EENG 284 – Digital Design

Lab 11 – Complete Stored Program Computer

Part 4 – Control, Datapath, RAM with load and store

# Objective

The objective of this lab is to add enhanced capabilities to our stored program computer – a load and store instructions. The addition of these two instructions makes our simple little computer, in theory, capable of executing any algorithm that can be written. It just might do it a little slower than your laptop.

**Today’s Computer**

In the last lab you added a control unit to automate the fetching and executing of R-type, I-type AND branch instructions. This week, you will incorporate two new instructions, load and store. The load instruction moves a word from the RAM into one of the registers from the register file. The store instruction moves one of the registers from the register file into the RAM. The addition of this pair of instructions will require three new registers in the datapath; the memory address register (MAR), the memory buffer register in (MBRin) and the memory buffer register out (MBRout) shown in Figure 1. This additional complexity will allow you to write programs that can read and write arrays.

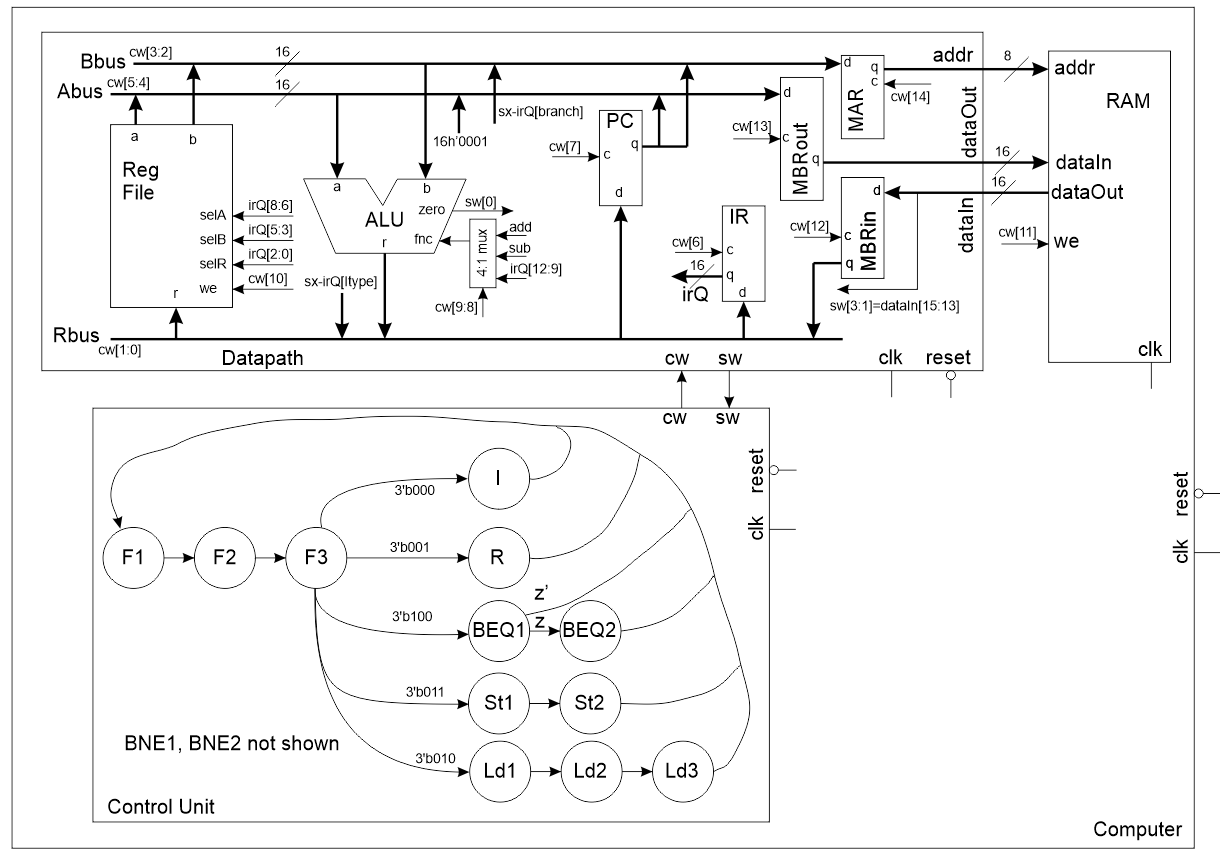


Figure : Incorporating load and store instructions requires new registers between the datapath and the RAM.

