Contents

1 Basic Test Results

```
********** TESTING FOLDER STRUCTURE START **********
Running test5.sh:
Your logins are: muaz.abdeen, is that ok?

********** TESTING FOLDER STRUCTURE END **********

********** PROJECT TEST START **********

Memory passed

********** PROJECT TEST END **********
```

2 README

```
muaz.abdeen
===============================================================================
Muaz Abdeen, ID 300575297, muaz.abdeen@mail.huji.ac.il
===============================================================================

                           Project 5 - Computer Architecture
                           ---------------------------------


Submitted Files
---------------
(1)  README         - This file.
(2)  Memory.hdl        - The Memory.hdl implementation.
(3)  CPU.hdl        - The CPU.hdl implementation.
(4)  Computer.hdl    - The Computer.hdl implementation.
(5)  ExtendAlu.hdl    - The ExtendAlu.hdl implementation.
(6)  CpuMul.hdl        - The CpuMul.hdl implementation.


Remarks
-------
* ...
```

3 CPU.hdl

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/05/CPU.hdl

/**
 * The Hack CPU (Central Processing unit), consisting of an ALU,
 * two registers named A and D, and a program counter named PC.
 * The CPU is designed to fetch and execute instructions written in
 * the Hack machine language. In particular, functions as follows:
 * Executes the inputted instruction according to the Hack machine
 * language specification. The D and A in the language specification
 * refer to CPU-resident registers, while M refers to the external
 * memory location addressed by A, i.e. to Memory[A]. The inM input
 * holds the value of this location. If the current instruction needs
 * to write a value to M, the value is placed in outM, the address
 * of the target location is placed in the addressM output, and the
 * writeM control bit is asserted. (When writeM==0, any value may
 * appear in outM). The outM and writeM outputs are combinational:
 * they are affected instantaneously by the execution of the current
 * instruction. The addressM and pc outputs are clocked: although they
 * are affected by the execution of the current instruction, they commit
 * to their new values only in the next time step. If reset==1 then the
 * CPU jumps to address 0 (i.e. pc is set to 0 in next time step) rather
 * than to the address resulting from executing the current instruction.
 */

CHIP CPU {

    IN  inM[16],         // M value input  (M = contents of RAM[A])
        instruction[16], // Instruction for execution
        reset;           // Signals whether to re-start the current
                         // program (reset==1) or continue executing
                         // the current program (reset==0).

    OUT outM[16],        // M value output
        writeM,          // Write to M?
        addressM[15],    // Address in data memory (of M)
        pc[15];          // address of next instruction

    PARTS:
    // Determine the instruction type A or C
    Mux16(a=instruction ,b=ALUout ,sel=instruction[15] ,out=inA);

    /**
    * A-register will load the instruction if it is:
    * (1) an A-instruction
    * (2) a C-instruction that stores the value in A (d1=1)
    */

    //Determine load status of A-register:
    Not(in=instruction[15] ,out=negateType);
    Or(a=negateType, b=instruction[5] ,out=loadA);
    ARegister(in=inA ,load=loadA ,out=outA ,out[0..14]=addressM);

    //writeM assertion (depends on ins[15] & d3)
    And(a=instruction[15] ,b=instruction[3] ,out=writeM);

    /**
    * ALU takes two 16-bit registers:
    * (1) D-register, depends on (ins[15] & d2)
    * (2) A-register or data memory, depends on (ins[15] & a)
    */

    //Determine the x-input of the ALU:
    And(a=instruction[15] ,b=instruction[4] ,out=loadD);
    DRegister(in=ALUout ,load=loadD ,out=ALUinX);

    //Determine the y-input of the ALU:
    And(a=instruction[15] ,b=instruction[12] ,out=loadY);
    Mux16(a=outA ,b=inM ,sel=loadY ,out=ALUinY);

    // pass input to ALU
    ALU(x=ALUinX ,y=ALUinY ,zx=instruction[11] ,nx=instruction[10] ,zy=instruction[9] ,ny=instruction[8] ,f=instruction[7] ,

    /**
    * whether to jump or not depents (beside instruction type) on the
    * existence of one of the following:
    * (1) j1=1 and ALUneg=1
    * (2) j2=1 and ALUzr=1
    * (3) j3=1 and not (ALUneg=1 or ALUzr=1)
    */

    // calculate the jump condition
    Or(a=ALUzr ,b=ALUneg ,out=notpositive);
    Not(in=notpositive ,out=ALUpos);

    And(a=instruction[2] ,b=ALUneg ,out=j1);
    And(a=instruction[1] ,b=ALUzr ,out=j2);
    And(a=instruction[0] ,b=ALUpos ,out=j3);

