Lab No. 0
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 goal of this lab was to implement the *Sweet16* Instruction Set Architecture (ISA) in the C programming language. In doing so, the Fetch/Decode/Execute process that is at the heart of the Instruction Set Processor will be examined in depth. By programming the *Sweet16* ISA in C, a simulator can be made that replicates the behavior of the *Sweet16* microprocessor with specific input. This simulator will become an "executable problem statement" which represents that an understanding has been reached regarding the problem specification.

The program that was designed to accomplish the given task is *sw16sim.c*. This program will take source files (.s) compiled by an adapted assembly language compiler (UPASM) and produce print statements which indicate the inner workings of the simulated processor. The program tries to replicate the internal processes of the *Sweet16* as closely as possible so as to more accurately predict the outcome of the assembly language programs that will be run on the *Sweet16*. In the end, the simulator will serve as a platform on which we may analyze the details of the implementation of this processor.

Component Design and Validation

In designing a simulator for the *Sweet16* microprocessor it was imperative that the program reproduce the internal architecture as closely as possible. The program *sw16sim.c*, designed in large part by Dr. Michel A. Lynch, deals in structures and objects that mirror the components of the *Sweet16* microprocessor.

The basic implementation of every instruction in the *Sweet16* ISA involves three steps: Fetch, Decode, and Execute. The Fetch portion of this cycle is the same for each instruction. In essence, the first, and possibly only, instruction word is "fetched" by retrieving the data at the memory location pointed to by the Program Counter (PC). This is accomplished in the simulator by addressing an array of bytes (the memory) with the PC. The instruction word is loaded into a temporary register one byte at a time (something unlike the actual implementation of *Sweet16*).

In the decode portion of the code, the upper byte of the instruction is compared against the opcodes in the instruction set. When a match is found, the execution path of the program is redirected to the portion of code that handles that particular instruction. Once this is done, the operands are fetched in preparation for the execution phase. Based on the type of address mode that the particular instruction employs, a subroutine is called to retrieve the operands and store them into temporary registers. For example, the ADDI instruction makes use of the Immediate address mode, and thus the ADDI program segment calls the *Immediate()* subroutine to fetch the two words that are to be added.

Finally, with the operands waiting in temporary registers, subroutines may be called to handle the execution of the instruction. Examples of this include subroutines to add, subtract, bitwise and, bitwise or, etc. Included in the execution phase is the setting/clearing of the status flags.

Subroutines were created to handle the two flag update requirements. Once the execution is complete, the program repeats the process, grabbing a new instruction from memory.
The flowchart of the main routine in the *Sweet16* simulator is shown below in Figure 1. It illustrates the Fetch/Decode/Execute nature of *main()*.

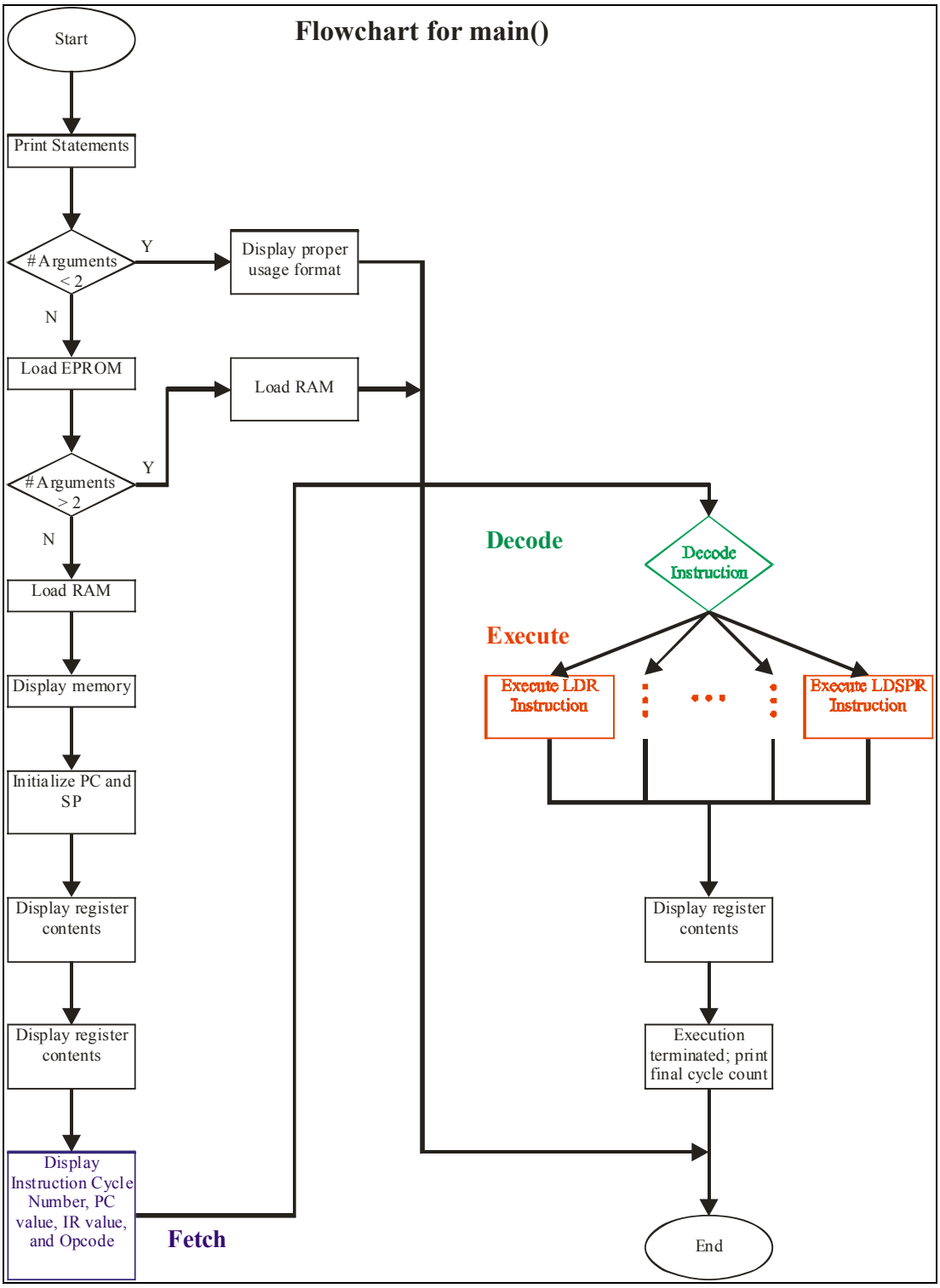

Figure 1: Flowchart for main method

All instructions that have the same address mode obtain their operands in the same manner. Using this principle, the *main()* subroutine was greatly simplified with the use of subroutines that retrieved operands based on the address mode. For example, the subroutine for the absolute address mode (shown below in Listing 1) uses the data in the second word of the instruction to point to the data that is to be used as one of the operands.

```
/*
        Absolute address mode function - Big Endian
*/
void absolute()
{
        df.db.upper_byte = mem[*pc]; incPC;
        df.db.lower_byte = mem[*pc]; incPC;
        printf("   ABSOLUTE Data Address: %04X,", (unsigned)df.d.data);
        temp = df.d.data;
        df.db.upper_byte = mem[(unsigned)temp];
        df.db.lower_byte = mem[(unsigned)temp + 1];
        temp1 = reg[(int)ir.instr.r1];
        temp2 = df.d.data;
        printf(" Data: %04X\n", (unsigned)df.d.data);
}
```

Listing 1: Absolute address mode handler

Any instruction that employs the absolute address mode will perform some data manipulation on the contents of *r1* and the data pointed to by the second word of the instruction. After calling this subroutine, those data will be stored in the first and second temporary registers, respectively.

With a standard location for operands, like temporary registers one and two, operations can then be performed on this data in a generalized manner. For example, the *add()* subroutine (shown below in Listing 2) is employed by all instructions that add without carry.

```
/*
        add() is  used to complete addition operations with no carry input
*/
void add()
{
        temp = temp1 + temp2;  //data is stored in temp1 and temp2
        reg[(int)ir.instr.r1] = temp & WORD_MASK;    /* write-back the result into the reg array
*/
        printf("    ADD:  reg[%d]  =  %04X  +  %04X  =  %04X\n",(int)ir.instr.r1,  temp1,  temp2,
reg[(int)ir.instr.r1] );
}
```

Listing 2: Addition method

This subroutine assumes that the data is already stored in the temporary registers. Once a procedure like this is called and executed, it is often necessary to call a special subroutine to set (or clear) the appropriate status flags. The *logic_flags()* function (shown in Listing 3) is one such subroutine that handles the updating of the status register. Often times, this is the last step in the execution of an instruction. Once this is complete, *main()* will repeat the Fetch/Decode/Execute process.

```
/*
    logic_flags() is used to set the logical flags
*/
void logic_flags()
{
//  zero flag: z
    if( temp == 0) {flags.z = 1;} else {flags.z = 0;}

//  sign flag: n
    if((temp & SIGN_MASK) == 0) {flags.s = 0;} else {flags.s = 1;}

    printf("\n  Logic Flags: c=%01X, v=%01X, s=%01X, z=%01X\n", flags.c, flags.v, flags.s,
flags.z);
}
```

Listing 3: Logic Flags method

As new instructions were being implemented in the simulator, they were individually tested with an assembly program. For example, the "Store with Indexed Addressing" command (*STAX*) was implemented as follows in the *Sweet16* simulator.

```
case STAX:                                  // Indexed with Offset address mode
       index_w_offset();
       mem[(unsigned)df.d.data] = (reg[(int)ir.instr.r1] >> 8);
       mem[(unsigned)df.d.data + 1] = reg[(int)ir.instr.r1];
       printf("   Storing $%04X at memory location $%04X\n",reg[(int)ir.instr.r1],
              (unsigned)df.d.data);
       break;
```

Listing 4: Store with Indexed Addressing instruction

This instruction required the use of an Indexed Addressing subroutine, *index_w_offset()*. Shown in Listing 5, this subroutine uses an offset stored in *r2* as well as a base address located in the second instruction word to address the data used in the calculation.

```
/*
    Indexed w/offset address mode function
*/
void index_w_offset()
{
    df.db.upper_byte = mem[*pc]; incPC;
    df.db.lower_byte = mem[*pc]; incPC;
    printf("   Base Address: %02X%02X\n", df.db.upper_byte, df.db.lower_byte);
    printf("   Offset: %04X\n", reg[(int)ir.instr.r2]);
    df.d.data = df.d.data + reg[(int)ir.instr.r2];
    temp2 = (mem[df.d.data] << 8) | mem[df.d.data + 1];
    printf("   Target Address: $%04X\n", df.d.data);
    printf("   Data at &%04X: $%04X\n", df.d.data, temp2);
}
```

Listing 5: Indexed with offset address mode handler

To test this particular instruction, an assembly language program was written (see Listing 6 on the next page). This program loads registers R0 with data and R1 with an offset of eight, and then it calls the STAX instruction. The result is that 0x0002 is stored at address DATA + 8 (see Appendix D, Simulation 1 for the results of the simulation).

```
* STAX_test.ASM
* By Casey T. Morrison
* Purpose: To test the STAX instruction

            NOLIST
            INCLUDE "sweet16.mac"
            LIST

            ORG     $0000

            LDI     R0,$0002
            LDI     R1,$0008
            STAX    R0,R1,DATA
            GFO

DATA:       dc.w        $1111
            dc.w        $2222
            dc.w        $3333
            dc.w        $4444
            dc.w        $5555       * Data + 8

            END
```

Listing 6: Test program for *STAX* method

Testing procedures such as this one were employed for each instruction that was added to the simulator. This ensured that the simulator worked properly throughout the design process.

System Design and Validation

The system on which this lab focuses consists of three main parts or stages (see Figure 2 below).

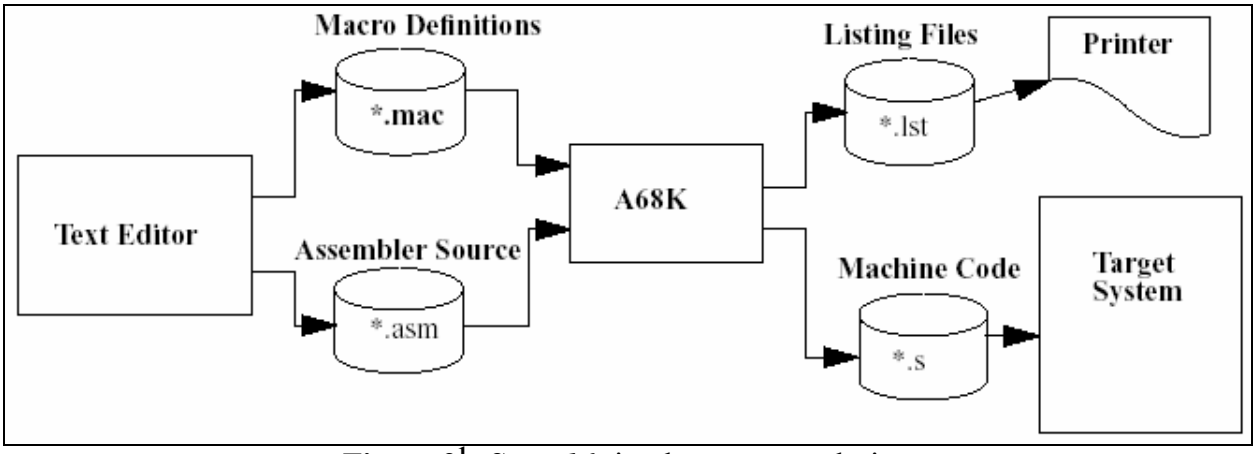

Figure 2[1]**:** *Sweet16* simulator system design

The first stage begins with the creation of an assembly language program using the instruction set of the *Sweet16* microprocessor. Designing this program is a crucial part of the entire process. An example of such an assembly language program is *mulrom.asm* shown on the next page (it may also be found in Appendix C, Program 1). This program utilizes the instructions available on the *Sweet16* in order to accomplish unsigned 16x16 multiplication.

[1] "Assemblers and Related Tools," Dr. Michel A. Lynch, http://www.hcs.ufl.edu/~radlinsk/eel4713/mod/resource/view.php?id=10

Assembly language programs, like *mulrom.asm*, are then assembled using the A68K assembler which was adapted from the Motorola UPASM assembler. This second stage of the process requires the use of the *sweet16.mac* file in order to interpret the instruction mnemonics. In this file, several macros were defined to initialize memory based on the instructions contained in the *.asm* source file. The assembler also takes advantage of the different address modes built-in to the *Sweet16* by using subroutines to make the code more reusable and functional.

```
* MULROM.ASM - Program that calls and tests a 16 bit MULTIPLY subroutine
*                Orged in ROM ($0000)
* Subroutine UMUL - unsigned 16 X 16 multiplication
*     * Multiplier is in R1, multiplicand is in R0 on entry
*     * Product (32 bits) is returned with most significant half in R0
*       and least significant half in R1.
* Other registers used: R2 contains Multiplicand during the operation
*                     : R3 contains the shift count
* Author: Dr M Lynch, EEL 4713, 1/7/2003
* Used by: Casey Morrison, EEL 4713, 1/15/04

        NOLIST
        INCLUDE "sweet16.mac"
        LIST

STACK   EQU     $100
COUNT   EQU     16
        ORG     $0000

*          Initialize Stack Pointer
 LDI       R2,STACK
 LDSPR     R2
*       Test program to multiply two numbers in R0 and R1
        LDI    R0,$FFFF
        LDI    R1,$FFFF
        CALL   UMUL
        GFO

*        Multiplication subroutine
UMUL:
        LDR    R2,R0    *Place multiplicand in R2 for duration of UMUL
        LDI    R0,0     *Clear upper half of product area
        LDI    R3,COUNT *Place shift count in R3
        CLRC                   *Clear carry flag
UMUL1:
        RORC    R0,1    *Rotate Product MSW right with LS bit into carry
        RORC    R1,1    *Rotate Multiplier right with LS bit into carry
        BCC     UMUL2   *Don't add Multiplicand if bit of multiplier is zero
        ADDR    R0,R2   *Add multiplicand to MSW of Product - note carry out
UMUL2:
        LSUBI   R3,1    *Subtract one from shift count - note carry is not generated
        BPL     UMUL1   *Go until shift count is equal zero
        RET

        END
```

Listing 7: *Mulrom.asm* program

Once the program is assembled, it is loaded into the test system in the form of a source (*.s*) file. The third stage, or Target System as it is referred to in Figure 2, consists of the *Sweet16* simulator mentioned earlier. This program takes *.s* files as input and produces text representing the response of the *Sweet16* microprocessor to that input.

For example, the test program *mulrom.asm* behaves in accordance with the flowchart in Figure 3 when run on the modified simulator. It uses a series of rotates, sums, and loops in order to accomplish the multiplication of two 16-bit numbers. The simulator produces a text file when running *mulrom.s* (see Appendix D, Simulation 2 for the simulation results). As is noted in Appendix D, the simulation of *mulrom.s* resulted in the correct product value being stored in R0 and R1.

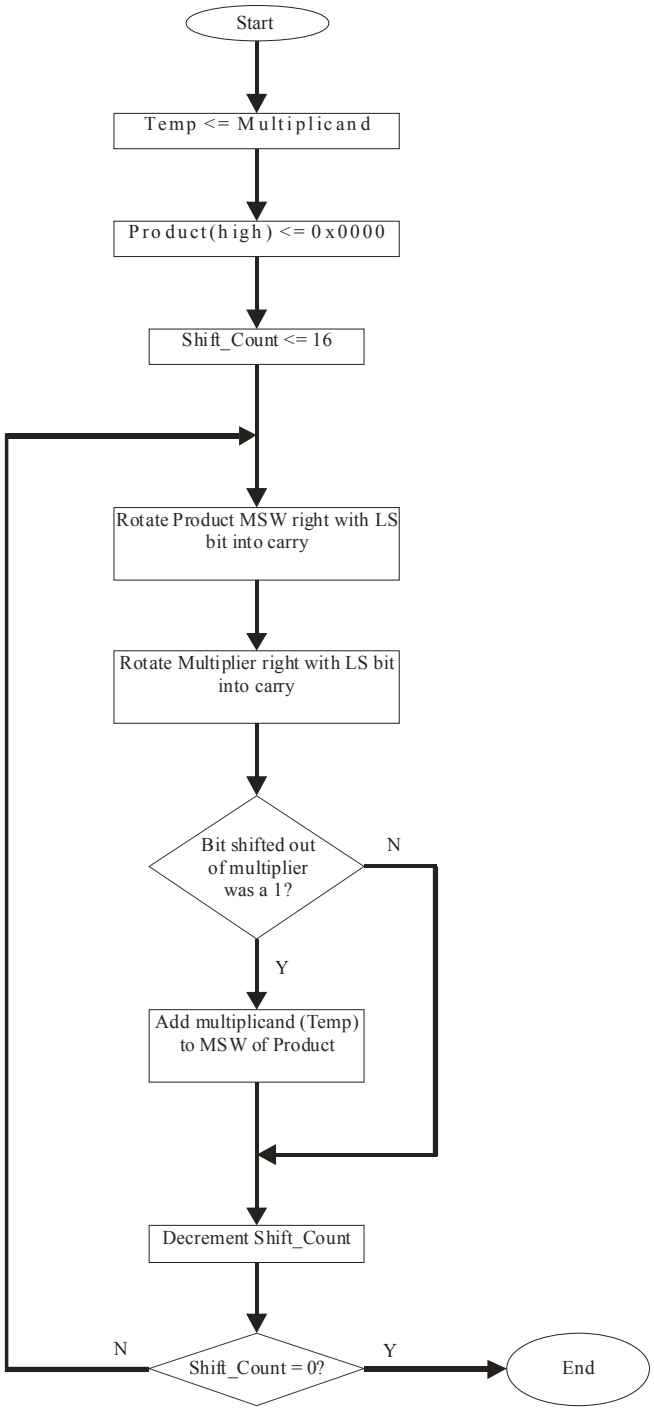

Figure 3: Flowchart for *mulrom.asm*

This assembly program was simplified with the creation of a UMUL macro that performed the 16x16 unsigned multiplication (see Listing 8 below or Appendix C, Program 2 for the code). The macro basically inserts portions of the original code in place of the macro calls. With this macro definition, assembly programs may make use of unsigned multiplication simply by calling UMUL. Like *mulrom.asm*, this assembly program was simulated and shown to work (see Appendix D, Simulation 3 for the simulation results).

```
* MULROM_MACRO.ASM - Program that calls and tests a 16 bit MULTIPLY subroutine
*                Orged in ROM ($0000)
* Subroutine UMUL - unsigned 16 X 16 multiplication
*      * Multiplier is in R1, multiplicand is in R0 on entry
*      * Product (32 bits) is returned with most significant half in R0
*        and least significant half in R1.
* Other registers used: R2 contains Multiplicand during the operation
*                     : R3 contains the shift count
* Author: Dr M Lynch, EEL 4713, 1/7/2003
* Edited by: Casey Morrison, EEL4713, 1/15/04


        NOLIST
        INCLUDE "sweet16.mac"
        LIST

STACK   EQU  $100
        ORG  $0000

*   Initialize Stack Pointer
    LDI       R2,STACK
    LDSPR     R2

* MACRO DEFINITION ***********************************************************
UMUL    macro \1,\2

*       Test program to multiply two numbers in R0 and R1
        LDI     R0,\1
        LDI     R1,\2

*        Multiplication subroutine
        LDR     R2,R0   *Place multiplicand in R2 for duration of UMUL
        LDI     R0,0    *Clear upper half of product area
        LDI     R3,16   *Place shift count in R3
        CLRC            *Clear carry flag
UMUL1:
        RORC    R0,1    *Rotate Product MSW right with LS bit into carry
        RORC    R1,1    *Rotate Multiplier right with LS bit into carry
        BCC     UMUL2   *Don't add Multiplicand if bit of multiplier is zero
        ADDR    R0,R2   *Add multiplicand to MSW of Product - note carry out
UMUL2:
        LSUBI   R3,1    *Subtract one from shift count - note carry is not generated
        BPL     UMUL1   *Go until shift count is equal zero
       endm
*******************************************************************************

* UTILIZATION OF MACRO
   UMUL      $11,$42
   GFO
```

Listing 8: *Mulrom* equivalent macro

The simulator was modified at length to include all the subroutines necessary to implement all of the instructions (see Appendix A, Program 1 for the complete C code). This simulator proved to execute the source files as desired.

