9-9-2007

SIMD pipelined processor implemented on a FPGA

Benjamin Mar

SIMD PIPELINED PROCESSOR IMPLEMENTED ON AN FPGA

BY

BENJAMIN D. MAR

B.S., ELECTRICAL & COMPUTER ENGINEERING
WORCESTER POLYTECHNIC INSTITUTE, 2005

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Engineering

The University of New Mexico
Albuquerque, New Mexico

July 2007

Benjamin D. Mar
Candidate

Electrical and Computer Engineering
Department

This thesis is approved, and it is acceptable in quality and form for publication on microfilm:

Approved by the Thesis Committee:

, Chairperson

Accepted:

Dean, Graduate School

Date

Acknowledgements

Many thanks to Dr. Marios Pattichis for his guidance, vision, and humor throughout the duration of implementing this thesis and for taking time to work with me to make sure everything fell into place.

Thanks to Sandia National Laboratories for supporting me through this thesis and going the extra mile to make sure I finished.

Thanks to my family and friends who gave me the strength and support I needed to continue my journey when times were hard. Without them, my thesis would not be what it is today.

SIMD PIPELINED PROCESSOR IMPLEMENTED ON AN FPGA

BY

BENJAMIN D. MAR

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Engineering

The University of New Mexico
Albuquerque, New Mexico

July, 2007

SIMD PIPELINED PROCESSOR IMPLEMENTED ON AN FPGA

by

Benjamin D. Mar

B.S. Electrical & Computer Engineering, Worcester Polytechnic Institute, 2005

M.S. Computer Engineering, University of New Mexico, 2007

Abstract

The goal of this thesis was to create a processor using VHDL that could be used for educational purposes as well as a stepping stone in creating a reconfigurable system for digital signal processing or image processing applications. To do this a subset of MIPS instructions were chosen to demonstrate functionality within a five stage pipeline (instruction fetch, instruction decode, execution, memory, and write back) processor in simulation and in synthesis. A hazard controller was implemented to handle data forwarding and stalling. The basic MIPS architecture was extended by adding single-cycle multiplication functionality and single-cycle SIMD instructions. The architecture contains parameters for easy modification of SIMD units depending on the needs of the processor.

The SIMD architecture was designed with distributed memory so that every memory unit received the same address. This simplifies the address logic so that the processor does not have to use a complex addressing mode. The memory can be pictured as row and columns method of access.

The SIMD instructions were chosen to be able to perform binary operations to implement future morphological operations and to use the multiply and add operations for implementing MACs to perform convolution and filtering operations in future image processing applications.

The board being used to verify the processor was a Xilinx University Program (XUP) board that contains Xilinx Virtex II Pro XC2VP30 FPGA, package FF896. The maximum number of units that can be instantiated in the FPGA on the XUP board is eight units which would use the entire FPGA slice area. This allows the processor to complete eight sets of 32-bit data operations per cycle when the SIMD pipeline is full.

The design was shown to operate at the maximum speed of 100 MHz and utilize all the area of the FPGA. The processor was verified in both simulation and synthesis. The new soft-core 32-bit SIMD processor extends existing soft-core processors in that it provides a reconfigurable SIMD-pipeline allowing it to operate on multiple inputs concurrently, with 32-bit operands and a single-cycle throughput.

CONTENTS

List of Figures

List of Tables

1. Introduction

1.1. Introduction

General purpose computing architectures must address many important questions such as: What will the instruction set architecture look like? How will the chip resources be used? What the hardware should be responsible for verses the software? etc. Each architecture answers these questions in a fashion to best meet the needs of the work to be performed. Some issues faced by designers are: the efficiency of area usage, the speed of the system, and the reconfigurability of the system – the flexibility of the system.

There are many different types of architectures implemented in modern processors. On one side of the spectrum there are the general purpose fixed processors that implement a complex instruction set like the Intel's Pentiums and on the other side there are soft-core processors that implement a reconfigurable, reduced instruction set like Xilinx's MicroBlaze.

Both processors can perform different applications but might not exactly meet the criteria needed. If the processor needs to process multiple data in a fast manner but also be flexible to change to different datapath parameters, neither processor meets both criteria. The Pentiums have single instruction multiple data (SIMD) capability [12] but the datapath is set in silicon and cannot be changed while the MicroBlaze can be reconfigured to conform to any datapath but does not have SIMD capability and arithmetic operations require multiple clock cycles to obtain results [18].

This thesis addresses these criteria and presents a synthesizable VHDL solution.

1.2. Thesis

This thesis is focused on the development of a processor architecture that would function as an educational tool for students, while also being a flexible foundation for real applications such as morphological or convolution operations for image processing. Two important metrics were used to measure the processor – performance and functionality. Performance was measured by the speed of the processor and throughput of data. The paradigm was to maximize the operating frequency while minimizing the area used by the core logic in order to place as many single instruction multiple data (SIMD) units in the field programmable gate array (FPGA) while supporting a complete core of arithmetic and logical instructions. The objective was to produce a synthesizable VHDL architecture that could be reconfigured to different specifications and retain readability for others to use.

1.3. Contributions

The contributions of this thesis are as follows:

- A synthesizable VHDL description of a five stage pipeline processor with hazard control using a subset of MIPS instructions.

- A synthesizable VHDL description of a SIMD version of the five stage pipeline processor with hazard control. The SIMD instructions are based on the original MIPS instructions for arithmetic and logical operations. The instruction set is chosen to be able to implement calculations such as convolution and morphological operations.
- An analysis of the processors' maximum frequency and area usage of the FPGA.

1.4. Thesis Summary

The rest of the thesis is organized as follows:

- Chapter 2 provides the reader with the history of the MIPS processor and follows with the idea of single instruction multiple data concept, then investigates past architectures using SIMD implementations.
- Chapter 3 describes the architecture of the processor built for this thesis. The architecture with single instruction single data (SISD) instructions will be described in the first section followed by the SIMD architecture modifications and additions. The test board will be introduced with discussions on how to modify the processor architecture to conform to different hardware environments.
- Chapter 4 discusses the results obtained with simulation and synthesis using the XUP FPGA board. The results include timing analysis and area usage of the FPGA.
- Chapter 5 contains an overview of the project and discusses future work and future research applications.

2. Background

2.1. MIPS

The MIPS processor was first designed in 1985 by John L. Hennessy although the research started in 1981 while at Stanford University. The goal of the MIPS processor design was to increase processor performance by using deep instruction pipelines. This allowed the clock frequency of the processor to be based on the critical path within one of the stages rather than the entire instruction cycle like the traditional designs that waited to complete an entire instruction before moving to the next.

A major aspect of the MIPS design was to demand that all instructions take only one cycle to complete, thereby removing any needs for interlocking. The design of the MIPS processor eliminated a number of useful instructions that would take multiple steps to complete. The long instructions were left out as it was thought the performance of the system would be dramatically improved due to the fact the processor could run at much higher clock rates. Increasing the speed was difficult with interlocking, as the locks took up additional chip area that slowed down the overall speed.

Many observers claimed that the MIPS design would not be used in commercial products with the elimination of instructions. The argument was one of CISC (complex instruction set computer) versus RISC (reduced instruction set computer): that if a complex instruction was replaced with many simpler instructions this would reduce the

speed. The argument ignored the fact that the speed of the design comes from the pipelines, not the instructions themselves [17].

The theory of the MIPS architecture is used in many university curriculums to discuss pipelining. The DLX processor is based off of the MIPS and implements VHDL code to simulate the processor, however it is not synthesizable without extensive modification [1]. The basic MIPS architecture pipeline can be seen in Figure 1.

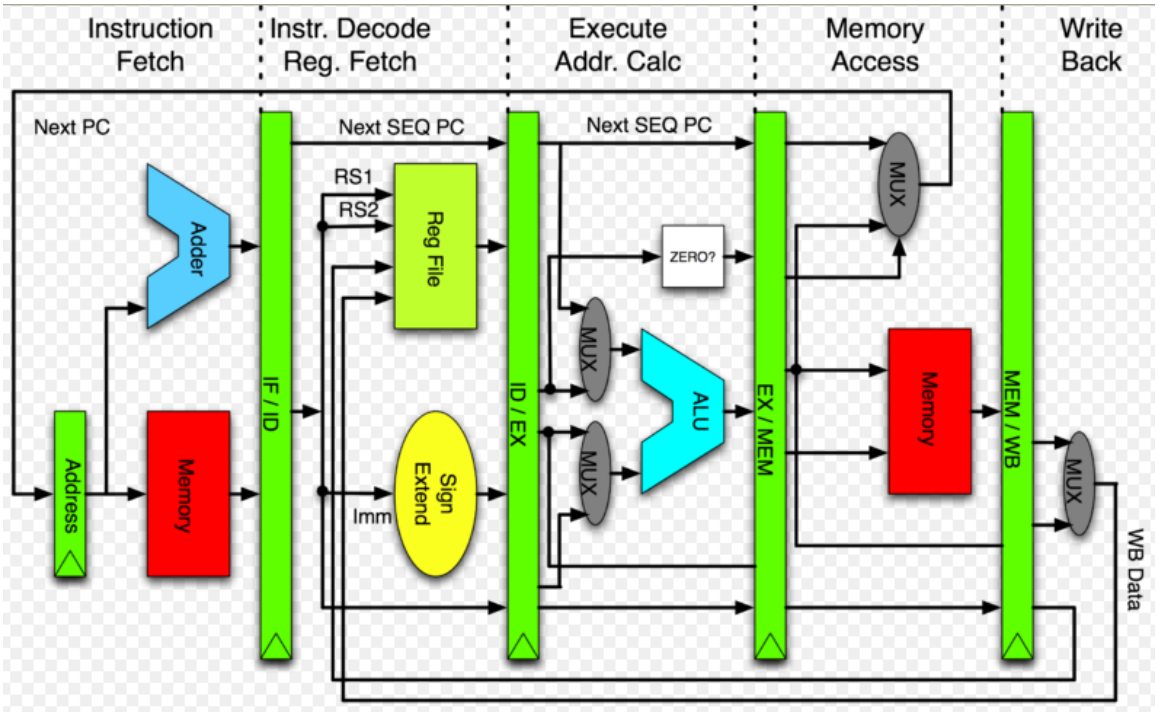


Figure 1: MIPS Basic Pipeline [17].

The basic MIPS architecture implemented a 5 stage pipeline that divided the combinational logic of the datapath into sections. The maximum frequency increased by dividing the processor into smaller sections; decreasing the critical path from the single cycle routing to a smaller section of that routing.

2.2. SIMD

The Single Instruction Multiple Data (SIMD) concept is a method of improving performance in applications where highly repetitive operations need to be performed. SIMD is a technique of performing the same operation, be it arithmetic or otherwise, on multiple pieces of data simultaneously.

Traditionally, when an application is being programmed and a single operation needs to be performed across a large dataset, a loop is used to iterate through each element in the dataset and perform the required procedure. During each iteration, a single piece of data has a single operation performed on it. This is known as Single Instruction Single Data (SISD) programming. Loops are very inefficient, as they can iterate thousands of times. Ideally, to increase performance, the number of iterations of a loop needs to be reduced.

One method of reducing iterations is known as loop unrolling [10]. This takes the single operation that was being performed in the loop, and carries it out multiple times in each iteration.

The SIMD concept takes loop unrolling one step further by incorporating the multiple actions in each loop iteration, and performing them simultaneously. With SIMD, not only can the number of loop iterations be reduced, but also the multiple operations that are required can be reduced to a single, optimized action.

SIMD does this through the use of vectors. A SIMD vector can be used as an argument for a specific instruction that will then be performed on all elements in the vector simultaneously. Because of this, the number of values that can be loaded into the

vector directly affects performance; the more values being processed at once, the faster a complete dataset can be completed. This size depends on two things: 1) The data size being used and 2) The SIMD implementation.

When values are stored in SIMD vectors and worked upon by a SIMD operation, they are actually moved to a special set of CPU registers where the parallel processing takes place. The size and number of these registers is determined by the SIMD implementation being used. SIMD makes use of multiple CPU functional units; independent functional units for arithmetic and Boolean operations that execute concurrently.

The SIMD implementation can be enhanced with pipelining the program instructions. Instruction pipelining is the decomposition of instruction execution into a linear series of autonomous stages, allowing each stage to simultaneously perform a portion of the execution process (such as decode, calculate effective address, fetch operand, execute, and store).

2.3. Related Work in SIMD

Many SIMD architectures make the use of multiple processors to carry out the multiple executions. These processors, also called processing elements (PE), must be connected in a network that can range from a mesh, a pyramid, to a hypercube formation.

The CLIP 4 [7], MPP [2], and GAPP [16] processors use a mesh network configuration. These systems use several bit serial processors to execute the same operation on the same image or images but on different neighborhoods. They have no

pipelining (only one level of operation going on at the same time), no parallel fetch of neighborhoods, and not even one pixel but one bit is to be fetched at a time to each processor. These processors utilize a central control unit that generates and broadcasts a global address and control word. This information is picked up by the memory modules and the processors.

Processors like WPM [15] use a pyramid network. A pyramid network is an extension of a mesh using a tree hierarchy. Each level has connections to the levels above and below, giving each internal PE nine connections. All of the PEs operate in a SIMD mode under the direction of a single controller. The pyramid layout is more difficult than the mesh layout, resulting in only small pyramids being built [4].

Other SIMD architectures have moved into the FPGA realm. VIP and MATRIX make use of the programmable logic cells for their PEs.

The VIP [3] SIMD architecture has a 2D torus interconnection topology of its PE. Each PE has a local memory. The VIP architecture is composed of three basic components: the SIMD controller, the processing matrix and the I/O controller. Those components are connected by a shared global bus and two control buses. The processing matrix is a set of identical PEs interconnected in a 2D grid topology. The I/O controller manages off-board communication and initiates memory transfers. The SIMD controller decodes and executes the program stored in its instruction memory, and read or write to its data memory.

The VIP system consists of four programmable chips. The processing matrix and SIMD controller are each implemented by an SRAM-based Altera EPF81500 FPGA that has approximately 16,000 usable gates. The I/O controller is implemented by an

EPROM-based Altera EPM7192 EPLD (3,750 usable gates), and an AMCC 55933 PCI controller with its configuration EPROM.

MATRIX [7] is composed of an array of identical, 8-bit functional units overlayed with a configurable network using FPGAs. Each functional unit contains a 256x 8-bit memory, an 8-bit ALU and multiply unit, and reduction control logic including a 20 x 8 NOR plane. The network is hierarchical supporting three levels of interconnect. Functional unit port inputs and non-local network lines can be statically configured or dynamically switched.

The ALU supports the standard set of arithmetic and logic functions including NAND, NOR, XOR, shift, and add. With optional input inversion, this extends to include OR, AND, XNOR, and subtract. A configurable carry chain between adjacent ALUs, allow cascading of ALUs to perform wide-word operations. The ALU also includes an 8 x 8 multiply-add-add operation; the multiply operation takes two operating cycles to complete producing the low 8 bits of the product on the first cycle and the high 8 bits on the second cycle.

Since the datapath size and assignment of control resources is not fixed for a MATRIX component, MATRIX has greater flexibility to match the datapath composition and granularity to the needs of the application.

Processor	Approach	Clock Period	Data size per PE
CLIP 4 (1980)	Mesh network, 9216 PEs, bit-serial processor with 32 bits of memory per PE	400ns	1 bit
GAPP (1984)	Mesh network, 72 bit-serial processors with 128 bits of memory per PE	100ns	1 bit
MPP (1983)	Mesh network, 16000 PEs with 1024 bits of memory per PE	100ns	1 bit
WPM (1989)	Pyramid network, Deeply Pipelined, 256 PEs, 8000 bits of memory off chip	100ns	16 bits
VIP (1996)	FPGA, Torus network, 28 PEs-4 FPGAs, 1.5MB external memory	60ns	32 bits
MATRIX (1996)	FPGA, Configurable network, 256 x 8-bit memory per PE	10ns	8 bits

Table 1: Overview of SIMD Architectures

3. Methodology

The goal of this thesis was to create a processor that could be used for educational purposes as well as a stepping stone in creating a reconfigurable system for digital signal processing or image processing applications. To do this a subset of instructions were chosen to demonstrate functionality within a five stage pipeline processor in simulation and in synthesis. Upon completion of the initial architecture, multiplication functionality and SIMD functionality were added.

3.1. Architecture

The architecture implemented was five stage pipeline that used the Harvard memory approach with a load-store 32-bit instruction set. The Harvard memory architecture was used to simplify the controls for memory access by using two separate memories for the program data (instruction memory) and the workable data (data memory). This resolves any structural hazard that a single memory could produce. The load-store or register-to-register architecture was chosen as storage internal to the processor is faster than external memory. Even though the memory IPs are within the processor for this project, later work might require larger external memory for data manipulations.

3.1.1. Instruction Set

The processor uses a subset of the MIPS instruction set. Each instruction is 32-bits in length and is identified by the six most significant bits or the six least significant bits. The MIPS uses three types of instructions: register to register (R-type), immediate to register (I-type), and jumps (J-type). R-type instructions are used primarily for work type instructions, i.e. addition, subtraction. I-type instructions are used much like the R-type instruction, but use a 16-bit immediate value in its instruction format. J-type instructions are used for jump instructions. The 28 instructions used [14] are given in Figure 2.

addu – unsigned addition of two registers,
addiu – unsigned addition of register and immediate value,
subu – unsigned subtraction of two registers,
mul – signed multiplication of two registers,
and – logical and of two registers,
andi – logical and of register and immediate value,
or – logical or of two registers,
ori – logical or of register and immediate value,
xor – logical xor of two registers,
xori – logical xor of register and immediate value,
sll – shift left logical,
srl – shift right logical,
sra – shift right arithmetic,
slt – set less than,
sltu – set less than unsigned,
slti – set less than immediate,
sltiu – set less than immediate unsigned,
lui – load upper immediate,
lw – load word,
sw – store word,
beq – branch on equal,
bne – branch not equal,
bgez – branch greater or equal zero,
bltz – branch less than zero,
j – jump,
jal – jump and link,
jr – jump register,
break – breakpoint exception.

Figure 2: SISD Instruction Set.

The full descriptions of these instructions can be seen in Appendix A: Instruction Reference. The mul instruction implemented in this processor takes the 16 least significant bits of each register and multiplies them instead of the entire 32 bits as regular MIPS uses. The mul instruction only takes 16 bits instead of 32 bits as the processor uses a Xilinx core IP for the multiplier, which does not support 32 bit operations.

These 28 instructions were chosen out of the 189 MIPS32 [14] instruction set as they provide the core functionality of the processor and completeness. The add, subtract,

and multiply cover the arithmetic operations; the and, or, and xor cover the logical operations; the sll, srl, and sra cover shifting needs; the set less than and branch instructions cover comparisons; lw and sw works with memory; the branch and jump instructions cover loops and section jumps; and the break instruction allows for program debugging and testing. The instructions left out of this processor included instructions that work with floating point values, caches, translation lookaside buffers, move register values, traps, and variations of what was implemented (branch, jump, memory access, etc).

The unsigned versions of the arithmetic operations were chosen as the term "unsigned" is a misnomer in MIPS32. The difference between the add and addu instruction is that the add instruction traps on overflow and the addu instruction does not trap. Since both instructions have the same functionally, besides the trapping, addu was preferred to avoid adding additional logic to handle traps and exceptions. This choice helps reduce the overall logic of the processor.

3.1.2. Pipeline

A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. These elements or stages of a pipeline are executed in parallel time-sliced fashion with some amount of buffer storage inserted between stages. The stages in this pipeline are: instruction fetch, instruction decode, execution, memory, and write back. These pipeline stages are the basic stages that were also implemented in the simple MIPS processor.

3.1.2.1. Instruction Fetch Stage

The Instruction Fetch (IF) stage directs the flow of the program and retrieves the instructions from memory. The IF stage consists of the program counter (PC) register, the synchronous instruction memory, and passed signals from the instruction decode stage that has calculated logic for branches, jumps as well as signals for stalling the PC. The IF stage schematic can be seen in Figure 3.

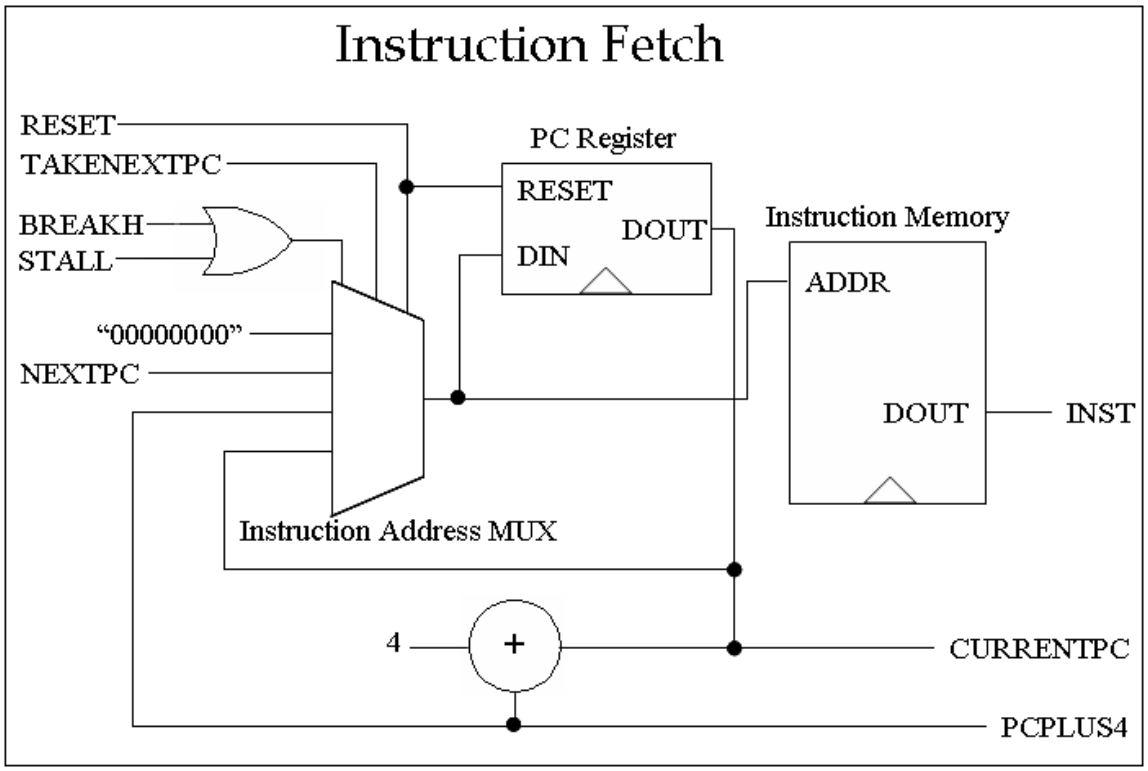


Figure 3: Schematic of Instruction Fetch Stage.

The PC is always enabled; combinational logic is used to determine which address should be registered and sent to the instruction memory. If the pipeline is stalled or if there is a break instruction, the PC holds its current value as to prevent losing an

instruction. If there is a branch or jump, then the PC will take the NEXTPC value that holds the destination address of the branch or jump. The default setting is to go to the sequential instruction which has the value of PC + 4.

The instruction memory is a Xilinx IP core: Single Port Block Memory v6.2. The memory is set up as a ROM, which reads in a .coe file that holds the instructions, with a width of 32 and depth of 1024. The depth can be modified as needed. Since the instruction memory is synchronous, the next address needs to be at the memory at the same time as the PC register so that it would not introduce another cycle into the pipeline.

3.1.2.2. Instruction Decode Stage

The instruction decode (ID) stage holds most of the control logic of the pipeline as it decodes what needs to happen for each instruction. The ID stage contains the register file, branch logic module, nextPC logic module, and controller. The ID stage schematic can be seen in Figure 4.

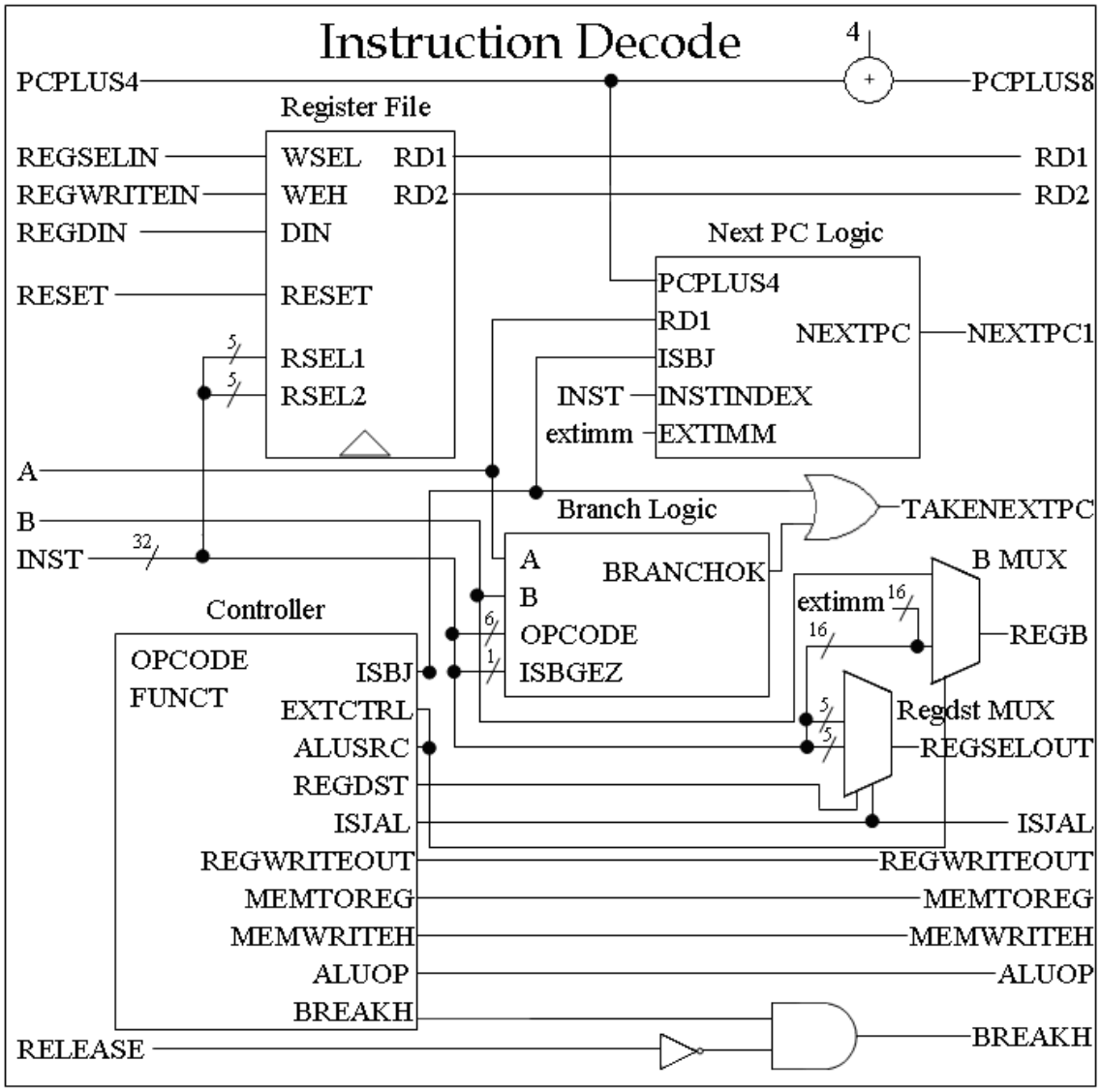


Figure 4: Schematic of Instruction Decode Stage.

This processor implementation allows for a branch delay slot; the instruction following the branch or jump always executes. The branch logic module exists so the pipeline does not have to kill an instruction and create a no operation in the killed instruction's place in the pipeline instruction stream. The branch logic could be resolved in the execution stage but by then there are two instructions in the pipeline. The

instruction in the ID stage would be executed but the one in the IF stage would need to be killed.

The branch logic module is used to determine if a beq, bne, bgez, or bltz instruction will actually take the branch. The inputs come from after the forwarding multiplexers (MUX) which are not in the ID stage which will be discussed in section 3.1.2.6 Hazard Detection. The branch logic module then sets BRANCHOK signal high if the branch is to be taken.

The register file is a collection of 31 general purpose registers and a zero register that cannot be modified. Register 31 is used for the return address in a jal instruction. The return address is the PCPLUS8 output, which saves the address after the branch delay slot.

