Compilers for Embedded Systems

Summer Term 2022

Heiko Falk

Institute of Embedded Systems
Electrical Engineering, Computer Science and Mathematics
Hamburg University of Technology

Chapter 8

Register Allocation

Outline

Chapter Contents

Motivation

Memory Hierarchies *(☞ Chapter 7 – Scratchpad Optimizations)*

Memories are the more efficient w.r.t. run-time and energy consumption ...

– ... the smaller they are, and ...

– ... the closer they are placed to the processor.

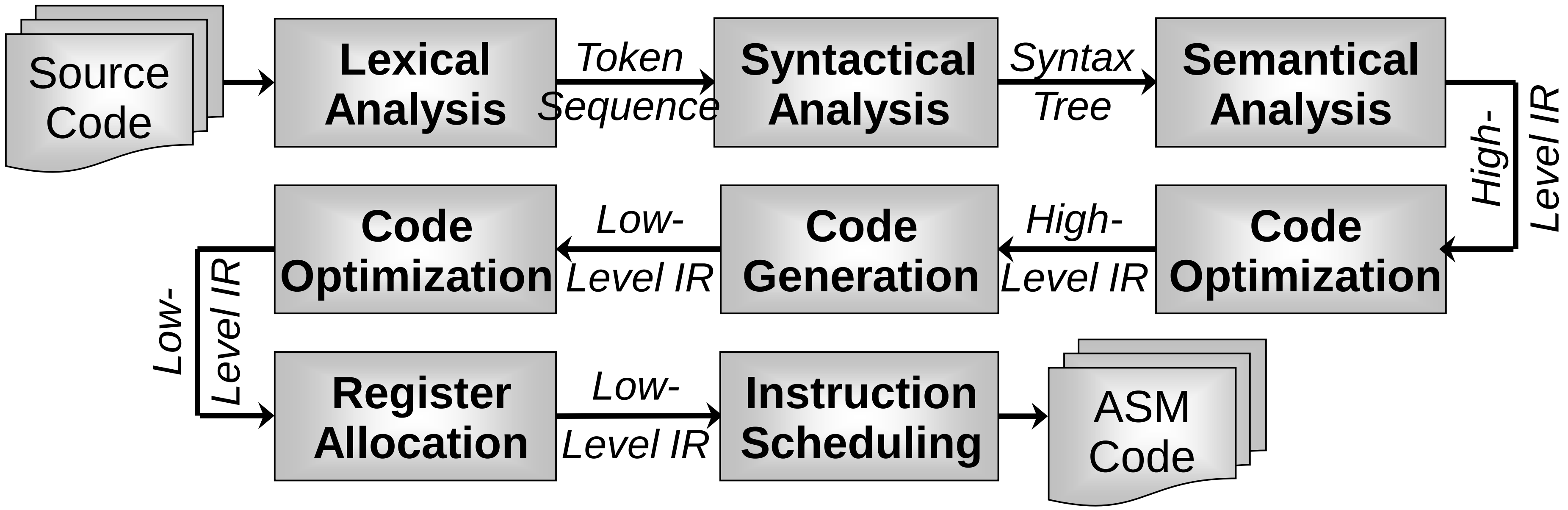
Registers

– Memory hierarchies of computers are usually mentioned in the context of hard disks, main memories or caches (e.g., in advertisements)

☞ *<u>But:</u>* Registers are those memories that, among all memories of a computer, are the smallest ones and that are placed directly *<u>inside</u>* the processor.

☞ ***Registers are by far the most efficient memories of a computer.***

Role of Register Allocation



Register Allocation

– Mapping of scalar data of an LIR to physical registers

– Best-possible exploitation of the (scarce) resource of processor registers

☞ *Is considered to be <u>the</u> most important compiler optimization ever*

Interdependencies Register Allocation ↔ Code Generation

Variant 1: ***Code Generator Produces Virtual Code***

– LIR uses infinite amount of virtual registers *(☞ chapter 3)*

☞ Register allocation has to map each individual (of the potentially many) virtual registers to a physical register.

Variant 2: ***Code Generator Produces Stack Accesses***

– Before each USE of an operand, the LIR contains a load instruction in order to fetch the operand from the stack

– Analogously: Each DEF of an LIR operand is followed by a store instruction

☞ Register allocation has to map the LIR operands to processor registers such that all these load and store instructions can mostly be removed.

