Introduction

• Customers using the EFLX compiler for the EFLX arrays have encountered the following with their RTL:
  ▪ Their RTL compiles without any changes or optimizations required
  ▪ Their RTL needs to have changes made because it doesn’t compile
  ▪ Their RTL needs to have changes made because of either the array resources required are too high or the performance is too low

• This application note addresses the last 2 issues
  ▪ Things to avoid in RTL in order to compile
  ▪ Optimizations in RTL for better utilization and/or performance
EFLX Core Architecture Overview

**Logic RBB**
- 4 dual 4-input LUTs
- 8 Flip Flops
- Carry chain
- Selectable clocks

**DSP RBB**
- MAC
- 22-bit pre adder
- 22x22 multiplier
- 48-bit post adder (accumulator)

**I/O RBB**
- 2 inputs, 2 outputs
- Inputs and outputs can be pass-through or flopped
Verilog Coding for Embedded FPGAs Summary

• Things to avoid in RTL in order to compile
  ▪ Avoid using latches. Use D flip flops whenever possible
  ▪ Four State values not supported
    ▪ ‘0’, ‘1’, ‘x’, ‘z’
    ▪ Only ‘0’ and ‘1’ supported
  ▪ Inout data type or tristate buffers not supported

• Optimizations in RTL for better utilization and/or performance
  ▪ Maintain Synchronous Sub-Blocks by Registering All Outputs
  ▪ Use registered inputs and outputs to the embedded FPGA fabric
  ▪ Use asynchronous resets instead of synchronous
  ▪ Use clock enables instead of multiple clocks
  ▪ Use 1-hot encoding instead of binary encoding for state machines
  ▪ Implement proper synchronization of all asynchronous signals
  ▪ Optimize Code for Embedded FPGA fabric
Things to avoid in RTL in order to compile
Inadvertent Latch Inference via incomplete combinatorial process

- In **Verilog**, if all conditions are not specified, a latch will be inferred.
- Latches are not supported in the embedded FPGA fabric and will yield unpredictable results.

```verilog
module IF_MUX (Sel, A,B,C, D, Z_out );
input [1:0] Sel ;
input A, B, C, D;
output Z_out ;
reg Z_out ;

always @ (A or B or C or D or Sel )
begin
    if ( Sel == 2'b00) Z_out = A ;
    else if ( Sel == 2'b01) Z_out = B ;
    else if ( Sel == 2'b10) Z_out = C ;
    else Z_out = D ;
end
endmodule

module CASE_MUX (Sel, A,B,C, D, Z_out );
input [1:0] Sel ;
input A, B, C, D;
output Z_out ;
reg Z_out ;

always @ (A or B or C or D or Sel )
begin
    case ( Sel )
        2'b00: Z_out = A ;
        2'b01: Z_out = B ;
        2'b10: Z_out = C ;
        2'b11: Z_out = D ;
    endcase
end
endmodule
```
Not supported language constructs

- Four State values within the FPGA fabric not supported (i.e. ‘0’, ‘1’, ‘x’, ‘z’)
  - Only ‘0’ and ‘1’ supported in the fabric
- Bidirectional buffers need to be generated outside the FPGA fabric
  - Avoid using inout datatype which will incur bidirectional IOBUF instances that are not supported in the fabric
- Muxes used instead of Hi-Z buffers

No Hi-Z Buffers

---

FPGA Fabric

Mux

Bidirectional Buffer

I/O Pads

OEn

Dout

Din

Sel

Out

In_A

0

1

In_B

Out

In_A

In_B

Sel

flexlogix Technologies, Inc.
Don’t Use IOBUF or BUFT

The EFLX compiler will treat BUFT as a regular buffer ignoring OE
Optimizations in RTL for better utilization and/or performance
Maintain Synchronous Sub-Blocks by Registering All Outputs

- Arrange the design boundary so that the outputs in each block are registered.
- Registering outputs helps the synthesis tool implement the combinatorial logic and registers in the same logic block.
- Registering outputs also makes the application of timing constraints easier since it eliminates possible problems with logic optimization across design boundaries.
- Using a single clock for each synchronous block significantly reduces the timing consideration in the block.

