MODULE 7

FPGA Applications

CONTENTS

- ★ Embedded system design using FPGAs,
- ★ DSP using FPGAs,
- ★ Dynamic architecture using FPGAs, reconfigurable systems, application case studies.
- ★ Simulation / implementation exercises of combinational, sequential and DSP kernels on Xilinx / Altera boards.

Embedded system design with FPGAs

Embedded system design with FPGAs (Field-Programmable Gate Arrays) is a powerful approach to developing specialized, high-performance computing systems.

FPGAs are reconfigurable hardware devices that can be programmed to implement complex digital circuits. This makes them highly suitable for applications requiring flexibility, parallel processing, and real-time performance.

Advantages of Using FPGAs in Embedded Systems

- High Performance: FPGAs provide low-latency, high-speed processing, making them ideal for real-time applications.
- Customization: Developers can design custom hardware architectures tailored for specific tasks.
- Parallelism: FPGAs inherently support parallel processing, which can accelerate data processing, unlike traditional CPU-based systems.
- Low Power Consumption: FPGAs can be more energy-efficient, especially in applications where only parts of the system need to be active at any time.

Design Flow for Embedded Systems with FPGAs

- Specification
- System Design
- Hardware Design
- Software Development
- Simulation and Testing
- Synthesis and Implementation
- Verification

Applications of FPGAs in Embedded Systems

- Automotive
- Telecommunications
- Industrial Automation
- Consumer Electronics
- Medical Devices

FPGA vs. ASIC (Application-Specific Integrated Circuit)

- Reconfigurability
- Development Time
- Cost

FPGA Platforms

- Xilinx: Offers the widely used Vivado Design Suite and a range of FPGA devices.
- Intel (formerly Altera): Provides Quartus Prime software and a family of FPGAs.
- Lattice Semiconductor: Known for low-power, small-footprint FPGAs.
- Microsemi: Specializes in FPGAs for aerospace, defense, and industrial applications.

Digital Signal Processing with FPGA

Digital Signal Processing (DSP) with Field-Programmable Gate Arrays (FPGAs) combines the flexibility of digital signal processing with the speed and power efficiency of hardware acceleration.

Digital Signal Processing (DSP) involves manipulating signals like audio, images, and video to filter, compress, and analyze data.

Field-Programmable Gate Arrays (FPGAs) are integrated circuits that can be reconfigured to perform custom logic operations, making them highly suitable for real-time, high-speed DSP applications.

