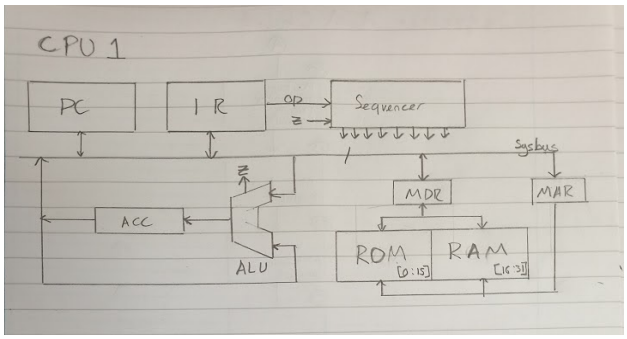
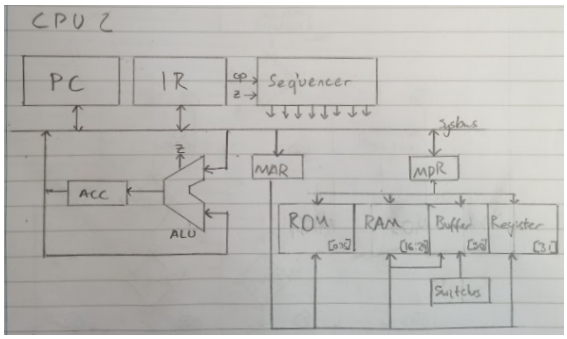
D2: Digital Systems and Microprocessor Design Exercise
https://stackoverflow.com/questions/16424726/what-is-the-difference-between-verilog-and

Understanding the Microprocessor - 23/01/21

Architecture:



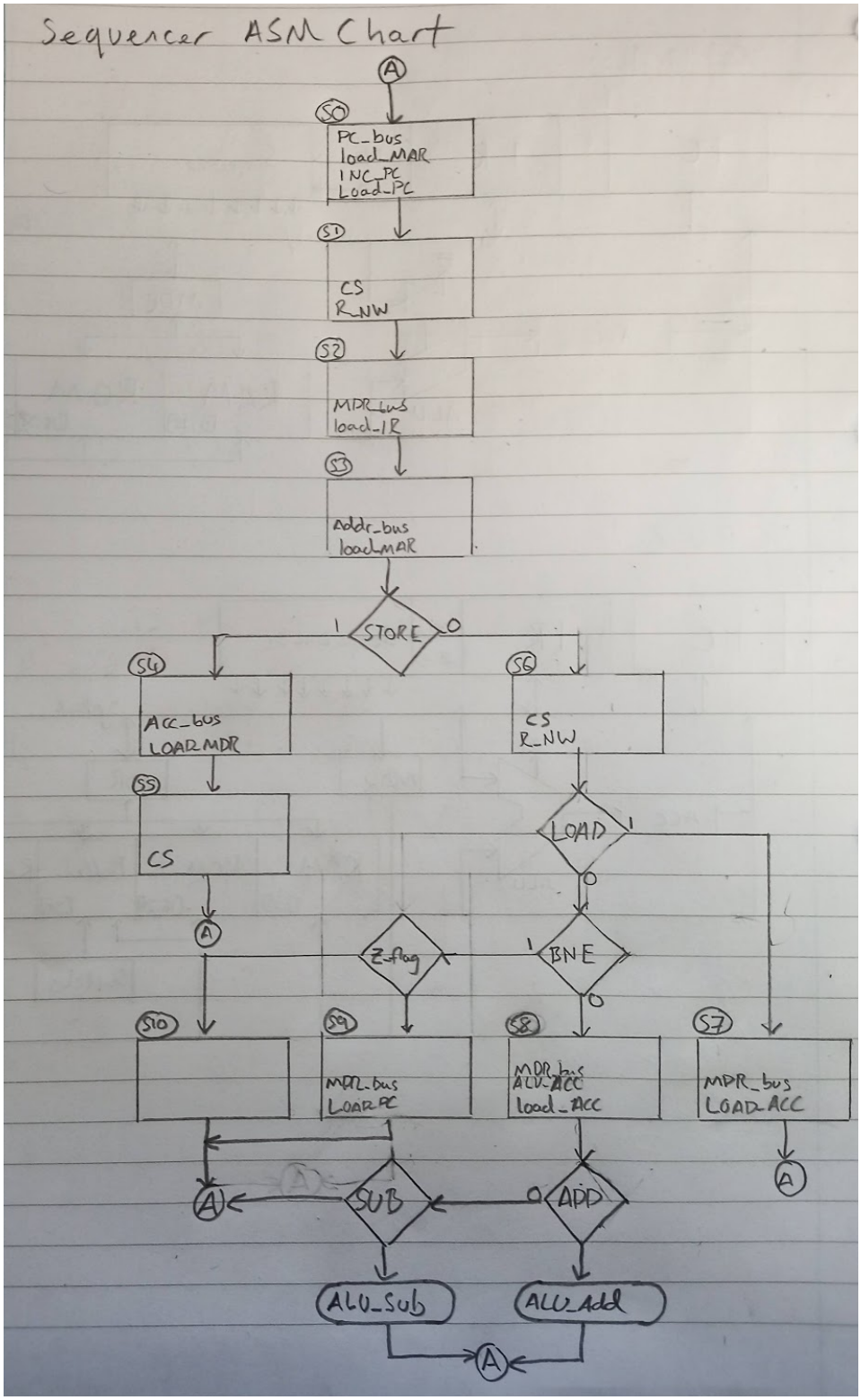


PC:
The program counter is a sequential logic module that records the position of the program through the ROM and outputs the position to the sysbus when the next instruction needs to be loaded.

