Lecture 2:
Register Allocation

CESE4085 Modern Computer Architecture Course
Part 2, L8.2
Carlo Galuzzi

TUDelft

LECTURE CONTENTS

What is Register Allocation?	Understand what register allocation is and why it matters.
Live Ranges and Interference	Learn how variable lifetimes lead to conflicts in register use.
Local vs Global Allocation	Explore the differences between single-block and full-function allocation.
Allocation Strategies	Study key approaches like linear scan, greedy, coloring, and others.
Spilling and Rematerialization	Examine how to handle excess live variables with memory and recomputation.
Examples and Case Studies	Analyze real-world code to see allocation challenges and solutions.
Comparison of Techniques	Compare methods in terms of quality, speed, and complexity.
AI in Register Optimization	Discover how AI can help automate and improve register allocation.

LEARNING GOALS:

By the end of this lecture, you should be able to:

- Explain the purpose and placement of register allocation in a compiler.
- Distinguish between local and global allocation strategies.
- Describe live ranges, interference, and register classes.
- Compare multiple register allocation strategies and their trade-offs.
- Analyze allocation behavior on real examples (LLVM + RISC-V).
- Discuss how AI techniques could influence future register optimizations.

EXTRA READING MATERIAL IN BRIGHSPACE

Register Allocation in the Compilation Process: Concepts, Techniques, and Implementation in LLVM for RISC-V

Structure of the Report

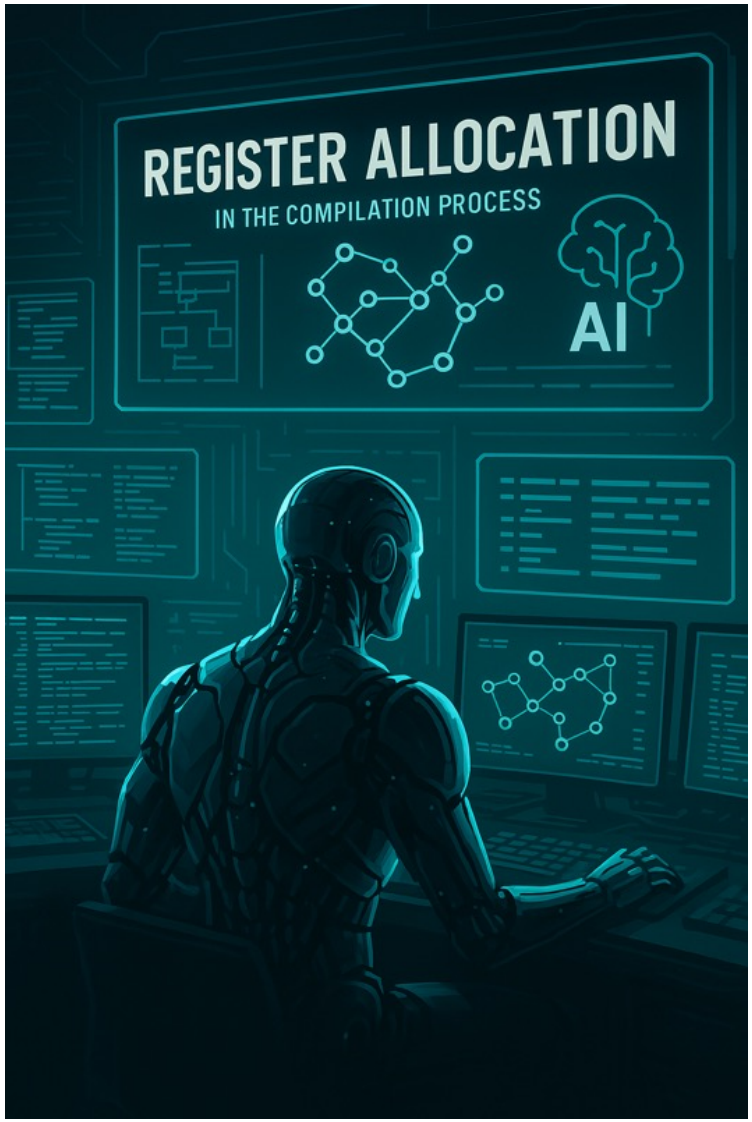
1. **Liveness and Interference**
   - A Name Space for Allocation: Live Ranges
   - Interference
   - Find Global Live Ranges
   - Build an Interference Graph
2. **Spilling and Register Classes**
   - Spill Code
   - Estimate Global Spill Costs
   - Insert Spill and Restore Code
   - Register Classes
   - Handling Overlapping Register Classes
3. **Local and Global Register Allocation**
   - Local Register Allocation
   - Renaming in the Local Allocator
   - Allocation and Assignment
   - Global Allocation via Coloring
   - Color the Graph
4. **Optimizations in Allocation**
   - Coalesce Copy Operations
   - Variations on Coalescing
   - Variations on Spilling
   - Other Forms of Live Ranges
5. **Comparison and Summary**
   - Summary and Perspective
   - Comparison of Register Allocation Techniques
6. **Appendix**
   - LLVM Commands for RISC-V Register Allocation
     - Explanation and examples of LLVM command-line tools and flags for inspecting and controlling register allocation

