VHDL modeling and design of an asynchronous version of the MIPS R30000 microprocessor

Paul Fanelli
VHDL MODELING AND DESIGN OF AN ASYNCHRONOUS VERSION OF THE MIPS R3000 MICROPROCESSOR

by

Paul Fanelli

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of MASTER OF SCIENCE in Computer Engineering

Approved by:
Graduate Advisor - Prof. George A. Brown
Department Chairman - Dr. Roy Czernikowski
Reader - Dr. Tony Chang

DEPARTMENT OF COMPUTER ENGINEERING COLLEGE OF ENGINEERING ROCHESTER INSTITUTE OF TECHNOLOGY ROCHESTER, NEW YORK

FEBRUARY, 1994
THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF ENGINEERING

Title: VHDL Modeling and Design of an Asynchronous Version of the MIPS R3000 Microprocessor.

I, Paul Fanelli, hereby deny permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part.

Date: 2/8/94
ABSTRACT

The goal of this thesis is to demonstrate the feasibility of converting a synchronous general purpose microprocessor design into one using an asynchronous methodology. This thesis is one of three parts that details the entire design of an asynchronous version of the MIPS R3000 microprocessor. The design includes the main architectural features of the R3000: the 5-stage pipeline, the thirty-two 32-bit register bank, and the 32-bit address and data paths. To limit the size of the project, the memory and coprocessor are excluded. Therefore, this design has implemented the entire set of instructions from the original synchronous version with the exception of the coprocessor support instructions.

The three participants in this project are Paul Fanelli, Kevin Johnson, and Scott Siers. Paul Fanelli developed the Very High Speed Integrated Circuit Hardware Description Language (VHDL) models for the processor. Three models, behavioral, dataflow, and structural, were constructed. Kevin Johnson designed the register bank, the arithmetic logic unit, and the shifter, including schematic diagrams and layouts. Scott Siers designed the pipeline stages, the multiplier/divider, the exception handler, and the completion signal generator, including schematic diagrams and layout. Each of the participants has written a separate thesis that covers one part of the total design.
# TABLE OF CONTENTS

ABSTRACT ........................................................................................................ iii

LIST OF FIGURES ........................................................................................... vi

LIST OF TABLES ............................................................................................. x

GLOSSARY OF TERMS ................................................................................... xi

1.0 INTRODUCTION ......................................................................................... 1

2.0 CONCEPTS .................................................................................................. 6
   2.1 ASYNCHRONOUS DESIGN .................................................................... 6
   2.2 HANDSHAKING CONTROL CIRCUIT ................................................... 7
   2.3 VHDL ................................................................................................. 11
   2.4 TOP DOWN DESIGN .......................................................................... 12
   2.5 DATA TYPES ..................................................................................... 15

3.0 BEHAVIORAL MODEL ............................................................................. 17
   3.1 INSTRUCTION FETCH ..................................................................... 21
   3.2 INSTRUCTION DECODE ..................................................................... 23
   3.3 INSTRUCTION EXECUTION ............................................................... 35

4.0 DATAFLOW MODEL ................................................................................. 43
   4.1 INSTRUCTION FETCH STAGE ............................................................ 46
   4.2 INSTRUCTION DECODE STAGE ......................................................... 50
   4.3 ARITHMETIC LOGIC UNIT STAGE .................................................... 66
   4.4 MEMORY STAGE ............................................................................... 68
   4.5 WRITEBACK STAGE ......................................................................... 72
   4.6 BUS CONTROL UNIT ...................................................................... 74
LIST OF FIGURES

Figure 1-1. Test Bench Block Diagram ................................................................. 3
Figure 2-1. HCC Architectural Organization .......................................................... 7
Figure 2-2. HCC Component Block Diagram .......................................................... 8
Figure 2-3. Handshaking Control Circuit (HCC) Schematic Diagram ....................... 9
Figure 2-4. HCC Waveforms ................................................................................. 10
Figure 3-1. Behavioral Model Test Bench .............................................................. 17
Figure 3-2. The Body Outline of the Processor Process .......................................... 19
Figure 3-3. Load Mode .......................................................................................... 20
Figure 3-4. Run Mode While-Loop Shell ............................................................... 20
Figure 3-5. Instruction Fetch ............................................................................... 21
Figure 3-6. Memory Read Procedure in Processor Process .................................... 22
Figure 3-7. R3000 Instruction Formats .................................................................. 23
Figure 3-8. Extracting the Op-code from the Instruction ......................................... 26
Figure 3-9. IF-ELSIF-ELSE Statement used for Instruction Decode .................... 27
Figure 3-10. Special Instruction Branch of IF-ELSIF-ELSE Statement ............... 28
Figure 3-11. Bcond Instruction Branch of IF-ELSIF-ELSE Statement ................. 29
Figure 3-12. Jump Instruction Branch of IF-ELSIF-ELSE Statement ................. 30
Figure 3-13. Branch Instruction Branch of IF-ELSIF-ELSE Statement ............... 31
Figure 3-14. ALU Immediate Instruction Branch of IF-ELSIF-ELSE Statement ...... 32
Figure 3-15. Load and Store Instruction Branch of IF-ELSIF-ELSE Statement ..... 33
Figure 3-16. Halt Instruction Branch of IF-ELSIF-ELSE Statement ..................... 33
Figure 3-17. Not Implemented Instruction Branch of IF-ELSIF-ELSE Statement ...... 34
Figure 3-18. Reserved Instruction Branch of IF-ELSIF-ELSE Statement .......... 34
Figure 3-19. The Shift Left Logical Instruction ........................................ 36
Figure 3-20. The Jump Register Instruction ............................................. 36
Figure 3-21. The Multiply Instruction ..................................................... 37
Figure 3-22. The Add Instruction ............................................................. 38
Figure 3-23. The Branch on Less Than Zero Instruction ............................... 39
Figure 3-24. The Jump Instruction ........................................................... 40
Figure 3-25. The Branch on Equal Instruction ............................................ 40
Figure 3-26. The Add Immediate Instruction ............................................. 41
Figure 3-27. The Load Byte Instruction ................................................... 42
Figure 4-1. Dataflow Model Test Bench ................................................. 43
Figure 4-2. Dataflow Model CPU Component ......................................... 45
Figure 4-3. Schematic of IF Stage ......................................................... 48
Figure 4-4. Waveforms of IF Stage ....................................................... 49
Figure 4-5. Instruction Decoder Component .......................................... 50
Figure 4-6. Schematic Diagram of ID Stage ........................................... 51
Figure 4-7. Code Excerpt of the ID Stage Instruction Decoder Architecture .... 54
Figure 4-8. Address Adder (AA) Component ........................................ 55
Figure 4-9. Code Excerpt of the Address Adder Architecture ..................... 56
Figure 4-10. Branch and Jump (BJBOX) Component ................................ 56
Figure 4-11. Code Excerpt of the BJBOX Architecture ................................ 57
Figure 4-12. Schematic Diagram of BJBOX .......................................... 58
Figure 4-13. Data Dependency Example ............................................. 59
Figure 4-14. The Dirty Box (DBOX) Component .................................... 60
Figure 4-15. Target Register Dirty Select (TRDS) Component .................... 61
LIST OF TABLES

Table 2-1. HCC Component Signal Names and Descriptions.............................................. 8
Table 3-1. R3000 Instruction Opcode Bit Encoding.......................................................... 25
Table 4-1. Instruction Decoder Select Lines........................................................................ 52
Table 4-2. Bit Encoding to Determine Destination Register.............................................. 52
Table 4-3. WB Stage Destination Register Bit Encoding Scheme..................................... 72
Table 5-1. ALUC Operation Encoding............................................................................... 90
Table 6-1. Gate Delay Times............................................................................................ 105
Table 6-2. Component Delay Times.................................................................................. 107
Table 6-3. Stage Delay Times.......................................................................................... 107
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>Address Adder, hardware unit that calculates the address of next instruction, located in the instruction decode stage of pipeline</td>
</tr>
<tr>
<td>Accusim</td>
<td>Mentor Graphics Corporation analog circuit simulator</td>
</tr>
<tr>
<td>ADD8</td>
<td>Unit inside the ALU stage that calculates the link address for branch conditional instructions</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit, third stage of the pipeline</td>
</tr>
<tr>
<td>ALUB</td>
<td>Arithmetic Logic Unit Block, one of the units that comprises the ALU stage; consists of the ALUC, ALUC decoder, shifter, shifter control, branch control, etc.</td>
</tr>
<tr>
<td>ALUC</td>
<td>Arithmetic Logic Unit Component, unit that performs addition, subtraction, and logical computations</td>
</tr>
<tr>
<td>bcond</td>
<td>Branch Conditional group of instructions</td>
</tr>
<tr>
<td>BCB</td>
<td>Bus Control Block, unit inside the ALUB that controls what gets placed on the A and B busses in the ALU stage</td>
</tr>
<tr>
<td>BCU</td>
<td>Bus Control Unit, unit that controls which stage (IF or MEM) is granted access to the address and data busses</td>
</tr>
<tr>
<td>BJBOX</td>
<td>Branch and Jump Box, unit inside the ID stage that controls what gets sent to the AA</td>
</tr>
<tr>
<td>bton</td>
<td>Bits-to-Natural function, converts bits to natural numbers</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>current_inst</td>
<td>Variable that holds the current instruction</td>
</tr>
<tr>
<td>DBOX</td>
<td>Dirty Box, unit inside the ID stage that handles data dependencies</td>
</tr>
<tr>
<td>DCVSL</td>
<td>Differential Cascade Voltage Switch Logic</td>
</tr>
</tbody>
</table>
DOD  Department of Defense

ea  Variable that holds the effective address, used to calculate effective address of load and store instructions

EH  Exception Handler, unit that handles exceptions from the IF, ID, ALU, and MEM stages and generates an interrupt vector that is sent to the IF

FLOW  Reorders a program that contains branch and jump instructions to follow the program flow.

FSM  Finite State Machine

funct  Function field that holds the minor operation code used for instruction decoding

GPR  General Purpose Register

HCC  Handshaking Control Circuit

ibo  Immediate or base-offset, one of the output signals of the instruction decoder in the ID stage that goes high either for an immediate instruction or for an instruction that needs a base-offset calculation

ID  Instruction Decode, second stage of pipeline

IE  Instruction Execution

IF  Instruction Fetch, first stage of pipeline

immed  Field that holds the immediate value used for instruction decoding

LSB  Least Significant Bit

MASS  MIPS Assembler, program written to convert MIPS assembly code into machine code

MDU  Multiplier/Divider Unit

MEM  Memory, fourth stage of pipeline

MERA  MIPS Expected Results Assembler, program written to help generate an expected results file used in conjunction with each models test bench
<p>| <strong>MIPS</strong> | Name of company that designed, developed, and built the R3000 microprocessor, also stands for Millions of Instructions Per Second which is a processor speed rating |
| <strong>MPP</strong> | MIPS Preprocessor, program written to load processor with test programs |
| <strong>MSB</strong> | Most Significant Bit |
| <strong>MU</strong> | Mask Unit, unit inside the MEM stage that determines the length of the requested data and whether the data is sign or zero-extended |
| <strong>ns</strong> | Nanoseconds |
| <strong>offset</strong> | Field that holds the offset value used for instruction decoding |
| <strong>op</strong> | Field that holds the major operation code used for instruction decoding |
| <strong>opcode</strong> | Operation Code |
| <strong>PC</strong> | Program Counter |
| <strong>pc_reg</strong> | Variable that holds the program counter value |
| <strong>R3000</strong> | Model number of processor this thesis is modeling |
| <strong>rd</strong> | Destination register field used for instruction decoding |
| <strong>RISC</strong> | Reduced Instruction Set Computer |
| <strong>rs</strong> | Source register field used for instruction decoding |
| <strong>rt</strong> | Target register field used for instruction decoding |
| <strong>shamt</strong> | Field that holds the shift amount |
| <strong>special</strong> | Special Instructions, name of group of MIPS instructions |
| <strong>SU</strong> | Shift Unit, unit inside the MEM stage that shifts data when it is not aligned on a word boundary |
| <strong>target</strong> | Field that holds the target value used for instruction decoding |</p>
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRDS</td>
<td>Target Register Dirty Select, unit that selects which register will be set dirty</td>
</tr>
<tr>
<td>vbt</td>
<td>Valid Byte Tag, this signal is generated by the decoder inside the MEM stage and is used by the GPR bank in the ALU stage; it specifies which bytes of data are valid and can be written back to a register</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>WB</td>
<td>Writeback, fifth stage of pipeline</td>
</tr>
</tbody>
</table>
1.0 INTRODUCTION

This thesis is one of three parts encompassing the modeling, design, and implementation of an asynchronous microprocessor. This project was performed cooperatively by Scott Siers, Kevin Johnson, and this author. The project has been divided so that each part forms a separate master's thesis. This paper discusses the Very High Speed Integrated Circuit Hardware Description Language (VHDL) modeling of this processor. The asynchronous processor uses most of the R3000's instruction set and architectural features but differs in implementation. Three models, behavioral, dataflow, and structural, are constructed. The behavioral model describes the functionality of the R3000 without regard to implementation. The dataflow model represents the pipeline of the processor. It models the data flowing through the pipeline stages. The structural model represents the processor at the gate or structural level. The dataflow and structural model delay times were back annotated from circuit simulation runs. Each model is tested using a VHDL test bench to verify correct operation. Kevin Johnson and Scott Siers were responsible for the Very Large Scale Integration (VLSI) design and implementation. Kevin Johnson designed the register bank, the arithmetic logic unit (ALU), and the shifter. Scott Siers designed the bus control logic, the multiplier/divider unit (MDU), and the pipeline structure. This author also participated in the design of the asynchronous processor from a modeling perspective. The hardware designs were changed or completely redone depending upon the results of the VHDL modeling.

The main purpose of this project was to investigate the feasibility of converting an existing synchronous processor design to an asynchronous design. The MIPS R3000 was chosen for this task for numerous reasons. The R3000 has the best combination of instruction set size, architectural features, and system complexity to fully exploit the differences between synchronous and asynchronous design. The R3000 was one of the first reduced instruction set computer (RISC) processors and is a very simplistic, concise,
and elegant architecture. The number of instructions is minimal and there are only three addressing modes. For a 32-bit machine it has a small architecture and hence makes an ideal processor for thesis work. There is an abundance of literature written on the MIPS R3000. For example, the paper written by Asada, Okura, and Cho in 1992 [1] discusses the design of an asynchronous implementation of the MIPS data path. Another paper by Ginosar and Michell [2] discusses converting the MIPS pipeline using an asynchronous design methodology. Also, the book MIPS RISC Architecture [3] gives extremely low-level details of the synchronous version of the processor. These features make the R3000 an ideal processor to model.

The MIPS R3000 is a general purpose microprocessor that includes a 32-bit data path with thirty-two 32-bit general purpose registers. The R3000 has a 5-stage pipeline. The five stages are instruction fetch (IF), instruction decode (ID), arithmetic logic unit (ALU), memory (MEM), and register writeback (WB). The three main addressing modes are register, immediate, and jump. The R3000 is a reduced instruction set computer (RISC). One characteristic of a RISC based architecture is that it is a register based design. The processor only works on data contained in the registers. The order of operations is that data is loaded from memory into the registers, the processor works on the data in the registers, the result is stored back into a register, and finally the result is put back into memory. The advantage of this architecture is that the instruction set is smaller (reduced) and consists of simpler instructions. This allows the cycle time for each instruction to be short. All of the R3000's instructions are 32 bits in length. This is another advantage of RISC machines. The complexity of the instruction decoder is minimal and the instruction set size is limited.

Before any modeling was done, certain issues had to be considered. What architectural features were to be modeled? Which instructions were to be modeled? How many specific models should be designed and at what level of detail? During preliminary discussions on this thesis topic it was decided that the main architectural features of the
R3000 would be modeled: the 5-stage pipeline, the thirty-two 32-bit register bank, the hi/lo registers, and the 32-bit address and data paths. The features that would be left out are the coprocessor and coprocessor support, the cache for memory and instruction fetch, and the memory management. Since it was decided that memory would not be modeled, memory management would not be implemented. The coprocessor and memory management instructions were left out. Due to these architectural decisions and to limit the size of this thesis, the asynchronous processor instruction set was reduced.

The VHDL modeling in this thesis consists of the following three models: behavioral, dataflow, and structural. All models consist of three modules: memory, central processing unit (CPU), and compare. These three modules when put together with the test program and the expected results file create a complete test bench. Figure 1-1 shows the test bench setup. Each model will use the test bench along with test programs to verify correct operation. The memory and compare modules are relatively similar for all three models. The memory module is used as the main memory storage for the

![Figure 1-1. Test Bench Block Diagram](image-url)
processor and holds the test program used in the test bench. The compare module is used to test the state of the processor after each instruction is executed and holds the expected results file. The CPU module corresponds to one of the three models.

Software was written for this thesis to assist in testing the models. An assembler, called MIPS Assembler (MASS), was written to convert MIPS assembly code into machine code that the models can understand. Also, an expected results program was written. This program, called MIPS Expected Results Assembler (MERA), allows the user to input the expected results data into a file. This file is loaded into the compare module and is tested against the state of the model after each instruction is executed. Another program was written for branch and jump instructions. This program, called FLOW, takes a program that contains branch and jump instructions and reorders the instructions to follow the program flow. The last program is called MIPS Preprocessor (MPP). This program takes the files created by MASS and MERA and copies them into two files that are used by the models. These two files, "machine" and "expected", are loaded by the memory and compare modules, respectively.

Six test programs were written for the models. Each of these tests correspond to a set of instructions. The program "ai.test" tests immediate arithmetic instructions. "ar.test" tests register arithmetic instructions. The third program, "jb.test", is used for jump and branch instructions. The program "ls.test" is used to test load and store instructions. The fifth test, "md.test", checks the multiplication and division instructions along with the move to and from the hi/lo registers. The last test file, "s.test", is used for the shift instructions.

The software tools used in this project are from Mentor Graphics Corporation and run on HP/Apollo Workstations. Five software tools were used: Design Architect, System-1076 (VHDL) editor and compiler, Quicksim II, Accusim, and IC Station. Design Architect is a schematic capture tool. The VHDL editor and compiler is incorporated into Design Architect. The digital simulator is Quicksim II. The VHDL simulator is
embedded into Quicksim II. Accusim is the analog circuit simulator. Finally, the mask layout editor is called IC Station.
2.0 CONCEPTS

One major feature of synchronous design is the use of a global clock. The very nature of synchronous design is to control all events based on this clock. Multiple events can happen but they will not be triggered until the next clock pulse. The order of events is of no concern. On the other hand, asynchronous design avoids the use of a global clock. Therefore, there is no convenient way to synchronize events. Here, the order of events is very important. A controller can be used between logic blocks as a communication device. The controller uses start and done signals as handshake signals. When one logic block is finished, it sends the controller a done signal. The controller can now send the next logic block a start signal. With synchronous design, the logic blocks can be tuned to start at a certain time based on the clock phase. However, the exact time at which a specific event starts or ends is not known in asynchronous design. This is why a controller is needed. It coordinates the timing of events through the use of handshaking signals.

2.1 ASYNCHRONOUS DESIGN

An asynchronous design approach was chosen over a synchronous one for many reasons. As transistor sizes in VLSI keep getting smaller, the major drawbacks to synchronous design become more apparent and difficult to tolerate. Two major disadvantages are the skew associated with a global clock and the increasing line delay that occurs when a signal is routed across a VLSI chip. One of the major design goals in synchronous design is to increase the performance of the processor by reducing the clock period. However, as the reduction in scale of VLSI systems continues, more and more of the clock period is used to account for clock skew. Global clock lines become more sensitive to loading and it becomes increasingly difficult to keep the various clock line signals in phase. The second issue involves line delay. In the past, the major delay in
circuit design was the transistor gate. Today, the line delay in signal routing is a major concern. To obtain substantial increases in performance, new architectures will have to be used to reduce the need for long metal lines. Circuit modules will have to be linked only by local interconnections and the modules will then communicate via self-timed handshaking schemes. The asynchronous design that eliminates the global clock and reduces the need and effects of long signal lines is an attractive solution in modern VLSI implementation.

2.2 HANDSHAKING CONTROL CIRCUIT

The handshaking control circuit (HCC) represents the control flow mechanism for the asynchronous machine. Every major logic block in the design has an HCC associated with it. The HCC synchronizes the events among the major logic blocks. The HCC coordinates control information from its own logic block and from other HCCs in the design. Figure 2-1 shows the HCC architectural organization. The major logic blocks shown in the diagram correspond to the five pipeline stages of the processor. An HCC is dependent on the previous and the next HCC. The HCCs on the ends of the pipeline are

![HCC Architectural Organization](image)

*Figure 2-1. HCC Architectural Organization*
only dependent on its one neighbor. For example, the ALU HCC is dependent on the ID and MEM HCCs. The ALU stage cannot begin operation until the ID stage is finished and the MEM stage has latched the previous data from the ALU stage. The HCC component block diagram is shown in Figure 2-2. The signal names and their descriptions are shown in Table 2-1.

![HCC Component Block Diagram](image)

**Figure 2-2. HCC Component Block Diagram**

<table>
<thead>
<tr>
<th>SIGNAL NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>Initializes the HCC</td>
</tr>
<tr>
<td>ok(n-1)</td>
<td>Previous stage has completed its operation</td>
</tr>
<tr>
<td>aout</td>
<td>Acknowledgment to previous stage that data has been latched</td>
</tr>
<tr>
<td>rout</td>
<td>Start signal to logic block</td>
</tr>
<tr>
<td>ready</td>
<td>Done signal from logic block</td>
</tr>
<tr>
<td>ok(n)</td>
<td>Signals next stage that present stage is completed</td>
</tr>
<tr>
<td>ain</td>
<td>Acknowledgment from next stage that data has been latched</td>
</tr>
</tbody>
</table>

**Table 2-1. HCC Component Signal Names and Descriptions**

The HCC schematic circuit diagram is shown in Figure 2-3 and its waveforms are shown in Figure 2-4. The operation of the HCC is as follows. The *init* signal is used to initialize the HCC and acts like a reset. The *ok(n-1)* signal is the start signal for the HCC. This signal comes from the previous HCC. When *ok(n-1)* goes low, it signals that the...
Figure 2-3. Handshaking Control Circuit (HCC) Schematic Diagram
Figure 2-4. HCC Waveforms
previous stage has valid data. The HCC sends the acknowledgment signal $aout$ high when $rout$ and $ready$ are both low. When $aout$ is high and $ain$ is low, $rout$ goes high. $rout$ is the start signal to the HCC's logic block. The HCC now waits for the completion of the logic block. This is signified by the $ready$ line going high. Once $ready$ is high, the HCC sends $ok(n)$ low. This signals to the next HCC that the logic block has valid data. The HCC now waits for $ain$ to go high which is an acknowledgment from the next HCC. $ain$ going high causes $rout$ to go low and $ok(n)$ to go high. When $rout$ goes low, $ready$ goes low. The cycle then repeats itself.

2.3 VHDL

VHDL stands for VHSIC Hardware Description Language. VHSIC stands for Very High Speed Integrated Circuit. The Department of Defense (DOD) initiated the VHSIC program. The DOD also initiated VHDL to create a hardware description language that all VHSIC contractors could use to specify their designs. More importantly, this would allow designs to be transferred from one company to another and be totally independent of the tools and the platforms they run on.

VHDL is a high level language. High level constructs are important for design specification (behavioral model) and testing. VHDL allows the user to concentrate on the behavioral aspects of the design and forget the low level details during the beginning stages of design. VHDL is an IEEE standard formalized in specification 1076-1987 and updated in specification 1076-1992. Writing in VHDL allows the user to port the source code over to another hardware platform with ease. All that needs to be done is to recompile the source code on the new platform and to run the new platform's simulator.

VHDL is a programming language that can simulate concurrent events. This allows the user to specify multiple events at the same simulation time. VHDL uses the
concept of delta delay, which is an infinitesimally small delay, to order events that occur at the same simulation time. VHDL software tools incorporate a simulator to test the design, and a test bench can be used to simplify the testing process. Also, VHDL has packages specifically designed to model hardware at different stages of design.

2.4 TOP DOWN DESIGN

Top down design is a popular design methodology. It guides a design from a high level to a low level of abstraction. The system's functionality is described at a high level of abstraction. Implementation of this functionality is not an issue at this level. On the contrary, the system's low level details are only considered at a low level of abstraction. The low level describes the gate level implementation. As the design progresses from one level of abstraction to the next, functionality of the design is completed and set at the higher levels and then the implementation of this functionality is created at the lower levels. This thesis uses this concept by using the three different models. Each model is another level of abstraction.

A behavioral model is a model that describes the behavior of the hardware entity under test. An entity is a hardware unit that can be as simple as a gate or as complex as an entire electronic system. A behavioral model does not explicitly specify the structure of the entity but specifies its functionality. Another way of looking at the behavioral style of modeling is the well known "black box" approach. The hardware unit is described in terms of its input-output mapping without specifying the model's technology, components, or dataflow.

A behavioral model projects a very high level of abstraction. At the first stages of design, the behavioral model relieves the user of the low-level details of the design and implementation of the entity. This frees the designer and/or user to concentrate on the
behavior of the system in question. Overall, the behavioral model provides a means to better understand the functionality of the entity. Finally, due to the high level of abstraction, the behavioral model executes much faster than other modeling schemes. This is advantageous at the beginning stages of a design when many simulations need to be done.

The behavioral modeling technique was used to model the functionality of the R3000 without regard to hardware implementation. The behavioral model was used to understand the operation of every instruction that was implemented. The entire behavioral model is actually one VHDL PROCESS statement. All instructions in a PROCESS statement are executed sequentially. Therefore, the pipeline was modeled in a sequential fashion. An instruction is fetched, decoded, executed, stored, and tested all within one cycle of the PROCESS statement. The next instruction is not worked on until the first instruction is finished. The data type used in this model is the BIT VECTOR. This choice is discussed in section 2.4. The computation instructions are implemented using functions and procedures since this model is only concerned with the operation of the processor and not on how it is implemented. These functions and procedures are located in a VHDL PACKAGE.

A dataflow model describes the behavior of the hardware entity just like the behavioral model but in more detail. Dataflow modeling involves some implementation details since it is concerned with the flow of data from one part to another. The dataflow model starts to break the functionality of the behavioral model down into compartments. An example of this is the breakdown of the pipeline. Each stage of the pipeline performs a different function. The dataflow model specifies how the data will flow from one section to another.

Dataflow modeling is similar to behavioral modeling since there are no gates to specify low level implementation, but it is also similar to structural modeling because of the use of components (even though the components are written using the dataflow style).
The dataflow model uses the same data types and package of declarations, functions, and procedures as the behavioral model. This was done to simplify the model. The main focus of attention was to get the pipeline stages talking to each other and to assure that each stage was decoding and working on the proper instruction. Also, to keep the dataflow model simple, only two logic states (0 and 1) were used to describe and simulate the model.

The dataflow model provided the next level of abstraction by modeling the pipeline stages concurrently. This model is used to design the handshaking interface protocol between the pipeline stages. The main objective of this model was to establish proper and efficient communication between the pipeline stages. A rough protocol would be designed and then tested using the dataflow model. Using the simulation waveform outputs of the model, the protocol was modified to improve the design. Also, design errors were corrected using the dataflow output waveforms. This continued throughout the entire design cycle of the asynchronous processor.

The dataflow model is largely a hierarchical structure of dataflow components. The top level component is the CPU. The CPU is then broken up into eight unique components: the five pipeline stages, the HCC, the bus controller (BC), and the exception handler (EH). Each stage of the pipeline has an HCC associated with it. Each of these components are made of other smaller components and primitives. Examples of the primitives are multiplexers, latches, and edge detectors. The components were used to speed up the model building process. When a particular component is needed, the proper code is called through use of the VHDL COMPONENT instantiation and PORT MAP statements.

The original dataflow model was constructed using arbitrary delay times just to get the model working. Back annotation was used once the model was completed, tested, and verified for correct operation. The new delay times were obtained from Accusim simulation runs of circuit descriptions of the various components and pipeline stages.
These simulations were performed and discussed in Scott Siers' thesis and Kevin Johnson's thesis "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" [10,19].

The structural model represents the processor at its gate or structural level. This is the lowest form of abstraction. It is the lowest and most detailed level of description. The structural model uses a set of components connected by signals. With the structural model, the behavior of the entity is not apparent from the model. This is unlike the behavioral model where the behavior or functionality is readily apparent. Component instantiation is the major VHDL device that facilitates a structural or gate level nature.

2.5 DATA TYPES

Originally, integers were chosen as the main modeling data type. However, this presented some problems. The representation of data by integers appears to be ideal but instructions need a more robust data type. Different pieces of the instruction represent different aspects and conditions of the computer. An instruction holds operation, register operand, and memory address information. Also, depending on the addressing mode, an instruction holds different types and amounts of information. A composite type called a record was considered in order to hold all the different fields of information contained in an instruction. However, Mentor Graphic's version of VHDL now in use (version 8.1) does not support records.

At this time, a decision was made to use a data type called a bit vector. Bit vectors are ideal to use because they represent the language that the computer understands. Bit vectors can be manipulated using two methods. Using the first method, the bit vector is converted to an integer, the integer is operated on, and then the integer is converted back to a bit vector. This method is faster to execute but involves much conversion. The
second method is to manipulate the individual bits. These bit manipulations are handled by procedures and functions. An example of this is adding two bit vectors together. The two bit vectors are passed to the overloaded "+" function. The "+" function adds the individual bits together and returns a bit vector result. These specialized functions add complexity and slow execution time.
3.0 BEHAVIORAL MODEL

The behavioral model is an instantiated component. The name of the model is CPU, since the central processing unit is the main part of the MIPS R3000 that is modeled. This CPU component is part of a test bench shown in Figure 3-1. The other two components of the test bench are the memory and compare modules.

The memory module is used as main memory storage for the processor. It is accessed by six interface signals. The memory control signals, `mem_control_sig` and

\[\text{TikZ Diagram of the Behavioral Model Test Bench}\]

*Figure 3-1. Behavioral Model Test Bench*
mem_ack_sig, provide a fully interlocked handshaking protocol between the CPU component and the memory component. The address bus is broken into two separate signals: addr_bus and addr_bus_lo. Addr_bus provides the upper 30 bits (bits 2 through 31) of the 32 bit address bus, which accesses a word of memory (a word of memory is 32 bits wide). Addr_bus_lo provides the lower two bits (bits 0 and 1) of the address bus, which accesses a byte of memory (a byte of memory is 8 bits wide). The data_bus is a 32-bit bi-directional bus. It is used to transfer data between the CPU and the memory module. Finally, the exception control signal, mem_exception_sig, is activated when a memory exception occurs.

The compare module tests the state of the processor after each instruction is executed. It is accessed by ten interface signals. The compare control signals, compare_control_sig and compare_ack_sig, provide the handshaking between the CPU and the compare module. The pc signal monitors the CPU program counter. Signals r1, r2, r3, r4, and r3l monitor the contents of the specified registers. The hi and lo signals monitor the multiplication and division storage registers.

The CPU module or component uses a VHDL PROCESS to model the processor. A PROCESS statement is a collection of sequential statements that describe the functionality or behavior of a portion of an ENTITY. A PROCESS is first entered during the initialization phase of a simulation. During this initialization, it continues to execute until it suspends due to an explicit WAIT statement or an implicit WAIT due to a sensitivity list. Also, once a PROCESS is entered, it is never exited. It is always in one of two states: active or suspended. A PROCESS is active when it is executing and suspended when it is waiting for a certain event to occur.

A PROCESS is sensitive to signals in a sensitivity list. If an event occurs on any one or more of the signals in the sensitivity list, the PROCESS is executed. The statements in the PROCESS are executed in a sequential fashion. It suspends after executing the last sequential statement and waits for another event to occur on a signal in
the sensitivity list. A WAIT statement can be used in place of a sensitivity list. A PROCESS executes until a WAIT statement is reached. The PROCESS is suspended until an event occurs on the signal in the WAIT statement.

The body outline of the PROCESS used in this model is shown in Figure 3-2. It is sensitive to the \textit{sys\_control\_sig} signal using a WAIT statement. The process is broken down, using a VHDL CASE statement, into four sections each corresponding to the four modes of operation: \textit{stop, reset, load,} and \textit{run}.


displayed code

\begin{verbatim}
processor: PROCESS
   -- process declarations
BEGIN
   WAIT ON sys_control_sig;
   CASE sys_control_sig IS
     WHEN stop =>
       ...
     WHEN reset =>
       ...
     WHEN load =>
       ...
     WHEN run =>
       ...
   END CASE;
END PROCESS processor;
\end{verbatim}

Figure 3-2. The Body Outline of the Processor Process

The \textit{load} mode, shown in Figure 3-3, is part of the system initialization. \textit{load} sends signals to the memory and compare components to load the system programs. More precisely, the \textit{load} mode initiates handshake signals with the memory and compare components. It sends the \textit{load} signal to the memory component via \textit{mem\_control\_sig} and to the compare component via \textit{compare\_control\_sig}. When both components are finished loading their respective programs, they send back their acknowledgment signals. A reset signal is then sent to both components.
WHEN load ->
  run_mode_flag := no;
  mem_control_sig <= load AFTER delay;
  WAIT UNTIL mem_ack_sig = yes;
  mem_control_sig <= reset AFTER delay;
  WAIT UNTIL mem_ack_sig = no;
  compare_control_sig <= load AFTER delay;
  WAIT UNTIL compare_ack_sig = yes;
  compare_control_sig <= reset AFTER delay;
  WAIT UNTIL compare_ack_sig = no;

Figure 3-3. Load Mode

Run mode, shown in Figure 3-4, is the largest part of the processor PROCESS. It starts by first setting run_mode_flag to yes. This flag controls the exit condition from a VHDL WHILE loop. While run_mode_flag is set to a value of yes, the processor continues to work as it cycles through the WHILE loop. When run_mode_flag changes to no, the run mode WHILE loop is exited and the processor ceases to work. The main part of the processor model is contained inside this WHILE loop. The WHILE loop is broken down into nine sections: instruction fetch (IF), instruction decode (ID), instruction execution (IE), exception handling, program counter update, memory latency handling, branch delay update, signal update, and testing.

WHEN run ->
  run_mode_flag := yes;
  WHILE run_mode_flag = yes LOOP
    -- instruction fetch
    -- instruction decode
    -- determine which type of addressing it is
    -- set fields accordingly
    -- exception handling
    -- delay slot/po increment
    -- latency of one instruction on register load
    -- set branch delay flag
    -- update signals
    -- compare machine state with expected results
  END LOOP;

Figure 3-4. Run Mode While-Loop Shell
3.1 INSTRUCTION FETCH

The instruction fetch portion of the code is very simple; it is only one line and can be seen in Figure 3-5. Instruction fetch is accomplished by calling the `mem_read` procedure. The `mem_read` procedure is discussed in the next section. The two arguments needed to do a memory read are the starting address of the data to read and the size of the data. The value in the program counter register (`pc_reg`) gives the starting address and the value `word` gives the data size. The data size is of type `word` since all instructions are 32-bits long. The procedure result is placed in `current_inst`, a 32-bit variable which stands for current instruction.

```plaintext
-- fetch next instruction
mem_read(pc_reg, word, current_inst);
```

*Figure 3-5. Instruction Fetch*

`mem_read`, shown in Figure 3-6, takes two arguments, the memory address and the memory size, and returns the result from the data bus. `mem_read` first sets up the address bus signals, `addr_bus` and `addr_bus_lo`, from the value passed to it. Depending upon what type of memory operation is to be performed, it selects from the `size` variable what value to assign to `mem_control_sig`. It now waits for the memory component to send back the memory acknowledge signal, `mem_ack_sig`. The data from the memory is now ready to be transferred to `result`. `mem_read` now sends a reset signal to the memory and waits again for an acknowledgment. This completes the fully interlocked handshaking scheme. Finally, the `mem_exception_sig` is checked to see if a memory exception has occurred.
PROCEDURE mem_read(addr: IN bit_32; size: IN size_type;
result: OUT bit_32) IS
BEGIN
    addr_bus <= addr[31 DOWNTO 2] AFTER delay;
    addr_bus_lo <= addr[1 DOWNTO 0] AFTER delay;
    CASE size IS
        WHEN byte =>
            mem_control_sig <= read_b AFTER delay;
        WHEN ubyte =>
            mem_control_sig <= read_ub AFTER delay;
        WHEN halfword =>
            mem_control_sig <= read_hw AFTER delay;
        WHEN uhalfword =>
            mem_control_sig <= read_uhw AFTER delay;
        WHEN word =>
            mem_control_sig <= read_w AFTER delay;
        WHEN lefty =>
            mem_control_sig <= read_l AFTER delay;
        WHEN righty =>
            mem_control_sig <= read_r AFTER delay;
    END CASE;
    WAIT UNTIL mem_ack_sig = yes;
    result := data_bus;
    mem_control_sig <= reset AFTER delay;
    WAIT UNTIL mem_ack_sig = no;
    IF mem_exception_sig = yes THEN
        exception_flag := addr_load;
    END IF;
END mem_read;

Figure 3-6. Memory Read Procedure in Processor Process
3.2 INSTRUCTION DECODE

Instruction decode involves extracting the op-code from the current instruction, determining what instruction grouping the instruction belongs to, and then setting the operand and address fields accordingly. The next instruction to execute is stored in the current_inst variable. An instruction can take on one of the three instruction formats shown in Figure 3-7. The major op-code is the six most significant bits of the instruction. On some instructions, a minor op-code is used. This minor op-code, also called a function field, is the least significant six bits of some instructions.

