CprE 381 - Project Part B

Group 9: Arjay Vander Velden, Brian Reber, Scott Connell

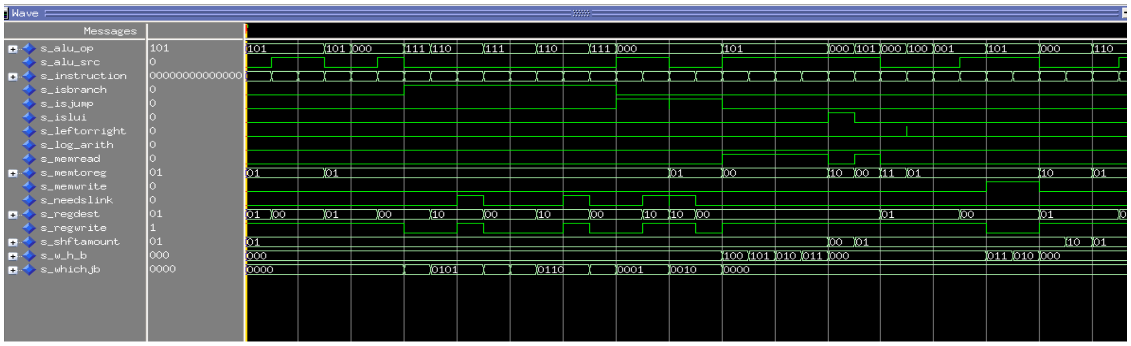
0) Prelab

This is shown in the file omniscient spreadsheet. This was verified by our TA.

1) Control Logic

(a) We created this spreadsheet and it is attached. See Omniscient Spreadsheet.

(b) To create the control logic module, we used with-select-then statements using the first 6 bits of the byte code; every time that the first 6 bits were identical, an intermediate signal would be set using another with-select-then statement using bits from another area of the byte code. We created a test bench to test the control logic module. The test bench confirmed that our control logic module correctly outputs the signals based on our Omniscient Spreadsheet. Shown below is a waveform from this test bench, showing a small sample of the correct results:

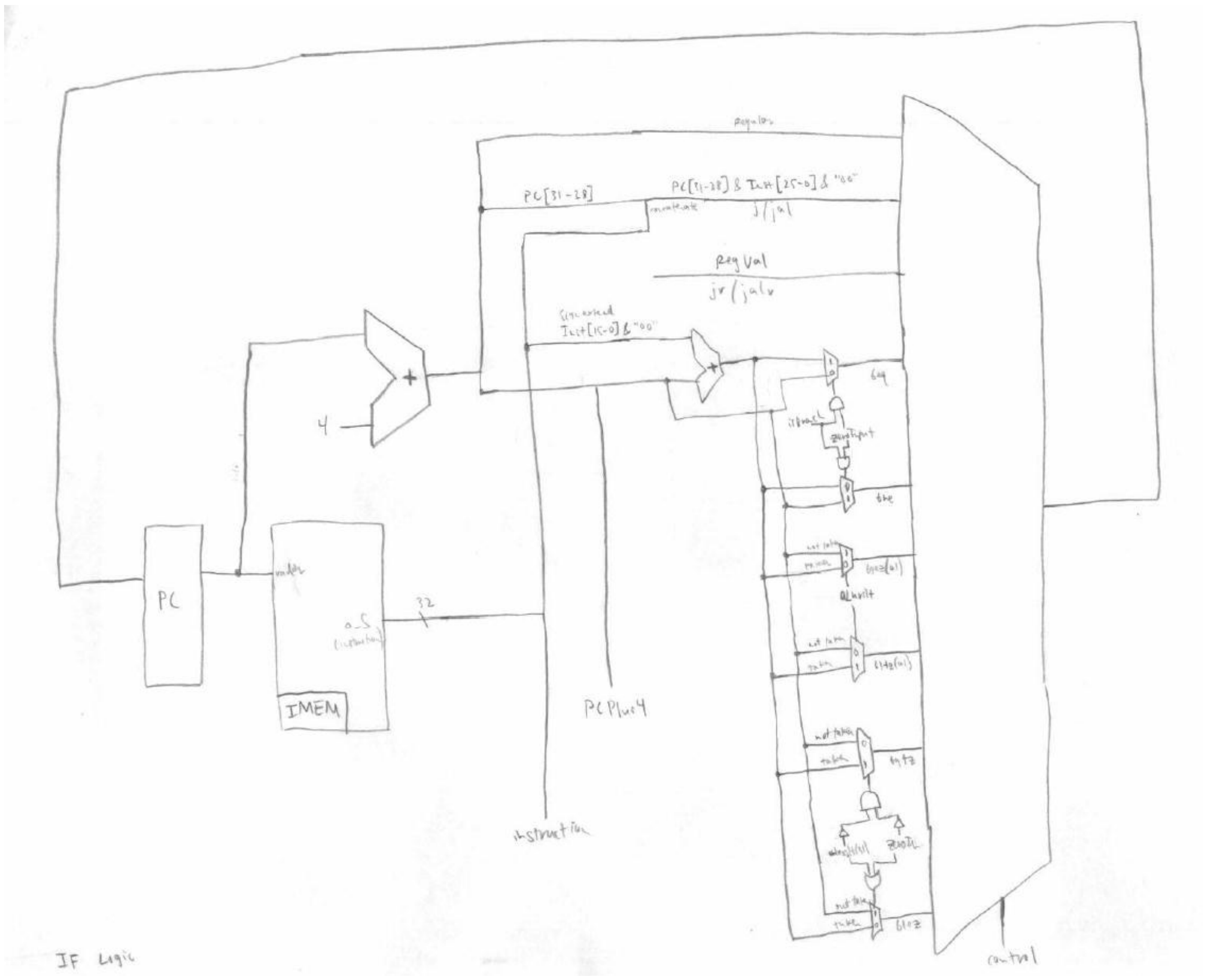


```
Messages
s_alu_op          101
s_alu_src         0
s_instruction     000000000000000
s_isbranch        0
s_isjump          0
s_islui           0
s_leftorright     0
s_log_arith       0
s_memread         0
s_memtoreg        01
s_memwrite        0
s_needslink       0
s_regdest         01
s_regwrite        1
s_shftamount      01
s_w_h_b           000
s_whichjb         0000
```

