

# Compilation (#7):

## Register Allocation on SSA

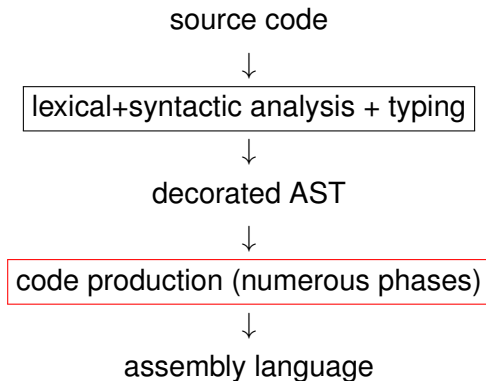
Laure Gonnord & Matthieu Moy & Gabriel Radanne & other

Master 1, ENS de Lyon et Dpt Info, Lyon1

2021-2022



# Where are we ?



- We work on IRs (Middle-end), more specifically: SSA

- 1 Properties in SSA: Liveness
- 2 Register Allocation with graph coloring
- 3 Register Allocation on SSA
- 4 LAB : smart code Generation

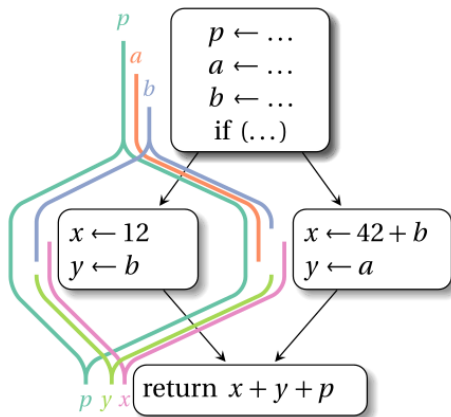
# Liveness: Recap

Liveness is essential for many optimization, notably register allocation.

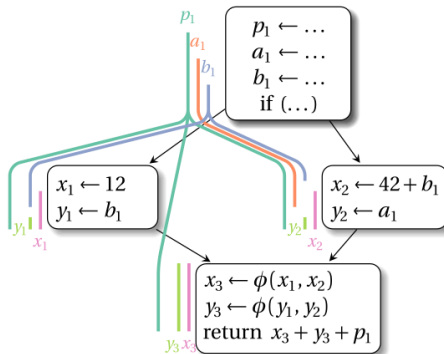
## Definition (Alive Variable)

*In a given program point, a variable is said to be alive if the value it contains may be used in the rest of the execution.*

# Liveness: SSA to the rescue



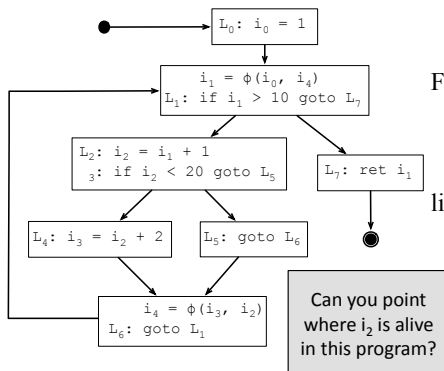
Live range on a CFG



Live range with SSA

## Liveness Analysis in SSA Form Programs

- The problem of determining the program points along which a variable is alive has a simple solution for SSA form programs.



For each statement  $S$  in the program:  
 $IN[S] = OUT[S] = \{\}$

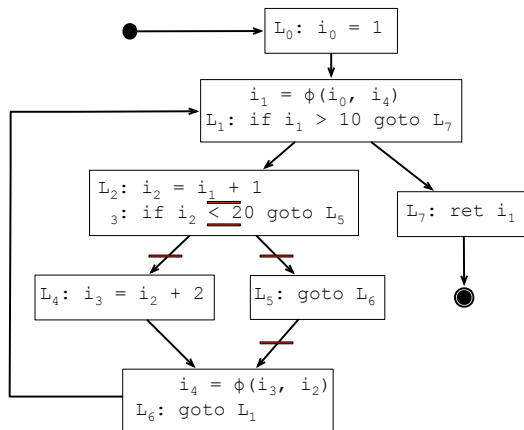
For each variable  $v$  in the program:  
 For each statement  $S$  that uses  $v$ :  
 $live(S, v)$

$live(S, v)$ :  
 $IN[S] = IN[S] \cup \{v\}$   
 For each  $P$  in  $pred(S)$ :  
 $OUT[P] = OUT[P] \cup \{v\}$   
 if  $P$  does not define  $v$   
 $live(P, v)$

## Liveness Analysis in SSA Form Programs

The points where  $i_2$  is alive  
have been marked with red  
rectangles.

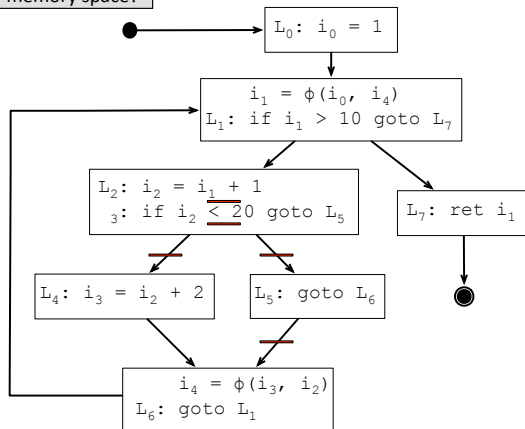
Tricky question:  
is  $i_2$  alive  
somewhere  
within block  $L_6$ ?



## Liveness Analysis in SSA Form Programs

The answer for the tricky question is **NO**. Uses of variables in phi-functions are considered in a different way. The variable is effectively used in the OUT set of the predecessor block where its definition comes from. In other words,  $i_2$  is alive at  $\text{OUT}[L_5]$ , but is not alive at  $\text{IN}[L_6]$ .

Could  $i_2$  and  $i_3$  be allocated into the same memory space?







## Liveness Analysis in SSA Form Programs

Why can we solve liveness analysis for SSA form programs without having to iterate through a fixed point algorithm?

For each statement  $S$  in the program:

$$\text{IN}[S] = \text{OUT}[S] = \{\}$$

For each variable  $v$  in the program:

For each statement  $S$  that uses  $v$ :

$$\text{live}(S, v)$$

$\text{live}(S, v)$ :

$$\text{IN}[S] = \text{IN}[S] \cup \{v\}$$

For each  $P$  in  $\text{pred}(S)$ :

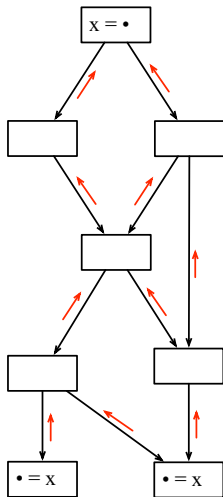
$$\text{OUT}[P] = \text{OUT}[P] \cup \{v\}$$

if  $P$  does not define  $v$

$$\text{live}(P, v)$$

⊕: Notice that phi-functions should be handled in a different way. Do you know why and how?