The controller is the component that decodes the instructions and sets signals to allow for proper execution. The controller only uses the opcode (the six MSB of the instruction) and the function (the six LSB of the instruction). Since all of the R-format instructions have the same opcode, they are grouped together. The jr instruction is included as an R-format instruction since the instruction code follows the R-format even though it jumps to a new address. The parameters for the opcode are given in Figure 5.

```
constant rtype: std_logic_vector (5 downto 0):= "000000";
constant addiu: std_logic_vector (5 downto 0):= "001001";
constant slti: std_logic_vector (5 downto 0):= "001010";
constant sltiu: std_logic_vector (5 downto 0):= "001011";
constant andi: std_logic_vector (5 downto 0):= "001100";
constant ori: std_logic_vector (5 downto 0):= "001101";
constant xori: std_logic_vector (5 downto 0):= "001110";
constant lui: std_logic_vector (5 downto 0):= "001111";
constant lw: std_logic_vector (5 downto 0):= "100011";
constant sw: std_logic_vector (5 downto 0):= "101011";
constant beq_bne: std_logic_vector (4 downto 0):= "00010";
constant bgez_bltz: std_logic_vector (5 downto 0):= "000001";
constant j: std_logic_vector (5 downto 0):= "000010";
constant jal: std_logic_vector (5 downto 0):= "000011";
constant mul: std_logic_vector (5 downto 0):= "011100";
```

Figure 5: Instruction Opcodes from controller.vhd.

These opcodes are used to determine the cases for all of the control signals. The code in Figure 6 shows the jump, branch, and break signal assignments:

```
jump <= '1' when OPCODE = rtype AND FUNCT = "001000" else '0';
jtype <= '1' when OPCODE(5 downto 1) = "00001" else '0';
branch <= '1' when OPCODE(5 downto 1) = beq_bne OR OPCODE = bgez_bltz
          else '0';
ISBJ <= jtype & jump & branch;
BREAKH <= '1' when OPCODE = rtype AND FUNCT = "001101" else '0';
ISJAL <= '1' when OPCODE = jal else '0';
```

Figure 6: Branch, Jump, and Break Signal Assignments from controller.vhd.

The JUMP signal denotes a jr instruction while the JTYPE signal is for either a j or jal instruction. ISBJ (is branch or jump) goes to the nextPC module as a 3 bit signal to determine with input should be used to calculate the next address. BREAKH goes to the IF stage and to the inter-stage registers in the processor to disable them from going to the next instruction. ISJAL is used inside the hazard controller to account for jal. REGWRITEH, MEMWRITEH, MEMTOREG, ALUSRC, and ALUOP all leave the ID

stage and connect to inter-stages registers. The following is what they do after they pass through the pipeline to where they are used:

- REGWRITEH goes to the register file and determines when to allow register writes.
- MEMWRITEH connects to the write enable on the data memory.
- MEMTOREG connects to the register write MUX that decides if memory data should be written to the register file or a calculation should be and also acts as the memory read enable.
- ALUSRC goes to the B MUX that determines to pass either value from forward MUXM or the output of the extender to the B source of the ALU.
- ALUOP goes to the ALU controller and will be discussed in section 3.1.2.3 Execution Stage.
- EXTCTRL connects to the extend logic to determine if it should zero or sign extend.
- REGDST is used to determine the register to write back to.

If an unknown opcode passes through the controller, then all of the enables become inactive so the data does not become corrupted.

The Next PC module uses the ISBJ signal and determines the next address. For a branch, the NEXTPC signal gets the current PC + 4 + the immediate value. For a jr, the NEXTPC signal gets the result from the Forward MUXA (see in section 3.1.2.7, Figure 15). For j or jal, the NEXTPC signal gets concatenation of the top 4 bits of PC + 4 with 26 bits of the immediate value with 2 zero bits. The NEXTPC is then passed out of the ID stage to the IF stage.

3.1.2.3. Execution Stage

The execution (EXE) stage carries out the operations of the instructions. The EXE stage consists of the shifter, comparator, ALU controller, ALU and multiplier. The EXE stage schematic can be seen in Figure 7.

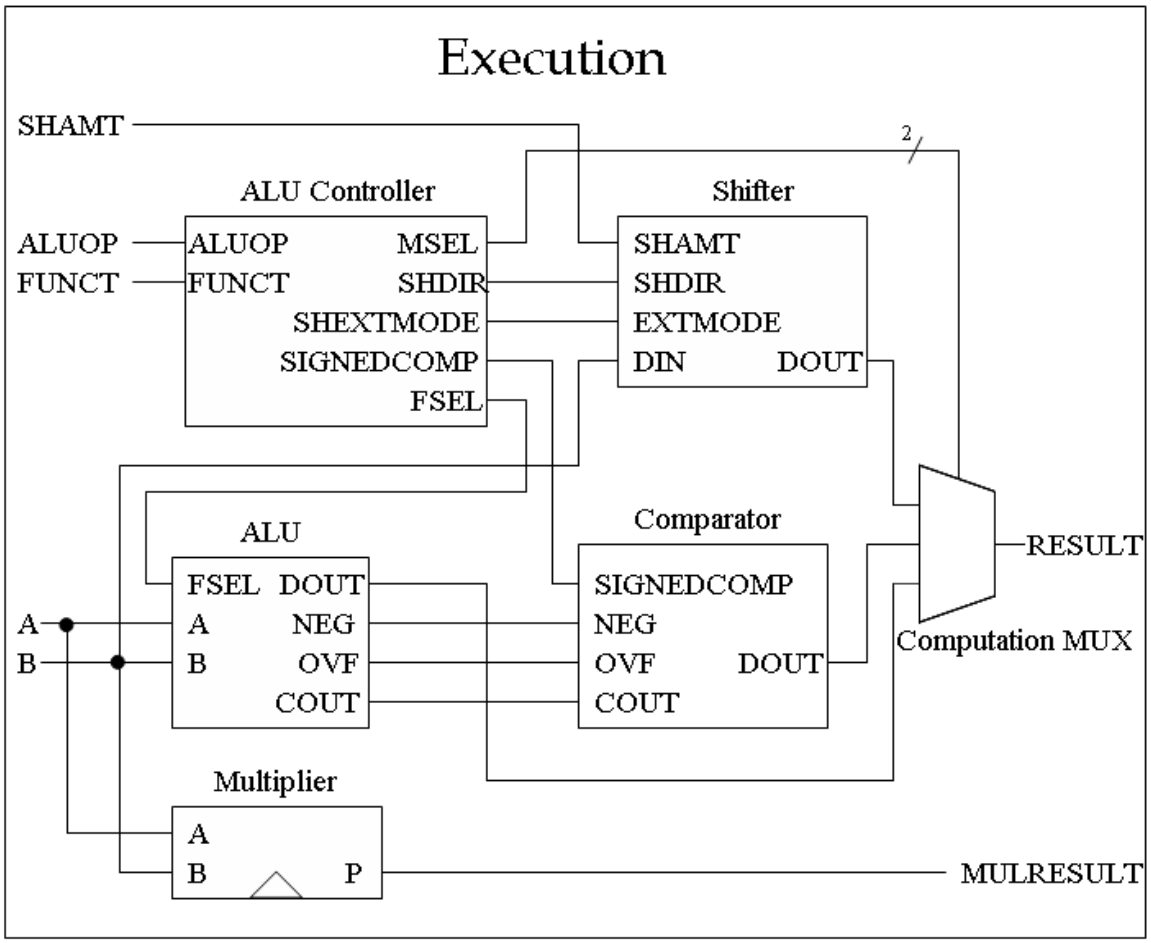


Figure 7: Schematic of Execution Stage.

The shifter is used for the sll, srl, and sra instructions. It shifts right or left by the shift amount (SHAMT) which comes from the instruction bits 10 downto 6 through an

inter-stage register. The shift direction (SHDIR) and extend mode (EXTMODE) are taken from the ALU controller.

The comparator is used for the slt, sltu, slti, and sltiu instructions. It compares the status flag from the ALU (carry out, negative, and overflow) for the correct result and uses determines the value of a signed or unsigned instruction with a signal from the ALU controller (SIGNEDCOMP).

The multiply module is used for the mul instruction. It multiplies the 16 least significant bits and returns a 32 bit result. The multiply is a Xilinx IP core: multiplier v9.0 and is also synchronous. Since the multiplier is synchronous, the multiplier result cannot be retrieved until the next clock cycle unlike the rest of the EXE stage components. This issue will be discussed in section 3.1.2.6 Hazard Detection.

The ALU controller controls what operation the ALU performs, which output reaches the result register, the shift direction and extension mode, and the signed mode for the comparator. The ALU controller is triggered on the FUNCT or the ALUOP and uses case statements to set the ALU controls FSEL for the ALU and MSEL for the 32x3 MUX output. The parameters for the ALUOP, FUNCT, and MSEL are given in Figure 8.

```
--ALUops
constant rtype: std_logic_vector (3 downto 0):= x"0";
constant addiu: std_logic_vector (3 downto 0):= x"1";
constant andi: std_logic_vector (3 downto 0):= x"2";
constant ori: std_logic_vector (3 downto 0):= x"3";
constant xori: std_logic_vector (3 downto 0):= x"4";
constant lui: std_logic_vector (3 downto 0):= x"5";
constant slti: std_logic_vector (3 downto 0):= x"6";
constant sltiu: std_logic_vector (3 downto 0):= x"7";

--Funct
constant addu: std_logic_vector (5 downto 0):= "100001";
constant subu: std_logic_vector (5 downto 0):= "100011";
constant and1: std_logic_vector (5 downto 0):= "100100";
constant or1: std_logic_vector (5 downto 0):= "100101";
constant xor1: std_logic_vector (5 downto 0):= "100110";
constant sll1: std_logic_vector (5 downto 0):= "000000";
constant srl1: std_logic_vector (5 downto 0):= "000010";
constant sra1: std_logic_vector (5 downto 0):= "000011";
constant slt: std_logic_vector (5 downto 0):= "101010";
constant sltu: std_logic_vector (5 downto 0):= "101011";

--MSEL
constant shifter: std_logic_vector (1 downto 0):= "00";
constant compare: std_logic_vector (1 downto 0):= "01";
constant alu: std_logic_vector (1 downto 0):= "10";
```

Figure 8: ALU Control Parameters from ALUcontroller.vhd.

ALUOP determines what FSEL and MSEL should be set to. When dealing with an R-type instruction, the FUNCT distinguishes what the select signals should be. The code in Figure 9 shows how this is implemented.

```
case ALUop is
      when rtype => case FUNCT is
                        when addu => FSEL <= "000"; MSEL <= alu;
                        when subu => FSEL <= "100"; MSEL <= alu;
                        when and1 => FSEL <= "001"; MSEL <= alu;
                        when or1 => FSEL <= "010"; MSEL <= alu;
                        when xor1 => FSEL <= "011"; MSEL <= alu;
                        when sll1 => FSEL <= "000"; MSEL <= shifter;
                        when srl1 => FSEL <= "000"; MSEL <= shifter;
                        when sra1 => FSEL <= "000"; MSEL <= shifter;
                        when slt => FSEL <= "100"; MSEL <= compare;
                        when sltu => FSEL <= "100"; MSEL <= compare;
                        when others => FSEL <= "000"; MSEL <= alu;
                        end case;
      when andi => FSEL <= "001"; MSEL <= alu;
      when ori => FSEL <= "010"; MSEL <= alu;
      when xori => FSEL <= "011"; MSEL <= alu;
      when lui => FSEL <= "101"; MSEL <= alu;
      when slti => FSEL <= "100"; MSEL <= compare;
      when sltiu => FSEL <= "100"; MSEL <= compare;
      when others => FSEL <= "000"; MSEL <= alu;
end case;
```

Figure 9: EXE Select Lines from ALUcontroller.vhd.

The FSEL signal determined which operation within the ALU would be preformed. The ALU operations are defined in Table 2.

FSEL	Operation
"000"	add
"001"	and
"010"	or
"011"	xor
"100"	sub
"101"	lui

Table 2: ALU Operations.

The MSEL chooses between the inputs of the shifter output, comparison output, and the ALU output using the computation MUX. The ALU controller also assigns signals for execution as given in Figure 10.

```
SHDIR <= '1' when ALUop = rtype AND FUNCT = sll1 else '0';
SHEXTMODE <= '1' when ALUop = rtype AND FUNCT = sra1 else '0';
SIGNEDCOMP <= '1' when ALUop = slti OR FUNCT = slt else '0';
```

Figure 10: EXE Control Signals from ALUcontroller.vhd.

- SHDIR goes to the shifter and determines which direction to shift.
- SHEXTMODE also goes to the shifter and determines between zero and sign extend a shift.
- SIGNEDCOMP goes to the comparator to choose to compare signed or unsigned.

The ALU module computes the arithmetic and logical results using the FSEL from the ALU controller to define which operation to perform on the data A and B that comes from the inter-stage registers.

3.1.2.4. Memory Stage

The Memory (MEM) stage accesses the data memory when either an sw or lw instruction is issued. The data memory is also a Xilinx IP core: Single Port Block Memory v6.2. The memory is set up to read and write and can be initialized with a .coe file that holds data. The MEM stage executes at the same time in the pipeline as the multiplier in the EXE stage. This is possible as the address for the data memory is calculated in the ALU and the result is passed to the memory before being registered in the inter-stage register. Since the memory is synchronous, the data for a load word is ready the next clock cycle.

3.1.2.5. Write Back Stage

The Write Back (WB) stage simply takes either the result value, multiply result, or data from memory and sends it back to the register file to write the appropriate register.

3.1.2.6. Hazard Detection

The hazard detection module has two purposes. The first is to forward data from instructions currently in the pipeline that have not written back to the register file to an instruction in the ID stage that needs the value. The second is to stall the pipeline when forwarding the correct value is not available for that clock cycle. The hazard controller prevents read after write hazards with the data forwarding. The write after write and write after read hazards cannot happen in this implementation since all instructions are executed in order according to the program being run.

The data forwarding problem occurs when an instruction writes to a register and then next instruction uses that register as one of its operands. Take the instruction sequence given in Figure 11.

```
addu   $1, $2, $3  ------ $1 = $2 + $3
 or     $4, $1, $3 ------ $4 = $1 | $3
subu   $5, $1, $2 ------- $5 = $1 – $2
and    $6, $1, $1 ------ $6 = $1 & $1
```

Figure 11: Forwarding Hazard Sequence.

When the addu enters the EXE stage, the or enters the ID stage. The or instruction needs the updated value for register $1, so the value must be forwarded from the EXE stage. The same method applies to the subu instruction. When the subu enters the ID stage the addu enters the WB stage. The value of register $1 is still not in the register file so the value must be forwarded from the WB stage. By the time the and instruction enters the ID stage the $1 value is written into the register file so it can retrieve the value from there. This process can be seen in Figure 12.

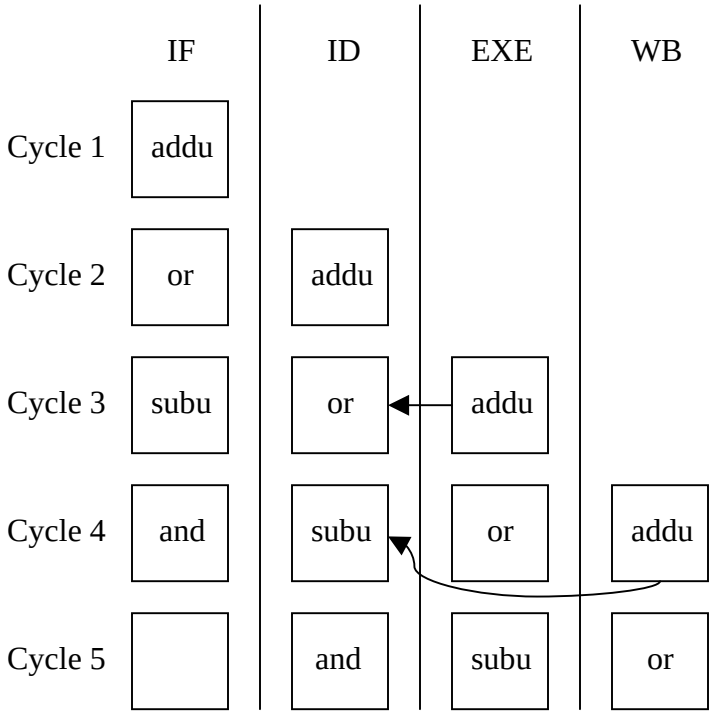


Figure 12: Forwarding Illustration in the Pipeline

The jal instruction also uses the forwarding multiplexers since it needs to store the current PC value plus two spaces (PCPLUS8). The control logic for one of the forwarding multiplexers select bits can be seen in Figure 13.

```
process(REGWRITEEXE, WBEXE, RSEL1ID, REGWRITEMEM, WBMEM, ISJAL)
begin
      if REGWRITEEXE = '1' AND WBEXE /= "00000" AND WBEXE = RSEL1ID
            then
                  FORWARDA <= computationout;
      elsif REGWRITEMEM = '1' AND WBMEM /= "00000" AND WBMEM = RSEL1ID
            then
                  FORWARDA <= memoryout;
      elsif ISJAL = '1' then
                  FORWARDA <= jal;
      else --takes care of RESET case
                  FORWARDA <= regfileout;
      end if;
end process;
```

Figure 13: Forwarding MUXA Select Line of Value A from hazardcontrol.vhd.

FORWARDA connects to the select bit of the forward MUXA. The arguments make sure that the zero register is never forwarded but all other registers are if the register select identification matches with the write back register select identification in either the EXE or WB stage. The default case uses the register file value when nothing needs to be forwarded.

The second issue the hazard detection monitors is the need to stall the pipeline. This occurs when a load word instruction is followed by an instruction that uses the register that the load word is loading data from memory into or an instruction that uses the register that stores the result of a multiply. The stall logic can be seen in Figure 14.

```
STALL <= '1' when REGWRITEEXE = '1' AND MEMTOREGEXE = '1' AND
                  WBEXE /= 0 AND WBEXE = RSEL2ID else
         '1' when REGWRITEEXE = '1' AND MEMTOREGEXE = '1' AND
                  WBEXE /= 0 AND WBEXE = RSEL1ID else
         '1' when ISMUL = '1' AND WBEXE /= 0 AND WBEXE = RSEL1ID else
         '1' when ISMUL = '1' AND WBEXE /= 0 AND WBEXE = RSEL2ID else
         '0';
```

Figure 14: Stall Logic from hazardcontrol.vhd.

A load word instruction can be seen in the pipeline when the REGWRITEEXE and MEMTOREGEXE signals are high. So if an instruction uses the same register that the load word is loading to or a multiply result is writing to (WBEXE = RESEL*ID), the STALL signal will go high.

The forwarding MUX would take care of this for the load word, however when the instruction that needs the loaded register is in the ID stage, the load word is in the EXE which only contains the address of where the data is located. Since the multiplier is synchronous, its result is not available until the WB stage like the load word instruction. The hazard control sends out a stall signal to stall the IF and ID stages while letting the rest of the stages continue. The next cycle the load word is in the WB stage and the value is at the memory output which can be forwarded back to the ID stage for the instruction that needs it.

3.1.2.7. Inter-stage Connections

The inter-stage connections connect all of the stages and hazard control together to form the processor. The processor schematic can be seen in Figure 15.

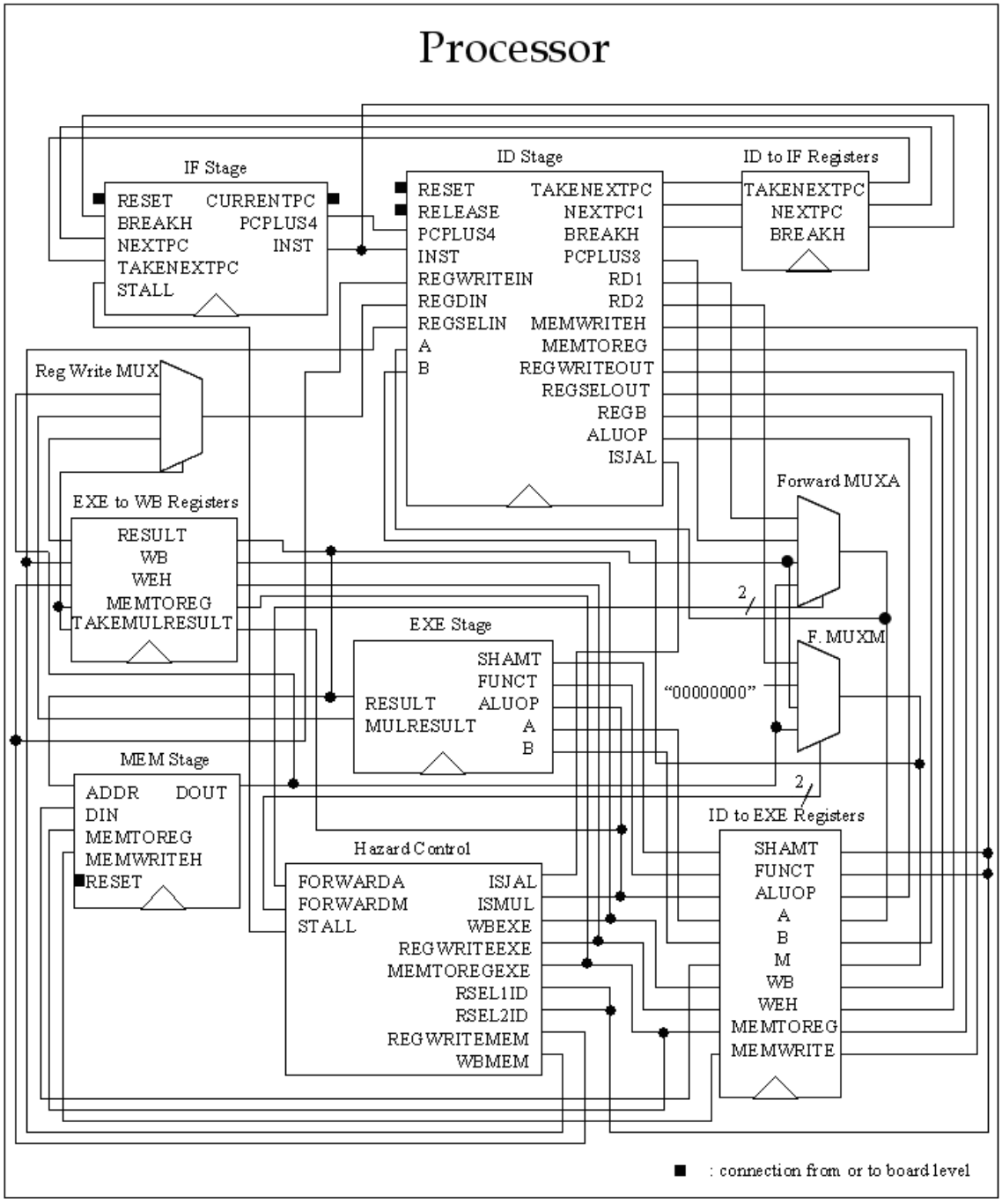


Figure 15: Schematic of Processor.

Between the IF and ID stages there are three inter-stage registers; one for TAKENEXTPC which enables the instruction address MUX in the IF to take NEXTPC instead of PC + 4, one for NEXTPC which holds the address for branches or jumps, and

the other for BREAKH which is inverted to enable the release of the break instruction. These registers are needed to create the delay slot for branches and jumps.

Along the ID and EXE stage division are the two forwarding MUX, inter-stage registers A and B for the EXE stage, inter-stage register M for the MEM stage, and inter-stage registers to hold control signals. The inter-stage registers for the control signals contain ALUOP, WB, WEH, MEMWRITE, and MEMTOREG. The outputs of the control signals go to as follows:

- ALUOP goes into EXE stage to the ALU controller with the most significant bit going to the TAKEMULRESULT register in between the EXE and WB stages.
- MEMWRITE and MEMTOREG go directly to the data memory.
- WB, WEH, and MEMTOREG go to the inter-stage control registers between EXE and WB stages.
- MEMTOREG goes to both the data memory and the next inter-stage control registers because it is the select of the register write multiplexer, along with TAKEMULRESULT, that determines whether to take the memory data, the calculated data, or the multiplier data and also is the memory read enable.

Since the memory is synchronous the enable must be there before the signal gets to the register write multiplexer. The select signal comes from the hazard control for both forwarding MUXA and forwarding MUXM. The assignments for forward MUXA can be seen in Figure 16.

```
forwardmuxA: component mux32x4 port map(SEL => forwardA,
                                        A => rd1,
                                        B => pcplus8,
                                        C => resultEXE,
                                        D => regdin,
                                        DOUT => regadin);
```

Figure 16: Forwarding MUX of Value A from processor.vhd.

The first input comes from the register file, rd1 for forward MUXA; rd2 for forward MUXM. The second input is for jal, PCplus8 for forward MUXA to store the address of the instruction after the delay slot; x"00000000" for forward MUXM as this is not needed for jal. The third input comes from the result of the comparator in the EXE stage and is the same for both forward MUXA and forward MUXM. The fourth input comes from after the register write multiplexer in the WB stage and is the same for both forwarding multiplexers. These four inputs account for all of the forwarding needs of the design and the outputs go to the inputs of the inter-stage registers for A and M. The input for inter-stage register B comes from B MUX (located in the ID stage) of the forward MUXM result and the immediate value from the extend logic within the ID stage. A and B outputs go to the comparator input in EXE stage and M output goes to the data in (DIN) input of the data memory.

Between the EXE and WB stage there are the inter-stage control registers for WB, WEH, TAKEMULRESULT, and MEMTOREG as well as an inter-stage result register. The inputs of the control registers come from the outputs of the previous inter-stage register and the outputs go to the register write multiplexer in the WB stage and to the register file for the register write back and write enable (WB and WEH). The input of the result register comes from the output of the computation module and the output connects to the register write multiplexer in the WB stage as the first input.

3.2. SIMD Architecture

The SIMD architecture builds off of the architecture in section 3.1. To achieve multiple data, multiple basic units needed to be added and connected within the architecture and the control logic also needed to be updated to account for the different datapaths. A block diagram of the new architecture is shown in Figure 17.

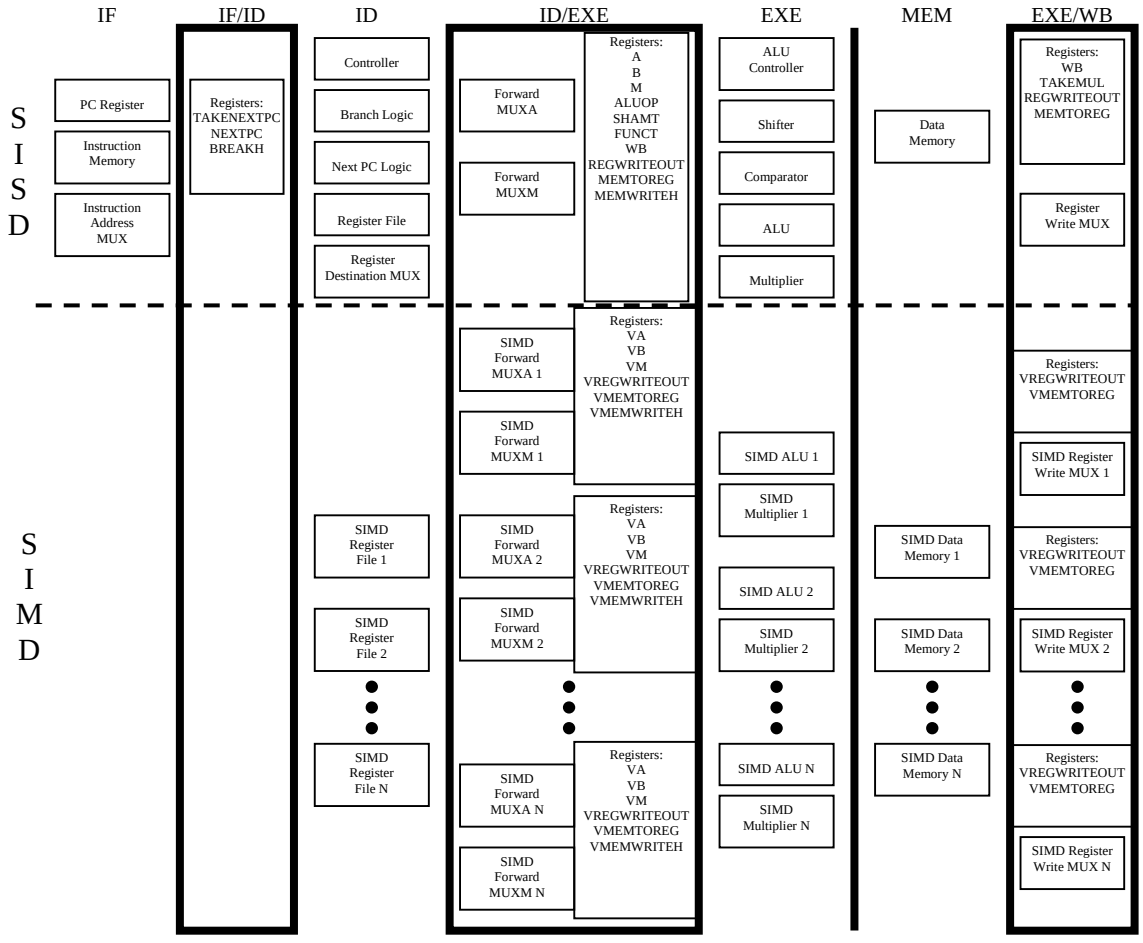


Figure 17: SISD and SIMD Block Diagram of Processor.

