DESIGN AND IMPLEMENTATION OF PORTABLE AND CONFIGURABLE RISC PROCESSOR ARCHITECTURE

by

Volodymyr Sergeyev, B.Eng,
Odessa Polytechnic University, 1987

A project
presented to Ryerson University
in partial fulfillment of the
requirement for the degree of
Master of Engineering
in the Program of
Electrical and Computer Engineering

Toronto, Ontario, Canada, 2010

Author's Declaration

I hereby declare that I am the sole author of this project.

I authorize Ryerson University to lend this project to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this project by photocopying or by other means, in total or in part at the request of other institutions or individuals for the purpose of scholarly research.

Instructions on Borrowers

Ryerson University requires the signatures of all persons using or photocopying this project. Please sign below, and give address and date.

Name	Signature	Address	Date

DESIGN AND IMPLEMENTATION OF PORTABLE AND CONFIGURABLE RISC PROCESSOR ARCHITECTURE

Volodymyr Sergeyev
Master of Engineering
Department of Electrical and Computer Engineering
Ryerson University, Toronto, 2010

Abstract

This project presents the configurable microprocessor design based on the MIPS architecture. The level of configurability includes a choice of the pipelined or unpipelined architecture, number of pipeline stages, data path bit-width, instruction subsetting, program and data memory size. The microprocessor design flow is supported by the set of standard and custom software tools. The wide spectrum of the microprocessor configurations provides an opportunity to optimize hardware for the specific application. The HDL design of the microprocessor is independent of the hardware platform. The portability of the design was verified on the competitive FPGA platforms and ASIC. The selected microprocessor configuration running the test application was successfully implemented and verified on the FPGA development board. The obtained implementation results were compared to the existing commercial and research microprocessors and critical advantages of the presented design were outlined.

Acknowledgments

It is a pleasure to thank those who made this project possible. I want to convey my deepest gratitude to my supervisor Dr. Adnan Kabbani whose assistance, encouragement and expertise significantly contributed to this work. Through the duration of project-writing period, he provided inspiration, guidance, and good advice. Work under his supervision gave me an invaluable experience.

I would like to thank the many people who have taught me during my graduate studies. I am especially grateful to professors Reza Sedaghat, Fei Yuan, Gul Khan, Vadim Geurkov, and Lev Kirischian. Their devotion, professional attitude, and teaching skills had a remarkable influence on the completion of my studies.

I wish to thank the Department of Electrical and Computer Engineering for providing research resources and technical facilities required for completion of this project. My special thanks to Jason Naughton whose knowledge and technical expertise helped me efficiently use the department resources.

Most of all I owe my thanks to my family for their patience, support, and understanding during my graduate study. Especially, I am grateful to my wife Iryna whose encouragement and endless love helped me finish this work.

Contents

List of Tables

List of Figures

List of Abbreviations

ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processor
ASP	Application Specific Processor
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GUI	Graphic User Interface
IDE	Integrated Development Environment
ISE	Integrated Software Environment
LE	Logic Element
LUT	Look-Up Table
M9K	Memory 9-Kbit Block
MIPS	Microprocessor without Interlocking Pipe Stages
MLAB	Memory Logic Array Block
MMU	Memory Management Unit
RAW	Read After Write
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
TLB	Translation Look-aside Buffer
UUT	Unit Under Test
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
WAR	Write After Read
WAW	Write After Write

Chapter 1

Introduction

1.1 Motivation

The contemporary microprocessor market offers a vast variety of chips for new designs. The right choice of a microprocessor to fit for a specific application is a challenging task nowadays. Utilization of a configurable processor can facilitate this problem. A hardware designer can choose the configuration of a microprocessor which precisely matches design requirements. The advantage of FPGA configurable computing has brought the idea of implementing a general-purpose microprocessor on an FPGA chip [1]. This concept along with the growing demands for customization of the processor and its peripherals inspires FPGA vendors to include microprocessors in FPGA architecture. Manufacturers offer this feature in a form of hard or soft core. More then 32% of hardware developers use embedded FPGA microprocessors in their designs [2]. Configurable hard microprocessors also follow this trend. Customizable microprocessors form Tensilica, ARC and Improv for ASIC implementation offer a broad range of microprocessor optional features. Designers can choose the instruction set architecture to suit the application requirements. The microprocessor data path and pipeline structure are also customizable in order to meet the design constrains. In the ladder of the microprocessor solutions the configurable microprocessors occupy a niche in the middle between implementation on the dedicated hardware and the software running on general-purpose microprocessors [3]. Most of commercially available solutions for configurable microprocessors are oriented on the particular technology. A designer has to choose the target technology up-front. It may be a choice of FPGA from different manufacturers or technology process. But since the design is implemented and verified on a

chosen target platform it becomes very difficult to migrate the design to a different platform. The common situation is when a verified, proven design is to be upgraded due to additional new requirements or obsolescence of the target chip. Another challenge is the verification of an ASIC implemented microprocessor on FPGA platform. Design seamless transition from FPGA to ASIC is normally offered within technologies provided by the same manufacturer (e.g. Altera Stratix to HardCopy [4]).

Beside of the problems related to a market-oriented engineering, the configurable processor architecture with a versatile set of configuration features offers the fine grain optimization of hardware resources required the specific application. The combination of the possible processor configurations creates the exploration space which provides the opportunity for the research on decision making in the selection of the specific configuration features. Moreover, the flexibility of the configurable and portable design provides an opportunity to obtain the architecture with features not available in general purpose processor architectures.

The motivation of this project is to develop a configurable microprocessor architecture independent of the target technology. This architecture facilitates the optimization of the microprocessor architecture for a specific user application. The choice of the microprocessor architecture is provided by the set of user selectable features.

The application set of the proposed microprocessor architecture may include the following:

- The applications where operation with non-standard data bitwidth (256+) is required
- The applications with limited number of the allocated hardware resources
- The algorithm intensive low speed applications where implementation of Finite State Machine (FSM) is very complicated.
- The prototyping applications where the portability of the design is critical

The terms processor and microprocessor are used in this project to identify the same object, since in modern technical literature the term microprocessor is frequently contracted to just a processor.

1.2 Objectives and Contributions

The focus of this project is to develop, implement, and verify a portable configurable RISC processor architecture. The following goals are to be achieved in this project:

1) To develop a processor architecture configurable for a user specification by selection of required features provided in the design.
2) To implement configurable RISC processor using Verilog HDL [5] as independent module suitable for integration as a processor core in the processor-based digital systems.
3) To develop GUI that facilitates the choice of features for the processor configuration.
4) To select and implement a testbench for verification of generated processor configurations.
5) To generate a set of the distinguished processor configurations.
6) To verify the set of generated processor configurations.
7) To verify the portability of the design by implementing the set of processor configurations on several FPGA and ASIC platforms.
8) To implement one processor configuration in hardware using FPGA development board.
9) To test and verify the demo application on the FPGA development board.

The following contributions were made into the development of the Portable and Configurable RISC Processor Architecture project:

1) The development of the specific tool for the conversion of the compiled software code into Verilog HDL.
2) The conduction of the literature review on the configurable processor systems.
3) The development of the specific design flow for the configuration and implementation of the proposed processor architecture
4) The development of the Configuration Manager, the GUI-based tool for the facilitating the right choice of the configuration options of the proposed processor architecture
5) The development and implementation of the Verilog description of portable and configurable processor design based on the MIPS processor architecture.

6) The development and implementation of the demonstration example implementing the proposed processor design as a part of the Fibonacci number computation and visualization system.
7) The utilization of the proposed processor configuration framework as an educational platform for the processor organization teaching courses.

1.3 Project Organization

The rest section of the project is organized as follows:

Chapter 2 provides a background on the processor architecture and reviews related research studies.

Chapter 3 describes the design of the proposed configurable processor. The chapter shows implementation of configurable features for different processor architectures. The development of support software tools is also covered in this chapter.

Chapter 4 describes synthesis and implementation of the processor core on different hardware platforms i.e. FPGA and ASIC. Various combinations of configurable features and processor architectures implemented on different platforms create an exploration space. The chapter shows the subset of variants in that space. The included demo design illustrates a practical utilization of the processor core.

Chapter 5 describes the verification methods used to prove the functionality of the design on behavioral and hardware levels. The development of the testbenches and their properties are discussed and analyzed.

Chapter 6 analyzes results of the processor implementation on different platforms. Implementation of the processor variants are compared and evaluated.

Chapter 7 summarizes the conducted work and accomplishments of this project.

Chapter 2

Background

2.1 Introduction

In this chapter the relevant background in the processor architecture is presented. The chapter focuses on the description of MIPS RISC processor which is adopted as a base architecture for this project. The following sections review the research development in the area related to the project theme. The state-of-art of research and commercial configurable processors is outlined and analyzed in order to determine a niche taken by the presented project in a domain of available solutions.

2.2 Basic MIPS Processor Architecture

J. L. Hennessy et al. designed MIPS (Microprocessor without Interlocking Pipe Stages) in 1981. It was a result of their research of the processor architecture optimization for pipelining. The MIPS architecture proposed in [6] was used as a teaching example in their classical academic textbook [7] about the processor architecture design. Nowadays MIPS is widely used for the educational purposes [8].

Further development of MIPS architecture brought a row of the revisions of this architecture MIPS-I, MIPS-II, MIPS-III, MIPS-IV, MIPS32, MIPS64 [9]. The major market of the latest MIPS processor is embedded applications. They are implemented in numerous Cisco and Linksys routers, ADSL modems, Sony PlayStation 2, Sony Playstation Portable and many handheld computers[10] [11].

The choice of the MIPS processor architecture as a template for the configurable processor design in this project is justified by following reasons. The MIPS pipeline structure and organization is very well studied and described [12][13]. The MIPS microprocessor becomes very popular for academic purposes. Many researchers implemented [14][15] and enhanced it [16][17]. Therefore, the modification of the existing simple MIPS to a

configurable architecture is easier than modification of a sparely specified commercial processor. As any processor with unique instruction set, MIPS requires a custom software compiler. In order to complete the set of development tools for the MIPS, several open-source compilers and simulators have been developed [18][19]. Utilization to the open source software tools facilitates the development of the configurable processor.

The classic pipelined or unpipelined MIPS is a 32-bit RISC processor. The instruction set has 32-bit width for all instructions. Load/store MIPS Instruction Set Architecture (ISA) contains register file, which consists from 32 registers 32 bits long each. Two of them are assigned as special purposes registers. Register 0 is read-only and carries 0 values. It is used as a zero operand eliminating necessity to keep zero value in memory. Register 31 is used by special jump instructions to store return address. These instructions are used for calls and returns from subroutines. MIPS program counter has a width of 32 bit similar to the data path. The potential MIPS address space is up to 2 GB.

MIPS instructions are divided into three types *R-type*, *I-type* and *J-type*. The instruction format is shown in Table 2.1. R-type defines instructions operating with registers only. The instruction contains addresses of two operand registers *Rs* and *Rt*, address of the destination register *Rd* for result storing and the code of the executed operation. The I-type instruction also contains *Rs* and *Rd* but instead of the second source register it carries the 16-bit constant value *immediate*. This constant is used as an operand in arithmetic operations and as an address offset in load/store operations. J-type instructions represent *jumps* which change the program counter with 26-bit *address* enclosed in the instruction.

The instruction opcode has 6-bits width with possible opportunity of 64 basic operations. This instruction spare space allows adding of extended instructions such as FPU support. The simplified instruction set of MIPS processor is shown in Table 2.2. The basic set includes only two branch, four jump, and two memory instructions. Other instructions are arithmetical.

The supported data types are 8-bit bytes, 16-bit half words, and 32-bit words for integer data. Bytes and half words are loaded into 32-bit in two ways: extra bits are filled with sign extension or with zeros. After the load transaction they are processed as 32-bit integer operands.

Table 2.1. MIPS instruction format

Format	Bits					
	31 26	25 21	20 16	15 11	10 6	5 0
R-type	*opcode*	Rs	Rt	Rd	*shamt*	*funct*
I-type	*opcode*	Rs	Rt	*immediate*		
J-type	*opcode*	*address*				

Table 2.2. MIPS instruction set

Instr.	Description	Format	Opcode/ Func (hex)	Operation (Verilog-style coding)
add	Add	R	0/20	R[rd]=R[rs]+R[rt]
addi	Add Immediate	I	8	R[rt]=R[rs]+SignExtImm
addiu	Add Imm. Unsigned	I	9	R[rt]=R[rs]+SignExtImm
addu	Add Unsigned	R	0/21	R[rd]=R[rs]+R[rt]
sub	Subtract	R	0/22	R[rd]=R[rs]-R[rt]
subu	Subtract Unsigned	R	0/23	R[rd]=R[rs]-R[rt]
and	And	R	0/24	R[rd]=R[rs]&R[rt]
andi	And Immediate	I	c	R[rt]=R[rs]&ZeroExtImm
nor	Nor	R	0/27	R[rd]= (R[rs]|R[rt])
or	Or	R	0/25	R[rd]=R[rs]|R[rt]
ori	Or Immediate	I	d	R[rt]=R[rs]|ZeroExtImm
xor	Xor	R	0/26	R[rd]=R[rs]^R[rt]
xori	Xor Immediate	I	e	R[rt]=R[rs]^ZeroExtImm
sll	Shift Left Logical	R	0/00	R[rd]=R[rt]<<shamt
srl	Shift Right Logical	R	0/02	R[rd]=R[rt]>>shamt
sra	Shift Right Arithmetic	R	0/03	R[rd]=R[rt]>>>shamt
sllv	Shift Left Logical Var.	R	0/04	R[rd]=R[rt]<<R[rs]
srlv	Shift Right Logical Var.	R	0/06	R[rd]=R[rt]>>R[rs]
srav	Shift Right Arithmetic Var.	R	0/07	R[rd]=R[rt]>>>R[rs]
slt	Set Less Than	R	0/2a	R[rd]=(R[rs]<R[rt])?1:0
slti	Set Less Than Imm.	I	a	R[rt]=(R[rs]<SignExtImm)?1:0
sltiu	Set Less Than Imm. Unsign.	I	b	R[rt]=(R[rs]<SignExtImm)?1:0
sltu	Set Less Than Unsigned	R	0/2b	R[rd]=(R[rs]<R[rt])?1:0
beq	Branch On	I	4	if(R[rs]==R[rt]) PC=PC+4+BranchAddr

	Equal			
bne	Branch On Not Equal	I	5	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr
j	Jump	J	2	PC=JumpAddr
jal	Jump And Link	J	3	R[31]=PC+8; PC=JumpAddr
jr	Jump Register	R	0/08	PC=R[rs]
jalr	Jump And Link Register	R	0/09	R[31]=PC+8; PC=R[rs]
lui	Load Upper Imm.	I	f	R[rt]={imm,16'b0}
lw	Load Word	I	23	R[rt]=M[R[rs]+SignExtImm]
sw	Store Word	I	2b	M[R[rs]+SignExtImm]=R[rt]

SignExtImm ={16{immediate[15]},immediate} – extension of the immediate operand with the sign bit;

ZeroExtImm ={16{1b'0},immediate } – extension of the immediate operand with "0" bit;

BranchAddr = {14{immediate[15]},immediate, 2'b0 } – extension of the immediate operand with the sign bit and multiplication by 4;

JumpAddr = {PC[31:28], address, 2'b0 } – concatenation of the immediate operand with four MSBs of program counter and multiplication by 4;

The ISA architecture of MIPS defines the organization of the processor data path. The simplified 5-cycle implementation without pipeline is shown in Figure 2.1. The following actions are performed during each cycle:

1. *Instruction fetch cycle* (IF):
```
IR ← Mem[PC]
NPC ← PC + 4
```
2. *Instruction decode/register fetch cycle* (ID):
```
A ← Regs[IR6..10];
B ← Regs[IR11..15];
Imm ← ((IR16)16##IR16..31
```
3. *Execution/effective address cycle* (EX):
```
ALUOutput ← A + Imm;
or
ALUOutput ← A func B;
or
ALUOutput ← A op Imm;
or
ALUOutput ← NPC + Imm;
Cond ←(A op 0)
```
4. *Memory access/branch completion cycle* (MEM):
```
LMD ← Mem[ALUOutput]
or
```

```
Mem[ALUOutput] ← B;
if (cond) PC ← ALUOutput
```

5. *Write-back cycle* (WB):

```
Regs[IR16..20] ← ALUOutput;
Regs[IR11..15] ← ALUOutput;
Regs[IR11..15] ← LMD;
```

The design of MIPS is refined for pipelining. The multi-cycle version of the MIPS can be smoothly augmented with a pipeline. In a pipelined architecture all instructions are executed in the same number of cycles. This organization allows one instruction per cycle throughput. The block diagram of the pipelined MIPS processor is shown in Figure 2.2. The standard MIPS processor incorporates 5-stages pipeline. A drawback of pipelining is hazards. The most common type of hazard is the data hazard. The data hazard is a situation when a fetched instruction reads the same operand as one of preceding instructions writes. If the preceding instruction still propagates through the pipeline, the fetched instruction may read a wrong value. This data hazard is called Read-After-Write (RAW). There are two ways to handle data hazard – stalling and forwarding. Stalling means an artificial insertion of NOP instruction in the pipeline. The processor stalls the pipeline until a hazard is over. This technique results in wasting of processor clock cycles. Whereas, the forwarding does not have that disadvantage. Forwarding uses the pipeline property when the result of the instruction is available in one of pipeline stages but not written yet in a register. An additional hardware supports forwarding of the result to the stage to another stage where it is required. Another type of hazard, the control hazard is caused by branch instructions. The next fetched instruction after a branch may not be executed due to a result of branching. This would cause the pipeline to flash its contents and to stall. As a result the processor's speed slows down which is called the branch penalty. There are many methods to reduce the branch penalty. The simplest is a delay slot. It is efficient when delay penalty is one clock cycle. An execution algorithm of the processor with delay slot presumes that a next instruction after the branch is executed regardless of the branching result, whether the branch is taken or not. The compiler has to reorder the instructions and put in the delay slot an instruction which is to be executed despite a result of the branch instruction.

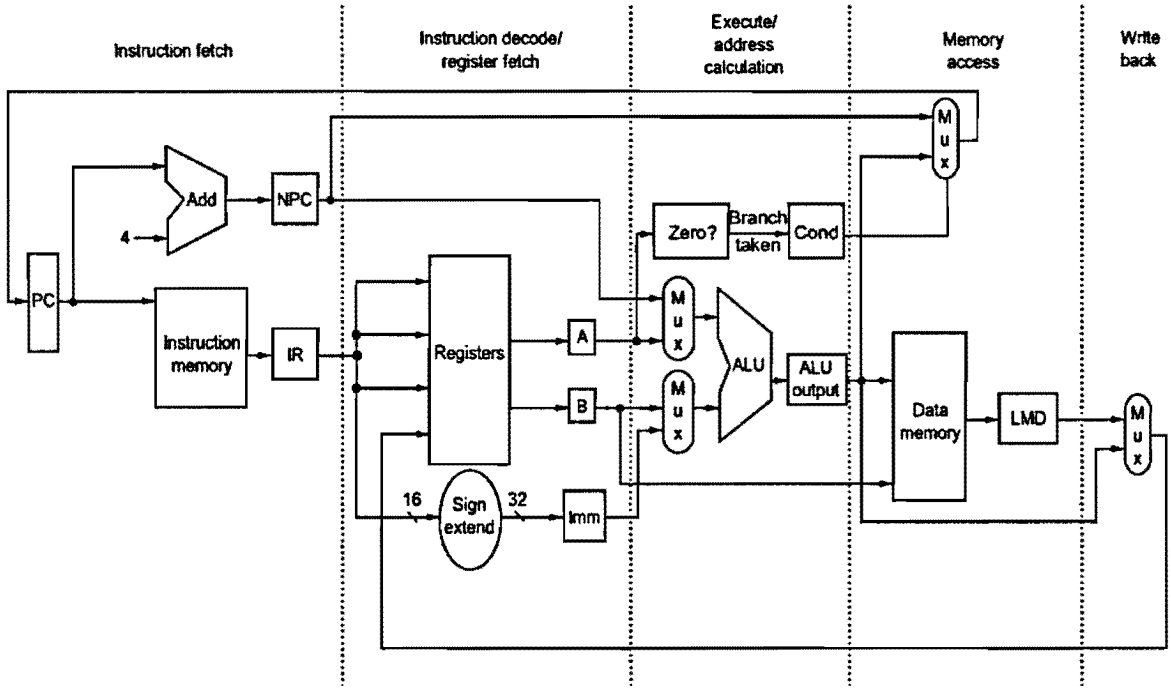


Figure 2.1. Block diagram of the multi-cycled MIPS processor.

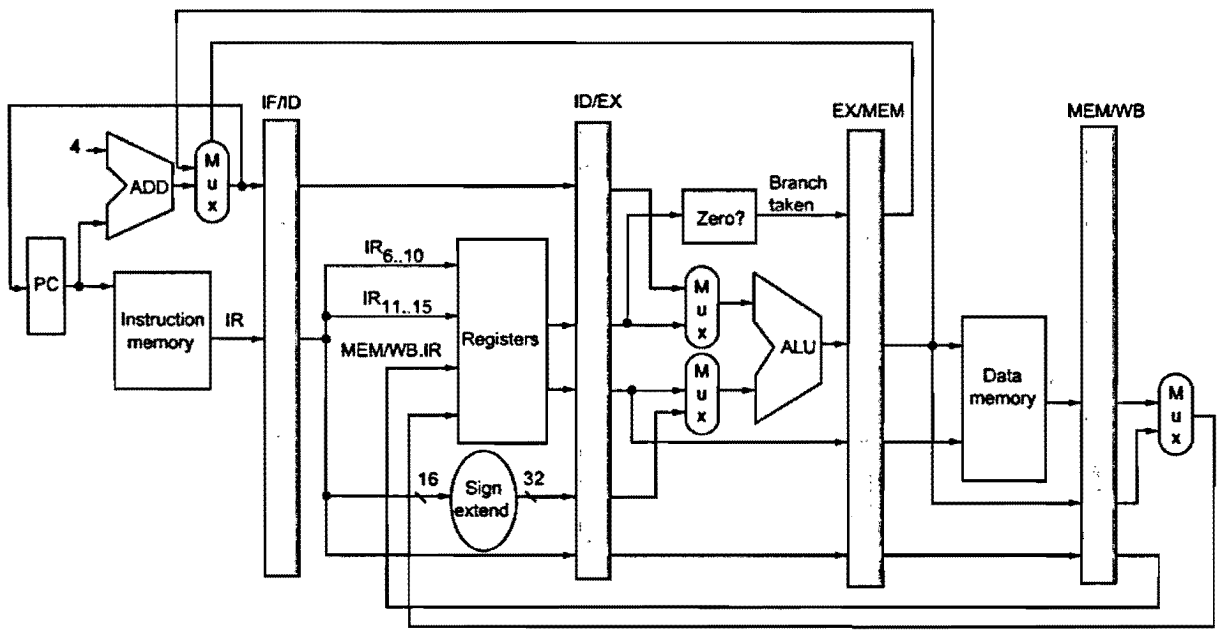


Figure 2.2. Block diagram of the pipelined MIPS processor

2.3 Closely Related Work

Two major approaches to automatic processor synthesis can be distinguished nowadays. One of them is template-based configurable processors. This methodology is mostly exploited by commercial products. The other one generates Application Specific Instruction set Processor (ASIP) based on Architecture Description Language (ADL) – a specific language developed for a processor architecture description.

2.3.1 Architecture Description Languages

Originally ADL was developed as a high level of abstraction description language for modeling processor's architectures. VHDL and Verilog languages do not completely suit for this purpose due to their orientation toward the hardware implementation.

Several ADLs were created in attempt to find the best way for processor architectural exploration and evaluation. One of them is nML which has been developed at TU Berlin [20]. This language is intended for automatic generation of the software tools for an explored processor architecture. In order to obtain a complete processor design, the developer has to create the separate ADL model and HDL description of the processor. It has limited ability to handle invalid instructions and can not describe architectures with parallel instructions [21]. A lack of the hardware generation feature in nML has been recently amended in its enhanced version Sim-nML [22]. A synthesizable Verilog description can be obtained with Structural Sim-HS tool included in Sim-nML. Generation of the processor RTL description from Sim-nML specification has been successfully tested with specifications of the microcontroller Motorola 68HC11 and microprocessor Intel 8085. Performance of the synthesized architecture has not been estimated.

Similar to nML, ISDL architecture description language was developed by MIT LCS [23]. ISDL is instruction set specific language, specifically oriented toward Very Long Instruction Word (VLIW) processor architectures. However, multi-cycle and multi-word instructions are not fully supported by ISDL. GENSIM system software automatically generates Instruction Level Simulator (ISL) specific to the developed architecture. The ISL is capable to simulate cycle-accurate and bit-true execution of the program. The achieved speed-up of such simulation compared to the simulation of the Verilog model is 34x [24]. Lately, HGEN tool

has been developed in order to automatically generate Verilog RTL model from ISDL description [25].

The language EXPRESSION [26] is capable to describe correctly multi-cycle and multi-word architectures. A characteristic feature of EXPRESSION is the partition of the design flow into two phases. Evaluation and exploration of the chosen architecture is performed in Exploration Phase. This phase is supported by Exploration Simulator and Exploration Compiler automatically generated by the software itself. The compiler and simulator allow rapid comparative estimation and simulation of candidate processor architectures. A chosen solution is finally adjusted in Refinement Phase. The software toolkit generates an optimized Instruction Level Parallelism (ILP) compiler and a cycle-accurate simulator from EXPRESSION description. Using these tools the developer may perform detailed processor evaluation and verification of memory hierarchy (e. g. cache, TLB). The link to RTL synthesis is provided by the HDLGen tool [27][28] which generates the VHDL model from the EXPRESSION description. In order to test results, the automatically synthesized DLX processor [29] has been compared to its hand-written version. Despite 20-40% worse results in terms of speed, power consumption, and area, it is shown that the design time is an order of magnitude less. The paper, however, do not compare the development efforts for the processors with reported degraded performance designed manually and automatically.

One of the most prominent and widely used ADL is Language for Instruction-Set Architectures (LISA) [30]. Due to its C-like syntax, LISA is very attractive for architect designers who are beginners in utilization of ADL for ASIP development. The structure of LISA allows a designer to specify details sufficient for automatic generation of the software tools set containing compiler, assembler, linker, and simulator. These tools are used in the stage of exploration of the developed processor architecture. During this phase a designer can tune and verify designed processor by changing the ADL description. Repetition of compilation cycles does not introduce a considerable delay due to complete automation of the development environment [31][32]. LISA description contains enough architectural information for generating a synthesizable HDL model.

The advantage of architectural exploration using LISA description inspired founders to develop the integrated LISA Processor Design Platform (LPDP) [33]. Efficiency of the LPDP has been evaluated on the example of ICORE processor. This ASIP processor is oriented

toward FFT realization, sampling–clock synchronization for interpolation and carrier frequency offset calculation. The handwritten version of ICORE has been compared with the automatically generated version. The generated ICORE shows the same clock speed, 1% area overhead, and 15% more power consumption. Design efforts for LPDP ICORE are approximately one month and a week vs. three months for the original handwritten version.

LPDP has apparent advantage of a fully developed and integrated system for ASIP design. The convenient user interface and support of different operational systems [34] distinguishes it from general research projects, where the integrated environment accelerates the exploration of various processor architectures. Performance of the LPDP generated processors can compete with commercial handwritten versions. However, the necessity of manual development of a processor data path diminishes the advantages of this system.

The advantages of utilizing LISA ADL have been recognized by many researchers. LISA has been used as a base ADL for numerous research projects focused on ASIP development [35][36][37]. High level of LISA development has stimulated its implementation in a commercial processor generation system. LISA 2.0 is used by CoWare company in CoWare Processor Designer [38]. This platform is dedicated for design and optimization of ASIP. LISA 2.0 architectural description was used to generate a full set of processor software tools and the RTL description in Verilog, VHDL, and SystemC. This commercial processor development system lately has been used in several ASIP research projects [39][40] for rapid processor architecture exploration.

2.3.2 Configurable Processors

The concept of flexible microprocessor architecture is exploited by many authors. Two major approaches can be distinguished: *configurable* and *reconfigurable* processors [41][42]. The term configurable presumes customization before manufacturing. Whereas, the reconfigurable processor implies configurability after manufacturing. Runtime dynamic changing of the configuration is a powerful feature of the reconfigurable architecture. This approach offers reuse of the same silicon design by multiple applications without additional manufacturing cost. Examples of the reconfigurable microprocessors are shown in [43][44]. These designs represent multiple computing units connected by a sophisticated reconfigurable network. Narrow reconfiguration ability is implemented in the computing units as well. Due

to implementation of the described processors using technology process, their flexibility is limited.

Configurable processors became very popular last decade. They can be divided into soft and hard-processors. Hard processors are intended for Application-Specific Integrated Circuit (ASIC) implementation. Xtensa LX3 offered by Tensilica [45] is an example of commercial configurable processors. This is 32-bit RISC ISA processor that allows the designer to perform the configuration by choosing predefined options from the menus. The following main groups of features can be added and tuned for Xtensa LX3:

- Execution Unit and ISA Options (multipliers, DSP engines, FPU, custom instructions, etc.)
- Interface Options (DMA, FIFO, GPIOs, interrupts, debug port, etc.)
- Memory Subsystem Options (caches, memory management unit (MMU), parity, cache organization, etc.)

The chosen configuration of the processor is automatically processed by Xtensa Processor Generator software. The complete solution is represented by the RTL description and EDA scripts. The example of the configurability of Xtensa processor is demonstrated in implementation of the multi- standard video decoder [46]. Two different processor configurations are used to create the stream processor and pixel processor. Each processor is enhanced with specific video instructions. The optimized ISA architecture of the processors allows video decoding in the software only.

A similar set of configurable features is proposed by the ARC for the ARC 600 Core and ARC 700 Core processor families [47][48]. ARChitect Processor Configurator [49] extends the processor design with Single Instruction Multiple Data (SIMD) instructions, integrated coprocessor instructions, compound instructions and many others. More than 20,000 preconfigured options can be selected by the developer.

The explicit benefit of these processor systems is that their template-base synthesis does not require an extensive knowledge of the processor architecture. A developer can obtain fully functional ASIP from a specification with very high level of abstraction. Rapid automatic synthesis allows fast evaluation of several solutions and optimization of the final ASIP. Flexibility of the template-based processor architecture is limited by the set of the predefined options which is not always suitable for research projects.

Along with commercial configurable products numerous research projects show interesting results in this field. Advantages of the implementation of Dolby Digital (AC-3) decoder with the hard configurable processor are shown in [50]. The paper convincingly proves that utilization of the configurable processor tuned for the specific audio application increases performance of the processor and reduces the required size of the die.

Very fast growth of the FPGA performance and density accompanied with sophisticated development tools has made FPGA devices very attractive for the implementation of a processor architecture. Reprogrammable nature of FPGA determines the definition of the processor implemented in FPGA as a soft processor. Therefore, it is not a surprise that the most well-known configurable soft processors are offered by major FPGA vendors Xilinx and Altera. Xilinx promotes 32-bit RISC soft-processor Microblaze [51] with configurable peripherals. It has limited configurable abilities for the core structure. This soft processor is proposed as alternative to the hard-processor core PowerPC 440 [52] implemented in the Xilinx Vertex-5 FXT FPGA family. Table 2.3 shows the performance of the MicroBlaze processor for different FPGA families.

Table 2.3: MicroBlaze Processor v7.2 Performance Levels

Architecture	Performance	Maximum Clock Frequency	Maximum Dhrystone 2.1 Performance
5-Stage Pipeline	1.19 DMIPs/MHz	235 MHz in Virtex®-5 FXT	280 DMIPS
3-Stage Pipeline	0.95 DMIPs/MHz	106 MHz in Spartan®-3A DSP	100 DMIPS

Along with high-end Microblaze processor Xilinx developed Picoblaze 8-bit Picoblaze soft processor [53] with no options for the configurability. Source code is open for evaluation and modification. The HDL model is offered on very low gate level description. Picoblaze can be implemented only on Xilinx FPGA platform. It became very popular due to its simplicity, free distribution, and availability of software tools. Popularity of Picoblaze inspired Bleyer [54] to develop Pacoblaze – a behavioral version of Picoblaze. This model incorporates maximum level of parameterization. The high level definition file represents a wizard for implementation of the possible versions of Pacoblaze. Low level definitions files comprise a hierarchical ladder, which an experienced user can employ for configuration of a

custom version of the Pacoblaze architecture. Using Verilog optional compilation the author has created the specific configuration language, which supports a variety of custom configurations.

Xilinx major competitor Altera offers Nios II – second generation of Altera's soft processors [55]. It is 32-bit RISC general purpose processor with 32-bit width instruction set similar to MIPS. Altera offers Nios II in three different configurations: economy (e), standard (s) and fast (f). Table 2.4 outlines specific features of each configuration. All versions allow adding up to 256 custom instructions. The choice of the required configuration is supported by SOPC Builder software. The resulting configuration is generated in a form of FPGA programming file.

Table 2.4. Nios II different version features.

Processor Version		Nios II/e	Nios II/s	Nios/f
Performance	DMIPS/MHz	0.16	0.75	1.17
	Max DMIPS	28	120	200
	Clock (MHz)	150	135	135
Area	LEs	600	1300	1800
Pipeline		unpiped	5	6
Branch	Prediction	-	static	dynamic
ALU	Multiplier	-	3-cycle	1-cycle
	Divider	-	-	optional
	Shifter	serial	3-cycle	1-cycle

The idea of modification of the commercially successful soft processors lies in the basis of UT Nios soft processor [56]. It is an attempt to use a different approach of configurability of the Altera Nios II. In contrast to original Nios II, UT Nios has optional 16/32-bit data path width and reduced 16-bit instruction word width. Instruction set supports five custom instructions. The register file has a configurable size with 32 visible registers window. There is an option of integer multiplication. Benchmark evaluation shows insignificant 1% average and 56% for particular applications speed-up. UT Nios requires 31% more FPGA resources than Altera Nois II.

Another attempt to use Altera Nios II architecture for the research project is UTMT II [57]. The design exploits the multithread processor architecture using different UTMT II

configurations with multiprocessor core structure. Despite the poorer performance then Altera Nios II, an advantage of the 45% area saving has been reported.

Four stages pipeline is used in UWindsor Nios II (UWN2) [58], another Altera Nios II compatible soft processor. The parameterization of the UWN2 is limited to 10 options. The best achieved area saving is 47%, while the clock speed is 7% worse than Nios II.

The success of commercial soft processors does not discourage numerous researchers to develop other configurable soft processor architectures. The example of a very well developed and tested configurable processor is LEON3 [59]. This 32-bit processor core is designed as synthesizable VHDL model compatible with SPRC V8 architecture. The open source design offers many configurable options for the optimization. The portability of the design is verified for Altera and Xilinx FPGA platforms on multiple development boards. The LEON3 is also suitable for ASIC implementation. The best achievable clock speed is 140 MHz for FPGA platform and 650 MHz for ASIC. The hardware design is supported by a set of software development tools including simulator, compiler, linker, Newlib embedded C-library, Eclipse based IDE, etc.

