A SYNTHESIZABLE CORE GENERATOR DESIGN IN VHDL FOR THE MOTOROLA 68XXX FAMILY OF MICROPROCESSORS

by

Alper Bürümcek

B.S. in Electric and Electronics Engineering, Boğaziçi University, 2002

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electric and Electronics Engineering
Boğaziçi University

2005

ACKNOWLEDGEMENTS

I would like to thank Prof. Ömer Cerid, my project supervisor for all the help and support given throughout the project. I would also like to thank my friend Uğur Bozkaya for his help in the Visual Basic design of the core generator. Finally I would like to thank my family for their patience and support.

ABSTRACT

A SYNTHESIZABLE CORE GENERATOR DESIGN IN VHDL FOR THE MOTOROLA 68XXX FAMILY OF MICROPROCESSORS

The goal of this project is to implement a core generetor to create a Motorola 68000 op-code compatible microprocessor capable of being synthesized using FPGA. The CPU is specified fully in VHDL and is designed to emulate the functionality of the MC68000 in terms of instruction set decoding, operand addressing and bus operation. The ultimate target device is fully enhanced, self testing, memory and cache optimized, improved version of the Motorola 68000 microprocessor. The core generator is to be fully automatic without any need of editing the VHDL code created. The core generator is to be functional, effective and modular for using with any microprocessor design. As a result of this, a large emphasis has been placed on how to create the microprocessor structures especially the instruction sets, arithmetic logic units and memory structure using a core generator.

ÖZET

MOTOROLA 68XXX MİKROİŞLEMCİ AİLESİ İÇİN SENTEZLENEBİLİR VHDL CPU YARATICI PROGRAM TASARIMI

Bu projenin amaçı FPGA kullanılarak sentezlenecek Motorola 68000 işlem kodu uyumlu bir mikroişlemci tasarlamaktır. Mikroişlemci tamamen VHDL kullanılarak dizayn edilecek ve MC68000 mikroişlemcisinin veri yolu operasyonu, operand adreslemesi ve işlem kümesi de dahil olmak üzere tüm fonksiyonlarına aynen sahiptır. Ulaşılan sonuç, gelişkin, kendi kendini test edebilen, hafıza ve önbellek sistemi optimize edilmiş bir Motorola 68000 mikroişlemcisidir. Bu sonuca ulaşmak ve projeyi daha ileri bir aşamaya getirebilmek için hedef mikroişlemci de dahil herhangi bir mikroişlemci kolayca VHDL ortamında oluşturabilecek bir 'Core Generator' da proje dahilinde tasarlanmıştır. Bu amaçla, gelecekteki tasarım yapılarını da etkileyecek olan bu teknolojiyle yapılacak projede, özellikle hafıza yapısı, aritmetik-mantık birimleri ve işlem kümelerinin tasarımı ve geliştirilmesine önem verilmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
ASD	Arithmetic Shift
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
BERR	Bus Error
BIST	Built In Self Test
CAD	Computer Aided Drafting
CCR	Conditional Code Register
CISC	Complex Instruction Set Computer
CM	Control Memory
CMOS	Complimentary Metal Oxide Semiconductor
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
DIVS	Divide Signed
DSP	Digital Signal Processor
DTACK	Data Acknowledge
EA	Effective Address
EAR	Effective Address Register
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HP	Hewlett Packard
IC	Integrated Circuit
IP	Intellectual Property
IR	Instruction Register
LSB	Least Significant Bit

LSI	Large Scale Integration
LSL	Logical Shift Left
LSR	Logical Shift Right
MB	Megabyte
MSB	Most Significant Bit
MSI	Medium Scale Integration
MULU	Multiply Unsigned
MUX	Multiplexer
NMOS	Negative Polarity Metal Oxide Semiconductor
PC	Program Counter
PLD	Programmable Logic Devices
RAM	Random Access Memory
RASSP	Rapid-Prototyping Application Specific Signal Processors
RISC	Reduced Instruction Set Computer
ROD	Rotate
ROL	Rotate Left
ROM	Read Only Memory
ROR	Rotate Right
ROXL	Rotate Extended Left
ROXR	Rotate Extended Right
RTL	Register Transfer Level
SoC	System on Chip
SSP	System Stack Pointer
SR	Status Register
STF2	Standard Tool Flow 2
TAS	Test and Set
USP	User Stack Pointer
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1. INTRODUCTION

This chapter serves as an introduction to the synthesizable Motorola 68000 CPU project in VHDL. The background, motivations and the intentions behind such a project are outlined. In basic words this chapter explores the design approach chosen and the reasons for choosing such an approach.

1.1. Motivation

The main purpose is to build a VHDL 68000 microprocessor using the HDL tools for observing their capabilities. Then to add superior functionality like self testing, memory and cache optimization and self-learning abilities to the cpu. Another trivial purpose is to produce the project code using a specifically designed core generator which can later be used for other types of microprocessor designs. The core generator produced will increase the project readability and will automatically create the documentation of the project. A good working core generator will provide the other designers and users with an effective tool for building their microprocessor cores in VHDL for various purposes. After the release of OpenRISC 1000 32-bit RISC CPU core by OpenCores Web Site, analysts say the offering, and others like it, may eventually alter the semiconductor IP landscape as radically as Linux has transformed the operating systems market. A core generator designed solely for microprocessor design will be the key feature for the open core movement development.

Once the processor had a bus interface capable of being used on board, it opens possibilities for expanding FPGA based microprocessor boards to allow users to configure various parts of the design to their particular needs and use different architectures within the same physical hardware paradigm. The most realistic motivation behind this work is to rebuild a core preserving all the functionality of the original processor

even to replace it when necessary. For example the Motorola 6800 processor, is today a major concern for French nuclear plants. Indeed, this component, selected 25 years ago to be part of high safety equipments, is no more available in the market, and the company is quickly running out of stocks. One of the solutions studied is to replace it by implementing the 6800 processor by a suitable emulation implemented by means of a programmable device . In the same manner the 68000 Microprocessor would help preserve the use of the Motorola processor as a teaching tool since it is currently used heavily in Computer Science and Electronics Engineering due to its simplicity when compared to other processors and would help to reproduce the core where necessary especially for military use since Motorola no longer produces these microprocessors.

Also, a motivation for me is to improve my abilities in VHDL programming and FPGA design, since they will replace the traditional chip design methodologies in a short time and the only way I could add functions designed by myself using different methods of research to a known processor and verify them, is designing the microprocessor using VHDL. Also a VHDL design programmed in an efficient and modular manner allows a user to examine a core component, completely ignoring the surrounding implementation.

1.2. Current Project

The objective of this project is to explore the capabilities of Hardware Description Languages (HDL) for designing complex cores and to reproduce the absolute microprocessors with improved Arithmetic Logic Unit (ALU), memory and instruction decoding structures using recent hardware advances. A secondary objective is to produce a special core generator for other microprocessor cores to be easily reproduced with minor changes in the base code. The continuing development of technology in medicine has enabled doctors and scientists to develop new kinds of image scanning machinery, computer software and methods of application. While the medical doctor is

the authority in the diagnosis of a patient, it is the engineer's expertise to develop new technologies and adapt them to diverse application fields, such as medical imaging.

Motorola's 68000 microprocessor with Complex Instruction Set Computer (CISC) strikes a nice balance between a powerful instruction set, advanced memory addressing capability and simplicity of interface. Largely due to the absence of legacy instruction support, floating point arithmetic, or multi-media extensions commonly found in today's processors, this balance is very desirable to be designed in VHDL.

The MC68000 possesses a 24-bit address bus, allowing 16 megabyte (MB) of in total addressable memory space. An 16-bit bi-directional data bus allows the MC68000 to communicate external on-board devices using asynchronous bus protocols defined by Motorola.

2. DESIGN APPROACH

There were many tasks to be completed and many research to be made before the implementation work of the CPU began. The first and most detailed task was to master the VHDL programming language. This was done by completing the course VHDL Modeling, then the book of "VHDL Analysis and Modeling of Digital Systems" [1] by Zainalabedin Navabi and its exercises are completed. A selection of VHDL exercises using primitive components and CPU design are completed. This coupled learning VHDL with re-familiarization of digital logic concepts.

Then the open core and fpga resources also provided a basic tutorial on how the Xilinx FPGA design and simulation platform is used. The next task was to research and learn CPU design implementation. A large part of this research work involved the study of Microprocessor Architecture and Advanced Digital Design.

To obtain an efficient and comfortable level of VHDL proficiency, before turning my attention to the 68000 microproessor, I have decided to design a simpler one named Parwan as described behaviorally in "VHDL Analysis and Modeling of Digital Systems" [1] by Zainalabedin Navabi. It was very easy to convert the VHDL code of the Parwan microprocessor to structural form as each part and every design step were fully described in the book. After simulating the Parwan microprocessor using various testbenches, it became obliged to start working on the 68000 microprocessor.The 68000 Programmers manual and the 68000 users manual were invaluable resources when it came to detailing 68000 internals and operation, especially when it came to instruction decoding and bus cycles. The Motorola instruction set was examined and broken down into smaller, more visibly structured code parts from which decisions about instruction decoding and hardware support were made.

Once this initial research had been completed, the hardware design process could start. The design is implemented entirely in VHDL using Leonardo Spectrum and Mentor tools for programming and synthesis together with ModelSim 5.6 and Symphony EDA VHDL Simuli for simulation. The PARWAN CPU modeled previously as described by Zainalabedin Navabi in VHDL Analysis and Modeling of Digital Systems[1] has been extensively adapted to create a VHDL CPU capable of executing Motorola 68000 op-code. At this step of the design I have decided to add new functionalities to the 68000 microprocessor so that it would be able to implement the full functionality of the original 68000 microprocessor with improved instruction code and additional functions. These function include hardware self testing, optimizing the memory and cache structures, and if possible having empty instruction codes to be used by the programmer to increase the efficiency of the microprocessor. Adding these instructions to the original CPU was a difficult task so again the Parwan CPU is used to test and verify the capabilities and efficiencies of these functions.

The 68000 CPU has been built in an incremental fashion starting with the register file. Each individual module has been programmed and tested under a behavioural simulation to verify functionality and then a post-synthesis simulation is applied to verify the result that could operate on an FPGA. The design approach taken is inline with the "VHDL Analysis and Modeling of Digital Systems" [1] by Zainalabedin Navabi. This design flow was applied to every component of the project from basic shifter to CPU package. The state machine design used together with the register transfer logic programming provided the Parwan CPU code to be easily converted to the 68000 CPU by redesigning ALU and instruction set and adding the extra registers and ports. Thus redesigning a simple core into a complex one easily with this method, gave birth to a core generator which can be used to create CPU's by defining the states, signals, registers and busses accordingly with a proper functional unit design.

2.1. Using Different Design Approaches for Microprocessor Design

The successful redesign of a microprocessor in VHDL, requires a suitable methodology to be used for the entire process [1]. First of all VHDL has three different approaches for hardware design; behavioral, dataflow and structural.

Behavioral design is the functional design of the hardware using variables and standard logic algorithms. This design methodology simply describes the functionality of the hardware, and is not suitable for hardware design. It is generally used in the early phases of a design for taking a glimpse of the result, and for verification and readability of the design. Behavioral Designs are very useful for fast functional tests and functional verification. Timing is bad, and mostly not synthesizable, and when synthesized produces unefficient results since the design is sequential, not concurrent. The dataflow methodology describes the hardware using the dataflow between busses and signals with proper timing. Sometimes called as register transfer logic this methodology describes the hardware with preserving all functionality and timing, by describing every state of the registers and signals at every clock pulse, which results in larger gate numbers and chip size than the original core. Dataflow designs are useful for testing timing constraints of the design. Generally synthesizable, and produces very efficient results if coded properly.

Structural design is the closest one to the real hardware design as it describes every component at the initialization phase then maps the connections between these components. In structural design, the hardware is fully specified, designing every combinatorial unit using gate level decriptions and connecting them properly which makes it a pure hardware design used for production and tests before production. Since structural design uses the same amount of detail and work effort as the original microprocessor design done by Motorola engineers, it creates no surplus as a science project and adds nothing to the technology of electronics. That's why it would be

meaningless to choose this type of VHDL programming as the main design methodology which directed me to use a dataflow design using register transfer logic with a control logic structure like the microprogram control; using nothing more than the real signals and registers in the original core, with their full functionality, which provided the core to produce the same responses as the original microprocessor.

This coding style allowed the registers and busses to be declared in a database structure, and also defined their relations using the most basic elements of VHDL, *"case"*, *"when"* and *"if"*, which allowed the code to be concurrent and synthesizable. Register Transfer Logic (RTL) type VHDL programming also lets the programmer to select the CPU state logic of the microprocessor, independent of the real core. Below are the examples of the design methodologies that can be used for building a CPU in VHDL.

- Basic Behavioral Description of a CPU

  Loop

  Fetch;

  Execute;

  Check for Interrupt;

  End;
- Behavioral Method Using IF-ELSE Statements

  BEGIN

  PROCESS

  Declare Variables

  BEGIN

  IF interrupt then handle interrupt;

  ELSE

  Read data

  Increment PC

Execute instructions

IF instruction1 then execute instruction1;

:

:

endif;

endif;

End Process;

- Behavioral Method Using Procedures

  Procedures are written for read,write,addressing modes, and ALU operations.

  BEGIN

  CASE DECODE(IR)

  WHEN MOVE GET SOURCE ADDRESS PROCEDURE

  READ PROCEDURE

  GET DESTINATION ADDRESS PROCEDURE

  WRITE PROCEDURE

- Concurrent Design using Microprogramming Architecture

  Originally proposed by M. V. Wilkes in 1951. Systematic control logic design method in which control signals are generated using a low-level program (a micro program) stored in a fast on-chip ROM (referred to as control memory (CM))

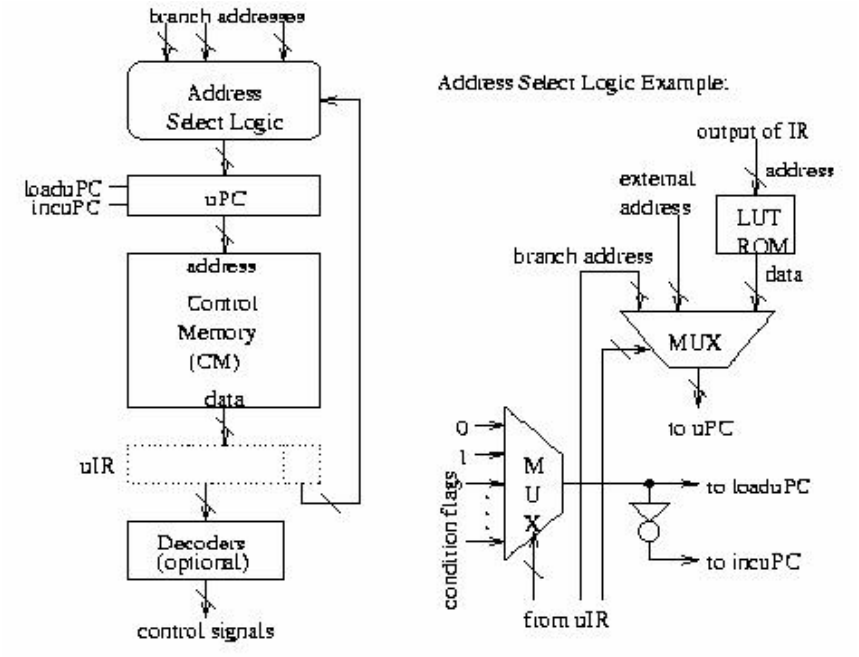

Figure 2.1. Concurrent design using microprogramming architecture[10]

- Dataflow Method using Signals Controlling Bus Transfer Logics

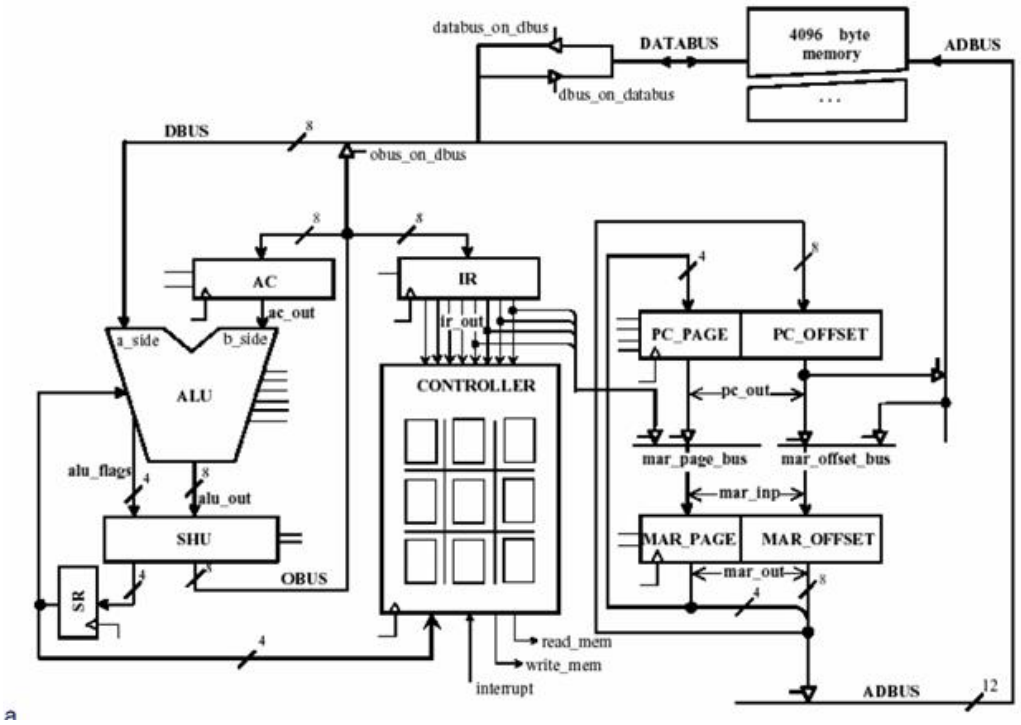

Figure 2.2. Dataflow method using signals controlling bus transfer logics[1]

The most suitable CPU design for a project like this would be a state machine design or Systematic control logic design method in which control signals are generated using a low-level program (a micro program) created by CPU . The state machine design is chosen with a dataflow type VHDL programming using register transfer logic with a control logic structure like the microprogram control, as it is more suitable for creating a core generator and it does not require additional circuit components like a ROM which is the key component for programming the micro codes into the CPU. Combining these two structures provided a very efficient result:

- The code is synthesizable, no unwanted latch and multisource.
- No variables are used only signal assignments and case logic, providing concurrent and stable design.
- The model is always in a known state during operation.
- Whenever data need to be stored, registers have been instantiated, so no need for unneccessary latches.