---

Introduction

Register allocation is a fundamental aspect of the compiler back-end, directly influencing the performance and efficiency of the generated machine code. Registers are the fastest storage locations available on a processor, yet their quantity is severely limited. This scarcity compels compilers to carefully decide which variables or intermediate results—known as temporaries—should reside in registers at any point in the program.

Unlike earlier compilation phases that focus on syntactic correctness and intermediate representation optimizations, register allocation is intimately tied to the target architecture. A poorly implemented allocator can negate many of the benefits of high-level optimizations. Therefore, the allocator must balance multiple



(AI Generated)

WHAT IS REGISTER ALLOCATION?

Register allocation is the task of assigning program variables (temporaries) to a limited set of physical registers available on a processor. **This phase is critical because registers are significantly faster than memory, but they are also extremely limited in number**.

- It takes place late in the compiler pipeline, after intermediate optimizations but before code generation.
- The compiler initially uses an unlimited number of virtual registers; register allocation resolves these into actual physical registers.
- If too many values are simultaneously live, some must be temporarily moved to memory—a process called *spilling*.
- Good allocation maximizes register reuse and minimizes memory access.
- Poor allocation can negate earlier optimizations by causing excessive spilling.

WHY IS IT HARD?

Register allocation is fundamentally **constrained by both hardware and program semantics**. The allocator must respect interference between live ranges, follow **Application Binary Interface** (ABI) and ISA constraints and make trade-offs between register pressure and performance.

- Each target architecture defines a small, fixed set of general-purpose registers.
- Temporaries may span across multiple basic blocks or functions.
- Some registers are reserved or dedicated (e.g., return address, stack pointer).
- Function calling conventions impose usage and preservation rules (callee-saved, caller-saved).
- Certain instructions mandate operands in specific registers (e.g., multiply may use specific RISC-V registers).
- Temporaries used in tight loops or frequently executed paths are especially valuable to keep in registers.

WHERE REGISTER ALLOCATION FITS IN THE PIPELINE

Register allocation occurs after high-level IR transformations and instruction selection/scheduling, and just before instruction emission. It transforms the intermediate representation into register-aware machine code.

- **Pipeline position:** Front-End → Optimizations ↔ **Instruction Selection** ↔ **Register Allocation** ↔ **Scheduling** → Emission.
- Virtual registers are used throughout earlier phases since they are unlimited.
- Physical register mapping is deferred until all major optimizations are complete.
- **Must coordinate with instruction scheduling to avoid register conflicts.**
- Decisions made during allocation have long-term effects on generated code quality.

ILLUSTRATIVE EXAMPLE

Even a small example reveals the complexity of managing live ranges under register pressure.

```
%x = add i32 %a, %b
%y = mul i32 %x, 2
%z = add i32 %y, %c
```

- %x is needed by %y, %y by %z.
- Live ranges overlap: cannot share registers.
- With 2 physical registers: at least one value must be spilled.
- Spill implies extra store/load instructions.
- The allocator must balance keeping frequently used values in registers vs. minimizing spill overhead.

