The beauty of trees and recursion is that you don’t need to think about the whole process and subtrees. You just implement one part and call it recursively.

Variable factor is only used when you use a variable. For example write x uses variable factor and for write f(x) we use functionfactor. Both x and f(x) are expressions and here in write statement you don’t need to be worried about function factor and variable factor.
There are many types of expressions - look at the grammar
Remember: AST only has the necessary details, it is easier to do these kind of checks by using AST
Your parser is is distinguishing when to use function factor and when to use variable factor.
In write statement, you don’t need to check the types of the expression while visiting the expression at the right hand side. Expression function will dispatch the necessary functionality

We will see how functions are implemented in the hardware level
We will have a review on how stack pointer and base pointer work.

New topic: pcode machine

Compiler is a big program that includes lots of different phases
We have processed the source code language so far
The next stage in compiler is generating code
In the class, instead of targeting a real hardware like intel, we will follow a hybrid approach: we will target pcode and we will write a virtual machine to run pcode
Like Matrix: the hardware is not real but a simulation implemented at software level. But, pcode is not aware of that.
It is just like an assembly language. Usually, assembly language is converted into machine code. Instead of that, we will write a program that directly interprets our assembly language: pcode

Code generation:

Your virtual machine has 16 registers.
What is instruction pointer and program counter.
The memory address of the instruction that you are going to run: Instruction pointer.

ln register: link register.
fp: frame pointer for functions.
sp: stack pointer.
ip: instruction pointer

The rest is general purpose registers

We have lots of different operations but some are very similar

Ershov number is helping us on how to allocate the registers for different operations.

read: take some number from input and store it in register dest.
wr: takes some register and print out their values.

dest: destination register. It usually goes first. and then we add the registers of source values which we want to apply the operations on.

For checking the conditions, we only can jump to other instructions. We cannot do anything else. For comparison we have branches for each items that we want to compare.

We can use memory like stack by using push and pop.
psh src r1: psh anything in the src on r1.

First set of artithmetic operations (add, sub etc.) takes the destion to write the result and the operands to the operations

Compare-and-branch operations: there are set of branch instructions that will jump according to some simple condition
We don’t have fancy conditions in hardware level
When we do a compare operation, some flags will be set and branch instruction will look at that flags to accomplish its task

We have an array instructions: code. Another array for memory. We have two flags: z and n. And, registers some of which are special purpose while the rest is general purpose.

Pcode is equilivantly powerful with any other programming languages we have: turing completeness

You can’t compute anything more with C that you can with pcode
movi r0 2: r0 = 2 : this will move the literal 2 to destination register r0
mov r1 r0: r1 = r0 : this will move the value inside source register r0 to destination register r1
addi r1 r1 1: r1 = r1 + 1 : add literal 1 to the value in source register r1 and write the result to destination register r1
div r2 r0 r1: r2 = r0 / r1

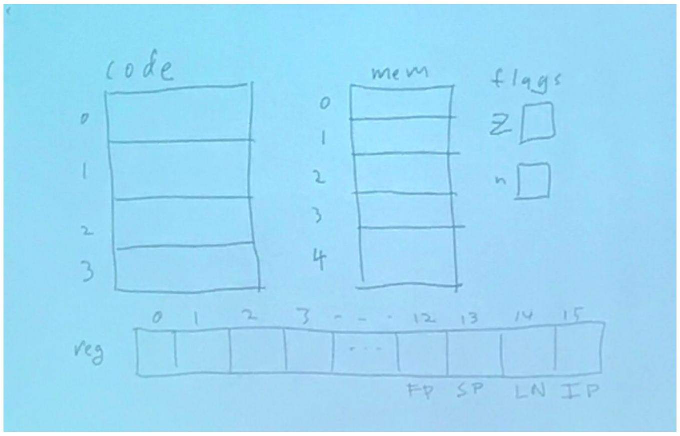
Remember fetch-execute cycle
Stored-program computers
In the code you will see a big while loop which does the fetch-execution cycle
To stop the execution of a program, we have hlt (halt) instruction
If you don’t have hlt, it will continue by reading corrupted memory

