1-1-2000

Partitioning of large HDL ASIC designs into multiple FPGA devices for prototyping and verification

Nilesh V Dhavlikar
University of Nevada, Las Vegas

Follow this and additional works at: <https://digitalscholarship.unlv.edu/rtds>

Repository Citation

Dhavlikar, Nilesh V, "Partitioning of large HDL ASIC designs into multiple FPGA devices for prototyping and verification" (2000). *UNLV Retrospective Theses & Dissertations*. 1240.
<http://dx.doi.org/10.25669/jrm5-17mc>

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

PARTITIONING OF LARGE HDL ASIC DESIGNS INTO MULTIPLE FPGA
DEVICES FOR PROTOTYPING AND VERIFICATION

by

Nilesh Dhavlikar

Bachelor of Science in Electrical Engineering
University of Pune, India
1998

A thesis submitted in partial fulfillment
of the requirements for the

Masters of Science Degree in Electrical Engineering
Department of Electrical and Computer Engineering
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2001

UMI Number: 1405097

Copyright 2001 by
Dhavalikar, Nilesh V.

All rights reserved.

UMI[®]

UMI Microform 1405097

Copyright 2001 by Bell & Howell Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

March 27, 2001

The Thesis prepared by

Nilesh V. Dhavlikar

Entitled

Partitioning of Large HDL ASIC Designer into Multiple FPGA Devices for
Prototyping and Verification

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

ABSTRACT

Partitioning Of Large HDL ASIC Designs Into Multiple FPGA Devices For Prototyping And Verification

by

Nilesh Dhavlikar

Dr. Henry Selvaraj, Examination Committee Chair
Professor of Electrical Engineering
University of Nevada, Las Vegas

The ASIC (Application specific Integrated Circuit) designs grow continuously bigger and bigger. This causes dramatic increase in the simulation run time. It is very hard to simulate these designs because the simulation time has risen from hours to days and weeks. Hardware Embedded Simulation (HES) is a technology that facilitates incremental design verification of ASICs. The FPGAs (Field Programmable Gate Arrays) can play an important role in ASIC design cycle. But it is not possible to fit an entire ASIC design into a single FPGA device. This problem can be solved by partitioning the given design into multiple small size designs (modules) and fitting those modules into multiple FPGAs. The purpose of my thesis is to take a large RTL (Register Transfer Level) design of an ASIC into consideration, write and test the software ("C" code) practically to synthesize each top level module and analyze the size of each module in terms of number of CLBs (Configurable Logic Blocks), I/Os, flip-flops, latches and apply the algorithm to partition it automatically into minimum number of FPGAs.

TABLE OF CONTENTS

ABSTRACT	iii
LIST OF FIGURES	iv
CHAPTER 1 - INTRODUCTION TO HES	1
Emulation Methodologies	2
HES- PCI Board	4
Partitioning	5
Synthesizing the ASIC design	6
CHAPTER 2 - INTEGRATED CIRCUITS DESIGN OVERVIEW	8
CHAPTER 3 - SYNTHESIS PROCEDURE USING FPGA EXPRESS.....	10
FPGA Express functions	12
DPM Interface To FPGA Express.....	13
CHAPTER 4 - PARTITIONING ALGORITHM.....	14
CHAPTER 5 - IMPLEMENTING THE ALGORITHM PRACTICALLY	20
CHAPTER 6 - SUMMARY, CONCLUSIONS AND RECOMMENDATIONS.....	37
APPENDIX	38
BASIC CONCEPTS OF VERILOG	38
FPGA EXPRESS PROJECT REPORT	56
BIBLIOGRAPHY	65
VITA	66

LIST OF FIGURES

TABLE 5.1	Results Generated after synthesizing Individual sub-modules	35
TABLE A.1	Primitive reference Count.....	57
TABLE A.2	Clocks	58
TABLE A.3	Timing Groups.....	58
TABLE A.4	Input Port Timing	58
TABLE A.5	Output Port Timing.....	62

ACKNOWLEDGEMENTS

It is a great pleasure for me to acknowledge the people who have helped me during the course of my thesis work. My special thanks to my advisor, Dr. Henry Selvaraj who has not only helped me but also guided me in a right direction. Because of his invaluable contribution towards my research topic I have come up with a totally new solution in Hardware Embedded Simulation world. At the same time I would like to thank my parents and my sister who have continuously boosted my morale over the period of 2 years. Otherwise it would have been very difficult for me to achieve this goal when staying very far away from my family. I would specially acknowledge Mr. Gregor Siwinsky of ALDEC, INC. for his contribution in giving me this topic and guiding me throughout my entire successful journey especially while generating the software.

CHAPTER 1

INTRODUCTION TO HARDWARE EMBEDDED SIMULATION (HES)

ASIC (Application Specific Integrated Circuit) designs are growing in size and complexity continuously. Typical ASIC design cycle stretches for a minimum of 3 to 4 years. Design verification (that includes functional and timing verification) of device takes the major part of this cycle. The increase in design size causes dramatic increase in the simulation run time. It is more and more hard to simulate today's designs in a short period of time because the simulation time has risen from hours to days and weeks. Hardware Embedded Simulation is a technology that facilitates incremental design verification of ASICs using FPGA (Field Programmable Gate Array) devices.

Emulation Dilemma

Emulation, often referred to as prototyping, has emerged as a major ASIC verification technology where speed of verification is of a primary consideration. Weeks and days of simulation time are reduced to hours and minutes. The emulation methodology uses commercially available Field Programmable Gate Arrays (FPGAs) or custom processors to duplicate ASIC design. Emulation is the only verification approach that can attain the speed required for verification of an ASIC design in the real operating environment. Testing in the real environment (in circuit) rather than a simulated environment significantly increases the probability that the device will perform as required.

Emulation Methodologies

The emulation market has evolved into two methodologies. "Black Box" Emulation e.g. Hardware/software systems such as Quickturn and Ikos designed specifically for emulation consisting of hundreds of FPGAs or custom processors prepackaged with proprietary software that programs and interconnects the components. Open System Emulation – Commonly referred to as "roll your own" emulation uses off the shelf FPGAs to duplicate or emulate the logic. These devices are then interconnected via custom printed circuit boards. The advantages of the two approaches are as follows:

Black Box Emulation – Although requiring significant expertise and support, it is a mature technology used since the late 1980's. It is perceived to be relatively automatic and turnkey because of well defined flow with a reasonable level of observability for debugging purposes.

Open System Emulation - Speed, the primary objective of emulation, is in most cases, an order of magnitude higher than Black Box Emulation. Hardware/software costs are at least an order of magnitude less than Black Box Emulation. It is often necessary to provide multiple copies for developing software or for sending to potential customers to test the ASIC in their environment. Unlike Black Box Emulation, copies or replicates are relatively inexpensive and easy to ship. Capacity and speed of new FPGAs increases at the same rate as ASICs. Open System Emulation can integrate advances in FPGA technology immediately thereby insuring scalability. Black Box Emulation can only integrate the latest FPGA technology at the beginning of the typical 1 to 3 year development cycle. Consequently, there is a high rate of obsolescence inherent to Black Box Emulation. Open System Emulation can be accomplished incrementally based on

verifying individual functional blocks with the results maintained as usable components. This facilitates a better flow for incremental changes or reusable logic. Black Box systems do not allow this level of granularity.

Neither emulation approach has attained mainstream verification status because: Black Box Emulation has a very high entry cost (over \$1M per system), high rate of obsolescence, relatively slow speed compared to Open System Emulation and the size and cost factor that make copies/replicates impractical. This along with the high level of expertise and support required for success has limited its use to only large projects with significant funding and no other alternatives. Since emulation rarely attains actual device speed, both Black Box and Open System Emulation have the problem of slowing down the rest of the system to the speed of the emulation prototype. There is typically a minimum threshold speed that the system can be reduced to. If the emulation prototype is slower than this threshold, "in-circuit" emulation is not an option thereby negating the major advantage of emulation. Since Open System Emulation typically runs faster than Black Box Emulation, the slow down problem is reduced and the speed is rarely below the threshold requirement. Even so the slow down problem is a major task for both methodologies. In spite of overcoming the major disadvantages of Black Box Emulation and reducing the going "in-circuit" speed issue, Open System Emulation is considered an extremely difficult technology to implement. The main difficulty is partitioning the logic among multiple FPGAs. Partitioning is a balancing act of trying to satisfy the fixed logic capacity and I/O constraints inherent to FPGAs. I/O distribution is by far the most challenging problem. The difficulty is in attaining full observability of all signals internal to the FPGA for debugging purposes. Normally all signal names change and others are

optimized out during the synthesis process. Prior to probing internal signals for debug purposes, the post synthesis signal name changes and absence of signals must be correlated with the original RTL signals that the design team is familiar with. The manual correlation process is long and tedious and many result in major bottlenecks.

