[7]),
           (read_addr[7] ^ read_addr[6]), (read_addr[6] ^ read_addr[5]),
           (read_addr[5] ^ read_addr[4]), (read_addr[4] ^ read_addr[3]),
           (read_addr[3] ^ read_addr[2]), (read_addr[2] ^ read_addr[1]),
           (read_addr[1] ^ read_addr[0]) };

always @(posedge read_clock or posedge fifo_gsr)
   if (fifo_gsr) read_addrgray <= 9'b100000001;
   else if (read_allow) read_addrgray <= read_nextgray;

always @(posedge read_clock or posedge fifo_gsr)
   if (fifo_gsr) read_lastgray <= 9'b100000011;
   else if (read_allow) read_lastgray <= read_addrgray;

/*********************************************************\
*                                                         *
*  Generation of Write address pointers.  Identical copy of  *
*  read pointer generation above, except for names.          *
*                                                         *
\*********************************************************/

always @(posedge write_clock or posedge fifo_gsr)
   if (fifo_gsr) write_addr <= 'h0;
   else if (write_allow) write_addr <= write_addr + 1;

always @(posedge write_clock or posedge fifo_gsr)
   if (fifo_gsr) write_nextgray <= 9'b100000000;
   else if (write_allow)
      write_nextgray <= { write_addr[8], (write_addr[8] ^ write_addr[7]),
           (write_addr[7] ^ write_addr[6]), (write_addr[6] ^ write_addr[5]),
```

```
          (write_addr[5] ^ write_addr[4]), (write_addr[4] ^ write_addr[3]),
          (write_addr[3] ^ write_addr[2]), (write_addr[2] ^ write_addr[1]),
          (write_addr[1] ^ write_addr[0]) };

always @(posedge write_clock or posedge fifo_gsr)
   if (fifo_gsr) write_addrgray <= 9'b100000001;
   else if (write_allow) write_addrgray <= write_nextgray;

/*********************************************************\
*                                                         *
*  Generation of FIFOstatus outputs.  Used to determine how  *
*  full FIFO is, based on how far the Write pointer is ahead *
*  of the Read pointer.  Additional precision can be gained  *
*  by using additional (top) bits of Gray-code addresses.    *
*                                                         *
\*********************************************************/

always @(posedge write_clock or posedge fifo_gsr)
// for 0 to 1/4 full (quads equal)
   if (fifo_gsr) fifostatus[0] <= 'b1;
   else fifostatus[0] <= (read_addrgray[8:7] == write_addrgray[8:7]) &&
                         ! (fifostatus[3] || fifostatus[4]);

always @(posedge write_clock or posedge fifo_gsr)
// for 1 byte to 1/2 full  (quad 1 ahead)
   if (fifo_gsr) fifostatus[1] <= 'b0;
   else fifostatus[1] <=
        ((read_addrgray[8:7]=='b00)&&(write_addrgray[8:7]=='b01)) ||
        ((read_addrgray[8:7]=='b01)&&(write_addrgray[8:7]=='b11)) ||
        ((read_addrgray[8:7]=='b11)&&(write_addrgray[8:7]=='b10)) ||
        ((read_addrgray[8:7]=='b10)&&(write_addrgray[8:7]=='b00));

always @(posedge write_clock or posedge fifo_gsr)
// for 1/4 to 3/4 full (quad 2 ahead)
   if (fifo_gsr) fifostatus[2] <= 'b0;
   else fifostatus[2] <=
        ((read_addrgray[8:7]=='b00)&&(write_addrgray[8:7]=='b11)) ||
        ((read_addrgray[8:7]=='b01)&&(write_addrgray[8:7]=='b10)) ||
        ((read_addrgray[8:7]=='b11)&&(write_addrgray[8:7]=='b00)) ||
        ((read_addrgray[8:7]=='b10)&&(write_addrgray[8:7]=='b01));

