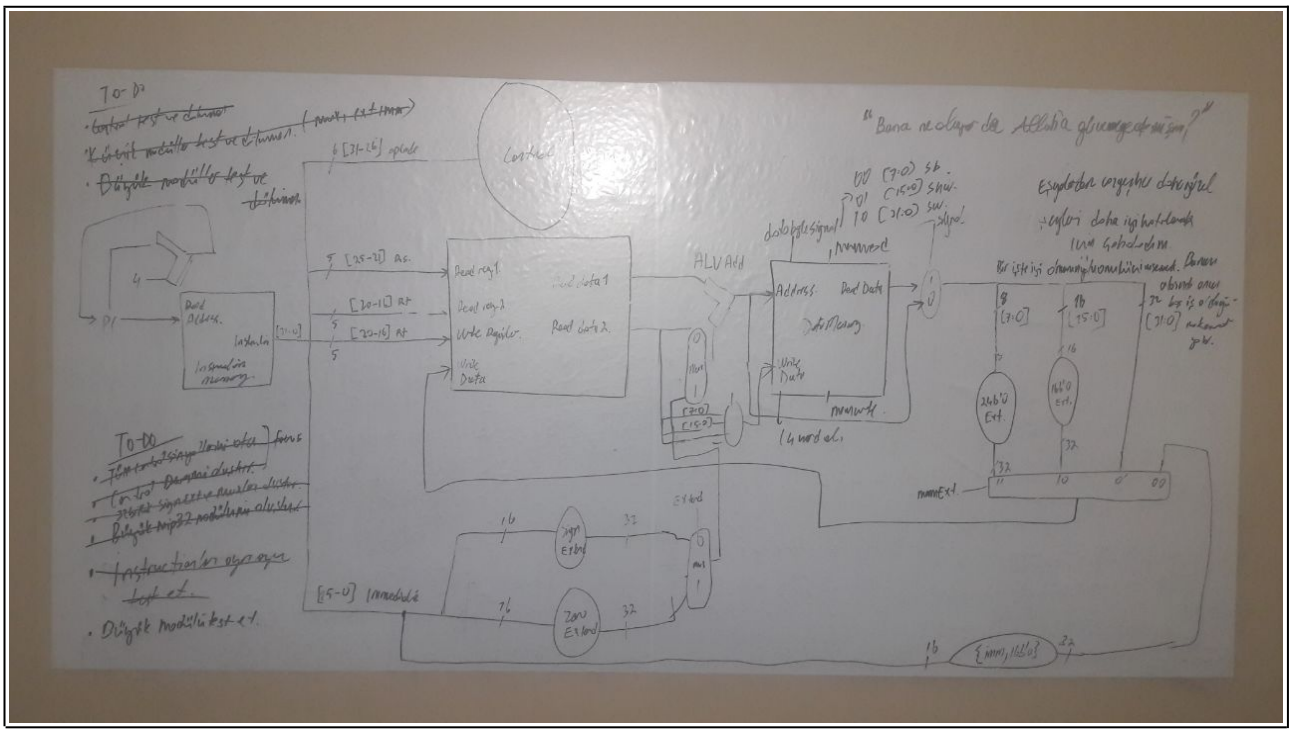
CSE331 Assignment Report

Ahmed Semih Özmekik

Design

Here is my design:



Problem (mips32_testbench.v)

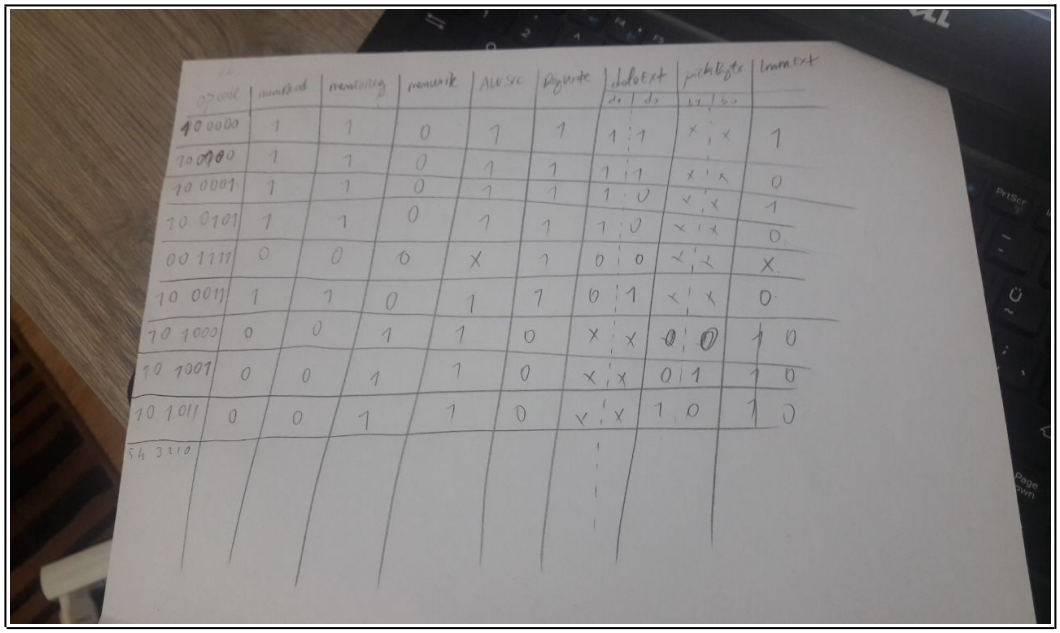
Everything is working fine in modulas. But for this homework, I have small problem that I couldn't figure it out with integration. Whenever I try to use Program Counter with instruction memory and start the mips processor; 2-3 of the instructions could not be fetched by instruction memory. Therefore can't be decoded and executed.

```
# PC -  1, write_data-         x     [xxxxxxxx]     [xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx].
# PC -  2, write_data-        12     [0000000c]     [00000000000000000000000000001100].
# PC -  3, write_data-        13     [0000000d]     [00000000000000000000000000001101].
# PC -  4, write_data-        14     [0000000e]     [00000000000000000000000000001110].
# PC -  5, write_data-        15     [0000000f]     [00000000000000000000000000001111].
# PC -  6, write_data-        16     [00000010]     [00000000000000000000000000010000].
# PC -  7, write_data-         x     [xxxxxxxx]     [xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx].
# PC -  8, write_data-3192944621     [be5077ed]     [10111110010100000111011111101101].
# PC -  9, write_data-    983040     [000F0000]     [00000000000011110000000000000000].
# PC - 10, write_data-   2031616     [001F0000]     [00000000000111110000000000000000].
# PC - 11, write_data-     59262     [0000e77e]     [00000000000000001110011101111110].
# PC - 12, write_data-         x     [0000xxxx]     [0000000000000000xxxxxxxxxxxxxxxx].
# PC - 13, write_data-     43806     [0000ab1e]     [00000000000000001010101100011110].
# PC - 14, write_data-         x     [0000xxxx]     [0000000000000000xxxxxxxxxxxxxxxx].
# PC - 15, write_data-        30     [0000001e]     [00000000000000000000000000011110].
# PC - 16, write_data-        17     [00000011]     [00000000000000000000000000010001].
# PC - 17, write_data-        30     [0000001e]     [00000000000000000000000000011110].
# PC - 18, write_data-        17     [00000011]     [00000000000000000000000000010001].
```

As you can see, 7th, 12th, and 14th instruction couldn't fetched. At the end of this document I will be showing that each of this instructions are working properly if I hardcode them into testbench. This test is done on file instruction_testbench.v. Other than this stated issue, there is no other problem.

