Hardware Implementation of a Partial Dynamic Reconfiguration Controller

by

Anup Viswanath Kini

LIU-IDALITH-EX-A—13/042-SE

June 2013, Linköping.
Hardware Implementation of a Partial Dynamic Reconfiguration Controller

Thesis Report
By
Anup Viswanath Kini

LIU-ID/A/LITH-EX-A—13/042-SE

Examiner: Dr. Petru Eles
Professor of Embedded Computer Systems
Department of Computer and Information Science (IDA).
Linköping University.

Supervisor: Adrian Alin Lifa
Ph. D student at Embedded Systems Laboratory
Department of Computer and Information Science (IDA).
Linköping University
Abstract

Partial Dynamic Reconfiguration (PDR) of Field Programmable Gate Arrays (FPGAs) was introduced to overcome the need for more resources on the FPGA fabric. This enabled parts of the device to be reconfigured at runtime, while the rest of the system continued to function without any interruptions. Therefore, PDR could change the functionality and efficiency of the system in order to accommodate more hardware modules, save power and fabric area. Typically, PDR involves the design of modules that are independent of each other so that they can be loaded on the same fabric area (reconfigurable region) one after the other.

This thesis will introduce a framework that enables designers to use PDR in their applications, without having to go into the details of the reconfiguration process. It provides an elegant interface, based on standalone IP modules and an API, which can be used to load modules on to the FPGA fabric at runtime with very little overhead to the main processor. The framework will copy the partial bit files from the configuration memory and reconfigure the FPGA while the application continues to execute useful computations. It will notify the application with an interrupt after completion of the reconfiguration process so that the application can switch context between hardware and software modules.

We validate our controller by using simple test cases that perform FPGA configuration prefetching. However, the framework presented in this thesis can be used as a foundation for many system optimizations targeted at dynamically reconfigurable platforms.
Acknowledgement

I would like to thank my examiner, Prof. Petru Eles, for believing in me and giving me an opportunity to work on this thesis. My supervisor, Adrian, has always been on my side helping and guiding me though the thesis. I would not have come to so far without your support. Thank You Adrian.

My Parents, a special thanks, for being on my side throughout my studies and supporting me. At last, my friends here in Linkoping for the wonderful time in Sweden. The memorable trips, events and pranks, I am grateful to you all.
Contents

1. Introduction ...................................................................................................................................... 1
   1.1. Background ............................................................................................................................... 1
   1.2. Motivation ................................................................................................................................. 1
   1.3. Contributions ............................................................................................................................ 2
   1.4. Thesis Overview ....................................................................................................................... 2
2. Literature Study ................................................................................................................................... 3
3. System and Application Model ........................................................................................................... 6
4. Runtime Partial Dynamic Reconfiguration Controller ......................................................................... 8
   4.1. Memory Read Write Controller IP ............................................................................................ 8
   4.2. ICAP DDR3 Memory Transfer IP .............................................................................................. 9
   4.3. ICAP Processor IP .................................................................................................................... 10
   4.4. Software API ........................................................................................................................... 12
5. Design Tools ....................................................................................................................................... 14
6. Results ................................................................................................................................................ 15
7. Conclusion and Future Work ............................................................................................................ 19
8. References ........................................................................................................................................ 20
Appendix 1: Design Tools ..................................................................................................................... 22
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Detailed structure of UPaRC [2]</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>PDR controller system overview</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>PDR controller block diagram</td>
<td>8</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Control signals from main CPU to Memory Read Write IP</td>
<td>9</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Control signals between ICAP-DDR3 Memory Transfer IP and ICAP Processor</td>
<td>10</td>
</tr>
<tr>
<td>Figure 6</td>
<td>State machine describing the ICAP Processor</td>
<td>10</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Block diagram describing signals between ICAP Processor and main CPU</td>
<td>11</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Experimental setup</td>
<td>15</td>
</tr>
<tr>
<td>Figure 9</td>
<td>PDR Test Case 1 menu screen</td>
<td>16</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Application activity diagram for Test Case 1</td>
<td>16</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Application activity diagram for Test Case 2</td>
<td>17</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Describing the MPMC port type selection</td>
<td>23</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Displaying bus connections between MPMC, mem_wr_rd and icap_ddr3_rd modules</td>
<td>24</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Address Map generated for various peripherals in the system.</td>
<td>24</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Port connections for GPIO and icap_ddr3_rd instances.</td>
<td>25</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Port connections for icap_processor_0 instance.</td>
<td>26</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Parameter update in system.mhs file.</td>
<td>26</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Project Navigator - Hierarchy and Processes window</td>
<td>28</td>
</tr>
<tr>
<td>Figure 19</td>
<td>PlanAhead Netlist window showing reconfigurable regions</td>
<td>29</td>
</tr>
<tr>
<td>Figure 20</td>
<td>A reconfigurable region with three modules added</td>
<td>30</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Physical Constraints window with two reconfigurable regions</td>
<td>30</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Pblock and its Physical Resource Estimates</td>
<td>31</td>
</tr>
<tr>
<td>Figure 23</td>
<td>PlanAhead tool strategy setup</td>
<td>32</td>
</tr>
</tbody>
</table>
1. Introduction

1.1. Background

Since the advent of transistors, Application Specific Integrated Circuits (ASICs) have emerged to enable designers to develop complex and sophisticated application specific designs on a single silicon chip. These ASICs add a lot of non-recurring engineering cost to the custom design of ICs. To overcome this, in recent years, Field Programmable Gate Arrays (FPGAs) have emerged. They combine different types and various number of programmable resources that can be configured during runtime by the application. This has led to shorter time-to-market and rapid development of ICs.

FPGAs consist of a large number of Programmable Logic Blocks (PLBs) and programmable interconnects. PLBs contain Look-Up Tables (LUTs) and Multiplexers (MUXs) that are used to implement combinational logic, and Flip-Flops (FFs) that are used to implement the sequential logic. These PLBs come in varying sizes and programmable flexibility, thereby allowing designers to express complex logic circuits with ease. Apart from this, FPGAs also host a number of analogue and digital blocks such as PLLs, RS232 connectors, VGA / HDMI connectors, Programmable Flash, DDR2/3 RAM and many more that enable them to realize most of the modern ASIC designs.

Partial Dynamic Reconfiguration (PDR) of FPGAs was introduced to overcome the need for more resource on the fabric. This enabled parts of the device to be reconfigured at runtime, while the rest of the system continued to function without any interruptions. Therefore, PDR could change the functionality and efficiency of the system in order to accommodate more hardware modules, save power and fabric area. Typically, PDR involves the design of modules that are independent of each other and could be loaded in the same fabric area (reconfigurable region) one after the other. Thus the reconfigurable region would change its behaviour during runtime, as desired by the application. This feature is now supported by many FPGA vendors, like Xilinx and Altera.