## Liveness Analysis in SSA Form Programs



Our algorithm works due to the key property of SSA form programs: every use of a variable  $v$  is dominated by the definition of  $v$ . Thus, we can traverse the CFG of the program, starting from the uses of a variable, until we stop at its definition. We are certain to stop, because of the key property. Otherwise, the variable is used without being defined. In this case, we will reach the root node of the CFG, and we assume that the variable is alive at the input of the program.

- 1 Properties in SSA: Liveness
- 2 Register Allocation with graph coloring
- 3 Register Allocation on SSA
- 4 LAB : smart code Generation

## Step 2: Interferences

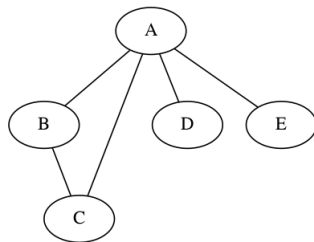
Here is the output of the liveness analysis for  $a + (b + c)$ :

	<i>tmp1</i>	<i>tmp2</i>	<i>tmp3</i>	<i>tmp4</i>	<i>tmp5</i>	<i>tmp6</i>
load <i>tmp1</i> , <i>la</i>						
load <i>tmp2</i> , <i>lb</i>	■					
load <i>tmp3</i> , <i>lc</i>	■	■				
ADD <i>tmp4</i> , <i>tmp2</i> , <i>tmp3</i>	■	■	■			
LETI <i>tmp5</i> , <i>tmp4</i>	■			■		
ADD <i>tmp6</i> , <i>tmp1</i> , <i>tmp5</i>	■				■	
⋮						■

- *tmp1* is in conflict with *tmp2* (because of instruction 3) denoted by  $tmp_1 \bowtie tmp_2$ .

## Interference graph

A denotes `tmp1`, ...  $\bowtie$  defines a graph:



We want a **correct allocation** with respect to  $\bowtie$ :

$$tmp_1 \bowtie tmp_2 \implies Alloc(tmp_1) \neq Alloc(tmp_2).$$

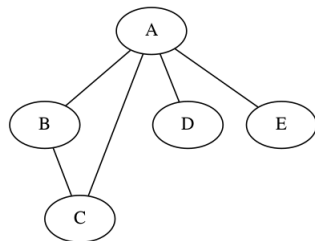
► Graph coloring.

# Live variables and Minimum registers

	<i>tmp1</i>	<i>tmp2</i>	<i>tmp3</i>	<i>tmp4</i>	<i>tmp5</i>	<i>tmp6</i>
load <i>tmp1</i> , <i>la</i>						
load <i>tmp2</i> , <i>lb</i>	■					
load <i>tmp3</i> , <i>lc</i>	■	■				
ADD <i>tmp4</i> , <i>tmp2</i> , <i>tmp3</i>	■	■	■			
LETI <i>tmp5</i> , <i>tmp4</i>				■		
ADD <i>tmp6</i> , <i>tmp1</i> , <i>tmp5</i>	■				■	
⋮						■

How many variables are live at the same point ?

How many registers do we need ?



# MinReg vs MaxLive : A pathological example

## Definition: MaxLive

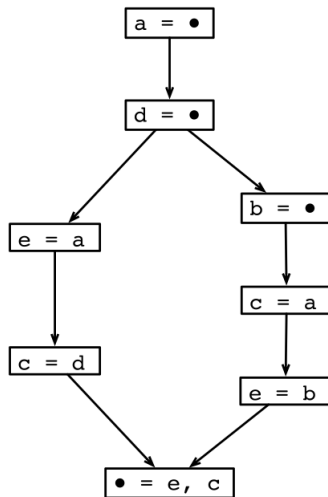
The maximum number of registers that are simultaneously alive at any program point of the program's control flow graph

## Definition: MinReg

The minimum number of registers that a program needs

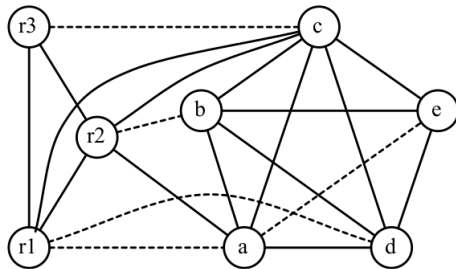
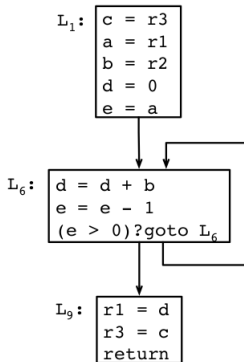
The difference is strict! There exists programs such that  $\text{MinReg} > \text{MaxLive}$

► Let's try on this example



## Running example

**Important:** in this example consider the  $r_i$  as temporary registers, like the others.

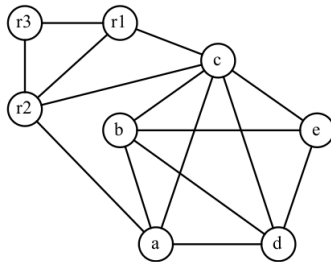
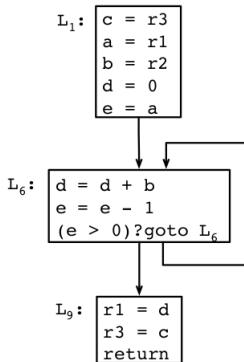


Dashed edges represent moves!



## Running example

**Important:** in this example consider the  $r_i$  as temporary registers, like the others.



Let's look at the graphs without moves first

## Kempe's simplification algorithm 1/2

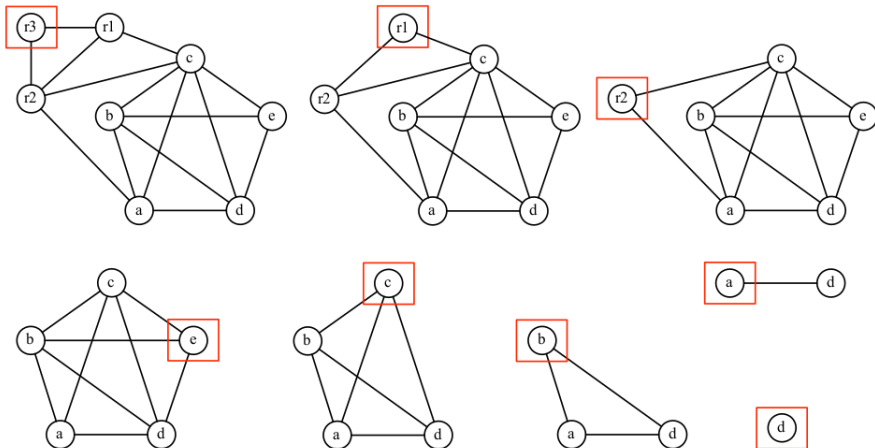
On the interference graph (without coalesce edges):

### Proposition (Kempe 1879)

Suppose the graph contains a node  $m$  with fewer than  $K$  neighbours. Then if  $G' = G \setminus \{m\}$  can be colored, then  $G$  can be colored as well.