<table>
<thead>
<tr>
<th>I-Type (Immediate)</th>
<th>31 26 25 21 20 16 15 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J-Type (Jump)</th>
<th>31 26 25 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R-Type (Register)</th>
<th>31 26 25 21 20 16 15 11 10 6 5 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
</tr>
</tbody>
</table>

where:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>is a 6-bit operation code</td>
</tr>
<tr>
<td>rs</td>
<td>is a 5-bit source register specifier</td>
</tr>
<tr>
<td>rt</td>
<td>is a 5-bit target (source/destination) register or branch condition</td>
</tr>
<tr>
<td>immediate</td>
<td>is a 16-bit immediate, branch displacement or address displacement</td>
</tr>
<tr>
<td>target</td>
<td>is a 26-bit jump target address</td>
</tr>
<tr>
<td>rd</td>
<td>is a 5-bit destination register specifier</td>
</tr>
<tr>
<td>shamt</td>
<td>is a 5-bit shift amount</td>
</tr>
<tr>
<td>funct</td>
<td>is a 6-bit function field</td>
</tr>
</tbody>
</table>

Figure 3-7. R3000 Instruction Formats
To understand instruction decoding, the R3000 instruction op-code bit encoding needs to be addressed. This bit encoding is shown in Table 3-1. Table 3-1 is composed of three tables: opcode, special, and bcond. The opcode table displays all the possible bit combinations of the major op-code. The rows of the opcode table represent the three most significant bits of the major op-code field. The columns represent the three least significant bits. When the major op-code of an instruction is equal to 00 octal, then the instruction is a special instruction. The special instructions are shown in the special table. The special instructions are encoded through the minor op-code field variable called funct (function field). The special table displays all the possible combinations of the special instructions. The rows of the special table represent the three most significant bits of funct. The columns represent the three least significant bits. When the major op-code is equal to 01 octal, then the instruction is a branch conditional (bcond) instruction. The branch conditional instructions are shown in the bcond table of Table 3-1. These instructions are encoded through a bcond field variable called reg_funct which is a five bit field. The rows of the bcond table represent the two most significant bits of the reg_funct field. The columns represent the three least significant bits of reg_funct.
| 31..29 | 28..26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|--------|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0      | special| bcond| i   | jal | beq | bne | blez | bgtz |
| 1      | addi   | addiu| slti| sltu| andi| ori | xor  | lui |
| 2      | cop0 **| cop1 **| cop2 **| cop3 ** | * | * | * | *
| 3      | * | * | * | * | * | * | * | *
| 4      | lb | lh | lw | lw | lbu | lhu | lwr | *
| 5      | sb | sh | swl | sw | * | * | * | *
| 6      | lwc0 **| lwc1 **| lwc2 **| lwc3 ** | * | * | * | *
| 7      | swc0 **| swc1 **| swc2 **| swc3 ** | * | * | * | *

<table>
<thead>
<tr>
<th>5..3</th>
<th>2..0</th>
<th>1..0</th>
<th>0..0</th>
<th>31..28</th>
<th>20..19</th>
<th>19..16</th>
<th>18..16</th>
<th>17..16</th>
<th>16..16</th>
<th>15..16</th>
<th>14..16</th>
<th>13..16</th>
<th>12..16</th>
<th>11..16</th>
<th>10..16</th>
<th>9..16</th>
<th>8..16</th>
<th>7..16</th>
<th>6..16</th>
<th>5..16</th>
<th>4..16</th>
<th>3..16</th>
<th>2..16</th>
<th>1..16</th>
<th>0..16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sll</td>
<td>*</td>
<td>srl</td>
<td>sra</td>
<td>sllv</td>
<td>*</td>
<td>srlv</td>
<td>srav</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>jr</td>
<td>jalr</td>
<td>*</td>
<td>*</td>
<td>syscall</td>
<td>break</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 2      | mfhi   | mthi | mthi | mflo   | mflo  | *    | *      | *
| 3      | multi  | multi| multi| multi  | multi | *    | *      | *
| 4      | add    | addu | sub  | subu   | subu  | and  | or     | xor    |
| 5      | *      | *    | *    | *      | *      | *    | *      | *
| 6      | *      | *    | *    | *      | *      | *    | *      | *
| 7      | *      | *    | *    | *      | *      | *    | *      | *

* Operation codes marked with an asterisk cause reserved instruction exceptions and are reserved for future versions of the architecture.

** Operation codes marked with two asterisks are not implemented in the asynchronous version.

*Table 3-1. R3000 Instruction Opcode Bit Encoding*
The op-code is extracted from current_inst by using a bit-vector array range, as shown in Figure 3-8. The top six bits of current_inst is extracted and stored in a variable called opcode using an array range (a bit-vector is an array of bits) from bit 31 down to bit 26. The op-code is further broken down by extracting a segment of the variable opcode. This is stored in a variable called opcode_segment. Opcode_segment is the three most significant bits of opcode and is used to determine the instruction grouping.

```vhdl
opcode := current_inst(31 DOWNTO 26);
opcode_segment := opcode(5 DOWNTO 3);
```

*Figure 3-8. Extracting the Op-code from the Instruction*

After the op-code is extracted from the current instruction, the instruction grouping is determined. The major op-code instructions are broken up into the following groups: special, bcond, jump, branch, immediate, load, and store instructions. A VHDL IF-ELSIF-ELSE statement, shown in Figure 3-9, is used to direct the program flow to the correct instruction grouping. All instructions are named the same as the R3000 instruction set except for the addition of the letter "i" and an underscore bar ("_i"). The R3000 major op-code instruction special is called i_special, for example. All the instructions are defined in a VHDL PACKAGE as constants to improve code readability and debugging. The i_special instruction is defined as a constant bit value of "000000", for example.

The IF-ELSIF-ELSE statement is broken up into nine branches. When the major op-code is equal to the constant i_special then the first branch is taken. This first branch handles all the R3000 special instructions. The second branch handles the four branch conditional instructions. This occurs when the major op-code is equal to the constant i_bcond. The two jump instructions, i_j and i_jal, are found in the third branch. The fourth branch holds the branch instructions and is taken when opcode_segment is equal to "000". The fifth branch holds the ALU immediate instructions and is taken when
opcode\_seg is equal to "001". The load and store instructions are grouped together in the sixth branch. The seventh branch is the \textit{halt} instruction. This instruction is used to stop the processor and is used at the end of every test program. It is not part of the R3000 instruction set. The eighth and ninth branches handle instructions \textit{not-implemented} and \textit{reserved} instructions, respectively. Each of the branches will be discussed in more detail in the following sections.

```plaintext
-- special instructions
IF opcode = i\_special THEN

-- conditional branch instructions
ELSIF opcode = i\_cond THEN

-- jump, jump and link instructions
ELSIF opcode = i\_j OR opcode = i\_jal THEN

-- branch instructions
ELSIF opcode\_seg = b^"000" THEN

-- ALU immediate instructions
ELSIF opcode\_seg = b^"001" THEN

-- load and store instructions
ELSIF opcode\_seg = b^"100" OR opcode\_seg = b^"101" THEN

-- halt instruction (not a mips instruction)
ELSIF opcode = i\_halt THEN

-- instructions not implemented
ELSIF (opcode\_seg = b^"010" OR opcode\_seg = b^"110" OR
     opcode\_seg = b^"111") AND opcode \_ (2) = \'0' THEN

-- reserved instruction
ELSE

END IF;
```

Figure 3-9. IF-ELSIF-ELSE Statement used for Instruction Decode
The *special* instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-10. It first extracts the fields from the current instruction. All *special* instructions use the register instruction format. This format consists of the minor op-code field (this is also known as the function field), the source register field (rs), the target register field (rt), the destination register field (rd), and the shift amount field (shamt). The minor op-code is calculated from the current instruction by extracting an array range on `current_inst` and storing it in `funct`. The register operand fields (rs, rt, rd, and shamt) are more complicated. First, the proper array range of `current_inst` is extracted. Next, this extracted array of bits is converted to a natural number using the bits-to-natural (`bton`) function. Lastly, the natural number, which specifies a register number from 0 to 31, is stored in the appropriate variable. Once all the fields are set accordingly, the program flow branches to the proper instruction. A VHDL CASE statement is used to select which *special* instruction is executed. The specific *special* instruction is denoted by the minor op-code which is stored in the `funct` field.

```
IF opcode = i_special THEN
    funct := current_inst(5 DOWNTO 0);
    rs := bton(current_inst(25 DOWNTO 21));
    rt := bton(current_inst(20 DOWNTO 16));
    rd := bton(current_inst(15 DOWNTO 11));
    shamt := bton(current_inst(10 DOWNTO 6));
    CASE funct IS
        WHEN i_all =>
            .
        WHEN i_srl =>
            .
        WHEN i_sra =>
            .
        WHEN i_sllv =>
            .
            .
        END CASE;
```

*Figure 3-10. Special Instruction Branch of IF-ELSIF-ELSE Statement*
The branch conditional (bcond) instructions of the IF-ELSIF-ELSE statement is shown in Figure 3-11. The bcond instructions use the immediate instruction format. Therefore, the fields that need to be set are the register source (rs), bcond function (reg_funct), and offset (offset). The bcond function, which is the branch conditional minor op-code, is stored in the reg_funct variable. The variable is called reg_funct because the minor op-code information field is located in the same place as the target register field for other instruction formats in the current_inst variable. The offset is a 16-bit value that is extracted from the lower 16 bits of the current instruction. The minor op-code is calculated from the current instruction by extracting an array range on current_inst and storing it in reg_funct. The rs field is calculated the same way as the rs field in the special instructions section. The array range is extracted from current_inst. The array range is then converted to a natural number using the bton function. Lastly, the value is stored in the rs field variable. Once all the fields are set accordingly, the program branches to the proper branch conditional instruction. This is calculated using a CASE statement and selecting on reg_funct.

```
ELSIF opcode = i_bcond THEN
    rs := bton(current_inst(25 DOWNTO 21));
    reg_funct := current_inst(20 DOWNTO 16);
    offset := current_inst(15 DOWNTO 0);

    CASE reg_funct IS
        WHEN i_bltz =>
            ...
        WHEN i_bgez =>
            ...
        WHEN i_bltzal =>
            ...
        WHEN i_bgezal =>
            ...
        END CASE;
```

Figure 3-11. Bcond Instruction Branch of IF-ELSIF-ELSE Statement
The jump instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-12. The jump instructions use the jump instruction format. This format uses a target field. The target field is a 26-bit jump target address and is extracted from the lower 26 bits of current_inst. Once the target field is set, the program branches to the proper jump instruction. This is calculated by using a CASE statement and selecting on opcode.

```
ELSIF opcode = i_j OR opcode = i_jal THEN
    target := current_inst[25 DOWNTO 0];
    CASE opcode IS
        WHEN i_j =>
            ...
        WHEN i_jal =>
            ...
    END CASE;
```

*Figure 3-12. Jump Instruction Branch of IF-ELSIF-ELSE Statement*

The branch instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-13. The branch instructions use the immediate instruction format. Therefore, the fields that need to be set are rs, rt, and offset. Both register operand fields, rs and rt, are first extracted from current_inst, converted using the bton function, and finally stored in their respective variables. The offset value is extracted from current_inst and stored in the offset variable. Once these three fields are set, the program branches to the proper branch instruction. This is calculated using a CASE statement and selecting on opcode.
ELSIF opcode seg = b"000" THEN
    rs := bton(current_inst(25 DOWNTO 21));
    rt := bton(current_inst(20 DOWNTO 16));
    offset := current_inst(15 DOWNTO 0);
    CASE opcode IS
        WHEN i_beq =>
            ...
        WHEN i_bne =>
            ...
        WHEN i_blez =>
            ...
        WHEN i_bgtz =>
            ...
        END CASE;

Figure 3-13. Branch Instruction Branch of IF-ELSIF-ELSE Statement

The ALU immediate instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-14. As the name implies, the ALU immediate instructions use the immediate instruction format. The three fields that are set in this instruction format are rs, rt, and immed. As stated before, rs stands for register source and rt stands for register target. Immed is a variable used for the immediate field. The immediate field is a 16-bit immediate, branch, or address displacement. The rs and rt fields are extracted, converted, and stored in their respective variables. The immediate field is extracted from current inst and stored in immed. Once the fields are set, the program branches to the proper ALU immediate instruction. This is done using a CASE statement with opcode as the selector.
ELSIF opcode_seg = b"001" THEN
  rs := bton(current_inst(25 DOWNTO 21));
  rt := bton(current_inst(20 DOWNTO 16));
  immed := current_inst(15 DOWNTO 0);
  CASE opcode IS
    WHEN i_addi =>
      ...
    WHEN i_addiu =>
      ...
    WHEN i_slti =>
      ...
  END CASE;

Figure 3-14. ALU Immediate Instruction Branch of IF-ELSIF-ELSE Statement

The load and store instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-15. The load and store instructions use the immediate instruction format with the following three fields: rt, base, and offset. rt stands for the target register, base is an alias for the rs field, and offset is an alias for the immediate field. Once all the fields are extracted, converted, and stored, the program branches to the proper load or store instruction depending on the value of opcode.
ELSIF opcode_seg = b"100" OR opcode_seg = b"101" THEN
    base := bton(current_inst(25 DOWNTO 21));
    rt := bton(current_inst(20 DOWNTO 16));
    offset := current_inst(15 DOWNTO 0);
    CASE opcode IS
        WHEN i_lb =>
            ...
        WHEN i_lh =>
            ...
        WHEN i_hb =>
            ...
        WHEN i_hh =>
            ...
        WHEN i_ab =>
            ...
        WHEN i_ac =>
            ...
        WHEN i_ah =>
            ...
    END CASE;

Figure 3-15. Load and Store Instruction Branch of IF-ELSIF-ELSE Statement

The _halt_ instruction branch is shown in Figure 3-16. This instruction is not part of the R3000 instruction set. It is added to aid testing and debugging. The _halt_ instruction is added at the end of every test program. It stops the processor by setting the _run_mode_flag_ to _no_. Note, the _inst_ signal is used to monitor the state of the current instruction with the VHDL simulator.

ELSIF opcode = i_halt THEN
    inst := op_halt;
    run_mode_flag := no;

Figure 3-16. Halt Instruction Branch of IF-ELSIF-ELSE Statement

The _not-implemented_ instruction branch is shown in Figure 3-17. These instructions are found in the R3000 instruction set but are not implemented in the models. As shown in the figure, the _inst_ and _exception_flag_ signals are set to _not_implmt._
The last instruction branch of the IF-ELSIF-ELSE statement is used for reserved instructions and is shown in Figure 3-18. These op-code values cause reserved instruction exceptions and are reserved for future versions of the architecture by the manufacturer. The inst signal is set to reserved and the exception_flag signal is set to the reserved_inst exception.
3.3 INSTRUCTION EXECUTION

The instructions are divided into the following six groups: *special, branch conditional, jump, branch, ALU immediate,* and *load/store* instructions. The groups are discussed separately with a few examples for each group. This instruction grouping follows the instruction op-code bit encoding chart previously shown in Table 3-1. Note, every instruction uses the *inst* signal to display which instruction is currently being executed. Also, since register 0 (*r0*) is hard wired to the value zero, every instruction that stores a value in a register has to check that the destination register is not zero. If the destination register is zero, then the instruction is ignored. Lastly, many instructions perform their operations by calling a function that is located in a VHDL PACKAGE.

**SPECIAL INSTRUCTIONS**

The *special* instructions are further divided into the following groups: *shift, jump register, special, multiply/divide,* and *3-operand register* instructions. There are six shift instructions: *shift-left logical (sll), shift-right logical (srl), shift-right arithmetic (sra), shift-left logical variable (sllv), shift-right logical variable (srlv),* and *shift-right arithmetic variable (srav).* The logical shift instructions insert zeroes into the vacant bit positions. If the shift is to the left, zeroes are inserted into the low order bits. A right shift inserts zeroes into the high order bits. An arithmetic shift uses sign extension when inserting values into the vacant bit positions. The arithmetic shift is only used on a right shift. Therefore when a right arithmetic shift is performed, the high order bits are sign extended. A variable shift gets its shift amount from the contents of a register. More precisely, the low order 5 bits of register *rs* specify the number of bits to shift. If the shift is not variable, then the shift amount is held in the *shamt* field of those particular shift instructions. As an example, the *sll* instruction is shown in Figure 3-19. *sll* performs the
shift by calling the \texttt{shift\_ll} function. \texttt{Shift\_ll} takes two arguments: a 32-bit value to shift and the shift amount. \texttt{shift\_ll} returns a 32-bit shifted value. More precisely, \texttt{sll} shifts the contents of register \texttt{rt} left by \texttt{shamt} bits, inserting zeroes into the low order bits. It then places the 32-bit result in register \texttt{rd}.

\begin{verbatim}
WHEN i_sll =>
   inst <= op_sll;
   IF rd /= 0 THEN
      reg(rd) := shift\_ll(reg(rt), shamt);
   END IF;
\end{verbatim}

\textit{Figure 3-19. The Shift Left Logical Instruction}

There are two \texttt{jump register} instructions: \texttt{jump register (jr)} and \texttt{jump and link register (jalr)}. Both instructions jump to an address contained in register \texttt{rs} with a one instruction delay. The \texttt{jalr} instruction also places the address of instruction following the delay slot in register \texttt{rd}. As an example, the \texttt{jr} instruction is shown in Figure 3-20. The address that is stored in register \texttt{rs} is the location to which the program jumps after a delay of one instruction. Since the instruction in the delay slot needs to be executed before the jump is executed, the address is stored in a temporary variable called \texttt{pc\_temp}. At the proper time, the program counter (PC) is loaded with the value in \texttt{pc\_temp}. The \texttt{delay\_slot\_flag} controls when PC is loaded with \texttt{pc\_temp}.

\begin{verbatim}
WHEN i_jr =>
   inst <= op_jr;
   pc_temp := reg(rs);
   delay_slot_flag := set;
\end{verbatim}

\textit{Figure 3-20. The Jump Register Instruction}

There are two \texttt{special} instructions: \texttt{syscall} and \texttt{break}. \texttt{syscall} initiates a system call trap and immediately transfers control to the exception handler. \texttt{break} initiates a
breakpoint trap and also immediately transfers control to the exception handler. Since an operating system will not be modeled, these instructions are "dummy" instructions and do not do anything useful. However, since the exception handling is modeled, syscall and break cause an exception which halts the processor.

There are eight multiply/divide instructions: move from hi (mfhi), move to hi (mthi), move from lo (mflo), move to lo (mtlo), multiply (mult), multiply unsigned (mulu), divide (div), and divide unsigned (divu). The first four instructions move data to and from the hi_reg and lo_reg registers. The hi_reg and lo_reg registers hold results of integer multiplication and division operations. As an example, the mult instruction is shown in Figure 3-21. The mult instruction multiplies the contents of registers rs and rt as two's complement values. The mult function returns a value that is placed in a temporary 64-bit variable mult_temp. mult_temp is divided into two 32-bit values representing the most and least significant 32 bits. The most significant 32 bits are stored in hi_reg. The least significant 32 bits are stored in lo_reg.

```
WHEN i_mult =>
  inst <= op_mult;
  mult_temp := mult(reg(rs), reg(rt));
  lo_reg := mult_temp(31 DOWNTO 0);
  hi_reg := mult_temp(63 DOWNTO 32);
```

*Figure 3-21. The Multiply Instruction*

There are ten 3-operand register instructions: add (add), add unsigned (addu), subtract (sub), subtract unsigned (subu), AND (and), OR (or), XOR (xor), NOR (nor), set on less than (slt), and set on less than unsigned (sltu). All ten instructions perform their operations on the 3-operand registers: rs, rt, and rd. The two operands are stored in the rs and rt registers. The results of the operation is stored in rd. The first four instructions are arithmetic instructions. The next four instructions perform logical operations. The
last two instructions are used to compare and set registers. As an example, the add instruction is shown in Figure 3-22. The add instruction adds the contents of registers rs and rt and places the 32-bit result in register rd. If an overflow occurs, the ovrflw bit is set to '1'. The overflow condition is checked using an IF statement. If an overflow condition exists, then exception_flag is set to enumerated value overflow.

```
WHEN l_add =>
    inst <= op_add;
    IF rd /= 0 THEN
        add_ovf(reg(ra), reg(rt), reg(rd), ovrflw);
        IF ovrflw = '1' THEN
            exception_flag := overflow;
        END IF;
    END IF;
END IF;
```

*Figure 3-22. The Add Instruction*

**BRANCH CONDITIONAL INSTRUCTIONS**

There are four branch conditional, or bcond, instructions: branch on less than zero (bltz), branch on greater than or equal to zero (bgez), branch on less than zero and link (bltzal), and branch on greater than or equal to zero and link (bgezal). These instructions change the control flow of a program depending on a condition. The link instructions save a return address in register 31 (r31). All branch instruction target addresses are computed by adding the address of the instruction in the delay slot with the 16-bit offset. The 16-bit offset is shifted left two bits and sign extended to 32 bits. All branches occur with a delay of one instruction. As an example, the bltz instruction is shown in Figure 3-23. The bltz instruction branches to the target address if register rs is less than zero. The check for less than zero is accomplished by testing bit 31 of the contents of rs. If bit 31 is '1' then the value in rs is negative. The pc_reg variable holds
the address of present instruction, the _bltz_ instruction. The address of the instruction in the delay slot is computed by adding four to _pc_reg_. The 16-bit _offset_ is manipulated by first sign-extending it to 32 bits and then shifting it left by two bits. Finally, the target address is computed by adding the modified _offset_ to the modified program counter. The target address is placed in a temporary program counter (_pc_temp_) since all branch instructions have a delay of one instruction. If the branch is to be taken, at the proper time, _pc_reg_ is updated with _pc_temp_. _delay_slot_flag_ is the variable that controls when _pc_reg_ is updated.

```
WHEN l_bltz =>
  inst <= op_bltz;
  IF reg(rs)(31) = '1' THEN
    pc_temp := pc_reg + x"0000_0004";
    pc_temp := pc_temp +
      shift_l1(s16to32(offset),2);
    delay_slot_flag := set;
  END IF;
```

*Figure 3-23. The Branch on Less Than Zero Instruction*

**JUMP INSTRUCTIONS**

There are two _jump_ instructions: _jump_ (_j_) and _jump and link_ (_jal_). Both instructions jump to an address contained in the _target_ field. More precisely, the 26-bit _target_ address is shifted left two bits and combined with the high-order 4 bits of the program counter. The program jumps to the address with a one instruction delay. The _jal_ instruction also places the address of instruction following the delay slot in the link register, _r31_. As an example, the _j_ instruction is shown in Figure 3-24.
BRANCH INSTRUCTIONS

There are four branch instructions: branch on equal (beq), branch on not equal (bne), branch on less than or equal to zero (blez), and branch on greater than zero (bgtz). The branch instructions are similar in operation to the branch condition instructions. The branch target address is calculated from the sum of the address of the instruction in the delay slot with the 16-bit offset. The 16-bit offset is shifted left two bits and sign-extended to 32 bits. As an example, the beq instruction is shown in Figure 3-25. The beq instruction branches to the target address if the contents of general register rs and rt are equal, with a delay of one instruction.
ALU IMMEDIATE INSTRUCTIONS

There are eight ALU immediate instructions: add immediate (addi), add immediate unsigned (addiu), set on less than immediate (slti), set on less than immediate unsigned (sltiu), AND immediate (andi), OR immediate (ori), XOR immediate (xori), and load upper immediate (lui). All eight instructions perform their operations using the two operand registers rs and rt, and an 16-bit immediate field. The first four instructions sign-extend the immediate field. The last four instructions zero-extend the immediate field. As an example, the addi instruction is shown in Figure 3-26. The addi instruction adds the sign-extended immediate field to the contents of general register rs to form a 32-bit result. This result is stored in general register rt. An overflow exception occurs if the two highest order carry-out bits differ. This is known as two's complement overflow.

```
WHEN 1_addi =>
  inst <= op_addi;
  IF rt /= 0 THEN
    add_ovf(reg(rs), salto32(immed), reg(rt),
            ovrfw);
    IF ovrfw = '1' THEN
      exception_flag := overflow;
    END IF;
  END IF;
```

Figure 3-26. The Add Immediate Instruction

LOAD/STORE INSTRUCTIONS

There are seven load instructions: load byte (lb), load halfword (lh), load word left (lw), load word (lw), load byte unsigned (lbu), load halfword unsigned (lhu), and load word right (lwr). There are five store instructions: store byte (sb), store halfword (sh), store word left (swl), store word (sw), and store word right (swr). All twelve load/store instructions calculate the effective address (ea) by sign-extending the 16-bit
offset and adding it to the contents of general register base. The contents of the memory location are sign-extended when the signed load instructions are used. When the unsigned load instructions are used, the contents are zero-extended. As an example, the lb instruction is shown in Figure 3-27. The ea is calculated by sign-extending the offset using the sel6to32 function and adding it to the contents of the base register. The memory is accessed by the mem_read procedure. This procedure passes the ea and the type of data to be read (byte) as its' arguments.

```
WHEN i_lb =>
    inst <= op_lb;
    IF rt /= 0 THEN
        ea := sel6to32(offset) + reg(base);
        mem_read(ea, byte, temp_reg_val_1);
        temp_reg_num_1 := rt;
        latency_flag := set;
    END IF;
```

*Figure 3-27. The Load Byte Instruction*
4.0 DATAFLOW MODEL

The dataflow model is a hierarchical structure of other dataflow components. The top level of the model consists of three components: the CPU, the memory, and the compare module. These three parts comprise the dataflow test bench as shown in Figure 4-1.
The memory module is accessed by eight interface signals. The `memory_load` and `memory_load_ack` signals provide the handshaking necessary to load the memory with the test program. This is done when the asynchronous processor is initialized. The `memory_req` and `memory_ack` signals provide the handshaking necessary to read and write data between the CPU and memory module. The `memory_w` and `memory_opcode` signals are used in conjunction with a memory write. The last two signals, `addr_bus` and `data_bus`, correspond to the address bus and data bus, respectively. All these signals are discussed in more detail in the following sections.

The compare module is accessed by 12 interface signals. The `compare_load` and `compare_load_ack` signals provide the handshaking to load the compare module with the expected results file during system initialization. The `compare` and `compare_ack` signals provide the handshaking to test the state of the processor after each instruction. The remaining eight signals, `pc_test` through `lo_test`, monitor the contents of the specified registers.

The CPU module, shown in Figure 4-2, is composed of eight unique dataflow components. The pipeline is made up of five of these components. The pipeline stages are: instruction fetch (IF), instruction decode (ID), arithmetic logic unit (ALU), memory (MEM), and writeback (WB). Each stage of the pipeline has a handshaking control circuit (HCC) associated with it. The HCC controls the operation of the specific stage. The bus control unit (BCU) acts as a high speed polling device between the IF and MEM stages. It grants access to the address and data busses. The last component is the exception handler (EH). It sends an interrupt request and vector to the IF stage when an exception occurs from either the IF, ID, ALU, or MEM stages.
Asynchronous Version of the MIPS R3000

Figure 4.2: Dataflow Model CPU Component
4.1 INSTRUCTION FETCH STAGE

The IF is the first stage of the pipeline and it fetches the next instruction from memory. The first item that the IF needs is a valid address. The valid address is calculated by the address adder (AA) in the ID stage. Therefore, the IF has to wait for the AA to calculate the new address before it can use it. However, on the first instruction the IF doesn't have to wait since the program counter (PC) is initially zero. This is essential to prevent deadlock. Since the ID cannot start until the IF finishes, IF cannot initially wait for the AA which is inside the ID. Once the IF gets the valid address, it needs access to the address and data busses to retrieve the data from memory. The IF sends out a request to the BCU. The BCU grants the address bus to the IF stage if it is free. The BCU is discussed in more detail in section 4.6.

The IF also handles branching to an interrupt vector. Under normal operations, the IF just continues to fetch the next instruction once it receives a valid address. However, when an exception occurs, an interrupt and interrupt vector is generated by the EH. The IF jumps to this interrupt vector instead of the new PC value. The IF stage can also generate an exception. This occurs when the address of an instruction is not aligned on a word boundary. In other words, if the two least significant bits are not zero then an address exception is generated.

The schematic to the IF stage is shown in Figure 4-3. The IF is started by the if_start signal going high. This start signal is generated by the IF's HCC. Once it receives the start signal, the IF waits for either addr_valid to go low or int_req to go high. The addr_valid signal going low signifies that the ID has calculated the new address and this address is valid. The int_req signals that an interrupt request has been generated. The IF receives its new address from either the new_pc or iv lines. The new_pc line comes from the AA. The iv stands for interrupt vector and comes from the EH. These two lines are multiplexed and selected by the int_req line. If an interrupt occurs, then the IF uses the
interrupt vector (iv) value. On the other hand, if the IF is in normal operation then the new PC value (new_pc) is used.

When addr_valid goes low, the IF sends a bus request (ir) to the BCU. This signal is also used to latch the address that comes from the AA. The IF now waits until the BCU sets ia high. The ia signal going high is an acknowledgment from the BCU that the IF has been granted access to the address bus. When ia goes high, it enables the tri-state buffer, placing the address on the address bus. This ia acknowledgment signal is needed to allow the address to be setup on the address bus before the memory read. Now the IF sends the BCU the il line high signaling that the address has been loaded on the address bus. The BCU can now initiate the memory request. The IF waits until ia goes low signaling that the data bus now contains a valid instruction. The ia line going low causes the instruction on the data bus to be latched and resets the ir and il signals. Resetting the ir line signals to the BCU that the IF is finished with the address bus. The waveforms showing the IF operation are shown in Figure 4-4.
Instruction Fetch Stage

Figure 4-3. Schematic of IF Stage
Figure 4.4. Waveforms of IF Stage

<table>
<thead>
<tr>
<th></th>
<th>if_start</th>
<th>addr_valid</th>
<th>if_bus_req</th>
<th>if_bus_ack</th>
<th>if_load_addr</th>
<th>if_done</th>
<th>inst_ifid</th>
<th>pc</th>
<th>addr_bus</th>
<th>data_bus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Waveforms of IF Stage
4.2 INSTRUCTION DECODE STAGE

The ID is the second stage of the pipeline and has three main tasks. The first task is to decode the instruction. The second task is to calculate the destination address for all branch and jump instructions, and to increment the PC on other instructions. The third task stalls the pipeline during data dependencies. These three tasks are explained in more detail in the following sections. The overall schematic diagram of ID is shown in Figure 4-6.

INSTRUCTION DECODER

The instruction decoder component, shown in Figure 4-5, decodes the instruction into 17 different select lines. These select lines are distributed within the ID stage and to the ALU stage. The 17 select lines are described in Table 4-1 and 4-2. A code excerpt of the instruction decoder is shown in Figure 4-7.
Figure 4-6. Schematic Diagram of ID Stage
<table>
<thead>
<tr>
<th>SIGNAL</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ill</td>
<td>illegal</td>
<td>This signal goes high when an illegal instruction is encountered</td>
</tr>
<tr>
<td>exc</td>
<td>exception</td>
<td>This signal goes high when the instruction is syscall or break</td>
</tr>
<tr>
<td>md</td>
<td>MDU select</td>
<td>This signal goes high when an instruction needs to use the MDU in the ALU stage</td>
</tr>
<tr>
<td>alu</td>
<td>ALU select</td>
<td>This signal goes high when an instruction needs to use the ALU in the ALU stage</td>
</tr>
<tr>
<td>add8</td>
<td>add8 unit select</td>
<td>This signal goes high when an instruction needs to use the add8 unit in the ALU stage</td>
</tr>
<tr>
<td>ibo</td>
<td>immediate or base offset</td>
<td>This signal goes high either for an immediate instruction or an instruction needs a base-offset calculation</td>
</tr>
<tr>
<td>b</td>
<td>branch</td>
<td>This signal goes high on a branch instruction</td>
</tr>
<tr>
<td>j</td>
<td>jump</td>
<td>This signal goes high on a jump instruction</td>
</tr>
<tr>
<td>r</td>
<td>&quot;register&quot; instruction</td>
<td>This signal goes high when the instruction is a jump register (jr) or a jump and link register (jalr)</td>
</tr>
<tr>
<td>l</td>
<td>&quot;link&quot; instruction</td>
<td>This signal goes high when the instruction is a jump and link (jal) or a jump and link register (jalr)</td>
</tr>
<tr>
<td>ts1</td>
<td>test 1st source register</td>
<td>This signal goes high when the first source register must be tested for a data dependency</td>
</tr>
<tr>
<td>ts2</td>
<td>test 2nd source register</td>
<td>This signal goes high when the second source register must be tested for a data dependency</td>
</tr>
<tr>
<td>ttar</td>
<td>test target register</td>
<td>This signal goes high when the destination register must be tested for a data dependency</td>
</tr>
<tr>
<td>thi</td>
<td>test hi register</td>
<td>This signal goes high when the hi register must be tested for a data dependency</td>
</tr>
<tr>
<td>tlo</td>
<td>test lo register</td>
<td>This signal goes high when the lo register must be tested for a data dependency</td>
</tr>
<tr>
<td>trs0, trs1</td>
<td>target register select bits</td>
<td>These signals are used to determine the destination register, SEE TABLE 4-2 for the bit encoding</td>
</tr>
</tbody>
</table>

Table 4-1. Instruction Decoder Select Lines

<table>
<thead>
<tr>
<th>TRS0</th>
<th>TRS1</th>
<th>5-BIT ENCODED DESTINATION REGISTER VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>bits 15-11 of the instruction</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>bits 20-16 of the instruction</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>set destination register to 0 (&quot;00000&quot;) - means no destination register</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>set destination register to 31 (&quot;11111&quot;) - used for link instructions</td>
</tr>
</tbody>
</table>

Table 4-2. Bit Encoding to Determine Destination Register
The first two signals, \textit{ill} and \textit{exc}, represent illegal and exception instructions, respectively. The \textit{ill} signal is set high when the instruction decoder finds an instruction that is not part of the instruction set. The instruction set can be found in Table 3-1. The \textit{exc} signal goes high on a \textit{syscall} or \textit{break} instruction. Both of these instructions cause a software exception.

The next three signals, \textit{md, alu,} and \textit{add8}, select the MDU, ALU, and ADD8 in the ALU stage, respectively. The MDU and ALU are never activated together by the same instruction. When the ALU is selected, the MDU is idle for the particular instruction. When the MDU is selected, the ALU is idle. Both lines are set low when an instruction does not need to use either unit. The ADD8 unit is only used by \textit{branch conditional} (\textit{bcond}) instructions which need to calculate a link address (\textit{bltzal} and \textit{bgezal}). The ADD8 unit is needed because the AA and the ALU are busy doing other calculations for the two \textit{bcond} link instructions. The AA calculates the branch destination address and the ALU calculates the condition code (whether or not to take the branch).

The fifth signal, \textit{ibo}, stands for immediate or base-offset. This signal controls a multiplexer to the ALU A-bus inside the ALU stage. It determines what value gets fed into the A input of the ALU (not the ALU stage but the ALU component). The \textit{ibo} signal is high for an immediate or base-offset instruction. Base-offset values are calculated for all load/store and branch instructions. However, the \textit{ibo} line is only used for load/store instructions because base-offset calculations for branch instructions are done by the AA.

The next four signals, \textit{b, j, r,} and \textit{l}, determine the type of jump or branch. A branch instruction is denoted when the \textit{b} line is high. The branch target address is (\textit{PC} + 4) + (offset * 4). This is calculated by adding the sum of the address of the instruction in the delay slot to a 16-bit offset which is shift left two bits and sign-extended to 32-bits. A jump instruction is denoted by the \textit{j} line. The jump target address is \textit{PC}(31:28) \& target \& "00". This is calculated by concatenating (\&\&) the four most significant bits of the PC, the 26-bit \textit{target}, and two zeros. The \textit{r} line goes high if the jump instruction is a "register"
Jump register (jr) and jump and link register (jalr) are the two "register" instructions. This r line is needed because a jump register instructions get their jump target address from the contents of a register. The l line goes high on a jump and link instruction. The two jump and link instructions are jump and link (jal) and jump and link register (jalr). For these instructions, the ID must give the ADD8 unit in the ALU stage the PC to calculate the link address of (PC + 8).

The next five signals, ts1, ts2, ttar, thi, and tlo, are for data dependency checking. Each of these signals, when set high, tests to see if respective register is dirty. IF a register is dirty then the ID is stalled until the previous instruction writes its data to this register. Data dependency is discussed in more detail in a later section.

The last two signals, trs0 and trs1, are used together to determine where in the instruction to get the destination register value. The four choices are shown in Table 4-2. When trs0 and trs1 equal "00" then the destination register value is found at bits 15-11 of the instruction. This choice is used for all special instructions. When trs0 and trs1 equal "01" then the destination is found at bits 20-16 of the instruction. This choice is used for loads and immediate instructions. These instructions use a 16-bit offset or immediate value located at bits 15-0 of the instruction. Therefore the destination bit location is
moved to bits 20-16. When \( trs0 \) and \( trs1 \) equal "11" then the destination register is set to 31 ("11111"). This is used by the jump and link (jal) instruction. All other instructions do not have a destination register. Therefore, the last choice sets the destination register to zero ("00000"). Register zero is hardwired to a value of zero.

**ADDRESS ADDER**

The address adder (AA) performs two functions: calculate destination addresses and increment the PC. If the instruction is a branch or a jump the AA calculates the destination address. For all other instructions, the AA increments the PC. The AA component is shown in Figure 4-8. The VHDL code is shown in Figure 4-9.

![Address Adder (AA) Component](image)

*Figure 4-8. Address Adder (AA) Component*

The AA has its own start and done signal. The AA start signal, \( aas \), goes high when certain conditions are met: \( aas = id\_start \) (NOT\( (b\_l) + ccd \)). The \( id\_start \) signal is the start signal for the ID stage. The \( b\_l \) signal is the output of a latch that stores whether the last instruction was a branch. The condition code done (\( ccd \)) signal is sent by the ALU stage and indicates when the condition code evaluation (whether or not to take a branch or jump) has finished. The \( aas \) signal goes high when \( id\_start \) is high and the last
instruction was not a branch. If the last instruction was a branch, then the AA has to wait until the ALU stage sends the ccd signal. Therefore, aas also goes high when id_start is high and ccd is high. The AA done signal, aad, is generated by the AA when it completes it operation.

```vhdl
ARCHITECTURE dfaa_a OF dfaa IS
BEGIN
  o <= a + b AFTER 7 ns WHEN s = '1' ELSE 'X'0000_0000';
  d <= '1' AFTER 0 ns WHEN s = '1' ELSE '0' AFTER 1 ns;
END dfaa_a;
```

*Figure 4-9. Code Excerpt of the Address Adder Architecture*

The AA input and output ports are all 30 bits. The two least significant bits are always zero so that the address of each instruction is aligned on a word boundary in memory. The A-input port, aa_a_input, gets it value from the PC. The B-input port, aa_b_input, gets is value from the output of the BJBOX (Branch and Jump Box) component. BJBOX selects what gets added to the PC (A-input) depending on the type of the previous instruction. The BJBOX component is shown in Figure 4-10. The schematic diagram is shown in Figure 4-12. A code excerpt is shown in Figure 4-11.

*Figure 4-10. Branch and Jump (BJBOX) Component*
The inst signal provides the lower 26 bits of the instruction. A jump instruction uses all 26 bits and is called the target value. A branch instruction uses only the lower 16 bits and is called an offset value. When the condition code, cc, is high then the branch is taken. The pc signal provides the upper four bits of the PC. This is needed to calculate the jump target address. The reg signal is used with the register jump instructions. This signal is the register jump target address. The b, j, and jr signals correspond to a branch, jump, and register jump instructions, respectively. The addr signal is the BJBOX output and is the address that is passed to the AA.

Figure 4-11. Code Excerpt of the BJBOX Architecture

BJBOX has to handle four different instruction cases: branches, jumps, register jumps, and all other instructions. For the branch instruction case, BJBOX sign-extends
Figure 4-12. Schematic Diagram of BJBOX
the 16-bit offset of the branch instruction and passes it to the AA. For the jump instruction case (j and jal), the pc, target, and two zeros ("00") are concatenated together to form the jump target address. For the register jump instruction case (jr and jalr), the value in the reg signal is loaded into the PC. The last case is used for all other instructions. Here, a value of four (x"0000_0004") is passed to the AA. This allows the PC to be incremented by one word.

DATA DEPENDENCIES

A data dependency occurs when two adjacent instructions try to share the same resources. As an example, consider the two instructions shown in Figure 4-13. The first instruction places its results in register three (r3) when it reaches the WB stage. However, before instruction 1 can write back its answer, the second instruction tries to use it. If this data dependency problem is left unchecked, instruction 2 would receive an invalid value for r3. For proper operation, the second instruction has to wait until the first instruction is finished and writes back its results. Only then can instruction 2 use the value in r3.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>add</td>
<td>r3</td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>(2)</td>
<td>add</td>
<td>r5</td>
<td>r3</td>
<td>r4</td>
</tr>
</tbody>
</table>

*Figure 4-13. Data Dependency Example* 

The data dependency problem is handled by tagging a register when it is "dirty" (i.e. specified as the destination by a previous instruction). This is done by using extra bits for each register, called dirty bits. Two bits are used to avoid a mutual exclusion problem. This problem exists because two different stages of the pipeline have to access these dirty bits. The ID has to set the dirty bits when a register is used as a destination. The WB has
to reset the dirty bits when it writes the data back to the register. Both stages could access the dirty bits simultaneously leading to unexpected results. The mutual exclusion problem is avoided by exclusive ORing the two bits together producing one "dirty bit". The two bits are named \textit{db\_id} and \textit{db\_wb}, and are set by the ID and WB stages, respectively. The operation of the "dirty bits" are as follows. When both bits are either zero ("00") or one ("11") the register is clean. However, if the bits differ ("01" or "10") then the register is dirty. For more information on data dependencies and the dirty bits, see "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" by Kevin Johnson [10].