Control Module (control_testbench.v)

First, I have created the truth table as given below:



Regarding to this, I have found the efficient logic equations to derive the circuit. Here is the test results showing the circuit is working correctly:

```
# opcode-20, MemRead-1, MemToReg-1, MemWrite-0, RegWrite-1, dataExt-11, pickByte-00, immExt-1
# opcode-24, MemRead-1, MemToReg-1, MemWrite-0, RegWrite-1, dataExt-11, pickByte-00, immExt-0
# opcode-21, MemRead-1, MemToReg-1, MemWrite-0, RegWrite-1, dataExt-10, pickByte-01, immExt-1
# opcode-25, MemRead-1, MemToReg-1, MemWrite-0, RegWrite-1, dataExt-10, pickByte-01, immExt-0
# opcode-0F, MemRead-0, MemToReg-0, MemWrite-0, RegWrite-1, dataExt-00, pickByte-10, immExt-0
# opcode-23, MemRead-1, MemToReg-1, MemWrite-0, RegWrite-1, dataExt-01, pickByte-10, immExt-0
# opcode-28, MemRead-0, MemToReg-0, MemWrite-1, RegWrite-0, dataExt-11, pickByte-00, immExt-0
# opcode-29, MemRead-0, MemToReg-0, MemWrite-1, RegWrite-0, dataExt-10, pickByte-01, immExt-0
# opcode-2b, MemRead-0, MemToReg-0, MemWrite-1, RegWrite-0, dataExt-01, pickByte-10, immExt-0
```

Control module works as it supposed to do. In any case if we face an issue in testing the mips32 module, the reason might the uncorrectly created truth table for opcodes. We will be testing the signals in mips32 module.

Instruction Memory Module (instruction_memory_testbench.v)

Here is *inst.txt*:

10000000001000110000000000000001
10010000100001100000000000001110
10010101001011110000000000100100
00111101110000000000000000111111
10100000011000010000000000000001
10100100001000000000000000000000
10101101001100010000000000000000

Here is the assembly:

load byte, 1, 3, 1
load byte unsigned, 4, 6, 14
load halfword, 9, 15, 36
load upper imm, 15, 0, 63
store byte, 3, 1, 1
store halfword, 1, 0, 0
store word, 9, 17, 0

Here is read instructions from inst.txt memory file:

```
# Inst [ 0] : 0x10000000001000110000000000000001
# Inst [ 1] : 0x10010000100001100000000000001110
# Inst [ 2] : 0x10010101001011110000000000100100
# Inst [ 3] : 0x00111101110000000000000000111111
# Inst [ 4] : 0x10100000011000010000000000000001
# Inst [ 5] : 0x10100100001000000000000000000000
# Inst [ 6] : 0x10101101001100010000000000000000
```

```
# Inst [ 0]   Opcode-100000 [20], Rs-00001 [01], Rt-00011 [03], Imm-0000000000000001 [0001]
# Inst [ 1]   Opcode-100100 [24], Rs-00100 [04], Rt-00110 [06], Imm-0000000000001110 [000e]
# Inst [ 2]   Opcode-100101 [25], Rs-01001 [09], Rt-01111 [0f], Imm-0000000000100100 [0024]
# Inst [ 3]   Opcode-001111 [0f], Rs-01110 [0e], Rt-00000 [00], Imm-0000000000111111 [003f]
# Inst [ 4]   Opcode-101000 [28], Rs-00011 [03], Rt-00001 [01], Imm-0000000000000001 [0001]
# Inst [ 5]   Opcode-101001 [29], Rs-00001 [01], Rt-00000 [00], Imm-0000000000000000 [0000]
# Inst [ 6]   Opcode-101011 [2b], Rs-01001 [09], Rt-10001 [11], Imm-0000000000000000 [0000]
```

There won't be any writings to instruction memory. And reading from instruction memory is working as expected.

Register Memory Module (register_testbench.v)

Here is *register.txt*, all 32 registers are zeroed:

00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
...
00000000000000000000000000000000

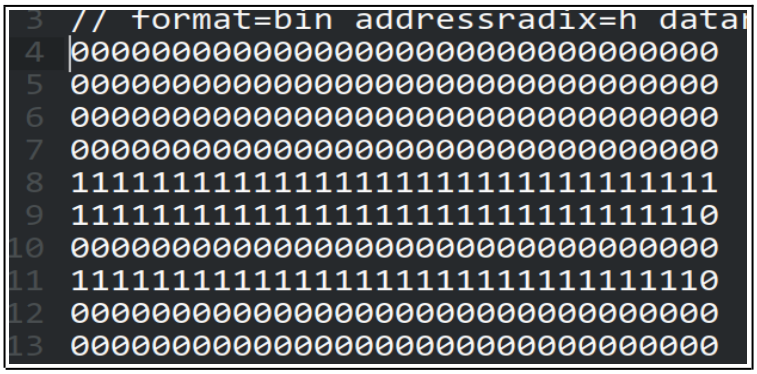
Here is test bench:

```
read_reg1 = 5'b00000;
read_reg2 = 5'b00001;
write_reg = 5'b00000;
reg_write = 0;
write_data = 32'b00000000000000000000000000000000 ;
#`DELAY;
read_reg1 = 5'b00010;
read_reg2 = 5'b00011;
write_reg = 5'b00000;
reg_write = 0;
write_data = 32'b00000000000000000000000000000000 ;
#`DELAY;
read_reg1 = 5'b00111;
read_reg2 = 5'b00110;
write_reg = 5'b00100;
reg_write = 1;
write_data = 32'b11111111111111111111111111111111 ;
#`DELAY;
read_reg1 = 5'b00100;
read_reg2 = 5'b00001;
write_reg = 5'b00101;
reg_write = 1;
write_data = 32'b11111111111111111111111111111110 ;
#`DELAY;
read_reg1 = 5'b00101;
read_reg2 = 5'b00001;
write_reg = 5'b00111;
reg_write = 1;
write_data = 32'b11111111111111111111111111111110 ;
#`DELAY;
```

```
# read_reg1-00000, read_reg2-00001, write_reg-00000, reg_write-0, read_data1-00000000000000000000000000000000,
# read_reg1-00010, read_reg2-00011, write_reg-00000, reg_write-0, read_data1-00000000000000000000000000000000,
# read_reg1-00111, read_reg2-00110, write_reg-00100, reg_write-1, read_data1-00000000000000000000000000000000,
# read_reg1-00100, read_reg2-00001, write_reg-00101, reg_write-1, read_data1-11111111111111111111111111111111,
# read_reg1-00101, read_reg2-00001, write_reg-00111, reg_write-1, read_data1-11111111111111111111111111111110,
# ** Note: $finish    : /home/drh0use/altera/13.1/hw2_restored/workspace/test/register_testbench.v(48)
#    Time: 220 ps  Iteration: 0  Instance: /register_testbench
```

```
read_data2-00000000000000000000000000000000,  write_data-00000000000000000000000000000000
read_data2-00000000000000000000000000000000,  write_data-00000000000000000000000000000000
read_data2-00000000000000000000000000000000,  write_data-11111111111111111111111111111111
read_data2-00000000000000000000000000000000,  write_data-11111111111111111111111111111110
read_data2-00000000000000000000000000000000,  write_data-11111111111111111111111111111110
```

Test #1 : It only reads from r0 and r1. Since they are both zero. read_data1, and read_data2 is 0.
Test #2: Same reading test done as the first test.
Test #3: Write -1, to r4.
Test #4: Read from r4. It is read as -1 as it supposed to do. Since we changed the r4 in Test #3. And write -2 to r5.
Test #5: Read from r5. It is read as -2. Write -2 to r7. Let's the final view of *register.txt.*



8th line : r4 = -1
9th line: r5 = -2
11th line: r7 = -2

They are changed as expected. Writing and reading from register memory is working correctly.