Let’s start our work on this lab by looking at the two new instructions that you are going to incorporate into your computer, load and store.

**The Instruction Set**

From the previous three labs you should be familiar with the 16-bit I-type, R-type and branch instruction shown in the top four rows of Table 1. The last two rows of this table introduce the binary format of the two new instructions, load (LD) and store (ST). These two instructions do not need all 16-bit available. I will assign these unused bits, labeled “unused” in Table 1, the bit value “0” for no particular reason – you can treat them as don’t cares.

Table : The instruction format for all the instruction types executed by our computer.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| I | 0 | 0 | 0 | x | x | Immediate | | | | | | | | dst | | |
| R | 0 | 0 | 1 | fnc | | | | selA | | | selB | | | dst | | |
| BEQ | 1 | 0 | 0 | Offset MS | | | | sel A | | | sel B | | | Offset – LS | | |
| BNE | 1 | 0 | 1 | Offset – MS | | | | sel A | | | sel B | | | Offset – LS | | |
| LD | 0 | 1 | 0 | unused | | | | src | | | addr | | | unused | | |
| ST | 0 | 1 | 1 | unused | | | | | | | addr | | | dst | | |

**The load (LD) and store (ST) instructions**

The load (LD) instruction moves one of the registers with index given by bits 8:6 of the instruction register into the memory address contained in the register with index given by bits 5:3 of the instruction register. Whenever you move data from the RAM into the CPU you are performing a read operation because this process is analogous to you (the CPU) reading information from a book (the RAM). You can think of the page number as the address in the RAM that you are reading from. This analogy breaks down when we realize that each page of a book contains more than the 16-bits found at one RAM address. When you write an assembly language command for load you put the destination register first and the register which contains the address second. Thus, the assembly language command, LD R2, M[R1] loads the RAM value contained at the address in register R1 into register R2. The “M” in the assembly instruction is a reference to “Memory” and is put there to help you remember that R1 is an address in memory. For example, if R1 contained the value 20, this assembly instruction would move the memory word stored at address 20 into register R1. The load instruction does not care what is stored at address 20 (it could be an instruction or data), it just performs the load.

The store (ST) instruction moves the memory word at the memory address contained in the register with index given by bits 5:3 of the instruction register into the register with index given by bits 2:0 of the instruction register. Whenever you move data from the CPU to the RAM into the CPU you are performing a write operation because this process is analogous to you (the CPU) writing information into a notebook (the RAM). You can think of the page number as the address in the RAM that you are writing to. Like the previous analogy, this one breaks down when you realize that each page in your notebook contains more than the 16-bits found at one RAM address. When you write an assembly language command for store you put the source register first and the register which contains the address second. Thus, the assembly language command, ST R2, M[R1] stores the contents of R2 at the RAM address contained in register R1. For example, if R1 contained the value 20, this assembly instruction would move the memory word stored the value contained in register R2 to address 20. The load instruction does not care what is stored at address 20 (it could be an instruction or data), it overwrites the old value with the new contents.

Let’s work an example to get a better understanding how load and store instructions operate, execute the program given in Listing 1 to update the value in Table 2.

Listing : Sample program that uses load and store instructions

1. LDI #1, R7

2. LDI #24, R1 // Load register 1 with 24

3. LD R2, M[R1] // Load R2 with value at RAM address in R1

4. LD R3, M[R2]

5. LDI #20, R1

6. ST R3, M[R1] // Store R3 at RAM address in R1

7. ADD R7, R1, R1 // Increment value in R1

8. ST R2, M[R1]

9. ADD R7, R1, R1

10. ST R1, M[R1]

To complete Table 2, go through execute each line of code in Listing 1. Each line will update either a register (R1, R2, R3 or R7) or a memory value. If an instruction on line x changes a register y’s value, put the new register value in column y of line x. If an instruction changes the RAM contents, then just write in the new value in the “RAM contents” column.

