ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

Computer Organization Project Report

Project NO	:	2
Due Date	:	24.05.2023
GROUP NO	:	G37

GROUP MEMBERS:

150200075	:	Matay AYDIN
150200096	:	Mustafa KIRCI (Group Representative)
150200916	:	Denıs Iurıe DAVIDOGLU

SPRING 2023

Contents

1 Introduction

Computers can be different by their architecture, instruction sets, peripherals and performance. Although it seems like every architectures are doing the same thing, the way an operation executes could vary significantly under the hood. Some architectures could have limited number of instructions and others may have limited number of registers.

This project aims to implement a hardwired control unit and 16 instructions for CPU system using Verilog HDL. Control unit, as the name suggest, will modify control inputs of the ALU system according to the instructions it reads from the memory. The ALU system from project 1 will be used for arithmetic logic operations in this project.

2 Project

2.1 Design

CPU system is simply made out of 2 parts. Those are control unit and ALU system. Since ALU system implementation details are out of scope for this project, only control unit will be explained in detail.
Control unit is works in basically 2 parts. Those are fetch decode cycle and instruction cycle. Fetch and decode cycles have fixed number of clock cycles but number instruction cycles depend on the type of the instruction. Every cycle is explained in great detail below.

2.2 Fetch Decode

T[0]: IR(0-7) ← M[PC]

- Instruction register is enabled and its function is set to LOAD.
- Lower part(0-7) of the instruction register is selected.
- Memory is enabled and memory mode is set to read.
- Program counter is connected to memory address input.
- Only program counter and past program counter are enabled in address register file.
- Address register file function is set to INCREMENT.
- Every register in register file is disabled.
- Sequence counter function is set to INCREMENT.

T[1]: IR(8-15) ← M[PC]

- Instruction register is enabled and its function is set to LOAD.
- Upper part(8-15) of the instruction register is selected.
- Memory is enabled and memory mode is set to read.
- Program counter is connected to memory address input.
- Only program counter and past program counter are enabled in address register file.
- Address register file function is set to INCREMENT.
- Every register in register file is disabled.
- Sequence counter function is set to INCREMENT.

T[2]

- Memory is disabled
- Sequence counter function is set to INCREMENT.
- Every register in the CPU except sequence counter register is disabled.

2.3 Instructions

Every instruction is implemented into an if block and that if block will execute only if the corresponding decoded instruction op code is evaluated to 1. How every instruction gets executed is explained below.

2.3.1 Sample Instruction

To better illustrate the operation, the indices of x, y, and z are assigned to decoder signals, and the indices of a, b, and c are assigned to the register of the corresponding register file.

- $x, y, z \in [0, 3]; a = x, b = y, c = z$

$$T_3 K_0 S1_x S2_y D_z : (RF)R_c \leftarrow (RF)R_a \wedge (RF)R_b, SC \leftarrow 0$$

$$[(RF)FunSel = 01, MuxCSel = 1, O1 = (4 + x)_2, O2 = (4 + y)_2]$$

- $x, y \in [0,3], z \in [4,7]; a = x, b = y, c = z - 4$

$$T_3K_0S1_xS2_yD_z : (ARF)R_c \leftarrow (RF)R_a \wedge (RF)R_b, SC \leftarrow 0$$

$$[(ARF)FunSel = 01, MuxCSel = 1, O1 = (4+x)_2, O2 = (4+y)_2]$$

- $y \in [0,3], x, z \in [4,7]; a = x - 4, b = y, c = z - 4$

$$T_3K_0S1_xS2_yD_z : (ARF)R_c \leftarrow (ARF)R_a \wedge (RF)R_b, SC \leftarrow 0$$

$$[(ARF)FunSel = 01, MuxCSel = 0, OutA = (x)_2, O2 = (4+y)_2]$$

- $y, z \in [0,3], x \in [4,7]; a = x - 4, b = y, c = z$

$$T_3K_0S1_xS2_yD_z : (RF)R_c \leftarrow (ARF)R_a \wedge (RF)R_b, SC \leftarrow 0$$

$$[(RF)FunSel = 01, MuxCSel = 0, OutA = (x)_2, O2 = (4+y)_2]$$

Control signals that are common in all cases are listed below:

$$[(ALU)FunSel = 0111, MuxASel = 00, MuxBSel = 00, MemCS = 1, IR_E = 0]$$

2.3.2 AND Instruction

T[3] && K[0]: DSTREG ← SREG1 AND SREG2

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.
- Source register 1 and source register 2 are connected from corresponding module's output to ALU's input.
- ALU's function is set to AND.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.3 OR Instruction

T[3] && K[1]: DSTREG ← SREG1 OR SREG2

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.

- Source register 1 and source register 2 are connected from corresponding module's output to ALU's input.
- ALU's function is set to OR.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.4 NOT Instruction

T[3] && K[2]: DSTREG ← NOT SREG

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.
- Source register is connected from corresponding module's output to ALU's input.
- ALU's function is set to NOT.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.5 ADD Instruction

T[3] && K[3]: DSTREG ← SREG1 ADD SREG2

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.
- Source register 1 and source register 2 are connected from corresponding module's output to ALU's input.
- ALU's function is set to ADD.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.6 SUB Instruction

T[3] && K[4]: DSTREG ← SREG1 SUB SREG2

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.

