# Compiler Construction

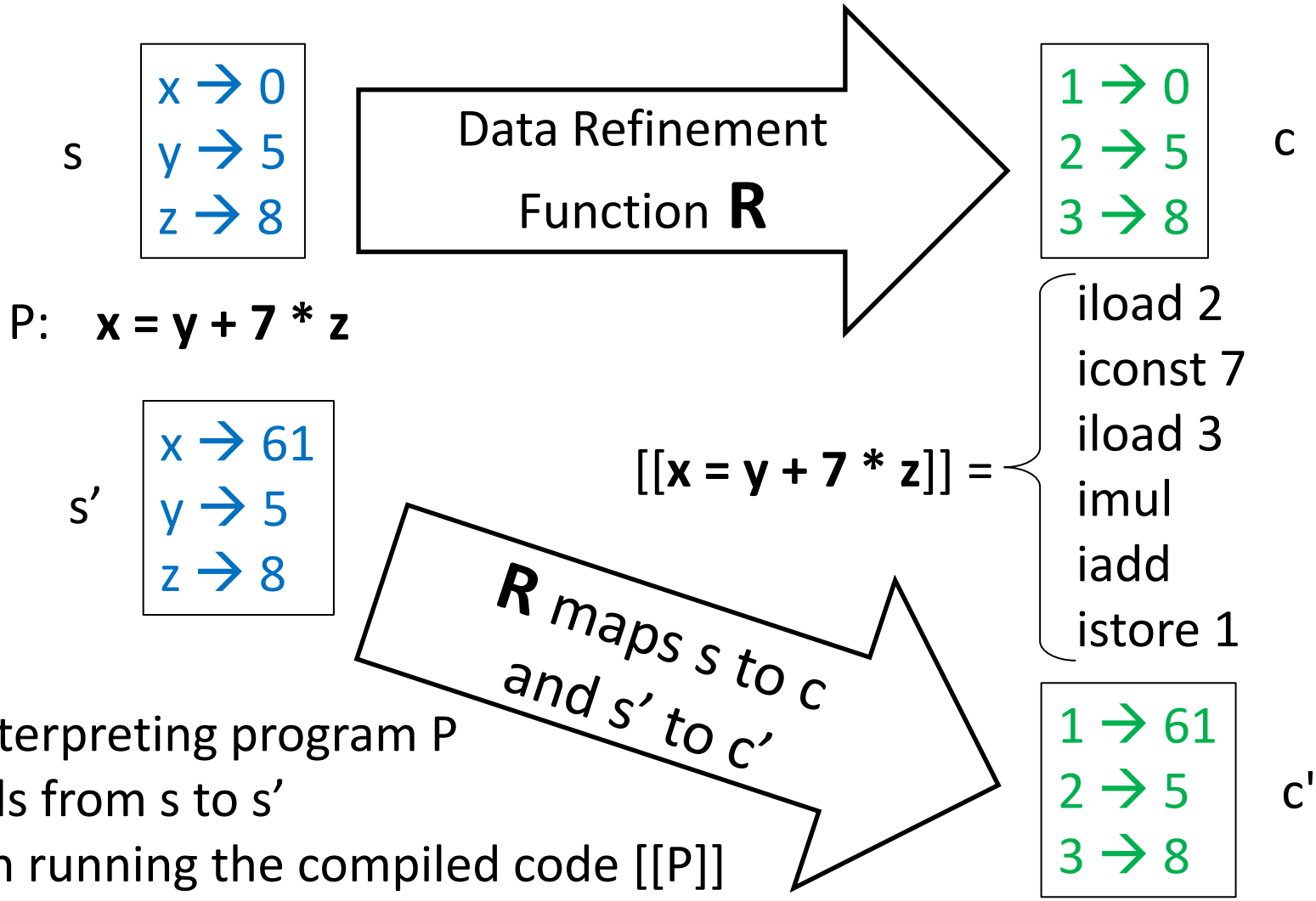
## Lecture 17

## Mapping Variables to Memory

# Original and Target Program have Different Views of Program State

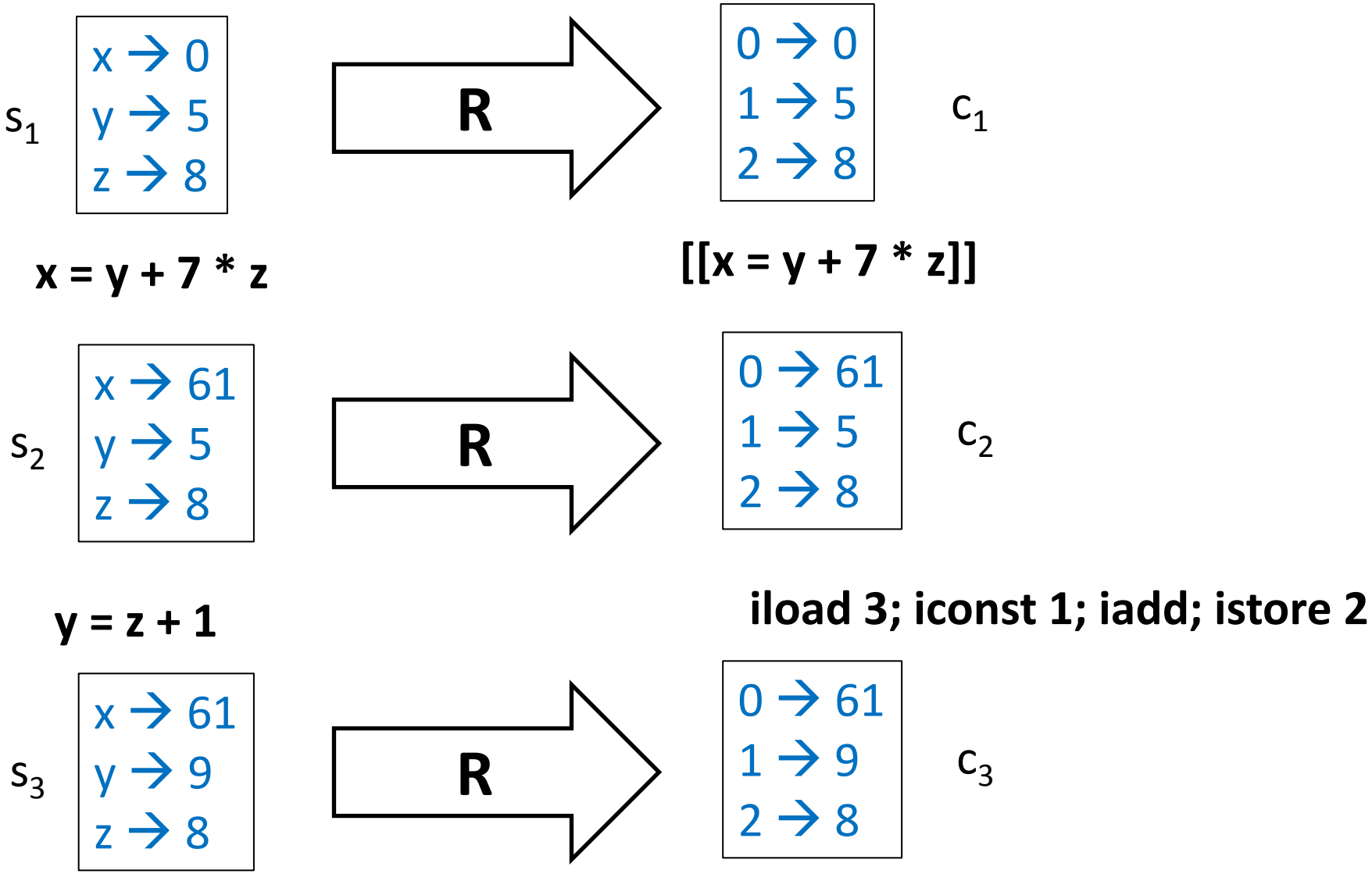
- Original program:
  - local variables given by names (any number of them)
  - each procedure execution has fresh space for its variables (even if it is recursive)
  - fields given by names
- Java Virtual Machine
  - local variables given by slots (0,1,2,…), any number
  - intermediate values stored in operand stack
  - each procedure **call** gets fresh slots and stack
  - fields given by names and object references
- **Machine code:** program state is a large arrays of bytes and a finite number of registers

# Compilation Performs Automated Data Refinement




If interpreting program P
leads from s to s'
then running the compiled code [[P]]
leads from R(s) to R(s')

# Inductive Argument for Correctness




(R may need to be a relation, not just function)