☞ ***In the following: Assumption of virtual code (variant 1)***

Chapter Contents

Motivation

When is a Mapping of Virtual to Physical Registers valid?

Two virtual registers r_0 and r_1 may be mapped to the same physical register if

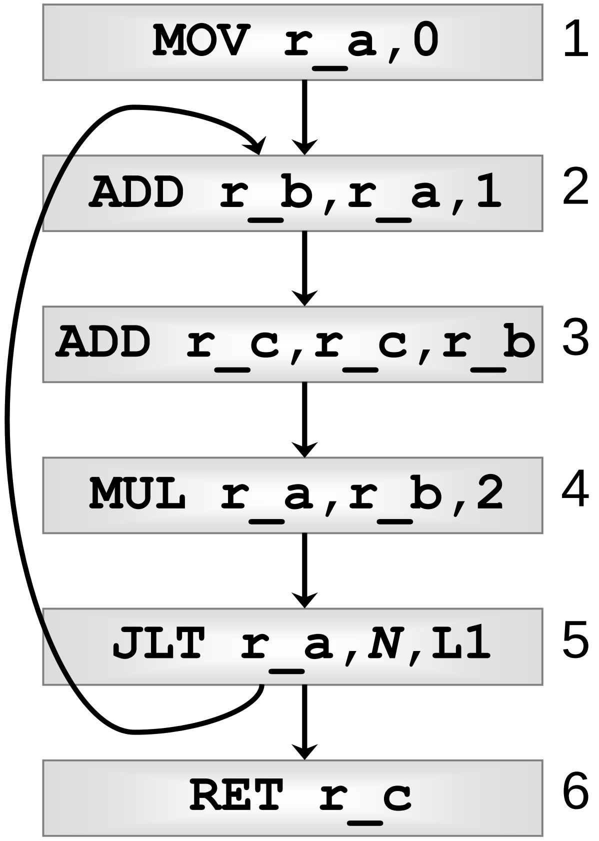
👍 r_0 and r_1 are never “in use” simultaneously.

When is a Virtual Register “in Use”, and when not?

– Life Time Analysis (LTA) determines when the life times of virtual registers start and end
– A virtual register is live if it holds a value that could eventually be used in the future
– LTA bases on the control flow graph and DEF/USE information

Example CFG

```
        MOV r_a,0;             # r_a = 0
L1:     ADD r_b,r_a,1;         # r_b = r_a+1
        ADD r_c,r_c,r_b;       # r_c = r_c+r_b
        MUL r_a,r_b,2;         # r_a = r_b*2
        JLT r_a,N,L1;          # if r_a<N goto L1
L2:     RET r_c;               # return r_c
```



- *Note:* CFG nodes here represent single machine instructions instead of basic blocks (*☞ cf. chapter 5)*
- Code makes use of three virtual registers: **`r_a`**, **`r_b`**, **`r_c`**

Example: Life Time of `r_b`

– A register is live if it holds a value that could eventually be used in the future

☞ LTA works “from the future” towards “the past”

– Last USE of **`r_b`**: Node 4

☞ **`r_b`** is live along edge (3, 4)

