RISC Processor Final Report

Bradley Jomard Quinn Magendanz

Introduction

The goal of this project was to implement a basic processor that can be used to run a reduced instruction set assembly language to implement features such as simple applications or a file system. Our design was based on the MIT 6.004 Beta for the initial set of instructions, and from there we implemented extra instructions and features, as well as some complex instructions.

Summary

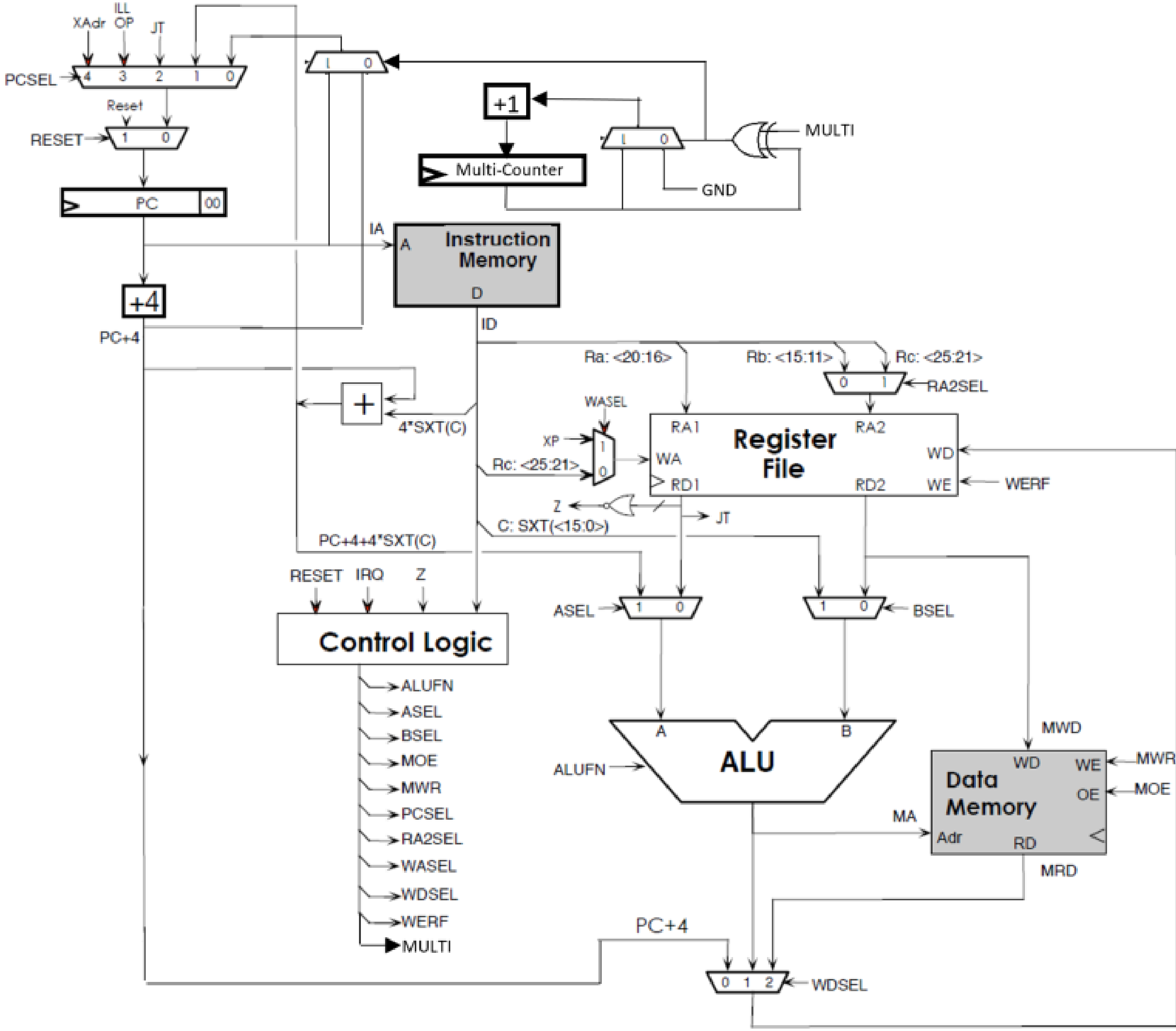
- RISC
- Iteration on 6.004 beta
    - More RISC commands
    - Complex commands
    - Performance optimizations
    - Hardware I/O

For our project we implemented a Reduced Instruction Set Computer (RISC) processor based on the MIT 6.004 Beta. We started by implementing all the basic modules outlined in the design. The modules we put together to compose our processor include the Arithmetic Logic Unit (ALU), that handles all the operations on register values, the Control Logic (CTL) that sets the current state of the processor with al; the wire values for the current instruction, the Register File that stores data in registers -- the fastest type of memory, the Instruction memory which stores the contains the read-only instructions that make up the programs the processor can run, the Program Counter that points to the current instruction being run, and the Data Memory that stores data which will be operated on by the current program.

