Harnessing FPGAs for Computer Architecture Education

Mark Holland

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
In Electrical Engineering

University of Washington

2002

Program Authorized to Offer Degree: Electrical Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a master's thesis by

Mark Holland

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

____
Scott Hauck

____
Carl Ebeling

Date: ________________

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature________________________________

Date____________________________________

University of Washington

Abstract

Harnessing FPGAs for Computer Architecture Education

Mark Holland

Chair of the Supervisory Committee:
Associate Professor, Scott Hauck
Electrical Engineering

Computer architecture and design is often taught by having students use software to design and simulate individual pieces of a computer processor. We have developed a method that will take this classwork beyond software simulation into actual hardware implementation. Students will be able to design, implement, and run a single-cycle MIPS processor on an FPGA. This paper presents the major steps in this work: the FPGA implementation of a MIPS processor, a debugging tool which provides complete control and observability of the processor, the reduction of the MIPS instruction set into eight instructions that will be used by the processor, and a assembler that can map any MIPS non-floating point instruction into the set of eight supported instructions.

TABLE OF CONTENTS

List of Figures	iii
List of Tables	iv
Introduction	1
Related Work	3
FPGA Implementation of the Processor	5
Debugging Tool	9
Reduction of MIPS Instruction Set	12
Assembler	15
Classroom Integration	16
Results and Discussion	17
Conclusion	18
Current Status and Future Work	19
Notes	20
End Notes	21
Bibliography	22
Appendix A – Verilog Code for the Single Cycle Processor	23
Appendix B – Schematics for the Single Cycle Processor	44
Appendix C – Verilog Code for the Pipelined Processor	60

Appendix D – Schematics for the Pipelined Processor **79**

Appendix E – Debugging Interface Code **91**

Appendix F – Lex Assembler Code **118**

Appendix G – Yacc Assembler Code **134**

LIST OF FIGURES

Figure 1: Single Cycle MIPS Processor [5] **2**

Figure 2: The XSV Board Schematic [12] **5**

Figure 3: Register File Implementation **6**

Figure 4: (a) Read, (b) Write Waveforms for the SRAM **7**

Figure 5: Processor Timing Diagram **8**

Figure 6: The Debugging Interface **10**

LIST OF TABLES

Table 1: Debugger Commands and OP Fields 10

Table 2: Reduced MIPS Instruction Set 11

Table 3: Instruction Inflation Due to Reducing Set 13

Table 4: Implementation Results 16

Introduction

Computer Design and Organization is a common upper-level engineering course that is offered at universities throughout the world. In this class students often learn computer design by implementing individual pieces of a computer processor. This approach has important limitations: while students can complete and simulate their designs using software, they do not get the chance to implement and run their designs. Studies of engineering curriculums have shown that students learn better when given active, hands-on projects [1], [2], suggesting that students would thrive on the opportunity to implement and run their processors in hardware.

This paper presents a Field Programmable Gate Array (FPGA) implementation of a computer processor, a processor debugging tool, the reduction of the MIPS instruction set into eight instructions, and a assembler that performs the instruction reduction. The processor can be used to enhance computer architecture education as part of the following process: we will implement selected CPU parts onto the processor, give the incomplete processors to students, and allow the students to design and integrate the missing pieces. The processor debugging tool would provide the students with complete control over their processor, allowing them to debug and fix their designs. Reducing the MIPS instruction set to eight instructions allows us to use a simpler implementation of the processor, and it also simplifies the parts that the students will design. Finally, the assembler can reduce a file of any non-floating point/coprocessor instructions into our set of eight supported instructions, outputting a machine language file that can be loaded directly into the processor. Combined in the classroom, these tools allow students to implement and run their processor designs on an actual chip.

FPGAs are logic chips that can be programmed any number of times. The area of an FPGA is devoted largely to reprogrammable logic, but vendors have recently added sizable memories to their chips in order to give them a wider range of capabilities. In Xilinx Virtex devices [3] this memory is called Block Ram (BRAM) [4]. This BRAM provides for shallow RAM structures to be implemented on the FPGA.

The reprogrammable nature of FPGAs makes them perfect for educational purposes because they can be reused year after year, resulting in low overhead costs. In addition, the reprogrammable nature of FPGAs allows computer architecture students to have as many iterations as necessary when implementing their processors.

Our work is modeled on the single-cycle implementation of the MIPS processor, which is shown in Figure 1. We also have a pipelined implementation of the MIPS processor, but most of the paper will focus on the single-cycle implementation as it will be our primary instructional tool. Both processors operate on the MIPS instruction set as described in *Computer Organization and Design* by David A. Patterson and John L. Hennessy [5]. This book is widely used in the teaching of computer architecture. Using Patterson and

Hennessy as a basis for our processor means that the FPGA-implemented processor presented here will be easily incorporated into a large number of classrooms.

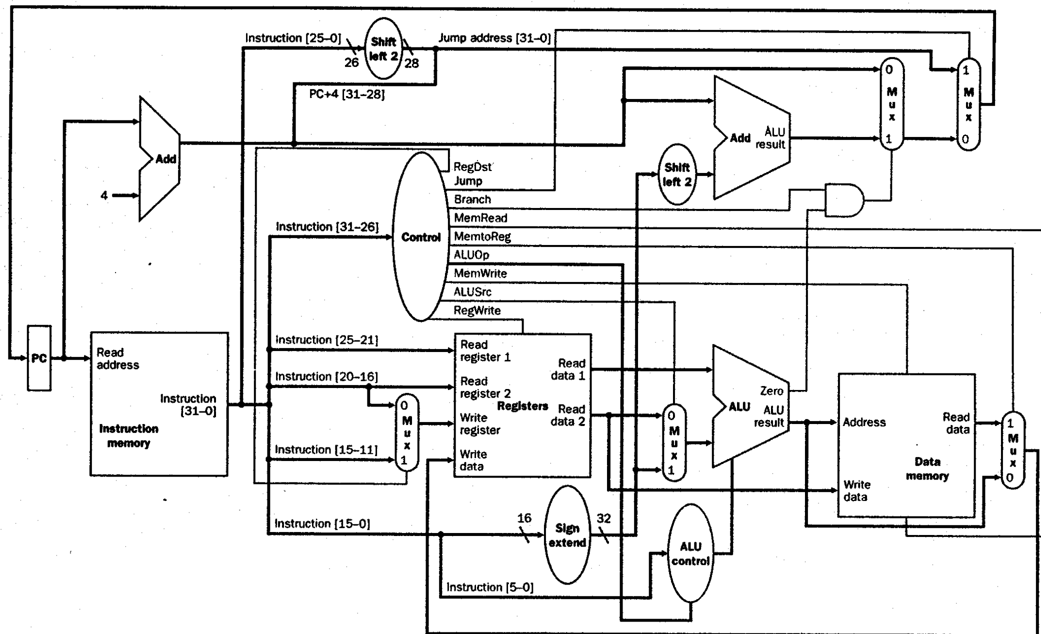


Figure 1: Single Cycle MIPS Processor [5]

Preliminary work on this project was conducted by Jered Aasheim during the summer of 2000. James Harris and Tim Midget were indispensable for their work on the reduction of the MIPS instruction set, as was Steve Detmer for working with and critiquing our beta version processor and debugging tool. Scott Hauck was supported by an NSF Career Award and a Sloan Foundation Fellowship, and Mark Holland was supported by an NSF Graduate Fellowship. Additionally, this project was funded in part by a donation from Xilinx.

Related Work

As early as 1990, researchers were considering how to increase the effectiveness of microprocessor architecture education. In *A reconfigurable microprocessor teaching tool* [6], Diab et al introduced a software package that could be used to help students understand the internal workings of a microprocessor. The tool aided students by walking them through a graphical demonstration of the internal behavior of a processor.

The package included an assembler, a control unit emulator, and a graphics simulator. The presence of the assembler allowed students to choose either assembly code or machine code to run on the mock processor. The graphics simulator would walk through the instructions and show the user how the instructions stimulated specific parts of the microprocessor, including the control logic created by the control unit emulator. Using this tool, the students could see how the different parts of the processor interact, and how they functionally combine into a working processor.

Our work is very much in the same flavor as the work done by Diab et al, but with the use of FPGAs we have been able to actually create a functional processor instead of simply displaying it graphically.

More recently, Salcic and Smailagic [7] designed a custom configurable microprocessor that they called SimP. While their microprocessor was not intended for use in schools, it does have many attributes that would make it a good educational tool for a microprocessor class, including a concise 16-bit instruction set, data registers, ALUs, and a simple datapath. The differences between SimP and the MIPS processor presented in Patterson and Hennessy, however, are drastic enough that to use SimP while teaching microprocessor design from Patterson and Hennessy would be comewhat counterproductive. Another option would be to teach microprocessor design using solely SimP and it's documentation, but the literature accompanying the SimP processor is too sparce to be very helpful in the teaching of a course.

Other groups have also successfully placed processors onto FPGAS, including companies such as Xilinx [8], Altera [9], [10], and Gray Research [11]. Their work, however, has been aimed at obtaining high performance processors or efficient mapping strategies for processors on FPGAs. Our design goals are quite different, as we are trying to use FPGA-implemented CPUs as tools for teaching computer architecture to students, meaning that the focus of the class must be on CPU design and not on FPGAs.

Our design goals did not allow us to use any of these commercial FPGA-optimized CPUs. One major drawback of their implementations is that the processors are invariably tied to the FPGAs they are implemented on: to teach CPU design using any of them would require almost as much background in FPGAs as it would in processor design. We want to keep the focus away from the FPGAs; they should simply be tools for learning CPU design. As such, we wanted to create a simple CPU that would require

almost no FPGA background.

Other design requirements also led us to create our own CPUs. In the classroom, our processors will need to be easily dissected and understood. Also, since our students will be putting their processors together one part at a time, we needed a processor that is easily tested and debugged even when incomplete. Considering these requirements, we determined that the best course of action was to design the processor from scratch, allowing us to build an integrated debugging tool that would be able to control and debug the processor.

FPGA Implementation of the Processor

The most important task in our work was to implement a single-cycle processor onto an FPGA so that it would look and behave like the processor presented in *Hennessey and Patterson*. We chose to do our implementation on a XESS-XSV board [12], shown in Figure 2. The board components used in this implementation are as follows: one XILINX Virtex XCV300 FPGA [3], two independent 512K x 16 SRAM banks [13], one parallel port interface [14] that will allow communication between a PC and the FPGA, and one push-button switch. The Virtex XCV300 FPGA contains 322,970 system gates and 65,535 BRAM bits. Xilinx's Foundation software was our design platform, and provided us with the bit streams that were used to program the FPGAs.

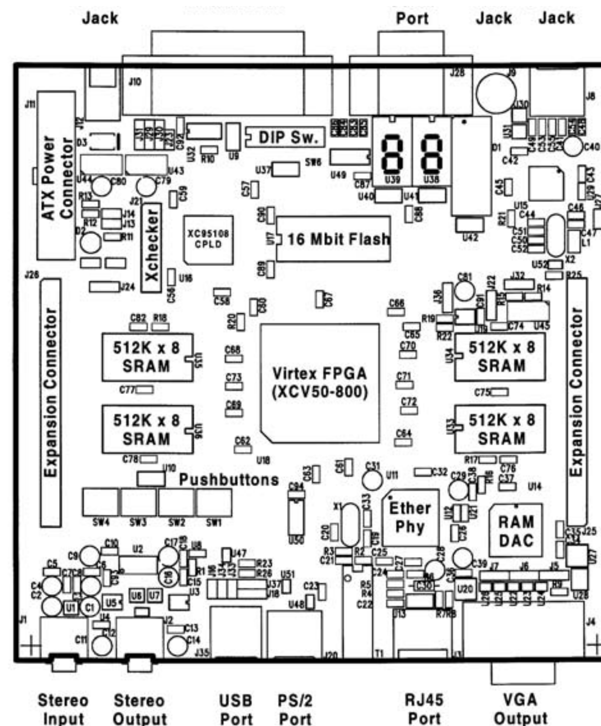


Figure 2: The XSV Board Schematic [12]

The FPGA is used for nearly all of the control and datapath of the processor. The only exceptions are the data and instruction memories, which are implemented in SRAM due to a lack of space on the FPGA. The parallel port is used for loading configurations to the FPGA as well as communicating with the PC, which is necessary for the debugging tool. The push-button is used to reset the processor.

While many of the smaller aspects of the processor are trivial to design and implement on the FPGA[1], considerable effort was needed to implement the processor's memories. This is because FPGAs are primarily designed to handle logic, not memories, and are therefore fabricated with a very small amount of memory space. Thus the register file, instruction memory, and data memory must be carefully implemented so that the chip's resources are not overused.

The register file is a 32-bit, 32-address memory that has two read ports and one write port. This means that during any clock cycle, the processor must be able to read values from two different memory addresses as well as write a value to a third address. In order to avoid taxing the standard logic resources of the FPGA, we implemented the register file using Block Ram (BRAM). A single BRAM can have data widths of no greater than 16 bits and has a maximum of two ports; multiple BRAMs were needed in order to implement the 32-bit, three-ported register file.

Figure 3 shows the implementation of the register file using four dual-ported 16-bit BRAMs. These four 16-bit BRAMs functionally combine to create two 32-bit BRAMs, which is what the processor requires. It is necessary that writes be performed on both of these 32-bit BRAMs so that a read from either of the BRAMs will return the proper value. One read port can then be assigned to each of the 32-bit BRAMs, providing us with a 32-bit, 32-address register file complete with two read ports and one write port.

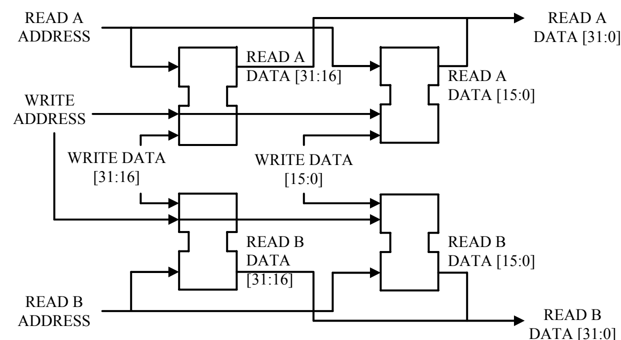


Figure 3: Register File Implementation

By itself, the FPGA does not have enough storage area to hold the instruction and data memories, so we chose to put these memories into off-chip SRAM. The instruction and data memories need to be large enough to hold a complete program in them, and the 2K addresses that are available in the FPGA's BRAM are insufficient. Combining the two SRAM banks results in 512K addresses, which is sufficiently large for the purposes of our instructional processor.

[1] The ALUs, shifters, control logic, sign extenders, and multiplexors can all be described in very few lines of verilog code or with very simple schematics.

Our goal was to present the processor exactly as it is shown in *Patterson and Hennessy*, which suggests that we should have implement the memories as two distinct units. Using one bank of off-chip SRAM for the memories therefore presented a problem: how would we make one memory appear to be two memories? Also, since we were now using resources that are not on the FPGA, how would we hide the complicated communications between the FPGA and the SRAM so that the students would not see them?

In the normal operation of the processor, it is common that the instruction memory and the data memory would both be accessed during the same clock cycle. Since both of these memories are implemented in the same SRAM and cannot be accessed simultaneously, this requires that the processor clock be divided into multiple clock cycles: one period for reading instructions, one period for reading data, and one period for storing data.

In addition, any memory reference requires that the FPGA send multiple signals to the SRAM at specific times in order to initiate a read or a write. The read and write waveforms for the SRAM, as implemented in our processor, are shown in Figure 4. These waveforms can each be produced using 4 clock cycles, with driven signals being changed on the rising edges of the clocks.

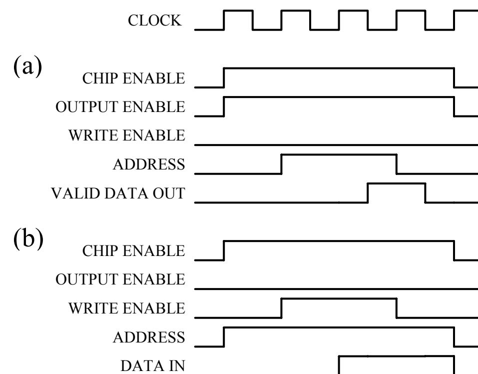


Figure 4: (a) Read, (b) Write Waveforms for the SRAM

Each communication with the SRAM requires a minimum of 4 clock cycles, which means that the memory now requires 12 clock cycles per CPU clock cycle in order to behave like two memories. To accommodate this, we use a clock of 16 times the frequency of the processor to control the instruction and data memory communications, supplying extra time where combinational delays exist. The 16-cycle sub-clock is hidden from the students so that the memory will look and behave like the idealized data and instruction memories from *Patterson and Hennessy*.

The overall timing diagram for our processor, including clocks and operations, is shown in Figure 5. In the single-cycle implementation the clock is always started prior to the current instruction fetch, is run for a

certain number of cycles, and then stopped after a memory load. In this way the clock can be run for 0 cycles, which would load an instruction and perform a memory load but would not commit any values to memory, allowing only combinational effects to occur. Another thing to notice is that the memory store and register writebacks are timed so that they are edge triggered with respect to the CPU clock.

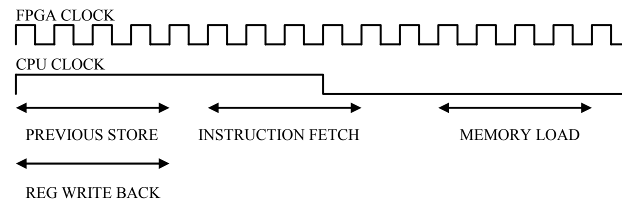


Figure 5: Processor Timing Diagram

The pipelined implementation of the processor had to use a slightly different clocking scheme than the single-cycle implementation. Using the single-cycle clocking scheme for the pipelined processor, a conflict occurs when a specific register is being written to and read from during the same clock cycle. The fact that the clock is always started before the instruction fetch means that the read would occur before the write, resulting in the wrong value being read from the register. In the pipelined processor, therefore, the write back must occur before the read occurs. This is accomplished by always starting the pipelined clock at the write back, allowing the proper value to be read from the same register during the same clock cycle.

Appendices A and B contain the verilog and schematic files used to create the single cycle processor, and Appendices C and D contain the additional verilog and schematic files that were needed for the pipelined processor.

Debugging Tool

The debugging tool is necessary for making our FPGA-implemented processor usable in the classroom. The tool allows the students to observe and control the internal states of their processors, a necessity for the designing and debugging processes. We chose to create the debugging tool using Visual Basic, and the code for the tool appears in Appendix E.

Communications between the PC and the FPGA are carried out over the parallel port. The parallel port provides a total of twelve signals that can be seen by both the PC and the FPGA. Eight of these signals (bits) are changeable only by the PC and four of them only by the FPGA. Both the PC and the FPGA can thus put bits onto the parallel port that can be seen by the other device.

Of these parallel port bits, we use one bit in each direction as an acknowledge signal. When the PC wants the FPGA to accept new data it changes the value of its acknowledge bit, which the FPGA recognizes as a prompt to read the other seven PC-controlled bits. When the FPGA has successfully read the bits, it changes the value of its own acknowledge bit, which tells the PC that the transaction was successful. This leaves seven actual data bits to be controlled by the PC, and three data bits to be controlled by the FPGA.

The interface of the debugging tool is shown in Figure 6. The tasks that it can perform can be broken into 5 basic categories: clock manipulation, reading of data lines, writing the program counter, reading or writing the register file, and reading or writing the instruction or data memory. These tasks provide the user with control of all vital parts of the processor as well as complete observability of the processor's internal state. The list of commands that the debugger can perform is shown in Table 1.

We supply two methods for controlling the clock: one which runs the clock for a specified number of cycles and one which runs the clock until the user gives a stop signal. The student can read data lines simply by selecting one of the provided lines: PC, INST, DATA (data memory output), REG1, REG2, ALU, MISC1, or MISC2. MISC1 and MISC2 are lines that the student can hook up to any part of the processor, providing added debugging flexibility. Students can write to the program counter, register file and memories by supplying the address and data value to be written, with the exception of the program counter which does not require an address. They can then perform reads of the register file and memory simply by supplying the read address.

Figure 6: The Debugging Interface

Table 1: Debugger Commands and OP Fields

DEBUGGING OPERATION	RESULT
CLOCK	
Run	Runs the processor clock for a specified number of cycles.
Start	Starts the processor clock.
Stop	Stops the processor clock.
READ DATA LINES	
PC	Returns the current value of the program counter
INST	Returns the current output of the instruction memory
DATA	Returns the current output of the data memory
REG1	Returns the current output of register 1
REG2	Returns the current output of register 2
ALU	Returns the current output of the datapath ALU
MISC1	Returns the current output of the first miscellaneous line (hooked up by the user to any output)
MISC2	Returns the current output of the second miscellaneous line (hooked up by the user to any output)
SRAM	
RUN	Runs a routine that tests the SRAM banks
Dump Memory	Dumps the contents of the SRAM to a file
Load File	Loads the specified file into the SRAM
WRITE DATA LINES	
MISC1	Set the first miscellaneous input line to a specified value (hooked up by the user to any input)
MISC2	Set the second miscellaneous input line to a specified value (hooked up by the user to any input)
MISC3	Set the third miscellaneous input line to a specified value (hooked up by the user to any input)
Update All Three	Update all three miscellaneous lines
PC: Write	Write a value to the program counter
REG: Write	Write a value to any register
REG: Read	Read a value from any register
MEM: Write	Write a value to any instruction or data address
MEM: Read	Read a value from any instruction or data address
Get Reg File	Obtain the complete state of the register file
INSTRUCTION MEMORY	Observe the instructions near the current instruction

Reduction of MIPS Instruction Set

The work described in this section was performed by James Harris and Tim Midget under the supervision of Scott Hauck, and is included in this document because it is an integral part of the project.

We determined that it would not be ideal to implement a processor that ran the complete MIPS instruction set: it would be too time consuming and too taxing on our FPGA resources, while providing negligible educational value to the students. At the same time we want our processor to be able to load and run programs, which would seem to require the ability to execute any instruction. To accommodate these goals we searched for a small set of instructions that could be used to describe the complete set, but that would allow us to retain the flavor of a real microprocessor.

At the far end of the spectrum, it is possible to create a "one instruction computer" using only one instruction to build the rest of the instruction set [15]. Another option we considered was using an 8-bit implementation rather than a 32-bit set. Both of these options, however, would oversimplify the processor to a point where it no longer behaved like a standard processor. Since the processor will be used solely for educational purposes, we wanted to make sure that it would retain enough characteristics of a standard processor that it would be useful in the classroom.

Other methods of reducing the instruction set proved themselves too inefficient. For example, we could have pre-loaded our processor with all 2^{32} possible 32-bit values: a negation could then have been accomplished simply by loading the proper value from memory. The memory overhead from a scheme like this is ridiculously large, however, which makes it impractical to use.

In the end, we chose to reduce MIPS to the eight instructions shown in Table 2. These are all standard instructions from the MIPS instruction set.

We can describe any MIPS instruction by some combination of these eight instructions. This set of instructions maintains the feel of a real processor by providing a wide range of functionality. It contains a logical operator (NOR), an arithmetic operator (SUBU), loading and storing instructions (LW, SW), branching and jumping capabilities (BGEZ, JALR), and exceptions (SYSCALL, BREAK). In practice, programs will be run through our assembler, which will reduce all instructions down to this set of eight. Instructions will then be turned into machine code, loaded onto the processor, and run.

Different approaches were needed for each type of instruction that we wanted to reduce out of the set. For example, an ADDU (add unsigned) can be accomplished by subtracting one of the values from 0 and then subtracting that value from the other. Similarly, an AND is accomplished by negating both values and then performing a NOR.

Table 2: Reduced MIPS Instruction Set

INSTRUCTION / FORMAT	OPERATION
NOR rd, rs, rt (Nor)	perform bit-wise logical NOR on the contents of register rs and rt, place the result in rd
SUBU rd, rs, rt (Subtract Unsigned)	subtract the contents of rt from rs and place in rd (unsigned)
LW rt, offset(base) (Load Word)	load into rt the value from memory location that is the sum of offset and the value in register base
SW rt, offset(base) (Store Word)	Store the value from register rt into the memory location that is the sum of offset and the value in register base
BGEZ rs, offset (Branch on Greater Than or Equal to 0)	if the contents of rs >= 0, branch to (PC + offset) (offset is shifted two bits and sign extended)
JALR rd, rs (Jump and Link Register)	jump to the address in register rs, placing the next instruction in rd
SYSCALL (System Call)	a system call exception transferring control to the exception handler
BREAK (Break)	A breakpoint trap occurs, transferring control to the exception handler

Many instructions make use of the fact that adding a number to itself is the same as shifting the number once to the left. Shift lefts and shift rights are easily accomplished using this concept, and multiplies are also aided by this fact. We perform our multiplies by shifting the multiplier left one bit at a time, each time checking the MSB. If the MSB is 1, the multiplicand is added to the product, which is also shifted each time we check the multiplier. Thus, a multiply is carried out in the same manner that a human would multiply two numbers by hand. (The multiplier is forced to be positive so that this will work. If a negative multiplier is present, we negate both the multiplier and multiplicand to achieve this. Checking the MSB of a number is done via BGEZ.)

The divide instruction is also implemented in the same way that a person would do it by hand. The dividend is shifted one bit at a time and is compared to the divisor upon each shift. At any step, if the dividend is greater than or equal to the divisor, the divisor is subtracted from the dividend and the quotient is incremented by one. Just like with the multiply, the quotient is shifted along with the dividend in order to keep the answer correct. At the end, the remaining dividend is placed into the remainder register.

Table 3 shows how many instructions must be used to replace each MIPS instruction. The total is the number of instructions used to replace the given instruction, while the average is the average number of instructions that will be executed in place of the given instruction. Note that the average can be greater than the total code expansion because of loops within the code. LUI and LI (load upper immediate and load immediate, which load 16-bit constants into a register) require only one instruction because the assembler assists them by placing the value to be loaded into memory prior to operation.

It is clear that by reducing the number of usable instructions we are consequently inflating the overall

instruction count and thus reducing our processor's performance. Remember, however, that performance is not one of our major design goals. A simpler implementation is more valuable to us than fast processing, so we believe the instruction inflation shown in Table 3 is acceptable.

Table 3: Instruction Inflation Due to Reducing Set

INST	TOT	AVG	INST	TOT	AVG	INST	TOT	AVG	INST	TOT	AVG
LB	13	12.5	J	2	2	ORI	3	3	DIVU	30	352.5
LBU	8	8	JAL	2	2	OR	2	2	SLTI	9	5.83
LH	15	9	JR	1	1	XORI	6	6	SLTIU	9	6
LHU	9	6.5	JALR	**1**	**1**	XOR	5	5	SLT	8	4.83
LW	**1**	**1**	BEQ	5	4	NOR	**1**	**1**	SLTU	9	5
LWL	22	47.25	BNE	6	3.63	ADDI	2	2	SLL	9	51.53
LWR	54	81.75	BLEZ	4	2.5	ADDIU	2	2	SRL	16	64.13
LUI	1	1	BGTZ	5	2.75	ADD	17	5.9	SRA	17	65.09
LI	1	1	BLTZ	2	1.5	ADDU	2	2	SLLV	8	50.53
SB	29	17.75	BGEZ	**1**	**1**	SUB	2	2	SRLV	15	63.13
SH	15	9.5	BLTZAL	3	2.5	SUBU	**1**	**1**	SRAV	16	64.09
SW	**1**	**1**	BGEZAL	2	2	MULT	39	212	SYSCALL	**1**	**1**
SWR	25	14	AND	3	3	MULTU	26	336.8	BREAK	**1**	**1**
SWL	26	16.25	ANDI	4	4	DIV	38	356.8			

Assembler

We have designed an assembler that is capable of mapping any of the MIPS non-floating point/coprocessor instructions to this set of eight. The assembler is written using LEX and YACC, and the code is included in Appendices F and G.

The assembler accepts an input file of assembly language code and goes through two basic steps: first, it maps the instructions to the set of eight that is supported by the processor, and secondly, it turns the assembly language code (of the eight instructions) into machine language code. The machine language file outputted by the assembler can be directly loaded into the processor on the FPGA.

Most of the MIPS instructions can be directly mapped to the supported set of eight, with the exception of immediate instructions. Since our processor does not support any immediate instructions, there can be no direct way to access the immediate value. To get around this, whenever the assembler sees an immediate instruction it writes the immediate value into a data memory section that appears after it's instruction memory section. It then adds a load instruction which loads the immediate value from data memory into a register, allowing the processor to access the immediate value as if it were a simple registered value.

Classroom Integration

We will provide the students with access to the following items: an XSV board, access to a computer with Xilinx Foundation Software, our processor debugging tool, a parallel port connector, a Foundation project containing an incomplete version of our processor, and a power supply for powering the board. With these tools the students will be able to design and implement a processor on their FPGA. A possible set of projects is suggested here in order to illustrate the use of our tools.

An early assignment for the students will likely be the designing of the register file. Each student will design their register file and add it to the Foundation project, then implement the design onto their FPGA. Once done, they can use their debugging tool to verify their register file design in multiple ways. The simplest way will be to write values directly to the register file and then read them to assure correct operation, but they could also use the debugging tool to simulate a specific instruction and see that the instruction has the proper effects on the register file.

The next design project might be the ALU. After designing the ALU and implementing it onto the FPGA along with their register file, a student would again have a number of ways to test their designs. First, they could simply use the debugging tool to stimulate the ALU's inputs and control values, observing that the proper value is outputted. They could also hook up the register file outputs to the ALU inputs, testing the two units together by giving addresses to the register file and observing the ALU output. A third option is to simulate a specific instruction and observe its propagation through the register file and ALU. Again, the debugging tool will provide many testing options to the students, allowing them to debug in almost any manner they wish.

When a student has designed a complete single-cycle processor, the debugging tool will then be used not only to stimulate inputs and read outputs, but also to control the CPU clock. A student will be able to read or write any of the processor's memories in order to obtain the state they want, and then they will be able to walk the CPU clock in any increment they wish. For example, a student might load a specific instruction into memory and then run the processor for 0 cycles, thereby loading the instruction and observing the propagation of values through the combinational paths of the CPU. The student could also run the processor for a full cycle, committing to memory whatever the instruction specifies, thereby making certain that the memories are working correctly. Again, a number of debugging options are made available, giving the students the freedom to debug however they wish.

Beyond a single-cycle implementation, the students might also be asked to design a multi-cycle or pipelined processor. In reality, these and almost any projects relating to processor design can be completed because the debugging tool supplies complete control and observability of the processor's internal state.

Results and Discussion

The results of our FPGA implementations of two processors appear in Table 4. For the single-cycle implementation we were able to fit our design quite easily into our XILINX Virtex XCV300 PFGA, using only 22% of the LUTs and 25% of the BRAM units. Our FPGA runs at a maximum frequency of 25MHz, which means that the actual processor runs at 1.5MHz (25/16). The pipelined processor produced similar utilization numbers, using 29% of the LUTs, 25% of the BRAMs, and running at 23 MHz (1.4MHz for the processor).

Table 4: Implementation Results

PROCESSOR TYPE	FPGA CLOCK FREQ	PROC. CLOCK FREQ	% LUTS USED	% BRAM USED
SINGLE-CYCLE MIPS	25 MHz	1.5 MHz	22%	25%
PIPELINED MIPS	23 MHz	1.4 MHz	29%	25%

The debugging tool has also been tested, and successfully supplies a user with complete control of the processor even when the processor is incomplete.

The work presented in this paper describes a completely functional and debuggable processor. While we have offered an outline of some possible class projects, instructors who wish to use our implementation will ultimately make the decision on what aspects of architecture to teach. By providing the students with a Foundation file that already has most parts of the processor, instructors will be able to have their students design and replace the missing parts. Upon completion, students will have a working processor that they can actually operate.

Conclusion

The FPGA-implemented processor, debugging tool, and reduced instruction set presented in this paper represent the primary step in our goal of introducing FPGAs into the teaching of computer architecture. The processor and debugging tool should be easy to integrate into the classroom, where they will provide students with a hands-on experience that was previously unavailable to them. We are convinced that when teachers incorporate these tools into their classrooms that students will display better retention of the intended lessons as well as increased enthusiasm about the work performed.

Current Status and Future Work

In the fall of 2001, an undergraduate student named Steve Detmer tested the basic lesson plan that we are developing for this project. Steve took a microprocessor design course (EE471) in the spring of 2001, and in the fall we had him implement his class designs onto the FPGA in order to test the debugging tool and FPGA-implemented processor. He was successful in using his designs to create a functional FPGA with the aid of our debugging tool, and the suggestions and criticisms that came out of Steve's work have since been integrated into the project and into this paper.

In addition, the assembler that we developed has been provided to the students in the spring 2002 offering of EE471. Use of the compiler will allow the students to create and run programs in software, adding to their understanding of the material. Feedback from the students will also enable us to fix any bugs or user-unfriendly aspects that are present in the compiler.

The next step is to get this complete project integrated into the teaching of Computer Organization and Design. We will be looking into the hardware and software requirements of incorporating our research into EE471 at the University of Washington, with the goal of integrating our work into the classroom in the fall of 2002.

In the future, there also exists the possibility of merging this work with student-built compilers and operating systems. By combining the teachings of processor design, compilers, and operating systems, students will eventually be able to turn chips into complete working computers. By doing this, the interactions and tradeoffs between these various fields can be explored in real terms, with concrete results.

Notes

Along with the previously mentioned eight instructions, our processor also supports the instructions that are presented in *Patterson and Hennessy*: BEQ, ADD, SUB, AND, OR, SLT, and J. This set includes simple and easily understood instructions, and supporting them will provide the students a larger range of instructions with which to test and debug their processor. Thus in total our processor supports 15 instructions.

End Notes

[1] Courter, S.S., S. B. Millar, and L. Lyons, "From the Students' Point of View: Experiences in a Freshman Engineering Design Course", *Journal of Engineering Education,* vol. 87, no. 3, 1998.

[2] Mahendran, M, "Project-Based Civil Engineering Courses", *Journal of Engineering Education,* vol. 84, no. 1, 1995.

[3] Xilinx Inc., *The Programmable Logic Data Book*, pp. 3.3-3.12, 1999.

[4] Xilinx Inc., *Dual Port Block Memory: Product Specification*, May 28 1999.

[5] Patterson, D. A., J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, San Francisco, 1998.

[6] Diab, H, and I. Demashkieh, "A reconfigurable microprocessor teaching tool", *IEEE Proceedings,* vol. 137, Pt. A, No. 5, September 1990.

[7] Salcic, Zoran and Asim Smailagic, *Digital Systems Design and Prototyping Using Field Programmable Logic,* pp. 203-242, Norwell, 1997.

[8] Alliance Core, *ARC 32-Bit Configurable RISC Processor: Datasheet*, July 3 2000.

[9] Altera Corporation, *ARM-Based Embedded Processor Device Overview: Datasheet*, ver. 1.2, February 2001.

[10] Altera Corporation, *MIPS-Based Embedded Processor Device Overview: Datasheet*, ver. 1.2, February 2001.

[11] Gray, J., "Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip", 2000.

[12] XESS Corporation, *XSV Board V1.0 Manual*, March 1 2000.

[13] Alliance Semiconductor, *5V/3.3V 512K x 8 CMOS SRAM: Datasheet*, ver. 1.0, January 12 2001.

[14] XESS Corporation, *XSV Parallel Port Interface: Application Note* by D. Vanden Bout, ver. 1.0, April 10 2000.

[15] Styer, Eugene, "One Instruction Computers", http://eagle.eku.edu/faculty/styer/oisc.html, 1996.

Bibliography

Alliance Core, *ARC 32-Bit Configurable RISC Processor: Datasheet*, July 3 2000.

Alliance Semiconductor, *5V/3.3V 512K x 8 CMOS SRAM: Datasheet*, ver. 1.0, January 12 2001.

Altera Corporation, *ARM-Based Embedded Processor Device Overview: Datasheet*, ver. 1.2, February 2001.

Altera Corporation, *MIPS-Based Embedded Processor Device Overview: Datasheet*, ver. 1.2, February 2001.

Courter, S.S., S. B. Millar, and L. Lyons, "From the Students' Point of View: Experiences in a Freshman Engineering Design Course", *Journal of Engineering Education,* vol. 87, no. 3, 1998.

Diab, H, and I. Demashkieh, "A reconfigurable microprocessor teaching tool", *IEEE Proceedings,* vol. 137, Pt. A, No. 5, September 1990.

Gray, J., "Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip", 2000.

Mahendran, M, "Project-Based Civil Engineering Courses", *Journal of Engineering Education,* vol. 84, no. 1, 1995.

Patterson, D. A., J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, San Francisco, 1998.

Salcic, Zoran and Asim Smailagic, *Digital Systems Design and Prototyping Using Field Programmable Logic,* pp. 203-242, Norwell, 1997.

Styer, Eugene, "One Instruction Computers", http://eagle.eku.edu/faculty/styer/oisc.html, 1996.

XESS Corporation, *XSV Board V1.0 Manual*, March 1 2000.

XESS Corporation, *XSV Parallel Port Interface: Application Note* by D. Vanden Bout, ver. 1.0, April 10 2000.

Xilinx Inc., *The Programmable Logic Data Book*, pp. 3.3-3.12, 1999.

Xilinx Inc., *Dual Port Block Memory: Product Specification*, May 28 1999.

