9-0: **Registers vs. Memory**

- Large quantity of registers (32) at our disposal
- Registers are much faster to access than main memory
- Currently generated code does not use registers efficiently
  - Variables are stored on the stack (main memory), instead of in registers

9-1: **Registers vs. Memory**

- Since registers are so much faster than main memory, use them as much as possible
- Two reasons why we can't use only registers for variables
  - Escaping Variables
  - Spilled Registers

9-2: **Escaping Variables**

- A variable *escapes* if we need to store it in memory
  - Calculate the address of a variable (implicitly or explicitly)
  - Use the address of the variable to access it
- When do we need to calculate the address of a variable?

9-3: **Escaping Variables**

- When do we need to calculate the address of a variable?
  - Explicitly use the & operator in C
  - Pass-by-reference (C++)
  - Arrays / strings
  - Global / extern variables

9-4: **Spilling Registers**

- Only have a finite number of registers (32 in MIPS)
- Programs can require an arbitrarily large number of variables
- Have more variables to store in registers than registers to use
  - Register *spill* into memory

9-5: **Register Allocation**

- How can we determine which variables are stored in which registers, and which values need to spill into memory?
- How can we determine which variables to store in which registers?
- Register assignment can be non-trivial

9-6: **Register Allocation**

```
a = 3           addi a, $zero, 3
c = a + 4       addi c, a, 4
b = 2           addi b, $zero, 2
c = b + c       add  c, b, c
a = c + 2       addi a, c, 2
c = a + 1       addi c, a, i
```

9-7: **Register Allocation**

```
a = 3           addi a, $zero, 3
c = a + 4       addi c, a, 4
b = 2           addi b, $zero, 2
c = b + c       add  c, b, c
a = c + 2       addi a, c, 2
c = a + 1       addi c, a, i
```

- Code has 3 variables
- Could implement it with only 2 registers
- Never need to keep track of value of `a` and `b` at the same time

9-8: **Interference**

- Need to maintain the values of two variables `x` and `y` simultaneously
- `x` and `y` *interfere* with each other
- Can't store `x` and `y` in the same register

9-9: **Register Allocation**

- Assume that we have an infinite number of registers
- Every time we declare a new variable, create a new abstract register
- After we have created assembly, replace abstract registers with actual registers
  - Use registers as efficiently as possible
  - May have to spill some of the abstract registers into memory

9-10: **Register Allocation**

- Create assembly that uses abstract registers
- Figure our which abstract registers interfere with each other
- Use this information to assign actual registers to abstract registers as efficiently as possible

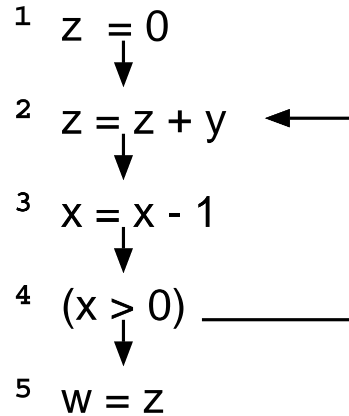
9-11: **Control Flow Graph**

- A Control Flow Graph describes the flow of control through an assembly language program
- Nodes in the graph are assembly language statements

- Edge from node A to node B if statement in node B can immediately follow the statement in node A

9-12: **Control Flow Graph**

```
    z = 0
L1: z = z + y
    x = x - 1
    if (x>0) goto L1
    w = z
```



9-13: **Definitions**

- Node p is a *successor* of Node n if there is an arc from Node n to Node p
- successors[n] = set of all successors of n
- Node p is a *predecessor* of Node n if there is an arc from Node p to Node n
- predecessors[n] = set of all predecessors of n

9-14: **Liveness Analysis**

- We will use the Control Flow Graph to do *Liveness Analysis*
- A variable is "live" on an edge in the control flow graph if changing the value of the variable at that point in the execution will alter the behavior of the program
- Variable is live on an edge if we care about the value of the variable at that point in the program

9-15: **Liveness Analysis**

- "Live In"
  - Variable is live-in to a node if it is live on any edge entering the node
  - live-in[n] = set of all variables live-in to n
- "Live Out"
  - Variable is live-out from a node if it is live on any edge leaving the node
  - live-out[n] = set of all variables live-out from n

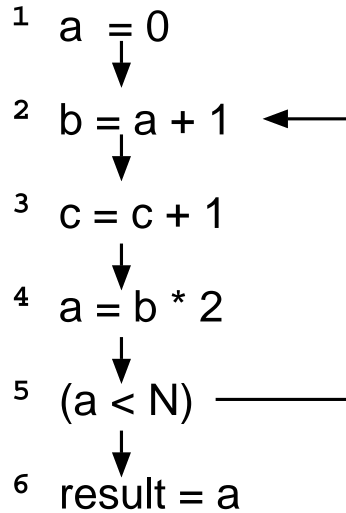
9-16: **Liveness Analysis**

- To calculate live-in[n] and live-out[n], we will need to know which variables are *used* at each node, and which variables are *defined* at each node
  - use[n] = set of all variables whose values are used at node n

- define[n] = set of all variables whose values are defined (set) at node n

9-17: **Control Flow Graph**

```
    a = 0
L1:
    b = a + 1
    c = c + 1
    a = b * 2
    if  (a < N) goto L1
    return a
```



9-18: **Use / Define**

Node	Use	Define
1	{ }	{a}
2	{a }	{b}
3	{c }	{c}
4	{b }	{a}
5	{a,N }	{ }
6	{a }	{result}

9-19: **Live-in / Live-out**

- live-in and live-out can be calculated as follows:

$$
\begin{aligned}
out[n] &= \bigcup_{s \in successors[n]} in[s] \\
in[n] &= use[n] \bigcup (out[n] - define[n])
\end{aligned}
$$

9-20: **Live-in / Live-out**

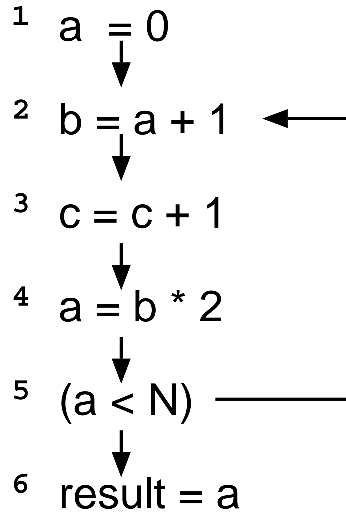
- To calculate live-in and live-out for all nodes:
  - Set in[n] = {}, out[n] = {} for all n
  - For each node n, recalculate in[n] and out[n] using

$$
\begin{aligned}
out[n] &= \bigcup_{s \in successors[n]} in[s] \\
in[n] &= use[n] \bigcup (out[n] - define[n])
\end{aligned}
$$

- Repeat until no changes are made to in[], out[]

9-21: **Liveness Analysis Example**

```
    a = 0
L1:
    b = a + 1
    c = c + 1
    a = b * 2
    if  (a < N) goto L1
    return a
```



9-22: **Liveness Analysis Example**

State	Use	Define	In	Out
1	{ }	{a }	{N, c}	{a, c, N }
2	{a}	{b }	{a, c, N}	{b, c, N}
3	{c}	{c }	{b, c, N}	{b, c, N}
4	{b}	{a }	{b, c, N}	{a, c, N}
5	{a, N}	{ }	{a, c, N }	{a, c, N}
6	{a }	{result }	{a }	{ }