We then added additional features on top of this basic implementation. We started by implementing more RISC commands like the MOV, MOD, and ZERO instructions, as well as adding complex commands (which take multiple clock cycles to complete) like PUSHA, that

push the first 8 register file values to memory in one instruction. Next we wanted a way to interact with the processor and demonstrate some programs, so we added input and output capabilities which allowed us to provide parameters to the test programs we wrote and view the resulting output on the Nexys board. Finally we did some performance optimization, mainly with the using fast RAM registers as our data memory since we did not have a large enough data set to justify using DDRAM. If the total memory used by the processor were to exceed the amount of fast block RAM provided by the Nexys (#instructions + #registers, #data_memory_words > 150,000), our modular design would allow easy transition to DDR-SDRAM.

Design



Arithmetic Logic Unit

(Bradley Jomard)

- Combinational logic
- I/O
- Operations Offered (plus selector constants)

The ALU performs basic logic computations on two 32-bit inputs. These operations are selected via a 6-bit selector input. The operations that are supported currently are equal, less than and less than or equal compare, add, subtract, multiply, divide, modulo, AND, OR, XOR, XNOR, shift left, shift right, shift right arithmetic, and the default outputs zero. The ALU performs the selected operation on two 32-bit registers that it gets as inputs from the register file, and outputs one value based on the operation. The 6-bit selectors for each function are :

ALUFN[5:0]	Operation	Output value Y[31:0]
000011	CMPEQ	Y = (A == B)
000101	CMPLT	Y = (A < B)
000111	CMPLE	Y = (A ≤ B)
010000	ADD	Y = A+B
010001	SUB	Y = A−B
101000	AND	Y[i] = A[i] · B[i]
101110	OR	Y[i] = A[i] + B[i]
100110	XOR	Y[i] = A[i] ⊕ B[i]
101001	XNOR	Y[i] = ~(A[i] ⊕ B[i])
101010	"A"	Y = A
110000	SHL	Y = A << B
110001	SHR	Y = A >> B
110011	SRA	Y = A >> B (sign extended)
100010	MUL	Y = A * B
100011	DIV	Y = A / B
100100	MOD	Y = A % B

Control Logic

(Quinn Magendanz)

- Combinational logic
- I/O
- Instructions offered

The control logic module uses combinational logic to assign values to constants that tell the register file, ALU and data memory when to read, write and what operations to do. The control logic module takes as input the operation that is going to be run, and then outputs all the constants listen below in the table

	RESET	IRQ	OP	OPC	LD	LDR	ST	JMP	BEQ	BNE	ILLOP	PUSHA
ALUFN[5:0]	--	--	F(op)	F(op)	"+"	"A"	"+"	--	--	--	--	"+"
ASEL	--	--	0	0	0	1	0	--	--	--	--	0
BSEL	--	--	0	1	1	--	1	--	--	--	--	1
MOE	--	--	--	--	1	1	0	--	--	--	--	0
MWR	0	0	0	0	0	0	1	0	0	0	0	1
PCSEL[2:0]	--	4	0	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	3	0
RA2SEL	--	--	0	--	--	--	1	--	--	--	--	1
WASEL	--	1	0	0	0	0	--	0	0	0	1	--
WDSEL[1:0]	--	0	1	1	2	2	--	0	0	0	0	--
WERF	--	1	1	1	1	1	0	1	1	1	1	0
MULTI	0	0	0	0	0	0	0	0	0	0	0	7

- ALUFN is the code for the function that the ALU needs to run, like add, substract, divide…
- ASEL is the selector for the first register inputted to the ALU
- BSEL is the selector for the second register inputted to the ALU
- MOE is 1 when we want to output the read data from memory
- MWR is 1 when we want to enable writing to memory
- PCSEL is the program counter selector
- RA2SEL is the second read address selector for the register file
- WASEL is the write address selector for the register file
- WDSEL selects what data is to be written into the register file
- WERF is 1 when writing to the register file is enabled

