Global Journal of Computer Science and Technology: A Hardware & Computation
Volume 15 Issue 2 Version 1.0 Year 2015
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Design of a Five Stage Pipeline CPU with Interruption System

By Abdulraqeb Abdullah Saeed Abdo & Professor. Liu Yijun
Guangdong University of Technology, China

Abstract- A central processing unit (CPU), also referred to as a central processor unit, is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system. The term has been in use in the computer industry at least since the early 1960s.The form, design, and implementation of CPUs have changed over the course of their history, but their fundamental operation remains much the same. A computer can have more than one CPU; this is called multiprocessing. All modern CPUs are microprocessors, meaning contained on a single chip. Some integrated circuits (ICs) can contain multiple CPUs on a single chip; those ICs are called multi-core processors. An IC containing a CPU can also contain peripheral devices, and other components of a computer system; this is called a system on a chip (SoC).Two typical components of a CPU are the arithmetic logic unit (ALU), which performs arithmetic and logical operations, and the control unit (CU), which extracts instructions from memory and decodes and executes them, calling on the ALU when necessary. Not all computational systems rely on a central processing unit. An array processor or vector processor has multiple parallel computing elements, with no one unit considered the "center". In the distributed computing model, problems are solved by a distributed interconnected set of processors.

Keywords: *CPU; MIPS; pipeline; Interruption.*

GJCST-A Classification : *B.2.1 B.5.1*

Strictly as per the compliance and regulations of:

Design of a Five Stage Pipeline CPU with Interruption System

Abdulraqeb Abdullah Saeed Abdo [α] & Professor. Liu Yijun [σ]

Abstract- A central processing unit (CPU), also referred to as a central processor unit, is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system. The term has been in use in the computer industry at least since the early 1960s.The form, design, and implementation of CPUs have changed over the course of their history, but their fundamental operation remains much the same. A computer can have more than one CPU; this is called multiprocessing. All modern CPUs are microprocessors, meaning contained on a single chip. Some integrated circuits (ICs) can contain multiple CPUs on a single chip; those ICs are called multi-core processors. An IC containing a CPU can also contain peripheral devices, and other components of a computer system; this is called a system on a chip (SoC).Two typical components of a CPU are the arithmetic logic unit (ALU), which performs arithmetic and logical operations, and the control unit (CU), which extracts instructions from memory and decodes and executes them, calling on the ALU when necessary. Not all computational systems rely on a central processing unit. An array processor or vector processor has multiple parallel computing elements, with no one unit considered the "center". In the distributed computing model, problems are solved by a distributed interconnected set of processors.

In this paper, firstly I introduce the development of CPU and the background of this paper. On the foundation of that I explicitly introduce the architecture of RISC CPU and MIPS CPU which based on RISC architecture, paving the way for the design of my paper. And then I discuss the design of a five stage pipeline CPU based on MIPS instruction. The CPU in this paper mainly includes pipeline module, control module, interruption module and RAM\ROM module. Using EDA verification software Modelsim to verify the design on functional level and gate level. Finally I download the design to a development-board based on Altera Cyclone4 FPGA. The result of the verification shows that all functions can be achieved.

Keywords: CPU; MIPS; pipeline; Interruption.

摘 要-

中央处理器广义上指一系列可以执行复杂的计算机程序的逻辑机器。这个空泛的定义很容易地将在“CPU”这个名称被普遍使用，之前的早期计算机也包括在内。无论如何，至少从20世纪60年代早期开始(Weik1961)，这个名称及其缩写已开始在电子计算机产业中得到广泛应用。尽管与早期相比，“中央处理器”在物理形态、设计制造和具体任务的执行上有了戏剧性的发展，但是其基本的操作原理一直没有改变。早期的中央处理器通常是为大型及特定应用的计算机而定制。但是，这种昂贵的为特定应用定制CPU的方法很大程度上已经让位于开发便宜、标准化、适用于一个或多个目的的处理器类。这个标准化趋势始于由单个晶体管组成的大型机和微机年代，随着集成电路的出现而加速。IC使得更为复杂的CPU可以在很小的空间中设计和制造（在微米的量级）。CPU的标准化和小型化都使得这一类数字设备在现代生活中的出现频率远远超过有限应用专用的计算机。现代微处理器出现在包括从汽车到手机到儿童玩具在内的各种物品中。

论文首先介绍了中央处理器发展的历史，以及本文设计的研究背景，并在此基础上着重介绍了精简指令RISCCPU的结构以及基于RISC结构的MIPSCPU的有关背景资料，为论文后续的设计做好铺垫。接着详细介绍了一款基于MIPS指令集的5级流水线CPU的设计。本CPU主要包括流水线模块，控制模块，中断处理模块，以及ROM和RAM模块。使用EDA验证软件Modelsim对设计进行了功能仿真和门级仿真，最后将设计下载到了基于Altera Cyclone4 FPGA 的开发板上.进行了验证验证结果表明本设计能实现所有功能。

关键词：CPU；MIPS；流水线；中断

I. Introduction

a) *Research status of CPU design and trend*

CPU is one of the main devices of a computer. Its main function is to explain computer's instruction and deal with the data of software. The programmable ability of computer generally means to program CPU. Central process unit, inner memory and input/output device are three core components of modern computer. Before 1970s, CPU is composed of several individual units. Later the CPU manufactured by semiconductor was developed. The complex circuits of a microprocessor can be made as a tiny unit with powerful function.

Central processor broadly means a series of logic machines that can perform complex computer programs. The term has been in use in the computer industry at least since the early 1960s. The form, design, and implementation of CPUs have changed over the course of their history, but their fundamental operation remains much the same.

A computer can have more than one CPU; this is called multiprocessing. All modern CPUs are microprocessors, meaning contained on a single chip. Some integrated circuits (ICs) can contain multiple CPUs on a single chip; those ICs are called multi-core processors. An IC containing a CPU can also contain peripheral devices, and other components of a computer system; this is called a system on a chip (SoC).

Author α σ: School of Computer Science Guangdong University of Technology Guangzhou, Guangdong, P. R. China, 510006.
e-mail: 970133919@qq.com

Two typical components of a CPU are the arithmetic logic unit (ALU), which performs arithmetic and logical operations, and the control unit (CU), which extracts instructions from memory and decodes and executes them, calling on the ALU when necessary.

Not all computational systems rely on a central processing unit. An array processor or vector processor has multiple parallel computing elements, with no one unit considered the "center". In the distributed computing model, problems are solved by a distributed interconnected set of processors.

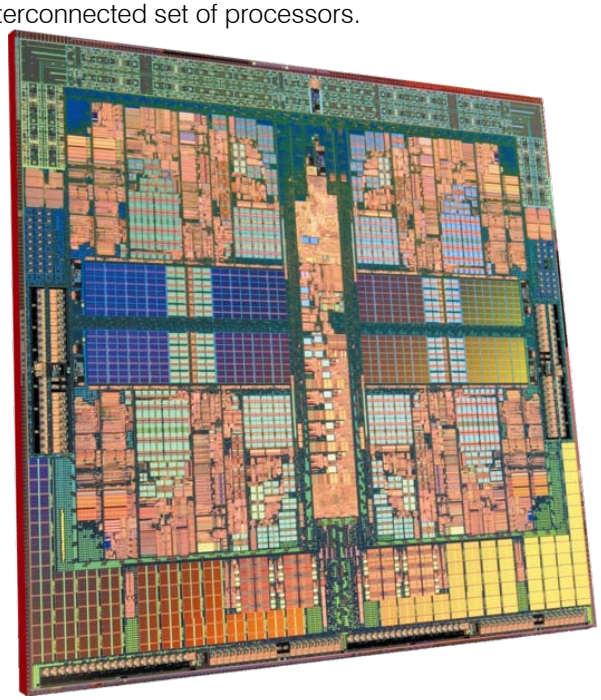


Figure 1-1 : Intel Phenom Quad-Core

Moore's Law makes us can expect the general situation in the future development of the CPU. Undoubtedly, high performance, low power consumption, high speed and low cost are the future direction of development.

1. Smaller wiring width and more transistors

Nowadays, Intel's and AMD's CPUs have used 0.18 or even 0.13 micrometer technology. For current silicon chips, reducing the wiring width is the key to raising the speed of the CPU.

Experts predict that the design of monolithic integrated chip system will reach such a number of indicators - the minimum feature size reaches 0.1 micrometer, chip integration reaches 200 million transistors. And some breakthroughs are also made from the aspect of the production process. IBM has developed a new chip packaging technology, by which the chip manufacturers can use aluminum instead of the traditional copper wire connections to connect transistors on a chip. Since copper conductors can be made thinner than the aluminum wire, so that the chip can be integrated on a larger number of transistors, which makes the packaging unit of the computing power has been greatly improved. Copper processor chip has become the future direction of development. Researches of copper chip have been performed by many chipmakers such as Intel and AMD.

2. 64-bit CPU chip manufacturers dominate the market

With the release of Intel Itanium, personal PC processor market will also be transited into 64-bit. 64-bitCPUcan handle64-bitdataand 64-bitaddresses and can provide higher accuracy and larger memory addressing range.

3. Higher bus speeds

Nowadays the bus has increasingly limited the performance of CPU. For which various manufacturers are seeking ways to improve bus speed. It's expected that within three years Bus speed should be able to exceed 1GHz.

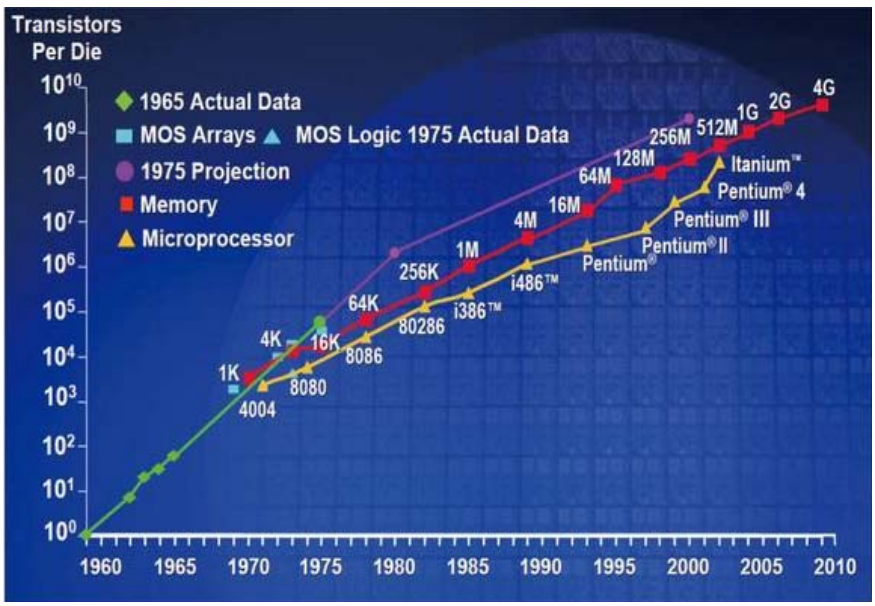


Figure 1-2 : Moore's Law and the development of IC integration

b) RISC and Pipeline processing

Reduced Instruction Set Computing is a design pattern for computer central processor. This design idea has reduced the number of instructions and addressing modes, making implementation easier, higher instructions parallelism, and a more efficient compiler. Current common RISC microprocessors includes DEC Alpha、ARC、ARM、AVR、MIPS、PA-RISC、Power Architecture (PowerPC、PowerXCell) and SPARC.

From the earliest, RISC's name comes from the Berkeley RISC project held by David Patterson in the University of California, Berkeley. But before him, people has been proposed a similar design philosophy. IBM 801 project, held by John Cork, started in 1975 and finished in 1980, probably is the first system designed under the concept of reduced instruction set. This design concept originated from the discovery that although many of the features of traditional processor are designed to make the code easier to write, but these complex features require several cycles to achieve and often are not used by the program. In addition, the difference between the speed of the processor and the main memory has become increasingly bigger. Prompted by these factors, a series of new technologies were introduced, making the processor's instruction executed in pipeline while reducing the number of processor memory access.

In the early period, Characteristics of such an instruction set is the small number of instructions, each instruction word in standard length, short execution time, and implementation details of the central processor for the machine-level program is visible and so. In fact in the later development, RISC and CISC have learned each other during the process of competitions. Now the RISC instruction set also has reached hundreds and operating cycle are no longer fixed. Nonetheless, fundamental principles of RISC design - optimization for pipelined processor – have not changed yet. And following this principle, a concurrent variant of RISC is developed – named VLIW – combining the short and length unity instructions into very long instruction. Each time you run a very long instruction, equal to concurrently run multiple short instructions.

On the other hand, the most common complex instruction set x86 CPU, although the instruction set is CISC, but it will make every effort to accelerate the hardware circuit to control commonly used simple instructions. Complex instruction which is not used often will be given to micro-code sequencer to “decode slowly and run slowly”. Hence it’s called "RISCy x86".

RISC processor should be designed to not only make effective execution pipeline processing, but also enable optimizing compiler optimized instruction generated code. Below, we will describe RISC processor design principles and techniques.

1. Efficient pipelining

The relation in pipeline means because there is some association in adjacent or similar instruction, later instruction cannot be executed within originally designated clock cycle. In general, the pipeline relation is divided into the following three types.

1. Data hazards Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions (sometimes known as race hazards).
2. Structural hazards A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A canonical example is a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory.[3] They can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline.
3. Control hazards (branch hazards) Branching hazards (also known as control hazards) occur with branches. On many instruction pipeline micro-architectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the fetch stage).

There are several methods used to deal with hazards, including pipeline stalls/pipeline bubbling, register forwarding, and in the case of out-of-order execution, the scoreboarding method and the Tomasulo algorithm.

2. Short cycle time

To increase the clock frequency by optimizing the process. To optimize circuit design structure, reduce instruction fetching time and read/write latency, thus reducing instruction period, which can greatly improve the efficiency of the machine.

3. Load/Store Structure

Load/Store Structure is used to transfer data between register file and memory. Load is used to fetch data from memory, while store is used to store data into memory. These two instructions are used frequently and is the most significant one in the instruction set. Because the other instructions can only handle register file.When data is in the memory, you have to load the data into register file, and store the data back into it after execution. In the register file, you don’t have to access the memory when data have to be used again. This Load/Store structure is the key for single period clock execution.

4. Simple fixed format instruction system

RISC designers focus on those frequently used commands, and try to make them simple and efficient features. For not commonly used functions we often

accomplished through a combination of instruction. Therefore, when implementing the special features on RISC machines, the efficiency may be lower, but you can use pipelining and superscalar techniques to improve and make up. While the CISC instruction set computer is rich, with special instructions to perform specific functions. Therefore, the efficiency of handling special tasks is higher.

5. No micro-code technology

Since RISC use Simple, rational and simplified instruction addressing modes, so it does not need micro-code technology, which means without micro-code ROM, but execute instruction directly in the hardware. This means eliminating the original machine microcode instructions into the intermediate step, and it reduce the number of machine cycles needed to execute an instruction. Also it saves space so that the chip can be saved using the microprocessor chip space expansion function.

6. Huge register file

A register file (register file) is an array consisting of a plurality of registers in the CPU, which usually realized by a fast static random access memory (SRAM).This RAM has a dedicated reading port and writing port, multiple concurrent accessing different registers. CPU's instruction set architecture is always defined a number of registers used for temporary storage of data between memory and CPU computing components. In a more simplified CPU, these architectures registers (architectural registers)-correspondence with the physical register within the CPU. In a more complex CPU, we use register renaming techniques, during the execution architecture which makes physical storage entry in the register which corresponds to the register file(physical entry stores)is dynamically changed. Register file is part of the instruction set architecture. The program can be accessed, which is transparent to the CPU cache (cache) different.

7. Harvard bus architecture

Harvard architecture is a memory structure to store program instructions and data separately. First, the CPU read program instruction in the program instruction memory. And then it gets the data address after decoding. Then it reads data in the according data memory, finally handle next execution (usually instruction). Instruction store and data store is separated, while Storing data and instructions can be simultaneously. Data and instruction can have different data width. For example, Microchip's PIC16's program instruction is 14 bit width, while data is 8 bit width. Harvard bus architecture CPU usually has relatively high execution efficient. Program instructions and data organization and storage instructions apart, implementation can be pre fetch the next instruction.