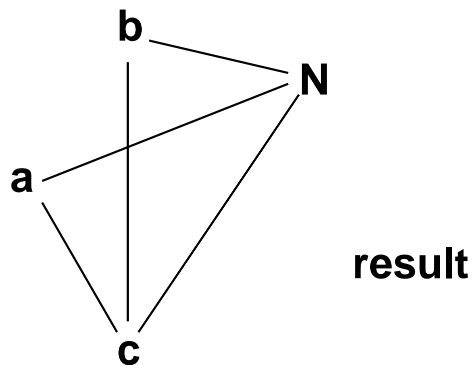
9-23: **Interference Graphs**

- We will use interference graphs to assist in register allocation
- Each node in the graph is a variable
- Edge between two variables if they interfere with each other
    - Need to be in different registers
    - Live at the same time

9-24: **Interference Graphs**

- For each node n in the control flow graph
    - For each variable $x \in$ define[n]
        - For each variable $y \in$ out[n], add an edge between x and y

9-25: **Interference Graphs**



9-26: **Register Allocation**

- Once we have the interference graph, register allocation is just graph coloring
  - Assign colors to all vertices so that adjacent vertices have different colors
  - Colors – actual registers

9-27: **Register Allocation**

- Once we have the interference graph, register allocation is just graph coloring
  - Assign colors to all vertices so that adjacent vertices have different colors
  - Colors – actual registers
- Graph coloring is NP-Complete
  - Use an approximate solution
  - Won't always guarantee the smallest number of colors are used
  - Run in polynomial time

9-28: **Graph Coloring**

- Don't need to find an optimal coloring
- Need to find a coloring that uses fewer than k colors
  - k is the number of actual registers
- Won't always be able to find a k-coloring when one exists (NP-Complete), but will find a k-coloring *most* of the time

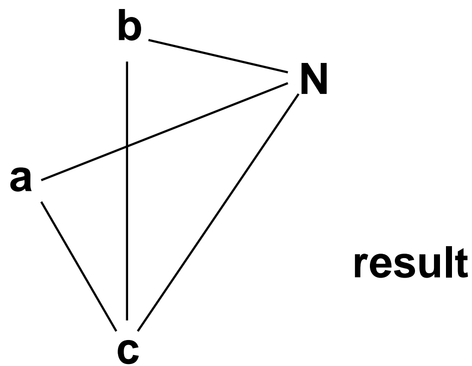
9-29: **Significant Degree**

- "Degree" of a vertex is the number of neighbors that it has
- Vertex with $\geq k$ neighbors is of "Significant Degree"
- If a vertex is *not* of significant degree, then it can always be colored – *regardless of the colors chosen for the rest of the graph*
  - Fewer than k neighbors
  - k colors – can always choose a color different from all neighbors

9-30: **Graph Coloring**

1. Initialize color stack to be empty
2. Select a node x that is not of significant degree. Push x on color stack, remove x from the graph
3. If there are any nodes left, go to step 2
4. While the color stack is not empty
   (a) Pop the top variable off the color stack
   (b) Color this variable with an arbitrary color, different from already colored neighbors

9-31: **Graph Coloring Example**



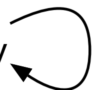
- 3-Color this graph

9-32: **Register Allocation**

Liveness Analysis

Build Interference Graph

Simplify



Color

9-33: **Spilling**

- Not every graph can be k-colored, for all k
- Previous graph cannot be 2-colored

- If we don't have enough registers, some values need to "spill" into memory

9-34: **Spilling**

1. Initialize the color stack to be empty
2. If there are any variables x that are not of significant degree
    - Remove x from the interference graph, and push x onto the color stack.

    else

    - Pick a node y to potentially store in memory. Push y on the color stack, and remove y from the interference graph
3. If there are any nodes left, go to step 2

9-35: **Spilling**

4. Restore the interference graph
5. While the color stack is not empty:
    (a) Pop the top variable off the color stack
    (b) If there is an available color for x that is different than the colors already chosen for x's neighbors:
        - Color x

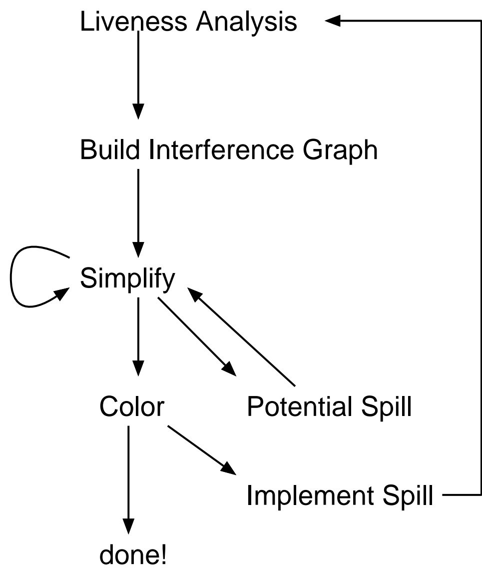
        else

        - Mark x as an actual spill
6. If no variables spilled, we're done. If at least one variable spilled, implement the spill.

9-36: **Spilling**

- If a register *spills*, it must be stored in memory
    - Before every use, read variable from memory
    - After every use, write variable back to memory
    - "Live" period of register will be very short
    - Should be able to color the graph

9-37: **Register Allocation**



9-38: **Register Allocation**

```
d = 2              addi d, zero, 2
b = 3              addi b, zero, 3
a = 4              addi a, zero 4
a = a + 1          addi a, a, 1
c = 4              addi c, zero, 4
b = b + c          add  b, b, c
d = d + 1          addi d, d, 1
```

9-39: **Register Allocation**

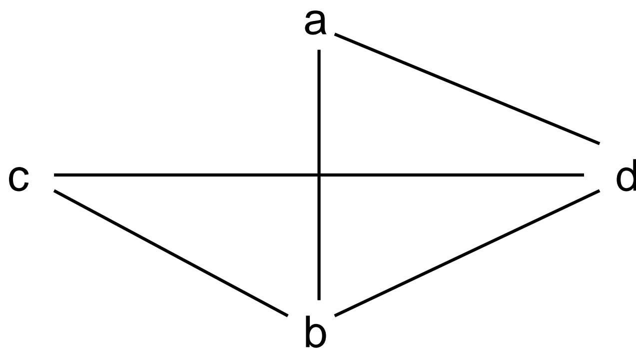
```
d = 2
b = 3
a = 4
a = a + 1
c = 4
b = b + c
d = d + 1
```

```
1 d = 2
2 b = 3
3 a = 4
4 a = a + 1
5 c = 4
6 b = b + c
7 d = d + 1
```

9-40: **Register Allocation**

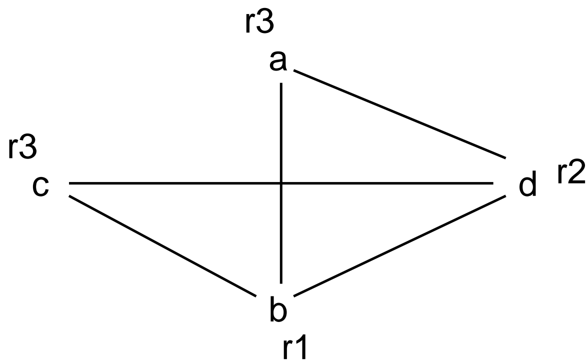
State	Use	Define	In	Out
1	{ }	{d }	{ }	{d }
2	{ }	{b }	{d}	{b, d}
3	{ }	{a }	{b, d}	{b, a, d}
4	{a}	{a }	{b, a, d}	{b, d}
5	{}	{c }	{b, d}	{b, c, d}
6	{b, c}	{b }	{b, c, d}	{d}
7	{d }	{d }	{d}	{ }