1.2. Motivation

A main disadvantage of PDR is its huge overhead to the main application in terms of time, which is required to move the partial bit files from memory to the FPGA fabric. The application needs to wait for the FPGA fabric to be reconfigured and then continue running by using the hardware modules. This overhead is sometimes more expensive than running the application completely in software. Several solutions have been proposed to reduce this overhead, e.g. bit stream compression [2], configuration caching [9] and configuration pre-fetching [10], [11].

In [2], the authors have proposed a solution that uses a compression algorithm on the partial bit files so that the memory access can be reduced. The compression reduces the size and hence the memory access time, but adds the overhead of decompression. The process of decompression can be carried out either in software or in a dedicated hardware module. In [9], the authors have proposed another design which uses configuration caching, where the partial bit files that are most likely to be loaded in the future or those which are frequently accessed are placed in the on-board configuration cache while the rest of the partial bit files are placed in an external memory.

With configuration pre-fetching, the application is profiled before execution and the time for context switch is determined. The hardware modules are requested ahead in time such that they are ready for execution when the application flow reaches the context switch block of code. The pre-fetch can either be decided statically (before executing the application) or dynamically (at runtime based
on application demands). In Static configuration pre-fetching (e.g. [8], [15]), the modules to be preloaded are decided before runtime and reconfiguration requests are placed early enough such that the reconfiguration is overlapped with useful application execution. In Dynamic configuration pre-fetching (e.g. [8], [7]), the decision whether to run the block of code in hardware or software and which hardware block to request is taken during runtime. This requires an efficient hardware software partitioning (e.g. [1]) and an intelligent pre-fetching of configurations. In [10], the authors have proposed a method of configuration pre-fetching, where the blocks of code to be run in hardware are preloaded so that the reconfiguration can be overlapped with some useful application execution. It requires the entire application to be profiled before execution. In [11], the same authors have developed a dynamic configuration perfecting technique based on Piecewise Linear Prediction. The method tries to learn the application behaviour at runtime and dynamically pre-fetches those modules that will promise to earn the biggest performance improvement.

To perform Configuration Pre-fetching with algorithms as discussed above, a framework is needed that can off-load the reconfiguration process from the main CPU. This will enable the application to issue requests to the framework, which will take care of accessing the partial bit files from memory and configuring the FPGA fabric. This thesis aims to propose such a framework.

1.3. Contributions

The framework would consist of both hardware IP blocks that can handle the memory access and configuration of FPGA fabric and software code that can interact with the application and accept requests as well as inform the application on completion of the reconfiguration tasks. Our contributions have been listed below:

- A VHDL IP based Partial Dynamic Reconfiguration Controller.
- A custom DMA controller that can access partial bit files from a high speed memory, like DDR3, and transfer files directly to the Internal Configuration Access Port (ICAP) to configure the FPGA fabric.
- A Memory Read-Write control logic to transfer data from Compact Flash to DDR3 through the embedded processor.
- An interrupt based interface that could start, stop and continue partial reconfigurations.
- A set of API functions that would include non-blocking request, blocking request and status of the reconfigurable modules.

1.4. Thesis Overview

In this report, Chapter 2 will discuss some of the important related works, applications and evaluation frameworks on reconfigurable platforms. It will also give a brief overview of ICAP interface and its use. Chapter 3 will describe the system overview and some details about how the PDR controller will interact with the application. Chapter 4 will discuss in detail about the various blocks of the PDR controller. The software section will provide more details on application request handling and how the application is informed once the reconfiguration is completed. Chapter 5 will discuss briefly the design tools used. Chapter 6 will discuss the experimental setup used to build the framework and the test cases used while validating the framework. At the end, in chapter 7, conclusion of the thesis and future work that can be expected from the framework is being discussed.
2. Literature Study

Partial Dynamic Reconfiguration has been a field of great interest and research from the early 2000’s. A lot of work and papers have been published since then. Xilinx introduced the Virtex Pro and Virtex Pro II series of FPGAs which have exposed its ICAP. The ICAP interface is similar to the external configuration ports like SelectMap and Jtag interface, which are used to configure the FPGA before start up using an external device like a Personal Computer (PC). The ICAP interface is accessible from general interconnects rather than the device pins [20]. This interface enables users to write a software/hardware module that can read and write to the FPGA configuration memory at runtime. Thus, the embedded processor can modify the circuit structures and functionality on the FPGA fabric during the circuit’s operation.

In order to enable an application to change the circuit structure, a set of circuit configurations known as Partial Reconfigurable Modules (PRMs) are required that can be swapped or placed in predefined regions on the FPGA fabric, known as Partial Reconfigurable Regions (PRRs). The Xilinx tool, PlanAhead, provides an easy interface to create PRRs on the Xilinx FPGA fabric and assign PRMs to each PRR. Each PRR will have its own partial bit file generated, using PlanAhead, that will contain the configuration data specific to it. These partial bit files are read by the embedded processor or any other control logic, written to the ICAP, where the bitstream will update the circuit in the PRR, thereby reflecting a change in the circuit behaviour, i.e. the required PRM.

The authors in [3] have given a brief description on using ICAP interface along with a detailed API and simulation model. Their design was composed of a MicroBlaze embedded processor, a BRAM configuration cache, control logic and the ICAP. The low level drivers were mainly responsible for reading/writing blocks of data from/to the ICAP. They also designed an API which included functions needed to transfer blocks of data, re-configuration of the FPGA and to interrupt the process of reconfiguration. This gave a base for future research to be carried out.

Following this, the authors in [12] have discussed a few designs which address the issue of memory access time while accessing the configuration memory. The first proposed design used a DMA engine which was customized with user registers like Starting Address, Destination Address and Length. The DMA process would remove the overhead of dynamic reconfiguration of the FPGA fabric from the main CPU. On the other hand, it involved the reading of partial bit-streams from an external configuration memory which was time consuming for large bit-file sizes. In the other design discussed, the authors used a large BRAM, sufficient to hold an entire partial bit stream, to reduce the memory access time as compared with the DMA design. The DMA design was suitable in cases where the bit-file sizes were large while the BRAM design was suitable otherwise. A more general design was not discussed which could adapt to the varying bit-file sizes.

