

Code Generation: Introduction

Example: gcc

test.c

```
#include <stdio.h>
int main() {
    int i = 0;
    int j = 0;
    while (i < 10) {
        printf("%d\n", j);
        i = i + 1;
        j = j + 2*i+1;
    }
}
```

gcc test.c -S

test.s

```
        jmp .L2
.L3:    movl -8(%ebp), %eax
        movl %eax, 4(%esp)
        movl $.LC0, (%esp)
        call printf
        addl $1, -12(%ebp)
        movl -12(%ebp), %eax
        addl %eax, %eax
        addl -8(%ebp), %eax
        addl $1, %eax
        movl %eax, -8(%ebp)
.L2:    cmpl $9, -12(%ebp)
        jle .L3
```

What did `(i<10)` compile to?

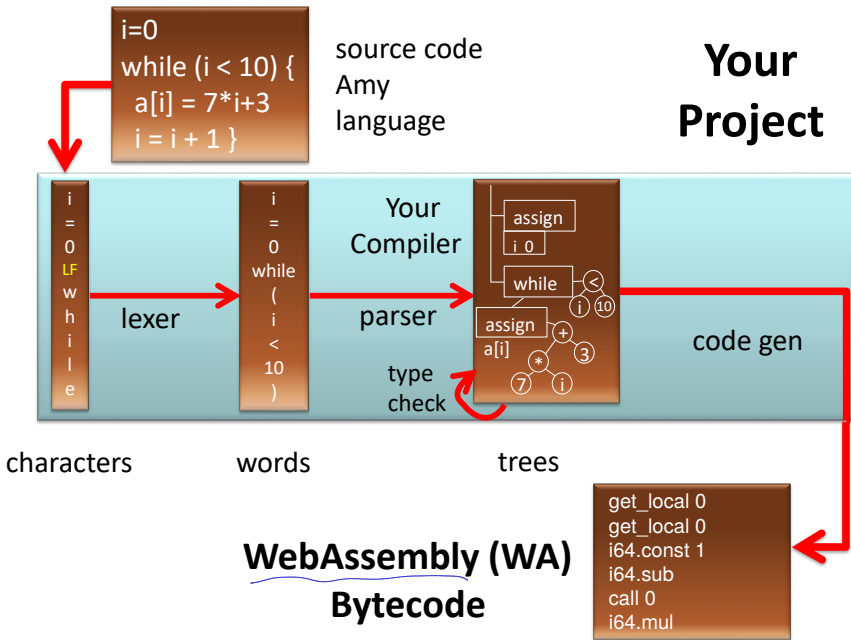
javac example

```
while (i < 10) {  
    System.out.println(j);  
    i = i + 1;  
    j = j + 2*i+1;  
}
```

javac Test.java
javap -c Test

```
4: iload_1  
5: bipush 10  
7: if_icmpge 32  
10: getstatic #2; //System.out  
13: iload_2  
14: invokevirtual #3; //println  
17: iload_1  
18: iconst_1  
19: iadd  
20: istore_1  
21: iload_2 ← j  
22: iconst_2 ← 2  
23: iload_1 ← i  
24: imul      2*i  
25: iadd      j + 2*i  
26: iconst_1  1  
27: iadd      j + 2*i + 1  
28: istore_2  
29: goto 4  
32: return
```

Guess what each JVM instruction for the highlighted expression does.