Synchronous Blocks with Registered Outputs

Merging Sharable Resource in the Same Block
Use Registered Inputs/Outputs

- Use Registered inputs and Outputs
- Registered inputs and outputs decouples the FPGA fabric timing from the SOC
- The Registers have to be in the IO RBB

Always @(posedge clk)
Din_i <= Din;
Or
Always @(posedge clk)
If (enable_input) Din_i <= Din;

Or
Always @(posedge clk)
Dout <= Dout_i;
Or
Always @(posedge clk)
If (enable_output) Dout <= Dout_i;
Asynchronous resets

- EFLX has asynchronous Chip Reset to every FF, so there is no need to include chip reset to your RTL.
  - Unless the FF is required to set to “1” during chip-reset, then please keep them in the RTL for correctness.

```verilog
always @(posedge clock or posedge CHIP_RST)
begin
  if (CHIP_RST)
    COUNTER <= 5'b00000;
  else
    COUNTER <= COUNTER + 1;
end

initial COUNTER = 5'b0 //reset value for simulations
initial STATE_MACHINE <= 5'b00000; // IDLE_STATE
always @(posedge clock)
begin
  COUNTER <= COUNTER + 1;
end

OR
Simple assign “CHIP_RST” to 0, and Synplify should optimize away the reset logic

assign CHIP_RST = 1'b0;
```
Asynchronous resets

- If additional resets are needed in your RTL, use Asynchronous reset instead of synchronous resets whenever possible
  - Synchronous resets with clock enable are implemented via the SW compiler due to HW limitation: FF assumes synchronous reset (SR) have priority over clock-enable (CE), which does not work with the clock gating cells used in EFLX core
  - SW compiler inserts an extra LUT to “OR” CE and SR together to achieve functionality
  - This applies to all EFLX cores, Synchronous resets without clock enable does not have this limitation
- DSP RBBs and legacy (2015) RBBs natively do not support synchronous reset - it is realized using synchronizer logic
  - Synchronous set and reset flip-flop (reset has priority) is implemented using additional logic inserted by SW compiler.

![Synchronous Reset + Clock Enable](image1)

![Compiler Fix](image2)

![Synchronous Reset for RBBs without synchronous reset](image3)

![Compiler Fix](image4)
Use clock enables instead of multiple clocks

- Avoid using gated, derived, or divided clocks
- Use clock enables instead of multiple clocks
- Multiple clock domains cause higher LUT/FF utilization

**Example: dout clocked out at clk divided by 256**

### Derived Clock, 2 clock domains

```verilog
module derived_clock (clk, din, dout);

input clk;
input [7:0] din;
output [7:0] dout;

reg [7:0] temp;
reg [7:0] dout;

always @(posedge clk)
    temp <= temp + 1;

always @(negedge temp[7])
    dout <= din;

endmodule
```

- 11 LUTs
- 25 FFs

### Using Clock Enable, 1 clock domain

```verilog
module derived_clock (clk, din, dout);

input clk;
input [7:0] din;
output [7:0] dout;

reg [7:0] temp;
reg [7:0] dout;

always @(posedge clk)
    temp <= temp + 1;

always @(posedge clk)
    if (temp == 8'hff) dout <= din;

endmodule
```

- 9 LUTs
- 23 FFs

18% LUT reduction
Use 1-Hot Encoding

1-hot encoding takes less EFLX array resources than using binary encoding, given the same compiler options.

### Resource comparison for an I²C Controller

<table>
<thead>
<tr>
<th></th>
<th>Binary state &amp; cmd encoding</th>
<th>1-hot state &amp; cmd encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic RBBs</td>
<td>94</td>
<td>88</td>
</tr>
<tr>
<td>RBB LUTs</td>
<td>256</td>
<td>244</td>
</tr>
<tr>
<td>Total LUTs</td>
<td>284</td>
<td>263</td>
</tr>
<tr>
<td>RBB FFs</td>
<td>155</td>
<td>153</td>
</tr>
<tr>
<td>Total FFs</td>
<td>162</td>
<td>161</td>
</tr>
<tr>
<td>IO RBBs</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Number of Tiles Needed</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Floorplan spec</td>
<td>2x2</td>
<td>3x1</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>31.90</td>
<td>31.97</td>
</tr>
</tbody>
</table>

### Binary Encoding

```c
// statemachine
parameter [2:0] ST_IDLE = 3'b000;
parameter [2:0] ST_START = 3'b011;
parameter [2:0] ST_READ = 3'b101;
parameter [2:0] ST_WRITE = 3'b111;
parameter [2:0] ST_ACK = 3'b100;
parameter [2:0] ST_STOP = 3'b101;
```

### 1-Hot Encoding

```c
// statemachine
parameter [5:0] ST_IDLE = 6'b00 0000;
parameter [5:0] ST_START = 6'b00 0110;
parameter [5:0] ST_READ = 6'b00 0010;
parameter [5:0] ST_WRITE = 6'b00 1000;
parameter [5:0] ST_ACK = 6'b01 0000;
parameter [5:0] ST_STOP = 6'b10 0000;
```

~7.5% less LUTs & 1 less tile
Implement proper synchronization of all asynchronous signals

