Lecture S:

Register Allocation

(Backend 3)

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- HW4 — *due* Thursday night (late days allowed)
- Thursday Section — Review of Backend Material & General Q&A
- Friday Lecture — Hal (Garbage Collection)
- Please nominate great TAs for the Bandes award!
  - (for this and other courses)

Final Exam

- **EXAM** — Tuesday 2:30-4:20pm **here** (CSE2 G10)
- Closed book like midterm
- Ok to have two 5x8 hand-written note-cards (you can keep your midterm card or replace it)
  - ✦ blank index cards available in class, section & review
- Topic list and old exams are posted on the class website
- Review Q&A Monday 2:30-3:30pm (**CSE2 G01**, not G10)
- HW4 sample solutions available outside Gilbert's office (CSE2 235) starting late Sunday (TAs will announce on Ed when available) — also available during Monday review

Outline

The Problem

Limited Registers (K)

- Intermediate code often assumes ∞ registers (e.g. SSA)
  - ✦ but the real machine has K registers
  - ✦ e.g. x86 K=16; most RISC ISAs K=32
- Goals
  - ✦ produce correct code that uses K or fewer registers
  - ✦ minimize use of additional loads and stores
  - ✦ minimize space required for spilled values
  - ✦ Run efficienty — e.g. O(n), O(n log n), …maybe $O(n^2)$

Register Allocation — Spilling

- If we can't keep everything in K registers…
  - At each point in the code, pick which values get to stay in registers
  - Insert code to move values between registers and memory
  - Minimize amount of inserted memory operations
  - (recall that if we insert additional operations, we may want to rerun the instruction scheduler after register allocation)

Different Ways to Spill

- Consider a single register and the value in it
- Dirty Values — The value only exists in the register. Inserting a spill requires adding both **store** and **load** instructions
- Clean Values — The value in the register *also exists* in memory. (i.e. the program already stores it for other reasons) So, inserting a spill only requires adding a **load**
- Rematerialized Values — The value in the register can also be *recomputed from values in other registers*. In this case, no load or store is necessary, but additional arithmetic is required to reconstruct the value

terminology from Cooper & Torczon

Allocation vs. Assignment

- Allocation — decide which values to keep in a register
- Assignment — decide which register holds which value
- Compiler must do both, but technically these are slightly different questions
- We will not focus much on the distinction for our discussion

Outline

The “Best” Algorithm

Local Register Allocation

- (recall local = in a basic block)
- Doing this produces reasonable register usage inside a basic block
- However, local register allocation is sub-optimal
  - ✦ There is no opportunity for reasoning about overlapping usage at the boundaries between basic blocks
  - ✦ There is no opportunity to reason about values that have to be retained (and used?) over the entire course of executing a given loop

Just Go For It!

- Keep a list of all available registers and which values they contain
- Scan the code top to bottom
- Allocate registers to hold values as needed
- Free registers as soon as possible (no longer live)
  - ✦ so we need a liveness analysis first
- What if we run out of registers?

Save the Last for Best!

- If no registers are free and we need a register, we need to *evict / spill* the contents of some register
- Criterion
  - ✦ Look at all of the ***next uses*** for each value in a register
  - ✦ Evict/Spill whichever register will be needed last
- Why is this a good idea?
- Yes, this choice maximizes the amount of time the spilled register will be available for

Our Unscheduled Example

```
 1 LOAD   v1 ← x
 2 LOAD   v2 ← y
 3 LOAD   v3 ← nx
 4 LOAD   v4 ← ny
 5 FMUL   v5 ← v1 * v3
 6 FMUL   v6 ← v2 * v4
 7 FADD   v7 ← v5 + v6
 8 FMUL   v8 ← v3 * v3
 9 FMUL   v9 ← v4 * v4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

```
r1
r2
r3
r4
r5
r6
```

Next Use

```
 v1
 v2
 v3
 v4
 v5
 v6
 v7
 v8
 v9
v10
v11
v12
v13
v14
v15
v16
v17
```

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   v2 ← y
 3 LOAD   v3 ← nx
 4 LOAD   v4 ← ny
 5 FMUL   v5 ← v1 * v3
 6 FMUL   v6 ← v2 * v4
 7 FADD   v7 ← v5 + v6
 8 FMUL   v8 ← v3 * v3
 9 FMUL   v9 ← v4 * v4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

r1 v1
r2
r3
r4
r5
r6

Next Use

v1	5
v2	
v3	
v4	
v5	
v6	
v7	
v8	
v9	
v10	
v11	
v12	
v13	
v14	
v15	
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   v3 ← nx
 4 LOAD   v4 ← ny
 5 FMUL   v5 ← v1 * v3
 6 FMUL   v6 ← v2 * v4
 7 FADD   v7 ← v5 + v6
 8 FMUL   v8 ← v3 * v3
 9 FMUL   v9 ← v4 * v4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

r1	v1
r2	v2
r3	
r4	
r5	
r6	

Next Use

v1	5
v2	6
v3	
v4	
v5	
v6	
v7	
v8	
v9	
v10	
v11	
v12	
v13	
v14	
v15	
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   v4 ← ny
 5 FMUL   v5 ← v1 * v3
 6 FMUL   v6 ← v2 * v4
 7 FADD   v7 ← v5 + v6
 8 FMUL   v8 ← v3 * v3
 9 FMUL   v9 ← v4 * v4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

r1	v1
r2	v2
r3	**v3**
r4	
r5	
r6	

Next Use

v1	5
v2	6
v3	**5**
v4	
v5	
v6	
v7	
v8	
v9	
v10	
v11	
v12	
v13	
v14	
v15	
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   v5 ← v1 * v3
 6 FMUL   v6 ← v2 * v4
 7 FADD   v7 ← v5 + v6
 8 FMUL   v8 ← v3 * v3
 9 FMUL   v9 ← v4 * v4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

r1	v1
r2	v2
r3	v3
r4	**v4**
r5	
r6	

Next Use

v1	5
v2	6
v3	5
v4	**6**
v5	
v6	
v7	
v8	
v9	
v10	
v11	
v12	
v13	
v14	
v15	
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   v6 ← v2 * v4
 7 FADD   v7 ← v5 + v6
 8 FMUL   v8 ← v3 * v3
 9 FMUL   v9 ← v4 * v4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

r1	v1
r2	v2
r3	v3
r4	v4
r5	v5
r6	

Next Use

v1	15
v2	6
v3	8
v4	6
v5	7
v6	
v7	
v8	
v9	
v10	
v11	
v12	
v13	
v14	
v15	
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   v7 ← v5 + v6
 8 FMUL   v8 ← v3 * v3
 9 FMUL   v9 ← v4 * v4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

r1	v1
r2	v2
r3	v3
r4	v4
r5	v5
r6	v6

Next Use

v1	15
v2	18
v3	8
v4	9
v5	7
v6	7
v7	
v8	
v9	
v10	
v11	
v12	
v13	
v14	
v15	
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   v8 ← v3 * v3
 9 FMUL   v9 ← v4 * v4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

