PIPELINED DESIGN APPROACH TO MICROPROCESSOR ARCHITECTURES
A PARTIAL IMPLEMENTATION: MIPS™ PIPELINED ARCHITECTURE ON FPGA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUZAFFER CAN ALTINİĞNELİ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

DECEMBER 2005

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. İsmet ERKMEN
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Hasan GÜRAN
Supervisor

Examining Committee Members

Assist. Prof. Dr. Cüneyt BAZLAMAÇCI	(METU, EE)	
Prof. Dr. Hasan GÜRAN	(METU, EE)	
Dr. Ece (GÜRAN) SCHMIDT	(METU, EE)	
Assist. Prof. Dr. İlkay ULUSOY	(METU, EE)	
M.S. Eng. Murat ŞANSAL	(ASELSAN)	

ABSTRACT

PIPELINED DESIGN APPROACH TO MICROPROCESSOR ARCHITECTURES
A PARTIAL IMPLEMENTATION: MIPS™ PIPELINED ARCHITECTURE ON FPGA

ALTINİĞNELİ, Muzaffer Can
M.S, Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Hasan GÜRAN

September 2005, 120 Pages

This thesis demonstrate how pipelining in a RISC processor is achieved by implementing a subset of MIPS R2000 instructions on FPGA. Pipelining, which is one of the primary concepts to speed up a microprocessor is emphasized throughout this thesis. Pipelining is fundamentally invisible for high level programming language user and this work reveals the internals of microprocessor pipelining and the potential problems encountered while implementing pipelining. The comparative and quantitative flow of this thesis allows to understand why pipelining is preferred instead of other possible implementation schemes. The methodology for programmable logic development and the capabilities of programmable logic devices are also given as background information. This thesis can be the starting point and reference for programmers who are willing to get familiar with microprocessors and pipelining.

Keywords: Microprocessor, MIPS, Pipelining, FPGA

ÖZ

MİKRO İŞLEMCİLERDE PIPELINED DİZAYN YAKLAŞIMI
MIPS™ PIPELINED İŞLEMCİ MİMARİSİNİN FPGA ÜZERİNDE KISMI BİR UYGULAMASI

ALTINİĞNELİ, Muzaffer Can
Yüksek Lisans, Elektrik Elektronik Mühendisliği
Tez Yöneticisi: Prof. Dr. Hasan GÜRAN

Eylül 2005, 120 Sayfa

Bu çalışmada, RISC işlemcilerde "Pipelining" konusu, FPGA üzerinde MIPS R2000 komut setinin bir kısmı tamamlanarak açıklanmıştır. Çalışma boyunca, Mikro İşlemcilerin hızlarının arttırılması konusunda temel bir unsur olan "Pipelining" konusu üzerinde durulmuştur. Temel olarak "Pipelining" işlevi, yüksek seviyede programlama yapan kişilere görünmezdir. Bu çalışma "Pipelining" işlevinin ayrıntılarını ve bu işlev gerçekleştirilirken karşılaşılan problemleri ortaya koymaktadır. "Pipelining" dışındaki diğer tasarım yaklaşımlarının neden uygulanamaz oldukları, bu tezin karşılaştırmalı ve nicel akışı sayesinde anlaşılabilir. Donanım tasarımında temel alınan metodolojiler ve donanımların kabiliyetleri hakkında tez boyunca bir alt yapı oluşturulmaya da çalışılmıştır. Bu tez, Mikro İşlemciler ve "Pipelining" işlevi ile tanışıklık kazanmak isteyen programcılar için bir başlangıç ve referans noktası olabilir.

Anahtar Kelimeler: Mikro İşlemci, MIPS, Pipeline, FPGA

To My Generous Family

ACKNOWLEDGMENTS

I owe much gratitude to my Advisor, Professor Dr. Hasan Güran, for inspiring me to carry out this thesis. His criticism and suggestions bring this work to this point and I am always aware during our work that this thesis is first of all for my benefit.

Everyone working at ASELSAN deserve my thanks, especially Erdinç Atılgan, Kemal Burak Codur and Murat Şansal. They guided me to right technical people, supported me technically and mentally during my work. This thesis ended up with an implementation because ASELSAN gave the hardware support without waiting any outcome despite it is a commercial organization.

I also owe lots to my father, mother and sister. They interested in all of my needs while I was embedded to my work. I also grasp the idea of being a family in addition to fundamentals of pipelining in micro-processors at the end of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
API	Application Interface
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CLK	CLocK
CPI	Clock cycle Per Instruction
CPLD	Complex Programmable Logic Device
DLL	Delay Locked Loop
EX	Execute (stage)
FF	Flip Flop
FPGA	Field Programmable Gate Array
GCC	Gnu C Compiler
GPR	General Purpose Register
HDL	Hardware Description Language
ID	Instruction Decode (stage)
IF	Instruction Fetch (stage)
IOB	Input Output Block
ISA	Instruction Set Architecture
ISE	Integrated Software Environment
LUT	Look Up Table
MEM	Memory (stage)
MIPS	Microprocessor without Interlocked Pipeline Stages
MUX	MUltiplXer
NOP	No Operation (instruction)
PAL	Programmable Array Logic
PC	Program Counter
PCB	Printed Circuit Board
PLA	Programmable Logic Array
PROM	Programmable Read Only Memory
RISC	Reduced Instruction Set Computer
SoRC	System on Re-programmable Chip
SPLD	Simple Programmable Logic Device
VHDL	Very high speed integrated HW Description Language
WB	Write Back (stage)
XST	Xilinx Synthesis Technology

CHAPTER 1

INTRODUCTION

Faster execution of computer programs was the one of the most challenging concerns of engineers in the past and also will be much more challenging in the future. Increased demands of the industry for real time applications yield the presence of faster and deterministic processor architectures in years in the market.

Developers have always been under the effect of their era's restrictions while determining their architectural approach. This was the reason why Complex Instruction Set based computers (CISC) came before the much simpler counter parts, the Reduced Instruction Set (RISC) based computers. Developers constructed first more challenging CISC because of memory restrictions and little compiler support. Developments in memory technology in parallel with compiler enhancements resulted in emergence of RISC based computers. They are much simpler to build, much simpler to understand; hence open for improvements and maintenance.

The number of high level programming language compilers developed and specialized for RISC architectures grew rapidly. High level programming became more popular over years and programmers kept away from low level error prone long lasting assembly programming. Another reason for choosing high level programming is that different vendors proposed different architectures; hence it was not feasible to learn the architecture specific assembly code. Pipelining is one way of increasing the processor's performance. It was proposed for RISC based computers mainly because of their regularity. Pipelining accompanied with improved compiler support gave superior performance and further improvements made by scaling these architectures.

The primary goal of this thesis is to grasp the idea behind pipelining by partially developing RISC architecture, specifically Microprocessor without Interlocked Pipeline Stages (MIPS), because of its simplicity and rich documentation.

Understanding the pipelining is important because pipelining is transparent to high level programmer. Programmers are aware of Program Counter (PC), register bank and memory when they debug their programs, but they can not observe the internal register blocks used for pipelining. Programmers can not understand why the assembly code generated by different compiler vendors is different for the same high level software without knowing the internals of pipelining even they know the compiler well.

The secondary goal of this thesis is to understand the problems faced in pipelining, because it is the first step that comes before the superscalar speculative architectures. To go one step further, problems in pipelining must be solved.

The last goal of this thesis is to get familiarity with hardware design process cycle and grasp internals of programmable logic design especially for Field Programmable Gate Arrays (FPGAs). FPGAs promise parallelism which is the key concept for speed. FPGAs are reprogrammable and are becoming more popular in the market. They replace to application specific integrated circuit (ASIC) and discrete processors and they are also called as system on reprogrammable chip (SoRC).

This thesis is organized as follows: Chapter 2 serves to provide necessary background for development environment, programmable logic design and FPGAs. Chapter 3 describes the different implementation schemes for the same instruction set and clarifies why pipelining is the best quantitatively. It also describes the problems encountered in pipelining and solution proposals. Chapter 4 gives the details of particular subset of MIPS implementation. Chapter 5 is devoted for formal verification of the partially implemented architecture by using in circuit debugging at runtime via specially developed software, MIPS Monitor. Chapter 6 gives the conclusions and makes remarks for further future work. The appendices presents the implemented instruction set assembly codes, instruction descriptions and some screen shots to demonstrate the usage of MIPS Monitor software.

CHAPTER 2

BACKGROUND AND MOTIVATION

This chapter serves for the following purposes:

(1) providing the necessary background for understanding the rest of thesis,

(2) motivations behind the usage of software and hardware development environments in thesis,

(3) internals of platform FPGA which was preferred as design solution,

Readers, who are quite familiar with these concepts, can skip this chapter and start reading Chapter 3 first.

2.1. Programmable Logic Design

Since late 1970s, programmable logic circuits are greatly enhanced and dominated the electronics market. Developers had a tendency to use reprogrammable devices (simple and complex programmable logic devices), instead of application specific integrated circuits (ASIC) to develop large and interoperable systems because of their following characteristics [XDRM99]:

- Low cost per gate.
- Reduces Risk; engineers can make design changes in minutes.

- Faster Testing and Manufacturing.
- Ease in Verification.
- Ability participating in Hardware-Software Co-Design.
- Versatile support for Input/Output Standards.

2.1.1. History of Programmable Logic

By the late 1970s, standard logic components were exclusively used as standard building blocks of logic circuits. These components (e.g., 74XX series TTL parts) were located on printed circuit boards (PCBs) and any change in logic resulted corresponding revision in PCB layout. The side effects encountered, when some part of design changed, was able to be avoided by replacing these components with programmable logic devices (PLDs). Given that the design in PLDs was flexible, no rewiring on PCBs was required. In addition, less board area and power was consumed by PLDs. PLDs can be divided in two sets as simple and complex PLD.

2.1.1.1. Simple Programmable Logic Device (SPLD)

These devices are mainly used for address decoding [Barr99].

2.1.1.1.1. Programmable Logic Array (PLA)

Ron Cline from Signetics™ put forward the idea of two programmable planes on 1975 [XPM04]. Any combinatorial logic can be expressed in the form of two level logics: as product of sums or sum of products. For that reason, by using PLA, any combinational logic can be implemented, if number of inputs and outputs are enough for required implementation. Despite the architecture is very flexible, because of high fuse count, propagation delay is higher than PAL. Unwanted connections (fuse) are blown after programming.

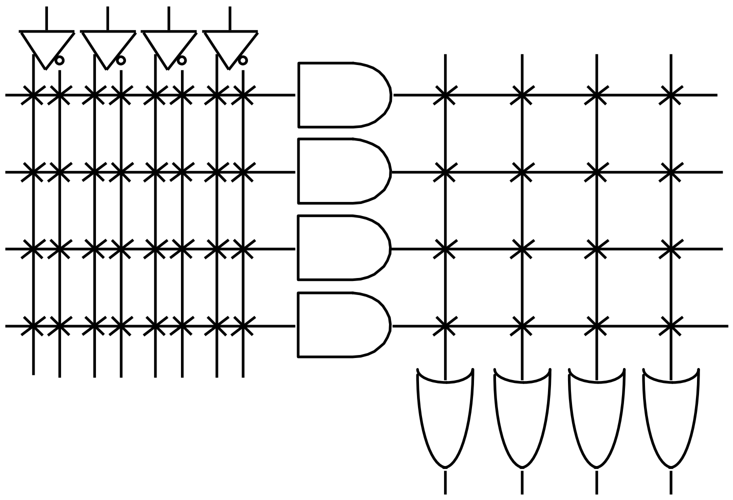


Figure 2.1: PLA Architecture

2.1.1.1.2. Programmable Array Logic (PAL)

John Birkner from MMI proposed a second alternative for the PLA array on 1978. Instead of one programmable planes, the OR array was fixed after fabrication [XPM04]. PALs are more constrained than PLAs, but, because of fewer connections, they have lower propagation delay.

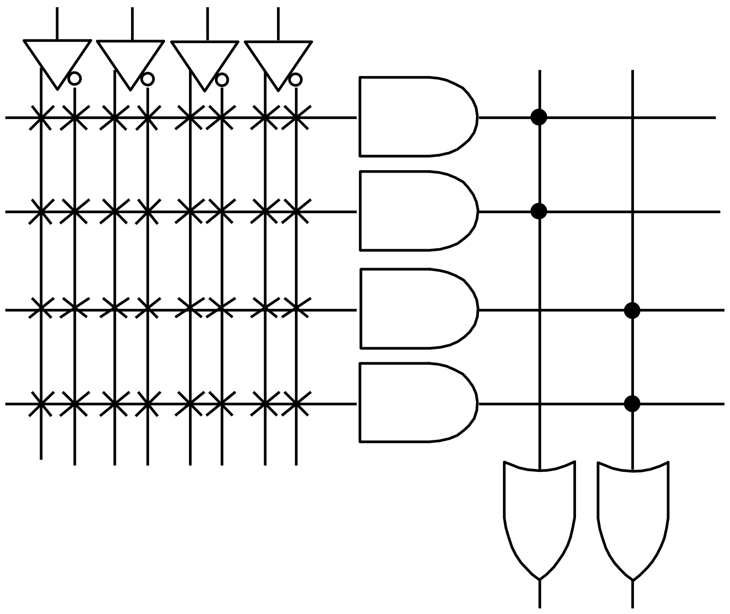


Figure 2.2: PAL Architecture

2.1.1.2. Complex Programmable Logic Device (CPLD)

Macrocells were obtained by extending PLDs with additional flip flops (FFs). CPLDs were simply combinations of these macrocells with programmable interconnects, switch matrix (SM). SM within CPLD may or may not be fully connected unlike the programmable interconnect within PLD. In other words, some of theoretically possible connections between PLDs may not actually be supported within a given CPLD. Therefore 100% utilization of macrocells is very difficult to achieve. Some designs will not fit a given CPLD, even though there are sufficient logic gates and FFs.

CPLDs can also be used as address decoders like PLDs, but more often as high performance control logic and finite state machines. Traditionally, CPLDs have been chosen over FPGAs, whenever high performance logic is required [Barr99].

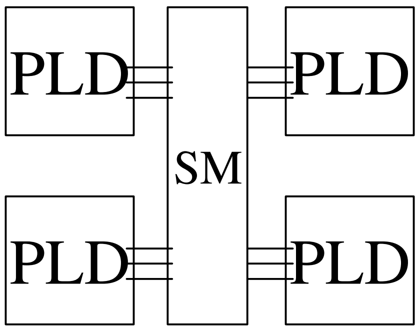


Figure 2.3: CPLD Architecture

2.1.1.3. Field Programmable Logic Gate Array (FPGA)

In 1985, a company called Xilinx™ introduced FPGAs, composed of configurable logic blocks (CLBs), which are surrounded by programmable interconnects and comprise function generators or look up tables (LUTs) and flip flops (FFs). FPGAs can be one time programmable similar to PLD or SRAM based (or reprogrammable). [XPM04] [TRENZ01] [BZEID]

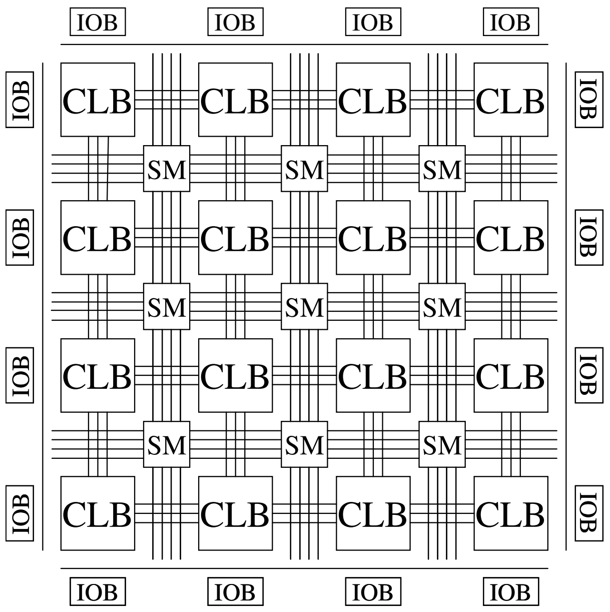


Figure 2.4: FPGA Architecture

2.1.2. Basic Design Process

Design entry or design specification can be in the form of schematic capture or hardware description language (HDL). In schematic form, after determining the capture tool and the manufacturer's library, designer can connect the gates from library with wires and then generates netlist, which is the textual description of the circuit. Schematic capture is not feasible for large designs because it is not scalable, not reusable, strongly vendor dependent and hard to maintain. In HDL design entry, the design is entered in high level description language emphasizing design's function or behavior and then synthesized by the vendor independent tool and netlist is generated. The design is more maintainable, scalable and reusable than schematic design entry.

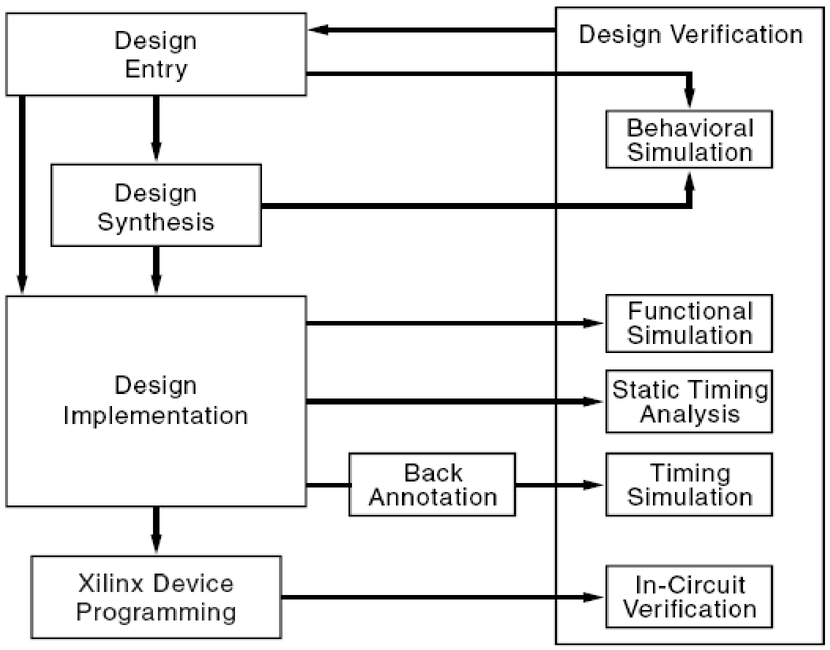


Figure 2.5: Basic Design Flow in FPGAs, ©Xilinx

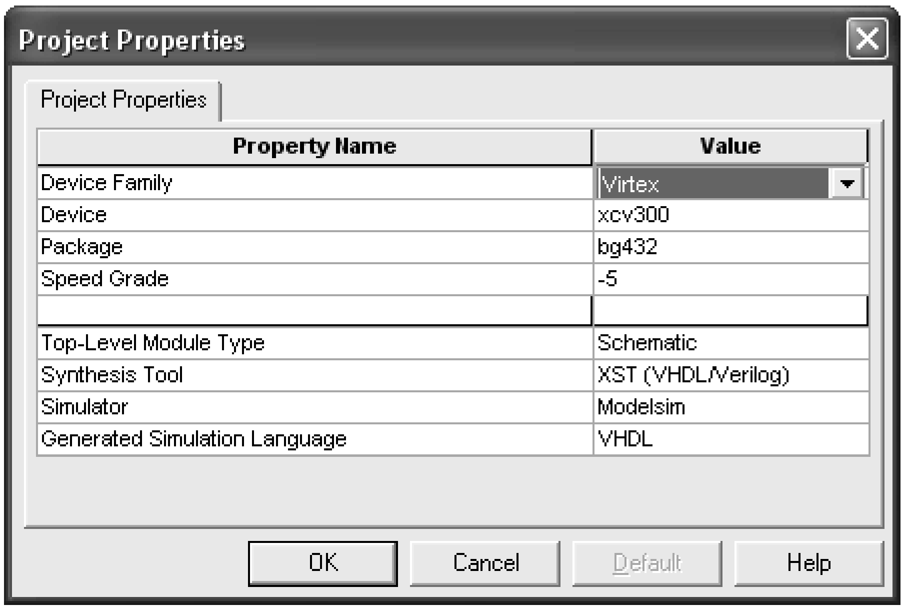
In design implementation, the first step is translation of low level and generic netlist file into device specific resources. After translation step, mapping step checks the design according to device specific rules, add further logic or make replications to meet the timing requirements using device resources. At last, in place and route step, already allocated resources are distributed along FPGA taking into account the physical constraints and routing resources. At this point physical layout is determined and timing information for design entities and interconnects (Back Annotation) is available. After routing, the device is ready to be programmed.

In device programming stage, the SRAM based FPGA's configuration, which is volatile after power on and also defining the logic and interconnect, is programmed to a Programmable Read Only Memory (PROM) device with part name xc18v02.

Design verification is a parallel process to design development. Design entry in either schematic or HDL form can be simulated behaviorally, while it can be tested based on the code syntax. After synthesis phase, generated netlist format can be simulated functionally by providing test vectors and tested by checking the desired output vector. Timing simulation comes after the place and route phase using back annotation.

2.2. Integrated Software Environment (ISE™)

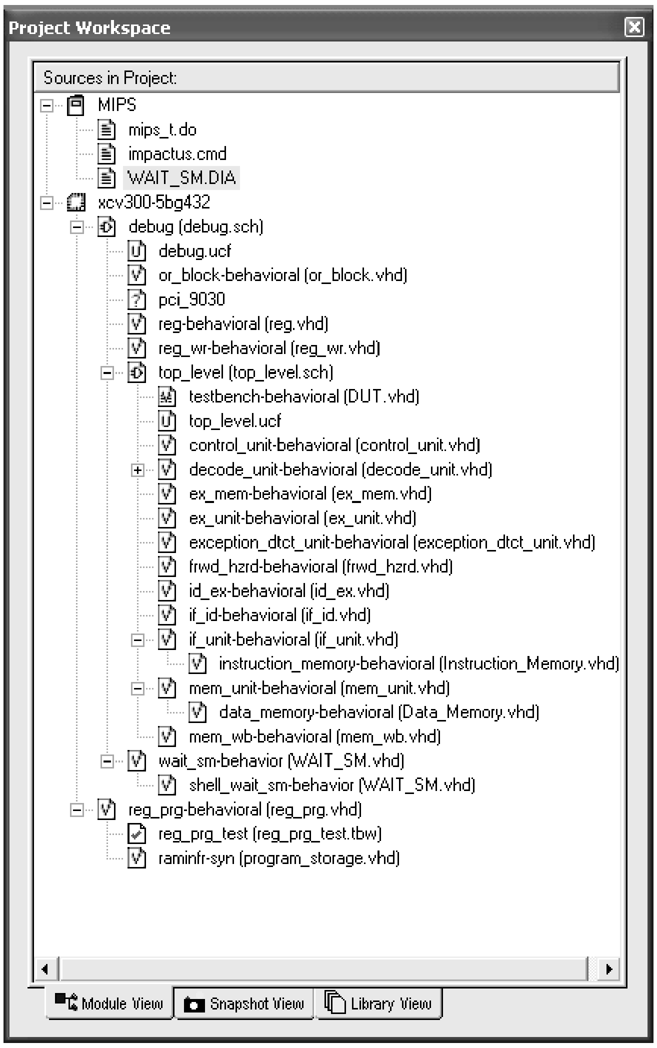
Integrated Software Environment is the environment provided by Xilinx™ for Design Entry, Design Synthesis, Design Implementation, Design Verification and Device Programming phases (described in 2.1.2) of design development [XISE03]. MIPS project was created in ISE with the project properties given in Figure 2.6.



Property Name	Value
Device Family	Virtex
Device	xcv300
Package	bg432
Speed Grade	-5
Top-Level Module Type	Schematic
Synthesis Tool	XST (VHDL/Verilog)
Simulator	Modelsim
Generated Simulation Language	VHDL

Figure 2.6: MIPS Project Properties Window

Top-Level module for Design Entry is selected as Schematic Capture for visualization purposes. All other sub-modules are coded in hardware description language VHDL [CDVHDL] [Perry02]. XST (Xilinx Synthesis Technology) tool was used to synthesize netlist from VHDL code. Modelsim® simulator was selected for post-place and route simulation purposes.



```
Project Workspace
Sources in Project:
MIPS
    mips_t.do
    impactus.cmd
    WAIT_SM.DIA
xcv300-5bg432
    debug (debug.sch)
        debug.ucf
        or_block-behavioral (or_block.vhd)
        pci_9030
        reg-behavioral (reg.vhd)
        reg_wr-behavioral (reg_wr.vhd)
        top_level (top_level.sch)
            testbench-behavioral (DUT.vhd)
            top_level.ucf
            control_unit-behavioral (control_unit.vhd)
            decode_unit-behavioral (decode_unit.vhd)
            ex_mem-behavioral (ex_mem.vhd)
            ex_unit-behavioral (ex_unit.vhd)
            exception_dtct_unit-behavioral (exception_dtct_unit.vhd)
            frwd_hzrd-behavioral (frwd_hzrd.vhd)
            id_ex-behavioral (id_ex.vhd)
            if_id-behavioral (if_id.vhd)
            if_unit-behavioral (if_unit.vhd)
                instruction_memory-behavioral (Instruction_Memory.vhd)
            mem_unit-behavioral (mem_unit.vhd)
                data_memory-behavioral (Data_Memory.vhd)
            mem_wb-behavioral (mem_wb.vhd)
        wait_sm-behavior (WAIT_SM.vhd)
            shell_wait_sm-behavior (WAIT_SM.vhd)
    reg_prg-behavioral (reg_prg.vhd)
        reg_prg_test (reg_prg_test.tbw)
        raminfr-syn (program_storage.vhd)
```

Figure 2.7: MIPS Project Source File Listing

MIPS project comprise source files describing the architecture of entities which are listed in (Figure 2.7) for the following purposes;

- Design Entry (e.g. file extensions *.vhd and *.sch)
- Physical and Timing user constraints files for Design Implementation (e.g. file extension *.ucf)
- Test Bench files for Post-Place and Route Simulation (e.g. file extensions *.vhd)

- Post-Place and Route simulation macro file which compiles the design and Test Bench files, invokes the simulator, loads signals to view windows and runs the simulation for specified time duration. (e.g. file extension *.do)
- State Machine editor file (e.g. file extension *.dia)
- Impactus command file for device programming (e.g. file extension *.cmd)

2.3. Virtex™ FPGA

MIPS project is implemented on an xcv300-5bg432 Virtex FPGA device with the following properties and layout (Figure 2.8): [XDS003-2] [SYNP99] [XCNSTR] [Brown96]

- 32x48 CLB Array provide functional elements for constructing logic connected by global routing matrix or switch matrix (Figure 2.4),
- VersaRing™ forms the interface between Input Output Blocks (IOBs) and CLBs,
- 16 Block Rams (BRAMs) each 4096x1 totally 65536x1 bits,
- 4 Delay-Locked Loops (DLLs) that eliminate the skew between the clock input pad and internal clock input pins throughout the device,
- Ball grid 432 package having 316 I/O pins reserved for users with speed grade -5 which yields system performance up to 200 MHz.

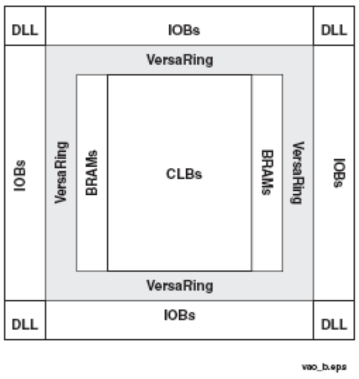


Figure 2.8: Virtex Architecture Overview ©Xilinx

2.3.1. Function Generation Capabilities of CLB

Each CLB comprises 4 function generator (LUTs) distributed into two slices. Each slice contains 2 function generators and additional logic that combines the outputs of LUTs and generates 5 (MUXF5) and 6 (MUXF6) input functions (Figure 2.9). Each slice can generate any functions of 5 inputs up to some functions of 9 inputs; hence any CLB can generate any functions of 6 inputs up to some functions of 19 inputs.

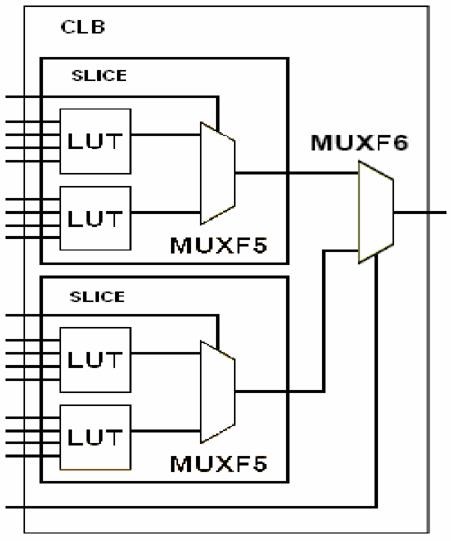


Figure 2.9: Function Generator Configuration of CLB

2.3.2. Distributed (Shallow) Memory Usage of CLB

Each LUT in a Slice can be configured as 16x1 bit synchronous RAM and two LUTs in a Slice can be configured as 16x2 bit or 16x1 bit dual port or 32x1 bit synchronous RAM.

2.3.3. Shift Register Configuration of CLB

Each LUT in a slice can be configured as dynamically addressable16 bit shift register.

2.3.4. Arithmetic Capabilities of CLB

Each LUT in a slice has a dedicated XORCY gate for single bit sum to form a full adder and dedicated carry path (Figure 2.10) which is using also dedicated routing resources along vertically adjacent CLBs [XAPP215]. By introducing the additional XORCY gate, 2 inputs of LUT left as spare and these inputs can be used to implement additional logic thereby increasing cell functionality. [TW04] [KCHAP93] [DFMULT]

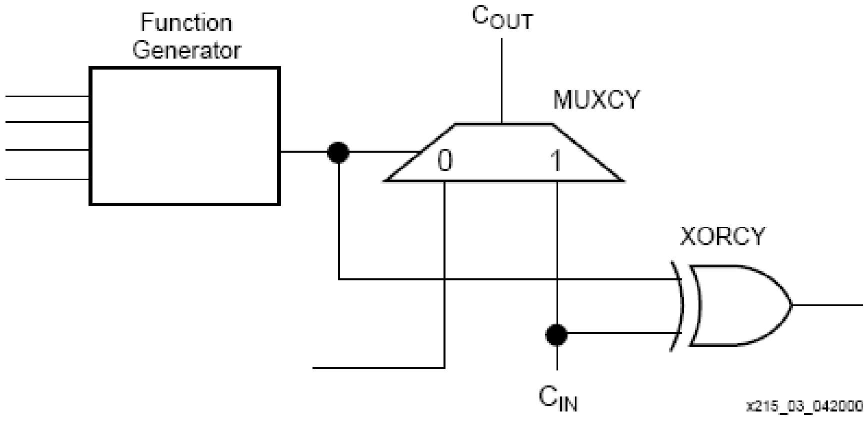


Figure 2.10: Carry Logic Diagram ©Xilinx

Multiplication in FPGA is performed by shifting and adding the partial products in parallel fashion. There exists 2 input AND gate per LUT to implement 1 bit multiplier [XAPP215] and this pattern repeats throughout the multiplier. In case of operands (partial products) are not equal to each other C_{IN} signal is propagated (Figure 2.11). Additional AND gate is essential to kill or generate C_{OUT} signal produced when the propagation of C_{IN} signal is stopped (when both operands equal) [HPCC].

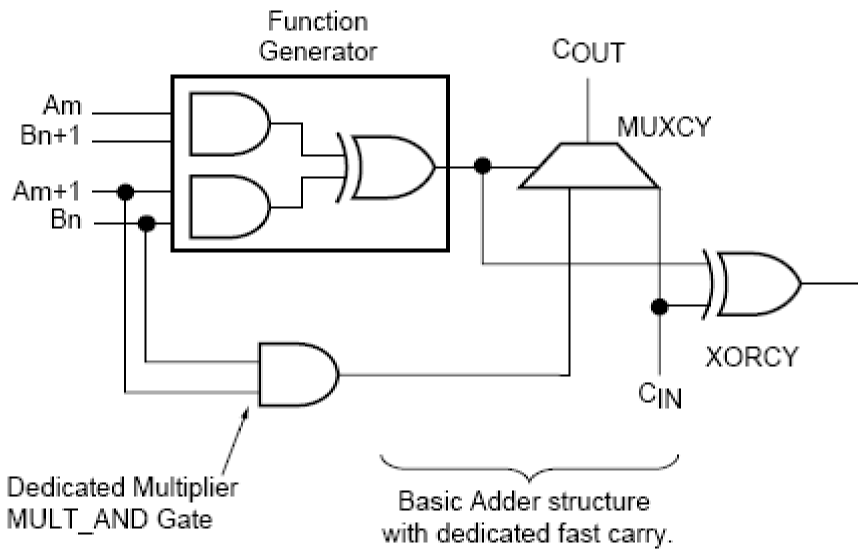

Function Generator
COUT
MUXCY
Am
Bn+1
Am+1
Bn
XORCY
CIN
Dedicated Multiplier MULT_AND Gate
Basic Adder structure with dedicated fast carry.

Figure 2.11: Multiplier Implementation ©Xilinx

2.4. PCI Host Software: In-Circuit Debugging of the Architecture

The "MIPS Monitor" (Figure 2.12) software which is running on PC was developed to debug the architecture after generated configuration was programmed into the target PROM or a new program is ready to be programmed while Virtex FPGA was running [PLXSDK01].
"MIPS Monitor" uses PCI Application Interface (API) provided by PLX Technology™ to read the FPGA's internal data and program memory, pipeline stage's inputs/outputs, pipeline register states and current PC.

It also enables the user to observe stalls and exceptions. It reflects information read by using PCI API to its graphical user interface, hence to user.

“MIPS Monitor” uses PCI API provided by PLX Technology™ to write the control signals to Virtex FPGA which resets the architecture or increment the PC by one thereby enabling single step operation.