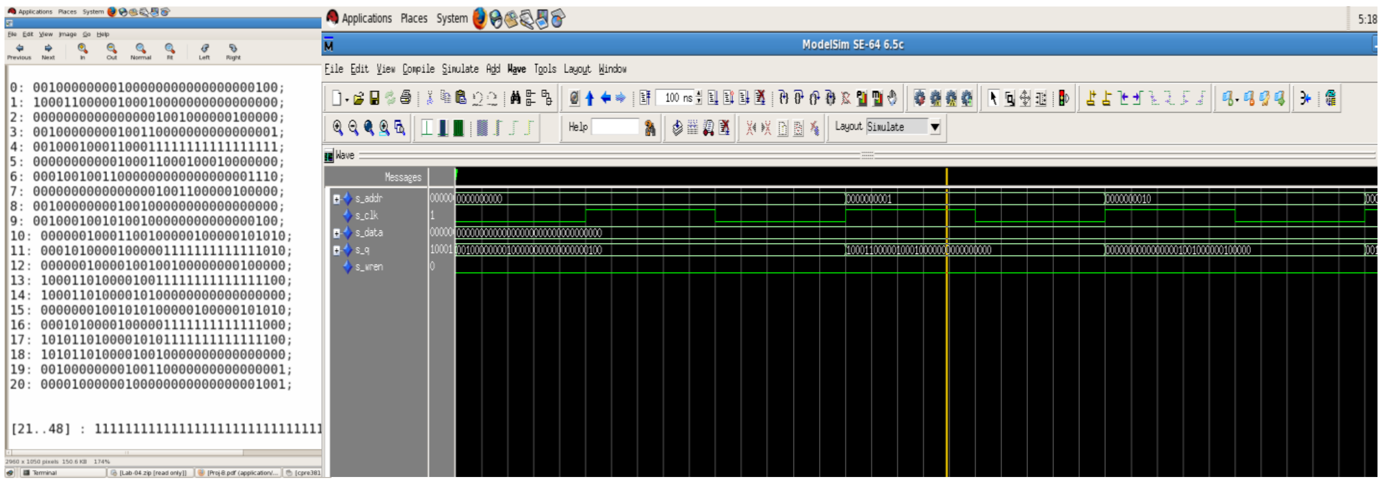
2) Instruction Fetch Logic

(a) The possibilities that our Instruction Fetch Logic must support include: PC+4 (this is just after a normal instruction), all jumps (j, jal, jr, jalr) and all branches (beq, bgez, bgezal, bgtz, blez, bltz, bltzal, bne). Look at Omniscient Spreadsheet for more information dealing with all of the control signals that these will require.

