EECS 583 – Class 14
Finish Modulo Scheduling
Register Allocation

University of Michigan

February 22, 2023

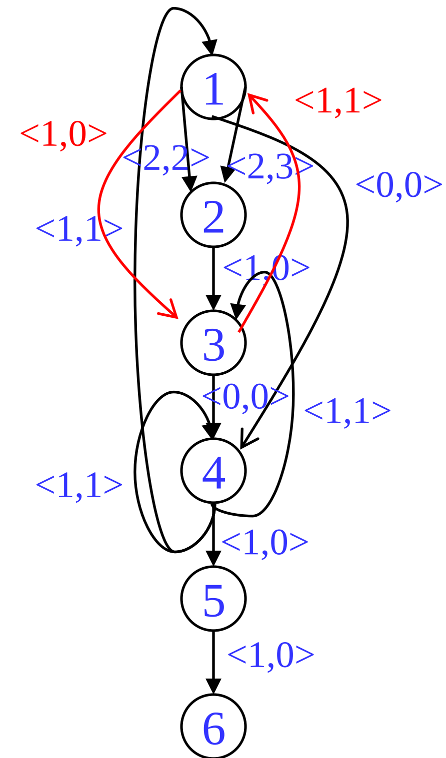
Announcements

- Project proposal meeting signups – this week
  - 10 minute Zoom meeting with Aditya and I
  - Planned for Mar 6, Mar 8, Mar 9: 10am-noon
- Midterm Exam – Wed Mar 15
  - Exam review – Mon Mar 13
  - Exam scope: Covers all lecture material through today's class
  - Exam format: Hybrid (Virtual or in-person)
    - In-person: 10:30-11:50, walk outside to get questions answered
      - Send Aditya and I email if you plan to take the exam in-person
    - Virtual: 10:30-11:50 + 15 mins extra time (Extra time for printing, scanning, uploading), post private questions on piazza to get answers
    - Piazza questions answered up to 12:00
- Today's class reading
  - "Register Allocation and Spilling Via Graph Coloring," G. Chaitin, Proc. 1982 SIGPLAN Symposium on Compiler Construction, 1982.

Homework Problem From Last Class

Latencies: ld = 2, st = 1, add = 1, cmpp = 1, br = 1
Resources: 1 ALU, 1 MEM, 1 BR

```
1: r1[-1] = load(r2[0])
2: r3[-1] = r1[1] – r1[2]
3: store (r3[-1], r2[0])
4: r2[-1] = r2[0] + 4
5: p1[-1] = cmpp (r2[-1] < 100)
remap r1, r2, r3
6: brct p1[-1] Loop
```



Calculate RecMII, ResMII, and MII

ResMII: ALU: 3 instrs / 1 unit = 3
MEM: 2 instrs / 1 unit = 2
BR: 1 instr / 1 unit = 1
MAX(3,2,1) = 3

RecMII: 4 → 4: 1/1 = 1
3 → 4 → 3: (0 + 1) / (0 +1) = 1
1 → 3 → 1: (1 + 1) / (0 + 1) = 2
1 → 2 → 3 → 1: (2+1+1) / (2+0+1) = 2
1 → 2 → 3 → 1: (2+1+1) / (3+0+1) = 1
MAX(1,1,2,2,1) = 2

MII = MAX(ResMII, RecMII) = MAX(3,2) = 3

From Last Class: Example – Step 11

Step11: calculate ESC, SC = ceiling(max unrolled sched length / ii)
unrolled sched time of branch = rolled sched time of br + (ii*esc)

SC = 6 / 2 = 3, ESC = SC – 1
time of br = 1 + 2*2 = 5

LC = 99

Loop:
```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
remap r1, r2, r3, r4
7: brlc Loop
```

Rolled Schedule

0	1 2 4
1	7 3 5

Unrolled Schedule

0	1 4
1	
2	2
3	
4	
5	3 5 7
6	

MRT

	alu0	alu1	mem	br
0	X	X	X	
1	X		X	X

From Last Class: Example – Step 12

Finishing touches - Sort ops, initialize ESC, insert BRF and staging predicate, initialize staging predicate outside loop

LC = 99
ESC = 2
p1[0] = 1

Staging predicate, each successive stage increment the index of the staging predicate by 1, stage 1 gets px[0]

```
Loop:  1: r3[-1] = load(r1[0]) if p1[0]
       2: r4[-1] = r3[-1] * 26 if p1[1]
       4: r1[-1] = r1[0] + 4 if p1[0]
       3: store (r2[0], r4[-1]) if p1[2]
       5: r2[-1] = r2[0] + 4 if p1[2]
       7: brlc Loop if p1[2]
```

Unrolled Schedule

0	1 4	Stage 1
1		
2	2	Stage 2
3		
4		Stage 3
5	3 5 7	
6		

From Last Class: Example – Dynamic Execution of the Code

LC = 99
ESC = 2
p1[0] = 1

Loop: 1: r3[-1] = load(r1[0]) if p1[0]
2: r4[-1] = r3[-1] * 26 if p1[1]
4: r1[-1] = r1[0] + 4 if p1[0]
3: store (r2[0], r4[-1]) if p1[2]
5: r2[-1] = r2[0] + 4 if p1[2]
7: brlc Loop if p1[2]

Total time = II(num_iteration + num_stages – 1)
= 2(100 + 3 – 1) = 204 cycles

time: ops executed

0: 1, 4
1:

2: 1,2,4
3:

4: 1,2,4
5: 3,5,7

6: 1,2,4
7: 3,5,7

…

198: 1,2,4
199: 3,5,7

200: 2
201: 3,5,7

202: -
203 3,5,7

Class Problem

latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

```
for (j=0; j<100; j++)
   b[j] = a[j] * 26
```

```
      LC = 99

Loop: 1: r3 = load(r1)
      2: r4 = r3 * 26
      3: store (r2, r4)
      4: r1 = r1 + 4
      5: r2 = r2 + 4
      7: brlc Loop
```

How many resources of each type are required to achieve an II=1 schedule?