TEMPORARIES AND LIVE RANGES

A **temporary** is any intermediate result produced during computation. Its **live range** defines the portion of code during which it must be kept in a register or otherwise accessible.

- Live range begins at a value's definition and ends at its last use.
- Live ranges can span several instructions or entire functions.
- Two temporaries **interfere** if their live ranges overlap.
- Interfering temporaries must not be assigned the same register.
- The compiler must build accurate live range data to inform allocation.

TRACKING LIVE RANGES

```
%x = add i32 %a, %b ; live from here
%y = mul i32 %x, 2 ; %x used here
%z = add i32 %y, %c ; %y used here
```

- `%x` is live from its definition until %y is computed.
- `%y` is live from definition to use in %z.
- `%z` is live from definition until return (not shown).
- Live ranges:
  - `%x: [def@1 → use@2]`
  - `%y: [def@2 → use@3]`
  - `%z: [def@3 → end]`
- These must be tracked to determine overlaps and avoid conflicts.

INTERFERENCE AND CONFLICT

Interference occurs when two variables are simultaneously live; assigning them the same register leads to incorrect behavior.

- Each overlapping pair must be assigned different registers.
- The set of interferences can be modeled using an *interference graph*.
- Nodes = virtual registers, edges = overlapping live ranges.
- Register allocation via graph coloring assigns registers as graph colors.

EXAMPLE: INTERFERENCE GRAPH

Based on:

```
%x = add i32 %a, %b
%y = mul i32 %x, 2
%z = add i32 %y, %c
```

We build the interference graph:

```
%x — %y
%y — %z
%x       %z
```

- `%x` and `%y` overlap → interference.
- `%y` and `%z` overlap → interference.
- `%x` and `%z` do not interfere.
- This means `%x` and `%z` *can* share a register.
- Graph coloring uses this graph to guide register assignments.

LOCAL VS GLOBAL LIVENESS

Live ranges can be local (within a single block) or global (spanning multiple blocks). Global liveness increases complexity and can introduce interference between distant values.

- **Local live ranges**:
  - Confined to one basic block.
  - Simpler to analyze and allocate.
  - Linear scan allocators often operate at this level.

- **Global live ranges**:
  - Cross basic block boundaries.
  - Require control-flow aware analysis.
  - Interference may arise through merge points or phi nodes.

- Global allocation enables more optimal code but requires more sophisticated analysis.

SPILLING: THE LAST RESORT

When registers are insufficient, some values must be temporarily stored in memory. This is called ***spilling*** and is a major source of performance degradation.

- Spilling adds store after definition and load before use.
- Memory is much slower than registers.
- Spill decision is guided by cost estimation:
  - Use frequency
  - Loop depth
  - Instruction distance
- Compiler tries to spill low-impact values.
- Excessive spills can negate prior optimizations.

SPILL CODE EXAMPLE

```
%x = add i32 %a, %b
store i32 %x, i32* %spill
%tmp = load i32, i32* %spill
%y = mul i32 %tmp, 2
```

- `%x` is spilled after computation.
- `%tmp` holds reloaded value.
- `%y` uses reloaded version.
- This pattern increases instruction count and memory pressure.

ESTIMATING SPILL COSTS

Not all spills are equal—some variables are more expensive to spill than others. Estimating spill cost is essential for strategic decision-making.

- High-cost to spill:
  - Values used in loops.
  - Frequently reused values.
  - Live across many blocks.
- Low-cost to spill:
  - Short-lived temporaries.
  - Rarely used intermediates.
- LLVM uses heuristics like use-count and loop nesting.
- Spill cost is integral to global allocation decisions.

REGISTER CLASSES

Not all registers are interchangeable. Target ISAs define ***register classes*** that constrain which registers may be used for which operations.

- Examples of RISC-V classes:
  - **Integer registers**: x1–x31
  - **Floating-point registers**: f0–f31
- Some instructions require operands from specific classes.
- ABI defines usage for argument and return value registers.
- Overlapping register classes (e.g., caller-saved and integer) must be handled carefully.

REGISTER CLASS CONSTRAINTS

Register allocation must comply with register class restrictions imposed by the architecture and ABI.