- Source register 1 and source register 2 are connected from corresponding module's output to ALU's input.
- ALU's function is set to SUB.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.7 LSR Instruction

T[3] && K[5]: DSTREG $\leftarrow$ LSR SREG

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.
- Source register is connected from corresponding module's output to ALU's input.
- ALU's function is set to LSR.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.8 LSL Instruction

T[3] && K[6]: DSTREG $\leftarrow$ LSL SREG

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.
- Source register is connected from corresponding module's output to ALU's input.
- ALU's function is set to LSL.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.9 INC Instruction

T[3] && K[7]: DSTREG $\leftarrow$ SREG

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.

- Source register is connected from corresponding module's output to ALU's input.
- ALU's function is set to A (NO_CHANGE).
- Memory is disabled and instruction register is disabled.

T[4] && K[7]: DSTREG $\leftarrow$ DSTREG + 1

- Corresponding module of destination register's function is selected to INCREMENT.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.10 DEC Instruction

T[3] && K[8]: DSTREG $\leftarrow$ SREG

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.
- Source register is connected from corresponding module's output to ALU's input.
- ALU's function is set to A (NO_CHANGE).
- Memory is disabled and instruction register is disabled.

T[4] && K[8]: DSTREG $\leftarrow$ DSTREG - 1

- Corresponding module of destination register's function is selected to DECREMENT.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.11 BRA Instruction

T[3] && K[9]: PC $\leftarrow$ VALUE

- Register file registers are disabled.
- Address register file input is connected to input register ouput (value).
- PC register is enabled in Address register file.
- Address register file function is set to LOAD.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.12 BNE Instruction

T[3] && K[10]: IF Z == 0: PC $\leftarrow$ VALUE

- Register file registers are disabled.
- Address register file input is connected to input register ouput (value).
- PC register is enabled in Address register file.
- Address register file function is set to LOAD.
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

T[3] && K[10]: IF Z == 1: PC $\leftarrow$ VALUE

- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.13 MOV Instruction

T[3] && K[11]: DSTREG $\leftarrow$ SREG

- MUXA and MUXB are set to select output of ALU.
- Destination registers are enabled and their corresponding module's function is set to LOAD.
- Source register is connected from corresponding module's output to ALU's input.
- ALU's function is set to A or B depending on the source register (NO_CHANGE).
- Memory is disabled, sequence counter is cleared and instruction register is disabled.

2.3.14 LD Instruction

T[3] && K[12]: Rx $\leftarrow$ VALUE

- Corresponding Rx in register file is enabled.
- Register file function is set to LOAD.
- If I is 0 (immediate address):

    Register file input is connected to instruction register's output.

- If I is 1 (direct address):

  Register file input is connected to the memory output.

  Memory address input is connected to address register in address register file.

  Memory is enabled and memory mode is set to READ.

- Every register in address register file is disabled.
- Sequence counter is cleared and instruction register is disabled.

2.3.15 ST Instruction

T[3] && K[13]: VALUE $\leftarrow$ Rx

- Memory is enabled and memory mode is set to WRITE.
- Memory address input is connected to address register in address register file.
- Memory input is connected to ALU's output.
- ALU's B input is connected to Rx.
- ALU's function is set to B (NO_CHANGE).
- Every register in address register file is disabled.
- Sequence counter is cleared and instruction register is disabled.

2.3.16 PUL Instruction

T[3] && K[14]: SP $\leftarrow$ SP + 1, Rx $\leftarrow$ M[SP]

- Register file input is connected to memory output.
- Only Rx is enabled in register file.
- Register file function is set to LOAD.
- Memory address input is connected to address register in address register file.
- Memory is enabled and memory mode is set to READ.
- Stack pointer in address register file is enabled.
- Adress register file function is set to INCREMENT.
- Sequence counter is cleared and instruction register is disabled.

2.3.17 PSH Instruction

T[3] && K[15]: SP ← SP - 1

- Memory mode is set to read.
- Only stack pointer in the address register file is enabled.
- Address register file function is set to DECREMENT.
- Instruction register is disabled.