always @(posedge write_clock or posedge fifo_gsr)
// for 1/2 to full (quad 3 ahead)
   if (fifo_gsr) fifostatus[3] <= 'b0;
   else fifostatus[3] <=
        ((read_addrgray[8:7]=='b00)&&(write_addrgray[8:7]=='b10)) ||
        ((read_addrgray[8:7]=='b01)&&(write_addrgray[8:7]=='b00)) ||
        ((read_addrgray[8:7]=='b11)&&(write_addrgray[8:7]=='b01)) ||
        ((read_addrgray[8:7]=='b10)&&(write_addrgray[8:7]=='b11));

always @(posedge write_clock or posedge fifo_gsr)
// for 3/4 to full (quad 4 ahead/equal)
   if (fifo_gsr) fifostatus[4] <= 'b0;
   else fifostatus[4] <= (read_addrgray[8:7] == write_addrgray[8:7]) &&
                         (fifostatus[3] || fifostatus[4]);
```

```
/*************************************************************\
*                                                             *
*  Allow flags determine whether FIFO control logic can       *
*  operate.  If read_enable is driven high, and the FIFO is   *
*  not Empty, then Reads are allowed.  Similarly, if the      *
*  write_enable signal is high, and the FIFO is not Full,     *
*  then Writes are allowed.                                   *
*                                                             *
\*************************************************************/

assign read_allow = (read_enable && ! empty);
assign write_allow = (write_enable && ! full);

/*************************************************************\
*                                                             *
*  The four conditions decoded with special carry logic are   *
*  Empty, AlmostEmpty, Full, and AlmostFull.  These are       *
*  used to determine the next state of the Full/Empty         *
*  flags.  Carry logic is used for optimal speed.             *
*                                                             *
*  When the Write/Read Gray-code addresses are equal, the     *
*  FIFO is Empty, and emptyg (combinatorial) is asserted.     *
*  When the Write Gray-code address is equal to the Next      *
*  Read Gray-code address (1 word in the FIFO), then the      *
*  FIFO potentially could be going Empty (if read_enable is   *
*  asserted, which is used in the logic that generates the    *
*  registered version of Empty).                              *
*                                                             *
*  Similarly, when the Write Gray-code address is equal to    *
*  the Last Read Gray-code address, the FIFO is full.  To     *
*  have utilized the full address space (512 addresses)       *
*  would have required extra logic to determine Full/Empty    *
*  on equal addresses, and this would have slowed down the    *
*  overall performance.  Lastly, when the Next Write Gray-    *
*  code address is equal to the Last Read Gray-code address   *
*  the FIFO is Almost Full, with only one word left, and      *
*  it is conditional on write_enable being asserted.          *
*                                                             *
\*************************************************************/

assign ecomp[0] = (write_addrgray[0] == read_addrgray[0]);
assign ecomp[1] = (write_addrgray[1] == read_addrgray[1]);
assign ecomp[2] = (write_addrgray[2] == read_addrgray[2]);
assign ecomp[3] = (write_addrgray[3] == read_addrgray[3]);
assign ecomp[4] = (write_addrgray[4] == read_addrgray[4]);
assign ecomp[5] = (write_addrgray[5] == read_addrgray[5]);
assign ecomp[6] = (write_addrgray[6] == read_addrgray[6]);
assign ecomp[7] = (write_addrgray[7] == read_addrgray[7]);
assign ecomp[8] = (write_addrgray[8] == read_addrgray[8]);

MUXCY_L emuxcyi (.DI(pwr), .CI(pwr),        .S(pwr),      .LO(ecin));
MUXCY_L emuxcy0 (.DI(gnd), .CI(ecin),       .S(ecomp[0]), .LO(emuxcyo[0]));
MUXCY_L emuxcy1 (.DI(gnd), .CI(emuxcyo[0]), .S(ecomp[1]), .LO(emuxcyo[1]));
MUXCY_L emuxcy2 (.DI(gnd), .CI(emuxcyo[1]), .S(ecomp[2]), .LO(emuxcyo[2]));
MUXCY_L emuxcy3 (.DI(gnd), .CI(emuxcyo[2]), .S(ecomp[3]), .LO(emuxcyo[3]));
MUXCY_L emuxcy4 (.DI(gnd), .CI(emuxcyo[3]), .S(ecomp[4]), .LO(emuxcyo[4]));
```