The unit that handles data dependencies is called Dirty Box (DBOX) and is shown in Figure 4-14. DBOX has two tasks. The first task is to set the destination register of the current instruction as dirty (if there is a destination register). The second task is to test the source and destination registers of the current instruction to see if they were set dirty by the previous instruction.

![Figure 4-14. The Dirty Box (DBOX) Component](image)

ct\_start
inst\_32
db\_32
hi\_db
lo\_db
ts1
ts2
ttar
thi
tlo
trs0
trs1

\textit{dirty\_reg}
\textit{clean}
The first task is implemented by a unit inside DBOX called Target Register Dirty Select (TRDS), shown in Figure 4-15. TRDS selects which register will be set dirty. TRDS input ports are inst, trs0, and trs1. The inst signal is needed to determine which register is the destination. The trs0 and trs1 signals specify where to find the destination register. These two signals are discussed in instruction decoder section and can be found in Tables 4-1 and 4-2. The schematic diagram of TRDS is shown in Figure 4-18. The VHDL code of the TRDS architecture is shown in Figure 4-16.

![Figure 4-15. Target Register Dirty Select (TRDS) Component](image)

```
ARCHITECTURE dftrds_a OF dftrds IS
BEGIN
  reg <= inst(15 DOWNTO 11) AFTER 0.4 ns WHEN -- special
    trs0 = '0' AND trs1 = '0' ELSE
    inst(20 DOWNTO 16) AFTER 0.4 ns WHEN -- imm or load
      trs0 = '0' AND trs1 = '1' ELSE
      b"11111" AFTER 0.4 ns WHEN -- jal
      trs0 = '1' AND trs1 = '1' ELSE
      b"00000" AFTER 0.4 ns;
END dftrds_a;
```

![Figure 4-16. VHDL Code of the TRDS Architecture](image)

The second task is handled by DBOX and the schematic diagram is shown in Figure 4-19. The dirty bit signals, db, hi_db, and lo_db, come from the register banks in the ALU stage and are used for dirty bit testing. The db signal is a 32-bit line, each line
representing one dirty bit of the 32 general purpose registers. The hi\(_{db}\) and lo\(_{db}\) signals represent the hi and lo register dirty bits, respectively. The ct\(_{start}\) line starts the dirty bit testing. When this line is low, the dirty bit testing is valid. The five signals, ts1, ts2, ttar, thi, and tlo, determine which registers to test. These signals are generated by the instruction decoder and are discussed in the instruction decoder section. Tables 4-1 and 4-2 give a brief description of each signal. The dirty\(_{reg}\) signal is five bits wide and is the TRDS output. It specifies which destination register will be set dirty. If there is no destination register for the current instruction, then register zero is specified. The last signal, clean, stalls the ID stage when there is a data dependency. The VHDL code of the DBOX architecture is shown in Figure 4-17.
ARCHITECTURE dfdbox_a OF dfdbox IS

COMPONENT dftrds
  PORT (inst: IN bit_32;
   trs0: IN bit;
   trs1: IN bit;
   reg: OUT bit_5);
END COMPONENT;

COMPONENT df32tolmux
  port (i: IN bit_32;
   s: IN bit_5;
   c: OUT bit);
END COMPONENT;

SIGNAL a: BIT := '1';
SIGNAL b, d, e, f, g, h: BIT;
SIGNAL s, p, q: BIT;
SIGNAL reg: bit_5;
SIGNAL c: bit;

BEGIN

clean <= NOT (a OR b OR c) AFTER 1.1 ns;
dirty_reg <= reg;
c <= NOT ct_start AFTER 0.3 ns;
a <= NOT (d AND e) AFTER 0.7 ns;
b <= NOT (f AND g AND h) AFTER 0.6 ns;
d <= NOT (hi_db AND thi) AFTER 0.7 ns;
e <= NOT (lo_db AND tlo) AFTER 0.7 ns;
f <= NOT (ta1 AND n) AFTER 0.7 ns;
g <= NOT (ts2 AND p) AFTER 0.7 ns;
h <= NOT (ttar AND q) AFTER 0.7 ns;
mux1: df32tolmux
  PORT MAP (db, inst[25 DOWNTO 21], n);
mux2: df32tolmux
  PORT MAP (db, inst[20 DOWNTO 16], p);
mux3: df32tolmux
  PORT MAP (db, reg, q);
trds: dftrds
  PORT MAP (inst, trs0, trs1, reg);
END dfdbox_a;

Figure 4-17. VHDL Code of the DBOX Architecture
Target Register Dirty Select (TRDS)

Figure 4-18. Schematic Diagram of TRDS
Figure 4-19. Schematic Diagram of DBOX
### 4.3 ARITHMETIC LOGIC UNIT STAGE

The ALU is the third stage of the pipeline and has three units running in parallel. They are the Arithmetic Logic Unit Block (ALUB), the Multiplier/Divider Unit (MDU), and the ADD8 unit. The ALU stage is responsible for all arithmetic calculations. The ALUB handles addition, subtraction, shifting, comparing, and logical operations. The MDU handles multiplication and division operations. The ADD8 unit is responsible for calculating link addresses for *branch conditional* instructions. The ALU stage also contains the general register bank and the hi/lo register bank. The ALU diagram is shown in Figure 4-20.

The MDU and ADD8 were designed by Scott Siers and are discussed in "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" [19]. The ALUB and register banks are discussed in section 5.
Figure 4-20. ALU Stage Block Diagram
4.4 MEMORY STAGE

The MEM is the fourth stage of the pipeline and handles all accesses to memory. This stage is very similar to the IF stage with the following exceptions. The MEM stage only accesses memory on load and store instructions while the IF does so for every instruction. MEM accesses memory by reading and writing while IF only reads. The final difference is that MEM needs to filter data read from memory. The schematic drawing is shown in Figure 4-21.

When MEM encounters a non load or store instruction, it passes the instruction on to the next stage. However, when the instruction is a load or store, the MEM stage has to gain access to the busses to carry out the instruction. MEM sends a bus request to the BCU. Once the address and data busses are free, The BCU grants them to MEM. The BCU is discussed in more detail in section 4.6.

Part of the MEM-to-memory interface involves the write and opcode signals. The write signal specifies whether the memory access will be a read or write. When write is high the access is a write. MEM has a write signal because it is the only stage that writes to memory (the IF stage only has to read instructions from memory). The opcode signal specifies the type of store instruction. The five choices are store byte (sb), store halfword (sh), store word left (swl), store word (sw), and store word right (swr). The write and opcode signals are set to default values to allow proper operation of the IF stage. The write signal is set to '0' to specify a read operation. The opcode signal is set to "011" to specify a word.

The MEM stage consists of three main units. They are the memory decoder, the mask unit (MU), and the shift unit (SU). The memory decoder provides necessary instruction decoding used by MEM and WB. The decoder select lines include vbt, load, store, sus0-3, and mus0-3. The valid byte tag (vbt) signal is used by the register bank in the ALU stage. It specifies which bytes of data are valid. Only a valid byte is written
Figure 4-21. Schematic Diagram of MEM Stage
back to a register. The load and store signals specify whether the instruction is a load, store, or neither. The shift unit select (sus0-3) and mask unit select (mus0-3) signals go to the SU and MU, respectively. The MU determines the length of the requested data, and whether the data is sign-extended or zero-extended. The SU shifts data when it is not aligned on a word boundary. The VHDL code excerpts of the MU and SU are shown in Figures 4-22 and 4-23, respectively. For a more complete discussion, see "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" by Scott Siers [19].

```
ARCHITECTURE dfmu_a OF dfmu IS
  -- component and signal declarations
BEGIN
  done <= '1' AFTER 3.4 ns WHEN start = '1' ELSE '0' AFTER 1 ns;
  mux0: df2tolmux8 PORT MAP
  (se_byte, data(7 DOWNTO 0), mus0, data_out(7 DOWNTO 0));
  mux1: df2tolmux8 PORT MAP
  (se_byte, data(15 DOWNTO 8), mus1, data_out(15 DOWNTO 8));
  mux2: df2tolmux8 PORT MAP
  (se_byte, data(23 DOWNTO 16), mus2, data_out(23 DOWNTO 16));
  mux3: df2tolmux8 PORT MAP
  (se_byte, data(31 DOWNTO 24), mus3, data_out(31 DOWNTO 24));
  se_byte <= se_nibble & se_nibble;
  se_nibble <= seb & seb & seb & seb;
  seb <= '1' AFTER 2.1 ns WHEN -- a1
  (inst(28) = '0' AND data(15) = '1' AND addr = "10") OR
  -- a2
  (inst(28) = '0' AND data(31) = '1' AND addr = "00") OR
  -- a3
  (inst(28) = '0' AND data(7) = '1' AND addr = "11") OR
  -- a4
  (inst(28) = '0' AND data(23) = '1' AND addr = "01") ELSE
  '0' AFTER 2.1 ns;
END dfmu_a;
```

Figure 4-22. Code Excerpt of the MU Architecture
ARCHITECTURE dfsu_a OF dfsu IS

COMPONENT df4tolmux8
    PORT (i0: IN bit_8;
    i1: IN bit_8;
    i2: IN bit_8;
    i3: IN bit_8;
    s0: IN bit;
    s1: IN bit;
    o: OUT bit_8);
END COMPONENT;

BEGIN

    done <= '1' AFTER 1.4 ns WHEN start = '1' ELSE '0' AFTER 1 ns;

    mux0: df4tolmux8
        PORT MAP(mu_data(7 DOWNTO 0), mu_data(15 DOWNTO 8),
        mu_data(23 DOWNTO 16), mu_data(31 DOWNTO 24),
        sus0(0), sus0(1), data_out(7 DOWNTO 0));

    mux1: df4tolmux9
        PORT MAP(mu_data(7 DOWNTO 0), mu_data(15 DOWNTO 8),
        mu_data(23 DOWNTO 16), mu_data(31 DOWNTO 24),
        sus1(0), sus1(1), data_out(15 DOWNTO 8));

    mux2: df4tolmux9
        PORT MAP(mu_data(7 DOWNTO 0), mu_data(15 DOWNTO 8),
        mu_data(23 DOWNTO 16), mu_data(31 DOWNTO 24),
        sus2(0), sus2(1), data_out(23 DOWNTO 16));

    mux3: df4tolmux8
        PORT MAP(mu_data(7 DOWNTO 0), mu_data(15 DOWNTO 8),
        mu_data(23 DOWNTO 16), mu_data(31 DOWNTO 24),
        sus3(0), sus3(1), data_out(31 DOWNTO 24));

END dfsu_a;

Figure 4-23. Code Excerpt of the SU Architecture
4.5 WRITEBACK STAGE

The WB is the fifth and final stage of the pipeline. It writes the results back into the general purpose register bank or into the hi/lo register bank, depending on the instruction. The schematic diagram is shown in Figure 4-24. WB consists mainly of the decoder unit, another TRDS unit, and an encoder. The decoder unit decodes the trs0 and trs1 select lines used by the TRDS unit. The TRDS unit specifies which of the general purpose registers is the destination (if there is one). A six-bit encoding scheme, shown in Table 4-3, is used to determine the destination register. If the most significant bit (MSB) is '0', the least significant five bits selects one of the general purpose registers as the destination. If the MSB is '1', then the least significant bit (LSB) selects the hi/lo register. If the LSB is '0' then the destination is the hi register. The LSB equal to '1' specifies the lo register.

<table>
<thead>
<tr>
<th>BITS 5-0</th>
<th>DESTINATION REGISTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0rrrrr</td>
<td>general purpose register &quot;rrrrr&quot;</td>
</tr>
<tr>
<td>1xxxx0</td>
<td>hi register</td>
</tr>
<tr>
<td>1xxxx1</td>
<td>lo register</td>
</tr>
</tbody>
</table>

note: "rrrrr" = 5 bits to select which GPR
'x' = don't care

*Table 4-3. WB Stage Destination Register Bit Encoding Scheme*
Writeback Stage

Figure 4.24. Schematic Diagram of WB Stage
4.6 BUS CONTROL UNIT

The BCU controls access to the data and address busses. It acts as an arbiter between the IF and MEM stages. When the BCU receives bus requests from both, it grants access to one stage and blocks the other. The stage that was denied access has to wait until the stage that has access is done using the busses. The interface between the BCU and each stage consists of three signals: bus request (ir or mr), bus acknowledgment (ia or ma), and address load (il or ml). The signal names that start with an "i" indicate the IF stage. Those with an "m" indicate the MEM stage. The BCU also has to interface with the memory. This interface consists of a memory request (req) and memory acknowledgment (ack). These interfaces are shown in Figure 4-25.

![Figure 4-25. BCU Interfaces to IF, MEM, and Memory](image)

The BCU schematic diagram is shown in Figure 4-26. The circuit operation is as follows. A stage requests memory by setting its request signal high (ir for IF and mr for MEM). The BCU selects the stage that will get the busses by setting the appropriate
Bus Control Unit (BCU)

Figure 4-26. Schematic Diagram of Bus Control Unit (BCU)
acknowledgment signal (*ia* for IF and *ma* for MEM). This decision is made by polling the request lines at a high speed. Once a stage sees that it's acknowledgment line is high, it loads the appropriate values on the busses. Only the address is loaded for the IF stage and a MEM read operation. Both the address and data busses are loaded for a MEM write operation. Once the busses are loaded, the stage sets its load signal (*il* for IF and *ml* for MEM). This alerts the BCU that the busses are setup. The BCU can now perform the specified memory operation. The BCU sets the *req* signal high to send a memory request. The memory responds by setting the acknowledgment (*ack*) signal high. When memory sends the *ack* signal low, the memory operation is complete. The BCU can now reset the *req* line back to low. When the memory operation is complete, the BCU sends the appropriate acknowledgment signal low. This signals to the stage that the operation is over. The stage now resets its' request and load lines. The BCU waveforms are shown in Figure 4-27.

The *write* and *opcode* signals are only used by MEM. The IF does not need these signals because it only performs a load word operation. Therefore, MEM has to set these signals to their default settings for proper IF operation. The *write* signal specifies whether the memory operation is a read or a write. The default setting is a '0' which is a read operation. The *opcode* signal specifies the type of operation (*byte*, *halfword*, or *word*). The default setting is the type *word*.

The BCU is implemented using a finite state machine (FSM). The FSM has four states and is shown in Figure 4-28. State zero represents the BCU polling the IF stage. The IF is granted access to the busses if the IF's bus request line (*ir*) goes high during state zero. State one represents the BCU polling the MEM stage. The MEM is granted access if the *mr* signal goes high during state one. BCU continues to alternate between state zero and one at a high speed. This is called high speed polling. The other two states represent a stage having control of the bus. State two represents the IF and state three
Figure 4.27. BCU Waveforms
represents the MEM having been granted access to the busses. The FSM will stay in state two or three until the memory operation is complete. The VHDL code that implements the FSM is shown in Figure 4-29.

\[
\text{input/output = ir, mr, d/q1, q0}
\]

\[
\text{STATE} \quad \text{DESCRIPTION}
\]

<table>
<thead>
<tr>
<th>STATE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BCU polling IF stage, IF granted busses if it requests them</td>
</tr>
<tr>
<td>1</td>
<td>BCU polling MEM stage, MEM granted busses if it requests them</td>
</tr>
<tr>
<td>2</td>
<td>IF stage has control of busses</td>
</tr>
<tr>
<td>3</td>
<td>MEM stage has control of busses</td>
</tr>
</tbody>
</table>

Figure 4-28. BCU FSM State Diagram

Figure 4-29. VHDL Code that Implements the FSM
5.0 STRUCTURAL MODEL

The structural model builds on the previous dataflow model. It uses the same dataflow test bench and top level components (memory module, compare module, and CPU component). The only modifications are to the ALU stage of the CPU component. The two units that are modeled at the structural level are the general register bank and the arithmetic logic unit block (ALUB), located in the ALU stage.

5.1 GENERAL PURPOSE REGISTER BANK

The general purpose register (GPR) bank is composed of thirty-two 32-bit general registers used by the processor for temporary storage for operands and results. The dirty bits for each register are also found in the GPR bank. The GPR bank component is shown in Figure 5-1. The *start* signal is generated from the *alu_start* signal in the ALU stage. It is only used to signal when to set the dirty bits. The *db_sel* signal is a 5-bit line that selects which register is to be set dirty. This signal comes from the ID stage. The *write* signal comes from the WB stage and signals when to write back the result to a register. It also signals when to clear the dirty bits. The *reg_sel* signal is a 5-bit line that selects

![General Purpose Register Bank Component](image)

*Figure 5-1. General Purpose Register Bank Component*
which register is the destination register (which register is written back to). Once the register is updated with the result, it's dirty bit is cleared. The data signal is a 32-bit line that contains the data that is placed in the destination register. The valid byte tag (vbt) signal is a 4-bit line and is generated by the MEM stage. The vbt signal is used to determine what portion of the data is valid. The registers are logically divided into four bytes. Each bit of the vbt signal represents one of these bytes. Each byte of data is valid and can be written back when the corresponding vbt bit is high. The db signal is 32-bits wide and represents the 32 dirty bits, one for each register. The signal is used to stall the ID stage when a data dependency exists. The a and b signals are 32-bits wide and represent the a and b busses in the ALU stage. The a_sel and b_sel signals are 5-bits wide and select which register to put on the a and b busses, respectively. The components used in the GPR bank are shown in Figures 5-2, 5-3, and 5-4. Figure 5-2 is a transmission gate, Figure 5-3 is a 8-bit register, and Figure 5-4 is a 32-bit register. The GPR bank was designed by Kevin Johnson and a more detailed discussion can be found in "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" [10].

ARCHITECTURE sttg8_a OF sttg8 IS
BEGIN
  c <= 1 AFTER 0.1 ns WHEN en = '1' ELSE
  "ZZZZZZZZ" AFTER 0.1 ns;
END sttg8_a;

Figure 5-2. VHDL Code of an 8-bit Wide Transmission Gate

ARCHITECTURE streg8_a OF streg8 IS
COMPONENT sttg8
  PORT(i: IN slv_8;
  en: IN std_logic;
  c: OUT slv_6);
END COMPONENT;

SIGNAL id_bar: std_logic := '0';
SIGNAL ml: slv_8 := "00000000b";
SIGNAL m2: slv_8 := "00000000";
SIGNAL m3: slv_8 := "00000000";

BEGIN

ld_bar <= NOT ld;
input_tg: sttg8
PORT MAP(i, ld, m1);

m2 <= NOT m1 AFTER 0.3 ns;

m3 <= NOT m2 AFTER 0.3 ns;

feedback_tg: sttg8
PORT MAP(m3, ld_bar, m1);

a_bus_tg: sttg8
PORT MAP(m3, as, a);

b_bus_tg: sttg8
PORT MAP(m3, bs, b);

p <= m3;

END streg8 a;

Figure 5-3. VHDL Code of an 8-bit Register

ARCHITECTURE streg32 a OF streg32 IS

COMPONENT streg8
PORT(i: IN slv_8;
ld: IN std_logic;
as: IN std_logic;
bs: IN std_logic;
a: OUT slv_8;
b: OUT slv_8;
p: OUT slv_8);
END COMPONENT;

SIGNAL d0, d1, d2, d3: slv_8;
SIGNAL ld0, ld1, ld2, ld3: std_logic;
SIGNAL a0, a1, a2, a3: slv_8;
SIGNAL b0, b1, b2, b3: slv_8;
SIGNAL dbid, dbid: std_logic := '0';
SIGNAL p0, p1, p2, p3: slv_8;

BEGIN

d0 <= data(7 DOWNTO 0);
d1 <= data(15 DOWNTO 8);
d2 <= data(23 DOWNTO 16);
d3 <= data(31 DOWNTO 24);

ld0 <= vbt(0) AND ld AFTER 1 ns;
ld1 <= vbt(1) AND ld AFTER 1 ns;

END;
Figure 5-4. VHDL Code of a 32-bit Register
5.2 ARITHMETIC LOGIC UNIT BLOCK

The ALUB consists of ten components: bus control block (BCB), arithmetic logic unit component (ALUC), ALUC decoder, overflow block, compare block, branch control, shifter unit, shifter unit control, set-on-less-than (SLT) unit, and output selector. The ALUB block diagram is shown in Figure 5-5.

BUS CONTROL BLOCK

The BCB, shown in Figure 5-6, controls what is placed on the a and b busses in the ALUB. These busses are fed into the ALUC. The BCB consists of the A-bus selector, the B-bus selector, and the bus selection decoder.

The A-bus selector, shown in Figure 5-7, consists of a 32-bit 4-to-1 multiplexer. This multiplexer selects from the following three input values: a, pcl, and the constant of zero. The a signal comes from the a output of the GPR bank. This signal is used when the instruction needs a value for the first source field (bits 25-21 of the instruction). The a output of the GPR bank supplies this first source field. The pcl signal is the current value of the PC and comes from the ID stage. This signal is used for jump-and-link instructions. It is used to calculate the link address that is stored in the link register (the bcond link instructions use the ADD8 unit to calculate the link address). The constant value of zero is needed for shift instructions. Shift instructions utilize the shifter unit and do not use the ALUC. Therefore, the data that is operated on has to be passed to the shifter unit unchanged. This data is stored in the general registers and is supplied on the b output of the GPR bank. The a input to the ALUC has to be zero so that when it is added to the b input it does not change. The jump (j) and shift (s) signals select which multiplexer input to use.
Figure 5.5. Arithmetic Logic Unit Block (ALUB) Diagram
Figure 5-6. Bus Control Block (BCB) Diagram
Figure 5-7. The A-Bus Selector Component
The B-bus selector, shown in Figure 5-8, also consists of a 32-bit 4-to-1 multiplexer and logic for immediate or base-offset instructions, sign-extension, and zero-extension. This multiplexer selects from the following three input values: \( b \), \( \text{immed} \), and the constant of eight. The \( b \) signal comes from the \( b \) output of the GPR bank. This signal is used when the instruction needs a value for the second source field (bits 20-16 of the instruction). The \( b \) output of the GPR bank supplies this second source field. The \( \text{immed} \) signal is the immediate field of the instruction (bits 15-0 of the instruction) that is sign or zero-extended to 32 bits. The immediate instructions that need to be sign-extended are \( \text{addi} \), \( \text{addiu} \), \( \text{slti} \), and \( \text{sltiu} \). The immediate instructions that need to be zero-extended are \( \text{andi} \), \( \text{ori} \), and \( \text{xori} \). The \( \text{immed} \) input is chosen for an immediate instruction or for a base-offset calculation (here, the \( \text{immed} \) signal is the offset). The \( \text{load} \) and \( \text{store} \) instructions need the base-offset calculations. The constant value of eight is needed for link instructions. The link address is found by adding the value eight to the PC. The A-bus selector provides the PC to the \( a \) input of the ALUC. The B-bus selector provides the constant eight. The \( \text{jump} \) (\( j \)) and \( \text{ibo} \) signals select which multiplexer input to use.

The bus selection decoder, shown in Figure 5-9, provides the A-bus and B-bus selectors with the multiplexer select lines. The bus selection decoder generates the \( j \) select line when the instruction is either a \( \text{jump and link} \) (\( \text{jal} \)) or \( \text{jump and link register} \) (\( \text{jalr} \)). It also generates the \( s \) select line for all \( \text{shift} \) instructions.

**ARITHMETIC LOGIC UNIT COMPONENT**

The ALUC, is the component that does the addition, subtraction, and logical calculations. The ALUC also does base-offset address calculations. The component, shown in Figure 5-10, consists of eight ports. The \( a \) and \( b \) ports are 32 bits wide and correspond to the \( a \) and \( b \) outputs of the BCB. The four select lines, \( s0 \) through \( s3 \),
Figure 5-8. The B-Bus Selector Component
Figure 5-9. The Bus Selection Decoder Component
control what operation the ALUC performs. Table 5-1 lists these operations. The out signal is the output of the ALUC. The last port, c_out, is the carry output of the ALUC. The ALUC for the asynchronous version could not be modeled exactly after the R3000's ALUC. This was due to a lack of information on the R3000's internal gate design. However, a suitable design was found in the Fairchild Advanced Schottky TTL Data Book. More details can be found in Kevin Johnson's thesis [10].

![Arithmetic Logic Unit Component (ALUC)](image)

**Figure 5-10. Arithmetic Logic Unit Component (ALUC)**

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a MINUS b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a PLUS b</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a XOR b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a OR b</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a AND b</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a NOR b</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*Table 5-1. ALUC Operation Encoding*
ALUC DECODER

Since the details of the synchronous R3000's ALUC were not known, the asynchronous version ALUC control signals do not match the instruction encoding inherent in the R3000. The ALUC decoder is a circuit that is required to decode the instruction before the ALUC can operate on it. This circuit is shown in Figure 5-11. A more detailed discussion on the ALUC decoder can be found in [10].

OVERFLOW BLOCK

The overflow block component, shown in Figure 5-12, detects an overflow condition from the ALUC. The three instructions that check for an overflow condition are addi, add, and sub. The carry signal, c, from the ALUC is XORed with the bit 31 of the data signal. The data signal is the output of the ALUC.

COMPARE BLOCK

The compare block component, shown in Figure 5-13, compares the output of the ALUC with the value of zero. It is used in conjunction with the branch control unit, discussed in the next section. The three output ports, ltz, gtz, and eqz, correspond to the three conditions: less than zero, greater than zero, and equal to zero.

BRANCH CONTROL

The branch control component, shown in Figure 5-14, determines if a branch is taken. The three inputs ltz, eqz, and gtz, are generated by the compare component discussed above. The inst input is used to determine the branch instruction type. The four
Arithmetic Logic Unit Component Decoder

Figure 5-11. The Arithmetic Logic Unit Component (ALUC) Decoder
Figure 5-12. The Overflow Block Component
Figure 5-13. The Compare Block Component
Figure 5-14. The Branch Control Component
regular branch instructions are branch on equal (beq), branch on not equal (bne), branch on less than or equal to zero (blez), and branch on greater than zero (bgtz). The four branch conditional (bcond) instructions are branch on less than zero (bltz), branch on greater than or equal to zero (bgez), branch on less than zero and link (bltzal), and branch on greater than or equal to zero and link (bgezal). The branch control output, cc, is the condition code and is high when the branch is taken.

**SHIFTER UNIT**

The shifter unit component, shown in Figure 5-15, provides the microprocessor with arithmetic and logical shift operations. The \( l_r \) signal controls the direction of the shift (left or right). The \( l_a \) signal controls the type of shift (logical or arithmetic). The five selector bits, \( s_0 \) through \( s_4 \), control the amount of bits to shift. The \( \text{in} \) and \( \text{out} \) signals correspond to the input and output of the shifter unit, respectively. A more complete discussion of the shifter unit can be found in Kevin Johnson's Thesis [10].

![Figure 5-15. The Shifter Unit Component](image-url)
SHIFTER UNIT CONTROL

The shifter unit control component, shown in Figure 5-16, provides the necessary signal inputs to the shifter. It generates the lr, la, and the five shift amount select lines (s0 through s4). The a input is used in conjunction with the variable shifting instructions, *shift left logical variable* (sllv), *shift right logical variable* (srlv), and *shift right arithmetic variable* (srav). The variable instructions have their variable shift field in bits 25-21 of the instruction. This shift field is placed on the A-bus. The a input is the lower five bits of the A-bus from the BCB.

SET ON LESS THAN UNIT

The set on less than unit component, shown in Figure 5-17, implements the four instructions, *set on less than* (slt), *set on less than unsigned* (sltu), *set on less than immediate* (slti), and *set on less than immediate unsigned* (sltiu). It consists of a 1-bit multiplexer, a 31-bit tri-state buffer, and decoding logic. All slt instructions output a value of one when the first source register (rs) is less than the second source register (rt). Otherwise they output a zero. For a slt instruction, the tri-state buffers are activated and the top 31 bits are set to zero. Bit zero is determined by the less than zero (ltz) signal. For non slt instructions, the tri-state buffers are deactivated and the input (in) passes to the output (out) unchanged.

OUTPUT SELECTOR

The output selector component, shown in Figure 5-18, is the final component in the ALUB. Its task is to route the proper lines to the ALUB output port. The output selector chooses from the following three inputs: *alu, hilo, or add8*. The alu signal input
Figure 5-16. The Shifter Unit Control Component
Figure 5-17. The Set On Less Than Unit Component
Figure 5.18. The Output Selector Component
comes from the output of the component chain in the ALUB. The hilo signal is used for the move from hi (mfhi) and move from lo (mflo) instructions. The add8 signal is used for the two branch conditional link instructions, branch on less than zero and link (bltzal) and branch on greater than or equal to zero and link (bgezal).
6.0 RESULTS

All three models were tested using a VHDL test bench. The test bench setup is shown in Figure 1-1. The behavioral model used the test bench shown in Figure 3-1. The dataflow and structural models used the test bench shown in Figure 4-1. Both test benches consists of three modules, CPU, memory, and compare. The memory module stores the test program used in the test bench. The compare module stores the expected results file. The output of the CPU module is compared against the expected results file after each instruction is executed.

6.1 TESTING PROCEDURE

The testing process consists of writing the test program, assembling the test program, generating an expected results file, processing the expected results file, preprocessing the test program and expected results file, loading the model into the digital simulator, and running the model. The model runs to completion if there are no discrepancies between the model and expected results. However, the user is warned if a discrepancy exists.

The test program can be written using any text editor. The programs are written in a pseudo MIPS assembly language. The program is saved with a "test" extension. An example of a filename is "filename.test" where filename is the name of the file.

The test program is assembled by the MIPS assembler (MASS) program, found in Appendix D. This assembler generates the machine code that the model can understand. The generated file is saved with a "m" extension which stands for the word machine. An example of this is "filename.m" where filename is the same name used as above.

The expected results file is generated using the MIPS expected results assembler (MERA) program, found in Appendix D. The user inputs the expected state of the
processor after every instruction into the assembler. This file is then used by the compare module. The file is saved with an "e" extension which stands for the word expected. An example of this is "filename.e".

The expected results file has to be modified for test programs with branches and jumps. The FLOW program was written to reorder such programs to follow the actual program flow. The instructions in the test program can now be compared to the reordered expected results file. The reordered expected results file is saved with a "f" extension. An example of this is "filename.f". The original expected results file, "filename.e", is unchanged.

The test files are preprocessed with the MIPS preprocessor (MPP) program. The preprocessor prepares the test files so the model can use them. The MPP program copies the appropriate test files to two files that the model opens. These two files are called "machine" and "expected" which represent the machine code file loaded into the memory module and the expected results file loaded into the compare module, respectively. The MPP program copies "filename.m" to "machine" and "filename.e" to "expected".

The model is loaded into the digital simulator. The name of the digital simulator is Quicksim and is manufactured by Mentor Graphics Corporation. The two test files, "machine" and "expected", need to be in the same directory as the model's design file. Quicksim is also invoked from this directory.

Running the model in Quicksim consists of setting up the simulator, setting some signal values, and running the simulator for some set time. Setting up the simulator involves opening the VHDL code windows, tracing pertinent signals, and opening list and monitor windows. When the signals are forced to specific values, they can be traced on the screen. The sys_control_sig signal is used to control the model. This signal is forced to a value of load for 100 nanoseconds (ns) and then forced to a value of run. The load value mode signals the test bench to load the test program ("machine" file) into the
memory module and the expected results file ("expected" file) into the compare module. The run value mode signals to the test bench to run the model for testing.

The models use VHDL ASSERT statements to alert the user if there is a problem with the simulation. If there are no problems and the state of the processor matches the expected results, then the model will run to completion. The user is only warned when an error or discrepancy exists.

The test benches were used throughout the entire design process. During the behavioral modeling as the instructions were coded, the test bench was used to test whether the model was performing up to specifications. During the dataflow modeling, the test bench was used first to verify the arbitrary simulation times and then used again for the back annotated times from the Accusim circuit simulation runs. All models were verified using their appropriate test benches.

6.2 DELAY TIMES

This section shows the gate, component and stage delay times that were back annotated into the model from Accusim circuit simulations. These times were taken from Kevin Johnson's thesis [10] and Scott Siers' thesis [19]. These circuit simulations are found in their separate documents.

GATE DELAY TIMES

Circuit simulations were performed on the individual gates. Table 6-1 shows these times. Delay times of complex circuits that are made up of these gates were calculated by determining the critical path and then adding up the times.
<table>
<thead>
<tr>
<th>GATE</th>
<th>DELAY TIME (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverter</td>
<td>0.3</td>
</tr>
<tr>
<td>NAND</td>
<td>0.7</td>
</tr>
<tr>
<td>NOR</td>
<td>1.0</td>
</tr>
<tr>
<td>AND</td>
<td>1.0</td>
</tr>
<tr>
<td>OR</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table 6-1. Gate Delay Times

COMPONENT DELAY TIMES

Primitive components delay times were calculated by simulating the component. More complex components are made up of primitive components. Their delay times were calculated by determining the critical path and adding up the times. This provides a worst case value. Table 6-2, shown below, shows the various component delay times used.

<table>
<thead>
<tr>
<th>COMPONENT FILENAME</th>
<th>COMPONENT</th>
<th>DELAY TIME (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>df2to1mux</td>
<td>1-bit 2 to 1 mux</td>
<td>0.3</td>
</tr>
<tr>
<td>df2to1mux16</td>
<td>16-bit 2 to 1 mux</td>
<td>0.3</td>
</tr>
<tr>
<td>df2to1mux3</td>
<td>3-bit 2 to 1 mux</td>
<td>0.3</td>
</tr>
<tr>
<td>df2to1mux30</td>
<td>30-bit 2 to 1 mux</td>
<td>0.3</td>
</tr>
<tr>
<td>df2to1mux32</td>
<td>32-bit 2 to 1 mux</td>
<td>0.3</td>
</tr>
<tr>
<td>df2to1mux8</td>
<td>8-bit 2 to 1 mux</td>
<td>0.3</td>
</tr>
<tr>
<td>df32to1mux</td>
<td>1-bit 32 to 1 mux</td>
<td>3.6</td>
</tr>
<tr>
<td>df4to1mux</td>
<td>1-bit 4 to 1 mux</td>
<td>0.4</td>
</tr>
<tr>
<td>df4to1mux10</td>
<td>10-bit 4 to 1 mux</td>
<td>0.4</td>
</tr>
<tr>
<td>df4to1mux16</td>
<td>16-bit 4 to 1 mux</td>
<td>0.4</td>
</tr>
<tr>
<td>df4to1mux32</td>
<td>32-bit 4 to 1 mux</td>
<td>0.4</td>
</tr>
<tr>
<td>df4to1mux4</td>
<td>4-bit 4 to 1 mux</td>
<td>0.4</td>
</tr>
<tr>
<td>df4to1mux8</td>
<td>8-bit 4 to 1 mux</td>
<td>0.4</td>
</tr>
<tr>
<td>dfaa</td>
<td>address adder</td>
<td>8</td>
</tr>
<tr>
<td>dfadd8</td>
<td>add8 unit</td>
<td>26</td>
</tr>
<tr>
<td>dfalu</td>
<td>ALU stage</td>
<td>see Table 6-3</td>
</tr>
<tr>
<td>dfalu32</td>
<td>ALU component (ALUC)</td>
<td>10</td>
</tr>
<tr>
<td>dfalublk</td>
<td>ALU block (ALUB)</td>
<td>30.1</td>
</tr>
<tr>
<td>dfaludec</td>
<td>ALU decoder</td>
<td>2.5</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
<td>-------</td>
</tr>
<tr>
<td>dfasel</td>
<td>A-bus selector</td>
<td>0.4</td>
</tr>
<tr>
<td>dfbc</td>
<td>bus controller</td>
<td></td>
</tr>
<tr>
<td></td>
<td>internal clock used for FSM high speed poller: 4</td>
<td></td>
</tr>
<tr>
<td>dfbctl</td>
<td>branch control</td>
<td>2</td>
</tr>
<tr>
<td>dfbjbox</td>
<td>branch and jump box</td>
<td>2.1</td>
</tr>
<tr>
<td>dfbsel</td>
<td>B-bus selector</td>
<td>2.0</td>
</tr>
<tr>
<td>dfbusctl</td>
<td>bus control block</td>
<td>6.2</td>
</tr>
<tr>
<td>dfbusdec</td>
<td>bus selection decoder</td>
<td>3.3</td>
</tr>
<tr>
<td>dfcomp</td>
<td>compare block</td>
<td>4.4</td>
</tr>
<tr>
<td>dfcompare</td>
<td>test bench compare module</td>
<td>N/A - (behavioral)</td>
</tr>
<tr>
<td>dfcpu</td>
<td>test bench CPU module</td>
<td></td>
</tr>
<tr>
<td></td>
<td>see Table 6-3</td>
<td></td>
</tr>
<tr>
<td>dfdbox</td>
<td>dirty box</td>
<td>3.1</td>
</tr>
<tr>
<td>dfeh</td>
<td>exception handler</td>
<td>1.5</td>
</tr>
<tr>
<td>dffed</td>
<td>falling edge detector</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delay time: 0.4 pulse width: 1.2</td>
<td></td>
</tr>
<tr>
<td>dfhcc</td>
<td>handshaking control circuit</td>
<td>5.7</td>
</tr>
<tr>
<td>dfhlreg</td>
<td>hi/lo register bank</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write: 0.3 read: 0.9</td>
<td></td>
</tr>
<tr>
<td>dfid</td>
<td>ID stage</td>
<td></td>
</tr>
<tr>
<td></td>
<td>see Table 6-3</td>
<td></td>
</tr>
<tr>
<td>dfif</td>
<td>IF stage</td>
<td></td>
</tr>
<tr>
<td></td>
<td>see Table 6-3</td>
<td></td>
</tr>
<tr>
<td>dfinstdec</td>
<td>ID instruction decoder</td>
<td>4.4</td>
</tr>
<tr>
<td>dfmd</td>
<td>multiplier/divider</td>
<td>1005</td>
</tr>
<tr>
<td>dfmem</td>
<td>MEM stage</td>
<td></td>
</tr>
<tr>
<td></td>
<td>see Table 6-3</td>
<td></td>
</tr>
<tr>
<td>dfmemdec</td>
<td>MEM stage decoder</td>
<td>5.3</td>
</tr>
<tr>
<td>dfmemory</td>
<td>test bench memory module</td>
<td>memory speed: 50 handshaking overhead: 5 total delay time: 55</td>
</tr>
<tr>
<td>dfmu</td>
<td>mask unit</td>
<td>3.4</td>
</tr>
<tr>
<td>dfoutsel</td>
<td>output selector</td>
<td>2.7</td>
</tr>
<tr>
<td>dfovrf</td>
<td>overflow block</td>
<td>4.6</td>
</tr>
<tr>
<td>dfred</td>
<td>rising edge detector</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delay time: 0.3 pulse width: 1.4</td>
<td></td>
</tr>
<tr>
<td>dfreg</td>
<td>1-bit register</td>
<td>0.9</td>
</tr>
<tr>
<td>dfreg32</td>
<td>32-bit register</td>
<td>0.9</td>
</tr>
<tr>
<td>dfreg4</td>
<td>4-bit register</td>
<td>0.9</td>
</tr>
<tr>
<td>dfreg5</td>
<td>5-bit register</td>
<td>0.9</td>
</tr>
<tr>
<td>dfregbank</td>
<td>thirty-two 32-bit general purpose registers</td>
<td>write: 0.3 read: 0.9</td>
</tr>
<tr>
<td>dfregr</td>
<td>1-bit register with reset</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>0.3 for reset</td>
<td></td>
</tr>
<tr>
<td>dfrslat</td>
<td>set-reset latch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set time: 1.0 reset time: 0.4</td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Delay (ns)</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>dfsectl</td>
<td>shifter unit control</td>
<td>2.7</td>
</tr>
<tr>
<td>dfshift</td>
<td>shifter unit</td>
<td>3.3</td>
</tr>
<tr>
<td>dfslt</td>
<td>set on less than unit</td>
<td>3.5</td>
</tr>
<tr>
<td>dfsu</td>
<td>shift unit</td>
<td>1.4</td>
</tr>
<tr>
<td>dftrds</td>
<td>target register dirty select</td>
<td>0.4</td>
</tr>
<tr>
<td>dfstsb32</td>
<td>32-bit tri-state buffer</td>
<td>0.3</td>
</tr>
<tr>
<td>dfwb</td>
<td>WB stage</td>
<td>see Table 6-3</td>
</tr>
</tbody>
</table>

*Table 6-2. Component Delay Times*

**STAGE DELAY TIMES**

The stage delay times vary depending on the instruction executed. Table 6-3 shows the delay times for each pipeline stage for various instructions. The times of all the stages are added together to obtain the time that it takes to execute a particular instruction type. This is the CPU processing time. The HCC overhead is taken into account.

