FPGA Emulation Of REDEFINE : A Polymorphic Asic

A project report
submitted in partial fulfillment of the
requirements for the Degree of

Master of Engineering

in

Microelectronic Systems

by

C. Madhava Krishna

under the guidance of

Prof. S. K. Nandy

Department of Electrical and Communication Engineering
INDIAN INSTITUTE OF SCIENCE
Bangalore - 560012

October 2013

Acknowledgements

I would like to acknowledge various people with whom I was directly and indirectly associated with during the course of my ME.

First and foremost, I express my gratitude to my advisor, Prof. S K Nandy for his guidance, motivation, encouragement and support throughout my stay at CAD Lab. I owe my thanks to faculty of ECE and CEDT department for their insightful course work. The course projects and assignments are very much helpful in building a strong foundation for my basics. My special thanks to Dr. Virendra Singh and Dr. Kurvilla Varghese for being my examiners for this work.

I would like to thank Dr. Ranjani Narayan, Vaibbhav and Ganesh (Morphing Machines pvt. ltd.) for their support during difficult times. I am very happy to be a part of CAD Lab. Special thanks to my lab mates Keshavan, Mythri, Adarsha, Rajdeep, Alex, Saptharshi, Sanjay, Farhad, Ramesh, Ashish, Kala, Hemanth, Shanti and Nandini for their help and support.

Special thanks are due to all my classmates and friends of IISc, who made my stay at IISc more colorful and memorable.

Finally, I thank the visionaries Swami Vivekananda and J N Tata for conceiving the institute that is IISc, of which I could become a tiny part of.

Madhava Krishna C
29^{th} June, 2011

Abstract

As complex SoCs have higher risk of respin, FPGA (Field Programmable Gate Array) emulation is more reliable way of pre-silicon SoC validation compared to software simulations. Other than functionality validation of SoC design, FPGA emulation allows concurrent development and testing of software that drives the SoC and field testing of SoC design.

Direct implementation of an ASIC design in a FPGA using the same RTL design methodology will not yield better performance and resource utilization. The design needs to be optimized for the FPGA architecture. This report presents the methods used to optimize a design, described in a high-level (above RTL) behavioral description. This also covers some of the coding practices that should be followed for better FPGA area utilization. The targeted FPGA is a Xilinx virtex-6 device using Xilinx synthesis tool.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview of REDEFINE

REDEFINE is a polymorphic ASIC comprises of a reconfigurable Fabric and a Support Logic to control the resources on the Fabric. Fabric is an array of tiles interconnected in a toroidal honeycomb topology. Each tile comprises a CE and a router. Figure 1.1 is a reproduction of the architecture that appears in [1]. Unlike FPGAs, where configurable logic blocks (CLBs) are SRAM based memory lookup tables used to define application specific data paths, in REDEFINE application specific data paths are defined in terms of computational structures at runtime. A computational structure is a physical aggregation of hardware resources (CEs and routers) that perform a coarse grained operation, referred as a Hyper Operation (HyperOp). In REDEFINE applications are specified in a High Level Language (HLL). RETARGET is the compiler tool chain that is used to compile applications to a intermediate form and convert it into data flow graphs [2]. The compiler determines HyperOps that are sub-graphs of application data flow graph and comprise elementary operations that have strong producer-consumer relationship. These Hyperops are further divided into several partial HyperOps (p-HyperOps) and each p-HyperOp is assigned to a CE. RETARGET captures the computations to be performed by each p-HyperOp in terms of Compute Metadata and inter/intra HyperOp communication in terms of Transport Metadata. The Support Logic, at runtime launches the HyperOps on the execution fabric following a dynamic data flow schedule. Each HyperOp is assigned to a set of CEs which are interconnected through routers during runtime to form a pattern

that closely matches the communication pattern of that particular application. This results in creation of different execution patterns on the Fabric both in space and time. REDEFINE has support for global memory, which is accessible through Load Store Unit (LSU) connected to access routers as shown in the figure. The global memory is used to support vector operands, pointer based accesses and input/output data.

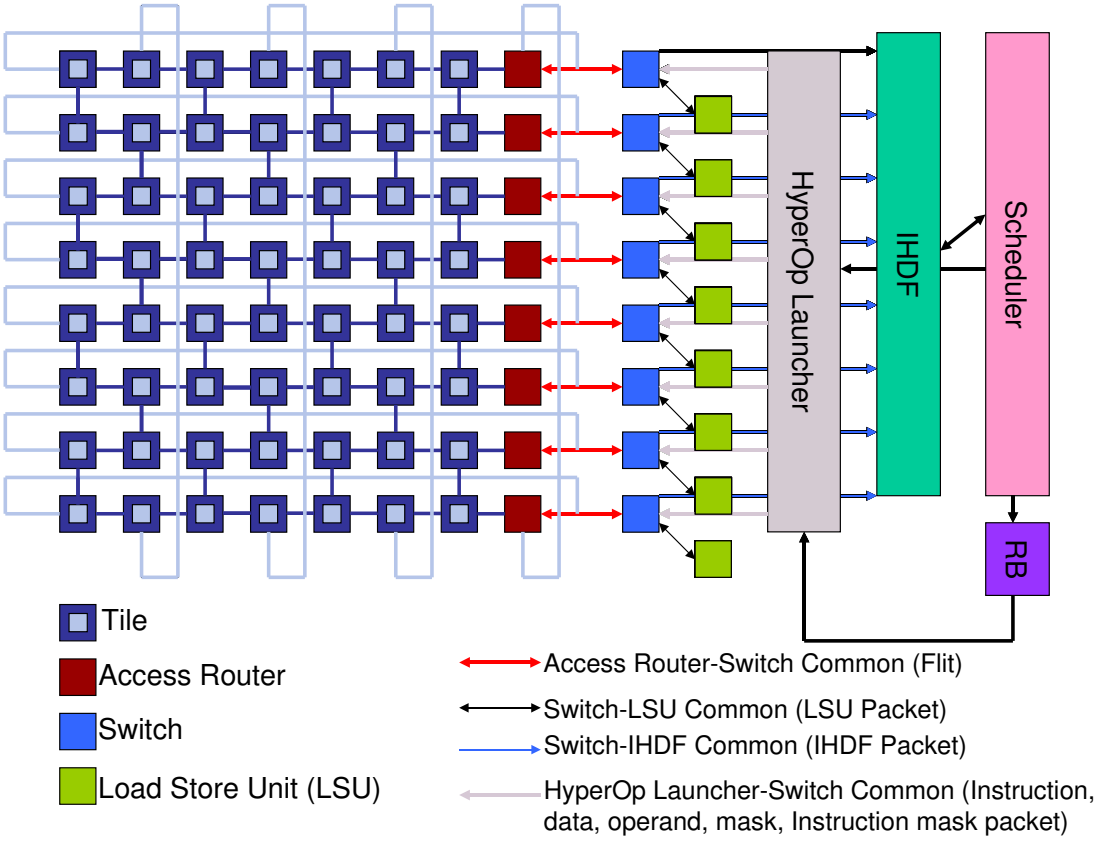


FIGURE 1.1: Architecture of REDEFINE

1.2 High-Level Synthesis

High-level synthesis allows the designer to write high-level (above RTL) behavioural descriptions of a design that can be automatically converted into synthesizable RTL code. High-level code describes the operation on different data types of a hardware design without specifying the required resources and schedule of cycles. Thus, high-level synthesis offers the advantage of automatic handling of the scheduling and synchronization issues of a design, which helps in faster architectural exploration leading to reduced design time and

increased productivity. Gate-level designs are further synthesized using other downstream tools to generate ASICs or FPGAs.

The REDEFINE architecture is designed using one such High-Level Synthesis tool called Bluespec Compiler, in which the high-level specifications of the hardware designs are described in Bluespec System Verilog (BSV).

1.3 Contribution of the Report

The BSV generated RTL for the Fabric without any modifications for FPGA architecture was not able fit in single FPGA device that is available in the market. As REDEFINE architecture supports parametrizable Fabric size (figure 1.1 shows 8×8 Fabric size), Fabric with size reduced to 6×6 was tried. This alone barely fits in the FPGA device Virtex-6 LX760T, the biggest one currently available in the market.