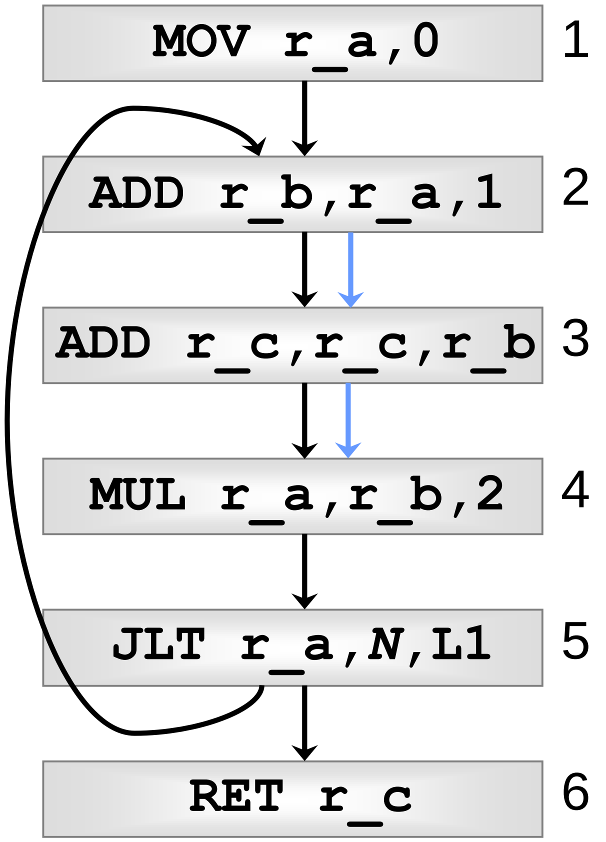
– Node 3 is not a DEF of **`r_b`**

☞ **`r_b`** is live along edge (2, 3), too

– Node 2 defines **`r_b`**:

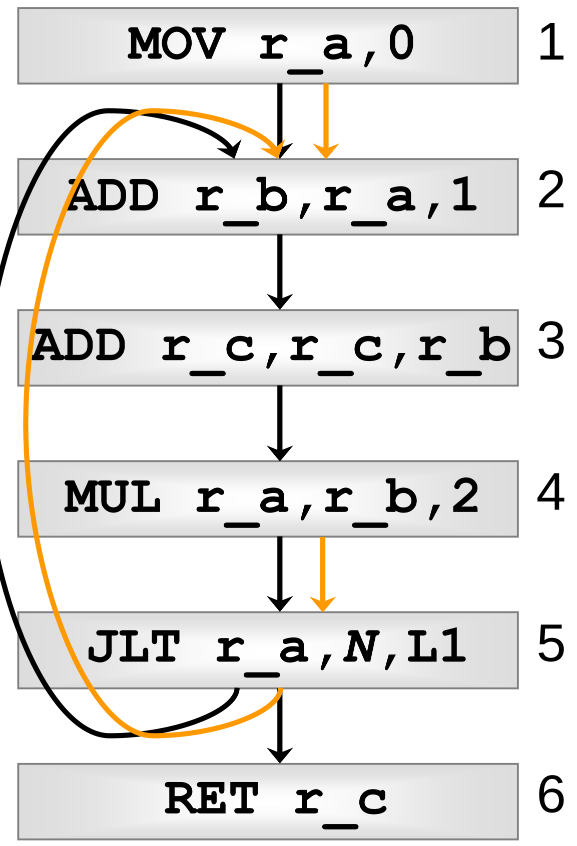
☞ Content of **`r_b`** irrelevant for node 2

☞ **`r_b`** is not live along edge (1, 2)