(b) Below is a scanned copy of our instruction fetch component. It is essentially a 9:1 mux that selects between one of our possible next PC values. We calculate all possible PC values concurrently, and then based on the control signal, we select which one to take. For our branch targets, we use combinational logic involving the isBranch signal from the control logic, the zero signal from the ALU, and the ALU result to decide whether a specific branch will be taken. There are two outputs of this component - the instruction, and the PC+4 value, which can be put in the register file in the case of a jump and link operation.



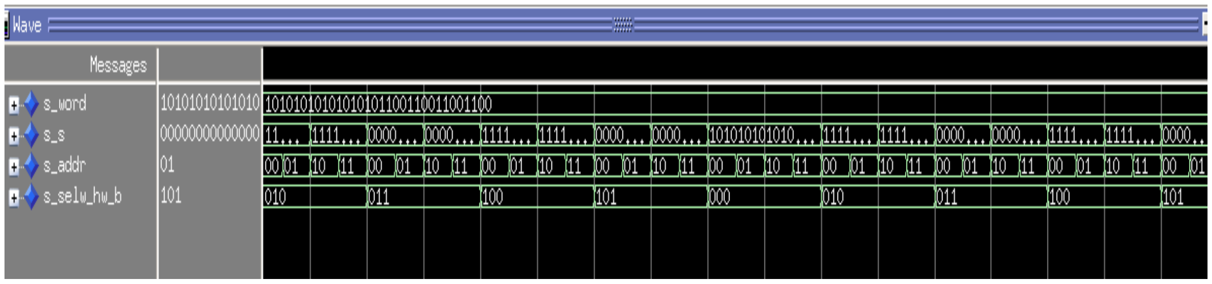
(c) Below is shown an image of a simple testbench that shows that the memory has been instantiated, given the input file, as pictured in the first screenshot. The instruction memory needs less specifity than the *mem.vhd* file has. The only ports that actually matter for the instruction memory are the clock, the input address, and the output. The byteena and data inputs both are never used for instruction memory (because the data is instantiated into the memory), so they don't matter. The write-enable port must be always set to 0, to ensure that new instructions are never written to the instruction memory.



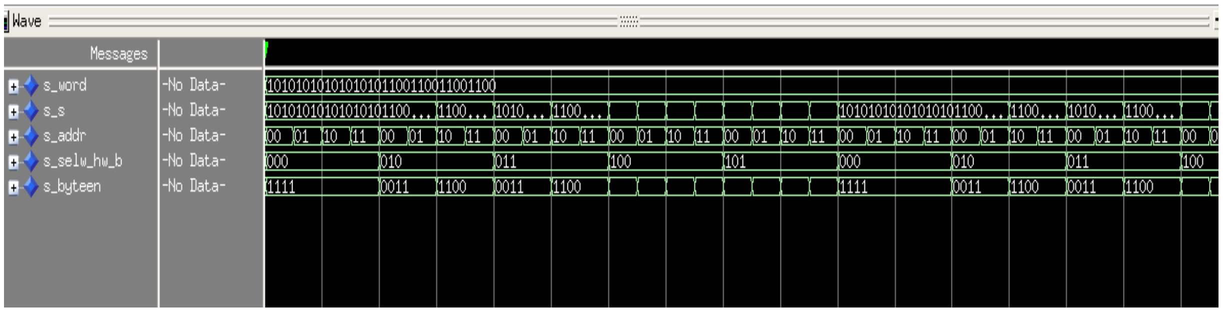
```
0: 00100000000100000000000000000100;
1: 10001100000100010000000000000000;
2: 00000000000000001001000000100000;
3: 00100000000100110000000000000001;
4: 00100010001100011111111111111111;
5: 00000000000100011000100010000000;
6: 00010010011000000000000000001110;
7: 00000000000000001001100000100000;
8: 00100000000100100000000000000000;
9: 00100010010100100000000000000100;
10: 00000010001100100000100000101010;
11: 00010100001000001111111111111010;
12: 00000010000100100100000000100000;
13: 10001101000010011111111111111100;
14: 10001101000010100000000000000000;
15: 00000001001010100000100000101010;
16: 00010100001000001111111111111000;
17: 10101101000010101111111111111100;
18: 10101101000010010000000000000000;
19: 00100000000100110000000000000001;
20: 00001000000100000000000000001001;


[21..48] : 11111111111111111111111111111
```