IR:
The instruction register is a sequential logic module that reads the opcode from the sysbus and passes it to the sequencer.

Sequencer:
The sequencer is a state machine that controls the order of operation for the other modules. This is important to stop any bus contention occurring.

ACC_bus	Drive bus with contents of ACC (enable three-state output)
load_ACC	Load ACC from bus
PC_bus	Drive bus with contents of PC
load_IR	Load IR from bus
load_MAR	Load MAR from bus
MDR_bus	Drive bus with contents of MDR
load_MDR	Load MDR from bus
ALU_ACC	Load ACC with result from ALU
INC_PC	Increment PC and save the result in PC
Addr_bus	Drive bus with operand part of instruction held in IR
CS	Chip select. Use contents of MAR to set up memory address
R_NW	Read, not write. When false, contents of MDR are stored in memory
ALU_add	Perform an add operation in the ALU
ALU_sub	Perform a subtract operation in the ALU


Sequencer ASM Chart
A
S0
PC_bus
load_MAR
INC_PC
Load_PC
S1
CS
R_NW
S2
MDR_bus
load_IR
S3
Addr_bus
load_MAR
STORE
1
0
S4
ACC_bus
LOAD_MDR
S5
CS
R_NW
S6
CS
LOAD
1
0
A
Z_flag
BNE
1
0
S10
S9
MDR_bus
LOAD_PC
S8
MDR_bus
ALU_ACC
load_ACC
S7
MDR_bus
LOAD_ACC
A
A
SUB
0
ADD
ALU_Sub
ALU_Add
A

Basic Operation - 23/01/21

1) The file "rom.sv" contains a program. What does this program do? Where does it store data?

The program in the ROM is stored in the memory addresses 0 to 6. This program uses memory address 16 to store its working, this is the first address in RAM.

2) What happens at address 0 and why? Where does the program loop back to?

At address 0, in the ROM, the program stores the current value in the accumulator (inside the ALU) to address 16, in the RAM.

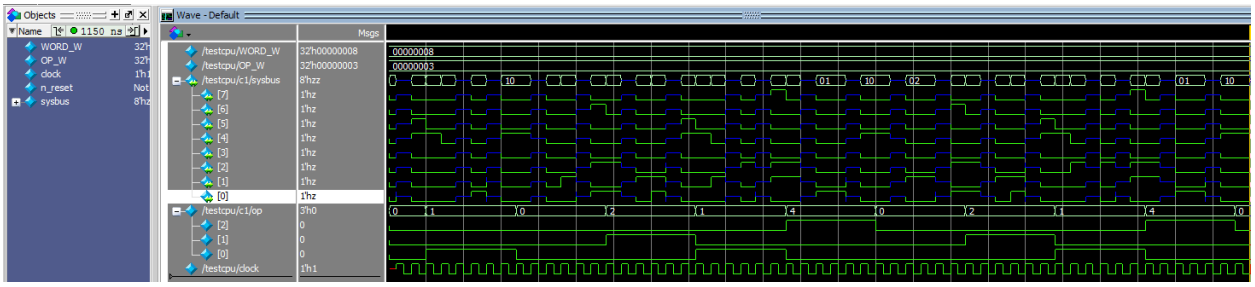
This program will loop back to address one because the BNE opcode sets the mdr to address one and the sequencer to load the program counter from the mdr.

3) The memory address registers in the ram and rom have 5 bits. How are the ram and rom organised to ensure that they do not overlap? How are the addresses decoded?

ROM covers the memory addresses 0 to 15.
RAM covers the memory addresses 16 to 31 (or 16 to 29 for CPU 2).

The three most significant bits of the sysbus is the opcode and so is ignored when looking for the address. The five least significant bits of the sysbus is the mar, which is a five bit address stored in binary.The ROM can only access the sysbus when the value of the most significant bit of the mar is 0.The RAM can only access the sysbus when the value of the most significant bit of the mar is 1. This stops bus contention.

4) How is the high impedance, 'Z', state of sysbus shown in the simulation?



In ModelSim the high impedance state is displayed as a blue line between the two logic levels.

5) SystemVerilog and ModelSim can represent ‘Z’ and ‘X’ states, but are these states “real”? In other words, is it possible to design logic that would detect ‘Z’ or ‘X’ states and display them?

‘Z’ and ‘X’ states are not “real” states as they cannot be verified with only voltage readings, which is all a digital system is capable of doing. However, it is possible to create logic that would identify ‘Z’ or ‘X’ states within itself as it would know at any point in its function when nothing is writing to the bus or if there is bus contention.

Memory Mapped Input and Output - 23/01/21

1) Sketch a diagram of the memory, going from address 0 to address 31, and showing which components are at each address.

Address	Function
[0:15]	ROM
[16:29]	RAM
[30]	Buffer
[31]	Register

2) Add the appropriate lines to cpu2 to instantiate one buffer and one register.

