

# Register allocation

Advanced Compiler Construction  
Michel Schinz – 2021-04-15

## Register allocation

**Register allocation** consists in:

- rewriting a program that makes use of an unbounded number of virtual or pseudo-registers,
- into one that only uses physical (machine) registers.

Some virtual registers might have to be **spilled** to memory.

Register allocation is done:

- very late in the compilation process – typically only instruction scheduling comes later,
- on an IR very close to machine code.

## Setting the scene

We will do register allocation on an RTL with:

- $n$  machine registers  $R_0, \dots, R_{n-1}$  (some with non-numerical indexes like the link register  $R_{LK}$ ),
- unbounded number of virtual registers  $v_0, v_1, \dots$

Of course, virtual registers are only available before register allocation.

## Running example

Euclid's algorithm to compute greatest common divisor.

### In $L_3$

```
(defrec gcd
  (fun (a b)
    (if (= 0 b)
      a
      (gcd b (% a b)))))
```

### In RTL

```
gcd:  R3 ← done
      if R2 = 0 goto R3
      R3 ← R2
      R2 ← R1 % R2
      R1 ← R3
      R3 ← gcd
      goto R3
done: goto RLK
```

Calling conventions:

- the arguments are passed in  $R_1, R_2, \dots$
- the return address is passed in  $R_{LK}$ ,
- the return value is passed in  $R_1$ .

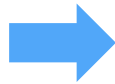
## Register allocation example

### Before register allocation

```
gcd:  v0 ← RLK
      v1 ← R1
      v2 ← R2
loop: v3 ← done
      if v2 = 0 goto v3
      v4 ← v2
      v2 ← v1 % v2
      v1 ← v4
      v5 ← loop
      goto v5
done: R1 ← v1
      goto v0
```

R<sub>1</sub>, R<sub>2</sub>: parameters  
R<sub>LK</sub>: return address

allocable  
registers:  
R<sub>1</sub>, R<sub>2</sub>, R<sub>3</sub>,  
R<sub>LK</sub>



### After register allocation

```
gcd:
loop: R3 ← done
      if R2 = 0 goto R3
      R3 ← R2
      R2 ← R1 % R2
      R1 ← R3
      R3 ← loop
      goto R3
done: goto RLK
```

Allocation:  
v0 → R<sub>LK</sub>  
v1 → R<sub>1</sub>  
v2 → R<sub>2</sub>  
v3, v4, v5 → R<sub>3</sub>

## Techniques

We will study two commonly used techniques:

1. register allocation by **graph coloring**, which:
  - produces good results,
  - is relatively slow,
  - is therefore used mostly in batch compilers,
2. **linear scan** register allocation, which:
  - produces average results,
  - is very fast,
  - is therefore used mostly in JIT compilers.

Both are **global**: they allocate registers for a whole function at a time.

## Technique #1: graph coloring

## Allocation by graph coloring

Register allocation can be reduced to graph coloring:

1. build the **interference graph**, which has:
  - one node per register – real or virtual,
  - one edge between each pair of nodes whose registers are live at the same time.
2. color the interference graph with at most K colors (K = number of available registers), so that all nodes have a different color than all their neighbors.

Problems:

- coloring is NP-complete for arbitrary graphs,
- a K-coloring might not even exist.

## Interference graph example

### Program

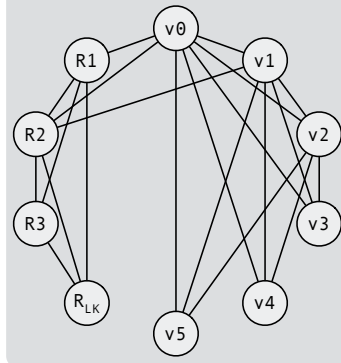
```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2=0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

### Liveness

{in} {out}

```
{R1, R2, RLK} {R1, R2, v0}
{R1, R2, v0} {R2, v0, v1}
{R2, v0, v1} {v0-v2}
{v0-v2} {v0-v3}
{v0-v3} {v0-v2}
{v0-v2} {v0-v2, v4}
{v0-v2, v4} {v0-v2, v4}
{v0-v2, v4} {v0-v2}
{v0-v2, v4} {v0-v2, v5}
{v0-v2, v5} {v0-v2}
{v0, v1} {R1, v0}
{R1, v0} {R1}
```

### Interference graph

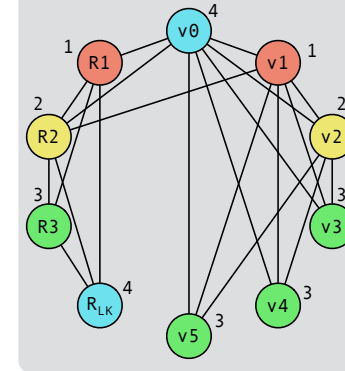


## Coloring example

### Original prog.