The general purpose traditional processor architecture is focused on the execution of different applications of the same hardware. While most of embedded processors run only one specific application. The Application Specific Instruction-set Processor (ASIP) incorporates an idea of optimization of the processor architecture according to a running task. This approach significantly improves performance and speed of the processor. Due to a high cost of implementation of ASIP in ASIC devices, the usage of FPGA for ASIP becomes very attractive. The design of video-processor [60] demonstrates the implementation of ASIP in FPGA. The targeted Altera Nios II/f processor core is augmented with the custom hardware for block manipulation. The efficient data reuse achieves three order of magnitude acceleration compared to software implementation.

Instruction Set Extensions (ISE) is another approach to optimization of ASIP architecture. The design of the E-ASIP (ETRI-Application Specific Instruction Processor) for the CAVLC of H.264/AVC decoder [61] uses this technique. In order to improve the processor performance for the specific function, the instruction set is extended with additional special purpose instructions. The number of basic instructions is reduced down to the minimum set required for the functionality.

Exploration of the different aspects of processor configurability is performed in the SIxD application-specific soft processor [62]. The processor incorporates several configurable options, such as length and width of the data space, custom branching and shifting instructions, ability to choose subset of instructions and the choice of SIMD mode. The design has been successfully tested with MPEG-7 Motion Activity Descriptors application.

Aside from the traditional RISC ISA architecture stands CUSTARD (CUStomizable Threaded Architecture) - a customizable threaded FPGA soft processor [63]. The available configurable features include data path width, number of threads, threading type, custom instructions, custom memory blocks, forwarding, and register file parameterization. The design is supported by the custom C-compiler developed by the authors. Reported evaluation results show a significant 2.41x average speed-up of the single-treaded CUSTARD with custom instructions compared to Xilinx Microblaze. The disadvantage of the considered processor is a twice larger area overhead.

The versatility of configurable processors shows the necessity of the performance evaluation for different configurations. Such estimation has been fulfilled using Soft-processor Rapid Exploration Environment (SPREE) [64]. This system is able to generate RTL description of the soft processor from the high level architecture description. The SPREE base configurable core is similar to MIPS and Altera Nios II processors. The textual description of the processor data path and ISA are used as an input for SPREE. Multiple generated processors were benchmarked and compared with all three versions of Atera Nios II e/s/f. The best variants show the same or better performance than Nios II. For example, 80- MHz three-stage pipelined processor generated by SPREE is 9% smaller and 11% faster than Nios II/s. The study examines the influence of different processor architectural features on the performance. The impact of the following options is investigated: shifter implementation, multiplication support, pipeline depth, pipeline organization, forwarding, application-specific architecture customization, and ISA subsetting. The optimized processors achieve improvement of the performance per area 24.5% on average. Saving of the power and area is obtained 25% on average.

2.4 Summary

In this chapter, the main concept of RISC processor architecture is presented. The MIPS processor specification and architecture are described. The related configurable solutions in the form of soft and hard processors are studied and analyzed. The complementary software tools are outlined and examined for the reviewed processors.

Chapter 3

Configurable Processor Proposed Design

This chapter describes the structural components of a configurable processor design and high level design of the processor data path. It also illustrates the design of the processor's control unit.

The data path design of the configurable MIPS processor is organized in the following sequence:

- 5-stages pipelined architecture
- 4-stage pipelined architecture
- Multi-cycle unpipelined architecture
- One-cycle unpiplined architecture

The methodology of the design of configurable processor architecture in this project differs from the classical approach described in [12]. The classical approach does not presume the configurability of the described processor architecture. The evolution from the simplified form to the more complex is chosen for the educational purpose in order to facilitate understanding. Instead of this approach, here the most complicated design is taken as an initial and gradually modified to reach the simplest. The most complicated design contains the majority of the components present in other architectures, therefore it is logical to use this design for the transformation into other architectures.

The 5-stage architecture is the most advanced and complicated design with maximum hardware overhead. All other architectures are derivatives from the initial 5-stage pipelined architecture. The configuration options control transformation of the initial design toward other three architectures. The major structural components are shared in all data path architectures. Components pertained only to a specific architecture included as options controlled by the configuration engine.

Since the control unit design highly depends on the data path realization, the control unit design has limited configurability. The control unit is unique for each of the four types of

architectures. The choice of the appropriate control unit is also supported by the configuration engine.

3.1 Datapath Components

3.1.1 ALU

In order to support MIPS instruction set, ALU is designed to have the capability of performing operations with two input operands *A, B*. Table 3.1 shows the correspondence between ALU operations and supported MIPS instructions.

Table 3.1: Supported ALU operations

ALU Operation	MIPS Instructions
Addition (result = A+B)	add, addu, addi, addiu, lw, sw
Subtraction (result = A-B)	sub, subu
Logical conjunction (result = A and B)	and, andi
Logical disjunction (result = A or B)	or, ori
Logical disjunction with negation (result = A nor B)	nor
Logical exclusive or (result = A xor B)	xor, xori
Pass operand (result = A)	jr
Load immediate to upper word (result = B << 16)[1]	lui
Logical right shift	srl, srlv
Logical left shift	sll, sllv
Arithmetic right shift	sra, srav
Compare result to 0	beq, bne
Set less than	slt, slti, sltu, sltui

[1] Shift for 16-bits applies only for classic MIPS instruction set. In general, this parameter is configurable in the design.

The symbol for ALU is shown in Figure 3.1. The list of the ALU signals and their description are shown in Table 3.2.

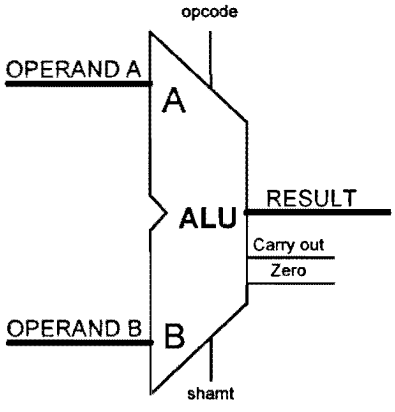


Figure 3.1: Processor ALU symbol

Table 3.2: ALU signals

Signals	*Dir*	*Bitwidth*	*Description*
OPERAND A	In	Data path	First arithmetic/logic operand.
OPERAND B	In	Data path	Second arithmetic/logic operand
RESULT	Out	Data path	Result of arithmetic/logic opration
opcode	In	3 or 4 bits. Depend on the instruction set	Code of the executed operation
shamt	In	Configurable; 5 bits by default	Shift amount. The number of bits to shift the operand B
Carry_out	Out	1 bit	Carry out of the arithmetic operations
Zero	Out	1 bit	Produces "1" when result is equal 0

The ALU module has a configurable bit-width and subset of operations. Support for shift commands and "Set Less Than" commands is optional and may be excluded from the design to reduce hardware overhead.

The ALU is an asynchronous device and consists of the combinational logic only.

3.1.2 Register File

The register file design consists of 32 registers with configurable bit-width. The register with address 0 does not have memory elements. It comprises hardwired 0's. The register file symbol is shown in Figure 3.2.

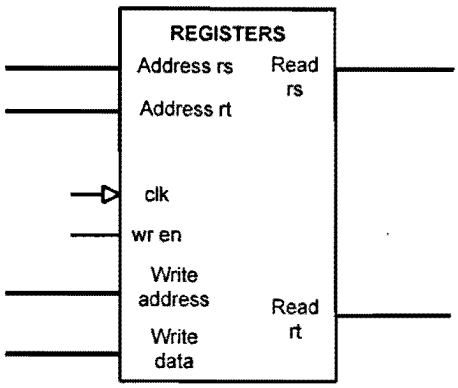


Figure 3.2: Register file symbol

The register file module works simultaneously in write and read modes. It asynchronously produces data on two output ports from two registers which addresses are set on the corresponding address inputs. It also synchronously writes data from the *Write data* input port to the register which address is set on the *Write address* input. Writing is controlled by *wr en* signal. In order to support configurability, two writing modes are implemented: Read-First Mode and Write-First Mode. The Figure 3.3 and Figure 3.4 show the difference between these two modes. The writing mode defines the order of access in case of simultaneous read-write access to the same register. In Read-First Mode the data set on the write port immediately appears on the corresponding read output port and is written to the register later. In Write-First Mode data is written to the register on the first clock edge and only after that the data is set on the read output port. The writing mode is a configurable feature of the register file module. The timing diagrams demonstrating the difference of Read-First and Write-First modes are shown in Figure 3.5 and Figure 3.6.

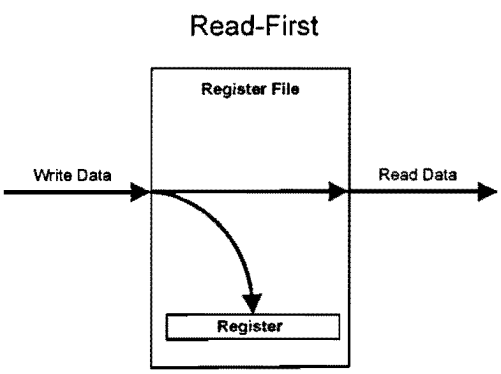


Figure 3.3: Regfile Read-First block diagram

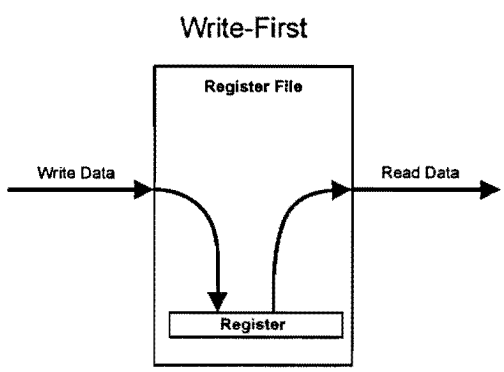


Figure 3.4: Regfile Write-First block diagram

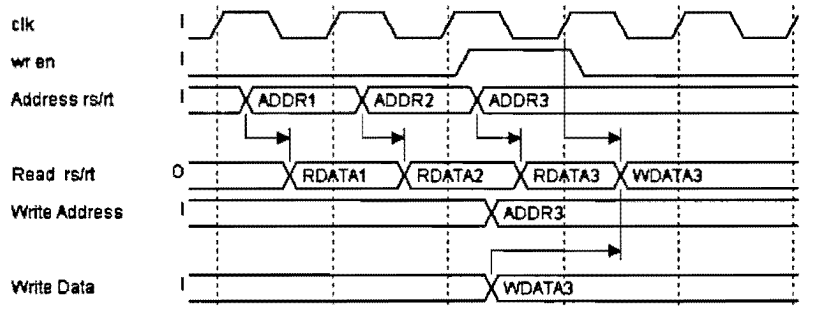


Figure 3.5: Regfile Write-First mode timing diagram

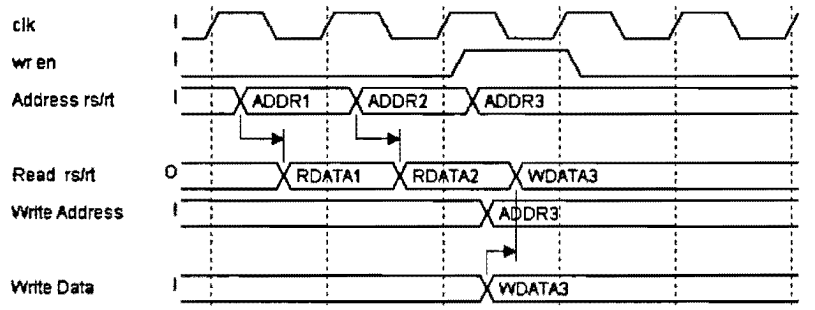


Figure 3.6: Regfile Read-First mode timing diagram

3.1.3 Instruction Memory

The instruction memory module is designed as an asynchronous static memory. The size and bit-width are configurable features of this module. The instruction memory has read-only access. The symbol of the instruction memory is shown in Figure 3.7. The design of the module provides the opportunity to use it as the built-in instruction memory in the microcontroller type of applications. The module also can be used as a prototype of the L1 cache design in the advanced processor applications.

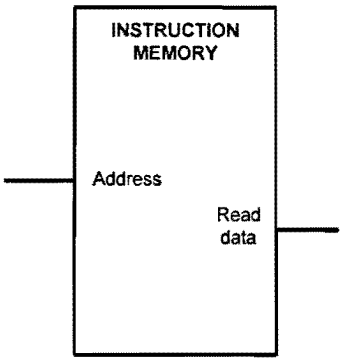


Figure 3.7: Instruction memory symbol

3.1.4 Data Memory

The data memory module has an asynchronous read access and synchronous write access. The writing access is enabled by *MemWrite* signal. The symbol of the data memory is shown in Figure 3.8. Due to MIPS load-store architecture, simultaneous read-write access to the same memory address in not possible. Table 3.3 shows the input/output signals of the data memory module. The timing diagram describing the read/write data memory access is shown in Figure 3.9. The size and bit-width are configurable features of this module. Similar to the instruction memory, the data memory module can used in microcontroller style applications and as a prototype of the L1 cache design.

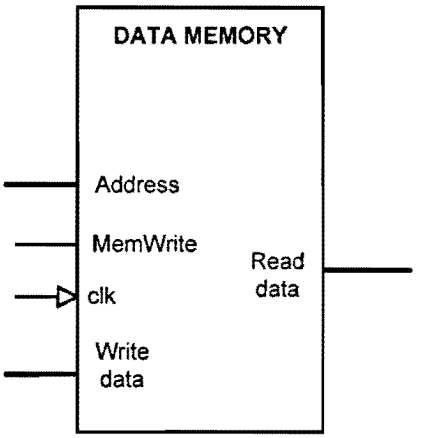


Figure 3.8: Data memory symbol

Table 3.3: Data memory signals

Signals	*Dir*	*Bitwidth*	*Description*
clk	In	1bit	Processor clock
Address	In	Data path	Memory address
Write data	In	Data path	Data to be written in the memory
MemWrite	In	1bit	"1" enables writing to the memory
Read data	Out	Data path	Data read from the memory

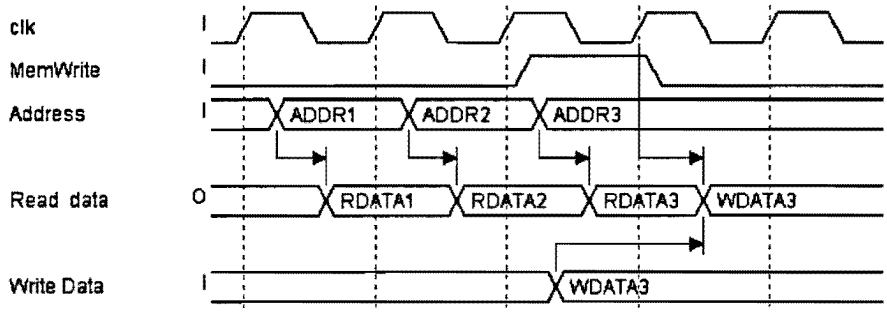


Figure 3.9: Data memory read/write timing diagram

3.1.5 Program Counter

The program counter shown in Figure 3.10 is a register with a write enable (*wr en*) input. It holds the current instruction address. The bit-width of the program counter is configurable and depends on the size of the instruction memory. The program counter module does not have a built-in counting capability. An external adder is required for implementing the counting function. The functionality of the program counter is illustrated by the timing diagram shown in Figure 3.11.

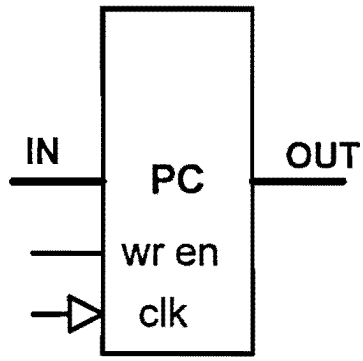


Figure 3.10: Program counter symbol

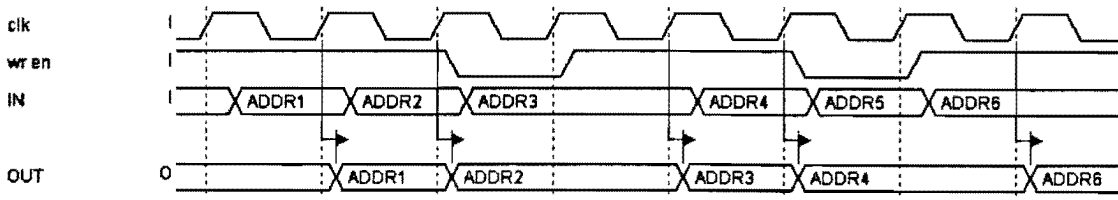


Figure 3.11: Program counter timing diagram

3.1.6 Sign Extension

In order to support I-type of commands, MIPS architecture requires extension of the 16-bit immediate operand to the bit-width of the processor datapath. In most cases it means filling the extra bits with the sign bit of the immediate operand. For commands *andi, ori, xori* it requires filling the extra bits with 0s.

The sign extension module has a configurable bit-width depending on the processor datapath bit-width. The extension mode is controlled by the input port *sign_ext*. The module is asynchronous and contains only the combinational logic.

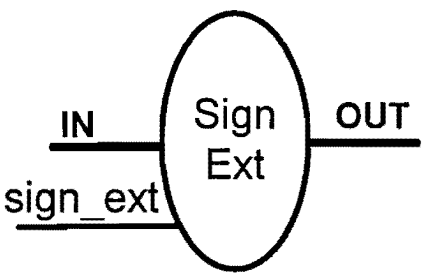


Figure 3.12: Sign Extension symbol

3.2 Control Unit Design

The control unit module contains all logic required for producing control signals for configurable architectures. The control unit design is different for all four architectures. Due to the significant variations of the processor architectures, a unified configurable design of the control unit is not feasible. Each control module comprises a unique design specific to the chosen processor architecture. The control unit is connected to the data path trough the unified interface. It consists of the same set of input and output signals. This set contains a maximum possible number of the control signals pertained to the most complicated processor configuration. Therefore instantiation of the specific control module in the chosen processor design is a call of the corresponding name of the module.

Due to redundancy in the connection interface, not all interface signals are used in the control unit design. They are just not connected to internal parts of the control unit. This does not introduce a problem since they are ignored by the synthesis tool.

All control units support the same set of instructions. The way they are decoded and handled differs in control units for different architectures. The distinguishable feature of the control module for the pipelined architectures is an internal pipelining of control signals. This internal pipelining corresponds to the pipelining of the data path. Detection and handling of all type of hazards native for a pipelined architecture occur in the control unit.

3.3 Pipelined Architecture Design

The design of the pipelined architecture follows the concept described in [12]. It exploits the Instruction Level Parallelism (ILP) when all stages of the processor execute different instructions simultaneously.

Configuration options for pipelined processor architecture in this project consider two architectures: five stages and four stages. The maximum number of stages can be identified as follows:

- Instruction Fetch (IF)
- Instruction Decode(ID)
- Execute (EX)
- Memory Access (MEM)
- Write Back (WB)

The designed architecture implements the following features of the pipelined MIPS processor organization:

- Branch delay slot
- Data hazards handling
- Control hazards handling
- Forwarding

In the pipelined architecture one instruction is issued every clock cycle. Therefore in the ideal situation, the throughput of the processor is equal to the clock speed. However, hazards cause the throughput to be reduced. The implementation of the branch delay slot reduces the penalty for control hazards but requires explicit support in the compiler. The data hazards are handled by forwarding and pipeline stalling. Read After Write (RAW) is the only type of data

hazard possible in the proposed pipelined architecture. The Write After Read (WAR) and Write After Write (WAW) hazards are not possible in the designed processor.

3.3.1 Five Stages Pipelined Processor

The block diagram of the processor with five pipeline stages is shown in Figure 3.13. The IF stage contains program counter, instruction memory, instruction address adder and multiplexer which selects the source of the next instruction address. An issued instruction is decoded in ID stage and analyzed in the control unit. Selection of the address source of the next instruction is controlled by signals from the control unit. In case of a sequenced order of execution, the address is increased by a number of bytes in the instruction word. Though this number is a configurable option, on practice 4 bytes organization is chosen. It allows a standard instruction set and compiler to be used. In case of a taken branch or jump instruction, the next instruction address is calculated in ID stage. Since *jr* instruction uses a register for the jump address, it may introduce a data hazard. The forwarding from EX, MEM, and WB stages is used to reduce or avoid a stalling penalty for data hazards. The ID stage includes Register File, adder for the taken branch address calculation, sign extension module, and comparator for the branch decision. Implementation of a separate comparator rather than using the ALU on EX stage reduces the branch hazard penalty down to one cycle. That penalty is covered by the branch delay slot technique. A drawback of this approach is a hardware overhead required for the extra comparator and forwarding multiplexers.

ID stage multiplexers support forwarding from EX, MEM, and WB stages. The jump address is obtained by combining lower bits from instruction and PC higher bits.

Instruction decoding is implemented by routing data and addresses contained in the instruction code to the corresponding recipients. The op-code and function code are decoded in the control unit.

The EX stage comprises ALU and multiplexers. Multiplexers are used to support the instruction set. They select ALU input data according to a processed instruction. They also support forwarding from MEM and WB stages. One of the multiplexers is dedicated for selection of the register address to be written in case of a register command.

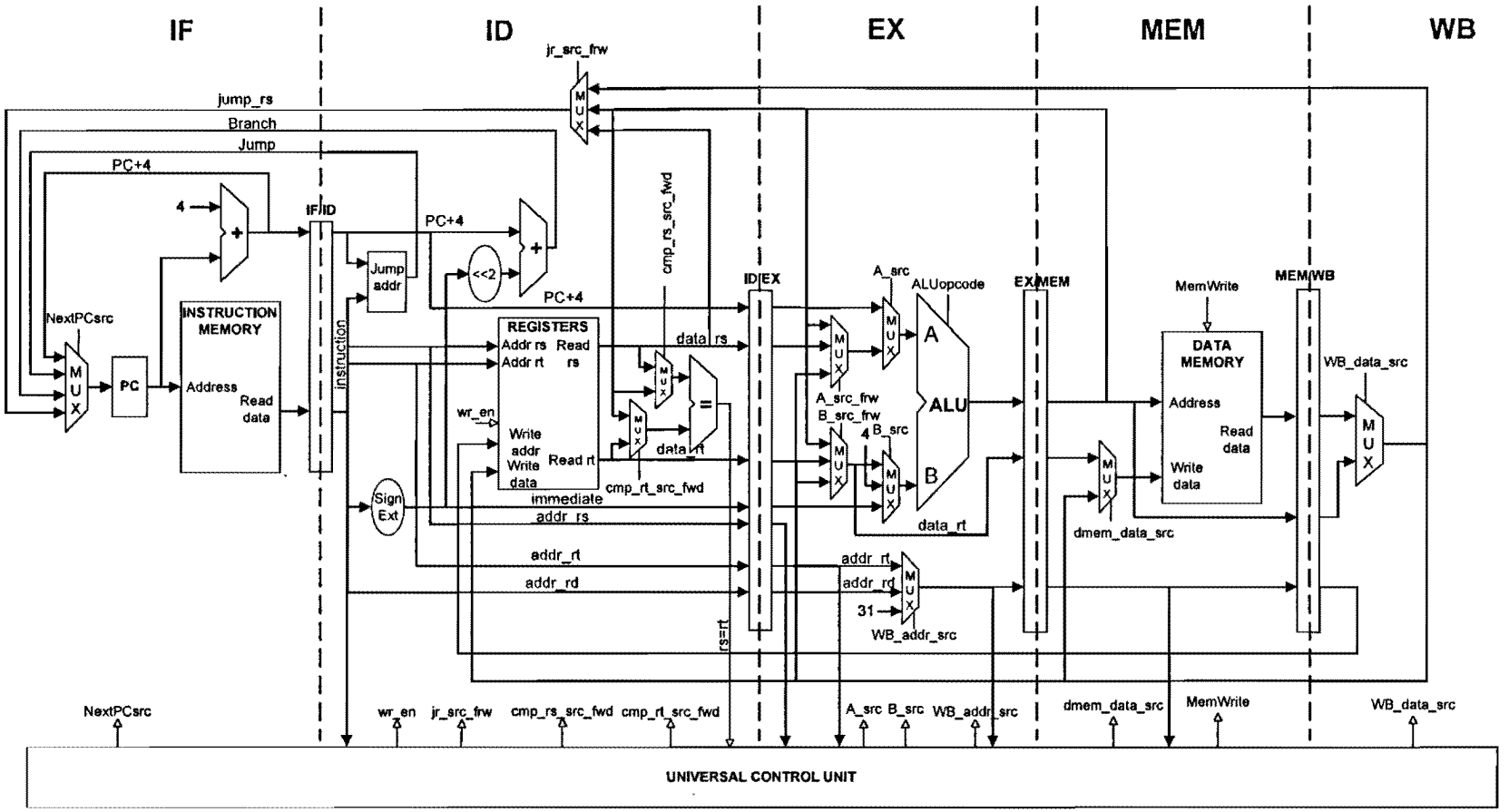


Figure 3.13: Block diagram of five stages pipelined processor

The MEM stage consists of data memory and forwarding multiplexer. If the propagated trough this stage instruction is not *load* or *store*, the MEM stage just pass all data through without any changes.

The WB stage graphically is represented only with multiplexer. This multiplexer selects pipelined data to write back either from the data memory or ALU. The choice depends on the processed instruction. Implicitly WB stage includes the writing portion of the register file. The writing to the register file occurs on this stage.

The control unit is shared between all stages. It gets data and generates control signals for all data path components in order to support proper functionality of the processor.

3.3.2 Four Stages Pipelined Processor

The classic five-stage pipelined processor has a disadvantage of having separated MEM stage. This stage is active only in case of load/store command and for other commands, data just pass though the pipeline. The design of the four-stage processor addresses that under-exploitation of hardware resources. This architecture combines EX and MEM stages in one EX stage. The block diagram of the described architecture is shown in Figure 3.14.

In the conventional five-stage processor the address for load/store operations executed in the MEM stage is calculated in the ALU located in the EXE stage. Executing these two operations in the same stage in series would drastically reduce the throughput. In order to avoid this reduction, an additional adder is placed in the ID stage. This adder calculates the address of the memory access.

Elimination of one stage excludes some scenarios of data and control hazards. The control unit has to handle lower number of forwarding and stalls. Therefore, the four-stage architecture has following advantages:

- Improved latency
- Reduced probability of data and control hazards
- Reduced complexity of the control unit
- Reduced hardware for the pipelining and forwarding

The trade-offs for implementing four-stage architecture are:

- Additional adder
- Possible slower clock speed

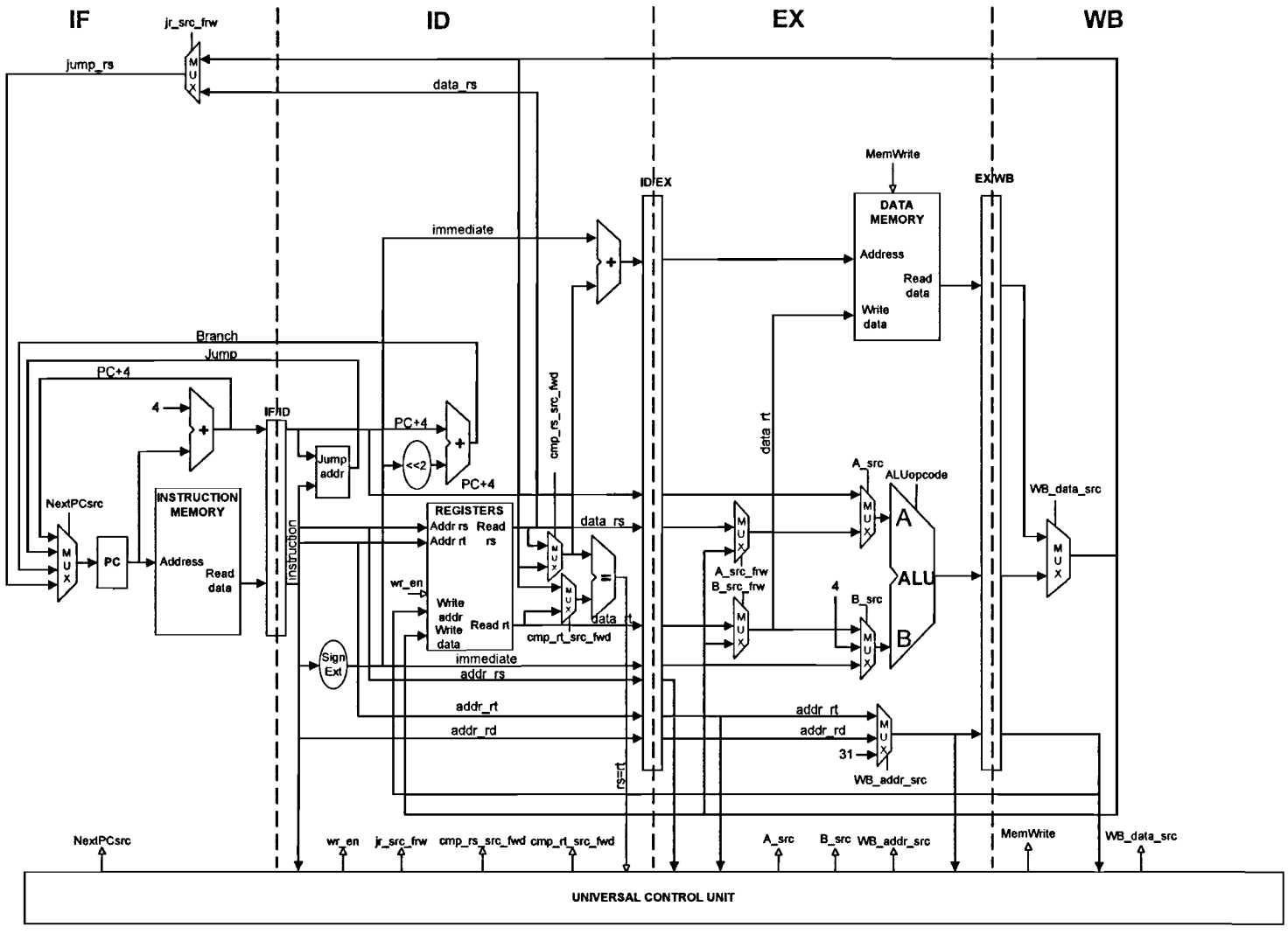


Figure 3.14: Block diagram of four stages pipelined processor

The comparator and address adder in this architecture work in parallel. In general, an adder structure is more complex than comparator's. Therefore, the additional adder may cause a longer delay in the ID stage. The delay highly depends on an implementation platform. In case of FPGA, implementation may not necessarily lead to a longer delay in the ID stage. See Chapter 4 for implementation details.

It is clear that the four-stage architecture can be derived from the five-stage one. The choice of a specific pipeline configuration is achieved by selecting particular compilation keys. The HDL descriptions of five-stage and four stage architectures are contained in the same module. The compilation keys control only the difference in the designs causing partial compilation of the code pertained to a specific architecture. The design of the control for four stages architecture is similar to five stages. Nevertheless due to structural differences, it is realized as a separate module. The choice of a control unit for the particular architecture is also controlled by compilation keys.

3.4 Unpipelined Architecture Design

The basic principle of the unpipelined MIPS architecture is described in [12]. Though inferior to a pipelined architecture as per clock speed, the unpipelined architecture offers benefits of simplicity and lower hardware overhead. For the design with limited hardware resources the unpipelined architecture may be preferable.

3.4.1 One-Cycle Processor

The one-cycle processor architecture is the least efficient from speed point of view. But it requires least amount of hardware resources. The execution of any instruction takes only one clock cycle. Therefore, the execution of the longest instruction defines clock speed of the processor. For the architecture shown in Figure 3.15 the longest is a *load* type instruction.

To transform the base five stages pipelined design to a one-cycle design, the following is performed:

- All pipeline registers are replaced with dummy pass modules
- All forwarding hardware is disabled
- A specific for one-cycle architecture control unit is implemented

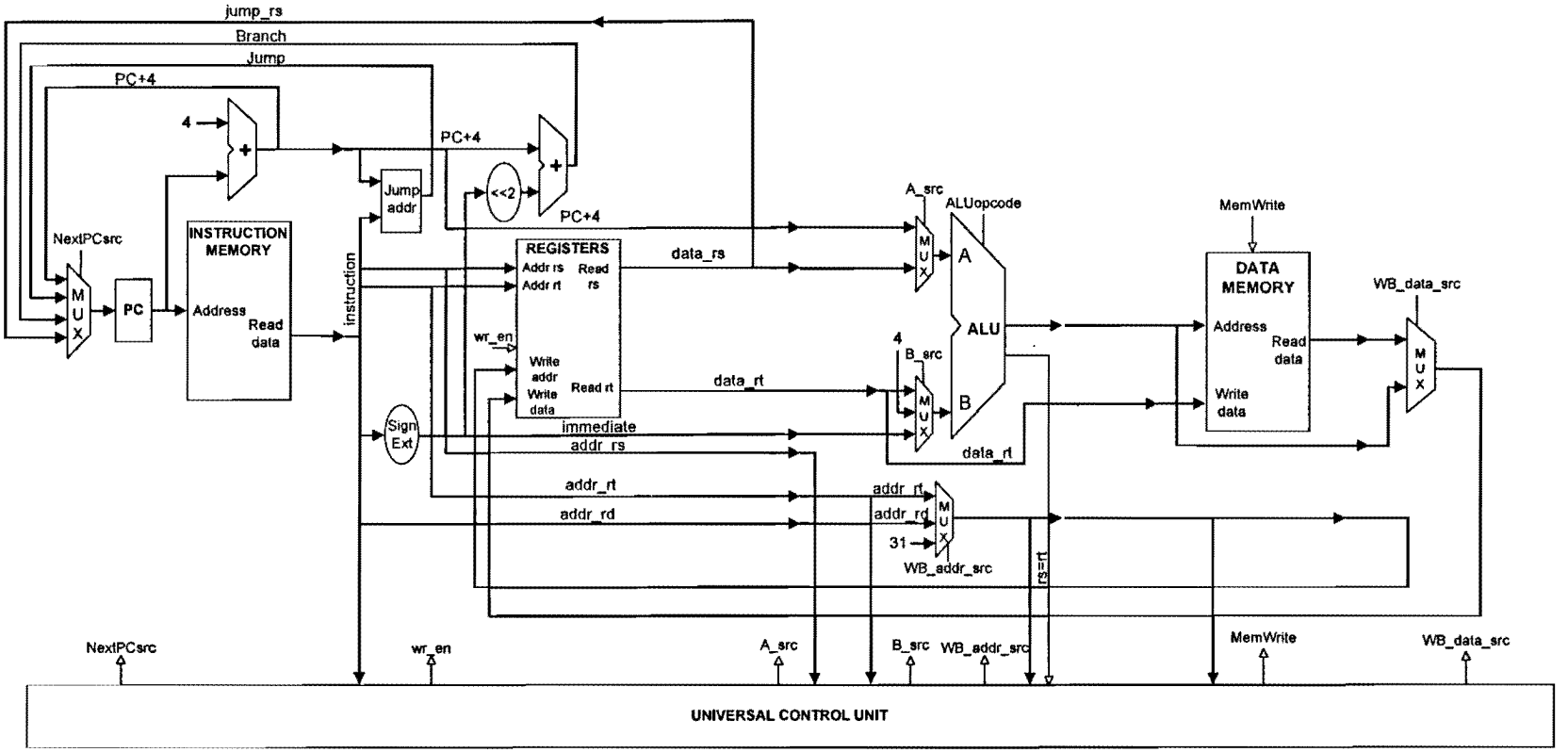


Figure 3.15: One-cycle processor architecture

In the architecture shown in Figure 3.15, only register file, program counter, and data memory are synchronous. The rest of the processor is asynchronous. The control unit in this architecture the control unit's design is very simple. This is because the handling of hazards, stalls, and forwarding is removed. Basically, only instruction decoding is left in the control unit module.