Appendix A – Verilog Code for the Single Cycle Processor

```
//File: add1.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//This is a simple ADD 1 Unit, where out = in + 1.

module add1(in, out);

input    [31:0]   in;
output   [31:0]   out;

assign out = in + 1;

endmodule
```

```
//File: adder.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//This 32 bit adder performs out = in1 + in2.

module adder(in1, in2, out);

input    [31:0]   in1;
input    [31:0]   in2;
output   [31:0]   out;

assign out = in1 + in2;

endmodule
```

```
//File: alu2.v
//author: Mark Holland
//Last Modified: May 29, 2002
//a 32-bit ALU composed of 8 4-bit ALUs.

//Performs AND, OR, ADD, SUB, and SLT on two 32-bit numbers.

module alu32(control, a, b, result, overflow, zero);

input    [2:0]    control;
input    [31:0]   a;
input    [31:0]   b;

output   [31:0]   result;
output   overflow;
output   zero;

wire zero;
wire overflow;
wire final_carry;

wire set;

wire [6:0] carry;

assign zero = ~| result[31:0];

assign overflow =         (!control[2] && control[1] && !control[0] && !a[31] && !b[31] && result[31])
+
                          (!control[2] && control[1] && !control[0] && a[31] && b[31] && !result[31]) +
                          (control[2] && control[1] && !control[0] && !a[31] && b[31] && result[31]) +
                          (control[2] && control[1] && !control[0] && a[31] && !b[31] && !result[31]) +
                          (!control[2] && control[1] && control[0] && result[31]);


alu4 adder4_0(control, control[2], a[3:0], b[3:0], {3'b000, set}, result[3:0], carry[0]);
alu4 adder4_1(control, carry[0], a[7:4], b[7:4], 4'b0000, result[7:4], carry[1]);
alu4 adder4_2(control, carry[1], a[11:8], b[11:8], 4'b0000, result[11:8], carry[2]);
alu4 adder4_3(control, carry[2], a[15:12], b[15:12], 4'b0000, result[15:12], carry[3]);
alu4 adder4_4(control, carry[3], a[19:16], b[19:16], 4'b0000, result[19:16], carry[4]);
alu4 adder4_5(control, carry[4], a[23:20], b[23:20], 4'b0000, result[23:20], carry[5]);
alu4 adder4_6(control, carry[5], a[27:24], b[27:24], 4'b0000, result[27:24], carry[6]);
alu4_top adder4_7(control, carry[6], a[31:28], b[31:28], 4'b0000, result[31:28], final_carry, set);

endmodule

//author: Mark Holland
//the upper most 4-bit ALU, which contains the set signal needed for SLT

module alu4_top(control, c_in, a, b, less, result, c_out, set);

input    [2:0]    control;
input    c_in;
input    [3:0]    a, b;
input    [3:0]    less;

output   [3:0]    result;
output   c_out;
output   set;

wire     [2:0]    carry;

alu1 adder_0(control, c_in, a[0], b[0], less[0], result[0], carry[0]);
alu1 adder_1(control, carry[0], a[1], b[1], less[1], result[1], carry[1]);
alu1 adder_2(control, carry[1], a[2], b[2], less[2], result[2], carry[2]);
alu1_top adder_3(control, carry[2], a[3], b[3], less[3], result[3], c_out, set);

endmodule

//author: Mark Holland
//a 4-bit ALU composed of 4 1-bit ALUs

module alu4(control, c_in, a, b, less, result, c_out);

input    [2:0]    control;
```

```
input    c_in;
input    [3:0]    a, b;
input    [3:0]    less;

output   [3:0]    result;
output   c_out;

wire     [2:0]    carry;

alu1 adder_0(control, c_in, a[0], b[0], less[0], result[0], carry[0]);
alu1 adder_1(control, carry[0], a[1], b[1], less[1], result[1], carry[1]);
alu1 adder_2(control, carry[1], a[2], b[2], less[2], result[2], carry[2]);
alu1 adder_3(control, carry[2], a[3], b[3], less[3], result[3], c_out);

endmodule

//author: Mark Holland
//the upper most 1-bit adder

//This differs from the normal 1-bit adder in that it will produce a set
//signal that will be 1 if a < b.

module alu1_top(control, c_in, a, b, less, result, c_out, set);

input    [2:0] control;
input    c_in;
input    a, b;
input    less;

output   result;
output   c_out;
output set;

reg result;
reg c_out;
reg set;


always @ (control or a or b or c_in or less)

        case(control)

        0:      result = a && b;
        1:      result = a || b;
        2:      begin
                result = a ^ b ^ c_in;
                c_out = (a && b) + (a && c_in) + (b && c_in);
                end
//I'll check this result for overflow for BGEZ
        3:      result = !a;
        4:      result = !a || !b;
        5:      result = !a && !b;
        6:      begin
                result = a ^ !b ^ c_in;
                c_out = (a && !b) + (a && c_in) + (!b && c_in);
                end
        7:      begin
                result = less;
                c_out = (a && !b) + (a && c_in) + (!b && c_in);

                case({a, b})

                0:      set = a ^ !b ^ c_in;
                1:      set = 0;
                2:      set = 1;
                3:      set = a ^ !b ^ c_in;

                endcase

                end

        endcase

endmodule
```

```
//author: Mark Holland
//a 1 bit ALU which performs AND, OR, ADD, SUB, NAND, NOR, and SLT

module alu1(control, c_in, a, b, less, result, c_out);

input    [2:0] control;
input    c_in;
input    a, b;
input    less;

output   result;
output   c_out;

reg result;
reg c_out;

always @ (control or a or b or c_in or less)

//control:       0 = and
//               1 = or
//               2 = add
//               3 = check sign
//               4 = nand
//               5 = nor
//               6 = subtract
//               7 = set on less than
//
//               && = and bits
//               || = or bits
//               ^  = xor bits
//               !  = negate bit
//               ~& = nand bits
//               ~| = nor bits

         case(control)

         0:      result = a && b;
         1:      result = a || b;
         2:      begin
                 result = a ^ b ^ c_in;
                 c_out = (a && b) + (a && c_in) + (b && c_in);
                 end
         4:      result = !a || !b;
         5:      result = !a && !b;
         6:      begin
                 result = a ^ !b ^ c_in;
                 c_out = (a && !b) + (a && c_in) + (!b && c_in);
                 end
         7:      begin
                 result = less;
                 c_out = (a && !b) + (a && c_in) + (!b && c_in);
                 end

         endcase

endmodule
```

```
//File: io_control_box.v
//Author: Mark Holland
//Module: io_control_box
//Last Modified: May 29, 2002
//Operation: This control box serves as the interface between the PC and
//the FPGA for all communications.  It also produces the CPU clock and
//the memory clocks that are used during processor operations, as well
//as special control signals needed by the processor for operation.


module io_control_box(inst_address, data_address, write_data, data_in, clock, reset, pc_ack,
parallel_in, reg1_in, reg2_in, misc1_in, misc2_in, MR, MW, RW, ERR, misc1_out, misc2_out, misc3_out,
instruction, read_data, address_out, data_out, v_ack, ce_bar, we_bar, oe_bar, reg_file_clock,
parallel_out, pc_write_value, pc_write, pc_clk, reg_write_value, reg_write_address, reg_read_address,
reg_we, clock_running, jalr_pc_store_value, error_reg_choose);

input    [31:0]   inst_address;
input    [31:0]   data_address;
input    [31:0]   write_data;
input    [31:0]   data_in;

input             clock;
input             reset;
input             pc_ack;
input    [6:0]    parallel_in;
//input  [31:0]   pc_in;
//input  [31:0]   inst_in;
//input  [31:0]   dataline_in;
input    [31:0]   reg1_in;
input    [31:0]   reg2_in;
//input  [31:0]   alu_in;
input    [31:0]   misc1_in;
input    [31:0]   misc2_in;
input             MR;
input             MW;
input             RW;
input             ERR;

//added 10-12 for debugging
output   [31:0]   misc1_out;
output   [31:0]   misc2_out;
output   [31:0]   misc3_out;

output   [31:0]   instruction;
output   [31:0]   read_data;

output   [18:0]   address_out;
output   [31:0]   data_out;

//output          cpu_clk;
//output          mem_clk;
output            v_ack;
output            ce_bar;
output            we_bar;
output            oe_bar;
output            reg_file_clock;

output   [2:0]    parallel_out;

output   [31:0]   pc_write_value;
output            pc_write;
output            pc_clk;

output   [31:0]   reg_write_value;
output   [4:0]    reg_write_address;
output   [4:0]    reg_read_address;
output            reg_we;

output            clock_running;


//added for jalr
output   [31:0]   jalr_pc_store_value;

//added for exception
```

```
output           error_reg_choose;


//added
wire    cpu_clk;
wire    mem_clk;

//added for debugging 10-12
reg     [31:0]  misc1_out;
reg     [31:0]  misc2_out;
reg     [31:0]  misc3_out;


//added for jalr
reg     [31:0]  jalr_pc_store_value;

//added for exception
reg             error_reg_choose;


reg     [10:0]  cpu_clock;
reg     [3:0]   state;
reg     [69:0]  shift_in;
reg     [63:0]  op_in;
reg     [31:0]  shift_out;
reg     [3:0]   ack_num;
reg             stop_clock;
reg     [18:0]  address_out;

reg             v_ack;
reg             ce_bar;
reg             we_bar;
reg             oe_bar;
reg             reg_file_clock;
reg     [31:0]  pc_write_value;
reg             pc_write;
reg             pc_clk;
reg     [31:0]  reg_write_value;
reg     [4:0]   reg_write_address;
reg     [4:0]   reg_read_address;
reg     [1:0]   reg_write;
reg     [31:0]  data_out;
reg     [2:0]   mem_write;
reg     [1:0]   reg_read;
reg     [2:0]   mem_read;
reg             reg_we;

reg     [31:0]  instruction;
reg     [31:0]  read_data;

reg     [1:0]   pc_num;


assign  parallel_out = shift_out[2:0];
assign  cpu_clk = cpu_clock[3];
assign  mem_clk = cpu_clock[0];

assign  clock_running = |cpu_clock;

parameter       TRUE = 1'b1,
                FALSE = 1'b0;

parameter       CLOCK_RESET = 11'b100_0000_0000,
                CLOCK_SET = 11'b111_1111_1111,
                CLOCK_SET2 = 11'b111_1111_1100;

parameter       ACK_IN_DONE = 4'b1010,
                ACK_OUT_LAST = 4'b1010;

parameter       ZERO4_ = 4'b0000,
                ZERO5_ = 5'b0_0000,
                ZERO11_ = 11'b000_0000_0000,
                ZERO19_ = 19'b000_0000_0000_0000_0000,
                ZERO32_ = 32'b0000_0000_0000_0000_0000_0000_0000_0000,
```

```
                 ZERO64_ =
64'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000,
                 ZERO70_ =
70'b00_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;

parameter        NO_OP = 32'b0000_0000_0000_0000_0000_0000_0010_0000;


parameter        WAITING =        4'b0000,
                 GET_OP =         4'b0001,
                 RUN_CLOCK =      4'b0010,
                 RUN_CLOCK2 =     4'b0011,
                 START_CLOCK =    4'b0100,
                 STANDARD_READ = 4'b0101,
                 WRITE_PC =       4'b0110,
                 WRITE_REG =      4'b0111,
                 WRITE_MEM =      4'b1000,
                 READ_FROM_REG = 4'b1001,
                 READ_FROM_MEM = 4'b1010;


parameter        ZERO =           4'b0000,
                 ONE =            4'b0001,
                 TWO =            4'b0010,
                 THREE =  4'b0011,
                 FOUR =           4'b0100,
                 FIVE =           4'b0101,
                 SIX =            4'b0110,
                 SEVEN =  4'b0111,
                 EIGHT =  4'b1000,
                 NINE =           4'b1001,
                 TEN =            4'b1010,
                 ELEVEN =         4'b1011,
                 TWELVE =         4'b1100,
                 THIRTEEN =       4'b1101,
                 FOURTEEN =       4'b1110,
                 FIFTEEN =        4'b1111;

parameter        ZERO6 =          6'b000000,
                 ONE6 =           6'b000001,
                 TWO6 =           6'b000010,
                 THREE6 =         6'b000011,
                 FOUR6 =  6'b000100,
                 FIVE6 =  6'b000101,
                 SIX6 =           6'b000110,
                 SEVEN6 =         6'b000111,
                 EIGHT6 =         6'b001000,
                 NINE6 =  6'b001001,
                 TEN6 =           6'b001010,
                 ELEVEN6 =        6'b001011,
                 TWELVE6 =        6'b001100,
                 THIRTEEN6 =      6'b001101,
                 FOURTEEN6 =      6'b001110,
                 FIFTEEN6 =       6'b001111,
                 SIXTEEN6 =       6'b010000,
                 SEVENTEEN6 =     6'b010001;

parameter        MEM_ZERO =       3'b000,
                 MEM_ONE =        3'b001,
                 MEM_TWO =        3'b010,
                 MEM_THREE =      3'b011,
                 MEM_FOUR =       3'b100;

parameter        REG_ZERO =       2'b00,
                 REG_ONE =        2'b01,
                 REG_TWO =        2'b10;

parameter        PC_ZERO =        2'b00,
                 PC_ONE = 2'b01,
                 PC_TWO =         2'b10;

always @ (posedge clock or negedge reset)

//On reset we initialize all values.
```

```
if (!reset) begin

        state = WAITING;
        shift_in = ZERO70_;
        ack_num = ZERO;
        v_ack = 0;
        op_in = ZERO64_;
        state = WAITING;
        cpu_clock = ZERO11_;
        we_bar = 1;
        oe_bar = 1;
        ce_bar = 1;
        reg_file_clock = 0;
        reg_write = REG_ZERO;
        stop_clock = 0;
        shift_out = ZERO32_;
        pc_write_value = ZERO32_;
        pc_write = 0;
        pc_clk = 0;
        reg_write_value = ZERO32_;
        reg_write_address = ZERO5_;
        reg_read_address = ZERO5_;
        data_out = ZERO32_;
        address_out = ZERO19_;
        mem_write = MEM_ZERO;
        reg_read = REG_ZERO;
        mem_read = MEM_ZERO;
        reg_we = 0;
        instruction = ZERO32_;
        read_data = ZERO32_;
        error_reg_choose = 0;
        pc_num = PC_ZERO;
        misc1_out = ZERO32_;
        misc2_out = ZERO32_;
        misc3_out = ZERO32_;

end
else begin

        case (state)

//The WAITING state of our FSM.  The FPGA is waiting for the PC debugging
//tool to give it an instruction.  Instructions get shifted in 7 bits at a
//time (70 bits -> 64 bits total).  When an instruction has been shifted in
//all the way we go to GET_OP and decode the instruction.

        WAITING: begin

                if (stop_clock) begin

                        stop_clock = 0;

                end

                if (pc_ack) begin

                        if (ack_num != ACK_IN_DONE) begin

                                shift_in = {parallel_in, shift_in[69:7]};
                                ack_num = ack_num + 1;
                                v_ack = !v_ack;

                        end
                        else begin

                                op_in = shift_in[63:0];
                                state = GET_OP;
                                ack_num = ZERO4_;
                                v_ack = !v_ack;

                        end

                end
```

```
        end

//State GET_OP.  Here we decode the instruction we got from the debugging tool
//and go to the proper next state.

//      OP   |   COMMAND
// __________|_____________________________________________________________
//      0    |   run clock for specified number of cycles
//      1    |   run clock until told to stop
//      2    |   read program counter
//      3    |   read instruction value
//      4    |   read data value
//      5    |   read register 1 value
//      6    |   read register 2 value
//      7    |   read ALU output value
//      8    |   write the program counter with the specified value
//      9    |   write the specified register with the specified value
//      10   |   write the specified memory address with the specified value
//      11   |   read the value from the specified register
//      12   |   read the value from the specified memory address
//      13   |  read misc1 value
//      14   |  read misc2 value
//      15   |  drive misc1_out value
//      16   |   drive misc2_out value
//      17   |  drive misc3_out value

//      0    |  11 sends for OP, 1 send to choose OP, 11 sends to receive return value

        GET_OP: begin

                if (pc_ack) begin

                        case (op_in[63:58])

//OP = 0: We set the clock to the number of cycles to run and go to state
//RUN_CLOCK.

                        ZERO6: begin

                                cpu_clock = {1'b0, op_in[5:0], 4'b1100};
//                              ce_bar = 0;
                                state = RUN_CLOCK;

                        end

//OP = 1: We set the clock to running and go to START_CLOCK.

                        ONE6: begin

                                cpu_clock = CLOCK_SET2;
//                              ce_bar = 0;
                                v_ack = !v_ack;
                                state = START_CLOCK;

                        end

//OP = 2: We put the program counter value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                        TWO6: begin

                                shift_out = inst_address;
                                ack_num = ack_num + 1;
                                v_ack = !v_ack;
                                state = STANDARD_READ;

                        end

//OP = 3: We put the instruction value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                        THREE6: begin

                                shift_out = instruction;
                                ack_num = ack_num + 1;
```

```
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 4: We put the data value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                FOUR6: begin

                                        shift_out = read_data;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 5: We put the first register's value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                FIVE6: begin

                                        shift_out = reg1_in;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 6: We put the second register's value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                SIX6: begin

                                        shift_out = reg2_in;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 7: We put the ALU output's value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                SEVEN6: begin

                                        shift_out = data_address;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 8: We grab the value that we wish to write to the program counter
//and go to state WRITE_PC.

                                EIGHT6: begin

//changed inst. to make sure that new value propogates through
                                        instruction = NO_OP;
                                        pc_num = PC_ZERO;
                                        pc_write_value = {13'b0_0000_0000_0000, shift_in[50:32]};
                                        state = WRITE_PC;

                                end

//OP = 9: We grab the address of the destination register and the write
//value and then go to state WRITE_REG.

                                NINE6: begin

                                        reg_write_value = shift_in[31:0];
                                        reg_write_address = shift_in[36:32];
                                        reg_write = REG_ZERO;
                                        reg_we = 1;
```

```
                                        state = WRITE_REG;

                                end

//OP = 10: We grab the address of memory and the write value and then
//go to state WRITE_MEM.

                                TEN6: begin

                                        data_out = shift_in[31:0];
                                        address_out = shift_in[50:32];
                                        mem_write = MEM_ZERO;
                                        state = WRITE_MEM;

                                end

//OP = 11: We grab the register address we wish to read and go to
//state READ_FROM_REG.

                                ELEVEN6: begin

                                        reg_read_address = shift_in[36:32];
                                        reg_read = REG_ZERO;
                                        state = READ_FROM_REG;

                                end

//OP = 12: We grab the memory address we wish to read from and go
//to state READ_FROM_MEM.

                                TWELVE6: begin

                                        address_out = shift_in[50:32];
                                        mem_read = MEM_ZERO;
                                        state = READ_FROM_MEM;

                                end

//OP = 13: We put the misc1 value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                THIRTEEN6: begin

                                        shift_out = misc1_in;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 14: We put the misc2 value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                FOURTEEN6: begin

                                        shift_out = misc2_in;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 15: We grab the misc1 value and output it to the CPU.  This
//line is used solely for debugging.

                                FIFTEEN6: begin

                                        misc1_out = shift_in[31:0];
                                        shift_out = shift_in[31:0];
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end
```

```
//OP = 16: We grab the misc2 value and output it to the CPU.  This
//line is used solely for debugging.

                        SIXTEEN6: begin

                                misc2_out = shift_in[31:0];
                                shift_out = shift_in[31:0];
                                ack_num = ack_num + 1;
                                v_ack = !v_ack;
                                state = STANDARD_READ;

                        end

//OP = 17: We grab the misc3 value and output it to the CPU.  This
//line is used solely for debugging.

                        SEVENTEEN6: begin

                                misc3_out = shift_in[31:0];
                                shift_out = shift_in[31:0];
                                ack_num = ack_num + 1;
                                v_ack = !v_ack;
                                state = STANDARD_READ;

                        end

                        default: begin

                        end

                        endcase

                end

        end

//State RUN_CLOCK.  Here we run the clock for the specified number
//of cycles.

//cpu_clock[10:4] is the number of cycles to run
//cpu_clock[3] is the main CPU clock
//cpu_clock[0] is the memory clock which is used for controlling
//      memory references

//The cpu_clock counts down until it equals zero, at which point
//the proper number of cycles has occured.  We then return to
//state WAITING.

//The case statement controls the memory references, making the
//instruction fetch, register manipulations, and data store/loads
//occur.  The standard CPU control lines control which of these
//occur.


        RUN_CLOCK: begin

                if (cpu_clock != ZERO11_) begin

                        case (cpu_clock[3:0])

                        ZERO: begin

                                ce_bar = 0;


                        end

                        FIFTEEN: begin

                                error_reg_choose = 0;
                                reg_we = RW || ERR;
                                we_bar = !MW;
                                pc_clk = 1;

                        end
```

```
FOURTEEN: begin

        reg_file_clock = RW || ERR;
        pc_clk = 0;

end

THIRTEEN: begin

        error_reg_choose = 1;
        reg_file_clock = 0;
        reg_we = ERR;
        we_bar = 1;

end

TWELVE: begin

        reg_file_clock = ERR;
        ce_bar = 1;
        address_out = inst_address[18:0];

end

ELEVEN: begin

        reg_file_clock = 0;
        ce_bar = 0;
        oe_bar = 0;
        reg_we = 0;

end

TEN: begin

end

NINE: begin

end

EIGHT: begin

        instruction = data_in;

end

SEVEN: begin

        ce_bar = 1;
        oe_bar = 1;
        reg_file_clock = 1;

end

SIX: begin

        reg_file_clock = 0;
        address_out = data_address[18:0];

end

FIVE: begin

        oe_bar = !MR;
        ce_bar = !MR;
end

FOUR: begin

end

THREE: begin
```

```
                        end

                        TWO: begin

                                if(MR) begin
                                        read_data = data_in;
                                end

                        end

                        ONE: begin

                                oe_bar = 1;
                                ce_bar = 1;
                                data_out = write_data;

                                jalr_pc_store_value = inst_address + 1;

                        end

                        default: begin

                        end

                        endcase

                        cpu_clock = cpu_clock - 1;

                end
                else begin

                        ce_bar = 1;
                        shift_out = inst_address;
                        ack_num = ack_num + 1;
                        v_ack = !v_ack;
                        state = STANDARD_READ;

                end

        end

//state START_CLOCK.  This operates the same as RUN_CLOCK, with one exception.
//In this state we run the clock until we receive a signal from the PC saying
//to stop.  This clock thus runs indefinitely.

//When the stop signal is given, the current cycle is allowed to finish and then
//we go to state WAITING.

        START_CLOCK: begin

                if (pc_ack) begin

                        stop_clock = 1;

                end

                case (cpu_clock[3:0])

                        ZERO: begin

                                if (stop_clock) begin

                                        ce_bar = 1;
//                                      cpu_clock = ZERO11_;
                                        shift_out = inst_address;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

                                else begin

                                ce_bar = 0;
```

```
        end

end

FIFTEEN: begin

        error_reg_choose = 0;
        reg_we = RW || ERR;
        we_bar = !MW;
        pc_clk = 1;

end

FOURTEEN: begin

        reg_file_clock = RW || ERR;
        pc_clk = 0;

end

THIRTEEN: begin

        error_reg_choose = 1;
        reg_file_clock = 0;
        reg_we = ERR;
        we_bar = 1;

end

TWELVE: begin

        reg_file_clock = ERR;
        ce_bar = 1;
        address_out = inst_address[18:0];

end

ELEVEN: begin

        reg_file_clock = 0;
        ce_bar = 0;
        oe_bar = 0;
        reg_we = 0;

end

TEN: begin

end

NINE: begin

end

EIGHT: begin

        instruction = data_in;

end

SEVEN: begin

        ce_bar = 1;
        oe_bar = 1;
        reg_file_clock = 1;

end

SIX: begin

        reg_file_clock = 0;
        address_out = data_address[18:0];

end
```

```
                        FIVE: begin

                                oe_bar = !MR;
                                ce_bar = !MR;
                        end

                        FOUR: begin

                        end

                        THREE: begin

                        end

                        TWO: begin

                                if(MR) begin
                                        read_data = data_in;
                                end

                        end

                        ONE: begin

                                oe_bar = 1;
                                ce_bar = 1;
                                data_out = write_data;

                                jalr_pc_store_value = inst_address + 1;

                        end

                        default: begin

                        end

                endcase


                case ({stop_clock, cpu_clock[3:0]})

                        5'b10000: begin

                                cpu_clock = ZERO11_;

                        end

                        default: begin

                                        if (cpu_clock == CLOCK_RESET) begin

                                                cpu_clock = CLOCK_SET;

                                        end

                                        else begin

                                                cpu_clock = cpu_clock - 1;

                                        end

                        end

                endcase

        end

//state STANDARD_READ.  Whenever we want to return a value to the PC, we
//do it in this state.  We send bits to the PC 3 at a time, offering 3 new
//bits upon requests from the PC.  This returns one 32-bit value to the PC
//debugging tool.

//When completed, we return to state WAITING.

        STANDARD_READ: begin
```

```
			if (pc_ack) begin

				if (ack_num != ACK_OUT_LAST) begin

					shift_out = {3'b0, shift_out[31:3]};
					ack_num = ack_num + 1;
					v_ack = !v_ack;

				end

				else begin

					shift_out = {3'b0, shift_out[31:3]};
					ack_num = ZERO4_;
					v_ack = !v_ack;
					state = WAITING;

				end
			end

		end

	//state WRITE_PC.  Here we write the specified value to the program counter.
	//This is done by first setting the write value (done in GET_OP), and then
	//pulling high the write enable line.

	//After writing the value, we go to STANDARD_READ and return the PC value
	//to the debugging tool to ensure that it was written correctly.  From there
	//we go to state WAITING.

		WRITE_PC: begin

			case (pc_num)

			PC_ZERO: begin

				pc_write = 1;
				pc_num = PC_ONE;

			end

			PC_ONE: begin

				pc_write = 0;
				pc_num = PC_TWO;

			end

			PC_TWO: begin

				pc_write_value = ZERO32_;
				shift_out = inst_address;
				ack_num = ack_num + 1;
				v_ack = !v_ack;
				state = STANDARD_READ;
				pc_num = PC_ZERO;

			end
			endcase

		end

	//state WRITE_REG.  Here we write the specified register with the
	//specified value.  We then go to STANDARD_READ to return the value
	//to the PC, and then go to state WAITING.

		WRITE_REG: begin

			case (reg_write)

			REG_ZERO: begin

				reg_file_clock = 1;
				reg_write = REG_ONE;
```

```
                end

                REG_ONE: begin

                        reg_file_clock = 0;
                        reg_write = REG_TWO;

                end

                REG_TWO: begin

                        reg_we = 0;
                        reg_write_value = ZERO32_;
                        reg_write_address = ZERO5_;
                        reg_write = REG_ZERO;
                        reg_read_address = shift_in[36:32];
                        reg_read = REG_ZERO;
                        state = READ_FROM_REG;

                end

                default: begin

                end

                endcase

        end

//state WRITE_MEM.  Here we write the specified memory address with the
//specified value.  We then go to STANDARD_READ to return the value
//to the PC, and then go to state WAITING.

        WRITE_MEM: begin

                case(mem_write)

                MEM_ZERO: begin

                        ce_bar = 0;
                        mem_write = MEM_ONE;

                end

                MEM_ONE: begin

                        we_bar = 0;
                        mem_write = MEM_TWO;

                end

                MEM_TWO: begin

                        mem_write = MEM_THREE;

                end

                MEM_THREE: begin

                        we_bar = 1;
                        mem_write = MEM_FOUR;

                end

                MEM_FOUR: begin

                        ce_bar = 1;
//                      mem_write = ZERO4_;
//                      data_out = ZERO32_;
                        mem_read = MEM_ZERO;
                        state = READ_FROM_MEM;

                end
```

```
                default: begin

                end

                endcase

        end

//state READ_FROM_REG.  Here we obtain the value from the specified
//register.  We then go to state STANDARD_RETURN and return the value
//to the debugging tool.  Finally, we go back to state WAITING.

        READ_FROM_REG: begin

                case (reg_read)

                REG_ZERO: begin

                        reg_file_clock = 1;
                        reg_read = REG_ONE;

                end

                REG_ONE: begin

                        reg_file_clock = 0;
                        reg_read = REG_TWO;

                end

                REG_TWO: begin

                        shift_out = reg1_in;
                        reg_read_address = ZERO5_;
                        ack_num = ack_num + 1;
                        v_ack = !v_ack;
                        state = STANDARD_READ;

                end

                default: begin

                end

                endcase

        end

//state READ_FROM_MEM.  Here we obtain the value from the specified
//memory address.  We then go to state STANDARD_RETURN and return the value
//to the debugging tool.  Finally, we go back to state WAITING.

        READ_FROM_MEM: begin

                case (mem_read)

                MEM_ZERO: begin

                        ce_bar = 0;
                        oe_bar = 0;
                        mem_read = MEM_ONE;

                end

                MEM_ONE: begin

                        mem_read = MEM_TWO;

                end

                MEM_TWO: begin

                        mem_read = MEM_THREE;

                end
```

```
                MEM_THREE: begin

                        shift_out = data_in;
                        mem_read = MEM_FOUR;

                end

                MEM_FOUR: begin

                        ce_bar = 1;
                        oe_bar = 1;
                        ack_num = ack_num + 1;
                        v_ack = !v_ack;
                        state = STANDARD_READ;

                end

                default: begin

                end

                endcase

        end

        default: begin

        end

        endcase

end

endmodule
```

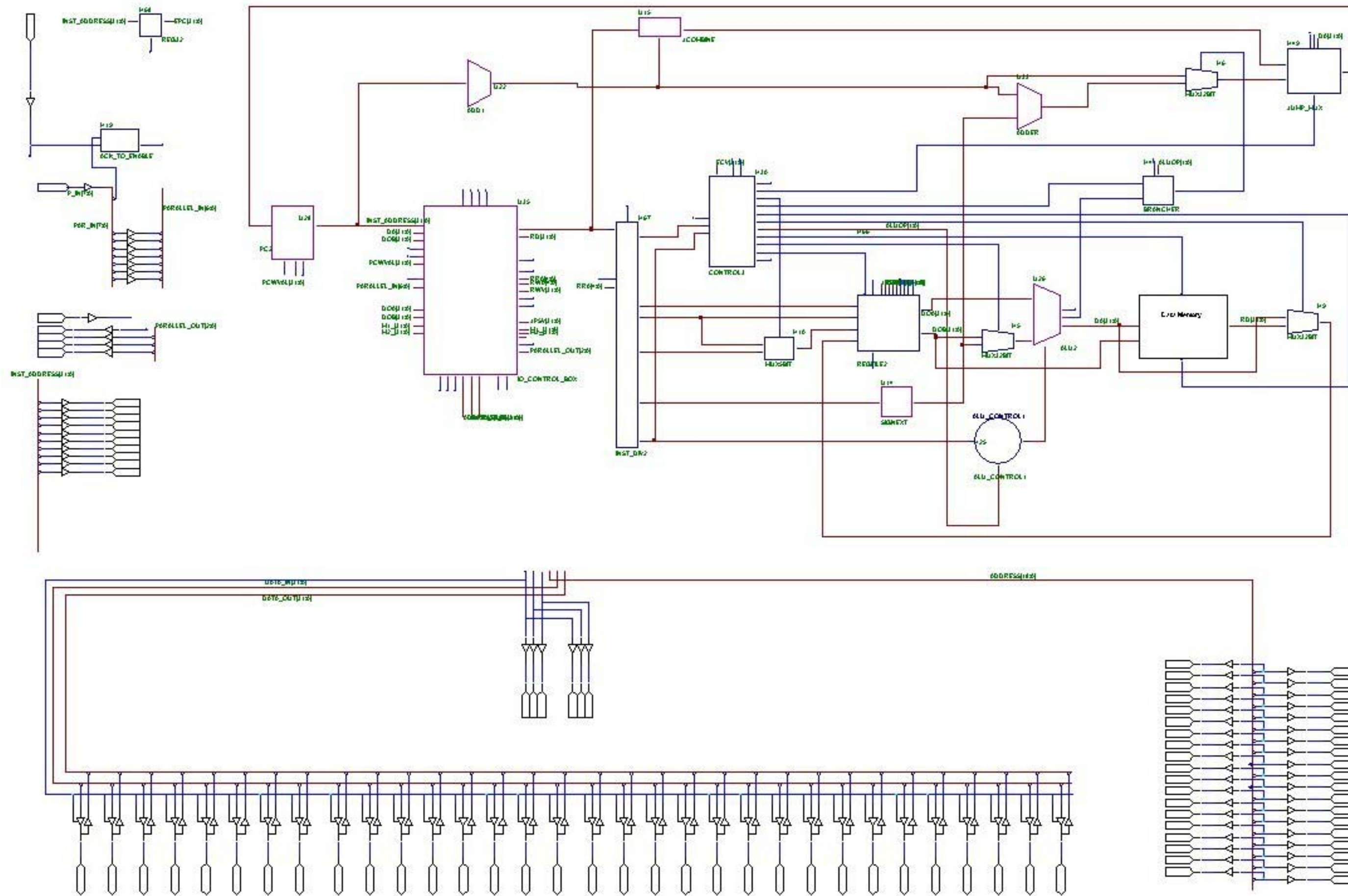
```
//File: jcombine.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//This module combines the lower 28 bits of one bus with the upper
//4 bits of another bus to make a 32-bit result.

module jcombine(in, pc, out);

input    [31:0]   in;
input    [31:0]   pc;
output   [31:0]   out;

assign out = {2'b00, pc[31:28], in[25:0]};

endmodule
```

```
//File: pc2.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//This module is a program counter, which is a 32-bit register
//with asynchronous write capability.

module pc2(clock, we, write_value, D, Q);

input             clock;
input             we;
input    [31:0]   write_value;
input    [31:0]   D;

output   [31:0]   Q;

reg      [31:0]   Q;

always @ (posedge clock or posedge we)

if (we) begin

        Q = write_value;

end else

        Q = D;

endmodule
```

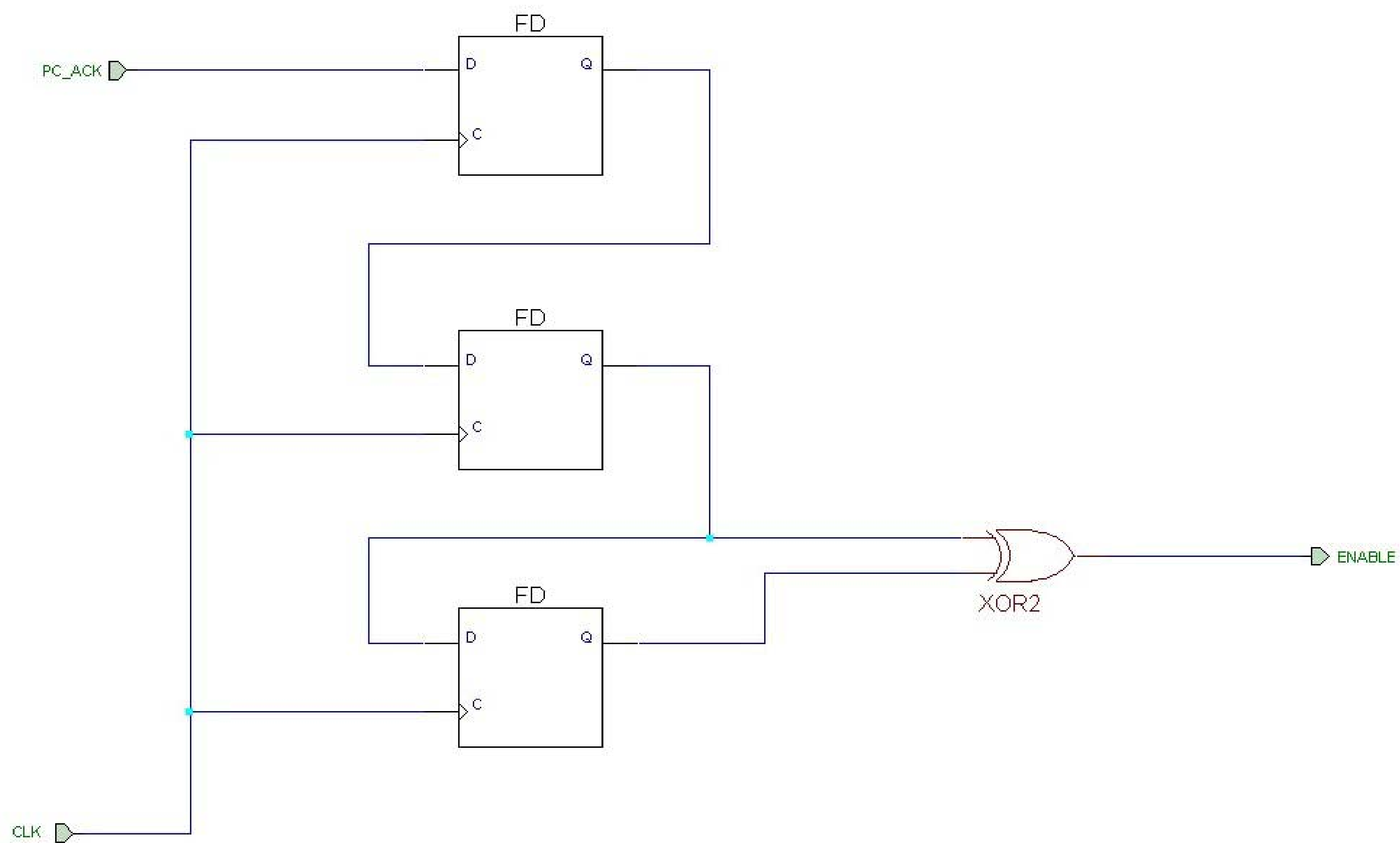
```
//File: signext.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//This module sign extends a 16-bit value to 32 bits.

module SignExt(in, out);

input    [15:0] in;

output   [31:0] out;

assign out[15:0] = in[15:0];
assign out[31:16] = {in[15], in[15], in[15], in[15], in[15], in[15], in[15], in[15], in[15], in[15],
in[15], in[15], in[15], in[15], in[15], in[15]};

endmodule
```

Appendix B – Schematics for the Single Cycle Processor

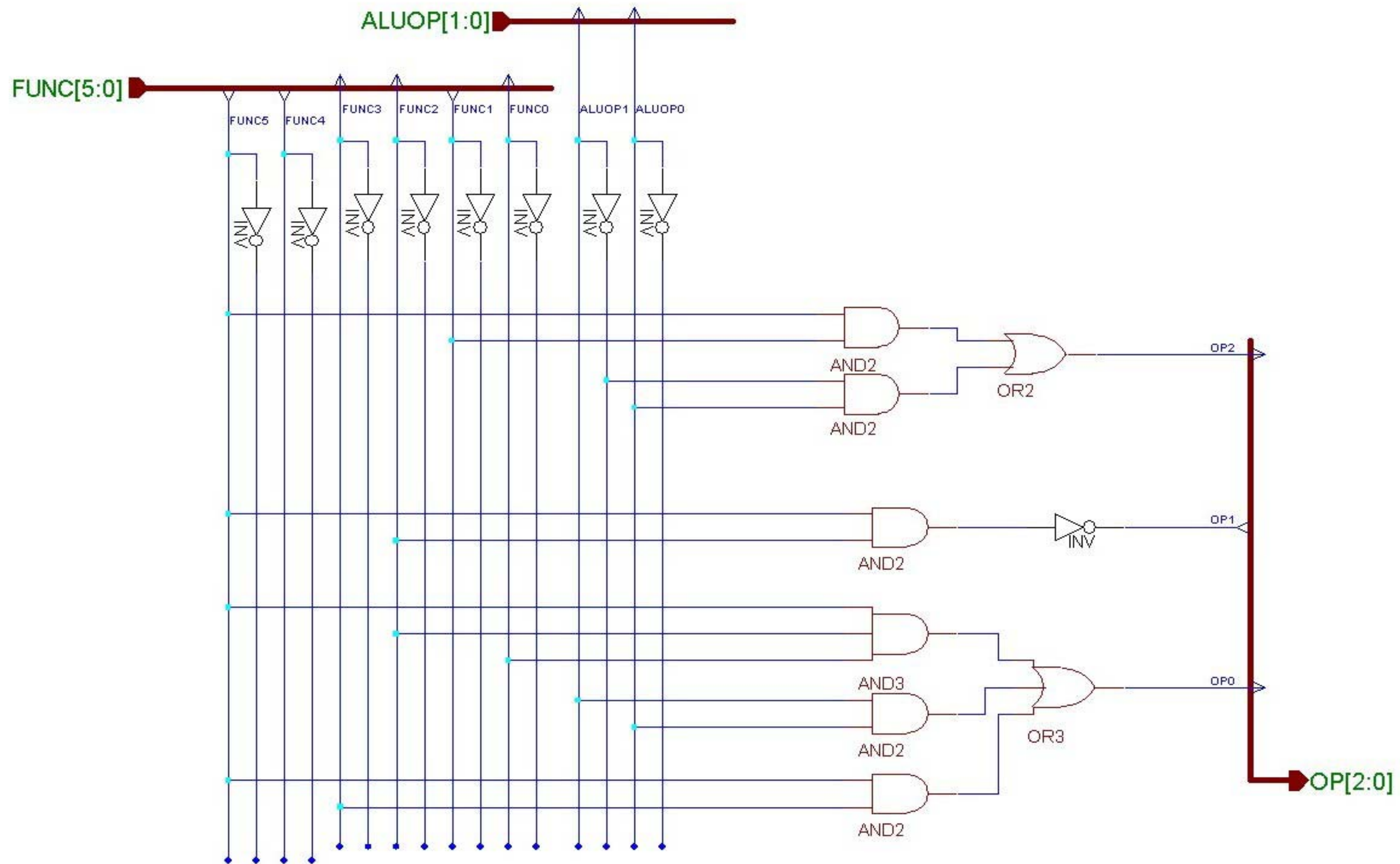
```
Schematic: full.sch
Description: Schematic for the single cycle processor.
```



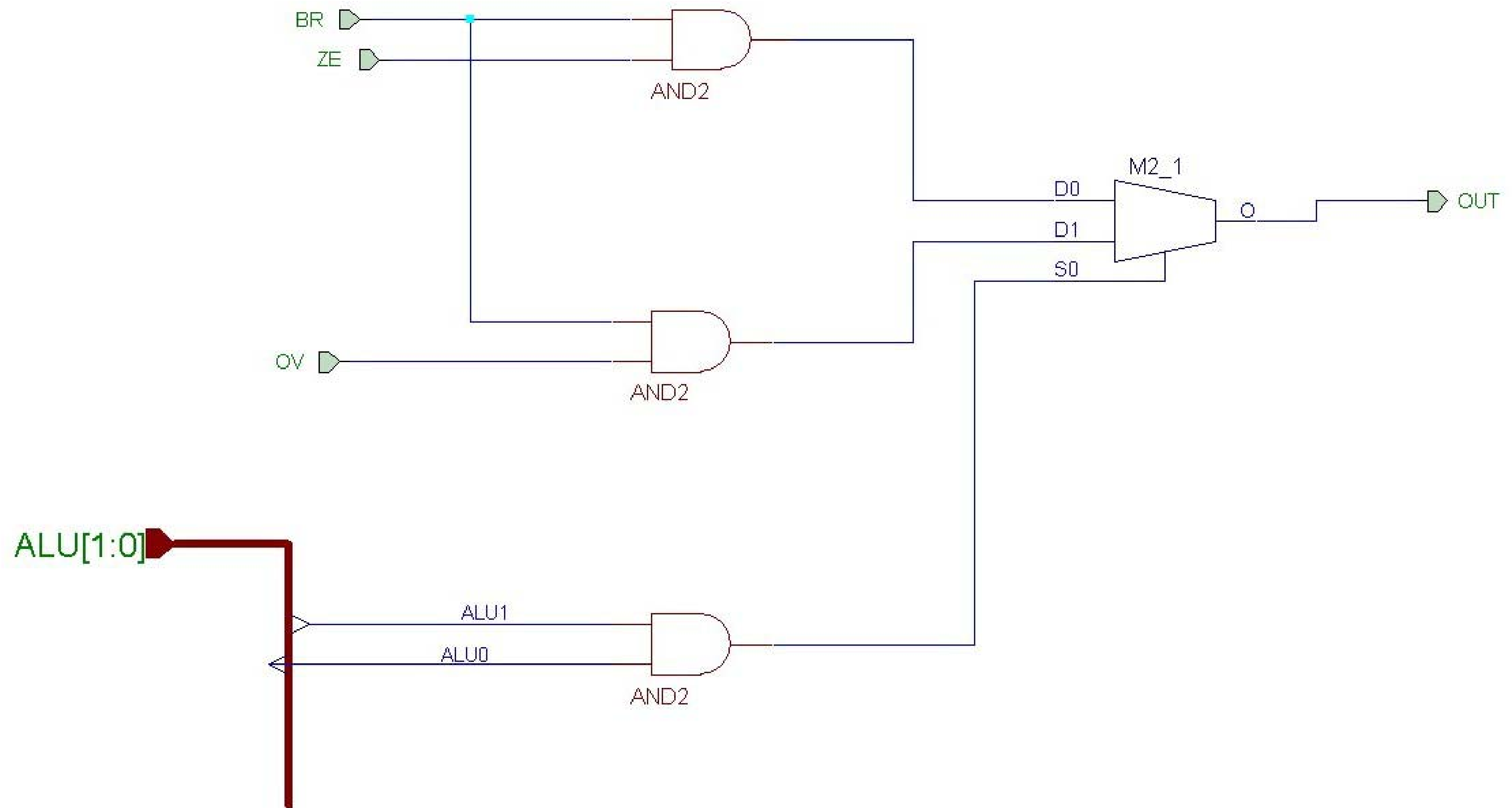
Schematic: ACK_TO_ENABLE.sch
Description: Creates the enable signal for PC to FPGA communications.