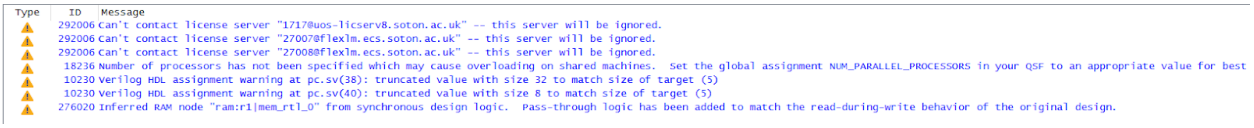
```
//instantiate 1 register and 1 buffer here

buffer #(.WORD_W(WORD_W), .OP_W(OP_W)) b1  (.*);
sequencer #(.WORD_W(WORD_W), .OP_W(OP_W)) s1  (.*);
ir #(.WORD_W(WORD_W), .OP_W(OP_W)) i1  (.*);
pc #(.WORD_W(WORD_W), .OP_W(OP_W)) p1 (.*);
alu #(.WORD_W(WORD_W), .OP_W(OP_W)) a1 (.*);
ram #(.WORD_W(WORD_W), .OP_W(OP_W)) r1 (.*);
rom #(.WORD_W(WORD_W), .OP_W(OP_W)) r2 (.*);
register #(.WORD_W(WORD_W), .OP_W(OP_W)) r3 (.*);
```

3) If the CPU attempts to read from address 30, both the buffer and the RAM will be enabled. Why is this going to be a problem?

This is a problem as both RAM and the buffer will assert their values onto the sysbus as they are both being read. The two values on the same bus will interfere with each other giving a signal that is likely neither of them. This is bus contention.

4) Modify this line so that mdr is not written to the bus when mar contains addresses 30 or 31. The RAM should continue to work correctly for other addresses.

```
//The following line in the ram module copies the contents of mdr to sysbus:
assign sysbus = (MDR_bus & mar[WORD_W-OP_W-1]
                  & ~(mar==30 | mar==31)) ? mdr : 'z;
```



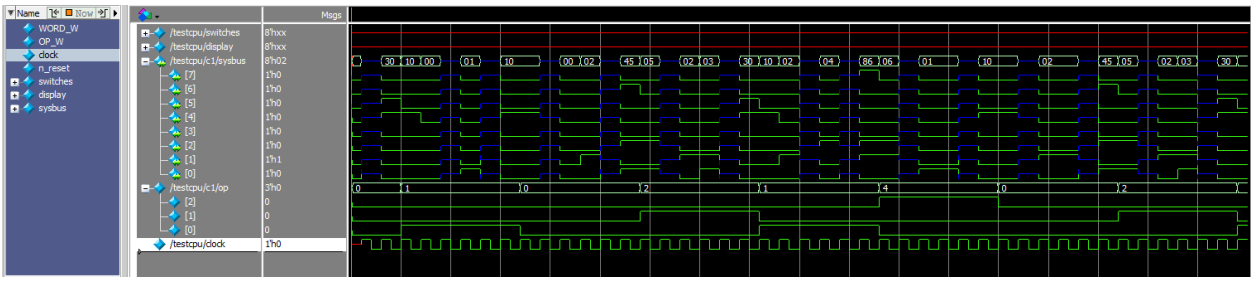
Mark Zwolinski
Sun 24/01/2021 16:11
To: butterworth j.d. (jdb1g20)

```
Logfile for user jdb1g20 attempt 5; ram.sv
Starting ModelSim
Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct  5 2016
Start time: 16:11:40 on Jan 24,2021
vlog -work jdb1g20work -sv ram.sv
-- Compiling module ram

Top level modules:
       ram
End time: 16:11:40 on Jan 24,2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
# vsim -L jdb1g20work -c -novopt -do "../do_vsim" ../work.testram
# Start time: 16:11:41 on Jan 24,2021
# Loading sv_std.std
# Loading ../work.testram
# Loading jdb1g20work.ram
# do ../do_vsim
# ---------------------------
# Modelsim Simulation
# ---------------------------
#
# RAM address        16 OK
# RAM address        17 OK
# RAM address        18 OK
# RAM address        19 OK
# RAM address        20 OK
# RAM address        21 OK
# RAM address        22 OK
# RAM address        23 OK
# RAM address        24 OK
# RAM address        25 OK
# RAM address        26 OK
# RAM address        27 OK
# RAM address        28 OK
# RAM address        29 OK
# RAM address        30 OK
# RAM address        31 OK
#
# ---------------------------
# End time: 16:11:41 on Jan 24,2021, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
```

5) Modify “testcpu” so that it uses cpu2 instead of cpu1. You will also need to include switches and display as variables in the testbench.

```
module testcpu;

parameter int WORD_W = 8, OP_W = 3;

logic  clock, n_reset;
logic [WORD_W-1:0] switches;
logic [WORD_W-1:0] display;
wire [WORD_W-1:0] sysbus;

cpu2 #(.WORD_W(WORD_W), .OP_W(OP_W)) c1 (.*);
```

6) What do you see in the Waveform window for display and switches?