```
r1 v1
r2 v2
r3 v3
r4 v4
r5 v7
r6
```

Next Use

```
 v1  15
 v2  18
 v3  8
 v4  9
 v5
 v6
 v7  12
 v8
 v9
v10
v11
v12
v13
v14
v15
v16
v17
```

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   v9 ← v4 * v4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

r1	v1
r2	v2
r3	v3
r4	v4
r5	v7
r6	**v8**

Next Use

v1	15
v2	18
v3	**14**
v4	9
v5	
v6	
v7	12
v8	**10**
v9	
v10	
v11	
v12	
v13	
v14	
v15	
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD  v10 ← v8 + v9
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

SPILL! @v2

RESTORE! @v2

Registers

r1	v1
r2	v9
r3	v3
r4	v4
r5	v7
r6	v8

Next Use

v1	15
v2	18
v3	14
v4	17
v5	
v6	
v7	12
v8	10
v9	10
v10	
v11	
v12	
v13	
v14	
v15	
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV  v11 ← 1.0 / v10
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

SPILL! @v2

RESTORE! @v2

Registers

```
r1 v1
r2 v10
r3 v3
r4 v4
r5 v7
r6
```

Next Use

```
 v1  15
 v2  18
 v3  14
 v4  17
 v5
 v6
 v7  12
 v8
 v9
v10  11
v11
v12
v13
v14
v15
v16
v17
```

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV   r2 ← 1.0 / r2     SPILL! @v2
12 FMUL  v12 ← 2.0 * v7
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14     RESTORE! @v2
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

r1	v1
r2	**v11**
r3	v3
r4	v4
r5	v7
r6	

Next Use

v1	15
v2	18
v3	14
v4	17
v5	
v6	
v7	12
v8	
v9	
v10	
v11	**13**
v12	
v13	
v14	
v15	
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV   r2 ← 1.0 / r2
12 FMUL   r5 ← 2.0 * r5
13 FMUL  v13 ← v12 * v11
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

SPILL! @v2

RESTORE! @v2

Registers

```
r1 v1
r2 v11
r3 v3
r4 v4
r5 v12
r6
```

Next Use

```
 v1  15
 v2  18
 v3  14
 v4  17
 v5
 v6
 v7
 v8
 v9
v10
v11  13
v12  13
v13
v14
v15
v16
v17
```

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV   r2 ← 1.0 / r2
12 FMUL   r5 ← 2.0 * r5
13 FMUL   r2 ← r5 * r2
14 FMUL  v14 ← v3 * v13
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

SPILL! @v2

RESTORE! @v2

Registers

```
r1 v1
r2 v13
r3 v3
r4 v4
r5
r6
```

Next Use

```
 v1  15
 v2  18
 v3  14
 v4  17
 v5
 v6
 v7
 v8
 v9
v10
v11
v12
v13  14
v14
v15
v16
v17
```

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV   r2 ← 1.0 / r2
12 FMUL   r5 ← 2.0 * r5
13 FMUL   r2 ← r5 * r2
14 FMUL   r3 ← r3 * r2
15 FSUB  v15 ← v1 - v14
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

SPILL! @v2

RESTORE! @v2

Registers

```
r1 v1
r2 v13
r3 v14
r4 v4
r5
r6
```

Next Use

```
 v1  15
 v2  18
 v3
 v4  17
 v5
 v6
 v7
 v8
 v9
v10
v11
v12
v13  17
v14  15
v15
v16
v17
```

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV   r2 ← 1.0 / r2      SPILL! @v2
12 FMUL   r5 ← 2.0 * r5
13 FMUL   r2 ← r5 * r2
14 FMUL   r3 ← r3 * r2
15 FSUB   r1 ← r1 - r3       RESTORE! @v2
16 STORE  xo ← v15
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

r1 v15
r2 v13
r3
r4 v4
r5
r6

Next Use

v1	
v2	18
v3	
v4	17
v5	
v6	
v7	
v8	
v9	
v10	
v11	
v12	
v13	17
v14	
v15	**16**
v16	
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV   r2 ← 1.0 / r2
12 FMUL   r5 ← 2.0 * r5
13 FMUL   r2 ← r5 * r2
14 FMUL   r3 ← r3 * r2
15 FSUB   r1 ← r1 - r3
16 STORE  xo ← r1
17 FMUL  v16 ← v4 * v13
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

SPILL! @v2

RESTORE! @v2

Registers

```
r1
r2 v13
r3
r4 v4
r5
r6
```

Next Use

```
 v1
 v2  18
 v3
 v4  17
 v5
 v6
 v7
 v8
 v9
v10
v11
v12
v13  17
v14
v15
v16
v17
```

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV   r2 ← 1.0 / r2      SPILL! @v2
12 FMUL   r5 ← 2.0 * r5
13 FMUL   r2 ← r5 * r2
14 FMUL   r3 ← r3 * r2
15 FSUB   r1 ← r1 - r3       RESTORE! @v2
16 STORE  xo ← r1
17 FMUL   r1 ← r4 * r2
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

Registers

```
r1 v16
r2
r3
r4
r5
r6
```

Next Use

```
 v1
 v2  18
 v3
 v4
 v5
 v6
 v7
 v8
 v9
v10
v11
v12
v13
v14
v15
v16  18
v17
```

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV   r2 ← 1.0 / r2
12 FMUL   r5 ← 2.0 * r5
13 FMUL   r2 ← r5 * r2
14 FMUL   r3 ← r3 * r2
15 FSUB   r1 ← r1 - r3
16 STORE  xo ← r1
17 FMUL   r1 ← r4 * r2
18 FSUB  v17 ← v2 - v16
19 STORE  yo ← v17
```

SPILL! @v2

RESTORE! @v2

Registers

r1 v16
r2 **v2**
r3
r4
r5
r6

Next Use

v1	
v2	18
v3	
v4	
v5	
v6	
v7	
v8	
v9	
v10	
v11	
v12	
v13	
v14	
v15	
v16	18
v17	

Our Unscheduled Example

```
 1 LOAD   r1 ← x
 2 LOAD   r2 ← y
 3 LOAD   r3 ← nx
 4 LOAD   r4 ← ny
 5 FMUL   r5 ← r1 * r3
 6 FMUL   r6 ← r2 * r4
 7 FADD   r5 ← r5 + r6
 8 FMUL   r6 ← r3 * r3
 9 FMUL   r2 ← r4 * r4
10 FADD   r2 ← r6 + r2
11 FDIV   r2 ← 1.0 / r2      SPILL! @v2
12 FMUL   r5 ← 2.0 * r5
13 FMUL   r2 ← r5 * r2
14 FMUL   r3 ← r3 * r2
15 FSUB   r1 ← r1 - r3       RESTORE! @v2
16 STORE  xo ← r1
17 FMUL   r1 ← r4 * r2
18 FSUB   r1 ← r2 - r1
19 STORE  yo ← r1
```

Registers

r1 v17
r2
r3
r4
r5
r6

Next Use

v1	
v2	
v3	
v4	
v5	
v6	
v7	
v8	
v9	
v10	
v11	
v12	
v13	
v14	
v15	
v16	
v17	19

Sheldon Best's Algorithm

- Invented in 1955 (Best) for the first Fortran compiler
  - ✦ Reinvented in 1965 (Laslo Belady) for analyzing paging algorithms in OSes
  - ✦ Reinvented in 1975 (William Harrison) — ECS compiler
  - ✦ Reinvented in 1989 (Chris Fraser) — LCC compiler
  - ✦ Reinvented in 1997 (Vincenzo Liberatore) — Rutgers
- Key point — applicable for any linearized resource allocation problem
- The algorithm really is "Best" — it's optimal (locally), but can only be realized statically (can't see the future)