Register File

(Bradley Jomard)

- 32 32-bit
- Reads are combinational
- Writes are sequential

- I/O

The register file controls access to a set of thirty two 32-bit registers. Registers are being used for the data to be operated on by the ALU because they are the fastest type of memory available. Data reads are combinational using wires so that they are available as fast as possible, and writes are sequential, done in one clock cycle starting at the clock posedge. Register 26 always stores the program selector, a value that determines what program from the instruction memory is being run (currently we have fibonacci, sort, save, load). Registers 24 and 25 are wired to the switches on the Nexys to allow the user to input values for the different programs that can be run, like saving an inputted value to an inputted memory address in data memory, or finding the n number from fibonacci. Registers 0-7 are also displayed on the Nexys to allow us to see the results of the programs we run.

Instruction Memory

(Quinn Magendanz)

- Combinational ROM
- Memory type - fast block RAM
  - Modular design allows for easy switch to DDRRAM if > 150,000
  - Clock speed (using DDR-SDRAM reduced clock speed by a factor of 3 increase)
  - Can combine with data memory to get better utilization and writable instructions
- Assembly macros
  - Instructions
  - Instruction structure (32-bit aligned)
  - Instruction constants

Instruction memory is read-only. As a result, we make it combinational logic which does not need to wait for a clock rise to perform the read. It reads out the 32-bit instruction at the address specified by the program counter. The instruction memory currently uses fast block RAM (registers) to store instructions because it if the fastest type of memory given the limited amount of instructions, data, and registers used by the processor. If the sum of those types of memory were exceeded (#instructions + #registers, #data_memory_words > 150,000), the modular design would allow transition to DDR-SRAM.

```
// Instructions
`define LD(ra, lit, rc)      {`op_LD, rc, ra, lit}
`define ST(rc, lit, ra)      {`op_ST, rc, ra, lit}
`define PUSHA(ra)            {`op_PUSHA, 5'b0, ra, 16'b0}
`define JMP(ra, rc)          {`op_JMP, rc, ra, 16'b0}
`define BEQ(ra, lit, rc)     {`op_BEQ, rc, ra, lit}
`define BNE(ra, lit, rc)     {`op_BNE, rc, ra, lit}
`define LDR(lit, rc)         {`op_LDR, rc, 5'b0, lit}
`define ADD(ra, rb, rc)      {`op_ADD, rc, ra, rb, 11'b0}
`define SUB(ra, rb, rc)      {`op_SUB, rc, ra, rb, 11'b0}
`define MUL(ra, rb, rc)      {`op_MUL, rc, ra, rb, 11'b0}
`define DIV(ra, rb, rc)      {`op_DIV, rc, ra, rb, 11'b0}
`define MOD(ra, rb, rc)      {`op_MOD, rc, ra, rb, 11'b0}
`define CMPEQ(ra, rb, rc)    {`op_CMPEQ, rc, ra, rb, 11'b0}
`define CMPLT(ra, rb, rc)    {`op_CMPLT, rc, ra, rb, 11'b0}
`define CMPLE(ra, rb, rc)    {`op_CMPLE, rc, ra, rb, 11'b0}
`define AND(ra, rb, rc)      {`op_AND, rc, ra, rb, 11'b0}
`define OR(ra, rb, rc)       {`op_OR, rc, ra, rb, 11'b0}
`define XOR(ra, rb, rc)      {`op_XOR, rc, ra, rb, 11'b0}
`define ZERO(ra)             {`op_XOR, ra, ra, ra, 11'b0}
`define XNOR(ra, rb, rc)     {`op_XNOR, rc, ra, rb, 11'b0}
`define SHL(ra, rb, rc)      {`op_SHL, rc, ra, rb, 11'b0}
`define SHR(ra, rb, rc)      {`op_SHR, rc, ra, rb, 11'b0}
`define ADDC(ra, lit, rc)    {`op_ADDC, rc, ra, lit}
`define MOV(ra, rc)          {`op_ADDC, rc, ra, 16'd0}
`define MOVC(lit, rc)        {`op_ADDC, rc, 5'd31, lit}
`define SUBC(ra, lit, rc)    {`op_SUBC, rc, ra, lit}
`define MULC(ra, lit, rc)    {`op_MULC, rc, ra, lit}
`define DIVC(ra, lit, rc)    {`op_DIVC, rc, ra, lit}
`define MODC(ra, lit, rc)    {`op_MODC, rc, ra, lit}
`define CMPEQC(ra, lit, rc) {`op_CMPEQC, rc, ra, lit}
`define CMPLTC(ra, lit, rc) {`op_CMPLTC, rc, ra, lit}
`define CMPLEC(ra, lit, rc) {`op_CMPLEC, rc, ra, lit}
`define ANDC(ra, lit, rc)    {`op_ANDC, rc, ra, lit}
`define ORC(ra, lit, rc)     {`op_ORC, rc, ra, lit}
`define XORC(ra, lit, rc)    {`op_XORC, rc, ra, lit}
`define XNORC(ra, lit, rc)   {`op_XNORC, rc, ra, lit}
`define SHLC(ra, lit, rc)    {`op_SHLC, rc, ra, lit}
`define SHRC(ra, lit, rc)    {`op_SHRC, rc, ra, lit}
`define SHRAC(ra, lit, rc)   {`op_SRAC, rc, ra, lit}
```