Data Memory Module (data_memory_testbench.v)

Here is *data_memory.txt* before start: 32 address exist with 32 bits length.

00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
...
00000000000000000000000000000000

Here is the test bench. (Left one is in above, right is in below.):

```
clk = 1;
address = 0; // do nothing.
write_data = 1;
mem_read = 0;
mem_write = 0;
pick_byte = 0;
#`DELAY;
address = 0; // read add0.
write_data = 1;
mem_read = 1;
mem_write = 0;
pick_byte = 0;
#`DELAY;
address = 0; // write a word to add0.
write_data = 32'hffffffff;
mem_read = 0;
mem_write = 1;
pick_byte = 2;
#`DELAY;
address = 0; // read from add0.
write_data = 8'hff;
mem_read = 1;
mem_write = 0;
pick_byte = 2;
```

```
address = 1; // write a halfword to add1.
write_data = 32'hffffffff;
mem_read = 1;
mem_write = 1;
pick_byte = 1;
#`DELAY;
address = 1; // read from add1.
write_data = 32'hffffffff;
mem_read = 1;
mem_write = 0;
pick_byte = 1;
#`DELAY;
address = 2; // write a byte to add2.
write_data = 32'hffffffff;
mem_read = 1;
mem_write = 1;
pick_byte = 0;
#`DELAY;
address = 2; // read from add2
write_data = 32'hffffffff;
mem_read = 1;
mem_write = 0;
pick_byte = 0;
#`DELAY;
```

```
# address -          0, write_data-00000001, mem_read-0, mem_write-0, pick_byte-00, read_data-xxxxxxxx
# address -          0, write_data-00000001, mem_read-1, mem_write-0, pick_byte-00, read_data-00000000
# address -          0, write_data-FFFFFFFF, mem_read-0, mem_write-1, pick_byte-10, read_data-00000000
# address -          0, write_data-000000FF, mem_read-1, mem_write-0, pick_byte-10, read_data-FFFFFFFF
# address -          1, write_data-FFFFFFFF, mem_read-1, mem_write-1, pick_byte-01, read_data-00000000
# address -          1, write_data-FFFFFFFF, mem_read-1, mem_write-0, pick_byte-01, read_data-0000FFFF
# address -          2, write_data-FFFFFFFF, mem_read-1, mem_write-1, pick_byte-00, read_data-00000000
# address -          2, write_data-FFFFFFFF, mem_read-1, mem_write-0, pick_byte-00, read_data-000000FF
```

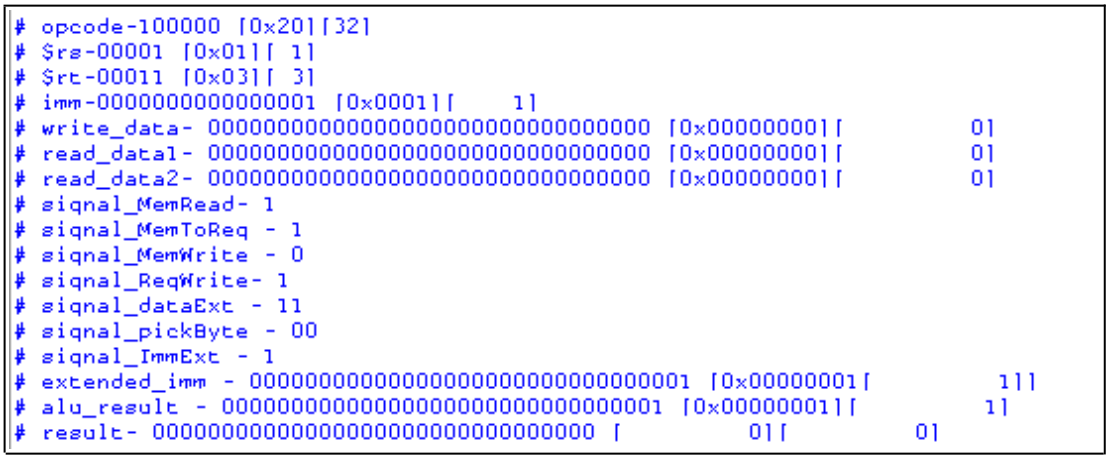
Test #1 : All signals are set 0. Read data is being don't care as expected.
Test #2: Read add0. Which is 0.
Test #3: Writes a word (0xffffffff) to add0. (pick_byte=0 byte, pick_byte=1 half_word, pick_byte=3 word.)
Test #4: Reads from add0, which is the value written in test #3.
Test #5: Writes a halfword to add1.
Test #6: Read from add1, which is the value written in test #5.
Test #7: Writes a byte to add2.
Test #8: Read from add2, which is the value written in test #7.

```
4  11111111111111111111111111111111
5  00000000000000001111111111111111
6  00000000000000000000000011111111
7  00000000000000000000000000000000
8  00000000000000000000000000000000
9  00000000000000000000000000000000
10 00000000000000000000000000000000
```

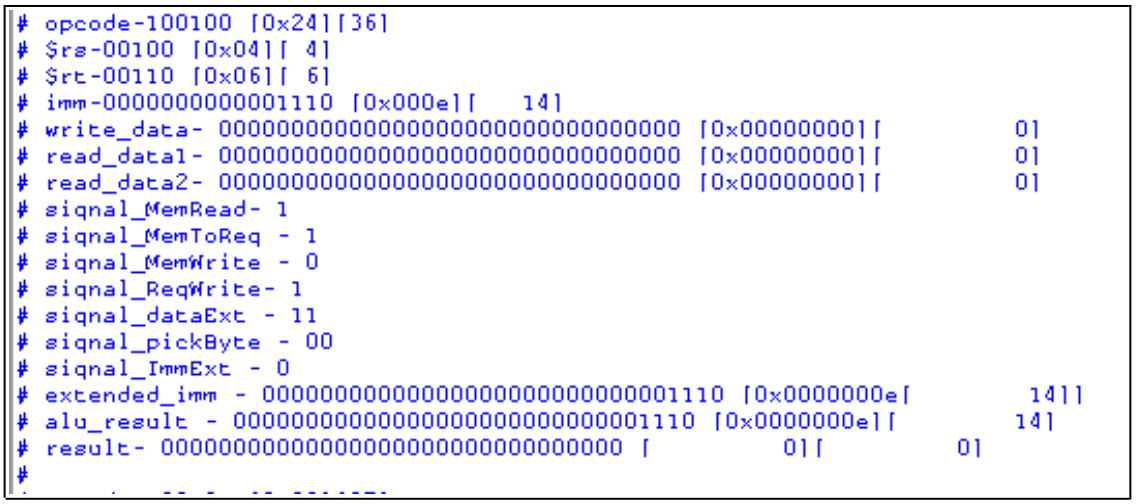
4th line : add0 = 0xffffffff
5th line : add1 = 0xffff
6th line: add2 = 0xffff
All signals are working correctly and results written to memory file as expected. Writing and reading from data memory is working correctly.

