Lab No. 7
Casey T. Morrison
EEL 4713 Section 2485 (Spring 2004)
Lab Meeting Date and Time: Monday E1-E3
TA: Grzegorz Cieslewski

I have performed this assignment myself. I have performed this work in accordance with the Lab Rules specifies in 4713 Lab No. 0 and the University of Florida's Academic Honesty manual. On my honor, I have neither given nor received unauthorized aid in doing this assignment.

_____

Introduction

The purpose of this lab was to study the *MIPS2000* Reduced Instruction Set Computer (RISC). The first step in this complex assignment was to prepare an Executable Problem Statement (EPS) that will allow the project developer to understand the details of the problem. In this case, the Instruction Set Architecture (ISA) of the *MIPS2000* must be studied in order to understand the implications of reducing the instruction set and address modes from those presented in the Complex Instruction Set Computer (CISC) *Sweet16*. The EPS will take the form of a *MIPS2000* instruction set simulator that is similar to the one used for the *Sweet16*.

Since only the Instruction Set Architecture of the *MIPS2000* is being examined, the C program *mipssim.c* will simulate only the "high-level" behavior of the architecture. This will come in the form of a Harvard architecture, which implies separate program and data memories with their own address and data buses. In addition, The *MIPS2000* processor supports a "Load/Store" architecture, which means that only Load and Store instructions have access to data memory outside of the CPU.

Component Design and Validation

There are three basic instruction formats in the *MIPS2000* processor: R-Type, I-Type, and J-Type. The instruction set for this processor is therefore divided into three groups based on the instruction format. The *MIPS2000* simulator developed in this lab, *mipssim.c*, was designed around the concept that instructions belonging to the same group have similar implementations.

A. R-Type Instruction Implementation

All R-Type instructions have the format shown in Figure 1 below.

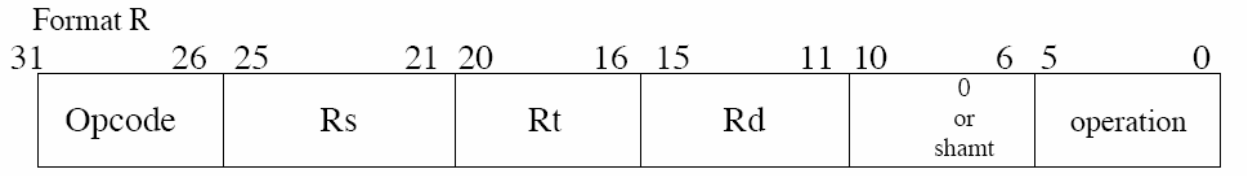

Figure 1[1]**:** R-Type instruction format

As with all *MIPS2000* instructions, the R-Type instructions are 32-bits wide. This format is ideal for logic and arithmetic operations—the type of operations that make up the bulk of the R-Type instructions. As illustrated in Figure 1, R-Type instructions can specify two source registers and one destination register. However, some R-Type instructions only specify one source register (for example, the shift instructions).

[1] "lab7.pdf" by Michel A. Lynch and Matthew Radlinski. http://www.hcs.ufl.edu/~radlinsk/eel4713/

The opcode for all R-Type instructions is 0x00. This is because the only real difference between the executions of one R-Type instruction versus another is the ALU function. The 6-bit "operation" field of each instruction corresponds to the function select of the *MIPS2000* ALU. Thus all R-Type instructions can have the same opcode because they all operate on data contained in registers, and they all alter the contents of a single destination register as a result of the specific ALU function performed.

The simulator developed in this lab effectively modeled the operation of a general microprocessor in that its main program consisted of a fetch/decode/execute cycle. The fetch portion of the cycle involved retrieving four Bytes from memory, and incrementing the Program Counter to point at the next instruction word. Since the smallest addressable unit of memory is the Byte, retrieving a full 32-bit instruction word required this procedure of four consecutive Byte fetches. The code for instruction fetch is shown below in Listing 1.

```
/* Instruction Fetch                                */
ir.db.upper_byte   = Pmem[prog_cntr]; incPC;
ir.db.upmid_byte   = Pmem[prog_cntr]; incPC;
ir.db.lowmid_byte  = Pmem[prog_cntr]; incPC;
ir.db.lower_byte   = Pmem[prog_cntr]; incPC;
print_ir();
```

Listing 1: Instruction fetch code

The next part of the cycle involves decoding the instruction. This is accomplished in the simulator via a "switch" statement that examines the opcode of each instruction. Since all R-Type instructions have the same opcode (0x00), a special switch statement, nested within the first, was used to direct the program flow to the proper location. A sample of this program structure is shown in Listing 2 below.

```
/* Instruction Decoder                    */
switch((unsigned)ir.R_instr.opcode){

// R-Format Instructions
     case SPECIAL:
          switch((unsigned)ir.R_instr.func){
          case ADD:
               printf("ADD:\n");
               // REGS[RD] := REGS[RS] + REGS[RT]
               reg[ir.R_instr.rd] = reg[ir.R_instr.rs1] +
                                    reg[ir.R_instr.rs2];
               break;
          case ADDU:
               printf("ADDU:\n");
               // REGS[RD] := REGS[RS] + REGS[RT]
               reg[ir.R_instr.rd] = reg[ir.R_instr.rs1] +
                                    reg[ir.R_instr.rs2];
               break;
          case SUB:
               printf("SUB:\n");
               // REGS[RD] := REGS[RS] - REGS[RT]
               reg[ir.R_instr.rd] = reg[ir.R_instr.rs1] -
                                    reg[ir.R_instr.rs2];
               break;
```

Listing 2: Instruction Decode stage

From this program segment the simplicity of the R-Type instructions is evident. Most of these instructions involve performing some operation on the contents of two registers and storing the results in a third register.

Shifting is slightly more complicated since some shifts (arithmetic shifts in particular) require that data be sign-extended. This was accomplished with the use of the bitwise shift operator in C. With the data temporarily treated as a signed quantity, the left and right shift operators (<< and >>, respectively) automatically preserve the sign. Listing 3 below shows a portion of the code developed to handle such situations.

```
case SRA:
        printf("SRA:\n");
        // REGS[RD] := Sign_Extended(REGS[RT] >> SHAMT)
        printf("   shamt: %02d\n",(int)ir.R_instr.shamt);
        if((reg[ir.R_instr.rs2] & SIGN_MASK) == SIGN_MASK) {
                temp = reg[ir.R_instr.rs2];
                temp = temp >> ir.R_instr.shamt;
                reg[ir.R_instr.rd] = temp;
        }
        else {
                reg[ir.R_instr.rd] = reg[ir.R_instr.rs2] >> ir.R_instr.shamt;
        }
        break;
case SRAV:
        printf("SRAV:\n");
        // REGS[RD] := Sign_Extended(REGS[RT] >> REGS[RS])
        printf("   shamt: %02d\n",(int)reg[ir.R_instr.rs1]);
        if((reg[ir.R_instr.rs2] & SIGN_MASK) == SIGN_MASK) {
                temp = reg[ir.R_instr.rs2];
                temp = temp >> ir.R_instr.rs1;
                reg[ir.R_instr.rd] = temp;
        }
        else {
                reg[ir.R_instr.rd] = reg[ir.R_instr.rs2] >> reg[ir.R_instr.rs1];
        }
        break;
```

Listing 3: Shift operations

One R-Type instruction that may not immediately appear to be an R-Type is the *jump register* instruction. This instruction alters the program flow by setting the Program Counter to the address contained in a specified register. This instruction, along with the similar *JALR* instruction, was relatively simple to implement and is shown below in Listing 4.

```
case JR:
        printf("JR:\n"); // PC := REGS[return_address_register];
        prog_cntr = reg[RA];
        break;
case JALR:
        printf("JALR:\n");
        // Unconditionally jump to the instruction whose address is in register rs.
        // Save the address of the next instruction in register rd (default: R31).
        if(ir.R_instr.rd == 0) {
                // RD not specified (zero). Store RA in default register (R31).
                reg[RA] = prog_cntr;
        }
        else {
                reg[ir.R_instr.rd] = prog_cntr;
        }
        prog_cntr = reg[ir.R_instr.rs1];
        break;
```

Listing 4: Jump operations

B. I-Type Instruction Implementation

All I-Type instructions have the format shown in Figure 2 below.

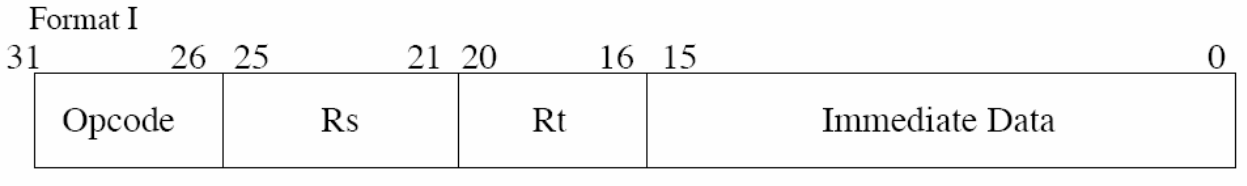

Figure 2[2]: I-Type instruction format

This format is ideal for logic and arithmetic operations involving immediate data as well as various load and store instructions. As illustrated in Figure 2, I-Type instructions can specify two sources of data, one being a register and the other being 16-bit immediate data. As with the R-Type instructions, there is also a destination register specified. The "immediate data" field of the instruction is sometimes interpreted as a 16-bit offset, depending on the nature of the instruction.

Implementing the arithmetic and logic I-Type instructions was very much like implementing their R-Type counterparts. The only difference is that the second source of data is a 16-bit immediate value instead of a 32-bit register value. Listing 5 below shows examples of these I-Type instructions.

```
case ANDI:
        printf("ANDI:\n");
        // REGS[RD] := REGS[RS] & IMMEDIATE_DATA
        reg[ir.I_instr.rd] = reg[ir.I_instr.rs1] & ir.I_instr.immd;
        break;
case ORI:
        printf("ORI:\n");
        // REGS[RD] := REGS[RS] | IMMEDIATE_DATA
        reg[ir.I_instr.rd] = reg[ir.I_instr.rs1] | ir.I_instr.immd;
        break;
case XORI:
        printf("XORI:\n");
        // REGS[RD] := REGS[RS] ^ IMMEDIATE_DATA
        reg[ir.I_instr.rd] = reg[ir.I_instr.rs1] ^ ir.I_instr.immd;
        break;
```

Listing 5: I-Type instructions

Another category of I-Type instructions are the PC-Relative branch instructions. These instructions test a specific condition and alter the program flow based on that condition. For example, the *BLEZ RS, label* instruction causes the program flow to jump to *label* if it is the case that the contents of register *RS* are less than or equal to zero.

Branch instructions use a signed 16-bit instruction *offset* field; hence they can jump $2^{15} - 1$ instructions (not bytes) forward or 2^{15} instructions backwards. This differs from *jump* instructions in that *jump* instructions have a 26-bit address field and can therefore jump farther forward and backwards.

[2] "lab7.pdf" by Michel A. Lynch and Matthew Radlinski. http://www.hcs.ufl.edu/~radlinsk/eel4713/

A portion of the code for some branch instructions is shown below in Listing 6.

```
void branch()
{
       // PC-relative branch
       printf("   Branch taken\n");
       if((ir.I_instr.immd & IMMED_SIGN_MASK) == IMMED_SIGN_MASK) {
                     printf("      Negative offset: -%04X\n",
                            ((~(ir.I_instr.immd) + 0x0001) << 2));
              prog_cntr = prog_cntr - ((~(ir.I_instr.immd) + 0x0001) << 2);
       }
       else {
              printf("      Positive offset: %04X\n", (ir.I_instr.immd << 2));
              prog_cntr = prog_cntr + (ir.I_instr.immd << 2);
       }
}

// more code in between

case BEQ:
       printf("BEQ:\n");
       // if(REGS[RS] == REGS[RT]) {PC := PC + OFFSET;)
       // else {PC := PC + 4;}
       if(reg[ir.I_instr.rs1] == reg[ir.I_instr.rd]) {
              branch();
       }
       else {
              printf("   Branch not taken");
       }
       break;
case BLEZ:
       printf("BLEZ:\n");
       // if(REGS[RS] <= 0]) {PC := PC + OFFSET;)
       // else {PC := PC + 4;}
       if(((reg[ir.I_instr.rs1] & SIGN_MASK) == SIGN_MASK) |
          (reg[ir.I_instr.rs1] == ZERO_MASK)) {
              branch();
       }
       else {
              printf("   Branch not taken");
       }
       break;
```

Listing 6: Branch instructions

As shown in Listing 6 above, all the branch instructions rely on the *branch()* method to alter the program flow according to the 16-bit signed offset imbedded in the instruction.

The third category of I-Type instructions consists of load and store instructions. These instructions either read from or write to the Data Memory. Since *MIPS2000* is a 32-bit processor, many loads and stores will involve 32-bit data addressed with a word-aligned address. However, since it is very common for processors to deal with 8-bit Bytes (for character manipulation, etc.), loads and stores can also deal with variable-length data flows.

Sign extension also comes into play when dealing with loads. When loading a signed 8-bit (or 16- or 24-bit) quantity, it is necessary to preserve the sign of that data when it is transferred to a 32-bit register. This requirement is handled with the use of "if" statements that determine whether sign extension is necessary during a particular load.

Since the smallest addressable unit of memory is a Byte, loads and stores need not occur at word-aligned addresses. However, it should be noted that it is not possible to load (or store) a word across a word-aligned boundary in *one* instruction (although several instructions could be used together to accomplish this task).

The most important aspect of all load and store instructions is that they all utilize index-with-offset addressing. That is to say that each load and store instruction specifies an offset (stored in a register) to the memory location indicated by the *label* in the assembly language instruction. For example, the instruction *LB RT,RS, address* loads register *RT* with the Byte at memory location (*address + [RS]).* A portion of the code for some load and store instructions is shown below in Listing 7.

```
case LB:
       printf("LB:\n");
       // REGS[RT] := Dmem[REGS[RS] + OFFSET]
       if((ir.I_instr.immd & IMMED_SIGN_MASK) == IMMED_SIGN_MASK) {
              printf("   Negative offset: -%04X\n", (~(ir.I_instr.immd) +
                     0x0001));
              address = reg[ir.I_instr.rs1] - (~(ir.I_instr.immd) + 1);
       }
       else {
              printf("   Positive offset: %04X\n", (ir.I_instr.immd));
              address = reg[ir.I_instr.rs1] + ir.I_instr.immd;
       }
       temp = Dmem[address] & 0X000000FF;
       if((temp & BYTE_SIGN_MASK) == BYTE_SIGN_MASK) {
              // byte being loaded is negative (must sign extend)
              reg[ir.I_instr.rd] = (temp | 0XFFFFFF00) & WORD_MASK;
       }
       else {
              // byte being loaded is positive
              reg[ir.I_instr.rd] = temp & WORD_MASK;
       }
       break;
case SH:
       printf("SH:\n");
       // Dmem[REGS[RS] + OFFSET] := REGS[RT]
       if((ir.I_instr.immd & IMMED_SIGN_MASK) == IMMED_SIGN_MASK) {
              printf("   Negative offset: -%04X\n", (~(ir.I_instr.immd) +
                     0x0001));
              address = reg[ir.I_instr.rs1] - (~(ir.I_instr.immd) + 1);
       }
       else {
              printf("   Positive offset: %04X\n", (ir.I_instr.immd));
              address = reg[ir.I_instr.rs1] + ir.I_instr.immd;
       }
       Dmem[address] = reg[ir.I_instr.rd] & 0X0000FF00;
       Dmem[address + 1] = reg[ir.I_instr.rd] & BYTE_MASK;
       print_Dmem();
       break;
```

Listing 7: Load and store instructions

Since the *MIPS2000* processor supports a “Load/Store” architecture, it is only with the I-Type Load and Store instructions that Data Memory may be accessed.

C. J-Type Instruction Implementation

All J-Type instructions have the format shown in Figure 3 below.

Figure 3[3]**:** J-Type instruction format

This format enables program *jumps* that alter the program flow with a greater range than the branch instructions. As illustrated in Figure 3, J-Type instructions have a 26-bit target address. This target address always specifies a memory location that is on a word-aligned boundary. Therefore, the actual target address is four times the 26-bit quantity specified in the "target" field (or "Destination Address / 4" field).

Because each target address, after being shifted to accommodate for the word-alignment, still only specifies 28 bits of the 32-bit address, a paging scheme is used to discern the exact jump address. The 4 GB memory space of the *MIPS2000* is divided into sixteen 256 MB "pages." Each page is specified by the most-significant nibble of the address. When a *jump* instruction is encountered in a program, the 28-bit word-aligned address specified by the "target" field of the instruction is concatenated with the most-significant nibble of the Program Counter. This essentially means that the programmer can jump to any location within the current 256 MB page. Jumping across page boundaries is not explicitly supported but can be accomplished in software.

The code for the two J-Type jump instructions is shown below in Listing 8.

```
// J_type instructions
        case J:
                printf("J:\n");
                // PC := TARGET
                printf("Jump within page: %01X\n", (prog_cntr >> 28));
                prog_cntr = ((ir.J_instr.offset << 2) & WORD_MASK) | (prog_cntr &
                            UPPER_DIGIT_MASK);
                printf("Target address: %08X\n", prog_cntr);
                break;
        case JAL:
                printf("JAL:\n");
                // PC := TARGET
                // REGS[return_address_register] := RETURN_ADDRESS
                reg[RA] = prog_cntr;
                printf("   Jump within page: %01X\n", (prog_cntr >> 28));
                prog_cntr = ((ir.J_instr.offset << 2) & WORD_MASK) | (prog_cntr &
                            UPPER_DIGIT_MASK);
                printf("   Target address: %08X\n", prog_cntr);
                printf("   Return address: %08X\n", reg[RA]);
                break;
```

Listing 8: Jump instructions

[3] "lab7.pdf" by Michel A. Lynch and Matthew Radlinski. http://www.hcs.ufl.edu/~radlinsk/eel4713/

The complete code for the *MIPS2000* simulator, *mipssim.c*, may be found in Appendix A, Program 4. The entire code for the *mipssim.h* file may be found in Appendix A, Program 5.

System Design and Validation

Once the code for the *MIPS2000* simulator was written, two test programs were designed to verify the functionality of the simulator. In order to be thorough, almost every instruction implemented in the simulator was explicitly utilized in the test programs.