(d) We created both a store-mux and a load-mux (see the graphic representing our whole processor's schematic to see the location). It was not a possibility to create a test bench to verify if our instruction fetch logic component was working or not, but we were able to at least create test benches to verify whether the load-mux and the store-mux were working. Shown below are waveforms of this output. We later verified that the instruction fetch logic component works by merely running the whole processor and watching the PC signal.

tb_loadmux:



tb_storemux:

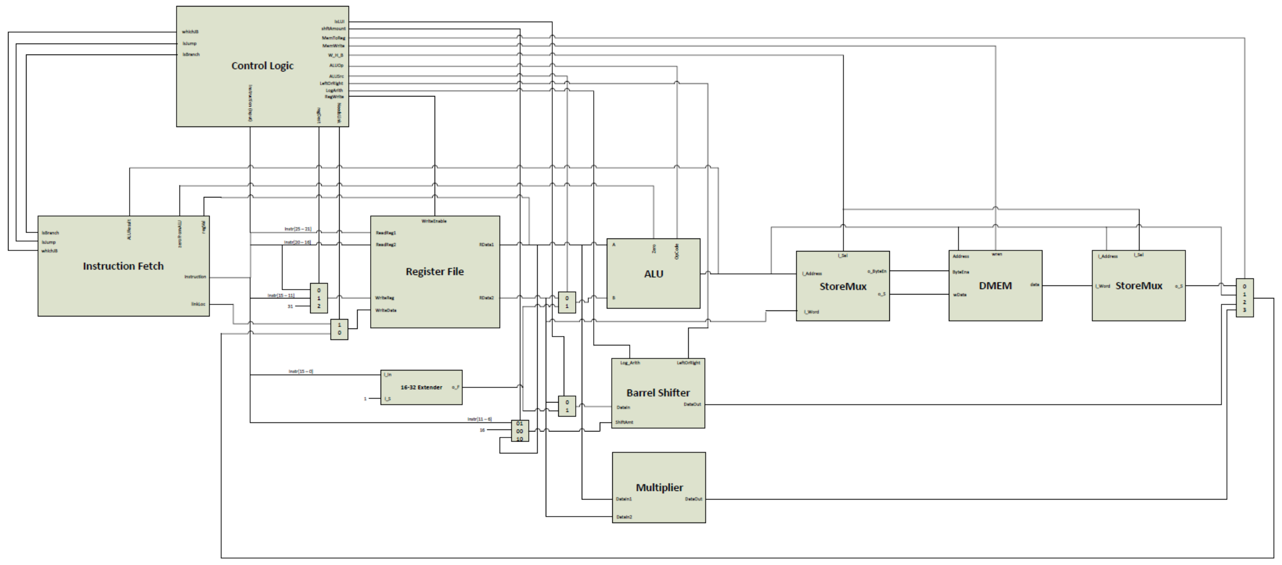


The waveforms indicate the inputs and the outputs of the load-mux and the store-mux.

3) MIPS Single-Cycle Processor

We faced a few challenges when bringing together the entire single-cycle processor. We had to add muxes and new control signals after we thought we had a complete processor. We had to change a few things dealing with muxes and control signals when we went through and found errors during the testing process as well. As far as challenges we faced while actually connecting the processor: we really didn't have any; all of our errors were exposed during the testing process.

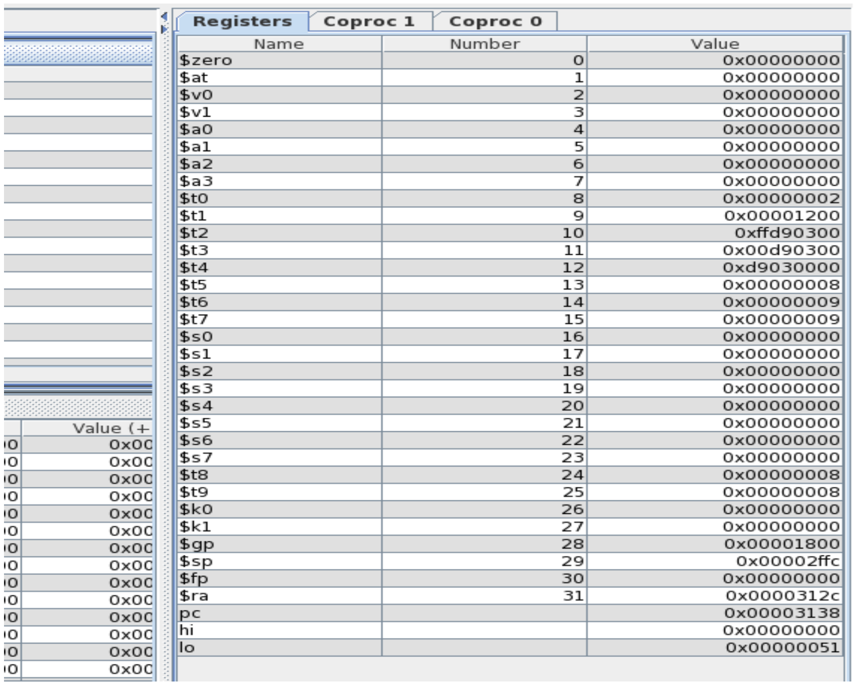
A schematic of our entire processor can be found in our attached zip file under the schematics folder. A screenshot of the file can be seen below. To actually be able to read the text, open the FullProcessor.pdf file from our schematics folder. Our schematic does not include a clock signal, as it would just clutter the diagram more than necessary.