```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2=0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

### Colored interference graph



### Rewritten prog.

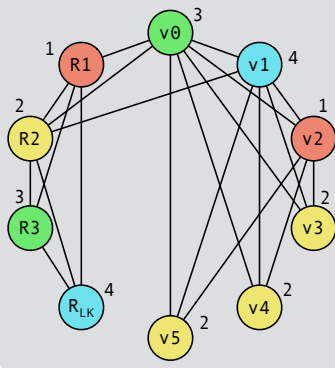
```
gcd: prog.
RLK ← RLK
R1 ← R1
R2 ← R2
loop:
  R3 ← done
  if R2=0 goto R3
  R3 ← R2
  R2 ← R1 % R2
  R1 ← R3
  R3 ← loop
  goto R3
done:
R1 ← R1
  goto RLK
```

## Coloring example (2)

### Original prog.

```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2=0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

### Colored interference graph



### Rewritten prog.

```
gcd: prog.
  R3 ← RLK
  RLK ← R1
  R1 ← R2
loop:
  R2 ← done
  if R1=0 goto R2
  R2 ← R1
  R1 ← RLK % R1
  RLK ← R2
  R2 ← loop
  goto R2
done:
  R1 ← RLK
  goto R3
```

This second coloring is also correct, but produces worse code!

## Coloring by simplification

**Coloring by simplification** is a heuristic technique to color a graph with  $K$  colors:

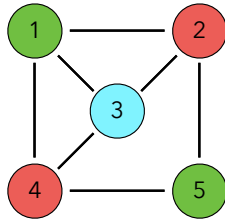
1. find a node  $n$  with less than  $K$  neighbors,
2. remove it from the graph,
3. recursively color the simplified graph,
4. color  $n$  with any color not used by its neighbors.

What if there is no node with less than  $K$  neighbors?

- a  $K$ -coloring might not exist,
- but simplification is attempted nevertheless.

## Coloring by simplification

Number of available colors (K): 3



Stack of removed nodes: 5 2 1 3

12

## Spilling

13

## (Optimistic) spilling

What if all nodes have K or more neighbors during simplification?

A node n must be chosen to be **spilled** and its value stored in memory instead of in a register:

- remove its node from the graph (assuming no interference between spilled value and other values),
- recursively color the simplified graph as usual.

Once recursive coloring is done, two cases:

1. by chance, the neighbors of n do not use all the possible colors, n is not spilled,
2. otherwise, n is really spilled.

14

## Spill costs

Which node should be spilled? Ideally one:

- whose value is not frequently used, and/or
- that interferes with many other nodes.

For that, compute the spill cost of a node n as:

$$\text{cost}(n) = (rw_0(n) + 10 \, rw_1(n) + \dots + 10^k \, rw_k(n)) / \text{degree}(n)$$

where:

- $rw_i(n)$  is the number of times the value of n is read or written in a loop of depth i,
- $\text{degree}(n)$  is the number of edges adjacent to n in the interference graph.

Then spill the node with lowest cost.

15

## Spilling of pre-colored nodes

The interference graph contains nodes corresponding to the physical registers of the machine:

- they are said to be **pre-colored**, as their color is given by the machine register they represent,
- they should never be simplified, as they cannot be spilled (they are physical registers!).

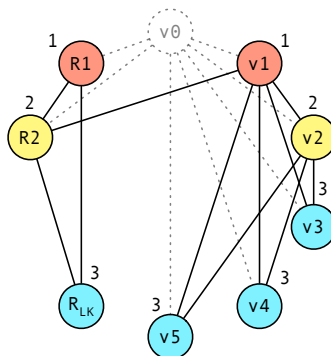
## Spilling example: costs

```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2=0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

node	rw <sub>0</sub>	rw <sub>1</sub>	deg.	cost
v <sub>0</sub>	2	0	7	0,29
v <sub>1</sub>	2	2	6	3,67
v <sub>2</sub>	1	4	6	6,83
v <sub>3</sub>	0	2	3	6,67
v <sub>4</sub>	0	2	3	6,67
v <sub>5</sub>	0	2	3	6,67

$$\text{cost} = (\text{rw}_0 + 10 \text{rw}_1) / \text{degree}$$

## Spilling example



## Consequences of spilling

After spilling, rewrite the program to:

- insert code just before the spilled value is read, to fetch it from memory,
- insert code just after the spilled value is written, to write it back to memory.

