Register Allocation

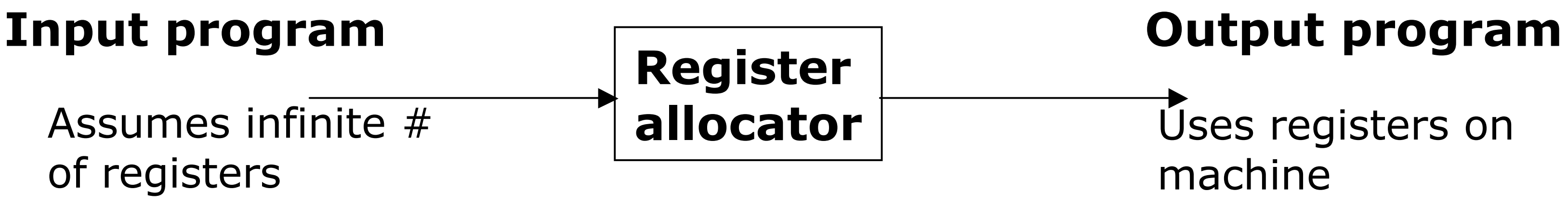
Register Allocation And Assignment

- Values in registers are easier and faster to access than memory
  - Reserve a few registers for stack pointers, base addresses etc
  - Efficiently utilize the rest of general-purpose registers
- Register allocation
  - At each program point, select a set of values to reside in registers
- Register assignment
  - Pick a specific register for each value, subject to hardware constraints
  - Register classes: not all registers are equal
- Optimal register allocation/assignment in general are NP-complete
  - Register assignment in many cases can be solved in polynomial time

```
……
    i := 0
s0: if i < 50 goto s1
     goto s2
s1: t1 := b * 2
     a := a + t1
     goto s0
S2: …
```

• Un-aliased calar variables
    i, a, b, t1 (can stay in registers)
• Need to know how variables will be used after each statement.
    Live variable analysis

The Register Allocation Problem



- At each point of execution, a program may have an arbitrary number of live variables
  - Only a subset may be kept in registers
  - If a value cannot be kept in register, it must be stored in memory and loaded again when next needed ➔ spilling value to register
- Goal: make effective use of registers
  - Minimize the number of loads and stores for spilling
- Register-to-register model
  - Early translation stores all values in registers; select values to spill to memory later
- Memory-to-memory model
  - Early translation stores all values in memory; promote values to register later
  - Must decide which values do not require memory storage