Table : Update the register and RAM values using the code in Listing 1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Line | R1 | R2 | R3 | R7 |  | RAM contents | Address |
| 1 |  |  |  | 1 |  | 55 | 20 |
| 2 | 24 |  |  |  |  | 23 | 21 |
| 3 |  | 23 |  |  |  | 22 | 22 |
| 4 |  |  | 55 |  |  | 55 | 23 |
| 5 | 20 |  |  |  |  | 23 | 24 |
| 6 |  |  |  |  |
| 7 | 21 |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |
| 9 | 22 |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |

**The Control Unit**

The state diagram for our control unit is shown in Figure 1. The status word input consists of the same 4-bit from last week; the upper 3 are the instruction opcode (upper 3 bits of the instruction register) and the least significant bit is the zero output from the ALU. The instruction opcode bits, sw[3:1] are listed by their 3-bit codes out of the state F3.

The design of the control unit will follow the steps outlined in the traffic light controller handout given in class. I would advise that you use the code from last week’s control unit as the starting point for this weeks lab.

**State definitions**

Use a dense coding to assign each state in a unique binary code (the choice of which code is assigned to each state is immaterial) using the localparam statement. Since there are more than 8 states, the state code will be 4-bit long. If you made alias for the op-codes, you should add the op-codes for the load and store instructions. The operation of the new states will be dicsussed in the next section, so for the time being just make dummy values for the control words assocaited with each state. Just note that the control word is 15 bits.

localparam F2\_STATE = 4'b0001;

localparam RTYPE\_OPCODE = 3'b001;

localparam FETCH\_CW = 15'b00000000001;

Even though BNE1 and BNE2 are not shown in Figure 1, they need to be incorporated into the control unit. I just didn’t have room to include them.

**Define the reset state**

In all our testbenches, we will briefly hold the reset line low when our simulation first starts. This reset should put your control unit in the **F1** state.

**Next state logic**

When you are including the next state logic, make sure to go back and update the names that may have changed (like F1).

**Output logic**

This section of logic is pretty straight forward, have a row for each state in the always/case statement. Just make sure to have one row for each state.

**Testbench**

There is no testbench for the control unit in this week’s lab.

**The Datapath**

There are two significant changes incorporated into the datapath shown in Figure 2 to accommodate the introduction of load and store instructions; the MAR, MBRin and MBRout registers. The memory address register (MAR) will hold the address being sent to the datapath. All memory references, either to instructions or data must pass through MAR. Thus, the PC must now be loaded into the MAR before an instruction is fetched. There are a pair of memory buffer registers (MBRs), one for data leaving the datapath, MBRout, and one for data coming into the datapath, MBRin. Thus the instruction must pass through the MBRin before its loaded into the instruction register.

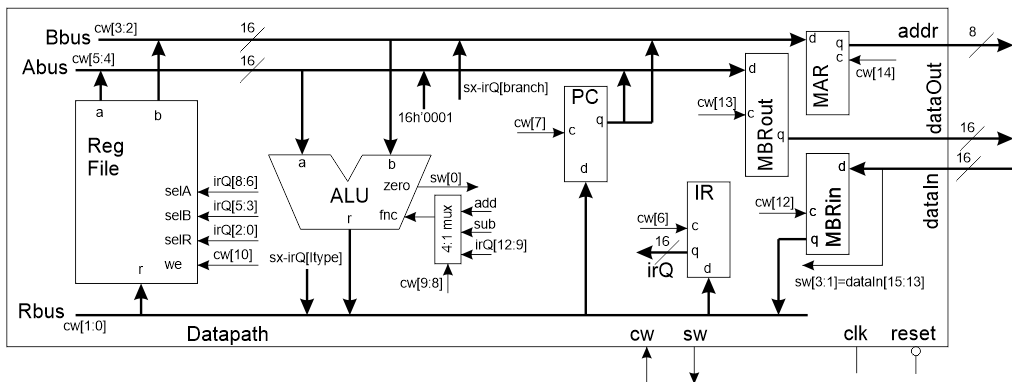


Figure : The datapath for this week's lab is able to execute I-type, R-type and branch instructions.

