Code Generation

Ronghui Gu

Fall 2022

Columbia University

* Course website: https://verigu.github.io/4115Fall2022/

The Final Exam

The Final Exam

75 minutes

Closed book

One double-sided sheet of notes of your own devising

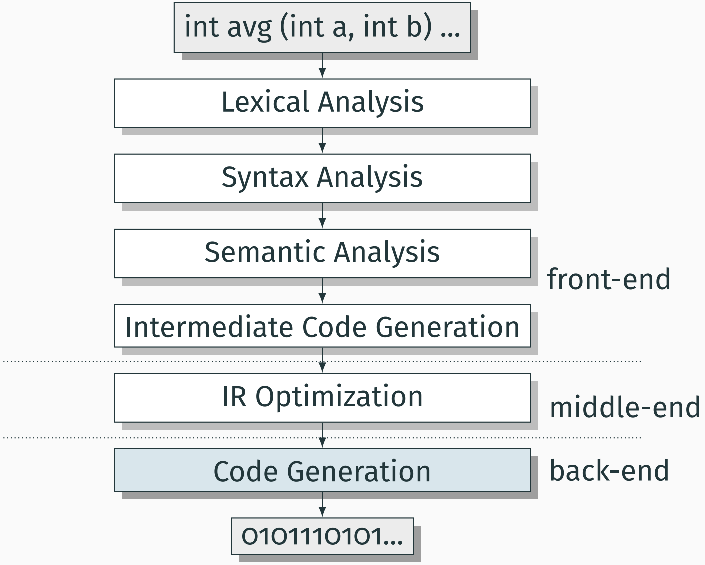
Anything discussed in class is fair game

Little, if any, programming

Details of OCaml/C/C++/Java syntax not required

Code Generation

Code Generation



Code Generation

- Choose the appropriate machine instructions for each IR instruction.
- Mange finite machine resources (e.g., registers).
- Implement runtime environment.

The Memory Hierarchy

Memory tradeoffs: there is an enormous tradeoff between speed and size in memory.

- Registers: 1 ns, 1 KB
- Per-CPU cache: 5 ns, 128 KB
- Shared cache: 25 ns, 6 MB
- Main memory: 100 ns, 16 GB
- Disk: 10 ms, 1 TB
- Network: 100 ms, huge

The Challenges of Code Generation

Goal: Try to get the best of all worlds by using multiple types of memory.

Challenges:

- All variables in TAC live in memory.
- Position objects in a way that takes maximum advantage of the memory hierarchy.
- Do so without hints from the programmer.

Register Allocation

Using registers intelligently is a critical step in any compiler.

Register allocation is the process of assigning variables to registers and managing data transfer in and out of registers.

Challenges:

- In TAC, there are an unlimited number of variables.
- On a physical machine there are a small number of registers.

Register Allocation

Explore three algorithms for register allocation:

- Naive (“no”) register allocation.
- Linear scan register allocation.
- Graph-coloring register allocation.

Naive Register Allocation

Naive Register Allocation

Idea: store every value in main memory, loading values only when they're needed.

- Insert load to pull the values from memory into registers before access.
- Insert store to store the values back into memory after access.

Naive Register Allocation

```
a = b + c;
```

```
d = a;
```

Naive Register Allocation

```
a = b + c;
lw $t0, -12(fp)
lw $t1, -16(fp)
add $t2, $t0, $t1
sw $t2, -8(fp)
d = a;
```

Naive Register Allocation

```
a = b + c;

lw $t0, -12(fp)

lw $t1, -16(fp)

add $t2, $t0, $t1

sw $t2, -8(fp)

d = a;

lw $t0, -8(fp)

sw $t0, -20(fp)
```

Naive Register Allocation

Advantages:

Disadvantages:

Naive Register Allocation

Advantages:

- Can easily translate IR to assembly.
- Never need to worry about running out of registers.

Disadvantages:

Naive Register Allocation

Advantages:

- Can easily translate IR to assembly.
- Never need to worry about running out of registers.

Disadvantages:

- Unnecessary loads and stores.
- Wastes space.
- Too slow.

Linear Scan Register Allocation

A Better Allocator

Goal: try to hold as many variables in registers as possible.

Register consistency:

- At each program point, each variable must be in the same location.
- At each program point, each register holds at most one live variable.