This works includes efforts made to reduce the area utilization if the REDEFINE design, with out considering the timing performance. The objective was to fit REDEFINE in single FPGA device by making coding modifications at BSV level and in the Bluespec Verilog library files, instead of doing them in Verilog files generated by Bluespec compiler. Some of the modifications proposed are not specific to the FPGA design.

1.4 Report Organization

The main theme of this work is to covers the coding practices that should be followed for better FPGA area utilization. It also covers some of the architectural changes that implements same functionality with less area utilization.

Chapter 2 covers the general optimization methods that are followed all over the REDEFINE design for better area utilization.

Chapters 3 and 4 covers the methods that are specific to the respective modules in REDEFINE. Chapter 3 covers the area efficient implementation of CE and Router and chapter 4 covers the area efficient implementation of Support Logic individual modules.

In Chapter 5, resource utilization of the individual modules of the REDEFINE when implemented on a Virtex-6 FPGA device are shown. A comparison is done between the FPGA resource utilization before and after applying the changes.

Chapter 2

General Optimization Methods

In this chapter, the methods used to reduce the FPGA resource utilization of REDEFINE design are discussed. These methods are used all over the design and are not unique to the REDEFINE architecture.

2.1 Restricted usage of Tagged Unions

A union is a variable that can hold (at different times) component values of different types and sizes, with complier keeping track of size and alignment requirements. When a component is injected into a union value, it loses its identity i.e., there is no way to know which summand it came from. In tagged unions, a union value always has a tag that "remembers" which summand it came from [3]. System Verilog has ordinary unions as well as tagged unions, but the BSV uses only tagged unions as they provide complete type-safety, greater brevity and more visual apparent notation.

A typical usage of tagged union

```
typedef union tagged { Bit#(32) Data;
                       Bit#(3) OpCode;
                     } U deriving (Bits);
U x;
```

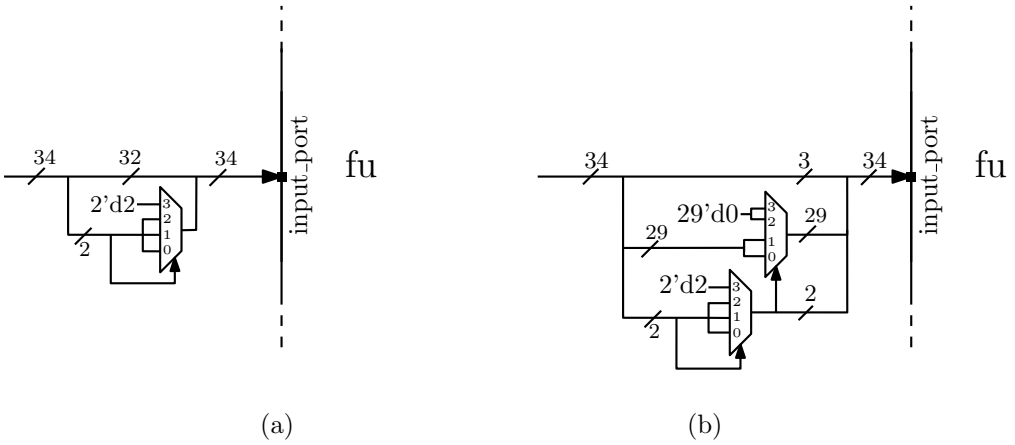
In a tagged union the member names are called tags. Tags play a very important safety role. In the above usage variable `x` not only contains the bits corresponding to one of its member types `Bit#(32)` or `Bit#(3)`, but also some extra bits (in this case just one bit) that remember the tag, 0 for `Data` and 1 for `OpCode`. When the tag is `Data`, it is impossible to read it as a `OpCode` member, and when the tag is `OpCode` it is impossible to read it as an `Data` member, i.e., the syntax and type checking ensure this. Thus, it is impossible to accidentally misread what is in a tagged union value. `deriving (Bits)` is optional, which tells the compiler to select a default bit-representation (`pack` and `unpack` functions [4]) for tagged unions. The representation consists of t+m bits, where t is the minimum number of bits to code the tags in this tagged union and m is the size of the largest member in bits. Every tagged union value has a code in the t-bit field that identifies the tag, concatenated with the bits of the corresponding member which is right-justified in the m-bit field. If the member needs less than m bits, the remaining bits (between the tag and the member bits) are undefined (don't care). If `deriving (Bits)` is not used, then the programmer has to define the bit-representation i.e., `pack` and `unpack` functions for the tagged unions explicitly along with the unused bits of the member.
Though usage of tagged unions simplifies verification and improves readability of the code, they have implementation overhead with respect to logic cells. So misuse of tagged unions implements redundant logic cells in the design. The BSV snippet below shows one such usage.

```
typedef union tagged { Bit#(32) Operand1;
                       Bit#(32) Operand2;
                       Bit#(3) OpCode;
                     } Packet deriving(Bits);

module mkUnion_sys(IFC);
        // Instantiation of mkFunctionalUnit module
        FU fu <- mkFunctionalUnit;
        Wire#(Packet) dataIn <- mkWire;
        .....
        rule rl_put;
                fu.input_port(dataIn);
```

```
        endrule
        .....
endmodule
```

Example 2.1 - Misuse of tagged Unions



(a) (b)

FIGURE 2.1: Logic generated from usage of tagged unions (a) Default bit-representation (b) User defined bit-representation with unused bits assigned to zero

In Example 2.1, `dataIn` wire in `mkunion_sys` module connects to `fu` module's `input_port`. As the data in wire is tagged union it is implemented with some logic overhead. The logic overhead is due to the mismatch in the number of members (3) defined and the possible number of members (4) that can be represented using tag-bits. As shown in Figure 2.1.(a), BSV implements a multiplexer at the tag-bits to ensure functional correctness (it ensures that tag-bits doesn't represent the members other than defined). If bit-representation is user defined then logic overhead may increase as shown in Figure 2.1.(b). In `mkUnion_sys` there is no requirement to examine the value in tagged union variable. So the data to be transferred to the `fu` module need not be a tagged union. The redundant logic can be avoided by restricting usage of tagged unions only when there is requirement to examine the value in the tagged union. Instead of using `Packet`, `Bit#(SizeOf#(Packet))` (`SizeOf` is used to convert a type t into the numeric type representing its bit size [5]) should be used and when ever there is requirement to examine the value it can be converted back to `Packet` using `unpack` function.

This applies to Enumeration data type also and same technique is used to avoid redundant logic generated from misuse of enumerations.

2.2 Multiplexer Restructuring

Multiplexers are the common building block for data-paths and are used extensively in number of applications. Xilinx FPGAs provide dedicated multiplexer resources, which can implement multiplexers efficiently. The inference of these dedicated FPGA resources is more sensitive to HDL coding style. BSV generated Verilog has multiplexers described in full parallel case statement and full non-parallel case statement and this coding style consumes more lookup tables (LUTs). This section discusses the multiplexers described using full parallel and full non-parallel case statements and method used to implement them efficiently on an FPGA.

- A "full" case statement is a case statement in which all possible case-expression binary patterns can be matched to a case item or to a case default [6].
- A "parallel" case statement is a case statement in which it is only possible to match a case expression to one and only one case item [6].