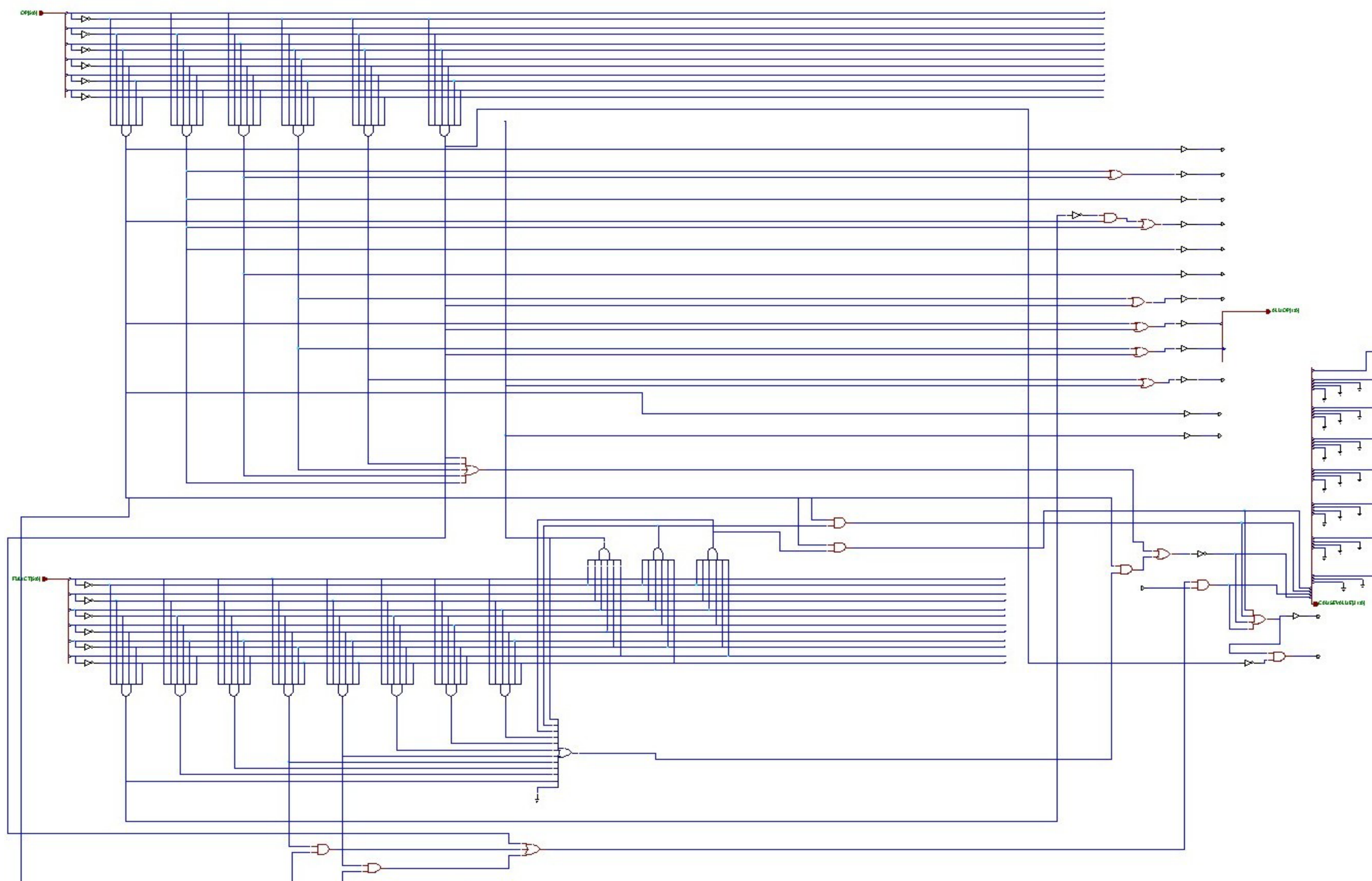
Schematic: ALU_CONTROL1.sch
Description: Control logic for the ALU following the register file.



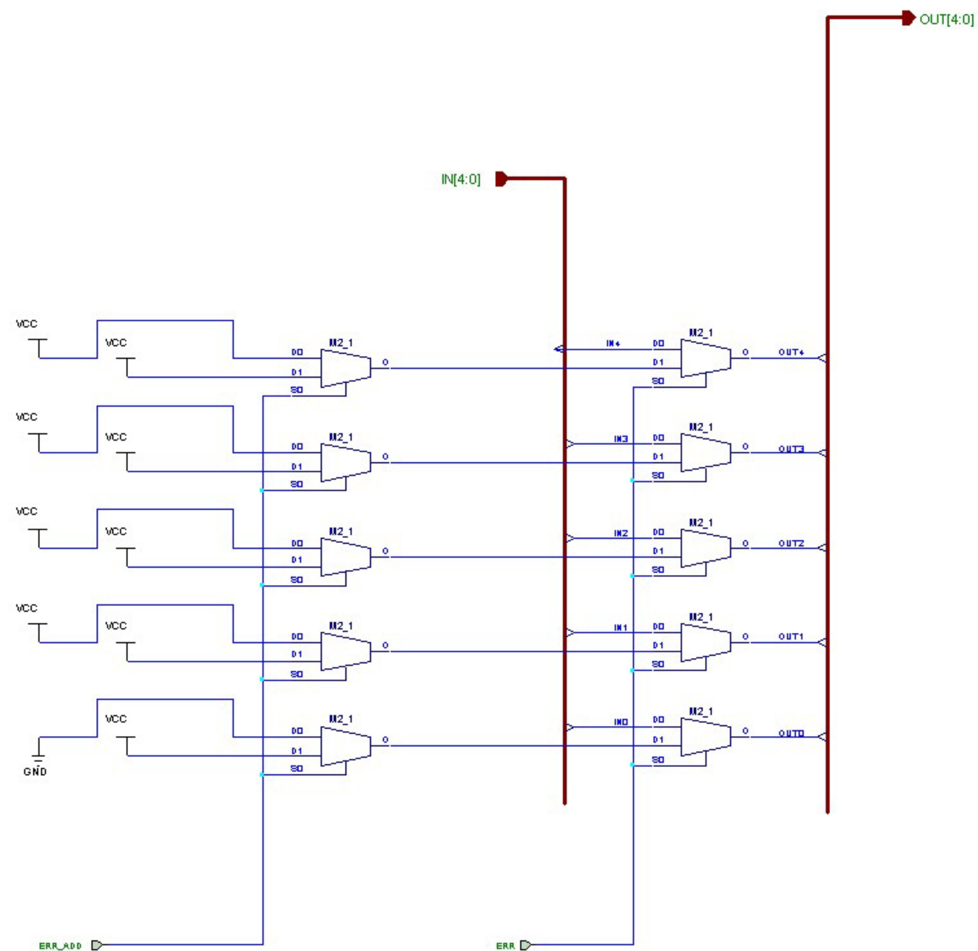
Schematic: BRANCHER.sch
Description: Logic to calculate whether a branch is going to be taken.



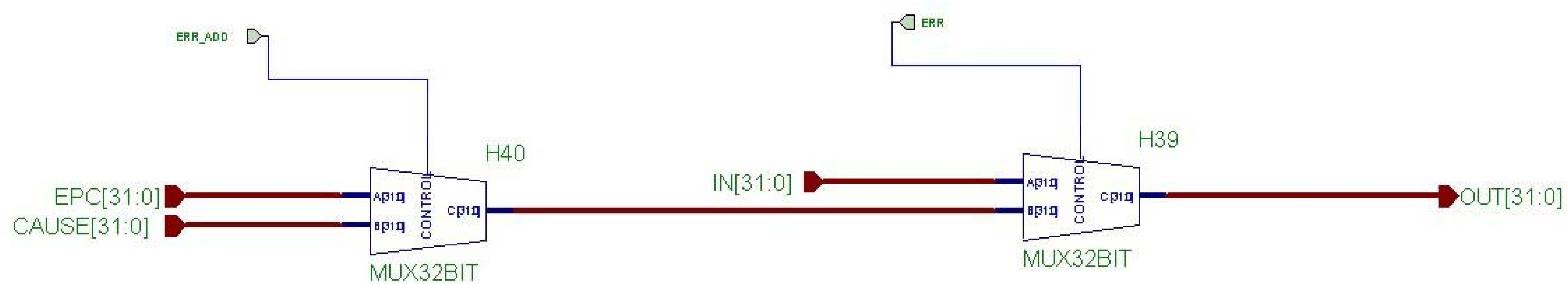
Schematic: CONTROL3.sch
Description: Control signals for the single cycle processor.



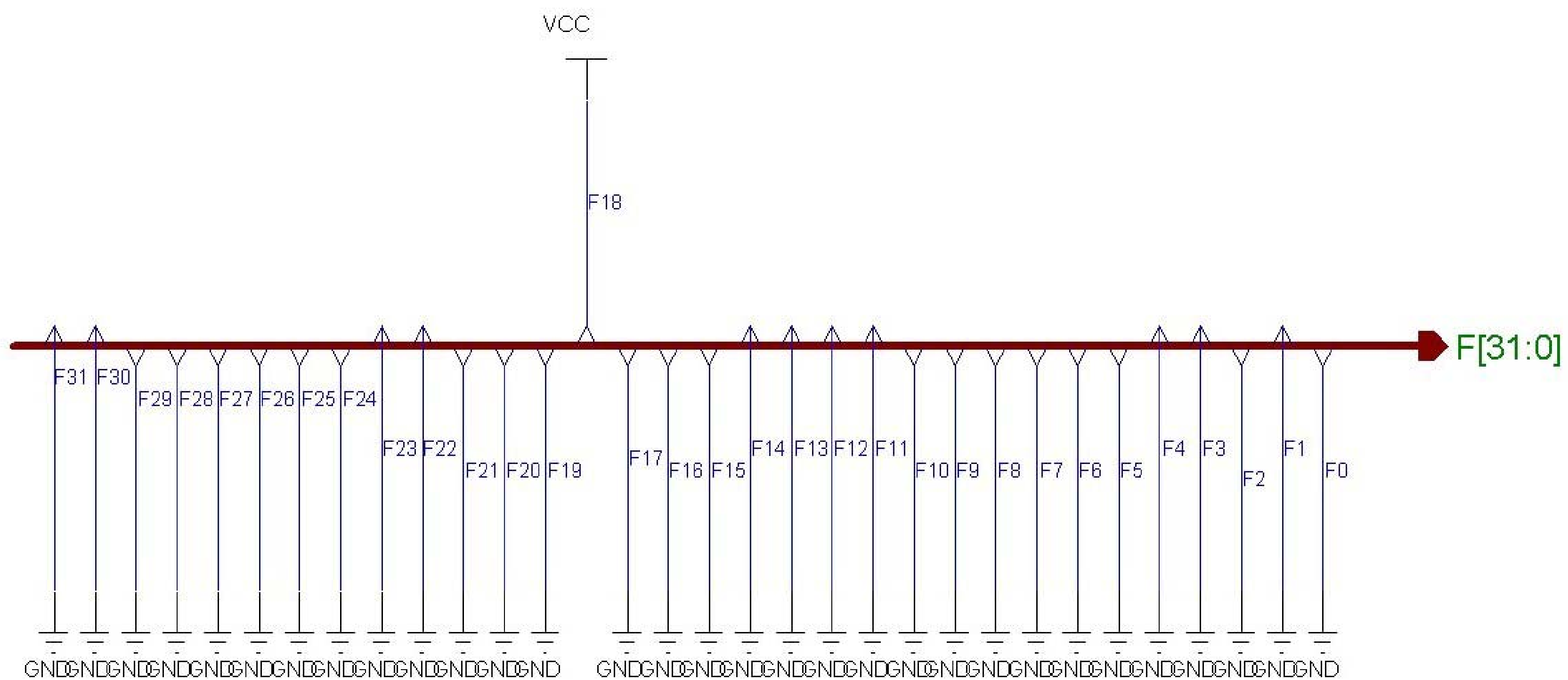
Schematic: ERR_ADD.sch
Description: The address in the register file that is written with information about any unrecognized instruction.



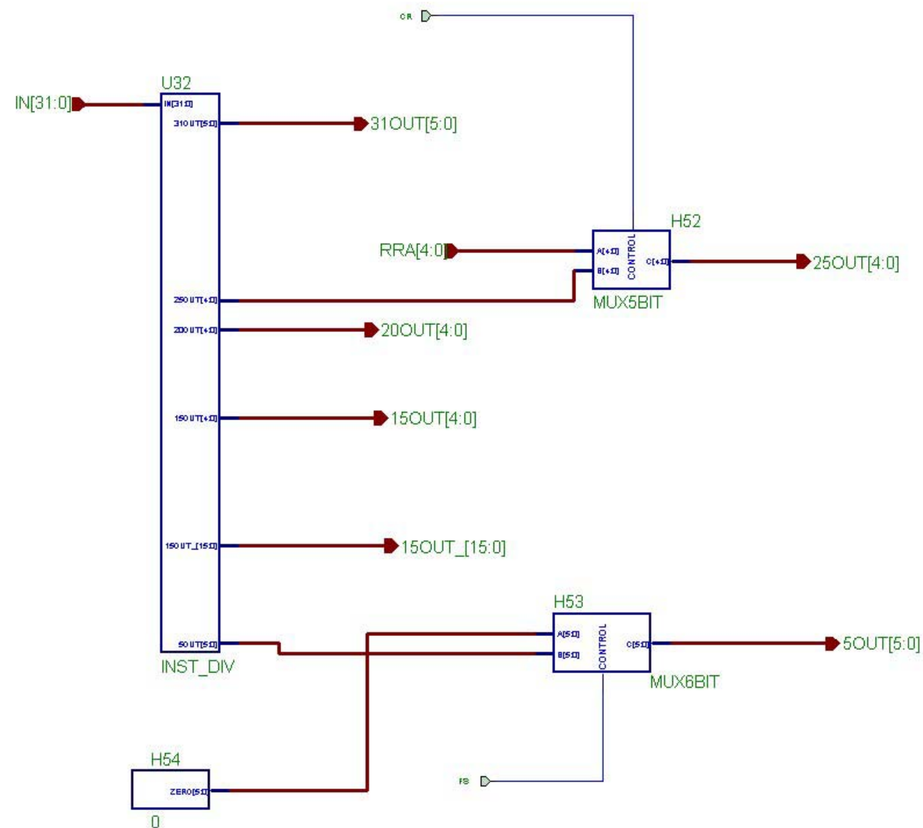
Schematic: ERR_DI.sch
Description: Controls the data to be written to the error register.



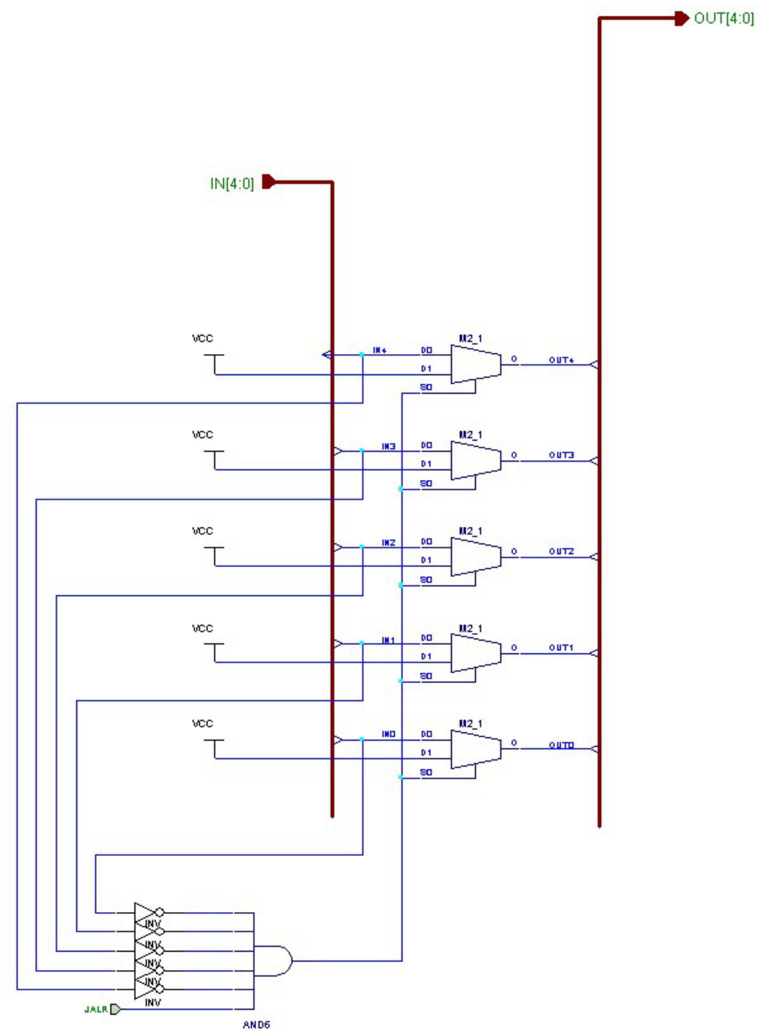
Schematic: EXC_ADD.sch
Description: The memory address of the exception handler.



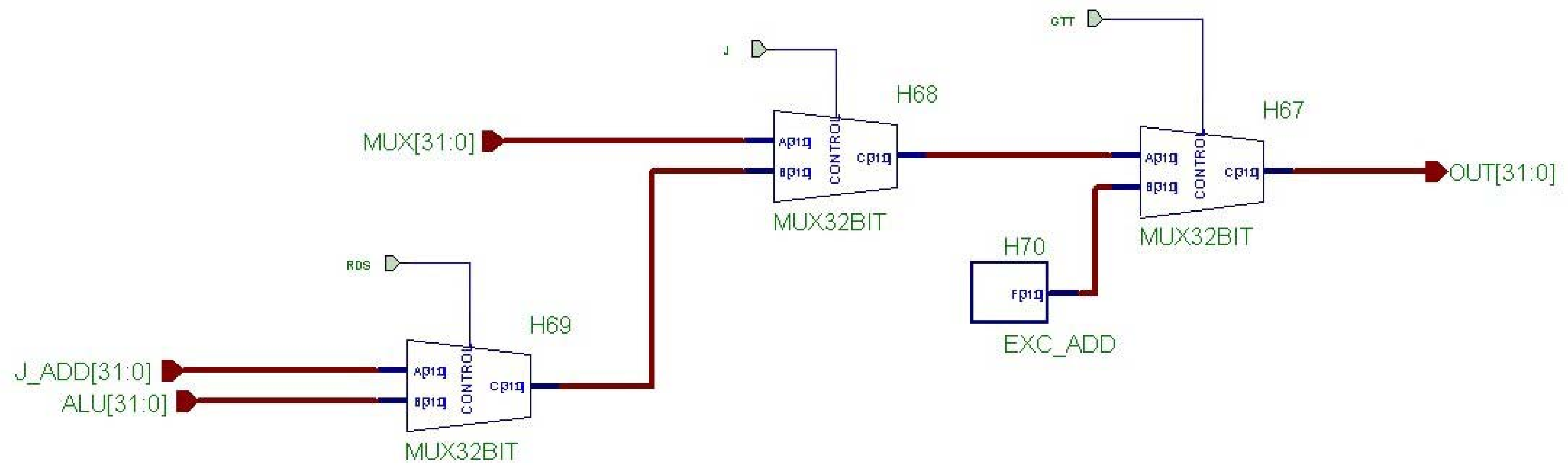
```
Schematic: INST_DIV2.sch
Description: Breaks an instruction into its separate fields.
```



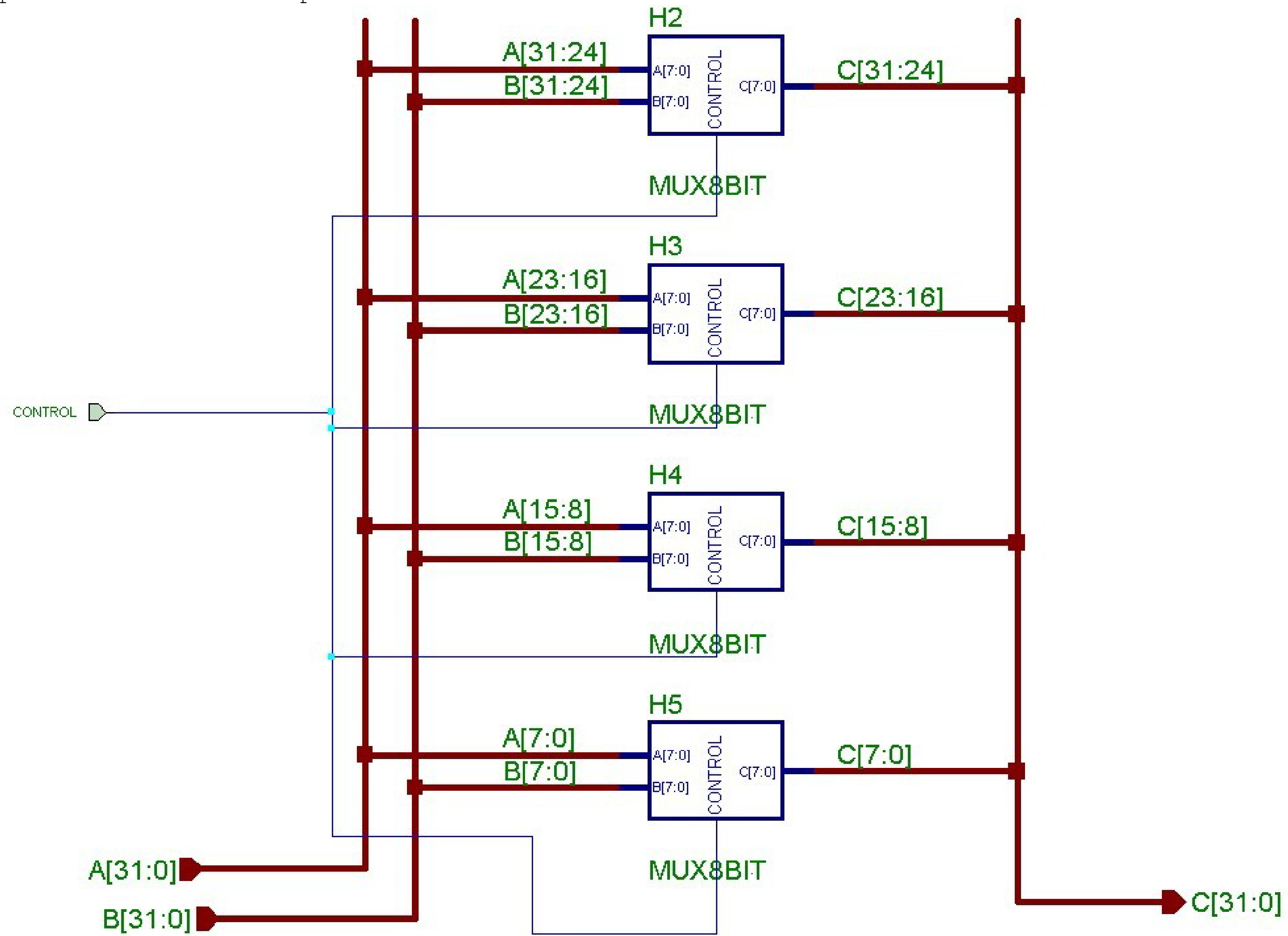
Schematic: JALR_DEFAULT.sch
Description: The default register for a JALR instruction.



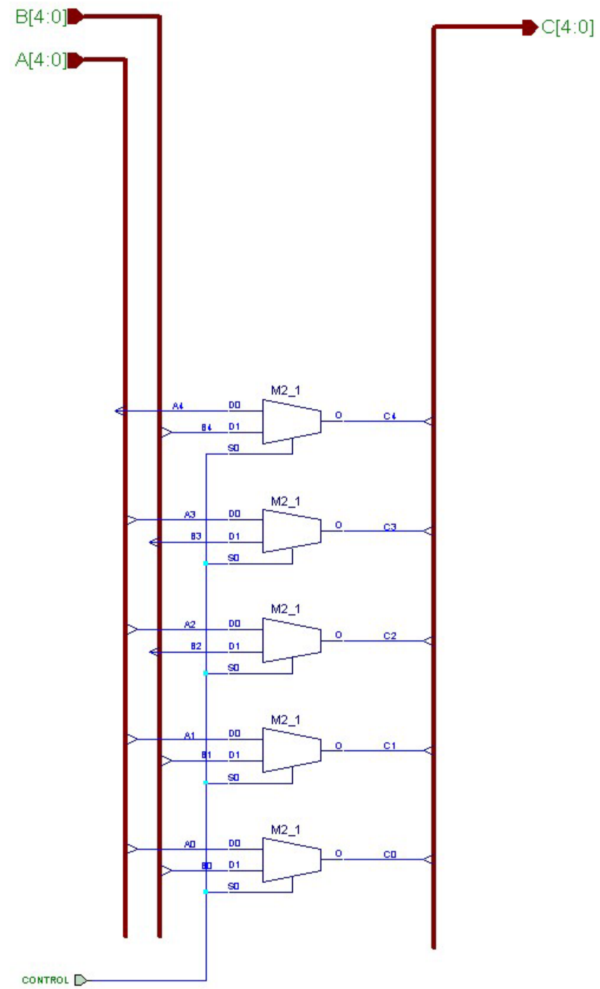
Schematic: JUMP_MUX.sch
Description: Calculates the jump address for a JALR instruction.



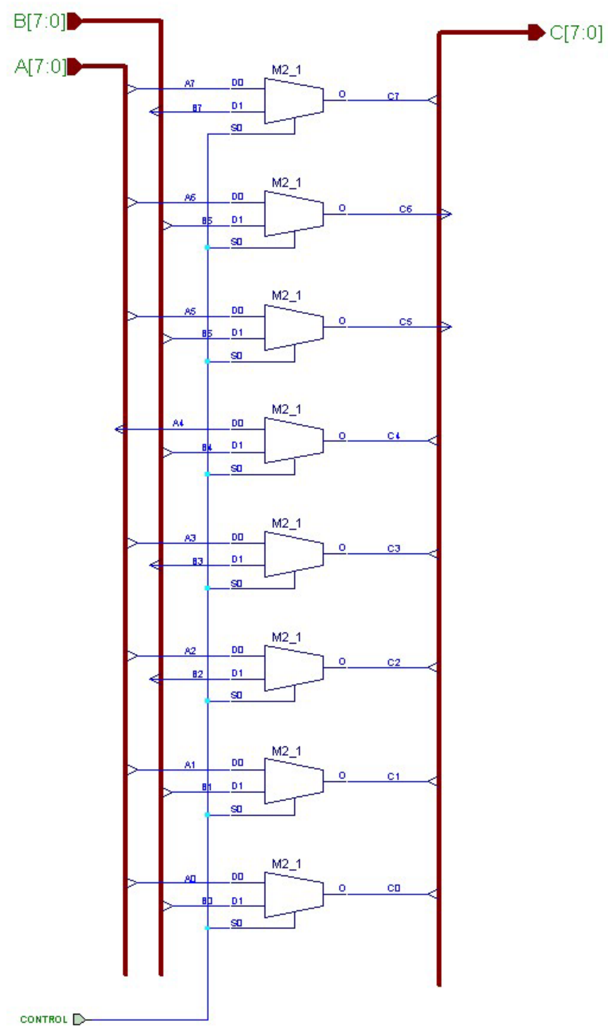
Schematic: MUX32BIT.sch
Description: A 32-bit multiplexor.



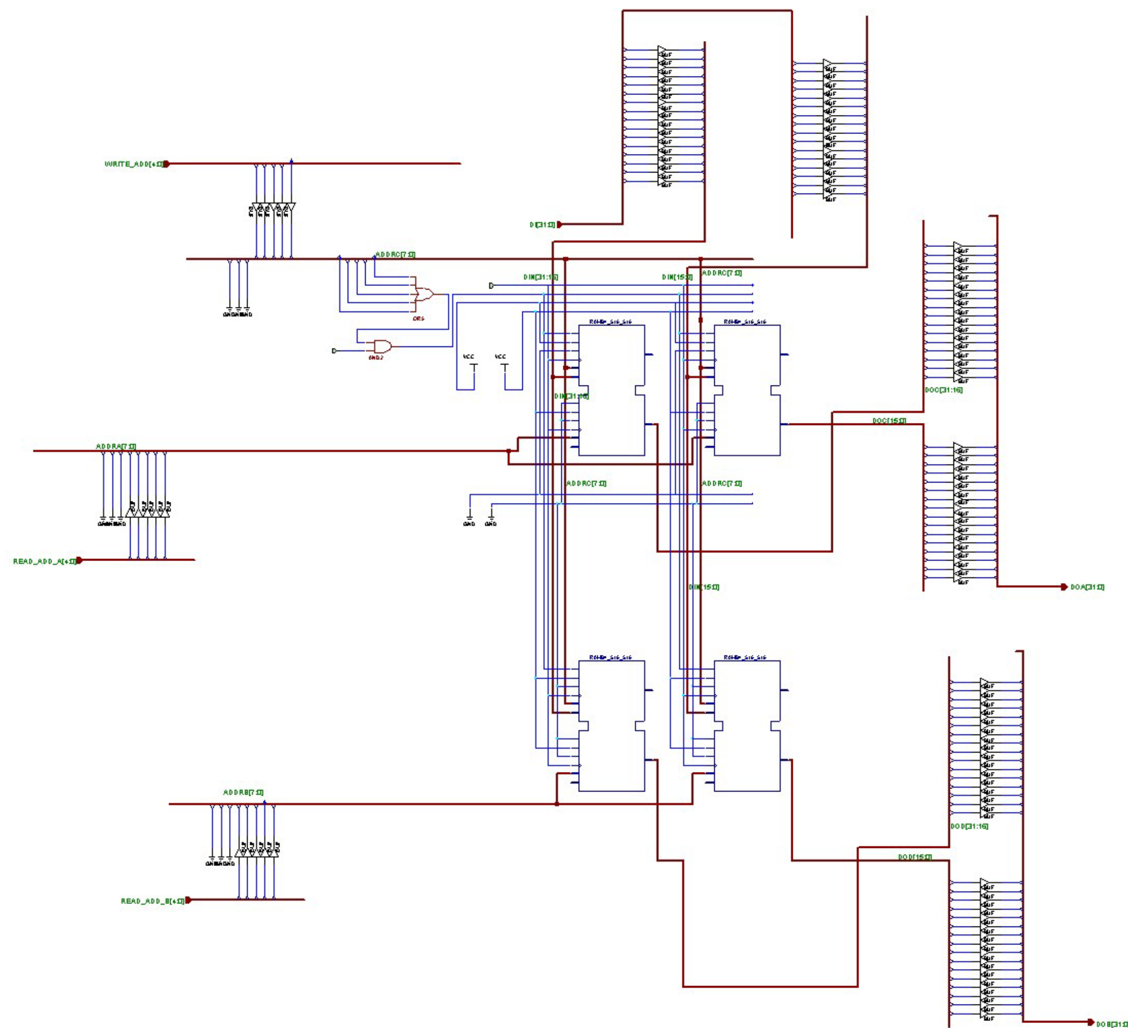
Schematic: MUX5BIT.sch
Description: A 5-bit multiplexor.



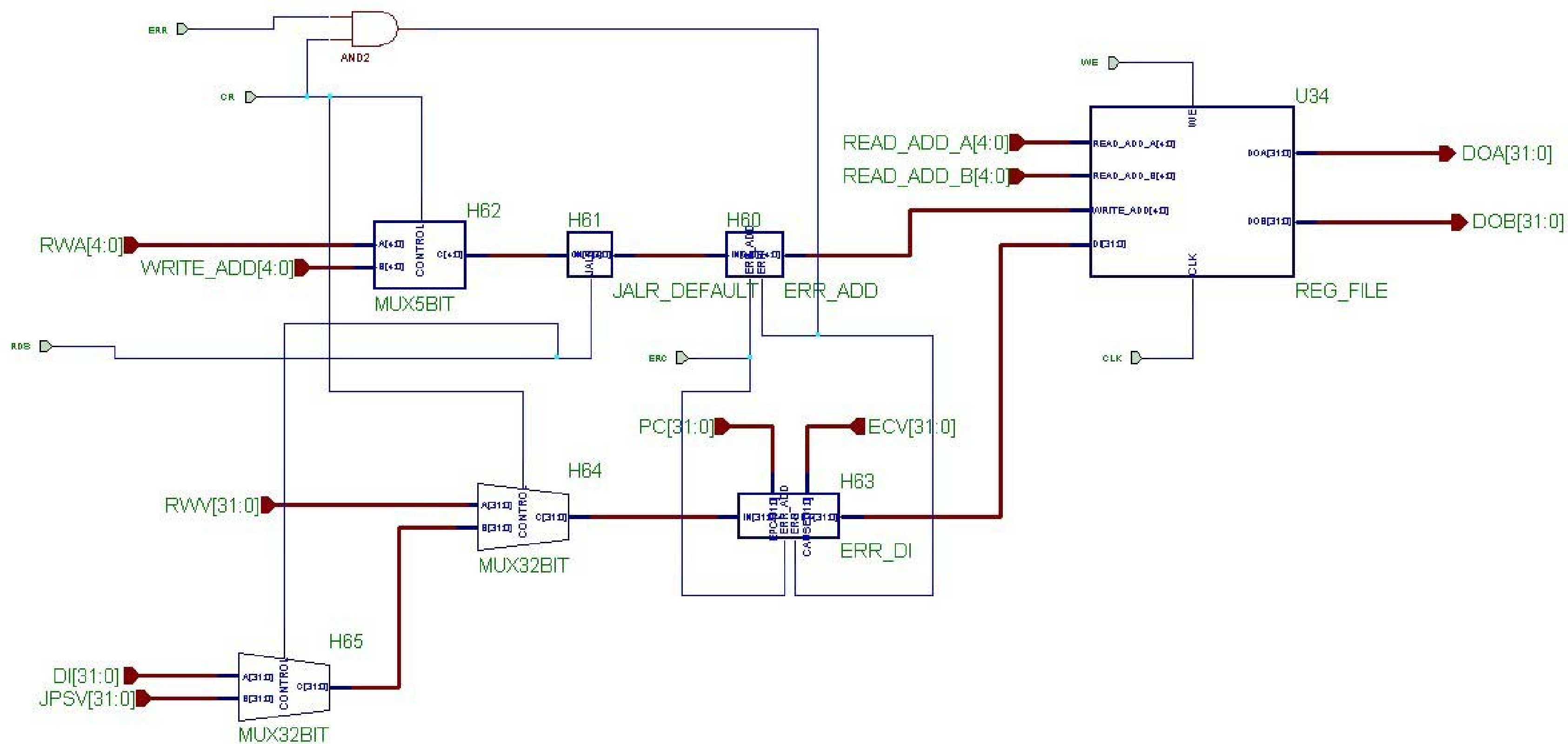
Schematic: MUX8BIT.sch
Description: An 8-bit multiplexor.



Schematic: REG_FILE.sch
Description: The register file, implemented using four Block RAM units.



Schematic: REGFILE2.sch
Description: The complete register file, along with logic to write the error registers.



