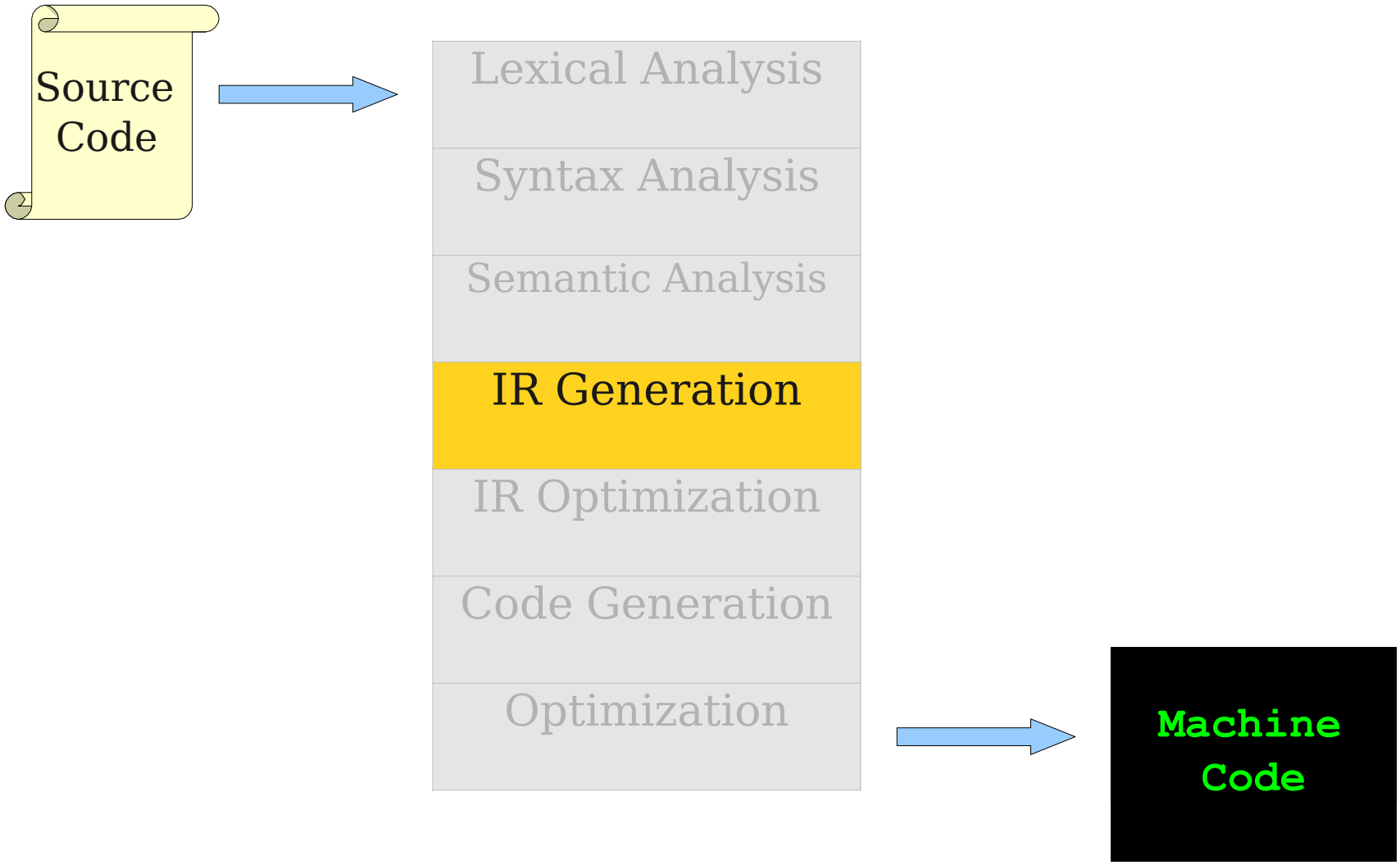
بسم الله الرحمن الرحيم

Three Address Code, Register Allocation

Compilers - 002 - S020

Three-Address Code IR

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

TAC

- TAC for expressions
- TAC for function call
- TAC for objects

Generating TAC

TAC commands

- var1 = [constant | var2];
- var1 = [constant | var2] op [constant | var3];
- *(var1 [+ constant]) = var2
- var1 = *(var2 [+ constant])

Sample TAC Code

```
int  a;
int  b;
int  c;
int d;

a = b + c + d;
b = a * a + b * b;
```

```
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```

cgen for Basic Expressions

cgen for Basic Expressions

```
cgen(k) = { // k is a constant
    Choose a new temporary t
    Emit( t = k );
    Return t
}
```

cgen for Basic Expressions

```
cgen(k) = { // k is a constant
    Choose a new temporary t
    Emit( t = k );
    Return t
}
cgen(id) = { // id is an identifier
    Choose a new temporary t
    Emit( t = id )
    Return t
}
```

cgen for Binary Operators

cgen for Binary Operators

cgen(e_1 + e_2) = {

Choose a new temporary t

Let t_1 = **cgen**(e_1)

Let t_2 = **cgen**(e_2)

Emit($t = t_1 + t_2$)

Return t

}

cgen for Simple Statements

TAC commands

- var1 = [constant | var2];
- var1 = [constant | var2] op [constant | var3];
- *(var1 [+ constant]) = var2
- var1 = *(var2 [+ constant])

- Labels,
  - Goto label
  - IfZ var Goto label

cgen for **`while`** loops

cgen(**`while`** **`(`***expr***`)`** *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before} **`:`**)

Let *t* = **cgen**(*expr*)

Emit(**`IfZ`** *t* **`Goto`** L_{after})

cgen(*stmt*)

Emit(**`Goto`** L_{before})

Emit(L_{after} **`:`**)

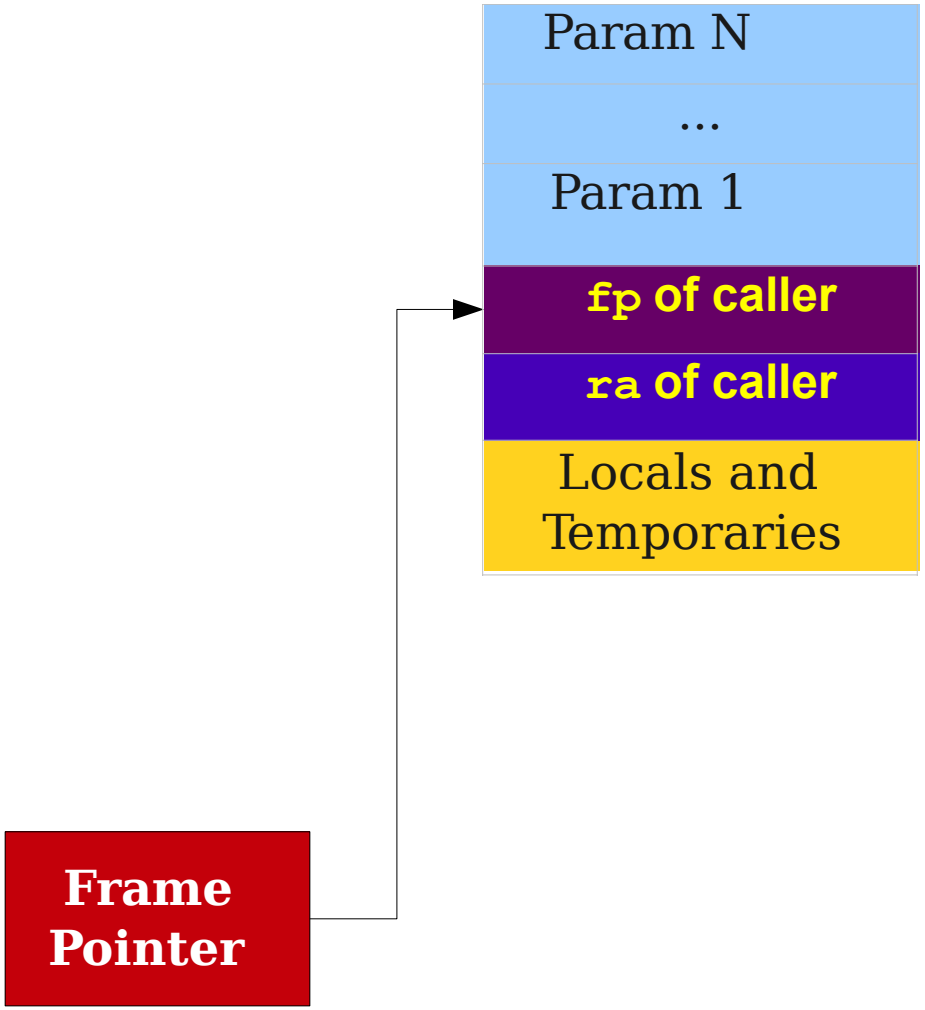
}

Compiling Functions

- BeginFunc N
  - Reserves N bytes
- EndFunction
  - Frees N bytes

- Call

Physical Stack Frames



A Complete Decaf Program

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

```
main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
_L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
_L1:
    EndFunc;
```

A Complete Decaf Program

```
void f(int a) {
}

void main() {
   int x, y;
   f(x);
}
```

```
f:
   BeginFunc 0;

   EndFunc;

main:
   BeginFunc 8;
   PushParam x;
   Call f;
   PopParams 4;
   EndFunc;
```

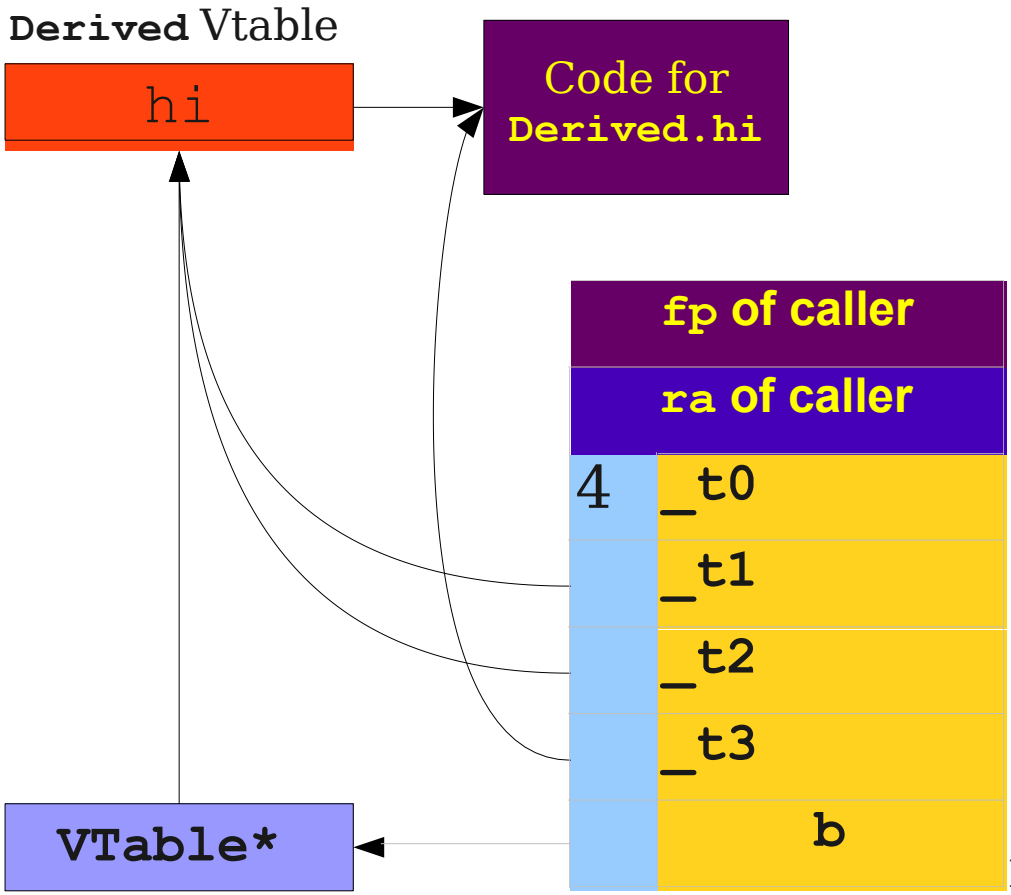
Compiling Function Calls

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main()
   {  SimpleFunction(137
   );
}
```

```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam  _t0;
    LCall  _SimpleFn;
    PopParams  4;
    EndFunc;
```

