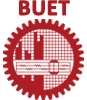
**BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY**



**Department of Electrical and Electronic Engineering**

**Assignment**

**4-BIT COMPUTER DESIGN**

**Course No :** EEE 415

**Course Title :** Microprocessor and Embedded Systems

**Student Name :** Mir Sayeed Mohammad

**Student ID :** 1606003

**Submission Date :** 1 April 2021

**Submitted to :** Dr. Sajid Muhaimin Choudhury

Sameia Zaman

**Objectives:**

1. Implementation of “computer” Verilog module for the 4-bit computer
2. Implementation of “testbench” module for programming and testing the computer
3. Assembler design in python for generating machine code to be used in the testbench code
4. Testing computer and generating results using web based tool “EDA-Playground”

**Software Used:**

1. Quartus II
2. EDA-Playground
3. Pycharm (python IDE)

**Design Principles:**

***Computer Module:***

1. A main module named “computer” contains the implemented architecture for the 4-bit computer implementation. This module can also be synthesized for FPGA hardware using Quartus II software.
2. The main module has the structure:

module computer (data\_in, data\_out, osc\_clock, reset, prog\_in, prog\_clk, prog\_add);

Table 1: Input output list of main module

|  |  |  |
| --- | --- | --- |
| Port name | Pin Count | Purpose |
| Data\_in | 4 | General input for the computer, doubles as the data input while uploading instructions |
| Data\_out | 4 | General output for the computer, used to drive display/output interface |
| Osc\_clock | 1 | Clock input for the computer, assuming that there is no internal clock available |
| Reset | 1 | Resets all general purpose registers to initial value |
| Prog\_in | 8 | Input port for loading instructions in the program memory |
| Prog\_clk | 1 | On positive edge of prog\_clk, data\_in loads data into data memory and prog\_in loads instructions in program memory on the prog\_add memory address |
| Prog\_add | 4 | Address for program and data memory while uploading instructions. |

There are several registers inside the main module helping with various operations

Table 2: Internal registers of main module

|  |  |  |
| --- | --- | --- |
| Reg Name | Size | Purpose |
| A, B, TMP | 4 | General purpose registers |
| PROG\_MEM | 16\*8 | 8 bit memory location with 16 addresses for storing instructions (opcode + operand) |
| DATA\_MEM | 16\*4 | 4 bit memory location with 16 addresses for storing data |
| STAK\_MEM | 16\*4 | 4 bit memory location with 16 addresses for use as stack |
| IP | 4 | Instruction pointer, acts as address for PROG\_MEM |
| SP | 4 | Stack pointer, acts as address for STAK\_MEM |
| INS | 4 | Instruction register, holds instruction fetched from PROG\_MEM |
| ADD | 4 | Holds operand fetched from PROG\_MEM and conditionally as address for DATA\_MEM |
| HALT\_FLAG | 1 | Stores halt state |
| ZF | 1 | Stores zero flag |
| CF | 1 | Stores carry flag |

1. The provided instruction set was:

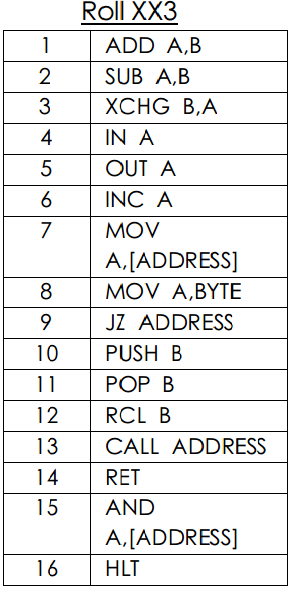


Fig: Instruction set

Based on the sequence of the instructions, 4 bits were assigned as op-code to each instruction. In this module, ***it is assumed that the instructions have fixed operands unless specified***. For example, ADD A, B implies that ADD op-code will always add values from register A and B and store them in A, and cannot call any other registers. On the other hand, MOV A, BYTE implies that a 4 bit operand will be provided, but the move operation will always work on the A register, and not any other registers.