WebAssembly example

C++

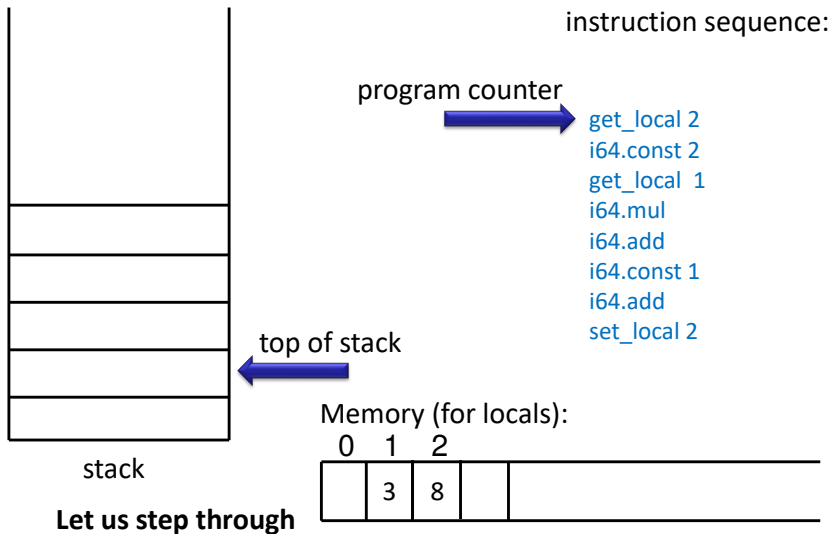
```
int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```

WebAssembly

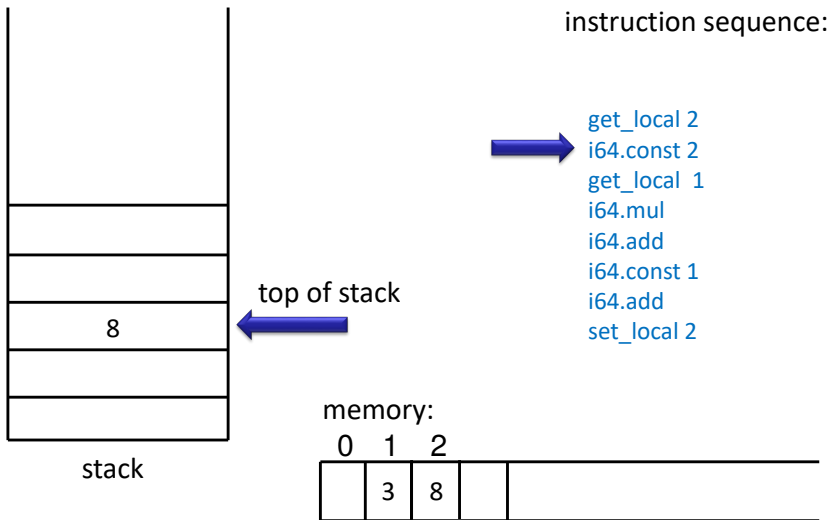
```
get_local 0    // n  
i64.const 0    // 0  
i64.eq         // n==0 ?  
if i64  
    i64.const 1 // 1  
else  
    get_local 0 // n  
    get_local 0 // n  
    i64.const 1 // 1  
    i64.sub     // n-1  
    call 0      // f(n-1)  
    i64.mul     // n*f(n-1)  
end
```

