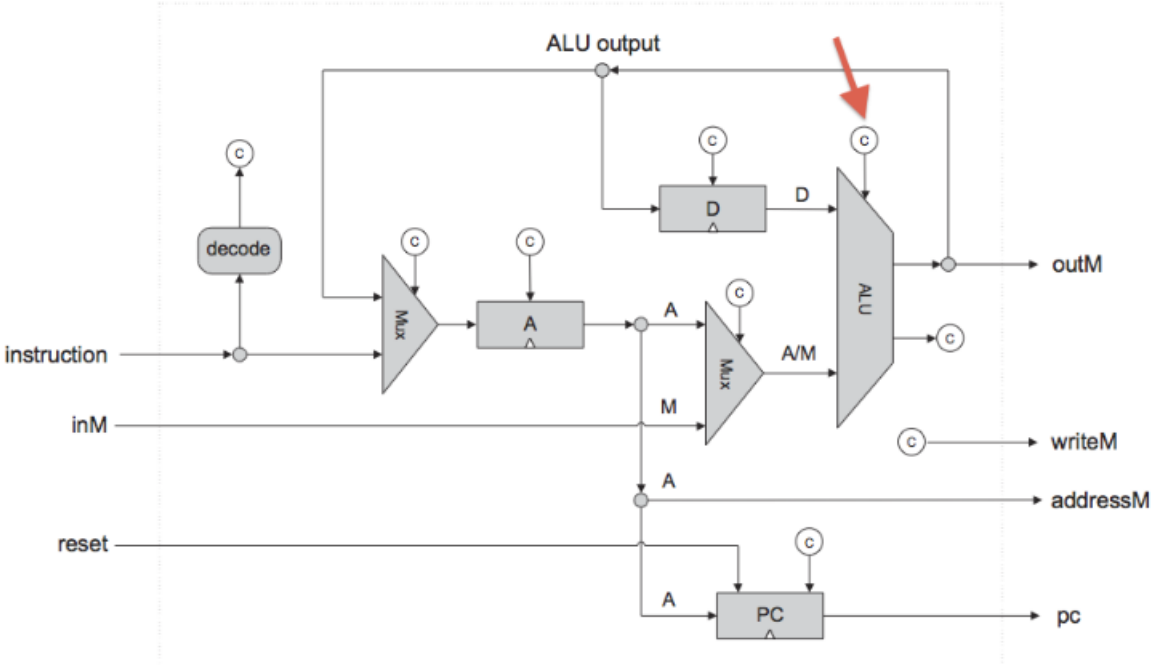
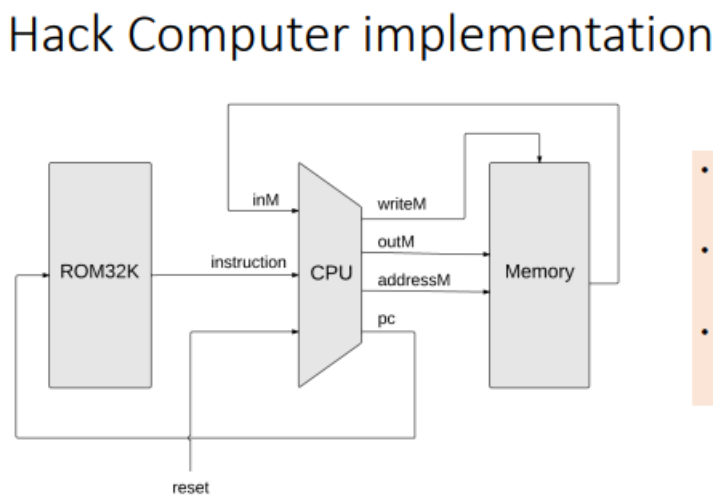
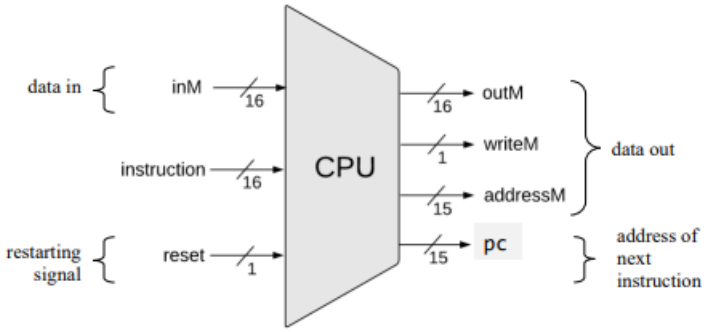


This diagram represents the proposed design for the CPU architecture that I will use to explain and implement the CPU's HDL code. The CPU is made up of a number of logic gates including the multiplexer ALU A-register and D-register and many more. To begin with let's start with an abstraction to this chip by hiding all the logic gates in the diagram so that we can quickly identify the input and output pins of the CPU.



