Resets in FPGA & ASIC control and data paths

Mufaddal Saifee, Jaymin Patel - August 31, 2016

Reset is an important mechanism to bring a digital system into a known state. The need for reset is governed by the system design and application, and various data and control paths are designed to use a reset signal. Flip-flops in the control path should have reset parameters to bring the system to a known state, while one can usually do without reset in the data path. Let’s discuss various use cases of resets in ASICs and FPGAs.

Introduction

Various data path systems are designed to synchronize with inputs, process them, and provide output. For such designs, if all the unused states are waiting for acquiring the sync state, then the design can do without ever being reset. For designs where the state machine logic has been optimized during synthesis by avoiding logic reduction, resets are a must to prevent the design from starting in a random state.

There are various scenarios apart from the device reboot where the resets are required in real-world designs. Some likely scenarios include:

- If a clock source drops out, then the design should enter the reset until PLL lock is acquired.
- Communication channel must be reset, when the link partner loses sync.
- A user-accessible push button reset.
- Reset from microprocessor after some event, example after expiry of a watch dog timer.
- Partial reconfiguration in modern FPGAs where the swapped logic block must be reset independently of its environment.
- Use of legacy IP from previous ASICs or an IP vendor requiring reset.
- Reset to start clean simulations.

To achieve better performance, designs nowadays have their data pipelined through chains of Flip-Flops. For such designs, the usage of reset can be eliminated in the pipelined Flops to achieve better area utilization and performance. Choosing reset usage and its strategies in ASIC/FPGA designs require many design considerations like whether every Flip-Flop will require a reset, whether to use a synchronous reset or asynchronous reset, how the reset tree will be structured and buffered, how to verify timing of reset tree, how test scan vectors are applied to check functionality of reset and how reset is handled in multi clock domain designs. In this article, we will see the reset usage in Data and Control path in ASIC and FPGA. We will also see how FPGA components like shift
registers, Block RAM, Distributed RAM, and DSP Slices can be used efficiently with proper use of reset.

**Reset usage in Data Path & Control Path in ASIC**

A common belief with most designers is that all the flops in the design should be having a reset. This is not true for data paths. Flip-Flops in control path do need a reset but Flip-Flops in data path can go without it. All these pipelining Flip-Flops or delay Flip-Flops don’t need resets. Moreover shift registers, data bus etc. which depend on some resets, to indicate valid data at any time can do without a reset signals.

For a design having a pipelined or delay chain, Flip-Flops with synchronous resets as shown in below Figure 1 can be implemented with a single resettable input Flip-Flop as shown in Figure 2. The only thing to be taken care of by the design shown in Figure 2 is that it should held the synchronous reset asserted for duration long enough to flush the entire pipeline.

![Figure 1: Pipeline Follower Flip-Flops with synchronous reset signal in data path](image1)

For the designs where the validity of data is indicated by a valid signal, we can remove reset signal in data-bus path and only provide reset to the valid signal generating Flip-Flop as shown in Figure 3.

![Figure 2: Pipeline Follower Flip-Flops without synchronous reset signal in data path](image2)

As we can see that the reset signal is not present in the data path and valid data is latched whenever the load signal is asserted. Once, data is captured on load signal, then it can be shifted freely in the pipeline. The control signal is also delayed with data path to synchronize the last stage Flip-Flop output.
Eliminating the need for reset signal in data path will reduce the total number of resettable Flip-Flops required in design and thus, reduce the overall number of gate counts. The benefits of using Flip-Flops without reset are,

- Reduced logic utilization
- Lesser routing challenge
- Thus lower routing run time
- Better timing performance.

The golden rule for minimal reset design is that all registers that receive state through feedback must be reset. Generally reset is needed in following conditions.

- Flip-Flops within a feedback loop
- Flip-Flops holding the state of the state machine
- IIR filter Flip-Flops

**Coding Consideration for the pipeline Flip-Flops**

The designer should not mix the resettable and non resettable Flip-Flops in the same Verilog procedural block or VHDL process block.

