# COMP 442 / 6421 Compiler Design

## Tutorial 6
## Code Generation

| | | |
|---|---|---|
| Instructor: | Dr. Joey Paquet | paquet@cse.concordia.ca |
| TAs: | Haotao Lai | h_lai@encs.concordia.ca |

# Lab Instructor

Section: lab hours NNK M------ 20:30-22:20 H819

Name: Haotao Lai (Eric)

Office: EV 8.241

Email: h_lai@encs.concordia

Website: http://laihaotao.me/ta

# Content

- Virtual Machine
- Moon Machine
- Assignment Guide

# Attention

There are two approaches to do the code generation:

- tag-based approach: cannot achieve all required functionalities, but it is simple
- stack-based approach: can achieve all requirement, but complicated

If you decide to achieve most of the functionalities you need to choose stack-based approach but it will require a lot of work.

# Virtual Machine

# Virtual Machine

In computing, a virtual machine (VM) is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer.

In computing, an **emulator** is hardware or software that enables one computer system (called the host) to behave like another computer system (called the guest).

Two kinds of VM:

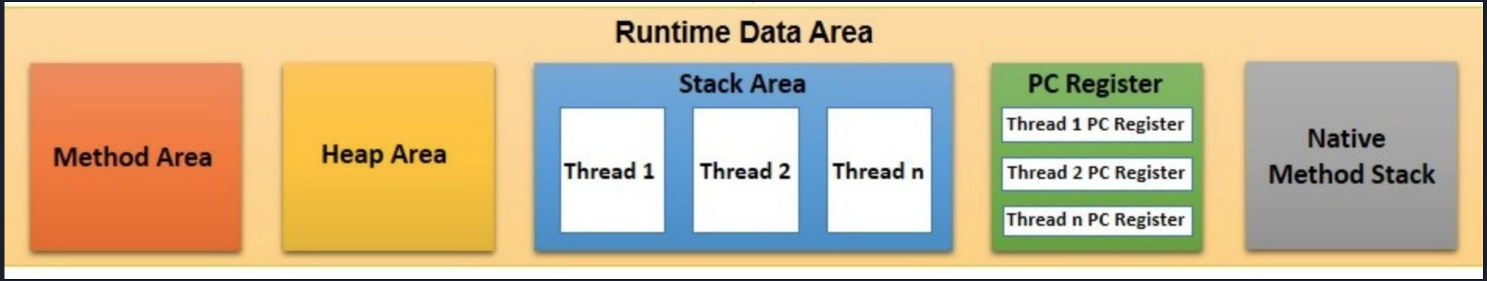
- System virtual machines
- **Process virtual machines**

# Java Virtual Machine

The Java virtual machine is an abstract (virtual) computer defined by a specification. It was designed to provide a platform-independent environment for the Java bytecode execution. By this definition, it is a process virtual machine.

We will take JVM as an example to see how it work and try to get some institutions for our assignment 4.

# Java Runtime Area

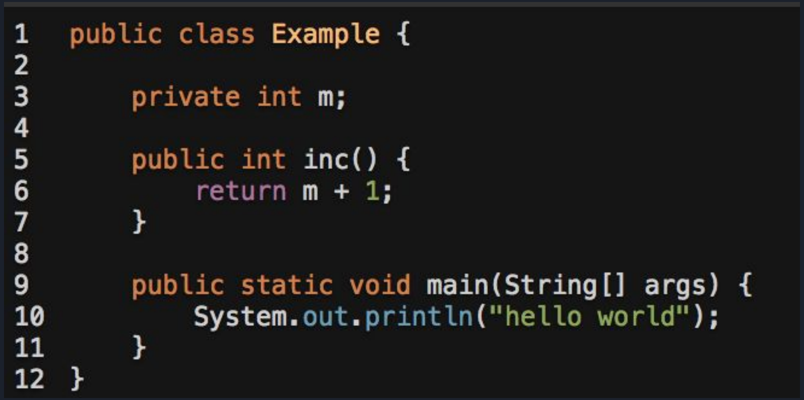


Method Area: store the loaded class objects
Heap Area: store the instance of the class objects
Native Method Stack: method execution stack provided by the OS

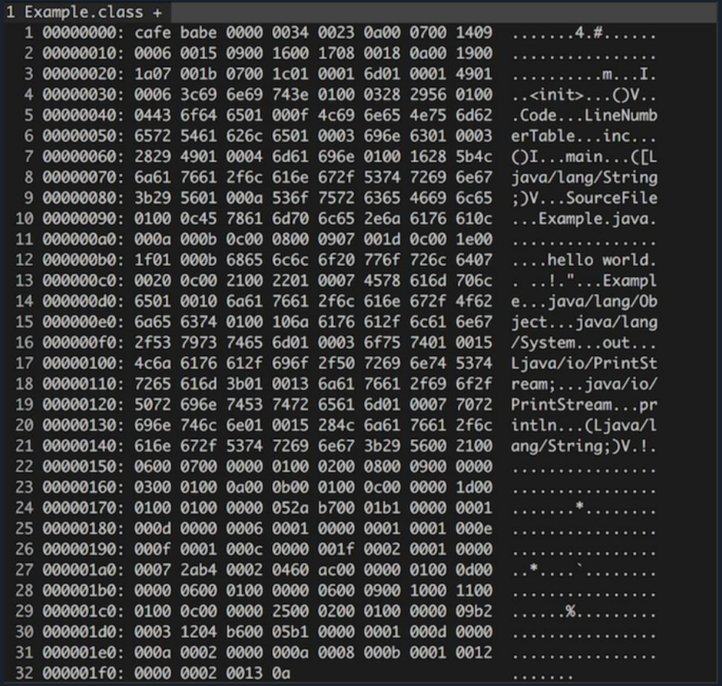
Stack Area: Java defined method execution stack
Program Counter: point to the address of next instruction

In our assignment, we don’t have pointer and we don’t worry about platform independent so we don’t need method area, heap and native method stack.