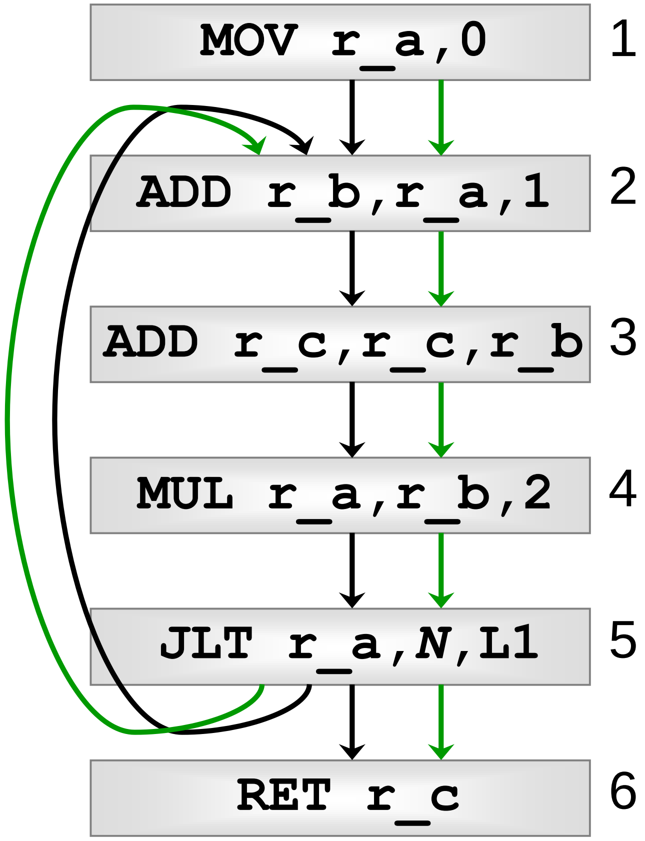
Example: Life Time of `r_a`

- Last USE of **`r_a`**: Node 5
- ☞ **`r_a`** is live along edge (4, 5)
- Node 4 defines **`r_a`**
- ☞ **`r_a`** is not alive along edge (3, 4)
- Node 2 is a USE of **`r_a`**:
- ☞ **`r_a`** is defined in node 1
- ☞ **`r_a`** is live along edge (1, 2)
- ☞ **`r_a`** is also live along edge (5, 2), since the DEF of **`r_a`** in node 4 reaches node 2 via the loop's back-edge



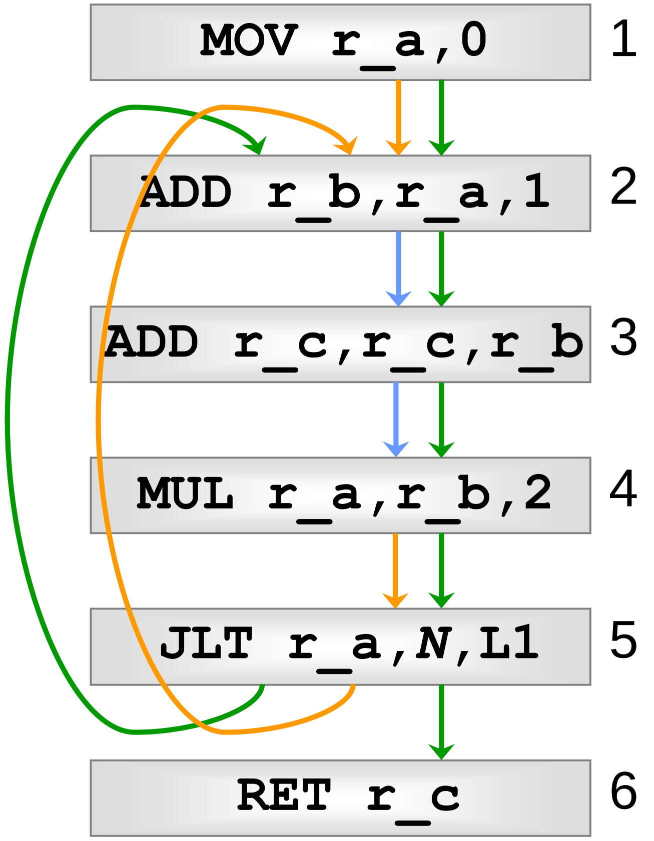
Example: Life Time of `r_c`

– Last USE of **`r_c`**: Node 6
☞ **`r_c`** is live along edge (5, 6)
☞ **`r_c`** is live along edges (4, 5) and (3, 4)
– Another USE of **`r_c`**: Node 3
☞ DEF of **`r_c`** in node 3 reaches the USE in node 3 via the loop’s back-edge
☞ **`r_c`** is also live along edges (2, 3) and (5, 2)
– Additionally:
☞ Since **`r_c`** is not properly initialized outside the loop: **`r_c`** is also live along edge (1, 2)



Example: Complete LTA

- – Life time of **`r_c`** overlaps that of **`r_a`** as well as that of **`r_b`**
- 👎 **`r_c`** must not be allocated to the same physical register as **`r_a`**
- 👎 **`r_c`** must not be allocated to the same physical register as **`r_b`**
- – Life times of **`r_a`** and **`r_b`** are disjoint
- 👍 **`r_a`** and **`r_b`** may/should share the same physical register



```
MOV r_a,0          1
ADD r_b,r_a,1      2
ADD r_c,r_c,r_b    3
MUL r_a,r_b,2      4
JLT r_a,N,L1       5
RET r_c            6
```

Definitions for LTA

- Let *v* be a node of the control flow graph
  - $pred_v$ / $succ_v$ Sets of all predecessors/successors of *v* in the CFG
  - def_v / use_v Sets of all virtual registers defined/used by *v*
- A virtual register *r* is live along a CFG edge if there is a directed path in the CFG from that edge to some use of *r* that contains no other definition of *r*.
  - Register *r* is *live-in* at a CFG node *v* if *r* is live along some incoming edge of *v*.
  - Register *r* is *live-out* at *v* if *r* is live along some outgoing edge of *v*.