HES - PCI board

The HES (Hardware Embedded Simulation) environment consists of software simulator and HES boards and is used for speedy design development and verification. This environment assures correct communication between modules located in silicon and in software simulator. HES is a flexible and easy to use for ASIC prototyping. If we verify any part of our design we can download it seamlessly into hardware that works on event-by-event basis with an RTL simulator.

1. If we use HES for ASIC designs, we need to convert RTL design into an FPGA code, using for example the Synplicity's Ceritify program.
2. HES can accommodate hardware models, which speeds the design verification.
3. HES can be used as functional simulation accelerator, providing typically 50 to 1000 times speed improvement over the best RTL simulators.
4. HES is very effective in IP core analysis and certification. It is easy to use.

We do not have to know how to place design modules in HES because the HES Wizard will guide us through each step of the way. We only need to decide which modules should be placed into the HES hardware and the rest will be done by the Wizard. Only synthesizable designs can be put into the HES hardware. However, if we are doing RTL designs, code should be synthesizable and ready for HES applications. If we have a

mixture of synthesizable and non-synthesizable code, the design will be split between HES (for synthesizable sections) and software simulator (for non-synthesizable) modules. The simulation will be somewhat slower but still we will reap the major benefits of HES technology such as instant design modifications, speedy checking of selected design paths, etc.. The first HES boards have been built with Xilinx FPGA Virtex devices. However, HES boards with Altera's CPLD Apex parts will appear shortly. Since the HES hardware implementation technology (FPGAs and CPLDs) is hidden from the user through the software layer, there should be no additional learning curve when switching from one HES technology to another.

Partitioning

Partitioning is the process of distributing the logic and I/Os among multiple FPGAs. The development of an RTL solution required major innovation. Early in the development cycle, it became apparent that the level of RTL granularity was not compatible with solving the I/O distribution problem and this restriction severely limited the efficiency of logic distribution. The task is one of grouping the most highly connected entities internal to the FPGA and the least connected entities between FPGAs. The smallest entity at the RTL level is a module. A module is a collection of processes and instantiations of other modules. Partitioning at the module level provides a limited number of solutions making it difficult to provide a balanced logic distribution; and in many I/O intensive cases, making it extremely difficult to find an I/O solution. Moving down to the next lower level, the gate level, provides too much granularity and the database size becomes unmanageable. The time required to solve the I/O distribution

becomes prohibitive. This is why other partitioning tools encountering I/O intensive designs go directly to a multiplexing scheme allowing a single pin to handle multiple signals. Multiplexing should be considered as the last resort to solving I/O distribution. Although it is an effective way to solve the problem, there is a severe speed penalty. The lower the emulation prototype speed, the more difficult going "in circuit" becomes. In most cases there is a threshold speed that must be attained to go in circuit. If the emulation prototype speed goes below the minimum threshold due to multiplexing, it is impossible to emulate in the external environment causing the emulation project to fail.

Synthesizing the ASIC design

Since the ASIC design has to be partitioned and put into multiple FPGAs, it is necessary to extract the information about overall size of the design. The size of the given design can be specified in terms of following parameters for each module (if the design is done using Verilog as the hardware description language) or the entity (if the design is done using VHDL) :

1. Total number of I/Os (Input Output devices).
2. Total number of CLBs (logic blocks).
3. Number of flip-flops
4. Number of latches.
5. Number of clocks in the design.

Use of a synthesis tool

All the above information about individual modules can be extracted only after synthesizing each module. Synthesis of given design means converting the given HDL

design into lower gate (RTL) level representation. Note that Hardware Description Languages like VHDL or Verilog are only meant for documenting the given design in a systematic higher level language format.

It is the job of the synthesis tool to express the design at the gate level representation. FPGA Express is a complete FPGA logic-synthesis and optimization tool developed by Synopsys company. With the help of this tool we can create optimized FPGA netlists from VHDL or Verilog HDL code.

Once all the parameters about the individual modules are extracted, partitioning algorithm needs to be applied to fit the given design into multiple FPGAs. The partitioning decision should be taken automatically. Once the algorithm is applied, again using the FPGA Express tool the individual FPGAs are synthesized and top level netlist is generated to connect all the modules in the given design.

CHAPTER 2

INTEGRATED CIRCUITS DESIGN OVERVIEW

Integrated circuit Design has always been a dynamically changing process due to rapid advancement in technology upon which the design process strongly depends.

Application specific Integrated Circuits (ASICs) are designed by the user himself. They have the big advantage that a special function can often be realized with fewer components by skillful integration of elements on one or more dies. The implying enhancement of integration density causes lower system costs, lower power consumption, higher speed, lower weight/ volume and lower risks for failures. Because of these advantages, the share of ASICs is constantly rising in the world sales of integrated circuits.

Top Down design

Instead of trying to implement the design of a large system all at once , a divide and conquer strategy is taken in a top down design process. Top down design is referred to as recursive partitioning of a system into sub-components until all the sub-components become manageable design parts. When the recursive partition procedure completes its job, a partition tree becomes available. After the completion of this top down design process, the bottom up implementation phase begins. In this phase, hardware components

corresponding to the terminals of the tree are recursively wired to form the hierarchical wiring of the complete system.

Figure 2.1: Top - down design , bottom-up implementation .(SUD = system under design, SSC= system, sub-components).

VHDL and VERILOG are the most popular hardware description languages to describe any digital hardware. Verilog is more popular for ASICs while VHDL is for FPGAs. The design is always specified in hierarchical format. The whole design consists of multiple top level modules and sub-modules. Since it is difficult to fit the whole design into a single device , partitioning needs to be done. The algorithm takes into consideration only the top level since the synthesis is done taking into consideration the sub-hierarchy.

CHAPTER 3

SYNTHESIS PROCEDURE USING FPGA EXPRESS

During the synthesis flow, FPGA Express performs the following steps:

We use the integrated text editor for entering VHDL and Verilog HDL source code for our design. We can also use the text editor in the analysis step (described next) for easy debugging of design source files.

1. Analyze HDL (VHDL and Verilog HDL) design source files for correct syntax using the Synopsys industry-standard HDL language support.
2. FPGA Express accepts any combination of VHDL, Verilog HDL, and FPGA netlist files as sources for a design. For example, we can use functions or subdesigns created through schematic capture and Verilog within a VHDL top-level design. After we add the design source files, FPGA Express automatically analyzes the HDL files. If the source files contain errors, the Output window and text editor help us find and correct the problems. FPGA Express synthesizes the logic of our design, using architecture-specific algorithms to target devices. In this part of the design flow, the program elaborates each design module and creates and links the design hierarchy to form a unique design implementation. After this step, FPGA Express generates schematics of our design, so we can verify the function and timing of our circuits.

3. Optimize logic for speed and area as directed by our design constraints, generating an FPGA netlist file that is ready for place and route. When it has completed optimization, the program generates a netlist ready for place and route. FPGA Express also creates reports of its results.
4. After this optimization, FPGA Express again generates schematics of our design, so we can verify the function and timing of our circuits.
5. Extract and display accurate post-synthesis delay information for timing analysis and debugging. FPGA Express displays timing information beside our design constraints and highlights timing violations. This information is linked directly to the schematics, so we can easily decide whether to place and route the design or make design changes.

Entering Designs into FPGA Express:

FPGA Express supports the following methods of describing FPGA designs:

HDL-based design methodology uses only HDL source code for design entry.

We can enter one or more VHDL or Verilog HDL files describing any hierarchical design structure. We can then split the design into hierarchical functional blocks or create a single flattened design description. This feature makes it easy to reuse modules from a common design library or design source.

Schematic-based design methodology uses schematics for design entry. This design methodology adds only one step to a traditional FPGA design process, but it can produce improved area and performance. Mixed design methodology uses a combination of HDL source code and schematics for design entry.

In this design methodology, the HDL source code is synthesized, combined with netlists from schematic entry, and optimized into a netlist ready for FPGA place and route tools. FPGA Express allows us to divide a design between HDL and schematic input in any proportion and create virtually any design hierarchy. We can also use functional blocks made from both schematics and HDL to reuse modules from common design libraries or design sources.