Testing Each Instruction (mip32_instruction_testbench.v)

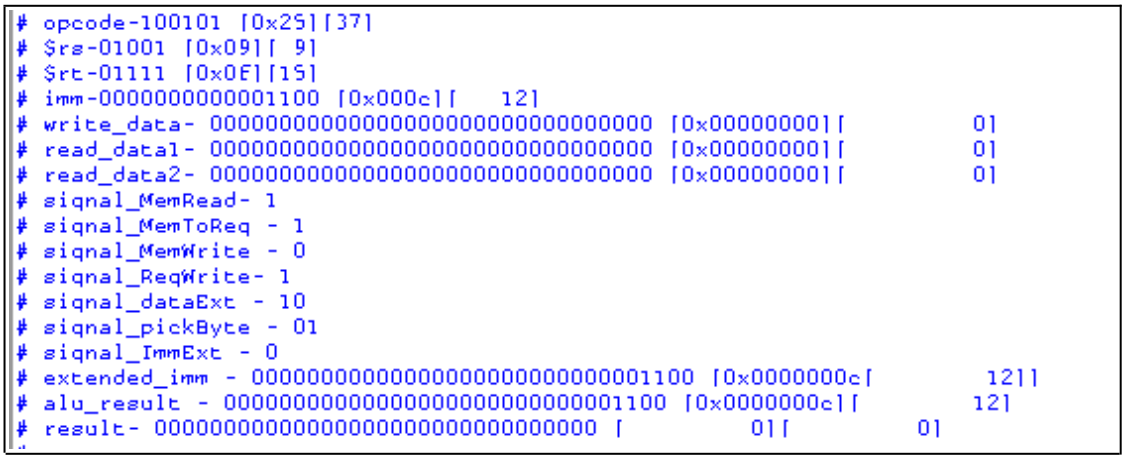
Instruction : load byte, 1, 3, 1



Instruction : load byte unsigned, 4, 6, 14



Instruction : load halfword, 9, 15, 12



Instruction : load upper imm, 15, 0, 63

```
# opcode-001111 [0x0F][15]
# $rs-01110 [0x0e][14]
# $rt-00000 [0x00][ 0]
# imm-0000000000111111 [0x003F][   63]
# write_data- 00000000001111110000000000000000 [0x003F0000][   4128768]
# read_data1- 00000000000000000000000000000000 [0x00000000][         0]
# read_data2- 00000000001111110000000000000000 [0x003F0000][   4128768]
# signal_MemRead- 0
# signal_MemToReg - 0
# signal_MemWrite - 0
# signal_RegWrite- 1
# signal_dataExt - 00
# signal_pickByte - 10
# signal_ImmExt - 0
# extended_imm - 00000000000000000000000000111111 [0x0000003F[        63]]
# alu_result - 00000000000000000000000000111111 [0x0000003F]][        63]
# result- 00000000001111110000000000000000 [   4128768]][   4128768]
#
# ----d- 101000 [0x28][40]
```

Instruction: store word, 2, 0, 2 // Mem[2] = $r0 ($r0 = 4128768 from above.)

```
# opcode-101011 [0x2b][43]
# $rs-00010 [0x02][ 2]
# $rt-00000 [0x00][ 0]
# imm-0000000000000010 [0x0002][    2]
# write_data- 00000000000000000000000000000010 [0x00000002][         2]
# read_data1- 00000000000000000000000000000000 [0x00000000][         0]
# read_data2- 00000000001111110000000000000000 [0x003F0000][   4128768]
# signal_MemRead- 0
# signal_MemToReg - 0
# signal_MemWrite - 1
# signal_RegWrite- 0
# signal_dataExt - 01
# signal_pickByte - 10
# signal_ImmExt - 0
# extended_imm - 00000000000000000000000000000010 [0x00000002[         2]]
# alu_result - 00000000000000000000000000000010 [0x00000002]][         2]
# result- 00000000000000000000000000000010 [         2]][         2]
#
```

Instruction: load word, 1, 2, 2 // $r2 = Mem[2] (Mem[2] = 4128768)

```
# opcode-100011 [0x23][35]
# $rs-00001 [0x01][ 1]
# $rt-00010 [0x02][ 2]
# imm-0000000000000010 [0x0002][    2]
# write_data- 00000000001111110000000000000000 [0x003F0000][   4128768]
# read_data1- 00000000000000000000000000000000 [0x00000000][         0]
# read_data2- 00000000000000000000000000000000 [0x00000000][         0]
# signal_MemRead- 1
# signal_MemToReg - 1
# signal_MemWrite - 0
# signal_RegWrite- 1
# signal_dataExt - 01
# signal_pickByte - 10
# signal_ImmExt - 0
# extended_imm - 00000000000000000000000000000010 [0x00000002[         2]]
# alu_result - 00000000000000000000000000000010 [0x00000002]][         2]
# result- 00000000001111110000000000000000 [   4128768]][   4128768]
#
```

Instruction: store byte, 3, 1, 1

```
# opcode-101000 [0x28][40]
# $rs-00011 [0x03][ 3]
# $rt-00001 [0x01][ 1]
# imm-0000000000000001 [0x0001][    1]
# write_data- 00000000000000000000000000000001 [0x00000001][         1]
# read_data1- 00000000000000000000000000000000 [0x00000000][         0]
# read_data2- 00000000000000000000000000000000 [0x00000000][         0]
# signal_MemRead- 0
# signal_MemToReg - 0
# signal_MemWrite - 1
# signal_RegWrite- 0
# signal_dataExt - 11
# signal_pickByte - 00
# signal_ImmExt - 0
# extended_imm - 00000000000000000000000000000001 [0x00000001[         1]]
# alu_result - 00000000000000000000000000000001 [0x00000001][         1]
# result- 00000000000000000000000000000001 [         1][         1]
#
```

Instruction: store halfword, 1, 0, 0

```
# opcode-101001 [0x29][41]
# $rs-00001 [0x01][ 1]
# $rt-00000 [0x00][ 0]
# imm-0000000000000000 [0x0000][    0]
# write_data- 00000000000000000000000000000000 [0x00000000][         0]
# read_data1- 00000000000000000000000000000000 [0x00000000][         0]
# read_data2- 00000000001111110000000000000000 [0x003F0000][   4128768]
# signal_MemRead- 0
# signal_MemToReg - 0
# signal_MemWrite - 1
# signal_RegWrite- 0
# signal_dataExt - 10
# signal_pickByte - 01
# signal_ImmExt - 0
# extended_imm - 00000000000000000000000000000000 [0x00000000[         0]]
# alu_result - 00000000000000000000000000000000 [0x00000000][         0]
# result- 00000000000000000000000000000000 [         0][         0]
#
```

All ALU results, parsers, signals, writing and readings are working as expected.