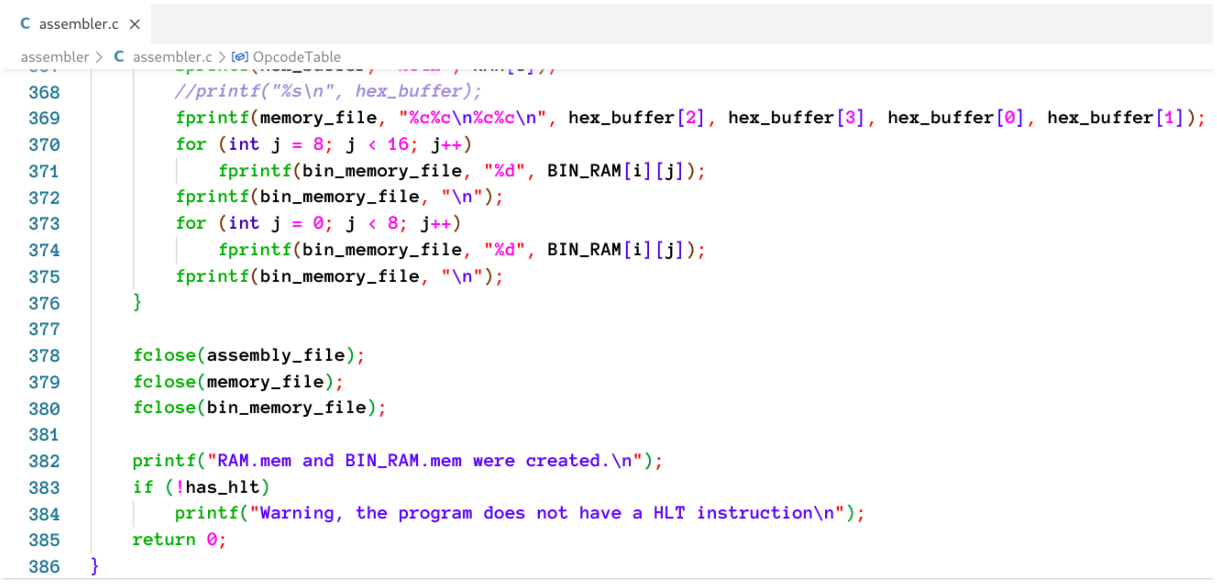
T[4] && K[15]: M[SP] ← Rx

- Memory is enabled and memory mode is set to WRITE.
- Memory address input is connected to stack pointer register.
- Every register in address file register is disabled.
- ALU's input is connected to Rx in the register file.
- ALU's function is set to Rx (B, NO_CHANGE) and connected to memory input.
- Sequence counter is cleared and instruction register is disabled.

3 Testing

Trying instructions on their own is not enough to ensure the overall proper function of our computer with more complex programs. Therefore, we developed an assembler in C programming language, which parses an assembly program, finds labels, calculates the correct positions in memory, translates all symbols to machine code and outputs binary and hexadecimal **.mem** files. Our assembler ignores all vertical and horizontal white-space, as well as the comments inserted by the # symbol. It can also recognize some errors in the assembly code and print error messages. It has two pseudo-instructions, ORG and END, and an extra instruction HLT, which was implemented in the Verilog design as well. ORG takes a hexadecimal value as a parameter, which represents the address location of the next instruction. Therefore, it is easy for a programmer to describe the contents of different parts of memory. END is optional, and it tells the assembler to stop at that line and to not process anything after it. Finally, the extra HLT instruction is important because it makes the machine stop execution and allow inspecting the results. In memory, it has hexadecimal value of 0xFFFF, and this value is checked by the control unit after the fetch and decode cycle. If the halt instruction is detected, then

$writememh and **$finish** is called in Verilog.

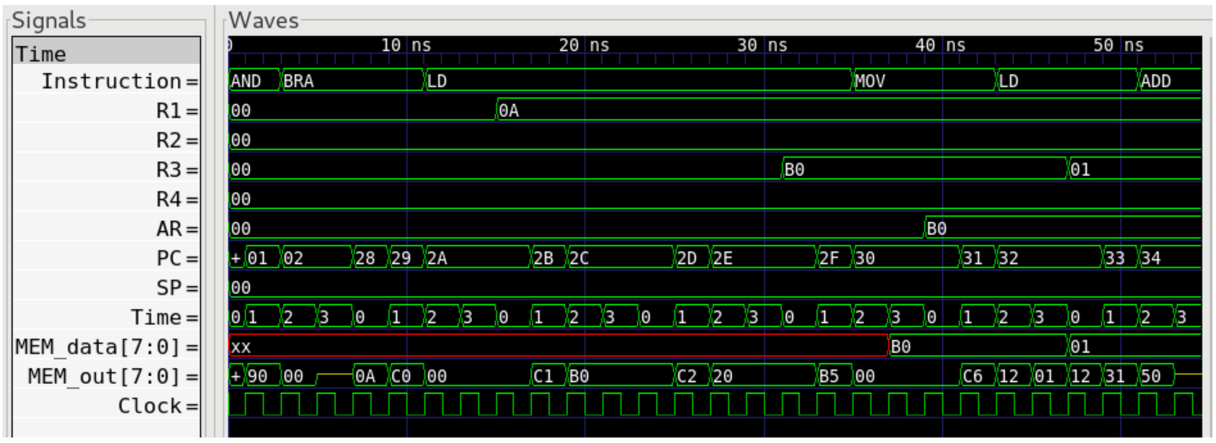


```c
        //printf("%s\n", hex_buffer);
        fprintf(memory_file, "%c%c\n%c%c\n", hex_buffer[2], hex_buffer[3], hex_buffer[0], hex_buffer[1]);
        for (int j = 8; j < 16; j++)
            fprintf(bin_memory_file, "%d", BIN_RAM[i][j]);
        fprintf(bin_memory_file, "\n");
        for (int j = 0; j < 8; j++)
            fprintf(bin_memory_file, "%d", BIN_RAM[i][j]);
        fprintf(bin_memory_file, "\n");
    }

    fclose(assembly_file);
    fclose(memory_file);
    fclose(bin_memory_file);

    printf("RAM.mem and BIN_RAM.mem were created.\n");
    if (!has_hlt)
        printf("Warning, the program does not have a HLT instruction\n");
    return 0;
}
```