# A Simple Theorem

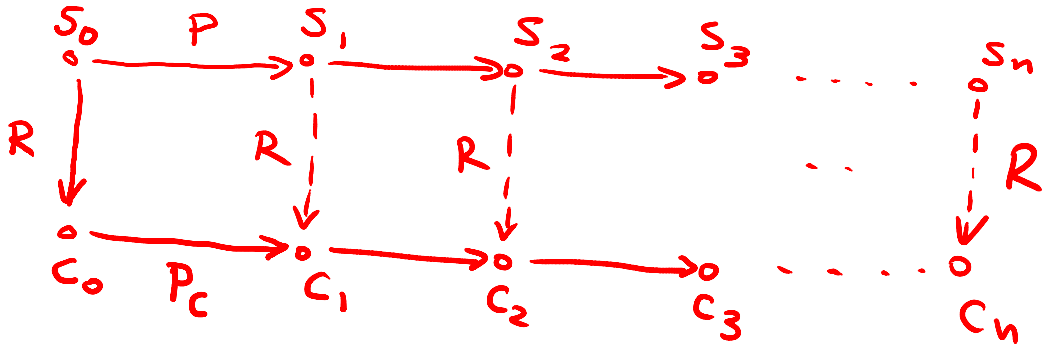
$P : S \rightarrow S$ is a program meaning function

$P_c : C \rightarrow C$ is meaning function for the compiled program

$R : S \rightarrow C$ is data representation function

Let $s_{n+1} = P(s_n)$, n = 0,1,… be interpreted execution

Let $c_{n+1} = P(c_n)$, n = 0,1,… be compiled execution




**Theorem:** If

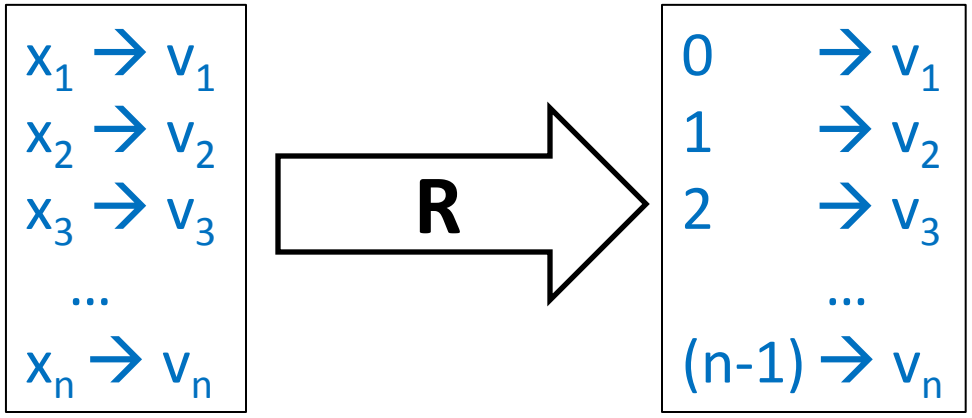
- $c_0 = R(s_0)$
- for all s, $P_c(R(s)) = R(P(s))$

then $c_n = R(c_n)$ for all n.

**Proof:** immediate, by induction. R is often called **simulation relation.**

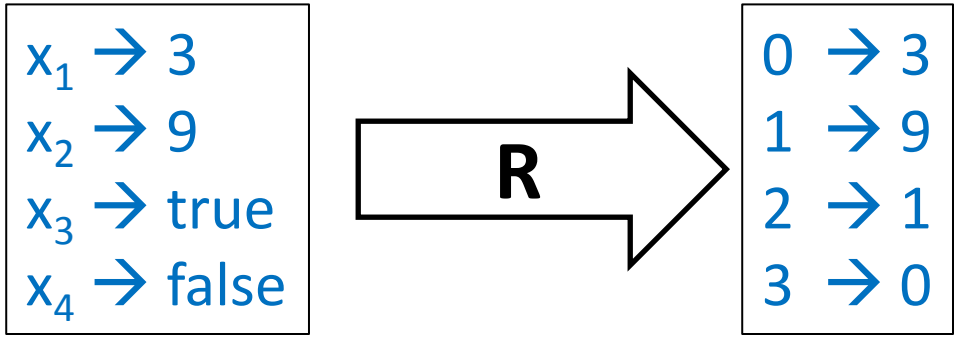
# Example of a Simple R

- Let the received, the parameters, and local variables, in their order of declaration, be $x_1, x_2 \ldots x_n$
- Then R maps program state with only integers like this:



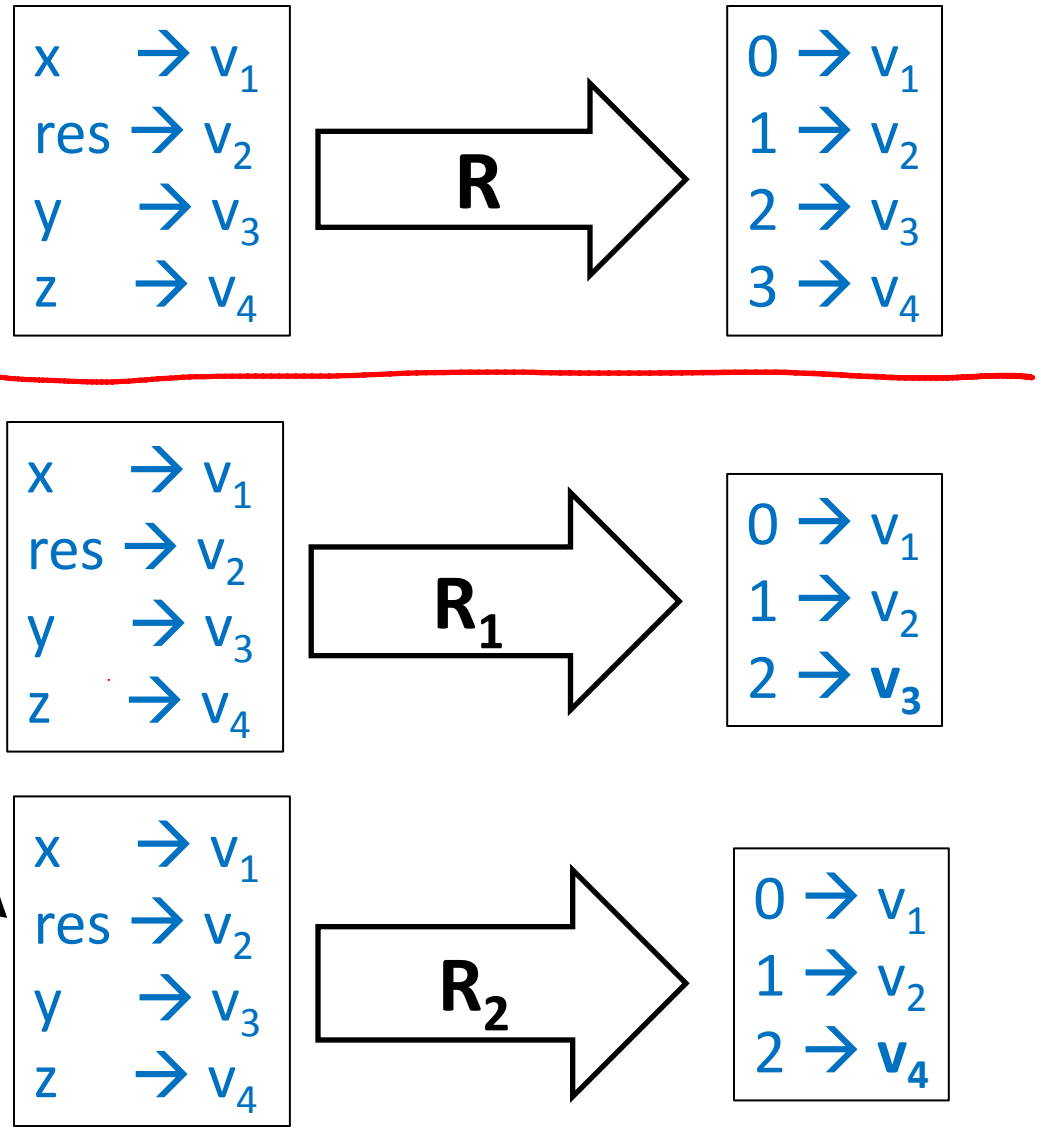
# R for Booleans

- Let the received, the parameters, and local variables, in their order of declaration, be $x_1, x_2 \ldots x_n$
- Then R maps program state like this, where $x_1$ and $x_2$ are integers but $x_3$ and $x_4$ are Booleans:



# R that depends on Program Point

```
def main(x:Int) {
  var res, y, z: Int
  if (x>0) {
    y = x + 1
    res = y
  } else {
     z = -x - 10
     res = z
  }
  ...
}
```



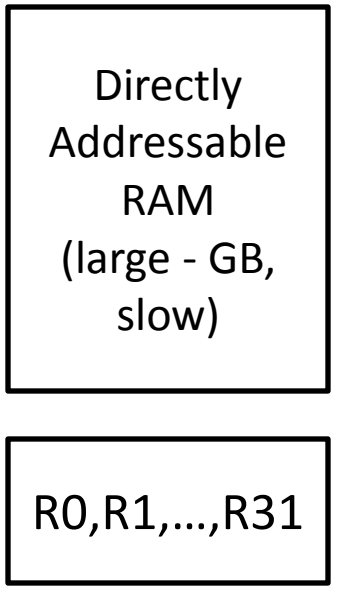

Map y,z to same slot.