<table>
<thead>
<tr>
<th>STAGE</th>
<th>DELAY TIMES (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ALU REG</td>
</tr>
<tr>
<td>IF</td>
<td>69</td>
</tr>
<tr>
<td>ID</td>
<td>15</td>
</tr>
<tr>
<td>ALU</td>
<td>35</td>
</tr>
<tr>
<td>MEM</td>
<td>11</td>
</tr>
<tr>
<td>WB</td>
<td>9</td>
</tr>
<tr>
<td>CPU *</td>
<td>162</td>
</tr>
</tbody>
</table>

* HCC overhead for each stage included

*Table 6-3. Stage Delay Times*

**6.3 EXAMPLE SIMULATIONS**

This section consists of five examples, each corresponding to the instruction type in Table 6-3. The examples are taken from a few instructions of the test programs found
in Appendix E. The times shown in Table 6-3 came from these example simulations. The five instruction groups are as follows: *ALU register, branch, jump, load, and multiplication.*

The first example is taken from instructions 4, 5, and 6 of the arithmetic register test program ("ar.test"). Instruction 4 (PC equal to '0C') loads register 1 with the value 'A'. Instruction 5 loads register 2 with the value '5'. Instruction 6 adds registers 1 and 2 together and then places the result in register 3. The waveforms are shown in Figure 6-1.

The second example is taken from instructions 38, 39, and 40 of the jump and branch test program ("jb.test"). Instruction 38 (PC equal to '94') loads register 2 with the value 'ABCD'. Instruction 39 loads register 3 with the value 'ABCD'. Instruction 40 compares instructions 38 and 39, and branches to the destination address (5 instructions after the delay slot) if they are equal. The waveforms are shown in Figure 6-2.

The third example is taken from the first instruction of the jump and branch test program ("jb.test"). Instruction 1 (PC equal to '00') jumps to the destination address unconditionally with a delay of one instruction. The destination address is calculated by shifting the 26-bit target address left two bits and combining it with the high order 4 bits of the PC. The waveforms are shown in Figure 6-3.

The fourth example is taken from instruction 21 of the load and store test program ("ls.test"). Instruction 21 (PC equal to '50') loads register three with the contents of memory location '4000'. This value will be available for use after a delay of one instruction. The waveforms are shown in Figure 6-4.

The fifth and final example is taken from instructions 1, 2, and 3 of the multiplication and division test program ("md.test"). Instruction 1 (PC equal to '00') loads register 1 with the value '8'. Instruction 2 loads register 2 with the value '9'. Instruction 3 multiplies register 1 and 2 together and places the results in the *hi* and *lo* registers. The waveforms are shown in Figure 6-5.
Figure 6.1. Example 1 - Arithmetic Register Instruction Waveforms
Figure 6.1 Continued
Figure 6-3. Example 3 - Jump Instruction Waveforms
Figure 6.4. Example 4 - Load Instruction Waveforms
Figure 6.4 Continued

| jle/dfcpu/pc | 00000050 | 000000654 | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | 00000058 | + | + |
| /addr_bus   | +        | 00000000 | 000004827 | + | + | + | + | + | + | + | + | + | + | + | + | + | + | 00000000 | + | 000004800 |
| /data_bus   | +        | 00000000 | 667BE47F  | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | ABCDEF01 | + | + |
| egbank/reg1 |          |           |           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| egbank/reg2 | 567BE47F | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| egbank/reg3 | 00000000 | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |

| j/if_bus_req| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| j/if_bus_ack| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| if_load_addr| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| /mem_bus_req| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| /mem_bus_ack| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| mem_load_addr|           |           |           |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| :pu/if_start| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| fcpu/if_done| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| :pu/id_start| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| fcpu/id_done| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| :alu,start  | +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| :alu/done   | +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| :mem,start  | +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| :mem/done   | +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| :wb,start   | +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |
| fcpu/wb_done| +        | +         | +         | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + | + |

| Time(ns)    | 3087.0   | 3111.5   | 3136.0   | 3160.5   | 3185.0   | 3209.5   | 3234.0   | 3258.5   | 3283.0   | 3307.5   | 3332.0   |

Figure 6.4 Continued
Figure 6.5. Example 5 - Multiplication Instruction Waveforms
6.4 PROCESSOR SIMULATION TIMES

A program was written to calculate the processor execution time, the processor speed rating in millions of instructions per second (MIPS), and the actual execution time on the Quicksim. This program, shown in Figure 6-6, consists of a ten instruction loop. Register 1 (r1) is set to a value of 1000 (3e8 hex). The value is decremented with each iteration of the loop. The loop is exited when the value in r1 equals zero.

![Program Code]

*Figure 6-6. Program to Calculate Processor Execution Times*

The MIPS rating is calculated by dividing the number of instructions executed by the simulation time. The actual execution time (how long it took Quicksim to run this program) is calculated using a wrist watch and viewing the trace window. The data and calculations are shown below.

Number of instructions executed: 1000

Total processor execution time (ns): 102,420

MIPS = \( \frac{1000 \text{ inst}}{102420 \text{ ns}} \times 10e9 \text{ ns} / 1 \text{ second} \) = 9.8

Actual execution time (minutes): 7.5
7.0 CONCLUSIONS

This thesis has shown that an asynchronous design of a microprocessor is a viable alternative to synchronous design. It has also shown that a VHDL top down design methodology is an effective approach to modern circuit design. Conclusions can also be drawn from the results of the model testing.

The VHDL dataflow model achieved an average speed rating of 9.8 MIPS, as shown in section 6.4. Any comparisons to the synchronous version are difficult to make due mainly to two reasons. First, the actual speed of the asynchronous processor is variable and depends on which instructions are executed. Secondly, the memory was not implemented in the asynchronous version.

The testing procedure using the VHDL test bench is easy to use and provides a fast, accurate, and efficient means to test the processor. Back annotating delay times from Accusim circuit simulations proved an effective way to design. It was easy to modify the design by changing the delay times and recompiling the source code. If a problem occurred during execution then the code could be stepped through with the VHDL debugger. Lastly, the example simulations proved that the models worked and that the asynchronous processor was designed and working to specifications.

The first advantage of the asynchronous design methodology is the elimination of the global clock. In synchronous design, the global clock is routed to all areas of the chip. This is done to synchronize all the circuit modules. These metal clock lines become very long since they are routed from one side of the chip to the other. Due to the different lengths of metal lines and different capacitive loads, one area of the chip may receive the clock signal earlier or later than another area, creating clock skew. Clock skew becomes more of a problem with this type of design as circuits become more dense. This is not a problem in asynchronous design because the global clock is eliminated.
The second advantage of asynchronous design is that every module can operate at maximum speed. In synchronous design, the clock cycle time is dictated by the slowest module or the longest instruction. The synchronous processor has to be designed to handle the worst case operation. This causes most instructions to use only part of the clock cycle. This leads to long periods of idle time for the processor. This is not the case in asynchronous design. Since this type of design is based on events, when a stage or module is complete it signals the next module which reduces or eliminates idle time between modules. A module can go as fast as a previous module can send data and the next module can receive data.

The third advantage of asynchronous design is that every component is modular in construction and use. Start and completion signals are inherent in asynchronous design. A start signal is sent to the module to begin processing. When a module is finished it sends a completion signal to the next module. This allows every component or module to be replaced and/or redesigned at any time. The only requirement is that input and output signals match (i.e. the "black box" approach). This modular concept is most helpful for specifying memory requirements. Memory of any speed can be used in asynchronous design. The memory interface will wait until the memory sends an acknowledgment that it is finished.

One disadvantage of asynchronous design is that events cannot be predicted. Asynchronous design is well suited for system designs that only require connections between adjacent modules. However, multi-connection modules are more difficult to coordinate and circuit complexity is increased. An example that makes this apparent is a branch instruction. Three stages of the pipeline (IF, ID, and ALU) need to be coordinated to execute a branch instruction. The IF stage has to know which instruction to execute next (i.e. whether or not the branch is taken). The ID stage has to calculate the branch destination address. The ALU stage determines whether or not the branch is taken. The synchronous version implements these tasks by using a two-phase clock. The IF can wait
until the second phase of the clock to perform an instruction fetch. This allows the ID to determine the state of the branch. However, in the asynchronous version, the IF has to stall until the other stages have done their tasks. Since every action is event based and not time based, it is not known when these tasks will start and finish. Controlling circuitry has to be added to the design to enforce an order of operations.

Another disadvantage of asynchronous design occurs when a completion signal is generated. The two methods used are dummy delay and differential cascade voltage switch logic (DCVSL). These issues are discussed in Kevin Johnson's thesis [10] and Scott Siers' thesis [19].

The flexibility of VHDL allows a seamless transition through different levels of abstraction. VHDL provides the designer a way of writing different model abstractions with the same familiar constructs. The high level constructs allow describing the behavior or function of a system and is very much like a typical programming language. It also provides a means of writing a test bench. The low level constructs allow the structure or gate level of a system to be modeled with precise timing information. If no particular type of model is adequate, VHDL allows the designer to mix different types of modeling styles.

Top down design provides the designer with a method to guide a design through all levels of abstraction. A behavioral model, which projects a high level of abstraction, frees the designer of implementation details and allows for concentration on the basic system functionality. Once the behavior is defined, then more precise models can be written describing lower level details. As timing information is annotated into the models, the models are verified for correct operation. A test bench provides a fast, accurate, and efficient means to accomplish verification. This constant switching between designing and testing allows the designer to progress at a reasonable fashion without going too far with a bad design. The designer is alerted to a possible design flaw in the early stages of design. At this point the designer could even go back to the behavioral model and reassess its
functionality. The actual board or component is not produced until the design process is complete, which saves time and money.

One general problem with this thesis was the decision to take a synchronous processor and convert it to an asynchronous design. Some of the instructions in the R3000's instruction set were not well suited to an asynchronous approach. As an example consider the two branch and link instructions. These instructions were very complicated because they have to do all the tasks associated with a branch (branch address calculation and branch determination) and also store a return address in the link register. Since both adders are busy (the AA is busy calculating the branch address and the ALUC determines if the branch is taken), another adder (add8) was designed to handle the link address calculation. All these tasks are done in parallel to prevent the pipeline from stalling. It would have been better to design a processor to match the strengths of asynchronous design.

The requirement that each thesis be an independent work was a major disadvantage to the concept of a team project. Unfortunately, the team members divided the project in a manner which created separate pieces that were difficult to work on concurrently. This ultimately limited the effectiveness of the team concept. All VHDL modeling should have been done before any circuit design and layout was attempted. Throughout the entire design cycle, the VHDL modeling and simulation caught many design errors that were not apparent at first to the individual designers. These design errors were substantial enough to force complete redesigns which cost the designers a great deal of time and effort.

Overall, this thesis was a tremendous learning experience. The team concept provided the vehicle to tackle this large project. This thesis has shown the benefits of asynchronous and top down design methodologies. VHDL has proved to be an indispensable tool for design and development of an electronic system.


APPENDIX A - FILE STRUCTURE

vhdl

my_packages

bhpack

behavioral package

dfpack

dataflow package

stpack

structural package

bh_comp

behavioral components

df_comp

dataflow components

st_comp

structural components

mips

behavioral model test bench

dfmips

dataflow model test bench

stmips

structural model test bench

LEGEND:

DIRECTORY

DESIGN OR PACKAGE

TEXT FILE

PROGRAM

MAPS.lmf

logical mapping file

filename.test

test program

filename.m

machine code file

filename.e

expected results file

MASS

MIPS Assembler

MERA

MIPS Expected Results Assembler

FLOW

Flow Control Program

MPP

MIPS Preprocessor

A-1
--- library and use clauses
library mips_portable, ieee;
use mips_portable.gal_logic.all;
use mips_portable.gal_relations.all;
use ieee.std_logic_1164.all;

-- use std.textio.all;
library my_packages;
use my_packages.package._library;
library my_components;
use my_components.cpu.all;
use my_components.memory.all;
use my_components.compare.all;

entity mips is
  end mips;

architecture mips_a of mips is

component memory
  port(mem_control_sig: in mem_control_type;
       addr_bus: in bit_30;
       mem_ack_sig: out mem_ack_type;
       mem_exception_type: out question_type;
       data_bus: inout bus_bit_32 bus;
       compare_control_type;
       compare_ack_type: out compare_type);
  end component;

  component cpu
    port(sys_control_type;
         mem_control_type;
         mem_ack_type;
         mem_exception_type;
         mem_control_type;
         addr_bus: out bit_30;
         addr_bus_lo: out bit_2;
         data_bus: inout bus_bit_32 bus;
         compare_control_type;
         compare_ack_type: out compare_type);
  end component;

  component compare
    port(compare_control_type;
         compare_ack_type: in question_type);
  end component;

begin
  mem: memory
    port map(mem_control_sig, addr_bus, addr_bus_lo, mem_ack_sig, mem_exception_sig, data_bus);
  proc: cpu
    port map(sys_control, mem_control, mem_exception, mem_control, addr_bus, addr_bus_lo, data_bus, compare_control, compare_ack);
  comp: compare
    port map(compare_control, pc, r1, r2, r3, r4, r5, hi, lo, mem, cause, compare_ack);
  end mips_a;

end component;

entity cpu is
  port(sys_control_type;
       mem_control_type;
       mem_ack_type;
       mem_exception_type;
       mem_control_type;
       addr_bus: out bit_30;
       addr_bus_lo: out bit_2;
       data_bus: inout bus_bit_32 bus;
       compare_control_type;
       compare_ack_type: out compare_type);
  end component;

begin
  mem: memory
    port map(mem_control_type, addr_bus, addr_bus_lo, mem_ack_type, mem_exception_type, data_bus);
  proc: cpu
    port map(sys_control_type, mem_control_type, mem_exception_type, mem_control_type, addr_bus, addr_bus_lo, data_bus, compare_control_type, compare_ack_type);
  comp: compare
    port map(compare_control_type, pc, r1, r2, r3, r4, r5, hi, lo, mem, cause, compare_ack);
  end cpu;