FPGA Express Functions

FPGA Express creates optimized FPGA netlists from VHDL code, Verilog HDL code, and existing, unoptimized netlists in the following design flow:

1. Analyzes VHDL and Verilog HDL source files for correct syntax using the Synopsys industry-standard HDL language policy.
2. Elaborates logic from VHDL, Verilog HDL, and FPGA netlist source files, targeting a specific FPGA architecture and device.
3. Optimizes logic for speed and area as directed by our design constraints, generating an FPGA netlist file ready for place-and-route.

FPGA Express accepts any combination and mix of VHDL, Verilog HDL, and FPGA netlist files as input sources for a single design. For example, we can use functions or subdesigns that are created in schematic capture and Verilog HDL within a VHDL top-level design, and vice versa. After we add the design sources, FPGA Express analyzes HDL files. If we have errors in the source files, the output window and text editor help we to find and correct problems.

Next, FPGA Express elaborates the logic for our design, using architecture-specific algorithms. During this part of the design flow, each design module is elaborated and the design hierarchy is created and linked to form a unique design implementation.

FPGA Express optimizes the design as directed by our design constraints. With the graphical user interface (GUI), we can enter constraints for our design in editable tables. The constraints contain performance requirements and optimization options for the architecture-specific optimization engines.

When it has completed optimization, FPGA Express generates a netlist ready for place-and-route. FPGA Express also creates Verilog and VHDL netlists for functional simulation, and reports that document its results. FPGA Express extracts and displays accurate post-synthesis delay information for timing analysis and debugging. FPGA Express displays timing information beside our design constraints so we can make quick, informed decisions.

DPM interface to FPGA Express:

The DPM package implements a set of API's for DPM partners to access the synthesis and optimization capability of FPGA Express. The major functionalities exported through this API are project mangement, constraint management, and error, warning and message management.

CHAPTER 4

PARTITIONING ALGORITHM

The given ASIC design is described using VHDL or Verilog as the Hardware Description Language. The partitioning algorithm takes into consideration only those designs which are fully synthesized by the FPGA Express tool.

Synthesizing the design before partitioning:

The given ASIC design consists of number of modules and sub-modules which are described using VHDL or Verilog. The design modules are arranged in a hierarchical way. It is not possible to fit all the modules into a single FPGA device because of the tremendously large size of the given ASIC design. Partitioning needs to be done to fit these modules into multiple FPGA devices. The FPGA express is invoked using a C code that will look for a correct DPM interface.

The following information needs to be supplied in the command line while executing the C code:- The project and design name, the list of all HDL files (.v for Verilog files and .vhd for VHDL files.) for the top level module and all the sub-modules connected to the same module and arranged in a hierarchical way and also specify the target vendor, the family and the corresponding device. In this “C” code Synposys defined FPGA Express functions are invoked to perform the following tasks: in FPGA Express, set up the design and analyze the source files. Analyzing the files involves checking the syntax

up the design and analyze the source files. Analyzing the files involves checking the syntax for VHDL or Verilog HDL language and to report errors/warnings.

1. once the library and the design name is provided and if there is no syntax error, then the tool accepts the target device and synthesizes as well as optimizes the HDL source code and captures netlists to produce an optimized EDIF or XNF netlist for placement and routing.
2. (optional) generate a Verilog or VHDL netlist for functional simulation.

Once the synthesis stage is over, the following parameters are derived for individual modules from project report generated by the synthesis tool.

1. Total number of I/Os (Input Output devices),
2. Total number of CLBs (logic blocks),
3. Number of flip-flops,
4. Number of latches,
5. Number of clocks in the design.

The aim of partitioning algorithm is to find the optimum number of devices and divide the entire logic evenly among them. The algorithm can be described as follows:

1. basically total number of I/Os, CLBs, Flip-flops and latches basically decide the number of FPGA devices needed to fit the entire ASIC design.
2. the given ASIC design consists of various modules and sub-modules arranged in a hierarchical manner. The top level connects all the individual modules together to form the entire ASIC design. The algorithm assumes that various modules and sub-

modules in the hierarchy are described using VHDL or Verilog as the hardware description language.

3. since most of the designs are I/O bound, the task of I/Os distribution is the toughest one. It is desired that the highly connected blocks should go into the same device so that those connections will be routed inside the device in order to reduce the I/O count. Once the design is synthesized and netlist is formed, all the information about the connections between different blocks can be achieved , based on which partitioning decision can be taken. But it is practically very difficult to read the entire netlist for big ASIC design as it is going to occupy a large memory space and the process is very much time-consuming.

4. so the algorithm eliminates the task of reading the netlist. First all the top level modules are individually taken into consideration and the size of each module is analyzed in terms of total number of I/Os, CLBs, flip-flops and latches.

If it exceeds the constraints of the given FPGA device, then it is essential to break down that particular top level module to the next hierarchical bottom level so that all the sub-modules in the sub-hierarchy are individually synthesized and all of them satisfy the given device constraints. If they do not, then it is necessary to traverse down further to the next sub-hierarchical level. For a particular module the optimum number of devices (N) is calculated based on the number of I/Os, CLBs, flip-flops and latches. The actual formula is given in the mathematical format of the algorithm that follows this section.

6. then all the sub-modules (M) are randomly distributed into N number of devices.

It is necessary that device constraints should be satisfied in terms of total number of I/Os , total number of CLBs as well as number of flip-flops and latches.

7. now the first device is selected and the constraints are checked. If they exceed the constraints specified by the manufacturer, then the module which is using largest number of resources (I/Os, CLBs, flip-flops and latches) is selected.
8. then so called the swapping procedure starts. This module will be swapped with a particular module in the successive devices such that the particular device constraints are satisfied.
9. like this, the swapping process continues for all the modules in the individual devices till the constraints for all the devices are satisfied.
10. now the next top level module is taken into consideration and steps 4-9 are repeated till all the top level modules in the entire ASIC design are covered.

Mathematically the algorithm can be described as follows:

1. Let M be the total number of top level modules present in a given ASIC design. Let N be the number of FPGA devices required to fit the entire ASIC design.

Let TIO be the total number of I/Os in the entire design and $TACTIO$ be the actual number of useful I/Os in a single FPGA device.

Similarly $TCLB$ is the total number of CLBs in the entire design and $TACTCLB$ is the actual number of useful CLBs in a single FPGA device. and $TFLIP$ is the total number of flip-flops in the design and $TACTFLIP$ is the actual number of flip-flops in a single FPGA device. Similarly $TLAT$ is the total number of latches in the

entire design and TACTLAT is the actual number of useful CLBs in a single FPGA device. Then the optimum value of N is calculated as:

$$N = \text{MAX} (TIO/ TACTIO, TCLB/ TACTCLB, TFLIP/TACTFLIP, \\ TLAT/TACTLAT)$$

2. Then distribute M modules randomly into N devices. Each device will incorporate maximum M/N modules.

3. Swapping process:

for (I=1, I<=N-1, I++)

{

for (J= I+1, J<=N, J++)

{

while (device constraints exceed the manufacturer's limits)

{

select the module using maximum resources (I/Os + CLBs + Flip-flops +
latches)

Find the module in the successive device which can satisfy the constraints.

Swap (module(I) , module(J))

select the next largest module in that device.

}

}

}

Repeat the above steps till all the top level modules in the design are covered.

CHAPTER 5

PRACTICAL EXAMPLE FOR IMPLEMENTING THE ALGORITHM

This example takes into consideration the ASIC design of INTEL 8085 microprocessor. Basically it focuses on the data-path section of this particular 8 bit microprocessor. The data-path section includes following units:

1. Arithmetic Logic Unit.
2. Shifter Unit.
3. Accumulator unit
4. Program counter
5. Flag byte register.
6. Memory address register.
7. Temporary register.
8. Instruction register unit.

All the above blocks are described in Verilog as a hardware description language.

