Hello, World: A Simple Application for the Field Programmable Port Extender (FPX)

John Lockwood and David Lim

The FPX provides simple and fast mechanisms to process cells or packets. By performing all computations in FPGA hardware, cells and packets can be processing at the full line speed of the card [currently 2.4 Gbits/sec]. A sample application, called 'Hello World' has been developed that illustrates how easily an application can be implemented on the FPX. This application uses the FPGA hardware to search for a string on a particular flow and selectively replace contents of the payload. The resulting circuit operates at 119 MHz on a Xilinx XCV 1000E-FG680-7, and occupies less than 1% of the available... **Read complete abstract on page** 2.

Hello, World: A Simple Application for the Field Programmable Port Extender (FPX)

John Lockwood and David Lim

Complete Abstract:

The FPX provides simple and fast mechanisms to process cells or packets. By performing all computations in FPGA hardware, cells and packets can be processing at the full line speed of the card [currently 2.4 Gbits/sec]. A sample application, called 'Hello World' has been developed that illustrates how easily an application can be implemented on the FPX. This application uses the FPGA hardware to search for a string on a particular flow and selectively replace contents of the payload. The resulting circuit operates at 119 MHz on a Xilinx XCV 1000E-FG680-7, and occupies less than 1% of the available gates on the device.

Hello, World: A Simple Application for the Field Programmable Port Extender (FPX)

John Lockwood, David Lim

July 11, 2000

Department of Computer Science
Applied Research Lab
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130

Abstract

The FPX provides simple and fast mechanisms to process cells or packets. By performing all computations in FPGA hardware, cells and packets can be processing at the full line speed of the card [currently 2.4 Gbits/sec]. A sample application, called 'Hello World' has been developed that illustrates how easily an application can be implemented on the FPX. This application uses the FPGA hardware to search for a string on a particular flow and selectively replace contents of the payload. The resulting circuit operates at 119 MHz on a Xilinx XCV1000E-FG680-7, and occupies less than 1% of the available gates on the device.

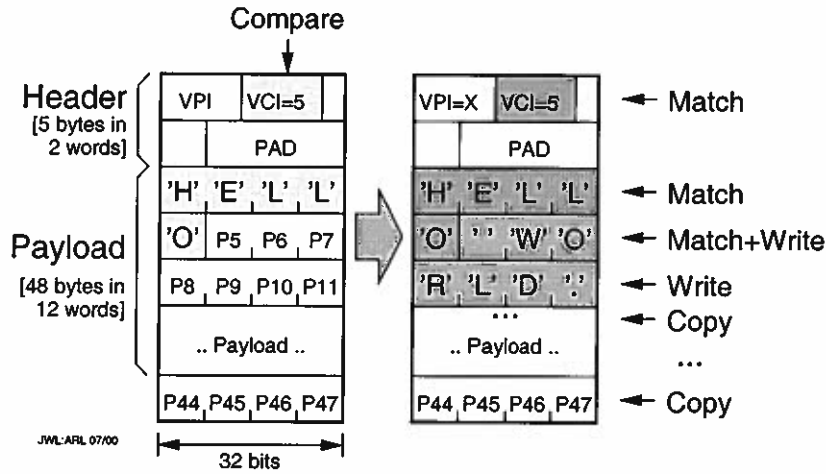# 1 Introduction

Figure 1: Cell processing for matching cell

As an example of a simple application for the FPX, consider an algorithm that searches the cells on a particular VCI that have payloads starting with the string "HELLO". If and only if we find such a match, we wish to concatenate that string with "World." A graphical view of how this algorithm operates on a cell is shown in Figure 1.