Objects

Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

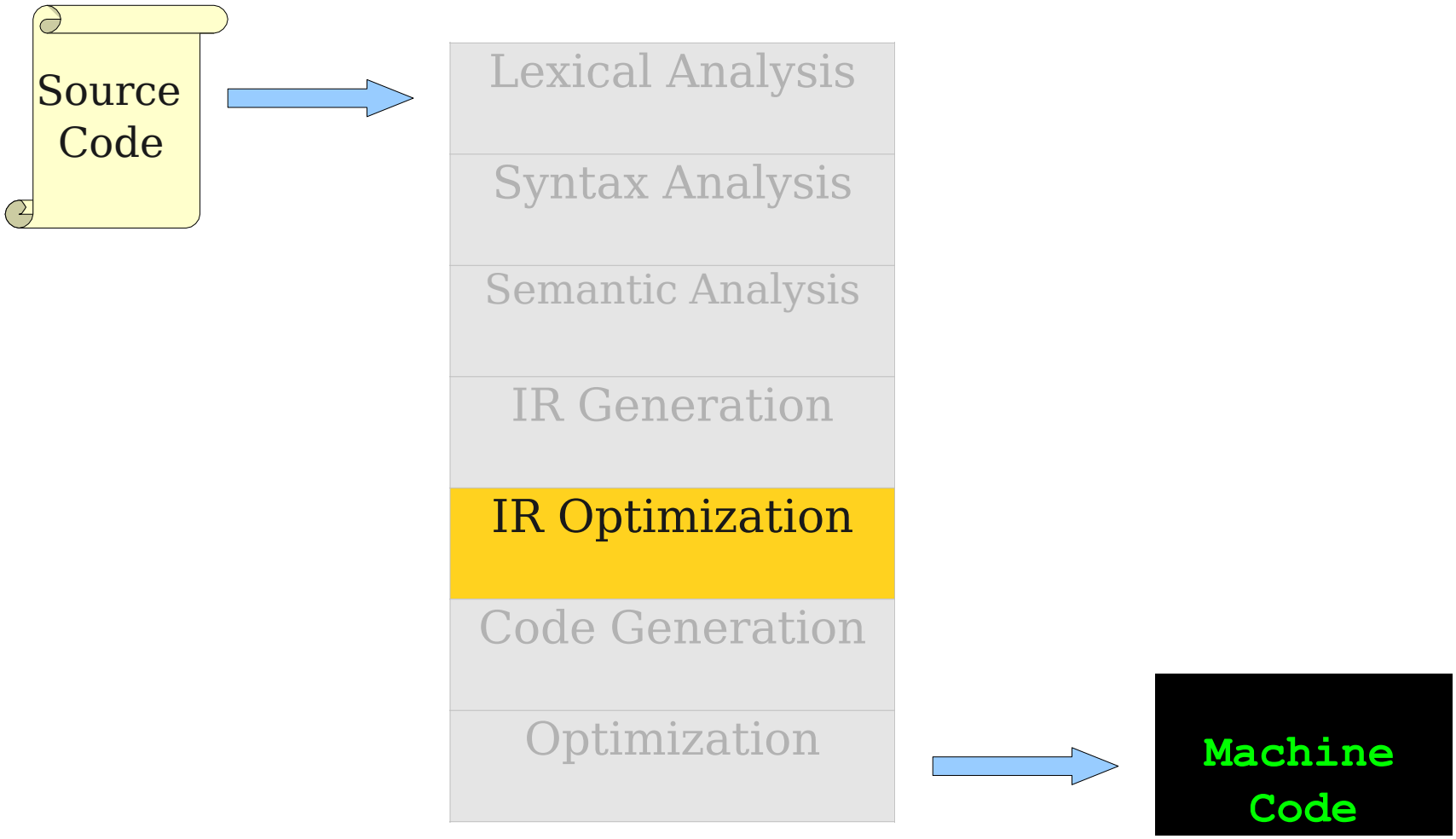
Allocate Object

Set Vtable

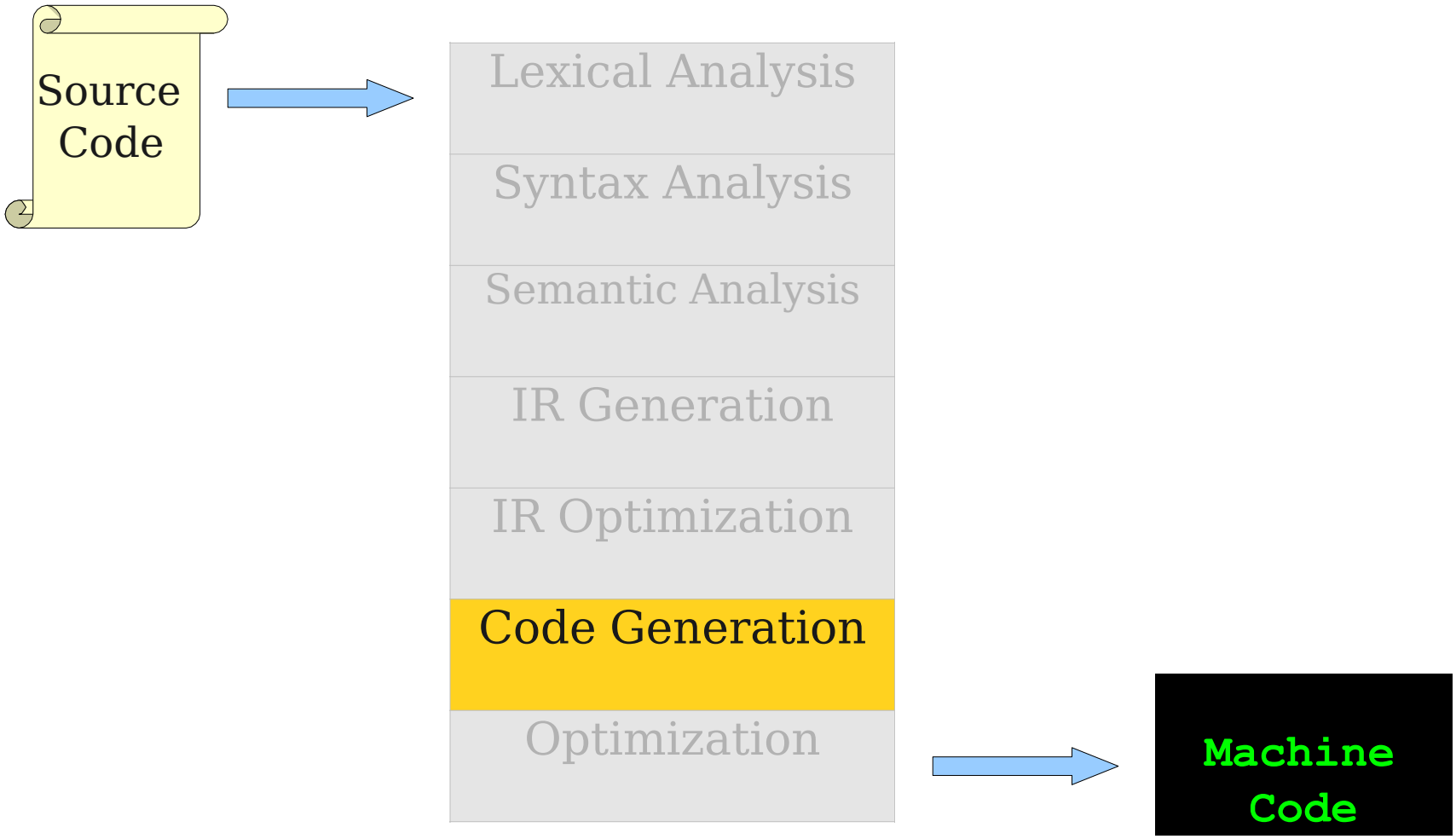
Load Function

Register Allocation

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Code Generation at a Glance

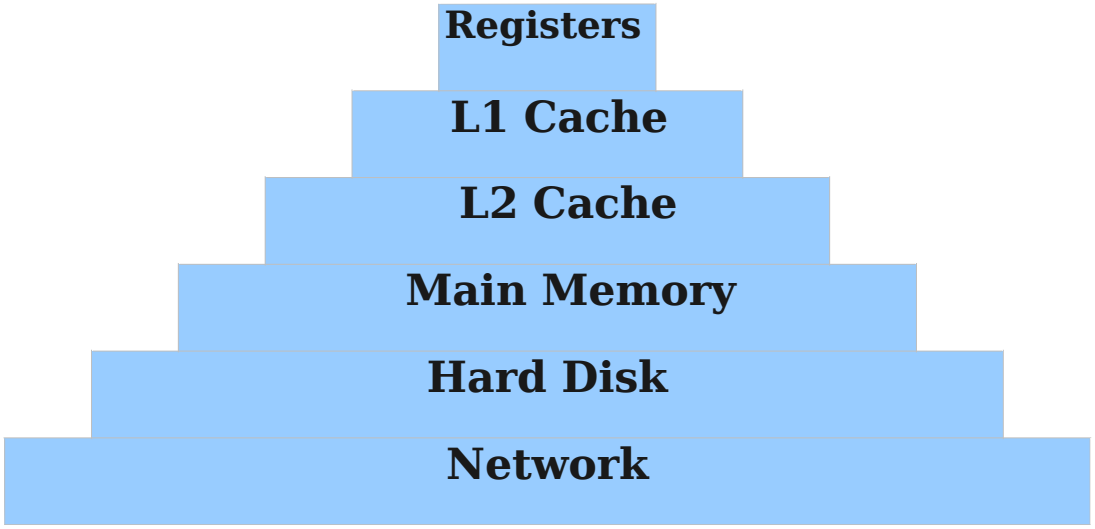
- At this point, we have optimized IR code that needs to be converted into the target language (e.g. assembly, machine code).
- Goal of this stage:
  - Choose the appropriate machine instructions for each IR instruction.
  - Divvy up finite machine resources (registers, caches, etc.)
  - Implement low-level details of the runtime environment.
  -
- Machine-specific optimizations are often done here, though some are treated as part of a final optimization phase.

Overview

- **Register Allocation** (Today)
  - How to assign variables to finitely many registers?
  - What to do when it can't be done?
  - How to do so efficienty?

The Memory Hierarchy

- **Idea**: Try to get the best of all worlds by using multiple types of memory.



256B - 8KB	0.25 – 1ns
16KB – 64KB	1ns – 5ns
1MB - 4MB	5ns – 25ns
4GB – 256GB	25ns – 100ns
500GB+	3 – 10ms
HUGE	10 – 2000ms

Registers

- Most machines have a set of **registers**, dedicated memory locations that
  - can be accessed quickly,
  - can have computations performed on them, and
  - exist in small quantity.
- Using registers intelligently is a critical step in any compiler.
  - A good register allocator can generate code orders of magnitude better than a bad register allocator.

Register Allocation

- In TAC, there are an unlimited number of variables.
- On a physical machine there are a small number of registers:
  - x86 has four general-purpose registers and a number of specialized registers.
  - MIPS has twenty-four general-purpose registers and eight special-purpose registers.
- **Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers.

Challenges in Register Allocation

- **Registers are scarce**.
  - Often substantially more IR variables than registers. Need to find a way to reuse registers whenever possible.
- **Registers are complicated**.
  - x86: Each register made of several smaller registers; can't use a register and its constituent registers at the same time.
  - x86: Certain instructions must store their results in specific registers; can't store values there if you want to use those instructions.
  - MIPS: Some registers reserved for the assembler or operating system.
  - Most architectures: Some registers must be preserved across function calls.

Goals for Today

- Introduce register allocation for a MIPS-style machine:
  - Some number of indivisible, general-purpose registers.
- Explore three algorithms for register allocation:
  - Naïve (“no”) register allocation.
  - Linear scan register allocation.
  - Graph-coloring register allocation.

An Initial Register Allocator

- **Idea**: Store every value in main memory, loading values only when they're needed.
- To generate a code that performs a computation:
  - Generate **load** instructions to pull the values from main memory into registers.
  - Generate code to perform the computation on the registers.
  - Generate **store** instructions to store the result back into main memory.

Our Register Allocator In Action

Our Register Allocator In Action

```
a = b  + c;
d = a;
c = a  + d;
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

Param N	**fp + 4N**
...	**...**
Param 1	**fp + 4**
Stored fp	**fp + 0**
Stored ra	**fp - 4**
a	**fp - 8**
b	**fp - 12**
c	**fp - 16**
d	**fp - 20**

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

```
lw  $t0, -12(fp)
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)

lw   $t0,  -8(fp)
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2, -8(fp)

lw   $t0, -8(fp)

sw   $t0, -20(fp)
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

```
lw   $t0,  -12(fp)
lw   $t1,  -16(fp)
add  $t2,  $t0,  $t1
sw   $t2,  -8(fp)
lw   $t0,  -8(fp)
sw   $t0,  -20(fp)
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2, -8(fp)
lw   $t0, -8(fp)
sw   $t0, -20(fp)
lw   $t0, -8(fp)
```

Our Register Allocator In Action

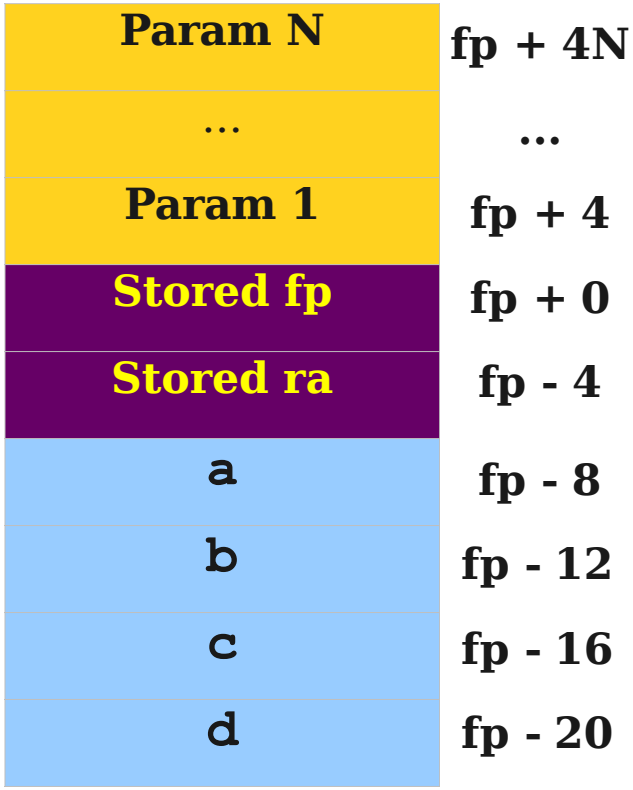
```
a = b + c;
d = a;
c = a + d;
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

```
lw  $t0, -12(fp)
lw  $t1, -16(fp)
add $t2, $t0, $t1
sw  $t2, -8(fp)
lw  $t0, -8(fp)
sw  $t0, -20(fp)
lw  $t0, -8(fp)
lw  $t1, -20(fp)
```

Our Register Allocator In Action

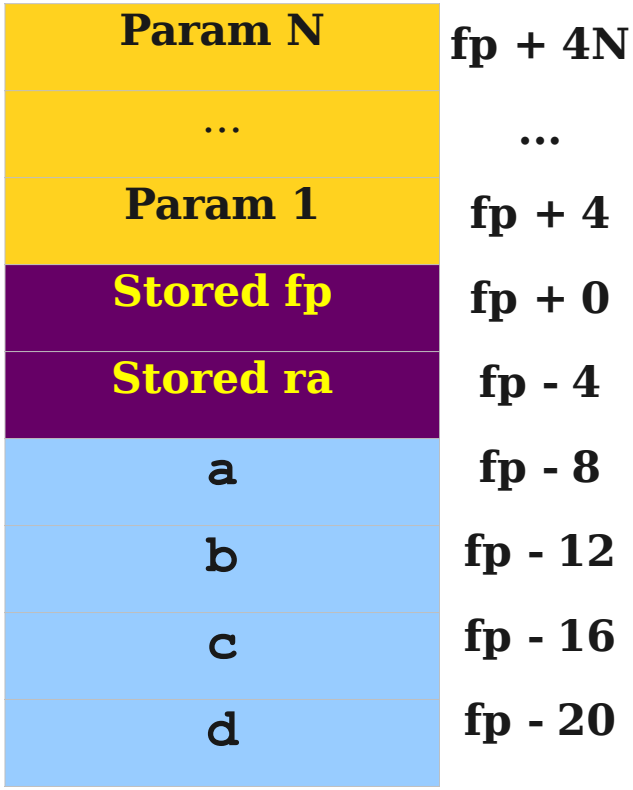
```
a = b + c;
d = a;
c = a + d;
```




```
lw  $t0, -12(fp)
lw  $t1, -16(fp)
add $t2, $t0, $t1
sw  $t2, -8(fp)
lw  $t0, -8(fp)
sw  $t0, -20(fp)
lw  $t0, -8(fp)
lw  $t1, -20(fp)
add $t2, $t0, $t1
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```




```
lw   $t0,  -12(fp)
lw   $t1,  -16(fp)
add  $t2,  $t0,  $t1
sw   $t2,  -8(fp)
lw   $t0,  -8(fp)
sw   $t0,  -20(fp)
lw   $t0,  -8(fp)
lw   $t1,  -20(fp)
add  $t2,  $t0, $t1
sw   $t2,  -16(fp)
```

Analysis of our Allocator

- Disadvantage: **Gross inefficiency**.

- Advantage: **Simplicity**.

Live Ranges and Live Intervals

- Recall: A variable is **live** at a particular program point if its value may be read later before it is written.
  - Can find this using global liveness analysis.
- The **live range** for a variable is the set of program points at which that variable is live.
- The **live interval** for a variable is the smallest subrange of the IR code containing all a variable's live ranges.
  - A property of the IR code, **not** the CFG.
  - Less precise than live ranges, but simpler to work with.