# Javac



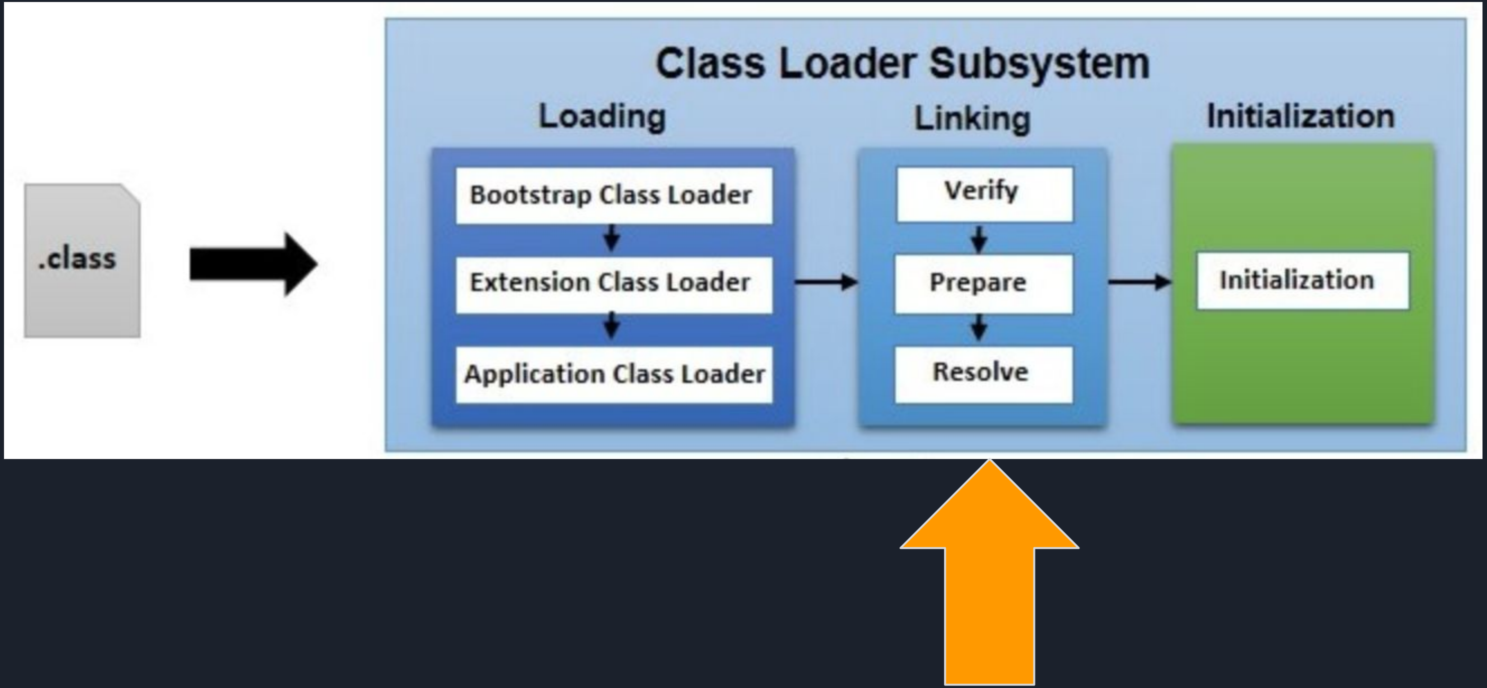
```java
public class Example {

    private int m;

    public int inc() {
        return m + 1;
    }

    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```



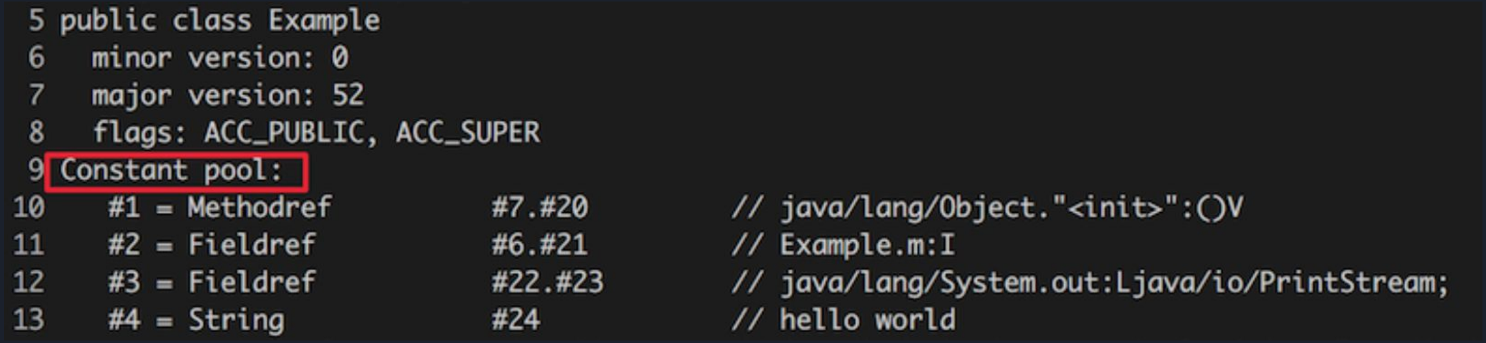
```
1 Example.class +
00000000: cafe babe 0000 0034 0023 0a00 0700 1409  .......4.#......
00000010: 0006 0015 0900 1600 1708 0018 0a00 1900  ................
00000020: 1a07 001b 0700 1c01 0001 6d01 0001 4901  ..........m...I.
00000030: 0006 3c69 6e69 743e 0100 0328 2956 0100  ..<init>...()V..
00000040: 0443 6f64 6501 000f 4c69 6e65 4e75 6d62  .Code...LineNumb
00000050: 6572 5461 626c 6501 0003 696e 6301 0003  erTable...inc...
00000060: 2829 4901 0004 6d61 696e 0100 1628 5b4c  ()I...main...([L
00000070: 6a61 7661 2f6c 616e 672f 5374 7269 6e67  java/lang/String
00000080: 3b29 5601 000a 536f 7572 6365 4669 6c65  ;)V...SourceFile
00000090: 0100 0c45 7861 6d70 6c65 2e6a 6176 610c  ...Example.java.
000000a0: 000a 000b 0c00 0800 0907 001d 0c00 1e00  ................
000000b0: 1f01 000b 6865 6c6c 6f20 776f 726c 6407  ....hello world.
000000c0: 0020 0c00 2100 2201 0007 4578 616d 706c  . ..!."...Exampl
000000d0: 6501 0010 6a61 7661 2f6c 616e 672f 4f62  e...java/lang/Ob
000000e0: 6a65 6374 0100 106a 6176 612f 6c61 6e67  ject...java/lang
000000f0: 2f53 7973 7465 6d01 0003 6f75 7401 0015  /System...out...
00000100: 4c6a 6176 612f 696f 2f50 7269 6e74 5374  Ljava/io/PrintSt
00000110: 7265 616d 3b01 0013 6a61 7661 2f69 6f2f  ream;...java/io/
00000120: 5072 696e 7453 7472 6561 6d01 0007 7072  PrintStream...pr
00000130: 696e 746c 6e01 0015 284c 6a61 7661 2f6c  intln...(Ljava/l
00000140: 616e 672f 5374 7269 6e67 3b29 5600 2100  ang/String;)V.!.
00000150: 0600 0700 0000 0100 0200 0800 0900 0000  ................
00000160: 0300 0100 0a00 0b00 0100 0c00 0000 1d00  ................
00000170: 0100 0100 0000 052a b700 01b1 0000 0001  .......*........
00000180: 000d 0000 0006 0001 0000 0001 0001 000e  ................
00000190: 000f 0001 000c 0000 001f 0002 0001 0000  ................
000001a0: 0007 2ab4 0002 0460 ac00 0000 0100 0d00  ..*....`........
000001b0: 0000 0600 0100 0000 0600 0900 1000 1100  ................
000001c0: 0100 0c00 0000 2500 0200 0100 0000 09b2  ......%.........
000001d0: 0003 1204 b600 05b1 0000 0001 000d 0000  ................
000001e0: 000a 0002 0000 000a 0008 000b 0001 0012  ................
000001f0: 0000 0002 0013 0a                        .......
```

Java code (left) and it corresponding bytecode (right)

# Class Loading Procedure



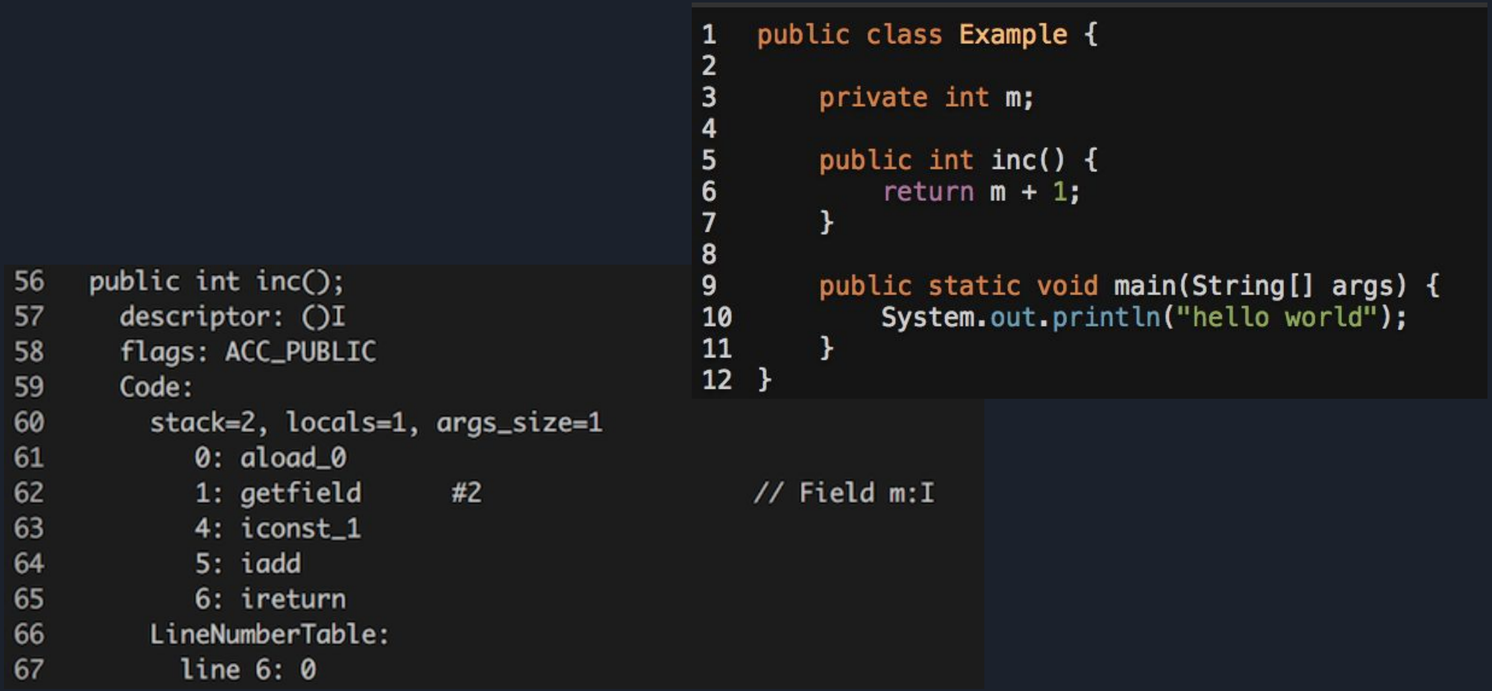
# Javap