2 Discussion
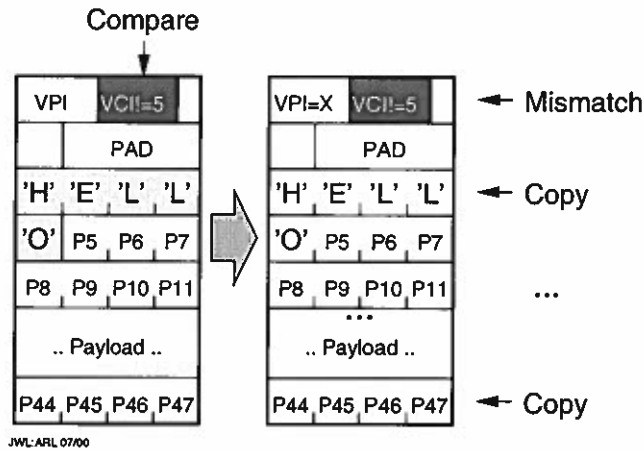
Figure 2: Cell processing for mismatched VCI

There are several cases in which the cell may not match. First, cells should only be processed if they arrive on the correct VCI. In this example, we have chosen to process cells on VCI=5. If the VCI doesn't match, the cell should should pass through the circuit without modification, as shown in Figure 2.
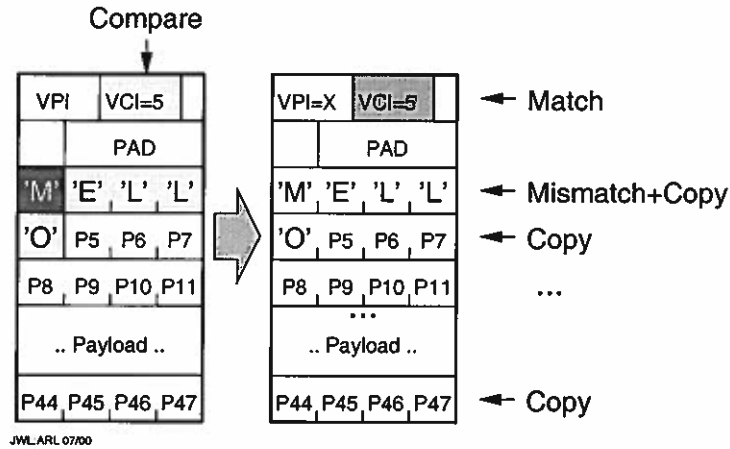
Figure 3: Cell processing for mismatched payload

Second, for those cells that do arrive on the correct VCI, the string must match over all words in the payload. For the string shown in Figure 3, a mismatch is found in the the first byte of the first word. Since the "MELLO" doesn't match "HELLO", the contents of the cell should be left unchanged.

3
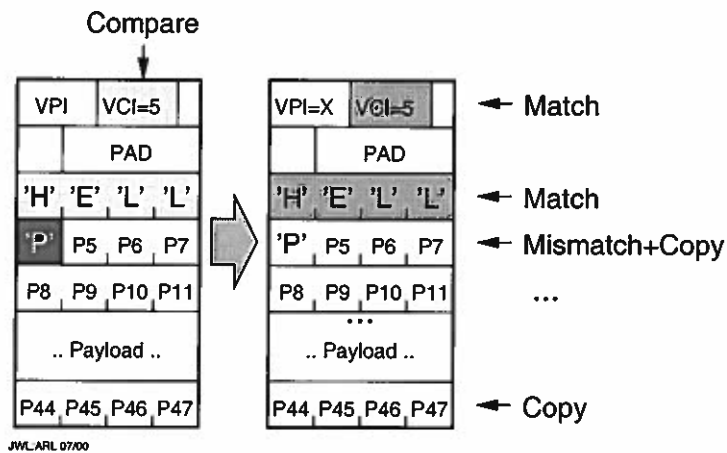
Figure 4: Cell processing for mismatched payload (2)

Performing a string match on the FPX is slightly complicated by the fact that the payload arrives as a stream of words; not all at once. Since an FPX module receives only one word per clock cycle, the circuit must know the status of previous comparisons to ensure that all current and previous words matched before it writes the word "WORLD." in the current and future clock cycles.