Complications — Spill Costs

- Dirty vs. Clean vs. Recomputable Values
  - ✦ Very different costs to spill
  - ✦ doing a bunch of simple, recomputable spills instead of spilling a dirty value that won't be needed for a while can be faster!
- Again, we can solve nicely posed problems optimally, without actually achieving optimal code

Outline

Graph Coloring

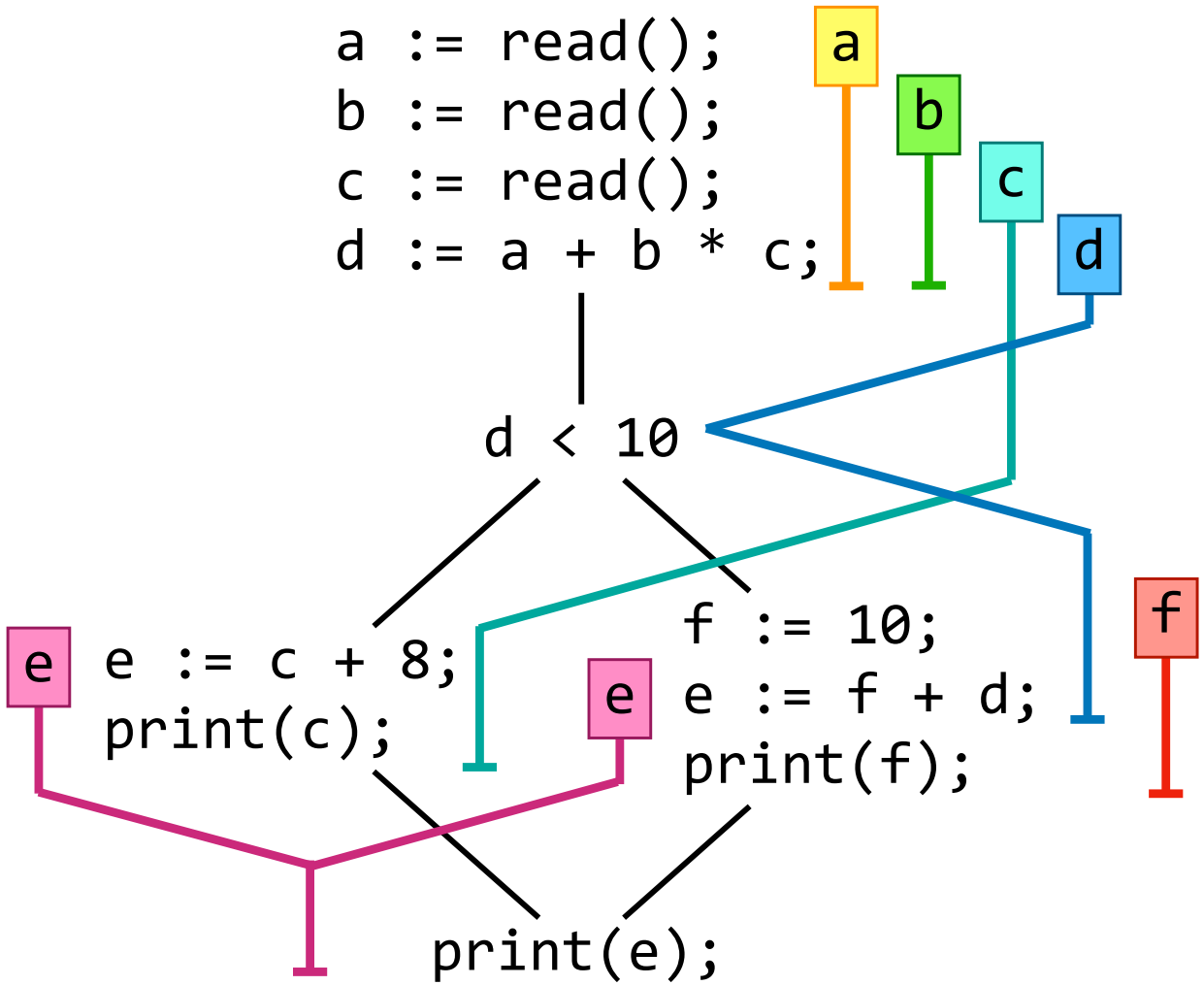
Global Register Allocation

- Not all code is “straight-line” basic blocks
  - ✦ register lifetimes escape the bounds of basic blocks
- Want to **globally** (i.e. in a procedure) allocate registers
- Graph Coloring — We are going to show that allocating registers is equivalent to coloring a graph
  - ✦ i.e. assign a color (register) to each node such that if two nodes share an edge, they must have different colors
  - ✦ In general, graph coloring is NP-Complete!
- But, which graph?

Step 1 — Run Liveness on CFG

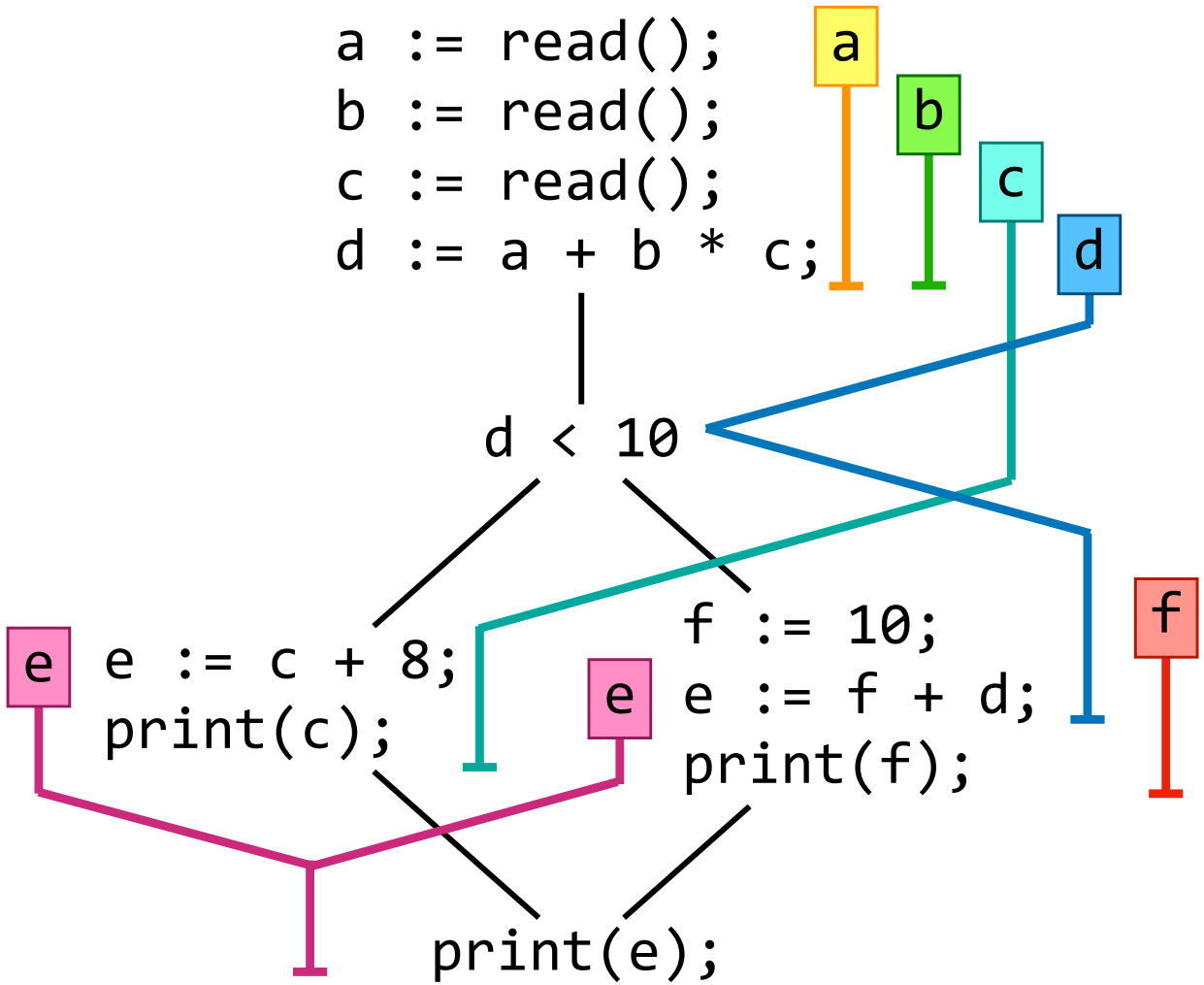
- Determine where/when each variable is live
- If two variables are live at the same time, then they can't be stored in the same register
  - ✦ we say the two variables ***interfere*** with each other
- From this, we can construct an undirected graph
  - ✦ Each node is a variable
  - ✦ Each edge is an interference
- Note — *we want our input CFG to use variables in a way as close to SSA as possible* (to prevent conflating interferences between what should be different variables)