If the resources are non-pipelined, how many resources of each type are required to achieve II=1

Assuming pipelined resources, generate the II=1 modulo schedule.

Class Problem – Answers in Red

latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

for (j=0; j<100; j++)
  b[j] = a[j] * 26

LC = 99

```
Loop:  1: r3 = load(r1)
       2: r4 = r3 * 26
       3: store (r2, r4)
       4: r1 = r1 + 4
       5: r2 = r2 + 4
       7: brlc Loop
```

How many resources of each type are required to achieve an II=1 schedule?
For II=1, each operation needs a dedicated resource, so: 3 ALU, 2 MEM, 1 BR

If the resources are non-pipelined, how many resources of each type are required to achieve II=1
Instead of 1 ALU to do the multiplies, 3 are needed, and instead of 1 MEM to do the loads, 2 are needed. Hence: 5 ALU, 3 MEM, 1 BR

Assuming pipelined resources, generate the II=1 modulo schedule.
See next few slides

Problem continued

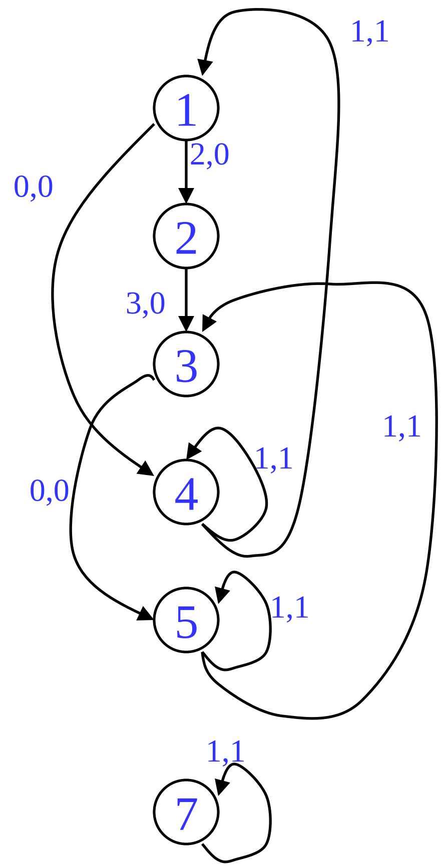
Assume II=1 so resources are: 3 ALU, 2 MEM, 1 BR

DSA converted code below (same as example in class)

LC = 99

Loop:
```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
remap r1, r2, r3, r4
7: brlc Loop
```

Dependence graph



RecMII = 1
RESMII = 1
MII = MAX(1,1) = 1

Priorities
1: H = 5
2: H = 3
3: H = 0
4: H = 4
5: H = 0
7: H = 0

Problem continued

resources: 3 alu, 2 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

LC = 99

Loop:

```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
remap r1, r2, r3, r4
7: brlc Loop
```

Rolled Schedule

0	7 1 4 2 3 5

Unrolled Schedule

0	1 4	stage 1
1		stage 2
2	2	stage 3
3		stage 4
4		stage 5
5	3 5 7	stage 6
6		

MRT

	alu0	alu1	alu2	m1	m2	br
0	X	X	X	X	X	X

Scheduling steps:

- Schedule brlc at time II-1
- Schedule op1 at time 0
- Schedule op4 at time 0
- Schedule op2 at time 2
- Schedule op3 at time 5
- Schedule op5 at time 5
- Schedule op7 at time 5

Problem continued

The final loop consists of a single MultiOp containing 6 operations, each predicated on the appropriate staging predicate. Note register allocation still needs to be performed.

```
    LC = 99
Loop:
    r3[-1] = load(r1[0]) if p1[0]; r4[-1] = r3[-1] * 26 if p1[2]; store (r2[0], r4[-1]) if p1[5]; r1[-1] = r1[0] + 4 if p1[0]; r2[-1] = r2[0] + 4 if p1[5]; brlc Loop
```

What if We Don't Have Hardware Support for Modulo Scheduling?

- No predicates
  - Predicates enable kernel-only code by selectively enabling/disabling operations to create prolog/epilog
  - Now must create explicit prolog/epilog code segments
- No rotating registers
  - Register names not automatically changed each iteration
  - Must unroll the body of the software pipeline, explicitly rename
    - Consider each register lifetime i in the loop
    - Kmin = min unroll factor = MAXi (ceiling((Endi – Starti) / II))
    - Create Kmin static names to handle maximum register lifetime
  - Apply modulo variable expansion