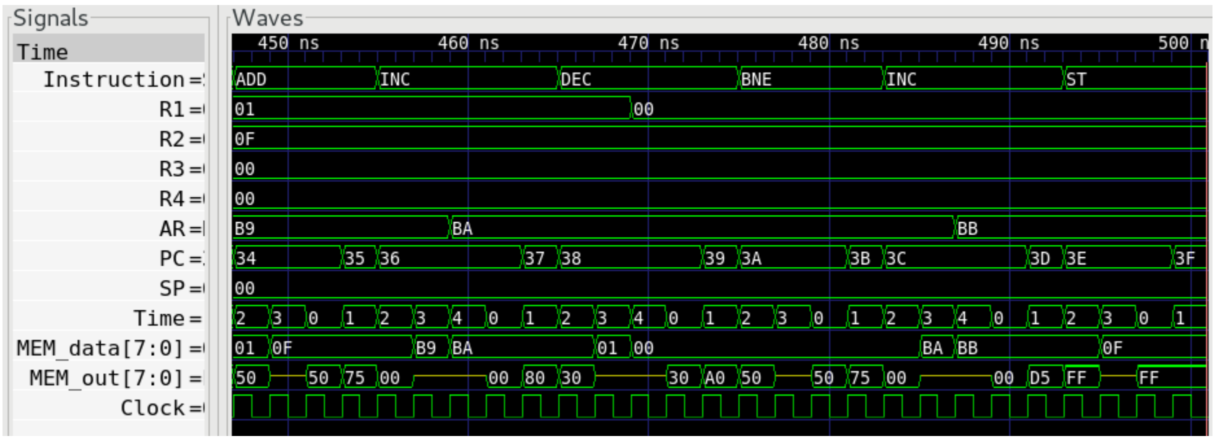
Last lines of our assembler program

The simple loop example given in the assignment was slightly modified and automatically assembled. Modifications include adding previously mentioned custom instructions, removing address mode specifiers for all instructions except LD (because it is the only one which does offer to choose the addressing mode), as well as assigning locations between 0xB0 and 0xBA with some data.

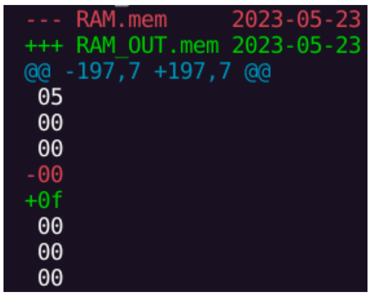
```
       BRA 0x28   # This instruction is written to the memory address 0x00,
                  # The first instruction must be written to the address 0x28

       ORG 0x28
       LD R1 IM 0x0A # This first instruction is written to the address 0x28,
                    # R1 is used for iteration number
       LD R2 IM 0x00 # R2 is used to store total
       LD R3 IM 0xB0
       MOV AR R3    # AR is used to track data address: starts from 0xB0
LABEL: LD R3 D      # R3    M[AR] (AR = 0xB0 to 0xBA)
       ADD R2 R2 R3 # R2    R2 + R3 (Total = Total + M[AR])
       INC AR AR    # AR    AR + 1 (Next Data)
       DEC R1 R1    # R1    R1    1 (Decrement Iteration Counter)
       BNE LABEL # Go back to LABEL if Z=0 (Iteration Counter > 0)
       INC AR AR    # AR    AR + 1 (Total will be written to 0xBB)
       ST R2      # M[AR]    R2 (Store Total at 0xBB)
       HLT

       ORG 0xB0 # place some values at 0xB0
       0x01
       0x02
       0x03
       0x04
       0x05
       END
```



(a) Beginning of execution



(b) End of execution



```
--- RAM.mem     2023-05-23
+++ RAM_OUT.mem 2023-05-23
@@ -197,7 +197,7 @@
 05
 00
 00
-00
+0f
 00
 00
 00
```