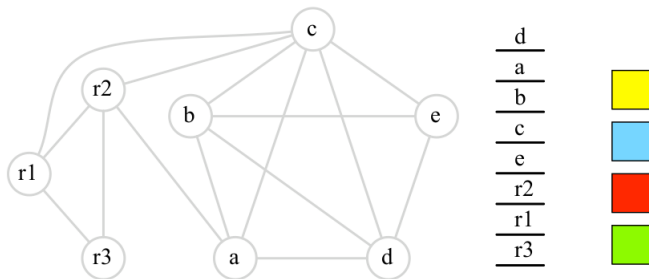
► Pick a low degree node, and remove it, and continue until remove all (the graph is  $K$ -colorable) or ...

# Kempe's simplification algorithm 2/2

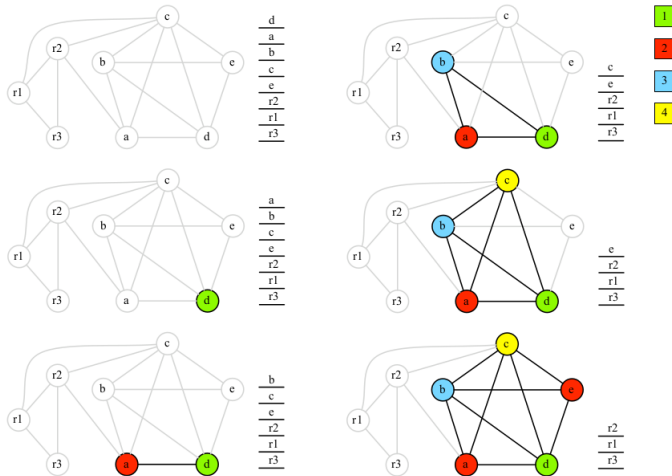


## Let's color!

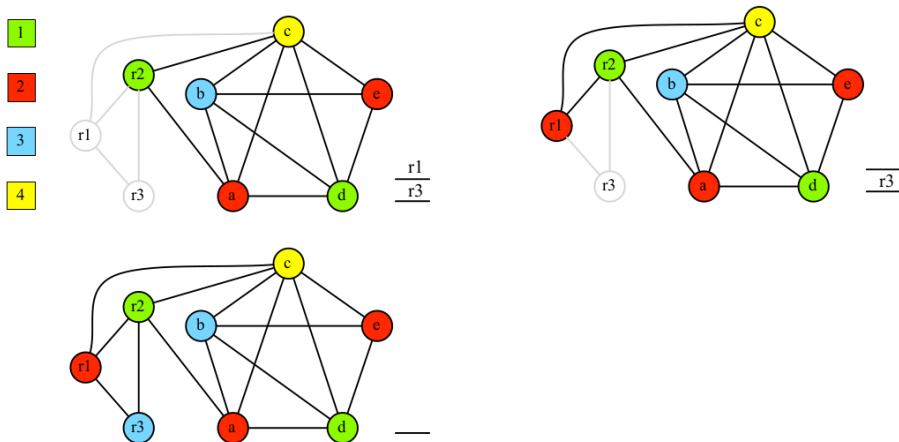
- We assign colors to the nodes greedily, in the reverse order in which nodes are removed from the graph.
- The color of the next node is the first color that is available, i.e. not used by any neighbour.



## Greedy coloring example 1/2



## Greedy coloring example 2/2



## If the graph is not colorable

Non-colored variables are named **spilled pseudo-registers**.

**Idea:** Modify the code to lower the number of simultaneously alive registers.

We call this Spilling.

There are many solutions to spill variables.

## A naive solution

- Launch the coloration algorithm with an infinite number of colors:
  - first colors are mapped to registers (used in priority by the coloring algorithm)
  - other colors are mapped to offsets in the stack, i.e. spilled to memory
- Drawback: we need a few registers to implement the spilling



## More sophisticated: Live range splitting

Invent 2 versions of the same variable (**live-range splitting**), and modify the code into:

```
ADD temp51, temp4, temp3
STORE temp51, [locationinmemory] # replace with actual location
..
LOAD temp52, [locationinmemory] #same
ADD temp6, temp52, #5
```

► But now we have to allocate these two new variables!

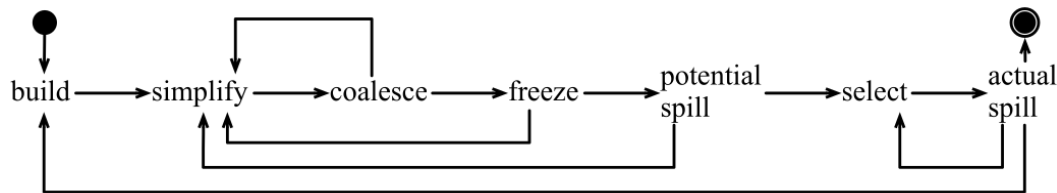
We relaunch the coloring algorithm. This is called **Iterative Register Coalescing**.

# Iterative Register Coalescing<sup>1</sup>

Two new optimizations to improve register allocation further

- 1 Register coalescing
- 2 Clever spilling

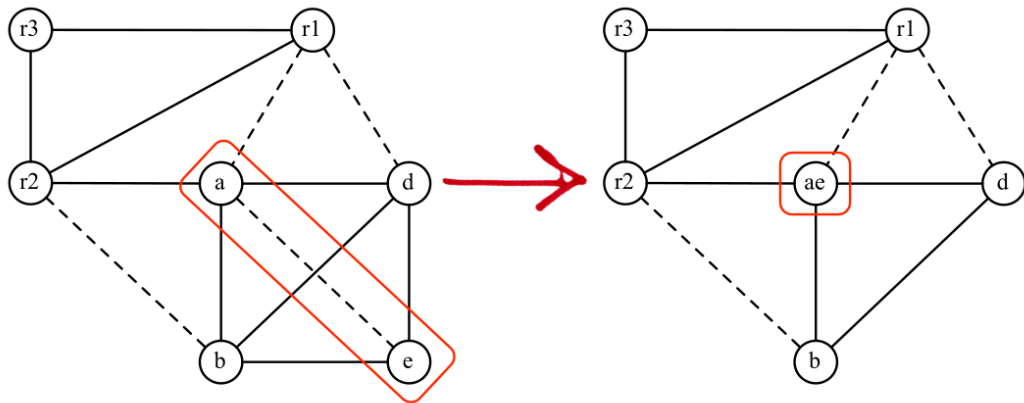
An iterative algorithm with many steps:



<sup>1</sup>Iterated Register Coalescing, TOPLAS (1996)

## Iterative Register Coalescing – Coalescing

**Coalescing** consists of collapsing two move related nodes together



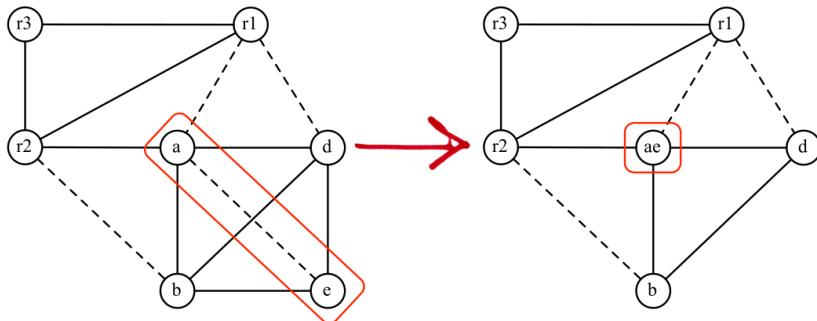
Which variables can be coalesced without causing spills ?