Register Allocation

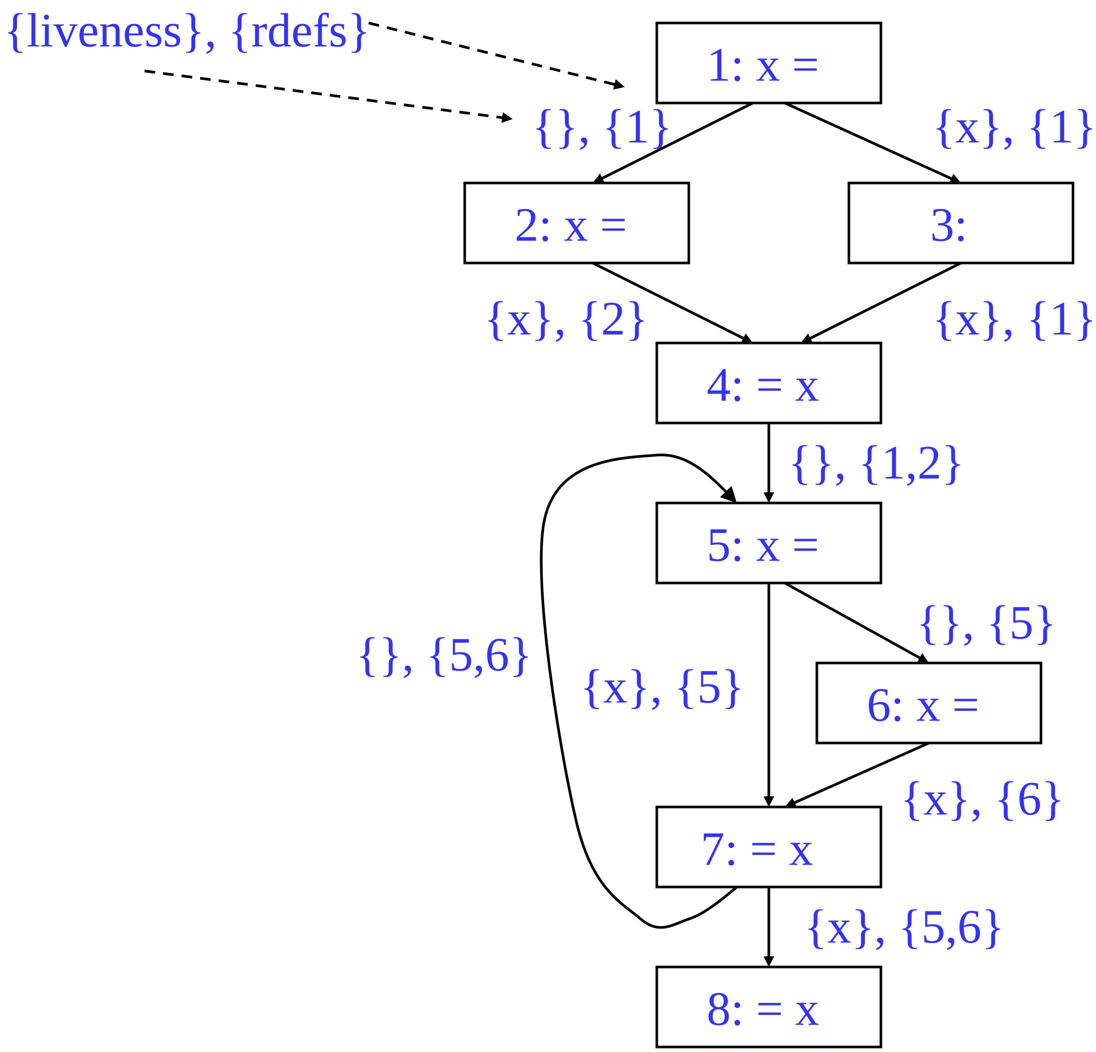
Register Allocation: Problem Definition

- Through optimization, assume an infinite number of virtual registers
  - Now, must allocate these infinite virtual registers to a limited supply of hardware registers
  - Want most frequently accessed variables in registers
    - Speed, registers much faster than memory
    - Direct access as an operand
  - Any VR that cannot be mapped into a physical register is said to be spilled
- Questions to answer
  - What is the minimum number of registers needed to avoid spilling?
  - Given n registers, is spilling necessary
  - Find an assignment of virtual registers to physical registers
  - If there are not enough physical registers, which virtual registers get spilled?

Live Range

- Value = definition of a register
- Live range = Set of operations
  - 1 more or values connected by common uses
  - A single VR may have several live ranges
- Live ranges are constructed by taking the intersection of reaching defs and liveness
  - Initially, a live range consists of a single definition and all ops in a function in which that definition is live

Example – Constructing Live Ranges

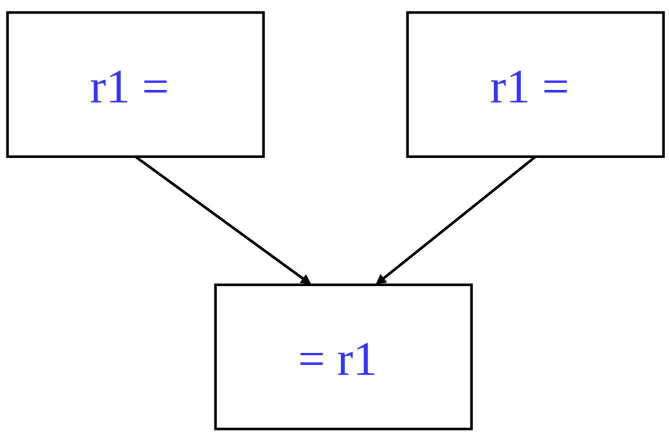


Each definition is the seed of a live range. Ops are added to the LR where <u>both the defn reaches and the variable is live</u>