8. Delayed branch

Insert one or several effective instruction in the branch instruction. When the program is executed, after these into the instruction execution is completed, then executes the instruction, therefore, transfer instruction seems to be delayed, this technique known as delayed transfer of technology.

9. Hard-wired controller

Once control unit was build, unless redesigned and remapping, it's impossible to add new functions. Hard-wired controller is one of the most complex logic components in the CPU. When executes different machine instructions, it decodes the instruction through activates a series of different control signals, making the control unit has few explicit structure and in a mess. Since that, hard-wired controller is replaced by micro-program controller. However, under the same semiconductor process, hard-wired controller is faster than micro-program controller.

10. Assembly technology optimization

11. High-level programming language oriented

c) Structure and content

This paper describes the design of a five-stage pipeline CPU with interruption system. Including CPU's research background, instruction set, pipeline data path and the design of interruption and exception system. And we use EDA tools for the simulation of the design. Finally we proof that the design meets the performance requirement.

The chapters are arranged as follows

Chapter one is the brief introduction of the research background. It mainly introduces the background and related research status and CPU's integrated circuit industry.

Chapter two is the brief introduction of the development platform and MIPS architecture. It mainly introduces the software and hardware development platform for the project and FPGA's design flow. It also introduces MIPS architecture.

Chapter three describes the design of pipeline data path. It introduces the pipeline design method, the composition of the pipeline and design and verification of associated component.

Chapter four describes the design of interruption and exception circuits. It introduces the principal of exception circuits and verification of related components.

Chapter five is CPU functional verification.

II. Development Platform and mips Architecture

This chapter mainly introduces the development platform of this paper – the EDA development and verification system based on Altera Cyclone4 FPGA and

Quartus + Modelsim. Then we will introduce the background of our design – MIPS instruction set and architecture.

a) Technology for CPU hardware design and implementation

i. *Hardware description language*

In electronics, a hardware description language or HDL is a specialized computer language used to program the structure, design and operation of electronic circuits, and most commonly, digital logic circuits.

A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis, simulation, and simulated testing of an electronic circuit. It also allows for the compilation of an HDL program into a lower level specification of physical electronic components, such as the set of masks used to create an integrated circuit.

A hardware description language looks much like a programming language such as C; it is a textual description consisting of expressions, statements and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time.

HDLs form an integral part of Electronic design automation systems, especially for complex circuits, such as microprocessors.

ii. *Structure of HDL*

HDLs are standard text-based expressions of the spatial and temporal structure and behavior of electronic systems. Like concurrent programming languages, HDL syntax and semantics include explicit notations for expressing concurrency. However, in contrast to most software programming languages, HDLs also include an explicit notion of time, which is a primary attribute of hardware. Languages whose only characteristic is to express circuit connectivity between a hierarchy of blocks are properly classified as netlist languages used in electric computer-aided design (CAD). HDL can be used to express designs in structural, behavioral or register-transfer-level architectures for the same circuit functionality; in the latter two cases the synthesizer decides the architecture and logic gate layout.

HDLs are used to write executable specifications for hardware. A program designed to implement the underlying semantics of the language statements and simulate the progress of time provides the hardware designer with the ability to model a piece of hardware before it is created physically. It is this excitability that gives HDLs the illusion of being programming languages, when they are more precisely classified as specification languages or modeling languages. Simulators capable of supporting discrete-event (digital) and continuous-time (analog) modeling exist, and HDLs targeted for each are available.

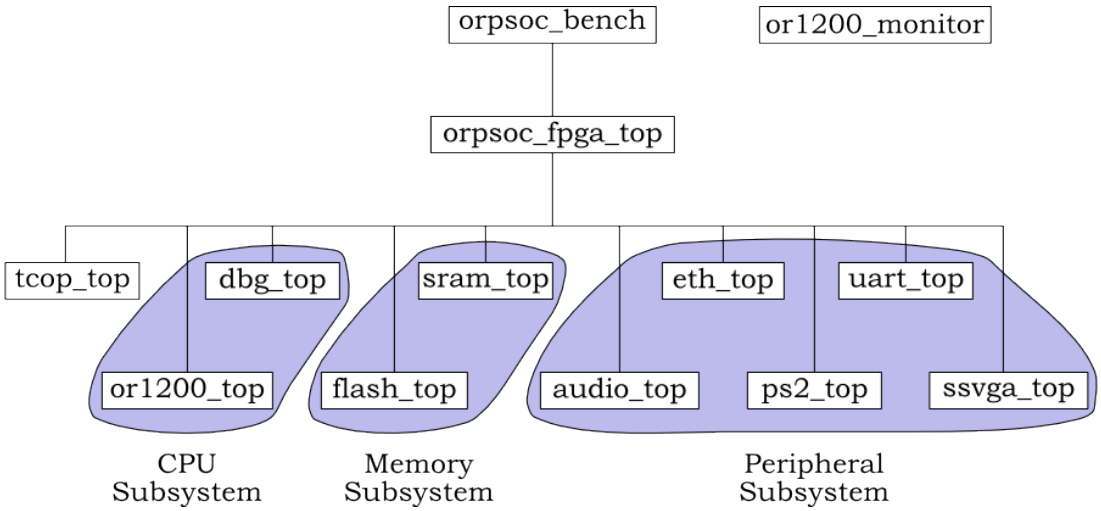


Figure 2-1 : an example of Verilog HDL hierarchy

iii. *Comparison with control-flow languages*

It is certainly possible to represent hardware semantics using traditional programming languages, which operate on control flow semantics as opposed to data flow, such as C++, although to function as such, programs must be augmented with extensive and unwieldy class libraries. Primarily, however, software programming languages do not include any capability for explicitly expressing time, and this is why they cannot function as hardware description languages. Before the recent introduction of SystemVerilog, C++ integration with a logic simulator was one of the few ways to use OOP in hardware verification. SystemVerilog is the first major HDL to offer object orientation and garbage collection.

Using the proper subset of hardware description language, a program called a synthesizer (or synthesis tool) can infer hardware logic operations from the language statements and produce an equivalent netlist of generic hardware primitives to implement the

specified behavior. Synthesizers generally ignore the expression of any timing constructs in the text. Digital logic synthesizers, for example, generally use clock edges as the way to time the circuit, ignoring any timing constructs. The ability to have a synthesizable subset of the language does not itself make a hardware description language.

iv. *Design using HDL*

As a result of the efficiency gains realized using HDL, a majority of modern digital circuit design revolves around it. Most designs begin as a set of requirements or a high-level architectural diagram. Control and decision structures are often prototyped in flowchart applications, or entered in a state-diagram editor. The process of writing the HDL description is highly dependent on the nature of the circuit and the designer's preference for coding style. The HDL is merely the 'capture language,' often beginning with a high-level algorithmic description such as a C++ mathematical model. Designers often use scripting languages (such as Perl) to automatically generate repetitive circuit structures in the HDL language. Special text editors offer features for automatic indentation, syntax-dependent coloration, and macro-based expansion of entity/architecture/signal declaration.

The HDL code then undergoes a code review, or auditing. In preparation for synthesis, the HDL description is subject to an array of automated checkers. The checkers report deviations from standardized code guidelines, identify potential ambiguous code constructs before they can cause misinterpretation, and check for common logical coding errors, such as dangling ports or shorted outputs. This process aids in resolving errors before the code is synthesized.

In industry parlance, HDL design generally ends at the synthesis stage. Once the synthesis tool has mapped the HDL description into a gate netlist, this netlist is passed off to the back-end stage. Depending on the physical technology (FPGA, ASIC gate array, ASIC standard cell), HDLs may or may not play a significant role in the back-end flow. In general, as the design flow progresses toward a physically realizable form, the design database becomes progressively more laden with technology-specific information, which cannot be stored in a generic HDL description. Finally, an integrated circuit is manufactured or programmed for use.

v. *Simulating and debugging HDL code*

Essential to HDL design is the ability to simulate HDL programs. Simulation allows an HDL description of a design (called a model) to pass design verification, an important milestone that validates the design's intended function (specification) against the code implementation in the HDL description. It also permits architectural exploration. The engineer can experiment with design choices by writing multiple variations of a base design, then comparing their behavior in simulation. Thus, simulation is critical for successful HDL design.

To simulate an HDL model, an engineer writes a top-level simulation environment (called a testbench). At minimum, a testbench contains an instantiation of the model (called the device under test or DUT), pin/signal declarations for the model's I/O, and a clock waveform. The testbench code is event driven: the engineer writes HDL statements to implement the (testbench-generated) reset-signal, to model interface transactions (such as a host–bus read/write), and to monitor the DUT's output. An HDL simulator — the program that executes the testbench — maintains the simulator clock, which is the master reference for all events in the testbench simulation. Events occur only at the instants dictated by the testbench HDL (such as a reset-toggle coded into the testbench), or in reaction (by the model) to stimulus and triggering events. Modern HDL simulators have full-featured graphical user interfaces, complete with a suite of debug tools. These allow the user to stop and restart the simulation at any time, insert simulator breakpoints (independent of the HDL code), and monitor or modify any element in the HDL model hierarchy. Modern simulators can also link the HDL environment to user-compiled libraries, through a defined PLI/VHPI interface. Linking is system-dependent (Win32/Linux/SPARC), as the HDL simulator and user libraries are compiled and linked outside the HDL environment.

Design verification is often the most time-consuming portion of the design process, due to the disconnect between a device's functional specification, the designer's interpretation of the specification, and the imprecision of the HDL language. The majority of the initial test/debug cycle is conducted in the HDL simulator environment, as the early stage of the design is subject to frequent and major circuit changes. An HDL description can also be prototyped and tested in hardware — programmable logic devices are often used for this purpose. Hardware prototyping is comparatively more expensive than HDL simulation, but offers a real-world view of the design. Prototyping is the best way to check interfacing against other hardware devices and hardware prototypes. Even those running on slow FPGAs offer much shorter simulation times than pure HDL simulation.

b) EDA system

i. *QuartusII*

Quartus II is a software tool produced by Altera for analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. The latest version is 13sp1 which is a service pack of version 13.

ii. *Modelsim*

Mentor Graphics was the first to combine single kernel simulator (SKS) technology with a unified debug environment for Verilog, VHDL, and SystemC. The combination of industry-leading, native SKS performance with the best integrated debug and analysis environment make ModelSim® the simulator of choice for both ASIC and FPGA designs. The best standards and platform support in the industry make it easy to adopt in the majority of process and tool flows.

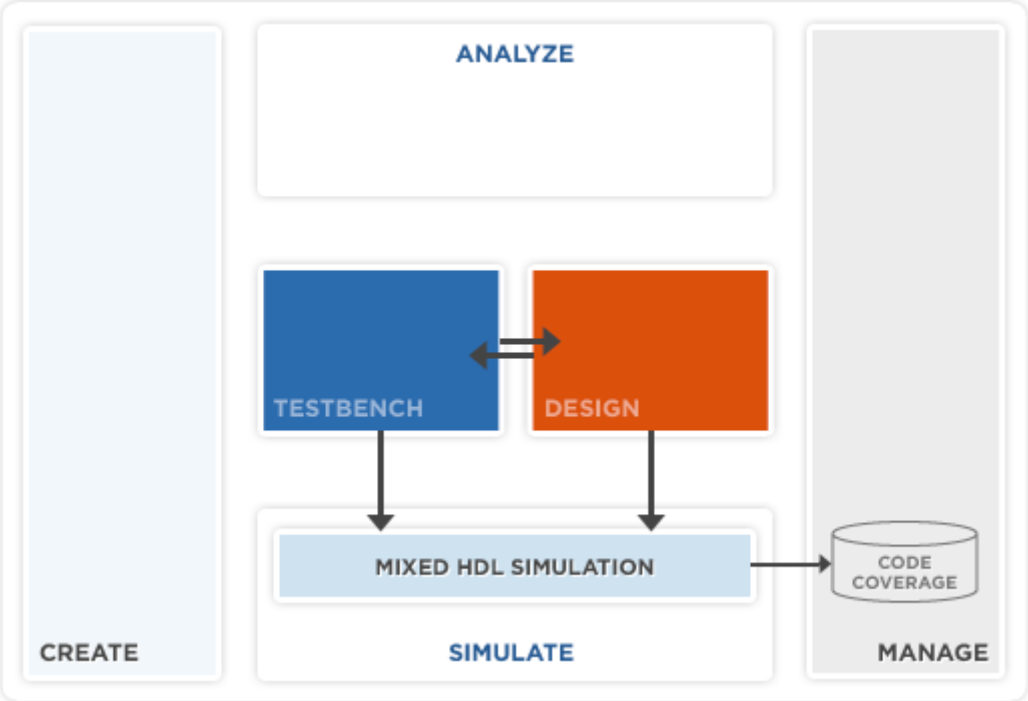


Figure 2-2 : Modelsim simulation structure

c) *FPGA design and verification*

Field-programmable gate array (FPGA) is a device that has numerous gate (switch) arrays and can be programmed on-board through dedicated Joint Test Action Group (JTAG) or on-board devices or using remote system through Peripheral Component Interconnect Express (PCIe), Ethernet, etc. FPGAs are based on static random-access memory (SRAM). The contents of the memory of an FPGA erase once the power is turned off. Usually, FPGAs can be programmed several thousands of times without the device getting faulty.

Fig. 2-3 shows the architecture of an FPGA. It includes logic blocks, input/output (I/O) cells, phase-locked loops/delay-locked loops (PLLs/DLLs), block RAM and interconnecting matrix. Nowadays, FPGAs are also coming up with several hard intellectual property (IP) blocks of PCIe, Ethernet, Rocket I/O, PHYs for DDR3 interfaces and processor cores (for example, PowerPC in Xilinx Virtex-5 FPGA and ARM cores in both Xilinx and Altera series FPGAs).

To level up with the new technology, both Xilinx and Altera have come up with new series of FPGAs (Virtex 7 from Xilinx and Stratix-V from Altera), which are manufactured with TSMC's 28nm silicon technology. These FPGAs focus on a high speed with low power consumption using various parameters and bringing down the FPGA core voltage to as low as 0.9V. Along with the new FPGAs, Xilinx and Altera are also focused on improving their synthesis tools to meet the routing constraints and to analyze the timing and power consumption of the FPGA.

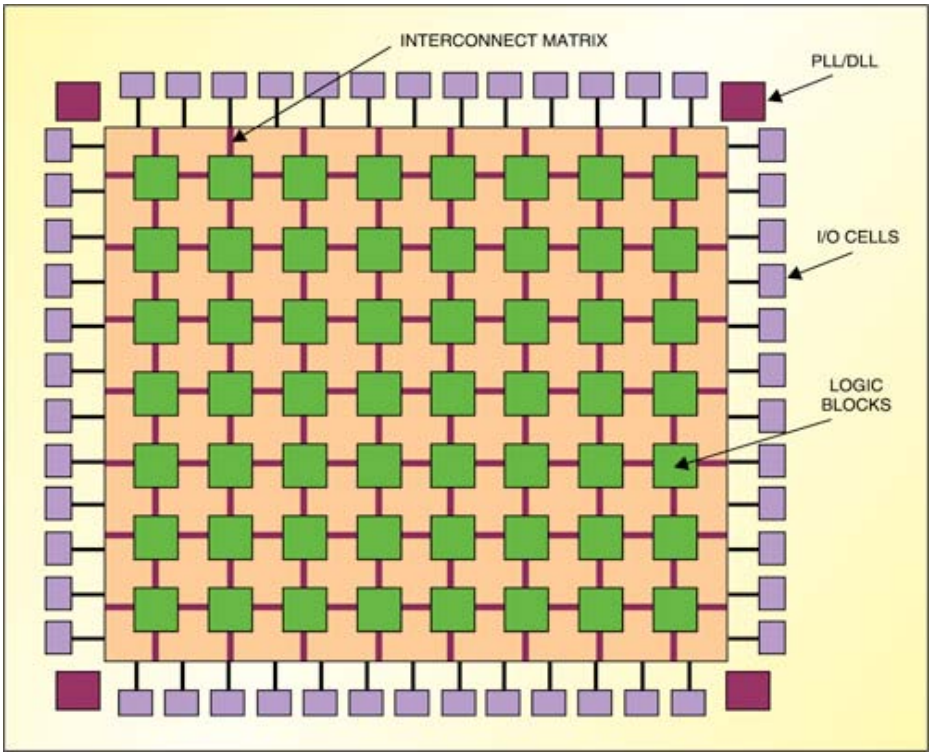


Figure 2-3 : FPGA architecture

As the aim here is to learn the basic technique of FPGA design to work with both the tools and devices, let's get back to the design flow through the steps.