But: spilling code introduces new virtual registers, so register allocation must be redone!

In practice, 1-2 iterations are enough in almost all cases.

## Spilling code integration

### Original program

```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2 = 0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

spilling  
of v<sub>0</sub>

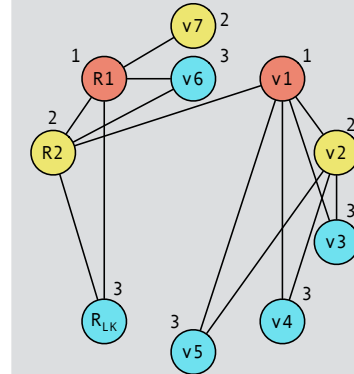


### Rewritten program

```
gcd:
  v6 ← RLK
  push v6
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2 = 0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  pop v7
  goto v7
```

## New interference graph

### Interference graph w/ spilling



### Final program

```
gcd:
  RLK ← RLK
  push RLK
  R1 ← R1
  R2 ← R2
loop:
  RLK ← done
  if R2 = 0 goto RLK
  RLK ← R2
  R2 ← R1 % R2
  R1 ← RLK
  RLK ← loop
  goto RLK
done:
  R1 ← R1
  pop R2
  goto R2
```

## Coalescing

## Coloring quality

Two valid K-colorings of an interference graph are not necessarily equivalent: one can lead to a much shorter program than the other.

Why? Because "move" instruction of the form

$v_1 \leftarrow v_2$

can be removed if  $v_1$  and  $v_2$  end up being allocated to the same register (also holds when  $v_1$  or  $v_2$  is a real register).

Goal: make this happen as often as possible.

## Coalescing

If  $v_1$  and  $v_2$  do not interfere, a move instruction of the form

$v_1 \leftarrow v_2$

can always be removed by replacing  $v_1$  and  $v_2$  by a new virtual register  $v_{1\&2}$ .

This is called **coalescing**, as the nodes of  $v_1$  and  $v_2$  in the interference graph coalesce into a single node.

26

## Coalescing issue

Coalescing is not always a good idea!

Might turn a graph that is K-colorable into one that isn't, which implies spilling.

Therefore: use conservative heuristics.

27

## Coalescing heuristics

**Briggs:** coalesce nodes  $n_1$  and  $n_2$  to  $n_{1\&2}$  iff:

$n_{1\&2}$  has less than K neighbors of significant degree (i.e. of a degree greater or equal to K),

**George:** coalesce nodes  $n_1$  and  $n_2$  to  $n_{1\&2}$  iff all neighbors of  $n_1$  either:

- already interfere with  $n_2$ , or
- are of insignificant degree.

Both heuristics are:

- safe: won't make a K-colorable graph uncolorable,
- conservative: might prevent a safe coalescing.

27

## Heuristic #1: Briggs

Briggs: coalesce nodes  $n_1$  and  $n_2$  to  $n_{1\&2}$  iff:

$n_{1\&2}$  has less than K neighbors of significant degree (i.e. of a degree  $\geq K$ ),

Rationale:

- during simplification, all the neighbors of  $n_{1\&2}$  that are of insignificant degree will be simplified;
- once they are,  $n_{1\&2}$  will have less than K neighbors and will therefore be simplifiable too.

28

## Heuristic #2: George

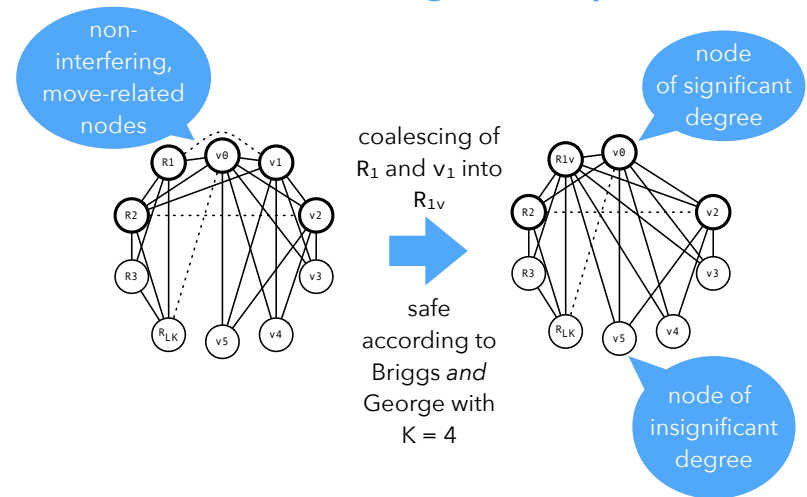
George: coalesce nodes  $n_1$  and  $n_2$  to  $n_{1\&2}$  iff all neighbors of  $n_1$  either:

- already interfere with  $n_2$ , or
- are of insignificant degree.

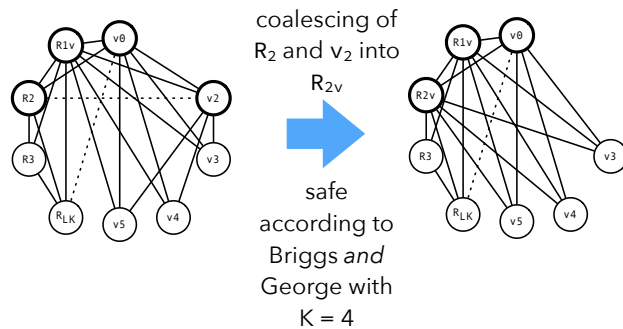
Rationale:

- the neighbors of  $n_{1\&2}$  will be:
  1. those of  $n_2$ , and
  2. the neighbors of  $n_1$  of insignificant degree,
- the latter ones will all be simplified,