```
public class Example
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #7.#20         // java/lang/Object."<init>":()V
   #2 = Fieldref           #6.#21         // Example.m:I
   #3 = Fieldref           #22.#23        // java/lang/System.out:Ljava/io/PrintStream;
   #4 = String             #24            // hello world
```

Constant pool is like symbol table in our case, but it is more complicated than ours.

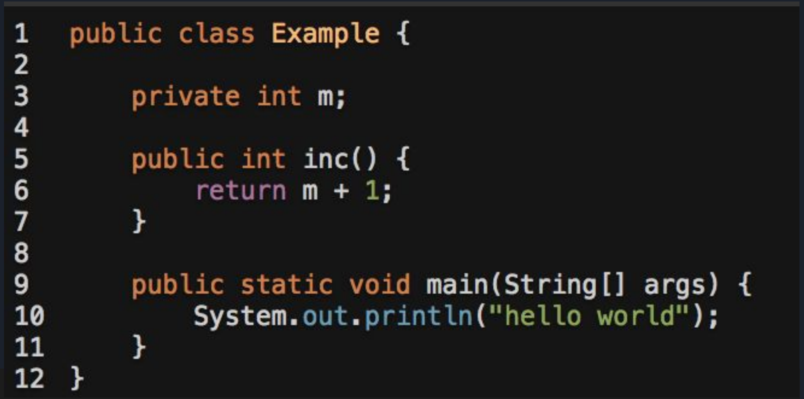
# Javap



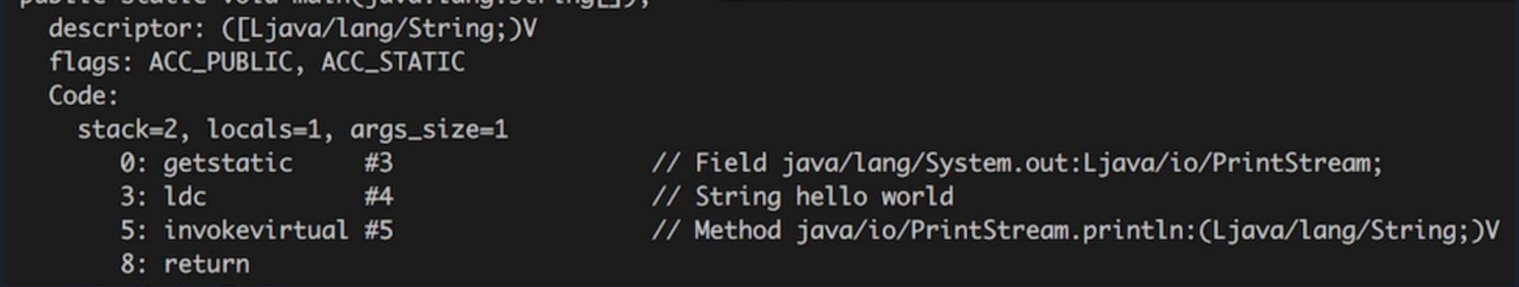
```java
public class Example {

    private int m;

    public int inc() {
        return m + 1;
    }

    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

```
public int inc();
  descriptor: ()I
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=1, args_size=1
       0: aload_0
       1: getfield      #2                  // Field m:I
       4: iconst_1
       5: iadd
       6: ireturn
    LineNumberTable:
      line 6: 0
```

# Javap



```java
public class Example {

    private int m;

    public int inc() {
        return m + 1;
    }

    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

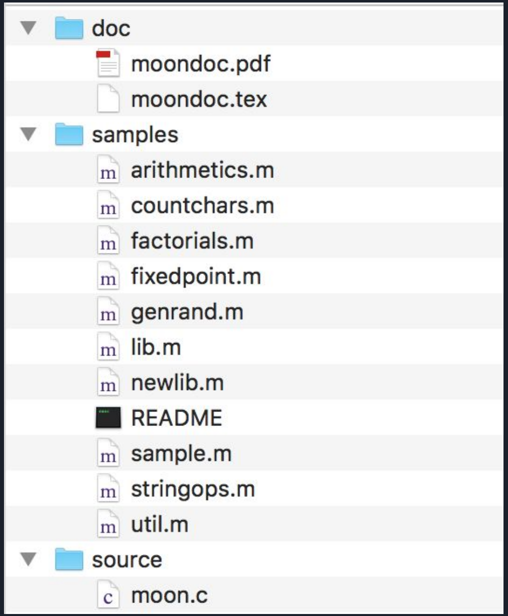


```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #4                  // String hello world
       5: invokevirtual #5                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
```

# Moon Machine

# Background

- The MOON processor is wrote by Dr. Peter Grogono, the last modification is on 30 January 1995;
- It is a kind of “virtual machine” we used to run our generated code (assembly language)
- You can get the source code of Moon in the bottom of the course website
- You need to have the very basic idea of assembly language



```
▼ doc
    moondoc.pdf
    moondoc.tex
▼ samples
    arithmetics.m
    countchars.m
    factorials.m
    fixedpoint.m
    genrand.m
    lib.m
    newlib.m
    README
    sample.m
    stringops.m
    util.m
▼ source
    moon.c
```

# How to compile MOON?

1. You need to have a C compiler (eg. gcc)
2. Download the source code and unzip it
3. Open Terminal, change your working directory to where you put the source code
4. Compile it using the very basic compile command

For example, if you are using **gcc**, just type the following command in the terminal:

```
gcc [-o executable_file_name] moon.c
```

If you don’t specify the name, the executable will be named “a” in Unix, Linux or macOS .

Note: there is a PDF file accompanying with the source code, you are strongly suggested to read that file before you ask any question.

# What Moon provides you?

- A set of instructions
- A whole blank addressable virtual (simulated) memory
- Total 16 simulated registers
- A program counter

# How to use MOON?

There are 4 types of instruction:

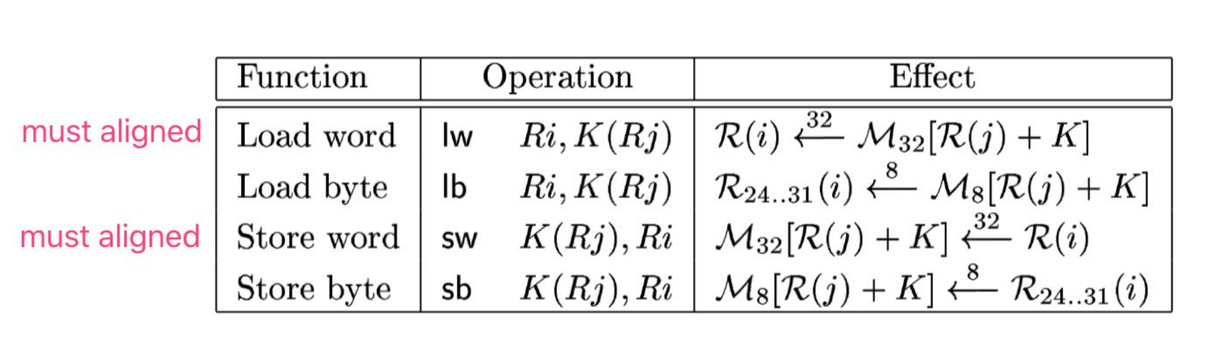
1. Data access instructions
2. Arithmetic instructions
3. Input and output instructions
4. Control instructions

Terminology

- $M_8$[K]: it denotes the **byte** stored at address K;
- $M_{32}$[K]: it denotes the **word** stored at address K, K + 1, K + 2 and K + 3;
- An address is **aligned** if it is a multiple of 4;
- An address is **legal** if the address byte exists;
- The name PC denotes the **program counter**;
- The name R0, R1, … denotes the **registers**;
- The symbol ← denotes data transfer;

Note: the slide cannot show all instructions provided by MOON, please consult the documentation for more detailed !

# Data Access Instructions



| | Function | Operation | | Effect |
|---|---|---|---|---|
| must aligned | Load word | lw | $Ri, K(Rj)$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{M}_{32}[\mathcal{R}(j) + K]$ |
| | Load byte | lb | $Ri, K(Rj)$ | $\mathcal{R}_{24..31}(i) \xleftarrow{8} \mathcal{M}_{8}[\mathcal{R}(j) + K]$ |
| must aligned | Store word | sw | $K(Rj), Ri$ | $\mathcal{M}_{32}[\mathcal{R}(j) + K] \xleftarrow{32} \mathcal{R}(i)$ |
| | Store byte | sb | $K(Rj), Ri$ | $\mathcal{M}_{8}[\mathcal{R}(j) + K] \xleftarrow{8} \mathcal{R}_{24..31}(i)$ |

Take load word as an example:

$R(i) \xleftarrow{32} M_{32}[R(j) + K]$ means take one word data stored in the address ( R(j) + K ) and put it into register R(i)

where K in the range of [-16384, 16384)

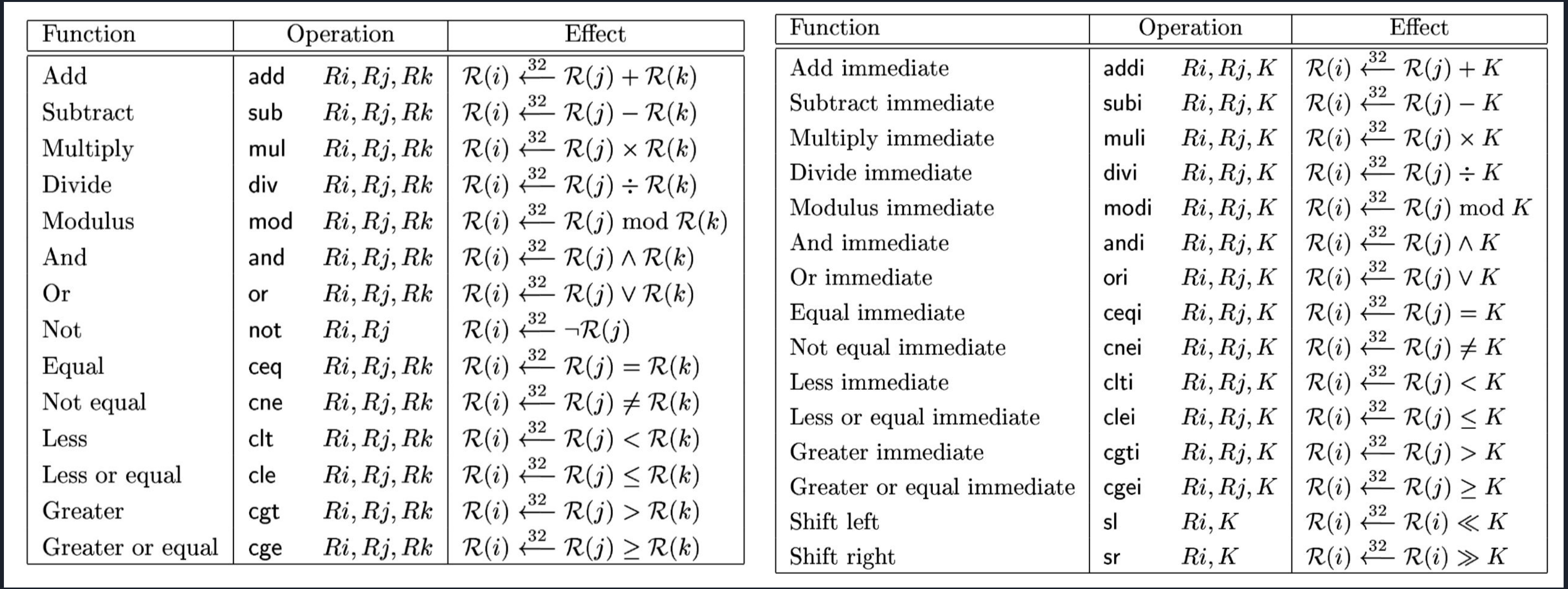
# Arithmetic Instructions

There are two types of arithmetic instructions:

1. $R(i) \leftarrow R(j) + R(k)$, sum up the second and third register’s value and put the result into the first register;
2. $R(i) \leftarrow R(j) + k$, sum up the second register’s value and the third value then put the result into the first register;

We call all productions like the second one shown above “instruction with immediate operand”.

# Arithmetic Instructions

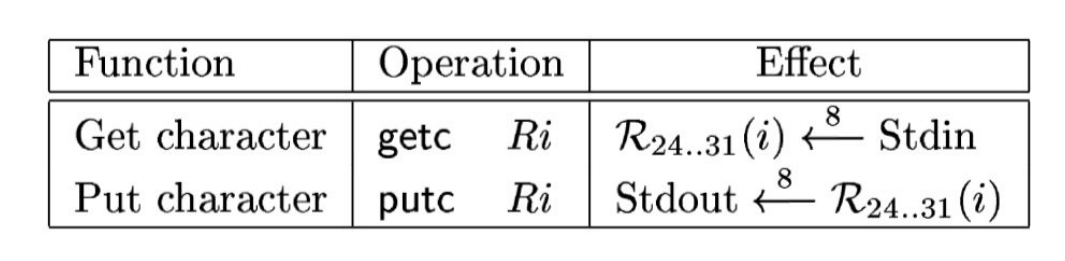


| Function | Operation | Effect |
|---|---|---|
| Add | add $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) + \mathcal{R}(k)$ |
| Subtract | sub $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) - \mathcal{R}(k)$ |
| Multiply | mul $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \times \mathcal{R}(k)$ |
| Divide | div $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \div \mathcal{R}(k)$ |
| Modulus | mod $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \bmod \mathcal{R}(k)$ |
| And | and $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \wedge \mathcal{R}(k)$ |
| Or | or $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \vee \mathcal{R}(k)$ |
| Not | not $Ri, Rj$ | $\mathcal{R}(i) \xleftarrow{32} \neg\mathcal{R}(j)$ |
| Equal | ceq $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) = \mathcal{R}(k)$ |
| Not equal | cne $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \neq \mathcal{R}(k)$ |
| Less | clt $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) < \mathcal{R}(k)$ |
| Less or equal | cle $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \leq \mathcal{R}(k)$ |
| Greater | cgt $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) > \mathcal{R}(k)$ |
| Greater or equal | cge $Ri, Rj, Rk$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \geq \mathcal{R}(k)$ |

| Function | Operation | Effect |
|---|---|---|
| Add immediate | addi $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) + K$ |