2.2. VHDL Programming

The design is implemented wholly in VHDL, mainly to maintain compatibility with other CPU cores on market. VHDL is chosen because it is a more reader-friendly language than the alternative, Verilog. VHDL is a hardware specification language that infers a hardware model by reducing a hardware design specified in a software model to primitive components[2]. VHDL is used for the description and modeling of digital systems; the description of a component in VHDL consists of an interface specification and an architecture specification. VHDL allows multiple architectural specifications, and allows an architecture to be configured for a specific technology or environment. Using physical parameters and analog design are also possible in VHDL. Also, VHDL provides the user with the chance to design and test any kind of hardware and while accomplishing this task, the testing capabilities combined with the SoC and FPGA

implementations makes VHDL a perfect tool for hardware design. This automatic hardware compilation is a very powerful way, although not always resource-optimal way, of creating a functional hardware design from a specification without having to wire primitive gates manually.

VHDL uses event-driven simulation, only the affected component changes as concurrency is the main concept in VHDL design. It means that instantiations and transfer statements are all executed such that in the end they appear to have been executed simultaneously. Concurrency is achieved by executing each statement only when an event occurs on a signal that the statement is sensitive to. Using generic parameters and configuration aspects a hardware can be redesigned and tested when necessary.

2.2.1. RTL Style

A key requirement for the success of this project is the synthesizability of design. Not only would a synthesizable design be downloadable to the target project board but would prove that a somewhat efficient and modular design is in place. In this phase of the design an efficient way of VHDL programming is essential. Every function used in the design of the CPU affects the resulting gate number, and clock frequency in a positive or negative way. This creates another problem for the designer; writing the code to represent the full functionality of the CPU is not the key requirement of a good design, but a synthesizable and efficient code is more valuable.That's why although VHDL supports many language constructs and methodologies only a subset of it is synthesizable and should be used in the design of the microprocessor.

The only VHDL language constructs (aside from declaration constructs and declaration assignments) used in the design are process, case, if, then, else, and concurrent signal assignment. Using these constraints do not only to create an efficient, synthesizable design, but a readable one capable of being modified and enhanced in the future.

With this in mind a Register Transfer Level of abstraction has been kept consistent throughout the design. As a result both synchronous and combinatorial logic has been defined through use of concurrently running processes. The sample code listing below shows an example of using a process to infer a multiplexer.

```
⋮

– Source 68000.vhd
– Infer a multiplexer to manage the instruction register load enable
case ir(3 downto 0) is
when "1100" =>
– ori = or immediate data to ccr
dbus_ctrl <= idle;
addr_ctrl <= idle;
databus_ctrl <= idle;
⋮
```

Care has been taken to avoid the inference of unwanted latches at every stage of the project thus ensuring that the 68000 Microprocessor is always in a known state during operation. In fact, the only use of latches is in the bus interface when latching lower order bytes of a higher order operation and when bus interfaces are driven by multiple sources. Wherever data need to be stored, registers have been instantiated. The register states are converted to binary numbers automatically by the synthesis tool which converts the register transfer logic style into a concurrent synthesizable gate level design.

2.2.2. Modeling with VHDL

VHDL provides an efficient , Computer Aided Drafting (CAD) oriented methodology for implementing digital designs. In the not so distant past, design engineers

were forced to develop logic functions using Karnaugh maps putting together their designs logic gate by logic gate. During the sixties and the seventies system level and microcomputer design entailed building systems out of many individual logic gates manufactured on Integrated Circuits (IC). This design technique was very costly and time consuming. As the technology moved forward from medium-scale integration (MSI) to large-scale integration (LSI), to very large-scale integration (VLSI), the need for new design tools became apparent. With the ability to incorporate many functions on one IC, engineers needed a way to quickly design a function and or circuit and test the design.

The standard known as VHDL, was first created in 1987. VHDL[3] allows for hardware description in a text based language. VHDL is similar to Ada, a standardized, portable, and object oriented software language. VHDL allows a design to model a digital system at many levels of abstraction. A description can be as simple as a 2-input logic circuit or entire digital system. As devised by the DARPA/TRI-Service Rapid-prototyping of Application Specific Signal Processors (RASSP) and some aerospace contractors, there exists taxonomy for VHDL models . There are five different levels of modeling: Performance modeling, Behavioral modeling, Instruction Set modeling, Register Transfer Level (RTL) modeling, and Gate-Level modeling. Performance modeling is used to specify the general make up of a complete system. Items that might be specified in performance modeling could include frames per second for a video processor or number of instructions per second for a processor. Behavioral modeling adds one level of detail. A Behavioral model will include a break down of the system into subcomponents and functional blocks. There is little detail to how the functions will be physically implemented.

The next level of detail is included in the Instruction Set model. This level of detail provides a model that can be used to simulate the instruction set of the microcontroller, microprocessor, or Digital Signal Processor (DSP) chip in question.

RTL modeling is a level of detail that specifies exactly how data will transfer from register to register. RTL is the most useful level of modeling to a designer who is going to actually implement his or her design in an Application-Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FGPA). There are many synthesis tools that will *"synthesize"* or translate the RTL model into a netlist of primitive technology cells for placement and routing. There has been a great deal of focus in the area of synthesis since this is where the designer can save many hours of work. Instead of drawing thousands upon thousands of gates in a schematic, the VHDL savvy designer can describe a design in a text-based language that can be automatically transferred to a schematic in a matter of seconds.

The final level of detail is the Gate-Level model. After synthesis is complete, the result is a netlist of primitive gates like NAND gates and NOR gates. This netlist can be simulated with timing information provided by the vendor of the target silicon. This simulation is called post synthesis pre place and route simulation.

2.2.3. Design For Testability with VHDL

Testability is one of the greatest advantages of using a HDL to design your microprocessor. The user can verify its project behaviorally before even starting the hardware design. Then at every stage of the design the user can test and verify the functionality of the components part by part. After the design is fully tested and synthesized , the user can create one or more testbenches to test and verify all the timing responses and every possible hardware state. VHDL uses simulators to test and verify the code created by the programmer. A VHDL simulator makes the user think that all executions are being done concurrently. For a successful simulation to occur for your design a proper testbench must be created. A testbench must contain the circuit under test and should have sources for providing data to its inputs. If desired, a specifially created testbench can scan every possible hardware state, which is very crucial when

using VHDL for designing new hardware.

Designers must develop a system-level test strategy before synthesizing and producing their core to prevent any working fault. The general method for building system-level test strategy is building test sets for cores, which generates functional vectors and fault grade interconnects. Also the designer must prepare cores for test application from primary inputs through access. If a Built-in Self Test (BIST) structure is present, the designer should schedule BIST application and a signature analysis[4].

3. MOTOROLA 68000 MICROPROCESSOR OVERVIEW

The 68000 known as a 16-bit CPU is internally a 32-bit CPU. The CPU has several features that make it a unique design. It has 32-bit program counter with no paging or segments and a 16-bit external data bus which needs 16-bit ROM and RAM for system. The CPU can access 8 or 16-bit memory and peripheral devices. It has a 16 megabyte (MB) linear addressing range (23-bit plus Upper and Lower data strobes for an effective 24-bit range). It has 56 instruction types, exception programming, seven levels of interrupts, tracing function, asynchronous bus structure, non-multiplexed address and data buses, bus arbitration control circuitry.

The eight 32-bit data registers, numbered D0-D7, are provided for data manipulation. In addition, 32-bit address registers A0-A6 are designed to hold memory addresses in order to aid data movement within the system and increase the complexity of memory addressing modes. Register A7 is a special case of such a register. It is known as the system stack pointer (SSP) when handling system control instructions (i.e. in supervisor mode), and user stack pointer (USP) in user programming mode (i.e. executing user code). A 32-bit program counter, of which the lower 24 bits are valid provides memory addressing for the currently executing instruction. A 16-bit system register holds information necessary for system functions such as branching condition codes and interrupt information.

The lower 8-bits, or user byte, hold condition codes such as overflow which are generated by the functional unit. The higher order byte, or the system byte, holds the current interrupt level priority mask used in peripheral control, the trace bit (whether or not to call a trace exception after each instruction) and the supervisor bit which is set when performing system control tasks. Fourteen addressing modes are available for the instructions.

Direct mode addressing transfers the data to or from any of the data or address registers. Indirect addressing is performed using an address register which contains the address in memory that that data to be manipulated resides. Post increment, pre-decrement and displacement variations of indirect addressing are implemented as modes useful during loop instructions and stack manipulation. Immediate addressing involves specifying an operand directly i.e. by encoding it in an extension to a core instruction word. Absolute addressing is similar with the exception that the immediate data specify a location in a specific part of memory.

The Motorola 68000 instruction set is very much a CISC entity. The 68000 boasts powerful and numerous addressing modes and multiple-cycle instructions such as multiply and divide. Such features are not found in comparable RISC processors, such as the SPARC-II or the Parwan Microprocessor designed previously, which have a load/store architecture with a minimal instruction set. The eighty-one (counting branch as one and omitting no-op) instructions available to the 68000 programmer can be grouped loosely into four categories. Data Transfer Group, Arithmetic, Logical and Bit-Manipulation group, Program Control Group and System Control Group. The Data Transfer Group includes instructions to move data between memory and the processor and also between internal registers. There are also instructions, such as LINK and PEA that aid memory manipulation and the use of stack data structures.

As is expected of a CISC instruction set, arithmetic instructions are plentiful. The 68000 is capable of manipulating 8, 16 or 32-bit signed operands. Instructions include ADD, SUB, MULU (Multiply Unsigned), DIVS (Divide Signed),TAS (Test and Set) and CMP(Compare). The 68000 is also capable of performing simple arithmetic on binary coded decimal. Shift operations are also available to the programmer. ASd (Arithmetic Shift) and LSd (Logical Shift) as well as ROd (Rotate)are the main shift instructions.

The last group deals with system control and features instructions that reset the processor, call external subroutines, handle exceptions and manipulate stack pointers. Examples include RESET, AND to CCR, ILLEGAL and TRAP.

An instruction can have no, one or two operands which can be encoded in the instruction or specified using one of the addressing modes. Instruction size, op1, op2 is the general syntax for two operand instructions. Instruction is any two operand instruction defined by the set, size is one of B, W, L for Byte, Word and Long operation and Op1 and Op2 are effective address that specify the location of the operands and the location to place the result, which is in most cases op2. One and zero operand instructions take the same form, e.g RESET or NEG (A0). A more detailed description of the 68000 instruction set is considered as the design of the VHDL CPU instruction decoder is explained in chapter 2. An exact explanation of the instruction set can be found in the Motorola 68000 Programmers Manual.

3.1. Hardware Design

Although implemented in VHDL this project is functionally a hardware based project. This chapter explores in detail the 68000 in terms of the hardware implemented to create the functioning model. At first, the functionality of the design is explained followed by how such a design is implemented on the VHDL platform. Initially the design of the data-path including functional unit and register file is considered. Next, Bus Interfacing as well as sequencing and control hardware are examined. Where code listings occur, the source file is quoted and can be found on the accompanying CD in the 68000 folder.

3.1.1. Overview

The data-path is modified to perform arithmetic on 8-bit, 16-bit and 32-bit signed or unsigned integers. It was also necessary to add support for the correct generation of condition codes as specified by Motorola [5, 6, 7, 8, 9]. The control path is modified to cater for 16-bit Motorola instruction decoding, the addressing modes outline earlier and CPU interrupt and exception handling. All internal data transfer occurs through a 32-bit bus. The lower order byte of this bus is valid for a byte (8-bit) operation and the lower order word is valid for word (16-bit) operation. The remaining bits are either 0 or to be ignored. If a register is being written using a byte or word operation the lower 8 or 16 bits are the only ones affected by the operation. The unaffected registers are kept idle to prevent unnecessary latches.

It is necessary to be able to perform the essential arithmetic and shift operations in this module. Condition code generation must also be considered. The Functional Unit consists of two main components, an ALU and a shifter unit. Two inputs, left and right feed the ALU operands and the shifter. A size control input signal is also needed to distinguish between operation sizes. This is particularly important for the rotate instructions of the shifter and for correct condition code generation. In keeping with the Motorola instruction set "00" represents a byte operation, "01" a word operation and "10" a long-word operation. The ALU is fed simultaneously like other registers and generates a result and appropriate condition codes.

The operations supported provide functionality identical to the Motorola instructions add, subtract, logic and shift operations. CC block generates negative (N) and zero (Z) condition codes. The Z condition is generated by examining the result and setting a flag if the lower order 8, 16 or 32 bits are all zero. In twos complement arithmetic , the most significant bit determines the sign. If it is 1 then the integer can be interpreted as negative.

3.1.1.1. RTL Logic Used. VHDL supports many language constructs and methodologies, but only an effective dataflow or structural mode programming is synthesizable and should be used in the design of the microprocessor. As mentioned in chapter 2 "the Design Approach" , a specific RTL logic is used for VHDL programming. The only VHDL language constructs (aside from declaration constructs and declaration assignments) used in the design are process, case, when, if, then, else, and concurrent signal assignment. These constraints provide the design to be coded simply but by describing every state and signal assignment during the CPU states, which creates a great load of code, as there are 56 instructions with over 1000 useful permutations. To code this structure without error, and in an efficient manner, a code generator is designed for building the code. This approach do not only to create an efficient, synthesizable design, but a readable one capable of being modified and enhanced in the future. The RTL logic is used by changing the necessary bits of the registers at every state of the CPU, thus using latches only when latching lower order bytes of a higher order operation and when bus interfaces are driven by multiple sources. Wherever data need to be stored, registers have been instantiated. The register states are converted to binary numbers automatically by the synthesis tool which converts the register transfer logic style into a concurrent synthesizable gate level design.

3.1.1.2. The Operation Principles of The Final Model. The model chosen for Microprocessor design consists of the main CPU registers, busses and ports declared as signals, and the control unit structure declared as multiplexers controlling the way the data move between the busses and the registers. The multiplexers also control the behaviour of the functional units including the arithmetic and logic unit, shifter and the status register. This design style is mainly known as Register Transfer Logic but, the model used in the project is a modified RTL to be used with the core generator.

The schematic description of the RTL can be used to understand the operation. The data pass from the databus to the registers and from registers to the databus using

multiplexers commanded by the control unit. The simple fecth-decode-execute cycle of the processor can be simply detailed using this model. After the reset operation which will be detailed later in the core generator part the processor commands the multiplexer controlling the program counter register operation to put the data in program counter to the databus, by opening the gates, and at the same time the processor also commands the multiplexer controlling the effective address register to load the databus to the register by again opening the gates. So the control unit copies the data from the program counter to the effective address register by changing two multiplexer signals. After the operation is complete, using the same methodology the control unit commands the effective address register to copy the register content to the address bus, and then the control unit starts the read cycle. During the read cycle the control unit directly controls the port signals without altering any multiplexer or register only by driving one or zero to the ports directly according to the state number of the read cycle. When the read operation is complete, the control unit commands the multiplexer controlling the instruction register to get the data from the databus by opening the gates so that the data read from the memory can be copied to the instruction register.

The processor then decodes the data on the instruction register, saves the instruction state accordingly, and jumps to the source effective address calculation state which calculates the source operand effective address according to the addressing mode encoded into the instruction opcode. If the state has a non operand instruction the processor directly jumps to the instruction execution state at this point. After the source effective address is calculated and the source data is fetched, according to the destination addressing mode the processor jumps to the destination effective address state or the instruction execution state. The destination effective address state calculates the effective address according to the addressing mode and reads the destination operand data if an operation would be made using this data, otherwise like in the move operation the source data is directly copied to the destination effective address calculated in the destination effective address calculation state. Then the processor jumps

to instruction execution state which in most cases uses the Arithmetic and Logic Unit to modify the destination data using the proper operation selected by the multiplexer controlling the Arithmetic and Logic Unit. After the operation is completed , the result of the arithmetic logic unit operation is copied to the databus again using the gates and then written to the calculated destination effective address which completes the main fetch-decode-execute cycle of the processor. At this point the interrupt check state controls the IPL input signals with the PPL embedded in the status register to check for interrupts. The control unit accomplishes this operation by copying these three bit datas to the ALU inputs using the databus and then comparing them using the Arithmetic and Logic Unit. If the result points to a proper interrupt request then the control unit jumps to the interrupt acknowledge cycle which pushes the PC and SR on the supervisor stack and jumps to the vector address calculation state. At this state if a DTACK is asserted the slave device the control unit fetches the calculates the interrupt vector by fetching it from the memory otherwise, the control unit automatically calculates the auto vector number according to the IPL level. After the vector number is calculated or fetched it is copied from the databus to the effective address register using the multiplexer controlling the effective address register to open the gates between the databus and the effective address register by the command *"load"* which is common in all of the register controlling multiplexers to get the data from the databus. This signal can be compared to the *"databus on register"* and *"register on databus"* signals used in the Parwan Model by Zainaleddin Navabi [1] which copies the contents of the databus to the registers or the contents of the registers to the databus by using gates. Next state is a common read operation where the address data in the vector is read from the memory to the databus. After the read operation is completed the data are copied to the program counter, by using the multiplexer controlling the PC to issue the order *"load"*. At this point the processor had saved the old PC and SR into the stack, and have a new PC to start the interrupt acknowledge program. The CPU goes on with the read instruction state until it decodes an operand to be RTE, which tells it to return from the interrupt. When the CPU reaches this state it pulls the PC

and SR from the stack and goes on with the old program which was being processed before the interrupt. This is a brief description of the operation of the CPU model coded with the core generator, the bus error, exception processing and the instruction execution details will be defined in the following parts.