A. System Test 1: mips_test.asm

A special test program was written to isolate and test a majority of the *MIPS2000* instructions implemented in the simulator (save for the store instructions). Although the resulting assembly program had no obvious function, it allowed for the examination of each instruction. Appendix C, Program 24 lists the code for the *mips_test.asm* program. Program 23 in the same appendix contains the macro file that facilitates the assembly of the *mips_test.asm* and other *MIPS2000* assembly programs.

This test program was assembled with *UPASM* and simulated with the *MIPS2000* simulator created in this lab. See Appendix D, Simulation 4 for the complete simulation results. After analyzing the simulation output, it was determined that all the instructions tested executed as designed.

B. System Test 2: mips_test_store.asm

Another test program was written to isolate and test the *MIPS2000* store instructions implemented in the simulator. Listing 9 on the next page shows the resulting assembly program. (also see Appendix C, Program 25 for the code for the *mips_test_store.asm* program).

This test program was assembled with *UPASM* and simulated with the *MIPS2000* simulator created in this lab. See Appendix D, Simulation 5 for the complete simulation results. After examining the simulation output, it was determined that the store instructions performed as desired.

```
* MIPS_TEST_STORE.ASM - Program that tests the operation of the MIPS
*                       simulator
*                    Orged in ROM ($0000)
* Author: Casey T. Morrison, EEL 4713, 3/20/2004

        nolist
        include "mips.mac"
        list

        ORG $0000

        li    R20,$1234
        sll   R20,R20,16
        ori   R20,R20,$5678
        li    R21,$F00D
        sll   R21,R21,16
        ori   R21,R21,$BEEF
        jal   sub1
        j     done

*****************************************************
* Subroutine 1: Test store instructions             *
*****************************************************

sub1:   sb    R20,R0,store1
        li    R14,$0001
        swl   R20,R14,store1
        sw    R21,R0,store2

exit1:  jr    R31

done:   GFO

* DATA ***********************
store1: ds.b  4
store2: ds.b  4

        end
```

Listing 9: *MIPS_Test_Store.asm* code

Conclusion

A. Summary

The differences between the *MIPS2000* RISC machine and the *Sweet16* CISC machine are more apparent now that both have been simulated in C. The most marked distinction is in the instruction format and length. While the *Sweet16* supported variable-length instructions of multiple addressing modes, the *MIPS2000* features fixed-length instructions with a limited number of addressing modes.

The advantages of fixed-length instructions are very much apparent even after limited exposure to the *MIPS2000* processor. The logic involved in executing each instruction was drastically reduced with the presumption that every instruction was no more than 32-bits long. In addition, the limited number of addressing modes allowed for less variation in the actual execution of the instructions.

In designing a simulator, practical knowledge of the fundamental operation of each instruction was acquired. While the hardware necessary to implement these instructions was not explored at all, the overall function of the processor was replicated and, in the process, more accurately understood.

Lab No. 8
Casey T. Morrison
EEL 4713 Section 2485 (Spring 2004)
Lab Meeting Date and Time: Monday E1-E3
TA: Grzegorz Cieslewski

I have performed this assignment myself. I have performed this work in accordance with the Lab Rules specifies in 4713 Lab No. 0 and the University of Florida's Academic Honesty manual. On my honor, I have neither given nor received unauthorized aid in doing this assignment.

_____

Introduction

The purpose of this lab was to study and construct the *MIPS2000* general architecture. This involved implementing a five-stage pipeline that supported 43 instructions. The reduced instruction set nature of *MIPS2000* greatly simplified the architecture and allowed for a relatively high throughput. The multi-stage pipeline design, however, increased the complexity of the data flow as well as the propensity for data dependencies and other hazardous situations.

In contrast to the *Sweet16*, the *MIPS2000* microprocessor has a fixed-length instruction word and a 32-bit Address/Data bus. Since the assumption in building the *MIPS2000* processor was that memory is fast and large, it is not an encumbrance to repeatedly fetch 32-bit instructions from memory. Thus the *MIPS2000* was designed to execute only a handful of instruction, but it was also optimized to potentially execute instructions at a rate of one per clock cycle. This is the essence behind the *MIPS2000* microprocessor—that by keeping the instruction set relatively small, the throughput can be increased greatly provided that memory is sufficiently fast.

Component Design and Validation

As mention previously, the *MIPS2000* utilizes a five-stage pipeline to decrease the Cycles Per Instruction (CPI) of the processor. The five stages are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). In between each stage is a pipeline register that stores the status of the previous stage's control and data signals. This structure allows multiple instructions to be executing at once in the architecture.

The five stages of the *MIPS2000* were designed separately according to their individual functions and requirements. In addition, the four pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB) were designed individually such that they may store whatever data is needed for the next stage(s).

A. Instruction Fetch Stage

During the Instruction Fetch (IF) stage, the first stage of the *MIPS2000* pipelined processor, there is a Program Counter (PC) that addresses the Program Memory in order to retrieve the next instruction for execution. Despite its name, the program counter is not a counter so much as it is a register. The input of the PC comes from another stage of the pipeline (the MEM stage) and determines the location of the next instruction to execute. In anticipation of sequential instruction execution, the IF stage automatically calculates the *PCplus4* address by adding 4 to the current PC value and storing the results in a register. The instruction word obtained from the Program Memory if formatted according Figure 1 on the next page depending on the type of instruction. The various fields of the instruction word are saved in the IF/ID Pipeline Register so that they may be propagated through the architecture.

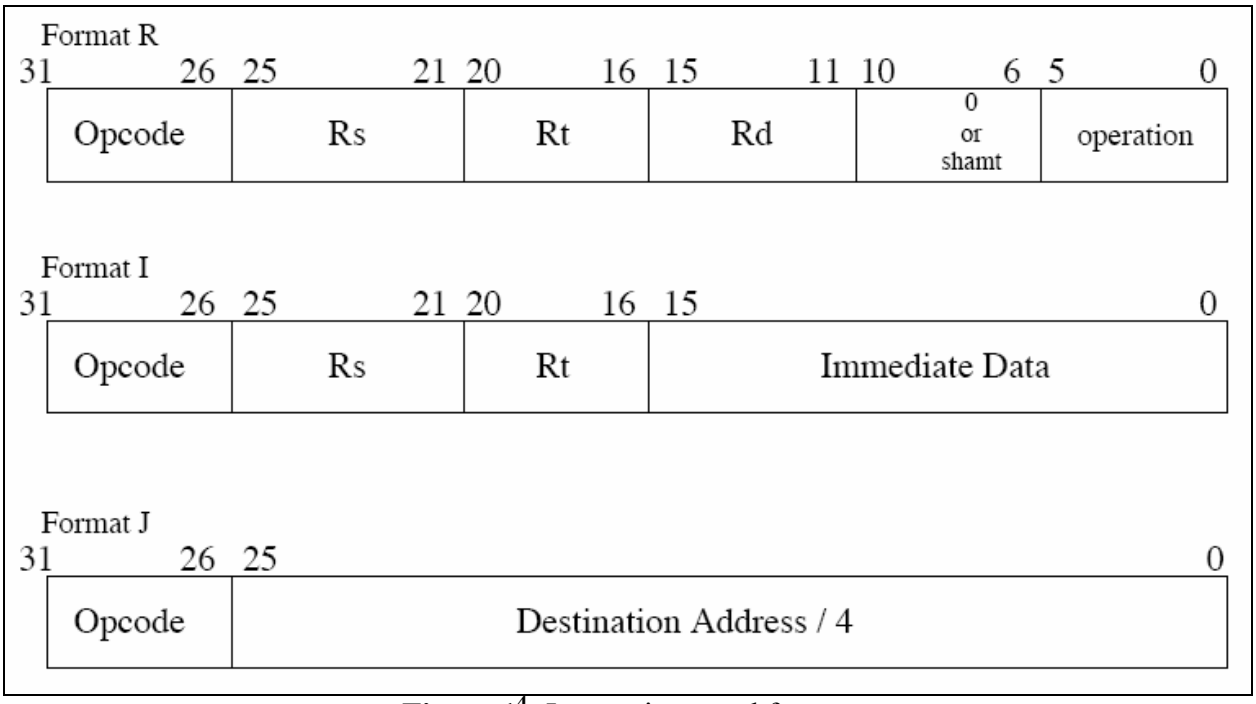

Figure 1[4]**:** Instruction word format

The *MIPS2000* Instruction Fetch stage was designed according to the schematic in Figure 2.

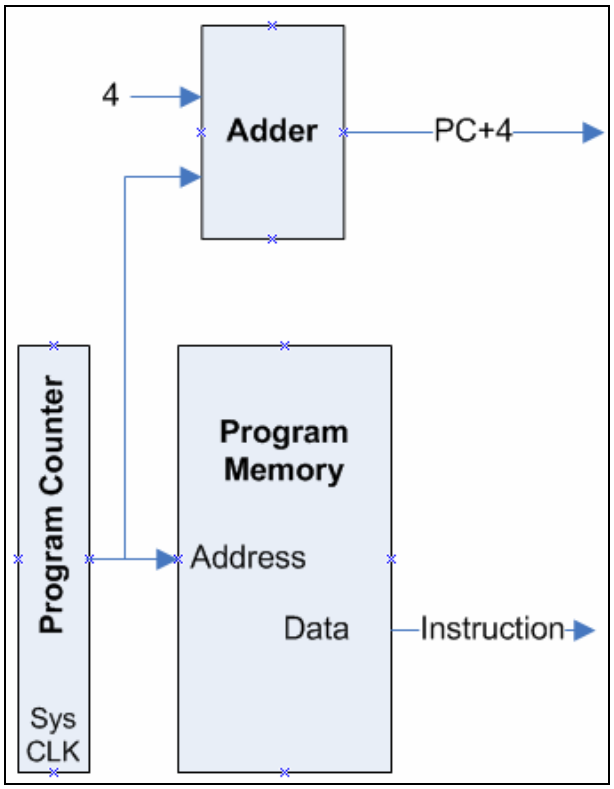

Figure 2: Instruction Fetch Stage

[4] "lab8.pdf" by Michel A. Lynch and Matthew Radlinski. http://www.hcs.ufl.edu/~radlinsk/eel4713/

The first component, the Program Counter, was created in VHDL. It was designed to be a 32-bit register with synchronous *clear* and *hold* signals that will be utilized in future labs. See Appendix F, Specification Sheet 11 for a complete description of this component. See Appendix B, Component 10 for the VHDL code for *Prog_Cntr*.

The adder used to compute the *PCplus4* value was also created in VHDL. It utilized the *ieee.std_logic_arith* library to perform the addition of the current PC value and the constant number four. See Appendix F, Specification Sheet 12 for a complete description of this component. See Appendix B, Component 11 for the VHDL code for *PC_Inc*.

The final component in the Instruction Fetch stage is the Program Memory. This asynchronous Read Only Memory (ROM) device consists of four 4Kx8 *LPM_ROM* components from the Max+Plus II *mega_lpm* library. These four ROMs comprise the upper-byte, upper-middle-byte, lower-middle-byte, and lower-byte of the instruction word. In order to combine these four ROMs into one 4Kx32 ROM, bits 13 through two of the *MIPS2000* address bus are used to address each ROM. Thus every address between word-aligned boundaries actually addresses the same location in each ROM. Since every instruction is word-aligned, there is never a need to access any Program Memory address other than a word aligned address (an address ending in 0x0, 0x4, 0x8, or 0xC). See Appendix F, Specification Sheet 13 for a complete description of the *MIPS2000* Program Memory.

These components were graphically combined, and the result is shown in Figures 3a and 3b below. See Appendix F, Specification Sheet 14 for a complete description of the *IF_stage* component.

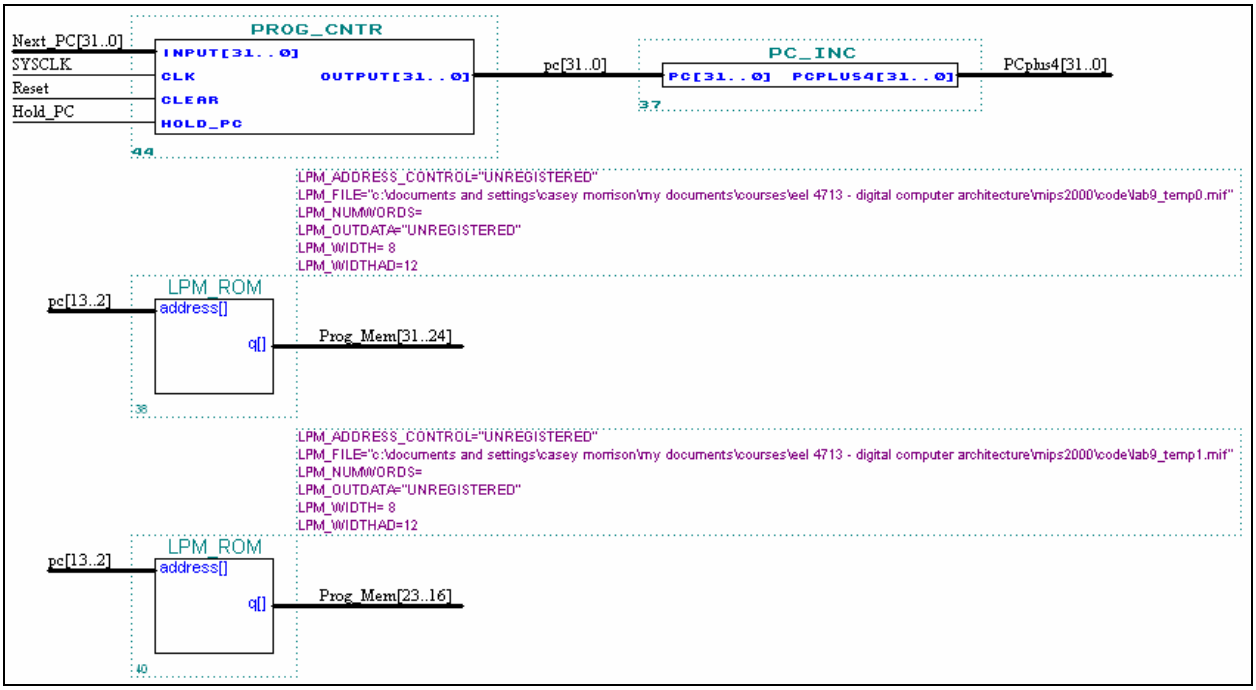

Figure 3a: Instruction Fetch stage

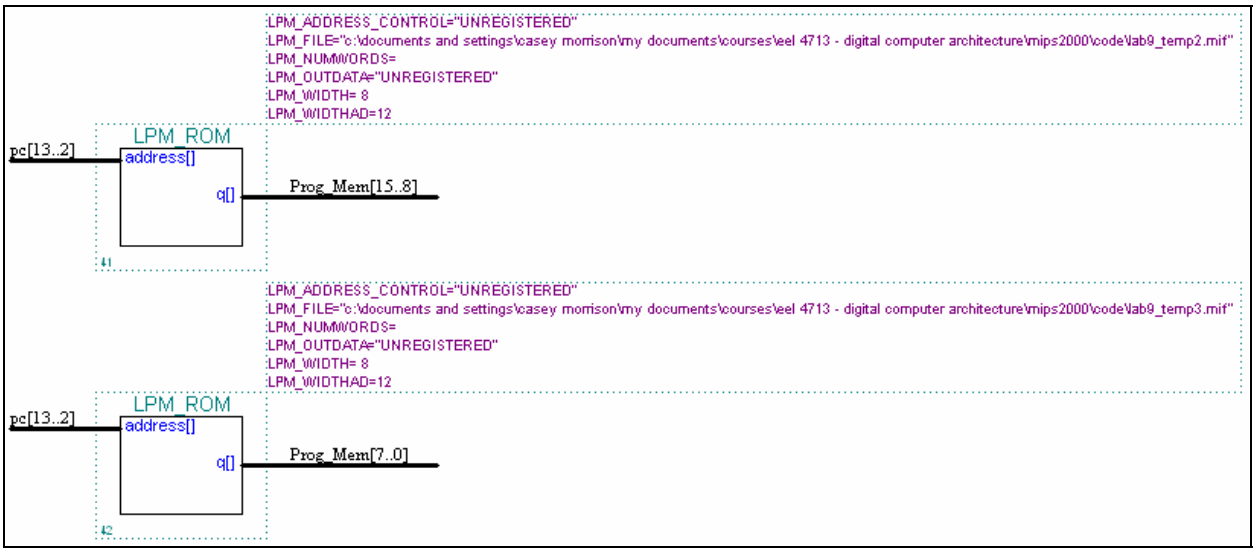

Figure 3b: Instruction Fetch stage

B. Instruction Decode Stage

The Instruction Decode (ID) stage of the pipeline is responsible for determining and preparing the operands for execution in the subsequent stage. This stage of the pipeline was designed based on the schematic shown in Figure 4 below.

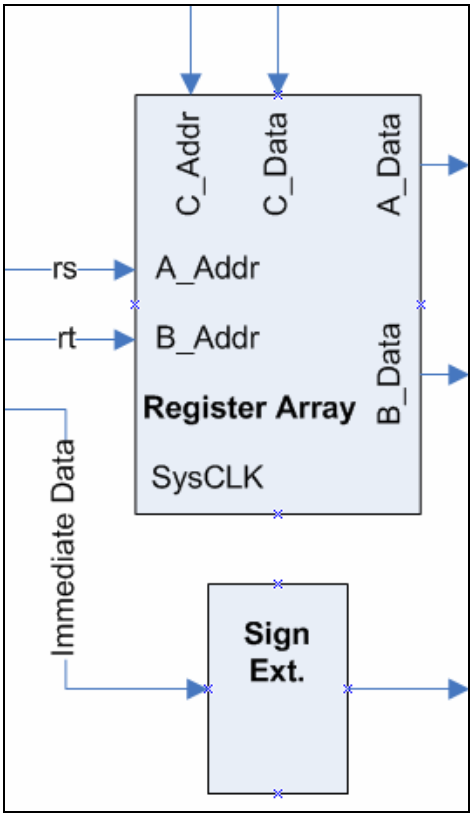

Figure 4: Instruction Decode Stage

The first component in the Instruction Decode stage, the 32x32 Register Array, was designed in VHDL as a dual-output array of 32-bit registers. Like the Register Array used in the *Sweet16*, the *MIPS2000* Register Array supports the addressing of three registers (one input and two outputs) at any given time. For reasons that will be discussed in future labs, this component was designed such that a read and a write could take place in a single clock cycle. In the event of a read and write from/to the same register, the VHDL code specifies that the data to be written is also passed through to the output. See Appendix F, Specification Sheet 15 for a complete description of the *MIPS2000* Register Array. See Appendix B, Component 12 for the VHDL code for *Reg_Array_32x32*.