More at: <https://mbebenita.github.io/WasmExplorer/>

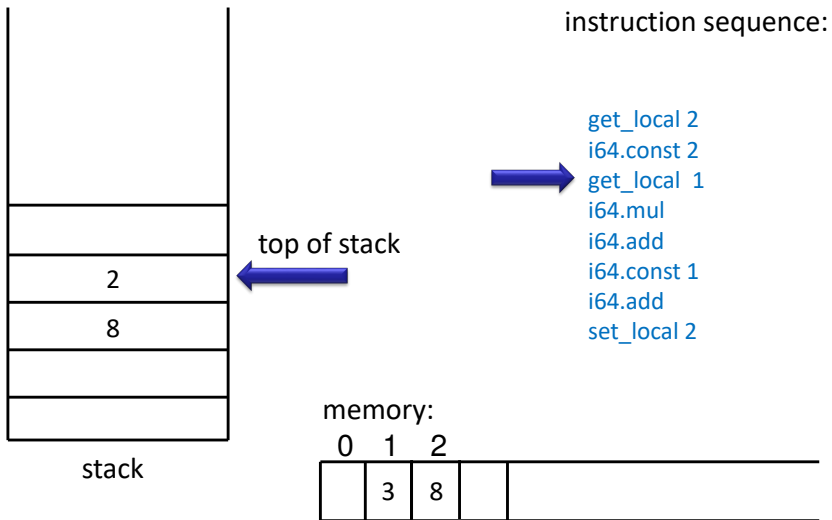
Stack Machine: High-Level Machine Code



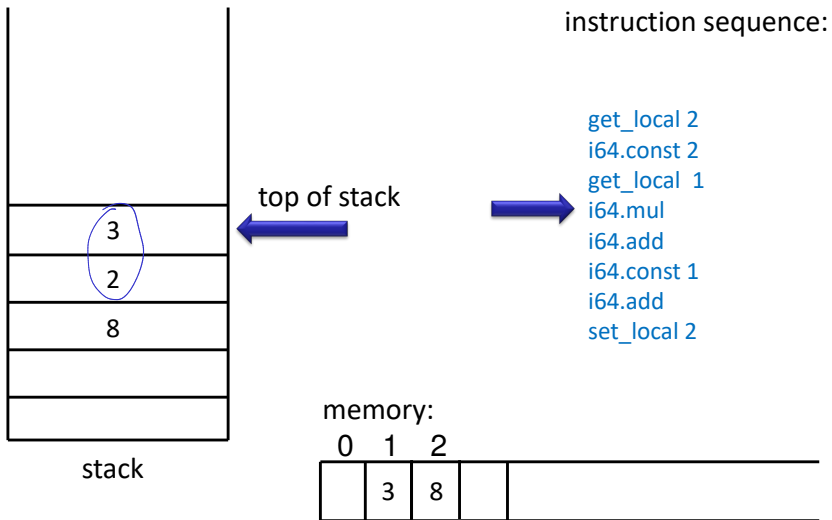
Operands are consumed from stack and put back onto stack



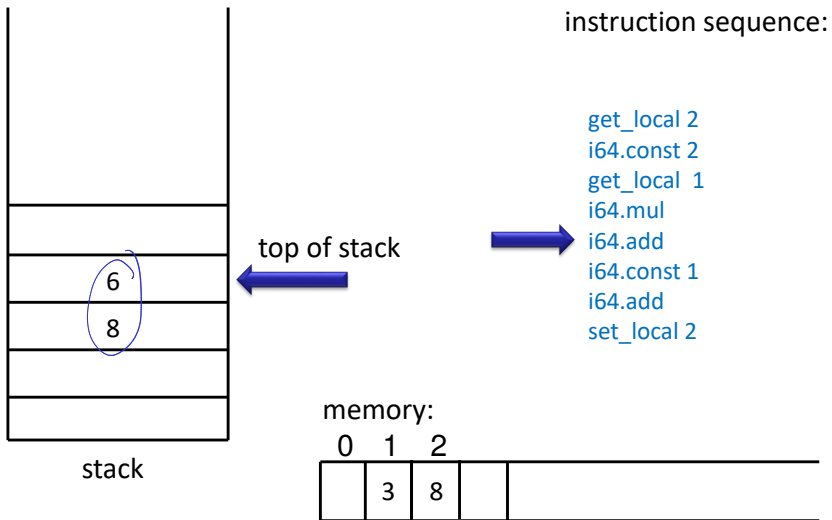
Operands are consumed from stack and put back onto stack



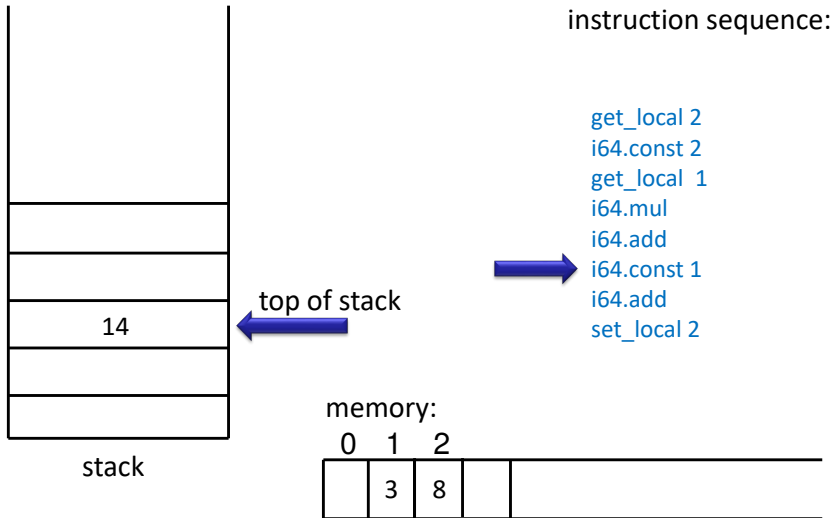
Operands are consumed from stack and put back onto stack



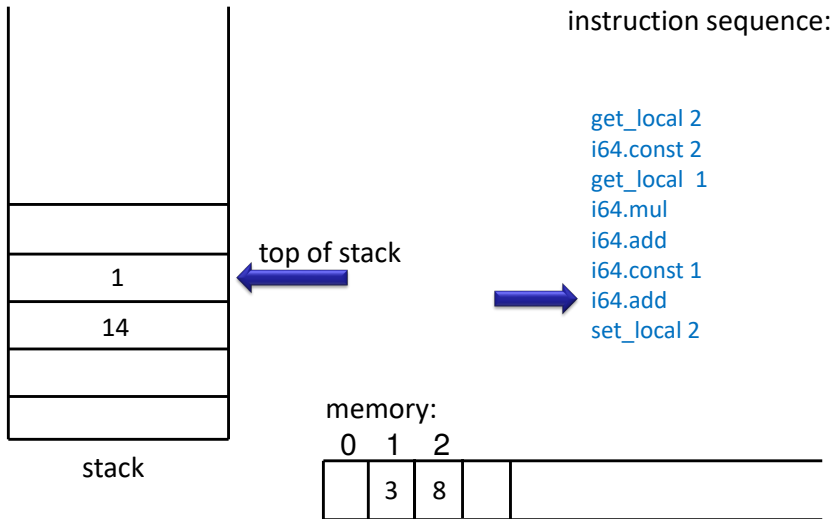