4) Testing

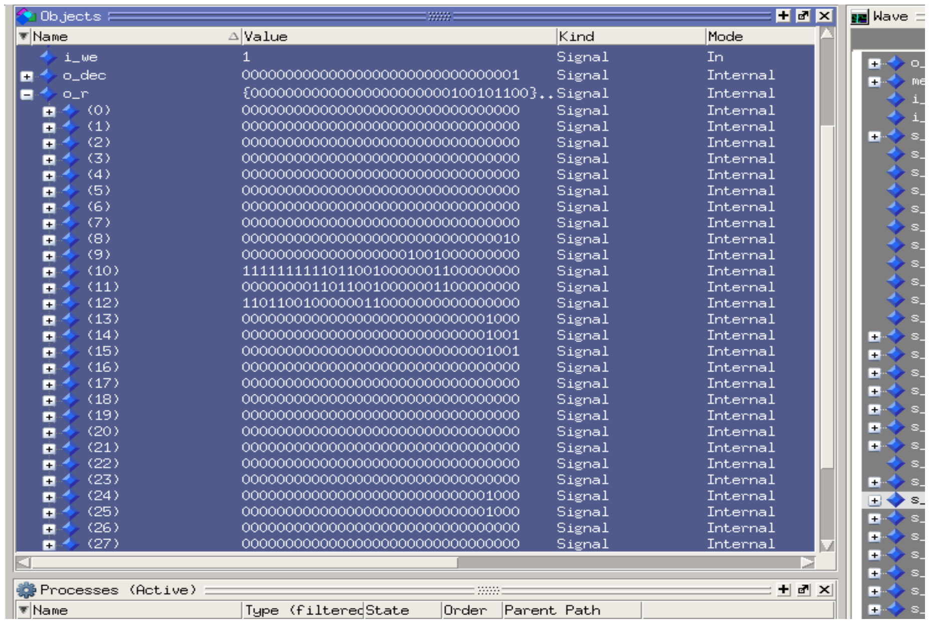
(a) See test/asm/allinstructions.s and test/mifs/allinstructions.mif for the test that makes use of every single instruction. It is important to note that this test does not do anything important, or even interesting. The screenshots below only contain the register information of the end-result just for quick verification of accuracy. This does very little with memory and is thus not important to add a screenshot of the memory output.

MARS screenshot of allinstructions.s:



Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000002
$t1	9	0x00001200
$t2	10	0xffd90300
$t3	11	0x00d90300
$t4	12	0xd9030000
$t5	13	0x00000008
$t6	14	0x00000009
$t7	15	0x00000009
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000008
$t9	25	0x00000008
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x0000312c
pc		0x00003138
hi		0x00000000
lo		0x00000051

ModelSim screenshot of allinstructions.mif:

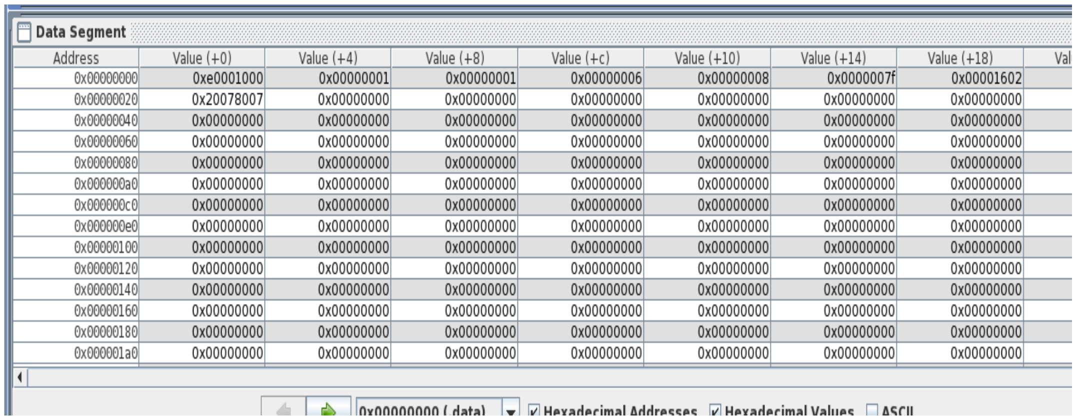


Name	Value	Kind	Mode
i_we	1	Signal	In
o_dec	00000000000000000000000000000001	Signal	Internal
o_r	{00000000000000000000000100101100}..	Signal	Internal
(0)	00000000000000000000000000000000	Signal	Internal
(1)	00000000000000000000000000000000	Signal	Internal
(2)	00000000000000000000000000000000	Signal	Internal
(3)	00000000000000000000000000000000	Signal	Internal
(4)	00000000000000000000000000000000	Signal	Internal
(5)	00000000000000000000000000000000	Signal	Internal
(6)	00000000000000000000000000000000	Signal	Internal
(7)	00000000000000000000000000000000	Signal	Internal
(8)	00000000000000000000000000000010	Signal	Internal
(9)	00000000000000000001001000000000	Signal	Internal
(10)	11111111110110010000001100000000	Signal	Internal
(11)	00000000110110010000001100000000	Signal	Internal
(12)	11011001000000110000000000000000	Signal	Internal
(13)	00000000000000000000000000001000	Signal	Internal
(14)	00000000000000000000000000001001	Signal	Internal
(15)	00000000000000000000000000001001	Signal	Internal
(16)	00000000000000000000000000000000	Signal	Internal
(17)	00000000000000000000000000000000	Signal	Internal
(18)	00000000000000000000000000000000	Signal	Internal
(19)	00000000000000000000000000000000	Signal	Internal
(20)	00000000000000000000000000000000	Signal	Internal
(21)	00000000000000000000000000000000	Signal	Internal
(22)	00000000000000000000000000000000	Signal	Internal
(23)	00000000000000000000000000000000	Signal	Internal
(24)	00000000000000000000000000001000	Signal	Internal
(25)	00000000000000000000000000001000	Signal	Internal
(26)	00000000000000000000000000000000	Signal	Internal
(27)	00000000000000000000000000000000	Signal	Internal

(b) See test/asm/bubblesort_if.s and test/mifs/bubblesort_if.mif for the code for this. The data (as shown below) is in the file test/mifs/bubblesort_data.mif. The way that we initialized memory was that the value at address 0 in dmem is the number of elements to be sorted, and then the following elements were the items to be sorted. Our data was initialized as follows:

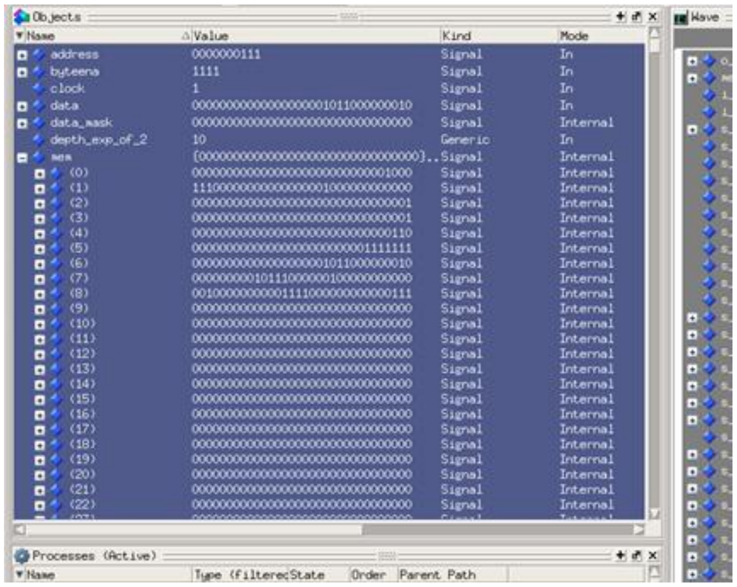
```
CONTENT
BEGIN

0: 00000000000000000000000000001000;
1: 00000000000000000000000000000110;
2: 00000000000000000000000000000001;
3: 00000000000000000001011000000010;
4: 00000000010111000000100000000000;
5: 11100000000000000001000000000000;
6: 00100000000000111100000000000111;
7: 00000000000000000000000000000001;
8: 00000000000000000000000001111111;
9: 00000000000000000000000000000000;

[10..1023] : 00000000000000000000000000000000;

END;
```