Full non-parallel case statement

Example 2.2 shows a case statement that is full and non-parallel as case-expression can be matched either to case item or to a case default and more than one case item can potentially match the case expression. This simulate like a priority encoder where the priority decreases along `a`,`b`,`c` and `d` respectively. This will also infer a priority encoder when synthesized. Figure 2.2 shows the logic generated, according to the functionally it is a 4:1 multiplexer with priority encoded select lines.

```
always@(a or b or c or d or in)
begin
        case(1'b1)
                a: out = in[0];
                b: out = in[1];
                c: out = in[2];
                d: out = in[4];
                default: out = 0; /* unspecified value */
        endcase
```

```
end
assign RDY_out = a || b || c || d;  //validity of out
```

Example 2.2 - Full non-parallel case statement

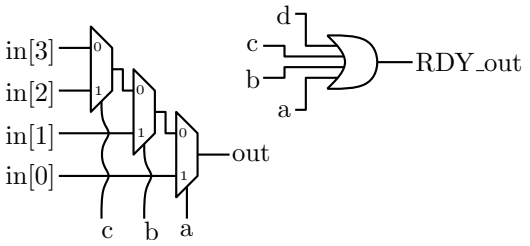


FIGURE 2.2: 4:1 MUX implementation from a non-parallel case statement

Full parallel case statement

Example 2.3 is same as Example 2.2 except that a "synopsys parallel_case" directive has been added to the case header. This will simulate like a priority encoder but will infer a non-priority encoder logic when synthesized, now the synthesized logic doesn't match the Verilog functional model. The " synopsys parallel_case" directive tells the synthesis tool that these case items (`a`,`b`,`c` and `d`) are mutually exclusive i.e., at any point of time exactly one case item is set to one and remain all are set to zero. Generally "parallel_case" directives are used in Verilog code for large ASIC design to remove stray priority encoders and infer a smaller and faster design. Figure 2.3 shows the logic generated, according to the functionally it is a 4:1 multiplexer with one-hot encoded select lines.

```
always@(a or b or c or d or in)
begin
        case(1'b1) \\ synopsys parallel_case
                a: out = in[0];
                b: out = in[1];
                c: out = in[2];
                d: out = in[4];
                default: out = 0; /* unspecified value */
        endcase
end
assign RDY_out = a || b || c || d;  //validity of out
```

Example 2.3 - Full parallel case statement

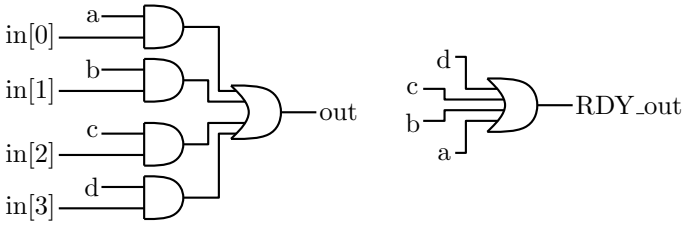


FIGURE 2.3: 4:1 MUX implementation from a parallel case statement

The resources required to implement a Boolean function on an FPGA depends only on the number of inputs and outputs of the function, not on the complexity of the function. The minimum number of inputs required for a Boolean function to implement a 4:1 multiplexer are 6, but in examples 2.2 and 2.4 8-inputs are used which is not efficient.

In Xilinx Virtex-6 FPGA each slice contains four 6-input LUTs and 3-multiplexers *F7AMUX*, *F7BMUX* and *F8MUX*. These multiplexers are used to combine up to four LUTs to generate any Boolean function of seven or eight inputs in a slice. These dedicated multiplexers also allows implementing 8:1 multiplexer and 16:1 multiplexer in one logic level using 2 LUTs and 4 LUTs respectively [7]. But there are limitations in the ability of the synthesis tool to infer these dedicated multiplexers. FPGA synthesis tool doesn't synthesize the multiplexers described in parallel and non-parallel using dedicated multiplexers present in the slice. To implement them efficiently the select lines should be compressed which consumes extra LUTs. If the multiplexers occur in buses then these select lines are shared among all the multiplexers in the bus. So when this technique is applied to buses multiplexers over all LUT count reduces. Figure 2.4 shows the efficient implementation of multiplexers. This technique can be applied to any multiplexer of size greater than 3:1 because 3:1 multiplexers described using parallel and non-parallel case statements is a 6-input boolean function, when implemented on a Virtex-6 FPGA consumes only 1 LUT for one bit data size.

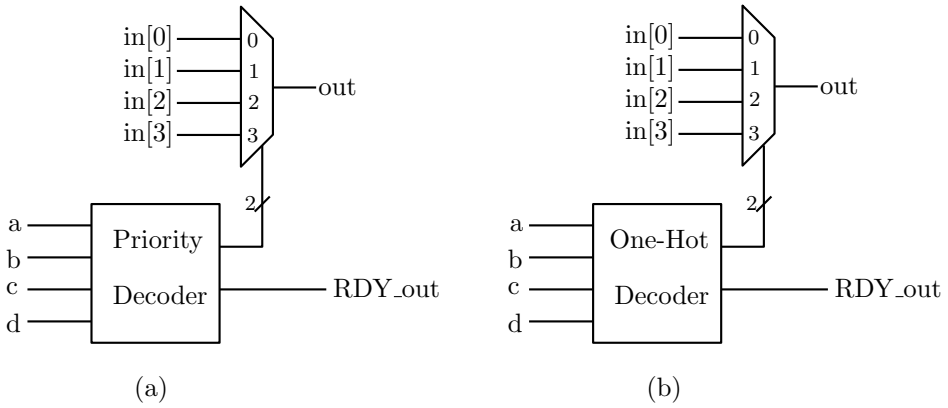


FIGURE 2.4: Restructuring multiplexers (a) non-parallel case statement (b) parallel case statement

2.3 Optimizing Bluespec verilog library modules for Xilinx FPGA

Fine-grain ASIC architecture have the ability to tolerate a wide range of RTL coding styles while still allowing designers to meet design goals. Course-grain FPGA architecture are more sensitive to coding styles and design practices. In many cases a slight modifications in coding practices can improve the system performance anywhere from 10% to 100% [8]. BSV has inbuilt packages for commonly used modules in a design. These modules are part of Bluespec library and should be added to the synthesis tool for complete synthesis of the design. FIFO2, SizedFIFO and BRAM2 are more commonly used Bluespec library modules in the REDEFINE design. This section discusses the changes made to these Verilog modules for better FPGA area utilization.

FIFO2 and SizedFIFO

The corresponding BSV modules for FIFO2 are mkFIFO and mkFIFOF, FIFOs of depth 2 and for SizedFIFO are mkSizedFIFO and mkSizedFIFOF, FIFOs of parametrizable depth. The main components of a FIFO are memory and write and read control units. The Verilog snippet below show the instantiation of memory in FIFO2.

```
\\ FIFO2.v
reg [width - 1 : 0]    data0_reg;
reg [width - 1 : 0]    data1_reg;
```

This synthesizes memory using registers and requires `width` size 2:1 multiplexer to read data from this registers. The better way to implement this memory is using distributed RAM, which eliminates data multiplexer. Similarly in SizedFIFO.v, to implement a n-depth FIFO the memory is split between memory of size n-1 and a register stage at read port, this is used to meet the timing performance in ASIC design. For better area utilization this is combined as single memory and implemented as distributed RAM for FIFO depth less than 64.

Simple Dual-Port BRAM

Xilinx BRAM resources can be configured as either simple dual-port (SDP) BRAM or true dual-port (TDP) BRAM. In SDP configuration, BRAM has one port for synchronous reads and one port for synchronous writes. In TDP configuration, BRAM has two port each can synchronously read and write. BRAM in SDP mode uses only half of the resources of BRAM in TDP mode of same size. In most of the time SDP serves the purpose but Bluespec BRAM packages doesn't have module that infer BRAM in SDP mode with simultaneous read and write access. A module that infers a BRAM in SDP mode with simultaneous write and read operations is written in Verilog and imported to BSV.

Chapter 3

Compute Element and Router

The execution engine of the REDEFINE, Fabric comprises a matrix of tiles. Each tile comprises a compute element (CE) and router that connects CEs. The set of router together serves as Network on Chip (NoC). CE comprises of Local Wait Match Unit (LWMU), ALU and Transporter.

