# Contents

# 1 Basic Test Results

```
********** FOLDER STRUCTURE TEST START **********
Extracting submission...
     Extracted zip successfully

Finding usernames...
     Submission logins are: nogafri
     Is this OK?

Checking for non-ASCII characters with the command 'grep -IHPnsr [^\x00-\x7F] <dir>' ...
     No invalid characters found.

**********  FOLDER STRUCTURE TEST END  **********


********** PROJECT TEST START **********
Running CPU test:
     CPU chip passed.

Running ComputerRect test:
     Computer chip passed.

Running Memory test:
     Memory chip passed.

Running ComputerMax test:
     Computer chip passed.

Running ComputerAdd test:
     Computer chip passed.

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


****************************************************
**********   PRESUBMISSION TESTS PASSED   **********
****************************************************

Note: the tests you see above are all the presubmission tests
for this project. The tests might not check all the different
parts of the project or all corner cases, so write your own
tests and use them!
```

## 2 AUTHORS

```
nogafri
Partner 1: Noga Friedman, noga.fri@mail.huji.ac.il, 209010479
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:

    // instruction[0..2] - Jump bits
    // instruction[3..5] - Destination load bits
    // instruction[6..11] - C instruction (ALU control bits)
    // instruction[12] - if a==0 A output, if a==1 inM (ALU control bits)
    // instruction[15] - Instruction type (1 for C, 0 for A)

    // get instruction type:
    Not(in=instruction[15], out= aInstruction); // True when instruction[15] == 0
    Not(in=aInstruction, out= cInstruction); // True when instruction[15] == 1

    // A register load:
    Or(a=aInstruction, b=instruction[5], out=ALoad);
    ARegister(in=Amux, load=ALoad, out=AOut, out[0..14]=addressM); // loads if instruction[15] == 0 or instruction[5] == 1

    // D register load:
    And(a=cInstruction, b=instruction[4], out=DLoad);
    DRegister(in=aluOut, load=DLoad, out=DOut);  // loads if instruction[4] == 1 and instruction[15] == 1

    And(a=cInstruction, b=instruction[3], out=writeM);

    // data transfer logic:
    Mux16(a=aluOut, b=instruction, sel=aInstruction, out=Amux);  // outputs instruction or ALU to A Register input
    Mux16(a=AOut, b=inM, sel=instruction[12], out=ALUmux);  // outputs A or M to ALU input

    ALU(x=DOut, y=ALUmux, zx=instruction[11], nx=instruction[10], zy=instruction[9], ny=instruction[8], f=instruction[7], nc

    // jump logic:
    Not(in=ALUng, out=zeropos);
    Not(in=ALUzr, out=notzero);
    And(a=zeropos, b=notzero, out=ALUpos);

    And(a=instruction[0], b=ALUpos, out=jpos);
    And(a=jpos, b=cInstruction, out=JGT);

    And(a=instruction[1], b=ALUzr, out=jzero);
    And(a=jzero, b=cInstruction, out=JEQ);

    And(a=instruction[2], b=ALUng, out=jneg);
    And(a=jneg, b=cInstruction, out=JLT);

    Or(a=JEQ, b=JLT, out=JLE);
    Or(a=JLE, b=JGT, out=jump);
    PC(in=AOut, 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=outmem, instruction=instruction, reset=reset, outM=inMem, writeM=loadMem, addressM=addressMem, pc=pc);
    Memory(in=inMem, load=loadMem, address=addressMem, out=outmem);
}
```