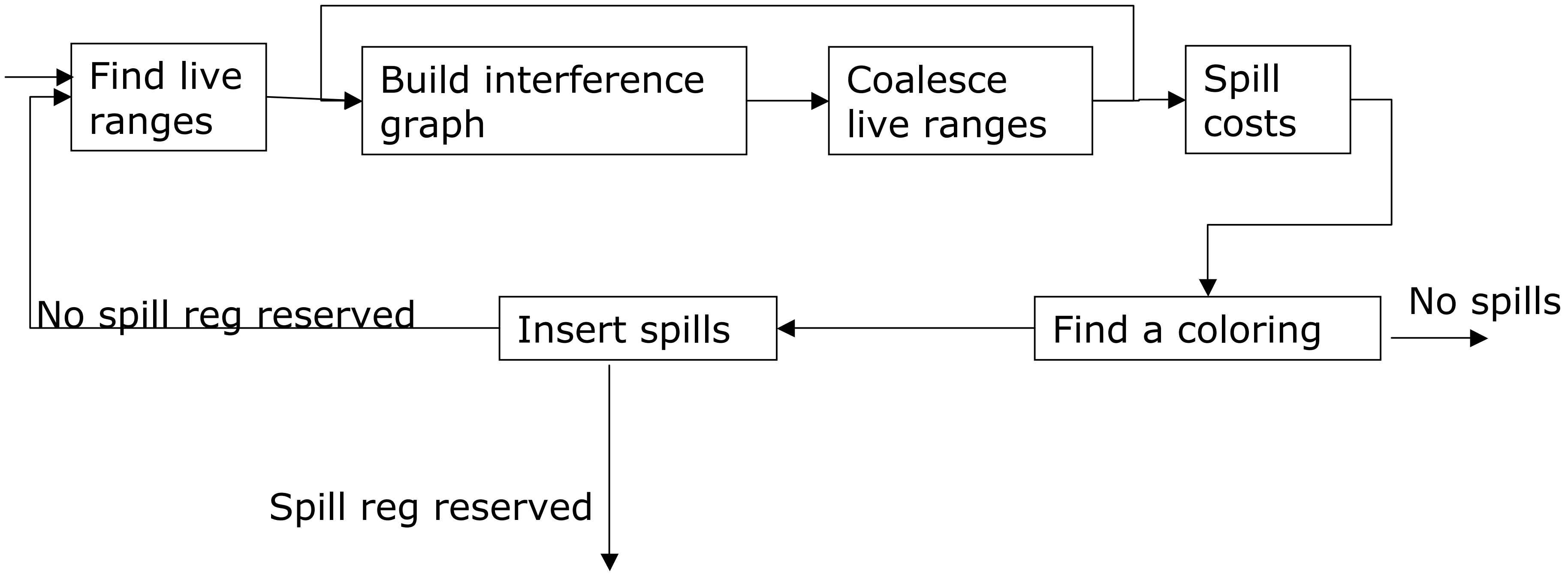
Local Register Allocation

- Allocating registers for a single basic block
  - Assumes register-to-register memory model
    - Input program assumes infinite # of registers
  - Assume all registers on target machine are equivalent
- Two approaches
  - Top-down: count the number of references to each value
    - the most heavily used values should reside in registers
    - Weakness: dedicate a register to a value for entire duration of the block
  - Bottom-up: spill the value that is needed the latest
    - For each variable use, compute the distance of its next use
    - process each instruction in evaluation order; when running out of registers, spill the value whose next use is farthest in the future
    - Produces excellent result in many cases
    - Not optimal: not all spilling takes the same number of cycles
      - Clean vs. dirty spill: has the variable been modified?

Global Register Allocation

- Allocate registers across basic block boundaries
  - Compute the live range of each variable
    - The collection of instructions that variables are alive
    - Global live variable (dataflow) analysis
  - Allocate registers to live ranges of variables
    - Rename variables so that distinct live ranges map to distinct names
    - Based on reaching definition analysis of variables
- Build an interference graph: overlapping live ranges cannot share a register
  - Nodes: live ranges of variables
  - Put an edge between (n1,n2) if their live ranges overlap
- Graph-Coloring Based Allocation
  - Assign a color (register) to each node of interference graph
  - The source and sink of each edge must have different colors
  - NP complete --- compilers must find fast approximations

A Global Register Allocator



Global Graph-coloring Register Allocation

- Build interference graph
  - Split live ranges: disjoint def-use groups of a single variable
  - Coalesce live range ➔ eliminate register copies
    - MOV LRi => LRj can be coalesced if they do not otherwise interfere
- Rank all live ranges according to their spilling cost
  - Minimize the spilling cost vs. maximize the # of uses
- Solve the k-coloring problem ---- NP complete
  - Remove all the unconstrained nodes (with <= k neighbors)
    - These nodes can always be colored
  - At each step, try color the current live range Ri with top priority
  - When no register remains, pick live ranges to split or spill
    - Spill: insert a store after every def and a load before every use
    - Split: break a live range into smaller but nontrivial pieces
- Modify interference graph and try to color the new graph

Building Global Interference Graph

- Two live ranges interfere only if one is alive at a definition of the other
  - at each operation, add interference between target of operation and each live range that is alive after the operation
- Variable copy requires special treatment
  - With x := y, if x and y do not interfere, can merge the live ranges of x and y
    - Can allocate x and y to the same register
    - Remove register copy

```
For each live range r
   create a graph node n
For each basic block b
   LIVENOW := LIVEOUT(b)
   for each instruction in b in reverse
order: op Ra, Rb ➔ Rc
      for each live range r ∈ LIVENOW
         add graph edge (Rc, r)
      remove Rc from LIVENOW
      add Ra and Rb to LIVENOW
```