“MIPS Monitor” graphical user interface enables the user by providing the following functionalities:

- Selecting the proper PCI 9030 device which is on the same board FPGA placed,
- Viewing the program which was already assembled and programmed to PROM,
- Viewing, loading and verifying a new program to local block instruction memory of FPGA.
- Inserting break points and running the architecture in single step or in free mode by using the graphical user interface of “MIPS Monitor”.

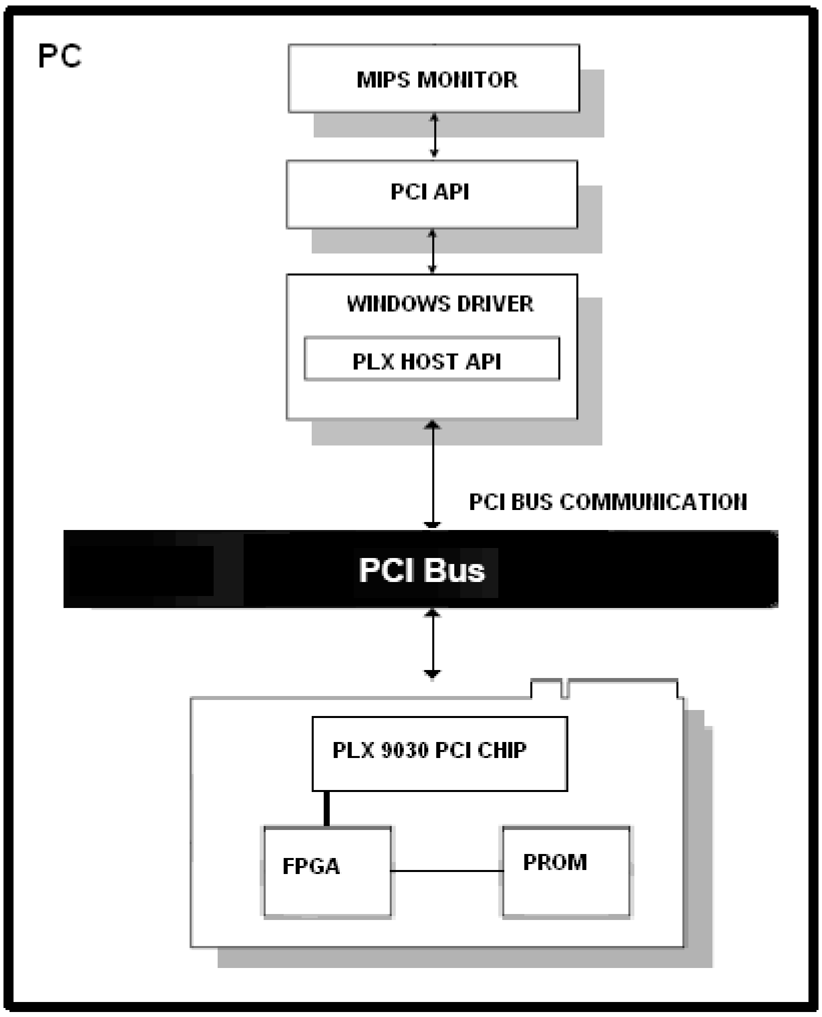


Figure 2.12: MIPS Monitor Software

The layout of the board used during this thesis is given in APPENDIX D, Layout of Board.

CHAPTER 3

RELATED RESEARCH

3.1. MIPS R2000 Instruction Set Architecture (ISA)

MIPS R2000 was first produced in 1988 by MIPS Computer Systems and was one of the RISC processors designed at that time. MIPS stands for Microprocessor without Interlocked Pipeline Stages and as its name implies, by eliminating pipeline interlocks between stages, instruction conflicts are resolved. Next generations are: R2010, also includes floating point co-processor, R3000 with cache control and lastly R4000 a 64 bit version of architecture. MIPS 32- and 64-bit architectures are used in networking and consumer device markets, such as in car navigation systems, digital television and cameras, video game controllers, switches and routers.
Primary metric to compare performance of Architectures is execution time of a program and it is presented in the following equation [COD98]:

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instruction Count}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

The multiplication factors on the right hand side of the equation do not determine performance individually, but have an affect. Selected ISA affects the instruction count. ISA Implementation scheme which will be described in section 3.3 affects clock cycles per instruction (CPI). The organization and technology of the architecture affects the clock rate.

These factors also depend on each other in inversely proportional relationship, making one better makes the other worse. For example making instructions complex reduces the instruction count but may decrease the clock rate. Good performance can be obtained by, first choosing ISA then determining the implementation scheme and last determining the technology.
MIPS (Microprocessor without interlocked Pipeline Stages) R2000 ISA has RISC based architecture obeying four design principles [COD98] [JGRAY00];

- *Smaller is faster*, MIPS have 32 general purpose register each 32 bits length. MIPS instructions operate only on registers. Registers are smaller hence faster than external memory.
- *Simplicity favors regularity,* MIPS's instructions have the same size each 32 bits length and the same number of operands, hence decoding and pipelining are simpler compared to variable length instructions present in CISC ISA.
- *Good design demand good compromises,* MIPS sticks to small number of instruction types and addressing modes.
- *Make common case fast (corollary of Amdahl's law),* implementing commonly used instructions in fast way makes the whole architecture faster.

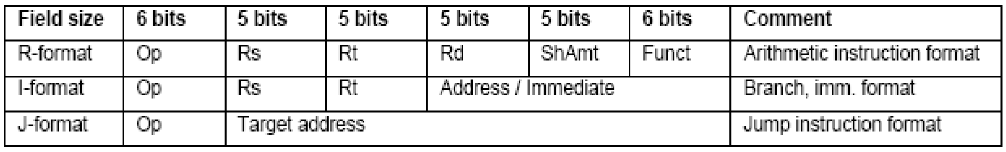
3.2. MIPS Instructions and MIPS Assembly Language

MIPS instructions can be grouped as Arithmetic, Transfer, Branch, Immediate and Jump instructions.
Arithmetic instructions operates on registers and requires three operands, two for source one for destination. The arithmetic or logical

operation takes place on two source operands and result is written back into destination register.
Transfer instructions are used for loading data from memory to registers or storing data from registers to memory. Transfer instructions require two operands. One register content is used as base address and the immediate field in the instruction as the offset from base, the other register is used either destination address of the value to be loaded or the source address of the value to be stored.
Branch instructions operate on two register operands, evaluate the condition and according the result continue execution or take the branch by modifying the PC.
Immediate instructions use the immediate field as an operand.
Jump instructions are use the immediate field to jump unconditionally by modifying the PC.
The detailed descriptions, functionalities and assembly language formats of MIPS R2000 instructions implemented and verified in this thesis are presented in APPENDIX A, Implemented Subset of MIPS R2000 ISA.

3.2.1. MIPS Instruction Format

General instruction format is given in Figure 3.1.



Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Comment
R-format	Op	Rs	Rt	Rd	ShAmt	Funct	Arithmetic instruction format
I-format	Op	Rs	Rt	Address / Immediate			Branch, imm. format
J-format	Op	Target address					Jump instruction format

Figure 3.1: MIPS Instruction Format

The Op field is the opcode of the instruction and used as the primary key in instruction decoding. Rs, Rt and Rd fields specify the address of register in operation. ShAmt field specify the shift amount in operation. Funct field selects the specific variant of the operation in opcode field.

3.2.2. MIPS Addressing Modes

Immediate addressing (Figure 3.2) means the operand is constant within the instruction itself;

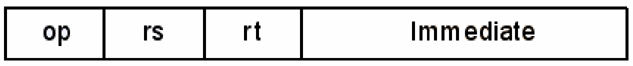


Figure 3.2: Immediate Addressing Mode

Register addressing (Figure 3.3) means where all operands are registers;

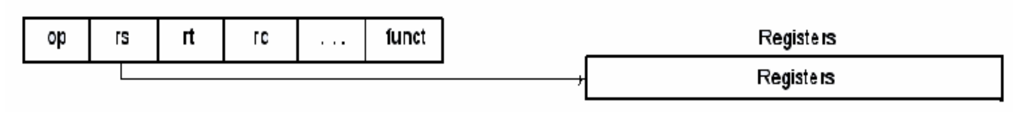


Figure 3.3: Register Addressing Mode

Base addressing (Figure 3.4) means where the operand is in memory whose address is calculated by adding base address in a register with an offset in immediate field. Addressing of memory is implemented as word (4 bytes) aligned.

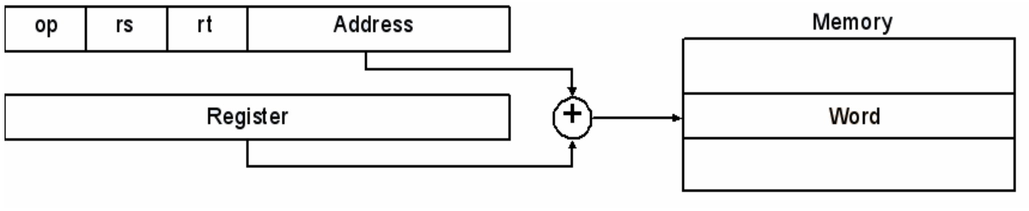


Figure 3.4: Base Addressing Mode

PC relative addressing (Figure 3.5) means that the instruction memory will be addressed by adding the present PC and the constant in the instruction.

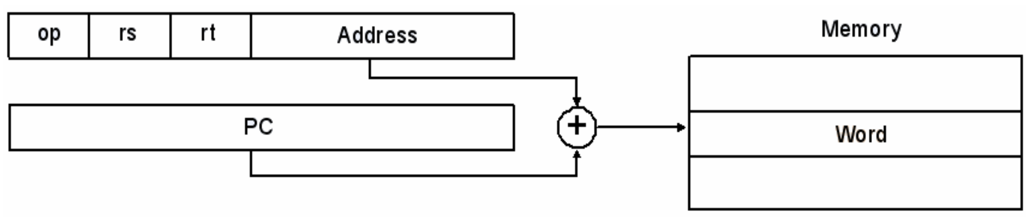


Figure 3.5: PC Relative Addressing Mode

Pseudo direct addressing (Figure 3.6) means the Address field in the instruction is concatenated with the program counter and the instruction memory than addressed.

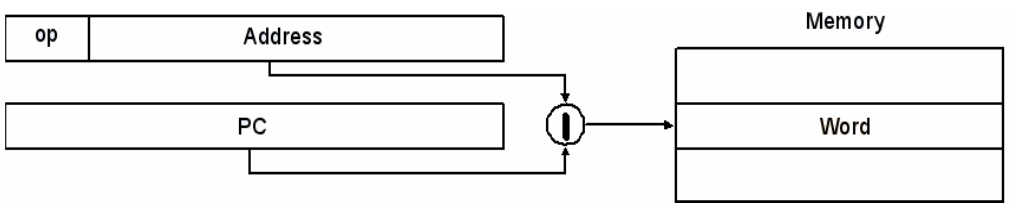


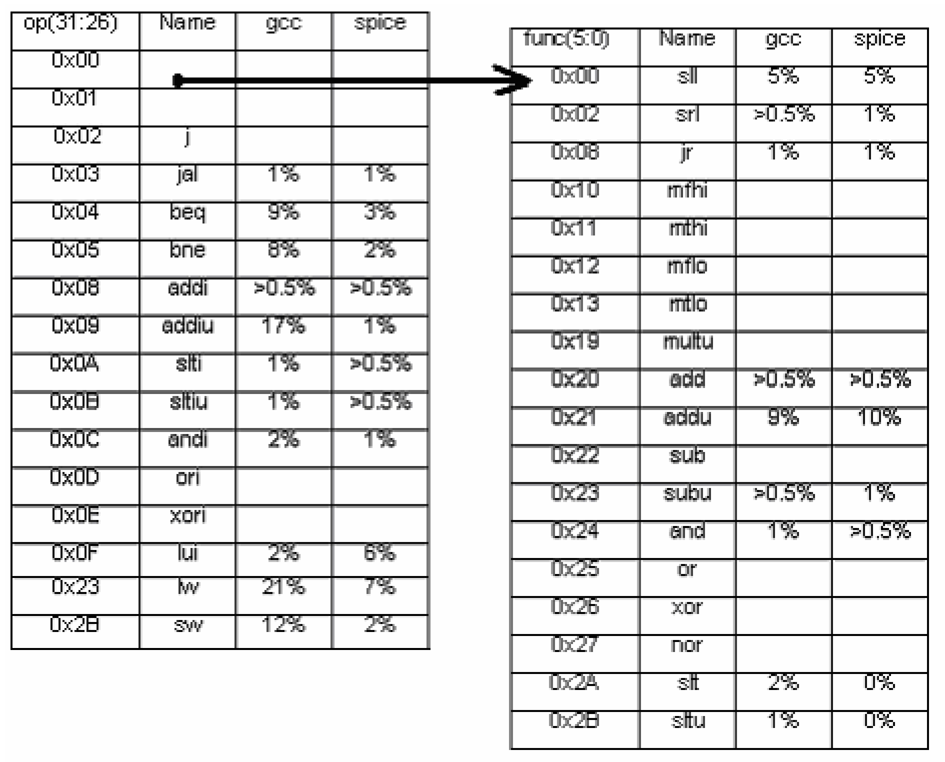
Figure 3.6: Pseudo Direct Addressing Mode

3.2.3. MIPS Instruction Decoding

MIPS R2000 instructions implemented and verified in this thesis were chosen according their frequency of usage in two totally different programs spice and gnu C compiler (gcc). These values were calculated from pixie which is an instruction measurement tool [COD98].

MIPS core instructions (all presented in Figure 3.7) cover 95% for gcc and 45% for spice. MIPS core instructions dominate gcc and integer plus floating point core instructions dominate spice. Instructions that did not cover in this thesis constitute the remaining part 5% for gcc and 55% for spice. 49% of spice can be covered by simply adding a floating point arithmetic core to architecture, which results in 5% for gcc and 6% for spice as uncovered.

Instructions are decoded and control signals are generated based on Figure 3.7. Related procedures will be described in detail in 0.



op(31:26)	Name	gcc	spice
0x00			
0x01			
0x02	j		
0x03	jal	1%	1%
0x04	beq	9%	3%
0x05	bne	8%	2%
0x08	addi	>0.5%	>0.5%
0x09	addiu	17%	1%
0x0A	slti	1%	>0.5%
0x0B	sltiu	1%	>0.5%
0x0C	andi	2%	1%
0x0D	ori		
0x0E	xori		
0x0F	lui	2%	6%
0x23	lw	21%	7%
0x2B	sw	12%	2%

func(5:0)	Name	gcc	spice
0x00	sll	5%	5%
0x02	srl	>0.5%	1%
0x08	jr	1%	1%
0x10	mfhi		
0x11	mthi		
0x12	mflo		
0x13	mtlo		
0x19	multu		
0x20	add	>0.5%	>0.5%
0x21	addu	9%	10%
0x22	sub		
0x23	subu	>0.5%	1%
0x24	and	1%	>0.5%
0x25	or		
0x26	xor		
0x27	nor		
0x2A	slt	2%	0%
0x2B	sltu	1%	0%

Figure 3.7: MIPS Opcode Map and Frequency of Instructions

3.3. Survey of Instruction Set Architectures Implementation Schemes

The path which is followed by instructions and data and controlled by signals generated by control unit called data path. Each type of instruction follows different path trough architecture because the operands on which instruction operates differ.

Data path is formed by state and combinational logic elements. These elements are combined in different organizations and different implantation schemes emerge.

Building architecture requires some sequential decompose and re-unite iterations. It is necessary to decompose in order to understand, and it is necessary to re-unit in order to build. There exists a contradiction, because it is necessary to decompose in order to reunite. This contradiction was used as a methodology and followed throughout the survey of implementation schemes. Big picture is given first. Then it is decomposed and fully understood.

3.3.1. Single Cycle Implementation Scheme

In this scheme (Figure 3.8) single instruction starts on clock edge and ends on the next clock edge. The clock rate is determined by the slowest instruction; in spite there exists faster instructions in ISA. Hence this scheme is impractical to implement but useful to understand. Each instruction irrespective of its instruction format is fetched from memory; the next PC is calculated by adding 4 byte offset to present PC and decoded according to its bit field based on Figure 3.1. The operation on registers is determined by the ALUOp control signal which depends on the Funct field of the instruction and determined in decode stage.

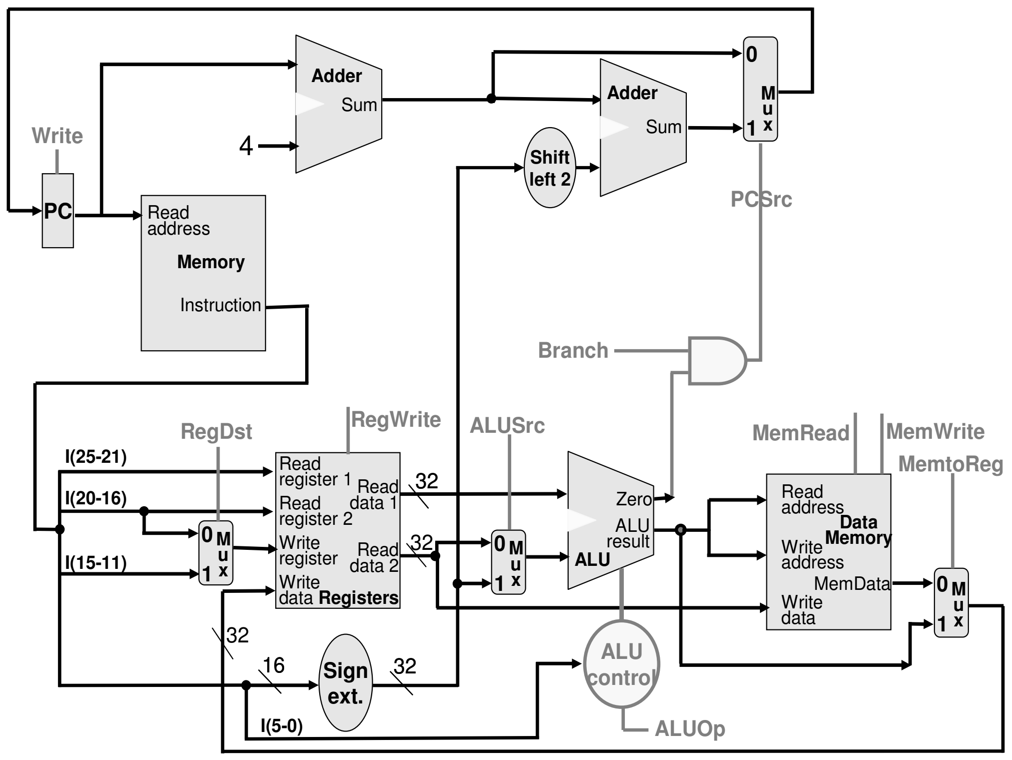


Figure 3.8: Single Cycle Implementation Scheme ©[COD98]

Multiplexers can be used to divide the architecture into smaller pieces. The presence of a multiplexer before an input element means that that element is used by as many different instruction types as the number of inputs of the multiplexer. The select signal, namely the instruction type determines the path of the data throughout the architecture for the present clock cycle. For instance, the multiplexer with control signal ALUSrc determines either ALU is used for address calculation for data memory load/store or arithmetic operation on register operands. In either case ALU can be used only by one instruction type in the same clock, hence some hardware duplications exist in the architecture for other calculations such as the adder for next program counter, despite the ALU can be used for this purpose. This is another fact which proves that this implementation scheme is impractical to implement and its problems will be solved in multi cycle implementation scheme which will be described in section 3.3.2.

Similarly, the multiplexer with control signal MemtoReg determines which data will be written to the register bank either the result calculated by ALU or the data loaded from data memory.

The multiplexer with control signal RegDst differentiate R-type and I-type instructions because the destination register address field is different for these types. For R-type instructions, the destination address is specified in Rd field whereas in I-type instructions the destination address is specified in Rt field (Figure 3.1).

The multiplexer with control signal PCSrc determines the next PC. The next PC is PC+4 bytes for all instruction types except from conditional branch. For branch instructions (Branch control signal is asserted) if the condition is satisfied (e.g. for “branch on equal” instruction, when the operands are the same, their difference will be zero. Hence the ALU’s zero output set to ‘1’) the next PC is calculated according to Figure 3.5.

3.3.2. Multi Cycle Implementation Scheme

In this scheme (Figure 3.9) instructions are executed in multi clock cycles. Register Blocks are added between functional units to hold the temporal values for using on a later clock cycle. Clock rate is determined by the slowest functional unit and functional units can be used more than once per instruction (e.g. single ALU is used instead of an ALU and two adders Figure 3.8) as long as access to this unit occurs on different clock cycles. Single memory unit is used instead of separate instruction and data memories and multiplexer with control signal IorD determines data or instruction access.

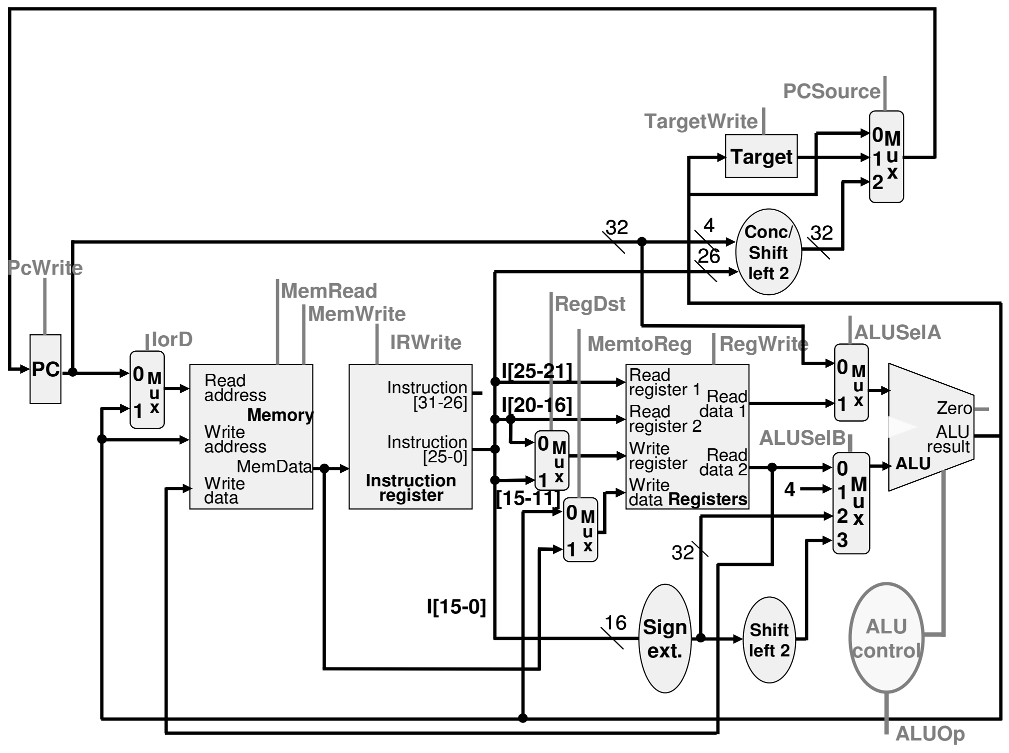


Figure 3.9: Multi Cycle Implementation Scheme ©[COD98]

Jump instruction is also shown in the scheme. The multiplexer with control signal PCSource selects next program counter calculated based on Figure 3.6 when unconditional jump instruction was fetched from memory. A more complex control logic compared to single cycle implementation scheme is needed and the state flow diagram of control unit is given in Figure 3.10.

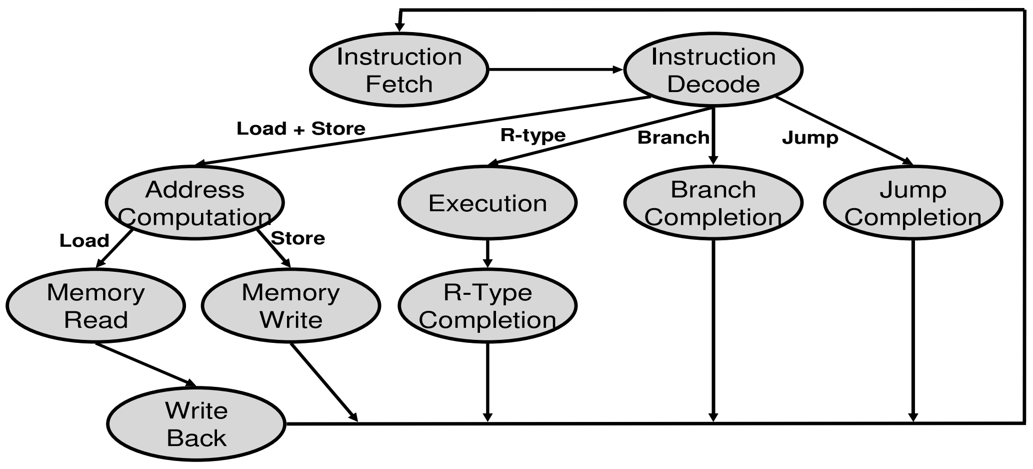


Figure 3.10: State Flow Diagram of Multi Cycle Scheme Control Unit

3.3.3. Pipelined Implementation Scheme

In this scheme (Figure 3.11), there exists single clock cycle between subsequent instructions like single cycle implementation scheme.

Clock rate is as high as multi cycle implementation scheme and is determined by the slowest functional unit similar to multi cycle implementation scheme. There exist register blocks between functional units, which are responsible for storing the information for the next clock cycle.

The difference between multi cycle scheme and pipelined scheme is that the instruction does not wait for the previous instruction until the end of write back stage and directly fetched from instruction memory while the previous instruction is being decoded.

The same control signals which are valid for single and multi cycle schemes are also valid for pipelined scheme, but in contrast to multi cycle implementation scheme, special control unit implementation (flow diagram was given in Figure 3.10) is not necessary for generation of these control signals. Sequencing is inherently present in this scheme and control signals generated in decode stage go with the instruction throughout the pipeline and are wasted up until the last stage.

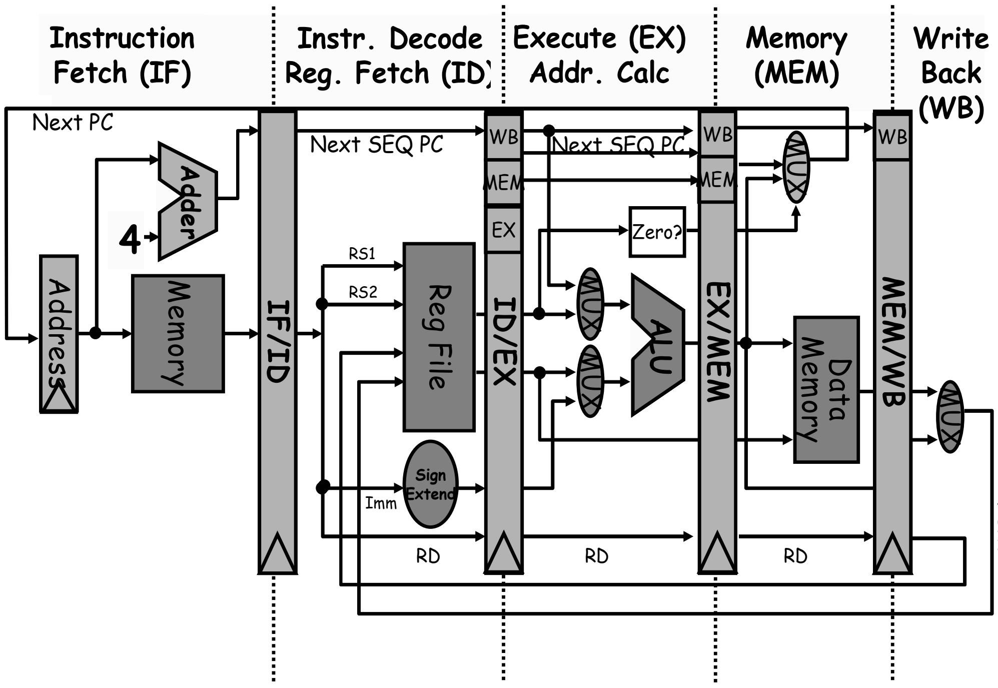


Figure 3.11: Pipelined Implementation Scheme ©[COD98]

Pipelining does not improve or speed up the functional units in the architecture, instead increases the throughput by decreasing the time between instructions. There exist as much instructions as the number of stages in the pipeline simultaneously, e.g. while the fifth instruction is being fetched (IF) from memory, in the same time, first instruction is in write back (WB) stage following five clock cycles its IF stage (Figure 3.12).

	CLK1	CLK2	CLK3	CLK4	CLK5
Instruction 1	IF	ID	EX	MEM	WB
Instruction 2		IF	ID	EX	MEM
Instruction 3			IF	ID	EX
Instruction 4				IF	ID
Instruction 5					IF

Figure 3.12: Simultaneously Executing Instructions in Pipeline

3.3.4. Quantitative Comparison of Implementation Schemes

Primary metric to compare performance of Architectures is execution time of a program as stated in section 3.1. Pipelined implementation scheme has the best features of other implementation schemes, low clock cycle per instruction like single cycle scheme which is optimally equal to 1 disregarding pipeline hazards described in section 3.4 and high clock rate like multi cycle implementation scheme; therefore it is expected to give the best performance. It will be a good practice to demonstrate the relative performances by giving a realistic example. MIPS instructions has the frequency of usage as stated in Figure 3.7 in gcc program and number of clock cycles as stated in Figure 3.10 which also summarized in Table 1.

CPI can be calculated by using this table adding the weighted sums of instructions in gcc program.

$$\begin{aligned} \text{CPI} &= 5 \times 0.23 + 4 \times 0.13 + 3 \times 0.19 + 3 \times 0.02 + 4 \times 0.43 \\ &= 4.02 \end{aligned}$$

Table 3.1: Calculation of CPI for Multi Cycle Implementation Scheme

Instruction Type	Frequency	Number of Clock Cycles
LOAD	23%	5
STORE	13%	4
BRANCH	19%	3
JUMP	2%	3
ALU	43%	4

The clock rate or clock cycle period is determined by the slowest stage in the pipeline. For second per instruction calculation, clock period shall be multiplied with CPI (equation given in section 3.1). Optimal speedup is obtained from pipelining by using balanced stages in pipeline. Say that each stage is balanced and takes T sec/clock cycle.

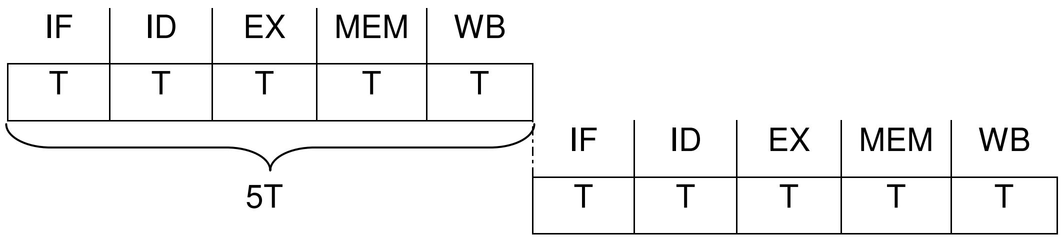


Figure 3.13: Single and Multi Cycle Instruction Sequence

For single cycle implementation scheme, single cycle clock period takes 5T seconds. For multi cycle implementation scheme, single cycle period takes T seconds similar to pipelined implementation scheme. Hence, the instruction times given in Table 3.2 were obtained. According to this table, it can be seen that, pipelined implementation is nearly 5 times faster than the other implementation schemes.

Table 3.2: Instruction Time Calculation for Implementation Schemes

Implementation Scheme	Seconds/Instruction (CPI x sec/clock)
Single Cycle	1 x 5T = 5T
Multi Cycle	4.02 x T = 4.02T
Pipelined	1 x T = T

3.4. Problems and Solutions in Pipelined Architectures

As stated in section 3.3.4, optimal performance and speedup can be obtained from pipelining by balancing the stages and full speed usage of the pipeline without stalls. In reality this can be not possible always. Even perfect balance between pipeline stages can not be adequate alone.

There may be existent restrictions;

- Dependencies between instructions,
- Some hardware restrictions to support pipelining,
- Branches can not be determined until Execute (EX) stage and following instructions can be fetched uselessly.

Detailed explanation how these cases are handled given in the following sections.

3.4.1. Structural Hazards

Structural hazards emerged because the underlying hardware does not support special instruction combinations which are simultaneously present in the pipeline. For example, the instructions 1 and 4 presented in Figure 3.12 access the memory in the same clock cycle, CLK4. If the instruction memory and the data memory are not separated physically, this architecture can not support this special combination. In clock cycle CLK5, both Instruction Decode (ID) and Write Back (WB) stages access the register bank, but in this case the hardware clash is avoided by using forwarding mechanism which will be described in the section 3.4.3.

3.4.2. Brach Hazards

Branch hazards emerged because three instructions, following the branch instruction, are already in the pipeline in any case until branch condition is evaluated or unconditional jump address determined (according to Figure 3.11). In case of branches are taken, these fetched instructions must be discarded and the goal of using pipeline in its full speed one instruction per clock cycle can not be achieved. Three clock cycles are wasted effectively in case of taken branch; assuming branch is not taken always.

In this thesis, the decision making and address calculation mechanism moved to ID stage to reduce the wasted time to one clock cycle. The assumption which is called delayed branch mechanism, “braches are always not taken” is followed. In this case, the following instruction is always fetched. In case of taken branch, one slot is left as discarded and useless. If the decision is left to compiler as in case in high level programming, compilers usually fill this slot with useful instructions which are independent from the branch condition. If useful instruction can not be found, this slot is filled with well known No Operation (NOP) instruction which does not change the internal state of microprocessor. A NOP instruction is added manually after every branch in this thesis, because programming is done in assembly and compiler support is not present. There exists no special implementation in this thesis which detects this hazard and flushes the fetched instruction.

One delay slot can be easily filled with NOP or with useful instruction, but as the pipeline gets bigger, filling slots with useful instructions gets also harder. There exists other mechanism proposed in the literature to solve this problem. Dynamic prediction mechanism with additional hardware is one of them, which depends on the past statistics collected for that branch point. The decision is made based on this statistics which is changing in time with conditions.