Operands are consumed from stack and put back onto stack



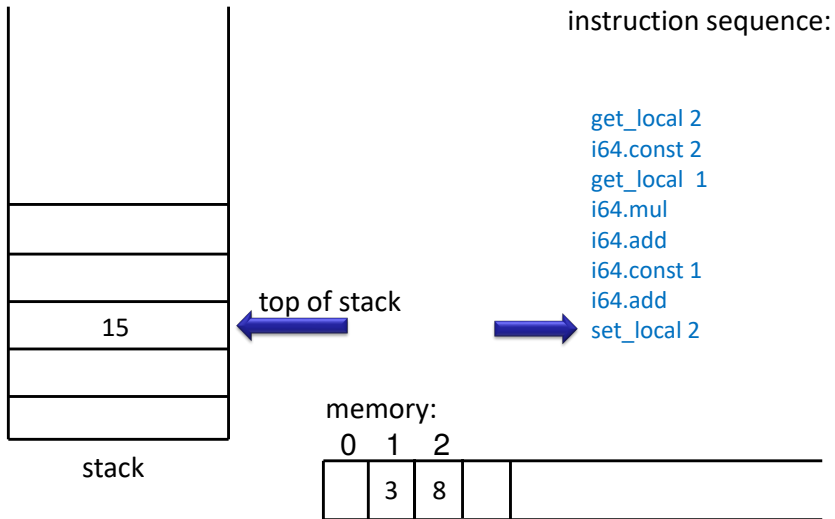
Operands are consumed from stack and put back onto stack



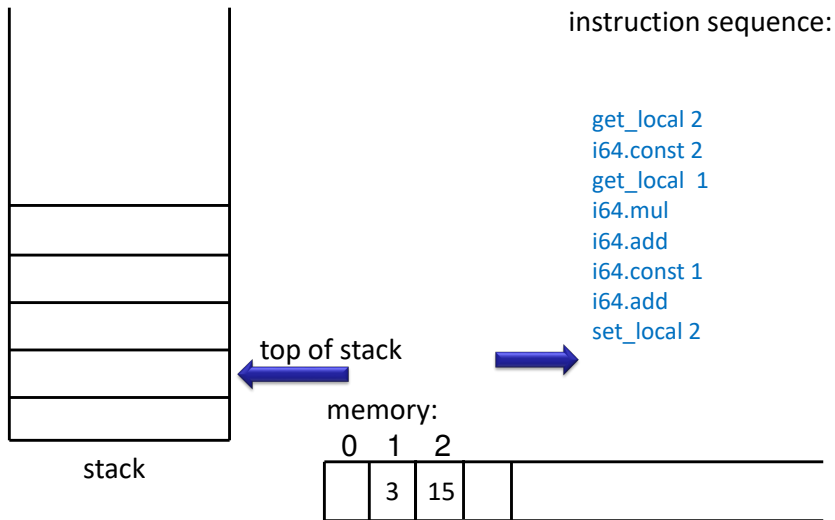
Operands are consumed from stack and put back onto stack



Operands are consumed from stack and put back onto stack



Operands are consumed from stack and put back onto stack

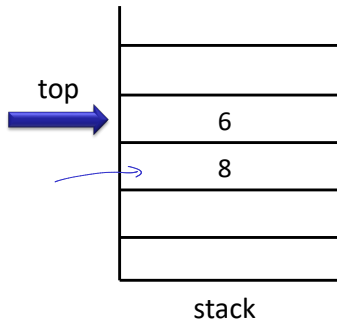


Stack Machine Simulator

