# CSCI 742 - Compiler Construction

Lecture 38
Register Allocation
Instructor: Hossein Hojjat

April 27, 2018

# Register Machines

- Debate topic: stack or register architecture?

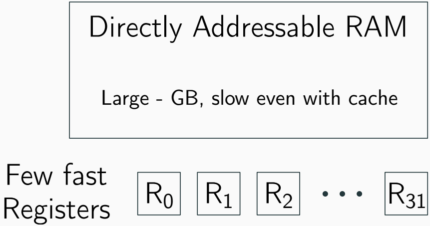
see e.g. Yunhe Shi et al. “Virtual Machine Showdown: Stack Versus Registers”
ACM Transactions on Architecture and Code Optimization, Vol. 4, No. 4, 2008

Register Machines Benefit:

- Closer to modern CPUs (RISC architecture) and control-flow graphs

Examples:

- RISC: ARM architecture, RISC-V
- CISC: x86 architecture



# Basic Instructions of Register Machines

- $R_i \leftarrow \mathrm{Mem}[R_j]$ load
- $\mathrm{Mem}[R_j] \leftarrow R_i$ store
- $R_i \leftarrow R_j \oplus R_k$ compute: for an operation $\oplus$

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

# State Mapped to Register Machine

Both dynamically allocated heap and stack expand

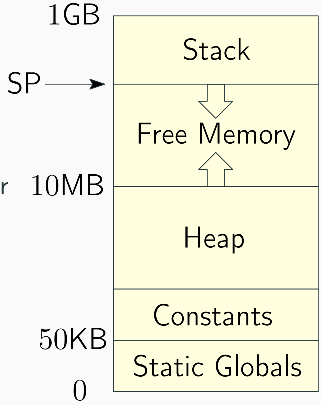
- Heap need not be contiguous; can request more memory from the OS if needed
- Stack grows downwards

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): $SP := SP + N$




(Exact picture may depend on hardware and operating system)

# JVM vs. General Register Machine Code

- Naïve Correct Translation

| JVM: | Register Machine: |
|---|---|
| `imul` | $R_1 \leftarrow \text{Mem[SP]}$ |
| | $\text{SP} = \text{SP} + 4$ |
| | $R_2 \leftarrow \text{Mem[SP]}$ |
| | $R_2 \leftarrow R_1 * R_2$ |
| | $\text{Mem[SP]} \leftarrow R_2$ |

# Using Registers

- Variables usually refer to memory
- `&x` yields a memory location
- Need to load variables into registers to perform operations on them

1. Load from memory into registers
2. Perform operation on registers
3. Store results from registers back to memory

# Example: How many variables?

- Do we need 7 distinct registers if we wish to avoid load and stores?
- Variables: `x , y , z , xy , yz , xz , r`

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

# Example: How many variables?

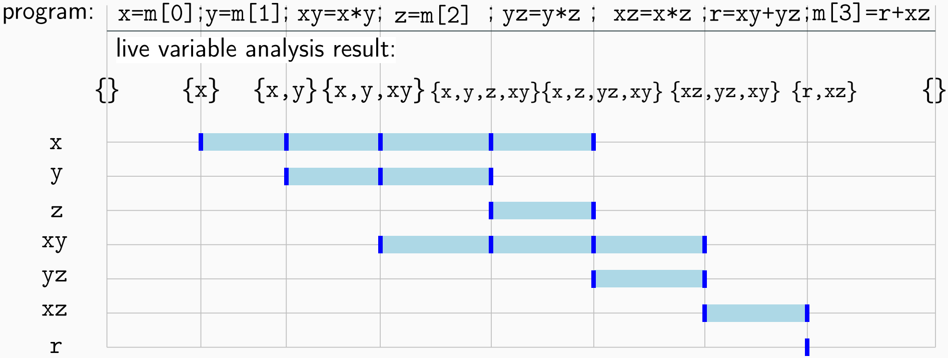
- Do we need 7 distinct registers if we wish to avoid load and stores?
- Variables: `x , y , z , xy , yz , xz , r`