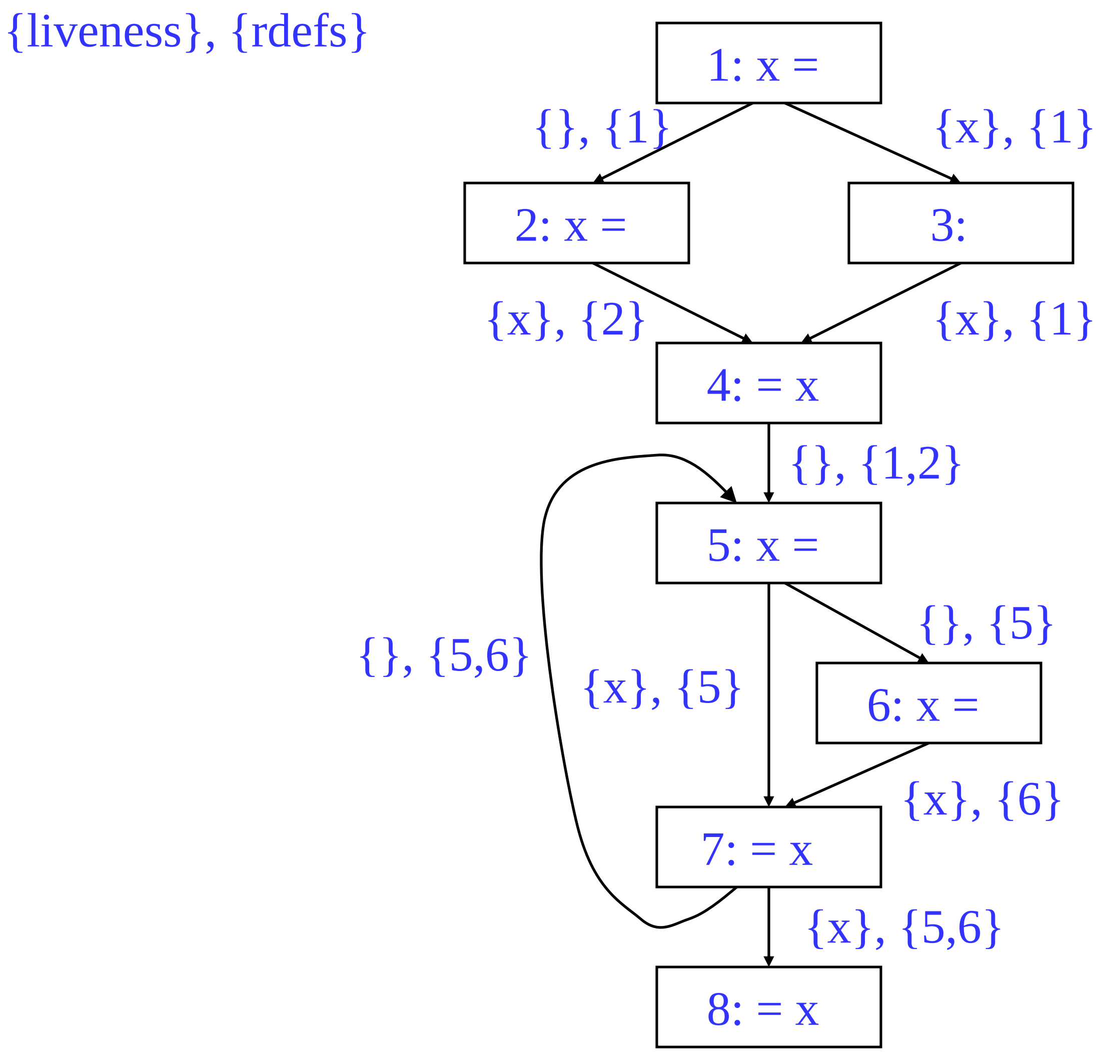
LR1 for def 1 = {1,3,4}
LR2 for def 2 = {2,4}
LR3 for def 5 = {5,7,8}
LR4 for def 6 = {6,7,8}

Merging Live Ranges

- If 2 live ranges for the same VR overlap, they must be merged to ensure correctness
  - » LRs replaced by a new LR that is the union of the LRs
  - » Multiple defs reaching a common use
  - » Conservatively, all LRs for the same VR could be merged
    - Makes LRs larger than need be, but done for simplicity
    - We will not assume this



Example – Merging Live Ranges



LR1 for def 1 = {1,3,4}
LR2 for def 2 = {2,4}
LR3 for def 5 = {5,7,8}
LR4 for def 6 = {6,7,8}