# Iterative Register Coalescing – Coalescing

Two heuristics for coalescing safely:

**Briggs** Nodes  $a$  and  $b$  can be coalesced if the resulting node  $ab$  will have fewer than  $K$  neighbors of high degree (i.e.,  $degree \geq K$  edges)

**George** Nodes  $a$  and  $b$  can be coalesced if, for every neighbor  $t$  of  $a$ , either  $t$  already interferes with  $b$ , or  $t$  is of low degree.



## Iterative Register Coalescing – Spilling

- ▶ How to choose with variables to spill ?

This is actually really hard:

- We want to spill variables that are less used dynamically
- We only have static information

We can use a heuristic:

`SPILLCOST(v)`

`cost = 0`

`foreach definition or use in block B`

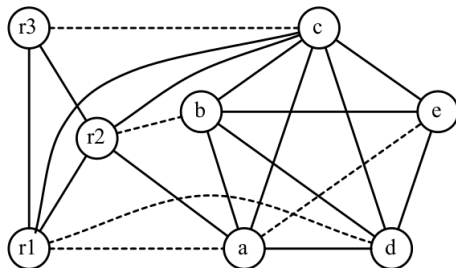
`cost += 10N/D, where`

`N` is `B`'s loop nesting factor

`D` is `v`'s degree in the interference graph

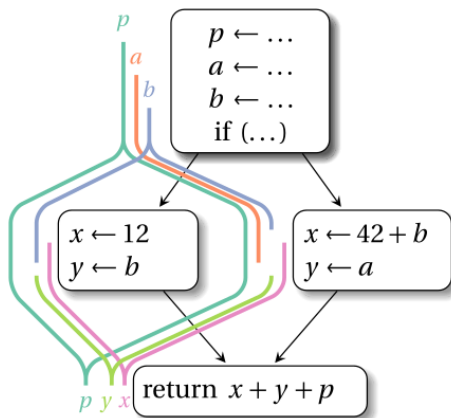
## Other Algorithms

- **Linear scan**: greedy coloring of interval graphs. (see Fernando Pereira's slides on register allocation: 18 to 35)
- **Iterative Register Coalescing** (George/Appel, TOPLAS, 1996)
- Plenty of other heuristics for spilling.

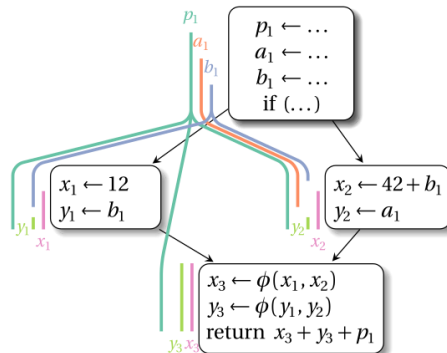


- 1 Properties in SSA: Liveness
- 2 Register Allocation with graph coloring
- 3 Register Allocation on SSA
  - Chordal graphs
  - Decoupled Register allocation
- 4 LAB : smart code Generation

# Liveness: SSA still to the rescue



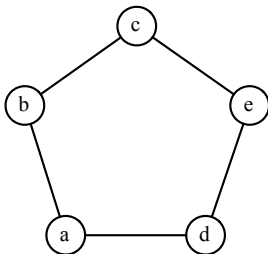
Live range on a CFG



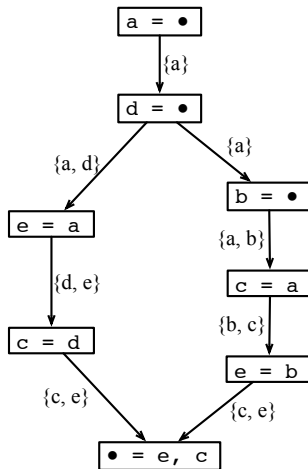
Live range with SSA



## The Example's Interference Graph

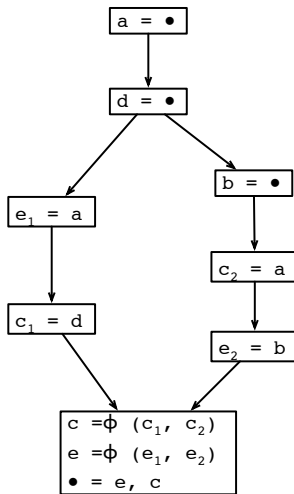


- 1) How many registers do we need, if we want to compile this program without spilling?
- 2) How this example would look like in SSA-form?



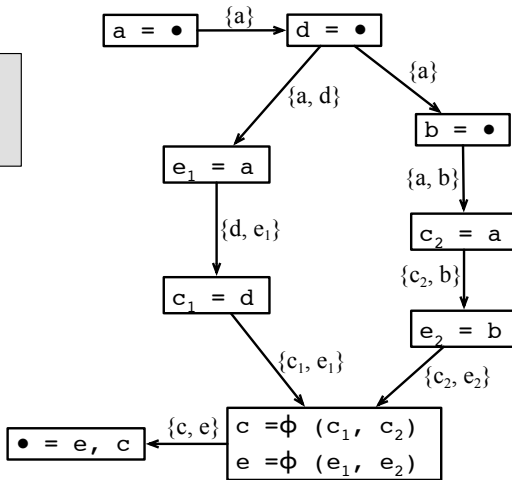
## Example in SSA form

Can you run a liveness analysis algorithm on this program?



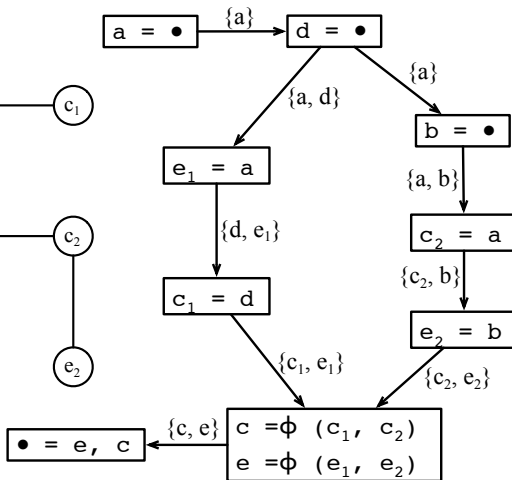
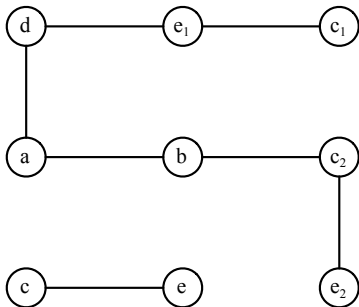
# Example in SSA form

How is the interference graph of this example?