9-41: **Register Allocation**



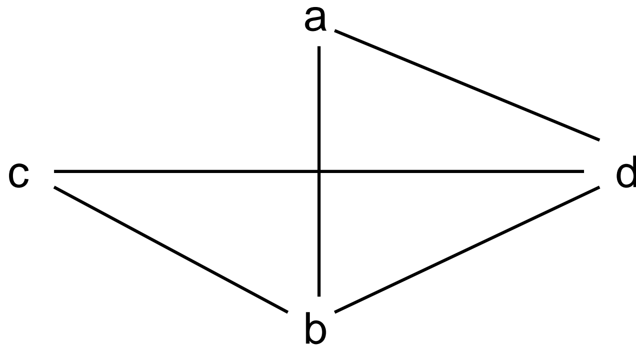
- 3-color

9-42: **Register Allocation**



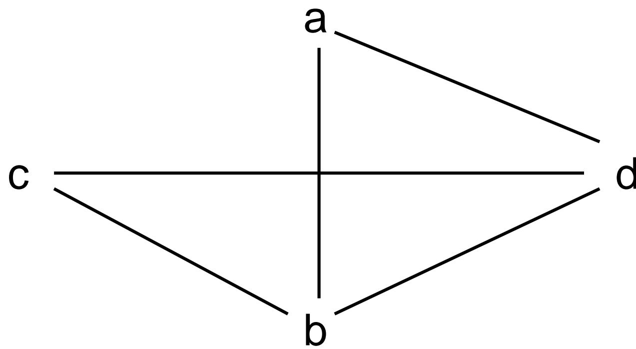
- 3-color

9-43: **Register Allocation**



- 2-color

9-44: **Register Allocation**



- 2-color
  - What should we pick as a potential spill?

9-45: **Register Allocation**

```
d = 2              addi d, zero, 2
b = 3              addi b, zero, 3
a = 4              addi a, zero 4
a = a + 1          addi a, a, 1
c = 4              addi c, zero, 4
b = b + c          add  b, b, c
d = d + 1          addi d, d, 1
```

9-46: **Register Allocation**

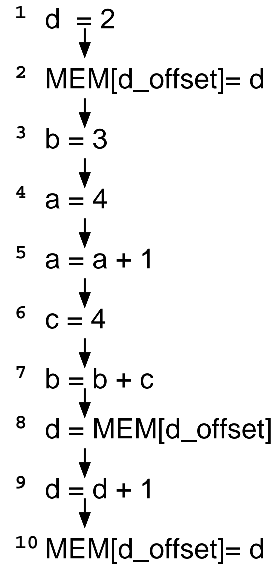
```
d = 2              addi d, zero, 2
MEM[d_offset] = d  sw   d, (d_offset) fp
b = 3              addi b, zero, 3
a = 4              addi a, zero 4
a = a + 1          addi a, a, 1
c = 4              addi c, zero, 4
b = b + c          add  b, b, c
```

```
d = MEM[d_offset]  lw   d, (d_offset) fp
d = d + 1          addi d, d, 1
MEM[d_offset] = d  sw   d, (d_offset) fp
```

9-47: **Register Allocation**

```
d = 2
MEM[d_offset] = d
b = 3
a = 4
a = a + 1
c = 4
b = b + c
d = MEM[d_offset]
d = d + 1
MEM[d_offset] = d
```

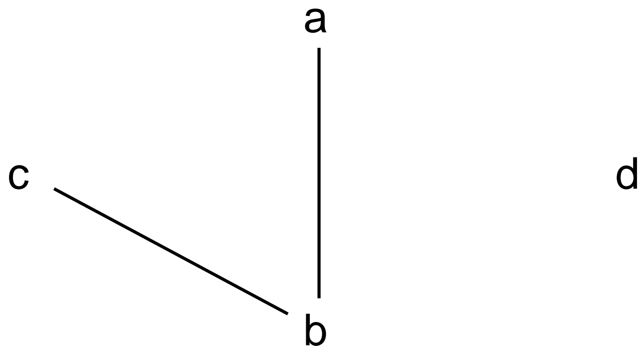
9-48: **Register Allocation**



9-49: **Register Allocation**

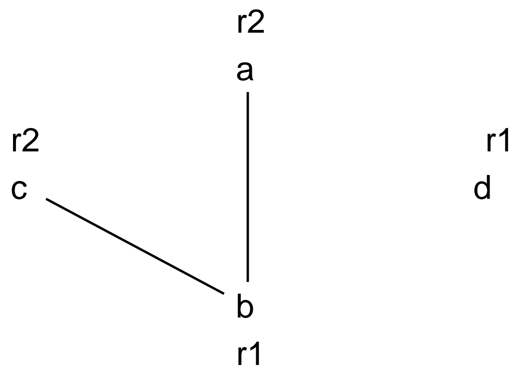
State	Use	Define	In	Out
1	{ }	{d}	{ }	{d}
2	{d}	{ }	{d}	{}
3	{ }	{b}	{}	{b}
4	{ }	{a}	{b}	{a, b}
5	{a}	{a}	{a, b}	{b}
6	{ }	{c}	{b}	{b, c}
7	{b, c}	{b}	{b, c}	{ }
8	{ }	{d}	{ }	{d}
9	{d}	{d}	{d}	{d}
10	{d}	{ }	{d}	{ }

9-50: **Register Allocation**



d

- 2-color



- 2-color

9-51: **Eliminating Moves**

- When two variables share the same value, we can use the same register for both of them
- Any move from one variable to the other can then be removed.

9-52: **Eliminating Moves**

```
a = 4
c = a + 1
b = c + 1
d = a
c = d + 1
b = d * c
```

- We can store `a` and `d` in the same register
- The assignment `d = a` is then a no-op, which can be removed

9-53: **Eliminating Moves**

- Any time there is a move instruction
  - `x = y` `(addi x, y, 0)`
- `x` and `y` do *not* interfere (unless they interfere elsewhere)
- If we place `x` and `y` in the same register, we can eliminate the move instruction

9-54: **Interference Graph, Part II**

- Interference Graph will have two kinds of edges
  - Interference edges
    - Interference edge between `x` and `y`
    - `x` and `y` must go in different registers
  - Move edges
    - Move edge between `x` and `y`

- Like `x` and `y` to go in the same register
- Eliminate a move instruction, save some cycles

9-55: **Interference Graph, Part II**

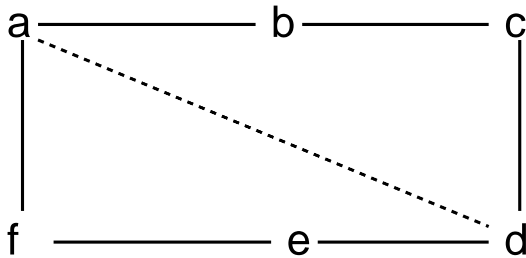
- For node n in the control flow graph
  - If n is a move instruction of the form x = z
    - Add a move arc between x and z
    - For each variable y in out[n] - z, add an undirected interference arc between x and y.

    else if n is not a move instruction
    - For each variable x in define[n] and each variable y in out[n] add an undirected interference arc between x and y
  - Interference Edges trump Move Edges
    - Add both an interference edge and move edge, interference edge wins

9-56: **Eliminating Moves & Spilling**

- Eliminating moves adds some efficiency
- Reducing spills is more important
  - Cost of going out to memory much higher than cost of an `addi`
- Careful not to be too aggressive in trying to eliminate moves

9-57: **Eliminating Moves & Spilling**



- Can 2-color this graph
- Cannot 2-color this graph if `a` and `d` have the same color

9-58: **Register Allocation**

1. Do dataflow analysis to calculate live-out for all nodes in the control-flow graph
2. Create the interference graph
3. Combine all pairs of nodes that are connected by a move arc that will not result in a node with k or more neighbors of significant degree (Coalesce step)

9-59: **Register Allocation**

4. Initialize the color stack to be empty

5. If there are any variables x that are not of significant degree
   - Remove x from the interference graph, and push x onto the color stack.