- once they are, the graph will be a sub-graph of the original one.

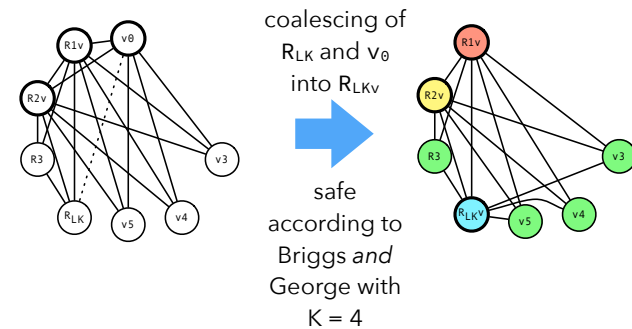
## Coalescing example



## Coalescing example (2)



## Coalescing example (3)



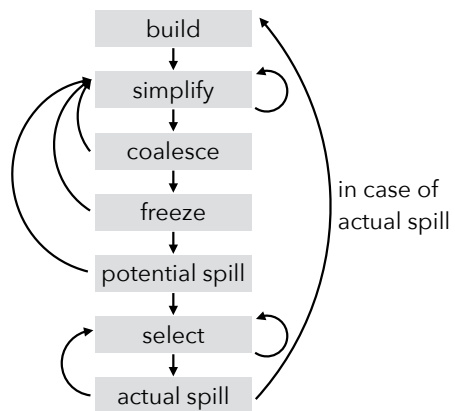
## Putting it all together

## Iterated register coalescing

Simplification and coalescing should be interleaved to get **iterated register coalescing**:

1. Interference graph nodes are partitioned in two classes: move-related or not.
2. Simplification is done on *not* move-related nodes (as move-related ones could be coalesced).
3. Conservative coalescing is performed.
4. When neither simplification nor coalescing can proceed further, some move-related nodes are **frozen** (marked as non-move-related).
5. The process is restarted at 2.

## Iterated register coalescing



## Assignment constraints

## Assignment constraints

Current assumption: a virtual register can be assigned to any free physical register.

Not always true because of **assignment constraints** due to:

- registers classes (e.g. integer vs. floating-point registers),
- instructions with arguments or result in specific registers,
- calling conventions.

A realistic register allocator has to be able to satisfy these constraints.

## Register classes

Most architectures have several register classes:

- integer vs floating-point,
- address vs data,
- etc.

To take them into account in a coloring-based allocator:

introduce artificial interferences between a node and all pre-colored nodes corresponding to registers to which it *cannot* be allocated.

## Calling conventions

How to deal with the fact that calling conventions pass arguments in specific registers?

At function entry, copy arguments to new virtual regs:

fact:

$v_1 \leftarrow R_1$  ; copy first argument to  $v_1$

Before a call, load arguments in appropriate registers:

$R_1 \leftarrow v_2$  ; load first argument from  $v_2$

CALL fact

Whenever possible, these instructions will be removed by coalescing.

## Caller/callee-saved registers

Calling conventions distinguish two kinds of registers:

- **caller-saved**: saved by the caller before a call and restored after it,
- **callee-saved**: saved by the callee at function entry and restored before function exit.

Ideally:

- virtual registers having to survive at least one call should be assigned to callee-saved registers,
- other virtual registers should be assigned to caller-saved registers.

How can this be obtained in a coloring-based allocator?

## Caller/callee-saved registers

Caller-saved registers do not survive a function call.

To model this:

Add interference edges between all virtual registers live across at least one call and (physical) caller-saved registers.

Consequence:

Virtual registers live across at least one call won't be assigned to caller-saved registers.

Therefore:

They will either be allocated to callee-saved registers, or spilled!

## Saving callee-saved registers

Callee-saved registers must be preserved by all functions, so:

- copy them to fresh temporary registers at function entry,
- restore them before exit.

## Saving callee-saved registers

For example, if  $R_8$  is callee-saved:

entry:

```
 $v_1 \leftarrow R_8$  ; save callee-saved  $R_8$  in  $v_1$   
... ; function body  
 $R_8 \leftarrow v_1$  ; restore callee-saved  $R_8$   
goto  $R_{LK}$ 
```

If register pressure is low:

- $R_8$  and  $v_1$  will be coalesced, and
- the two move instructions will be removed.

If register pressure is high:

- $v_1$  will be spilled, making  $R_8$  available in the function (e.g. to store a virtual register live across a call).

## Technique #2: linear scan

## Linear scan

The basic linear scan technique is very simple:

- the program is linearized – i.e. represented as a linear sequence of instructions, not as a graph,
- a unique live range is computed for every variable, going from the first to the last instruction during which it is live,
- registers are allocated by iterating over the intervals sorted by increasing starting point: each time an interval starts, the next free register is allocated to it, and each time an interval ends, its register is freed,
- if no register is available, the active range ending last is chosen to have its variable spilled.

## Linear scan example

Linearized version of GCD computation:

### Program

```

1 gcd:  v0 ← RLK
2      v1 ← R1
3      v2 ← R2
4 loop: v3 ← done
5      if v2=0 goto v3
6      v4 ← v2
7      v2 ← v1 % v2
8      v1 ← v4
9      v5 ← loop
10     goto v5
11 done: R1 ← v1
12     goto v0
    