With so many controllers for Partial Reconfiguration, the authors in [14] have proposed an effective framework for rapid evaluation of Partial Reconfiguration on FPGA. The setup included a Virtex-II Pro FPGA, a logic analyser and a PC. An embedded processor, Power PC, was used to handle the reconfiguration process and the bit files were saved on Compact Flash, handled using the System ACE controller from Xilinx. The configuration files are moved from the Compact Flash to the processor memory and then to the Hardware Internal Configuration Access Port (HWICAP) module which pushed the configuration data to the internal ICAP. This three step process added a lot of overhead to the reconfiguration process in terms of time required. The design gave a brief overview of the effect of bit file sizes and varying memory caches on the reconfiguration time and bandwidth throughput. The
experimental results were then used to determine the correct sizes for the various caches in a reconfiguration controller.

The authors in [16] have proposed an implementation of a self-aware adaptive system based on FPGA which blends techniques of reconfigurable hardware, performance assertions, monitoring and adaptation. They designed a framework that keeps track of the application heartbeat (the performance metrics of the desired system) and accordingly switch between software or hardware blocks to achieve the goals of the application. The design implemented a continuous monitoring system that evaluates various parameters like time for reconfiguration, bandwidth requirement, time to goal completion and so on. This framework hides all the complexity and the application is unaware of the number of instances of each implementation. It uses a Linux operating system to communicate between the FPGA fabric and the application. It keeps in place both software and hardware versions of each module and decides with the help of an Implementation Switch Service (ISS), which version to be used. This gave rise to autonomous systems that could be designed and configured for a given performance metric and would, at runtime, change their design to meet the goals of maintaining power, voltage, temperature and other essential factors.

The authors in [6] proposed a high speed reconfiguration controller that had a separate bus to access the configuration memory. Thereby, the processor bus has no configuration data traffic and, thus, enables the main processor to communicate with other devices during the reconfiguration process. The embedded processor is used initially to load the partial bit files to the configuration memory. The controller runs independent of the main CPU during reconfiguration and generates an interrupt once the FPGA fabric is re-configured. The design also provides read and write access to the ICAP registers with a few internal functions. The configuration memory, like external DDR, is accessed through a Multi-Port Memory Controller (MPMC) and backed up with FIFO to compensate for different memory clocks. The authors have used active feedback using the Xilinx System Monitor tool, to overclock the ICAP and to monitor the device temperature and voltages in order to maintain them in nominal range. This design is highly flexible with respect to varying bit file sizes since there is no configuration cache used and bit-files are directly read and fed to ICAP. The work does not provide any framework that will help the designer to incorporate this controller into an existing design. The designer has to understand the controller flow and configure it through the application, during runtime, which requires a lot of time and effort to implement.

The authors in [2] have developed a state of the art design, a high speed reconfiguration controller for the Virtex-5 and Virtex-6 series of Xilinx FPGAs, called UPaRC—Ultra-fast power-aware reconfiguration controller. The controller, as described in Figure 1 below, consisted of a BRAM configuration cache, a configuration manager and dynamic clock generator which will handle power-management and speed of reconfiguration. In UPaRC, an embedded processor, like MicroBlaze, is used to perform configuration pre-loading and frequency adaptation for power control. This causes an overhead on the main CPU but is compensated by the speed and data throughput obtained from the high speed ICAP interface. An open source compression algorithm is used to compress the bit stream when its size is greater than the BRAM cache size. The decompression also causes some overhead when performed by the main CPU but has an optional Hardware Description Language (HDL) decompression unit that can be configured on the FPGA fabric. They have attained very high bandwidth, of around 1433 MB/s with uncompressed bit stream loaded from the BRAM cache by overclocking the ICAP. The proposed architecture does not provide a framework that will enable the
user to access the controller from the application directly. Any existing application will be required to be modified to control and provide necessary signals to the controller, which means, the designer will have to have a clear knowledge of the UPaRC framework. Even though the controller is state of the art, it lacks a software interface which will enable the application designer to access the controller through simple function calls.

The current thesis will propose and evaluate a framework that will consist of a PDR controller along with a software API. The API will enable the application designer to access the PDR controller through simple function calls. Thus, the framework can be easily integrated into any application where context switching between hardware and software modules is used.
3. System and Application Model

The system illustrated in Figure 2 is our target architecture. This is composed of a PDR controller to perform reconfiguration of the FPGA fabric, on-board memory in which the partial bit files are stored and the reconfigurable regions defined on the FPGA fabric. In addition, it provides an API to enable the application and the embedded processor to communicate with the PDR controller.

The embedded processor is the main CPU that will run the application and handle the entire system. It would interact with the various peripherals on the Processor Local Bus (PLB) while letting the PDR handle the reconfiguration tasks on a separate bus.

The application that runs on the embedded processor would be profiled to enable Hardware Software partitioning. The authors (Lifa .et. al, 2012), in [10], have provided a motivational example which discusses methods proposed for static configuration pre-fetching. The application is profiled and the requests for modules are placed, keeping in mind the reconfiguration times. Our framework would provide a set of functions to enable the application to place requests for hardware modules, check the current status and also place priority based requests to configure the modules immediately. A set of hardware VHDL IPs will read the partial bit files from the memory and transfer them to the ICAP to configure the FPGA fabric, without the supervision of the embedded processor.

The software API will have a queue which holds the current requests that are to be processed by the PDR controller. The API will also have structures that will contain the details of each hardware module that can be configured on the FPGA fabric. Each of these structures will contain details on whether the module is ready or yet to be configured, location of the partial bit file in

![Figure 2 - PDR controller system overview](image-url)
the memory and whether the block of code is to be run in hardware or software. The API also provides a set of functions that will enable the application to place requests and priority calls. The load function is used to place a request to the PDR controller by placing the corresponding module’s structure on to the queue. The load_now_RM() function is a priority based call that will pause the current reconfiguration, save the state, reconfigure the FPGA fabric with the requested hardware module and then resume the previous reconfiguration.

The PDR controller will accept the structure from the API functions. It will access the corresponding partial bit file from DDR3 memory and configure the FPGA fabric using ICAP. On completion it will inform the embedded processor using an interrupt. Thus, the application continues to run useful computation while the PDR controller completes the reconfiguration of the requested modules.

The System Ace Compact Flash is used as external memory that stores the partial bit files initially before the system start. On start-up, the embedded processor will transfer these partial bit files from Compact Flash to DDR3 memory and store the address of each bit file in the structure for the corresponding hardware module.

The Reconfigurable Region will contain the hardware blocks that will be configured at runtime. The modules can either be placed beside each other in different regions or loaded on top of each other in the same region. The designer or the pre-fetching technique has to determine the placement of the modules and this is not a part of the framework proposed in the current work. The modules that overlap should be selected carefully for example using existing methods, such as the technique presented in [21], [5].
4. Runtime Partial Dynamic Reconfiguration Controller

The system consists of three IPs, namely, the ICAP Processor IP, the Memory Transfer IP and the ICAP-DDR3 Transfer IP as shown in figure 3. The embedded processor, MircoBlaze (main CPU), would run the application and use an API that would enable it to communicate with the ICAP Processor. The main CPU initially would transfer all the partial bit files from Compact Flash to DDR3 memory using the Memory Transfer IP. The main CPU running the application would use the API to request for modules to the ICAP processor which would load the bit files from DDR3 directly to ICAP. The modules will be explained in detail in the following sections.