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

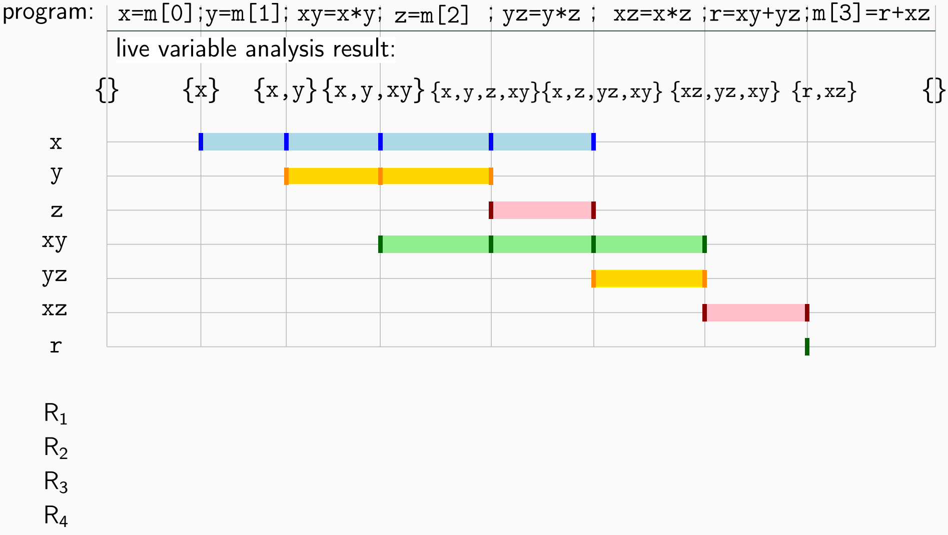
```
x = m[0];
y = m[1];
xy = x * y;
z = m[2];
yz = y*z;
y = x*z;      // reuse y
x = xy + yz; // reuse x
m[3] = x + y;
```

- Can do it with 5 only!

# Idea of Register Allocation

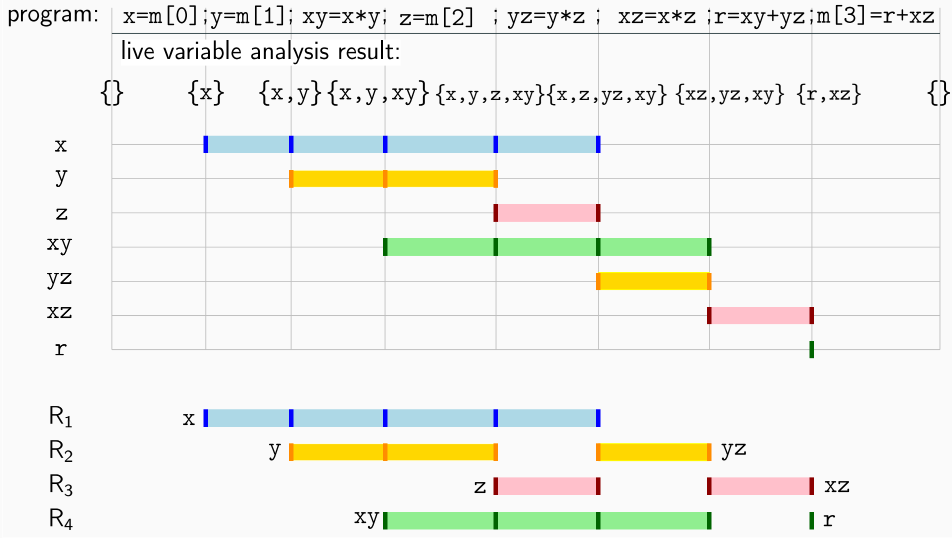


# Idea of Register Allocation



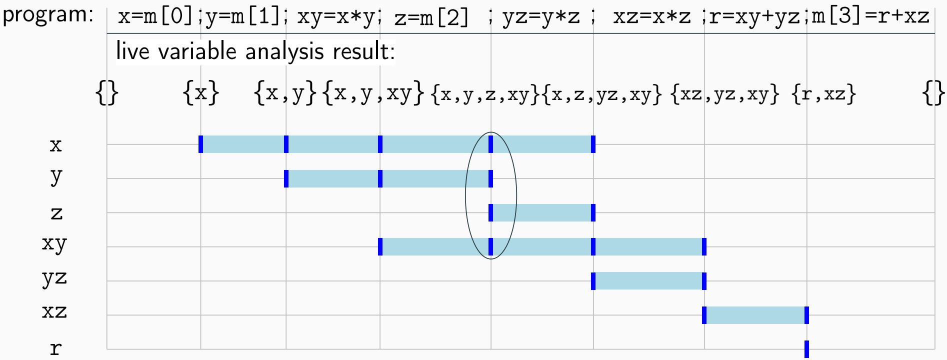

- Each color denotes a register
- Avoid overlap of same colors
- 4 registers are enough for this 7-variable program