(c) Differences in memory

Loop example program's results

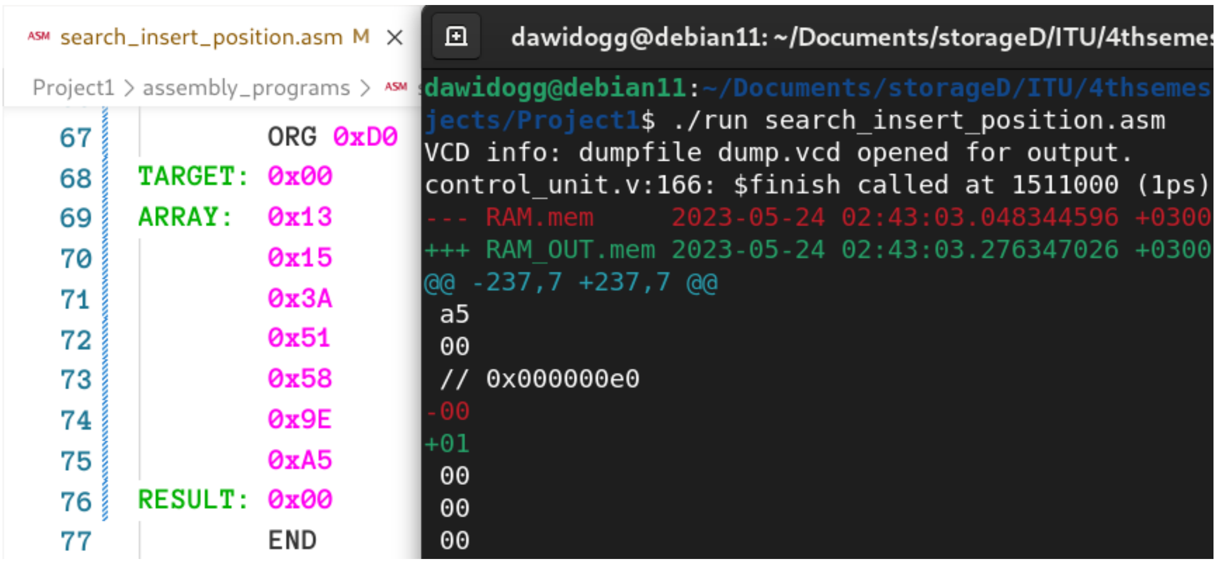
After the execution, our computer writes 0x0F to the memory, which is indeed the sum of values between the addresses 0xB0 and 0xBA.

As an example from our part, we present a solution to a LeetCode easy question in our computer's assembly language:

```
# https://leetcode.com/problems/search-insert-position/
# Given a sorted array of distinct integers and a target value,
# return the index if the target is found. If not, return the
# index where it would be if it were inserted in order.

ORG 0x0
# initialize the stack
LD R1 IM 0xFF
MOV SP R1
# load the target to R1
LD R1 IM TARGET
MOV AR R1
LD R1 D
# load the array pointer to AR
LD R2 IM ARRAY
MOV AR R2

LOOP_1:    BRA IS_SMALLER
RETURN:    PUL R3 # take the result of the function
           LD R4 IM 0x01 # check if true
           SUB R2 R3 R4
           BNE CONTINUE_3
           BRA EXIT
CONTINUE_3: INC AR AR
           INC AR AR
           # check if the end of array is reached
           LD R3 D
           MOV R3 R3
           BNE LOOP_1
           BRA EXIT

# R1 - Target, R2 - Value-to-Compare, R3 - Limit, R4 - Guessing Pointer
IS_SMALLER: MOV R4 R1     # set the guessing pointer as the target initially
LOOP_2:    LD R2 D        # R2 stores the value currently pointed
           SUB R3 R2 R4
           BNE CONTINUE_1
           BRA RETURN_TRUE # if guessing pointer reaches the value-to-compare
```

```
                          # before overflow, the result is true
CONTINUE_1: LD R3 IM 0xFF # store limit
           SUB R2 R3 R4
           BNE CONTINUE_2
           BRA RETURN_FALSE # if guessing pointer reaches the limit without
                          # finding the value-to-compare, the result is false
CONTINUE_2: INC R4 R4
           BRA LOOP_2
RETURN_FALSE: LD R3 IM 0x00
              PSH R3 # push false
              BRA RETURN
RETURN_TRUE:  LD R3 IM 0x01
              PSH R3 # push true
              BRA RETURN

       # save result
EXIT:  LD R1 IM ARRAY
       SUB R1 AR R1
       LSR R1 R1
       INC R1 R1
       LD R2 IM RESULT
       MOV AR R2
       ST R1
       # clear stack
       LD R1 IM 0x00
       PSH R1
       PUL R1
       HLT

       ORG 0xD0
TARGET: 0x99
ARRAY:  0x13
       0x15
       0x3A
       0x51
       0x58
       0x9E
       0xA5
RESULT: 0x00
       END
```