The second component in the ID stage is the Sign Extender. This VHDL component conditionally sign extends the immediate data from the Instruction Register. The only instructions for which this component does not sign extend the immediate data are the logical immediate instructions (*andi*, *ori*, and *xori*). See Appendix F, Specification Sheet 16 for a complete description of the Sign Extender. See Appendix B, Component 13 for the VHDL code for *Sign_Extender*.

These components were graphically combined, and the result is shown in Figure 5 below. See Appendix F, Specification Sheet 17 for a complete description of the *ID_stage* component.

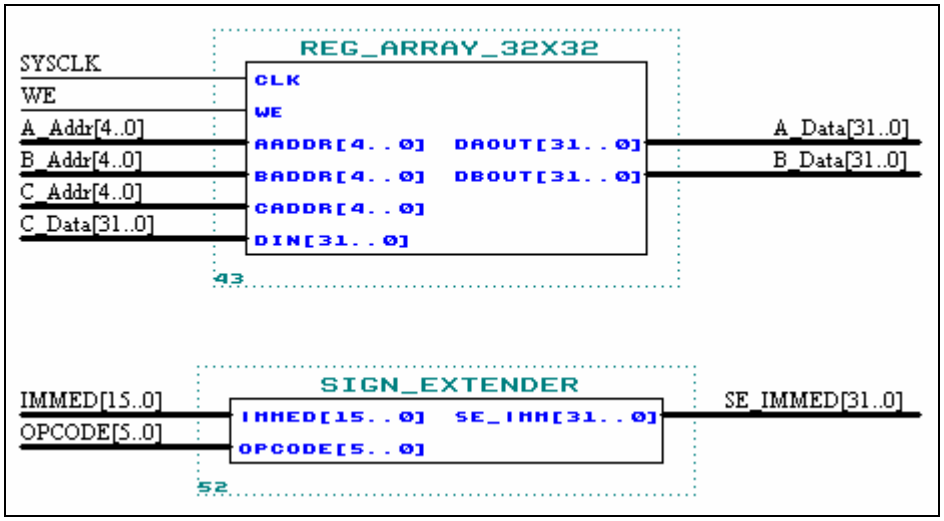

Figure 5: Instruction Decode Stage

C. Execute Stage

The Execute (EX) stage of the pipeline is responsible for making all necessary calculations—whether they are for the purposes of computing a result, an address, or a condition. This stage of the pipeline was designed based on the schematic shown in Figure 6 on the next page.

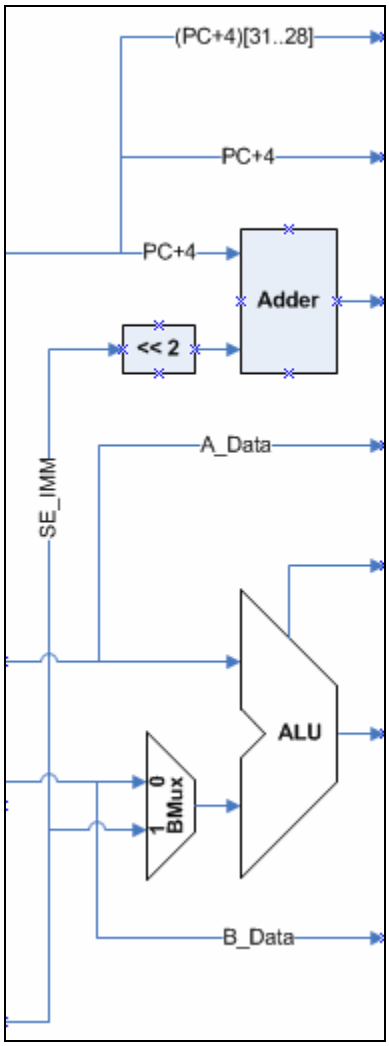

Figure 6: Execute Stage

The first and most prominent component of this stage is the Arithmetic Logic Unit (ALU). This unit actually consists of two sub-units, the *ALU_32_Turbo* and the *Barrel_Shifter_AL_32*. The former is an adaptation of the standard *MIPS2000* ALU. In addition to the basic functions of the provided *MIPS2000* ALU, the *ALU_32_Turbo* supports an "exclusive or" operation, a "nor" operation, as well as an additional branch condition evaluation operation. See Appendix F, Specification Sheet 18 for a complete description of the *ALU_32_Turbo*. See Appendix B, Component 14 for the VHDL code for this unit.

Since the *MIPS2000* ISA supports operations with immediate data, there is a two-input 32-bit multiplexer at the B-side of the *ALU_32_Turbo*. This multiplexer selects the type of data that the *ALU_32_Turbo* will perform computations with—either register data or sign-extended immediate data. See Appendix F, Specification Sheet 19 for a complete description of this component. See Appendix B, Component 15 for the VHDL code for this multiplexer.

The second sub-unit of the ALU is the 32-bit Barrel Shifter. This component is responsible for performing all shift operations. The source for the shift amount can either be the contents of a

register or the immediate data. For this reason, the *shift amount* input to the Barrel Shifter is selected from two possible sources via a two-input 5-bit multiplexer (See Appendix F, Specification Sheet 21 and Appendix B, Component 17 for descriptions of this multiplexer). The Barrel Shifter has other inputs to determine whether the shift is arithmetic or logical and whether the direction is to the left or to the right. See Appendix F, Specification Sheet 20 for a description of this component. See Appendix B, Component 16 for the VHDL code for the Barrel Shifter. To select between the output of the ALU Turbo and the Barrel Shifter, another two-input 32-bit multiplexer is used. The final output of the ALU is representative of the opcode, and the two sub-units (the ALU Turbo and the Barrel Shifter) may be thought of as one unit—the ALU.

The component responsible for generating all of the control signals necessary for the proper operation of the Execution Stage is the Execution Stage Controller. This VHDL component examines the opcode and function code of the instruction to determine how to manipulate the data passed to the Execution Stage by the Instruction Decode Stage. See Appendix F, Specification Sheet 22 for a description of this controller. See Appendix B, Component 18 for the VHDL code for the Execution Stage Controller.

The final component in the Execution Stage is the Branch Address Generator. This VHDL unit simply sums the sign-extended immediate data and the *PCplus4* value propagated from the previous stage to produce the branch target address. See Appendix F, Specification Sheet 23 for a complete description of this unit. See Appendix B, Component 19 for the VHDL code for the Branch Address Generator.

These components were graphically combined, and the result is shown in Figure 7 below. See Appendix F, Specification Sheet 24 for a complete description of the *EX_stage* component.

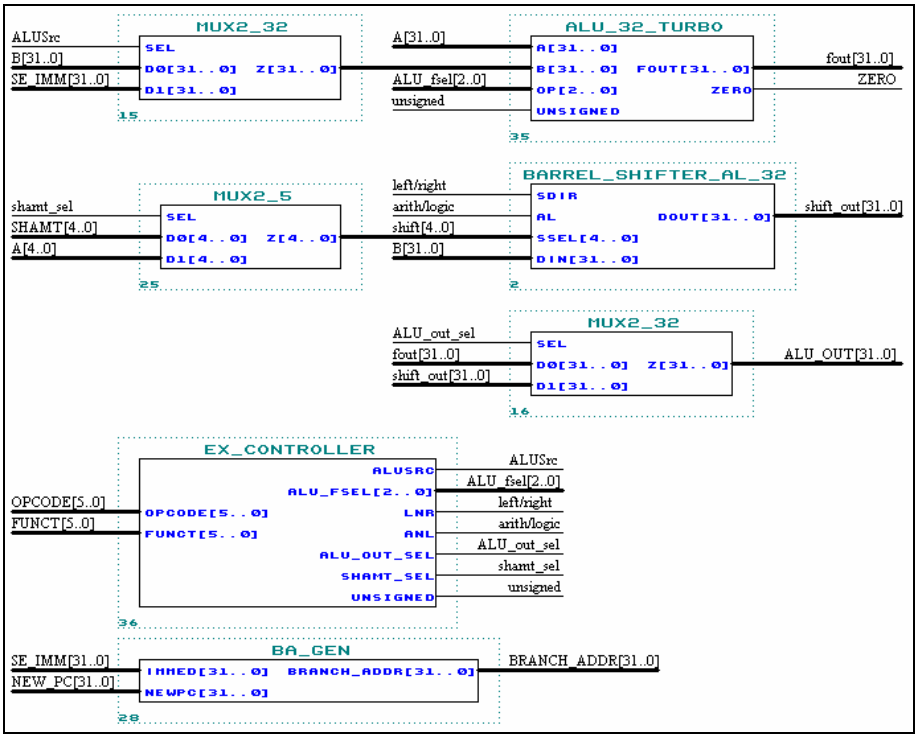

Figure 7: Execution Stage

D. Memory Access Stage

The Memory Access (MEM) stage of the pipeline is responsible for storing data to or retrieving data from the Data Memory. In addition, the source of the next Program Counter address is determined in this stage. The MEM stage of the pipeline was designed based on the schematic shown in Figure 8 below.

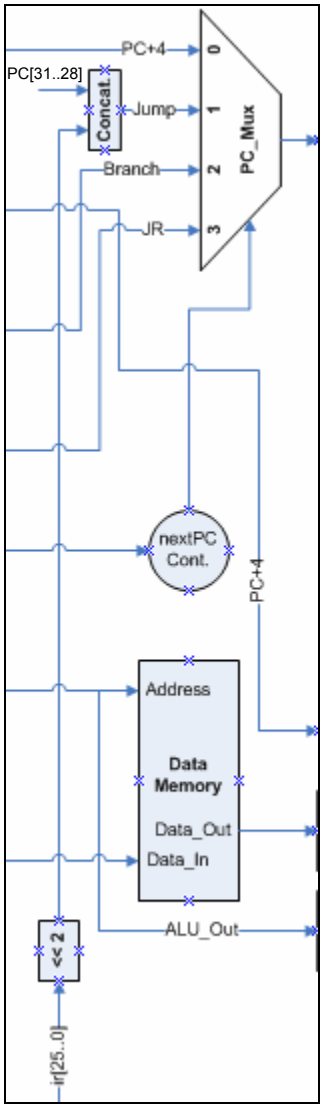

Figure 8: Memory Access Stage

The first component of the Memory Access stage is the Data Memory. This 4Kx32 RAM is comprised of four instances of a 4Kx8 *LPM_RAM_DQ* from the *mega_lpm* library. The 12-bit address bus is connected to the output of the ALU from the previous stage. The input data (for memory writes) is connected to the *B_Data* signal propagated through from the Instruction Decode stage. The output data (for memory reads) is passed on to the next stage. Each of the four 4Kx8 RAMs has its own write enable signal such that the smallest writeable unit of data is a byte. See Appendix F, Specification Sheet 25 for a complete description of the Data Memory.

The next components within the MEM stage are responsible for controlling the flow of data in to and out of the Data Memory. The Memory Control is composed of two control units, the *MEM_Writer* and the *MEM_Reader* units. The *MEM_Writer* modifies the data that will be written to the Data Memory, and it sets the Write Enable signals accordingly. This component examines the opcode and the effective address of the current instruction to determine how to prepare the data for writing. With byte and half-word data writes supported in the *MIPS2000* ISA, this unit is responsible for directing the data to its proper destination. See Appendix F, Specification Sheet 26 for a description of this component. See Appendix B, Component 20 for the VHDL code for the *MEM_Writer* unit. The *MEM_Reader* modifies the data read from the memory so as to accommodate different types of load instructions (i.e. *lb*, *lw*, *lh*, etc.). See Appendix F, Specification Sheet 27 for a description of this component. See Appendix B, Component 21 for the VHDL code for the *MEM_Writer* unit.

The final component in the Memory Access Stage is the Next PC Source Selector. This consists of a four-input 32-bit multiplexer that selects one of four possible sources for the next Program Counter value: the *PCplus4* value, the Jump address, the Branch address, or the address contained in a register (i.e. the Return Address register). See Appendix F, Specification Sheet 28 for a description of the four-input 32-bit multiplexer. See Appendix B, Component 22 for the VHDL code for this component.

The component responsible for generating the select signal for the Next PC Source Selector is the *NextPC_Gen* unit. This VHDL component examines the opcode and function code of the instruction to determine where the Next PC value should be selected from. See Appendix F, Specification Sheet 29 for a description of the *NextPC_Gen* unit. See Appendix B, Component 23 for the VHDL code for this component.

These components were graphically combined, and the result is shown in Figures 9a (below), 9b (next page), and 9c (next page). See Appendix F, Specification Sheet 30 for a complete description of the *MEM_stage* component.

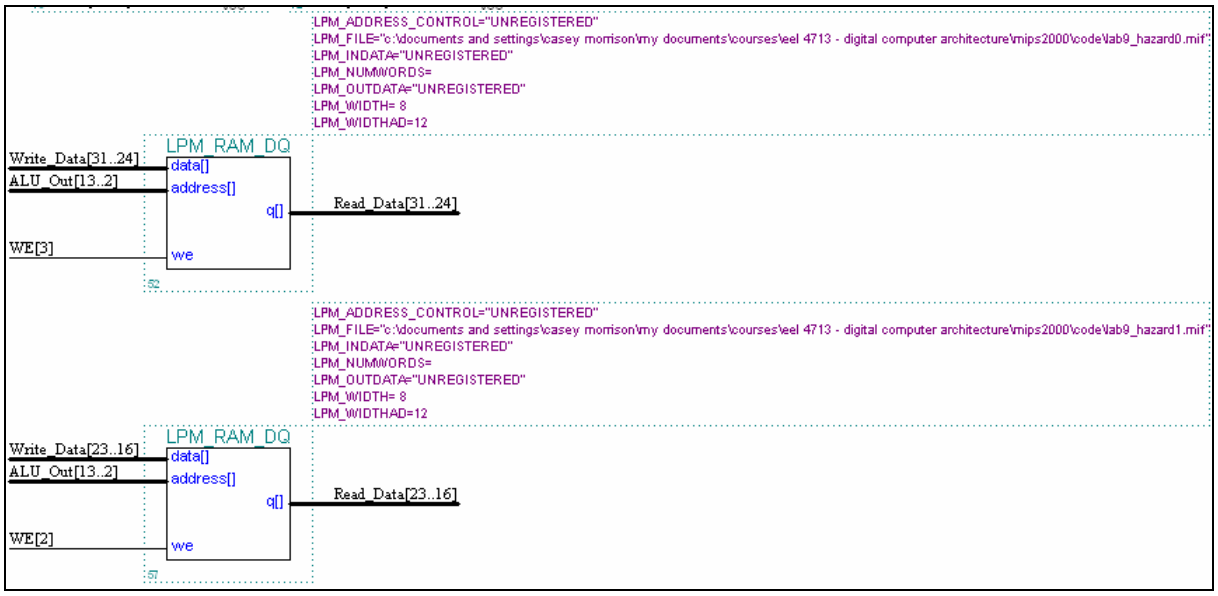

Figure 9a: Memory Access Stage

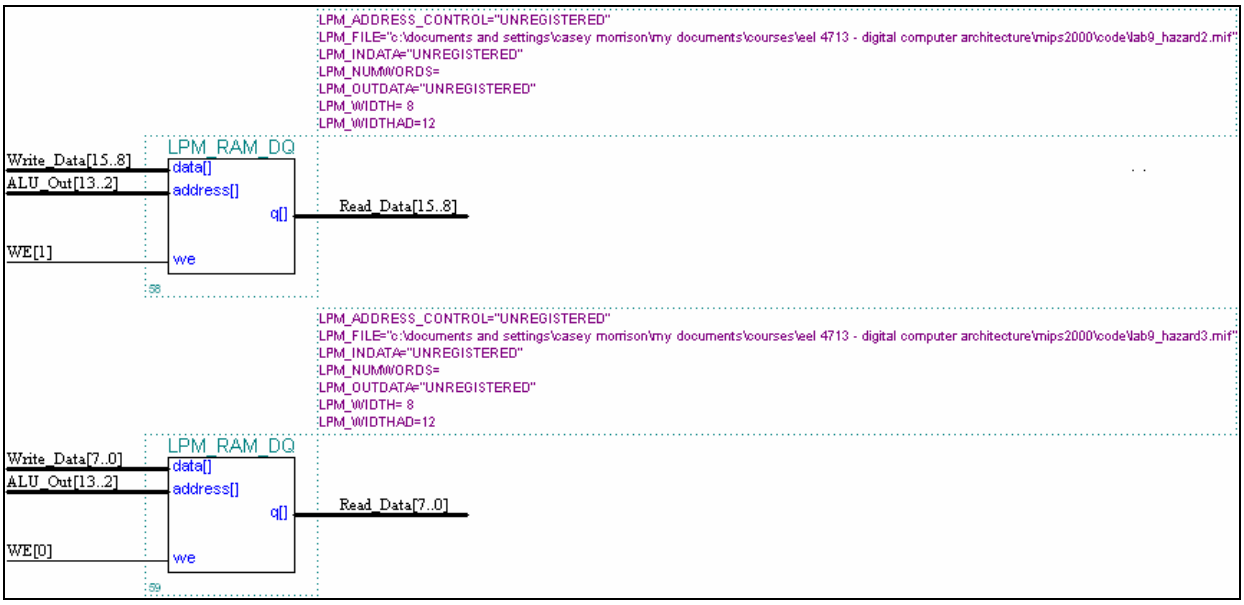

Figure 9b: Memory Access Stage

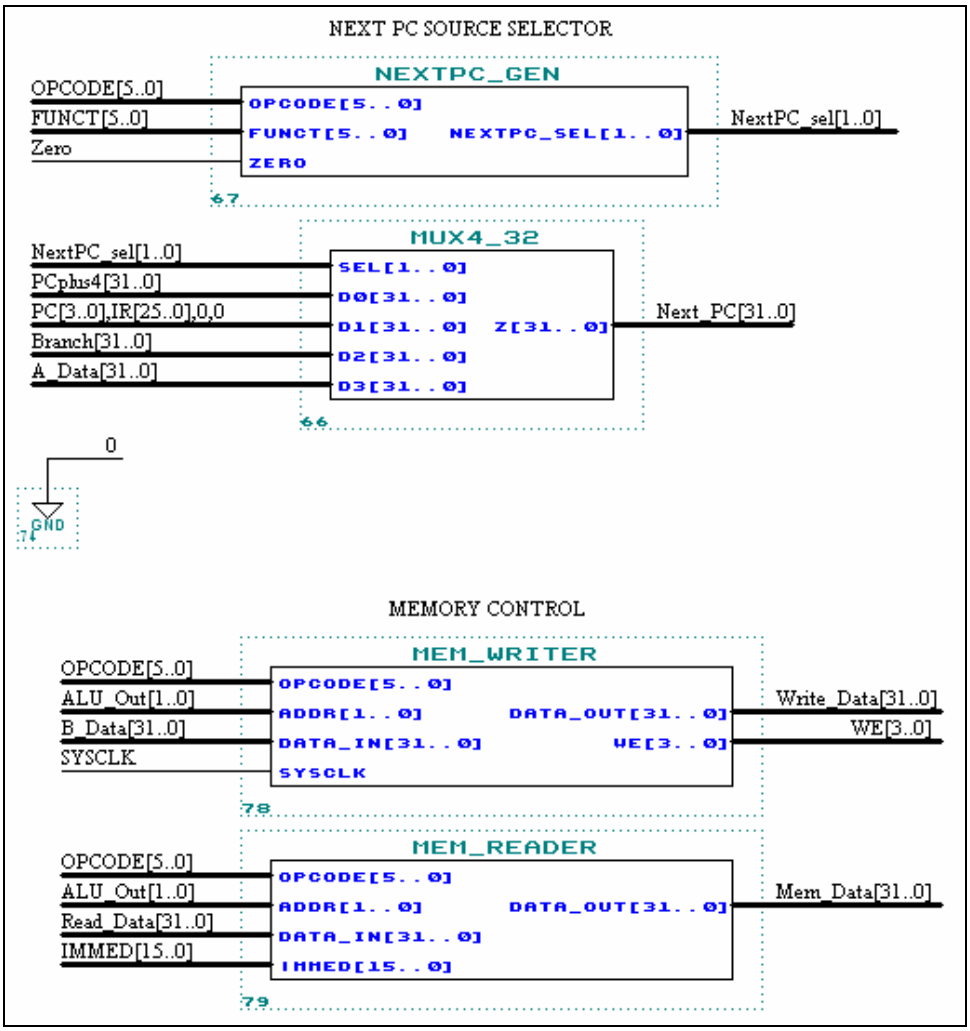

Figure 9c: Memory Access Stage

E. Write Back Stage

The Write Back (WB) stage of the pipeline is responsible for selecting the data that is to be written back to the register array. In addition, the instruction must be examined to determine if, in fact, a write back should occur. The Write Back stage of the pipeline was designed based on the schematic shown in Figure 10 below.