3.4.3. Data Hazards

Data hazards emerged because an instruction which depends on the previous instruction is in the pipeline and previous instruction did not finish its work, for example does not write back the calculated result to destination register. In this type of hazard, the solution is not left to compilers entirely like the branch hazard described in 3.4.2 and tried be solved with hardware if possible. The hazard will appear when the destination register of the previous instruction in either EX, MEM or WB stage is the same as the one of the source registers of the current instruction which is in the ID stage. In Figure 26, data hazard is resolved by forwarding data from EX, MEM and WB stages of the first instruction to ID stages of following instructions which has a without waiting to complete first instruction to WB its destination register R1.

	CLK1	CLK2	CLK3	CLK4	CLK5
add **R1**,R2,R3	IF	ID	EX ●	MEM ●	WB ●
sub R4,**R1**,R2		IF	↓ ID	EX	MEM
xor R6,R7,**R1**			IF	↓ ID	EX
add R8,**R1**,**R1**				IF	↓ ID
sw R9, 100(R1)					IF

Figure 3.14: Data Hazard Solution by Forwarding

The data hazard must be resolved in ID stage before register bank access and branch decision. A NOP instruction is inserted into the instruction sequence, if hazard can not be solved and time is gained for resolution by using forwarding in the next clock cycles. In Figure 3.15, hazard can not be solved by just using forwarding, because the result for destination register R2 will be not available until memory access. Therefore, pipeline is stalled for one clock cycle and data hazard is resolved in the next clock cycle by forwarding data from Data Memory (MEM) stage of previous instruction to ID stage of the current instruction.

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6
lw **R2**,100(R1)	IF	ID	EX	MEM ●	WB	
and R4,**R2**,R5		STALL	IF	↓ ID	EX	MEM
or R8,**R2**,R6				IF	ID	EX

Figure 3.15: Data Hazard Solution by Stalling and Forwarding

Some extra precautions must be taken into account while using forwarding mechanism. In Figure 3.16, the result obtained in clock cycle CLK4 from the addition of second instruction is forwarded from EX stage instead of the result obtained in clock cycle CLK3 from MEM stage, because it is more recent.

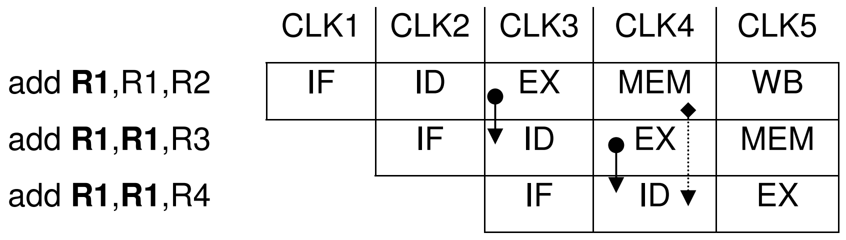


Figure 3.16: Forwarding of the Most Recent Data

3.4.4. Exception Hazard

Hardware shall prevent completion of instructions which are following the instruction which cause exception and let all prior instructions to complete. Internal register blocks shall be flushed to prevent them to effect Register Bank and Data Memory. Program Counter shall be equated to special address like Branch or Jump instruction case. This address is generally called as interrupt or exception vector.

CHAPTER 4

IMPLEMENTATION OF MIPS PIPELINED ARCHITECTURE

This chapter describes the internal structure of the processor and the auxiliary structures to monitor and manipulate the internal registers of the processor. Internal structures of the processor are constituted by combining the following primary units and their subunits. (Figure 4.1: Internal Structure of the Pipelined Processor)

- Instruction Fetch Unit (IF_Unit)
    - Instruction Memory (256x32bit block memory)
- Instruction Decode Unit (ID_Unit)
    - Register Bank (dual port 32x32bit block memory)
- Forwarding and Hazard detection Unit (FORWD_HZRD Unit)
- Control Unit (CONTROL_Unit)
- Execute Unit (EXECUTE_Unit)
- Data Memory Unit (256x32bit block memory)
- Exception Detection Unit (EXCEPTION_DTCT_UNIT)
- Four register blocks responsible for storing information between clock cycles and located between Units;
    - Instruction Fetch - Instruction Decode (IF_ID Unit)
    - Instruction Decode - Execute (ID_EX Unit)
    - Execute - Data Memory (EX_MEM Unit)
    - Data Memory – Instruction Decode (MEM_WB Unit)

Auxiliary structures of the processor are constituted by combining the following units. Units and their interconnections are presented in Figure 4.2.

- Clock Delay Locked Loop to eliminate the skew between clock input pad and the internal clock input pins (CLKDLL Unit)
- Interface between the processor and the PCI Bridge (pci_9030 Unit)
- External reset of the processor (reg_wr Unit)
- External programming of the Instruction Memory (reg_prg Unit)
- External single step execution of processor (wait_sm Unit)
- External reading of internal state of register blocks (reg Units)
- Processor itself (top_level Unit)

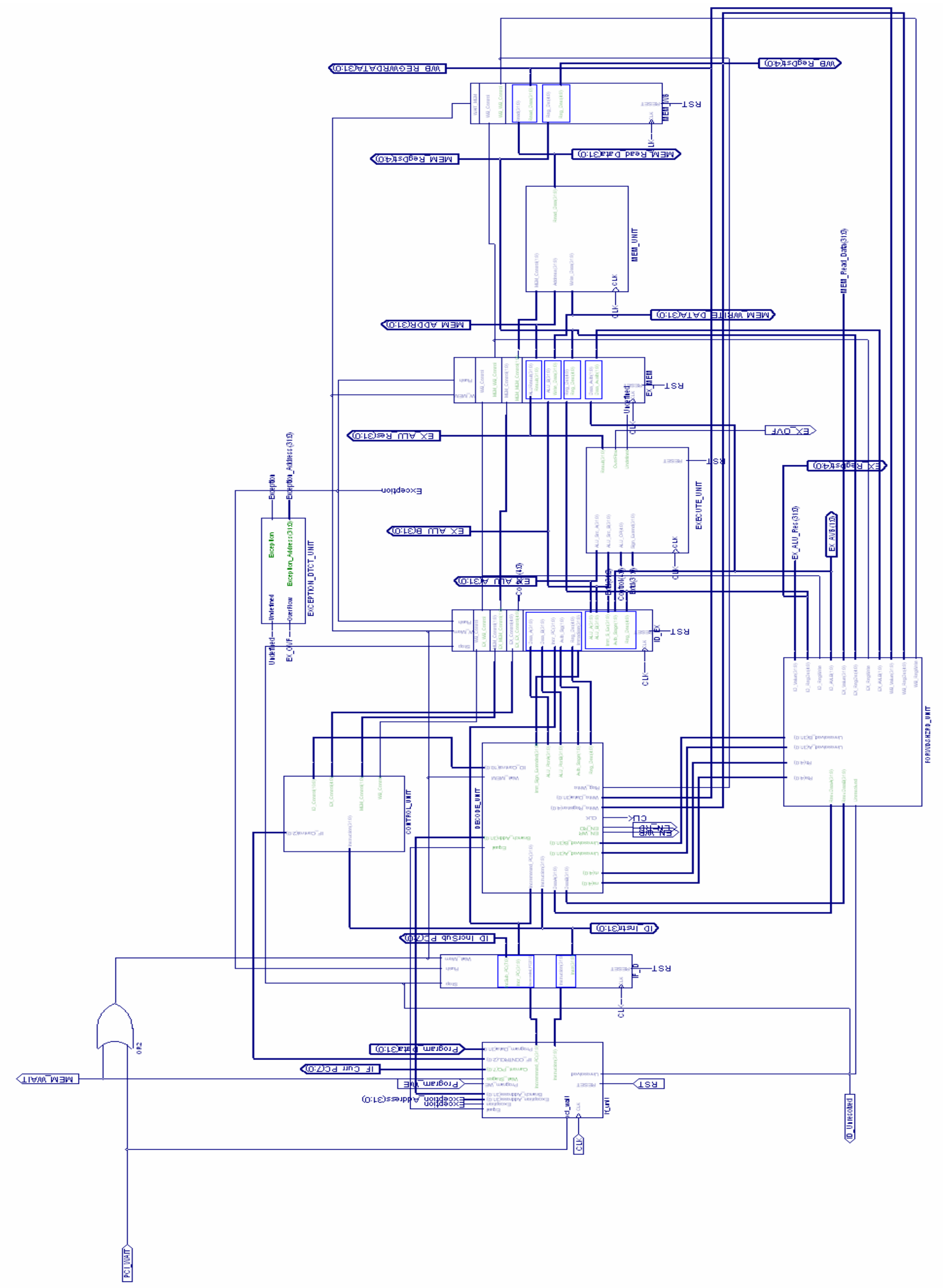


Figure 4.1: Internal Structure of the Pipelined Processor

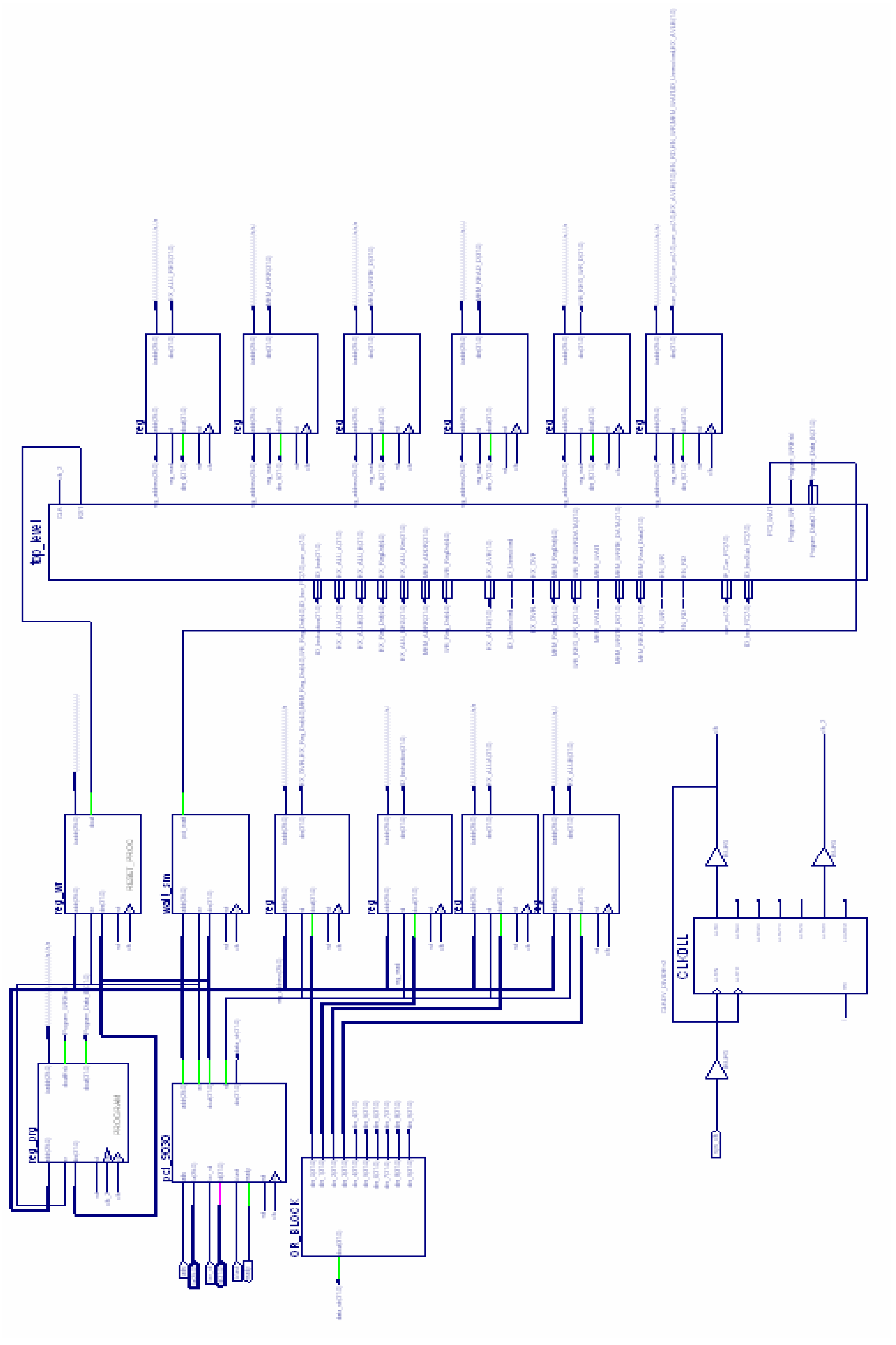


Figure 4.2: External Structure of the Pipelined Processor

4.1. Internal Structure of the Processor

In this section the primary building blocks are described in detail by stating their functions and input/output signals (in figures, inputs are placed on the left and outputs are placed on the right). General signals which are common for majority of building blocks are described here. Remaining signals are described in related building block sections. Every signal is described once that means the same input signal of various blocks is also an output signal of single block; therefore there will be a cross reference (links can be followed by CTRL + Click in this document) input signal definition section of each block to output signal definition section of source block of the signal in which the same signal is described in detail to avoid redefinition. During definition of signal levels, ”set” means logic level 1 and reset means logic level 0.
<u>CLK (1 bit) and RESET (1 bit):</u> Internal clock (20 MHz) and internal reset signals. These signals are active high signals.
<u>Register Dest (5 bit):</u> This signal is transferred across all pipelines for instructions which will write to Register Bank in WB stage.

4.1.1. Instruction Fetch Unit

The design of the Instruction Fetch Unit is realized by using HDL Design entry method. Instruction Fetch Unit includes the subunit Instruction Memory (256x32bit block memory) from which instructions are fetched in every clock cycle except when an unresolved (load/store) hazard exists in the pipeline which ends up with pipeline stall. The hardware flow diagram of this building block is given in APPENDIX C, Figure C.1: Instruction Fetch Unit Flow Diagram.

4.1.1.1. Input/Output Signals of Instruction Fetch Unit

The connections of Instruction Fetch Unit with other units can be seen in Figure 4.1: Internal Structure of the Pipelined Processor. All Input/Output signals can be seen in Figure 4.3: Input/Output Signals of Instruction Fetch Unit.

Output signals are as the following;

Current PC (8 bit): Signal goes to auxiliary structures to monitor the present state of the Program Counter.

Incremented PC (32 bit): Signal goes to Instruction Decode Unit and forwarded until WB stage for jal instruction, because this instruction writes the return address into Register Bank address 31 for later usage in return from subroutine (by using jr instruction). This signal is also used in instruction decode stage to calculate the branch and jump address.

Instruction (32 bit): Signal which is fetched from instruction memory goes to Instruction Decode and Control Units. Instruction is parsed into fields according to Figure 3.1 in Instruction Decode unit and control signals are generated in Control Unit. These signals are passed to internal register blocks for further evaluation of the parsed fields in the following clock cycles after decode stage.

Wait Stages (1 bit): Signal is OR'ed with pci_wait signal and goes to all internal registers between building blocks. If this signal is set that means, memory access (instruction memory, data memory and Register Bank access requires one clock cycle) is taking place and all processor stages are stopped during this signal is set which corresponds to one clock cycle period. Program Counter is also not updated during this signal is set.

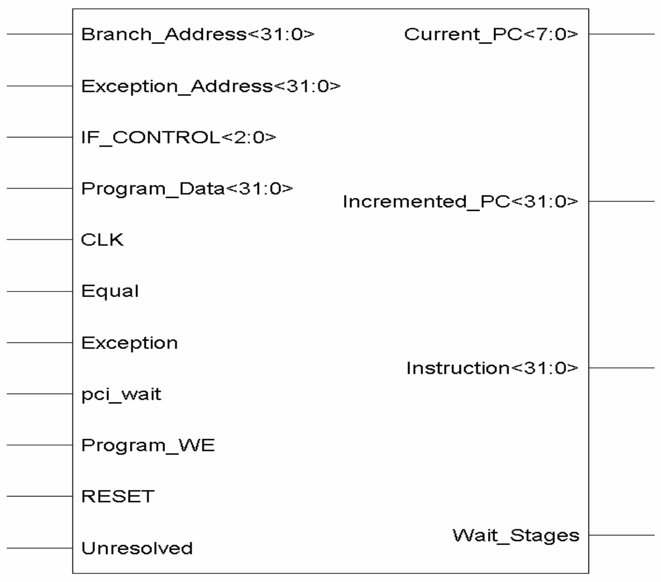


Figure 4.3: Input/Output Signals of Instruction Fetch Unit

Input Signals are as the following;

<u>Exception (1 bit):</u> Exception Detection Unit output signal.

<u>Exception_Address (32 bit):</u> Exception Detection Unit output signal.

<u>Branch_Addr (32 bit):</u> Instruction Decode Unit output signal.

<u>Equal (1 bit):</u> Instruction Decode Unit output signal.

<u>IF_Control (3 bit):</u> Control Unit output signal.

<u>Program_Data (31 bit) and Program_WE (1 bit):</u> Signals are fed from external sources and used when in external programming mode. These signals are useless in normal operating mode of the processor.

<u>Pci_wait (1 bit):</u> Signal comes from external source and used as single step execution trigger. Program Counter is updated during the clock rising edges if and only if this signal is not set.

<u>Unresolved (1 bit):</u> Forwarding and Hazard detection Unit output signal.

4.1.1.2. Function of Instruction Fetch Unit

The primary function of Instruction Fetch Unit is to fetch instructions from Instruction memory and send it to Control and Decode Units for processing. If Wait_Stages or Pci_wait or Unresolved signal is set, current program counter retains its value, hence the same instruction is fetched from memory on the next clock cycle. If a branch or jump instruction is in decode stage inspecting the IF_CONTROL signal, next program counter is determined according to evaluation of Equal and Branch_Address signals. During instruction memory access, Wait_Stages signal is set and processor is stopped for one clock cycle. On the next clock cycle, Wait_Stages signal will be in reset state and processor is allowed to run, hence during operation of processor Wait_Stages signal toggles. This halves the processor's effective clock speed from 20 MHz to 10 MHz. If RESET signal is set, Program Counter is set to byte address 16 after overflow exception vector. In case of an exception PC is set to proper exception vector. If Program_WE signal is set, Instruction memory enters in external programming mode and on every clock cycle Program_Data signal is written to Instruction Memory sequentially.

4.1.2. Instruction Decode Unit

The design of the Instruction Decode Unit is realized by using HDL Design entry method. Instruction Decode unit includes the subunit Register Bank (dual port 32x32bit block memory) from which operands on which operations take place are fetched and to which operation results or loaded data from data memory are stored in every clock cycle. The hardware flow diagram of this building block is given in APPENDIX C, Figure C.2: Instruction Decode Unit Flow Diagram.

4.1.2.1. Input/Output Signals of Instruction Decode Unit

The connections of Instruction Decode Unit with other units can be seen in Figure 4.1: Internal Structure of the Pipelined Processor. All Input/Output signals can be seen in Figure 4.4: Input/Output Signals of Instruction Decode Unit.

Output signals are as the following;
ALU PORTA (32 bit): Signal goes to ALU port A for evaluation according to instruction present in EX stage. This signal can come from the other stages by forwarding or represents shift amount for sll and srl instructions.
ALU PORTB (32 bit): Signal goes to ALU port B for evaluation according to instruction present in EX stage. This signal can come from the other stages by forwarding or represents Incremented Program Counter for jal instruction or zero or sign extended immediate field according to control signal. For memory store operation sw, this signal represents the data which will be stored to data memory and directly forwarded to MEM stage.
Avlb Stage (2 bit): Signal goes to Forwarding and Hazard detection Unit and is used to determine if unresolved data hazard which ends up with pipeline stall is present. If the result of the instruction in EX stage will be available in MEM stage (lw instruction's Avlb_Stage is equal to MEM) and the destination of the instruction is the same as the one of the source operands of the instruction present in ID stage then pipeline is stalled for one clock cycle and data hazard is resolved using forwarding mechanism.

Branch_Addr (32 bit): Signal goes to Instruction Fetch Unit and used to determine the value of next program counter if a conditional or unconditional branch instruction is present in instruction decode stage.

Imm_Sign_Extended (32 bit): Signal goes to Execute Unit and used to calculate the destination register address for sw instruction. The base address is carried to Execute Unit via Port A like lw instruction, but the offset can not be carried via Port B. Port B represents the data which will be stored in data memory for this instruction hence this signal was needed to be transferred.

Register_Dest (5 bit): General signal which represents the destination register which will be used in WB stage.

rs (5 bit), rt (5 bit), Unresolved_A (32 bit) and Unresolved_B (32 bit): Signals go to Forwarding and Hazard detection Unit. Rs and Rt represent the source addresses of operand registers and are compared with instruction's destination register address in either EX, MEM or WB stages. Forwarding Unit will determine the data hazard is present. If no hazard is detected, the Unresolved_A and Unresolved_B which represent the values in register Bank addresses Rs and Rt will be forwarded to ALU ports.

EN_RD (1 bit) and EN_WR (1 bit): Signals go to auxiliary structures to monitor the present state of the read and write enable pins of Register Bank They were used during development and currently not used.

Equal (1 bit): Signal goes to Instruction Fetch Unit and if set that means operands on which conditional branch instruction was applied are equal, if not set, inequality condition is true.

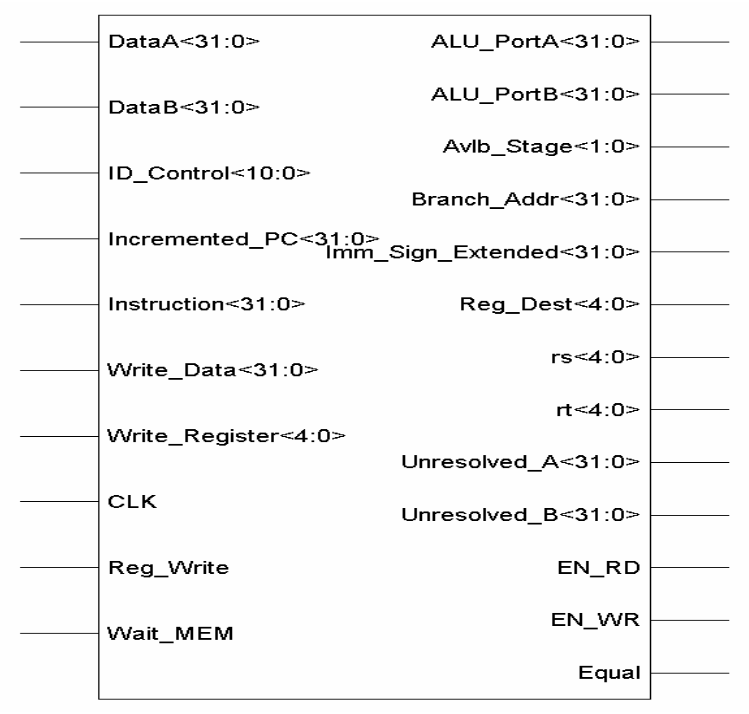


Figure 4.4: Input/Output Signals of Instruction Decode Unit

Input signals are as the following;

<u>DataA (32 bit) and DataB (32 bit):</u> Forwarding and Hazard detection Unit output signals. (ResvDataA and ResvDataB)

<u>ID_Control (11 bit):</u> Control Unit output signal.

<u>Incremented_PC (32 bit):</u> Instruction Fetch Unit output signal.

<u>Instruction (32 bit):</u> Instruction Fetch Unit output signal.

<u>Write_Data (32 bit), Write_Register (5 bit) and Reg_Write (1 bit):</u> These signals are WB stage signals and Write_Register determines the address of the Register Bank in which the Write_Data will be written if Reg_Write signal is set and Write_Register (destination address) is not equal to 0, because the register address 0 is named as $zero register and it is not allowed writing to this address.

<u>Wait_MEM (1 bit):</u> Signal is generated by OR'ing the output signal Wait_Stages of Instruction Fetch Unit and the external one step execute

trigger signal Pci_wait. If this signal is set, the EN_WR signal is set and if this signal is reset EN_RD signal is set, hence the Register Bank is written first and after that it is read.

4.1.2.2. Function of Instruction Decode Unit

The functions of Instruction Decode Unit are;

- Preparing the Register Bank addresses and register contents to determine final resolved values on which the instruction in ID stage will operate in following stages,
- Access the Register Bank for writing and reading,
- Make the evaluation of conditional branch and determine the final branch and jump address and fed it to Instruction Fetch Unit.

4.1.3. Forwarding and Hazard Detection Unit

The design of the Forwarding and Hazard Detection Unit is realized by using HDL Design entry method. The hardware flow diagram of this building block is given in APPENDIX C, Figure C.3: Forwarding and Hazard Detection Unit Flow Diagram.

4.1.3.1. Input/Output Signals of Forwarding and Hazard Detection Unit

The connections of Forwarding and Hazard Detection Unit with other units can be seen in Figure 4.1: Internal Structure of the Pipelined Processor. All Input/Output signals can be seen in Figure 4.5: Input/Output Signals of Forwarding and Hazard Detection Unit.
Output signals are as the following;

ResvDataA (32 bit) and ResvDataB (32 bit): Signals go to the DataA and DataB inputs of Instruction Decode Unit and then forwarded to ALU ports taking into account the control signals. The final values of these signals are determined by using the input signals and VHDL code is given below;

Table 4.1: Forwarding Mechanism for Register Bank Primary Port

```
ResvDataA <= ID_Value when ((ID_RegWrite = '1') and (ID_RegDst = Rs) and (ID_RegDst /= "00000"))
else EX_Value when ((EX_RegWrite = '1') and (EX_RegDst = Rs) and (EX_RegDst /= "00000"))
else WB_Value when ((WB_RegWrite = '1') and (WB_RegDst = Rs)   and (WB_RegDst /= "00000"))
else Unresolved_A;
```

Table 4.2: Forwarding Mechanism for Register Bank Secondary Port

```
ResvDataB <= ID_Value when ((ID_RegWrite = '1') and (ID_RegDst = Rt) and (ID_RegDst /= "00000"))
else EX_Value when ((EX_RegWrite = '1') and (EX_RegDst = Rt) and (EX_RegDst /= "00000"))
else WB_Value when ((WB_RegWrite = '1') and (WB_RegDst = Rt) and (WB_RegDst /= "00000"))
else  Unresolved_B;
```

Unresolved (1 bit): Signal goes to Instruction Fetch Unit and like the pci_wait signal, Program Counter is updated during the clock rising edges if and only if this signal is not set. When this signal is set that means an unresolved (load/store) hazard exists in the pipeline which ends up with pipeline stall. Program Counter and also IF_ID are not updated during to stall because it is desired to not to lose instruction fetched and decoded during stall. NOP instruction is inserted in ID_EX stage when this signal is set.

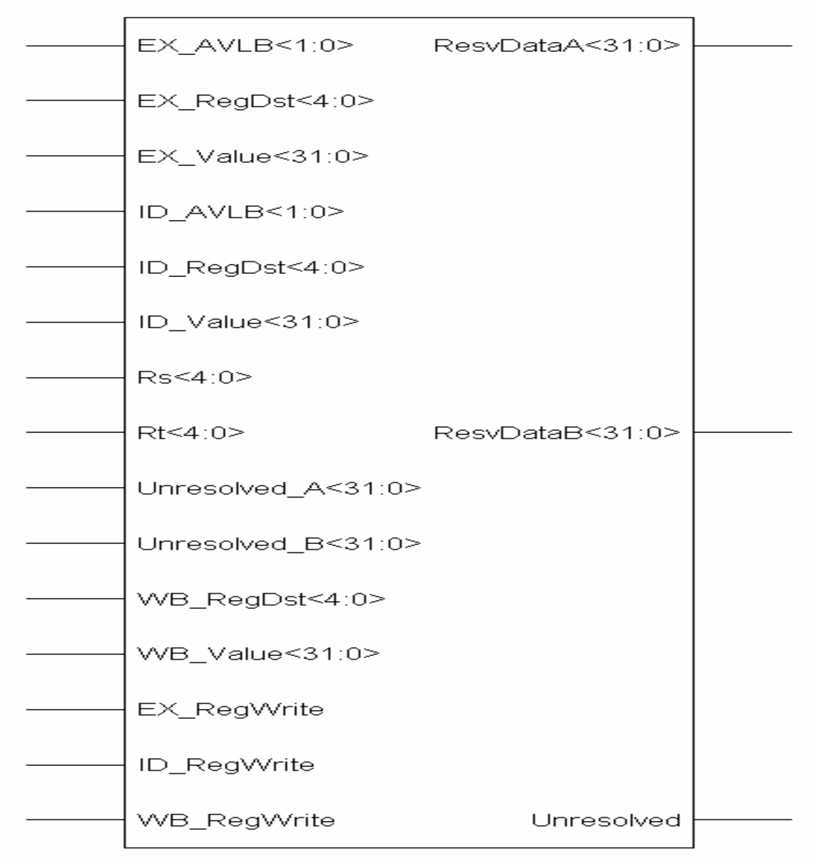


Figure 4.5: Input/Output Signals of Forwarding and Hazard Detection Unit

Input signals are as the following;

ID_AVLB (2 bit), ID_RegDst (5 bit), ID_Value (32 bit), ID_RegWrite (1 bit): These signals come from ID_EX register block which is located between ID and EX stages. These values are written by the instruction which is currently in EX stage and these values are used to determine the ResvDataA and ResvDataB. ID_AVLB and ID_RegDst are used to determine the value of Unresolved.

EX_AVLB (2 bit), EX_RegDst (5 bit), EX_Value (32 bit), EX_RegWrite (1 bit): These signals come from EX_MEM register block which is located between EX and MEM stages. These values are written by the instruction which is currently in MEM stage and these values are used to determine the ResvDataA and ResvDataB. EX_AVLB is not used for any purpose.

WB_RegDst (5 bit), WB_Value (32 bit) and WB_RegWrite (1 bit): These signals come from MEM_WB register block which is located between MEM and WB stages. These values are written by the instruction which is currently in WB stage and these values are used to determine the ResvDataA and ResvDataB.

Rs (5 bit), Rt (5 bit), Unresolved_A (32 bit) and Unresolved_B (32 bit): Instruction Decode Unit output signals.

4.1.3.2. Function of Forwarding and Hazard Detection Unit

The function of Forwarding and Hazard Detection Unit is to determine data hazards and if possible solving this hazards either by forwarding or stalling the pipeline.

4.1.4. Control Unit

The design of the Control Unit is realized by using HDL Design entry method. The hardware flow diagram of this building block is not given in APPENDIX C, because the outputs of this block goes to other blocks as input and all of this signals are defined in destination unit's flow diagrams.

4.1.4.1. Input/Output Signals of Control Unit

The connections of Control Unit with other units can be seen in Figure 4.1: Internal Structure of the Pipelined Processor. All Input/Output signals can be seen in Figure 4.6: Input/Output Signals of Control Unit. Output signals are as the following;

IF Control (3 bit): Signal goes to Instruction Fetch Unit and first bit (MSB), if set means beq instruction is present in decode stage, second bit, if set means bne instruction is present in decode stage and third bit (LSB), if set means either j, jal or jr instruction is present in decode stage.

ID Control (11 bit): Signal goes to Instruction decode unit and the control word bits are set according to instructions present in ID stage. The resulting signals describe the operands, destination register and effect the branch address calculation. The dependency between ID Control word, the instruction present in ID and the effected outputs are given in Table 4.3.

Table 4.3: ID_Control Signal Fields

10	9	8	7	6	5	4	3	2	1	0
beq, bne	jal	jr	Not Used	lw o/w Not Used		00XX→ ALUA, ALUB are registers values, Reg_Dest→ Rd 1X0X→ ALUA = 0, Reg_Dest→ Rd 1X1X→ ALUA = Shift Amount, Reg_Dest→ Rd For the following instructions if word start with 01, Reg_Dest→ Rt else Reg_Dest→ Rd X1X0→ ALUB = Zero Extended Immediate X1X1→ ALUB = Sign Extended Immediate				Not Used

EX Control (5 bit): Signal goes to ID_EX register block and consumed in EX stage. Signal identifies ALU operation applied to inputs at ALU ports and also called ALUOp signal. The numeric and literal ALUOp values are given in Table 4.4.

Table 4.4: EX_Control ALUOp Signal Values

Literal ALUOp	Numeric ALUOp	Comment
ALU_ADD	00000	rd <= rs+rt, signed, overflow exception generated
ALU_ADDU	00001	rd <= rs+rt, unsigned, overflow exception NOT generated
ALU_AND	00010	rd <= rs AND rt
ALU_EMPTY	00011	ALU_RESULT <= TRUE
ALU_MFHI	00100	ALU internal multiplication register to general purpose register

Literal ALUOp	Numeric ALUOp	Comment
		(GPR), rd <= HI
ALU_MFLO	00101	ALU internal multiplication register to GPR, rd <= LO
ALU_MTHI	00110	GPR to ALU internal multiplication Register, HI <= rs
ALU_MTLO	00111	GPR to ALU internal multiplication Register, LO <= rs
ALU_MULT	01000	HILO <= rs * rt, signed (not implemented)
ALU_MULTU	01001	HILO <= rs * rt, unsigned
ALU_NOR	01010	rd <= rs NOR rt
ALU_OR	01011	rd <= rs OR rt
ALU_SLL	01100	rd <= (rt << shift amount)
ALU_SLT	01101	rd <= (rs < rt), signed
ALU_SLTU	01110	rd <= (rs < rt), unsigned
ALU_SRL	01111	rd <= (rt >> sa)
ALU_SUB	10000	rd <= rs-rt, signed, overflow exception generated
ALU_SUBU	10001	rd <= rs-rt, unsigned, overflow exception NOT generated
ALU_XOR	10010	rd <= rs XOR rt
ALU_DATAB	10011	ALU_RESULT <= OperandB
ALU_BEQ	10100	if (op1 == op2) then branch, 18-bit signed offset added to PC, +-128KBytes
ALU_BNE	10101	if (op1 != op2) then branch, 18-bit signed offset added to PC, +-128KBytes
ALU_LUI	10110	rt <= (immediate<<16)
ALU_SW	10111	MEM[$rs + signed(Immediate)] <= rt
ALU_EXPT	11000	Undefined Instruction in Decode stage, Exception will be generated

MEM Control (2 bit): Signal goes to ID_EX register block and consumed in MEM stage. First bit (MSB) if set indicates a memory read operation will take place (e.g. for lw instruction) in MEM stage, second bit (LSB) if set indicates a memory write operation will take place (e.g. for sw instruction) in MEM stage.