- Some registers must be preserved across calls (callee-saved).
- Others may be overwritten (caller-saved).
- Instruction encoding may only support certain registers.
- Allocation decisions must account for:
  - ISA-encoded constraints.
  - Calling conventions.
  - Reserved hardware registers.
- LLVM includes register class information in its target descriptions.
- Allocators must query class compatibility during assignment.

- **Caller-saved registers:** Must be **saved by the calling function** *before* the call if it needs their values later. The callee is **free to overwrite** them. Used for **temporary values**, often faster.
- **Callee-saved registers**: Must be **preserved by the called function**. If the callee modifies them, it must **save and restore** them. Used for **long-lived values**, more stable across calls.

LOCAL REGISTER ALLOCATION

Local register allocation considers only one basic block at a time. It is fast and simple but can lead to redundant loads and stores across blocks.

- Operates within the scope of one basic block.
- Often uses linear scan due to simplicity and speed.
- No need for global liveness analysis.
- Advantages:
  - Very fast (used in JIT compilers).
  - Works well for small functions or hot loops.
- Limitations:
  - Redundant loads/stores between blocks.
  - Cannot exploit reuse across control paths.

EXAMPLE: LOCAL ALLOCATION

```
%1 = add i32 %a, %b
%2 = sub i32 %1, 3
%3 = mul i32 %2, 2
```

- Local allocator assigns:
  - `%1 → x5`
  - `%2 → x6`
  - `%3 → x5 (reusing after %1 is dead)`
- Benefits:
  - Quick register reuse.
  - Efficient within isolated blocks.
- Drawbacks:
  - Misses opportunities across blocks.
  - Suboptimal under register pressure.

RENAMING IN LOCAL ALLOCATION

Renaming allows reusing registers when values are no longer live, improving register utilization.

- Temporaries with non-overlapping lifetimes can share registers.
- Live interval analysis enables safe renaming.
- Especially useful in tight loops.
- Reduces number of active physical registers.
- LLVM applies this via live interval splitting and coalescing.

Example:

```
%t1 = add i32 %x, %y
%t2 = mul i32 %z, %w
```

If `%t1` is dead before `%t2` starts, `%t1` and `%t2` can be assigned the same register.

ALLOCATION AND ASSIGNMENT

Allocation selects which values go into registers;
assignment chooses which specific physical register each gets.

- **Allocation** = “who gets a register.”
- **Assignment** = “which register do they get.”
- Some registers are reserved or have constraints.
- Separation allows flexible allocation strategies:
  - Greedy, coloring, etc.
- Assignment must respect:
  - Register classes
  - Interference
  - Instruction-specific restrictions

GLOBAL REGISTER ALLOCATION

Global allocation considers live ranges that span multiple basic blocks. It uses control-flow-aware analysis to make better decisions across the entire function.

- Tracks global liveness using control-flow graph.
- Identifies long-lived values that span blocks.
- Typically involves interference graph construction.
- Can reuse registers across non-overlapping paths.
- Requires more computation than local allocation.
- LLVM uses **LiveIntervals** and global interference tracking.

GRAPH COLORING FOR ALLOCATION

Graph coloring models register allocation as a **coloring problem**, where colors represent registers and no two interfering values can share the same color.

- **Interference graph**: nodes = temporaries, edges = conflicts.
- **Goal**: color the graph using ≤ number of physical registers.
- **Uncolorable graphs** → spilling.
- **Coloring heuristics** include:
  - Degree-based simplification
  - Spill cost prioritization
  - Move coalescing when safe

GRAPH COLORING EXAMPLE

```
%a = add i32 %x, %y
%b = mul i32 %a, 3
%c = sub i32 %b, 5
```

- Live ranges: `%a: [1-2], %b: [2-3], %c: [3-4]`
- Interference:

  `%a—%b, %b—%c`
- Graph:

  `%a — %b — %c`

- Minimum colors needed: 2
- If only 2 registers: allocatable without spilling

COALESCING MOVES

Redundant copy instructions like `%b = copy %a` waste instructions. **Coalescing** eliminates these, when possible, without increasing interference.