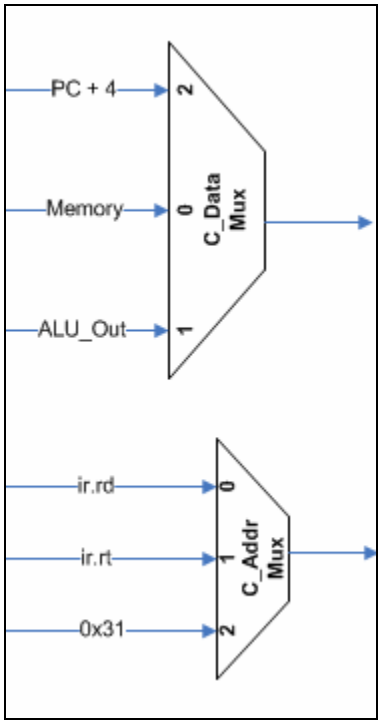

Figure 10: Write Back Stage

The first component of the Write Back stage is the C-Data Selector. This is a VHDL description of a three-input 32-bit multiplexer that selects between the three possible sources of data for the write back register: the Data Memory, the ALU, or the *PCplus4* value (for storing return addresses). See Appendix F, Specification Sheet 31 for a description of this multiplexer. See Appendix B, Component 24 for the VHDL code for the *Mux3_32* component.

The next component of the Write Back stage is the C-Address Selector. This is a VHDL description of a three-input 5-bit multiplexer that selects between the three possible sources of the address for the write back register: the *rd* field of the Instruction Register, the *rt* field of the Instruction Register, or a hard-coded value *0x31* (default Return Address Register). See Appendix F, Specification Sheet 32 for a description of this multiplexer. See Appendix B, Component 25 for the VHDL code for the *Mux3_5* component.

The final component in the Write Back Stage is the Write Back Controller. This unit determined the select signals for the aforementioned multiplexers as well as the Write Enable signal for the Register Array. These decisions are based on the instruction opcode and function code. See Appendix F, Specification Sheet 33 for a description of this controller. See Appendix B, Component 26 for the VHDL code for the *WB_Controller* component.

These components were graphically combined, and the result is shown in Figure 11 below. See Appendix F, Specification Sheet 34 for a complete description of the *WB_stage* component.

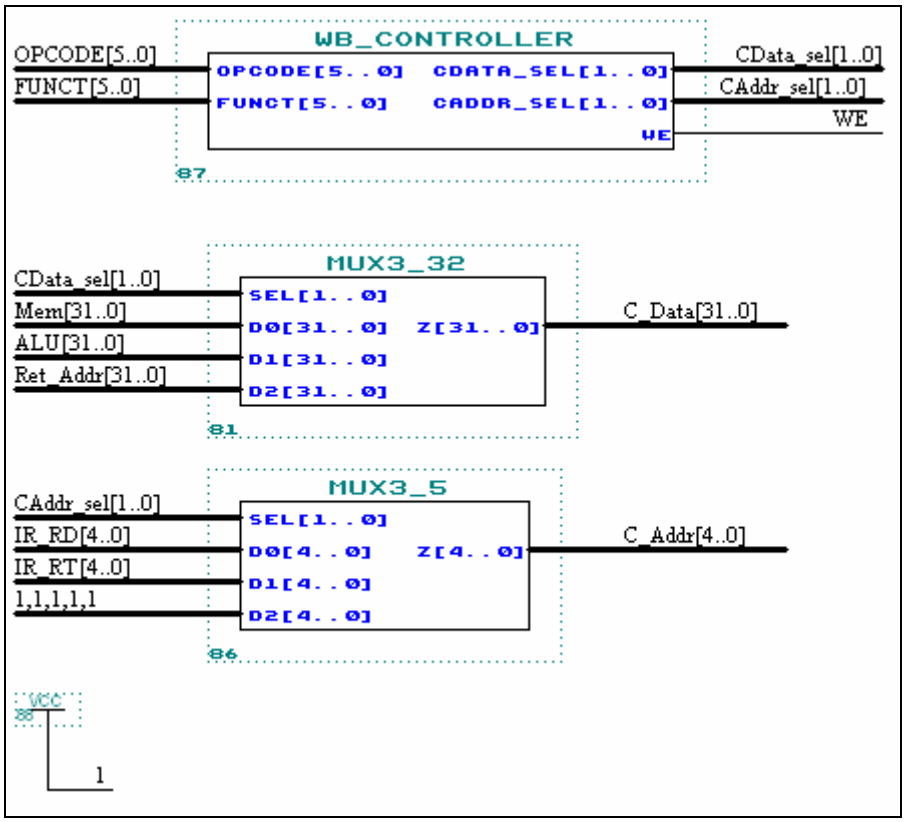

Figure 11: Write Back Stage

F. Pipeline Registers

In between each of the five stages lies a pipeline register. These registers are responsible for storing the instruction that is currently in the stage, as well as a variety of other signals that need to be propagated through the architecture. For example, since the *NextPC* value is not calculated until the Memory Access stage, the *PCplus4* signal must be propagated through the IF, ID, and EX stages so that is may be chosen as the *NextPC* in the MEM stage.

To construct the four pipeline registers, several registers of various sizes were built in VHDL. One such register, the *Instr_Reg* component, was used in each pipeline register to store the current instruction word.

Figures 12 through 15 on the next pages show all four pipeline registers. Note that all *Intr_Reg* components are clearable so that upon a system reset, all instructions in the pipeline will default to *NOP*s.

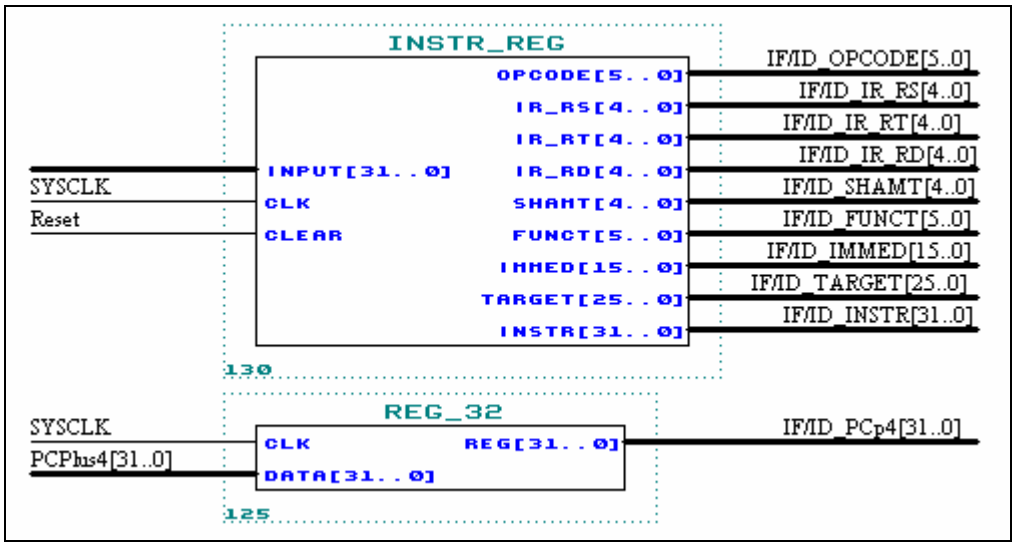

Figure 12: IF/ID Pipeline Register

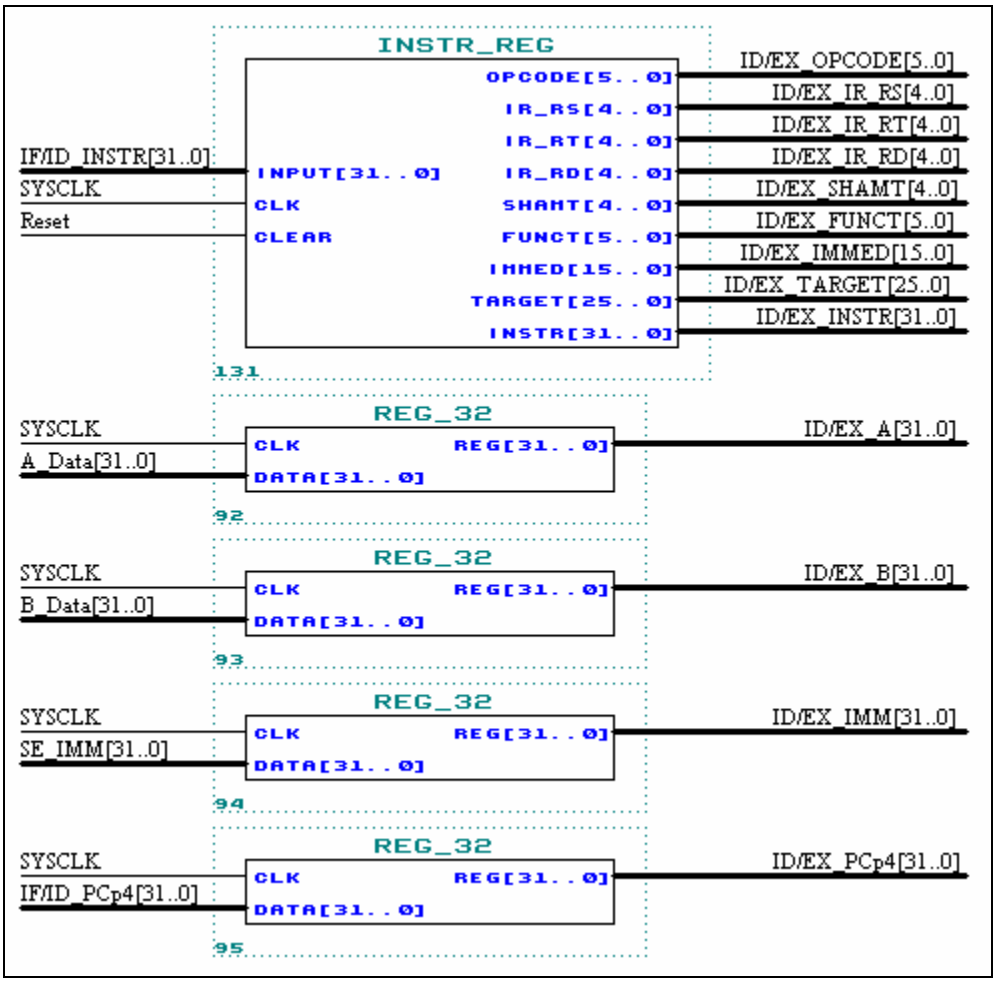

Figure 13: ID/EX Pipeline Register

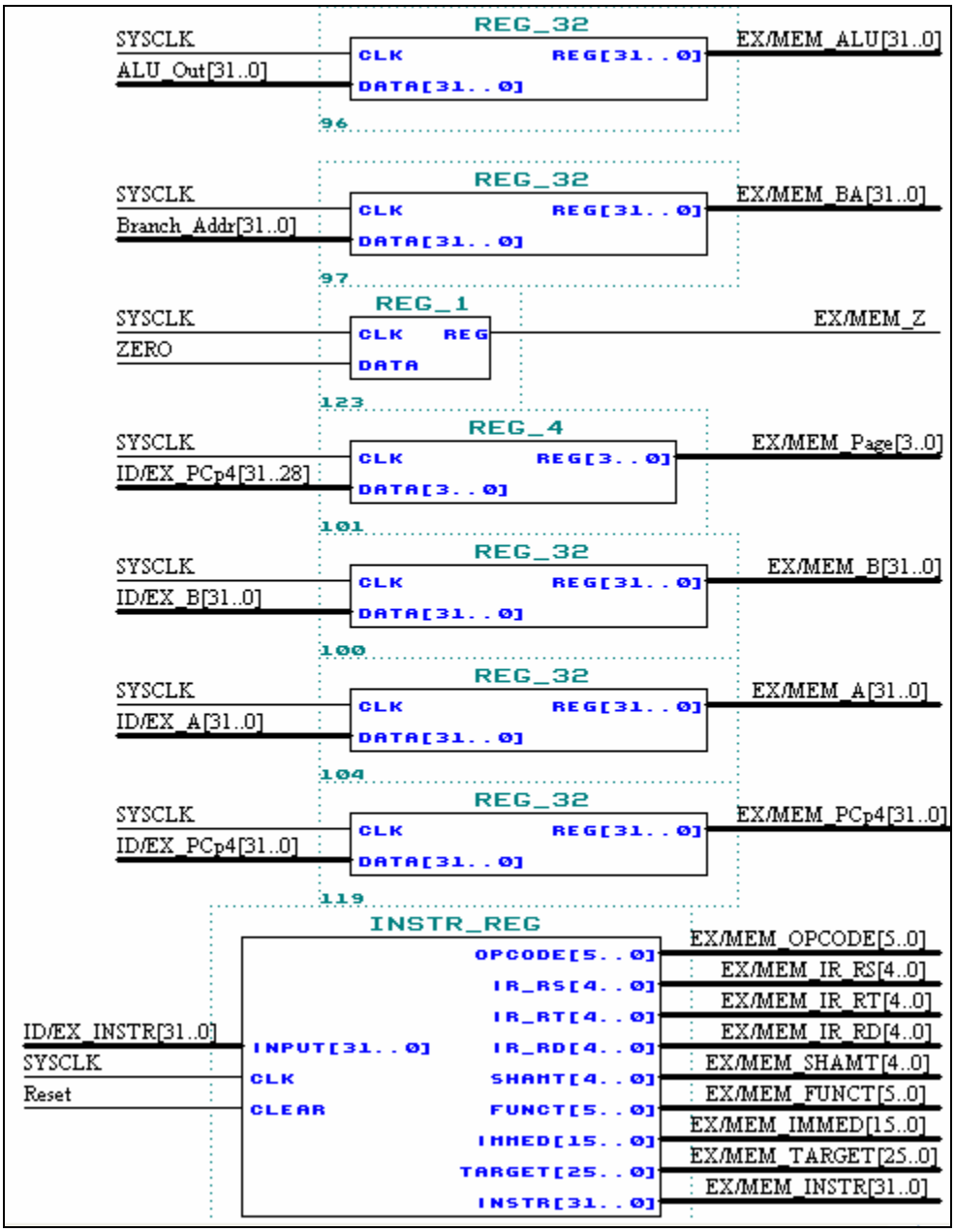

Figure 14: EX/MEM Pipeline Register

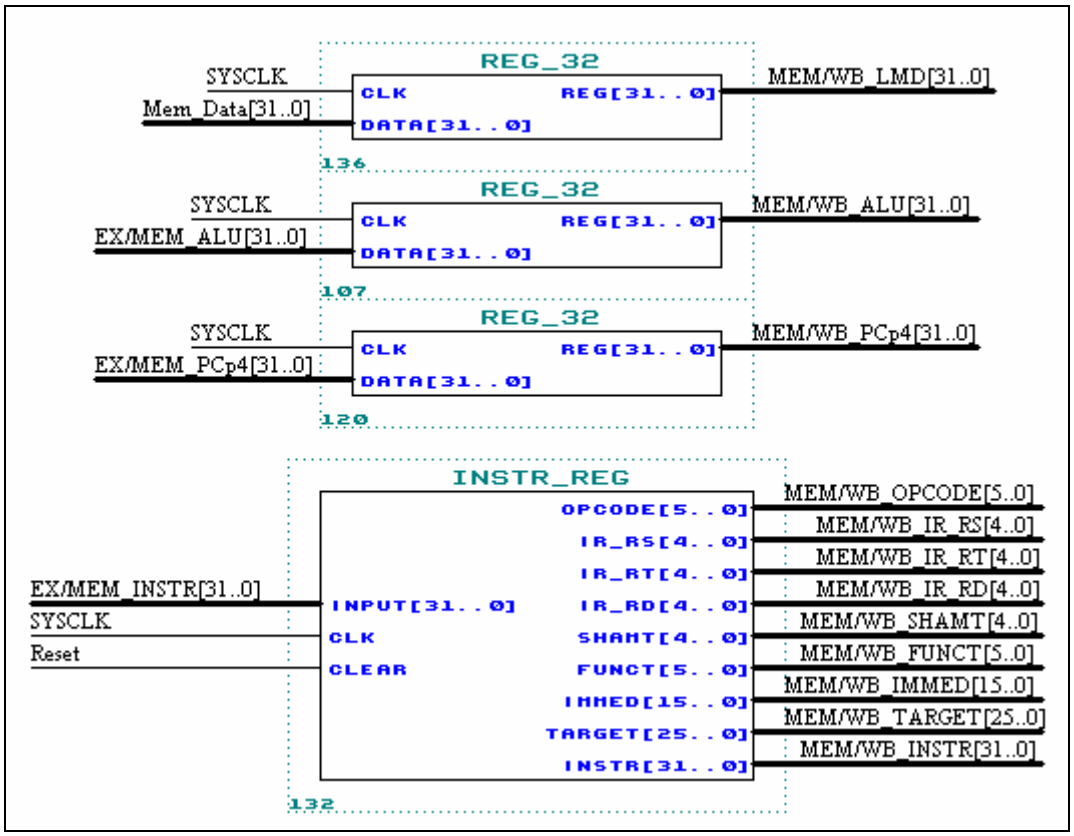

Figure 15: MEM/WB Pipeline Register

See Appendix F, Specification Sheets 35 through 38 for descriptions of the various components that comprise the pipeline registers. See Appendix B, Components 27 through 30 for the VHDL code for these components.