The following screenshots depict the sorted memory after execution in both MARS and ModelSim. It is really only of use to view the memory after execution, to see that the data has been sorted correctly.
MARS screenshot of bubblesort_if.s:



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Val
0x00000000	0xe0001000	0x00000001	0x00000001	0x00000006	0x00000008	0x0000007f	0x00001602	
0x00000020	0x20078007	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	
0x000001a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	

ModelSim screenshot of bubblesort_if.mif (note that address 0 is merely the value of the number of elements, so that is not sorted) :

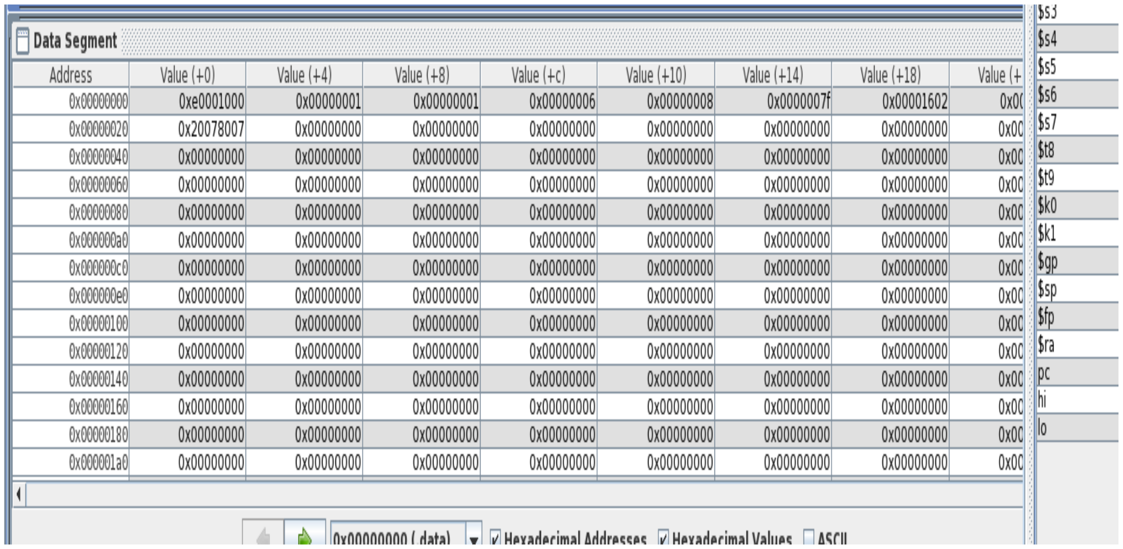


Name	Value	Kind	Mode
address	0000000111	Signal	In
byteena	1111	Signal	In
clock	1	Signal	In
data	00000000000000000010110000000010	Signal	In
data_mask	00000000000000000000000000000000	Signal	Internal
depth_exp_of_2	10	Generic	In
mem	{00000000000000000000000000000000}..	Signal	Internal
(0)	00000000000000000000000000001000	Signal	Internal
(1)	11100000000000000001000000000000	Signal	Internal
(2)	00000000000000000000000000000001	Signal	Internal
(3)	00000000000000000000000000000001	Signal	Internal
(4)	00000000000000000000000000000110	Signal	Internal
(5)	00000000000000000000000001111111	Signal	Internal
(6)	00000000000000000010110000000010	Signal	Internal
(7)	00000000101110000001000000000000	Signal	Internal
(8)	00100000000001111000000000000111	Signal	Internal
(9)	00000000000000000000000000000000	Signal	Internal
(10)	00000000000000000000000000000000	Signal	Internal
(11)	00000000000000000000000000000000	Signal	Internal
(12)	00000000000000000000000000000000	Signal	Internal
(13)	00000000000000000000000000000000	Signal	Internal
(14)	00000000000000000000000000000000	Signal	Internal
(15)	00000000000000000000000000000000	Signal	Internal
(16)	00000000000000000000000000000000	Signal	Internal
(17)	00000000000000000000000000000000	Signal	Internal
(18)	00000000000000000000000000000000	Signal	Internal
(19)	00000000000000000000000000000000	Signal	Internal
(20)	00000000000000000000000000000000	Signal	Internal
(21)	00000000000000000000000000000000	Signal	Internal
(22)	00000000000000000000000000000000	Signal	Internal

Name	Type (filtered)	State	Order	Parent Path

(c) For the MergeSort algorithm, see test/asm/MergeSort.s, test/asm/merge.mif, and for the data initialization, we used the same file as the BubbleSort algorithm, test/mifs/bubblesort_data.mif. The data was initialized in the exact same way as for the BubbleSort algorithm, and even the same data was used. Attached are screenshots of both MARS and ModelSim, showing that the outputs are the same. These outputs will be the same as the outputs from the BubbleSort algorithm, they were just attained using a different algorithm.