```

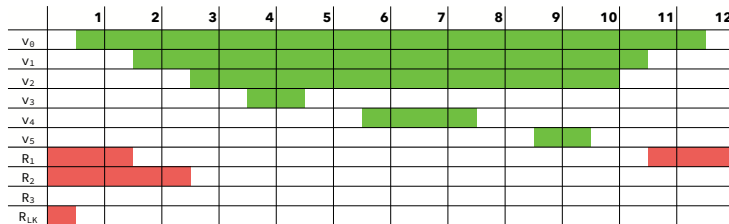
### Live ranges

$v_0$ : [1<sup>+</sup>,12<sup>-</sup>]  
 $v_1$ : [2<sup>+</sup>,11<sup>-</sup>]  
 $v_2$ : [3<sup>+</sup>,10<sup>+</sup>]  
 $v_3$ : [4<sup>+</sup>,5<sup>-</sup>]  
 $v_4$ : [6<sup>+</sup>,8<sup>-</sup>]  
 $v_5$ : [9<sup>+</sup>,10<sup>-</sup>]

Notation:

$i^+$  entry of instr.  $i$   
 $i^-$  exit of instr.  $i$

## Linear scan example (4 r.)

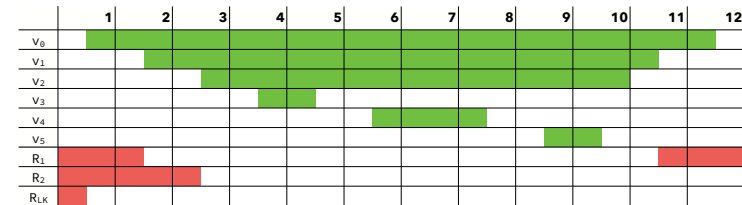


### time active intervals

1 <sup>+</sup> [1 <sup>+</sup> ,12 <sup>-</sup> ]	$v_0 \rightarrow R_3$
2 <sup>+</sup> [2 <sup>+</sup> ,11 <sup>-</sup> ], [1 <sup>+</sup> ,12 <sup>-</sup> ]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1$
3 <sup>+</sup> [3 <sup>+</sup> ,10 <sup>+</sup> ], [2 <sup>+</sup> ,11 <sup>-</sup> ], [1 <sup>+</sup> ,12 <sup>-</sup> ]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2$
4 <sup>+</sup> [4 <sup>+</sup> ,5 <sup>-</sup> ], [3 <sup>+</sup> ,10 <sup>+</sup> ], [2 <sup>+</sup> ,11 <sup>-</sup> ], [1 <sup>+</sup> ,12 <sup>-</sup> ]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_{LK}$
6 <sup>+</sup> [6 <sup>+</sup> ,8 <sup>-</sup> ], [3 <sup>+</sup> ,10 <sup>+</sup> ], [2 <sup>+</sup> ,11 <sup>-</sup> ], [1 <sup>+</sup> ,12 <sup>-</sup> ]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_4 \rightarrow R_{LK}$
9 <sup>+</sup> [9 <sup>+</sup> ,10 <sup>-</sup> ], [3 <sup>+</sup> ,10 <sup>+</sup> ], [2 <sup>+</sup> ,11 <sup>-</sup> ], [1 <sup>+</sup> ,12 <sup>-</sup> ]	$v_0 \rightarrow R_3, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_5 \rightarrow R_{LK}$

Result: no spilling

## Linear scan example (3 r.)



### time active intervals

1 <sup>+</sup> [1 <sup>+</sup> ,12 <sup>-</sup> ]	$v_0 \rightarrow R_{LK}$
2 <sup>+</sup> [2 <sup>+</sup> ,11 <sup>-</sup> ], [1 <sup>+</sup> ,12 <sup>-</sup> ]	$v_0 \rightarrow R_{LK}, v_1 \rightarrow R_1$
3 <sup>+</sup> [3 <sup>+</sup> ,10 <sup>+</sup> ], [2 <sup>+</sup> ,11 <sup>-</sup> ], [1 <sup>+</sup> ,12 <sup>-</sup> ]	$v_0 \rightarrow R_{LK}, v_1 \rightarrow R_1, v_2 \rightarrow R_2$
4 <sup>+</sup> [4 <sup>+</sup> ,5 <sup>-</sup> ], [3 <sup>+</sup> ,10 <sup>+</sup> ], [2 <sup>+</sup> ,11 <sup>-</sup> ]	$v_0 \rightarrow S, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_3 \rightarrow R_{LK}$
6 <sup>+</sup> [6 <sup>+</sup> ,8 <sup>-</sup> ], [3 <sup>+</sup> ,10 <sup>+</sup> ], [2 <sup>+</sup> ,11 <sup>-</sup> ]	$v_0 \rightarrow S, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_4 \rightarrow R_{LK}$
9 <sup>+</sup> [9 <sup>+</sup> ,10 <sup>-</sup> ], [3 <sup>+</sup> ,10 <sup>+</sup> ], [2 <sup>+</sup> ,11 <sup>-</sup> ]	$v_0 \rightarrow S, v_1 \rightarrow R_1, v_2 \rightarrow R_2, v_5 \rightarrow R_{LK}$

Result:  $v_0$  is spilled *during its whole life time!*

## Linear scan improvements

The basic linear scan algorithm is very simple but still produces reasonably good code. It can be – and has been – improved in many ways:

- the liveness information about virtual registers can be described using a sequence of disjoint intervals instead of a single one,
- virtual registers can be spilled for only a part of their whole life time,
- more sophisticated heuristics can be used to select the virtual register to spill,
- etc.