Program Counter

(Quinn Magendanz)

- Sequential logic
- Selection of next instruction

- Multi-counter

The heart of the PC module is the 32-bit register that holds the current value of the program counter, which is the address in main memory of the instruction to be executed in the current clock cycle.

The 5-input 32-bit PCSEL multiplexer selects the value to be loaded into the PC register at next rising edge of the clock.

The PC+4 adder simply adds 4 to the output of the PC register, computing the address of the instruction following the current instruction. The output of this adder forms the PC_INC output of the PC module.

The branch-offset adder is responsible for computing the address of instruction specified by the literal field in BEQ and BNE instructions. That address is (PC+4)+4*SXT(ID[15:0]), i.e., multiplying the sign-extended 16-bit literal field of the instruction by 4 to convert the word offset into a branch offset, then adding that to the address of the next instruction. Specifically:

- PCSEL=0. This input is selected during "normal" execution when the next instruction to be executed is the one after the current instruction.
- PCSEL=1. This input is selected when a branch instruction is "taken".
- PCSEL=2. This input is selected during a JMP instruction, when we use the contents of the register selected by the RA field of the instruction as the address of the next instruction.
- PCSEL=3. This input is selected when the current instruction has an illegal opcode, so we want to set the next PC to 0x80000004. It's easy to create a 32-bit-wide wire with the appropriate constant value: In this case we want the label 0x80000004'32.
- PCSEL=4. This input is selected when the Beta is taking an interrupt, so we want to set the next PC to 0x80000008.

We added an additional register to this module to provide support for instructions which take multiple clock cycles to complete. The Multi-Counter register increments itself every clock cycle in order to track how far along complex instructions are. When the complex instruction completes, its multi-counter will equal the multi value specified by the control module, and the program counter will then move to the next instruction.

Data Memory

(Bradley Jomard)

- Reads are combinational
- Writes are sequential
- Memory type - fast block RAM
  - Modular design allows for easy switch to DDRRAM if > 150,000

The register file controls access to a set of 128 32-bit registers. Since the registers are 32-bit, hence 4 bytes each, the indexes for the memory addresses should be multiples of 4, so that values saved don't overlap on each other. Originally the goal was to use DRAM for the data memory as opposed to registers. However, we looked at the specifications for the Nexys and saw that it had enough memory for around 150,000 32-bit instructions, data memory and register file registers, which was more than enough for us considering the small size of the programs we are running. However the modular design would still allow us to easily switch to DRAM if we ever needed more memory. Data reads are combinational using wires so that they are available as fast as possible, and writes are sequential, done in one clock cycle starting at the clock posedge, similarly to the register file. The Save and Load programs allow the user to save an inputted value into memory and load a value from memory into the register file. The Sort program sorts the values from the memory addresses 0-7.

Testing

This section contains a description of each of the programs loaded into the instruction ROM as well as the assembly code describing the exact execution of the test program and the simulation graph showing the values of processor wires throughout its execution.

I/O

(Quinn Magendanz)

In order to directly interact with the processor, we hard-wired the first eight switches to register 24 and the next seven switches to register 25. Register 25 was used in our testing programs to

indicate a memory address that a program was to operate at. Register 24 specified the input value to that program.

The lower four bits of the first eight registers were also hard-wired to the display panel on the Nexys board. This allowed us to load the output of the currently running function into the first eight registers to view the output.