3.4.2 Multi-Cycle Processor

The multi-cycle architecture is a mid-way between the pipelined and one cycle designs. It has more hardware overhead than one-cycle architecture, but benefits from the shorter average execution time of instruction. Similar to the pipelined architecture, it is divided into five sections where the execution of each section occurs in one clock cycle. The execution time of each instruction depends on the type of the instruction. It varies from three clock cycles for branch type instructions to five cycles for load type. Each stage is separated by a register that holds the data produced by each stage. Since each instruction is issued after completion of a previous instruction, no forwarding components are required. They are disabled by appropriate compilation keys. Also some supplement pipelining provisions are removed. Removal of the next instruction address adder also contributes to a hardware reduction. ALU is used for the calculation of the next instruction address and for instruction operation because they occur in different processor cycles. In the same way, ALU is used to calculate the branch address, sparing another adder. The block diagram of the multi-cycle architecture is shown in Figure 3.16.

The control unit significantly differs from all other architectures. Since the execution of an instruction requires several clock cycles, the design of the control unit utilizes Finite State Machine (FSM).

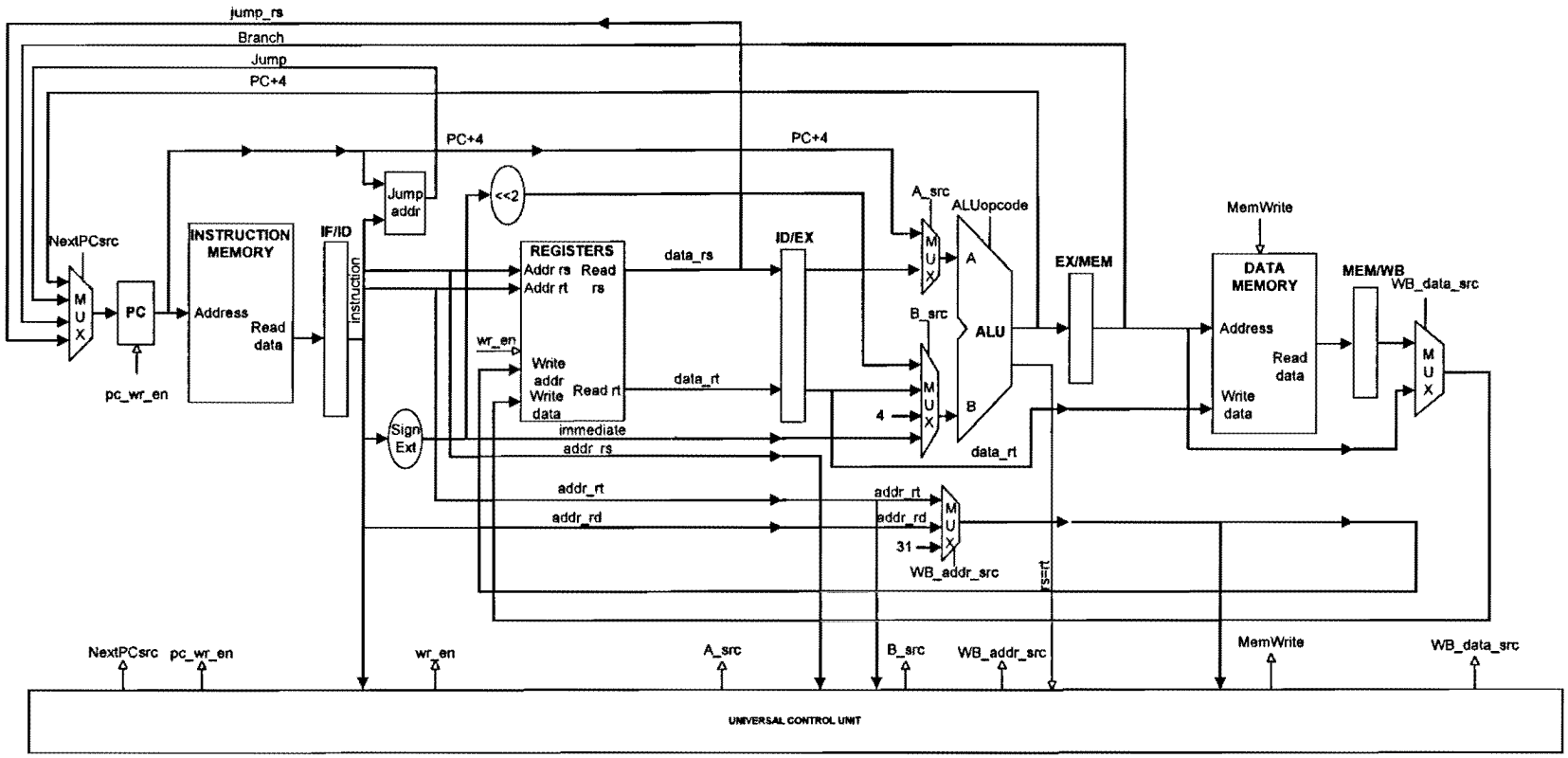


Figure 3.16: Multi-cycle processor architecture

3.5 Configuration Control

Selection of a processor architecture is controlled by a set of compilation keys. Configurability of the design exploits Verilog HDL capability of macro definitions and optional compilation. The configuration control is based on the following Verilog compiler directives:

- `define
- `ifdef, `elsif, `else, `endif
- `ifndef

The structure of all design modules contains a base part common to all processor architectures and optional parts pertained to the chosen configuration. All configuration control definitions are contained in one file *processor_config_flat.v*. In order to change the processor configuration, it is necessary to enable or disable definitions provided in the configuration file or change a defined number in the macro in case of a numerical configuration option (e.g. bit-width). Modification of the configuration file can be achieved in two ways: manually and by software. The manual modification requires basic skills and knowledge in processor architecture. In order to facilitate a choice of a processor configuration, the specific GUI based wizard is developed. Using the wizard requires only a basic specification of the processor configuration.

The style of HDL design defines all possible flexibility in the configuration file. The current state of the design explores a limited subset of the possible options defined in the configuration file. Other options are left for the further development.

3.6 Configurable Features

The configurability of the proposed processor design is not limited to the choice of four architectures. Each processor architecture has a subset of configurable features. The subset of features is common for all architectures. The combination of features and processor architectures creates a significant exploration space of possible processor designs. The design configuration space is shown in Table 3.4.

Table 3.4: Set of configuration features available for the processor core

Configurable features	Processor architectures			
	One-Cycle	Multi-Cycle	4-stage Pipeline	5-stage Pipeline
Data path bitwidth	x	x	x	x
Instruction set subsetting	x	x	x	x
Data memory size	x	x	x	x
Instruction memory size	x	x	x	x
I/O address space size	x	x	x	x
I/O bitwidth	x	x	x	x
FPGA optimization	x	x	x	x

3.6.1 Data Path Width Parameterization

The processor data-path bit-width is not limited. In theory, an arbitrary number can be chosen. For sake of practicality, the only bit-widths 16, 32, 64, 128, and 512 are explored. The original MIPS instruction set is designed for 32-bit processor. Therefore, the full utilization of wider data bus requires extension of the existing instruction set with additional commands. Nevertheless, the MIPS instruction set fully compatible with wide data path processors with full support of R-type commands and limited support of I-type commands.

For all data path sizes the bit-width of an instruction remains 32-bit. This constrain insures the compatibility of the designed processor with MIPS instruction set and therefore utilization of MIPS software tools (e.g. compiler, linker, simulator).

3.6.2 Instructions Set Parameterization

The configurability of the instruction set is limited to the choice of enabling/disabling two sets of commands: Shift and Set Than Less. Selection of these sets includes a barrel shifter and comparator in the ALU module. In case of an application specific processor design, saving of hardware resources may be achieved when these instructions are not used.

Due to the open configurable architecture of the processor, any custom instruction can be added to the design. The support of a custom instruction may require a modification of the control unit and data path design.

3.6.3 Data Memory Parameterization

The following parameters of the data memory can be configured:

- Memory size (address bit-width)
- Organization – number of bytes in one memory word
- Mapping window – limits in the processor address space allocated for the data memory

The data bus width of the data memory is defined by the processor bit-width.

3.6.4 Instruction Memory Parameterization

Parameterization of the instruction memory supports the following options:

- Memory size (address bit-width)
- Organization – number of bytes in one memory word

Mapping is not required for the instruction memory due to a separated access interface for this type of memory.

3.6.5 I/O Memory Parameterization

The I/O memory interface features the following configuration options:

- Memory size (address bit-width) represents the allocated for input/output address space size
- Data bit-width
- Mapping window defines limits for the processor address space allocated for the input/output

Data bit-width is not limited but should be equal or less than processor bit-width. A choice violating this rule would cause a waste of hardware resources.

3.6.6 FPGA Optimization

To reduce the consumed hardware resource of the target FPGA, BlockRAM (BRAM) can be utilized. In modern FPGAs built-in hardware [65] such as BRAM is made available to the designers to improve the design speed and minimize the consumed FPGA's area.

By default, an FPGA compiler implements memory components in BRAM whenever it is possible. In order to facilitate BRAM implementation, the memory HDL behavioral description must contain a dedicated address register [66]. The typical BRAM organization with built-in address register is shown in Figure 3.17.

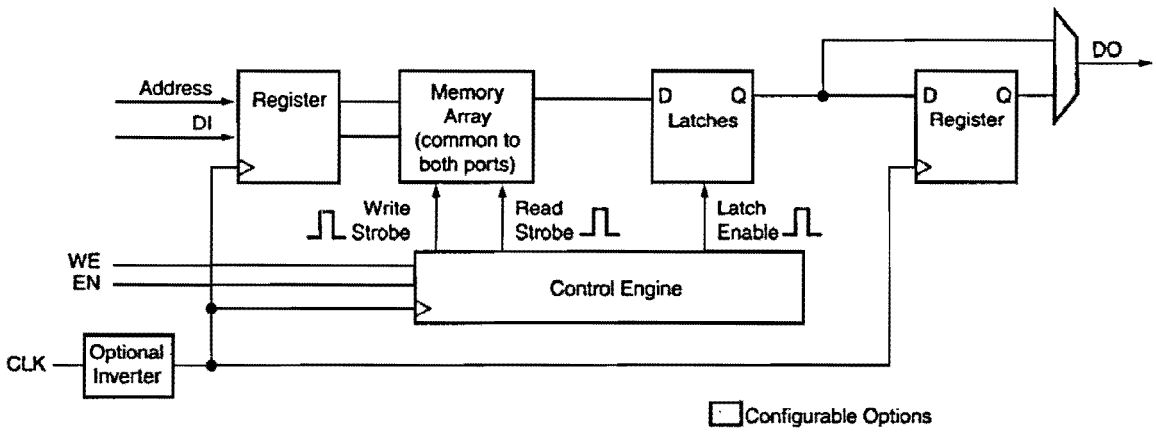


Figure 3.17: BRAM logic diagram

Implementation of the instruction memory with a separate address register requires duplication of the PC register. This register is implied as a part of BRAM by the synthesis tool. Therefore, this apparent increase in the number of structural components in HDL leads to a reduction in hardware overhead in final implementation.

Utilization of BRAM for the data memory requires duplication of the register allocated for the pipelining of the data memory address. Modification of the five stages pipelined architecture for BRAM optimization is shown in Figure 3.18. Dotted line shows hardware blocks implemented in BRAM.

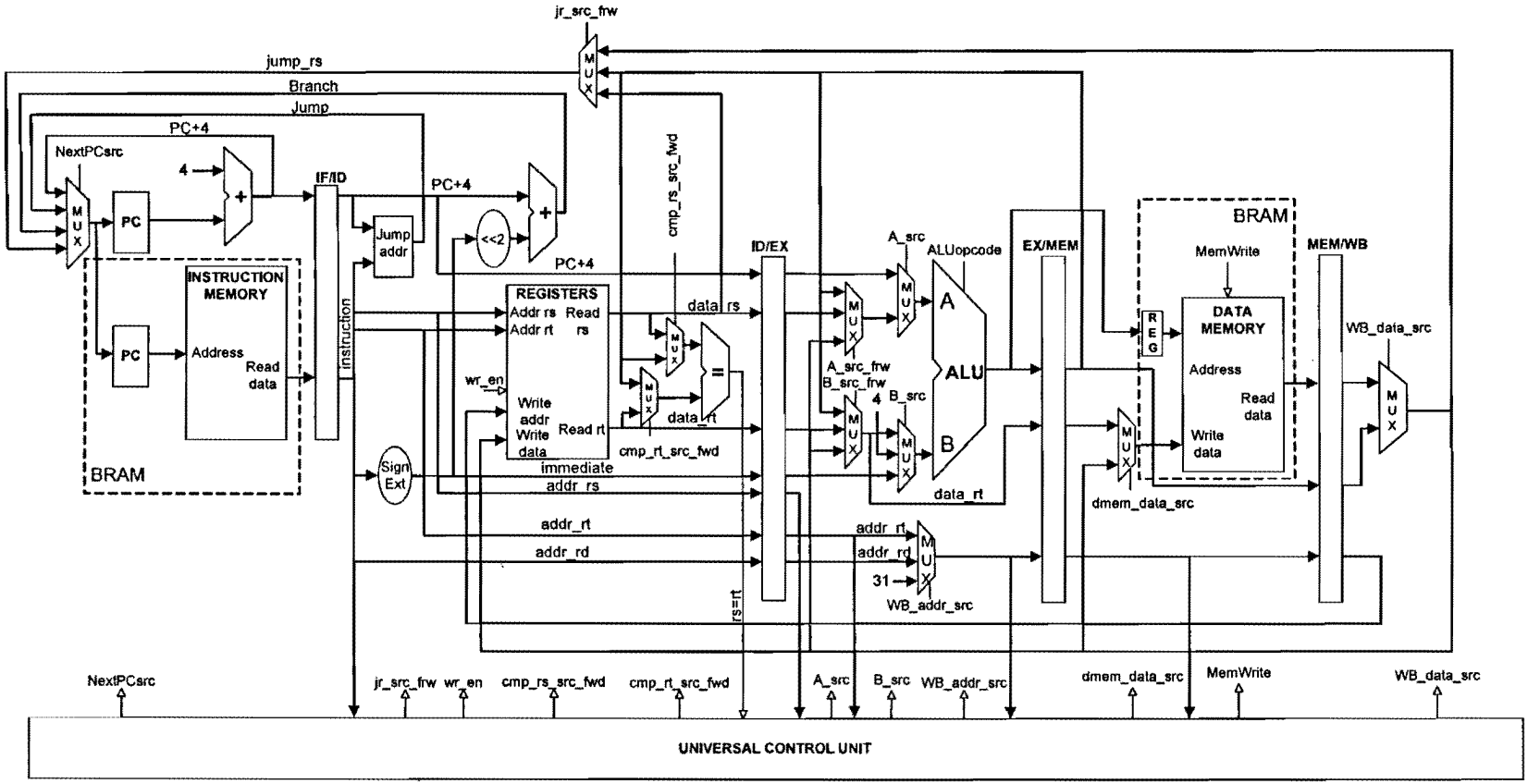


Figure 3.18: BRAM optimization for five stages architecture

3.7 Input/Output Interface

Practical usage of the processor core requires a hardware provision for communication with external devices. The MIPS instruction set does not have specific input/output commands. The general approach in this case is to use a data memory address space for input/output interface mapping. Figure 3.19 shows the hardware implementation of the external memory sharing the address space with an internal memory of the processor core.

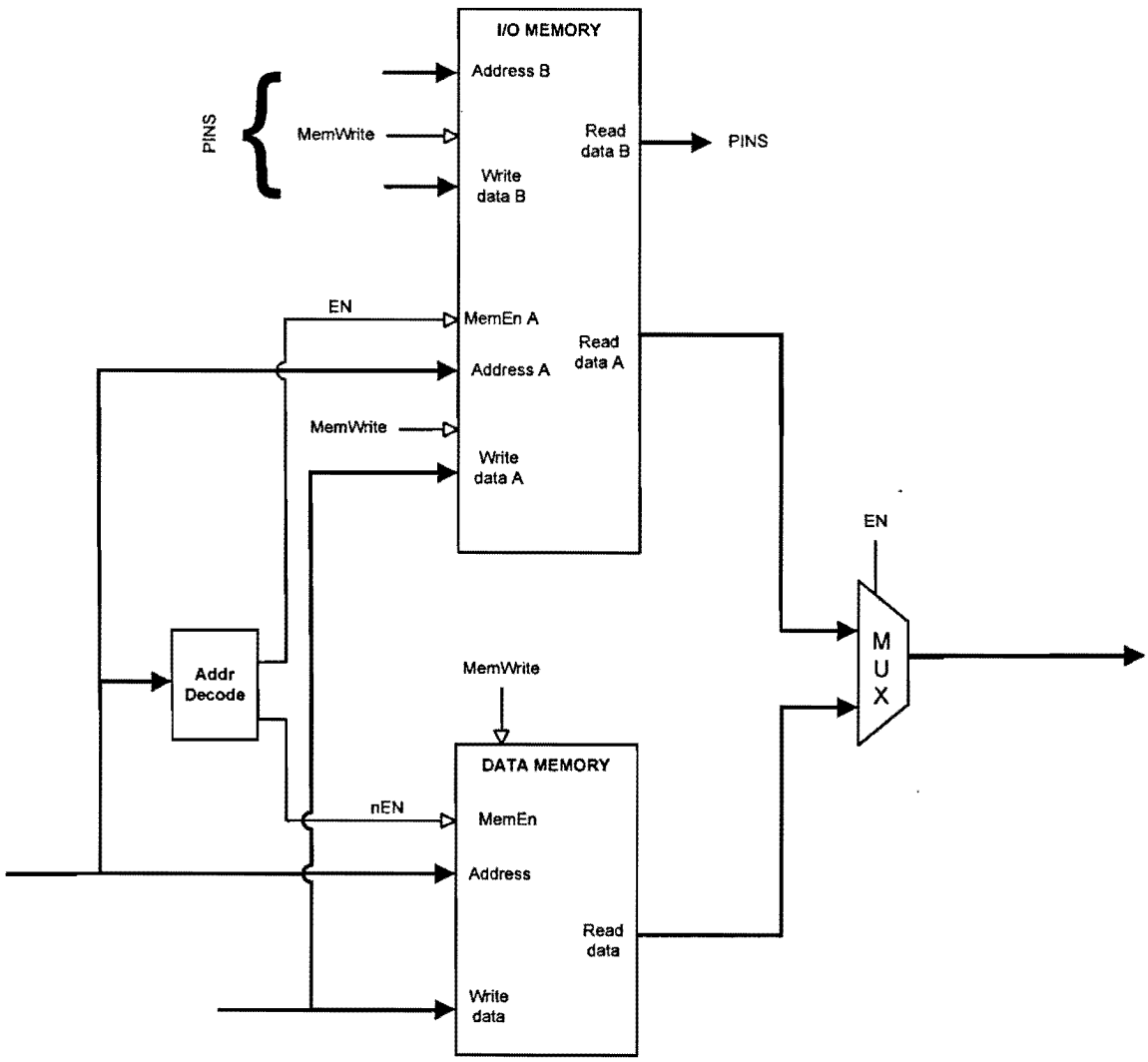


Figure 3.19: Block diagram of the input/output interface organization

I/O memory module shown in Figure 3.19 may encapsulate any input/output structure. It can be a set of input/output registers, multiple or single external memory blocks, or any

combination of them. The internal address decoder defines only two types of accesses – to internal data memory or external memory. In order to use the external memory address space, an additional hardware overhead is required, e. g. supplementary address decoder, registers, memory, etc. The example of connection of an external device to the designed processor core is described in the section 4.5.3 "Demo Platform Interface Design" of this project.

The I/O memory interface is the only connection of the designed processor core to the external world. In case of a standalone implementation of the processor core, I/O memory interface is routed to pins of a chip of the chosen platform. This interface is used for connection of the testbench in this project.

3.8 Configuration GUI

Selection of the combination of architecture and features of the implemented processor core requires certain skills in a processor architecture design. Manual tuning of the configuration file requires attention to a correlation of compilation keys. A wrong combination of the selected keys may leads to a non-functional processor design. In order to ensure selection of the correct set of compilation keys, GUI wizard software is developed. The appearance of the wizard software is shown in Figure 3.20. The wizard software is written using VB.NET language in Microsoft Visual Studio 2008 Integrated Development Environment (IDE).

The GUI has two tabs: *Configuration* and *Architecture*. The *Configuration* tab contains comprehensible menu with available configuration options. Each drop list in the menu contains the commonly used numbers for practical processor configuration. If the choice is not in the list of suggested options, it can be entered manually in the textbox. The default values of the menu pertain to 32-bit five stages pipelined processor with predefined parameters. Therefore, configuration of the default design does not need any processor architecture knowledge.

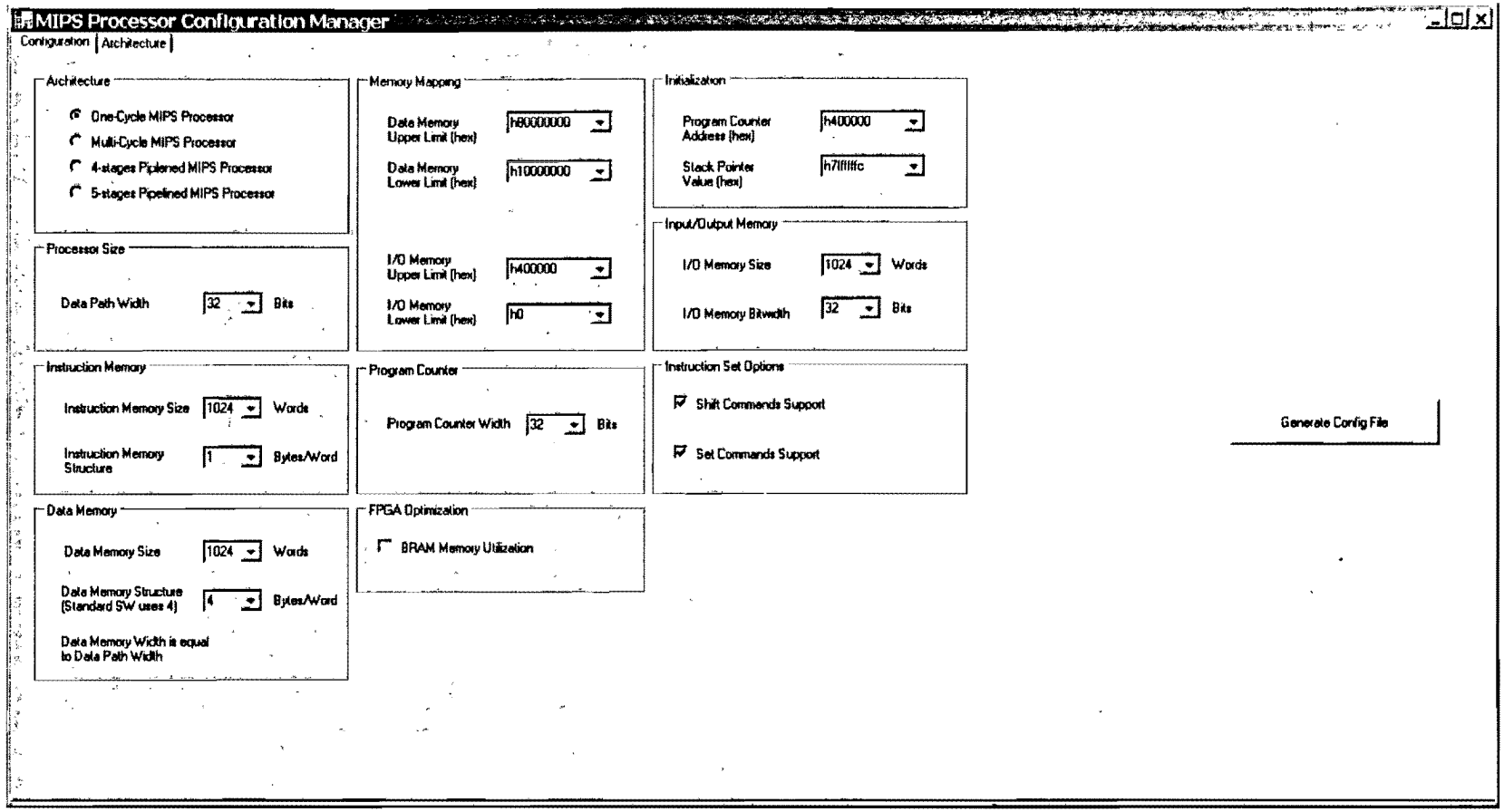


Figure 3.20: Configuration GUI wizard screenshot

The *Architecture* tab shows the block diagram of the selected processor architecture. If user changes the chosen configuration the block diagram dynamically changes. The pictures of the block diagrams are shown in Figure 3.13, Figure 3.14, Figure 3.15, and Figure 3.16. That visualizes the selected architecture and helps to make the choice of other options.

The generation of the resulting configuration file occurs when *Generate Config File* button is pressed. If the file already exists the dialog window appears for confirmation.

3.9 Summary

This chapter has described the details of the design of the MIPS based configurable processor. The described processor design has a choice of four different major processor architectures. In addition to architectural configurability, the multiple structural options are presented for each processor architecture. The specific software tool is developed to facilitate and coordinate the choice of multiple configuration options. In order to show capability of the designed processor core, the demo design is developed for the existing FPGA development board. The demo design includes the hardware and software parts. The hardware section comprises the selected processor core and interface peripherals. The software section contains demo program and LCD driver.

Chapter 4

Implementation

In this chapter the implementation of the proposed configurable processor on FPGA and ASIC platforms is explored. The design and implementation flow is described in details. Various processor variants are generated and synthesized using the custom-proposed configuration tool and standard FPGA/ASIC development tools. The obtained implementation results characterize the maximum clock speed and hardware resources of each processor configuration. The detailed interpretation of the implementation data is provided in chapter 6. A practical implementation of the proposed processor core is shown on the example of the demo design implemented on FPGA development board.

4.1 Hardware Components and Development Tools

Assorted software and hardware tools were used in this project. The development tools used for implementation and verification of the design are shown in Table 4.1. The Xilinx development board "Spartan-3E FPGA Starter Kit Board" was used for hardware implementation of the demo design.

Table 4.1: Implementation and development tools

Name	Usage Description
Xilinx ISE 9.2.03i	Development and Xilinx FPGA synthesis
Altera Quartus II 9.0sp2	Development and Altera FPGA synthesis

Mentor Graphics ModelSim SE 6.2g	Behavioral and FPGA post-route simulation
Synopsys Design Analyzer	ASIC synthesis
MipsIT 1.3.0	Compilation and linking of MIPS software
sreg2vlog 1.0	Custom format conversion (*.srec → Verilog *.v)
MIPS Processor Configuration Manager	Custom processor configuration
MipsSim 1.5.1	MIPS software simulation
Microsoft Visual Studio 2008	IDE for development of sreg2vlog 1.0 and MIPS Processor Configuration Manager GUI

4.2 Design and Implementation Flow

The design-implementation flow of the proposed configurable processor involves the usage of different tools from different vendors. The major stages of the flow are shown in Figure 4.1.

Architecture Block Diagram stage is dedicated for the design of the high level architecture described in Chapter 3. At this stage one of the four proposed architectures is to be selected and the set of configuration options is to be defined for the implementation.

MIPS Test Program Software is the software development stage where the test or any other specific application is designed. In general, the executed software can be designed using arbitrary development tools supporting MIPS architecture. This project uses MipsIt IDE for development and MipsSim for simulation. The IDE supports both C/C++ and Assembler languages. The linker is capable to produce multiple output formats. None of them can be implemented directly into the instruction memory module. Therefore, the Motorola S-record format [67] is selected for the conversion at the next stage.

MIPS Code to Verilog Conversion stage utilizes the custom conversion tool *srec2vlog* developed in this project. The tool converts Motorola S-record format to Verilog assignment operators for ROM as shown in Figure 4.2.

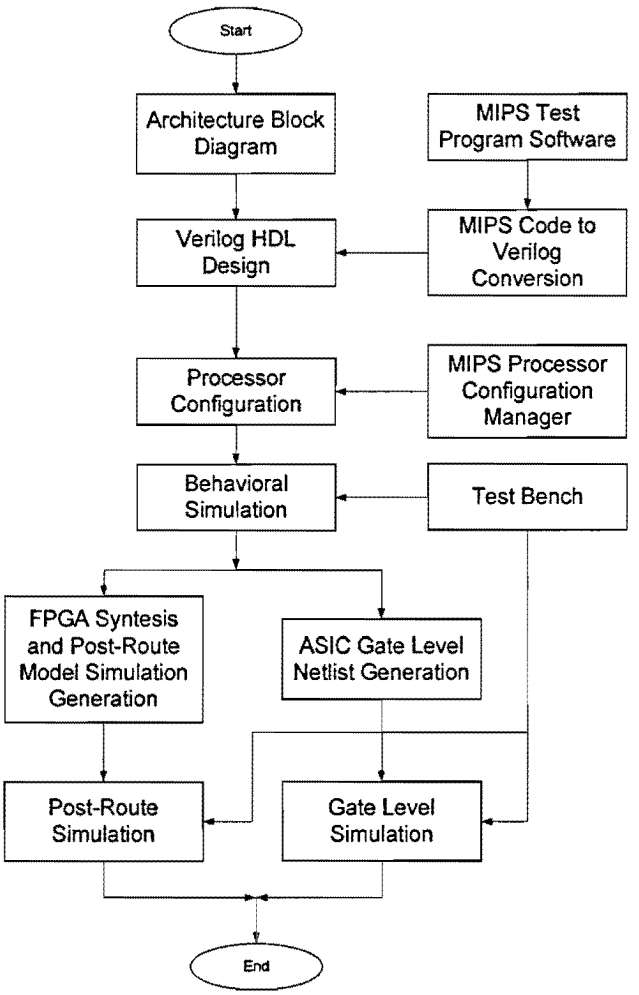


Figure 4.1: The configurable MIPS processor design flow

assign [address0] = data0;

assign [address1] = data1;

...

assign [addressN] = dataN;

address – instruction memory address

data – instruction memory data

Figure 4.2: Format of the pseudo code inserted in the instruction memory module by the proposed custom conversion tool

Verilog HDL Design is the hardware development stage where the HDL description of the proposed processor was developed. After the completion of the processor development this stage is intended for the processor customization beyond the options presented in this project. Also at this stage the file generated by the converter is included in the instruction memory module using the *`include* directive. The software program to be executed by the processor becomes encapsulated in the ROM (Read Only Memory) instruction memory.

Processor Configuration stage is assigned for the selection of the required processor architecture and configuration. All four possible architectures are described in a form of Verilog HDL using optional compilation keys for selection of the particular architecture. The HDL programming style congregates all configuration changes in one configuration file *processor_config.v*. A specific configuration of the processor can be chosen manually by modifying the configuration file or with the help of ***MIPS Processor Configuration Manager***. At the output of the stage the fully developed HDL description of the specific processor configuration is obtained.

At the ***Behavioral Simulation*** stage the fully defined processor design is combined with the corresponding testbench file. The co-design of the processor software program and testbench supports functional verification of the processor hardware. The behavioral simulation is performed by ModelSim tool [68]. The verified processor design can be synthesized on either FPGA or ASIC.

FPGA Synthesis and Post-Route Simulation Model Generation stage produces the bit-stream file for FPGA configuration and model for post-route simulation.

ASIC Gate Level Netlist Generation stage creates the gate level netlist for the further fabrication on silicon and gate level simulation.

At the ***Post-Route Simulation and Gate Level Simulation*** stages the synthesized model/netlist is verified again using the same testbench module interacting with the same build-in program in the instruction memory as the testbench and program applied on the behavioral simulation stage.

4.3 FPGA Implementation

The implementation of the proposed configurable processor on the FPGA platform is performed targeting devices from two major vendors – Altera and Xilinx where various configurations of the processor are fit into different FPGA device families. In order to compare the implemented configurations, the chip that is capable of incorporating the largest number of possible configurations is chosen.

The base design is developed and verified in Xilinx ISE environment. The Altera Quartus II is used as an alternative for portability verification of the base design. The implementation of the Xilinx-oriented features (i.e. FPGA optimization) of the design on the Altera platform leads to a significant difference in the processor area and clock speed. The obtained results may facilitate the selection of the optimal platform for the implementation of any application targeting ASIP with specific timing and area requirements.

4.3.1 Project Files

Figure 4.3 illustrates the hierarchical structure of the top module *mips_dlx* module for the five-stage pipelined processor configuration in Xilinx ISE project view window. Each design module is defined in separate file. The description of each project file is shown in Table 4.2. The structure of the design comprises two levels of hierarchy. The top level is represented by the module *mips_dlx* that contains the description of interconnects and instantiations of the data path modules. The second level of the hierarchy constitutes modules containing the HDL designs of the major data path components described in §3.1. The verification process introduces an additional level of the hierarchy. The testbench includes the *mips_dlx* module as a UUT (Unit Under Test). This three-level structure is used only for the purpose of verification, and it is not synthesizable.