Conclusion

A. Summary

In this lab a system was designed to replicate the behavior of the *Sweet16* microprocessor. Assembly language programs were assembled using a modified assembler that utilized macros to encode each instruction mnemonic into an instruction word(s). The resulting sequence of instruction words was virtually “loaded” into a memory (an array structure in C). The *Sweet16* simulator, *sw16sim.c*, then initiated the Fetch/Decode/Execute process to complete the desired program. This simulator was designed with functional units that enabled several instructions to

be implemented with minimal code. Whenever possible, subroutines were created to simplify the implementation of each individual instruction. In the end, a complete simulator was created that very nearly replicated the internal processes of the *Sweet16* microprocessor.

B. Questions

1. *On a listing segment of the file sw16sim.c, identify the important features specified in the Instruction Fetch/Decode/Execute Flowchart.*

   The following portion of *sw16sim.c* demonstrates the important features of the Fetch/Decode/Execute cycle of this simulator.

```
for (i=0; i<MAXREPS; i++) /* run the main simulation loop MAXREPS times */
   {
      printf("Instruction Cycle Number: %d\n",i);
      printf("\n PC = %04X: ",*pc);

      ir.db.upper_byte = mem[*pc]; incPC;
      ir.db.lower_byte = mem[*pc]; incPC;
      printf("upper_byte: %02X, lower_byte: %02X,",ir.db.upper_byte,
             ir.db.lower_byte);
      printf("  opcode: %02X, %s, r1: %01X, r2: %01X\n",ir.instr.opcode,
             opmnem[(int)ir.instr.opcode], ir.instr.r1, ir.instr.r2);

      switch((int)ir.instr.opcode){
// 16-bit Integer Data Type - Basic Instruction Set
//  Load
    case LDR:                                   // RR address mode
       reg_reg();
       load();
    break;
```


Listing 9: Fetch/Decode/Execute cycle of the *Sweet16* simulator

2. *Account for the variation in coding needed to accommodate the "Little-Endian" host as compared to the version of sw16sim.c, which was prepared for a "Big- Endian" host.*

   The adjustments that need to be made in order to accommodate for a Little-Endian host computer as opposed to a Big-Endian host are associated with the definition of structures. In the *sweet16_le_host.h* file the "word" structure is defined to be the combination of a "lower byte" and an "upper byte." In *sweet16_be_host.h*, however, a "word" is the combination of an "upper byte" and a "lower byte," in that order. This semantic difference simply accounts for the Endian-ness of the host machine on which the simulator will run.

3. *Where are the values for the symbolic labels used in each case statement initialized? Give the actual locations of the definition and an example.*

   The values for the symbolic labels used in each case statement within *sw16sim.c* are initialized in the *sweet16_le_host.h* and *sweet16_be_host.h* files. For example, the Branch PC-Relative Instructions are defined as shown in Listing 10 in the aforementioned files.

```
//       Define the Branch PC-Relative Instructions
//        EA = PC + Sign_Extended_offset
//        Bcc Offset, if f{cc} then EA --> PC + offset
#define B       0x13
#define CALL    0x14
#define JMP     0x15
#define CALLX   0x16
#define JMPX    0x17
#define STA     0x18
#define STAX    0x19
#define LAA     0x1A
#define LAX     0x1B
```

Listing 10: Symbol definitions

4. *Is the statement "incPC" a function? If not, what is it? Where and what is its definition? What does it accomplish?*

   The keyword *incPC* used throughout *sw16sim.c* is a macro, rather than a function, that is defined in the *sweet16_le_host.h* and *sweet16_be_host.h* files (see Listing 11). This macro serves to increment the Program Counter (PC) so that it points to the next data byte.

```
/* C Macros                    */

#define incPC *pc = (unsigned) WORD_MASK & ((*pc) + 1)
#define incSP *sp = (unsigned) WORD_MASK & ((*sp) + 1)
#define decSP *sp = (unsigned) WORD_MASK & ((*sp) - 1)
```

Listing 11: Macro definition of *incPC*, *incSP*, and *decSP*

5. *Describe how the union of structures was used to give direct addressing to the register bit-field in an instruction, while the data that was moved was transferred as upper and lower bytes on the data bus.*

   The *sweet16_le_host.h* and *sweet16_be_host.h* files employ a Union of Structures in order to be able to access the data retrieved from memory in different ways. The *inst_flow* and *data_flow* Unions allow 16-bit chunks of data to be accessed in 16-, 8- and 4-bit chunks when appropriate. This is to accommodate for the need to access 16-bit data, 8-bit opcodes, and 4-bit register fields. By combining same-length, variably-partitioned structures (like the *word*, *inst_reg*, and *databus*) into one Union, this goal was accomplished.

6. *The Branch on Condition and conditional call and return instructions use the function eval_cc() to determine the success of a condition. What is the numeric value of the symbol "CC" in that function? What is returned for the "CC" condition if the flag vector was {1,1,0,1}? What type of number was the programmer using when he interpreted the flags using the "LT" condition? What is returned for the "LT" condition if the flag vector was {1,1,0,1}?*

   The symbol "CC" used in *eval_cc()* function evaluates to 0x0 as defined in the *sweet16_le_host.h* and *sweet16_be_host.h* files. If the flag vector passed to this function was {1,1,0,1}, then the return for the "CC" condition would be False or 0. When using the "LT"

or “less than” condition, the programmer is explicitly dealing with signed numbers. If the flag vector passed to this function was {1,1,0,1}, then the return for the “LT” condition would be False or 0.

7. *If the number of instructions in the source program for the first and second versions of mulrom.asm were compared, i.e., UMUL subroutine versus UMUL macro, which has the most instructions? Use sections of the listing files to support your answer.*

   The *mulrom_macro.asm* program has two fewer instructions (15 compared to 17) than the *mulrom.asm* program. This is because *mulrom.asm* includes a “CALL” and a “RET” instruction that *mulrom_macro.asm* does not require, for it uses a macro definition instead of a subroutine call. Listings 7 and 8 show the code for both programs.

8. *At this time you have a running “computer” in the form of the program sw16sim.c. Compare the cost of this solution with that of a “hardware” implementation. What positive contributions does the “hardware” implementation make?*

   The *sw16sim.c* implementation of the *Sweet16* microprocessor has its advantages and disadvantages over the actual hardware implementation. One major advantage is the flexibility of this software implementation. Instructions may be added, altered, or deleted with very little time and effort. Altering the hardware implementation would be significantly more costly because it is not as simple as a software upgrade. In fact, It might even involve making entirely new Printed Circuit (PC) boards. However, a single hardware implementation would cost considerably less than the computer required to execute the *sw16sim.c* program and the UPASM assembler.

Lab No. 1
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 build and test a VHDL structural model of a Register Arithmetic-Logic Unit (RALU). This unit will be used as the heart of the internal architecture of the Complex Instruction Set Computer (CISC). Furthermore, many of the components of the RALU will be utilized in the Reduced Instruction Set Computer (RISC).

As opposed to a typical Arithmetic-Logic Unit, the RALU has the added feature of a register array. This allows the data calculated by the ALU to be stored into one of 20 registers. In addition, the RALU was specifically designed to be able to implement the *Sweet16* Instruction Set Architecture (ISA). Therefore the RALU has the ability to calculate branch target addresses, data shifts, arithmetic/logic operations, etc.

The main design platform for the RALU and its internal components was the Altera Max+Plus II VHDL editor. This allowed each component to be designed individually and then combined into a VHDL package. This type of compartmentalized design replicated the provided schematic drawings and simplified the design of the entire system.

Component Design and Validation

The individual components of the RALU were designed separately in VHDL. These components include multiplexers, registers, shifters, a register array, and an Arithmetic-Logic Unit.

A. 20x16 Register Array

One of the important features of the RALU is that it can store the results of its various calculations. Data is stored in one of the 16 General-Purpose Registers (GPRs) or in one of the four additional registers (i.e. the status flag register). This feature is realized in the 20x16 Register Array appropriately named *reg_array_20x16.vhd* (see Appendix B, Component 1 for the complete VHDL code for this component).

The main structure of this component is simple. There are three address inputs, one to address the first “data register,” one to address the second “data register,” and one to address the “write register.” The two “data registers” supply the ALU with the data it needs to make meaningful calculations. The “write register” is the location to which the result of the ALU calculation will be stored. It is worth mentioning, however, that not all instructions require two “data registers,” and not all ALU calculations are meant to be stored in a “write register.”

Another feature of this unit is its ability to deal with both word-size and byte-size data. The input *WnB* stands for “word, not byte” and indicates how the data should be treated—as a 16-bit word or as an 8-bit byte. Figure 1 on the next page shows the Max+Plus II graphical representation of *reg_array_20x16.vhd*.

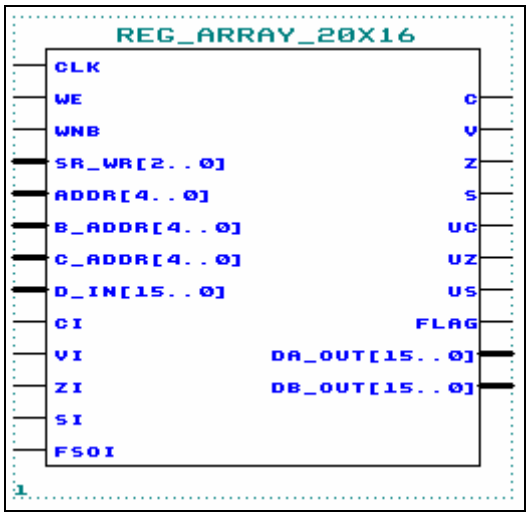

Figure 1: Register Array

B. 16-Bit Arithmetic-Logic Unit (ALU)

Arguably the most important part of the RALU is the Arithmetic-Logic Unit (ALU). This component performs all of the arithmetic and logical calculations that are necessary for executing the *Sweet16* instructions. In addition to calculating various operations, this unit also generates the carry and overflow flags for the RALU (see Appendix B, Component 2 for the complete VHDL code for this component). The functions that this ALU performs are varied and diverse. Figure 2 lists the functions which were implemented in the *ALU_16a.vhd*.

Function Select (*FSel*)	**Iterate (*i1*)**	**Iterate (*i0*)**	**Function**
0	X	X	Add inputs A and B with Carry In; F = A plus B plus Cin
1	X	X	Add input B to Carry In; F = B plus Cin
2	X	X	Subtract input B from A with Borrow; F = A plus (not B) plus Cin
3	X	X	Subtract input A from B with Borrow; F = (not A) plus B plus Cin
4	X	X	F = A and B
5	X	X	F = A or B
6	X	X	F = A xor B
7	X	X	F = not A
8	X	X	F = A minus 1 plus Cin (F = A plus 0xFFFF plus Cin)
9	X	X	F = B minus 1 plus Cin (F = B plus 0xFFFF plus Cin)
A	0	X	unsigned multiply iterate, F = B
A	1	X	unsigned multiply iterate, F = A plus B
B	0	X	signed multiply iterate, F = B
B	1	X	signed multiply iterate, F = A plus B
C	0	X	signed multiply terminate, F = B
C	1	X	signed multiply terminate, F = B minus A (F=B plus (not A) plus 1)
D	X	0	Nonrestoring divide, F = A plus B
D	X	1	Nonrestoring divide, F = B minus A (F = (not A) plus B plus 1)
E	X	X	F = 0 for later expansion
F	X	X	F = 0 for later expansion

Figure 2: ALU function map

As is shown in Figure 2, this component was left open for expansion. If, along the process of designing a CISC or RISC architecture, we discover that additional types of computations are necessary, then they may be added into this design relatively easily. The Max+Plus II graphical representation of this unit is shown below in Figure 3.

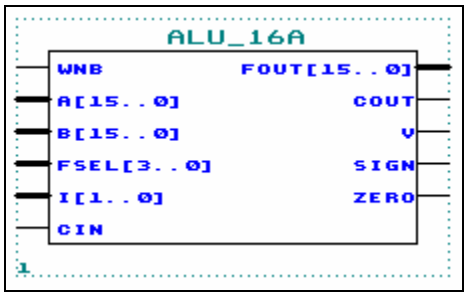

Figure 3: *Sweet16* ALU

As shown in Figure 3, this component requires two 16-bit inputs—the inputs on which it performs calculations. The resulting 16-bit output is some combination of the two 16-bit inputs and the 4-bit function select (along with a few other input signals).

C. Shifters

To handle the *shift* and *rotate* instructions (and eventually the *multiply* and *divide* instructions as well), shifting units were added to the output of the ALU. These components were designed in VHDL, and their Max+Plus II graphical representation are shown below in Figure 4 (see Appendix B, Components 3 and 4 for the VHDL code for the *ALU_shifter_16* and the *Ext_shifter_16*, respectively).

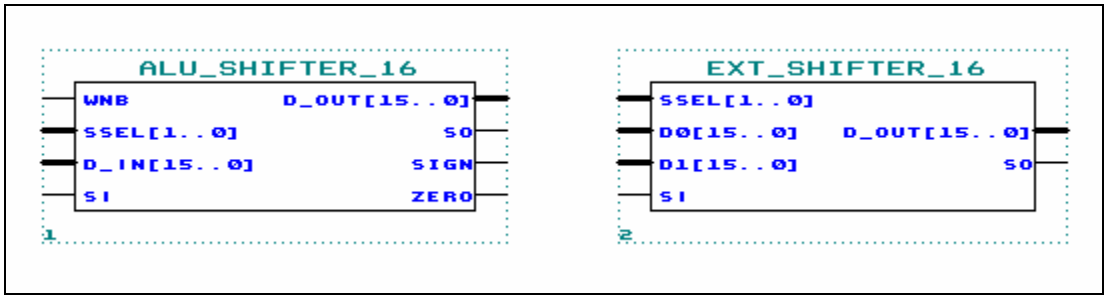

Figure 4: Two *Sweet16* shifters

It is worthwhile to note that these units are responsible for generating the *sign* and *zero* flags. Since each of these flags can be altered as a result of a shift/rotate, and since every output of the ALU must pass through the shifters (even if no shifting is taking place), it is advantageous to have the shifter units generate these flags rather than the ALU.

D. Multiplexers

Various multiplexers (otherwise known as selectors) were utilized in designing the RALU. The main difference between these multiplexers is the size of their inputs/outputs and the number of inputs they have. Figure 5 below shows the Max+Plus II graphical representation of the multiplexers used in the RALU implementation.

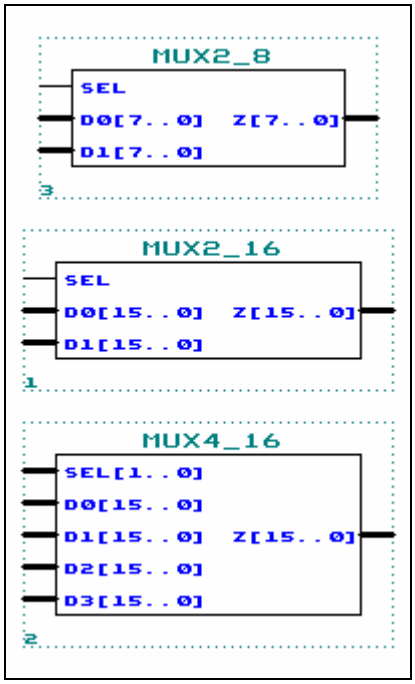

Figure 5: RALU multiplexers

The VHDL code for the *MUX2_8*, *MUX2_16*, and *MUX4_16* components can be found in Appendix B (Components 5, 6, and 7, respectively).

System Design and Validation

The functional units discussed in the previous section were combined into one VHDL design to form the *RALU_16* (see Appendix F, Specification Sheet 1 for a description of this component). The components of the *RALU_16* were assembled and interconnected according to the schematic drawing in Figure 6 on the next page.

It is clear, from Figure 6, that the inputs to this system are numerous. The majority of the inputs are control signals that come from the *Sweet16* controller—a unit that will be constructed in a later lab. The outputs of the system include the flags and the output data, among other signals. See Appendix B, Component 8 for the VHDL code for the *RALU_16*.

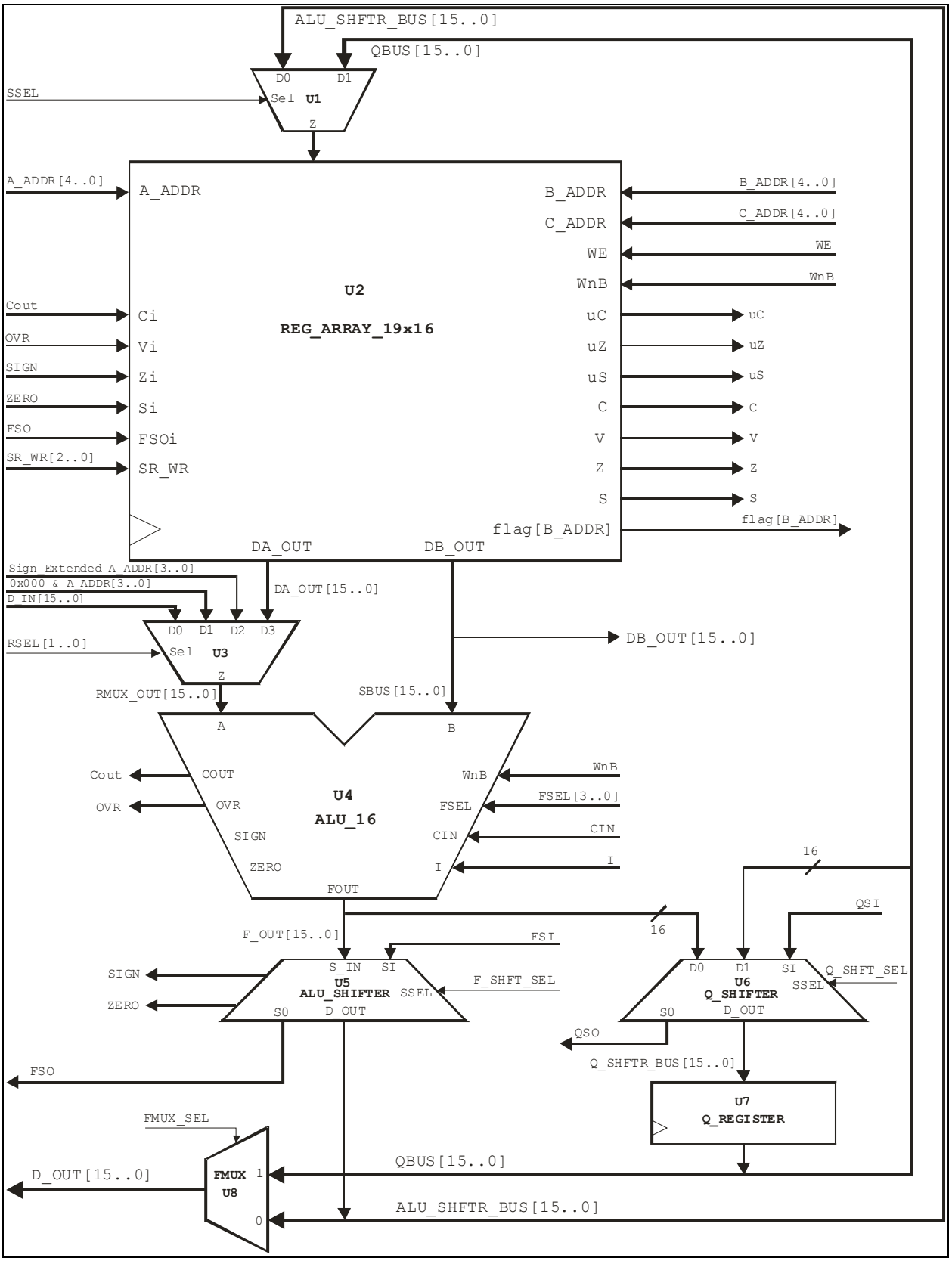

Figure 6: RALU schematic

One of the most important features of the *RALU_16* is the ALU input selector mux (component U3 in Figure 6). This multiplexer allows branch target addresses to be computed. It also allows immediate data (both short and long) to be used in calculations. Finally, it can also select register data to be used in calculations. This expands the versatility of the ALU and of the *RALU_16*.

Once assembled in VHDL, this design was functionally compiled and simulated to verify the accuracy of its design. All of the *ALU_16*s functions were tested along with the shifting/rotating capabilities of the *RALU_16*. Of particular interest was the response of the flag outputs to various arithmetic/logical operations. Upon close examination of the simulation results (see Appendix E, Waveform Simulation 1), it was determined that the *RALU_16* behaved as designed.

Conclusion

A. Summary

The system designed in this lab is the computational heart of the *Sweet16* microprocessor. All arithmetic and logical calculations embedded within the *Sweet16* ISA are performed by this unit. Its design lends itself to the complex instruction set around which the *Sweet16* was developed. Combining a register array with an Arithmetic-Logic Unit (ALU) enables the *RALU_16* to fulfill the data manipulation requirements of the ISA which it implements.

Designing this system in discrete functional units instead of a single, comprehensive unit allowed for a more versatile and understandable system. Furthermore, this system may be altered in order to add/remove functionality where necessary.

B. Questions

1. *Argue from the architecture presented in Fig. 1 that the following operation can be performed in a single clock cycle:*

   Reg_Array[R5] = (Reg_Array[R5] plus Reg_Array[R1] plus Cin) / 2 (*Instruction 1*)

   Notice the divide by 2. You should suggest connections that will help preserve the sign of a 2's complement number for the division. Also suggest which component(s) do what, along with the values needed for any "select" lines. The above operation should be tried using unsigned numbers by preparing and running a sequence of test vectors described in a waveform file. Do it. You will need to "load" R5 and R1 from the D_IN port before you do the actual test.

This operation can be accomplished in one clock cycle with the following control signals:

Signal	Binary Value	Explanation
WnB	1	Treat data as a 16-bit word, not an 8-bit byte
A_ADDR[4..0]	0 0101	Read data from register five
B_ADDR[4..0]	0 0001	Read data from register one
C_ADDR[4..0]	0 0101	Write result to register five
RSEL[1..0]	11	Select register data for ALU input A
FSEL[3..0]	0000	Perform: F = A + B + Cin
F_SHFT_SEL[1..0]	010	Shift the sum right in order to divide by two
FSI	F_OUT[15]	Preserve the sign of the number when dividing (shift right)
SSEL	0	Select the data from the shift bus to be written to the register array
WE	1	Allow data to be written to the register addressed by C_ADDR

Figure 7: Control signals required to accomplish *Instruction 1*.