System Design and Validation

A. MIPS2000 Complete Design

Once all the individual stages and pipeline registers were constructed, the *MIPS2000* microprocessor was assembled according to the schematic drawing in Figure 16 on the next page.

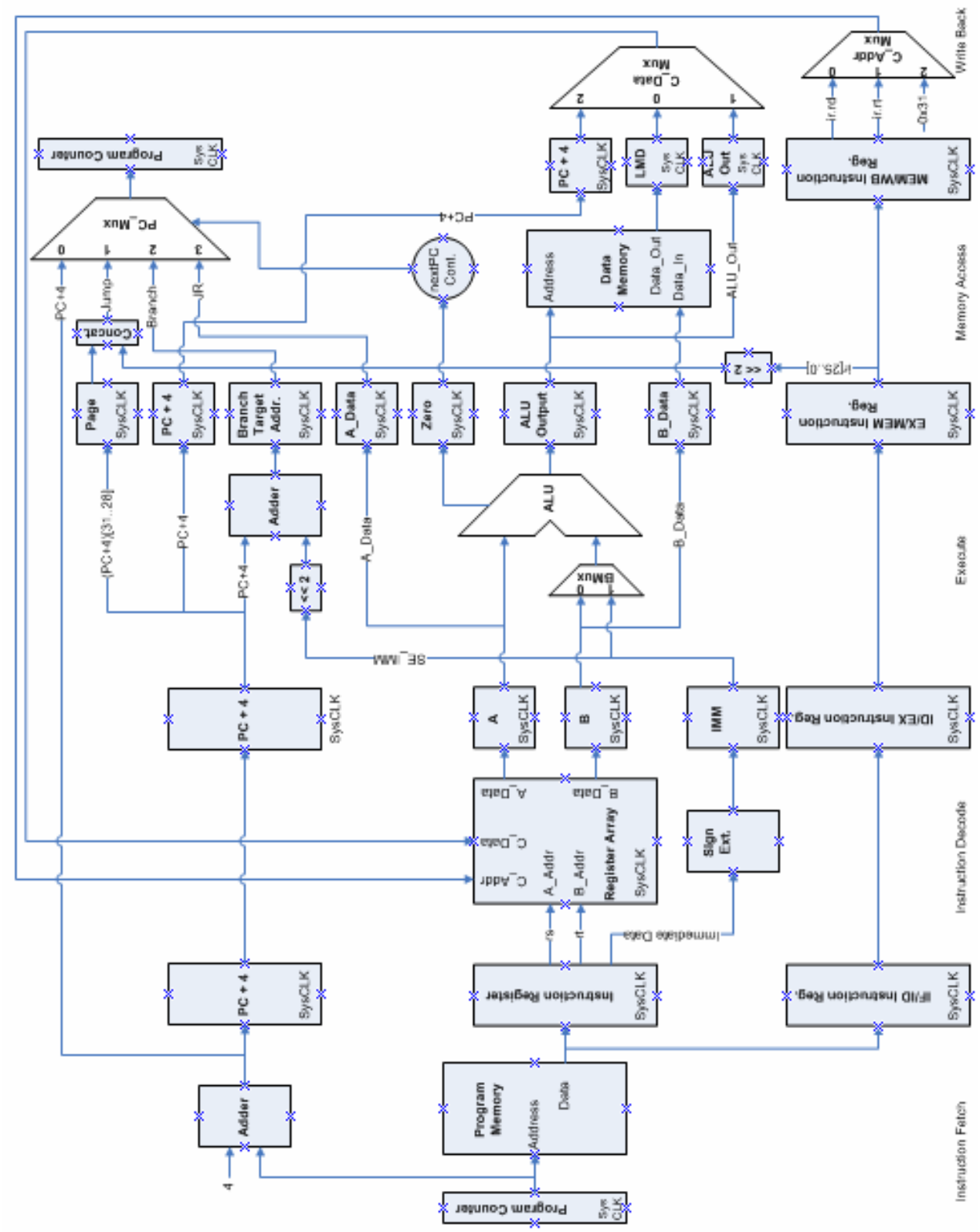

Figure 16: *MIPS2000* pipelined datapath

The assembly of the complete datapath was trivial once all of the individual stages and inter-stage pipelines were created. The resulting design is shown in Figures 17a, 17b, 17c, and 17d below and on the next page.

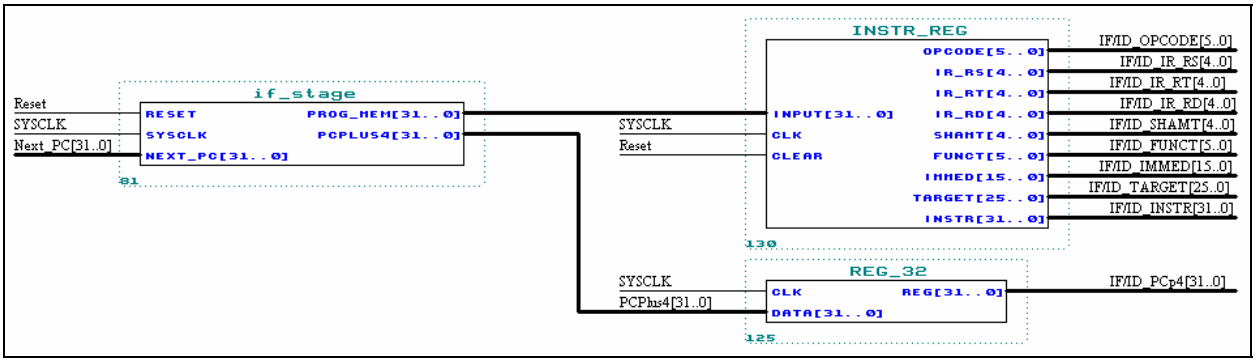

Figure 17a: *MIPS2000 complete architecture*

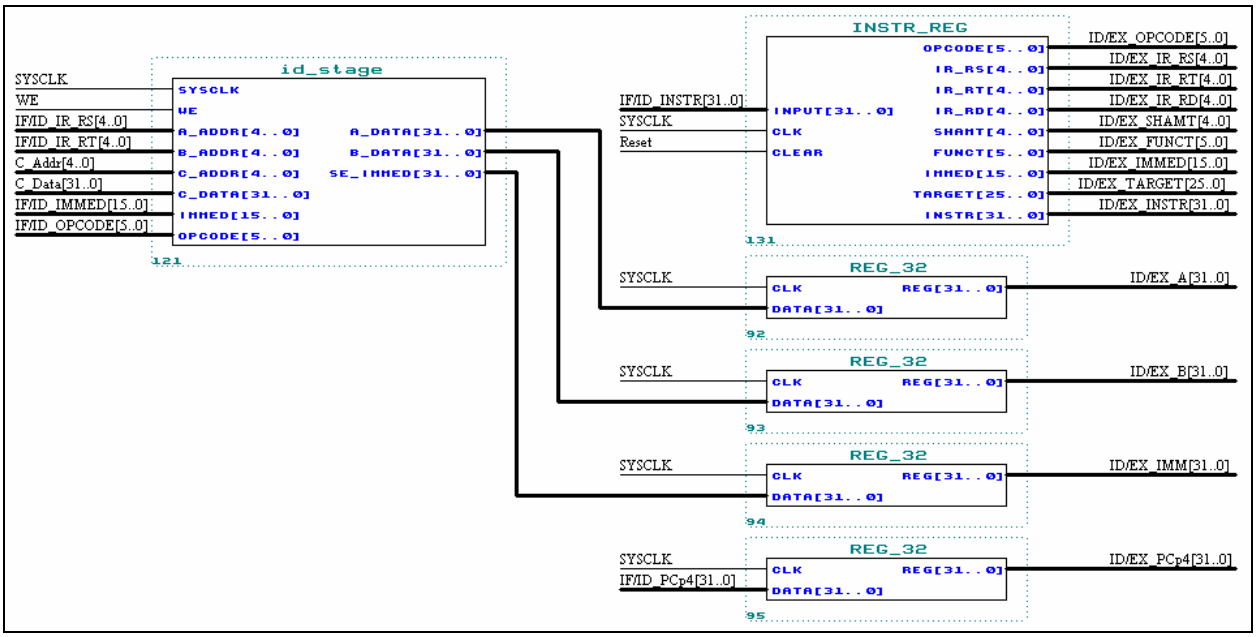

Figure 17b: *MIPS2000 complete architecture*

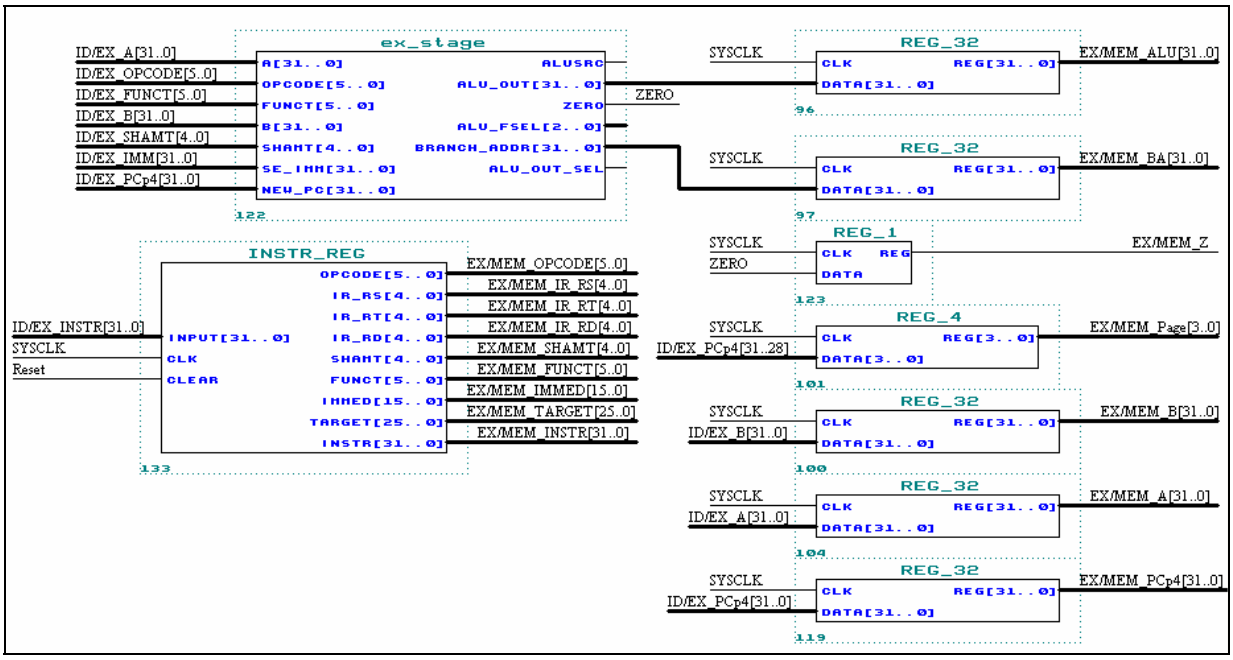

Figure 17c: *MIPS2000 complete architecture*

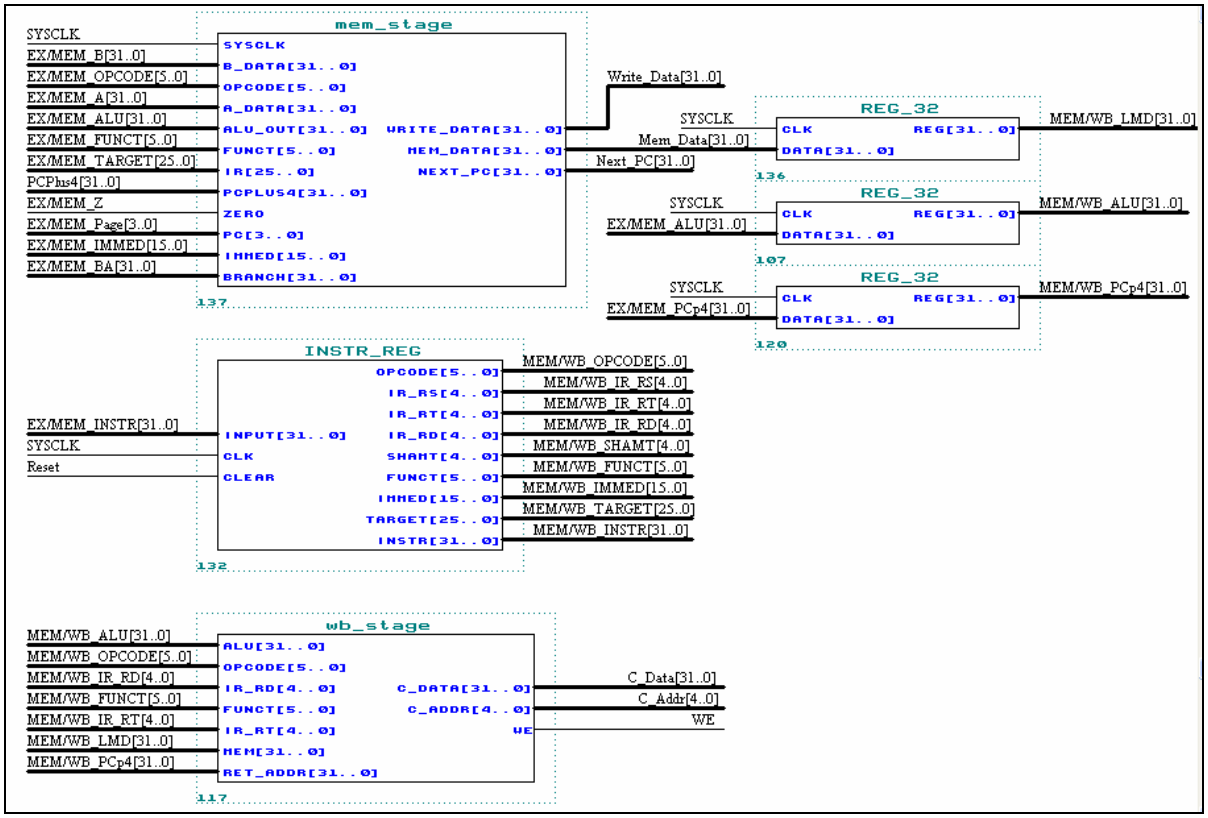

Figure 17d: *MIPS2000 complete architecture*

This system was functionally compiled to eliminate all errors. See Appendix F, Specification Sheet 39 for a description of the *MIPS2000* component.

B. System Test: Lab8_test.asm

In order to test the operation of the newly constructed *MIPS2000* microprocessor, a test program was written. This program, *Lab8_test.asm*, evaluated a variety of arithmetic, logic, immediate, load, store, branch, and jump functions. Listing 1 below shows the program developed to evaluate the performance of the *MIPS2000*.

```
        nolist
        include "mips.mac"
        list

        ORG $0000

        li      R2,$9ABC
        li      R3,$1234
        li      R4,16
        li      R5,$D
        li      R6,$5678
        li      R7,$B0
        add     R20,R3,R3
        sub     R20,R2,R3
        and     R20,R2,R6
        or      R20,R2,R3
        xor     R20,R2,R3
        sll     R21,R2,16
        addi    R20,R3,$20F6
        ori     R20,R3,$0055
        xori    R20,R6,$5678
        slt     R20,R6,R2
        sltu    R20,R6,R2

        jal     sub1
        nop
        nop
        nop
        and     R20,R0,R0
        bne     R0,R0,finish
        nop
        nop
        nop

        bgez    R6,finish
        nop
        nop
        nop

        ORG $00B0
sub1:   sw      R6,R0,data1
        sb      R2,R0,data2
        jr      R31
        nop
        nop
        nop

finish: li      R9,$EEEE
        lw      R10,R0,data1        ;Check if previous store was successful
        lb      R11,R0,data2

        GFO

        ORG $00D0
data1   1
data2   ds.l    1

        end
```

Listing 1: Lab8_test.asm

It is important to notice that following every load, branch, jump, and other data-dependant instructions, *NOPs* have been inserted to avoid hazards. In Lab 9, the architecture will be adjusted to mitigate and compensate for these hazards; however in this lab, those hazards are avoided all together by inserting *NOPs* after all instructions that may cause a data dependency.

This program (also found in Appendix C, Program 26) was compiled with *UPASM* and converted into four Memory Initialization Files (MIFs). This is the file format for the four Program Memory ROMs contained in the IF stage as well as the four Data Memory RAMs contained in the MEM stage. In order to convert the assembled s-record into four MIFs, the *S2MIF* program used earlier for the *Sweet16* had to be adapted to generate the appropriate MIFs. As a result, four *S2MIF* programs were created, one for each ROM/RAM in the *MIPS2000* architecture. See Appendix A, Program 6 through 9 for the *s2mif0*, *s2mif1*, *s2mif2*, and *s2mif3* C programs.

When the *Lab8_test.asm* program was simulated on the *MIPS2000* architecture, the results demonstrated that the design worked properly. See Appendix E, Waveform Simulation 14 for the complete simulation results.

Conclusion

A. Summary

The *MIPS2000* is very modular in nature with its five-stage pipeline and multiplexer-oriented design. This characteristic allowed each individual part to be tested and verified prior to the assembly of the entire processor.

With a pipelined datapath such as that utilized in the *MIPS2000*, the Cycles Per Instruction (CPI) rate can be dramatically improved since up to four instructions can be "executed" at any given time. Although the concept of pipeline registers greatly improves the CPI of the machine, it is also makes it susceptible to hazardous situations in which data conflicts and discrepancies occur. This topic will be discussed at length in Lab 9, in which several strategies will be implemented to lessen the effects—and perhaps avoid all together—of data dependencies and hazards.