The top level module (data-path) connects all these blocks together to form the data path section of the microprocessor. Once the coding part is done , first the top level module (data-path) is synthesized using FPGA Express source code. The target device is Xilinx-XC3000: 3120A. Since it is not possible to fit this entire module into a single device , the

partitioning needs be done. It is necessary to synthesize all the sub-modules and extract the required parameters about them.

```
// {module {data_path_8085.v}}

module data_path_8085 (AD, A, clk,ALE, mem_add_bus, reset_pc, load_ac,load_pc,
    zero_ac, load_ir,inc_pc,load_temp_reg,load_flag_byte, cm_carry, set_carry,
    out_flag_byte,alu_or,alu_xor, alu_and, alu_not, alu_a, alu_add, alu_temp,
    alu_sub,ir_lines,flag_on_dbus,rotate_left, rotate_right, rotate_left_c,
    rotate_right_c,ac_on_dbus,dbus_on_ac);

inout [7:0] AD;                //lower multiplexed address/data bus (AD7-AD)
output [15:8] A;                // higher address bus (A15-A8)

output [7:0] out_flag_byte;

output [7:0] ir_lines;

input [15:0] mem_add_bus;

input clk;

input ALE;

input flag_on_dbus;

input reset_pc;

input load_pc;

input load_ac;

input zero_ac;

input load_ir;

input load_temp_reg;
```

```
input inc_pc;

input load_flag_byte;

input cm_carry;

input set_carry;

input ac_on_dbus;

input dbus_on_ac;

input rotate_left;

input rotate_right;

input rotate_left_c;

input rotate_right_c;

//input [4:0] in_flag;

input alu_or;

input alu_xor;

input alu_and;

input alu_not;

input alu_a;

input alu_add;

input alu_temp;

input alu_sub;

// Define the intermediate signals.

wire [7:0] ac_out, obus, data_int_bus, ir_out, temp_out, alu_out;

wire [4:0] shu_flags, alu_flags, flag_out;
```

```

wire [15:0] pc_out;

// Instantiation of various components.

accumulator_unit r1 (obus ,clk, load_ac,zero_ac, ac_out);

temp_reg_unit r2 (obus, clk, load_temp_reg, temp_out);

instruction_reg_unit r3(obus, clk, load_ir, ir_out);

program_counter r4 (mem_add_bus, clk, load_pc, inc_pc, reset_pc, pc_out);

flag_byte_reg r5 (clk, load_flag_byte,set_carry,cm_carry,flag_out,shu_flags);

arithmetic_logic_unit r6 (ac_out, temp_out,alu_or, alu_xor, alu_and, alu_not,
                           alu_a, alu_add, alu_temp, alu_sub, flag_out,
                           alu_flags, alu_out);

shifter r7 (ac_out, rotate_left, rotate_right, rotate_left_c,
            rotate_right_c, alu_flags, obus, shu_flags);

address_buff r8 ( pc_out, ALE, clk, A, AD, data_int_bus);

assign data_int_bus = ac_on_dbus ? (ac_out) : (8'bzzzzzzzz);

assign obus = dbus_on_ac ? (data_int_bus) : (8'bzzzzzzzz);

assign ir_lines = ir_out;

assign data_int_bus = flag_on_dbus ? (out_flag_byte) : (8'bzzzzzzzz);

assign out_flag_byte = {flag_out[4],flag_out[3],1'b0,flag_out[2],
                        1'b0,flag_out[1],1'b0,flag_out[0]};

endmodule

```



```
//{module {temp_reg_unit}}  
  
module temp_reg_unit (i8, clk6, load, o8);  
  
input [7:0] i8;  
  
input clk6;  
  
input load;  
  
output [7:0] o8;  
  
reg [7:0] o8;  
  
always @(posedge clk6)  
- if (load)  
o8 <= i8;  
  
endmodule
```

```
//{module {instruction_reg_unit.v}}  
  
module instruction_reg_unit (i8, clk5, load, o8);  
  
input [7:0] i8;  
  
input clk5;  
  
input load;  
  
output [7:0] o8;  
  
reg [7:0] o8;  
  
always @(posedge clk5)  
  
if (load)
```

```

o8 <= i8;

endmodule

//{module { accumulator_unit.v } }

module accumulator_unit (i8, clk4, load, zero, o8);

input [7:0] i8;

input clk4;

input load;

input zero;

output [7:0] o8;

reg [7:0] o8;

always @(posedge clk4)

    if(load) begin

        if(zero)

            o8 <= 8'b00000000;

        else

            o8 <= i8;

        end

    endmodule

//{module { arithmetic_logic_unit.v } }

module arithmetic_logic_unit (a_side, temp_side, alu_or, alu_xor, alu_and, alu_not,

    alu_a, alu_add, alu_temp, alu_sub, in_flags, out_flags, alu_out);

```

```

input [7:0] a_side;

input [7:0] temp_side;

input alu_or;

input alu_xor;

input alu_and;

input alu_not;

input alu_a;

input alu_add;

input alu_temp;

input alu_sub;

input [4:0] in_flags;

output [4:0] out_flags;

output [7:0] alu_out;

reg [7:0] tr;

reg s2,z2,ac2,p2,c2;

reg [1:0] temp_ac;

always @(a_side or temp_side or alu_or or alu_xor or alu_and or alu_not or alu_a
        or alu_add or alu_temp or alu_sub or in_flags)
begin
    {s2,z2,ac2,p2,c2} = in_flags[4:0];

    case ({alu_or,alu_xor,alu_and,alu_not,alu_a,alu_add,alu_temp,alu_sub})
        8'b10000000: begin
            // a_or_temp
            tr = a_side | temp_side; c2 = in_flags[0];

```

```

        if (tr[7] == 1'b1) s2 = 1'b1;

        else s2 = 1'b0;

    if (tr == 8'b00000000) z2 = 1'b1;

    else z2=1'b0;

    end

8'b01000000: begin                                // a_exor_temp

    tr = a_side ^ temp_side; c2 = in_flags[0];

    if (tr[7] == 1'b1) s2 = 1'b1;

    else s2 = 1'b0;

    if (tr == 8'b00000000) z2 = 1'b1;

    else z2=1'b0;

    end

8'b000000100: begin                                //a_add_temp

    {c2,tr} = a_side + temp_side + in_flags[0];

    if (tr[7] == 1'b1) s2 = 1'b1;

    else s2 = 1'b0;

    temp_ac = a_side[3] + temp_side[3];

    if(temp_ac[1]==1'b1) ac2 = 1'b1;

    else ac2 = 1'b0;

    end

8'b000000001: begin                                // a_sub_temp

```

```

    {c2,tr} = a_side - temp_side - in_flags[0];

    if (tr[7] == 1'b1) s2 = 1'b1;

    else s2 = 1'b0;

    if (tr == 8'b00000000) z2 = 1'b1;

    else z2=1'b0;

    end

8'b00100000: begin                                // a_and_temp

    tr = a_side & temp_side; c2 = in_flags[0];

    if (tr[7] == 1'b1) s2 = 1'b1;

    else s2 = 1'b0;

    if (tr == 8'b00000000) z2 = 1'b1;

    else z2=1'b0;

    end

8'b00001000: begin                                // a_input

    tr = a_side; c2=in_flags[0]; s2= in_flags[4];

    end

8'b00000010: begin                                // temp_input

    tr = temp_side; c2=in_flags[0]; s2= in_flags[4];

    end

8'b00010000: begin                                // alu_not

    tr = ~a_side; c2=in_flags[0];

    if (tr[7] == 1'b1) s2 = 1'b1;

```

```

        else s2 = 1'b0;

        if (tr == 8'b00000000) z2 = 1'b1;

        else z2=1'b0;

        end

    default: begin

        tr = 8'b00000000; c2=1'b0; s2= 1'b0;

    end

endcase

    p2 = ~(^tr) ;

    end

assign alu_out = tr;

assign out_flags = {s2,z2,ac2,p2,c2};

    // out_flags = {s,z,1'bx,ac,1'bx,p,1'bx,c};

endmodule

//{module {address_buff_high}}

module address_buff (i16, ALE, clk3, add_out_high, add_low_data_inout,data_internal);

input [15:0] i16;

input ALE;

input clk3;

output [7:0] add_out_high;

inout [7:0] add_low_data_inout;

```

```

inout [7:0] data_internal;

reg [7:0] address_high;

reg [7:0] data;

reg [7:0] address_low;


always @(posedge clk3) begin

    if (ALE) begin

        address_high = i16[15:8];

        address_low = i16[7:0];

    end

else

    data = add_low_data_inout;

end

assign add_out_high = address_high;

assign data_internal = data;

assign add_low_data_inout = address_low;

endmodule


//{{module {program_counter}}

module program_counter (i16, clk2, load, increment, reset, o16);

input [15:0] i16;

input clk2;