WB Control (1 bit): Signal goes to ID_EX register block and consumed in WB stage. Signal is also called RegWrite and indicates a register write operation will take place in WB stage.

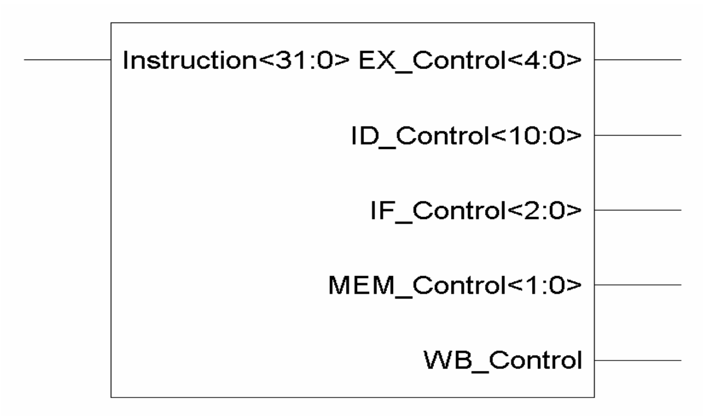


Figure 4.6: Input/Output Signals of Control Unit

Input signals are as the following;

Instruction (32 bit): Instruction Fetch Unit output signal.

4.1.4.2. Function of Control Unit

The function of Control Unit is to determine control signal values of an instruction which is in decode stage. These control signals move with the instruction throughout the pipeline and are wasted up until the last WB stage.

4.1.5. Execute Unit

The design of the Execute Unit is realized by using HDL Design entry method. The hardware flow diagrams of this building block are given in APPENDIX C, Figure C.4: Instruction Execute Unit Flow Diagram and Figure C.5: Instruction Execute Unit (continued) Flow Diagram.

4.1.5.1. Input/Output Signals of Execute Unit

The connections of Execute Unit with other units can be seen in Figure 4.1: Internal Structure of the Pipelined Processor. All Input/Output signals can be seen in Figure 4.7: Input/Output Signals of Execute Unit. Output signals are as the following;

Result (32 bit): Signal goes to Data Memory Unit and if result contains the memory address for load/store instructions, signal will be wasted in MEM stage, else if this result represents a register write operation signal will be wasted in WB stage.

OverFlow (1 bit): Signal goes to Exception Detection Unit and indicates that there is an arithmetic overflow occurred in signed operation.

Undefined (1 bit): Signal goes to Exception detection Unit and indicates that there was an undefined instruction (an instruction which is not defined in APPENDIX A, Implemented Subset of MIPS R2000 ISA) in ID stage in previous clock cycle.

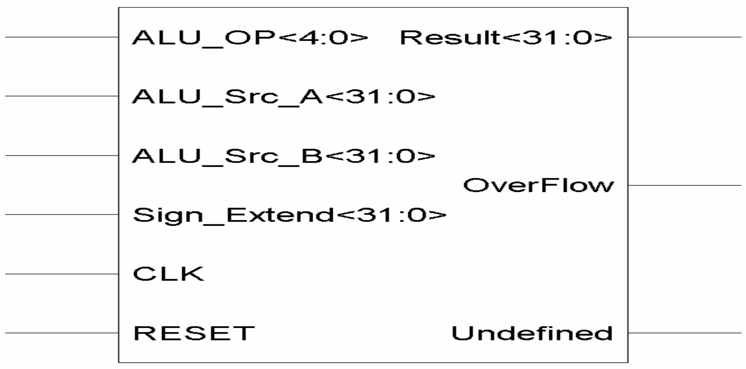


Figure 4.7: Input/Output Signals of Execute Unit

Input signals are as the following;

ALU_OP (5 bit): Control Unit output signal (EX_Control).

ALU_Src_A (32 bit): Instruction Decode Unit output signal (ALU_PORTA).
ALU_Src_B (32 bit): Instruction Decode Unit output signal (ALU_PORTB).
Sign_Extend (32 bit): Instruction Decode Unit output signal (Imm_Sign_Extended).

4.1.5.2. Function of Execute Unit

The function of Execute Unit is to realize the arithmetic and logical operations (Table 4.4) and generate overflow, undefined exception and result signals accordingly and to calculate memory addresses for data memory access operations.

4.1.6. Data Memory Unit

The design of the Data Memory Unit is realized by using HDL Design entry method. Data Memory Unit includes the subunit Data Memory (256x32bit block memory) from which data is retrieved with lw instruction and to which data is stored with sw instruction in every clock cycle. The hardware flow diagram of this building block is given in APPENDIX C, Figure C.6: Data Memory Unit Flow Diagram.

4.1.6.1. Input/Output Signals of Data Memory Unit

The connections of Data Memory Unit with other units can be seen in Figure 4.1: Internal Structure of the Pipelined Processor. All Input/Output signals can be seen in Figure 4.8: Input/Output Signals of Data Memory Unit.
Output signals are as the following;

Read_Data (32 bit): Signal goes to WB stage. Signal includes either the result of ALU operation obtained in EX stage in case MEM_Control signal does not indicate a memory read operation or the content of the data memory at Address signal in case MEM_Control signal indicates a memory read operation.

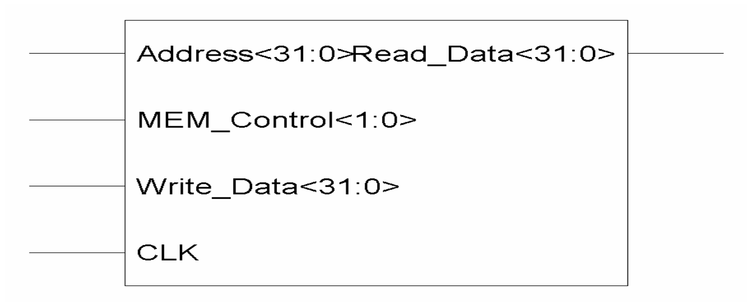


Figure 4.8: Input/Output Signals of Data Memory Unit

Input signals are as the following;
Address (32 bit): Execute Unit output signal (Result).
MEM_Control (2 bit): Control Unit output signal.
Write_Data (32 bit): Decode Unit output signal (ALU_PORTB).

4.1.6.2. Function of Data Memory Unit

The function of Memory Unit is to realize data memory access operations either read or write according to control signal MEM_Control. Data fetched from data memory is forwarded WB stage via Read_Data signal.

4.1.7. Exception Detection Unit

The design of the Exception Detection Unit is realized by using HDL Design entry method. The hardware flow diagram of this building block is given in APPENDIX C, Figure C.7: Exception Detection Unit Flow Diagram.

4.1.7.1. Input/Output Signals of Exception Detection Unit

The connections of Exception Detection Unit with other units can be seen in Figure 4.1: Internal Structure of the Pipelined Processor. All Input/Output signals can be seen in Figure 4.9: Input/Output Signals of Exception Detection Unit.

Output signals are as the following;

Exception (1 bit): Signal goes to Instruction Fetch Unit and to flush pin of internal register block s IF_ID, ID_EX and EX_MEM. Internal register blocks flush their contents when this signal is set. The internal register block MEM_WB will not be flushed, because exception did occur after the instructions which are currently (while exception occurred) in MEM and WB stage. It is allowed these instructions to complete. Instruction Fetch Unit uses this signal to determine next program counter. This signal has precedence over Branch instructions.

Exception Address (32 bit): Signal goes to Instruction Fetch Unit and is used an equated to Next Program Counter, when Exception signal is set. Byte address 0 in Instruction Memory is reserved for undefined instruction exception and there is an infinite loop located at this position. Byte address 8 is reserved for overflow exception and there is another infinite loop at this position. These 4 word address region can not be programmed by the user and can be thought as the exception handling routines.

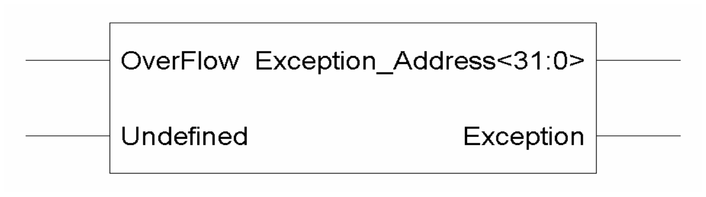


Figure 4.9: Input/Output Signals of Exception Detection Unit

Input signals are as the following;
<u>OverFlow (1 bit):</u> Execute Unit output signal.
<u>Undefined (1 bit):</u> Execute Unit output signal.

4.1.7.2. Function of Exception Detection Unit

The function of Exception Detection Unit is to set Exception signal in case either OverFlow or Undefined signal is set in EX stage. The exception address vectors are located at byte address 0 for undefined instruction and 8 for overflow exception in arithmetic instruction.

4.1.8. Register Blocks between Stages of Processor

Register Blocks are simply blocks which retain information for one clock cycle period and no arithmetic processing takes place on data. Starting with current clock edge, processing also starts and must end on next clock edge, because register blocks will be overwritten. These elements are placed between:

- Instruction Fetch - Instruction Decode (IF_ID Unit)
- Instruction Decode - Execute (ID_EX Unit)
- Execute - Data Memory (EX_MEM Unit)
- Data Memory – Instruction Decode (MEM_WB Unit)

The hardware flow diagram of this building block is given in APPENDIX C, Figure C.8: Register Block Unit Flow Diagram.

Input signals are as the following in general;

Unresolved (1 bit): Forwarding and Hazard Detection Unit output signal.

Wait Stages (1 bit): Instruction Fetch Unit output signal.

Exception (1 bit): Exception Detection Unit output signal.

4.2. External Structure of the Processor

In this section auxiliary structures are described in detail by stating their functions and input/output signals (in figures, inputs are placed on the left and outputs are placed on the right). Auxiliary structures are implemented to reveal the internal state of the processor by monitoring register blocks, which are placed between building blocks. In addition, auxiliary structures enable the user to manipulate the processor, e.g. user can reset the processor, execute the program on instruction memory for single step and program instruction memory of the processor externally.

Host monitor software (MIPS Monitor software described in section 2.4) writes to PCI and reads from PCI local addresses by using PlxApi library. PlxApi runs on host platform accessing to PCI bus which operates with 33 MHz and 32 bits wide. Pci_9030 interface monitors read and write transactions on PCI Bus initiated by MIPS Monitor software staying on local side which operates with 40 MHz local bus clock and 32 bits wide. The procedure how Pci_9030 interface detects transactions is described in [PLXSDK02]. Hence external structures of processor operate at 40 MHz while processor is operating at 20 MHz. This can be achieved by using CLKDLL Unit. CLKDLL Unit minimizes the clock skew between the input pad from which clock enters to FPGA and distributed clock across the FPGA. CLKDLL can also change the phase or the frequency of the clock by multiplying or dividing it by a constant. The clock frequency is divided by two to obtain the 20 MHz in the clock pins of the processor [XLBR04].

4.2.1. External Monitoring of the Processor

Reg Unit is developed for this purpose. MIPS Monitor software sends a PCI read request from a specified local address. Reg Unit (Figure 4.10) takes the local address from addr (26 bit) signal and compares it with the baddr (26 bit) signal. If they are equal and the rd signal is set, dout (32 bit) is forwarded to pci_9030 interface and then PCI bus. MIPS software reflects this information to the user via its graphical user interface.

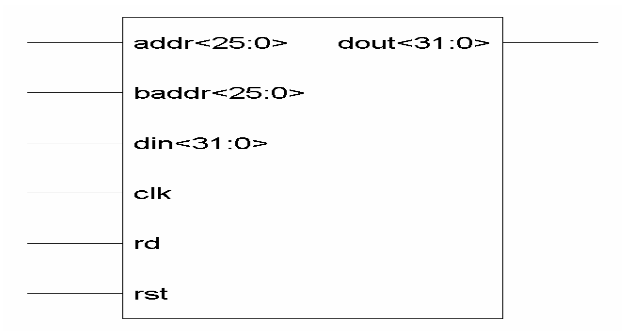


Figure 4.10: Input/Output Signals of Reg Unit

Base addresses from 1 to 10 (total 40 bytes) is reserved for monitoring of internal signals of the processor. The stage names attached to signal names represents the stage from which the signal is monitored (e.g. EX_Reg_Dst signal represents the destination register of the instruction which is currently in EX stage, similarly MEM_Reg_Dst represents the destination of the instruction in MEM stage and WB_Reg_Dst represents the destination of the instruction in WB stage). Base addresses their corresponding processor register blocks are given Table 4.5:

Table 4.5: Base Addresses of Processor's Internal Signals

Base Address	Internal Signals that can be Presented by MIPS Monitor Software
1	EX_OVFL, EX_Reg_Dst(5 bit), MEM_Reg_Dst(5 bit), WB_Reg_Dst(5 bit), ID_Incr_PC(8 bit), curr_pc(8 bit)
2	ID_Instruction (32 bit)
3	EX_ALUA (32 bit)
4	EX_ALUB (32 bit)
5	EX_ALU_RES (32 bit)
6	MEM_ADDR (32 bit)
7	MEM_WRITE_DATA (32 bit)
8	MEM_READ_DATA (32 bit)
9	WB_REG_WR_DATA (32 bit)
10	EN_RD, EN_WR, MEM_WAIT, ID_Unresolved, EX_AVLB(1:0)

4.2.2. External Manipulation of the Processor

Reg_Wr, Reg_Prg and Wait_Sm Units are developed to manipulate the state of the processor. MIPS Monitor software sends a PCI write request and data to a specified local address. According to data, next action will be determined.

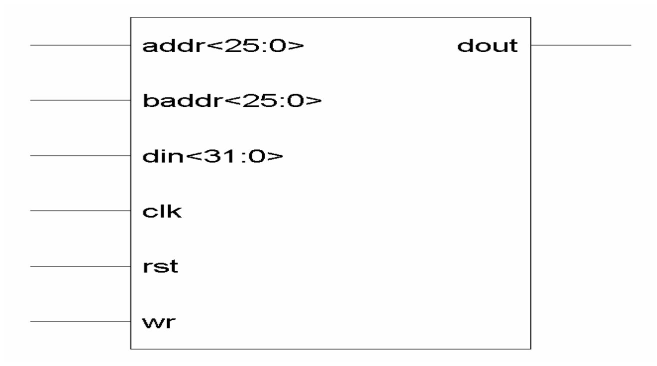


Figure 4.11: Input/Output Signals of Reg_Wr Unit

Reg_Wr Unit (Figure 4.11) which is developed to enable of external reset of the processor takes the local address from addr (26 bit) signal and compares it with the baddr (26 bit) signal. If they are equal and the wr signal is set and the din (32 bit) is equal to 2 then dout which is connected to reset pin of the processor is set.

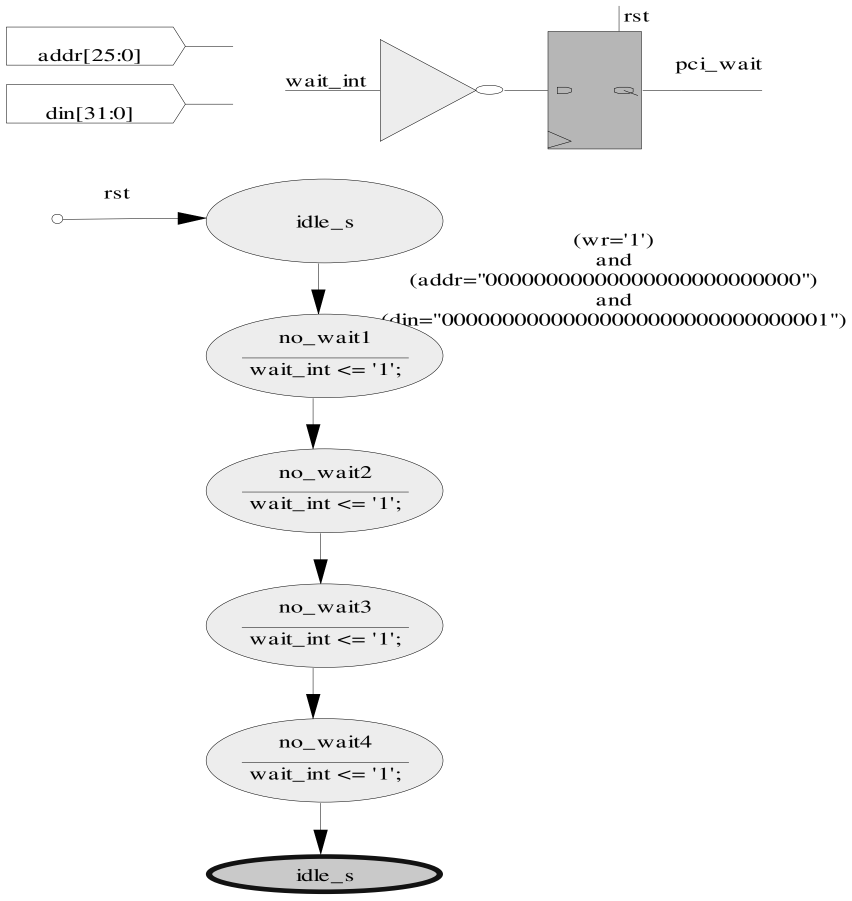


Figure 4.12: StateCAD Diagram of Wait_Sm Unit

Wait_Sm Unit (Figure 4.12) is developed to enable the processor for single step operation. The Input/Output signals are quite similar to Reg_wr Unit. The only difference is, instead of dout output, pci_wait signal is outputted from Wait_Sm Unit. The design of the Wait_Sm is realized by using state machine entry method StateCAD tool provided by Xilinx ISE. If addr signal is base address (base address 0 is reserved for single cycle operation), signal wr is set and din equals to 1, then pci_wait output stays reset during four clock cycles and processor is enabled to operate during this interval. Since processor operate at half frequency of external world, this duration corresponds to two processor clock cycles. Processor access memory and pipeline advances one step within this time.

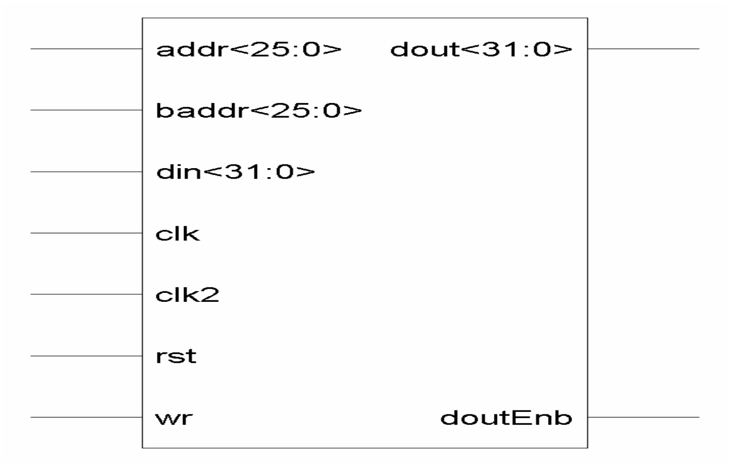


Figure 4.13: Input/Output Signals of Reg_Prg Unit

Reg_Prg Unit (Figure 4.13) which is developed for external programming and includes a program memory (256x32 bits). Reg_Prg takes the local address from addr (26 bit) signal and compares it with the baddr (26 bit) signal (Base address 11 is reserved for external programming). If they are equal and the wr signal is set and the din (32 bit) is not equal to X"FFFF_FFFF" then din is written at each clk edge (clk connected of external clock operating at 40 MHz) to internal memory. When din is equals to X"FFFF_FFFF", writing sequence to internal memory is finished and another writing sequence from Reg_Prg Unit memory to instruction memory of processor is started. This process is managed by clk2 signal (operating at 20 MHz) which is also the clock of the processor. Since both clock are the same, different clock domains problem is solved. It was foreseen as the fastest way during design.

CHAPTER 5

VERIFICATION OF MIPS PIPELINED ARCHITECTURE

The operation of the architecture is verified with MIPS Monitor software with following the steps:

- Verification of correct operation of instructions,
- Verification of proper hazard detection and solution,
- Verification of proper exception detection and handling.

The details of how the use of MIPS Monitor software is described in APPENDIX B, MIPS Monitor Software and the operation is described in section 2.4. The mnemonic names and the corresponding numeric values of MIPS registers are given at the end of in APPENDIX A, Implemented Subset of MIPS R2000 ISA in Table A.1.

5.1. Verification of Correct Operation of Instructions

Instructions described in APPENDIX A, Implemented Subset of MIPS R2000 ISA are tested and the procedure of testing and the observed results are stated in this section.

The test program given in Table 5.1 is written and then downloaded to processor to demonstrate that all instructions are tested. A requirement number (as R#) is given in the comment section of the code and the clock cycle in which the requirement is fulfilled is pointed out in the first column of Table 5.2.

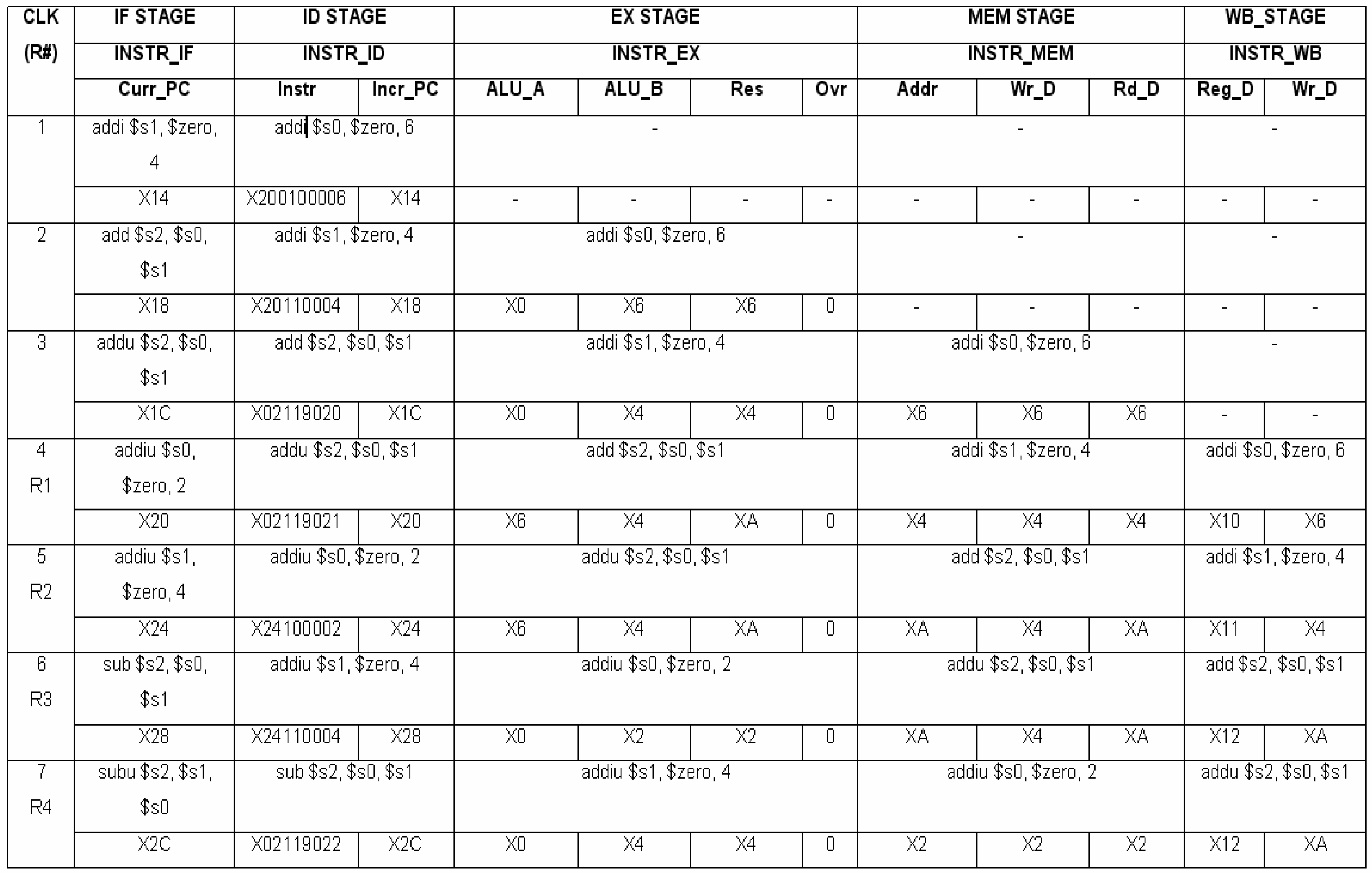
Results of operations and contents of stages are read by using MIPS Monitor software and results are tabulated in Table 5.2.

Table 5.1: Verification of Correct Instruction Operation

```
################################################
#
# TEST_1
#
# Created by Can Altınığneli
# To demonstrate the instructions defined in APPENDIX A correctly implemented
################################################
UNDEFINED:
beq $zero, $zero, UNDEFINED             # UNDEFINED EXCEPTION VECTOR
nop

OVERFLOW:
beq $zero, $zero, OVERFLOW              # OVERFLOW EXCEPTION VECTOR
nop

START:
#ADD, ADDI and ADDU are verified
addi    $s0, $zero, 0x6                 # $s0 shall = x6, DestAdr:16, R1
addi    $s1, $zero, 0x4                 # $s1 shall = x4, DestAdr:17, R2
add     $s2, $s0, $s1                   # $s2 shall = xA, DestAdr:18, R3
addu    $s2, $s0, $s1                   # $s2 shall = xA, DestAdr:18, R4

#ADDIU, SUB and SUBU are verified
addiu   $s0, $zero, 0x2                 # $s0 shall = x2, DestAdr:16, R5
addiu   $s1, $zero, 0x4                 # $s1 shall = x4, DestAdr:17, R6
sub     $s2, $s0, $s1                   # $s2 shall = xFFFF_FFFE, DestAdr:18, R7
subu    $s2, $s1, $s0                   # $s2 shall = x2, DestAdr:18, R8

#OR, ORI, AND, ANDI, XOR, XORI, NOR, SRL, SLL, LUI are verified
ori     $t0, $zero, 0xFFFF              # $t0 shall = x0000FFFF, DestAdr:8 , R9
lui     $t1, 0xFFFF                     # $t1 shall = xFFFF0000, DestAdr:9, R10
or      $t2, $t0,$t1                    # $t2 shall = xFFFF_FFFF, DestAdr:10,  R11
and     $t2, $t0,$t1                    # $t2 shall = x0000_0000, DestAdr:10,  R12
xor     $t2, $t0,$t1                    # $t2 shall = xFFFF_FFFF, DestAdr:10, R13
nor     $t2, $t0,$t1                    # $t2 shall = x0000_0000, DestAdr:10, R14
andi    $t0,$t0, 0x0000                 # $t0 shall = x0000_0000, DestAdr:8, R15
srl     $t1,$t1,16                      # $t1 shall = x0000_FFFF, DestAdr:9, R16
sll     $t1,$t1,16                      # $t1 shall = xFFFF_0000, DestAdr:9, R17
xori    $t1,$t1,0xFFFF                  # $t1 shall = xFFFF_FFFF, DestAdr:9, R18

#SLT, SLTI, BEQ, BNE, NOP are verified
LOOP_3TIMES:
```

```
subi    $s0, $s0, 1                 # $s0 shall = x1, DestAdr:16, R19
slti    $t0, $s0, 0x0               # $t0 shall = x1, if $s0 negative, signed comparison, R20
beq     $t0, $zero, LOOP_3TIMES
nop                                 #after 3 iterations exit from loop
slt     $t1, $s0, $zero             #s0 shall = xFFFF_FFFF, therefore $t1 shall = x1, R21
bne     $t1, $zero, JUMP_POINT
nop

#SLTIU, SLTU, MULTU, MFHI, MFLO, MTHI, MTLO, SW, LW, JR, J, JAL  are verified
MULTIPLY:
addi    $s0, $zero, -1              # $s0 shall = xFFFF_FFFF, DestAdr:16, R22
addi    $s1, $zero, -2              # $s1 shall = xFFFF_FFFE, DestAdr:17, R23
multu   $s0, $s1                    # HI shall  = xFFFF_FFFD, LO shall = x2
mfhi    $t0                         # $t0 shall = xFFFF_FFFD, DestAdr:8, R24
mflo    $t1                         # $t1 shall = x2, DestAdr:9, R25
mthi    $zero
mtlo    $zero
mfhi    $s0                         # $s0 shall = 0, DestAdr:16, R26
mflo    $s1                         # $s1 shall = 0, DestAdr:17, R27
addi    $s1, $s1, 0x4               # $s1 shall = 4, DestAdr:17, R28
sw      $t0, 0($s0)                 # MEM[0] shall store xFFFF_FFFD, R29
sw      $t1, 0($s1)                 # MEM[4] shall store x2, R30
jr      $ra                         # Jump after jal instruction, R31
nop
JUMP_POINT:
jal MULTIPLY
nop
lw      $t2, 0($s0)                 # MEM[0]-->$t2 shall = xFFFF_FFFD, DestAdr:10, R32
lw      $t3, 0($s1)                 # MEM[1]-->$t3 shall = x2, DestAdr:11, R33
sltiu   $t0, $t2, 1                 # $t0 shall = 0,because $t2 > 1, R34
bne     $t0, $zero, START           # shall not jump to START, R35
nop
sltu    $t0, $t2, $zero             # $t0 shall = 0,because $t2 > 0, R36
bne     $t0, $zero, START           # shall not jump to START, R37
nop
j START                             # shall jump to START, R38
nop
Eternity:
beq $zero, $zero, Eternity
nop
```

Table 5.2: Timing Diagram for Instruction Operation Verification



CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
1	addi $s1, $zero, 4	addi $s0, $zero, 6		-				-			-	
	X14	X200100006	X14	-	-	-	-	-	-	-	-	-
2	add $s2, $s0, $s1	addi $s1, $zero, 4		addi $s0, $zero, 6				-			-	
	X18	X20110004	X18	X0	X6	X6	0	-	-	-	-	-
3	addu $s2, $s0, $s1	add $s2, $s0, $s1		addi $s1, $zero, 4				addi $s0, $zero, 6			-	
	X1C	X02119020	X1C	X0	X4	X4	0	X6	X6	X6	-	-
4 R1	addiu $s0, $zero, 2	addu $s2, $s0, $s1		add $s2, $s0, $s1				addi $s1, $zero, 4			addi $s0, $zero, 6	
	X20	X02119021	X20	X6	X4	XA	0	X4	X4	X4	X10	X6
5 R2	addiu $s1, $zero, 4	addiu $s0, $zero, 2		addu $s2, $s0, $s1				add $s2, $s0, $s1			addi $s1, $zero, 4	
	X24	X24100002	X24	X6	X4	XA	0	XA	X4	XA	X11	X4
6 R3	sub $s2, $s0, $s1	addiu $s1, $zero, 4		addiu $s0, $zero, 2				addu $s2, $s0, $s1			add $s2, $s0, $s1	
	X28	X24110004	X28	X0	X2	X2	0	XA	X4	XA	X12	XA
7 R4	subu $s2, $s1, $s0	sub $s2, $s0, $s1		addiu $s1, $zero, 4				addiu $s0, $zero, 2			addu $s2, $s0, $s1	
	X2C	X02119022	X2C	X0	X4	X4	0	X2	X2	X2	X12	XA

Table 5.2: Timing Diagram for Instruction Operation Verification (continued)

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
8	ori $t0, $zero, -1	subu $s2, $s1, $s0		sub $s2, $s0, $s1				addiu $s1, $zero, 4			addiu $s0, $zero, 2	
R5	X30	X02309023	X30	X2	X4	Xfffffffe	0	X4	X4	X4	X10	X2
9	lui $t1, 65535	ori $t0, $zero, -1		subu $s2, $s1, $s0				sub $s2, $s0, $s1			addiu $s1, $zero, 4	
R6	X34	X3408ffff	X34	X4	X2	X2	0	Xfffffffe	X4	Xfffffffe	X11	X4
10	or $t2, $t0, $t1	lui $t1, 65535		ori $t0, $zero, -1				subu $s2, $s1, $s0			sub $s2, $s0, $s1	
R7	X38	X3C09ffff	X38	X0	X0000ffff	X0000ffff	0	X2	X2	X2	X12	Xfffffffe
11	and $t2, $t0, $t1	or $t2, $t0, $t1		lui $t1, 65535				ori $t0, $zero, -1			subu $s2, $s1, $s0	
R8	X3C	X01095025	X3C	X0	X0000ffff	Xffff0000	0	X0000ffff	X0000ffff	X0000ffff	X12	X2
12	xor $t2, $t0, $t1	and $t2, $t0, $t1		or $t2, $t0, $t1				lui $t1, 65535			ori $t0, $zero, -1	
R9	X40	X01095024	X40	X0000ffff	Xffff0000	Xffffffff	0	Xffff0000	X0000ffff	Xffff0000	X8	X0000ffff
13	nor $t2, $t0, $t1	xor $t2, $t0, $t1		and $t2, $t0, $t1				or $t2, $t0, $t1			lui $t1, 65535	
R10	X44	X01095026	X44	X0000ffff	Xffff0000	X0	0	Xffffffff	Xffff0000	Xffffffff	X9	Xffff0000
14	andi $t0, $t0, 0	nor $t2, $t0, $t1		xor $t2, $t0, $t1				and $t2, $t0, $t1			or $t2, $t0, $t1	
R11	X48	X01095027	X48	X0000ffff	Xffff0000	Xffffffff	0	X0	Xffff0000	X0	XA	Xffffffff
15	srl $t1, $t1, 16	andi $t0, $t0, 0		nor $t2, $t0, $t1				xor $t2, $t0, $t1			and $t2, $t0, $t1	
R12	X4C	X31080000	X4C	X0000ffff	Xffff0000	X0	0	Xffffffff	Xffff0000	Xffffffff	XA	X0
16	sll $t1, $t1, 16	srl $t1, $t1, 16		andi $t0, $t0, 0				nor $t2, $t0, $t1			xor $t2, $t0, $t1	
R13	X50	X00094C02	X50	X0000ffff	X0	X0	0	X0	Xffff0000	X0	XA	Xffffffff
17	xori $t1, $t1, -1	sll $t1, $t1, 16		srl $t1, $t1, 16				andi $t0, $t0, 0			nor $t2, $t0, $t1	
R14	X54	X00094C00	X54	X10	Xffff0000	X0000ffff	0	X0	X0	X0	XA	X0
18	addi $s0, $s0, -1	xori $t1, $t1, -1		sll $t1, $t1, 16				srl $t1, $t1, 16			andi $t0, $t0, 0	
R15	X58	X3929ffff	X58	X10	X0000ffff	Xffff0000	0	X0000ffff	Xffff0000	X0000ffff	X8	X0