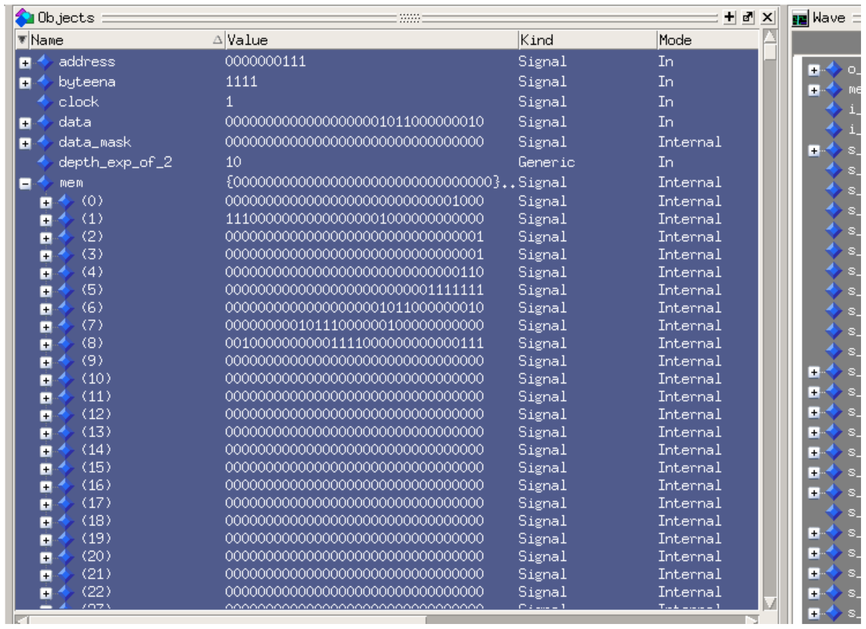
MARS screenshot of MergeSort.s:



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+
0x00000000	0xe0001000	0x00000001	0x00000001	0x00000006	0x00000008	0x0000007f	0x00001602	0x0(
0x00000020	0x20078007	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x00000100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x00000120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x00000140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x00000160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x00000180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00
0x000001a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00

ModelSim screenshot of merge.mif (note again that address 0 is just the number of elements to be sorted):



Objects

Name	Value	Kind	Mode
address	0000000111	Signal	In
byteena	1111	Signal	In
clock	1	Signal	In
data	00000000000000000001011000000010	Signal	In
data_mask	00000000000000000000000000000000	Signal	Internal
depth_exp_of_2	10	Generic	In
mem	{00000000000000000000000000000000}..	Signal	Internal
(0)	00000000000000000000000000001000	Signal	Internal
(1)	11100000000000000001000000000000	Signal	Internal
(2)	00000000000000000000000000000001	Signal	Internal
(3)	00000000000000000000000000000001	Signal	Internal
(4)	00000000000000000000000000000110	Signal	Internal
(5)	00000000000000000000000001111111	Signal	Internal
(6)	00000000000000000001011000000010	Signal	Internal
(7)	00000000101110000000100000000000	Signal	Internal
(8)	00100000000001111000000000000111	Signal	Internal
(9)	00000000000000000000000000000000	Signal	Internal
(10)	00000000000000000000000000000000	Signal	Internal
(11)	00000000000000000000000000000000	Signal	Internal
(12)	00000000000000000000000000000000	Signal	Internal
(13)	00000000000000000000000000000000	Signal	Internal
(14)	00000000000000000000000000000000	Signal	Internal
(15)	00000000000000000000000000000000	Signal	Internal
(16)	00000000000000000000000000000000	Signal	Internal
(17)	00000000000000000000000000000000	Signal	Internal
(18)	00000000000000000000000000000000	Signal	Internal
(19)	00000000000000000000000000000000	Signal	Internal
(20)	00000000000000000000000000000000	Signal	Internal
(21)	00000000000000000000000000000000	Signal	Internal
(22)	00000000000000000000000000000000	Signal	Internal