3.1.1.3. General Code Structure in The Model. The CPU model described in the previous section summarized the coding style combined with the register transfer logic in a brief manner. In this section the code will be examined deeper using parts of the generated CPU core, as examples. The signal declarations and the entity declarations are used to define the ports and the signals using standard logic package defined by IEEE. This package is very useful in realizing a real chip since it defines "Z" as high impedance state and "X" as unknown state in addition to standard "0" and "1" signal values. The entity declaration and the signal declarations used in the real core are shown below as examples.

⋮

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity
port (
dbus: inout std_logic_vector( 15 downto 0);
adbus: out std_logic_vector( 23 downto 0);
as: out std_logic;
rw: out std_logic;
uds: out std_logic;
lds: out std_logic;
```

```
dtack: in std_logic;
berr: in std_logic;
rst: inout std_logic;
halt: inout std_logic;
br: in std_logic;
bg: out std_logic;
bgack: in std_logic;
ipl: in std_logic_vector( 2 downto 0);
fc: out std_logic_vector( 2 downto 0);
clk: in std_logic;
e: out std_logic;
vma: out std_logic;
vpa: in std_logic;
);
end;
⋮
```

The library declaration shows the path name of the library package being used and the use command enables the core to use the declarations [10, 11], functions and signals declared in the package. After the library declaration comes the entity declaration, where the ports of the CPU core is declared , using the standard logic package declared above, with the knowledge of the signal direction and the size of the port. The ports using more than one bits are declared as standard logic vectors with the bits counted from the highest to the lowest. The signal declarations used in the core are shown below.

```
⋮
```

```
signal databus: std_logic_vector( 31 downto 0); –internal databus
```

```
signal ir: std_logic_vector( 15 downto 0); –ir
signal dreg0: std_logic_vector( 31 downto 0); –data register 0
signal dreg1: std_logic_vector( 31 downto 0); –data register 1
signal dreg2: std_logic_vector( 31 downto 0); –data register 2
signal dreg3: std_logic_vector( 31 downto 0); –data register 3
signal dreg4: std_logic_vector( 31 downto 0); –data register 4
signal dreg5: std_logic_vector( 31 downto 0); –data register 5
signal dreg6: std_logic_vector( 31 downto 0); –data register 6
signal dreg7: std_logic_vector( 31 downto 0); –data register 7
signal areg0: std_logic_vector( 31 downto 0); –address register 0
signal areg1: std_logic_vector( 31 downto 0); –address register 1
signal areg2: std_logic_vector( 31 downto 0); –address register 2
signal areg3: std_logic_vector( 31 downto 0); –address register 3
signal areg4: std_logic_vector( 31 downto 0); –address register 4
signal areg5: std_logic_vector( 31 downto 0); –address register 5
signal areg6: std_logic_vector( 31 downto 0); –address register 6
signal sr: std_logic_vector( 15 downto 0); –status register
signal ear: std_logic_vector( 31 downto 0); –effective address register
signal temp: std_logic_vector( 31 downto 0); –temporary register
signal pc: std_logic_vector( 31 downto 0); –program counter
signal state: state_type; –
signal next_state: state_type; –
signal saved_state: state_type; –
signal return_state: state_type; –
signal state_stack: stack_type; –type stack_type is array(2 downto 0) of state_type;
signal st_ctrl: st_type; –
signal ipl_reg_ctrl: ipl_reg_type; –
signal be_ctrl: be_type; –
signal dbus_ctrl: dbus_type; –
```

signal addr_ctrl: addr_type; –
signal databus_ctrl: databus_type; –
signal ir_ctrl: ir_type; –
signal reg_ctrl: reg_type; –
⋮

The signals used internally in the CPU core also uses the standard logic type. The signals declared mainly show the registers, internal signals and the busses. No variables are used for achieving the concurrency in the CPU core and the signals declared and used internally are nothing more than the signals and busses used in the original Motorola 68000 CPU core. After the internal signals,registers and busses are declared the signals used for controlling the register multiplexers are declared. These signals do not use the type standard logic package, instead they use the types declared for register multiplexer states. For example the addr_ctrl signal commanded by the control unit, gets the state types of load, reset and idle as mostly all the register control multiplexers have in common. These state types are declared in the next part defined by their usage in the functional unit design which means that the state types are created after the functional unit is coded. The type declarations for the multiplexer and CPU states can be found below.
⋮

type state_type is (–control unit states
decode,
read_ir0,
ori_ccr,
ori_bwl,
ori_w_sr,
andi_b_ccr,

```
andi_bwl,
andi_w_sr,
type st_type is (
idle, –latch
push, –save return_state
pull –go to saved state
);
type ipl_reg_type is (
reset, –"111"
idle, –ipl
load –databus(3 downto 1)
);
type be_type is (
reset, –"0"
idle, –be
load –"1"
);
type dbus_type is (
reset, –"ZZZZZZZZZZZZZZZZ"
idle, –dbus
load, –databus
load_hi, –databus(31 downto 16)
load_lo, –databus(15 downto 0)
load_byte –databus(7 downto 0)– digerleri z olacak
);
type addr_type is (
reset, –"ZZZZZZZZZZZZZZZZZZZZZZZZ"
idle, –addr
load –databus
```

);
⋮

The state_type is the set of CPU states declared and used in the code generated. The state list under the declaration of state_type is produced by scanning all the states the CPU can go and then eliminating the same ones. That's why this type has the longest list of entries since it includes, the instruction execution states, addressing mode states, fetch-decode-execute states which have the largest operation since all the instructions are decoded and initial operations are made, interrupt acknowledge states, bus error states and the exception states , in simpler words all the states the CPU can use when it is processing.

The other type declarations show the register control multiplexer states, where the multiplexer controls the register content to get the value in the comment lines. For example; the multiplexer controlling the address bus uses the addr_type as the type listing, and causes the address bus to go to the high impedance value when reset command is given by the control unit, do not change when the command is idle and get the contents of the databus when the order is load. These three multiplexer state types are common in most of the register control multiplexers as mentioned before, and as one can guess the idle command is the most frequently used multiplexer state type. After the CPU states and the multiplexer states are declared, the function unit declaration is made where the register gets the values using the gates according to the multiplexer states, by only using concurrent signal transfers of VHDL. The code details can be seen below.
⋮

```
– Addr
–
_____-
addr_mux: process( clk, addr_ctrl, databus,next_clock)
begin
if (clkevent and clk = 0 and next_clock = 0) or (clkevent and clk = 1 and next_clock
= 1) then
case addr_ctrl is
when reset=>
addr <= "ZZZZZZZZZZZZZZZZZZZZZZZZ";
when load =>
addr <= databus(23 downto 0);
when others=>
addr <= addr;
end case;
end if;
end process;

_____-
⋮
```

The coding style used in the VHDL CPU is concurrent, including signal transfers, and `case` and `if` structures. No variables are used and the `case` structures scan all possible values to prevent the usage of unnecessary latches. In this particular example the *"addr_ctrl"* signal declared as a *"addr_type"* signal, gets the possible multiplexer states declared in the *"addr_type"* type declaration shown in the previous coding example. The possible states are *"reset"*,*"load"* and *"others"* which mean *"idle"* and the other possible states. *"Others"* command is a very important tool in VHDL as it shows all other possible values for the variable being scanned by the case operation and its

usage is mandatory in our coding style. The `case` structure automatically generates a multiplexer when synthesized, as one can notice, because it causes the address bus to attain certain values when the *"addr_ctrl"* signal changes to certain states, as a multiplexer does when its control signal changes. For example the address bus goes to the high impedance state when the *"reset"* order is asserted to the *"addr_ctrl"* signal of the multiplexer by the control unit, it loads the contents of the databus when the *"load"* order is asserted to the *"addr_ctrl"* signal and does not change in any other cases. One can easily wonder how the VHDL code gets the string variables as a signal assignment, as shown before in the type declarations part. To be more specific the *"addr_ctrl"* signal gets the values *"reset"*,*"load"* and *"idle"* as values instead of binary numbers. The answer is simple, and hidden in the last word of the previous sentence, the string variables are converted to binary numbers according to their number by the VHDL synthesis tool. For example in this particular example, probably the control unit will assert "00" to the addr_ctrl signal to give the *"reset"* order, "01" to give the *"load"* order and "11" to give the *"idle"* order when synthesized using a VHDL synthesizer. The usage of string variables as state types thus makes the code readable, prevents any errors to happen and helps the user to watch the simulation state by state as the VHDL simulation shows the states by their names, which provides an incredible time save when testing the produced code, as one mainly tests if the CPU follows the right order of states, and the registers get the right values. So the usage of state types makes the code more readable, easy to write and easy to test, which shows it is the best way to write a RTL type VHDL coding.The last and the longest part is the control unit declaration shwon below.

⋮

```
process( state, saved, ir )
begin
case state is
```

```
when decode =>
case ir(15 downto 12) is
when "0000" =>

when destEA_ind17 =>
if berr = '0' and be= '1' then
ipl_reg_ctrl <= idle;
be_ctrl <= idle;
dbus_ctrl <= idle;
addr_ctrl <= idle;
databus_ctrl <= idle;
ir_ctrl <= idle;
reg_ctrl <= idle;
source_reg_ctrl <= idle;
source_mode_ctrl <= idle;
dest_mode_ctrl <= idle;
size_ctrl <= idle;
usp_ctrl <= idle;
ssp_ctrl <= idle;
sr_ctrl <= idle;
ear_ctrl <= idle;
source_ctrl <= idle;
dest_ctrl <= idle;
temp_l_ctrl <= idle;
temp_ctrl <= idle;
pc_ctrl <= idle;
alu_ctrl <= nop;
st_ctrl <= idle;
return_state <= read_ir0;
```

```
next_state <= buserror;
next_clock <= '1';
cc_out_ctrl <= idle;
as_ctrl <= idle;
rw_ctrl <= idle;
uds_ctrl <= idle;
fc_ctrl <= idle;
else
ipl_reg_ctrl <= idle;
be_ctrl <= reset;
dbus_ctrl <= idle;
addr_ctrl <= reset;
databus_ctrl <= idle;
ir_ctrl <= idle;
reg_ctrl <= idle;
source_reg_ctrl <= idle;
source_mode_ctrl <= idle;
dest_reg_ctrl <= idle;
dest_mode_ctrl <= idle;
size_ctrl <= idle;
usp_ctrl <= idle;
ssp_ctrl <= idle;
sr_ctrl <= idle;
ear_ctrl <= idle;
source_ctrl <= idle;
dest_ctrl <= idle;
temp_l_ctrl <= idle;
temp_ctrl <= idle;
pc_ctrl <= idle;
```