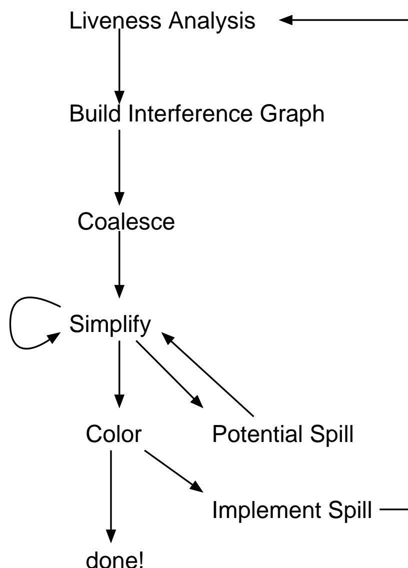
   else
   - Pick a node y to potentially store in memory. Push y on the color stack, and remove y from the interference graph
6. If there are any nodes left, go to step 5

9-60: **Register Allocation**

7. Restore the interference graph
8. While the color stack is not empty:
   (a) Pop the top variable off the color stack
   (b) If there is an available color for x that is different than the colors already chosen for x's neighbors:
   - Color x

   else
   - Mark x as an actual spill
9. If no variables spilled, we're done. If at least one variable spilled, implement the spill and start over.

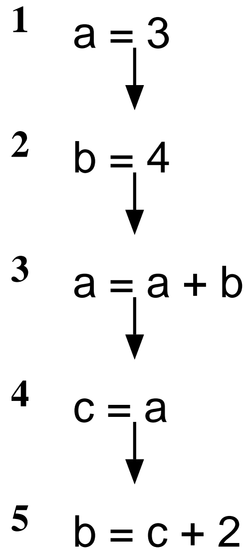
9-61: **Register Allocation**



9-62: **Register Allocation**

```
a = 3          addi a, $zero, 3
b = 4          addi b, $zero, 4
a = a + b      add  a, a, b
c = a          addi c, a, 0
b = c + 2      addi b, c, 2
```

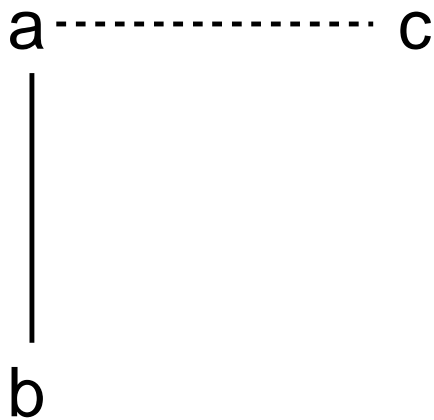
9-63: **Register Allocation**



9-64: **Register Allocation**

State	Use	Define	In	Out
1	{ }	{a}	{ }	{a}
2	{ }	{b}	{a}	{a, b}
3	{a, b}	{a}	{a, b}	{a}
4	{a}	{c}	{a}	{c}
5	{c}	{b}	{c}	{ }

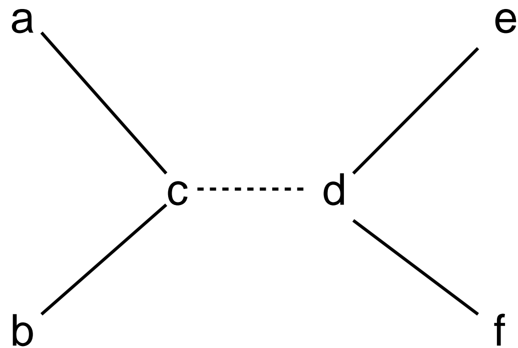
9-65: **Register Allocation**



9-66: **Coalescing**

- We need to be sure that no coalesce steps cause unnecessary spills
- If we do all coalescing before any simplifying, we will not always take advantage of move elimination

9-67: **Coalescing**



- 3-color this graph

9-68: **Coalescing**

- We need to be sure that no coalesce steps cause unnecessary spills
- If we do all coalescing before any simplifying, we will not always take advantage of move elimination
- Interleave Coalesce and Simplify steps
    - Coalesce as much as we can
    - Simplify as much as we can (not simplifying any node that we could possible coalesce later)
    - repeat

9-69: **Register Allocation (Final)**

1. Do dataflow analysis to calculate live-out for all nodes in the control-flow graph
2. Create the interference graph (including move edges)
3. Simplify any variables that are not of significant degree that have no move edges. That is, remove the node from the graph and add it to the color stack.
4. Coalesce (combine) all pairs of nodes that are connected by a move arc that will not result in a node of significant degree.

9-70: **Register Allocation (Final)**

5. If there are any nodes that are not of significant degree that are not involved in any move, go to step 3.
6. If there is a node x that is not of significant degree, which is involved in at least one move, then freeze x – remove all move edges adjacent to x – and goto step 3.
7. If there are any remaining nodes (at this point they all must be of significant degree), pick a node x to potentially spill. Remove x from the graph and place it on the color stack. Goto step 3.

9-71: **Register Allocation (Final)**

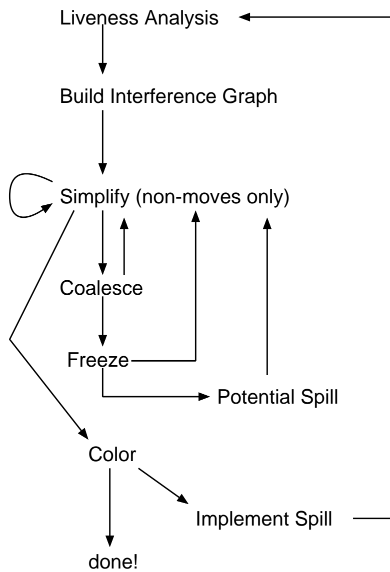
8. At this point, the graph should be empty. Restore the interference graph
9. While the color stack is not empty:

   (a) Pop the top variable off the color stack
   (b) If there is an available color for x that is different than the colors already chosen for x's neighbors:
      - Color x

      else
      - Mark x as an actual spill

10. If no variables spilled, we're done. If at least one variable spilled, implement the spill and start over.

9-72: **Register Allocation (Final)**

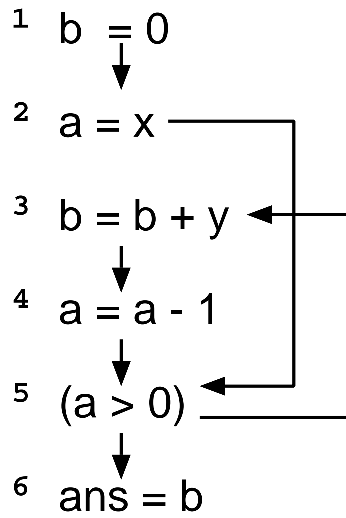


9-73: **Register Allocation Example**

```
     b = 0                         addi b, zero, 0
     a = x                         addi a, x, 0
     goto test                     b test
body:                         body:
     b = b + y                     add  b, b, y
     a = a - 1                     addi a, a, -1
test:                         test:
     if (a>0) goto body            bgt a
     ans = b                       addi ans, b, 0
```

9-74: **Register Allocation Example**

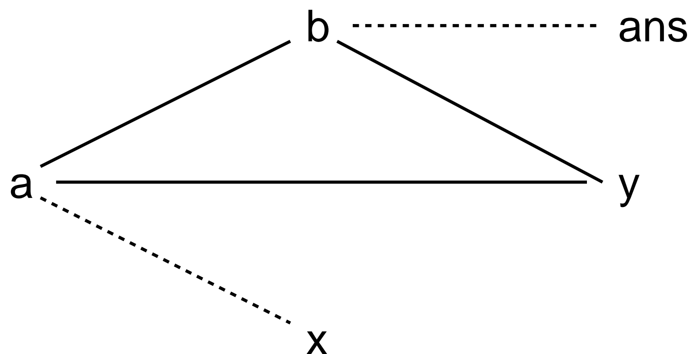
```
    b = 0
    a = x
    goto test
body:
    b = b + y
    a = a - 1
test:
    if (a>0) goto body
    ans = b
```