```
MUXCY_L emuxcy5 (.DI(gnd), .CI(emuxcyo[4]), .S(ecomp[5]), .LO(emuxcyo[5]));
MUXCY_L emuxcy6 (.DI(gnd), .CI(emuxcyo[5]), .S(ecomp[6]), .LO(emuxcyo[6]));
MUXCY_L emuxcy7 (.DI(gnd), .CI(emuxcyo[6]), .S(ecomp[7]), .LO(emuxcyo[7]));
MUXCY_L emuxcy8 (.DI(gnd), .CI(emuxcyo[7]), .S(ecomp[8]), .LO(emptyg));

assign aecomp[0] = (write_addrgray[0] == read_nextgray[0]);
assign aecomp[1] = (write_addrgray[1] == read_nextgray[1]);
assign aecomp[2] = (write_addrgray[2] == read_nextgray[2]);
assign aecomp[3] = (write_addrgray[3] == read_nextgray[3]);
assign aecomp[4] = (write_addrgray[4] == read_nextgray[4]);
assign aecomp[5] = (write_addrgray[5] == read_nextgray[5]);
assign aecomp[6] = (write_addrgray[6] == read_nextgray[6]);
assign aecomp[7] = (write_addrgray[7] == read_nextgray[7]);
assign aecomp[8] = (write_addrgray[8] == read_nextgray[8]);

MUXCY_L aemuxcyi (.DI(pwr), .CI(pwr),         .S(pwr),       .LO(aecin));
MUXCY_L aemuxcy0 (.DI(gnd), .CI(aecin),       .S(aecomp[0]), .LO(aemuxcyo[0]));
MUXCY_L aemuxcy1 (.DI(gnd), .CI(aemuxcyo[0]), .S(aecomp[1]), .LO(aemuxcyo[1]));
MUXCY_L aemuxcy2 (.DI(gnd), .CI(aemuxcyo[1]), .S(aecomp[2]), .LO(aemuxcyo[2]));
MUXCY_L aemuxcy3 (.DI(gnd), .CI(aemuxcyo[2]), .S(aecomp[3]), .LO(aemuxcyo[3]));
MUXCY_L aemuxcy4 (.DI(gnd), .CI(aemuxcyo[3]), .S(aecomp[4]), .LO(aemuxcyo[4]));
MUXCY_L aemuxcy5 (.DI(gnd), .CI(aemuxcyo[4]), .S(aecomp[5]), .LO(aemuxcyo[5]));
MUXCY_L aemuxcy6 (.DI(gnd), .CI(aemuxcyo[5]), .S(aecomp[6]), .LO(aemuxcyo[6]));
MUXCY_L aemuxcy7 (.DI(gnd), .CI(aemuxcyo[6]), .S(aecomp[7]), .LO(aemuxcyo[7]));
MUXCY_L aemuxcy8 (.DI(gnd), .CI(aemuxcyo[7]), .S(aecomp[8]), .LO(almostemptyg));

assign fcomp[0] = (write_addrgray[0] == read_lastgray[0]);
assign fcomp[1] = (write_addrgray[1] == read_lastgray[1]);
assign fcomp[2] = (write_addrgray[2] == read_lastgray[2]);
assign fcomp[3] = (write_addrgray[3] == read_lastgray[3]);
assign fcomp[4] = (write_addrgray[4] == read_lastgray[4]);
assign fcomp[5] = (write_addrgray[5] == read_lastgray[5]);
assign fcomp[6] = (write_addrgray[6] == read_lastgray[6]);
assign fcomp[7] = (write_addrgray[7] == read_lastgray[7]);
assign fcomp[8] = (write_addrgray[8] == read_lastgray[8]);