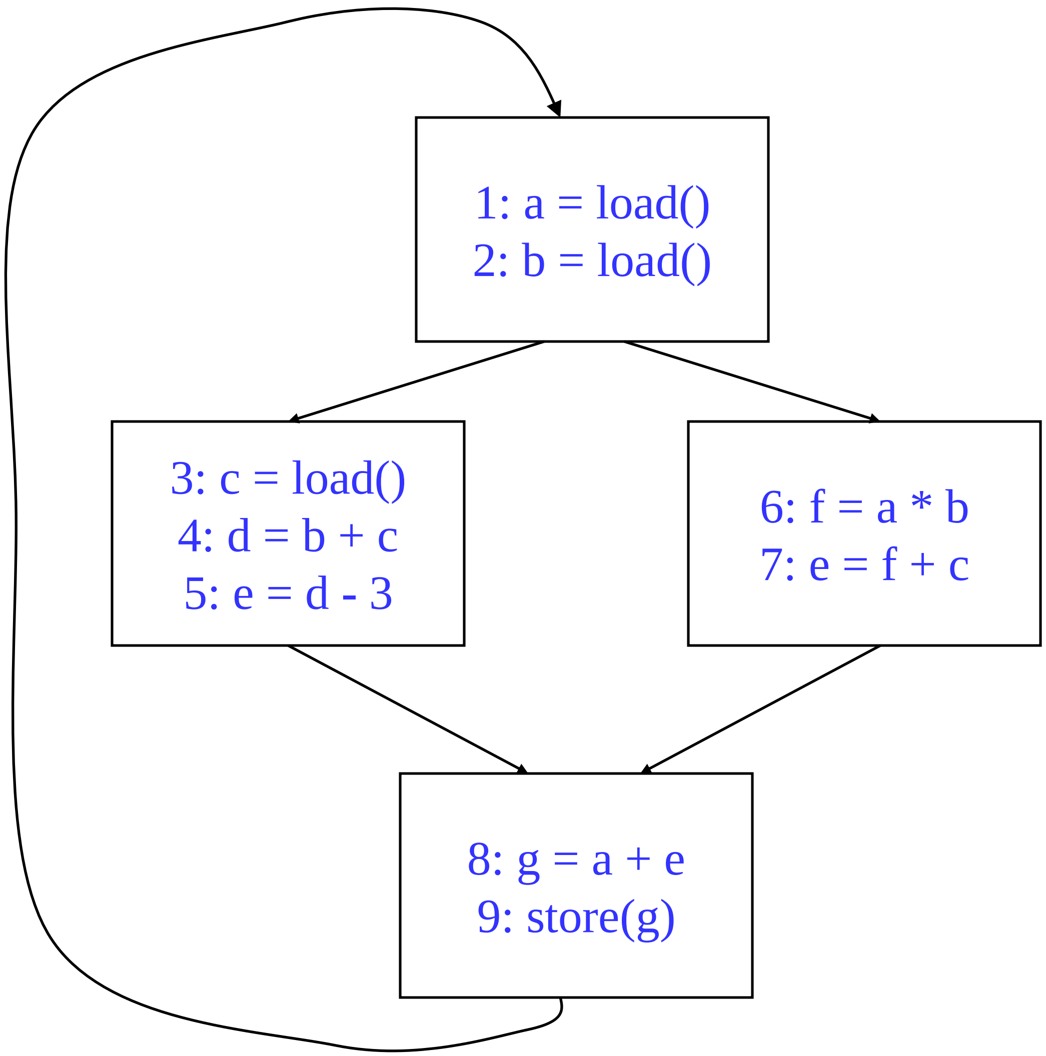
Merge LR1 and LR2, LR3 and LR4

LR5 = {1,2,3,4}
LR6 = {5,6,7,8}

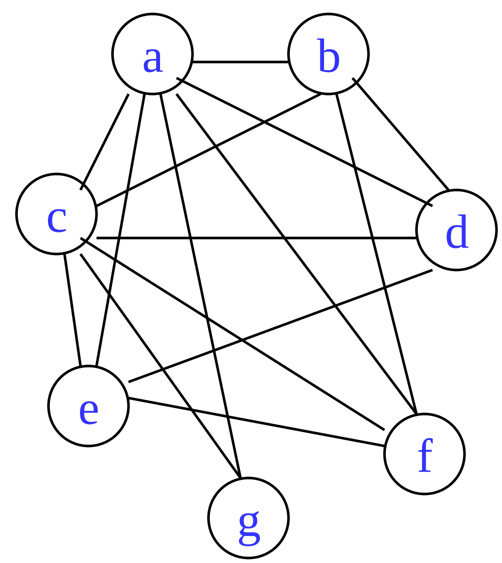
Interference

- Two live ranges interfere if they share one or more ops in common
  - Thus, they cannot occupy the same physical register
  - Or a live value would be lost
- Interference graph
  - Undirected graph where
    - Nodes are live ranges
    - There is an edge between 2 nodes if the live ranges interfere
  - What’s not represented by this graph
    - Extent of interference between the LRs
    - Where in the program is the interference

Example – Interference Graph



lr(a) = {1,2,3,4,5,6,7,8}
lr(b) = {2,3,4,6}
lr(c) = {1,2,3,4,5,6,7,8,9}
lr(d) = {4,5}
lr(e) = {5,7,8}
lr(f) = {6,7}
lr{g} = {8,9}



Graph Coloring

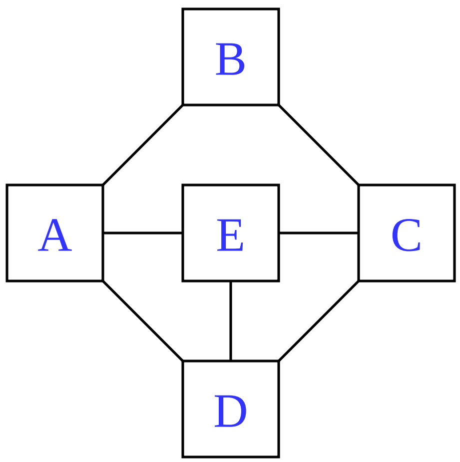
- A graph is n-colorable if every node in the graph can be colored with one of the n colors such that 2 adjacent nodes do not have the same color
  - Model register allocation as graph coloring
  - Use the fewest colors (physical registers)
  - Spilling is necessary if the graph is not n-colorable where n is the number of physical registers
- Optimal graph coloring is NP-complete for n > 2
  - Use heuristics proposed by compiler developers
    - "Register Allocation Via Coloring", G. Chaitin et al, 1981
    - "Improvement to Graph Coloring Register Allocation", P. Briggs et al, 1989
  - **Observation** – a node with degree < n in the interference can always be successfully colored given its neighbors colors

Coloring Algorithm

- 1. While any node, x, has < n neighbors
  - » Remove x and its edges from the graph
  - » Push x onto a stack
- 2. If the remaining graph is non-empty
  - » Compute cost of spilling each node (live range)
    - For each reference to the register in the live range
      - Cost += (execution frequency * spill cost)
  - » Let NB(x) = number of neighbors of x
  - » Remove node x that has the smallest cost(x) / NB(x)
    - Push x onto a stack (mark as spilled)
  - » Go back to step 1
- While stack is non-empty
  - » Pop x from the stack
  - » If x's neighbors are assigned fewer than R colors, then assign x any unsigned color, else leave x uncolored

Example – Finding Number of Needed Colors

How many colors are needed to color this graph?