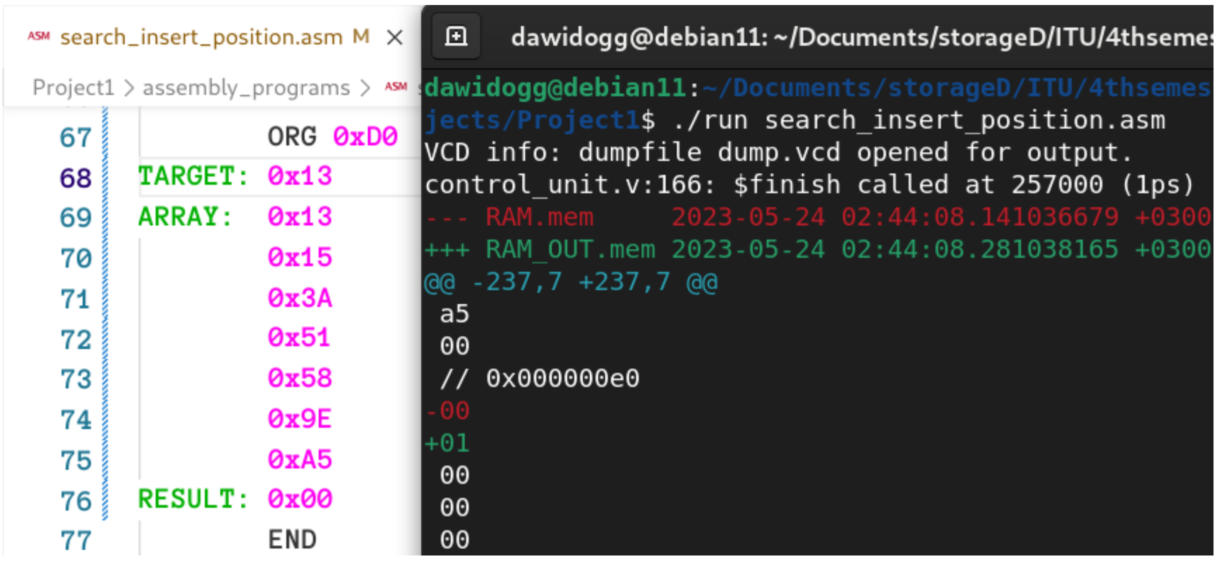
To find a specific element in an array, one must be able to compare the values. However, in our instruction set we have just one conditional branch, the BNE instruction, which branches if the ALU output is not zero. The program would have been much shorter if we made use of hardware optimization. Instead, we wrote a subroutine called IS_SMALLER, which compares the registers R1 and R2, and pushes either true or false to the stack. Later in the main loop, the returned value is pulled and decisions are made. The main loop continues to search for a suitable position in array, and when it finds, the program jumps to the EXIT routine. There, some operations are performed before the output can be stored. The value returned by the program is the index where the target was found or can be inserted, with one as the first index of the array. The program functions correctly as it can be seen from the figures below:



```
        ORG 0xD0
TARGET: 0x00
ARRAY:  0x13
        0x15
        0x3A
        0x51
        0x58
        0x9E
        0xA5
RESULT: 0x00
        END
```

```
dawidogg@debian11: ~/Documents/storageD/ITU/4thseme
dawidogg@debian11:~/Documents/storageD/ITU/4thsemes
jects/Project1$ ./run search_insert_position.asm
VCD info: dumpfile dump.vcd opened for output.
control_unit.v:166: $finish called at 1511000 (1ps)
--- RAM.mem     2023-05-24 02:43:03.048344596 +0300
+++ RAM_OUT.mem 2023-05-24 02:43:03.276347026 +0300
@@ -237,7 +237,7 @@
 a5
 00
 // 0x000000e0
-00
+01
 00
 00
 00
```

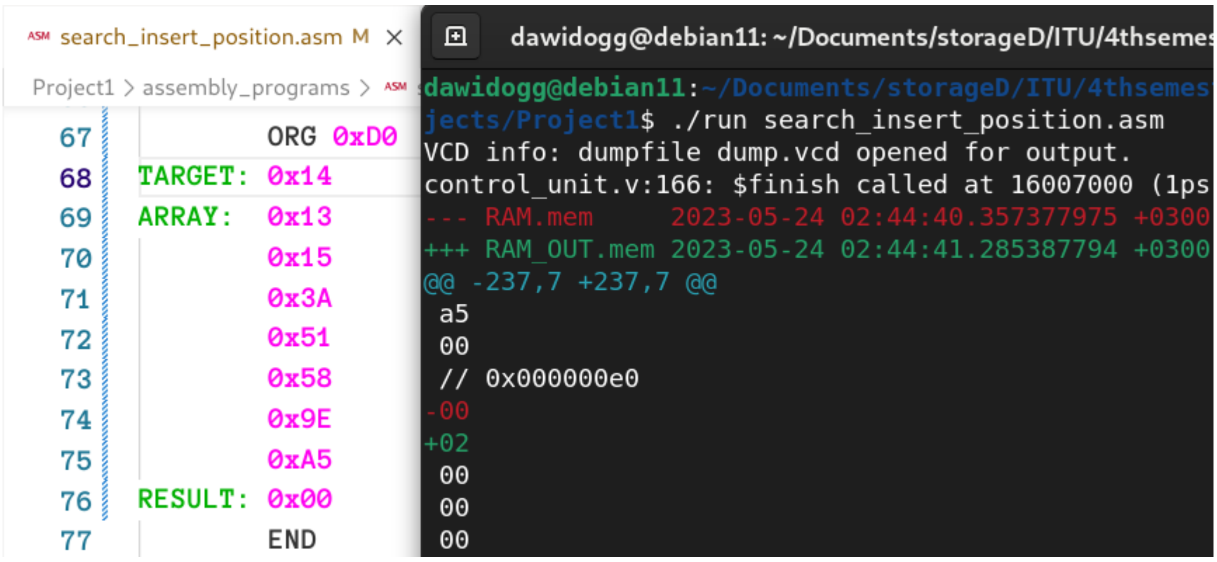
Test with input 0x00. Since it is the smallest, it should be inserted to the first position.



```
search_insert_position.asm M ×
Project1 > assembly_programs >
              ORG 0xD0
TARGET:  0x13
ARRAY:   0x13
         0x15
         0x3A
         0x51
         0x58
         0x9E
         0xA5
RESULT:  0x00
         END
```

```
dawidogg@debian11: ~/Documents/storageD/ITU/4thseme
dawidogg@debian11:~/Documents/storageD/ITU/4thsemes
jects/Project1$ ./run search_insert_position.asm
VCD info: dumpfile dump.vcd opened for output.
control_unit.v:166: $finish called at 257000 (1ps)
--- RAM.mem     2023-05-24 02:44:08.141036679 +0300
+++ RAM_OUT.mem 2023-05-24 02:44:08.281038165 +0300
@@ -237,7 +237,7 @@
 a5
 00
 // 0x000000e0
-00
+01
 00
 00
 00
```