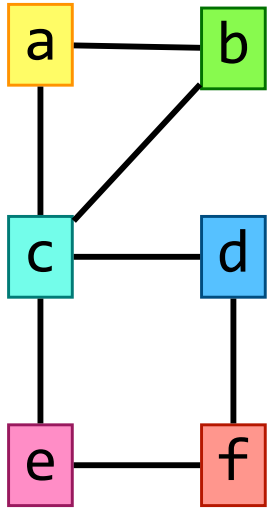
Live Variable Analysis



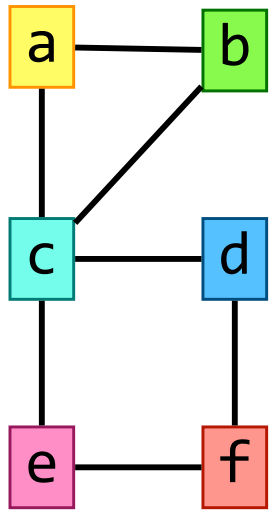

```
a := read();
b := read();
c := read();
d := a + b * c;
if (d < 10) then
    e := c + 8;
    print(c);
else
    f := 10;
    e := f + d;
    print(f);
fi
print(e);
```

Variable Interference Graph



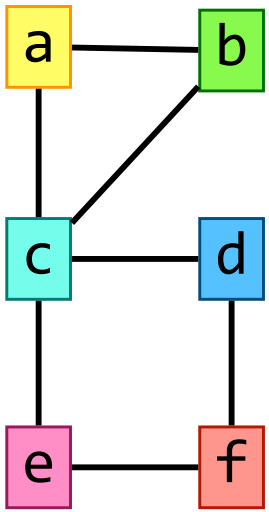


Step 2 — Graph Coloring



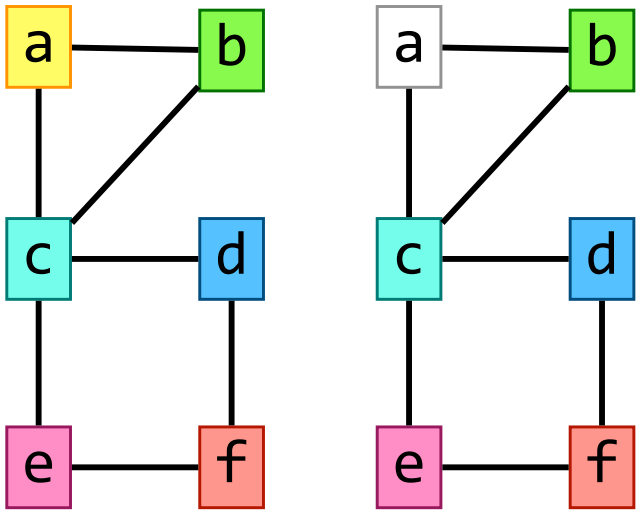
- NP Complete Problem in general
- Heuristic — ***color easy nodes last*** — So…
  - find node **n** with the lowest ***degree***
  - remove **n** from the graph (and record)
  - now color the simplified graph…
  - …we're back, and set the color of **n** to the first color that's not used by any of **n**'s neighbors (does some such color have to exist…?)
- Basic ideas due to Chaitin (1982), refined by Briggs (1992)