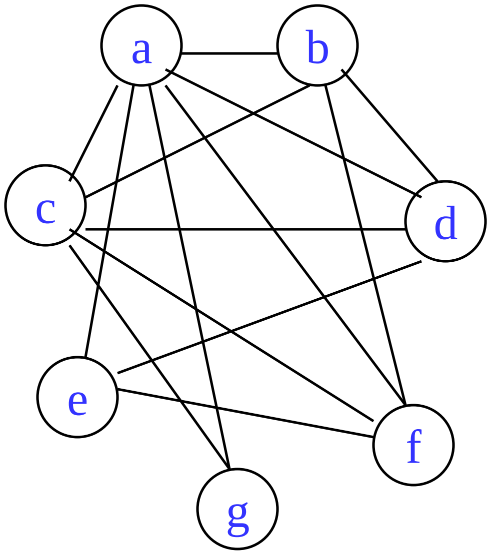
Try n=1, no, cannot remove any nodes

Try n=2, no again, cannot remove any nodes

Try n=3,

Remove B
Then can remove A, C
Then can remove D, E
Thus it is 3-colorable

Example – Do a 3-Coloring



lr(a) = {1,2,3,4,5,6,7,8}
refs(a) = {1,6,8}
lr(b) = {2,3,4,6}
refs(b) = {2,4,6}
lr(c) = {1,2,3,4,5,6,7,8,9}
refs(c) = {3,4,7}
lr(d) = {4,5}
refs(d) = {4,5}
lr(e) = {5,7,8}
refs(e) = {5,7,8}
lr(f) = {6,7}
refs(f) = {6,7}
lr{g} = {8,9}
refs(g) = {8,9}

Profile freqs
1,2 = 100
3,4,5 = 75
6,7 = 25
8,9 = 100

Assume each spill requires 1 operation

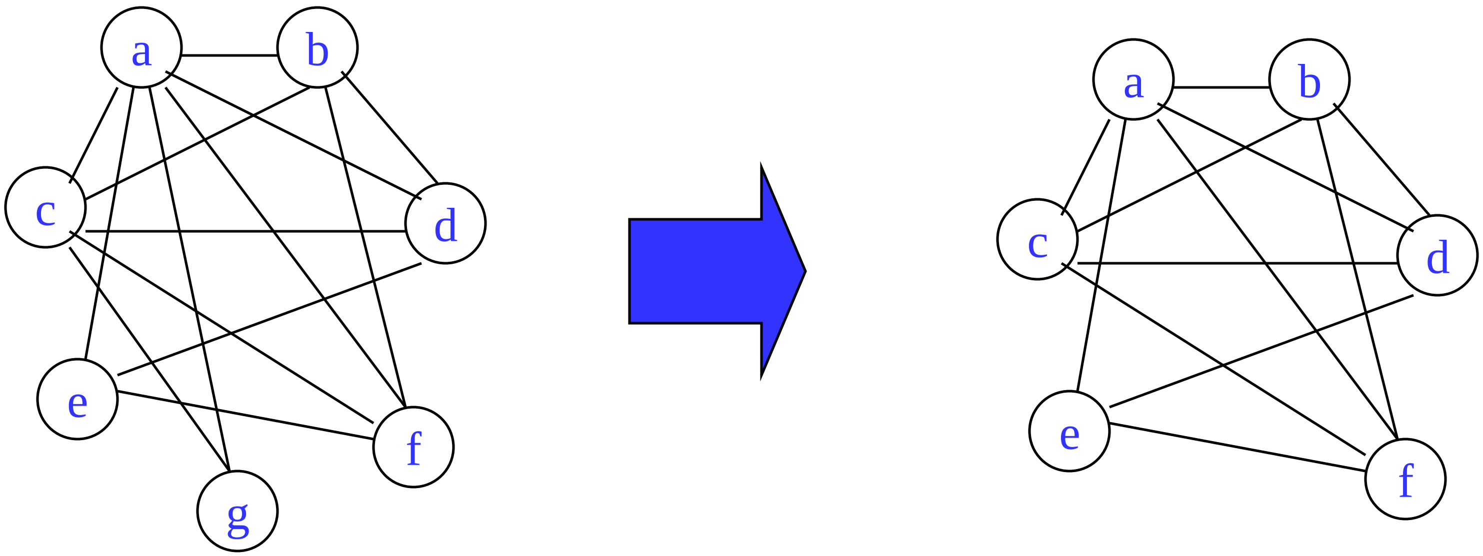
	a	b	c	d	e	f	g
cost	225	200	175	150	200	50	200
neighbors	6	4	5	4	3	4	2
cost/n	37.5	50	35	37.5	66.7	12.5	100

Example – Do a 3-Coloring (2)