Step 1: Requirement analysis and SRS preparation

Before starting work on the design, all requirements should be documented as system requirement specification (SRS) by designers and approved by various levels in the organization, and most importantly, the client. During this phase, FPGA designers, along with the hardware team, should identify suitable FPGAs for the project. This is very important because designers need to know parameters such as the I/O voltage levels, operating frequency and external peripheral interfaces.

It is also important to determine which IP cores are available with the tools or FPGA family used for the project. Some IP cores are free, while others are licensed and paid for. This cost should be reviewed several times by the team before releasing it to the client and listed separately for approval from the client or management.

The SRS should contain the following (the list pertains to the FPGA only):

1. Aim of the project
2. Functionalities to be handled by the design, followed by a short description
3. A concept-level block diagram depicting the major internal peripherals/IPs of the FPGA
4. FPGA vendor, family, speed grade, package, core voltage, supported I/O levels, commercial/industrial type
5. List of blocks that will be used as IPs. Mention clearly what's available for free with the vendor-provided IPs, hard IPs available within the FPGA and paid licensed IPs to be used
6. Type of processor interfaces used (soft processor or external processor interfaces)
7. Type of memory interfaces used
8. A section about the timing diagram of the major peripheral interfaces such as the processor interface and flash interface.
9. Type of FPGA configurations to be used
10. Reset and clock interface planned
11. A brief summary of the estimated resources required for implementation of the logic and I/O pins to be used
12. HDL (VHDL, Verilog, 'C' or mixed) used for RTL coding, tools and version to be used for synthesis, implementation and simulation

To calculate the approximate resources required, go through the IP datasheets for the resources used for each IP, and also calculate the resources used by custom RTL. There is no rule of thumb for calculating resources at this level. These can be calculated approximately based on experience, reviews or analysis. The most important thing is to get the resource requirement reviewed by the hardware team, software team and a third party several times before submitting it to the client.

Step 2: Detailed design document preparation

Once the SRS is approved by the client, the next phase is to make the detailed design document.

This document should consist of:

1. Brief introduction to the project
2. FPGA part details with proper specification
3. Detailed block diagram depicting the internal modules of the FPGA design
4. Top-level module block diagram showing input and output ports with their active levels and voltage levels which are connected to the external peripherals, connectors and debug points
5. Hierarchical tree of the modules
6. Each module should have:
   i. Detailed explanation of the functionality
   ii. Register information
   iii. List of input and output ports with source and destination module name, and active level of the signal
   iv. A block diagram/digital circuit diagram of finite-state machines indicating how the RTL will be implemented
   v. Clock frequency to be used, if a synchronous module is used
   vi. Reset logic implementation
   vii. File name which will be implemented
   viii. Approximate FPGA resource utilization
   ix. Testbench for testing each module independently
7. Input system clock frequency and reset level
8. Explanation of how the internal clock frequencies are derived—using phase-locked loop (PLL) or delay-locked loop (DLL) with the input clock. Also, explain how the global clock buffers are used. Mention clock signals with their frequency and voltage levels that are driven out of the FPGA for external peripherals.
9. A simulation environment setup for the design (called 'device under test') with a top-level testbench. A block diagram indicating how the clock source, reset and pattern generators, and bus functional modes are connected to the top-level module under testing will be helpful here. Mention how log files are used to register the activity of the required signal
10. Make a page with the heading 'FPGA Synthesis and Resource Utilization.' Keep it blank with a note that once the final implementation is done, this page will be updated
11. Under the heading 'Timing Analysis,' mention the major timing parameters of the control signals to be maintained, with a timing budget and waveform drawn manually or using timing analyzer tool. Mention the major timing constraints that will be used in the UCF or QSF files of the design

As mentioned in Step 1, the FPGA team members, hardware and software team members and architects should review the document at several stages before releasing it to the client.

Step 3: Design entry and functional simulation

Each module owner should develop a testbench for his module, capture simulated waveforms or assertion-based log report, and get it reviewed by the team lead. Before going for synthesis, every module should be verified thoroughly for functionality using simulation. Regular code review will help to reduce errors and simulation time. Once the simulation of individual modules is done, the next step is to integrate the module and do full-system-level functional simulation with assertion-based log report.

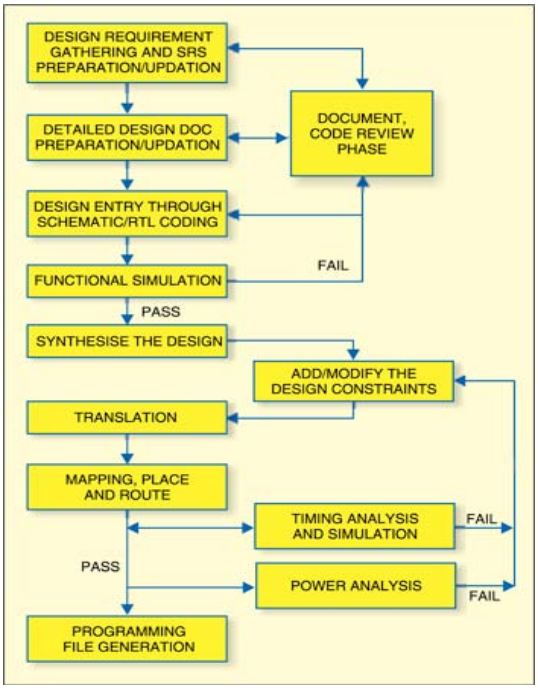


Figure 2-4 : FPGA design flowchart

Step 4: Synthesis

If the functional simulation satisfies the requirement, the next phase is synthesis.

In this phase, the integrated project is synthesized using a vendor-specific synthesis tool based on the optimization settings. Whenever RTL is modified, it is always good to complete Step 3 with unit-level and full-system functional simulation. Always follow vendor-specific coding guidelines and library modules for better optimization of the design.

During this phase, synthesis tools verify the design for syntax errors and do block-level floor planning.

Step 5: Adding design constraints

Once synthesis is complete, constraints can be added to the design. These constraints are usually included in a separate flee where the designer lists out the signal with its corresponding FPGA pin number, I/O voltage levels, current-driving strength for output signals, input clock frequency, hard block or module location, timing paths to be ignored, false paths, other IP-specific constraints recommended by the vendor, etc. This information is passed on to the placement phase.

Step 6: Placement and routing phase

Before routing, the synthesis tool maps the buffers, memory and clock buffers to the specific vendor libraries. That is, in this phase, logical blocks are translated into physical file format. Then, in the place-and-route process, the tool places and routes the design considering the user constraints and optimization techniques. Timing simulation can be done at this stage to verify the functionality, so that the design meets all the functional and timing requirements.

Step 7: Programming file generation

After obtaining a satisfactory timing and functional behavior of the design, it is time to generate the bit file that is downloaded to the FPGA to test the functionality on the board with actual peripherals.

For each stage, the tool will provide the corresponding report files, using which the designer can analyses time delays, power, resource usage, unrouted signals and I/O pins list.

In short

To summarize the above points, the FPGA design flow is shown as a simple flow-charting Figure 2-3. There may be minor variations in the design flow during the requirement stage and the design and document preparation phase, from one organization or project to another, but the overall FPGA design flow remains the same.

d) MIPS architecture

MIPS (originally an acronym for Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies (formerly MIPS Computer Systems, Inc.). The early MIPS architectures were 32-bit, with 64-bit versions added later. Multiple revisions of the MIPS instruction set exist, including MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS32, and MIPS64. The current revisions are MIPS32 (for 32-bit implementations) and MIPS64 (for 64-bit implementations). MIPS32 and MIPS64 define a control register set as well as the instruction set.

i. *MIPS classical five-stage pipeline*

MIPS architecture is designed based on pipeline architecture. Every MIPS instruction is divided into five stages once it is fetched from cache and every stage takes stable time. Usually each stage takes one clock cycle, while RD/WB operation takes half clock cycle. The execution process of MIPS processor is divided into five stages as follow.

1. Instruction Fetch (IF) Stage
   a. Instruction Fetch

Instruction's address in PC is applied to instruction memory that causes the addressed instruction to become available at the output lines of instruction memory.

   b. Updating PC

The address in PC is incremented by 4 but what is written in PC is determined by the control signal PCSrc. Depending upon the status of control signal PCSrc, PC is either written by the branch target address (BTA) or the sequential address (PC + 4).

2. Instruction Decode (ID) Stage

a. Instruction is decoded by the control unit that takes 6-bit opcode and generates control signals.
b. The control signals are buffered in the pipeline registers until they are used in the concerned stage by the corresponding instruction.
c. Registers are also read in this stage. Note that the first source register's identifier in every instruction is at bit positions [25:21] and second source register's identifier (if any) is at bit positions [20:16].
d. The destination register's identifier is either at bit positions [15:11] (for R-type) or at [20:16] (for lw and addi). The correct destination register's identifier is selected via multiplexer controlled by the control signal RegDst. However, this multiplexer is placed in the EX stage because the instruction decoding is not finished until the second stage is complete. But this identifier is buffered until the WB stage because an instruction write sa register in the WB stage.

3. Execution (EX) Stage

a. This stage is marked by the use of ALU that performs the desired operation on registers(R-type), calculates address (memory reference instructions), or compares registers (branch).

b. An ALU control accepts 6-bitfunctfield and 2-bit control signal ALU Op to generate the required control signal for the ALU.
c. BTA is also calculated in the EX stage by a separate adder

4. Memory (M) Stage

a. Data memory is read (lw) or written (sw) using the address calculated by the ALU in EXstage.
b. ZERO output of ALU and BRANCH signal generated by the control unit are ANDed to determine the fate of branch (taken or not taken).

5. Write Back (WB) Stage

a. Result produced by ALU in EX stage (R-type) or data read from data memory in M stage(lw) is written in destination register. The data to be written in destination register is selected via multiplexer controlled by the control signal MemToReg.

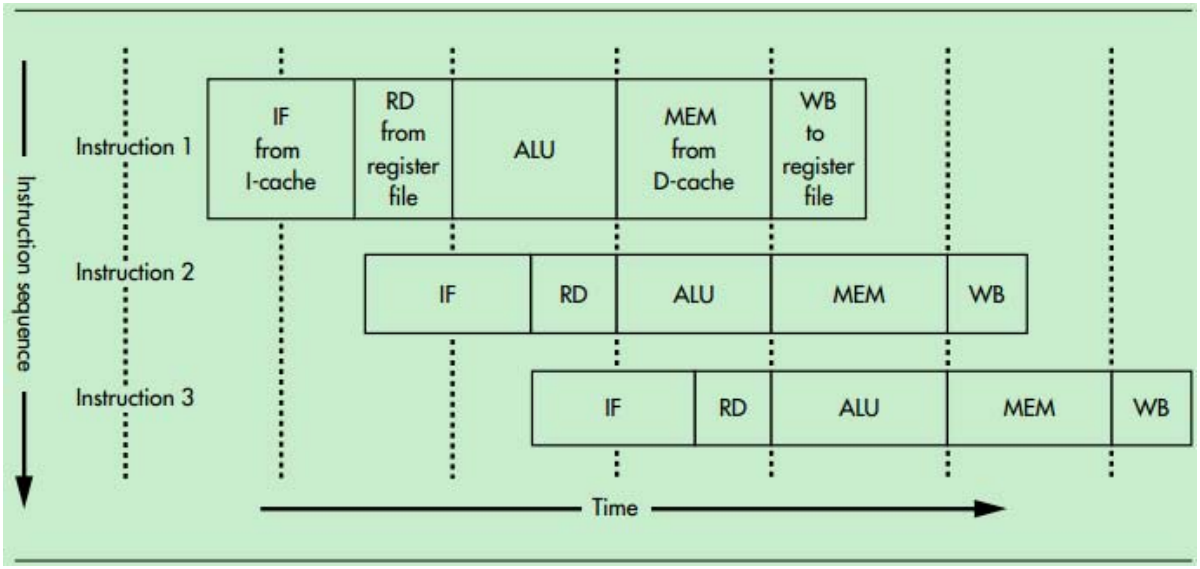


Figure 2-5 : MIPS five-stage pipeline

ii. *MIPS register*

MIPS have 32 common register ($0-$31). The Table 2-1 below describes the aliases and function of these 32 registers.

Table 2-1 : MIPS register

;REGISTER	NAME	USAGE
$0	$zero	constant value 0
$1	$at	Reserved for assembler
$2-$3	$v0-$v1	values for results and expression evaluation
$4-$7	$a0-$a3	arguments
$8-$15	$t0-$t7	Temporary or random
$16-$23	$s0-$s7	saved
$24-$25	$t8-$t9	Temporary or random
$28	$gp	Global Pointer
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	return address

iii. *MIPS instruction set*

Instructions are divided into three types: R, I and J. Every instruction starts with a 6-bit opcode. In addition to the opcode, R-type instructions specify three registers, a shift amount field, and a function field; I-type instructions specify two registers and a 16-bit immediate value; J-type instructions follow the opcode with a 26-bit jump target.

Since the MIPS instruction set instruction involves many, not all will be used in our design, so only the selection and design-related instructions are described in this article.

Now the MIPS instruction used in this article are listed below, there are two main type of instruction - integer instruction and interrupt and exception handling instructions.

Table 2-2 : MIPS instruction set

integer instruction	meaning
add	add rd, rs, rt #rd ← rs op rt
sub	sub rd, rs, rt #rd ← rs op rt
and	and rd, rs, rt #rd ← rs op rt
Or	or rd, rs, rt #rd ← rs op rt
Xor	xor rd, rs, rt #rd ← rs op rt
Sll	sll rd, rt, sa #rd ← rt shift sa
Srl	srl rd, rt, sa #rd ← rt shift sa
Sra	sra rd, rt, sa #rd ← rt shift sa
Jr	jr rs #PC ← rs
addi	addi rt, rs, imm #rt ← rs + imm
andi	andi rt, rs, imm #rt ← rs op imm
Ori	ori rt, rs, imm #rt ← rs op imm
xori	xori rt, rs, imm #rt ← rs op imm
Lw	lw rt, offset(rs) #rt ← memory[rs + offset]
Sw	sw rt, offset(rs) #memory[rs + offset] ← rt
beq	beq rs, rt, label #if (rs==rt) PC ← label
bne	bne rs, rt, label #if (rs!=rt) PC ← label
Lui	j target #PC ← target
J	jal target #r31← PC+8 ; PC ← target
Jal	jr rs #PC ← rs

Interruption and exception instruction	meaning
syscall	System call
eret	Exception execution return
mfc0	Fetch control word
mtc0	Store control word

III. Design of the Pipeline Circuit

a) The basic concepts of pipelining

CPU pipeline is a kind of technology that decomposes instruction into multiple steps, making each step of the operation overlapped, so as to realize a few instructions in parallel processing and to speed up the programming process. Each step has its dependent circuit to handle. When a step is finished, it goes into the next step, and the further step handles the next instruction. When the pipeline technique is adopted, there is no acceleration of single instruction execution, operation steps for each instruction doesn't reduce yet. While different steps of instructions executed at the same time, therefore looked from the overall it speeds up the instruction process, shortens the program execution time. In order to meet the higher clock frequency that common pipeline design can't adapt to, pipeline depth in the high end CPU gradually increases. When the pipeline depth at the 5~6 level and above, usually called super pipelining structure (Super Pipeline).Obviously, the more pipeline stages, each stage time shorter, clock cycle can be designed more short, instruction faster, instruction average execution time is short. Pipelining is by increasing the computer hardware to achieve. It requires each functional section can work independently of each other, which should increase the hardware, correspondingly increase the complexity of control. Without the operating components independent of each other, is likely to occur in various conflicts. For example, to be able to prefetch instructions, we need to increase the hardware instruction, and store the fetched instructions in the instruction queue buffer, so the microprocessor can fetch and execute instructions to operate at the same time.

b) Design of each stage of the pipeline

i. *The design of the instruction fetch stage IF*

1. Functional description

IF stage is the first stage of the pipeline, it has four main functions.