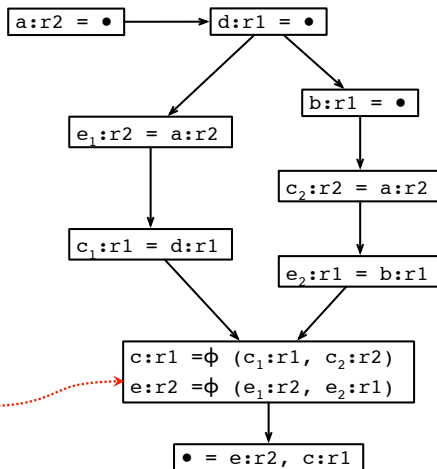
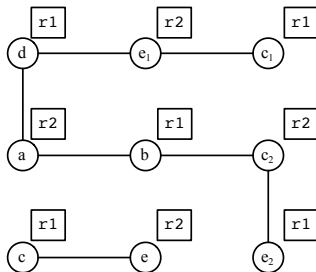
# Example in SSA form

What's the chromatic number of this graph?



# MinReg = MaxLive

This result is no coincidence.  
We shall talk more about it!



But we still have a very serious problem: how can we translate **these phi-functions** to assembly, respecting the register allocation?

## Swaps

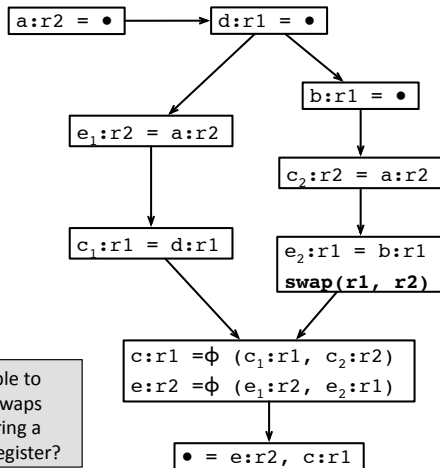
We need to copy the contents of  $e_2$  to  $e$ . Similarly, we need to copy  $c_2$  to  $c$ . But these variables have been allocated to different registers. If we have a third register to spare, we could do a swap like:

```
tmp = r1
r1 = r2
r2 = tmp
```

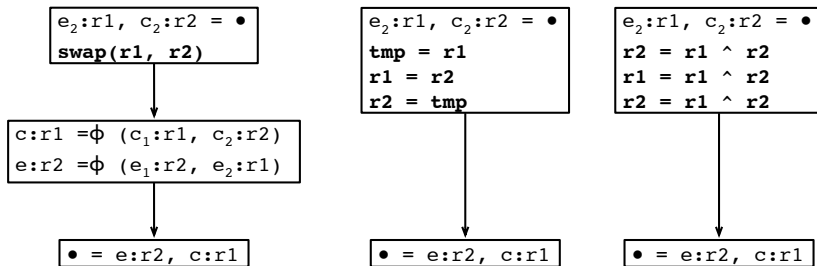
Yet, we may not have this register.

1) What is the problem of separating a register to do the swaps?

2) Is it possible to implement swaps without sparing a temporary register?



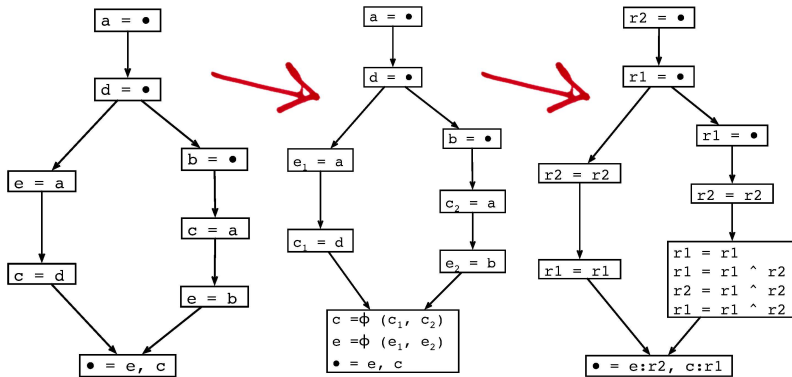
## Swaps



There are ways to implement swaps, without the need of a temporary location. One of these ways is the well-known hacking of using three xor operations to exchange two integer locations. There are other ways, though. Some architectures provide instructions to swap two registers. The x86, for instance, provides the instruction `xchg(r1, r2)`, which exchanges the contents of r1 and r2.

Can you think about other ways to swap the contents of registers?

So, in the end we get...

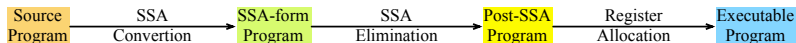




## SSA-Based Register Allocation

- SSA-based register allocation is a technique to perform register allocation in SSA-form programs.
  - Simpler algorithm.
    - Decoupling of spilling and register assignment
  - Less spilling.
    - Smaller live ranges
    - Polynomial time minimum register assignment

### Traditional Register Allocation



### SSA-Based Register Allocation



- 1 Properties in SSA: Liveness
- 2 Register Allocation with graph coloring
- 3 Register Allocation on SSA
  - Chordal graphs
  - Decoupled Register allocation
- 4 LAB : smart code Generation

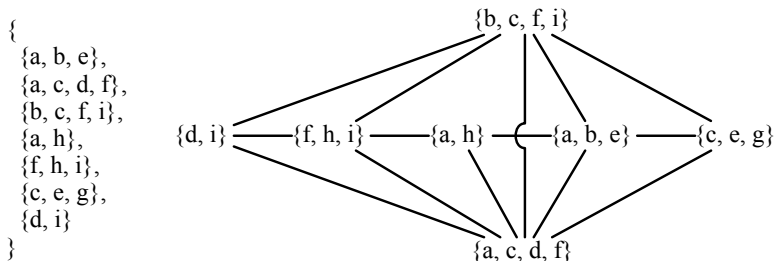
## A nice result

### Chordal graphs are P-colorable

For certain classes of graphs, graph coloring is P. This is the case for **chordal graphs** where every cycle with 4 or more edges has a chord (connects 2 vertices in the cycle but not part of the cycle).

## Intersection Graphs

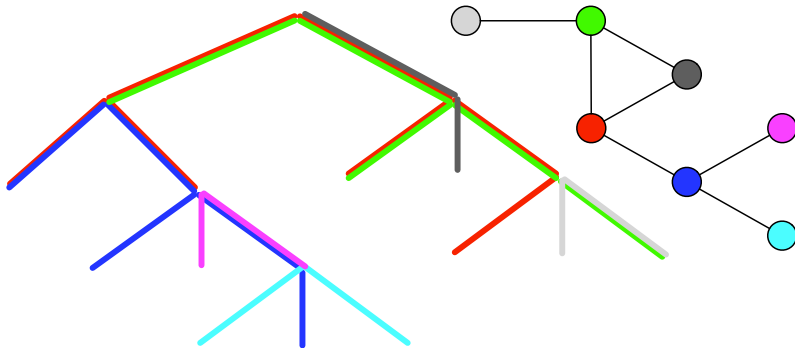
- If  $S$  is a set of sets, then we define an intersection graph  $G = (V, E)$  as follows:
  - For each set  $s \in S$ , we have a vertex  $v \in V$
  - If  $s_0, s_1 \in S$ , and  $s_0 \cap s_1 \neq \{\}$ , then we have an edge  $(v_0, v_1) \in E$