9-75: **Register Allocation Example**

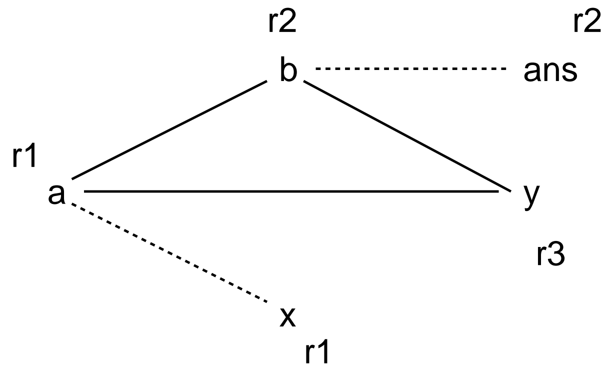
State	Use	Define	In	Out
1	{ }	{b}	{x, y}	{b, x, y}
2	{x}	{a}	{b, x, y}	{a, b, y}
3	{b, y}	{b}	{a, b, y}	{a, b, y}
4	{a}	{a}	{a, b, y}	{a, b, y}
5	{a}	{ }	{a, b, y}	{a, b, y}
6	{b}	{ans}	{b}	{}

9-76: **Register Allocation Example**



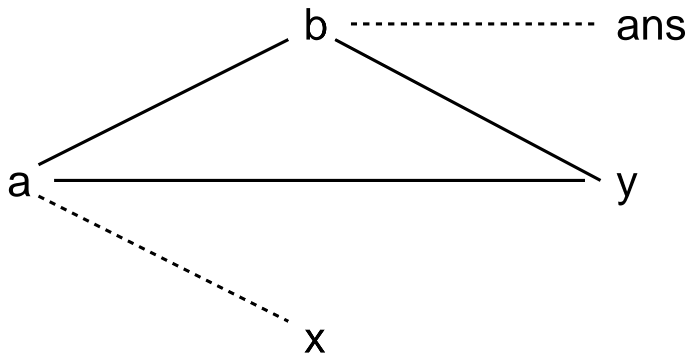
- 3-color graph

9-77: **Register Allocation Example**



- 3-color graph

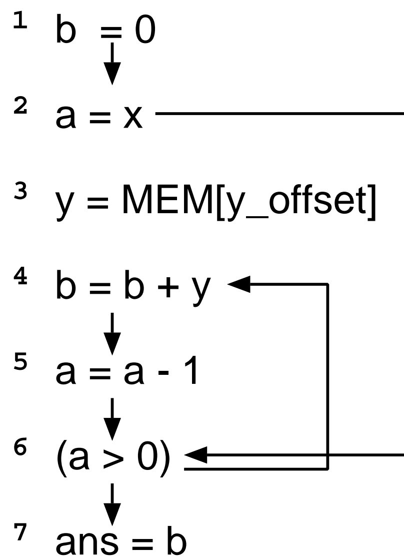
9-78: **Register Allocation Example**



- 2-color graph

9-79: **Register Allocation Example**

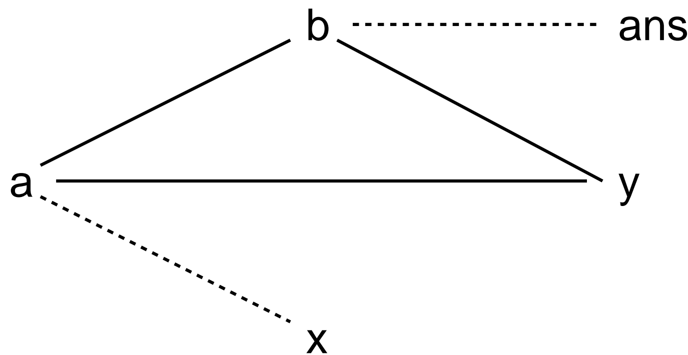
```
      b = 0
       a = x
       goto test
body:
       y=MEM[y_offset]
        b = b + y
        a = a - 1
test:
        if (a>0) goto body
        ans = b
```



9-80: **Register Allocation Example**

State	Use	Define	In	Out
1	{ }	{b}	{x}	{b, x}
2	{x}	{a}	{b, x}	{a, b}
3	{ }	{y}	{a, b}	{a, b, y}
4	{b, y}	{b}	{a, b, y}	{a, b}
5	{a}	{a}	{a, b}	{a, b}
6	{a}	{ }	{a, b}	{a, b}
7	{b}	{ans}	{b}	{}

9-81: **Register Allocation Example**



- Whoops!
- Only 2 registers, need to store everything in memory

9-82: **Precolored Registgers**

- Some registers (FP, SP, etc) are not variables
- Need to assign specific registers to them
- “Precolor” some of the variables in the graph
    - FP
    - SP
    - ... etc

9-83: **Parameter Passing**

- Pass parameters on the stack, saving & restoring registers to memory
- Better idea – use registers to pass parameters
    - Identify some registers as “parameter registers”
    - Copy paramters into these registers before function calls
    - Move values out of these registers at start of the function
- Function calls within function calls won’t cause problems
- Might be able to eliminate some of the moves

9-84: **Parameter Passing**

- To call a function, move values of actuals into special (precolored) parameter registers
- At the start of each function, move values of parameters into other variables
  - May be able to color these variables with the same color as precolored – eliminate the move

9-85: **Parameter Passing**

- Most functions have a (relatively) small number of parameters
- Wasteful to denote too many registers as parameter registers
- Only use 4-5 parameter registers (other parameters are passed on the stack as normal

9-86: **Caller/Callee Saved**

- One solution to register allocation:
  - Create a control flow graph of the entire program (all function calls, etc)
  - Do dataflow analysis on this graph
  - Allocate registers based on dataflow analysis
- What is wrong with this approach?

9-87: **Caller/Callee Saved**

- One solution to register allocation:
  - Create a control flow graph of the entire program (all function calls, etc)
  - Do dataflow analysis on this graph
  - Allocate registers based on dataflow analysis
- What is wrong with this approach?
  - Separately compiled functions
  - Libraries
  - etc.

9-88: **Caller/Callee Saved**

- Do register allocation on a function by function basis
- Denote some registers as "caller-saved"
  - Assume that values of these registers will be destroyed by function calls
  - Save them on the stack if they are live across other function calls
- Denote some registers as "callee-saved"
  - If use these registers, save old values on the stack

9-89: **Caller/Callee Saved**

- At beginning of a function:
  - Move all callee-saved registers other temporary locations

  - Move all parameters into other locations
- If the function uses any callee saved registers, then the temporary locations will spill into memory – automatically saving registers
- If the function does not need to use any callee saved registers, no unnecessary movement to memory

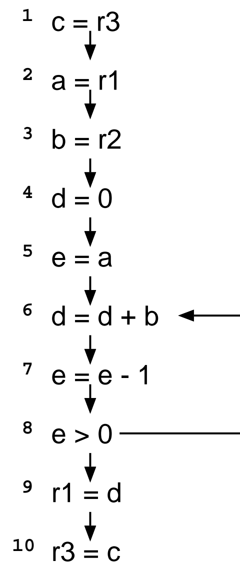
9-90: **Complete Example**

```
int f(int a, int b) {
   int d = 0
   in e = a;
   do {
     d = d + b;
     e = e - 1;
   while (e > 0);
   return d;
```

- 3 registers, r1, r2, r3
  - r1, r2 are parameter registers
  - r1 is also the result register
  - r3 is callee-saved