Appendix C – Verilog Code for the Pipelined Processor

```
//File: branch_calculator.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//This calculates whether a branch is going to occur in the pipelined processor

module branch_calculator(a_in, b_in, b_type, b_take);

input    [31:0]   a_in;
input    [31:0]   b_in;
input             b_type;
output            b_take;

assign b_take = b_type ? (!a_in[31]) : (a_in == b_in);

endmodule
```

```
//File: branch_forwarder.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//This branch forwarder is used in the pipelined MIPS processor.
//It looks at control lines in the ex/mem and id/ex sections
//of the processor and forwards the necessary values to earlier
//elements.

module branch_forwarder(rs, rt, id_ex_rd, ex_mem_rd, id_ex_rw, ex_mem_rw, for_a, for_b);

input    [4:0]    rs;
input    [4:0]    rt;
input    [4:0]    id_ex_rd;
input    [4:0]    ex_mem_rd;
input             id_ex_rw;
input             ex_mem_rw;
output   [1:0]    for_a;
output   [1:0]    for_b;

parameter         r0 = 5'b00000;

assign for_a[1] = ((rs == id_ex_rd) && id_ex_rw && (id_ex_rd != r0));
assign for_a[0] = ((rs == ex_mem_rd) && ex_mem_rw && (ex_mem_rd != r0));

assign for_b[1] = ((rt == id_ex_rd) && id_ex_rw && (id_ex_rd != rs) && (id_ex_rd != r0));
assign for_b[0] = ((rt == ex_mem_rd) && ex_mem_rw && !((id_ex_rd == rt) && id_ex_rw) && (ex_mem_rd !=
r0));

endmodule
```

```
//File: control_staller
//Author: Mark Holland
//Last Modified: May 29, 2002
//This module will stall the control signals if the pipelined processor stalls.

module control_staller(causevalue_in, causevalue_out, err_in, err_out, gototrap_in, gototrap_out,
regdatasrc_in, regdatasrc_out, jump_in, jump_out, regdst_in, regdst_out, branch_in, branch_out,
memread_in, memread_out, memtoreg_in, memtoreg_out, aluop_in, aluop_out, memwrite_in, memwrite_out,
alusrc_in, alusrc_out, regwrite_in, regwrite_out, functsrc_in, functsrc_out, stall);

input    [31:0]   causevalue_in;
output   [31:0]   causevalue_out;

input             err_in;
output            err_out;

input             gototrap_in;
output            gototrap_out;

input             regdatasrc_in;
output            regdatasrc_out;

input             jump_in;
output            jump_out;

input             regdst_in;
output            regdst_out;

input             branch_in;
output            branch_out;

input             memread_in;
output            memread_out;

input             memtoreg_in;
output            memtoreg_out;

input    [1:0]    aluop_in;
output   [1:0]    aluop_out;

input             memwrite_in;
output            memwrite_out;

input             alusrc_in;
output            alusrc_out;

input             regwrite_in;
output            regwrite_out;

input             functsrc_in;
output            functsrc_out;

input             stall;

assign causevalue_out = stall ? 32'b0000_0000_0000_0000_0000_0000_0000_0000 : causevalue_in;
assign err_out = stall ? 0 : err_in;
assign gototrap_out = stall ? 0 : gototrap_in;
assign regdatasrc_out = stall ? 0 : regdatasrc_in;
assign jump_out = stall ? 0 : jump_in;
assign regdst_out = stall ? 0 : regdst_in;
assign branch_out = stall ? 0 : branch_in;
assign memread_out = stall ? 0 : memread_in;
assign memtoreg_out = stall ? 0 : memtoreg_in;
assign aluop_out = stall ? 2'b00 : aluop_in;
assign memwrite_out = stall ? 0 : memwrite_in;
assign alusrc_out = stall ? 0 : alusrc_in;
assign regwrite_out = stall ? 0 : regwrite_in;
assign functsrc_out = stall ? 0 : functsrc_in;

endmodule
```

```
//File: forward_mux.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//This 32-bit mux takes three inputs and decides which
//value to output according to the value outputted by the
//forwarder.

module forward_mux(ex_val, mem_val, rb_val, pick, out_);

input    [31:0]   ex_val;
input    [31:0]   mem_val;
input    [31:0]   rb_val;
input    [1:0]    pick;
output   [31:0]   out_;

assign out_ = pick[1] ? (mem_val) : (pick[0] ? rb_val :ex_val);

endmodule
```

```
//File: forwarder.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//The forwarder forwards signals to the earlier parts of the pipelined
//MIPS processor according to the control lines coming into it.

module forwarder(rs, rt, ex_mem_rd, mem_wb_rd, ex_mem_rw, mem_wb_rw, for_a, for_b);

input    [4:0]    rs;
input    [4:0]    rt;
input    [4:0]    ex_mem_rd;
input    [4:0]    mem_wb_rd;
input             ex_mem_rw;
input             mem_wb_rw;
output   [1:0]    for_a;
output   [1:0]    for_b;

parameter         r0 = 5'b00000;

assign for_a[1] = ((rs == ex_mem_rd) && ex_mem_rw && (ex_mem_rd != r0));
assign for_a[0] = ((rs == mem_wb_rd) && mem_wb_rw && !((ex_mem_rd == rs) && ex_mem_rw) && (mem_wb_rd
!= r0));

assign for_b[1] = ((rt == ex_mem_rd) && ex_mem_rw && (ex_mem_rd != r0));
assign for_b[0] = ((rt == mem_wb_rd) && mem_wb_rw && !((ex_mem_rd == rt) && ex_mem_rw) && (mem_wb_rd
!= r0));

endmodule
```

```
//File: hazard_detecter.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//This module detects hazards and determines whether a stall is necesary.

module hazard_detecter(rs, rt, id_ex_rd, id_ex_mr, stall, branch, ex_mem_rd, ex_mem_mr);

input    [4:0]    rs;
input    [4:0]    rt;
input    [4:0]    id_ex_rd;
input             id_ex_mr;
output            stall;

input             branch;
input    [4:0]    ex_mem_rd;
input             ex_mem_mr;

assign stall = (id_ex_mr && ((id_ex_rd == rs) || (id_ex_rd == rt))) || (ex_mem_mr && branch &&
((ex_mem_rd == rs) || (ex_mem_rd == rt)));

endmodule
```

```
//File: io_control_box.v
//Author: Mark Holland
//Last Modified: May 29, 2002
//Module: io_control_box
//Operation: This control box serves as the interface between the PC and
//the FPGA for all communications.  It also produces the CPU clock and
//the memory clocks that are used during processor operations, as well
//as special control signals needed by the processor for operation.


module io_control_box(inst_address, data_address, write_data, data_in, clock, reset, pc_ack,
parallel_in, inst_in, reg1_in, reg2_in, misc1_in, misc2_in, MR, MW, RW, ERR, misc1_out, misc2_out,
misc3_out, instruction, read_data, address_out, data_out, v_ack, ce_bar, we_bar, oe_bar,
reg_file_clock, parallel_out, pc_write_value, pc_write, pc_clk, reg_write_value, reg_write_address,
reg_read_address, reg_we, clock_running, jalr_pc_store_value, error_reg_choose);

input    [31:0]   inst_address;
input    [31:0]   data_address;
input    [31:0]   write_data;
input    [31:0]   data_in;

input             clock;
input             reset;
input             pc_ack;
input    [6:0]    parallel_in;
//input  [31:0]   pc_in;
input    [31:0]   inst_in;
//input  [31:0]   dataline_in;
input    [31:0]   reg1_in;
input    [31:0]   reg2_in;
//input  [31:0]   alu_in;
input    [31:0]   misc1_in;
input    [31:0]   misc2_in;
input             MR;
input             MW;
input             RW;
input             ERR;

//added 10-12 for debugging
output   [31:0]   misc1_out;
output   [31:0]   misc2_out;
output   [31:0]   misc3_out;

output   [31:0]   instruction;
output   [31:0]   read_data;

output   [18:0]   address_out;
output   [31:0]   data_out;

//output          cpu_clk;
//output          mem_clk;
output            v_ack;
output            ce_bar;
output            we_bar;
output            oe_bar;
output            reg_file_clock;

output   [2:0]    parallel_out;

output   [31:0]   pc_write_value;
output            pc_write;
output            pc_clk;

output   [31:0]   reg_write_value;
output   [4:0]    reg_write_address;
output   [4:0]    reg_read_address;
output            reg_we;

output            clock_running;


//added for jalr
output   [31:0]   jalr_pc_store_value;

//added for exception
```

```
output          error_reg_choose;


//added
wire    cpu_clk;
wire    mem_clk;

//added for debugging 10-12
reg     [31:0]  misc1_out;
reg     [31:0]  misc2_out;
reg     [31:0]  misc3_out;

//added for jalr
reg     [31:0]  jalr_pc_store_value;

//added for exception
reg             error_reg_choose;


reg     [10:0]  cpu_clock;
reg     [3:0]   state;
reg     [69:0]  shift_in;
reg     [63:0]  op_in;
reg     [31:0]  shift_out;
reg     [3:0]   ack_num;
reg             stop_clock;
reg     [18:0]  address_out;

reg             v_ack;
reg             ce_bar;
reg             we_bar;
reg             oe_bar;
reg             reg_file_clock;
reg     [31:0]  pc_write_value;
reg             pc_write;
reg             pc_clk;
reg     [31:0]  reg_write_value;
reg     [4:0]   reg_write_address;
reg     [4:0]   reg_read_address;
reg     [1:0]   reg_write;
reg     [31:0]  data_out;
reg     [2:0]   mem_write;
reg     [1:0]   reg_read;
reg     [2:0]   mem_read;
reg             reg_we;

reg     [31:0]  instruction;
reg     [31:0]  read_data;

reg     [1:0]   pc_num;


assign  parallel_out = shift_out[2:0];
assign  cpu_clk = cpu_clock[3];
assign  mem_clk = cpu_clock[0];

assign  clock_running = |cpu_clock;

parameter       TRUE = 1'b1,
                FALSE = 1'b0;

parameter       CLOCK_RESET = 11'b100_0000_0000,
                CLOCK_SET = 11'b111_1111_1111,
                CLOCK_SET2 = 11'b111_1111_0000;

parameter       ACK_IN_DONE = 4'b1010,
                ACK_OUT_LAST = 4'b1010;

parameter       ZERO4_ = 4'b0000,
                ZERO5_ = 5'b0_0000,
                ZERO11_ = 11'b000_0000_0000,
                ZERO19_ = 19'b000_0000_0000_0000_0000,
                ZERO32_ = 32'b0000_0000_0000_0000_0000_0000_0000_0000,
```

```
                ZERO64_ =
64'b0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000,
                ZERO70_ =
70'b00_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;

parameter       NO_OP = 32'b0000_0000_0000_0000_0000_0000_0010_0000;


parameter       WAITING =        4'b0000,
                GET_OP =         4'b0001,
                RUN_CLOCK =      4'b0010,
                RUN_CLOCK2 =     4'b0011,
                START_CLOCK =    4'b0100,
                STANDARD_READ = 4'b0101,
                WRITE_PC =       4'b0110,
                WRITE_REG =      4'b0111,
                WRITE_MEM =      4'b1000,
                READ_FROM_REG = 4'b1001,
                READ_FROM_MEM = 4'b1010;


parameter       ZERO =           4'b0000,
                ONE =            4'b0001,
                TWO =            4'b0010,
                THREE =  4'b0011,
                FOUR =           4'b0100,
                FIVE =           4'b0101,
                SIX =            4'b0110,
                SEVEN =  4'b0111,
                EIGHT =  4'b1000,
                NINE =           4'b1001,
                TEN =            4'b1010,
                ELEVEN =         4'b1011,
                TWELVE =         4'b1100,
                THIRTEEN =       4'b1101,
                FOURTEEN =       4'b1110,
                FIFTEEN =        4'b1111;

parameter       ZERO6 =          6'b000000,
                ONE6 =           6'b000001,
                TWO6 =           6'b000010,
                THREE6 =         6'b000011,
                FOUR6 =  6'b000100,
                FIVE6 =  6'b000101,
                SIX6 =           6'b000110,
                SEVEN6 =         6'b000111,
                EIGHT6 =         6'b001000,
                NINE6 =  6'b001001,
                TEN6 =           6'b001010,
                ELEVEN6 =        6'b001011,
                TWELVE6 =        6'b001100,
                THIRTEEN6 =      6'b001101,
                FOURTEEN6 =      6'b001110,
                FIFTEEN6 =       6'b001111,
                SIXTEEN6 =       6'b010000,
                SEVENTEEN6 =     6'b010001;

parameter       MEM_ZERO =       3'b000,
                MEM_ONE =        3'b001,
                MEM_TWO =        3'b010,
                MEM_THREE =      3'b011,
                MEM_FOUR =       3'b100;

parameter       REG_ZERO =       2'b00,
                REG_ONE =        2'b01,
                REG_TWO =        2'b10;

parameter       PC_ZERO =        2'b00,
                PC_ONE = 2'b01,
                PC_TWO =         2'b10;

always @ (posedge clock or negedge reset)

//On reset we initialize all values.
```

```
if (!reset) begin

        state = WAITING;
        shift_in = ZERO70_;
        ack_num = ZERO;
        v_ack = 0;
        op_in = ZERO64_;
        state = WAITING;
        cpu_clock = ZERO11_;
        we_bar = 1;
        oe_bar = 1;
        ce_bar = 1;
        reg_file_clock = 0;
        reg_write = REG_ZERO;
        stop_clock = 0;
        shift_out = ZERO32_;
        pc_write_value = ZERO32_;
        pc_write = 0;
        pc_clk = 0;
        reg_write_value = ZERO32_;
        reg_write_address = ZERO5_;
        reg_read_address = ZERO5_;
        data_out = ZERO32_;
        address_out = ZERO19_;
        mem_write = MEM_ZERO;
        reg_read = REG_ZERO;
        mem_read = MEM_ZERO;
        reg_we = 0;
        instruction = ZERO32_;
        read_data = ZERO32_;
        error_reg_choose = 0;
        pc_num = PC_ZERO;
        misc1_out = ZERO32_;
        misc2_out = ZERO32_;
        misc3_out = ZERO32_;

end
else begin

        case (state)

//The WAITING state of our FSM.  The FPGA is waiting for the PC debugging
//tool to give it an instruction.  Instructions get shifted in 7 bits at a
//time (70 bits -> 64 bits total).  When an instruction has been shifted in
//all the way we go to GET_OP and decode the instruction.

        WAITING: begin

                if (stop_clock) begin

                        stop_clock = 0;

                end

                if (pc_ack) begin

                        if (ack_num != ACK_IN_DONE) begin

                                shift_in = {parallel_in, shift_in[69:7]};
                                ack_num = ack_num + 1;
                                v_ack = !v_ack;

                        end
                        else begin

                                op_in = shift_in[63:0];
                                state = GET_OP;
                                ack_num = ZERO4_;
                                v_ack = !v_ack;

                        end

                end
```

```
        end

//State GET_OP.  Here we decode the instruction we got from the debugging tool
//and go to the proper next state.

//      OP   |  COMMAND
// __________|_____________________________________________________________
//      0    |  run clock for specified number of cycles
//      1    |  run clock until told to stop
//      2    |  read program counter
//      3    |  read instruction value
//      4    |  read data value
//      5    |  read register 1 value
//      6    |  read register 2 value
//      7    |  read ALU output value
//      8    |  write the program counter with the specified value
//      9    |  write the specified register with the specified value
//      10   |  write the specified memory address with the specified value
//      11   |  read the value from the specified register
//      12   |  read the value from the specified memory address
//      13   | read misc1 value
//      14   | read misc2 value
//      15   | drive misc1_out value
//      16   |  drive misc2_out value
//      17   | drive misc3_out value

//      0    | 10 sends for OP, 1 send to choose OP, 11 sends to receive return value

        GET_OP: begin

                if (pc_ack) begin

                        case (op_in[63:58])

//OP = 0: We set the clock to the number of cycles to run and go to state
//RUN_CLOCK.

                        ZERO6: begin
//changed to 4'b0000 for pipelining 9/25/01
                                cpu_clock = {1'b0, op_in[5:0], 4'b0000};
//                              ce_bar = 0;
                                state = RUN_CLOCK;

                        end

//OP = 1: We set the clock to running and go to START_CLOCK.

                        ONE6: begin

                                cpu_clock = CLOCK_SET2;
//                              ce_bar = 0;
                                v_ack = !v_ack;
                                state = START_CLOCK;

                        end

//OP = 2: We put the program counter value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                        TWO6: begin

                                shift_out = inst_address;
                                ack_num = ack_num + 1;
                                v_ack = !v_ack;
                                state = STANDARD_READ;

                        end

//OP = 3: We put the instruction value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                        THREE6: begin

                                shift_out = inst_in;
                                ack_num = ack_num + 1;
```

```
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 4: We put the data value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                FOUR6: begin

                                        shift_out = read_data;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 5: We put the first register's value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                FIVE6: begin

                                        shift_out = reg1_in;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 6: We put the second register's value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                SIX6: begin

                                        shift_out = reg2_in;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 7: We put the ALU output's value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                SEVEN6: begin

                                        shift_out = data_address;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 8: We grab the value that we wish to write to the program counter
//and go to state WRITE_PC.

                                EIGHT6: begin

//changed inst. to make sure that new value propogates through
                                        instruction = NO_OP;
                                        pc_num = PC_ZERO;
                                        pc_write_value = {13'b0_0000_0000_0000, shift_in[50:32]};
                                        state = WRITE_PC;

                                end

//OP = 9: We grab the address of the destination register and the write
//value and then go to state WRITE_REG.

                                NINE6: begin

                                        reg_write_value = shift_in[31:0];
                                        reg_write_address = shift_in[36:32];
                                        reg_write = REG_ZERO;
                                        reg_we = 1;
```

```
                                        state = WRITE_REG;

                                end

//OP = 10: We grab the address of memory and the write value and then
//go to state WRITE_MEM.

                                TEN6: begin

                                        data_out = shift_in[31:0];
                                        address_out = shift_in[50:32];
                                        mem_write = MEM_ZERO;
                                        state = WRITE_MEM;

                                end

//OP = 11: We grab the register address we wish to read and go to
//state READ_FROM_REG.

                                ELEVEN6: begin

                                        reg_read_address = shift_in[36:32];
                                        reg_read = REG_ZERO;
                                        state = READ_FROM_REG;

                                end

//OP = 12: We grab the memory address we wish to read from and go
//to state READ_FROM_MEM.

                                TWELVE6: begin

                                        address_out = shift_in[50:32];
                                        mem_read = MEM_ZERO;
                                        state = READ_FROM_MEM;

                                end

//OP = 13: We put the misc1 value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                THIRTEEN6: begin

                                        shift_out = misc1_in;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 14: We put the misc2 value on the shift_out bus and go
//to state STANDARD_READ to shift out the value.

                                FOURTEEN6: begin

                                        shift_out = misc2_in;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end

//OP = 15: We grab the misc1 value and output it to the CPU.  This
//line is used solely for debugging.

                                FIFTEEN6: begin

                                        misc1_out = shift_in[31:0];
                                        shift_out = shift_in[31:0];
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;

                                end
```

```
//OP = 16: We grab the misc2 value and output it to the CPU.  This
//line is used solely for debugging.

                          SIXTEEN6: begin

                                  misc2_out = shift_in[31:0];
                                  shift_out = shift_in[31:0];
                                  ack_num = ack_num + 1;
                                  v_ack = !v_ack;
                                  state = STANDARD_READ;

                          end

//OP = 17: We grab the misc3 value and output it to the CPU.  This
//line is used solely for debugging.

                          SEVENTEEN6: begin

                                  misc3_out = shift_in[31:0];
                                  shift_out = shift_in[31:0];
                                  ack_num = ack_num + 1;
                                  v_ack = !v_ack;
                                  state = STANDARD_READ;

                          end

                          default: begin

                          end

                          endcase

                  end

          end

//State RUN_CLOCK.  Here we run the clock for the specified number
//of cycles.

//cpu_clock[10:4] is the number of cycles to run
//cpu_clock[3] is the main CPU clock
//cpu_clock[0] is the memory clock which is used for controlling
//        memory references

//The cpu_clock counts down until it equals zero, at which point
//the proper number of cycles has occured.  We then return to
//state WAITING.

//The case statement controls the memory references, making the
//instruction fetch, register manipulations, and data store/loads
//occur.  The standard CPU control lines control which of these
//occur.


          RUN_CLOCK: begin

                  if (cpu_clock != ZERO11_) begin

                          case (cpu_clock[3:0])

                          ZERO: begin

                                  ce_bar = 0;
                                  data_out = write_data;
                                  address_out = data_address[18:0];
                                  jalr_pc_store_value = inst_address + 1;
                                  pc_clk = 0;


                          end

                          FIFTEEN: begin

                                  error_reg_choose = 0;
                                  reg_we = RW || ERR;
```

```
                                we_bar = !MW;
//moved for pipelining                          pc_clk = 1;

                        end

                        FOURTEEN: begin

                                reg_file_clock = RW || ERR;
//moved for pipelining                          pc_clk = 0;

                        end

                        THIRTEEN: begin

                                error_reg_choose = 1;
                                reg_file_clock = 0;
                                reg_we = ERR;
                                we_bar = 1;

                        end

                        TWELVE: begin

                                reg_file_clock = ERR;
                                ce_bar = 1;
                                address_out = inst_address[18:0];

                        end

                        ELEVEN: begin

                                reg_file_clock = 0;
                                ce_bar = 0;
                                oe_bar = 0;
                                reg_we = 0;

                        end

                        TEN: begin

                        end

                        NINE: begin

                        end

                        EIGHT: begin

                                instruction = data_in;

                        end

                        SEVEN: begin

                                ce_bar = 1;
                                oe_bar = 1;
                                reg_file_clock = 1;
                        end

                        SIX: begin

                                reg_file_clock = 0;
                                address_out = data_address[18:0];
                        end

                        FIVE: begin

                                oe_bar = !MR;
                                ce_bar = !MR;
                        end

                        FOUR: begin

                        end
```

```
                        THREE: begin

                        end

                        TWO: begin

                                if(MR) begin
                                        read_data = data_in;
                                end

                        end

                        ONE: begin

                                oe_bar = 1;
                                ce_bar = 1;
                                pc_clk = 1;

//moved to 0 for pipelining                             data_out = write_data;

//moved to 0 for pipelining                             jalr_pc_store_value = inst_address + 1;

                        end

                        default: begin

                        end

                        endcase

                        cpu_clock = cpu_clock - 1;

                end
                else begin

                        ce_bar = 1;
//added for pipelining
                        pc_clk = 0;
                        shift_out = inst_address;
                        ack_num = ack_num + 1;
                        v_ack = !v_ack;
                        state = STANDARD_READ;

                end

        end

//state START_CLOCK.  This operates the same as RUN_CLOCK, with one exception.
//In this state we run the clock until we receive a signal from the PC saying
//to stop.  This clock thus runs indefinitely.

//When the stop signal is given, the current cycle is allowed to finish and then
//we go to state WAITING.

        START_CLOCK: begin

                if (pc_ack) begin

                        stop_clock = 1;

                end

                case (cpu_clock[3:0])

                        ZERO: begin

                                if (stop_clock) begin

                                        ce_bar = 1;
//                                      cpu_clock = ZERO11_;
                                        shift_out = inst_address;
                                        ack_num = ack_num + 1;
                                        v_ack = !v_ack;
                                        state = STANDARD_READ;
```

```
                        end

                        else begin

                        ce_bar = 0;
                        data_out = write_data;
                        address_out = data_address[18:0];
                        jalr_pc_store_value = inst_address + 1;
                        pc_clk = 0;

                        end

                end

                FIFTEEN: begin

                        error_reg_choose = 0;
                        reg_we = RW || ERR;
                        we_bar = !MW;
//moved for pipelining                  pc_clk = 1;

                end

                FOURTEEN: begin

                        reg_file_clock = RW || ERR;
//moved for pipelining                  pc_clk = 0;

                end

                THIRTEEN: begin

                        error_reg_choose = 1;
                        reg_file_clock = 0;
                        reg_we = ERR;
                        we_bar = 1;

                end

                TWELVE: begin

                        reg_file_clock = ERR;
                        ce_bar = 1;
                        address_out = inst_address[18:0];

                end

                ELEVEN: begin

                        reg_file_clock = 0;
                        ce_bar = 0;
                        oe_bar = 0;
                        reg_we = 0;

                end

                TEN: begin

                end

                NINE: begin

                end

                EIGHT: begin

                        instruction = data_in;

                end

                SEVEN: begin

                        ce_bar = 1;
                        oe_bar = 1;
                        reg_file_clock = 1;
```

```
			end

			SIX: begin

				reg_file_clock = 0;
				address_out = data_address[18:0];
			end

			FIVE: begin

				oe_bar = !MR;
				ce_bar = !MR;
			end

			FOUR: begin

			end

			THREE: begin

			end

			TWO: begin

				if(MR) begin
					read_data = data_in;
				end

			end

			ONE: begin

				oe_bar = 1;
				ce_bar = 1;
				pc_clk = 1;
//moved to 0 for pipelining				data_out = write_data;
//moved to 0 for pipelining				jalr_pc_store_value = inst_address + 1;
			end

			default: begin

			end

		endcase


		case ({stop_clock, cpu_clock[3:0]})

			5'b10000: begin

				cpu_clock = ZERO11_;

			end

			default: begin

					if (cpu_clock == CLOCK_RESET) begin

						cpu_clock = CLOCK_SET;

					end

					else begin

						cpu_clock = cpu_clock - 1;

					end

				end

		endcase
```

```
        end

//state STANDARD_READ.  Whenever we want to return a value to the PC, we
//do it in this state.  We send bits to the PC 3 at a time, offering 3 new
//bits upon requests from the PC.  This returns one 32-bit value to the PC
//debugging tool.

//When completed, we return to state WAITING.

        STANDARD_READ: begin

                if (pc_ack) begin

                        if (ack_num != ACK_OUT_LAST) begin

                                shift_out = {3'b0, shift_out[31:3]};
                                ack_num = ack_num + 1;
                                v_ack = !v_ack;

                        end

                        else begin

                                shift_out = {3'b0, shift_out[31:3]};
                                ack_num = ZERO4_;
                                v_ack = !v_ack;
                                state = WAITING;

                        end
                end

        end

//state WRITE_PC.  Here we write the specified value to the program counter.
//This is done by first setting the write value (done in GET_OP), and then
//pulling high the write enable line.

//After writing the value, we go to STANDARD_READ and return the PC value
//to the debugging tool to ensure that it was written correctly.  From there
//we go to state WAITING.

        WRITE_PC: begin

                case (pc_num)

                PC_ZERO: begin

                        pc_write = 1;
                        pc_num = PC_ONE;

                end

                PC_ONE: begin

                        pc_write = 0;
                        pc_num = PC_TWO;

                end

                PC_TWO: begin

                        pc_write_value = ZERO32_;
                        shift_out = inst_address;
                        ack_num = ack_num + 1;
                        v_ack = !v_ack;
                        state = STANDARD_READ;
                        pc_num = PC_ZERO;

                end
                endcase

        end

//state WRITE_REG.  Here we write the specified register with the
//specified value.  We then go to STANDARD_READ to return the value
```

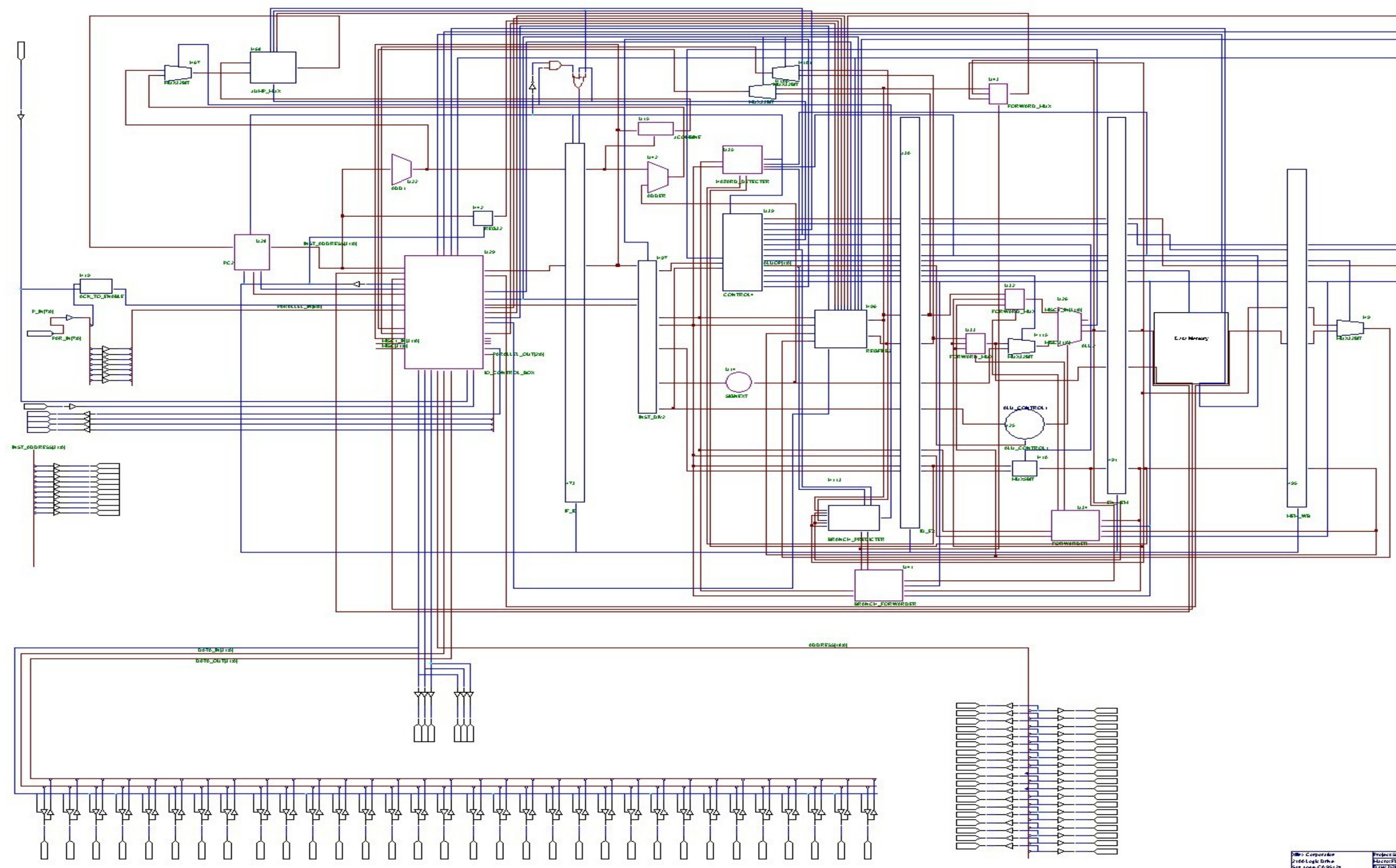
```
//to the PC, and then go to state WAITING.

        WRITE_REG: begin

                case (reg_write)

                REG_ZERO: begin

                        reg_file_clock = 1;
                        reg_write = REG_ONE;

                end

                REG_ONE: begin

                        reg_file_clock = 0;
                        reg_write = REG_TWO;

                end

                REG_TWO: begin

                        reg_we = 0;
                        reg_write_value = ZERO32_;
                        reg_write_address = ZERO5_;
                        reg_write = REG_ZERO;
                        reg_read_address = shift_in[36:32];
                        reg_read = REG_ZERO;
                        state = READ_FROM_REG;

                end

                default: begin

                end

                endcase

        end

//state WRITE_MEM.  Here we write the specified memory address with the
//specified value.  We then go to STANDARD_READ to return the value
//to the PC, and then go to state WAITING.

        WRITE_MEM: begin

                case(mem_write)

                MEM_ZERO: begin

                        ce_bar = 0;
                        mem_write = MEM_ONE;

                end

                MEM_ONE: begin

                        we_bar = 0;
                        mem_write = MEM_TWO;

                end

                MEM_TWO: begin

                        mem_write = MEM_THREE;

                end

                MEM_THREE: begin

                        we_bar = 1;
                        mem_write = MEM_FOUR;

                end
```

```
                MEM_FOUR: begin

                        ce_bar = 1;
//                      mem_write = ZERO4_;
//                      data_out = ZERO32_;
                        mem_read = MEM_ZERO;
                        state = READ_FROM_MEM;

                end

                default: begin

                end

                endcase

        end

//state READ_FROM_REG.  Here we obtain the value from the specified
//register.  We then go to state STANDARD_RETURN and return the value
//to the debugging tool.  Finally, we go back to state WAITING.

        READ_FROM_REG: begin

                case (reg_read)

                REG_ZERO: begin

                        reg_file_clock = 1;
                        reg_read = REG_ONE;

                end

                REG_ONE: begin

                        reg_file_clock = 0;
                        reg_read = REG_TWO;

                end

                REG_TWO: begin

                        shift_out = reg1_in;
                        reg_read_address = ZERO5_;
                        ack_num = ack_num + 1;
                        v_ack = !v_ack;
                        state = STANDARD_READ;

                end

                default: begin

                end

                endcase

        end

//state READ_FROM_MEM.  Here we obtain the value from the specified
//memory address.  We then go to state STANDARD_RETURN and return the value
//to the debugging tool.  Finally, we go back to state WAITING.

        READ_FROM_MEM: begin

                case (mem_read)

                MEM_ZERO: begin

                        ce_bar = 0;
                        oe_bar = 0;
                        mem_read = MEM_ONE;

                end

                MEM_ONE: begin
```

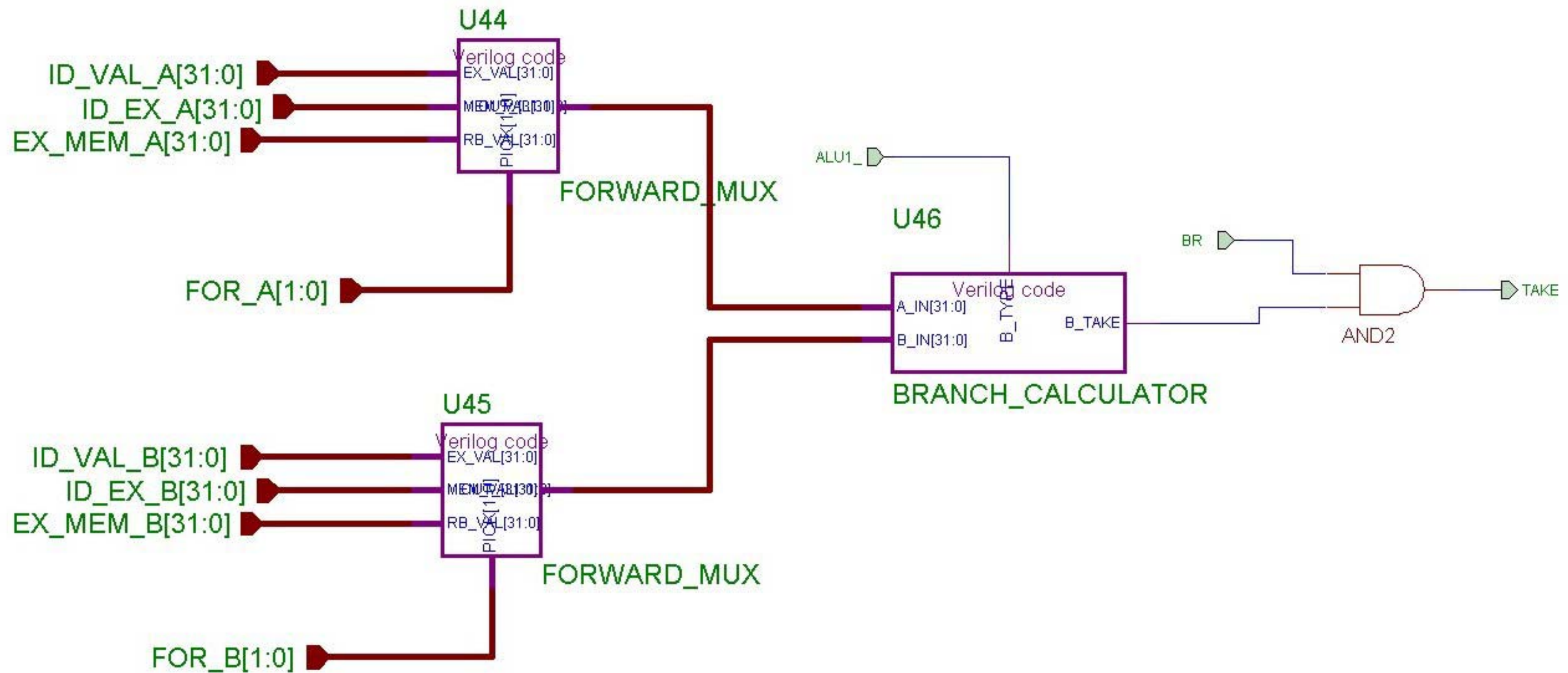
```
                        mem_read = MEM_TWO;

                end

                MEM_TWO: begin

                        mem_read = MEM_THREE;

                end

                MEM_THREE: begin

                        shift_out = data_in;
                        mem_read = MEM_FOUR;

                end

                MEM_FOUR: begin

                        ce_bar = 1;
                        oe_bar = 1;
                        ack_num = ack_num + 1;
                        v_ack = !v_ack;
                        state = STANDARD_READ;

                end

                default: begin

                end

                endcase

        end

        default: begin

        end

        endcase

end

endmodule
```

Appendix D – Schematics for the Pipelined Processor

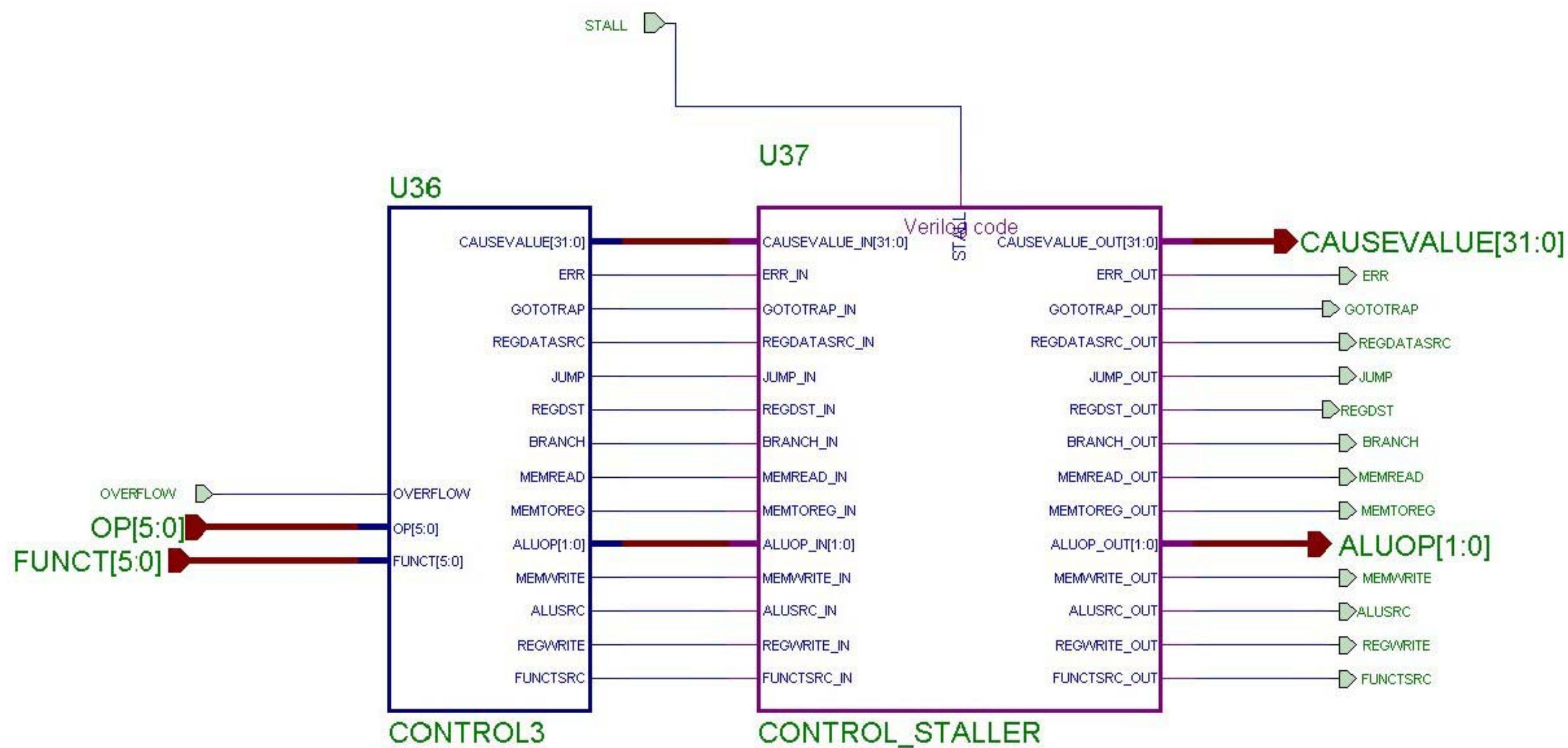
```
Schematic: full.sch
Description: Schematic for the pipelined processor.
```



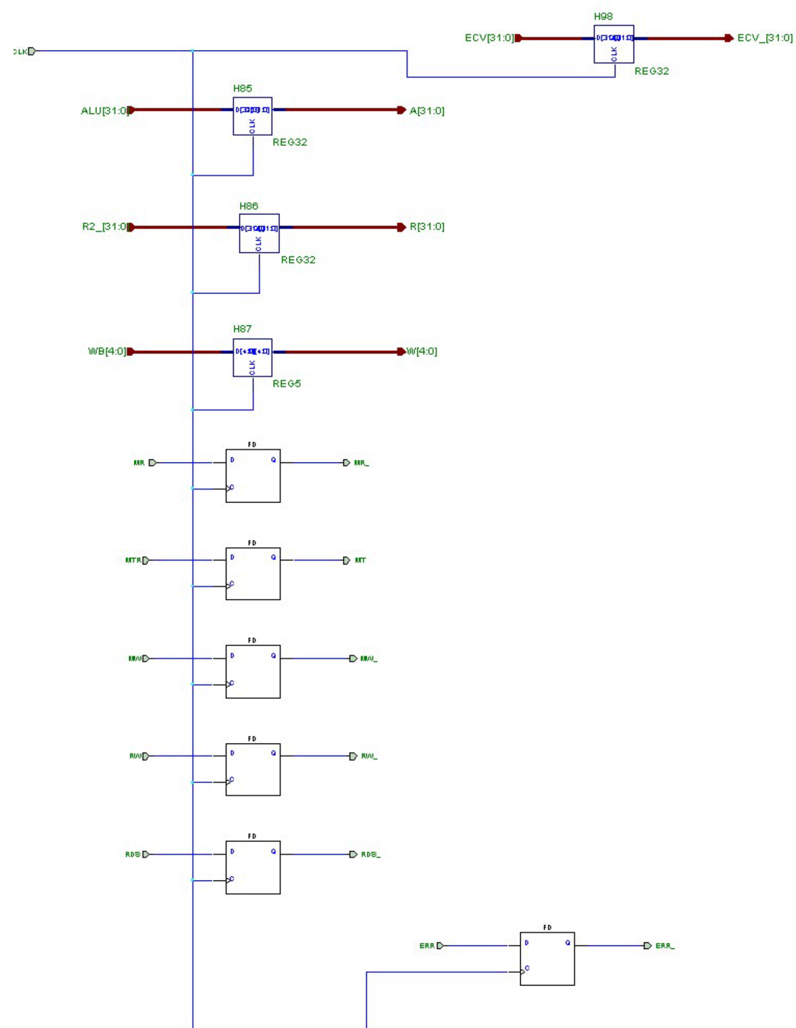
Schematic: BRANCH_PREDICTER.sch
Description: Predicts whether a branch is going to be taken.



