Contents

1 Basic Test Results

```
********** TEST START **********

  preparing sub.tar
dos2unix: converting file /tmp/bodek.L6reHy/nand2tet/Project05/eliavsd/presubmission/testdir/stud/sub.tar/README to Unix for
  checking sub.tar
CPU passed
Memory passed
CpuMul passed
ExtendAlu passed
Computer passed
Computer passed
Computer passed

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

2 README

```
eliavsd,avikor13
==============================================================================
Eliav Shames, ID 317390268, eliavsd@cs.huji.ac.il
Avi Korzec, ID 305088106, avikor13@cs.huji.ac.il
==============================================================================

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


Submitted Files
---------------
README - This file.
Memory.hdl - The chip that simulates the memory in the hack machine
CPU.hdl - The chip that simulates the CPU in the hack machine
Computer.hdl - The chip that simulates the hack machines topmost chip
ExtendAlu.hdl - An extension of the ALU that supports additional operations
CpuMul.hdl - This chip is an extension of the book CPU by using the extended ALU


Additional Files
----------------

A 3-way multiplexor was created to allow the ExtendALU chip to disregard
instruction[8] if instruction[7] is zero.

In addition, an Or16Way chip was built, based on the Or8Way chip designed
in project 1 to allow us to easily compare a 16-bit input to zero.
```

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:
    // if instruction[15] == 0 then a-instruction (e.g, 0vvv...v)
    // need to load the instruction to the A Register, therefore Priority(aInst) > Priority(ALUout)
    Not(in=instruction[15], out=aInst);
    Mux16(a=ALUout, b=instruction, sel=aInst, out=ALUxorAinst);
    // Even if not ab a-instruction, we still want to write to ARegister if a c-instruction needs to
    Or(a=aInst, b=instruction[5], out=loadA);
    ARegister(in=ALUxorAinst, load=loadA, out=Aout, out[0..14]=addressM);

    // otherwise c-instruction (e.g, 111ac1c2c3c4c5c6d1d2d3j1j2j3)
    // ALU - What to compute? (Inputs) Where to store output? (Writing) What to do next? (Jumping)
    //   ALU - Inputs: D Register, M/A Register, control bits
    //   ALU - Inputs: D Register
    And(a=instruction[15], b=instruction[4], out=writeToD);
    DRegister(in=ALUout, load=writeToD, out=Dout);

    //   ALU - Inputs: M or A Register. Deciding using the a-bit.
    And(a=instruction[15], b=instruction[12], out=aBit);
    Mux16(a=Aout, b=inM, sel=aBit, out=AxorMout);
```

```
60
61      // ALU Computing: using c1 to c6 bits.
62      ALU(x=Dout, y=AxorMout, zx=instruction[11], nx=instruction[10], zy=instruction[9],
63          ny=instruction[8], f=instruction[7], no=instruction[6], zr=zr, ng=ng, out=outM, out=ALUout);
64
65      //   Jumping (instruction[0..2]): 000 - no jump, 001 - JGT, 010 - JEQ, 011 - JGE, 100 - JLT,
66      //            101 - JNE, 110 - JLE, 111 - JMP. Controlled using d1 to d3 and zr, ng.
67      And(a=instruction[15], b=instruction[0], out=JGT);
68      And(a=instruction[15], b=instruction[1], out=JEQ);
69      And(a=instruction[15], b=instruction[2], out=JLT);
70      Not(in=ng, out=pos);
71      Not(in=zr, out=notZR);
72      And(a=pos, b=notZR, out=res1);
73      And(a=JGT, b=res1, out=res2);
74      And(a=JEQ, b=zr, out=res3);
75      And(a=JLT, b=ng, out=res4);
76      Or(a=res2, b=res3, out=res5);
77      Or(a=res4, b=res5, out=JMP);
78
79      // if its a c-instruction and instruction[3] == 1 then write to M
80      And(a=instruction[15], b=instruction[3], out=writeM);
81
82     // PC handling.
83     PC(in=Aout, inc=true, load=JMP, reset=reset, out[0..14]=pc);
84  }
```