4.1. Memory Read Write Controller IP

The Memory Read Write IP is used by the main CPU to transfer the partial bit files from Compact Flash (CF) to DDR3 memory. This is performed at the startup, by the main processor. This is done for the following reasons:

- Reading speed from Compact Flash is very low as compared to DDR3.
- The reading from configuration memory to high speed DDR3 has to be controlled by the main CPU, which causes a lot of over-head to the application running.

The IP has been designed using the Xilinx IP Design Tool. It consist of six software mapped registers, Enable, Address, Data_In, Write/Read, Data_Out and Done, shown in figure 4 and explained below, so that the application and the main CPU can send/receive data directly to/from DDR3. The connection to MPMC has been configured through a Xilinx NPI bus interface.
The Reset signal is asserted during the initial phase of main CPU start-up. The Enable is an active high signal that is triggered throughout the access time of the IP. The Write/Read signal (High for Write and Low for Read) determines whether the main CPU wants to read or write the data. The Address signal is passed by adding a certain offset to the base address such that the location points to the user location on DDR3. The Data_In is used while writing data on to the DDR3 i.e. a 32 bit write process while the Data_Out is used for a 32 bit read process, when the main CPU requests for some data from DDR3. The Done is an acknowledgement signal that tells the main CPU that the data on the Data_Out bus is valid to be accessed or data on the Data_In bus has been registered by MPMC.

In the VHDL model a state machine representing the timing diagram for a NPI 32 bit read and write process (given in [18]) has been realised. It also links the user registers to the corresponding ports on the Xilinx NPI bus interface. It takes care of the synchronisation and varying speeds of the two devices also using the FIFO provided along with the bus interface.

4.2. ICAP DDR3 Memory Transfer IP

This IP, as shown in Figure 5, is a customized DMA engine that transfers data between the DDR3 memory and ICAP. The address and synchronisation is handled by the ICAP Processor. This IP handles the varying clock speed of DDR3 memory using an inbuilt FIFO and hence does not require any additional cache. This has been built using the IP generation tool from Xilinx.

The ICAP Processor handles the data transfer between the ICAP and DDR3. The ICAP_DDR3_Reset signal is triggered at the beginning along with the system reset. The ICAP_DDR3_E is the enable signal that is triggered along with the ICAP_DDR3_Address bus. The address is obtained from the main CPU and contains the exact location to fetch data from DDR3. Once the memory location is accessed, the data is put on the ICAP_DDR3_Data bus and the ICAP_DDR3_Done signal is triggered. These signals are directly controlled from the ICAP Processor and are handled by the state machine that keeps a sync of these signals.
4.3. ICAP Processor IP

This IP is the heart of the reconfiguration controller that handles most of the tasks and synchronises itself and other IPs used with the main CPU. The IP consist of a state machine, as shown in Figure 6, which waits for the main CPU to request for reconfigurable modules that are not already configured on the FPGA fabric. Once the module is configured it informs the main CPU with a done signal in the form of an interrupt, ICAP_Done signal. The IP can also be interrupted from the current reconfiguration process to allow the main CPU to request for the reconfiguration of a higher priority module. Thus it supports pause and resume for the reconfiguration process.
When the main CPU issues an *ICAP_Go* signal the state machine moves from Wait-for-start state to the next state, namely Process-packet. In this state, the FSM loads the address of the bit file in the DDR3 memory on to the ICAP-DDR3 Memory Transfer IP and in the next clock cycle the data is moved on to the ICAP Data_In bus for reconfiguration. It also keeps track of the number of bytes written to the FPGA fabric and compares this to the bit file length for finish. Once the entire bit file has been configured, it then moves on to the next state Reset-partition. In this state the module will be reset such that it is in a known valid state. Finally, it completes the process by sending an *ICAP_Done* signal to the main CPU, which is received in the form of an interrupt. The main CPU acknowledges this interrupt and updates with the next module needed as seen in the application queue.

The IP design also includes a priority based request from the main CPU i.e. the main CPU can request the ICAP Processor to pause the current reconfiguration process and attend to a more urgent need for a module. In such cases, the main CPU issues an *ICAP_Stop* signal. The *ICAP_Int_Enbl* signal is made low throughout the priority request call which disables the ICAP Processor from sending an interrupt on completion of the reconfiguration process. The IP stops the current reconfiguration process, updates the number of bytes written on to the fabric back to the main CPU to update in its structures. The main CPU then updates the address field with new partial bit file, starts the reconfiguration by issuing the *ICAP_Go* signal. The main CPU waits for the completion of the reconfiguration process by reading the *ICAP_Busy* signal. Once completed, the main CPU updates the *ICAP_Bitstream_Address* and *ICAP_Bitstream_Length* registers with the address and length of the previous module that was paused and asserts the *ICAP_Go* signal. The main CPU also makes the *ICAP_Int_Enbl* signal to go high again. The ICAP Processor continues the reconfiguration from there on. On completion, the Interrupt Service Routine (ISR) updates the ICAP Processor with the details of the next module to be configured from the queue.

![Figure 7 - Block diagram describing signals between ICAP Processor and main CPU](image-url)
4.4. Software API

Apart from the VHDL modules, a part of the PDR also runs on the main CPU, enabling it to track the requests, get address of bit-files on DDR3 and store the current state of various reconfigurable modules in structures.

4.4.1. Structures and Queue

The API consists of a structure for each reconfigurable module that stores information regarding the status, address and bitstream_length for the main CPU to access.

```c
typedef struct {
    int address;
    int bitstream_cnt;
    int bitstream_length;
    int status;
    bool is_hw;
} RM;
```

A structure is defined for each reconfigurable module in the application. The address and bitstream_length fields are populated by the main CPU during the start-up when the bit-files are moved from Compact Flash to DDR3.

The bitstream_cnt field is populated by the main CPU with the value returned by the ICAP Processor, i.e. ICAP_Bitstream_Count, when a current reconfiguration is paused requesting for a high priority module to be reconfigured. This field stores the current number of bytes already reconfigured on the FPGA fabric. Once the high priority module is configured, the main CPU will update the ICAP_Bitstream_Address register with address (address + bitstream_cnt) and ICAP_Bitstream_Length register with length (length – bitstream_cnt) and assert the ICAP_Go signal for the ICAP processor to resume the previous reconfiguration. This feature enables the PDR to pause and continue, thereby saving time and not having to start over with reconfiguring the paused module again.