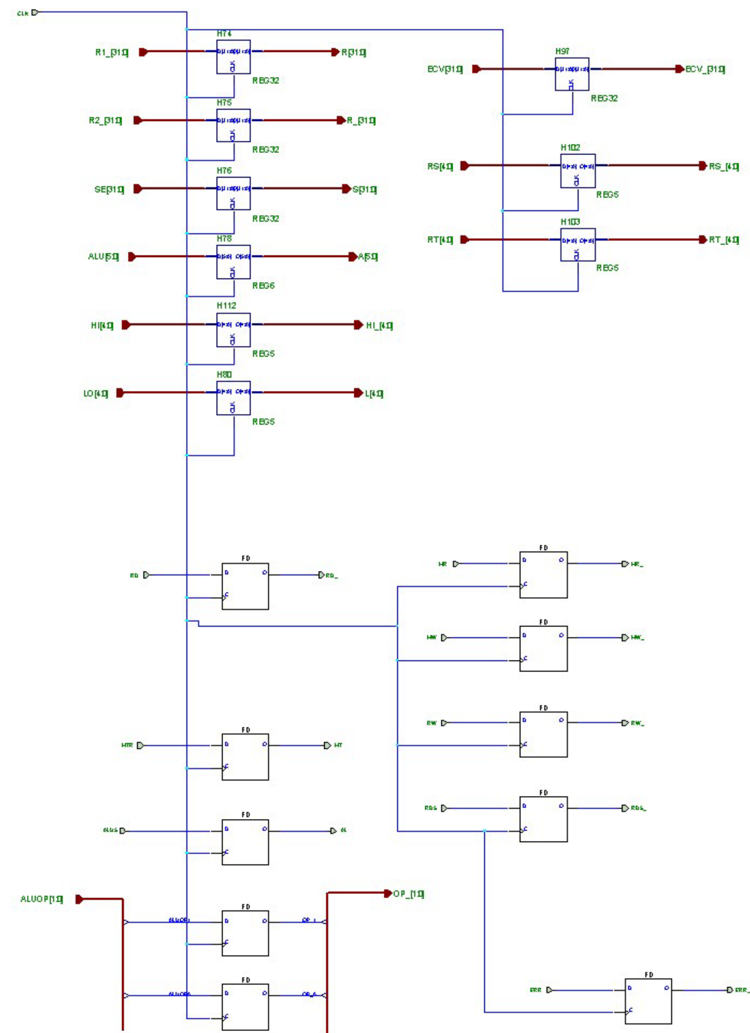
Schematic: CONTROL4.sch
Description: Control signals for the pipelined processor.



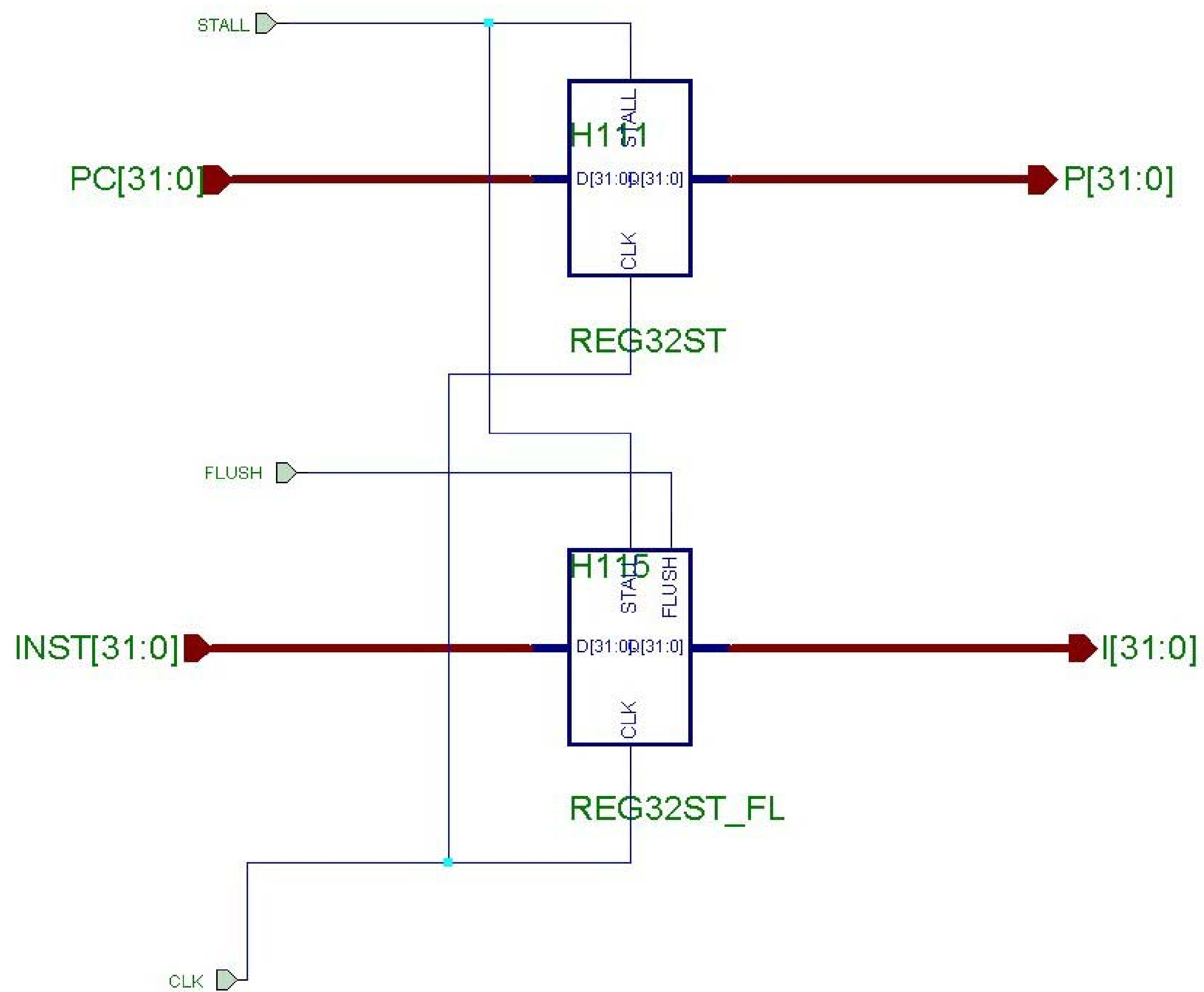
Schematic: EX_MEM.sch
Description: The Execution/Memory pipeline register.



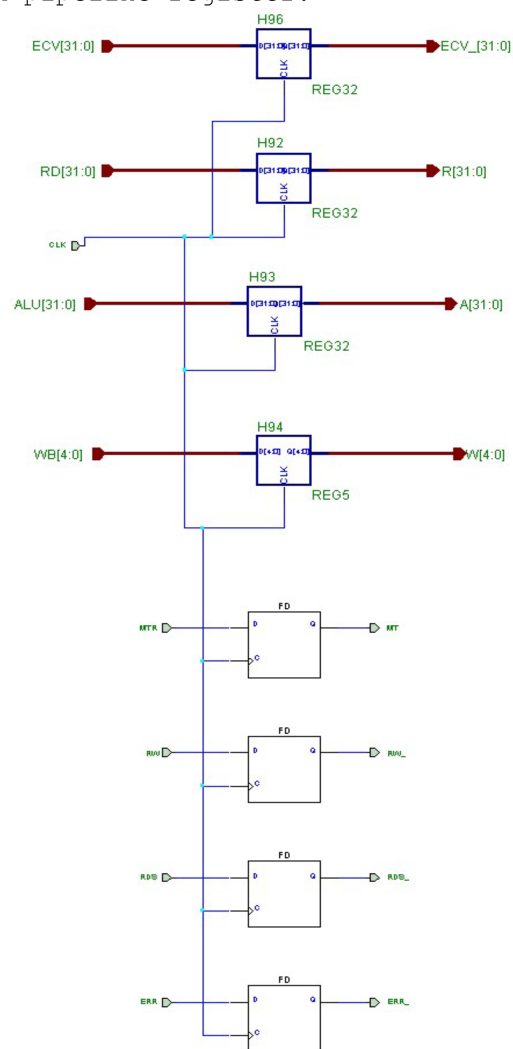
Schematic: ID_EX.sch
Description: The Instruction Decode/Execution pipeline register.



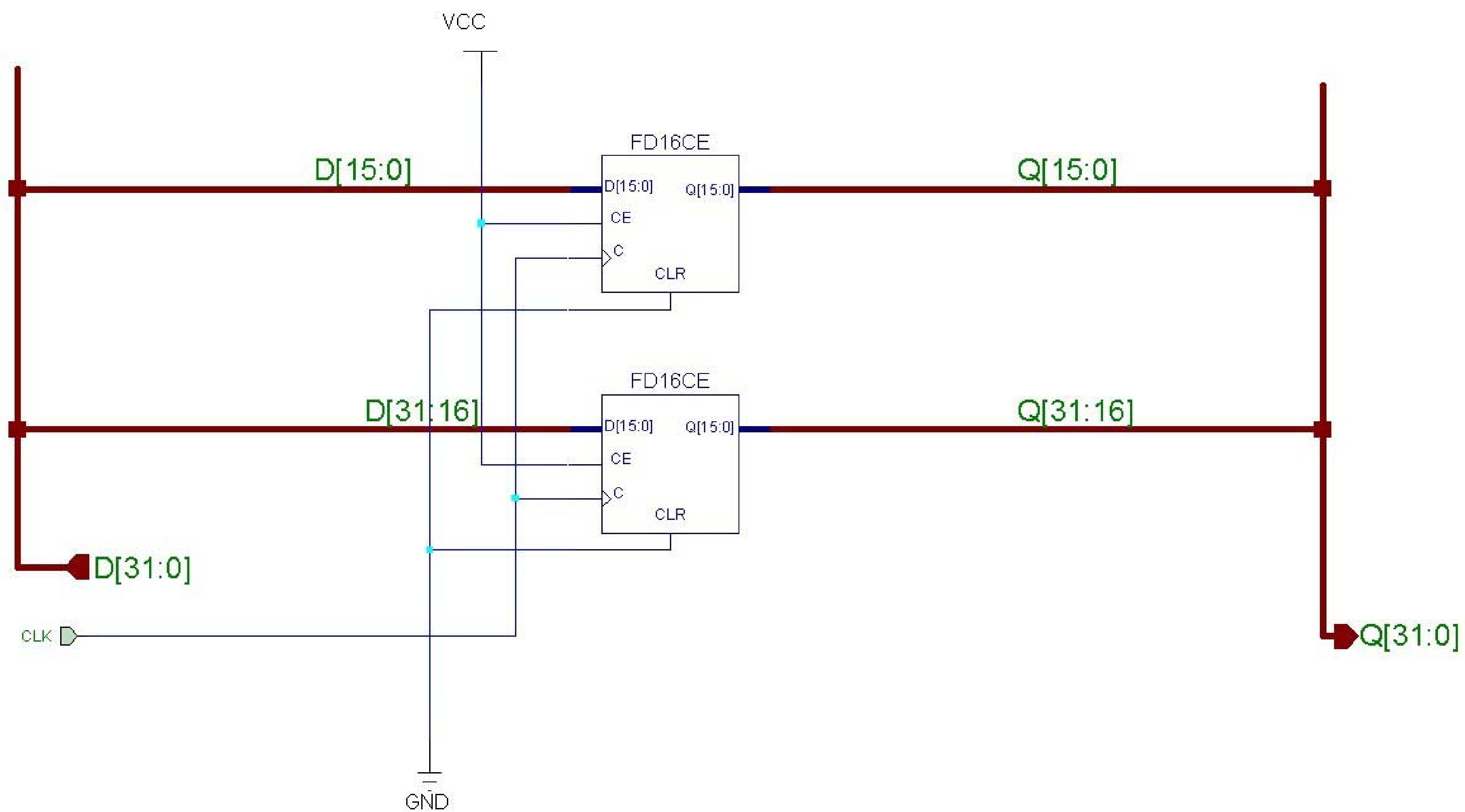
Schematic: IF_ID.sch
Description: The Instruction Fetch/Instruction Decode pipeline register.



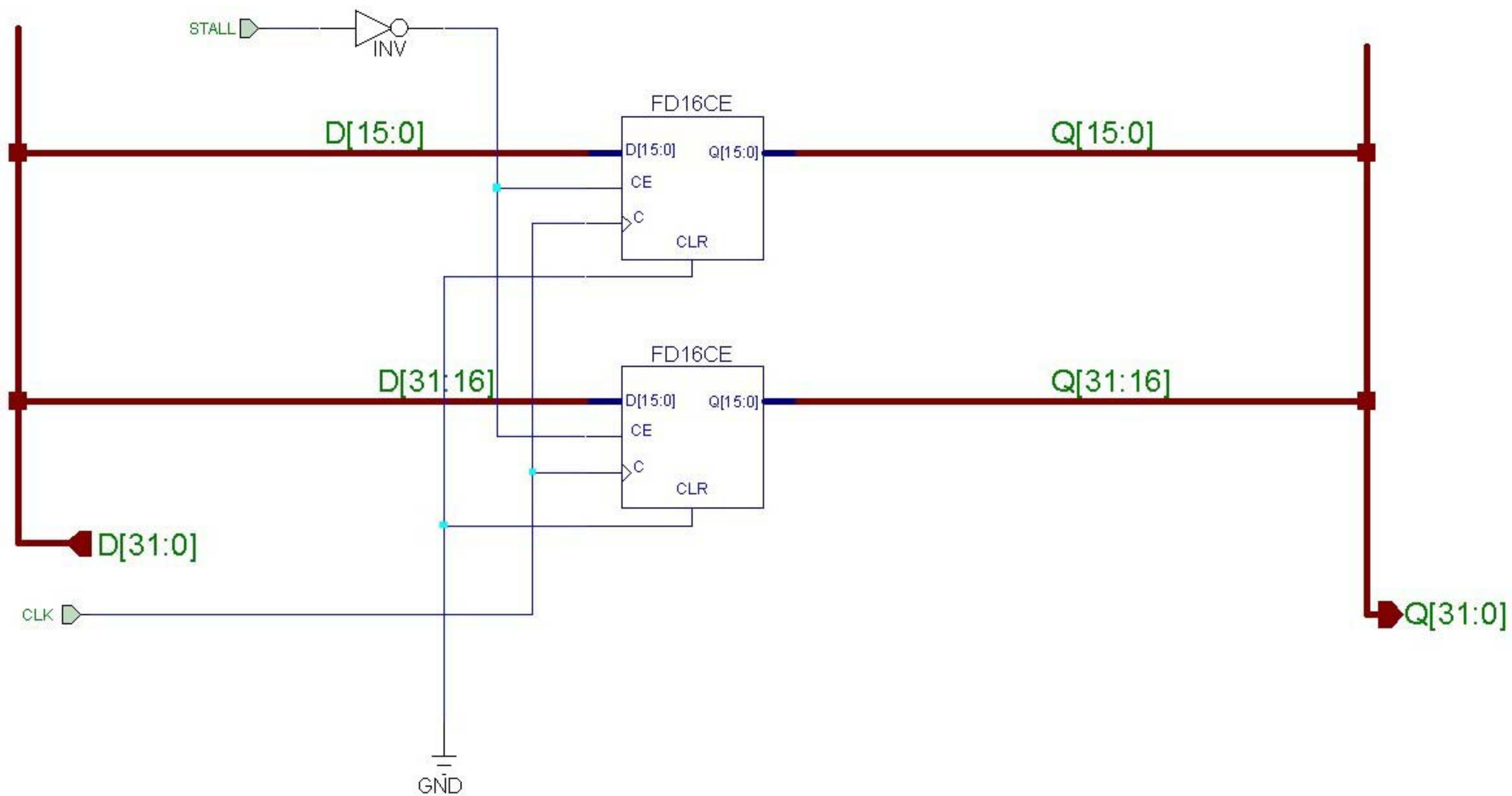
Schematic: MEM_WB.sch
Description: The Memory/Write Back pipeline register.



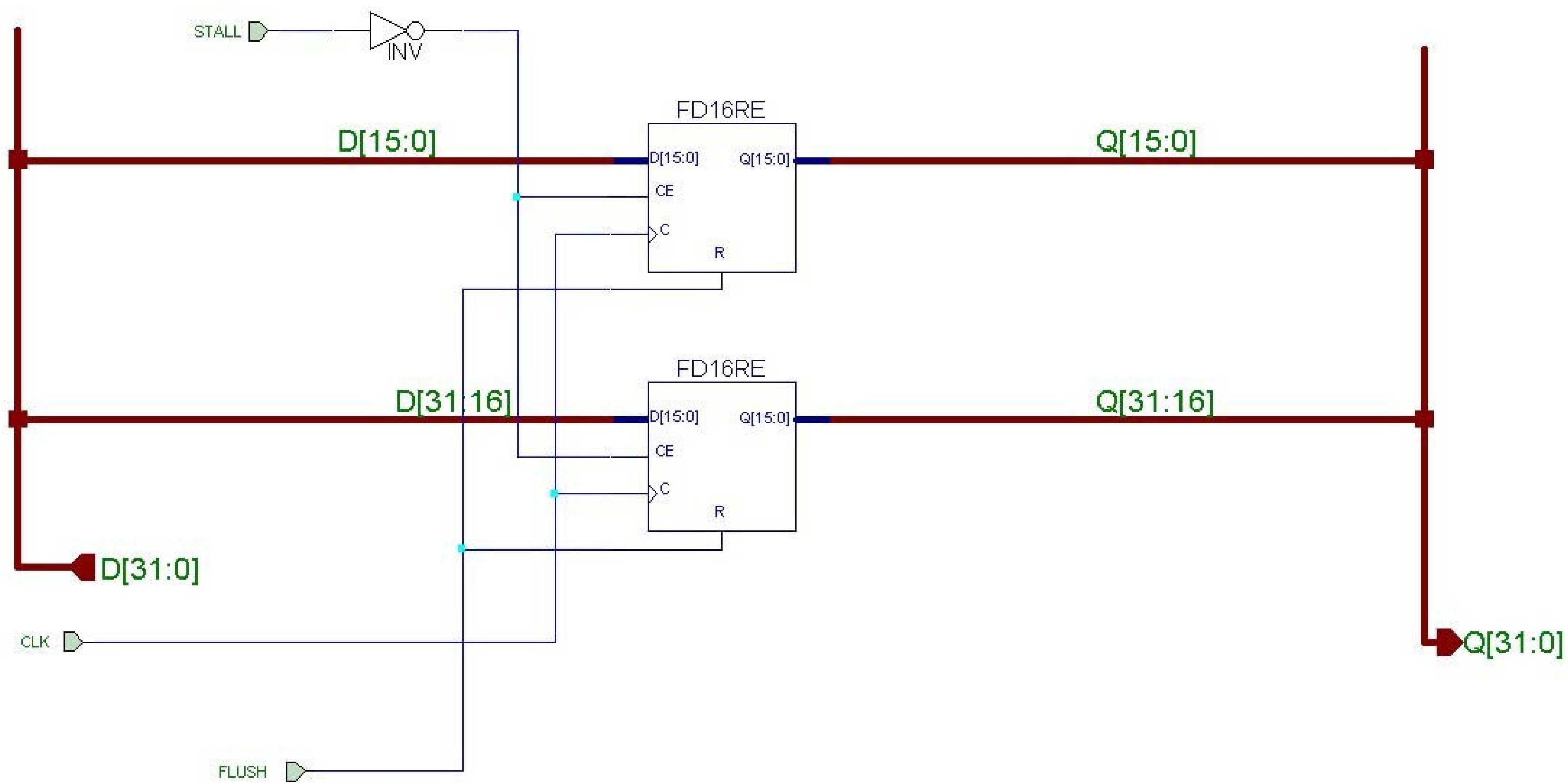
```
Schematic: REG32.sch
Description: A 32-bit register.
```



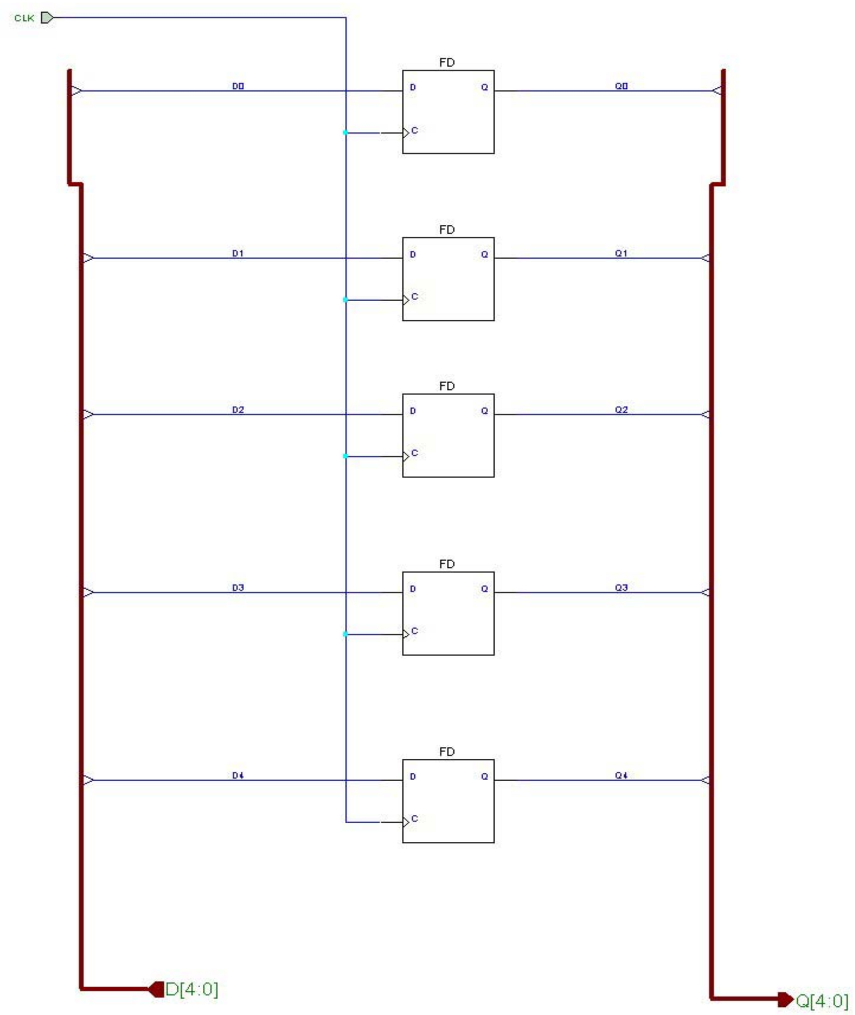
Schematic: REG32ST.sch
Description: A 32-bit register that can be stalled.



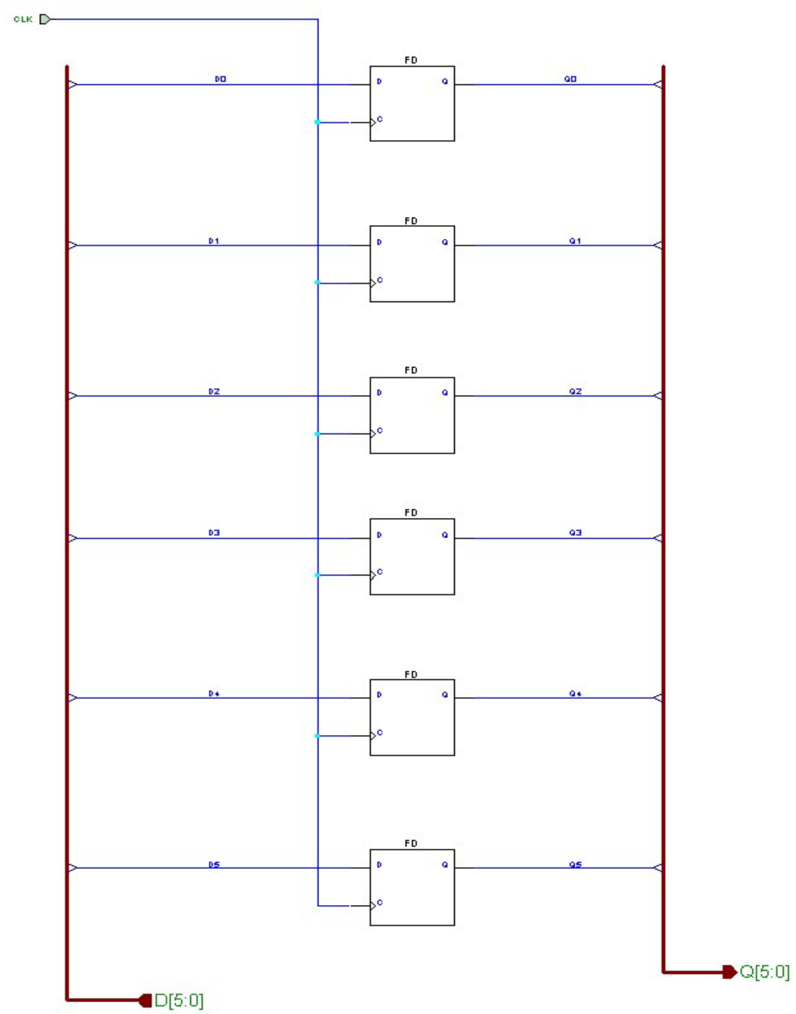
Schematic: REG32ST_FL.sch
Description: A 32-bit register that can be stalled or flushed.



Schematic: REG5.sch
Description: A 5-bit register.



Schematic: REG6.sch
Description: A 6-bit register.