Table 5.2: Timing Diagram for Instruction Operation Verification (continued)

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
19	slti $t0, $s0, 0	addi $s0, $s0, -1		xori $t1, $t1, -1				sll $t1, $t1, 16			srl $t1, $t1, 16	
R16	X5C	X2210ffff	X5C	Xffff0000	X0000ffff	Xffffffff	0	Xffff0000	X0000ffff	Xffff0000	X9	X0000ffff
20 R17	beq $zero, $t0, -3	slti $t0, $s0, 0		addi $s0, $s0, -1				xori $t1, $t1, -1			sll $t1, $t1, 16	
	X60	X2A080000	X60	X2	Xffffffff	X1	0	Xffffffff	X0000ffff	Xffffffff	X9	Xffff0000
21	nop	beq $zero, $t0, -3		slti $t0, $s0, 0				addi $s0, $s0, -1			xori $t1, $t1, -1	
R18	X64	X1008fffd	X64	X1	X0	X0	0	X1	Xffffffff	X1	X9	Xffffffff
22	addi $s0, $s0, -1	nop		beq $zero, $t0, -3				slti $t0, $s0, 0			addi $s0, $s0, -1	
R19	X58	X00000000	X68	X0	X0	X0	0	X0	X0	X0	X10	X1
23	slti $t0, $s0, 0	addi $s0, $s0, -1		nop				beq $zero, $t0, -3			slti $t0, $s0, 0	
	X5C	X2210ffff	X5C	X0	X0	X0	0	X0	X0	X0	X8	X0
24	beq $zero, $t0, -3	slti $t0, $s0, 0		addi $s0, $s0, -1				nop			beq $zero, $t0, -3	
	X60	X2A080000	X60	X1	Xffffffff	X0	0	X0	X0	X0	X1f	X0
25	nop	beq $zero, $t0, -3		slti $t0, $s0, 0				addi $s0, $s0, -1			nop	
	X64	X1008fffd	X64	X0	X0	X0	0	X0	Xffffffff	X0	X0	X0
26	addi $s0, $s0, -1	nop		beq $zero, $t0, -3				slti $t0, $s0, 0			addi $s0, $s0, -1	
	X58	X00000000	X68	X0	X0	X0	X0	X0	X0	X0	X10	X0
27	slti $t0, $s0, 0	addi $s0, $s0, -1		nop				beq $zero, $t0, -3			slti $t0, $s0, 0	
	X5C	X2210ffff	X5C	X0	X0	X0	0	X0	X0	X0	X8	X0

Table 5.2: Timing Diagram for Instruction Operation Verification (continued)

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
28	beq $zero, $t0, -3	slti $t0, $s0, 0		addi $s0, $s0, -1				nop			beq $zero, $t0, -3	
	X60	X2A080000	X60	X0	Xffffffff	Xffffffff	0	X0	X0	X0	X1f	X0
29	nop	beq $zero, $t0, -3		slti $t0, $s0, 0				addi $s0, $s0, -1			nop	
	X64	X1008fffg	X64	Xffffffff	X0	X1	0	Xffffffff	Xffffffff	Xffffffff	X0	X0
30 R21	slt $t1, $s0, $zero	nop		beq $zero, $t0, -3				slti $t0, $s0, 0			addi $s0, $s0, -1	
	X68	X0	X68	X0	X1	X0	0	X1	X0	X1	X10	Xffffffff
31 R20	bne $t1, $zero, 15	slt $t1, $s0, $zero		nop				beq $zero, $t0, -3			slti $t0, $s0, 0	
	X6C	X0200482A	X6C	X0	X0	X0	X0	X0	X1	X0	X8	X1
32	nop	bne $t1, $zero, 15		slt $t1, $s0, $zero				nop			beq $zero, $t0, -3	
	X70	X1409000f	X70	Xffffffff	X0	X1	0	X0	X0	X0	X1f	X0
33	jal 0x001D	nop		bne $t1, $zero, 15				slt $t1, $s0, $zero			nop	
	XAC	X00000000	X74	X0	X1	X0	0	X1	X0	X1	X0	X0
34 R21	nop	jal 0x001D		nop				bne $t1, $zero, 15			slt $t1, $s0, $zero	
	XB0	X0c00001D	XB0	X0	X0	X0	X0	X0	X1	X0	X9	X1
35	addi $s0, $zero, -1	nop		jal 0x001D				nop			bne $t1, $zero, 15	
	X74	X00000000	XB4	X0	XB0	XB0	0	X0	X0	X0	X0	X0

Table 5.2: Timing Diagram for Instruction Operation Verification (continued)

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
36	addi $s1, $zero, -2	addi $s0, $zero, -1		nop				jal 0x001D			nop	
	X78	X2010ffff	X78	X0	X0	X0	0	XB0	XB0	XB0	X0	X0
37	multu $s0, $s1	addi $s1, $zero, -2		addi $s0, $zero, -1				nop			jal 0x001D	
	X7C	X2011fffe	X7C	X0	Xffffffff	Xffffffff	0	X0	X0	X0	X1f	XB0
38	mfhi $t0	multu $s0, $s1		addi $s1, $zero, -2				addi $s0, $zero, -1			nop	
	X80	X02110019	X80	X0	Xfffffffe	Xfffffffe	0	Xffffffff	Xffffffff	Xffffffff	X0	X0
39	mflo $t1	mfhi $t0		multu $s0, $s1				addi $s1, $zero, -2			addi $s0, $zero, -1	
R22	X84	X00004010	X84	Xffffffff	Xfffffffe	X0	0	Xfffffffe	Xfffffffe	Xfffffffe	X10	Xffffffff
40	mthi $zero	mflo $t1		mfhi $t0				multu $s0, $s1			addi $s1, $zero, -2	
R23	X88	X00004812	X88	X0	X4010	Xfffffffd	0	X0	Xfffffffe	X0	X11	Xfffffffe
41	mtlo $zero	mthi $zero		mflo $t1				mfhi $t0			multu $s0, $s1	
	X8C	X00000011	X8C	X0	X4812	X2	0	Xfffffffd	X4010	Xfffffffd	X0	X0
42	mfhi $s0	mtlo $zero		mthi $zero				mflo $t1			mfhi $t0	
R24	X90	X00000013	X90	X0	X11	X0	0	X2	X4812	X2	X8	Xfffffffd
43	mflo $s1	mfhi $s0		mtlo $zero				mthi $zero			mflo $t1	
R25	X94	X00008010	X94	X0	X13	X0	0	X0	X11	X0	X9	X2
44	addi $s1, $s1, 4	mflo $s1		mfhi $s0				mtlo $zero			mthi $zero	
	X98	X00008812	X98	X0	X8010	X0	0	X0	X13	X0	X0	X0
45	sw $t0, 0($s0)	addi $s1, $s1, 4		mflo $s1				mfhi $s0			mtlo $zero	
	X9C	X22310004	X9C	X0	X8812	X0	0	X0	X8010	X0	X0	X0

Table 5.2: Timing Diagram for Instruction Operation Verification (continued)

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
46	sw $t1, 0($s1)	sw $t0, 0($s0)		addi $s1, $s1, 4				mflo $s1			mfhi $s0	
R26	XA0	XAE080000	XA0	X0	X4	X4	0	X0	X8812	X0	X10	X0
47	jr $ra	sw $t1, 0($s1)		sw $t0, 0($s0)				addi $s1, $s1, 4			mflo $s1	
R27	XA4	XAE290000	XA4	X0	Xfffffffd	X0	0	X4	X4	X4	X11	X0
48	nop	jr $ra		sw $t1, 0($s1)				sw $t0, 0($s0)			addi $s1, $s1, 4	
R28 R29	XA8	X03E00008	XA8	X4	X2	X4	0	X0	Xfffffffd	X0	X11	XX
49	nop	nop		jr $ra				sw $t1, 0($s1)			sw $t0, 0($s0)	
R30 R31	XB0	X00000000	XAC	XB0	X8	X0	0	X4	X2	X4	X0	X0
50	lw $t2, 0($s0)	nop		nop				jr $ra			sw $t1, 0($s1)	
	XB4	X00000000	XB4	X0	X0	X0	0	X0	X8	X0	X0	
51	lw $t3, 0($s1)	lw $t2, 0($s0)		nop				nop			jr $ra	
	XB8	X8E0A0000	XB8	X0	X0	X0	0	X0	X0	X0	X0	X0
52	sltiu $t0, $t2, 1	lw $t3, 0($s1)		lw $t2, 0($s0)				nop			nop	
	XBC	X8E2B0000	XBC	X0	X0	X0	0	X0	X0	X0	X0	X0
53	bne $t0, $zero, -45	sltiu $t0, $t2, 1		lw $t3, 0($s1)				lw $t2, 0($s0)			nop	
	XC0	X2D480001	XC0	X4	X0	X4	0	X0	X0	Xfffffffd	X0	X0
54	nop	bne $t0, $zero, -45		sltiu $t0, $t2, 1				lw $t3, 0($s1)			lw $t2, 0($s0)	
R32	XC4	X1408ffd3	XC4	Xfffffffd	X1	X0	0	X4	X0	X2	XA	Xfffffffd

Table 5.2: Timing Diagram for Instruction Operation Verification (continued)

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
55 R33	sltu $t0, $t2, $zero	nop		bne $t0, $zero, -45				sltiu $t0, $t2, 1			lw $t3, 0($s1)	
R35	XC8	X00000000	XC8	X0	X0	X0	0	X0	X1	X0	XB	X2
56	bne $t0, $zero, -48	sltu $t0, $t2, $zero		nop				bne $t0, $zero, -45			sltiu $t0, $t2, 1	
R34	XCC	X0140402B	XCC	X0	X0	X0	0	X0	X0	X0	X8	X0
57	nop	bne $t0, $zero, -48		sltu $t0, $t2, $zero				nop			bne $t0, $zero, -45	
	XD0	X1408ffd0	XD0	Xfffffffd	X0	X0	0	X0	X0	X0	X1f	X0
58	j 0x0004	nop		bne $t0, $zero, -48				sltu $t0, $t2, $zero			nop	
R37	XD4	X00000000	XD4	X0	X0	X0	X0	X0	X0	X0	X0	X0
59	nop	j 0x0004		nop				bne $t0, $zero, -48			sltu $t0, $t2, $zero	
R36	XD8	X08000004	XD8	X0	X0	X0	X0	X0	X0	X0	X8	X0
60 R38	addi $s0, $zero, 6	nop		j 0x0004				nop			bne $t0, $zero, -48	
	X10	X00000000	XDC	X0	X4	X0	0	X0	X0	X0	X1f	X0

5.2. Verification of Hazard Detection and Handling

The test program given in Table 5.3 is downloaded to processor to demonstrate that Data Hazards are resolved using the feedback paths between stages. The pipeline is halted in case of the presence of an unresolved hazard. A requirement number (as R#) is given in the comment section of the code and the clock cycle in which the requirement is fulfilled is pointed out in the first column of Table 5.4.

Table 5.3: Verification of Hazard Detection and Handling

```
################################################
#
# TEST_2
#
# Created by Can Altınığneli
# To demonstrate data hazards are correctly handled
################################################
UNDEFINED:
beq $zero, $zero, UNDEFINED           # UNDEFINED EXCEPTION VECTOR
nop

OVERFLOW:
beq $zero, $zero, OVERFLOW            # OVERFLOW EXCEPTION VECTOR
nop

START:
add  $a0, $zero,$zero                 # $a0 shall = 0, DestAdr:x4, R1
addi $t1, $zero, 5                    # $t1 shall = 5, DestAdr:x9, R2
addi $t0, $zero, 1                    # $t0 shall = 1, DestAdr:x8, R3
nop
nop
nop

# Feedback path exists between ID and EX, MEM, WB stages. Most recent
# result is written to destination register. The code snippet below shows
# that there is no need to wait until to WB stage of an instruction and
# architecture correctly handles up-to-dateness problem, R4
add $t0, $t0, $t0                     # $t0 = $t0 + $t0, $t0 shall = x2
add $t0, $t0, $t0                     # $t0 = $t0 + $t0, $t0 shall = x4
add $t0, $t0, $t0                     # $t0 = $t0 + $t0, $t0 shall = x8
add $t0, $t0, $t0                     # $t0 = $t0 + $t0, $t0 shall = x16
nop
nop
nop


# Feedback path exists between ID and EX stages.Data Hazard resolved, R5
subi $t0, $t0, 1                      # $t0 = $t0 - 1, $t0 shall = xF
subi $t0, $t0, 3                      # $t0 = $t0 - 3, $t0 shall = xC
nop
nop
nop
```

```
# Feedback path exists between ID and MEM stages.Data Hazard resolved, R6
subi $t0, $t0, 1                        # $t0 = $t0 - 1, $t0 shall = xB
nop
subi $t0, $t0, 3                        # $t0 = $t0 - 3, $t0 shall = x8
nop
nop
nop


# Feedback path exists between ID and WB stages.Data Hazard resolved, R7
subi $t0, $t0, 2                        # $t0 = $t0 - 3, $t0 shall = x5
nop
nop
subi $t0, $t0, 4                        # $t0 = $t0 - 5, $t0 shall = x2
nop
nop
nop
sw $t0, 0($a0)                          # MEM[0] shall store x2
nop
nop

# Although feedback path exists between ID and EX stages,
# Data Hazard can not be resolved by this path. A NOP instruction
# is inserted between "add" and "lw" instructions and hazard is resolved
# by feedback path between ID and MEM stages on the next clock cycle, R8
lw $t0, 0($a0)                          # MEM[0]-->$t0 = x2, DestAdr:8
add $t2, $t0, $t1                       # $t2 = $t0 + $t1, $t2 shall = x7, DestAdr:10, R9

Eternity:
beq $zero, $zero, Eternity     # Infinite Loop
nop
```

Results of operations and contents of stages are read by using MIPS Monitor software and results are tabulated in Table 5.4.

Table 5.4: Timing Diagram for Handling Hazard Verification

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
1	addi $t1, $zero, 5	add $a0, $zero, $zero		-				-			-	
	X14	X00002020	X14	-	-	-	-	-	-	-	-	-
2	addi $t0, $zero, 1	addi $t1, $zero, 5		add $a0, $zero, $zero				-			-	
	X18	X20090005	X18	X0	X0	X0	0	-	-	-	-	-
3	nop	addi $t0, $zero, 1		addi $t1, $zero, 5				add $a0, $zero, $zero			-	
	X1C	X20080001	X1C	X0	X5	X5	0	X0	X0	X0	-	-
4	nop	nop		addi $t0, $zero, 1				addi $t1, $zero, 5			add $a0, $zero, $zero	
R1	X20	X00000000	X20	X0	X1	X1	0	X5	X5	X5	X4	X0
5	nop	nop		nop				addi $t0, $zero, 1			addi $t1, $zero, 5	
R2	X24	X00000000	X24	X0	X0	X0	0	X1	X1	X1	X9	X5
6	add $t0, $t0, $t0	nop		nop				nop			addi $t0, $zero, 1	
R3	X28	X00000000	X28	X0	X0	X0	0	X0	X0	X0	X8	X1
7	add $t0, $t0, $t0	add $t0, $t0, $t0		nop				nop			nop	
	X2C	X01084020	X2C	X0	X0	X0	0	X0	X0	X0	X0	X0
8	add $t0, $t0, $t0	add $t0, $t0, $t0		add $t0, $t0, $t0				nop			nop	
	X30	X01084020	X30	X1	X1	X2	0	X0	X0	X0	X0	X0
9	add $t0, $t0, $t0	add $t0, $t0, $t0		add $t0, $t0, $t0				add $t0, $t0, $t0			nop	
	X34	X01084020	X34	X2	X2	X4	0	X2	X1	X2	X0	X0
10	nop	add $t0, $t0, $t0		add $t0, $t0, $t0				add $t0, $t0, $t0			add $t0, $t0, $t0	
	X38	X01084020	X38	X4	X4	X8	0	X4	X2	X4	X8	X2
11	nop	nop		add $t0, $t0, $t0				add $t0, $t0, $t0			add $t0, $t0, $t0	
	X3C	X00000000	X3C	X8	X8	X10	0	X8	X4	X8	X8	X4
12	nop	nop		nop				add $t0, $t0, $t0			add $t0, $t0, $t0	
	X40	X00000000	X40	X0	X0	X0	0	X10	X8	X10	X8	X8

Table 5.4: Timing Diagram for Handling Hazard Verification(continued)

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
13	addi $t0, $t0, -1	nop		nop				nop			add $t0, $t0, $t0	
R4	X44	X00000000	X44	X0	X0	X0	0	X0	X0	X0	X8	X10
14	addi $t0, $t0, -3	addi $t0, $t0, -1		nop				nop			nop	
	X48	X2108ffff	X48	X0	X0	X0	0	X0	X0	X0	X0	X0
15	nop	addi $t0, $t0, -3		addi $t0, $t0, -1				nop			nop	
R5	X4C	X2108fffd	X4C	X10	Xffffffff	Xf	0	X0	X0	X0	X0	X0
16	nop	nop		addi $t0, $t0, -3				addi $t0, $t0, -1			nop	
	X50	X00000000	X50	Xf	Xfffffffd	XC	0	Xf	Xffffffff	Xf	X0	X0
17	nop	nop		nop				addi $t0, $t0, -3			addi $t0, $t0, -1	
	X54	X00000000	X54	X0	X0	X0	0	XC	Xfffffffd	XC	X8	Xf
18	addi $t0, $t0, -1	nop		nop				nop			addi $t0, $t0, -3	
	X58	X00000000	X58	X0	X0	X0	0	X0	X0	X0	X8	XC
19	nop	addi $t0, $t0, -1		nop				nop			nop	
	X5C	X2108ffff	X5C	X0	X0	X0	0	X0	X0	X0	X0	X0
20	addi $t0, $t0, -3	nop		addi $t0, $t0, -1				nop			nop	
	X60	X00000000	X60	XC	Xffffffff	XB	0	X0	X0	X0	X0	X0
21	nop	addi $t0, $t0, -3		nop				addi $t0, $t0, -1			nop	
R6	X64	X2108fffd	X64	X0	X0	X0	0	XB	Xffffffff	XB	X0	X0
22	nop	nop		addi $t0, $t0, -3				nop			addi $t0, $t0, -1	
	X68	X00000000	X68	XB	Xfffffffd	X8	0	X0	X0	X0	X8	XB
23	nop	nop		nop				addi $t0, $t0, -3			nop	
	X6C	X00000000	X6C	X0	X0	X0	0	X8	Xfffffffd	X8	X0	X0
24	addi $t0, $t0, -2	nop		nop				nop			addi $t0, $t0, -3	
	X70	X00000000	X70	X0	X0	X0	0	X0	X0	X0	X8	X8

Table 5.4: Timing Diagram for Handling Hazard Verification(continued)

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
25	nop	addi $t0, $t0, -2		nop				nop			nop	
	X74	X2108fffe	X74	X0	X0	X0	0	X0	X0	X0	X0	X0
26	nop	nop		addi $t0, $t0, -2				nop			nop	
	X78	X00000000	X78	X8	Xfffffffe	X6	0	X0	X0	X0	X0	X0
27	addi $t0, $t0, -4	nop		nop				addi $t0, $t0, -2			nop	
	X7C	X00000000	X7C	X0	X0	X0	0	X6	Xfffffffe	X6	X0	X0
28	nop	addi $t0, $t0, -4		nop				nop			addi $t0, $t0, -2	
R7	X80	X2108fffc	X80	X0	X0	X0	0	X0	X0	X0	X8	X6
29	nop	nop		addi $t0, $t0, -4				nop			nop	
	X84	X00000000	X84	X6	Xfffffffc	X2	0	X0	X0	X0	X0	X0
30	nop	nop		nop				addi $t0, $t0, -4			nop	
	X88	X00000000	X88	X0	X0	X0	0	X2	Xfffffffc	X2	X0	X0
31	sw $t0, 0($a0)	nop		nop				nop			addi $t0, $t0, -4	
	X8C	X00000000	X8C	X0	X0	X0	0	X0	X0	X0	X8	X2
32	nop	sw $t0, 0($a0)		nop				nop			nop	
	X90	XAC880000	X90	X0	X0	X0	0	X0	X0	X0	X0	X0
33	nop	nop		sw $t0, 0($a0)				nop			nop	
	X94	X00000000	X94	X0	X2	X0	0	X0	X0	X0	X0	X0
34	lw $t0, 0($a0)	nop		nop				sw $t0, 0($a0)			nop	
	X98	X00000000	X98	X0	X0	X0	0	X0	X2	X0	X0	X0
35	add $t2, $t0, $t1	lw $t0, 0($a0)		nop				nop			sw $t0, 0($a0)	
	X9C	X8C880000	X9C	X0	X0	X0	0	X0	X0	X0	X0	X0
36	beq $zero, $zero, -1	add $t2, $t0, $t1		lw $t0, 0($a0)				nop			nop	
	XA0	X01095020	XA0	X0	X0	X0	0	X0	X0	X0	X0	X0

Table 5.4: Timing Diagram for Handling Hazard Verification(continued)

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
37	beq $zero, $zero, -1	add $t2, $t0, $t1		nop				lw $t0, 0($a0)			nop	
R8	XA0	X01095020	XA0	X0	X0	X0	0	X0	X0	X2	X0	X0
38	nop	beq $zero, $zero, -1		add $t2, $t0, $t1				nop			lw $t0, 0($a0)	
	XA4	X1000ffff	XA4	X2	X5	X7	0	X0	X0	X0	X8	X2
39	beq $zero, $zero, -1	nop		beq $zero, $zero, -1				add $t2, $t0, $t1			nop	
	XA0	X00000000	XA8	X0	X0	X0	0	X7	X5	X7	X0	X0
40	nop	beq $zero, $zero, -1		nop				beq $zero, $zero, -1			add $t2, $t0, $t1	
R9	XA4	X1000ffff	XA4	X0	X0	X0	0	X0	X0	X0	XA	X7

5.3. Verification of Exception Handling

First, the test program given in Table 5.5 is downloaded to processor to demonstrate that “ADDU” and “ADD” instructions generate exceptions according to definitions in APPENDIX A, Implemented Subset of MIPS R2000 ISA.

A requirement number (as R#) is given in the comment section of the code and the clock cycle in which the requirement is fulfilled is pointed out in the first column of Table 5.6.

Table 5.5: Verification of Exception Handling “ADDU” and “ADD”

```
################################################
#
# TEST_3
#
# Created by Can Altınığneli
# To demonstrate ADDU and ADD instructions generate overflow exceptions according to APPENDIX A.
################################################
UNDEFINED:
beq $zero, $zero, UNDEFINED          # UNDEFINED EXCEPTION VECTOR
nop

OVERFLOW:
beq $zero, $zero, OVERFLOW           # OVERFLOW EXCEPTION VECTOR
nop

START:
add   $t0, $zero,$zero               # $t0 shall = 0
lui   $t0, 0x8000                    # $t0 shall = x8000_0000,  DestAdr:8, R1
addu  $t1, $t0, $t0                  # $t1 shall = x0000_0000,  DestAdr:9, No Exception shall be
generated, R2
add   $t0, $t0, $t0                  # $t0 shall = x0000_0000,  DestAdr:8, Exception shall be
generated,
                                     # and pipeline register blocks IF_ID, ID_EX and EX_MEM are
flushed, R3
Eternity:
beq $zero, $zero, Eternity
nop
```

Results of operations and contents of stages are read by using MIPS Monitor software and results are tabulated in Table 5.6.

Table 5.6: Timing Diagram for Exception Handling of ADDU and ADD

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
1	lui $t0, 32768	add $t0, $zero, $zero		-				-			-	
	X14	X00004020	X14	-	-	-	-	-	-	-	-	-
2	addu $t1, $t0, $t0	lui $t0, 32768		add $t0, $zero, $zero				-			-	
	X18	X3C088000	X18	X0	X0	X0	0	-	-	-	-	-
3	add $t1, $t0, $t0	addu $t1, $t0, $t0		lui $t0, 32768				add $t0, $zero, $zero			-	
	X1C	X01084821	X1C	X0	X8000	X80000000	0	X0	X0	X0	-	-
4 R2	beq $zero, $zero, -1	add $t1, $t0, $t0		addu $t1, $t0, $t0				lui $t0, 32768			add $t0, $zero, $zero	
	X20	X01084820	X20	X80000000	X80000000	X0	0	X80000000	X8000	X80000000	X8	X0
5	nop	beq $zero, $zero, -1		add $t1, $t0, $t0				addu $t1, $t0, $t0			lui $t0, 32768	
R1 R3	X24	X1000ffff	X24	X80000000	X80000000	X0	1	X0	X80000000	X0	X8	X80000000
6 R2	beq $zero, $zero, -1	nop		beq $zero, $zero, -1				add $t1, $t0, $t0			addu $t1, $t0, $t0	
R3	X8	X00000000	X0	X0	X0	X0	0	X0	X0	X0	X9	X0
7	nop	beq $zero, $zero, -1		nop				beq $zero, $zero, -1			add $t1, $t0, $t0	
	XC	X1000ffff	XC	X0	X0	X0	0	X0	X0	X0	X0	X0

After verifying “ADDU” and “ADD” instructions exception handling mechanism, the test program given in Table 5.7 is downloaded to processor to demonstrate that “SUBU” and “SUB” instructions generate exceptions according to definitions in APPENDIX A, Implemented Subset of MIPS R2000 ISA.

A requirement number (as R#) is given in the comment section of the code and the clock cycle in which the requirement is fulfilled is pointed out in the first column Table 5.8.

Table 5.7: Verification of Exception Handling “SUBU” and “SUB”

```
#################################################
#
# TEST_4
#
# Created by Can Altınığneli
# To demonstrate ADDU and ADD instructions generate overflow exceptions according to APPENDIX A.
#################################################
UNDEFINED:
beq $zero, $zero, UNDEFINED          # UNDEFINED EXCEPTION VECTOR
nop

OVERFLOW:
beq $zero, $zero, OVERFLOW           # OVERFLOW EXCEPTION VECTOR
nop

START:
add   $t0, $zero, $zero              # $t0 shall = 0
lui   $t0, 0x8000                    # $t0 shall = x8000_0000,  DestAdr:8
addi  $t1, $zero, 1                  # $t1 shall = 1,  DestAdr:9
subu  $t2, $t0, $t1                  # No Exception shall be generated, R1
sub   $t2, $t0, $t1                  # Exception shall be generated, R2

Eternity:
beq $zero, $zero, Eternity
nop
```

Results of operations and contents of stages are read by using MIPS Monitor software and results are tabulated in Table 5.8.

Table 5.8: Timing Diagram for Exception Handling of SUBU and SUB

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
1	lui $t0, 32768	add $t0, $zero, $zero		-				-			-	
	X14	X00004020	X14	-	-	-	-	-	-	-	-	-
2	addi $t1, $zero, 1	lui $t0, 32768		add $t0, $zero, $zero				-			-	
	X18	X3C088000	X18	X0	X0	X0	0	-	-	-	-	-
3	subu $t2, $t0, $t1	addi $t1, $zero, 1		lui $t0, 32768				add $t0, $zero, $zero			-	
	X1C	X20090001	X1C	X0	X8000	X80000000	0	X0	X0	X0	-	-
4	sub $t2, $t0, $t1	subu $t2, $t0, $t1		addi $t1, $zero, 1				lui $t0, 32768			add $t0, $zero, $zero	
	X20	X01095023	X20	X0	X1	X1	0	X80000000	X8000	X80000000	X8	X0
5	beq $zero, $zero, -1	sub $t2, $t0, $t1		subu $t2, $t0, $t1				addi $t1, $zero, 1			lui $t0, 32768	
R1	X24	X01095022	X24	X80000000	X1	X7ffffff	0	X1	X1	X1	X8	X80000000
6	nop	beq $zero, $zero, -1		sub $t2, $t0, $t1				subu $t2, $t0, $t1			addi $t1, $zero, 1	
R2	X28	X1000ffff	X28	X80000000	X1	X7ffffff	1	X7ffffff	X1	X7ffffff	X9	X1
7	beq $zero, $zero, -1	nop		beq $zero, $zero, -1				sub $t2, $t0, $t1			subu $t2, $t0, $t1	
R2	X8	X00000000	X0	X0	X0	X0	0	X0	X0	X0	XA	X7ffffff
8	nop	beq $zero, $zero, -1		nop				beq $zero, $zero, -1			sub $t2, $t0, $t1	
	XC	X1000ffff	XC	X0	X0	X0	0	X0	X0	X0	X0	X0

Lastly, to verify "ADDIU" and "ADDI" instructions exception handling mechanism, the test program given in Table 5.9 is downloaded to processor. A requirement number (as R#) is given in the comment section of the code and the clock cycle in which the requirement is fulfilled is pointed out in the first column in Table 5.10.

Table 5.9: Verification of Exception Handling "ADDIU" and "ADDI"

```
################################################
#
# TEST_5
#
# Created by Can Altınığneli
# To demonstrate ADDIU and ADDI instructions generate overflow exceptions according to APPENDIX A.
################################################
UNDEFINED:
beq $zero, $zero, UNDEFINED          # UNDEFINED EXCEPTION VECTOR
nop

OVERFLOW:
beq $zero, $zero, OVERFLOW           # OVERFLOW EXCEPTION VECTOR
nop

START:
add    $t0, $zero,$zero              # $t0 shall = 0
addiu  $t0, $t0, 0xFFFF              # $t0 shall = xFFFF_FFFF,  DestAdr:8
addiu  $t0, $t0, 1                   # No Exception shall be generated, R1
lui    $t0, 0x8000                   # $t0 shall = x8000_0000,  DestAdr:8
addi   $t0, $t0, -1                  # Exception shall be generated, R2

Eternity:
beq $zero, $zero, Eternity
nop
```

Results of operations and contents of stages are read by using MIPS Monitor software and results are tabulated in Table 5.10.

Table 5.10: Timing Diagram for Exception Handling of ADDIU and ADDI

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
1	addiu $t0, $t0, 65535	add $t0, $zero, $zero		-				-			-	
	X14	X00004020	X14	-	-	-	-	-	-	-	-	-
2	addiu $t0, $t0, 1	addiu $t0, $t0, 65535		add $t0, $zero, $zero				-			-	
	X18	X2508ffff	X18	X0	X0	X0	0	-	-	-	-	-
3	lui $t0, 32768	addiu $t0, $t0, 1		addiu $t0, $t0, 65535				add $t0, $zero, $zero			-	
	X1C	X25080001	X1C	X0	Xffffffff	Xffffffff	0	X0	X0	X0	-	-
4	addi $t0, $t0, -1	lui $t0, 32768		addiu $t0, $t0, 1				addiu $t0, $t0, 65535			add $t0, $zero, $zero	
R1	X20	X3C088000	X20	Xffffffff	X1	X0	0	Xffffffff	Xffffffff	Xffffffff	X8	X0
5	beq $zero, $zero, -1	addi $t0, $t0, -1		lui $t0, 32768				addiu $t0, $t0, 1			addiu $t0, $t0, 65535	
	X24	X2108ffff	X24	X0	X8000	X80000000	0	X0	X1	X0	X8	Xffffffff
6	nop	beq $zero, $zero, -1		addi $t0, $t0, -1				lui $t0, 32768			addiu $t0, $t0, 1	
	X28	X1000ffff	X28	X80000000	Xffffffff	X7fffffff	1	X80000000	X8000	X80000000	X8	X0
7	beq $zero, $zero, -1	nop		beq $zero, $zero, -1				addi $t0, $t0, -1			lui $t0, 32768	
	X8	X00000000	X0	X0	X0	X0	0	X0	X0	X0	X8	X80000000
8	nop	beq $zero, $zero, -1		nop				beq $zero, $zero, -1			addi $t0, $t0, -1	
	XC	X1000ffff	XC	X0	X0	X0	0	X0	X0	X0	X0	X0

To verify undefined instruction exception handling, the machine code of the program given in Table 5.9 is modified as given in Table 5.11, hence an undefined instruction is generated. Processor will raise an undefined exception while the modified instruction is in EX stage and this result can be observed by inspecting Table 5.12.