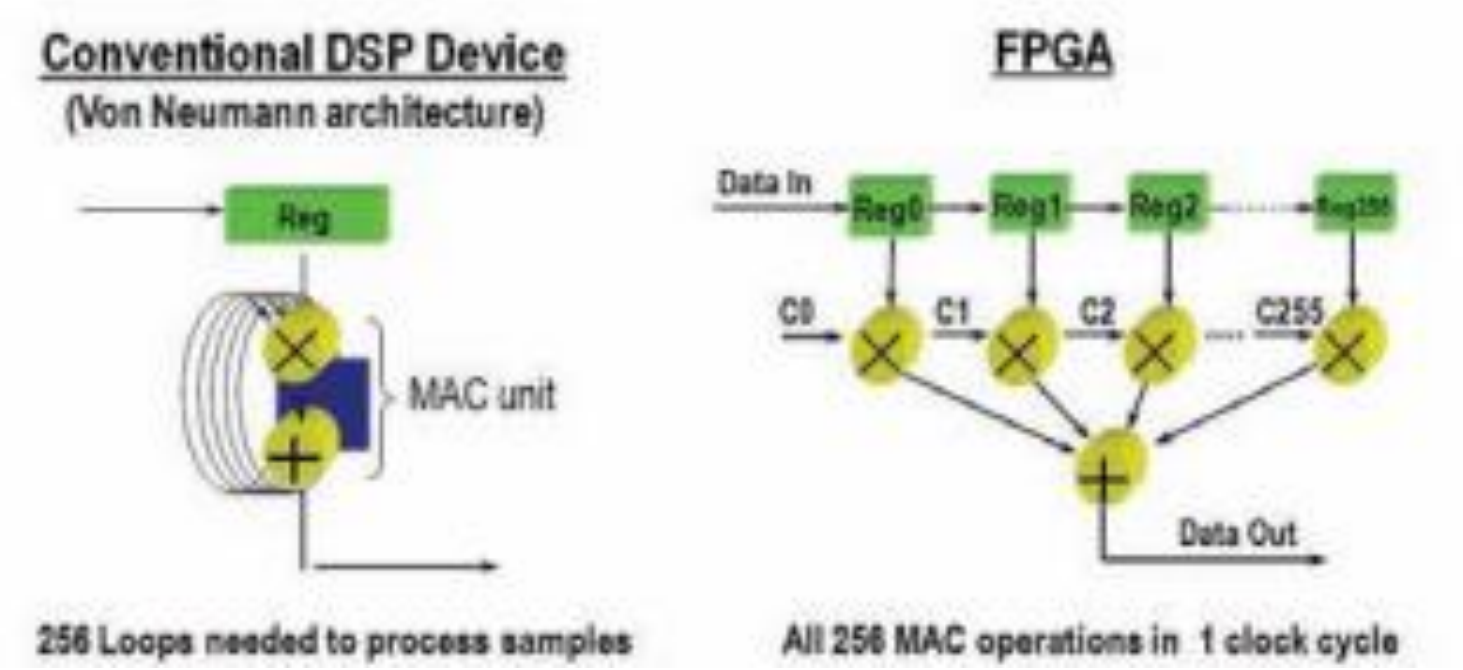
Why Use FPGAs for DSP

Parallel Processing: FPGAs can execute many tasks simultaneously, crucial for real-time DSP where high-throughput and low latency are needed.

Customizability: FPGAs allow specific optimizations for tasks, making them more power-efficient than general-purpose processors.

Reduced Latency: Directly handling data in hardware minimizes delays, essential for time-sensitive applications like communications and video processing.

FIR filter comparison between DSP and FPGA



Applications of DSP with FPGAs

Signal Filtering: High-speed filtering in applications like communication systems.

Image and Video Processing: Real-time video scaling, object detection, and more in areas like medical imaging and surveillance.

Wireless Communication: Processing signals in base stations for faster data rates and more efficient spectrum usage.

Key Concepts in DSP Implementation on FPGAs

Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) Filters: FPGAs can implement these filters with high precision and speed.

Fast Fourier Transform (FFT): FPGAs accelerate FFT calculations for applications in audio processing, radar, and wireless communications.

Pipelining and Parallelism: FPGAs allow these techniques, enhancing processing speed and efficiency.

Development Workflow

Design and Simulation: Using tools like MATLAB and Simulink for DSP algorithm development and simulation.

Synthesis and Implementation: FPGA tools (e.g., Xilinx Vivado, Intel Quartus) translate DSP algorithms into hardware code.

Verification: Testing DSP functions in hardware to ensure correct performance.

Challenges

Resource Constraints: Balancing performance requirements with FPGA resources like logic blocks and memory.

Complexity in Design: Designing DSP applications on FPGAs requires a strong grasp of both DSP algorithms and digital design principles.