Liveliness of Virtual Registers

- Let *v* be a CFG node, *r* a virtual register

– If $r \in use_v$:	*r* is live-in at *v*
– If *r* is live-in at *v*:	*r* is live-out at all $w \in pred_v$
– If *r* is live-out at *v* and $r \notin def_v$:	*r* is live-in at *v*

- Data flow equations for life time analysis:
  - in_v = Set of all registers that are live-in at *v* (out_v analogously)
  - $in_v = use_v \cup \{out_v \backslash def_v\}$
  - $out_v = \bigcup_{s \in succ_v} in_s$

Algorithm for LTA

Given: Control flow graph $G = (V, E)$ at the level of machine instructions

Approach: Iterative resolution of data flow equations from previous slide

```
– for ( <all nodes v ∈ V> )
      $in_v = out_v = \emptyset$
– do
      – for ( <all nodes v ∈ V in reverse-topological order> )
            – $in'_v = in_v; out'_v = out_v$
            – $out_v = \bigcup_{s \in succ_v} in_s$
            – $in_v = use_v \cup \{out_v \backslash def_v\}$
– while ( $(in_v \neq in'_v) || (out_v \neq out'_v)$ for some arbitrary v ∈ V )
```

Remarks

Reverse-Topological Order

- Create graph *G'* from control flow graph *G*, by reverting the sense of all edges
- Perform a depth-first search (DFS) on *G'*, starting at the source nodes of *G'*, i.e., those nodes without incoming edges. During DFS, create the post-order sequence of visited nodes
- Revert the post-order in which the nodes of *G'* are visited during the DFG traversal

Example from Slide 10

- DFS post-order of the “reverted” graph: 1, 2, 3, 4, 5, 6
- Reverse-topological order: 6, 5, 4, 3, 2, 1

Chapter Contents

Graph Coloring and Register Allocation (1)

Definition *(Graph Coloring)*

Let $G = (V, E)$ be an undirected graph, $K \in \mathbb{N}$

Graph coloring denotes the problem to assign each node $v \in V$ one color $k_v \in \{1, \ldots, K\}$ such that:

$$\forall e = \{v, w\} \in E : k_v \neq k_w$$

(No two adjacent nodes may have the same color)

Idea of a Graph-Coloring Based Register Allocation

– Create graph G with one node v per virtual register

– Color G using K colors where the considered target processor has K physical registers

– The color k_v specifies to which physical register the virtual register associated with node v is allocated

Graph Coloring and Register Allocation (2)

Definition *(Interference Graph)*

Let $R_v = \{r_1, ..., r_n\}$ be the set of all virtual registers occurring in an LIR, and let $R_p = \{R_1, ..., R_K\}$ be the set of all physical registers.

The interference graph is an undirected graph $G = (V, E)$ with

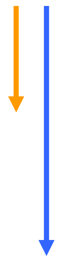
- $V = R_v \cup R_p$ and
- $e = \{v, w\} \in E$ if v and w may never share the same physical register, i.e., if v and w interfere.

Register Interference: A virtual register r_i and ...

- ... a virtual register r_j interfere if their life times overlap.
- ... a physical register R_j interfere if some LIR operation `op` uses or defines r_i, but `op` cannot address the physical register R_j.

Graph Coloring and Register Allocation (3)

Special Case: Register Moves



```
MOV r0, r1;                 /* DEF: r0, USE: r1 */
...
ADD ri, rj, r0;             /* USE: r0 */
...
MUL rk, rl, r1;             /* USE: r1 */
```

– Lite times of **`r0`** and **`r1`** overlap: Strictly speaking, an edge {**`r0`**, **`r1`**} has to be inserted into *G*.

– But: The edge {**`r0`**, **`r1`**} is not necessary, since **`r0`** and **`r1`** carry the same value. **`r0`** and **`r1`** may share the same physical register in this situation.