Table 5.11: Verification of Exception Handling Undefined Instructions

[0x000000]	0x1000FFFF	# beq $zero, $zero, -1
[0x000004]	0x00000000	# nop
[0x000008]	0x1000FFFF	# beq $zero, $zero, -1
[0x00000C]	0x00000000	# nop
[0x000010]	0x00004020	# add $t0, $zero, $zero
[0x000014]	0x2508FFFF	# addiu $t0, $t0, 65535
[0x000018]	~~0x25080001~~ → changed as 0xFF080001	# addiu $t0, $t0, 1
[0x00001C]	0x3C088000	# lui $t0, 32768
[0x000020]	0x2108FFFF	# addi $t0, $t0, -1
[0x000024]	0x1000FFFF	# beq $zero, $zero, -1
[0x000028]	0x00000000	# nop

Table 5.12: Timing Diagram for Undefined Instruction Exception Handling

CLK	IF STAGE	ID STAGE		EX STAGE				MEM STAGE			WB_STAGE	
(R#)	INSTR_IF	INSTR_ID		INSTR_EX				INSTR_MEM			INSTR_WB	
	Curr_PC	Instr	Incr_PC	ALU_A	ALU_B	Res	Ovr	Addr	Wr_D	Rd_D	Reg_D	Wr_D
1	addiu $t0, $t0, 65535	add $t0, $zero, $zero		-				-			-	
	X14	X00004020	X14	-	-	-	-	-	-	-	-	-
2	addiu $t0, $t0, 1	addiu $t0, $t0, 65535		add $t0, $zero, $zero								
	X18	X2508ffff	X18	X0	X0	X0	0	-	-	-	-	-
3	lui $t0, 32768	addiu $t0, $t0, 1		addiu $t0, $t0, 65535				add $t0, $zero, $zero			-	
	X1C	XFF080001	X1C	X0	Xffffffff	Xffffffff	0	X0	X0	X0	-	-
4	addi $t0, $t0, -1	lui $t0, 32768		addiu $t0, $t0, 1				addiu $t0, $t0, 65535			add $t0, $zero, $zero	
	X20	X3C088000	X20	X0	Xffffffff	X0	0	Xffffffff	Xffffffff	Xffffffff	X8	X0
5	beq $zero, $zero, -1	addi $t0, $t0, -1		lui $t0, 32768				addiu $t0, $t0, 1			addiu $t0, $t0, 65535	
	X0	X0	X0	X0	X0	X0	0	X0	X0	X0	X8	Xffffffff
6	nop	beq $zero, $zero, -1		addi $t0, $t0, -1				lui $t0, 32768			addiu $t0, $t0, 1	
	X4	X1000ffff	X4	X0	X0	X0	0	X0	X0	X0	X0	X0

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Pipelining, basic way of obtaining faster processor, was inspected in detail throughout this thesis and the basic principles were applied by implementing a pipelined processor on a real hardware (FPGA).

It was aimed to clarify why pipelining is preferred instead of other possible implementation schemes by comparing them quantitatively and after that it was concluded that the best performance can be obtained by applying Pipelined Implementation Scheme.

Different solution proposals were stated for problems faced while implementing pipelining. It became clearer that the main point causing problems was the dependencies between instructions. These dependencies degrades the instruction throughput and CPI can be greater than one which is optimal solution and this problem was resolved by constituting forwarding (bypass) lines between stages. Structural deficiencies are overcame by using separate Instruction and Data Memory. The Control (Branch) hazards caused by conditional or unconditional branches are overcame by making the decision in ID stage instead of EX in the expense of using extra hardware. It is tried to be explained how exceptions shall be handled in a pipelined architecture. After all of these statements and giving implementation details, architecture was verified with test programs and results were tabulated.

There exist unimplemented instructions in MIPS R2000 ISA, because the first goal of this thesis is to reveal the internals of pipelining and not to implement a complete processor. The most frequently instructions were chosen and implemented. A custom exception handing mechanism was implemented instead of implementing a complete co-processor for similar reasons.

There are many directions in which the work described in this thesis can be extended. There can be a research in the future which can propose a method to measure the orthogonality of ISA which is the primary metric for the effectiveness of pipelining. The processor can be extended to completely cover all instructions in MIPS R2000 ISA. Dynamic prediction mechanism can be used to branch decision instead of simple delayed branch approach. As a further step, processor can be upgraded by adding a floating point co-processor and virtual memory support to implement R3000 ISA. A more overwhelming work is to operate with 64 bit instructions and converge to R4000 ISA architecture which is commercially available today.

Another direction to extend this research is to inspect the effects of using longer pipelines, fetching longer instructions like in R4000 from memory and implementing sequencing and some handling mechanisms for all of these circumstances.

REFERENCES

[Barr99] Barr Michael, “Programmable Logic: What is it to Ya?” Embedded System Programming, pages 74-84, June 99

[Brown96] Stephen Brown, Jonathan Rose, “FPGA and CPLD Architectures: A Tutoral”, IEEE Design and Test for Computers, 1996

[BZEID] Bob Zeidman, Introduction to CPLD and FPGA Design

[CDVHDL] Volnei A. Pedroni, “Circuit Design with VHDL”, MIT Press 2004

[COD98] David A. Patterson and John L. Hennessy “Computer Organization and Design”, Chapters 3-6, 1998

[DFMULT] J. Senthil Kumar, G. Lakshminarayanan, B. Venkataramani, G. Siriram, M.S. Jambunathan, “Design and Implementation of FPGA based Fast Multipliers with Optimum Placement and Routing using Structure Organizer”

[Perry02] Douglas L. Perrry, “VHDL Programming by Example”, 2002

[HPCC] Scott Hauck, Mathew M. Hosler, Thomas W. Fry, “High Performance Carry Chains for FPGAs”

[JGRAY00] Jan Gray, “Building a RISC System on FPGA”, Circuit Cellar The Magazine for Computer Applications, March 2000

[KCHAP93] Ken Chapman, “Fast Integer Multipliers, Engineering Design Magazine’s Design Ideas Column, March 1993

[PLXSDK01] PLX SDK User’s Manual section 4, March 2001

[PLXSDK02] PLX PCI 9030 Data Book, v14, Page 2.7, 2002

[SYNP99] “Synthesis for 1 Milion Gate FPGAs: Synplicity Support for Xilinx Virtex Series”, Synplicity Inc. 1999

[TRENZ01] Trenz Electronic, “Introduction to FPGA Technology”, November 2001

[TW04] R.H. Turner, R.F. Woods, “High Efficient Limited Range Multipliers for LUT Based FPGA Architectures”, IEEE Transactions on very large scale integrated systems, vol 12, No:10, October 2004

[XAPP215] Xilinx Application Note, Design Tips for HDL Implementation of Arithmetic Functions, June 2000

[XCNSTR] Xilinx 5.xi Constraints, Understanding Timing and Placement Constraints

[XDRM99] Xilinx Design Reuse Methodology for ASIC and FPGA Designers, System on Chip Design reuse Solutions, An Addendum to Reuse Methodology Manual for SoC Design, pages 1-27, October 99

[XDS003-2] Xilinx Data Sheet for Virtex™ 2.5V FPGA, pages 5-24, December 2002

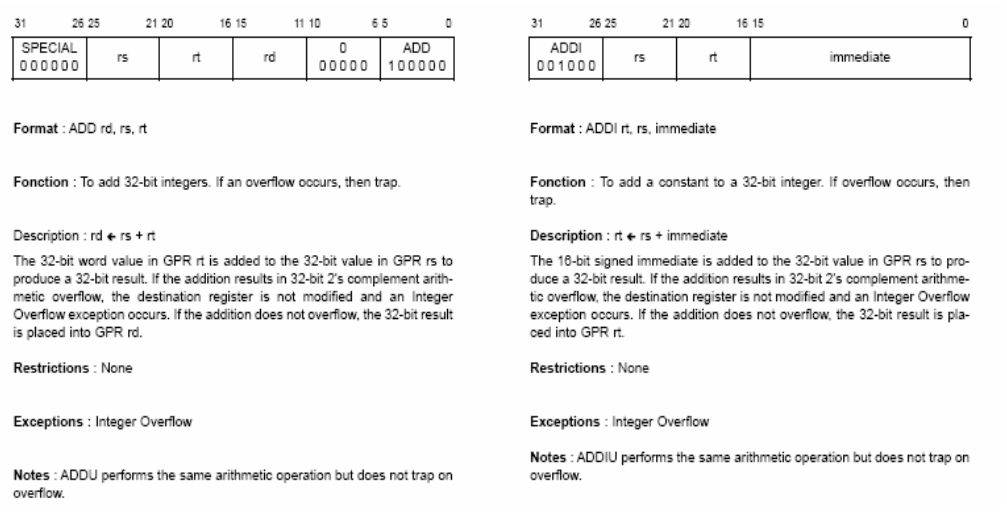
[XISE03] Xilinx ISE Quick Start Tutorial, pages 12-17, June 2003

[XLBR04] Xilinx Libraries Guide, V6.3i, pages 321-323

[XPM04] Karen Parnell, Nick Mehta, Xilinx Programmable Logic Design Quick Start Hand Book, pages 1-20, April 2004

APPENDIX A

IMPLEMENTED SUBSET OF MIPS R2000 ISA



31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000

Format : ADD rd, rs, rt

Fonction : To add 32-bit integers. If an overflow occurs, then trap.

Description : rd ← rs + rt

The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs. If the addition does not overflow, the 32-bit result is placed into GPR rd.

Restrictions : None

Exceptions : Integer Overflow

Notes : ADDU performs the same arithmetic operation but does not trap on overflow.

31 26	25 21	20 16	15 0
ADDI 001000	rs	rt	immediate

Format : ADDI rt, rs, immediate

Fonction : To add a constant to a 32-bit integer. If overflow occurs, then trap.

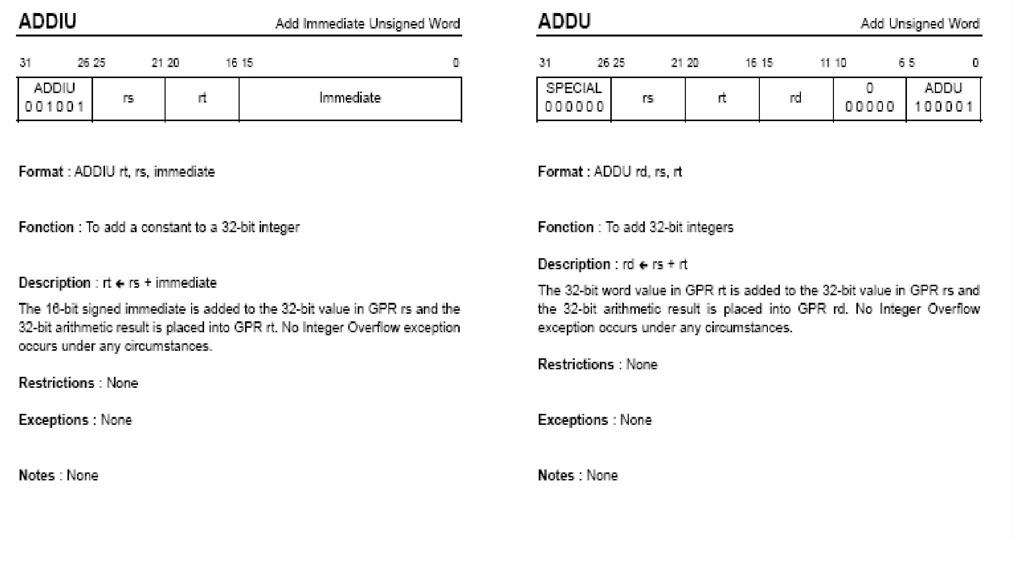
Description : rt ← rs + immediate

The 16-bit signed immediate is added to the 32-bit value in GPR rs to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs. If the addition does not overflow, the 32-bit result is placed into GPR rt.

Restrictions : None

Exceptions : Integer Overflow

Notes : ADDIU performs the same arithmetic operation but does not trap on overflow.



ADDIU Add Immediate Unsigned Word

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	Immediate

Format : ADDIU rt, rs, immediate

Fonction : To add a constant to a 32-bit integer

Description : rt ← rs + immediate

The 16-bit signed immediate is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rt. No Integer Overflow exception occurs under any circumstances.

Restrictions : None

Exceptions : None

Notes : None

ADDU Add Unsigned Word

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001

Format : ADDU rd, rs, rt

Fonction : To add 32-bit integers

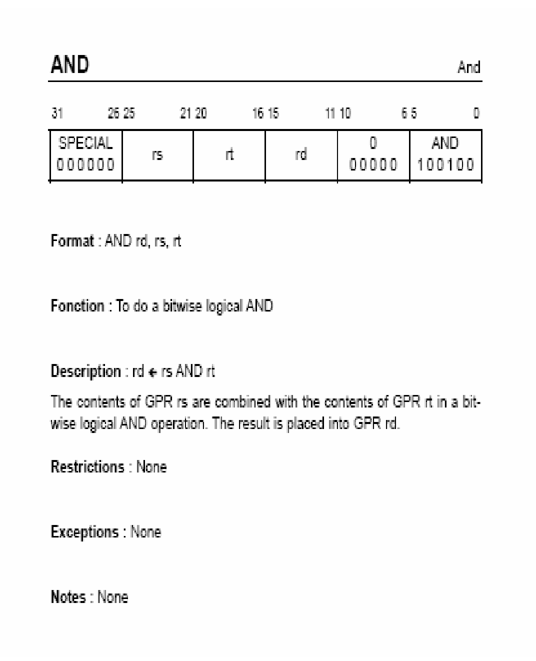
Description : rd ← rs + rt

The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rd. No Integer Overflow exception occurs under any circumstances.

Restrictions : None

Exceptions : None

Notes : None



AND — And

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100

Format : AND rd, rs, rt

Fonction : To do a bitwise logical AND

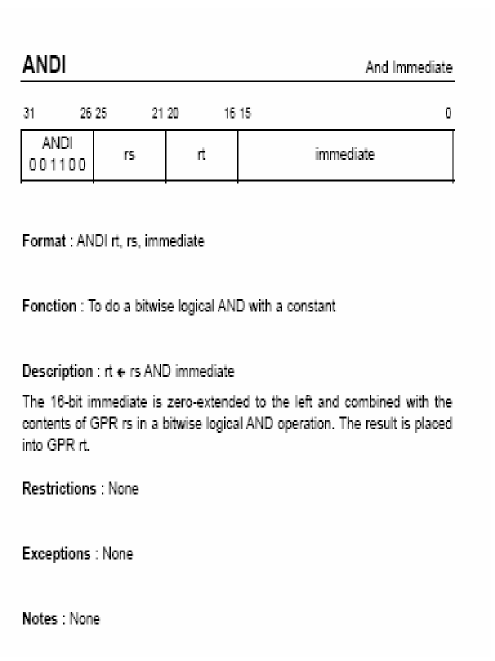
Description : rd ← rs AND rt

The contents of GPR rs are combined with the contents of GPR rt in a bit-wise logical AND operation. The result is placed into GPR rd.

Restrictions : None

Exceptions : None

Notes : None



ANDI — And Immediate

31 26	25 21	20 16	15 0
ANDI 001100	rs	rt	immediate

Format : ANDI rt, rs, immediate

Fonction : To do a bitwise logical AND with a constant

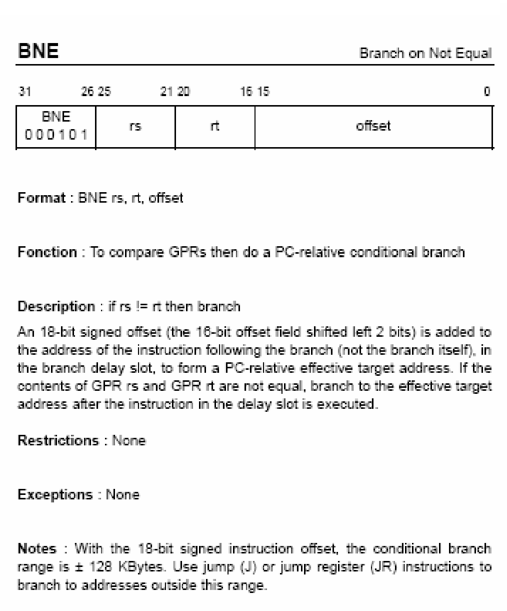
Description : rt ← rs AND immediate

The 16-bit immediate is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical AND operation. The result is placed into GPR rt.

Restrictions : None

Exceptions : None

Notes : None



BNE — Branch on Not Equal

31 26	25 21	20 16	15 0
BNE 000101	rs	rt	offset

Format : BNE rs, rt, offset

Fonction : To compare GPRs then do a PC-relative conditional branch

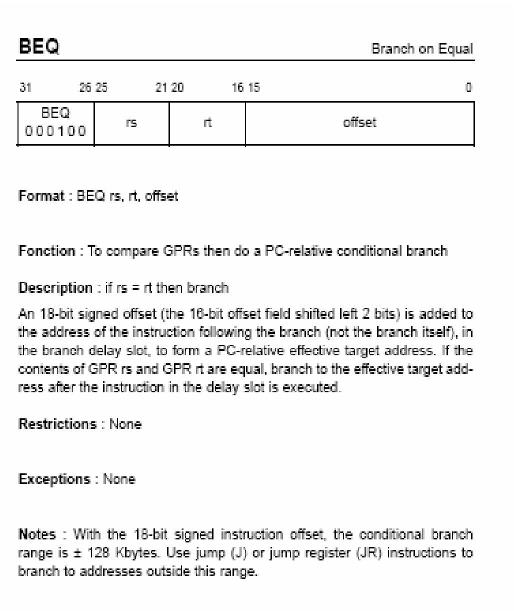
Description : if rs != rt then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address. If the contents of GPR rs and GPR rt are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions : None

Exceptions : None

Notes : With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.



BEQ — Branch on Equal

31 26	25 21	20 16	15 0
BEQ 000100	rs	rt	offset

Format : BEQ rs, rt, offset

Fonction : To compare GPRs then do a PC-relative conditional branch

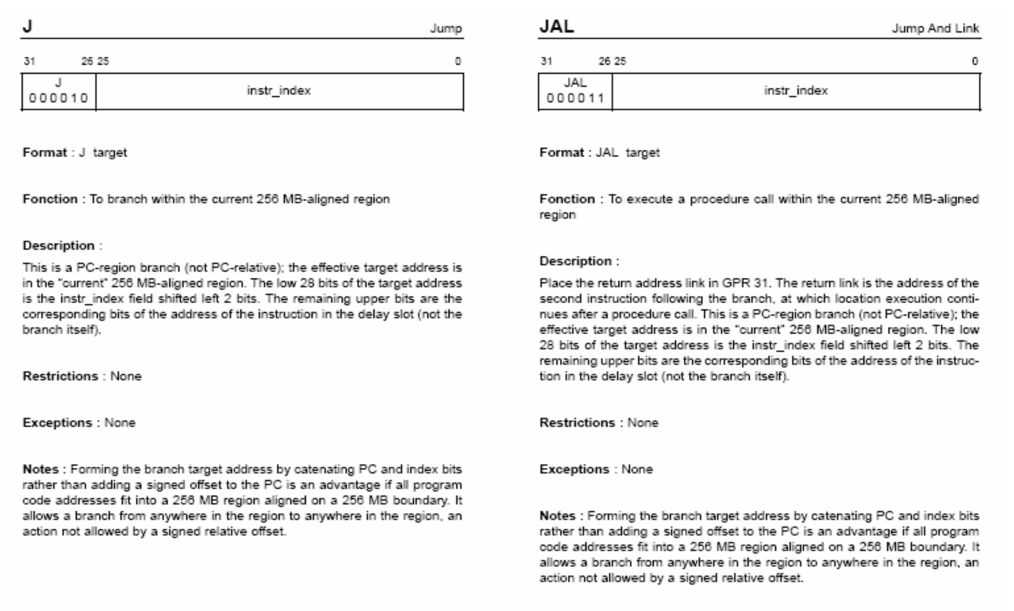
Description : if rs = rt then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address. If the contents of GPR rs and GPR rt are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions : None

Exceptions : None

Notes : With the 18-bit signed instruction offset, the conditional branch range is ± 128 Kbytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.



J Jump

31 26	25 0
J 000010	instr_index

Format : J target

Fonction : To branch within the current 256 MB-aligned region

Description :

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB-aligned region. The low 28 bits of the target address is the instr_index field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Restrictions : None

Exceptions : None

Notes : Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

JAL Jump And Link

31 26	25 0
JAL 000011	instr_index

Format : JAL target

Fonction : To execute a procedure call within the current 256 MB-aligned region

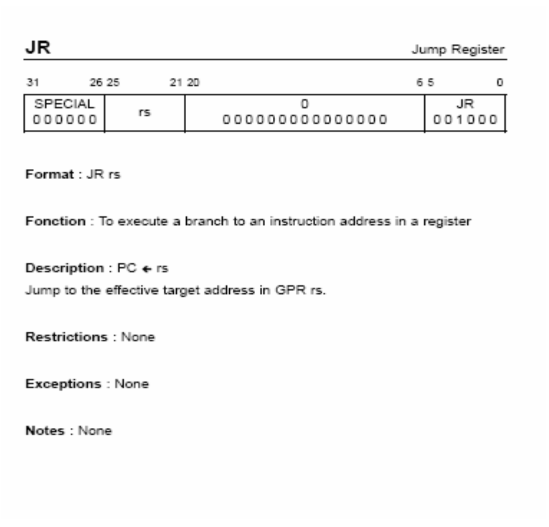
Description :

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call. This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB-aligned region. The low 28 bits of the target address is the instr_index field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Restrictions : None

Exceptions : None

Notes : Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.



JR Jump Register

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000000000000000	JR 001000

Format : JR rs

Fonction : To execute a branch to an instruction address in a register

Description : PC ← rs

Jump to the effective target address in GPR rs.

Restrictions : None

Exceptions : None

Notes : None

LUI — Load Upper Immediate

31 26	25 21	20 16	15 0
LUI 001111	0 00000	rt	immediate

Format : LUI rt, immediate

Fonction : To load a constant into the upper half of a word

Description : rt ← immediate || 0^{16}

The 16-bit immediate is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is placed into GPR rt.

Restrictions : None

Exceptions : None

Notes : None

LW — Load Word

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset

Format : LW rt, offset(base)

Fonction : To load a word from memory as a signed value

Description : rt ← memory[base+offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions : None

Exceptions : None

Notes : None

MFHI — Move From HI register

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 0000000000	rd	0 00000	MFHI 010000

Format : MFHI rd

Fonction : To copy the special purpose HI register to a GPR

Description : rd ← HI

The contents of special register HI are loaded into GPR rd.

Restrictions : None

Exceptions : None

Notes : None

MFLO — Move From LO register

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 0000000000	rd	0 00000	MFLO 010010

Format : MFLO rd

Fonction : To copy the special purpose LO register to a GPR

Description : rd ← LO

The contents of special register LO are loaded into GPR rd.

Restrictions : None

Exceptions : None

Notes : None

MTHI
Move To HI register

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000000000000000	MTHI 010001

Format : MTHI rs

Fonction : To copy a GPR to the special purpose HI register

Description : HI ← rs
The contents of GPR rs are loaded into special register HI.

Restrictions : None

Exceptions : None

Notes : None

MTLO
Move To LO register

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000000000000000	MTLO 010011

Format : MTLO rs

Fonction : To copy a GPR to the special purpose LO register

Description : HI ← rs
The contents of GPR rs are loaded into special register LO.

Restrictions : None

Exceptions : None

Notes : None

MULTU
Multiply Unsigned Word

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 0000000000	MULTU 011001

Format : MULTU rs, rt

Fonction : To multiply 32-bit unsigned integers

Description : (LO, HI) ← rs x rt
The 32-bit word value in GPR rt is multiplied by the 32-bit value in GPR rs, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register LO, and the high-order 32-bit word is placed into special register HI. No arithmetic exception occurs under any circumstances.

Restrictions : None

Exceptions : None

Notes : None

NOR
Not Or

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111

Format : NOR rd, rs, rt

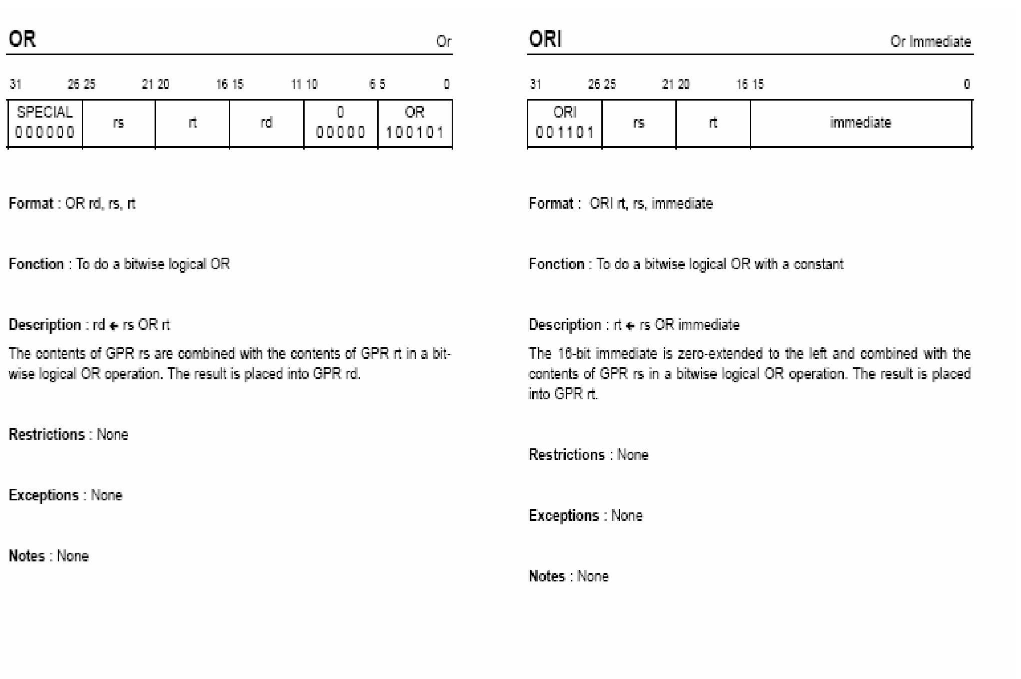
Fonction : To do a bitwise logical NOT OR

Description : rd ← rs NOR rt
The contents of GPR rs are combined with the contents of GPR rt in a bit-wise logical NOR operation. The result is placed into GPR rd.

Restrictions : None

Exceptions : None

Notes : None



OR Or

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101

Format : OR rd, rs, rt

Fonction : To do a bitwise logical OR

Description : rd ← rs OR rt

The contents of GPR rs are combined with the contents of GPR rt in a bit-wise logical OR operation. The result is placed into GPR rd.

Restrictions : None

Exceptions : None

Notes : None

ORI Or Immediate

31 26	25 21	20 16	15 0
ORI 001101	rs	rt	immediate

Format : ORI rt, rs, immediate

Fonction : To do a bitwise logical OR with a constant

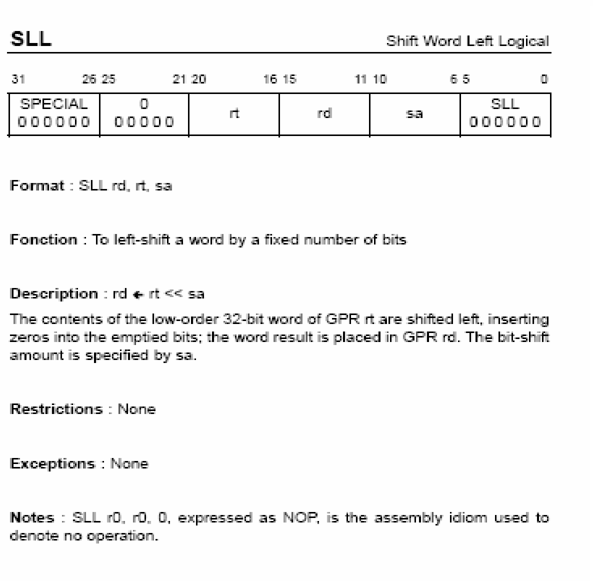
Description : rt ← rs OR immediate

The 16-bit immediate is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical OR operation. The result is placed into GPR rt.

Restrictions : None

Exceptions : None

Notes : None



SLL Shift Word Left Logical

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SLL 000000

Format : SLL rd, rt, sa

Fonction : To left-shift a word by a fixed number of bits

Description : rd ← rt << sa

The contents of the low-order 32-bit word of GPR rt are shifted left, inserting zeros into the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by sa.

Restrictions : None

Exceptions : None

Notes : SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLT
Set On Less Than

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010

Format : SLT rd, rs, rt

Fonction : To record the result of a less-than comparison

Description : rd ← (rs < rt)

Compare the contents of GPR rs and GPR rt as signed integers and record the Boolean result of the comparison in GPR rd. If GPR rs is less than GPR rt, the result is 1 (true); otherwise, it is 0 (false). The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions : None

Exceptions : None

Notes : None

SLTI
Set on Less Than Immediate

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate

Format : SLTI rt, rs, immediate

Fonction : To record the result of a less-than comparison with a constant

Description : rt ← (rs < immediate)

Compare the contents of GPR rs and the 16-bit signed immediate as signed integers and record the Boolean result of the comparison in GPR rt. If GPR rs is less than immediate, the result is 1 (true); otherwise, it is 0 (false). The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions : None

Exceptions : None

Notes : None

SLTIU
Set on Less Than Immediate Unsigned

31 26	25 21	20 16	15 0
SLTIU 001011	rs	rt	immediate

Format : SLTIU rt, rs, immediate

Fonction : To record the result of an unsigned less-than comparison with a constant

Description : rt ← (rs < immediate)

Compare the contents of GPR rs and the sign-extended 16-bit immediate as unsigned integers and record the Boolean result of the comparison in GPR rt. If GPR rs is less than immediate, the result is 1 (true); otherwise, it is 0 (false). The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions : None

Exceptions : None

Notes : None

SLTU
Set on Less Than Unsigned

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLTU 101011

Format : SLTU rd, rs, rt

Fonction : To record the result of an unsigned less-than comparison

Description : rd ← (rs < rt)

Compare the contents of GPR rs and GPR rt as unsigned integers and record the Boolean result of the comparison in GPR rd. If GPR rs is less than GPR rt, the result is 1 (true); otherwise, it is 0 (false). The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions : None

Exceptions : None

Notes : None

SRL Shift Word Right Logical

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRL 000010

Format : SRL rd, rt, sa

Fonction : To execute a logical right-shift of a word by a fixed number of bits

Description : rd ← rt >> sa

The contents of the low-order 32-bit word of GPR rt are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR rd. The bit-shift amount is specified by sa.

Restrictions : None

Exceptions : None

Notes : None

SUB Substract Word

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010

Format : SUB rd, rs, rt

Fonction : To subtract 32-bit integers. If overflow occurs, then trap

Description : rd ← rs - rt

The 32-bit word value in GPR rt is subtracted from the 32-bit value in GPR rs to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR rd.

Restrictions : None

Exceptions : Integer Overflow

Notes : SUBU performs the same arithmetic operation but does not trap on overflow.

SUBU Substract Unsigned Word

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011

Format : SUBU rd, rs, rt

Fonction : To subtract unsigned 32-bit integers

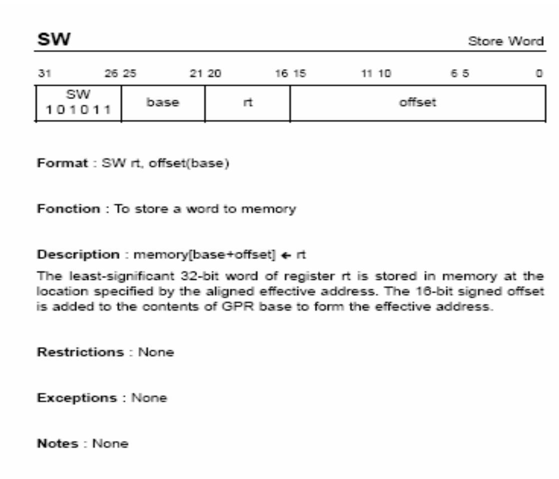
Description : rd ← rs - rt

The 32-bit word value in GPR rt is subtracted from the 32-bit value in GPR rs and the 32-bit arithmetic result is and placed into GPR rd. No integer overflow exception occurs under any circumstances.

Restrictions : None

Exceptions : None

Notes : None



SW Store Word

31–26	25–21	20–16	15–0
SW 101011	base	rt	offset

Format : SW rt, offset(base)

Fonction : To store a word to memory

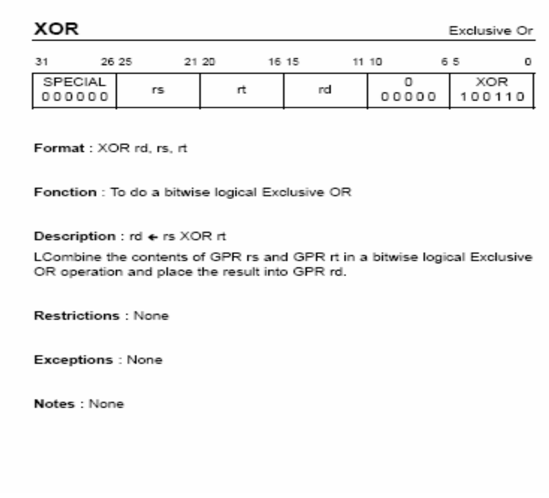
Description : memory[base+offset] ← rt

The least-significant 32-bit word of register rt is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address.

Restrictions : None

Exceptions : None

Notes : None



XOR Exclusive Or

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110

Format : XOR rd, rs, rt

Fonction : To do a bitwise logical Exclusive OR

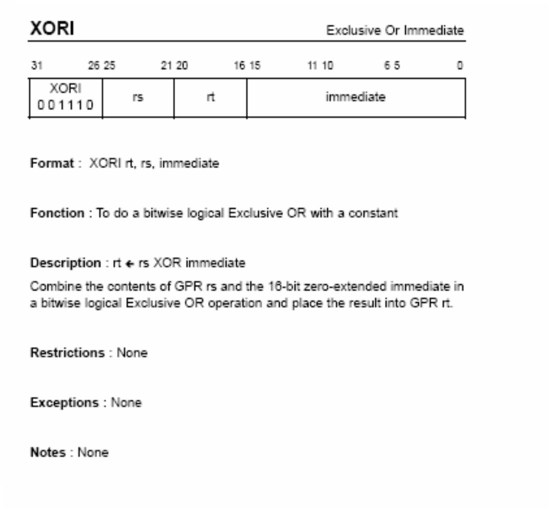
Description : rd ← rs XOR rt

LCombine the contents of GPR rs and GPR rt in a bitwise logical Exclusive OR operation and place the result into GPR rd.