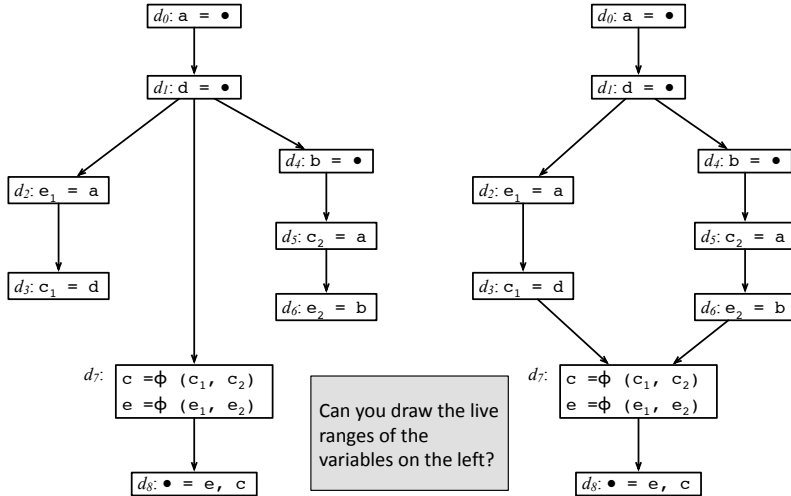
## Chordal Graphs

- The intersection graph of subtrees of a tree is a *chordal graph*.


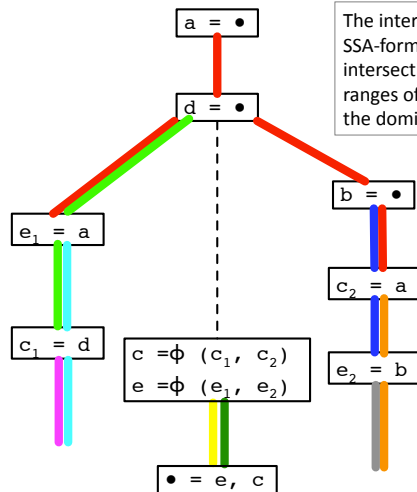
The interference graph of programs in SSA form is chordal. Any intuition on why?



# Dominance Trees



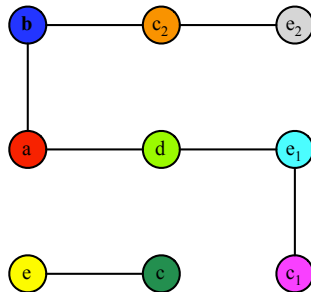
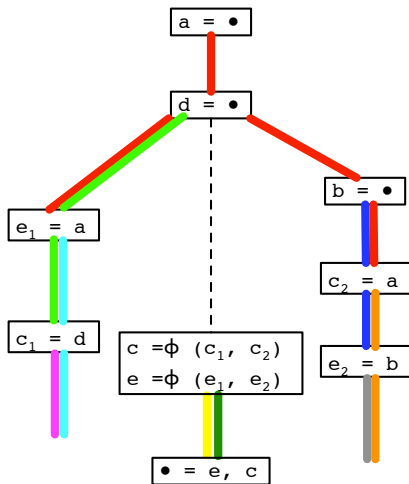
# Dominance Trees

a b  $c_1$   $c_2$  d  $e_1$   $e_2$  c e 

The interference graph of a SSA-form program is the intersection graph of the live ranges of the variables on the dominance tree<sup>♠</sup>.

<sup>♠</sup>: Allocation de Registres et Vidage en Memoire, 2005

# Intersection Graph of Live Ranges





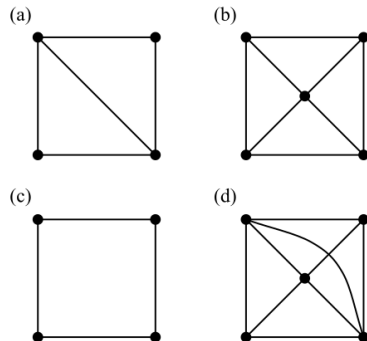
# An alternative definition of Chordality

## Definition: Induced subgraph

If  $G = (V, E)$  is a graph, then  $S = (V', E')$  is an induced subgraph of  $G$  if  $V' \subset V$ , and  $(v_i, v_j) \in E'$  if, and only if,  $(v_i, v_j) \in E$ .

## Theorem: Triangular graphs are chordal

A graph is Chordal if, and only if, it has no induced subgraphs isomorphic to  $C_n$ , where  $C_n$  is the cycle with  $n$  nodes,  $n > 3$ .



Which graphs are chordal ?

## Nice results on chordal graphs

### Theorem

The greedy coloring algorithm (with a judiciously chosen ordering) is exact on chordal graphs and takes polynomial time

- ▶ Also means computing the chromatic number of a graph is easy.

# Chordality and SSA

## Theorem

The interference graph of an SSA-form program is chordal

“Interference Graphs of Programs in SSA-form” by Sebastian Hack

- We can use this to simplify our register allocation algorithm

- 1 Properties in SSA: Liveness
- 2 Register Allocation with graph coloring
- 3 Register Allocation on SSA
  - Chordal graphs
  - Decoupled Register allocation
- 4 LAB : smart code Generation

# Liveness and Domination

In SSA, live ranges must follow the domination tree.

This gives us a nice property:

## Theorem: Max Live = Max Clique

Let  $P$  be an SSA-form program, and  $G = (V, E)$  be its interference graph. For each clique  $C = x_1, \dots, x_n$  in  $G$  there exists a program point in  $P$ , where all the variables  $c_i$  interfere.

A clique is a (sub)graph that is “fully connected”.

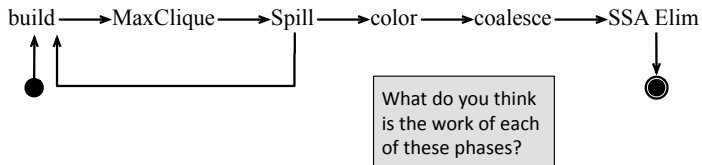


## Decoupled Spilling

- Because the maximum clique of the interference graph equals the minimum number of registers necessary to compile the program, we can lower register pressure until  $\text{MaxLive} = K$ , and just then we perform register assignment.
- This technique is called the *decoupled approach* to register allocation.
  - First we spill
  - Then we do register assignment
- As we have already seen, there exist an exact, polynomial time, algorithm to find out the chromatic number of a chordal graph.

## Decoupled Spilling

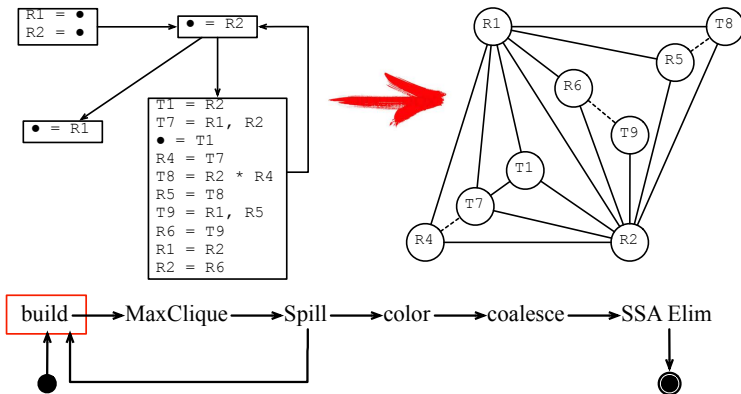
- The possibility of being able to spill, until we reach a colorable graph, gives us the opportunity to try many different algorithmic designs.
- Below we show the design used in the first register allocator based on the coloring of chordal graphs<sup>◇</sup>:



<sup>◇</sup>: Register Allocation via the Coloring of Chordal Graphs (2005)

## Build

- In the build phase we produce an interference graph out of liveness analysis.