Appendix E – Debugging Interface Code

```
'*********************************************************************
'* Tester.vbp
'* Author: Mark Holland
'* Last Revised: May 3 2002
'*
'* Purpose: This visual basic program runs the PC side of the debugging
'* interface for the FPGA-implemented processor that Mark Holland
'* developed for his Master's thesis, to be used in EE471.
'*
'* The DLLs that are referenced at the start of the program are specific
'* to Windows95, and at the current time that is the only platform that
'* this tool will run on.
'*
'* The communications between the PC and the FPGA are all initiated from
'* this program, and therefore from the PC.  Responses are returned from
'* the FPGA to this program and displayed on the debugging interface
'* in a user friendly fashion.
'*
'* All of the methods/functions have descriptions and whatever is not
'* commented about should be somewhat self explanatory.
'*********************************************************************




Private Declare Sub vbOut Lib "WIN95IO.DLL" (ByVal nPort As Integer, ByVal nData As Integer)
Private Declare Sub vbOutw Lib "WIN95IO.DLL" (ByVal nPort As Integer, ByVal nData As Integer)
Private Declare Function vbInp Lib "WIN95IO.DLL" (ByVal nPort As Integer) As Integer
Private Declare Function vbInpw Lib "WIN95IO.DLL" (ByVal nPort As Integer) As Integer
Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)
Private Declare Function GetTickCount Lib "kernel32" () As Long


'*******************************************************************
'************************** Used Everywhere ************************
'*******************************************************************
Dim Bit(0 To 69) As Byte
Dim Rbit(0 To 32) As Byte
Dim BitsToSend(0 To 9) As Byte
Dim Clock_Running As Boolean


Dim Temp As Double
'Dim Index As Integer
'Dim Bits As Integer
'Dim Val As Double
Dim TempNum As Double

'*************************
'*** Used in Get_Bit() ***
'*************************
Dim Port_num_gb As Byte
Dim Bit_pos_gb As Byte
Dim Num_bits_gb As Byte
Dim Port_gb As Byte
Dim Bit_gb(0 To 7) As Byte

'*************************
'***** Used in Send() ****
'*************************
Dim PC_ack_s As Byte
Dim Send_value_s As Integer
Dim V_ack_s As Byte
Dim New_ack_s As Byte
Dim Bit_Index As Byte

'*************************
```

```
'***** Used in Recv() ****
'*************************
Dim Port1_v As Byte
Dim PC_ack_v As Byte
Dim Port2_v As Byte
Dim V_ack_v As Byte
Dim New_ack_v As Byte

'*********************************************************************
'************************** Used for IO Test *************************
'*********************************************************************
Dim Errors As Double
Dim Iterations As Double
Dim Rand(0 To 9) As Byte
Dim Ret(0 To 10) As Byte
Dim Returned_Value As Byte

'*********************************************************************
'************************ Used for Register Op ***********************
'*********************************************************************
Dim Returned(0 To 10) As Byte
Dim Received(0 To 10) As Byte
Dim Reg_ As Byte
Dim ReturnedBit(0 To 32) As Byte
'Dim AddressBinary As Boolean
'Dim ValueBinary As Boolean

Dim HexNum As Byte
Dim HexBit(0 To 7) As String

Dim Time(0 To 20) As Long

Dim misc1_num As Double

Dim TempDecimal As Long
Dim PCAddress As Long


'*************************************************************************
'*************************************************************************
'****************************                ****************************
'****************************   CODE START   ****************************
'****************************                ****************************
'*************************************************************************
'*************************************************************************

'*************************************************************************
'************** ClockStart_Click()
'**************
'************** Starts the clock, which will stop when ClockStop is hit.
'**************
'************** Uses:   Standard_Send
'*************************************************************************

Private Sub ClockStart_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0

Next

Bit(58) = 1

Clock_Running = True

Standard_Send

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If
```

```
End Sub

'**********************************************************************
'************** ClockStop_Click()
'**************
'************** Stops the clock.
'**************
'************** Uses:   Standard_Send, vbInp, Get_bit
'**********************************************************************


Private Sub ClockStop_Click()

If Clock_Running = True Then

Clock_Running = False

Port1_v = vbInp(888)

PC_ack_v = Get_bit(888, 7)

V_ack_v = Get_bit(889, 6)

If PC_ack_v = 1 Then

    Port1_v = Port1_v - 128

Else

    Port1_v = Port1_v + 128

End If

vbOut 888, Port1_v

New_ack_v = Get_bit(889, 6)

Do While V_ack_v = New_ack_v

    New_ack_v = Get_bit(889, 6)

Loop

Standard_Return

ReadPC_Click

ReadInst_Click

ReadData_Click
ReadReg1_Click
ReadReg2_Click
ReadALU_Click
ReadMISC1_Click
ReadMISC2_Click
GetRegFile_Click
GetInstMem_Click


Else

MsgBox ("You must start the CPU clock before you may do this!")

End If

End Sub

'**********************************************************************
'************** ClockRun_Click()
'**************
'************** Runs the clock for the specified number of cycles.
'**************
'************** Uses:   DecimalToBits, Standard_Send, Standard_Return
'**********************************************************************
```

```
Private Sub ClockRun_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0

Next

Temp = ClockCycles.Text

DecimalToBits 0, 6, Temp

Standard_Send
'*ADDED
Standard_Return

'PCValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

ReadPC_Click

ReadInst_Click

ReadData_Click
ReadReg1_Click
ReadReg2_Click
ReadALU_Click
ReadMISC1_Click
ReadMISC2_Click
GetRegFile_Click
GetInstMem_Click


Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub


'**************************************************************************
'************** PCWrite_Click()
'**************
'************** Writes the specified value to the PC.
'**************
'************** Uses:   Standard_Send, Standard_Return
'**************************************************************************

Private Sub PCWrite_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0

Next

For j = 0 To 18

    Bit(32 + j) = PCVal(j).Caption

Next

Bit(61) = 1

Standard_Send
```

```
'added
Standard_Return
'added
PCValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24) &
Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)


Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub



'**************************************************************************
'************** RegWrite_Click()
'**************
'************** Writes the specified value to the specified register.
'**************
'************** Uses:   Standard_Send, Standard_Return
'**************************************************************************

Private Sub RegWrite_Click()

If Clock_Running = False Then

For i = 37 To 69

    Bit(i) = 0

Next

For j = 0 To 31

    Bit(j) = RegVal(j).Caption

Next

For k = 0 To 4

    Bit(32 + k) = RegAdd(k).Caption

Next

Bit(58) = 1
Bit(61) = 1

Standard_Send
'added
Standard_Return
'added
RegValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)


Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub

'**************************************************************************
'************** MemWrite_Click()
'**************
'************** Writes the specified value to memory.
'**************
```

```
'************** Uses:   Standard_Send, Standard_Return
'***********************************************************************

Private Sub MemWrite_Click()

If Clock_Running = False Then

For i = 51 To 69

    Bit(i) = 0

Next

For j = 0 To 31

    Bit(j) = MemVal(j).Caption

Next

For k = 0 To 18

    Bit(32 + k) = MemAdd(k).Caption

Next

Bit(59) = 1
Bit(61) = 1

Standard_Send
'added
Standard_Return
'added
MemValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If


End Sub

'***********************************************************************
'************** RegRead_Click()
'**************
'************** Reads the specified register.
'**************
'************** Uses:   Standard_Send, Standard_Return
'***********************************************************************

Private Sub RegRead_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0

Next

For j = 0 To 4

    Bit(32 + j) = RegAdd(j).Caption

Next

Bit(58) = 1
Bit(59) = 1
Bit(61) = 1

Standard_Send
```

```
Standard_Return

RegValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub

'*************************************************************************
'************** MemRead_Click()
'**************
'************** Reads the specified memory address.
'**************
'************** Uses:   Standard_Send, Standard_Return
'*************************************************************************

Private Sub MemRead_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0

Next

For j = 0 To 18

    Bit(32 + j) = MemAdd(j).Caption

Next

Bit(60) = 1
Bit(61) = 1

Standard_Send

Standard_Return

MemValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub

'*************************************************************************
'************** ReadALU_Click()
'**************
'************** Reads the ALU output.
'**************
'************** Uses:   Standard_Send, Standard_Return
'*************************************************************************

Private Sub ReadALU_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0
```

```
Next

Bit(58) = 1
Bit(59) = 1
Bit(60) = 1

Standard_Send

Standard_Return

ALUValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub

'*************************************************************************
'************** ReadData_Click()
'**************
'************** Reads the data line.
'**************
'************** Uses:   Standard_Send, Standard_Return
'*************************************************************************

Private Sub ReadData_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0

Next

Bit(60) = 1

Standard_Send

Standard_Return

DataValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub

'*************************************************************************
'************** ReadInst_Click()
'**************
'************** Reads the instruction line.
'**************
'************** Uses:   Standard_Send, Standard_Return
'*************************************************************************

Private Sub ReadInst_Click()

If Clock_Running = False Then

For i = 0 To 69
```

```
    Bit(i) = 0

Next

Bit(58) = 1
Bit(59) = 1

Standard_Send

Standard_Return

InstValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub

'*************************************************************************
'************** ReadPC_Click()
'**************
'************** Reads the program counter (PC).
'**************
'************** Uses:   Standard_Send, Standard_Return
'*************************************************************************

Private Sub ReadPC_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0

Next

Bit(59) = 1

Standard_Send

Standard_Return

PCValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24) &
Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub

'*************************************************************************
'************** ReadReg1_Click()
'**************
'************** Reads Register 1 output.
'**************
'************** Uses:   Standard_Send, Standard_Return
'*************************************************************************

Private Sub ReadReg1_Click()

If Clock_Running = False Then

For i = 0 To 69
```

```
    Bit(i) = 0

Next

Bit(58) = 1
Bit(60) = 1

Standard_Send

Standard_Return

Reg1Value.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub

'**************************************************************************
'************** ReadReg2_Click()
'**************
'************** Reads Register 2 output.
'**************
'************** Uses:   Standard_Send, Standard_Return
'**************************************************************************

Private Sub ReadReg2_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0

Next

Bit(59) = 1
Bit(60) = 1

Standard_Send

Standard_Return

Reg2Value.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub


'**************************************************************************
'************** ReadMISC1_Click()
'**************
'************** Reads MISC1 output.
'**************
'************** Uses:   Standard_Send, Standard_Return
'**************************************************************************

Private Sub ReadMISC1_Click()

If Clock_Running = False Then
```

```
For i = 0 To 69

    Bit(i) = 0

Next

Bit(58) = 1
Bit(60) = 1
Bit(61) = 1

Standard_Send

Standard_Return

MISC1Value.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) &
Rbit(24) & Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) &
Rbit(15) & Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) &
Rbit(6) & Rbit(5) & Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub


'**************************************************************************
'************** ReadMISC2_Click()
'**************
'************** Reads MISC2 output.
'**************
'************** Uses:   Standard_Send, Standard_Return
'**************************************************************************

Private Sub ReadMISC2_Click()

If Clock_Running = False Then

For i = 0 To 69

    Bit(i) = 0

Next

Bit(59) = 1
Bit(60) = 1
Bit(61) = 1

Standard_Send

Standard_Return

MISC2Value.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) &
Rbit(24) & Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) &
Rbit(15) & Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) &
Rbit(6) & Rbit(5) & Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Else

MsgBox ("You must stop the CPU clock before you may do this!")

End If

End Sub


'**************************************************************************
'************** PCVal_Click()
'**************
'************** Sets the specified PC value bit.
'**************
'************** Uses:
'**************************************************************************
```

```
Private Sub PCVal_Click(Index As Integer)

If PCVal(Index).Caption = 1 Then

    PCVal(Index).Caption = 0

Else

    PCVal(Index).Caption = 1

End If


End Sub

'**********************************************************************
'************** RegVal_Click()
'**************
'************** Sets the specified register value bit.
'**************
'************** Uses:
'**********************************************************************

Private Sub RegVal_Click(Index As Integer)

If RegVal(Index).Caption = 1 Then

    RegVal(Index).Caption = 0

Else

    RegVal(Index).Caption = 1

End If


End Sub

'**********************************************************************
'************** RegAdd_Click()
'**************
'************** Sets the specified register address bit.
'**************
'************** Uses:
'**********************************************************************

Private Sub RegAdd_Click(Index As Integer)

If RegAdd(Index).Caption = 1 Then

    RegAdd(Index).Caption = 0

Else

    RegAdd(Index).Caption = 1

End If


End Sub

'**********************************************************************
'************** MemVal_Click()
'**************
'************** Sets the specified memory value bit.
'**************
'************** Uses:
'**********************************************************************

Private Sub MemVal_Click(Index As Integer)

If MemVal(Index).Caption = 1 Then

    MemVal(Index).Caption = 0
```

```
Else

    MemVal(Index).Caption = 1

End If


End Sub

'*************************************************************************
'************** MemAdd_Click()
'**************
'************** Sets the specified memory address bit.
'**************
'************** Uses:
'*************************************************************************

Private Sub MemAdd_Click(Index As Integer)

If MemAdd(Index).Caption = 1 Then

    MemAdd(Index).Caption = 0

Else

    MemAdd(Index).Caption = 1

End If


End Sub

'*************************************************************************
'************** DecimalToBits(Index, Bits, Val)
'**************
'************** Takes the decimal value Val and makes Bits bits starting
'************** at index Index
'**************
'************** Uses:
'*************************************************************************

Sub DecimalToBits(Index, Bits, Val)

Temp = Val

For i = 0 To (Bits - 1)

    Bit(Index + i) = Temp Mod 2

    Temp = Fix(Temp / 2)

Next

End Sub

'*************************************************************************
'************** BitsToDecimal(Index, Bits)
'**************
'************** Takes the bits starting at index Index, Bits long, and
'************** puts their decimal value into TempDecimal
'**************
'************** Uses:
'*************************************************************************

Sub BitsToDecimal(Index, Bits)

TempDecimal = 0

For i = 0 To (Bits - 1)

    TempDecimal = TempDecimal + ((2 ^ i) * Bit(Index + i))

Next
```

```
End Sub

'**********************************************************************
'************** BinaryToBits(Index, Bits, Val)
'**************
'************** Takes the binary value Val and makes Bits bits starting
'************** at index Index
'**************
'************** Uses:
'**********************************************************************

Sub BinaryToBits(Index, Bits, Val)

Temp = Val

For i = 0 To (Bits - 1)

    Bit(Index + i) = Temp Mod 10

    Temp = Fix(Temp / 10)

Next

End Sub

'**********************************************************************
'************** FixBits()
'**************
'************** Fixes the bits so that they are sendable.  The top two
'************** bits of each send must be inverted.
'**************
'************** Uses:
'**********************************************************************

Sub FixBits()

For i = 0 To 9

    If Bit(7 * i) = 0 Then

        Bit(7 * i) = 1

    Else

        Bit(7 * i) = 0

    End If

    If Bit(7 * i + 1) = 0 Then

        Bit(7 * i + 1) = 1

    Else

        Bit(7 * i + 1) = 0

    End If

Next

End Sub

'**********************************************************************
'************** MakeBitsSendable()
'**************
'************** Puts each 7 bit send into a value to be sent.
'**************
'************** Uses:
'**********************************************************************

Sub MakeBitsSendable()

For i = 0 To 9
```

```
    BitsToSend(i) = Bit((7 * i)) + 2 * Bit((7 * i) + 1) + 4 * Bit((7 * i) + 2) + 8 * Bit((7 * i) + 3)
+ 16 * Bit((7 * i) + 4) + 32 * Bit((7 * i) + 5) + 64 * Bit((7 * i) + 6)

Next

End Sub

'************************************************************************
'************** Get_bit(Port_num_gb, Bit_pos_gb) As Byte
'**************
'************** Returns the bit from position Bit_pos_gb of port Port_num_gb.
'**************
'************** Uses:   vbInp
'************************************************************************

Function Get_bit(Port_num_gb, Bit_pos_gb) As Byte

'This function returns the specified bit from the specified port

'    Num_bits_gb = 8

Port_gb = vbInp(Port_num_gb)

For i = 0 To (7)

Time(1) = GetTickCount

    Bit_gb(i) = Port_gb Mod 2

Time(2) = GetTickCount
Time(10) = Time(10) + Time(2) - Time(1)

    Port_gb = Fix(Port_gb / 2)

Time(3) = GetTickCount
Time(11) = Time(11) + Time(3) - Time(2)
Time(12) = Time(12) + Time(3) - Time(1)

Next

Get_bit = Bit_gb(Bit_pos_gb)


End Function

'************************************************************************
'************** Standard_Send
'**************
'************** Sends the value to the FPGA.
'************** Sends 10 values + 2 extra PC_ACKs, waits for last V_ACK.
'************** In all: 11 PC_ACKS
'**************
'************** Uses:   vbInp, Get_bit, vbOut
'************************************************************************

Sub Standard_Send()

'MsgBox ("OP:" & Bit(63) & Bit(62) & Bit(61) & Bit(60) & Bit(59) & Bit(58))

FixBits

MakeBitsSendable

For i = 0 To 9

    PC_ack_s = Get_bit(888, 7)

    V_ack_s = Get_bit(889, 6)

    If PC_ack_s = 1 Then

        vbOut 888, BitsToSend(i)

    Else
```

```
        BitsToSend(i) = BitsToSend(i) + 128

        vbOut 888, BitsToSend(i)

    End If

    New_ack_s = Get_bit(889, 6)

    Do While V_ack_s = New_ack_s

        New_ack_s = Get_bit(889, 6)

    Loop

Next

For j = 0 To 1

    V_ack_s = Get_bit(889, 6)

    If BitsToSend(9) < 128 Then

        BitsToSend(9) = BitsToSend(9) + 128

    Else

        BitsToSend(9) = BitsToSend(9) - 128

    End If

    vbOut 888, BitsToSend(9)

    New_ack_s = Get_bit(889, 6)

    Do While V_ack_s = New_ack_s

        New_ack_s = Get_bit(889, 6)

    Loop

Next

End Sub

'**************************************************************************
'************** Standard_Return()
'**************
'************** Gets a return value from the FPGA.  Does 10 PC_ACKS and
'************** waits for last V_ACK.
'**************
'************** Uses:   vbInp, Get_bit, vbOut
'**************************************************************************

Sub Standard_Return()

For i = 0 To 9

    Port2_v = vbInp(889)

    'MsgBox ("Port 2: " & Port2_v)

    V_ack_v = Get_bit(889, 6)

    If V_ack_v = 1 Then

        Port2_v = Port2_v - 199

    Else

        Port2_v = Port2_v - 135

    End If

    Received(i) = Port2_v / 8
```

```
    For j = 0 To 2
        Rbit(3 * i + j) = Received(i) Mod 2
        Received(i) = Fix(Received(i) / 2)
    Next

    Port1_v = vbInp(888)
    PC_ack_v = Get_bit(888, 7)
    V_ack_v = Get_bit(889, 6)
    If PC_ack_v = 1 Then
        Port1_v = Port1_v - 128
    Else
        Port1_v = Port1_v + 128
    End If
    vbOut 888, Port1_v
    New_ack_v = Get_bit(889, 6)
    Do While V_ack_v = New_ack_v
        New_ack_v = Get_bit(889, 6)
    Loop
Next
    i = 10
    Port2_v = vbInp(889)
    'MsgBox ("Port 2: " & Port2_v)
    V_ack_v = Get_bit(889, 6)
    If V_ack_v = 1 Then
        Port2_v = Port2_v - 199
    Else
        Port2_v = Port2_v - 135
    End If
    Received(i) = Port2_v / 8
    For j = 0 To 2
        Rbit(3 * i + j) = Received(i) Mod 2
        Received(i) = Fix(Received(i) / 2)
    Next

End Sub
'**************************************************************************
'************** Run_Click()
'**************
'************** Runs the memory checker ... UNDER CONSTRUCTION!
'**************
'************** Uses:
'**************************************************************************
```

```
Private Sub Run_Click()

Dim nFileNum As Integer

' Get a free file number
nFileNum = FreeFile

' Create Test.txt
Open App.Path & "\super.txt" For Output As nFileNum

' Write the contents of TextBox1 to Test.txt
Write #nFileNum, "ADDRESS" & "                        " & "DATA"

For j = 0 To 1

    For i = 51 To 69

        Bit(i) = 0

    Next

    Bit(59) = 1
    Bit(61) = 1


    Make_Random
'MsgBox ("Sending")

    Write #nFileNum, Bit(50) & Bit(49) & Bit(48) & Bit(47) & Bit(46) & Bit(45) & Bit(44) & Bit(43) &
Bit(42) & Bit(41) & Bit(40) & Bit(39) & Bit(38) & Bit(37) & Bit(36) & Bit(35) & Bit(34) & Bit(33) &
Bit(32) & "       " & Bit(31) & Bit(30) & Bit(29) & Bit(28) & Bit(27) & Bit(26) & Bit(25) & Bit(24) &
Bit(23) & Bit(22) & Bit(21) & Bit(20) & Bit(19) & Bit(18) & Bit(17) & Bit(16) & Bit(15) & Bit(14) &
Bit(13) & Bit(12) & Bit(11) & Bit(10) & Bit(9) & Bit(8) & Bit(7) & Bit(6) & Bit(5) & Bit(4) & Bit(3) &
Bit(2) & Bit(1) & Bit(0)

    Standard_Send
'MsgBox ("Receiving")
    Standard_Return
'MsgBox ("Done Receiving")


MemValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) & Rbit(24)
& Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) & Rbit(15) &
Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) & Rbit(6) & Rbit(5)
& Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)



    ' Close the file

Next

    Close nFileNum


End Sub

'**************************************************************************
'************** Make_Random()
'**************
'************** Makes 51 random bits.
'**************
'************** Uses:
'**************************************************************************


Function Make_Random()


For i = 0 To 50

    Randomize
```

```
    Bit(i) = Int(2 * Rnd)

Next

End Function

'************************************************************************
'************** Test_Send
'**************
'************** Sends the value to the FPGA.
'************** Sends 10 values and waits for last V_ACK.
'************** In all: 10 PC_ACKS
'**************
'************** Uses:   vbInp, Get_bit, vbOut
'************************************************************************

Sub Test_Send()

'MsgBox ("OP:" & Bit(63) & Bit(62) & Bit(61) & Bit(60) & Bit(59) & Bit(58))

FixBits

MakeBitsSendable

For i = 0 To 9

    PC_ack_s = Get_bit(888, 7)

    V_ack_s = Get_bit(889, 6)

    If PC_ack_s = 1 Then

        vbOut 888, BitsToSend(i)

    Else

        BitsToSend(i) = BitsToSend(i) + 128

        vbOut 888, BitsToSend(i)

    End If

    New_ack_s = Get_bit(889, 6)

'MsgBox ("up to loop")

    Do While V_ack_s = New_ack_s

'MsgBox ("into loop")

        New_ack_s = Get_bit(889, 6)

    Loop

Next
'
'V_ack_s = Get_bit(889, 6)
'
'If BitsToSend(9) < 128 Then
'
'    BitsToSend(9) = BitsToSend(9) + 128
'
'Else
'
'    BitsToSend(9) = BitsToSend(9) - 128
'
'End If
'
'vbOut 888, BitsToSend(9)
'
'New_ack_s = Get_bit(889, 6)
'
'Do While V_ack_s = New_ack_s
'
```

```
'    New_ack_s = Get_bit(889, 6)
'
'Loop

End Sub

'**************************************************************************
'************** Test_Return()
'**************
'************** Gets a return value from the FPGA.  Does 11 PC_ACKS and
'************** waits for last V_ACK.
'**************
'************** Uses:   vbInp, Get_bit, vbOut
'**************************************************************************

Sub Test_Return()

For i = 0 To 10

MsgBox ("Return: " & i)

    Port1_v = vbInp(888)

    PC_ack_v = Get_bit(888, 7)

    V_ack_v = Get_bit(889, 6)

    If PC_ack_v = 1 Then

        Port1_v = Port1_v - 128

    Else

        Port1_v = Port1_v + 128

    End If

    vbOut 888, Port1_v

    New_ack_v = Get_bit(889, 6)

    Do While V_ack_v = New_ack_v

        New_ack_v = Get_bit(889, 6)

    Loop

    Port2_v = vbInp(889)

    'MsgBox ("Port 2: " & Port2_v)

    V_ack_v = Get_bit(889, 6)

    If V_ack_v = 1 Then

        Port2_v = Port2_v - 199

    Else

        Port2_v = Port2_v - 135

    End If

    Received(i) = Port2_v / 8

'MsgBox ("Received: " & Received(i))

    For j = 0 To 2

        Rbit(3 * i + j) = Received(i) Mod 2

        Received(i) = Fix(Received(i) / 2)

    Next
```

```
Next

End Sub


'*************************************************************************
'************** GetRegFile_Click()
'**************
'************** Gets the complete status of the register file.
'**************
'************** Uses:   DecimalToBits, Standard_Send, Standard_Return, MakeHexBits
'*************************************************************************


Private Sub GetRegFile_Click()

For m = 0 To 31

    For i = 0 To 69

        Bit(i) = 0

    Next

Bit(58) = 1
Bit(59) = 1
Bit(61) = 1


    DecimalToBits 32, 5, m

    Standard_Send

    Standard_Return

    MakeHexBits

    RegFileValue(m).Text = HexBit(7) & HexBit(6) & HexBit(5) & HexBit(4) & HexBit(3) & HexBit(2) &
HexBit(1) & HexBit(0)

Next

End Sub

'*************************************************************************
'************** MakeHexBits()
'**************
'************** Turns the binary bits into hex Strings.
'**************
'************** Uses:
'*************************************************************************

Sub MakeHexBits()

For i = 0 To 7

    HexNum = Rbit(4 * i) + 2 * Rbit(4 * i + 1) + 4 * Rbit(4 * i + 2) + 8 * Rbit(4 * i + 3)

    HexBit(i) = Hex$(HexNum)

'MsgBox HexNum & " => " & HexBit(i)

Next

End Sub

'*************************************************************************
```

```
'************** ReadFile_Click()
'**************
'************** Reads the file specified in FileName.Text into memory.
'************** See Test.txt for an example of the file formatting.
'**************
'************** Uses:   DecimalToBits, Standard_Send, Standard_Return
'**************************************************************************


Private Sub ReadFile_Click()

Dim nFileNum As Integer, RR As String
Dim Count As Integer

' Get a free file number
nFileNum = FreeFile

' Open Test.txt for input. App.Path returns the path your app is saved in
Open App.Path & "\" & FileName.Text For Input As nFileNum

Count = 0

Do While Not EOF(nFileNum)

    ' Read the contents of the file
    For i = 0 To 31

        Bit(31 - i) = Input(1, nFileNum)

    Next

    Line Input #nFileNum, RR

    For j = 51 To 69

        Bit(j) = 0

    Next

    DecimalToBits 32, 19, Count

    Bit(59) = 1
    Bit(61) = 1

    'Line Input #nFileNum, RR

    '    MemValue.Text = Bit(31) & Bit(30) & Bit(29) & Bit(28) & Bit(27) & Bit(26)


    Standard_Send
    'added
    Standard_Return
    'added
    MemValue.Text = Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) & Rbit(25) &
Rbit(24) & Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) & Rbit(16) &
Rbit(15) & Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) & Rbit(7) &
Rbit(6) & Rbit(5) & Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)
    'If Count = 5 Then

    '    Done = True

    'End If


    Count = Count + 1

Loop
MsgBox "Done"

    ' Close the file
    Close nFileNum

End Sub
```

```
'***********************************************************************
'************** DumpMem_Click()
'**************
'************** Dumps the memory to Dump.txt in the VB directory.
'**************
'************** Uses:   DecimalToBits, Standard_Send, Standard_Return
'***********************************************************************


Private Sub DumpMem_Click()

Dim nFileNum As Integer

Time(0) = GetTickCount

' Get a free file number
nFileNum = FreeFile

' Create Test.txt
Open App.Path & "\Dump.txt" For Output As nFileNum

For j = 0 To 100

For i = 0 To 69

        Bit(i) = 0

    Next

    DecimalToBits 32, 19, j

    Bit(60) = 1
    Bit(61) = 1

    Standard_Send

    Standard_Return

    Write #nFileNum, j & ": " & Rbit(31) & Rbit(30) & Rbit(29) & Rbit(28) & Rbit(27) & Rbit(26) &
Rbit(25) & Rbit(24) & Rbit(23) & Rbit(22) & Rbit(21) & Rbit(20) & Rbit(19) & Rbit(18) & Rbit(17) &
Rbit(16) & Rbit(15) & Rbit(14) & Rbit(13) & Rbit(12) & Rbit(11) & Rbit(10) & Rbit(9) & Rbit(8) &
Rbit(7) & Rbit(6) & Rbit(5) & Rbit(4) & Rbit(3) & Rbit(2) & Rbit(1) & Rbit(0)

Next

'MsgBox "Done"
' Write the contents of TextBox1 to Test.txt
'Write #nFileNum, TextBox1.Text

' Close the file
Close nFileNum

Time(6) = GetTickCount
Time(15) = Time(6) - Time(0)

MsgBox "Time 0: " & Time(0) & vbCrLf & "Time 1: " & Time(1) & vbCrLf & "Time 2: " & Time(2) & vbCrLf &
"Time 3: " & Time(3) & vbCrLf & "Time 4: " & Time(4) & vbCrLf & "Time 5: " & Time(5) & vbCrLf & "Time
6: " & Time(6) & vbCrLf & "Time 7: " & Time(7) & vbCrLf & "Time 8: " & Time(8) & vbCrLf & "Time 9: " &
Time(9) & vbCrLf & "Time 10: " & Time(10) & vbCrLf & "Time 11: " & Time(11) & vbCrLf & "Time 12: " &
Time(12) & vbCrLf & "Time 13: " & Time(13) & vbCrLf & "Time 14: " & Time(14) & vbCrLf & "Time 15: " &
Time(15) & vbCrLf & "Time 16: " & Time(16) & vbCrLf & "Time 17: " & Time(17) & vbCrLf & "Time 18: " &
Time(18)

End Sub

'***********************************************************************
'************** WriteMISC1_Click()
'**************
'************** Writes the specified value to the MISC1 input bus.
'**************
'************** Uses:   DecimalToBits, Standard_Send, Standard_Return
'***********************************************************************
```

```
Private Sub WriteMISC1_Click()

For i = 0 To 69

    Bit(i) = 0

Next

misc1_num_high = Val("&H" + MISC1WriteValueHigh.Text)
misc1_num_low = Val("&H" + MISC1WriteValueLow.Text)

'MsgBox misc1_num_high
'MsgBox misc1_num_low

If (misc1_num_high < 0) Then
    Bit(31) = 1
    misc1_num_high = misc1_num_high + 32768
    DecimalToBits 16, 15, misc1_num_high

Else

    DecimalToBits 16, 16, misc1_num_high

End If

If (misc1_num_low < 0) Then
    Bit(15) = 1
    misc1_num_low = misc1_num_low + 32768
    DecimalToBits 0, 15, misc1_num_low

Else

    DecimalToBits 0, 16, misc1_num_low

End If

Bit(61) = 1
Bit(60) = 1
Bit(59) = 1
Bit(58) = 1

Standard_Send

Standard_Return

ReadMISC1_Click
ReadMISC2_Click

'MsgBox Bit(31) & Bit(30) & Bit(29) & Bit(28) & Bit(27) & Bit(26) & Bit(25) & Bit(24) & Bit(23) &
Bit(22) & Bit(21) & Bit(20) & Bit(19) & Bit(18) & Bit(17) & Bit(16) & Bit(15) & Bit(14) & Bit(13) &
Bit(12) & Bit(11) & Bit(10) & Bit(9) & Bit(8) & Bit(7) & Bit(6) & Bit(5) & Bit(4) & Bit(3) & Bit(2) &
Bit(1) & Bit(0)

End Sub

'***********************************************************************
'************** WriteMISC2_Click()
'**************
'************** Writes the specified value to the MISC2 input bus.
'**************
'************** Uses:   DecimalToBits, Standard_Send, Standard_Return
'***********************************************************************


Private Sub WriteMISC2_Click()

For i = 0 To 69

    Bit(i) = 0

Next

misc2_num_high = Val("&H" + MISC2WriteValueHigh.Text)
```

```
misc2_num_low = Val("&H" + MISC2WriteValueLow.Text)

'MsgBox misc2_num_high
'MsgBox misc2_num_low

If (misc2_num_high < 0) Then
    Bit(31) = 1
    misc2_num_high = misc2_num_high + 32768
    DecimalToBits 16, 15, misc2_num_high

Else

    DecimalToBits 16, 16, misc2_num_high

End If

If (misc2_num_low < 0) Then
    Bit(15) = 1
    misc2_num_low = misc2_num_low + 32768
    DecimalToBits 0, 15, misc2_num_low

Else

    DecimalToBits 0, 16, misc2_num_low

End If

Bit(62) = 1

Standard_Send

Standard_Return

ReadMISC1_Click
ReadMISC2_Click

'MsgBox Bit(31) & Bit(30) & Bit(29) & Bit(28) & Bit(27) & Bit(26) & Bit(25) & Bit(24) & Bit(23) &
Bit(22) & Bit(21) & Bit(20) & Bit(19) & Bit(18) & Bit(17) & Bit(16) & Bit(15) & Bit(14) & Bit(13) &
Bit(12) & Bit(11) & Bit(10) & Bit(9) & Bit(8) & Bit(7) & Bit(6) & Bit(5) & Bit(4) & Bit(3) & Bit(2) &
Bit(1) & Bit(0)

End Sub

'**************************************************************************
'************** WriteMISC3_Click()
'**************
'************** Writes the specified value to the MISC3 input bus.
'**************
'************** Uses:   DecimalToBits, Standard_Send, Standard_Return
'**************************************************************************


Private Sub WriteMISC3_Click()

For i = 0 To 69

    Bit(i) = 0

Next

misc3_num_high = Val("&H" + MISC3WriteValueHigh.Text)
misc3_num_low = Val("&H" + MISC3WriteValueLow.Text)

'MsgBox misc3_num_high
'MsgBox misc3_num_low

If (misc3_num_high < 0) Then
    Bit(31) = 1
    misc3_num_high = misc3_num_high + 32768
    DecimalToBits 16, 15, misc3_num_high

Else

    DecimalToBits 16, 16, misc3_num_high
```

```
End If

If (misc3_num_low < 0) Then
    Bit(15) = 1
    misc3_num_low = misc3_num_low + 32768
    DecimalToBits 0, 15, misc3_num_low

Else

    DecimalToBits 0, 16, misc3_num_low

End If

Bit(62) = 1
Bit(58) = 1

Standard_Send

Standard_Return

ReadMISC1_Click
ReadMISC2_Click

'MsgBox Bit(31) & Bit(30) & Bit(29) & Bit(28) & Bit(27) & Bit(26) & Bit(25) & Bit(24) & Bit(23) &
Bit(22) & Bit(21) & Bit(20) & Bit(19) & Bit(18) & Bit(17) & Bit(16) & Bit(15) & Bit(14) & Bit(13) &
Bit(12) & Bit(11) & Bit(10) & Bit(9) & Bit(8) & Bit(7) & Bit(6) & Bit(5) & Bit(4) & Bit(3) & Bit(2) &
Bit(1) & Bit(0)

End Sub


'**************************************************************************
'************** MISCUpdate_Click()
'**************
'************** Writes the specified values to the MISC input busses.
'**************
'************** Uses:   MISC1, MISC2, MISC3
'**************************************************************************



Private Sub MISCUpdate_Click()

WriteMISC1_Click
WriteMISC2_Click
WriteMISC3_Click

End Sub

Private Sub GetInstMem_Click()

'   BitsToDecimal(Index, Bits)

    ReadPC_Click

    For i = 0 To 18

        Bit(i) = Rbit(i)

    Next

    BitsToDecimal 0, 19

    PCAddress = TempDecimal

    For m = 0 To 6

        For j = 0 To 69

            Bit(j) = 0

        Next

        Bit(60) = 1
```

```
        Bit(61) = 1

        DecimalToBits 32, 19, (PCAddress + m)

        Standard_Send

        Standard_Return

        MakeHexBits

        InstVal(m).Text = HexBit(7) & HexBit(6) & HexBit(5) & HexBit(4) & HexBit(3) & HexBit(2) &
HexBit(1) & HexBit(0)

        DecimalToBits 0, 32, (PCAddress + m)

        For i = 0 To 31

            Rbit(i) = Bit(i)

        Next

        MakeHexBits

        InstAdd(m).Text = HexBit(7) & HexBit(6) & HexBit(5) & HexBit(4) & HexBit(3) & HexBit(2) &
HexBit(1) & HexBit(0)

        Next

End Sub
```

Appendix F – Lex Assembler Code

```
%{

/***************************************************
 * File: comp.l
 * Author: Mark Holland
 * Last Modified: May 6, 2002
 *
 * This is the lex file for my assembler, written
 * in lex and yacc.  This file matches input
 * strings and hands tokens to the yacc file,
 * which will use the tokens to match specific
 * grammars.
 *
 * The assembler takes a file of all MIPs non-floating
 * point/coprocessor instructions and maps them to
 * the following set: LW, SW, JALR, BGEZ, NOR, SUBU,
 * SYSCALL, and BREAK.
 *
 * We go through a total of 9 assembler passes.
 * The first 4 passes perform instruction expansion,
 * the 5th and 6th passes perform label offset
 * computations, the 7th does data insertions for
 * LI instructions, the 8th drops the labels and
 * allows the data to become binary in form, and the
 * 9th pass maps to machine code.
 ***************************************************/


#include <stdlib.h>
#include "comp.tab.h"
#include "string.h"

#define LOOKUP -1
#define FALSE 0
#define TRUE 1
#define FIVE 5
#define SIXTEEN 16
#define TWENTY_SIX 26
#define THIRTY_TWO 32



int initialized = FALSE;
int load_imm = FALSE;
int negative = FALSE;
int num_length;                    //num_length is needed for knowing
                                   //which instructions have what
                                   //data field lengths
int line_number = 1;
int output_line_number = 0;        //keeps track of output line no.
int error_cause = 0;               //specifies error and error message
int fill_int = 0;         //fills depending on pos. or neg.
int compiler_pass = 1;             //which pass we're on
int num_instructions = 0; //keeps track of input line no.
int label_type = 0;
char *  make_binary(long num, int bits);
char *  reg_number(char *reg);
%}

%%

data     {

//The data memory section is created on the fifth
//pass, so the sixth pass is the first time
//it must be parsed.

        if(compiler_pass < 6) {
        error_cause = 2;
        return ERROR;
        }
        num_length = THIRTY_TWO;
```

```
        return DATA;
        }

nop     {
        return NOP;
        }

lb      {
        num_length = SIXTEEN;
        return LB;
        }

lbu     {
        num_length = SIXTEEN;
        return LBU;
        }

lh      {
        num_length = SIXTEEN;
        return LH;
        }

lhu     {
        num_length = SIXTEEN;
        return LHU;
        }

lw      {
        num_length = SIXTEEN;
        return LW;
        }

lwl     {
        num_length = SIXTEEN;
        return LWL;
        }

lwr     {
        num_length = SIXTEEN;
        return LWR;
        }

lui     {
        num_length = SIXTEEN;
        return LUI;
        }

li      {
        num_length = SIXTEEN;
        return LI;
        }

li32    {
        load_imm = TRUE;
        num_length = THIRTY_TWO;
        return LI;
        }

lij     {
        label_type = 1;
        return LIJ;
        }

sb      {
        num_length = SIXTEEN;
        return SB;
        }

sh      {
        num_length = SIXTEEN;
        return SH;
        }

sw      {
        num_length = SIXTEEN;
```

```
        return SW;
        }

swr     {
        num_length = SIXTEEN;
        return SWR;
        }

swl     {
        num_length = SIXTEEN;
        return SWL;
        }

j       {
        num_length = TWENTY_SIX;
        return J;
        }

jal     {
        num_length = TWENTY_SIX;
        return JAL;
        }

jr      {
        return JR;
        }

jalr    {
        return JALR;
        }

beq     {
        num_length = SIXTEEN;
        return BEQ;
        }

bne     {
        num_length = SIXTEEN;
        return BNE;
        }

blez    {
        num_length = SIXTEEN;
        return BLEZ;
        }

bgtz    {
        num_length = SIXTEEN;
        return BGTZ;
        }

bltz    {
        num_length = SIXTEEN;
        return BLTZ;
        }

bgez    {
        num_length = SIXTEEN;
        label_type = 0;
        return BGEZ;
        }

bltzal  {
        num_length = SIXTEEN;
        return BLTZAL;
        }

bgezal  {
        num_length = SIXTEEN;
        return BGEZAL;
        }

and     {
        return AND;
        }
```

```
andi     {
         num_length = SIXTEEN;
         return ANDI;
         }

andi32   {
         num_length = THIRTY_TWO;
         return ANDI;
         }

or       {
         return OR;
         }

ori      {
         num_length = SIXTEEN;
         return ORI;
         }

ori32    {
         num_length = THIRTY_TWO;
         return ORI;
         }

xor      {
         return XOR;
         }

xori     {
         num_length = SIXTEEN;
         return XORI;
         }

xori32   {
         num_length = THIRTY_TWO;
         return XORI;
         }

nor      {
         return NOR;
         }

addi     {
         num_length = SIXTEEN;
         return ADDI;
         }

addi32   {
         num_length = THIRTY_TWO;
         return ADDI;
         }

addiu    {
         num_length = SIXTEEN;
         return ADDIU;
         }

addiu32  {
         num_length = THIRTY_TWO;
         return ADDIU;
         }

add      {
         return ADD;
         }

addu     {
         return ADDU;
         }

sub      {
         return SUB;
         }
```

```
subu    {
        return SUBU;
        }

mult    {
        return MULT;
        }

multu   {
        return MULTU;
        }

div     {
        return DIV;
        }

divu    {
        return DIVU;
        }

slti    {
        num_length = SIXTEEN;
        return SLTI;
        }

slti32  {
        num_length = THIRTY_TWO;
        return SLTI;
        }

sltiu   {
        num_length = SIXTEEN;
        return SLTIU;
        }

sltiu32 {
        num_length = THIRTY_TWO;
        return SLTIU;
        }

slt     {
        return SLT;
        }

sltu    {
        return SLTU;
        }

sll     {
        num_length = FIVE;
        return SLL;
        }

srl     {
        num_length = FIVE;
        return SRL;
        }

sra     {
        num_length = FIVE;
        return SRA;
        }

sllv    {
        return SLLV;
        }

srlv    {
        return SRLV;
        }

srav    {
        return SRAV;
        }
```

```
syscall {
        return SYSCALL;
        }

break   {
        return BREAK;
        }


\$t[0-2]|\$s[0-4]|\$zero|\$gp|\$fp|\$sp|\$ra|\$v[0-1]|\$a[0-3]      {

//These are the user's alloted registers.

                 yylval.sval = strdup(yytext);

                 if(compiler_pass == 9) {
                         yytext = reg_number(yylval.sval);
                         yylval.sval = strdup(yytext);
                 }

                 return REG;
                 }

\$t[3-9]|\$s[5-7]|\$at|\$k[0-1]    {

//These registers are reserved for the compiler, so they can
//only be used by intermediate compiler files.

                 yylval.sval = strdup(yytext);
                 if(compiler_pass == 1) {
                         error_cause = 2;
                         return ERROR;
                 }

                if(compiler_pass == 9) {
                        yytext = reg_number(yylval.sval);
                         yylval.sval = strdup(yytext);
                }

                 return REG;
                 }

[A-Z][A-Z0-9_]*: {

//The code for a label destination.  On compiler pass 5 the
//label destinations are all recorded so that on pass 6 the
//offsets can be calculated and the labels can be removed.

                 yylval.sval = strdup(yytext);

                 if(compiler_pass == 5)
                         add_label(line_number, yylval.sval);

                if(compiler_pass == 8)
                        yylval.sval = "";

                 return LABEL_DEST;
                 }

[A-Z][A-Z0-9_]*  {

//The code for a label.  On compiler pass 6 the labels are
//matched with their destination so that their offsets can
//be calculated and the actual labels can be removed.

                 int label_line;
                 yylval.sval = strdup(yytext);

                 if(compiler_pass == 6) {
                         label_line = lookup_label(strcat(yylval.sval,":"));
                         if(label_line != LOOKUP) {

//This is for calculating the offset
//and replacing the labels with it.
```

```
                                if(label_type == 0)
                                        yytext = make_binary((label_line - line_number - 1), 18);
                                else {
                                        yytext = make_binary((4 * (label_line - 1)), 16);
                                        if(yytext[0] == '0')
                                                fill_int = 0;
                                        else
                                                fill_int = 1;
                               }
                                yylval.sval = strdup(yytext);
                        }
                        else {
                        error_cause = 3;
                        return ERROR;
                        }
                }

                return LABEL;
                }

[A-Z][A-Z0-9_#]*:       {

//The code for a label destination that was created by the
//compiler in an intermediate file as part of the expansion.
//Treated just as the previous label destinations were.

                if(compiler_pass == 1) {
                        error_cause = 2;
                        return ERROR;
                }
                else {
                yylval.sval = strdup(yytext);

                        if(compiler_pass == 5)
                        add_label(line_number, yylval.sval);

                        if(compiler_pass == 8)
                                yylval.sval = "";

                return LABEL_DEST;
                }
                }


[A-Z][A-Z0-9_#]* {

//The code for a label that was created by the compiler in
//an intermediate file, as part of the expansion.  Treated just
//as the previous labels were.

                int label_line;
                if(compiler_pass == 1) {
                        error_cause = 2;
                        return ERROR;
                }
                else {
                yylval.sval = strdup(yytext);

                        if(compiler_pass == 6) {
                        label_line = lookup_label(strcat(yylval.sval,":"));
                        if(label_line != LOOKUP) {
                                        if(label_type == 0)
                                        yytext = make_binary((label_line - line_number - 1), 18);
                                else
                                         yytext = make_binary((4 * (label_line - 1)), 18);
                                yytext[0] = '0';
                                        yytext[1] = 'b';
                                          yylval.sval = strdup(yytext);
                                }
                        else {
                                error_cause = 3;
                                return ERROR;
                                }
                }
```

```
                 return LABEL;
                }

                 }



\*TRAP\* {

//For the ADD instruction, if there is an overflow we must jump
//to the exception handler.  The *TRAP* label will branch to the
//end of the instruction memory section, where I have inserted a
//jump instruction that will jump to the exception handler.

                 if(compiler_pass == 1) {
                 error_cause = 2;
                 return ERROR;
                }
                 else {
                         yylval.sval = strdup(yytext);
                         return TRAP;
                 }
                 }


0b[01]{32}       {

//A 32 bit binary number.  I must handle it in a special manner
//because the largest data type I can use is only a 32 bit
//unsigned number, while I need a 32 bit signed number.  Therefore
//I make note of the sign of the number (i.e. the 32nd bit) and
//only use a 31 bit number, replacing the sign bit later.
//The 32 bit numbers are mostly for immediate fields, and on
//compiler pass 7 are turned into data entries.

                 long num;
                 negative = FALSE;
                 yytext[0] = '0';
                 yytext[1] = '0';
                 if(yytext[2] == '1') {
                         yytext[2] = '0';
                         negative = TRUE;
                 }
                 else {
                         negative = FALSE;
                 }

                 num = strtol(yytext, (char **)NULL, 2);

                 if((compiler_pass == 7) && (load_imm == TRUE)) {
                         load_imm = FALSE;
                         add_data(num, negative);
                 }

                 yytext = make_binary(num, 32);
                 if(negative == TRUE) {
                         yytext[0] = '1';
                 }
                 yylval.sval = strdup(yytext);
                 return VALUE_32;
                 }

0b[01]{26}       {

//A 26 bit binary number, used for jumps.

                 long num;
                 yytext[0] = '0';
                 yytext[1] = '0';
                 num = strtol(yytext, (char **)NULL, 2);
                yytext = make_binary(num, 26);
                 yylval.sval = strdup(yytext);
                 return VALUE_26;
                 }
```

```
0b[01]{16}       {

//A 16 bit binary number, used in many branch and imediate
//instructions.  The fill int will allow the value to
//be sign extended if/when I turn it into a 32 bit value.

                 long num;
                 yytext[0] = '0';
                 yytext[1] = '0';
                 num = strtol(yytext, (char **)NULL, 2);
                 if(num >= 32768) {
                         fill_int = 1;
                 }
                 else {
                         fill_int = 0;
                 }
                yytext = make_binary(num, 16);
                 yylval.sval = strdup(yytext);
                 return VALUE_16;
                 }

0b[01]{5}        {

//A 5 bit binary number.

                 long num;
                 yytext[0] = '0';
                 yytext[1] = '0';
                 num = strtol(yytext, (char **)NULL, 2);
                 yytext = make_binary(num, 5);
                 yylval.sval = strdup(yytext);
                 return VALUE_5;
                 }

0x[0-9a-fA-F]{2} {

//A 2 digit hex number signifies a 5 bit number.

                 long num = strtol(yytext, (char **)NULL, 0);
                 yytext = make_binary(num, 5);
                 yylval.sval = strdup(yytext);
                 return VALUE_5;
                 }

0x[0-9a-fA-F]{4} {

//A 4 digit hex number signifies a 16 bit number.  The fill
//int is used for sign extension if/when I turn the number
//to 32 bits.

                 long num = strtol(yytext, (char **)NULL, 0);
                 if(num >= 32768) {
                         fill_int = 1;
                 }
                 else {
                         fill_int = 0;
                 }
                 yytext = make_binary(num, 16);
                yylval.sval = strdup(yytext);
                return VALUE_16;
                 }

0x[0-9a-fA-F]{7} {

//A 7 digit hex number signifies a 26 bit number.

                 long num = strtol(yytext, (char **)NULL, 0);
                 yytext = make_binary(num, 26);
                yylval.sval = strdup(yytext);
                return VALUE_26;
                 }

0x[0-9a-fA-F]{8} {
```

```
//An 8 digit hext number signifies a 32 bit number.  I use
//the first digit to tell whether the number is positive or
//negative, which allows me to use only a 31 bit number
//for manipulation purposes.  I turn it back into 32 bits
//later.  This is all done due to the lack of a 32 bit
//signed data type.

                long num;
                negative = FALSE;

                if(yytext[2] == '8') {
                        yytext[2] = '0';
                        negative = TRUE;
                }
                else if(yytext[2] == '9') {
                        yytext[2] = '1';
                        negative = TRUE;
                }
                else if(yytext[2] == 'A') {
                        yytext[2] = '2';
                        negative = TRUE;
                }
                else if(yytext[2] == 'B') {
                        yytext[2] = '3';
                        negative = TRUE;
                }
                else if(yytext[2] == 'C') {
                        yytext[2] = '4';
                        negative = TRUE;
                }
                else if(yytext[2] == 'D') {
                        yytext[2] = '5';
                        negative = TRUE;
                }
                else if(yytext[2] == 'E') {
                        yytext[2] = '6';
                        negative = TRUE;
                }
                else if(yytext[2] == 'F') {
                        yytext[2] = '7';
                        negative = TRUE;
                }
                else {
                        negative = FALSE;
                }

                num = strtol(yytext, (char **)NULL, 0);
                yytext = make_binary(num, 32);
                if(negative == TRUE) {
                        yytext[0] = '1';
                }
                yylval.sval = strdup(yytext);
                return VALUE_32;
                }

-?[0-9]+        {

//A decimal value.  It could signify a 5, 16, 26, or 32
//bit value, and I use num_length to determine which
//it is.

                long num = atoi(yytext);

                negative = FALSE;

                yytext = make_binary(num, num_length);

                yylval.sval = strdup(yytext);
                if(num_length == FIVE)
                        return VALUE_5;
                if(num_length == SIXTEEN) {
                        if(num >= 32768 || num < 0) {
                                fill_int = 1;
                        }
                        else {
```

```
                                        fill_int = 0;
                                }
                                return VALUE_16;
                        }
                        if(num_length == TWENTY_SIX)
                                return VALUE_26;
                        if(num_length == THIRTY_TWO)
                                return VALUE_32;
                        }

\n                      {

//The end of line (EOL) term.  I keep track of the line
//number for error reporting/debugging purposes.

                        line_number++;
                        }

[\t ]+                  /* ignore whitespace*/ ;

"//"[^\n]*              /* ignore parts that are commented out with // */ ;

,                       return yytext[0];

\(                      return yytext[0];

\)                      return yytext[0];

.                       {

//Anything that is unmatched is an error, so
//print an error message to the user.

                        yylval.sval = strdup(yytext);
                        error_cause = 1;
                        return ERROR;
                        }


%%

#include <stdlib.h>
#define NUM_ENTRIES 197

//This struct keeps track of the labels by
//name and line number, as well as the next label
//they point to in the linked list.

struct LabelData {
        char *name;
        int line;
        struct LabelData *next;
};

//This struct is for the data memory entries,
//and is in linked list form.

struct DataEntry {
         int value;
         struct DataEntry *next;
};

struct LabelData *hash[NUM_ENTRIES];

extern void *malloc();

/*
 *
 *My function for keeping track of label locations, adding them into
 *a hash table of size 197 dependent upon the values of their ascii
 *characters.  This will provide a sufficiently random distribution
 *for my purposes.
 *
 */
```

```
int add_label(int line_number, char *word)
{

        struct LabelData *label;

        label = (struct LabelData *) malloc(sizeof(struct LabelData));

        label->next = hash[compute_hash(word)];

        label->name = (char *) malloc(strlen(word)+1);
        strcpy(label->name, word);
        label->line = line_number;
        hash[compute_hash(word)] = label;

//Debug lines which can be handy

//      printf("Word: %s         Line: %d\n", label->name, label->line);
//      printf("Hash: %d\n", compute_hash(word));

}


/*
 *
 *This is my function for computing the hash value for a given
 *label.  The hash value depends upon the ascii values of
 *the characters, multiplied by the position place of the
 *character.  This provides a sufficiently random distribution.
 *
 */

int compute_hash(char *word)
{
        int hashValue = 0;
        int i = 0;

        while(word[i] != '\0'){
                hashValue += (word[i] * (i + 1));
                i++;
        }

        return (hashValue % NUM_ENTRIES);
}

/*
 *
 *My function for looking up a label in the hash table.
 *If a match is found, I return the line that it was
 *found on.  This allows me to compute jump and branch
 *destination addresses.
 *
 */

int lookup_label(char *word)
{
//debug lines which can be handy

//      printf("Word: %s\n", word);
//      printf("Hash: %d\n", compute_hash(word));

        struct LabelData *label = hash[compute_hash(word)];

        for(; label; label = label->next) {
                if(strcmp(label->name, word) == 0)
                        return label->line;
        }


        return LOOKUP;

}



/*
```

```
 *
 *This is my linked list of data values.  "Load immediates"
 *require the data value to be written to memory so that
 *a "load word" can retrieve the value, since we support
 *no immediate instructions.
 *
 */

struct DataEntry *data;
struct DataEntry *data2;
extern void *malloc();


int add_data(int data_value, int negative)
{


        struct DataEntry *new_data;

        new_data = (struct DataEntry *) malloc(sizeof(struct DataEntry));

                new_data->next = data;
        if(negative == FALSE) {
                new_data->value = data_value;
        }
        else {
                new_data->value = (data_value - 2147483647 - 1);
        }

                data = new_data;

//a debug line which can be handy

//      printf("Data value: %d recorded.\n", new_data->value);

}

/*
 *
 *This simply reverses the order of the data list so that I can
 *print it easily.  I use a second linked list and copy the
 *first one into it in reversed order.
 *
 */

int copy_data()
{
        for(; data; data = data->next) {

        struct DataEntry *new_data;

        new_data = (struct DataEntry *) malloc(sizeof(struct DataEntry));
        new_data->next = data2;
        new_data->value = data->value;

        data2 = new_data;
        }
}

/*
 *My function for printing the data values.  This occurs
 *during pass 7 of the assembler.
 *
 */

int print_values()
{

        copy_data();

        for(; data2; data2 = data2->next) {
                fprintf(yyout,"  data %d\n", data2->value);
        }
}
```

```
/*
 *
 *This is my function for turning numbers into binary.
 *Some of the data fields I deal with are signed 32-bit
 *numbers, and there is no data type that I can easily
 *use to represent such a large number.  Instead I
 *sometimes record the sign of the number and turn it
 *into a 31 bit number so that it'll fit into the long
 *data type, and I use the sign information to add the
 *top bit later.
 *
 */

char *
make_binary(long num, int bits)
{
        char *value;
        int i;
        int temp;

        value = (char *)malloc(sizeof(char) * (bits + 1));
        if(num >= 0){
                for(i = 1; i <= bits; i++){
                        temp = num % 2;
                        if(temp == 0)
                                value[bits-i] = '0';
                        else
                                value[bits-i] = '1';
                        num = num / 2;
                }
        }
        else{
                num = num + 1;
                num = num * -1;
                for(i = 1; i <= bits; i++){
                        temp = num % 2;
                        if(temp == 0)
                                value[bits-i] = '1';
                        else
                                value[bits-i] = '0';
                        num = num / 2;
                }
        }
        value[bits] = '\0';
        return value;
}

/*
 *
 *This is my function for turning a register
 *into it's numerical representation.  This occurs
 *during the last pass of the assembler.
 *
 */

char *
reg_number(char *reg) {

        char *number;

        number = (char *)malloc(sizeof(char) * 5);

        if(!strcmp(reg, "$zero")) {
                number = "00000";
                return number;
        }
        else if(!strcmp(reg, "$at")) {
                number = "00001";
                return number;
        }
        else if(!strcmp(reg, "$v0")) {
                number = "00010";
                return number;
        }
```

```
else if(!strcmp(reg, "$v1")) {
        number = "00011";
        return number;
}
else if(!strcmp(reg, "$a0")) {
        number = "00100";
        return number;
}
else if(!strcmp(reg, "$a1")) {
        number = "00101";
        return number;
}
else if(!strcmp(reg, "$a2")) {
        number = "00110";
        return number;
}
else if(!strcmp(reg, "$a3")) {
        number = "00111";
        return number;
}
else if(!strcmp(reg, "$t0")) {
        number = "01000";
        return number;
}
else if(!strcmp(reg, "$t1")) {
        number = "01001";
        return number;
}
else if(!strcmp(reg, "$t2")) {
        number = "01010";
        return number;
}
else if(!strcmp(reg, "$t3")) {
        number = "01011";
        return number;
}
else if(!strcmp(reg, "$t4")) {
        number = "01100";
        return number;
}
else if(!strcmp(reg, "$t5")) {
        number = "01101";
        return number;
}
else if(!strcmp(reg, "$t6")) {
        number = "01110";
        return number;
}
else if(!strcmp(reg, "$t7")) {
        number = "01111";
        return number;
}
else if(!strcmp(reg, "$s0")) {
        number = "10000";
        return number;
}
else if(!strcmp(reg, "$s1")) {
        number = "10001";
        return number;
}
else if(!strcmp(reg, "$s2")) {
        number = "10010";
        return number;
}
else if(!strcmp(reg, "$s3")) {
        number = "10011";
        return number;
}
else if(!strcmp(reg, "$s4")) {
        number = "10100";
        return number;
}
else if(!strcmp(reg, "$s5")) {
        number = "10101";
        return number;
```

```
        }
        else if(!strcmp(reg, "$s6")) {
                number = "10110";
                return number;
        }
        else if(!strcmp(reg, "$s7")) {
                number = "10111";
                return number;
        }
        else if(!strcmp(reg, "$t8")) {
                number = "11000";
                return number;
        }
        else if(!strcmp(reg, "$t9")) {
                number = "11001";
                return number;
        }
        else if(!strcmp(reg, "$k0")) {
                number = "11010";
                return number;
        }
        else if(!strcmp(reg, "$k1")) {
                number = "11011";
                return number;
        }
        else if(!strcmp(reg, "$gp")) {
                number = "11100";
                return number;
        }
        else if(!strcmp(reg, "$sp")) {
                number = "11101";
                return number;
        }
        else if(!strcmp(reg, "$fp")) {
                number = "11110";
                return number;
        }
        else if(!strcmp(reg, "$ra")) {
                number = "11111";
                return number;
        }

        return number;

}
```