The status is a flag used to inform the main CPU, whether a module is already configured (status = 1) and ready to be used or it is being currently reconfigured (status = 2) or if the main CPU has to place a request for the module to be reconfigured (status = 0).

The is_hw is used in cases where, the reconfigurable module has both a software and hardware version. In many applications, not only speed of execution, but many other factors such as time it takes to transfer processing data to the hardware module and get back the results to the software layer are considered. In order to make the controller versatile, this flag has been added, such that, at run-time, an algorithm can determine whether a software or hardware version of the reconfigurable module will enable the application to reach its desired goals on time.

The API also maintains a First In First Out (FIFO) queue, where it stores the requests from the main CPU and updates the ICAP processor. This queue receives the structure for each RM module. When the ICAP processor completes reconfiguring a module, it generates an interrupt, ICAP_Done signal. In the Interrupt Service Routine (ISR), this queue is updated by the pop-out of the first value and then the next module’s values are passed on to the ICAP processor.
4.4.2. API Functions

The API also provides a few functions that enable the designer to place requests for the reconfigurable modules. The CF2DDR3 is used during the system startup to load bitfiles from Compact Flash (CF) to DDR3 memory. The Load_RM and Load_RM_now are used to place requests for the reconfigurable modules on the queue. The ICAP_ISR is to be called when to service the ICAP Processor interrupt. The functions have been explained in detail below.

- **CF2DDR3 (string “bit file_name”, RM_Struct *rm, int PRR)**

  This function is called during the initial startup of the application, where the partial bit files are moved from the Compact Flash (CF) memory to the onboard DDR3 memory. The designer of the application will have to provide the name of each bit file stored on the CF, a new instance of the RM_Struct and the region in which the module will be reconfigured. The function, with the help of the main CPU, will move the partial file from CF to DDR3. The System Ace library from Xilinx is used to read data from CF and Memory Read Write Controller IP is used to write to DDR3. The function will also update the corresponding module structure with the starting address of the bit file on DDR3 to address field, length of the bit file to the length field and reset the module status to not configured by setting the status field to ‘0’.

  The designer will have to make a note of the instance names of RM_Struct used for each partial bit file, since the rest of the API can access only the RM_Struct and not the partial bit file names.

- **load_RM (RM_struct *rm)**

  This function is a non-blocking call. It takes the RM structure as an input and puts the RM in a queue that holds the list of RM that need to be configured in order. It checks the application queue whether this module is already exists and adds it only when the RM is not available in the queue.

- **load_RM_now (RM_struct *rm)**

  This function is a priority call for the application, when it needs a module which is not ready and cannot wait in a queue. It checks whether the ICAP Processor is busy, by reading the ICAP_Busy signal. If the IP is busy, then issues an ICAP_Stop signal. It copies the value from ICAP_Bitstream_Count register to bitstream_cnt field in the structure of the currently reconfigured module. It then makes the ICAP_Int_Enbl signal low. It copies the address and length fields to ICAP_Bistream_Address and ICAP_Bitstream_Lengt registers of ICAP Processor. The ICAP_Go signal is asserted to start reconfiguration. It waits for the reconfiguration signal by checking the ICAP_Busy signal and on completion, changes the status register of the current module to ‘1’. It then loads the ICAP_Bistream_Address and ICAP_Bitstream_Length registers with the updated values of address (address + bitstream_cnt) and length (length – bitstream_cnt) fields from the previously paused module. It changes the value of ICAP_Int_Enbl signal to high so that the ICAP Processor can now generate interrupt. It asserts the ICAP_Go signal and hands over the control back to the application.

- **ICAP_ISR ()**

  This is the ISR that gets called when the ICAP Processor sends the ICAP_Done signal after completion of the reconfiguration process. The ISR updates the first structure in the FIFO queue, setting the status of the module as ready (status = 1). It then pops the first structure. In the next structure, the address and bitstream_length are populated into the user registers ICAP_Bistream_Address and ICAP_Bitstream_Lengt of the ICAP Processor. The ICAP_Go signal is made high for one clock cycle so that the ICAP processor can start with the reconfiguration of the new module. The ISR returns the control to the main CPU.
5. Design Tools

The Xilinx Design Suite 13.2 System Edition was the software toolset used and the Xilinx Virtex 6 FPGA in the Xilinx ML605 Embedded Evaluation Kit was used to design and test the framework.

To begin with, the Xilinx Platform Studio (XPS) was used to build, connect and configure the embedded processor based system. MicroBlaze was the embedded processor used and its peripherals including the UART and Multi-Port Memory Controller (MPMC) for onboard DDR3 memory were configured using the XPS Base System Builder wizard. The netlist for the entire design was generated and exported to the application development platform.

The Xilinx Software Development Kit (Xilinx SDK) was used to build a test application, the software API for the framework and to configure the heap and stack size for the application. The application is linked to the hardware platform using the Executable and Linkable Format (ELF) file which is generated by the Xilinx SDK.

The PlanAhead tool was used to create the various PRRs on the FPGA fabric and to allocate the different PRM to the PRRs. The PlanAhead tool also generated the partial bit files corresponding to each PRM in the design. These partial bitfiles are used by the ICAP Processor to configure the FPGA fabric.

The detailed procedure on using each of the tool and integrating the framework to an existing application has been explained in Appendix 1.
6. Results

An application consisting of 4 Light Emitting Diodes (LEDs) shift register and a Math module with an Adder and a Multiplier have been designed. The application will request the user to select options that will load specific modules onto the FPGA fabric.

6.1. Experimental Setup

- The Xilinx Virtex 6 ML605 Evaluation Board has been used to design and test the framework.
- The board runs the application on bare-metal i.e. without any Operating System to control the application running on the board.
- A CF reader is used to transfer the bit files generated, using the PlanAhead tool, into the CF card.
- A UART connection (at 115200 bauds, 8 bits data, 1 stop bit and no parity bit) to the PC is used to run the test application. Tera Term is the serial communication utility used on PC.