To better understand these changes, how to implement them, and how they are used, let’s examine the control word shown in Table 3 for the datapath shown in Figure 2. The top row of the control word table is the bit index of the signal. Devices that require 1-bit of control have a single index and are control with that bit of the control word. Devices with two bits of control are given two indices and their control word is formed from the sub-vector of these two indices.

You job will be to fill in the control word bits for each state according to the needs of that state. To help you, the following list explains what each state needs to accomplish.

Fetch 1: Copy the PC into the MAR (over the B bus) and increment the PC

Fetch 2: Read the instruction located at the address in the MAR and store into the MBRin

Fetch 3: Copy the MBRin into the IR

I type: Store the sign extended immediate to the RD file register

R: Send RS1 and RS2 to the ALU, perform ALU operation specified by IR bits and store

the ALU result into RD. Stores the value on the R bus to the RD file register

BEQ1, BNE1: Subtract RS1 and RS2 setting the zero bit if the two are equal

BEQ1, BEQ2: Add PC and sign extended IR and store result back into the PC

L1: Copy the register at index IR[5:3] to the MAR

L2: Read the data located at the address in the MAR and store into the MBRin

L3: Copy MBRin into the register at index IR[8:6]

S1: Copy reg. at index IR[5:3] to the MAR, copy reg. at index IR[2:0] to MBRout

S2: Write value in MBRout to address in MAR

Now complete the control word table given at the bottom of Table 3.

Table : The control words for all the states of the control unit for the datapath shown in Figure 2.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 14 | 13 | 12 | 11 | 10 | 9,8 | 7 | 6 | 5,4 | 3,2 | 1,0 |
| Code | MAR | MBRout | MBRin | RAM WE | RF WE | ALU | PC | IR | ABUS | BBUS | RBUS |
| 00 or 0 | hold | hold | hold | no write | No | IR[12:9] | Hold | Hold | RFa | RFb | MBR |
| 1 |  |  |  |  |  |  |  |  | 0x01 |  |  |
| 10 |  |  |  |  |  | Sub |  |  |  | IR[BR] | IR[IMM] |
| 11 or 1 | load | load | load | write | Write | Add | Load | Load | PC | PC | ALU |
| FETCH1 | 1 | 0 | 0 | 0 | 0 | 11 | 1 | 0 | 01 | 11 | 11 |
| FETCH2 | 0 | 0 | 1 | 0 | 0 | 00 | 0 | 0 | 00 | 00 | 00 |
| FETCH3 | 0 | 0 | 0 | 0 | 0 | 00 | 0 | 1 | 00 | 00 | 00 |
| ALU\_OP | 0 | 0 | 0 | 0 | 1 | 00 | 0 | 0 | 00 | 00 | 11 |
| LOAD\_IMM | 0 | 0 | 0 | 0 | 1 | 00 | 0 | 0 | 00 | 00 | 10 |
| BEQ1 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 00 | 00 | 00 |
| BEQ2 | 0 | 0 | 0 | 0 | 0 | 11 | 1 | 0 | 11 | 10 | 11 |
| LOAD1 | 1 | 0 | 0 | 0 | 0 | 00 | 0 | 0 | 00 | 00 | 00 |
| LOAD2 | 0 | 0 | 1 | 0 | 0 | 00 | 0 | 0 | 00 | 00 | 00 |
| LOAD3 | 0 | 0 | 0 | 0 | 1 | 00 | 0 | 0 | 00 | 00 | 00 |
| STORE1 | 1 | 1 | 0 | 0 | 0 | 00 | 0 | 0 | 00 | 00 | 00 |
| STORE2 | 0 | 0 | 0 | 1 | 0 | 00 | 0 | 0 | 00 | 00 | 00 |

**The Computer**

The computer module contains three lines of Verilog code, one each to instantiate the datapath, control unit and RAM. Listing 2 shows the module declaration and internal signs for the computer. It’s remarkably simple, just send in a clock and reset signal and the datapath and control will execute the instructions stored in the RAM.