Simplifying…

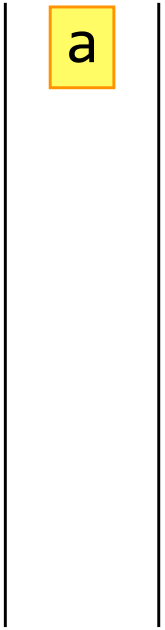


Stack

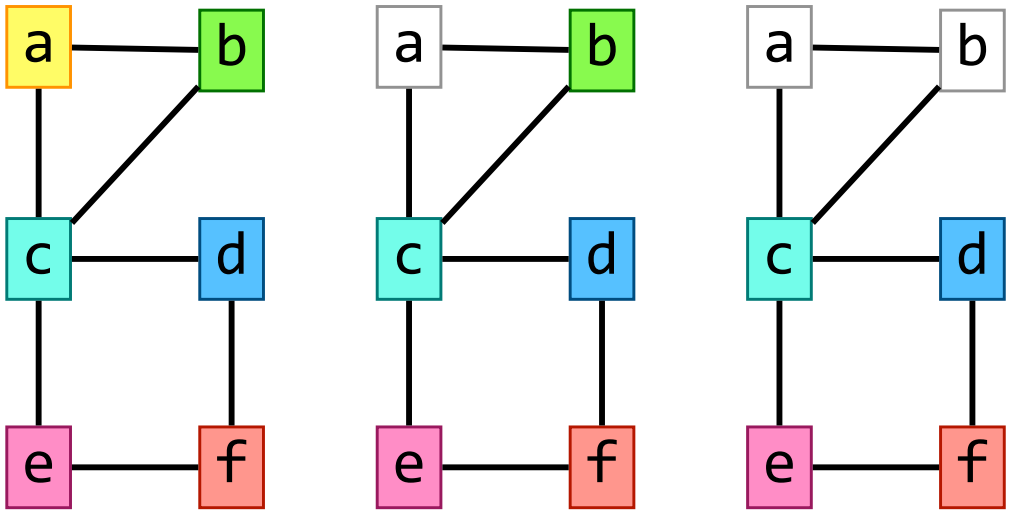
Simplifying…



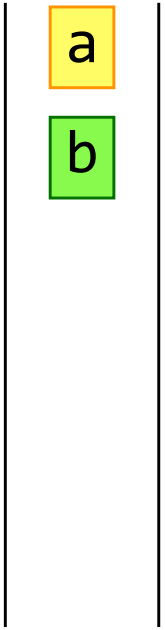
Stack



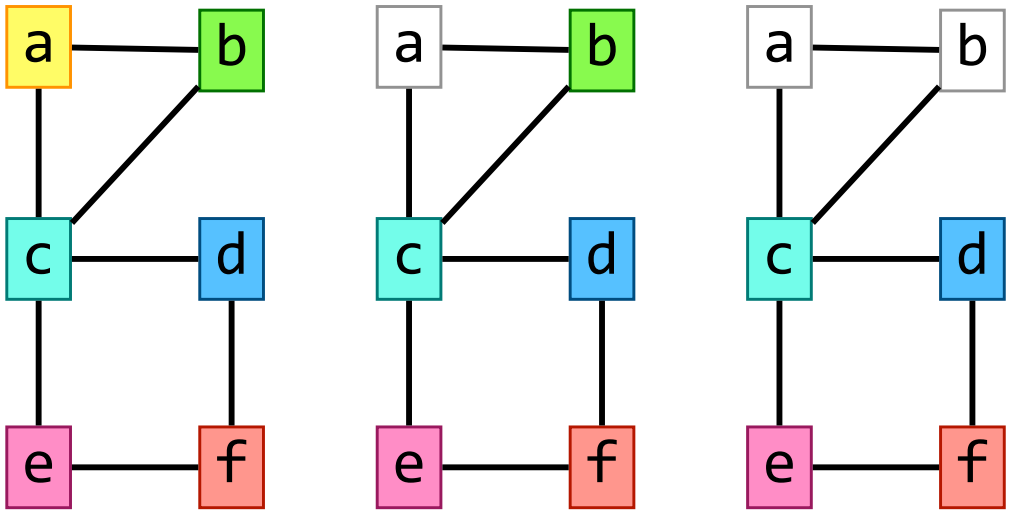
Simplifying…

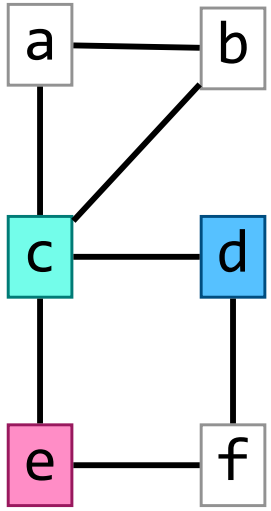


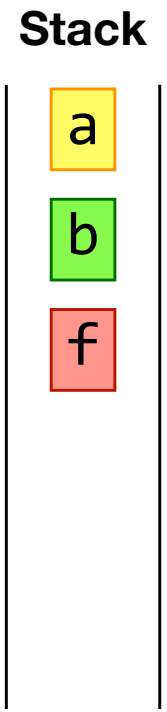
Stack



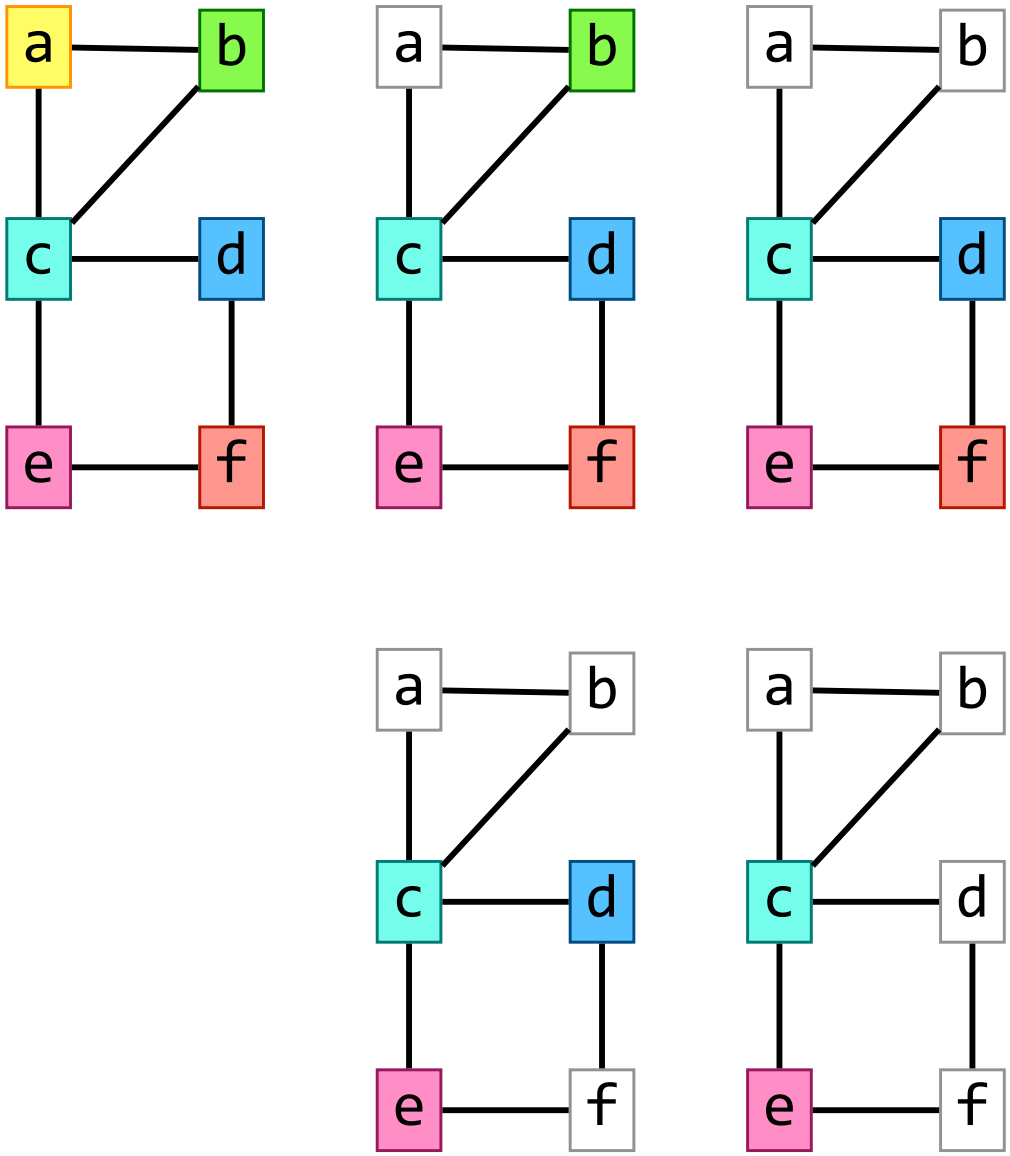
Simplifying…



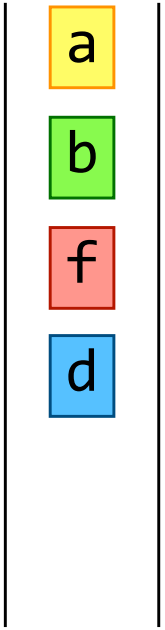




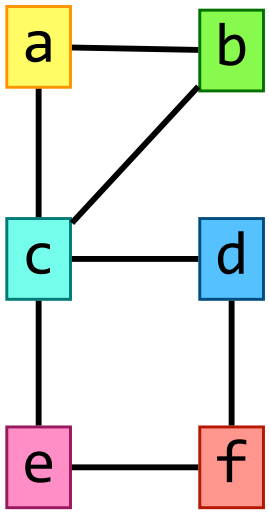
Simplifying…

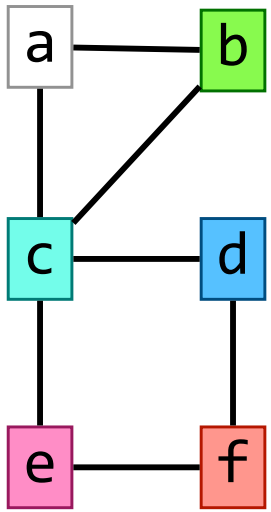


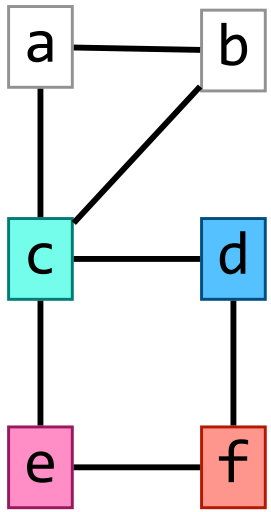
Stack

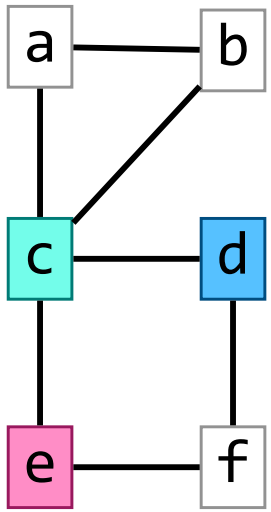


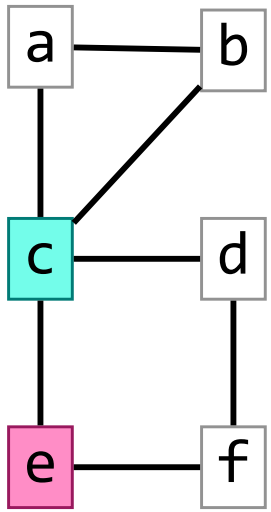
Simplifying…

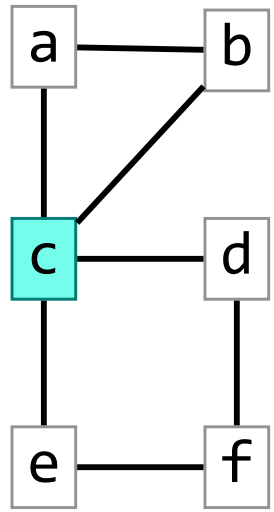


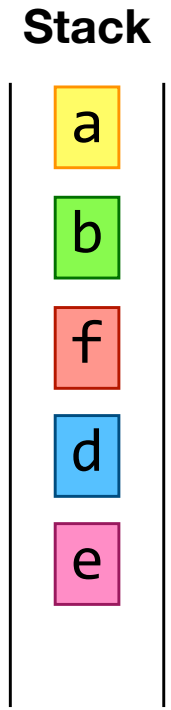




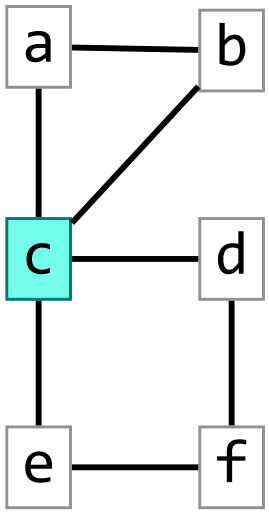




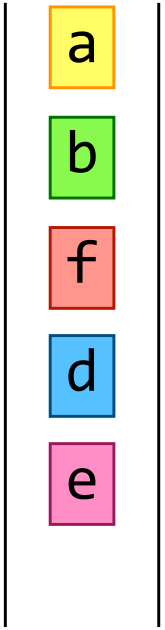




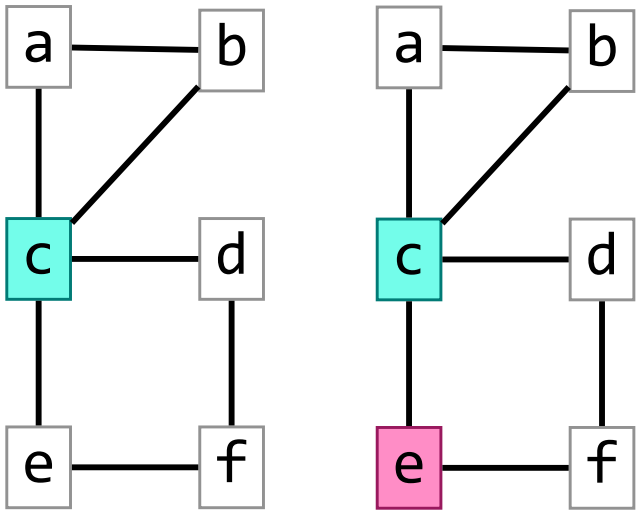
Coloring…



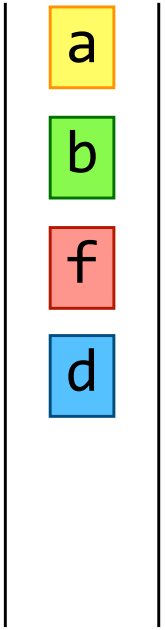
Stack



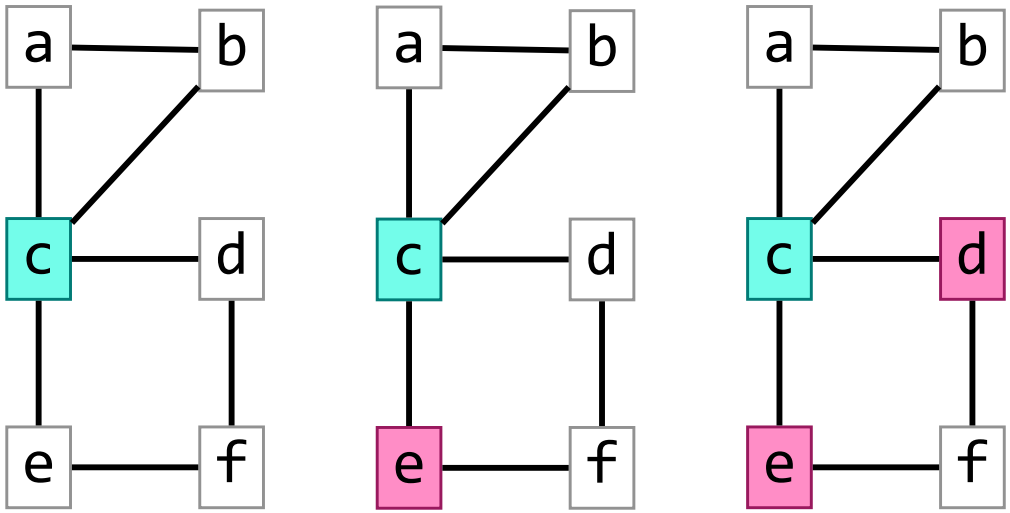
Coloring…



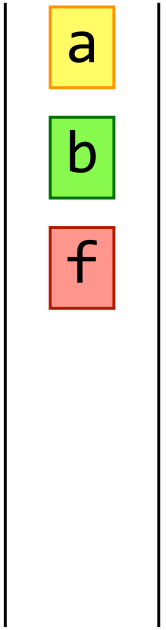
Stack



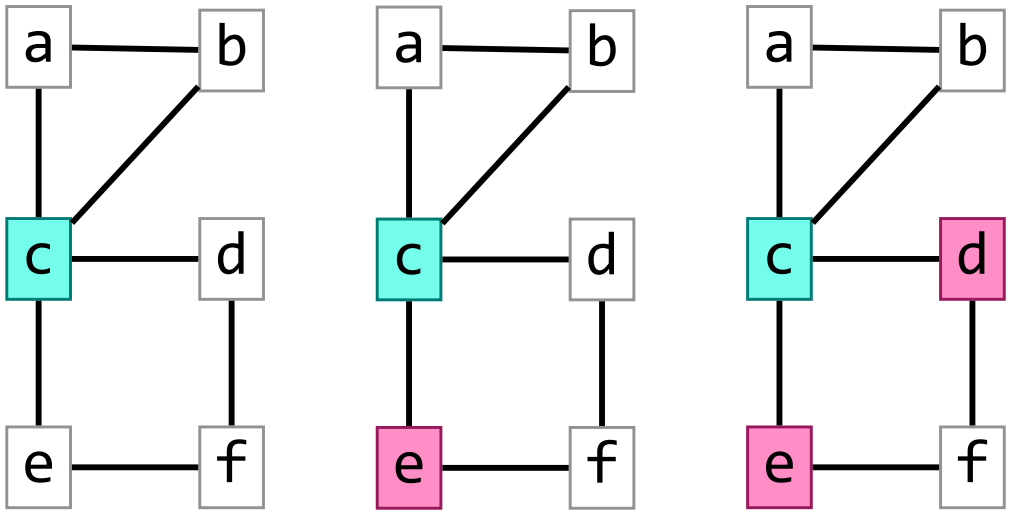
Coloring…

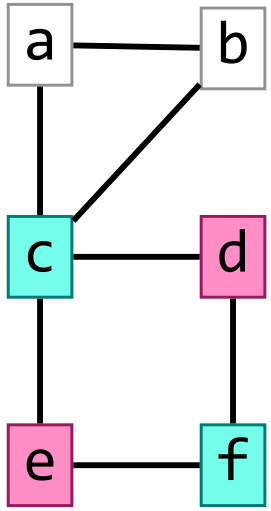