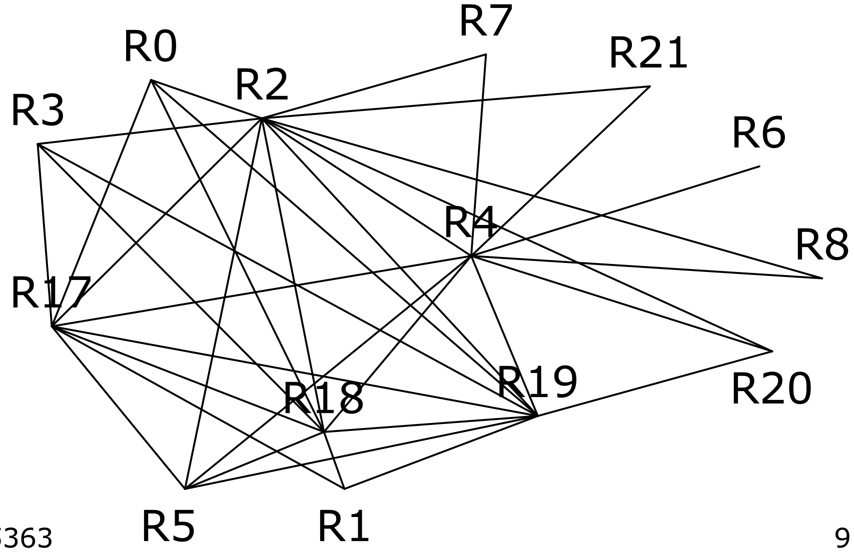
Example: Global Interference Graph

```
…=>r0,…=>r17,
 …=>r18, …=>r19
B1: loadI 1          => r1
    i2i   r1         => r2
    loadAI r0,@m     => r3
    i2i   r3         =>r4
    cmp_LT r2,r4     => r5
    cbr   r5         => B2a,B3
B2a:mult r17,r18     => r20
     add  r19, r20   => r21
     i2i   r21       => r8
B2: addI  r2, 1      => r6
    i2i   r6         => r2
    cmp_GT  r2, r4   => r7
    cbr  r7     => B3,B2
B3: return
```

	UEvar	Varkill	LiveOut	LiveOut
B1	r0	r2,r3, r4,r5	∅	r2,r4,r17 r18,r19
B2a	r17 r18 r19	r20,r21 r8	∅	r2,r4
B2	r2,r4	r6,r2,r7	∅	r2,r4
B3	∅	∅	∅	∅

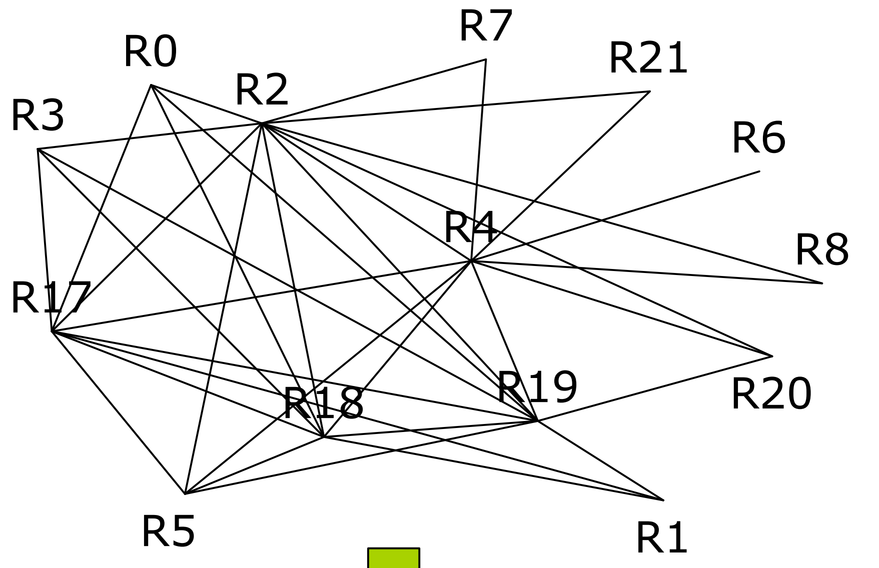
CFG:

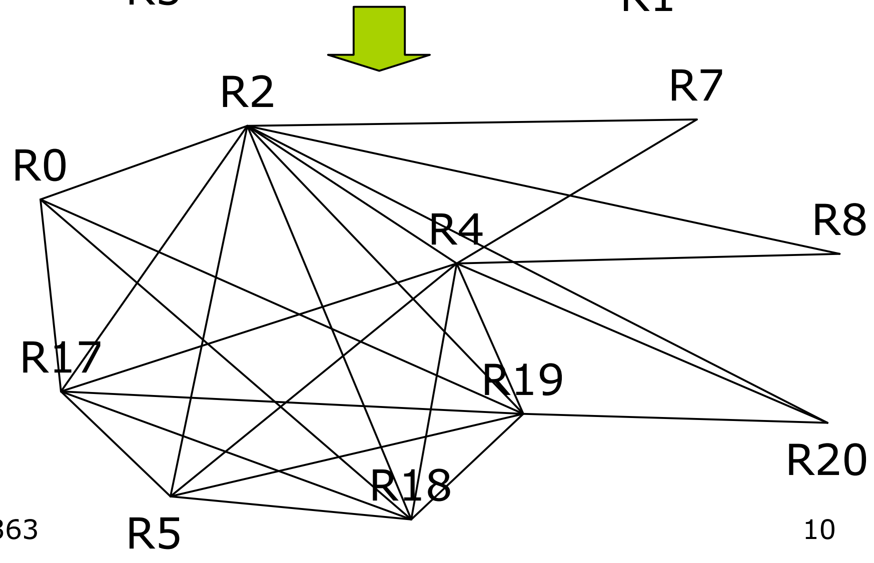
$$\begin{tikzcd} B1 \arrow[r] \arrow[rrr, bend right] & B2a \arrow[r] & B2 \arrow[r] \arrow[loop above] & B3 \end{tikzcd}$$



After Coalescing Live Ranges

```
B1: loadI 1         => r2
    loadAI r0,@m => r4
    cmp_LT r2,r4 => r5
    cbr   r5          => B2a,B3
B2a:mult r17,r18    => r20
     add  r19, r20 => r8
B2: addI  r2, 1      => r2
    cmp_GT  r2, r4 => r7
    cbr  r7     => B3,B2
B3: return
```





Merge live ranges:

r1 → r2
r3 → r4
r21→r8
r6→r2

Estimating register spilling cost

Cost = (address calculation + memory load/store)*frequency

- When insufficient registers are available, must choose registers to spill into memory
  - Choose the variables with the lowest spilling cost
  - Address calculation --- where to spill
    - Compilers can choose where to spill values
    - E.g. Register-save area of local activation record
- Spilling cost: (memory load/store cost) * (# of spills)
  - Negative spill costs
    - live ranges that contain a single load /store and no other uses
  - Infinite spill costs
    - live ranges short enough that spilling never helps
    - E.g., a use immediately following a definition
  - Frequency of basic block execution
    - Compilers annotate each block with an execution count
    - E.g., assume each loop executes 10 times, and each unpredictable branch is evaluated 50% of times

Estimating Spilling Cost

```
1   B1: loadI 1          => r2
2       loadAI r0,@m => r4
3       cmp_LT r2,r4 => r5
4       cbr   r5         => B2a,B3
5   B2a:mult r17,r18    => r20
6        add  r19, r20 => r8
7   B2: addI  r2, 1      => r2
8       cmp_GT  r2, r4 => r7
9       cbr  r7     => B3,B2
10  B3: return
```

CFG:

$$\begin{tikzcd} B1 \arrow[r] \arrow[rrr, bend right] & B2a \arrow[r] & B2 \arrow[r] \arrow[loop above] & B3 \end{tikzcd}$$

	Live ranges	spill cost
R2	r2(1),r2(3),r2(7), r2(7w),r2(8)	96
R0	r0(2)	3
R4	r4(2),r4(3),r4(8)	36
R5	r5(3),r5(4)	∞
R17	r17(5)	3
R18	r18(5)	3
R20	r20(5),r20(6)	∞
R19	r19(6)	3
R8	r8(6)	3
R7	r7(8), r7(9)	∞

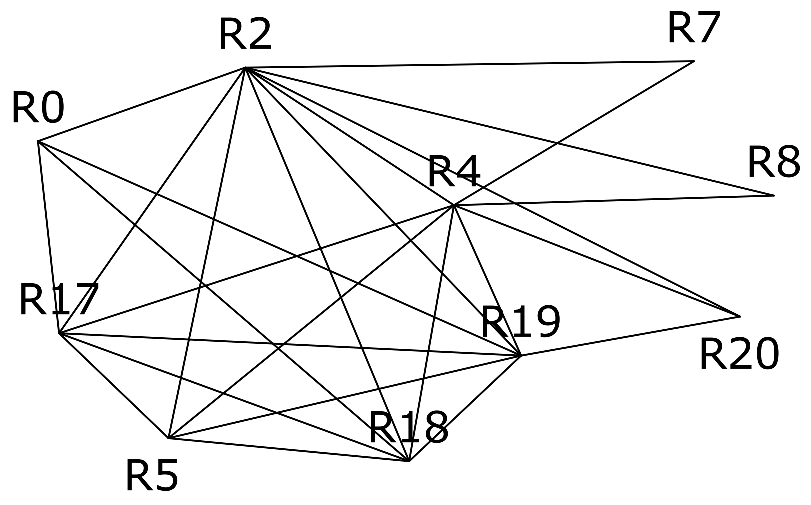
Assume address calc. has no cost
Each load/store: 3cycles
Execution frequency:
B1(1),B2a(1),B2(10),B3(1)