1) Automatically adds 4 to PC address according to the clock.
2) Take PC address to the instruction memory, and fetch the next instruction from the instruction memory, and pass it to the pipeline register in the next level.
3) Make decision for the instruction process flow. First, when the CPU processes according to the sequence of the instruction address, we choose the address of the next instruction as the address of the previous instruction plus 4. Second, when the CPU performs conditional branch instruction, we use mux to choose branch address. Third, when the CPU performs register branch instruction, we register branch address according to the mux. Fourth, when the CPU performs jump instruction, we use mux to choose branch address.

4) When the control hazard occurs, the CPU fetches temporary instruction and send empty instruction to the decode stage.

2. Module division

As Figure 3-1 shows, IF stage is made of two modules – program pointer register PC and program memory module.

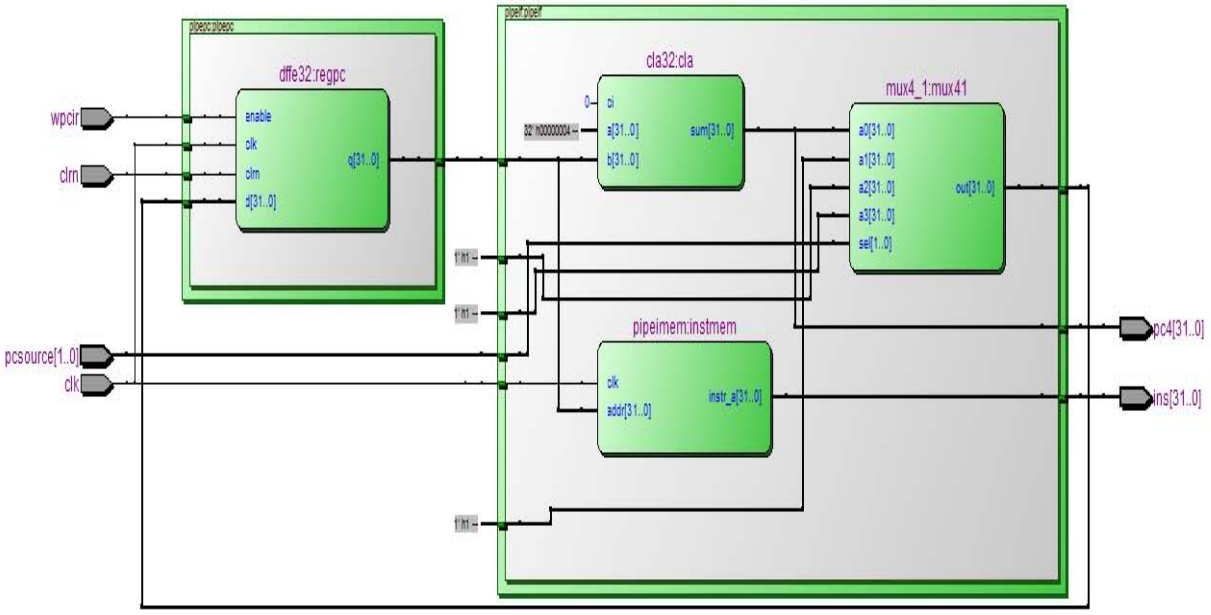


Figure 3-1 : IF stage module division

3. Logic implementation

The design uses 32 bit register with enable bit to implement program pointer register PC. The automatically adding of the address is done by a constant adder with incremental value 4. We use Altera LPM_Mem IP to implement instruction memory quickly.

ii. *The design of the instruction decode stage ID*

1. Functional description

ID stage is the second stage of the pipeline, it has three main functions.

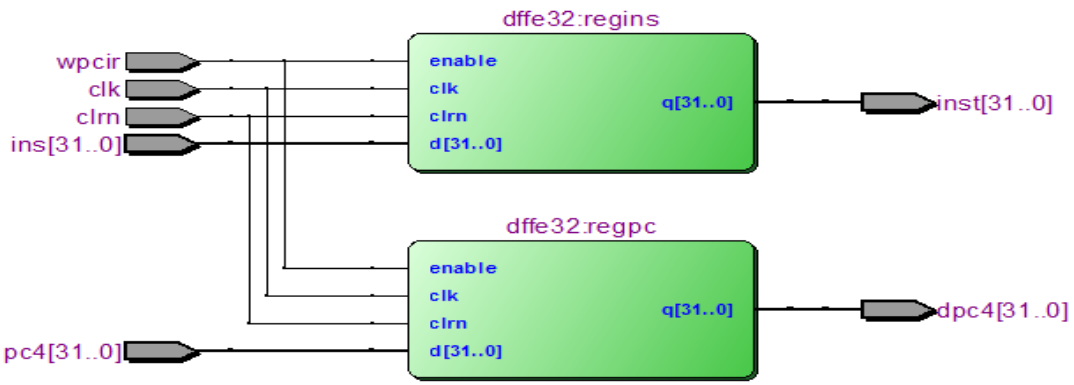
(1) Decode the instruction and control each module of the CPU according to the decoding result.

(2) Implement register file

(3) Control the pipeline process through control unit.

2. Module division

As Figure 3-2 shows, ID stage is made of three modules –pipeline register, common register file and control unit.



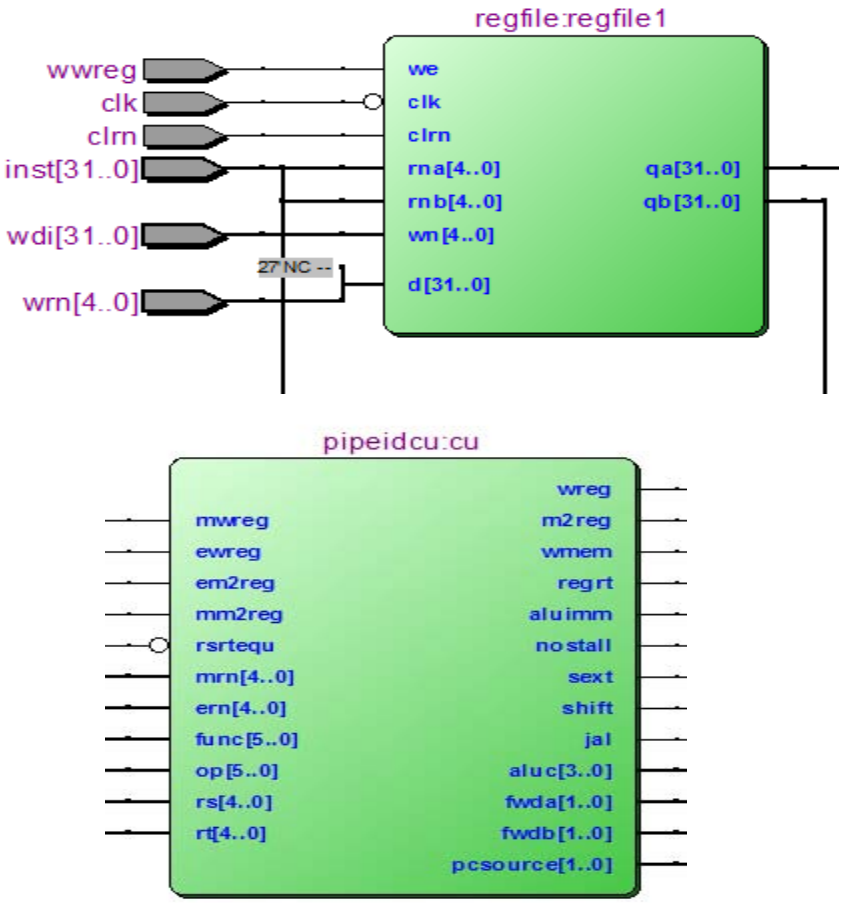


Figure 3-2 : ID stage module division

3. Logic implementation

Realization method of pipeline registers is identical with the PC in the front section, so we will not repeat them. Regfile uses multiplexer for multiple address choice. At the same time, according to MIPS architecture, we set the value of register 0 as constant 0. The control unit all uses a hard-wired logic circuits to achieve.

iii. *The design of the execution stage EXE*

1. Functional description

EXE stage is the thirdstage of the pipeline. Its main functions areto calculate the input data and other logic process according to the control signal aluc.
Control signal aluc is defined as follows

Table 3-1 : aluc control signal

aluc[3:0]	
X000	ADD
X100	SUB
X001	AND
X101	OR
X010	XOR
X110	LUI
0011	SLL
0111	SRL
1111	SRA

2. Module division

As Figure 3-3 shows, EXE stage is made of ALU and multiplexer.

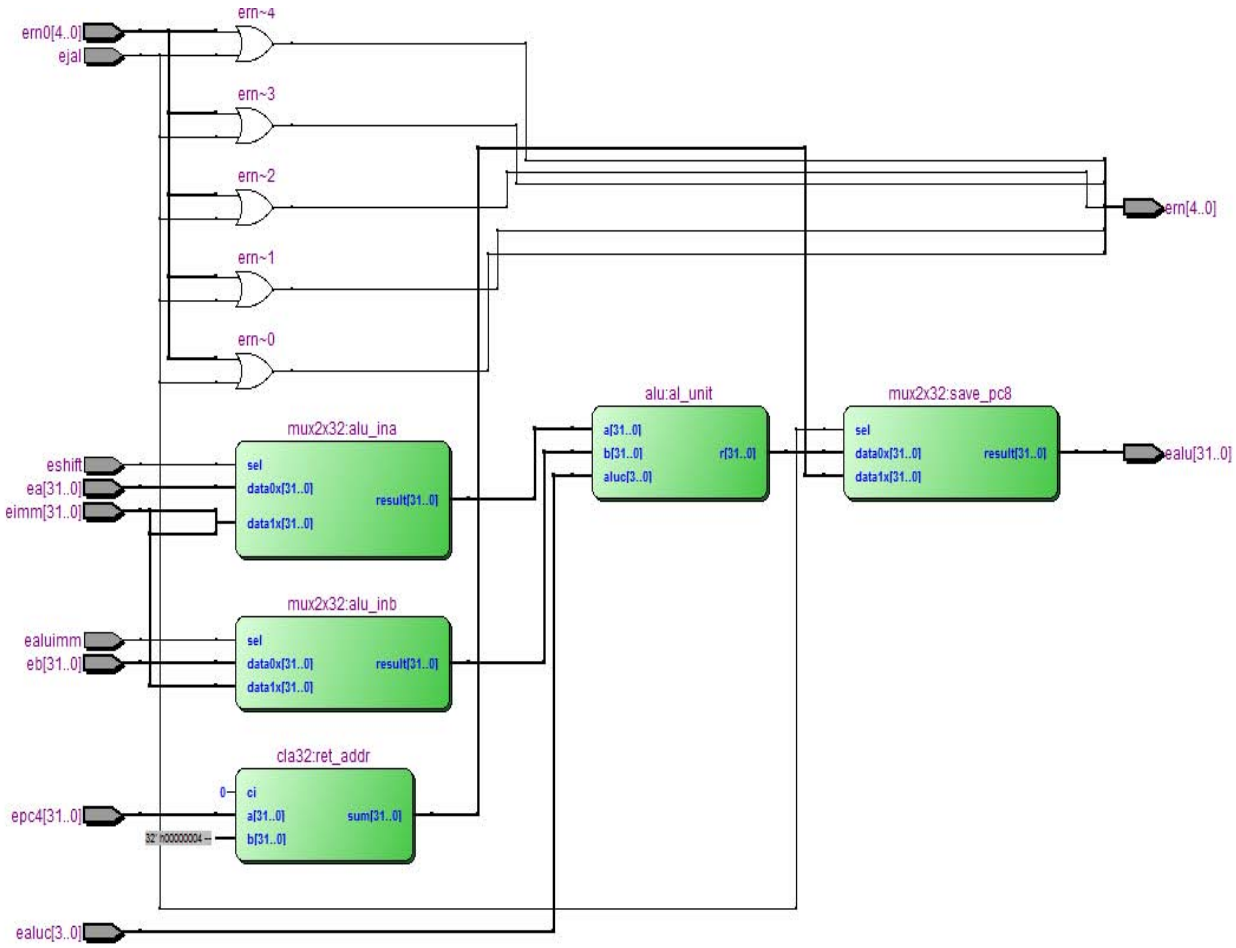


Figure 3-3 : EXE stage module division

3. Logic implementation

Because the ALU to complete a series of logic operations such as addition and subtraction shift, so we need to use special optimization algorithms and architectures, in order to realize the fast operation, and shorten the critical path delay line.

iv. *The design of the memory stage MEM*

1. Functional description

MEM stage is the fourth stage of the pipeline. Its main function is to read and write data memory.

2. Module division

As Figure 3-4 shows, MEM stage is made of a RAM module.

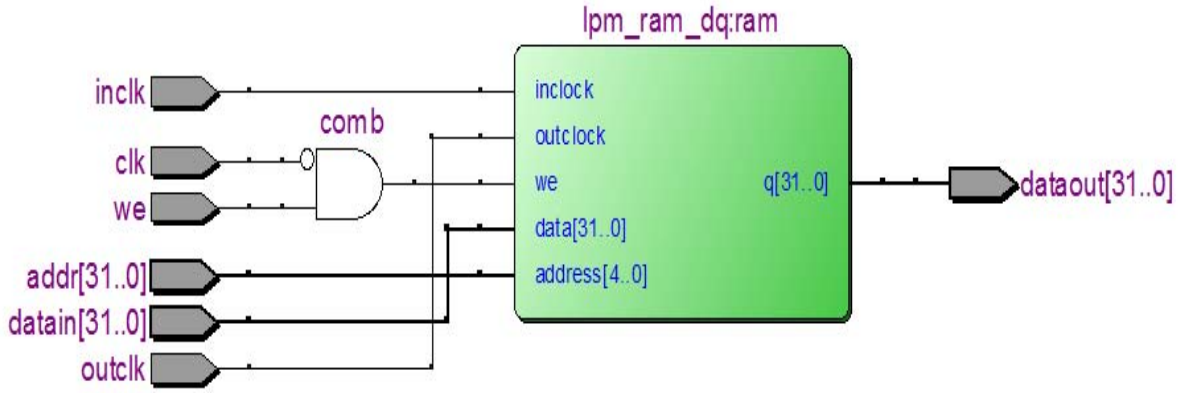


Figure 3-4 : MEM stage module division

3. Logic implementation

We use Altera's LPM_Mem Ram IP core to build the RAM module.

2. Module division

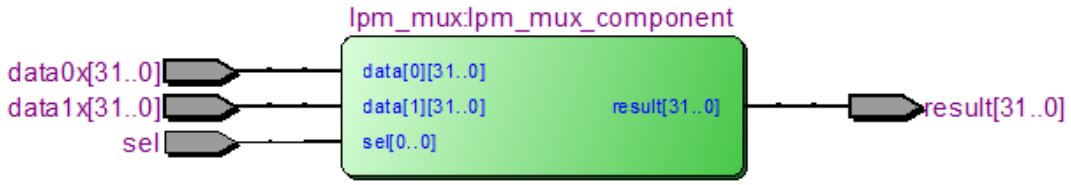


Figure 3-5 : WB stage module division

3. Logic implementation

Write back stage is only made of a multiplexer.

c) Solve the pipeline hazard

Foreword

Hazard means

In CPU design, Hazards are problems with the instruction pipeline in central processing unit (CPU) microarchitectures when the next instruction cannot execute in the following clock cycle, and can potentially lead to incorrect computation results. There are typically three types of hazards.