- Coalescing merges live ranges of related temporaries.
- Safe if their merged live range doesn't introduce new conflicts.
- Improves code compactness and reduces register pressure.
- LLVM applies conservative and optimistic coalescing.
- Too aggressive coalescing may lead to spills due to increased graph density.

EXAMPLE OF COALESCING

```
%1 = add i32 %x, %y
%2 = copy %1
%3 = mul i32 %2, 2
```

- `%1` and `%2` used sequentially
- Coalescing merges them → one live range
- No need for move:

```
%1 = add i32 %x, %y
%3 = mul i32 %1, 2
```

- Conditions:
  - `%1` and `%2` must not interfere.
  - Live range merge must not exceed register constraints.

LIVE RANGE SPLITTING

Large live ranges can increase interference and register pressure. **Splitting** them reduces their span and allows better reuse of registers.

- Temporaries used far apart in code can be split.
- Enables use of same register for split segments.
- Reduces spill cost by limiting range of conflict.
- LLVM performs live interval splitting during allocation.

Example:

```
%x = compute
... ; x not used
%y = use %x
```

Split at gap to reuse registers for temporaries in the gap.

REMATERIALIZATION

Instead of reloading spilled values, recompute them if the operation is cheap. This is called **rematerialization**.

- Applies to constants or pure operations.
- Reduces load/store traffic.
- **Used when recomputation is cheaper than memory access**.
- LLVM supports rematerialization during allocation.

Example:

```
%x = add i32 0, %a
store %x ; spilled
load %x
```

→ Replace with:

```
%x = add i32 0, %a ; rematerialize
```

LINEAR SCAN ALLOCATION

Linear Scan Allocation is a fast, one-pass allocation strategy suited for local allocation and JITs. Assigns registers based on the ordering of live intervals.

- Sorts live intervals by start point.
- Assigns first available register.
- Releases registers at interval end.
- Simple, efficient, but may cause suboptimal spilling.
- Used in practice for JIT compilers.

LINEAR SCAN EXAMPLE

```
%a = ...
%b = ...
%c = ...
```

Live ranges:

`%a: [1-3], %b: [2-4], %c: [4-5]`

Allocation:

- `%a → R1`
- `%b → R2`
- `%c → reuse R1 or R2` (if freed)
- **Pros**:
  - Simple implementation
  - Fast compile times
- **Cons**:
  - Misses reuse across blocks
  - May spill unnecessarily under pressure

GREEDY ALLOCATOR (LLVM DEFAULT)

LLVM's default register allocator uses a priority-based greedy approach. Balances spill costs, interference, and class constraints.

- Assigns registers to most profitable values first.
- Estimates spill cost: loop depth, use count.
- Applies live interval splitting and coalescing.
- Recalculates interference dynamically.
- Falls back to spilling when needed.
- Suitable balance between quality and speed.

CHAITIN-BRIGGS ALLOCATOR (1)

Chaitin-Briggs is a foundational approach in global register allocation. It combines graph coloring with several optimization phases to systematically allocate registers while minimizing spills.

- Works on a full-function interference graph.
- **Simplify Phase:** Remove low-degree nodes that can be safely assigned a register later.
- **Coalesce Phase:** Merges copy-related variables to eliminate redundant move instructions if safe.
- **Spill Phase:** Identifies nodes with high interference and spill cost when the graph is uncolorable.
- **Select (Color) Phase:** Reintroduces nodes and assigns colors (registers).

CHAITIN-BRIGGS ALLOCATOR (2)

Chaitin-Briggs is a foundational approach in global register allocation. It combines graph coloring with several optimization phases to systematically allocate registers while minimizing spills.

- **Advantages:**
  - Theoretically strong and intuitive.
  - Produces efficient code with good reuse.

- **Disadvantages:**
  - Slower and more complex than greedy approaches.
  - Not used in default LLVM pipelines but still informative.

PBQP ALLOCATION

PBQP (**Partitioned Boolean Quadratic Programming**) models register allocation as a numerical optimization problem.

It allows encoding of interference, preferences, costs, and constraints in a unified way, offering flexibility beyond traditional coloring.