![Figure 8 - Experimental setup](image)

6.2. Test Case 1:

The main purpose of this test case is to check the functionality of the Load_RM() function in the API. The test application will request the user to select one of the displayed options in Figure 9, which will load the corresponding module onto the FPGA fabric using the load_RM() function. It will place the module structure in the queue, start the ICAP Processor by asserting the ICAP_Go signal and continue processing user requests. In the meantime, the ICAP Processor will reconfigure the module and interrupt the main CPU on completion. In the Interrupt Sub Routine (ISR), the module structure is removed from the queue and the status of the module is updated. The result can be seen as either a change in the direction of LED movements or the application requesting for input operands for addition or multiplication depending on the option selected.
Figure 10 shows the activity diagram for the first test case, which involves the LED shifting modules. The activity diagram shows that the reconfiguration process happens independently and does not obstruct the software application running on MicroBlaze. In addition, the example also tests the functionality of the start-up function CF2DDR3() and ISR. The following describes each step in the activity diagram in detail:

- **F1** – The start-up function of the main CPU where the application and other peripherals are loaded and the system starts to execute the application.
- **F2** – The application runs the CF2DDR3() function, that loads the various partial bit files from CF to DDR3 memory.
- **F3** – The application runs on the UART terminal on the PC giving options to the user and the user selects an option. The corresponding structure is loaded onto the queue and ICAP_Go signal is asserted.
- **F4** – ICAP Processor reconfigures the FPGA fabric.
- **F5** – The application continues to execute.
- **F6** – The interrupt from the ICAP Processor is processed by the application.
- **F7** – The LED module starts blinking the LED.
- **F8** – The application continues to execute.
6.3. Test Case 2:

The main purpose of this test case is to check the functionality of the Load_RM_now() function in the API. This test application will request the user to select one of the math modules, addition or multiplication. On selecting, the application will call the load_RM() function to place the corresponding structure on the queue. It will assert the ICAP_Go signal to start the reconfiguration process.

Next, the application will place another request using load_now_RM() for changing the direction of LED movement. This function will pause the current reconfiguration process by asserting the ICAP_Stop signal, save the state of the current reconfiguration on to the structure. It will then load the module requested through load_now_RM(), by writing its address and length on the ICAP Processor registers assert the ICAP_Go signal. Here, the function (also the application and main CPU) will wait for completion of the reconfiguration process. On completion, the load_now_RM() loads the saved state of the previously paused module, asserts the ICAP_Go signal again and hands over the control back to the application. The application continues by requesting the user to enter the operands for the math module.

Once the reconfiguration of the math module is complete, the ISR will change the status of the math module to ready. The application receives the operands from the user and computes the mathematical operation.

In Figure 11, the activity diagram for Test Case 2 can be observed. Each step in the activity diagram has been described in brief in the following:

- **F1** – The start-up function of the main CPU where the application and other peripherals are loaded and the system starts to execute the application.
- **F2** – The application runs the init_DDR3() function, that loads the various partial bit files from CF to DDR3 memory.
- **F3** – The application runs on the UART terminal on the PC giving options to the user and the user selects an option. The corresponding structure is loaded onto the queue and icap_go signal is asserted.
- **F4** – ICAP Processor start reconfiguring the FPGA fabric.
- **F5** – The application calls the load_now_RM() functions to change the direction of LED. It asserts an icap_stop to save the current reconfiguration process, loads the new module and asserts the icap_go signal again and waits for the ICAP Processor to complete. On completion, it loads the
previous state of ICAP Processor and issues `icap_go` signal. It hands over the control to the application.

- **F6** – ICAP Processor reconfiguring the FPGA fabric.
- **F7** – The LED module changes the direction of the LED shifting.
- **F8** – ICAP Processor continues previously paused reconfiguration.
- **F9** – The application requests users for operands for Math module.
- **F10** – The ISR reset the math module.
- **F11** – Math module is reconfigured and ready for execution.
- **F12** – The application executes the math operation using the hardware module.
7. Conclusion and Future Work

7.1. Conclusion

The aim of the thesis was to develop and implement an efficient, flexible and easy to integrate framework that would help designers to implement Partial Dynamic Reconfiguration in their existing applications without having to go into the details of the reconfiguration controller. The following are the achievements from this thesis work:

1. An IP based PDR Controller has been designed with the following features:
   - The controller facilitates pause and resume of partial reconfiguration of the FPGA.
   - The controller removes from the main CPU the overhead of transferring the partial bit files from configuration memory to FPGA fabric.
   - A custom IP has been designed to transfer data from the CF to DDR3 at the application start-up.

2. An API has been designed with the following features:
   - Blocking and Non-Blocking calls to request for reconfigurable modules.
   - Elegant interrupt based update to the application on completion of the reconfiguration process.
   - A queue to hold a large number of reconfiguration requests from the application.

7.2. Future Work

As mentioned above, this thesis aims at developing a framework that can easily be used as a foundation for the future research in the area of dynamically reconfigurable system.

We have identified two orthogonal directions of future work:

The first direction concerns the applications that could benefit from the framework developed here. Just to give an example, we think that all the system optimizations that use FPGA configuration prefetching techniques, as discussed in [10], [11], [13] & [7], would greatly benefit from our framework. Also, any other technique that requires support for PDR (in order to perform optimization of power consumption, temperature, performance etc...) could build upon our framework and specialize it to suit their purpose.

The second direction of future work deals with the potential improvements and refinements that could be done to the proposed framework itself. For example, as pointed out in [17], dynamically reconfigurable systems are not reliable with respect to soft errors and faults. Thus, a fault tolerance techniques could be applied to make our framework more reliable. Apart from this, the system can also be improved by introducing a Dynamic Clock Generation circuit as proposed in [2], which can be used to control the power consumption and temperature of the reconfiguration circuit.

To conclude, this thesis has introduced a framework for PDR that opens up many new opportunities for future improvements and applications.
8. References


Appendix 1: Design Tools.

This section provides detailed instructions to enable the designer to integrate the Partial Dynamic Reconfiguration Framework into his/her project.

The steps in each of the following sub-sections have been written by following UG744 – Partial Reconfiguration of a Processor Peripheral Tutorial document from Xilinx [19].

8.1. Creating an Embedded Processor System

The Embedded Processor System is created using the Xilinx Platform Studio (XPS) which is a part of the Xilinx Design Suite. In the current work, MicroBlaze has been selected as the embedded microprocessor, the main CPU that will run the application.

2. In the application, click Create New Project Using Base System Builder. Click on OK.
3. In the Project File, select the location where the project is to be saved using the browse button.
4. Select PLB System in the Interconnect Type section. Click OK.

The thesis has been implemented on a Virtex 6 Embedded Evaluation Board. The Base System Builder Wizard window will be filled with the following details.

1. Select Create a new design. Click Next.
2. In the Board section, select Virtex 6 ML605 Evaluation Platform as Board Name and D as Board Version. Click Next.
4. Select MicroBlaze as Processor Type, System Clock Frequency as 100.00 MHz and Local Memory as 64 KB. Click Next.
5. In the Peripheral Configuration section, keep the following peripherals and remove the rest.
   - DDR3_SDRAM
   - RS232_Uart_1
   - SysACE_CompactFlash
   - dlmb_cntrl
   - Ilmb_cntrl
   - xps_bram_if_cntrl

6. Change the Uart baud rate to 115200.
7. Change the xps_bram_if_cntrl size to 64KB. Click Next.
8. Click Next and then click Finish.