end CPU;
```
architecture cpu_a of cpu is

-- signals for processor

signal inst: inst_type; -- instruction mnemonic

--signal pc: bit_32; -- program counter

--signal r6: bit_32; -- registers r6 thru r31
--signal r2: bit_32;
--signal r3: bit_32;
--signal r4: bit_32;
--signal r5: bit_32;
--signal r8: bit_32;
--signal r9: bit_32;
--signal r10: bit_32;
--signal r11: bit_32;
--signal r12: bit_32;
--signal r13: bit_32;
--signal r14: bit_32;
--signal r15: bit_32;
--signal r16: bit_32;
--signal r17: bit_32;
--signal r18: bit_32;
--signal r19: bit_32;
--signal r20: bit_32;
--signal r21: bit_32;
--signal r22: bit_32;
--signal r23: bit_32;
--signal r24: bit_32;
--signal r25: bit_32;
--signal r26: bit_32;
--signal r27: bit_32;
--signal r28: bit_32;
--signal r29: bit_32;
--signal r30: bit_32;
--signal r31: bit_32;

--signal hi: bit_32; -- hi, lo registers used for mult and div

--signal lo: bit_32;

signal exception_sig: exception_type;

--signal epc: bit_32; -- exception handling

--signal cause: bit_32;

begin

processor: process

begin

--variables for control

variable run_mode_flag: question_type;

variable delay_slot_flag: delay_slot_type;

variable latency_flag: delay_slot_type;

variable left_flag: delay_slot_type;

variable right_flag: delay_slot_type;

variable exception_flag: exception_type;

variable bf_flag: question_type;

end

--variables for storage

variable pc_reg: bit_32;

variable pc_temp: bit_32;

variable hi_reg: bit_32;

variable lo_reg: bit_32;

variable inst_type: bit_32;

variable r6: bit_32;

variable r2: bit_32;

variable r5: bit_32;

variable cause: bit_32;

variable epc_reg: bit_32;

variable current_inst: bit_32;

variable exception_flag: bit_32;

variable variable variable delay_slot_type variable run_mode_flags

variable right_flag: bit_32;

variable base: bit_32;

variable pc_reg: bit_32;

variable target: bit_32;

variable temp_reg: bit_32;

variable epc_reg: bit_32;

variable question_type)

variable exception_type;

variable exception_type;

variable exception_type;

variable exception_type;

variable run_mode_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exception_flag: bit_32;

variable exceptionフラグ := addr_load;

procedure mem_read(addr: in bit_32; size: in size_type);

begin

addr_bus := addr(31 downto 2) after delay;

addr_bus_1o := addr(1 downto 0) after delay;

case size is

when byte =>

mem_control_sig <= read_b after delay;

when ubyte =>

mem_control_sig <= read_ub after delay;

when halfword =>

mem_control_sig <= read_h after delay;

when word =>

mem_control_sig <= read_w after delay;

end case;

wait until mem_ack_sig = yes;

result := data_bus;

mem_control_sig <= reset after delay;

if mem_exception_sig = yes then

exception_flag := addr_load;

end
```
```
end if;
end mem_read;

procedure mem_write(addr in bit_32; size in size_type;
                      data in bit_32) is
begin
  addr_bus <= addr(3) downto 2) after delay;
  addr_bus_lo <= addr(1 downto 0) after delay;
  data_bus <= data after delay;
  case size is
    when byte =>
      mem_control_sig <= write_b after delay;
    when halfword =>
      mem_control_sig <= write_hw after delay;
    when word =>
      mem_control_sig <= write_w after delay;
    when lefty =>
      mem_control_sig <= write_l after delay;
    when righty =>
      mem_control_sig <= write_r after delay;
    when others =>
      null;
  end case;
  wait until mem_ack_sig = yes;
  mem_control_sig <= reset after delay;
  wait until mem_ack_sig = no;
  data_bus <= null after delay;
  if mem_exception_sig = yes then
    exception_flag := addr_store;
  end if;
end mem_write;

begin
  wait on sys_control_sig; -- entry into processor process
  case sys_control_sig is
  when reset =>
    run_mode_flag := no;
  when stop =>
    run_mode_flag := no;
  when others =>
    null;
  end case;
  wait until mem_ack_sig = yes;
  mem_control_sig <= reset after delay;
  wait until mem_ack_sig = no;
  data_bus <= null after delay;
  if mem_exception_sig = yes then
    exception_flag := addr_store;
  end if;
end if;
end mem_write;

reg(7) := x"0000_0000";
reg(8) := x"0000_0000";
reg(9) := x"0000_0000";
reg(10) := x"0000_0000";
reg(11) := x"0000_0000";
reg(12) := x"0000_0000";
reg(13) := x"0000_0000";
reg(14) := x"0000_0000";
reg(15) := x"0000_0000";
reg(16) := x"0000_0000";
reg(17) := x"0000_0000";
reg(18) := x"0000_0000";
reg(19) := x"0000_0000";
reg(20) := x"0000_0000";
reg(21) := x"0000_0000";
reg(22) := x"0000_0000";
reg(23) := x"0000_0000";
reg(24) := x"0000_0000";
reg(25) := x"0000_0000";
reg(26) := x"0000_0000";
reg(27) := x"0000_0000";
reg(28) := x"0000_0000";
reg(29) := x"0000_0000";
reg(30) := x"0000_0000";
reg(31) := x"0000_0000";
epc_reg := x"0000_0000";
cause_reg := x"0000_0000";
-- load program into memory
when load =>
  run_mode_flag := no;
  mem_control_sig <= load after delay;
-- memory handshake
  wait until mem_ack_sig = yes;
  mem_control_sig <= reset after delay;
  wait until mem_ack_sig = no;
  compare_control_sig <= load after delay;
-- compare handshake
  wait until compare_ack_sig = yes;
  compare_control_sig <= reset after delay;
  wait until compare_ack_sig = no;
-- special instructions
if opcode = 1_special then
  -- use register instruction format
  -- (special, re, rt, rd, shamt, funct)
  -- except for break and syscall instruction
  funct := current_inst(5 downto 0);
  re := bton(current_inst(25 downto 21));
  rt := bton(current_inst(20 downto 16));
  rd := bton(current_inst(15 downto 11));
  shamt := bton(current_inst(10
downto 6));

B-3
case funct is
when l_syscall =>
  -- (special, 0 , syscall)
  inst <= op_syscall;
  exception_flag :=

syscall_trap;
when l_break =>
  -- (special, code, break)
  inst <= op_break;
  -- code := current Inst(25 downto 6);
  exception_flag :=

breakpt_trap;
when l_all =>
  inst <= op_all;
  if rd /= 0 then
    reg(rd) :=

shift_ll(reg(rt), shamt);
  end if;
when l_srl =>
  inst <= op_srl;
  if rd /= 0 then
    reg(rd) :=

shift_rl(reg(rt), shamt);
  end if;
when l_sra =>
  inst <= op_sra;
  if rd /= 0 then
    reg(rd) :=

shift_ra(reg(rt), shamt);
  end if;
when l_sllv =>
  inst <= op_sllv;
  if rd /= 0 then
    reg(rd) :=
x'0000_001f';
  end if;
when l_srav =>
  inst <= op_srav;
  if rd /= 0 then
    reg(rd) :=
x'0000_001f';

 shift_ra(reg(rt), shamt);
end if;
when l_sllv =>
  inst <= op_sllv;
  if rd /= 0 then
    reg(rd) :=
x'0000_001f';
  end if;
when l_srlv =>
  inst <= op_srlv;
  if rd /= 0 then
    reg(rd) :=
x'0000_001f';

 shift_rl(reg(rt), shamt);
end if;
when i_jr =>
  -- need one instruction delay
  inst <= op_jr;
  pc_temp := reg(rs);
  delay_slot_flag := set;
when i_fair =>
  -- need one instruction delay
  inst <= op_fair;
  pc_temp := reg(rs);
  -- the address of the inst
  after the delay slot
  -- is placed in rd
  if rd /= 0 then
    reg(rd) := po_reg +
x'0000_0008';
  end if;
  delay_slot_flag := set;
when i_mfhi =>
  inst <= op_mfhi;
  if rd /= 0 then
    reg(rd) := hi_reg;
  end if;
when i_mfhi =>
  inst <= op_mfhi;
  hi_reg := reg(rs);
when i_mflo =>
  inst <= op_mflo;
  if rd /= 0 then
    reg(rd) := lo_reg;
  end if;
when i_mflo =>
  inst <= op_mflo;
  lo_reg := reg(rs);
when i_mult =>
  inst <= op_mult;
  mult_temp := mult(reg(rs),
lo_reg := mult_temp(31 downto 0);
  hi_reg := mult_temp(63 downto 32));
when i_div =>
  inst <= op_div;
  mult_temp := div(reg(rs),
lo_reg := mult_temp(31 downto 0);
  hi_reg := mult_temp(63 downto 32));
when i_add =>
  inst <= op_add;
  if rd /= 0 then
    add_ovf(reg(rs), reg(rt),
  reg(rd), ovf);
  if ovf = '1' then
    exception_flag :=
  overflow;
  end if;
  end if;
when i_adda =>
  -- same as add except never
  inst <= op_adda;
  if rd /= 0 then
    reg(rd) := reg(rs) +
  end if;
when i_sub =>
  inst <= op_sub;
  if rd /= 0 then
    sub_ovf(reg(rs), reg(rt),
  reg(rd), ovf);
  if ovf = '1' then
    exception_flag :=
  overflow;
  end if;
when i_suba =>
  -- same as sub except never
  inst <= op_suba;
  if rd /= 0 then
    reg(rd) := reg(rs) -
  end if;
when i_and =>
  inst <= op_and;
  if rd /= 0 then
  reg(rd) := reg(rs) and
when i_or =>
  inst <= op_or;
  if rd /= 0 then
  reg(rd) := reg(rs) or
when i_xor =>
  inst <= op_xor;
  if rd /= 0 then
  reg(rd) := reg(rs) xor
when i_not =>
  inst <= op_not;
  if rd /= 0 then
  reg(rd) := reg(rs) ~
when l_or =>
  inst <= op_or;
  if rd /= 0 then
    reg(rd) := reg(rs) or
    reg(rt);
  end if;
when l_xor =>
  inst <= op_xor;
  if rd /= 0 then
    reg(rd) := reg(rs) xor
    reg(rt);
  end if;
when l_nor =>
  inst <= op_nor;
  if rd /= 0 then
    reg(rd) := reg(rs) nor
    reg(rt);
  end if;
when l_slt =>
  inst <= op_slt;
  if rd /= 0 then
    bton(reg(rs)) <
    bton(reg(rt)) then
      reg(rd) :=
      x"0000_0001";
    else
      reg(rd) :=
      x"0000_0000";
    end if;
  end if;
when l_sltu =>
  inst <= op_sltu;
  if rd /= 0 then
    bton(reg(rs)) <
    bton(reg(rt)) then
      reg(rd) :=
      x"0000_0001";
    else
      reg(rd) :=
      x"0000_0000";
    end if;
  end if;
when others =>
  -- error, reserved instruction
  inst <= reserved;
  exception_flag :=
  reserved_inst;
end case;

-- conditional branch (bcond)

  elseif opcode = l_bcond then
    -- use immediate instruction
    format
    -- function is found at rt field
    -- (bcond, rs, func, offset)
    ex := bton(current_inst(25 downto
    21));
    reg_funct := current_inst(20
downto 16);
    offset := current_inst(15 downto
    0);
    case reg_funct is
      when l_bnez =>
        inst <= op_bnez;
        if reg(rs)(31) = '1' then
          pc_temp := pc_temp +
          shift_ll(se16to32(offset),2);
          delay_slot_flag := set;
          end if;
        x"0000_0004";
      when l_bnez =>
        inst <= op_bnez;
        if reg(rs)(31) = '0' then
          pc_temp := pc_temp +
          shift_ll(se16to32(offset),2);
          delay_slot_flag := set;
          end if;
        x"0000_0004";
      when l_beq =>
        inst <= op_beq;
        reg(rs) := pc_reg +
        x"0000_0008";
        if reg(rs)(31) = '1' then
          pc_temp := pc_temp +
          shift_ll(se16to32(offset),2);
          delay_slot_flag := set;
          end if;
        x"0000_0004";
      when l_beq =>
        inst <= op_beq;
        reg(rs) := pc_reg +
        x"0000_0008";
        if reg(rs)(31) = '0' then
          pc_temp := pc_temp +
          shift_ll(se16to32(offset),2);
          delay_slot_flag := set;
          end if;
        x"0000_0004";
      when others =>
        -- error, reserved
        instruction
        inst <= reserved;
        exception_flag :=
        reserved_inst;
    end case;

-- jump, jump and link (using target)

  elseif opcode = l_j or opcode = l_jal then
    -- use jump instruction format
    -- (op, target)
    target := current_inst(25 downto
    0);
    case opcode is
      when l_j =>
        inst <= op_j;
        pc_temp := pc_reg(31 downto
        20) & target & b"00";
        delay_slot_flag := set;
        when l_jal =>
          inst <= op_jal;
          pc_temp := pc_reg(31 downto
          20) & target & b"00";
          x"0000_0000";
          delay_slot_flag := set;
          when others =>
            null;
      end case;

-- branch instructions

  elseif opcode_reg = b"00" then
    -- for l.beq, l_bne, l_blez,
    l_bnez
    -- use immediate instruction
    format
    -- (op, rs, rt, offset)
    ex := bton(current_inst(25 downto
    21));
    rt := bton(current_inst(20
downto 16));
    offset := current_inst(15 downto
    0);
    case opcode is
      when l_beq =>
inst <- op_beq;
if reg(rs) = reg(rt) then
  pc_temp := pc_reg +
  shift_ll(pc_temp, 2);
end if;
when i_bnez =>
  inst := op_bnez;
  if reg(rs) /= reg(rt) then
    pc_temp := pc_reg +
    shift_ll(pc_temp, 2);
  end if;
end case;

-- ALU immediate instructions

elsif opcode_seg = b"001" then
  -- use immediate instruction
  format
  -- (op, rs, rt, immediate)
  case opcode is
    when i_addi =>
      inst := op_addi;
      if rt /= 0 then
        overflow := (x"0000_0004" =
          sel6to32(immed), reg(rt), ovrfw);
        if ovrfw = '1' then
          exception_flag :=
          add_ovf(reg(rs),
          sel6to32(immed), reg(rt), ovrfw);
        end if;
      end if;
    end if;
  end case;

-- load and store instructions

elsif opcode_seg = b"101" or
  opcode_seg = b"100" then
  -- use immediate instruction
  format
  -- (op, base, rt, offset)
if rt /= 0 then
  ea := sel16to32(offset) +
  mem_read(ea, lefty);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_lw =>
  inst <= op_lw;
  if rt /= 0 then
    ea := sel16to32(offset) +
    temp_reg_val1 := rt;
    latency_flag := set;
  end if;
when l_lbu =>
  inst <= op_lbu;
  if rt /= 0 then
    ea := sel16to32(offset) +
    temp_reg_val1 := rt;
    latency_flag := set;
  end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_lwr =>
  inst <= op_lwr;
  ea := sel16to32(offset) +
  mem_write(ea, halfword);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_sw =>
  inst <= op_sw;
  ea := sel16to32(offset) +
  mem_write(ea, byte, reg(rt));
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_swl =>
  inst <= op_swl;
  ea := sel16to32(offset) +
  mem_write(ea, byte, lefty);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_sh =>
  inst <= op_sh;
  ea := sel16to32(offset) +
  mem_write(ea, halfword, reg(rt));
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_swr =>
  inst <= op_swr;
  ea := sel16to32(offset) +
  mem_write(ea, word, reg(rt));
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when i_sh =>
  inst <= op_i_sh;
  ea := sel16to32(offset) +
  mem_read(ea, word);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when i_shl =>
  inst <= op_i_shl;
  ea := sel16to32(offset) +
  mem_read(ea, byte);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when i_lw =>
  inst <= op_lw;
  if rt /= 0 then
    ea := sel16to32(offset) +
    mem_read(ea, word);
    temp_reg_val1 := rt;
    latency_flag := set;
  end if;
when i_lbu =>
  inst <= op_lbu;
  if rt /= 0 then
    ea := sel16to32(offset) +
    mem_read(ea, ubyte);
    temp_reg_val1 := rt;
    latency_flag := set;
  end if;
when i_lwr =>
  inst <= op_lwr;
  if rt /= 0 then
    ea := sel16to32(offset) +
    mem_read(ea, uhalfword);
    temp_reg_val1 := rt;
    latency_flag := set;
  end if;
when i_lwr =>
  inst <= op_lwr;
  ea := sel16to32(offset) +
  mem_read(ea, righty);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when i_lbu =>
  inst <= op_lbu;
  ea := sel16to32(offset) +
  mem_read(ea, ubyte);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when i_lwr =>
  inst <= op_lwr;
  ea := sel16to32(offset) +
  mem_read(ea, righty);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_ae =>
  inst <= op_ae;
  ea := sel16to32(offset) +
  mem_read(ea, word);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_aw =>
  inst <= op_aw;
  ea := sel16to32(offset) +
  mem_read(ea, word);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_awr =>
  inst <= op_awr;
  ea := sel16to32(offset) +
  mem_write(ea, righty);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_aewr =>
  inst <= op_aewr;
  ea := sel16to32(offset) +
  mem_write(ea, word);
end if;

reg(base);
temp_reg_val_l1;
downto 0;

when l_aewr =>
  inst <= op_aewr;
  ea := sel16to32(offset) +
  mem_write(ea, righty);
null;
end case;
run_mode_flag := no;
else
-----------------------------
delay_slot
-----------------------------
if delay_slot_flag /= ignore then
if delay_slot_flag = set then
delay_slot :=

-- increment pc
pc_reg := pc_reg +
x"0000_0004";
else
delay_slot_flag := ignore;
pc_reg := pc_temp;
end if;
else
-- increment pc
pc_reg := pc_reg + x"0000_0004";
end if;

---------------
-- latency of one instruction on
memory to register loads
---------------
if latency_flag /= ignore then

if latency_flag = set then

reg(temp_reg_num_2)
(23 downto 0);

when b'11' =>
reg(temp_reg_num_2)
:= temp_reg_val_2;
end case;
right_flag := ignore;
else
reg(temp_reg_num_2) :=
temp_reg_val_2;
end if;

---------------
-- set branch delay flag
---------------
if delay_slot_flag = delay_slot then
bd_flag := yes;
else
bd_flag := no;
end if;

---------------
-- update signals
---------------
counter
pc <= pc_reg;
-- program

----------------
-- registers r0
----------------

r0 <= reg(0);
-- registers r0
thru r31

r1 <= reg(1);
r2 <= reg(2);
r3 <= reg(3);
r4 <= reg(4);
r5 <= reg(5);
r6 <= reg(6);
r7 <= reg(7);
r8 <= reg(8);
r9 <= reg(9);
r10 <= reg(10);
r11 <= reg(11);
r12 <= reg(12);
r13 <= reg(13);
r14 <= reg(14);
r15 <= reg(15);
r16 <= reg(16);
r17 <= reg(17);
r18 <= reg(18);
r19 <= reg(19);
r20 <= reg(20);
r21 <= reg(21);
r22 <= reg(22);
r23 <= reg(23);
r24 <= reg(24);
r25 <= reg(25);
r26 <= reg(26);
r27 <= reg(27);
r28 <= reg(28);
r29 <= reg(29);
r30 <= reg(30);
r31 <= reg(31);

hi <= hi_reg;
-- hi, lo
lo <= lo_reg;

----------------
-- exception handling
----------------
epc <= epc_reg;
-- exception

cause <= cause_reg;
exception_sig <= exception_flag;
-- compare machine state with
expected results
compare_control_sig <= test;
wait until compare_sok_sig = yes;
compare_control_sig <= reset;
wait until compare_sok_sig = no;
end loop; -- end while run_mode_flag = yes;
when others =>
null; -- illegal system

control command
end case;

-- end case for system command

end process processor;

end cpu_a;

COMPONENT: MEMORY
FILENAME: MEMORY_A.VHDL
DESCRIPTION: Memory component of behavioral model test bench (entity)

-- library and use clauses
library mpc_portable, ieee;
use mpc_portable.qsim_logic.all;
use mpc_portable.qsim_relations.all;
use ieee.std_logic_1164.all;
use std.textio.all;
library my_package;
use my_package.package_1.all;

entity memory is
ported
mem_control_sig in mem_control_type;
addr_bus in bit_30;
addr_bus_lo in bit_32;
mem_ack_sig out question_type;
mem_exception_sig out question_type;
data_bus inout bus_bit_32 bus;
end memory;

COMPONENT: MEMORY
FILENAME: MEMORY_A.VHDL
DESCRIPTION: Memory component of behavioral model test bench (architecture)

architecture memory_a of memory is
begin
memory: process
constant low_address: integer := 0;
constant high_address: integer := 65535;
type memory_array is
array (integer range low_address to high_address) of bit_32;
variable mem: memory_array;
variable addr: integer range 0 to high_address;
variable temp: bit_32;

file ini: text is in "machine";
variable line: line;
variable inst: bit_32;

begin
wait on mem_control_sig; -- entry into memory

data_bus <= null;

if mem_control_sig /= reset or mem_control_sig /= load then
assert addr_bus(29 downto 14) = x'0000'
report "ADDRESS OUT OF RANGE"
end if;

if addr_bus(29 downto 14) = x'0000' then
if read_w =>
addr := bton(addr_bus);
temp := mem(addr);
case addr_bus_lo is
when b'00' =>
data_bus <= inout_bus_bit_32;
when others =>
-- flag address exception
mem_exception_sig <= yes;
end case;
mem_ack_sig <= yes after delay;
end if;

when read_b =>
addr := bton(addr_bus);
temp := mem(addr);
case addr_bus_lo is
when b'00' =>
data_bus <= se8to32(temp(15 downto 0));
when b'10' =>
data_bus <= se16to32(temp(15 downto 0));
when others =>
-- flag address exception
mem_exception_sig <= yes;
end case;
mem_ack_sig <= yes after delay;
end if;

when read_uw =>
addr := bton(addr_bus);
temp := mem(addr);
case addr_bus_lo is
when b'00' =>
data_bus <= x'00000000' & temp(31 downto 16);
when b'10' =>
data_bus <= x'00000000' & temp(15 downto 0);
when others =>
-- flag address exception
mem_exception_sig <= yes;
end case;
mem_ack_sig <= yes after delay;
end if;

when read_uh =>
addr := bton(addr_bus);
temp := mem(addr);
case addr_bus_lo is
when b'00' =>
data_bus <= se8to32(temp(31 downto 24));
when b'10' =>
data_bus <= se8to32(temp(23 downto 16));
when b'10' =>
data_bus <= se8to32(temp(15 downto 0));
when others =>
data_bus <= se8to32(temp(7 downto 0));
end case;
mem_ack_sig <= yes after delay;
end if;

component
end component;

end process;

end memory_a;
begin
    case addr_bus(lo) is
    when b'111' =>
        mem(addr) := data_bus(23 downto 0);
    when b'110' =>
        mem(addr) := data_bus(16 downto 0);
    when b'101' =>
        mem(addr) := data_bus(9 downto 0);
    when b'100' =>
        mem(addr) := data_bus(2 downto 0);
    when b'011' =>
        mem(addr) := data_bus(1 downto 0);
    when b'010' =>
        mem(addr) := data_bus(14 downto 0);
    when b'001' =>
        mem(addr) := data_bus(17 downto 0);
    when b'000' =>
        mem(addr) := data_bus(15 downto 0);
    end case;
end process;
end component;

architecture compare_a of compare is
begin
    compare: process
    begin
        type test_field_type is array (0 to 9) of bit_32;
        type test_array_type is array (0 to 1000) of
        test_field_type;
        variable expect: test_array_type;
        variable index: integer range 0 to 1000;
        file ini: text in "expected";
        variable linel: line;
        variable inst: bit_32;
        begin
            wait on compare_control_sig;
            case compare_control_sig is
                when no_delay =>
                    null;
                when after_delay =>
                    delay;
                when after_2_delay =>
                    delay;
                when after_3_delay =>
                    delay;
                case addr_bus: when others =>
                    illegal_error;
                end case;
                when load =>
                    addr := 0;
                    while endfile(inline) = false loop
                        readlinel(inlinel, linel);
                        if line = inst then
                            addr := inst + 1;
                        end loop;
                        mem(addr) := data_bus(31 downto 0);
                    end while;
                    end loop;
                when others =>
                    error, illegal_operation;
            end case;
        end process;
    end component;
end memory_a;

COMPONENT: COMPARE
FILENAME: COMPARE_E.VHDL
DESCRIPTION: Compare component of behavioral model test bench (entity)
-- library and use clauses
library mgc_portable, ieee;
use mgc_portable.qsim_logic.all;
use mgc_portable.qsim_relations.all;
use ieee.std_logic_1164.all;
use std.textio.all;
library my_packages;
use my_packages.package_1.all;
entity compare is
    port(compare_control_sig: in
    compare_control_type;
    pc: in bit_32;
    r1: in bit_32;
    r2: in bit_32;
    r3: in bit_32;
    r4: in bit_32;
    hi: in bit_32;
    lo: in bit_32;
    eps: in bit_32;
    cause: in bit_32;
    compare_ack_slg: out question_type);
end compare;

COMPONENT: COMPARE
FILENAME: COMPARE_A.VHDL
DESCRIPTION: Compare component of behavioral model test bench (architecture)
architecture compare_a of compare is
begin
    compare: process
    type test_field_type is array (0 to 9) of bit_32;
    type test_array_type is array (0 to 1000) of
test_field_type;
    variable expect: test_array_type;
    variable index: integer range 0 to 1000;
    file ini: text in "expected";
    variable linel: line;
    variable inst: bit_32;
    begin
        wait on compare_control_sig;
        case compare_control_sig is
            when no_delay =>
                null;
            when after_delay =>
                delay;
            when after_2_delay =>
                delay;
            when after_3_delay =>
                delay;
            case addr_bus: when others =>
                illegal_error;
            end case;
        end case;
    end process;
end compare_a;
function wired_or (driver: bit_32_array) return bit_32;

subtype bus_bit_32 is wired_or bit_32;

type eye_control_type is (stop, run, load, reset);
type mem_control_type is (reset, read_b, read_ab, read_hw, read_ah);

write_8, write_1, read_r;

write_x, load;

type question_type is (no, yes);
type delay_slot_type is (ignore, delay_slot, set);
type inst_type is (nop, not_implemented, reserved);

op_bltz, op_bltzal, op_bgtz, op_bgtzal,

op_jr, op_jalr, op_jr,

op_lw, op lw, op lw,

op_mfhi, op mflo, op_mfhi,

op_lw, op lw, op lw, op lw, op lw, op lw, op lw, op lw,

op_mfhi, op mflo, op_mfhi,

op_lw, op lw, op lw, op lw, op lw, op lw, op lw, op lw,

op_mfhi, op mflo, op_mfhi,

op_lw, op lw, op lw, op lw, op lw, op lw, op lw, op lw,

op_mfhi, op mflo, op_mfhi,

op_lw, op lw, op lw, op lw, op lw, op lw, op lw, op lw,

op_mfhi, op mflo, op_mfhi,

op_lw, op lw, op lw, op lw, op lw, op lw, op lw, op lw,

op_mfhi, op mflo, op_mfhi,

op_lw, op lw, op lw, op lw, op lw, op lw, op lw, op lw,

op_mfhi, op mflo, op_mfhi,
function wired_or(drivers: bit_32_array) return
bit_32 is
variable result: bit_32 := x"0000_0000";
begin
for i in drivers'range loop
result := result or driver(i);
end loop;
return result;
end wired_or;

function btof(bits: in bit_vector) return integer is
variable temp: bit_vector(bits'range);
variable result: integer := 0;
begin
if bits(bits'left) = '1' then -- negative number
    temp := not bits;
else
    temp := bits;
end if;
for index in bits'range loop
result := result * 2 +
bit'pos(temp(index));
end loop;
if bits(bits'left) = '1' then
    result := (-result) - 1;
else
    return result;
end if;
end btof;

procedure itob(int: in integer; bits: out
bit_vector) is
variable temp: integer;
variable result: bit_vector(int'range);
begin
if int < 0 then
    result := (-int) + 1;
else
    temp := int;
end if;
for index in int'range loop
result := bit'val(temp rem 2);
if int < 0 then
    result(index) := bit'val('left) - 1;
end if;
bits(index) := result;
end loop;
end itob;

function shift_ll(arg_in: in bit_vector; amt: in
integer) return bit_vector;
begin
variable result: bit_vector(amt'range);
if int < 0 then
    result := (-int) + 1;
else
    temp := int;
end if;
for index in int'range loop
result(index) := bit'val(temp rem 2);
if int < 0 then
    result(index) := bit'val('left) - 1;
end if;
bits(index) := result;
end loop;
end shift_ll;

function shift_rl(arg_in: in bit_vector; amt: in
integer) return bit_vector;
begin
variable result: bit_vector(amt'range);
for j in 1 to amt loop
for index in arg_in'range loop
    result(index) := result(index - 1); 
end if;
end loop;
end shift_rl;
for index in arg_in.reverse_range loop
  if index /= arg_in'high then
    result(index) := result(index + 1);
  else
    result(index) := '0';
  end if;
end loop;
end loop;
return result;
end shift_ra;

function shift_ra(arg_in: in bit_vector; amt: in integer) return bit_vector is
  variable result: bit_vector(arg_in.range);
begin
  result := arg_in;
  if arg_in(arg_in'left) = '1' then
    for j in 1 to amt loop
      for index in arg_in.reverse_range loop
        if index /= arg_in'high then
          result(index) := result(index + 1);
        else
          result(index) := '1';
        end if;
      end loop;
    end loop;
  else
    for j in 1 to amt loop
      for index in arg_in.reverse_range loop
        if index /= arg_in'high then
          result(index) := result(index + 1);
        else
          result(index) := '0';
        end if;
      end loop;
    end loop;
  end if;
end loop;
return result;
end shift_ra;

function se16to32(arg_in: in bit_16) return bit_32 is
  variable result: bit_32;
begin
  if arg_in(arg_in'left) = '1' then
    result := x'ffff' & arg_in;
  else
    result := x'0000' & arg_in;
  end if;
end function se16to32;

function se8to32(arg_in: in bit_8) return bit_32 is
  variable result: bit_32;
begin
  if arg_in(arg_in'left) = '1' then
    result := b'1111_1111_1111_1111_1111_1111_1111_1111' & arg_in;
  else
    result := b'0000_0000_0000_0000_0000_0000_0000_0000' & arg_in;
  end if;
end function se8to32;

function se24to32(arg_in: in bit_24) return bit_32 is
  variable result: bit_32;
begin
  if arg_in(arg_in'left) = '1' then
    result := b'1111_1111_1111_1111_1111_1111_1111_1111' & arg_in;
  else
    result := b'0000_0000_0000_0000_0000_0000_0000_0000' & arg_in;
  end if;
end function se24to32;

function se16to30(arg_in: in bit_16) return bit_30 is
  variable result: bit_30;
begin
  if arg_in(arg_in'left) = '1' then
    result := b'11_1111_1111_1111_1111_1111_1111_1111' & arg_in;
  else
    result := b'00_0000_0000_0000_0000_0000_0000_0000' & arg_in;
  end if;
end function se16to30;

procedure add_ovf(l, r: in bit_32; s: out bit_32; c: out bit) is
  variable sum: bit_32;
  variable result: bit_32;
begin
  for i in 0 to 31 loop
    if carry = '0' then
      sum(i) := l(i) xor r(i);
      if 1 = 30 then
        carry30 := carry;
      else
        carry := carry(s);
        carry9 := carry(i);
      end if;
    else
      sum(i) := not(l(i) xor r(i));
      if 1 = 30 then
        carry30 := carry;
      else
        carry := carry(s);
        carry9 := carry(i);
      end if;
    end if;
  end loop;
end procedure add_ovf;

function "="(l, r: in bit_32) return bit_32 is
  variable temp: result: bit_32;
begin
  temp := (not l) xor x'0000_0000_0000_0000_0000_0000_0000_0000';
  result := l + temp;
end function "=";

procedure sub_ovf(l, r: in bit_32; s: out bit_32; c: out bit) is
  variable temp: result: bit_32;
begin
  temp := (not l) xor x'0000_0000_0000_0000_0000_0000_0000_0000';
  add_ovf(l, temp, s, c);
end procedure sub_ovf;

function """"(t, b: in bit_32) return bit_64 is
  type multi_array is array (0 to 31) of bit_64;
  variable ma: multi_array;
  variable result: bit_64 := '0';
begin
  for i in 0 to 31 loop
    if b(i) = '0' then
      ma(i) := x'0000_0000_0000_0000_0000_0000_0000_0000';
    else
      ma(i) := x'0000_0000_0000_0000_0000_0000_0000_0000';
    end if;
  end loop;
end function """";
ma(bot)(31 downto 0) := t;
ma(bot) := shift_l1(ma(bot), bot);
end if;
end loop;
for j in 0 to 30 loop
carry := '0';
for l in 0 to 63 loop
if carry = '0' then
carry := ma(j)(l) and ma(j + 1)(l);  
ma(j + 1)(l) := ma(j)(l) or ma(j + 1)(l);
end if;
end loop;
end loop;
return ma(31);
end if;
return temp;
end mult;

function mult(a, b: in bit_32) return bit_64 is
variable temp: bit_64;
variable a_twos_comp: bit_32;
variable b_twos_comp: bit_32;
begin
if a(31) = '0' and b(31) = '0' then
temp := a * b;
elsif a(31) = '0' and b(31) = '1' then
b_twos_comp := (not b) + x"0000_0001";
temp := a * b_twos_comp;
temp := add_64(not temp, x"0000_0000_0000_0001");
elsif a(31) = '1' and b(31) = '0' then
a_twos_comp := (not a) + x"0000_0001";
temp := a_twos_comp * b;
temp := add_64(not temp, x"0000_0000_0000_0001");
else
a_twos_comp := (not a) + x"0000_0001";
b_twos_comp := (not b) + x"0000_0001";
temp := a_twos_comp * b_twos_comp;
end if;
return temp;
end mult;

function "//(dividend, divs: in bit_32) return bit_64 is
variable divd_temp: bit_32;
variable divs_temp: bit_32;
variable quot: bit_32;
variable result: bit_64;
variable divd: bit_32;
begin
divd := dividend;
for i in 31 downto 0 loop
divd_temp := shift_r1(divd, i);
if divd > divd_temp then
quot(i) := '1';
else
quot(i) := '0';
end if;
divd_temp := shift_l1(divs, i);
divd := divd - divd_temp;
end loop;
result(31 downto 0) := quot;
result(63 downto 32) := divd;
return result;
end "//";

function div(a, b: in bit_32) return bit_64 is
variable temp: bit_64;
variable a_twos_comp: bit_32;
variable b_twos_comp: bit_32;
begin
if a(31) = '0' and b(31) = '0' then
temp := a / b;
elsif a(31) = '0' and b(31) = '1' then
b_twos_comp := (not b) + x"0000_0001";
temp := a / b_twos_comp;
temp(31 downto 0) := (not temp(31 downto 0)) + x"0000_0001";
elsif a(31) = '1' and b(31) = '0' then
a_twos_comp := (not a) + x"0000_0001";
temp := a_twos_comp / b;
temp(31 downto 0) := (not temp(31 downto 0)) + x"0000_0001";
else
a_twos_comp := (not a) + x"0000_0001";
b_twos_comp := (not b) + x"0000_0001";
temp := a_twos_comp / b_twos_comp;
temp := add_64(not temp, x"0000_0000_0000_0001");
end if;
return temp;
end div;
library my_packages;
use my_packages.package_1.all;
use my_packages.dfpack.all;

library df_comp;
use df_comp.dfmemory.all;
use df_comp.dcompare.all;

df memory Req, memory load, question_type, compare, compare load Ack, compare load, r2, r3, r4, r2 test, hi test, lo test;
begin

cpu module:
dfp
port map(e eye_control sig, e system control sig, memory load ack, compare, compare load, po test, r1 test, r2 test, r2 test, r3 test, r4 test, hi test, lo test, compare load ack, compare, compare load ack);
end dfmips;


<<H450,540>>

dfmips
entity dfmips is
end dfmips;

COMPONENT: DFMPH
FILENAME: DFMPH.VHDL
DESCRIPTION: Test bench for dataflow model of asynchronous version of MIPS R3000 microprocessor (entity)

library my_packages;
use my_packages.package_1.all;
use my_packages.dfpack.all;
library df_comp;
use df_comp.dfmemory.all;
use df_comp.dcompare.all;
entity dfmips is
end dfmips;

COMPONENT: DFMPH
FILENAME: DFMPH_A.VHDL
DESCRIPTION: Test bench for dataflow model of asynchronous version of MIPS R3000 microprocessor (architecture)

architecture dfmips_a of dfmips is
component dfcpu
port(e eye_control sig, in eye_control_type; memory load ack, in question_type; compare, in question_type; memory load ack, in question_type; addr bus, inbus bit_32 bus; data bus, inbus bit_32 bus; memory req, out bit; memory w, out bit; memory opmode, out bit_3; memory load, out question_type; compare, out question_type; compare load, out question_type; pc test, out bit_32; r1 test, out bit_32; r2 test, out bit_32; r2 test, out bit_32; r3 test, out bit_32; r3 test, out bit_32; r4 test, out bit_32; r4 test, out bit_32; r31 test, out bit_32; hi test, out bit_32; lo test, out bit_32); end component;

component dfmemory
port(load, in question_type; reg, in bit; w, in bit; opcode, in bit_3; ack, out bit; load Ack, out question_type; addr bus, inbus bit_32 bus; data bus, inbus bit_32 bus); end component;

component dcompare
port(compare, in question_type; compare load, in question_type; pc test, in bit_32; r1 test, in bit_32; r2 test, in bit_32; r2 test, in bit_32; r3 test, in bit_32; r3 test, in bit_32; r4 test, in bit_32; r4 test, in bit_32; r31 test, in bit_32; hi test, in bit_32; lo test, in bit_32; compare req, cut question_type; compare load req, cut question_type); end component;

signal e eye_control, e eye_control_type;
signal memory Opcode, bit_3;
signal addr bus, data bus, bus bit_32;
signal compare, compare load, question_type;
signal compare, compare load, question_type;
signal p o test, r1 test, r2 test, r3 test, h i test, l o test, bit_32;
signal r4 test, r31 test, h i test, l o test, bit_32;
begin

cpu module:
dfp
port map(e eye_control sig, e system control sig, memory load ack, compare, compare load, po test, r1 test, r2 test, r2 test, r3 test, r4 test, r3 test, r3 test, hi test, lo test);

memory module:
dfmemory
port map(memory load, memory req, memory Opcode, memory load Ack, addr bus, data bus);
compare module:
dcompare
port map(compare, compare load, po test, r1 test, r2 test, r2 test, r3 test, r4 test, r3 test, r3 test, hi test, lo test, compare load Ack, compare, compare load Ack);
end dfmips_a;

COMPONENT: DFPACK
FILENAME: DFPACK_H.VHDL
DESCRIPTION: Dataflow model package (header)

library my_packages;
use my_packages.package_1.all;

package dfpack is
subtype bit_10 is bit_vector (9 downto 0);
constant m lb, bit_3 := 0^0;
constant m lb, bit_3 := 0^1;
constant m lw1, bit_3 := 0^2;
constant m lw, bit_3 := 0^3;
constant m sb, bit_3 := 0^4;
constant m sh, bit_3 := 0^5;
constant m hu, bit_3 := 0^6;
constant m lw1, bit_3 := 0^7;
constant m sw, bit_3 := 0^8;
constant m lw1, bit_3 := 0^9;
constant m sw, bit_3 := 0^10;
constant m lw1, bit_3 := 0^11;
constant m sw, bit_3 := 0^12;
constant m lw1, bit_3 := 0^13;
constant m sw, bit_3 := 0^14;
constant m lw1, bit_3 := 0^15;
constant m sw, bit_3 := 0^16;
end dfpack;

COMPONENT: DATAFLOW COMPONENTS
FILENAME: TF77.E.VHDL for entity and TF77.A.VHDL for architecture
DESCRIPTION: All components used in dataflow model follow (entity shown first)

entity dfstomux is
port(l0) in bit;
ii in bit;
s in bit;
end entity;

<<H450,540>>

DATAFLOW COMPONENTS
entity dfstomux is
port(l0) in bit;
ii in bit;
s in bit;
end entity;

<<H450,540>>

entity dfstomux is
port(l0) in bit;
ii in bit;
s in bit;
end entity;

<<H450,540>>

entity dfstomux is
port(l0) in bit;
ii in bit;
s in bit;
end entity;
entity df2tolmux
  use
    my_packages.package_l.all;
    df2tolmux_a;
  end
end df2tolmux;

architecture df2tolmux_a of df2tolmux is
  begin
    df2tolmux_a;
  end
end df2tolmux_a;

architecture df2tolmux16 of df2tolmux16 is
  begin
    df2tolmux16;
  end
end df2tolmux16;

architecture df2tolmux16_a of df2tolmux16 is
  begin
    df2tolmux16_a;
  end
end df2tolmux16_a;

architecture df2tolmux30 of df2tolmux30 is
  begin
    df2tolmux30;
  end
end df2tolmux30;

architecture df2tolmux30_a of df2tolmux30 is
  begin
    df2tolmux30_a;
  end
end df2tolmux30_a;

architecture df2tolmux32 of df2tolmux32 is
  begin
    df2tolmux32;
  end
end df2tolmux32;

architecture df2tolmux32_a of df2tolmux32 is
  begin
    df2tolmux32_a;
  end
end df2tolmux32_a;

architecture df2tolmuxx8 of df2tolmuxx8 is
  begin
    df2tolmuxx8;
  end
end df2tolmuxx8;

architecture df2tolmuxx8_a of df2tolmuxx8 is
  begin
    df2tolmuxx8_a;
  end
end df2tolmuxx8_a;

architecture df2tolmuxx32 of df2tolmuxx32 is
  begin
    df2tolmuxx32;
  end
end df2tolmuxx32;

architecture df2tolmuxx32_a of df2tolmuxx32 is
  begin
    df2tolmuxx32_a;
  end
end df2tolmuxx32_a;

architecture df4tolmux of df4tolmux is
  begin
    df4tolmux;
  end
end df4tolmux;
architecture df4tolmux32 of df4tolmux is
begin
  o <= 10 after 0.4 ns when s1 = '0' and s0 = '0'
else
  11 after 0.4 ns when s1 = '0' and s0 = '1'
else
  12 after 0.4 ns when s1 = '1' and s0 = '0'
else
  13 after 0.4 ns;
end df4tolmux32;

DE4TO1MUX16

library my_packages;
use my_packages.package_1.all;
entity df4tolmux16 is
port (10: in bit_16;
11: in bit_16;
12: in bit_16;
13: in bit_16;
s0: in bit;
s1: in bit;
o: out bit_16);
end df4tolmux16;
architecture df4tolmux16 of df4tolmux is
begin
  o <= 10 after 0.4 ns when s1 = '0' and s0 = '0'
else
  11 after 0.4 ns when s1 = '0' and s0 = '1'
else
  12 after 0.4 ns when s1 = '1' and s0 = '0'
else
  13 after 0.4 ns;
end df4tolmux16;

DE4TO1MUX32

library my_packages;
use my_packages.package_1.all;
entity df4tolmux32 is
port(10: in bit_32;
11: in bit_32;
12: in bit_32;
13: in bit_32;
s0: in bit;
s1: in bit;
o: out bit_32);
end df4tolmux32;
architecture df4tolmux32 of df4tolmux is
begin
  o <= 10 after 0.4 ns when s1 = '0' and s0 = '0'
else
  11 after 0.4 ns when s1 = '0' and s0 = '1'
else
  12 after 0.4 ns when s1 = '1' and s0 = '0'
else
  13 after 0.4 ns;
end df4tolmux32;
library my_packages;
use my_packages.package_1.all;

entity dfadd8 is
port(i: in bit_32;
start: in bit;
o: out bit_32;
done: out bit);
end dfadd8;

architecture dfadd8_a of dfadd8 is
signal temp: bit_32;
signal dummy, done_temp: bit;
begin
    temp <= i * x'0000_0008' after 25 ns when start = '1'
    else
        temp;
    end
    o <= temp;
    dummy <= '1' when start = '1' else '0';
    done_temp <= '1' after 26 ns when dummy = '1' else '0';
    done <= done_temp;
end dfadd8_a;

library my_packages;
use my_packages.package_1.all;
library df_comp;
use df_comp.dfreg32.all;
use df_comp.dfreg.all;
use df_comp.dfred.all;
use df_comp.dffslat.all;
use df_comp.dffed.all;
use df_comp.dfregbank.all;
use df_comp.dfalublk.all;
use df_comp.df2tolmux32.all;
use df_comp.dfalu.all;

entity dfalu is
port(alu_start: in bit;
ib: in bit;
alu_select: in bit;
mselect: in bit;
add_out: in bit;
reg1_out: in bit_5;
reg2_out: in bit_5;
treg_out: in bit_5;
pol: in bit_32;
inst_in: in bit_32;
wb_data: in bit_32;
wb_sel: in bit_8;
vbt: in bit_4;
write: in bit;
jjr: in bit;
set_hi_db: in bit;
set_lo_db: in bit;
c: out bit;
hl_db: out bit;
db: out bit_32;
reg: out bit_32;
alu_add: out bit;
inst_out: out bit_32;
data_out: out bit_32;
data2_out: out bit_32;
cod: out bit;
r1_test: out bit_32;
r2_test: out bit_32;
r3_test: out bit_32;
r4_test: out bit_32;
r5_test: out bit_32;
alu_done: out bit);
end dfalu;

architecture dfalu_a of dfalu is
component dfreg32
port(d: in bit_32;
o: in bit;
q: out bit_32;
end component;

component dfreg
port(d: in bit;
o: in bit;
x: in bit;
q: out bit);
end component;

component dfred
port(i: in bit;
o: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;

component dfalublk
port(a_in: in bit_32;
b_in: in bit_32;
inst_in: in bit_32;
pcl_in: in bit_32;
ib: in bit;
start: in bit;
lst: in bit;
add: in bit_32;
ex: out bit;
b: out bit_32;
pcl_out: out bit_32;
alu_done: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;

component dfalu
port(i: in bit;
o: out bit);
end component;
component dfalu32 is
  port (a, b, in: bit_32);
end component;

use library my_packages;

begin
  dfalu32: dfalu32
    port map(a, b, in);
end dfalu32;
component dfalu32_a;

end component;

component dfalu32

end component;

component dfalu32e

end component;

component dfalu32e

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;

component dfalu32

end component;
library my_packages;
use my_packages.package_l.all;

entity dfaludec is
  port(i: in bit);
  s0: out bit;
  s1: out bit;
  s2: out bit;
  s3: out bit;
end dfaludec;

architecture dfaludec_a of dfaludec is
begin
  s0 <= '1' after 2.5 ns when -- c
    i(31) = '0' and
    i(29 downto 26) = 'b"11011") or
    -- e
    i(29 downto 28) = 'b"00'
  or
    i(26) = '0' and
    i(4 downto 0) = 'b"01")
  or
    f
    i(29 downto 28) = 'b"00'
  or
    h
    i(29 downto 26) = 'b"1111") or
    -- j
    i(29 downto 27) = 'b"100'
  or
    m
    i(29 downto 28) = 'b"00'
  and
    i(26) = '0' and
    i(5) = '0' or
    n
    i(29 downto 27) = 'b"001
  or
    u
    i(31) = '1' else
    '0' after 2.5 ns;
  s1 <= '1' after 2.5 ns when -- a
    i(31) = '0' and
    i(29 downto 26) = 'b"1111") or
    -- k
    i(29 downto 26) = 'b"00'
  or
    i(26) = '0' and
    i(5) = '0' or
    -- p
    i(29) = '0' and
    i(5) = '1' or
    -- q
    i(29) = '0' and
    i(2) = '0' or
    -- r
    i(29) = '0' and
    i(26) = '1' or
    -- s
    i(29 downto 28) = 'b"10"
  or
    t
    i(29 downto 28) = 'b"01"
  or
    u
    i(31) = '1' else
    '0' after 2.5 ns;
end dfaludec_a;

architecture dfasel_a of dfasel is
begin
  s2 <= '1' after 2.5 ns when -- b
    i(31) = '0' and
    i(29 downto 28) = 'b"00'
  or
    i(24) = '0' and
    i(5) = '1' and
    i(3 downto 2) = 'b"01")
  or
    c
    i(31) = '0' and
    i(29 downto 26) = 'b"11011") or
    -- g
    i(31) = '0' and
    i(29 downto 28) = 'b"11'
  or
    i(24) = '0' else
    '0' after 2.5 ns;
  s3 <= '1' after 2.5 ns when -- a
    i(31) = '0' and
    i(29 downto 28) = 'b"00'
  or
    i(24) = '0' and
    i(5) = '1' and
    i(3 downto 0) = 'b"0111")
  else
    '0' after 2.5 ns;
end dfasel_a;

architecture dfasel_a of dfasel is
begin
  s2 <= '1' after 2.5 ns when -- b
    i(31) = '0' and
    i(29 downto 28) = 'b"00'
  or
    i(24) = '0' and
    i(5) = '1' and
    i(3 downto 2) = 'b"01")
  or
    c
    i(31) = '0' and
    i(29 downto 26) = 'b"11011") or
    -- g
    i(31) = '0' and
    i(29 downto 28) = 'b"11'
  or
    i(24) = '0' else
    '0' after 2.5 ns;
end dfasel_a;

library my_packages;
use my_packages.package_l.all;

library df_comp;
use df_comp.df4toulmux32.all;

entity dfasel is
  port(a_in: in bit);
  s1: out bit;
  s2: out bit;
  s3: out bit;
end dfasel;

architecture dfasel_a of dfasel is
begin
  s2 <= '1' after 2.5 ns when -- b
    i(31) = '0' and
    i(29 downto 28) = 'b"01")
  end component;

constant zero32: bit_32 := 'x"0000_0000";
library df_comp;
use df_comp.dfred.all;
use df_comp.dfredfeed.all;
use df_comp.dfslat.all;

entity dfbc is
port (tx: in bit);

end dfbc;

architecture dfbc_a of dfbc is
    component dfred
      port(i: in bit;
      o: out bit);
    end component;

    component dfredfeed
      port(i: in bit;
      o: out bit);
    end component;

    component dfslat
      port(s: in bit;
      q: out bit);
    end component;

    signal q0, q1, clk, d; bit;
    signal i, ml, ml_r, ack_f; bit;
    signal ia_sel, ia_sel_r, ma_sel, ma_sel_r; bit;
    signal i_f, q_f; bit;

    begin
        clk <= not clk after 4 ns;
        q0 <= ((not q1) and (not q0) and (not d)) or
        ((not q1) and (not q0) and (not d)) or
        (q1 and (not q0) and (not d))
        when clk'event and clk = '1' else q0;
        q1 <= ((not q1) and (not q0) and (not d)) or
        ((not q1) and (not q0) and (not d)) or
        (q1 and (not q0) and (not d))
        when clk'event and clk = '1' else q1;
        ia_sel <= q1 and (not q0) after 1 ns;
        ma_sel <= q1 and q0 after 1 ns;
    end dfbc_a;

DFBCTL

library my_packages;
use my_packages.package_1.all;
use my_packages.package_2.all;

entity dfbctl is
port (inst: in bit;
req: in bit;
egz: in bit;
e: out bit);
end dfbctl;

architecture dfbctl_a of dfbctl is
    signal cot: bit;

    begin
        cot <= eqz after 2 ns when -- eqz
        begin
            inst(27 downto 26) = "00" and
            inst(31 downto 26) = "01" and
            inst(31 downto 26) = "10" and
            inst(31 downto 26) = "000001";
        end
        when inst(27 downto 26) = "00" and
        when inst(27 downto 26) = "01" and
        when inst(27 downto 26) = "10" and
        when inst(27 downto 26) = "000001";
        begin
            inst(16) = '0' and
            inst(31 downto 26) = "000001";
        end
        when inst(27 downto 26) = "00" and
        when inst(27 downto 26) = "01" and
        when inst(27 downto 26) = "10" and
        when inst(27 downto 26) = "000001";
        begin
            cot;
        end
    end dfbctl_a;

DFBJBOX

library my_packages;
use my_packages.package_1.all;
use my_packages.package_2.all;

library df_comp;
use df_comp.dfcomp.all;
use df_comp.dfcompall.all;

library df_comp;
use df_comp.dfjboxall.all;
use df_comp.dfjboxall16.all;

entity dfjbox is
    port (inst26: in bit_26;
p4: in bit_4;
reg: in bit_38;
b: in bit;
jt: in bit;
jr: in bit;
ce: in bit;
addr30: out bit_30);
end dfjbox;

architecture dfjbox_a of dfjbox is


library my_packages;
use my_packages.package_l.all;

architecture dfbussedc.a of dfbussedc is begin
  j <= '1' after 3.7 ns when -- jal or jalr
    (inst(31 downto 26) = "000011") or
    (inst(31 downto 26) = "001000" and
    inst(5 downto 0) = "001011") else
  '0' after 3.6 ns;
  s <= '1' after 3.6 ns when -- sxx

  "000000" and
  inst(5 downto 2) = "0001"
end dfbussedc.a;

DFCOMP

library my_packages;
use my_packages.package_l.all;

entity dfcomp is
  port(i: in bit_32;
    lts: out bit;
    eqz: out bit;
    gtz: out bit;
    o: out bit);
end dfcomp;

architecture dfcomp_a of dfcomp is begin
  o <= 1;
  lts <= 1(31);
  eqz <= '1' after 4.4 ns when 1(31) = '0' and 1 = x"0000_0000" else
    '0';
  gtz <= '1' after 3.1 ns when 1 = x"0000_0000" else
    '0';
end dfcomp_a;

DFCOMPARE

library my_packages;
use my_packages.package_l.all;
use std.textio.all;

entity dfcompare is
  port(compare: in question_type;
    compare_load: in question_type;
    pc_test: in bit_32;
    r1_test: in bit_32;
    r2_test: in bit_32;
    r3_test: in bit_32;
    r4_test: in bit_32;
    r31_test: in bit_32;
    hi_test: in bit_32;
    lo_test: in bit_32;
    compare_ack: out question_type;
    compare_load_ack: out question_type);
end dfcompare;