- Each variable has a set of possible register assignments.
- Each assignment has a cost (spill cost, class restriction, etc.).
- Pairwise costs define interference and assignment preferences.
- Solves for the minimum-cost assignment satisfying constraints.
- LLVM uses this method for targets with complex requirements (e.g., DSPs, GPUs).

COMPARING ALLOCATION STRATEGIES

No single allocation strategy is optimal for all use cases. Each method balances trade-offs between compile-time speed, code quality, and architectural constraints.

Strategy	Compile Time	Spill Efficiency	Global Awareness	Used In
Linear Scan	Very Low	Low	No	JIT, embedded
Greedy (LLVM)	Medium	Medium-High	Partial	General-purpose
Chaitin-Briggs	High	High	Full	Legacy, research
PBQP	Very High	Very High	Full	Experimental

- **Linear Scan:** Best for fast codegen; weak under pressure.
- **Greedy:** Practical for everyday use; well-balanced.
- **Chaitin-Briggs:** Ideal when compile time is not critical.
- **PBQP:** Best for constrained/heterogeneous systems.

COMMON PITFALLS IN ALLOCATION

Despite sophisticated algorithms, many challenges persist in practice.

- **Excessive Spilling**: Caused by poor cost estimation or aggressive coalescing.
- **Register Starvation**: Especially in inlined functions or deep nesting.
- **Incorrect Class Assignment**: Violating ABI or ISA constraints.
- **Naive Coalescing**: Merges that increase register pressure.
- **Failure to Split**: Long live ranges not broken appropriately.
- LLVM mitigates these through:
  - Greedy + Splitting + Rematerialization
  - Late-stage validation passes

AI AND REGISTER OPTIMIZATION (1)

Modern AI techniques offer new ways to automate, adapt, and optimize compiler backends.

- **Learning-based allocation:**
  - Use supervised models trained on optimized programs.
  - Predict register pressure zones, splitting points.
- **Reinforcement learning:**
  - Learn register choices by trial and error.
  - Objective = minimize spills or execution time.
- **Data-driven heuristics:**
  - Replace hand-tuned spill costs with learned models.
- **Research Examples:**
  - Meta-compilation (e.g., MLIR with AI policy tuning)
  - Program embeddings to guide allocation phases

AI AND REGISTER OPTIMIZATION (2)

- **Opportunities:**
  - Adapt to program-specific patterns.
  - Handle irregular hardware constraints dynamically.
  - Integrate with scheduling and codegen for joint optimization.

- **Challenges:**
  - Requires large training datasets with ground truth.
  - Must maintain correctness and safety guarantees.
  - Interpretability and debugging can be difficult.

- LLVM is exploring machine-learning-guided tuning of optimization pipelines.

- **Future**: feedback loops between execution profiles and allocation behavior.

FINAL RECAP

Register allocation bridges optimized IR and machine-level code, making it critical for performance and correctness.

- Must satisfy liveness, interference, and architectural constraints.
- Multiple strategies exist, each with trade-offs.
- LLVM uses a hybrid approach, defaulting to greedy allocation.
- Key concepts:
- Temporaries and live ranges
- Interference graphs
- Spilling and rematerialization
- Allocation vs. assignment
- Final takeaway: Register allocation is a delicate balance of correctness, performance, and architectural precision.

CONCLUSION

- Thank you for your engagement throughout the lectures and lab sessions! Much appreciated ☺
- We've covered essential concepts of compilers, including theoretical foundations, LLVM integration, and future directions. I hope you had fun learning ☺

SEE YOU AT THE EXAM!
(Your notes, slides, extra material, labs.
No Smart Devices, no books)

- I will post a set of practice questions soon to help you prepare.
- Feel free to reach out if you need clarification on any topic.

GOOD LUCK AND STUDY WELL!