- data hazards
- structural hazards
- control hazards (branching hazards)

There are several methods used to deal with hazards, including pipeline stalls/pipeline bubbling, register forwarding, and in the case of out-of-order execution, the scoreboarding method and the Tomasulo algorithm.

i. *Data hazard*

Data hazard

Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions (sometimes known as race hazards). There are three situations in which a data hazard can occur:

1. Read after write (RAW),a true dependency
2. Write after read (WAR),an anti-dependency
3. Write after write (WAW), an output dependency

For simple pipeline, only RAW may result in data hazard. Other two circumstances can only occur in superscalar CPU. So we will only discuss RAW data hazard in this paper.

Solution

First we'll analyze what lead to data hazard.

v. *The design of the write back stage WB*

1. Functional description

Write back stage is the fifth stage of the pipeline. Its main function is to put the result of previous stage back to the register file.

Situation 1 data hazard occurs when the previous instruction doesn't finish, while its next instruction will use its results. For register level, see the following instruction sequence.

add r3, r1, r2
sub r4, r9, r3
or r5,r3, r9
xor r6,r3,r9
and r7, r3, r9

The first instruction put the result of the adding process into register r3.In this case, the following instructions sub, or, xor cannot take the right result in the ID stage.

There are two ways to solve this problem.

1. Stall the pipeline. Although this way can fundamentally solve data hazard, it will make pipeline stall and reduce instruction number in unit time. Therefore it is the worst.
2. Use internal forwarding. Let's look into the first and the second instruction. Because when ALU is doing subtraction, addition has been completed. Therefore we allow ALU sent the result of the addition directly to the next instruction in the ID stage to use. This method will not stall the pipeline. So it has advantages over method one.

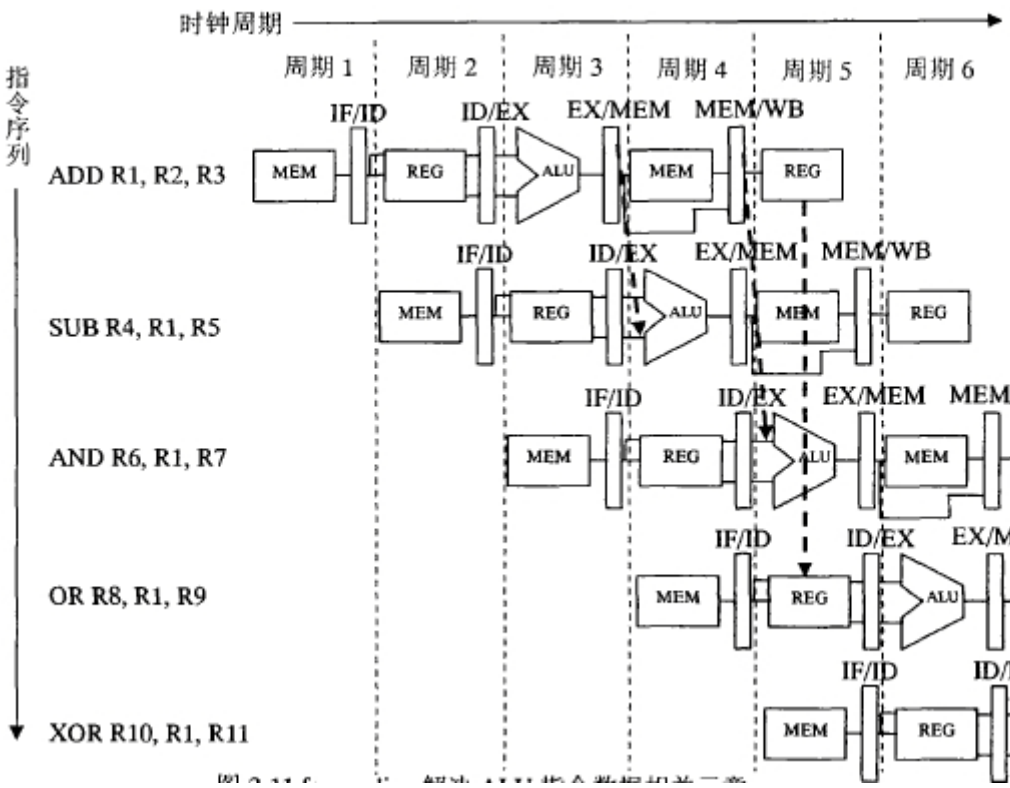


Figure 3-6 : MEM stage module division

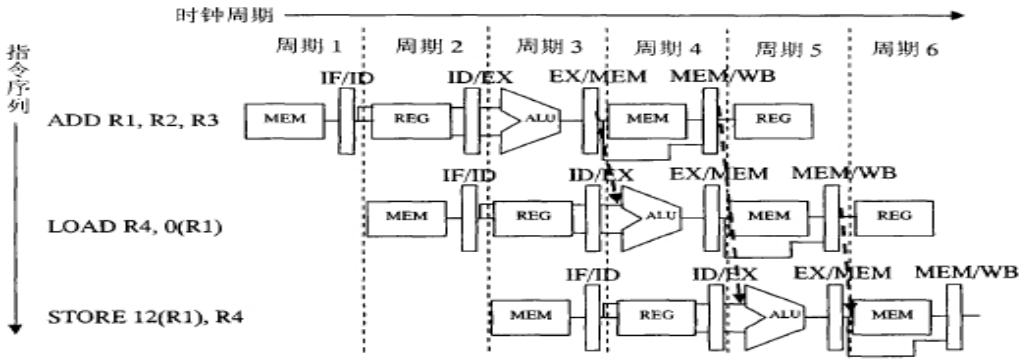


Figure 3-7 : MEM stage module division

When specific to the logic circuit design, consider by increasing the connection from the EXE stage and MEM stage to the ID stage to achieve Internal Forwarding.

Make different stages of the internal forwarding pipeline input into one MUX, and the control unit control the MUX channel selection by forwarding conditions.

The main circuit is as follow

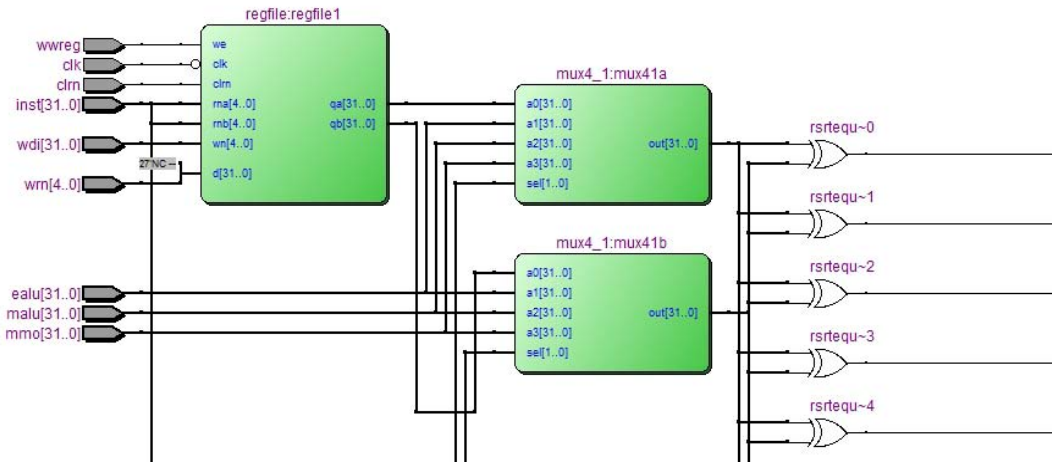


Figure 3-8 : internalforwarding

As Figure 3-6 shows, signal ealu[31:0] is the internal forwarding signal of ALU's output, malu[31:0] is the internal forwarding signal of MEM stage. mmo[31:0] is the internal forwarding signal of data cache's output. We use MUX to connect each data path.

Situation 2 Can we solve all the data hazard by internal forwarding? No. The result of ALU can push forward from EXE stage and MEM stage to ID stage, while the data that instruction lw read from data memory can only push from MEM to ID. That means if the next instruction is associated with the LW instruction, we have to stall the pipeline for one cycle–results in pipeline bubble.

See the following instruction sequence.

lw r3, 0(r1)
sub r4, r9, r3
or r5, r3, r9
xor r6, r3, r9
and r7, r3, r9

As Figure 3-9 shows, when the CPU processing the second instruction, it has to stall the pipeline for one cycle to ensure that ID stage gets the right input data, which means the CPU has to repeat the execution for one time.

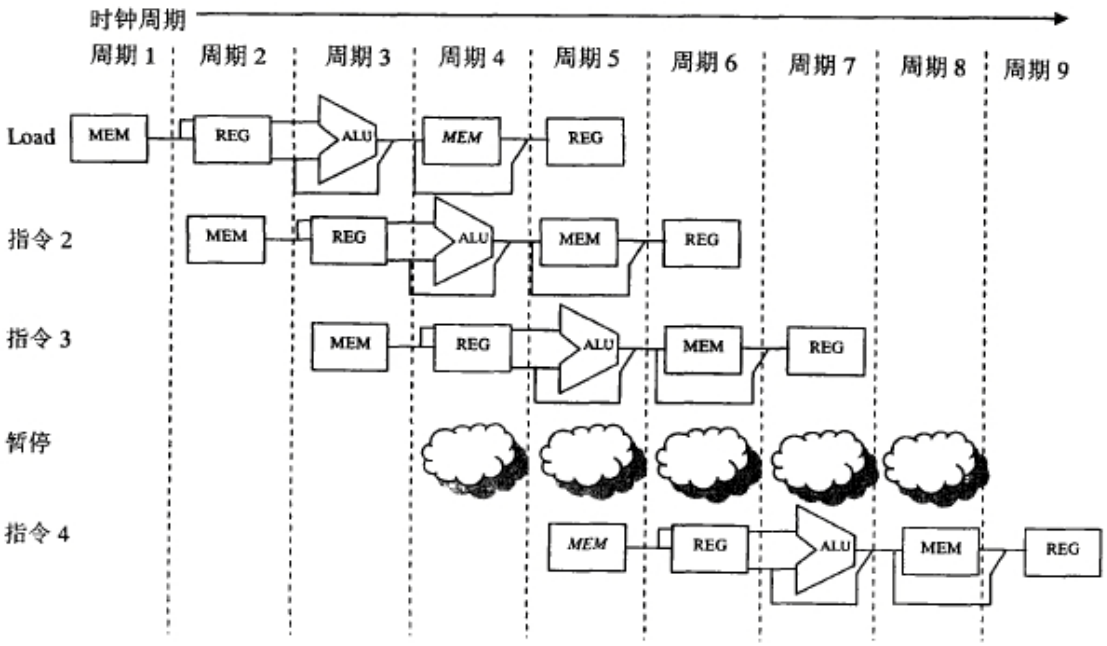


Figure 3-9 : pipeline bubble

When specific to the logic circuit design, consider by adding enable control signal to control PC and ID stage's pipeline register, thus to stall the pipeline. Corresponding to this, we can use logic statements to judge if there is data hazard. Since if we do not take other measures, stalling the pipeline will result in the re-

execution of the instruction in the IR stage, so the CPU has to discard the execution for one time. We can achieve this by banning modifying CPU state – to block the write register signal wreg and write memory signal wmem.

HDL code is as follows.

Stall = ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) | i_rt & (ern == rt)) ;

Among them, stall is the control signal to stall the pipeline when data hazard occurs. Ewreg is the signal for writing register file in the EXE stage. Em2reg signal controls the data memory to write data into register file.If the condition is true the stall signal turns high, the line suspension.

ii. *Control hazard*

Control hazard

Branching hazards (also known as control hazards) occur with branch. On many instruction pipeline microarchitectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the fetch stage).

Solution

Traditional ways

1. Delay the pipeline for two cycles. Since the address and condition for branch target are identified in EXE stage, so the next two instructions after beq have already been put into the pipeline. As Figure 3-10 shows.

周期	1	2	3	4	5	6	7	8	9
分支	IF	ID	EXE	MEM	WB				
延迟槽		IF	ID	EXE	MEM	WB			
后继指令 1			stall	stall	IF	ID	EXE	MEM	WB
后继指令 2						IF	ID	EXE	MEM

Figure 3-10 : Delay the pipeline for two cycles

2. Delay the pipeline for one cycle. If we can identify the address and condition in ID stage, then only one subsequent instruction will put into the pipeline. .As Figure 3-11 shows.

周期	1	2	3	4	5	6	7	8	9
分支	IF	ID	EXE	MEM	WB				
延迟槽		IF	ID	EXE	MEM	WB			
后继指令 1			stall	IF	ID	EXE	MEM	WB	
后继指令 2					IF	ID	EXE	MEM	WB

Figure 3-11 : pipeline bubble

Solutions for control hazard in MIPS architecture

MIPS architecture introduces Branch delay slot concept and it solves the problem of control hazard. Branch delay is an instruction after a branch instruction. No matter branch occurs or not it's always executed. Besides, the instruction in the delay slot is committed before branch instruction.

In the pipeline, branch instruction has to wait until the second stage to identify the address of next instruction. The instruction fetch stage of pipeline will not work until branch instruction is executed, therefore the pipeline has to waste (block) a time slice.

To use this time slice, we define a time slice after branch instruction as branch delay slot. In the branch delay slot the instruction is always executed, and branching occurs whether or not it doesn't matter. In this way we efficiently take advantage of a time slice, eliminating a "bubble line".

Solution methods in this paper

According to the MIPS architecture, this paper chooses a design method which using a delay slot to identify branch target address and condition in ID stage. Whether to branch or not, the one (instruction i) after branch instruction (instruction i+1) is always executed. As if it's instruction i-1. As Figure 3-12 shows.

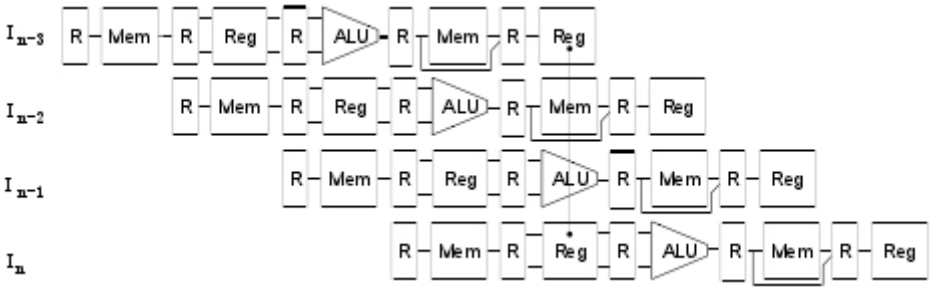


Figure 3-12 : pipeline bubble

For the pipeline CPU in this paper, there're 5 branch instructions – jr, beq, bne, j and jal. Since j, jal and jr are unconditional jump instruction, the CPU can identify branch target in ID level. For instruction beq and bne, we consider using XOR gate and NOR gate to realize these two comparisons in ID level.

iii. *Structure hazard*

Structure hazard

Structure hazard occurs when multiple instructions visit a hardware component of the processor at the same time. A typical example is an instruction fetches operands from a storage unit while the other one writes into it.

Let's take MIPS pipeline for example. For the early processors, programs and data memory are not separated, as Figure 3-13 shows, there're memory access at the same time in IF and MEM stage. This results in that one of the accesses has to wait for a cycle. For modern processors, the program is stored in L1P Cache and the data is stored in L1D Cache. They are accessed separately so structure hazard is not a problem.

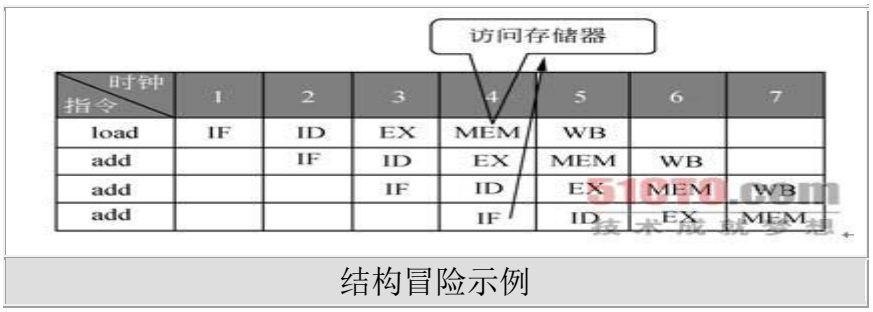


Figure 3-13 : structure hazards

Solution methods in this paper

In this paper, since we use separated I-cache and D-cache, structure hazard is avoided.