architecture dfcompare_a of dfcompare is begin
  compare_block: process
    type test_field_type is array (0 to 9) of bit_32;
    type test_array_type is array (0 to 1000) of test_field_type;
    variable expect: test_array_type;
    variable index: integer range 0 to 1000;
    file ini: text in "expected";
    variable line: line;
    variable inst: bit_32;
    variable skip: question_type;
    begin
      wait on compare, compare_load;
      if skip = '0' then
        if compare_load = yes then
          index := 0;
          while endfile(ini) = false loop
            if endline(ini) = false then
              readline(ini, line);
              expect(index) := inst;
              expect(index)(5) := inst;
              expect(index)(6) := inst;
              end if;
          end loop;
        end if;
      end if;
    end process;

    for i in 0 to 9 loop
      for j in 0 to 26 loop
        if expect(i) = line(j) then
          h(0) := h(0) + 1;
        end if;
      end loop;
    end loop;

  end if;
end dfcompare;
DFCPU

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfcomp.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
end component;

component dfcomp
port (init: in bit);
end component;

component dfalu
port (alu_start: in bit);
end component;

component dfmem
port (mem_start: in bit);
end component;

architecture dfcpu_a of dfcpu is

index := index + 1;
end loop;
index := 0;
skip := yes;
compare_load_ack <= yes after delay;
end if;
else
if compare = yes then
assert pc_test = expect(index)(0) report "PC REGISTER ERROR" severity warning;
assert r1_test = expect(index)(1) report "REGISTER 1 ERROR" severity warning;
assert r2_test = expect(index)(2) report "REGISTER 2 ERROR" severity warning;
assert r3_test = expect(index)(3) report "REGISTER 3 ERROR" severity warning;
assert r4_test = expect(index)(4) report "REGISTER 4 ERROR" severity warning;
assert r31_test = expect(index)(5) report "REGISTER 31 ERROR" severity warning;
assert hi_test = expect(index)(6) report "REGISTER HI ERROR" severity warning;
assert lo_test = expect(index)(7) report "REGISTER LO ERROR" severity warning;
index := index + 1;
compare_ack <= yes after delay;
end if;
else
compare_ack <= no after delay;
end if;
-- if skip...
end process compare_block;
end dfcompare_a;

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfcomp.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
use df_comp.dfalu.all;
end component;

component dfcomp
port (init: in bit);
end component;

component dfalu
port (alu_start: in bit);
end component;

component dfmem
port (mem_start: in bit);
end component;

architecture dfcpu_a of dfcpu is
-- component declarations
component dfcomp
port (init: in bit);
end component;

component dfalu
port (alu_start: in bit);
end component;

component dfmem
port (mem_start: in bit);
end component;
ma: in bit;
instr_in: in bit_32;
data_in: in bit_32;
add_bus: inout bus_bit_32;
data_bus: inout bus_bit_32;
mem_exo: out bit;
mx: out bit;
ml: out bit;
w: out bit;
opcode: out bit_32;
inst_out: out bit_32;
data_out: out bit_32;
vbt: out bit_4;
mem_done: out bit;
end component;

component dfwb
port (wb_start: in bit;
inst_in: in bit_32;
data_in: in bit_32;
vbt_in: in bit_4;
data_out: out bit_32;
vbt: out bit_4;
data_out: out bit_4;
mem_done: out bit);
end component;

component dfhe
port (illegal: in bit;
if_exo: in bit;
id_exo: in bit;
alu_exo: in bit;
mem_exo: in bit;
int_req: out bit;
int_vector: out bit_32);
end component;

component dfhrelat
port(s: in bit;
r: in bit;
q: out bit);
end component;

component dfhec
port (d: in bit_32;
c: in bit;
q: out bit_32);
end component;

-- system control
signal start: bit;
signal begin_in: bit;
-- pipeline handshake
signal begin_ok, if_ok, id_ok, alu_ok, mem_ok,
w_b: bit := '1';
signal if_ack, id_ack, alu_ack, mem_ack, wb_ack,
and_ack: bit;
signal if_start, id_start, alu_start, mem_start,
w_start: bit;
signal if_done, id_done, alu_done, mem_done,
w_done: bit;
signal init: bit;
-- interrupt/exception control
signal if_exo, id_exo, alu_exo, mem_exo: bit;
signal illegal, int_req: bit;
signal int_vector: bit_32;
signal int_latch_q: bit;
constant zero_constant: bit := '0';
signal int_reset: bit := zero_constant;
-- bus control
signal if_bus_req, mem_bus_req: bit;
signal if_bus_ack, mem_bus_ack: bit;
signal if_load_addr, mem_load_addr: bit;
-- data, control
library my_packages;
use my_packages.package_l.all;

library df_comp;
use df_comp.df32tolmux.all;
use df_comp.dftrds.all;

entity dfdbx is
  port (inst: in bit_32;
        db: in bit_32;
        hi_db: in bit;
        lo_db: in bit;
        ct_start: in bit;
        tl: in bit;
        t2: in bit;
        t3: in bit;
        th: in bit;
        jr: in bit;
        id_done: in bit;
        dirty_reg: out bit_5);
end dfdbx;

architecture dfbox_a of dfdbx is
  component dftrds
    port(i: in bit_32;
         pb: in bit_32;
         pc: in bit_32);
  end component;

  component df32tolmux
    port(i: in bit_32;
         o: out bit_32);
  end component;

begin
  clean <= not (a or b or c) after 1.1 ns;
  dirty_reg <= reg;
  c <= not ct_start after 0.3 ns;
  a <= not (d and e) after 0.7 ns;
  b <= not (f and g and h) after 0.8 ns;
  d <= not (hi_db and th) after 0.7 ns;
  e <= not (lo_db and tlo) after 0.7 ns;
  f <= not (ileg and n) after 0.7 ns;
  g <= not (trol and p) after 0.7 ns;
  h <= not (ctar and q) after 0.7 ns;
  mux1: df32tolmux
    port(i: in bit_32;
         o: out bit_32);
  end component;

  mux2: df32tolmux
    port(i: in bit_32;
         o: out bit_32);
  end component;

  trds: dftrds
    port(i: in bit_32;
         o: out bit_32);
  end component;

end dfbox_a;

architecture dfbox is
begin
  null;
end dfbox;
entity dffed is
port (i: in bit;
     o: out bit);
end dffed;

architecture dffed_a of dffed is
begin
  temp <= '1' after 0.4 ns when i'event and i = '0'
else
  '0' after 1.2 ns when temp = '1'
end architecture;

entity dfhcc is
port (init: in bit;
     prev_ok: in bit;
     ready: in bit;
     sin: in bit;
     sout: out bit;
     rout: out bit;
     ok: out bit);
end dfhcc;

architecture dfhcc_a of dfhcc is
begin
  dout <= '1' after 2.2 ns when prev_ok = '0'
and
  rout_temp = '0'
and
  '0' after 2.2 ns when prev_ok = '1'
else
  dout <= dout_temp;
  rout_temp <= '1' after 2.5 ns when rout_temp = '1'
and
  sin = '0' else
  '0' after 1.3 ns when sin = '1' else
  rout_temp;
  rout <= rout_temp;
  ok_temp <= '1' after 1 ns when rout_temp = '0'
else
  ok_temp;
end architecture;

entity dfhlreg is
port (id_start: in bit;
     hi_in: in bit;
     load_hi: in bit;
     load_lo: in bit;
     set_hi_db: in bit;
     set_lo_db: in bit;
     hi_out: out bit;
     lo_out: out bit;
     hi_db: out bit;
     lo_db: out bit);
end dfhlreg;

architecture dfhlreg_a of dfhlreg is
begin
  hi_reg <= hi_in after 0.3 ns when load_hi'event and load_hi = '1'
end architecture;

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfflatall;
use df_comp.dfreadall;
use df_comp.dfledall;
use df_comp.dfreg32all;
use df_comp.dfreg9all;
use df_comp.dfenteredall;
use df_comp.df32to1uxall;
use df_comp.df32to1uxall;
use df_comp.df32to1uxall;
use df_comp.dfboxall;
use df_comp.dfboxall;

entity dfid is
port (id_start: in bit;
     hi_db: in bit;
     lo_db: in bit;
     db: in bit_32;
     inst: in bit_32;
     cs: in bit;
     cod: in bit;
     po: in bit;
     reg: in bit_32;
     illegal: out bit;
     id_exx: out bit;
     lxo: out bit;
     alu_select: out bit;
     md_select: out bit;
     addr_select: out bit;
     reg1_out: out bit;
     reg2_out: out bit;
     treg_out: out bit;
     addr_valid: out bit;
     new_po: out bit;
     pol: out bit_32;
     inst_out: out bit_32;
     set_hi_db: out bit;
     set_lo_db: out bit;
     jff: out bit;
     id_done: out bit);
end dfid;
architecture dfid_a of dfid is

component dfred
  port(i: in bit;
  o: out bit);
end component;

component dfared
  port(i: in bit;
  o: out bit);
end component;

component dfss
  port(a: in bit;
  s: in bit;
  r: in bit;
  q: out bit);
end component;

component dfreg5
  port(d: in bit;
  o: in bit;
  q: out bit_5);
end component;

component dfreg
  port(d: in bit;
  o: in bit;
  q: out bit);
end component;

component dfss
  port(a: in bit;
  s: in bit;
  r: in bit;
  q: out bit);
end component;

component dfreg5
  port(d: in bit_5;
  o: in bit;
  q: out bit_5);
end component;

component dfreg
  port(d: in bit;
  o: in bit;
  q: out bit);
end component;

component dfreg5
  port(d: in bit_5;
  o: in bit;
  q: out bit_5);
end component;

component dfreg
  port(d: in bit;
  o: in bit;
  q: out bit);
end component;

begin
  inst decoder: dfinstdec
    port(map(instl, md_select, alu_select, lo, b, j, illegal, 
      ts1, ts2, t.tar, th, tbl, id_exc, r, l, trs0, trs1, 
      add8_select);
      dirty_box: dfdbox
        port(map(instl, db, hi_db, lo_db, ct_start, 
          ts1, ts2, t.tar, th, tbl, trs0, trs1, clean, treg_out);
          reg1 <= instl(25 downto 21);
          reg2 <= instl(20 downto 16);
          inst_out <= instl;
          reg1_latch: dfreg5
            port(map(reg1, id_done_r, reg1_out));
          reg2_latch: dfreg5
            port(map(reg2, id_done_r, reg2_out));
          ct_latch: dfslat
            port(map(av, id_done_r, ct_start);
          id_doneLatch: dfslat
            port(map(clean, id_start_r, id_done_out);
          id_done <= id_done_out;
          id_done_red: dfred
            port(map(id_done_out, id_done_r);
          avLatch: dfslat
            port(map(iX, av, av);
          add_val <= av;
          av_far: dfred
            port(map(iX, av);
          frd: dfred
            port(map(ir, ir);
          id_start_red: dfred
            port(map(id_start_r, id_done_r);
          id_start_red: dfred
            port(map(id_start_r, id_done_r);
          branch_latch: dfreg
            port(map(b, iX, b_sel);
          jump_latch: dfreg
            port(map(j, iX, j_sel);
          jump_reg_latch: dfreg
            port(map(r, iX, r_sel);
        dirty_reg: out bit_5);
end component;
architecture dfif_a of dfif is

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfrslat.all;
use df_comp.dffed.all;
use df_comp.dfreg32.all;
use df_comp.df2tolmux32.all;

entity dfif is
  port (if_start: in bit;
    ia: in bit;
    int_req: in bit;
    addr_valid: in bit;
    iv: in bit;
    new po: in bit;
    addr_bus: inout bus_bit_32;
    data_bus: inout bus_bit_32;
    ir: out bit;
    il: out bit;
    inst: out bit;
    po: out bit;
    if ex: out bit;
    if done: out bit);
end dfif;

architecture dfif_a of dfif is

component dfrslat
  port(i: in bit;
    o: in bit;
    q: out bit);
end component;

component dffed
  port(i: in bit;
    o: in bit;
    q: out bit);
end component;

component dfreg32
  port (d: in bit_32;
    i: in bit;
    o: out bit_32);
end component;

component df2tolmux32
  port (d0: in bit_32;
    d1: in bit_32;
    i: in bit;
    o: out bit_32);
end component;

component dfid_a
  port(map(data_bus, addr_bus, new_pc, addr_valid, int_req, addr30, b"00");
begin
  if_start_f when 1.3 ns after fstart, if_start_r when 1.3 ns after fstart;
  signal start_f, start_r, start_latch;
  signal ia_f, ia_r, ia_latched;
  signal lq_f, lq_r, lq_latched;
  signal ir_f, inst_latched;
end ifid_a;

library dfid_a;
port(map(ia, ia_f, ia_r, ia_latched, lq_f, lq_r, lq_latched, ir_f, inst_latched);
begin
  if_start_f when 1.3 ns after fstart, if_start_r when 1.3 ns after fstart;
  signal start_f, start_r, start_latch;
  signal ia_f, ia_r, ia_latched;
  signal lq_f, lq_r, lq_latched;
  signal ir_f, inst_latched;
end component;

begin
  f_latch: dfrslat
    port(map(start_x, la_f, ir);
  start_r when not (a or c) after 1 ns;
  a when not (int_req or b) after 1 ns;
  b when addr_valid after 0.3 ns;
  c when not if_start after 0.3 ns;
  start_red: dfrslat
    port(map(start, start_r);
  ia_red: dfrslat
    port(map(ia, ia_f);
  la_red: dfrslat
    port(map(ia, ia_f, ilq);
  ilq_f, inst_latched;
end ifid_a;
library my_packages;
use my_packages.package_1.all;

entity dfinstdec is
  port(i: in bit_32;
    alu: out bit;
    lbo: out bit;
    b: out bit;
    j: out bit;
    j1: out bit;
    j2: out bit;
    ts1: out bit;
    tstar: out bit;
    thl: out bit;
    tio: out bit;
    ex0: out bit;
    x: out bit;
    l: out bit;
    trs0: out bit;
    trs1: out bit;
    add: out bit);
  end dfinstdec;

architecture dfinstdec_a of dfinstdec is
begin
  alu <= '1' after 4.4 ns when -- a2
    (i(31) = '0' and
     i(29 downto 26) = "0000" and
     i(4 downto 3) = "11")
  else '0' after 4.4 ns;
    alu <= '1' after 4.4 ns when -- a2
    (i(31) = '0' and
     i(29 downto 26) = "0000" and
     i(5 downto 2) = "0010" and
     i(0) = '1' or
     i(1) = '0' and
     i(20 downto 26) = "0000" and
     i(3) = '0' and
     i(0) = '0' or
     -- a23
     (i(31) = '0' and
      i(29 downto 26) = "0000" and
      i(4 downto 3) = "00") or
      -- a20
      (i(27) = '0' and
       i(4 downto 3) = "10") or
       i(0) = '1' or
       -- a21
       (i(31) = '0' and
        i(29 downto 26) = "0000" and
        i(4 downto 3) = "11") or
        -- a25
        (i(31) = '0' and
         i(29 downto 26) = "0000" and
         i(5 downto 2) = "0000" and
         i(0) = '1' or
         -- a26
         (i(31) = '1' and
          i(29) = '1' or
          -- a27
          (i(31) = '0' and
           i(29) = '0' and
           i(27 downto 26) = "01") or
           -- a29
           (i(31) = '0' and
            i(29 downto 26) = "01") or
            -- a30
            (i(31) = '1' and
             i(29) = '0' or
             -- a31
             (i(31) = '0' and
              i(29) = '2') else
              "0" after 4.4 ns;

  lbo <= '1' after 4.4 ns when -- a26
    (i(31) = '1' and
     i(29) = '1') or
     -- a30
     (i(31) = '1' and
      i(29) = '0') or
      -- a31
      (i(31) = '0' and
       i(29) = '2') else
       "0" after 4.4 ns;
  b <= '1' after 4.4 ns when -- a27
    (i(31) = '0' and
     i(29) = '0' and
     i(27 downto 26) = "01") or
     -- a29
     (i(31) = '0' and
      i(29 downto 20) = "01") else '0' after 4.4 ns;
  j <= '1' after 4.4 ns when -- a4
    (i(31) = '0' and
     i(29 downto 26) = "0000" and
     i(5 downto 2) = "0010") or
     -- a33
     (i(31) = '0' and
      i(29 downto 27) = "001") else '0' after 4.4 ns;
  ill <= '1' after 4.4 ns when -- a1
    (i(31) = '0' and
     i(29 downto 26) = "0000" and
     i(5 downto 0) = "01") or
     -- a7
     (i(31) = '0' and
      i(29 downto 26) = "0001" and
      i(5 downto 0) = "01") or
      -- a5
      (i(31) = '0' and
       i(29 downto 26) = "001" and
       i(5 downto 2) = "01") or
       -- a8
       (i(31) = '0' and
        i(29 downto 26) = "011" or
        i(3 downto 0) = "11") or
        -- a10
        (i(31) = '0' and
         i(29 downto 26) = "111") or
         -- a11
         (i(31) = '0' and
          i(29 downto 26) = "0000" and
          i(4) = '1' and
          i(2) = '1') or
          -- a12
          (i(31) = '0' and
           i(29 downto 26) = "0000" and
           i(4) = '1' and
           i(2) = '1') or
           -- a16
           (i(31) = '1' and
            i(29 downto 27) = "11") or
            -- a17
            (i(31) = '0' and
             i(29 downto 26) = "0001" and
             i(4) = '1' and
             i(2) = '1') or
             -- a19
             (i(31) = '0' and
              i(29 downto 26) = "0001" and
              i(4) = '1') or
              "0" after 4.4 ns;

end dfinstdec_a;
"0001" and
    (1(31) = '0' and
     1(29 downto 26) =
     1(19) = '1') or
    -- a26
    1(30) = '1' else
    '0' after 4.4 ns;

    tl3 <= '1' after 4.4 ns when -- a4
    (1(31) = '0' and
     1(29 downto 26) =
     "0000" and
     "0010") or
    -- a15
    (1(31) = '0' and
     1(29 downto 26) =
     "0000" and
     or
    -- a26
    (1(27) = '0' and
     1(19) = '1') or
    -- a27
    (1(31) = '0' and
     1(29 downto 26) =
     "01") or
    -- a30
    (1(31) = '0' and
     1(29 downto 28) =
     "01") or
    -- a35
    (1(31) = '0' and
     1(29) = '0') or
    -- a36
    (1(31) = '0' and
     1(29) = '1') else
    '0' after 4.4 ns;

    tl2 <= '1' after 4.4 ns when -- a14
    (1(31) = '0' and
     1(29 downto 26) =
     "0000" and
     or
    -- a15
    (1(31) = '0' and
     1(29 downto 26) =
     "0000" and
     or
    -- a24
    (1(27) = '0' and
     1(29 downto 27) =
     "010") or
    -- a25
    (1(31) = '0' and
     1(29 downto 26) =
     "0000" and
     or
    -- a26
    (1(31) = '1' and
     1(29) = '1') else
    '0' after 4.4 ns;

    tttar <= '1' after 4.4 ns when -- a2
    (1(31) = '0' and
     1(29 downto 26) =
     "0000" and
     or
    -- a24
    (1(27) = '0' and
     1(29 downto 26) =
     "00010") else
    '0' after 4.4 ns;

    tr20 <= '1' after 4.4 ns when -- a23
    (1(31) = '0' and
     1(29 downto 27) =
     "0000" and
     or
    -- a26
    (1(31) = '1' and
     1(29 downto 27) =
     "001") or
    -- a27
    (1(31) = '0' and
     1(29) = '0' and
     1(27 downto 26) =
     '0' after 4.4 ns;
entity dfmd is
end

architecture dfmd_a of dfmd is

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfreg32.all;
use df_comp[dfred.all];
use df_comp.dfmemdec.all;
use df_comp.dfmu.all;
use df_comp.dftabl32.all;
use df_comp.df101mux3.all;

entity dfmem is
port (mem_start: in bit;
ns: in bit_32;
data_in: in bit_32;
data_out: in bit_32;
data_bus: inout bus_bit_32;
mem_exc: out bit;
m: out bit;
ml: out bit;
mls: out bit;
opcode: out bit_3);

mem_done: out bit);
end dfmem;

architecture dfmem_a of dfmem is

signal temp: bit_64;
signal dummy: bit;
signal done_temp2: bit;

begin

if x = y then
else

-- a29
(1(31) = ' 0' and 1(29 downto 28) = '01')

end

-- a30
(1(31) = ' 0' and 1(29 downto 28) = '0011')

end

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfreg32.all;
use df_comp.dfred.all;
use df_comp.dfmemdec.all;
use df_comp.dfmu.all;
use df_comp.dftabl32.all;
use df_comp.df101mux3.all;

entity dfmem is
port (mem_start: in bit;
ns: in bit_32;
data_in: in bit_32;
data_out: in bit_32;
data_bus: inout bus_bit_32;
mem_exc: out bit;
m: out bit;
ml: out bit;
mls: out bit;
opcode: out bit_3;
mem_done: out bit);
end dfmem;

architecture dfmem_a of dfmem is

component dfreg32
port (d: in bit_32;
o: in bit;
q: out bit_32);
end component;

component dfred
port(s: in bit;
z: in bit;
q: out bit);
end component;

component dfmemdec
port(start: in bit;
inst: in bit_32;
addr: in bit_32;
exo: out bit;
vbt: out bit_4;
store: out bit;
load: out bit;
sus1: out bit_3;
sus2: out bit_3;
sus3: out bit_3;
msl: out bit;
msg: out bit;
mus1: out bit;
mus2: out bit;
mus3: out bit;
done: out bit);
end component;

component dfmu
port(start: in bit;
inst: in bit_32;
data: in bit_32;
addr: in bit_32;
us1: in bit;
us2: in bit;
us3: in bit;
data_out: out bit_32;
end component;
entity dfmemdec is
  port(start: in bit);
  end component;

component dfmu
  port(start: in bit;
  mu_data: in bit_32;
  mus: in bit_2;
  mus1: in bit_2;
  sus2: in bit_2;
  sus3: in bit_3;
  data_out: out bit_32;
  done: out bit);
  end component;

component dftolmux32
  port(10: in bit_32;
  11: in bit_32;
  12: in bit_32;
  13: in bit_32;
  a: in bit;
  b: in bit;
  c: out bit_32);
  end component;

component dftolmux3
  port(10: in bit_3;
  11: in bit_3;
  a: in bit;
  b: in bit;
  c: out bit_3);
  end component;

signal mem_start_f, mem_start_r, bit;
signal inst_l, data1_l, data2_l, bit_32;
signal st, ld, pass, std, std_r, bit;
signal dummy_delay, mem_dec_done, store, load, bit;
signal sus3, mus, sus2, sus3: bit_3;
signal sus0, sus1, sus2, sus3: bit;
signal cm, cm_start, cm_start_r, bit;
signal mu_start, mu_start_latch, mu_data: bit;
begin
  inst_latch: dfrgl32
    port(map(inst_l, mem_start_r, inst_l);
  inst_out <= inst_l;
  data1latch: dfrg32
    port(map(data1_l, mem_start_r, data1_l);
  data2latch: dfrg32
    port(map(data2_l, mem_start_r, data2_l));
  mem_start_red: dfrd
    port(map(mem_start, mem_start_r);
  mem_start_red: dfrd
    port(map(mem_start, mem_start_r);
  mem_start_red: dfrd
    port(map(mem_start, mem_start_r);
  mem_done <= done;
  mem_done_red: dfrd
    port(map(don, done_r);
  dummy_delay <= '1' after 2 ns when mem_start = '1'
  else
    '0' after 1 ns;
  memory_decoder: dfmemdec
    port(map(dummy_delay, inst_l, data_1[1 downto 0], mem_exc, vbt, store, load, sus0, sus1, sus2, sus3, sus0, sus1, sus2, sus3, mem_dec_done);
  mem_start <= mem_start_done and (store or load) after 2.3 ns;
  mr_start_red: dfrd
    port(map(mr_start, mr_start_r);
  std <= st or ld or pass after 1.4 ns;
  std_red: dfrd
    port(map(std, std_r);
  st <= memdec_done and store and mu_start after 1.1 ns;
  ld <= memdec_done and load and mu_done after 1.1 ns;
  pass <= memdec_done and (not load) and (not store) after 1.4 ns;
  mr_start_latch: dfrslat
    port(map(mrq, done_r, mu_start);
  mu_data_latch: dfrg32
    port(map(data_bus, ma_f, data_bus_1);
  mask_unit: dfmu
    port(map(mu_start, inst_l, data_bus_1, data2_l[1 downto 0]), sus0, sus1, sus2, sus3, mu_data, su_start);
  shift_unit: dfmu
    port(map(start, mu_data, su0, su1, su2, su3, mu_data, su_done);
  data_mux: dftolmux3
    port(map(data1_l, mu_data, data2_l, data2_l, load, store, data_out);
  data_bus_tab: dftolmux3
    port(map(data2_l, db_tab_en, data_bus);
  addr_bus_tab: dftolmux3
    port(map(tab_in, ab_tab_en, addr_bus);
  tab_in <= data1_l[31 downto 0] & tab_in & tab_in;
  tbl_in <= data1_l[4] and store;
  tbl_in <= data1_l[1] and store;
  db_table_en <= store and ma after 1 ns;
  ab_table_en <= lons and ma after 1 ns;
  lons <= load or store after 1.3 ns;
  ma_red: dfrd
    port(map(ma, ma_r);
  ma_fed: dfrd
    port(map(ma_f, ma_f);
  mr_latch: dfrslat
    port(map(mr_start_r, ma_f, mrq);
  mr <= mrq;
  mrq_fed: dfrd
    port(map(mrq, mrq_f);
  ml_latch: dfrslat
    port(map(ma, ma_f, ml);
  w <= store and ma;
  type_mux: dfrtolux3
    port(map(default_rd_type, inst_l[28 downto 26], db_tab_en, opcode))
end dfmemdec;
architecture dfmdecode is

signal vbt0, vbt1, vbt2, vbt3: bit;
signal sus00, sus01, sus10, sus11: bit;
signal sus20, sus21, sus30, sus31: bit;

begin

done <= '1' after 5.3 ns when start = '1' else '0' after 1 ns;
exo <= '1' after 4.3 ns when -- a3

(vbt1 <= '1' after 4.3 ns when -- a13
 (inst(31) = '1' and inst(29 downto 27) = "011" and
 addr = '01')

"011" and
addr = '00') or
-- a14

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a15

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a16

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a17

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a18

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a19

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a20

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a21

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a22

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a23

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a24

(inst(31) = '1' and
inst(29 downto 27) = "011" and
addr = '00') or
-- a25

(inst(31) = '0')

end dfmdecode;
vbc3 <= '1' after 4.3 ns when -- a13
   (inst(31) = '1' and inst(29 downto 27) = '00' or
     inst(27 downto 26) = '11' or
     inst(27 downto 26) = '00' and inst(29 downto 26) = '11') or
   '0' after 4.3 ns;

store <= '1' after 4.3 ns when -- a12
   (inst(31) = '1' and inst(29) = '0' and
    inst(27) = '0' and
    addr = '10') or
   '0' after 4.3 ns;

load <= '1' after 4.3 ns when -- a9
   (inst(31) = '1' and inst(29) = '1')
else
   '0' after 4.3 ns;

sus01 <= '1' after 4.3 ns when -- a13
   (inst(31) = '1' and inst(29 downto 27) = '00' or
     inst(27 downto 26) = '11') or
   '0' after 4.3 ns;

sus10 <= '1' after 4.3 ns when -- a22
   (inst(31) = '1' and inst(29) = '0' and
    inst(27) = '0' and
    addr = '00') or
   '0' after 4.3 ns;

sus00 <= '1' after 4.3 ns when -- a12
   (inst(31) = '1' and inst(29) = '0' and
    addr = '11') or
   '0' after 4.3 ns;

addr = '10' or
-- a19 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a17 (inst(31) = '1' and inst(29 downto 27) = '11') or
-- a16 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a15 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a14 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a13 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a12 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a11 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a10 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a9 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a8 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a7 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a6 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a5 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a4 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a3 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a2 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a1 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a0 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a9 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a8 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a7 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a6 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a5 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a4 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a3 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a2 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a1 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a0 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a9 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a8 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a7 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a6 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a5 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a4 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a3 (inst(31) = '1' and inst(29 downto 27) = '00') or
-- a2 (inst(31) = '1' and inst(29 downto 27) = '01') or
-- a1 (inst(31) = '1' and inst(29 downto 27) = '10') or
-- a0 (inst(31) = '1' and inst(29 downto 27) = '00') or

sus11 <= '1' after 4.3 ns when -- a19  
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(29 \text{ downto } 27) = 011 \text{ and } \text{addr} = "010" \text{ or } \text{-- a13} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(29 \text{ downto } 27) = 011 \text{ and } \text{addr} = "011" \text{ or } \text{-- a25} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(29 \text{ downto } 27) = \text{else} \text{-- a25} \]

sus21 <= '1' after 4.3 ns when -- a19  
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(29 \text{ downto } 27) = 011 \text{ and } \text{addr} = "010" \text{ or } \text{-- a23} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(29 \text{ downto } 27) = 011 \text{ and } \text{addr} = "011" \text{ or } \text{-- a24} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(29 \text{ downto } 27) = \text{else} \text{-- a24} \]

sus31 <= '1' after 4.3 ns when -- a16  
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = 01 \text{ and } \text{addr} = "00" \text{ or } \text{-- a22} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = 01 \text{ and } \text{addr} = "01" \text{ or } \text{-- a27} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = \text{else} \text{-- a27} \]

sus31 <= '1' after 4.3 ns when -- a16  
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = 01 \text{ and } \text{addr} = "00" \text{ or } \text{-- a19} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = 01 \text{ and } \text{addr} = "01" \text{ or } \text{-- a13} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = \text{else} \text{-- a13} \]

sus30 <= '1' after 4.3 ns when -- a20  
\[ \text{inst}(28 \text{ downto } 27) = 01 \text{ and } \text{addr} = '0' \text{ or } \text{-- a31} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(29 \text{ downto } 27) = 01 \text{ and } \text{addr} = '0' \text{ or } \text{-- a32} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(29 \text{ downto } 27) = \text{else} \text{-- a32} \]

sus20 <= '1' after 4.3 ns when -- a11  
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = 01 \text{ and } \text{addr} = "010" \text{ or } \text{-- a17} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = 01 \text{ and } \text{addr} = "011" \text{ or } \text{-- a21} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = \text{else} \text{-- a21} \]

sus2 <= '1' after 4.3 ns when -- a11  
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = 01 \text{ and } \text{addr} = '00' \text{ or } \text{-- a16} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = 01 \text{ and } \text{addr} = '01' \text{ or } \text{-- a17} \]
\[ \text{inst}(31) = '1' \text{ and } \text{inst}(28 \text{ downto } 27) = \text{else} \text{-- a17} \]
use std.textio.all;

library my_packages;
use my_packages.package_1.all;
use my_packages.dfpack.all;

entity dfmemory is
  port (load: in question_type;
     req: in bit;
     w: in bit;
     opcode: in bit_3;
     ack: out bit;
     load_ack: out question_type;
     addr_bus: inout bus_bit_32 bus;
     data_bus: inout bus_bit_32 bus);
end dfmemory;

architecture dfmemory_a of dfmemory is
  signal ack_temp: bit;
  signal addr_temp: integer;
  signal m4000, m4004, m4008, m400c, m4010: bit_32;
  signal m414, m418, m41c, m420, m424: bit_32;

begin
  memory: process
    constant low_address: integer := 0;
    constant high_address: integer := 65535;
    type memory_array is
      array (integer range low_address to high_address) of bit_32;
    variable mem: memory_array;
    variable addr: integer range 0 to high_address;
    variable temp: bit_32;
    file inl: text is in "machine";
    variable linel: line;
    variable inst: bit_32;
    variable skip: question_type;
    variable addr_word: bit_32;
    variable addr_byte: bit_2;
  begin
    wait on req, load;
    if skip = no then
      if load = yes then
        addr := 0;
        while endfile(inl) = false loop
          readline(inl, linel);
          read(linel, inst);
          mem(addr) := inst;
          addr := addr + 1;
          end loop;
        skip := yes;
        load_ack <= yes after delay;
        end if;
      else
        if req = '1' then
          addr_word := addr_bus(31 downto 2);
          addr_byte := addr_bus(1 downto 0);
          -- send acknowledge signal to bus
        end if;
      controller
        ack <= '1' after 5 ns;
        ack_temp <= '1' after 5 ns;
        assert addr_word(29 downto 14) = x'0000' report "ADDRESS OUT OF RANGE";
        if addr_word(29 downto 14) = x'0000' then
          if w = '0' then
            addr := btrim(addr_word);
            temp := mem(addr);
          -- read operations
          end if
        else
          if opcode = m_lb then
            case addr_byte is
              when b'00' =>
                data_bus <= seento32(temp(31 downto 24));
              when b'01' =>
                data_bus <= seento32(temp(27 downto 16));
              when b'10' =>
                data_bus <= seento32(temp(30 downto 25));
            end case;
          end if;
        end if;
      end controller
    end if;
  end memory:
end dfmemory_a;

data_bus <= se8to32(temp(15 downto 0));
when b"11" =>
data_bus <= se8to32(temp(7 downto 0));
end case;
elif opcode = m_lh then
case addr_byte is
when b"00" =>
data_bus <= temp(31 downto 8);data_bus(7 downto 8) <=
when b"01" =>
data_bus <= temp(31 downto 0); data_bus <= x"00";
else
-- reserved instructions
end if; -- end read operations
else
addr := bton(addr_word);
-- write operations
if opcode = m_sh then
case addr_byte is
when b"00" =>
mem(addr(31 downto 24)) :=
data_bus(7 downto 0);
when b"01" =>
mem(addr(23 downto 16)) :=
data_bus(7 downto 0);
when b"10" =>
mem(addr(15 downto 8)) :=
data_bus(7 downto 0);
when b"11" =>
mem(addr(7 downto 0)) :=
end case;
elif opcode = m_lw then
case addr_byte is
when b"00" =>
data_bus(15 downto 0) :=
data_bus(15 downto 0);
when b"01" =>
mem(addr(15 downto 0)) :=
data_bus(15 downto 0);
else
-- flag address exception
end case;
elif opcode = m_sw then
case addr_byte is
when b"00" =>
mem(addr) := data_bus;
when b"01" =>
mem(addr(23 downto 0)) :=
data_bus(31 downto 8);
when b"10" =>
mem(addr(15 downto 0)) :=
data_bus(31 downto 16);
when b"11" =>
mem(addr(7 downto 0)) :=
data_bus(31 downto 24);
end case;
elif opcode = mlw then
case addr_byte is
when b"00" =>
data_bus <= x"000000" &
temp(31 downto 24);
when b"01" =>
data_bus <= x"000000" &
temp(23 downto 16);
when b"10" =>
data_bus <= x"000000" &
temp(15 downto 8);
when b"11" =>
data_bus <= x"000000" &
temp(7 downto 0);
end case;
etc.
library my_packages;
use my_packages.package_l.all;
library df4tolmux32.all;
use df4tolmux32.all;
entity dfoutsel is
  port(in: in bit_32;
  alu: in bit_32;
  hilo: in bit_32;
  output: out bit_32);
end dfoutsel;
architecture dfoutsel_a of dfoutsel is
  component df4tolmux32
    port(10: in bit_32;
    11: in bit_32;
    12: in bit_32;
    13: in bit_32;
    14: in bit;
    15: in bit;
    16: in bit;
    17: in bit;
    18: in bit;
    19: in bit;
    20: in bit;
    data_out: out bit_32;
    done: out bit);
  end component;
begin
  signal s0, s1: bit;
  signal sO, si, sN: bit;
  signal sO, si: bit;
  signal s0, s1: bit;
begin
  done <= '1' after 3.4 ns when start = '1' else '0' after 1 ns;
  mux0: df4tolmux8
    port map(s0, data(7 downto 0), seb, data_out(7 downto 0));
  mux1: df4tolmux8
    port map(s1, data(15 downto 8), mus0, data_out(15 downto 8));
  mux2: df4tolmux8
    port map(s1, data(23 downto 16), mus2, data_out(23 downto 16));
  mux3: df4tolmux8
    port map(s1, data(31 downto 24), mus3, data_out(31 downto 24));
  se_byte <= s0_nibble & s1_nibble;
  se_nibble <= seb & seb & seb & seb;
end dfoutsel_a;
architecture dfoutsel_a of dfoutsel is
  component df4tolmux32
    port(10: in bit_32;
    11: in bit_32;
    12: in bit_32;
    13: in bit_32;
    14: in bit;
    15: in bit;
    16: in bit;
    data_out: out bit_32;
    done: out bit);
  end component;
begin
  signal s0, s1: bit;
begin
  done <= '1' after 3.4 ns when start = '1' else '0' after 1 ns;
  mux0: df4tolmux8
    port map(s0, data(7 downto 0), seb, data_out(7 downto 0));
  mux1: df4tolmux8
    port map(s1, data(15 downto 8), mus0, data_out(15 downto 8));
  mux2: df4tolmux8
    port map(s1, data(23 downto 16), mus2, data_out(23 downto 16));
  mux3: df4tolmux8
    port map(s1, data(31 downto 24), mus3, data_out(31 downto 24));
  se_byte <= s0_nibble & s1_nibble;
  se_nibble <= seb & seb & seb & seb;
end dfoutsel_a;
architecture dfoutsel_a of dfoutsel is
  component df4tolmux32
    port(10: in bit_32;
    11: in bit_32;
    12: in bit_32;
    13: in bit_32;
    14: in bit;
    15: in bit;
    16: in bit;
    data_out: out bit_32;
    done: out bit);
  end component;
begin
  signal s0, s1: bit;
begin
  done <= '1' after 3.4 ns when start = '1' else '0' after 1 ns;
  mux0: df4tolmux8
    port map(s0, data(7 downto 0), seb, data_out(7 downto 0));
  mux1: df4tolmux8
    port map(s1, data(15 downto 8), mus0, data_out(15 downto 8));
  mux2: df4tolmux8
    port map(s1, data(23 downto 16), mus2, data_out(23 downto 16));
  mux3: df4tolmux8
    port map(s1, data(31 downto 24), mus3, data_out(31 downto 24));
  se_byte <= s0_nibble & s1_nibble;
  se_nibble <= seb & seb & seb & seb;
end dfoutsel_a;

add instruction

(begin

(temp <= '1' after 0.3 ns when l'event and l = '1'
else
  '0' after 1.1 ns when temp = '1' else

begin

(temp <= '1' after 0.3 ns when l'event and l = '1'
else
  '0' after 1.1 ns when temp = '1' else

end dfr5_a;

library my_packages;
use my_packages.package_1.all;

entity dfreg5 is
  port (d: in bit; c: in bit; q: out bit);
end dfreg5;

architecture dfreg5_a of dfreg5 is
  signal temp: bit;
begin
  temp <= '1' after 0.3 ns when c'event and c = '1'
else
    '0' after 1.1 ns when temp = '1' else
end dfreg5_a;

library my_packages;
use my_packages.package_1.all;

entity dfregbank_a is
  port (data: in bit_32;
    reg_sel: in bit_5;
    a_sel: in bit_5;
    b_sel: in bit_5;
    write: in bit;
    bton: in bit;
    vbt: in bit_4;
    a: out bit_32;
    b: out bit_32;
    db: out bit_32;
    r1: out bit_32;
    r2: out bit_32;
    r3: out bit_32;
    r4: out bit_32;
    r5: out bit_32;
end dfregbank_a;

architecture dfregbank_a of dfregbank is
  type register_byte is array (0 to 31) of bit_8;
  signal s, r1, r2, r3: register_byte;
  signal an, bn, regn, dbn: bit_5_range;
  signal db_id, db_wb: bit_32;
  signal d0, d1, d2, d3: bit_8;
begin
  an <= bton(a_sel);
  bn <= bton(b_sel);
  regn <= bton(reg_sel);
  dbn <= bton(db_sel);
  d0 <= data(7 downto 0);
  d1 <= data(15 downto 8);
  d2 <= data(23 downto 16);
  d3 <= data(31 downto 24);
  r0(regn) <= d0 after 0.3 ns when
          regn /= 0 and vbt(0) = '1' and
          write'event and write = '1' else
  r1(regn) <= d1 after 0.3 ns when
          regn /= 0 and vbt(1) = '1' and
          write'event and write = '1' else
  r2(regn) <= d2 after 0.3 ns when
          regn /= 0 and vbt(2) = '1' and
          write'event and write = '1' else
  r3(regn) <= d3 after 0.3 ns when
          regn /= 0 and vbt(3) = '1' and

C-27
write'event and write = '1' else r3(regn);
A <= r3(ans) & r2(ans) & r1(ans) & r0(ans) after 0.6 ns;
b <= r3(bns) & r2(bns) & r1(bns) & r0(bns) after 0.6 ns;
db_id(dbn) <= db_id(dbn) xor '1' after 1 ns when start'event and start = '1' else db_id(dbn);
db_wb(regn) <= db_wb(regn) xor '1' after 1 ns when write'event and write = '1' else db_wb(regn);
db <= (db_id xor db_wb) and x"ffffffffff" after 1 ns;
rl_test <= r3(4) & r2(4) & r1(4) & r0(4);
rl_test <= r3(31) & r2(31) & r1(31) & r0(31);
end dfregbank_a;

---

library my_packages;
use my_packages.package_l.all;
entity dfsctl is
my_packages.package_l.all;
use
my_packages;
library
dfrslat
end entity dfsctl;

architecture dfsctl_a of dfsctl is
begin
temp <= 4 after 0.9 ns when c'event and c = '1' and r = '0' else '0' after 0.3 ns when r = '1' else temp;
q <= temp;
end dfsctl_a;

dfrslat is
begin
temp <= '1' after 1 ns when s = '1' and r = '0' else '0' after 0.4 ns when s = '0' and r = '1' else temp;
q <= temp;
end dfsrlat_a;

library my_packages;
use my_packages.package_l.all;
entity dfsctl is
port (inst in bit_32;
a: in bit_5;
li: in bit;
la: out bit;
sel: out bit_5);
end dfsctl;

architecture dfsctl_a of dfsctl is
begin
s <= '1' when inst(31 downto 26) = "000000" and inst(5 downto 3) = "000" else '0';
lui <= '1' when inst(31 downto 26) = "001111" else '0';
li <= '0' after 2.7 ns when lui = '1' else inst(1) after 1 ns;
la <= '0' after 2.7 ns when lui = '1' else inst(0) after 1 ns;
sel <= "100000" after 2.7 ns when lui = '1' else inst(10 downto 6) after 2.7 ns when
s = '1' and inst(2) = '0' else a after 2.7 ns when
s = '1' and inst(2) = '1' else "000000" after 2.7 ns;
end dfsctl_a;

---

DFSHIFT

library my_packages;
use my_packages.package_l.all;
entity dfshift is
port (i in bit_32;
lr in bit;
la in bit;
sel in bit_5;
o out bit_32);
end dfshift;

architecture dfshift_a of dfshift is
begin
o <= shift_ll(i, bton(sel)) after 3.3 ns when
s = '0' else shift_rl(i, bton(sel)) after 3.3 ns when
s = '0' else shift_ra(i, bton(sel)) after 3.3 ns when
s = '1' else 1 after 3.3 ns;
end dfshift_a;

---

dfsit is
begin
temp <= x"00000000" & "000" & lts;
o <= temp after 3.5 ns when -- slti, sltiu inst(31 downto 27) = "001011" or
-- slti, sltu inst(31 downto 26) = "000000" and inst(5 downto 1) = "101011" else
1 after 3.5 ns;
end dfsit_a;
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dftsr32.all;

entity dftrds is
port (inst: in bit;
    trs0: in bit;
    trs1: in bit;
    reg: out bit_5);
end dftrds;

architecture dftrds_a of dftrds is
begin
    reg <= inst(15 downto 11) after 0.4 ns when
    -- special
        trs0 = '0' and trs1 = '0'
    else
        inst(20 downto 16) after 0.4 ns when
        -- imm or load
    end if;
end dftrds_a;

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.df2tolmux.all;

entity df2tolmux is
end entity df2tolmux;

architecture df2tolmux_a of df2tolmux is
component df4tolmux8
    port (10: in bit_8;
        9: in bit_8;
        8: in bit_8;
        7: in bit_8;
        6: in bit;
        5: in bit;
        4: out bit_8;
        3: out bit);
end component;

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dftsr32.all;

entity df0tolmux is
end entity df0tolmux;

architecture df0tolmux_a of df0tolmux is
component df4tolmux8
    port (10: in bit_8;
        9: in bit_8;
        8: in bit_8;
        7: in bit_8;
        6: in bit;
        5: in bit;
        4: out bit_8;
        3: out bit);
end component;

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dffed.all;

entity dfred is
end entity dfred;

architecture dfred_a of dfred is
component dfreg32
    port (d: in bit_32;
        c: in bit;
        q: out bit_32);
end component;

component dfssl atl
    port (s: in bit;
        r: in bit;
        q: out bit);
end component;

component dfred
    port (i: in bit;
        o: out bit);
end component;

component dfssl atl
    port (s: in bit;
        r: in bit;
        q: out bit);
end component;

component dfred
    port (i: in bit;
        o: out bit);
end component;

library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.df2tolmux.all;

entity df2tolmux is
end entity df2tolmux;

architecture df2tolmux_a of df2tolmux is
component df4tolmux8
    port (10: in bit_8;
        9: in bit_8;
        8: in bit_8;
        7: in bit_8;
        6: in bit;
        5: in bit;
        4: out bit_8;
        3: out bit);
end component;
component dfreg4
    port (d: in bit_4;
         c: in bit;
         q: out bit_4);
end component;

signal wb_start_r, wb_start_f, dummy_r, dummy; bit;
signal trsO, trsl; bit;
signal instl; bit_32;
signal reg; bit_5;
signal moveto_sel, hilo_sel; bit;
signal count: integer := 0;
begin
inst_latch: dfreg32
    port map(inst_in, wb_start_r, instl);
data_latch: dfreg32
    port map(data_in, wb_start_r, wb_data);
vbt_latch: dfreg4
    port map(vbt_in, wb_start_r, vbt);
wb_start_red: dfred
    port map(wb_start, wb_start_r);
wb_start_fed: dffed
    port map(wb_start, wb_start_f);
done_latch: dfslat
    port map(dummy_r, wb_start_f, wb_done);
dummy <= '1' after 8 ns when wb_start='1' event and
         wb_start = '1' else
         '0' after 1 ns;
dummy_red: dfred
    port map(dummy, dummy_r);
trs: dftrds
    port map(instl, trsO, trsl, reg);
trsO <= '1' after 4.4 ns when -- a23
    [instl(31) = '0' and
     instl(29 downto 27) =
     "001"] or
    -- a26
     [instl(31) = '1' and
      instl(29) = '1'] or
    -- a27
     [instl(31) = '0' and
      instl(29) = '0' and
      instl(27 downto 24) =
     "01"] or
    -- a29
     [instl(31) = '0' and
      instl(29 downto 28) =
     "01"] else
         '0' after 4.4 ns;
trsl <= '1' after 4.4 ns when -- a21
    [instl(31) = '0' and
     instl(29 downto 26) =
     "01"] or
    -- a30
     [instl(31) = '1' and
      instl(29) = '0'] or
    -- a31
     [instl(31) = '0' and
      instl(29) = '1'] or
    -- a32
     [instl(31) = '0' and
      instl(29 downto 26) =
     "0001" and
      instl(20) = '1'] else
         '0' after 4.4 ns;
wb_sel <= moveto_sel & reg(4 downto 1) & hilo_sel;
moveto_sel <= '1' after 2.6 ns when -- special and
move to instr.
    instl(31 downto 26) =
    "000000" and
    instl(5 downto 4) = "01" and
    instl(0) = '1' else
         '0' after 2.6 ns;
wb_sel_mux: df2tolmux
    port map(reg(0), instl(1), moveto_sel, hilo_sel);
count <= count + 1 when (wb_start='1' event and
    wb_start = '1') else count;
APPENDIX D - STRUCTURAL MODEL SOURCE CODE

COMPONENT: STMIPS
FILENAME: STMIPS.E.VHDL
DESCRIPTION: Test bench for structural model of
synchronous version of MIPS R3000 microprocessor (entity)

library my_packages;
use my_packages package_1.all;
use my_packages df_comp.all;

library df_comp;
use df_comp dfmemory.all;

library st_comp;
use st_comp stcpu.all;

entity stcpu is
end stcpu;

component stcpu
is
port
  sys_control: in std_logic_vector;
  memory_ack: in bit;
  memory_load_ack: in bit;
  addr_bus: inout bus_bit_32 bus;
  data_bus: inout bus_bit_32 bus;
  memory_req: out bit;
  memory_opcode: out bit;
  memrory_load: out question_type;
  memory_load_ack: out question_type;
  memory_load: out question_type;
  memory_load_ack: out question_type;
  compare, memory_load: out question_type;
  compare_load: out question_type;
  compare_load_ack: out question_type;
  compare: out question_type;
  compare_load: out question_type;
  compare_load_ack: out question_type;
  compare: out question_type;
  compare_load: out question_type;
  compare_load_ack: out question_type;
  compare: out question_type;
  compare_load: out question_type;
  compare_load_ack: out question_type;
begin
  begin
    end begin;
end component;

COMPONENT: STMIPS_A
FILENAME: STMIPS_A.VHDL
DESCRIPTION: Test bench for structural model of
synchronous version of MIPS R3000 microprocessor (architecture)

architecture stcpu_a of stcpu is
component stcpu
end component;

component dfmemory
port
  req: in bit;
  w: in bit;
  ocpes: in bit_3;
  ack: out bit;
  load_ack: out question_type;
  addr_bus: inout bus_bit_32 bus;
  data_bus: inout bus_bit_32 bus;
end component;

component dfcompare
port
  compare: in question_type;
  compare_load: in question_type;
  compare_load_ack: out question_type;
end component;

signal sys_control, sys_control_type;
signal memory_load, memory_load_ack, question_type;
signal memory_req, memory_ack, memory_w: bit;
signal memory_opcode: bit_3;
signal addr_bus, data_bus, bus_bit_32;
signal compare_ack, compare_load_ack;
signal compare, compare_load, question_type;
signal compare, memory_ack, memory_w, memory_opcode, memory_load, compare;
signal compare_load, memory_load_ack, data_bus;

begin
  cpu_module:
    port
      memory_load_ack, compare_ack, compare_load, memory_req, memory_w, memory_opcode, memory_load, compare, compare_load, memory_load_ack, data_bus;
    component
      dfmemory
    port
      map(memory_load, memory_load_ack, memory_req, memory_w, memory_opcode, memory_load, compare, compare_load, memory_load_ack, data_bus);
    component
      dfcompare
    port
      map(compare, compare_load, memory_load, compare_ack, compare_load_ack, compare_load, compare_load_ack, compare, compare_load, compare_load_ack, compare, compare_load, compare_load_ack, compare, compare_load, compare_load_ack, compare);
    end component;
end component;

COMPONENT: STPACK
FILENAME: STPACK.B.VHDL
DESCRIPTION: Structural model package (header)

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;
package stpack is
  subtype slv_2 is std_logic_vector (1 downto 0);
  subtype slv_3 is std_logic_vector (2 downto 0);
  subtype slv_4 is std_logic_vector (3 downto 0);
  subtype slv_5 is std_logic_vector (4 downto 0);
  subtype slv_6 is std_logic_vector (5 downto 0);
  subtype slv_7 is std_logic_vector (6 downto 0);
  subtype slv_8 is std_logic_vector (7 downto 0);
  subtype slv_9 is std_logic_vector (8 downto 0);