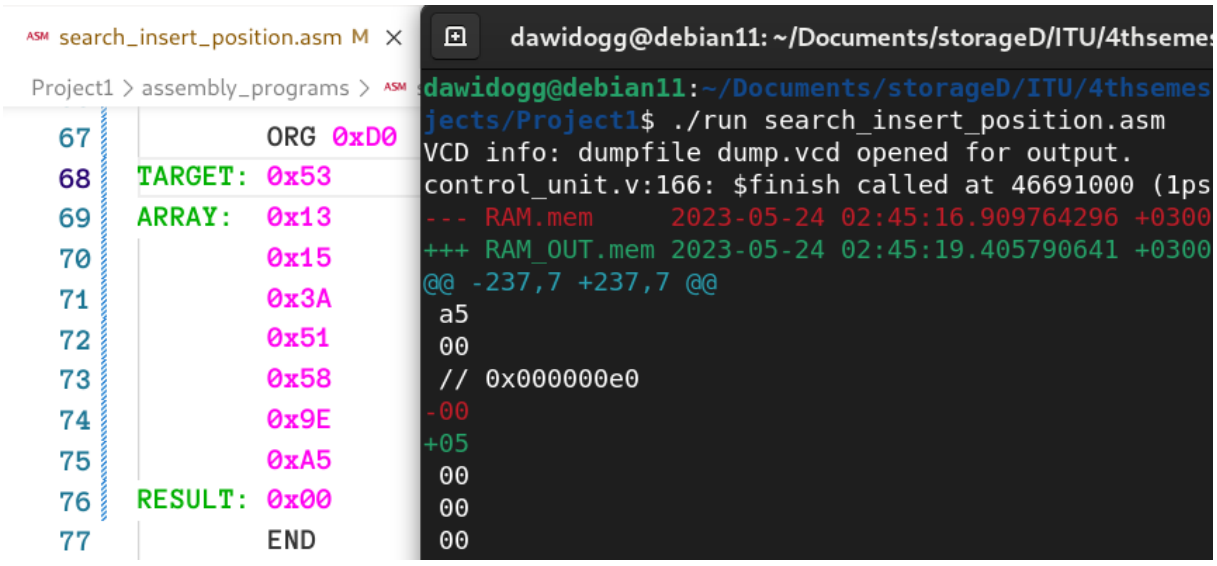
Test with input 0x13. Here we found the target, and the result is its position, one.



```
search_insert_position.asm M ×
Project1 > assembly_programs >
              ORG 0xD0
TARGET:  0x14
ARRAY:   0x13
         0x15
         0x3A
         0x51
         0x58
         0x9E
         0xA5
RESULT:  0x00
         END
```

```
dawidogg@debian11: ~/Documents/storageD/ITU/4thseme
dawidogg@debian11:~/Documents/storageD/ITU/4thsemes
jects/Project1$ ./run search_insert_position.asm
VCD info: dumpfile dump.vcd opened for output.
control_unit.v:166: $finish called at 16007000 (1ps
--- RAM.mem     2023-05-24 02:44:40.357377975 +0300
+++ RAM_OUT.mem 2023-05-24 02:44:41.285387794 +0300
@@ -237,7 +237,7 @@
 a5
 00
 // 0x000000e0
-00
+02
 00
 00
 00
```