Table 3: Instruction as used in the computer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Serial | Instruction | Machine Opcode | Operand | Pseudocode |
| 1 | ADD A, B | 0000 | - | A = A + B |
| 2 | SUB A, B | 0001 | - | A = A – B |
| 3 | XCHG B, A | 0010 | - | Swap A, B |
| 4 | IN A | 0011 | - | A = data\_in |
| 5 | OUT A | 0100 | - | data\_out = A |
| 6 | INC A | 0101 | - | A = A + 1 |
| 7 | MOV A, [ADDRESS] | 0110 | ADD | A = DATA\_MEM[ADD] |
| 8 | MOV A, BYTE | 0111 | ADD | A = ADD |
| 9 | JZ ADDRESS | 1000 | ADD | IP = ADD |
| 10 | PUSH B | 1001 | - | STAK\_MEM[SP] = B  SP = SP + 1 |
| 11 | POP B | 1010 | - | SP = SP – 1  B = STAK\_MEM[SP] |
| 12 | RCL B | 1011 | - | {CF, B} = {B, CF} |
| 13 | CALL ADDRESS | 1100 | ADD | STAK\_MEM[SP] = IP  SP = SP + 1  IP = ADD |
| 14 | RET | 1101 | - | SP = SP – 1  IP = STAK\_MEM[SP] |
| 15 | AND A, [ADDRESS] | 1110 | ADD | A = A & DATA\_MEM[ADD] |
| 16 | HLT | 1111 | - | HALT\_FLAG = 1 |

\*here the comment column includes the registers explained in table 2

4. The computer module is designed to work in the following process:

a) On negative edge of clock,

INS = PROG\_MEM[IP][7:4];

ADD = PROG\_MEM[IP][3:0];

Meaning that instructions and operands will be fetched and stored in INS and ADD registers. IP points to which instruction is to be fetched.

(b) On positive edge of clock,

IP = IP + 1; // Instruction pointer increments

Using if-else statements on the value of INS, appropriate code will be executed based on the pseudo code of table-3. For example,

if (INS == 4’b0000)

A = A + B;

(c) If HALT\_FLAG == 1, steps (a) and (b) wont be executed

(d) If reset == 1, all registers will be set to zero and stack pointer to 15.

(e) If prog\_clk gets a positive edge, data will be loaded into memory such that

always @(posedge prog\_clk) begin

PROG\_MEM[prog\_add] = prog\_in;

DATA\_MEM[prog\_add] = data\_in;

end

This is used to upload program to the memory of the computer.

***Test-bench Module:***

1. Test-bench module creates an instance of the computer module
2. The inputs to computer module are declared as registers and the outputs are declared as wire in the test-bench module
3. Timing delay is used to produce clock pulse feeding the computer module
4. A copy of PROG\_MEM and DATA\_MEM is declared in the test-bench module. Unlike computer module, here they are initialized inside an “initial” block. By this initialization, it is possible to write programs for our computer.
5. A loop iterates 16 times to upload instructions and data to the computer module address by address from PROG\_MEM and DATA\_MEM in test-bench using the data\_in, prog\_in, prog\_add and prog\_clk pins (reset pin set to high)
6. After uploading code, the test-bench sets the reset pin to zero, and the computer module starts executing the loaded instructions.

***Python Assembler:***

1. The python assembler is called from the windows command line “cmd.exe” as follows:

>> python assembler.py “filename.asm”

It expects an argument in the .asm file format, and generates a .v file with the same name.

1. The python assembler parses the text information inside the assembly file. As a general rule, data is supposed to be contained inside a section starting with “***.DATA***” and ending with “***.END\_DATA***”. Similarly, code is supposed to be contained inside a section starting with “***.CODE***” and ending with “***.END\_CODE***”

The assembler supports compilation for up to one procedure. It has to be declared inside the code segment, starting with “***.PROC***” and ending with “***.END\_PROC***”

Jumps flags described as “***SOME\_JUMP:***” can be used both inside the main code or the procedures as necessary. The assembler automatically detects on which location the jump has been defined. The jump flags can have any names.

1. At the beginning of execution, the assembler separates data, code and procedure segment. Data segment expects data declared as variables, an example:

.DATA

X 5

Y 10

M 4

.END\_DATA

These variable names are set as python dictionary keys, and starting from the first variable, the dictionary stores the variable location in memory as well as the value.

1. After creating a data dictionary, the code is interpreted almost in a similar fashion to how we defined our computer Verilog module. Using if-else statements, instructions are interpreted. In case where operands are pointing to a variable, for example:

MOV A, X

Here it automatically detects where X is stored in the data dictionary, and uses its corresponding address. As of now, the assembler cannot take addresses directly, but uses variables instead.

Jump and calling addresses are implemented in a similar fashion.

**Simulation and Output:**

***Example 1: Directly coded on testbench(addition, and, output)***

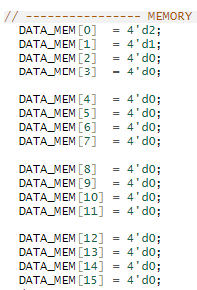
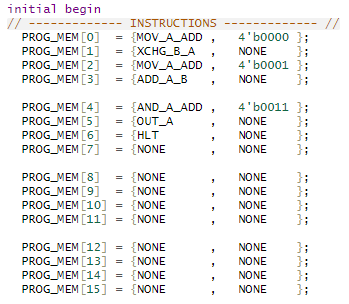