9-91: **Complete Example**

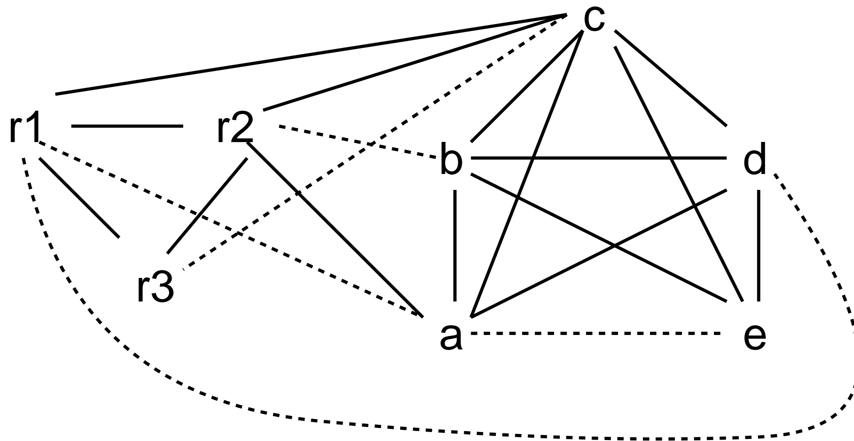
```
   c = r3
   a = r1
   b = r2
   d = 0
   e = a
loop:
   d = d + b
   e = e - 1
   if e > 0 goto loop
   r1 = d
   r3 = c
```



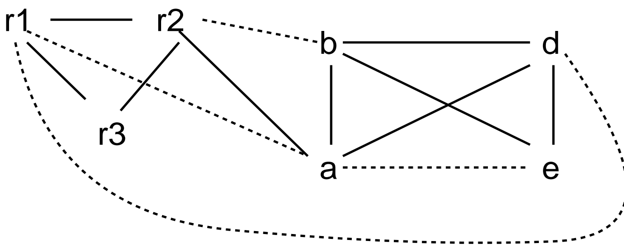
9-92: **Complete Example**

State	Use	Define	In	Out
1	{r3}	{c}	{r1, r2, r3}	{c, r1, r2}
2	{r1}	{a}	{c, r1, r2}	{a, c, r2}
3	{r2}	{b}	{a, c, r2}	{a, b, c}
4	{ }	{d}	{a, b, c}	{a, b, c, d}
5	{a}	{e}	{a, b, c, d}	{b, c, d, e}
6	{b, d}	{d}	{b, c, d, e}	{b, c, d, e}
7	{e}	{e}	{b, c, d, e}	{b, c, d, e}
8	{e}	{ }	{b, c, d, e}	{b, c, d, e}
9	{d}	{r1}	{c, d}	{r1, c}
10	{c}	{r3}	{r1, c}	{r1, r3}