3.2.1. SIMD Instruction Set

There are 11 instructions implemented for SIMD use in addition to the 28 MIPS instructions implemented previously. All instructions, whether SISD or SIMD, are stored in the one instruction memory (see Figure 17). Each instruction is 32-bits in length and is identified by the six most significant bits. The 11 instructions used are shown in Figure 18.

vaddu – unsigned addition of n-pairs of two registers,
vaddiu – unsigned addition of n-pairs of register and immediate value,
vand – logical and of n-pairs of two registers,
vandi – logical and of n-pairs of register and immediate value,
vor – logical or of n-pairs of two registers,
vori – logical or of n-pairs of register and immediate value,
vxor – logical xor of n-pairs of two registers,
vxori – logical xor of n-pairs of register and immediate value,
vmul – signed multiplication of n-pairs of two registers,
vlw – n load words,
vsw – n store words.

Figure 18: SIMD Instruction Set

Each of the SIMD instructions are coded like the SISD instructions except that they differ by one bit except for vandi. The adds, and, ors, xnors, load and store words have the 31st bit high as seen in Figure 19 for the addiu versus the vaddiu.

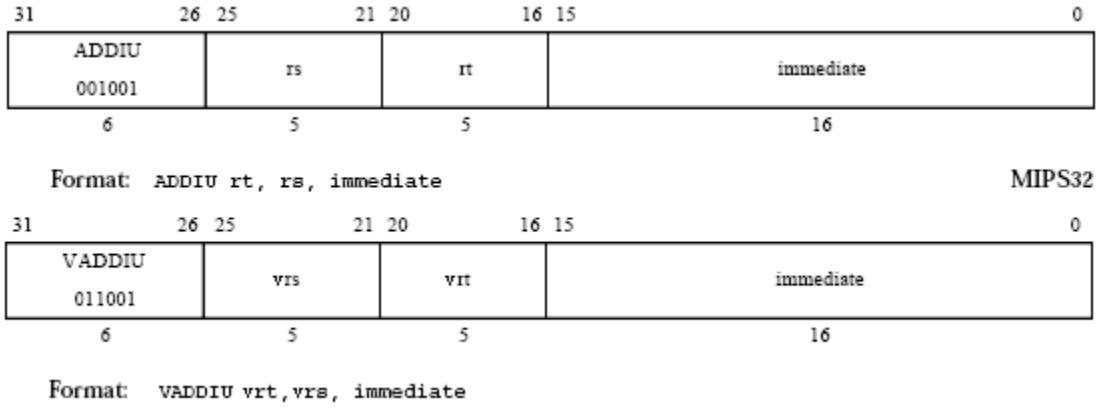


Figure 19: Addiu and Vaddiu Instruction Fields.

The vaddiu instruction operates as: vrt ← vrs + immediate. The multiply has the 32nd bit high as seen in Figure 20 for the mul versus the vmul.

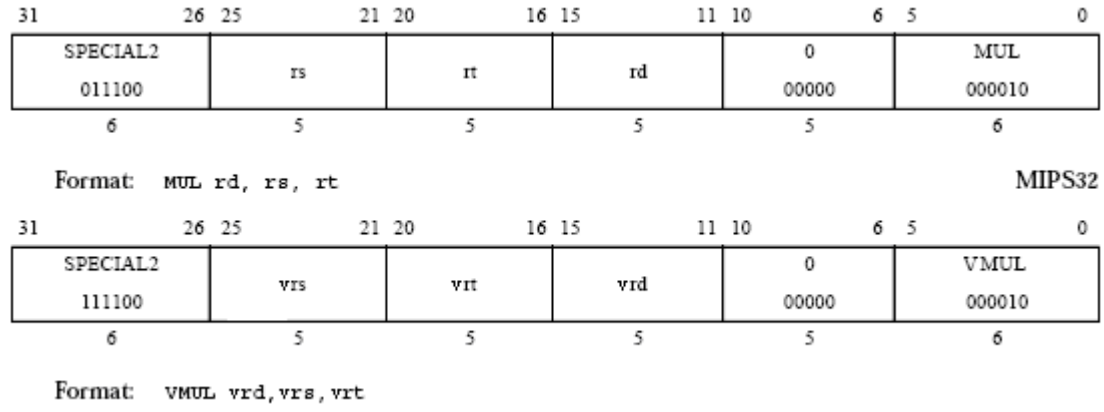


Figure 20: Mul and Vmul Instruction Fields.

The vmul instruction operates as: vrd ← vrs * vrt. Changing one bit allows the use of opcodes that are not in use for the 28 SISD instructions. The vandi instruction would have followed this method but the opcode would be the same as the mul SISD instruction. The opcode for the vandi instruction changes 3 bits from the andi instruction as seen in Figure 21.

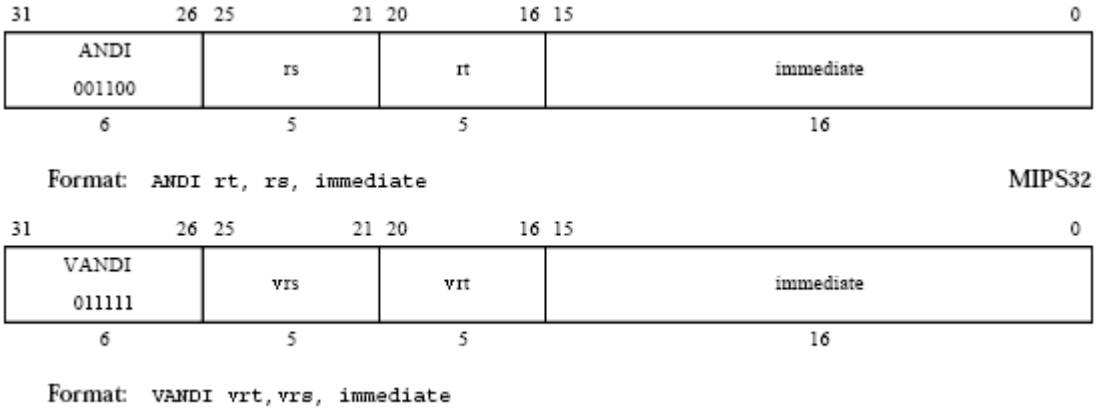


Figure 21: Andi and Vandi Instruction Fields.

The vandi instruction operates as: vrt ← vrs & immediate.

These SIMD instruction could be utilized to perform binary operations to implement morphological operations and could use the multiply and add operations for a MAC to perform convolution and filtering operations in image processing applications.

3.2.2. SIMD Pipeline

The SIMD pipeline only requires modifications in the ID, EXE, MEM, and WB stages as well as the hazard control and inter-stage connections as seen in Figure 17. The modules contain parameters so that the number of units can be changed depending on the needs of the processor. The default values instantiates four units (N = 4). A unit consists of a register file, ALU, multiplier, memory, and inter-stage registers and forwarding logic.

The SIMD instructions do not change the instruction flow of the program so all the modifications needed in the ID stage were to the controller and to add additional

register files. The controller needed three additional control signals to handle the register write (VREGWRITEH) and the memory reads (VMEMTOREG) and writes (VMEMWRITEH) for the SIMD instructions so the SISD datapath would not conflict with the SIMD datapath. Those three signals control the N SIMD units as each unit will perform the same function. The rest of the control signals (excluding the branch and jump logic) were modified to activate for the SIMD instructions as well as the SISD instructions. A segment of the modifications of the controller can be seen in Figure 22.

```
--SIMD assignments
VREGWRITEH <= '1' when OPCODE = vlw OR OPCODE = vrtype OR OPCODE = vmul
                  else
              '1' when OPCODE = vandi OR OPCODE = vori OR
                  OPCODE = vxori else
              '0';

VMEMWRITEH <= '1' when OPCODE = vsw else '0';

VMEMTOREG <= '1' when OPCODE = vlw OR OPCODE = vsw else '0';
-------------------

ALUSRC <= '1' when OPCODE(5 downto 3) = "001" AND OPCODE /= "001000"
            else
          '1' when OPCODE = lw OR OPCODE = sw else
          '1' when OPCODE = vlw OR OPCODE = vsw else --SIMD
          '1' when OPCODE = vandi OR OPCODE = vori OR OPCODE = vxori
            else --SIMD
          '0';
```

Figure 22: SIMD Control Signal Modifications from controller.vhd.

The architecture was designed with distributed memory so that every memory unit received the same address. This simplifies the address logic so that the processor does not have to use a complex addressing mode. The memory can be pictured as row and columns method of access (see Figure 17).

The number of SIMD units can be increased or decreased by changing the parameters in the generic port list and some modifications with signals associated with them. Figure 23 shows the generic ports of the processor.

```
entity processor is
       Generic (datasize : natural := 127;
                size : natural := 4);
       Port ( CLK : in  STD_LOGIC;
              RESET : in  STD_LOGIC;
              RELEASE : in  STD_LOGIC;
              CURRENT_PC : out STD_LOGIC_VECTOR (31 downto 0);
              CURRENT_INST : out STD_LOGIC_VECTOR (31 downto 0);
              STAT : out STD_LOGIC_VECTOR (7 downto 0);
              STALLED : out  STD_LOGIC);
```

Figure 23: SIMD Processor Entity from processor.vhd.

The size parameter is the number of SIMD units. The datasize parameter is the number of SIMD units * 32 – 1. Size is used for generating the correct number of units with generate statements. Datasize is used to prevent changing the port lists of all the stages; all of the different 32-bit values are combined into one port. Figure 24 shows how the SIMD Forward MUXA is instantiated.

```
type data_array is array (1 to size) of STD_LOGIC_VECTOR (31 downto 0);
----------------
vrd1_array(4) <= vrd1(31 downto 0);
vrd1_array(3) <= vrd1(63 downto 32);
vrd1_array(2) <= vrd1(95 downto 64);
vrd1_array(1) <= vrd1(127 downto 96);

genmuxandreg: for I in 1 to size generate
SIMDmuxA: component mux32x3 port map(SEL => vforwardA,
                                     A => vrd1_array(I),
                                     B => vresultEXE_array(I),
                                     C => vregdin_array(I),
                                     DOUT => vregadin(I));
end generate;

va <= vregAdout(1) & vregAdout(2) & vregAdout(3) & vregAdout(4);
```

Figure 24: SIMD Forward MUXA from processor.vhd.

The SIMD data signals use an array type to connect to the SIMD ALU, MUX, and registers. The total length of the data_array type corresponds to the parameter datasize. Since the data ports are of type STD_LOGIG_VECTOR(datasize downto 0), the data needs to be converted into the data_array type as is done for vrd1_array in Figure 24. These lines would need to be modified along with the parameters to instantiate different SIMD unit amounts.

3.3. XUP Board

The board being used to verify the processor was a Xilinx University Program (XUP) board that contains Xilinx Virtex II Pro XC2VP30 FPGA, package FF896. The processor was instantiated into an FPGA module to connect internal signals of the processor to the IO of the board (push buttons, dip switch, and LEDs). The board used was modified by adding additional connections for the LEDs seen in the bottom right corner of Figure 25.

Figure 25: XUP Board

The FPGA module supports two modes of operation: the onboard 31.25 MHz clock or a step clock triggered by pressing a push button. The clock mode is set by dip switch 1. An onboard 100 MHz clock is also available for use by modifying the user constraint file (UCF) which can be seen in Appendix D: FPGA User Constraint File.

3.4. Modules Specific to Hardware

The processor architecture can be used in different FPGAs or boards but might need some modification depending on the technology used. The memories and multipliers are Xilinx IP cores which can be used for other Xilinx FPGA that support them but would need to be changed for FPGAs not compatible. Changing the use of boards requires the user to use the UCF associated with the new board.

4. Results

The processor was completed in two phases; functionality of the SISD architecture and functionality of the SIMD architecture added on. Each phase was tested using the program instruction tests that can be seen in Appendix C: Program Codes. The programs were verified in simulation using ModelSim and in the XUP board using Xilinx ISE 8.2i Project Navigator.

4.1. Test Programs

The suite of test programs tested the correct functionality of the instructions, forwarding of data, and stalling the pipeline. The basic test tested the functionality of the instructions without any data hazards. The hazard test tested all three for the SISD. The SIMD test tested the SIMD instructions for all three categories.

The programs were converted from word description to hexadecimal values using MIPSASM v1.3 from UC Berkeley [13] for the SISD tests and hand coded for the SIMD tests. The test programs were configured with the instruction memory IP core using a .coe file with the hexadecimal values inside. The .coe must have the header as seen in Figure 26, followed by the hexadecimal data.

```
memory_initialization_radix=16;
memory_initialization_vector=
```

Figure 26: Header for .coe Initialized Memory File.

4.2. Simulation

All of the programs were verified in ModelSim to run correctly. Two example of test code are shown in Figure 27 and Figure 29.

```
        addiu   $2, $0, 5       #testing addiu
        addiu   $3, $0, 5       #testing addiu
        addiu   $4, $0, -5
        addiu   $23, $0, 0      #used to keep track of failed tests

        beq     $2, $2, L1      #testing beq
        ori     $0, $0, 0
        addiu   $2, $2, -5
L1:
        bne     $3, $0, L2      #testing bne
```

Figure 27: Basic Test Code Segment.

Figure 28 shows the simulation waveform of the code in Figure 27.

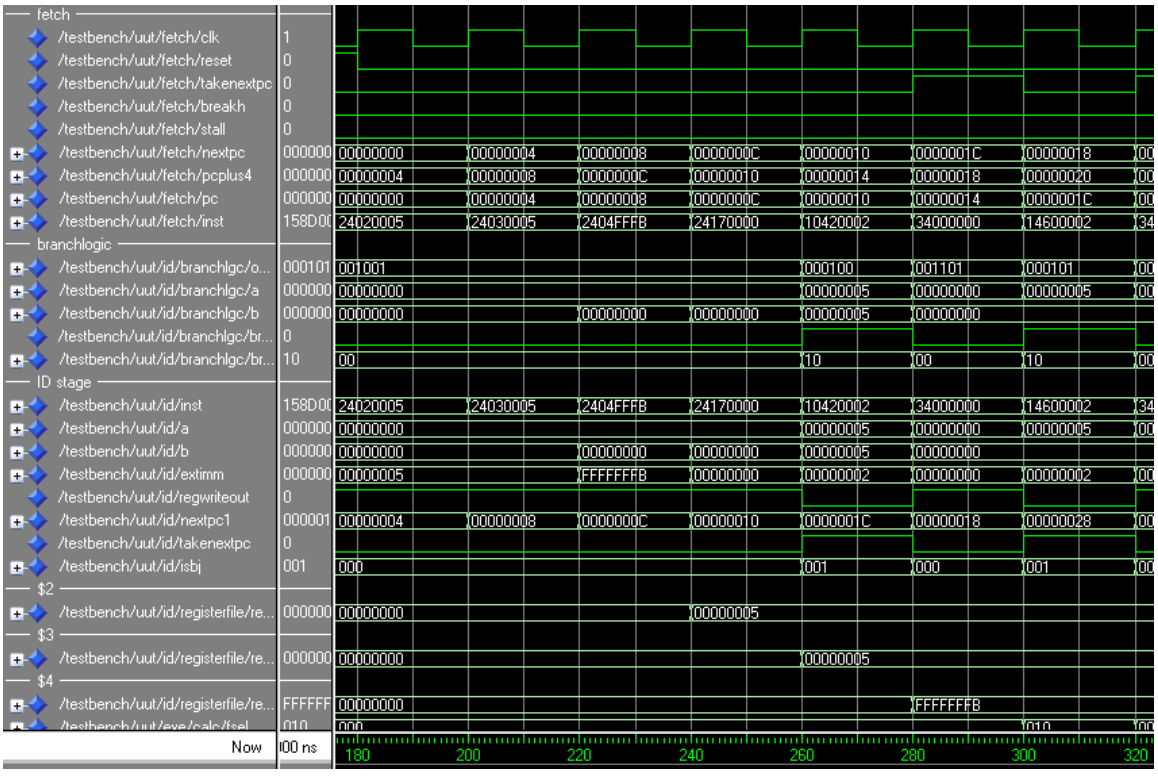


Figure 28: Simulation of Basic Test.

This waveform shows the full execution of the addiu instructions and the beq. At time 180 ns, the RESET goes low and the first addiu is in the IF stage. The result can be seen written in the register $2 at time 240 ns. The other addiu instructions can be followed the same way. The beq (inst = 0x10420002) instruction enters the pipeline at 260 ns. The next cycle the beq is in the ID stage and can be seen to branch by looking at TAKENEXTPC in that cycle and the PC in the next cycle. The TAKENEXTPC goes high starting at 280 ns which makes the PC take the NEXTPC value of 0x0000001C instead of the PCPLUS4 value of 0x00000018 in the next cycle (300 ns).

```
vaddu $20, $1, $2          #testing vaddu
vaddu $21, $20, $3
vaddu $22, $21, $20
vsw $22, 0($0)             #testing vsw
vlw $25, 0($0)             #testing vlw
vaddiu $23, $25, 0x1111    #testing vlw hazard
vaddiu $24, $0, 0x1000
```

Figure 29: SIMD Test Code Segment.

Figure 30 shows the simulation waveform of the code in Figure 29.

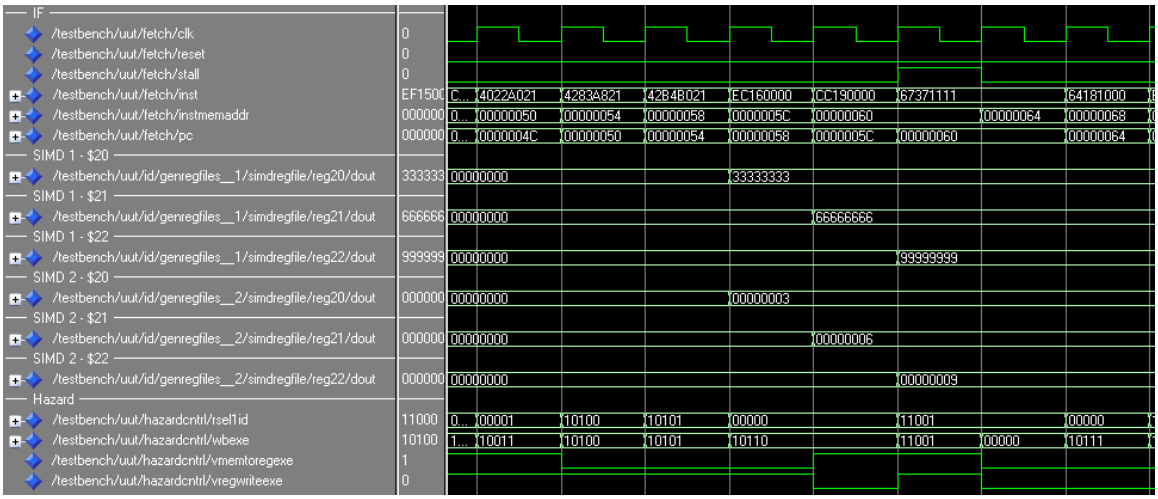


Figure 30: Simulation of SIMD Test

This waveform shows the full execution of the vaddu, vlw, and vsw instructions. The first vaddu is in the IF stage in the first full clock period. The result can be seen written in the SIMD 1 and SIMD 2 register $2 three cycles later. The other vaddu instructions can be followed the same way. In between the vlw (inst = 0xCC190000) and the vaddiu (inst = 0x67371111) the stall signal goes high to verify the vlw hazard detection works.

4.3. Board Testing

The board testing required internal signal to be pulled up to the FPGA_TOP module and connected to the LEDs. The internal signals pulled up to board level are the current PC, the instruction, the break status, and registers $23 of the first and second SIMD unit. Xilinx Chipscope could have been used to check the internal signals however Chipscope would be included in the synthesis and take up additional slices on the FPGA.

The synthesis verification used two clock modes to check the programs in the processor: the board clock mode and clock step mode. In board clock mode the processor clock receives the board clock of 31.25 MHz. The programs had to use the break instruction to see the values of the processor due to the speed of the clock. Without the break instructions the LEDs would look stable at a level of brightness related to how many times a LED was activated. In clock step mode the processor clock receives the clock signal from the bottom push button. This mode allows the tester to cycle through the entire program and trace the instruction flow. If resetting or releasing a break in the

clock step mode, the reset or release button must be held down while the clock button is pushed otherwise the release will not take hold and the reset will work but it will bypass the first instruction of the program.

The default display setting displays the PC and instruction. The break status and SIMD registers can be seen by changing the positions of the dip switches.

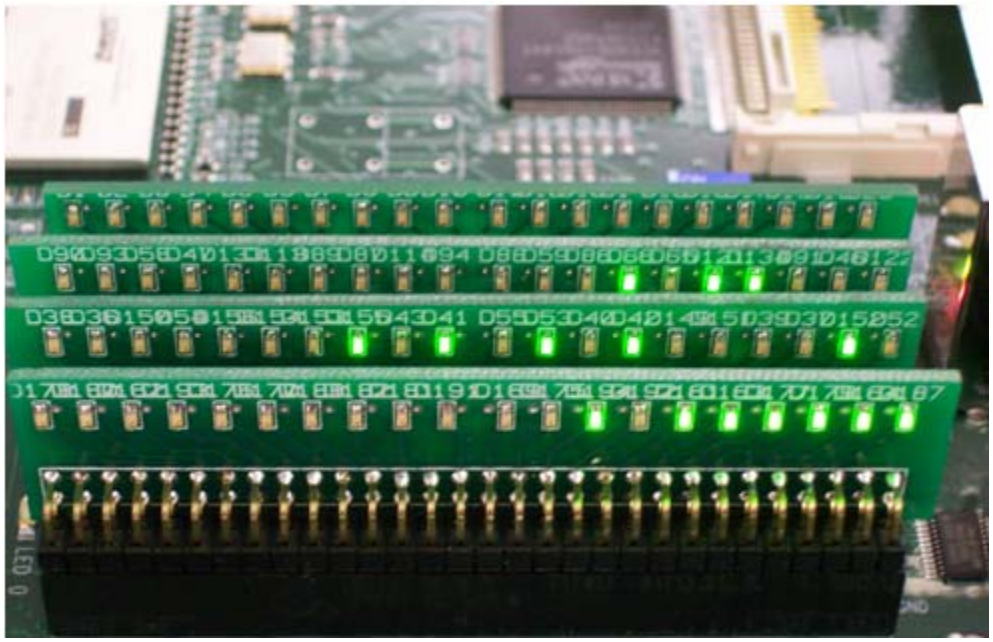


Figure 31: Synthesis Demonstration of Working Processor.

Figure 31 illustrates the display of the PC and instruction value. The bottom row of LEDs shows the 16 LSB of the instruction, the middle row shows the 16 MSB of the instruction, and the top row shows the PC. The four left most LEDs are not used on all rows. The display is read as: instruction 0x154200bf (bne) at address 0x00000058. This corresponds to the 23rd instruction in the hazard test which can be seen in Appendix C: Program Codes.

The SISD programs were verified by looking at the current PC, instruction, and break status while the SIMD programs were verified by the break status and registers.

While not all internal signals were verified as working properly these three signals are efficient for the SISD as the branches control most of the instruction flow. If a branch instruction produces a route counter to the actual route, then the processor is broken. The SIMD requires registers pulled up since there are no branch or jump instruction that run through the SIMD datapath, so just looking at the current PC and instruction would not validate the SIMD.

4.4. Maximum Frequency Minimum Clock Period

Utilizing the Xilinx timing analysis tools, the timing report indicates that the delay of the critical path of the processor with SISD implemented only is 20.933 ns or a maximum frequency of about 48 MHz. The delay of the critical path of the processor with SISD implemented with SIMD is 21.222 ns or a maximum frequency of about 47 MHz.

The critical path of both phases is excited by a branch instruction. The critical path is as follows: The instruction comes out of the instruction memory and reads the values of the registers from the register file in the ID stage. The register file values connect to the forwarding multiplexer. The branch address must be computed by taking the correct value from the hazard control multiplexer, then shifted by four, then sent to the branch logic module to determine if the branch should be taken, and finally the branch logic signals are registered at the next inter-stage register.

In theory if both critical paths are the same, the minimum periods should be the same as well, however, the delay is measured by the logical delay with the routing delay. The routing delay was measured to be about 70% of the total delay in both architectures.

	All Instructions	Without Multiply	Without Multiply, Branches, Jumps	One Unconditional Jump (j)
Number of Slices	2033	1879	1641	1683
% of Slices Used	14%	13%	11%	12%
Minimum Period	20.933 ns	22.116 ns	8.437 ns	8.940 ns

Table 3: Basic Pipeline Results.

Eliminating instructions could reduce the minimum period. Table 3 shows that by removing the datapath for the branch and jump instructions the processor could operate at the 100 MHz board frequency. If the j instruction was left in the architecture, the processor could run double the speed as well. This would allow for a program to stay in an infinite loop which is usable for some image processing applications.

	All Instructions	Without vmul	Without Branches and Jumps	One Unconditional Jump (j)
Number of Slices	7879	6994	7568	7598
% of Slices Used	57%	51%	55%	55%
Minimum Period	21.222 ns	19.903 ns	9.432 ns	7.784 ns

Table 4: SIMD Pipeline Results.

The results from just the SISD logic applies when the SIMD is included. The maximum frequency could double without having the SISD branch instructions.

This is shown to be an improvement or on the same level as previously designed architectures, VIP and MATRIX. The minimum period with all the SISD and SIMD instruction is 3 times faster than the minimum period of the VIP processor due to the

pipeline design (MIPS SIMD) versus one instruction operating at a time (VIP) and the processor without the branches is a few nanoseconds faster than the MATRIX. Beyond the MATRIX capabilities, the new SIMD processor can execute (in the EXE stage) a multiplication instruction in a single cycle. Furthermore, the new SIMD processor operates on 32-bits as opposed to the 8-bit operands for the MATRIX and it is a fully pipelined, single-cycle throughput as opposed to the multi-cycle architecture of the VIP.

4.5. FPGA Area

The Xilinx Virtex II Pro FPGA on the XUP board contains 13,696 slices. The SIMD values in Table 5 represent the default value of four units.

	SISD	SISD and SIMD
Number of Slices	2033	7879
% of Slices Used	14%	57%
Minimum Period	20.933 ns	21.222 ns

Table 5: SISD Versus SIMD Pipeline.

The maximum number of units that can be instantiated in the FPGA on the XUP board is eight units which would use the entire FPGA slice area. This allows the processor to complete eight sets of 32-bit data per cycle. Compared to the VIP processor, this SIMD processor is able to fit more SIMD units in the same space than the VIP. The VIP was only able to fit seven units in the Altera FPGA used which had about 2,000 more slices than the Xilinx FPGA Virtex II Pro.

5. Conclusions and Future Work

5.1. Conclusions

The goals of this thesis were met with the construction of a new SIMD-pipelined 32-bit MIPS-based processor. The SIMD processor implemented 39 instructions in a five stage pipeline that used the Harvard memory approach. The instruction set was chosen to provide the core functionality of the processor and completeness. The SIMD architecture was designed with distributed memory so that every memory unit received the same address, allowing the memory to be pictured as row and columns method of access. To achieve multiple data, multiple basic units were added and connected within the architecture and the control logic was updated to account for the different datapaths.

The design was shown that it could operate at the maximum speed of 100 MHz (single-cycle throughput) and utilize all the area of the FPGA. The processor was verified in both simulation and synthesis. This is shown to be an improvement or on the same level as previously designed architectures. This processor performed better with a pipeline design than the VIP which executed one instruction at a time and this SIMD processor is able to fit more SIMD units in the same space than the VIP. Another advantage of this 32-bit SIMD processor over the 8-bit MATRIX is that it can perform a multiplication instruction as well as the arithmetic and logical instruction that the MATRIX can at the same frequency.

5.2. Future Work

With the foundation of the SIMD processor complete, modifications can be made to research image processing applications. The SIMD processor is a good choice to use binary operations to implement morphological operations and could use the multiply and add operations for a MAC to perform convolution and filtering operations. These applications could lead further research into extending this architecture into video processing, where each SIMD unit could operate on a row of image data.

In another interesting area of research, the SIMD processor could be developed into a real-time reconfigurable system. Having a real-time reconfigurable SIMD system would be powerful as the area of the FPGA would be configured to run an application efficiently and then switch configurations to run the next application with the best efficiency specific to the application.

As mentioned in the result section, the speed of the processor could possibly be increased with efficient routing if the user wanted to maintain the full functionality as originally designed. Further research is needed to confirm this idea that the processor could be improved by manually setting the routing paths instead of allowing the Xilinx synthesis to automatically route everything.

A new means of transferring data into memory from I/O would be beneficial after the VHDL has been synthesized into a bit file. This would allow for easy data switching instead of regenerating the memory IP with the new file every time a new program or data set is needed.

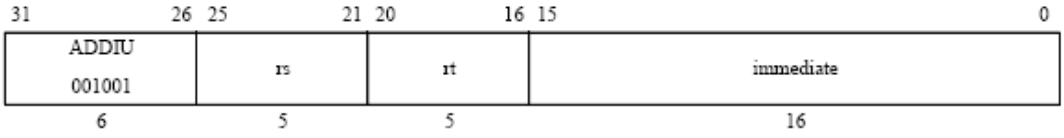
In addition to transferring data external to the processor, internal transfers of data between SIMD units could be beneficial for programs. These data transfers could move data in a nearest neighbor network and could be implemented by adding a move instruction to the instruction set.