The correct way to code pipeline Flip-Flops with two Verilog procedural blocks is shown below. These coding styles will generate the logic shown in Figure 4.

```verilog
module follower_FF (Clk,Rst,d,q1);

input Clk;

input Rst;

input d;
```
output q1;

reg q;
reg q1;

always @ (posedge Clk)
begin
    if(Rst)
        begin
            q <= 1'b0;
        end
    else
        begin
            q <= d;
        end
end

always @ (posedge Clk)
begin
    q1 <= q;
end
endmodule

Figure 4: Follower Flip-Flop without reset
As shown in below code, if both the Flip-Flops are modeled in the same procedural block than it would generate logic shown in Figure 5. The below coding style generates extra logic by using reset signal Rst as a data enable for the second Flop.

```verilog
module follower_FF (Clk,Rst,d,q1);

input Clk;
input Rst;
input d;
output q1;

reg q;
reg q1;

always @ (posedge Clk)
begin
  if(Rst)
    begin
      q <= 1'b0;
    end
  else
    begin
      q   <= d;
      q1 <= q;
    end
end
endmodule
Reset usage in Data Path & Control Path in FPGA

Implementation of resets in FPGA are associated with the corresponding vendors. Therefore, before implementing FPGA hardware with a kind of reset, it is recommended to carefully read the FPGA vendor’s datasheet and HDL coding guidelines. Some FPGA hard macros, like SRL, DSP Blocks, BRAM Blocks, and SERDES support only limited reset options, or none at all. Therefore, resetting such logic resources incorrectly, might prevent inferring dedicated hard macros and affect the utilization and performance by inferring LUT logic and Slices Flip-Flops in place. Hence, reset issues such as synchronous or asynchronous, active high or low should be decided on the basis of datasheets, and configurable logic capabilities of the corresponding FPGA.

Proper use of reset signal for these hard macros will improve performance, timing and lead to have better device utilization. Below section summarize the usage of reset for various hard macros in FPGA.

Active High reset

Xilinx FPGA Flip-Flop has active high control port SR for preset / clear / set / reset functionality as shown in below Figure 6. The SR port will be inferred as an asynchronous preset / clear or as a synchronous set / reset depending on the type of Flip-Flop, the RTL code describes.

Active low usage of preset / clear / set / reset functionality in RTL code will require an inverter before the Flip-Flop to drive the active high SR port. This inverter will be inferred in the LUT affecting the timing, power and device utilization.
If somehow we can not control the polarity of external signal and it still uses the active low polarity, then it is recommended to invert the active low reset signal in top level hierarchy and then route that signal to all other registers. This will infer the inverter into IO logic and no additional FPGA routing or logic will be used for remaining registers.

**Synchronous Reset**

SR control port of all the Flip-Flops within a slice is common. Therefore different types of Flip-Flops cannot be packed into the same slice. As shown in Figure 7, three Flip-Flops with different asynchronous control signal are used and thus they consume 3 different slices. But if the synchronous control signals were used, then the synthesis tool maps the synchronous control signals into LUT and won’t use control port of Flip-Flops. This will allow packing of different control signals Flip-Flops in the same slice. This may increase the LUT utilization but improves the slice utilization. Thus it’s better to use synchronous control signals in order to have better slice utilization.

![Control set optimization on SR](image)

**Figure 7 : Control set optimization on SR**

**SRL Component**

In Xilinx FPGA, there is a dedicated SRL component available which can be used to infer shift register functionality. SRL component uses SLICEM LUT logic to infer the shift register. If the code describes the shift register with synchronous or asynchronous reset then tool will use Flip-Flops for the shift register. While if shift register is coded without reset then the tool will infer the dedicated
SRLs within the slices. Thus for better device utilization it is advisable not to use resets for inferring the shift registers in the hardware.

Below is Verilog code written with reset which will infer Flip-Flops for shift register.

```verilog
always @ (posedge Clk or posedge Rst)
begin
    if(Rst)
        begin
            q <= 0;
        end
    else
        begin
            q <= {q[6:0],Din};
        end
end

assign q1 = q[7];
```

But if code is written without reset as shown below then dedicated SRL components will be inferred.