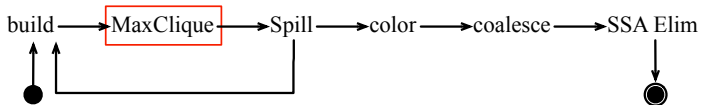
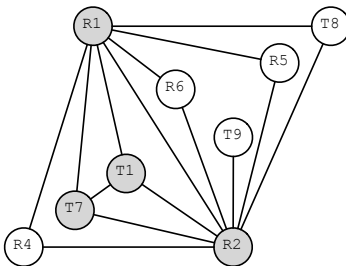




## MaxClique

In the MaxClique phase we try to find cliques with more than K nodes in the interference graph, where K is the maximum number of available registers.

$$\sigma = \mathbf{T7, R1, R2, T1, R5, R4, T8, R6, T9}$$

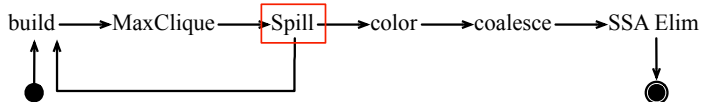
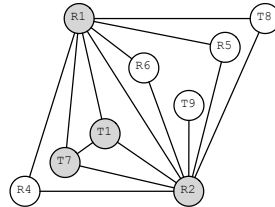
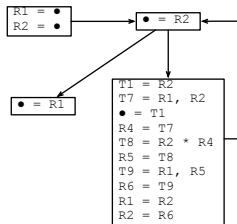


## Spill

- If we have cliques with more than K nodes, then we must choose a few of these nodes to spill.
- The problem of finding the minimum number of nodes to spill, so that we get a K colorable graph is NP-complete<sup>♠</sup>.

- How do we choose which node to spill?
- In this example, we have a clique of four nodes. If we have only three registers, which node do we spill?

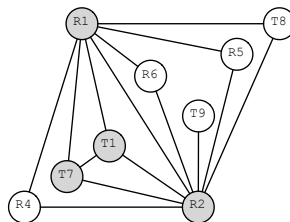
R1 = •  
R2 = •



<sup>♠</sup>: The Maximum k-Colorable Subgraph Problem for Chordal Graphs (1987)

## Spill

We can use the same formula that we have used in the design of Iterated Register Coalescing (Remember last class?) to compute spill costs. This formula takes into consideration the program, and the structure of its interference graph.



Which variable should we spill in this example?

Spill\_Cost(v)

cost = 0

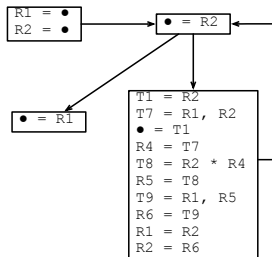
foreach definition at block B, or use at block B

cost =  $(\sum (S_B \times 10^N)) / D$ , where

$S_B$  is the number of uses and defs at B

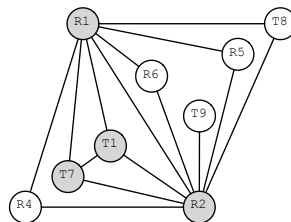
N is B's loop nesting factor

D is v's degree in the interference graph



# Spill

Node	Formula	Spilling Cost
T7	$(2 * 10) / 3$	6.66
R1	$(1 + 1 + 3 * 10) / 7$	4.57
R2	$(1 + 1 + 5 * 10) / 8$	6.5
T1	$(2 * 10)/3$	6.66



Which variable should we spill in this example?

$\text{Spill\_Cost}(v)$

cost = 0

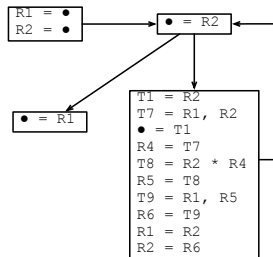
foreach definition at block B, or use at block B

cost =  $(\sum (S_B \times 10^N)) / D$ , where

$S_B$  is the number of uses and defs at B

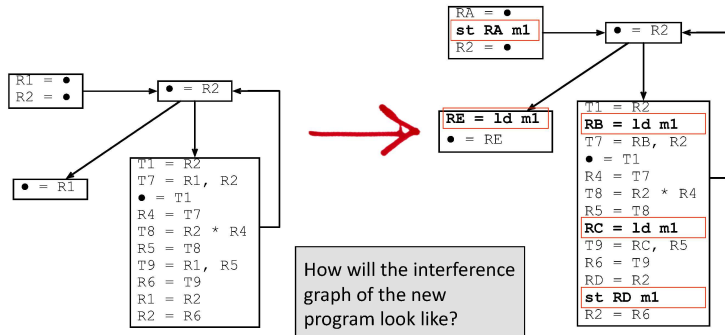
N is B's loop nesting factor

D is v's degree in the interference graph



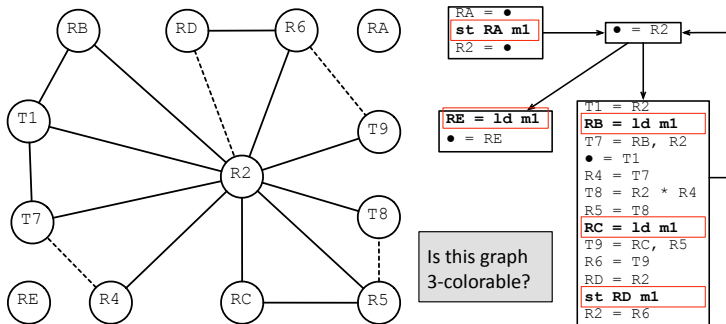
## Rebuild

- Once we spill, we must insert loads and stores in the code, to preserve the semantics of the original program.
- After scattering loads and stores around, we rebuild the interference graph.



## Rebuild

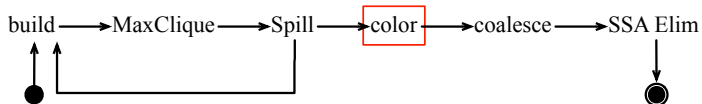
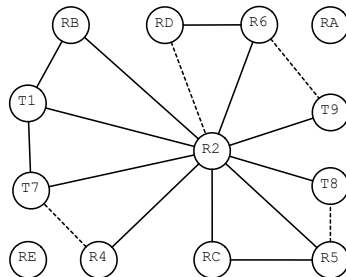
- Once we spill, we must insert loads and stores in the code, to preserve the semantics of the original program.
- After scattering loads and stores around, we rebuild the interference graph.





## Register Assignment

- Once we are down to a chordal graph whose largest clique has no more than  $K$  nodes, we are guaranteed to find a  $K$ -coloring to it.
- To find this coloring, we simply apply the greedy coloring on the simplicial elimination ordering that we obtain.