Appendices

Appendix A: Instruction Reference

All Instruction descriptions are from [14] MIPS Instruction Set Reference.

Add Immediate Unsigned Word **ADDIU**



Format: `ADDIU rt, rs, immediate` **MIPS32**

Purpose:

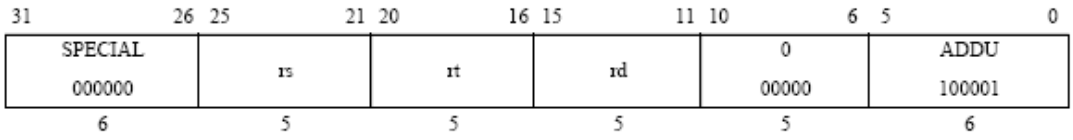
To add a constant to a 32-bit integer

Description: `rt ← rs + immediate`

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Add Unsigned Word **ADDU**



Format: `ADDU rd, rs, rt` **MIPS32**

Purpose:

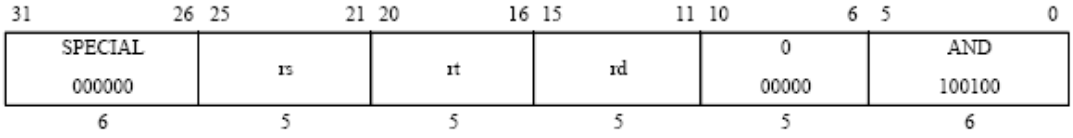
To add 32-bit integers

Description: `rd ← rs + rt`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

And **AND**



Format: `AND rd, rs, rt` **MIPS32**

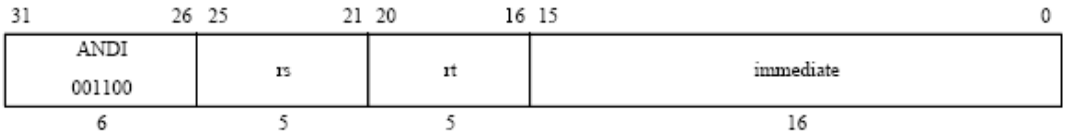
Purpose:

To do a bitwise logical AND

Description: `rd ← rs AND rt`

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

And Immediate — ANDI



Format: `ANDI rt, rs, immediate` — MIPS32

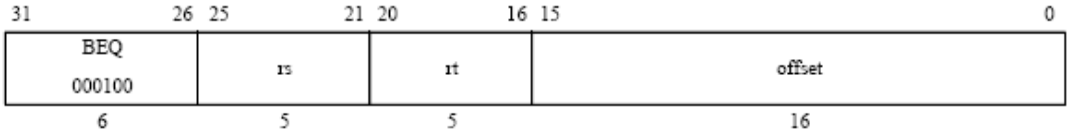
Purpose:

To do a bitwise logical AND with a constant

Description: `rt ← rs AND immediate`

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

Branch on Equal — BEQ



Format: `BEQ rs, rt, offset` — MIPS32

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if rs = rt then branch`

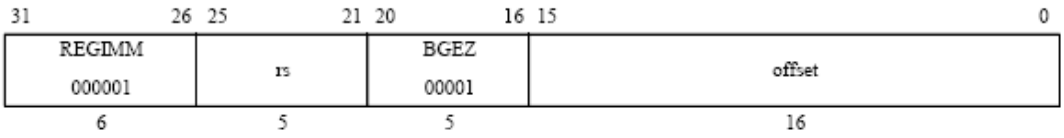
An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Branch on Greater Than or Equal to Zero — BGEZ



Format: `BGEZ rs, offset` MIPS32

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs ≥ 0 then branch`

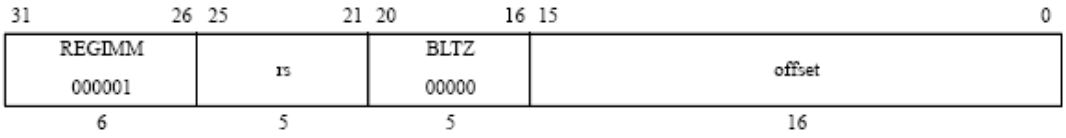
An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Branch on Less Than Zero — BLTZ



Format: `BLTZ rs, offset` MIPS32

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs < 0 then branch`

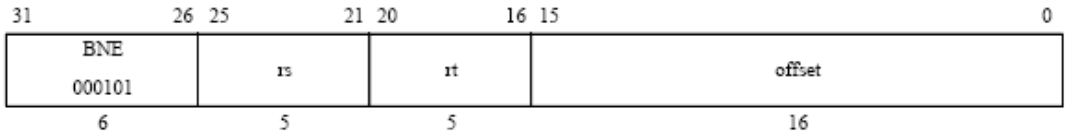
An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Branch on Not Equal — BNE



Format: `BNE rs, rt, offset` — MIPS32

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if rs ≠ rt then branch`

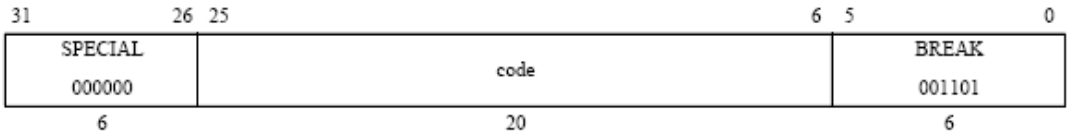
An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Breakpoint — BREAK



Format: `BREAK` — MIPS32

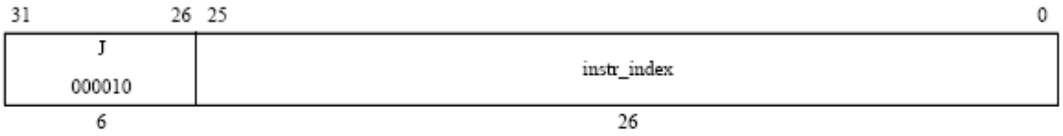
Purpose:

To cause a Breakpoint exception

Description:

A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler. The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Jump J



Format: `J target` MIPS32

Purpose:

To branch within the current 256 MB-aligned region

Description:

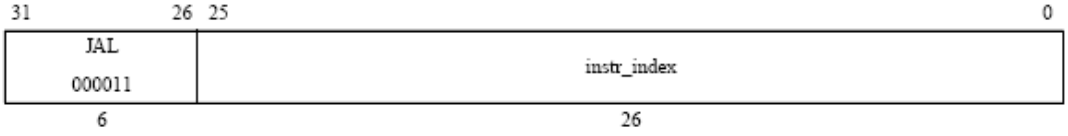
This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay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

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Jump and Link JAL



Format: `JAL target` MIPS32

Purpose:

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

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

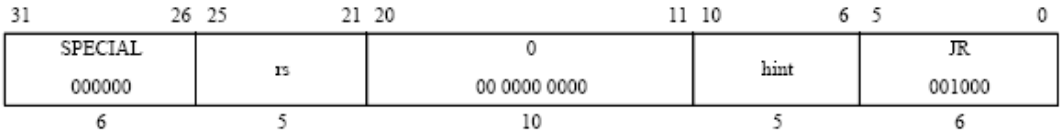
This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay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

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Jump Register — JR



Format: `JR rs` — MIPS32

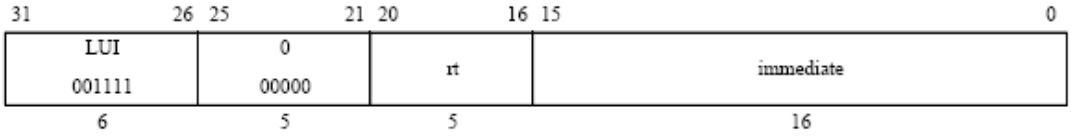
Purpose:

To execute a branch to an instruction address in a register

Description: `PC ← rs`

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

Load Upper Immediate — LUI



Format: `LUI rt, immediate` — MIPS32

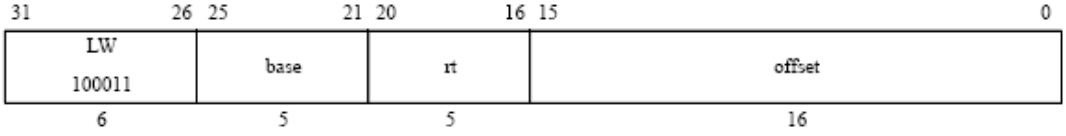
Purpose:

To load a constant into the upper half of a word

Description: `rt ← immediate || 0^16`

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

Load Word — LW



Format: `LW rt, offset(base)` — MIPS32

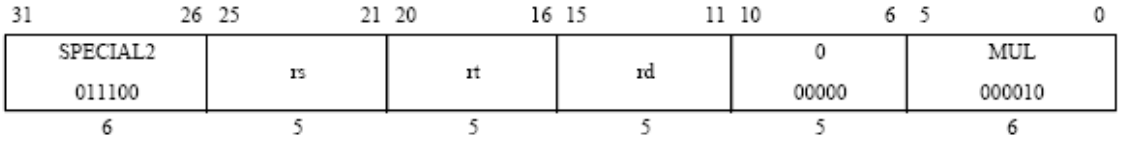
Purpose:

To load a word from memory as a signed value

Description: `rt ← memory[base+offset]`

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

Multiply Word to GPR — MUL

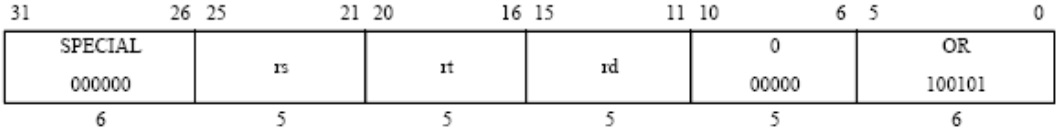


Format: `MUL rd, rs, rt` — MIPS32

Purpose:

To multiply two words and write the result to a GPR.

Or — OR



Format: `OR rd, rs, rt` — MIPS32

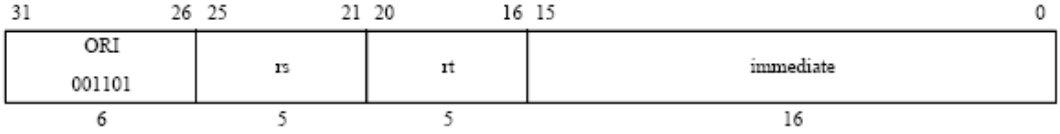
Purpose:

To do a bitwise logical OR

Description: `rd ← rs or rt`

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

Or Immediate — ORI



Format: `ORI rt, rs, immediate` — MIPS32

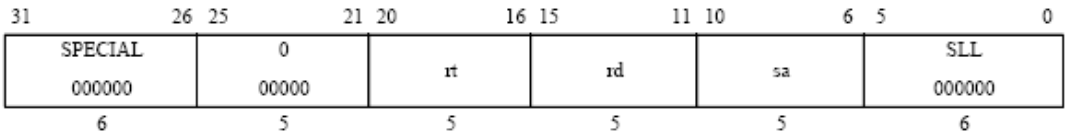
Purpose:

To do a bitwise logical OR with a constant

Description: `rt ← rs or immediate`

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

Shift Word Left Logical — SLL



Format: `SLL rd, rt, sa` — MIPS32

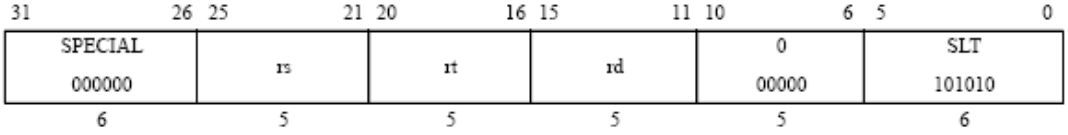
Purpose:

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

Description: `rd ← rt << sa`

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

Set on Less Than — SLT



Format: `SLT rd, rs, rt` — MIPS32

Purpose:

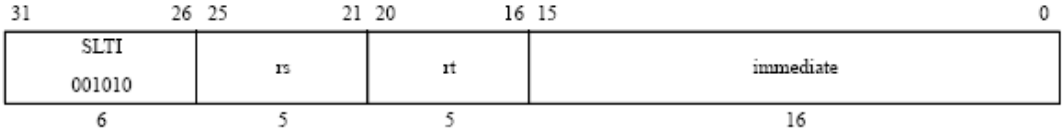
To record the result of a less-than comparison

Description: `rd ← (rs < rt)`

Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Set on Less Than Immediate — SLTI



Format: `SLTI rt, rs, immediate` — MIPS32

Purpose:

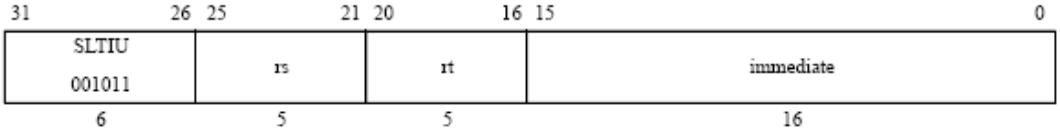
To record the result of a less-than comparison with a constant

Description: `rt ← (rs < immediate)`

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Set on Less Than Immediate Unsigned — SLTIU



Format: `SLTIU rt, rs, immediate` — MIPS32

Purpose:

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

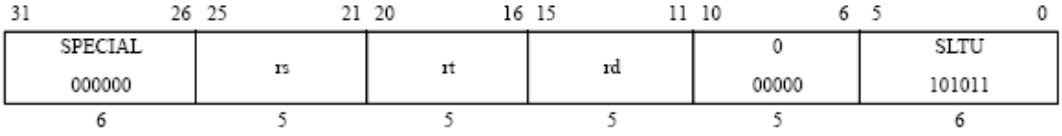
Description: `rt ← (rs < immediate)`

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Set on Less Than Unsigned — SLTU



Format: `SLTU rd, rs, rt` — MIPS32

Purpose:

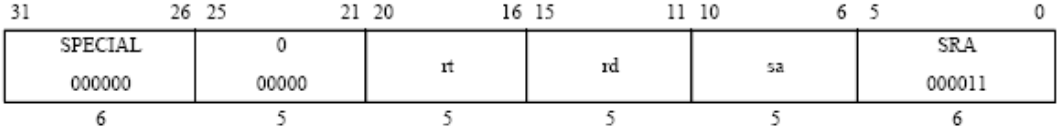
To record the result of an unsigned less-than comparison

Description: `rd ← (rs < rt)`

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Shift Word Right Arithmetic — SRA



Format: `SRA rd, rt, sa` — MIPS32

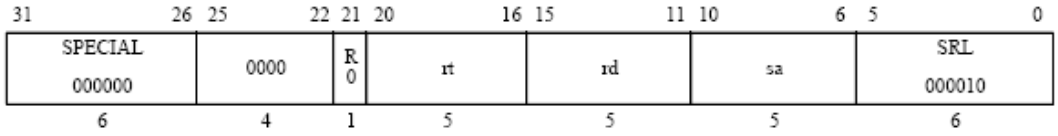
Purpose:

To execute an arithmetic right-shift of a word by a fixed number of bits

Description: `rd ← rt >> sa (arithmetic)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Shift Word Right Logical — SRL



Format: `SRL rd, rt, sa` — MIPS32

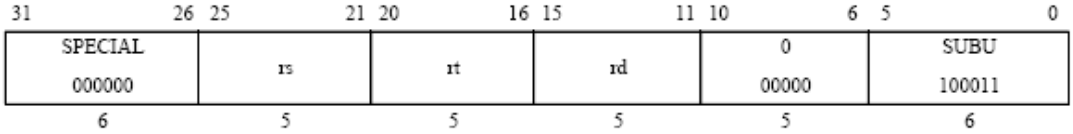
Purpose:

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

Description: `rd ← rt >> sa (logical)`

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

Subtract Unsigned Word — SUBU



Format: `SUBU rd, rs, rt` — MIPS32

Purpose:

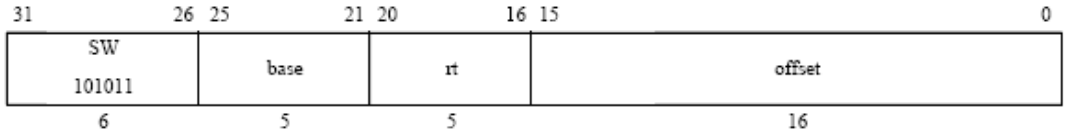
To subtract 32-bit integers

Description: `rd ← rs - rt`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is and placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Store Word — SW



Format: `SW rt, offset(base)` — MIPS32

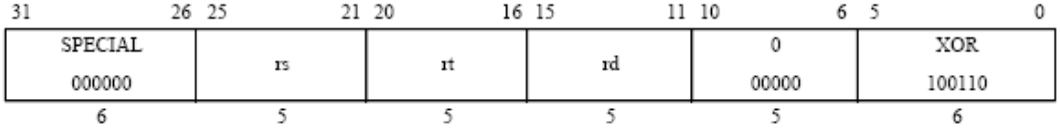
Purpose:

To store a word to memory

Description: `memory[base+offset] ← rt`

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

Exclusive OR — XOR



Format: `XOR rd, rs, rt` MIPS32

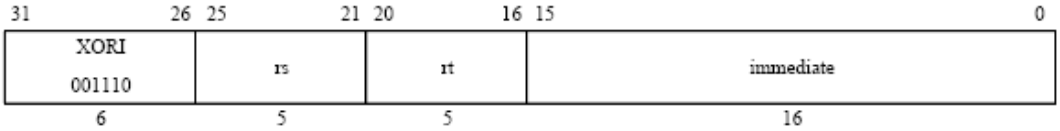
Purpose:

To do a bitwise logical Exclusive OR

Description: `rd ← rs XOR rt`

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

Exclusive OR Immediate — XORI



Format: `XORI rt, rs, immediate` MIPS32

Purpose:

To do a bitwise logical Exclusive OR with a constant

Description: `rt ← rs XOR immediate`

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

Appendix B: VHDL Files

FPGA_TOP.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity FPGA_TOP is
        Port (SYS_CLK : in STD_LOGIC;
              RESET_low : in STD_LOGIC;
              SINGLE_CLK_low : in STD_LOGIC;
              RELEASE_low : in STD_LOGIC;
              DIPSWITCH : in STD_LOGIC_VECTOR (3 downto 0);
              PC_DISPLAY : out STD_LOGIC_VECTOR (15 downto 0);
              INST_STAT_DISPLAY : out STD_LOGIC_VECTOR (31 downto 0);
              STALL : out STD_LOGIC);
end FPGA_TOP;

architecture Behavioral of FPGA_TOP is

component processor is
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          RELEASE : in  STD_LOGIC;
          CURRENT_PC : out STD_LOGIC_VECTOR (31 downto 0);
          CURRENT_INST : out STD_LOGIC_VECTOR (31 downto 0);
          STAT : out STD_LOGIC_VECTOR (7 downto 0);
          STALLED : out  STD_LOGIC);
end component processor;

component edgedetect is
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          DIN : in  STD_LOGIC;
          DOUTSTEPCLK : out STD_LOGIC;
          DOUTBOARDCLK : out  STD_LOGIC);
end component edgedetect;

signal step_clk, processor_clk : std_logic;
signal stalled : std_logic;
signal stat, statout : std_logic_vector(7 downto 0);
signal pc, inst : std_logic_vector(31 downto 0);
signal released releasestep, releaseboard : std_logic;
begin

clkparse: component edgedetect port map(CLK => SYS_CLK,
                                        RESET => '0', --NOT RESET_low,
                                        DIN => NOT SINGLE_CLK_low,
                                        DOUTSTEPCLK => step_clk);

processor_clk <= step_clk when DIPSWITCH(0) = '0' else SYS_CLK;
```

```
releaseparse: component edgedetect port map(CLK => SYS_CLK,
                                            RESET => NOT RESET_low,
                                            DIN => NOT RELEASE_low,
                                            DOUTSTEPCLK => releasestep,
                                            DOUTBOARDCLK => releaseboard);

released <= releasestep when DIPSWITCH(0) = '0' else releaseboard;

proc: component processor port map(CLK => processor_clk,
                                   RESET => NOT RESET_low,
                                   RELEASE => released,
                                   CURRENT_PC => pc,
                                   CURRENT_INST => inst,
                                   STAT => stat,
                                   STALLED => stalled,
                                   BOARDREG23_1 => reg23_1,
                                   BOARDREG23_2 => reg23_2);

STALL <= NOT stalled;
statout <= NOT stat;
PC_DISPLAY <= NOT pc;
INST_STAT_DISPLAY <= x"FFFFFF" & statout when DIPSWITCH(3 downto 2) = "00" else
                     NOT reg23_1 when DIPSWITCH(3 downto 2) = "01" else
                     NOT reg23_2 when DIPSWITCH(3 downto 2) = "10" else
                     NOT inst;