Live Intervals

Live interval: the smallest subrange of the IR code containing all a variable's live ranges.

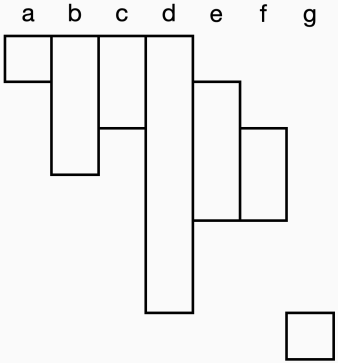
```
e = d + a;

f = b + c;

f = f + b;

d = e + f;

  g = d;
```

Live Intervals

Live interval: the smallest subrange of the IR code containing all a variable’s live ranges.

```
 { d, b, c, a }
   e = d + a;
  { e, b, c }
   f = b + c;
  { e, f, b }
   f = f + b;
    { e, f }
   d = e + f;
      { d }
     g = d;
      { g }
```

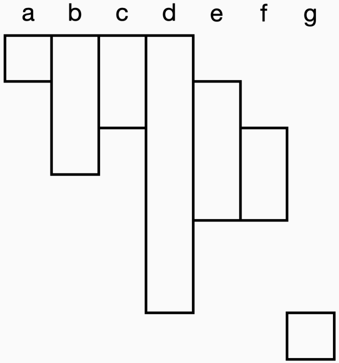
Linear Scan

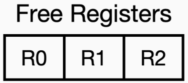


Free Registers

R0	R1	R2	R3

Another Example





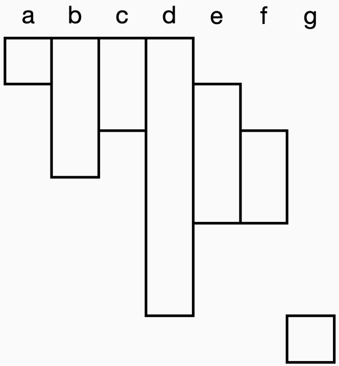
Register Spilling

If a register cannot be found for a variable v, we may need to spill a variable.

When a variable is spilled, it is stored in memory rather than a register.

Spilling is slow, but sometimes necessary.

Another Example



Free Registers

R0	R1	R2

Linear Scan Algorithm

Advantages

Disadvantages

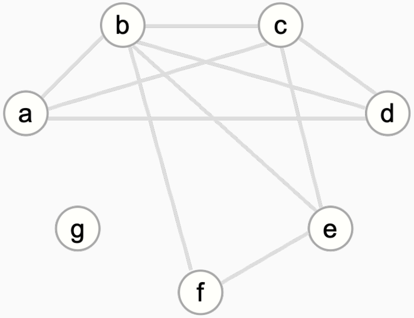
Graph-coloring Register Allocation

The Register Interference Graph (RIG)

```
{ d, b, c, a }
  e = d + a;
 { e, b, c }
  f = b + c;
 { e, f, b }
  f = f + b;
  { e, f }
  d = e + f;
   { d }
   g = d;
   { g }
```

The Register Interference Graph (RIG)

```
{ d, b, c, a }
  e = d + a;
 { e, b, c }
  f = b + c;
 { e, f, b }
  f = f + b;
  { e, f }
  d = e + f;
   { d }
   g = d;
   { g }
```

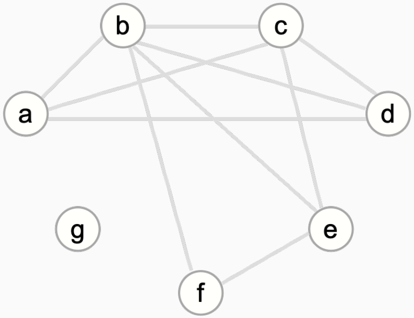


Graph Coloring

Graph coloring is NP-complete if there are at least three registers.

Chaitin’s Algorithm: we can delete the node with fewer than k edges from the graph and color what remains with k colors.

Chaitin’s Algorithm



One Problem

What if we can't find a node with fewer than k neighbors?

Choose and remove an arbitrary node, marking it troublesome.

When adding node back in, it may be possible to find a valid color.

Otherwise, we have to spill that node.

Chaitin’s Algorithm Reloaded