☞ In this particular case, *no* edge {**`r0`**, **`r1`**} is created

☞ If **`r0`** and **`r1`** get the same color *k* later, the MOV operation is redundant and can simply be removed

Creation of the Interference Graph (1)

Given

- LIR *L*
- Set $R_p = \{R_1, ..., R_K\}$ of all physical registers

Base Algorithm

```
– $G = (V, E) = (R_v \cup R_p, \emptyset)$;
– for ( <all functions $f \in L$> )
    – for ( <all basic blocks $b \in f$> )
        – for ( <all instructions $i \in b$> )
            – $live = out_i \cup def_i$ ;
            – for ( <all pairs $(r_j, r_k)$ with $r_j \in def_i$ and $r_k \in live$, $j \neq k$> )
                – if ( !isMOV( $i$ ) || ( isMOV( $i$ ) && ( $r_k \notin use_i$ ) ) )
                  $E = E \cup \{r_j, r_k\}$;
```

Creation of the Interference Graph (2)

Extension of the Base Algorithm

Processor-specific interferences have to be added to *G* explicitly after the base algorithm, e.g., if

- an LIR operation cannot access all physical registers
- conventions for calling functions and for returning from function calls enforce that parameter or return values must reside in specific physical registers (so-called *Calling Conventions*)
- the use of extended registers *(☞ chapter 2)* imposes additional restrictions on the register allocator: For a virtual extended register **`E_0`** consisting of sub-registers **`d_0`** and **`d_1`**, it has to be ensured that, e.g., **`d_0`** is always allocated to an “even” physical register, and **`d_1`** to the next following “odd” physical register

Graph Coloring by Simplification (1)

1. **Build:** Creation of the interference graph *G*
2. **Simplify:** Repeatedly remove nodes *v* from *G* that have a degree less than *K*, i.e., that have at most *K*-1 neighbors in the graph. Push these nodes on some compiler-internal stack *S*.
   (☞ Such nodes v are a priori always K-colorable, since there must always be a free color for v that is not used in the neighborhood of v.)
3. **Spill:** Step 2 stops if all remaining graph nodes have degree ≥ *K*. Then, *one* of these remaining high-degree nodes *v* is chosen, marked as *potential spill*, removed from *G* and pushed onto *S*.
   (☞ Spilling = Swapping in/out of a register v from/to the main memory if no physical register is available for v.)
4. **Repeat** Simplify and Spill until *G* = Ø.

Graph Coloring by Simplification (2)

5. **Select:** Repeatedly pop nodes *v* from the stack *S* and re-insert them into *G*. If *v* is not a potential spill, *v* *<u>must</u>* be colorable. If *v* is a potential spill, *v* *<u>may</u>* be colorable. In both cases, assign *v* a free color k_v. If a potential spill is not colorable, mark *v* as *actual spill*.
6. **Spill Code Generation:** For each actual spill *v*, insert a load operation before each USE of *v* and a store operation after each DEF of *v*.
   (☞ This way, the life time of the virtual register v is split into many tiny intervals that are likely to be colored in the next round of the algorithm.)
7. **Start Over:** If *G* still contains uncolored nodes, go to step 1.
8. **MOV Operations** with source register = target register are removed.

Coalescing (1)

Register Moves

– During creation of the interference graph, no artificial edges were inserted for MOV operations. This happened in the hope that source and target of the MOV are allocated to the same physical register.