# Idea of Register Allocation

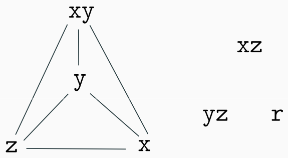



- Each color denotes a register
- Avoid overlap of same colors
- 4 registers are enough for this 7-variable program

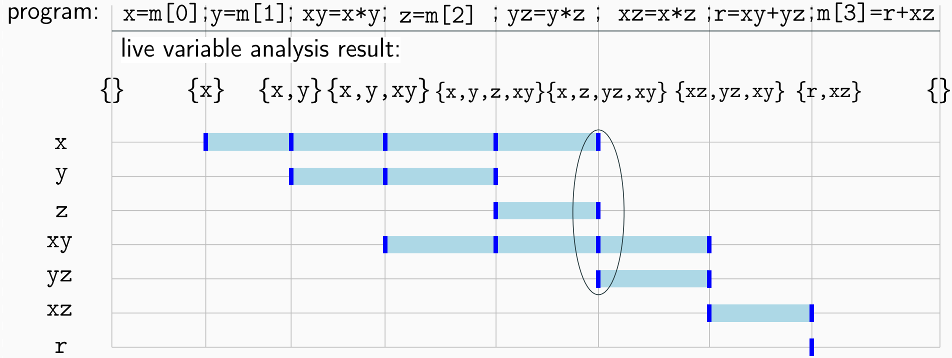
# Idea of Register Allocation



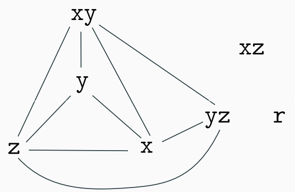

- For each pair of variables determine if there is a point at which they are both alive
- Construct interference graph



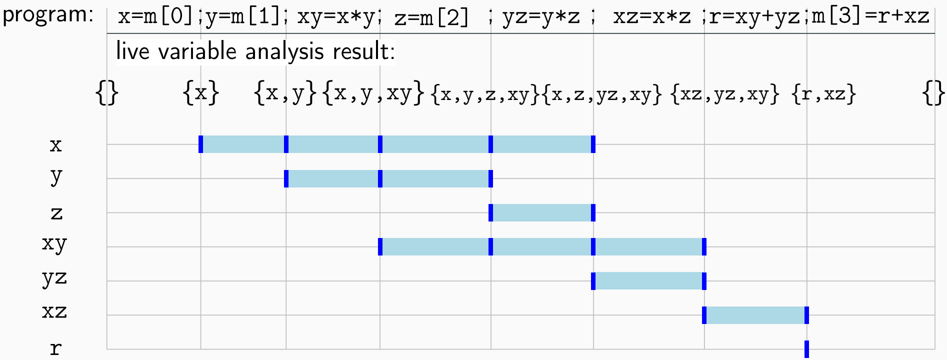
# Idea of Register Allocation



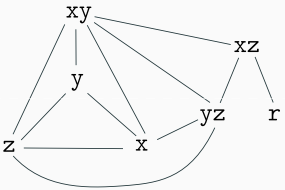

- For each pair of variables determine if there is a point at which they are both alive
- Construct interference graph



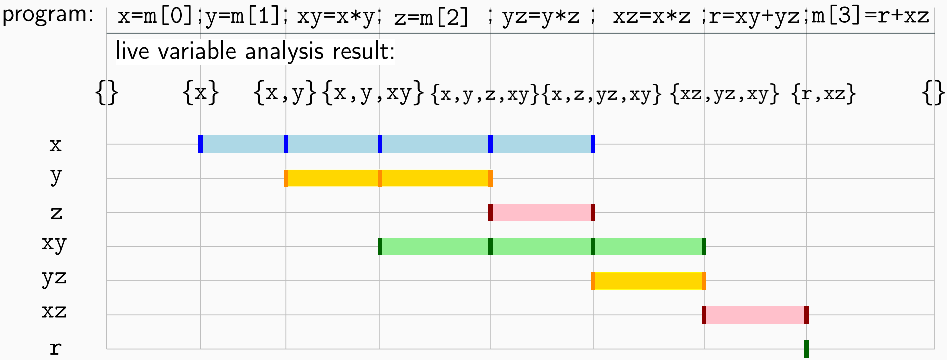
# Idea of Register Allocation



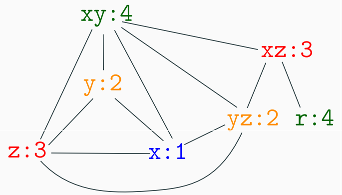

- For each pair of variables determine if there is a point at which they are both alive
- Construct interference graph