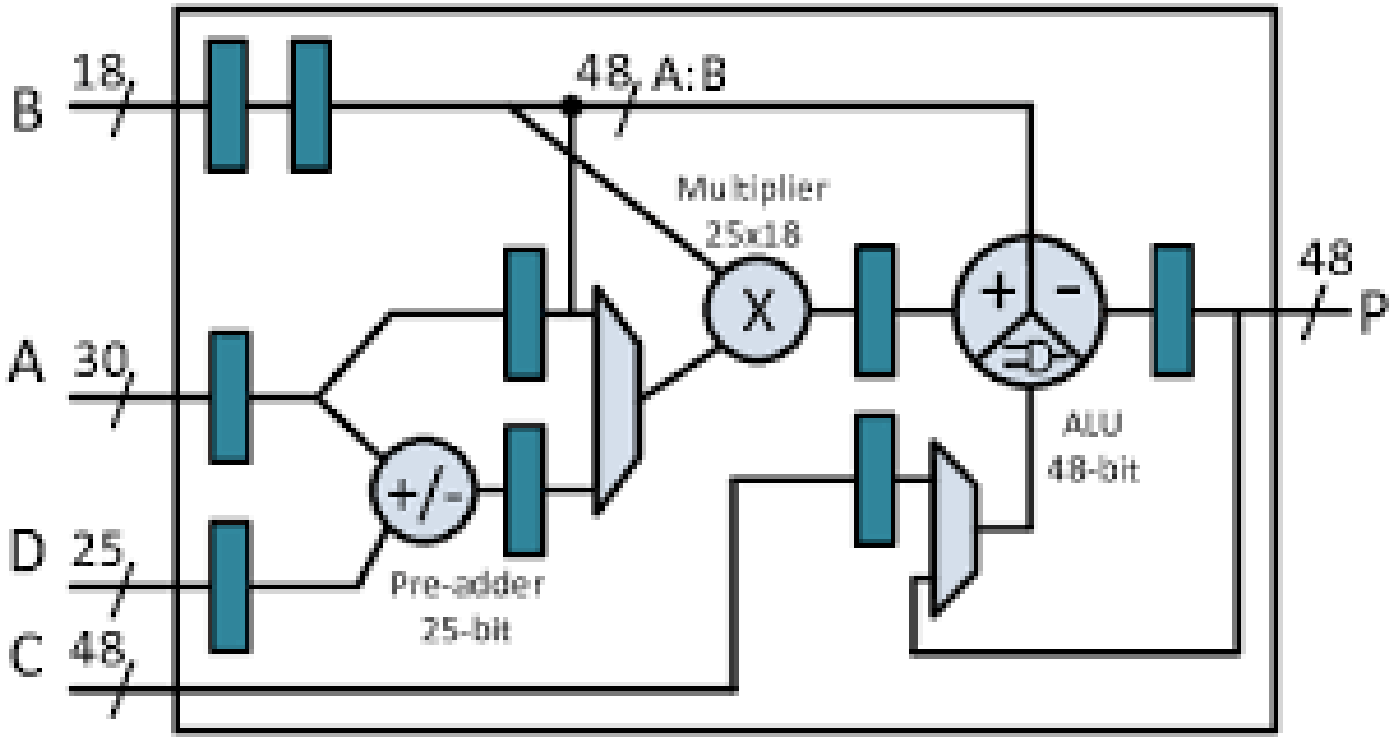
DSP BLOCKS IN FPGAS

- Digital signal processing (DSP) blocks in FPGAs are dedicated blocks that can perform basic arithmetic functions and accelerate digital signal processing applications. They can be used to implement tasks like fast Fourier transforms (FFTs) and finite impulse response filtering (FIR).

Features of DSP blocks in FPGAs

- Arithmetic functions: DSP blocks can perform basic arithmetic functions like addition, subtraction, and multiplication.
- Performance: DSP blocks can be used to build more efficient implementations that reduce the demand for area.
- Precision: DSP blocks on modern FPGAs can operate in different modes, such as standard precision, high precision, or floating-point mode.
- Variable precision: Variable precision DSP blocks can balance compute resources with precision.
- Input shift registers: DSP blocks contain input shift registers to implement digital filter applications.
- Multipliers: DSP blocks can implement up to eight 9 Ã— 9 multipliers, six 12 x 12 multipliers, four 18 Ã— 18 multipliers, or two 36 Ã— 36 multipliers.
- Tensors: Some DSP blocks include tensors, which are mathematical objects that represent multi-dimensional arrays of data.

DSP BLOCK IN XILINX 7 SERIES



content

Operating Modes

- The DSP block operates in the following operation modes:

Operation mode	Definition
Simple Multiplier	A single 9-bit, 12-bit, 18-bit, 27-bit, or 36-bit multiplier.
Multiply Accumulator	A single 18-bit multiplier feeding an accumulator.
Two-Multipliers Adder	Two 9-bit or 18-bit multipliers feeding an adder.
Four-Multipliers Adder	Four 9-bit or 18-bit multipliers feeding an adder.

Dynamic architecture using FPGAs

- designing reconfigurable and adaptive hardware structures that can change their functionality on the fly
- Key Concepts:
  - Reconfigurability
    - reprogrammed to adapt to new tasks or updates, enabling dynamic modifications to the architecture
  - Parallel Processing
    - can execute multiple operations concurrently
  - Partial Reconfiguration
    - only a portion of the FPGA is reprogrammed while the rest of the device continues to operate
  - Adaptable Logic Blocks (ALBs)
    - configurable logic elements that can be programmed to perform various digital logic functions

Reconfigurable Example: Adder and Multiplier

```
module reconfigurable_unit(
 input wire clk,
 input wire reset,
 input wire mode,
                // 0 for add, 1 for multiply
 input wire [7:0] data_in1,
 input wire [7:0] data_in2,
 output reg [15:0] result
);
always @(posedge clk or posedge reset)
begin
 if (reset) begin
 result <= 16'b0;
end else begin
 if (mode == 1'b0) begin
 result <= data_in1 + data_in2;
                                  // Add operation
 end else begin
 result <= data_in1 * data_in2;
                                  // Multiply operation
 end
 end
end
endmodule
```