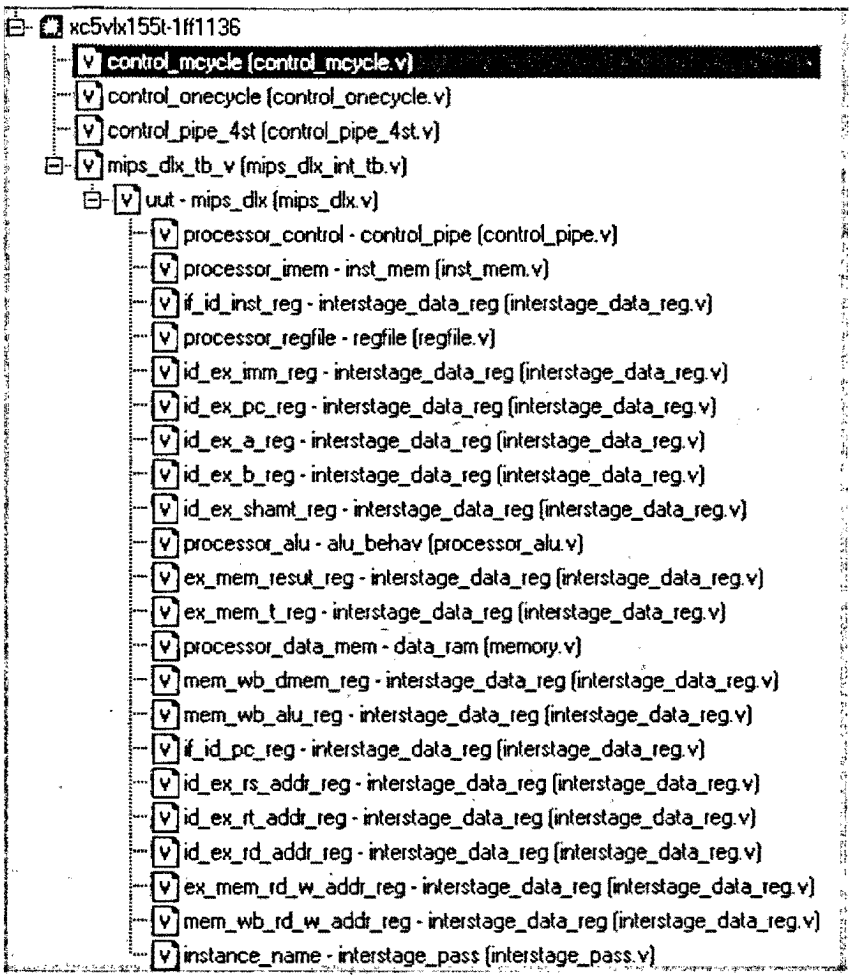


Figure 4.3 : MIPS_DLX project modules hierarchy.

Table 4.2: MIPS_DLX project files description

File Name	Description
mips_dlx.v	Main project file. Defines all interconnects between processor components, declaration of the *mips_dlx* module.
Processor_control.v	Comprises the module of the control unit for the 5-stage pipelined processor configuration, declaration of *mips_dlx* module, and declaration of the *control_pipe* module.
Processor_imem.v	Comprises the processor instruction memory module

interstage_data_reg.v	Comprises the parameterizable module of the pipeline inter-stage register, declaration of the *interstage_data_reg* module.
Processor_regfile.v	Comprises module of the processor register file, declaration of the *regfile* module.
Processor_alu.v	Comprises the processor ALU module, declaration of the *alu_behav* module.
Memory.v	Comprises the processor data memory module, declaration of the *data_ram* module.
Control_mcycle.v	Comprises the module of the control unit for the multi-cycle unpipelined processor architecture, declaration of *control_mcycle* module.
Control_onecycle.v	Comprises the module of the control unit for the one-cycle unpipelined processor architecture, declaration of the *control_onecycle* module.
Control_pipe_4st.v	Comprises module of the control unit for 4-stage pipelined processor architecture, declaration of the *control_pipe_4st* module.
Interstage_pass.v	Comprises the module of the dummy pass inter-stage register, declaration of the *interstage_pass* module.
Processor_config.v	Comprises definitions and configuration control macros
auto_config_part.v	The automatically generated part of the configuration file. Included in *processor_config.v* by *`include* directive
utils.v	Comprises functions used in the design but not available in Verilog
Ext_Imm.v	Comprises the function *ExtImm* of the immediate operand extension.
mips_dlx_int_tb.v	Comprises the verification testbench. Not synthesizable.

4.3.2 Architecture

The graphical symbol of the proposed configurable processor is shown in Figure 4.4. The figure reflects the configurations with 32-bit I/O interface and 10-bit address of I/O memory space. The symbol represents all possible architectures and configurations and depends only on the configuration of I/O interface which affects the bit-width of the address and data lines. The description of the input/output signals of the configurable processor is shown in Table 4.3.

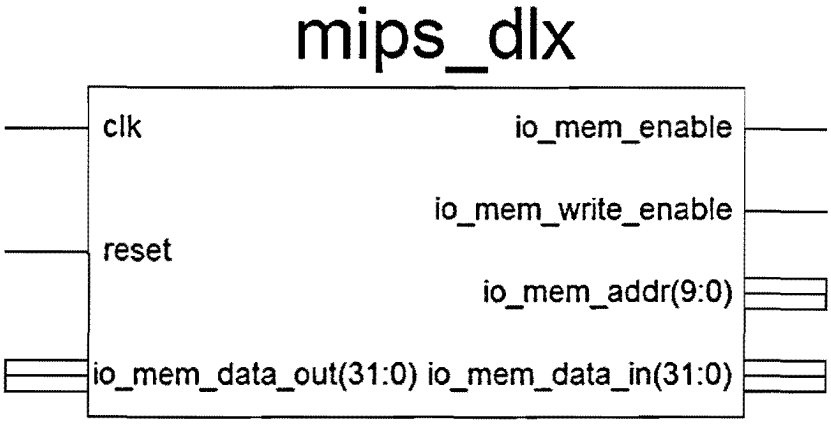


Figure 4.4: 32-bit MIPS processor module

Table 4.3: MIPS interface signals

Signal Name	Direction	Description
clk	In	Main clock signal
reset	In	Global reset signal
io_mem_data_out	In	Data signals of I/O interface. Inputs for MIPS processor, outputs for I/O memory components. This signal is used for entering data into the processor
io_mem_enable	Out	Enable access to I/O memory interface
io_mem_write_enable	Out	Write enable to I/O memory interface
io_mem_addr	Out	Address signals of I/O memory
io_mem_data_out	Out	Data signals of I/O interface. Outputs for MIPS processor, inputs for I/O memory components. This signal is used for processor data outputting.

4.3.3 BRAM Optimization

The implementation of the processor with BRAM optimization is verified on the synthesis stage. In case of implementation of the instruction and data memories in BRAM the Xilinx synthesis tool reports about implementation of the instruction and data memory as a BRAM and absorption of the implied address register into BRAM.

If the design is implemented without the FPGA optimization option the synthesis tool reports about implementation of instruction and data memory on LUTs due to the asynchronous read of memory. The excerpts from the synthesis reports for BRAM and LUTs implementations are shown in Appendix C.2

The synthesis report also contains information about the number of BRAM blocks used in the design. If FPGA optimization is not used, the synthesis tool does not report any BRAM utilization. See the summary reports in Appendix C.1.

4.3.4 Timing Constraints

The only constraint applied for the processor core is the clock speed. The maximum speed of the designed processor depends on the chosen configuration. In order to estimate the maximum speed, the clock constraint is changed according to the configuration. The interactive technique is used for the optimal constraint retrieving. The initial constraint is obtained from the synthesis report which defines the possible maximum speed without actual routing. The initial constraint is changed and tried until synthesis and routing fail. If the clock constraint is set with a big slack, the synthesis tool does not generate a design with maximum possible speed. If the clock constraint is too tight, the implementation fails on the mapping stage. The constraint for which the tool generates the best achievable clock period can have negative and positive slack. Timing constraints for different configurations are shown in Table 4.4. The one-cycle processor variant does not have the EXE/MEM register holding the memory address. Consequently, there is no register to be accommodated in BRAM. Therefore, the implementation of the one-cycle variant with BRAM optimization is not performed.

Table 4.4: Clock period timing constraints (ns)

Bit Width	Instr. Set	Architecture							
		5-stage		4-stage		Multi-Cycle		One-Cycle	
		no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM
256	Full	25	18	25	25	15	16	25	N/A
	Reduced	14	14	21	20	16	15	23	N/A
128	Full	13	12	18	16	14	14	20	N/A
	Reduced	11	11	14	14	14	14	18	N/A
64	Full	11	10	16	16	11	11	18	N/A
	Reduced	10	10	14	14	10	10	16	N/A
32	Full	10	9	10	13	10	10	16	N/A
	Reduced	9	12	11	10	9	9	14	N/A
16	Full	10	10	11	12	9	9	16	N/A
	Reduced	8.9	10	11	10	9	9	12	N/A

4.3.5 Xilinx Platform Implementation

For Xilinx FPGA platform implementation, Virtex-5 FPGA device is selected with following specifications:

Device: XC5VLX50

Package: FF324

Speed grade: -1

The essential FPGA [69] characteristics that affect the maximum clock speed and occupied area of the implemented processor are:

- 550 MHz max clock
- 7200 slices
- 28800 LUTs

Table 4.5 shows the best achievable clock speed for the set of the processor configurations implemented into Xilinx FPGA.

Table 4.5: Maximum clock speed (MHz) of the processor configurations implemented in Xilinx FPGA

Bit Width	Instr. Set	Architecture							
		5-stage		4-stage		Multi-Cycle		One-Cycle	
		no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM
256	Full	44	56	40	40	51	52	40	N/A
	Reduced	71	72	46	50	63	60	43	N/A
128	Full	83	84	56	55	71	69	50	N/A
	Reduced	90	91	65	72	83	72	56	N/A
64	Full	91	100	63	64	91	81	56	N/A
	Reduced	101	100	72	72	100	94	63	N/A
32	Full	100	111	73	77	100	86	63	N/A
	Reduced	112	111	91	96	112	109	73	N/A
16	Full	100	100	86	83	107	102	64	N/A
	Reduced	113	100	92	95	112	111	83	N/A

The utilization of the FPGA resources used for the implementation of various processor configurations is shown in Table 4.6. The sizes of instruction and data memories are chosen equal for all configurations as shown below:

- Instruction memory size – 1024 words
- Data memory size – 1024 words

The detailed implementation reports for the selected processor configurations are shown in Appendix C.

In order to verify the extremities, the 512-bit 5-stage pipelined processor configuration with the full instruction set support has been implemented and verified. The implementation results are shown in Table 4.7. For this implementation the selection of the lager FPGA device is required. The chosen device is XC5VLX155 with 97280 LUTs available.

Table 4.6: Xilinx FPGA resources (LUTs) used for the implementation of the different processor configurations

Bit Width	Instr. Set	Architecture							
		5-stage		4-stage		Multi-Cycle		One-Cycle	
		no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM
256	Full	9927	8858	9761	8720	8850	7845	8956	N/A
	Reduced	5073	4004	4960	3894	3896	2831	3907	N/A
128	Full	4757	4237	4657	4114	4196	3609	4271	N/A
	Reduced	2732	2179	2570	2018	2008	1457	2015	N/A
64	Full	2541	2260	2348	2048	2081	1821	2099	N/A
	Reduced	1444	1147	1359	1061	1062	767	1132	N/A
32	Full	1369	1203	1242	1069	1106	935	1142	N/A
	Reduced	811	642	770	601	589	422	648	N/A
16	Full	734	647	654	569	570	466	599	N/A
	Reduced	507	403	469	364	361	257	419	N/A

Table 4.7: Implementation results of 512-bit 5-stages pipelined processor configuration

Parameters	no BRAM	BRAM
Area	20193 LUTs	18106 LUTs
Max clock speed	32.8 MHz	42.4 MHz
Clock constraint	45 ns	20 ns

4.3.6 Altera Platform Implementation

In order to compare the Altera and Xilinx implementations, the compatible Altera FPGA is chosen for the processor implementation. The Altera and Xilinx FPGAs are different by many aspects including a structure of the base elements. The match for Virtex-5 is Stratix III family fabricated with utilization of the same 65 nm silicon technology. The base element of Xilinx technology is LUT and for Altera it is the Adaptive Logic Module (ALM). A comparative ratio (i.e. ratio of the LUT quantity to the quantity of ALMS required for the

implementation of the same design) of ALM vs. LUT is 1.8x by Altera sources [70] and 1.2x by Xilinx sources [71]. The compatibility of the families used in the project is based on ~1.5x practical ratio. Therefore, the closest available Altera analog of Xilinx XC5VLX50 has the following parameters:

Device: EP3SL70

Package: F780

Speed grade: 4L

Capacity: 27000 ALMs

Max internal clock speed: 600 MHz

The set of processor configurations implemented in Altera FPGA is similar to the set of Xilinx implementations. Table 4.8 shows the maximum clock speed of the implementation in Altera FPGA.

Table 4.8: Maximum clock speed (MHz) of the processor implemented in Altera FPGA

Bit Width	Instr. Set	Architecture							
		5-stage		4-stage		Multi-Cycle		One-Cycle	
		no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM
256	Full	51.78	49.25	27.16	27.07	39.72	38.56	X	N/A
	Reduced	48.01	49.49	36.98	38.1	45.28	43.57	X	N/A
128	Full	72.11	68.95	46.74	46.19	63.69	58.09	X	N/A
	Reduced	74.83	73.37	54.95	56.75	64.21	63.38	47.44	N/A
64	Full	85.14	85.32	62.28	60.38	77.63	66.22	49.63	N/A
	Reduced	86.22	94.67	69.81	69.57	90.59	73.06	52.25	N/A
32	Full	96.93	96.41	74.42	73.69	89.35	83.52	54.97	N/A
	Reduced	100.02	98.01	81.18	76.51	98.82	94.66	60.31	N/A
16	Full	102.03	104.64	83.17	81.0	107.38	93.39	61.4	N/A
	Reduced	119.52	104.01	85.02	87.57	121.68	98.92	66.42	N/A

The time analysis for the three top configurations of the one-cycle architecture is unavailable due to unsuccessful fitting of the design into the chosen FPGA device. The size of the design

exceeds the fitting capability (27000 ALMs) of the chip. Nevertheless, the synthesis tool produces the amount of the required hardware resources for the implementation of oversized configurations. Table 4.9 shows the FPGA hardware resources required for the implementation of the selected processor configurations. The numbers of ALMs shown in the shaded cells indicate the unsuccessful implementations.

Table 4.9: Altera FPGA resources (ALMs) used for the implementation of the different processor configurations

Bit Width	Instr. Set	Architecture							
		5-stage		4-stage		Multi-Cycle		One-Cycle	
		no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM
256	Full	6066	6066	6278	6351	5511	5557	40136	N/A
	Reduced	3127	3281	2309	2335	2399	2307	40017	N/A
128	Full	2993	2934	2903	2915	3195	2661	31477	N/A
	Reduced	1582	1572	1116	1114	1404	1415	23417	N/A
64	Full	1435	1385	1665	1531	1453	1474	12014	N/A
	Reduced	806	754	732	641	757	641	11298	N/A
32	Full	739	745	702	647	672	654	6166	N/A
	Reduced	469	485	428	455	392	382	5809	N/A
16	Full	396	425	219	461	343	383	4266	N/A
	Reduced	290	302	275	312	244	260	4138	N/A

4.4 ASIC Implementation

In order to verify portability of the proposed design, the selected processor configurations were compiled using TSMC 0.18 μm technology process. The synthesis of the considered processor is performed by CMC recommendations for the Digital IC Design Flow [72] [73]. The synthesis tool Synopsys Design Analyzer was used to obtain the gate level netlist. The compilation is performed under the control of the script shown in Appendix D. The synthesized set has a reduced number of implemented configurations compared to FPGA

implementations. Table 4.10 contains the data about maximum clock speed of the selected processor configurations implemented using the technology process. The exhaustive implementation of all possible configurations requires unreasonable amount of time. Therefore, it is not practical to implement a complete configuration set. The proof of the design portability and acquisition of the data for a comparative analysis can be achieved with a lower number of implementations.

Table 4.10: Maximum clock speed (MHz) of the processor configurations implemented using 0.18 μm technology process

Bit Width	Instr. Set	Architecture			
		5-stage	4-stage	Multi-Cycle	One-Cycle
256	Full	165	100	134	87
	Reduced	N/A	N/A	N/A	N/A
128	Full	N/A	N/A	N/A	N/A
	Reduced	N/A	N/A	N/A	N/A
64	Full	N/A	N/A	N/A	N/A
	Reduced	N/A	N/A	N/A	N/A
32	Full	200	158	168	139
	Reduced	200	161	176	116
16	Full	200	186	182	148
	Reduced	200	174	204	156

The same 4 ns clock timing constraint was applied for all configurations. In all cases the constraint was violated with a different negative slack. The minimum achievable clock period is calculated as the sum of the applied constraint and reported slack.

The important data retrieved from the compilation report includes the total cell area occupied by the implemented design. This data is shown in Table 4.11 for the implemented processor configurations. The area units are relative and specific for the Artisan tpz973gwc cell library used for the synthesis. For Artisan library the units are μm^2.

The implementation of the design using the technology process was performed successfully without any modifications to the Verilog code used for the FPGA implementations. Therefore, the portability of the proposed design is fully verified.

Table 4.11: Total cell area (μm^2) occupied by the processor configurations implemented using 0.18 µm technology process

Bit Width	Instr. Set	Architecture			
		5-stage	4-stage	Multi-Cycle	One-Cycle
256	Full	8309477	8241904	8039663	6732384
	Reduced	N/A	N/A	N/A	N/A
128	Full	N/A	N/A	N/A	N/A
	Reduced	N/A	N/A	N/A	N/A
64	Full	N/A	N/A	N/A	N/A
	Reduced	N/A	N/A	N/A	N/A
32	Full	1065715	1075418	1044842	947651
	Reduced	1032431	1046299	1022721	886046
16	Full	540470	556243	526176	460024
	Reduced	525973	539625	523056	458357

4.5 Demo Platform Design and Implementation

The purpose of the demo platform design is to show a practical utilization of the configurable processor core. Through the memory mapped I/O interface connected to the user interface the selected processor core obtains input data, processes the collected data, and shows the results on the display.

4.5.1 Hardware Platform Description

The hardware setup used to implement the processor design is *Spartan-3E FPGA Starter Kit Board* [74]. This development board incorporates Xilinx Spartan-3 FPGA chip XC3S500E and variety of peripherals. The block diagram of the implemented setup is shown in Figure 4.5. The Rotary Shaft Encoder is chosen as an input device for the demo design. A photo of the encoder is shown in Figure 4.6.

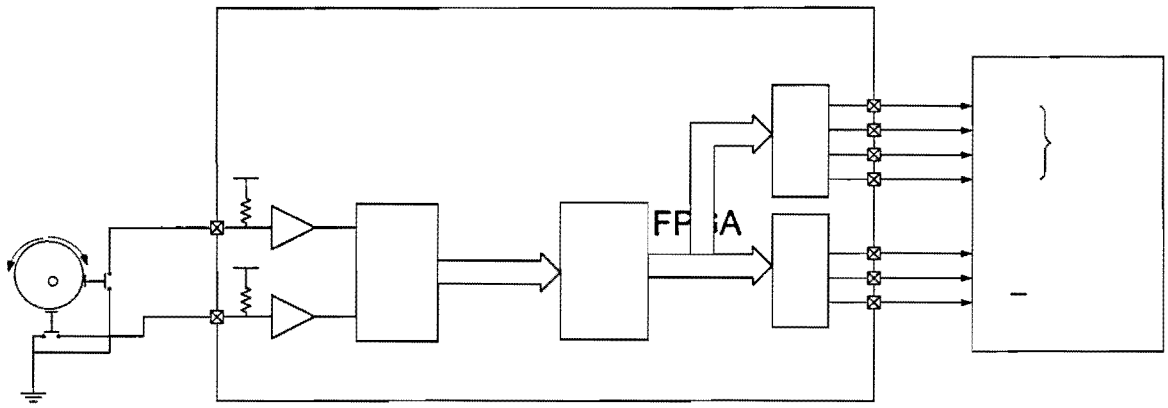


Figure 4.5: Demo platform block diagram

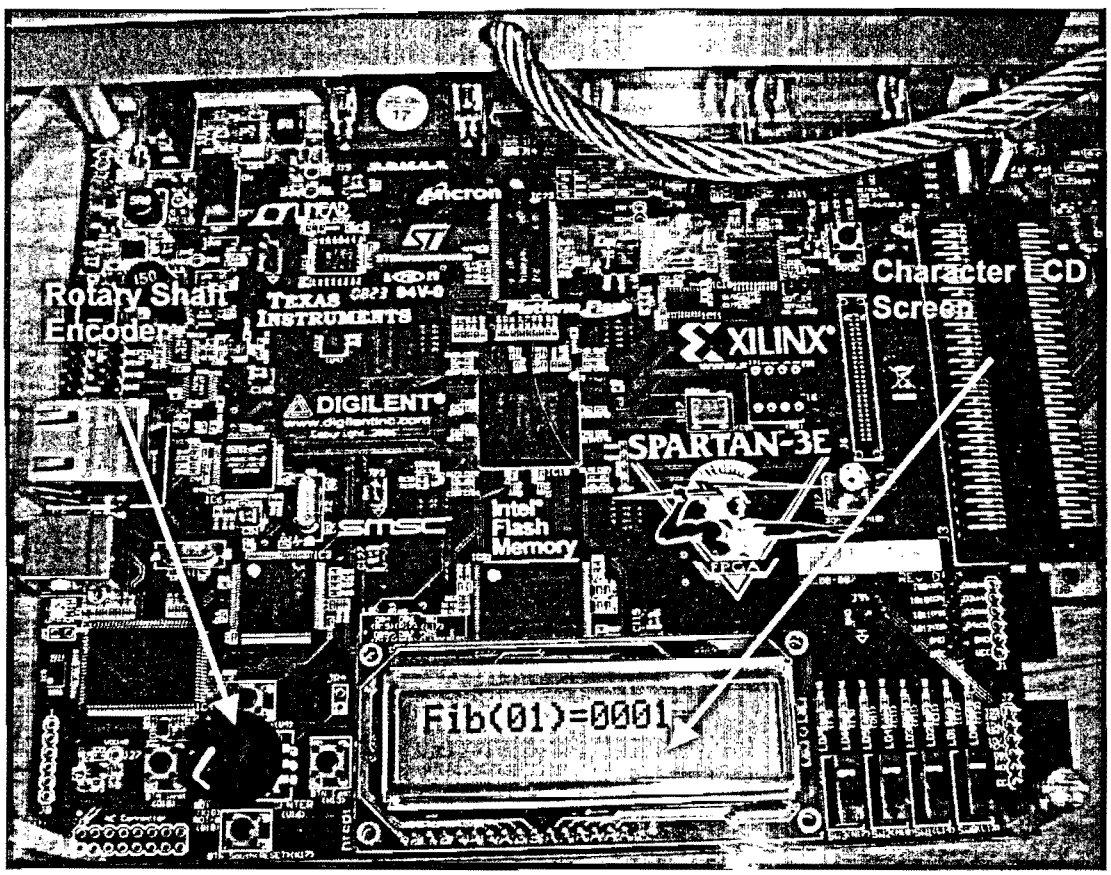


Figure 4.6: Spartan-3E startup kit FPGA board

Depending on a direction, rotation increases or decreases a default number read by the processor core. The default number is 5. The input number is used to calculate Fibonacci number [75]. The input data indicates the position in the sequence of Fibonacci numbers. The calculated result is shown on the character LCD display in the following form:

Fib(N)=M,

N – order of Fibonacci number

M – calculated Fibonacci number

Rotation of the shaft causes a change of both *N* and *M* numbers dynamically. The numbers are represented in hexadecimal radix. The number of characters allocated for *N* and *N* are two and four respectively.

4.5.2 Processor Core Configuration

The demo design utilizes the processor core with following configuration:

- Five stages pipelined architecture
- 32-bit data path
- Shift commands support
- Set Than Less commands support
- Instruction memory size – 1024 words
- Instruction memory count step – 1 word
- Data memory size – 1024 words
- Data memory count step – 4 bytes
- Upper limit of data memory address space – h8fffffff
- Lower limit of data memory address space – h10000000
- I/O memory allocated size – 1024 words
- I/O memory data bit-width – 32 bits
- Upper limit of I/O memory address space – h00400000
- Lower limit of I/O memory address space – h0
- Program counter bit-width – 32 bits
- Reset address – h80020000
- Stack pointer initial data – h800bc000

The limits of data memory, I/O memory, initialization values for program counter and stack pointer are chosen to be compatible with MIPS simulation model in the simulator MipsIt [76].

4.5.3 Demo Platform Interface Design

The signals from Rotary Shaft encoder are processed in the Decoder module. The module performs debouncing of input signals, detection of a shaft turn event and direction of a turn. The HDL design of the Decoder module is based on Xilinx reference design [77]. The direction and turn event change the initial number for Fibonacci calculation. The register that storing the number is connected to I/O memory interface. The decoding in I/O memory address space is supported by a dedicated decoder.

In addition to Fibonacci number calculation, the processor core runs a custom driver for LCD screen. The driver software generates data and control signals according to the specification of LCD screen interface. LCD signals are written to registers mapped to the I/O memory address space. The mapping of LCD signals and Fibonacci initial number are shown in Table 4.12.

Table 4.12: Mapping of the Demo design signals in I/O memory address space

Signal Name	MIPS I/O Address	Bit range	Direction	Description
fib_in_number	0x00000008	31:0	Input	Order of Fibonacci number
lcd_output_data	0x00000014	7:4	Output	LCD data to be written
lcd_drive	0x00000014	3	Output	Enable signal for whole LCD interface
lcd_rs	0x00000014	2	Output	Command/Data select signal
lcd_rw_control	0x00000014	1	Output	Read/Write control signal
lcd_e	0x00000014	0	Output	Read/Write enable signal

4.5.4 Software/Hardware Co-Design

Timing and electrical specification of LCD interface requires an algorithm intensive control capability. That capability is supported in the software running by the processor core. The following functions are performed in the software:

- LCD power-on initialization
- LCD configuration

- Writing data to LCD

Writing data to LCD in the loop sustains a dynamic refreshment of the visualized information on the screen when the shaft is rotated. All timing delays for the LCD interface are implemented in software loops. Pre-processing of the rotary shaft encoder signals is realized in hardware in a form of HDL code. The code also has a provision for the control of LEDs and reading of buttons available on the board. This provision is used for development and debugging of Demo software/hardware co-design.

The driver software is written in C++ language. The limitations of the implemented processor core are reflected in the programming technique. For instance, only unsigned integer variables are used in the code. The compiled code is examined in order to eliminate unsupported commands and variables. The configuration of the processor core is optimized to support the demo program. The optimization includes a choice of the supported command subset, size of instruction and data memory. The full code of the demo design program is shown in Appendix A.

4.5.5 Demo Design Implementation

The choice of FPGA device for the demo design implementation is defined by the chip installed on the development board. The parameters of FPGA are shown below:

Device: Xilinx Spartan-3 XC3S500E

Package: FG320

Speed grade: -5

Capacity: 9312 LUTs

The time constraint is determined by 50 MHz clock available on the development board and connected to the FPGA input. All other constraints are related to the pin assignments. The full set of the demo design constraints is shown in Appendix E. The synthesis of the demo design was successfully performed with the following results:

Number of used LUTs: 4888

Number of used I/O blocks: 30

Minimum period: 19.971ns (Maximum frequency: 50.073MHz)

Device utilization: 52%

Since the FPGA optimization option is not used, the synthesis report does not have the information about BRAM utilization. The full implementation report is shown in Appendix E.

4.6 Summary

This chapter showed the successful implementation of the proposed configurable MIPS-like processor on the two competitive FPGA platforms and ASIC implementation using 0.18μm TSMC technology process. The applied design flow and tools used on every design stage were described in details. The justification of the selected implementation platforms was based on the project requirements and FPGA devices compatibility. As a result of implementation using standard FPGA/ASIC tools, the data about hardware resources utilization and maximum possible clock speed was obtained for the selected configurations. The demo design was successfully implemented in the selected development platform meeting all design constraints.

Chapter 5

Design Verification

This chapter discusses the verification scheme of the proposed microprocessor architecture. The process is achieved in two stages. The scheme is conducted at two stages of the design flow. The first stage is at the HDL development level of the design where a testbench is built to verify the design's HDL code. The second is at post- implementation level, where the implemented design on the target platform is tested. At the HDL level, the design's code is tuned until the correct behavior of the design is reached. The post-implementation verification depends on the target platform. For example, in case of FPGA, the main objective of this verification is the post-place & pout simulation model. However, in case of ASIC implementation, the objective is the gate level netlist.

The design of a testbench must cover as much as possible functionality of the simulated processor. The large number of the verified processor configurations and implementation platforms appeals to the stringent requirements for the testbench. In order to be able to verify numerous variants, the testbench output shall simply state whether the tested design failed or passed. The functional verification of the design is performed for the selected processor configurations. When all bugs are eliminated, the set of selected configurations is verified with post-route simulation. Table 5.1 shows the subset of the processor configurations which were simulated where the simulated configurations are referred to by x sign. The shown selection was driven by the following reasons:

- BRAM optimization is independent of the instruction set and bit-width. It may be verified once per chosen architecture

- Different from the standard 32-bit processor configuration may be verified only at extremities. In the considered case, extremities are 16-bit and 256-bit
- Reduced instruction set is independent of bit-width. It may be verified once per chosen architecture

Table 5.1: Verification matrix for the processor configurations set

Bit Width	Instr. Set	Architecture							
		5-stage		4-stage		Multi-Cycle		One-Cycle	
		no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM	no BRAM	BRAM
256	Full	X	X	X	X	X	X	X	
	Reduced								
128	Full								
	Reduced								
64	Full	X		X		X		X	
	Reduced								
32	Full	X	X	X	X	X	X	X	
	Reduced	X		X		X		X	
16	Full	X		X		X		X	
	Reduced								

5.1 Testbench Design

The testbench is created to verify the functionality of the developed processor. The recursive algorithm of calculation of the Fibonacci numbers is widely used as an evaluation example of the functionality of the MIPS processor [78] [79]. The Fibonacci calculation algorithm can be implemented into a compact program due to its recursive nature. Translated into machine codes it utilizes most of the proposed instruction set including the complicated instructions (e. g. function calls, branches, jumps etc.). Moreover, the translated program contains all RAW hazard scenarios possible in the proposed pipelined architecture. The

external data required for the computation is minimal. The described set of properties of the Fibonacci algorithm makes it very practical for validation of the proposed processor.

5.1.1 Fibonacci Number Test Program

The Fibonacci number sequence was introduced by the medieval mathematician Fibonacci (Leonardo Pisano) as a solution to the logical puzzle. Later it was found that the sequence reflects many processes in nature [75]. The Fibonacci numbers comprise the following sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765 ...

The C-style description of the recursive function for the calculation of the Fibonacci numbers is shown below.

```
int fib(int n) {
   if (n==0) {return 1;}
   if (n==1) {return 1;}
   return (fib(n-1) + fib(n-2));
   {
```

The assembler code obtained after translation is augmented manually for communication with the high level Verilog testbench. The assembler code of the test program is shown below.

```
lw $29, 4($0)          ;  lw $sp, 4($zero) //load from I/O mem
lw $4, 8($0)           ;  lw $a0, 8($zero) //load from I/O mem
addi $29, $29, -12     ;  addi $sp, $sp, -12
fib:
sw $31, 8($29)         ;  sw $ra, 8($sp)
sw $16, 4($29)         ;  sw $s0, 4($sp)
addi $2, $0, 1         ;  addi $v0, $zero, 1
beq $4, $0, 52         ;  beq $a0, $zero, fin
addi $8, $0, 1         ;  addi $t0, $zero, 1
beq $4, $8, 40         ;  beq $a0, $t0, fin
nop                    ;  nop (delay slot)
addi $4, $4, -1        ;  addi $a0, $a0, -1
sw $4, 0($29)          ;  sw $a0, 0($sp)
jal 0x00400008 [fib];  jal fib
lw $4, 0($29)          ;  lw $a0, 0($sp)
addi $4, $4, -1        ;  addi $a0, $a0, -1
add $16, $2, $0        ;  add $s0, $v0, $zero
jal 0x00400008 [fib];  jal fib
add $2, $2, $16        ;  add $v0, $v0, $s0
```

```
sw $2, 12($0)          ;  sw $v0,12($zero)//store to I/O mem
lw $16, 4($29)         ;  lw $s0, 4($sp)
fin:
lw $31, 8($29)         ;  lw $ra, 8($sp)
addi $29, $29, 12      ;  addi $sp, $sp, 12
sw $29, 16($0)         ;  sw $sp,16($zero)//store to I/O mem
jr $31                 ;  jr $ra
nop                    ;  nop
```

The additional instructions have been inserted at the beginning of the program to retrieve the initial stack value and the order of calculated Fibonacci number. The program performs calculation starting from the value in *r4* and places the calculated result in *r2*. Moreover, the program frequently sends values of the stack and *r4* to I/O interface. This method allows monitoring of the values by an external testbench.

The translated machine codes are contained in a separated file which is stored in the instruction ROM. Appendix F shows two variants of the developed test program. The variation reflects the differences of the processor architectures. The applicability of the variants is shown below:

- Delay slot with reordering – pipelined architecture
- No delay slot – unpipelined architecture

If reordering is not used for the pipelined architecture, the additional *nop* instructions have to be placed instead of reordering. This option can be chosen in the compiler.

A recursive nature of the test program allows testing of all possible data and control hazards for the implemented instruction set.

5.1.2 Verilog Testbench

The organization of the testbench and test program allows them to be applicable at the behavioral and post-rout levels. The testbench does not interact directly with the internal processor registers. Though acceptable for the behavioural simulation, the direct access is not applicable on the post-route level. The testbench *mips_dlx_tb.v* comprises the instance of the tested processor configuration and memory accessible by the testbench. The memory is connected to the processor as I/O memory and mapped to the processor memory address

space. The mapping of the I/O memory is shown in Table 5.2. The testbench also contains the pre-calculated array of Fibonacci numbers.

Table 5.2: Mapping of testbench in I/O memory address space

I/O Memory Address	Direction (processor scope)	Description
1	Input	Initial stack value
2	Input	Order of Fibonacci number
3	Output	Calculated Fibonacci number
4	Output	Current stack value

The test program loads *r4* with the order of the calculated Fibonacci number and *r29* (stack pointer) with the initial value. The program performs calculation continuously updating I/O memory with current values of the stack and Fibonacci number. The simulation finishes when the stack pointer returns back to its initial value. The block diagram of the developed Fibonacci number testbench is shown in Figure 5.1.