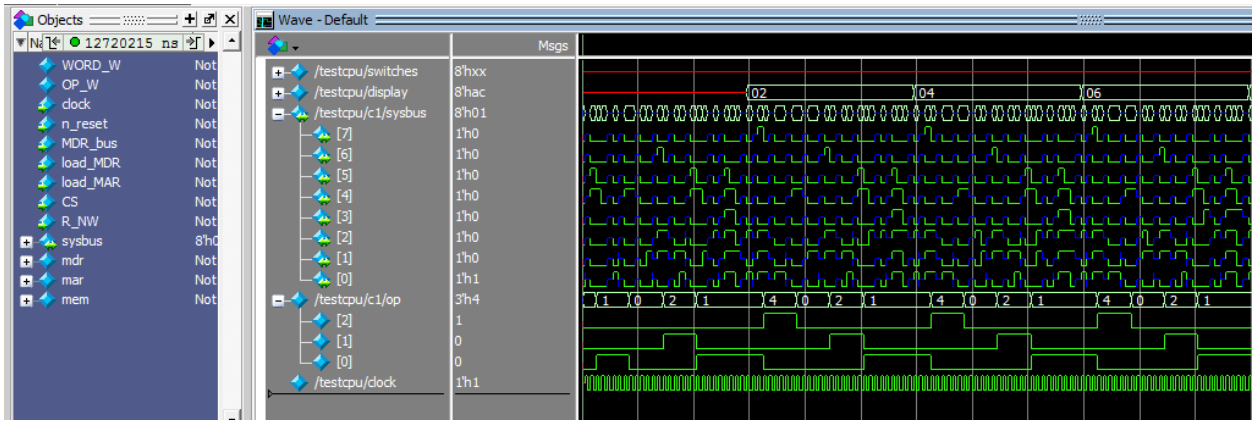


Both the display register and the switches are of unknown value as they are not defined at any point in the testbench or the program.

7) Modify the program in the rom module such that the result in the accumulator is STOREd to address 31 (display) as well as address 16. Note that you will have to move the contents of the ROM at addresses 4 to 6 up by one place and modify the ADD and BNE instructions. What do you now see in the simulation?

```
    0: mdr = {`STORE, 5'd16}; //Store the contents of accumulator
    1: mdr = {`LOAD, 5'd16};  //Load the contents of address into the
accumulator
    2: mdr = {`ADD, 5'd6};    //Add the contents of address to the accumulator
    3: mdr = {`STORE, 5'd16}; //Store the contents of accumulator
    4: mdr = {`STORE, 5'd31}; //Store the contents of accumulator
    5: mdr = {`BNE, 5'd7};    //Branch if result of last arithmetic operation
is not zero
    6: mdr = 2;               //contents used by another instruction
    7: mdr = 1;               //contents used by another instruction
```

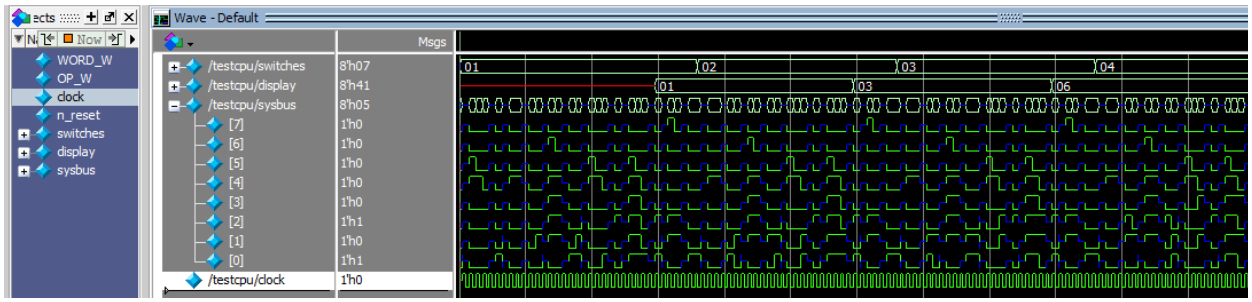
My modified program would output the value to the display register. This clearly shows the program counts up in twos.



8) Make a second change to the program in the rom module such that the data for the ADD operation comes from address 30 – the switches. Modify “testcpu” to apply different values to the switches during the simulation. What do you now see in the simulation?

```
    0: mdr = {`STORE, 5'd16}; //Store the contents of accumulator
    1: mdr = {`LOAD, 5'd16};  //Load the contents of address into the
accumulator
    2: mdr = {`ADD, 5'd30};    //Add the contents of address to the accumulator
    3: mdr = {`STORE, 5'd16}; //Store the contents of accumulator
    4: mdr = {`STORE, 5'd31}; //Store the contents of accumulator
    5: mdr = {`BNE, 5'd6};    //Branch if result of last arithmetic operation
is not zero
    6: mdr = 1;               //contents used by another instruction
```

My new program would take the values from the switches and add it to the current value, outputting it to the display register.



Extending the Microprocessor - 24/01/21

Creating a new version of the CPU, cpu3, I added the bitwise xor and bitwise not (complement) functions to act on the accumulator. This meant defining the opcodes, adding new enable lines between the ALU and sequencer, adding the functionality to the ALU and Sequencer and writing a new program in ROM to use these commands.

ALU:

```
    if (load_ACC)
      if (ALU_ACC)
        begin
        if (ALU_add)
          acc <= acc + sysbus;
        else if (ALU_sub)
          acc <= acc - sysbus;
        else if (ALU_xor)
          acc <= acc ^ sysbus;
        else if (ALU_comp)
          acc <= ~acc;
        end
      else
        acc <= sysbus;
```

Sequencer:

```
 s8: begin
       MDR_bus = 1'b1;
       ALU_ACC = 1'b1;
       load_ACC = 1'b1;
       if (op == `ADD)
         ALU_add = 1'b1;
       else if (op == `SUB)
         ALU_sub = 1'b1;
       else if (op == `XOR)
         ALU_xor = 1'b1;
       else if (op == `COMP)
         ALU_comp = 1'b1;
     Next_State = s0;
       end
```