3 Logical Implementation
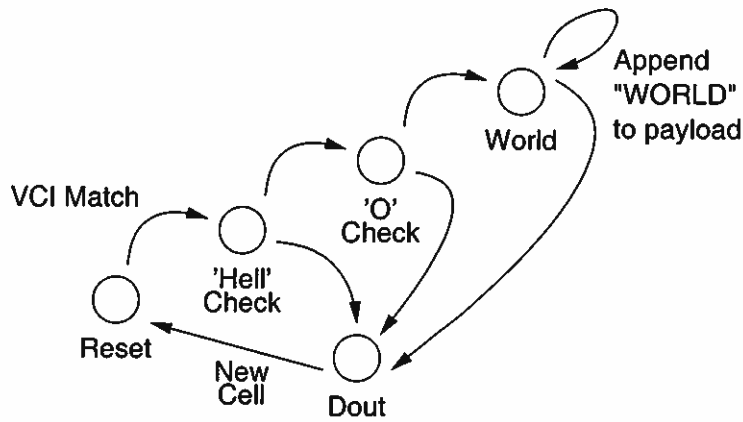
Figure 5: State machine for the Hello World circuit

There are several ways to implement the "HELLO WORLD." circuit on the FPX. One such implementation uses a word counter and the state machine shown in Figure 5.

The system begins in the reset state. When a new cell arrives, it compares the value of the VCI. If the VCI doesn't match, it jumps to the 'Dout' state. In the 'Dout' state, all of the remaining data in the cell is simply written out with the same value they had when it arrived. If the VCI matches, the circuit next scans the first word of the payload for the letters "HELL". If the string doesn't match, the state machine jumps to 'Dout' If the cell still matches, the state machine next checks the contents of second payload word for the letter "O". If that letter doesn't match, the state machine again jumps to 'Dout'.

For cells that do match, the state machine jumps to the 'World' state. It stays here for multiple clock cycles as the "WORLD" string is written to the payload.

5

4 Simplified RAD Entity

Simplified RAD Module

Figure 6: Simplified RAD Entity

The most simple configuration for a RAD module is shown in Figure 6. As with all RAD modules, the circuit operates at the 100 MHz frequency of RAD_Clock. RAD_Reset is asserted, active high, synchronously with RAD_Clock in advance of data arrivals.
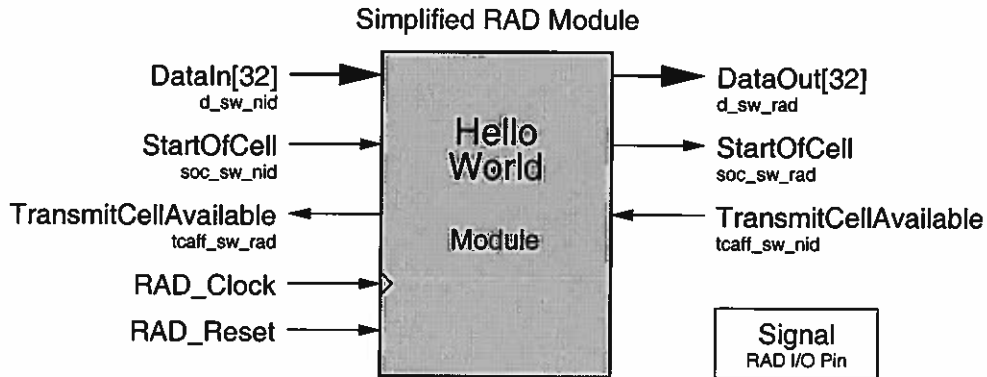
Data arrives as cells on a 32-bit data bus, DataIn[32]. Using the switch-side of the RAD logic, this bus corresponds to the "d_sw_nid" I/O pins.

The arrival of a new cell on the bus is indicated by the StartOfCell (SOC) signal. This signal goes high to indicate that the bus contains the first word of the cell.

The TransmitCellAvailable signal (TCA) is used for flow control. A module can block the arrival of a new cell by asserting this signal no less than 4 cycles before the end of the previous cell.