```
alu_ctrl <= nop;
st_ctrl <= idle;
return_state <= read_ir0;
next_state <= destEA_ind18;
next_clock <= '1';
cc_out_ctrl <= idle;
as_ctrl <= reset;
rw_ctrl <= idle;
uds_ctrl <= reset;
fc_ctrl <= idle;
end if;
when destEA_ind18 =>
– load function codes, set rw and calculate EA
⋮
```

The last part of the code is the process where the control unit states are described in detail. The first thing one can realize is at every state, every register control multiplexer gets a value, which is the key to prevent unnecessary latches and code the CPU like a finite state machine which resembles the original Motorola 68000 CPU that uses a control ROM for the instructions. The control ROM provides values for the registers for every instruction and addressing mode programmed into a ROM, which is similar to the state machine coding style in our code. The first state shown above is the decode state where the instruction register is decoded using `case` structures and when an instruction is found the necessary register operations are made using the register control multiplexers and the next state is programmed for the instruction to be executed. The state control can push and pull states which means when an instruction is found using the `case` structure in the decode state, but the source and destination contents have to be read from the memory first, the control unit can save the instruction execution state as the return state and can jump to the read source or destination states. This

reduces the state numbers of the finite state machine, since the read, write and effective address calculation states are same for most of the instructions.

The second part of the code shows a destination effective address calculation program in its fourth state where the read operation will go on, wait states will be asserted or a bus error is asserted by the slave device. This differentiation is accomplished by using if clause embedded into the `case` structure which does not affect the concurrency of the code. One can see the next states and the register control multiplexer values change according to the if condition, and the proper next state is written below as *"destEA_ind5_state"*. The control unit coding declared above is a very long code since it declares every register control multiplexer value for every state or every case, which makes the overall code to be longer than thirty five thousand lines, which is nearly impossibe to write by hand without using a tool. So when I decided to use this coding style as it was the most suitable one to define a CPU in VHDL with the correct bus cycles, minimum gate number and minimum unnecessary latches, easily testable finite state machine design, and with a reoccuring coding style, I realized that I had to produce my own tool. That is how the core generator design came into life. In the fifth section the core generator will be described in detail.

3.1.1.4. The Synthesized Model. The model is synthesized using Leonardo Spectrum using the library Xilinx Virtex II with 2V80fg256 technology file [12]. The results show exactly the model designed in the previous parts which is expected as the coding style is uniform and efficient. The FPGA design shows the data and address busses in bold and shows the gates between these busses and the registers. Also one can see the register control multiplexers driven by the control unit on the left and right edges of the design. The ports are shown on the right and left side of the diagram, with the internal databus connected to most of them. The latches are minimum as expected.These details show that the design shown below perfectly resembles the code style used for the project and it is very efficient as expected instead of the very long

coding.

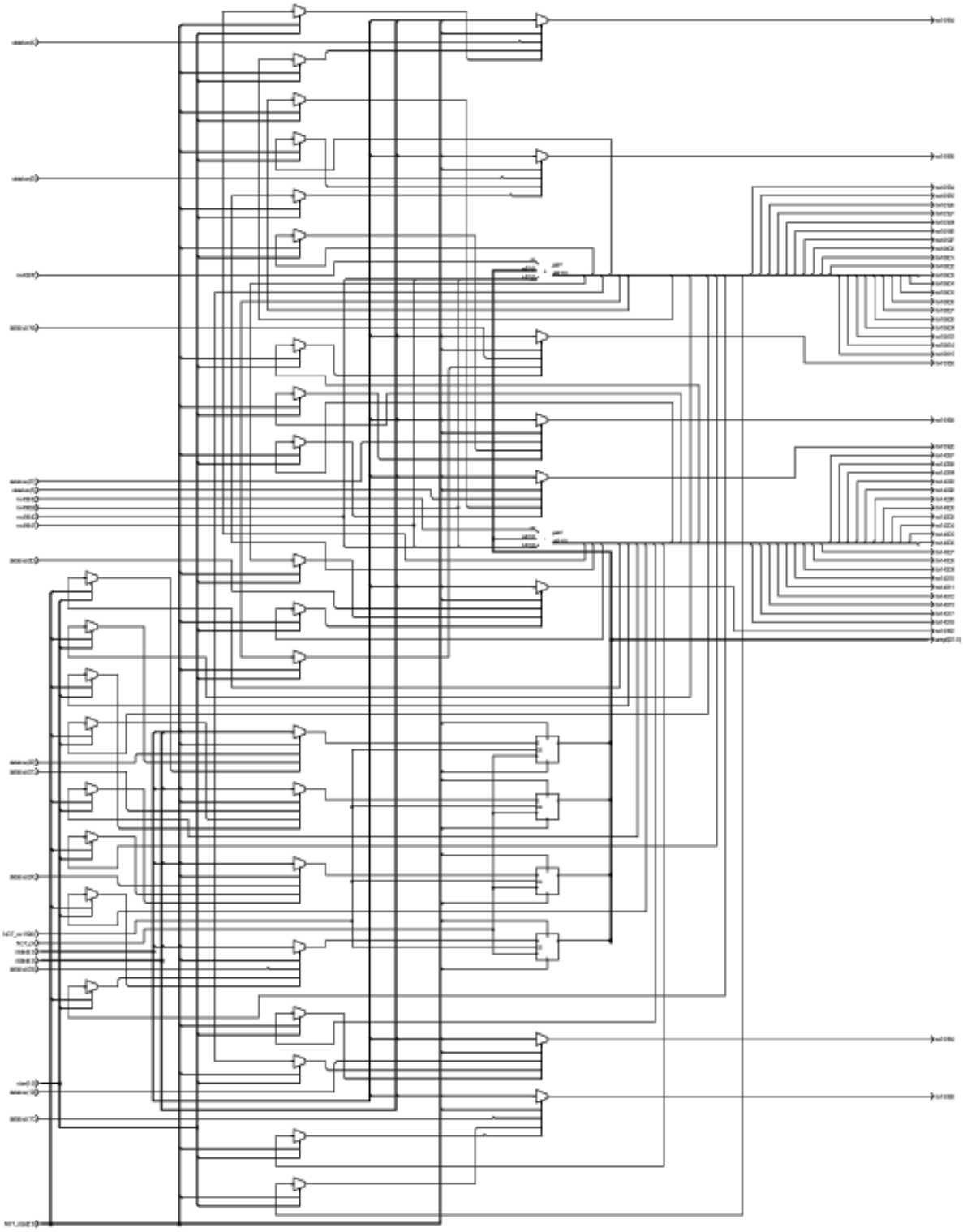

Figure 3.1. The synthesized model of the CPU generated

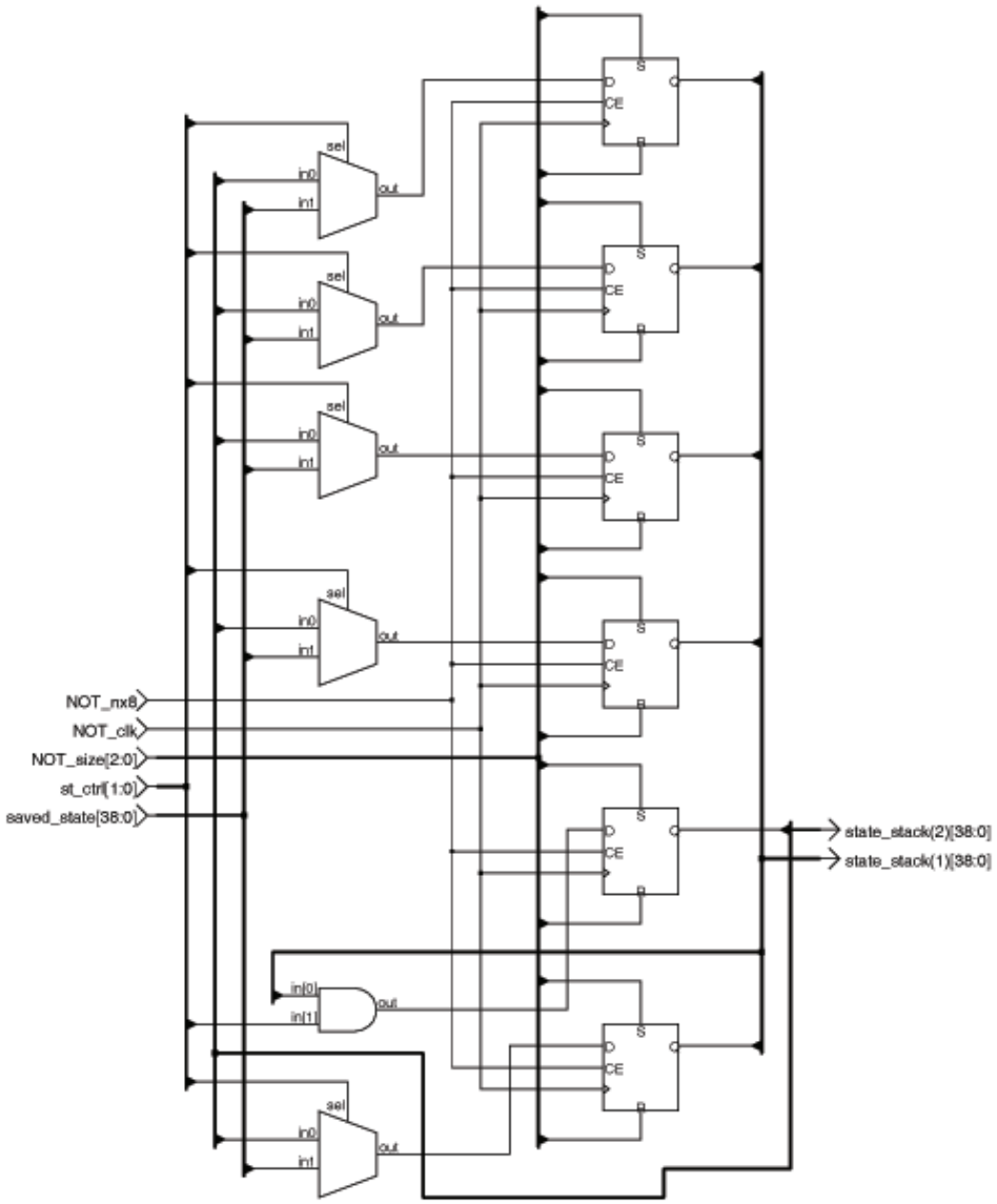

Figure 3.2. The state stack implementation of the synthesized model

4. CORE GENERATOR DESIGN

The general code structure detailed in the previous sections consist of five main uniform parts using the same code structure. The `case` structure and register transfer logic cause a very large coding which is nearly impossible to do using manual code writing techniques. Since the coding is standard for each part of the design and can be created automatically, the coding for each part can be distinguished into unique databases. Also while entering the data to the database, certain coding requirements like `case-when`, `if` structures and punctuation are not necessary. Another simplification can be done by changing only the necessary register control multiplexer values for each state, the other register control multiplexers will be commanded to be *"idle"* which can be done automatically. Another tricky and helpful part of the database is to produce the states and function tables automatically by only entering them once in a function state table. These optimizations and the structure of the core generator for each code part will be defined in detail in the following sections.

The core generator uses Microsoft Excel Worksheets as database entities [13]. It then uses visual basic script to convert the data in the database to a valid VHDL code, which also generates some of the database sections and controls the data entry sections for correct values automatically. The general structure is shown below in Figure 4.1.

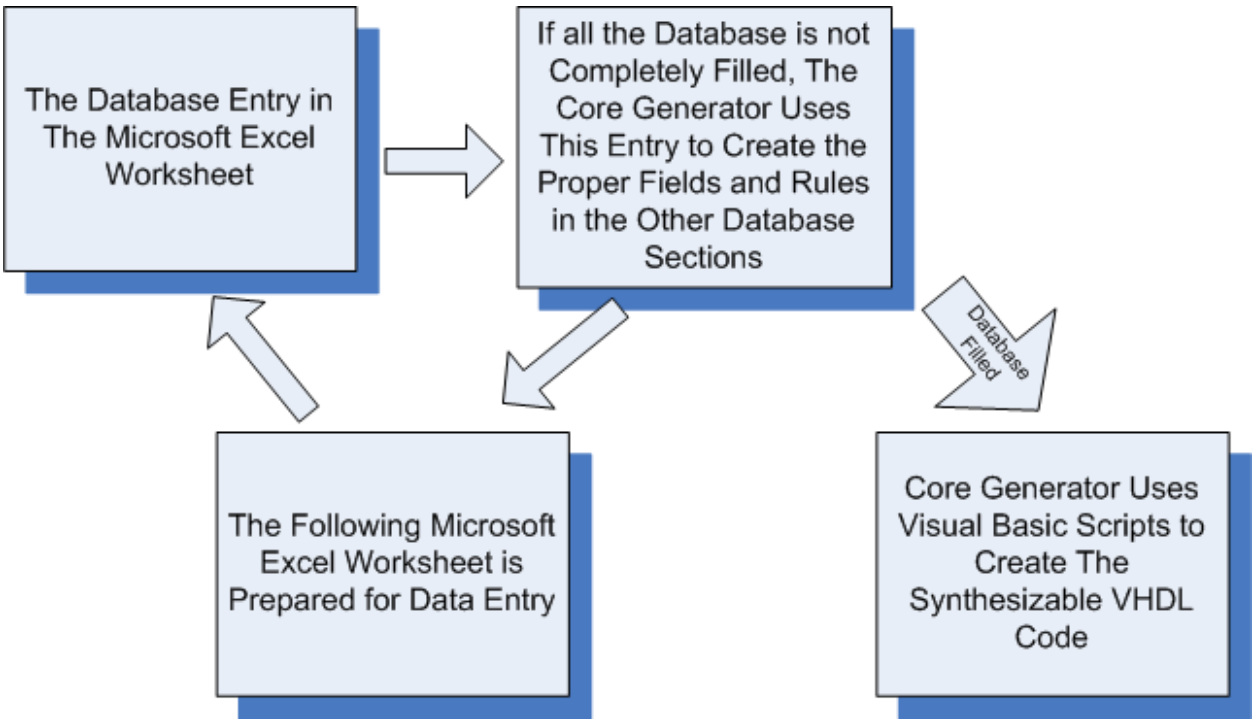

Figure 4.1. Core generator working principle

The core generator as its name implies can be used to create any CPU core written in advanced RTL logic as a finite state machine. The core generator can be modified for any CPU by only changing the database entries, and functional unit design. The way to create another CPU with the core generator designed for Motorola 68000 CPU code will be detailed in the following sections after describing the core generator structure.

4.1. General VHDL Model

The general VHDL code structure consists of entity declaration, signal and variable declaration and the architecture. The architecture consists of the processes using these signals and variables to produce certain results and to transfer data between these entities. The general VHDL model of our CPU core is detailed in subsection 3.1.1.3. In the following sections the details of each part of the model and how they are entered into the database of the core generator can be found.

4.1.1. Entity Declaration

The entity declaration of the CPU core is the simplest part of the code, including the names of the ports, the size of the ports in bits, and their signal type declaration which is standard logic package of IEEE. To generate this part of the code the user has to enter the port names, direction of the ports (in, out or inout) and the size of the ports into the database section named entity declaration. The sample database entry and the produced code are shown below in Table 4.1 and Table 4.2.

Table 4.1. Entity declaration using core generator

signal	number of bits	direction
dbus	16	inout
adbus	24	out
as	1	out
rw	1	out
uds	1	out
lds	1	out
dtack	1	in
berr	1	in
rst	1	inout
halt	1	inout
br	1	in
bg	1	out
bgack	1	in
ipl	3	in
fc	3	out
clk	1	in
e	1	out
vma	1	out
vpa	1	in

Table 4.2. Entity declaration VHDL code created by core generator

```
dbus:  inout std_logic_vector( 15 downto 0);
adbus:  out std_logic_vector( 23 downto 0);
as:  out std_logic;
rw:  out std_logic;
uds:  out std_logic;
lds:  out std_logic;
dtack:  in std_logic;
berr:  in std_logic;
rst:  inout std_logic;
halt:  inout std_logic;
br:  in std_logic;
bg:  out std_logic;
bgack:  in std_logic;
ipl:  in std_logic_vector( 2 downto 0);
fc:  out std_logic_vector( 2 downto 0);
clk:  in std_logic;
e:  out std_logic;
vma:  out std_logic;
vpa:  in std_logic;
```

The code that converts the upper table into the lower one is a basic Microsoft Excel function code, concatenating the singal name , number of bits and the direction of the port by adding proper VHDL constructs. The programmer can fill this database easily to use the core generator when creating other CPU's. The programmer needs to enter the correct port names to the first column, the size of the port by bits to the second column and the direction of the port to the third column. After the data are filled to the database section the user can see the code the core generator created on the right side of the database which enables him to check the coding of the CPU entity declaration.

4.1.2. Signal Declaration

The signal declaration part of the database is very similar to the entity declaration part. It uses the signal name, the size of the signal and the signal type. There is no direction for these signals as they are used inside the CPU. The signal listing below shows the internal registers, busses and necessary signals declared using the core generator. The core generator only needs the signal name and size for declaring the internally used signals, which is very easy to generate.

After the internal registers and busses are generated the multiplexer control signals need to be declared. These signals are mainly used to control the gates and the operations on the registers and the busses. Since these signals control the states of the registers, they are represented with state names instead of bitwise numbers. The state names are converted to bitwise numbers by the synthesis tool, according to the number of states that can be declared by binary numbers. A four state multiplexer control signal will be defined by a two bit signal by the synhesizer, which automatically replaces the state names with two bit numbers "00","01","10" and "11". These signals are entered into the database by using only one variable, state name or the register name. If the signal is a state controlling multiplexer the core generator adds a *"_state"*

clause to the end of the signal name, then declares the type of the signal as state_type. If the signal is register multiplexer control signal the core generator adds a *"_ctrl"* clause to the signal name in a logical fashion declaring it is a multiplexer control signal and declares the type of the signal as a *"_type"* clause added to the signal name. The state stack used for saving the instruction state when reading or writing the source or destination operands, is the only one which does not fit these rules. It uses a state type vector as a stack and declared accordingly.

The state types used in this part will be defined in the next part according to the multiplexer control signal values and the control unit states produced by the state machine design of the core generator.The tables showing the signal declaration in the database and the resulting VHDL code are shown below in Table 4.3, Table 4.4, and Table 4.5.

The code that converts the upper two table into the lower one is a basic Microsoft Excel function code, concatenating the singal name and the number of bits or state name according to the data type entered in the cell by adding proper VHDL constructs. The signal names and the multiplexer control signal names entered in this database section will be used to create the fields to be filled in the next database sections suitable to the working principle of the core generator show in Figure 4.1.

Table 4.3. Signal declaration using core generator

signal	number of bits	region controlled
databus	32	internal databus
ir	16	ir
dreg0	32	data register 0
dreg1	32	data register 1
dreg2	32	data register 2
dreg3	32	data register 3
dreg4	32	data register 4
dreg5	32	data register 5
dreg6	32	data register 6
dreg7	32	data register 7
areg0	32	address register 0
areg1	32	address register 1
areg2	32	address register 2
areg3	32	address register 3
areg4	32	address register 4
areg5	32	address register 5
areg6	32	address register 6
usp	32	user stack pointer
ssp	32	system stack pointer
sr	16	status register
ear	32	effective address regist
temp	32	temporary register
pc	32	program counter

Table 4.4. Signal declaration for states using core generator

signal	number of bits
state	state_type
next_state	state_type
saved	state_type
return_state	state_type
state_stack	stack_type
st_ctrl	st_type
ipl_reg_ctrl	ipl_reg_type
be_ctrl	be_type
dbus_ctrl	dbus_type
addr_ctrl	addr_type
databus_ctrl	databus_type
ir_ctrl	ir_type
reg_ctrl	reg_type
source_reg_ctrl	source_reg_type
source_mode_c	source_mode_type
dest_reg_ctrl	dest_reg_type
dest_mode_ctrl	dest_mode_type
size_ctrl	size_type
usp_ctrl	usp_type
ssp_ctrl	ssp_type
sr_ctrl	sr_type
ear_ctrl	ear_type
source_ctrl	source_type
dest_ctrl	dest_type
temp_l_ctrl	temp_l_type
temp_ctrl	temp_type
pc_ctrl	pc_type
alu_ctrl	alu_type

Table 4.5. Signal declaration VHDL code created by core generator

```
signal databus:   std_logic_vector( 31 downto 0); --internal databus
signal ir:   std_logic_vector( 15 downto 0); --ir
signal dreg0:   std_logic_vector( 31 downto 0); --data register 0
signal dreg1:   std_logic_vector( 31 downto 0); --data register 1
signal dreg2:   std_logic_vector( 31 downto 0); --data register 2
signal dreg3:   std_logic_vector( 31 downto 0); --data register 3
signal dreg4:   std_logic_vector( 31 downto 0); --data register 4
signal dreg5:   std_logic_vector( 31 downto 0); --data register 5
signal dreg6:   std_logic_vector( 31 downto 0); --data register 6
signal dreg7:   std_logic_vector( 31 downto 0); --data register 7
signal areg0:   std_logic_vector( 31 downto 0); --address register 0
signal areg1:   std_logic_vector( 31 downto 0); --address register 1
signal areg2:   std_logic_vector( 31 downto 0); --address register 2
signal areg3:   std_logic_vector( 31 downto 0); --address register 3
signal areg4:   std_logic_vector( 31 downto 0); --address register 4
signal areg5:   std_logic_vector( 31 downto 0); --address register 5
signal areg6:   std_logic_vector( 31 downto 0); --address register 6
signal usp:   std_logic_vector( 31 downto 0); --user stack pointer
signal ssp:   std_logic_vector( 31 downto 0); --system stack pointer
signal sr:   std_logic_vector( 15 downto 0); --status register
signal ear:   std_logic_vector( 31 downto 0); --effective address regist
signal temp:   std_logic_vector( 31 downto 0); --temporary register
signal pc:   std_logic_vector( 31 downto 0); --program counter
signal ipl_reg:   std_logic_vector( 2 downto 0); --ipl_reg
signal be: std_logic ;
signal source_reg:   std_logic_vector( 2 downto 0); --source_reg
signal source_mode:   std_logic_vector( 2 downto 0); --source_mode
signal dest_reg:   std_logic_vector( 2 downto 0); --dest_reg
signal dest_mode:   std_logic_vector( 2 downto 0); --dest_mode
signal source:   std_logic_vector( 31 downto 0); --source
signal dest:   std_logic_vector( 31 downto 0); --dest
signal temp_l:   std_logic_vector( 31 downto 0); --temp_l
next_clock:  next_clock std_logic;
signal size:   std_logic_vector( 1 downto 0); --size of the operation
signal cc_out:   std_logic_vector( 7 downto 0); --output of alu that uplo
signal alu_out:   std_logic_vector( 31 downto 0); --Output of alu

signal state:  state_type; --
signal next_state:  state_type; --
signal saved:  state_type; --
signal return_state:  state_type; --
signal state_stack:  stack_type; --type stack_type is array(2 downto 0
signal st_ctrl:  st_type; --
signal ipl_reg_ctrl:  ipl_reg_type; --
signal be_ctrl:  be_type; --
signal dbus_ctrl:  dbus_type; --
signal addr_ctrl:  addr_type; --
signal databus_ctrl:  databus_type; --
signal ir_ctrl:  ir_type; --
signal reg_ctrl:  reg_type; --
```

The programmer can fill this database easily to use the core generator when creating other CPU's. The programmer needs to enter the correct signal names to the first column, the size of the signal by bits to the second column and task of the signal as a comment to the third column. After the data are filled to the database section the user can see the code the core generator created on the right side of the database which enables him to check the coding of the CPU entity declaration.

For the register control multiplexer and state control multiplexer signals the programmer need to fill the state name and the state type to the first and second columns respectively. If there is a state name in the first column the database, the core generator automatically creates the *"state_type"* as a signal type. If there is no state name in the first column of the database the core generator adds a *"_ctrl"* clause to the end of the signal name to represent it is a multiplexer control signal and adds a *"_type"* to the second clause to represent the states the signal can have. The state_stack is an exception to this case and it is created automatically by the core generator without being added to the database. Again the programmer can see the code produced on the right hand side of the database section to check the coding.

4.1.3. Name Declarations for The Multiplexer Control Signals

This part of the core generator creates the state names for the state type declaration and the values of the registers for these states to be used in the functional unit declaration part. The database is composed of the state names as the row and the signal control multiplexers as the column. If the multiplexer uses a particular state to copy data or alter the register it controls, the database unit where the row of that particular state and the column of that register intersects is filled with the value of the register in that particular multiplexer state. The multiplexer controlling the address bus gets the *"load"* state as a signal from the control unit and with this order copies the contents of the databus to the address bus. This operation can be declared in the

database by entering the *"databus"* string to the cell where the *"load"* row and *"addr"* column intersects. The core generator scans the cells between the cells and rows for the intersecting ones and creates the state types for the multiplexers and state machine. The core generator also produces the base of the functional unit by assigning the value in the cell to the register or bus, when the multiplexer has the state in the intersecting row. The database entries in Table 4.6 and Table 4.7 and the produced source code is shown below.

Table 4.6. State name declaration using core generator

states	ipl_reg	be	dbus	addr
reset	"111"	"0"	"ZZZZZZZZZZZZZZZZ"	"ZZZZZZZ
idle	ipl	be	dbus	addr
load	databus(3	"1"	databus	**databus**
pull				
load_ear				
load_r				
load_w				
load_temp				
load_ipl				
load_ppl				
sub				
load_alu				
load_pc				
incr				
load_source_reg				
load_dest_reg				
load_source				
signex_word				
add				
load_hi			databus(31 downto 16)	

Table 4.7. State names created by the core generator

type ipl_reg_type is (	
	reset, --"111"
	idle, --ipl
	load --databus(3 downto 1)
);	
type be_type is (	
	reset, --"0"
	idle, --be
	load --"1"
);	
type dbus_type is (	
	reset, --"ZZZZZZZZZZZZZZZZ"
	idle, --dbus
	load, --databus
	load_hi, --databus(31 downto 16)
	load_lo, --databus(15 downto 0)
	load_byte --databus(7 downto 0)-- diğerleri z olacak
);	
type addr_type is (	
	reset, --"ZZZZZZZZZZZZZZZZZZZZZZZZ"
	idle, --addr
	load --databus
);	

⋮