```

```

input load;

input increment;

input reset;

output [15:0] o16;

reg [15:0] count;

always @(posedge clk2 or posedge reset)

    if (reset)

        count <= 16'b0000000000000000;

    else begin

        if (load)

            count <= i16;

        else

            if(increment)

                count <= count + 1;

        end

        assign o16 = count;

endmodule

//{module {flag_byte_reg}}

module flag_byte_reg (clk1, load, stc, cmc, out_flags, in_flags);

input [4:0] in_flags;

input clk1;

```



```

input load;

input stc;

input cmc;

output [4:0] out_flags;

reg s,z,ac,p,c;

always @(posedge clk1) begin
    if (load)
        {s,z,ac,p,c} <= in_flags[4:0];
    else begin
        if (cmc) c <= 1'b0;
        if (stc) c <= 1'b1;
    end
    end

    assign out_flags = {s,z,ac,p,c};
endmodule

//{module {shifter}}

module shifter (alu_side, rotate_left, rotate_right, rotate_left_c,
               rotate_right_c, in_flags, obus_side, out_flags);

input [7:0] alu_side;

input rotate_left;

```

```

input rotate_right;

input rotate_left_c;

input rotate_right_c;

input [4:0] in_flags;

output [7:0] obus_side;

output [4:0] out_flags;

reg [7:0] tr;

reg s1,z1,ac1,p1,c1;

always @ ( alu_side or rotate_left or rotate_right or rotate_left_c
           or rotate_right_c or in_flags)
begin
    {s1,z1,ac1,p1,c1} = in_flags[4:0];

    case ({rotate_left,rotate_right,rotate_left_c,rotate_right_c})

        4'b1000: begin                //rotate left

            tr[7:0] = {alu_side[6:0],alu_side[7]};

            c1 = alu_side[7];

        end

        4'b0100: begin                // rotate right

            tr[7:0] = {alu_side[0],alu_side[7:1]};

            c1 = alu_side[0];

```

```

end

4'b0010: begin                // rotate left through carry
    tr[7:0] = {alu_side[6:0], in_flags[0]};
    c1 = alu_side[7];
end

4'b0001: begin                // rotate right through carry
    tr[7:0] = {in_flags[0], alu_side[7:1]};
    c1 = alu_side[0];
end

default: begin
    tr = 8'b0; c1=1'b0;
end

endcase

end

assign obus_side = tr;

assign out_flags = {s1,z1,ac1,p1,c1};

endmodule

```

From the project Report (See Appendix II) generated after synthesizing the top level

Data-path module we collect the following information:-

Number of resources used by the top level data-path module.

1. Number of I/Os : 75

2. Number of flip-flops: 69
3. Number of latches: 0
4. Number of CLBs: 203

Specification for Xilinx device:- XC3000: 3142ATQ144

1. Useful Number of I/Os : 96
2. Total Number of useful CLBs : 144
3. Number of Flip-flops : 480
4. Number of gates : 2000 to 3000

Since number of resources required by the data-path module exceed the specifications of the device, the partitioning needs to be done. Also it is necessary to synthesize all the sub-modules to extract all the information needed to take the partitioning decision.

Table 5.6 Results generated after synthesizing all the individual sub-modules.

	Arithmetic Logic Unit	Shifter Unit	Accumulator unit	Program counter.
CLBs:	139	17	8	38
Flip-flops:	0	0	8	16

	Flag byte reg	Memory address reg	Temporary reg	Instruction reg
CLBs:	2	1	0	0
Flip-flops:	5	24	8	8

The partitioning algorithm works as follows:

1. Here for the sub-modules we don't have to consider number of I/Os since most of the I/O connections will be routed internal to the device reducing the total I/O count.
2. Total number of FPGA devices (Xilinx : XC3000: 3142ATQ144) = 2.
3. Let us assume that first 4 modules (Arithmetic logic unit, shifter, accumulator and program counter) go into the first device and the remaining modules go into the second device. Then the constraints of the first device will be exceeded so as per the swapping part of the algorithm, the arithmetic logic unit will be swapped with flag byte reg. module.
4. Thus the partitioning algorithm succeeds to satisfies the constraints of both the devices and the entire logic is evenly distributed.

CHAPTER 6

SUMMARY, CONCLUSION AND RECOMMENDATIONS

HES is a totally new concept. In my thesis I have theoretically as well as practically proved that HES technology is an innovative solution provided to the industries to increase the simulation speed by 10 to 100 times compared to the speed of software simulator. In the previous example we saw that the partitioning algorithm gives the best results. It is capable of handling the large ASIC designs and trying to fit them into multiple FPGA devices. FPGA Express functions are invoked using the DPM interface and based on the report generated by the tool, the partitioning decisions are taken to evenly distribute the entire logic into multiple FPGA devices.

This thesis mainly concentrates on externally using the FPGA Express Synthesis tool through DPM Interface. The “C” code is capable of doing the entire synthesizing procedure by executing a single command. Afterwards the partitioning software takes the partitioning decisions automatically. Finally based on these results, the individual FPGA devices need to be synthesized and the final top level netlist needs to be generated. This netlist can be fed to the re-programmable PCB (Printed Circuit Board) to connect all the devices together. This approach doesn’t talk about the connections between modules on the same device (routing internally or externally) since it is beyond the scope of this thesis.

Also partitioning of ASIC design is going to affect the timing constraints specified by the given ASIC design. It is the job of synthesis, placement and routing tool to find out the critical paths and correct the timings. Again this part is beyond the scope of my thesis. The suggestions that I would recommend is put the critical paths inside the FPGA device and put the less critical paths on the PCB.

Execution of the software code

Following are the steps needed for the execution of the software code written in C language. Basically the code is capable of synthesizing and partitioning the given HDL ASIC Design into multiple FPGA devices. First step is synthesize the individual top level modules. It can be done by executing the following command on the MS-DOS command prompt.

Synthesis -f filename -d design -p project -t target -r c:\synopsys\fpaga_express\lib

Filename specifies given VHDL (.vhd file) or Verilog (.v file).

Design stands for the name of the design

Project stands for the name of the project.

Target stands for the final FPGA target device. (e.g. XC3000, SPARTAN/XL/2 XC4000 {E|EX|L|XL|XLA|XV}, XC5200, XC500/XL/XV, VIRTEX/E/2APEX20K, APEX20KE, FLEX6000, FLEX8000, FEX10K, FLEX10KA, ACEX1K, MAX7000, MAX9000, ORCA2{A|TB}, ORCA3{C|FPSC|L}, ACT2-1200XL, A3200DX, A42MX, A54SX, A54SXA, etc.)

-r specifies the root directory file for the FPGA EXPRESS library files.

Before executing this command it is necessary to produce synthesis.exe file using nmake command. Note that the synthesis command can be executed only in Windows 2000 environment along-with FPGA EXPRESS 3.4 version and Visual Studio 6.0 version available in the system.

e.g. command to synthesize tutor.vhd module.

```
synthesis -f tutor.vhd -d tutor -p proj_vhdl -t XC4000E -r c:\synopsys\fpga_express\lib
```

In order to partition different modules it is necessary to supply the list of all files along with the top level module in the command. (e.g -f tutor.vhd, and.vdh,or.vhd)

File.cpp is capable of reading the project report generated by the synthesis software. This report is required for taking the partitioning decisions. The next step is to partition the modules using partition software.

Before that it is necessary to give following information about modules in .txt file which can be given as an input to the software.

Number of modules.

Names of different modules,

Specification of the target device. (i.e. number of useful I/Os, flip-flops, latches, CLBs)

e.g. for SUN'S PICO_JAVA processor floating point unit the data.txt file looks like

9

module0:codeseq

module1:codeckt

module2:dp

module3:exponent

,

module4:exrc

module5:exp-dp

module6:f-fpu

module7:mantiss

module8:multi-array

144 96 480 680

where there are 9 overall individual modules named module0 to module8.