Fig: (a) Program (b) Data on testbench

Table 4: Instruction set for example 1

|  |  |  |
| --- | --- | --- |
| **Address** | **Instruction Memory** | **Data Memory** |
| 0 | **MOV A, [0000]** | **D2** |
| 1 | **XCHG B, A** | **D1** |
| 2 | **MOV A, [0001]** | **D0** |
| 3 | **ADD A, B** | **D0** |
| 4 | **AND A, [0011]** | **-** |
| 5 | **OUT A** | **-** |
| 6 | **HLT** | **-** |
| 7 | **-** | **-** |
| 8 | **-** | **-** |
| 9 | **-** | **-** |
| 10 | **-** | **-** |
| 11 | **-** | **-** |
| 12 | **-** | **-** |
| 13 | **-** | **-** |
| 14 | **-** | **-** |
| 15 | **-** | **-** |

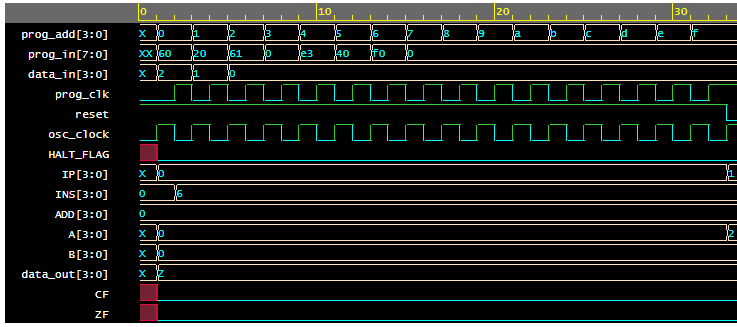


Fig: Output waveshape (first part – uploading)

In the first part of the output waveshape, the prog\_add can be seen incrementing, and the prog\_clk pulsating. At this stage, program and data is loaded on to the computer.

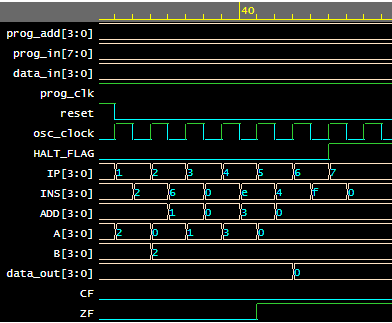


Fig: Output waveshape (second part – execution)

Step 1: A = 2

Step 2: B = 2, A = 0

Step 3: A = 1

Step 4: A = A + B = 3

Step 5: A = A & 0 = 0

Step 6: Output A

Step 7: Halt

***Example 2: Directly coded on testbench(call, ret, push, pop)***

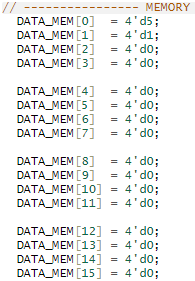
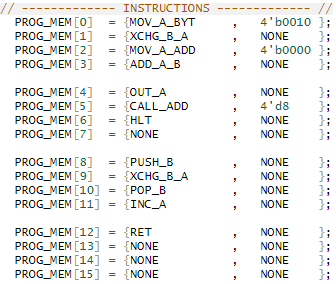


Fig: (a) Program (b) Data on testbench

Table 5: Instruction set for example 2

|  |  |  |
| --- | --- | --- |
|  | **Instruction Memory** | **Data** |
| 0 | **MOV A, 0010** | **D5** |
| 1 | **XCHG B, A** | **D1** |
| 2 | **MOV A, [0000]** | **-** |
| 3 | **ADD A, B** | **-** |
| 4 | **OUT A** | **-** |
| 5 | **CALL [d8]** | **-** |
| 6 | **HLT** | **-** |
| 7 | **-** | **-** |
| 8 | **PUSH B** | **-** |
| 9 | **XCHG B, A** | **-** |
| 10 | **POP B** | **-** |
| 11 | **INC A** | **-** |
| 12 | **RET** | **-** |
| 13 | **-** | **-** |
| 14 | **-** | **-** |
| 15 | **-** | **-** |

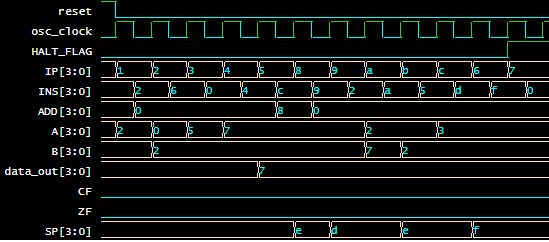


Fig: Output waveshape (second part – execution)

Step 1: A = 2

Step 2: B = 2, A = 0

Step 3: A = 5

Step 4: A = A + B = 7

Step 5: Output A = 7

Step 6: Call instruction on location 8, push current address to stack, SP = 14

Step 7: Push B = 2 to stack, SP = 13

Step 8: B = 7, A = 2

Step 9: A = 3 (incremented)