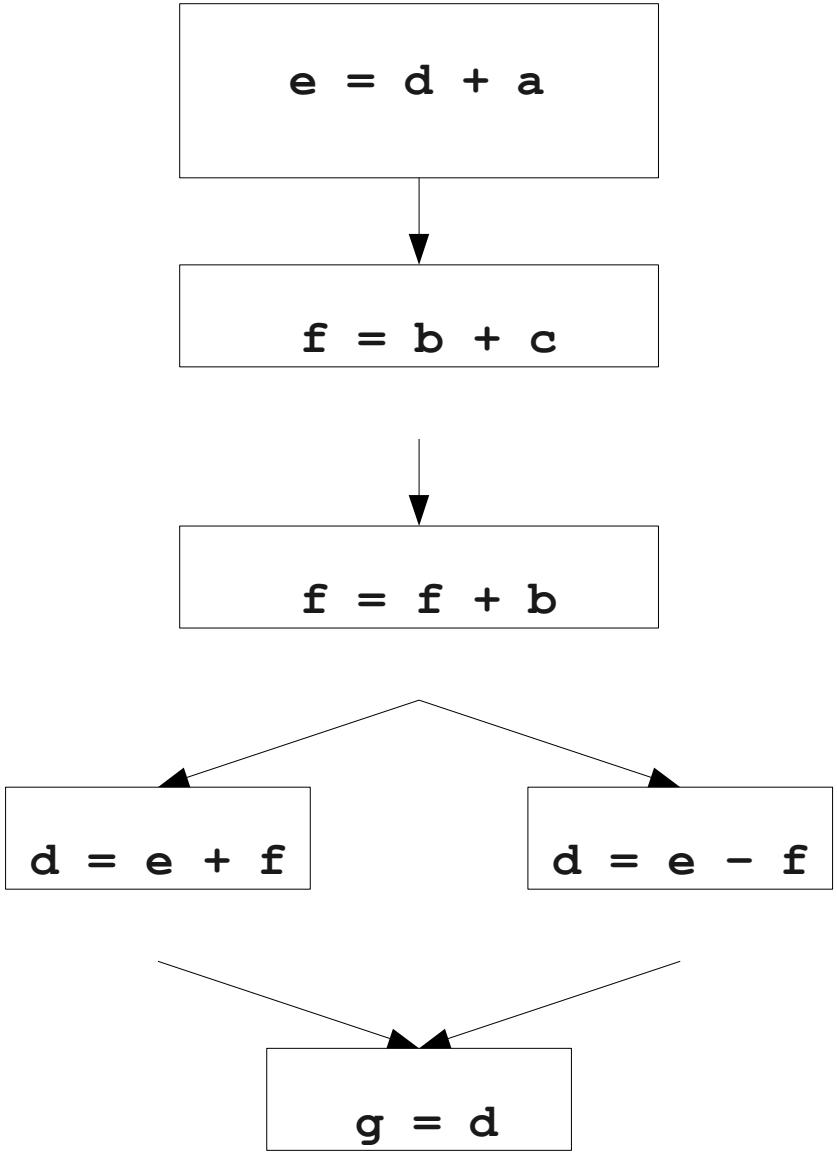
Live Ranges and Live Intervals

Live Ranges and Live Intervals

```
    e = d + a
    f = b + c

    f = f + b
    IfZ e Goto _L0

    d = e + f

    Goto
    _L1;

_L0:

    d = e -
    f

_L1:

    g = d
```

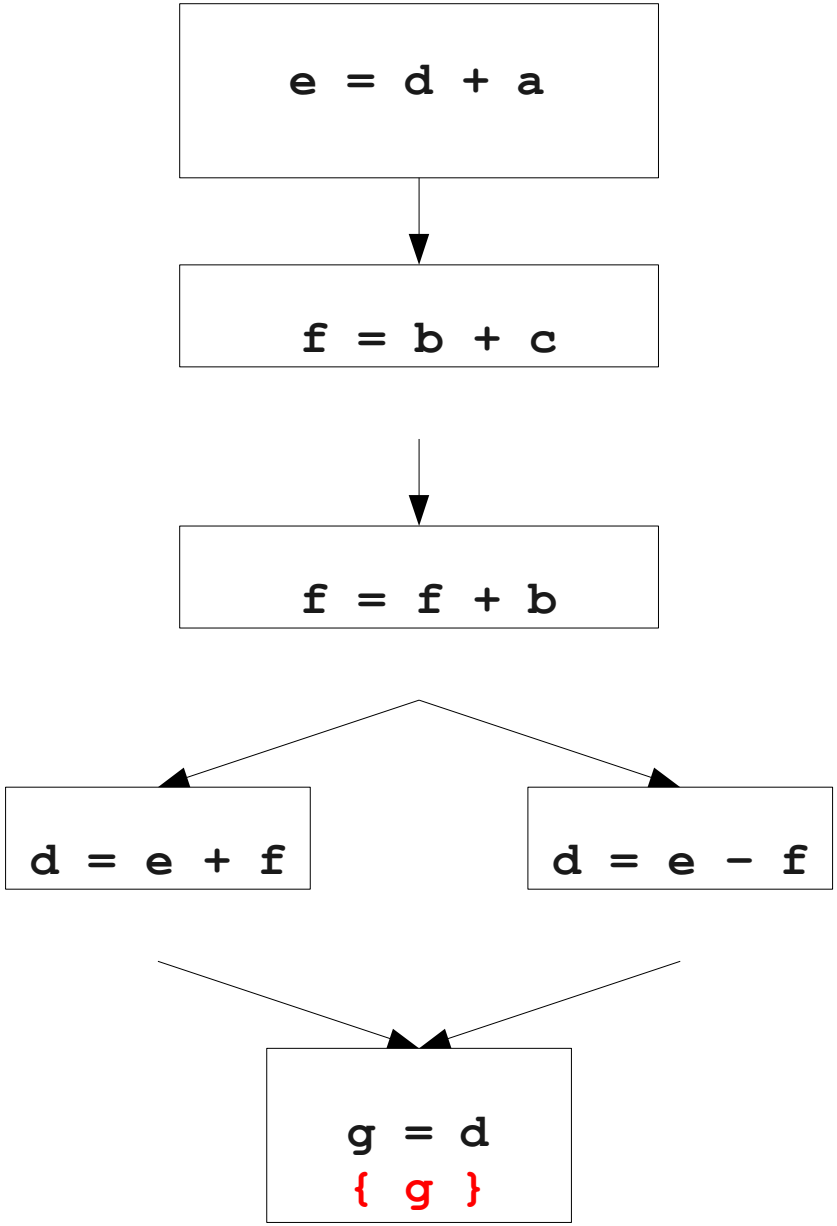
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



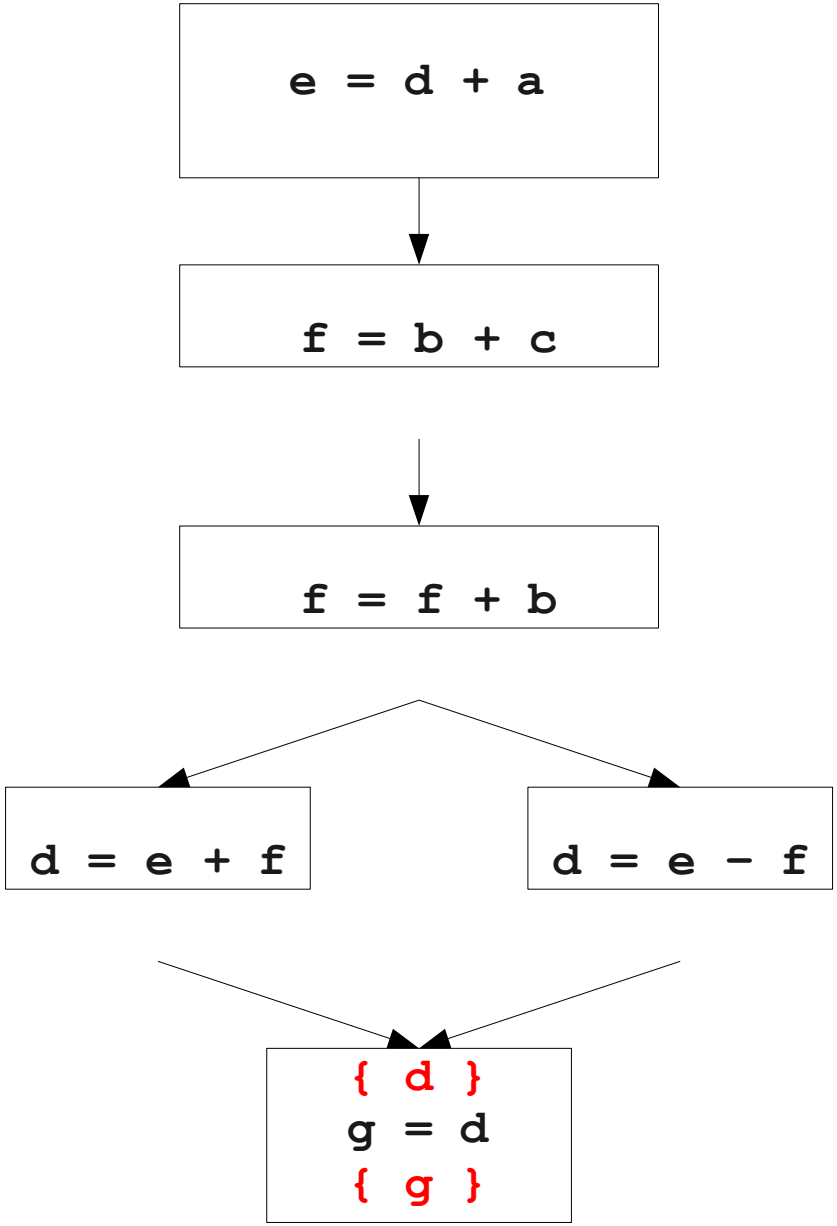
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



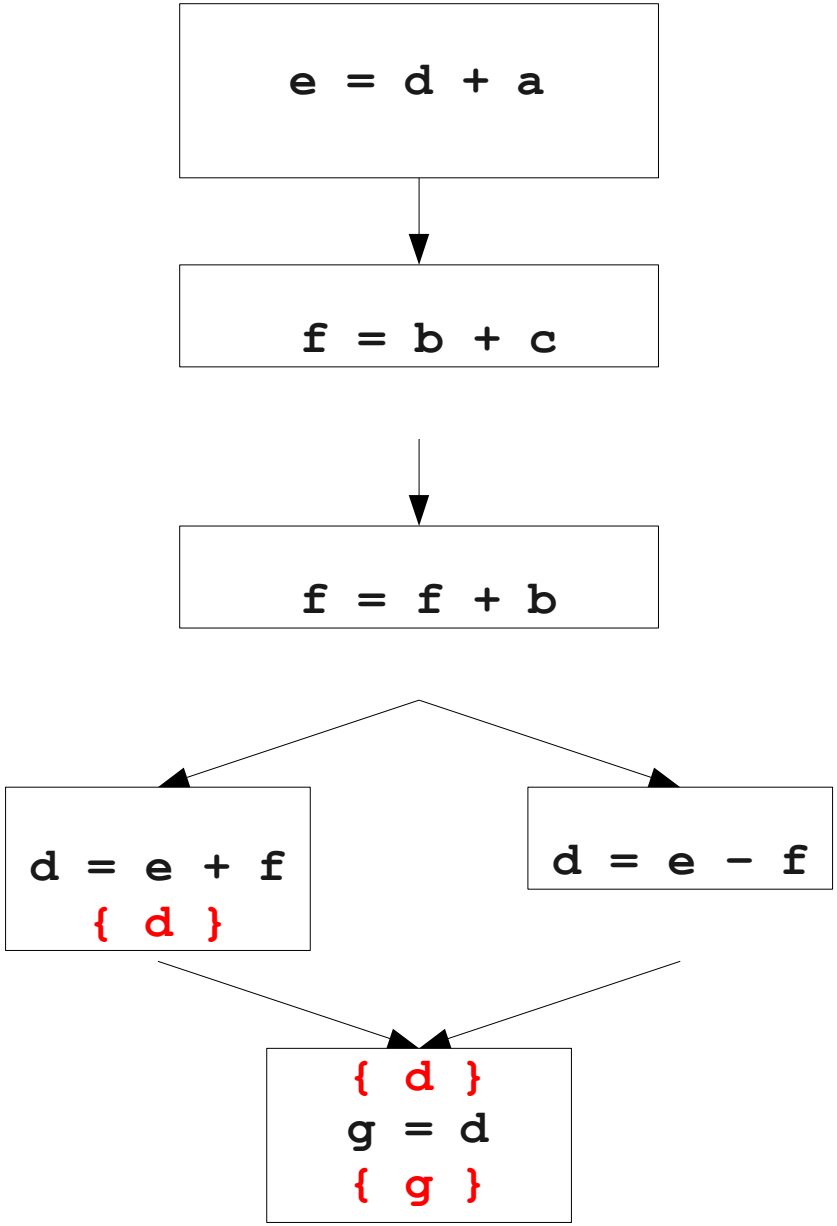
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



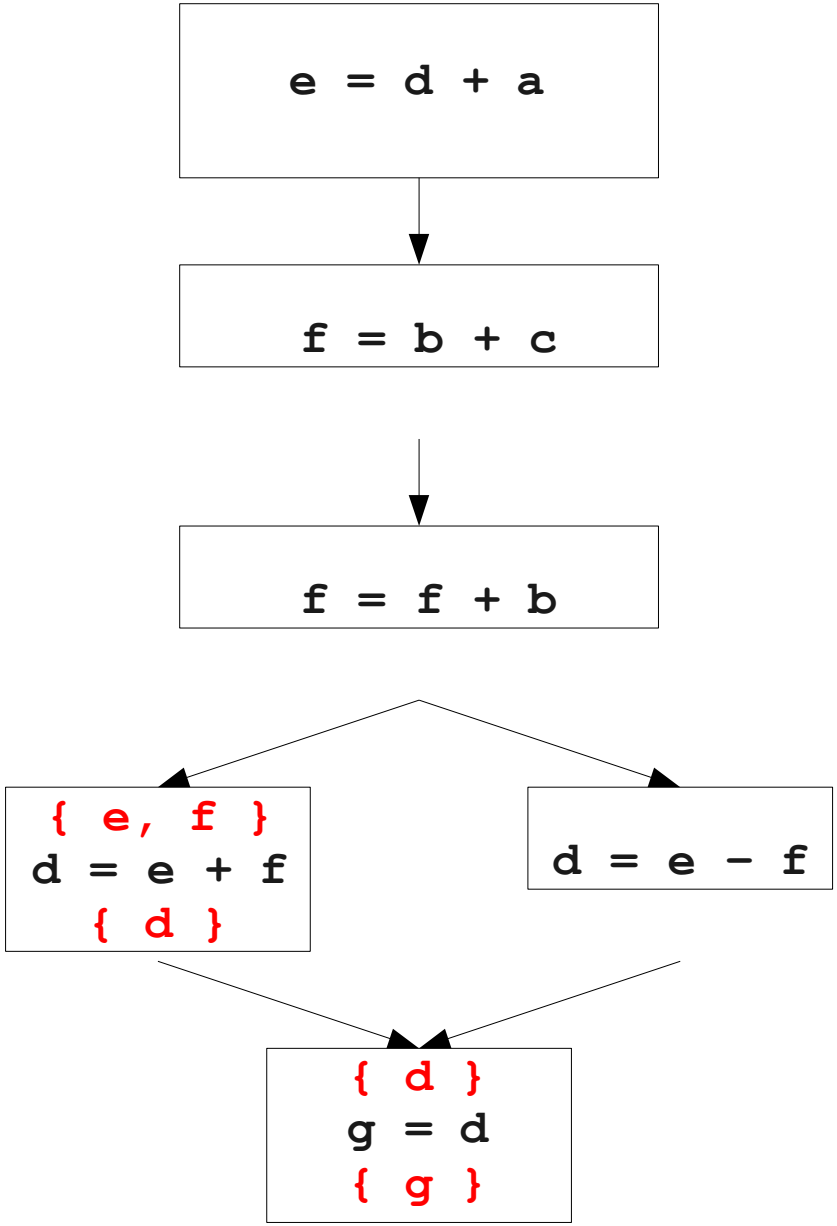
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