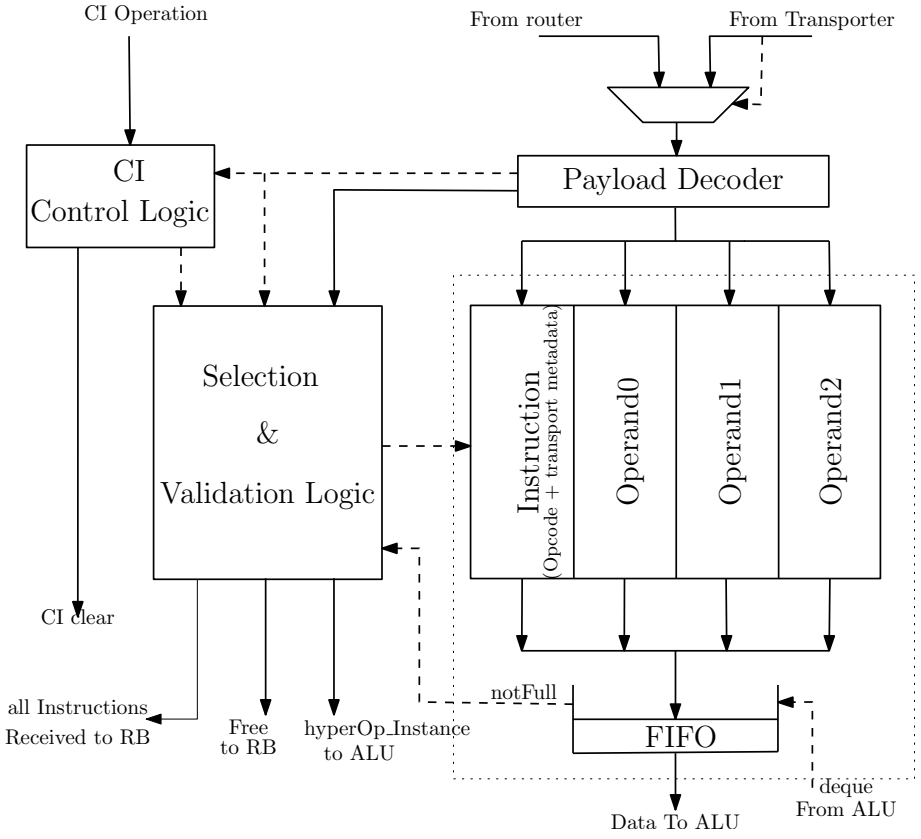


FIGURE 3.1: Local Wait Match Unit

LWMU shown in figure 3.1, consists of a storage unit, a selection and validation unit and a custom instructions (CI) control unit. LWMU receives payloads which can be an instruction or an operand or a control packet. The instructions and operands are stored in storage unit.The control packet are directed to CI control unit or selection and validation unit depending on its tag value. Selection and validation unit is responsible for launching the instructions that are ready to be executed. Once all instructions in the storage unit are executed it declares itself as free. In order to reduce the delay associated with assignment of HyperOps to CEs, which get repeatedly executed, HyperOps can be made persistent across multiple iterations of execution. If the instructions in the CE are persistent then CI control unit keeps track of number of iterations the instructions in storage unit are executed and restart every iteration depending on CI operation input and CI control packet. LWMU forwards the instruction and their operands to the ALU for execution. ALU is a combination of different functional units, depending on the opcode of the instruction the data is transferred to corresponding functional unit. The ALU

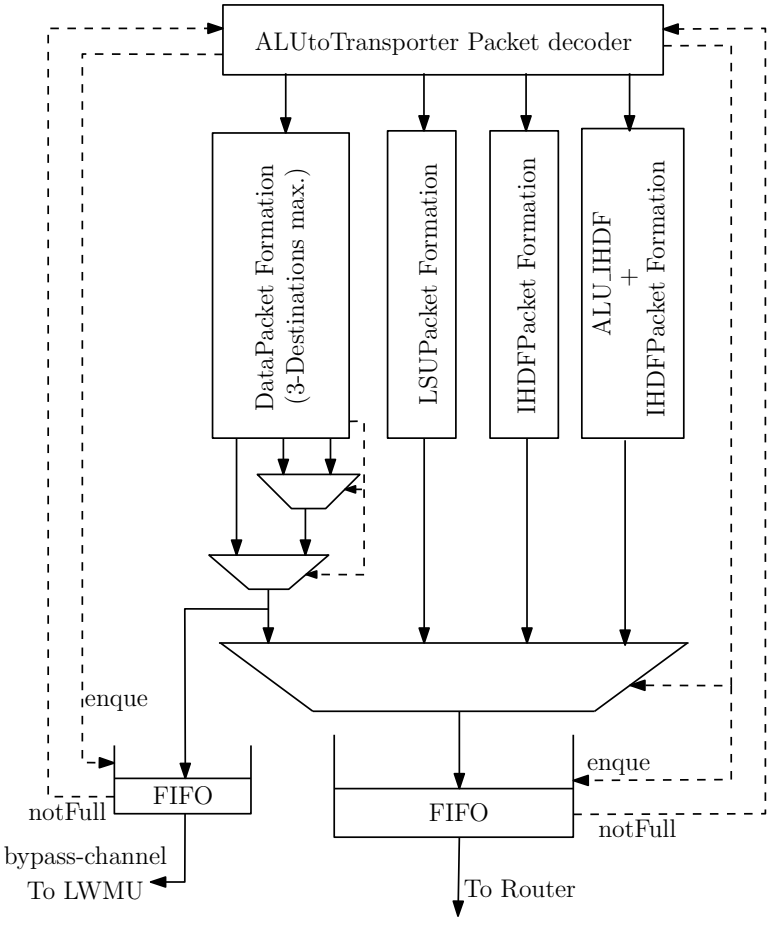


FIGURE 3.2: Transporter

computes the result and forwards it to the Transporter along with transport metadata. Figure 3.2 shows the block diagram of Transporter. The destination of the result from ALU can be same CE or other CEs or Inter HyperOp Data Forwarder (IHDF) or Load Store Unit (LSU). Using transport metadata the Transporter packs the result and forwards it bypass-channel, if it is for the same CE otherwise it is forwarded to the router. The router is responsible for sending packets to their respective destinations. The routers implement a deterministic routing policy which allows in order packet delivery between one source and destination pair. The router also supports deadlock free routing. Each router has 4 input ports and 4 output ports, among which one input port and one output port are connected to CE which is in the same tile as the router and the remaining 3 pair of ports are connected to the other 3 routers in the three adjacent tiles. The access routers differ from the ordinary routers in such way that instead of making a connection with CE, it is connected to Support Logic. The access routers serve as gateway of communication between Fabric and Support Logic. Figure 3.3 shows the block diagram of router, the output-port to which the CE or Support Logic is connected will not have address update logic as it is the destination port.

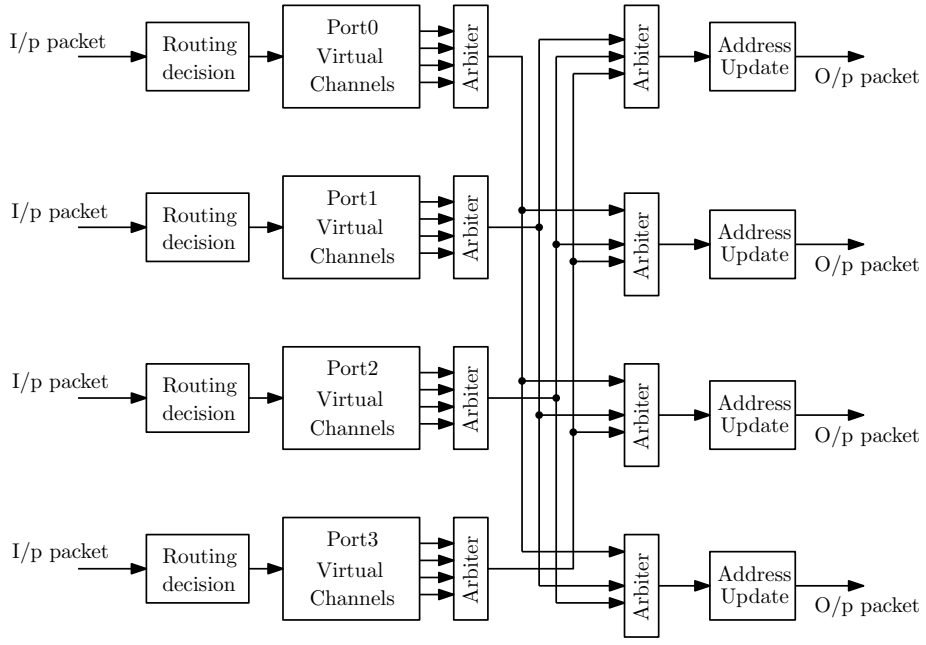


FIGURE 3.3: Router

3.1 Modifications done to reduce the FPGA area utilization

LWMU

The storage unit and the FIFO at read port of the storage unit in the figure 3.1 are replaced with a simple dual-port BRAM.