MUXCY_L fmuxcyi (.DI(pwr), .CI(pwr),        .S(pwr),      .LO(fcin));
MUXCY_L fmuxcy0 (.DI(gnd), .CI(fcin),       .S(fcomp[0]), .LO(fmuxcyo[0]));
MUXCY_L fmuxcy1 (.DI(gnd), .CI(fmuxcyo[0]), .S(fcomp[1]), .LO(fmuxcyo[1]));
MUXCY_L fmuxcy2 (.DI(gnd), .CI(fmuxcyo[1]), .S(fcomp[2]), .LO(fmuxcyo[2]));
MUXCY_L fmuxcy3 (.DI(gnd), .CI(fmuxcyo[2]), .S(fcomp[3]), .LO(fmuxcyo[3]));
MUXCY_L fmuxcy4 (.DI(gnd), .CI(fmuxcyo[3]), .S(fcomp[4]), .LO(fmuxcyo[4]));
MUXCY_L fmuxcy5 (.DI(gnd), .CI(fmuxcyo[4]), .S(fcomp[5]), .LO(fmuxcyo[5]));
MUXCY_L fmuxcy6 (.DI(gnd), .CI(fmuxcyo[5]), .S(fcomp[6]), .LO(fmuxcyo[6]));
MUXCY_L fmuxcy7 (.DI(gnd), .CI(fmuxcyo[6]), .S(fcomp[7]), .LO(fmuxcyo[7]));
MUXCY_L fmuxcy8 (.DI(gnd), .CI(fmuxcyo[7]), .S(fcomp[8]), .LO(fullg));

assign afcomp[0] = (write_nextgray[0] == read_lastgray[0]);
assign afcomp[1] = (write_nextgray[1] == read_lastgray[1]);
assign afcomp[2] = (write_nextgray[2] == read_lastgray[2]);
assign afcomp[3] = (write_nextgray[3] == read_lastgray[3]);
assign afcomp[4] = (write_nextgray[4] == read_lastgray[4]);
assign afcomp[5] = (write_nextgray[5] == read_lastgray[5]);
assign afcomp[6] = (write_nextgray[6] == read_lastgray[6]);
assign afcomp[7] = (write_nextgray[7] == read_lastgray[7]);
assign afcomp[8] = (write_nextgray[8] == read_lastgray[8]);
```

```
MUXCY_L afmuxcyi (.DI(pwr), .CI(pwr),         .S(pwr),       .LO(afcin));
MUXCY_L afmuxcy0 (.DI(gnd), .CI(afcin),       .S(afcomp[0]), .LO(afmuxcyo[0]));
MUXCY_L afmuxcy1 (.DI(gnd), .CI(afmuxcyo[0]), .S(afcomp[1]), .LO(afmuxcyo[1]));
MUXCY_L afmuxcy2 (.DI(gnd), .CI(afmuxcyo[1]), .S(afcomp[2]), .LO(afmuxcyo[2]));
MUXCY_L afmuxcy3 (.DI(gnd), .CI(afmuxcyo[2]), .S(afcomp[3]), .LO(afmuxcyo[3]));
MUXCY_L afmuxcy4 (.DI(gnd), .CI(afmuxcyo[3]), .S(afcomp[4]), .LO(afmuxcyo[4]));
MUXCY_L afmuxcy5 (.DI(gnd), .CI(afmuxcyo[4]), .S(afcomp[5]), .LO(afmuxcyo[5]));
MUXCY_L afmuxcy6 (.DI(gnd), .CI(afmuxcyo[5]), .S(afcomp[6]), .LO(afmuxcyo[6]));
MUXCY_L afmuxcy7 (.DI(gnd), .CI(afmuxcyo[6]), .S(afcomp[7]), .LO(afmuxcyo[7]));
MUXCY_L afmuxcy8 (.DI(gnd), .CI(afmuxcyo[7]), .S(afcomp[8]), .LO(almostfullg));

endmodule
```

Figure 2: Verilog source code for fifoctrl_cc.v

```
/*********************************************************************
*                                                                    *
*  Module      : fifoctlr_cc.v                  Last Update: 10/19/98  *
*                                                                    *
*  Description : FIFO controller top level.                          *
*                Implements a 511x8 FIFO with common read/write clocks. *
*                                                                    *
*  The following Verilog code implements a 511x8 FIFO in a Virtex       *
*  device.  The inputs are a Clock, a Read Enable, a Write Enable,      *
*  Write Data, and a FIFO_gsr signal as an initial reset.  The outputs  *
*  are Read Data, Full, Empty, and the FIFOcount outputs, which         *
*  indicate roughly how full the FIFO is.                            *
*                                                                    *
\*********************************************************************/