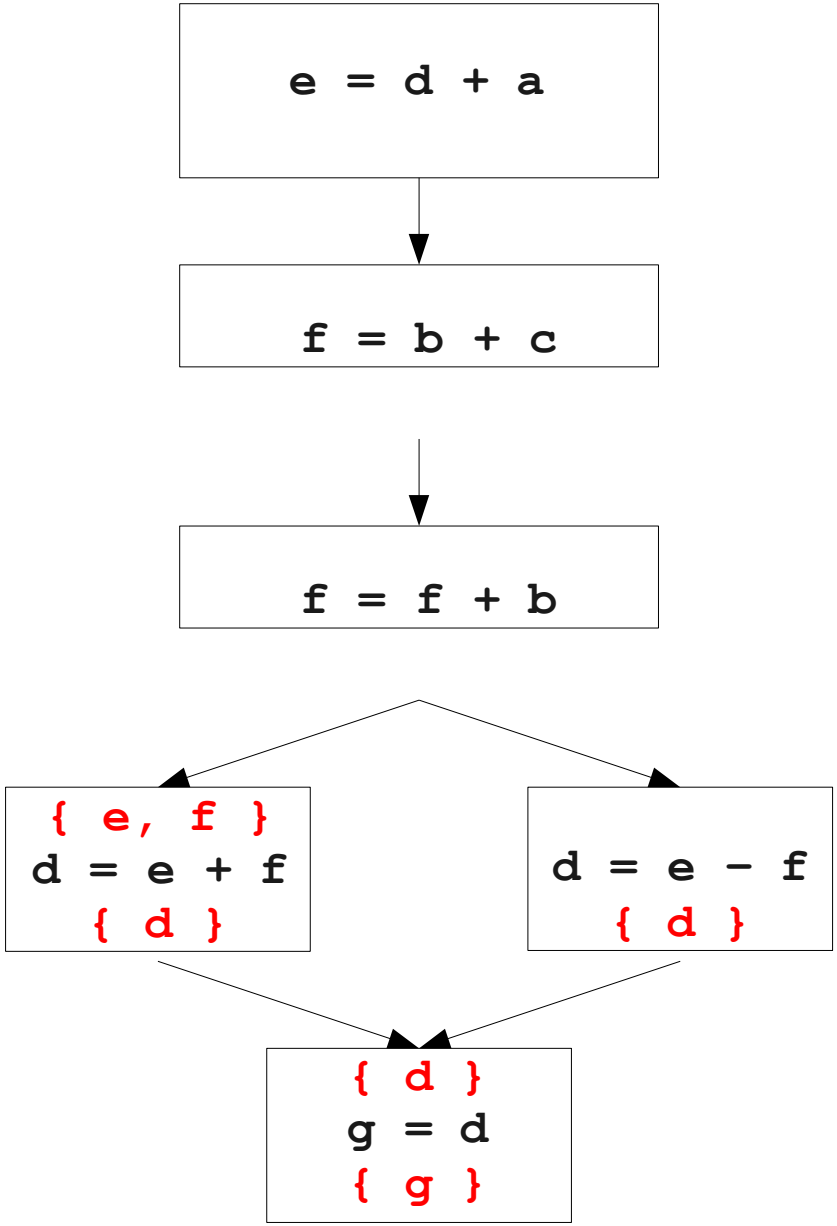
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



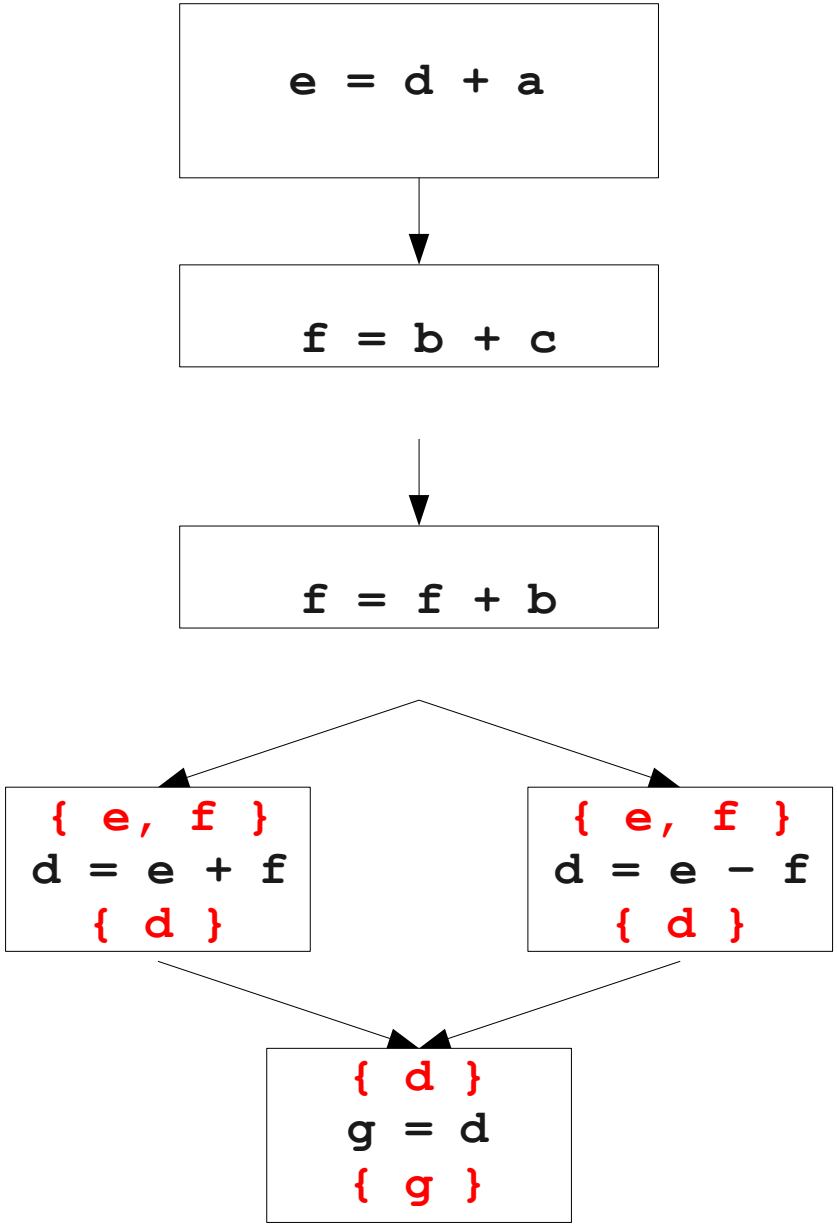
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



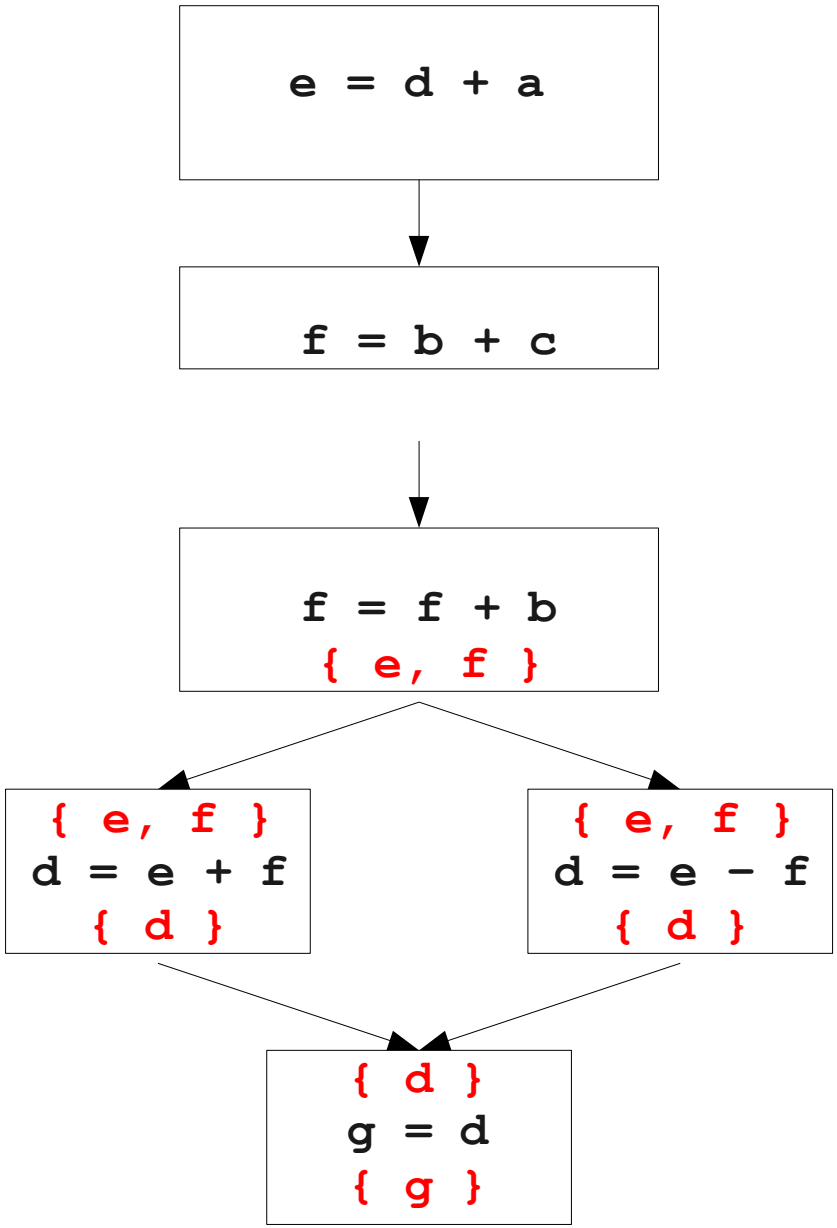
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



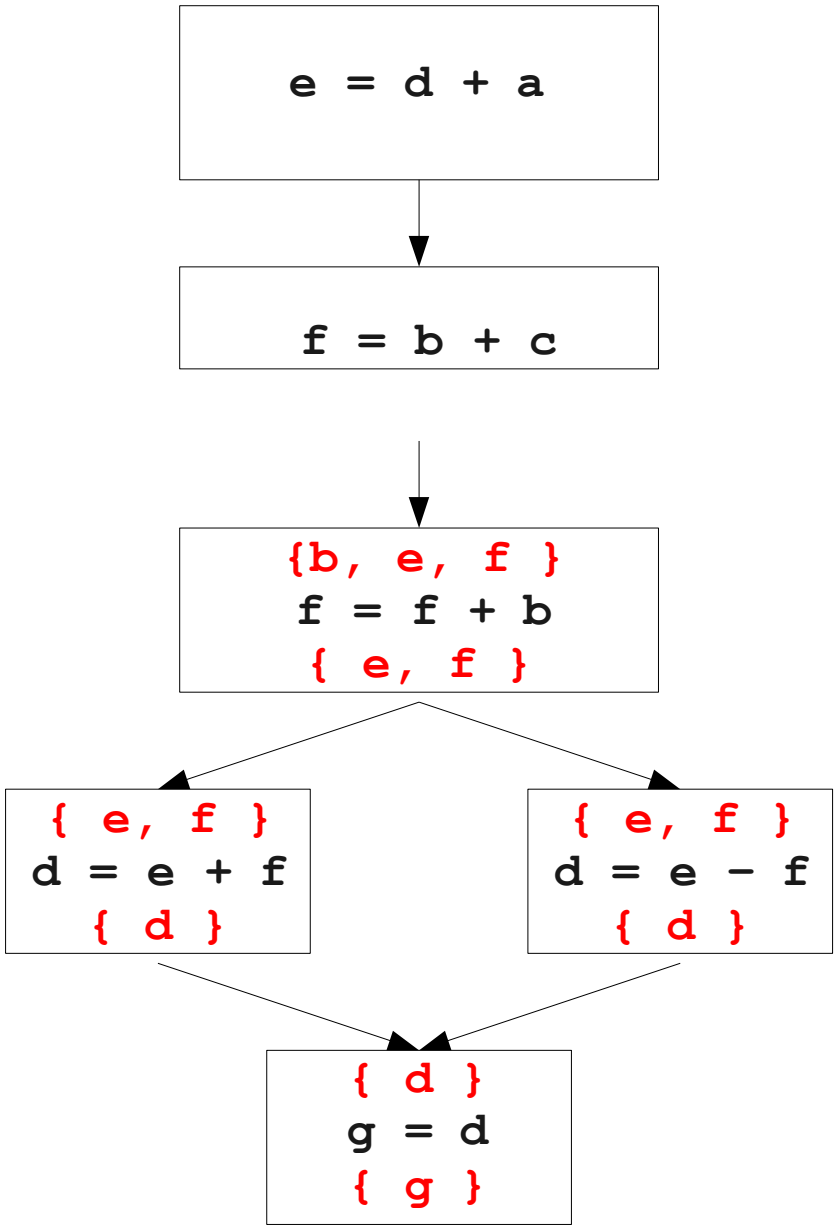
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



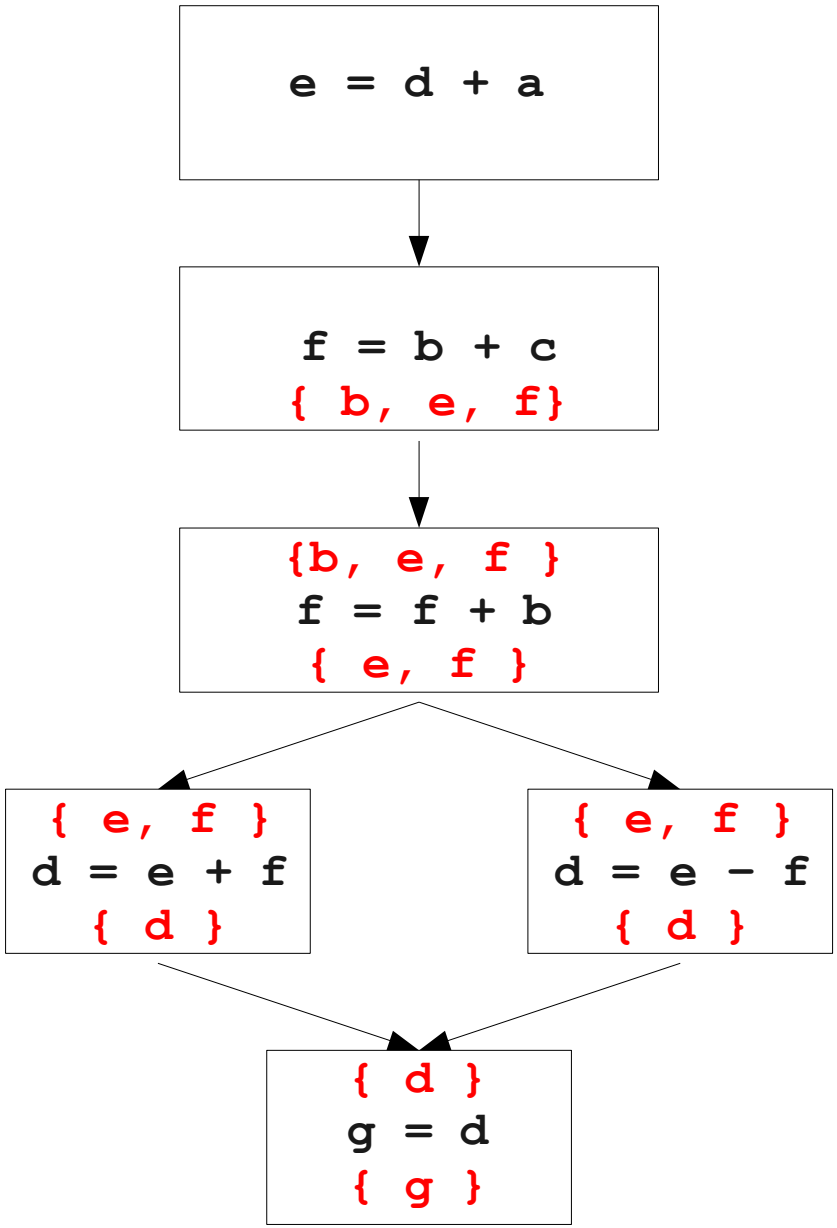
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