Specification for Xilinx device:- XC3000: 3142ATQ144

1. Total Number of useful CLBs : 144
2. Useful Number of I/Os : 96
3. Number of Flip-flops : 480
4. Number of latches: 680

Command to be executed:

partition < data.txt > result.txt

It will take input from data.txt and produce output file as result.txt

module0:codeseq

module1:codeckt

module2:dp

module3:exponent

module4:exrc

module5:exp-dp

module6:f-fpu

module7:mantiss
module8:multi-array
number of modules 9
number of devices 5
chip 0 contains module 0
total clbs in chip 0 is16
total I/Os in chip 0 is91
total flip-flops in chip 0 is300
total latches in chip 0 is0
chip 1 contains module 1
total clbs in chip 1 is133
total I/Os in chip 1 is30
total flip-flops in chip 1 is150
total latches in chip 1 is0
chip 2 contains module 2
total clbs in chip 2 is60
total I/Os in chip 2 is35
total flip-flops in chip 2 is200
total latches in chip 2 is0
chip 3 contains module 3
chip 3 contains module 4
total clbs in chip 3 is86

total I/Os in chip 3 is 90
total flip-flops in chip 3 is 368
total latches in chip 3 is 0
chip 4 contains module 5
total clbs in chip 4 is 66
total I/Os in chip 4 is 88
total flip-flops in chip 4 is 34
total latches in chip 4 is 0
total clbs in chip 0 is 16
total I/Os in chip 0 is 91
total flip-flops in chip 0 is 300
total latches in chip 0 is 0
total clbs in chip 1 is 133
total I/Os in chip 1 is 30
total flip-flops in chip 1 is 150
total latches in chip 1 is 0
chip 2 contains module 6
total clbs in chip 2 is 138
total I/Os in chip 2 is 47
total flip-flops in chip 2 is 223
total latches in chip 2 is 0
total clbs in chip 3 is 86

total I/Os in chip 3 is 90

total flip-flops in chip 3 is 368

total latches in chip 3 is 0

total clbs in chip 4 is 66

total I/Os in chip 4 is 88

total flip-flops in chip 4 is 34

total latches in chip 4 is 0

now number of devices is : 6

chip 5 contains module 7

chip 5 contains module 8

total clbs in chip 5 is 28

total I/Os in chip 5 is 57

total flip-flops in chip 5 is 168

total latches in chip 5 is 0

partitioning algorithm successful

APPENDIX I

INTRODUCTION TO VERILOG

Verilog is a hardware description language. It was invented as a simulation language.

Verilog allows engineers to write models that represent a device.(examples: memory and microprocessor.)

Similarity to other languages

Syntax command '**if**' is the same as in **C++**

Blocks **begin-end** are identical to those in **Pascal**.

Block **always** in Verilog is very similar to **process** in VHDL.

Module and **endmodule** similar to **main()** block in C++.

Need of Verilog

Simulation- We can simulate models before and after synthesis / place and routing.

Most popular language for ASICs and very common for FPGAs.

“Verilog is easier to use than other hardware description languages.”

Language basic rules

Keywords are reserved, lower case words

Verilog is case sensitive!!

Identifiers are name of objects:

Must start with letter or underscore

May contain letters, digits , underscores and dollar (\$) sign.

Are case sensitive!!

Comments : 2 types

single line comment

Multi Line comment.

Keywords :

module, endmodule, initial ,always.

Identifiers

My_decoder , _Mydecoder, My\$Decoder

comments

single line comment.

// single line comment.

Multi Line comment.

/* Multi line comment enclosed between this */

Sample Model In Verilog

AND gate model:

Model code is between **module** and **endmodule** keywords.

Keywords **input** and **output** are used for a,b and c variable declarations.

assign c = a &b ;

is bit-wise assignment of a and b variables to the c variable.

module and_gate(a, b, c);

```

input a ,b;

output c;

    assign c = a & b;

endmodule;

```

In **Verilog** there are four basic values for hardware modelling:

Value	Logic state
0	Logic zero / False condition
1	Logic one / True condition
X	Unknown Value
Z	High Impedance

// value identifiers are case insensitive.

Data types

Verilog contains 2 main groups of data types:

1. Nets:

.wire

.wand, wor, tri, triand, trior

tri0, tri1, trireg, supply0, supply1

2. Registers:

reg

integer

real

time

realtime

Nets are used to connect structural entities.

Numbers

There are 2 types of numbers:

Sized number:

`<size> '<base> <value>`

unsized number:

`'<base> <value>`

size is width of number in bits and it is always decimal.

Default base is decimal if base is not specified.

Sized numbers:

`8'b 10110010 // 8 bit binary value`

`16'h 97fx // 16 bit hex value`

(last 4 bits unknown "x")

Unsize numbers

`'d 34 //decimal value 34`

`'o 274 // octal value 1010011`

“Wire” data type

The “wire” represents physical connection between structural entities such as gates.

It doesn't store any value, its value is determined by the values of its drivers.

module test;

wire result;


```
/* declare one bit wire data type
```

```
named result */
```

```
    assign result = variable;
```

```
endmodule
```

“reg” data type

A verilog “reg” type is a variable that maintains its value until updated

```
reg reset;
```

```
/* declaration of one bit reg data type (named reset) */
```

```
initial begin
```

```
    reset = 1'b1;
```

```
    # 100 reset = 1'b 0;
```

```
end
```

```
/* #100 - it is a delay. = 100 units of time. */
```

“Vectors”

In Verilog, any data type may be represented as a vector. The vectors provide means of modeling buses.

```
<reg, wire> [Msb#: Lsb#] <identifier>
```

Msb is always on left in verilog.

Part select means a continuous group of bits in vector.

Bit select means one bit in a vector.

```
// vectors:
```

```
reg [0:7] control;
```

```

reg [15:0] data;

wire[7:0] data_1;

// Part select :

data[15:12] = data_1[3:0];

/* data_1's least significant bits are assigned to the data's most significant bit */

// Bit select

data [12] = data_1 [2];

```

Module

Module is an equivalent to the hardware model. It contains all information about this model or part of it.

Interface: connection with other modules.

Body : Description of model.

```
`include "time_scale.v"
```

```
/* include directive allows to include entire contents of a Verilog source file in another
Verilog file during compilation.*/
```

```
module M(Q, IN);
```

```
output [3:0] Q;
```

```
input [3:0] IN;
```

```
reg [3:0] Q;
```

```
initial begin
```

```
#10; Q[0] = IN[3];
```

```
#10; Q[1] = IN[2];
```

end;

endmodule

Logic gates

In Verilog there are pre-defined gate level primitives: and, or, xor, nand, nor, xnor.

These gates have multiple inputs and only one output.

Keyword identifiers are used to reference gate primitives along with the output and input signals.

1. **Output** is always first in the port list even if gate is multi-outputs.
2. Note: Instance names are not required but they should be used , especially in case of more than one instantiations of the same module or primitive.

and and3 (out, In1, In2, In3);

Assignment Statements

Continuous assignment:

are the basic constructs for data flow modeling

are used for updating nets. (The left-hand side of an assignment is the net. The right-hand side can be any expression.)

In the following example output out changes value whenever any of the input (a or b) change.

module data_flow (out, a, b);

output out;

input a ,b;

assign out = a & b;

endmodule

Data flow Coding

module mux (a,b,sel, out);

output out;

input a, b, sel;

wire N1, N2;

assign N1 = (a & sel);

assign N2 = (b & ~sel);

assign out = (N1 | N2);

endmodule

// assign out = (a & sel) | (b & ~sel);

// assign out = sel ? a : b ;

// Multiplexer 2:1

Initial vs. always

The **initial** statement is executed only once during the simulation.

The **always** statement is executed continuously during the simulation.

The **always** statement should have at least one timing control statement (delay or event control.)

The **always** statement can be used to detect and react on specific event (e.g. positive edge on clock).

initial a = 1'b0;

```
always #5 clk = !clk;
```

```
initial begin
```

```
  a = 1'b0;
```

```
  b = 1'b1;
```

```
end
```

```
always @ (posedge clk) q = d;
```

The initial and always statement are executed in parallel.

Blocking assignments

Shall be executed before the execution of the statement that follow it in sequential block.

Shall not prevent the execution of statement that follow it in a parallel block.

Non-blocking assignments:

Allows assignments scheduling without blocking the procedural flow.

Can be used whenever several register assignments can be made without regard to other or dependence upon each other.

Blocking assignments:

```
register [4] = 1'b1;
```

```
register [4:0] = 5'b10110;
```

Non-blocking assignments:

```
always @ (a or b)
```

```
begin
```

```
  a <= b;
```

```
c <= d;
```

```
end
```

Blocking vs. non-blocking assignments

Blocking assignments:-

```
1. begin
```

```
    a = # 20 c;
```

```
    b = # 30 d;
```

```
end
```

```
// a = c at time 20;
```

```
// b=d at time 50;
```

```
2.    // a = 50;
```

```
begin
```

```
    a = #20 20;
```

```
    b = #10 a+100;
```

```
end
```

```
// b = 120;
```

Non- Blocking assignments:

```
1. begin
```

```
    a <= # 20 c;
```

```
    b <= # 30 d;
```

```
end
```

```
// a = c at time 20;
```

```
// b=d at time 30;

2. // a = 50;

begin

    a <= #20 20;

    b <= #10 a+100;

end

// b = 150;
```

Compiler directives:

`timescale - specify the time unit and time precision of the modules that follow it.

