Register Machines

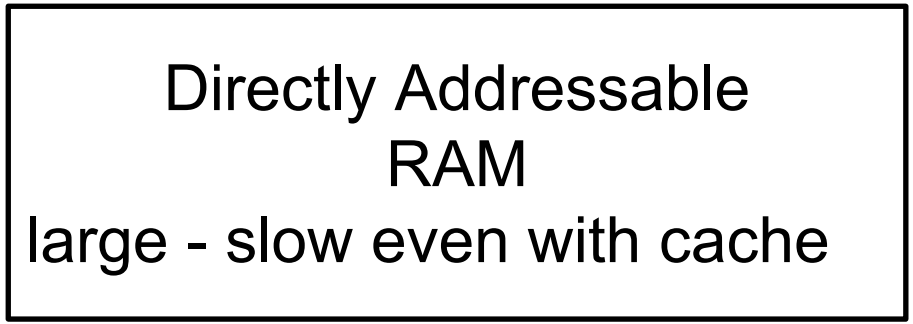
Better for most purposes than stack machines

- closer to modern CPUs (RISC architecture)
- closer to control-flow graphs
- simpler than stack machine (but register set is finite)

Examples:

ARM architecture

RISC V: http://riscv.org/



A **few fast** registers

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

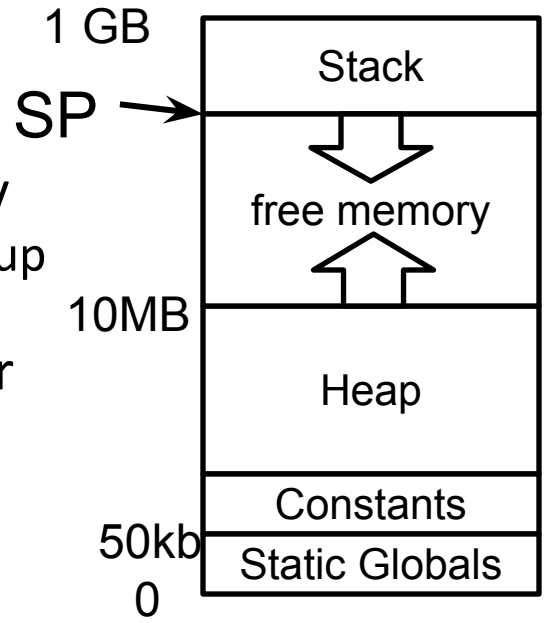
Both dynamically allocated heap and stack expand

Heap is **more general**:

- Can allocate, read/write, 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 efficient: top of stack pointer (SP) is a register

- allocation is simple: increment, decrement
- to allocate N bytes on stack (**push**): **SP := SP - N**
- to deallocate N bytes on stack (**pop**): **SP := SP + N**



Exact picture varies depend on hardware, OS, language runtime

WASM vs General Register Machine Code
Naïve Correct Translation

WASM:

imul.32

Register Machine:

R1 ← Mem[SP]

SP = SP + 4

R2 ← Mem[SP]

R2 ← R1 * R2

Mem[SP] ← R2

Register Allocation

How many variables?

x,y,z,xy,xz,res1

Do we need 7 distinct registers if we wish to avoid load and stores?

```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
xz = x*z
res1 = xy + yz
m[3] = res1 + xz
```

7 variables:
x,y,z,xy,yz,xz,res1

```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
y = x*z          // reuse y
x = xy + yz      // reuse x
m[3] = x + y
```

can do it with 5 only!