9-93: **Complete Example**

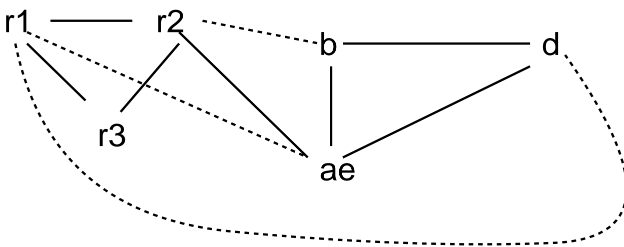


9-94: **Complete Example**



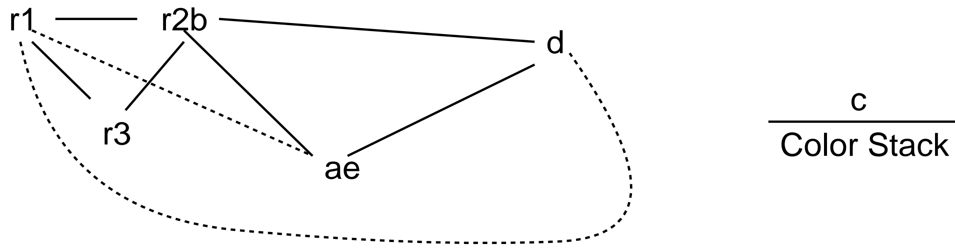
c
Color Stack

9-95: **Complete Example**

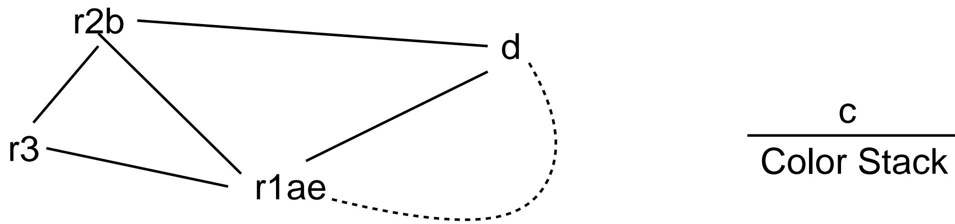


c
Color Stack

9-96: **Complete Example**

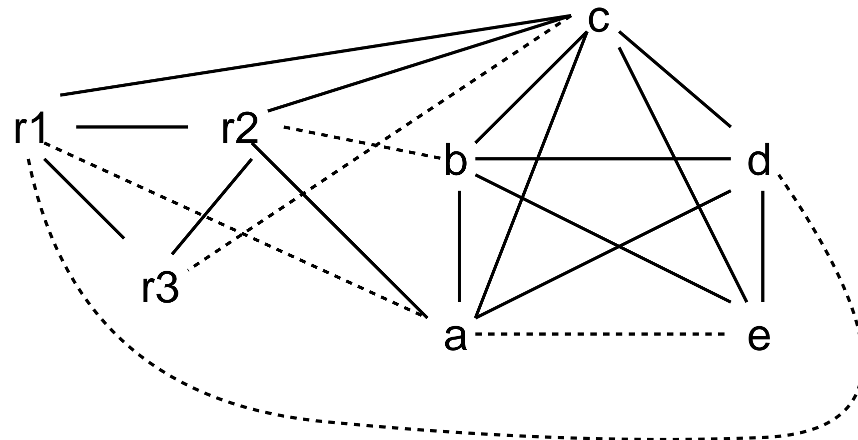


9-97: **Complete Example**

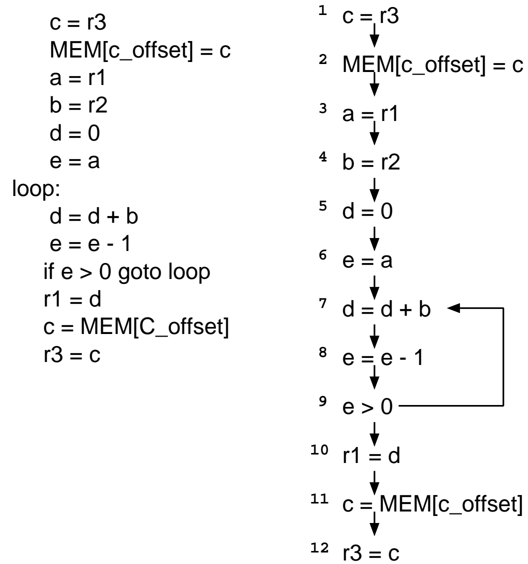


9-98: **Complete Example**

9-99: **Complete Example**



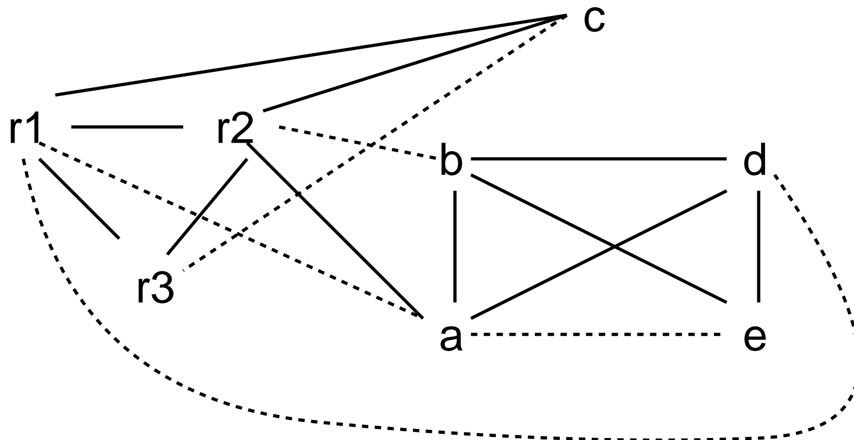
9-100: **Complete Example**



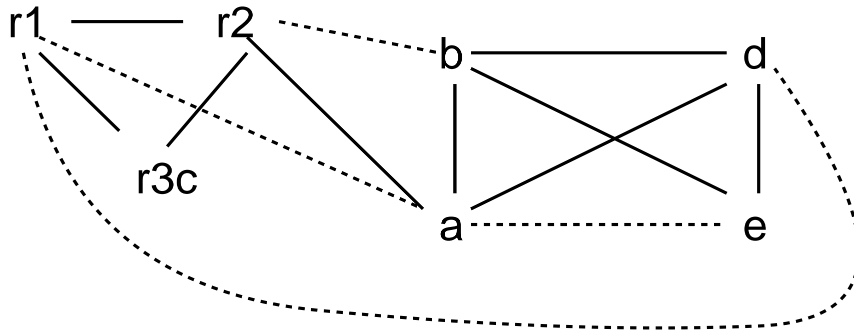
9-101: **Complete Example**

State	Use	Define	In	Out
1	{r3}	{c}	{r1, r2, r3}	{c, r1, r2}
2	{c}	{ }	{c, r1, r2,}	{r1, r2}
3	{r1}	{a}	{r1, r2}	{a, r2}
4	{r2}	{b}	{a,r2}	{a, b}
5	{ }	{d}	{a, b}	{a, b, d}
6	{a}	{e}	{a, b, d}	{b, d, e}
7	{b, d}	{d}	{b, d, e}	{b, d, e}
8	{e}	{e}	{b, d, e}	{b, d, e}
9	{e}	{ }	{b, d, e}	{b, d, e}
10	{d}	{r1}	{d}	{r1}
11	{}	{c}	{r1}	{c, r1}
12	{c}	{r3}	{c, r1}	{r1, r3}

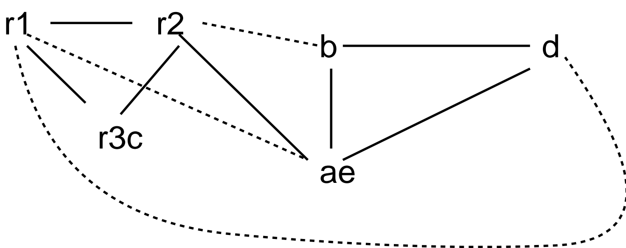
9-102: **Complete Example**



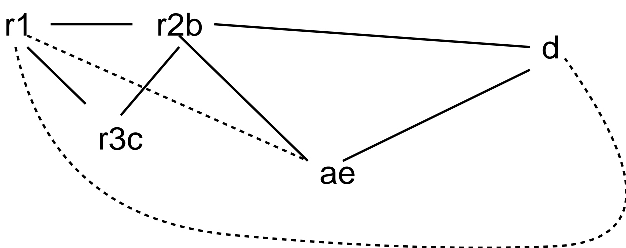
9-103: **Complete Example**



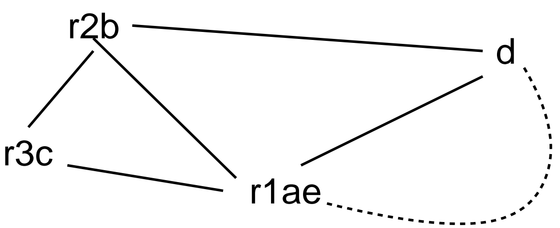
9-104: **Complete Example**



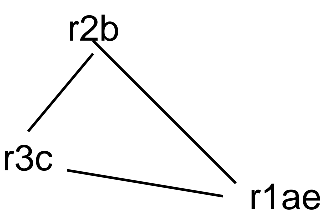
9-105: **Complete Example**



9-106: **Complete Example**



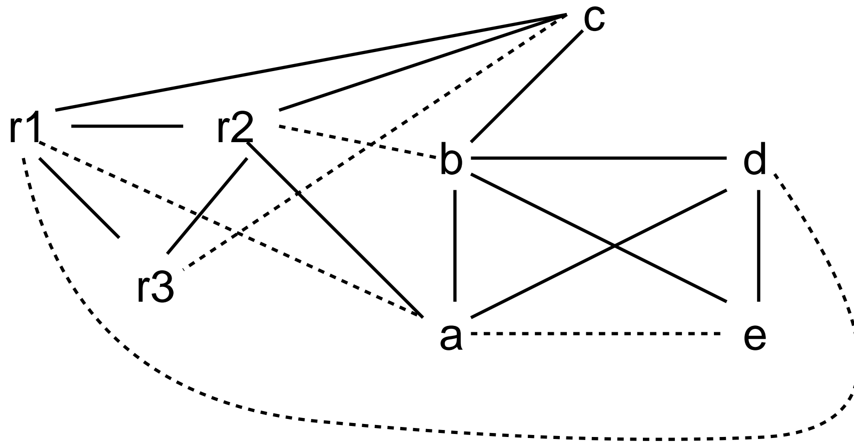
9-107: **Complete Example**



d

Color Stack

9-108: **Complete Example**



9-109: **More examples**

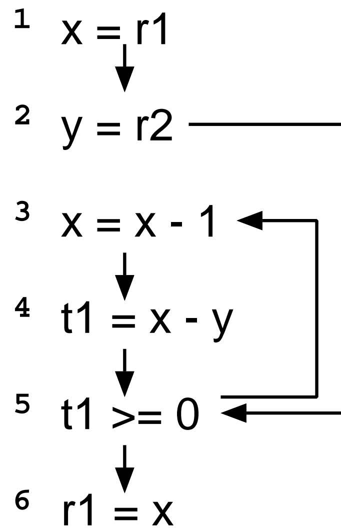
```
int mod(int x, int y) {
   while (x >= y)
       x = x - y;
   return x;

}
```

```
mod:
    x = r1
    y = r2
    goto test
body:
    x = x - 1
test:
    t1 = x - y
    if (t1 >= 0) goto body
    r1 = x
```

- r1, r2 are parameter registers
- r1 is (also) the result register

9-110: **More examples**

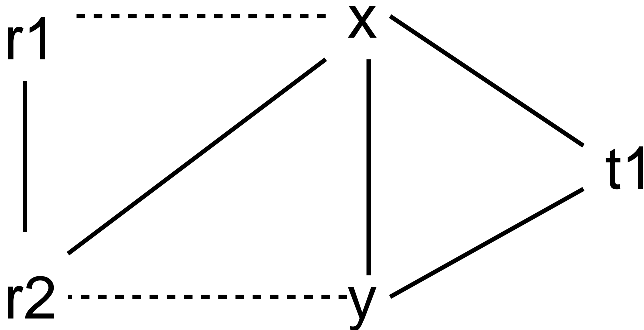
```
mod:
      x = r1
      y = r2
      goto test
body:
      x = x - 1
test
      t1 = x - y
      if (t1>=0) goto body
      r1 = x
```



9-111: **More examples**

State	Use	Define	In	Out
1	{r1}	{x}	{r1, r2}	{x, r2 }
2	{r2}	{y}	{x, r2}	{x, y}
3	{x}	{x}	{x, y}	{x, y}
4	{x, y}	{t1}	{x, y}	{x, t1}
5	{t1}	{ }	{x, t1}	{x}
6	{x}	{r1}	{x}	{ }