Mips32 Module (instruction_testbench.v)

There will be 2 test samples for each instruction.

$r2 = 0xbeefdead

$r3 = 10

$r4 = 0xbabadede

$r6 = 0xba5eba11

$r7 = 0xbedabb1e

$r20 = 20

M[17] = 0xbe5077ed

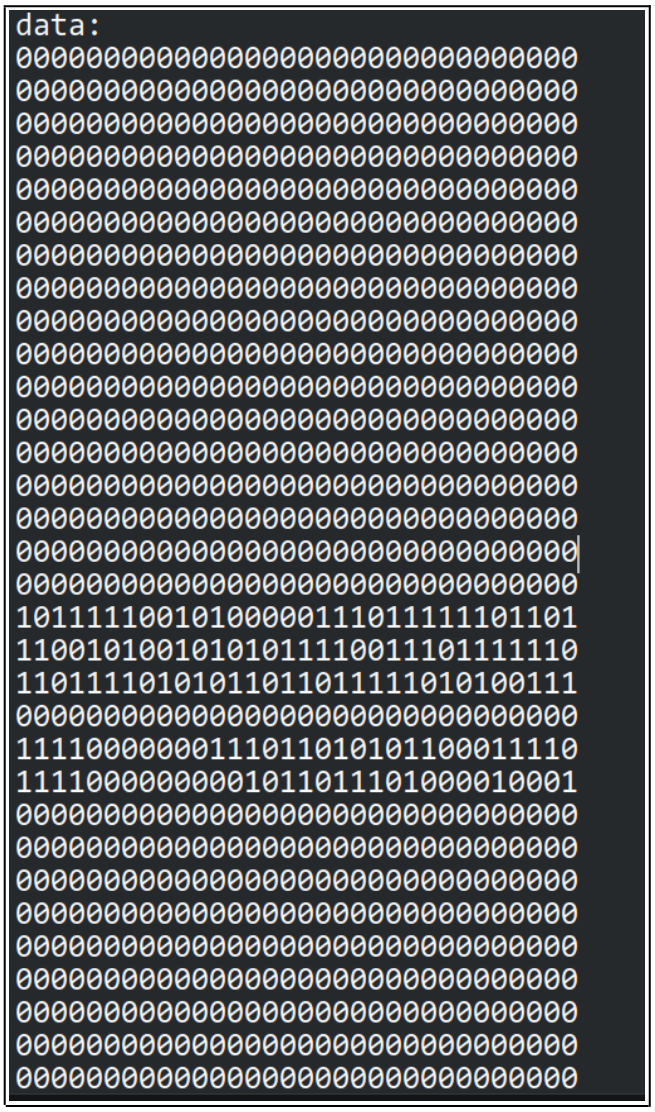
M[18] = 0xca55e77e

M[19] = 0xdeadbea7

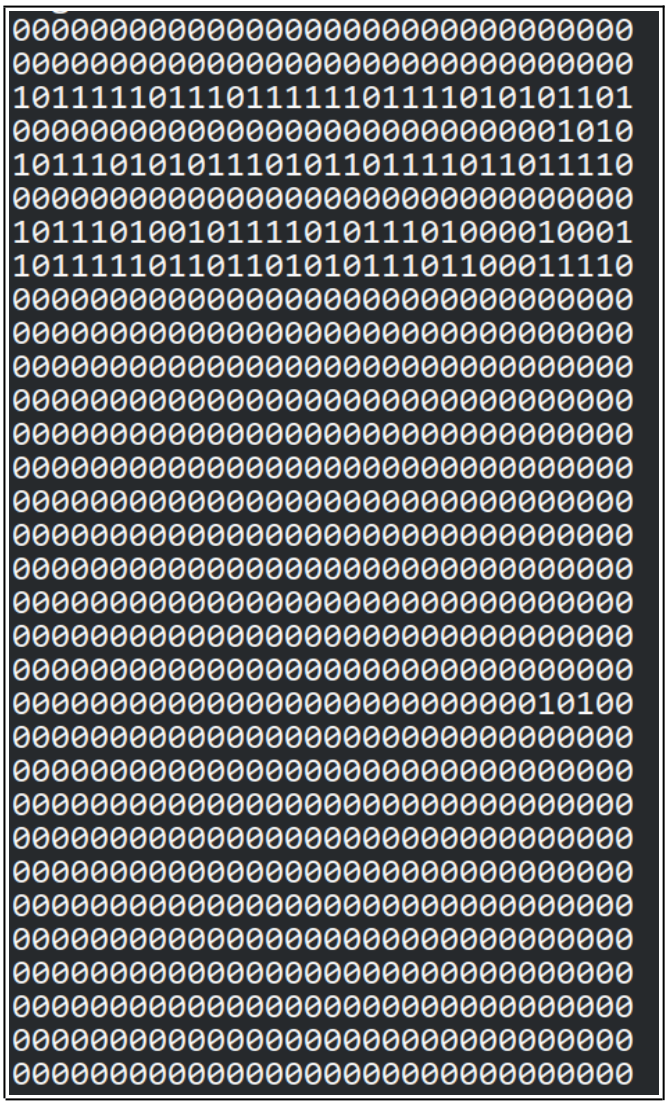
M[21] = 0xf01dab1e

M[22] = 0xf005ba11

Memory Before:



Register Before:



Store Word

Sw, 3, 4, 1 (M[$r3 + 1] = $r4)

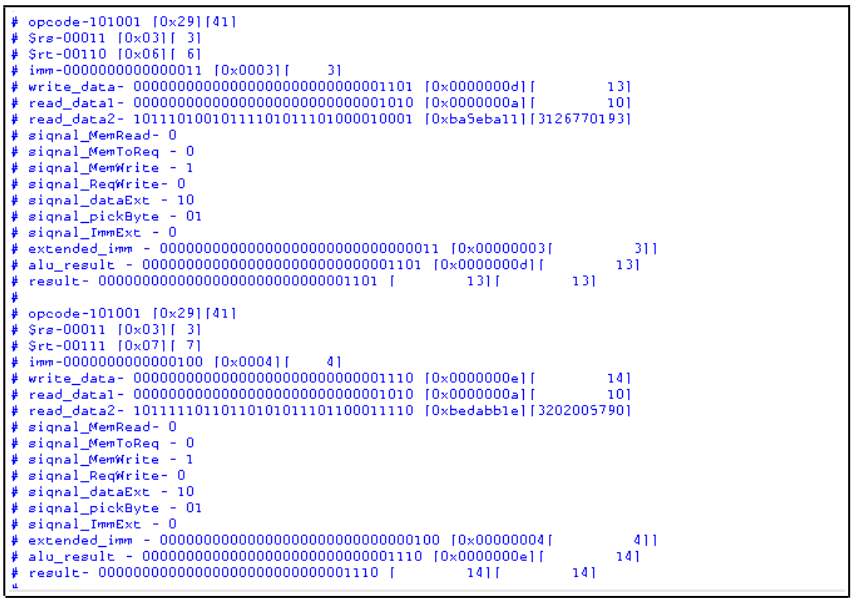
sw, 3, 2, 2 (M[$r3 + 2] = $r2)

```
# opcode-101011 [0x2b][43]
# $rs-00011 [0x03][ 3]
# $rt-00100 [0x04][ 4]
# imm-0000000000000001 [0x0001][    1]
# write_data- 00000000000000000000000000001011 [0x0000000b][        11]
# read_data1- 00000000000000000000000000001010 [0x0000000a][        10]
# read_data2- 10111010101110101101111011011110 [0xbabadede][3132808926]
# signal_MemRead- 0
# signal_MemToReg - 0
# signal_MemWrite - 1
# signal_RegWrite- 0
# signal_dataExt - 01
# signal_pickByte - 10
# signal_ImmExt - 0
# extended_imm - 00000000000000000000000000000001 [0x00000001[          1]]
# alu_result - 00000000000000000000000000001011 [0x0000000b][        11]
# result- 00000000000000000000000000001011 [        11][        11]
#
# opcode-101011 [0x2b][43]
# $rs-00011 [0x03][ 3]
# $rt-00010 [0x02][ 2]
# imm-0000000000000010 [0x0002][    2]
# write_data- 00000000000000000000000000001100 [0x0000000c][        12]
# read_data1- 00000000000000000000000000001010 [0x0000000a][        10]
# read_data2- 10111110111011111110111010101101 [0xbeefdead][3203391149]
# signal_MemRead- 0
# signal_MemToReg - 0
# signal_MemWrite - 1
# signal_RegWrite- 0
# signal_dataExt - 01
# signal_pickByte - 10
# signal_ImmExt - 0
# extended_imm - 00000000000000000000000000000010 [0x00000002[          2]]
# alu_result - 00000000000000000000000000001100 [0x0000000c][        12]
# result- 00000000000000000000000000001100 [        12][        12]
#
```