ALU

- DSP48E1 slices are used to implement 32-bit arithmetic and logical operations. The synthesis tool infers DSP48E1 slice from RTL code with $*$ operator. For other arithmetic and logical operations DSP48E1 is instantiated using the macro provided in the Xilinx library.
- The ASIC design of field multiplier functional unit does field multiplication in 3-modes (8, 16 and 32-bit modes), Barrette reduction, shifting and rotating operations. The resource utilized by field multiplier designed for ASIC on Virtex-6 FPGA is 1662 LUTs. It is redesigned to reduce the LUT count at the cost of increased cycles per operation. The FPGA design of field multiplier does field multiplication in 3-modes and Barrette reduction, a separate functional unit is designed to do shifting and rotation operation. Table 3.1 shows the comparison of field multiplier design for ASIC and FPGA.

Table 3.1: Comparison of ASIC and FPGA designs of Filed Multiplier

	ASIC design	FPGA design
LUT Utilization	1662	410 (FM + Shifter)
Cycles/Operation		
Shifting	1	1
Rotation	1	1
FM 8-bit Mode	1	4
FM 16-bit Mode	2	8
FM 32-bit Mode	4	16
Barrette Reduction	4	16

- In Verilog, a case statement with address as the selector expression and constants assigned to those address will infer a distributed ROM in FPGA design. Example

3.1 shows Verilog snippet that will infer 256 × 8-bit ROM when synthesized on a FPGA.

```
always@(sbox_lut2_arg$wget)
  begin
    case (sbox_lut2_arg$wget)
      8'd0: sbox_lut2_res$wget = 8'h00;
      8'd1: sbox_lut2_res$wget = 8'h01;
      8'd2: sbox_lut2_res$wget = 8'h8D;
              .......
      8'd254: sbox_lut2_res$wget = 8'h41;
      8'd255: sbox_lut2_res$wget = 8'h1C;
    endcase
end
```

Example 3.1 - Verilog HDL construct that infers ROM

The Bluespec generated Verilog will not assign constants to the address, if constant and address are same. The Verilog snippet below is the Bluespec generated Verilog, which will not infer a ROM when synthesized on a FPGA.

```
always@(sbox_lut2_arg$wget)
  begin
    case (sbox_lut2_arg$wget)
      8'd0,8'd1: sbox_lut2_res$wget = sbox_lut2_arg$wget;
      8'd2: sbox_lut2_res$wget = 8'h8D;
              .......
      8'd254: sbox_lut2_res$wget = 8'h41;
      8'd255: sbox_lut2_res$wget = 8'h1C;
    endcase
end
```

Example 3.2 - Bluespec generated Verilog that implements a ROM functionality

Though the functionality implemented by Verilog constructs in example 3.1 and 3.2 is same, design with example 3.1 will give better area and timing performance when

synthesized on a FPGA. The AES custom functional unit in ALU has such ROM implementation. For better performance the Bluespec generated Verilog has to be modified as in example 3.1.

Router

To support deadlock free, the router uses virtual channels (VCs). Each input port of the router has 4-virtual channel. Each virtual channel is implemented as FIFO of depth 4. The incoming packet includes the destination VC number. Depending on availability of packet in the VC and the availability of the destination port, one of the VC is selected and its packet is sent to the output port. At any point of time, data is written to one of the VCs and read from one of them. So instead of four memories and four FIFO write and read control units, same functionality can be implemented using one memory and four FIFO read and write control units which is more area efficient. Figure 3.4.(a) show the implementation of virtual channel 3.4.(b) show the area efficient implementation of virtual channel.

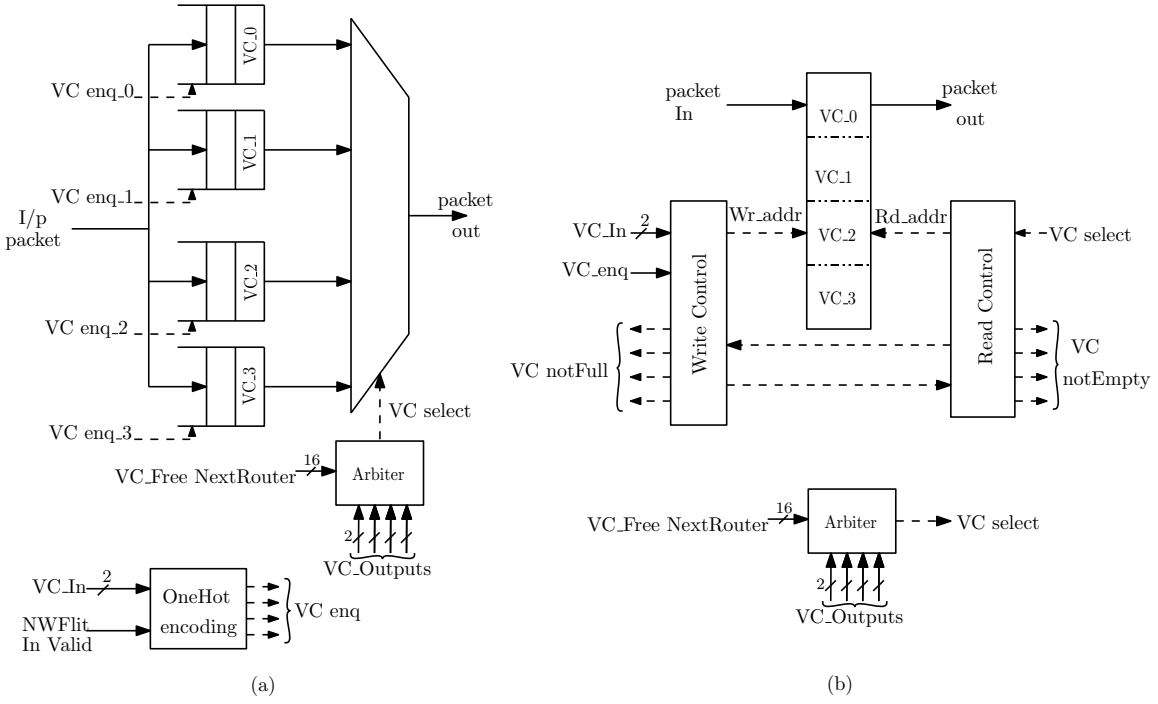


FIGURE 3.4: (a) Virtual Channel of the Router (b) Area efficient implementation of Virtual Channel

Chapter 4

Support Logic

Support Logic controls the resources on the Fabric, it comprises of Scheduler, HyperOp Launcher (HL), Resource Binder (RB) and Inter-HyperOp Data Forwarder (IHDF). This chapter covers the functionality of support logic components and the modification made to reduce the implementation logic of these components.