Remove all nodes < 3 neighbors

So, g can be removed

Stack
g

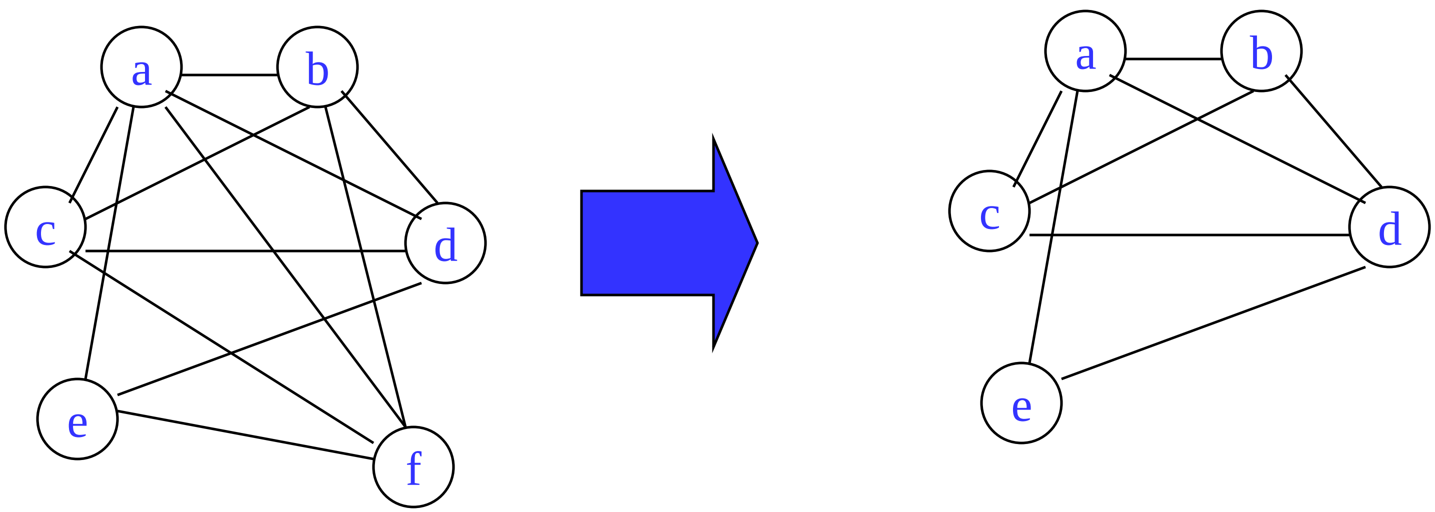


Example – Do a 3-Coloring (3)

Now must spill a node

Choose one with the smallest cost/NB → f is chosen

Stack
f (spilled)
g

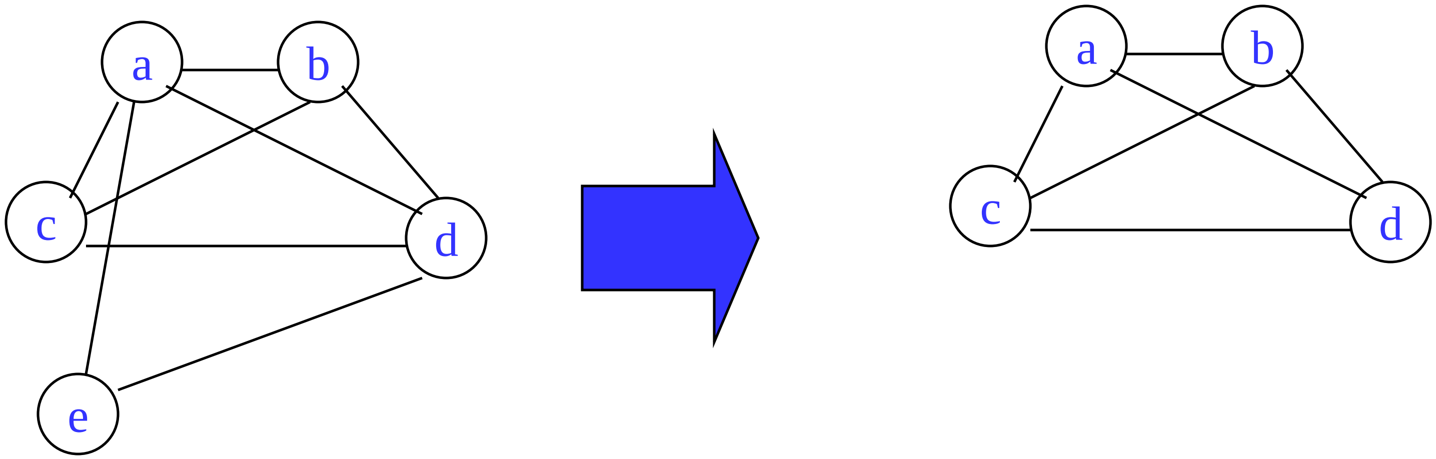


Example – Do a 3-Coloring (4)

Remove all nodes < 3 neighbors

So, e can be removed

<u>Stack</u>
e
f (spilled)
g

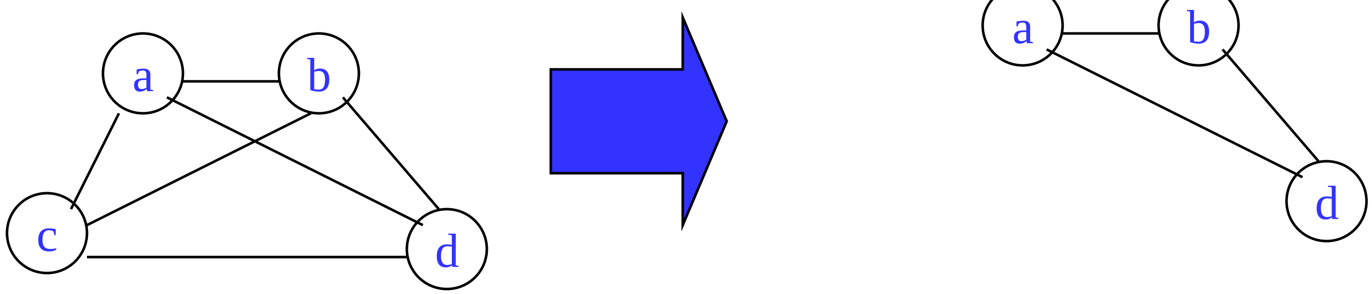


Example – Do a 3-Coloring (5)

Now must spill another node

Choose one with the smallest cost/NB → c is chosen

<u>Stack</u>
c (spilled)
e
f (spilled)
g

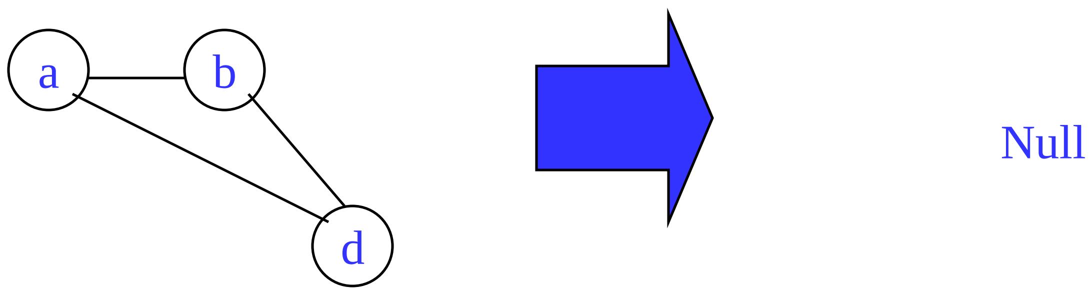


Example – Do a 3-Coloring (6)