This operation was simulated using the control signals in Figure 7. The results of the simulation, shown in Figure 8 below, prove that this operation can be performed in one clock cycle.

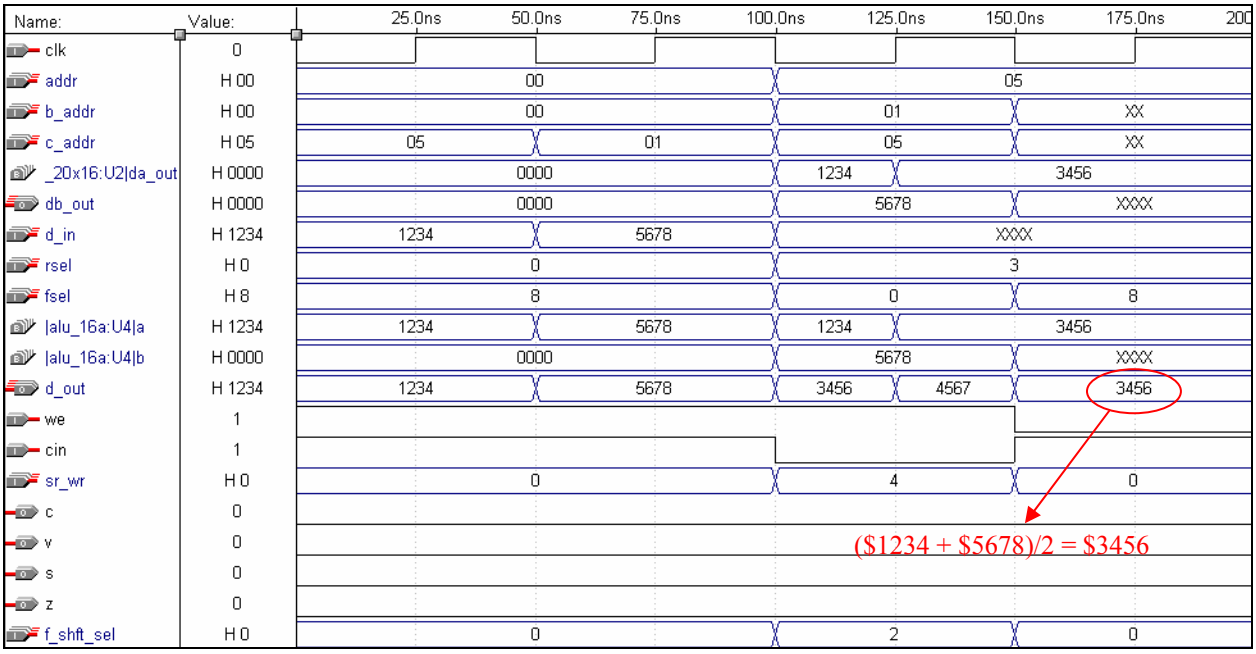

Figure 8: Simulation of *Instruction 1*

2. *So, now you've got a good part of a microprocessor in your hands. What's its clock speed? If you don't know, what sort of information would you need to determine the clock speed? Propose a test to determine the clock speed at which your microprocessor should run.*

In order to determine the clock speed for this microprocessor, a timing analysis must be performed. The clock speed must not be faster than the slowest delay through the RALU (likely the slowest component of the microprocessor). To conduct this analysis, the *RALU_16* component must be compiled using the Timing SNF Extractor so as to account for internal propagation delays. When compiling this way, the compiler will include the propagation delays inherent in a specific target device of your choosing. Once compiled, a Timing Analysis can be performed that will list the worst-case delay through each path on the circuit. The clock period must be greater than the longest delay. Thus the clock frequency must be the inverse of the longest time delay through the circuit. This will prevent the possibility of data being clocked at a rate faster than the time it takes for the data to stabilize.

Lab No. 3
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 design and assemble the complete *Sweet16* Central Processing Unit (CPU). This CISC architecture was built in VHDL in three distinct portions, the internal architecture, the *Sweet16* controller, and the auxiliary components. The internal architecture contains the *RALU_16* designed in an earlier lab together with some "glue logic" that enables the interface with the controller. The *Sweet16* controller consists of the *upcont_56* Microprogrammed Controller constructed earlier along with an Instruction Register and a MapROM. Finally, the auxiliary components comprises are comprised of the I/O interface buffer as well as the Memory Address Register (MAR).

These three components combined form a fully functioning CPU. Once this is complete, the only modifications that need to be made are concerning the microprogram that resides in the Microprogram Memory. Once the proper microprogram is installed, one that fully describes and implements the *Sweet16* instruction set, the final addition will be an external architecture that contains memory and I/O ports.

Component Design and Validation

As mentioned in the introduction, the *Sweet16* CPU consists of the internal architecture, the *Sweet16* controller, and the auxiliary components. Each of these components was assembled using Max+Plus II's Graphic Editor.

A. Internal Architecture

The *Sweet16* internal architecture consists of the 16-bit Register Arithmetic-Logic Unit (*RALU_16*) designed in Lab 1 (see Appendix F, Specification Sheet 1 for a description of this component). To properly interface this unit with the 42 control signals generated by the *Sweet16* controller, some "glue logic" was added to the perimeter. The internal architecture was designed based on the schematic drawing shown in Figure 1 (see Appendix F, Specification Sheet 3 for a description of this component).

The most interesting feature of the internal architecture is the register selection scheme. It is important to notice that the destination register is the same as the first source register. This corresponds with the convention that only two registers are specified in any given instruction. These two registers are the source registers, one of which also serves as the destination register. By virtue of multiplexers U2 and U3, the source and destination registers may be determined by the register fields of the instruction or by the *Sweet16* controller itself. In addition, this feature makes accommodations for 32-bit operations by isolating the least significant bit (LSB) of the register address from the other bits. This will allow the controller to toggle the LSB in order to perform computation on 32-bit numbers stored in two adjacent registers.

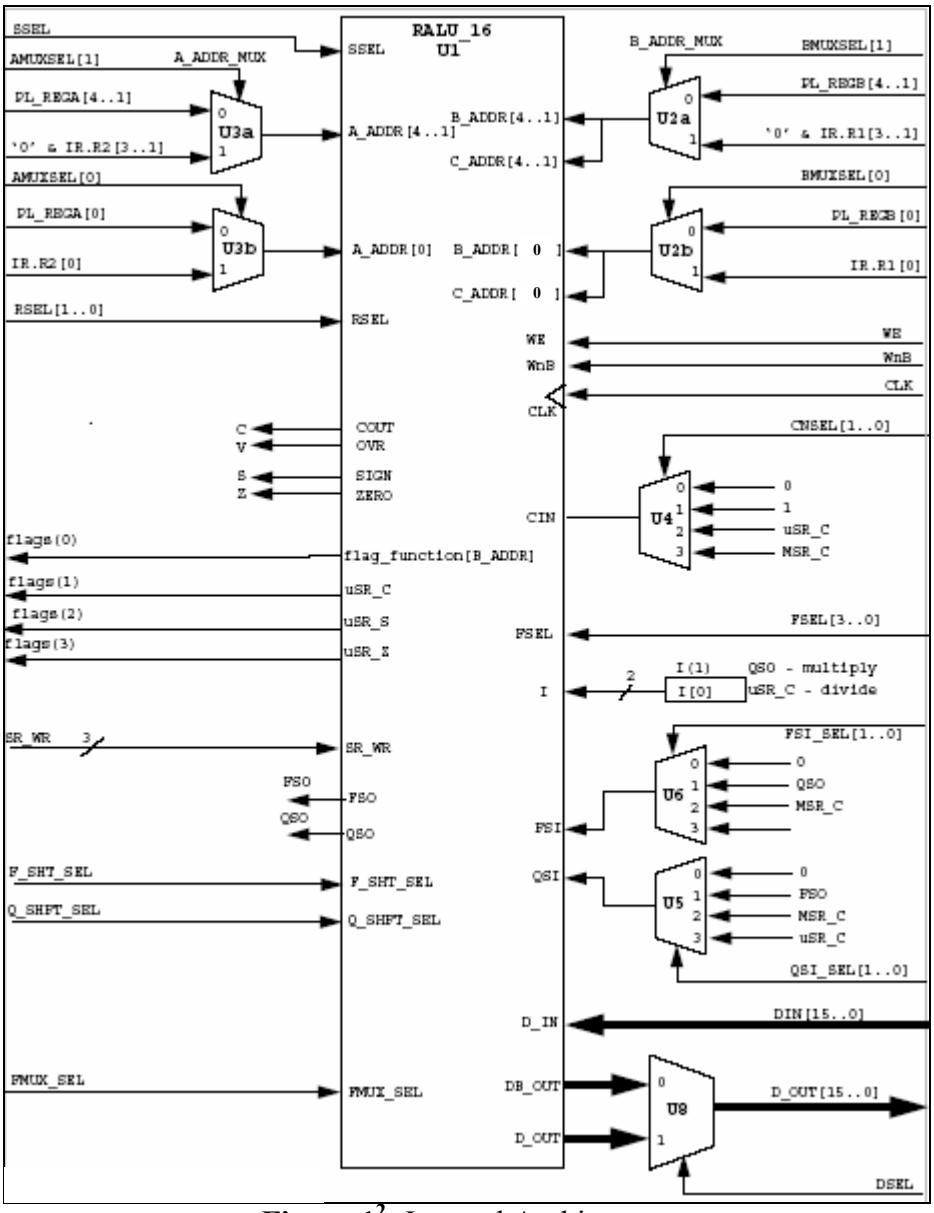

Figure 1[2]: Internal Architecture

The internal architecture design also makes it possible to select the carry input to the *RALU_16*. This is useful for incrementing register values, for passing register contents to the data bus, for subtraction, and for various other arithmetic processes. The shifting scheme also employs the use of multiplexers. This is to allow for various operations including rotation, multiplication, division, and of course shifting.

The schematic drawing in Figure 1 was implemented in Max+Plus II's Graphic Editor. The resulting Graphic Design File, *sw16intarch.gdf*, is shown in Figures 2a and 2b on the next two pages.

[2] http://www.hcs.ufl.edu/~radlinsk/eel4713

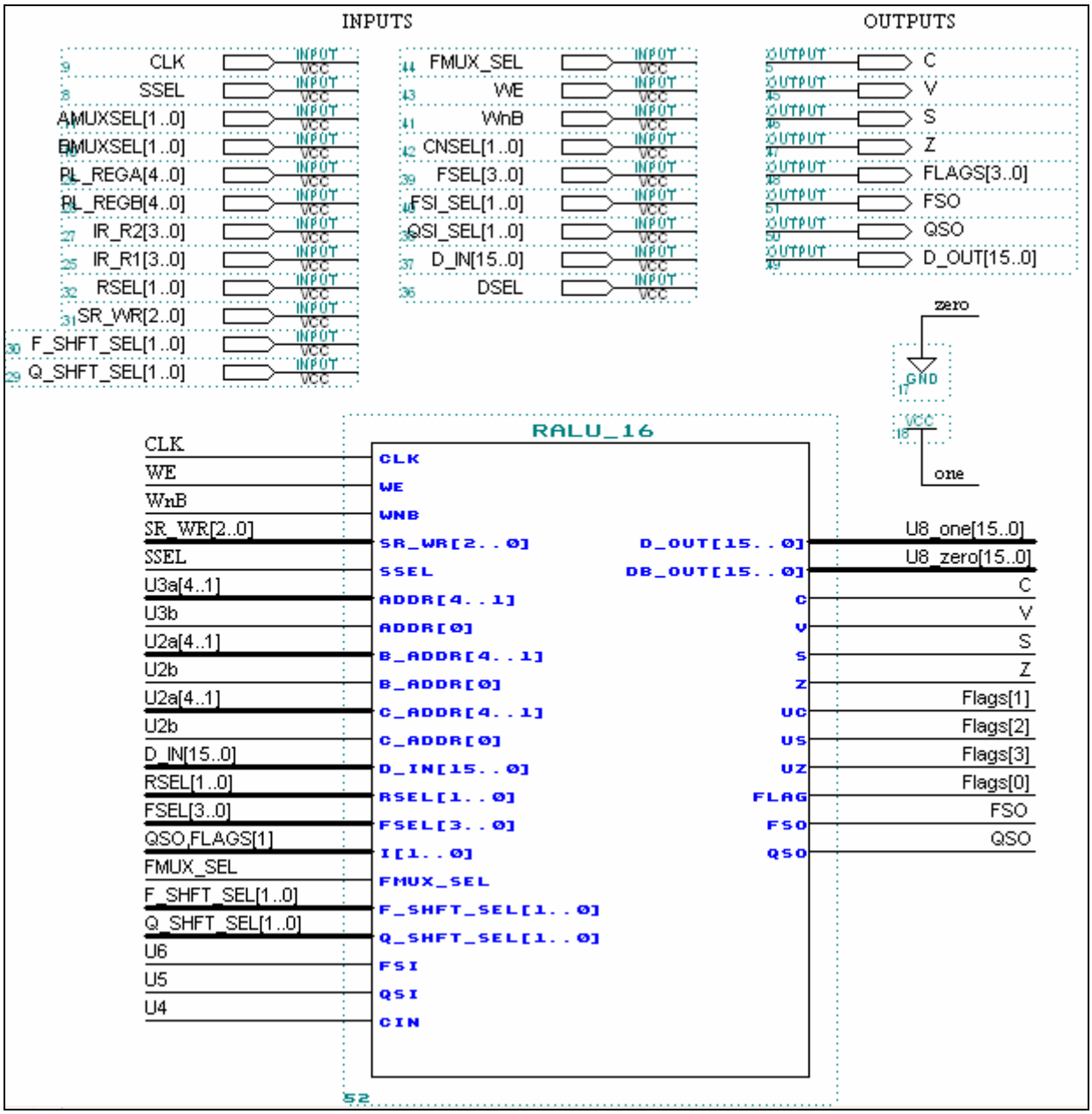

Figure 2a: Graphic design of the Internal Architecture

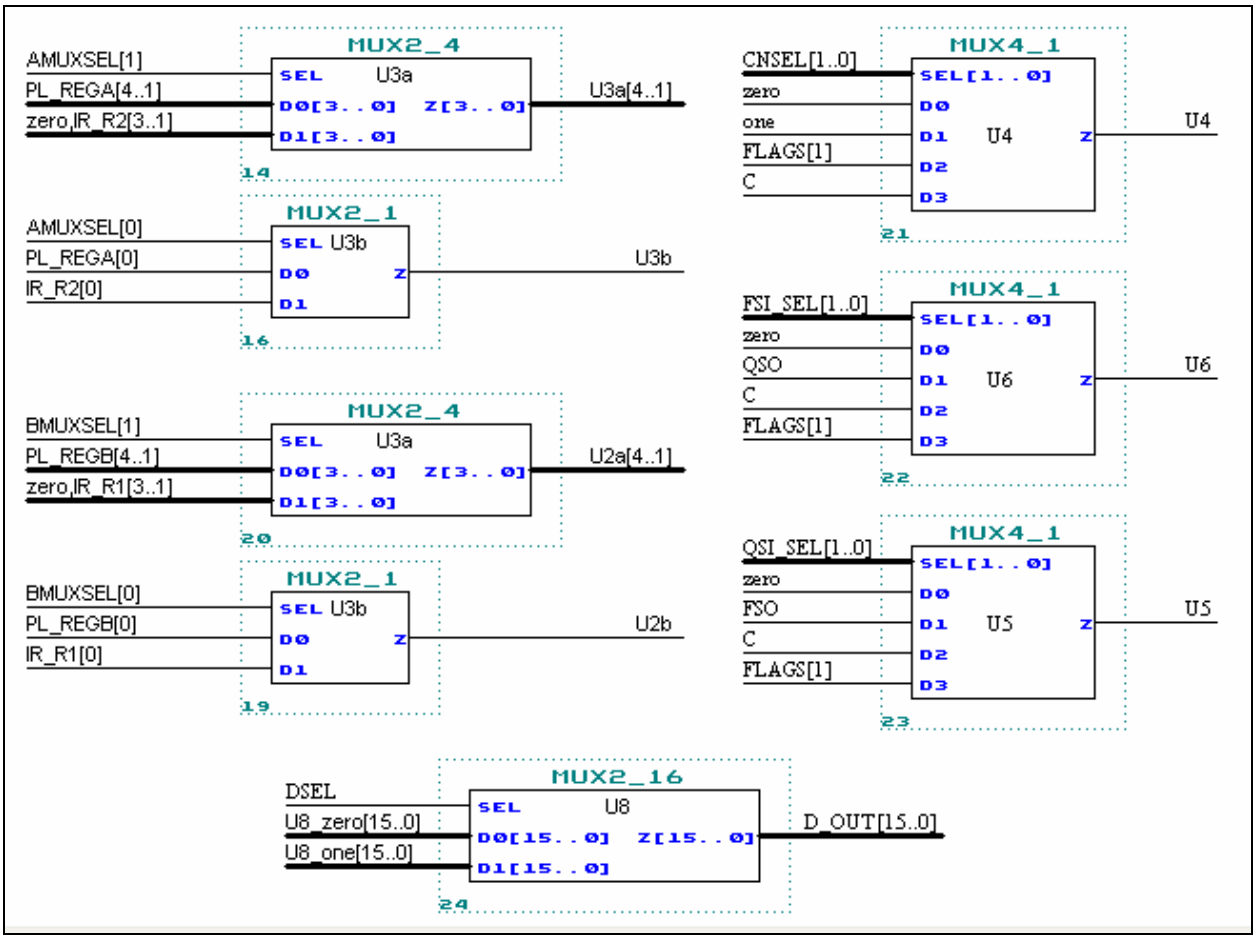

Figure 2b: Graphic design of the Internal Architecture

Once functionally compiled, this design was simulated for appropriate test vectors to verify the accuracy of its design. See Appendix E, Waveform Simulation 5 for the complete and annotated results of this simulation. Upon close examination of the simulation results, it was determined that this component performed as desired.

B. Sweet16 Controller

The *Sweet16* controller consists of the *upcont* designed in Lab 2, the Microprogram Memory, the Pipeline Register, the MapROM, and the Instruction Register (IR). These components were combined according to the schematic drawing in Figure 3 on the next page (see Appendix F, Specification Sheet 4 for a description of the *Sweet16* controller). The main component in this controller is the *upcont* unit (see Appendix F, Specification Sheet 2 for a complete description of this component). This unit controls the sequence of microinstructions to be executed throughout the *Sweet16* CPU. It does so my determining the appropriate address sequence for the Microprogram Memory, which in turn generates the control signals for the *Sweet16* CPU.

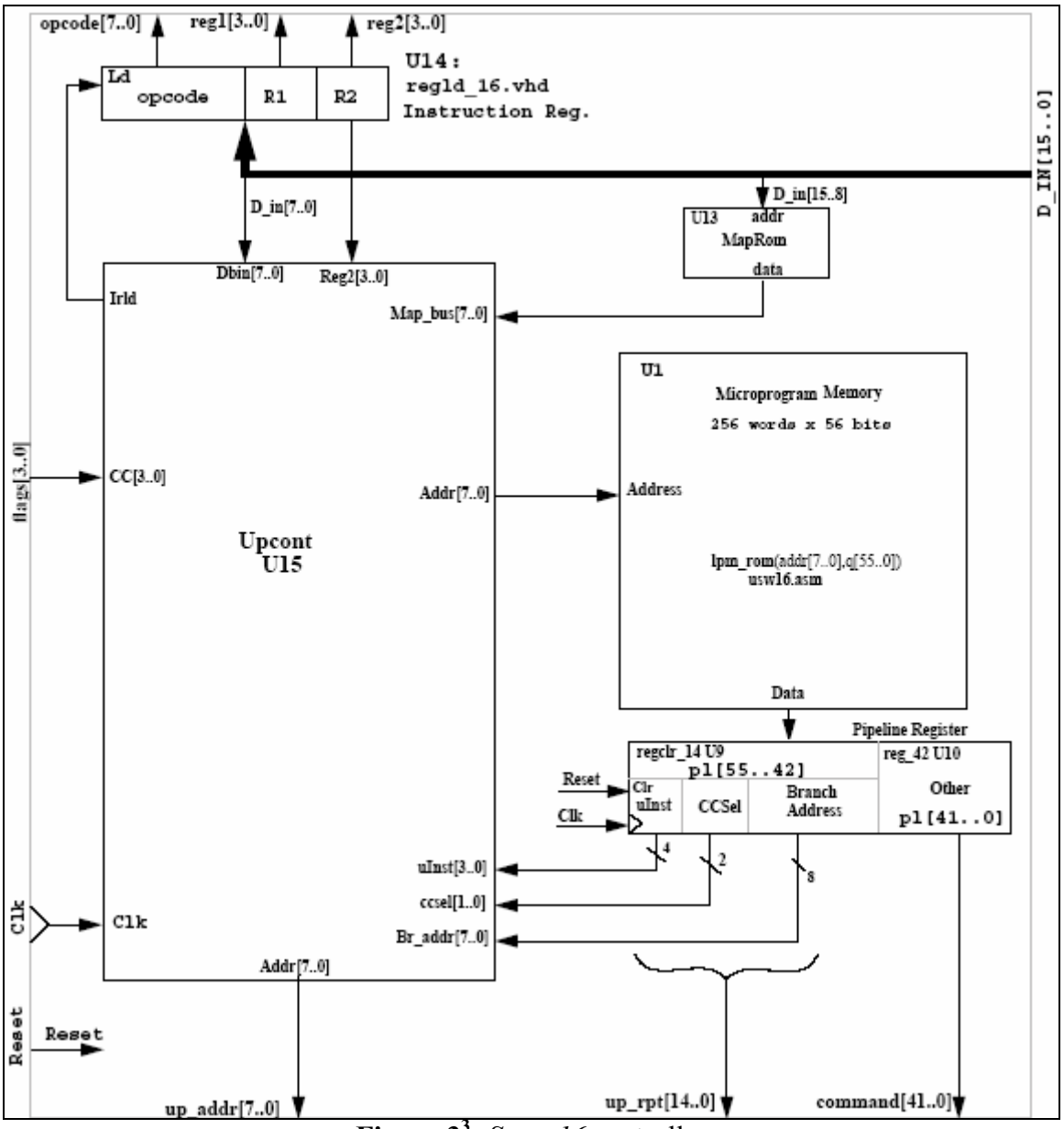

Figure 3[3]: *Sweet16* controller

The IR contains the opcode and operands for the current *Sweet16* instruction. This information is utilized by the MapROM to determine the starting address for each sequence of microinstructions. The *upcont* is then in charge of "stepping through" that sequence of microinstructions in the proper order so as to execute the intended *Sweet16* instruction. An *ir_ld* signal was added to the *upcont* unit to control when the IR would load a new instruction from the data bus. This signal was issued every time a sequence of microprograms neared completion. Thus, a new *Sweet16* instruction was loaded into the IR after the previous instruction was fully executed.