Before taking an idepth look inside the cpu, lets first see the overall hack computer implementation for better understanding. In this diagram the CPU is designed to take an instruction from ROM32K, process it, and interact with Memory accordingly. It takes input 'inM' and outputs 'outM', 'writeM', and 'addressM'. program counter keeps track of where in the program we are currently executing. 2. - The image shows a schematic representation of Hack Computer implementation. - There are three main components labeled: ROM32K, CPU, and Memory. - Arrows indicate data flow between these components. - ROM32K sends an "instruction" to CPU. - CPU has four outputs: "outM", "writeM", "addressM", and "pc" connected to Memory. - There's also an "inM" input into the CPU which likely comes from Memory (though it's not explicitly shown). - A "reset" line is connected to both ROM32K and CPU indicating system or program reset functionality.

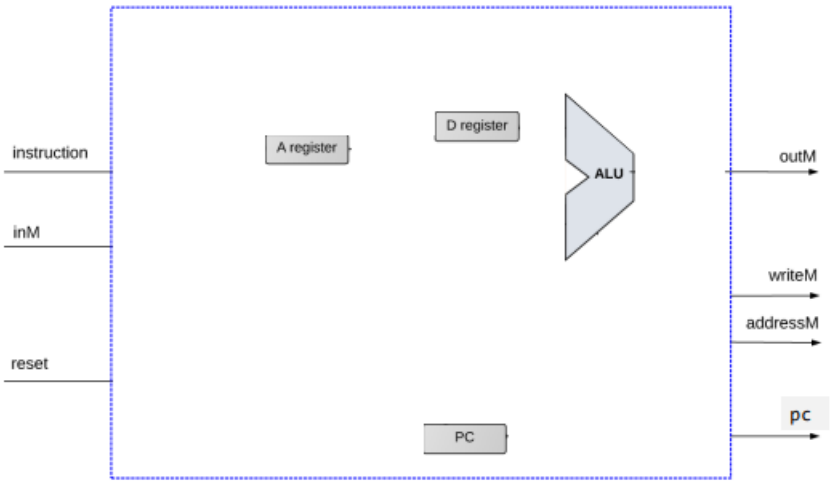
Hack CPU Interface



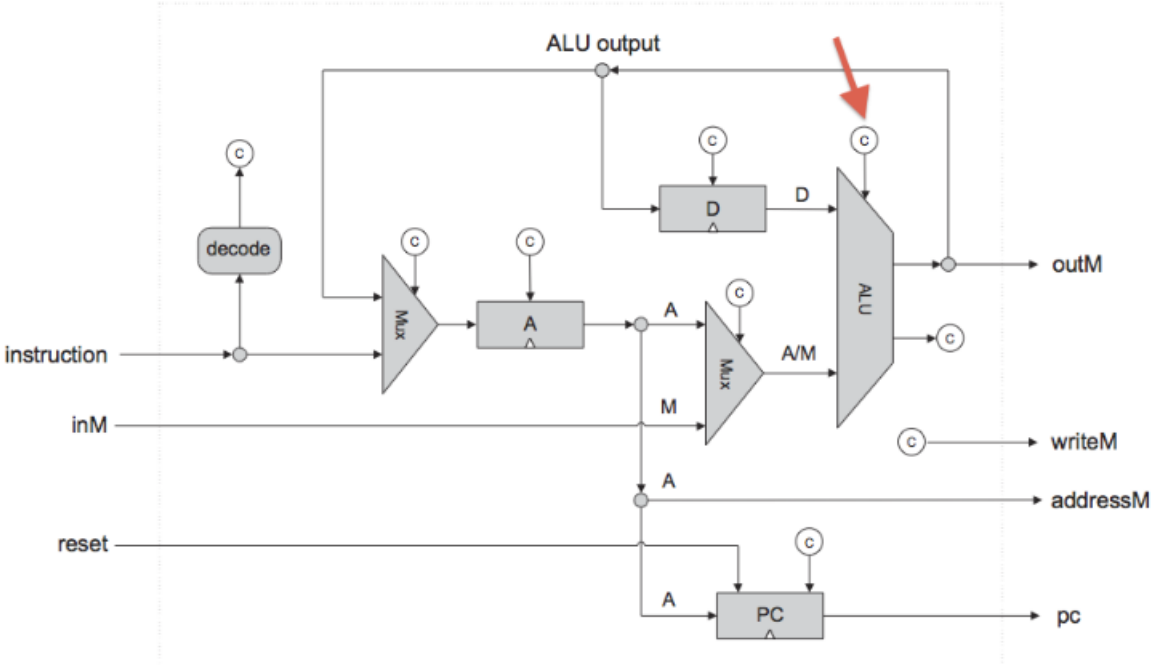
In this diagram we can see the interface of the hack cpu.