    Or(a=j1 ,b=j2 ,out=j1orj2);
    Or(a=j1orj2 ,b=j3 ,out=jOrs);

    And(a=jOrs ,b=instruction[15] ,out=jump);

    /**
    * in the programe counter increment is always true because
    * if no reset or jump the PC must be incremented
    */

    // set the programe counter
    PC(in=outA ,load=jump ,inc=true ,reset=reset ,out[0..14]=pc);
}
```

4 Computer.hdl

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/05/Computer.hdl

/**
 * The HACK computer, including CPU, ROM and RAM.
 * When reset is 0, the program stored in the computer's ROM executes.
 * When reset is 1, the execution of the program restarts.
 * Thus, to start a program's execution, reset must be pushed "up" (1)
 * and "down" (0). From this point onward the user is at the mercy of
 * the software. In particular, depending on the program's code, the
 * screen may show some output and the user may be able to interact
 * with the computer via the keyboard.
 */

CHIP Computer {

    IN reset;

    PARTS:

    ROM32K(address=pc ,out=instruction);

    CPU(inM=inM ,instruction=instruction ,reset=reset ,outM=outM ,writeM=writeM ,addressM=addressM ,pc=pc);

    Memory(in=outM ,load=writeM ,address=addressM ,out=inM);
}
```

5 CpuMul.hdl

```
/**
* This chip is an extension of the book CPU by using the extended ALU.
* More specificly if instruction[15]==0 or (instruction[14] and instruction[13] equals 1)
* the CpuMul behave exactly as the book CPU.
* While if it is C instruction and instruction[13] == 0 the output will be D*A/M
* (the choice between multiplying D by A or D by M is according to instruction[12]).
* Moreover, if it is c instruction and instruction[14] == 0 it will behave as follows:
*
* instruction:  | 12 | 11 | 10 |
* ______________________________
* shift left D  | 0  | 1  | 1  |
* shift left A  | 0  | 1  | 0  |
* shift left M  | 1  | 1  | 0  |
* shift right D | 0  | 0  | 1  |
* shift right A | 0  | 0  | 0  |
* shift right M | 1  | 0  | 0  |
**/

CHIP CpuMul{

    IN  inM[16],         // M value input  (M = contents of RAM[A])
        instruction[16], // Instruction for execution
        reset;           // Signals whether to re-start the current
                         // program (reset=1) or continue executing
                         // the current program (reset=0).

    OUT outM[16],        // M value output
        writeM,          // Write into M?
        addressM[15],    // Address in data memory (of M)
        pc[15];          // address of next instruction

    PARTS:
    // Determine the instruction type A or C
    Mux16(a=instruction ,b=exALUout ,sel=instruction[15] ,out=inA);

    /**
    * A-register will load the instruction if it is:
    * (1) an A-instruction
    * (2) a C-instruction that stores the value in A (d1=1)
    */

    //Determine load status of A-register:
    Not(in=instruction[15] ,out=negateType);
    Or(a=negateType, b=instruction[5] ,out=loadA);
    ARegister(in=inA ,load=loadA ,out=outA ,out[0..14]=addressM);

    //writeM assertion (depends on ins[15] & d3)
    And(a=instruction[15] ,b=instruction[3] ,out=writeM);

    /**
    * exALU takes two 16-bit registers:
    * (1) D-register, depends on (ins[15] & d2)
    * (2) A-register or data memory, depends on (ins[15] & a)
    */

    //Determine the x-input of the exALU:
    And(a=instruction[15] ,b=instruction[4] ,out=loadD);
    DRegister(in=exALUout ,load=loadD ,out=exALUinX);

    //Determine the y-input of the exALU:
    And(a=instruction[15] ,b=instruction[12] ,out=loadY);
    Mux16(a=outA ,b=inM ,sel=loadY ,out=exALUinY);

    //prepare the extended ALU instructoin[7..8]
    Mux(a=true ,b=instruction[14], sel=instruction[15], out=ins8);
    Mux(a=true ,b=instruction[13], sel=instruction[15], out=ins7);

    // pass input to the extended ALU
    ExtendAlu(x=exALUinX ,y=exALUinY ,instruction[8]=ins8 ,instruction[7]=ins7 ,instruction[0..6]=instruction[6..12] ,out=ou

    /**
    * whether to jump or not depents (beside instruction type) on the
    * existence of one of the following:
    * (1) j1=1 and exALUneg=1
    * (2) j2=1 and exALUzr=1
    * (3) j3=1 and not (exALUneg=1 or exALUzr=1)
    */

    // calculate the jump condition
    Or(a=exALUzr ,b=exALUneg ,out=notpositive);
    Not(in=notpositive ,out=exALUpos);

    And(a=instruction[2] ,b=exALUneg ,out=j1);
    And(a=instruction[1] ,b=exALUzr ,out=j2);
    And(a=instruction[0] ,b=exALUpos ,out=j3);

    Or(a=j1 ,b=j2 ,out=j1orj2);
    Or(a=j1orj2 ,b=j3 ,out=jOrs);

    And(a=jOrs ,b=instruction[15] ,out=jump);

    /**
    * in the programe counter increment is always true because
    * if no reset or jump the PC must be incremented
    */

    // set the programe counter
    PC(in=outA ,load=jump ,inc=true ,reset=reset ,out[0..14]=pc);

}
```