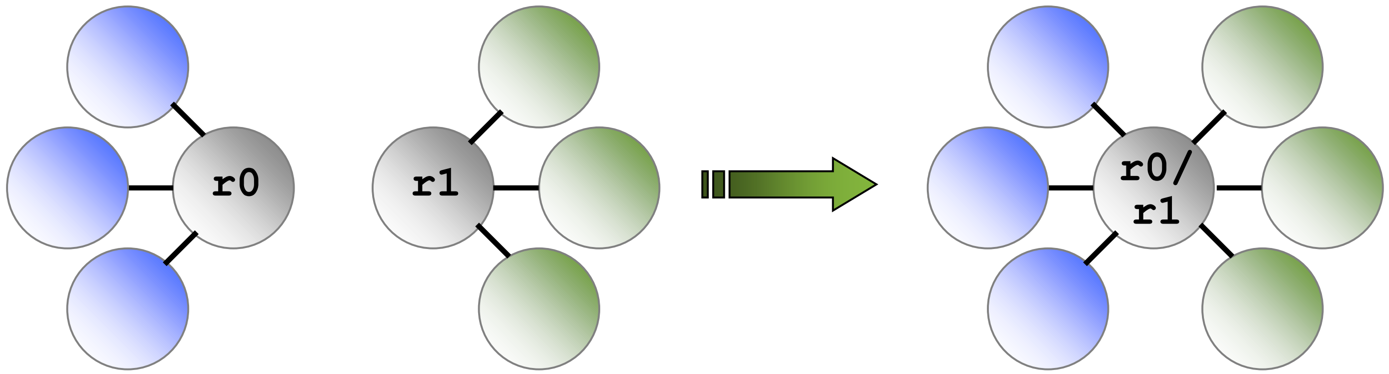
– *But:* The graph coloring algorithm from slides 25 & 26 does not actively enforce that source and target are actually allocated to the same physical register.

Register Coalescing

– For an operation **`MOV r0, r1`** with non-interfering virtual source and target operands, *coalescing* merges the nodes of **`r0`** and **`r1`** in the interference graph to one single super-node so that it is enforced by construction that **`r0`** and **`r1`** are colored with the same color.

Coalescing (2)

Effects of Coalescing `MOV r0, r1;`



– **Pros:** Lots of unnecessary register MOVes are removed, machine operations store their results directly in those registers where the results are effectively required, without the need to move results around.

– **Cons:** A coalesced node can have a higher degree than the original nodes so that an originally *K*-colorable interference graph can become uncolorable after coalescing.

Coalescing (3)

Safe Coalescing

– Coalescing is called *safe* if it can never happen that a previously *K*-colorable interference graph becomes *K*-uncolorable after coalescing.

☞ Safe coalescing thus does eventually not remove all possible register MOVes from the code.

☞ But: The remaining MOV operations are still better and more acceptable than spill code that is generated for a *K*-uncolorable interference graph.

Workflow

– Coalescing is performed after step “Simplify” and before step “Spill”. If there are opportunities for coalescing, graph coloring continues with “Simplify” afterwards.

– “Simplify” only removes such nodes *v* from *G* that are neither source nor target of a MOV operation.

Coalescing (4)

Safe Coalescing According to Briggs

– Coalesce two nodes v_0 and v_1 only if the resulting coalesced node v_0/v_1 has less than K neighbors with degree ≥ K.

– If v_0/v_1 has less than K neighbors with degree ≥ K after coalescing, then each of the two nodes v_0 and v_1 must also have less than K neighbors with degree ≥ K before coalescing.

– If the interference graph is K-colorable before coalescing, it must also be K-colorable afterwards, since v_0/v_1 will be removed during the following “Simplify” step.

☞ Coalescing according to Briggs is safe.

Coalescing (5)

Safe Coalescing According to George

– Coalesce two nodes v_0 and v_1 only if for each neighbor v_i of v_0 it holds: either v_i interferes with v_1, or v_i has degree less than K.

– Neighbors v_i with degree $< K$ also have degree $< K$ after coalescing and will thus be removed during the subsequent “Simplify” step.

– Other neighbors v_i that interfere with v_1 before coalescing have by definition two edges $\{v_\mathrm{i}, v_0\}$ and $\{v_\mathrm{i}, v_1\}$.
After coalescing, these two edges collapse to one single edge $\{v_\mathrm{i}, v_0/v_1\}$ so that the degrees of the involved nodes can only become smaller.

☞ Coalescing according to George is safe.

References

Life Time Analysis and Register Allocation

– Andrew W. Appel. *Modern compiler implementation in C*. Cambridge University Press, 2004.
ISBN 0-521-60765-5

Summary

Life Time Analysis

– Computation of the starts and ends of life times of registers
– Virtual registers may only share the same physical register if they are not simultaneously live
– Iterative solving of data flow equations

Register Allocation by Graph Coloring

– Interference graph *G* models overlapping life times of virtual registers, plus processor-specific allocation constraints
– Coloring of *G* represents a mapping of virtual to physical registers
– Graph coloring done by iterative simplification, spilling and coloring
– Safe Coalescing to remove redundant register MOVes