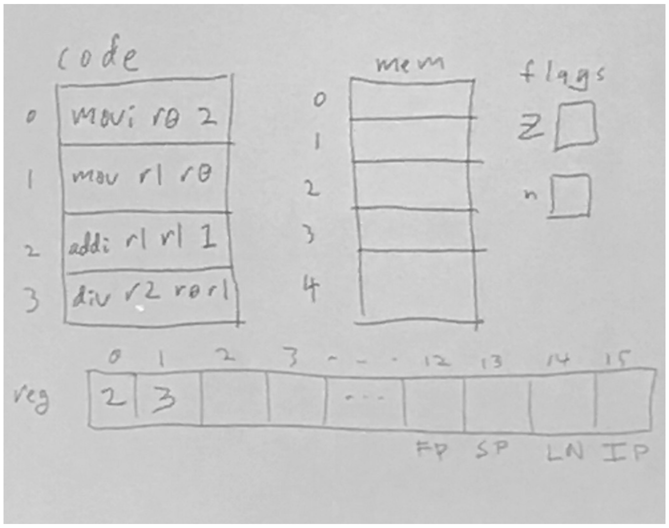
cmp r0 r1: compare registers r0 and r1. It will update the flags: z and n. z means zero. You subtract the numbers from each other and set the z flag if the result is 0. n flag is set if the first register is less than the second one. Therefore: z = (r0 - r1 == 0),   n = (r0 < r1)
beg 2: branch if equal. If the result of the comparison resulted that the numbers were equal, branch 2. 2 is a offset here. Move the instruction pointer by 2.
bne -1: branch if not equal. Move the instruction pointer by -1.
You can express all different relational operations by using just those two flags z and n.
psh r0 r13: takes the value in r13, increments the value inside r13, sets the memory location given by r13 to the value of the destination r0. r13 here is stack pointer. Pop operation does it in reverse.

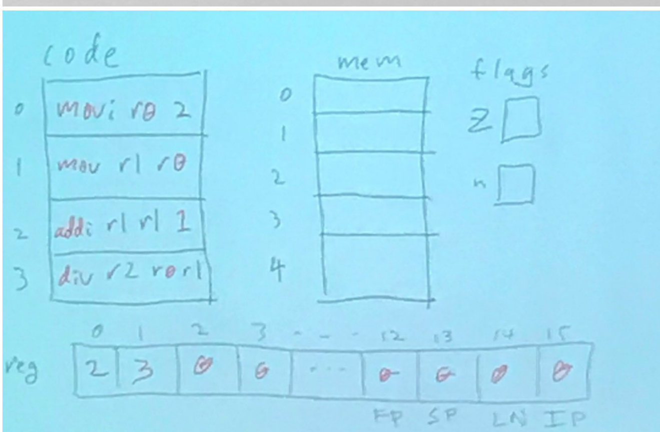
Frame pointer is store local variables for functions: we will see how it works next time

Let's take a look at some examples:

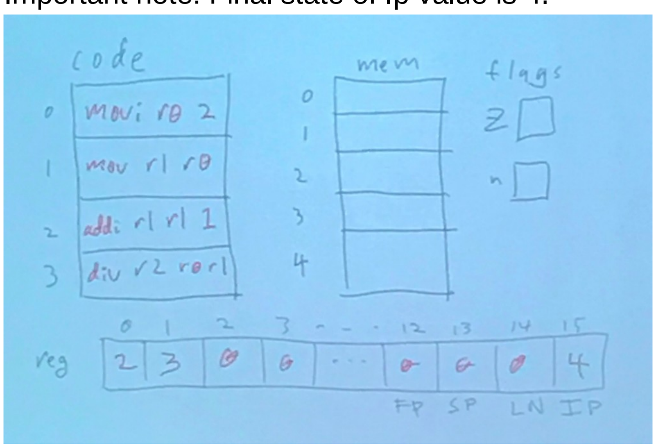
1. Arithmetic



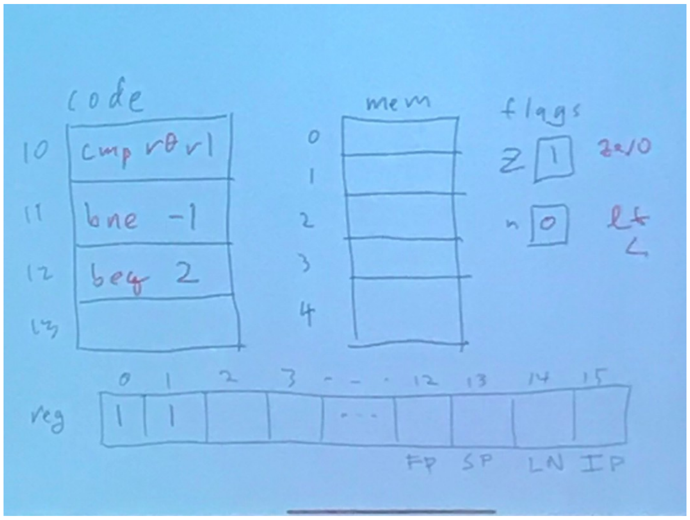




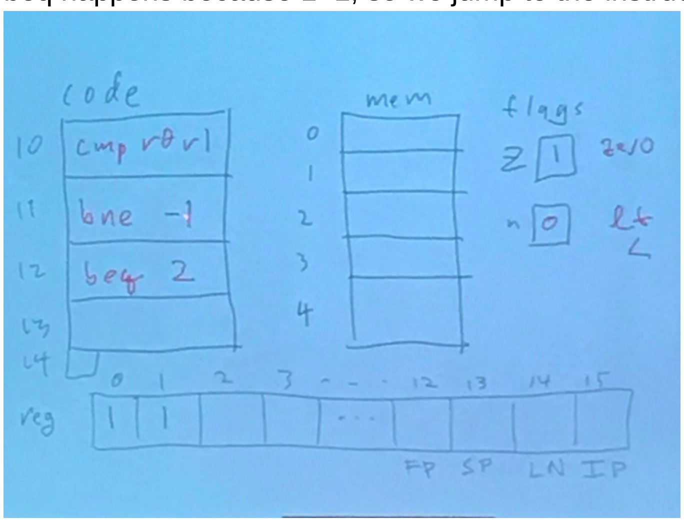
Important note: Final state of Ip value is 4.



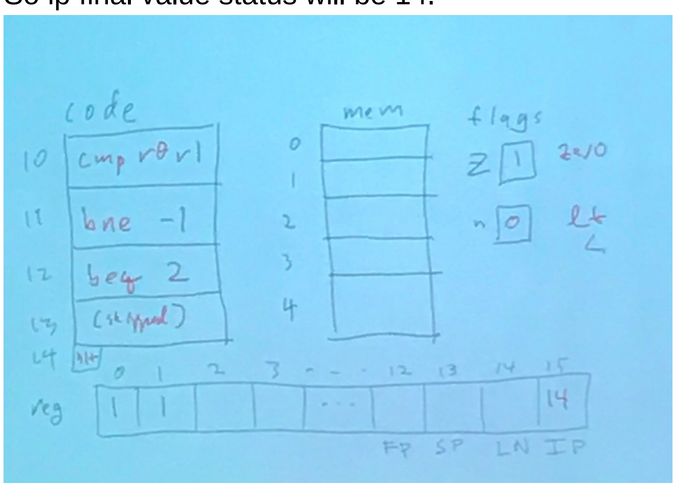
2. Comparison instruction:



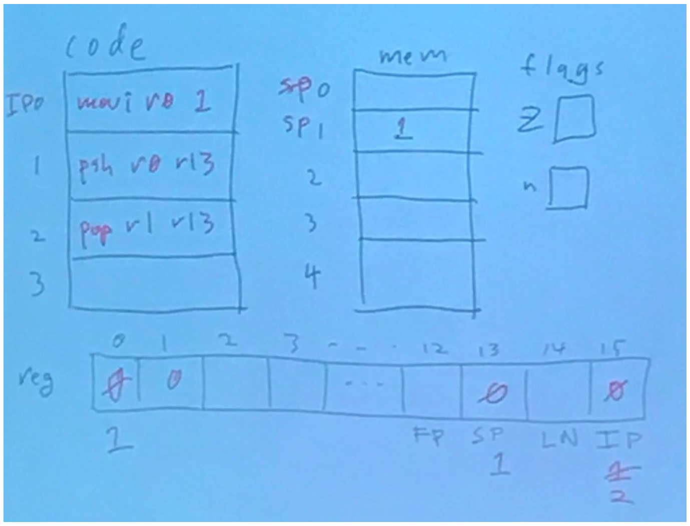
beq happens because 1=1, so we jump to the instruction 14.

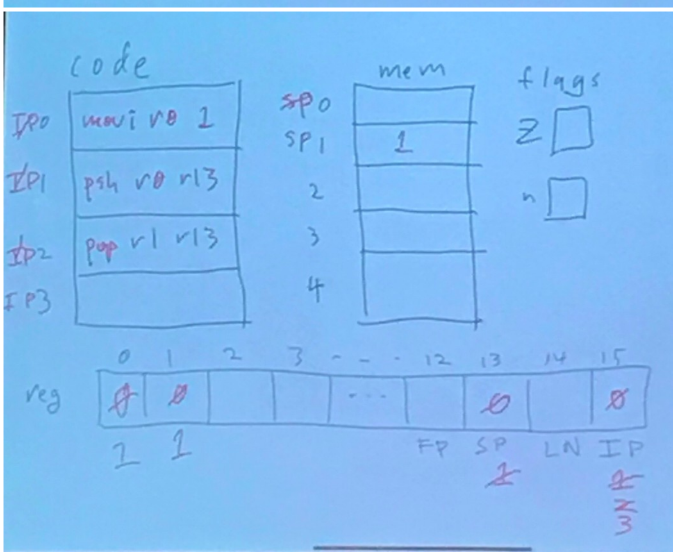


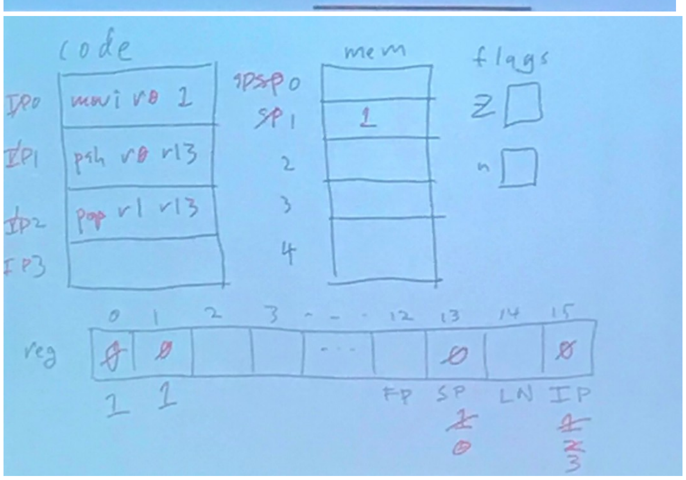
So ip final value status will be 14.



3. Memory instruction







Let’s go through the code.

Read and write:

```
reg[instruction.arg1] = num
fprintf(vmout,”%d\n”, reg[instruction.arg1])

case OP_LD:
        reg[instruction.arg1] = mem[reg[instruction.arg2 + instruction.arg3]]

case OP_BL: (branch and link)
        LN = IP + i;
        next = IP + instruction.arg1;

case OP_RET; (return) return to whatever address in register
        next = reg[instruction.arg1];
```

Bunch of code for VM is given to you
With it, we will read the pcode and process it (run it)
Check the global variables inside the C file
There is a big, infinite loop, which runs until it sees a halt instruction
Check vm_types.h to see the structures and their content
Don’t worry about the IO: implementation of read and write instructions are given
You are just required to translate the pseudo-code for vm to C code to complete implementation of VM

If you run the given pcodes given to you, you will all the execution history in the output: the state of the machine before and after you run each instruction