```
addr: process( clk, addr_ctrl, databus,next_clock)
begin
if (clkevent and clk = 0 and next_clock = 0) or (clkevent and clk = 1 and next_clock
= 1) then
case addr_ctrl is
when reset=>
addr <= "ZZZZZZZZZZZZZZZZZZZZZZZZ";
when load =>
addr <= databus(23 downto 0);
when others=>
addr <= addr;
end case;
end if;
end process;
```

⋮

The code that converts the upper table into the lower two is a Microsoft Visual Basic script that scans the table of entries and enters the filled cells as types under their column name for producing the first table which is the state name declaration for the register control multiplexers. The functional code is produced by the values in the intersecting cells but needs to be modified, mostly for the arithmetic logic unit and the condition code register controls. The state names found for all the register control multiplexer signals by the core generator will be used for data validation in the

state table declaration database section when the user enters the multiplexer values for the states. The visual basic code used in this section is shown below for a better understanding of the core generator.

⋮

```
Sub function_Button1_Click()
arow = 1
acolumn = 50 'these can be designed to be variables to be assigned the number of rows
'and columns that are filled with data
arecord = 1
statenumber = 76

For mycolumn = 2 To 28

tPhrase = ""
tPhrase = "type "  Trim(ActiveSheet.Cells(1, mycolumn).Value)  "_type is ("
ActiveSheet.Cells(arow, acolumn).Value = tPhrase
arow = arow + 1

For myrow = 2 To statenumber

tPhrase = ""
If (Trim(ActiveSheet.Cells(myrow, mycolumn).Value)) <> "" Then
tPhrase = Trim(ActiveSheet.Cells(myrow, 1).Value)  ", –"  Trim(ActiveSheet.Cells(myrow,
mycolumn).Value)

ActiveSheet.Cells(arow, acolumn + 1).Value = tPhrase
arow = arow + 1
```

```
End If

Next myrow

ActiveSheet.Cells(arow - 1, acolumn + 1).Value = Replace(ActiveSheet.Cells(arow - 1,
acolumn + 1).Value, ", –", " –")

ActiveSheet.Cells(arow, acolumn).Value = ");"
arow = arow + 1

Next mycolumn
End Sub
⋮
```

The programmer can fill this database easily to use the core generator when creating other CPU's. The programmer needs to fill the first row with the register and signal names controlled by a multiplexer which are declared with a *"_ctrl"* suffix added to the signal or register name in the prevous signal declaration database. After the names are filled starting with the second column, the states these multiplexers can have as control signals will be filled vertically to the first column starting with the second row. The *"reset"*,*"idle"* and *"load"* states are standard for nearly every multiplexer so they are filled automatically in the core generator template. Other states used for the control multiplexers will be added by the programmer according to the CPU being created. The programmer also needs to fill the intersection cells of the states and signal names with the value, the register or bus will have, if the multiplexer can go to that state in that intersection row. After the table is properly filled with these values, the core generator automatically generates the multiplexer control type declarations and an initial functional unit code as shown above.

4.1.4. State Table Declaration

State table is the most important and the most complex part of the core generator which produces the main finite state machine architecture. The state table includes the state control and signal control multiplexers with their respective signal name as the first row of the state table declaration database. The `case` and `if` structure with the comment columns also have their place in the first row. The other rows of the table show CPU states, and the cells where the CPU state is intersected with the register control multiplexers and the if-case statements have the value of that multiplexer and if-`case` structure of that state. For example the code below is produced from the row 139 of the decode state of the CPU. The `case` structure is produced according to the database entries betwenn columns C and H and the register control multiplexer state values are filled according to their values on that particular row. The database entry and the produced code are shown below in Table 4.8 and Table 4.9.

Table 4.8. State table declaration using core generator

case	sig	up	dow	value to be checked	comment	databus	size	alu	st	next_state
case	ir	7	4	"1100"	bos 3_	idle	idle	nop	idle	read_ir0
case	ir	3	0	"1000"|"1001"|"1010"|"1011"|"1100"|"1101"|"1110"|"1111"	dbcc	load_source_r	load_word	nop	idle	dbcc
				others	scc	idle	load_byte	nop	idle	scc
				"1101"|"1110"|"1111"	scc	idle	load_byte	nop	idle	scc
				others	subq_bwl	idle	load_zz	nop	push	subq_bwl
				others	2_	idle	idle	nop	idle	read_ir0
				"0110"	bos 1_	idle	idle	nop	idle	read_ir0
case	ir	11	8	"0000"	bra	idle	reset	nop	idle	bra_NOP
				"0001"	bsr	idle	reset	nop	idle	bsr_NOP
				others	bcc	idle	reset	nop	idle	bcc_NOP
				"0111"	1_moveq_l	load_dest_reg	load_long	nop	push	sourceEA0
				"1000"	bos 1_	idle	idle	nop	idle	read_ir0
case	ir	11	8	"0000"|"0010"|"0100"|"1000"|"0110"|"1100"|"1010"|"1110"	bos 2_	idle	idle	nop	idle	read_ir0
case	ir	7	4	"1100"|"1101"|"1110"|"1111"	divu_w	load_dest_reg	load_word	nop	push	divu_w
				others	or_bwl	load_dest_reg	load_zz	nop	push	or_bwl
				"0001"|"0011"|"0101"|"0111"|"1001"|"1011"|"1101"|"1111"	bos 2_	idle	idle	nop	idle	read_ir0
case	ir	7	4	"1100"|"1101"|"1110"|"1111"	divu_w	load_dest_reg	load_word	nop	push	divu_w

Table 4.9. State table created by the core generator

case ir(3 downto 0) is		
	when "1000"|"1001"|"1010"|"1011"|"1100"|"1101"|"1110"|"1111" =>	
		-- dbcc
		ipl_reg_ctrl <= idle;
		be_ctrl <= idle;
		dbus_ctrl <= idle;
		addr_ctrl <= idle;
		databus_ctrl <= load_source_reg;
		ir_ctrl <= idle;
		reg_ctrl <= idle;
		source_reg_ctrl <= load_ir4;
		source_mode_ctrl <= load_d;
		dest_reg_ctrl <= reset;
		dest_mode_ctrl <= reset;
		size_ctrl <= load_word;
		usp_ctrl <= idle;
		ssp_ctrl <= idle;
		sr_ctrl <= idle;
		ear_ctrl <= idle;
		source_ctrl <= load;
		dest_ctrl <= idle;
		temp_l_ctrl <= load;
		temp_ctrl <= idle;
		pc_ctrl <= idle;
		alu_ctrl <= nop;
		st_ctrl <= idle;
		return_state <= read_ir0;
		next_state <= dbcc;
		next_clock <= 1;
		cc_out_ctrl <= idle;
		as_ctrl <= idle;
		rw_ctrl <= idle;
		uds_ctrl <= idle;
		fc_ctrl <= idle;
	when others =>	
		-- scc

The code that converts the upper table into the lower table is a Microsoft Visual Basic script that scans the table of entries and enters the rows according to the register control multiplexer signal values and the `case` and `if` structures. The visual basic code used in this section is shown below for a better understanding of the core generator.
⋮

```
Sub Button5_Click()

arow = 1 'yazacagi satir
acolumn = 125 'yazacagi kolon
arecord = 10 'aktif olan kod

ifcounter = 0

'For arecord = 2 To 800
For arecord = 2 To 1223

'A kolonu icin
tPhrase = ""
If (Trim(ActiveSheet.Cells(arecord, 1).Value) <> "") Then
tPhrase = "when "  ActiveSheet.Cells(arecord, 1).Value  "_state => "

ActiveSheet.Cells(arow, acolumn - 1).Value = tPhrase
arow = arow + 1
End If

'B kolonu icin
tPhrase = ""
```

```
If (Trim(ActiveSheet.Cells(arecord, 2).Value) <> "" And Left(Trim(ActiveSheet.Cells(arecord,
2).Value), 3) <> "bos") Then
tPhrase = "– "  ActiveSheet.Cells(arecord, 2).Value

ActiveSheet.Cells(arow, acolumn + 2).Value = tPhrase
arow = arow + 1
End If

'C kolonu icin -3

If (Trim(ActiveSheet.Cells(arecord, 3).Value) = "case") Then
⋮
```

There is no chance for the core generator to produce any errors while producing these state codes as the columns under each register controlling multiplexer name can only have the values of the proper state names of the signal control or register control multiplexer. The core generator automatically gets these proper state names from the state name declaration database detailed in the previous section. For example the column labeled as *"addr"* can only have the state names *"load"*,*"reset"*,*"idle"* as declared in the state name declaration database, or can be left blank. Leaving the database cells blank has two main advantages. First of all the the programmer only changes the register controls that are affected in that particular state of the CPU. The other blank cells are assumed as idle by the core generator when generating the code. The second advantage is when reading the code using the database, the user can easily control the register states for the particular CPU state.

When filling the state table the programmer must fill the corresponding row according to the control state the CPU is operating. For example before reading the value of the instruction register, the address bus has to get the Program Counter value

as the proper address. To accomplish this, when the processor is in the *"read_ir0"* state the databus control multiplexer state is entered into the database as *"load_pc"* which enables the databus to get the value of the PC. At the same state the effective address register control multiplexer state is entered into the database as *"load"* which is the standard operation for all registers to copy the contents of the databus. The *"rw"* signal controller is also issued with the command *"load"* to start a read operation. These are the only multiplexer that has changed in this *"read_ir0"* state, and by entering these values to the database the programmer has finished creating this state. The core generator will automatically fill the empty cells with *"idle"* command thus issuing an order to every multiplexer at every state which prevents the use of unnecessary latches. Next state is filled with *"read_ir1"* as expected and there *"st_ctrl"* is *"idle"* as there is no need to push or pull a state onto the stack. The visual basic code filling the empty cells according to this rule is shown below.

⋮

```
Sub Button2_Click()


For arecord = 2 To 1223
For arow = 10 To 40
stringvalue = "idle"

If (arow = 31) Then
stringvalue = "nop"
End If
If (arow = 33 Or arow = 34) Then
stringvalue = "read_ir0"
End If
```

```
If (arow = 35) Then
stringvalue = "1"
End If

If (Left(Trim(ActiveSheet.Cells(arecord, 8).Value), 3) <> "bos") Then

If (Trim(ActiveSheet.Cells(arecord, arow).Value) = "") Then
ActiveSheet.Cells(arecord, arow).Value = stringvalue
End If

End If


Next arow

Next arecord

End Sub
⋮
```

In some states the operations change according to a signal or register value. For example the decode state is the biggest state as it decodes the Instruction Register using case clauses and changes the state values according to the instruction decoded. This is done by filling the "C" column with the `case` structure, then filling the "D" column with the name of the signal or register checked. After that the bits to be checked will be entered in the next two columns "F" and "G". The value of the case will be entered into the "H" column and the "I" column is the comment module. By this method, in the decode state the instruction register is decoded to find an instruction and the next

state signal is changed into the instruction name found. The inner register values are changed according to the instruction, size of the operation and the addressing mode to the state names entered into the state name database. If the states use if clause for distinguishing between states, if clause is written to the"D" column and the signal value to be compared is entered into the "E" column. If an *"else"* keyword is used, it is entered into the "C" column. The other columns are the same as the `case` structure. The current state is entered into the first column and the next state is entered into the "AH" column which can change according the CPU generated, since the register control multiplexer numbers will probably differ.

After the coding is completed by filling the proper cells in the state table the core generator fills the empty cells with the *"idle"* command automatically and then produces the state name list and the code. The state name list is produced by scanning all the values in the next state column where the state names of the following states are entered, and eliminating the same ones if there is any. Here is a part of the state name list of the Motorola 68000 CPU generated using the core generator, the visual basic code to generate this list is also given.

```
type state_type is (   --control_unit states
reset,
decode,
read_ir0,
decode2,
ori_ccr,
ori_bwl,
ori_w_sr,
andi_b_ccr,
andi_bwl,
```

Figure 4.2. State types created by the core generator

⋮

```
Sub states_vbalacak_Button1_Click()

Sheets("68000 states_vb alacak").Activate

arecord = 1
Do While ActiveSheet.Cells(arecord, 11).Value <> ""
ActiveSheet.Cells(arecord, 11).Value = ""
arecord = arecord + 1
Loop

targetrecordnumber = 1

For arecord = 2 To 1223

Sheets("68000 state table").Activate
temp1 = Trim(ActiveSheet.Cells(arecord, 1).Value)
temp2 = Trim(ActiveSheet.Cells(arecord, 33).Value)
temp3 = Trim(ActiveSheet.Cells(arecord, 34).Value)

If (temp1 <> "" Or temp2 <> "" Or temp3 <> "") Then
Sheets("68000 states_vb alacak").Activate

isduplicateTemp1 = False
isduplicateTemp2 = False
isduplicateTemp3 = False
```

```
'listeye eklediklerim arasinda kontrol ediyorum
For targetrows = 1 To targetrecordnumber
If temp1 = ActiveSheet.Cells(targetrows, 11).Value Then
isduplicateTemp1 = True
End If
Next targetrows
'eger tekrarlanmamissa listenin sonuna ekliyorum
If (isduplicateTemp1 = False And temp1 <> "") Then
ActiveSheet.Cells(targetrecordnumber, 11).Value = temp1
targetrecordnumber = targetrecordnumber + 1
End If