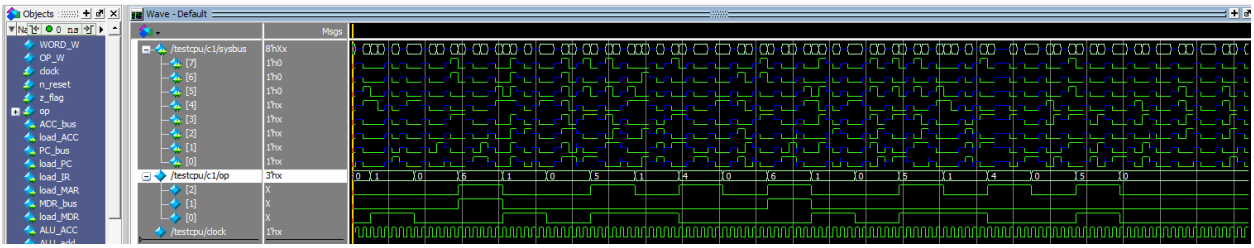
CPU:

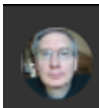
```
module cpu1 #(parameter WORD_W = 8, OP_W = 3)
                  (input logic clock, n_reset,
                   inout wire [WORD_W-1:0] sysbus);

logic ACC_bus, load_ACC, PC_bus, load_PC, load_IR, load_MAR,
MDR_bus, load_MDR, ALU_ACC, ALU_add, ALU_sub, ALU_xor, ALU_comp,
INC_PC, Addr_bus, CS, R_NW, z_flag;
```

ROM:

```
  case (mar)
    0:  mdr = {`STORE,  5'd16}; //Store the contents of accumulator
    1:  mdr = {`LOAD,   5'd16}; //Load the contents of address into the
accumulator
    2:  mdr = {`COMP,   5'd8};  //Complement the contents of the accumulator
    3:  mdr = {`STORE,  5'd16}; //Store the contents of accumulator
    4:  mdr = {`LOAD,   5'd16}; //Load the contents of address into the
accumulator
    5:  mdr = {`XOR,    5'd9};  //Xor the contents of address with the
accumulator
    6:  mdr = {`STORE,  5'd16}; //Store the contents of accumulator
    7:  mdr = {`BNE,    5'd10}; //Branch if result of last arithmetic operation
is not zero
    8:  mdr = 1;                //contents used by another instruction
    9:  mdr = 170;              //contents used by another instruction
    10: mdr = 1;                //contents used by another instruction
    default: mdr = 0;           //rest of ROM is 0
  endcase
```

I then verified that the program was working correctly by simulating it in modelsim and going through each operation. I found that by the end of the first cycle the value in ram was 1'h55, which was to be expected when 1'hFF and 1'hAA are XORed together.





Mark Zwolinski
Sun 24/01/2021 15:02
To: butterworth j.d. (jdb1g20)

```
Logfile for user jdb1g20 attempt 1; alu.sv, sequencer.sv
Starting ModelSim
Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct  5 2016
Start time: 15:02:00 on Jan 24,2021
vlog -work jdb1g20work -sv alu.sv
-- Compiling module alu

Top level modules:
        alu
End time: 15:02:00 on Jan 24,2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct  5 2016
Start time: 15:02:00 on Jan 24,2021
vlog -work jdb1g20work -sv sequencer.sv
-- Compiling module sequencer

Top level modules:
        sequencer
End time: 15:02:00 on Jan 24,2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
# vsim -L jdb1g20work -c -novopt -do "../do_vsim" ../work.testALU_seq
# Start time: 15:02:01 on Jan 24,2021
# Loading sv_std.std
# Loading ../work.testALU_seq
# Loading jdb1g20work.sequencer
# Loading jdb1g20work.alu
# do ../do_vsim
# ---------------------------
# Modelsim Simulation
# ---------------------------
#
# XOR operation works correctly
#
# ---------------------------
# End time: 15:02:01 on Jan 24,2021, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
================================================================
Starting Verilator
================================================================
Starting Quartus
```

Encryption and Decryption - 25/01/21

l	0	1	1	0	0
k	0	1	0	1	1
output	0	0	1	1	1
output	0	0	1	1	1
l	0	1	1	0	0
key	0	1	0	1	1

To decrypt a XOR encrypted message the decryption is to XOR the encrypted message with the key again.

As the key is unknown to us and the number of possible keys (32) is small it is easy to brute force the solution by testing every possible key.

To test the output when the string “oiytmmvk” is decrypted with every possible key I wrote a program in ROM that would load in each value from the switches into a position in the RAM. It would then XOR each of these with a key outputting it to the display. Finally it would increment the key by one and repeat. This program meant I also had to modify the testbench so that it would input the value wanted.

As the length of my program was now longer than even the 32 total addresses in the 8 bit system, I had to increase the system size to a ten bit system. This provided me with 128 total addresses for my program to be stored.