[3] http://www.hcs.ufl.edu/~radlinsk/eel4713

The schematic drawing in Figure 3 was implemented in Max+Plus II's Graphic Editor. The resulting Graphic Design File, *sw16cont.gdf*, is shown in Figures 4a and 4b below.

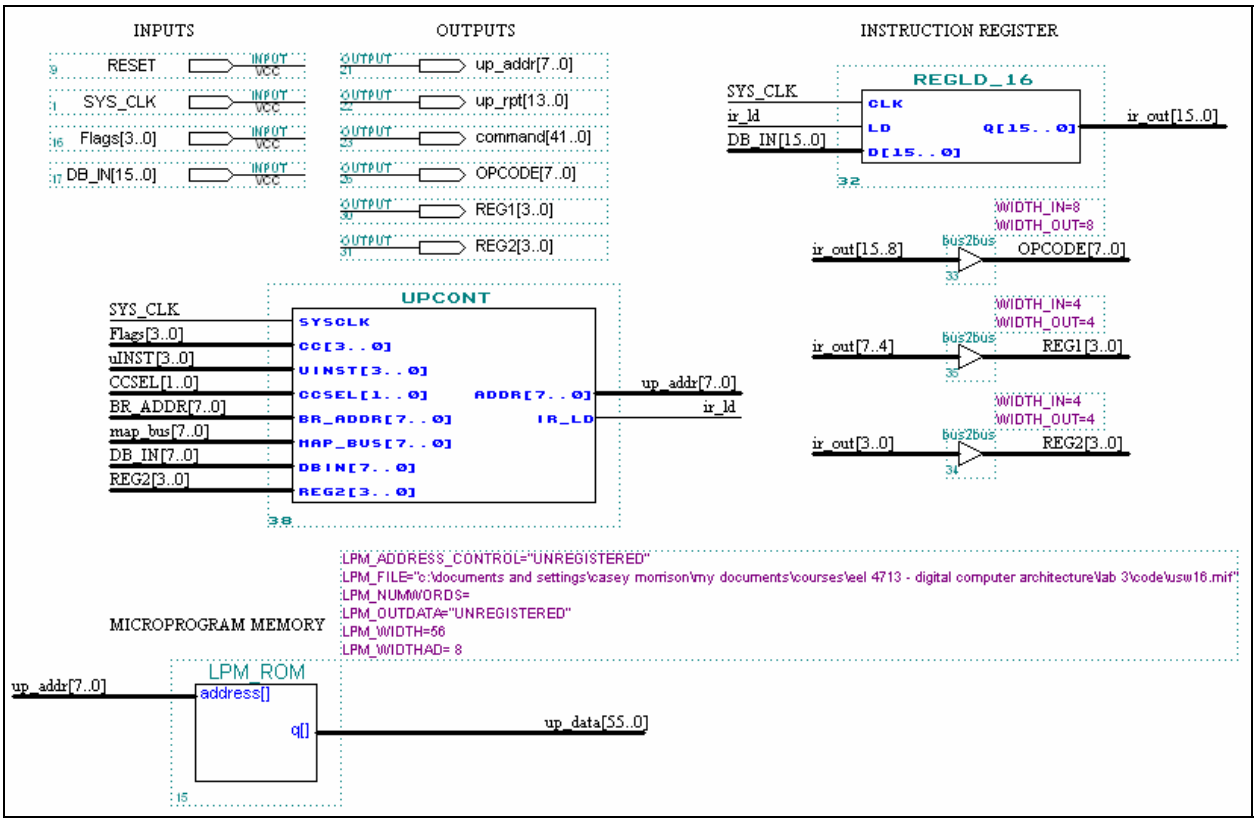

Figure 4a: Graphic design of the *Sweet16* controller

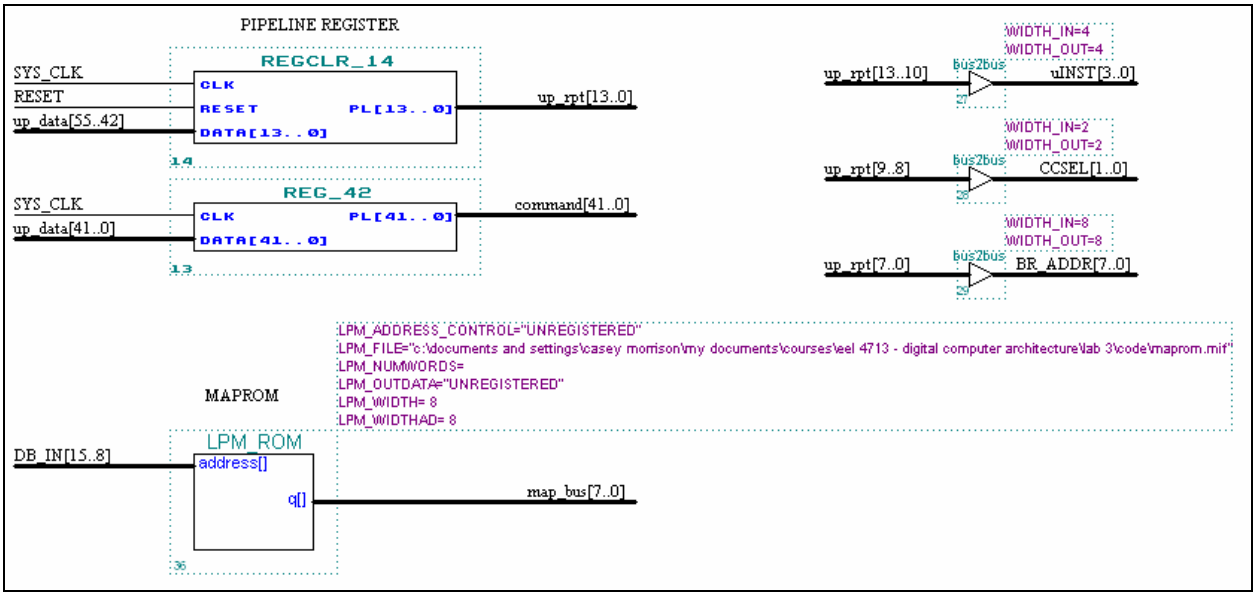

Figure 4b: Graphic design of the *Sweet16* controller

Once functionally compiled, this design was simulated for appropriate test vectors to verify the accuracy of its design. See Appendix E, Waveform Simulation 6 for the complete and annotated results of this simulation. Upon close examination of the simulation results, it was determined that this component performed as desired.

C. Auxiliary components

The auxiliary components of the *Sweet16* CPU consist of an Input/Output buffer and a Memory Address Register (MAR). The former allows the external data bus to be bidirectional, while avoiding unwanted data collisions at the same time. This buffer is controlled by the controller and facilitates the flow of data in and out of the *Sweet16* CPU.

The MAR is a register that stores a pointer to main memory. The address in the MAR can point to either an instruction or data. The contents of the memory location pointed to by the MAR would be transmitted to the *Sweet16* CPU via the aforementioned bidirectional data bus.

Ideally other devices would interface with the I/O buffer such as an input or output port. Such additions may, however, come with the cost of added control signals.

The auxiliary components were implemented in Max+Plus II's Graphic Editor. Figure 5 below shows the graphical representation of this design.

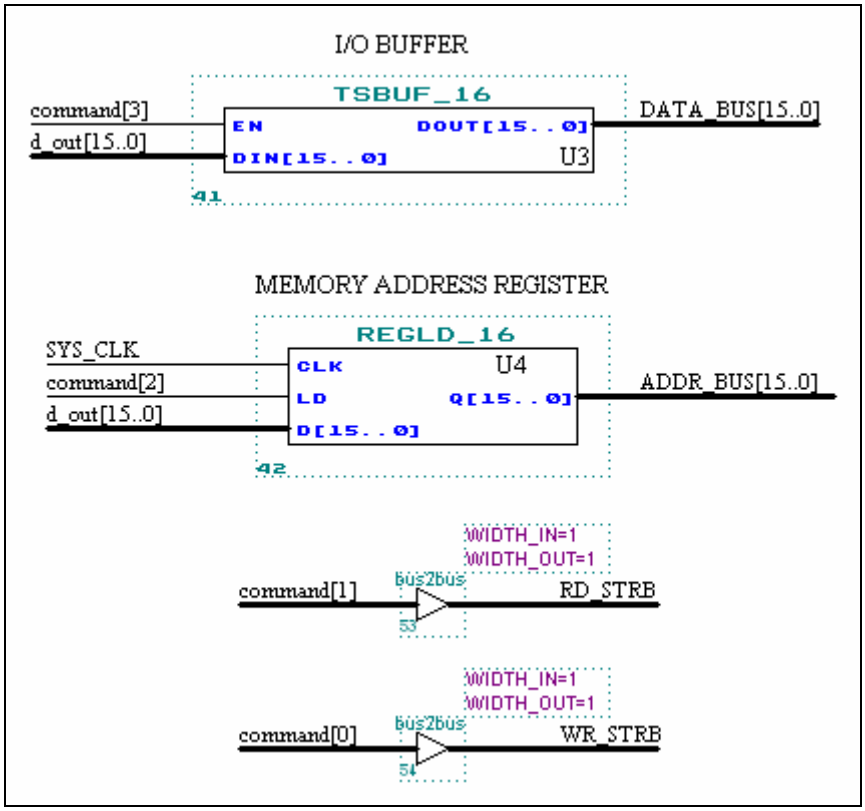

Figure 5: Graphical design of auxiliary components

System Design and Validation

The individual components of the *Sweet16* CPU were combined into one graphical design. Figure 6 shows the resulting design in Max+Plus II's Graphic Design editor (see Appendix F, Specification Sheet 5 for a complete description of this design).

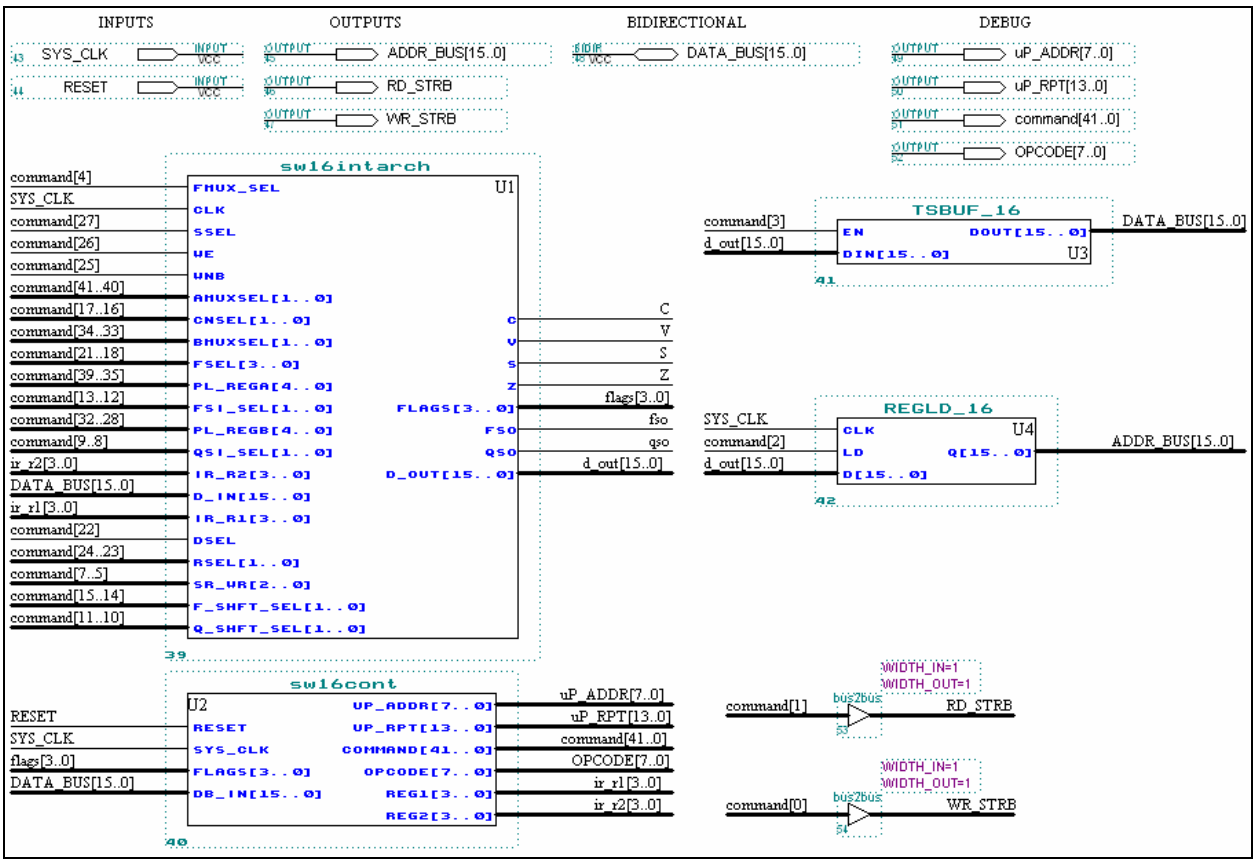

Figure 6: *Sweet16* CPU

A. System Test

The operation of this system was tested with a rudimentary microprogram called *usw16.asm* (see Appendix C, Program 6 for the abbreviated assembly code). This assembly program acts as the microprogram memory initialization file. It provides an uncompleted set of subroutines that are used to execute *Sweet16* instructions. The *usw16.asm* file was assembled with *UPASM* and converted into a memory initialization file (.mif) with the *s2mif56* tool. The resulting *usw16.mif* file was loaded into the Microprogram Memory.

In addition to initializing the Microprogram Memory, the MapROM also needed to be initialized. The *maprom.mif* file that was provided was used to initialize the MapROM. This file allows the MapROM to interpret the opcode of the current *Sweet16* instruction and direct the controller to the appropriate location in the Microprogram Memory.

Using the *mcsw16* tool provided for this lab, the *usw16.s* file was converted into a human-readable form. A portion of the resulting conversion is shown in Listing 1 below.

```
                                              F   Q
                                              _   _       D
                                              s   s     F b
                    B A  P B  P               h F h Q   m _
                    r m  l m  l               f s f s   u d M R W
               u C  _ u  _ u  _             C t i t i S x v a d r
               I C  A x  r x  r S     R D F n _ _ _ _ r _ r r _ _
               n s  d s  e s  e s   W s s s s s s s s _ s _ _ s s
               s e  d e  g e  g e W n e e e e e e e e w e e l t t
               t l  r l  a l  b l e B l l l l l l l l r l n d r r

State Address +-+-+--+-+--+-+--+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
000    000    |1|0|00|0|11|0|11|0|1|1|3|0|6|0|0|0|0|0|0|0|0|0|0|0|
001    001    |8|0|02|0|11|0|11|0|1|1|3|0|1|1|0|0|0|0|0|0|0|1|0|0|
002    002    |9|0|02|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|1|0|
003    003    |F|0|00|0|11|0|11|0|1|1|3|0|1|1|0|0|0|0|0|0|0|0|1|0|
004    004    |1|0|00|0|11|0|11|0|1|1|3|0|1|1|0|0|0|0|0|0|0|1|0|0|
005    005    |1|0|00|0|11|0|11|0|1|1|3|0|1|1|0|0|0|0|0|0|0|0|1|0|
006    006    |6|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|1|0|
007    007    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
008    008    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
009    009    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
00A    00A    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
00B    00B    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
00C    00C    |6|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|1|0|
020    020    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
021    021    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
022    022    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
023    023    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
024    024    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
025    025    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
026    026    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
027    027    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
028    028    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
029    029    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
02A    02A    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
02B    02B    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
02C    02C    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
02D    02D    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
02E    02E    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
02F    02F    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
030    030    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
031    031    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
032    032    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
033    033    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
034    034    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
035    035    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
036    036    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
037    037    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
038    038    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
039    039    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
03A    03A    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
03B    03B    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
03C    03C    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
03D    03D    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
03E    03E    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
03F    03F    |2|0|01|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
040    040    |1|0|00|0|00|0|00|0|0|1|3|0|8|0|0|0|0|0|0|0|0|0|0|0|
```

Listing 1: *usw16.mc* file output

The control signals in Listing 1 make up the 42-bit wide *command* bus. The structure of this bus is shown in Figure 7 below.

41 40	39 35	34 33	32 28	27	26	25	24 23	22	21 18	17 16
AMUXSEL	PL_REGA	BMUXSEL	PL_REGB	SSEL	WE	WnB	RSEL	DSEL	FSEL	CNSEL

15 14	13 12	11 10	9 8	7 5
F_SHFT_SEL	FSI_SEL	Q_SHFT_SEL	QSI_SEL	SR_WR

4	3	2	1	0
FMUXSEL	DB_DVR_EN	MAR_LD	RD_STR	WR_STR

Figure 7: Command bus structure

With the two memory devices initialized with the appropriate Memory Initialization Files, the complete system was tested in a Waveform Simulation. The operation of the *Sweet16* CPU was monitored for the input of a specific instruction: LDR R3, R4. This arbitrary instruction allowed the resulting command signals to be verified for correctness. The result of this simulation is shown in Appendix E, Waveform Simulation 7. It clearly depicts the instruction fetch sequence implemented in the Microprogram Memory. Many steps are involved in this process. The contents of the Program Counter must be placed in the MAR. During the subsequent clock cycles, the contents of memory as indexed by the MAR are placed on the data bus. During this waiting period, the program counter is twice incremented to point to the next instruction- or data-word. Once the memory releases its data, the Instruction register is loaded with the current instruction, and instruction execution follows. This process was demonstrated in the waveform simulation of *sw16cpu.gdf*.

Conclusion

A. Summary

The system designed in this lab represents the complete *Sweet16* CPU. It combined the internal architecture, controller, and auxiliary components to form a fully functioning microprocessor. The only modifications that need to be made are concerning the Microprogram Memory. Once a more complete Memory Initialization File is developed, this design will be able to execute the entire *Sweet16* instruction set.

Further additions to this design will include an external architecture. This will include main memory wherein a program may be loaded for the *Sweet16* to execute. In addition to a main memory, it may be desirable to add an input or output port. All of these devices in the external architecture will have to be mapped in the 16-bit, 64 kiloByte memory space.

B. Questions

1. *In which clock cycles (of the first 6) was the ISP's PC incremented?*

   The Program Counter (PC) was incremented in the second and sixth clock cycles.

2. *Why was the ISP's PC incremented twice? Could we have added two to it in a single cycle? Be sure to account for extra hardware needed. Did the technique used cost any time?*

   The ISP's PC was incremented twice because this CPU deals with 16-bit words, whereas the memory is structured in 8-bit bytes. Thus the PC must increment/decrement in intervals of two to point to the appropriate word-aligned address.

   It would require additional hardware to increment the PC by two in one clock cycle. For example, this could be accomplished by expanding the ALU input mux to contain a hard-coded input of two (or one if the carry-input is going to be used in conjunction). This could also be accomplished by adding a new register to the register array—one that contains a hard-coded two. The ladder of the two would cost less in terms additional hardware.

   The technique currently in place does not cost additional time, for this time is spend waiting for memory to place its data onto the data bus. Thus the time it takes to increment the PC is essentially "free time."

3. *What are the units of address in the ISP? Does this change because we have a 16-bit data bus for Sweet16?*

   The units of address in the ISP are 16-bit words. Because we have a 16-bit data bus, we must address 16-bit words in memory.

4. *Why were there so many cycles used to access external memory during the instruction fetch sequence?*

   The large number of cycles used to access external memory during the instruction fetch can be attributed to the slow nature of external memory. It takes several cycles for the memory to place its data onto the bus, so the processor must wait until it is sure the data is ready to be latched.

5. *Describe, with the help of the proper logic diagrams, the path that the ISP's opcode follows to become the address of the microinstruction that implements the opcode. Use the LDR R3,R4 example suggested above.*

   What is the address of the first microinstruction in the "LDR" execution sequence in the microprogram? This number can actually be seen in the CPU simulation on the up_addr bus (one of the test data buses used to observe the CPU).

Refer to Figure 3 for the *Sweet16* controller schematic. The instruction from main memory is placed onto the data bus, and the opcode portion is received by the MapROM. The MapROM, in turn, generates a microprogram address corresponding to the microcode needed to execute the given opcode. The Next Address Logic unit then selects the MapROM data as the next address for the Microprogram Memory.

For example, if the *Sweet16* instruction were LDR R3, R4, then the opcode for this instruction would be 0x2B. The contents of the MapROM at address 0x2B is 0x6F (according to *maprom.mif*). Thus the location of the microcode associated with the LDR instruction is at Microprogram Memory address 0x6F.

Lab No. 4
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 design and assemble the *Sweet16* External Architecture. This architecture consists of five components: a 1k x 16 ROM, a 1k x 16 RAM, an input port, an output port, and a Memory Decoder. The two memory devices are used to store the program and data for the *Sweet16* microprocessor. The two ports allow data to flow in and out of the *Sweet16* microprocessor. Finally, the Memory Decoder is a combinatorial logic circuit that generates the control signals for the memory and I/O devices

One important factor in designing this external architecture was the necessity to prevent data collisions on shared busses. With the use of strictly monitored control signals as well as tri-state buffers, unwanted data collisions were prevented. Another interesting aspect of this design is the accommodations that were made to allow the architecture to incorporate 8-bit operations in the future.

Component Design and Validation

As mentioned in the introduction, the *Sweet16* External Architecture consists of five components: a 1k x 16 ROM, a 1k x 16 RAM, an input port, an output port, and a Memory Decoder.

A. 1k x 16 ROM

The Read Only Memory (ROM) utilized in the *Sweet16* memory map is 1 kilo-byte large and 16 bits wide. This is accomplished with the use of two 1k x 8 ROMs. As will be discussed later, one ROM will contain the least-significant 8 bits of data, and the other will contain the most-significant 8 bits of data (see Appendix F, Specification Sheet 6 for a description of *rom_1kx8*).

The 1k x 8 ROM component was designed graphically in Max+Plus II (see Figure 1 below). The memory component was chosen from the *mega_lpm* library.