Store Half Word

Sh, 3, 6, 3 (M[$r3 + 3] = $r6[0:15])

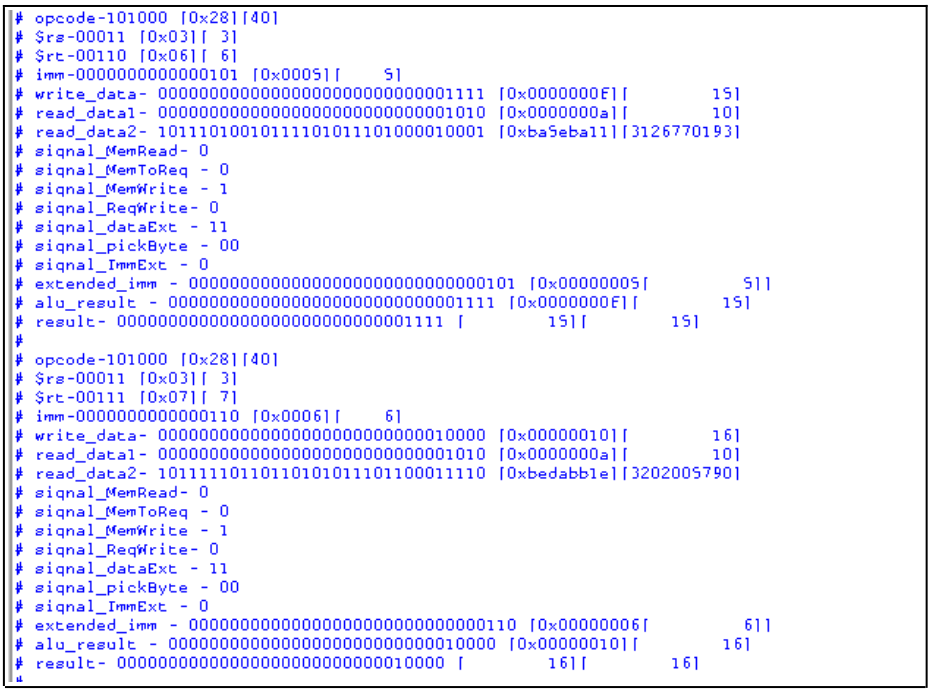
sh, 3, 7, 4 (M[$r3 + 4] = $r7[0:15])



Store Byte

Sb, 3, 6, 5 (M[$r3 + 5] = $r6[0:7])

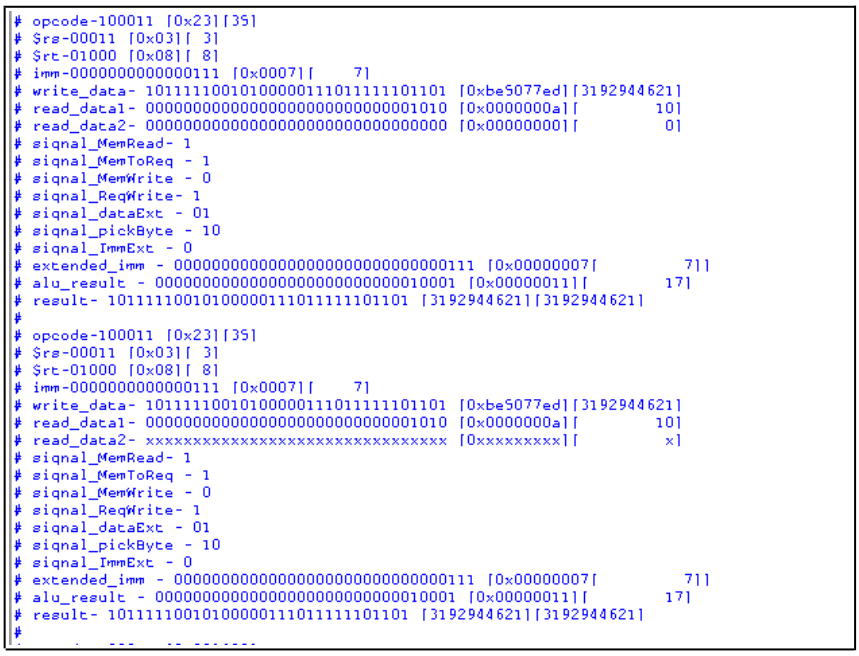
sb, 3, 7, 6 (M[$r3 + 6] = $r7[0:7])



Load Word

Lw, 3, 8, 7 ($r8 = M[$r3 + 7])

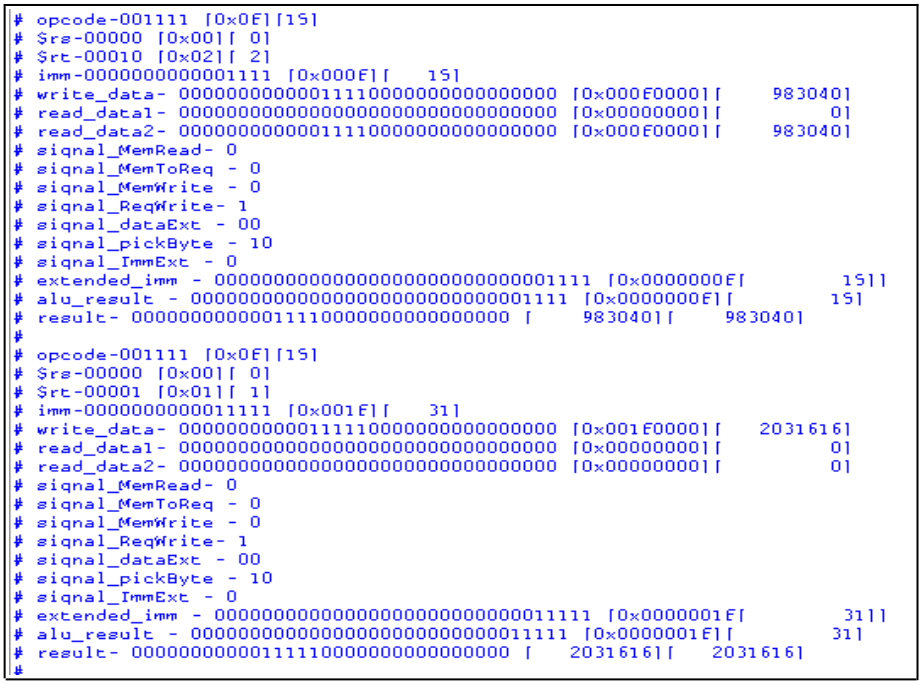
lw, 3, 9, 8 ($r9 = M[$r3 + 8])