```
var code : Array[Instruction]
var pc : Int // program counter
var local : Array[Int] // for local variables
var operand : Array[Int] // operand stack
var top : Int
```

while (true) step

```
def step = code(pc) match {
  case ladd() =>
    operand(top - 1) = operand(top - 1) + operand(top)
    top = top - 1 // two consumed, one produced
  case lmul() =>
    operand(top - 1) = operand(top - 1) * operand(top)
    top = top - 1 // two consumed, one produced
```



Stack Machine Simulator: Moving Data

```
case iconst(c) =>  
  operand(top + 1) = c // put given constant 'c' onto stack  
  top = top + 1  
case lgetlocal(n) =>  
  operand(top + 1) = local(n) // from memory onto stack  
  top = top + 1  
case lsetlocal(n) =>  
  local(n) = operand(top) // from stack into memory  
  top = top - 1 // consumed  
}  
if (notJump(code(n)))  
  pc = pc + 1 // by default go to next instructions
```

WebAssembly reference interpreter in ocaml:

<https://github.com/WebAssembly/spec/tree/master/interpreter>

Selected Instructions

Reading and writing locals (and parameters):

- **get_local**: read the current value of a local variable
- **set_local**: set the current value of a local variable
- **tee_local**: like set_local, but also returns the set value

Arithmetic operations (take args from stack, put result on stack):

i32.add: sign-agnostic addition

i32.sub: sign-agnostic subtraction

i32.mul: sign-agnostic multiplication (lower 32-bits)

i32.div_s: signed division (result is truncated toward zero)

i32.rem_s: signed remainder (result has the sign of the dividend x in x%y)

i32.and: sign-agnostic bitwise and

i32.or: sign-agnostic bitwise inclusive or

i32.xor: sign-agnostic bitwise exclusive or

Comparisons, stack, memory

i32.eq: sign-agnostic compare equal

i32.ne: sign-agnostic compare unequal

i32.lt_s: signed less than

i32.le_s: signed less than or equal

i32.gt_s: signed greater than

i32.ge_s: signed greater than or equal

i32.eqz: compare equal to zero (return 1 if operand is zero, 0 otherwise)

There are also: 64 bit integer operations **i64._** and floating point **f32._**, **f64._**

drop: drop top of the stack

i32.const C: put a given constant **C** on the stack

Access to memory (given as one big array):

i32.load: get memory index from stack, load 4 bytes (little endian), put on stack

i32.store: get memory address and value, store value in memory as 4 bytes

Can also load/store small numbers by reading/writing fewer bytes, see

<http://webassembly.org/docs/semantics/>

Example: Area

```
int fact(int a, int b, int c) {  
    return ((c+a)*b + c*a) * 2;  
}
```

```
(module (type $type0 (func (param i32 i32 i32)  
                            (result i32))))  
  
(table 0 anyfunc) (memory 1)  
(export "memory" memory)  
(export "fact" $func0)  
  
(func $func0 (param $var0 i32)  
              (param $var1 i32)  
              (param $var2 i32) (result i32)  
  
    get_local $var2    c  
    get_local $var0    a  
    i32.add  
    get_local $var1    b  
    i32.mul  
    get_local $var2    c  
    get_local $var0    a  
    i32.mul  
    i32.add  
    i32.const 1  
    i32.shl             // shift left, i.e. *2  
    ))
```