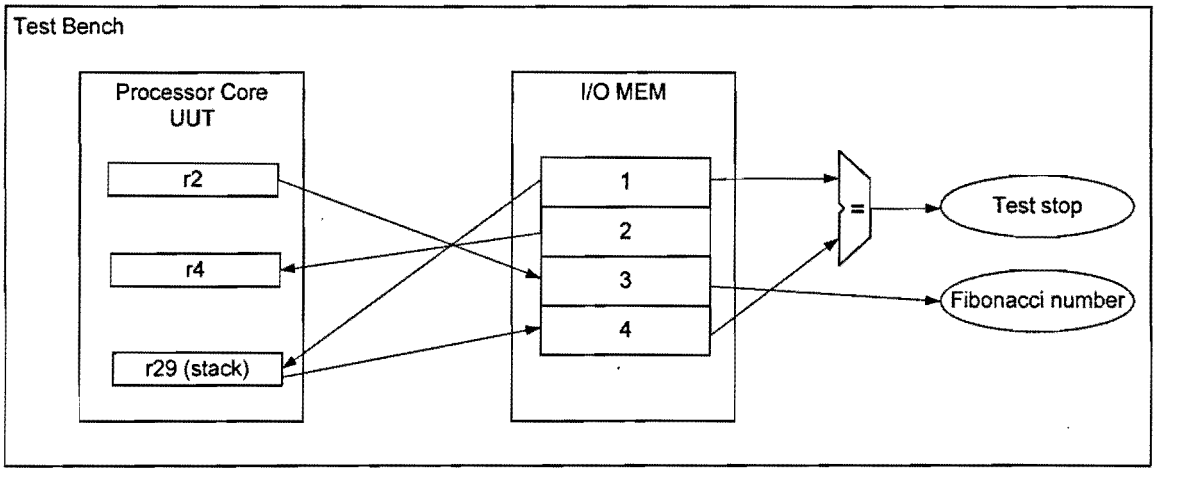


Figure 5.1: Block diagram of the Fibonacci number testbench

The testbench program compares the result contained in I/O memory with the true pre-calculated Fibonacci number and reports whether the test is successfully completed. The examples of the successful and unsuccessful simulation reports are shown in Appendix G. The waveform of the simple case *Fib(3) =3* is also shown in Appendix G. Due to the recursive nature of the algorithm, the calculation of Fibonacci numbers of the higher order requires substantially longer time. For instance, the calculation of *Fib (16) = 1597* requires 52447 machine cycles of the processor with one-cycle architecture. In order to reduce the

verification time, the Fibonacci number calculation case *Fib (5) = 8* is chosen for testing of all implemented configurations. The higher order of calculation does not increase the confidence in verification results but only the execution time and size of the data placed in the stack.

5.2 Pipelined Architecture Verification

The five-stage architecture was simulated and the following message was obtained for all tested configurations of this architecture:

```
# Fibonacci number test SUCCESSFULLY completed, Fib (  5) =  8
# Test finished after        286 machine cycles
```

The same test is performed four-stage architecture and the following message was obtained:

```
# Fibonacci number test SUCCESSFULLY completed, Fib ( 5) =   8
# Test finished after        278 machine cycles
```

The important part of the pipelined architecture testing is verification of the hazards handling. The following sequence of the instructions creates a data hazard:

```
0:      lw $29, 4($0)              //WB stage
1:      lw $4, 8($0)               //MEM stage
2:      addi $29, $29, -12         //EX stage
```

r29 value used on the EX stage by instruction 2 is fetched from the regfile before it is written to the regfile by instruction 0. Therefore *r29* value must be forwarded from the WB stage to EX stage. The waveform in Figure 5.2 shows how the instruction *addi* (address 2) fetched in the IF stage from the instruction memory triggers the change of the forwarding multiplexer in the EX stage selecting ALU operands *A* and *B* from the WB stage (multiplexer address 2). Another case of hazards created by an instruction sequence:

```
                                   //1st hazard //2nd hazard
2:      addi $29, $29, -12         //MEM stage  //WB stage
3:      sw $31, 8($29)             //EX stage   //MEM stage
4:      sw $16, 4($29)             //ID stage   //EX stage
```

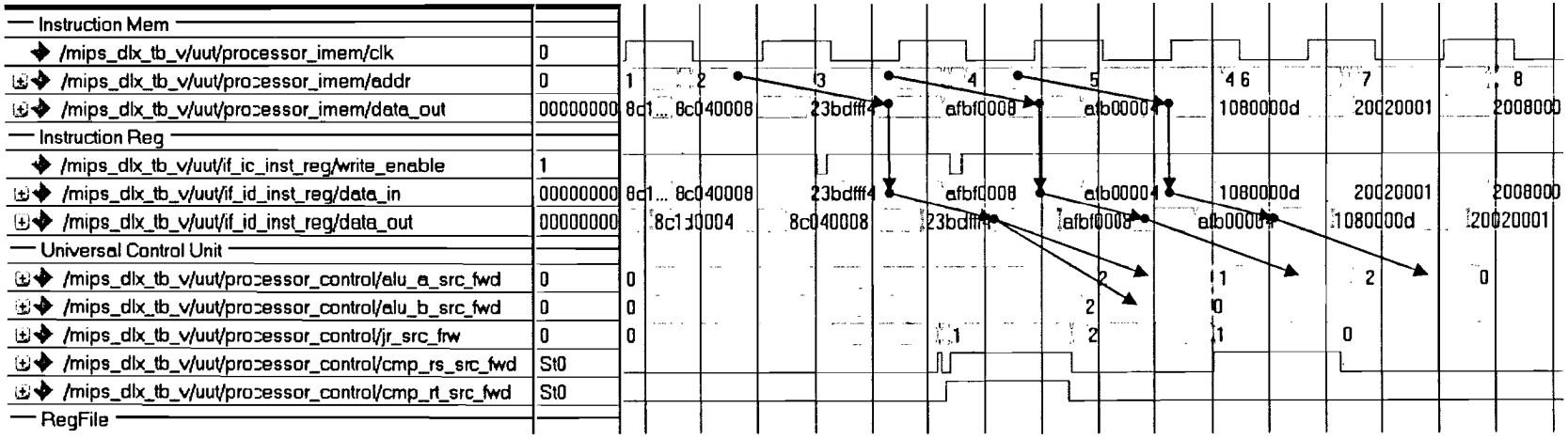


Figure 5.2: Forwarding WB→EX and MEM→ EX in the pipelined architecture (ModelSim waveform)

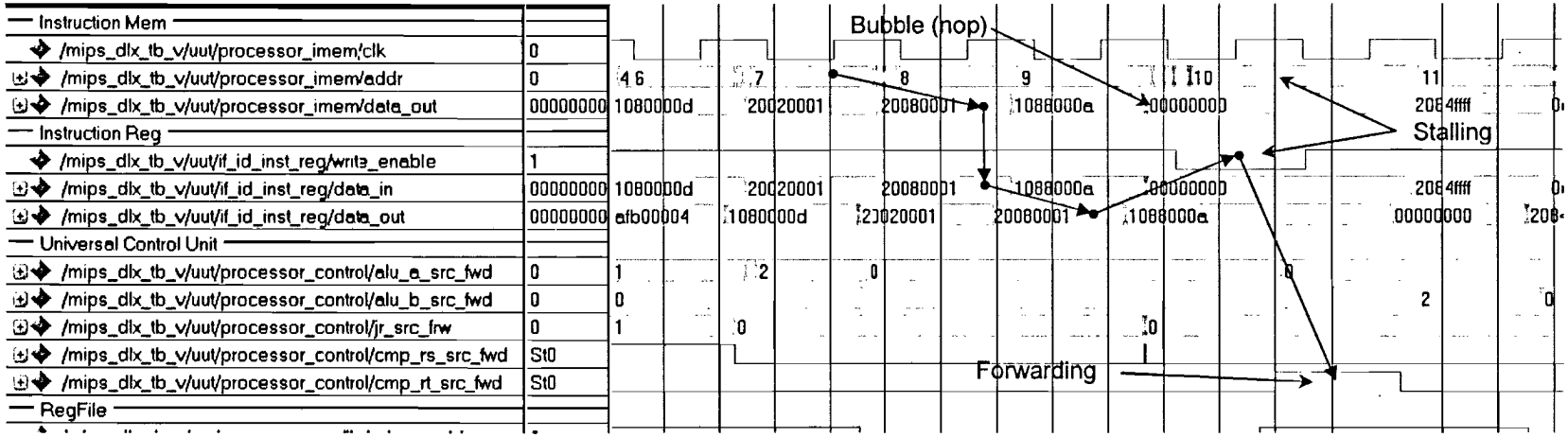


Figure 5.3: Stalling and forwarding MEM→ID in the pipelined architecture (ModelSim waveform)

The shown hazards are similar to the previously described hazard, where *r29* value is used on the EX stage before it becomes available from the regfile. The first hazard is handled by forwarding from the MEM stage and the second hazard is handled by forwarding from WB stage. This forwarding is also shown in Figure 5.2. The instruction *sw* (address 3) triggers the forwarding of the ALU operand *A* from the MEM stage (multiplexer address 1). In the same way the next instruction sw causes the forwarding of the operand A from the WB stage (multiplexer address 2).

Since not all hazards can be handled by forwarding, the verification of stalling is to be verified as well. The following instruction sequence creates the hazard to be handled by stalling and forwarding:

```
7:      addi $8, $0, 1              //EX stage
8:      beq $4, $8, 40              //ID stage
```

The instruction *beq* (address 8) uses *r8* value on the ID stage when the previous instruction *addi* (address 7) that changed *r8* did not yet write it in the regfile. Moreover, the result of addi instruction is not available yet from the EX stage. Therefore, the immediate forwarding cannot help. The stalling for one clock cycle and forwarding from the MEM stage allows handling that type of hazards. Figure 5.3 shows the detailed waveform of the described hazard handling in the simulation of the five-stage architecture with BRAM optimization.

The shown examples of forwarding and stalling do not cover the whole set of RAW hazard handling that are implemented in the proposed processor design. The complete set of verified hazard scenarios handled in the pipelined configurations is as shown in Table 5.3

Table 5.3: Data hazards handled by forwarding and stalling in the pipelined architectures

Pipeline Length	Fibonacci Test	Hazards Handled by Forwarding	Hazards Handled by Stalling
4 stages	`Fib(5)=8`	2	2
5 stages	`Fib(5)=8`	4	5

The Fibonacci number testbench contains all described hazard scenarios. The correct handling of these hazards was simulated and verified. The post-route simulation was successfully performed for the configurations shown in Table 5.1. The only type of hazard possible in the proposed pipelined architecture is the RAW hazard, therefore handling of WAR and WAW hazards is not required.

5.3 Multi-Cycle Design Verification

Due to the significant difference in the control units between multi-cycle and pipelined architectures, the verification of the control unit functionality for the multi-cycle architecture is crucial. The simpler control unit design contains no hazard handling to be verified, though FSM functionality is to be validated. Figure 5.4 shows the selected waveforms of the 32-bit multi-cycle processor running *Fib(3) =3* program. The highlighted waveforms of FSM states (i. e. *state & nextstate*) represent a typical example of FSM operation.

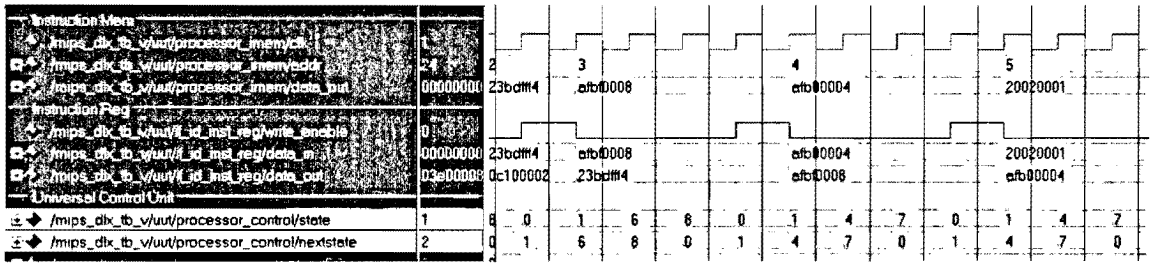


Figure 5.4: Waveform of the multi-cycle architecture simulation

The shown waveforms reflect the following instruction sequence:

```
2:      addi $29, $29, -12          //I-type instruction
3:      sw $31, 8($29)              //MEM-type instruction
```

There is the following correspondence between FSM states shown and actions performed by the control unit:

State 0→1→6→8→0→1→4→7

Action FETCH→DECODE→EXECUTE_IMM_TYPE→WR_BACK→
→FETCH→DECODE→EXECUTE_MEM_TYPE →MEM_ACCESS

Table 5.4: FSM action description

FSM Action	Description
FETCH	An instruction is fetched from the program memory and stored in the instruction register. The program counter is updated with the next instruction address. ALU increments the next instruction address
DECODE	The instruction is decoded. The operands are retrieved from the regfile and stored in ID/EX register. ALU calculates the next instruction address for a possible branch instruction.

EXECUTE_IMM_TYPE	ALU performs an arithmetic/logic operation with the immediate operand
EXECUTE_MEM_TYPE	ALU performs adds a memory address offset calculating a memory access address
MEM_ACCESS	Memory load/store operation
WR_BACK	Writing the result of load operation or ALU calculation in the regfile

Table 5.4 describes major processor actions performed under control of the FSM incorporated in the control unit.

The post-route simulation the testbench *Fib(5) =8* was performed for the multi-cycle architecture configurations shown in Table 5.1 and the following message was reported for all configurations:

```
# Fibonacci number test SUCCESSFULLY completed, Fib (  5) =    8
# Test finished after          964 machine cycles
```

5.4 One-Cycle Design Verification

The simplest by data path and control organization one-cycle architecture was verified for the functionality of the processor design by successful completion of the testbench. The example of the post-route simulation waveforms of the one-cycle processor simulation is shown in Appendix G. The adduced waveforms validate the correct operation of this processor architecture.

The post-route simulation the testbench *Fib(5) =8* was performed for the one-cycle architecture configurations shown in Table 5.1 and the following message was reported for all configurations:

```
# Fibonacci number test SUCCESSFULLY completed, Fib (  5) =    8
# Test finished after          243 machine cycles
```

5.5 Demo Platform Design Verification

The verification of the demo platform design is more complicated than the verification of the processor core itself due to the increased design complexity. The demo design is verified

at behavioral and post-route stages by checking waveforms on the LCD interface lines by means of the software simulator. The timing diagram and data send through the interface complies with LCD screen interface specification. The correctness of the displayed information is checked by comparing waveforms of the data sent to the LCD screen with the expected values.

The final validation is performed by loading the design bit-stream into the target FPGA on the development board and comparing the values displayed on the screen with pre-calculated Fibonacci values. The pictures of the development board running the demo design are shown in Appendix H. All numbers are displayed in the hexadecimal format. The change of the order of Fibonacci number is performed by turning of the rotary shaft. The maximum obtained correct number *Fib(0x17) =0xB520* is limited by the four character positions allocated for the resulting Fibonacci number. The Fibonacci numbers shown on the LCD screen coincide with the pre-calculated values.

5.6 Summary

This chapter outlined the verification process applied for validating the proposed configurable processor. The design of the testbench was described in details. Also, the results of the testbench execution were presented. The verification of the critical design features was outlined and shown in examples presented in this chapter. The practical implementation of the proposed processor was verified on the development board. The obtained testing results concurred with expected values. The proposed design was verified successfully.

Chapter 6

Result Analysis

This chapter analyzes the implementation and verification results obtained in the chapters 4 & 5. The difference between implemented architectures, as it was outlined in chapters four and five, is based on the structural features of a particular design. In this chapter the stated assumptions are supported by the actual data. The comparative analysis of the considered processor configurations illustrates the advantages and disadvantages of the proposed architectures. The conducted analysis may facilitate the selection of the considered processor configuration for the specific application.

6.1 Xilinx FPGA Implementation Evaluation

The performance evaluation of the proposed processor is conducted for corresponding configurations of the different architectures. Figure 6.1 shows the maximum clock speed and occupied hardware resources of the architecture variants of the 32-bit processor implemented in Xilinx FPGA. The chart also compares implementations with *full/reduced* instruction sets and *BRAM/no BRAM* optimization options. The five-stage pipelined architecture shows the highest clock speed. The trade-off for this advantage is the largest number of the FPGA resources required for the implementation. The multi-cycle architecture offers very close clock speed with saving 19% of hardware resources. But simple comparison of the clock speed of the considered architectures would be misleading. However, the throughput can be used as a parameter that reflects more precisely the performance of the architecture. The five-stage architecture has throughput of one instruction-per-cycle while multi-cycle architecture has a variable throughput of 3-5 cycles per instruction. The Fibonacci testbench executed on

the five-stage processor requires 3.4x times less clock cycles than executed on the multi-cycle processor regardless of the bit-width and chosen configuration. Therefore, the multi-cycle architecture is, in fact, the slowest of all considered implementations. The benefit of multi-cycle architecture is the reduced utilization of the hardware resources.

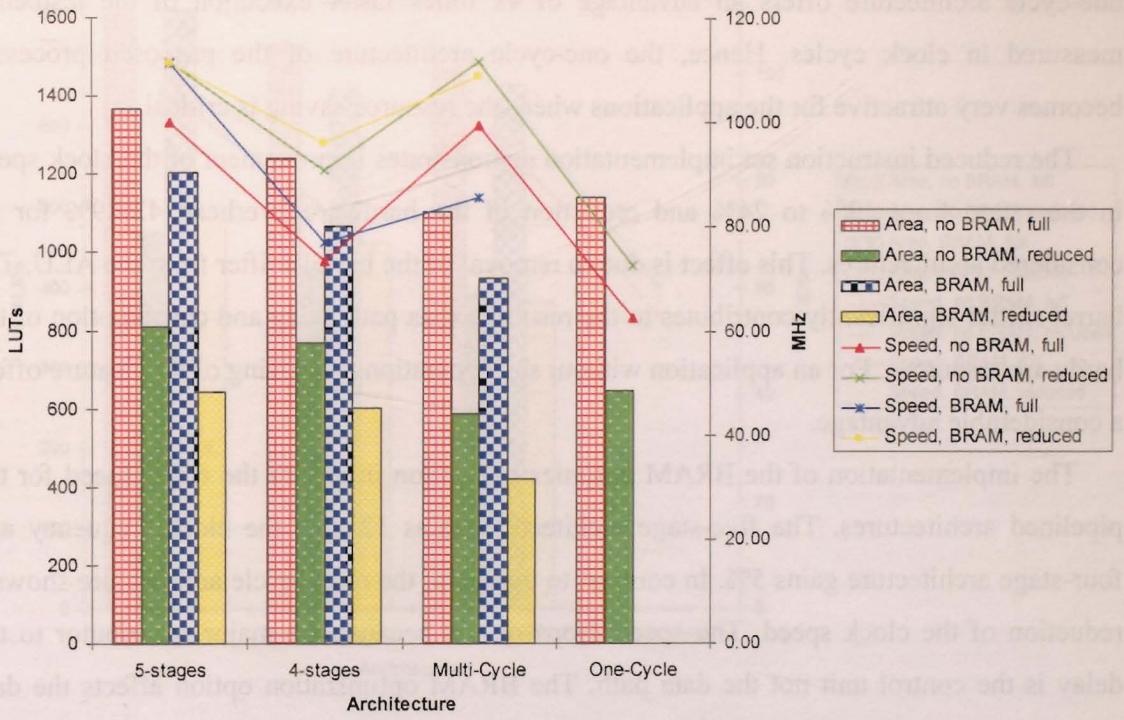


Figure 6.1: Evaluation chart of the architecture variants of 32-bit processor implemented in Xilinx FPGA

The comparative analysis of two pipelined architectures shows that the four-stage architecture consumes 9% less area at the expense of 26% reduction in the clock speed. Though the theoretical throughput is one-instruction-per-cycle for both architectures, the reduced number of hazards handled by stalling causes 3% faster testbench execution measured in clock cycles for the four-stage architecture. Despite the fact that the execution of another testbench may result in better clock cycle performance, the combined speed performance of the four-stage architecture is worse than the five-stage architecture.

The clock speed of the one-cycle architecture is lowest of all. This result is expected and can be explained by the longest asynchronous path of the one-cycle architecture. The difference of 37% between one-cycle and multi-cycle architectures in the clock speed is

considerably less than stated in [12]. This difference can be related to the dissimilarity in the control units where the control unit of the multi-cycle architecture introduces more delay. The control unit of the multi-cycle architecture contributes to the resulting delay significantly more than the control unit of the one-cycle architecture. Having 3% hardware overhead, the one-cycle architecture offers an advantage of 4x times faster execution of the testbench measured in clock cycles. Hence, the one-cycle architecture of the proposed processor becomes very attractive for the applications where the resource saving is critical.

The reduced instruction set implementation demonstrates improvement of the clock speed in the range from 12% to 24% and reduction of the hardware overhead 42-49% for all considered architectures. This effect is due to removal of the barrel shifter from the ALU. The barrel shifter significantly contributes to the resulting data path delay and consumption of the hardware resources. For an application without shift operations removing of this feature offers a considerable advantage.

The implementation of the BRAM optimization option improves the clock speed for the pipelined architectures. The five-stage architecture gains 12% of the clock frequency and four-stage architecture gains 5%. In contrast to that gain, the multi-cycle architecture shows a reduction of the clock speed. The speed drops occurs because the major contributor to the delay is the control unit not the data path. The BRAM optimization option affects the data path only leaving the control unit intact. The additional delay for multi-cycle architecture is caused by the memory control signals which have a longer delay for BRAM.

The distinguishable feature of the BRAM optimization option is saving of general hardware resources (i. e. LUTs) by implementing parts of the design into specialized FPGA blocks (i.e. BRAM). This saving reaches 12-15% for all architectures with the full instruction set and 21-28% for architectures with the reduced instruction set.

The results of the architecture comparative analysis conduced for 32-bit implementations can be extended for the implementations of other bit-widths.

6.2 Altera FPGA Implementation Evaluation

The implementation of the proposed processor variants on the competitive Altera FPGA platform demonstrates results different from the Xilinx implementation. These results are illustrated on Figure 6.2 showing the quantitative relation of the implementation variants of

the 32-bit processor. The FPGA resources for the one-cycle architecture are omitted for better visibility of the graph because the hardware recourses occupied by this type of architecture exceed the closest value by eight times.

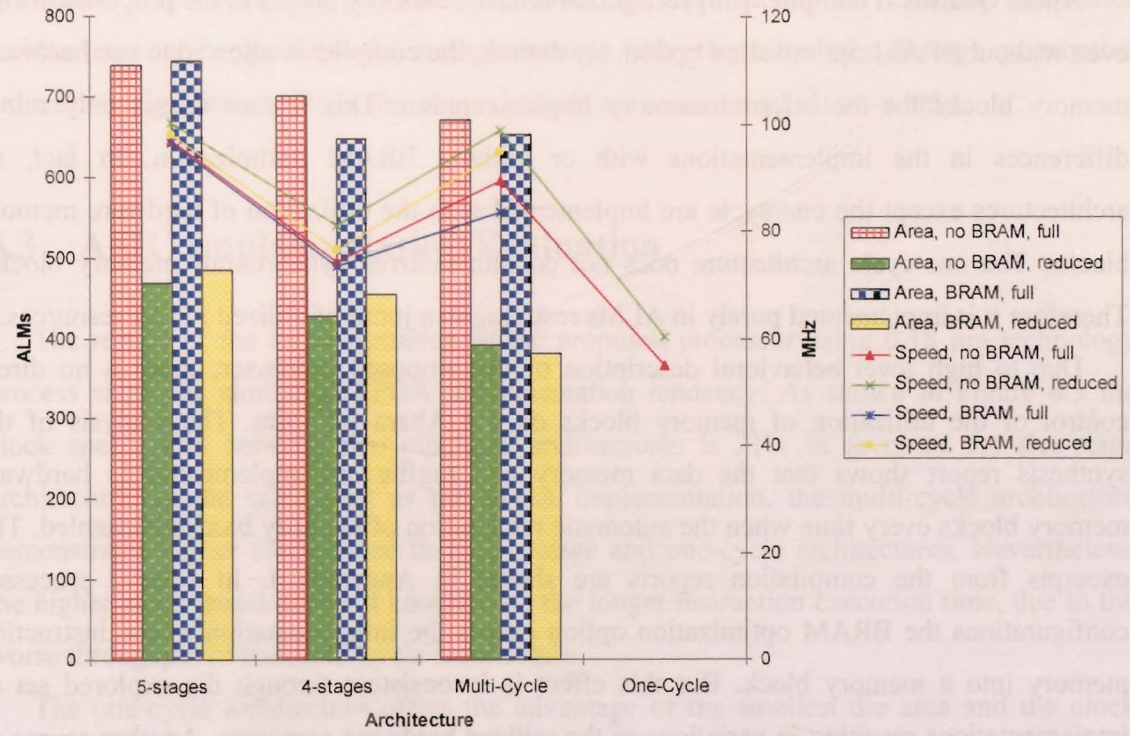


Figure 6.2: Evaluation chart of the architecture variants of 32-bit processor implemented in Altera FPGA

The comparative analysis of the clock speed between the architectures shows the results close to Xilinx implementation. The five-stage architecture is 23% faster than the four-stage architecture. The multi-cycle architecture demonstrates the similar to five-stage clock speed while showing the worse testbench execution time measured in clock cycles. The one-cycle architecture is 43% slower than the pipelined five-stage architecture.

The distribution of the hardware resources between the architectures for the Altera implementation is similar to the Xilinx implementation in case of the pipelined and multi-cycle architectures. The difference in resources between five and four stages pipelined architectures is 5%. Consequently, the multi-cycle architecture requires 9% less resources than the five-stage architecture. The major difference in implementations on these two FPGA platforms is the hardware resources required by the one-cycle architecture. Implemented on

the Altera platform the one-cycle processor occupies 734% resources more than the five-stage architecture. This phenomenon can be explained by the difference in FPGA organization and compilation tools for Xilinx and Altera.

Altera Quartus II compiler [80] recognizes inferred memory blocks in the proposed design even without BRAM optimization option. By default, the compiler is allowed to use hardware memory blocks for the inferred memory implementation. This feature causes only minor differences in the implementations with or without BRAM optimization. In fact, all architectures except the one-cycle are implemented with the utilization of hardware memory blocks. The one-cycle architecture does not contain inferred synchronous memory blocks. Therefore it is implemented purely in ALMs resulting in a jump of utilized FPGA resources.

Due to high level behavioral description of the proposed processor, there is no direct control of the utilization of memory blocks on the Altera platform. The analysis of the synthesis report shows that the data memory and regfile are implemented in hardware memory blocks every time when the automatic recognition of memory blocks is enabled. The excerpts from the compilation reports are shown in Appendix I. In several processor configurations the BRAM optimization option causes the implementation of the instruction memory into a memory block. But this effect is inconsistent through the explored set of implementations resulting in variations of the utilized hardware resources. Another source of variation is the utilization of Memory Logic Array Blocks (MLAB) or Memory 9-Kbit blocks (M9K) [81] for the regfile implementation. The compiler uses either option justifying the choice by the optimization strategy. In order to have a full control of the design implementation, it is necessary to use specific to Altera FPGA hardware resources (i.e. memory blocks, megafunctions) in the HDL code of the design. This approach would negate the portability concept critical to the proposed design. The variation in the implementation results on the Altera platform is a trade-off for the design portability.

The predictable and consistent reduction or the utilized hardware resources is achieved by implementation or the reduced ISA support. The removal of the barrel shifter and extra ALU operations benefits in 5% hardware saving for 32-bit one-cycle architecture and 30-42% for other architectures. The removed hardware is the constant number of ALMs for all architectures while the remaining occupied resources vary in different architectures. That

explains the significant difference in the ratio of occupied resources for the considered architectures.

Based on the data shown in Table 4.8 and Table 4.9 it may be deduced that the conducted evaluation of the 32-bit processor variants implemented in the different processor architectures can be applied to the implementations with other bit-widths. Though quantitative values may significantly differ, the general trends are common for all bit-widths.

6.3 ASIC Implementation Evaluation

The results of the implementation of the proposed processor using 0.18 μm technology process show the similar to FPGA implementation tendency. As shown in Figure 6.3 the clock speed ratio between two pipelined architectures is 21% in favor of the five-stage architecture. In the same way as the FPGA implementation, the multi-cycle architecture demonstrates higher clock speed than four-stage and one-cycle architectures. Nevertheless, the higher clock speed does not compensate the longer instruction execution time, due to the worse throughput of the multi-cycle architecture

The one-cycle architecture offers the advantage of the smallest die area and the clock speed just 30% less then the fastest five-stage pipelined architecture.

The distinguishable feature of the ASIC implementation is a minor variability of the occupied area for five-stage, four-stage and multi-cycle architectures. The difference does not exceed 3%. That effect can be explained by the strategy used for the design compilation. The compilation tool breaks the hierarchy of the design and optimizes it. The actual difference in hardware resources (i. e. registers, combinational logic, and memory) between these three architectures corresponds to the obtained values. The reduction of the die area for the one-cycle architecture is caused by absence of the inter-stage registers in this architecture.

The most attractive choice for the speed oriented application using the technology process is the five-stage pipelined architecture. This architecture features the fastest execution speed and 9% more die area than the most resource efficient one-cycle architecture.

In case of a tight die area constraint the best choice is the one-cycle architecture which occupies the smallest of all architectures area and offers the execution speed (testbench adjusted) 22% less than the fastest five-stage pipelined architecture.

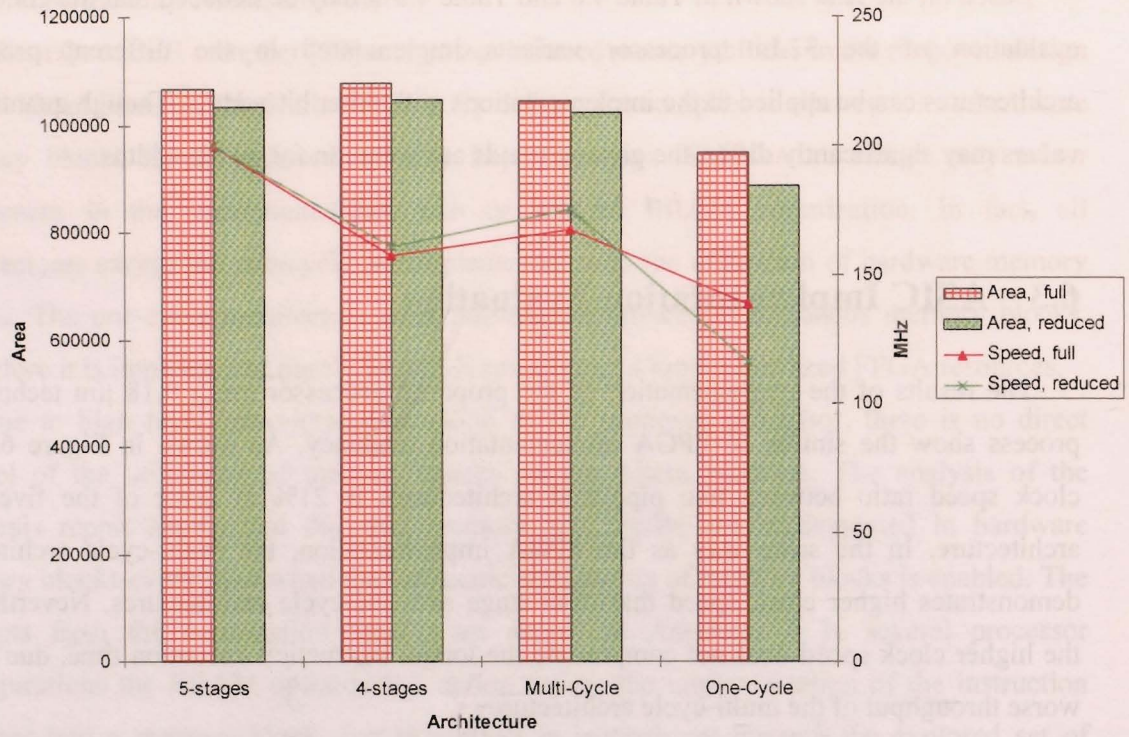


Figure 6.3: Evaluation chart of the architecture variants of 32-bit processor implemented using 0.18 μm technology process

The implementation of the reduced instruction set does not significantly affect the occupied die area. The maximum area saving is 6%. That can be explained by the more efficient implementation of the barrel shifter using the technology process compared to FPGA platform.

6.4 Evaluation Against Existing Solutions

The comparison of the proposed processor against the existing processors with similar architectures and implementation platforms is complicated by the differences in details of a particular design. The closest processor solutions were selected for the comparative evaluation. Table 6.1 outlines the major features of two configurations of the proposed 32-bit processor implemented on the Altera Stratix-III FPGA platform vs. similar Altera Nios II/s (small) and Nios II/e (economic) soft processors.

Table 6.1: Configurable MIPS processor variants vs. Altera Nios II/s/e

Feature	Processor Core			
	Config. MIPS	Nios II/s	Config. MIPS	Nios II/e
Bit-width	32 bits	32 bits	32 bits	32 bits
F max	96 MHz	165 MHz	89 MHz	200 MHz
Area	739 ALMs	700 ALMs	672 ALMs	350 ALMs
Pipeline	5 stages	5 stages	No	No
Cycles/Instruction	1	1	5 max	6 max
Hardware Multiply	No	3-cycle	No	No
Shifter	1-cycle barrel	3-cycle shift	1-cycle barrel	1 cycle-per-bit

The five-stage pipelined architecture yields to Nios II/s by the speed and occupied area due to the optimization of Nios II for the Altera FPGA architecture. Also the proposed configurable MIPS processor design uses more advanced barrel shifter which requires an additional area and contributes to the delay of the EXE stage. The same causes are responsible for the superiority of NIOS II/e over the multi-cycle architecture of the proposed processor implemented on the Altera platform.

The close contemporary solutions implemented on the Xilinx Virtex-5 FPGA platform are represented by Xilinx Microblaze v7.0 [82] and Leon3 [83] soft processors. Table 6.2 compares the critical specification details of these competitive processors with the proposed five-stage processor.

Due to the optimization of the Microblaze organization and instruction set for the specific structure of Xilinx FPGA, the Micoblaze speed performance greatly exceeds the configurable MIPS. By the same reason, Microblaze occupies 28% less hardware resources. On the other hand, Leon3, a portable research solution not optimized for a specific FPGA also demonstrates the inferior speed performance compared to Microblaze. The reduced clock speed is a trade-off for portability common for Leon3 and proposed configurable MIPS processors. As a result, the Leon3 is just 25% faster than the configurable MIPS. This advantage is stipulated by the longer pipeline which, in general, requires hardware overhead for data path and control elements.

Table 6.2: Configurable MIPS processor vs. Xilinx Microblaze and Leon3

Feature	Processor Core		
	Config. MIPS	**Microblaze**	**Leon3**
Bit-width	32 bits	32 bits	32 bits
F max	100 MHz	220 MHz	125 MHz
Area	1369 LUTs	980 LUTs	3500 LUTs
Pipeline	5 stages	5 stages	7 stages
Cycles/Instruction	1	1	1
Hardware Multiply	No	Optional	Optional
Shifter	1-cycle barrel	1-cycle barrel	3-cycle shift
Portability	Altera/Xilinx FPGA, ASIC	Xilinx FPGA only	Altera/Xilinx/Actel/Lattice FPGA, ASIC

Despite the fact that the proposed processor shows the performance worse than the commercial solutions, it has an advantage of an open design available for customization and tuning. It also offers the high portability which is not available for the considered commercial products. A distinguishable benefit of the proposed processor is the configurable data path bit-width. Such level of configurability is not offered by any processor selected for the comparative evaluation.