‘timescale 1ns / 10ps

‘define DATA_WIDTH 7:0
‘define ADDR_WIDTH 8:0

module fifoctlr_cc (clock_in, read_enable_in, write_enable_in,
                    write_data_in, fifo_gsr_in, read_data_out,
                    full_out, empty_out, fifocount_out );

input clock_in, read_enable_in, write_enable_in, fifo_gsr_in;
input  [‘DATA_WIDTH] write_data_in;
output [‘DATA_WIDTH] read_data_out;
output full_out, empty_out;
output [3:0] fifocount_out;

wire read_enable = read_enable_in;
wire write_enable = write_enable_in;
wire fifo_gsr = fifo_gsr_in;
wire [‘DATA_WIDTH] write_data = write_data_in;
wire [‘DATA_WIDTH] read_data;
assign read_data_out = read_data;
reg full, empty;
assign full_out = full;
assign empty_out = empty;
```

```
reg [‘ADDR_WIDTH] read_addr, write_addr, fcounter;

wire read_allow, write_allow;

assign read_allow = (read_enable && ! empty);
assign write_allow = (write_enable && ! full);

wire gnd = 0;
wire pwr = 1;

/*****************************************************************\
*                                                                 *
* A global buffer is instantiated to avoid skew problems.         *
*                                                                 *
\*****************************************************************/

BUFGP gclk1 (.I(clock_in), .O(clock));

/*****************************************************************\
*                                                                 *
* BLOCK RAM instantiation for FIFO. Module is 512x8, of which one *
* address location is sacrificed for the overall speed of the design. *
*                                                                 *
\*****************************************************************/

RAMB4_S8_S8 bram1 ( .ADDRA(read_addr), .ADDRB(write_addr), .DIB(write_data),
                    .DIA({gnd, gnd, gnd, gnd, gnd, gnd, gnd, gnd}),
                    .WEA(gnd), .WEB(write_allow), .CLKA(clock),
                    .CLKB(clock), .RSTA(gnd), .RSTB(gnd),
                    .ENA(read_allow), .ENB(pwr), .DOA(read_data) );

/*********************************************************\
*                                                         *
*  Empty flag is set on fifo_gsr (initial), or when on the  *
*  next clock cycle, Write Enable is low, and either the    *
*  FIFOcount is equal to 0, or it is equal to 1 and Read    *
*  Enable is high (about to go Empty).                      *
*                                                         *
\*********************************************************/

always @(posedge clock or posedge fifo_gsr)
   if (fifo_gsr) empty <= 1;
   else empty <= (! write_enable && (fcounter[8:1] == 8’h0) &&
                  ((fcounter[0] == 0) || read_enable));

/*********************************************************\
*                                                         *
*  Full flag is set on fifo_gsr (but it is cleared on the   *
*  first valid clock edge after fifo_gsr is removed), or    *
*  when on the next clock cycle, Read Enable is low, and    *
*  either the FIFOcount is equal to 1FF (hex), or it is     *
*  equal to 1FE and the Write Enable is high (about to go   *
*  Full).                                                   *
*                                                         *
\*********************************************************/
```

```
always @(posedge clock or posedge fifo_gsr)
   if (fifo_gsr) full <= 1;
   else full <= (! read_enable && (fcounter[8:1] == 8'hFF) &&
                 ((fcounter[0] == 1) || write_enable));