| Subtract immediate | subi $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) - K$ |
| Multiply immediate | muli $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \times K$ |
| Divide immediate | divi $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \div K$ |
| Modulus immediate | modi $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \bmod K$ |
| And immediate | andi $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \wedge K$ |
| Or immediate | ori $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \vee K$ |
| Equal immediate | ceqi $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) = K$ |
| Not equal immediate | cnei $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \neq K$ |
| Less immediate | clti $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) < K$ |
| Less or equal immediate | clei $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \leq K$ |
| Greater immediate | cgti $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) > K$ |
| Greater or equal immediate | cgei $Ri, Rj, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \geq K$ |
| Shift left | sl $Ri, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(i) \ll K$ |
| Shift right | sr $Ri, K$ | $\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(i) \gg K$ |

- the logical operation operate on each bit of the word
- the comparison operator store result either “1” (true) or “0” (false)
- in the right side table, the operand K is a signed 16-bit quantity, negative numbers like -1 is interpreted as -1 not 65535

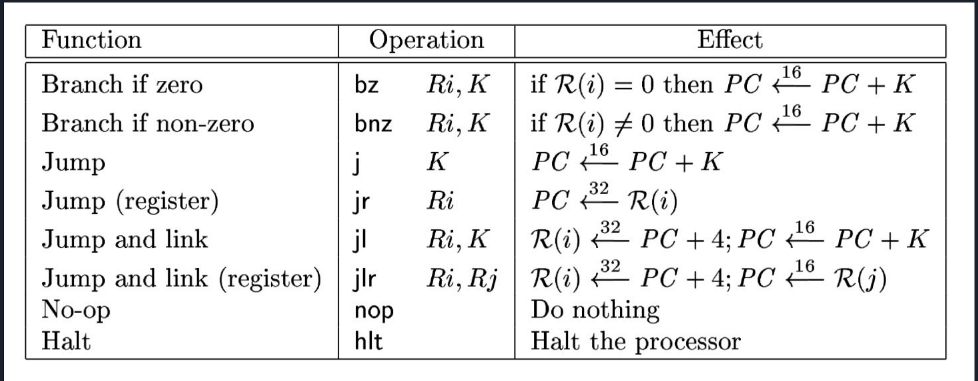
# Input and Output Instructions



| Function | Operation | Effect |
|---|---|---|
| Get character | getc $Ri$ | $\mathcal{R}_{24..31}(i) \xleftarrow{8} \text{Stdin}$ |
| Put character | putc $Ri$ | $\text{Stdout} \xleftarrow{8} \mathcal{R}_{24..31}(i)$ |

This two instructions are useful when you try to out the result of your program to show it really worked during the final demo.

# Control Instructions



| Function | Operation | | Effect |
|---|---|---|---|
| Branch if zero | bz | $Ri, K$ | if $\mathcal{R}(i) = 0$ then $PC \xleftarrow{16} PC + K$ |
| Branch if non-zero | bnz | $Ri, K$ | if $\mathcal{R}(i) \neq 0$ then $PC \xleftarrow{16} PC + K$ |
| Jump | j | $K$ | $PC \xleftarrow{16} PC + K$ |
| Jump (register) | jr | $Ri$ | $PC \xleftarrow{32} \mathcal{R}(i)$ |
| Jump and link | jl | $Ri, K$ | $\mathcal{R}(i) \xleftarrow{32} PC + 4; PC \xleftarrow{16} PC + K$ |
| Jump and link (register) | jlr | $Ri, Rj$ | $\mathcal{R}(i) \xleftarrow{32} PC + 4; PC \xleftarrow{16} \mathcal{R}(j)$ |
| No-op | nop | | Do nothing |
| Halt | hlt | | Halt the processor |

- when you use branch, remember to set the PC (program counter) correctly
- jump instruction will be useful when you generate function code, you need to store the return address properly

# Code Generation

Tag-based

# Tag-based Approach

The way to do it is straightforward and simple, for each variable you allocate a memory for it and associate it with a unique tag which is stored in the symbol table.

Next time, when you want to access  this variable (in-memory location) you can just get its address by using that predefined tag in your table.

# Code Generation

Stack-based

# The key of code generation → offset

Recall

- How can you know whether a variable has been declared or not when you try to use it ?
- How many column you have in your symbol table ? What do they use for ?

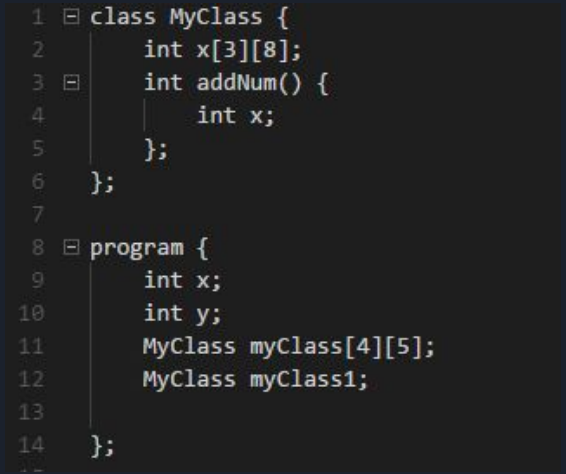
Offset

- It represent how far a variable away from a base address;
- For example, a member variable of a class, offset of the variable means how far this variable’s first address away from the first address of the class;

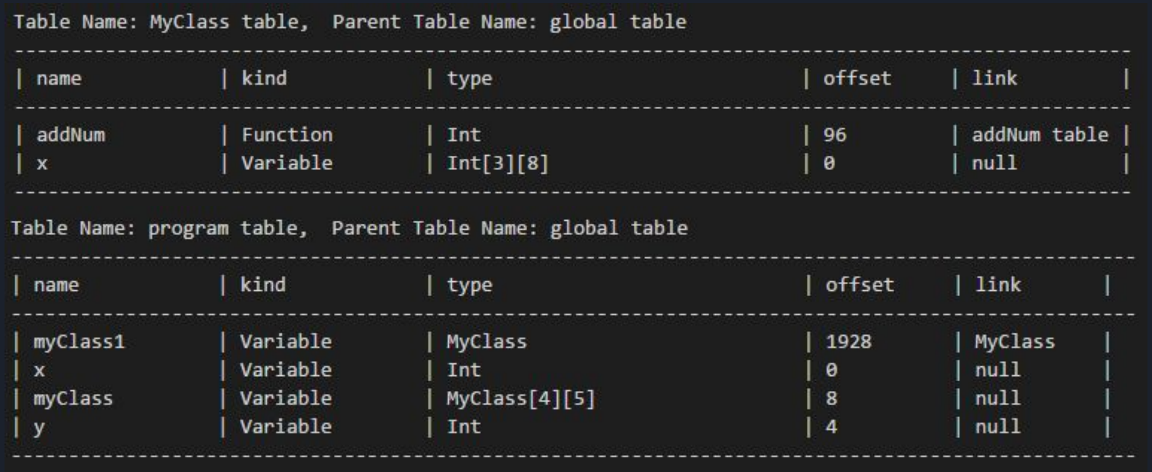
In order to achieve code generation:

- Add a new column to your symbol table → offset
- Calculate the offset of each data type when you add that entry into your table

# Offset Example → the fourth column