6.5 Summary

This chapter analyzed the results of implementation of the proposed processor on different hardware platforms. The detailed analysis was performed for the 32-bit processor configuration implemented using various options for the four considered architectures. The conducted analysis defined a correlation between a chosen processor configuration and technical parameters obtained after the implementation. The specific to a hardware platform implementation features were identified and explained using the obtained results. The selected processor configurations were compared with similar contemporary processor solutions. The advantages and disadvantages of the evaluated designs were described in details.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this project the configurable processor with MIPS compatible instruction set was developed. The configurability of the design offers a choice of four possible processor architectures:

- Five-stage pipelined
- Four-stages pipelined
- Multi-cycle unpipelined
- One-cycle unpipelined

Within each architecture the configurable options include data path bit-width, organization and size of the data and instruction memories, instruction subsetting, I/O space size and bit-width. The configuration tool was developed in order to facilitate configuration of the processor for the chosen specification. The processor design flow was established starting from a specification stage and finishing with the processor implementation in hardware and running an application program. The design flow invokes tools from different vendors and augmented with the custom tool connecting design stages. Following the design flow the selected processor configurations were implemented on Xilinx and Altera FPGA platforms. The full portability of the design was verified using 0.18 μm TSMC technology process for implementation. In order to maximize verification coverage, the specific testbench was developed and used for the design validation. The simulation of the design was performed on the behavioral and post-route levels for the selected configurations. In order to increase

verification confidence, one selected processor configuration was implemented in the hardware development board. The additional I/O hardware interfaces were developed connecting the selected processor core with LCD screen and inputting device. In addition to the testbench, the specific software driver was developed for communication with the development board I/O devices. As a result the demo design was successfully verified.

The implementation of the proposed processor variants on the different hardware platforms created the exploration space. This space was analyzed in order to determine the dependencies between implemented configuration options and technical parameters of the resulted processor. Major trends and rules for utilization of the specific configuration options were derived. The wide spectrum of the possible processor configurations provides an opportunity to optimize the selection of a processor for the specific application.

Comparison to the existing commercial and research solutions revealed advantages and disadvantages of the designed processor. The advantages include high portability, unique configuration options, and wide range of the processor variants.

7.2 Future Work

The processor design presented in this project explored only a limited number of the configuration options. The potential for the further improvement includes many advanced features inherent in modern processors. The list of the processor enhancements that can increase the performance of the presented processor includes but not limited by the following:

- Multiplier
- Divider
- Multilevel instruction and data caches
- Floating Point Unit (FPU)
- Memory Management Unit (MMU)
- Translation Look-aside Buffer (TLB)
- Dynamic branch prediction
- Superscalar architecture
- Unaligned data memory access

The proposed processor pipelined architecture considers only five and four stages variants. The future work may explore the architectures with shorter or longer pipelines and define benefits and trade-offs of these designs.

The presented processor design is based on the existing MIPS instruction set which is oriented toward the standard 32-bit data path. Therefore, the expansion of the instruction set with specific commands capable of the efficient handling of wider data words would increase the performance of processor configurations with wide data paths. The standard MIPS software development tools cannot support all possible improvements. Hence the development of the advanced software tools is required in order to use efficiently all described processor enhancements.

The developed *Configuration Manager* tool may include the additional feature providing the speed and hardware resources for the selected processor configuration. This feature would facilitate a choice of the processor configuration optimized for the specific application. Moreover, the configuration process may be completely automatic driven by the application parameters entered by a user.

The portability of the presented design has been verified only on three different hardware platforms. The future work may extend the implementation hardware with other FPGA families such as Actel, Latice, Atmel, Achronix, etc.

The performance of the proposed processor configurations may be estimated with better accuracy by execution of the standard benchmarks such as Dhrystone, SPEC, Whetstone, etc.

Evolution of the presented configurable processor introduces an opportunity for development of a processor solution far beyond the explored in this project.

Appendix A

Demo Design Program Code

```
// I/O ports mapped to dmem addresses
#define FIB_OUT ((volatile int*) 0x00000010)
#define LCD_OUT_PORT ((volatile int*) 0x00000014) // 5 :LCD_OUT_PORT
{lcd_output_data, lcd_drive, lcd_rs, lcd_rw_control, lcd_e}
#define FIB_IN_NUM          ((volatile int*) 0x00000008)  // 2 :Input for
Fibonacci number calculation
// Delays definitions
#define DELAY40ns 0
#define DELAY230ns 1
#define DELAY15ms 75000
#define DELAY4_1ms 20500
#define DELAY100us 500
#define DELAY40us 200
#define DELAY1us 5


int fib();
void Wait15ms ();
void Wait4_1ms ();
void Wait100us ();
void Wait40us ();
void Wait40ns ();
void Wait230ns ();
void Wait1us ();
void Write4bitsLCD ();
void Write8bitsLCD ();
void DisplayInit ();
void DisplayConfig ();
void SetAddr();
void WriteData();
void DrawFib();
void DrawEqu();
void DrawFib_N();
void DrawHexNum();//"Draw 4 hex digits number
unsigned int Hex2LcdChar();


int start(){
int N;
//int result;
//*FIB_OUT = fib(N);
*LCD_OUT_PORT = 0xfffffffe;
DisplayInit();
DisplayConfig();
for (; ; ) // Display in loop
      {
      // Display "Fib(" N ")=" fib(N)
            N = *FIB_IN_NUM;
```

```
            SetAddr(0);
            DrawFib();
            DrawFib_N(N);
            DrawEqu();
            DrawHexNum(fib(N));
      }
}

int fib(int n) {
      if (n==0) {return 1;}
if (n==1) {return 1;}
return (fib(n-1) + fib(n-2));
}

 void Wait15ms ()
{
      unsigned int i;
      for (i = 0; i < DELAY15ms; i++);
}

 void Wait4_1ms ()
{
      unsigned int i;
      for (i = 0; i < DELAY4_1ms; i++);
}


 void Wait100us ()
{
      unsigned int i;
      for (i = 0; i < DELAY100us; i++);
}


 void Wait40us ()
{
      unsigned int i;
      for (i = 0; i < DELAY40us; i++);
}

 void Wait40ns ()
{
      unsigned int i;
      for (i = 0; i < DELAY40ns; i++);
}

 void Wait230ns ()
{
      unsigned int i;
      for (i = 0; i < DELAY230ns; i++);
}

 void Wait1us ()
{
      unsigned int i;
      for (i = 0; i < DELAY1us; i++);
}
```

```
 void Write4bitsLCD (unsigned int lcddata, unsigned int lcd_rs)
{
      unsigned int datasend = 0xffffffff;

      datasend = lcddata;

      datasend = datasend<<1;
      datasend = datasend | 0x1; //lcd_drive =1

      datasend = datasend<<1; // lcd_rs =1

      datasend =  datasend | lcd_rs; // 1 ->data / 0 -> command
      datasend = datasend<<2; // lcd_rw_control = 0, lcd_e = 0
      *LCD_OUT_PORT = datasend; // send to port

      Wait40ns();

      datasend = datasend | 0x1; //lcd_e = 1
      *LCD_OUT_PORT = datasend; // send to port

      Wait230ns();

      datasend = datasend & 0xfffffffe; //lcd_e = 0
      *LCD_OUT_PORT = datasend; // send to port

      datasend = datasend | 0x0a; //cd_rw_control = 1
      *LCD_OUT_PORT = datasend; // send to port
}

 void Write8bitsLCD (unsigned int lcddata, unsigned int lcd_rs)
{
      unsigned int upper_nibble;
      upper_nibble = lcddata>>4;
      Write4bitsLCD (upper_nibble, lcd_rs); // write upper nibble

      Wait1us();

      Write4bitsLCD (lcddata, lcd_rs); // write low nibble

      Wait40us();
}

 void DisplayInit ()
{
      *LCD_OUT_PORT = 0xfffffffe; //set lcd_e low
      Wait15ms();

      Write4bitsLCD (0x03,0);
      Wait4_1ms();

      Write4bitsLCD (0x03,0);
      Wait100us();

      Write4bitsLCD (0x03,0);
      Wait40us();
```

```
	Write4bitsLCD (0x02,0);
	Wait40us();
}

 void DisplayConfig ()
{
	Write8bitsLCD (0x28,0); //Function Set
	Write8bitsLCD (0x06,0); //Entry Mode Set
	Write8bitsLCD (0x0c,0); //Display On/Off
	Write8bitsLCD (0x01,0); //Clear Display
	Wait4_1ms();
}
void SetAddr(unsigned int addr)
{
	unsigned int addr_send = addr | 0x80;

	Write8bitsLCD (addr_send,0); //Set DD RAM Address
}

void WriteData(unsigned int data)
{
	Write8bitsLCD (data,1); //Write Data to CG RAM or DD RAM
}

void DrawFib()//"Draw "Fib("
{
	WriteData(0x46); // draw F
	WriteData(0x69); // draw i
	WriteData(0x62); // draw b
	WriteData(0x28); // draw (
}

void DrawFib_N(unsigned int fib_num)//"Draw N
{
	unsigned int send_nibble;

	send_nibble = fib_num >> 4;
	WriteData( Hex2LcdChar(send_nibble) );

	send_nibble = fib_num;
	WriteData( Hex2LcdChar(send_nibble) );     // draw lower nibble
}

void DrawEqu()//"Draw ")="
{
	WriteData(0x29); // draw )
	WriteData(0x3d); // draw =
}

void DrawHexNum(unsigned int hex_num)//"Draw 4 hex digits number
{
	unsigned int send_nibble;

	send_nibble = hex_num >> 12;
	WriteData( Hex2LcdChar(send_nibble) ); // draw 4th nibble

	send_nibble = hex_num >> 8;
```

```
        WriteData( Hex2LcdChar(send_nibble) ); // draw 3rd nibble

        send_nibble = hex_num >> 4;
        WriteData( Hex2LcdChar(send_nibble) );  // draw 2nd nibble

        send_nibble = hex_num ;
        WriteData( Hex2LcdChar(send_nibble) ); // draw lower nibble
}

unsigned int Hex2LcdChar(unsigned int hex_nib)//Convert hex nibble to LCD
char
{
        unsigned int lcd_nibble;

        lcd_nibble = hex_nib & 0x0000000f;
        if (lcd_nibble >9)
        {
                lcd_nibble = (lcd_nibble & 0x07) -1; // 0111, convert to A-F
                lcd_nibble |= 0x40;
        }
        else lcd_nibble |= 0x30;

        return      lcd_nibble;
}
```

Appendix B

Processor Configuration File

B.1 Base Configuration File Template

```
`ifdef _PROCESSOR_CONFIG_FLAT_V_
`else
`define _PROCESSOR_CONFIG_FLAT_V_

//-----------------------------------------------------------------------
//Processor architecture definitions
//-----------------------------------------------------------------------
`define MIPS_PROCESSOR
`define processor_inst_width  32 //not reconfigurable(reserved)
// Instruction format
`define inst_opcode_width            6
`define inst_funct_width             6
`define inst_shamt_width             5
//The branch bug handling in the PCSpim compiler
`define PCSpim_compiler
`ifdef  PCSpim_compiler
`define Branch_jump PC_next<=PC+immediate;
`else
`define Branch_jump PC_next<=PC+`imem_step+immediate;
`endif
// debug provision
`define BREAK_ADDR0    'h1c/4
`define BREAK_ADDR1    'h274/4
`define BREAK_ADDR2    'h390/4
`define BREAK_ADDR3    'h31c/4

//`include "auto_config_part.v"
//***********************************************************************
// Automatically Generated Section of Configuration File
//***********************************************************************

//***********************************************************************
// Global processor definitons
//***********************************************************************
`define SHIFT_COMMANDS
`define SET_COMMANDS
//-----------------------------------------------------------------------
`define processor_data_width   32
// Instruction Memory definitions
`define imem_size  1024
`define imem_step  1
`define imem_addr_width    10
`define imem_shift 0
```

```
// Data Memory definitions
`define dmem_size  1024
`define dmem_step  4
`define dmem_addr_width    10
// Data Memory Mapping
`define dmem_up_limit  'h8fffffff
`define dmem_down_limit    'h10000000
// I/O Memory definitions
`define io_mem_size    1024
`define io_mem_width   32
`define io_addr_width  10
// I/O memory mapping
`define io_mem_down_limit  'h0
`define io_mem_up_limit    'h400000
// PC width
`define PC_width   32
//PC start address
`define reset_addr 'h80020000
// SP start address
`define SP_INIT    'h800bc000

//_____________________________________________________________________
//End of global processor definitons
//_____________________________________________________________________
//*********************************************************************
//Register file definitons
//*********************************************************************
`define REGISTER_FILE regfile //regfile module name
`define regfile_addr_width 5 ///< Register file address width
`define regfile_size (1<<`regfile_addr_width) ///< Register file size
`define regfile_width `processor_data_width ///< Register file width
`define link_addr 31 //Address of link register (return address - ra)
//End of register file definitons
//*********************************************************************
//-----------------Architecture Configuration--------------------------
//*********************************************************************
//Pipeline Interstage Register  definitons
//---------------------------------------------------------------------
// interstage_pass --> dummy reg pass
// interstage_data_reg --> reg
//---------------------------------------------------------------------
// Control Unit configuration
//---------------------------------------------------------------------
`ifdef MIPS_MULTICYCLE

`define IF_ID_INST_REG                      interstage_data_reg
`define ID_EX_IMM_REG                       interstage_pass
`define ID_EX_PC_REG                               interstage_pass
`define ID_EX_A_REG                                interstage_data_reg
`define ID_EX_B_REG                  interstage_data_reg
`define ID_EX_SHAMT_REG                            interstage_pass
`define EX_MEM_RESUT_REG             interstage_data_reg
`define EX_MEM_T_REG                 interstage_pass
`define MEM_WB_DMEM_REG              interstage_data_reg
`define MEM_WB_ALU_REG                     interstage_pass
`define IF_ID_PC_REG                 interstage_pass
`define ID_EX_RS_ADDR_REG        interstage_pass
`define ID_EX_RT_ADDR_REG        interstage_pass
`define ID_EX_RD_ADDR_REG                  interstage_pass
`define EX_MEM_RD_W_ADDR_REG     interstage_pass
`define MEM_WB_RD_W_ADDR_REG     interstage_pass
//---------------------------------------------------------------------
`define NO_FORWARDING
//---------------------------------------------------------------------
`define NO_DELAY_SLOT
//---------------------------------------------------------------------
`define CONTROL_UNIT                        control_mcycle//control unit module name
//---------------------------------------------------------------------

`elsif  ONECYCLE
```

```
`define IF_ID_INST_REG                      interstage_pass
`define ID_EX_IMM_REG                       interstage_pass
`define ID_EX_PC_REG                              interstage_pass
`define ID_EX_A_REG                               interstage_pass
`define ID_EX_B_REG                 interstage_pass
`define ID_EX_SHAMT_REG                           interstage_pass
`define EX_MEM_RESUT_REG            interstage_pass
`define EX_MEM_T_REG                interstage_pass
`define MEM_WB_DMEM_REG             interstage_pass
`define MEM_WB_ALU_REG                    interstage_pass
`define IF_ID_PC_REG                interstage_pass
`define ID_EX_RS_ADDR_REG       interstage_pass
`define ID_EX_RT_ADDR_REG       interstage_pass
`define ID_EX_RD_ADDR_REG                 interstage_pass
`define EX_MEM_RD_W_ADDR_REG    interstage_pass
`define MEM_WB_RD_W_ADDR_REG    interstage_pass
//-----------------------------------------------------------------------
`define NO_FORWARDING
//-----------------------------------------------------------------------
`define NO_DELAY_SLOT
//-----------------------------------------------------------------------
`define CONTROL_UNIT                      control_onecycle//control unit module name
//-----------------------------------------------------------------------

`elsif  FOURSTAGES

`define IF_ID_INST_REG                      interstage_data_reg
`define ID_EX_IMM_REG                       interstage_data_reg
`define ID_EX_PC_REG                              interstage_data_reg
`define ID_EX_A_REG                               interstage_data_reg
`define ID_EX_B_REG                 interstage_data_reg
`define ID_EX_SHAMT_REG                           interstage_data_reg
`define EX_MEM_RESUT_REG            interstage_pass
`define EX_MEM_T_REG                interstage_pass
`define MEM_WB_DMEM_REG             interstage_data_reg
`define MEM_WB_ALU_REG                    interstage_data_reg
`define IF_ID_PC_REG                interstage_data_reg
`define ID_EX_RS_ADDR_REG       interstage_data_reg
`define ID_EX_RT_ADDR_REG       interstage_data_reg
`define ID_EX_RD_ADDR_REG                 interstage_data_reg
`define EX_MEM_RD_W_ADDR_REG    interstage_pass
`define MEM_WB_RD_W_ADDR_REG    interstage_data_reg
//-----------------------------------------------------------------------
`define FORWARDING_4_STAGES
//-----------------------------------------------------------------------
`define CONTROL_UNIT                      control_pipe_4st      //control unit module name
//-----------------------------------------------------------------------

`else  // 5-stages pipeline

`define IF_ID_INST_REG                      interstage_data_reg
`define ID_EX_IMM_REG                       interstage_data_reg
`define ID_EX_PC_REG                              interstage_data_reg
`define ID_EX_A_REG                               interstage_data_reg
`define ID_EX_B_REG                 interstage_data_reg
`define ID_EX_SHAMT_REG                           interstage_data_reg
`define EX_MEM_RESUT_REG            interstage_data_reg
`define EX_MEM_T_REG                interstage_data_reg
`define MEM_WB_DMEM_REG             interstage_data_reg
`define MEM_WB_ALU_REG                    interstage_data_reg
`define IF_ID_PC_REG                interstage_data_reg
`define ID_EX_RS_ADDR_REG       interstage_data_reg
`define ID_EX_RT_ADDR_REG       interstage_data_reg
`define ID_EX_RD_ADDR_REG                 interstage_data_reg
`define EX_MEM_RD_W_ADDR_REG    interstage_data_reg
`define MEM_WB_RD_W_ADDR_REG    interstage_data_reg
//-----------------------------------------------------------------------
`define FORWARDING_5_STAGES
//-----------------------------------------------------------------------
`define CONTROL_UNIT                      control_pipe   //control unit module name
//-----------------------------------------------------------------------
```

```
`endif
//End of Configurable Architecture  definitons
//_____________________________________________________________________

//*********************************************************************
//ALU definitons
//*********************************************************************
`define ALU alu_behav // alu module name
`define alu_operand_width `processor_data_width ///< ALU operand bit width

//ALU opcodes
`define alu_add         'h0
`define alu_subb        'h1
`define alu_and         'h2
`define alu_or          'h3
`define alu_nor         'h4
`define alu_xor         'h5
`define alu_up          'h6
`define alu_a                   'h7
`define alu_opcode_width 3 ///< ALU opcode bit width

`ifdef SHIFT_COMMANDS
`define alu_sll                 'h8
`define alu_srl         'h9
`define alu_sra                 'ha
`define alu_sllv         'hb
`define alu_srlv         'hc
`define alu_srav         'hd
//................................................
`define alu_opcode_width 4 ///< ALU opcode bit width
`endif

`ifdef SET_COMMANDS
`define alu_slt         'he
`define alu_sltu        'hf
`define alu_opcode_width 4 ///< ALU opcode bit width
`endif

`define alu_opcodes_number (1<<`alu_opcode_width) ///< Number of ALU opcodes

//////////////////////
//End of ALU definitons
//_____________________________________________________________________
//*********************************************************************
//Data memory definitons
//*********************************************************************
`define DATA_MEMORY data_ram //data memory module name
`define dmem_width  `processor_data_width //Data memory bit width
//End of data memory definitons
//_____________________________________________________________________
//*********************************************************************
//Input/Output memory map definitons
//*********************************************************************
`define io_mem_step 4 //I/O memory step (how sw addresses it)
//End of data memory definitons
//_____________________________________________________________________

//*********************************************************************
//Instruction register definitons
//*********************************************************************
`define INST_REG inst_reg //instruction register module name
`define inst_reg_width  `processor_inst_width //Data memory bit width
//End of instruction register definitons
//_____________________________________________________________________
//*********************************************************************
//Instruction memory definitons
//*********************************************************************
`define INST_MEMORY inst_mem //instruction memory module name
`define imem_width `processor_inst_width //Bit width of the instruction memory
//End of instruction memory definitons
```

```
//_____________________________________________________________________
//*********************************************************************
//Control module definitons
//*********************************************************************
//----------FSM states-------------------------------------------------
`define FETCH                                   0
`define DECODE                                  1
`define EXECUTE_R_TYPE          2
`define EXECUTE_BRANCH          3
`define EXECUTE_MEM_TYPE        4
`define EXECUTE_J_TYPE          5
`define EXECUTE_IMM_TYPE        6
`define MEM_ACCESS                      7
`define WR_BACK                                 8
`define LAST_STATE                      9
//*********************************************************************
//Control commands definitions
//*********************************************************************
// Nex PC source controls
//---------------------------------------------------------------------
`define next_instruction        0
`define imm_jump                                1
`define branch                                  2
`define rs_jump                                 3
//`define stall_pc                              4
`define nextPCsrc_width     2 //log2(`rs_jump) ->width of nextPCsrc
//---------------------------------------------------------------------
// Write Back Register destination controls
//---------------------------------------------------------------------
`define ra_reg          0
`define rt_reg          1
`define rd_reg          2
`define wb_addr_src_width 2 //log2(`rd_reg) ->width of wb_addr_src
//---------------------------------------------------------------------
// ALU forwarding controls
//---------------------------------------------------------------------
`define reg_ex          0  // regile data from ID/EX stage
`define fwd_mem         1       // forwarding from EX/MEM stage
`define fwd_wb          2       // forwarding from MEM/WB stage
`define alu_src_fwd_width 2 //log2(`fwd_wb) ->width of alu_a_src_frw/alu_b_src_frw
//---------------------------------------------------------------------
// ALU operand source controls
//---------------------------------------------------------------------
//`define reg_ex                0  // regile data from ID/EX stage
// Operand A
`define pc_ex                   1 // PC from ID/EX stage
`define alu_a_src_width 1 //log2(`pc_ex) ->width of alu_a_src
// Operand B
`define imm_ex                  1 // Immediate from ID/EX stage
`define plus_step               2 // + imem_step (support of jal command R[31]=PC+8;PC=JumpAddr)
`define shift_step      3 // multicycle verson bqe/bne support
`define alu_b_src_width 2 //log2(`shift_step) ->width of alu_b_src
//---------------------------------------------------------------------
// Write Back data source controls
//---------------------------------------------------------------------
`define alu_wb          0  // ALU data
`define dmem_wb         1       // Data Mem data
`define wb_data_src_width       1 //log2(`dmem_wb) ->width of wb_data_src
//---------------------------------------------------------------------
// Data MEM input data source controls
//---------------------------------------------------------------------
`define dmem_data       0  // DMEM regular data input
`define dmem_fwd                1       // DMEM forwarded data input from WB stage
`define dmem_data_src_width     1 //log2(`dmem_fwd) ->width of dmem_data_src
//---------------------------------------------------------------------
//Rs jump source controls
//---------------------------------------------------------------------
`define j_rs_id         0  // rs data from ID stage
`define fwd_mem         1       // forwarding from MEM stage
`define fwd_wb          2       // forwarding from WB stage
`define jr_src_frw_width 2 //log2(`fwd_wb) ->width of jr_src_frw
```

```
//-----------------------------------------------------------------------
// Comparator forwarding controls
//-----------------------------------------------------------------------
`define cmp_reg        0  // regile data from ID/EX stage
`define cmp_fwd        1       // forwarding from EX/MEM
`define cmp_src_fwd_width 1 //log2(`cmp_fwd) ->width of cmp_rs_src_fwd/cmp_rt_src_fwd
//_____________________________________________________________________
//*********************************************************************
//Parse instruction definitons
//*********************************************************************
//Processor opcode up limit
`define opcode_up `processor_inst_width-1
//Processor opcode down limit
`define opcode_down `processor_inst_width-`inst_opcode_width
//Processor immediate up limit
`define immediate_up `processor_inst_width -`inst_opcode_width-(`regfile_addr_width*2)-1
//Processor shift amount up limit
`define shamt_up `processor_inst_width -`inst_opcode_width-(`regfile_addr_width*3)-1
//Processor shift amount down limit
`define shamt_down `shamt_up - `inst_shamt_width +1
//Processor jump address up limit
`define jump_addr_up `processor_inst_width -`inst_opcode_width-1
//Processor sign extension width
`define sign_ext_width `processor_data_width-(`immediate_up+1)
//Processor sign extension width
`define zero_ext_width `processor_data_width-(`immediate_up)
// Register Rs
`define rs_addr_up `inst_reg_width-`inst_opcode_width-1
`define rs_addr_down   `rs_addr_up-`regfile_addr_width+1
// Register Rt
`define rt_addr_up `rs_addr_down - 1
`define rt_addr_down `rt_addr_up - `regfile_addr_width + 1
// Register Rd
`define rd_addr_up `rt_addr_down-1
`define rd_addr_down `rd_addr_up-`regfile_addr_width+1
//End of parse instruction definitons
//_____________________________________________________________________

`ifdef MIPS_PROCESSOR

//*********************************************************************
//MIPS instruction Set definitions
//*********************************************************************
//R type instructon functions
`define ADD         'h20
`define ADDU        'h21
`define SUB         'h22
`define SUBU        'h23
`define AND         'h24
`define OR          'h25
`define XOR         'h26
`define NOR         'h27
`define SLT         'h2a
`define SLTU        'h2b
`define SLL         'h0
`define SRL         'h2
`define SRA         'h3
`define SLLV        'h4
`define SRLV        'h6
`define SRAV        'h7
`define JR          'h8
`define JARL        'h9
`define MOVZ        'ha
`define MOVN        'hb
`define SYSCALL     'hc
`define BREAK       'hd
`define SYNC        'hf
`define MFHI        'h10
`define MTHI        'h11
`define MFLO        'h12
`define MTLO        'h13
```

```
`define MULT        'h18
`define MULU        'h19
`define DIV         'h1a
`define DIVU        'h1b
`define TGE         'h30
`define TGEU        'h31
`define TLTT        'h32
`define TLTU        'h33
`define TEQ         'h34
`define TNE         'h36

// Instruction opcodes
`define R_TYPE      'h0
`define J           'h2
`define JAL         'h3
`define BEQ         'h4
`define BNE         'h5
`define BLEZ        'h6
`define BGTZ        'h7
`define ADDI        'h8
`define ADDIU       'h9
`define SLTI        'ha
`define SLTIU       'hb
`define ANDI        'hc
`define ORI         'hd
`define XORI        'he
`define LUI         'hf
`define LB          'h20
`define LH          'h21
`define LWL         'h22
`define LW          'h23
`define LBU         'h24
`define LHU         'h25
`define LWR         'h26
`define SB          'h28
`define SH          'h29
`define SWL         'h2a
`define SW          'h2b
`define SWR         'h2e
`define CACHE       'h2f
`define LL          'h30
`define LWC1        'h31
`define LWC2        'h32
`define PREF        'h33
`define LDC1        'h35
`define LDC2        'h36
`define SC          'h38
`define SWC1        'h39
`define SWC2        'h3a
`define SDC1        'h3d
`define SDC2        'h3e

//////////////////////
//End of MIPS instruction Set definitions
//_____________________________________________________________________
`endif // MIPS
`ifdef DLX_PROCESSOR

//*********************************************************************
//DLX instruction Set definitions
//*********************************************************************
//R type instructon functions
`define ADD         'h20
`define SUB         'h22
`define AND         'h24
`define OR          'h25
`define XOR         'h26

`define SEQ         'h28
`define SLT         'h2a
`define SLE         'h2c
```

```
`define SNE         'h29

`define SLL         'h4
`define SRL         'h6
`define SRA         'h7

// Instruction opcodes
`define R_TYPE      'h0
`define J           'h2
`define JAL         'h3
`define BEQZ        'h4
`define BNEZ        'h5
`define ADDI        'h8
`define SUBI        'ha
`define ANDI        'hc
`define ORI         'hd
`define XORI        'he
`define LHI         'hf
`define JR          'h12
`define JARL        'h13
`define SLLI        'h14
`define SRLI        'h16
`define SEQI        'h18
`define SRAI        'h17
`define SNEI        'h19
`define SLTI        'h1a
`define SLEI        'h1c
`define LW          'h23
`define SW          'h2b


//////////////////////
//End of DLX instruction Set definitions
//_____________________________________________________________________
`endif // DLX
`endif // _PROCESSOR_CONFIG_FLAT_V_
```

B.2 Automatically Generated Part of Configuration File

The example is shown for the 5-stages architecture with the full support of the instruction set and BRAM optimizaiton.

```
//*****************************************************************************
// Automatically Generated Section of Configuration File
//*****************************************************************************

//*****************************************************************************
// Global processor definitons
//*****************************************************************************
`define SHIFT_COMMANDS
`define SET_COMMANDS
`define processor_data_width   256
// Instruction Memory definitions
`define imem_size  1024
`define imem_step  1
`define imem_addr_width    10
`define imem_shift 0
// Data Memory definitions
`define dmem_size  1024
```

```
`define dmem_step  4
`define dmem_addr_width    10
// Data Memory Mapping
`define dmem_up_limit  'h80000000
`define dmem_down_limit    'h10000000
// I/O Memory definitions
`define io_mem_size    1024
`define io_mem_width   32
`define io_addr_width  10
// I/O memory mapping
`define io_mem_down_limit  'h0
`define io_mem_up_limit    'h400000
// PC width
`define PC_width   32
//PC start address
`define reset_addr 'h400000
// SP start address
`define SP_INIT    'h7ffffffc
// Padding of PC bitwidth natural(`processor_data_width-`PC_width)
`define PC_padding    224
`define FPGA_BRAM
```

Appendix C

Implementation Reports

C.1 Xilinx Summary Reports

Implementation report for the following processor configuration:

- 4-stage architecture
- 256-bit data path
- BRAM optimization
- Full instruction set support

MIPS_DLX Project Status			
Project File:	mips_dlx.ise	**Current State:**	Placed and Routed
Module Name:	mips_dlx	• **Errors:**	No Errors
Target Device:	xc5vlx50-1ff324	• **Warnings:**	318 Warnings
Product Version:	ISE 9.2.04i	• **Updated:**	Tue Feb 2 19:18:58 2010

MIPS_DLX Partition Summary
No partition information was found.

Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	1,635	28,800	5%	
Number used as Flip Flops	1,634			
Number used as Latch-thrus	1			
Number of Slice LUTs	8,720	28,800	30%	
Number used as logic	8,203	28,800	28%	
Number using O6 output only	8,063			
Number using O5 and O6	140			
Number used as Memory	516	7,680	6%	
Number used as Dual Port RAM	516			
Number using O6 output only	4			

Number using O5 output only	4			
Number using O5 and O6	508			
Number used as exclusive route-thru	1			
Number of route-thrus	1	57,600	1%	
Number using O5 and O6	1			
Slice Logic Distribution				
Number of occupied Slices	2,879	7,200	39%	
Number of LUT Flip Flop pairs used	10,330			
Number with an unused Flip Flop	8,695	10,330	84%	
Number with an unused LUT	1,610	10,330	15%	
Number of fully used LUT-FF pairs	25	10,330	1%	
Number of unique control sets	9			
IO Utilization				
Number of bonded IOBs	78	220	35%	
Specific Feature Utilization				
Number of BlockRAM/FIFO	9	48	18%	
Number using BlockRAM only	9			
Total primitives used				
Number of 36k BlockRAM used	7			
Number of 18k BlockRAM used	3			
Total Memory used (KB)	306	1,728	17%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Total equivalent gate count for design	1,321,378			
Additional JTAG gate count for IOBs	3,744			
Performance Summary				

Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports

Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	Tue Feb 2 18:39:14 2010	0	140 Warnings	7 Infos
Translation Report	Current	Tue Feb 2 18:40:14 2010	0	0	0
Map Report	Current	Tue Feb 2 18:56:39 2010	0	177 Warnings	6 Infos
Place and Route Report	Current	Tue Feb 2 19:17:29 2010	0	1 Warning	1 Info
Static Timing Report	Current	Tue Feb 2 19:18:57 2010	0	0	2 Infos
Bitgen Report					

Implementation report for the following processor configuration:

- 5-stages architecture
- 128-bit data path
- No BRAM optimization
- Full instruction set support

MIPS_DLX Project Status			
Project File:	mips_dlx.ise	**Current State:**	Placed and Routed
Module Name:	mips_dlx	• **Errors:**	No Errors
Target Device:	xc5vlx50-1ff324	• **Warnings:**	218 Warnings
Product Version:	ISE 9.2.04i	• **Updated:**	Sun Jan 17 23:41:03 2010

MIPS_DLX Partition Summary
No partition information was found.

Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	1,006	28,800	3%	
Number used as Flip Flops	1,005			
Number used as Latch-thrus	1			
Number of Slice LUTs	4,757	28,800	16%	
Number used as logic	3,988	28,800	13%	
Number using O6 output only	3,887			
Number using O5 and O6	101			
Number used as Memory	768	7,680	10%	
Number used as Dual Port RAM	256			
Number using O5 and O6	256			
Number used as Single Port RAM	512			
Number using O6 output only	512			
Number used as exclusive route-thru	1			
Number of route-thrus	1	57,600	1%	
Number using O5 and O6	1			
Slice Logic Distribution				

Number of occupied Slices	1,841	7,200	25%	
Number of LUT Flip Flop pairs used	5,739			
Number with an unused Flip Flop	4,733	5,739	82%	
Number with an unused LUT	982	5,739	17%	
Number of fully used LUT-FF pairs	24	5,739	1%	
Number of unique control sets	7			
IO Utilization				
Number of bonded IOBs	78	220	35%	
Specific Feature Utilization				
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Total equivalent gate count for design	169,476			
Additional JTAG gate count for IOBs	3,744			

Performance Summary

Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports

Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	Sun Jan 17 15:01:28 2010	0	124 Warnings	7 Infos
Translation Report	Current	Sun Jan 17 23:32:07 2010	0	1 Warning	0
Map Report	Current	Sun Jan 17 23:39:34 2010	0	92 Warnings	6 Infos
Place and Route Report	Current	Sun Jan 17 23:40:38 2010	0	1 Warning	1 Info
Static Timing Report	Current	Sun Jan 17 23:41:02 2010	0	0	2 Infos
Bitgen Report					

Implementation report for the following processor configuration:

- One-cycle architecture
- 16-bit data path
- No BRAM optimization
- Reduced instruction set

MIPS_DLX Project Status			
Project File:	mips_dlx.ise	**Current State:**	Placed and Routed
Module Name:	mips_dlx	• **Errors:**	No Errors
Target Device:	xc5vlx50-1ff324	• **Warnings:**	130 Warnings
Product Version:	ISE 9.2.04i	• **Updated:**	Thu Feb 11 19:14:57 2010

MIPS_DLX Partition Summary
No partition information was found.

Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	10	28,800	1%	
Number used as Flip Flops	10			
Number of Slice LUTs	419	28,800	1%	
Number used as logic	323	28,800	1%	
Number using O6 output only	323			
Number used as Memory	96	7,680	1%	
Number used as Dual Port RAM	32			
Number using O5 and O6	32			
Number used as Single Port RAM	64			
Number using O6 output only	64			
Number of route-thrus	1	57,600	1%	
Number using O5 output only	1			
Slice Logic Distribution				
Number of occupied Slices	135	7,200	1%	
Number of LUT Flip Flop pairs used	419			

Number with an unused Flip Flop	409	419	97%	
Number with an unused LUT	0	419	0%	
Number of fully used LUT-FF pairs	10	419	2%	
Number of unique control sets	3			
IO Utilization				
Number of bonded IOBs	62	220	28%	
Specific Feature Utilization				
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Total equivalent gate count for design	18,941			
Additional JTAG gate count for IOBs	2,976			

Performance Summary			
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Thu Feb 11 19:09:19 2010	0	100 Warnings	3 Infos
Translation Report	Current	Thu Feb 11 19:13:03 2010	0	5 Warnings	0
Map Report	Current	Thu Feb 11 19:14:14 2010	0	24 Warnings	6 Infos
Place and Route Report	Current	Thu Feb 11 19:14:43 2010	0	1 Warning	1 Info
Static Timing Report	Current	Thu Feb 11 19:14:57 2010	0	0	2 Infos
Bitgen Report					

C.2 BRAM Utilization Reports

Excerpt from the synthesis report in case of the successful BRAM utilization:

```
INFO:Xst:2694 - Unit <inst_mem> : The ROM <Mrom_data_out_asynch> will be
implemented as a read-only BLOCK RAM, absorbing the register: <data_out>.
INFO:Xst:2690 - Unit <data_ram> : The RAM <Mram_ram> will be implemented as
BLOCK RAM
```

Excerpt from the synthesis report if BRAM optimization is not used:

```
HDL ADVISOR - Unit <data_ram> : The RAM <Mram_ram> will be implemented on
LUTs either because you have described an asynchronous read or because of
currently unsupported block RAM features. If you have described an
asynchronous read, making it synchronous would allow you to take advantage
of available block RAM resources, for optimized device usage and improved
timings. Please refer to your documentation for coding guidelines.
```

Appendix D

TSMC 0.18 μm Process Implementation

Synopsys Design Analyzer script for compilation of the five-stage pipelined architecture

```
sh rm -Rf Work
sh mkdir Work

define_design_lib Work -path "./Work"

analyze -format verilog -lib WORK {"control_pipe.v"}
analyze -format verilog -lib WORK {"mips_dlx.v"}
analyze -format verilog -lib WORK {"memory.v"}
analyze -format verilog -lib WORK {"processor_alu.v"}
analyze -format verilog -lib WORK {"interstage_pass.v"}
analyze -format verilog -lib WORK {"interstage_data_reg.v"}
analyze -format verilog -lib WORK {"regfile.v"}
analyze -format verilog -lib WORK {"inst_mem.v"}

elaborate mips_dlx -arch "verilog" -lib DEFAULT -update

set_load 20 "io_mem_write_enable"
set_load 20 "io_mem_enable"
set_load 20 "io_mem_addr*"
set_load 20 "io_mem_data_in*"

create_clock -name "clk" -period 4 -waveform {   "0" "2" } {   "clk" }
set_dont_touch_network find( clock, "clk")
set_clock_skew -propagated clk
set_clock_skew -plus_uncertainty 0.1 "clk"
set_clock_skew -minus_uncertainty 0.1 "clk"
set_fix_multiple_port_nets -all
write -format db -hierarchy -output "mips_constrained.db"
{"mips_dlx.db:mips_dlx"}

remove_design find(design "*")
read -format db {"mips_constrained.db"}
current_design "mips_constrained.db:mips_dlx"
compile -ungroup_all
current_design "mips_constrained.db:mips_dlx"
write -format db -hierarchy -output "mips_compile1.db"
{"mips_constrained.db:mips_dlx"}
```

```
report_area
report_constraints
report_timing -path full -delay max -max_paths 1 -nworst 1
check_design

remove_design find(design "*")
read -format db {"mips_compile1.db"}
current_design "mips_compile1.db:mips_dlx"
compile -map_effort high -incremental_map

write -format db -hierarchy -output "mips_compile2.db"
{"mips_compile1.db:mips_dlx"}
report_area
report_constraints

report_timing -path full -delay max -max_paths 1 -nworst 1

check_design
change_names -hier -rule verilog
write -format verilog -hierarchy -output "mips_gate.v"
{"mips_compile1.db:mips_dlx"}
```

Appendix E

Demo Design Constraints and Report

```
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 20.0ns HIGH 50 %;
NET "clk" USELOWSKEWLINES;

#
# soldered 50MHz Clock.
#
NET "clk" LOC = "C9" | IOSTANDARD = LVTTL;
#
#
# Simple LEDs
# Require only 3.5mA.
#
NET "led<0>" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
NET "led<1>" LOC = "E12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
NET "led<2>" LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
NET "led<3>" LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
NET "led<4>" LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
NET "led<5>" LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
NET "led<6>" LOC = "E9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
NET "led<7>" LOC = "F9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
#
#
# LCD display
# Very slow so can use lowest drive strength.
#
NET "lcd_rs"   LOC = "L18" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
NET "lcd_rw"   LOC = "L17" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
NET "lcd_e"    LOC = "M18" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
```

```
NET "lcd_d<4>" LOC = "R15" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
NET "lcd_d<5>" LOC = "R16" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
NET "lcd_d<6>" LOC = "P17" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
NET "lcd_d<7>" LOC = "M15" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
#
# Strata Flash (need to disable to use LCD display)
#
NET "strataflash_oe" LOC = "C18" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
NET "strataflash_ce" LOC = "D16" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
NET "strataflash_we" LOC = "D17" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 2;
#
#
# Simple switches
#   Pull UP resistors used to stop floating condition during switching.
#
NET "switch<0>" LOC = "L13" | IOSTANDARD = LVTTL | PULLUP;
NET "switch<1>" LOC = "L14" | IOSTANDARD = LVTTL | PULLUP;
NET "switch<2>" LOC = "H18" | IOSTANDARD = LVTTL | PULLUP;
NET "switch<3>" LOC = "N17" | IOSTANDARD = LVTTL | PULLUP;
#
#
# Press buttons
#   Must have pull DOWN resistors to provide Low when not pressed.
#
NET "btn_north" LOC = "V4"  | IOSTANDARD = LVTTL | PULLDOWN;
NET "btn_east"  LOC = "H13" | IOSTANDARD = LVTTL | PULLDOWN;
NET "btn_south" LOC = "K17" | IOSTANDARD = LVTTL | PULLDOWN;
NET "btn_west"  LOC = "D18" | IOSTANDARD = LVTTL | PULLDOWN;
#
# Rotary encoder.
#   Rotation contacts require pull UP resistors to provide High level.
#   Press contact requires pull DOWN resistor to provide Low when not pressed..
#
NET "rotary_a"     LOC = "K18" | IOSTANDARD = LVTTL | PULLUP;
NET "rotary_b"     LOC = "G18" | IOSTANDARD = LVTTL | PULLUP;
NET "rotary_press" LOC = "V16" | IOSTANDARD = LVTTL | PULLDOWN;
#
#
# End of File
#
```

MIPS_DLX_DEMO Project Status			
Project File:	mips_dlx_demo.ise	**Current State:**	Programming File Generated
Module Name:	demo_lcd_fib	• **Errors:**	No Errors
Target Device:	xc3s500e-5fg320	• **Warnings:**	148 Warnings
Product Version:	ISE 9.2.04i	• **Updated:**	Thu Dec 31 20:46:33 2009

MIPS_DLX_DEMO Partition Summary
No partition information was found.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	1,399	9,312	15%	
Number of 4 input LUTs	4,355	9,312	46%	
Logic Distribution				
Number of occupied Slices	3,369	4,656	72%	
Number of Slices containing only related logic	3,369	3,369	100%	
Number of Slices containing unrelated logic	0	3,369	0%	
Total Number of 4 input LUTs	4,888	9,312	52%	
Number used as logic	4,355			
Number used as a route-thru	19			
Number used for 32x1 RAMs	512			
Number used as Shift registers	2			
Number of bonded IOBs	30	232	12%	
IOB Flip Flops	15			
Number of GCLKs	1	24	4%	
Total equivalent gate count for design	106,751			
Additional JTAG gate count for IOBs	1,440			

Performance Summary			
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Wed Dec 9 22:39:46 2009	0	141 Warnings	6 Infos
Translation Report	Current	Wed Dec 9 22:40:08 2009	0	1 Warning	0
Map Report	Current	Wed Dec 9 22:40:37 2009	0	3 Warnings	3 Infos
Place and Route Report	Current	Wed Dec 9 22:44:01 2009	0	3 Warnings	0
Static Timing Report	Current	Wed Dec 9 22:44:18 2009	0	0	2 Infos
Bitgen Report	Current	Wed Dec 9 22:46:13 2009	0	0	0

Appendix F

Fibonacci Test Program

The assembler and machine code of Fibonacci number calculation program is shown below. The values are presented in hexadecimal format. This code is a part of the instruction memory module. Each row contains Verilog assignment for ROM address and comments showing a corresponding assembler code.

F.1 Program With Delay Slot And Reordering

```
`ifdef _FIB_PROGRAM_V_
`else
`define _FIB_PROGRAM_V_

assign ram[ 0]='h8c1d0004;//lw $29, 4($0)       ;40_0000: lw $sp, 4($zero) //load from I/O mem
assign ram[ 1]='h8c040008;//lw $4, 8($0)        ;40_0004: lw $a0, 8($zero) //load from I/O mem
assign ram[ 2]='h23bdfff4;//addi $29, $29, -12  ; 40_0008: addi $sp, $sp, -12 <-- fib
assign ram[ 3]='hafbf0008;//sw $31, 8($29)      ; 40_000c: sw $ra, 8($sp)
assign ram[ 4]='hafb00004;//sw $16, 4($29)      ; 40_0010: sw $s0, 4($sp)
assign ram[ 6]='h20020001;//addi $2, $0, 1      ; 40_0018: addi $v0, $zero, 1
assign ram[ 5]='h1080000d;//beq $4, $0, 52      ; 40_0014: beq $a0, $zero, fin
assign ram[ 7]='h20080001;//addi $8, $0, 1      ; 40_001c: addi $t0, $zero, 1
assign ram[ 8]='h1088000a;//beq $4, $8, 40      ; 40_0020: beq $a0, $t0, fin
assign ram[ 9]='h00000000;//nop                 ; 40_0024: nop (delay slot)
assign ram[10]='h2084ffff;//addi $4, $4, -1     ; 40_0028: addi $a0, $a0, -1
assign ram[12]='hafa40000;//sw $4, 0($29)       ; 40_0030: sw $a0, 0($sp)
assign ram[11]='h0c100002;//jal 0x00400008 [fib]; 40_002c: jal fib
assign ram[13]='h8fa40000;//lw $4, 0($29)       ; 40_0034: lw $a0, 0($sp)
assign ram[14]='h2084ffff;//addi $4, $4, -1     ; 40_0038: addi $a0, $a0, -1
assign ram[16]='h00408020;//add $16, $2, $0     ; 40_0040: add $s0, $v0, $zero
assign ram[15]='h0c100002;//jal 0x00400008 [fib]; 40_003c: jal fib
assign ram[17]='h00501020;//add $2, $2, $16     ; 40_0044: add $v0, $v0, $s0
assign ram[18]='hac02000c;//sw $2, 12($0)       ; 40_0048: sw $v0,12($zero)//store to I/O mem
assign ram[19]='h8fb00004;//lw $16, 4($29)      ; 40_004c: lw $s0, 4($sp)          <-- fin
assign ram[20]='h8fbf0008;//lw $31, 8($29)      ; 40_0050: lw $ra, 8($sp)
assign ram[21]='h23bd000c;//addi $29, $29, 12   ; 40_0054: addi $sp, $sp, 12
assign ram[22]='hac1d0010;//sw $29, 16($0)      ; 40_0058: sw $sp,16($zero)//store to I/O mem
assign ram[23]='h03e00008;//jr $31              ; 40_005c: jr $ra
assign ram[24]='h00000000;//nop                 ; 40_0060: nop
assign ram[25]='h0000000c;//syscall             ; 40_0064: syscall

assign ram[26]='h0000000c;//
assign ram[27]='h00000000;////
assign ram[28]='h00000000;////
assign ram[29]='h00000000;////

`endif // _FIB_PROGRAM_V_
```

Note. The reordering is shown in bold font.

F.2 Program Without Delay Slot And Reordering

```
`ifdef _FIB_PROGRAM_V_
`else
`define _FIB_PROGRAM_V_

assign ram[ 0]='h8c1d0004;//  lw $29, 4($0)  ; 40_0000: lw $sp, 4($zero) //load from I/O mem
assign ram[ 1]='h8c040008;//  lw $4, 8($0)   ; 40_0004: lw $a0, 8($zero) //load from I/O mem
assign ram[ 2]='h23bdfff4;//  addi $29, $29, -12  ; 40_0008: addi $sp, $sp, -12 <-- fib
assign ram[ 3]='hafbf0008;//  sw $31, 8($29)      ; 40_000c: sw $ra, 8($sp)
assign ram[ 4]='hafb00004;//  sw $16, 4($29)      ; 40_0010: sw $s0, 4($sp)
assign ram[ 5]='h20020001;//  addi $2, $0, 1      ; 40_0014: addi $v0, $zero, 1
assign ram[ 6]='h1080000d;//  beq $4, $0, 52      ; 40_0018: beq $a0, $zero, fin
assign ram[ 7]='h20080001;//  addi $8, $0, 1      ; 40_001c: addi $t0, $zero, 1
assign ram[ 8]='h1088000a;//  beq $4, $8, 40      ; 40_0020: beq $a0, $t0, fin
assign ram[ 9]='h00000000;//  nop                 ; 40_0024: nop
assign ram[10]='h2084ffff;//  addi $4, $4, -1     ; 40_0028: addi $a0, $a0, -1
assign ram[11]='hafa40000;//  sw $4, 0($29)       ; 40_002c: sw $a0, 0($sp)
assign ram[12]='h0c100002;//  jal 0x00400008 [fib]; 40_0030: jal fib
assign ram[13]='h8fa40000;//  lw $4, 0($29)       ; 40_0034: lw $a0, 0($sp)
assign ram[14]='h2084ffff;//  addi $4, $4, -1     ; 40_0038: addi $a0, $a0, -1
assign ram[15]='h00408020;//  add $16, $2, $0     ; 40_003c: add $s0, $v0, $zero
assign ram[16]='h0c100002;//  jal 0x00400008 [fib]; 40_0040: jal fib
assign ram[17]='h00501020;//  add $2, $2, $16     ; 40_0044: add $v0, $v0, $s0
assign ram[18]='hac02000c;//  sw $2, 12($0)      ; 40_0048: sw $v0,12($zero)//store to I/O mem
assign ram[19]='h8fb00004;//  lw $16, 4($29)      ; 40_004c: lw $s0, 4($sp)       <-- fin
assign ram[20]='h8fbf0008;//  lw $31, 8($29)      ; 40_0050: lw $ra, 8($sp)
assign ram[21]='h23bd000c;//  addi $29, $29, 12   ; 40_0054: addi $sp, $sp, 12
assign ram[22]='hac1d0010;//  sw $29, 16($0)     ; 40_0058: sw $sp,16($zero)//store to I/O mem
assign ram[23]='h03e00008;//  jr $31              ; 40_005c: jr $ra
assign ram[24]='h00000000;//  nop                                ; 40_0060: nop
assign ram[25]='h0000000c;//  syscall             ; 40_0064: syscall
assign ram[26]='h0000000c;//
assign ram[27]='h00000000;////
assign ram[28]='h00000000;////
assign ram[29]='h00000000;////

`endif // _FIB_PROGRAM_V_
```

Appendix G

Configurable Processor Verification

G.1 Verification Reports

Typical successful completion report:

```
# Fibonacci number test SUCCESSFULLY completed, Fib (  5) = 8
# Test finished after          242 machine cycles
```

Typical unsuccessful completion report with calculation errors:

```
# Test completed with ERRORS: Expected Fib(  5)= 8,
obtained Fib( 5)=   x
# Test finished after          242 machine cycles
```

Typical unsuccessful completion report due to a timeout:

```
# Test test finished UNSUCCESSFULLY due to the timeout
# Test finished after        50001 machine cycles
```

G.2 Simulation Waveforms

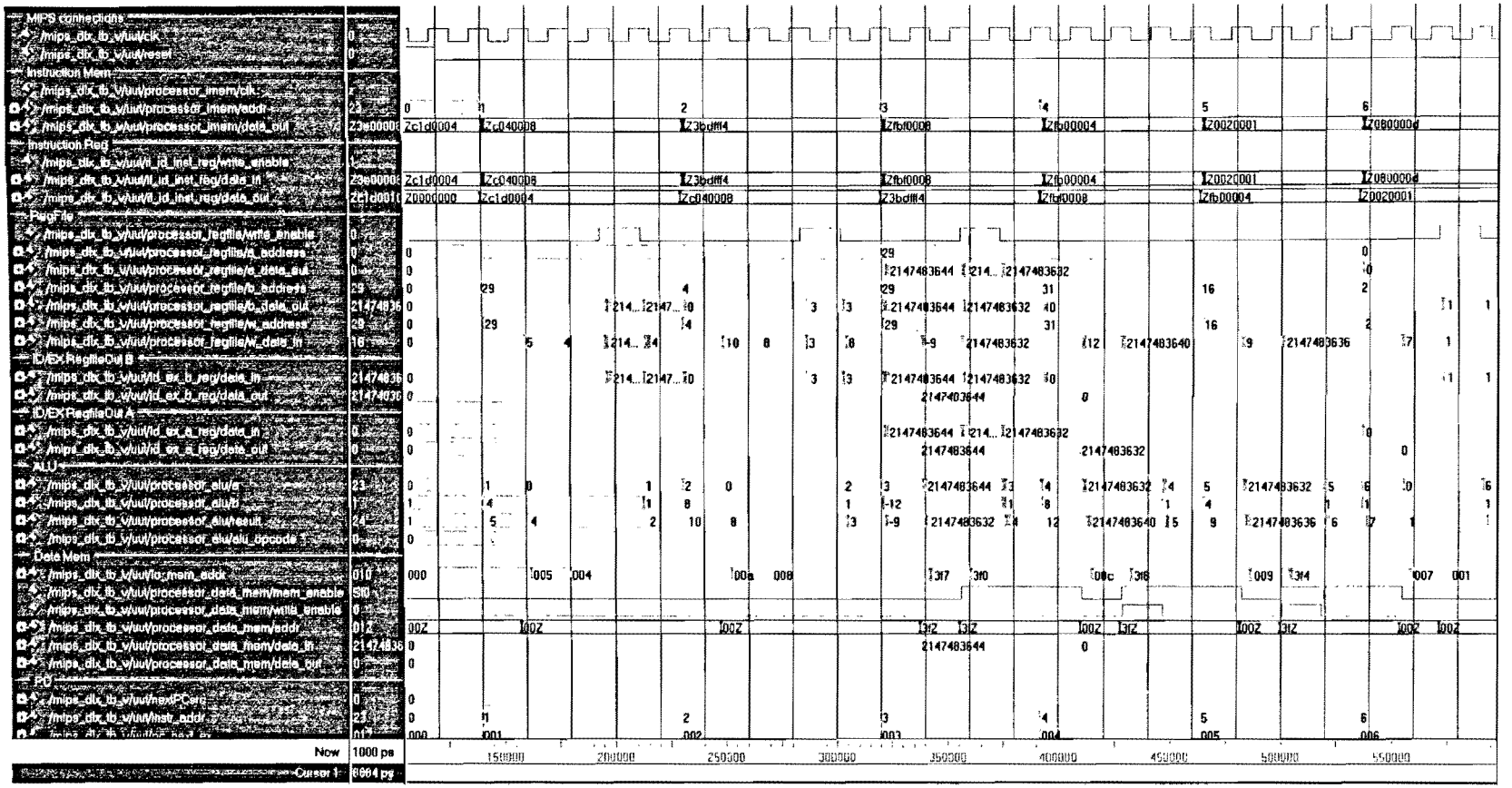


Figure G.1: Post-route simulation waveforms of the 32-bit configuration of multi-cycle architecture

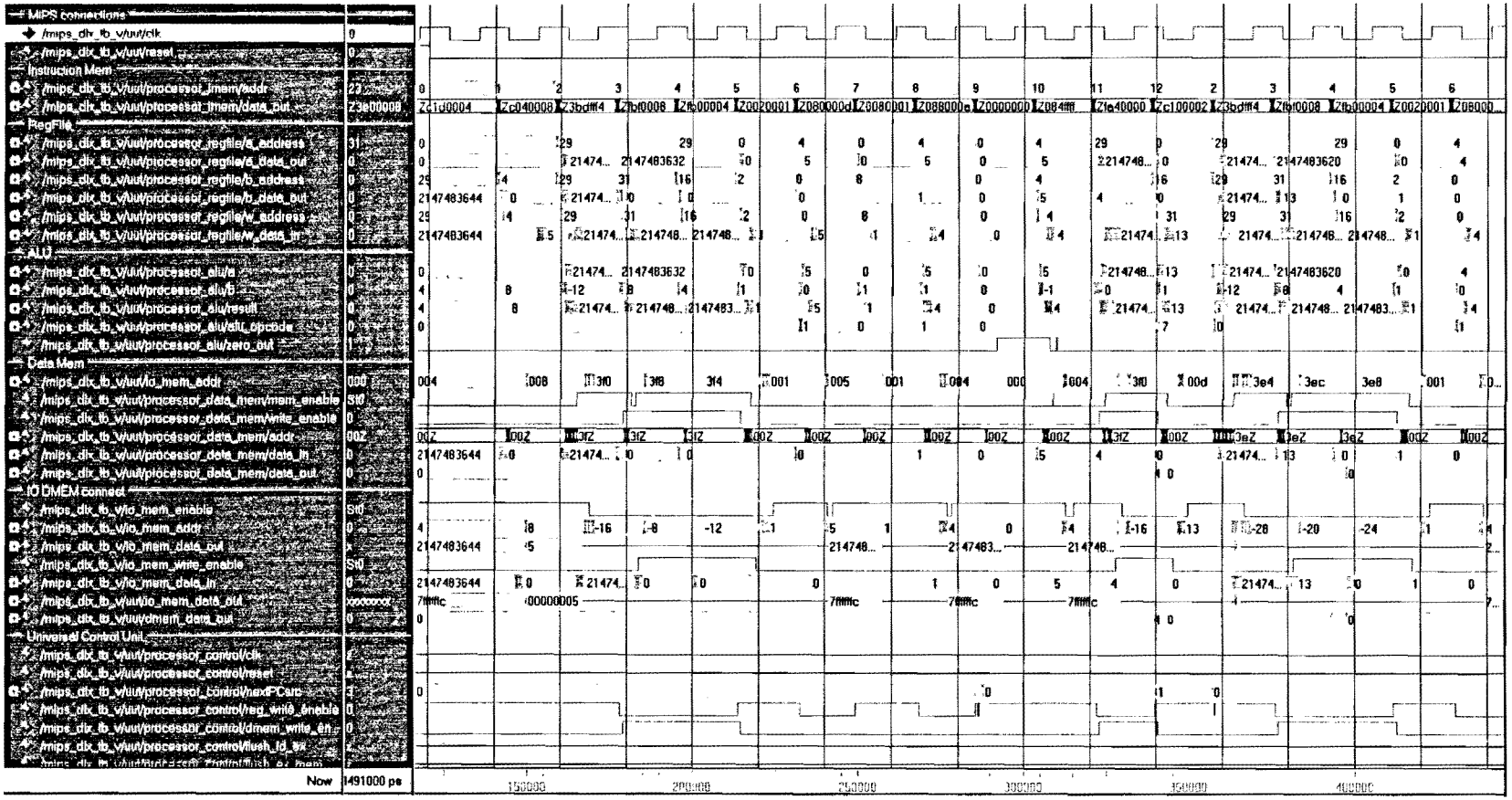


Figure G.2: Post-route simulation waveforms of the 32-bit configuration of one-cycle architecture

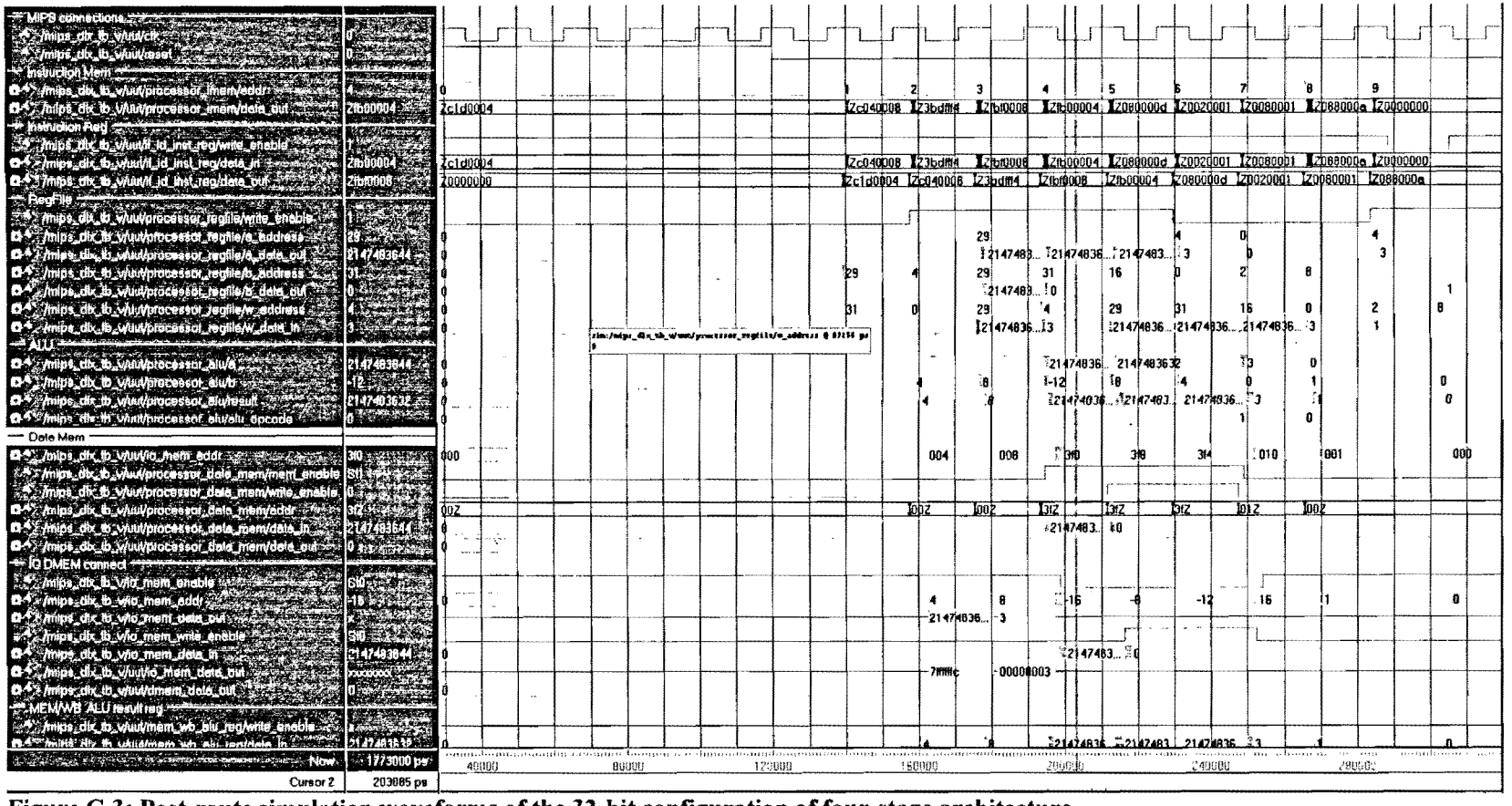


Figure G.3: Post-route simulation waveforms of the 32-bit configuration of four-stage architecture

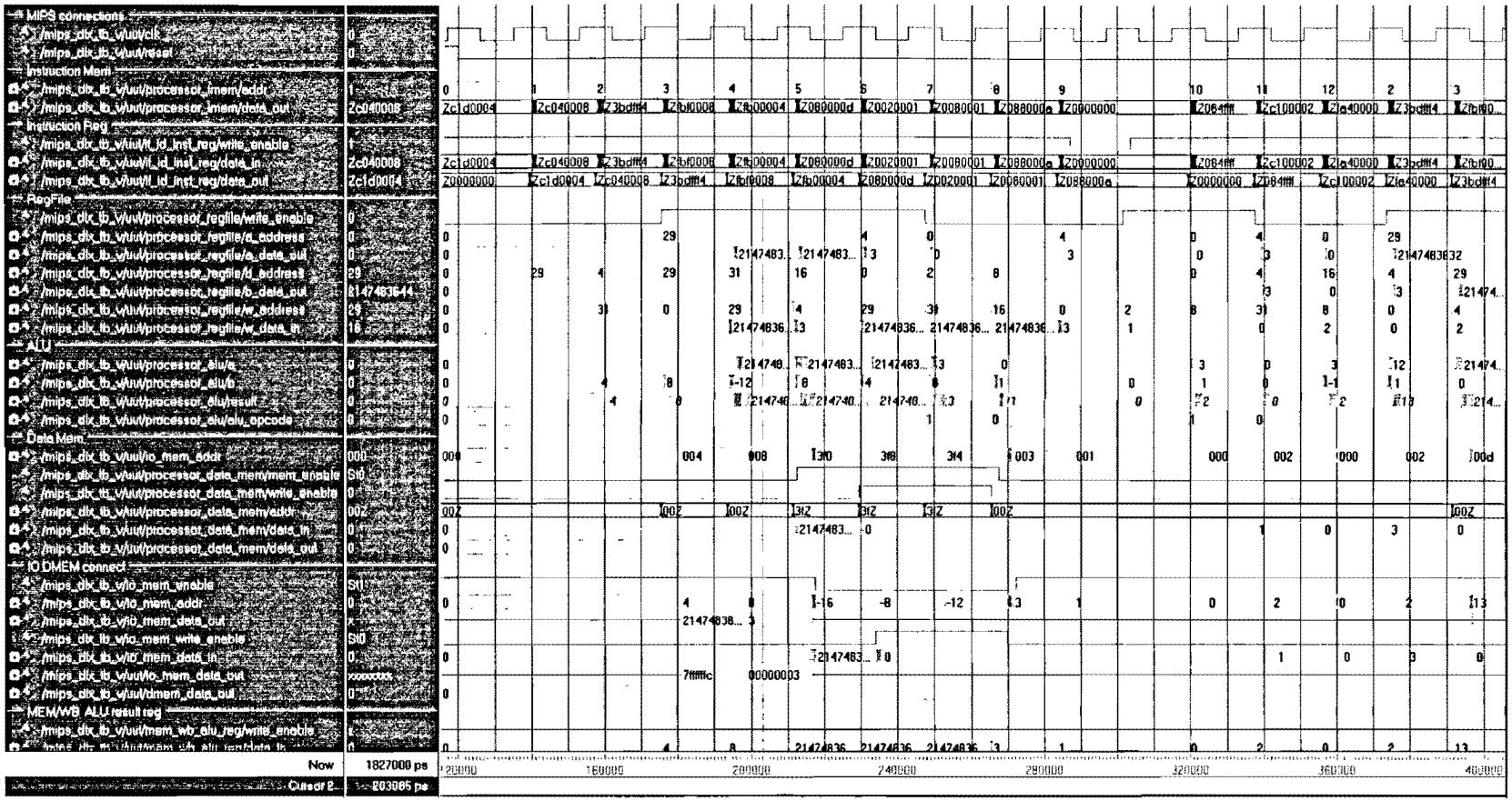
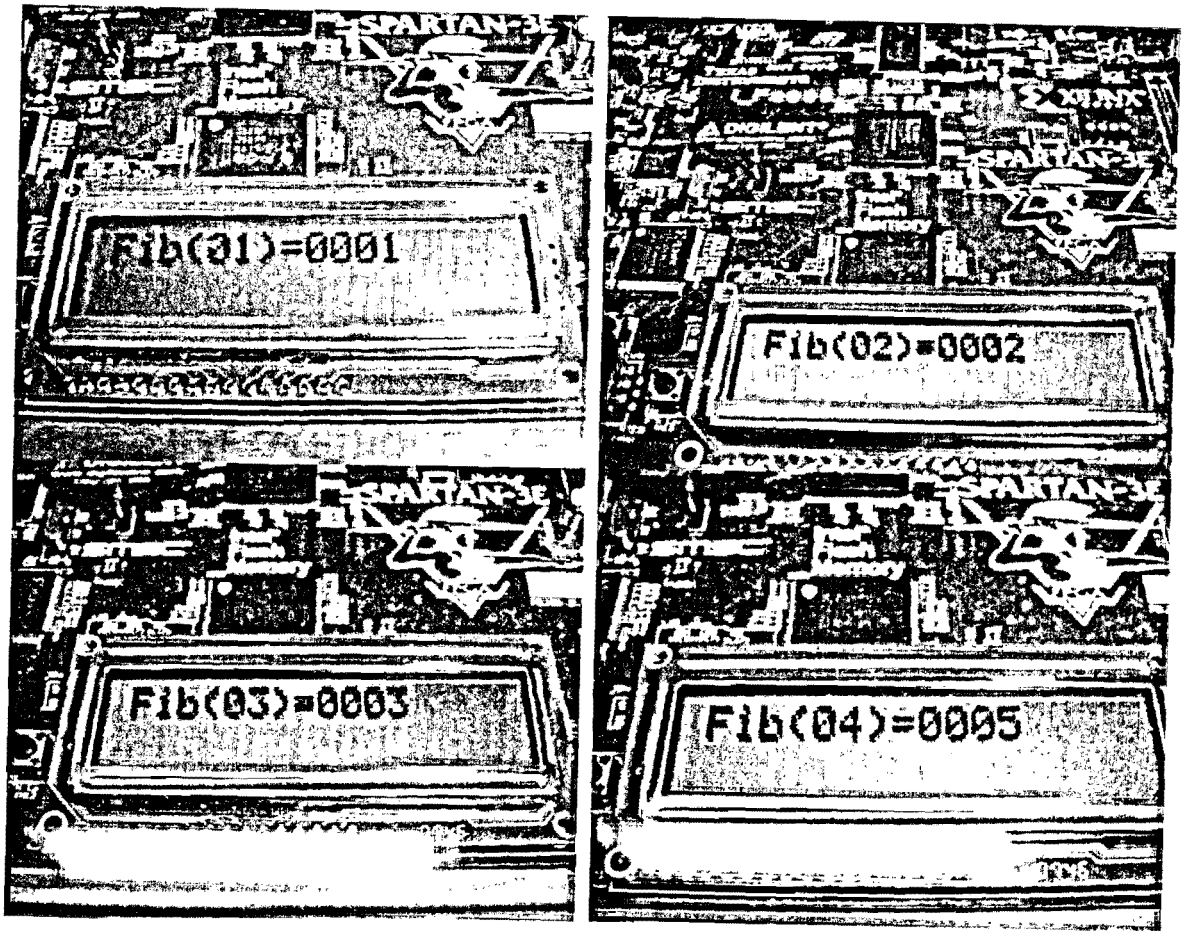