Consume fewer slots!

# Packing Variables into Memory

- If values are not used at the same time, we can store them in the same place
- This technique arises in
  - **Register allocation**: store frequently used values in a bounded number of fast registers
  - ‘malloc’ and ‘free’ manual memory management: *free* releases memory to be used for later objects
  - Garbage collection, e.g. for JVM, and .NET as well as languages that run on top of them (e.g. Scala)

# Register Machines




Better for most purposes than stack machines

- closer to modern CPUs (RISC architecture)
- closer to control-flow graphs
- simpler than stack machine

Example: ARM architecture

From article on RISC architectures:

"The ARM architecture dominates the market for high performance, low power, low cost embedded systems (typically 100–500 MHz in 2008). ARM Ltd., which licenses intellectual property rather than manufacturing chips, reported 10 billion licensed chips shipped in early 2008 [7]. ARM is deployed in countless mobile devices such as: Samsung Galaxy (ARM11), Apple iPods (custom ARM7TDMI SoC) Apple iPhone (Samsung ARM1176JZF), Palm and PocketPC PDAs and smartphones (Marvell XScale family, Samsung SC32442 - ARM9), Nintendo Game Boy Advance (ARM7TDMI), Nintendo DS (ARM7TDMI, ARM946E-S), Sony Network Walkman (Sony in-house ARM based chip)

# Basic Instructions of Register Machines

$R_i \leftarrow Mem[R_j]$ load

$Mem[R_j] \leftarrow R_i$ store

$R_i \leftarrow R_j * R_k$ compute: for an operation *

Efficient register machine code uses as few loads and stores as possible.

# State Mapped to Register Machine

Both dynamically allocated heap and stack expand

- heap need not be contiguous can request more memory from the OS if needed
- stack grows downwards

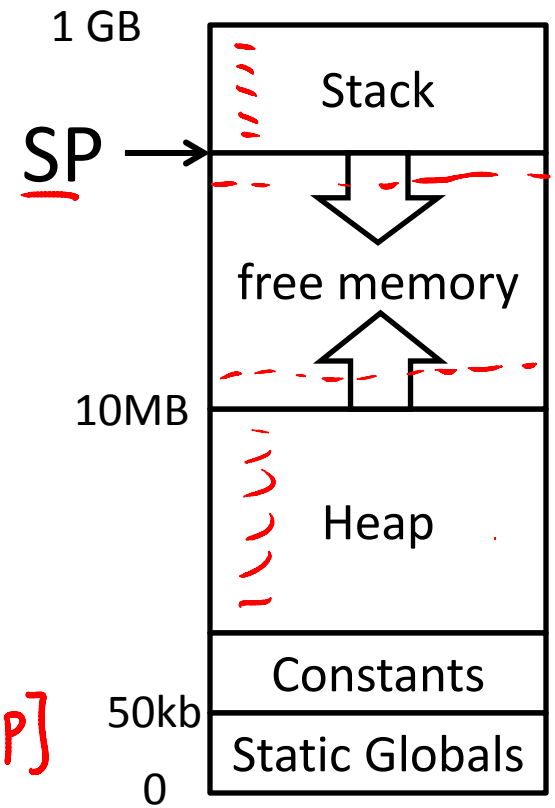
Heap is more general:

- Can allocate, read/write, and deallocate, in any order
- Garbage Collector does deallocation automatically
  - Must be able to find free space among used one, group free blocks into larger ones (compaction),…

Stack is more efficient:

- allocation is simple: increment, decrement
- top of stack pointer (SP) is often a register
- if stack grows towards smaller addresses:
  - to allocate N bytes on stack (**push**): **SP := SP - N**
  - to deallocate N bytes on stack (**pop**): **SP := SP + N**

Mem[SP]