```
class MyClass {
    int x[3][8];
    int addNum() {
        int x;
    };
};

program {
    int x;
    int y;
    MyClass myClass[4][5];
    MyClass myClass1;

};
```



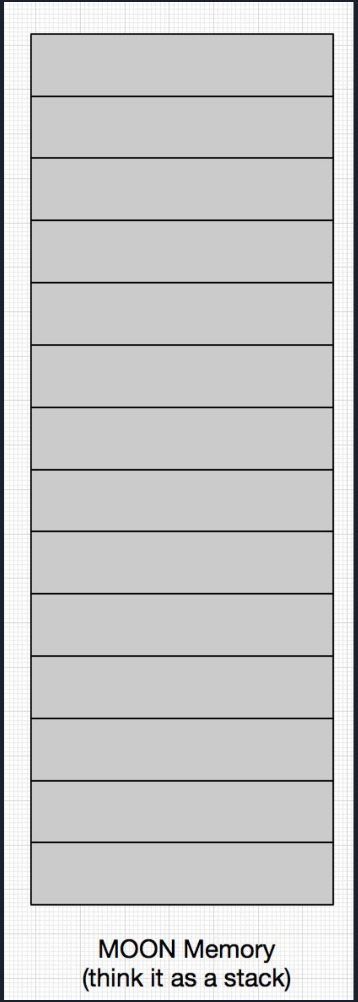
```
Table Name: MyClass table,  Parent Table Name: global table
--------------------------------------------------------------------------------------------
| name            | kind            | type                          | offset     | link         |
--------------------------------------------------------------------------------------------
| addNum          | Function        | Int                           | 96         | addNum table |
| x               | Variable        | Int[3][8]                     | 0          | null         |
--------------------------------------------------------------------------------------------
Table Name: program table,  Parent Table Name: global table
--------------------------------------------------------------------------------------------
| name            | kind            | type                          | offset     | link       |
--------------------------------------------------------------------------------------------
| myClass1        | Variable        | MyClass                       | 1928       | MyClass    |
| x               | Variable        | Int                           | 0          | null       |
| myClass         | Variable        | MyClass[4][5]                 | 8          | null       |
| y               | Variable        | Int                           | 4          | null       |
--------------------------------------------------------------------------------------------
```

# Stack Mechanism

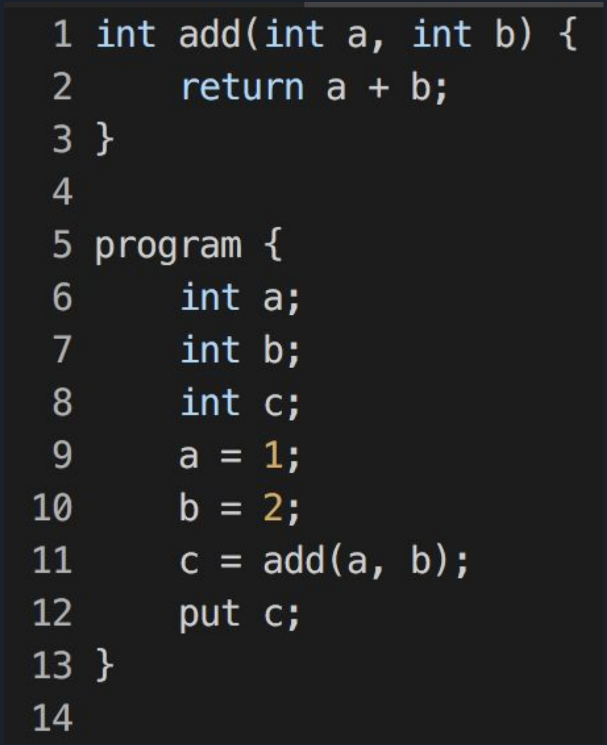
# What is the stack?

The stack we talk about here is not the real “data structure stack”. It is a function invocation stack. When a function being called, its frame will be pushed into the stack and when the function return the corresponding frame will be popped out.

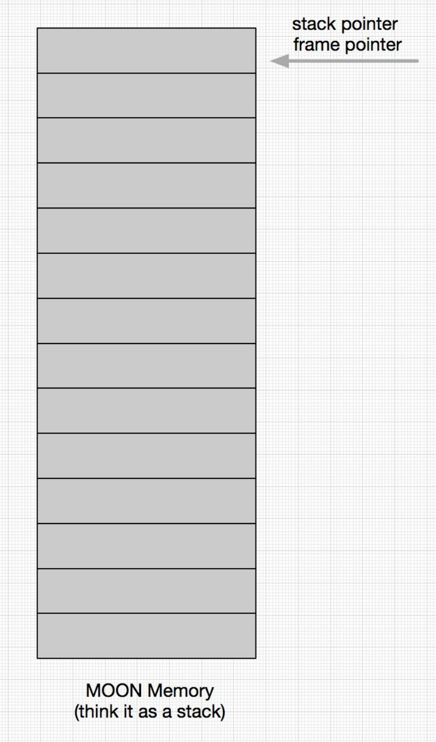
In our case, we treat the MOON’s memory as a stack.



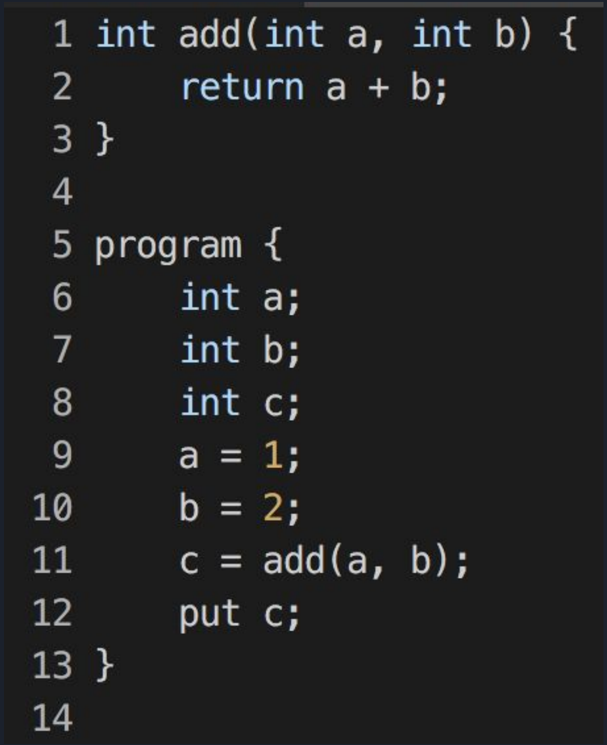
# Stack-based Function Call Mechanism



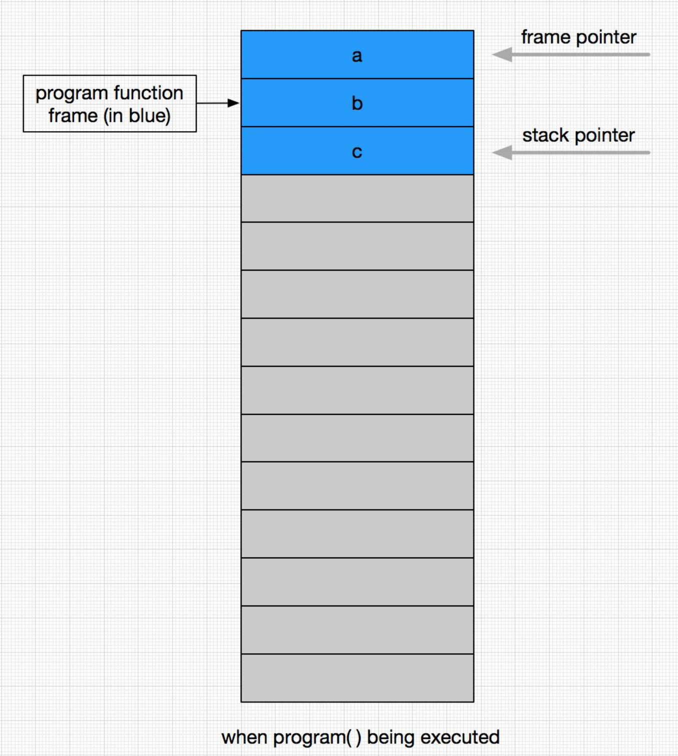
```
int add(int a, int b) {
    return a + b;
}