For targetrows = 1 To targetrecordnumber
⋮
```

4.2. System Programming with Core Generator

The declaration parts of the core generator are used to define the simple descriptions of entity and signals in a easier way without using much of an automatic generation script. The state name declaration part, the functional unit declaration and the most important part which is the state table declaration part are the parts where the core generator really achieves a great task of creating the finite state machine.

The state machine declaration database of the core generator is the place where all the states and the corresponding operations of that states are declared so in a manner it is the part of the core generator where the CPU is created. The state names, are entered into the first column and then the case or `if` structure used in that state is entered into the next five columns if there are any. After that the values of the control multiplexer values that are used in that state are entered to the cells

corresponding to the signal names in the columns. At last the next state and if any the return state is filled properly with the following state names. If the state machine will save a state or will jump to a saved state the *"st_ctrl"* multiplexer represented by the column labeled *"st"* will have a *"push"* or *"pull"* value accordingly. After the user fills the table properly the remaining job is done by the core generator to create the state machine code in VHDL. The functional unit should also be checked and if necessary altered by the user after it is generated by the core generator, using the state name declaration database, because especially the Arithmetic Logic Unit and the other special functional units can not be created properly by the core generator.

4.2.1. Write Cycle Created by The Core Generator

The write cycle is almost the same for all operations in the CPU and there is no need to enter the write cycles to the database more than once since the other states who will use the write state as the next state can use the state stack to jum to the write state and come back when it is finished, instead of writing a new write state.

The coding of the write cycle in the state table database is given in Table 4.10. Here we will do a decoding of the state table to check if the write states are written properly by comparing every state of the original write cycle with the one created by the core generator.

Table 4.10. Write cycle database entry of the core generator

STATE	case	if	signa	up	do	value	databus	return	fetch
write0									write1
write1							load_ear		write2
write2									write3
write3	case	size		1	0	"10"	load_dest		write4
						"01"	load_dest		write4
						others	load_dest		write4
write4									
		if	dtack	0	0	"0"			write5
	else	if	berr	0	0	"0"			write5
	else	if	vpa	0	0	"1"		write4	peripheral
	else							write4	wwaitt
write5							load_ipl		write6
write6									write7
write7		if	berr	0	0	"0" and be= '1'			buserror
	else	if	size	1	0	"10"			write8
	else								saved
write8									write9
write9							load_ear		write10
write10									write11
write11							load_dest		write12
write12									
		if	dtack	0	0	"0"			write13
	else	if	berr	0	0	"0"			write13
	else	if	vpa	0	0	"1"		write12	peripheral
	else							write12	wwaitt
write13							load_ipl		write14
write14									write15
write15		if	berr	0	0	"0" and be= '1'			buserror
	else								saved

The write cycle for byte and word takes four clock cycles which means it finishes at seventh write state as seen above, otherwise the cycle takes eight clock cycles to write the long sized data to the memory and finishes at fifteenth state. In the first state named *"write0"* only the FC control multiplexer and RW control multiplexer get the state value of *"load"* which loads the proper function codes to the fc output ports and assert the *"RW"* signal. In the second state named write1 the databus control multiplexer gets the value *"load_ear"* and the address bus control multiplexer gets the value *"load"*, which loads the contents of the Effective Address Register into the databus by opening the gates and then copies that databus content to the address bus by opening the gates between the bidirectional busses. The reader can have a little suspicion at this point about the concurrency issues, but the VHDL solves this type of coding where in one clock pulse a register value or bus value is used on both right and left sides of equation by entering virtual sigma delays that have no time value, between the equations. By doing this the code can assign the right values to the registers or signals at the left hand side of the equations without affecting the concurrency. In our case the VHDL compiler uses sigma delays to solve the concurrency problem and copies the contents of the Effective Address Register to the databus firstly and copies the altered contents of the databus to the address bus secondly, which is exactly what we want to accomplish and what is happening through the busses in real life CPU's. In the third state the *"AS"* signal is asserted by the *"load"* command and the *"RW"* is negated by the *"reset"* command. In the fourth state the dbus port gets the value of the inner databus which has the contents of the destination data, according to the size of the operation by the usage of the if clause. In the fifth state the CPU waits for a DTACK, BERR or VPA signal from the slave device and acts according to the the incoming signal. If none of them is asserted during the state the state jumps to a two clock wait state and after that returns to the same state. The sixth state and the seventh state goes without any operation and at the eighth state the processor checks the size and if the operand size is long goes on with the write cycle, otherwise it jumps back to the state which has called the write cycle.

4.2.2. Read Cycle Created by The Core Generator

The read cycle is similar to the write cycle, and it is shown below with a simulation of it with the generated CPU. Here a decoding of the state table will be accomplished like the previous section.The read cycle for byte and word takes four clock cycles but it finishes at sixth read state because the saved state can be an instruction which takes zero clock cycles to operate and uses the last state of the read operation for execution. If the operand size is long the cycle takes eight clock cycles to read the long sized data from the memory and but in the same manner finishes at fourteenth state. The database entry for the read cycle is shown below in Table 4.11 with the simulation in Figure 4.3 [14, 15].

Table 4.11. Read cycle database entry of the core generator

STATE	case	if	signa	up	dov	valu	ipl_r	be	addr	databus	source	temp_l	st	return_sta	fetch
read0															read1
read1									load	load_ear					read2
read2															read4
read4		if	dtack	0	0	"0"		reset							read5
	else	if	berr	0	0	"0"		load							read5
	else	if	vpa	0	0	"1"							push	read4	**peripheral**
	else												push	read4	wwaitt
read5							load			load_ipl					read6
read6		if	size	1	0	"10"				load	load_hi	load_hi			read7
	else									load	load	load	pull		**saved**
read7		if	berr	0	0	"0" and be= '1'									**buserror**
	else							reset	reset						
read8															read16
read9									load	load_ear					read10
read10															read12
read12		if	dtack	0	0	"0"		reset							read13
	else	if	berr	0	0	"0"		load							read13
	else	if	vpa	0	0	"1"							push	read12	**peripheral**
	else												push	read12	wwaitt
read13							load			load_ipl					read14
read14										load	load_lo	load_lo	pull		**saved**

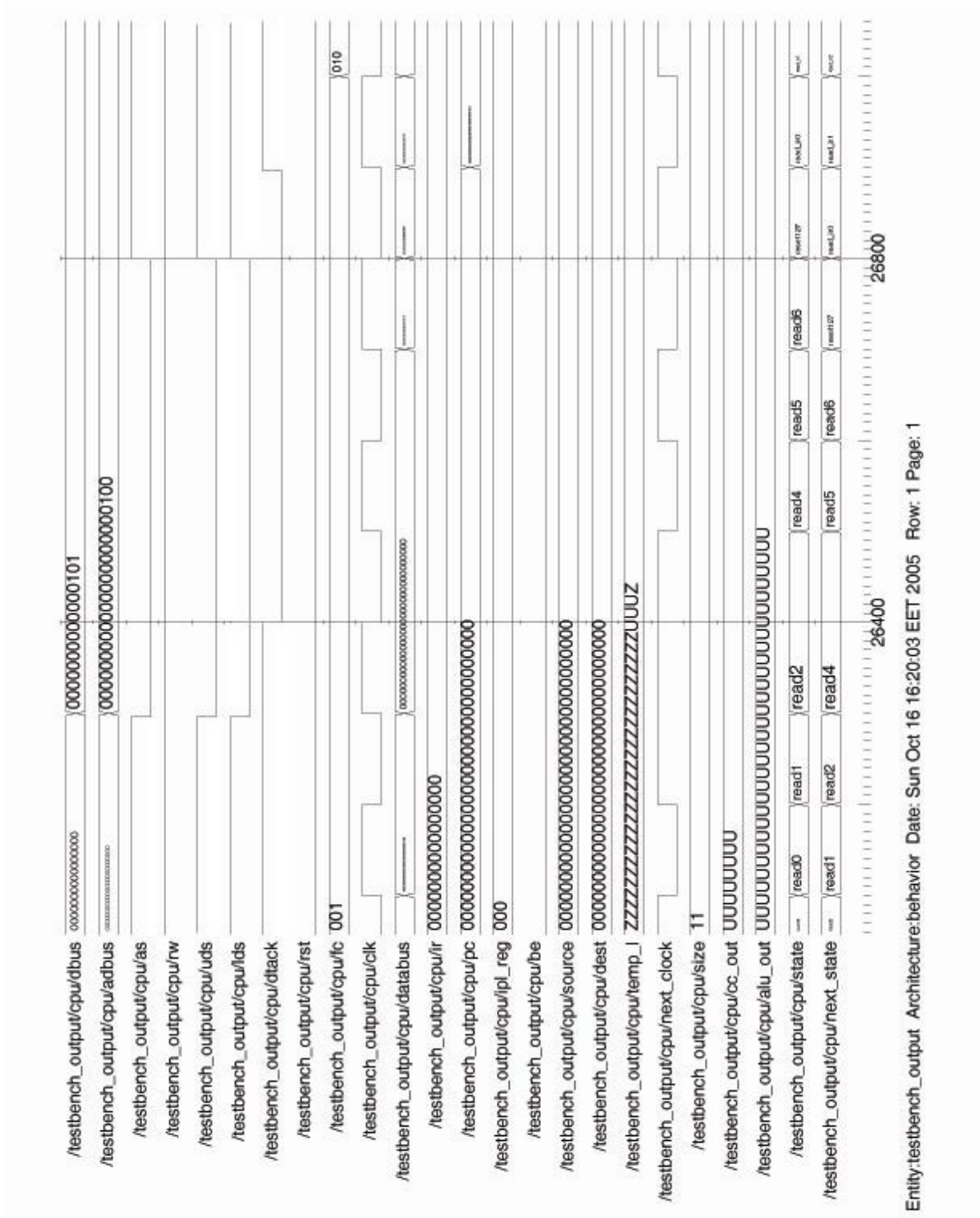

Figure 4.3. Simulation of the read cycle

The read cycle for byte and word takes four clock cycles but it finishes at sixth read state because the saved state can be an instruction which takes zero clock cycles to operate and uses the last state of the read operation for execution. If the operand size is long the cycle takes eight clock cycles to read the long sized data from the memory and but in the same manner finishes at foureenth state.

In the first state named read0 only the FC control multiplexer and RW control multiplexer get the state value of *"load"* which loads the proper function codes to the fc output ports and assert the *"RW"* signal. In the second state named read1 the databus control multiplexer gets the value *"load_ear"* and the address bus control multiplexer gets the value *"load"*, which loads the contents of the Effective Address Register into the databus by opening the gates and then copies that databus content to the address bus by opening the gates between the bidirectional busses. In the third state the *"AS"* signal is asserted by the *"load"* command and the *"UDS"* is loaded according to the read operation and the size by the *"load_r"* command. The CPU jumps directly to the fifth read state as the fourth state there is no alteration of the registers or signals. In the fifth state the CPU waits for a DTACK, BERR or VPA signal from the slave device and acts according to the the incoming signal. If none of them is asserted during the state the state jumps to a two clock wait state and after that returns to the same state. The sixth state goes without any operation and at the seventh state the processor checks the size and if the operand size is long goes on with the read cycle, otherwise it jumps back to the state which has called the read cycle.

4.2.3. Interrupt Acknowledge Cycle Created by The Core Generator

The Interrupt Acknowledge cycle is activated if there is a proper interrupt request by the slave devices which are higher than the inner interrupt acknowledge level saved in the Status Register. The Interrupt Acknowledge Cycle details are shown below:

Table 4.12. Interrupt acknowledge cycle database entry of the core generator

STATE	case	if	signal	up	dov	valu	addr	databus	size	sr	ear	dest	temp_	temp	pc	alu	st	re	fetch
int_check0							reset	load_ipl_reg						load					int_check1
int_check1								load_ppl					load			sub_3bit			int_check2
int_check2		if	sr(0)	0	0	"1"		load_pc	load_long			load					push	int	write_ssp0
	else																		**read_ir0**
int_check3								load_sr	load_word			load		load			push	int	write_ssp0
int_check4								load_ipl_reg		load_ipl									int_check5
int_check5							load	load_ipl_reg		sets_clrt									int_check6
int_check6																			int_check7
int_check7																			int_check8
int_check8		if	dtack	0	0	"0"													int_check9_read
	else																		int_check9_auto
int_check9_read								load_byte					load			shift_l			int_check10_read
int_check10_read								load_alu					load			shift_l			int_check11_read
int_check11_read								load_alu	load_long		load						push	int	read
int_check12_read								load_temp_l							load				**read_ir0**
int_check9_auto								load_ipl_add					load	load_int		add			int_check10_auto
int_check10_auto								load_alu					load			shift_l			int_check11_auto
int_check11_auto								load_alu					load			shift_l			int_check12_auto
int_check12_auto							load	load_alu	load_long		load						push	int	read
int_check12w1_auto								loasd_temp_l							load				read_ir0

The interrupt level check starts in the first state named *"int_check0"* where the address bus is put in high impedance state with the order reset and the databus is loaded with the value of the *"IPL"* port which is then copied to the temporary data register of the Arithmetic Logic Unit. In the second interrupt check level named *"int_check1"* the *"PPL"* value is obtained from the *"SR"* to the databus in the same way and then copied to the other ALU input, and the ALU control multiplexer is issued with the command *"sub_3bit"* which commands the ALU to make a subtraction on the three bit numbers. In the third state, according to the resulting conditional code register the interrupt acknowledge cycle goes on with pushing the PC and SR into the supervisor stack, or jumps back to the first state named read_ir0 where the IR is being read using the standard read operation. If the choice is to push the registers into the stack the *"write_ssp"* cycle handles that operation and then jumps back to the fifth interrupt check state named *"int_check4"*. In this state the IPL value is copied into the PPL bits of the SR to prevent further interrupt occurences, because of the same request coming from the slave device. In the next state the trace bit is cleared and the supervisory bit is set for the *"SR"* by the command *"sets_clrt"* and the address bus has the value of IPL as an address. In the seventh state named *"int_check6"* the *"AS"* and *"UDS"* is

negated by the common multiplexer command *"reset"*. The next state is empty and at the ninth state the interrupt acknowledge cycle checks for the *"DTACK"* input and jumps to the states in which there is an auto vector calculation or vector number fetch. If the vector number is to be read by the device the interrupt acknowledge cycle jumps to the *"int_check9_read"* state where it makes a byte sized operand read operation, and in the next states multiplies the value read by four using the ALU to shift left the result twice. Then in the state *"int_check_11"* the databus is used to copy the result to the Effective Address Register and jumps to a standard read operation by setting the operand size to be long. After the read operation the PC is loaded with the value read from the proper vector and the cycle finishes by jumping to the *"read_ir0"* state where all the operation cycle of the CPU starts over. If the interrupt acknowledge cycle goes on with the auto vector calculation state, instead of reading data from the memory it simply calculates the auto vector number by adding twenty four to the ipl number in the tenth state and making the same operations by multiplying the value by four and reading the vector value to the PC, in the following states. After certain operations when the instruction decoder finds a *"RTE"* operation, the PC and SR values are pulled from the stack and the processor goes on operating with the old PC. In the same manner the addressing mode cycles, bus error cycles or exception cycles can be decoded for a detailed operation but these three main examples are enough to take a glimpse of the state machine structure used in the core generator.

4.2.4. Fetch-Decode-Execute Cycle Created by the Core Generator

The fetch-decode-execute cycle is the most characteristic part of the operation for a CPU. In our model the fetch is a standard read cycle shown in detail in Table 4.11. After the data are read and copied to the databus the Instruction Register gets the contents of the databus by the standard *"load"* operation of the multiplexer. Then the CPU decodes the IR and sets some registers and jumps to the instruction, source effective address calculation or destination effective address calculation according to