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=inst);
    Memory(in=ValToStore, load=bool, address=addr, out=data);
    CPU(inM=data, instruction=inst, reset=reset, outM=ValToStore, writeM=bool, addressM=addr, pc=pc);
}
```

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
* (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:
    // Extract the different bits from the instruction code
    And16(a=instruction, b=true, out[0]=j3, out[1]=j2, out[2]=j1,
                          out[3]=d3, out[4]=d2, out[5]=d1,
                          out[6]=c6, out[7]=c5, out[8]=c4,
                          out[9]=c3, out[10]=c2, out[11]=c1,
                          out[12]=aBit, out[15]=aOrC);

    Not(in=aOrC, out=aInstruction);
    // If instruction[15] = 0 then we're processing an address, otherwise
    // We're processing a C-instruction. Store the result as an input to A
    Mux16(a=instruction, b=aluOut, sel=aOrC, out=aInput);
    // If instruction[15] = 0 or d1(instruction[5])=1 then we know A is a destination (loadA=1).
    Mux(a=true, b=d1, sel=aOrC, out=loadA);
    // Load the aInput into the A register if loadA=1
    ARegister(in=aInput, load=loadA, out=aOut);

    // If instruction[15] (is_C_Instruction)=1 and d3=1 we will be writing to M
    And(a=aOrC, b=d3, out=writeM);

    // If we are performing any one of the ALU operations we send the instruction
    // To the extended ALU we designed
    // If bits 13 and 14 are 1 then we know we'll be behaving like the book CPU
    // And ALU.
    And(a=instruction[14], b=instruction[13], out=bookCPU);

    // If we are processing an A-instruction, then x*y will not be performed
    // (alu_instruction[7]=1 --> the ALU will not perform x*y)
```

```
60      // Otherwise, refer to instruction[13] to decide
61      Mux(a=instruction[13], b=true, sel=aInstruction, out=instBit7);
62
63      // If we behave like the book CPU or we have an A-Instruction, then the ExtendALU
64      // Chip should either perform x*y or behave like the book ALU (depending on the previous (7th bit))
65      Or(a=bookCPU, b=aInstruction, out=instBit8);
66
67      // Use the A-bit to decide whether the A or M register value is sent to the ALU
68      Mux16(a=aOut, b=inM, sel=aBit, out=sendToALU);
69
70      // Finally we send the D register data and the A/M register data to the ALU
71      // With the appropriate instructions
72      ExtendAlu(x=dOutput, y=sendToALU, instruction[8]=instBit8,
73       instruction[7]=instBit7, instruction[6]=false,
74       instruction[0..5]=instruction[6..11], out=aluOut, zr=zr, ng=ng);
75
76      // The ALU has finished running, do we store the result in D?
77      And(a=aOrC, b=d2, out=loadD);
78      // If true, store the ALU output in D
79      DRegister(in=aluOut, load=loadD, out=dOutput);
80
81      // Here we deal with all the jump cases in order to decide what
82      // PC needs to do.
83
84      //Not Zero?
85      Not(in=zr, out=nzr);
86      // Non-negative (JGE)?
87      Not(in=ng, out=nng);
88      // Not zero and non-negative (positive GT)?
89      And(a=nzr, b=nng, out=pos);
90      // j1 + negative = JLT
91      And(a=j1, b=ng, out=JLT);
92      // j2 + equaltozero = JEQ
93      And(a=j2, b=zr, out=JEQ);
94      // j3 + positive = JGT
95      And(a=j3, b=pos, out=JGT);
96      // JLT || JEQ = JLE
97      Or(a=JLT, b=JEQ, out=JLE);
98      // JLE || JGT = JMP unconditional
99      Or(a=JLE, b=JGT, out=JMP);
100     // C-Instruction + JMP = we will jump
101     And(a=aOrC, b=JMP, out=isJump);
102
103     //OUTPUTS
104     // Output value of M
105     And16(a=aluOut, b=true, out=outM);
106     // Output address of M
107     And16(a=aOut, b=true, out[0..14]=addressM);
108     // Send A (register value or jump address) + jump bit to pc
109     PC(in=aOut, load=isJump, inc=true, reset=reset, out[0..14]=pc);
110     }
```

6 ExtendAlu.hdl

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

     PARTS:
     // First we'll calculate each result seperately and then
     // We'll use a custom 3-way Mux to decide which one is the output

     // First we'll multiply
     Mul(a=x, b=y, out=mulOut);

     // Now we'll calculate the 4 possible shifts and use a 4-way Mux to
     // Determine which one we may have been instructed to calculate
     ShiftLeft(in=x, out=xLeft);
     ShiftRight(in=x, out=xRight);
     ShiftLeft(in=y, out=yLeft);
     ShiftRight(in=y, out=yRight);

     // If instruction[4]=0 we shift Y, otherwise X. if instruction[5]=0 we
     // Shift right, otherwise left.
     // After this operation, if the instruction is a shift, shiftOut will
     // Contain the correct shift output
     Mux4Way16(a=yRight, b=xRight, c=yLeft, d=xLeft, sel=instruction[4..5],
      out=shiftOut);

     //Calculate the ALU output
     ALU(x=x, y=y, zx=instruction[5], nx=instruction[4], zy=instruction[3],
      ny=instruction[2], f=instruction[1], no=instruction[0], out=aluOut,
       zr=aluZr, ng=aluNg);

     // Here we determine whether we were instructed to calculate
     // alu,mul or shift
     Mux3Way16(a=mulOut, b=shiftOut, c=aluOut, sel=instruction[7..8],
     out[15]=ng, out=tmpOut, out=out);

     // If there is a non-zero bit in tmpOut, then zr=false
     Or16Way(in=tmpOut, 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:
    // if address[14] == 1 (address is eq/bg than 16384) then handle Screen
    // otherwise handle RAM16K
    DMux(in=load, sel=address[14], a=RAMbit, b=SCRbit);
    RAM16K(in=in, load=RAMbit, address=address[0..13], out=RAMout);
    Screen(in=in, load=SCRbit, address=address[0..12], out=SCRout);
    Keyboard(out=KBDout);
    // if address[13..14] == 00 or 01 then we've handled RAM16K
    // if address[13..14] == 10 then we've handled Screen
    // if address[13..14] == 11 then we've handled Keyboard (1100000000000000)
    Mux4Way16(a=RAMout, b=RAMout, c=SCRout, d=KBDout, sel=address[13..14], out=out);
}
```

8 Mux3Way16.hdl

```
/**
 * Custom 3-way 16-bit multiplexor chip:
 * out = a if sel == 10 or 00 (mul)
 *       b if sel == 01 (shift)
 *       c if sel == 11 (alu)
 */

CHIP Mux3Way16 {
    IN a[16], b[16], c[16], sel[2];
    OUT out[16];

    PARTS:
    Mux16 (a=b, b=c, sel=sel[1], out=bOrC);
    Mux16 (a=a, b=bOrC, sel=sel[0], out=out);
}
```

9 Or16Way.hdl

```
/**
 * 16-way or gate: out = in[0] or in[1] or ... or in[15].
 * Based on the Or8Way chip which was also built in project 1
 */
CHIP Or16Way {
    IN in[16];
    OUT out;

    PARTS:

    Or8Way(in=in[0..7], out=half1);
    Or8Way(in=in[8..15], out=half2);

    // in[0..7] or in[8..15]
    Or(a=half1, b=half2, out=out);

}
```