Data leaves the module on the DataOut[32] bus. In general, a module can add, modify, delete, or delay cells. The module simply asserts SOC when it has a new cell ready to transmit. Modules must defer the transmission of cells if the outgoing interface is congested, as indicated by downstream TCA.

Since the "Hello world" application never adds cells or delays cells by more than a few clock cycles, it never creates congestion. "Hello world", therefore, can can simply map the outgoing TCA indicator to the incoming interface.

6

5 VHDL Source Code

```
-- Hello World: Sample FPX Application
-- Operates as Ingress (switch-side) cell processor of RAD
-- Copyright: July 2000, John Lockwood, David Lim
-- Washington University, Applied Research Lab

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity HelloWorld is
        port (rad_clk: in std_logic; -- 100 MHz RAD Clock
                rad_reset: in std_logic; -- Asserted low at startup
                soc_sw_nid: in std_logic; -- Start of Cell [active high]
                tcaff_sw_nid: in std_logic; -- Transmit Cell Available
                d_sw_nid: in std_logic_vector (31 downto 0); -- Data In
                soc_sw_rad: out std_logic; -- Outgoing Start of Cell
                tcaff_sw_rad: out std_logic; -- [pass through]
                d_sw_rad: out std_logic_vector (31 downto 0) -- Data Out
        );
end HelloWorld;


architecture Hello_arch of HelloWorld is
        type state_type is (rst, dout, hell_check, o_check, world);

        -- "rst": reset state;
        -- "dout": output of the circuit equal to the input of the circuit;
        -- "hell_check": checks for the letters "HELL" in the incoming word;
        -- "o_check": checks for the letter "O";
        -- "world": circuit writes out the word "WORLD".

        signal state, nx_state : state_type;
        signal counter, nx_counter : std_logic_vector (3 downto 0);
        signal CEN, nx_CEN : std_logic;

        -- buffer signals to meet timing:
        signal BData_Out : std_logic_vector (31 downto 0);
        signal BData_in  : std_logic_vector (31 downto 0);
        signal BSOC_In   : std_logic;
        signal BTCA_In   : std_logic;
        signal BSOC_Out  : std_logic;
        signal BTCA_Out  : std_logic;
        signal clkin     : std_logic;

begin
   counter_process: process (CEN, counter) begin
           if CEN = '0' then
                   nx_counter <= "0001";
           else
```

7

```vhdl
                    nx_counter <= unsigned (counter) + 1;
            end if;
end process;

-- State Transitions
state_machine_process: process (BSOC_In, state, counter, BData_In, rad_reset, CEN)
begin
        if ( rad_reset = '1' ) then
                    nx_state <= rst;
                nx_CEN <= '0';
        elsif ( BSOC_In = '1' and
                BData_In(19 downto 4) = "0000000000000101" ) then
                -- checks to see if VCI = 5, if so: next check payload
                nx_state <= hell_check;
                nx_CEN <= '1';
        elsif ( BSOC_In = '1' and
                BData_In(19 downto 4) /= "0000000000000101" ) then
                -- VCI != 5
                nx_state <= dout; nx_CEN <= '1';
        elsif ( state = hell_check and counter = "0010" and
                BData_In="01001000010001010100110001001100" ) then
                -- checks to see if first payload word has letters "HELL"
                nx_state <= o_check;
                nx_CEN <= '1';
        elsif ( state = hell_check and counter = "0010" ) then
                -- Payload[0] != "HELL"
                nx_state <= dout;
                nx_CEN <= '1';
        elsif ( state = o_check and counter = "0011" and
                BData_In(31 downto 24) = "01001111" ) then
                -- checks to see if second payload word has the letter "O"
                nx_state <= world;
                nx_CEN <= '1';
        elsif ( state = o_check and counter = "0011" ) then
                -- Payload[1] != "O*"
                nx_state <= dout;
                nx_CEN <= '1';
        elsif ( state = world and counter = "0100" ) then
                nx_state <= dout;
                -- Output rest of payload, unchanged.
                nx_CEN <= '1';
        elsif ( state = dout and counter = "1100" ) then
                nx_state <= rst;
                -- Start over for next cell
                nx_CEN <= '0';
        elsif ( state = dout or state = hell_check or state = rst ) then
                nx_state <= state;
                -- same state
                nx_CEN <= CEN;
        else
                nx_state <= state;
                nx_CEN <= 'X';
        end if;
end process;
```