  subtype slv_10 is std_logic_vector (9 downto 0);
  subtype slv_11 is std_logic_vector (10 downto 0);
  subtype slv_12 is std_logic_vector (11 downto 0);
  subtype slv_13 is std_logic_vector (12 downto 0);
  subtype slv_14 is std_logic_vector (13 downto 0);
  subtype slv_15 is std_logic_vector (14 downto 0);
  subtype slv_16 is std_logic_vector (15 downto 0);
  subtype slv_17 is std_logic_vector (16 downto 0);
  subtype slv_18 is std_logic_vector (17 downto 0);
  subtype slv_19 is std_logic_vector (18 downto 0);
  subtype slv_20 is std_logic_vector (19 downto 0);
  subtype slv_21 is std_logic_vector (20 downto 0);
  subtype slv_22 is std_logic_vector (21 downto 0);
  subtype slv_23 is std_logic_vector (22 downto 0);
  subtype slv_24 is std_logic_vector (23 downto 0);
  subtype slv_25 is std_logic_vector (24 downto 0);
  subtype slv_26 is std_logic_vector (25 downto 0);
  subtype slv_27 is std_logic_vector (26 downto 0);
  subtype slv_28 is std_logic_vector (27 downto 0);
  subtype slv_29 is std_logic_vector (28 downto 0);
  subtype slv_30 is std_logic_vector (29 downto 0);
  subtype slv_31 is std_logic_vector (30 downto 0);
  subtype slv_32 is std_logic_vector (31 downto 0);
  subtype slv_33 is std_logic_vector (32 downto 0);
  subtype slv_34 is std_logic_vector (33 downto 0);
  subtype slv_35 is std_logic_vector (34 downto 0);
  subtype slv_36 is std_logic_vector (35 downto 0);
  subtype slv_37 is std_logic_vector (36 downto 0);
  subtype slv_38 is std_logic_vector (37 downto 0);
  subtype slv_39 is std_logic_vector (38 downto 0);
  subtype slv_40 is std_logic_vector (39 downto 0);
  subtype slv_41 is std_logic_vector (40 downto 0);
  subtype slv_42 is std_logic_vector (41 downto 0);
  subtype slv_43 is std_logic_vector (42 downto 0);
  subtype slv_44 is std_logic_vector (43 downto 0);
  subtype slv_45 is std_logic_vector (44 downto 0);
  subtype slv_46 is std_logic_vector (45 downto 0);
  subtype slv_47 is std_logic_vector (46 downto 0);
  subtype slv_48 is std_logic_vector (47 downto 0);
  subtype slv_49 is std_logic_vector (48 downto 0);
  subtype slv_50 is std_logic_vector (49 downto 0);
  subtype slv_51 is std_logic_vector (50 downto 0);
  subtype slv_52 is std_logic_vector (51 downto 0);
  subtype slv_53 is std_logic_vector (52 downto 0);
  subtype slv_54 is std_logic_vector (53 downto 0);
  subtype slv_55 is std_logic_vector (54 downto 0);
  subtype slv_56 is std_logic_vector (55 downto 0);
  subtype slv_57 is std_logic_vector (56 downto 0);
  subtype slv_58 is std_logic_vector (57 downto 0);
  subtype slv_59 is std_logic_vector (58 downto 0);
  subtype slv_60 is std_logic_vector (59 downto 0);
  subtype slv_61 is std_logic_vector (60 downto 0);
  subtype slv_62 is std_logic_vector (61 downto 0);
  subtype slv_63 is std_logic_vector (62 downto 0);
  subtype slv_64 is std_logic_vector (63 downto 0);
end stpack;

COMPONENT: STRUCUARL COMPONENTS
FILENAME: ???_E.VHDL for entity
and ???.A.VHDL for architecture
DESCRIPTION: All components used in structural model follow (entity shown first)
library ieee;
use ieee.std_logic_1164.all;
useieee.std_logic_1164 extensions.all;

library my_packages;
use my_packages.all;

library std_comp;
use std_comp.all;

entity stalu32 is
port(a: in std_logic);
begin
alu2to15: stalu4
port map(a(12), a(13), a(14), a(15), b(12), b(13), b(14), b(15),
ocenl_o2, a0, a1, a2, a3, o(12),
g3, p3, cout3);
alu18to19: stalu4
port map(a(16), a(17), a(18), a(19), b(16), b(17), b(18), b(19),
cout3, a0, a1, a2, a3, o(16), o(17),
o(18), o(19),
g4, p4, cout4);
alu24to27: stalu4
port map(a(24), a(25), a(26), a(27), b(24), b(25), b(26), b(27),
ocenl_o2, a0, a1, a2, a3, o(24),
o(25), o(26), o(27),
g6, p6, cout6);
alu28to31: stalu4
port map(a(28), a(29), a(30), a(31), b(28), b(29), b(30), b(31),
cout6, a0, a1, a2, a3, o(28), o(29),
o(30), o(31),
g7, p7, cout7);
end stalu32;

architecture stalu32_a of stalu32 is
component stalu4
port(a: in std_logic);
begin
alu0to3: stalu4
port map(a(0), a(1), a(2), a(3), b(0), b(1),
b(2), b(3),
cin, a0, a1, a2, a3, o(0), o(1), o(2),
o(3),
g0, p0, cout0);
alu1to4: stalu4
port map(a(4), a(5), a(6), a(7), b(4), b(5),
b(6), b(7),
cout0, a0, a1, a2, a3, o(4), o(5),
o(6), o(7),
g1, p1, cout1);
alu5to11: stalu4
port map(a(8), a(9), a(10), a(11), b(8), b(9),
b(10), b(11),
ocenl_o1, a0, a1, a2, a3, o(8), o(9),
o(10), o(11),
g2, p2, cout2);
alu12to15: stalu4
port map(a(12), a(13), a(14), a(15), b(12), b(13), b(14), b(15),
ocenl_o2, a0, a1, a2, a3, o(12),
g3, p3, cout3);
alu18to19: stalu4
port map(a(16), a(17), a(18), a(19), b(16), b(17), b(18), b(19),
cout3, a0, a1, a2, a3, o(16), o(17),
o(18), o(19),
g4, p4, cout4);
alu24to27: stalu4
port map(a(24), a(25), a(26), a(27), b(24), b(25), b(26), b(27),
ocenl_o2, a0, a1, a2, a3, o(24),
o(25), o(26), o(27),
g6, p6, cout6);
alu28to31: stalu4
port map(a(28), a(29), a(30), a(31), b(28), b(29), b(30), b(31),
cout6, a0, a1, a2, a3, o(28), o(29),
o(30), o(31),
g7, p7, cout7);
end component;

end architecture;
signal n51, n52, n53, n54, n55, n56, n57, n58, n59, n60; std_logic;
signal n51, n52, n53, n54, n55, n56, n57, n58, n59, n60; std_logic;
signal n71, n72, n73, n74, n75, n76, n77, n78, n79, n80; std_logic;
signal n81, n82, n83, n84, n85, g_temp; std_logic;
begin
  n1 <= not n2 after 0.3 ns;
  n2 <= not n3 after 0.3 ns;
  n3 <= not n4 after 0.3 ns;
  n4 <= not n5 after 0.3 ns;
  n5 <= not n6 after 0.3 ns;
  n6 <= not n7 after 0.3 ns;
  n7 <= not n8 after 0.3 ns;
  n8 <= not n9 after 0.3 ns;
  n9 <= not n10 after 0.3 ns;
  n10 <= stalu4_a end

library mgcPortable;
use mgcPortable.gin_te.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;
library my_packages;
use my_packages.package_1.all;
use my_packages.stackall;
library df_comp;
use df_comp.direg32.all;
use df_comp.direg32.all;
use df_comp.direg32.all;
use df_comp.direg32.all;
use df_comp.direg32.all;
use df_comp.direg32.all;
use df_comp.direg32.all;
use df_comp.direg32.all;
use df_comp.direg32.all;
library st_comp;
use st_comp.stregbank.all;
use st_comp.stalibk.all;

entity stalu is
  port (alu_start: in bit;
        lho: in bit;
        alu_select: in bit;
        md_select: in bit;
        add8_select: in bit;
        reg1_out: out bit;
        reg2_out: out bit;
        treg_out: out bit;
        pol: in bit;
        inst_lsa: in bit;
        wb_data: in bit;
        wb_sel: in bit;
        vct: in bit;
        write: in bit;
        jir: in bit;
        set_hi_db: in bit;
        set_lo_db: in bit;
        co: out bit;
        hi_db: out bit;
        lo_db: out bit;
        db: out bit;
        reg: out bit;
        alu_exe: out bit;
        inst_out: out bit;
        data2_out: out bit;
        cod: out bit;
        r1_test: out bit;
        r2_test: out bit;
        r3_test: out bit;
        r4_test: out bit;
        r5_test: out bit;
        hi_test: out bit;
        lo_test: out bit);
end stalu;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;
library my_packages;
use my_packages.package_1.all;
use my_packages.stpack.all;

library df_comp;
use df_comp.dfbusctl.all;
use df_comp.dfalu BaseController.all;
use df_comp.dfalu_ctrl.all;
use df_comp.dfaluDec.all;
use df_comp.dfaluShift.all;
use df_comp.dfaluStall.all;
use df_comp.dfortune.all;

library at_comp;
use at_comp.atalu32.all;

entity stalublk is
port(a_in: in std_logic_vector(31 downto 0); b_in: in std_logic_vector(31 downto 0); start: in std_logic; inst_in: in std_logic_vector(31 downto 0); load_hi, load_lo: out std_logic_vector(31 downto 0); set_hi_db, set_lo_db: out std_logic_vector(31 downto 0); inst_out: out std_logic_vector(31 downto 0); alub_done: out std_logic_vector(31 downto 0))
end entity;

architecture stalublk_a of stalublk is

component dfbusctl
port(i: in std_logic_vector(31 downto 0); a_in: in std_logic_vector(31 downto 0); b_in: in std_logic_vector(31 downto 0); inst_in: in std_logic_vector(31 downto 0); load_hi, load_lo, inst_out: out std_logic_vector(31 downto 0); alub_done: out std_logic_vector(31 downto 0))
end component;

component dfalu BaseController
port(i: in std_logic_vector(31 downto 0); inst_in: in std_logic_vector(31 downto 0); load_hi, load_lo: out std_logic_vector(31 downto 0); start: in std_logic; inst_out: out std_logic_vector(31 downto 0); alub_done: out std_logic_vector(31 downto 0))
end component;

component dfaluDec
port(i: in std_logic_vector(31 downto 0); inst_in: in std_logic_vector(31 downto 0); load_hi, load_lo: out std_logic_vector(31 downto 0); start: in std_logic; inst_out: out std_logic_vector(31 downto 0); alub_done: out std_logic_vector(31 downto 0))
end component;

component dfaluShift
port(i: in std_logic_vector(31 downto 0); inst_in: in std_logic_vector(31 downto 0); load_hi, load_lo: out std_logic_vector(31 downto 0); start: in std_logic; inst_out: out std_logic_vector(31 downto 0); alub_done: out std_logic_vector(31 downto 0))
end component;

component dfaluStall
port(i: in std_logic_vector(31 downto 0); inst_in: in std_logic_vector(31 downto 0); load_hi, load_lo: out std_logic_vector(31 downto 0); start: in std_logic; inst_out: out std_logic_vector(31 downto 0); alub_done: out std_logic_vector(31 downto 0))
end component;

component dfaluStall2
port(i: in std_logic_vector(31 downto 0); inst_in: in std_logic_vector(31 downto 0); load_hi, load_lo: out std_logic_vector(31 downto 0); start: in std_logic; inst_out: out std_logic_vector(31 downto 0); alub_done: out std_logic_vector(31 downto 0))
end component;

component dfaluStall3
port(i: in std_logic_vector(31 downto 0); inst_in: in std_logic_vector(31 downto 0); load_hi, load_lo: out std_logic_vector(31 downto 0); start: in std_logic; inst_out: out std_logic_vector(31 downto 0); alub_done: out std_logic_vector(31 downto 0))
end component;
end component;

component dfoutsel
port (inst: in bit_32;
a: in bit_5;
lr: in bit;
l: in bit;
sl: in bit_5;
o: out bit_32);
end component;

component dfshift
port (i: in bit_32;
lr: in bit;
l: in bit;
sl: in bit;
se: in bit_5;
o: out bit_32);
end component;

component dfslt
port (i: in bit_32;
inst: in bit_32;
lts: in bit;
o: out bit_32);
end component;

component dfoutsel
port (inst: in bit_32;
alu: in bit_32;
add$: in bit_32;
hi$: in bit_32;
output: out bit_32);
end component;

signal instl, a_line, b_line, alu_out, comp_out: bit_32;
signal shift_out, slt_out: bit_32;
signal a: bit_5;
signal s0, sl, s2, s3, c_out, lts, eqz, gtz, lr, la:
bit;
signal sel: bit_5;
signal Ca_line, Ch_line, Calu_out: slv_32;
signal Ca0, Ca1, Ca2, Cs3, Ccout: std_logic;
begins

bus_control: dfbusctl
port map (inst_in, a_in, b_in, pol_in, start,
lat_inst, ibo, instl, a_line, b_line,
a, b_out, pol_out, alu_done);
in

inst_out <= instl;
alu_decode: dfaludecode
port map (instl, s0, sl, s2, s3);
Ca_line <= To_StdLogicVector (a_line);
Ch_line <= To_StdLogicVector (b_line);
Ca0 <= To_StdLogic (s0);
Ca1 <= To_StdLogic (sl);
Ca2 <= To_StdLogic (s2);
Ca3 <= To_StdLogic (s3);
c_out <= To_bit (Cout);
alu_out <= To_bitvector (Calu_out);
alu: stalu32
port map (Ca_line, Ch_line, Ca0, Ca1,
Ca2, Cs3, Cout, Calu_out);

orf: dforrf
port map (instl, alu_out, c_out, exo);

compare: dfcomp
port map (alu_out, lts, eqz, gtz, comp_out);

branchCtl: dfb01
port map (instl, lts, eqz, gts, cc);

shiftCtl: dfactl
port map (instl, a, lr, la, sel);

shifter: dfshift
port map (comp_out, lr, la, sel, shift_out);

slt_box: dfslt
port map (shift_out, instl, lts, slt_out);

output_selector: dfoutsel
port map (instl, slt_out, add$, hilo, output);
end stalu38_a;

component dfmem
port (mem_start: in bit;
mem_ly: in bit;
mem_bus_req: out bit;
mem_load_addr: in bit_32;
mem_data: out bus_bit_32;
mem_done: out bit;
mem_ack: out bit);
end component;

component dfwb
port (wb_start: in bit;
w_bus_valid: out bit;
wb_data: in bit_32;
wb_start: in bit;
w_bus_ack: out bit;
w_data: out bit_32;
w_sel: out bit_6;
w_data: out bit_32);
end component;

component dfsb
port (illeg: in bit;
alu_exc: in bit;
mem_exc: in bit;
alu_start, mem_start: out bit;
alu_ack, mem_ack, wb_ack: in bit;
alu_done, mem_done: out bit);
end component;

component dfreg2
port (d: in bit_32;
d_start: out bit);
end component;

-- system control
signal start: bit;
signal begin_in: bit;

-- pipeline handshake
signal begin_ok, if_ok, id_ok, alu_ok, mem_ok,
wb_ok: bit := '1';
signal if_sack, id_sack, alu_sack, mem_sack, wb_sack, end_sack: bit;
signal if_start, id_start, alu_start, mem_start,
wb_start: bit;
signal if_done, id_done, alu_done, mem_done,
wb_done: bit;
signal init: bit;

-- interrupt/exception control
signal if_exc, id_exc, alu_exc, mem_exc: bit;
signal illegal, int_reg: bit;
signal int_vector: bit_32;
signal intLatch: bit;
constant zero_constant: bit := '0';
signal int_reset: bit := zero_constant;

-- bus control
signal if_bus_req, mem_bus_req: bit;
signal if_bus_ack, mem_bus_ack: bit;
signal if_load_addr, mem_load_addr: bit;
architecture stdece32 is
begin
  o6 <= '1' after 0.3 ns when i = '0000' else '0';
o1 <= '1' after 0.3 ns when i = '0001' else '0';
o2 <= '1' after 0.3 ns when i = '0010' else '0';
o3 <= '1' after 0.3 ns when i = '0011' else '0';
o4 <= '1' after 0.3 ns when i = '0100' else '0';
o5 <= '1' after 0.3 ns when i = '0101' else '0';
o6 <= '1' after 0.3 ns when i = '0110' else '0';
o7 <= '1' after 0.3 ns when i = '0111' else '0';
o8 <= '1' after 0.3 ns when i = '1000' else '0';
o9 <= '1' after 0.3 ns when i = '1001' else '0';
o10 <= '1' after 0.3 ns when i = '1010' else '0';
o11 <= '1' after 0.3 ns when i = '1011' else '0';
o12 <= '1' after 0.3 ns when i = '1100' else '0';
o13 <= '1' after 0.3 ns when i = '1101' else '0';
o14 <= '1' after 0.3 ns when i = '1110' else '0';
o15 <= '1' after 0.3 ns when i = '1111' else '0';
o16 <= '1' after 0.3 ns when i = '10101' else '0';
o17 <= '1' after 0.3 ns when i = '10110' else '0';
o18 <= '1' after 0.3 ns when i = '10111' else '0';
o19 <= '1' after 0.3 ns when i = '11000' else '0';
o20 <= '1' after 0.3 ns when i = '11001' else '0';
o21 <= '1' after 0.3 ns when i = '11010' else '0';
o22 <= '1' after 0.3 ns when i = '11011' else '0';
o23 <= '1' after 0.3 ns when i = '11100' else '0';
o24 <= '1' after 0.3 ns when i = '11101' else '0';
o25 <= '1' after 0.3 ns when i = '11110' else '0';
o26 <= '1' after 0.3 ns when i = '11111' else '0';
o27 <= '1' after 0.3 ns when i = '10110' else '0';
o28 <= '1' after 0.3 ns when i = '10111' else '0';
o29 <= '1' after 0.3 ns when i = '11010' else '0';
o30 <= '1' after 0.3 ns when i = '11011' else '0';
o31 <= '1' after 0.3 ns when i = '11100' else '0';
end stdece32;

architecture stdece32_a of stdece32 is
begin
  o0 <= '1' after 0.3 ns when i = '00000' and e'event and e = '1' else '0';
o1 <= '1' after 0.3 ns when i = '00001' and e'event and e = '1' else '0';
o2 <= '1' after 0.3 ns when i = '00010' and e'event and e = '1' else '0';
o3 <= '1' after 0.3 ns when i = '00011' and e'event and e = '1' else '0';
o4 <= '1' after 0.3 ns when i = '00110' and e'event and e = '1' else '0';
o5 <= '1' after 0.3 ns when i = '00111' and e'event and e = '1' else '0';
o6 <= '1' after 0.3 ns when i = '01110' and e'event and e = '1' else '0';
o7 <= '1' after 0.3 ns when i = '01111' and e'event and e = '1' else '0';
o8 <= '1' after 0.3 ns when i = '10100' and e'event and e = '1' else '0';
o9 <= '1' after 0.3 ns when i = '10101' and e'event and e = '1' else '0';
o10 <= '1' after 0.3 ns when i = '10110' and e'event and e = '1' else '0';
o11 <= '1' after 0.3 ns when i = '10111' and e'event and e = '1' else '0';
o12 <= '1' after 0.3 ns when i = '11000' and e'event and e = '1' else '0';
o13 <= '1' after 0.3 ns when i = '11001' and e'event and e = '1' else '0';
o14 <= '1' after 0.3 ns when i = '11010' and e'event and e = '1' else '0';
o15 <= '1' after 0.3 ns when i = '11011' and e'event and e = '1' else '0';
o16 <= '1' after 0.3 ns when i = '11100' and e'event and e = '1' else '0';
o17 <= '1' after 0.3 ns when i = '11101' and e'event and e = '1' else '0';
o18 <= '1' after 0.3 ns when i = '11110' and e'event and e = '1' else '0';
o19 <= '1' after 0.3 ns when i = '11111' and e'event and e = '1' else '0';
end stdece32_a;

architecture stdece32_b of stdece32 is
begin
  o0 <= '1' after 0.3 ns when i = '00000' and e'event and e = '1' else '0';
o1 <= '1' after 0.3 ns when i = '00001' and e'event and e = '1' else '0';
o2 <= '1' after 0.3 ns when i = '00010' and e'event and e = '1' else '0';
o3 <= '1' after 0.3 ns when i = '00011' and e'event and e = '1' else '0';
o4 <= '1' after 0.3 ns when i = '00110' and e'event and e = '1' else '0';
o5 <= '1' after 0.3 ns when i = '00111' and e'event and e = '1' else '0';
o6 <= '1' after 0.3 ns when i = '01110' and e'event and e = '1' else '0';
o7 <= '1' after 0.3 ns when i = '01111' and e'event and e = '1' else '0';
o8 <= '1' after 0.3 ns when i = '10100' and e'event and e = '1' else '0';
o9 <= '1' after 0.3 ns when i = '10101' and e'event and e = '1' else '0';
o10 <= '1' after 0.3 ns when i = '10110' and e'event and e = '1' else '0';
o11 <= '1' after 0.3 ns when i = '10111' and e'event and e = '1' else '0';
o12 <= '1' after 0.3 ns when i = '11000' and e'event and e = '1' else '0';
o13 <= '1' after 0.3 ns when i = '11001' and e'event and e = '1' else '0';
o14 <= '1' after 0.3 ns when i = '11010' and e'event and e = '1' else '0';
o15 <= '1' after 0.3 ns when i = '11011' and e'event and e = '1' else '0';
o16 <= '1' after 0.3 ns when i = '11100' and e'event and e = '1' else '0';
o17 <= '1' after 0.3 ns when i = '11101' and e'event and e = '1' else '0';
o18 <= '1' after 0.3 ns when i = '11110' and e'event and e = '1' else '0';
o19 <= '1' after 0.3 ns when i = '11111' and e'event and e = '1' else '0';
end stdece32_b;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164.all;
library my_packages;
use my_packages.all;
library my_packages;
use my_packages.all;
library my_packages;
use my_packages.all;
library st_comp;
use st_comp.all;
library st_comp;
use st_comp. package.all;
library st_comp;
use st_comp. package. all;
library st_comp;
use st_comp. package. all;
library st_comp;
use st_comp. package. all;
library st_comp;
use st_comp. package. all;
library st_comp;
use st_comp. package. all;
library st_comp;
use st_comp. package. all;
library st_comp;
use st_comp. package. all;

entity streg32 is
port(data: in std_logic_32;
  ld: in std_logic;
  as: in std_logic;
  bs: in std_logic;
  db: in std_logic;
  db_set: in std_logic;
  a: out std_logic;
  b: out std_logic;
  p: out std_logic);
end streg32;

architecture streg32_a of streg32 is
component streg8
port(port(i: in std_logic;
  ld: in std_logic;
  as: in std_logic;
  bs: in std_logic;
  db: in std_logic;
  en: in std_logic;
  db_set: in std_logic:
  a: out std_logic;
  b: out std_logic;
  p: out std_logic));
end component;

signal d1, d2, d3, d4, d5: std_logic;
signal ld1, ld2, ld3: std_logic;
signal al, a2, a3: std_logic;
signal bl, b1, b2, b3: std_logic;
signal dbid, dbwb: std_logic := '0';
signal pl, p1, p2, p3: std_logic;
begin
  ld0 <= data(7 downto 8);
  ld1 <= data(15 downto 8);
  ld2 <= data(23 downto 16);
  ld3 <= data(31 downto 24);
  ld0 <= vbt(0) and ld after 1 ns;
  ld1 <= vbt(1) and ld after 1 ns;
  ld2 <= vbt(2) and ld after 1 ns;
  ld3 <= vbt(3) and ld after 1 ns;
  reg_bytex: streg8
  port(map(d0, ld0, as, bs, a0, b0, p0));
  reg_bytex: streg8
  port(map(d1, ld1, as, bs, a1, b1, p1));
  reg_bytex: streg8
  port(map(d2, ld2, as, bs, a2, b2, p2));
  reg_bytex: streg8
  port(map(d3, ld3, as, bs, a3, b3, p3));
  a <= a0 & a2 & a1 & a3;
  b <= b0 & b2 & b1 & b3;
  p <= p0 & p2 & p1 & p3;
  dbid <= dbid xor '1' after 1 ns when db_set'event and db_set = '1'
  else dbid;
  dbwb <= dbwb xor '1' after 1 ns when ld'event and ld = '1'
  else dbwb;
  db <= dbid xor dbwb after 1 ns;
end streg32_a;
end stregbank;

architecture stregbank_a of stregbank is

component streg32

port(data: in slv_32;
ls: in std_logic;
ms: in std_logic;
vs: in std_logic;
reg: in slv_4;
db_e: in std_logic;
reg: out slv_32;
pi: out slv_32);
end component;

component stdec32

port(i: in slv_5;
pl: out std_logic;
sli: out std_logic;
o: out std_logic;
s2: out std_logic;
sl3: out std_logic;
s4: out std_logic;
s5: out std_logic;
s6: out std_logic;
s7: out std_logic;
s8: out std_logic;
s9: out std_logic;
s10: out std_logic;
s11: out std_logic;
s12: out std_logic;
s13: out std_logic;
s14: out std_logic;
s15: out std_logic;
s16: out std_logic;
s17: out std_logic;
s18: out std_logic;
s19: out std_logic;
s20: out std_logic;
s21: out std_logic;
s22: out std_logic;
s23: out std_logic;
s24: out std_logic;
s25: out std_logic;
s26: out std_logic;
s27: out std_logic;
s28: out std_logic;
s29: out std_logic;
s30: out std_logic;
s31: out std_logic);
end component;

component stdec32_1

port(i: in slv_5;
pl: out std_logic;
sli: out std_logic;
o: out std_logic;
s2: out std_logic;
sl3: out std_logic;
s4: out std_logic;
s5: out std_logic;
s6: out std_logic;
s7: out std_logic;
s8: out std_logic;
s9: out std_logic;
s10: out std_logic;
s11: out std_logic;
s12: out std_logic;
s13: out std_logic;
s14: out std_logic;
s15: out std_logic;
s16: out std_logic;
s17: out std_logic;
s18: out std_logic;
s19: out std_logic;
s20: out std_logic;
s21: out std_logic;
s22: out std_logic;
s23: out std_logic;
s24: out std_logic;
s25: out std_logic;
s26: out std_logic;
s27: out std_logic;
s28: out std_logic;
s29: out std_logic;
s30: out std_logic;
s31: out std_logic);
end component;

constant zero32: slv_32 := "0000000000000000000000000000000000000000000000000000000000000000";

signal zero32: slv_32 := zero32_constant;

signal id0, id1, id2, id3, id4, id5, id6: std_logic;
signal id0, id9, id10, id11, id12, id13, id14, id15: std_logic;
signal id14, id19, id18, id19, id20, id21, id22, id23: std_logic;
signal as0, as1, as2, as3, as4, as5, as6, as7: std_logic;
signal as8, as9, as10, as11, as12, as13, as14, as15: std_logic;
signal as16, as17, as18, as19, as20, as21, as22, as23: std_logic;
signal as24, as25, as26, as27, as28, as29, as30, as31: std_logic;
signal b0, b1, b2, b3, b4, b5, b6, b7: std_logic;
signal b8, b9, b10, b11, b12, b13, b14, b15: std_logic;
signal b16, b17, b18, b19, b20, b21, b22, b23: std_logic;
signal b24, b25, b26, b27, b28, b29, b30, b31: std_logic;
signal dbs0, dbs1, dbs2, dbs3, dbs4, dbs5, dbs6, dbs7: std_logic;
signal db14, db15: std_logic;
signal db16, db17, db18, db19, db20, db21, db22, db23, db24, db25, db26, db27, db28, db29, db30, db31: std_logic;
signal p0, p1, p2, p3, p4, p5, p6, p7: slv_32;
signal p8, p9, p10, p11, p12, p13, p14, p15: slv_32;
signal p16, p17, p18, p19, p20, p21, p22, p23: slv_32;

begin
reg_sel_dec: stdce32
port map(reg_sel, write, id0, id1, id2, id3, id4, id5, id6, id7, id8, id9, id10, id11, id12, id13, id14, id15, id16, id17, id18, id19, id20, id21, id22, id23, id24, id25, id26, id27, id28, id29, id30, id31);

a_sel_dec: stdce32
port map(a_sel, as0, as1, as2, as3, as4, as5, as6, as7, as8, as9, as10, as11, as12, as13, as14, as15, as16, as17, as18, as19, as20, as21, as22, as23, as24, as25, as26, as27, as28, as29, as30, as31);

b_sel_dec: stdce32
port map(b_sel, bs0, bs1, bs2, bs3, bs4, bs5, bs6, bs7, bs8, bs9, bs10, bs11, bs12, bs13, bs14, bs15, bs16, bs17, bs18, bs19, bs20, bs21, bs22, bs23, bs24, bs25, bs26, bs27, bs28, bs29, bs30, bs31);

db_sel_dec: stdce32
port map(db_sel, db0, db1, db2, db3, db4, db5, db6, db7);

end.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;
library my_packages;
use my_packages.stpack.all;

entity sttg8 is
  port(i: in std_logic;
       o: out std_logic);
end sttg8;

architecture sttg8_a of sttg8 is
begin
  o <= i after 0.1 ns when en = '1' else
       "$22222222" after 0.1 ns;
end sttg8_a;
APPENDIX E - SUPPORT PROGRAMS

#include <stdio.h>
#include <math.h>

#define WHITE_SPACE/tab token == ' ' || \n
token == '\t'

#define COMMENT token == '#'

#define DIRECTIVE token == ':'

#define DATA token == EOT

#define CODE strcmp(directive, "code") == 0

#define NOT_DELIMITER =

#define END strcmp(directive, "end") == 0

#define NOP甾
tokendirective, "data") == 0

#define DATA_DATADirectivestrcmp(tokens, ".data") == 0

#define END DIRECTIVE strcmp(tokens, ".end") == 0

#define PAD5 000000
#define PAD10 0000000000
#define PAD15 00000000000000
#define PAD20 000000000000000000
#define NEW_LINE 

#define NOP 000000000000000000000000000000000
#define HALF "1111110000000000000000000000000000"

#define SPECIAL "000000"
#define SLL "000010"
#define SRL "000011"
#define SLV "000100"
#define SRV "000110"
#define JR "001000"
#define JALR "001011"
#define SYSCALL "001100"
#define BREAK "001110"
#define MWI "010000"
#define MSH "010001"
#define MFLO "010100"
#define MFLO "010101"
#define MUL "011000"
#define MULSU "011001"
#define DIV "011010"
#define DIVU "011100"
#define ADDI "011000"
#define ADDU "100001"
#define SUB "100010"
#define SHU "100111"
#define OR "101001"
#define XOR "101010"
#define NOR "101011"
#define SLF "101100"
#define SRLU "101110"

#define BCOND "$000001"
#define BLTZ "$000001"
#define BGEZ "$000001"
#define BGEZAL "$000001"

#define O "000010"
#define JAL "$000011"
#define B "000100"
#define BLEZ "$000110"
#define BLEZ "$000111"

#define ADD "$010000"
#define ADD "$010001"
#define SUB "$010010"
#define SUBU "$010011"
#define OR "$010101"
#define XOR "$010110"
#define NOR "$010111"
#define LB "$100000"
#define LH "$100010"
#define LW "$100110"
#define LBU "$100111"
#define LBU "$101000"
#define LBU "$101001"
#define LBU "$101010"
#define LBU "$101011"
#define SB "$101000"
#define SH "$101001"
#define SH "$101010"
#define SH "$101011"
#define SW "$101100"
#define SW "$101101"
#define SW "$101110"
#define SW "$101111"

eval boolean (FALSE, TRUE);
typePenn boolean boolean;

FILE * in;
FILE * out;

void itob((int intval, char * binary)
{
  int i;
  for (i = 0; i <= 4; i++)
    if (intval == pow(2, (i - 1)))
      binary[0] = '1';
    else
      binary[0] = '0';
}

void itob16((int intval, char * binary)
{
  int i;
  for (i = 0; i <= 15; i++)
    if (intval == pow(2, (i - 1)))
      binary[1] = '1';
    else
      binary[1] = '0';
}

void itob32((int intval, char * binary)
{
  int i;
  for (i = 0; i <= 31; i++)
    if (intval == pow(2, (i - 1)))
      binary[1] = '1';
}

void itob64((int intval, char * binary)
{
  int i;
  for (i = 0; i <= 63; i++)
    if (intval == pow(2, (i - 1)))
      binary[1] = '1';
}
intval = intval - pow(2, (25 - i));
} else {
    binary[i] = '0';
}
}
binary[26] = '\0';

int b32(int intval, char * binary)
{
    int i;
    for (i = 0; i <= 31 - i++;)
    {
        if (intval >= pow(2, (31 - 1)))
            {  
                binary[i] = '1';
                intval = intval - pow(2, (31 - i));
            }
        else
            {  
                binary[i] = '0';
            }
    }
    binary[32] = '\0';
}

void outf(char * arg)
{
    fprintf(out, "%s", arg);
}

void in5(char * binary)
{
    /* char tokens[10];
    int arg;
    get_token(tokens);
    printf("%s ", tokens);
    if (tokens[0] == '$')
        {  
            sscanf(tokens + 1, "%d", &arg);
        }
    itob5(arg, binary);
    int arg;
    fscanf(in, "%d", &arg);
    printf("%d ", arg);
    itob5(arg, binary);
    }
}

void in6(char * binary)
{
    int arg;
    fscanf(in, "%d", &arg);
    printf("%d ", arg);
    itob6(arg, binary);
}

void ln26(char * binary)
{
    int arg;
    fscanf(in, "%x", &arg);
    printf("%x ", arg);
    itob26(arg, binary);
}

void ln5(char * binary)
{
    int arg;
    fscanf(in, "%d", &arg);
    printf("%d ", arg);
    itob5(arg, binary);
    int index = 0;
    boolean done = FALSE;
    While (NOT_EOF && done == FALSE)
        {  
            token = fgetc(in);
            If (WHITE_SPACE_CHAR & NOT_EOF)
                {  
                    tokens[index] = token;
                    token = fgetc(in);
                }
            else
                {  
                    if (COMMENT)
                        {  
                            while (NOT_DELIMITER & NOT_EOF)
                                {  
                                    tokens[index] = token;
                                    token = fgetc(in);
                                    index++;
                                }
                            tokens[index] = '\0';
                            done = TRUE;
                        }
                    else
                        {  
                            if (NOT_EOL)  
                                {  
                                    while (NOT_DELIMITER & NOT_EOL)
                                        {  
                                            token = fgetc(in);
                                            index++;
                                        }
                                    tokens[index] = '\0';
                                    done = TRUE;
                                }
                        }
        }
    printf("%d data lines found", data_count);
    printf("end parse data segment
")
    return data_count;
}

void ln6(char * binary)
{
    int arg;
    fscanf(in, "%x", &arg);
    printf("%x ", arg);
    itobl6(arg, binary);
}

int pass_one( void )
{
    printf("PASS ONE...
");
    return 1;
}

void get_filename(char * name, char * mname)
{
    char temp[20];
    printf("ENTER FILENAME <== ");
    scanf("%s", temp);
    printf("\n\n");
    sprintf(name, "%s.test", temp);
    sprintf(mname, "%s.m", temp);
    int get_token( char * tokens )
    {
        char token = ' ';  
        int index = 0;
        boolean done = FALSE;
        int eot = 0;
        While (NOT_EOL & done == FALSE)
            {  
                token = fgetc(in);
                if (NOT_EOL & done == FALSE)
                    {  
                        token = fgetc(in);
                    }
                else
                    {  
                        if (NOT_EOL)  
                            {  
                                while (NOT_DELIMITER & NOT_EOL)
                                    {  
                                        tokens[index] = token;
                                        index++;
                                    }
                                tokens[index] = '\0';
                                done = TRUE;
                            }
                    }
    }
    printf("%s", tokens);"
    return (int)token;
}

int message( void )
{
    printf("\n\n");
    printf("# MIPS ASSEMBLER, VERSION 1.0 \#\n");
    printf("# WRITTEN BY PAUL PANELLI, 1993 \#\n");
    printf("\n\n");
}

void parse_data_seg_pi(void)
{
    char tokens[40];
    boolean done = FALSE;
    int data_count = 0;
    int eof = 0;
    printf("parse data segment
");
    while ( done == FALSE &.eof == EOF)
        {  
            token = get_token(tokens);
            if (END_DIRECTIVE)
                {  
                    done = TRUE;
                }
            else
                {  
                    data_count++;
                }
    printf("%d data lines found", data_count);
    printf("end parse data segment
")
    return data_count;
}

void itob5(int intval, char * binary)
{
    intval = intval - (pow(2, (25 - i)));
    if (intval >= (pow(2, (31 - 1))))
        {  
            binary[i] = '1';
            intval = intval - (pow(2, (31 - i)));
        }
    else
        {  
            binary[i] = '0';
        }
    binary[26] = '\0';
}

void ln46(char * binary)
{
    int arg;
    fscanf(in, "%x", &arg);
    printf("%x ", arg);
    itob26(arg, binary);
}

void ln67(char * binary)
{
    int arg;
    fscanf(in, "%d", &arg);
    printf("%d ", arg);
    itob6(arg, binary);
}

void message(void)
{
    printf("\n\n");
    printf("# MIPS ASSEMBLER, VERSION 1.0 \#\n");
    printf("# WRITTEN BY PAUL PANELLI, 1993 \#\n");
    printf("\n\n");
}

int pass_one( void )
{
    char tokens[40];
    char directive[10];
    int data_count = 0;
    boolean done = FALSE;
    int eof = 0;
    printf("PASS ONE...\n");
}
while(done == FALSE && eof == EOF)
{
    eof = get_token(tokens);
    if (DATA_DIRECTIVE)
    {
        printf("found data segment\n");
        data_count = parse_data_seg_p1();
        printf("\n\n");
        data_count = 0;
    }
    if (data_count == 0)
    {
        printf("no data segment found\n");
    }
    return data_count;
}

void
parse_data_seg_p2()
{
    char tokens[40];
    boolean done = FALSE;
    int dat;
    char binary[33];
    int eof = 0;
    printf("parse data segment\n");
    while (done == FALSE && eof == EOF)
    {
        eof = get_token(tokens);
        if (END_DIRECTIVE)
        {
            done = TRUE;
        }
        else
        {
            sscanf(tokens, "\%x", &dat);
            itoh32(dat, binary);
            outf(binary);
            printf("\n\n");
            outf(NEXTLINE);
            printf("\n\n");
            outf(NEXTLINE);
        }
    }
    printf("end parse data segment\n");
}

void
code(void)
{
    char op[7];
    char rs[6];
    char rt[6];
    char rd[6];
    char shamt[6];
    char base[6];
    char offset[17];
    char immed[17];
    char target[27];
    char token;

    while (get_token(op) == EOF)
    {
        /*
        */
        if (strcmp(op, "nop") == 0)
        {
            outf(MOP);
            outf(NEXTLINE);
        }
        /* sll rd, rt, shamt */
        else if (strcmp(op, "sll") == 0)
        {
            outf(SPECIAL);
            */
            outf(PAD5);
            */
            outf(shamt);
            outf(rt);
            */
            outf(rd);
            outf(NEXTLINE);
        }
        /* sra rd, rt, shamt */
        else if (strcmp(op, "sra") == 0)
        {
            outf(SPECIAL);
            */
            outf(PAD5);
            */
            outf(rd);
            outf(SRA);
            outf(NEXTLINE);
        }
        /* srlv rd, rt, rs */
        else if (strcmp(op, "srlv") == 0)
        {
            outf(SPECIAL);
            */
            outf(rd);
            outf(rs);
            outf(SRLV);
            outf(NEXTLINE);
        }
    }
}
/* jalr rd, ra */
else if (strcmp(op, "jalr") == 0)
{
    /* opcode special */
    outf(SPECIAL);
    /* get fields */
    in5(ra);
    /* write fields */
    outf(PAD15);
    outf(JR);
    outf(NEWLINE);
}

/* mthi rd */
else if (strcmp(op, "mthi") == 0)
{
    /* opcode special */
    outf(SPECIAL);
    /* get fields */
    in5(rd);
    /* write fields */
    outf(PAD5);
    outf(rd);
    outf(PAD5);
    outf(MTHI);
    outf(NEWLINE);
}

/* mfhi rd */
else if (strcmp(op, "mfhi") == 0)
{
    /* function */
    outf(SPECIAL);
    /* get fields */
    in5(rd);
    /* pad */
    outf(PAD5);
    outf(rd);
    outf(MFHI);
    outf(NEWLINE);
}

/* mflo rd */
else if (strcmp(op, "mflo") == 0)
{
    /* function */
    outf(SPECIAL);
    /* get fields */
    in5(rd);
    /* pad */
    outf(PAD5);
    outf(rd);
    outf(MFLO);
    outf(NEWLINE);
}

/* mllo rd */
else if (strcmp(op, "mllo") == 0)
{
    /* function */
    outf(SPECIAL);
    /* get fields */
    in5(rd);
    /* pad */
    outf(PAD5);
    outf(rd);
    outf(MFLO);
    outf(NEWLINE);
}

/* mult ra, rt */
else if (strcmp(op, "mult") == 0)
{
}
} /* subu rd, ra, rt */
else if (strcmp(op, "subu") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(ra);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PADS);
    outf(PHED);
    outf(NEWLINE);
}

/* and rd, ra, rt */
else if (strcmp(op, "and") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(ra);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PADS);
    outf(AND);
    outf(NEWLINE);
}

/* xor rd, ra, rt */
else if (strcmp(op, "xor") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(ra);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PADS);
    outf(XOR);
    outf(NEWLINE);
}

/* nor rd, ra, rt */
else if (strcmp(op, "nor") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(ra);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PADS);
    outf(NOR);
    outf(NEWLINE);
}

/* or rd, ra, rt */
else if (strcmp(op, "or") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(ra);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PADS);
    outf(OR);
    outf(NEWLINE);
}

/* sltu rd, ra, rt */
else if (strcmp(op, "sltu") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(ra);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PADS);
    outf(SLT);
    outf(NEWLINE);
}

/* beq rd, ra, offset */
else if (strcmp(op, "beq") == 0)
{
    outf(BCOND);
    in5(rd);
    in5(ra);
    outf(BEQ);
    in5(offset);
    outf(offset);
    outf(NEWLINE);
}

/* bgezal rd, ra, offset */
else if (strcmp(op, "bgezal") == 0)
{
    outf(BCOND);
    in5(rd);
    in5(ra);
    outf(BGEZAL);
    in5(offset);
    outf(offset);
    outf(NEWLINE);
}

/* bltzal rd, ra, offset */
else if (strcmp(op, "bltzal") == 0)
{
    outf(BCOND);
    in5(rd);
    in5(ra);
    outf(BLZTAL);
    in5(offset);
    outf(offset);
    outf(NEWLINE);
}

/* bgez rd, ra, offset */
else if (strcmp(op, "bgez") == 0)
{
    outf(BCOND);
    in5(rd);
    in5(ra);
    outf(BGEZ);
    in5(offset);
    outf(offset);
    outf(NEWLINE);
}

/* bltz rd, ra, offset */
else if (strcmp(op, "bltz") == 0)
{
    outf(BCOND);
    in5(rd);
    in5(ra);
    outf(BLZT);
    in5(offset);
    outf(offset);
    outf(NEWLINE);
}

/* jal target */
else if (strcmp(op, "jal") == 0)
{
    outf(JAL);
    in26(target);
    outf(target);
    outf(NEWLINE);
}

/* j target */
else if (strcmp(op, "j") == 0)
{
    outf(J);
    in26(target);
    outf(target);
    outf(NEWLINE);
}

/* beqz rd, ra, offset */
else if (strcmp(op, "beqz") == 0)
{
    outf(BEQZ);
    in5(rd);
    in5(ra);
    in5(rt);
    in5(offset);
    outf(rs);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}
/* bne rs, rt, offset */
else if (strcmp(op, "bne") == 0)
    {
        outf(INM);
        in5(rs);
        in5(rt);
        in6(offset);
        outf(rs);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

/* blez rs, offset */
else if (strcmp(op, "blez") == 0)
    {
        outf(BBLEZ);
        in5(rs);
        in6(offset);
        outf(rs);
        outf(PAD5);
        outf(offset);
        outf(NEWLINE);
    }

/* bgtz rs, offset */
else if (strcmp(op, "bgtz") == 0)
    {
        outf(BGTZ);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* blez rs, offset */
else if (strcmp(op, "blez") == 0)
    {
        outf(BBLEZ);
        in5(rs);
        in6(offset);
        outf(rs);
        outf(PAD5);
        outf(offset);
        outf(NEWLINE);
    }

/* bgtz rs, offset */
else if (strcmp(op, "bgtz") == 0)
    {
        outf(BGTZ);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* addi rt, rs, immediate */
else if (strcmp(op, "addi") == 0)
    {
        outf(ADDI);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* addiu rt, rs, immediate */
else if (strcmp(op, "addiu") == 0)
    {
        outf(ADDIU);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* slti rt, rs, immediate */
else if (strcmp(op, "slti") == 0)
    {
        outf(SLTI);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* sltiu rt, rs, immediate */
else if (strcmp(op, "sltiu") == 0)
    {
        outf(SLTIU);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* andi rt, rs, immediate */
else if (strcmp(op, "andi") == 0)
    {
        outf(ANDI);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* orl rt, rs, immediate */
else if (strcmp(op, "orl") == 0)
    {
        outf(ORL);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* xorl rt, rs, immediate */
else if (strcmp(op, "xorl") == 0)
    {
        outf(XORL);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* lb rt, offset(base) */
else if (strcmp(op, "lb") == 0)
    {
        outf(LB);
        in5(rt);
        in6(offset);
        in5(base);
        outf(base);
        outf(offset);
        outf(NEWLINE);
    }

/* lwl rt, offset(base) */
else if (strcmp(op, "lwl") == 0)
    {
        outf(LWL);
        in5(rt);
        in6(offset);
        in5(base);
        outf(base);
        outf(offset);
        outf(NEWLINE);
    }

/* lw rt, offset(base) */
else if (strcmp(op, "lw") == 0)
    {
        outf(LW);
        in5(rt);
        in6(offset);
        in5(base);
        outf(base);
        outf(offset);
        outf(NEWLINE);
    }

/* lui rt, offset(base) */
else if (strcmp(op, "lui") == 0)
    {
        outf(LUI);
        in5(rt);
        in5(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* li rt, offset(base) */
else if (strcmp(op, "li") == 0)
    {
        outf(LI);
        in5(rt);
        in6(offset);
        in5(base);
        outf(base);
        outf(offset);
        outf(NEWLINE);
    }

/* add rt, rs, immediate */
else if (strcmp(op, "add") == 0)
    {
        outf(ADD);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* addi rt, rs, immediate */
else if (strcmp(op, "addi") == 0)
    {
        outf(ADDI);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* or rt, rs, immediate */
else if (strcmp(op, "or") == 0)
    {
        outf(OR);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* xor rt, rs, immediate */
else if (strcmp(op, "xor") == 0)
    {
        outf(XOR);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* orl rt, rs, immediate */
else if (strcmp(op, "orl") == 0)
    {
        outf(ORL);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* xorl rt, rs, immediate */
else if (strcmp(op, "xorl") == 0)
    {
        outf(XORL);
        in5(rt);
        in5(rs);
        in6(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* lb rt, offset(base) */
else if (strcmp(op, "lb") == 0)
    {
        outf(LB);
        in5(rt);
        in6(offset);
        in5(base);
        outf(base);
        outf(offset);
        outf(NEWLINE);
    }

/* lwl rt, offset(base) */
else if (strcmp(op, "lwl") == 0)
    {
        outf(LWL);
        in5(rt);
        in6(offset);
        in5(base);
        outf(base);
        outf(offset);
        outf(NEWLINE);
    }

/* lw rt, offset(base) */
else if (strcmp(op, "lw") == 0)
    {
        outf(LW);
        in5(rt);
        in6(offset);
        in5(base);
        outf(base);
        outf(offset);
        outf(NEWLINE);
    }

/* lui rt, offset(base) */
else if (strcmp(op, "lui") == 0)
    {
        outf(LUI);
        in5(rt);
        in5(immed);
        outf(rs);
        outf(rt);
        outf(lammed);
        outf(NEWLINE);
    }

/* li rt, offset(base) */
else if (strcmp(op, "li") == 0)
    {
        outf(LI);
        in5(rt);
        in6(offset);
        in5(base);
        outf(base);
        outf(offset);
        outf(NEWLINE);
    }
/* lw rt. offset(base) */
else if (strcmp(op, "lw") == 0)
{
    outf(LW);
    in5(rt);
    in6(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* lb rt. offset(base) */
else if (strcmp(op, "lb") == 0)
{
    outf(LB);
    in5(rt);
    in6(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* lbu rt. offset(base) */
else if (strcmp(op, "lbu") == 0)
{
    outf(LBU);
    in5(rt);
    in6(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* lwr rt. offset(base) */
else if (strcmp(op, "lwr") == 0)
{
    outf(LWR);
    in5(rt);
    in6(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* sb rt. offset(base) */
else if (strcmp(op, "sb") == 0)
{
    outf(SB);
    in5(rt);
    in6(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* sh rt. offset(base) */
else if (strcmp(op, "sh") == 0)
{
    outf(SH);
    in5(rt);
    in6(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* sw rt. offset(base) */
else if (strcmp(op, "sw") == 0)
{
    outf(SW);
    in5(rt);
    in6(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* swr rt. offset(base) */
else if (strcmp(op, "swr") == 0)
{
    outf(SWR);
    in5(rt);
    in6(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* halt instruction */
else if (strcmp(op, "halt") == 0)
{
    outf(HALT);
    outf(NEWLINE);
}
else
{
    printf("\nERROR: UNKNOWN COMMAND\n");
}

printf("\n\n");

/* END WHILE FSCANF \n EOF */

void
pass_two(int data_count)
{
    char binary[33];
    char tokens[40];
    boolean done = FALSE;
    int eof = 0;

    printf("PASS TWO...\n");
    if (data_count == 0)
    {
        /* ADD JUMP INSTRUCTION TO JUMP OVER DATA SEGMENT */
        outf(J);
        itoh2((data_count + 2, binary);
        outf(binary);
        outf(NEWLINE);

        /* ADD NOP TO FILL DELAY SLOT */
        outf(NOP);
        outf(NEWLINE);

        /* GET DATA SEGMENT */
        while (done == FALSE && eof == EOF)
        {
            eof = get_token(tokens);
            if (DATA_DIRECTIVE)
            {
                parse_data_seg_p2();
                done = TRUE;
            }
        }
    }

    code();
}

int
main(void)
{
    char name[20];
    char name2[20];
    int data_count;
    message();
    get_filename(name, name2);

    in = fopen(name, "r");
    data_count = pass_one();
    fclose(in);

    in = fopen(name, "r");
    out = fopen(name2, "w");
    pass_two(data_count);
    fclose(in);
    fclose(out);

    printf("CONVERSION DONE\n");

}
MERA-C - MIPS EXPECTED RESULTS ASSEMBLER

#include <stdio.h>
#include <math.h>

#define WHITE_SPACE_CEAR token == ' ' || \
    token == '
' || \
    token == '\t' || \
#define COMMENT token == '#'
#define NOT_HASH token == '#'
#define NOT_DELIMITER token == '#' && token != 'n'
#define NOT_NEWLINE token == 'n'

enum boolean (FALSE, TRUE);
typedef enum boolean boolean;

FILE * in = NULL;
FILE * out = NULL;
char assem[1000][80];
int expect[1000][10];
char modify[1000][10];
int num_of_inst = 0;
int index = 0;
boolean p_mode;

void
btoi32(int integer, char * bits)
{
    int 1;
    int temp;
    if (integer < 0)
    {
        temp = -(integer + 1);
    }
    else
    {
        temp = integer;
    }
    for (i = 0; i <= 31; i++)
    {
        if ((temp % 2) == 1)
        {
            bits[31-i] = '1';
        }
        else
        {
            bits[31-i] = '0';
        }
        temp = temp / 2;
    }
    if (integer < 0)
    {
        for (i = 0; i <= 31; i++)
        {
            if (bits[i] == '0')
            {
                bits[i] = '1';
            }
            else
            {
                bits[i] = '0';
            }
        }
        bits[32] = '0';
    }
}

int
power2(int x)
{
    if (x == 0) return 1;
    else return 2 * power2(x-1);
}

int
btoi32(char * bits)
{
    int 1;
    int result = 0;
    boolean negative;
    negative = (bits[0] == '1');
void view(void)
{
  int i;
  printf("***************\n");
  printf("\n");
}

void view(void)
{
  int i;
  printf("***************\n");
  printf("\n");
}

int command_menu(void)
{
  int ch;
  printf("Command menu\n");
  printf("\n");
  if (ch == 'R')
    ch = getchar();
  printf("\n");
  return ch;
}

int assemble_menu(void)
{
  int ch;
  printf("Assemble menu\n");
  printf("\n");
  if (ch == 'R')
  ch = 
  printf("\n");
  return ch;
}
```c
void build_modify_array(void)
{
    int j;
    int i;
    for (j = 0; j < num_of_inst; j++)
    {
        for (i = 0; i <= 9; i++)
        {
            if (j == 0)
                if (expect[0][i] == 0)
                    (void) modify[j][i] = ' ';
            else
                (modify[j][i] = 'x';
        }
    }
}