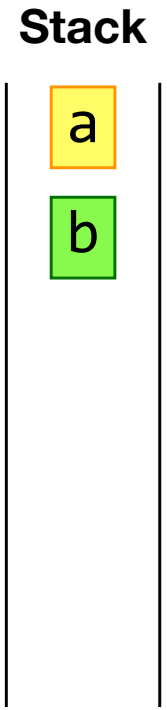
Stack



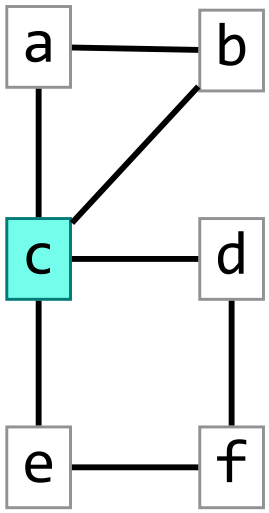
Coloring…

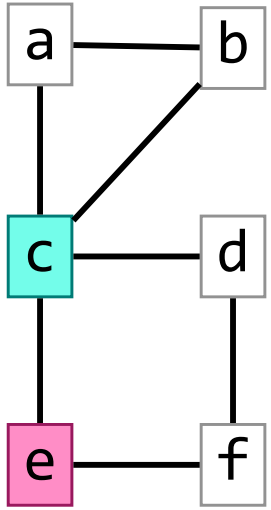


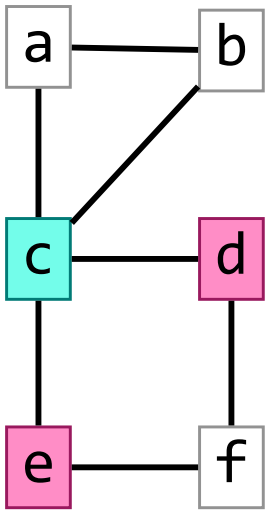


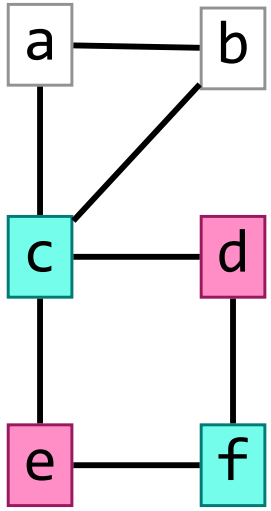


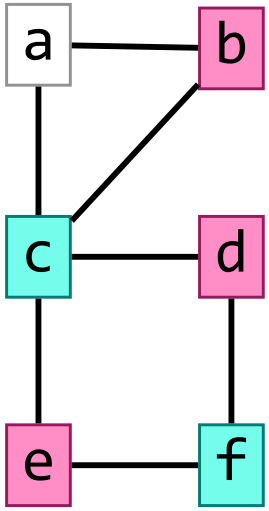
Coloring…

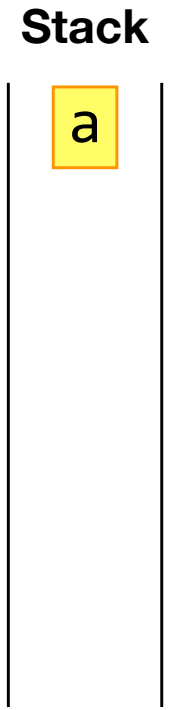




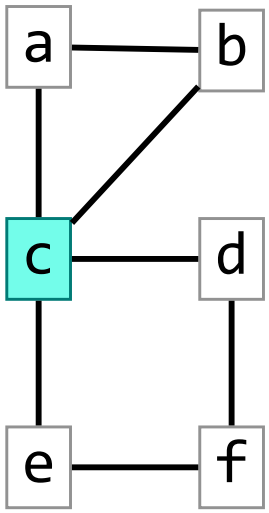


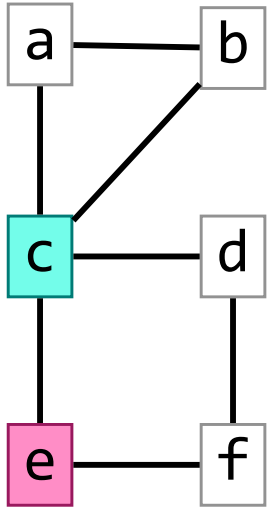


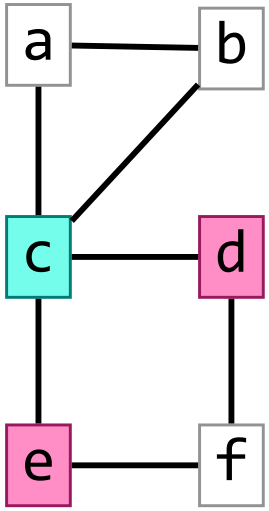


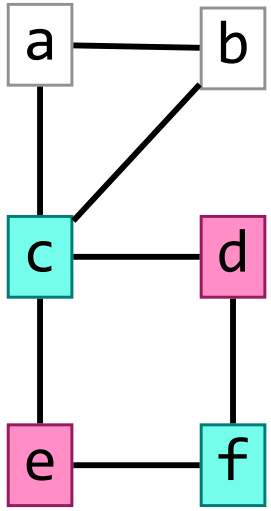


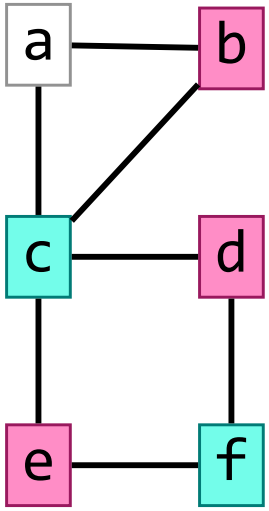
Coloring…

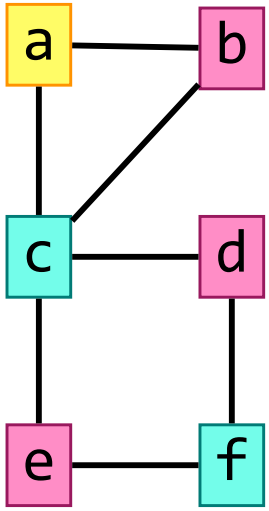




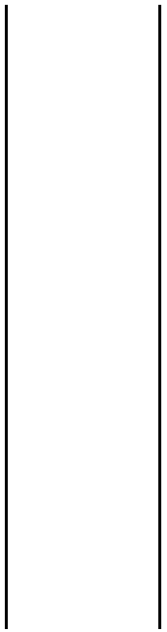




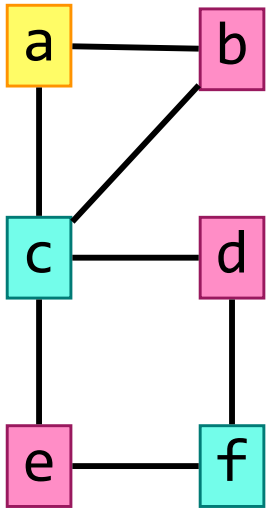




Stack



The Final Assignment is…




```
a := read();
b := read();
c := read();
d := a + b * c;
if (d < 10) then
     e := c + 8;
     print(c);
else
     f := 10;
     e := f + d;
     print(f);
fi
print(e);
```

What if We Run Out of Colors?

- It's possible!
- If it happens, then we insert spill code for one or more of the variables we get stuck at
  - Note — there must be at least K+1 variables with degree K among themselves for this to happen!