Now that we can clearly see the inputs and outputs of the CPU as shown in this diagram, there are three inputs coming into the CPU from different sources, the instruction memory and the data memory. The instruction input comes from the instruction memory which is the value of the selected instruction memory. The instruction can either be A- instruction or C-instruction. On the other hand, M is the value of the currently selected data memory register and is the value that the CPU used to perform its operations

Hack CPU Implementation



Now let's take a look at the key components and input/outputs of the Hack CPU discussed so far. The question is how to interconnect them to make the CPU operate as required.



Now this is the main implementation inside the CPU. As shown in the first slide,

This image is a schematic diagram of a CPU architecture. - It shows various components and their connections, including "decode", "ALU output", "outM", "writeM", "addressM", and "pc". - The “instruction” input is connected to a “decode” block that interprets the instruction. - There are three primary inputs labeled as “instruction”, “inM”, and “reset”. - The ALU (Arithmetic Logic Unit) is central in this architecture, with inputs from A register, D register, and M register. - There are several control bits around the ALU that determine its operation based on the decoded instruction. - Outputs include ALU output, outM which appears to be data output, writeM for writing data into memory, addressM for memory addressing and pc for program counter updates.

Now that we have identified the input and the outputs of the CPU architecture, we can now implement the code of the CPU and discuss how it works * open the next slide I’ve broken down the code into parts so that it would be easy to explain and understand how the code works

```
IN  inM[16],       // M value input  (M = contents of RAM[A])
    instruction[16], // Instruction for execution
    reset;         // Signals whether to restart 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
```

These lines define the inputs and outputs to the CPU. inM is the M value input, instruction is the instruction for execution, and reset is a signal to either restart or continue the current program. outM is the M value output, writeM is a flag indicating whether to write to M, addressM is the address in data memory of M, and pc is the address of the next instruction.

If we see this code line by line, the

IN inM[16], instruction[16], reset;: This line declares input pins for the chip, namely inM (16-bit input for memory), instruction (16-bit instruction input), and reset (reset signal).

OUT outM[16], writeM, addressM[15], pc[15];: This line declares output pins for the chip, including outM (16-bit output for memory), writeM (write signal for memory), addressM (15-bit output representing the address in data memory), and pc (15-bit output representing the program counter).

```
// get type of instruction
Not(in=instruction[15], out=Ainstruction);
Not(in=Ainstruction, out=Cinstruction);
```

These lines determine the type of instruction. If the 15th bit of the instruction is 0, it's an A-instruction. If it's 1, it's a C-instruction.

If we see this code line by line, the

Not(in=instruction[15], out=Ainstruction);: Inverts the most significant bit of the instruction input and stores the result in Ainstruction.

Not(in=Ainstruction, out=Cinstruction);: Inverts the value of Ainstruction and stores the result in Cinstruction.

And(a=Cinstruction, b=instruction[5], out=ALUtoA); // C-inst and dest to A-reg?

Mux16(a=instruction, b=ALUout, sel=ALUtoA, out=Aregin);

These lines check if the destination of the C-instruction is the A-register. If it is, the output of the ALU is selected as the input to the A-register.

And(a=Cinstruction, b=instruction[5], out=ALUtoA);: Performs a bitwise AND operation between Cinstruction and the 6th bit of instruction, storing the result in ALUtoA.

Mux16(a=instruction, b=ALUout, sel=ALUtoA, out=Aregin);: Chooses between the entire instruction and the ALU output based on the value of ALUtoA, storing the result in Aregin.

Or(a=Ainstruction, b=ALUtoA, out=loadA); // load A if A-inst or C-inst&dest to A-reg

ARegister(in=Aregin, load=loadA, out=Aout);

These lines load the A-register if the instruction is an A-instruction or if the destination of the C-instruction is the A-register.

Or(a=Ainstruction, b=ALUtoA, out=loadA);: Performs a bitwise OR operation between Ainstruction and ALUtoA, storing the result in loadA.