the result. In our model the decode state is the longest state written in the state table, as it has a huge `case` structure scanning all the possible IR values. The `case` structure is very easy to build as told in the state table declaration subsection 4.1.4. When the IR is decoded in the `case` structure to produce a proper instruction the decode state checks the addressing mode and the structure of the instruction. If the instruction is a no operand instruction like *"TAS"*,*"LEA"* and etc., the next state value becomes the execution of these instructions. If the instruction is using only one operand like *"CLR"*,*"NEG"* and etc., the decode state checks if the operand is a register or is in memory by checking if the addressing mode is bigger than "001". If the operand is a register the state gets the operand value to the databus and jumps to the instruction execution state. If the operand is in memory the instruction jumps to the source effective address calculation state, by pushing the instruction execution with modify suffix as the return state. This means after the source address is calculated according to the addressing mode in the source effective address calculation state, the instuction will operate as a read-modify-write cycle. If the instruction decoded has two operands, the possibilities of the next state and the return state increases. If both operands are registers the decode state jumps to instruction execution; if one of the operands are registers according to the operand in memory, the decode state jumps to source effective address calculation or destination effective address calculation by pushing the instruction execution state as the return state. If both operands are in memory, the decode state jumps to the source effective address calculation and saves the instruction execution state as the return state. At this point the last state of the source effective address calculation state checks the addressing mode for the destination operand to decide if it will jump to the saved state or destination effective address calculation state, which will be the destination effective address calculation in this case. This is a brief summary of the operations of the decode cycle, now it is time for examining some of the instruction execution state sequences to see how the core generator handles instruction execution.

4.2.4.1. Reset Instruction. The reset operation is the first operation that becomes active when the CPU starts to operate. The reset instruction has 124 empty clocks which are declared by the empty states in the state table declaration of the core generator. After the empty clocks are passed the last five states accomplish the reset operation. The first operational state of the reset cycle, enters the reset value to the all multiplexers controlling register,signal and bus operation. So all the registers will have the values defined in their reset state in the functional unit declaration. At the next state named reset124 the databus loads the value of the first vector and the effective address register copies the contents of the databus by the *"load"* operation of the multiplexer controlling it. The next state is a standard read state which will read the content of the address in the first vector and jump back to the saved state which is reset 125. The reset125 state loads the SSP with the value read from the vector and jumps to the next state. The reset126 state is similar to the reset124 state except the vector number is two for this state. After the data are read from the address in the second vector it is copied to the program counter in the reset127 state ending the reset cycle. The state table and the VHDL simulation produced are shown below in Table 4.13 and Figure 4.4.

Table 4.13. Reset cycle database entry of the core generator

STATE	ipl_reg	be	dbus	addr	databus	sr	ear	source	dest	temp_l	temp	pc	alu	st	return_s	fetch
reset118																reset119
reset119																reset120
reset120																reset121
reset121																reset122
reset122																reset123
reset123	reset	reset	reset	reset	reset	reset	reset	reset	reset	reset	reset	reset	nop			reset124
reset124					load_vect0		load							push	reset125	read0
reset125					load_temp_l											reset126
reset126					load_vect1		load							push	reset127	read0
reset127					load_temp_l							load				read_ir0

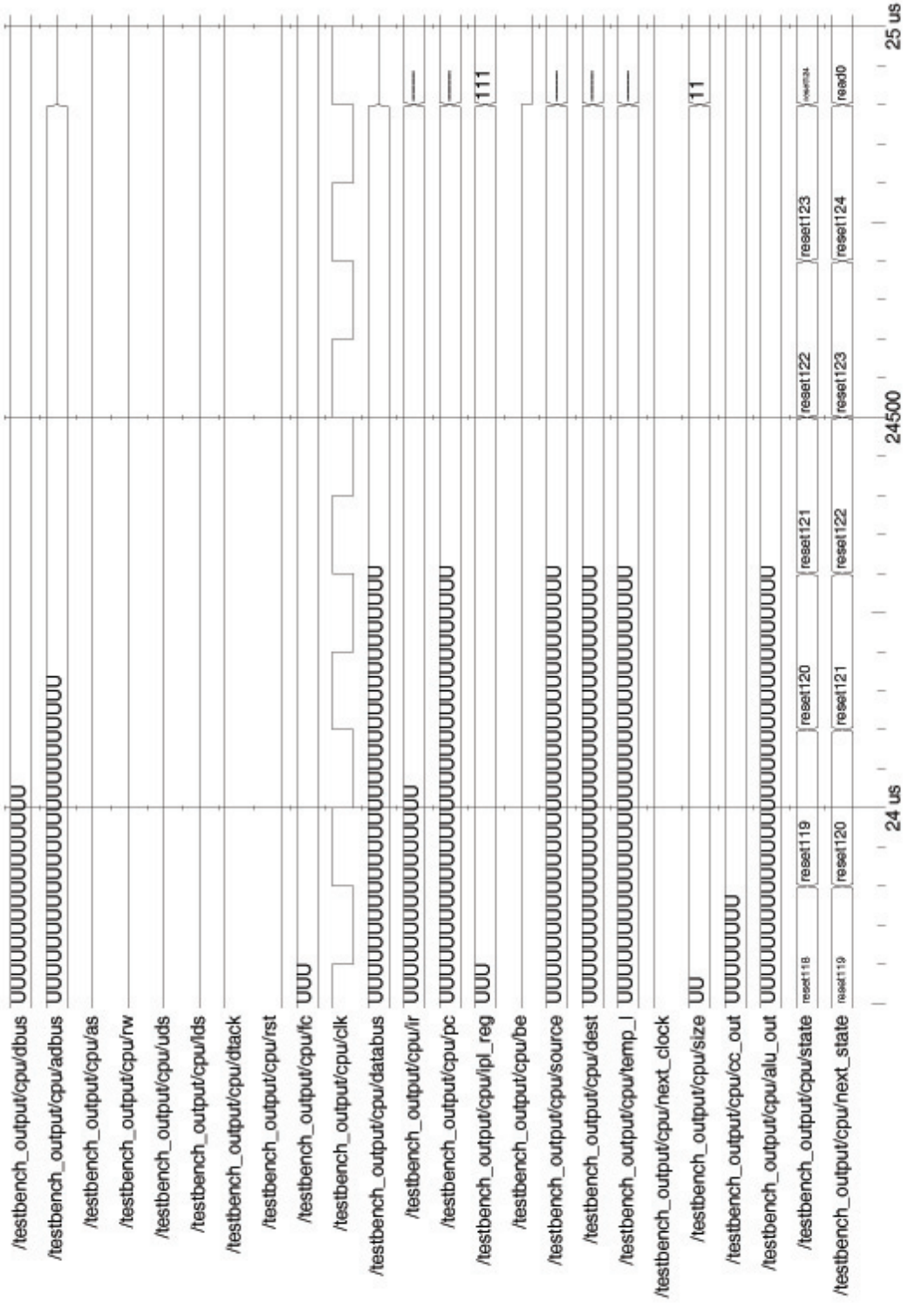

Figure 4.4. Reset cycle simulation

4.2.4.2. MOVE Instruction. The move instruction is a special intruction for the Motorola 68000. The addressing mode resides in different bits than the other instructions and it does not read the destination data as the other instructions do. That is why it is very simple to execute the state if the decode state correctly decoded the instruction without making any failure in the destination mode and the addressing mode. When the move instruction state is operational it already has the source operand coming from the source effective address calculation cycle or from the decode state according to the source to be in memory or to be a register. The move instruction calculates the effective address and starts a write operation if the destination is a memory location, otherwise it only writes the source operand to destination register in one state without spending any clock cycle as shown in the Table 4.14 below.

Table 4.14. Move cycle database entry of the core generator

STATE	databus	reg	source_reg	source	fetch
move_w_sr			load_absshort	load_d	move_w_sr1
move_w_sr1	load_sr	load_source_word			int_check0

4.2.4.3. MULU Instruction. The MULU instruction is one of the most specific instructions of The Motorola 68000 Microprocessor. The MULU instruction uses shift and add algorithm to produce a 32-bit result from two 16-bit operands and it is one of the few instructions in the 68000 instruction set that changes the clock cycles according to the operands. The clock count for this instruction is 38 constant clocks plus two clock cycles for every one in the multiplier causing a maximum of 70 clock cycles for the whole operation. The MULU isntruction makes a standard shift left operation at each cycle for sixteen cycles, which makes 32 clocks, and spends six clocks for reading the operands and writing them to the destination effective address. The other clocks are used for addition operation using the ALU if the multiplier has a one in the least

significant bit after the shift operation. If there is a zero, the CPU does not have to make an addition operation so the CPU can go on with shifting the next bit of the multiplier. The database entry of the MULU instruction is shown below.

Table 4.15. Multiplication cycle database entry of the core generator

STATE	case	if	sign	up	dow	value	be	addr	databus	source	dest	temp_l	temp	pc	alu	fetch
mulu_w_source7		if	berr	0	0	"0" and be= '1'										buserror
	else						reset	reset	load_dest				load		**nop**	**mulu0**
mulu0									load_dest			load			signex_one	mulu1
mulu1									load_alu		load		reset			mulu2
mulu2	case	dest(0)		0	0	"1"			load_source			load				mulu3
						others			load_dest			load			shift_r	mulu6
mulu3															add	mulu4
mulu4									load_alu				load			mulu5
mulu5																mulu6
mulu6									load_dest			load			shift_r	mulu7
mulu7									load_alu		load					mulu8
mulu8									load_source			load			shift_l	mulu9
mulu9									load_alu	load						mulu10
mulu10	case	dest		31	16	"0000000000000000"										mulu_write_dest
						others										mulu2
mulu_write_dest									load_temp							mulu_wait0
mulu_wait0																mulu_wait1
mulu_wait1																mulu_wait2
mulu_wait2																mulu_wait3
mulu_wait3																mulu_wait4
mulu_wait4																mulu_wait5
mulu_wait5																int_check0

In the first state called mulu0 the destination register value is copied into the databus which is then copied to the temporary data register by the standard *"load"* command. After the data are fed to the temporary data register the Arithmetic and Logic Unit is issued the command *"signex_one"* to sign extend the 16-bit operand to 32-bit operands entering ones to the upper sixteen bits. This ones will be used to check if the register has reached the sixteenth cycle when there will be no ones at the most significant bit. In the second state named *"mulu1"* the databus loads the ALU result and loads the destination operand with the new sign extended value. The third state is the loop state where the least significant bit is checked and if there is a one the next state is the *"mulu3"* where the addition of the operands is accomplished, otherwise the next state is *"mulu6"*. In the *"mulu4"* the added result is saved into the temporary

data register by loading the databus with the ALU output and copying the data to the temporary data register. The state then goes to *"mulu6"* where the standard shifting operations are accomplished. The operands are shifted left and right in the next states and in the state *"mulu10"* the multiplier is checked if it is shifted right for sixteen times and if the result is true the instruction writes the addition result to the destination effective address, otherwise the instruction jumps back to *"mulu2"* state where the least significant bit of the multiplier is checked again. The produced instruction works exactly as expected, taking seventy clock cycles at maximum.

4.3. Core Generator Implementation for Motorola 6809 Microprocessor

The main purpose of the core generator is to create synthesizable VHDL CPU cores easily by filling the core generator database properly. The functionality of the core generator depends on its ability to create other CPU cores than its design purpose Motorola 68000. To test the functionality the database structure is used to create a Motorola 6809 microprocessor using the core generator. The 6809 Microprocessor is a 8-bit microprocessor which makes the design completely different than the 68000 Microprocessor but also the design is not that difficult since it is the member of the same family of microprocessors.

The database fields are filled accordingly using the old entries and the design finished earlier than expected proving the effectiveness of the core generator. The synthesized model of the 6809 can be seen below in Figure 4.5.

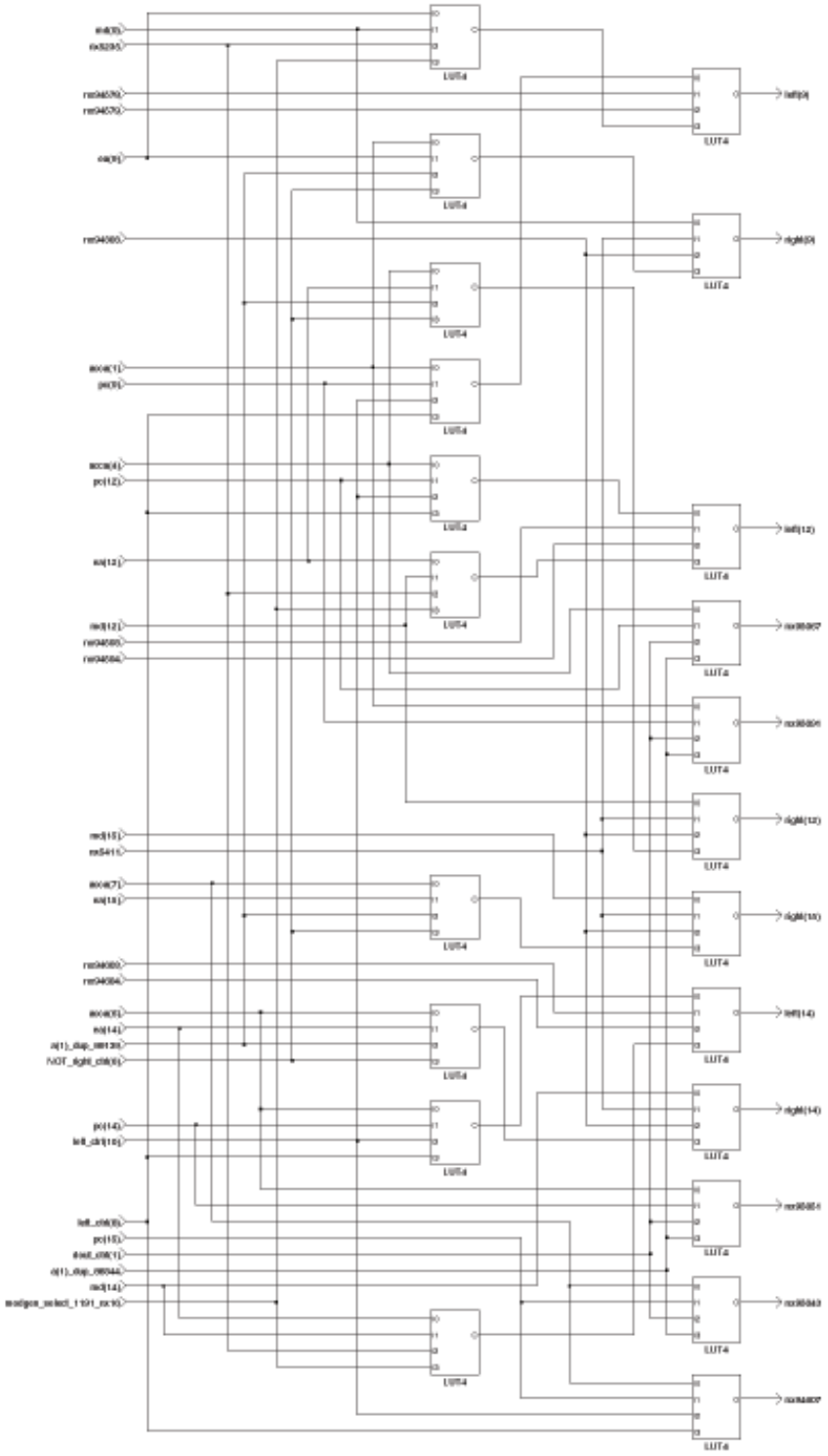

Figure 4.5. Motorola 6809 CPU synthesis result

5. TESTING AND CONCLUSIONS

This chapter details what has actually been accomplished during the project by Exploring some test results and stepping through a demonstration of a 68000 assembly program. Suggestions for further avenues of work on this project are also explored.

5.1. Testing and Development Progress

All design testing has been performed using ModelSim. A simulation on this platform allows the designer to isolate any wire, register or signal contained within the design. This very powerful tool was a vital component in debugging the operation of the VHDL CPU.

Each individual program part has been tested using test benches designed to verify the functionality on all possible inputs, or when that is not feasible, a set of inputs that practically verifies module operation. Top level testing was accomplished through strategic implementation of micro code. As micro coding the system for the full instruction set is a time consuming task, a focus has been placed on using selected micro coded applications to test the fully inter connected system. As a result the micro code in place to date will verify the inter operation of all the modules. For example, MULU will test branching and sequencing, MOVE the addressing mode capability and the bus interface and RESET the handling of exceptions. Most of the problems encountered throughout this project were results of implementation mistakes. For example, an early implementation of the control path attempted to specify the unit in such a way that the ModelSim tool would automatically infer control logic. This proved to be an error-prone and unreliable way of implementing control as the design was unnecessarily complex and hard to follow. It was at this early stage that the decision to keep to an RTL level of abstraction was made.

Other problems stemmed from common mistakes such as incorrect behavioural implementation results. In an example the ALU was not generating proper results and condition codes for the subtract function. The ModelSim platform enabled examination of each part of the design to isolate this problem to a programming mistake. Also the asynchronous communication, functional code generator and core generator provided problems. As previously outlined, the project has been developed to a stage where all necessary hardware needed for a VHDL op code compatible CPU has been put in place. All modules have been tested and confirmed operational under behavioral and post synthesis simulation. In addition a substantial subset of the micro-coding necessary for full instruction set decoding has been completed and tested under simulation.

5.2. A 68000 demonstration

For demonstration purposes a 68000 project style environment has been modelled in VHDL [16, 17, 18, 19]. A 68000 has been instantiated within this model and a ROM and RAM compatible with the 68000 bus interface have both been memory mapped to the specified locations through proper generation of RTL logic multiplexer enable signals as would be done by a programmer in the Architecture project outlined in the introduction. The ROM would be loaded with an initial stack pointer, an initial PC and a demonstration program. The synthesis results of the 68000 VHDL CPU and the 6809 VHDL CPU are shown below. The results are very satisfactory, with clock frequencies larger than 30 MHz. The gate instances are nearly same for the two processors because the 6809 CPU is generated by the core generator in a quick programming style using most of the remaining design tables of the 68000 without any optimization. The area report of the 68000 CPU is shown below:

⋮

****

Cell: cpu68000 View: CPU_CORE Library: work

****

Cell Library References Total Area

BUFGP xcv2 1 x 1 1 BUFGP
FDE_1 xcv2 463 x 1 463 Dffs or Latches
FD_1 xcv2 40 x 1 40 Dffs or Latches
GND xcv2 1 x 1 1 GND
IBUF xcv2 17 x 1 17 IBUF
LD xcv2 102 x 1 102 Dffs or Latches
LUT1 xcv2 2 x 1 2 Function Generators
LUT2 xcv2 165 x 1 165 Function Generators
LUT2_L xcv2 480 x 1 480 Function Generators
LUT3 xcv2 199 x 1 199 Function Generators
LUT4 xcv2 1196 x 1 1196 Function Generators
MUXCY_L xcv2 465 x 1 465 MUX CARRYs
MUXF5 xcv2 19 x 1 19 MUXF5
OBUF xcv2 43 x 1 43 OBUF
OBUFT xcv2 8 x 1 8 OBUFT
VCC xcv2 1 x 1 1 VCC
XORCY xcv2 480 x 1 480 XORCY

Number of ports : 77
Number of nets : 3700
Number of instances : 3682

Number of references to this view : 0

Total accumulated area :

Number of BUFGP : 1

Number of Dffs or Latches : 605

Number of Function Generators : 2042

Number of GND : 1

Number of IBUF : 17

Number of MUX CARRYs : 465

Number of MUXF5 : 19

Number of OBUF : 43

Number of OBUFT : 8

Number of VCC : 1

Number of XORCY : 480

Number of gates : 2037

Number of accumulated instances : 3682

****

Device Utilization for 2V250fg256

****

Resource Used Avail Utilization

IOs 77 172 44.77

Function Generators 2042 3072 66.47

CLB Slices 1021 1536 66.47

Dffs or Latches 605 3588 16.86

Block RAMs 0 24 0.00

Block Multipliers 0 24 0.00

⋮

The timing delay report of 68000 is given below in detail.

⋮

Clock Frequency Report

Clock : Frequency

---

clk : 37.6 MHz

Critical Path Report

Critical path 1, (path slack = 73.4):

NAME GATE ARRIVAL LOAD

---

clock information not specified

delay thru clock network 0.54 (worst case)

reg_state(29)/Q FDE_1 0.00 1.67 up 0.50

nx95437/I1 LUT4 0.00 1.67 up 0.40

nx95437/O LUT4 0.89 2.56 up 0.40

a(3)/I3 LUT4 0.00 2.56 up 0.10

a(3)/O LUT4 0.47 3.03 up 0.10

nx94484/I3 LUT4 0.00 3.03 up 0.40

nx94484/O LUT4 0.89 3.92 up 0.40

nx95436/I3 LUT4 0.00 3.92 up 0.40

nx95436/O LUT4 0.89 4.81 up 0.40