Load Upper Imm

Lui , 0, 2, 15 ($r2 = 0xF0000)

Lui, 0, 1, 31 ($r1 = 0x1F0000)



Load Halfword Unsigned

Lhu, 20, 10, -1 ($r10 = M[$20 - 1])

Lhu, 20, 11, 1 ($r11 = M[$r20 + 1])

Load Byte Unsigned

Lbu 20, 12, -1 ($r12 = M[$r20 – 1])

lbu 20, 13, 1 ($r13 = M[$r20 + 1])

Load Halfword

Lh, 20, 14, 2 ($r14 = M[$20 + 2])

Lh, 20, 15, 1 ($r15 = M[$r20 + 1])

```
# opcode-100100 [0x24][36]
# $rs-10100 [0x14][20]
# $rt-01110 [0x0e][14]
# imm-0000000000000010 [0x0002][    2]
# write_data- 00000000000000000000000000010001 [0x00000011][        17]
# read_data1- 00000000000000000000000000010100 [0x00000014][        20]
# read_data2- 00000000000000000000000000011110 [0x0000001e][        30]
# signal_MemRead- 1
# signal_MemToReq - 1
# signal_MemWrite - 0
# signal_ReqWrite- 1
# signal_dataExt - 11
# signal_pickByte - 00
# signal_ImmExt - 0
# extended_imm - 00000000000000000000000000000010 [0x00000002[         2]]
# alu_result - 00000000000000000000000000010110 [0x00000016][        22]
# result- 00000000000000000000000000010001 [        17][        17]
#
# opcode-100100 [0x24][36]
# $rs-10100 [0x14][20]
# $rt-01111 [0x0f][15]
# imm-0000000000000001 [0x0001][    1]
# write_data- 00000000000000000000000000011110 [0x0000001e][        30]
# read_data1- 00000000000000000000000000010100 [0x00000014][        20]
# read_data2- 00000000000000000000000000000000 [0x00000000][         0]
# signal_MemRead- 1
# signal_MemToReq - 1
# signal_MemWrite - 0
# signal_ReqWrite- 1
# signal_dataExt - 11
# signal_pickByte - 00
# signal_ImmExt - 0
# extended_imm - 00000000000000000000000000000001 [0x00000001[         1]]
# alu_result - 00000000000000000000000000010101 [0x00000015][        21]
# result- 00000000000000000000000000011110 [        30][        30]
#
```

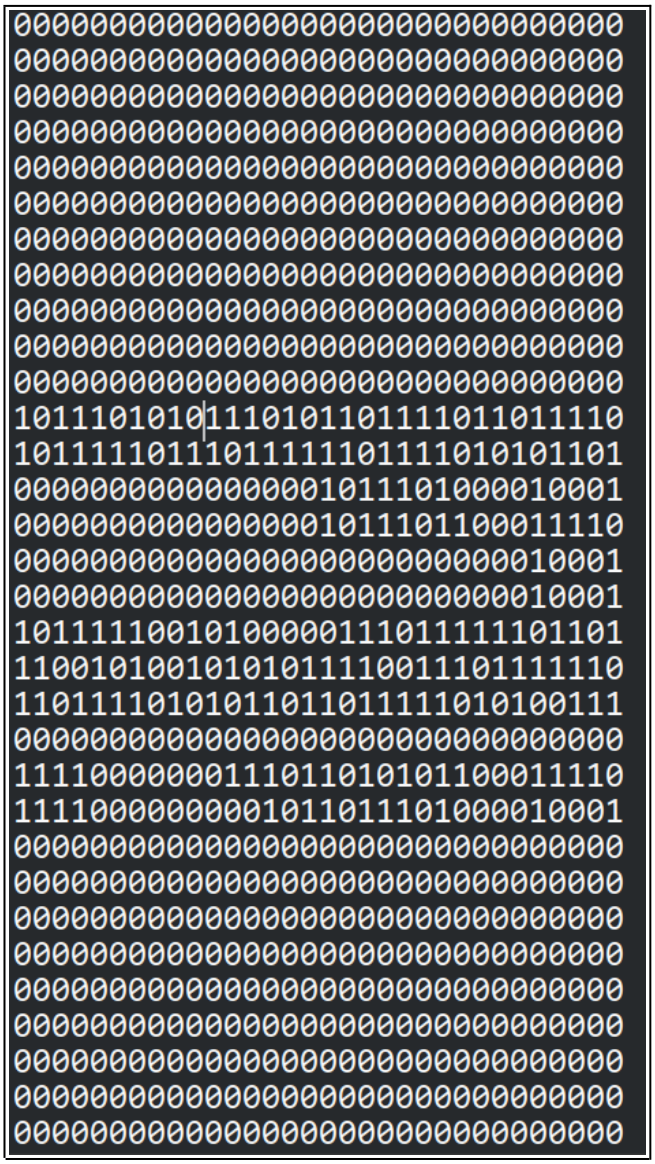
Load Byte

Lb, 20, 16, 2 ($r16 = M[$20 + 2])