Appendix G – Yacc Assembler Code

```
%{

/****************************************************
 * File: comp.y
 * Author: Mark Holland
 * Last Modified: May 6, 2002
 *
 * This is the yacc file for my assembler, written
 * in lex and yacc.  This file accepts tokens from
 * the lexer and matches them in specific grammars,
 * which are basically legal instructions.
 *
 * The assembler takes a file of all MIPs non-floating
 * point/coprocessor instructions and maps them to
 * the following set: LW, SW, JALR, BGEZ, NOR, SUBU,
 * SYSCALL, and BREAK.
 ****************************************************/


#include <stdio.h>

#define FALSE 0
#define TRUE 1

//These extern variables are used in both the lex and
//yacc programs, so they are declared external here.

extern FILE *yyout;               //The output file
extern int fill_int;              //The fill integer, for
                                  //sign extending
extern int output_line_number;    //output line number
extern int compiler_pass; //pass of the compiler
extern int num_instructions;      //Number of total insts
extern int negative;              //Whether a number is <0
int label_num = 0;                //Used for giving unique
                                  //label names.

//Below are the tokens that come from the lexer.  They
//signify different parts of the instructions that the
//lexer sees.

//Following the token definitions is the grammar section.
//When a specific set of tokens matches a complete grammar,
//the proper text is outputted.  So if an input instruction
//that our processor doesn't support is seen, the instruction
//will be matched by it's grammar and we will output the
//expansion of that instruction.

//The bulk of this file is simple expansions of input
//instructions.

%}

%union {
        char *sval;
}

%token <sval> REG
%token <sval> LABEL
%token <sval> LABEL_DEST
%token <sval> TRAP
%token <sval> VALUE_32
%token <sval> VALUE_26
%token <sval> VALUE_16
%token <sval> VALUE_5
%token <sval> ERROR

%token <sval> DATA
%token <sval> NOP
%token <sval> LB
%token <sval> LBU
%token <sval> LH
```

```
%token <sval> LHU
%token <sval> LW
%token <sval> LWL
%token <sval> LWR
%token <sval> LUI
%token <sval> LI
%token <sval> LIJ
%token <sval> SB
%token <sval> SH
%token <sval> SW
%token <sval> SWR
%token <sval> SWL
%token <sval> J
%token <sval> JAL
%token <sval> JR
%token <sval> JALR
%token <sval> BEQ
%token <sval> BNE
%token <sval> BLEZ
%token <sval> BGTZ
%token <sval> BLTZ
%token <sval> BGEZ
%token <sval> BLTZAL
%token <sval> BGEZAL
%token <sval> AND
%token <sval> ANDI
%token <sval> OR
%token <sval> ORI
%token <sval> XOR
%token <sval> XORI
%token <sval> NOR
%token <sval> ADDI
%token <sval> ADDIU
%token <sval> ADD
%token <sval> ADDU
%token <sval> SUB
%token <sval> SUBU
%token <sval> MULT
%token <sval> MULTU
%token <sval> DIV
%token <sval> DIVU
%token <sval> SLTI
%token <sval> SLTIU
%token <sval> SLT
%token <sval> SLTU
%token <sval> SLL
%token <sval> SRL
%token <sval> SRA
%token <sval> SLLV
%token <sval> SRLV
%token <sval> SRAV
%token <sval> SYSCALL
%token <sval> BREAK


%%

list:           instruction
        |       label instruction
        |       label label instruction
        |       list instruction
        |       list label instruction
        |       list label label instruction
        ;

instruction:    LB REG ',' VALUE_16 '(' REG ')'
                {
                if(fill_int == 0) {
                        fprintf(yyout,"  addi32 $t3,%s,0b0000000000000000%s\n", $6, $4);
output_line_number++;
                }
                else {
                        fprintf(yyout,"  addi32 $t3,%s,0b1111111111111111%s\n", $6, $4);
output_line_number++;
                }
```

```
                fprintf(yyout,"  ori32 $t4,$t3,0xFFFFFFFC\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000004\n"); output_line_number++;
                fprintf(yyout,"  subu $t3,$t3,$t4\n"); output_line_number++;
                fprintf(yyout,"  lw %s,0($t3)\n", $2); output_line_number++;
                fprintf(yyout,"  sll $t4,$t4,3\n"); output_line_number++;
                fprintf(yyout,"  srlv %s,%s,$t4\n", $2, $2); output_line_number++;
                fprintf(yyout,"  andi32 $t3,%s,0x00000080\n", $2); output_line_number++;
                fprintf(yyout,"  subu $t3,$zero,$t3\n"); output_line_number++;
                fprintf(yyout,"  bgez $t3,POS#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  or %s,%s,$t3\n", $2, $2); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"POS#%d:    andi32 %s,%s,0x000000FF\n", label_num, $2, $2);
output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       LBU REG ',' VALUE_16 '(' REG ')'
                {
                if(fill_int == 0) {
                        fprintf(yyout,"  addi32 $t3,%s,0b0000000000000000%s\n", $6, $4);
output_line_number++;
                }
                else {
                        fprintf(yyout,"  addi32 $t3,%s,0b1111111111111111%s\n", $6, $4);
output_line_number++;
                }
                fprintf(yyout,"  ori32 $t4,$t3,0xFFFFFFFC\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000004\n"); output_line_number++;
                fprintf(yyout,"  subu $t3,$t3,$t4\n"); output_line_number++;
                fprintf(yyout,"  lw %s,0($t3)\n", $2); output_line_number++;
                fprintf(yyout,"  sll $t4,$t4,3\n"); output_line_number++;
                fprintf(yyout,"  srlv %s,%s,$t4\n", $2, $2); output_line_number++;
                fprintf(yyout,"  andi32 %s,%s,0x000000FF\n", $2, $2); output_line_number++;
                }

        |       LH REG ',' VALUE_16 '(' REG ')'
                {
               if(fill_int == 0) {
                       fprintf(yyout,"    addi32 $t3,%s,0b0000000000000000%s\n", $6, $4);
output_line_number++;
                        fprintf(yyout,"  li32 $t4,0b0000000000000000%s\n", $4); output_line_number++;
                }
               else {
                       fprintf(yyout,"    addi32 $t3,%s,0b1111111111111111%s\n",$6, $4);
output_line_number++;
                        fprintf(yyout,"  li32 $t4,0b1111111111111111%s\n", $4); output_line_number++;
                }
                fprintf(yyout,"  ori32 $t4,$t3,0xFFFFFFFC\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000002\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,LOADHI#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  lw $t3,0($t3)\n"); output_line_number++;
                fprintf(yyout,"  andi32 $t4,$t3,0x00008000\n"); output_line_number++;
                fprintf(yyout,"  subu $t4,$zero,$t4\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,POS#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  or %s,$t3,$t4\n", $2); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"POS#%d:    andi32 %s,$t3,0x0000FFFF\n", label_num, $2);
output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"LOADHI#%d: lw %s,-2($t3)\n", label_num, $2); output_line_number++;
                fprintf(yyout,"  sra %s,%s,16\n", $2, $2); output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       LHU REG ',' VALUE_16 '(' REG ')'
                {
                if(fill_int == 0) {
                        fprintf(yyout,"  addi32 $t3,%s,0b0000000000000000%s\n", $6, $4);
output_line_number++;
                }
                else {
                        fprintf(yyout,"  addi32 $t3,%s,0b1111111111111111%s\n", $6, $4);
output_line_number++;
```

```
                }
                fprintf(yyout,"  ori32 $t4,$t3,0xFFFFFFFC\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000002\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,LOADHI#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  lw $t3,0($t3)\n"); output_line_number++;
                fprintf(yyout,"  andi32 %s,$t3,0x0000FFFF\n", $2); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"LOADHI#%d: lw $t4,-2($t3)\n", label_num); output_line_number++;
                fprintf(yyout,"  srl %s,%s,16\n", $2, $2); output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       LW REG ',' VALUE_16 '(' REG ')'
                {
                if(compiler_pass != 9) {
                        fprintf(yyout,"  lw %s,0b%s(%s)\n", $2, $4, $6); output_line_number++;
                }
                else {
                        fprintf(yyout,"100011%s%s%s\n", $6, $2, $4); output_line_number++;
                }
                }

        |       LWL REG ',' VALUE_16 '(' REG ')'
                {
                if(fill_int == 0) {
                        fprintf(yyout,"  addi32 $t3,%s,0b0000000000000000%s\n", $6, $4);
output_line_number++;
                }
                else {
                        fprintf(yyout,"  addi32 $t3,%s,0b1111111111111111%s\n", $6, $4);
output_line_number++;
                }
                fprintf(yyout,"  andi32 $t4,$t3,0x00000003\n"); output_line_number++;
                fprintf(yyout,"  subu $t3,$t3,$t4\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0xFFFFFFFD\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,AL3#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  lw $t3,0($t3)\n"); output_line_number++;
                fprintf(yyout,"  andi32 $t5,%s,0x000000FF\n", $2); output_line_number++;
                fprintf(yyout,"  addi32 $t6,$zero,0x00000007\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,SHIFT#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  andi32 $t5,%s,0x0000FFFF\n", $2); output_line_number++;
                fprintf(yyout,"  addi32 $t6,$t6,0x00000008\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,SHIFT#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  andi32 $t5,%s,0x00FFFFFF\n", $2); output_line_number++;
                fprintf(yyout,"  addi32 $t6,$t6,0x00000008\n"); output_line_number++;
                fprintf(yyout,"SHIFT#%d:  addi32 $t6,$t6,0xFFFFFFFF\n", label_num);
output_line_number++;
                fprintf(yyout,"  addu $t3,$t3,$t3\n"); output_line_number++;
                fprintf(yyout,"  bgez $t6,SHIFT#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  or %s,$t5,$t3\n", $2); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"AL3#%d:    lw %s,0($t3)\n", label_num, $2); output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       LWR REG ',' VALUE_16 '(' REG ')'
                {
                if(fill_int == 0) {
                        fprintf(yyout,"  addi32 $t3,%s,0b0000000000000000%s\n", $6, $4);
output_line_number++;
                }
                else {
                        fprintf(yyout,"  addi32 $t3,%s,0b1111111111111111%s\n", $6, $4);
output_line_number++;
                }
                fprintf(yyout,"  andi32 $t4,$t3,0x00000003\n"); output_line_number++;
                fprintf(yyout,"  subu $t3,$t3,$t4\n"); output_line_number++;
                fprintf(yyout,"  lw $t3,0($t3)\n"); output_line_number++;
                fprintf(yyout,"  blez $t4,ALIGNED#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0xFFFFFFFC\n"); output_line_number++;
                fprintf(yyout,"  subu $t4,$zero,$t4\n"); output_line_number++;
```

```
			fprintf(yyout,"  addi32 $t6,$zero,0x00000007\n"); output_line_number++;
			fprintf(yyout,"AL1#%d:    addi32 $t6,$t6,0xFFFFFFFF\n", label_num);
output_line_number++;
			fprintf(yyout,"  subu $t5,$zero,$t3\n"); output_line_number++;
			fprintf(yyout,"  bgez $t3,NOT1#%d\n", label_num); output_line_number++;
			fprintf(yyout,"  subu $t3,$t3,$t5\n"); output_line_number++;
			fprintf(yyout,"  addi32 $t3,$t3,0x00000001\n"); output_line_number++;
			fprintf(yyout,"  bgez $t6,AL1#%d\n", label_num); output_line_number++;
			fprintf(yyout,"  bgez $zero,FIN1#%d\n", label_num); output_line_number++;
			fprintf(yyout,"NOT1#%d:   subu $t3,$t3,$t5\n", label_num); output_line_number++;
			fprintf(yyout,"  bgez $t6,AL1#%d\n", label_num); output_line_number++;
			fprintf(yyout,"FIN1#%d:   addi32 $t4,$t4,0xFFFFFFFF\n", label_num);
output_line_number++;
			fprintf(yyout,"  bgtz $t4,AL2#%d\n", label_num); output_line_number++;
			fprintf(yyout,"  andi32 $t3,$t3,0x000000FF\n"); output_line_number++;
			fprintf(yyout,"  andi32 %s,%s,0xFFFFFF00\n", $2, $2); output_line_number++;
			fprintf(yyout,"  or %s,%s,$t3\n", $2, $2); output_line_number++;
			fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
			fprintf(yyout,"AL2#%d:    addi32 $t6,$zero,0x00000007\n", label_num);
output_line_number++;
			fprintf(yyout,"LOOP2#%d:  addi32 $t6,$t6,0xFFFFFFFF\n", label_num);
output_line_number++;
			fprintf(yyout,"  subu $t5,$zero,$t3\n"); output_line_number++;
			fprintf(yyout,"  bgez $t3,NOT2#%d\n", label_num); output_line_number++;
			fprintf(yyout,"  subu $t3,$t3,$t5\n"); output_line_number++;
			fprintf(yyout,"  addi32 $t3,$t3,0x00000001\n"); output_line_number++;
			fprintf(yyout,"  bgez $t6,LOOP2#%d\n", label_num); output_line_number++;
			fprintf(yyout,"  bgez $zero,FIN2#%d\n", label_num); output_line_number++;
			fprintf(yyout,"NOT2#%d:   subu $t3,$t3,$t5\n", label_num); output_line_number++;
			fprintf(yyout,"  bgez $t6,LOOP2#%d\n", label_num); output_line_number++;
			fprintf(yyout,"FIN2#%d:   addi32 $t4,$t4,0xFFFFFFFF\n", label_num);
output_line_number++;
			fprintf(yyout,"  bgtz $t4,AL3#%d\n", label_num); output_line_number++;
			fprintf(yyout,"  andi32 $t3,$t3,0x0000FFFF\n"); output_line_number++;
			fprintf(yyout,"  andi32 %s,%s,0xFFFF0000\n", $2, $2); output_line_number++;
			fprintf(yyout,"  or %s,%s,$t3\n", $2, $2); output_line_number++;
			fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
			fprintf(yyout,"AL3#%d:    addi32 $t6,$zero,0x00000007\n", label_num);
output_line_number++;
			fprintf(yyout,"LOOP3#%d:  addi32 $t6,$t6,0xFFFFFFFF\n", label_num);
output_line_number++;
			fprintf(yyout,"  subu $t5,$zero,$t3\n"); output_line_number++;
			fprintf(yyout,"  bgez $t3,NOT3#%d\n", label_num); output_line_number++;
			fprintf(yyout,"  subu $t3,$t3,$t5\n"); output_line_number++;
			fprintf(yyout,"  addi32 $t3,$t3,0x00000001\n"); output_line_number++;
			fprintf(yyout,"  bgez $t6,LOOP3#%d\n", label_num); output_line_number++;
			fprintf(yyout,"  bgez $zero,FIN3#%d\n", label_num); output_line_number++;
			fprintf(yyout,"NOT3#%d:   subu $t3,$t3,$t5\n", label_num); output_line_number++;
			fprintf(yyout,"  bgez $t6,LOOP3#%d\n", label_num); output_line_number++;
			fprintf(yyout,"FIN3#%d:   andi32 $t3,$t3,0x00FFFFFF\n", label_num);
output_line_number++;
			fprintf(yyout,"  andi32 %s,%s,0xFF000000\n", $2, $2); output_line_number++;
			fprintf(yyout,"  or %s,%s,$t3\n", $2, $2); output_line_number++;
			fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
			fprintf(yyout,"ALIGNED#%d:        subu %s,$t3,$zero\n", label_num, $2);
output_line_number++;
			fprintf(yyout,"END#%d:", label_num);
		       label_num++;
			}

	|		LUI REG ',' VALUE_16
			{
			fprintf(yyout,"  li32 %s,0b%s0000000000000000\n", $2, $4); output_line_number++;
			}

	|		LUI REG ',' VALUE_32
			{
			fprintf(yyout,"  li32 %s,0b%s\n", $2, $4); output_line_number++;
			}

	|		LI REG ',' VALUE_16
			{
			if(fill_int == 0) {
				fprintf(yyout,"  li32 %s,0b0000000000000000%s\n", $2, $4);
output_line_number++;
```

```
                }
                else {
                        fprintf(yyout,"  li32 %s,0b1111111111111111%s\n", $2, $4);
output_line_number++;
                }
                }

        |       LI REG ',' VALUE_32
                {
                if(compiler_pass != 7) {
                        fprintf(yyout,"  li32 %s,0b%s\n", $2, $4); output_line_number++;
                }
                else {
                        fprintf(yyout,"  lw %s,%d($zero)\n", $2, (4 * num_instructions));
output_line_number++;
                        num_instructions++;
                }
                }

        |       LI REG ',' BLTZAL
                {
                if(compiler_pass != 5) {
                        fprintf(yyout,"  li %s,bltzal\n", $2); output_line_number++;
                }
                else {
                        fprintf(yyout,"  li %s,%d\n", $2, (output_line_number + 20));
output_line_number++;
                }
                }

        |       LI REG ',' BGEZAL
                {
                 if(compiler_pass != 5) {
                        fprintf(yyout," li %s,bgezal\n", $2); output_line_number++;
                }
                 else {
                        fprintf(yyout,"  li %s,%d\n", $2, (output_line_number + 19));
output_line_number++;
                }
                }

        |       LIJ REG ',' LABEL
                {
                if(compiler_pass == 6) {
                        if(fill_int == 0) {
                                fprintf(yyout,"  li32 %s,0b0000000000000000%s\n", $2, $4);
output_line_number++;
                        }
                        else {
                                fprintf(yyout,"  li32 %s,0b1111111111111111%s\n", $2, $4);
output_line_number++;
                        }
                }
                else {
                        fprintf(yyout,"  lij %s,%s\n", $2, $4); output_line_number++;
                }
                }

        |       SB REG ',' VALUE_16 '(' REG ')'
                {
                if(fill_int == 0) {
                        fprintf(yyout,"  li32 $t4,0b0000000000000000%s\n", $4); output_line_number++;
                }
                else {
                        fprintf(yyout,"  li32 $t4,0b1111111111111111%s\n", $4); output_line_number++;
                }
                fprintf(yyout,"  add $t3,%s,$t4\n", $6); output_line_number++;
                fprintf(yyout,"  ori32 $t4,$t3,0xFFFFFFFC\n"); output_line_number++;
                fprintf(yyout,"  subu $t3,$t3,$t4\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t3,$t3,0xFFFFFFFC\n"); output_line_number++;
                fprintf(yyout,"  lw $t5,0($t3)\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,STOREMSB#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,STOREHI#%d\n", label_num); output_line_number++;
```

```
                fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,STORELO#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  andi32 $t4,$t5,0xFFFFFF00\n"); output_line_number++;
                fprintf(yyout,"  andi32 $t5,%s,0x000000FF\n", $2); output_line_number++;
                fprintf(yyout,"  bgez $zero,COMBINE#%d\n", label_num); output_line_number++;
                fprintf(yyout,"STORELO#%d:        andi32 $t4,$t5,0xFFFF00FF\n", label_num);
output_line_number++;
                fprintf(yyout,"  andi32 $t5,%s,0x000000FF\n", $2); output_line_number++;
                fprintf(yyout,"  li32 $t6,0x00000008\n"); output_line_number++;
                fprintf(yyout,"  bgez $zero,COMBINE#%d\n", label_num); output_line_number++;
                fprintf(yyout,"STOREHI#%d:        andi32 $t4,$t5,0xFF00FFFF\n", label_num);
output_line_number++;
                fprintf(yyout,"  andi32 $t5,%s,0x000000FF\n", $2); output_line_number++;
                fprintf(yyout,"  li32 $t6,0x00000010\n"); output_line_number++;
                fprintf(yyout,"  bgez $zero,COMBINE#%d\n", label_num); output_line_number++;
                fprintf(yyout,"STOREMSB#%d:       andi32 $t4,$t5,0x00FFFFFF\n", label_num);
output_line_number++;
                fprintf(yyout,"  andi32 $t5,%s,0x000000FF\n", $2); output_line_number++;
                fprintf(yyout,"  li32 $t6,0x00000018\n"); output_line_number++;
                fprintf(yyout,"COMBINE#%d:        sllv $t5,$t5,$t6\n", label_num);
output_line_number++;
                fprintf(yyout,"  or $t4,$t4,$t5\n"); output_line_number++;
                fprintf(yyout,"  sw $t4,0($t3)\n"); output_line_number++;
               label_num++;
                }

        |       SH REG ',' VALUE_16 '(' REG ')'
                {
                if(fill_int == 0) {
                        fprintf(yyout,"  addi32 $t3,%s,0b0000000000000000%s\n", $6, $4);
output_line_number++;
                }
                else {
                        fprintf(yyout,"  addi32 $t3,%s,0b1111111111111111%s\n", $6, $4);
output_line_number++;
                }
                fprintf(yyout,"  ori32 $t4,$t3,0xFFFFFFFC\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000002\n"); output_line_number++;
                fprintf(yyout,"  bgez $t4,LOADHI#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  lw $t4,0($t0)\n"); output_line_number++;
                fprintf(yyout,"  andi32 $t4,$t4,0xFFFF0000\n"); output_line_number++;
                fprintf(yyout,"  andi32 $t5,%s,0x0000FFFF\n", $2); output_line_number++;
                fprintf(yyout,"  or $t4,$t4,$t5\n"); output_line_number++;
                fprintf(yyout,"  sw $t4,0($t3)\n"); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"LOADHI#%d: lw $t4,-2($t3)\n", label_num); output_line_number++;
                fprintf(yyout,"  andi32 $t4,$t4,0x0000FFFF\n"); output_line_number++;
                fprintf(yyout,"  sll $t5,%s,16\n", $2); output_line_number++;
                fprintf(yyout,"  or $t4,$t4,$t5\n"); output_line_number++;
                fprintf(yyout,"  sw $t4,-2($t3)\n"); output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       SW REG ',' VALUE_16 '(' REG ')'
                {
                if(compiler_pass != 9) {
                        fprintf(yyout,"  sw %s,0b%s(%s)\n", $2, $4, $6); output_line_number++;
                }
                else {
                      fprintf(yyout,"101011%s%s%s\n", $6, $2, $4); output_line_number++;
                }
                }

        |       SWR REG ',' VALUE_16 '(' REG ')'
                {
                if(fill_int == 0) {
                        fprintf(yyout,"  li32 $t4,0b0000000000000000%s\n", $4); output_line_number++;
                }
                else {
                        fprintf(yyout," li32 $t4,0b1111111111111111%s\n", $4); output_line_number++;
                }
                fprintf(yyout,"  add $t3,%s,$t4\n", $6); output_line_number++;
                fprintf(yyout,"  ori32 $t4,$t3,0xFFFFFFFC\n"); output_line_number++;
                fprintf(yyout,"  subu $t3,$t3,$t4\n"); output_line_number++;
```

```
                 fprintf(yyout,"  addi32 $t3,$t3,0xFFFFFFFC\n"); output_line_number++;
                 fprintf(yyout,"  lw $t5,0($t3)\n"); output_line_number++;
                 fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,STOREMSB#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,STOREHI#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,STORELO#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  sw %s,0($t3)\n", $2); output_line_number++;
                 fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"STOREMSB#%d:       andi32 $t4,$t5,0x00FFFFFF\n", label_num);
output_line_number++;
                 fprintf(yyout,"  li32 $t5,0x00000018\n"); output_line_number++;
                 fprintf(yyout,"  bgez $zero,COMBINE#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"STOREHI#%d:        andi32 $t4,$t5,0x0000FFFF\n", label_num);
output_line_number++;
                 fprintf(yyout,"  li32 $t5,0x00000010\n"); output_line_number++;
                 fprintf(yyout,"  bgez $zero,COMBINE#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"STORELO#%d:        andi32 $t4,$t5,0x000000FF\n", label_num);
output_line_number++;
                 fprintf(yyout,"  li32 $t5,0x00000008\n"); output_line_number++;
                 fprintf(yyout,"COMBINE#%d:        sllv $t5,%s,$t5\n", label_num, $2);
output_line_number++;
                 fprintf(yyout,"  or $t4,$t4,$t5\n"); output_line_number++;
                 fprintf(yyout,"  sw $t4,0($t3)\n"); output_line_number++;
                 fprintf(yyout,"END#%d:", label_num);
                label_num++;
                 }

        |        SWL REG ',' VALUE_16 '(' REG ')'
                 {
                 if(fill_int == 0) {
                         fprintf(yyout,"  li32 $t4,0b0000000000000000%s\n", $4); output_line_number++;
                 }
                 else {
                         fprintf(yyout,"  li32 $t4,0b1111111111111111%s\n", $4); output_line_number++;
                 }
                 fprintf(yyout,"  add $t3,%s,$t4\n", $6); output_line_number++;
                 fprintf(yyout,"  ori32 $t4,$t3,0xFFFFFFFC\n"); output_line_number++;
                 fprintf(yyout,"  subu $t3,$t3,$t4\n"); output_line_number++;
                 fprintf(yyout,"  addi32 $t3,$t3,0xFFFFFFFC\n"); output_line_number++;
                 fprintf(yyout,"  lw $t5,0($t3)\n"); output_line_number++;
                 fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,STOREMSB#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,STOREHI#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,STORELO#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"  andi32 $t4,$t5,0xFFFFFF00\n"); output_line_number++;
                 fprintf(yyout,"  li32 $t5,0x00000018\n"); output_line_number++;
                 fprintf(yyout,"  bgez $zero,COMBINE#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"STORELO#%d:        andi32 $t4,$t5,0xFFFF0000\n", label_num);
output_line_number++;
                 fprintf(yyout,"  li32 $t5,0x00000010\n"); output_line_number++;
                 fprintf(yyout,"  bgez $zero,COMBINE#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"STOREHI#%d:        andi32 $t4,$t5,0xFF000000\n", label_num);
output_line_number++;
                 fprintf(yyout,"  li32 $t5,0x00000008\n"); output_line_number++;
                 fprintf(yyout,"  bgez $zero,COMBINE#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"STOREMSB#%d:       subu $t5,$zero,$zero\n", label_num);
output_line_number++;
                 fprintf(yyout,"COMBINE#%d:        srlv $t5,%s,$t5\n", label_num, $2);
output_line_number++;
                 fprintf(yyout,"  or $t4,$t4,$t5\n"); output_line_number++;
                 fprintf(yyout,"  sw $t4,0($t3)\n"); output_line_number++;
                label_num++;
                 }

        |        J LABEL
                 {
                 fprintf(yyout,"  lij $t3,%s\n", $2); output_line_number++;
                 fprintf(yyout,"  jalr $t3,$t3\n"); output_line_number++;
                 }
```

```
|       JAL LABEL
        {
        fprintf(yyout,"  lij $t3,%s\n", $2); output_line_number++;
        fprintf(yyout,"  jalr $ra,$t3\n"); output_line_number++;
        }

|       JR REG
        {
        fprintf(yyout,"  jalr $t3,%s\n", $2); output_line_number++;
        }

|       JALR REG ',' REG
        {
        if(compiler_pass != 9) {
                fprintf(yyout,"  jalr %s,%s\n", $2, $4); output_line_number++;
        }
        else {
                fprintf(yyout,"000000%s00000%s00000001001\n", $4, $2); output_line_number++;
        }
        }

|       JALR REG
        {
        if(compiler_pass != 9) {
                fprintf(yyout,"  jalr $ra,%s\n", $2); output_line_number++;
        }
        else {
                fprintf(yyout,"000000%s000001111100000001001\n", $2); output_line_number++;
        }
        }

|       BEQ REG ',' REG ',' LABEL
        {
        fprintf(yyout,"  subu $t3,%s,%s\n", $4, $2); output_line_number++;
        fprintf(yyout,"  bgez $t3,BEQ1#%d\n", label_num); output_line_number++;
        fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
        fprintf(yyout,"BEQ1#%d:   subu $t3,$zero,$t3\n", label_num); output_line_number++;
        fprintf(yyout,"  bgez $t3,%s\n", $6); output_line_number++;
       fprintf(yyout,"END#%d:", label_num);
       label_num++;
        }

|       BNE REG ',' REG ',' LABEL
        {
        fprintf(yyout,"  subu $t3,%s,%s\n", $4, $2); output_line_number++;
        fprintf(yyout,"  bgez $t3,BNE1#%d\n", label_num); output_line_number++;
        fprintf(yyout,"  bgez $zero,2\n"); output_line_number++;
        fprintf(yyout,"BNE1#%d    subu $t3,$zero,$t3\n", label_num); output_line_number++;
        fprintf(yyout,"  bgez $t3,END#%d\n", label_num); output_line_number++;
        fprintf(yyout,"  bgez $zero,%s\n", $6); output_line_number++;
       fprintf(yyout,"END#%d:", label_num);
       label_num++;
        }

|       BLEZ REG ',' LABEL
        {
        fprintf(yyout,"  bgez %s,BLEZ1#%d\n", $2, label_num); output_line_number++;
        fprintf(yyout,"  bgez $zero,%s\n", $4); output_line_number++;
        fprintf(yyout,"BLEZ1#%d:  subu $t7,$zero,%s\n", label_num, $2); output_line_number++;
        fprintf(yyout,"  bgez $t7,-3\n"); output_line_number++;
       label_num++;
        }

|       BGTZ REG ',' LABEL
        {
        fprintf(yyout,"  bgez %s,BGTZ1#%d\n", $2, label_num); output_line_number++;
        fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
        fprintf(yyout,"BGTZ1#%d:  subu $t3,$zero,%s\n", label_num, $2); output_line_number++;
        fprintf(yyout,"  bgez $t3,END#%d\n", label_num); output_line_number++;
        fprintf(yyout,"  bgez $zero,%s\n", $4); output_line_number++;
       fprintf(yyout,"END#%d:", label_num);
       label_num++;
        }

|       BLTZ REG ',' LABEL
```

```
		{
		fprintf(yyout,"  bgez %s,END#%d\n", $2, label_num); output_line_number++;
		fprintf(yyout,"  bgez $zero,%s\n", $4); output_line_number++;
		fprintf(yyout,"END#%d:", label_num);
		label_num++;
		}

	|	BGEZ REG ',' LABEL
		{
		fprintf(yyout,"  bgez %s,%s\n", $2, $4); output_line_number++;
		}

	|	BGEZ REG ',' TRAP
		{
		if(compiler_pass != 5) {
			fprintf(yyout,"  bgez %s,%s\n", $2, $4); output_line_number++;
		}
		else {
			fprintf(yyout,"  bgez %s,%d\n", $2, (num_instructions - output_line_number -
3));

			output_line_number++;
		}
		}

	|	BGEZ REG ',' VALUE_16
		{
		if(compiler_pass != 9) {
			fprintf(yyout,"  bgez %s,0b%s\n", $2, $4); output_line_number++;
		}
		else {
			fprintf(yyout,"000001%s00001%s\n", $2, $4); output_line_number++;
		}
		}

	|	BLTZAL REG ',' LABEL
		{
		fprintf(yyout,"  addi $ra,$zero,bltzal\n"); output_line_number++;
		fprintf(yyout,"  bgez %s,END#%d\n", $2, label_num); output_line_number++;
		fprintf(yyout,"  bgez $zero,%s\n", $4); output_line_number++;
		fprintf(yyout,"END#%d:", label_num);
		label_num++;
		}

	|	BGEZAL REG ',' LABEL
		{
		fprintf(yyout,"  addi $ra,$zero,bgezal\n"); output_line_number++;
		fprintf(yyout,"  bgez %s,%s\n", $2, $4); output_line_number++;
		}

	|	AND REG ',' REG ',' REG
		{
		fprintf(yyout,"  nor $s5,%s,%s\n", $4, $4); output_line_number++;
		fprintf(yyout,"  nor $at,%s,%s\n", $6, $6); output_line_number++;
		fprintf(yyout,"  nor %s,$s5,$at\n", $2); output_line_number++;
		}

	|	ANDI REG ',' REG ',' VALUE_16
		{
		fprintf(yyout,"  li32 $at,0b0000000000000000%s\n", $6); output_line_number++;
		fprintf(yyout,"  nor $s5,%s,%s\n", $4, $4); output_line_number++;
		fprintf(yyout,"  nor $at,$at,$at\n"); output_line_number++;
		fprintf(yyout,"  nor %s,$s5,$at\n", $2); output_line_number++;
		}

	|	ANDI REG ',' REG ',' VALUE_32
		{
		fprintf(yyout,"  li32 $at,0b%s\n", $6); output_line_number++;
		fprintf(yyout,"   nor $s5,%s,%s\n", $4, $4); output_line_number++;
		fprintf(yyout,"   nor $at,$at,$at\n"); output_line_number++;
		fprintf(yyout,"   nor %s,$s5,$at\n", $2); output_line_number++;
		}

	|	OR REG ',' REG ',' REG
		{
		fprintf(yyout,"  nor $t7,%s,%s\n", $4, $6); output_line_number++;
```

```
        fprintf(yyout,"  nor %s,$t7,$t7\n", $2); output_line_number++;
        }

|       ORI REG ',' REG ',' VALUE_16
        {
        fprintf(yyout,"  li32 $t7,0b0000000000000000%s\n", $6); output_line_number++;
        fprintf(yyout,"  nor $t7,%s,$t7\n", $4); output_line_number++;
        fprintf(yyout,"  nor %s,$t7,$t7\n", $2); output_line_number++;
        }

|       ORI REG ',' REG ',' VALUE_32
        {
        fprintf(yyout,"  li32 $t7,0b%s\n", $6); output_line_number++;
        fprintf(yyout,"  nor $t7,%s,$t7\n", $4); output_line_number++;
        fprintf(yyout,"  nor %s,$t7,$t7\n", $2); output_line_number++;
        }

|       XOR REG ',' REG ',' REG
        {
        fprintf(yyout,"  nor $t3,%s,%s\n", $4, $4); output_line_number++;
        fprintf(yyout,"  nor $t4,%s,%s\n", $6, $6); output_line_number++;
        fprintf(yyout,"  nor $t3,$t3,$t4\n"); output_line_number++;
        fprintf(yyout,"  nor $t4,%s,%s\n", $4, $6); output_line_number++;
        fprintf(yyout,"  nor %s,$t3,$t4\n", $2); output_line_number++;
        }

|       XORI REG ',' REG ',' VALUE_16
        {
        fprintf(yyout,"  li32 $t5,0b0000000000000000%s\n", $6); output_line_number++;
        fprintf(yyout,"  nor $t3,%s,%s\n", $4, $4); output_line_number++;
       fprintf(yyout,"   nor $t4,$t5,$t5\n"); output_line_number++;
       fprintf(yyout,"   nor $t3,$t3,$t4\n"); output_line_number++;
       fprintf(yyout,"   nor $t4,%s,$t5\n", $4); output_line_number++;
       fprintf(yyout,"   nor %s,$t3,$t4\n", $2); output_line_number++;
        }

|       XORI REG ',' REG ',' VALUE_32
        {
       fprintf(yyout," li32 $t5,0b%s\n", $6); output_line_number++;
       fprintf(yyout," nor $t3,%s,%s\n", $4, $4); output_line_number++;
       fprintf(yyout," nor $t4,$t5,$t5\n"); output_line_number++;
       fprintf(yyout," nor $t3,$t3,$t4\n"); output_line_number++;
       fprintf(yyout," nor $t4,%s,$t5\n", $4); output_line_number++;
       fprintf(yyout," nor %s,$t3,$t4\n", $2); output_line_number++;
       }


|       NOR REG ',' REG ',' REG
        {
        if(compiler_pass != 9) {
                fprintf(yyout,"  nor %s,%s,%s\n", $2, $4, $6); output_line_number++;
        }
        else {
                fprintf(yyout,"000000%s%s%s00000100111\n", $4, $6, $2); output_line_number++;
        }
        }

|       ADDI REG ',' REG ',' VALUE_16
        {
        if(fill_int == 0) {
                fprintf(yyout,"  li32 $s5,0b0000000000000000%s\n", $6); output_line_number++;
        }
        else {
                fprintf(yyout,"  li32 $s5,0b1111111111111111%s\n", $6); output_line_number++;
        }
        fprintf(yyout,"  add %s,%s,$s5\n", $2, $4); output_line_number++;
        }

|       ADDI REG ',' REG ',' VALUE_32
        {
        fprintf(yyout,"  li32 $s5,0b%s\n", $6); output_line_number++;
        fprintf(yyout,"  add %s,%s,$s5\n", $2, $4); output_line_number++;
        }

|       ADDI REG ',' REG ',' BLTZAL
```

```
		{
		fprintf(yyout,"  li $s5,bltzal\n"); output_line_number++;
		fprintf(yyout,"  add %s,%s,$s5\n", $2, $4); output_line_number++;
		}

	|	ADDI REG ',' REG ',' BGEZAL
		{
		fprintf(yyout,"  li $s5,bgezal\n"); output_line_number++;
		fprintf(yyout,"  add %s,%s,$s5\n", $2, $4); output_line_number++;
		}

	|	ADDIU REG ',' REG ',' VALUE_16
		{
		if(fill_int == 0) {
			fprintf(yyout,"  li32 $s5,0b0000000000000000%s\n", $6); output_line_number++;
		}
		else {
			fprintf(yyout,"  li32 $s5,0b1111111111111111%s\n", $6); output_line_number++;
		}
		fprintf(yyout,"  addu %s,%s,$s5\n", $2, $4); output_line_number++;
		}

	|	ADDIU REG ',' REG ',' VALUE_32
		{
		fprintf(yyout,"  li32 $s5,0b%s\n", $6); output_line_number++;
		fprintf(yyout,"  addu %s,%s,$s5\n", $2, $4); output_line_number++;
		}

	|	ADD REG ',' REG ',' REG
		{
		fprintf(yyout,"  bgez %s,POS1#%d\n", $4, label_num); output_line_number++;
		fprintf(yyout,"  bgez %s,OPP2#%d\n", $6, label_num); output_line_number++;
		fprintf(yyout,"  subu $at,$zero,%s\n", $6); output_line_number++;
		fprintf(yyout,"  subu $at,%s,$at\n", $4); output_line_number++;
		fprintf(yyout,"  bgez $at,*TRAP*\n"); output_line_number++;
		fprintf(yyout,"  bgez $zero,FINISH#%d\n", label_num); output_line_number++;
		fprintf(yyout,"POS1#%d:   bgez %s,POSIT#%d\n", label_num, $6, label_num);
output_line_number++;
		fprintf(yyout,"  subu $at,$zero,%s\n", $4); output_line_number++;
		fprintf(yyout,"  bgez $zero,OPP1#%d\n", label_num); output_line_number++;
		fprintf(yyout,"OPP2#%d:   subu $at,$zero,%s\n", label_num, $6); output_line_number++;
		fprintf(yyout,"OPP1#%d:   subu %s,%s,$at\n", label_num, $2, $4); output_line_number++;
		fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
		fprintf(yyout,"POSIT#%d:  subu $at,$zero,%s\n", label_num, $6); output_line_number++;
		fprintf(yyout,"  subu $at,%s,$at\n", $4); output_line_number++;
		fprintf(yyout,"  bgez $at,FINISH#%d\n", label_num); output_line_number++;
		fprintf(yyout,"  bgez $zero,*TRAP*\n"); output_line_number++;
		fprintf(yyout,"FINISH#%d: subu %s,$at,$zero\n", label_num, $2); output_line_number++;
		fprintf(yyout,"END#%d:", label_num);
		label_num++;
		}

	|	ADDU REG ',' REG ',' REG
		{
		fprintf(yyout,"  subu $at,$zero,%s\n", $6); output_line_number++;
		fprintf(yyout,"  subu %s,%s,$at\n", $2, $4); output_line_number++;
		}

	|	SUB REG ',' REG ',' REG
		{
		fprintf(yyout,"  subu $t3,$zero,%s\n", $6); output_line_number++;
		fprintf(yyout,"  add %s,%s,$t3\n", $2, $4); output_line_number++;
		}

	|	SUBU REG ',' REG ',' REG
		{
		if(compiler_pass != 9) {
			fprintf(yyout,"  subu %s,%s,%s\n", $2, $4, $6); output_line_number++;
		}
		else {
			fprintf(yyout,"000000%s%s%s00000100011\n", $4, $6, $2); output_line_number++;
		}
		}

	|	MULT REG ',' REG
```

```
                {
                fprintf(yyout,"  subu $t5,%s,$zero\n", $4); output_line_number++;
                fprintf(yyout,"  subu $t4,$zero,$zero\n"); output_line_number++;
                fprintf(yyout,"  bgez %s,NOCMAX#%d\n", $2, label_num); output_line_number++;
                fprintf(yyout,"  subu $t3,$zero,%s\n", $2); output_line_number++;
                fprintf(yyout,"  bgez $t3,NOCMAX#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  bgez $t5,MAXMPOS#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  subu $t5,$zero,$t5\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t4,$t4,0x00000001\n"); output_line_number++;
                fprintf(yyout,"MAXMPOS#%d:        addi32 $t4,$t4,0xFFFFFFFF\n", label_num);
output_line_number++;
                fprintf(yyout,"  bgez $zero,COUNTER#%d\n", label_num); output_line_number++;
                fprintf(yyout,"NOCMAX#%d: subu $t3,%s,$zero\n", label_num, $2); output_line_number++;
                fprintf(yyout,"  bgez $t5,NOMAXMPOS#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  subu $t5,$zero,$t5\n"); output_line_number++;
                fprintf(yyout,"  subu $t3,$zero,$t3\n"); output_line_number++;
                fprintf(yyout,"NOMAXMPOS#%d:      bgez $t3,COUNTER#%d\n", label_num, label_num);
output_line_number++;
                fprintf(yyout,"  addi32 $t4,$zero,0xFFFFFFFF\n"); output_line_number++;
                fprintf(yyout,"COUNTER#%d:        subu $s6,$zero,$zero\n", label_num);
output_line_number++;
                fprintf(yyout,"  subu $s7,$zero,$zero\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t6,$zero,0xFFFFFFE0\n"); output_line_number++;
                fprintf(yyout,"LOOP#%d:   bgez $t5,ZERO#%d\n", label_num, label_num);
output_line_number++;
                fprintf(yyout,"  bgez $s6,CHKCARRY#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  bgez $t3,ERCARRY#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  addu $s6,$s6,$t3\n"); output_line_number++;