nx94483/I3 LUT4 0.00 4.81 up 0.40

nx94483/O LUT4 0.89 5.70 up 0.40

nx36047/I0 LUT4 0.00 5.70 up 0.30

nx36047/O LUT4 0.75 6.45 up 0.30

nx94517/I0 LUT4 0.00 6.45 up 0.40

nx94517/O LUT4 0.89 7.34 up 0.40

nx94514/I1 LUT4 0.00 7.34 up 0.40

nx94514/O LUT4 0.89 8.23 up 0.40

nx94511/I2 LUT4 0.00 8.23 up 0.40

nx94511/O LUT4 0.89 9.12 up 0.40

right_ctrl(2)/I1 LUT4 0.00 9.12 up 0.20

right_ctrl(2)/O LUT4 0.61 9.73 up 0.20

nx94530/I1 LUT4 0.00 9.73 up 0.40

nx94530/O LUT4 0.89 10.62 up 0.40

nx94529/I2 LUT4 0.00 10.62 up 0.40

nx94529/O LUT4 0.89 11.50 up 0.40

nx95459/I2 LUT4 0.00 11.50 up 0.40

nx95459/O LUT4 0.89 12.39 up 0.40

nx5411/I3 LUT4 0.00 12.39 up 0.20

nx5411/O LUT4 0.61 13.00 up 0.20

a(1)_dup_81026/I2 LUT3 0.00 13.00 up 0.30

a(1)_dup_81026/O LUT3 0.75 13.75 up 0.30

a(0)_dup_81045/I2 LUT3 0.00 13.75 up 0.20

a(0)_dup_81045/O LUT3 0.61 14.37 up 0.20

nx94533/I3 LUT4 0.00 14.37 up 0.40

nx94533/O LUT4 0.89 15.25 up 0.40

right(0)/I2 LUT3 0.00 15.25 up 0.30

right(0)/O LUT3 0.75 16.00 up 0.30

modgen_add_223_ix45/I1 LUT2_L 0.00 16.00 up 0.00

modgen_add_223_ix45/LO LUT2_L 0.33 16.34 up 0.20

modgen_add_223_ix49/LO MUXCY_L 0.40 16.74 up 0.20

modgen_add_223_ix55/CI MUXCY_L 0.00 16.74 up 0.00

modgen_add_223_ix55/LO MUXCY_L 0.04 16.78 up 0.20

modgen_add_223_ix61/CI MUXCY_L 0.00 16.78 up 0.00

modgen_add_223_ix61/LO MUXCY_L 0.04 16.82 up 0.20

modgen_add_223_ix67/CI MUXCY_L 0.00 16.82 up 0.00

modgen_add_223_ix67/LO MUXCY_L 0.04 16.86 up 0.20

modgen_add_223_ix73/CI MUXCY_L 0.00 16.86 up 0.00

modgen_add_223_ix73/LO MUXCY_L 0.04 16.90 up 0.20

modgen_add_223_ix79/CI MUXCY_L 0.00 16.90 up 0.00

modgen_add_223_ix79/LO MUXCY_L 0.04 16.94 up 0.20

modgen_add_223_ix85/CI MUXCY_L 0.00 16.94 up 0.00

modgen_add_223_ix85/LO MUXCY_L 0.04 16.98 up 0.20

modgen_add_223_ix91/CI MUXCY_L 0.00 16.98 up 0.00

modgen_add_223_ix91/LO MUXCY_L 0.04 17.02 up 0.20

modgen_add_223_ix97/CI MUXCY_L 0.00 17.02 up 0.00

modgen_add_223_ix97/LO MUXCY_L 0.04 17.06 up 0.20

modgen_add_223_ix103/CI MUXCY_L 0.00 17.06 up 0.00

modgen_add_223_ix103/LO MUXCY_L 0.04 17.10 up 0.20

modgen_add_223_ix109/CI MUXCY_L 0.00 17.10 up 0.00

modgen_add_223_ix109/LO MUXCY_L 0.04 17.14 up 0.20

modgen_add_223_ix115/CI MUXCY_L 0.00 17.14 up 0.00

modgen_add_223_ix115/LO MUXCY_L 0.04 17.18 up 0.20

modgen_add_223_ix121/CI MUXCY_L 0.00 17.18 up 0.00

modgen_add_223_ix121/LO MUXCY_L 0.04 17.22 up 0.20

modgen_add_223_ix127/CI MUXCY_L 0.00 17.22 up 0.00

modgen_add_223_ix127/LO MUXCY_L 0.04 17.26 up 0.20

modgen_add_223_ix131/CI XORCY 0.00 17.26 up 0.00

modgen_add_223_ix131/O XORCY 1.52 18.78 up 0.40

nx94667/I1 LUT4 0.00 18.78 up 0.40

nx94667/O LUT4 0.89 19.67 up 0.40

nx95393/I2 LUT4 0.00 19.67 up 0.40

nx95393/O LUT4 0.89 20.56 up 0.40

nx95571/I3 LUT4 0.00 20.56 up 0.40

nx95571/O LUT4 0.89 21.45 up 0.40

out_alu(14)/I2 LUT3 0.00 21.45 up 0.20

out_alu(14)/O LUT3 0.61 22.06 up 0.20

nx94851/I0 LUT4 0.00 22.06 up 0.40

nx94851/O LUT4 0.89 22.95 up 0.40

nx94850/I2 LUT4 0.00 22.95 up 0.20

nx94850/O LUT4 0.61 23.56 up 0.20

nx95608/I1 LUT4 0.00 23.56 up 0.40

nx95608/O LUT4 0.89 24.45 up 0.40

ix95642/I0 MUXF5 0.00 24.45 up 0.00

ix95642/O MUXF5 0.95 25.39 up 0.40

nx3902/I0 LUT2 0.00 25.39 up 0.40

nx3902/O LUT2 0.89 26.28 up 0.40

reg_cc(2)/D FDE_1 0.00 26.28 up 0.00

data arrival time 26.28

data required time (default specified - setup time) 99.71

---

data required time 99.71

data arrival time 26.28

---

slack 73.43

⋮

The 6809 synthesis report for the area used are shown below:

⋮

****

Cell: cpu09 View: CPU_ARCH Library: work

****

Cell Library References Total Area

BUFGP xcv2 1 x 1 1 BUFGP

FDE_1 xcv2 718 x 1 718 Dffs or Latches

GND xcv2 1 x 1 1 GND

IBUF xcv2 14 x 1 14 IBUF

LUT1 xcv2 8 x 1 8 Function Generators

LUT1_L xcv2 35 x 1 35 Function Generators

LUT2 xcv2 440 x 1 440 Function Generators

LUT2_L xcv2 56 x 1 56 Function Generators

LUT3 xcv2 351 x 1 351 Function Generators

LUT3_L xcv2 4 x 1 4 Function Generators

LUT4 xcv2 1646 x 1 1646 Function Generators

MUXCY_L xcv2 89 x 1 89 MUX CARRYs

MUXF5 xcv2 35 x 1 35 MUXF5

OBUF xcv2 26 x 1 26 OBUF

VCC xcv2 1 x 1 1 VCC

XORCY xcv2 95 x 1 95 XORCY

Number of ports : 41

Number of nets : 3535

Number of instances : 3520

Number of references to this view : 0

Total accumulated area :

Number of BUFGP : 1

Number of Dffs or Latches : 718

Number of Function Generators : 2540

Number of GND : 1

Number of IBUF : 14

Number of MUX CARRYs : 89

Number of MUXF5 : 35

Number of OBUF : 26

Number of VCC : 1

Number of XORCY : 95

Number of gates : 2530

Number of accumulated instances : 3520

****

Device Utilization for 2V250fg256

****

Resource Used Avail Utilization

IOs 41 172 23.84

Function Generators 2540 3072 82.68

CLB Slices 1270 1536 82.68

Dffs or Latches 718 3588 20.01

Block RAMs 0 24 0.00

Block Multipliers 0 24 0.00

⋮

The clock frequency and the delay report of the CPU 6809 is shown below:

⋮

Clock Frequency Report

Clock : Frequency

---

clk : 100.8 MHz

NOT_nx24852 : 100.8 MHz

NOT_nx27336 : 123.3 MHz

NOT_nx24932 : 122.8 MHz

NOT_nx25012 : 122.8 MHz

NOT_nx29728 : 134.2 MHz

NOT_nx29824 : 134.2 MHz

NOT_nx29934 : 134.2 MHz

Critical Path Report

Critical path 1, (path slack = 90.1):

NAME GATE ARRIVAL LOAD

---

clock information not specified

delay thru clock network 0.00 (ideal)

reg_state(32)/Q FD_1 0.00 1.55 up 0.80

state(32) (net) 0.00 1.55 up (fan) 41.00

alu_ctrl(4)/I0 LUT2 0.00 1.55 up 0.80

alu_ctrl(4)/O LUT2 1.45 2.99 up 0.80

alu_ctrl(4) (net) 0.00 2.99 up (fan) 31.00

nx52587/I1 LUT4 0.00 2.99 up 0.40

nx52587/O LUT4 0.89 3.88 up 0.40

nx52587 (net) 0.00 3.88 up (fan) 1.00

nx53220/I2 LUT4 0.00 3.88 up 0.40

nx53220/O LUT4 0.89 4.77 up 0.40

nx53220 (net) 0.00 4.77 up (fan) 1.00

nx52523/I3 LUT4 0.00 4.77 up 0.80

nx52523/O LUT4 1.45 6.22 up 0.80

nx52523 (net) 0.00 6.22 up (fan) 32.00

nx52593/I3 LUT4 0.00 6.22 up 0.40

nx52593/O LUT4 0.89 7.11 up 0.40

nx52593 (net) 0.00 7.11 up (fan) 1.00

alu_out(29)/I2 LUT4 0.00 7.11 up 0.30

alu_out(29)/O LUT4 0.75 7.86 up 0.30

alu_out(29) (net) 0.00 7.86 up (fan) 3.00

nx52594/I1 LUT4 0.00 7.86 up 0.40

nx52594/O LUT4 0.89 8.75 up 0.40

nx52594 (net) 0.00 8.75 up (fan) 1.00

modgen_select_68_nx14/I2 LUT4 0.00 8.75 up 0.40

modgen_select_68_nx14/O LUT4 0.89 9.64 up 0.40

modgen_select_68_nx14 (net) 0.00 9.64 up (fan) 1.00

reg_ear(29)/D FDE_1 0.00 9.64 up 0.00

data arrival time 9.64

data required time (default specified - setup time) 99.71

---

data required time 99.71

data arrival time 9.64

---

slack 90.08

⋮

The resulting clock frequencies and the gate numbers prove the core generator to be an effective and useful tool. With small improvements the core generator can be widely used for design and test of CPU's built in VHDL.

5.3. Conclusions

The first motivation behind this project was to generate a Motorola 68000 CPU using VHDL. A requirement of the design was to be capable of download to the FPGA project board. The design has been demonstrated as synthesizable and downloadable to the FPGA. As required a compatible bus interface has been successfully implemented in VHDL. The 68000 CPU model, as demonstrated, is capable of executing code as generated by the 68k cross assembler. While only a subset of the instruction set is implemented the instructions that have been demonstrate there is a solid foundation on which to add the remainder.

It is noted here that a rearrangement of hardware in the 68000 design, perhaps omitting hardware duplicated for ease of operation e.g. the Displacement Register or the Vector Decoder could reduce the resources used by the 68000. Also improving the functional unit by the usage of structurally produced full adders and shifters, can decrease the number of gates used. Both of these operations would remove replicated hardware across the CPU model and decrease the resources taken up by the CPU. Further reduction in the use of the most valuable resource, FPGA slices, may be accomplished by taking better advantage of the FPGA resources.

The second motivation behind this design was to create a readable, well documented design model. At that point a core generator idea has occurred and implement using Visual Basic and Excel datasheets. Sufficient documentation has been produced by the core generator for a user to understand and expand on the current design. The core generator worked in a very efficient way to produce correct synthesizable codes out of database entries, and that's why became the main attention in the whole project. The core generator is not fully automatic but it save an enourmous time against manual code writin. To conclude, the main objectives of the project a functional, readable VHDL 68000-opcode compatible CISC implementation have been achieved to an extent

allowable given the time constraints of the project, and in addition a core generator for future designs is provided.

5.4. Skills Acquired and Lessons Learned During the Design

The main skills acquired in implementing this project are proficiency in VHDL programming, knowledge of the HDL design flow. Also a more detailed knowledge of processor design has been acquired together with a more in-depth insight into how a processor, such as the 68000, is implemented in real terms. At last as a result of the concentrated long work, the knowledge of designing an advanced core generator, which was the most trivial but resourceful part of the project, is acquired. There have been several lessons taken from the completion of this project. At first, I have learned that time management skills are important when balancing a large project with course-based study. Finally, I have learned when building a substantial project, using an incremental functional design technique is key to generating an all round functional design.

5.5. Further Work

Opportunities exist for further work both in the completion of the current design and the expansion of the current design. The first goal has to be the completion of micro coding. Some suggestions for such a project include:

- Redesign of the instruction decoder to include generation of comments, Redesign of the peripherals, and asyncronhous communication of the core.
- Efficient implementation of all complex instructions such as TAS,LEA and etc.
- Downloading of the current VHDL unit to the project board.
- Implement the supervisor/user programming model as defined by Motorola.
- Examine design efficiency and how it may be improved.

- Improve the core generator to a point where the core generator works fully autoamatic and creates the functional unit, given the necessary details.

APPENDIX A: CORE GENERATOR USER MANUAL

A brief user manual of the core generator will be described here as the basic operaion of the core generator is detailed in the previous sections. The core generator modifies the Microsoft Excel database created by the user with the functional unit declaration of the CPU to be created in ascii text format, and combines these databases in a specific manner to create the VHDL core. The user must fill the Microsoft Excel database named *"template.xls"*, according to the properties of the CPU to be designed. The worksheets must be filled in order from the first to the last one because the core generator uses the data entered in the first few sheets to create the validation of the data in the next worksheets and for creating the row and column names. The user can use the templates and styles in the Microsoft Excel datasheets created for the Motorola 68000 CPU in the file *"68000.xls"*. After all the worksheets are filled accordingly, the user must execute the file named *"project.exe"*. The faceplate of the program waits for the user to browse for the file he created, by pressing the ">>" button or file menu and find the template file he or she had filled. By pressing the *"RUN"* button the core generator starts to create the CPU file and shows the progress in the faceplate. When the program finishes the user can rename the *"output.txt"* created by the core generator in the folder where the program is executed, with the format *"filename.vhd"* and use this file to simulate and synthesize the result. The faceplate of the executable is shown below in Figure A.1.

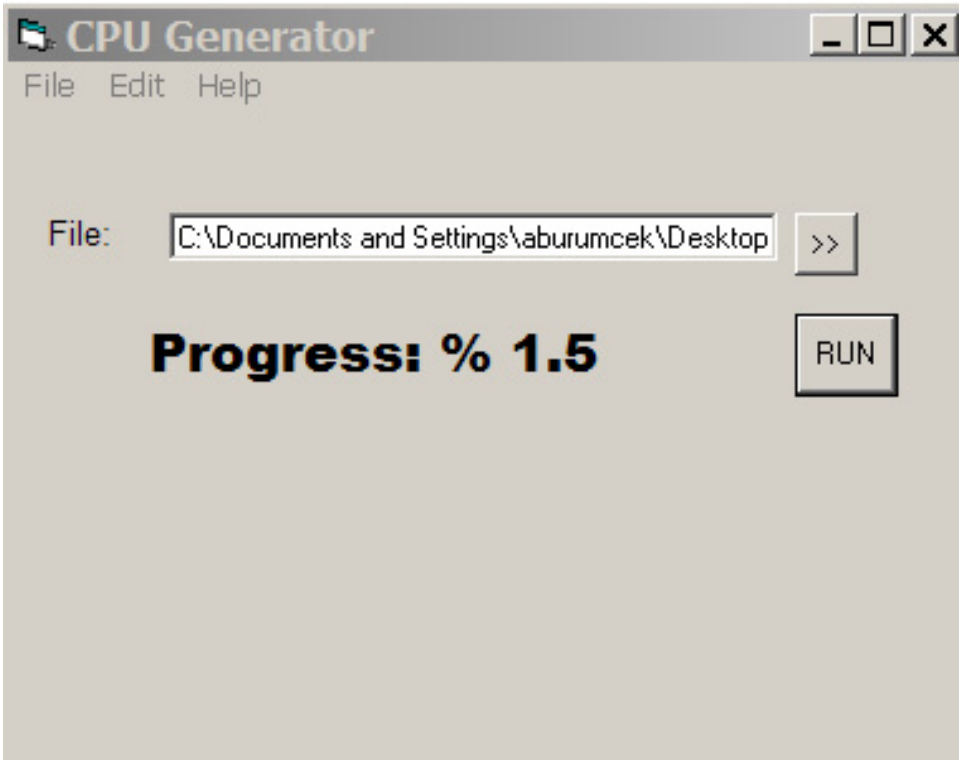

Figure A.1. The core generator program faceplate

REFERENCES

1. Navabi, Z., ": Analysis and Modeling of Digital Systems", *McGraw-Hill*, New York, December 1997.

2. Glaurt, W., Online VHDL Tutorial, http://www.vhdl-online.de.

3. "IEEE Standard VHDL Reference Manual", IEEE Std. 1076-1987, 1987.

4. Wei-Cheng L., Krstic A. , Cheng K. T., "Functionally Testable Path Delay Faults on a Microprocessor", *IEEE Design & Test of Computers*, October-December 2000.

5. Wakerly, J., "Microcomputer Architecture and Programming The 68000 Family", *John WileySons Inc.*, 1989.

6. Coffron, J. W., "Using and Troubleshooting the MC68000", *Boston Publishing Company Inc.*, 1983.

7. "Motorola Inc. Motorola M68000 Family Programmers Reference Manual", Available Online, http://www.motorola.com.

8. "Motorola Inc. Motorola M68000 Users Manual", ninth ed. , *Prentice Hall*, PTR, ISBN: 0-13-566695-3 / 0135666953 .

9. Bodur M., "Bilgisayar Organizasyonu ve RISC Donanımına Giriş", Bileşim Yayınları 97.

10. Lin J. J., "Fully Synthesizable Microprocessor Core via HDL Porting", *Hewlett Packard Journal*, Article 14, August 1997.

11. Gray J., , "Building a RISC System in an FPGA: Part 1: Tools, Instruction Set, and Datapath; Part 2: Pipeline and Control Unit Design; Part 3: System-on-a-Chip Design", *Circuit Cellar Magazine*, 116-118, March-May 2000.

12. Mentor Graphics Web site, Available online, www.mentorg.com/products/seamless/ .

13. OpenCores open source group, Online http://www.opencores.org.

14. Leef, S. and Klieg, R. ,"Hardware, Software Join in Simulation", *EE Times*, June 3, 1996.

15. Model Sim Inc., "Modelsim Xilinx User's Manual.", Available from: http://www.model.com.

16. Tinoosh M., "Design and Evaluation of FPGA based Gigabit Ethernet/PCI Network Interface Card", Rice University, Texas Houston, March 2004.

17. Xilinx Inc, "FPGA Xpower tutorial". Available from: http://toolbox.xilinx.com/docsan/xilinx5/help/xpower/xpower.htm.

18. Xilinx Inc, "PowerPC Embedded Processor Solution" available from: http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=v2p_powerpc.

19. Xilinx Inc, Xilinx foundation tool product guide, Available online, http://www.xilinx.com.

REFERENCES NOT CITED

Fpga-cpu open source group, Online, http://www.fpgacpu.org.

"The SPARC V8 Architecture Manual", v80 ed. Available online, http: //www.sparc.com /standards/V8.pdf.

0-In Design Automation, "Methodology Overview", Available online, http://www.0-in.com.

Satoru, N., S. Katsuyuki, S. Naoki, and K. Norio, "Fieldbus Communication Controller Chips Developed Using Our 'CEEDS-ASIC' System" , *Yokogawa Technical Report English Version*, No. 19, 1994.

Field Programmable Port Extender Homepage, Available from: "http: //www.arl.wustl.edu /projects/fpx", Aug.2000.

Chen L. and Sujit D., "Software-Based Diagnosis for Processors", Dept. ECE, University of California at San Diego, La Jolla, CA 92093-0407, Available from: http: //esdat.ucsd.edu/ lichen/dac2002.

DeHon A., "DPGA-Coupled 26 Microprocessors: Commodity ICs for the Early 21st Centuary", *IEEE Workshop on FPGAs for Custom Computing Machines. D.A. Beull and K. L. Pocek (eds.)*, IEEE Computer Society Press, 1994.