9-112: **More examples**

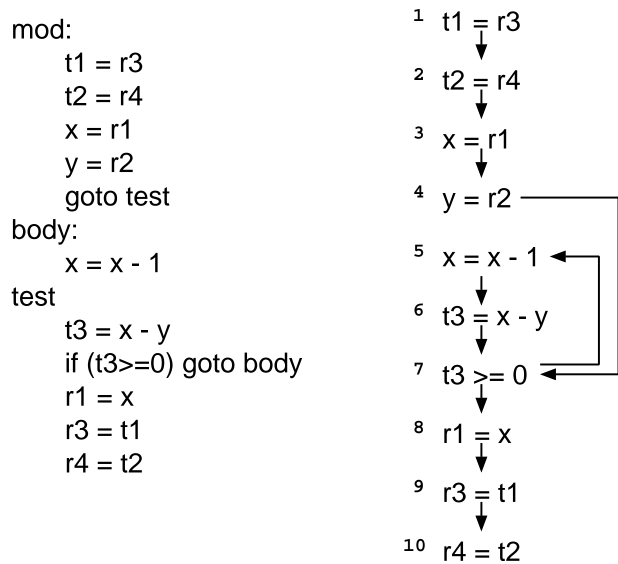


9-113: **More examples**

```
int mod(int x, int y) {        mod:
   while (x >= y)                  t1 = r3
       x = x - y                   t2 = r3
   return x                        x = r1
                                   y = r2
                                   goto test
                               body:
}                                  x = x - 1
                               test:
                                   t3 = x - y
                                   if (t3 >= 0) goto body
                                   r1 = x
                                   r3 = t1
                                   r4 = t2
```

- r1, r2 are parameter registers
- r1 is (also) the result register
- r3, r4 are callee-save

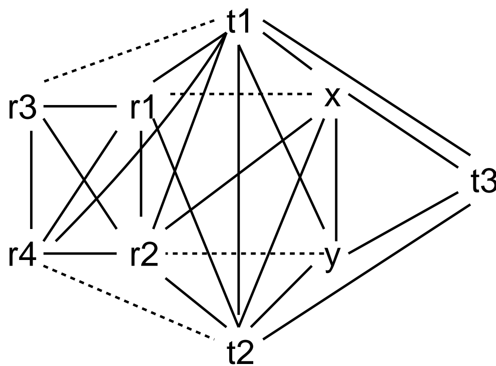
9-114: **More examples**



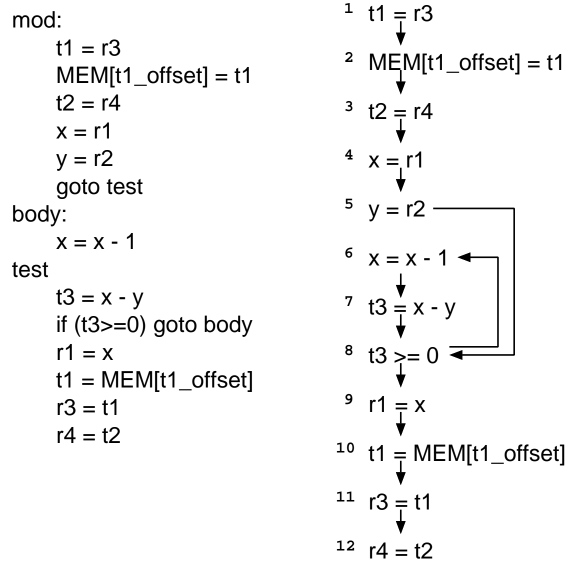
9-115: **More examples**

State	Use	Define	In	Out
1	{r3}	{t1}	{r1,r2,r3,r4}	{t1,r1,r2,r4}
2	{r4}	{t2}	{t1,r1,r2,r4}	{t1,t2,r1,r2}
3	{r1}	{x}	{t1,t2,r1,r2}	{t1,t2,x,r2}
4	{r2}	{y}	{t1,t2,x,r2}	{t1,t2,x,y}
5	{x}	{x}	{t1,t2,x,y}	{t1,t2,x,y}
6	{x,y}	{t3}	{t1,t2,x,y}	{t1,t2,t3,x,y }
7	{t1}	{ }	{t1,t2,t3,x,y}	{t1,t2,x}
8	{x}	{r1}	{t1,t2,x}	{t1,t2}
9	{t1}	{r3}	{t1,t2}	{t2}
10	{t2}	{r4}	{t2}	{ }

9-116: **More examples**



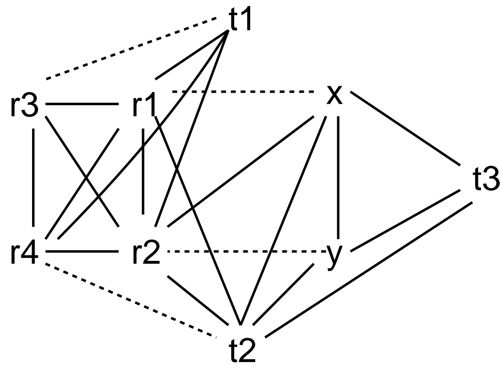
9-117: **More examples**



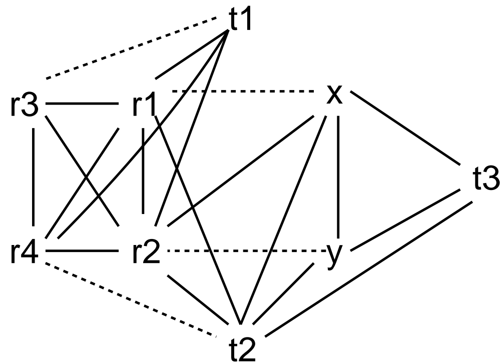
9-118: **More examples**

State	Use	Define	In	Out
1	{r3}	{t1}	{r1,r2,r3,r4}	{t1,r1,r2,r4}
2	{t1}	{ }	{t1,r1,r2,r4}	{r1,r2,r4}
3	{r4}	{t2}	{r1,r2,r4}	{t2,r1,r2}
4	{r1}	{x}	{t2,r1,r2}	{t2,x,r2}
5	{r2}	{y}	{t2,x,r2}	{t2,x,y}
6	{x}	{x}	{t2,x,y}	{t2,x,y}
7	{x,y}	{t3}	{t2,x,y}	{t2,t3,x,y }
8	{t1}	{ }	{t2,t3,x,y}	{t2,x}
9	{x}	{r1}	{t2,x}	{t2}
10	{t1}	{ }	{t2}	{t1,t2}
11	{t1}	{r3}	{t1,t2}	{t2}
12	{t2}	{r4}	{t2}	{ }

9-119: **More examples**



9-120: **More examples**



9-121: **More examples**

```
mod:
  t1 = r3
  MEM[t1_offset] = t1
  t2 = r4
  x = r1
  y = r2
  goto test
body:
   x = x - 1
test:
  t3 = x - y
  if (t3 >= 0) goto body
  r1 = x
  MEM[t1_offset] = t1
  r3 = t1
  r4 = t2
```

9-122: **More examples**

```
mod:
  r3 = r3
  MEM[t1_offset] = r3
  r4 = r4
  r1 = r1
  r2 = r2
  goto test
body:
  r1 = r1 - 1
test:
  r3 = r1 - r2
  if (r3 >= 0) goto body
  r1 = r1
  MEM[t1_offset] = r3
  r3 = r3
  r4 = r4
```

9-123: **More examples**

```
mod:
  MEM[t1_offset] = r3
  goto test
body:
  r1 = r1 - 1
test:
  r3 = r1 - r2
  if (r3 >= 0) goto body
  MEM[t1_offset] = r3
```

9-124: **Calling functions**

- Any function call (jump to register)
  - Defines all callee-saved registers
  - Defines all paramter registers
  - Defines result register
- All necesary saving of registers to the stack then happens automatically