8

```
-- Upper 16-bits of Data Output
DataOut_31downto16_process: process (clkin) begin
    if clkin'event and clkin = '1' then
            -- checks to see if the intput data has the letter "O"...
            if ( state = o_check and BData_In(31 downto 24) ="01001111" ) then
            -- writes out "O " for the higher two bytes of the output
                    BData_Out(31 downto 16) <= "0100111101011111"; -- ("O ")
            elsif ( state = world and counter = "0100" ) then
                    BData_Out(31 downto 16) <= "0101001001001100"; -- ("RL")
            elsif ( state = rst and BSOC_In /= '1' ) then
                    BData_Out(31 downto 16) <= "0000000000000000";
            elsif ( state = dout or state=hell_check or BSOC_In = '1' ) then
                    BData_Out(31 downto 16) <= BData_In(31 downto 16);
            else
                    BData_Out(31 downto 16) <= "XXXXXXXXXXXXXXXX";
            end if;
    end if;
end process;

-- Lower 16-bits of Data Output
Data_Out_15downto0_process: process (clkin) begin
    if clkin'event and clkin = '1' then
            -- checks to see if the input data has the letter "O"...
            if ( state = o_check and BData_In(31 downto 24) = "01001111" ) then
            -- writes out "WO" for the lower two bytes of the output
                    BData_Out(15 downto 0) <= "0101011101001111"; -- ("WO")
            elsif ( state = world and counter = "0100" ) then
                    BData_Out(15 downto 0) <= "0100010000101110"; -- ("D.")
            elsif ( state = rst and BSOC_In /= '1' ) then
                    BData_Out(15 downto 0) <= "0000000000000000";
            elsif ( state = dout or state=hell_check or BSOC_In = '1' ) then
                    BData_Out(15 downto 0) <= BData_In(15 downto 0);
            else
                    BData_Out(15 downto 0) <= "XXXXXXXXXXXXXXXX";
            end if;
    end if;
end process;

BData_Out_process: process (clkin) begin
    -- buffer signal assignments:
    if clkin'event and clkin = '1' then
        d_sw_rad <= BData_Out; -- (Data_Out = d_sw_rad)
        BData_in <= d_sw_nid; -- (Data_In = d_sw_nid)
        BSOC_In <= soc_sw_nid; -- (SOC_In = soc_sw_nid)
        BSOC_Out <= BSOC_In;
        soc_sw_rad <= BSOC_Out;  -- (SOC_Out = tcaff_sw_rad)
        BTCA_In <= tcaff_sw_nid; -- (TCA_In = tcaff_sw_nid)
        BTCA_Out <= BTCA_In;
        tcaff_sw_rad <= BTCA_Out; -- (TCA_Out = tcaff_sw_rad)
        counter <= nx_counter; -- next state assignments
        state <= nx_state; -- next state assignments:
        CEN <= nx_cen;
    end if;
```

9

```vhdl
    end process;

    clkin <= rad_clk;

end Hello_arch;
```

6 RAD Ingress Module I/O Pin Mapping

The RAD has two interfaces: one interface typically used for data from the switch (egress), and the other typically used for data from the line card (ingress). Modules can be mapped to either interface. For this interface, the design is mapped to the switch (sw) side of the RAD.