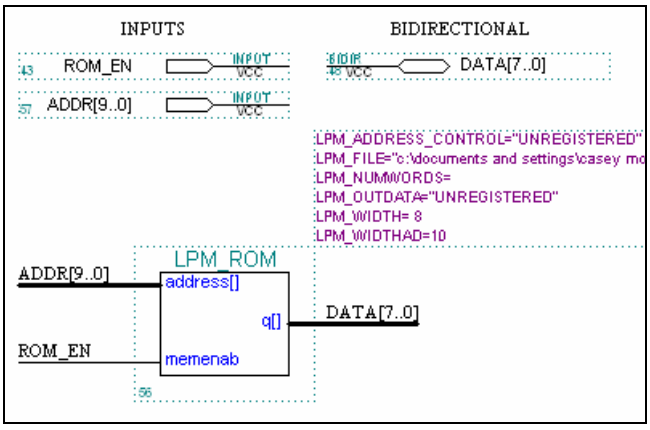

Figure 1: 1k x 8 ROM

This component was compiled and simulated for the test program *mulrom.asm*. The results of the simulation, shown below in Figure 2, illustrate the fact that each ROM will contribute one byte to the overall data word. This simulation is for the *ROM_High*, and thus it contains the most significant byte for each data word in *mulrom.asm*.

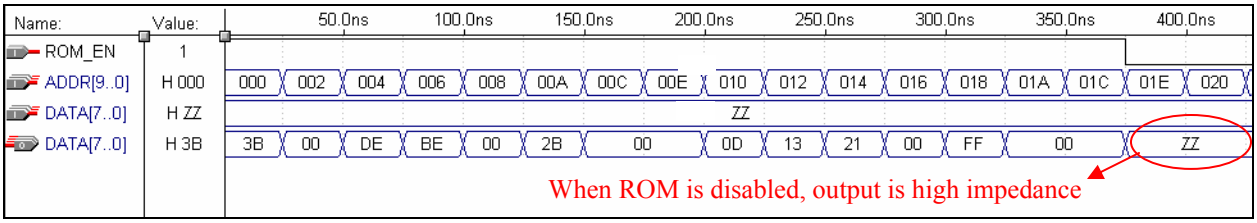

Figure 2: ROM simulation

Listing 1 below verifies the contents of this ROM. It represents the most significant byte for each data word in *mulrom.asm* (see Appendix C, Program 1 for the code for *mulrom.asm*).

```
% Output produced from S-record by s2mif0 %
% This file should be used with the ROM selected %
%  by an even address, i.e., on upper half of databus. %
Depth = 1024;
Width = 8;
Address_radix = hex;
Data_radix = hex;
% Program RAM Data %
Content
  Begin
0000 : 3B;
0001 : 01;
0002 : 00;
0003 : 3B;
0004 : DE;
0005 : 3B;
0006 : BE;
0007 : 14;
0008 : 00;
0009 : FF;
000A : 2B;
000B : 3B;
000C : 00;
000D : 3B;

0010 : 0D;
0011 : 0D;
0012 : 13;
0013 : 00;
0014 : 21;
0015 : 36;
0016 : 00;
0017 : 13;
0018 : FF;
0019 : 06;
[001A..03FF] : 00;
End;
```

Listing 1: ROM contents

B. 1k x 16 RAM

The Random Access Memory (RAM) utilized in the *Sweet16* memory map is 1 kilo-byte large and 16 bits wide. This is accomplished with the use of two 1k x 8 RAMs. As will be discussed later, one RAM will contain the least-significant 8 bits of data, and the other will contain the most-significant 8 bits of data (see Appendix F, Specification Sheet 7 for a description of *ram_1kx8*).

The 1k x 8 RAM component was designed graphically in Max+Plus II (see Figure 3 below). The memory component was chosen from the *mega_lpm* library.

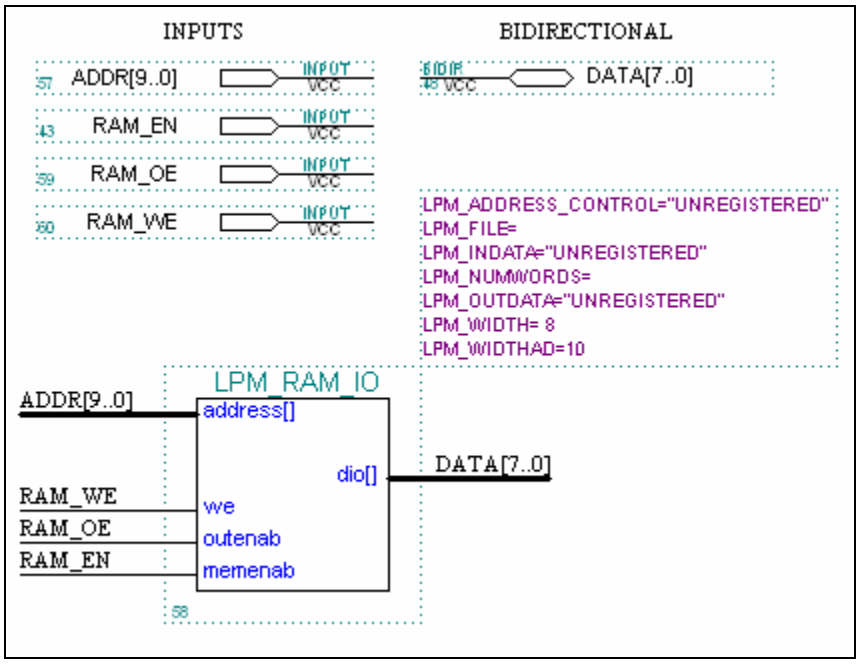

Figure 3: 1k x 8 RAM

This component was compiled and simulated for appropriate test vectors. The results of the simulation, shown below in Figure 4, prove the functionality of this component.

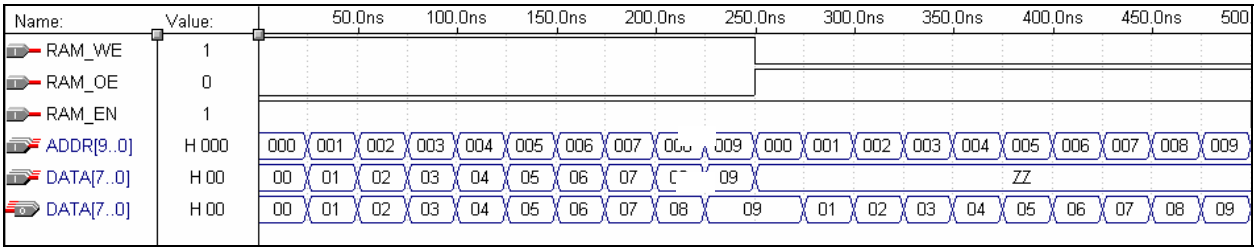

Figure 4: RAM simulation

C. Input Port

The input port for the *Sweet16* microprocessor is simply a 16-bit tri-state buffer. This prevents unwanted data collisions from occurring. By enabling the input port at specific times, this allows the data bus to be driven by the input port only when it is desired. The Max+Plus II graphical representation of this component is shown below in Figure 5.

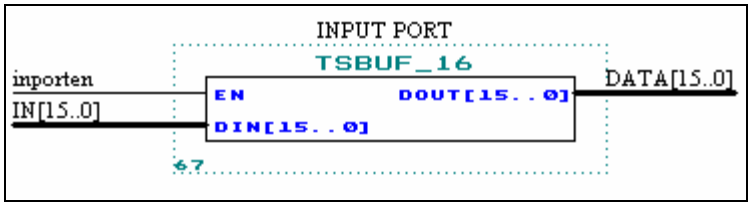

Figure 5: Input Port

D. Output Port

The output port for the *Sweet16* microprocessor consists of a 16-bit register. This allows data to be latched at specific times and for specific address locations. The Max+Plus II graphical representation of this component is shown below in Figure 6.

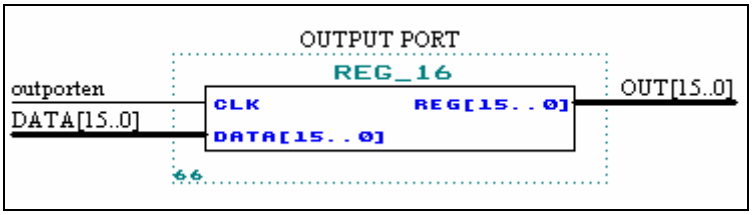

Figure 6: Output Port

E. Memory Decoder

To control the memory and I/O devices, a Memory Decoder was implemented in VHDL. The memory map on which this decoder was based is shown in Figure 7 on the next page. Each component in the external architecture is distinctly addressable. The enable signals for each of these components are partially based on the address that the CPU is writing/reading to/from. The read and write strobes also play an integral role in decoding the memory and I/O enable signals. See Appendix F, Specification Sheet 8 for a description of the Memory Decoder.

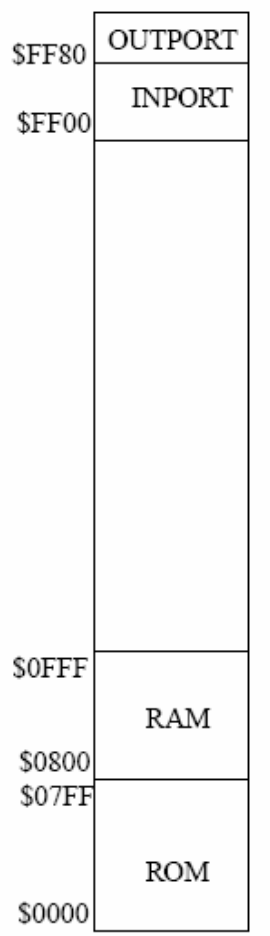

Figure 7[4]: *Sweet16* memory map

The VHDL for the Memory Decoder is shown in Listing 2 on the next page. Because of the way the VHDL was written, and because of the design of the memory map, no two components are ever simultaneously enabled. This ensures that data collisions will not occur and that the data flow will behave as desired.

[4] http://www.hcs.ufl.edu/~radlinsk/eel4713/course/view.php?id=2

```
Entity decoder is
Port(
    -- Inputs
    ADDR:   in std_logic_vector (15 downto 1);
    ADDR_0: in std_logic;
    RD_STR: in std_logic;
    WR_STR: in std_logic;

    -- Outputs
    ROM_HI_EN: out std_logic;
    ROM_LO_EN: out std_logic;
    RAM_HI_EN: out std_logic;
    RAM_LO_EN: out std_logic;
    RAM_WE:    out std_logic;
    RAM_OE:    out std_logic;
    OUTPORTEN: out std_logic;
    INPORTEN:  out std_logic
);
End decoder;

Architecture Behavior of decoder is
signal rom_en, ram_en: std_logic;
Begin
rom_en <= not ADDR(15) and not ADDR(14) and not ADDR(13) and not ADDR(12) and
          not ADDR(11) and RD_STR       and not WR_STR;

ram_en <= not ADDR(15) and not ADDR(14) and not ADDR(13) and not ADDR(12) and
              ADDR(11) and (RD_STR or WR_STR);

RAM_WE <= not ADDR(15) and not ADDR(14) and not ADDR(13) and not ADDR(12) and
              ADDR(11) and WR_STR       and not RD_STR;

RAM_OE <= not ADDR(15) and not ADDR(14) and not ADDR(13) and not ADDR(12) and
              ADDR(11) and RD_STR       and not WR_STR;

ROM_HI_EN <= rom_en; -- Temporary until 8-bit addressing is implemented
ROM_LO_EN <= rom_en; -- Temporary until 8-bit addressing is implemented

RAM_HI_EN <= ram_en; -- Temporary until 8-bit addressing is implemented
RAM_LO_EN <= ram_en; -- Temporary until 8-bit addressing is implemented

OUTPORTEN <= ADDR(15) and ADDR(14) and ADDR(13) and ADDR(12) and ADDR(11) and
             ADDR(10) and ADDR(9)  and ADDR(8)  and ADDR(7)  and WR_STR   and
             not RD_STR;

INPORTEN  <= ADDR(15) and ADDR(14) and ADDR(13) and ADDR(12) and ADDR(11) and
             ADDR(10) and ADDR(9)  and ADDR(8)  and not ADDR(7)  and RD_STR and
             not WR_STR;
End Behavior;
```

Listing 2: Memory Decoder VHDL code

The Max+Plus II graphical representation of this component is shown below in Figure 8.

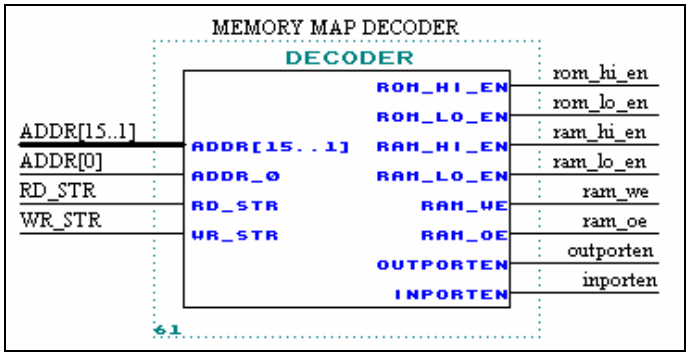

Figure 8: Memory Map Decoder

System Design and Validation

The individual components of the *Sweet16* External Architecture were combined according to the schematic shown in Figure 9.

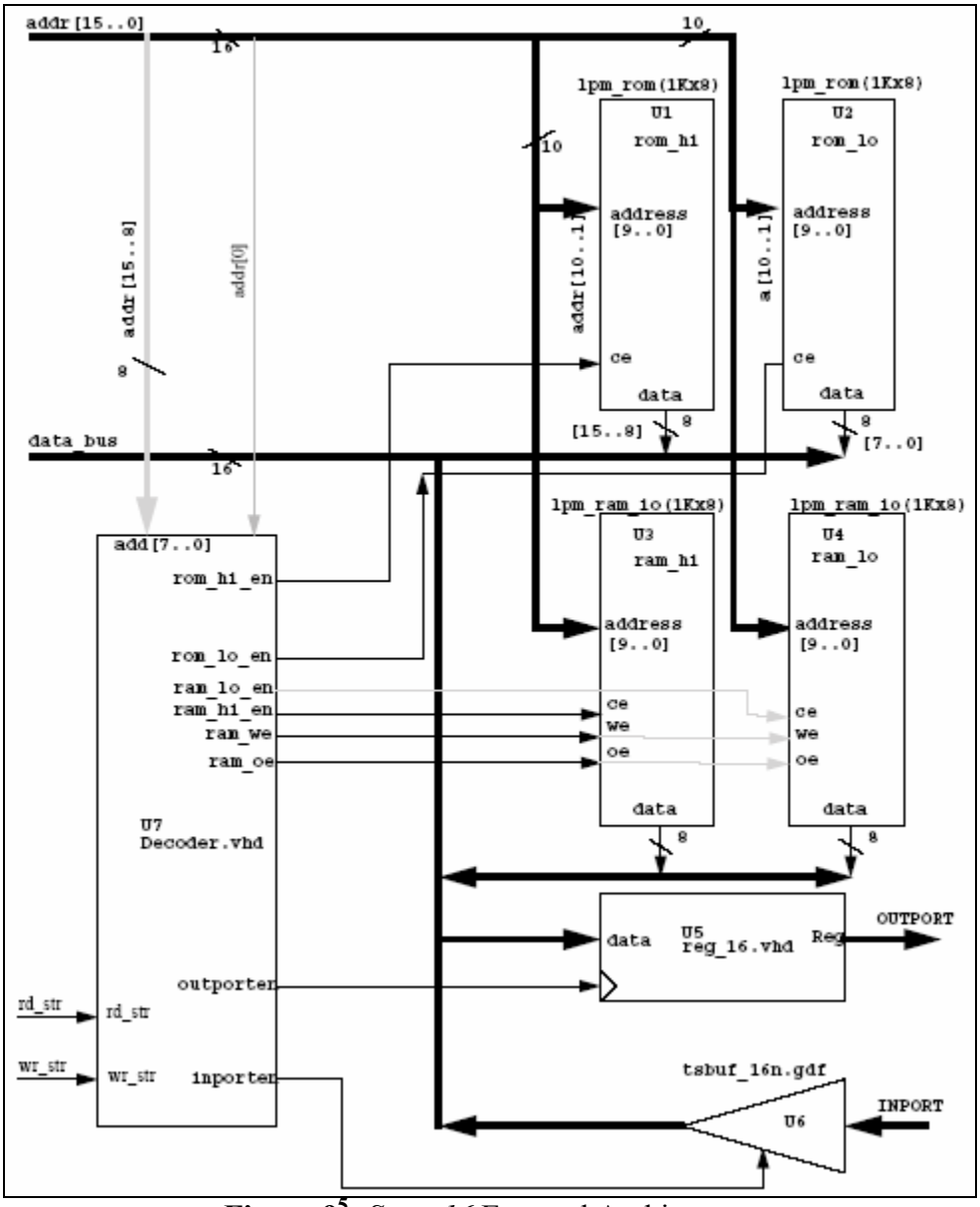

Figure 9[5]**:** *Sweet16* External Architecture

It is obvious that the inputs to this system are the data bus, the inport bus, the address bus, and the strobe signals. The only output (excluding the bidirectional data bus) is the outport bus. Figure 10 on the next page shows the Max+Plus II Graphic Design File that was created by integrating the previously designed components. See Appendix F, Specification Sheet 9 for a description of the complete External Architecture.

[5] http://www.hcs.ufl.edu/~radlinsk/eel4713/course/view.php?id=2

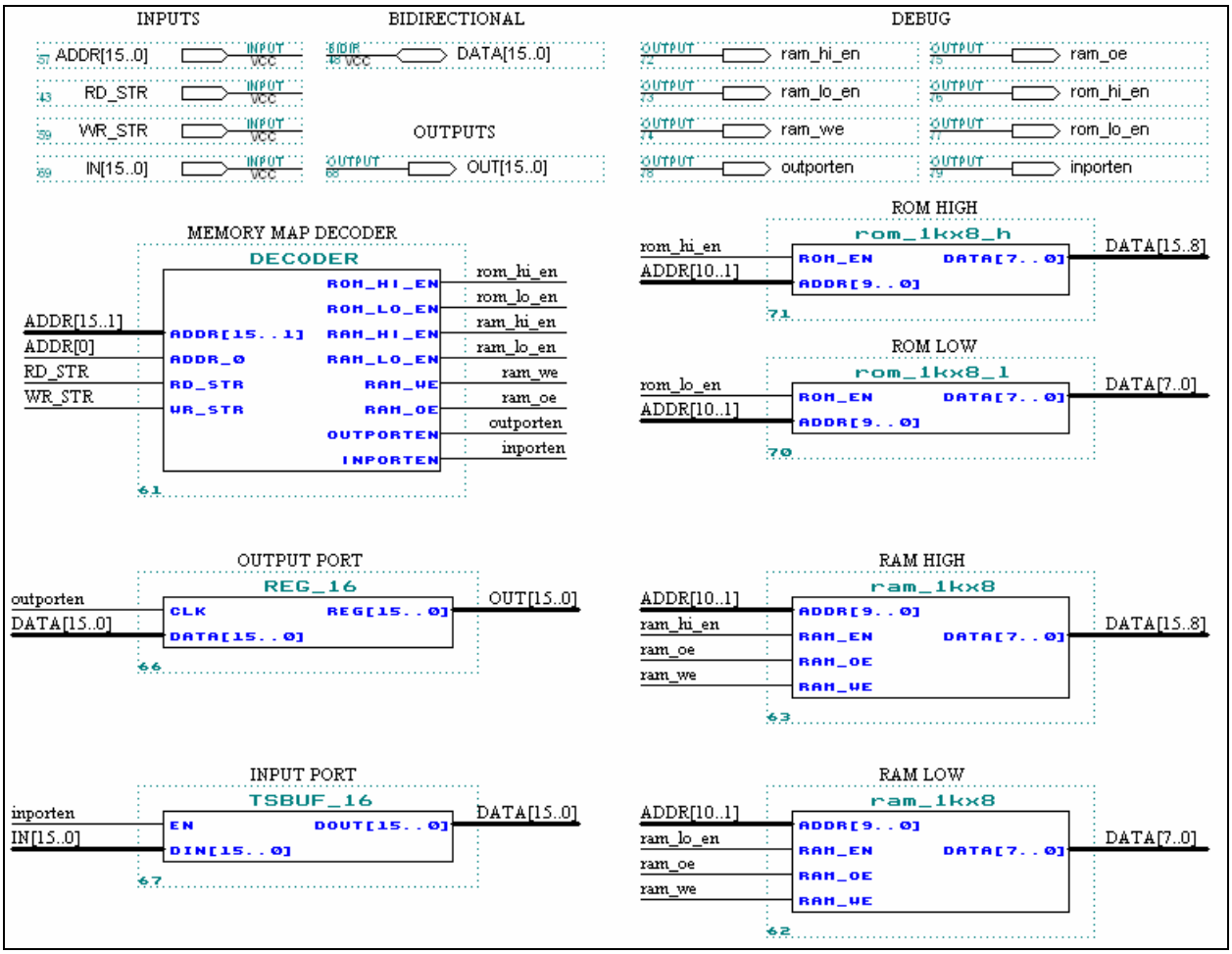

Figure 10: Graphic design for External Architecture

A. System Test

The operation of this system was tested with a series of input vectors. This test was designed to verify the proper functioning of the Memory Decoder as well as the I/O and memory components. Each area of the memory map was appropriately read from or written to so as to examine the behavior of the external architecture under various circumstances. The verification performed in lab is shown in Figure 11 on the next page. A more involved simulation, performed prior to the laboratory demonstration, can be found in Appendix E, Waveform Simulation 8.

With both of these simulations, an exhaustive test was performed on the *Sweet16* external architecture. The system as a whole proved to perform as desired.