4.1 Scheduler

The primary task of the scheduler is to schedule the launching of a particular HyperOp on to the Fabric. The scheduling is achieved in a data driven manner, based on the availability of the input operands of the HyperOp. The Scheduler includes a Global Wait Match Unit (GWMU) which holds the operands for all HyperOp instances that are yet to be executed. Figure 4.1 shows the block diagram of a Scheduler. The GWMU unit is organized as a set of lines, where each line can accommodate the maximum possible inputs for a HyperOp instance. A HyperOp instance does not have an existence in the global wait-match unit prior to the arrival of its first data input. A HyperOp instance ceases to exist within the GWMU after it has been scheduled. Scheduler Lookup table (Scheduler LUT) memory module stores the expected number of inputs for all HyperOps, HyperOp type and the HyperOp predicate requirement, this data is generated by the compiler and called as HyperOp metadata. Scheduler fetches HyperOp metadata, when it receives the first input of the HyperOp from IHDF and stores them in tracker bits. On the arrival of each input operand, the count field in the tracker bits is updated. When

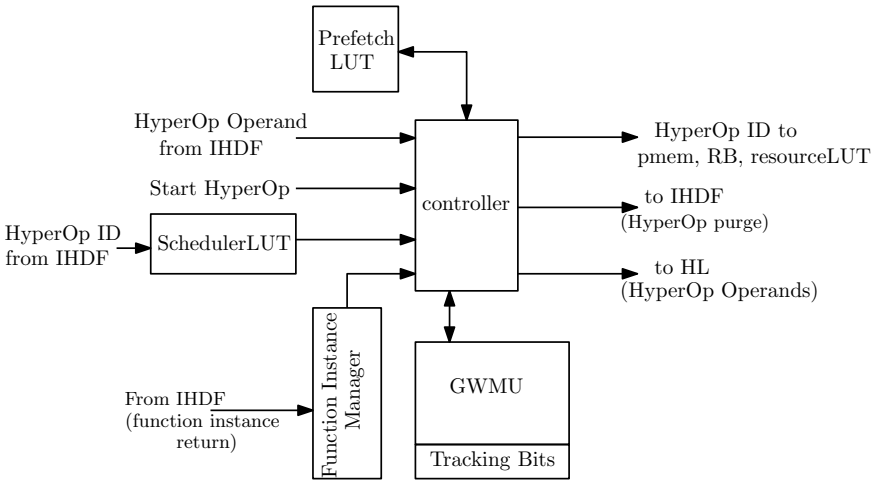


FIGURE 4.1: Scheduler

the count becomes equal to the expected count then the HyperOp instance is considered ready to launch. The controller consults the tracker bits to determine the total number of inputs expected for the HyperOp. HyperOp selection logic, which is implemented as a multistage priority encoder selects one the HyperOps that can be scheduled and sends HyperOp ID to RB, pMem and Resource LUT and HyperOp input operands to HL. The function instance manager dynamically allocates a tag for the HyperOp if it is of "Function HyperOp" type, this tag becomes part of the HyperOp instance. Once the function return is indicated by the IHFD, the instance number of the corresponding function is freed by the function instance manager. To reduce the launching time of an HyperOp and scheduler predicts the next HyperOp that may be launched based on the current HyperOp. This is implemented as static prediction based on the profiling. The prediction data is preloaded into prefetch LUT. Prefetch LUT returns the next HyperOp ID on providing the current HyperOp ID. Scheduler forwards this HyperOp ID to RB and Resource LUT and a flag bit to HL indicating HyperOp scheduling in predict mode. If the scheduler finds that the next HyperOp is the predicted HyperOp, it sends a request to the RB to confirm the speculative placement and forwards input operands of the HyperOp to HL. If the prediction was wrong then scheduler sends "squash" request to the RB to squash the allocation of previous HyperOp.

4.1.1 Modifications done to reduce the FPGA area utilization

The current version of REDEFINE has 64 lines in the GWMU. The size of tracking bits for this GWMU are

- HyperOp ID : 64×10 - bits
- HyperOp Instance : 64×13 - bits
- Predicate : 64×1 - bits
- HyperOp Operand Count : 64×4 - bits
- HyperOp MetaData : 64×9 - bits
- Ready Status : 64×1 - bits
- Entry Valid : 64×1 - bits

These tracking bits are implemented as registers (has reset input). The GWMU holds the operands of the HyperOp and the tracking bits holds the same HyperOp's additional information (HyperOp ID, instance number, predicate expected, operand count in GMU, ready status, validity and metadata). Figure 4.2 shows the selection logic implementation

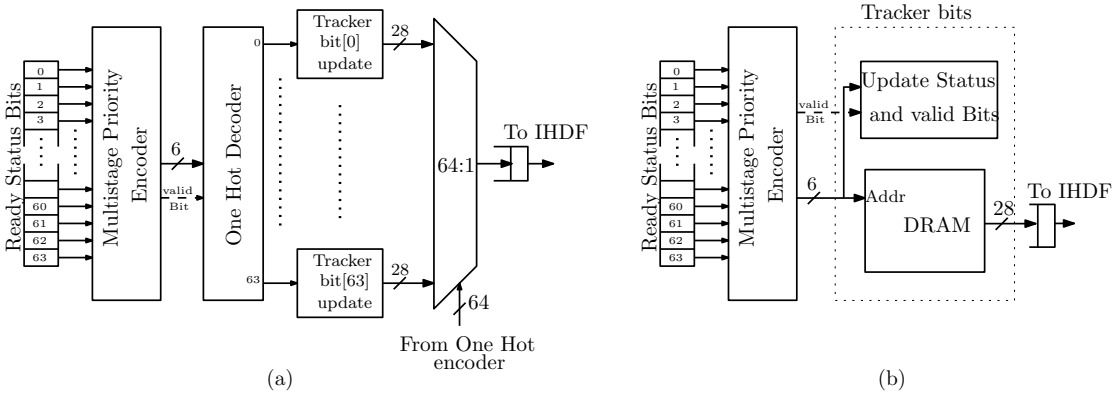


FIGURE 4.2: (a) Selection logic (b) Area efficient implementation of selection logic

in the scheduler. "Ready Status" bits indicates that the HyperOp in the corresponding scheduler line is ready to launch. The multistage priority encoder selects one of the HyperOp among the ready HyperOps. The selected HyperOp ID and HyperOp instances are send to the IHDF, which informs that the corresponding HyperOp is launched and its

translation can be removed. This is implemented using 64:1 28-bit multiplexer described using parallel case statement as shown in figure 4.2.(a). Except "Ready Status" and "Entry Valid" bits, if these tracking bits are implemented as distributed RAM using LUTs, the same functionality can be generated without this huge multiplexer.

4.2 HyperOp Launcher

HL transfers the instructions, constants, control packets, and input operands of a HyperOp on to the Fabric for execution. Figure 4.3 shows the block diagram of HL and its interfaces for the Fabric of size 6×6. HyperOp is the atomic entity of execution and each HyperOp

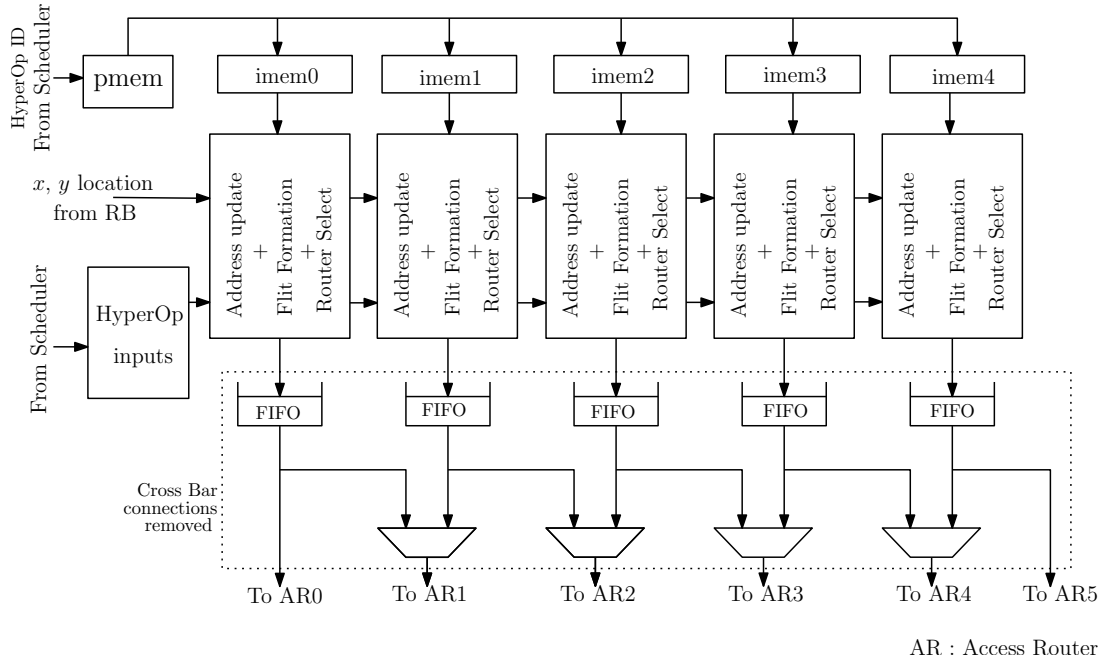


FIGURE 4.3: HyperOp Launcher

can occupy maximum of 25 (5×5) CEs on Fabric in the present version of REDEFINE. Each HyperOp is partitioned into partial HyperOps (p-HyperOps) and each p-HyperOp is configured to one CE in the Fabric. The resources (CEs) required to synthesize a HyperOp on the Fabric is calculated at compile time and stored in the memory called Resource Lookup Table (Resource LUT) as HyperOp configuration matrix which is of size 5×5 (should be less than or equal to Fabric size). Each element in this matrix represents one CE. The instructions, constants and control inputs to the HyperOp are generated at compile time and stored in "imems". The 5 (should be equal to row size