void change_pc(void)
{
    char input[33];
    int temp;
    printf("OLD PC VALUE (HEX) = \x\n",
        expect[index][0]);
    printf("NEW PC VALUE (HEX) - use (l) to increment by 4 \x\n",
        scanf("%s", input);
    printf("\n\n");
    if (input[0] == 'l')
        expect[index][0] = expect[index][0] + 4;
    else
```
void
{
  change_r3
  
  change_r4
  
  change_rl

  get_input
  
  int
  
  scanf("\%s", input);
  printf("\n");
  if (input[0] == 'h')
  {
    scanf(input+1, "\%s", &temp);
  }
  else
  {
    scanf(input, "\%d", &temp);
  }
  return temp;
}

void
change_rl(void)
{
  printf("OLD R1 VALUE = \%d\n", expect[index][1]);
  printf("NEW R1 VALUE = use (h) for hex \%s\n");
  expect[index][1] = get_input();
  view_state(1);
}

void
change_r2(void)
{
  int input;
  printf("OLD R2 VALUE = \%d\n", expect[index][2]);
  printf("NEW R2 VALUE = use (h) for hex \%s\n");
  expect[index][2] = get_input();
  view_state(2);
}

void
change_r3(void)
{
  int input;
  printf("OLD R3 VALUE = \%d\n", expect[index][3]);
  printf("NEW R3 VALUE = use (h) for hex \%s\n");
  expect[index][3] = get_input();
  view_state(3);
}

void
change_r4(void)
{
  int input;
  printf("OLD R4 VALUE = \%d\n", expect[index][4]);
  printf("NEW R4 VALUE = use (h) for hex \%s\n");
  expect[index][4] = get_input();
  view_state(4);
}

void
change_r31(void)
{
  int input;
  printf("OLD R31 VALUE = \%d\n", expect[index][5]);
  printf("NEW R31 VALUE = use (h) for hex \%s\n");
  expect[index][5] = get_input();
  view_state(5);
}

void
change_hi(void)
{
  int input;
  printf("OLD HI VALUE = \%d\n", expect[index][6]);
  printf("NEW HI VALUE = use (h) for hex \%s\n");
  expect[index][6] = get_input();
  view_state(6);
}

void
change_lo(void)
{
  int input;
  printf("OLD LO VALUE = \%d\n", expect[index][7]);
  printf("NEW LO VALUE = use (h) for hex \%s\n");
  expect[index][7] = get_input();
  view_state(7);
}

void
change_esc(void)
{
  int input;
  printf("OLD ESC VALUE (HEX) = \%x\n",
  expect[index][8]);
  printf("NEW ESC VALUE (HEX) \%s\n");
  scanf("\%x", &input);
  printf("\n");
  expect[index][8] = input;
  view_state(8);
}

boolean
query_new_state(void)
{
  int selector;
  boolean done = FALSE;
  boolean next = FALSE;
  int i;

  while (done == FALSE & next == FALSE)
  {
    selector = assemble_menu();
    switch (selector)
    {
      case 'p':
        change_pc();
        break;
      case 'l':
        change_rl();
        break;
      case '2':
        change_r2();
        break;
      case '3':
        change_r3();
        break;
      case '4':
        change_r4();
        break;
      case '5':
        change_r31();
        break;
      case 'h':
        change_hi();
        break;
      case 'l':
        change_lo();
        break;
      case 'e':
        change_esc();
        break;
      case 'c':
        change_cause();
        break;
      case 'b':
        index = index - 1;
        next = TRUE;
    }
  }
}
void assemble_new_f
  {num_of_inst = load_assem_flie();
   if (get_filename(name, ename))
     return;
   in = fopen(name, "r");
   fclose(in);
   initialize();
   view();
   }
AGAIN

boolean query_save(char * ename)
{
    int ch;
    printf("SAVE CHANGES TO file (y,n)? " , ename);
    ch = getchar();
    if (ch == 'y')
    {
        ch = getchar();
    }
    printf("\n");
    if (ch == 'y')
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }

    int main(void)
    {
        int selector;
        boolean done = FALSE;
        char name[20];
        char ename[20];
        boolean save;
        message();
        help_message();
        while (done == FALSE)
        {
            selector = command_menu();
            switch (selector)
            {
                case 'n':
                    index = 0;
                    assemble_new_file(name, ename);
                    save = TRUE;
                    break;
                case 'o':
                    index = 0;
                    change_old_file(name, ename);
                    save = TRUE;
                    break;
                case 'a':
                    assemble();
                    save = TRUE;
                    break;
                case 'c':
                    clear();
                    save = TRUE;
                    break;
                case 'v':
                    view();
                    break;
                case 'r':
                    save_file(ename);
                    save = FALSE;
                    break;
                case 'q':
                    if (save == TRUE)
                    {
                        if (query_save(ename) == TRUE)
                        {
                            save_file(ename);
                            save = FALSE;
                        }
                    }
                    done = quit();
                    break;
                case '?':
                    command_help();
                    break;
                default:
                    printf("ILLEGAL COMMAND - TRY AGAIN\n");
                    return 0;
            }
        }
    }

    void message(void)
    {
        printf("\n");
        printf("************\n");
        printf("************\n");
        printf("************\n");
        printf("************\n");
        printf("************\n");
    }

    void help_message(void)
    {
        printf(" *** USE THE (?) KEY FOR HELP AT THE MENU PROMPTS \n");
        printf(" *** USE THE (RETURN) KEY RETURN FROM MENU\n");
        printf("\n");
    }

    void command_help(void)
    {
        printf("*************** COMMAND MENU \n");
        printf("*************** COMMAND MENU \n");
        printf("*************** COMMAND MENU \n");
        printf("*************** COMMAND MENU \n");
        printf("*************** COMMAND MENU \n");
    }

    void assemble_help(void)
    {
        printf("*************** ASSEMBLE MENU \n");
        printf("*************** ASSEMBLE MENU \n");
        printf("*************** ASSEMBLE MENU \n");
        printf("*************** ASSEMBLE MENU \n");
        printf("*************** ASSEMBLE MENU \n");
    }

    #include "stdio.h"
    #define WHITE_SPACE_CHAR token == ' ' || \" \
    #define CENTER token == '\n'
    #define NEW_LINE token == '\n'
    #define NOT_DELIMITER token == '#' || token == '\n'
    #define NOT_EOF token == EOF
    enum boolean (FALSE, TRUE);
    typedef enum boolean boolean;
    FILE * in = NULL;
    FILE * out = NULL;
    FILE * out2 = NULL;
    char asname[1000][80];
    int order[1000];
    int index = 0;
    int start = 0;
    int finish = 0;
    void assemble(void)
    {
        printf("***************\n");
        printf("***************\n");
        printf("***************\n");
        printf("***************\n");
        printf("***************\n");
    }

    void message(void)
    {
        printf("\n");
        printf("\n");
        printf("\n");
        printf("\n");
        printf("\n");
    }

    void help_message(void)
    {
        printf(" *** USE THE (?) KEY FOR HELP AT THE MENU PROMPTS \n");
        printf(" *** USE THE (RETURN) KEY RETURN FROM MENU\n");
        printf("\n");
    }

    void command_help(void)
    {
        printf("*************** COMMAND MENU \n");
        printf("*************** COMMAND MENU \n");
        printf("*************** COMMAND MENU \n");
        printf("*************** COMMAND MENU \n");
        printf("*************** COMMAND MENU \n");
    }

    void assemble_help(void)
    {
        printf("*************** ASSEMBLE MENU \n");
        printf("*************** ASSEMBLE MENU \n");
        printf("*************** ASSEMBLE MENU \n");
        printf("*************** ASSEMBLE MENU \n");
        printf("*************** ASSEMBLE MENU \n");
    }
printf("(1)instruction - goto instruction\n")
printf("(2)view - view present contents\n")
printf("(3)state - view present state\n")
printf("(4)range - set view range\n")
printf("(5)help - print out this help listing\n")
printf("***************\n")
printf("***************\n");
printf("\n");

int command_menu(void)
{
  int ch;
  printf("COMMAND(0,0,a,v,q,t) == "");
  ch = getchar();
  if (ch == '\n')
    { ch = getchar();
    }
  printf("\n");
  return ch;
}

int assemble_menu(void)
{
  int ch;
  printf("ASSEMBLE(e,b,f,i,v,e,x,return,?) == "");
  ch = getchar();
  if (ch == '\n')
    { ch = getchar();
    }
  printf("\n");
  return ch;
}

void view(void)
{
  int i;
  printf("*************** ORIGINAL ***************\n");
  for (i=start; 1 <= finish; i++)
    { printf("%d \& %d => %20s %d \& %20s\n", i, assem[i], order[i], assem[order[i]]);
    }
  printf("\n");
}

void view_state(void)
{
  printf("INST & %d, \& %d => %20s\n", index, order[index], assem[order[index]]);
  printf("\n");
}

void load_assem_file(void)
{
  char token = ' ';
  char tokens[80];
  int i = 0;
  int j = 0;
  boolean done = FALSE;
  while (NOT_EOF)
    {
    while (NOT_EOF & done == FALSE)
      { 
      token = fgetc(in);
      if (isspace(token) & NOT_EOF)
      token = fgetc(in);
      if (COMMENT)
        { 
        while (NOT_NEWLINE)
        token = fgetc(in);
        }
      else
        { 
        if (NOT_EOF)
        }
    }

    while (NOT_EOF & done == FALSE)
      { token = fgetc(in);
      while (isspace(token) & NOT_EOF)
      token = fgetc(in);
      if (COMMENT)
        { 
        while (NOT_NEWLINE)
        token = fgetc(in);
        }
      else
        { 
        if (NOT_EOF)
        }
printf("NEW VALUE == ");
order[index] = get_input();
printf("\n");
}

void set_view_range(void)
{
printf("ENTER VIEW RANGE (LO,HI) == ");
scanf("%d,%d", &start, &finish);
printf("\n");
}

change_old_file(char * name, char * oname, char * fname)
{
if (get_filename(name, oname, fname) == TRUE)
return;
in = fopen(name, "r");
load_asm_file();
fclose(in);
in = fopen(oname, "r");
initialize();
load_order_file();
fclose(in);
view();
assemble();
}

void save_file(char * oname, char * fname)
{
int i;
if (index != 0)
{
out = fopen(name, "w");
out2 = fopen(oname, "w");
for (j = 0; j < index; j++)
{
fprintf(out, "%a", assembl[order[j]]);
fprintf(out, "%a\n", order[j]);
}
fclose(out);
fclose(out2);
}
else
{
printf("!! WARNING: NO DATA TO SAVE !!\
");
printf("\n");
}
}

void clear(void)
{
int i;
for (i = 0; i < 1000; i++)
order[i] = 999;
}

boolean quit()
{
int ch;
printf("DO YOU REALLY WANT TO QUIT (y,n) == ");
ch = getchar();
if (ch == 'n')
{
    return TRUE;
}
else
{
    return FALSE;
}
}

boolean query_save(char * oname)
{
int ch;
printf("SAVE CHANGES TO %a (y,n) == ");
ch = getchar();
if (ch == 'n')
{
    return TRUE;
}
else
{
    return FALSE;
}
}

int goto_instruction(void)
{
int input;
printf("GOTO INSTRUCTION NUMBER == ");
scanf("%d", &input);
printf("\n");
return input;
}

}
int main(void) {
    int selector;
    boolean done = FALSE;
    char name[20];
    char oname[20];
    char fname[20];
    boolean save;
    message();
    help_message();
    while (done == FALSE) {
        selector = command_menu();
        switch (selector) {
        case 'n':
            index = 0;
            assemble_new_file(name, oname, fname);
            save = TRUE;
            break;
        case 'o':
            index = 0;
            change_old_file(name, oname, fname);
            save = TRUE;
            break;
        case 'a':
            assemble();
            break;
        case 'c':
            clear();
            break;
        case 'v':
            view();
            break;
        case 's':
            save_file(oname, fname);
            break;
        case 'q':
            if (save == TRUE) {
                if (query_save(oname) == TRUE) {
                    save_file(oname, fname);
                }
            }
            done = quit();
            break;
        case '?':
            command_help();
            break;
        default:
            printf("ILLEGAL COMMAND - TRY AGAIN");
        }
    }
    return 0;
}

void copy_file(void) {
    char token = ' ';
    while (token != EOF) {
        token = fgetc(in);
        if (token != EOF) {
            fputc(token, out);
        }
    }
}

int main(void) {
    char mname[20];
    char ename[20];
    printf("*** MIPS PREPROCESSOR ***\n\n");
    get_filename(mname, ename);
    in = fopen(mname, "r");
    out = fopen("machine", "w");
    copy_file();
    fclose(in);
    fclose(out);
    in = fopen(ename, "r");
    out = fopen("expected", "w");
    copy_file();
    fclose(in);
    fclose(out);
    printf("*** PREPROCESSOR COMPLETE ***\n\n");
    return 0;
}
APPENDIX F - TEST PROGRAMS

<table>
<thead>
<tr>
<th>ALTEST - ALU IMMEDIATE</th>
<th>ALTEST - ALU REGISTER</th>
<th>ALTEST - ALU IMMEDIATE</th>
</tr>
</thead>
<tbody>
<tr>
<td># test file for alu immediate arithmetic instructions</td>
<td># test file for arithmetic instructions # 3 operand, register type</td>
<td># subu - subtract unsigned</td>
</tr>
<tr>
<td># lui - load upper immediate</td>
<td># lui - load upper immediate</td>
<td>lui 1 0 0</td>
</tr>
<tr>
<td>lui 1 0</td>
<td>lui 1 0</td>
<td>lui 1 0</td>
</tr>
<tr>
<td>lui 1 1</td>
<td>lui 1 1</td>
<td>lui 1 1</td>
</tr>
<tr>
<td>lui 1 a</td>
<td>lui 1 a</td>
<td>lui 1 a</td>
</tr>
<tr>
<td>lui 1 f</td>
<td>lui 1 f</td>
<td>lui 1 f</td>
</tr>
<tr>
<td>lui 1 ffff</td>
<td>lui 1 ffff</td>
<td>lui 1 ffff</td>
</tr>
<tr>
<td># ori - or immediate</td>
<td># ori - or immediate</td>
<td>ori 1 0 0</td>
</tr>
<tr>
<td>ori 1 0 0</td>
<td>ori 1 0 0</td>
<td>ori 1 0 a</td>
</tr>
<tr>
<td>ori 1 2 5</td>
<td>ori 1 2 5</td>
<td>ori 1 2 5</td>
</tr>
<tr>
<td>ori 1 2 f</td>
<td>ori 1 2 f</td>
<td>ori 1 2 f</td>
</tr>
<tr>
<td>ori 1 5555</td>
<td>ori 1 5555</td>
<td>ori 1 5555</td>
</tr>
<tr>
<td>ori 1 2 1 0</td>
<td>ori 1 2 1 0</td>
<td>ori 1 2 1 0</td>
</tr>
<tr>
<td>ori 2 1 1</td>
<td>ori 2 1 1</td>
<td>ori 2 1 1</td>
</tr>
<tr>
<td>ori 2 1 2</td>
<td>ori 2 1 2</td>
<td>ori 2 1 2</td>
</tr>
<tr>
<td>ori 2 1 ffff</td>
<td>ori 2 1 ffff</td>
<td>ori 2 1 ffff</td>
</tr>
<tr>
<td># andi - and immediate</td>
<td># andi - and immediate</td>
<td>ori 1 0 a</td>
</tr>
<tr>
<td>andi 2 1 0</td>
<td>andi 2 1 0</td>
<td>ori 2 0 0</td>
</tr>
<tr>
<td>andi 2 1 1</td>
<td>andi 2 1 1</td>
<td>ori 2 0 5</td>
</tr>
<tr>
<td>andi 2 1 2</td>
<td>andi 2 1 2</td>
<td>addi 3 1 2</td>
</tr>
<tr>
<td>andi 2 1 ffff</td>
<td>andi 2 1 ffff</td>
<td>addi 3 1 2</td>
</tr>
<tr>
<td># xor - exclusive or immediate</td>
<td># xor - exclusive or immediate</td>
<td>addi 3 1 2</td>
</tr>
<tr>
<td>xor 2 1 0</td>
<td>xor 2 1 0</td>
<td>addi 3 1 2</td>
</tr>
<tr>
<td>xor 2 1 1</td>
<td>xor 2 1 1</td>
<td>addi 3 1 2</td>
</tr>
<tr>
<td>xor 2 1 2</td>
<td>xor 2 1 2</td>
<td>addi 3 1 2</td>
</tr>
<tr>
<td>xor 2 1 ffff</td>
<td>xor 2 1 ffff</td>
<td>addi 3 1 2</td>
</tr>
<tr>
<td># addi - add immediate</td>
<td># addi - add immediate</td>
<td>addi 3 1 2</td>
</tr>
<tr>
<td>addi 2 0 0</td>
<td>addi 2 0 0</td>
<td>ori 1 0 a</td>
</tr>
<tr>
<td>addi 2 0 0</td>
<td>addi 2 0 0</td>
<td>ori 2 0 0</td>
</tr>
<tr>
<td>addi 2 0 0</td>
<td>addi 2 0 0</td>
<td>addi 3 2 1</td>
</tr>
<tr>
<td>addi 2 1 ffff</td>
<td>addi 2 1 ffff</td>
<td>addi 3 2 1</td>
</tr>
<tr>
<td>lui 1 ffff</td>
<td>lui 1 ffff</td>
<td>addi 3 2 1</td>
</tr>
<tr>
<td>addi 1 2 1 0 0 0</td>
<td>addi 1 2 1 0 0 0</td>
<td>addi 3 2 1</td>
</tr>
<tr>
<td>addi 2 1 5</td>
<td>addi 2 1 5</td>
<td>addi 3 2 1</td>
</tr>
<tr>
<td>addi 2 1 ffff</td>
<td>addi 2 1 ffff</td>
<td>addi 3 2 1</td>
</tr>
<tr>
<td># addiu - add immediate unsigned</td>
<td># addiu - add immediate unsigned</td>
<td>addi 3 2 1</td>
</tr>
<tr>
<td>addiu 1 0 0</td>
<td>addiu 1 0 0</td>
<td>ori 1 0 a</td>
</tr>
<tr>
<td>addiu 2 1 0</td>
<td>addiu 2 1 0</td>
<td>ori 2 0 0</td>
</tr>
<tr>
<td>lui 1 ffff</td>
<td>lui 1 ffff</td>
<td>addiu 3 2 1</td>
</tr>
<tr>
<td>addiu 1 2 1</td>
<td>addiu 1 2 1</td>
<td>lui 1 ffff</td>
</tr>
<tr>
<td>addiu 2 1 ffff</td>
<td>addiu 2 1 ffff</td>
<td>lui 1 ffff</td>
</tr>
<tr>
<td>addiu 1 2 ffff</td>
<td>addiu 1 2 ffff</td>
<td>addiu 1 2 ffff</td>
</tr>
<tr>
<td># slti - set on less than immediate</td>
<td># slti - set on less than immediate</td>
<td>ori 1 0 a</td>
</tr>
<tr>
<td>slti 1 0 2</td>
<td>slti 1 0 2</td>
<td>ori 2 0 0</td>
</tr>
<tr>
<td>slti 2 1 ffff</td>
<td>slti 2 1 ffff</td>
<td>sub 3 1 2</td>
</tr>
<tr>
<td>slti 2 1 0</td>
<td>slti 2 1 0</td>
<td>ori 1 0 a</td>
</tr>
<tr>
<td>slti 2 1 1</td>
<td>slti 2 1 1</td>
<td>ori 2 0 0</td>
</tr>
<tr>
<td>slti 2 1 2</td>
<td>slti 2 1 2</td>
<td>add 3 2 1</td>
</tr>
<tr>
<td>slti 2 1 3</td>
<td>slti 2 1 3</td>
<td>add 3 2 1</td>
</tr>
<tr>
<td>addi 1 0 fffe</td>
<td>addi 1 0 fffe</td>
<td>ori 1 0 a</td>
</tr>
<tr>
<td>slti 1 2 ffff</td>
<td>slti 1 2 ffff</td>
<td>ori 2 0 0</td>
</tr>
<tr>
<td>slti 2 1 0</td>
<td>slti 2 1 0</td>
<td>sub 3 1 2</td>
</tr>
<tr>
<td>slti 2 1 1</td>
<td>slti 2 1 1</td>
<td>sub 3 2 1</td>
</tr>
<tr>
<td># sltiu - set on less than immediate unsigned</td>
<td># sltiu - set on less than immediate unsigned</td>
<td>add 3 2 1</td>
</tr>
<tr>
<td>sltiu 1 0 2</td>
<td>sltiu 1 0 2</td>
<td>sltiu 2 1 0</td>
</tr>
<tr>
<td>sltiu 2 1 0</td>
<td>sltiu 2 1 0</td>
<td>sltiu 2 1 1</td>
</tr>
<tr>
<td>sltiu 2 1 1</td>
<td>sltiu 2 1 1</td>
<td>sltiu 2 1 2</td>
</tr>
<tr>
<td>sltiu 2 1 2</td>
<td>sltiu 2 1 2</td>
<td>sltiu 2 1 3</td>
</tr>
<tr>
<td>halt</td>
<td>halt</td>
<td>halt</td>
</tr>
</tbody>
</table>
model test - move to/from hi/lo registers

lui 3 abod
ori 3 3 1234
mvhi - move to hi
mvhi 3

# div - divide
ori 1 0 48
div 1 2 # 72d/9d
addi 2 0 ffff
div 1 2
addi 1 0 ffff
div 1 2
ori 2 0 9
div 1 2
ori 2 0 9
div 1 2
# divu - divide unsigned
ori 1 0 48
ori 2 0 9
divu 1 2 # 72d/9d
addi 2 0 ffff
divu 1 2 # 72d/9d
addi 1 0 ffff
divu 1 2 # -72d/-9d
ori 2 0 9
divu 1 2 # -72d/-9d
ori 2 0 9
divu 1 2
addi 2 0 ffff
divu 1 2
addi 1 0 ffff
divu 1 2
ori 2 0 9
divu 1 2
ori 2 0 9
divu 1 2

# sva - shift right arithmetic
sra 0 0 0
sra 0 0 0
lui 1 0000
ori 1 1
ori 1 1
lor 1 3333
ori 1 1 2222
sra 2 1
sra 2 1
lor 1 4444
ori 1 1 4444
sra 2 1
sra 2 1
sra 2 1 # run for 2000

# sllv - shift left logical
ori 2 0 0
ori 1 64
ori 3 0 4
srlv 2 13
ori 3 0 10
srlv 2 13

# sllv - shift right logical
variable
ori 2 0 0
lui 1 0000
ori 1 1
ori 3 0 4
srlv 2 13

# sra - shift right arithmetic
srl 0 0 0
srl 0 0 0
srl 1 2222
srl 1 2222
sra 2 1
sra 2 1
lor 1 4444
ori 1 1 4444
sra 2 1
sra 2 1
sra 2 1 # run for 2000

NOTE: ONLY BEHAVIORAL MODEL TEST PROGRAMS USE THE HALT INSTRUCTION. HALT IS REPLACED WITH THE BREAK INSTRUCTION FOR DATAFLOW AND STRUCTURAL MODELS.

S1 TEST - SHIFT

# test file for shift functions

$