# 5 CpuMul.hdl

```
// This file is part of nand2tetris, as taught in The Hebrew University, and
// was written by Aviv Yaish. It is an extension to the specifications given
// [here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
// as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
// Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).

// This chip is an extension of the regular CPU that uses the extended ALU.
// If instruction[15]==0 or (instruction[14]==1 and instruction[13]==1),
// then CpuMul behaves exactly the same as the regular CPU.
// If instruction[15]==1 and instruction[14]==0 the chip will behave as follows:
// | Instruction           | 15 | 14 | 13 | a | c1 | c2 | c3 | c4 | c5 | c6 |
// |-----------------------|:--:|:--:|:--:|:-:|:--:|:--:|:--:|:--:|:--:|:--:|
// | Regular a-instruction |  0 |  * |  * | * |  * |  * |  * |  * |  * |  * |
// | Regular c-instruction |  1 |  1 |  1 | * |  * |  * |  * |  * |  * |  * |
// | dest=A<<;jump         |  1 |  0 |  1 | 0 |  1 |  0 |  0 |  0 |  0 |  0 |
// | dest=D<<;jump         |  1 |  0 |  1 | 0 |  1 |  1 |  0 |  0 |  0 |  0 |
// | dest=M<<;jump         |  1 |  0 |  1 | 1 |  1 |  0 |  0 |  0 |  0 |  0 |
// | dest=A>>;jump         |  1 |  0 |  1 | 0 |  0 |  0 |  0 |  0 |  0 |  0 |
// | dest=D>>;jump         |  1 |  0 |  1 | 0 |  0 |  1 |  0 |  0 |  0 |  0 |
// | dest=M>>;jump         |  1 |  0 |  1 | 1 |  0 |  0 |  0 |  0 |  0 |  0 |
// Where:
// - "<<" is a left shift, and ">>" is a right shift, as defined in project 2.
//   These notations were chosen because they are used in real programming
//   languages.
// - dest and jump can take the same values as in the regular CPU.

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:
    // get instruction type:
    Not(in=instruction[15], out= aInstruction); // True when instruction[15] == 0
    Not(in=aInstruction, out= cInstruction); // True when instruction[15] == 1

    // A register load:
    Or(a=aInstruction, b=instruction[5], out=ALoad);
    ARegister(in=Amux, load=ALoad, out=AOut, out[0..14]=addressM); // loads if instruction[15] == 0 or instruction[5] == 1

    // D register load:
    And(a=cInstruction, b=instruction[4], out=DLoad);
    DRegister(in=aluOut, load=DLoad, out=DOut);  // loads if instruction[4] == 1 and instruction[15] == 1

    And(a=cInstruction, b=instruction[3], out=writeM);

    // data transfer logic:
    Mux16(a=aluOut, b=instruction, sel=aInstruction, out=Amux);  // outputs instruction or ALU to A Register input
    Mux16(a=AOut, b=inM, sel=instruction[12], out=ALUmux);  // outputs A or M to ALU input

    ExtendAlu(x=DOut, y=ALUmux, instruction[0..5]=instruction[6..11], instruction[6]=false, instruction[7]=instruction[15],

    // jump logic:
    Not(in=ALUng, out=zeropos);
    Not(in=ALUzr, out=notzero);
    And(a=zeropos, b=notzero, out=ALUpos);

    And(a=instruction[0], b=ALUpos, out=jpos);
    And(a=jpos, b=cInstruction, out=JGT);

    And(a=instruction[1], b=ALUzr, out=jzero);
    And(a=jzero, b=cInstruction, out=JEQ);

    And(a=instruction[2], b=ALUng, out=jneg);
    And(a=jneg, b=cInstruction, out=JLT);

    Or(a=JEQ, b=JLT, out=JLE);
    Or(a=JLE, b=JGT, out=jump);
    PC(in=AOut, load=jump, inc=true, reset=reset, out[0..14]=pc);
}
```

# 6 ExtendAlu.hdl

```
// This file is part of nand2tetris, as taught in The Hebrew University, and
// was written by Aviv Yaish. It is an extension to the specifications given
// [here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
// as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
// Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).


// The ExtendAlu chip is an extension of the standard ALU which also supports
// shift operations.
// The inputs of the extended ALU are instruction[9], x[16], y[16].
// The "ng" and "zr" output pins behave the same as in the regular ALU.
// The "out" output is defined as follows:
// If instruction[8]=1 and instruction[7]=1 the output is identical to the
// regular ALU, where:
// instruction[5]=zx, instruction[4]=nx, ..., instruction[0]=no
// Else, if instruction[8]=0 and instruction[7]=1, the output is a shift:
// - If instruction[4] == 0, the input "y" will be shifted, otherwise "x".
// - If instruction[5] == 0, the shift will be a right-shift, otherwise left.
// - All other inputs are undefined.

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

     PARTS:
     // out:
     And(a=instruction[8], b=instruction[7], out=ogALU);  // og ALU if True

     Not(in=instruction[8], out=not8);
     And(a=not8, b=instruction[7], out=isShift);  // Extended ALU if True

     // if no shift (original ALU):
     ALU(x=x, y=y, zx=instruction[5], nx=instruction[4], zy=instruction[3], ny=instruction[2], f=instruction[1], no=instruct

     // if shift:
     Mux16(a=y, b=x, sel=instruction[4], out=toshift);  // determine which input to shift

     ShiftRight(in=toshift, out=shiftedRight);
     ShiftLeft(in=toshift, out=shiftedLeft);
     Mux16(a=shiftedRight, b=shiftedLeft, sel=instruction[5], out=shifted);  // outputs the shifted object in the requested

     // output the shifted object or the original-ALU output object:
     Mux16(a=aluOut, b=shifted, sel=isShift, out=out, out[0..7]=out1, out[8..15]=out2, out[15]=allout);

     // zr:
     Or8Way(in=out1, out=out1res);
     Or8Way(in=out2, out=out2res);
     Or(a=out1res, b=out2res, out=outres);
     Mux(a=true, b=false, sel=outres, out=zr);

     // ng:
     FullAdder(a=allout, b=true, sum=overflow, carry=ng);
}
```

# 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:
    DMux4Way(in=load, sel=address[13..14], a=ram1, b=ram2, c=screenLoad, d=kbd);
    Or(a=ram1, b=ram2, out=ramLoad);

    RAM16K(in=in, load=ramLoad, address=address[0..13], out=ramOut);
    Screen(in=in, load=screenLoad, address=address[0..12], out=screenOut);
    Keyboard(out=kbdOut);

    // out=ramOut if address[13..14] in {00, 01}
    // out=screenOut if address[13..14] in {10}
    // out=kbdOut if address[13..14] in {11}
    Mux4Way16(a=ramOut, b=ramOut, c=screenOut, d=kbdOut, sel=address[13..14], out=out);
}
```