Program Selector

(Quinn Magendanz)

The program selector allowed us to specify a program to run given an input from the Nexys board. The program selector was hard-wired to register 26. Given the current programs in ROM, the processor will spin at instruction zero until this register is changed. Once a program is specified, the program selector will jump to the address of that program.

```
// Program Selector
// Takes in literals of the addresses of four other programs to jump to.
// This program should be located at address 0.
// r26 is the register linked to input specifying which function to jump to.
// r0 copies r26 and is used to check which function to jump to.
// r1 is used to store this address to jump.

`d(0) `BEQ(5'd26, -16'd1, 5'd31);    // Remain at first instruction until r0 != 0
`d(1) `MOV(5'd26, 5'd0);
`d(2) `SUBC(5'd0, 16'd1, 5'd0);    // Jump to program specified in r0.
`d(3) `BNE(5'd0, 16'd2, 5'd31);
`d(4) `MOVC(a1, 5'd1);
`d(5) `JMP(5'd1, 5'd31);
`d(6) `SUBC(5'd0, 16'd1, 5'd0);
`d(7) `BNE(5'd0, 16'd2, 5'd31);
`d(8) `MOVC(a2, 5'd1);
`d(9) `JMP(5'd1, 5'd31);
```

```
`d(10) `SUBC(5'd0, 16'd1, 5'd0);
`d(11) `BNE(5'd0, 16'd2, 5'd31);
`d(12) `MOVC(a3, 5'd1);
`d(13) `JMP(5'd1, 5'd31);
`d(14) `SUBC(5'd0, 16'd1, 5'd0);
`d(15) `BNE(5'd0, 16'd2, 5'd31);
`d(16) `MOVC(a4, 5'd1);
`d(17) `JMP(5'd1, 5'd31);
`d(18) `SUBC(5'd0, 16'd1, 5'd0);
`d(19) `BNE(5'd0, 16'd2, 5'd31);
`d(20) `MOVC(a5, 5'd1);
`d(21) `JMP(5'd1, 5'd31);
`d(22) `JMP(5'd31, 5'd31);        // Should never reach here.
```

Fibonacci

(Quinn Magendanz)