end Behavioral;
```

Processor.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity processor is
        Generic (datasize : natural := 127;
                 size : natural := 4);
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          RELEASE : in  STD_LOGIC;
          CURRENT_PC : out STD_LOGIC_VECTOR (31 downto 0);
          CURRENT_INST : out STD_LOGIC_VECTOR (31 downto 0);
          STAT : out STD_LOGIC_VECTOR (7 downto 0);
          STALLED : out  STD_LOGIC;
          BOARDREG23_1 : out  STD_LOGIC_VECTOR (31 downto 0);
          BOARDREG23_2 : out  STD_LOGIC_VECTOR (31 downto 0));
end processor;

architecture Behavioral of processor is

component FETCHstage is
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          TAKENEXTPC : in  STD_LOGIC;
          BREAKH : in  STD_LOGIC;
```

```
            STALL : in  STD_LOGIC;
            NEXTPC : in  STD_LOGIC_VECTOR (31 downto 0);
            INST : out  STD_LOGIC_VECTOR (31 downto 0);
            CURRENTPC : out  STD_LOGIC_VECTOR (31 downto 0);
            PCPLUS4 : out  STD_LOGIC_VECTOR (31 downto 0));
end component FETCHstage;

component IDstage is
             Generic (datasize : natural := 127;
                      size : natural := 4);
    Port ( CLK : in  STD_LOGIC;
           RESET : in  STD_LOGIC;
           REGWRITEIN : in  STD_LOGIC;
           RELEASE : in  STD_LOGIC;
           REGSELIN : in  STD_LOGIC_VECTOR (4 downto 0);
           PCPLUS4 : in  STD_LOGIC_VECTOR (31 downto 0);
           INST : in  STD_LOGIC_VECTOR (31 downto 0);
           REGDIN : in  STD_LOGIC_VECTOR (31 downto 0);
           A : in  STD_LOGIC_VECTOR (31 downto 0);
           B : in  STD_LOGIC_VECTOR (31 downto 0);
           ISJAL : out  STD_LOGIC;
           TAKENEXTPC : out  STD_LOGIC;
           BREAKH : out  STD_LOGIC;
           REGSELOUT : out  STD_LOGIC_VECTOR (4 downto 0);
           PCPLUS8 : out  STD_LOGIC_VECTOR (31 downto 0);
           RD1 : out  STD_LOGIC_VECTOR (31 downto 0);
           RD2 : out  STD_LOGIC_VECTOR (31 downto 0);
           NEXTPC1 : out  STD_LOGIC_VECTOR (31 downto 0);
           REGB : out  STD_LOGIC_VECTOR (31 downto 0);
           REGWRITEOUT : out  STD_LOGIC;
           MEMTOREG : out  STD_LOGIC;
           MEMWRITEH : out  STD_LOGIC;
           ALUOP : out  STD_LOGIC_VECTOR (4 downto 0);
           --SIMD Ports
           VREGWRITEIN : in  STD_LOGIC;
           VREGDIN : in  STD_LOGIC_VECTOR (datasize downto 0);
           VRD1 : out  STD_LOGIC_VECTOR (datasize downto 0);
           VRD2 : out  STD_LOGIC_VECTOR (datasize downto 0);
           VREGWRITEOUT : out  STD_LOGIC;
           VMEMTOREG : out  STD_LOGIC;
           VMEMWRITEH : out  STD_LOGIC;
           EXTMODE : out STD_LOGIC;
           IMMVALUE : out STD_LOGIC;
           BOARDREG23_1 : out  STD_LOGIC_VECTOR (31 downto 0);
           BOARDREG23_2 : out  STD_LOGIC_VECTOR (31 downto 0));
end component IDstage;

component reg32 is
    Port ( CLK : in  STD_LOGIC;
           ENABLEH : in  STD_LOGIC;
           RESET : in  STD_LOGIC;
           DIN : in  STD_LOGIC_VECTOR (31 downto 0);
           DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component reg32;

component mux32x4 is
```

```
   Port ( SEL : in  STD_LOGIC_VECTOR (2 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          C : in  STD_LOGIC_VECTOR (31 downto 0);
          D : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component mux32x4;

component mux32x3 is
   Port ( SEL : in  STD_LOGIC_VECTOR (1 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          C : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component mux32x3;

component hazardcontrol is
   Port ( MEMTOREGEXE : in  STD_LOGIC;
          REGWRITEEXE : in  STD_LOGIC;
          REGWRITEMEM : in  STD_LOGIC;
          ISJAL : in  STD_LOGIC;
          ISMUL : in STD_LOGIC;
          RSEL1ID : in  STD_LOGIC_VECTOR (4 downto 0);
          RSEL2ID : in  STD_LOGIC_VECTOR (4 downto 0);
          WBEXE : in  STD_LOGIC_VECTOR (4 downto 0);
          WBMEM : in  STD_LOGIC_VECTOR (4 downto 0);
          STALL : out  STD_LOGIC;
          FORWARDA : out  STD_LOGIC_VECTOR (2 downto 0);
          FORWARDM : out  STD_LOGIC_VECTOR (2 downto 0);
          --SIMD Ports
          VMEMTOREGEXE : in  STD_LOGIC;
          VREGWRITEEXE : in  STD_LOGIC;
          VREGWRITEMEM : in  STD_LOGIC;
          VFORWARDA : out  STD_LOGIC_VECTOR (1 downto 0);
          VFORWARDM : out  STD_LOGIC_VECTOR (1 downto 0));
end component hazardcontrol;

component EXEstage is
           Generic (datasize : natural := 127;
                    size : natural := 4);
   Port ( CLK : in STD_LOGIC;
          ALUop : in  STD_LOGIC_VECTOR (4 downto 0);
          SHAMT : in  STD_LOGIC_VECTOR (4 downto 0);
          FUNCT : in  STD_LOGIC_VECTOR (5 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          MULRESULT : out STD_LOGIC_VECTOR (31 downto 0);
          RESULT : out  STD_LOGIC_VECTOR (31 downto 0);
          --SIMD Ports
          VA : in  STD_LOGIC_VECTOR (datasize downto 0);
          VB : in  STD_LOGIC_VECTOR (datasize downto 0);
          VMULRESULT : out  STD_LOGIC_VECTOR (datasize downto 0);
          VRESULT : out  STD_LOGIC_VECTOR (datasize downto 0));
end component EXEstage;

component MEMstage is
```

```
            Generic (datasize : natural := 127;
                     size : natural := 4);
    Port ( CLK : in  STD_LOGIC;
           RESET : in  STD_LOGIC;
           ADDR : in  STD_LOGIC_VECTOR (31 downto 0);
           DIN : in  STD_LOGIC_VECTOR (31 downto 0);
           MEMTOREG : in  STD_LOGIC;
           MEMWRITEH : in  STD_LOGIC;
           DOUT : out  STD_LOGIC_VECTOR (31 downto 0);
           --SIMD Ports
           VADDR : in STD_LOGIC_VECTOR (31 downto 0);
           VDIN : in STD_LOGIC_VECTOR (datasize downto 0);
           VMEMTOREG : in STD_LOGIC;
          VMEMWRITEH : in STD_LOGIC;
          VDOUT : out STD_LOGIC_VECTOR (datasize downto 0));
end component MEMstage;

component mux32x2 is
    Port ( SEL : in  STD_LOGIC;
             A : in  STD_LOGIC_VECTOR (31 downto 0);
             B : in  STD_LOGIC_VECTOR (31 downto 0);
             DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component mux32x2;

signal takenextpcdelay : std_logic;
signal nextpc : std_logic_vector(31 downto 0);
signal breakh : std_logic;
signal inst : std_logic_vector(31 downto 0);
signal pcplus4 : std_logic_vector(31 downto 0);
signal stall : std_logic;

signal regdin : std_logic_vector(31 downto 0);
signal wbID, wbEXE, wbMEM : std_logic_vector(4 downto 0);
signal memtoregID, memtoregEXE, memtoregMEM : std_logic;
signal aluopID, aluopEXE : std_logic_vector(3 downto 0);
signal wehID, wehEXE, wehMEM : std_logic;
signal regadin, regbdin, regmdin : std_logic_vector(31 downto 0);
signal memwriteID, memwriteMEM : std_logic;
signal shamtEXE : std_logic_vector(4 downto 0);
signal functEXE : std_logic_vector(5 downto 0);
signal pcplus8 : std_logic_vector(31 downto 0);
signal rd1, rd2 : std_logic_vector(31 downto 0);
signal nextpcdelay : std_logic_vector(31 downto 0);
signal takenextpc : std_logic;
signal isjal : std_logic;

signal forwardA, forwardM : std_logic_vector(2 downto 0);
signal resultEXE, resultMEM : std_logic_vector(31 downto 0);
signal regAdout, regBdout, regMdout : std_logic_vector(31 downto 0);

signal memdout : std_logic_vector(31 downto 0);
signal regdintemp : std_logic_vector(31 downto 0);

signal resetorstall, notbreaken : std_logic;

signal breakhtemp :std_logic;
```

```
signal takemulresult : std_logic;
signal mulresult : std_logic_vector(31 downto 0);
signal datawritebacksel : std_logic_vector(1 downto 0);

--SIMD signals
type data_array is array (1 to size) of STD_LOGIC_VECTOR (31 downto 0);
signal vrd1_array, vrd2_array, regdatain_array : data_array;
signal vregadin, vregbdin, vregmdin : data_array;
signal vregAdout, vregBdout, vregMdout : data_array;

signal vrd1, vrd2 : std_logic_vector (datasize downto 0);

signal vforwardA, vforwardM : std_logic_vector(1 downto 0);

signal vwehID, vwehEXE, vwehMEM : std_logic;
signal vmemtoregID, vmemtoregEXE, vmemtoregMEM : std_logic;
signal vmemwriteID, vmemwriteMEM : std_logic;

constant mask0: std_logic_vector(15 downto 0):= x"0000";
constant mask1: std_logic_vector(15 downto 0):= x"FFFF";
signal extimm : std_logic_vector(31 downto 0);
signal extmode, immvalue : std_logic;

signal va, vb, vm, vresult, vmultresult : std_logic_vector (datasize downto 0);
signal vresultEXE_array, vresultMEM_array, vmultresult_array : data_array;

signal vmemdout, vregdin : std_logic_vector (datasize downto 0);
signal vmemdout_array, vregdin_array, vregdintemp_array : data_array;
signal vdatawritebacksel : std_logic_vector(1 downto 0);


begin

STALLED <= stall;
CURRENT_INST <= inst;

process (breakhtemp, inst) is
   begin
      if (inst(31 downto 26) = "000000" AND inst(5 downto 0) = "001101") then
         breakhtemp <= '1';
      else breakhtemp <= '0';
   end if;
      if breakhtemp = '1' then
         STAT(6 downto 0) <= inst(12 downto 6);
      else STAT(6 downto 0) <= "0000000";
   end if;
   STAT(7) <= breakhtemp;
end process;

fetch: component FETCHstage port map(CLK => CLK,
                                     RESET => RESET,
                                     TAKENEXTPC => takenextpcdelay,
                                     BREAKH => breakh,
                                     STALL => stall,
                                     NEXTPC => nextpc,
```

```
                                        INST => inst,
                                        CURRENTPC => CURRENT_PC,
                                        PCPLUS4 => pcplus4);

id: component IDstage port map(CLK => CLK,
                                        RESET => RESET,
                                        RELEASE => RELEASE,
                                        REGWRITEIN => wehMEM,
                                        REGSELIN => wbMEM,
                                        PCPLUS4 => pcplus4,
                                        INST => inst,
                                        REGDIN => regdin,
                                        A => regadin,
                                        B => regmdin,
                                        ISJAL => isjal,
                                        TAKENEXTPC => takenextpc,
                                        BREAKH => breakh,
                                        REGSELOUT => wbID,
                                        PCPLUS8 => pcplus8,
                                        RD1 => rd1,
                                        RD2 => rd2,
                                        NEXTPC1 => nextpcdelay,
                                        REGB => regbdin,
                                        REGWRITEOUT => wehID,
                                        MEMTOREG => memtoregID,
                                        MEMWRITEH => memwriteID,
                                        ALUOP => aluopID,
                                        --SIMD Ports
                                        VREGWRITEIN => vwehMEM,
                                        VREGDIN => vregdin,
                                        VRD1 => vrd1,
                                        VRD2 => vrd2,
                                        VREGWRITEOUT => vwehID,
                                        VMEMTOREG => vmemtoregID,
                                        VMEMWRITEH => vmemwriteID,
                                        EXTMODE => extmode,
                                        IMMVALUE => immvalue);

vrd1_array(4) <= vrd1(31 downto 0);
vrd1_array(3) <= vrd1(63 downto 32);
vrd1_array(2) <= vrd1(95 downto 64);
vrd1_array(1) <= vrd1(127 downto 96);
vrd2_array(4) <= vrd2(31 downto 0);
vrd2_array(3) <= vrd2(63 downto 32);
vrd2_array(2) <= vrd2(95 downto 64);
vrd2_array(1) <= vrd2(127 downto 96);

--moved out B logic from ID stage for SIMD
extimm <= mask1 & inst(15 downto 0) when extmode = '1' and inst(15) = '1' else
                                mask0 & inst(15 downto 0);

genregBconnection: for I in 1 to size generate

vregbdin(I) <= extimm when immvalue = '1' else vregmdin(I);

end generate;
```

```
----------

resetorstall <= RESET or stall;
notbreaken <= not breakh;

nextpcreg: component reg32 port map(CLK => CLK,
                                    ENABLEH => notbreaken,
                                    RESET => resetorstall,
                                    DIN => nextpcdelay,
                                    DOUT => nextpc);

process (CLK) is
        begin
                if rising_edge(CLK) then
                        if RESET = '1' or stall = '1' then
                                takenextpcdelay <= '0';
                        elsif breakh = '0' then
                                takenextpcdelay <= takenextpc;
                        else
                                takenextpcdelay <= takenextpcdelay;
                end if;
        end if;
end process;


forwardmuxA: component mux32x4 port map(SEL => forwardA,
                                        A => rd1,
                                        B => pcplus8,
                                        C => resultEXE,
                                        D => regdin,
                                        DOUT => regadin);


Areg: component reg32 port map(CLK => CLK,
                               ENABLEH => '1',
                               RESET => RESET,
                               DIN => regadin,
                               DOUT => regAdout);


Breg: component reg32 port map(CLK => CLK,
                               ENABLEH => '1',
                               RESET => RESET,
                               DIN => regbdin,
                               DOUT => regBdout);

forwardmuxM: component mux32x4 port map(SEL => forwardM,
                                        A => rd2,
                                        B => X"00000000",
                                        C => resultEXE,
                                        D => regdin,
                                        DOUT => regmdin);


Mreg: component reg32 port map(CLK => CLK,
                               ENABLEH => '1',
```

```
                                        RESET => resetorstall,
                                        DIN => regmdin,
                                        DOUT => regMdout);

genmuxandreg: for I in 1 to size generate
SIMDmuxA: component mux32x3 port map(SEL => vforwardA,
                                              A => vrd1_array(I),
                                              B => vresultEXE_array(I),
                                              C => vregdin_array(I),
                                              DOUT => vregadin(I));


SIMDmuxM: component mux32x3 port map(SEL => vforwardM,
                                               A => vrd2_array(I),
                                              B => vresultEXE_array(I),
                                              C => vregdin_array(I),
                                              DOUT => vregmdin(I));

SIMDAreg: component reg32 port map(CLK => CLK,
                                             ENABLEH => '1',
                                             RESET => RESET,
                                             DIN => vregadin(I),
                                             DOUT => vregAdout(I));

SIMDBreg: component reg32 port map(CLK => CLK,
                                             ENABLEH => '1',
                                             RESET => RESET,
                                             DIN => vregbdin(I),
                                             DOUT => vregBdout(I));

SIMDMreg: component reg32 port map(CLK => CLK,
                                             ENABLEH => '1',
                                             RESET => resetorstall,
                                             DIN => vregmdin(I),
                                             DOUT => vregMdout(I));

end generate;

va <= vregAdout(1) & vregAdout(2) & vregAdout(3) & vregAdout(4);
vb <= vregBdout(1) & vregBdout(2) & vregBdout(3) & vregBdout(4);
vm <= vregMdout(1) & vregMdout(2) & vregMdout(3) & vregMdout(4);

process (CLK) is
        begin
                if rising_edge(CLK) then
                        if RESET = '1' or stall = '1' then

                                        aluopEXE <= "00000";
                                        wbEXE <= "00000";
                                        wehEXE <= '0';
                                        memtoregEXE <= '0';
                                        memwriteMEM <= '0';
                                        shamtEXE <= "00000";
                                        functEXE <= "000000";
                                        vwehEXE <= '0';
                                        vmemtoregEXE <= '0';
```

```
                        vmemwriteMEM <= '0';
                    else
                        aluopEXE <= aluopID;
                        wbEXE <= wbID;
                        wehEXE <= wehID;
                        memtoregEXE <= memtoregID;
                        memwriteMEM <= memwriteID;
                        shamtEXE <= inst(10 downto 6);
                        functEXE <= inst(5 downto 0);
                        vwehEXE <= vwehID;
                        vmemtoregEXE <= vmemtoregID;
                        vmemwriteMEM <= vmemwriteID;
                end if;
        end if;
end process;

hazardcntrl: component hazardcontrol port map(MEMTOREGEXE => memtoregEXE,
                                              REGWRITEEXE => wehEXE,
                                              REGWRITEMEM => wehMEM,
                                              ISJAL => isjal,
                                              ISMUL => aluopEXE(3),
                                              RSEL1ID => inst(25 downto 21),
                                              RSEL2ID => inst(20 downto 16),
                                             WBEXE => wbEXE,
                                             WBMEM => wbMEM,
                                              STALL => stall,
                                              FORWARDA => forwardA,
                                              FORWARDM => forwardM,
                                              --SIMD Ports
                                             VMEMTOREGEXE => vmemtoregEXE,
                                             VREGWRITEEXE => vwehEXE,
                                             VREGWRITEMEM => vwehMEM,
                                             VFORWARDA => vforwardA,
                                             VFORWARDM => vforwardM);

EXE: component EXEstage port map(CLK => CLK,
                                 ALUop => aluopEXE,
                                 SHAMT => shamtEXE,
                                 FUNCT => functEXE,
                                 A => regAdout,
                                 B => regBdout,
                                 MULRESULT => mulresult,
                                 RESULT => resultEXE,
                                 --SIMD Ports
                                VA => va,
                                VB => vb,
                                VMULRESULT => vmultresult,
                                VRESULT => vresult);


vresultEXE_array(4) <= vresult(31 downto 0);
vresultEXE_array(3) <= vresult(63 downto 32);
vresultEXE_array(2) <= vresult(95 downto 64);
vresultEXE_array(1) <= vresult(127 downto 96);

vmultresult_array(4) <= vmultresult(31 downto 0);
```

```
vmultresult_array(3) <= vmultresult(63 downto 32);
vmultresult_array(2) <= vmultresult(95 downto 64);
vmultresult_array(1) <= vmultresult(127 downto 96);

resultreg: component reg32 port map(CLK => CLK,
                                    ENABLEH => '1',
                                    RESET => RESET,
                                    DIN => resultEXE,
                                   DOUT => resultMEM);

genresultreg: for I in 1 to size generate
SIMDresultreg: component reg32 port map(CLK => CLK,
                                        ENABLEH => '1',
                                        RESET => RESET,
                                        DIN => vresultEXE_array(I),
                                        DOUT => vresultMEM_array(I));
end generate;

process (CLK) is
        begin
                if rising_edge(CLK) then
                        if RESET = '1' then

                                wbMEM <= "00000";
                                wehMEM <= '0';
                                memtoregMEM <= '0';
                                takemulresult <= '0';
                                vwehMEM <= '0';
                                vmemtoregMEM <= '0';
                        else
                                wbMEM <= wbEXE;
                                wehMEM <= wehEXE;
                                memtoregMEM <= memtoregEXE;
                                takemulresult <= aluopEXE(3);
                                vwehMEM <= vwehEXE;
                                vmemtoregMEM <= vmemtoregEXE;
                end if;
        end if;
end process;

MEM: component MEMstage port map(CLK => CLK,
                                 RESET => RESET,
                                 ADDR => resultEXE,
                                 DIN => regMdout,
                                 MEMTOREG => memtoregEXE,
                                 MEMWRITEH => memwriteMEM,
                                 DOUT => memdout,
                                 --SIMD Ports
                                 VADDR => vresult(127 downto 96),
                                 VDIN => vm,
                                 VMEMTOREG => vmemtoregEXE,
                                 VMEMWRITEH => vmemwriteMEM,
                                 VDOUT => vmemdout);

vmemdout_array(4) <= vmemdout(31 downto 0);
vmemdout_array(3) <= vmemdout(63 downto 32);
```

```
vmemdout_array(2) <= vmemdout(95 downto 64);
vmemdout_array(1) <= vmemdout(127 downto 96);

datawritebacksel <= takemulresult & memtoregMEM;
vdatawritebacksel <= takemulresult & vmemtoregMEM;

memoryresult: component mux32x3 port map(SEL => datawritebacksel,
                                          A => resultMEM,
                                          B => memdout,
                                          C => mulresult,
                                          DOUT => regdintemp);

genmemoryreg: for I in 1 to size generate
SIMDmemoryresult: component mux32x3 port map(SEL => datawritebacksel,
                                             A => vresultMEM_array(I),
                                             B => vmemdout_array(I),
                                             C => vmultresult_array(I),
                                             DOUT => vregdintemp_array(I));


vregdin_array(I) <= X"00000000" when wbMEM = "00000" else  vregdintemp_array(I);

end generate;

vregdin <= vregdin_array(1) & vregdin_array(2) & vregdin_array(3) & vregdin_array(4);

regdin <= X"00000000" when wbMEM = "00000" else regdintemp;

end Behavioral;
```

Fetchstage.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FETCHstage is
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          TAKENEXTPC : in  STD_LOGIC;
          BREAKH : in  STD_LOGIC;
          STALL : in  STD_LOGIC;
          NEXTPC : in  STD_LOGIC_VECTOR (31 downto 0);
          INST : out  STD_LOGIC_VECTOR (31 downto 0);
          CURRENTPC : out  STD_LOGIC_VECTOR (31 downto 0);
          PCPLUS4 : out  STD_LOGIC_VECTOR (31 downto 0));
end FETCHstage;

architecture Behavioral of FETCHstage is

component reg32 is
   Port ( CLK : in  STD_LOGIC;
          ENABLEH : in  STD_LOGIC;
```

```
        RESET : in  STD_LOGIC;
        DIN : in  STD_LOGIC_VECTOR (31 downto 0);
        DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component reg32;

component instmem is
        port (
        addr: in std_logic_vector(9 downto 0);
        clk: in std_logic;
        dout: out std_logic_vector(31 downto 0));
end component instmem;

constant enablepc : std_logic := '1';
signal instmemaddr : std_logic_vector(31 downto 0);
signal pc, pc4 : std_logic_vector(31 downto 0);

begin

CURRENTPC <= pc;

pcreg: component reg32 port map(CLK => CLK,
                                ENABLEH => enablepc,
                                RESET => RESET,
                                DIN => instmemaddr,
                                DOUT => pc);

pc4 <= pc + 4;

instmemaddr <= X"00000000" when RESET = '1' else
                pc when BREAKH = '1' or STALL = '1' else
                NEXTPC when TAKENEXTPC = '1' else
                pc4;

PCPLUS4 <= pc4;

instructionmemory: component instmem port map(addr => instmemaddr(11 downto 2),
                                              clk => CLK,
                                              dout => INST);

end Behavioral;
```

Instmem.vhd:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- synopsys translate_off
Library XilinxCoreLib;
-- synopsys translate_on
ENTITY instmem IS
        port (
        addr: IN std_logic_VECTOR(9 downto 0);
        clk: IN std_logic;
        dout: OUT std_logic_VECTOR(31 downto 0));
END instmem;
```

```
ARCHITECTURE instmem_a OF instmem IS
-- synopsys translate_off
component wrapped_instmem
	port (
	addr: IN std_logic_VECTOR(9 downto 0);
	clk: IN std_logic;
	dout: OUT std_logic_VECTOR(31 downto 0));
end component;

-- Configuration specification
	for all : wrapped_instmem use entity XilinxCoreLib.blkmemsp_v6_2(behavioral)
		generic map(
			c_sinit_value => "0",
			c_has_en => 0,
			c_reg_inputs => 0,
			c_yclk_is_rising => 1,
			c_ysinit_is_high => 1,
			c_ywe_is_high => 1,
			c_yprimitive_type => "16kx1",
			c_ytop_addr => "1024",
			c_yhierarchy => "hierarchy1",
			c_has_limit_data_pitch => 0,
			c_has_rdy => 0,
			c_write_mode => 0,
			c_width => 32,
			c_yuse_single_primitive => 0,
			c_has_nd => 0,
			c_has_we => 0,
			c_enable_rlocs => 0,
			c_has_rfd => 0,
			c_has_din => 0,
			c_ybottom_addr => "0",
			c_pipe_stages => 0,
			c_yen_is_high => 1,
			c_depth => 1024,
			c_has_default_data => 0,
			c_limit_data_pitch => 18,
			c_has_sinit => 0,
			c_mem_init_file => "instmem.mif",
			c_yydisable_warnings => 1,
			c_default_data => "0",
			c_ymake_bmm => 0,
			c_addr_width => 10);
-- synopsys translate_on
BEGIN
-- synopsys translate_off
U0 : wrapped_instmem
		port map (
			addr => addr,
			clk => clk,
			dout => dout);
-- synopsys translate_on

END instmem_a;
```

IDstage.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity IDstage is
            Generic (datasize : natural := 127;
                     size : natural := 4);
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          REGWRITEIN : in  STD_LOGIC;
          RELEASE : in  STD_LOGIC;
          REGSELIN : in  STD_LOGIC_VECTOR (4 downto 0);
          PCPLUS4 : in  STD_LOGIC_VECTOR (31 downto 0);
          INST : in  STD_LOGIC_VECTOR (31 downto 0);
          REGDIN : in  STD_LOGIC_VECTOR (31 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          ISJAL : out  STD_LOGIC;
          TAKENEXTPC : out  STD_LOGIC;
          BREAKH : out  STD_LOGIC;
          REGSELOUT : out  STD_LOGIC_VECTOR (4 downto 0);
          PCPLUS8 : out  STD_LOGIC_VECTOR (31 downto 0);
          RD1 : out  STD_LOGIC_VECTOR (31 downto 0);
          RD2 : out  STD_LOGIC_VECTOR (31 downto 0);
          NEXTPC1 : out  STD_LOGIC_VECTOR (31 downto 0);
          REGB : out  STD_LOGIC_VECTOR (31 downto 0);
          REGWRITEOUT : out  STD_LOGIC;
          MEMTOREG : out  STD_LOGIC;
          MEMWRITEH : out  STD_LOGIC;
          ALUOP : out  STD_LOGIC_VECTOR (3 downto 0);
          --SIMD Ports
          VREGWRITEIN : in  STD_LOGIC;
          VREGDIN : in  STD_LOGIC_VECTOR (datasize downto 0);
          VRD1 : out  STD_LOGIC_VECTOR (datasize downto 0);
          VRD2 : out  STD_LOGIC_VECTOR (datasize downto 0);
          VREGWRITEOUT : out  STD_LOGIC;
          VMEMTOREG : out  STD_LOGIC;
          VMEMWRITEH : out  STD_LOGIC;
          EXTMODE : out STD_LOGIC;
          IMMVALUE : out STD_LOGIC;
          BOARDREG23_1 : out  STD_LOGIC_VECTOR (31 downto 0);
          BOARDREG23_2 : out  STD_LOGIC_VECTOR (31 downto 0));
end IDstage;

architecture Behavioral of IDstage is

component regfile is
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          WEH : in  STD_LOGIC;
          DIN : in  STD_LOGIC_VECTOR (31 downto 0);
          WSEL : in  STD_LOGIC_VECTOR (4 downto 0);
          RSEL1 : in  STD_LOGIC_VECTOR (4 downto 0);
```

```
            RSEL2 : in  STD_LOGIC_VECTOR (4 downto 0);
            RD1 : out  STD_LOGIC_VECTOR (31 downto 0);
            RD2 : out  STD_LOGIC_VECTOR (31 downto 0);
            REG23OUT : out STD_LOGIC_VECTOR (31 downto 0));  --Board IO port
end component;

component nextPC is
   Port ( ISBJ : in  STD_LOGIC_VECTOR (2 downto 0);
           INSTINDEX : in  STD_LOGIC_VECTOR (25 downto 0);
           PCPLUS4 : in  STD_LOGIC_VECTOR (31 downto 0);
           RD1 : in  STD_LOGIC_VECTOR (31 downto 0);
           EXTIMM : in  STD_LOGIC_VECTOR (31 downto 0);
           NEXTPC : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component branchlogic is
   Port ( ISBGEZ : in  STD_LOGIC;
          OPCODE : in  STD_LOGIC_VECTOR (5 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          BRANCHOK : out  STD_LOGIC);
end component;

component controller is
   Port ( OPCODE : in  STD_LOGIC_VECTOR (5 downto 0);
           FUNCT : in  STD_LOGIC_VECTOR (5 downto 0);
           REGWRITEH : out  STD_LOGIC;
           MEMWRITEH : out  STD_LOGIC;
           REGDST : out  STD_LOGIC;
           MEMTOREG : out  STD_LOGIC;
           ALUSRC : out  STD_LOGIC;
           EXTCTRL : out  STD_LOGIC;
           BREAKH : out  STD_LOGIC;
           ISJAL : out  STD_LOGIC;
           ISBJ : out  STD_LOGIC_VECTOR (2 downto 0);
          ALUOP : out  STD_LOGIC_VECTOR (4 downto 0);
           -- SIMD Ports
           VREGWRITEH : out STD_LOGIC;
           VMEMWRITEH : out STD_LOGIC;
           VMEMTOREG : out STD_LOGIC);
end component;

signal extctrl, alusrc, branchok, regdst, breakhtemp, isjaltemp: std_logic;
signal isbj: std_logic_vector(2 downto 0);
signal extimm, rd1temp: std_logic_vector(31 downto 0);
constant mask0: std_logic_vector(15 downto 0):= x"0000";
constant mask1: std_logic_vector(15 downto 0):= x"FFFF";

--SIMD signals
type data_array is array (1 to size) of STD_LOGIC_VECTOR (31 downto 0);
signal vrd1_array, vrd2_array, regdatain_array, reg23_array : data_array;

begin
control: component controller port map(OPCODE => INST(31 downto 26),
                                       FUNCT => INST(5 downto 0),
                                       REGWRITEH => REGWRITEOUT,
```

```
                                        MEMWRITEH => MEMWRITEH,
                                        REGDST => regdst,
                                        MEMTOREG => MEMTOREG,
                                        ALUSRC => alusrc,
                                        EXTCTRL => extctrl,
                                        BREAKH => breakhtemp,
                                        ISJAL => isjaltemp,
                                        ISBJ => isbj,
                                       ALUOP => ALUOP,
                                       VREGWRITEH => VREGWRITEOUT,
                                       VMEMWRITEH => VMEMWRITEH,
                                       VMEMTOREG => VMEMTOREG);

registerfile: component regfile port map(CLK => CLK,
                                         RESET => RESET,
                                         WEH => REGWRITEIN,
                                         DIN => REGDIN,
                                         WSEL => REGSELIN,
                                        RSEL1 => INST(25 downto 21),
                                        RSEL2 => INST(20 downto 16),
                                        RD1 => rd1temp,
                                        RD2 => RD2);


regdatain_array(4) <= VREGDIN(31 downto 0);
regdatain_array(3) <= VREGDIN(63 downto 32);
regdatain_array(2) <= VREGDIN(95 downto 64);
regdatain_array(1) <= VREGDIN(127 downto 96);

genregfiles: for I in 1 to size generate

SIMDregfile: component regfile port map(CLK => CLK,
                                        RESET => RESET,
                                        WEH => VREGWRITEIN,
                                        DIN => regdatain_array(I),
                                        WSEL => REGSELIN,
                                        RSEL1 => INST(25 downto 21),
                                        RSEL2 => INST(20 downto 16),
                                        RD1 => vrd1_array(I),
                                        RD2 => vrd2_array(I)),
                                        REG23OUT => reg23_array(I));
end generate;

BOARDREG23_1 <= reg23_array(1);
BOARDREG23_2 <= reg23_array(2);

VRD1 <= vrd1_array(1) & vrd1_array(2) & vrd1_array(3) & vrd1_array(4);
VRD2 <= vrd2_array(1) & vrd2_array(2) & vrd2_array(3) & vrd2_array(4);


branchlgc: component branchlogic port map(ISBGEZ => INST(16),
                                          OPCODE => INST(31 downto 26),
                                          A => A,
                                          B => B,
                                          BRANCHOK => branchok);
```

```
EXTMODE <= extctrl; -- for SIMD, prevents porting in the outputs of SIMD MUXM
IMMVALUE <= alusrc; -- for SIMD, prevents porting in the outputs of SIMD MUXM

PCPLUS8 <= PCPLUS4 + x"00000004";
extimm <= mask1 & INST(15 downto 0) when extctrl = '1' and INST(15) = '1' else
          mask0 & INST(15 downto 0);

REGB <= extimm when alusrc = '1' else B;
TAKENEXTPC <= branchok OR isbj(2) OR isbj(1);
BREAKH <= breakhtemp AND (NOT RELEASE);

process (regdst, isjaltemp, INST) is
   begin
     if (isjaltemp = '1') then
     REGSELOUT <= "11111";
     elsif regdst = '1' then
     REGSELOUT <= INST(15 downto 11);
     else
     REGSELOUT <= INST(20 downto 16);
     end if;
end process;

ISJAL <= isjaltemp;
RD1 <= rd1temp;

nextpcmodule: component nextpc port map(ISBJ => isbj,
                                        INSTINDEX => INST(25 downto 0),
                                        PCPLUS4 => PCPLUS4,
                                        RD1 => A,
                                        EXTIMM => extimm,
                                        NEXTPC => NEXTPC1);

end Behavioral;
```

Controller.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity controller is
   Port ( OPCODE : in  STD_LOGIC_VECTOR (5 downto 0);
          FUNCT : in  STD_LOGIC_VECTOR (5 downto 0);
          REGWRITEH : out  STD_LOGIC;
          MEMWRITEH : out  STD_LOGIC;
          REGDST : out  STD_LOGIC;
          MEMTOREG : out  STD_LOGIC;
          ALUSRC : out  STD_LOGIC;
          EXTCTRL : out  STD_LOGIC;
          BREAKH : out  STD_LOGIC;
          ISJAL : out  STD_LOGIC;
          ISBJ : out  STD_LOGIC_VECTOR (2 downto 0);
          ALUOP : out  STD_LOGIC_VECTOR (4 downto 0);
          -- SIMD Ports
```

```
        VREGWRITEH : out STD_LOGIC;
        VMEMWRITEH : out STD_LOGIC;
        VMEMTOREG : out STD_LOGIC);
end controller;

architecture Behavioral of controller is

constant rtype: std_logic_vector (5 downto 0):= "000000";
constant addiu: std_logic_vector (5 downto 0):= "001001";
constant slti: std_logic_vector (5 downto 0):= "001010";
constant sltiu: std_logic_vector (5 downto 0):= "001011";
constant andi: std_logic_vector (5 downto 0):= "001100";
constant ori: std_logic_vector (5 downto 0):= "001101";
constant xori: std_logic_vector (5 downto 0):= "001110";
constant lui: std_logic_vector (5 downto 0):= "001111";
constant lw: std_logic_vector (5 downto 0):= "100011";
constant sw: std_logic_vector (5 downto 0):= "101011";
constant beq_bne: std_logic_vector (4 downto 0):= "00010";
constant bgez_bltz: std_logic_vector (5 downto 0):= "000001";
constant j: std_logic_vector (5 downto 0):= "000010";
constant jal: std_logic_vector (5 downto 0):= "000011";
constant mul: std_logic_vector (5 downto 0):= "011100";
--SIMD opcodes
constant vlw: std_logic_vector (5 downto 0):= "110011";
constant vsw: std_logic_vector (5 downto 0):= "111011";
constant vaddiu: std_logic_vector (5 downto 0):= "011001";
constant vandi: std_logic_vector (5 downto 0):= "011111";
constant vori: std_logic_vector (5 downto 0):= "011101";
constant vxori: std_logic_vector (5 downto 0):= "011110";
constant vrtype: std_logic_vector (5 downto 0):= "010000";
constant vmul: std_logic_vector (5 downto 0):= "111100";

signal jump, jtype, branch: std_logic;

begin

jump <= '1' when OPCODE = rtype AND FUNCT = "001000" else '0';
jtype <= '1' when OPCODE(5 downto 1) = "00001" else '0';
branch <= '1' when OPCODE(5 downto 1) = beq_bne OR OPCODE = bgez_bltz else '0';
ISBJ <= jtype & jump & branch;
BREAKH <= '1' when OPCODE = rtype AND FUNCT = "001101" else '0';
ISJAL <= '1' when OPCODE = jal else '0';

REGWRITEH <= '1' when OPCODE = rtype AND FUNCT /= "001101" else
                    '1' when OPCODE(5 downto 3) = "001" AND OPCODE /= "001000" else
                    '1' when OPCODE = lw OR OPCODE = jal OR OPCODE = mul else
                    '0';

MEMWRITEH <= '1' when OPCODE = sw else '0';

MEMTOREG <= '1' when OPCODE = lw OR OPCODE = sw else '0';

--SIMD assignments
VREGWRITEH <= '1' when OPCODE = vlw OR OPCODE = vrtype OR OPCODE = vmul else
                    '1' when OPCODE = vandi OR OPCODE = vori OR OPCODE = vxori OR
                            OPCODE = vaddiu else
```

```
                        '0';

VMEMWRITEH <= '1' when OPCODE = vsw else '0';

VMEMTOREG <= '1' when OPCODE = vlw OR OPCODE = vsw else '0';
-------------------

ALUSRC <= '1' when OPCODE(5 downto 3) = "001" AND OPCODE /= "001000" else
                                '1' when OPCODE = lw OR OPCODE = sw else
                                '1' when OPCODE = vlw OR OPCODE = vsw else --SIMD
                                '1' when OPCODE = vandi OR OPCODE = vori OR
                                        OPCODE = vxori OR OPCODE = vaddiu else
                                '0';

EXTCTRL <= '0' when OPCODE = ori OR OPCODE = andi OR OPCODE = xori else
                  '0' when OPCODE = vori OR OPCODE = vandi OR OPCODE = vxori else
                  '1';

REGDST <= '1' when OPCODE = rtype AND FUNCT /= "001101" else
                '1' when OPCODE = mul else
                '1' when OPCODE = vrtype OR OPCODE = vmul else -- SIMD
                '0';

ALUOP <= "00000" when OPCODE = rtype OR OPCODE = vrtype else
                "00001" when OPCODE = addiu OR OPCODE = vaddiu else
                "00010" when OPCODE = andi OR OPCODE = vandi else
                "00011" when OPCODE = ori OR OPCODE = vori else
                "00100" when OPCODE = xori OR OPCODE = vxori else
                "00101" when OPCODE = lui else
                "00110" when OPCODE = slti else
                "00111" when OPCODE = sltiu else
                "01000" when OPCODE = mul OR OPCODE = vmul else
                "10000";

end Behavioral;
```