Listing : The module declaration of the computer only has a clock and reset signal.

module computer(clk, reset);

input wire clk, reset;

wire [14:0] cw;

wire [15:0] fromDPtoRAM;

wire [15:0] fromRAMtoDP;

wire [7:0] address;

wire [3:0] sw;

You could easily and quickly modify the top-level module from the previous assignment to finish this module.

**Test Program**

The test program for the computer is shown in Listing 3. This program computes pseudo-random numbers using an 8-bit linear feedback shift-register (LFSR), similar to the algorithm you used in Lab 4. You may remember that that algorithm computed a 7-bit LFSR by XORing bits 0, 4 and 6 of the current random number. Then the current random number was shifted left and the XOR bit was inserted into the LSB.

Listing : A linear feedback shift register program to generate 8-bit pseudo-random numbers and store them in an array.

0 LDI #0, R0

1 LDI #0x01, R2

2 LDI #0x20, R7

3 LDI #1, R1

4 main ST R2, M[R7]

5 ADD R1, R7, R7

6 LDI 0b01110001, R3

7 AND R2, R3, R4

8 LDI #7, R5

9 LDI #0, R6

10 loop AND R1, R4, R3

11 ADD R3, R6, R6

12 SHR R4, R4

13 SUB R5, R1, R5

14 BNE R5, R0, loop (-5)

15 SHR R2, R2

16 AND R1, R6, R6

17 BEQ R6, R0, main (-14)

18 LDI #0b0100 0000, R3

19 SHL R3, R3

20 OR R3, R2, R2

21 BEQ R0, R0, main (-18)

If the algorithm in Listing 3 was a 7-bit LFSR then it would calculate the XOR by counting the number of “1” bits in position 0, 4 and 6 in the current random number (stored in R2). The least significant bit of the number of 1’s is the same as the XOR. To verify that this works for 3-bits, complete Table 4.

* In the “XOR” column, compute the XOR of Bit 0, Bit 4 and Bit 6
* In the “count” column put the decimal count of the number of 1’s in Bit 0, Bit 4 and Bit 6
* In the “count binary” column put the decimal count of the number of 1’s in Bit 0, Bit 4 and Bit 6
* In the “LSB count” put the least significant bit from the corresponding “count binary”

Table : Computing the XOR of a collection of bits is the same as the LSB of the count of the number of 1's in that collection.