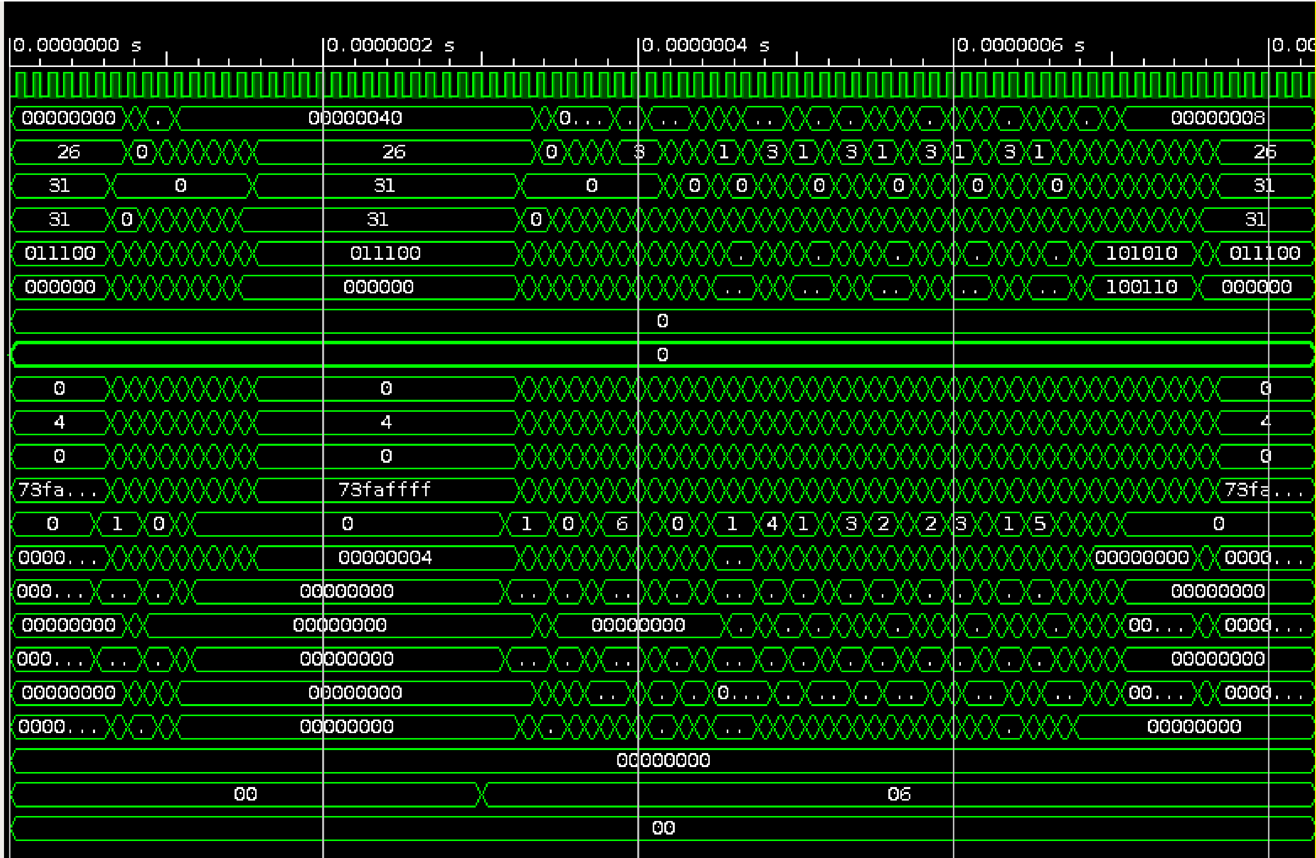
Fibonacci is a test program which takes in a number of the fibonacci sequence which we need to calculate through register 24 and returns that number of the sequence in register 0.

```
// Fibonacci
// Generates the n'th fibonacci number.
// r3 stores the current n fibbonacci number.
// r0 stores the value of the current fibbonacci number.
// r1 stores n-1.
// r2 stores n-2.

`f(0) `MOV(5'd24, 5'd3);
`f(1) `BNE(5'd3, 16'd2, 5'd31);    // If n == 0
`f(2) `ZERO(5'd0);
`f(3) `JMP(5'd31, 5'd31);
```

```
`f(4) `SUBC(5'd3, 16'd1, 5'd3);
`f(5) `BNE(5'd3, 16'd2, 5'd31);    // If n ==1
`f(6) `MOVC(15'd1, 5'd0);
`f(7) `JMP(5'd31, 5'd31);

`f(8) `ZERO(5'd2);                  // Init r2 = 0
`f(9) `MOVC(16'd1, 5'd1);    // Init r1 = 1
`f(10) `SUBC(5'd3, 16'd1, 5'd3);    // Loop through
`f(11) `ADD(5'd1, 5'd2, 5'd0);
`f(12) `MOV(5'd1, 5'd2);
`f(13) `MOV(5'd0, 5'd1);
`f(14) `BNE(5'd3, -16'd5, 5'd31);    // End loop
`f(15) `ZERO(5'd1);
`f(16) `ZERO(5'd2);
`f(17) `ZERO(5'd3);
`f(18) `ZERO(5'd4);
`f(19) `ZERO(5'd5);
`f(20) `ZERO(5'd6);
`f(21) `ZERO(5'd7);
`f(22) `JMP(5'd31, 5'd31);
```


Name
Value
clock 1
reg_data[31:0] 00000008
ra[5:0] 26
rb[5:0] 31
rc[5:0] 31
op[5:0] 011100
alufn[5:0] 000000
multi[4:0] 0
multi_counter[4:0] 0
pc[31:0] 0
pc_inc[31:0] 4
pc_offset[31:0] 0
id[31:0] 73faffff
jt[31:0] 0
wdata[31:0] 00000004
radata[31:0] 00000000
rbdata[31:0] 00000000
a[31:0] 00000000
b[31:0] 00000000
alu_out[31:0] 00000000
first_eight[31:0] 00000000
input1[7:0] 06
input2[6:0] 00
0.0000000 s
0.0000002 s
0.0000004 s
0.0000006 s

Sort

(Quinn Magendanz)