On this interface, I/O pins of V1000E-FG680 device are mapped as follows:

```
##   File:  rad.ucf
##   Backend constraints file for RAD FPGA
##   Switch (SW) Side Module

## DataIn (Linecard interface, from NID)
NET d_sw_nid(0)   LOC=B31;
NET d_sw_nid(1)   LOC=C31;
NET d_sw_nid(2)   LOC=C32;
NET d_sw_nid(3)   LOC=D30;
NET d_sw_nid(4)   LOC=B33;
NET d_sw_nid(5)   LOC=D32;
NET d_sw_nid(6)   LOC=A31;
NET d_sw_nid(7)   LOC=D31;
NET d_sw_nid(8)   LOC=A33;
NET d_sw_nid(9)   LOC=C34;
NET d_sw_nid(10)  LOC=A34;
NET d_sw_nid(11)  LOC=D34;
NET d_sw_nid(12)  LOC=B32;
NET d_sw_nid(13)  LOC=B36;
NET d_sw_nid(14)  LOC=A35;
NET d_sw_nid(15)  LOC=D35;
NET d_sw_nid(16)  LOC=B37;
NET d_sw_nid(17)  LOC=D33;
NET d_sw_nid(18)  LOC=A36;
NET d_sw_nid(19)  LOC=B34;
NET d_sw_nid(20)  LOC=B35;
NET d_sw_nid(21)  LOC=D37;
NET d_sw_nid(22)  LOC=C33;
NET d_sw_nid(23)  LOC=F37;
NET d_sw_nid(24)  LOC=G37;
NET d_sw_nid(25)  LOC=C35;
NET d_sw_nid(26)  LOC=F36;
NET d_sw_nid(27)  LOC=E38;
NET d_sw_nid(28)  LOC=E37;
NET d_sw_nid(29)  LOC=G36;
NET d_sw_nid(30)  LOC=D38;
NET d_sw_nid(31)  LOC=C38;

## DataOut (Linecard interface, from RAD)
NET d_sw_rad(0)   LOC=B20;
NET d_sw_rad(1)   LOC=B21;
NET d_sw_rad(2)   LOC=E22;
NET d_sw_rad(3)   LOC=A21;
```

```
NET d_sw_rad(4)   LOC=D22;
NET d_sw_rad(5)   LOC=C22;
NET d_sw_rad(6)   LOC=D23;
NET d_sw_rad(7)   LOC=A22;
NET d_sw_rad(8)   LOC=B22;
NET d_sw_rad(9)   LOC=E23;
NET d_sw_rad(10)  LOC=B23;
NET d_sw_rad(11)  LOC=A23;
NET d_sw_rad(12)  LOC=C23;
NET d_sw_rad(13)  LOC=A24;
NET d_sw_rad(14)  LOC=C24;
NET d_sw_rad(15)  LOC=B24;
NET d_sw_rad(16)  LOC=A25;
NET d_sw_rad(17)  LOC=D26;
NET d_sw_rad(18)  LOC=B25;
NET d_sw_rad(19)  LOC=D25;
NET d_sw_rad(20)  LOC=D24;
NET d_sw_rad(21)  LOC=C26;
NET d_sw_rad(22)  LOC=C28;
NET d_sw_rad(23)  LOC=C25;
NET d_sw_rad(24)  LOC=B27;
NET d_sw_rad(25)  LOC=A27;
NET d_sw_rad(26)  LOC=C27;
NET d_sw_rad(27)  LOC=A29;
NET d_sw_rad(28)  LOC=B29;
NET d_sw_rad(29)  LOC=A28;
NET d_sw_rad(30)  LOC=B28;
NET d_sw_rad(31)  LOC=A26;

## Start of Cell
NET soc_sw_rad  LOC=D27;
NET soc_sw_nid  LOC=A32;
## TCA
NET tcaff_sw_nid  LOC=B26;
NET tcaff_sw_rad  LOC=D39;

## clock
NET rad_clk  LOC=AW19;
## Reset
NET rad_reset  LOC=B30;
```
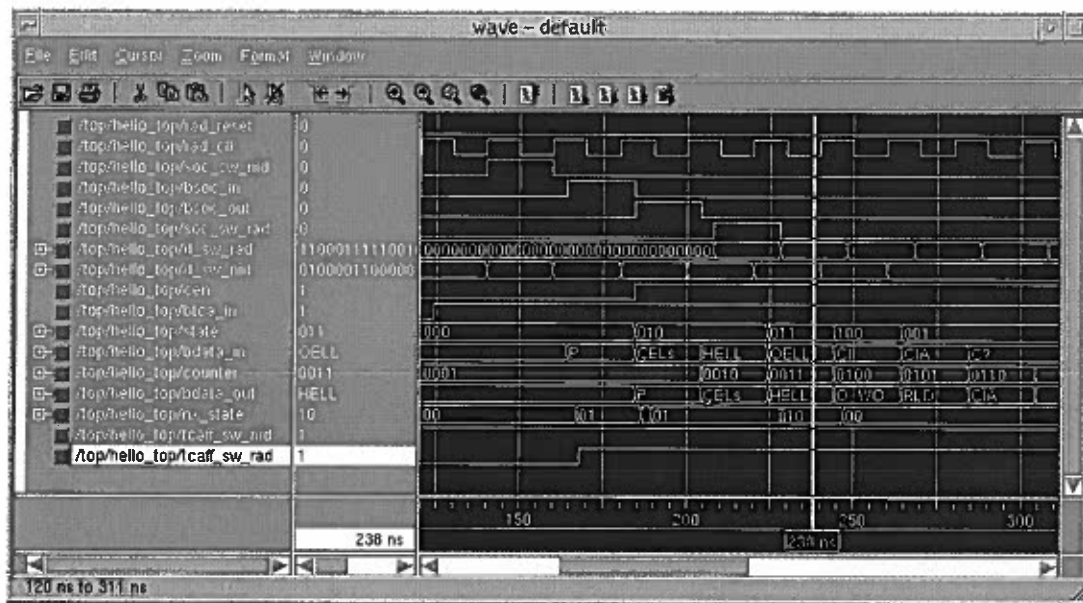
Figure 7: Post-synthesis signal timing relationships