B. Questions

1) *Which component in the MIPS integer pipeline is used to determine the clock rate of the entire pipeline? Why?*

   The slowest component in the *MIPS2000* integer pipeline is used to determine the clock rate, for the clock rate cannot exceed the inverse of the largest propagation delay through the circuit. This component happens to be the 32-bit ALU (the adder in particular) since the assumption in *MIPS* is that the memory is fast.

2) *Are the program and data memory sizes reasonable for this computer? What should happen to the propagation time for the memories when they are realized using components that have a size appropriate for this machine?*

   The Program and Data Memory sizes (currently 4KB) are not that reasonable considering that they each have a potential of being 1GB given the size of the address bus. When larger memories are implemented in this architecture, the propagation time for the memories will increase accordingly. The generally rule for memory is that larger means slower, and this will be the case if the memory sizes are increased to a more “reasonable” size.

Lab No. 9
Casey T. Morrison
EEL 4713 Section 2485 (Spring 2004)
Lab Meeting Date and Time: Monday E1-E3
TA: Grzegorz Cieslewski

I have performed this assignment myself. I have performed this work in accordance with the Lab Rules specifies in 4713 Lab No. 0 and the University of Florida's Academic Honesty manual. On my honor, I have neither given nor received unauthorized aid in doing this assignment.

_____

Introduction

The purpose of this lab was to study and construct the *MIPS2000* advanced architecture. This involved making modifications to the general architecture designed in Lab 8. The *MIPS2000* microprocessor designed in Lab 8 was susceptible to two types of hazards: data hazards and branch hazards. For this reason, several *NOP* instructions had to be manually inserted into the code to avoid data dependencies, data hazards, and branch hazards.

To mitigate the effects of these hazards, this lab focused on three hardware solutions: forwarding, stalling, and branch prediction. Forwarding attempts to resolve data dependencies, stalling handles situations in which forwarding is insufficient, and branch prediction attempts to reduce the cost associated with branches. Since each is a hardware solution, the original code that was executed on the Lab 8 version of *MIPS2000* will still execute on the advanced *MIPS* architecture, except with a markedly better performance.

Component Design and Validation

As mention previously, the *MIPS2000* advanced architecture features three hardware improvements from its Lab 8 counterpart. Forwarding, stalling, and branch prediction all attempt to avoid hazardous situations and improve performance.

A. Forwarding

In some instances, a pipelined datapath (such as the *MIPS2000*) may encounter data dependencies during execution. For example, the code sequence below incurs two data dependencies involving register R1.

ADD ***R1****,R2,R3*
*SUB R5,R4,****R1***
*ORI R6,****R1****,$1234*

The result of the addition of registers R2 and R3 is not stored back into the Register Array until the *Write Back* stage. However, the following two instructions require that data before it is actually available in the Register Array.

The solution to this type of situation is “Forwarding.” The result of the addition of registers R2 and R3 is computed in the *Execute* stage. Thus if this result is required for the subsequent instructions, it may be “forwarded” to the *Execute* stage of those instructions. This bypasses the Register Array and ensures that the following instructions are using the correct data.

Forwarding is accomplished with multiplexers and a Forwarding Unit. The multiplexers are placed at the input to the ALU so that the Forwarding Unit may decide which data the ALU will operate with. If the conditions for forwarding are met, then the Forwarding Unit selects the

forwarded data as the input to the ALU. The multiplexer scheme implemented in the *Execute* stage is shown below in Figure 1.

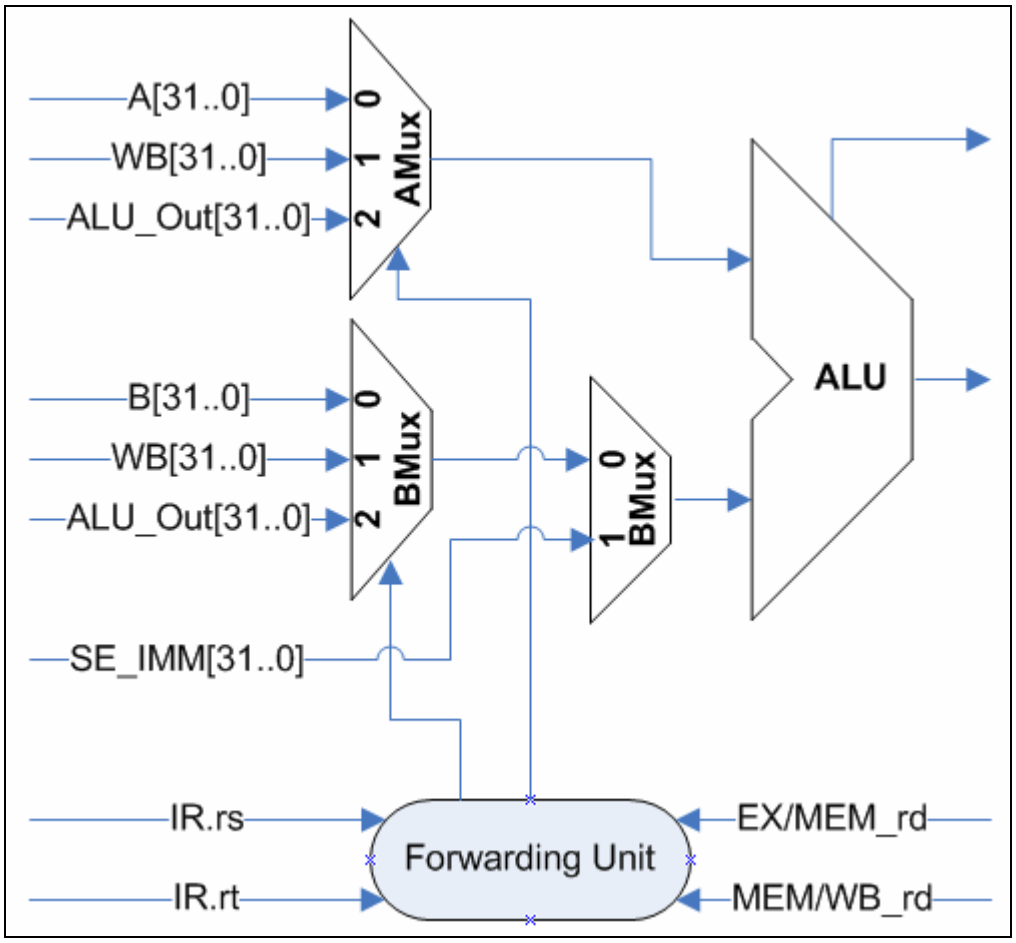

Figure 1: Forwarding scheme

The Max+Plus II graphical realization of this scheme is shown below in Figure 2. See Appendix F, Specification Sheet 31 for a description of the *Mux3_32* component.

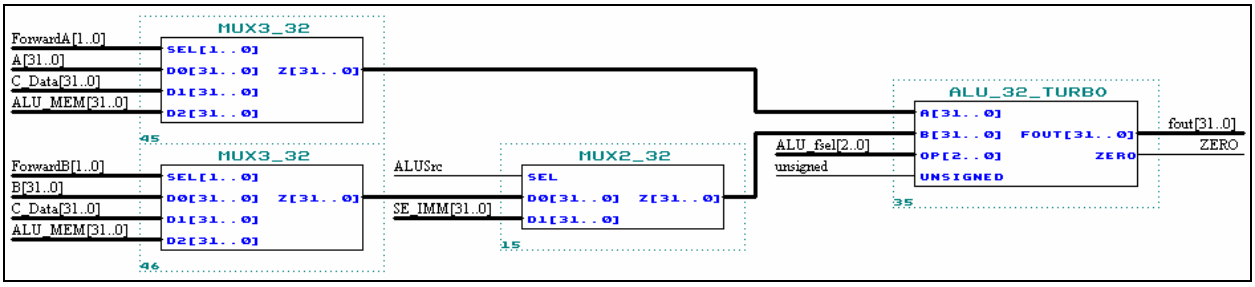

Figure 2: Forwarding Multiplexers in Max+Plus II

The control signals for the two forwarding multiplexers are generated by the Forwarding Unit. This VHDL component examines the various conditions that indicate a need for forwarding and issues the corresponding control signals. The conditions used to determine the control signals are

shown below in Listing 1. See Appendix F, Specification Sheet 40 for a description of this component. See Appendix B, Component 31 for the VHDL code for the Forwarding Unit.

```
-- Determine ForwardA
if ((WB_stage_WE = '1') and (MEM_WB_RD /= "00000") and (EX_MEM_RD /= ID_EX_RS) and
     (MEM_WB_RD = ID_EX_RS)) then
      ForwardA <= "01";
elsif  ((MEM_stage_WE = '1') and (EX_MEM_RD /= "00000") and (EX_MEM_RD = ID_EX_RS)) then
      ForwardA <= "10";
else
      ForwardA <= "00";
end if;

-- Determine ForwardB
if ((WB_stage_WE = '1') and (MEM_WB_RD /= "00000") and (EX_MEM_RD /= ID_EX_RT) and
     (MEM_WB_RD = ID_EX_RT)) then
      ForwardB <= "01";
elsif ((MEM_stage_WE = '1') and (EX_MEM_RD /= "00000") and (EX_MEM_RD = ID_EX_RT)) then
      ForwardB <= "10";
else
      ForwardB <= "00";
end if;
```

Listing 1: Forwarding control logic

With the new Forwarding Unit and multiplexers in place, the *Execute* stage was successfully modified to handle most data dependencies. Figure 3 below shows the new *Execute* stage graphical design. See Appendix F, Specification Sheet 41 for a description of the *EX* stage component.

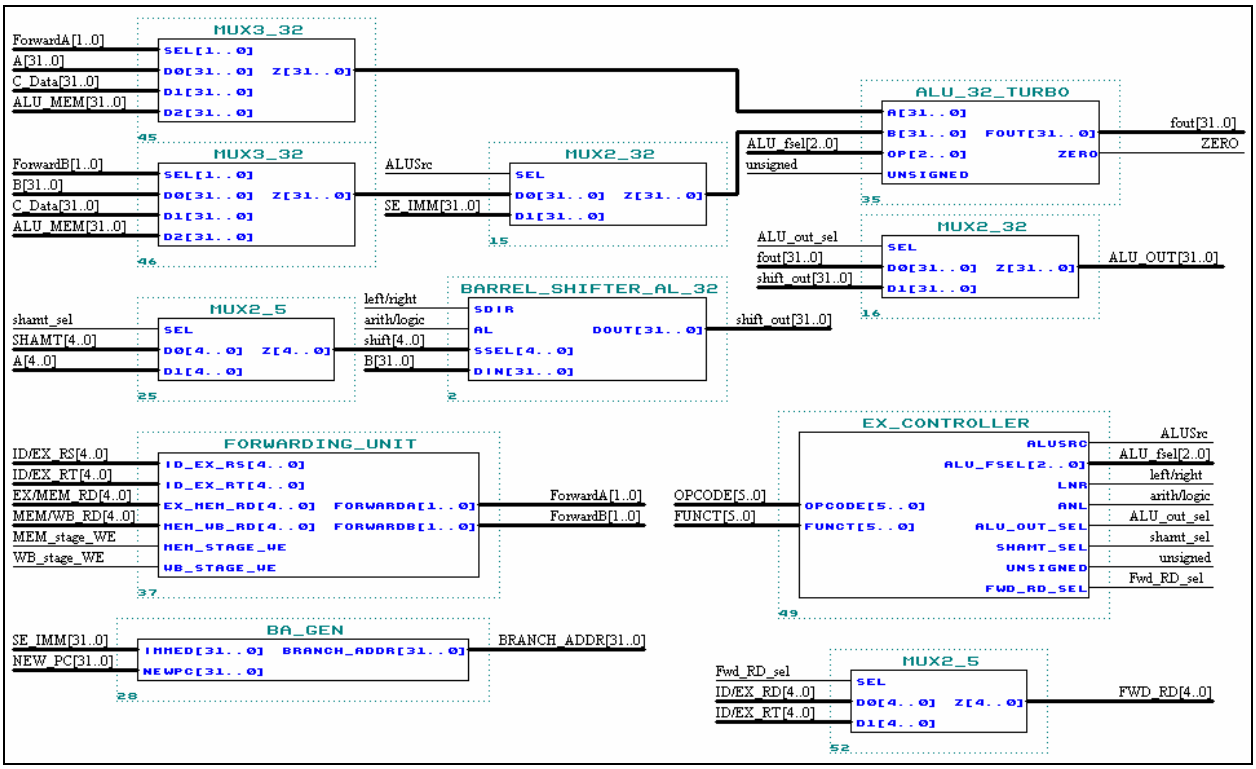

Figure 3: *Execute* stage with Forwarding capabilities

B. Pipeline Stalling

Some data dependencies cannot be solved with forwarding. For example, the code segment below contains a data dependency that is not resolved until the first instruction is done with the *MEM* stage.

LW ***R1****,4(R2)*
*XOR R2,R3,****R1***
*SLL R4,****R1****,16*

The output of the data memory cannot be forwarded to the *EX* stage in time for the *XOR* instruction to execute with the proper operands. Forwarding *will* work for the *SLL* instruction because the *XOR* instruction serves as a delay slot. This means that the output of the data memory is available in time for the *EX* stage of the *SLL* instruction.

To resolve the hazard created by this type of data dependency, the most practical solution is to stall the pipeline for one clock cycle so that the output of the data memory may then be forwarded to the *EX* stage should it be needed. Using this method, the dependency must be recognized early on in the pipeline—in the *ID* stage to be exact. Using a VHDL-described Hazard Detection Unit, a *stall* signal may be issued throughout the pipeline to avoid the hazard.

When a *stall* is issued, a couple registers must hold their data, while others must clear their output. In order to preserve the correct sequence of instructions in the pipeline, the Program Counter and the IF/ID Pipeline Register must hold their data. In addition, the ID/EX Pipeline Register must clear its data so that the stall (implemented as a *NOP* instruction) may be inserted into the pipeline. The holding and clearing of registers is accomplished with some VHDL modifications.

The Hazard Detection Unit in the *ID* stage uses a series of logic equations to determine is a hazard will be encountered. It uses a Load Detection Unit to determine if an instruction will be loading data from memory to a register (See Appendix F, Specification Sheet 42 and Appendix B, Component 32 for a description of the Load Detection Unit). This is accomplished by examining the opcode of the instruction in the ID/EX Pipeline Register. Listing 2 below shows the logic used to detect hazards in the pipeline. See Appendix F, Specification Sheet 43 for a description of the Hazard Detection Unit. See Appendix B, Component 33 for the VHDL code for this component.

```
if  (ID_EX_MemRead = '1' and ((ID_EX_RT = IF_ID_RS) or (ID_EX_RT = IF_ID_RT))) then
       Stall <= '1';
else
       Stall <= '0';
end if;
```

Listing 2: Hazard Detection logic

After modifying the *ID* stage to include the Hazard Detection Unit, the resulting graphic design is shown in Figure 4 on the next page. See Appendix F, Specification Sheet 44 for a description of the modified *ID* stage.

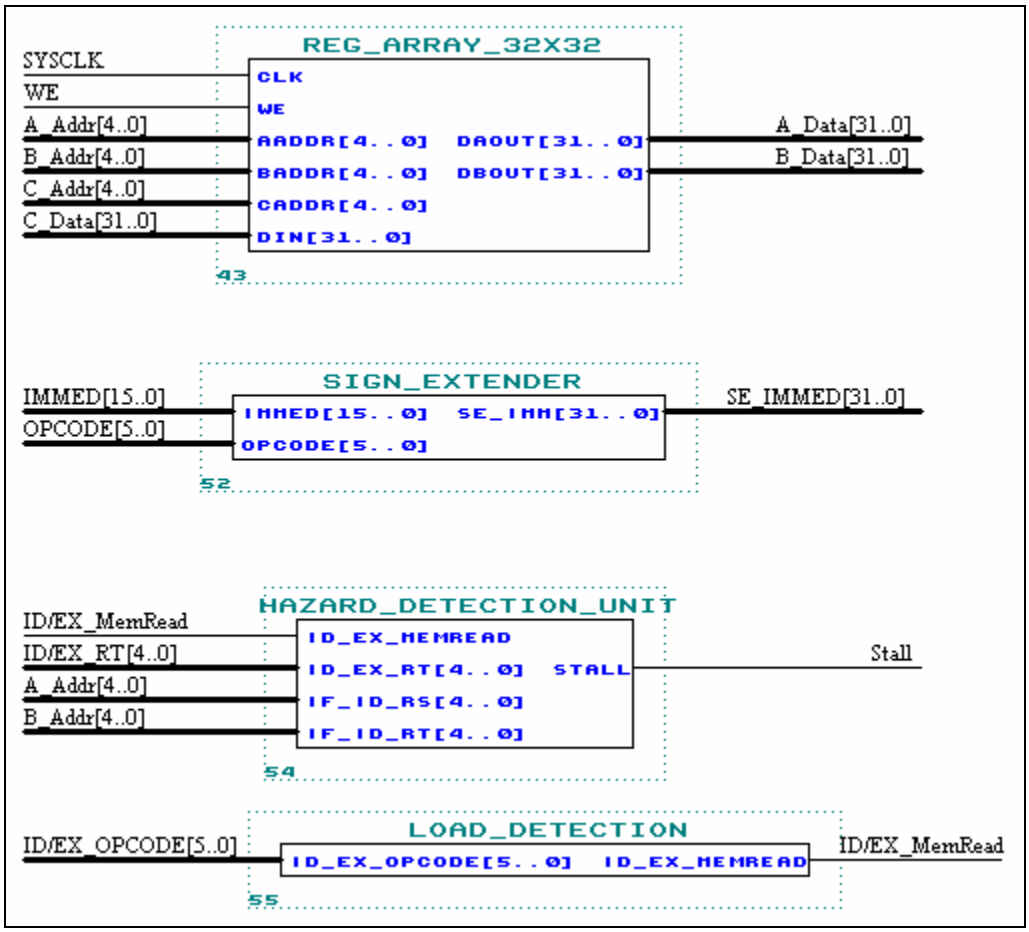

Figure 4: *Instruction Decode* stage with Hazard Detection

C. Branch Prediction

In a pipelined processor (such as the *MIPS2000*) it is often beneficial to employ a branch prediction scheme to improve the performance of the machine. Since the next Program Counter address is not calculated until the *MEM* stage, the three instructions in the pipeline before the *MEM* stage may turn out to be the wrong three instructions when the next PC is calculated. To avoid this problem in Lab 8, three *NOP*s were inserted following every branch and jump so as to avoid the possibility of executing the wrong instructions following a branch/jump.