of HyperOp configuration matrix) imems stores information required for the CEs in each row of the HyperOp configuration matrix. The "pmem" provides the starting address and the number of packets to be read from imems for each HyperOp. If the scheduling is in pre-fetch mode the HL receives only instructions from imems. Scheduler sends the input operands of the HyperOp to HL only when HyperOp scheduling is confirmed or scheduled in non-pre-fetch mode. RB sends the position of CE corresponding to the $(0, 0)$ element in the HyperOp configuration matrix on Fabric. The HL updates the destination address of the received packets with the (x, y) coordinates received from the RB and launches them on to the Fabric through the access router nearest to the destination.

4.2.1 Modifications done to reduce the FPGA area utilization

The figure 4.3 includes the modification. In the previous implementation, HL has a crossbar connections between Access Routers and imems as shown in the figure 4.4. As

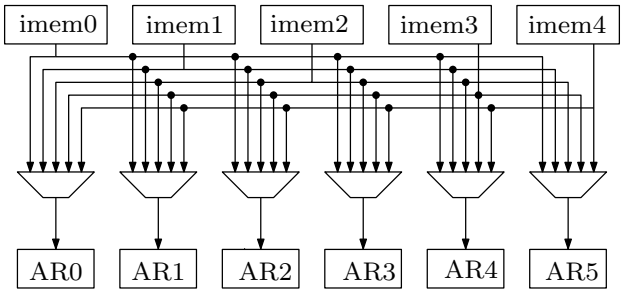


FIGURE 4.4: HL crossbar connections to Access Routers for the Fabric of size 6×6

current implementation of RB doesn't perform wrap search in the Fabric there is no need for this crossbar connection. Without wrap search the maximum number of Access Routers to which an imem can connect is *(no. of Access Routers - no. of imems) + 1*. For the Fabric of size 6×6 (6 Access Routers) and configuration matrix with 5 rows (5 imems), this is equal to 2. By removing this crossbar connection, five 5:1 76-bit multiplexers are replaced with four 2:1 76-bit multiplexers. A parametrized BSV code is written to implement HL without crossbar connections for any size of Fabric and configuration matrix.

4.3 Inter-HyperOp Data Forwarder

The IHDF enables exchange of data between two HyperOps. It is responsible for

1. Computing the tag for the destination HyperOps i.e. HyperOp Instance.
2. Keeps track of the location of the HyperOp so as to forward the results to it.
3. Store loop invariants (template data) and deliver them as and when needed.

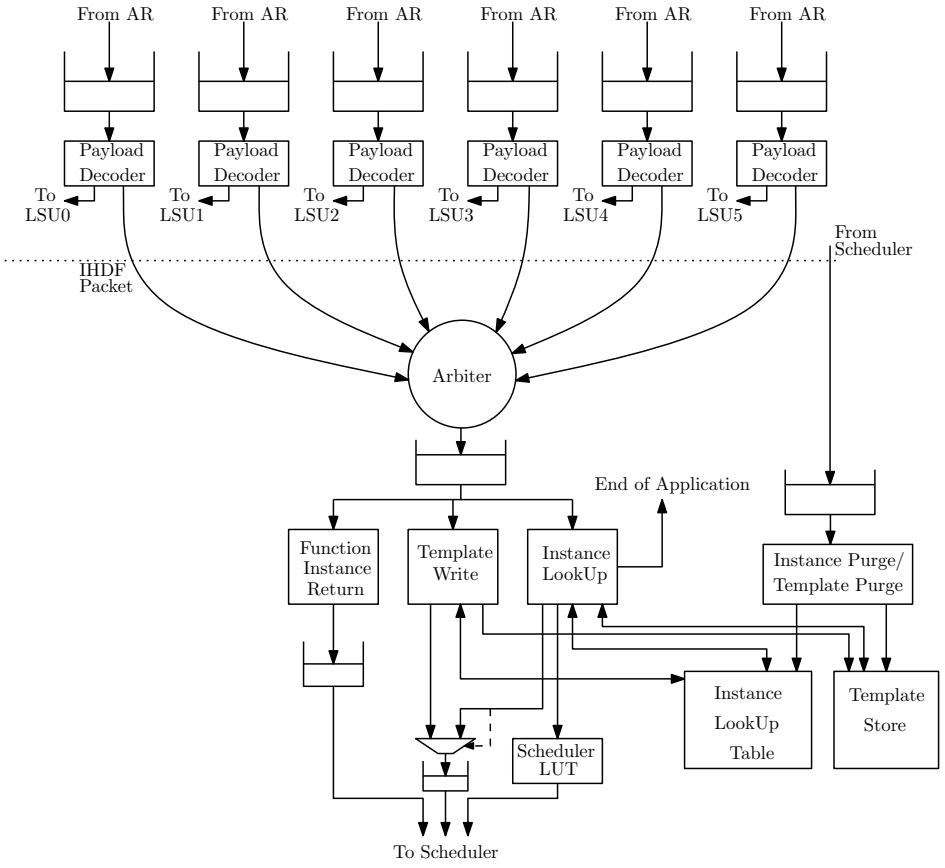


FIGURE 4.5: Inter-HyperOp Data Forwarder

Figure 4.5 shows the block diagram of IHDF. IHDF receives input operands for the HyperOps from Fabric and forwards them to Scheduler. IHDF computes the instance of the destination HyperOp using the producer instance number and a compiler hint. Based on the HyperOp ID and the HyperOp instance IHDF performs a lookup in Instance Lookup Table (Instance LUT) to find the location allocated for this HyperOp instance in the GWMU. If a match is found data is forwarded to the identified location in the

GMWU. If the match is not found a new entry is allocated and data is forwarded to that location in GWMU.

If the data received is template data it is stored in Template Store.The Template Store identifies the HyperOp Id and the instance number pattern of the HyperOps which are expected to consume this data. Whenever the IHDF encounters a data transfer for a HyperOp that matches the said ID and whose instance number matches the pattern determined by the compiler, this template data is transferred to the global wait match unit.

Other than these two, IHDF receives function return request. On receiving function return request IHDF informs the scheduler to free the corresponding function instance. When a HyperOp is scheduled, then the Scheduler sends the HyperOp ID and instance to the IHDF to invalidate the translation maintained for that HyperOp in Instance LUT and Template Store.

4.3.1 Modifications done to reduce the FPGA area utilization

The modules Instance Lookup Table and Template Store are organized as caches. Instance Lookup Table is 4 way set associative cache and Temple Store is a directed mapped cache. Memory with Tag and Data fields are implemented using BRAM and valid bits are implemented using registers. Both have 1024 cache lines. To check the validity we access 4 valid bits corresponding to the cache line in Instance LUT and 1 valid bit in Template Store which requires 1024:1 4-bit multiplexer and 1024:1 1-bit multiplexer respectively. In BSV the Instance LUT valid bit registers are described as a register of size 4096 bits, one 4096-bit left-shifter and one 4096-bit right-shifter are implemented to read these valid bits. Instead it should be described as vector of 1024 registers of size 4-bits to implement a 1024:1 4-bit multiplexer. Similar with the Template Store too.