- Also, need to make sure to handle special/dedicated registers
  - e.g. the color of function return and argument registers is pre-determined by the calling convention!
  - (we can pre-color some nodes to achieve this goal)

Coalescing Live Ranges

- Idea: if two live ranges are connected by a copy operation (MOV ri → rj) but do not otherwise interfere, then the live ranges can be coalesced (combined)
  - Rewrite all references to rj to use ri
  - Remove the copy instruction
- Then need to fix up interference graph

Advantages?

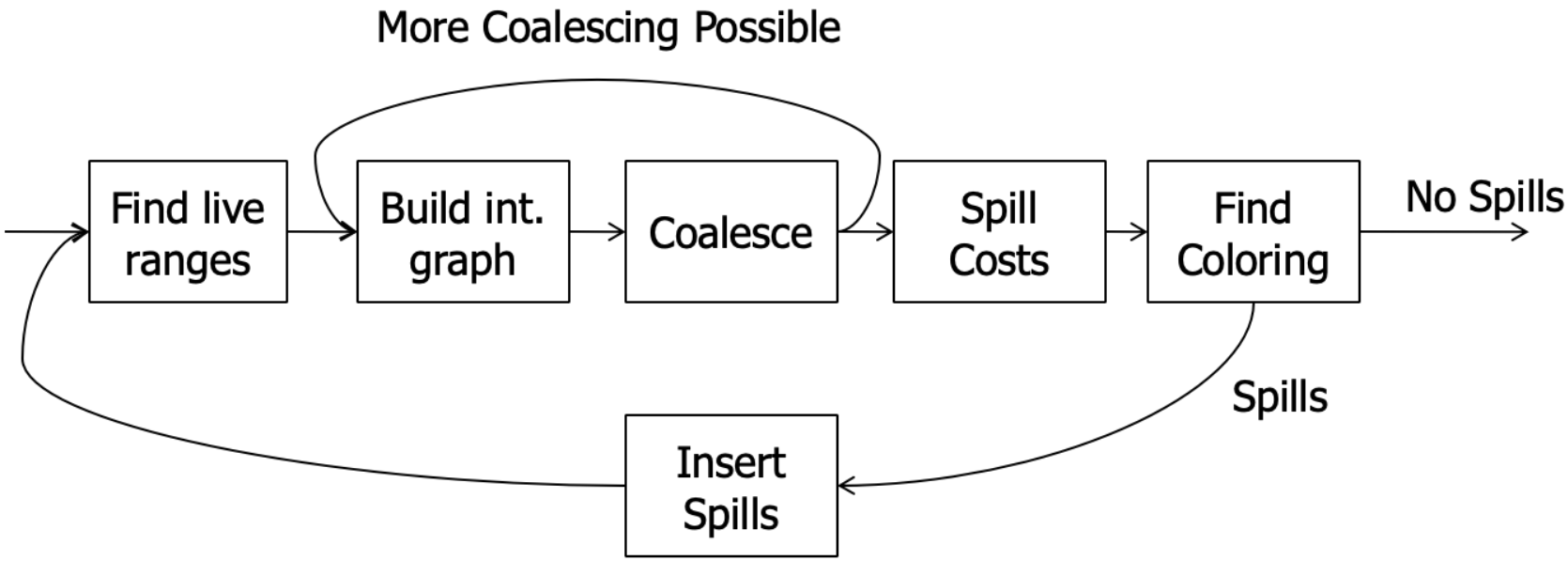
- Makes the code smaller, faster (no copy operation)
- Shrinks set of live ranges
- Reduces the degree of any live range that interfered with both live ranges ri, rj
- But: coalescing two live ranges can prevent coalescing of others, so ordering matters
  - Best: Coalesce most frequently executed ranges first (e.g., inner loops)
- Can have a substantial payoff – do it!

Graph Representation

- The interference graph representation drives the time and space requirements for the allocator (and maybe the compiler)
- Not unknown to have O(5K) nodes and O(1M) edges
- Dual representation works best
  - Triangular bit matrix for efficient access to interference information
  - Vector of adjacency vectors for efficient access to node neighbors

If at first you don't succeed…

- Iterate through stages of register allocation until you get an allocation
- You may also want to interleave instruction selection & scheduling to resolve various issues



End of the Quarter! …?

We Built a Compiler

- Really, look back at what you did!
  - ✦ Scanner, Parser, Checker, Codegen
  - ✦ (and we covered the backend in lecture)
- Give yourself a big pat on the back!

It takes a team…

- None of this would be possible without a lot of other people
- TAs — Eric Chen, Karen Haining, Sriya Bulusu, Bill Baxter
- Much Help & Advice from Hal Perkins
  - (He's giving the Garbage Collection Lecture on Friday)

Thanks to YOU!

- You should be proud of what you've accomplished in a 10 week quarter! Take care of yourself, your friends and your community.
- Congratulations!
- Please feel free to drop by my office in the future or send me a message and let me know what you've been up to after compilers!