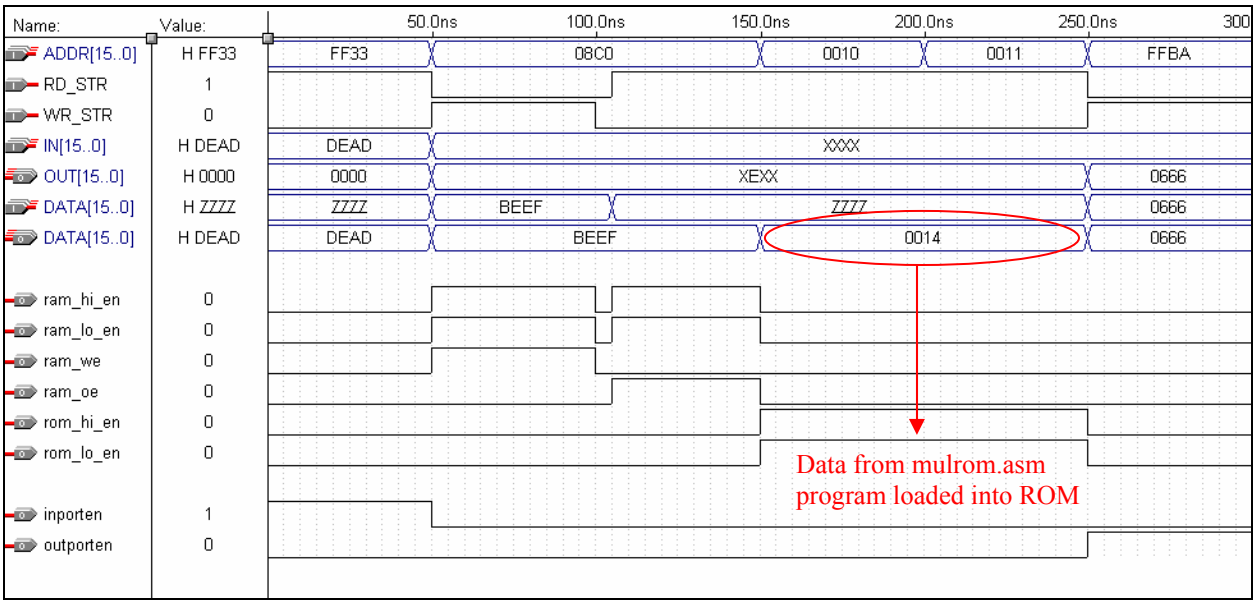

Figure 11: External Architecture simulation

Conclusion

A. Summary

The system designed in this lab represents the *Sweet16* External Architecture. This system interfaces with the previously designed *Sweet16* CPU to form a complete microprocessor. It is the external architecture that allows to the whole system to interface with the outside world. The ability to read from and write to memory, as well as the capability of transmitting and receiving data from exterior components, is integral to the overall functionality of a computing system.

The external architecture designed in this lab represents a mere fraction of this system's potential. It is entirely possible to expand upon this architecture in terms of serial communication, VGA interfacing, digital to analog conversion, and much more. The Instruction Set Architecture of this microprocessor makes it capable of expanding to various applications.

B. Questions

1. *Explain why addr[0] was not connected to the ROM and RAM instances.*

   The RAM and ROM components are each 8 bits wide; however, the *Sweet16* is a 16-bit processor with a 16-bit data bus. Therefore, with the smallest addressable memory space being 8 bits (this will be exploited in future labs), the least-significant bit of the address is neglected so that two bytes (the low and the high) can be read at once and concatenated to form a 16-bit word.

2. *Notice that addr[0] could be used to discriminate between data in the devices on the "upper-" or "lower-half" of the data bus. Write the RAM chip enable equations if one wished to enable the RAM on the upper half of the data bus when addr[0] = 0 and the RAM on the lower half of the data bus when addr[0] = 1. Identify whether this arrangement is "Big Endian" or "Little Endian" in structure. Explain your reasoning.*

```
RAM_HI_EN   <=    not ADDR(15) and not ADDR(14) and not ADDR(13) and
                  not ADDR(12) and     ADDR(11) and not ADDR(0)  and
                  (RD_STR or WR_STR);

RAM_LO_EN   <=    not ADDR(15) and not ADDR(14) and not ADDR(13) and
                  not ADDR(12) and     ADDR(11) and     ADDR(0)  and
                  (RD_STR or WR_STR);
```

Listing 3: RAM chip enable equations

This arrangement is "Big Endian" because the most-significant byte comes first in memory (even addresses starting with zero), and the least-significant byte come second in memory (odd addresses starting with one).

3. *Account for the reason, during testing of a bus connected to a port signal characterized as "bidir" or "inout" that there were two signals presented in the waveform editor's diagram, one labeled "in" the other labeled "out". Why did you have to initialize the "in" signal to "ZZZZ" during times that the bus was being driven "out"? What happened if this point was neglected?*

The bidirectional port signal has an input and an output because it can be driven either internally or externally. That is to say that when the bus is being driven by some device in the system, the input port of the bidirectional bus may not be driven externally. In such situations, the input for the bus must be forced to high impedance so as not to cause a data collision. Similarly, when no internal devices are driving the bidirectional bus, the input for the bus may be driven by the user (or an external device) without fear of a data collision. If this precaution is ignored, the simulator will produce a "logic contention" error notifying the user that he/she is attempting to drive the bus from two different sources.

Lab No. 5
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 create the microprogram necessary to implement several instructions in the *Sweet16* Instruction Set. Of particular interest were the instructions contained in the *mulrom.asm* multiplication subroutine (see Appendix C, Program 1 for the *mulrom.asm* code). A genuine understanding of the *Sweet16* architecture was necessary in order to compose the microcode necessary to realize each instruction.

With the *Sweet16* microprocessor architecture completely assembled, the final tasks in making this design a fully-functioning system is to provide it with a set of microinstructions that describe how to manipulate its individual components in order to accomplish a given task. The resulting microcode will be the true brains behind this processor. It will be responsible for "tweaking" the control signals so as to coax the processor into computing meaningful values that the programmer can make use of.

Component Design and Validation

Up to this point in the design of the *Sweet16* microprocessor, we have the *Sweet16* CPU (*sw16cpu.gdf*) and the *Sweet16* External Architecture (*sw16extarch.gdf*). In this lab, these components were combined into one architecture (*sweet16.gdf*). Once this was accomplished, the microinstructions required to implement several of the *Sweet16* instructions were written. Once all of these components were combined, the *Sweet16* could then execute a multiplication program, *mulrom.asm*.

A. Sweet16 Microprocessor Architecture

At a very high level, the *Sweet16* microprocessor is composed of the *Sweet16* CPU (see Appendix F, Specification Sheet 5 for a description of this component) and the *Sweet16* External Architecture (see Appendix F, Specification Sheet 9 for a description of this component). These two components were combined into one graphical design according to the schematic drawing in Figure 1 on the next page.

The complete *Sweet16* microprocessor hardware, as viewed from the highest level, is shown in Figure 2 on the next page. There are three inputs to this processor: the system clock, the reset signal, and the input port data bus. The only real output is the output port data bus. However, extra output signals were brought out of the design so that several important internal signals could be monitored. These signals are labeled "Test Signals" in Figures 1 and 2.

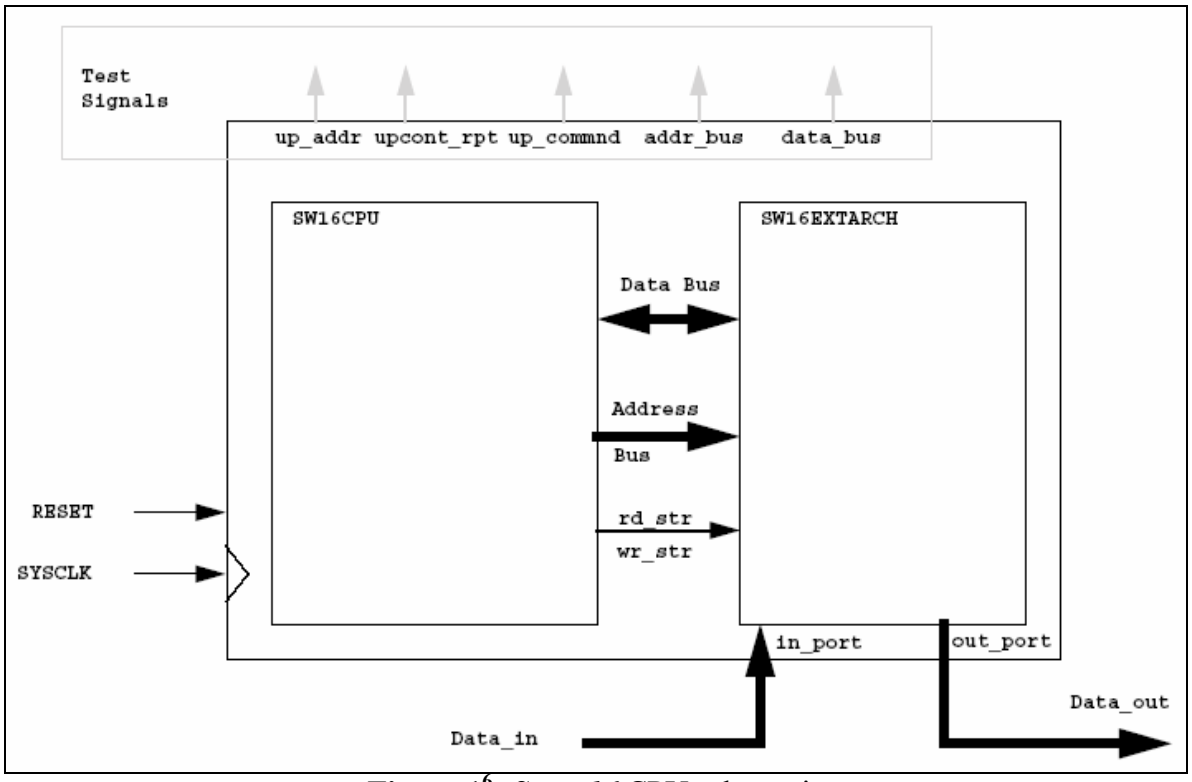

Figure 1[6]: *Sweet16* CPU schematic

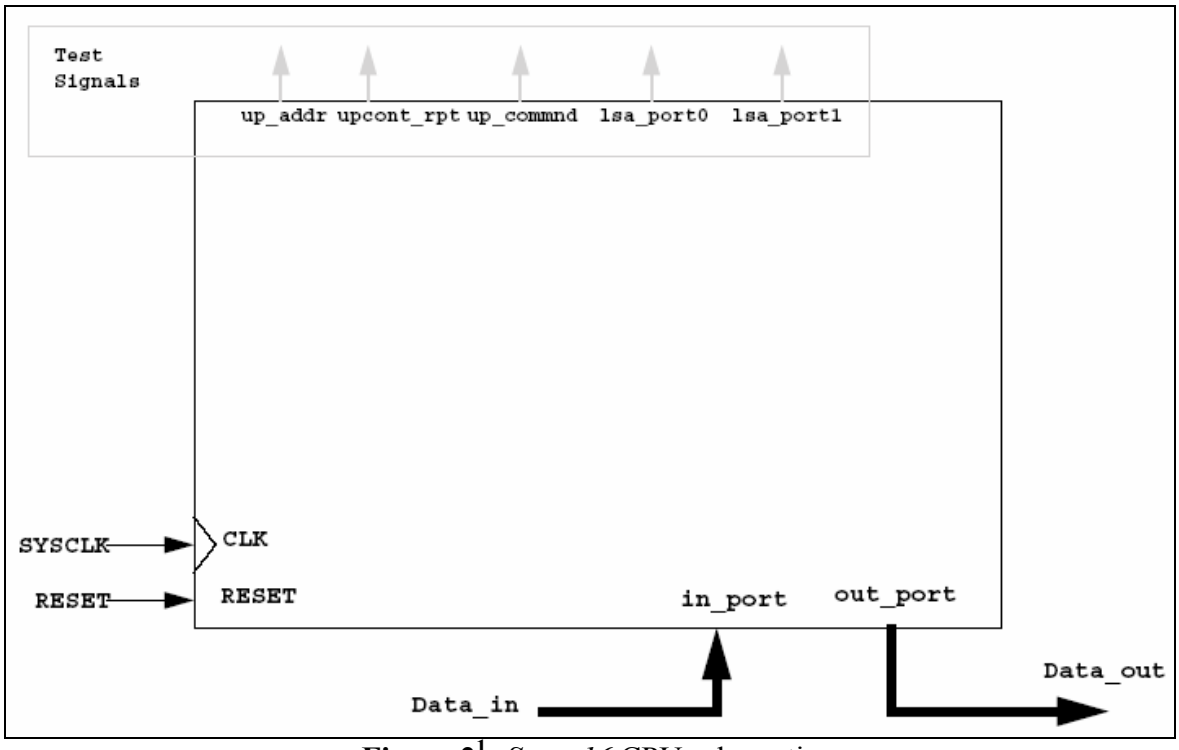

Figure 2[1]: *Sweet16* CPU schematic

[6] http://www.hcs.ufl.edu/~radlinsk/eel4713/course/view.php?id=2

The two high-level components that make up the *Sweet16* were combined in Max+Plus II's Graphic Editor. The resulting graphic design file is shown below in Figure 3.

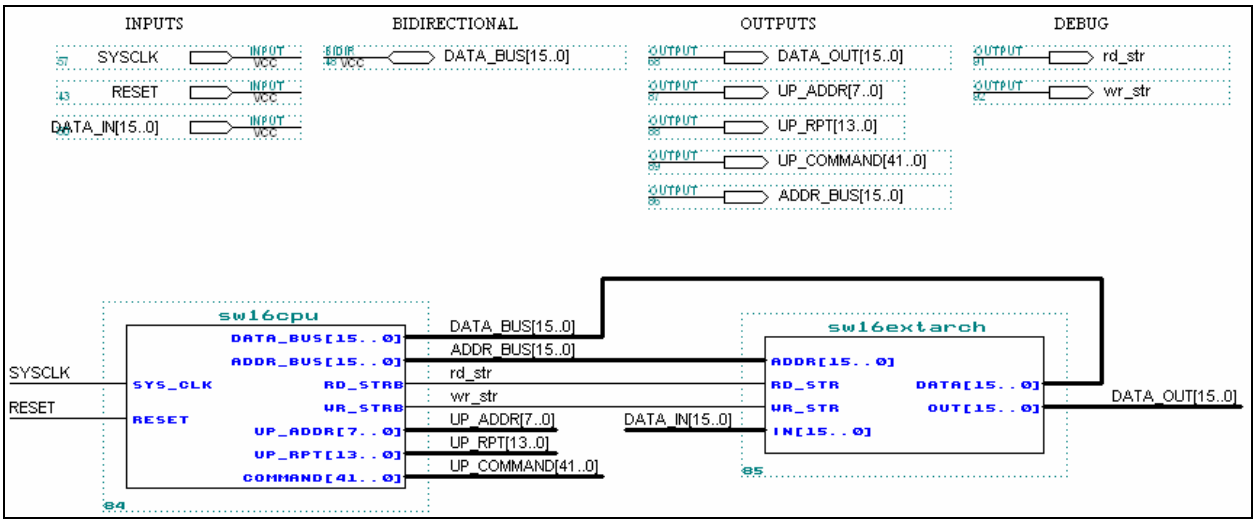

Figure 3: *Sweet16* CPU high-level design

Notice that these two components use the address bus, data bus, and the read and write strobes to communicate between each other. The result of combining these two entities, *sweet16.gdf*, is shown below in Figure 4 (see Appendix F, Specification Sheet 10 for a description of the *Sweet16* component).

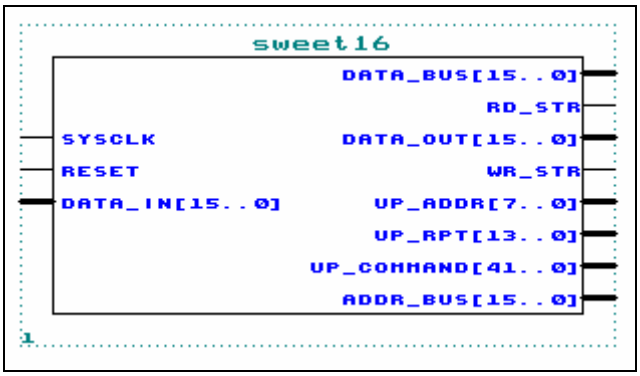

Figure 4: *Sweet16* CPU highest-level design

This design was loaded with the multiplication program *mulrom.asm*. Despite the fact that a majority of the microprogram memory was empty, this program was simulated to examine the internal behavior of the *Sweet16*. See Appendix E, Waveform Simulation 9 for the results of this simulation.

B. Sweet16 Microcode

To complete the *Sweet16* microprocessor design, the last step was to compose the microcode that enables the *Sweet16* instructions to be executed. When executing a program, this processor converts each opcode into an address that points to a certain portion of the microcode wherein the microinstructions necessary to execute each macroinstruction reside. As mentioned in earlier labs, this task is accomplished by the MapROM. The Microprogram Memory, to which the MapROM points, must be written in assembly language and converted into a Memory Initialization File (MIF). The structure of this assembly language program is organized by instruction and mapped out by the MapROM. Each *Sweet16* instruction will have a sequence of microinstructions associated with it.

The Microprogram Memory contains several important subroutines as well. For example, every instruction execution cycle consists of fetch, decode, and execute. Therefore a subroutine was created to accomplish the fetch portion of the instruction execution cycle. This subroutine retrieves the data at the memory location pointed to by the Program Counter (PC), increments the PC twice to point at the next data/instruction word, and directs the microprogram flow towards the address pointed to by the MapROM (essentially the decode portion of the cycle). The microcode for the fetch subroutine is shown below in Listing 1.

```
* Begin Instruction Fetch/Decode Sequence ------------
* IR <= mem[PC]; PC <= PC + 2; - 3 microinstructions, 5 clock cycles
FETCH:
        CNTR_LOAD  2     * Load Loop Counter
        INC_REG    PC    * Increment the Program Counter
        DB_OUT           * Put RALU DB bus on internal databus
        STORE_MAR        * Store the PC to MAR via DB
        endsc

FETCH1:
        LOOP_TC    FETCH1
        MEM_READ         * Assert RD_STR, set databus inbound
        endsc

DECODE:
        JUMP_MAP         * Decode OPCODE, MAPROM[OPCODE]
        INC_REG    PC    * Increment the Program Counter
        MEM_READ         * Assert RD_STR, set databus inbound
        endsc
```

Listing 1: Microcode for Fetch subroutine

In addition to the fetch subroutine, subroutines for different address modes were also created. For example, the Immediate Address mode defines the word following the instruction word to be the data associated with the instruction. This subroutine, therefore, retrieves that data and adjusts the PC accordingly. In general, the goal of the subroutines associated with the various address modes is to retrieve data. The microcode for the immediate and absolute address modes is shown in Listing 2 and Listing 3, respectively, on the next page.

```
* Immediate Address Mode: Databus <= Mem[PC]; PC <= PC + 2;
* 3 microinstructions, 4 clock cycles
* The execute state must maintain MEM_READ
IMMEDIATE:
        CNTR_LOAD  1        * Load Loop Counter
        INC_REG     PC      * Increment the PC
        DB_OUT              * Put RALU DB bus on internal databus
        STORE_MAR           * Store the PC to MAR via DB
        endsc

IMMEDIATE1:
        LOOP_TC    IMMEDIATE1  * Wait for 2 cycles
        MEM_READ               * Assert RD_STR, set databus inbound
        endsc

        RETURN
        INC_REG     PC      * Increment the PC
        MEM_READ            * Assert RD_STR, set databus inbound
        endsc
```

Listing 2: Microcode for Immediate Address Mode

```
* Absolute Address Mode: Databus <= Mem[Mem[PC]]
* 6 microinstructions, 6 clock cycles
* The execute state must maintain MEM_READ
ABSOLUTE:
        INC_REG  PC     * Increment the PC
        DB_OUT          * Put RALU DB bus on internal databus
        STORE_MAR       * Store the PC to MAR via DB
        endsc

* Get second word of instruction
        INC_REG  PC     * Increment the PC
        MEM_READ        * Assert RD_STR, set databus inbound
        endsc

        MEM_READ        * Assert RD_STR, set databus inbound
        endsc

        STORE_MAR       * Load the MAR at the end of the state
        MEM_READ        * Assert RD_STR, set databus inbound
        endsc

* Get data pointed at by second word of instruction
        MEM_READ        * Assert RD_STR, set databus inbound
        endsc

        RETURN
        MEM_READ        * Assert RD_STR, set databus inbound
        endsc
```

Listing 3: Microcode for Immediate Address Mode

With these tools it was possible to construct the remainder of the *Sweet16* instructions in microcode. The instructions of interest for this lab were those utilized in the *mulrom.asm* multiplication program.

One of the most challenging instructions to implement was the *CALL* instruction. The flow chart developed to describe the steps that were necessary to execute this instruction is shown below in Figure 5.

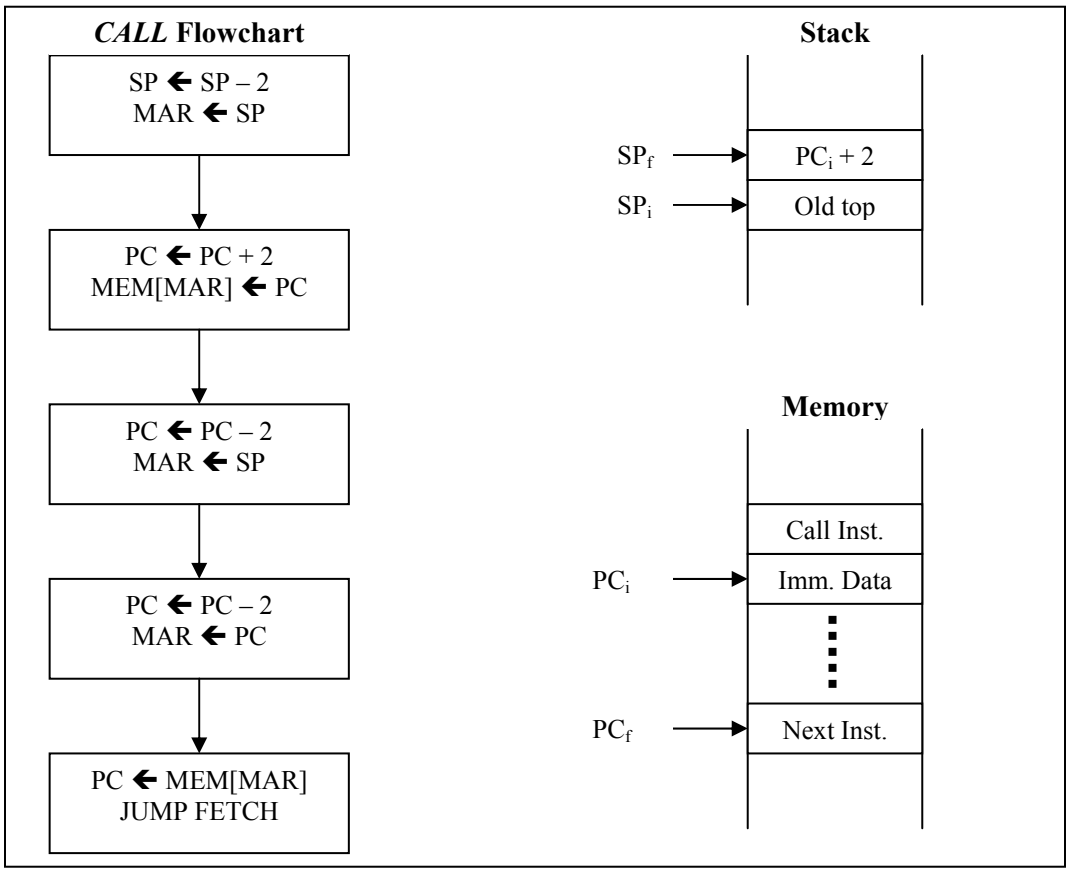

Figure 5: *CALL* execution flowchart

From this logic, the microcode shown in Listing 4 was composed in order to implement the *CALL* instruction. The code for this instruction, like many others, takes advantage of the microcontroller's ability to jump around in the Microprogram Memory. This means that the microcode for any instruction does not have to be contiguous, but rather it can be segmented at the programmer's discretion.