Restrictions : None

Exceptions : None

Notes : None



XORI Exclusive Or Immediate

31–26	25–21	20–16	15–0
XORI 001110	rs	rt	immediate

Format : XORI rt, rs, immediate

Fonction : To do a bitwise logical Exclusive OR with a constant

Description : rt ← rs XOR immediate

Combine the contents of GPR rs and the 16-bit zero-extended immediate in a bitwise logical Exclusive OR operation and place the result into GPR rt.

Restrictions : None

Exceptions : None

Notes : None

Table A.1: MIPS Registers

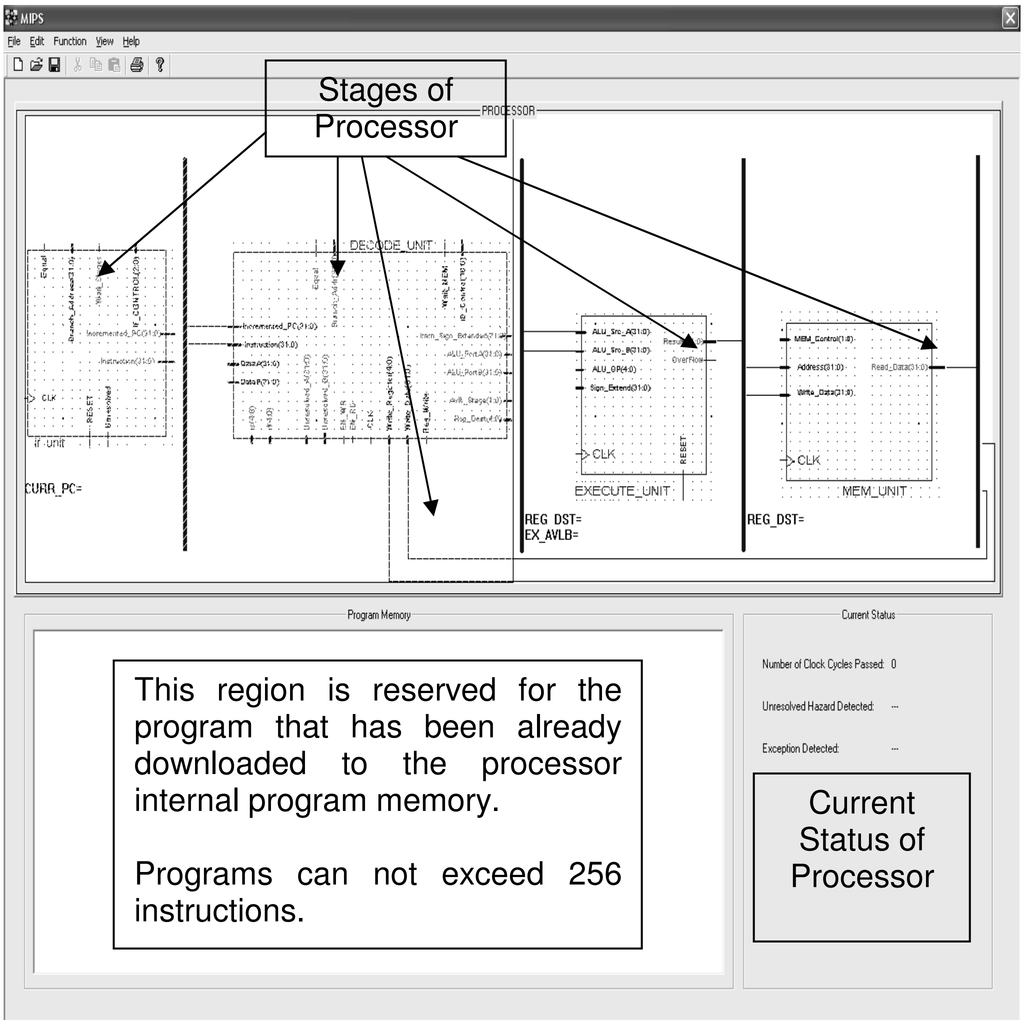
Name	Register number	Usage
$zero	0	the constant value 0
$at	1	reserved for the assembler
$v0–$v1	2–3	values for results and expression evaluation
$a0–$a3	4–7	arguments
$t0–$t7	8–15	temporaries
$s0–$s7	16–23	saved
$t8–$t9	24–25	more temporaries
$k0–$k1	26–27	reserved for the operating system (OS)
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

APPENDIX B

MIPS MONITOR SOFTWARE

MIPS Monitor Software is written to monitor internal state of the processor, to externally stimulate the processor and to verify correctness of its operation. MIPS Monitor Software is written in C++ and developed in Microsoft™ Visual C++ Environment. Document-View architecture is used during is development. This Appendix is prepared to serve as a user manual of MIPS Monitor Software.

The main screen of MIPS Monitor software is given below:



This region is reserved for the program that has been already downloaded to the processor internal program memory.

Programs can not exceed 256 instructions.

Number of Clock Cycles Passed: 0

Unresolved Hazard Detected: ---

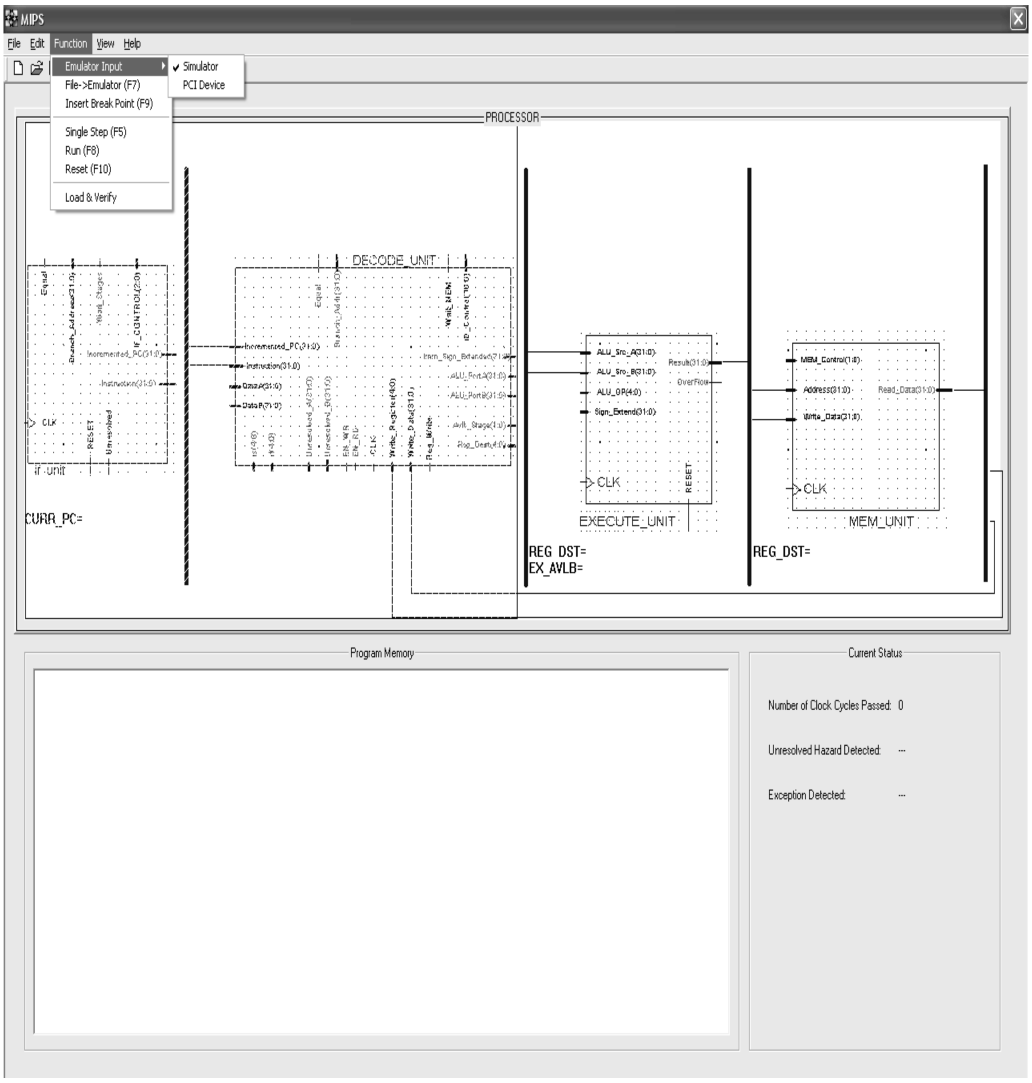
Exception Detected: ---

Figure B.1: Main Screen of MIPS Monitor Software

The main functions of MIPS Monitor software is collected under Function menu. These functions can be summarized as:

Emulator Input: This option is used to run with real hardware or to test the graphical interface with simulator without hardware. This interface was used during development while the hardware was not present and “Simulator” option was disabled after development. “PCIDevice” option must be chosen before starting the MIPS Monitor software for proper operation. After that, the “PCI Device Selection Dialog” (Figure B.3) will appear and user can select the bridge on which interface transactions will occur.

<u>File→Emulator (F7):</u> A "File Open Dialog" will appear after selecting this option. The selected program will be loaded Program Memory section of main screen, but this program is not downloaded to processor.

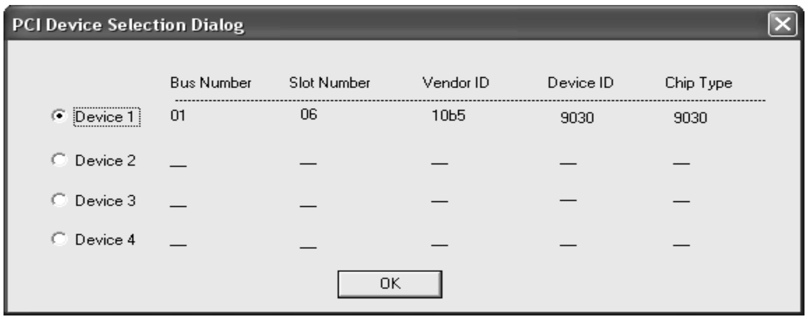


Number of Clock Cycles Passed: 0

Unresolved Hazard Detected: ---

Exception Detected: ---

Figure B.2: Main Functions of MIPS Monitor Software


PCI Device Selection Dialog

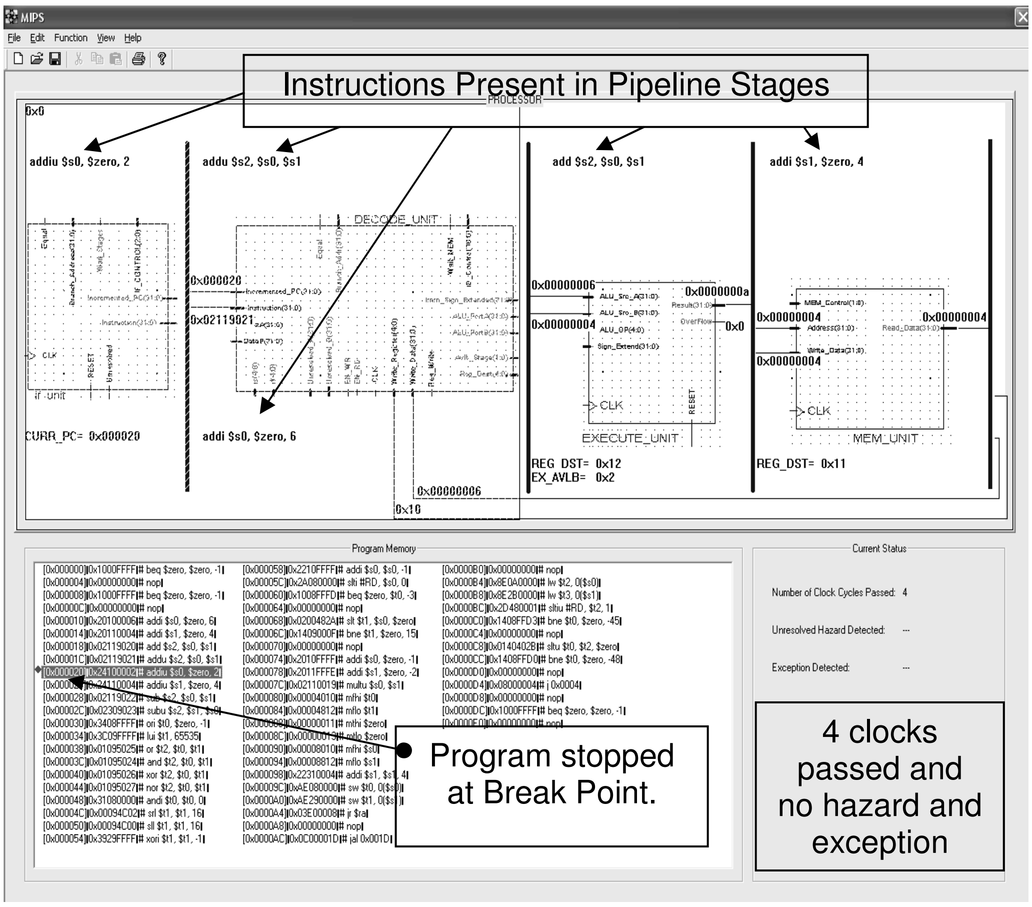
	Bus Number	Slot Number	Vendor ID	Device ID	Chip Type
(●) Device 1	01	06	10b5	9030	9030
( ) Device 2	—	—	—	—	—
( ) Device 3	—	—	—	—	—
( ) Device 4	—	—	—	—	—

OK

Figure B.3: PCI Device Selection Dialog

Insert Break Point (F9): This option enables the user to insert break points to stop the processor at a desired point while running or before Run (F8) option is selected. A red diamond will appear to indicate the point where the processor will stop its operation.

Single Step (F5): This option enables the user to trigger the processor for single step running. It is stated in Table 4.5 which fields of the IF, ID, EX, MEM and WB stages can be observed by using the MIPS Monitor software.



```
[0x000000]|0x1000FFFF|# beq $zero, $zero, -1|
[0x000004]|0x00000000|# nop|
[0x000008]|0x1000FFFF|# beq $zero, $zero, -1|
[0x00000C]|0x00000000|# nop|
[0x000010]|0x20100006|# addi $s0, $zero, 6|
[0x000014]|0x20110004|# addi $s1, $zero, 4|
[0x000018]|0x02119020|# add $s2, $s0, $s1|
[0x00001C]|0x02119021|# addu $s2, $s0, $s1|
[0x000020]|0x24100002|# addiu $s0, $zero, 2|
[0x000024]|0x24110004|# addiu $s1, $zero, 4|
[0x000028]|0x02119022|# sub $s2, $s0, $s1|
[0x00002C]|0x02309023|# subu $s2, $s1, $s0|
[0x000030]|0x3408FFFF|# ori $t0, $zero, -1|
[0x000034]|0x3C09FFFF|# lui $t1, 65535|
[0x000038]|0x01095025|# or $t2, $t0, $t1|
[0x00003C]|0x01095024|# and $t2, $t0, $t1|
[0x000040]|0x01095026|# xor $t2, $t0, $t1|
[0x000044]|0x01095027|# nor $t2, $t0, $t1|
[0x000048]|0x31080000|# andi $t0, $t0, 0|
[0x00004C]|0x00094C02|# srl $t1, $t1, 16|
[0x000050]|0x00094C00|# sll $t1, $t1, 16|
[0x000054]|0x3929FFFF|# xori $t1, $t1, -1|

[0x000058]|0x2210FFFF|# addi $s0, $s0, -1|
[0x00005C]|0x2A080000|# slti #RD, $s0, 0|
[0x000060]|0x1008FFFD|# beq $zero, $t0, -3|
[0x000064]|0x00000000|# nop|
[0x000068]|0x0200482A|# slt $t1, $s0, $zero|
[0x00006C]|0x1409000F|# bne $t1, $zero, 15|
[0x000070]|0x00000000|# nop|
[0x000074]|0x2010FFFF|# addi $s0, $zero, -1|
[0x000078]|0x2011FFFE|# addi $s1, $zero, -2|
[0x00007C]|0x02110019|# multu $s0, $s1|
[0x000080]|0x00004010|# mfhi $t0|
[0x000084]|0x00004812|# mflo $t1|
[0x000088]|0x00000011|# mthi $zero|
[0x00008C]|0x00000013|# mtlo $zero|
[0x000090]|0x00008010|# mfhi $s0|
[0x000094]|0x00008812|# mflo $s1|
[0x000098]|0x22310004|# addi $s1, $s1, 4|
[0x00009C]|0xAE080000|# sw $t0, 0($s0)|
[0x0000A0]|0xAE290000|# sw $t1, 0($s1)|
[0x0000A4]|0x03E00008|# jr $ra|
[0x0000A8]|0x00000000|# nop|
[0x0000AC]|0x0C00001D|# jal 0x001D|

[0x0000B0]|0x00000000|# nop|
[0x0000B4]|0x8E0A0000|# lw $t2, 0($s0)|
[0x0000B8]|0x8E2B0000|# lw $t3, 0($s1)|
[0x0000BC]|0x2D480001|# sltiu #RD, $t2, 1|
[0x0000C0]|0x1408FFD3|# bne $t0, $zero, -45|
[0x0000C4]|0x00000000|# nop|
[0x0000C8]|0x0140402B|# sltu $t0, $t2, $zero|
[0x0000CC]|0x1408FFD0|# bne $t0, $zero, -48|
[0x0000D0]|0x00000000|# nop|
[0x0000D4]|0x08000004|# j 0x0004|
[0x0000D8]|0x00000000|# nop|
[0x0000DC]|0x1000FFFF|# beq $zero, $zero, -1|
[0x0000E0]|0x00000000|# nop|
```

```
Number of Clock Cycles Passed:  4
Unresolved Hazard Detected:     ---
Exception Detected:             ---
```

Figure B.4: PCI Device Selection Dialog

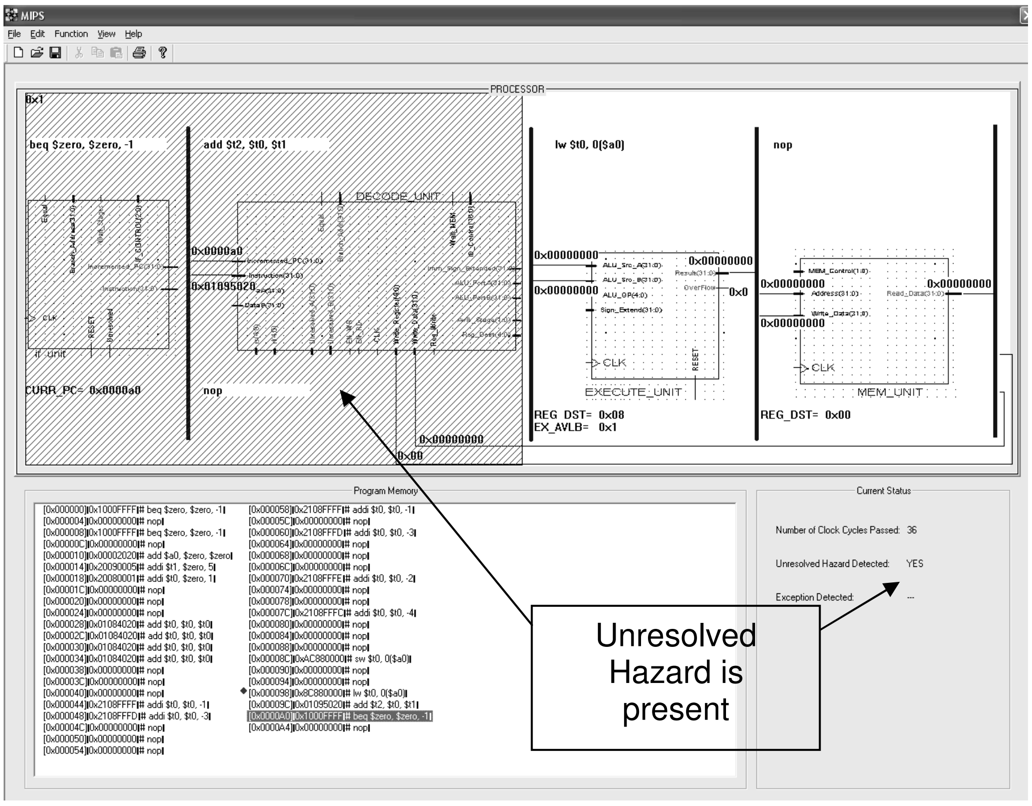
<u>Run (F8):</u> This option when selected runs the processor up to a Break Point is encountered.

<u>Reset (F10):</u> This option when selected resets the processor externally.

<u>Load & Verify:</u> A “File Open Dialog” will appear after selecting this option. The selected program will be loaded Program Memory section of main screen and also this program is downloaded to processor.

MIPS Monitor Software can notify the programmer about the presence of unresolved hazard in the pipeline by drawing a dashed box around the IF and ID stages and stating the status in “Current Status” section of

Main Screen. Programmer can expect a nop instruction insertion into EX stage in the next clock cycle (Figure B.5).



Program Memory

```
[0x000000]|0x1000FFFF|# beq $zero, $zero, -1|      [0x000058]|0x2108FFFF|# addi $t0, $t0, -1|
[0x000004]|0x00000000|# nop|                        [0x00005C]|0x00000000|# nop|
[0x000008]|0x1000FFFF|# beq $zero, $zero, -1|      [0x000060]|0x2108FFFD|# addi $t0, $t0, -3|
[0x00000C]|0x00000000|# nop|                        [0x000064]|0x00000000|# nop|
[0x000010]|0x00002020|# add $a0, $zero, $zero|     [0x000068]|0x00000000|# nop|
[0x000014]|0x20090005|# addi $t1, $zero, 5|        [0x00006C]|0x00000000|# nop|
[0x000018]|0x20080001|# addi $t0, $zero, 1|        [0x000070]|0x2108FFFE|# addi $t0, $t0, -2|
[0x00001C]|0x00000000|# nop|                        [0x000074]|0x00000000|# nop|
[0x000020]|0x00000000|# nop|                        [0x000078]|0x00000000|# nop|
[0x000024]|0x00000000|# nop|                        [0x00007C]|0x2108FFFC|# addi $t0, $t0, -4|
[0x000028]|0x01084020|# add $t0, $t0, $t0|         [0x000080]|0x00000000|# nop|
[0x00002C]|0x01084020|# add $t0, $t0, $t0|         [0x000084]|0x00000000|# nop|
[0x000030]|0x01084020|# add $t0, $t0, $t0|         [0x000088]|0x00000000|# nop|
[0x000034]|0x01084020|# add $t0, $t0, $t0|         [0x00008C]|0xAC880000|# sw $t0, 0($a0)|
[0x000038]|0x00000000|# nop|                        [0x000090]|0x00000000|# nop|
[0x00003C]|0x00000000|# nop|                        [0x000094]|0x00000000|# nop|
[0x000040]|0x00000000|# nop|                      ◆ [0x000098]|0x8C880000|# lw $t0, 0($a0)|
[0x000044]|0x2108FFFF|# addi $t0, $t0, -1|         [0x00009C]|0x01095020|# add $t2, $t0, $t1|
[0x000048]|0x2108FFFD|# addi $t0, $t0, -3|         [0x0000A0]|0x1000FFFF|# beq $zero, $zero, -1|
[0x00004C]|0x00000000|# nop|                        [0x0000A4]|0x00000000|# nop|
[0x000050]|0x00000000|# nop|
[0x000054]|0x00000000|# nop|
```

Current Status

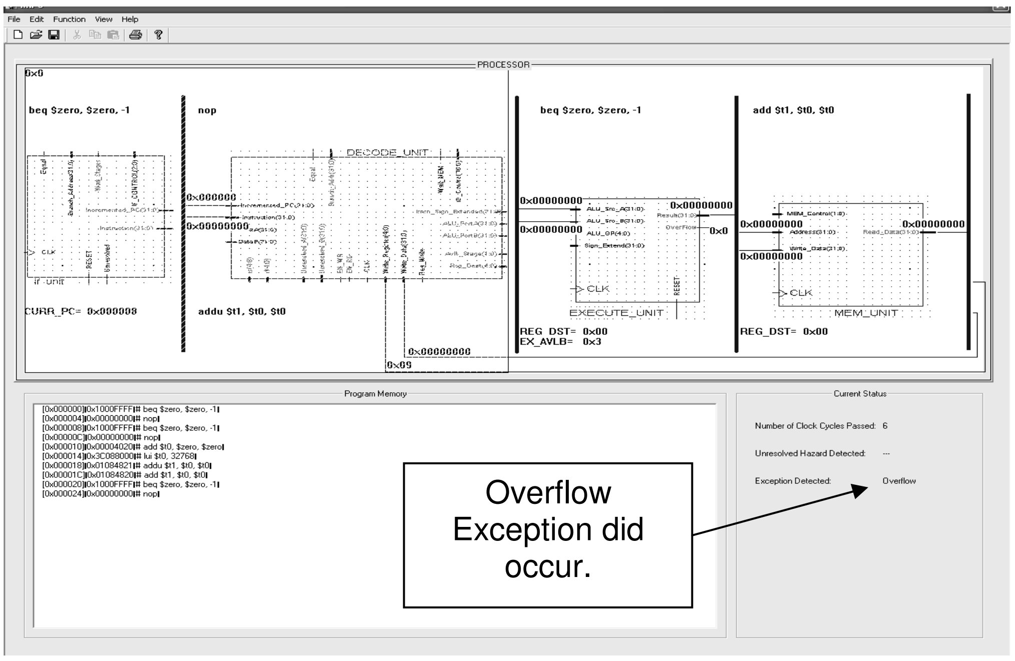
Number of Clock Cycles Passed: 36

Unresolved Hazard Detected: YES

Exception Detected: ---

Figure B.5: Unresolved Hazards View

MIPS Monitor software also has the ability to inform the programmer about the presence and sort of the exception in the pipeline. This information is presented in “Current Status” section of Main Screen.
The Overflow Exception is detected and reflected to Programmer as in Figure B.6.

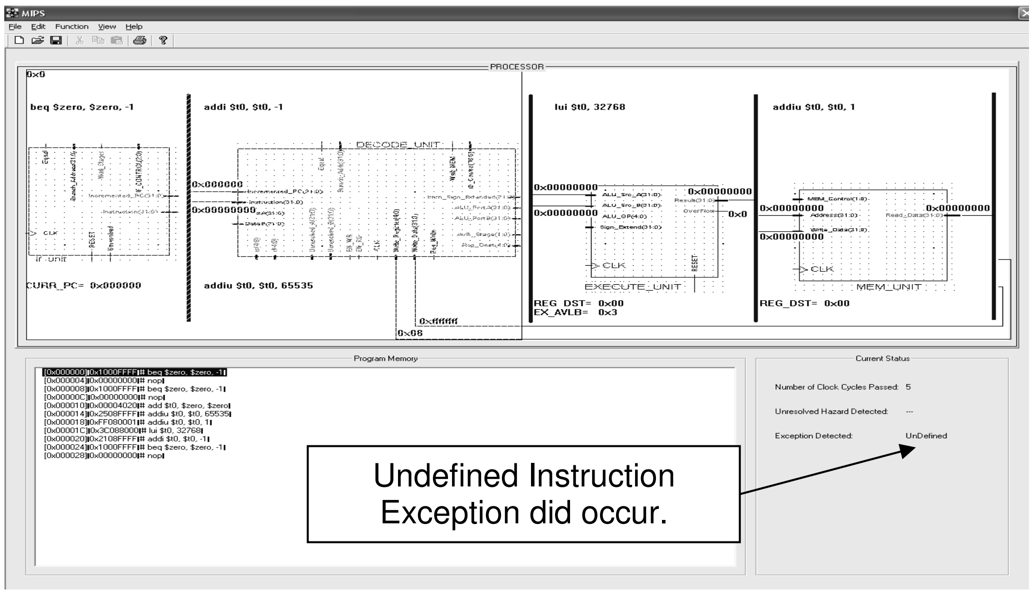


```
[0x000000]|0x1000FFFF|# beq $zero, $zero, -1|
[0x000004]|0x00000000|# nop|
[0x000008]|0x1000FFFF|# beq $zero, $zero, -1|
[0x00000C]|0x00000000|# nop|
[0x000010]|0x00004020|# add $t0, $zero, $zero|
[0x000014]|0x3C088000|# lui $t0, 32768|
[0x000018]|0x01084821|# addu $t1, $t0, $t0|
[0x00001C]|0x01084820|# add $t1, $t0, $t0|
[0x000020]|0x1000FFFF|# beq $zero, $zero, -1|
[0x000024]|0x00000000|# nop|
```

Number of Clock Cycles Passed: 6
Unresolved Hazard Detected: ---
Exception Detected: Overflow

Figure B.6: Overflow Exception Detection View

The Undefined Instruction is detected and reflected to Programmer as in Figure B.7.



```
[0x000000]|0x1000FFFF|# beq $zero, $zero, -1|
[0x000004]|0x00000000|# nop|
[0x000008]|0x1000FFFF|# beq $zero, $zero, -1|
[0x00000C]|0x00000000|# nop|
[0x000010]|0x00004020|# add $t0, $zero, $zero|
[0x000014]|0x2508FFFF|# addiu $t0, $t0, 65535|
[0x000018]|0xFF080001|# addiu $t0, $t0, 1|
[0x00001C]|0x3C088000|# lui $t0, 32768|
[0x000020]|0x2108FFFF|# addi $t0, $t0, -1|
[0x000024]|0x1000FFFF|# beq $zero, $zero, -1|
[0x000028]|0x00000000|# nop|
```

Number of Clock Cycles Passed: 5
Unresolved Hazard Detected: ---
Exception Detected: UnDefined

Figure B.7: Undefined Instruction Exception Detection View

APPENDIX C

FLOW DIAGRAMS ARCHITECTURE ELEMENTS

Instruction Fetch Unit Flow Diagram

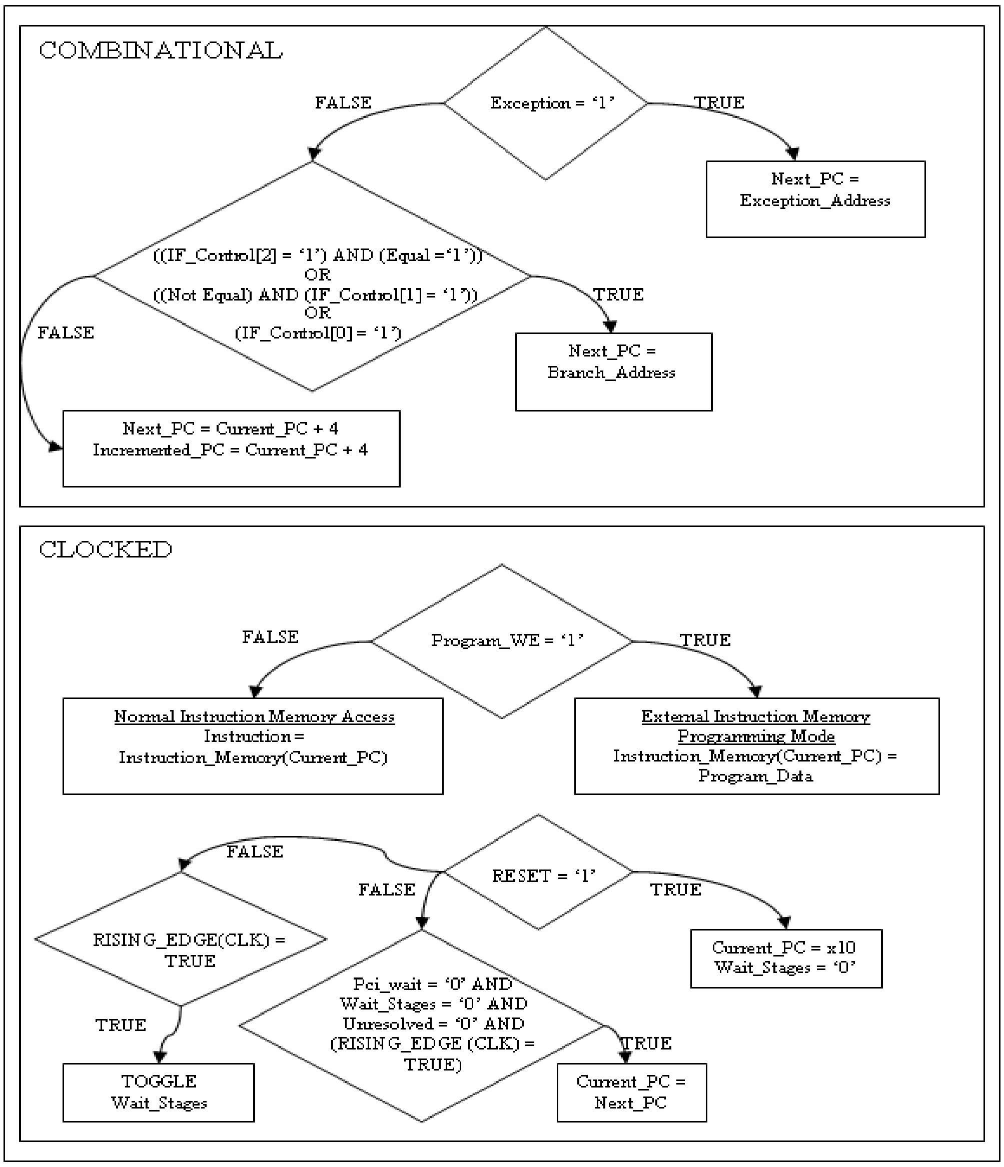

COMBINATIONAL
Exception = '1'
FALSE
TRUE
Next_PC = Exception_Address
((IF_Control[2] = '1') AND (Equal ='1')) OR ((Not Equal) AND (IF_Control[1] = '1')) OR (IF_Control[0] = '1')
FALSE
TRUE
Next_PC = Branch_Address
Next_PC = Current_PC + 4
Incremented_PC = Current_PC + 4
CLOCKED
Program_WE = '1'
FALSE
TRUE
Normal Instruction Memory Access
Instruction = Instruction_Memory(Current_PC)
External Instruction Memory Programming Mode
Instruction_Memory(Current_PC) = Program_Data
RESET = '1'
FALSE
FALSE
TRUE
Current_PC = x10
Wait_Stages = '0'
RISING_EDGE(CLK) = TRUE
TRUE
TOGGLE Wait_Stages
Pci_wait = '0' AND Wait_Stages = '0' AND Unresolved = '0' AND (RISING_EDGE (CLK) = TRUE)
TRUE
Current_PC = Next_PC