… AND

- If you are interested in compiler and you want to do your thesis in this context, just send me an email to c.galuzzi-2@tudelft.nl. Possible thesis' topics:

- **Machine Learning for Register Allocation**: Train models to predict spill locations or choose allocation strategies adaptively.
- **LLVM Backend for Emerging Architectures:** Develop or improve register allocators for RISC-V variants, vector processors, or custom ISAs.
- **SSA-Based Allocation and Optimization:** Investigate algorithms that work directly on SSA without full destruction during allocation.
- **Pressure-Aware Loop Optimizations:** Combine allocation with loop transformations to reduce register pressure.
- **Co-optimization of Scheduling and Allocation:** Study integrated strategies that jointly solve allocation and instruction scheduling.
- **Energy-Efficient Code Generation:** Explore register allocation's role in reducing energy consumption on embedded systems.
- **Secure and Predictable Allocation for Real-Time Systems:** Design allocators that are deterministic and verifiable.
- …

See you at the Exam!

Don’t forget to consult the Q&A blog on Brightspace!

BACKUP SLIDE: KEY TERMS

Let’s ensure all foundational terms are clearly understood.

- **Temporary**: Compiler-generated value resulting from IR operations.
- **Live Range**: Span from a temporary’s definition to its last use.
- **Interference**: When two temporaries are live at the same time and cannot share a register.
- **Spilling**: Moving a temporary from a register to memory due to lack of registers.
- **Rematerialization**: Recomputing spilled values instead of reloading from memory.
- **Register Class**: Set of registers eligible for assignment to a temporary.

Understanding these is essential to grasp allocation algorithms.

BACKUP SLIDE: CHAITIN-BRIGGS EXAMPLE

```
%a = add i32 %x, %y
%b = copy %a
%c = mul i32 %b, 3
```

- **Simplify:** If `%c` has low interference, remove and defer assignment.
- **Coalesce:** `%a` and `%b` are copy-related. Try to merge to eliminate `%b` = copy `%a`.
- **Spill:** If graph becomes too dense, mark the highest-cost node (e.g., %b) for spilling.
- **Color:** Reinsert and assign safe registers to each node.
- **Resulting benefits:**
  - Eliminates unnecessary copies.
  - Balances interference vs. spill cost.
  - More stable than linear scan under register pressure.

BACKUP SLIDE: PBQP EXAMPLE AND STRENGTHS

Three temporaries `%a`, `%b`, and `%c` with different spill costs and register compatibility.

- Cost matrix:
  - Assign %a → r1: cost 1, r2: cost 2, spill: cost 5
  - Interference with %b and %c: costly if same register assigned
- PBQP encodes:
  - Individual costs (assignment penalties)
  - Pairwise interference (matrix penalties)
- **Strengths:**
  - Very flexible: handles overlapping classes, subregisters.
  - Good for highly constrained architectures.
  - Captures global cost models more effectively than greedy strategies.
- **Limitations:**
  - Slower to solve, not scalable for very large graphs.

BACKUP SLIDE: ALLOCATION EXAMPLE

Let's examine a realistic allocation scenario without using phi nodes, highlighting live range merging and cross-path interference.

```
entry:
        %a = add i32 %x, %y
        %b = mul i32 %a, 2
        %c = sub i32 %b, 1
        br i1 %cond, label %then, label %else
then:
        %d = add i32 %c, 10
        %e = mul i32 %d, 3
        br label %exit
else:
        %f = sub i32 %c, 4
        %g = mul i32 %f, 5
        br label %exit
exit:
        %h = add i32 %e, %g
        %i = add i32 %h, 1
        ret i32 %i
```

- %c is live into both branches.
- %e and %g are live at the entry of exit.
- %h requires both values.
- Even without phi nodes, interference is significant.

BACKUP SLIDE: ALLOCATION STRATEGY COMPARISON ON EXAMPLE

Let's compare how each allocator handles the above example assuming only 3 registers are available.

Linear Scan:

- Treats paths sequentially.
- Likely to spill either %e or %g.
- Ignores control-flow merging.

Greedy:

- Prioritizes %c, %e, %g, %h, %i.
- Spills %b or %d as needed.
- Manages pressure with interval splitting.

Chaitin-Briggs:

- Models full interference.
- May coalesce %c and %d.
- Smartly spills low-cost nodes.

PBQP:

- Optimizes spill placement globally.
- Finds best cost model even under branching.
- Ideal when spilling must be precise.