`include - insert a source file in another file during compilation.

```
`timescale 1 ns / 1 ps
```

```
// time unit = 1 ns
```

```
// time precision = 1 ps.
```

```
`include "time_scale.v"
```

```
// insert file - time_scale.v
```

Asynchronous counter

Design an asynchronous decade counter.

Input : CLK , Enable, Reset

Outputs : Q_OUT[3:0] , FULL.

Reset is asynchronous.

When Q_OUT changes from 9 to 0 - FULL is set to 1 for one CLK pulse.

Verilog Code:

```

// counter.v

module counter (CLK, Enable , Reset , Q_OUT, FULL) ;

input CLK ;

input Enable;

input Reset;

inout FULL ;

reg [3:0] Q;

reg FULL_internal ;

// verilog code continued

always @ (posedge CLK or posedge Reset )

begin

    if ( Reset != 1'b1 )

        begin

            if ( Enable == 1'b1 )

                begin

                    Q = Q + 1'b1;

                    if ( Q == 10)

                        begin

                            Q = 1'b 0;

                            FULL_internal = 1'b 1;

                        end

                    else FULL_internal = 1'b 0;

                end

            end

        end

    end

```



```

                                end
                        end
                else
                        begin
                                Q = 1'b 0;
                                FULL_internal = 1'b0;
                        end
                end

        end

assign Q_OUT = Q;
assign FULL = FULL_internal;
endmodule

```

Assign and deassign

Used only for registers

If a positive edge occurs on the clock (clk) then the counter value (q) will be incremented by one.

If the reset (r) is high , override regular , assignment to q with the new values , using procedural continuous assignments.

If the reset is high then the counter value will be set to 0. When the next positive edge occurs on the clock then the counter value will not be incremented.

If the reset goes low, remove the overriding values by deassigning the register q.

```

module counter (q, clk, r)

output [3:0] q ;

```

```

reg [3:0] q ;

input  clk , r ;

always @ (posedge clk)

q = q + 1 ;

always @ ( r )

if ( r )

    assign q = 0;

else deassign q ;

endmodule

```

Force and release

Used with nets and registers.

If en is low then y will be driven by a continuous assignment. (keyword assign).

If en is high then y will be driven by a procedural continuous assignment (keyword force.)

But when en goes from high to low then the procedural statement will be ended and y will be again driven by the continuous assignment.

```

module circuit (y, a , b, c ,en);

    output y;

    input a, b ,c ,en ;

    assign y = ( a | b) && c;

    always @ (en)

        if (en )

            force y = (a && b) || c ;

```

```

else
    release y;
endmodule

```

Intra-assignments Vs. delays

Delay with assignment :

wait for a specific time

evaluate an expression

assign a new value

intra - assignment:

evaluate an expression

wait for a specific time

assign a new value.

// delay with assignment:

```
# 5 a = b;
```

//equivalent to :

```
# 5;
```

```
a = b;
```

//Intra-assignment

```
a = # 5 b;
```

//equivalent to

```
temp = b;
```

```
# 5;
```

a = temp;

Delay control

Delays are used to specify timing dependencies between the assignment and value change.

A delay control begins with the # value character followed by the delay value.

A delay definition can contain up-to 3 values :

A change to 0 value (positive edge)

A change to 0 value (negative edge.)

A change to high impedance value

If no delay is specified or value changes to the unknown (x) value then the smallest of the specified delays will be used.

2 /* one delay specified. Same value for all changes. */

(5, 2)

Two Delays specified : -

For changes to 1 the first value (5) will be used

Value 2 will be used for changes to 0.

For changes to x or z smallest (2) will be used.

(1, 3, 5)

for changes to x the smallest value (1) will be used.

Min : Typ : Max Delays

All delays can be specified using three values:

Min : minimum delay

Max : maximum delay

Typ : typical delay

(1 : 2 : 3)

One delay specified :

For minimum delay 1 will be used .

typical : 2 ; maximum : 3

(4:5:6 , 1:2:3)

(0:1:2 , 3:4:5 , 6:7:8)

Memories

- Memory - array of vectors

Declaration of memory :

type identifier [l_index : r_index];

range declarations can be used only for memories declared as reg data types

access to memory word :

memory_name [index]

```
module memory ( d_out , d_in , addr , clk, rw);
```

```
output [7:0 ] d_out;
```

```
reg [7:0] d_out;
```

```
input [7:0] d_in;
```

```
input [4:0] addr;
```

```
input clk , rw;
```

```
reg [7:0] mem [31:0];
```

```
always @ (posedge clk )  
  
    if (rw)  
  
        d_out = mem [addr];  // read  
  
    else  
  
        mem [addr] = d_in;  
  
endmodule
```

APPENDIX II

FPGA EXPRESS PROJECT REPORT

Device data_path_8085-Optimized

Summary Information:

Type: Optimized implementation

Source: data_path_8085, up to date

Status: 0 errors, 107 warnings, 0 messages

Export: not exported since last optimization

Target Information:

Vendor: Xilinx

Family: XC3000

Device: 3142ATQ144

Speed: -09

Device Parameters:

Optimize for: Speed

Optimization effort: High

Frequency: 10 MHz

Is module: No

Keep io pads: No

Number of flip-flops: 69

Number of latches: 0

Device Design Hierarchy:

data_path_8085: defined in C:\My_Designs\alu\src\data_path_8085.v

accumulator_unit - r1: defined in C:\My_Designs\alu\src\accumulator_unit.v

temp_reg_unit - r2: defined in C:\My_Designs\alu\src\temp_reg_unit.v

instruction_reg_unit - r3: defined in C:\My_Designs\alu\src\instruction_reg_unit.v

program_counter - r4: defined in C:\My_Designs\alu\src\program_counter.v

flag_byte_reg - r5: defined in C:\My_Designs\alu\src\flag_byte_reg.v

shifter - r7: defined in C:\My_Designs\alu\src\shifter.v

arithmetic_logic_unit - r6: defined in C:\My_Designs\alu\src\arithmetic_logic_unit.v

address_buff - r8: defined in C:\My_Designs\alu\src\address_buff.v

Table A.1 Primitive reference count:

ACLK	1
DFF	69
EQN	203
IBUF	42
OBUF	32
TBUF	24

Table A.2 Clocks:

			Required	Estimated	
Period	Rise	Fall	Freq	Freq	Signal
(ns)	(ns)	(ns)	(MHz)	(MHz)	
100	0	50	10.00	-1.00	default
-1	-1	-1	-1000.00	13.72	clk_ACLKed

Table A.3 Timing Groups:

Name	Description
(I)	Input ports
(O)	Output ports
(RC,clk_ACLKed)	Clocked by rising edge of clk_ACLKed

Table A.4 Input Port Timing:

		Required	Estimated	
Port	Delay	Slack		
Name	(ns)	(ns)	To-Group	
AD<7>	n/a	n/a	(RC,clk_ACLKed)	
AD<7>	n/a	n/a	(RC,clk_ACLKed)	
AD<7>	100.00	87.84	(RC,clk_ACLKed)	
AD<7>	100.00	87.84	(RC,clk_ACLKed)	
AD<6>	n/a	n/a	(RC,clk_ACLKed)	

AD<6>	n/a	n/a	(RC,clk_ACLKed)
AD<6>	100.00	87.84	(RC,clk_ACLKed)
AD<6>	100.00	87.84	(RC,clk_ACLKed)
AD<5>	n/a	n/a	(RC,clk_ACLKed)
AD<5>	n/a	n/a	(RC,clk_ACLKed)
AD<5>	100.00	87.84	(RC,clk_ACLKed)
AD<5>	100.00	87.84	(RC,clk_ACLKed)
AD<4>	n/a	n/a	(RC,clk_ACLKed)
AD<4>	n/a	n/a	(RC,clk_ACLKed)
AD<4>	100.00	87.84	(RC,clk_ACLKed)
AD<4>	100.00	87.84	(RC,clk_ACLKed)
AD<3>	n/a	n/a	(RC,clk_ACLKed)
AD<3>	n/a	n/a	(RCclk_ACLKed)
AD<3>	100.00	87.84	(RCclk_ACLKed)
AD<3>	100.00	87.84	(RCclk_ACLKed)
AD<2>	n/a	n/a	(RCclk_ACLKed)
AD<2>	n/a	n/a	(RCclk_ACLKed)
AD<2>	100.00	87.84	(RCclk_ACLKed)
AD<2>	100.00	87.84	(RC,clk_ACLKed)
AD<1>	n/a	n/a	(RC,clk_ACLKed)
AD<1>	n/a	n/a	(RC,clk_ACLKed)
AD<1>	100.00	87.84	(RC,clk_ACLKed)