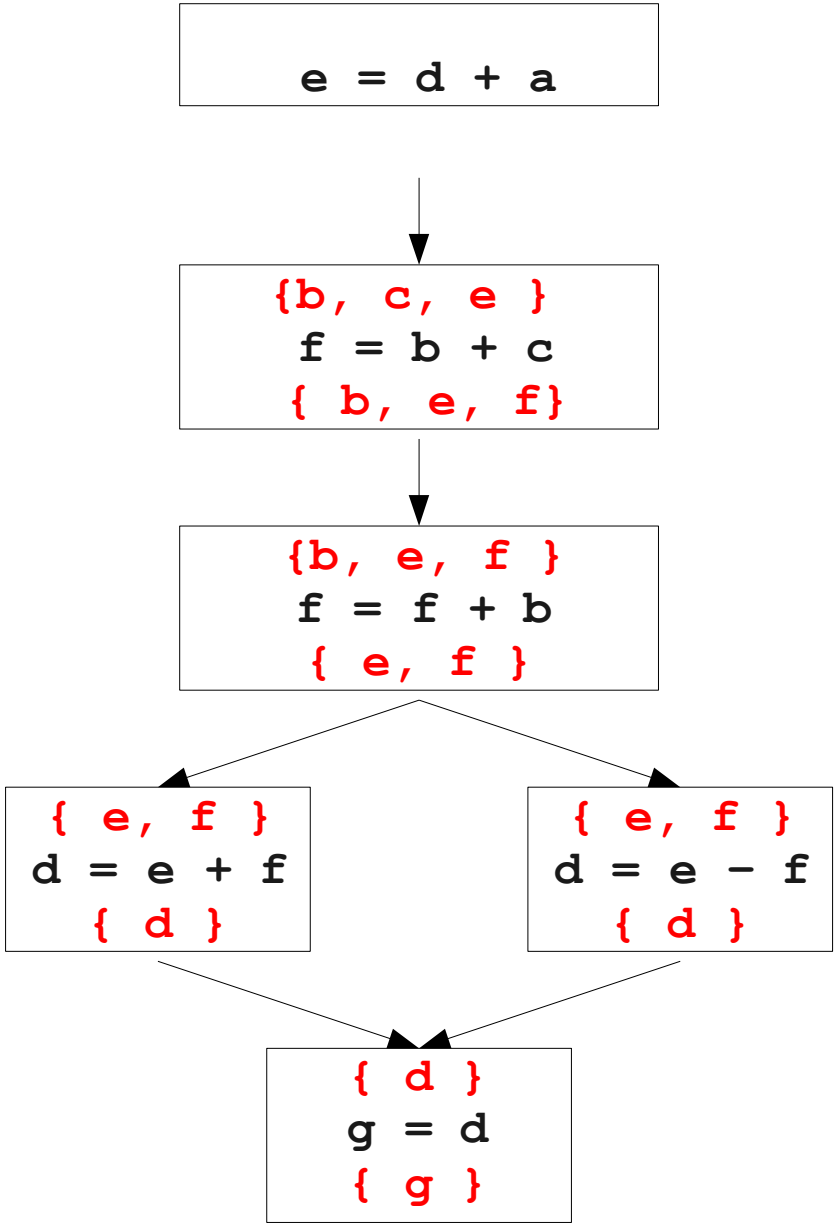
Live Ranges and Live Intervals

```
    e = d + a
    f = b +

    c  f = f

    + b

    IfZ e Goto _L0

    d = e + f

    Goto _L1;

_L0:

    d = e - f

_L1:

    g = d
```



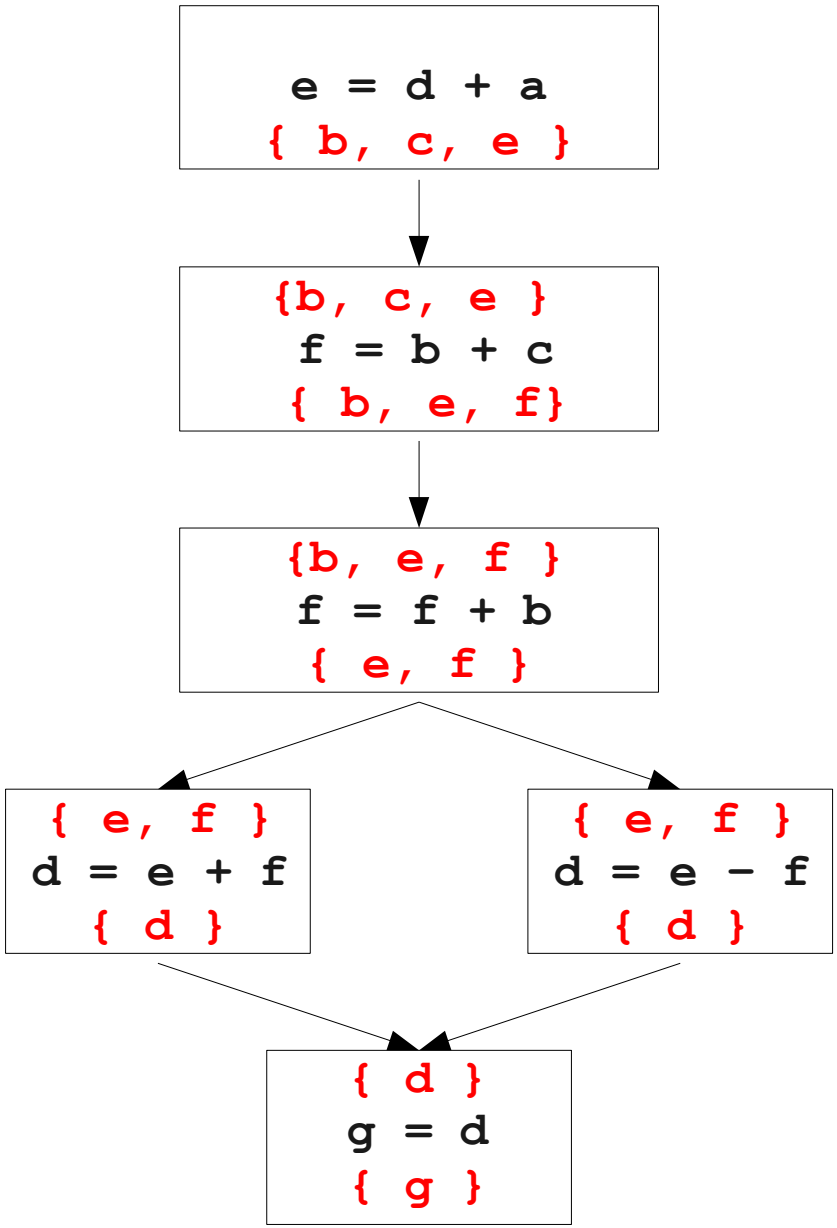
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto
    _L1;
_L0:
    d = e -
    f
_L1:
    g = d
```



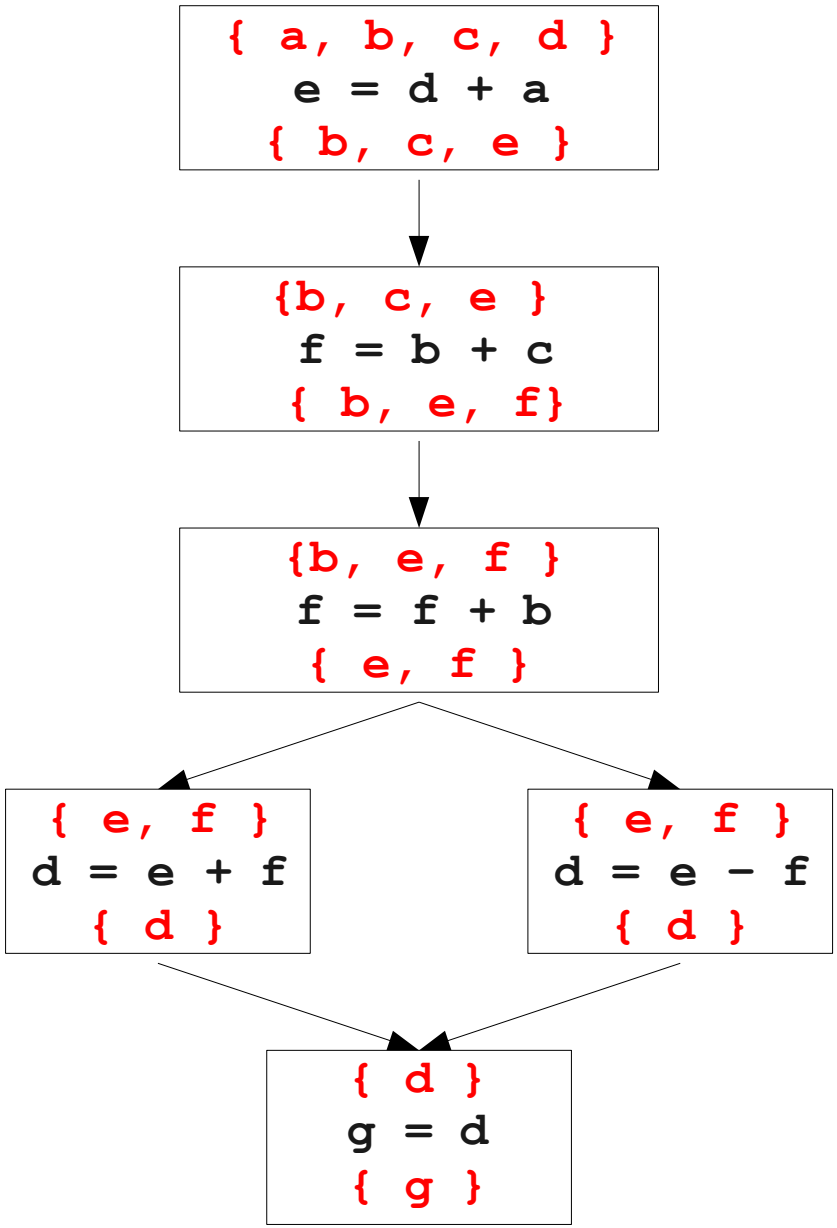
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto
    _L1;
_L0:
    d = e -
    f
_L1:
    g = d
```