Parallel Processing

```
module parallel_processing_unit(
 input wire clk,
 input wire reset,
 input wire [7:0] data_in1,
 input wire [7:0] data_in2,
 output reg [15:0] result_add,
 output reg [15:0] result_mult
);
always @(posedge clk or posedge reset)
begin
 if (reset) begin
 result_add <= 16'b0;
 result_mult <= 16'b0;
end else begin
 // Parallel operations
 result_add <= data_in1 + data_in2;
 result_mult <= data_in1 * data_in2;
 end
end
endmodule
```

- Both addition and multiplication are performed in parallel, and their results are stored in separate registers.
- This shows the ability of FPGAs to handle multiple computations concurrently, enhancing throughput.

Partial Reconfiguration

```
module partial_reconfig_unit(
 input wire clk,
 input wire reset,
 input wire part_select, // Select part of the module to be active
 input wire [7:0] data_in1,
 input wire [7:0] data_in2,
 output reg [15:0] result_part1,
 output reg [15:0] result_part2
);
// Part 1: Simple addition operation
always @(posedge clk or posedge reset)
begin
 if (reset) begin
 result_part1 <= 16'b0;
end else if (part_select == 1'b0) begin
 result_part1 <= data_in1 + data_in2;
 end
end
// Part 2: Simple subtraction operation
always @(posedge clk or posedge reset) begin
 if (reset) begin
 result_part2 <= 16'b0;
 end else if (part_select == 1'b1) begin
 result_part2 <= data_in1 - data_in2;
 end
end
endmodule
```

Adaptable Logic Blocks - Adaptable logic in FPGAs can be demonstrated by using a simple ALU (Arithmetic Logic Unit) that adapts based on a control input.

```
module adaptable_alu(
 input wire clk,
 input wire reset,
 input wire [1:0] operation_select,
// 2-bit control signal to select
operation
 input wire [7:0] data_in1,
 input wire [7:0] data_in2,
 output reg [15:0] result
);
always @(posedge clk or posedge
reset) begin
 if (reset) begin
 result <= 16'b0;
 end else begin
 case (operation_select)
  2'b00: result <= data_in1 + data_in2;
// Addition
  2'b01: result <= data_in1 - data_in2;
// Subtraction
  2'b10: result <= data_in1 * data_in2;
// Multiplication
  2'b11: result <= data_in1 & data_in2;
// AND operation
  default: result <= 16'b0;
 endcase
 end
end
endmodule
```

Partial Reconfiguration in FPGA hardware

Programming an FPGA

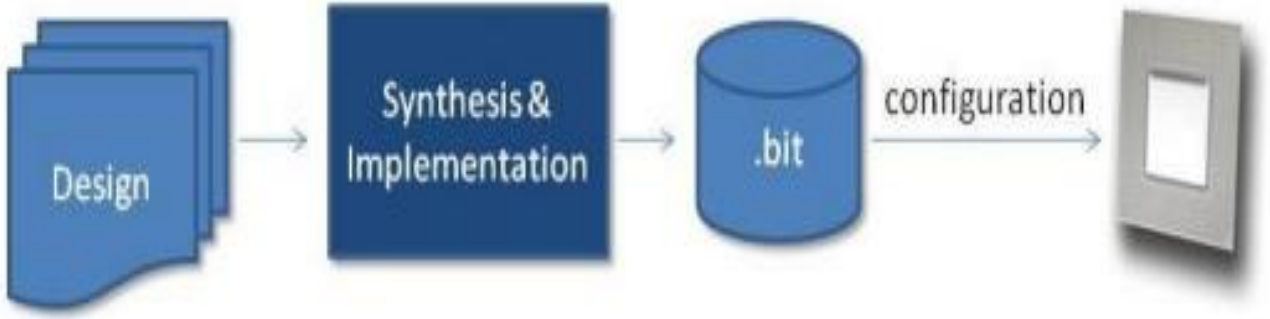
- An FPGA is, by definition, a programmable device.
- An FPGA designer defines the function that the FPGA is to implement using one or more design entry methods.
- These include schematics, hardware description languages such as VHDL or Verilog, and tools to implement DSP algorithms and embedded processors.
- A combination of synthesis and implementation tools converts the design into a bitstream, which is used to program the FPGA.