Exact picture may depend on hardware and OS

# JVM vs General Register Machine Code

JVM:

imul

Register Machine:

R1 ← Mem[SP]

SP = SP + 4

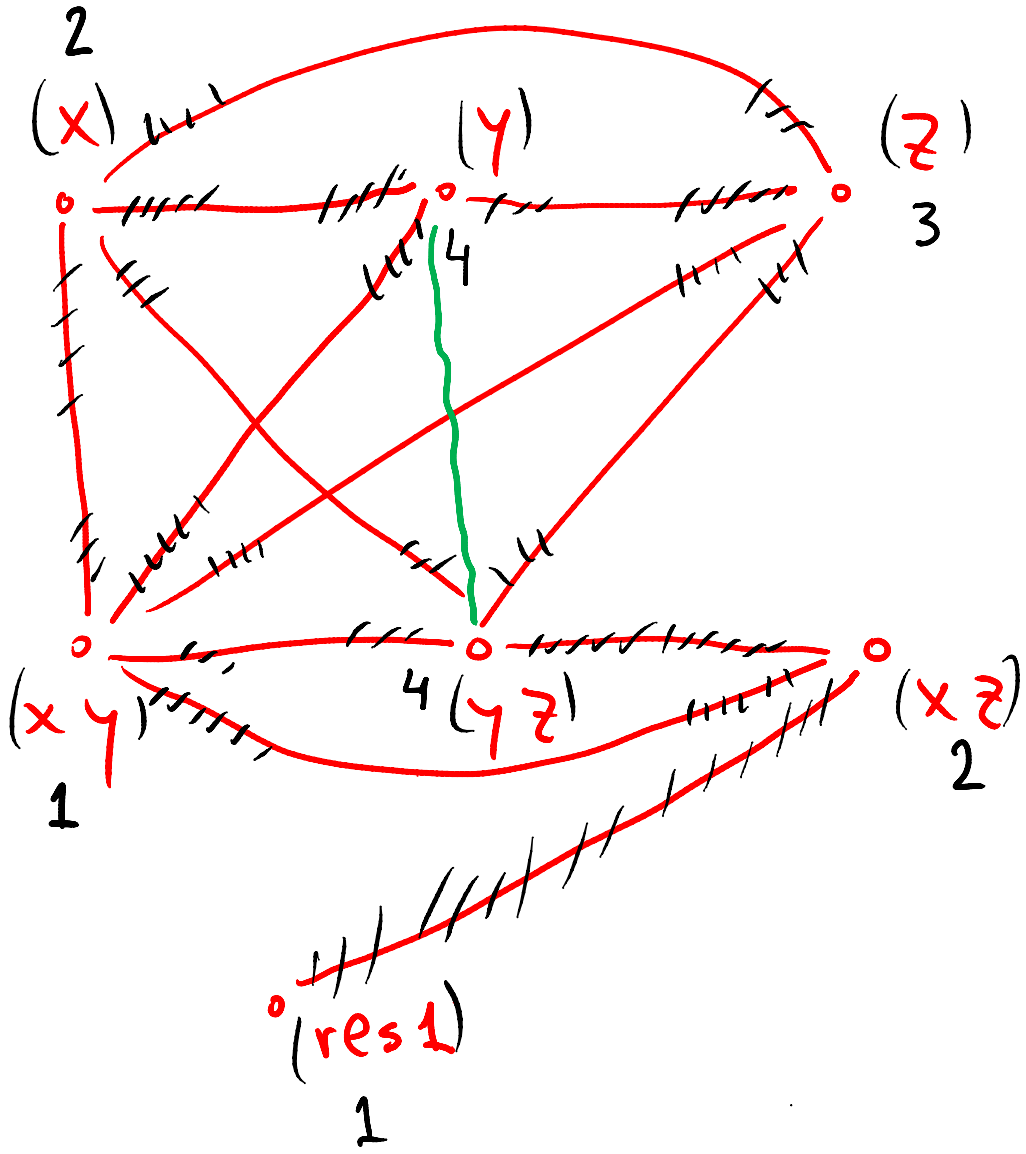
R2 ← Mem[SP]

R2 ← R1 * R2

Mem[SP] ← R2

$(x,y) \in E$ if $\{x,y\} \subseteq live(p)$

$K=4$ registers




res 1 : R1

xz : R2

| |
|---|
| y : R4 |
| yz : R4 |
| z : R3 |
| x : R2 |
| xy : R1 |