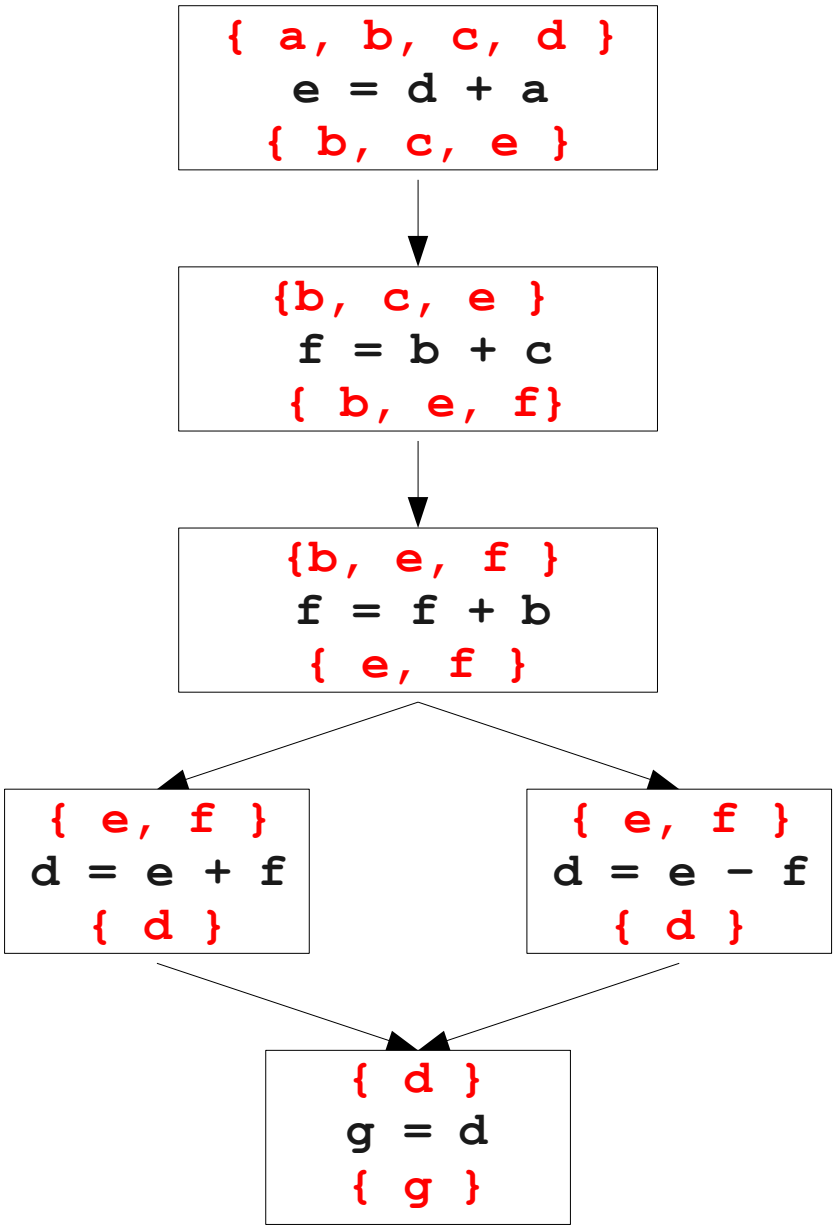
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto
    _L1;
_L0:
    d = e -
    f
_L1:
    g = d
```

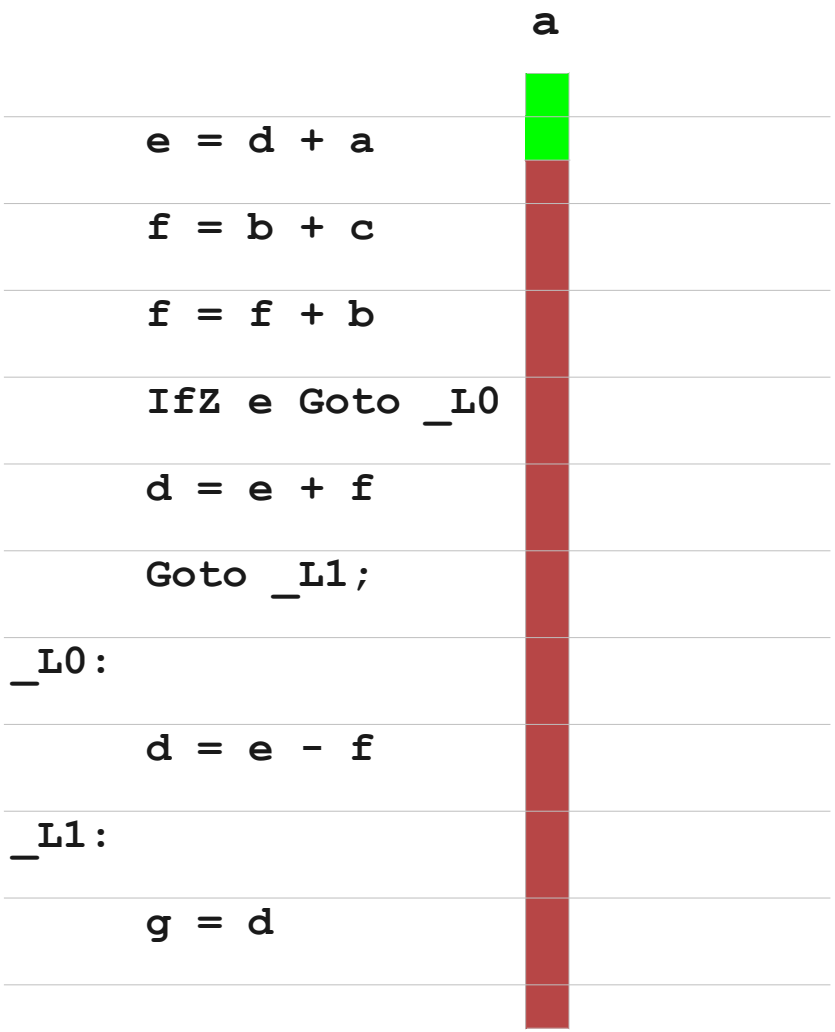


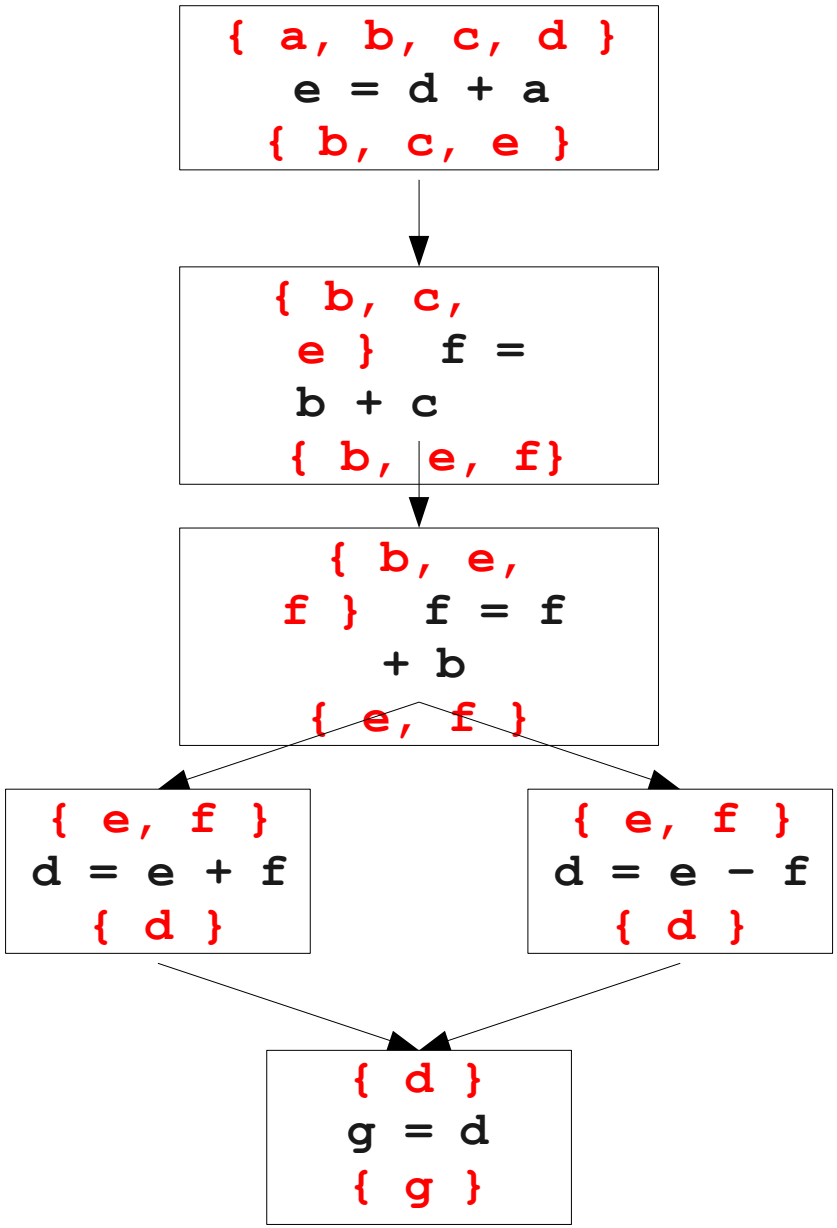
Live Ranges and Live Intervals

```
     e = d + a

     f = b + c

     f = f + b

IfZ      e Goto _L0

      d   e + f
      =

     Goto _L1;

_L0:

     d = e - f

_L1:

     g = d
```

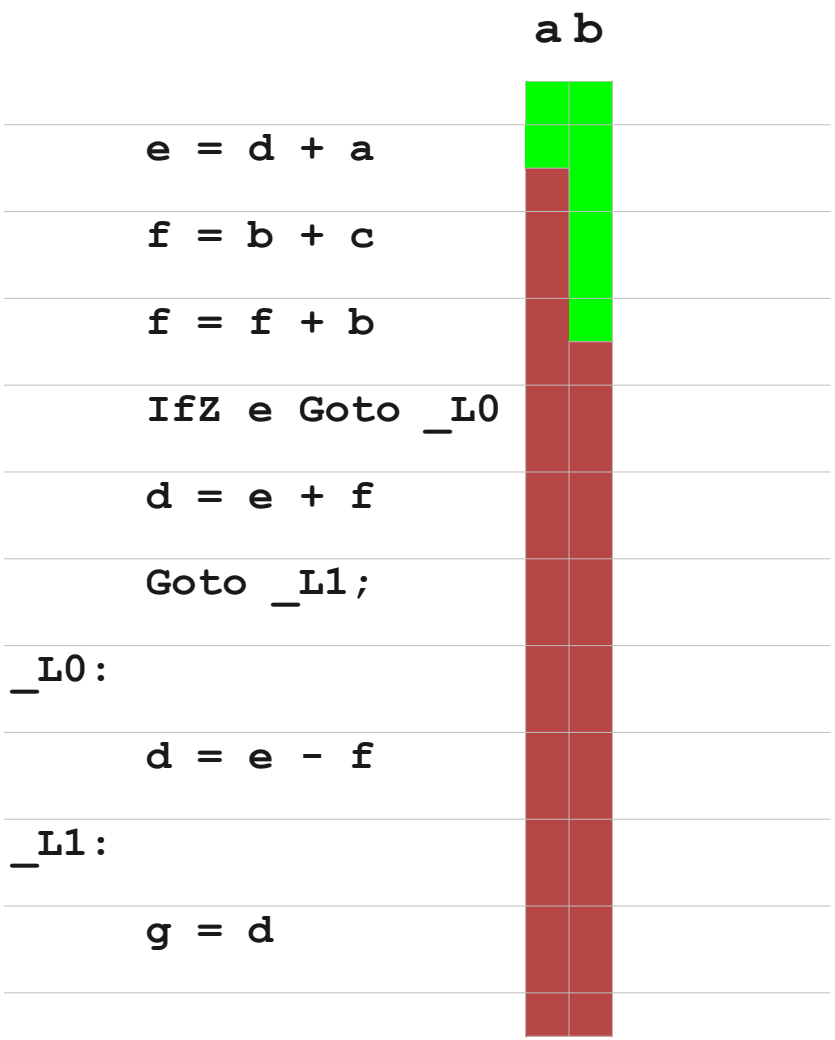


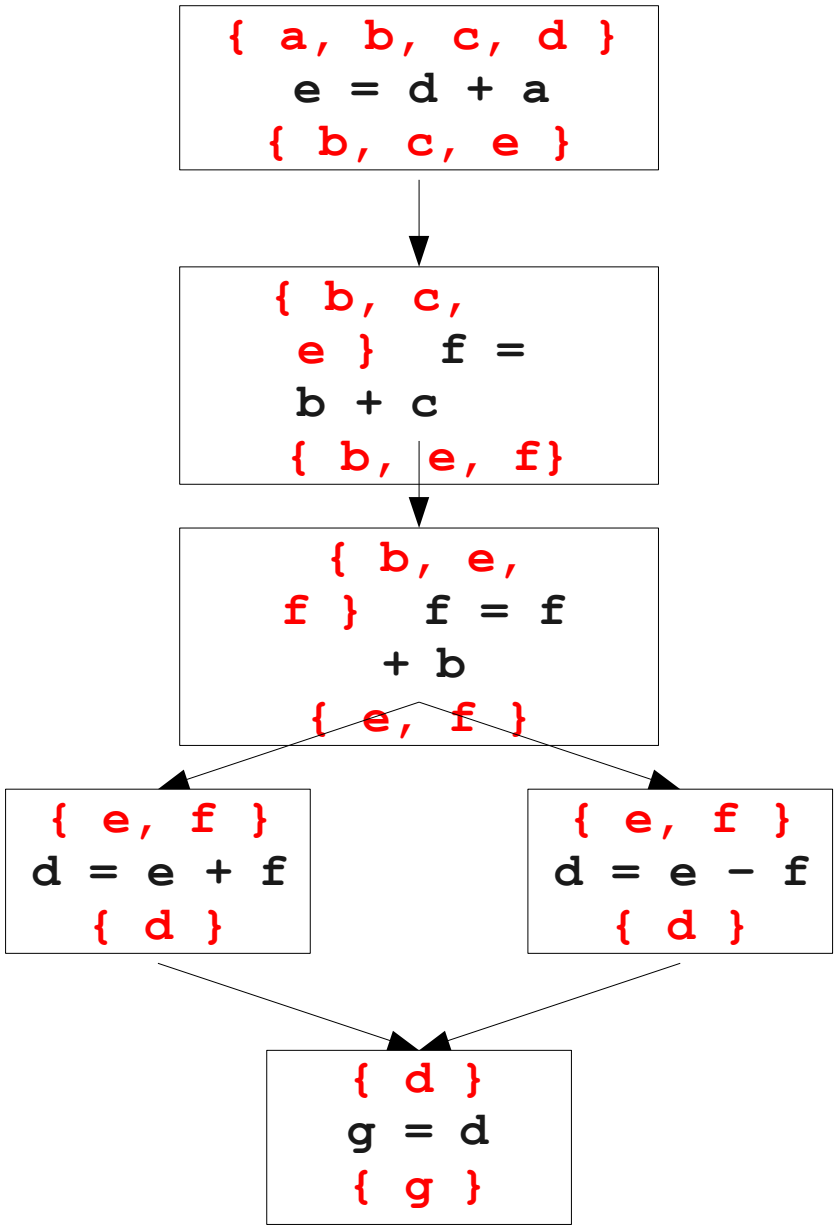
Live Ranges and Live Intervals



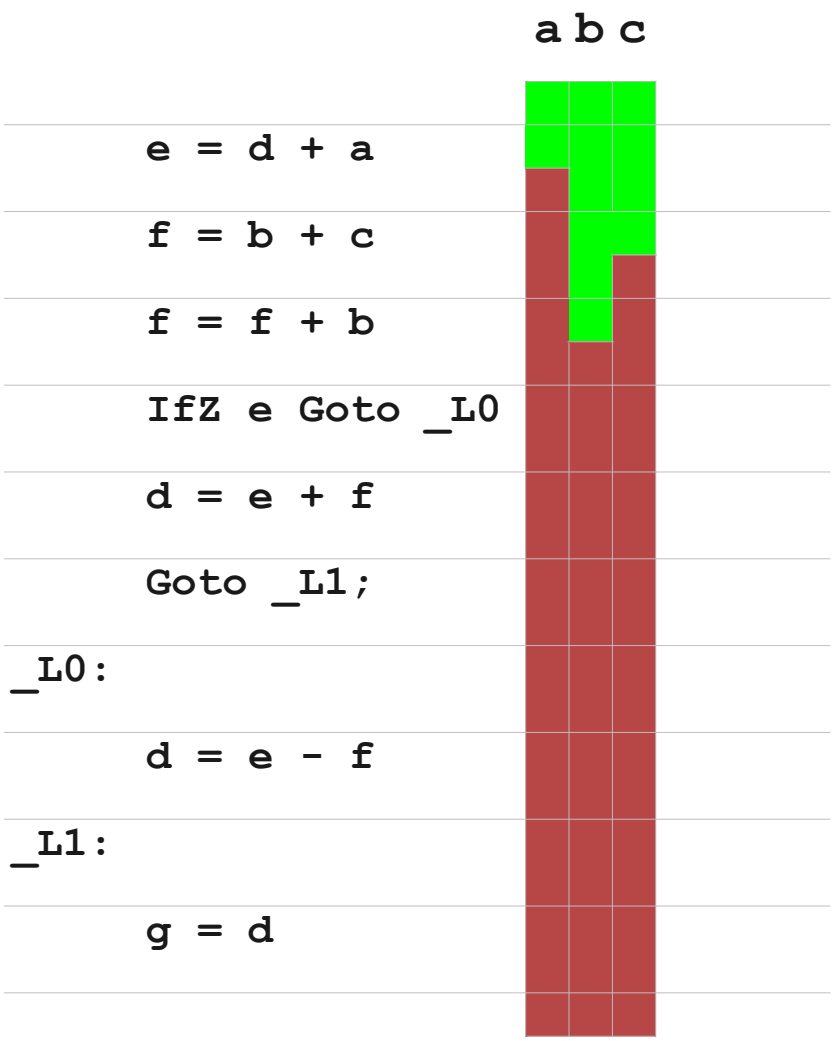


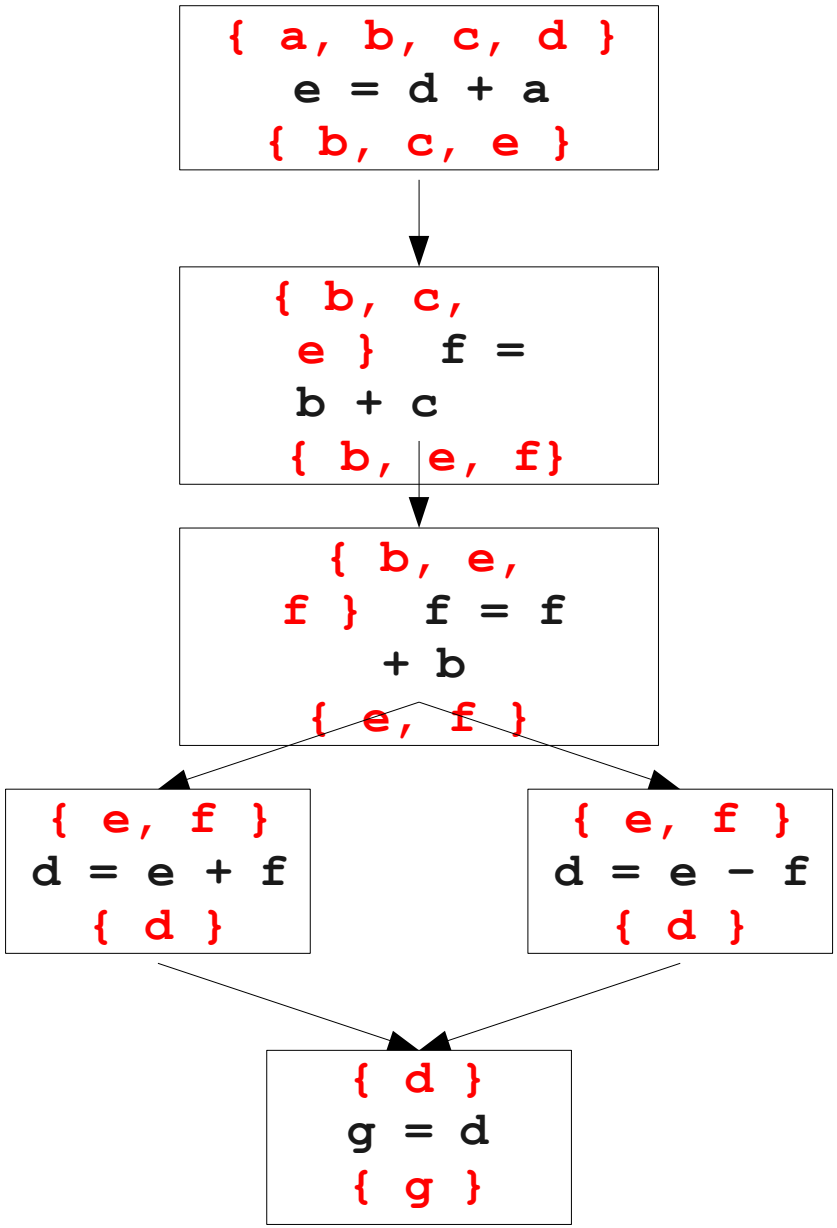
Live Ranges and Live Intervals



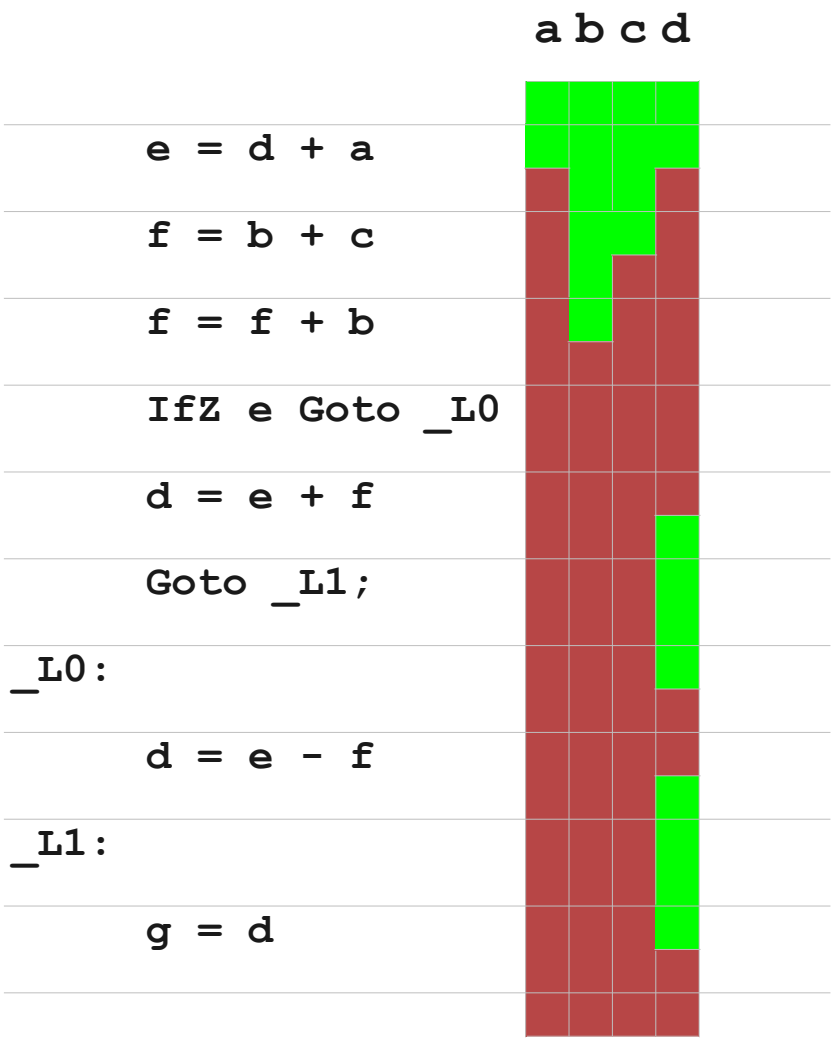


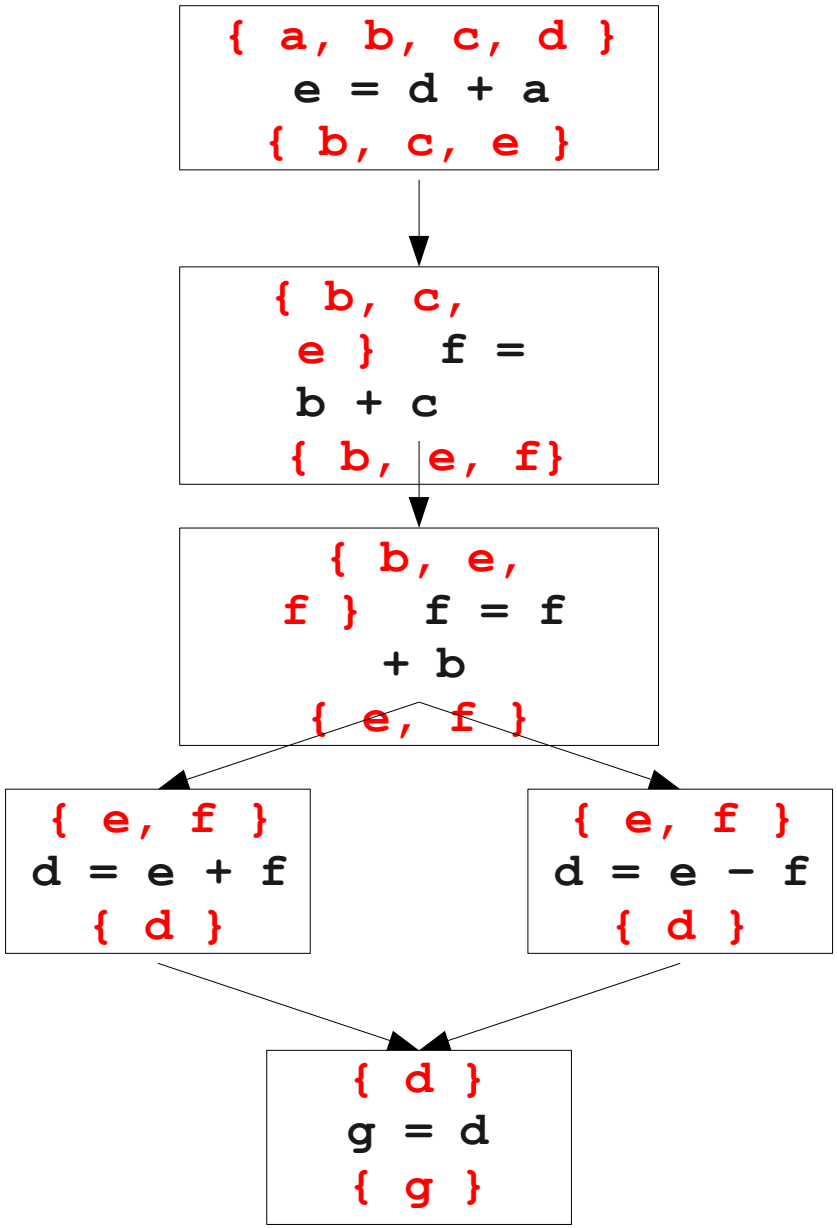
Live Ranges and Live Intervals



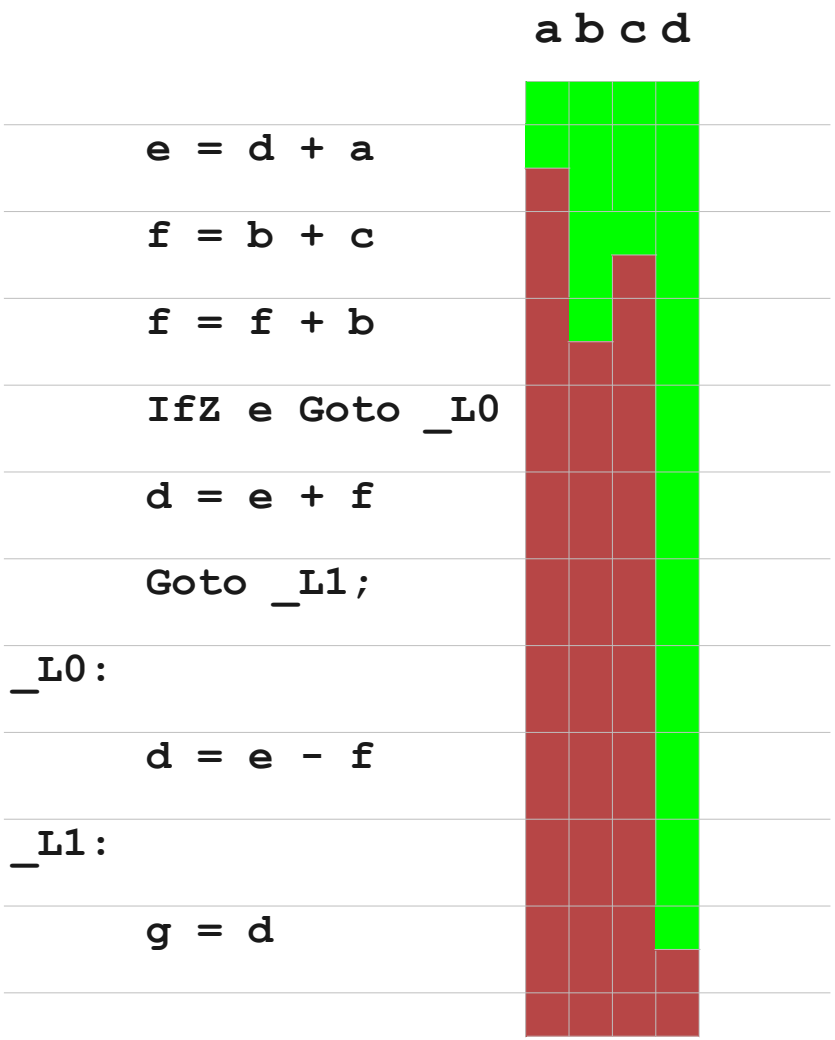