Branchlogic.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity branchlogic is
   Port ( ISBGEZ : in  STD_LOGIC;
          OPCODE : in  STD_LOGIC_VECTOR (5 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          BRANCHOK : out  STD_LOGIC);
end branchlogic;

architecture Behavioral of branchlogic is

signal neg, ovf, negxorovf, doutzero: std_logic;
```

```
signal branch: std_logic_vector (1 downto 0);
signal btemp, dout: std_logic_vector (31 downto 0);

begin

branch <= "01" when OPCODE = "000001" else  --for bgez and bltz
            "10" when OPCODE = "000100" OR OPCODE = "000101" else --for beq and bne
            "00";

btemp <= x"00000000" when OPCODE = "000001" else --for bgez and bltz
           B;

process (btemp, A, dout, neg, ovf, negxorovf, ISBGEZ, OPCODE, doutzero, branch) is
begin
dout <= A - btemp;
neg <= dout(31);

if ( A(31) = '1' AND btemp(31) = '0' AND dout(31) = '0' ) then
   ovf <= '1';
elsif ( A(31) = '0' AND btemp(31) = '1' AND dout(31) = '1' ) then
   ovf <= '1';
else ovf <= '0';
end if;

negxorovf <= neg XOR ovf;

if dout = x"00000000" then
   doutzero <= '1';
else doutzero <= '0';
end if;


if branch = "01" then
   BRANCHOK <= negxorovf XOR ISBGEZ;
elsif branch = "10" then
   BRANCHOK <= doutzero XOR OPCODE(0);
else BRANCHOK <= '0';
end if;

end process;

end Behavioral;
```

NextPC.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity nextPC is
   Port ( ISBJ : in  STD_LOGIC_VECTOR (2 downto 0);
          INSTINDEX : in  STD_LOGIC_VECTOR (25 downto 0);
          PCPLUS4 : in  STD_LOGIC_VECTOR (31 downto 0);
          RD1 : in  STD_LOGIC_VECTOR (31 downto 0);
```

```
        EXTIMM : in  STD_LOGIC_VECTOR (31 downto 0);
        NEXTPC : out  STD_LOGIC_VECTOR (31 downto 0));
end nextPC;

architecture Behavioral of nextPC is

constant branch: std_logic_vector (2 downto 0):= "001";
constant jr: std_logic_vector (2 downto 0):= "010";
constant j_jal: std_logic_vector (2 downto 0):= "100";

signal branch_val: std_logic_vector (31 downto 0);

begin

branch_val <= EXTIMM(29 downto 0) & "00";

NEXTPC <= PCPLUS4 + branch_val when ISBJ = branch else
              RD1 when ISBJ = jr else
              PCPLUS4(31 downto 28) & INSTINDEX & "00" when ISBJ = j_jal else
              PCPLUS4;

end Behavioral;
```

Registerfile.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity regfile is
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          WEH : in  STD_LOGIC;
          DIN : in  STD_LOGIC_VECTOR (31 downto 0);
          WSEL : in  STD_LOGIC_VECTOR (4 downto 0);
          RSEL1 : in  STD_LOGIC_VECTOR (4 downto 0);
          RSEL2 : in  STD_LOGIC_VECTOR (4 downto 0);
          RD1 : out  STD_LOGIC_VECTOR (31 downto 0);
          RD2 : out  STD_LOGIC_VECTOR (31 downto 0);
          REG23OUT : out STD_LOGIC_VECTOR (31 downto 0));  --Board IO port
end regfile;

architecture Behavioral of regfile is

component reg32
   Port ( CLK : in  STD_LOGIC;
          ENABLEH : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          DIN : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

signal en1, en2, en3, en4, en5, en6, en7, en8, en9, en10, en11, en12, en13, en14, en15, en16: std_logic;
```

```
signal en17, en18, en19, en20, en21, en22, en23, en24, en25, en26, en27, en28, en29, en30, en31:
std_logic;
signal rout0, rout1, rout2, rout3, rout4, rout5, rout6, rout7, rout8, rout9: std_logic_vector (31 downto 0);
signal rout10, rout11, rout12, rout13, rout14, rout15, rout16, rout17, rout18, rout19: std_logic_vector (31
downto 0);
signal rout20, rout21, rout22, rout23, rout24, rout25, rout26, rout27, rout28, rout29: std_logic_vector (31
downto 0);
signal rout30, rout31: std_logic_vector (31 downto 0);

begin

en1 <= '1' when WSEL = "00001" AND WEH = '1' else '0';
en2 <= '1' when WSEL = "00010" AND WEH = '1' else '0';
en3 <= '1' when WSEL = "00011" AND WEH = '1' else '0';
en4 <= '1' when WSEL = "00100" AND WEH = '1' else '0';
en5 <= '1' when WSEL = "00101" AND WEH = '1' else '0';
en6 <= '1' when WSEL = "00110" AND WEH = '1' else '0';
en7 <= '1' when WSEL = "00111" AND WEH = '1' else '0';
en8 <= '1' when WSEL = "01000" AND WEH = '1' else '0';
en9 <= '1' when WSEL = "01001" AND WEH = '1' else '0';
en10 <= '1' when WSEL = "01010" AND WEH = '1' else '0';
en11 <= '1' when WSEL = "01011" AND WEH = '1' else '0';
en12 <= '1' when WSEL = "01100" AND WEH = '1' else '0';
en13 <= '1' when WSEL = "01101" AND WEH = '1' else '0';
en14 <= '1' when WSEL = "01110" AND WEH = '1' else '0';
en15 <= '1' when WSEL = "01111" AND WEH = '1' else '0';
en16 <= '1' when WSEL = "10000" AND WEH = '1' else '0';
en17 <= '1' when WSEL = "10001" AND WEH = '1' else '0';
en18 <= '1' when WSEL = "10010" AND WEH = '1' else '0';
en19 <= '1' when WSEL = "10011" AND WEH = '1' else '0';
en20 <= '1' when WSEL = "10100" AND WEH = '1' else '0';
en21 <= '1' when WSEL = "10101" AND WEH = '1' else '0';
en22 <= '1' when WSEL = "10110" AND WEH = '1' else '0';
en23 <= '1' when WSEL = "10111" AND WEH = '1' else '0';
en24 <= '1' when WSEL = "11000" AND WEH = '1' else '0';
en25 <= '1' when WSEL = "11001" AND WEH = '1' else '0';
en26 <= '1' when WSEL = "11010" AND WEH = '1' else '0';
en27 <= '1' when WSEL = "11011" AND WEH = '1' else '0';
en28 <= '1' when WSEL = "11100" AND WEH = '1' else '0';
en29 <= '1' when WSEL = "11101" AND WEH = '1' else '0';
en30 <= '1' when WSEL = "11110" AND WEH = '1' else '0';
en31 <= '1' when WSEL = "11111" AND WEH = '1' else '0';

rout0 <= x"00000000";
reg1: component reg32 port map(CLK,en1,RESET,DIN,rout1);
reg2: component reg32 port map(CLK,en2,RESET,DIN,rout2);
reg3: component reg32 port map(CLK,en3,RESET,DIN,rout3);
reg4: component reg32 port map(CLK,en4,RESET,DIN,rout4);
reg5: component reg32 port map(CLK,en5,RESET,DIN,rout5);
reg6: component reg32 port map(CLK,en6,RESET,DIN,rout6);
reg7: component reg32 port map(CLK,en7,RESET,DIN,rout7);
reg8: component reg32 port map(CLK,en8,RESET,DIN,rout8);
reg9: component reg32 port map(CLK,en9,RESET,DIN,rout9);
reg10: component reg32 port map(CLK,en10,RESET,DIN,rout10);
reg11: component reg32 port map(CLK,en11,RESET,DIN,rout11);
reg12: component reg32 port map(CLK,en12,RESET,DIN,rout12);
```

```
reg13: component reg32 port map(CLK,en13,RESET,DIN,rout13);
reg14: component reg32 port map(CLK,en14,RESET,DIN,rout14);
reg15: component reg32 port map(CLK,en15,RESET,DIN,rout15);
reg16: component reg32 port map(CLK,en16,RESET,DIN,rout16);
reg17: component reg32 port map(CLK,en17,RESET,DIN,rout17);
reg18: component reg32 port map(CLK,en18,RESET,DIN,rout18);
reg19: component reg32 port map(CLK,en19,RESET,DIN,rout19);
reg20: component reg32 port map(CLK,en20,RESET,DIN,rout20);
reg21: component reg32 port map(CLK,en21,RESET,DIN,rout21);
reg22: component reg32 port map(CLK,en22,RESET,DIN,rout22);
reg23: component reg32 port map(CLK,en23,RESET,DIN,rout23);
reg24: component reg32 port map(CLK,en24,RESET,DIN,rout24);
reg25: component reg32 port map(CLK,en25,RESET,DIN,rout25);
reg26: component reg32 port map(CLK,en26,RESET,DIN,rout26);
reg27: component reg32 port map(CLK,en27,RESET,DIN,rout27);
reg28: component reg32 port map(CLK,en28,RESET,DIN,rout28);
reg29: component reg32 port map(CLK,en29,RESET,DIN,rout29);
reg30: component reg32 port map(CLK,en30,RESET,DIN,rout30);
reg31: component reg32 port map(CLK,en31,RESET,DIN,rout31);

RD1 <= rout0 when RSEL1 = 0 else
                rout1 when RSEL1 = 1 else
                rout2 when RSEL1 = 2 else
                rout3 when RSEL1 = 3 else
                rout4 when RSEL1 = 4 else
                rout5 when RSEL1 = 5 else
                rout6 when RSEL1 = 6 else
                rout7 when RSEL1 = 7 else
                rout8 when RSEL1 = 8 else
                rout9 when RSEL1 = 9 else
                rout10 when RSEL1 = 10 else
                rout11 when RSEL1 = 11 else
                rout12 when RSEL1 = 12 else
                rout13 when RSEL1 = 13 else
                rout14 when RSEL1 = 14 else
                rout15 when RSEL1 = 15 else
                rout16 when RSEL1 = 16 else
                rout17 when RSEL1 = 17 else
                rout18 when RSEL1 = 18 else
                rout19 when RSEL1 = 19 else
                rout20 when RSEL1 = 20 else
                rout21 when RSEL1 = 21 else
                rout22 when RSEL1 = 22 else
                rout23 when RSEL1 = 23 else
                rout24 when RSEL1 = 24 else
                rout25 when RSEL1 = 25 else
                rout26 when RSEL1 = 26 else
                rout27 when RSEL1 = 27 else
                rout28 when RSEL1 = 28 else
                rout29 when RSEL1 = 29 else
                rout30 when RSEL1 = 30 else
                rout31 when RSEL1 = 31 else
                rout0;

RD2 <= rout0 when RSEL2 = 0 else
                rout1 when RSEL2 = 1 else
```

```
                rout2 when RSEL2 = 2 else
                rout3 when RSEL2 = 3 else
                rout4 when RSEL2 = 4 else
                rout5 when RSEL2 = 5 else
                rout6 when RSEL2 = 6 else
                rout7 when RSEL2 = 7 else
                rout8 when RSEL2 = 8 else
                rout9 when RSEL2 = 9 else
                rout10 when RSEL2 = 10 else
                rout11 when RSEL2 = 11 else
                rout12 when RSEL2 = 12 else
                rout13 when RSEL2 = 13 else
                rout14 when RSEL2 = 14 else
                rout15 when RSEL2 = 15 else
                rout16 when RSEL2 = 16 else
                rout17 when RSEL2 = 17 else
                rout18 when RSEL2 = 18 else
                rout19 when RSEL2 = 19 else
                rout20 when RSEL2 = 20 else
                rout21 when RSEL2 = 21 else
                rout22 when RSEL2 = 22 else
                rout23 when RSEL2 = 23 else
                rout24 when RSEL2 = 24 else
                rout25 when RSEL2 = 25 else
                rout26 when RSEL2 = 26 else
                rout27 when RSEL2 = 27 else
                rout28 when RSEL2 = 28 else
                rout29 when RSEL2 = 29 else
                rout30 when RSEL2 = 30 else
                rout31 when RSEL2 = 31 else
                rout0;

REG23OUT <= rout23;

end Behavioral;
```

Register.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity reg32 is
   Port ( CLK : in  STD_LOGIC;
          ENABLEH : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          DIN : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end reg32;

architecture Behavioral of reg32 is

begin
          process(CLK, RESET)
```

```
	begin
		if RESET = '1' then
			DOUT <= x"00000000";
		elsif rising_edge(CLK) then
			if ENABLEH = '1' then
				DOUT <= DIN;
			end if;
		end if;
	end process;

end Behavioral;
```

Hazardcontrol.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity hazardcontrol is
   Port ( MEMTOREGEXE : in  STD_LOGIC;
          REGWRITEEXE : in  STD_LOGIC;
          REGWRITEMEM : in  STD_LOGIC;
          ISJAL : in  STD_LOGIC;
          ISMUL : in STD_LOGIC;
          RSEL1ID : in  STD_LOGIC_VECTOR (4 downto 0);
          RSEL2ID : in  STD_LOGIC_VECTOR (4 downto 0);
          WBEXE : in  STD_LOGIC_VECTOR (4 downto 0);
          WBMEM : in  STD_LOGIC_VECTOR (4 downto 0);
          STALL : out  STD_LOGIC;
          FORWARDA : out  STD_LOGIC_VECTOR (2 downto 0);
          FORWARDM : out  STD_LOGIC_VECTOR (2 downto 0);
          --SIMD Ports
          VMEMTOREGEXE : in  STD_LOGIC;
          VREGWRITEEXE : in  STD_LOGIC;
          VREGWRITEMEM : in  STD_LOGIC;
          VFORWARDA : out  STD_LOGIC_VECTOR (1 downto 0);
          VFORWARDM : out  STD_LOGIC_VECTOR (1 downto 0));
end hazardcontrol;

architecture Behavioral of hazardcontrol is

constant regfileout: std_logic_vector (2 downto 0):= "000";
constant jal: std_logic_vector (2 downto 0):= "001";
constant computationout: std_logic_vector (2 downto 0):= "010";
constant memoryout: std_logic_vector (2 downto 0):= "011";

constant vregfileout: std_logic_vector (1 downto 0):= "00";
constant valuout: std_logic_vector (1 downto 0):= "01";
constant vmemoryout: std_logic_vector (1 downto 0):= "10";

begin

process(REGWRITEEXE, WBEXE, RSEL1ID, REGWRITEMEM, WBMEM, ISJAL)
begin
```

```
	if REGWRITEEXE = '1' AND WBEXE /= "00000" AND WBEXE = RSEL1ID then
			FORWARDA <= computationout;
	elsif REGWRITEMEM = '1' AND WBMEM /= "00000" AND WBMEM = RSEL1ID then
			FORWARDA <= memoryout;
	elsif ISJAL = '1' then
			FORWARDA <= jal;
	else --takes care of RESET case
			FORWARDA <= regfileout;
	end if;
end process;

process(REGWRITEEXE, WBEXE, RSEL2ID, REGWRITEMEM, WBMEM, ISJAL)
begin
	if REGWRITEEXE = '1' AND WBEXE /= "00000" AND WBEXE = RSEL2ID then
			FORWARDM <= computationout;
	elsif REGWRITEMEM = '1' AND WBMEM /= "00000" AND WBMEM = RSEL2ID then
			FORWARDM <= memoryout;
	elsif ISJAL = '1' then
			FORWARDM <= jal;
	else --takes care of RESET case
			FORWARDM <= regfileout;
	end if;
end process;

--SIMD processes
process(VREGWRITEEXE, WBEXE, RSEL1ID, VREGWRITEMEM, WBMEM)
begin
	if VREGWRITEEXE = '1' AND WBEXE /= "00000" AND WBEXE = RSEL1ID then
			VFORWARDA <= valuout;
	elsif VREGWRITEMEM = '1' AND WBMEM /= "00000" AND WBMEM = RSEL1ID then
			VFORWARDA <= vmemoryout;
	else --takes care of RESET case
			VFORWARDA <= vregfileout;
	end if;
end process;

process(VREGWRITEEXE, WBEXE, RSEL2ID, VREGWRITEMEM, WBMEM)
begin
	if VREGWRITEEXE = '1' AND WBEXE /= "00000" AND WBEXE = RSEL2ID then
			VFORWARDM <= valuout;
	elsif VREGWRITEMEM = '1' AND WBMEM /= "00000" AND WBMEM = RSEL2ID then
			VFORWARDM <= vmemoryout;
	else --takes care of RESET case
			VFORWARDM <= vregfileout;
	end if;
end process;
-----------------

STALL <= '1' when REGWRITEEXE = '1' AND MEMTOREGEXE = '1' AND WBEXE /= 0 AND
WBEXE = RSEL2ID else
	  '1' when REGWRITEEXE = '1' AND MEMTOREGEXE = '1' AND WBEXE /= 0 AND
WBEXE = RSEL1ID else
	 '1' when ISMUL = '1' AND WBEXE /= 0 AND WBEXE = RSEL1ID else
	 '1' when ISMUL = '1' AND WBEXE /= 0 AND WBEXE = RSEL2ID else
	 '1' when VREGWRITEEXE = '1' AND VMEMTOREGEXE = '1' AND WBEXE /= 0 AND
WBEXE = RSEL2ID else
```

```
            '1' when VREGWRITEEXE = '1' AND VMEMTOREGEXE = '1' AND WBEXE /= 0 AND
WBEXE = RSEL1ID else
            '0';

end Behavioral;
```

EXEstage.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity EXEstage is
          Generic (datasize : natural := 127;
                   size : natural := 4);
   Port ( CLK : in STD_LOGIC;
          ALUop : in  STD_LOGIC_VECTOR (4 downto 0);
          SHAMT : in  STD_LOGIC_VECTOR (4 downto 0);
          FUNCT : in  STD_LOGIC_VECTOR (5 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          MULRESULT : out  STD_LOGIC_VECTOR (31 downto 0);
          RESULT : out  STD_LOGIC_VECTOR (31 downto 0);
          --SIMD Ports
         VA : in  STD_LOGIC_VECTOR (datasize downto 0);
         VB : in  STD_LOGIC_VECTOR (datasize downto 0);
         VMULRESULT : out  STD_LOGIC_VECTOR (datasize downto 0);
         VRESULT : out  STD_LOGIC_VECTOR (datasize downto 0));
end EXEstage;

architecture Behavioral of EXEstage is

component computation is
   Port ( SHDIR : in  STD_LOGIC;
          SHEXTMODE : in  STD_LOGIC;
          SIGNEDCOMP : in  STD_LOGIC;
          MSEL : in  STD_LOGIC_VECTOR (1 downto 0);
          FSEL : in  STD_LOGIC_VECTOR (2 downto 0);
          SHAMT : in  STD_LOGIC_VECTOR (4 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component multiply is
          port (
          clk: IN std_logic;
          a: IN std_logic_VECTOR(15 downto 0);
          b: IN std_logic_VECTOR(15 downto 0);
          p: OUT std_logic_VECTOR(31 downto 0));
end component multiply;

component ALUcontroller is
   Port ( FUNCT : in  STD_LOGIC_VECTOR (5 downto 0);
```

```
        ALUop : in  STD_LOGIC_VECTOR (4 downto 0);
        SIGNEDCOMP : out  STD_LOGIC;
        SHDIR : out  STD_LOGIC;
        SHEXTMODE : out  STD_LOGIC;
        MSEL : out  STD_LOGIC_VECTOR (1 downto 0);
        FSEL : out  STD_LOGIC_VECTOR (2 downto 0));
end component;

component ALU is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
        B : in  STD_LOGIC_VECTOR (31 downto 0);
        DOUT : out  STD_LOGIC_VECTOR (31 downto 0);
        FSEL : in  STD_LOGIC_VECTOR (2 downto 0);
        COUT : out  STD_LOGIC;
        ZERO : out  STD_LOGIC;
        NEG : out  STD_LOGIC;
        OVF : out  STD_LOGIC);
end component ALU;

signal signedcomp, shdir, shextmode: std_logic;
signal msel: std_logic_vector (1 downto 0);
signal fsel: std_logic_vector (2 downto 0);

--SIMD signals
type data_array is array (1 to size) of STD_LOGIC_VECTOR (31 downto 0);
signal va_array, vb_array, vresult_array, vmultresult_array : data_array;

begin

ALUcontrol: component ALUcontroller port map(FUNCT => FUNCT,
                                             ALUop => ALUop,
                                             SIGNEDCOMP => signedcomp,
                                             SHDIR => shdir,
                                             SHEXTMODE => shextmode,
                                             MSEL => msel,
                                             FSEL => fsel);

mult: component multiply port map(clk => CLK,
                                  a => A(15 downto 0),
                                  b => B(15 downto 0),
                                  p => MULRESULT);


calc: component computation port map(SHDIR => shdir,
                                     SHEXTMODE => shextmode,
                                     SIGNEDCOMP => signedcomp,
                                     MSEL => msel,
                                     FSEL => fsel,
                                     SHAMT => SHAMT,
                                     A => A,
                                     B => B,
                                     DOUT => RESULT);

va_array(4) <= VA(31 downto 0);
va_array(3) <= VA(63 downto 32);
va_array(2) <= VA(95 downto 64);
```

```
va_array(1) <= VA(127 downto 96);

vb_array(4) <= VB(31 downto 0);
vb_array(3) <= VB(63 downto 32);
vb_array(2) <= VB(95 downto 64);
vb_array(1) <= VB(127 downto 96);

genalu: for I in 1 to size generate
genSIMDalu: component ALU port map(A => va_array(I),
                                   B => vb_array(I),
                                   DOUT => vresult_array(I),
                                   FSEL => fsel,
                                   COUT => open,
                                   ZERO => open,
                                   NEG => open,
                                   OVF => open);

genSIMDmult: component multiply port map(clk => CLK,
                                         a => va_array(I)(15 downto 0),
                                         b => vb_array(I)(15 downto 0),
                                         p => vmultresult_array(I));

end generate;

VRESULT <= vresult_array(1) & vresult_array(2) & vresult_array(3) & vresult_array(4);
VMULRESULT <= vmultresult_array(1) & vmultresult_array(2) & vmultresult_array(3) &
vmultresult_array(4);

end Behavioral;
```

ALUcontroller.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ALUcontroller is
   Port ( FUNCT : in  STD_LOGIC_VECTOR (5 downto 0);
          ALUop : in  STD_LOGIC_VECTOR (3 downto 0);
          SIGNEDCOMP : out  STD_LOGIC;
          SHDIR : out  STD_LOGIC;
          SHEXTMODE : out  STD_LOGIC;
          MSEL : out  STD_LOGIC_VECTOR (1 downto 0);
          FSEL : out  STD_LOGIC_VECTOR (2 downto 0));
end ALUcontroller;

architecture Behavioral of ALUcontroller is

--ALUops
constant rtype: std_logic_vector (4 downto 0):= "00000"; --vrtype as well
constant addiu: std_logic_vector (4 downto 0):= "00001"; --vaddiu as well
constant andi: std_logic_vector (4 downto 0):= "00010"; --vandi as well
constant ori: std_logic_vector (4 downto 0):= "00011"; --vori as well
constant xori: std_logic_vector (4 downto 0):= "00100"; --vxori as well
constant lui: std_logic_vector (4 downto 0):= "00101";
```

```
constant slti: std_logic_vector (4 downto 0):= "00110";
constant sltiu: std_logic_vector (4 downto 0):= "00111";
--"01000" is for mul, also acts at TAKEMULRESULT in processor
--"10000" is for others

--Funct
constant addu: std_logic_vector (5 downto 0):= "100001"; --vaddu as well
constant subu: std_logic_vector (5 downto 0):= "100011";
constant and1: std_logic_vector (5 downto 0):= "100100";  --vand as well
constant or1: std_logic_vector (5 downto 0):= "100101";    --vor as well
constant xor1: std_logic_vector (5 downto 0):= "100110";  --vxor as well
constant sll1: std_logic_vector (5 downto 0):= "000000";
constant srl1: std_logic_vector (5 downto 0):= "000010";
constant sra1: std_logic_vector (5 downto 0):= "000011";
constant slt: std_logic_vector (5 downto 0):= "101010";
constant sltu: std_logic_vector (5 downto 0):= "101011";

--MSEL
constant shifter: std_logic_vector (1 downto 0):= "00";
constant compare: std_logic_vector (1 downto 0):= "01";
constant alu: std_logic_vector (1 downto 0):= "10";

begin

SHDIR <= '1' when ALUop = rtype AND FUNCT = sll1 else '0';
SHEXTMODE <= '1' when ALUop = rtype AND FUNCT = sra1 else '0';
SIGNEDCOMP <= '1' when ALUop = slti OR FUNCT = slt else '0';

process(ALUop, FUNCT)
begin
	case ALUop is
		when rtype => case FUNCT is
					when addu => FSEL <= "000"; MSEL <= alu;
					when subu => FSEL <= "100"; MSEL <= alu;
					when and1 => FSEL <= "001"; MSEL <= alu;
					when or1 => FSEL <= "010"; MSEL <= alu;
					when xor1 => FSEL <= "011"; MSEL <= alu;
					when sll1 => FSEL <= "000"; MSEL <= shifter;
					when srl1 => FSEL <= "000"; MSEL <= shifter;
					when sra1 => FSEL <= "000"; MSEL <= shifter;
					when slt => FSEL <= "100"; MSEL <= compare;
					when sltu => FSEL <= "100"; MSEL <= compare;
					when others => FSEL <= "000"; MSEL <= alu;
					end case;
		when andi => FSEL <= "001"; MSEL <= alu;
		when ori => FSEL <= "010"; MSEL <= alu;
		when xori => FSEL <= "011"; MSEL <= alu;
		when lui => FSEL <= "101"; MSEL <= alu;
		when slti => FSEL <= "100"; MSEL <= compare;
		when sltiu => FSEL <= "100"; MSEL <= compare;
		when others => FSEL <= "000"; MSEL <= alu;
	end case;
end process;

end Behavioral;
```

Computation.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity computation is
   Port ( SHDIR : in  STD_LOGIC;
          SHEXTMODE : in  STD_LOGIC;
          SIGNEDCOMP : in  STD_LOGIC;
          MSEL : in  STD_LOGIC_VECTOR (1 downto 0);
          FSEL : in  STD_LOGIC_VECTOR (2 downto 0);
          SHAMT : in  STD_LOGIC_VECTOR (4 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end computation;

architecture Behavioral of computation is

component ALU is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0);
          FSEL : in  STD_LOGIC_VECTOR (2 downto 0);
          COUT : out  STD_LOGIC;
          ZERO : out  STD_LOGIC;
          NEG : out  STD_LOGIC;
          OVF : out  STD_LOGIC);
end component;

component shifter is
   Port ( SHDIR : in  STD_LOGIC;
          EXTMODE : in  STD_LOGIC;
          SHAMT : in  STD_LOGIC_VECTOR (4 downto 0);
          DIN : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component shifter;

component comparator is
   Port ( SIGNEDCOMP : in  STD_LOGIC;
          COUT : in  STD_LOGIC;
          NEG : in  STD_LOGIC;
          OVF : in  STD_LOGIC;
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component mux32x3 is
   Port ( SEL : in  STD_LOGIC_VECTOR (1 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          C : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component;
```

```
signal cout, neg, ovf, zero: std_logic;
signal ALUdata, shiftdata, comparedata: std_logic_vector(31 downto 0);

begin

alu1: component ALU port map(A => A,
                             B => B,
                             DOUT => ALUdata,
                             FSEL => FSEL,
                             COUT => cout,
                             ZERO => zero,
                             NEG => neg,
                             OVF => ovf);

shift: component shifter port map(SHDIR => SHDIR,
                                  EXTMODE => SHEXTMODE,
                                  SHAMT => SHAMT,
                                  DIN => B,
                                  DOUT => shiftdata);

compare: component comparator port map(SIGNEDCOMP => SIGNEDCOMP,
                                       COUT => cout,
                                       NEG => neg,
                                       OVF => ovf,
                                       DOUT => comparedata);

datamux: component mux32x3 port map(SEL => MSEL,
                                    A => shiftdata,
                                    B => comparedata,
                                    C => ALUdata,
                                    DOUT => DOUT);

end Behavioral;
```