• Adding the PDR IP to the Processor System.
   Using a file explorer, copy the contents of the provided pcores folder i.e. icap_processor, icap_ddr3_rd, math and mem_wr_rd folders into Project_Location/edk/pcores/ folder.

9. Rescan the User Repositories in XPS by selecting Project -> Rescan User Repositories.
   In the IP Catalog tab, Project Local Pcores -> User will be updated with above mentioned IPs.
10. Double click on each of these IPs to add them to the System.
11. From the IP Catalog tab, add seven instances of **XPS General Purpose IO** from the General Purpose IO folder accepting the default settings.
12. Rename the instances as *icap_bitstreamaddr, icap_bitstreamlength, icap_bitstreamcnt, icap_busy_in, icap_go_out, icap_stop_out, icap_int_enbl*.
13. Click on the drop-down against SPLB for each of these GPIO instances and select *mb_plb* from the options.
14. Right click on *icap_busy_in* and select **Configure IP**. Select Channel 1 and change the GPIO Data Channel Width to 1. Click **OK**.
15. Similarly, change the GPIO Data Channel Width to 1 for *icap_go_out, icap_stop_out* and *icap_int_enbl*.

- Adding an additional Processor Local Bus (PLB) bus for the PDR Controller.
16. From the IP Catalog tab, add **Processor Local Bus (PLB) 4.6** IP, under the Bus and Bridge section, to the system by double clicking on it. Click **OK** in the pop-up window to accept the default settings.
17. From the same section, add **PLBV46 to PLBV46** Bridge also accepting default settings.

- Setting up the MPMC DDR3 SDRAM:
  The Multi-Port Memory Controller will enable us to access DDR3 from more than one port simultaneously. The following steps are required to add read/write ports to DDR3.

18. Right click on *DDR3_SDRAM* in the system assembly view and select **Configure IP**.
19. In Port Type Configuration window, select Port 1 and Port 2 as NPI ports as shown in Figure 8 below.
20. In the Memory Interface tab, make sure that the correct DDR3 memory device is selected by looking into the device number on the hardware.
21. In the Port Configuration tab, select each port active and change the width to **32**. Click **OK** to complete the configuration.

![Figure 12 - Describing the MPMC port type selection](image-url)
In the Bus Interface Tab, DDR3_SDRAM will now have three ports, one connected to mb_plb i.e. the embedded processor bus and two additional ports with no connection.

22. Click on MPMC_PIM1 drop-down and select mem_wr_rd_0_XIL_NPI_Port1.
23. Click on MPMC_PIM2 drop-down and select icap_ddr3_rd_0_XIL_NPI_Port1.
24. Click on the drop-down against SPLB for icap_ddr3_rd_0 and select plb_v46_0.
25. Click on the drop-down against SPLB for mem_wr_0 and select mb_plb.

After this, the port connections should look like Figure 13. This completes the assignment of various IP on the system bus and PDR IP on the additional PLB bus.

26. Select the Address tab and click Generate Addresses to assign the addresses to the added instances. Figure 14 shows the Addresses tab after address generation.

- Port connections between various IP.
27. Select the Ports tab.
28. Expand the clock_generator_0 instance. Click on the drop-down for CLKOUT0 port and select Make External.
29. Expand the icap-processor_0 instance. Click on the drop-down for RP_enable and RP_reset ports and select Make External.
30. Expand the `microblaze_0` instance. Click on drop-down for INTERRUPT port and select Make External.

31. Expand the remaining instances and make connections as shown in the Figure 15.

![Figure 15 - Port connections for GPIO and icap_ddr3_rd instances.](image-url)
32. Expand the `icap_processor_0` instance and make connections as shown below in Figure 16.

![Figure 16 - Port connections for icap_processor_0 instance.](image1.png)

By default, the MicroBlaze interrupt is level sensitive and the level is high. Since the ICAP Processor runs at a higher clock speed, the hold time for the level sensitive interrupt cannot be met. Hence the MicroBlaze interrupt has to be changed to edge sensitive.

33. In the Project tab, right click on `system.mhs` file and select open. Under the definition for MicroBlaze processor add the following two parameters shown in Figure 17.

```plaintext
BEGIN microblaze
PARAMETER INSTANCE = microblaze_0
PARAMETER C_USE_BARREL = 1
PARAMETER C_DEBUG_ENABLED = 1
PARAMETER HW VER = 0.20.a
PARAMETER C_INTERRUPT_IS_EDG = 1
PARAMETER C EDGE IS POSITIVE = 1
BUS_INTERFACE DLMB = dimb
BUS_INTERFACE ILMB = ilmb
BUS_INTERFACE DPLL = mb_plb
BUS_INTERFACE IPLB = mb_pld
BUS_INTERFACE DEBUG = microblaze_0 mcm_bus
PORT MB_RESET = mb_reset
PORT INTERRUPT = microblaze_0 INTERRUPT
END
```

![Figure 17 - Parameter update in system.mhs file.](image2.png)

34. Select Hardware -> Generate Netlist to run the PlatGen tool. This will generate the peripherals and system netlists and system_stub.bmm files required in the following steps.
8.2. Creating the Software Project

Once the netlist is generated from XPS, the next step would be to build the application and integrate the thesis framework API to the application. This will be done using the Xilinx Software Development Kit (Xilinx SDK).

1. In the XPS window, select Project -> Export Hardware Design to SDK.
2. Uncheck the Include bitstream and BMM file option and then click on Export & Launch SDK.
3. Browse to project_location/edk/SDK/SDK_Export and click OK.

A board support package is required which would contain drivers required for MicroBlaze to communicate with the peripherals that were added in beginning of the XPS flow.

5. In the Board Support Package settings window, select xilfatfs. Click OK.

The xilfatfs library provides FAT file system support for accessing data on Compact Flash (CF) from the application.

7. Select Empty Application in the Select Project Template section. Enter a suitable project name (TestApp in our case) and click Next.
8. Select Target an existing Board Support Package and click Finish.
9. Right click on TestApp and select Import.
10. Navigate through General Folder -> File System and browse to project_location/resources and select the icap_api.c and icap_api.h. Click Finish.
11. In the main application file, include the icap_api.h file.
12. Right Click on TestApp and select Generate Linker Script. Change Heap and Stack sizes to 8 KB. Click Generate and answer Yes to over-write the existing file.

This will recompile the application. Fix any errors or warnings that are detected. This completes the application build process.

8.3. Creating the Top Level Design

2. Create a new project by selecting File -> New Project.
3. Browse to the project_location/. Select a suitable project name (top in our case). Click Next.
5. Click Next and Click Finish.
6. In the Hierarchy window, select the device id (xc6vlx240t-ff1156 in our case). Right click and select Add copy of source.