7 Results of Physical Implementation

The circuit to implement the "Hello World" module was synthesized for RAD FPGA (a Xilinx V1000E-FG680-7 device) using Exemplar and Xilinx back-end synthesis tools. The resulting circuit operates at: 119 MHz. The 8.4ns critical path in this circuit is well within the 10ns period provided by the RAD's clock. Since the circuit can handle back-to-back cells; this circuit achieves the maximum packet processing rate of (100 MHz)/(14 Clocks/Cell)=7.1 Million packets per second.

Other statistics for this the circuit are listed below:

- Chip utilization: 1% (49/12,288 slices)
- Edge Flops: DataIn + DataOut + SOCs + TCAs = 32+32+4 = 68
- Internal Flops : BufferedData + SOCs + TCAs + state + counter = 32+4+6=42
- IOBs: DataIn + DataOut + SOCs + TCAs + Reset = 69
- GCLKS: RAD-CLK = 1

The post-synthesis timing relationships among the signals in the "Hello World" are shown in Figure 7. Note that the signals, like 'Start of Cell (SOC)' and data, are buffered both internally and across the edge flops of the FPGA. Observe how the buffered data output (bdata_out) are modified from the buffered data input (bdata_in). The payload of the cell was transformed from "HELLOELLO.." to "HELLO WORLD.".

8 Exercises

- The "Hello World" circuit could have been implemented with fewer states. Determine the minimum number of states needed to implement the "Hello World" circuit.
- Identify the states for the circuit described above.
- Modify the VHDL, and re-synthesize the "Hello World" circuit using your optimized design.

13

- Remove any critical paths in your circuit that are longer than 10 nanoseconds.
- Implement a circuit which performs the matching algorithm over multiple cells. Use AAL5 to encapsulate a frame.
- Consider how an FPX could be used with an SPC to implement hybrid hardware and software packet processing functions.

9   Conclusions

The FPX provides a simple and efficient platform for the implementation of certain types of cell and packet processing applications. The "Hello World." application detailed here is a complete and working example of a simple hardware module implemented on the the RAD.

10   References

Additional Information about the FPX is available on-line:

```
http://www.arl.wustl.edu/arl/projects/fpx/
```