Sort is a test program which takes the first eight words in memory and sorts then with the lowest address being the smallest value and the largest address being the largest value. Sort than takes these values and loads them into the first eight registers so that the sorted values can be viewed on the board.

```
// Sort
// Sort the values stored in the first 8 memory addresses..
// r0 - c
// r1 - d
// r2 - d-1
// r3 - arr[d]
// r4 - arr[d-1]
// r5 - comparison bit

`s(0) `MOVC(16'd4, 5'd0);  // for (c = 1

`s(1) `MOV(5'd0, 5'd1);            // d = c

`s(2) `SUBC(5'd1, 16'd4, 5'd2);  // d - 1
`s(3) `LD(5'd1, 16'd0, 5'd3);          // arr[d]
`s(4) `LD(5'd2, 16'd0, 5'd4);          // arr[d-1]
`s(5) `CMPLT(5'd31, 5'd1, 5'd5);    // d > 0
`s(6) `CMPLT(5'd3, 5'd4, 5'd6);      // arr[d-1] > arr[d]
`s(7) `AND(5'd5, 5'd6, 5'd5);          // &&
`s(8) `BEQ(5'd5, 16'd4, 5'd31);      // while

`s(9) `ST(5'd4, 16'd0, 5'd1);          // arr[d] = arr[d-1]
`s(10) `ST(5'd3, 16'd0, 5'd2);        // arr[d-1] = arr[d]
```

```
`s(11) `SUBC(5'd1, 16'd4, 5'd1);    // d = d - 1;
`s(12) `BEQ(5'd31, -16'd11, 5'd31);           // end loop

`s(13) `ADDC(5'd0, 16'd4, 5'd0);    // c++
`s(14) `SUBC(5'd0, 16'd32, 5'd5);  // c < n
`s(15) `BNE(5'd5, -16'd15, 5'd31); // end loop

`s(16) `LD(5'd31, 16'd0, 5'd0);
`s(17) `LD(5'd31, 16'd4, 5'd1);
`s(18) `LD(5'd31, 16'd8, 5'd2);
`s(19) `LD(5'd31, 16'd12, 5'd3);
`s(20) `LD(5'd31, 16'd16, 5'd4);
`s(21) `LD(5'd31, 16'd20, 5'd5);
`s(22) `LD(5'd31, 16'd24, 5'd6);
`s(23) `LD(5'd31, 16'd28, 5'd7);

`s(24) `JMP(5'd31, 5'd31);
```

0.000001995000 s

Name	Value
clock	1
reg_data[31:0]	00000000
ra[5:0]	26
rb[5:0]	31
rc[5:0]	31
op[5:0]	011100
alufn[5:0]	000000
multi[4:0]	00
multi_counter[4:0]	00
pc[31:0]	0
pc_inc[31:0]	4
pc_offset[31:0]	0
id[31:0]	73faffff
jt[31:0]	0
wdata[31:0]	00000004
radata[31:0]	00000000
rbdata[31:0]	00000000
a[31:0]	00000000
b[31:0]	00000000
alu_out[31:0]	00000000
first_eight[31:0]	23456789
input1[7:0]	02
input2[6:0]	1c

0.0000000 s | 0.0000005 s | 0.0000010 s | 0.0000015 s | 0.0000020 s | 0.0000025 s | 0.0000030 s | 0.0000035 s | 0.0000040 s | 0.0000045 s | 0.0000050 s | 0.0000055 s | 0.0000060 s

Save and Load

(Bradley Jomard)

Save is a simple program that allows the user to save a value into a specific address in data memory and zero's the first 8 registers from the register file. The program saves the value from register 24, that is wired to the first 8 switches on the Nexys (switches 0-7), and stores it at the memory address from register 25, which is wired to the next 8 switches from the Nexys (switches 8-14). This means that the memory addresses have to be mutliples of 4 since the values are saved as 32-bits, hence 4 bytes.

Load is a simple program that allows the user to load a value from data memory into the first register of the register file and zeros registers 1-7 of the register file. The program loads the value that is saved at the memory address taken from register 25, that is wired to switches 8-14 on the Nexys.