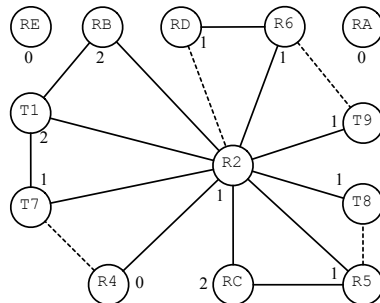


## Register Assignment

Now, assuming that we have **three colors**, what do we get when we apply greed coloring on this  $\sigma$ ?



$\sigma = R4, R2, R6, T9, T8, T7, T1, RB, R5,$   
 $RC, RD, RA, RE,$





r3

● = R2

```

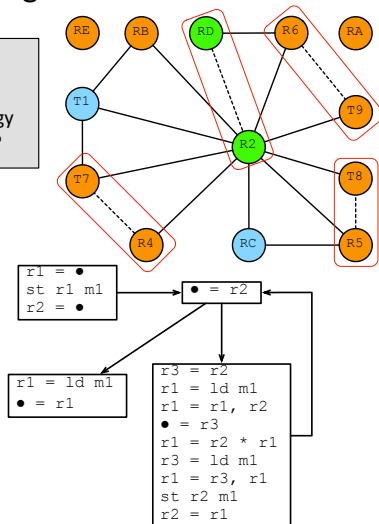
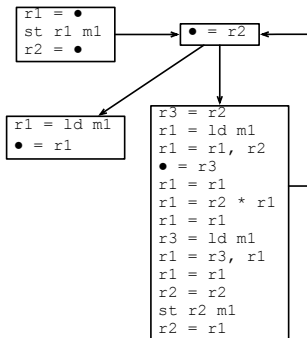
T1 = R2
RB = ld m1
T7 = RB, R2
● = T1
R4 = T7
T8 = R2 * R4
R5 = T8
RC = ld m1
T9 = RC, R5
R6 = T9
RD = R2
st RD m1
R2 = R6

```

# Register Assignment

1) We have been lucky: all the coalescible nodes have been coalesced. Actually, the greedy coloring helps coalescing a little bit. Why?

2) Can you think about a simple coalescing strategy to our algorithm?

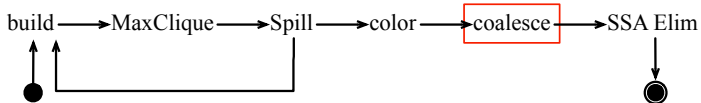


## Best Effort Coalescing

- Because we have  $K$  colors to play with, some of them may end up not being used in some neighborhood of the interference graph.
- We can use these extra colors to maximize the amount of copy instructions that we can coalesce away.

1) How likely are we to have an unused color in some neighborhood of the interference graph?

2) This coalescing technique is rather simple. Can you think about anything stronger?





## Best Effort Coalescing

### *Best Effort Coalescing*

**input:** list  $L$  of copy instructions,  $G = (V, E)$ ,  $K$

**output:**  $G'$ , the coalesced graph  $G$

$G' = G$

**for all**  $x = y \in L$  **do**

**let**  $S_x$  be the set of colors in  $N(x)$

**let**  $S_y$  be the set of colors in  $N(y)$

**if**  $\exists c, c < K, c \notin S_x \cup S_y$  **then**

**let**  $xy, xy \notin V$  be a new node **in**

            add  $xy$  to  $G'$  with color  $c$

            make  $xy$  adjacent to every  $v, v \in N(x) \cup N(y)$

            replace occurrences of  $x$  or  $y$  in  $L$  by  $xy$

            remove  $x$  from  $G'$

            remove  $y$  from  $G'$

What is the  
complexity of  
this algorithm?

- 1 Properties in SSA: Liveness
- 2 Register Allocation with graph coloring
- 3 Register Allocation on SSA
- 4 LAB : smart code Generation

# Smart Code Generation

Input: a MiniC file:

```
int n;  
n=6;
```

Output: a RISCv file:

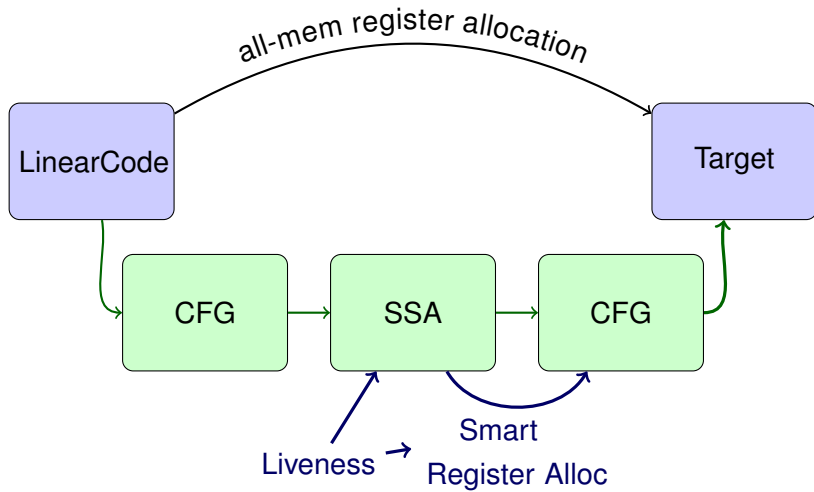
---

```
1      ;; (stat (assignment n = (expr (atom 6)) ;))  
2      li t6, 6  
3      mv t7, t6
```

---

► but with a smart allocation of registers and memory.

# Smart Code Generation



- 1 Previous labs
- 2 Lab TP5a
- 3 Lab TP5b

## Smart Code Generation – More details

- ① Implement Liveness on SSA
- ② Naive coloring/spilling strategy
  - ① Color with an infinite number of colors
  - ② Spill everything that doesn't fit
- ③ No coalescing
- ④ Naive SSA exit (we reserve SSA registers to generate moves)



# Summary

- 1 Properties in SSA: Liveness
- 2 Register Allocation with graph coloring
- 3 Register Allocation on SSA
  - Chordal graphs
  - Decoupled Register allocation
- 4 LAB : smart code Generation

Follow up: functions, program analyses, language extensions: **ENSL Only**

## Liveness and Allocation – Straight line code

We consider the following program:

```
int x,y,z,t;
x=12; y=3+x; z=4+y; t=x-y+z;
```

- ① Compute the live-out at each instruction
- ② Draw and color the interference graph
- ③ Do the register allocation with 2 registers
- ④ Generate the final code

With  $t, z, y, x \mapsto tmp\_0, tmp\_1, tmp\_2, tmp\_3$ , we obtain:

```
1 li tmp_4, 12
2 mv tmp_3, tmp_4
3 li tmp_5, 3
4 add tmp_6, tmp_5, tmp_3
5 mv tmp_2, tmp_6
6 li tmp_7, 4
7 add tmp_8, tmp_7, tmp_2
8 mv tmp_1, tmp_8
9 sub tmp_9, tmp_3, tmp_2
10 add tmp_10, tmp_9, tmp_1
11 mv tmp_0, tmp_10
```

# Liveness and Allocation – CFG

We consider the program on the right

- 1 Compute the live-out at each instruction
- 2 Draw and color the interference graph
- 3 Do the register allocation with 2 registers
- 4 Generate the final code