Many of the instructions executed in microcode call upon macros defined in the *usw16.mac* file. These macros simplify the task of setting (or clearing) the 42 control signals that allow the *Sweet16* to operate. See Appendix C, Program 10 for the complete *usw16.mac* file.

```
        ORGA    $51
CALL:
        DEC_REG SP
        endsc

        DEC_REG SP
        D_OUT
        STORE_MAR
        endsc

        INC_REG PC
        CNTR_LOAD       2
        endsc

        INC_REG PC
        D_OUT
        MEM_WRITE
        JUMP    FINISH_CALL1
        endsc
        ORGA    $E8
        *
        *
        *
FINISH_CALL1:
        LOOP_TC         FINISH_CALL1    * Wait for 2 cycles
        REG_TO_BUS      PC              * Put current PC on data bus
        MEM_WRITE
        endsc

        DEC_REG PC
        endsc

        DEC_REG PC
        D_OUT
        STORE_MAR
        CNTR_LOAD       1
        endsc

FINISH_CALL2:
        LOOP_TC         FINISH_CALL2    * Wait for 2 cycles
        MEM_READ                        * Assert RD_STR, set databus inbound
        endsc

        MEM_READ                        * Assert RD_STR, set databus inbound
        STORE_PC                        * Store new PC value
        JUMP    FETCH
        endsc
```

Listing 4: *CALL* instruction microcode

System Design and Validation

Once all of the instructions necessary to execute the *mulrom.asm* program were implemented in microcode, the microcode program, *usw16.asm*, was assembled with *UPASM* and converted into a Memory Initialization File (MIF). See Appendix C, Program 6 for the complete *usw16.asm* file.

A. System Test

The *usw16* MIF was loaded into the *Sweet16* Microprogram Memory. In addition, *mulrom.asm* was compiled and converted into two MIFs, *mulrom0.mif* and *mulrom1.mif*. As mentioned in earlier labs, this technique is employed due to the fact that the *Sweet16* has two 1k x 8 ROMs, one for the most significant 8-bits of the instruction/data word, and one for the least significant 8-bits. These two MIFs were loaded into their respective ROMs. The resulting Max+Plus II project was compiled and simulated for an appropriate length of time (40 µs).

The *mulrom.asm* program utilizes a series of shifts and adds in order to compute the product of two 16-bit numbers. In essence, whenever a one (1) is encountered in the multiplier, the current value of the product is shifted and added to the multiplicand. This process of shifting and adding is repeated 16 times so that each bit of the multiplier can be examined.

After 36.7 µs of simulation, the multiplication algorithm had completed its final iteration and had computed the correct product for $ABBA multiplied by $DABA. The multiplicand and the multiplier were predetermined by the laboratory instructor, and the results, $92B9 2924, were confirmed. Figure 6 below shows the portion of the simulation at which point the results of the multiplication were computed. See Appendix E, Waveform Simulation 10 for the abridged simulation of *mulrom.asm*.

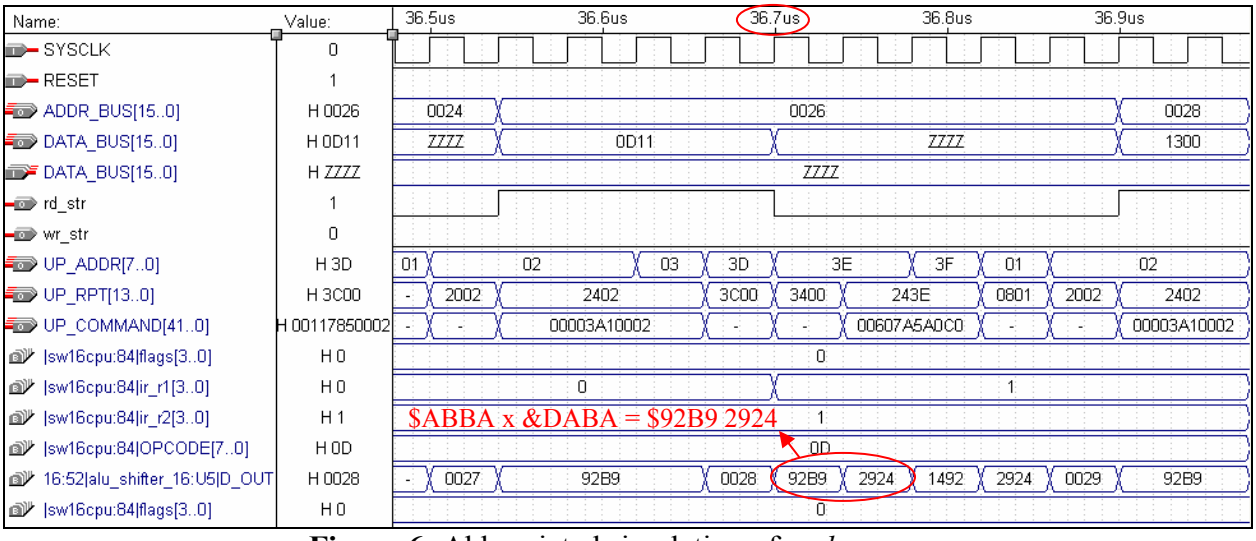

Figure 6: Abbreviated simulation of *mulrom.asm*

Conclusion

A. Summary

The *Sweet16* microprocessor underwent the final stages of its hardware design in this lab. The External Architecture was combined with the *Sweet16* CPU to form the *Sweet16* microprocessor. While not totally complete, this processor was also given a microprogram that enabled it to execute a multiplication subroutine. In the coming labs, this microprogram will be expanded upon to make a more complete *Sweet16* microprocessor.

The most interesting aspects of the design involved in this lab are the tools that were used to create the *Sweet16* microprogram. There does not exist a compiler designed specifically for the *Sweet16* microprocessor and its instruction set. Therefore existing tools (i.e. *UPASM*) were adapted and harnessed so that they could accommodate the *Sweet16*. This allowed the programmer to essentially devise the language in which the microcode was written. By defining macros and subroutines, the microcode was modularized to a significant degree. This greatly simplified the process of composing the microcode for the *Sweet16* Instruction Set.

Lab No. 6
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 examine the meaning and essence of the Complex Instruction Set Computer (CISC). In previous labs, intricate operations such as multiplication were accomplished by combining several less-intricate instructions, such as shifting and adding. Although this approach succeeded in computing the desired result, it proved to be an inefficient method of accomplishing its task in that it consumed more time than was necessary given the capabilities of the *Sweet16* architecture.

This lab harnessed the capabilities of the *Sweet16* and showed that complex processes could be accomplished faster when explicit instructions are devoted to these processes. This fact is based on the principle that software implementations of complex operations are costly in terms of the time they spend on fetching multiple instructions from memory. In a CISC architecture such as the *Sweet16*, whole instructions may be devoted to complex operations so as to avoid the added cost of multiple instruction fetches. Of particular interest in this lab are the benefits obtained from CISC-style implementations of multiplication and division.

Component Design and Validation

At this stage in the design of the *Sweet16* microprocessor, the hardware is all but complete, and many of the *Sweet16* instructions have been implemented in microcode. This lab will focus on expanding the microcode to include the unsigned multiplication, unsigned division, and 32-bit addition instructions.

A. Unsigned Multiplication

Unsigned binary multiplication may be approached in several ways. Single-cycle 16-bit multiplication can be accomplished; however such multipliers are very logic-intensive and therefore costly. Conversely, there are several iterative approaches to 16-bit unsigned binary multiplication. One impractical approach is to realize what multiplication really is: the repeated summing of one number with itself. Although this method seams to be ideal for multiplying small numbers, it becomes grossly impractical when dealing with larger numbers. This leads us to the more viable of the two iterative methods. Returning to an elementary-style of computing products, it becomes obvious that the best way to accomplish multiplication is to use a series of shifts and additions.

According to the multiplication algorithm used in this lab, for each binary “one” that appears in the multiplier, the multiplicand is added to the current product and then the product is shifted to account for the bit position of the “one” in the multiplier. This conditional addition is accomplished with the use of the built-in unsigned multiplication function in the ALU. The algorithm for 16-bit multiplication is similar to the algorithm for 4-bit multiplication illustrated in Figure 1 on the next page.

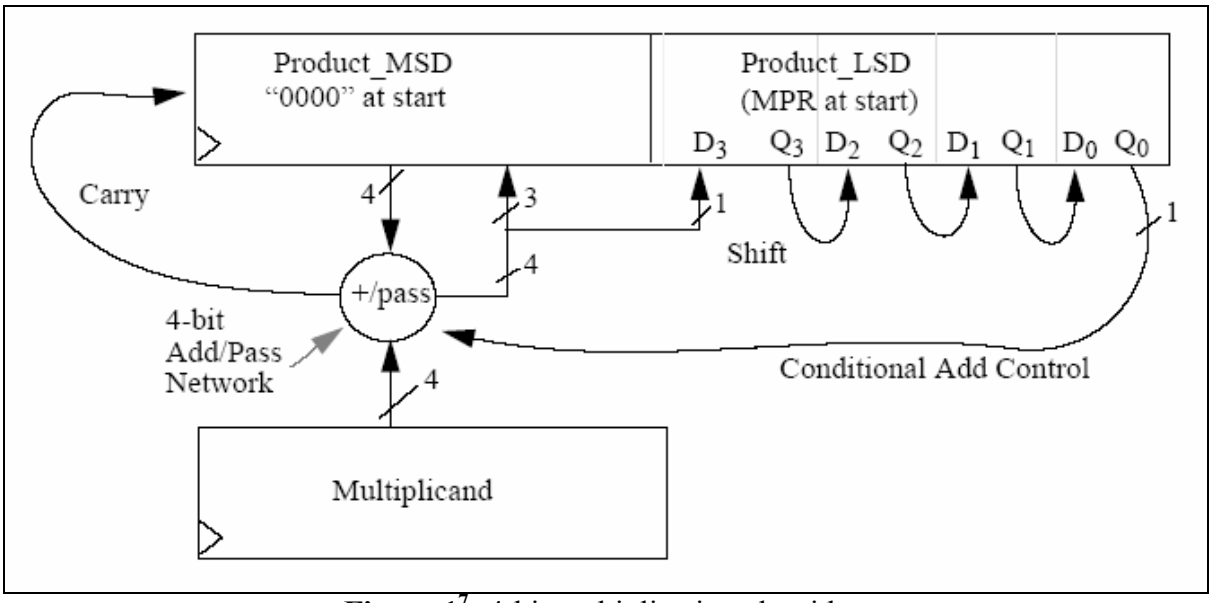

Figure 1[7]**:** 4-bit multiplication algorithm

From this method of multiplication, a flowchart was created to assist in developing microcode for the *UMUL* instructions. Figure 2 below shows the resulting multiplication flowchart.

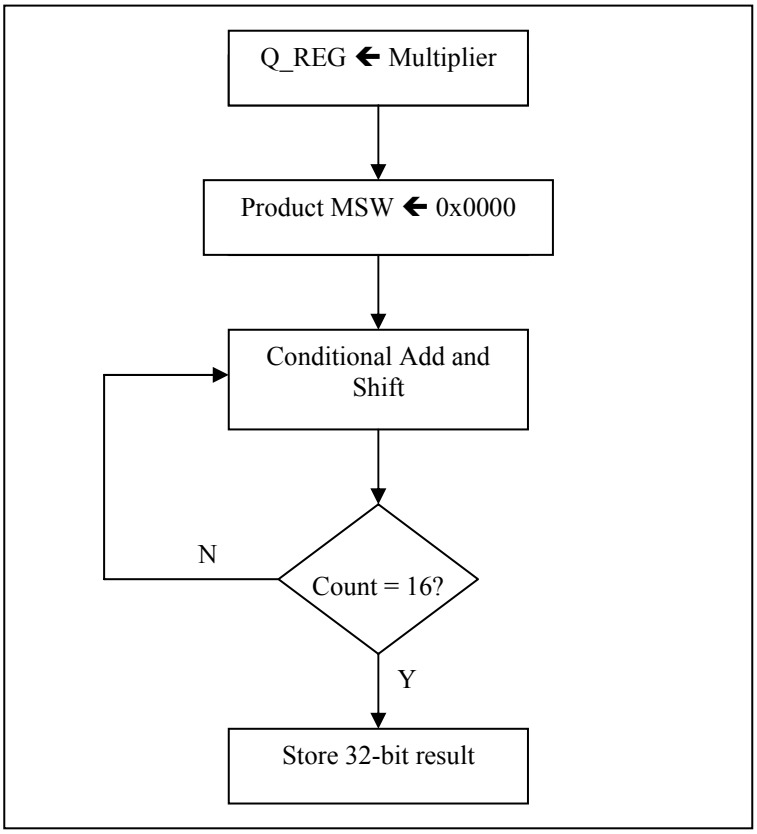

Figure 2: Flowchart for multiplication

[7] "unsigned_multiplication.pdf" By Michel Lynch, http://www.hcs.ufl.edu/~radlinsk/eel4713/course/view.php?id=2

When this algorithm was implemented in microcode, the following portion of assembly code was developed for the *UMULR* instruction (see Appendix C, Program 6 for the complete *usw16.asm* code).

```
        ORGA      $70
UMULR:
        REGMUX    ,R1B                * select multiplier to pass through ALU
        ALU       ,PASS_BC,C_ONE      * pass R1B through ALU
        Q_SHIFT   ,LOAD_D0            * load Q_REG with R1B
        JUMP      FINISH_UMUL_R
        endsc
        ORGA      $F0
FINISH_UMUL_R:
*       Zero out MSW of product
        REGMUX    ,R1B                * Write to R1
        ALU       ,F_ZERO,C_ZERO      * Zero out R1
        ALU_SHIFT ,PASS               * pass zero through alu_shifter
        SMUX      ALU_SHFT_BUS        * select alu_shift_bus as input to RA
        WRITE_REG_ARR       TRUE      * write result back using B_ADDR
        CNTR_LOAD 15                  * prepare to loop 16 times
        endsc

*       Multiplication loop
FINISH_UMUL_R2:
        LOOP_TC   FINISH_UMUL_R2      * repeat multiplication iteration
        REGMUX    R2A,R1B             * select Multiplicand (R2) and Product MSW (R1)
        ALU       REG_ARR,UMULIT,C_ZERO         * conditionally add multiplicand (R2) and
                                                *    Product MSW (R1)
        Q_SHIFT   FSO,SHFT_RIGHT      * shift Q_REG right (shift out Product LSb)
        ALU_SHIFT ALU_COUT,SHFT_RIGHT
        SMUX      ALU_SHFT_BUS        * select alu_shift_bus as input to RA
        WRITE_REG_ARR       TRUE      * write result back using B_ADDR
        FLAGS     ARITH
        endsc

*       Store results
        REGMUX    ,TOGGLE_B           * Write Product LSW to IR.R1 + 1
        PL_REG    ,ONE
        SMUX      QBUS                * write R2 with contents of Q_REG
        WRITE_REG_ARR       TRUE      * write result back using B_ADDR
        JUMP      FETCH
        endsc
```

Listing 1: *UMULR* implementation in microcode

The result of a 16-bit multiplication is a 32-bit number. Therefore the product of the two register contents (for *UMULR*) will be stored in the concatenation of those two registers, provided that the registers are an "even-odd" pair (at a word-aligned boundary). This is accomplished by exploiting the *B_ADDR(0)* bit that can be controlled by the controller. The least-significant word of the product is stored in the register [IR.R0 + 1] by toggling the *B_ADDR(0)* bit.

B. Unsigned Non-Restoring Division

The algorithm for unsigned non-restoring division is similar to that for unsigned multiplication. Instead of right shifts and addition, the algorithm for division essentially consists of subtractions and left shifts. Like in multiplication, the ALU designed for the *Sweet16* has a built-in function for non-restoring division. This function accomplishes the conditional subtraction that is necessary for division. The algorithm used to accomplish unsigned non-restoring division is illustrated in Figure 3 on the next page.

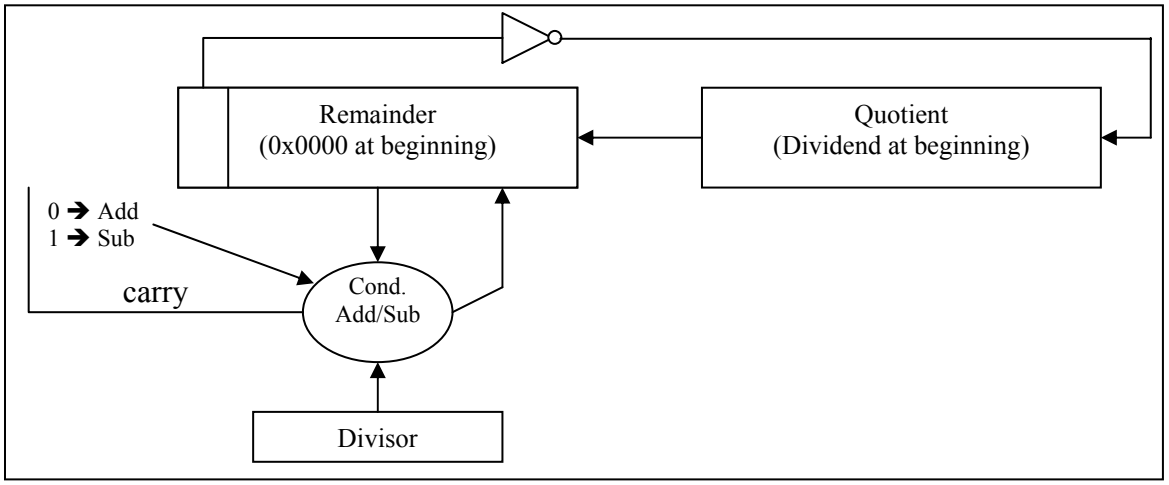

Figure 3: Unsigned non-restoring division algorithm

From this method of division, a flowchart was created to assist in developing microcode for the *UDIVR* instruction. Figure 4 below shows the resulting division flowchart.