Ranking:
R5(∞),R20(∞),R7(∞),R2(96),R4(36),R0(3),R17(3),R18(3),R19(3),R8(3)

Graph-Coloring

- Rank all live ranges
  - Live ranges with high spilling costs are ranked higher
- Color constrained live ranges first
  - Live ranges with more than k interfering neighbors
  - Unconstrained live ranges can always be colored
- At each step, try to color the current live range Ri with top priority

  -
    ```
    if neighbors of Ri have not taken all the colors
            assign an available color (register) to Ri
    else  /*no color is available for Ri*/
            invoke spilling or splitting mechanisms
    ```



Assume 5 physical registers: P1-P5
Unconstrained nodes:
    R0,R7,R8,R20
Ordering of nodes for coloring
    R5 ➔ P1; R2 ➔ P2 ; R4 ➔ P3;
    R17➔ P4; R18 ➔ P5 ; R19➔spill
    R0 ➔ P1; R7 ➔ P1; R8 ➔ P1;
    R20➔ P1;

Result of register allocation

```
r0➔P1; r17➔P4; r18➔P5;
storeAI r19➔rarp,@m_r19
B1: loadI 1          => P2
     loadAI P1,@m  => P3
     cmp_LT P2,P3 => P1
     cbr   P1           => B2a,B3
B2a:mult P4,P5    => P1
      loadAI rarp, @m_r19 => Pr
      add  Pr, P1 => P1
B2: addI  P2, 1      => P2
     cmp_GT  P2, P3 => P1
     cbr  P1    => B3,B2
B3: return
```

R5 ➔ P1; R2 ➔ P2 ; R4 ➔ P3;
R17➔ P4; R18 ➔ P5 ; R19➔spill
R0 ➔ P1; R7 ➔ P1; R8 ➔ P1;
R20➔ P1;

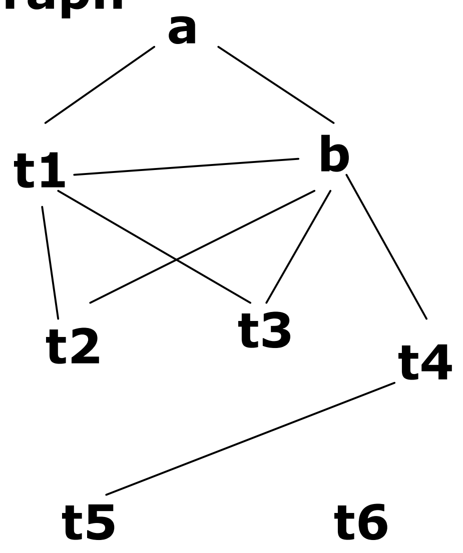
Appendix: Local Register Allocation via Graph Coloring

- Local live variable analysis
  - Set every variable ``not alive"
  - Scan statements in reverse order
    at every i: x := y op z
    - Alive(i) = current live variables
    - Set x to "not alive"
    - Set y and z to "alive"

instruction	**Alive**
	a, b
(1) t1 := a * a	**t1, a, b**
(2) t2 := a * b	**t1, b, t2**
(3) t3 := 2 * t2	**t1, t3, b**
(4) t4 := t1+t3	**t4, b**
(5) t5 := b * b	**t4, t5**
(6) t6 := t4+t5	**none**

variable	**live range**	**# of uses**
a	**(1)-(2)**	**3**
b	**(1)-(5)**	**3**
t1	**(2)-(4)**	**2**
t2	**(3)**	**1**
t3	**(4)**	**1**
t4	**(5)-(6)**	**1**
t5	**(6)**	**1**
t6	**none**	**0**

Interference graph


a
t1
b
t2
t3
t4
t5
t6