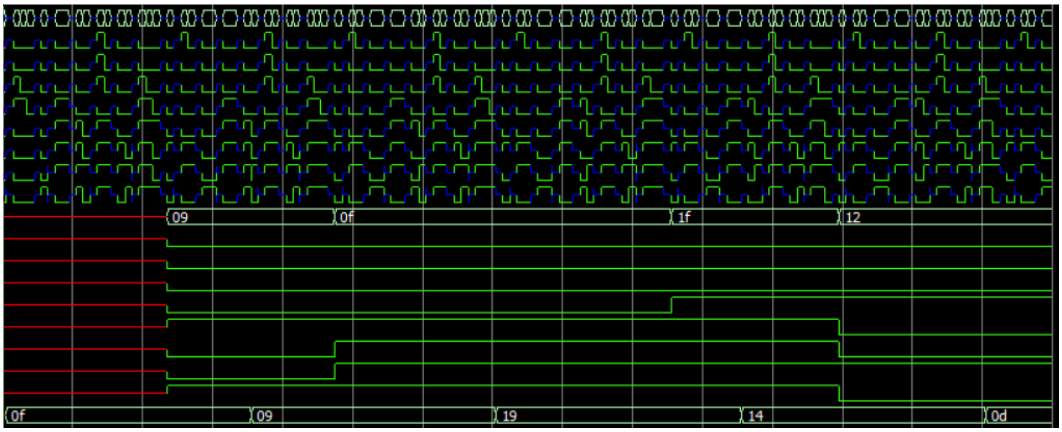
Updating to 10 bit from 8 bit meant that I had increase the word width so that the system could target each address. I did this by changing the parameter in each file. I also moved the addresses of the the tri-state buffer and the display registers, to positions 126 and 127, so that they weren’t taking ROM slots part way through the program.

I found that I did not need the subtract operation or the complement operation (that I implemented). However, as the total number of operations was still five, requiring 3 bits to encode them still, I did not see the value in removing them from my design.

```
  case (mar)
    0:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    1:  mdr = {`STORE,  7'd64};   //Store the contents of accumulator
    2:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    3:  mdr = {`STORE,  7'd65};   //Store the contents of accumulator
    4:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    5:  mdr = {`STORE,  7'd66};   //Store the contents of accumulator
    6:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    7:  mdr = {`STORE,  7'd67};   //Store the contents of accumulator
    8:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    9:  mdr = {`STORE,  7'd68};   //Store the contents of accumulator
    10: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    11: mdr = {`STORE,  7'd69};   //Store the contents of accumulator
    12: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    13: mdr = {`STORE,  7'd70};   //Store the contents of accumulator
    14: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    15: mdr = {`STORE,  7'd71};   //Store the contents of accumulator
    16: mdr = {`BNE,    7'd52};

    22: mdr = {`LOAD,   7'd64};   //Load the character 1 into the accumulator
    23: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    24: mdr = {`STORE,  7'd127};  //Output to display
    25: mdr = {`LOAD,   7'd65};   //Load the character 2 into the accumulator
    26: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    27: mdr = {`STORE,  7'd127};  //Output to display
    28: mdr = {`LOAD,   7'd66};   //Load the character 3 into the accumulator
    29: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    30: mdr = {`STORE,  7'd127};  //Output to display
    31: mdr = {`LOAD,   7'd67};   //Load the character 4 into the accumulator
    32: mdr = {`XOR,    7'd80};   //XOR accumulator with key
```

```
    33: mdr = {`STORE,  7'd127};  //Output to display
    34: mdr = {`LOAD,   7'd68};   //Load the character 5 into the accumulator
    35: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    36: mdr = {`STORE,  7'd127};  //Output to display
    37: mdr = {`LOAD,   7'd69};   //Load the character 6 into the accumulator
    38: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    39: mdr = {`STORE,  7'd127};  //Output to display
    40: mdr = {`LOAD,   7'd70};   //Load the character 7 into the accumulator
    41: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    42: mdr = {`STORE,  7'd127};  //Output to display
    43: mdr = {`LOAD,   7'd71};   //Load the character 8 into the accumulator
    44: mdr = {`XOR,    7'd80};   //XOR accumulator with key
    45: mdr = {`STORE,  7'd127};  //Output to display
    46: mdr = {`LOAD,   7'd80};   //Load key
    47: mdr = {`ADD,    7'd53};   //Add 1
    48: mdr = {`STORE,  7'd80};   //Store key
    49: mdr = {`XOR,    7'd54};
    50: mdr = {`BNE,    7'd55};   //Loop until all keys are tried

    52: mdr = 22;
    53: mdr = 1;
    54: mdr = 8;
    55: mdr = 16;
        /*RAM:
64: encrypted character 1
65: encrypted character 2
66: encrypted character 3
67: encrypted character 4
68: encrypted character 5
69: encrypted character 6
70: encrypted character 7
71: encrypted character 8
80: key
*/
    default: mdr = 0;             //rest of ROM is 0
  endcase
  end
```

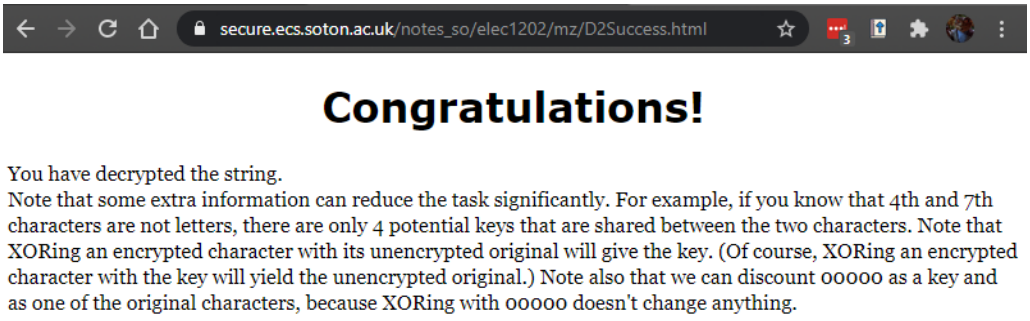
This is the start of the program in modelSIM as the value on the switches is loaded in.