AD<1>	100.00	87.84	(RC,clk_ACLKed)
AD<0>	n/a	n/a	(RC,clk_ACLKed)
AD<0>	n/a	n/a	(RC,clk_ACLKed)
AD<0>	100.00	87.84	(RC,clk_ACLKed)
AD<0>	100.00	87.84	(RC,clk_ACLKed)
clk	98.66	98.66	(RC,clk_ACLKed)
ALE	n/a	n/a	(RC,clk_ACLKed)
mem_add_bus<15>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<14>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<13>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<12>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<11>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<10>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<9>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<8>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<7>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<6>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<5>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<4>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<3>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<2>	93.10	93.10	(RC,clk_ACLKed)
mem_add_bus<1>	93.10	93.10	(RC,clk_ACLKed)

mem_add_bus<0>	93.10	93.10	(RC,clk_ACLKed)
reset_pc	n/a	n/a	(RC,clk_ACLKed)
load_ac	n/a	n/a	(RC,clk_ACLKed)
load_pc	89.40	89.40	(RC,clk_ACLKed)
zero_ac	90.22	90.22	(RC,clk_ACLKed)
load_ir	n/a	n/a	(RC,clk_ACLKed)
inc_pc	n/a	n/a	(RC,clk_ACLKed)
load_temp_reg	n/a	n/a	(RC,clk_ACLKed)
load_flag_byte	92.69	92.69	(RC,clk_ACLKed)
cm_carry	n/a	n/a	(RC,clk_ACLKed)
set_carry	92.69	92.69	(RC,clk_ACLKed)
alu_or	58.24	58.24	(RC,clk_ACLKed)
alu_xor	58.24	58.24	(RC,clk_ACLKed)
alu_and	58.24	58.24	(RC,clk_ACLKed)
alu_not	58.24	58.24	(RC,clk_ACLKed)
alu_a	57.83	57.83	(RC,clk_ACLKed)
alu_add	27.90	27.90	(RC,clk_ACLKed)
alu_temp	57.83	57.83	(RC,clk_ACLKed)
alu_sub	61.93	61.93	(RC,clk_ACLKed)
flag_on_dbus	83.13	83.13	(RC,clk_ACLKed)
rotate_left	79.16	79.16	(RC,clk_ACLKed)
rotate_right	79.16	79.16	(RC,clk_ACLKed)

rotate_left_c	79.16	79.16	(RC,clk_ACLKed)
rotate_right_c	79.16	79.16	(RC,clk_ACLKed)
ac_on_dbus	83.13	83.13	(RC,clk_ACLKed)
dbus_on_ac	85.13	85.13	(RC,clk_ACLKed)

Table A.5 Output Port Timing:

Required Estimated			
Port	Delay	Slack	
Name	(ns)	(ns)	From-Group
AD<7>	n/a	n/a	(RC,clk_ACLKed)
AD<7>	n/a	n/a	(RC,clk_ACLKed)
AD<7>	100.00	87.84	(RC,clk_ACLKed)
AD<7>	100.00	87.84	(RC,clk_ACLKed)
AD<6>	n/a	n/a	(RC,clk_ACLKed)
AD<6>	n/a	n/a	(RC,clk_ACLKed)
AD<6>	100.00	87.84	(RC,clk_ACLKed)
AD<6>	100.00	87.84	(RC,clk_ACLKed)
AD<5>	n/a	n/a	(RC,clk_ACLKed)
AD<5>	n/a	n/a	(RC,clk_ACLKed)
AD<5>	100.00	87.84	(RC,clk_ACLKed)
AD<5>	100.00	87.84	(RC,clk_ACLKed)
AD<4>	n/a	n/a	(RC,clk_ACLKed)

AD<4>	n/a	n/a	(RC,clk_ACLKed)
AD<4>	100.00	87.84	(RC,clk_ACLKed)
AD<4>	100.00	87.84	(RC,clk_ACLKed)
AD<3>	n/a	n/a	(RC,clk_ACLKed)
AD<3>	n/a	n/a	(RC,clk_ACLKed)
AD<3>	100.00	87.84	(RC,clk_ACLKed)
AD<3>	100.00	87.84	(RC,clk_ACLKed)
AD<2>	n/a	n/a	(RC,clk_ACLKed)
AD<2>	n/a	n/a	(RC,clk_ACLKed)
AD<2>	100.00	87.84	(RC,clk_ACLKed)
AD<2>	100.00	87.84	(RC,clk_ACLKed)
AD<1>	n/a	n/a	(RC,clk_ACLKed)
AD<1>	n/a	n/a	(RC,clk_ACLKed)
AD<1>	100.00	87.84	(RC,clk_ACLKed)
AD<1>	100.00	87.84	(RC,clk_ACLKed)
AD<0>	n/a	n/a	(RC,clk_ACLKed)
AD<0>	n/a	n/a	(RC,clk_ACLKed)
AD<0>	100.00	87.84	(RC,clk_ACLKed)
AD<0>	100.00	87.84	(RC,clk_ACLKed)
A<15>	100.00	88.25	(RC,clk_ACLKed)
A<14>	100.00	88.25	(RC,clk_ACLKed)
A<13>	100.00	88.25	(RC,clk_ACLKed)

A<12>	100.00	88.25	(RC,clk_ACLKed)
A<11>	100.00	88.25	(RC,clk_ACLKed)
A<10>	100.00	88.25	(RC,clk_ACLKed)
A<9>	00.00	88.25	(RC,clk_ACLKed)
A<8>	100.00	88.25	(RC,clk_ACLKed)
out_flag_byte<7>	100.00	87.43	(RC,clk_ACLKed)
out_flag_byte<6>	100.00	87.43	(RC,clk_ACLKed)
out_flag_byte<5>	n/a	n/a	(RC,clk_ACLKed)
out_flag_byte<4>	100.00	87.43	(RC,clk_ACLKed)
out_flag_byte<3>	n/a	n/a	(RC,clk_ACLKed)
out_flag_byte<2>	100.00	87.84	(RC,clk_ACLKed)
out_flag_byte<1>	n/a	n/a	(RC,clk_ACLKed)
out_flag_byte<0>	100.00	86.19	(RC,clk_ACLKed)
ir_lines<7>	100.00	88.25	(RC,clk_ACLKed)
ir_lines<6>	100.00	88.25	(RC,clk_ACLKed)
ir_lines<5>	100.00	88.25	(RC,clk_ACLKed)
ir_lines<4>	100.00	88.25	(RC,clk_ACLKed)
ir_lines<3>	100.00	88.25	(RC,clk_ACLKed)
ir_lines<2>	100.00	88.25	(RC,clk_ACLKed)
ir_lines<1>	100.00	88.25	(RC,clk_ACLKed)
ir_lines<0>	100.00	88.25	(RC,clk_ACLKed)

BIBLIOGRAPHY

SYNOPSYS FPGA Express Online Help: www.synopsys.com

Synplicity Online Help: www.synplicity.com

HES Online Help: www.aldec.com, www.speedgateinc.com

Verilog Digital System Design : by Zainalabedin Navabi

Microprocessor and Micro-controller Fundamentals: 8085 and 8051

By William Kleitz

VITA
Graduate College
University of Nevada, Las Vegas
Nilesh Dhavlikar

Local Address:
4236 Grove Circle, apt 1
Las Vegas, NV- 89119

Degree:
Bachelor of science in Electrical Engineering, 1998
University of Pune, India

Thesis Title: Partitioning of large ASIC designs into multiple FPGA devices for
Prototyping and Verification.

Thesis Examination Committee:
Chairperson- Dr. Henry Selvaraj, Ph.D.
Committee member- Dr. Rama Venkat, Ph.D.
Committee member- Dr. Shahram Latifi, Ph.D.
Graduate College Faculty Representative- Dr. Ajoy Datta, Ph.D.