                fprintf(yyout,"CARRYOUT#%d:       addi32 $s7,$s7,0x00000001\n", label_num);
output_line_number++;
                fprintf(yyout,"  bgez $zero,PHIMULT#%d\n", label_num); output_line_number++;
                fprintf(yyout,"CHKCARRY#%d:       bgez $t3,PNOCARRY#%d\n", label_num, label_num);
output_line_number++;
                fprintf(yyout,"ERCARRY#%d:        addu $s6,$s6,$t3\n", label_num);
output_line_number++;
                fprintf(yyout,"  bgez $s6,CARRYOUT#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  bgez $zero,PHIMULT#%d\n", label_num); output_line_number++;
                fprintf(yyout,"PNOCARRY#%d:       addu $s6,$s6,$t3\n", label_num);
output_line_number++;
                fprintf(yyout,"PHIMULT#%d:        addu $s7,$s7,$t4\n", label_num);
output_line_number++;
                fprintf(yyout,"ZERO#%d:   addi32 $t6,$t6,0x00000001\n", label_num);
output_line_number++;
                fprintf(yyout,"  bgez $t6,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  addu $s7,$s7,$s7\n"); output_line_number++;
                fprintf(yyout,"  bgez $s6,PHIZERO#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  addiu32 $s7,$s7,0x00000001\n"); output_line_number++;
                fprintf(yyout,"PHIZERO#%d:        addu $s6,$s6,$s6\n", label_num);
output_line_number++;
                fprintf(yyout,"  addu $t5,$t5,$t5\n"); output_line_number++;
                fprintf(yyout,"  bgez $zero,LOOP#%d\n", label_num); output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       MULTU REG ',' REG
                {
                fprintf(yyout,"  subu $t5,%s,$zero\n", $4); output_line_number++;
                fprintf(yyout,"  subu $t3,%s,$zero\n", $2); output_line_number++;
               fprintf(yyout,"   subu $t4,$zero,$zero\n"); output_line_number++;
               fprintf(yyout,"   subu $s6,$zero,$zero\n"); output_line_number++;
                fprintf(yyout,"  subu $s7,$zero,$zero\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t6,$zero,0xFFFFFFE0\n"); output_line_number++;
               fprintf(yyout,"LOOP#%d:    bgez $t5,ZERO#%d\n", label_num, label_num);
output_line_number++;
               fprintf(yyout,"   bgez $s6,CHKCARRY#%d\n", label_num); output_line_number++;
               fprintf(yyout,"   bgez $t3,ERCARRY#%d\n", label_num); output_line_number++;
               fprintf(yyout,"   addu $s6,$s6,$t3\n"); output_line_number++;
               fprintf(yyout,"CARRYOUT#%d:        addi32 $s7,$s7,0x00000001\n", label_num);
output_line_number++;
               fprintf(yyout,"   bgez $zero,PHIMULT#%d\n", label_num); output_line_number++;
               fprintf(yyout,"CHKCARRY#%d:        bgez $t3,PNOCARRY#%d\n", label_num, label_num);
output_line_number++;
               fprintf(yyout,"ERCARRY#%d: addu $s6,$s6,$t3\n", label_num);  output_line_number++;
               fprintf(yyout,"   bgez $s6,CARRYOUT#%d\n", label_num); output_line_number++;
```

```
                fprintf(yyout,"   bgez $zero,PHIMULT#%d\n", label_num); output_line_number++;
                fprintf(yyout,"PNOCARRY#%d:        addu $s6,$s6,$t3\n", label_num);
output_line_number++;
                fprintf(yyout,"PHIMULT#%d: addu $s7,$s7,$t4\n", label_num); output_line_number++;
                fprintf(yyout,"ZERO#%d:    addi32 $t6,$t6,0x00000001\n", label_num);
output_line_number++;
                fprintf(yyout,"   bgez $t6,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"   addu $s7,$s7,$s7\n"); output_line_number++;
                fprintf(yyout,"   bgez $s6,PHIZERO#%d\n", label_num); output_line_number++;
                fprintf(yyout,"   addiu32 $s7,$s7,0x00000001\n"); output_line_number++;
                fprintf(yyout,"PHIZERO#%d: addu $s6,$s6,$s6\n", label_num);  output_line_number++;
                fprintf(yyout,"   addu $t5,$t5,$t5\n"); output_line_number++;
                fprintf(yyout,"   bgez $zero,LOOP#%d\n", label_num); output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
                label_num++;
                 }

        |        DIV REG ',' REG
                 {
                 fprintf(yyout,"  subu $s7,$zero,$zero\n"); output_line_number++;
                 fprintf(yyout,"  subu $t3,%s,$zero\n", $2); output_line_number++;
                 fprintf(yyout,"  subu $t8,%s,$zero\n", $4); output_line_number++;
                 fprintf(yyout,"  subu $s6,$zero,$zero\n"); output_line_number++;
                 fprintf(yyout,"  subu $t4,$zero,$zero\n"); output_line_number++;
                 fprintf(yyout,"  subu $t5,$zero,$zero\n"); output_line_number++;
                 fprintf(yyout,"  subu $t6,$t3,$zero\n"); output_line_number++;
                 fprintf(yyout,"  addi32 $t7,$zero,0xFFFFFFDF\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t3,TEST1#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  subu $t4,$zero,$t8\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,NEGS#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addi32 $t5,$zero,0xFFFFFFFF\n"); output_line_number++;
                 fprintf(yyout,"  subu $t3,$zero,$t3\n"); output_line_number++;
                 fprintf(yyout,"  bgez $zero,LOOP#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"TEST1#%d:  bgez $t8,LOOP#%d\n", label_num, label_num);
output_line_number++;
                 fprintf(yyout,"  addi32 $t5,$zero,0xFFFFFFFF\n"); output_line_number++;
                 fprintf(yyout,"  subu $t8,$zero,$t8\n"); output_line_number++;
                 fprintf(yyout,"  bgez $zero,LOOP#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"NEGS#%d:   subu $t3,$zero,$t3\n", label_num); output_line_number++;
                 fprintf(yyout,"  subu $t8,$zero,$t8\n"); output_line_number++;
                 fprintf(yyout,"  subu $t4,$zero,$zero\n"); output_line_number++;
                 fprintf(yyout,"LOOP#%d:   addu $s6,$s6,$s6\n", label_num); output_line_number++;
                 fprintf(yyout,"  subu $t9,$s7,$t8\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t9,SUBTRACT#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  bgez $zero,FINAL#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"SUBTRACT#%d:       subu $s7,$t9,$zero\n", label_num);
output_line_number++;
                 fprintf(yyout,"  addiu32 $s6,$s6,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"FINAL#%d:  addi32 $t7,$t7,0x00000001\n", label_num);
output_line_number++;
                 fprintf(yyout,"  bgez $t7,LAST#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addu $s7,$s7,$s7\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t3,SHIFT#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addiu32 $s7,$s7,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"SHIFT#%d:  addu $t3,$t3,$t3\n", label_num); output_line_number++;
                 fprintf(yyout,"  bgez $zero,LOOP#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"LAST#%d:   bgez $t5,DENDNEG#%d\n", label_num, label_num);
output_line_number++;
                 fprintf(yyout,"  subu $s6,$zero,$s6\n"); output_line_number++;
                 fprintf(yyout,"DENDNEG#%d:        bgez $s1,END#%d\n", label_num, label_num);
output_line_number++;
                 fprintf(yyout,"  subu $s7,$zero,$s7\n"); output_line_number++;
                 fprintf(yyout,"END#%d:", label_num);
                label_num++;
                 }

        |        DIVU REG ',' REG
                 {
                fprintf(yyout,"   subu $s6,$zero,$zero\n");   output_line_number++;
                 fprintf(yyout,"  sltu $t4,%s,%s\n", $2, $4); output_line_number++;
                 fprintf(yyout,"  subu $t4,$zero,$t4\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,DODIV#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  subu $s7,%s,$zero\n", $2); output_line_number++;
                 fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
```

```
                 fprintf(yyout,"DODIV#%d:  bgez %s,POS#%d\n", label_num, $4, label_num);
output_line_number++;
                 fprintf(yyout,"  addiu32 $t3,%s,0x80000000\n", $2); output_line_number++;
                 fprintf(yyout,"  addiu32 $t8,%s,0x80000000\n", $4); output_line_number++;
                 fprintf(yyout,"  bgez $zero,NEG#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"POS#%d:    subu $t3,%s,$zero\n", label_num, $2); output_line_number++;
                 fprintf(yyout,"  subu $t8,%s,$zero\n", $4); output_line_number++;
                 fprintf(yyout,"NEG#%d:    subu $s7,$zero,$zero\n", label_num); output_line_number++;
                 fprintf(yyout,"  subu $t4,$zero,$zero\n"); output_line_number++;
                 fprintf(yyout,"  subu $t5,$zero,$zero\n"); output_line_number++;
                 fprintf(yyout,"  subu $t6,$t3,$zero\n"); output_line_number++;
                 fprintf(yyout,"  addi32 $t7,$zero,0xFFFFFFDF\n"); output_line_number++;
                 fprintf(yyout,"LOOP#%d:   addu $s6,$s6,$s6\n", label_num); output_line_number++;
                 fprintf(yyout,"  subu $t9,$s7,$t8\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t9,SUBTRACT#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  bgez $zero,FINAL#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"SUBTRACT#%d:       subu $s7,$t9,$zero\n", label_num);
output_line_number++;
                 fprintf(yyout,"  addiu32 $s6,$s6,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"FINAL#%d:  addi32 $t7,$t7,0x00000001\n", label_num);
output_line_number++;
                 fprintf(yyout,"  bgez $t7,END#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addu $s7,$s7,$s7\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t3,SHIFT#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addiu32 $s7,$s7,0x00000001\n"); output_line_number++;
                 fprintf(yyout,"SHIFT#%d:  addu $t3,$t3,$t3\n", label_num); output_line_number++;
                 fprintf(yyout,"  bgez $zero,LOOP#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"END#%d:", label_num);
                label_num++;
                }

        |        SLTI REG ',' REG ',' VALUE_16
                 {
                 if(fill_int == 0) {
                         fprintf(yyout,"  li32 $t4,0b0000000000000000%s\n", $6); output_line_number++;
                 }
                 else {
                         fprintf(yyout,"  li32 $t4,0b1111111111111111%s\n", $6); output_line_number++;
                 }
                 fprintf(yyout,"  bgez %s,5\n", $4); output_line_number++;
                 fprintf(yyout,"  bgez $t4,2\n"); output_line_number++;
                 fprintf(yyout,"  subu $t3,%s,$t4\n", label_num, $4); output_line_number++;
                 fprintf(yyout,"  bgez $t3,3\n"); output_line_number++;
                 fprintf(yyout,"  li32 %s,0x00000001\n", label_num, $2); output_line_number++;
                 fprintf(yyout,"  bgez $zero,2\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,-5\n", label_num); output_line_number++;
                 fprintf(yyout,"  subu %s,$zero,$zero\n", label_num, $2); output_line_number++;
                label_num++;
                 }

        |        SLTI REG ',' REG ',' VALUE_32
                 {
                 fprintf(yyout,"  li32 $t4,0b%s\n", $6); output_line_number++;
                 fprintf(yyout," bgez %s,S1POS#%d\n", $4, label_num); output_line_number++;
                fprintf(yyout," bgez $t4,SET1#%d\n", label_num);   output_line_number++;
                fprintf(yyout,"T1POS#%d:   subu $t3,%s,$t4\n", label_num, $4); output_line_number++;
                fprintf(yyout," bgez $t3,SET0#%d\n", label_num); output_line_number++;
                fprintf(yyout,"SET1#%d:    li32 %s,0x00000001\n", label_num, $2); output_line_number++;
                fprintf(yyout," bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"S1POS#%d:   bgez $t4,T1POS#%d\n", label_num, label_num);
output_line_number++;
                fprintf(yyout,"SET0#%d:    subu %s,$zero,$zero\n", label_num, $2);
output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
                label_num++;
                 }

        |        SLTIU REG ',' REG ',' VALUE_16
                 {
                 if(fill_int == 0) {
                         fprintf(yyout,"  li32 $t3,0b0000000000000000%s\n", $6); output_line_number++;
                 }
                 else {
                         fprintf(yyout,"  li32 $t3,0b1111111111111111%s\n", $6); output_line_number++;
                 }
```

```
                fprintf(yyout,"  bgez %s,S1POS#%d\n", $4, label_num); output_line_number++;
                fprintf(yyout,"  bgez $t3,SET0#%d\n", label_num); output_line_number++;
                fprintf(yyout,"IMMPOS#%d: subu $t3,%s,$t3\n", label_num, $4); output_line_number++;
                fprintf(yyout,"  bgez $t3,SET0#%d\n", label_num); output_line_number++;
                fprintf(yyout,"SET1#%d:   li32 %s,0x00000001\n", label_num, $2); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"S1POS#%d:  bgez $t3,IMMPOS#%d\n", label_num, label_num);
output_line_number++;
                fprintf(yyout,"  bgez $zero,SET1#%d\n", label_num); output_line_number++;
                fprintf(yyout,"SET0#%d:   subu %s,$zero,$zero\n", label_num, $2);
output_line_number++;
               fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       SLTIU REG ',' REG ',' VALUE_32
                {
                fprintf(yyout,"  li32 $t3,0b%s\n", $6); output_line_number++;
                fprintf(yyout," bgez %s,S1POS#%d\n", $4, label_num); output_line_number++;
               fprintf(yyout," bgez $t3,SET0#%d\n", label_num); output_line_number++;
               fprintf(yyout,"IMMPOS#%d:  subu $t3,%s,$t3\n", label_num, $4); output_line_number++;
               fprintf(yyout," bgez $t3,SET0#%d\n", label_num); output_line_number++;
               fprintf(yyout,"SET1#%d:    li32 %s,0x00000001\n", label_num, $2); output_line_number++;
               fprintf(yyout," bgez $zero,END#%d\n", label_num); output_line_number++;
               fprintf(yyout,"S1POS#%d:   bgez $t3,IMMPOS#%d\n", label_num, label_num);
output_line_number++;
               fprintf(yyout," bgez $zero,SET1#%d\n", label_num); output_line_number++;
               fprintf(yyout,"SET0#%d:    subu %s,$zero,$zero\n", label_num, $2);
output_line_number++;
               fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       SLT REG ',' REG ',' REG
                {
                fprintf(yyout,"  bgez %s,S1POS#%d\n", $4, label_num); output_line_number++;
                fprintf(yyout,"  bgez %s,SET1#%d\n", $6, label_num); output_line_number++;
                fprintf(yyout,"S2POS#%d:  subu $t3,%s,%s\n", label_num, $4, $6); output_line_number++;
                fprintf(yyout,"  bgez $t3,SET0#%d\n", label_num); output_line_number++;
                fprintf(yyout,"SET1#%d:   li32 %s,0x00000001\n", label_num, $2); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"S1POS#%d:  bgez %s,S2POS#%d\n", label_num, $6, label_num);
output_line_number++;
                fprintf(yyout,"SET0#%d:   subu %s,$zero,$zero\n", label_num, $2);
output_line_number++;
               fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       SLTU REG ',' REG ',' REG
                {
                fprintf(yyout,"  bgez %s,S1POS#%d\n", $4, label_num); output_line_number++;
                fprintf(yyout,"  bgez %s,SET0#%d\n", $6, label_num); output_line_number++;
                fprintf(yyout,"S1POS#%d:  subu $at,%s,%s\n", label_num, $4, $6); output_line_number++;
                fprintf(yyout,"  bgez $at,SET0#%d\n", label_num); output_line_number++;
                fprintf(yyout,"SET1#%d:   li32 %s,0x00000001\n", label_num, $2); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"S1POS#%d:  bgez %s,S2POS#%d\n", label_num, $6, label_num);
output_line_number++;
                fprintf(yyout,"  bgez $zero,SET1#%d\n", label_num); output_line_number++;
                fprintf(yyout,"SET0#%d:   subu %s,$zero,$zero\n", label_num, $2);
output_line_number++;
               fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       SLL REG ',' REG ',' VALUE_5
                {
                fprintf(yyout,"  li32 $t3,0b000000000000000000000000000%s\n", $6);
output_line_number++;
                fprintf(yyout,"  andi32 $t3,$t3,0x0000001F\n"); output_line_number++;
                fprintf(yyout,"  addi32 $t3,$t3,0xFFFFFFFF\n"); output_line_number++;
                fprintf(yyout,"  subu %s,%s,$zero\n", $2, $4); output_line_number++;
                fprintf(yyout,"  bgez $t3,LBL1#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
```

```
                 fprintf(yyout,"LBL1#%d:   addu %s,%s,%s\n", label_num, $2, $2, $2);
output_line_number++;
                 fprintf(yyout,"  addi32 $t3,$t3,0xFFFFFFFF\n"); output_line_number++;
                 fprintf(yyout,"  bgez $zero, LBL1#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"END#%d:", label_num);
                label_num++;
                 }

        |        SRL REG ',' REG ',' VALUE_5
                 {
                 fprintf(yyout,"  li32 $t9,0b000000000000000000000000000%s\n", $6);
output_line_number++;
                 fprintf(yyout,"  andi32 $t9,$t9,0x0000001F\n"); output_line_number++;
                 fprintf(yyout,"  addi32 $t9,$t9,0xFFFFFFFF\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t9,NOTZERO#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  subu %s,%s,$zero\n", $2, $4); output_line_number++;
                 fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"NOTZERO#%d:        addi32 $t9,$t9,0xFFFFFFE2\n", label_num);
output_line_number++;
                 fprintf(yyout,"  subu $t9,$zero,$t9\n"); output_line_number++;
                 fprintf(yyout,"  subu $t8,%s,$zero\n", $4); output_line_number++;
                 fprintf(yyout,"  subu %s,$zero,$zero\n", $2); output_line_number++;
                 fprintf(yyout,"LBL1#%d:   addi32 $t9,$t9,0xFFFFFFFF\n", label_num);
output_line_number++;
                 fprintf(yyout,"  addu %s,%s,%s\n", $2, $2, $2); output_line_number++;
                 fprintf(yyout,"  bgez $t8,LBL2#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addi32 %s,%s,0x00000001\n", $2, $2); output_line_number++;
                 fprintf(yyout,"LBL2#%d:   addu $t8,$t8,$t8\n", label_num); output_line_number++;
                 fprintf(yyout,"  bgez $zero,LBL1#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"END#%d:", label_num);
                label_num++;
                 }

        |        SRA REG ',' REG ',' VALUE_5
                 {
                 fprintf(yyout,"  li32 $t4,0b000000000000000000000000000%s\n", $6);
output_line_number++;
                 fprintf(yyout,"  andi32 $t4,$t3,0x0000001F\n"); output_line_number++;
                 fprintf(yyout,"  addi32 $t4,$t4,0xFFFFFFFF\n"); output_line_number++;
                 fprintf(yyout,"  bgez $t4,NOTZERO#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  subu %s,%s,$zero\n", $2, $4); output_line_number++;
                 fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"NOTZERO#%d:        addi32 $t4,$t4,0xFFFFFFE2\n", label_num);
output_line_number++;
                 fprintf(yyout,"  subu $t4,$zero,$t4\n"); output_line_number++;
                 fprintf(yyout,"  subu $t3,%s,$zero\n", $4); output_line_number++;
                 fprintf(yyout,"  subu %s,$zero,$zero\n", $2); output_line_number++;
                 fprintf(yyout,"  bgez $t3,LBL1#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  li32 %s,0xFFFFFFFF\n", $2); output_line_number++;
                 fprintf(yyout,"LBL1#%d:   addi32 $t4,$t4,0xFFFFFFFF\n", label_num);
output_line_number++;
                 fprintf(yyout,"  addu %s,%s,%s\n", $2, $2, $2); output_line_number++;
                 fprintf(yyout,"  bgez $t3,LBL2#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  addi32 %s,%s,0x00000001\n", $2, $2); output_line_number++;
                 fprintf(yyout,"LBL2#%d:   addu $t3,$t3,$t3\n", label_num); output_line_number++;
                 fprintf(yyout,"  bgez $zero,LBL1#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"END#%d:", label_num);
                label_num++;
                 }

        |        SLLV REG ',' REG ',' REG
                 {
                 fprintf(yyout,"  andi32 $t8,%s,0x0000001F\n", $6); output_line_number++;
                 fprintf(yyout,"  addi32 $t8,$t8,0xFFFFFFFF\n"); output_line_number++;
                 fprintf(yyout,"  subu %s,%s,$zero\n", $2, $4); output_line_number++;
                 fprintf(yyout,"  bgez $t8,LBL1#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"LBL1#%d:   addu %s,%s,%s\n", label_num, $2, $2, $2);
output_line_number++;
                 fprintf(yyout,"  addi32 $t8, $t8,0xFFFFFFFF\n"); output_line_number++;
                 fprintf(yyout,"  bgez $zero,LBL1#%d\n", label_num); output_line_number++;
                 fprintf(yyout,"END#%d:", label_num);
                label_num++;
                 }
```

```
        |       SRLV REG ',' REG ',' REG
                {
                fprintf(yyout,"  andi32 $t9,%s,0x0000001F\n", $6); output_line_number++;
                fprintf(yyout,"  addi32 $t9,$t9,0xFFFFFFFF\n"); output_line_number++;
                fprintf(yyout,"  bgez $t9,NOTZERO#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  subu %s,%s,$zero\n", $2, $4); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"NOTZERO#%d:        addi32 $t9,$t9,0xFFFFFFE2\n", label_num);
output_line_number++;
                fprintf(yyout,"  subu $t9,$zero,$t9\n"); output_line_number++;
                fprintf(yyout,"  subu $t8,%s,$zero\n", $4); output_line_number++;
                fprintf(yyout,"  subu %s,$zero,$zero\n", $2); output_line_number++;
                fprintf(yyout,"LBL1#%d:   addi32 $t9,$t9,0xFFFFFFFF\n", label_num);
output_line_number++;
                fprintf(yyout,"  addu %s,%s,%s\n", $2, $2, $2); output_line_number++;
                fprintf(yyout,"  bgez $t8,LBL2#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  addi32 %s,%s,0x00000001\n", $2, $2); output_line_number++;
                fprintf(yyout,"LBL2#%d:   addu $t8,$t8,$t8\n", label_num); output_line_number++;
                fprintf(yyout,"  bgez $zero,LBL1#%d\n", label_num); output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       SRAV REG ',' REG ',' REG
                {
                fprintf(yyout,"  andi32 $t9,%s,0x0000001F\n", $6); output_line_number++;
                fprintf(yyout,"  addi32 $t9,$t9,0xFFFFFFFF\n"); output_line_number++;
                fprintf(yyout,"  bgez $t9,NOTZERO#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  subu %s,%s,$zero\n", $2, $4); output_line_number++;
                fprintf(yyout,"  bgez $zero,END#%d\n", label_num); output_line_number++;
                fprintf(yyout,"NOTZERO#%d:        addi32 $t9,$t9,0xFFFFFFE2\n", label_num);
output_line_number++;
                fprintf(yyout,"  subu $t9,$zero,$t9\n"); output_line_number++;
                fprintf(yyout,"  subu $t8,%s,$zero\n", $4); output_line_number++;
                fprintf(yyout,"  subu %s,$zero,$zero\n", $2); output_line_number++;
                fprintf(yyout,"  bgez $t8,LBL1#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  li32 %s,0xFFFFFFFF\n", $2); output_line_number++;
                fprintf(yyout,"LBL1#%d:   addi32 $t9,$t9,0xFFFFFFFF\n", label_num);
output_line_number++;
                fprintf(yyout,"  addu %s,%s,%s\n", $2, $2, $2); output_line_number++;
                fprintf(yyout,"  bgez $t8,LBL2#%d\n", label_num); output_line_number++;
                fprintf(yyout,"  addi32 %s,%s,0x00000001\n", $2, $2); output_line_number++;
                fprintf(yyout,"LBL2#%d:   addu $t8,$t8,$t8\n", label_num); output_line_number++;
                fprintf(yyout,"  bgez $zero,LBL1#%d\n", label_num); output_line_number++;
                fprintf(yyout,"END#%d:", label_num);
               label_num++;
                }

        |       NOP
                {
               if(compiler_pass != 9) {
                        fprintf(yyout,"  nop\n"); output_line_number++;
               }
               else {
                      fprintf(yyout,"00000000000000000000000000000000\n"); output_line_number++;
               }
                }

        |       DATA VALUE_32
                {
                if(compiler_pass != 9) {
                        fprintf(yyout,"  data 0b%s\n", $2); output_line_number++;
                }
                else {
                        fprintf(yyout,"%s\n", $2); output_line_number++;
                }
                }

        |       SYSCALL
                {
                if(compiler_pass != 9) {
                        fprintf(yyout,"  syscall\n"); output_line_number++;
                }
                else {
                        fprintf(yyout,"00000000000000000000000000001100\n"); output_line_number++;
```

```
                }
                }

        |       BREAK
                {
                if(compiler_pass != 9) {
                        fprintf(yyout,"  break\n"); output_line_number++;
                }
                else {
                        fprintf(yyout,"00000000000000000000000000001101\n"); output_line_number++;
                }
                }

        ;

label:          LABEL_DEST
                {
                fprintf(yyout,"%s", $1);
                }
        ;



%%

#include <unistd.h>

//These variables are used by both the lex
//and the yacc programs.

extern FILE *yyin;
extern FILE *yyout;
extern int line_number;
extern int error_cause;
extern int compiler_pass;
extern int num_instruction;
char *filename;

//Main goes through a total of 9 assembler passes.
//The first 4 passes perform instruction expansion,
//the 5th and 6th passes perform label offset
//computations, the 7th does data insertions for
//LI instructions, the 8th drops the labels and
//allows the data to become binary in form, and the
//9th pass maps to machine code.

//The command line argument is:
//      mps infile <outfile>
//where the infile is the input instruction language
//file and the outfile is the file you want the
//machine language outputted to.  If outfile is not
//specified, the machine language is written to
//assembler.out.

main(argc, argv)
int argc;
char **argv;
{

//Open the specified input file, if specified
//It will write to "assembler.out" if an input file is specified,
//otherwise it will output to the screen.

        if (argc > 1) {
                FILE *infile;

                infile = fopen(argv[1], "r");
                if (!infile) {
                        fprintf(stderr,"not a file: %s\n", argv[1]);
                        exit(1);
                }
                yyin = infile;
                yyout = fopen("assembler_temp1", "w");
                filename = argv[1];
        }
```

```
        else{
                filename = "stderr";
        }

//Parse the input (file) for assembler pass #

        do
        {
                yyparse();
        }
        while(!feof(yyin));

        fprintf(yyout,"  li32 $t3,0x00040000\n"); output_line_number++;
        fprintf(yyout,"  jalr $t3\n"); output_line_number++;

//handy debug lines

//        fprintf(yyout,"//Number of lines in input file: %d\n", line_number);
//        fprintf(yyout,"//Number of lines in temp file: %d\n", output_line_number);
//        fprintf(yyout,"//This was compiler pass %d\n", compiler_pass);

        compiler_pass++;
        line_number = 1;
        output_line_number = 0;
        fclose(yyout);


/*************************************************************
 *                                                           *
 * Set up the input file for assembler pass #2               *
 *                                                           *
 *************************************************************/

        FILE *infile2;

        infile2 = fopen("assembler_temp1", "r");
        if (!infile2) {
                fprintf(stderr,"not a file: %s\n", "assembler_temp1");
                exit(1);
        }
        yyin = infile2;
        yyout = fopen("assembler_temp2", "w");
        filename = "assembler_temp1";

//Parse the input (file) for assembler pass #2
        do
        {
                yyparse();
        }
        while(!feof(yyin));

//handy debug lines

//        fprintf(yyout,"//Number of lines in input file: %d\n", line_number);
//        fprintf(yyout,"//Number of lines in temp file: %d\n", output_line_number);
//       fprintf(yyout,"//This was compiler pass %d\n", compiler_pass);

        compiler_pass++;
        line_number = 1;
        output_line_number = 0;
        fclose(yyout);


/*************************************************************
 *                                                           *
 * Set up the input file for assembler pass #3               *
 *                                                           *
 *************************************************************/

        FILE *infile3;

        infile3 = fopen("assembler_temp2", "r");
        if (!infile3) {
                fprintf(stderr,"not a file: %s\n", "assembler_temp2");
```

```
                exit(1);
        }
        yyin = infile3;
        yyout = fopen("assembler_temp3", "w");
        filename = "assembler_temp2";

//Parse the input (file) for the third run through

        do
        {
                yyparse();
        }
        while(!feof(yyin));

//handy debug lines

//       fprintf(yyout,"//Number of lines in input file: %d\n", line_number);
//      fprintf(yyout,"//Number of lines in temp file: %d\n", output_line_number);
//      fprintf(yyout,"//This was compiler pass %d\n", compiler_pass);

        compiler_pass++;
        line_number = 1;
        output_line_number = 0;
        fclose(yyout);

/************************************************************
 *                                                          *
 * Set up the input file for assembler pass #4              *
 *                                                          *
 ************************************************************/

        FILE *infile4;

        infile4 = fopen("assembler_temp3", "r");
        if (!infile4) {
                fprintf(stderr,"not a file: %s\n", "assembler_temp3");
                exit(1);
        }
        yyin = infile4;
        yyout = fopen("assembler_temp4", "w");
        filename = "assembler_temp3";

//Parse the input (file) for the fourth run through

        do
        {
                yyparse();
        }
        while(!feof(yyin));

//handy debug lines

//       fprintf(yyout,"//Number of lines in input file: %d\n", line_number);
//       fprintf(yyout,"//Number of lines in temp file: %d\n", output_line_number);
//      fprintf(yyout,"//This was compiler pass %d\n", compiler_pass);

        compiler_pass++;
        line_number = 1;
       num_instructions = output_line_number;
        fprintf(yyout,"//Number of instructions recorded: %d\n", num_instructions);
        output_line_number = 0;
        fclose(yyout);

/************************************************************
 *                                                          *
 * Set up the input file for assembler pass #5              *
 *                                                          *
 ************************************************************/

       FILE *infile5;

       infile5 = fopen("assembler_temp4", "r");
       if (!infile5) {
               fprintf(stderr,"not a file: %s\n", "assembler_temp4");
               exit(1);
```

```
        }
        yyin = infile5;
        yyout = fopen("assembler_temp5", "w");
        filename = "assembler_temp4";

//Parse the input (file) for the fifth run through

        do
        {
                yyparse();
        }
        while(!feof(yyin));

//handy debug lines

//        fprintf(yyout,"//Number of lines in input file: %d\n", line_number);
//        fprintf(yyout,"//Number of lines in temp file: %d\n", output_line_number);
//        fprintf(yyout,"//This was compiler pass %d\n", compiler_pass);

        compiler_pass++;
        line_number = 1;
        output_line_number = 0;
        fclose(yyout);

/**************************************************************
 *                                                            *
 * Set up the input file for assembler pass #6                *
 *                                                            *
 **************************************************************/

        FILE *infile6;

        infile6 = fopen("assembler_temp5", "r");
        if (!infile6) {
                fprintf(stderr,"not a file: %s\n", "assembler_temp5");
                exit(1);
        }
        yyin = infile6;
        yyout = fopen("assembler_temp6", "w");
        filename = "assembler_temp5";

//Parse the input (file) for the sixth run through

        do
        {
                yyparse();
        }
        while(!feof(yyin));

//handy debug lines

//        fprintf(yyout,"//Number of lines in input file: %d\n", line_number);
//        fprintf(yyout,"//Number of lines in temp file: %d\n", output_line_number);
//        fprintf(yyout,"//This was compiler pass %d\n", compiler_pass);

        compiler_pass++;
        line_number = 1;
        output_line_number = 0;
        fclose(yyout);

/**************************************************************
 *                                                            *
 * Set up the input file for assembler pass #7                *
 *                                                            *
 **************************************************************/

        FILE *infile7;

        infile7 = fopen("assembler_temp6", "r");
        if (!infile7) {
                fprintf(stderr,"not a file: %s\n", "assembler_temp6");
                exit(1);
        }
        yyin = infile7;
        yyout = fopen("assembler_temp7", "w");
```

```
        filename = "assembler_temp6";

//Parse the input (file) for the seventh run through

        do
        {
                yyparse();
        }
        while(!feof(yyin));

        print_values();

//handy debug lines

//        fprintf(yyout,"//Number of lines in input file: %d\n", line_number);
//        fprintf(yyout,"//Number of lines in temp file: %d\n", output_line_number);
//        fprintf(yyout,"//This was compiler pass %d\n", compiler_pass);

        compiler_pass++;
        line_number = 1;
        output_line_number = 0;
        fclose(yyout);


/****************************************************************
 *                                                              *
 * Set up the input file for assembler pass #8                  *
 *                                                              *
 ****************************************************************/

        FILE *infile8;

        infile8 = fopen("assembler_temp7", "r");
        if (!infile8) {
                fprintf(stderr,"not a file: %s\n", "assembler_temp7");
                exit(1);
        }
        yyin = infile8;
        yyout = fopen("assembler_temp8", "w");
        filename = "assembler_temp7";

//Parse the input (file) for the eighth run through

        do
        {
                yyparse();
        }
        while(!feof(yyin));

//handy debug lines

//        fprintf(yyout,"//Number of lines in input file: %d\n", line_number);
//        fprintf(yyout,"//Number of lines in temp file: %d\n", output_line_number);
//        fprintf(yyout,"//This was compiler pass %d\n", compiler_pass);

        compiler_pass++;
        line_number = 1;
        output_line_number = 0;
        fclose(yyout);



/****************************************************************
 *                                                              *
 * If an output file is specified, set it up here.  This will  *
 * be assembler pass #9, the last one.                          *
 *                                                              *
 ****************************************************************/
        if (argc > 2) {
                FILE *outfile;

                outfile = fopen(argv[2], "w");
                if (!outfile) {
                        fprintf(stderr,"opening file failed\n");
```

```
                    exit(1);
            }
            yyout = outfile;
        }
        else {
            yyout = fopen("assembler.out", "w");
        }


        yyin = fopen("assembler_temp8", "r");
        filename = "assembler_temp8";

        do
        {
                yyparse();
        }
        while(!feof(yyin));

//handy debug lines

//      fprintf(yyout,"//Number of lines in temp file: %d\n", line_number);
//      fprintf(yyout,"//Number of lines in output file: %d\n", output_line_number);
//      fprintf(yyout,"//This was compiler pass %d\n", compiler_pass);

        compiler_pass++;
        line_number = 1;
        output_line_number = 0;
        fclose(yyout);

//Remove all the temporary files

        unlink("assembler_temp1");
        unlink("assembler_temp2");
        unlink("assembler_temp3");
        unlink("assembler_temp4");
        unlink("assembler_temp5");
        unlink("assembler_temp6");
        unlink("assembler_temp7");
        unlink("assembler_temp8");

}




//This is the error function.  It uses the integer "error_cause"
//to determine which error message to give to the user, and then
//reports the error along with the line number that caused the
//error.

yyerror(s)
char *s;
{
        if(error_cause == 3)
        {
                fprintf(stderr,"\n***ERROR*** on line %d of %s: the label %s was unmatched.\n",
line_number, filename, yylval.sval);
                fprintf(stderr,"Program will terminate.\n\n");
                error_cause = 0;
                exit(1);
        }
        else if(error_cause == 2)
        {
                fprintf(stderr,"\n***ERROR*** on line %d of %s: register %s\n", line_number,
filename, yylval.sval);
                fprintf(stderr,"is reserved for the compiler\n\n");
                fprintf(stderr,"Program will terminate.\n\n");
                error_cause = 0;
                exit(1);
        }
        else if(error_cause == 1)
        {
                fprintf(stderr,"\n***ERROR*** on line %d of %s: string beginning\n", line_number,
filename);
                fprintf(stderr,"    with %s is unrecognized.\n", yylval.sval);
```

```
                fprintf(stderr,"Program will terminate.\n\n");
                error_cause = 0;
                exit(1);
        }
        else
        {
                fprintf(stderr,"\n***ERROR*** on line %d of %s: the token %s makes\n", line_number,
filename, yylval.sval);
                fprintf(stderr,"    this instruction illegal.\n");
                fprintf(stderr,"Program will terminate.\n\n");
                exit(1);
        }
}
```