Figure C.1: Instruction Fetch Unit Flow Diagram

Instruction Decode Unit Flow Diagram

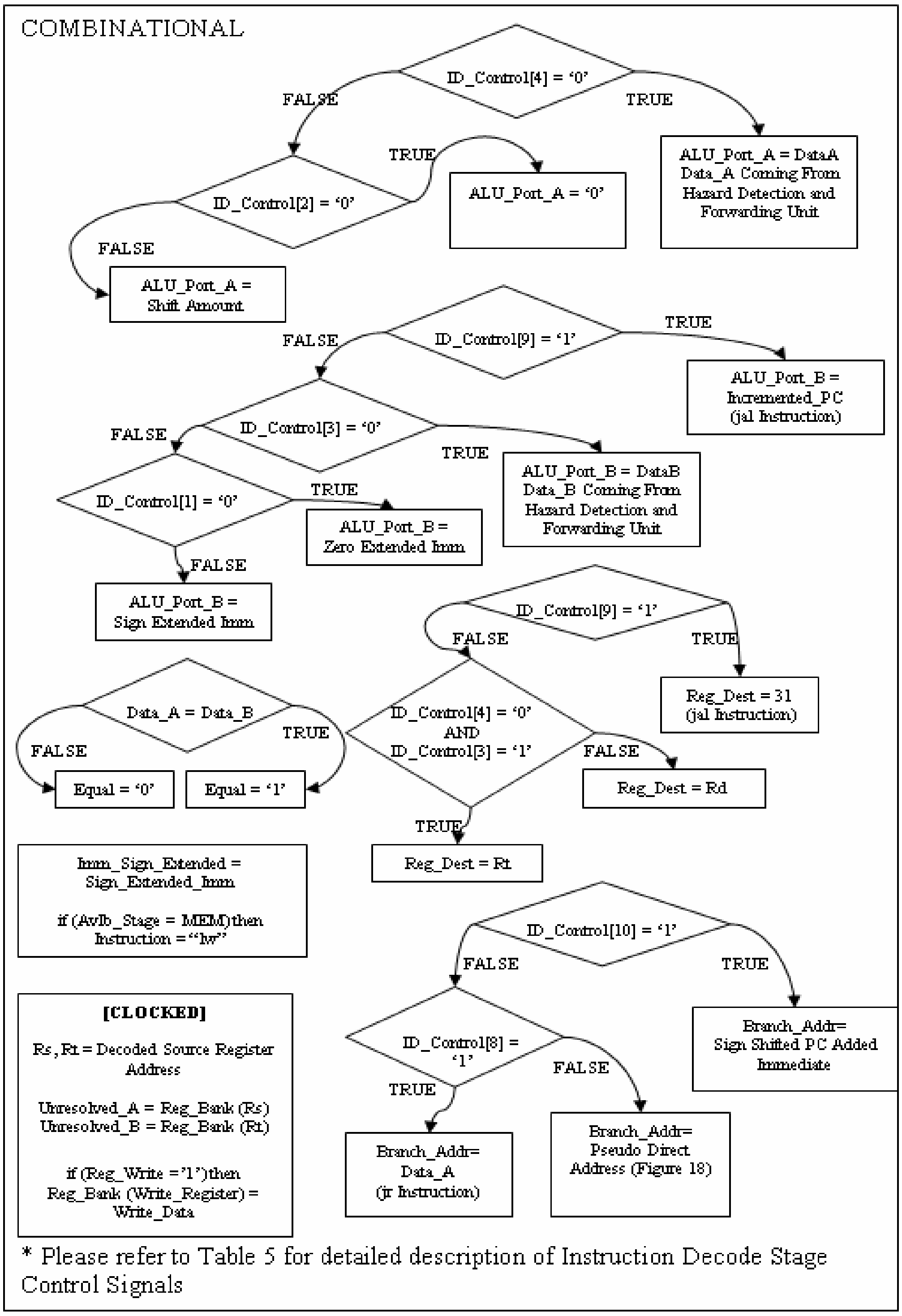
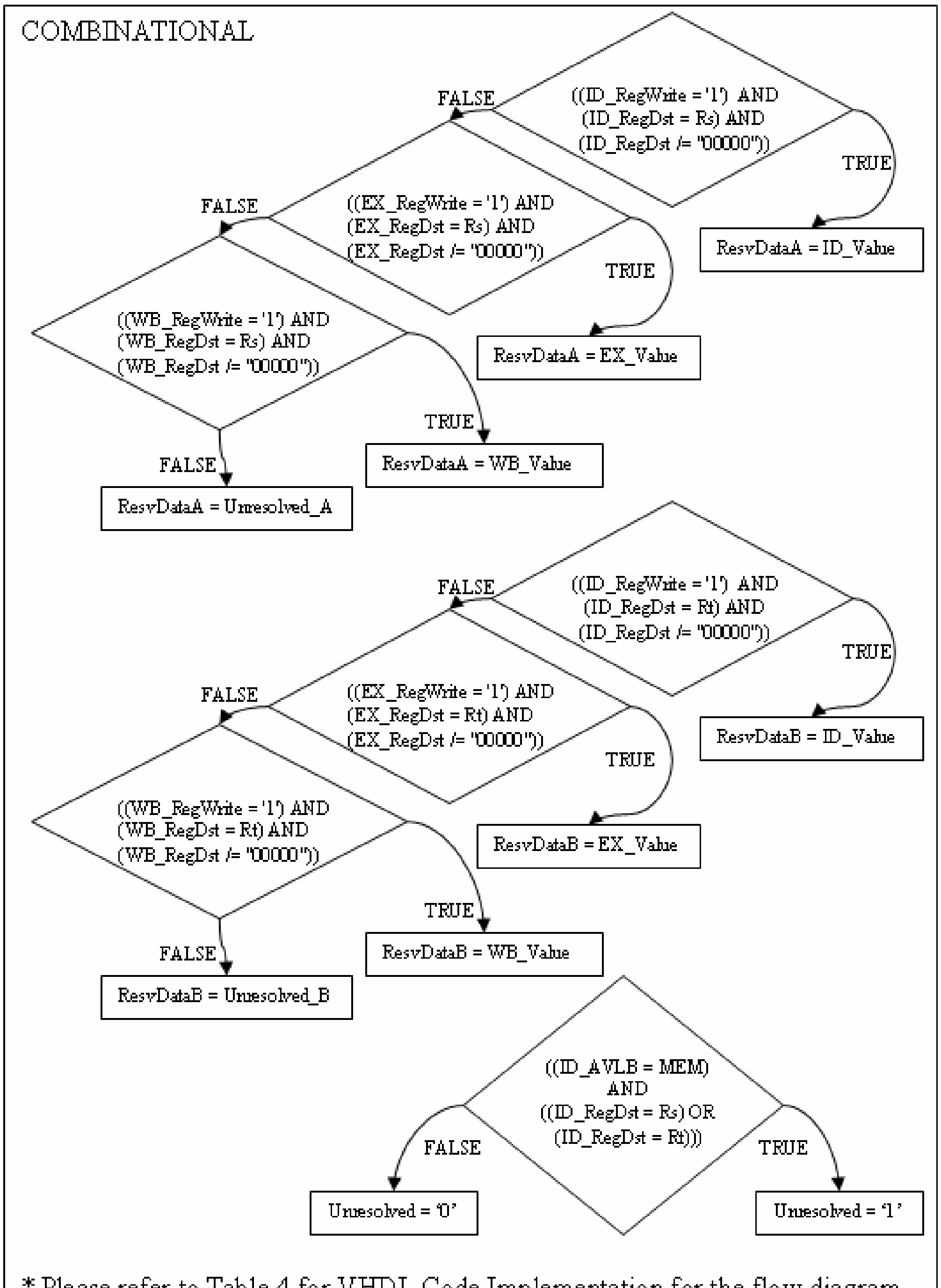

COMBINATIONAL
ID_Control[4] = '0'
FALSE
TRUE
ALU_Port_A = DataA Data_A Coming From Hazard Detection and Forwarding Unit
ID_Control[2] = '0'
TRUE
ALU_Port_A = '0'
FALSE
ALU_Port_A = Shift Amount
ID_Control[9] = '1'
FALSE
TRUE
ALU_Port_B = Incremented_PC (jal Instruction)
ID_Control[3] = '0'
FALSE
TRUE
ALU_Port_B = DataB Data_B Coming From Hazard Detection and Forwarding Unit
ID_Control[1] = '0'
TRUE
ALU_Port_B = Zero Extended Imm
FALSE
ALU_Port_B = Sign Extended Imm
ID_Control[9] = '1'
FALSE
TRUE
Reg_Dest = 31 (jal Instruction)
Data_A = Data_B
FALSE
TRUE
Equal = '0'
Equal = '1'
ID_Control[4] = '0' AND ID_Control[3] = '1'
FALSE
Reg_Dest = Rd
TRUE
Reg_Dest = Rt
Imm_Sign_Extended = Sign_Extended_Imm
if (Avlb_Stage = MEM) then Instruction = "lw"
ID_Control[10] = '1'
FALSE
TRUE
Branch_Addr= Sign Shifted PC Added Immediate
[CLOCKED]
Rs, Rt = Decoded Source Register Address
Unresolved_A = Reg_Bank (Rs)
Unresolved_B = Reg_Bank (Rt)
if (Reg_Write = '1') then Reg_Bank (Write_Register) = Write_Data
ID_Control[8] = '1'
TRUE
FALSE
Branch_Addr= Data_A (jr Instruction)
Branch_Addr= Pseudo Direct Address (Figure 18)
* Please refer to Table 5 for detailed description of Instruction Decode Stage Control Signals

Figure C.2: Instruction Decode Unit Flow Diagram

Forwarding and Hazard Detection Unit Flow Diagram


COMBINATIONAL
((ID_RegWrite = '1') AND (ID_RegDst = Rs) AND (ID_RegDst /= "00000"))
FALSE
TRUE
ResvDataA = ID_Value
((EX_RegWrite = '1') AND (EX_RegDst = Rs) AND (EX_RegDst /= "00000"))
FALSE
TRUE
ResvDataA = EX_Value
((WB_RegWrite = '1') AND (WB_RegDst = Rs) AND (WB_RegDst /= "00000"))
FALSE
TRUE
ResvDataA = WB_Value
ResvDataA = Unresolved_A
((ID_RegWrite = '1') AND (ID_RegDst = Rt) AND (ID_RegDst /= "00000"))
FALSE
TRUE
ResvDataB = ID_Value
((EX_RegWrite = '1') AND (EX_RegDst = Rt) AND (EX_RegDst /= "00000"))
FALSE
TRUE
ResvDataB = EX_Value
((WB_RegWrite = '1') AND (WB_RegDst = Rt) AND (WB_RegDst /= "00000"))
FALSE
TRUE
ResvDataB = WB_Value
ResvDataB = Unresolved_B
((ID_AVLB = MEM) AND ((ID_RegDst = Rs) OR (ID_RegDst = Rt)))
FALSE
TRUE
Unresolved = '0'
Unresolved = '1'

* Please refer to Table 4 for VHDL Code Implementation for the flow diagram given above.

Figure C.3: Forwarding and Hazard Detection Unit Flow Diagram

Instruction Execute Unit Flow Diagram

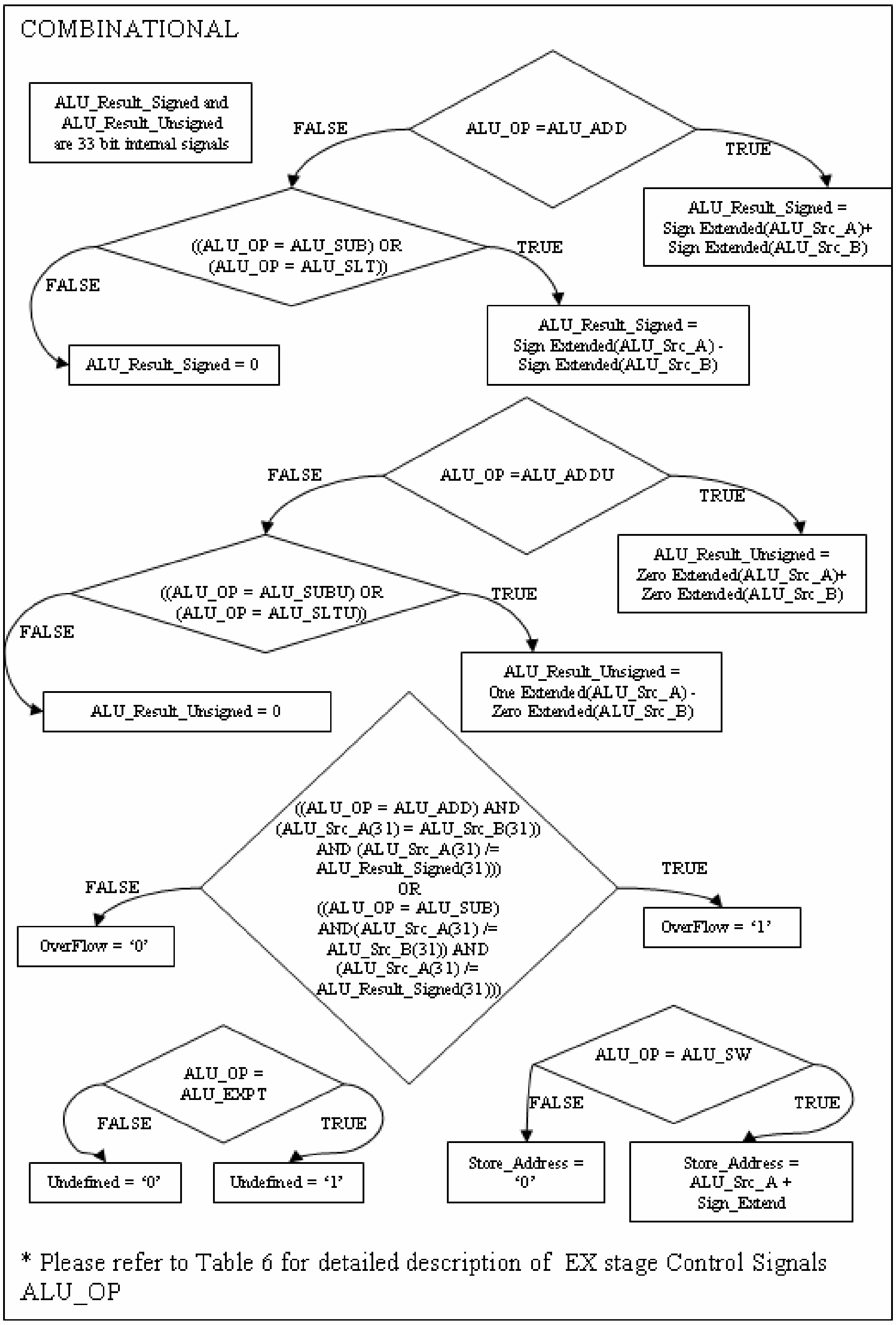
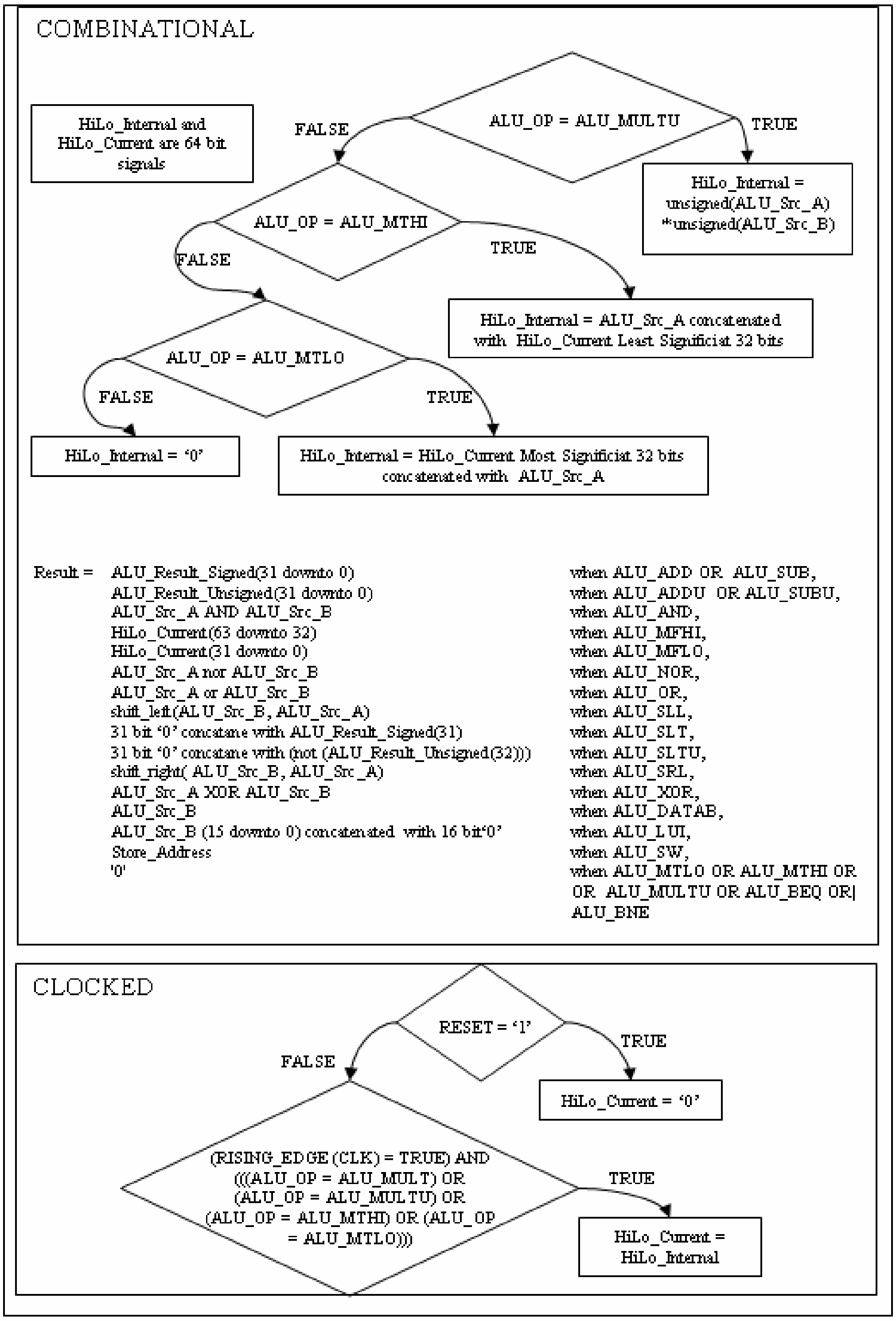

COMBINATIONAL
ALU_Result_Signed and ALU_Result_Unsigned are 33 bit internal signals
ALU_OP =ALU_ADD
FALSE
TRUE
ALU_Result_Signed = Sign Extended(ALU_Src_A)+ Sign Extended(ALU_Src_B)
((ALU_OP = ALU_SUB) OR (ALU_OP = ALU_SLT))
FALSE
TRUE
ALU_Result_Signed = 0
ALU_Result_Signed = Sign Extended(ALU_Src_A) - Sign Extended(ALU_Src_B)
ALU_OP =ALU_ADDU
FALSE
TRUE
ALU_Result_Unsigned = Zero Extended(ALU_Src_A)+ Zero Extended(ALU_Src_B)
((ALU_OP = ALU_SUBU) OR (ALU_OP = ALU_SLTU))
FALSE
TRUE
ALU_Result_Unsigned = 0
ALU_Result_Unsigned = One Extended(ALU_Src_A) - Zero Extended(ALU_Src_B)
((ALU_OP = ALU_ADD) AND (ALU_Src_A(31) = ALU_Src_B(31)) AND (ALU_Src_A(31) /= ALU_Result_Signed(31))) OR ((ALU_OP = ALU_SUB) AND(ALU_Src_A(31) /= ALU_Src_B(31)) AND (ALU_Src_A(31) /= ALU_Result_Signed(31)))
FALSE
TRUE
OverFlow = ‘0’
OverFlow = ‘1’
ALU_OP = ALU_EXPT
FALSE
TRUE
Undefined = ‘0’
Undefined = ‘1’
ALU_OP = ALU_SW
FALSE
TRUE
Store_Address = ‘0’
Store_Address = ALU_Src_A + Sign_Extend
* Please refer to Table 6 for detailed description of EX stage Control Signals ALU_OP

Figure C.4: Instruction Execute Unit Flow Diagram

Instruction Execute Unit Flow Diagram (continued)


COMBINATIONAL
HiLo_Internal and HiLo_Current are 64 bit signals
ALU_OP = ALU_MULTU
FALSE
TRUE
HiLo_Internal = unsigned(ALU_Src_A) *unsigned(ALU_Src_B)
ALU_OP = ALU_MTHI
FALSE
TRUE
HiLo_Internal = ALU_Src_A concatenated with HiLo_Current Least Significiat 32 bits
ALU_OP = ALU_MTLO
FALSE
TRUE
HiLo_Internal = '0'
HiLo_Internal = HiLo_Current Most Significiat 32 bits concatenated with ALU_Src_A
CLOCKED
RESET = '1'
FALSE
TRUE
HiLo_Current = '0'
(RISING_EDGE (CLK) = TRUE) AND (((ALU_OP = ALU_MULT) OR (ALU_OP = ALU_MULTU) OR (ALU_OP = ALU_MTHI) OR (ALU_OP = ALU_MTLO)))
TRUE
HiLo_Current = HiLo_Internal

```
Result =   ALU_Result_Signed(31 downto 0)                                   when ALU_ADD OR ALU_SUB,
           ALU_Result_Unsigned(31 downto 0)                                 when ALU_ADDU OR ALU_SUBU,
           ALU_Src_A AND ALU_Src_B                                          when ALU_AND,
           HiLo_Current(63 downto 32)                                       when ALU_MFHI,
           HiLo_Current(31 downto 0)                                        when ALU_MFLO,
           ALU_Src_A nor ALU_Src_B                                          when ALU_NOR,
           ALU_Src_A or ALU_Src_B                                           when ALU_OR,
           shift_left(ALU_Src_B, ALU_Src_A)                                 when ALU_SLL,
           31 bit '0' concatane with ALU_Result_Signed(31)                  when ALU_SLT,
           31 bit '0' concatane with (not (ALU_Result_Unsigned(32)))        when ALU_SLTU,
           shift_right( ALU_Src_B, ALU_Src_A)                               when ALU_SRL,
           ALU_Src_A XOR ALU_Src_B                                          when ALU_XOR,
           ALU_Src_B                                                        when ALU_DATAB,
           ALU_Src_B (15 downto 0) concatenated with 16 bit'0'              when ALU_LUI,
           Store_Address                                                    when ALU_SW,
           '0'                                                              when ALU_MTLO OR ALU_MTHI OR
                                                                            OR ALU_MULTU OR ALU_BEQ OR|
                                                                            ALU_BNE
```

Figure C.5: Instruction Execute Unit (continued) Flow Diagram

Data Memory Unit Flow Diagram

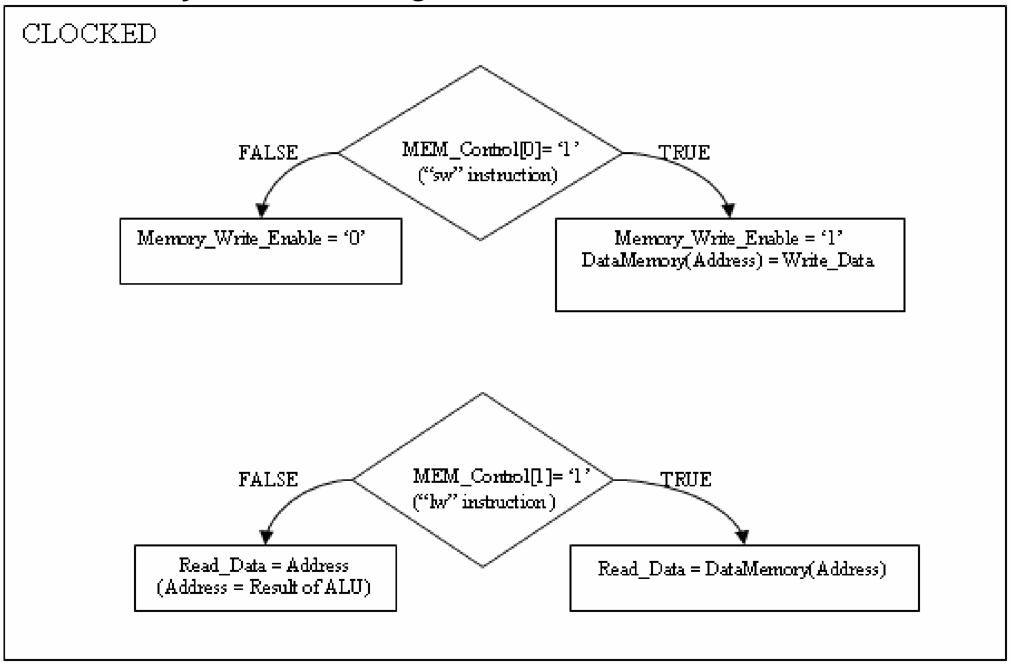

CLOCKED
MEM_Control[0]= '1'
("sw" instruction)
FALSE
TRUE
Memory_Write_Enable = '0'
Memory_Write_Enable = '1'
DataMemory(Address) = Write_Data
MEM_Control[1]= '1'
("lw" instruction)
FALSE
TRUE
Read_Data = Address
(Address = Result of ALU)
Read_Data = DataMemory(Address)

Figure C.6: Data Memory Unit Flow Diagram

Exception Detection Unit Flow Diagram

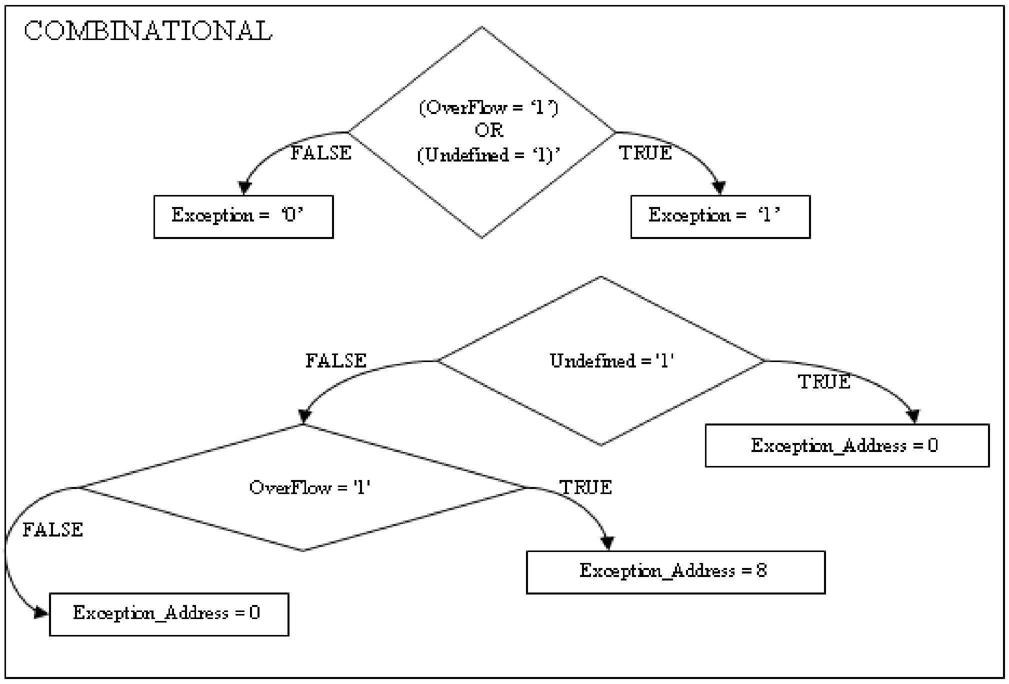


Figure C.7: Exception Detection Unit Flow Diagram

Register Block Unit Flow Diagram

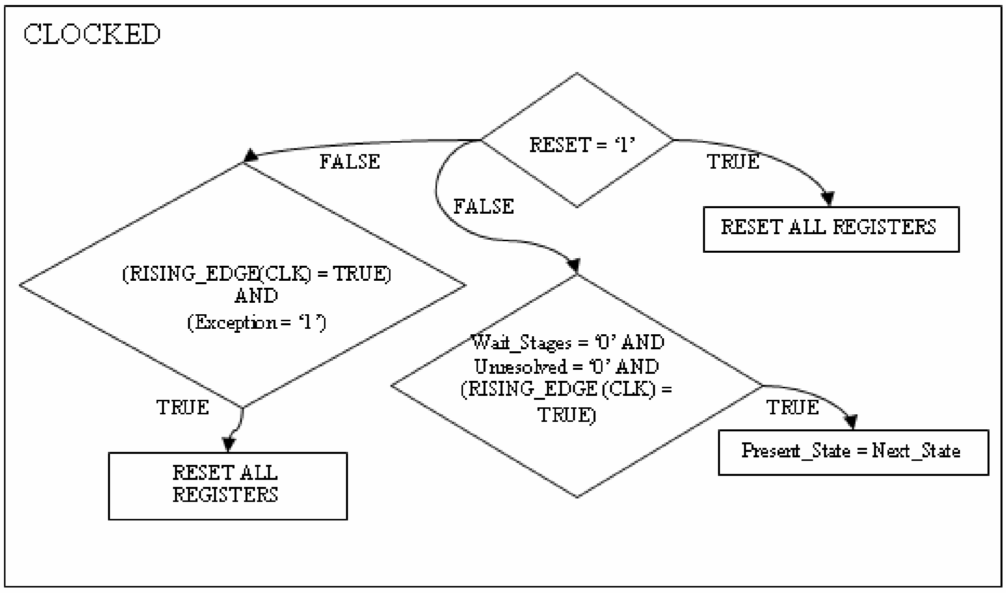


Figure C.8: Register Block Unit Flow Diagram

APPENDIX D

LAYOUT OF BOARD

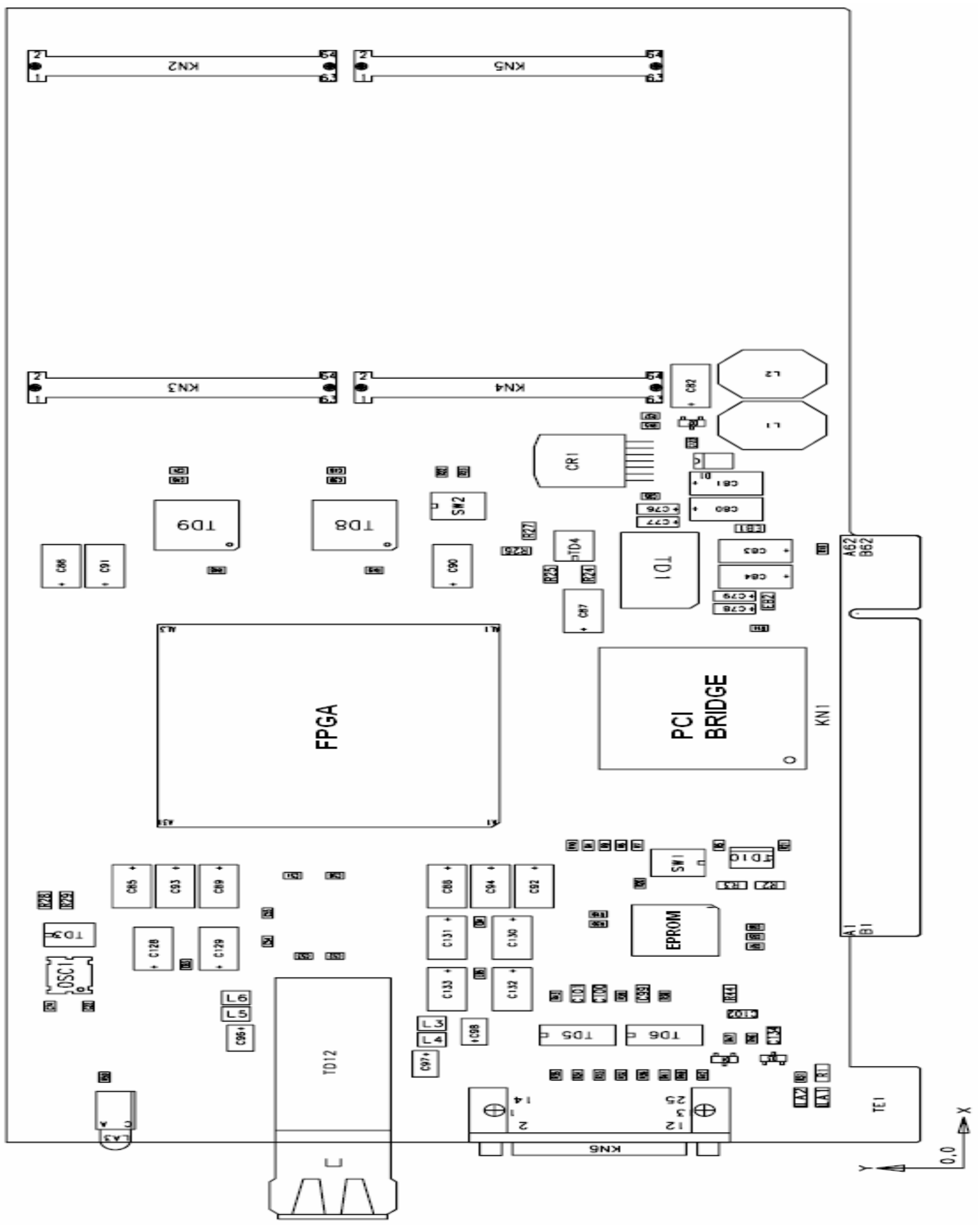


Figure D.1: Layout of Board

APPENDIX E

RESOURCES IN THIS THESIS

A soft copy of this thesis, in addition to all of the source codes of hardware and software mentioned about in this thesis are collected and presented in the CD attached to back cover.