Partial Reconfiguration in FPGA hardware

Configuration

- Many FPGAs are SRAM-based: the bitstream is loaded into the FPGA's configuration SRAM, which in turn programs the logic in the FPGA.
- The **process of loading the FPGA with a bitstream** is called configuration.
- Configuration usually takes place during system bootup.
- During development and debugging, configuration may be performed interactively, for example by downloading the bitstream from a PC using a USB cable.

Partial Reconfiguration in FPGA hardware

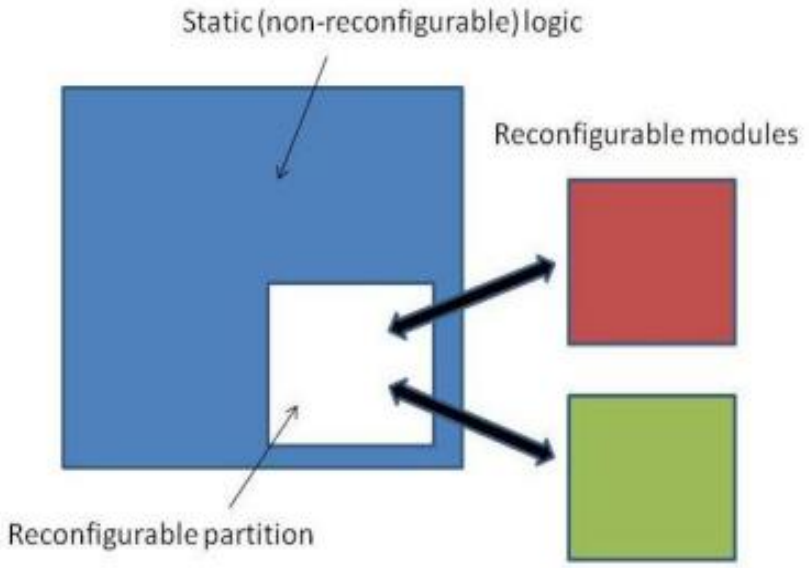


- Once an FPGA has been configured in this way, it starts to perform the required function.
- To change the FPGA's function normally requires that a new bitstream be downloaded, completely overwriting the original configuration; a system reset or power cycle may be required.

Partial Reconfiguration in FPGA hardware

What is Partial Reconfiguration?

- Partial Reconfiguration is where part of the FPGA is reprogrammed, using a partial bitstream, while the rest of the FPGA continues to operate without interruption.



Partial Reconfiguration in FPGA hardware

Why bother?

- to ‘time multiplex’ the function of an FPGA
  - reduces chip count: saves space and reduces cost
  - saves power by ‘swapping out’ unused functionality or I/O standards
  - creates a more flexible system: choose appropriate algorithms and protocols
- to provide a faster boot-up time
  - for example, to meet PCI Express (Peripheral Component Interconnect) enumeration time requirement of 100ms
- to enhance system security
  - meet security requirements for classified bitstreams
  - enable more secure storage of private keys for decryption