/*********************************************************\
*                                                         *
*  Generation of Read and Write address pointers.  They use  *
*  LFSR counters, which are very fast.  Because of the       *
*  nature of LFSR's, one address is sacrificed.              *
*                                                         *
\*********************************************************/

wire read_linearfeedback, write_linearfeedback;

assign read_linearfeedback =  ! (read_addr[8] ^ read_addr[4]);
assign write_linearfeedback = ! (write_addr[8] ^ write_addr[4]);

always @(posedge clock or posedge fifo_gsr)
   if (fifo_gsr) read_addr <= 'h0;
   else if (read_allow)
      read_addr <= { read_addr[7], read_addr[6], read_addr[5],
                     read_addr[4], read_addr[3], read_addr[2],
                     read_addr[1], read_addr[0], read_linearfeedback };

always @(posedge clock or posedge fifo_gsr)
   if (fifo_gsr) write_addr <= 'h0;
   else if (write_allow)
      write_addr <= { write_addr[7], write_addr[6], write_addr[5],
                      write_addr[4], write_addr[3], write_addr[2],
                      write_addr[1], write_addr[0], write_linearfeedback };

/*********************************************************\
*                                                         *
*  Generation of FIFOcount outputs.  Used to determine how   *
*  full FIFO is, based on a counter that keeps track of how  *
*  many words are in the FIFO.  Also used to generate Full   *
*  and Empty flags.  Only the upper four bits of the counter *
*  are sent outside the module                               *
*                                                         *
\*********************************************************/

always @(posedge clock or posedge fifo_gsr)
   if (fifo_gsr) fcounter <= 'h0;
   else if ((! read_allow && write_allow) || (read_allow && ! write_allow))
      begin
         if (write_allow)      fcounter <= fcounter + 1;
         else                  fcounter <= fcounter - 1;
      end

assign fifocount_out = fcounter[8:5];

endmodule
```

XILINX®

The Programmable Logic Company℠

Headquarters

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124
U.S.A.
Tel: 1 (800) 255-7778
or 1 (408) 559-7778
Fax: 1 (408) 559-7114
Email: hotline@xilinx.com
Web: http://www.xilinx.com

North America

Irvine, California
Tel: (949) 727-0780

Englewood, Colorado
Tel: (303) 220-7541

Sunnyvale, California
Tel: (408) 245-9850

Schaumburg, Illinois
Tel: (847) 605-1972

Nashua, New Hampshire
Tel: (603) 891-1098

Raleigh, North Carolina
Tel: (919) 846-3922

West Chester, Pennsylvania
Tel: (610) 430-3300

Dallas, Texas
Tel: (972) 960-1043

Europe

Xilinx Sarl
Jouy en Josas, France
Tel: (33) 1-34-63-01-01
Email: frhelp@xilinx.com

Xilinx GmbH
München, Germany
Tel: (49) 89-93088-0
Email: dlhelp@xilinx.com

Xilinx, Ltd.
Byfleet, United Kingdom
Tel: (44) 1-932-3494013
Email: ukhelp@xilinx.com

XILINX Italia
Via Alberto Mario 26
20149 Milano
Italia
Tel: (39) 02-49-877-16
FAX: (39) 02-49-877-27
Email: roberto.rosaia@xilinx.com

Japan

Xilinx, K.K.
Tokyo, Japan
Tel: (81) 3-5321-7711
Email: jhotline@xilinx.com

Asia Pacific

XILINX Asia Pacific
Unit 4312, Tower II
Metroplaza
Hing Fong Road
Kwai Fong, N.T.
Hong Kong
Tel: 852-2-424-5200
FAX: 852-2-494-7159
Email: hongkong@xilinx.com

XILINX Korea
Rm. 901, Sambo-Hojung Bldg.
14-24, Yoido-Dong,
Youngdeungpo-Ku
Seoul, 150-715, Korea
Tel : 822-761-4277
FAX : 822-761-4278
Email: korea@xilinx.com

Xilinx Taiwan
Rm. 1006, 10F, No.2. Lane 150
Sec. 5, Hsin Yin Rd.
Taipei, Taiwan, R.O.C.
Tel: 886-2-2758-8373
Tel: 886-2-2758-8353
Fax: 886-2-2758-8367