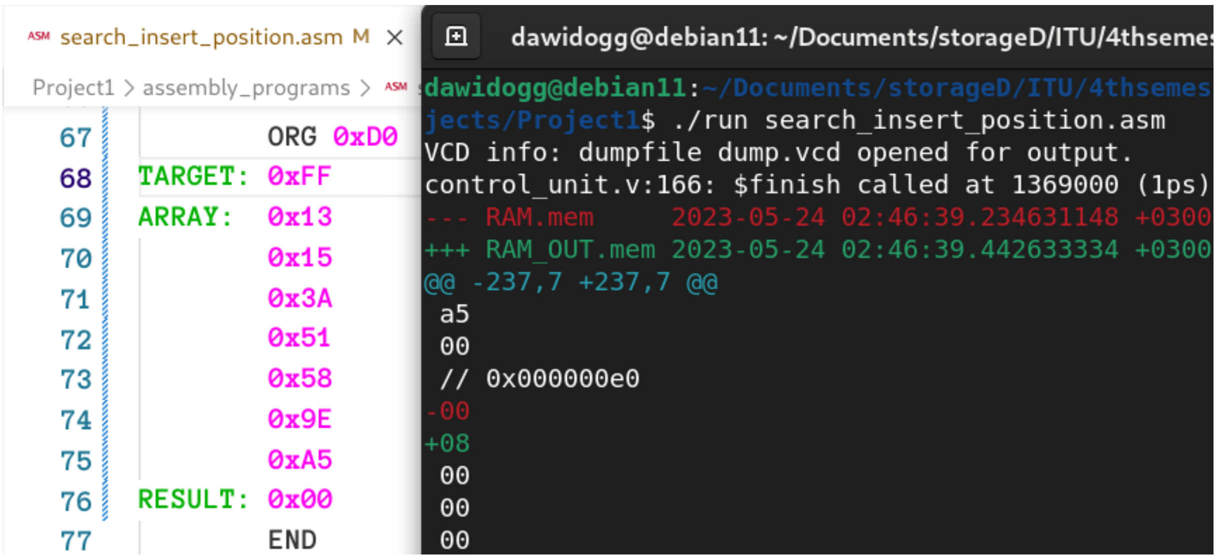
Test with input 0x14. Target is placed between 0x13 and 0x15, and gets the second position.



```
search_insert_position.asm M ×
Project1 > assembly_programs >
        ORG 0xD0
TARGET: 0x53
ARRAY:  0x13
        0x15
        0x3A
        0x51
        0x58
        0x9E
        0xA5
RESULT: 0x00
        END
```

```
dawidogg@debian11: ~/Documents/storageD/ITU/4thseme
dawidogg@debian11:~/Documents/storageD/ITU/4thsemes
jects/Project1$ ./run search_insert_position.asm
VCD info: dumpfile dump.vcd opened for output.
control_unit.v:166: $finish called at 46691000 (1ps
--- RAM.mem     2023-05-24 02:45:16.909764296 +0300
+++ RAM_OUT.mem 2023-05-24 02:45:19.405790641 +0300
@@ -237,7 +237,7 @@
 a5
 00
 // 0x000000e0
-00
+05
 00
 00
 00
```

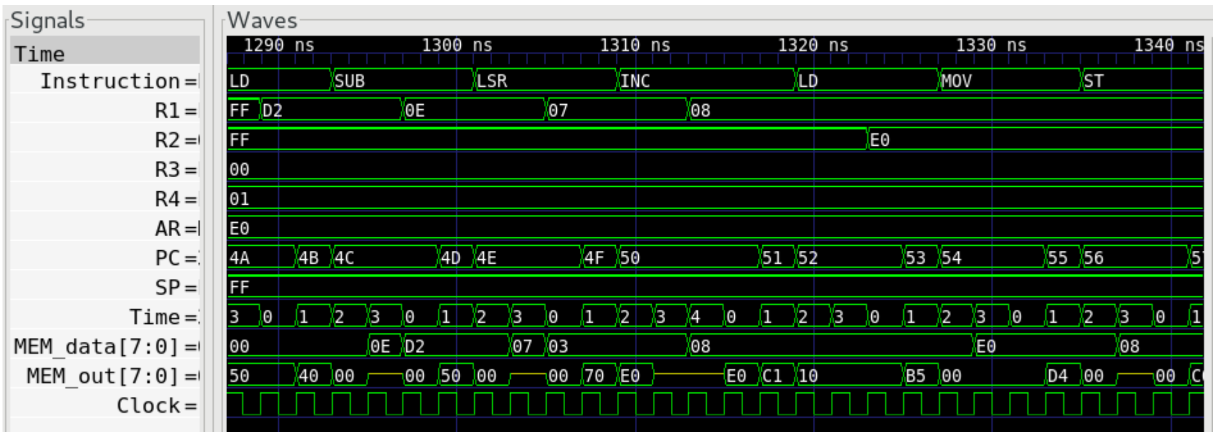
Test with input 0x53. Output is the 5th position.



```
search_insert_position.asm M ×
Project1 > assembly_programs >
        ORG 0xD0
TARGET: 0xFF
ARRAY:  0x13
        0x15
        0x3A
        0x51
        0x58
        0x9E
        0xA5
RESULT: 0x00
        END
```

```
dawidogg@debian11: ~/Documents/storageD/ITU/4thseme
dawidogg@debian11:~/Documents/storageD/ITU/4thsemes
jects/Project1$ ./run search_insert_position.asm
VCD info: dumpfile dump.vcd opened for output.
control_unit.v:166: $finish called at 1369000 (1ps)
--- RAM.mem     2023-05-24 02:46:39.234631148 +0300
+++ RAM_OUT.mem 2023-05-24 02:46:39.442633334 +0300
@@ -237,7 +237,7 @@
 a5
 00
 // 0x000000e0
-00
+08
 00
 00
 00
```

Test with input 0xFF. It is the biggest value, so it is placed to the end of array, at the 8th position.



Waveform at the moment when the output is stored.

4 Conclusion

In conclusion, this project helped us gain a deeper understanding of basic computer and how it works. By designing our own control unit, we were able to implement our instructions. To test our implementations, we created machine codes for input and evaluated the results of each instruction. Overall, this project has been an excellent learning experience that has helped us develop our skills in basic computer organization and Verilog.