In this lab, the "assume not taken" strategy was adopted to modestly predict the outcome of branches. Using this strategy, all branches and jumps were "assumed" not to be taken, and the three following instructions were fed into the pipeline. During the *MEM* stage, if the branch/jump was, in fact, not taken, then the prediction was correct and execution continues uninterrupted. If, however, the branch condition evaluates to true, and the branch *was* supposed to be taken, then the pipeline must be flushed and execution must stall.

Since the three instructions following the branch/jump do not reach the *WB* stage by the time the prediction is evaluated, The consequences of executing these instructions through the *EX* stage is

negligible. If the prediction was found to be incorrect, then the IF/ID, ID/EX, and EX/MEM Pipeline Registers must be cleared and the pipeline must begin executing the proper instructions. When the IF/ID, ID/EX, and EX/MEM Pipeline Registers are cleared, this is equivalent to inserting *NOP*s into the pipeline. The penalty, therefore, for a mis-predicted branch is three clock cycles.

The simple hardware used to issue the *Flush_Pipeline* command was added to the *MEM* stage of the pipeline. If the source of the next Program Counter value is either the branch target address or the jump target address, then the prediction was wrong, and the pipeline must be flushed. Figure 5 below shows this hardware in detail.

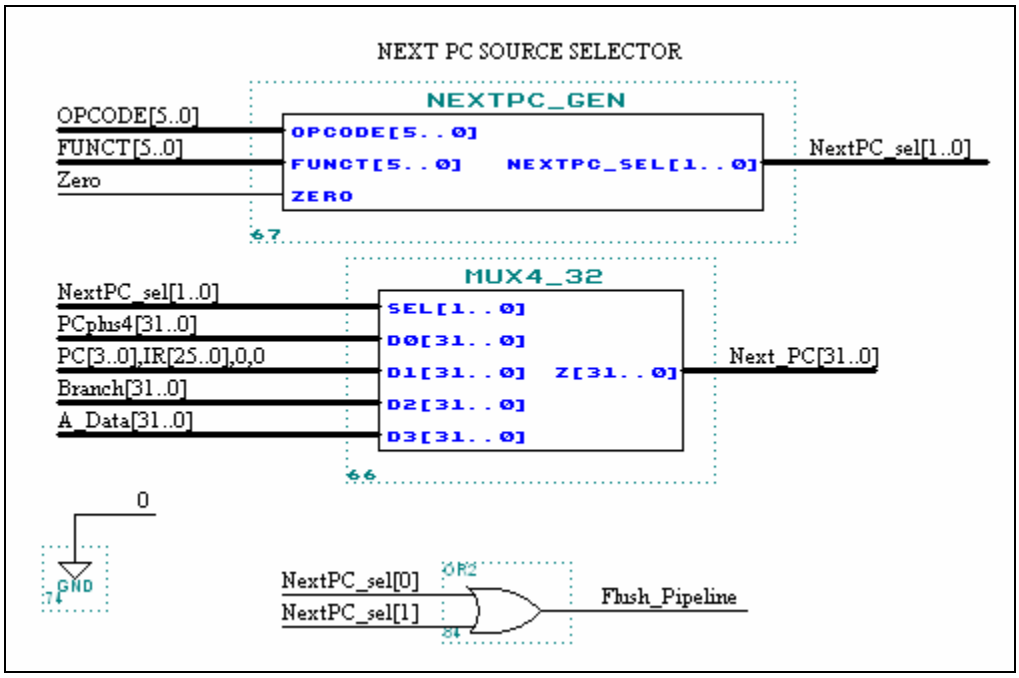

Figure 5: Hardware for issuing *Flush_Pipeline* command

The *Flush_Pipeline* was brought out of the *MEM_Stage* component and used to trigger the *clear* signal on the modified IF/ID, ID/EX, and EX/MEM Pipeline Registers. When a *Flush_Pipeline* is issued, these registers will all clear their outputs—in effect filling those stages of the pipeline with *NOP*s. See Appendix F, Specification Sheet 45 for a description of the modified *MEM* stage component.

System Design and Validation

A. MIPS2000 Complete Design

Once all the aforementioned improvements were made, the individual components of the *MIPS2000* microprocessor were assembled into one graphic design. The assembly of the complete datapath was trivial once all of the individual stages and inter-stage pipelines were modified from their Lab 8 version. The resulting design is shown in Figures 6a, 6b, 6c, and 6d below and on the next pages.

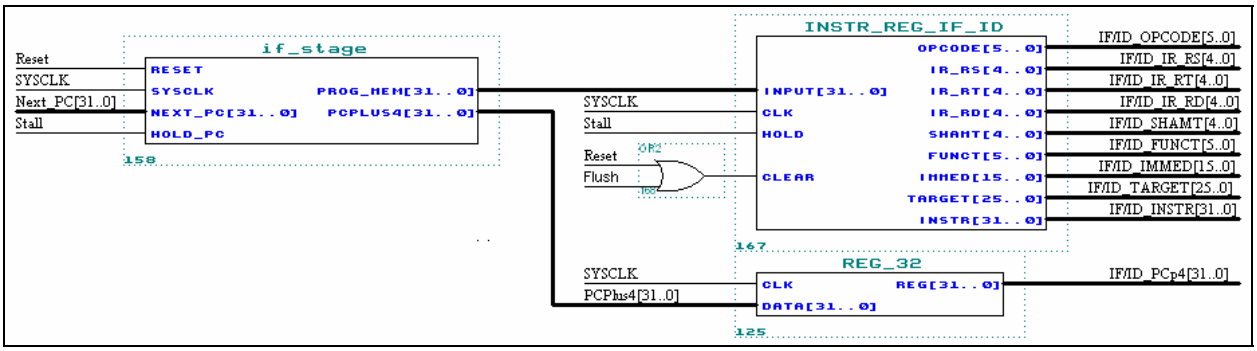

Figure 6a: *Instruction Fetch* Stage

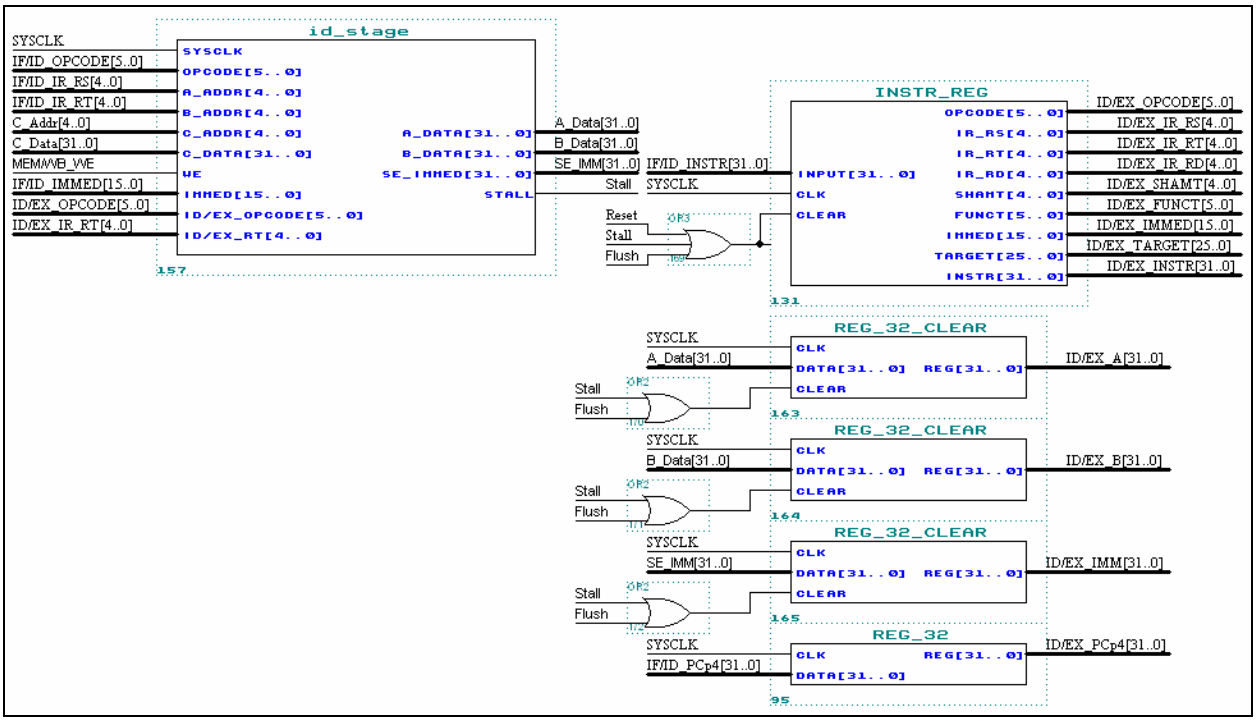

Figure 6b: *Instruction Decode* Stage

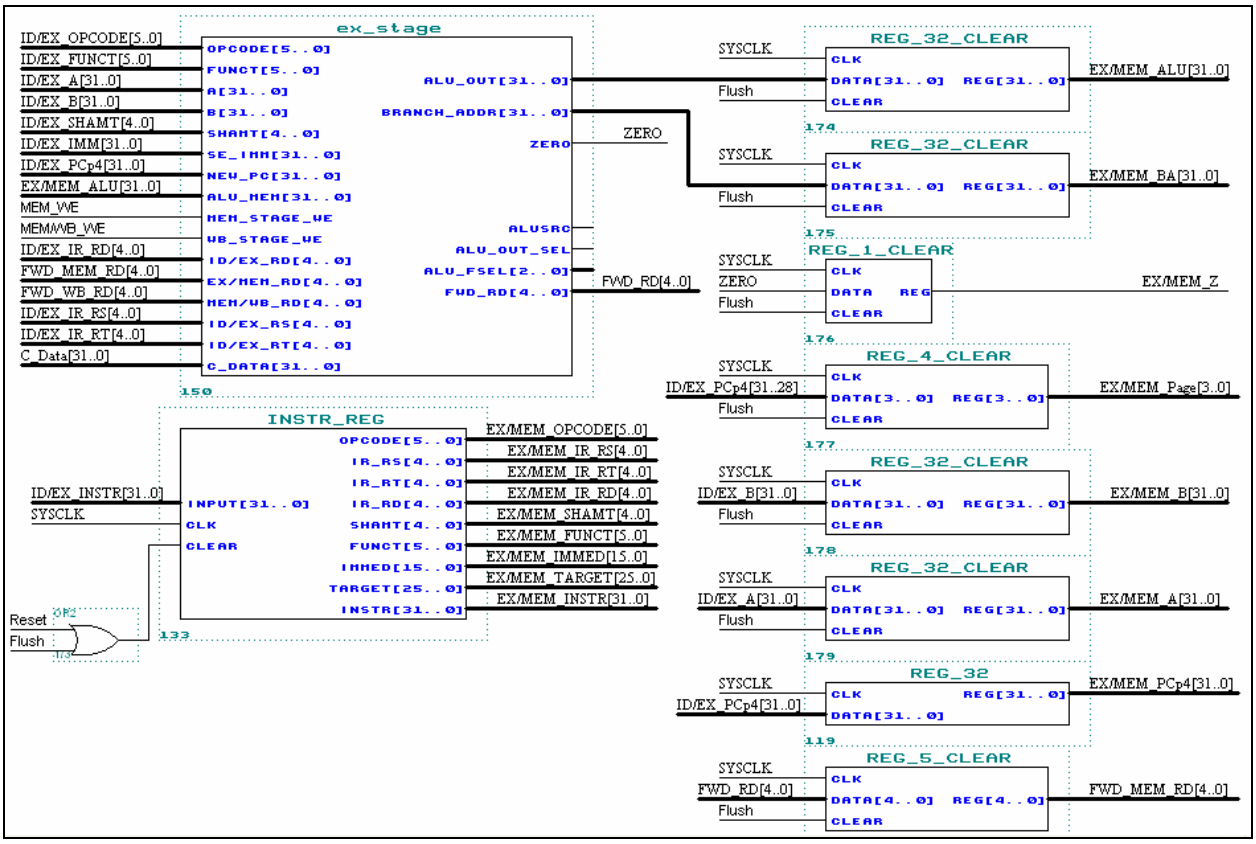

Figure 6c: *Execute* Stage

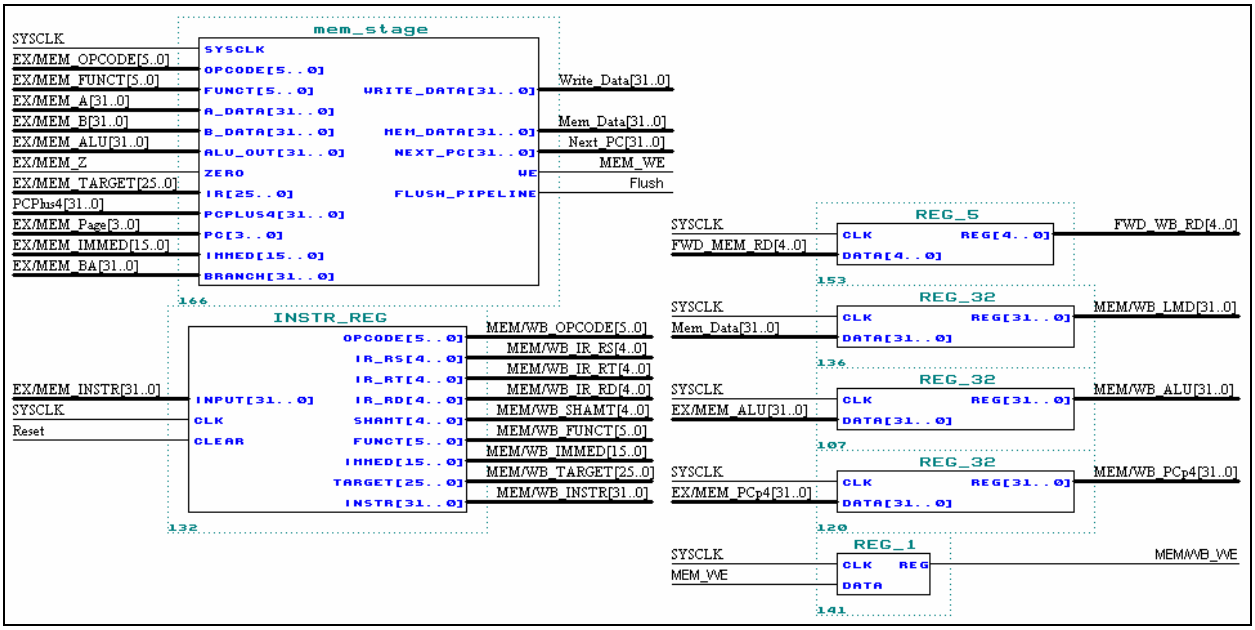

Figure 6d: *Memory* Stage

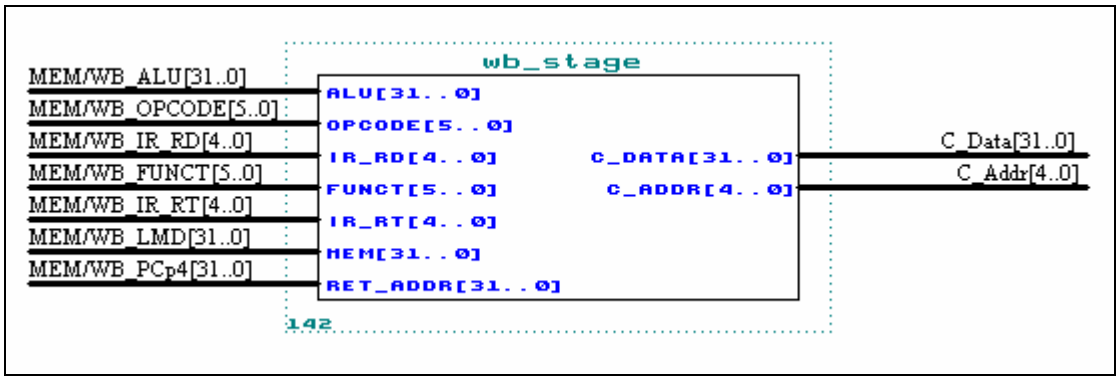

Figure 6e: *Write Back* Stage

This system was functionally compiled to eliminate all errors. See Appendix F, Specification Sheet 46 for a description of the *MIPS2000* component.

B. System Test: Lab9_test.asm

In order to test the operation of the newly constructed *MIPS2000* microprocessor, a test program was written. This program, *Lab9_test.asm*, evaluated a variety of arithmetic, logic, immediate, load, store, branch, and jump functions that created dependencies and hazards. Listing 3 on the next page shows the program developed to evaluate the performance of the *MIPS2000*.

The bolded registers in Listing 3 represent those registers that are involved in a data dependency or hazard. The first data dependency encountered (annotation 1) can be solved with forwarding since the result of the subtraction (computed at the end of the *EX* stage) may be forwarded to the beginning of the *EX* stage for the following addition.

The second data hazard (annotation 2) cannot be solved with forwarding. The *load* into register 4 and the subsequent *and* using register 4's data is impossible to resolve with forwarding. This hazard will be resolved by the Hazard Detection Unit with the run-time insertion of a *NOP* instruction in between the *load* and the *and*. The remaining dependencies in that sequence will be resolved with forwarding.

Throughout the test program there are several *branch* and *jump* instructions. These instructions will employ the "assume branch not taken" scheme when they are encountered in the pipeline. If the branch was not supposed to be taken at run-time, then the subsequent instruction will have already been fed into the pipeline. If the branch *was* supposed to be taken at run-time, then the prediction was wrong and the *MEM* stage of the pipeline will issue a pipeline flush to prevent the wrong instructions from completing their execution.