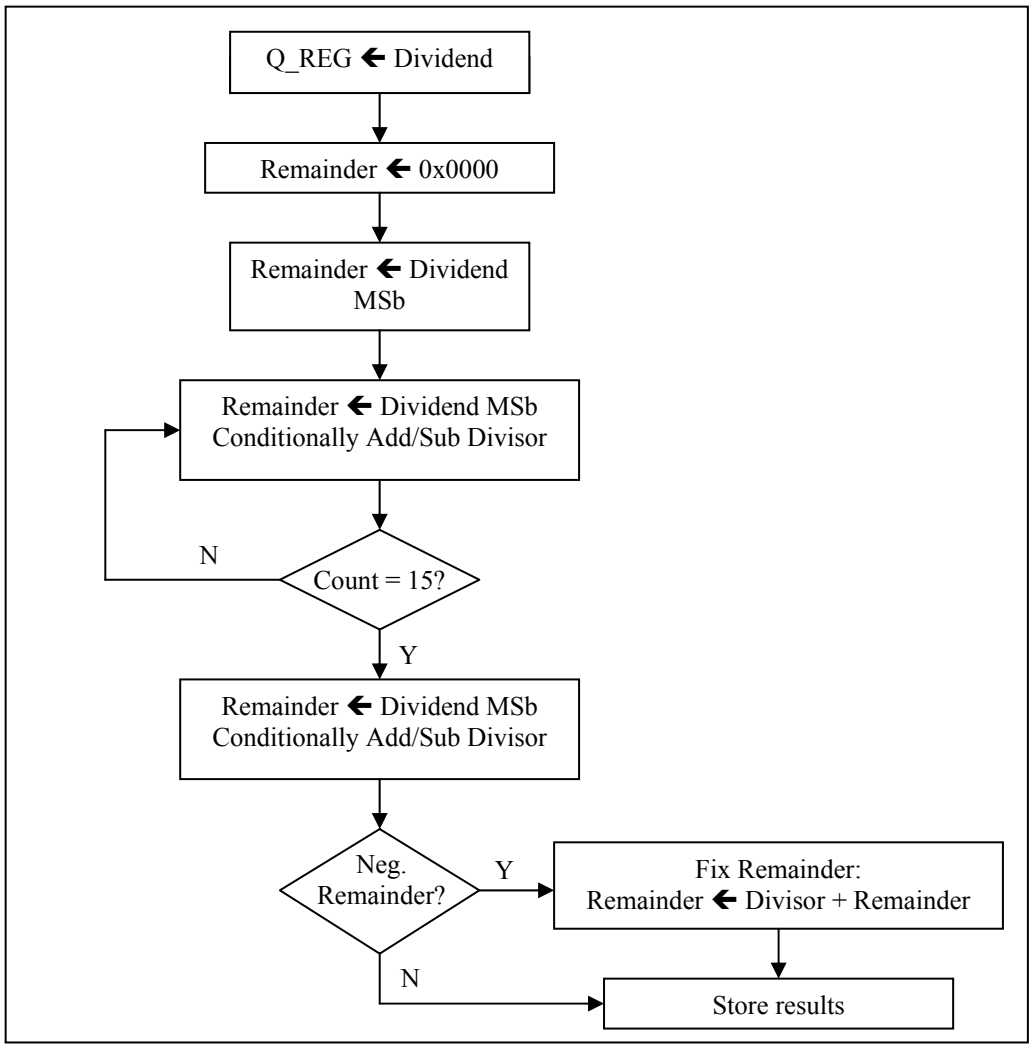

Figure 4: Flowchart for division

When this algorithm was implemented in microcode, the following portion of assembly code was developed for the *UDIVR* instruction.

```
        ORGA            $72
UDIVR:  REGMUX          ,R1B                    * select multiplier to pass through ALU
        ALU ,PASS_BC,C_ONE                      * pass R1B through ALU
        Q_SHIFT         ,LOAD_D0                * load Q_REG with R1B
        JUMP            FINISH_UDIV_R
        endsc

        ORGA            $F7
FINISH_UDIV_R:
        REGMUX          ,R1B                    * Write to R1 (remainder)
        ALU ,F_ZERO,C_ZERO                      * Zero out R1 (remainder)
        ALU_SHIFT       ,PASS                   * pass zero through alu_shifter
        SMUX            ALU_SHFT_BUS            * select alu_shift_bus as input to RA
        WRITE_REG_ARR   TRUE                    * write result back using B_ADDR
        FLAGS           uFlags                  * clear uSR_C, uSR_S
        endsc

        Q_SHIFT         ZERO,SHFT_LEFT          * shift Q_REG left (shift out Dividend MSb)
        ALU_SHIFT       QSO,SHFT_LEFT           * shift remainder left
        REGMUX          ,R1B
        ALU ,PASS_BC,C_ONE
        SMUX            ALU_SHFT_BUS            * select alu_shift_bus as input to RA
        WRITE_REG_ARR   TRUE
        CNTR_LOAD       14                      * prepare to loop 15 times
        endsc

        REGMUX          TMP,R1B
        ALU REG_ARR,SB_BAC,C_ONE
        WRITE_REG_ARR   FALSE
        FLAGS           uFlags
        endsc

FINISH_UDIV_R2:
        LOOP_TC         FINISH_UDIV_R2          * repeat division iteration
        REGMUX          R2A,R1B                 * select Divisor (R2) and Remainder (R1)
        ALU REG_ARR,NDIVT,                      * conditionally add/subtract
        Q_SHIFT         uSR_C,SHFT_LEFT         * shift Q_REG left (shift out dividend LSb)
        ALU_SHIFT       QSO,SHFT_LEFT
        SMUX            ALU_SHFT_BUS            * select alu_shift_bus as input to RA
        WRITE_REG_ARR   TRUE                    * write result back using B_ADDR
        FLAGS           uFlags
        endsc

        REGMUX          R2A,R1B                 * select Divisor (R2) and Remainder (R1)
        ALU REG_ARR,NDIVT,                      * conditionally add/subtract
        Q_SHIFT         uSR_C,SHFT_LEFT         * shift Q_REG left (shift out dividend LSb)
        ALU_SHIFT       ,PASS
        SMUX            ALU_SHFT_BUS            * select alu_shift_bus as input to RA
        WRITE_REG_ARR   TRUE                    * write result back using B_ADDR
        FLAGS           uFlags
        COND_JUMP       FLAGS_uSR_S,FIX_REMAINDER
        endsc

        REGMUX          ,TOGGLE_B               * write Quotient to IR.R1 + 1
        PL_REG          ,ONE
        SMUX            QBUS                    * write R2 with contents of Q_REG
        WRITE_REG_ARR   TRUE                    * write result back using B_ADDR
        JUMP            FETCH
        endsc

FIX_REMAINDER:
        REGMUX          R2A,R1B
        ALU REG_ARR,ABC,C_ZERO
        ALU_SHIFT       ,PASS
        SMUX            ALU_SHFT_BUS
        WRITE_REG_ARR   TRUE
        endsc

        REGMUX          ,TOGGLE_B               * write Quotient to IR.R1 + 1
        PL_REG          ,ONE
        SMUX            QBUS                    * write R2 with contents of Q_REG
        WRITE_REG_ARR   TRUE                    * write result back using B_ADDR
        JUMP            FETCH
        endsc
```

Listing 2: *UDIVR* implementation in microcode

The result of a 16-bit division is a 16-bit quotient and a 16-bit remainder. Therefore the results of a *UDIVR* instruction will be stored in the concatenation of the two registers that contained the operands, provided that the registers are an "even-odd" pair (at a word-aligned boundary). This is accomplished by exploiting the *B_ADDR(0)* bit that can be controlled by the controller. The remainder is stored in register [IR.R0], and the quotient is stored in register [IR.R0 + 1] by toggling the *B_ADDR(0)* bit.

C. 32-Bit Long Addition

The final complex instruction implemented in this lab was 32-bit long addition, or "Add with Carry Long." This instruction adds two numbers that are each 32-bits long and are located in the General Purpose Registers. Each 32-bit number requires two registers to hold its most- and least-significant parts. "Even-odd" register pairs are used to house each number in the register array. For example, if you execute the instruction *ADCLR R0,R2*, then the fist 32-bit long word is stored in registers 0 and 1, and the second 32-bit long word is stored in registers 2 and 3. By convention, the 32-bit result is stored in the concatenation of register [IR.R1] and [IR.R1 + 1].

The flowchart in Figure 5 below illustrates the algorithm used to accomplish 32-bit long addition.

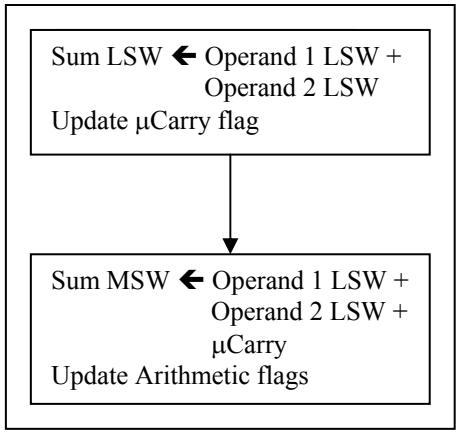

Figure 5: 32-bit long addition algorithm

From this algorithm, the microcode for the *ADCLR* instruction was developed. Listing 3 on the next page shows the resulting microcode.

```
     ORGA    $E2
ADCLR:
     REGMUX  TOGGLE_A,TOGGLE_B
     PL_REG  ONE,ONE
     ALU     REG_ARR,ABC,C_MACRO
     FLAGS   uFlags
     ALU_SHIFT       ,PASS   * pass through alu_shifter
     SMUX    ALU_SHFT_BUS    * select alu_shift_bus as input to Reg_Arr
     WRITE_REG_ARR   TRUE    * write result back using B_ADDR
     endsc

     REGMUX  TOGGLE_A,TOGGLE_B
     PL_REG  ZERO,ZERO
     ALU     REG_ARR,ABC,C_MICRO
     FLAGS   ARITH
     ALU_SHIFT       ,PASS   * pass zero through alu_shifter
     SMUX    ALU_SHFT_BUS    * select alu_shift_bus as input to RegArr
     WRITE_REG_ARR   TRUE    * write result back using B_ADDR
     JUMP    FETCH
     endsc
```

Listing 3: *ADCLR* implementation in microcode

System Design and Validation

Once the new, complex instructions were implemented in microcode, test programs were developed for each instruction and simulations were performed to evaluate the performance of these operations.

A. Unsigned Multiplication Verification

A special test program was written to isolate and test the 16-bit unsigned multiplication instructions *UMULR* and *UMULI*. These instructions both perform unsigned multiplication, except one utilizes the Register-Register address mode while the other utilizes the Immediate address mode. The test program developed to verify the correctness of these instructions, *umul_test.asm*, is shown in Listing 4 on the next page.

The updated *usw16* Memory Initialization File (MIF) was loaded into the *Sweet16* Microprogram Memory (see Appendix C, Program 6 for the entire *usw16.asm* code). In addition, the test program was compiled and loaded into the *Sweet16* ROM (see Appendix C, Programs 12, 13, and 14 for the *umul_test.s*, *umul_test0.mif*, and *umul_test1.mif* files, respectively). A simulation was performed to examine the *Sweet16*'s behavior during the execution of the multiplication instructions. Figure 6 on the next page shows a portion of the simulation results (see Appendix E, Waveform Simulation 11 for the complete simulation results).

```
* UMUL_TEST.ASM - Program that tests the operation of the UMULR and UMULI functions
*                  Orged in ROM ($0000)
* Author: Casey T. Morrison, EEL 4713, 2/28/2004

        NOLIST
        INCLUDE "sweet16.mac"
        LIST

        ORG     $0000

*       Multiply using UMULR
        LDI     R0,$F00D
        LDI     R1,$BEEF

        UMULR           R0,R1

*       Save results in R2 ans R3
        LDR  R2,R0
        LDR  R3,R1

*       Multiply using UMULI
        LDI     R0,$DEAD

        UMULI           R0,$BEA7

*       Save results in R4 ans R5
        LDR  R4,R0
        LDR  R5,R1

*       Infinite loop
*       Display results of first multiplication
*       R2:  $B309
*       R3:  $C223
*       R4:  $A5D5
*       R5:  $A8DB
LOOP:   LDR  R2,R2
        LDR  R3,R3
        LDR  R4,R4
        LDR  R5,R5
        CLRC
        BCC  LOOP

        END
```

Listing 4: *UMUL_Test.asm* code

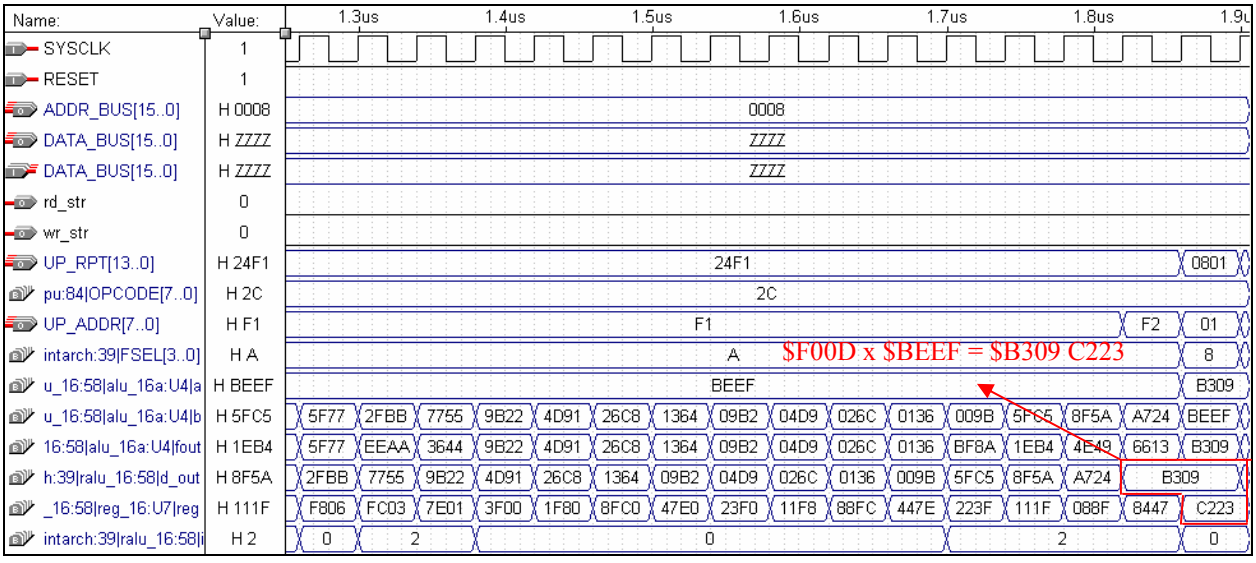

Figure 6: Abbreviated simulation of *UMUL_Test.asm*

Upon analyzing the results of this simulation, it was determined that the multiplication instructions performed as desired.

B. Non-Restoring Division Verification

A test program was written to isolate and test the 16-bit non-restoring division instruction *UDIVR*. This test program, *udiv_test.asm*, was developed to verify the correctness of the *UDIVR* instruction and is shown in Listing 5 below. After being compiled and loaded into the *Sweet16* ROM, a simulation was performed to examine the *Sweet16*'s behavior during the execution of the division instruction (see Appendix C, Programs 16, 17, and 18 for the *udiv_test.s*, *udiv_test0.mif*, and *udiv_test1.mif* files, respectively). Figure 7 on the next page shows a portion of the simulation results (see Appendix E, Waveform Simulation 12 for the complete simulation results).

```
* UDIV_TEST.ASM - Program that tests the operation of the UDIV function
*                    Orged in ROM ($0000)
* Author: Casey T. Morrison, EEL 4713, 2/28/2004


        NOLIST
        INCLUDE "sweet16.mac"
        LIST

        ORG     $0000

*       Divide using UDIV (R0/R1)
        LDI    R0,$F00D
        LDI    R1,$000F

        UDIVR  R0,R1

*       Infinite loop
*       Display results of division
*       R2:  $000D   Remainder
*       R3:  $1000   Quotient
LOOP:   LDR  R2,R0
        LDR  R3,R1
        CLRC
        BCC  LOOP

        END
```

Listing 5: *UDIV_Test.asm* code

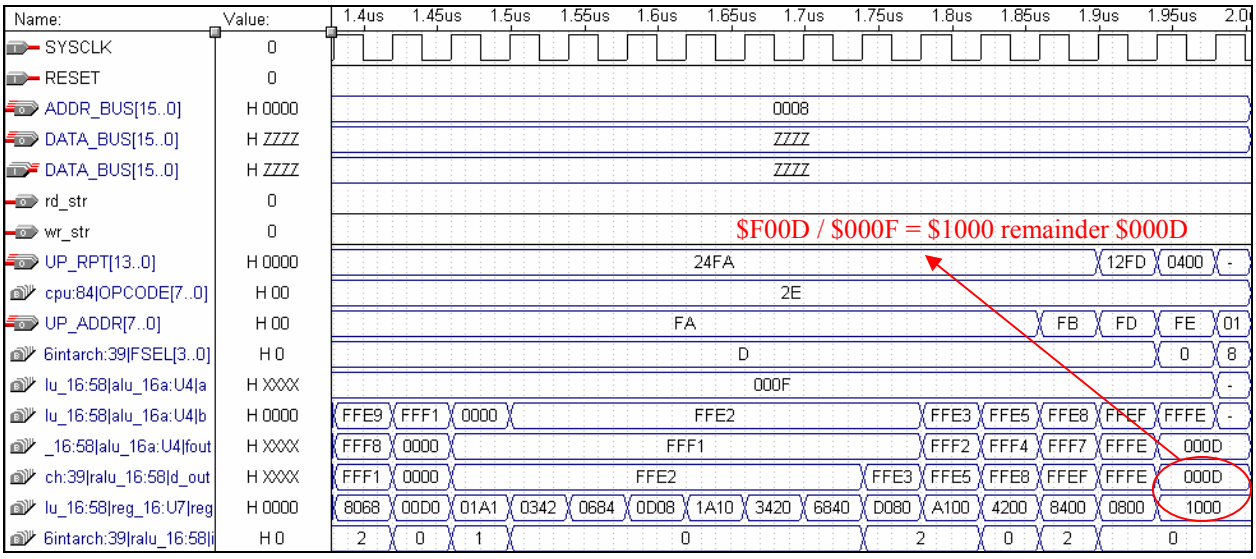

Figure 7: Abbreviated simulation of *UDIV_Test.asm*

Upon analyzing the results of this simulation, it was determined that the division instruction performed as desired.

C. 32-Bit Long Addition Verification

A test program was written to isolate and test the 32-bit long addition instruction *ADCLR*. This test program, *adclr_test.asm*, was developed to verify the correctness of the *ADCLR* instruction and is shown in Listing 6 on the next page. After being compiled and loaded into the *Sweet16* ROM, a simulation was performed to examine the *Sweet16*'s behavior during the execution of the long addition instruction (see Appendix C, Programs 20, 21, and 22 for the *adclr_test.s*, *adclr_test0.mif*, and *adclr_test1.mif* files, respectively). Figure 8 on the next page shows a portion of the simulation results (see Appendix E, Waveform Simulation 13 for the complete simulation results).

Upon analyzing the results of this simulation, it was determined that the 32-bit long addition instruction performed as desired.

```
* ADCLR_TEST.ASM – Program that tests the operation of the ADCLR function
*                    Orged in ROM ($0000)
* Author: Casey T. Morrison, EEL 4713, 2/28/2004

        NOLIST
        INCLUDE "sweet16.mac"
        LIST

        ORG     $0000

        LDI     R0,$F00D
        LDI     R1,$F00D
        LDI     R2,$BEEF
        LDI     R3,$000F

        ADCLR   R0,R2

*       Infinite loop
*       Display results of addition
*       R2:     $AEFC
*       R3:     $F01C
*       Carry:  1
LOOP:   LDR     R2,R0
        LDR     R3,R1
        CLRC
        BCC     LOOP

      END
```

Listing 6: *ADCLR_Test.asm* code

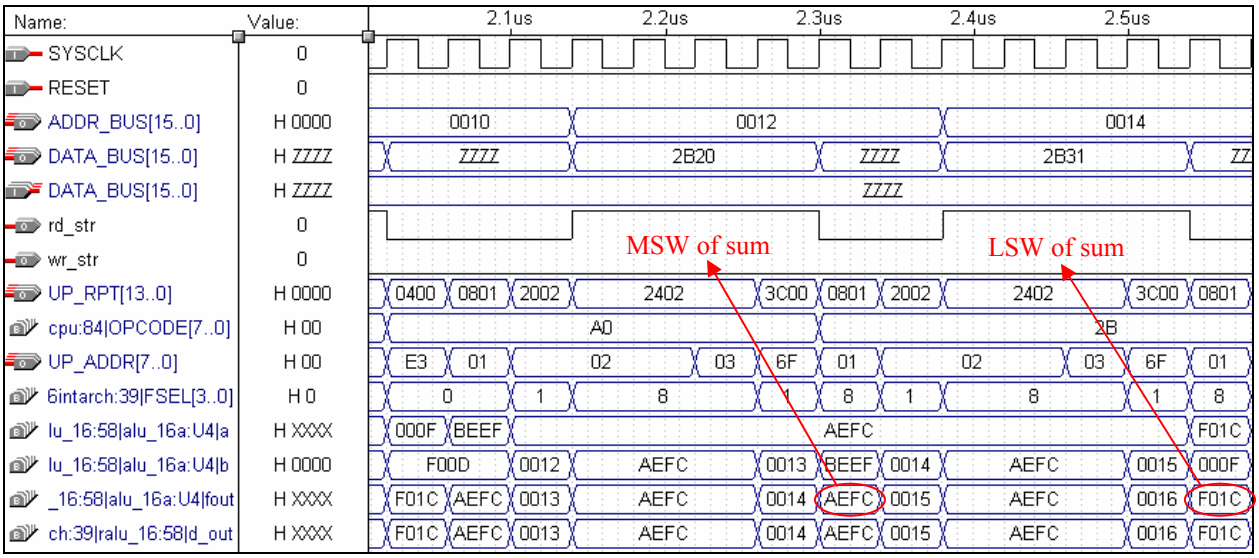

Figure 8 Abbreviated simulation of *ADCLR_Test.asm*

Conclusion

A. Summary

The Complex Instruction Set nature of the *Sweet16* microprocessor was exploited in this lab to implement three complex instructions. Although these instructions could have been accomplished in software, the benefits of devoting explicit opcodes to these operations were obvious. Instead of fetching several instructions from memory in an iterative fashion, only one instruction was fetched from memory and the computational power of the *Sweet16* was utilized to obtain the results in a fraction of the time. This is the essence of Complex Instruction Set Computing.

The advantages inherent in this method of computing can be extended to other instructions. For example, signed multiplication and division (instead of unsigned) can be accomplished with the existing hardware. It is this computational power that makes the *Sweet16* a very versatile and robust machine.

B. Questions

1. *How many clock cycles are needed for each algorithm? Does this number depend on the values of the data?*

   The unsigned multiplication instructions take 19 clock cycles once the data has been retrieved. The division instruction takes 21 or 22 cycles depending on whether the remainder must be adjusted at the end. The long addition instruction takes two clock cycles. The number of clock cycles for the multiplication and long addition instructions does not depend on the data. The number of clock cycles for the division instruction, however, does depend on the data; for some divisions require the remainder to be adjusted at the end.

2. *Which control paths for each algorithm do not pass through the microprogrammed controller in the multiplication and division instructions? Identify the flip-flop that each control path begins on, ends on. At which point does the control path become a data combination path?*

   The “iterate” control path does not go through the microprogrammed controller in the multiplication and division instructions. This control signal is generated by either the output of the Q-Shifter or the micro carry flag. The “iterate” control path determined the operation performed by the ALU during the multiplication and division iterations.

   For the multiplication algorithm, the control path begins at the Q-Shifter, where the least-significant bit of the multiplier determined whether or not to sum the multiplicand and the product. This control path ends at the register array, where the results of the sum are stored.

   For the division algorithm, the control path begins at the register array with the micro carry flag that determines whether or not the ALU adds or subtracts the remainder and divisor.

This path ends at the register array as well where the results of the addition or subtraction are stored.

Both of these control paths become data combination paths once they reach the ALU, where additions and/or subtractions are performed based on the "iterate" control signal.

3. Compare the number of clock cycles needed to perform the original mulrom.asm program, i.e., using a UMUL subroutine, with the number required by the UMULR instruction. Discuss how the CISC concept resulted in a faster machine. Hint: How many clock cycles were used fetching instructions?

The *UMUL* subroutine in the *mulrom.asm* program took 893 clock cycles to execute completely. This is more than 37 times greater than the 24 clock cycles that it took to execute the *UMULR* instruction. Much of this difference is attributed to the repeated instructions fetches (at five clock cycles a piece) that the *UMUL* subroutine relies upon. This illustrates the advantage of the CISC architecture—the concept that complex instructions can avoid costly memory fetches and accomplish operations faster than their software equivalent.

4. Discuss the difference in the execution time of a program using the address mode examined in procedure 6 with one not using it.

The use of memory-indirect address modes incurs additional execution time attributed to the added memory fetch(es) that is/are necessary. While zero additional memory fetches are requires for Register-Register address mode instructions, and one additional memory fetch is required for Immediate address mode instructions, two additional memory fetches are required for the memory-indirect address mode instructions. The first additional fetch is to retrieve the pointer, and the second additional fetch is to retrieve the data pointed to by the pointer. Thus instructions employing this address mode will incur the additional execution time associated with the additional memory fetches.