# Idea of Register Allocation



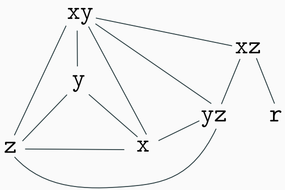

- Need to assign colors (register numbers) to nodes such that:
- If there is an edge between nodes, then those nodes have different colors
- Standard graph vertex coloring problem



# Register Interference Graph (RIG)

- Indicate whether there exists a point of time where both variables are alive
- Look at the sets of live variables at all program points after running live-variable analysis
- If two variables occur together, draw an edge
- We aim to assign different registers to such these variables
- Finding assignment of variables to $K$ registers:
  corresponds to coloring graph using $K$ colors

# Graph Coloring Problem

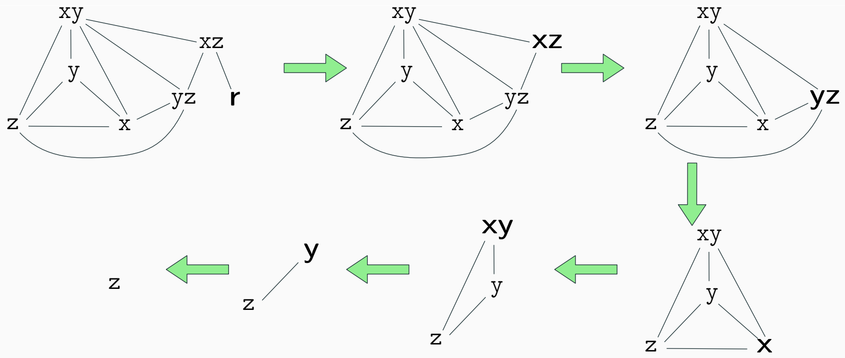



- NP hard
- In practice, there are heuristics that work for typical graphs
- If we cannot fit it all variables into registers, perform a **spill**: Store variable into memory and load later when needed

**Simplify:**

- If there is a node with less than $K$ neighbors, we will always be able to color it!
- So we can remove such node from the graph
  - (if it exists, otherwise remove other node)
- This reduces graph size. It is useful, even though incomplete

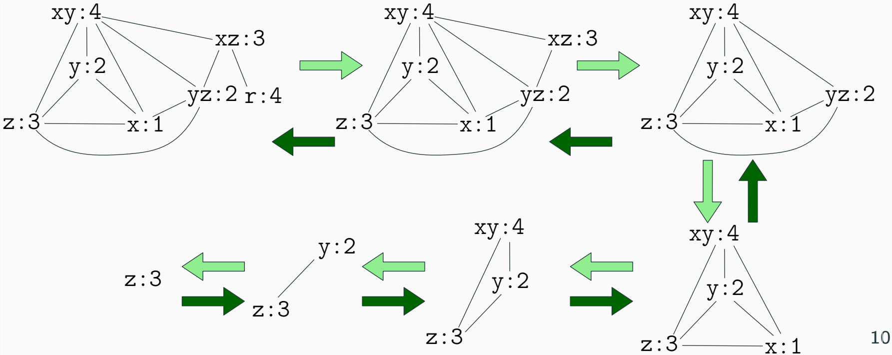
(e.g. can color planar by at most 4 colors, yet can have nodes with many neighbors)



# Heuristic for Coloring with $K$ Colors

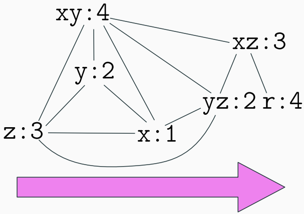
**Select:**

- Assign colors backwards, adding nodes that were removed
- If the node was removed because it had $< K$ neighbors, we will always find a color
- If there are multiple possibilities, we can choose any color



# Use Computed Registers

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




```
R1 = m[0]
R2 = m[1]
R4 = R1*R2
R3 = m[2]
R2 = R2*R3
R3 = R1*R3
R4 = R4 + R2
m[3] = R4 + R3
```

# Summary of Heuristic for Coloring

**Simplify (forward, safe):**

If there is a node with less than $K$ neighbors, we will always be able to color it, so we can remove it from the graph

**Potential Spill (forward, speculative):**

If every node has $K$ or more neighbors, we still remove one of them we mark it as node for potential spilling. Then remove it and continue

**Select (backward):**

Assign colors backwards, adding nodes that were removed

- If we find a node that was spilled, we check if we are lucky, that we can color it. If yes, continue
- If not, insert instructions to save and load values from memory (**actual spill**)
  Restart with new graph
  (graph is now easier to color as we killed a variable)