6 ExtendAlu.hdl

```
/**
* The inputs of the extends ALU is instruction[9], x[16] and y[16].
* The output is define as follows:
* 1. If instruction[7..8]=1,1 the output is identical to the regular
* ALU, where instruction[5]=zx, instruction[4]=nx, ..., instruction[0]=no.
*
* 2. Else, if instruction[7]=0 the output will be x*y. Disregard the rest
* of the instruction.
*
* 3. Else, if instruction[8]=0 the output will be a shift.
*    If instruction[4]=0 shift y, otherwise shift x, moreover if
*    instruction[5]=0 return a right shift, otherwise left shift.
*
* instruction[6] is unused.
* The ng and zr pins behave the same as in the regular ALU.
**/


CHIP ExtendAlu{
     IN x[16],y[16],instruction[9];
     OUT out[16],zr,ng;

     PARTS:
     // Regular ALU result
     ALU(x=x ,y=y ,zx=instruction[5] ,nx=instruction[4] ,zy=instruction[3] ,ny=instruction[2] ,f=instruction[1] ,no=instruct

     // Multiplication
     Mul(a=x ,b=y ,out=product);

     // Shift Y
     ShiftRight(in=y ,out=RShiftY);
     ShiftLeft(in=y ,out=LShiftY);

     // Shift X
     ShiftRight(in=x ,out=RShiftX);
     ShiftLeft(in=x ,out=LShiftX);

     // select the shift output
     Mux4Way16(a=RShiftY ,b=RShiftX ,c=LShiftY ,d=LShiftX ,sel=instruction[4..5] ,out=shifted);

     // select the final output, and determine if (out < 0)
     Mux16(a=product ,b=ALUout ,sel=instruction[7] ,out=result);
     Mux16(a=shifted ,b=result ,sel=instruction[8] ,out[15]=ng ,out[0..7]=temp1, out[8..15]=temp2 ,out=out);

     // determine if (out == 0)
     Or8Way(in=temp1 ,out=zrRes1);
     Or8Way(in=temp2 ,out=zrRes2);
     Or(a=zrRes1 ,b=zrRes2, out=Notzr);
     Not(in=Notzr ,out=zr);

}
```

7 Memory.hdl

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/05/Memory.hdl

/**
 * The complete address space of the Hack computer's memory,
 * including RAM and memory-mapped I/O.
 * The chip facilitates read and write operations, as follows:
 *     Read:  out(t) = Memory[address(t)](t)
 *     Write: if load(t-1) then Memory[address(t-1)](t) = in(t-1)
 * In words: the chip always outputs the value stored at the memory
 * location specified by address. If load==1, the in value is loaded
 * into the memory location specified by address. This value becomes
 * available through the out output from the next time step onward.
 * Address space rules:
 * Only the upper 16K+8K+1 words of the Memory chip are used.
 * Access to address>0x6000 is invalid. Access to any address in
 * the range 0x4000-0x5FFF results in accessing the screen memory
 * map. Access to address 0x6000 results in accessing the keyboard
 * memory map. The behavior in these addresses is described in the
 * Screen and Keyboard chip specifications given in the book.
 */

CHIP Memory {
    IN in[16], load, address[15];
    OUT out[16];

    PARTS:
    // Determine which part of Memory to write to (KBD is read-only)
    DMux(in=load ,sel=address[14] ,a=ram ,b=screen);

    // Pass input to RAM and Screen
    RAM16K(in=in ,load=ram ,address=address[0..13] ,out=ramOut);
    Screen(in=in ,load=screen ,address=address[0..12] ,out=scrOut);

    // Get the KBD output
    Keyboard(out=kbdOut);

    // Select the right final output
    Mux4Way16(a=ramOut ,b=ramOut ,c=scrOut ,d=kbdOut ,sel=address[13..14] ,out=out);
}
```