program {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
    c = add(a, b);
    put c;
}

```



# Stack-based Function Call Mechanism

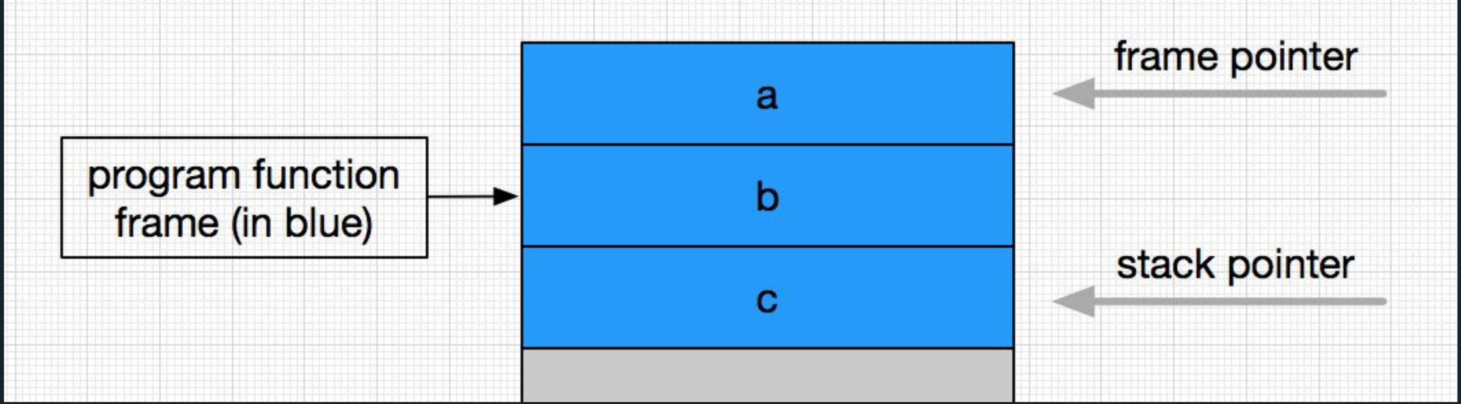


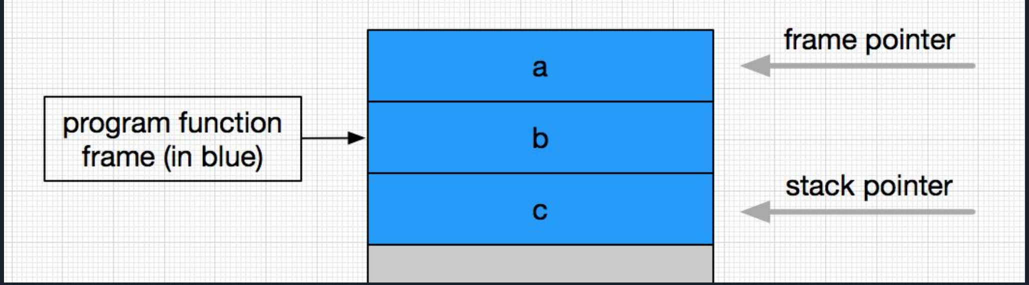
```
int add(int a, int b) {
    return a + b;
}

program {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
    c = add(a, b);
    put c;
}

```



when program( ) being executed

# How you can know where you should put a, b, c and how to locate them?

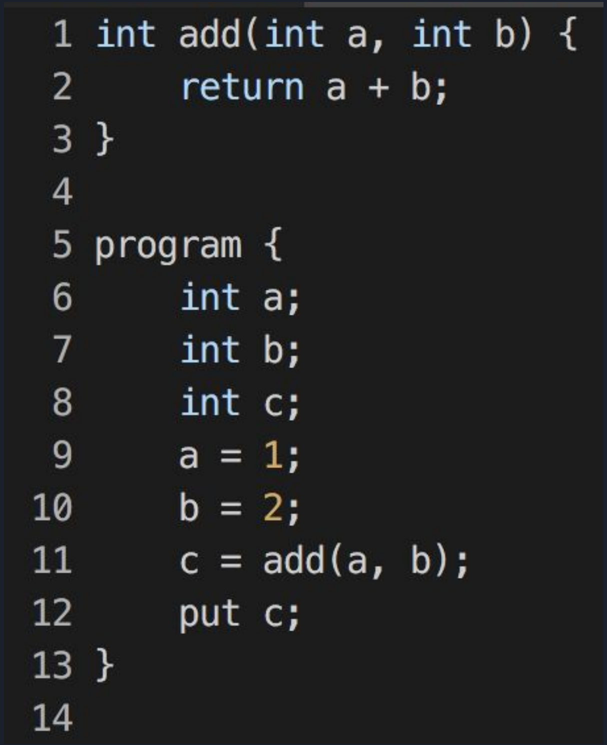




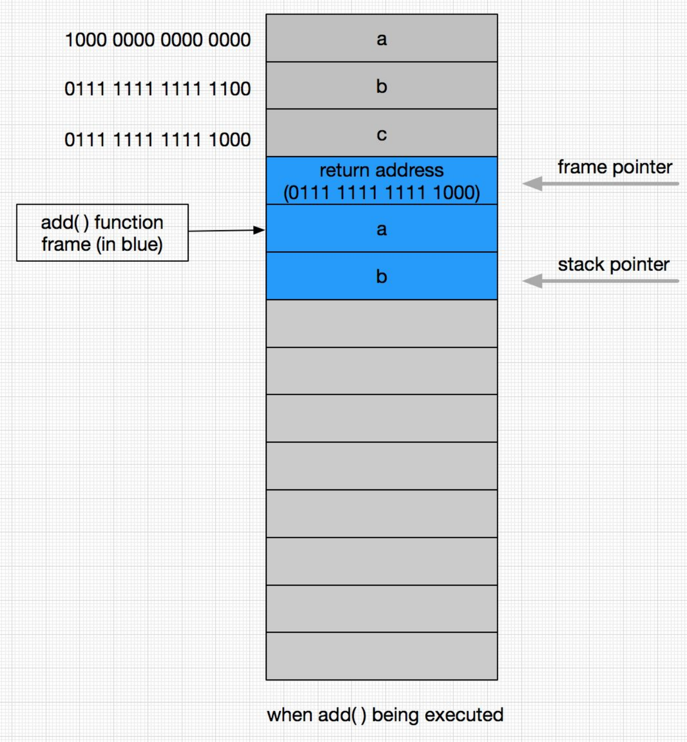
| name | offset |
|---|---|
| a | 0 |
| b | 4 |
| c | 8 |

Remember we have offset!
offset → the distance from the variable cell to the frame pointer (current function's base address).
stack pointer → where the new function frame should be put.

# Stack-based Function Call Mechanism



```
int add(int a, int b) {
    return a + b;
}

program {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
    c = add(a, b);
    put c;
}

```



1000 0000 0000 0000 a

0111 1111 1111 1100 b

0111 1111 1111 1000 c

return address (0111 1111 1111 1000)

frame pointer

add( ) function frame (in blue)

a

b

stack pointer

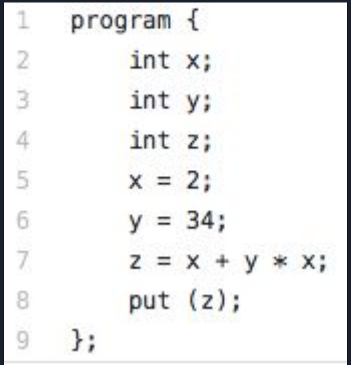
when add( ) being executed

# Function Call Mechanism further considerations

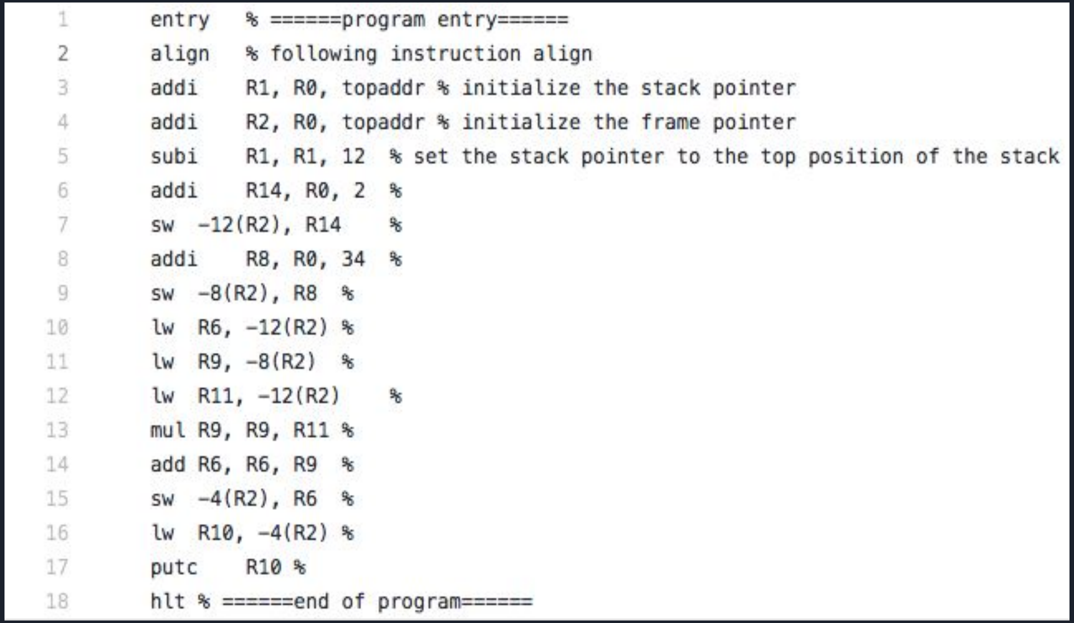
- how to pass the parameter into the new function?
- where should frame and stack pointer go when the executing function is done?
- how to refer to the data member inside a member function?
- how to pass the return value back to the caller function?
- etc … …

# Example

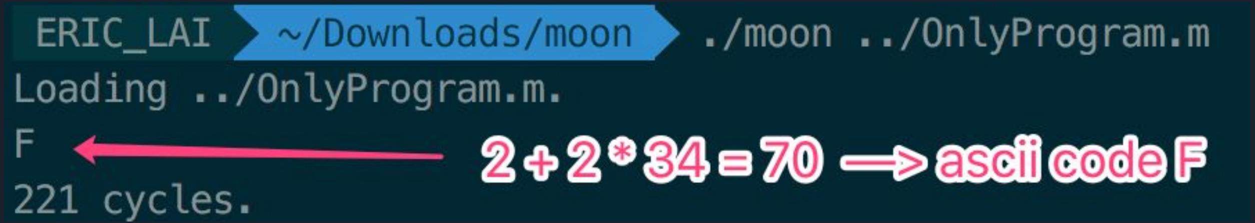
source code → assembly code