```verilog
always @ (posedge Clk)
begin
    q <= {q[6:0],Din};
end

assign q1 = q[7];
```
RAM

There are two types of RAM provided in Xilinx FPGA. One is dedicated Block RAM and second is distributed RAM inferred within CLB LUTs. An explicit reset cannot be used to reset the content of the RAMs. Thus while writing a code intended to infer the RAMs, one should code without reset.

The type of RAM inferred from the code depends on the type of read coded for the RAM. If the read is synchronous then BRAMs are inferred while if read is asynchronous then Distributed LUT RAMs are inferred.

The below coding style with asynchronous read will infer **Distributed RAM** Macro in RTL.

//Dual-Port RAM With Asynchronous Read Verilog Coding Example

```verilog
module rams_00 (clk, we, a, dpra, di, spo, dpo);
  input clk;
  input we;
  input [5:0] a;
  input [5:0] dpra;
  input [15:0] di;
  output [15:0] spo;
  output [15:0] dpo;
  reg [15:0] ram [63:0];
  always @(posedge clk)
    begin
      if(we)
        begin
          ram[a] <= di;
        end
    end
  assign spo = ram[a];
```

The below coding style with asynchronous read will infer **Distributed RAM** Macro in RTL.

//Dual-Port RAM With Asynchronous Read Verilog Coding Example

```verilog
module rams_00 (clk, we, a, dpra, di, spo, dpo);
  input clk;
  input we;
  input [5:0] a;
  input [5:0] dpra;
  input [15:0] di;
  output [15:0] spo;
  output [15:0] dpo;
  reg [15:0] ram [63:0];
  always @(posedge clk)
    begin
      if(we)
        begin
          ram[a] <= di;
        end
    end
  assign spo = ram[a];
```
assign dpo = ram[dpra];
endmodule

Below coding style with synchronous read will infer **Block RAM** Macro in RTL.

// Dual-Port RAM with Synchronous Read (Read Through)
module rams_01 (clk, we, a, dpra, di, spo, dpo)
input clk;
input we;
input [5:0] a;
input [5:0] dpra;
input [15:0] di;
output [15:0] spo;
output [15:0] dpo;

reg [15:0] ram [63:0];
reg [5:0] read_a;
reg [5:0] read_dpra;

always @(posedge clk)
begin
  if(we)
  begin
    ram[a] <= di;
    read_a <= a;
  end
end
read_dpra <= dpra;
end

assign spo = ram[read_a];
assign dpo = ram[read_dpra];
endmodule

Xilinx BRAM contains output register primitive at the data output. To utilize that register in BRAM, one must use the synchronous reset in BRAM. Otherwise it will not use that dedicated output register and use Flip-Flop from a CLB.

**DSP Slices**

DSP48 slices are available in Xilinx FPGA to accelerate DSP algorithms and solve complex DSP challenges. DSP48 Slices have pipeline registers both in control and data path to enhance the performance of the design. All these DSP48 internal registers have a synchronous reset. So if asynchronous reset is used in coding then the synthesis tool won’t be able to pack these into DSP48 block and would require the additional CLB registers to implement the asynchronous reset functionality. This not only leads to poor device utilization but also impacts performance and power. Therefore it is recommended to use synchronous reset to infer the internal registers of DSP48 slices for improved performance.

**Conclusions**

The Flip-Flops in the control path should have reset to bring the digital system to a known state, while one can do without reset for Flip-Flops in data path. As a rule, all the Flip-Flops that receive their state through feedback should have reset. However, Flip-Flops that are pipelined or follower flip-flops, or updated with some valid logic, do not need reset. Optimal usage of reset in designs leads to improved logic, timing, power, and routing.

Implementation of resets in an FPGA is dependent on the vendor. Hence, reset types or their usage for the hard macro should be decided on the basis of the corresponding FPGA vendor datasheet and their configurable logic capabilities. The recommendation and guidelines provided here will help designers choose the appropriate reset type and usage in their designs.

**REFERENCES**

[5] Xilinx WP275 (v1.0.1) October 22, 2007, Get your Priorities Right - Make your Design Up to 50% Smaller

Mufaddal A. Saifee works at eInfochips as Senior Engineer. Mufaddal has experience in ASIC and FPGA design and verification cycle. He has worked on complete FPGA system implementation flow covering micro-architecture design, RTL coding, gate level simulation, synthesis, PAR, timing closure and board validation. He has designed and implemented various Pipelined Application Specific Processors. He is well versed with different technologies, protocols, algorithm and their implementations. He has worked on variety of projects covering applications in domains like video processing, networking, DSP and TMR-Avionics.

Jaymin Patel works at eInfochips as Engineer. He has 4+ years of experience in FPGA design cycle. He has worked on complete FPGA system implementation flow including micro-architecture design, RTL coding, Verification, Gate level simulation, Synthesis, PAR, timing closure and board level validation. He has worked on various image processing algorithms and also worked on variety of projects covering video processing domain.