Step 10: Pop B = 2, SP = 14

Step 11: Return, SP = 15, current instruction pointer at location 6

Step 12: Reached halt

***Example 3:*** ***Assembly file, python assembler and testbench(simple arithmetic)***

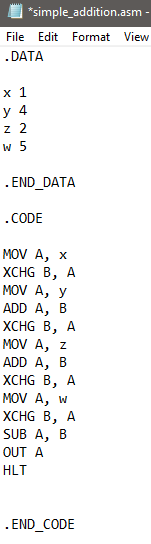
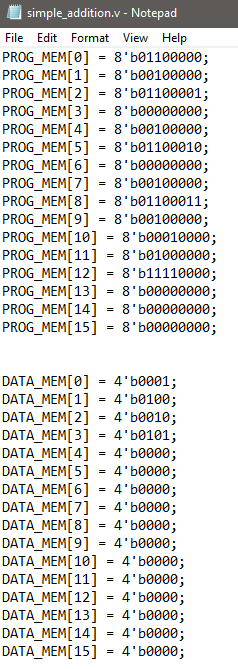
 

Fig: (a) Assembly code (b) Generated verilog snippet of machine code

The text from output was transferred to the testbench module and the simulation was run on EDA playground.

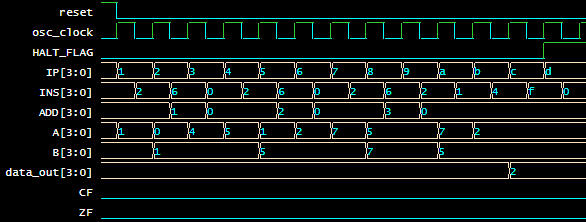


Fig: Output waveshape

Step 1: A = x = 1

Step 2: B = 1, A = 0

Step 3: A = y = 4

Step 4: A = A + B = 5

Step 5: B = 5, A = 1

Step 6: A = z = 2

Step 7: A = A + B = 7

Step 8: B = 7, A = 5

Step 9: A = w = 5

Step 10: B = 5, A = 7

Step 11: A = A – B = 2

Step 12: Output A = 2

Step 13: Halt

***Example 4:*** ***Assembly file, python assembler and testbench(procedure call, jump)***

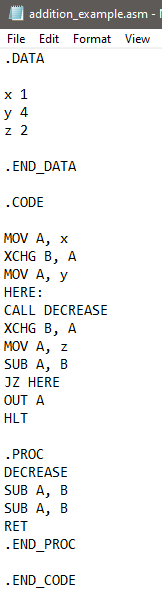
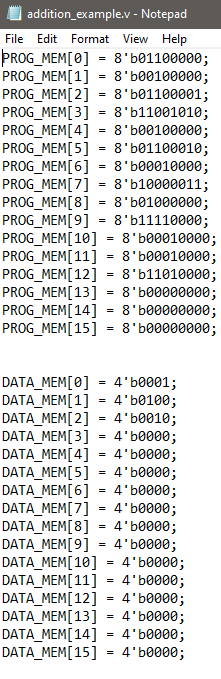
 

Fig: (a) Assembly code (b) Generated verilog snippet of machine code

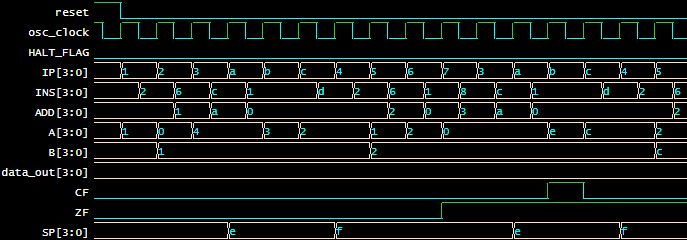


Fig: Output waveshape

In this example, the procedure call and conditional jump statement keeps the program running until the simulation time runs out (keeps iterating).

**Discussion:**

1. The implementation of instructions was limited by the provided instruction set.
2. The variety of programs running on the computer is also highly limited due to the instruction set limit.
3. The computer architecture does not follow any specific architecture such as SAP-1, 2, 3 or 8086 architectures. However due to the nature of the synthesis, it can execute instructions in each clock cycle, making the computer very fast (although not efficient in terms of FPGA resource usage).
4. The assembler is limited to maximum 1 instruction, and excludes syntax/error checking. However it can be used to write programs for the computer intuitively and effectively.

**APPENDIX**

* EDA playground link : <https://www.edaplayground.com/x/pwQy>
* Link to all the codes, presentation: <https://github.com/ClockWorkKid/Microprocessor/tree/main/Assignment%204-Bit%20Computer>