4.4 Resource Binder

The resource binder determines the location on the fabric where the HyperOp is to be placed. The HyperOps are dynamically placed to enable multiple applications to run

simultaneously on the fabric. Fixing the location at compile time would allow only co-compiled applications to run simultaneously. Every HyperOps resource requirement is available in the Resource LUT. The requirement clearly specifies the number of CEs and their relative position with regard to each other. The requirement is specified in the form of a binary matrix. A 1 indicates that the CE at that position is desired. This matrix is smaller than the fabric size. The current matrix size used is 5×5. The resource binder tries to match the resource requirement with the CE availability matrix also referred to as the fabric status. When a match is found, the resource binder returns the location in the fabric status matrix where (0, 0) location of the resource requirement matrix matched. Figure 4.6 shows the block diagram of RB. If the Fabric is heterogeneous i.e., some of the CEs have special resources then HyperOps with some special functionality are synthesized on these special CEs. In such case Resource LUT provides the location on the Fabric where the HyperOp is to be launched. If the request from Scheduler is to "squash" the previously launched HyperOp, RB sends reset to all the CEs of the previously HyperOp once they have received the all instructions. This makes sure that the squash HyperOp instructions doesn't sit in LWMU of the CEs.

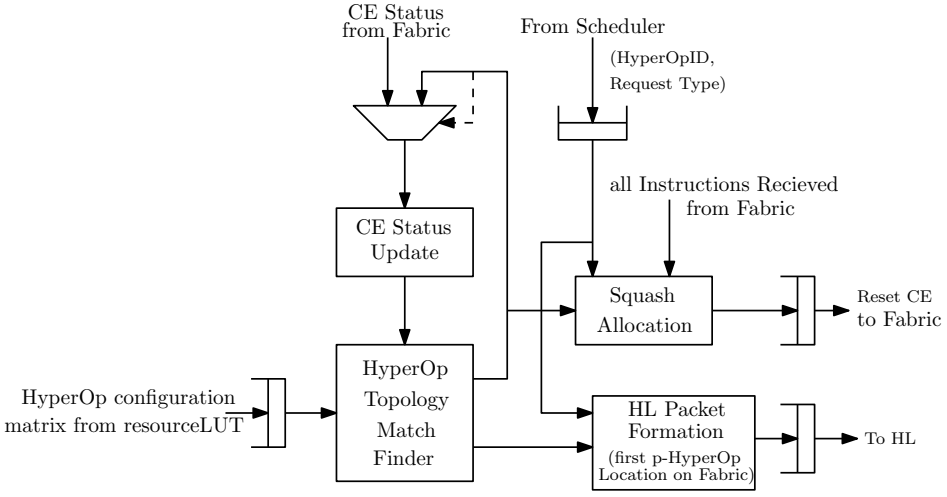


FIGURE 4.6: Resource Binder

4.4.1 Modifications done to reduce the FPGA area utilization

The number of row searches that RB need to perform is equal to *(no. of Fabric rows - no. of HyperOp configuration matrix rows) + 1.* The variable that holds this data should be of size $log_2((no.of Fabric rows - no.of HyperOp configuration matrix rows) + 1)$. For the

current implementation this count is 2 and the variable that holds this count should be of size 1-bit. But it was declared as a 3-bit size variable. In BSV as the search algorithm was described using loops this implements 8-searches which is redundant. Thus this variable is changed to 1-bit size.

The notation followed by the RB to address a CE on Fabric is as follows *($y \times$ no. of rows in the Fabric + x)*, where x and y numbers are column and row positions of CE respectively both starting from 0. The search algorithm returns the row (y) and column (x) position of the CE corresponding to (0,0) element in HyperOp configuration matrix. Once the RB finds a location on the Fabric, it should declare the CEs that are assigned to the HyperOp as busy. Due to the above notation used, multipliers are used to calculate the position of the CE as these x and y values are not compiler time constants. This can be avoided if the CEs are addressed using two dimensional notation.

Chapter 5

Summary and Results

The on-chip memories of the REDEFINE are implemented using single port block RAMs with read and write on the same port. The address width of imems has be reduced from 17-bits from 10-bits. Similarly address width of LSU has be reduced from 17-bits to 10-bits. Table 5.1 shows different on chip memories and the BRAM resource utilization, this doesn't include the BRAMs used in the LWMU of the CEs. All the BRAMs are inferred using the coding techniques and the synthesis attributes. DSP48E1 is the only FPGA resource that has been instantiated in the design. The instantiation done by importing the DSP48E1 primitive macro to BSV. Each CE in the design utilizes 4 DSP48E1

Table 5.1: REDEFINE memories and their BRAM resource utilization

Memory	Address width (bits)	Data width (bits)	Banks	BRAMs (18 Kb)
imem	10	66	5[1]	5×4
LSU (dmem)	11	4×8	6[2]	6×4
Instance LUT	10	19	1	2
prefetch LUT	10	11	1	1
Resource LUT	10	32	1	2
Template Tracker	10	23	1	2
Template Store	10	461	1	26
Scheduler LUT	10	8	1	1
Scheduler Memory (GWMU)	6	14×32	1	14

slices, 3 are inferred for 32-bit multiplier and 1 is instantiated to perform 32-bit addition,

[1]depends on HyperOp configuration matrix row size
[2]depends on no. of Access Routers in the Fabric

subtraction, comparison and logical operations. For the current Fabric size (6×6), 120 DSP slices are used.

Using the methods discussed in Chapter 2, 3 and 4 significant reduction in the resource utilization has be obtained. Table 5.2 shows the comparison of number of slice LUTs used for the implementation of individual modules of REDEFINE with out without optimization.

TABLE 5.2: Comparison of LUT Utilization of individual modules of REDEFINE with and without optimizations

LUT Utilization	Before Optimization	After Optimization
ALU	12085	2975
Transporter	547	206
LWMU	1628	1194
CE	14432	3912
Router	3344	1135
Scheduler	—	1946
HL	4425	1881
IHDF	25967	8082
RB	1423	818

5.1 Conclusion

High-level synthesis tools allows designers to write high-level behavioral description of the design, still the designer should be the completely aware of hardware generated for the given design. Synthesis tools are able to infer and map complex arithmetic and memory descriptions onto the dedicated hardware blocks of particular FPGA device. However given a particular RTL description, there is only so much the tools can do to maximize performance. If high performance is needed in a design proper coding of the design should be followed.

Bibliography

[1] Alexander Fell, Mythri Alle, Keshavan Varadarajan, Prasenjit Biswas, Saptarsi Das, Ju- gantor Chetia, S.K. Nandy, and Ranjani Narayan. Streaming FFT on REDEFINE-v2: an application-architecture design space exploration. In *CASES 09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 127136, New York, NY, USA, 2009. ACM.

[2] Mythri Alle, Keshavan Varadarajan, Alexander Fell, Nimmy Joseph, C.Ramesh Reddy, Saptarsi Das, Prasenjit Biswas, Jugantor Chetia, S. K. Nandy, and Ranjani Narayan. REDEFINE: Runtime Reconfigurable Polymorphic ASIC. *ACM Transactions on Embedded Computing Systems, Special Issue on Configuring Algorithms, Processes and Architecture*, 9(2), September 2009.

[3] Rishiyur S. Nikhil, "System Verilog Tagged Unions and Pattern Matching" An extension to System Verilog 3.1 proposed to Accellera, October 2003.

[4] Bluespec System Verilog Reference Guide, Revision: October 2009, pages 121,149.

[5] Bluespec System Verilog Reference Guide, Revision: October 2009, pages 122,170

[6] Clifford E. Cummings, Sunburst Design, Inc., "full_case parallel_case", the Evil Twins of Verilog Synthesis, SNUG99 Boston.

[7] Virtex-6 FPGA Configurable Logic Block UG364(v1.1), Xilinx Corporation, 2009

[8] ASIC to FPGA Design Methodology & Guidelines, AN-311-1.0, Altera Corporation, 2003.