IV. Design of the Interrupt and Exceptional Circuit

In this chapter we introduce design of the Interrupt and Exceptional Circuit. First we introduce the concept of interrupt, exception and precise interrupt. And then we introduce the hardware interrupt processing structure with MIPS architecture and the related interrupt exception handling instruction set. Finally, we discuss the pipelined CPU terminal and exception handling circuit realization. Meanwhile, we will provide the RTL diagram and related codes.

a) The MIPS exception and interrupt handling principle

i. *Exception, interrupt and precise exception*

In systems programming, an interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing (the current thread). The processor responds by suspending its current activities, saving its state, and executing a small program called an interrupt handler (or interrupt service routine, ISR) to deal with the event. This interruption is temporary, and after the interrupt handler finishes, the processor resumes execution of the previous thread.

A hardware interrupt is an electronic alerting signal sent to the processor from an external device, either a part of the computer itself such as a disk controller or an external peripheral. For example, pressing a key on the keyboard or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position. Unlike the software type (below), hardware interrupts are asynchronous and can occur in the middle of instruction execution, requiring additional care in programming. The act of initiating a hardware interrupt is referred to as an interrupt request (IRQ).

A software interrupt is caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed. The former is often called a trap or exception and is used for errors or events occurring during program execution that are exceptional enough that they cannot be handled within the program itself. For example, if the processor's arithmetic logic unit is commanded to divide a number by zero, this impossible demand will cause a divide-by-zero exception, perhaps causing the computer to abandon the calculation or display an error message. Software interrupt instructions function similarly to subroutine calls and are used for a variety of purposes, such as to request services from low level system software such as device drivers. For example, computers often use software interrupt instructions to communicate with the disk controller to request data be read or written to the disk.

ii. *Exception and interrupt handling in MIPS*

Interrupts

The processor supports eight interrupt requests, broken down into four categories:

- Software interrupts - Two software interrupt requests are made via software writes to bits IP0 and IP1 of the Cause register.
- Hardware interrupts - Up to six hardware interrupt requests numbered 0 through 5 are made via implementation-dependent external requests to the processor.
- Timer interrupt - A timer interrupt is raised when the Count and Compare registers reach the same value.
- Performance counter interrupt - A performance counter interrupt is raised when the most significant bit of the counter is a one, and the interrupt is enabled by the IE bit in the performance counter control register.

Timer interrupts, performance counter interrupts, and hardware interrupt 5 are combined in an implementation dependent way to create the ultimate hardware interrupt 5.

Exceptions

Normal execution of instructions may be interrupted when an exception occurs. Such events can be generated as a by-product of instruction execution (e.g., an integer overflow caused by an add instruction or a TLB miss caused by a load instruction), or by an event not directly related to instruction execution (e.g., an external interrupt). When an exception occurs, the processor stops processing instructions, saves sufficient state to resume the interrupted instruction stream, enters Kernel Mode, and starts a software exception handler. The saved state and the address of the software exception handler are a function of both the type of exception, and the current state of the processor.

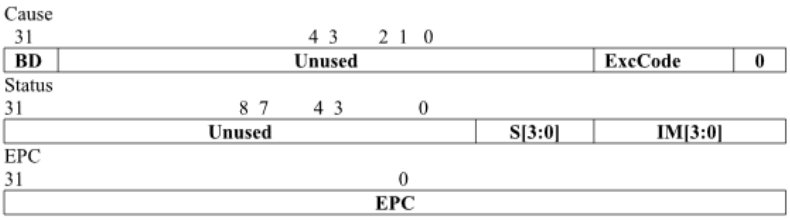
b) Pipeline CPU precise exception and interrupt processing circuit

The complexity of pipelined CPU exception and interrupt handling is mainly reflected in two respects. (1) Pipeline CPU has multiple instructions simultaneously in operation. There is not a time point that all the instructions are executed. (2) MIPS pipeline allows the branch delay. If exception and interrupt occurs in the delay slot of ID stage, then the return address will not be judged. Therefore, to achieve precise exception and interrupt handling, we must carefully study the characteristics of CPU and design of hardware.

i. *Types of exception and interrupt and associated registers*

The registers for pipeline CPU exceptions and interrupts are shown as below. The sixth to second bit of the cause register is the codes that generate exception and interrupt. IM[3:0] in the status register is a 4 bit mask. Each corresponding to an exception or interrupt mask bit, 1 allows the exception or interrupt and 0 bans it. S[3:0] is IM[3:0] which is left shifted by 4 bits. EPC is used for saving the return address. If the instruction that causes exception is in the delay slot of branch or jump instruction, then the BD bit is set to 1. Under normal circumstances we set BD to 0.

Table 4-1 : MIPS exception and interrupt registers



The following table lists the exceptions and interrupts may appear which levelin the pipeline.

Table 4-2 : MIPS exception and interrupt

ExcCode	Alias	Type	Mask	Description	Stage
0	Int	Int	IM[0]	External Interrupt	Any stage
1	Sys	Except	IM[1]	Syscall	ID stage
2	Unimpl	Except	IM[2]	Non-exist instruction	ID stage
3	Ov	Except	IM[3]	Overflow	EXE stage

When interrupt occurs EPC saves the return address, and when exception occurs EPC saves the address that causes exception. However, if the abnormal instruction is in delay slots, EPC save the delayed branch instruction address. Therefore, we must have some means to determine whether an instruction in the delay slot. When an exception or interrupt occurs, we need to scrap the subsequent instruction and even the current instruction.

ii. *Interrupt response process of the pipeline CPU*

In order to realize the precise interrupt, we divide the interrupt request into the following 3 kinds of circumstances.

(1) Interrupt occurs while ID's executing the transfer instruction

The design idea is to put the address of branch instruction into EPC register, and the base address of interrupt execution into PC. When returns from interrupt execution, the CPU re-execute branch instruction.

As Figure 4-1 shows, if the address of branch instruction is PC in the IF level, then in the ID level is PCD. When ID finish, the CPU write PCD into EPC, and put the entrance address of exception and interrupt into PC. Meanwhile, the cancel signal generated at the ID level is also written to the pipeline register, its output at EXE stage is e_cancel, to abolish the next instruction. When it returns from interrupt, directly write the value of EPC into PC, and re-execute the branch instruction.

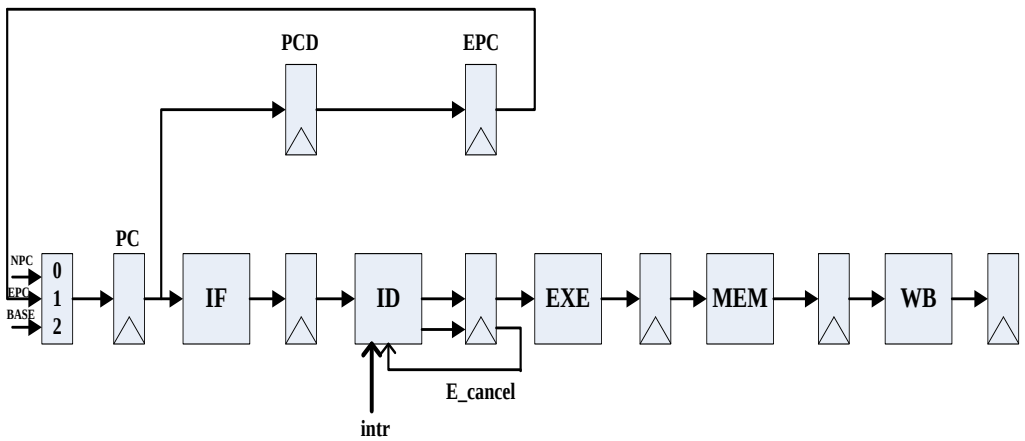


Figure 4-1 : Interrupt occurs while ID's executing the transfer instruction

(2) Interrupt occurs while ID's the delay slot

The design idea is to let the instruction in the delay slot be finished. Because the branch address is decoded in ID stage, so we put the address value of PC into EPC. Meanwhile the BD bit in the Cause register should be set to 1.

As Figure 4-2 shows, At the ID level delay slot instruction at the end, the branch target address instruction has been taken into the pipeline, we disable it using e_cancel.

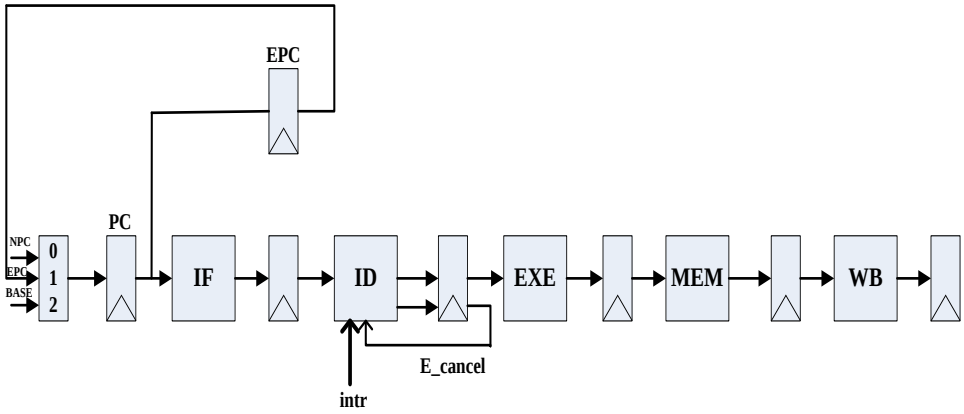


Figure 4-2 : Interrupt occurs while ID's the delay slot

(3) Interrupt occurs in general situation

The design idea in this circumstance is to response to the interrupt in ID stage, to abolish the next instruction, and write the address of the next instruction into EPC. In this case we don't need to set the BD to 1. The pipeline design in this case is identical to the second situation, so It is unnecessary to go into details here.

iii. *Precise interrupt of the pipeline CPU*

The pipeline CPU puts the address that causes exception into EPC. If this instruction is in the delay slot, then put its previous branch or jump address into EPC and set the BD as 1. The following descripts in detail the methods to handle pipeline exception.

(1) Syscall

No matter using assembly language or high level language to program, we can let the syscall instruction not to appear in the delay slot. Therefore, we consider only usually system call instructions implementation. Figure 4-3 shows the pipeline progress that CPU executing syscall instruction. Jump to exception and interrupt handling program and abolish its next instructions. EPC saves PCD -the address of syscall instruction. As the figure shows the input of the EPC connects to a mux, when it modifies EPC it chooses DATA.

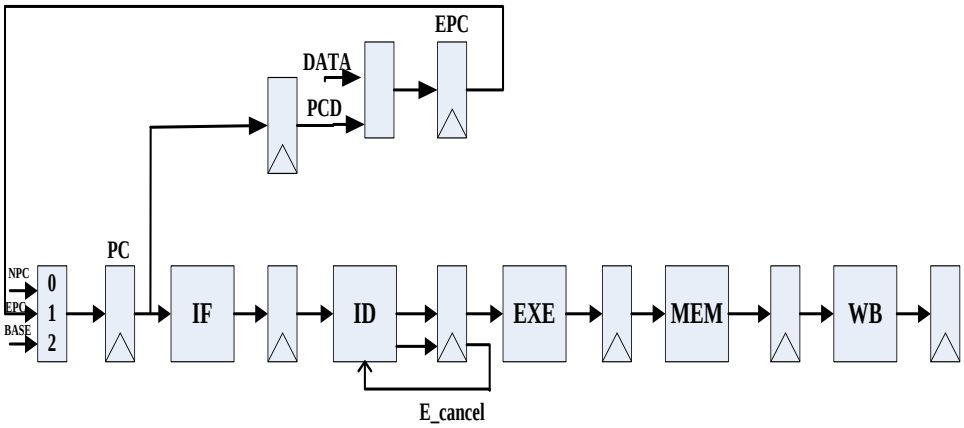


Figure 4-3 : Interrupt occurs in general situation

(2) Unimplemented instruction

The following picture shows the pipeline that CPU's executing unimplemented instructions in the delay slot. In this case EPC will save the address of its previous instruction PCE, the BD bit of Cause register should be set to 1.

Figure 4-4 shows the pipeline progress that CPU executing unimplemented instruction. It is similar with the implementation of the syscall instruction.

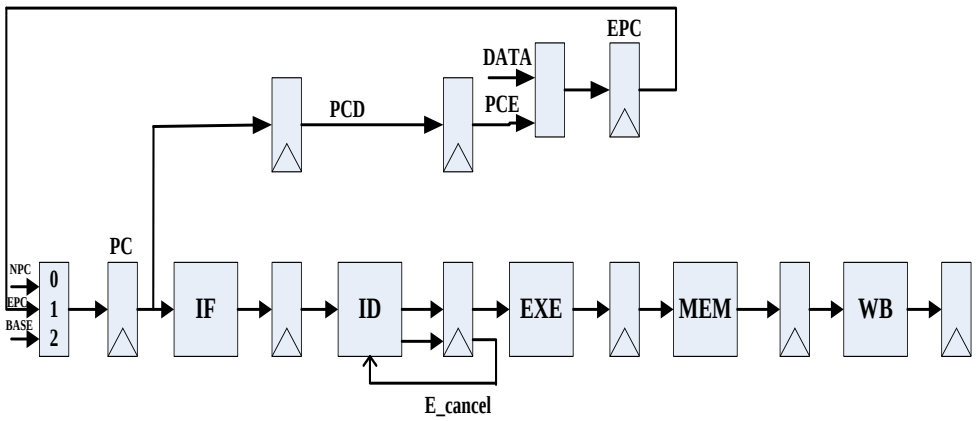


Figure 4-4 : Interrupt occurs in general situation

(3) Calculation result overflow

Calculation result overflow appears in EXE stage, and the result of overflow cannot be saved into register file. So the CPU has to block the wreg signal in the EXE stage. Besides, instructions in ID stage should be abolish too.

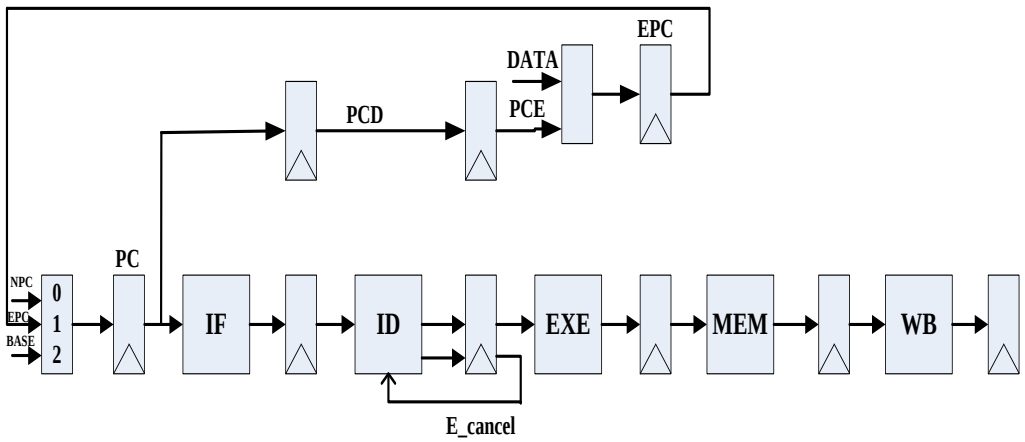


Figure 4-5 : Interrupt occurs in general situation

Figure 4-4 is the pipeline that when CPU is in the delay slot overflow. At this time EPC has to save the PCM address of its previous instruction. The BD bit of cause register should be set to 1.

Figure 4-5 is the pipeline that overflows occurs in normal situation. EPC saves the PCE address of overflow instruction.

V. CPU Verification

Figure 5-1 shows the typical FPGA design and verification overflow. After the design personnel will be HDL code input and comprehensive utilization comprehensive tool, will carry on the first simulation of design: functional simulation. Functional simulation is not with circuit delay parameters, only validation logic function is correct. If the function simulation, then the layout of design, after second times simulation: timing simulation is also the gate level simulation. With Gate level simulation with circuit delay parameters, the result is more accurate, more close to the actual device performance. After passing through the gate level simulation, finally the design is downloaded to the FPGA device.