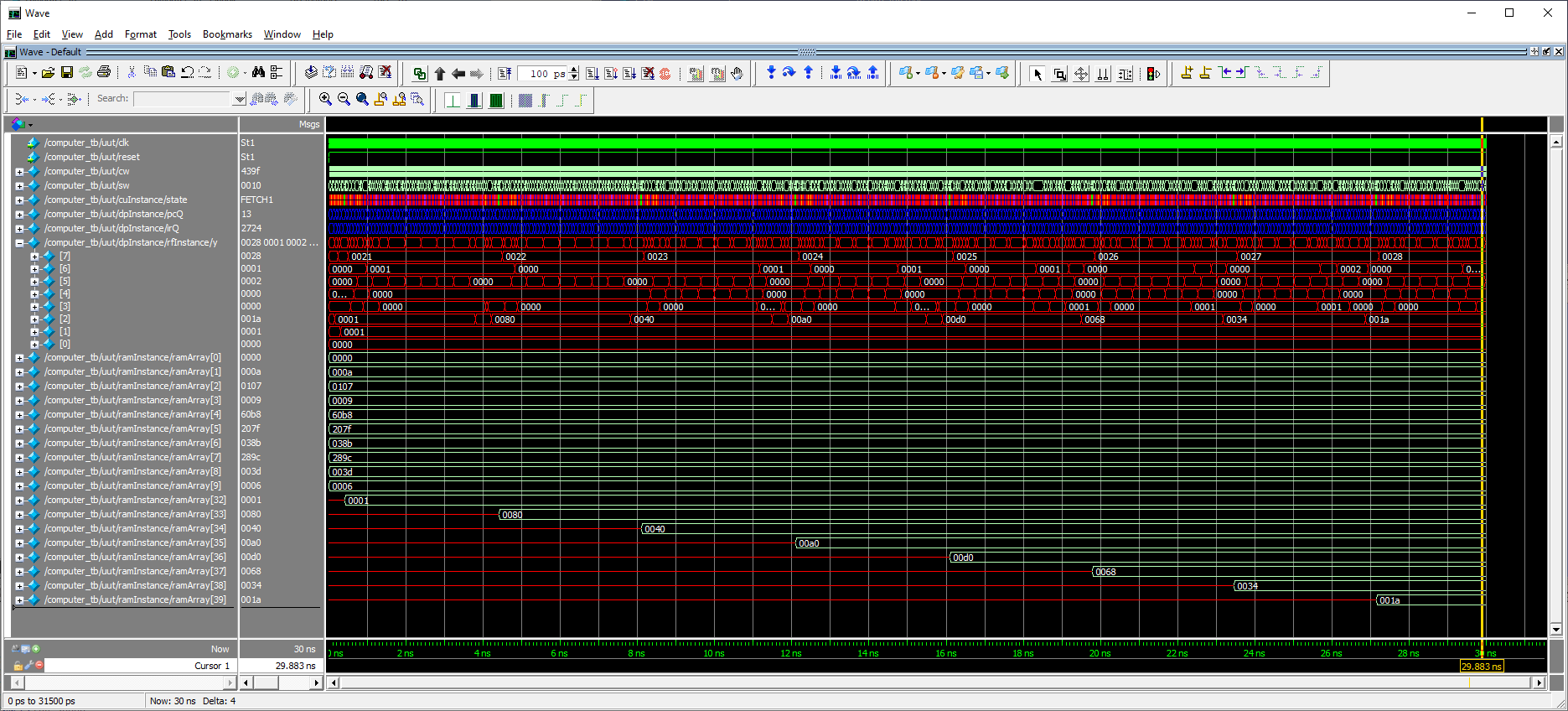
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Bit 0 | Bit 4 | Bit 6 | XOR | count | count binary | LSB count |
| 0 | 0 | 0 | 0 | 0 | 00 | 0 |
| 0 | 0 | 1 | 1 | 1 | 01 | 1 |
| 0 | 1 | 0 | 1 | 1 | 01 | 1 |
| 0 | 1 | 1 | 0 | 2 | 10 | 0 |
| 1 | 0 | 0 | 1 | 1 | 01 | 1 |
| 1 | 0 | 1 | 0 | 2 | 10 | 0 |
| 1 | 1 | 0 | 0 | 2 | 10 | 0 |
| 1 | 1 | 1 | 1 | 3 | 11 | 1 |

This counting process is performed by the loop on lines 10 through 15. Since our algorithm is a 8-bit LFSR it must compute the XOR of bits 0, 4, 5, 6. Register R3 has 1’s in the bits positions that need to be checked. R3 is then ANDed with the current random number and stored in R4. It is R4 that has its 1’s counted in the loop on line 10 through 15.

You will need to convert this algorithm to machine code using the instruction format shown in Table 1. I would suggest using the excel file posted on the class web page to help facilitate the conversion process.

**Testbench**

The testbench for the computer should generate 8-bit LFSR values and store them at incremental address starting at address 32. The following image shows the output after 30ns, this means simulating the computer for 30,000 time steps – there is a lot going on there!



**Deliverables**

**The Instruction set**

* Complete Table 2.

**The Control Unit**

* Verilog code for the body of the control unit.

**The Datapath**

* Verilog code for the body of the datapath.
* Complete Table 3.

**Computer**

* Complete machine code for the program in Listing 3.
* Complete Table 4.
* Complete testbench simulation for computer running program in Listing 3.