```
program {
    int x;
    int y;
    int z;
    x = 2;
    y = 34;
    z = x + y * x;
    put (z);
};
```

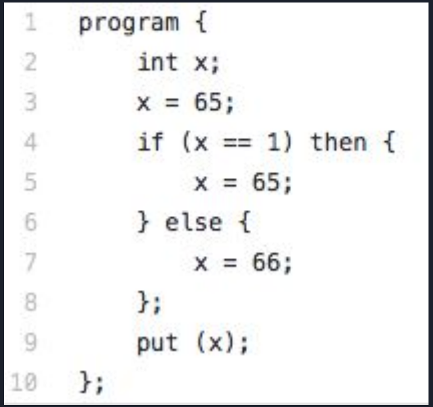


```
        entry   % ======program entry======
        align   % following instruction align
        addi    R1, R0, topaddr % initialize the stack pointer
        addi    R2, R0, topaddr % initialize the frame pointer
        subi    R1, R1, 12  % set the stack pointer to the top position of the stack
        addi    R14, R0, 2  %
        sw  -12(R2), R14    %
        addi    R8, R0, 34  %
        sw  -8(R2), R8  %
        lw  R6, -12(R2) %
        lw  R9, -8(R2)  %
        lw  R11, -12(R2)    %
        mul R9, R9, R11 %
        add R6, R6, R9  %
        sw  -4(R2), R6  %
        lw  R10, -4(R2) %
        putc    R10 %
        hlt % ======end of program======
```

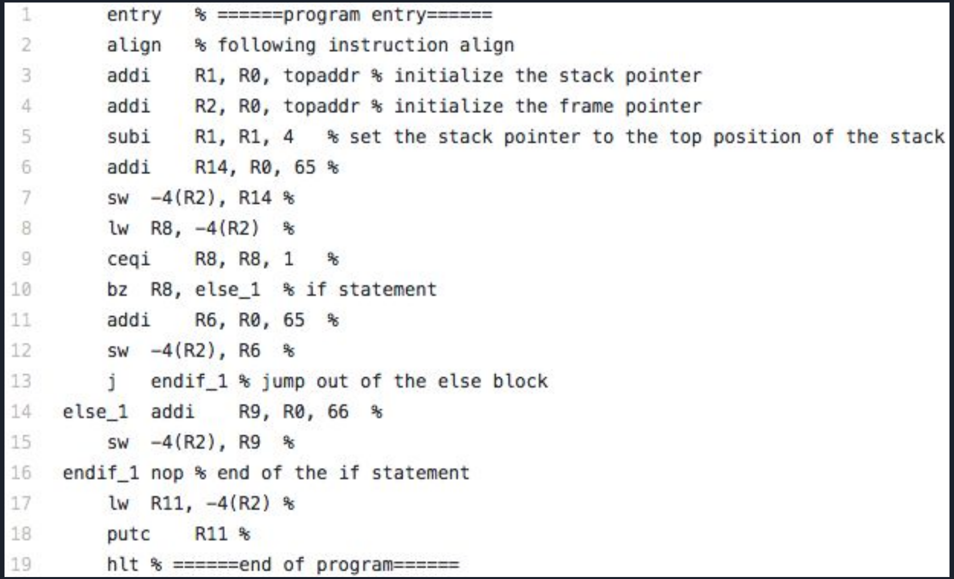


```
ERIC_LAI  ~/Downloads/moon  ./moon ../OnlyProgram.m
Loading ../OnlyProgram.m.
F
221 cycles.
```

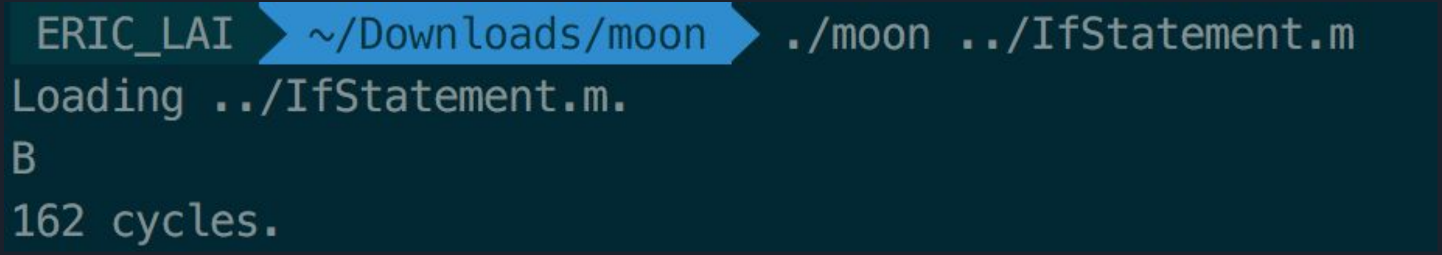
2+2*34=70 —> ascii code F



```
program {
    int x;
    x = 65;
    if (x == 1) then {
        x = 65;
    } else {
        x = 66;
    };
    put (x);
};
```



```
    entry   % ======program entry======
    align   % following instruction align
    addi    R1, R0, topaddr % initialize the stack pointer
    addi    R2, R0, topaddr % initialize the frame pointer
    subi    R1, R1, 4   % set the stack pointer to the top position of the stack
    addi    R14, R0, 65 %
    sw  -4(R2), R14 %
    lw  R8, -4(R2)  %
    ceqi    R8, R8, 1   %
    bz  R8, else_1  % if statement
    addi    R6, R0, 65  %
    sw  -4(R2), R6  %
    j   endif_1 % jump out of the else block
else_1  addi    R9, R0, 66  %
    sw  -4(R2), R9  %
endif_1 nop % end of the if statement
    lw  R11, -4(R2) %
    putc    R11 %
    hlt % ======end of program======
```



```
 ERIC_LAI  ~/Downloads/moon  ./moon ../IfStatement.m
Loading ../IfStatement.m.
B
162 cycles.
```

# Reference

- https://en.wikipedia.org/wiki/Virtual_machine
- https://en.wikipedia.org/wiki/Emulator
- https://blogitwithsatyam.com/2018/06/19/jvm-architecture-in-depth/
- http://laihaotao.me/2018/04/28/jvm-classfile.html