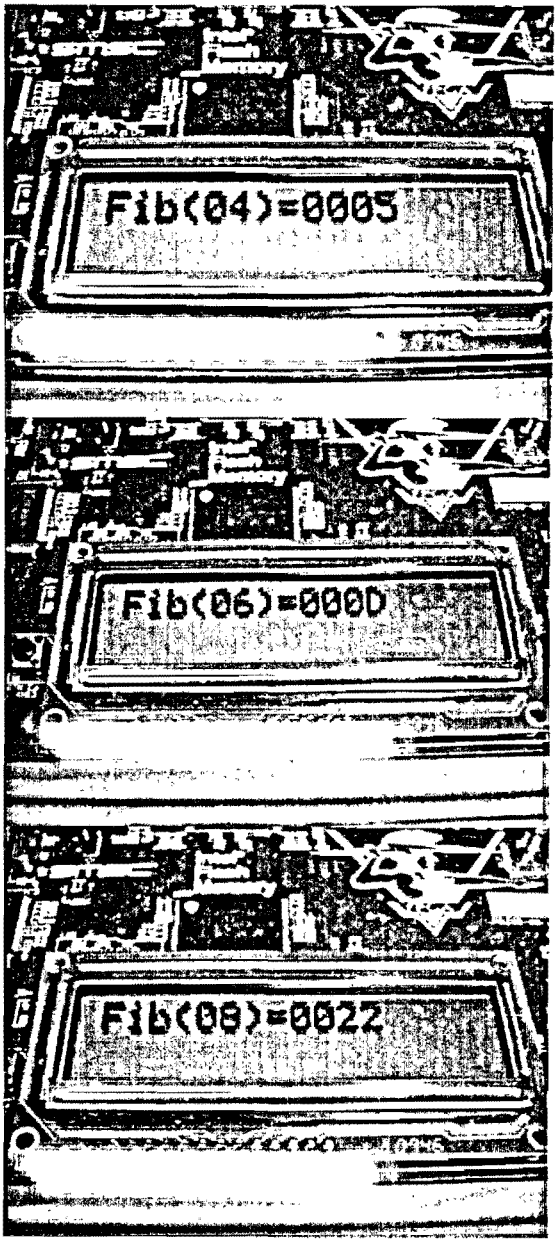
Figure G.4: Post-route simulation waveforms of the 32-bit configuration of five-stage architecture

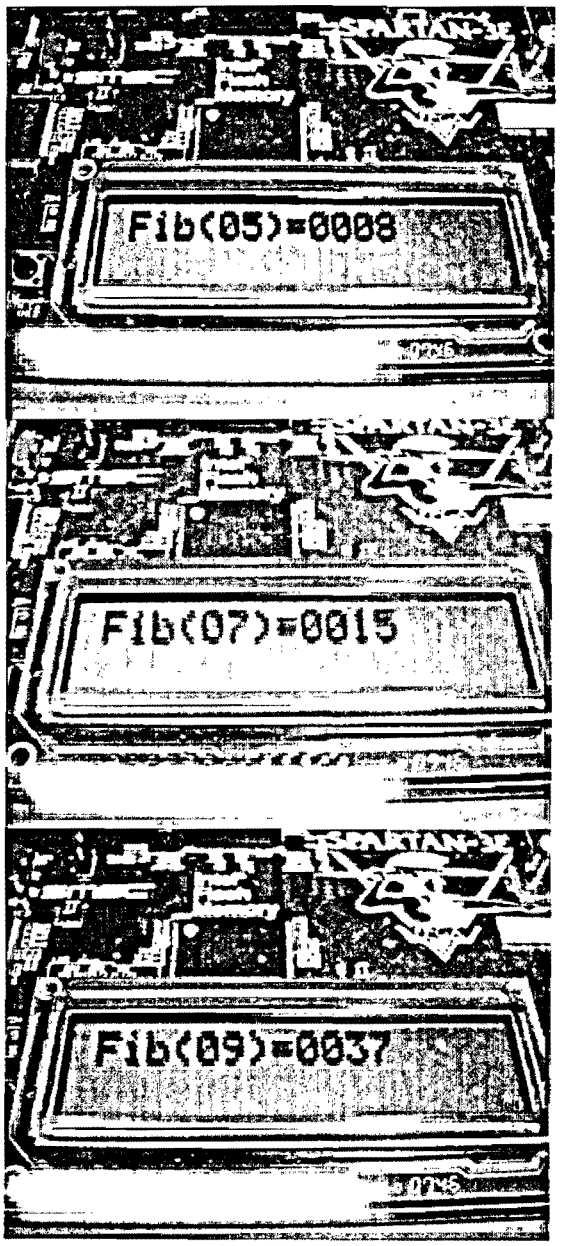
Appendix H

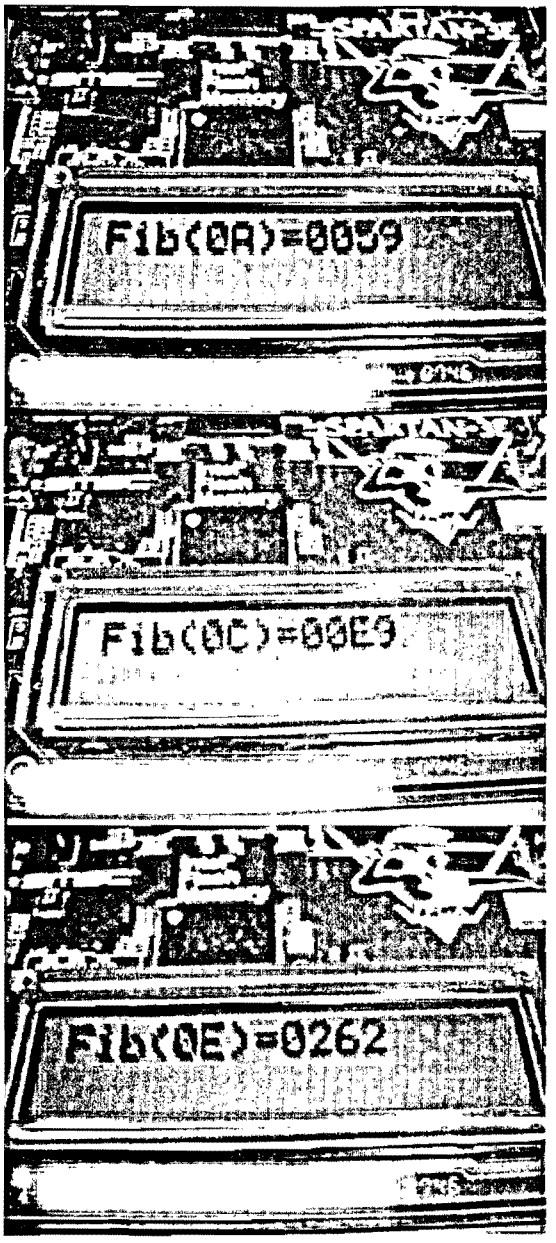
Images of Demo Design Example

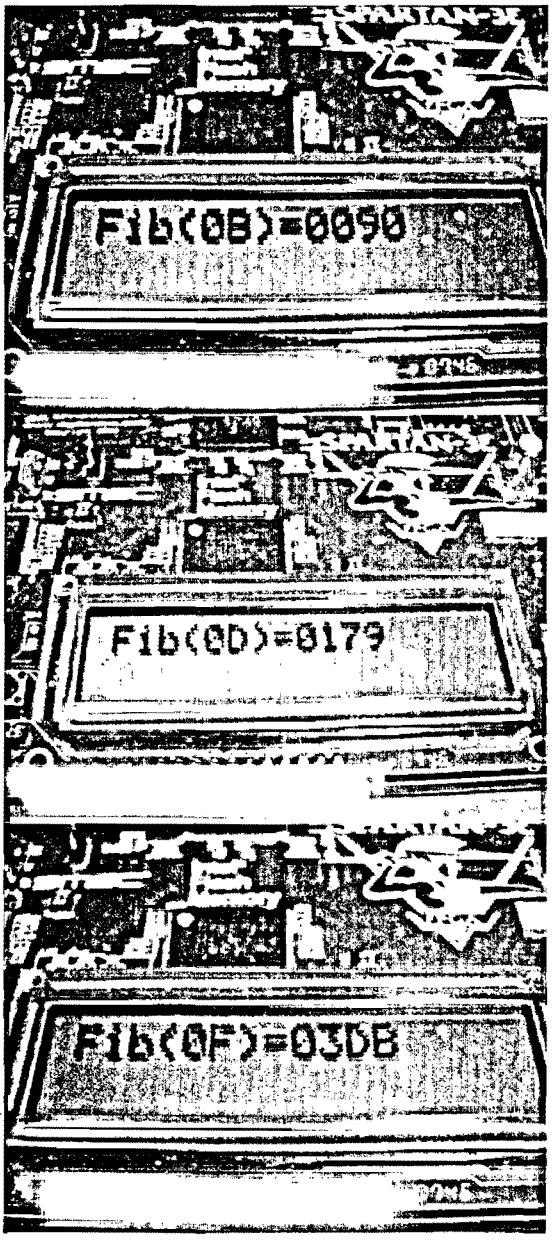
All numbers have a hexadecimal radix.




Fib(04)=0005
Fib(06)=000D
Fib(08)=0022


Fib(05)=0008
Fib(07)=0015
Fib(09)=0037


Fib(0A)=0059

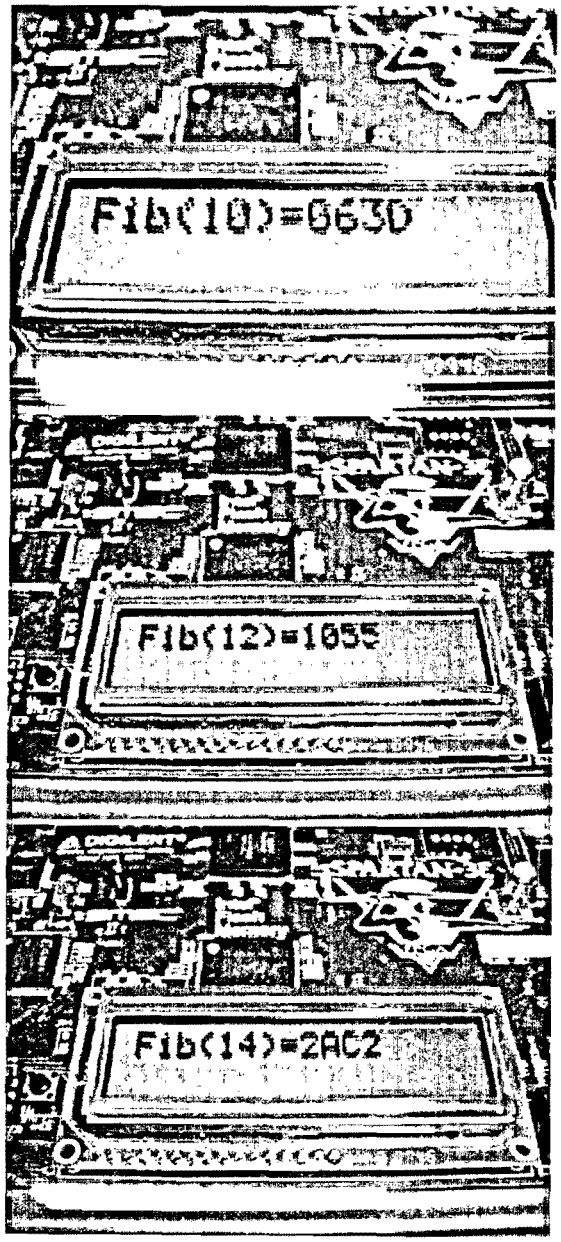

Fib(0B)=0090

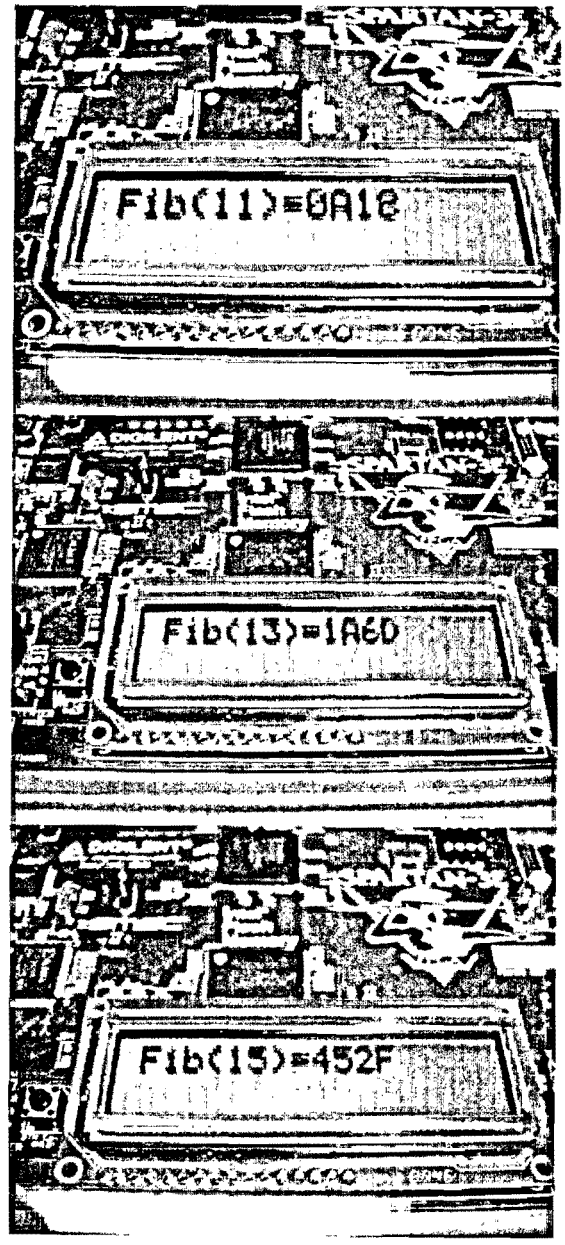
Fib(0C)=00E9

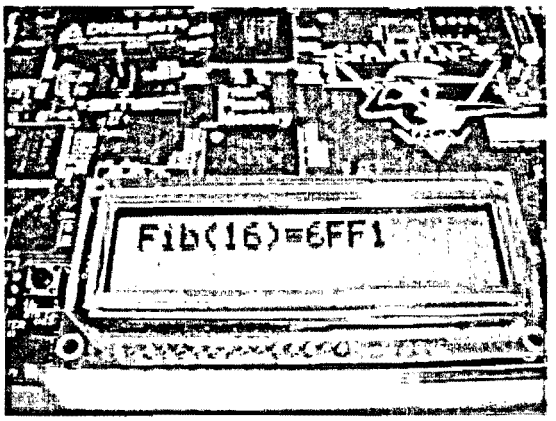
Fib(0D)=0179

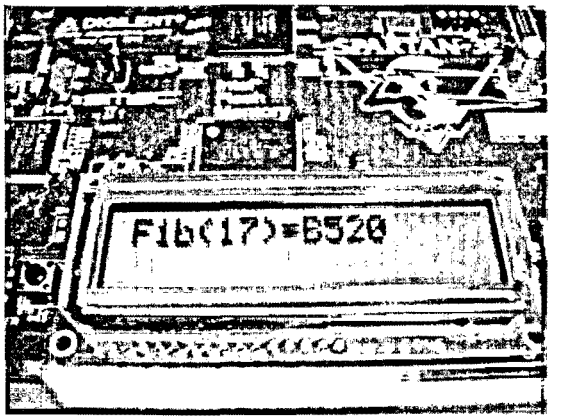
Fib(0E)=0262

Fib(0F)=03DB


Fib(10)=0630
Fib(12)=1055
Fib(14)=2AC2


Fib(11)=0A18
Fib(13)=1A6D
Fib(15)=452F


Fib(16)=6FF1


Fib(17)=B520

Appendix I

Altera FPGA Implemention

Excerpt from the Altera Ouartus II fitter report for the processor with following configuration:

- 5-stages pipelined
- 128-bit width
- Full ISA support
- BRAM optimization

; Fitter RAM Summary

Name	Type	Mode
data_ram:processor_data_mem|altsyncram:ram_rtl_2|altsyncram_f1j1:auto_generated|ALTSYNCRAM	AUTO	Simple Dual Port
inst_mem:processor_imem|altsyncram:Mux31_rtl_3|altsyncram_3571:auto_generated|ALTSYNCRAM	AUTO	ROM
regfile:processor_regfile|altsyncram:data_register_rtl_0|altsyncram_pti1:auto_generated|ALTSYNCRAM	MLAB	Simple Dual Port
regfile:processor_regfile|altsyncram:data_register_rtl_1|altsyncram_pti1:auto_generated|ALTSYNCRAM	MLAB	Simple Dual Port

Clock Mode	Port A Depth	Port A Width	Port B Depth	Port B Width	Port A Input Registers	Port A Output Registers
Dual Clocks	256	128	256	128	yes	no
Single Clock	1024	21	--	--	yes	no
--	32	128	32	128	yes	no
--	32	128	32	128	yes	no

Port B Input Registers	Port B Output Registers	Size	Implementation Port A Depth	Implementation Port A Width	Implementation Port B Depth
yes	no	32768	256	128	256
--	--	21504	1024	21	--
yes	no	4096	32	128	32
yes	no	4096	32	128	32

Implementation Port B Width	Implementation Bits	M9K blocks	M144K blocks	MLAB cells	MIF
128	32768	4	0	0	None
--	21504	3	0	0	mips_cfg.mips_dlx0.rtl.mif
128	4096	0	0	256	None
128	4096	0	0	256	None

Excerpt from the Altera Ouartus II fitter report for the processor with following configuration:

- 5-stages pipelined
- 128-bit width
- Full ISA support
- No BRAM optimization

; Fitter RAM Summary

Name	Type	Mode
data_ram:processor_data_mem|altsyncram:ram_rtl_2|altsyncram_f1j1:auto_generated|ALTSYNCRAM	AUTO	Simple Dual Port
regfile:processor_regfile|altsyncram:data_register_rtl_0|altsyncram_pti1:auto_generated|ALTSYNCRAM	MLAB	Simple Dual Port
regfile:processor_regfile|altsyncram:data_register_rtl_1|altsyncram_pti1:auto_generated|ALTSYNCRAM	MLAB	Simple Dual Port

Clock Mode	Port A Depth	Port A Width	Port B Depth	Port B Width	Port A Input Registers	Port A Output Registers
Dual Clocks	256	128	256	128	yes	no
--	32	128	32	128	yes	no
--	32	128	32	128	yes	no

Port B Input Registers	Port B Output Registers	Size	Implementation Port A Depth	Implementation Port A Width	Implementation Port B Depth
yes	no	32768	256	128	256
yes	no	4096	32	128	32
yes	no	4096	32	128	32

Implementation Port B Width	Implementation Bits	M9K blocks	M144K blocks	MLAB cells	MIF
128	32768	4	0	0	None
128	4096	0	0	256	None
128	4096	0	0	256	None

Bibliography

[1] P. Yiannacouras, JG Steffan, J Rose, "Soft vector processors vs FPGA custom hardware: measuring and reducing the gap", in *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, Monterey, California, USA, pp. 277-277, 2009

[2] I. Kuon, R. Tessier, J. Rose, "FPGA Architecture: Survey and Challenges", Foundations and Trends® in Electronic Design Automation,Vol. 2 , no. 2, pp. 135-253, Feb. 2008

[3] J. Nurmi, "Processor Design: System-On-Chip Computing for ASICs and FPGAs", Springer, 2007

[4] "HardCopy IV Device Handbook", Altera, Jul. 2009

[5] "IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (1364-2001)", IEEE Inc., 3 Park Avenue, New York, NY 10016-5997, USA, 2001

[6] J. L. Hennessy, N. P. Jouppi, J. Gill, F. Baskett, A. Strong, T. R. Gross, C. Rowen, J. Leonard, "The MIPS machine", in *Proceedings of 24th IEEE Computer Society International Conference*, IEEE Compcon, San Francisco, Feb. 1982, pp. 2-7.

[7] J. L. Hennessy, D. A. Patterson, "Computer Architecture: A Quantitative Approach", 4th edition, Morgan Kaufmann, 2007

[8] N. Pinckney, T. Barr, M. Dayringer, M. McKnett, N. Jiang; C. Nygaard, D. Money Harris, J. Stanley, B. Phillips, "A MIPS R2000 implementation", in *Proceedings of the 45th annual Design Automation Conference*, Anaheim, California, pp. 102-107, 2008

[9] "MIPS32® Architecture For Programmers Vol. I: Introduction to the MIPS32® Architecture", MIPS Technologies, Rev 2.60, June 25, 2008

[10] D. Sweetman, "See MIPS Run Linux", Morgan Kaufmann, 2nd ed., 2007

[11] H. Tago, K. Hashimoto, N. Ikumi, M. Nagamatsu, M. Suzuoki, Y. Yamamoto, "CPU for PlayStation(R)2", in *Proceedings of Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001*. p. 696, 13-16 March 2001

[12] J. L. Hennessy, D. A. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 4th edition, Morgan Kaufmann, 2008

[13] W. Stallings, "Computer Organization and Architecture: Designing for Performance", 8th edition, Pearson Education Inc, 2009

[14] Y. Liu, Z. Chen, Y. Chen, "How to Optimize the Cryptographic Symmetric Primitives on Loongson-2E Microprocessor", in *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications – Workshops*, pp. 608-614, 2008

[15] W. Hu, J. Wang, X. Gao, Y. Chen. "Microarchitecture of Godson-3 Multi-Core Processor", in *Proceedings of the 20th Hot Chips*, 2008.

[16] N. Seki, Lei Zhao, J. Kei, D. Ikebuchi, Yu. Kojima, Hasegawa Yohei, H. Amano, T. Kashima, S. Takeda, T. Shirai, M. Nakata, K. Usami, T. Sunata, J Kanai, M. Namiki, M. Kondo, H. Nakamura, "A fine-grain dynamic sleep control scheme in MIPS R3000", *IEEE International Conference on Computer Design*, pp. 612 – 617, Oct. 2008

[17] HG Kim, HC Oh, "A DSP-Enhanced 32-Bit Embedded Microprocessor", *Journal of Embedded Computing*, Springer, 2009

[18] J. Larus, "SPIM S20: A MIPS R2000 Simulator", University of Wisconsin-Madison, Tech. Rep., 2004

[19] R. Stallman and the GCC Developer Community, "Using the GNU Compiler Collection : For gcc version 4.4.2", GNU Press a division of the Free Software Foundation, 2008

[20] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML", in *Proceedings European Design and Test Conference, Paris*, France, pp. 503–507, Mar. 1995

[21] P. Mishra, N. Dutt , " Processor Description Languages", Morgan Kaufmann, 2008

[22] S. Basu, R.Moona, "High level synthesis from Sim-nML processor models", in *Proceedings of the 16th International Conference on VLSI Design*, pp. 255 – 260, Jan. 2003

[23] G. Hadjiyiannis, S. Hanono, S. Devadas, "ISDL: An Instruction Set Description Language for Retargetability", in *Proceedings of the 34th Design Automation Conference*, pp 299–302, June 1997

[24] G. Hadjiyiannis, P. Russo, S. Devadas, "Automatic architecture evaluation for hardware/software codesign", in *Proceedings of the 6th IEEE International Conference on Electronics, Circuits and Systems*, Vol. 1, pp. 47 – 53, Sept. 1999

[25] G. Hadjiyiannis, P. Russo, S. Devadas, "A methodology for accurate performance evaluation in architecture exploration", in *Proceedings of the 36th Design Automation Conference*, pp. 927 - 932, June 1999

[26] A. Halambi, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability", in *Proceedings on the Design, Automation and Test in Europe*, pp. 100–104, Mar. 1999

[27] A. Kejariwal et al. "HDLGen: Architecture Description Language driven HDL Generation for Pipelined Processors", CECS TR 03-04, University of California, Irvine, 2003.

[28] P. Mishra, N. Dutt, "Specification-driven directed test generation for validation of pipelined processors", *ACM Transactions on Design Automation of Electronic Systems*, Vol.13 , no. 3, July 2008,

[29] J. L. Hennessy, D, A. Patterson, "Computer Architecture : A Quantitative Approach", Morgan Kaufmann, 1997

[30] S. Pees, A. Hoffmann, V. Zivojnovic, H. Meyr, "LISA-machine description language for cycle-accurate models of programmable DSP architectures", in *Proceedings of the36th Design Automation Conference*, pp. 933 – 938, June 1999

[31] O. Schliebusch, H. Meyr, R. Leupers, "Optimized ASIP Synthesis from Architecture Description Language Models", Springer, 2007

[32] M. Hohenauer, R. Leupers , "C Compilers for ASIPs: Automatic Compiler Generation with LISA", Springer, 2009

[33] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language", *Computer-Aided Design of*

Integrated Circuits and Systems, IEEE Transactions, Vol. 20, no. 11, pp. 1338 - 1354, Nov. 2001

[34] H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, K. Karuri, J. Ceng, R. Leupers, G. Ascheid, H. Meyr, "ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study", in *Proceedings of the Workshop on Software and Compilers for Embedded Systems*, Vol. 6, no. 2, May 2007

[35] A.R. Jafri, A. Baghdadi, M. Jezequel, "Rapid Prototyping of ASIP-based Flexible MMSE-IC Linear Equalizer", *IEEE/IFIP International Symposium on Rapid System Prototyping*, pp.130-133, 2009

[36] A. Chattopadhyay, A. Sinha, D. Zhang, R. Leupers, G. Ascheid, H. Meyr, "Integrated Verification Approach during ADL-Driven Processor Design", *Microelectronics Journal*, no. 40, pp. 1111– 1123, 2009

[37] U. Meyer-Bäse, A. Vera, S. Rao, K. Lenk, M. Pattichis, "FPGA wavelet processor design using language for instruction-set architectures (LISA)", in *Proceedings SPIE*, Vol. 6576, Apr. 2007

[38] "CoWare® Processor Designer, Programmable Accelerators for Platform-Driven ESL Design", CoWare, 2006

[39] B. Sander, J. Schnerr, O. Bringmann, "ESL power analysis of embedded processors for temperature and reliability estimations", in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, Grenoble, France, pp. 239-248, 2009

[40] S. Yang, Y. Qian, Z. Tie-Jun, S. Rui, H. Chao-Huan, "A new HW/SW co-design methodology to generate a system level platform based on LISA", in *Proceedings of the 6th International Conference On ASIC*, Vol. 1, pp. 163 - 167, Oct. 2005

[41] A. Jain, "FPGA versus configurable processors: selecting the right device for your application", *IET Seminar on the Latest Technologies and Tools in Electronics Design*, Bangalore, India, p.3, Sept. 2008,

[42] M. Purnaprajna, M. Porrmann, U. Rueckert, "Run-time reconfigurability in embedded multiprocessors", *ACM SIGARCH Computer Architecture News*, Vol. 37 , no. 2, pp. 30-37, May 2009

[43] Y. Saito, T.Shirai, T. Nakamura, T. Nishimura, Y. Hasegawa, S. Tsutsumi, T. Kashima, M. Nakata, S. Takeda, K. Usami, "Leakage Power Reduction For Coarse Grained Dynamically Reconfigurable Processor Arrays With Fine Grained Power Gating Technique", *Field-Programmable Technology, International Conference*, pp. 329 - 332, Dec. 2008,

[44] H.P. Huynh, T. Mitra, "Runtime Adaptive Extensible Embedded Processors—A Survey", in *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, pp.: 215 - 225, 2009

[45] T. Halfhill, "Tensilica Tweaks Xtensa: Xtensa LX3 and Xtensa 8 Cores Boost Performance, Cut Power", Microprocessor Report, Tensilica, 2009

[46] T. Tohara, G. Ezer, "Multi-Standard Video Decoder Using Configurable Microprocessor Technology", *Consumer Electronics, Digest of Technical Papers. International Conference*, pp. 1 – 2, Jan. 2008

[47] "ARC® 700 Core Family", ARC International, Online, http://www.arc.com/configurablecores/arc700/, Retrieved on Jan. 2010

[48] "ARC® 600 Configurable Core Family ", Product brief, ARC International, 2007

[49] "ARChitect Processor Configuratior", Product brief, ARC International, 2007

[50] G. Brown, "Configurable Microprocessor Implementation of Low Bit Rate Audio Decoding", *Audio Engineering Society, Convention Paper, Presented at the 113th Convention*, Los Angeles, USA, Oct. 2005

[51] "MicroBlaze Processor Reference Guide", Xilinx, UG081 (v10.3), 2009

[52] "Virtex-5 FPGA Embedded Processor Block with PowerPC 440 Processor", Product Specification, DS621, Xilinx, 2009

[53] "PicoBlaze 8-bit Embedded Microcontroller User Guide, Spartan-6, Spartan-3, Virtex-6, and Virtex-5 FPGA Devices", UG129 (v1.2), Xilinx, November 11, 2009

[54] P. Bleyer, "Pacoblaze - a synthesizable behavioral verilog picoblaze clone," Online, January 2010, <http:// bleyer.org/pacoblaze/>

[55] "Nios II Processor Reference Handbook, ver 9.1", Altera Corporation, Nov. 2009

[56] F. Plavec, B. Fort, Z.G. Vranesic, S.D. Brown, "Experiences with Soft-Core Processor Design", in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, Washington, DC, USA, pp.167b-167b, Apr. 2005

[57] B. Fort, D. Capalija, Z. G. Vranesic, S. D. Brown, "A Multithreaded Soft Processor for SoPC Area Reduction", in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pp. 131-142, 2006

[58] O. Rayahi, M. Khalid, "UWindsor Nios II: A soft-core processor for design space exploration", in *Proceedings of IEEE International Conference on Electro/Information Technology*, pp. 451-457, June 2009

[59] J. Gaisler, S. Habinc, E. Catovic, "GRLIB IP Library User's Manual", Gaisler Research, 2009

[60] M.-A. Daigneault, J.M.P Langlois, J.P. David, "Application Specific Instruction Set Processor Specialized for Block Motion Estimation", in *Proceedings of IEEE International Conference on Computer Design,* pp. 266-271, Oct. 2008

[61] J. Lee, J. Lee, M. Jeong, N. Eum, S. Park, "A 100MHz ASIP (application specific instruction processor) for CAVLC of H.264/AVC decoder", in *Proceedings of IEEE International Symposium on Circuit and Systems, ISCAS 2008*, Seattle, WA, pp. 3462 - 3465, May 2008

[62] N. Sonmez, A. Yurdakul, "SIxD:A Configurable Application-Specific SISD/SIMD Microprocessor Soft-Core", in *Proceedings of International Symposium on System-on-Chip*, Tampere, pp. 1-4, Nov. 2006

[63] R. Dimond, O. Mencer, W. Luk, "Application-specific customisation of multi-threaded soft processors", *IEE Proceedings Computers & Digital Techniques,* Vol. 153, no. 3, pp.173 - 180, May 2006

[64] P. Yiannacouras, J. G. Steffan, J. Rose, "Exploration and Customization of FPGA-Based Soft Processors", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, no. 2, pp. 266-277, Feb. 2007

[65] "BlockRAM (BRAM), Block(v1.00a)", Product Specification DS444, Xilinx, Dec. 2009

[66] "XST User Guide 9.2i", Xilinx, 2007

[67] "ColdFire® Family Programmer's Reference Manual", Freescale Semiconductor, Rev. 3, Mar. 2005

[68] "ModelSim® SE User's Manual: Software Version 6.2g", Mentor Graphics, 2007

[69] "Virtex-5 Family Overview: Product Specification", DS100 (v5.0), Xininx, Feb. 2009

[70] "Stratix III FPGAs vs. Xilinx Virtex-5 Devices: Architecture and Performance Comparison", White paper, ver. 2.1, Altera Corporation, Oct. 2007

[71] A. Percey, "Advantages of the Virtex-5 FPGA 6-Input LUT Architecture", White Paper, WP284 (v1.0), Xilinx, Dec. 2007

[72] "Digital IC Design Flow: A Tutorial on RMC's Digital Design Flow (based on CMOSP18 Artisan) , ver. 5.0", RMC Microelectronic Lab, Jan. 2008

[73] "Digital IC Design Flow: Tutorial. Document ICI-134", CMC, July 2004

[74] "Spartan-3E FPGA Starter Kit Board, User Guide", Xilinx, UG230 (v1.1) June 20, 2008

[75] A. Posamentier, I. Lehmann "The (Fabulous) FIBONACCI Numbers", Prometheus Books, pp. 305, 2007

[76] M. Brorsson, "MipsIt: A simulation and development environment using animation for computer architecture education", in *Proceedings of the 2002 workshop on Computer architecture education*, Anchorage, AK, pp. 65-72, May 2002.

[77] K.Chapman, "Rotary Encoder Interface for Spartan-3E Starter Kit," Xilinx, Feb. 20, 2006

[78] A. Ziebinski, S. Swierc, "The VHDL Implementation of Reconfigurable MIPS Processor", Book Chapter, "Man-Machine Interactions", Springer Berlin/Heidelberg, 2009

[79] R. Kastner, "Computer Organization", ECE 15B - Spring 2006, Lecture 8, April 27, 2006, http://www.ece.ucsb.edu/~kastner/ece15b/slides/lecture08.pdf

[80] "Quartus II Handbook Version 9.1 Vol. 1: Design and Synthesis", Altera Corporation, November 2009

[81] "Stratix III Device Handbook, Vol. 1: TriMatrix Embedded Memory Blocks in Stratix III Devices", Altera Corporation, May 2009

[82] T. R. Halfhill, "MicroBlaze v7 Gets an MMU: Memory Manager Brings Full-Fledged Linux to Xilinx Processor Core", *Xcell Journal*, 2008

[83] "SPARC V8 32-bit Processor LEON3 / LEON3-FT: CompanionCore Data Sheet, Version 1.0.3", Aeroflex Gaisler AB, December 2008