This paper will adopt the design process, optimizing the design details, in order to achieve high quality and efficient design objective.

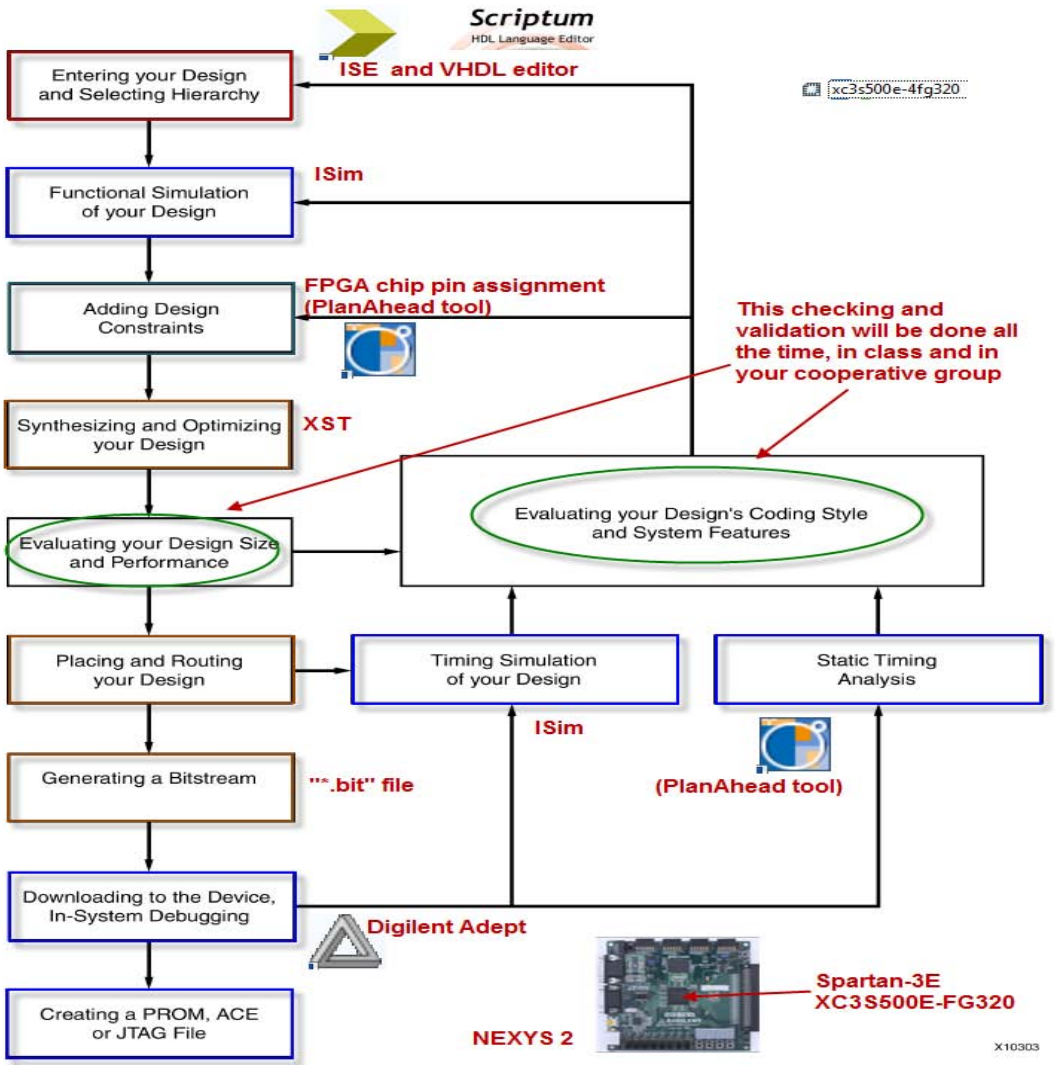


Figure 5-1 : Typical FPGA design & verification overflow

a) Pipeline verification

We implement the idea of bottom-up, hierarchical verification. Firstly, we run functional simulation for single module. If the circuit can realize logic function then we gate-level simulation for every single module. When all the sub-module pass verification, we run simulation for each stage of the pipeline and finally the whole pipeline circuit.

- IF stage

IF stage has two functions: (1) calculate the instruction address (2) fetch instruction and pipeline process. As Figure 5-2 shows, when reset signal clrn is low, register PC is set to initial value 0. Signal PC4 is the address value plus 4, at this time the enable signal of register PC is low, and stage IF doesn't work yet. When reset signal and enable signal turns high, register PC starts to work. We can see that PC of continues adding 4 with the change of clk signal, which proves that stage IF can finish address calculating. Meanwhile, instruction signal ins changes with the change of PC4 signal, and the output corresponding with ROM address of the instruction value, proving that IF stage can complete instruction fetching and pipeline process.

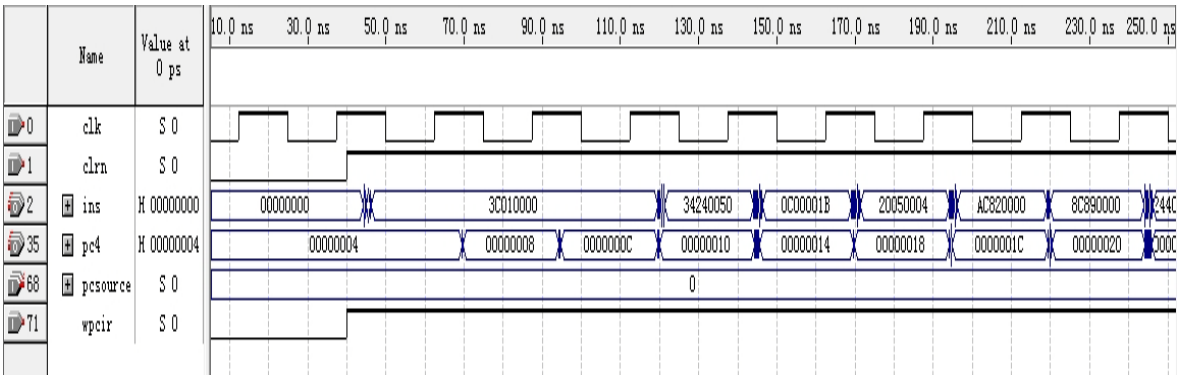


Figure 5-2 : IF stage verification

- ID stage

ID stage has three functions: (1) decode the instruction fetched from instruction cache (2) put the corresponding control word, immediate word and address value to control unit and register file (3) control unit output corresponding signals according to the input logic. In this chapter we will verify the regfile and control unit.

As Figure 5-3 shows, it's each signal of register file and their simulation waves. Clrn is reset signal. d is input data value. qa is the output value of register file output a. qb is the output value of register file output b. ma is qa's output address, which means corresponding register number. We is write-enable signal, wn is write register number.

When clrn is low, register file is reset, and all the value of registers are set to 0. When clrn turns to high, register file starts to work. When write signal we turns high, wn controls the write register address. As Figure 5-3 shows, the CPU writes value into 0 to 4 register. According to MIPS architecture, 0 register cannot be modified, thus the output of 0 register is always 0. For register 1 to 3, the output is the input value when we is high. So this verifies that register file can work normally and realize the logic function.

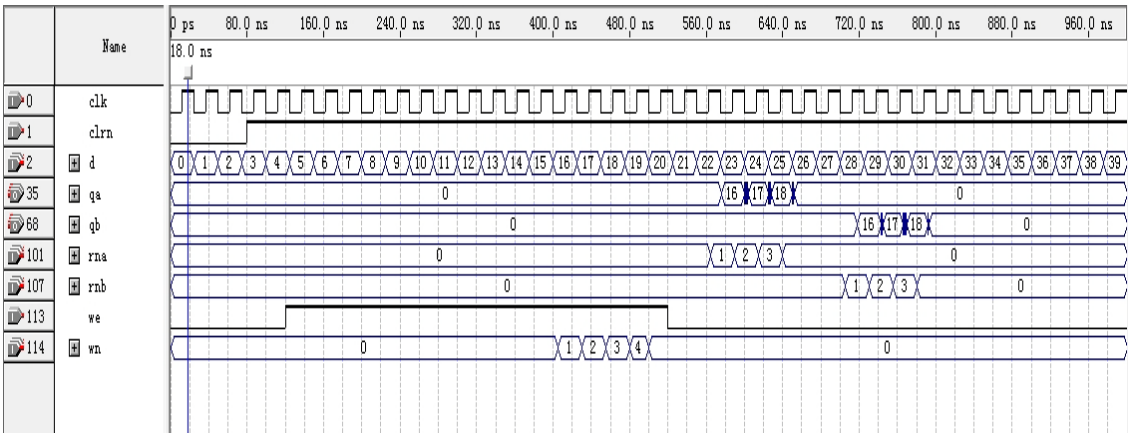


Figure 5-3 : regfileverification

Each signal of control unit and corresponding simulation waves are shown in Figure 5-4. Since control unit is pure logic circuit, we can see apparent glitch in the output wave of the circuit. The main signals are as follows: aluc controls the output of signal alu. Op is instruction word, representing instruction code. Pcsource controls the mux for address source of pc. We can see from the simulation waves that the control unit can output correct control logic. Further verification will be shown in the later chapter.

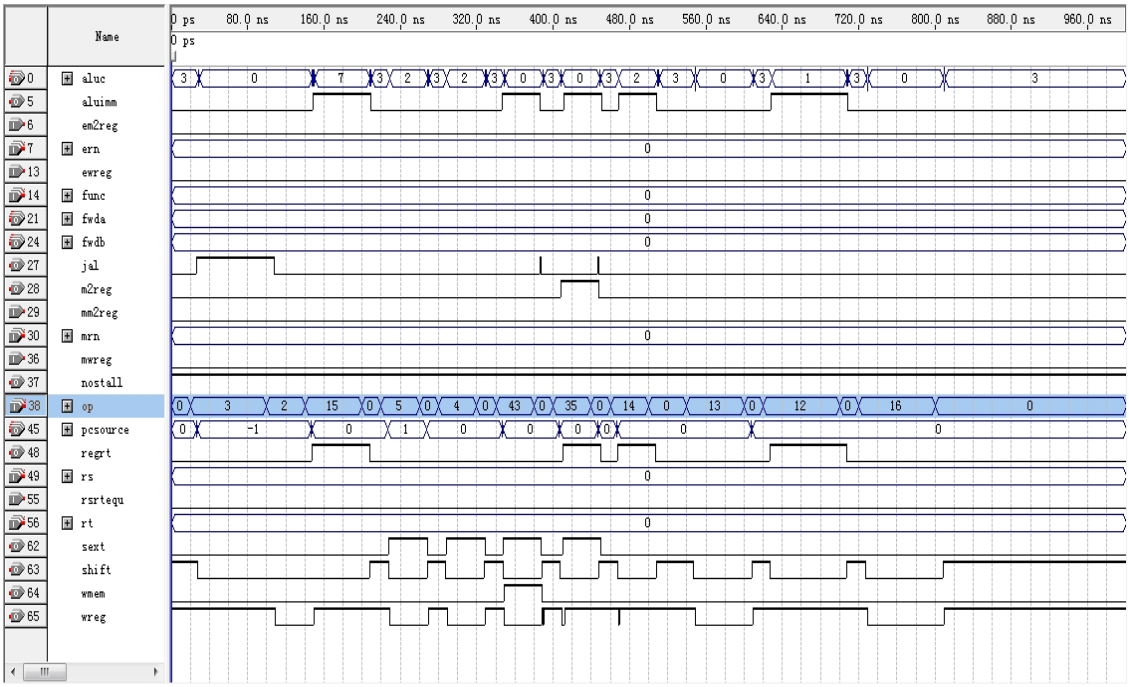


Figure 5-4 : Control Unit verification

- EXE stage

The function of EXE stage is to calculate the value from ID and put the result to the next stage. According to the control signal passed from ID stage, EXE will finish corresponding calculation in the pipeline clock cycle. Since arithmetic unit may cause long delay, so EXE stage is the critical path of the pipeline. We will verify ALU in this chapter.

As Figure 5-5 shows, it's each signal of ALU and their simulation waves. The definition of the control signals of ALU sees the 3.2.3 section. When ALUC=0 the two input numbers are added, the result is correct. When ALUC=4 the two input numbers are subtracted, the result is correct. The other functions of ALU are verified as above.

We can see from the waveform that, because the signal judging overflow and zero is used in combinational logic circuit, so it will easily produce burr.

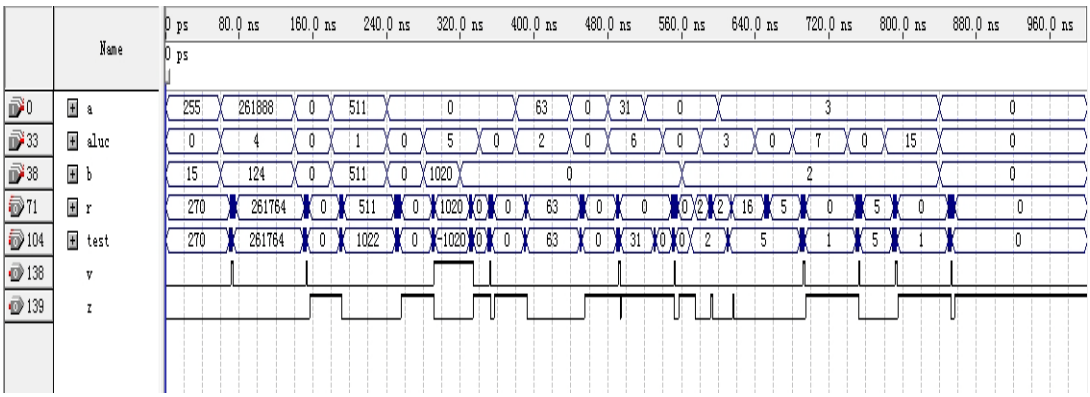


Figure 5-5 : ALU verification

- MEM stage

The function of MEM stage is to load and store data. The main module in MEM stage is data memory. In this paper we use the LPM Ram provided by Altera to achieve this.

As Figure 5-6 shows, it's the simulation wave for data memory. Addr controls the RAM address of data input and output, and we is write enable signal. We can see from the waveform that, when signal we is low, dataout reads data from RAM according to the value of addr. When signal we turns high, dataout writes data into RAM according to the value of addr too. For data 0x0000007F, we can see that it's written into corresponding address, and is read from it. The simulation waves prove that the data memory works correctly.

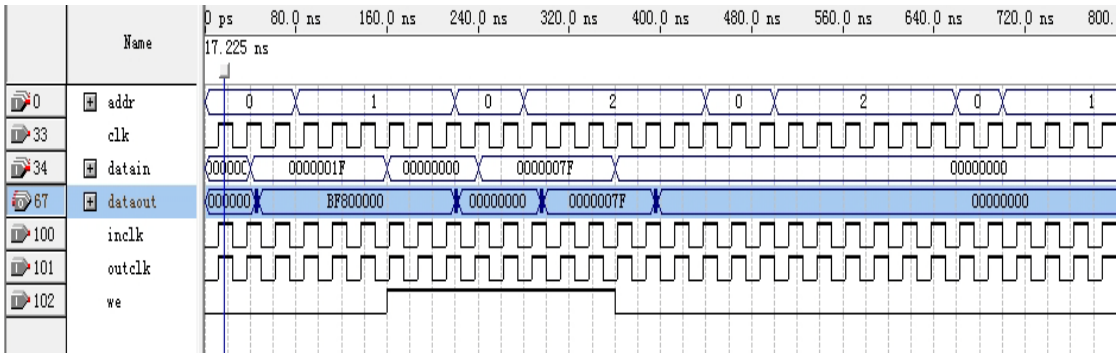


Figure 5-6 : Memory module verification

- WB stage

The function of MEM stage is to store the result calculated from EXE or data in MEM stage. Since there are many data paths, we use mux to achieve that. Because the circuit of this stage is simple, it's unnecessary to run simulation specifically in this chapter.