ARegister(in=Aregin, load=loadA, out=Aout);: Updates the A-register based on the input Aregin and the load signal loadA.

Mux16(a=Aout, b=inM, sel=instruction[12], out=AMout); // select A or M based on a-bit

This line selects either the A-register output or the M input based on the a-bit of the instruction.

Mux16(a=Aout, b=inM, sel=instruction[12], out=AMout);: Chooses between the A-register output (Aout) and the memory input (inM) based on the 13th bit of instruction, storing the result in AMout.

And(a=Cinstruction, b=instruction[4], out=loadD);

DRegister(in=ALUout, load=loadD, out=Dout); // load the D register from ALU

These lines load the D-register from the ALU if the instruction is a C-instruction and the destination is the D-register.

And(a=Cinstruction, b=instruction[4], out=loadD);: Performs a bitwise AND operation between Cinstruction and the 5th bit of instruction, storing the result in loadD.

DRegister(in=ALUout, load=loadD, out=Dout);: Updates the D-register based on the input ALUout and the load signal loadD.

ALU(x=Dout, y=AMout, zx=instruction[11], nx=instruction[10],

zy=instruction[9], ny=instruction[8], f=instruction[7],

no=instruction[6], out=ALUout, zr=ZRout, ng=NGout); // calculate

This line performs the ALU operation based on the D-register output, the selected A or M value, and the control bits of the instruction.

// Set outputs for writing memory

Or16(a=false, b=Aout, out[0..14]=addressM);

Or16(a=false, b=ALUout, out=outM);

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

These lines set the outputs for writing to memory. The addressM output is set to the A-register output, the outM output is set to the ALU output, and the writeM output is set if the instruction is a C-instruction and the destination is M.

Or16(a=false, b=Aout, out[0..14]=addressM);: Performs a bitwise OR operation between false and Aout, storing the result in addressM (setting the least significant bit of addressM to 0).

Or16(a=false, b=ALUout, out=outM);: Performs a bitwise OR operation between false and ALUout, storing the result in outM (setting the least significant bit of outM to 0).

And(a=Cinstruction, b=instruction[3], out=writeM);: Performs a bitwise AND operation between Cinstruction and the 4th bit of instruction, storing the result in writeM.

// calc PCload & PCinc - whether to load PC with A reg

And(a=ZRout, b=instruction[1], out=jeq); // is zero and jump if zero

And(a=NGout, b=instruction[2], out=jlt); // is neg and jump if neg

Or(a=ZRout, b=NGout, out=zeroOrNeg);

Not(in=zeroOrNeg, out=positive); // is positive (not zero and not neg)

And(a=positive, b=instruction[0], out=jgt); // is pos and jump if pos

Or(a=jeq, b=jlt, out=jle);

Or(a=jle, b=jgt, out=jumpToA); // load PC if cond met and jump if cond

And(a=Cinstruction, b=jumpToA, out=PCload); // Only jump if C instruction

Not(in=PCload, out=PCinc); // only inc if not load

PC(in=Aout, inc=PCinc, load=PCload, reset=reset, out[0..14]=pc);

These lines calculate whether to load the PC with the A-register or increment the PC. This is based on the jump bits of the instruction and the zero and negative flags from the ALU.

And(a=ZRout, b=instruction[1], out=jeq);: Checks if the Zero flag is set (ZRout) and the 2nd bit of instruction is 1, storing the result in jeq.

And(a=NGout, b=instruction[2], out=jlt);: Checks if the Negative flag is set (NGout) and the 3rd bit of instruction is 1, storing the result in jlt.

Or(a=ZRout, b=NGout, out=zeroOrNeg);: Performs a bitwise OR operation between ZRout and NGout, storing the result in zeroOrNeg.

Not(in=zeroOrNeg, out=positive);: Inverts the value of zeroOrNeg and stores the result in positive.

And(a=positive, b=instruction[0], out=jgt);: Checks if positive is true and the least significant bit of instruction is 1, storing the result in jgt.

Or(a=jeq, b=jlt, out=jle);: Performs a bitwise OR operation between jeq and jlt, storing the result in jle.

Or(a=jle, b=jgt, out=jumpToA);: Performs a bitwise OR operation between jle and jgt, storing the result in jumpToA.

And(a=Cinstruction, b=jumpToA, out=PCload);: Performs a bitwise AND operation between Cinstruction and jumpToA, storing the result in PCload.

Not(in=PCload, out=PCinc);: Inverts the value of PCload and stores the result in PCinc.

PC(...);: Updates the program counter based on the inputs Aout, PCinc, PCload, and reset, storing the result in pc.