Here, you need to browse and select the top VHDL file which contains the entity and port map declarations to the entire system. This needs to be designed by the user.

7. In the Hierarchy window, select the device id. Right click and select Add Source. Browse to project_location/edk/system.xmp. Click Open.
8. Select the top module, right click on the Synthesize-XST, in the Processes window, and select Process Properties as shown in Figure 18.
9. In the GUI, change the keep_heirarchy setting to Yes in the drop-down. Click OK.
10. Double click on Synthesize-XST in the processes window.
11. Once the synthesis completes, close Project Navigator.

8.4. Creating the PlanAhead Project.

In this section, the PlanAhead tool will be used to floor plan the FPGA fabric with the reconfigurable regions and assign the reconfigurable modules to each region. In the current example, we will create two reconfigurable regions with two modules each. The first region will contain LED shifting control logic with two modules i.e. one to control the left shift of LEDs and the other to control the right shift. In the second region, we will have two math modules, an adder and a multiplier. These modules have been provided in the Xilinx Partial Reconfigurable Design labs in [19].

1. Open PlanAhead by selecting Start -> All Programs -> Xilinx ISE Design Suite 13.2 -> PlanAhead -> PlanAhead.
2. Click on Create New Project.
3. Click Next. Browse to the project_location folder and give a project name. Click Next.
4. Select Specify Synthesized (EIDF or NGC) netlist option, check the Enable Partial Reconfiguration checkbox. Click Next.
5. In the Add Netlist Sources window, click on Add Files, browse to project_location/top (the project that was created using ISE Project Navigator). Select the top.ngc file. Click Open.

6. Click on Add Directories, browse to project_location/edk. Select the implementation folder and click Select. Click Next.

7. In the Add Constraints window, click on Add Files. Browse to project_location/edk/data/. Select the system.ucf file, click OK.

8. Click Next twice and then click Finish.

9. Click on Netlist Design in the Project Manager window to load the netlists for the design.

- Defining a Reconfigurable Partition.
  
The following steps will define the various modules that will be placed in each of the reconfigurable regions in the design.

10. Select one of the reconfigurable regions in the netlist window as seen in Figure 19. Right click and select Set Partition.

![Netlist](image.png)

Figure 19 - PlanAhead Netlist window showing reconfigurable regions.
11. In the Set Partition window, click Next twice. Enter a suitable name (led_BB or math_BB) in the Name field and select Add this Reconfigurable Module as a black box without a netlist.

12. Click Next and then click Finish.

A black box will be created for the selected partition. The remaining modules that need to be added in the same region should be compiled and netlist should to be generated either using XPS, Project Navigator or any other tool.

13. Select the reconfigurable region in netlist window, right click and select Add Reconfigurable Module.

14. In the Set partition window, click Next twice. Enter a suitable name for the module and select Netlist already available for this Reconfigurable Module. Click Next.

15. In the Import Netlist window, browse the location where the netlist for the current module is located and click Open.

16. Click Next twice and then click Finish.

```plaintext
Figure 20 - A reconfigurable region with three modules added.
```

Similarly more reconfigurable modules can be added to the reconfigurable region by repeating steps 14 to 17.

- Defining a Partial Reconfigurable Region on the FPGA fabric.

17. Select the pblock corresponding to the reconfigurable partition from the physical constraints window as shown in Figure 21. (If not found, select Window -> Physical Constraints).

```plaintext
Figure 21 - Physical Constraints window with two reconfigurable regions.
```

18. Right click on the pblock and select Set Pblock Size.
19. Move to the Device tab and draw a region big enough, on the FPGA, such that the statistics tab in the Pblock Properties window shows that all the requirements are satisfied as seen in Figure 22.

![Figure 22 - Pblock and its Physical Resource Estimates.](image)

Similarly, create partitions on the FPGA for the rest of the Pblocks in your design.

Now, it is time to generate the bit files for the static region and the reconfigurable modules. It is always a good idea to run Design Rule Checker (DRC) to catch any errors.

20. Select Tools > Run DRC.
21. Deselect all and select only Partial Reconfig. (You can also select the remaining tests depending on your interests) Click OK.
22. Fix any errors detect by DRC and then proceed.

- Creating Configuration, Implementing and Promoting.
  A configuration is a rule that defines the various reconfigurable modules that can be placed in the reconfigurable regions at a given time and also generated partial bit files for each of those modules.

25. Click on + button to create a new strategy. Name the new strategy as ISE_BM and set the Type to Implement. Click OK.
26. Under Translate (ngdbuild), under the More Options field type `bm ..\..\..\edk\implementation\system.bmm`. Click Apply and then click OK.

- Selecting reconfigurable modules for the configuration.
27. At the bottom of the PlanAhead window, select the Design Runs tab, select config_1.
28. In the Implementation Run Properties window, select the General tab.
29. Rename the configuration to match the reconfigurable modules. Click Apply.
30. In the Partitions tab, select the module variant from the drop-down. Click Apply.

31. In the Design Runs tab, right click on the configuration and select Launch Runs to Implement.
32. Select the Launch Runs on Local Host option. Click OK. Save the project if asked.
33. On completion, select Promote Partitions and click OK twice.

34. To create more runs, select Flow -> Create Runs.
35. Click Next twice. Now, change the name (if required) to a suit the modules to be loaded.
36. More runs can be added by clicking on the More button. Click Next and then Finish.
37. To select the module variant for the Implementation Runs, repeat steps 28 to 32.
38. On completion of the runs, click Cancel.

- Generation of Bit Files.
39. In the Design Runs tab, select all the implemented runs using the shift key, right click and select Generate Bitstream.
40. In the Properties window, under the more options section type the following –g configfallback:Disable. Click OK to generate the bitstreams.

The bit streams for each configuration and the partial bit streams for each of the reconfigurable modules will be placed under project_location/project_1/project_1.runs/ directory.

- Creating the download.bit and system.ace file.
42. Type the following command to generate the download.bit file with the software application included and any one variant of the implementation run.

```bash
data2mem -bm ..\edk\implementation\system_bd -bt ..\PlanAhead\PlanAhead.runs\adder\adder.bit -bd ..\edk\SDK\SDK_Export\TestApp\Debug\TestApp.elf tag microblaze_0 -o b download.bit
```

43. To generate the system.ace file from the download.bit file,

```bash
xmd -tcl genace.tcl -jprog -target mdm -hw download.bit -board ml605 -ace system.ace
```

Both, the system.ace and download.bit file will be generated in the current directory. Using a File Browser copy these two files along with the other partial bit files to Compact Flash.