- Asynchronous signals need to be synchronized before used by FPGA fabric
- Double flop async signals for demetastabilization
- EFLX compiler will place demetastable flops close together

Asynchronous Domain Crossing

Demetastabilization Flip Flops

Always @ (posedge CLK)
Dout' <= Din;
Dout <= Dout';
Optimizing RTL Code

Two methods of optimizing RTL to take advantage of EFLX fabric

1. Rewriting RTL code to fit into fabric in optimal way

2. Using the reconfigurability of the EFLX fabric to minimize configuration/mode registers and functional modes
Optimizing code example

Non Optimized Serial-to-Parallel Code

```
if (rx_cnt > 0 && rx_cnt < 9)
begin
rx_reg[rx_cnt-1] <= rx_d2;
```

- 1 mux per D-input
- 8 muxes total
- 8 FFs
- 6 RBB LUTs

Optimized Serial-to-Parallel Code

```
if (rx_cnt > 0 && rx_cnt < 9)
begin
    rx_reg[7] <= rx_d2;
    rx_reg[6] <= rx_reg[7];
    rx_reg[5] <= rx_reg[6];
    rx_reg[4] <= rx_reg[5];
    rx_reg[3] <= rx_reg[4];
    rx_reg[2] <= rx_reg[3];
    rx_reg[1] <= rx_reg[2];
    rx_reg[0] <= rx_reg[1];
end
```

- 0 muxes
- 8 FFs
- 4 RBB LUTs

33% LUT reduction

Simpler Logic uses less LUTs
32-bit FIR Filter RTL Code - Non Optimized


- Utilizes 10 DSP RBBs
- Plus additional 284 Logic RBBs
- Additional Logic RBBs required due to 32bit*16bit multiplier not fitting into 22bit*22bit DSP MAC
32-Bit FIR Filter RTL Code- Optimized

Optimizations of the RTL code to better fit into the EFLX DSP MAC structure (i.e. 22bit*22bit=48bit)

- Utilizes 10 DSP RBBs and only 8 Logic RBBs

- Splitting the 32 bit data into upper word and lower word
- Shifting the upper word accumulator left by 16 and adding to lower word accumulator
- Result is 48-bits
## EFLX Utilization

<table>
<thead>
<tr>
<th></th>
<th>32-bit FIR Non Optimized</th>
<th>32-bit FIR Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP RBBs used</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Logic RBB used</td>
<td>284</td>
<td>8</td>
</tr>
<tr>
<td>LUTs used</td>
<td>192</td>
<td>32</td>
</tr>
<tr>
<td>RBB FF used</td>
<td>493</td>
<td>48</td>
</tr>
<tr>
<td>DSP tiles used</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Logic tiles- used</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Total tiles used</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

2.2x reduction of total tiles required
FIR Filter with programmable Coefficient Registers

Programmable Coefficient Registers take additional LUTs
FIR Filter with Coefficients Configured in the EFLX Fabric

- Use Programmability of FPGA for Coefficient values
- Coefficients as assign statements

 Registers Not Required
Example RTL: Before vs. After FIR Coefficient Optimization

```verilog
reg [15:0] CI_0, CI_1, CI_2, CI_3, CI_4;

always @ (posedge clk_i)
begin
  if (rst_i)
  begin
    CI_0 <= 16'd0;
    CI_1 <= 16'd0;
    CI_2 <= 16'd0;
    CI_3 <= 16'd0;
    CI_4 <= 16'd0;
  end
  else if (we_i)
  begin
    switch (adr_i[2:0])
    begin
      case 0: CI_0 <= dat_i;
      case 1: CI_1 <= dat_i;
      case 2: CI_2 <= dat_i;
      case 3: CI_3 <= dat_i;
      case 4: CI_4 <= dat_i;
    end
  end
end

wire [15:0] CI_0, CI_1, CI_2, CI_3, CI_4;
// hard-wire the filter coefficients
assign CI_0 = 16'd2;
assign CI_1 = 16'd13;
assign CI_2 = 16'd542;
assign CI_3 = 16'd985;
assign CI_4 = 16'd8712;
```
## LUT Savings

<table>
<thead>
<tr>
<th></th>
<th>16-bit FIR with programmable Coefficient Registers</th>
<th>16-bit FIR with Coefficients configured in the fabric</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP RBBs used</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Logic RBB used</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>RBB LUTs used</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>RBB FF used</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>DSP tiles used</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Total Available RBB LUTs</td>
<td>264</td>
<td>264</td>
</tr>
</tbody>
</table>

| LUT Utilization        | 20%                                              | 0%                                                |

20% LUT utilization reduction