This program was compiled with *UPASM* and separated into four Memory Initialization Files (MIFs) using the four *S2MIF* programs created in Lab 8. These MIFs were loaded into the Data and Program Memories and the project was simulated using the Max+Plus II Waveform Editor. See Appendix E, Waveform Simulation 15 for the complete results of the simulation.

```
* LAB9_TEST.ASM - Program that demonstrates the features added
*                 in Lab 9.
* Author: Casey T. Morrison, EEL 4713, 4/3/2004

          nolist
          include "mips.mac"
          list

          ORG       $0000

          lw        R5,R0,data1
          lw        R6,R0,data2
          lw        R7,R0,data3
          lw        R8,R0,data4

          sub       R1,R6,R5
          addi      R2,R1,$959B         ; result should be 1

          jal       sub1
done:     beq       R0,R0,done

sub1:     bgez      R1,skip1

          xor       R1,R5,R5  ; set R1 = 0x00000000
          nor       R2,R0,R0  ; set R2 = 0xFFFFFFFF

skip1:    sltu      R3,R2,R1  ; if working properly, R3 = 0x00000001

          lw        R4,R0,data5
          and       R4,R4,R5            ; result should be: R4 = 0x12245008
          or        R4,R4,R6            ; result should be: R4 = 0x1234D0DE
          sub       R4,R4,R6            ; result should be: R4 = 0x00001000
          beq       R4,R8,skip2

          li        R4,$EEEE

skip2:    ori       R4,R4,$0234
          jr        R31

          GFO

          ORG $0080
data1:    dc.l      $12345678
data2:    dc.l      $1234C0DE
data3:    dc.l      $FADE1EEF
data4:    dc.l      $00001000
data5:    dc.l      $BEEFF00D
          end
```

1

2

Listing 3: Lab9_test.asm

Conclusion

A. Summary

The improvements upon the Lab 8 general *MIPS2000* design made in this lab greatly improved the performance of the microprocessor. The hardware was adapted to handle various types of data dependencies and hazards. Because of these modifications, the programmer no longer needs to worry about manually inserting *NOP*s into the program to ensure dependency-free code.

The performance of this machine can improve even more with the adoption of a more effective branch prediction scheme. For example, making use of a Branch History Table or a Branch Prediction Buffer would significantly reduce the frequency of branch mis-predictions. Other, more advanced performance boosts (such as out-of-order execution, branch delay slot, etc.) can also improve the throughput of the machine.

B. Questions

1) *What would be required in the way of a hardware modification to cause the machine to fetch instructions from the "destination address" of a branch if the branch was toward a lower memory address or continue with the instructions immediately following the branch if the branch was toward a higher memory address.*

   To accomplish this task you would need to add a new combinatorial logic block that detects if the branch offset is negative (if the most-significant bit is a one). If this is the case, then an adder dedicated to this task would compute the "destination address" and immediately feed that into the Program Counter. Otherwise, the combinatorial logic block would instruct the Program Counter next address selector to choose the *NextPC* value (PC + 4) as the next Program Counter address.

2) *Would the strategy in question 1 result in a means of branch prediction that was more efficient than the one chosen for our machine in lab? Why? You can use the sorting program for actual numerical evidence.*

   The strategy in question 1 would probably be better than the "assume not taken" strategy implemented in this lab. If a branch is toward a lower address, then this branch is most likely associated with a loop in the code. By nature, loops generally execute more than once, so predicting that the loop branch will be taken—as this scheme does—is a sound strategy that will certainly boost performance. If the destination is a higher address, then the branch is most likely *not* associated with a loop and it would be equally wise to "assume not taken" as it would be to "assume taken."

3) *If a program consisted of a large number of loads and stores, what would the effective instruction execution rate be? It is not necessary to be numerically explicit, just give your analysis of how you would approach the problem given your integer pipeline and a hypothetical program that does a lot of data moving. You should check the assembler language version of the sorting program to see if it is such a program.*

   If a program had a lot of loads and stores, it could potentially run into many data hazards. If the program frequently loaded data into a register then attempted to store that data to memory, this would just as frequently cause a stall in the pipeline due to the data hazard created by the load/store dependency. Thus the instruction execution rate could potentially be 33% slower than if there were no load/store dependencies. This is because for every load/store dependency, there must be one stall. If every load and store created a dependency, this would lead to one stall for every load/store, hence a 33% reduction in execution rate.

Lab No. 10
Casey T. Morrison
EEL 4713 Section 2485 (Spring 2004)
Lab Meeting Date and Time: Monday E1-E3
TA: Grzegorz Cieslewski

I have performed this assignment myself. I have performed this work in accordance with the Lab Rules specifies in 4713 Lab No. 0 and the University of Florida's Academic Honesty manual. On my honor, I have neither given nor received unauthorized aid in doing this assignment.

_____

Introduction

The purpose of this lab was to study the behavior of an instruction cache for the *MIPS2000* microprocessor. Instead of a separate Program Memory, this lab introduced the concept of an Instruction Cache that works in tandem with an all-purpose memory. This two-way set associative cache would theoretically reduce the penalty associated with reading instructions from memory. Since caches are often close in proximity to the microprocessor, their information may be read much faster than main memory's. Therefore with certain schemes caches can greatly reduce the memory read time for a microprocessor.

In general, caches can provide performance benefits by means of *spatial* and *temporal* locality. Spatial locality takes advantage of the concept that if an item in memory is referenced, then items whose addresses are close by will tend to be referenced soon. Temporal locality capitalizes on the fact that if an item in memory is referenced, then it will tend to be referenced again soon. The specific cache structure chosen in this lab will utilize both localities to maximize performance.

Component Design and Validation

A. Instruction Cache Structure

There are various cache structures each with their own advantages. In this lab, a two-way set associative cache with "least-frequently used" replacement strategy was utilized for its spatial and temporal locality advantages. In this Instruction Cache, each cache line is four *MIPS2000* instructions long, and each cache set consists of two lines. With four sets total, the Instruction Cache has an overall size of 128 bytes. Figure 1 shows the structure of the Instruction Cache.

	First Half							Second Half						
Set	**LFU**	**Valid**	**Tag**	**Data0**	**Data1**	**Data2**	**Data3**	**LFU**	**Valid**	**Tag**	**Data0**	**Data1**	**Data2**	**Data3**
Addr[5..4] (2 bits)		*1 bit*	*Addr[31..6] (26 bits)*	*32 bits*	*32 bits*	*32 bits*	*32 bits*		*1 bit*	*Addr[31..6] (26 bits)*	*32 bits*	*32 bits*	*32 bits*	*32 bits*
0														
1														
2														
3														

Figure 1: Instruction Cache structure

With this structure in mind, the *MIPSsim* header file (*mipssim.h*) was modified to support the aforementioned cache structure. In doing so, several requirements were imposed so that the resulting structure would be compatible with the operation and function of the cache. The strategy developed was to separate the data aspects of the cache from the address aspects. In addition, the data portion of the cache was further divided into *instruction* data and *status* data.

Several structures were created to help map the *MIPS2000* addresses into cache addresses. The least significant two bits of every *MIPS2000* instruction address are always zero due to the byte-addressable nature of the system memory. The next two bits (bits two and three) were used to address the four instruction words that each cache line contained. The two subsequent bits (bits

four and five) were used to distinguish between the four sets that the cache contained. Finally, the remaining 26 bits were used as the tag for the cache entry. As a result of this strategy, the following C structures were created.

```
#ifdef B_ENDIAN
struct block
        {
                unsigned block_addr: 30;
                unsigned block_off: 2;
        };
#else
struct block
        {
                unsigned block_off: 2;
                unsigned block_addr: 30;
        };
#endif
/*
          31                                                    2 1         0
         +-------------------------------------------------------+-----------+
         |                     block_addr                         | block_off |
         +-------------------------------------------------------+-----------+ */

#ifdef B_ENDIAN
struct slot
        {
                unsigned line_addr: 28;
                unsigned word_index: 2;
                unsigned block_off: 2;
        };
#else
struct slot
        {
                unsigned block_off: 2;
                unsigned word_index: 2;
                unsigned line_addr: 28;
        };
#endif
/*
          31                                       4 3          2 1         0
         +-----------------------------------------+------------+-----------+
         |               line_addr                  | word_index | block_off |
         +-----------------------------------------+------------+-----------+ */

#ifdef B_ENDIAN
struct line
        {
                unsigned tag: 26;
                unsigned set_index: 2;
                unsigned word_index: 2;
                unsigned block_off: 2;
        };
#else
struct line
        {
                unsigned block_off: 2;
                unsigned word_index: 2;
                unsigned set_index: 2;
                unsigned tag: 26;
        };
#endif
/*
          31                           6 5          4 3          2 1         0
         +-----------------------------+------------+------------+-----------+
         |            tag               | set_index | word_index | block_off |
         +-----------------------------+------------+------------+-----------+ */
```

Listing 1: C Structures for Instruction Cache

These structures were united to form the cache address union. Listing 2 below shows the resulting C Union.

```
union cache_address
       {
              struct word addr;
              struct block b_addr;
              struct slot s_addr;
              struct line l_addr;
       }cache_addr;
/*
cache_addr.b_addr.block_off-----------------------------------------+
cache_addr.b_addr.block_addr--------+                                |
          31                        |                          2 1   |     0
         +--------------------------v--------------------------+----v------+
         |                     block_addr                       | block_off |
         +--------------------------------------+-------------+-----------+
         |               line_addr              | word_index  | block_off |
         +--------------------------------------+-------------+-----------+
         |            tag             | set_index | word_index | block_off |
         +----------------------------+--+--------+------+----+-----------+
         |  upper_byte   |  upmid_byte   |  lowmid_byte   |   lower_byte   |
         +-------^-------+------^--------+--------^-------+-------^--------+
                 |              |                 |               |
                 +---------+    |                 |               |
                           |    |                 |               |
cache_addr.addr.upper_byte-+    |                 |               |
cache_addr.addr.upmid_byte------+                 |               |
cache_addr.addr.lowmid_byte-----------------------+               |
cache_addr.addr.lower_byte----------------------------------------+
*/
```

Listing 2: C Union for *Cache_Address*

With the cache address structure in place, the cache data components were created. The space in the cache for the actual instruction words was represented as a three-dimensional array. The first index of the array addresses the set, the second index addresses the "way" or half of the cache, and the third index addresses the specific word within the four-word line. The three-dimensional illustration of this structure is shown below in Figure 2.

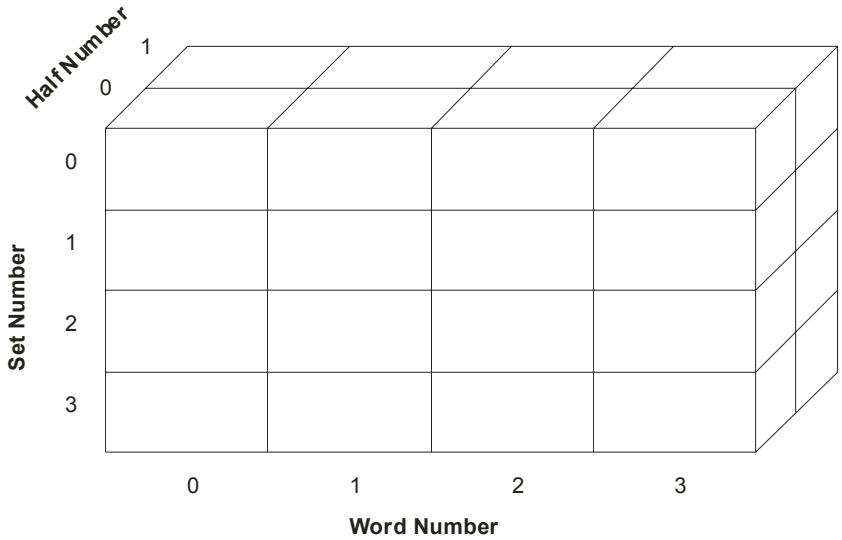

Figure 2: Instruction Cache data structure

The other component of the cache data is the status information. Each cache entry has tag bits, a valid bit, and a frequency associated with it. The cache status was organized into a two-dimensional array. The first index of the array (first dimension) is used to distinguish between the four sets in the cache. The second index (second dimension) is used to select the tag bits, valid bit, or frequency of the two entries (one in the first half of the array, one in the second). Figure 3 below illustrated the structure of the cache status array.

Set #	Freq. 0	Freq. 1	Tag 0	Tag 1	Valid 0	Valid 1
0						
1						
2						
3						

Figure 3: Instruction Cache status array

See Appendix A, Program 10 for the complete *mipssim.h* code.

B. Instruction Cache Operation

The operation of the Instruction Cache was described in the *MIPS2000* simulator (*mipssim.c*). For each instruction fetch, the *accessCache()* method was called to retrieve the desired instruction word from the cache. In the event that the instruction was not in the cache, a cache miss was recorded and the instruction (along with the three instructions surrounding it) were retrieved from main memory and placed in the cache. If the desired instruction *was* in the cache, then a cache hit was recorded and the instruction was returned.

When cache misses occur and the cache must be updated with the desired information, a decision must often be made regarding the replacement strategy. Since this cache is two-way set associative, there can be two sets of four instruction words that map to the same cache line. Thus when a third group of four instruction words is to be written to an already full cache set, one of the two groups already in the cache must be replaced. In this lab, the "Least-Frequently Used" (LFU) cache replacement strategy was employed. When a cache entry must be replaced, the entry that is least-frequently used is the one selected for replacement.

Keeping track of the frequencies was left to the *accessCache()* method. When an existing entry was accessed, its frequency was incremented in the cache status array. When a new entry was placed in the cache, it was assigned a frequency of one, meaning that it had been accessed one time.

The primary role of the *accessCache()* method is to determine if the desired instruction is in the cache (cache hit) or if it is not (cache miss). This is done by comparing the upper 26 bits of the address with the tag bits of the two cache lines in the set specified by bits four and five of the address. If this comparison is successful, and the data in the cache is valid, then a cache hit has occurred. If the comparison is negative, or if the data is not valid, then a cache miss has occurred, and the desired instruction must be fetched from main memory. See Appendix A, Program 11 for the C code for the *MIPS2000* simulator complete with cache simulation.

System Design and Validation

A. MIPS2000 Complete Simulator

Once the header file (*mipssim.h*) and the C file (*mipssim.c*) were modified to support a two-way set associative Instruction Cache, the project was compiled and rid of all errors. The simulator was designed to keep track of the overall number of cache hits and cache misses during the execution of a program. In addition, the cache hit rate and cache miss rate were calculated for additional performance statistics.

B. System Test 1: Bubble Sort

In order to test the operation of the *MIPS2000* simulator complete with Instruction Cache, several test programs were run to evaluate performance and accuracy. The first test program was a precompiled version of Bubble Sort. This sorting algorithm repeatedly compares two adjacent numbers in an array and swaps them if they are out of sorted order.

After simulating the Bubble Sort program on the *MIPS2000* simulator, it was determined that the newly implemented cache structure performed as desired and even offered some promising results in terms of the hit/miss rate. The total hit rate during the execution of the Bubble Sort program was 74.5% (589 out of 791 cycles) and the miss rate was 25.5% (202 out of 791 cycles). This indicated that the structure chosen for the Instruction Cache was beneficial to the performance of the machine, since an overwhelming majority of the instruction fetches resulted in cache hits. See Appendix D, Simulation 6 for an abbreviated listing of this simulation.

C. System Test 2: Insertion Sort

To further test the behavior of the *MIPS2000* simulator, another sorting program was run. The precompiled Insertion Sort program was simulated and analyzed. As with the Bubble Sort, a favorable hit/miss rate resulted from the simulation. This time, 67.5% of cache accesses were hits, and only 32.5% were misses. In addition , the function of the program (to sort data) was not hindered by the addition of the cache function. See Appendix D, Simulation 7 for an abbreviated listing of this simulation.

Conclusion

A. Summary

The two-way set associative Instruction Cache implemented in this lab can prove to bring great performance benefits to the *MIPS2000* microprocessor. By taking advantage of spatial and temporal locality, this cache structure saw hit rates of more than 75%. With fetches from main memory being much more costly than fetches from the cache, this hit rate translates into a real performance boost.

More performance benefits may be achieved by increasing the associativity of the cache. With more associativity, temporal locality can be enhanced and the overall cache performance may increase. However, with more associativity comes a more complex replacement scheme. In addition, more hardware must be added to accomplish the comparisons necessary. In addition, the benefits of spatial locality may be increased with a larger cache block size. This entails storing more instruction words into the cache each time an instruction is fetched from main memory as a result of a cache miss. Although this would increase performance, it would also involve additional hardware.

Cache schemes can be implemented with data memory as well as program memory. When doing this, a new level of complexity is added. Strategies must be developed to handle situations in which data is *written* to memory. With the right strategy, however, the performance benefits can heavily outweigh the hardware/complexity additions.

B. Questions

1. *What interesting things did you notice about the instructions following the load and branch instructions produced by the C compiler? Why were there sometimes instructions instead of NOPs following branches? Could this code run on our VHDL simulator and why? If not, what changes would we have to make to the VHDL simulator to execute this code?*

   The C compiler inserted instructions following loads and branches in accordance with the "branch delay slot" and "load delay slot" schemes. In other words, the compiler recognized the need to stall after a load or branch, and it inserted an independent instruction after a load or a branch so that work could still be done in these situations. If the compiler could not find a satisfactory independent instruction, it would insert a *NOP* instead.

   This code could not run on the existing VHDL simulator because that design uses the "assume not taken" strategy instead of the "branch delay slot" strategy. In the existing VHDL simulator the instruction following a branch will begin execution, but could be cleared if it is discovered that the branch was supposed to be taken. Thus the instruction in the branch delay slot will not always get executed.

In order to execute this code on the VHDL simulator, the next address decision must be moved forward to the *EX* stage so that the decision to branch or not can be made immediately after the branch delay slot instruction has entered the pipeline. Thus the correct execution path can be decided in time, and there would be no need to flush the pipeline.

2. *Using a 4 to 1 ratio between CPU speed and main memory speed, estimate, using methods shown in reference 1, the cache performance of the final MIPSsim as it executes the sort program. Remember that the NOPs introduced by the compiler must still be counted since they compensate for shortcomings in the integer pipeline.*

   After executing the Bubble Sort program, the following performance statistics were observed.

Cache type	Total cycles	# Hits	# Misses	Cost	Performance Ratio
MIPS2000	791	598	202	1397	$\frac{1397}{791} = 1.766$
Perfect	791	791	0	791	

Figure 4: Cache performance comparison

3. *If the data cache for the MIPS2000 was added, what new behaviors were needed in the implementation?*

   In order to implement a data cache, a strategy must be developed in order to handle memory writes. *Write Back* and *Write Through* are two common strategies employed to handle such situations.

4. *The authors of reference 1 suggested that the cost of a cache-miss should affect the choice of data replacement algorithms. Is the cost of a cache-miss for our cache(s) high or low compared with the cost of a "page-fault" in the computer's main memory? Hint: scan section 7.4 in reference 1. Realize that main memory is just a cache for the swapping area on the hard drive. What is the typical access time to data on the hard drive? How does this compare with the typical access time to data in main memory, in cache? The answer is in the speed ratios.*

   A page fault costs significantly more than a cache miss. The typical access time for RAM is 5 to 120 ns, while the typical access time for the hard disk is 10 to 20 million ns. Thus the cost for a page fault is between 167,000- and 4,000,000-times greater than the cost of a cache miss.