- pipeline integral verification

x. *Verification program*

See appendix A.

xi. *Verification waves*

As Figure 5-7 shows, clock is system clock. memclock is ram's clock. Signal inst is the corresponding instruction in a clock cycle. Ealu, malu, walu is the pipeline register value of the output of alu. we can see that corresponding instructions are put into the pipeline with the change of PC. And the output of Alu value has been sent to the next level with the pipeline. The simulation waves prove that the pipeline works correctly.

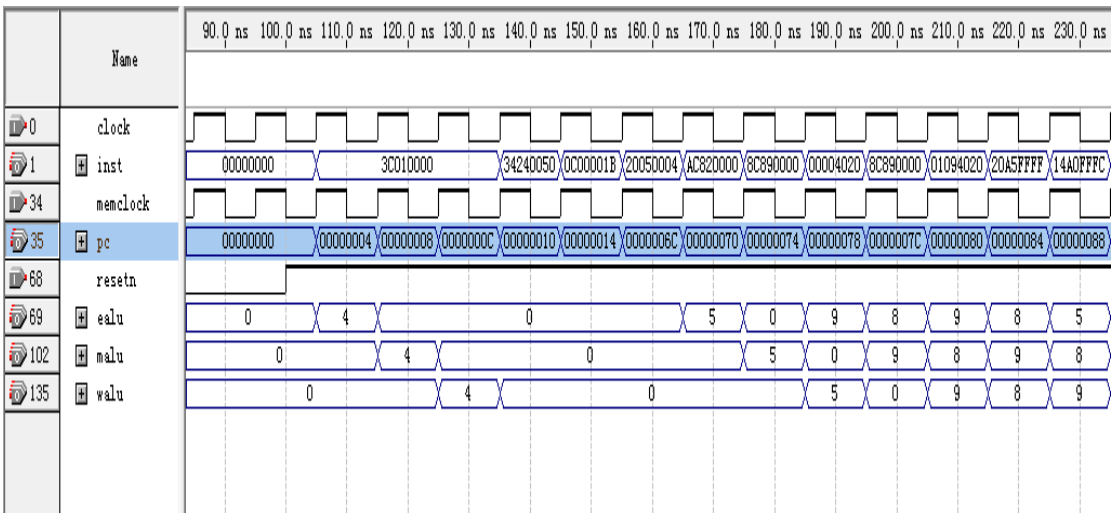


Figure 5-7 : Pipeline integral verification

b) Interrupt and exception circuit verification

i. *Verification program*

See appendix B.

ii. *Verification waves*

As Figure 5-8 shows, clock is system clock. memclock is ram's clock. Signal inst is the corresponding instruction in a clock cycle. Ealu, malu, walu is the pipeline register value of the output of alu. The output of Alu value has been sent to the next level with the pipeline. The simulation waves prove that the Interrupt and exception circuit works correctly.

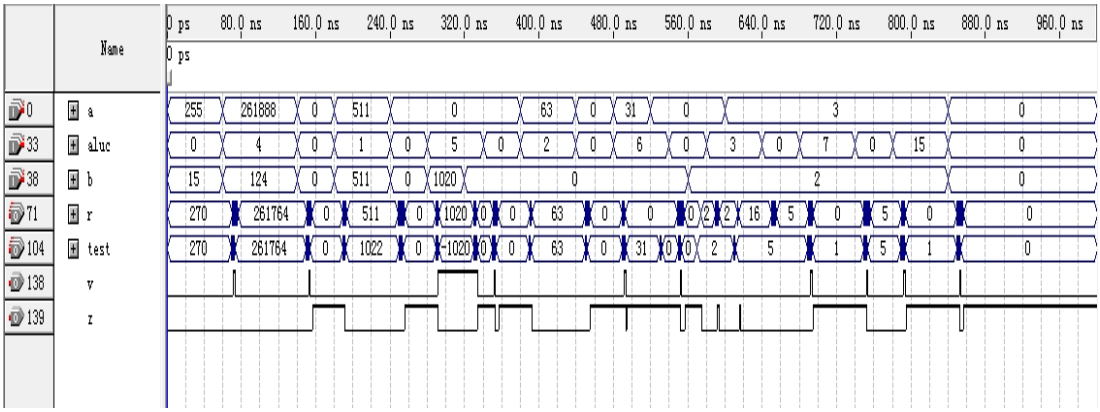


Figure 5-8 : Interrupt and exception circuit verification

VI. Summary and Future Work

Through this thesis, Verilog HDL code for a Altera cycloe4 FPGA board was developed, on which a pipeline CPU runs. In chapter 1 we make a brief introduction of the research background. It mainly introduces the background and related research status and CPU's integrated circuit industry. In chapter 2 we describe the development platform and MIPS architecture. There we also introduce the software and hardware development platform for the project and FPGA's design. At the same time, we describe the registers and instructions in our design and MIPS architecture. In chapter 3 we firstly discuss the design of pipeline data path. And we work out some methods for solving pipeline hazards. In Chapter three we describe the design of pipeline data path. It introduces the pipeline design method, the composition of the pipeline and design and verification of associated component. After that, we make the interrupt circuit and whole verification.

In the future, there are two ways to improve this simple CPU. Firstly we will add some modules to the original design, including timer, bus, and the whole CP0 coprocessor. Secondly we will improve the circuit thus to make the whole circuit run in a higher frequency. Generally speaking, more performance analysis such as studying the impact of exceptions on the core performance can be done further to see some interesting and important results.

References Références Referencias

1. N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, "FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores Within a Canonical Superscalar Template", Proceedings of the 38th Annual International Symposium on Computer Architecture. ACM, 2011, pp. 11. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000067
2. Naoki Fujieda, Takefumi Miyoshi, and Kenji Kise "SimMips A MIPS System Simulator".
3. MIPS® Architecture For Programmers Volume III-A: Introduction to the MIPS32® Architecture.
4. MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture.
5. MIPS® Architecture For Programmers Volume II-A: Introduction to the MIPS32® Architecture.
6. Linux Porting Guide. [Online]. Available: http://www.embedded.com/design/embedded/4023297/Linux-Porting-Guide
7. Simple Scalar Simulator Toolset. [Online]. Available: http://www.simplescalar.com/
8. M. A. Khalighi, N. Schwartz, N. Aitamer, and S. Bourennane, "Fading reduction by aperture averaging and spatial diversity in optical wireless systems," *Journal of Optical Communications and Networking, IEEE/OSA* vol. 1, pp. 580-593, 2009.
9. A. O. Aladeloba, A. J. Phillips, and M. S. Woolfson, "Performance evaluation of optically preamplified digital pulse position modulation turbulent free-space optical communication systems," *IET Optoelectronics,* vol. 6, pp. 66-74, February 2012.
10. L. C. Andrews, R. L. Phillips, and C. Y. Hopen, "Aperture averaging of optical scintillations: power fluctuations and the temporal spectrum," *Waves Random Media,* vol. 10, pp. 53–70, 2000.
11. S. Bloom, E. Korevaar, J. Schuster, and H. A. Willebrand, "Understanding the performance of free-space optics," *Journal of Optical Networking,* vol. 2, pp. 178-200, June 2003.
12. D. K. Borah and D. G. Voelz, "Pointing error effects on free-space optical communication links in the presence of atmospheric turbulence," *Journal of Lightwave Technology,* vol. 27, pp. 3965-3973, 2009.
13. A. A. Farid and S. Hranilovic, "Outage capacity optimization for free-space optical links with pointing errors," *Journal of Lightwave Technology,* vol. 25, pp. 1702-1710, 2007.
14. H. G. Sandalidis, T. A. Tsiftsis, G. K. Karagiannidis, and M. Uysal, "BER performance of FSO links over

strong atmospheric turbulence channels with pointing errors," *IEEE Communications Letters,* vol. 12, pp. 44-46, 2008.
15. A. J. Phillips, "Power penalty for burst mode reception in the presence of interchannel crosstalk," *IET Optoelectronics,* vol. 1, pp. 127-134, 2007.
16. K. W. Cattermole and J. J. O'Reilly, *Mathematical topics in telecommunications volume 2: problems of randomness in communication engineering*, Pentech Press Limited, Plymouth, 1984.
17. I. T. Monroy and E. Tangdiongga, *Crosstalk in WDM communication networks*, Kluwer Academic Publishers, Norwell, Massachusetts, USA, 2002.
18. J. O'Reilly and J. R. F. Da Rocha, "Improved error probability evaluation methods for direct detection optical communication systems," *IEEE Transactions on Information Theory,* vol. 33, pp. 839-848, 1987.
19. L. F. B. Ribeiro, J. R. F. Da Rocha, and J. L. Pinto, "Performance evaluation of EDFA preamplified receivers taking into account intersymbol interference," *Journal*
20. Mishra Prabhat, DuttNikil, Nicolau Alex. Specification of Hazards, Stalls, Interrupts, and Exceptions in Expression, Technical Report #01-05, Dept. of Information and Computer Science, University of California, Irvine, CA 92697, 2001,USA
21. Smith James.E., Plezskun Andrew R, Implementing Precise Interrupts in Pipelined Processors, IEEE Transactions on Computers. 1988, 37(5), pp. 562-573
22. Wang Chia-Jiu, Emnett Frank. Implementing Precise Interruptions in Pipelined RISC Processors. IEEE, Micro. 1993,13(4) , pp. 36-43
23. KE Xi-ming. Implementation Mechanism of Precise Interrupts in Microprocessors, High Performance Computing Technology. 2003, 160, pp. 45～48
24. XI Chen, ZHANG Sheng-bing, SHEN Xu-bang, et al. New precise interrupt mechanism based on backup- buffer. Computer Engineering and Applications. 2007,43(6) , pp. 95- 98.
25. Liu Shibin, GaoDeyuan, Fan Xiaoya, et al. Design of Instruction Decoder for Use in China for an Embedded MPU, Journal of Northwestern Polytechnic University. 2001,19(1) , pp. 1-5
26. D. W. Anderson, F. 1. Sparacio, and F. M. Tomasulo, 'The IBM systeml360 Model 91 : Machine philosophy and instruction handling, "IBM 1. Res. Develop., vol. 11, pp. 8-24, Jan. 1967.
27. Ozer, E.; Sathaye, S.W.; Menezes, K.N.; Banerjia, S.; Jennings, M.D.; Conte, T.M.; "A fast interrupt handling scheme for VLIW processors", Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on Digital Object Identifier: IO.l109IPACT.l998.727184 Publication Year: 1998 , Page(s): 136 - 141
28. 1. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," IEEE Trans. Comput., vol. C-37, no. 5, pp. 562-573,May 1988.
29. W-M.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines," IEEE Trans. Computers, Vol. C-36, No. l2,Dec. 1987, pp. 1,515-1,522.
30. Pericas, M., Cristal, A., Gonzalez, R., Jimenez, D.A., Valero, M., "A decoupled KILO-instruction processor", High-Performance Computer Architecture, 2006. The Twelfth International Symposium on, On page(s)
31. Dominic Sweetman, See MIPS Run, Academic Press, 2002.
32. David A Patterson, John L Hennessy, Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann Publishers, Inc. 1998
33. Stephen Brown, ZvonkoVranesic. Fundamentals of Digital Logic with VHDL Design, McGraw-Hill, 2000.
34. Zhu Ziyu, Li Yamin. CPU Chip Logic Design. Tsinghua University Press. 2005
35. "Altera university program—Learning through innovation," Altera Corporation, San Jose, CA, 2011 [Online]. Available: http://www.altera.com/education/univ/unv-index.html
36. A. Clements, "The undergraduate curriculum in computer architecture," IEEE Micro, vol. 20, no. 3, pp. 13–21, May–Jun. 2000.
37. J. Djordjevic, B. Nikolic, T. Borozan, and A. Milenkovie, "CAL2: Computer aided learning in computer architecture laboratory," Comput. Appl. Eng. Educ., vol. 16, pp. 172–188, 2008.
38. B. Nikolic, Z. Radivojevic, J. Djordjevic, and V. Milutinovic, "A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization," IEEE Trans. Educ., vol. 52, no. 4, pp. 449–458, Nov. 2009.
39. H. Oztekin, F. Temurtas, and A. Gulbag, "BZK.SAU: Implementing a hardware and software-based computer architecture simulator for educational purpose," in Proc. 2nd Int. Conf. Comput. Design Appl., 2010, pp. 490–497.
40. V. Gustin and P. Bulic, "Learning computer architecture concepts with the FPGA-based "Move" microprocessor," Comput. Appl. Eng. Educ., vol. 14, pp. 135–141, 2006.

VII. Acknowledgements

Foremost, I would like to express my sincere gratitude to my Supervisor Prof.Liu for the continuous support of my thesis work, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me to perform my thesis work and in writing this thesis report.

I also would like to say a big thank you to all my colleagues, and the researchers in the Computer Engineering Center at the Guangdong University of

Technology, China, for their support and for making my time in Guangzhou an enjoyable one.

Most importantly, I would like to thank my wife Fathai and my children - for their wonderful support throughout my study. This dissertation is dedicated to them as a token of my gratitude.

学位论文独创性声明

本人郑重声明：所呈交的学位论文是我个人在导师的指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明，并表示了谢意。本人依法享有和承担由此论文所产生的权利和责任。

论文作者签名：日期：

学位论文版权使用授权声明

本学位论文作者完全了解学校有关保存、使用学位论文的规定，同意授权广东工业大学保留并向国家有关部门或机构送交该论文的印刷本和电子版本，允许该论文被查阅和借阅。同意授权广东工业大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印、扫描或数字化等其他复制手段保存和汇编本学位论文。保密论文在解密后遵守此规定。

论文作者签名：　　日期：

指导教师签名：　　日期：

Appendix A Pipeline Verification Program

```
0: main :lui r1, 0 # address of data[0]
1: ori r4, r1, 80 # address of data[0]
2: call : jal sum # call function
3: dslot1: addi r5, r0, 4 # counter, DELAYED SLOT(DS)
4: return: sw r2, 0(r4) # store result
5: lw r9, 0(r4) # check sw
6: sub r8, r9, r4 # sub: r8 ← r9 – r4
7: addi r5, r0, 3 # counter
8: loop:  addi r5, r5, -1 # counter – 1
9: ori r8, r5, 0xffff # zero-extended : 0000ffff
A: xori  r8, r8, 0x5555#zero-extended : 0000aaaa
B: addi  r9, r0, -1 # sign-extended :ffffffff
C: andi  r10, r9, 0xffff # zero-extended : 0000ffff
D: or r6, r10, r9 # or: ffffffff
E: xor r8, r10, r9 # xor: ffff0000
F: and r7, r10, r6 # and: 0000ffff
10: beq r5, r0, shift # if r5 =0, goto shift
11: dslot2:nop # DS
12: j loop2 # jump loop2
13: dslot3: nop # DS
14: shift: addi r5, r0, -1 # r5 = ffffffff
15: sll r8, r5, 15 # <<15 = ffff8000
16: sll r8, r8, 16 # <<16 = 80000000
17: sra r8, r8, 16 # >>16 = ffff8000(arith)
18: srl r8, r8, 15 # >>15 = 0001ffff(logic)
19: finish: j finish # dead loop
20: dslot4: nop # delay slot
```

Appendix B Interrupt and Exception Verification Program

```
0: reset : j start # entry on reset
1: nop
2:EXC_BASE: mfc0 r26, C0_CAUSE # read cp0 Cause reg
3: andi r27, r26, 0xc # get ExcCode, 2 bits here
4: lw r27, j_table (r27) # get address from table
5: nop
6: jr r27 # jump to that address
7: nop
c: int_entry: nop #0.interrupt handler  deal with interrupt here
d: eret # return from interrupt
e: nop
f:sys_entry: nop # SysCall handler
10: epc_plus4: mfc0 r26, C0_EPC # get EPC
11: addir26, r26, 4 #EPC + 4
12: mtc0 r26, C0_EPC #EPC ← EPC +4
13: eret #return from exception
14: nop
15: uni_entry: nop
16: j epc_plus4 #return
17: nop
1a: ovf_entry: nop #overflow handler
1b: j epc_plus4  #return
1c: nop
1d: start: addi r8, r0, 0xf  # IM[3:0] ← 1111
1e: mtc0 r8, C0_STATUS # exc/intr enable
```