Brute forcing through all 32 combinations gives the solutions:

0	OIYTMMVK	11	DBR FF]@	22	Y OB[[@]
1	NHXULLWJ	12	CEUXAAZG	23	X.NCZZA\
2	MK[VOOTI	13	BDTY@@[F	24	WQALUUNS
3	LJZWNNUH	14	AGWZCCXE	25	VP@MTTOR
4	KM]PIIRO	15	@FV[BBYD	26	USCNWWLQ
5	JL/QHHSN	16	YID]]F[	27	TRBOVVMP
6	IO RKKPM	17	.XHE//GZ	28	SUEHQQJW
7	HN.SJJQL	18	][KF DY	29	RTDIPPKV
8	GAQ/EE.C	19	/ZJG..EX	30	QWGJSSHU
9	F@P]DD B	20	[]M@YYB	31	PVFKRRIT
10	ECS.GG/A	21	Z/LAXXC.		

The tenth key is a link to the ecs intranet and provides the webpage below:



Encryption and Decryption with loops - 25/01/21

Before my invigilator made it clear that a brute force attack was sufficient. I began to try and make a program that could operate loops in such a way that it could change the target address.

It would do this by loading a “function” into memory it could then update the address by overwriting later lines with commands dependent on the outcome from the operation.

This method required the entire program to be written in ROM and the space for the largest function and the auto loader to be in the RAM. This meant that this design still required more address than the eight bits would allow.

I abandoned this method upon understanding that brute force was sufficient. By this point, I had written in pseudo code an auto-loader to load the program. A function to load the inputs and the brute force method with a loop to target different addresses.

```
//auto Loader
90: mdr = {`LOAD,   (`LOAD position_rom_start)};
91: mdr = {`ADD,    load_position};
92: mdr = {`STORE,  96}; //update next line
93: mdr = {`LOAD,   (`STORE position_ram_start)};
94: mdr = {`ADD,    position_load};
95: mdr = {`STORE,  97}; //update next line
96: mdr = {`LOAD,   program_in};
97: mdr = {`STORE,  program_out};
98: mdr = {`BNE,    90};

//input
100: mdr = {`LOAD,   switches};
101: mdr = {`STORE,  ACC_TEMP};
102: mdr = {`LOAD,   (`STORE position_character_start)};
103: mdr = {`ADD,    character_position};
104: mdr = {`STORE,   106};
105: mdr = {`LOAD,    ACC_TEMP};
106: mdr = {`STORE,  position};
107: mdr = {`BNE,   100};
```

```
//encryption-decryption
100: mdr = {`LOAD,   (`LOAD character_in_start)};
101: mdr = {`ADD,    position};
102: mdr = {`STORE,  106}; //update next line
103: mdr = {`LOAD,   (`STORE character_out_start)};
104: mdr = {`ADD,    position};
105: mdr = {`STORE,  108}; //update next line
106: mdr = {`LOAD,   character_in};
107: mdr = {`XOR,    key};
108: mdr = {`STORE,  character_out};
109: mdr = {`STORE,  Display};
110: mdr = {`LOAD,   position};
111: mdr = {`ADD,    1};
112: mdr = {`STORE,  position};
113: mdr = {`XOR,    length};
114: mdr = {`BNE,    100};
115: mdr = {`LOAD,   key};
116: mdr = {`ADD,    1};
117: mdr = {`STORE,  key};
118: mdr = {`XOR,    possible_keys};
119: mdr = {`BNE,    100};
```

XOR Then XNOR Encryption - 26/01/21

I changed the complement into a XNOR function.

```
always_ff @(posedge clock, negedge n_reset)
  begin
  if (!n_reset)
    acc <= 0;
  else
    if (load_ACC)
      if (ALU_ACC)
        begin
        if (ALU_add)
          acc <= acc + sysbus;
        else if (ALU_sub)
          acc <= acc - sysbus;
        else if (ALU_xor)
          acc <= acc ^ sysbus;
        else if (ALU_xnor)
          acc <= ~(acc ^ sysbus);
        end
      else
        acc <= sysbus;
  end
endmodule
```

The XNOR function will add further complexity to the encryption as it adds a second key on top of the existing function.