Multiply.vhd:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- synopsys translate_off
Library XilinxCoreLib;
-- synopsys translate_on
ENTITY multiply IS
        port (
        clk: IN std_logic;
        a: IN std_logic_VECTOR(15 downto 0);
        b: IN std_logic_VECTOR(15 downto 0);
        p: OUT std_logic_VECTOR(31 downto 0));
END multiply;

ARCHITECTURE multiply_a OF multiply IS
-- synopsys translate_off
component wrapped_multiply
        port (
        clk: IN std_logic;
```

```
        a: IN std_logic_VECTOR(15 downto 0);
        b: IN std_logic_VECTOR(15 downto 0);
        p: OUT std_logic_VECTOR(31 downto 0));
end component;

-- Configuration specification
        for all : wrapped_multiply use entity XilinxCoreLib.mult_gen_v9_0(behavioral)
                generic map(
                        c_a_width => 16,
                        c_b_type => 0,
                        c_ce_overrides_sclr => 0,
                        c_opt_goal => 1,
                        c_has_sclr => 0,
                        c_round_pt => 0,
                        c_out_high => 31,
                        c_mult_type => 0,
                        c_ccm_imp => 0,
                        c_has_load_done => 0,
                        c_pipe_stages => 1,
                        c_has_ce => 0,
                        c_has_zero_detect => 0,
                        c_round_output => 0,
                        c_mem_init_prefix => "mgv9",
                        c_xdevicefamily => "virtex2p",
                        c_a_type => 0,
                        c_out_low => 0,
                        c_b_width => 16,
                        c_b_value => "10000001");
-- synopsys translate_on
BEGIN
-- synopsys translate_off
U0 : wrapped_multiply
                port map (
                        clk => clk,
                        a => a,
                        b => b,
                        p => p);
-- synopsys translate_on

END multiply_a;
```

ALU.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ALU is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0);
          FSEL : in  STD_LOGIC_VECTOR (2 downto 0);
          COUT : out  STD_LOGIC;
```

```
        ZERO : out  STD_LOGIC;
        NEG : out  STD_LOGIC;
        OVF : out  STD_LOGIC);
end ALU;

architecture Behavioral of ALU is

signal sum, tempsum, tempsub, tempa, tempb : std_logic_vector (32 downto 0);
signal douttemp: std_logic_vector (31 downto 0);

begin


tempa <= '0'& A ;
tempb <= '0' & B;
tempsum <= tempa + tempb;          --ADD
tempsub <= tempa - tempb; --SUB

process(A, B, FSEL, douttemp, tempsum, tempsub, sum)
        begin
                case FSEL is
                        when "000" => sum <= tempsum;--'0'& A + '0' & B;  --ADD
                                        douttemp <= sum(31 downto 0);
                                        DOUT <= douttemp;
                                        COUT <= sum(32);
                                        if douttemp = x"00000000" then
                                                ZERO <= '1';
                                        else
                                                ZERO <= '0';
                                        end if;
                                        NEG <= douttemp(31);
                                        if A(31) = '0' AND B(31) = '0' AND douttemp(31) = '1' then
                                                OVF <= '1';
                                        elsif A(31) = '1' AND B(31) = '1' AND douttemp(31) = '0' then
                                                OVF <= '1';
                                        else
                                                OVF <= '0';
                                        end if;
                        when "001" => douttemp <= A AND B;      --AND
                                        DOUT <= douttemp;
                                        COUT <= '0';
                                        if douttemp = x"00000000" then
                                                ZERO <= '1';
                                        else
                                                ZERO <= '0';
                                        end if;
                                        NEG <= douttemp(31);
                                        OVF <= '0';
                        when "010" => douttemp <= A OR B;         --OR
                                        DOUT <= douttemp;
                                        COUT <= '0';
                                        if douttemp = x"00000000" then
                                                ZERO <= '1';
                                        else
                                                ZERO <= '0';
                                        end if;
```

```
                                        NEG <= douttemp(31);
                                        OVF <= '0';
                                when "011" => douttemp <= A XOR B;       --XOR
                                        DOUT <= douttemp;
                                        COUT <= '0';
                                        if douttemp = x"00000000" then
                                                ZERO <= '1';
                                        else
                                                ZERO <= '0';
                                        end if;
                                        NEG <= douttemp(31);
                                        OVF <= '0';
                                when "100" => sum <= tempsub;--'0'&A - '0'&B;       --SUB
                                        douttemp <= sum(31 downto 0);
                                        DOUT <= douttemp;
                                        COUT <= sum(32);
                                        if douttemp = x"00000000" then
                                                ZERO <= '1';
                                        else
                                                ZERO <= '0';
                                        end if;
                                        NEG <= douttemp(31);
                                        if A(31) = '1' AND B(31) = '0' AND douttemp(31) = '0' then
                                                OVF <= '1';
                                        elsif A(31) = '0' AND B(31) = '1' AND douttemp(31) = '1' then
                                                OVF <= '1';
                                        else
                                                OVF <= '0';
                                        end if;
                                when "101" => DOUT(31 downto 16) <= B(15 downto 0);
                                        DOUT(15 downto 0) <= x"0000";
                                        COUT <= '0';
                                        ZERO <= '0';
                                        NEG <= '0';
                                        OVF <= '0';
                                when others => DOUT <= x"00000000";
                                        COUT <= '0';
                                        ZERO <= '1';
                                        NEG <= '0';
                                        OVF <= '0';
                        end case;
                end process;
end Behavioral;
```

Comparator.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity comparator is
    Port ( SIGNEDCOMP : in  STD_LOGIC;
           COUT : in  STD_LOGIC;
           NEG : in  STD_LOGIC;
```

```
        OVF : in  STD_LOGIC;
        DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end comparator;

architecture Behavioral of comparator is

component mux32x2 is
   Port ( SEL : in  STD_LOGIC;
        A : in  STD_LOGIC_VECTOR (31 downto 0);
        B : in  STD_LOGIC_VECTOR (31 downto 0);
        DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

signal negxorovf: std_logic;
signal zeros: std_logic_vector (30 downto 0):= "000" & x"0000000";
signal atemp, btemp: std_logic_vector (31 downto 0);

begin

negxorovf <= NEG XOR OVF;
atemp <= zeros & COUT;
btemp <= zeros & negxorovf;
slt_umux:component mux32x2 port map(SIGNEDCOMP, atemp, btemp, DOUT);

end Behavioral;
```

Shifter.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity shifter is
   Port ( SHDIR : in  STD_LOGIC;                          -- 0 = right, 1 = left
        EXTMODE : in  STD_LOGIC;
        SHAMT : in  STD_LOGIC_VECTOR (4 downto 0);
        DIN : in  STD_LOGIC_VECTOR (31 downto 0);
        DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end shifter;

architecture Behavioral of shifter is

type stage_array is array(0 to 4) of std_logic_vector(31 downto 0);
signal sll_array, srl_array, sra_array : stage_array;

begin

sll_array(0) <= DIN when SHAMT(4) = '0' else DIN(15 downto 0) & X"0000";
sll_array(1) <= sll_array(0) when SHAMT(3) = '0' else sll_array(0)(23 downto 0) & X"00";
sll_array(2) <= sll_array(1) when SHAMT(2) = '0' else sll_array(1)(27 downto 0) & X"0";
sll_array(3) <= sll_array(2) when SHAMT(1) = '0' else sll_array(2)(29 downto 0) & "00";
sll_array(4) <= sll_array(3) when SHAMT(0) = '0' else sll_array(3)(30 downto 0) & '0';

srl_array(0) <= DIN when SHAMT(4) = '0' else X"0000" & DIN(31 downto 16);
```

```
srl_array(1) <= srl_array(0) when SHAMT(3) = '0' else X"00" & srl_array(0)(31 downto 8);
srl_array(2) <= srl_array(1) when SHAMT(2) = '0' else X"0" & srl_array(1)(31 downto 4);
srl_array(3) <= srl_array(2) when SHAMT(1) = '0' else "00" & srl_array(2)(31 downto 2);
srl_array(4) <= srl_array(3) when SHAMT(0) = '0' else '0' & srl_array(3)(31 downto 1);

sra_array(0) <= DIN when SHAMT(4) = '0' else X"FFFF" & DIN(31 downto 16);
sra_array(1) <= sra_array(0) when SHAMT(3) = '0' else X"FF" & sra_array(0)(31 downto 8);
sra_array(2) <= sra_array(1) when SHAMT(2) = '0' else X"F" & sra_array(1)(31 downto 4);
sra_array(3) <= sra_array(2) when SHAMT(1) = '0' else "11" & sra_array(2)(31 downto 2);
sra_array(4) <= sra_array(3) when SHAMT(0) = '0' else '1' & sra_array(3)(31 downto 1);


DOUT <= sll_array(4) when SHDIR = '1' else
              sra_array(4) when EXTMODE = '1' and DIN(31) = '1' else
               srl_array(4);


end architecture Behavioral;
```

Mux32x4.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux32x4 is
   Port ( SEL : in  STD_LOGIC_VECTOR (2 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          C : in  STD_LOGIC_VECTOR (31 downto 0);
          D : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end mux32x4;

architecture Behavioral of mux32x4 is

begin

DOUT <= A when SEL = "000" else
              B when SEL = "001" else
              C when SEL = "010" else
              D when SEL = "011" else
              x"deaddead";

end Behavioral;
```

Mux32x3.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity mux32x3 is
   Port ( SEL : in  STD_LOGIC_VECTOR (1 downto 0);
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          C : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end mux32x3;

architecture Behavioral of mux32x3 is

begin

DOUT <= A when SEL = "00" else
              B when SEL = "01" else
              C when SEL = "10" else
              x"deaddead";

end Behavioral;
```

Mux32x2.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux32x2 is
   Port ( SEL : in  STD_LOGIC;
          A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          DOUT : out  STD_LOGIC_VECTOR (31 downto 0));
end mux32x2;

architecture Behavioral of mux32x2 is

begin

DOUT <= A when SEL = '0' else
              B when SEL = '1';

end Behavioral;
```

MEMstage.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity MEMstage is
            Generic (datasize : natural := 127;
                     size : natural := 4);
   Port ( CLK : in  STD_LOGIC;
```

```
            RESET : in  STD_LOGIC;
            ADDR : in  STD_LOGIC_VECTOR (31 downto 0);
            DIN : in  STD_LOGIC_VECTOR (31 downto 0);
            MEMTOREG : in  STD_LOGIC;
            MEMWRITEH : in  STD_LOGIC;
            DOUT : out  STD_LOGIC_VECTOR (31 downto 0);
            --SIMD Ports
            VADDR : in STD_LOGIC_VECTOR (31 downto 0);
            VDIN : in STD_LOGIC_VECTOR (datasize downto 0);
            VMEMTOREG : in STD_LOGIC;
            VMEMWRITEH : in STD_LOGIC;
            VDOUT : out STD_LOGIC_VECTOR (datasize downto 0));
end MEMstage;

architecture Behavioral of MEMstage is

component datamem is
            port (
            addr: in std_logic_vector(9 downto 0);
            clk: in std_logic;
            din: in std_logic_vector(31 downto 0);
            dout: out std_logic_vector(31 downto 0);
            en: in std_logic;
            sinit: in std_logic;
            we: in std_logic);
end component datamem;

component simd_memory_1 is
            port (
            addr: IN std_logic_VECTOR(9 downto 0);
            clk: IN std_logic;
            din: IN std_logic_VECTOR(31 downto 0);
            dout: OUT std_logic_VECTOR(31 downto 0);
            en: in std_logic;
            sinit: in std_logic;
            we: in std_logic);
end component simd_memory_1;

component simd_memory_2 is
            port (
            addr: IN std_logic_VECTOR(9 downto 0);
            clk: IN std_logic;
            din: IN std_logic_VECTOR(31 downto 0);
            dout: OUT std_logic_VECTOR(31 downto 0);
            en: in std_logic;
            sinit: in std_logic;
            we: in std_logic);
end component simd_memory_2;

type data_array is array (1 to size) of STD_LOGIC_VECTOR (31 downto 0);
signal vdin_array, vdout_array : data_array;

begin

datamemory: component datamem port map(addr => ADDR(11 downto 2),
                                       clk => CLK,
```

```
                                        din => DIN,
                                        dout => DOUT,
                                        en => MEMTOREG,
                                        sinit => RESET,
                                        we => MEMWRITEH);

vdin_array(4) <= VDIN(31 downto 0);
vdin_array(3) <= VDIN(63 downto 32);
vdin_array(2) <= VDIN(95 downto 64);
vdin_array(1) <= VDIN(127 downto 96);


SIMDmem1: component simd_memory_1 port map(addr => VADDR(11 downto 2),
                                           clk => CLK,
                                           din => vdin_array(1),
                                           dout => vdout_array(1),
                                           en => VMEMTOREG,
                                           sinit => RESET,
                                           we => VMEMWRITEH);

SIMDmem2: component simd_memory_2 port map(addr => VADDR(11 downto 2),
                                           clk => CLK,
                                           din => vdin_array(2),
                                           dout => vdout_array(2),
                                           en => VMEMTOREG,
                                           sinit => RESET,
                                           we => VMEMWRITEH);

SIMDmem3: component simd_memory_2 port map(addr => VADDR(11 downto 2),
                                           clk => CLK,
                                           din => vdin_array(3),
                                           dout => vdout_array(3),
                                           en => VMEMTOREG,
                                           sinit => RESET,
                                           we => VMEMWRITEH);

SIMDmem4: component simd_memory_2 port map(addr => VADDR(11 downto 2),
                                           clk => CLK,
                                           din => vdin_array(4),
                                           dout => vdout_array(4),
                                           en => VMEMTOREG,
                                           sinit => RESET,
                                           we => VMEMWRITEH);


VDOUT <= vdout_array(1) & vdout_array(2) & vdout_array(3) & vdout_array(4);

end Behavioral;
```

Datamem.vhd:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- synopsys translate_off
Library XilinxCoreLib;
```

```
-- synopsys translate_on
ENTITY datamem IS
	port (
	addr: IN std_logic_VECTOR(9 downto 0);
	clk: IN std_logic;
	din: IN std_logic_VECTOR(31 downto 0);
	dout: OUT std_logic_VECTOR(31 downto 0);
	en: IN std_logic;
	sinit: IN std_logic;
	we: IN std_logic);
END datamem;

ARCHITECTURE datamem_a OF datamem IS
-- synopsys translate_off
component wrapped_datamem
	port (
	addr: IN std_logic_VECTOR(9 downto 0);
	clk: IN std_logic;
	din: IN std_logic_VECTOR(31 downto 0);
	dout: OUT std_logic_VECTOR(31 downto 0);
	en: IN std_logic;
	sinit: IN std_logic;
	we: IN std_logic);
end component;

-- Configuration specification
	for all : wrapped_datamem use entity XilinxCoreLib.blkmemsp_v6_2(behavioral)
		generic map(
			c_sinit_value => "0",
			c_has_en => 1,
			c_reg_inputs => 0,
			c_yclk_is_rising => 1,
			c_ysinit_is_high => 1,
			c_ywe_is_high => 1,
			c_yprimitive_type => "16kx1",
			c_ytop_addr => "1024",
			c_yhierarchy => "hierarchy1",
			c_has_limit_data_pitch => 0,
			c_has_rdy => 0,
			c_write_mode => 0,
			c_width => 32,
			c_yuse_single_primitive => 0,
			c_has_nd => 0,
			c_has_we => 1,
			c_enable_rlocs => 0,
			c_has_rfd => 0,
			c_has_din => 1,
			c_ybottom_addr => "0",
			c_pipe_stages => 0,
			c_yen_is_high => 1,
			c_depth => 1024,
			c_has_default_data => 1,
			c_limit_data_pitch => 18,
			c_has_sinit => 1,
			c_yydisable_warnings => 1,
			c_mem_init_file => "mif_file_16_1",
```

```
                        c_default_data => "0",
                        c_ymake_bmm => 0,
                        c_addr_width => 10);
-- synopsys translate_on
BEGIN
-- synopsys translate_off
U0 : wrapped_datamem
                port map (
                        addr => addr,
                        clk => clk,
                        din => din,
                        dout => dout,
                        en => en,
                        sinit => sinit,
                        we => we);
-- synopsys translate_on

END datamem_a;
```

Appendix C: Program Codes

Hazard Test:

```
main:
CaseA:
        ori     $v0, $0, 87
        or      $v1, $v0, $0
        beq     $v0, $v1, CaseB
        sll     $0, $0, 0
        j       failA
        sll     $0, $0, 0
# tested basic function of or and ori, tested beq for branch to correct address.

CaseB:
        or      $0, $v1, $v0
        beq     $0, $t1, CaseC
        sll     $0, $0, 0
        j       failB
        sll     $0, $0, 0
# tested register zero by trying to or a value into the register.

CaseC:
        addiu   $v0, $0, 54
        addu    $v1, $v0, $0
        bne     $v1, $v0, failC1
        subu    $t1, $v1, $v0
        beq     $t1, $0, CaseD
        sll     $0, $0, 0
        j       failC2
        sll     $0, $0, 0
# tested basic function of addu, subu, addiu.

CaseD:
        addiu   $v0, $0, 64
        addiu   $v1, $0, 80
        and     $t2, $v1, $v0
        bne     $t2, $v0, failD1
        sll     $0, $0, 0
        addiu   $v0, $0, 170
        xor     $t3, $t2, $v0
        addu    $v1, $t2, $v0
        bne     $t3, $v1, failD2
        sll     $0, $0, 0
        xori    $t4, $t3, 255
        addiu   $v0, $0, 21
        bne     $t4, $v0, failD3
        sll     $0, $0, 0
        andi    $t5, $v0, 255
        bne     $t5, $v0, failD4
        sll     $0, $0, 0
        lui     $at, 64
        sll     $0, $0, 0
        ori     $v0, $at, 240
```

```
        addiu   $t6, $at, 240
        bne     $t6, $v0, failD5
        addiu   $v0, $0, 180
        jr      $v0
        sll     $0, $0, 0
        addiu   $v0, $0, 240
# tested basic functions of and, xor, xori, andi, lui.  Also tested if jr would jump to the correct address.

CaseE:
        addiu   $v1, $0, 180
        bne     $v1, $v0, failE1
        sll     $0, $0, 0
        addiu   $v0, $v1, 2
        beq     $v1, $v0, failE2
        sll     $0, $0, 0
        j       CaseF
        addiu   $v1, $0, 60
        lui     $at, 64
        sll     $0, $0, 0
        ori     $v0, $at, 208
        jr      $v0
        addiu   $at, $0, 77
#tested for delay slot for jr, beq functionality and started test for j delay slot.  From the jr in CaseD, it
#jumped to the lui instruction.  To test for the delay slot: set up an instruction to change a register where
#the nop should be and after it jumps, change another register and compare it to see if it was the same.

CaseF:
        addiu   $t7, $0, 60
        bne     $t7, $v1, failF
        sll     $0, $0, 0
        beq     $0, $0, CaseG
        addiu   $v0, $0, 125
        addiu   $at, $0, 60
#finish test for j delay slot and start test for beq branch location and delay slot.

CaseG:
        beq     $at, $t7, failG1
        sll     $0, $0, 0
        addiu   $v1, $0, 125
        bne     $v1, $v0, failG2
        sll     $0, $0, 0
        bne     $v1, $0, CaseH
        addiu   $v0, $0, 47
        addiu   $t2, $0, 88
#finish test for beq location and delay slot and start test for bne branch location and delay slot.  The first
#test is to make sure that beq actually branched off as the addiu $at, $0, 60 instruction in CaseF should
#never be executed.  The second test checks the delay slot as the same logic as written in CaseE test.

CaseH:
        addiu   $t3, $0, 88
        beq     $t3, $t2, failH1
        sll     $0, $0, 0
        addiu   $v1, $0, 47
        bne     $v1, $v0, failH2
        sll     $0, $0, 0
        jal     CaseI
```

```
        addiu   $v0, $0, 99
        j       endoftest
#finish test for bne location and delay slot and start test for jal jump location, delay slot and return address.
#To test for the delay slot: set up an instruction to change a register where the nop should be and after it
#branches, change another register and compare it to see if it was the same.

CaseI:
        addiu   $v1, $0, 99
        bne     $v1, $v0, failI1
        sll     $0, $0, 0
        lui     $at, 64
        sll     $0, $0, 0
        ori     $v0, $0, 0x0140
        bne     $v0, $ra, failI2
        sll     $0, $0, 0
#finish test for jal location, delay slot and return address.  If jal did not branch then the j endoftest would
#have executed.  The first test here looks to see if the registers are the same for the delay slot.  The second
#test shows functionality of the return address.
        addiu   $v1, $0, 15
        sll     $v0, $v1, 16
        addiu   $t4, $0, 0
        lui     $t4, 15
        sll     $0, $0, 0
        bne     $t4, $v0, failI3
        sll     $0, $0, 0
        srl     $v0, $v0, 8
        addiu   $t5, $0, 3840
        bne     $t5, $v0, failI4
        sll     $0, $0, 0
        sra     $v1, $v0, 8
        addiu   $t6, $0, 15
        bne     $t6, $v1, failI5
        sll     $0, $0, 0
        sll     $v1, $v1, 31
        sra     $v1, $v1, 5
        lui     $at, 64512
        sll     $0, $0, 0
        bne     $v1, $at, failI6
        sll     $0, $0, 0
# tested sll, srl, and sra to show shifting instructions work correctly and extend the MSB appropriately.
        addiu   $t7, $0, 48
        addiu   $t6, $0, 0
        lui     $t7, 4
        sll     $0, $0, 0
        lui     $t6, 4
        sll     $0, $0, 0
        bne     $t6, $t7, failI7
        sll     $0, $0, 0
#tested lui extend.
        addu    $t2, $0, $0
        addu    $t3, $0, $0
        addiu   $t2, $0, -42
        addiu   $t3, $0, 42
        addu    $t4, $t3, $t2
        bne     $t4, $0, failI8
        sll     $0, $0, 0
```

```
        j       CaseJ
        sll     $0, $0, 0
#tested addiu sign extend.

CaseL:
        addiu   $v0, $0, 27
        sw      $v0,20,($0)
        lw      $t3,20,($0)
        bne     $t3, $v0, failL1
        sll     $0, $0, 0
        addiu   $v1, $0, 44
        sw      $v1,8,($0)
        sll     $0, $0, 0
        sll     $0, $0, 0
        lw      $t4,8,($0)
        bne     $t4, $v1, failL2
        sll     $0, $0, 0
        j       finishtest
        sll     $0, $0, 0
#tested sw and lw.  Makes sure that a load that follows a store to the same address reads the appropriate
#data

CaseK:
        addiu   $v0, $0, 15
        sll     $v0, $v0, 28
        sra     $v0, $v0, 28
        addiu   $v1, $v0, 1
        bne     $v1, $0, failK1
        sll     $0, $0, 0
        addiu   $t4, $0, 15
        addu    $t4, $t4, $v0
        addiu   $t5, $0, 14
        bne     $t5, $t4, failK2
        sll     $0, $0, 0
        subu    $t6, $t4, $v0
        addiu   $t5, $0, 15
        bne     $t5, $t6, failK3
        sll     $0, $0, 0
#tested for overflow of  arithmetic and logical instructions.
        addiu   $v0, $0, 32767
        andi    $v1, $v0, 255
        addiu   $t3, $0, 255
        bne     $t3, $v1, failK4
        sll     $0, $0, 0
        xori    $t4, $t3, 0
        bne     $t4, $t3, failK5
        sll     $0, $0, 0
        jal     CaseL
        sll     $0, $0, 0
        j       endoftest
        sll     $0, $0, 0
#tested andi and xori for extend.  This completes the tests for arithmetic and logical instructions with
#immediate field extends.


CaseJ:
```

```
        addiu   $v0, $0, 15
        sll     $v0, $v0, 28
        sra     $v0, $v0, 28
        addiu   $v1, $0, 52
        slt     $t2, $v0, $v1
        addiu   $t1, $0, 1
        bne     $t1, $t2, failJ1
        sll     $0, $0, 0
        slti    $t3, $v0, 138
        bne     $t1, $t3, failJ2
        sll     $0, $0, 0
        slt     $t4, $v1, $v0
        beq     $t4, $t1, failJ3
        sll     $0, $0, 0
        slti    $t5, $v1, -294
        beq     $t5, $t1, failJ4
        sll     $0, $0, 0
        addiu   $v0, $0, 14296
        addiu   $v1, $v0, 3920
        sll     $v0, $v0, 4
        slt     $t6, $v1, $v0
        bne     $t1, $t6, failJ5
        sll     $0, $0, 0
        slti    $t7, $v0, 5333
        beq     $t1, $t7, failJ6
        sll     $0, $0, 0
        sltu    $t2, $v1, $v0
        bne     $t1, $t2, failJ7
        sll     $0, $0, 0
        sltiu   $t3, $v0, 5411
        beq     $t1, $t3, failJ8
        sll     $0, $0, 0
        j       CaseK
        sll     $0, $0, 0
#tested slt, slti, stliu, and stlu.  These 8 tests look at both bit patterns representing negative two's
#complement integers and large natural numbers comparisons. Also tested jumping backwards.