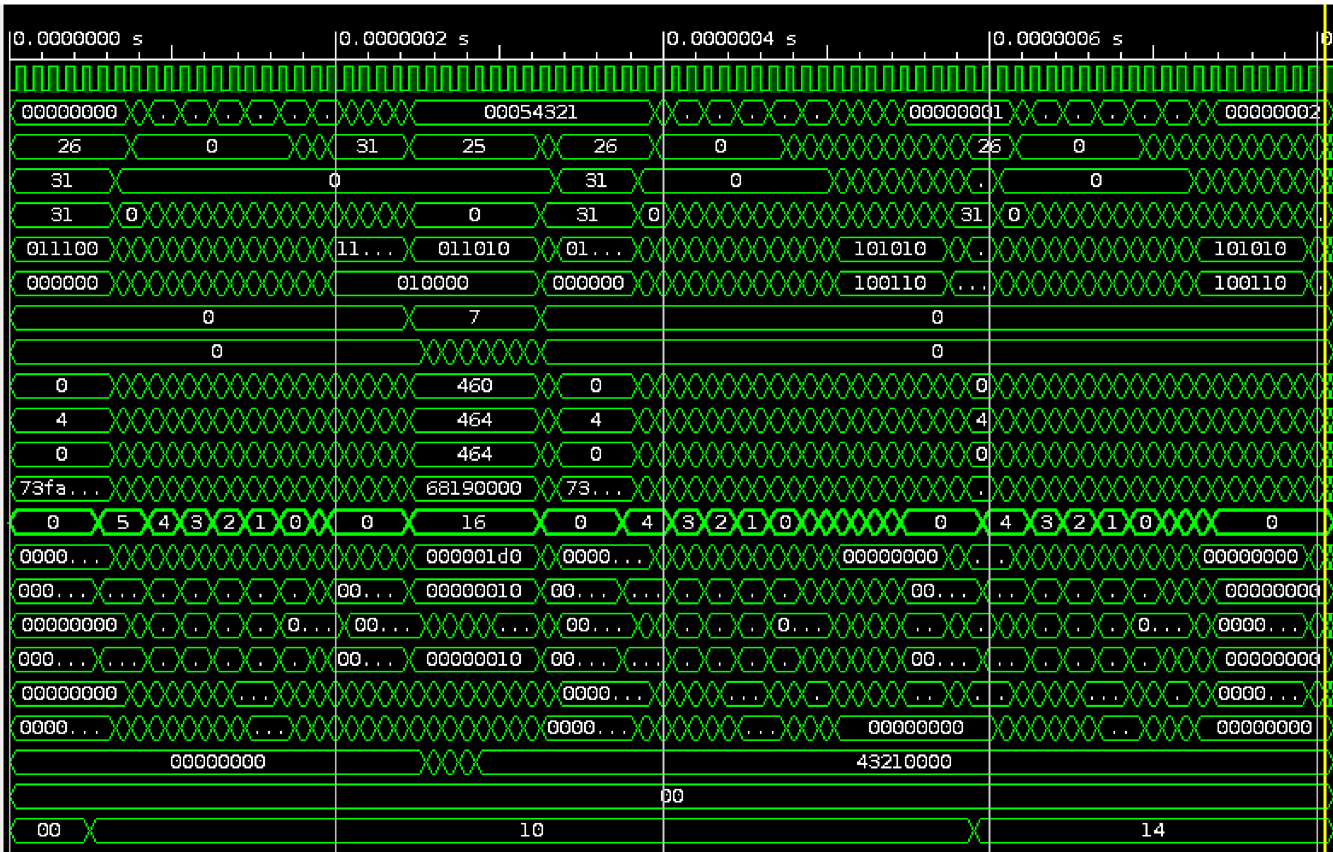
The following simulation was used to test the save and load programs simultaneously. The simulation first sets the values for registers 24 and 25, then runs the save program to save the value into memory and then runs the load program to load it into the first register from the register file.

```
//	// Save and load test
//	input1 = 3;
//	input2 = 4;
//	show_output = 1;
//	#20 run_save = 1;
//	#20 run_save = 0;
//	#200 run_load = 1;
//	#20 run_load = 0;
//	#200 input1 = 6;
//	input2 = 8;
//	run_save = 1;
//	#20 run_save = 0;
//	#200 run_load = 1;
```

```
//	#20 run_load = 0;
//	#200 input2 = 4;
//	run_load = 1;
//	#20 run_load = 0;
//	#500;
```

Pusha

(Quinn Magendanz)



Name
Value
clock 1
reg_data[31:0] 00000002
ra[5:0] 26
rb[5:0] 31
rc[5:0] 31
op[5:0] 011100
alufn[5:0] 000000
multi[4:0] 0
multi_counter[4:0] 0
pc[31:0] 0
pc_inc[31:0] 4
pc_offset[31:0] 0
id[31:0] 73faffff
jt[31:0] 0
wdata[31:0] 00000004
radata[31:0] 00000000
rbdata[31:0] 00000000
a[31:0] 00000000
b[31:0] 00000000
alu_out[31:0] 00000000
first_eight[31:0] 43210000
input1[7:0] 00
input2[6:0] 14
0.0000000 s
0.0000002 s
0.0000004 s
0.0000006 s