To decrypt this some one will have to know both keys and the order they were applied in. This increases the number of possible sets of keys to 2048. This is still manageable to brute force but will take orders of magnitude longer.

```
    case (mar)
    0:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    1:  mdr = {`STORE,  7'd64};   //Store the contents of accumulator
    2:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    3:  mdr = {`STORE,  7'd65};   //Store the contents of accumulator
    4:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    5:  mdr = {`STORE,  7'd66};   //Store the contents of accumulator
    6:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    7:  mdr = {`STORE,  7'd67};   //Store the contents of accumulator
    8:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    9:  mdr = {`STORE,  7'd68};   //Store the contents of accumulator
    10: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    11: mdr = {`STORE,  7'd69};   //Store the contents of accumulator
    12: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    13: mdr = {`STORE,  7'd70};   //Store the contents of accumulator
    14: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    15: mdr = {`STORE,  7'd71};   //Store the contents of accumulator

    16: mdr = {`LOAD,   7'd64};   //Load the character 1 into the accumulator
    17: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    18: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    19: mdr = {`STORE,  7'd127};  //Output to display
    20: mdr = {`LOAD,   7'd65};   //Load the character 2 into the accumulator
    21: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    22: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    23: mdr = {`STORE,  7'd127};  //Output to display
    24: mdr = {`LOAD,   7'd66};   //Load the character 3 into the accumulator
    25: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    26: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
```

```
    27: mdr = {`STORE,  7'd127};  //Output to display
    28: mdr = {`LOAD,   7'd67};   //Load the character 4 into the accumulator
    29: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    30: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    31: mdr = {`STORE,  7'd127};  //Output to display
    32: mdr = {`LOAD,   7'd68};   //Load the character 5 into the accumulator
    33: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    34: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    35: mdr = {`STORE,  7'd127};  //Output to display
    36: mdr = {`LOAD,   7'd69};   //Load the character 6 into the accumulator
    37: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    38: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    39: mdr = {`STORE,  7'd127};  //Output to display
    40: mdr = {`LOAD,   7'd70};   //Load the character 7 into the accumulator
    41: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    42: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    43: mdr = {`STORE,  7'd127};  //Output to display
    44: mdr = {`LOAD,   7'd71};   //Load the character 8 into the accumulator
    45: mdr = {`XOR,    7'd51};   //XOR accumulator with XOR key
    46: mdr = {`XNOR,   7'd52};   //XNOR accumulator with XNOR key
    47: mdr = {`STORE,  7'd127};  //Output to display

    51: mdr = 21; //XOR Key
    52: mdr = 10; //XNOR key
    53: mdr = 1;
    54: mdr = 8;
    55: mdr = 16;
    default: mdr = 0;             //rest of ROM is 0
  endcase
  end
```

XOR With Multiple Keys Encryption - 26/01/21

Finally I implemented a new program that would encrypt or decrypt the message using a key of multiple bytes. This program would read in both the string and a key, of length 3.

To brute force a key length of three it would have to try 32768 possible combinations of keys. However, if the length of the key is unknown it becomes exponentially harder to brute force the attack.

This method is scalable with different key lengths but can still be decrypted as long as the length of the key is less than the length of the encrypted string.

```
  case (mar)
    0:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    1:  mdr = {`STORE,  7'd64};   //Store the contents of accumulator
    2:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    3:  mdr = {`STORE,  7'd65};   //Store the contents of accumulator
    4:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    5:  mdr = {`STORE,  7'd66};   //Store the contents of accumulator
    6:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    7:  mdr = {`STORE,  7'd67};   //Store the contents of accumulator
    8:  mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    9:  mdr = {`STORE,  7'd68};   //Store the contents of accumulator
    10: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    11: mdr = {`STORE,  7'd69};   //Store the contents of accumulator
    12: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    13: mdr = {`STORE,  7'd70};   //Store the contents of accumulator
    14: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    15: mdr = {`STORE,  7'd71};   //Store the contents of accumulator
    16: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    17: mdr = {`STORE,  7'd72};   //Store the contents of accumulator
```

```
    18: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    19: mdr = {`STORE,  7'd73};   //Store the contents of accumulator
    20: mdr = {`LOAD,   7'd126};  //Load the contents of address into the
accumulator
    21: mdr = {`STORE,  7'd74};   //Store the contents of accumulator

    22: mdr = {`LOAD,   7'd64};   //Load the character 1 into the accumulator
    23: mdr = {`XOR,    7'd72};   //XOR accumulator with key
    24: mdr = {`STORE,  7'd127};  //Output to display
    25: mdr = {`LOAD,   7'd65};   //Load the character 2 into the accumulator
    26: mdr = {`XOR,    7'd73};   //XOR accumulator with key
    27: mdr = {`STORE,  7'd127};  //Output to display
    28: mdr = {`LOAD,   7'd66};   //Load the character 3 into the accumulator
    29: mdr = {`XOR,    7'd74};   //XOR accumulator with key
    30: mdr = {`STORE,  7'd127};  //Output to display
    31: mdr = {`LOAD,   7'd67};   //Load the character 4 into the accumulator
    32: mdr = {`XOR,    7'd72};   //XOR accumulator with key
    33: mdr = {`STORE,  7'd127};  //Output to display
    34: mdr = {`LOAD,   7'd68};   //Load the character 5 into the accumulator
    35: mdr = {`XOR,    7'd73};   //XOR accumulator with key
    36: mdr = {`STORE,  7'd127};  //Output to display
    37: mdr = {`LOAD,   7'd69};   //Load the character 6 into the accumulator
    38: mdr = {`XOR,    7'd74};   //XOR accumulator with key
    39: mdr = {`STORE,  7'd127};  //Output to display
    40: mdr = {`LOAD,   7'd70};   //Load the character 7 into the accumulator
    41: mdr = {`XOR,    7'd72};   //XOR accumulator with key
    42: mdr = {`STORE,  7'd127};  //Output to display
    43: mdr = {`LOAD,   7'd71};   //Load the character 8 into the accumulator
    44: mdr = {`XOR,    7'd73};   //XOR accumulator with key
    45: mdr = {`STORE,  7'd127};  //Output to display
    46: mdr = {`BNE,    7'd55};   //Loop until all keys are tried

    53: mdr = 1;
    54: mdr = 8;
    55: mdr = 22;

    /*RAM:
```

```
64: encrypted character 1
65: encrypted character 2
66: encrypted character 3
67: encrypted character 4
68: encrypted character 5
69: encrypted character 6
70: encrypted character 7
71: encrypted character 8
72: key 1
73: key 2
74: key 3
*/
    default: mdr = 0;              //rest of ROM is 0
  endcase
  end
```