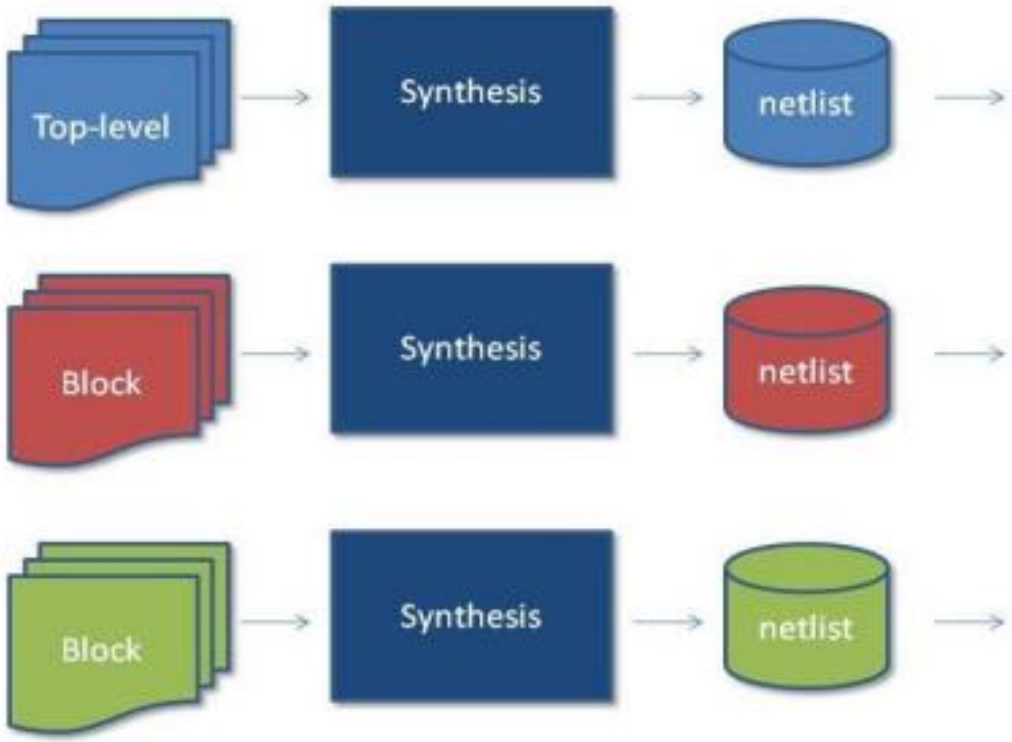
Partial Reconfiguration in FPGA hardware

How does it work?

- HDL and Synthesis

1. Break the design into a hierarchy, where each reconfigurable partition is in its own hierarchical block. Ideally, both the inputs and outputs of each block should be registered.
2. Synthesize each such block as a separate netlist. There will be two or more alternative netlists for each reconfigurable partition. Each of these is termed a reconfigurable module.

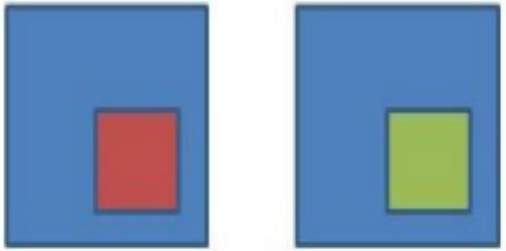
Partial Reconfiguration in FPGA hardware



Partial Reconfiguration in FPGA hardware

- ***Implementation***

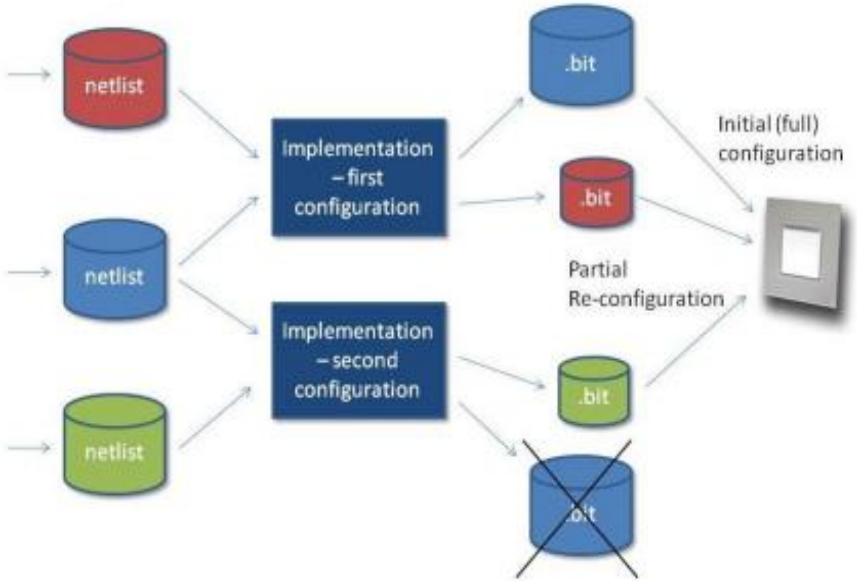
3. Create partitions and area constraints (a floorplan) for these hierarchical blocks and define them to be reconfigurable partitions.
4. Allocate reconfigurable modules (netlists) to these reconfigurable partitions.
5. Define one or more configurations. Be careful about the terminology – a ‘configuration’ in this context comprises a number of netlists, where each reconfigurable partition is associated with one netlist, or none: the partition could be a ‘black box’.



Alternative configurations

Partial Reconfiguration in FPGA hardware

6. Implement a default configuration and any additional configurations. Each configuration will generate one full bitstream and one partial bitstream for each reconfigurable partition/module.