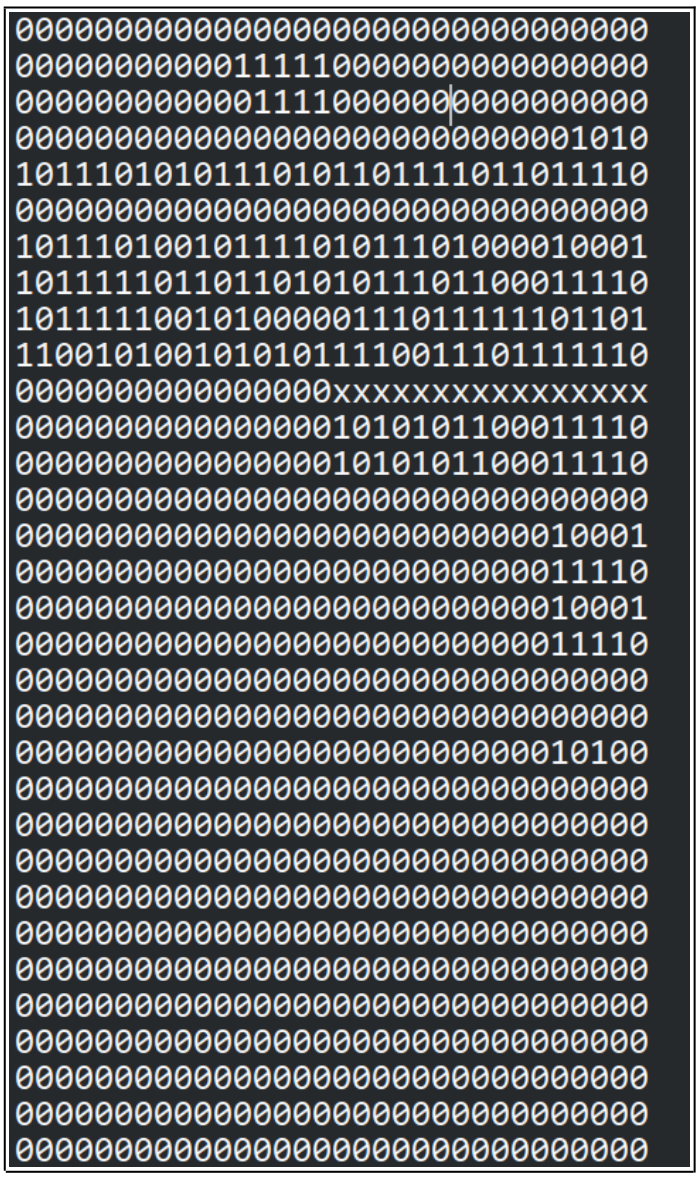
Lb, 20, 17, 1 ($r17 = M[$r20 + 1])

```
# opcode-100000 [0x20][32]
# $rs-10100 [0x14][20]
# $rt-10000 [0x10][16]
# imm-0000000000000010 [0x0002][    2]
# write_data- 00000000000000000000000000010001 [0x00000011][        17]
# read_data1- 00000000000000000000000000010100 [0x00000014][        20]
# read_data2- 00000000000000000000000000011110 [0x0000001e][        30]
# signal_MemRead- 1
# signal_MemToReq - 1
# signal_MemWrite - 0
# signal_ReqWrite- 1
# signal_dataExt - 11
# signal_pickByte - 00
# signal_ImmExt - 1
# extended_imm - 00000000000000000000000000000010 [0x00000002[         2]]
# alu_result - 00000000000000000000000000010110 [0x00000016][        22]
# result- 00000000000000000000000000010001 [        17][        17]
#
# opcode-100000 [0x20][32]
# $rs-10100 [0x14][20]
# $rt-10001 [0x11][17]
# imm-0000000000000001 [0x0001][    1]
# write_data- 00000000000000000000000000011110 [0x0000001e][        30]
# read_data1- 00000000000000000000000000010100 [0x00000014][        20]
# read_data2- 00000000000000000000000000000000 [0x00000000][         0]
# signal_MemRead- 1
# signal_MemToReq - 1
# signal_MemWrite - 0
# signal_ReqWrite- 1
# signal_dataExt - 11
# signal_pickByte - 00
# signal_ImmExt - 1
# extended_imm - 00000000000000000000000000000001 [0x00000001[         1]]
# alu_result - 00000000000000000000000000010101 [0x00000015][        21]
# result- 00000000000000000000000000011110 [        30][        30]
#
```

Memory After:



Register After:



Those instructions executed in order. Regarding to given order, you can compare the memories and check that they are working properly.

Small Modules

2X1 MUX (mux_2X1_testbench.v)

S0: B, S1:A

```
# a-0, b-0, s-0, y-0
# a-0, b-1, s-0, y-1
# a-1, b-0, s-0, y-0
# a-1, b-1, s-0, y-1
# a-0, b-0, s-1, y-0
# a-0, b-1, s-1, y-0
# a-1, b-0, s-1, y-1
# a-1, b-1, s-1, y-1
```

32 bit 2X1 MUX (mux_2X1_32bit_testbench.v)

S0: B, S1: A. As it can be seen from the picture, the select bit picks the correct 32 bit number:

```
# a-       123, b-        10, s-0, y-        10
# a-    817238, b-    989123, s-0, y-    989123
# a-       928, b-     19283, s-0, y-     19283
# a-        62, b-       109, s-0, y-       109
# a-    129389, b-      1235, s-1, y-    129389
# a-      1212, b-     12567, s-1, y-      1212
# a-      9512, b-      2396, s-1, y-      9512
# a-   2982943, b-    192895, s-1, y-   2982943
```

4X1 MUX (mux_4X1_testbench.v)

```
# time -  0, input - 0001, s0-0, s1-0, y-1
# time - 20, input - 0010, s0-0, s1-1, y-1
# time - 40, input - 0100, s0-1, s1-0, y-1
# time - 60, input - 1000, s0-1, s1-1, y-1
```

32 bit 4X1 MUX (mux_4X1_32bit_testbench.v)

```
# i0-      1234, i1-       567, i2-      7890, i3-     98786, s-00, y-      1234
# i0-     91238, i1-    293849, i2-      1255, i3-     11111, s-01, y-    293849
# i0-      2222, i1-      3333, i2-      4444, i3-      5555, s-10, y-      4444
# i0-      6666, i1-      7777, i2-       888, i3-      9991, s-11, y-      9991
```

16 bit Sign Extend (sign_extend_testbench.v)

Even though the Modelsim does not show the minus sign (-) before the first two inputs, as you can see from the binary representation, they are negative numbers, and with sign extension module they remain the same negative number.

```
# num-  1234, num_extended-       1234, num-1111101100101110, num_extended-11111111111111111111101100101110
# num-  1111, num_extended-       1111, num-1111101110101001, num_extended-11111111111111111111101110101001
# num-     1, num_extended-          1, num-0000000000000001, num_extended-00000000000000000000000000000001
# num-    82, num_extended-         82, num-0000000001010010, num_extended-00000000000000000000000001010010
```

16 bit Zero Extend (zero_extend_testbench.v)

First two numbers are negative.

```
# num-  1234, num_extended-      64302, num-1111101100101110, num_extended-00000000000000001111101100101110
# num-  1111, num_extended-      64425, num-1111101110101001, num_extended-00000000000000001111101110101001
# num-     1, num_extended-          1, num-0000000000000001, num_extended-00000000000000000000000000000001
# num-    82, num_extended-         82, num-0000000001010010, num_extended-00000000000000000000000001010010
```

24 bit Zero Extend (zero_extend_24bit_testbench.v)

First two numbers are negative.

```
# num-  12, num_extended-        244, num-11110100, num_extended-00000000000000000000000011110100
# num-   7, num_extended-        249, num-11111001, num_extended-00000000000000000000000011111001
# num-  99, num_extended-         99, num-01100011, num_extended-00000000000000000000000001100011
# num-  82, num_extended-         82, num-01010010, num_extended-00000000000000000000000001010010
```

End Zero Extend (end_zero_extend_testbench.v)

{num, 16b'0} = num_extended.

```
# num-    12, num_extended-     786432, num-1111111111110100, num_extended-11111111111101000000000000000000
# num-     7, num_extended-     458752, num-1111111111111001, num_extended-11111111111111001000000000000000
# num-  1123, num_extended-   73596928, num-0000010001100011, num_extended-00000100011000110000000000000000
# num-    82, num_extended-    5373952, num-0000000001010010, num_extended-00000000010100100000000000000000
```