Remove all nodes < 3 neighbors

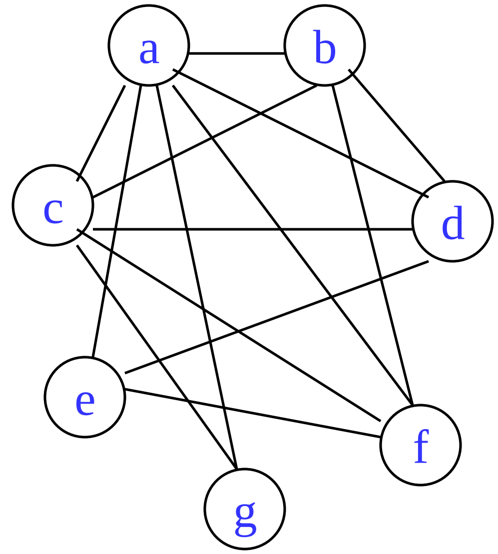
So, a, b, d can be removed

Stack
d
b
a
c (spilled)
e
f (spilled)
g



Example – Do a 3-Coloring (7)

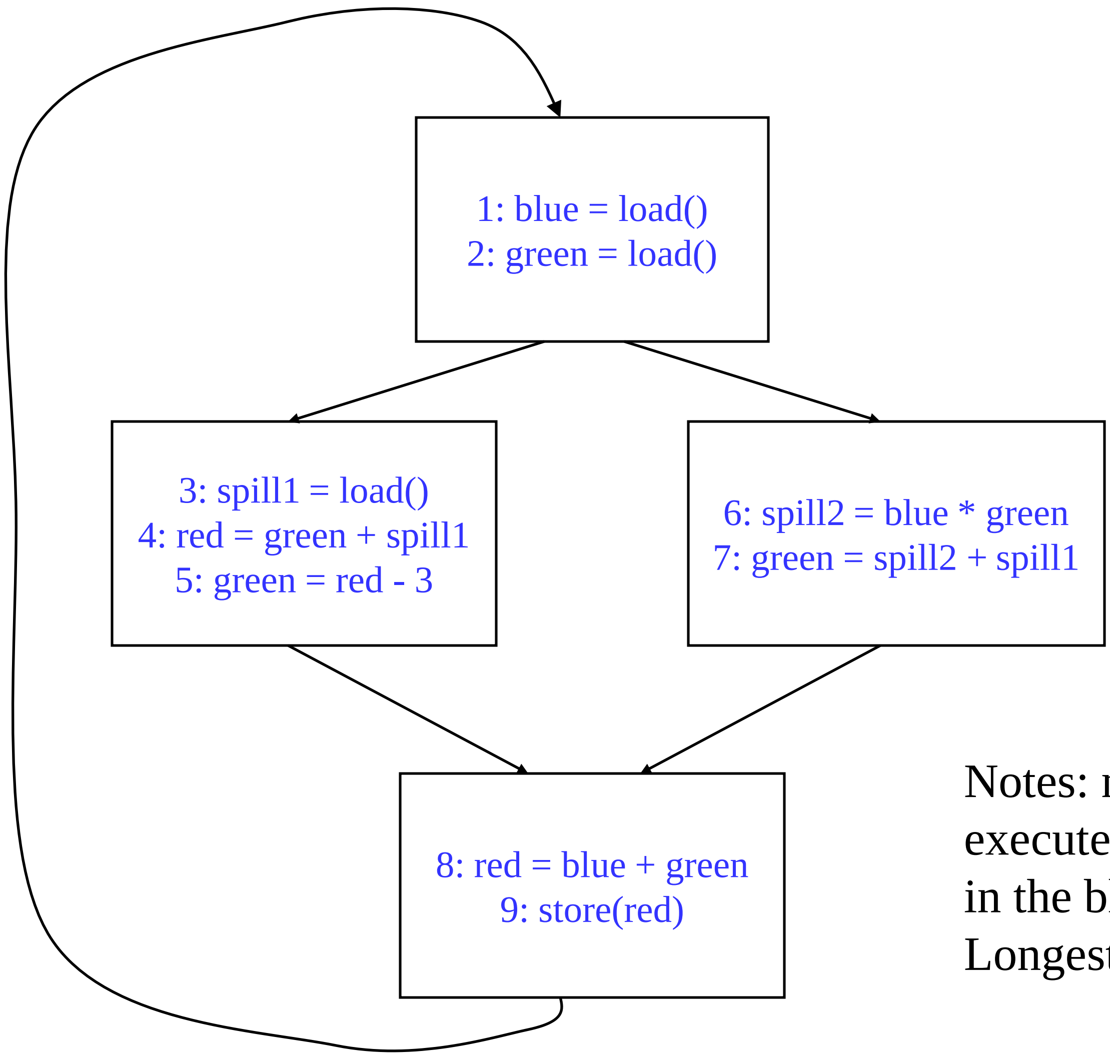
Stack
d
b
a
c (spilled)
e
f (spilled)
g



Have 3 colors: red, green, blue, pop off the stack assigning colors
only consider conflicts with non-spilled nodes already popped off stack

d → red
b → green (cannot choose red)
a → blue (cannot choose red or green)
c → no color (spilled)
e → green (cannot choose red or blue)
f → no color (spilled)
g → red (cannot choose blue)

Example – Do a 3-Coloring (8)



d → red
b → green
a → blue
c → no color
e → green
f → no color
g → red

Notes: no spills in the blocks executed 100 times. Most spills in the block executed 25 times. Longest lifetime (c) also spilled