Partial Reconfiguration in FPGA hardware

7. For Design Implementation using the Xilinx ISE tools, Partial Reconfiguration is supported with the PlanAhead tool and also from the command-line. It is not supported in the Project Navigator.

- ***Configuration***

8. Configure the FPGA using the full bitstream.

9. Perform partial reconfiguration using one of the partial bitstreams.

Partial Reconfiguration in FPGA hardware

- *Special Considerations*

1. ***Clocks*** – have to be very careful with global clocks. Because of the way the tools reserve clock resources, the static (non-reconfigurable) logic may be starved of global clock resources.
2. ***Resets*** – Xilinx FPGAs include a global set/reset (GSR) signal, which is asserted during configuration. This ensures that after configuration all flip-flops are in a known state. However, the GSR is NOT asserted after a partial reconfiguration. Therefore, need to provide a separate, local reset for the reconfigurable modules.

Partial Reconfiguration in FPGA hardware

- ***Special Considerations***

3. ***Simulation*** – can simulate any given design configuration, but at present the actual process of partial reconfiguration can't be simulated.
4. ***Decoupling*** – the design should be created so that it works correctly for every combination of reconfigurable modules, and during partial reconfiguration itself. To achieve this, special decoupling logic may need to be added to the design. For example, a reconfigurable module could be held in a reset state during reconfiguration or the flip-flops at the module's inputs and outputs could have their clock enables deasserted.

Verilog code --- 8-tap FIR Filter

```
module Fir_lowpass8(Data_in,Data_out,Clk,Reset);
parameter order = 8,
          word_size_in = 8,
          word_size_out = 2 * word_size_in + 2;
parameter h0 = 8'h1f,
          h1 = 8'h24,
          h2 = 8'hd1,
          h3 = 8'h11,
          h4 = 8'h21,
                    h5 = 8'hf3,
                     h6 = 8'h17,
                    h7 = 8'hab;
output [word_size_out - 1:0]Data_out;
input  [word_size_in - 1:0]Data_in;
input Clk,Reset;
reg [word_size_in - 1:0] samples[0:order-1];
```

```
integer k;
assign
        Data_out = h0 * samples[0] + h1 * samples[1]+
                   h2 * samples[2] + h3 * samples[3]+
                                          h4 * samples[4] + h5 *
samples[5]+
                                          h6 * samples[6] + h7 *
samples[7];
always @ (posedge Clk)
if (Reset==1)
begin
     for(k=0;k < order;k = k + 1)
     samples[k] <= 0;
end
else
begin
 samples[0] <= Data_in;
   for (k=1;k < order; k = k+1)
   samples[k] <= samples[k-1];
end
endmodule
```

DSP KERNELS EXAMPLE --- Moving Average Filter

- The moving average is the most common filter in DSP, it is simple, the moving average filter is optimal for a common task:
- Reduce random noise while retaining a sharp step response.

$$y[i] = \frac{1}{M}\sum_{j=0}^{M-1} x[i+j]$$

where x [i] is the input signal, y [i] is the output signal, and M is the number of points in the average. For example, in a 5 point moving average filter, point 80 in the output signal is given by:

$$y[80] = \frac{x[80] + x[81] + x[82] + x[83] + x[84]}{5}$$

Moving Average Filter -- Verilog code

```
module moving_avg
(clk,reset,datain,dataout);
parameter WL=8, M=5;

input clk,reset;
input [WL-1:0] data_in;
output [WL-1:0]dataout;

reg [WL-1:0] sample [4:0] ;
wire  [WL-1:0] sum,temp1,temp2;
always @ (posedge clk)
 begin
if (reset)
      begin
        sample[0] <=8'b0;
        sample[1] <=8'b0;
        sample[2] <=8'b0;
        sample[3] <=8'b0;
        sample[4] <=8'b0;
    end
```

Moving Average Filter -- Verilog code

```
else
begin
        sample[0] <= datain;
        sample[1] <= sample[0];
        sample[2] <= sample[1];
        sample[3] <= sample[2] ;
        sample[4] <= sample[3] ;
    end
end
```

```
assign sum =
sample[0]+sample[1]+sample[2]+
sample[3]+sample[4];
assign temp1= sum >> 2;
// dividing by 4
assign temp2 = temp1-sum;
// division by 5
assign dataout = temp2;

endmodule
```