#following are error codes to see where an error occurs
failA:
        addiu   $t0, $0, 1
        j       endoftest
        sll     $0, $0, 0

failB:
        addiu   $t0, $0, 2
        j       endoftest
        sll     $0, $0, 0

failC1:
        addiu   $t0, $0, 3
        j       endoftest
        sll     $0, $0, 0

failC2:
        addiu   $t0, $0, 4
        j       endoftest
```

```
        sll     $0, $0, 0


failD1:
        addiu   $t0, $0, 5
        j       endoftest
        sll     $0, $0, 0

failD2:
        addiu   $t0, $0, 6
        j       endoftest
        sll     $0, $0, 0

failD3:
        addiu   $t0, $0, 7
        j       endoftest
        sll     $0, $0, 0

failD4:
        addiu   $t0, $0, 8
        j       endoftest
        sll     $0, $0, 0

failD5:
        addiu   $t0, $0, 9
        j       endoftest
        sll     $0, $0, 0

failE1:
        addiu   $t0, $0, 10
        j       endoftest
        sll     $0, $0, 0

failE2:
        addiu   $t0, $0, 11
        j       endoftest
        sll     $0, $0, 0

failF:
        addiu   $t0, $0, 12
        j       endoftest
        sll     $0, $0, 0

failG1:
        addiu   $t0, $0, 13
        j       endoftest
        sll     $0, $0, 0

failG2:
        addiu   $t0, $0, 14
        j       endoftest
        sll     $0, $0, 0

failH1:
        addiu   $t0, $0, 15
        j       endoftest
```

```
        sll     $0, $0, 0

failH2:
        addiu   $t0, $0, 16
        j       endoftest
        sll     $0, $0, 0

failI1:
        addiu   $t0, $0, 17
        j       endoftest
        sll     $0, $0, 0

failI2:
        addiu   $t0, $0, 18
        j       endoftest
        sll     $0, $0, 0

failI3:
        addiu   $t0, $0, 19
        j       endoftest
        sll     $0, $0, 0

failI4:
        addiu   $t0, $0, 20
        j       endoftest
        sll     $0, $0, 0

failI5:
        addiu   $t0, $0, 21
        j       endoftest
        sll     $0, $0, 0

failI6:
        addiu   $t0, $0, 22
        j       endoftest
        sll     $0, $0, 0

failI7:
        addiu   $t0, $0, 23
        j       endoftest
        sll     $0, $0, 0

failI8:
        addiu   $t0, $0, 24
        j       endoftest
        sll     $0, $0, 0

failJ1:
        addiu   $t0, $0, 25
        j       endoftest
        sll     $0, $0, 0

failJ2:
        addiu   $t0, $0, 26
        j       endoftest
        sll     $0, $0, 0
```

```
failJ3:
        addiu   $t0, $0, 27
        j       endoftest
        sll     $0, $0, 0

failJ4:
        addiu   $t0, $0, 28
        j       endoftest
        sll     $0, $0, 0

failJ5:
        addiu   $t0, $0, 29
        j       endoftest
        sll     $0, $0, 0

failJ6:
        addiu   $t0, $0, 30
        j       endoftest
        sll     $0, $0, 0

failJ7:
        addiu   $t0, $0, 31
        j       endoftest
        sll     $0, $0, 0

failJ8:
        addiu   $t0, $0, 32
        j       endoftest
        sll     $0, $0, 0

failK1:
        addiu   $t0, $0, 33
        j       endoftest
        sll     $0, $0, 0


failK2:
        addiu   $t0, $0, 34
        j       endoftest
        sll     $0, $0, 0

failK3:
        addiu   $t0, $0, 35
        j       endoftest
        sll     $0, $0, 0

failK4:
        addiu   $t0, $0, 36
        j       endoftest
        sll     $0, $0, 0

failK5:
        addiu   $t0, $0, 37
        j       endoftest
        sll     $0, $0, 0
```

```
failL1:
        addiu   $t0, $0, 38
        j       endoftest
        sll     $0, $0, 0

failL2:
        addiu   $t0, $0, 39
        j       endoftest
        sll     $0, $0, 0

finishtest:
        addiu   $v0, $0, 15
        sll     $v0, $v0, 28
        sra     $v0, $v0, 28
        addu    $v1, $v0, $0
        addu    $t0, $v0, $0

endoftest:
        sll $0, $0, 0
```

Basic Test:

```
main:
test0:
        addiu   $2, $0, 5       #testing addiu
        addiu   $3, $0, 5       #testing addiu
        addiu   $4, $0, -5
        addiu   $23, $0, 0      #used to keep track of failed tests

        beq     $2, $2, L1      #testing beq
        ori     $0, $0, 0
        addiu   $2, $2, -5
L1:
        bne     $3, $0, L2      #testing bne
        ori     $0, $0, 0
        addiu   $3, $3, -5
L2:
        beq     $2, $0, test0_fail
        ori     $0, $0, 0
        bne     $2, $3, test0_fail
        ori     $0, $0, 0
        beq     $2, $0, test0_fail
        ori     $0, $0, 0
        bne     $2, $2, test0_fail
        ori     $0, $0, 0
        ori     $0, $0, 0

        lui     $5, 0xFFFF
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $5, $5, 0xFFFB
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $4, $5, test0_fail
```

```
        ori     $0, $0, 0
        j test1
        ori     $0, $0, 0#NOP
test0_fail:
        ori     $23, $23, 0x1
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        addiu   $23, $23, 0
        break 2

test1: #test addu, addiu, subu
        addiu   $6, $0, 1       #$6 = small number
        addiu   $7, $0, 52      #$7 = another number
        lui     $8, 0x8000      #$8 = smallest negative number (0x80000000)
        addiu   $9, $0, -5      #$9 = negative number
        addiu   $10, $0, -48    #$10 = negative number
        ori     $0, $0, 0
        addiu   $11, $8, -1     #$11 = largest number(0x7FFFFFFF)

        addu    $12, $6, $7     #testing addu with 2 positive numbers
        addiu   $13, $0, 53
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $12, $13, test1_fail

        addu    $14, $9, $10    #testing addu for adding 2 negative numbers
        addiu   $15, $0, -53
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $14, $15, test1_fail

        addu    $16, $6, $9     #test addu with one pos and one neg number
        addiu   $17, $0, -4
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $16, $17, test1_fail

        addiu   $18, $6, 52     #testing addiu with 2 positive numbers
        addiu   $19, $0, 53
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $18, $19, test1_fail

        addiu   $20, $9, -48    #testing addiu for adding 2 negative numbers
        addiu   $21, $0, -53
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $20, $21, test1_fail

        addiu   $24, $6, -5     #test addiu with one pos and one neg number
```

```
        addiu   $25, $0, -4
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $24, $25, test1_fail

        subu    $26, $6, $7         #testing subu with 2 positive numbers
        addiu   $27, $0, -51
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $26, $27, test1_fail

        subu    $2, $9, $10         #testing subu for adding 2 negative numbers
        addiu   $3, $0, 43
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $2, $3, test1_fail

        subu    $4, $6, $9          #test subu with one pos and one neg number
        addiu   $5, $0, 6
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $4, $5, test1_fail
        ori     $0, $0, 0
        j test2
        ori     $0, $0, 0#NOP
test1_fail:
        ori     $23, $23, 0x2
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        addiu   $23, $23, 0
        break 2

test2:  #test and, andi, or, ori
        addiu   $2, $0, 0x5A5A          #$2 = 0x00005A5A
        addiu   $3, $0, 0x2525          #$3 = 0x00002525
        ori     $4, $0, 0xFFFF          #$4 = 0x0000FFFF
        ori     $0, $0, 0
        ori     $0, $0, 0

        and     $5, $2, $3              #test and
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $5, $0, test2_fail

        lui     $6, 0x6789              #test and for no sign extension
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        and     $7, $6, $4
        ori     $0, $0, 0
```

```
ori     $0, $0, 0
ori     $0, $0, 0
bne     $7, $0, test2_fail

andi    $8, $2, 0x2525          #test andi
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $8, $0, test2_fail

lui     $9, 0x6789              #test andi for no sign extension
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
andi    $10, $9, 0xFFFF
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $10, $0, test2_fail

or      $11, $2, $3             #test or
addiu   $12, $0, 0x7F7F
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $11, $12, test2_fail

lui     $13, 0x6789             #test or for no sign extension
addiu   $14, $0, -1
ori     $0, $0, 0
or      $13, $13, $4
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
beq     $13, $14, test2_fail

ori     $15, $2, 0x2525         #test ori
addiu   $16, $0, 0x7F7F
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $15, $16, test2_fail

lui     $17, 0x6789             #test ori for no sign extension
addiu   $18, $0, -1
ori     $0, $0, 0
ori     $0, $0, 0
ori     $17, $17, 0xFFFF
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
beq     $17, $18, test2_fail

ori     $0, $0, 0#NOP
j test3
ori     $0, $0, 0#NOP
```

```
test2_fail:
        ori     $23, $23, 0x4
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        addiu   $23, $23, 0
        break 2

test3:  #test xor, xori, lui
        addiu   $2, $0, 0x0F0F          #$2 = 0x00000F0F
        ori     $3, $0, 0xFF00          #$3 = 0x0000FF00
        ori     $4, $0, 0xF00F          #$4 = 0x0000F00F
        ori     $0, $0, 0
        ori     $0, $0, 0

        xor     $5, $2, $3              #testing xor
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $5, $4, test3_fail

        xori    $6, $3, 0x0F0F          #testing xori
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $6, $4, test3_fail

        xori    $7, $2, 0xFF00          #testing xori for no sign extension
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $7, $4, test3_fail

        addiu   $8, $0, 0x0001          #testing lui, especially for setting last 16bits to 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        lui     $8, 0xFFFF
        addiu   $9, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        lui     $9, 0xFFFF
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $8, $9, test3_fail

        ori     $0, $0, 0#NOP
        j test4
        ori     $0, $0, 0#NOP
test3_fail:
        ori     $23, $23, 0x8
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
```

```
        addiu   $23, $23, 0
        break 2

test4:  #test sll, sra, srl
        addiu   $2, $0, 0x5A5A          #$2 = 0x00005A5A
        addiu   $3, $0, 0x2525          #s1 = 0x00002525
        ori     $0, $0, 0
        ori     $0, $0, 0

        sll     $4, $2, 16              #testing sll
        addiu   $5, $0, 0
        lui     $5, 0x5A5A
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $4, $5, test4_fail

        #sll    $6, $2, -5              #testing sll with negative shift
        #addiu  $7, $0, 0
        #lui    $7, 0xD000
        #ori    $0, $0, 0
        #ori    $0, $0, 0
        #ori    $0, $0, 0
        #bne    $6, $7, test4_fail

        sll     $8, $4, 16              #testing that the bits at the end fall off
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $8, $0, test4_fail

        lui     $9, 0x0FFF              #test sra
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        sra     $10, $9, 16
        ori     $11, $0, 0x0FFF
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $10, $11, test4_fail

        lui     $12, 0xFFFF             #test sra for sign extension
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        sra     $13, $12, 16
        addiu   $14, $0, -1
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $13, $14, test4_fail

        sra     $15, $10, 16            #testing that the bits at end fall off
        ori     $0, $0, 0
        ori     $0, $0, 0
```

```
        ori     $0, $0, 0
        bne     $15, $0, test4_fail

        lui     $16, 0x0FFF             #test srl
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        srl     $17, $16, 16
        ori     $18, $0, 0x0FFF
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $17, $18, test4_fail

        lui     $19, 0xFFFF             #test srl for no sign extension
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        srl     $20, $19, 16
        ori     $21, $0, 0xFFFF
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $20, $21, test4_fail

        srl     $22, $20, 16            #testing that the bits at end fall off
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $22, $0, test4_fail

        ori     $0, $0, 0#NOP
        j test5
        ori     $0, $0, 0#NOP
test4_fail:
        ori     $23, $23, 0x10
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        addiu   $23, $23, 0
        break 2

test5: #testing slt, sltu
        addiu   $2, $0, 1       #$2 = small number
        addiu   $3, $0, 52      #$3 = another number
        lui     $4, 0x8000      #$4 = smallest negative number (0x80000000)
        addiu   $5, $0, -5      #$5 = negative number
        addiu   $6, $0, -48     #$6 = negative number
        ori     $0, $0, 0
        addiu   $7, $4, -1      #$7 = largest number(0x7FFFFFFF)

        slt     $8, $2, $3      #testing slt with 2 pos num
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        beq     $8, $0, test5_fail
```

```
slt     $9, $3, $2      #with 2 pos num, reg reversed
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $9, $0, test5_fail

slt     $10, $5, $6     #with 2 neg num
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $10, $0, test5_fail

slt     $11, $6, $5     #with 2 neg num, reg reversed
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
beq     $11, $0, test5_fail

slt     $12, $3, $6     #with one pos and one neg number
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $12, $0, test5_fail

slt     $13, $6, $3     #with one pos and one neg num, reg reversed
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
beq     $13, $0, test5_fail

slt     $14, $6, $6     #test if both reg are equal
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $14, $0, test5_fail

slt     $15, $4, $4     #test if both reg are equal
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $15, $0, test5_fail

slt     $16, $4, $2     #test for ovf condition
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
beq     $16, $0, test5_fail

slt     $17, $7, $5     #test for ovf condition
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $17, $0, test5_fail

sltu    $18, $2, $3     #testing sltu with 2 pos num
```

```
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        beq     $18, $0, test5_fail

        sltu    $19, $3, $2     #with 2 pos num, reg reversed
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $19, $0, test5_fail

        sltu    $20, $6, $6     #test if both reg are equal
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $20, $0, test5_fail

        sltu    $21, $4, $4     #test if both reg are equal
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $21, $0, test5_fail

        sltu    $22, $4, $2     #test for ovf condition
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $22, $0, test5_fail

        sltu    $24, $7, $5     #test for ovf condition
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        beq     $24, $0, test5_fail

        sltu    $25, $7, $4
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        beq     $25, $0, test5_fail
        ori     $0, $0, 0#NOP
        j test6
        ori     $0, $0, 0#NOP
test5_fail:
        ori     $23, $23, 0x20
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        addiu   $23, $23, 0
        break 2
test6:  #testing slti, sltiu
        slti    $8, $2, 52      #testing slti with 2 pos num
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        beq     $8, $0, test6_fail
```

```
slti    $9, $3, 1       #with 2 pos num, reg reversed
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $9, $0, test6_fail

slti    $10, $5, -48    #with 2 neg num
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $10, $0, test6_fail

slti    $11, $6, -5     #with 2 neg num, reg reversed
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
beq     $11, $0, test6_fail

slti    $12, $3, -48    #with one pos and one neg number
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $12, $0, test6_fail

slti    $13, $6, 52     #with one pos and one neg num, reg reversed
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
beq     $13, $0, test6_fail

slti    $14, $6, -48    #test if both reg are equal
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $14, $0, test6_fail

slti    $16, $4, 1      #test for ovf condition
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
beq     $16, $0, test6_fail

slti    $17, $7, -5     #test for ovf condition
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
bne     $17, $0, test6_fail

sltiu   $18, $2, 52     #testing sltiu with 2 pos num
ori     $0, $0, 0
ori     $0, $0, 0
ori     $0, $0, 0
beq     $18, $0, test6_fail

sltiu   $19, $3, 1      #with 2 pos num, reg reversed
```

```
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $19, $0, test6_fail

        sltiu   $20, $6, -48        #test if both reg are equal
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $20, $0, test6_fail

        sltiu   $22, $4, 1          #test for ovf condition
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $22, $0, test6_fail

        sltiu   $24, $7, -5         #test for ovf condition
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        beq     $24, $0, test6_fail

        ori     $0, $0, 0           #NOP
        j test7
        ori     $0, $0, 0           #NOP
test6_fail:
        ori     $23, $23, 0x40
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        addiu   $23, $23, 0
        break 2

test7: #test lw, sw
        addiu   $29, $0, 0x0100
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        addiu   $29, $29, -8
        addiu   $2, $0, 0x5A5A
        addiu   $3, $0, 0xFFFF
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0

        sw      $2, 0($29);
        sw      $3, 4($29);
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        lw      $4, 0($29);
        lw      $5, 4($29);
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
```

```
        bne     $2, $4, test7_fail
        ori     $0, $0, 0
        bne     $3, $5, test7_fail
        ori     $0, $0, 0

        sw      $3, 0($29);
        lw      $4, 0($29);
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        bne     $4, $3, test7_fail

        addiu   $29, $29, 8
        ori     $0, $0, 0           #NOP
        j test8
        ori     $0, $0, 0           #NOP
test7_fail:
        ori     $23, $23, 0x80
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        addiu   $23, $23, 0
        break 2
test8:  #bgez, bltz, j, jr, jal
        addiu   $2, $0, 5
        addiu   $3, $0, -5
        addiu   $4, $0, 4
        addiu   $5, $0, 5
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0

        bgez    $2, L3              #test bgez
        ori     $0, $0, 0
        j       test8_fail
        ori     $0, $0, 0
L3:
        bgez    $0, L4
        ori     $0, $0, 0
        j       test8_fail
        ori     $0, $0, 0
L4:
        bgez    $3, L5
        ori     $0, $0, 0
        j       L6
        ori     $0, $0, 0
L5:
        sll     $0, $0, 1
        j       test8_fail
        ori     $0, $0, 0
L6:
        bltz    $3, L7              #test bltz
        ori     $0, $0, 0
        j       test8_fail
        ori     $0, $0, 0
L7:
```

```
        bltz    $0, L8
        ori     $0, $0, 0
        j       L9
        ori     $0, $0, 0
L8:
        j       test8_fail
        ori     $0, $0, 0
L9:
        bltz    $2, L10
        ori     $0, $0, 0
        j       L11
        ori     $0, $0, 0
L10:
        j       test8_fail
        ori     $0, $0, 0
L11:
        jal     test8_2
        addiu   $4, $0, 10

        addiu   $4, $0, 5       #return from jr, make $4 = 5
        beq     $4, $5, L12     #make sure we JRed to the right place
        ori     $0, $0, 0       #NOP

test8_fail:
        ori     $23, $23, 0x100
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        addiu   $23, $23, 0
        break 2
L12:
        j done
        ori     $0, $0, 0       #NOP
done:
        j       exit
        ori     $0, $0, 0
test8_2:
        sll     $0, $0, 0
        sll     $0, $0, 0
        sll     $0, $0, 0
        jr      $31
        ori     $0, $0, 0
        j       test8_fail
        ori     $0, $0, 0
exit:
        addiu   $23, $23, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
        sw      $23, 16($0)
        ori     $0, $0, 0
        ori     $0, $0, 0
        ori     $0, $0, 0
exit2:
        break 3
        ori     $0, $0, 0
```

```
break 4
ori     $0, $0, 0
sll   $0, $0, 0
j       exit2
ori     $0, $0, 0
```

Mul Test:

```
ori $1, $0, 0x2          #$1 = 2
mul $2, $1, $1           #$2 = 4
ori $3, $0, 0x4          #$3 = 4
bne $2, $3, fail1
ori $4, $0, 0x2          #$4 = 2
ori $1, $0, 0x8          #$1 = 8
mul $5, $4, $3           #$5 = 8
mul $6, $5, $2           #$6 = 32, test mul hazard
bne $5, $1, fail2        #test mul result forward
ori $2, $0, 0x20         #$2 = 32
bne $6, $2, fail3
ori $5, $0, 0x1111       #$5 = 4369
mul $6, $5, $5           #$6 = 19088161
lui $2, $0, 0x0123
ori $2, $2, 0x4321       #$2 = 19088161
bne $2, $6, fail4
ori $0, $0, 0xffff
ori $0, $0, 0x0fff
ori $0, $0, 0x00ff
j passedtest
ori $0, $0, 0x000f

fail1:
break 1
fail2:
break 2
fail3:
break 3
fail4:
break 4
passedtest:
break 15
```

SIMD Test:

```
vlw $1, 0($0)            #load data from memories into register files
vlw $2, 4($0)
vlw $3, 8($0)
vlw $4, 12($0)
vlw $5, 16($0)
vlw $6, 20($0)
vlw $7, 24($0)
vlw $8, 28($0)
vlw $9, 32($0)
vlw $10, 36($0)
vlw $11, 40($0)
```

```
vlw $12, 44($0)
vlw $13, 48($0)
vlw $14, 52($0)
vlw $15, 56($0)
vlw $16, 60($0)
vlw $17, 64($0)
vlw $18, 68($0)
vlw $19, 72($0)
vaddu $23, $1, $2          #test vaddu
vaddu $24, $23, $3
vaddu $25, $24, $23
vsw $25, 0($0)
break 1
vlw $20, 0($0)             #test vlw hazard
vaddiu $21, $20, 0x1111    #test vaddiu
vaddiu $22, $0, 0x1000
vsw $23, 0($22)            #test vsw and vlw
vsw $24, 8($22)
vsw $25, 24($22)
vlw $23, 8($22)
vlw $24, 24($22)
vlw $25, 0($22)
vandi $24, $8, 0x8888      #test vandi
break 2
vori $23, $24, 0x8888      #test vori
vxor $23, $23, $9          #test vxor
vor $23, $23, $10          #test vor
break 3
break 4
break 5
vxori $23, $23, 0x0765     #test vori
vand $23, $18, $23         #test vand
vaddiu $23, $23, 0x8000
break 6
break 7
break 8
vor $23, $16, $7
vmul $23, $23, $6          #test vmul
vmul $23, $23, $17         #test vmul hazard
break 9
break 10
break 11
```

Appendix D: FPGA User Constraint File

FPGA_TOP.ucf

```
##NET "SYS_CLK" LOC = "AJ15";
##NET "SYS_CLK" IOSTANDARD = LVCMOS25;
##NET "SYS_CLK" TNM_NET = "SYS_CLK";
##TIMESPEC "TS_SYS_CLK" = PERIOD "SYS_CLK" 10.00 ns HIGH 50 %;

NET "SYS_CLK" LOC = "AH15";
NET "SYS_CLK" IOSTANDARD = LVCMOS25;
NET "SYS_CLK" TNM_NET = "SYS_CLK";
TIMESPEC "TS_SYS_CLK" = PERIOD "SYS_CLK" 31.25 ns HIGH 50 %;

NET "LED_0" LOC = "AC4";
NET "LED_1" LOC = "AC3";
NET "STALL" LOC = "AA6";
NET "LED_3" LOC = "AA5";

##NET "LED_0" IOSTANDARD = LVTTL;
##NET "LED_1" IOSTANDARD = LVTTL;
##NET "STALL" IOSTANDARD = LVTTL;
##NET "LED_3" IOSTANDARD = LVTTL;

##NET "LED_0" DRIVE = 12;
##NET "LED_1" DRIVE = 12;
##NET "STALL" DRIVE = 12;
##NET "LED_3" DRIVE = 12;

##NET "LED_0" SLEW = SLOW;
##NET "LED_1" SLEW = SLOW;
##NET "STALL" SLEW = SLOW;
##NET "LED_3" SLEW = SLOW;

NET "DIPSWITCH<0>" LOC = "AC11";
NET "DIPSWITCH<1>" LOC = "AD11";
NET "DIPSWITCH<2>" LOC = "AF8";
NET "DIPSWITCH<3>" LOC = "AF9";

##NET "DIPSWITCH<0>" IOSTANDARD = LVCMOS25;
##NET "DIPSWITCH<1>" IOSTANDARD = LVCMOS25;
##NET "DIPSWITCH<2>" IOSTANDARD = LVCMOS25;
##NET "DIPSWITCH<3>" IOSTANDARD = LVCMOS25;

##NET "PB_ENTER" LOC = "AG5";
##NET "PB_UP" LOC = "AH4";
NET "SINGLE_CLK_low" LOC = "AG3";
NET "RESET_low" LOC = "AH1";
NET "RELEASE_low" LOC = "AH2";

##NET "PB_ENTER" IOSTANDARD = LVTTL;
```

```
##NET "PB_UP" IOSTANDARD = LVTTL;
##NET "SINGLE_CLK_" IOSTANDARD = LVCMOS25;
##NET "RESET_" IOSTANDARD = LVTTL;
##NET "RELEASE_" IOSTANDARD = LVTTL;

NET "EXP_IO_0" LOC = "K2";
NET "EXP_IO_1" LOC = "L2";
NET "EXP_IO_2" LOC = "N8";
NET "EXP_IO_3" LOC = "N7";
NET "PC_DISPLAY<31>" LOC = "K4";
NET "PC_DISPLAY<30>" LOC = "K3";
NET "PC_DISPLAY<29>" LOC = "L1";
NET "PC_DISPLAY<28>" LOC = "M1";
NET "PC_DISPLAY<27>" LOC = "N6";
NET "PC_DISPLAY<26>" LOC = "N5";
NET "PC_DISPLAY<25>" LOC = "L5";
NET "PC_DISPLAY<24>" LOC = "L4";
NET "PC_DISPLAY<23>" LOC = "M2";
NET "PC_DISPLAY<22>" LOC = "N2";
NET "PC_DISPLAY<21>" LOC = "P9";
NET "PC_DISPLAY<20>" LOC = "R9";
NET "PC_DISPLAY<19>" LOC = "M4";
NET "PC_DISPLAY<18>" LOC = "M3";
NET "PC_DISPLAY<17>" LOC = "N1";
NET "PC_DISPLAY<16>" LOC = "P1";
NET "EXP_IO_20" LOC = "P8";
NET "EXP_IO_21" LOC = "P7";
NET "EXP_IO_22" LOC = "N4";
NET "EXP_IO_23" LOC = "N3";
NET "PC_DISPLAY<15>" LOC = "P3";
NET "PC_DISPLAY<14>" LOC = "P2";
NET "PC_DISPLAY<13>" LOC = "R8";
NET "PC_DISPLAY<12>" LOC = "R7";
NET "PC_DISPLAY<11>" LOC = "P5";
NET "PC_DISPLAY<10>" LOC = "P4";
NET "PC_DISPLAY<9>" LOC = "R2";
NET "PC_DISPLAY<8>" LOC = "T2";
NET "PC_DISPLAY<7>" LOC = "R6";
NET "PC_DISPLAY<6>" LOC = "R5";
NET "PC_DISPLAY<5>" LOC = "R4";
NET "PC_DISPLAY<4>" LOC = "R3";
NET "PC_DISPLAY<3>" LOC = "U1";
NET "PC_DISPLAY<2>" LOC = "V1";
NET "PC_DISPLAY<1>" LOC = "T5";
NET "PC_DISPLAY<0>" LOC = "T6";
NET "EXP_IO_40" LOC = "T3";
NET "EXP_IO_41" LOC = "T4";
NET "EXP_IO_42" LOC = "U2";
NET "EXP_IO_43" LOC = "U3";
NET "INST_STAT_DISPLAY<31>" LOC = "T7";
NET "INST_STAT_DISPLAY<30>" LOC = "T8";
NET "INST_STAT_DISPLAY<29>" LOC = "U4";
NET "INST_STAT_DISPLAY<28>" LOC = "U5";
NET "INST_STAT_DISPLAY<27>" LOC = "V2";
NET "INST_STAT_DISPLAY<26>" LOC = "W2";
NET "INST_STAT_DISPLAY<25>" LOC = "T9";
```

```
NET "INST_STAT_DISPLAY<24>" LOC = "U9";
NET "INST_STAT_DISPLAY<23>" LOC = "V3";
NET "INST_STAT_DISPLAY<22>" LOC = "V4";
NET "INST_STAT_DISPLAY<21>" LOC = "W1";
NET "INST_STAT_DISPLAY<20>" LOC = "Y1";
NET "INST_STAT_DISPLAY<19>" LOC = "U7";
NET "INST_STAT_DISPLAY<18>" LOC = "U8";
NET "INST_STAT_DISPLAY<17>" LOC = "V5";
NET "INST_STAT_DISPLAY<16>" LOC = "V6";
NET "EXP_IO_60" LOC = "Y2";
NET "EXP_IO_61" LOC = "AA2";
NET "EXP_IO_62" LOC = "V7";
NET "EXP_IO_63" LOC = "V8";
NET "INST_STAT_DISPLAY<15>" LOC = "W3";
NET "INST_STAT_DISPLAY<14>" LOC = "W4";
NET "INST_STAT_DISPLAY<13>" LOC = "AA1";
NET "INST_STAT_DISPLAY<12>" LOC = "AB1";
NET "INST_STAT_DISPLAY<11>" LOC = "W5";
NET "INST_STAT_DISPLAY<10>" LOC = "W6";
NET "INST_STAT_DISPLAY<9>" LOC = "Y4";
NET "INST_STAT_DISPLAY<8>" LOC = "Y5";
NET "INST_STAT_DISPLAY<7>" LOC = "AA3";
NET "INST_STAT_DISPLAY<6>" LOC = "AA4";
NET "INST_STAT_DISPLAY<5>" LOC = "W7";
NET "INST_STAT_DISPLAY<4>" LOC = "W8";
NET "INST_STAT_DISPLAY<3>" LOC = "AB3";
NET "INST_STAT_DISPLAY<2>" LOC = "AB4";
NET "INST_STAT_DISPLAY<1>" LOC = "AB2";
NET "INST_STAT_DISPLAY<0>" LOC = "AC2";
```

References

[1] P. J. Ashenden, The Designer's Guide to VHDL 2nd ed, Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[2] K. E. Batcher, Design of a Massively Parallel Processor, IEEE Trans. Computers, Vol c29, No 9, September 1980 p. 836-840.

[3] F. R. Boyer, J. Cloutier, et al, VIP: an FPGA-based Processor for Image Processing and Neural Networks, Proceedings of 5th International Conference on Microelectronics for Neural Networks, Lausanne, Switzerland, Feb 1996, p. 330-336.

[4] R. Cypher and J. L.C. Sanz, The SIMD Model of Parallel Computation. Springer-Verlag, New York, NY, 1994.

[5] D. M. Dahle, J. D. Hirschberg, et al, Kestrel: Design of an 8-bit SIMD Parallel Processor, Proceedings of 17th Conference on Advanced Research in VLSI, Ann Arbor, MI, Sept 1997, p. 145-162.

[6] A. DeHon and E. Mirsky, MATRIX: a Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources, IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, Apr 1996, p. 157-166.

[7] M. J. Duff, Real Applications on CLIP4, Integrated Technology for Parallel Image Processing, London, 1985, p.153-165.

[8] R. Duncan, A Survey of Parallel Computer Architectures, Computer, Vol 23, Issue 2, Feb 1990, p. 5-16.

[9] J. Fazekas, A. K. Jones, et al, An FPGA-based VLIW Processor with Custom Hardware Execution, ACM 13th International Symposium on Field Programmable Gate Arrays, Monterey, CA, 2005, p. 107-117.

[10] J. L. Hennessy and D. A. Patterson, Computer Architecture A Quantitative Approach 3rd ed, Morgan Kaufmann Publishers, San Francisco, CA, 2003.

[11] J. L. Hennessy and D. A. Patterson, Computer Organization and Design, Morgan Kaufmann Publishers, San Francisco, CA, 2005.

[12] Intel, Available: http://www.intel.com

[13] J. Lazzaro, MIPSASM, Available: http://inst.eecs.berkeley.edu/~cs152/mipsasm/

[14] MIPS Instruction Set Reference, Available: http://www.mips.com/products/resource_library/product_materials/MIPS_Architecture.php

[15] R. G. Nudd, R. A. Packwood, et al, WPM: a Multiple-SIMD Architecture for Image Processing, 3rd International Conference on Image Processing and its Applications, Jul 1989, p. 161-165.

[16] T. M. Silberberg, "The Hough Transform on the Geometric Arithmetic Parallel Processor" IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management, Nov 1985, p. 387-393.

[17] Wikipedia English Dictionary, Available: http://www.wikipedia.org

[18] Xilinx: MicroBlaze Architecture, Available: http://www.xilinx.com