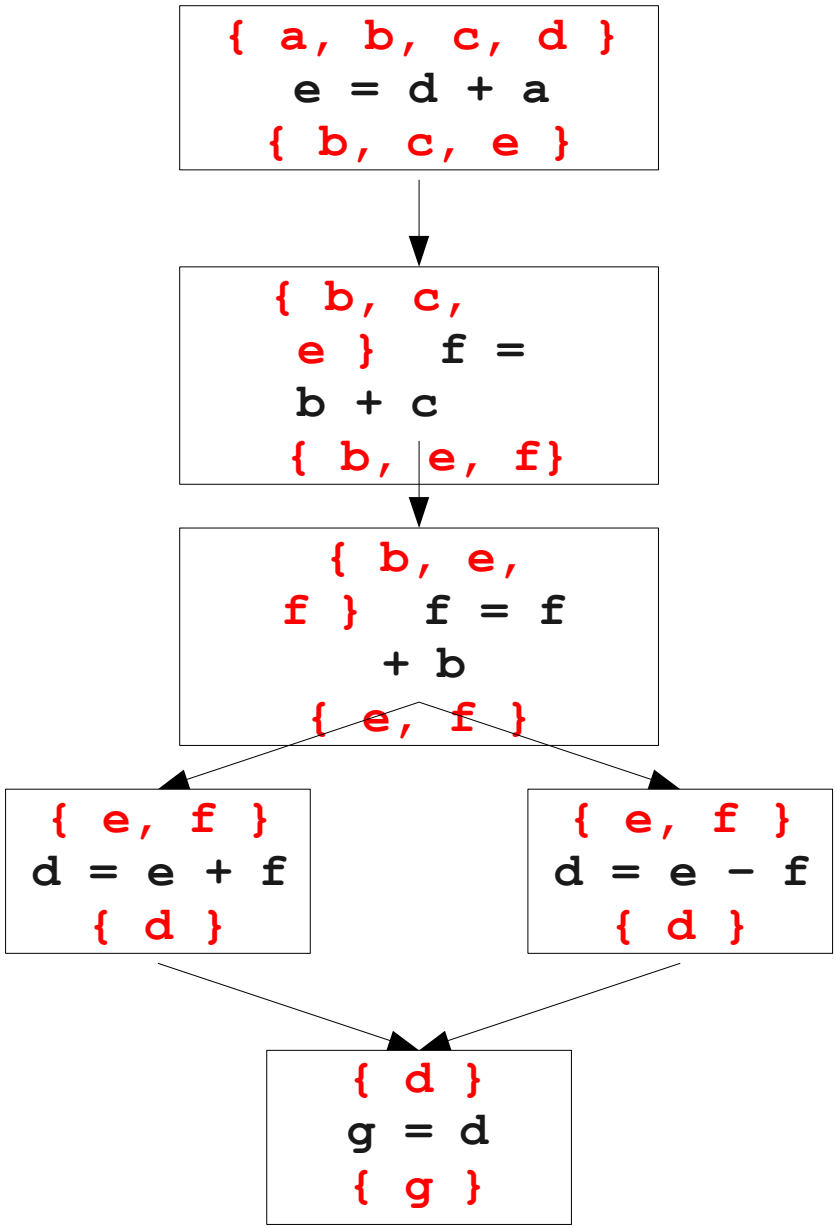
Live Ranges and Live Intervals



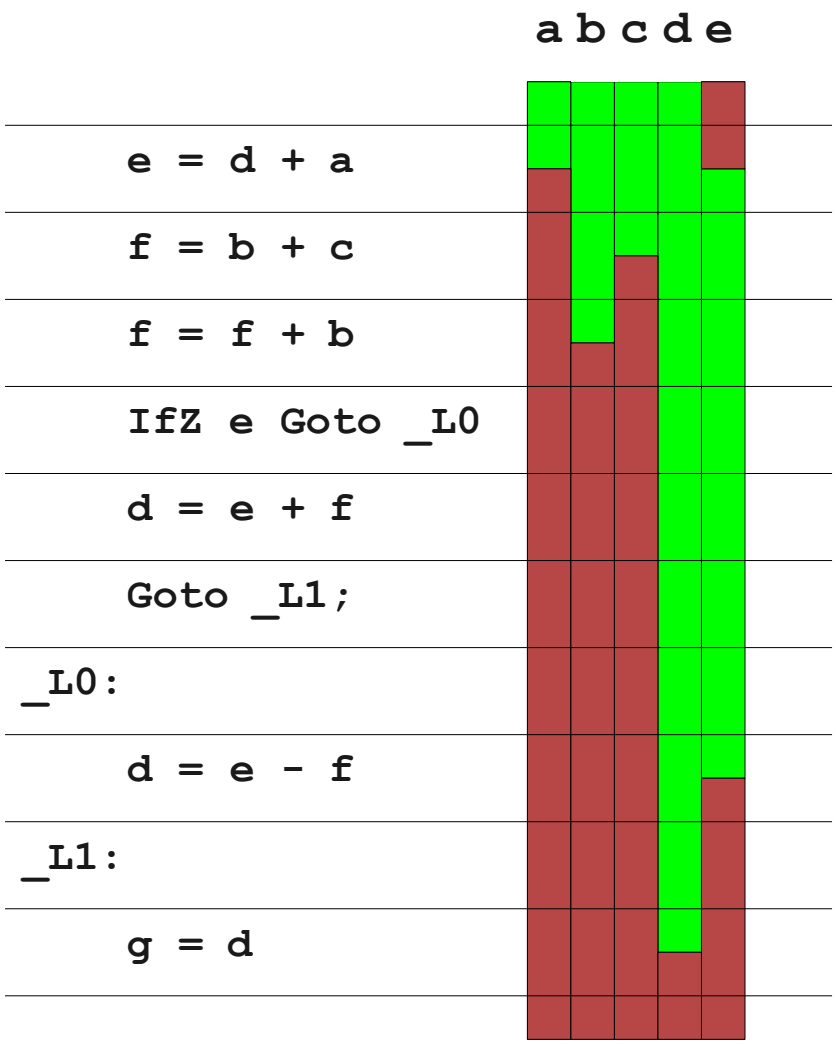


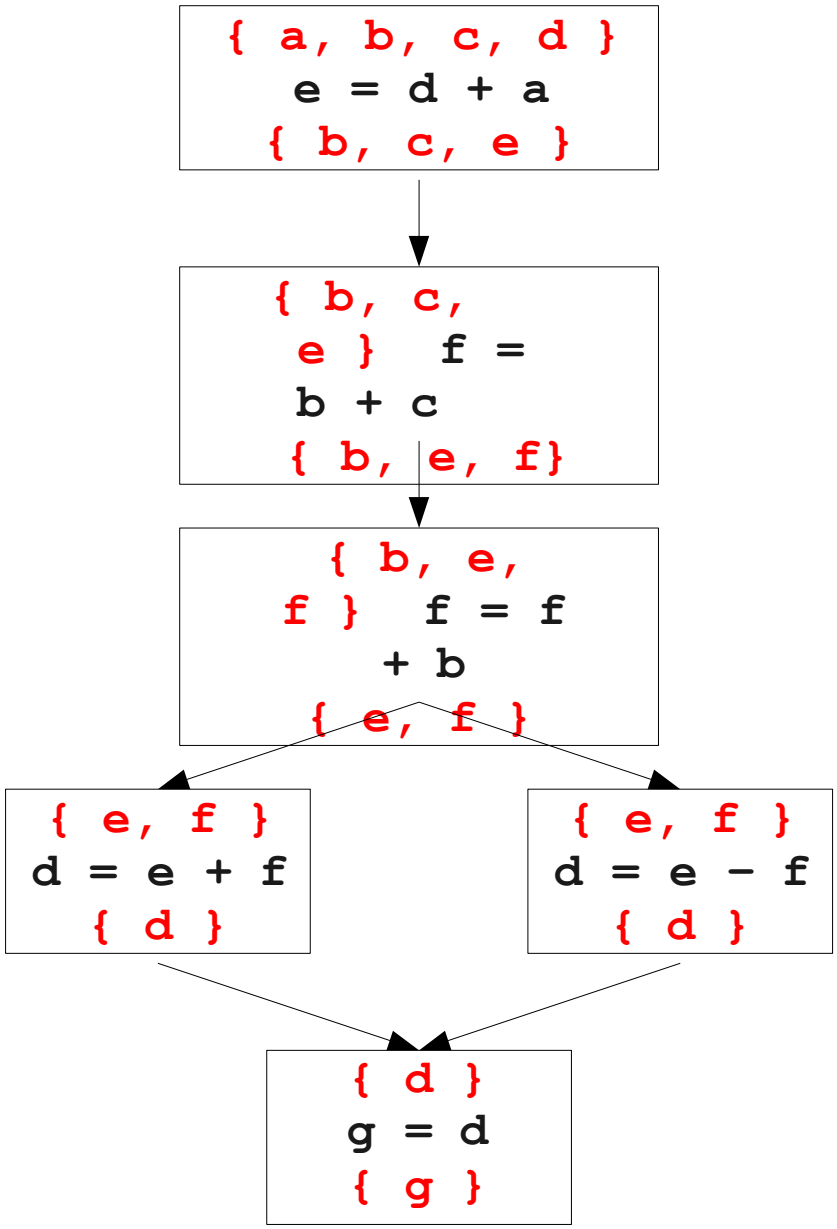
Live Ranges and Live Intervals



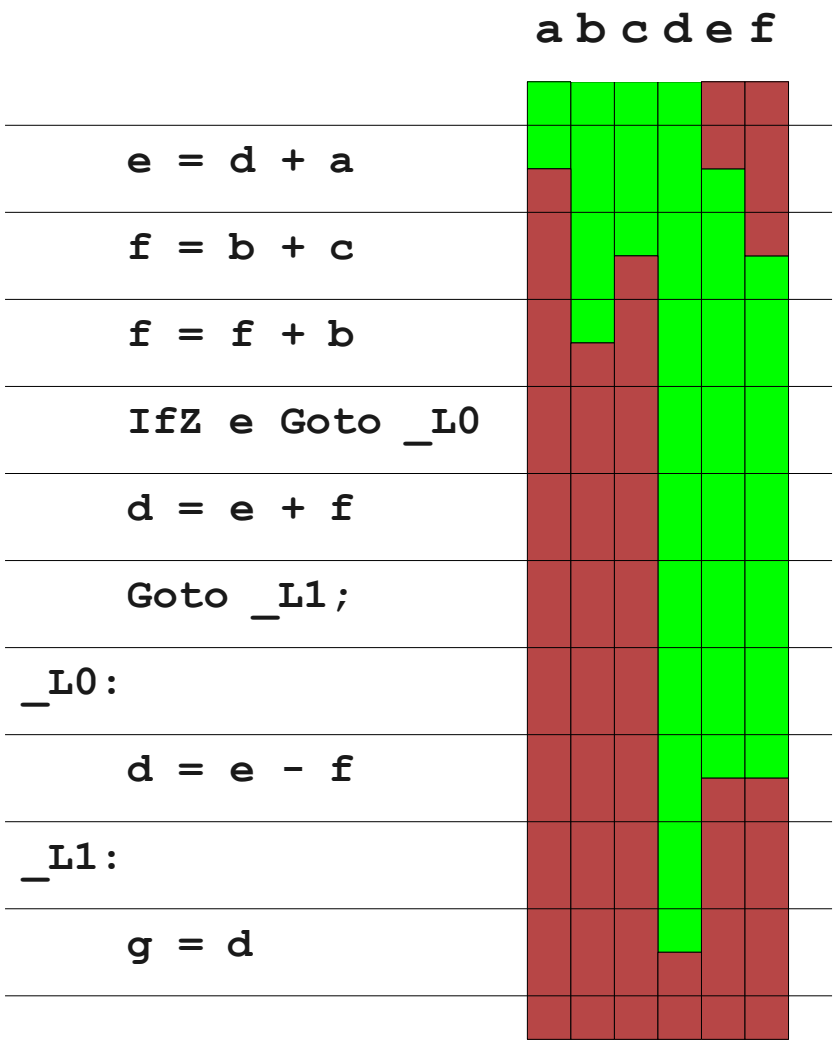


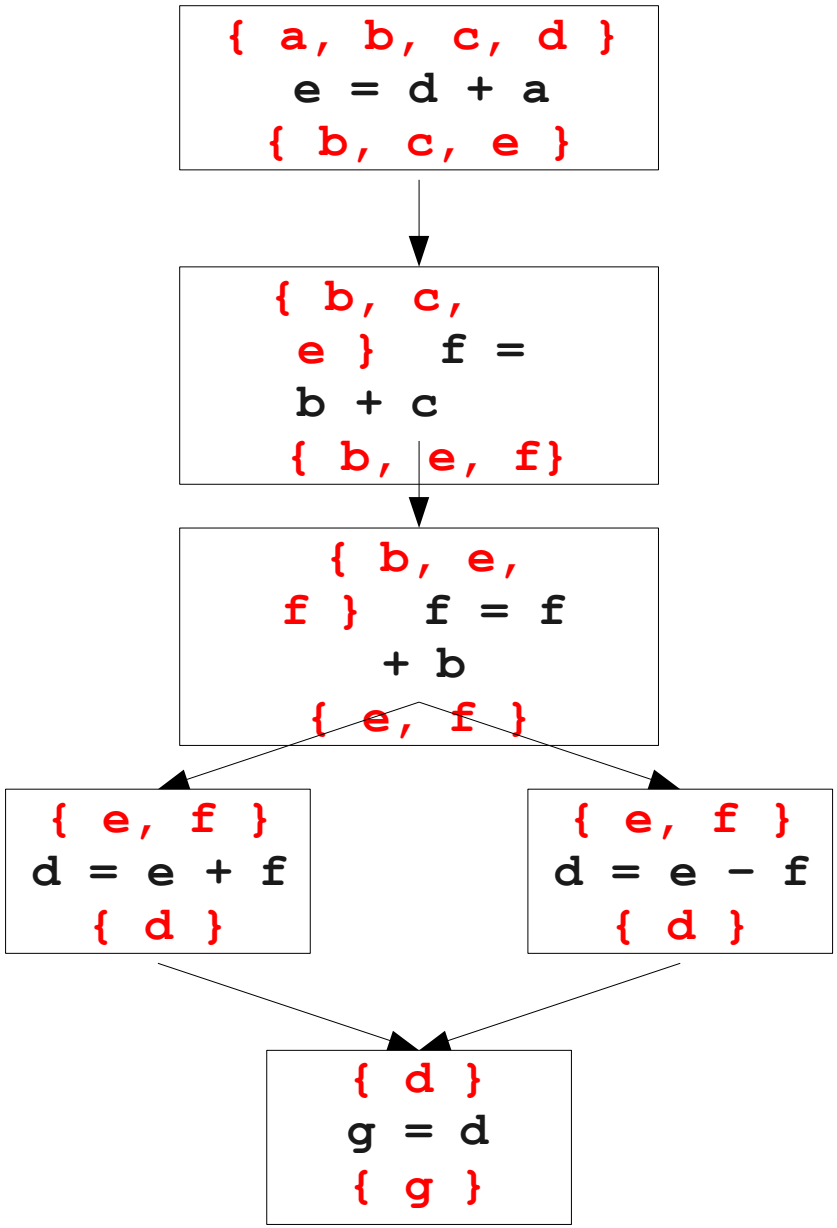
Live Ranges and Live Intervals





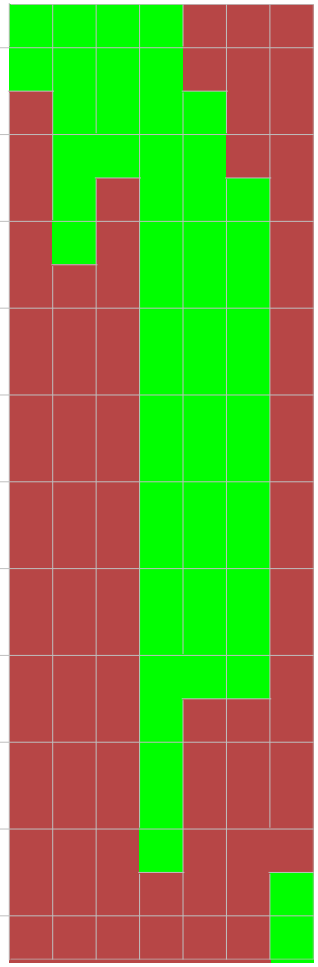
Live Ranges and Live Intervals

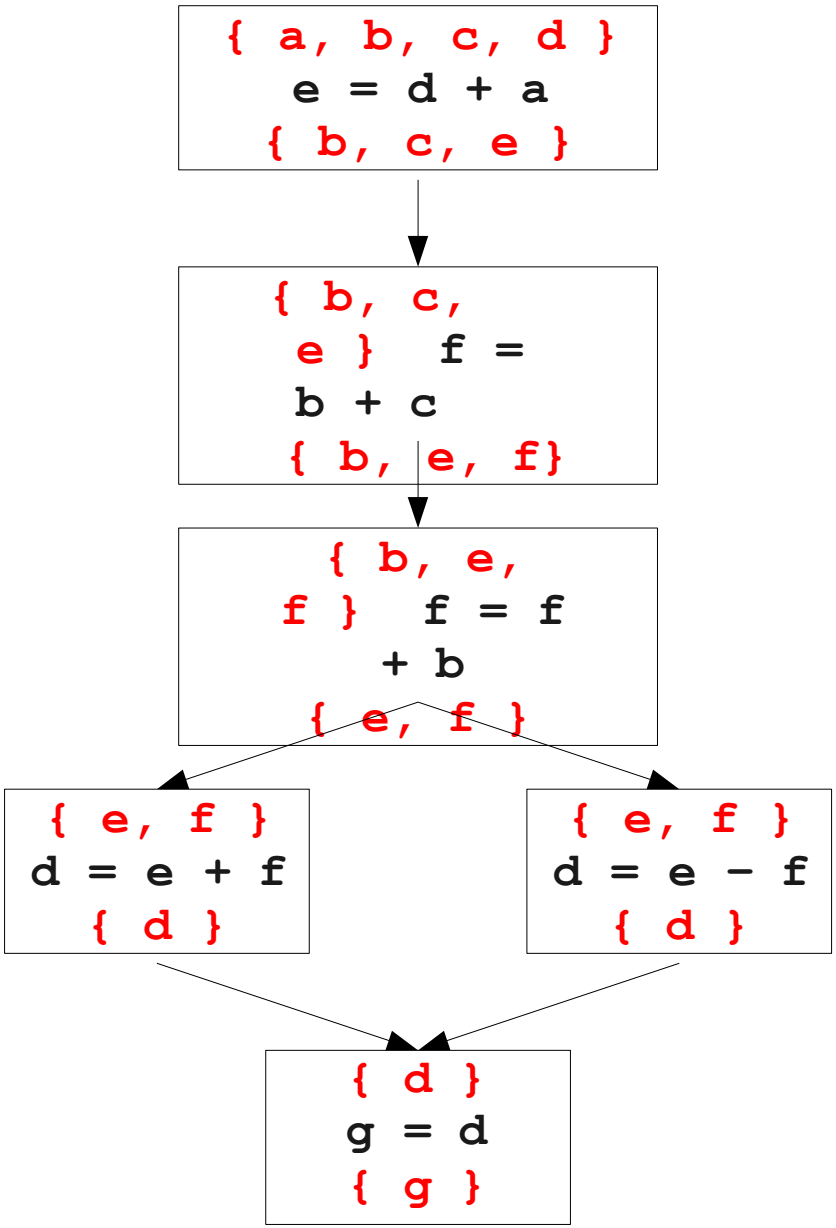




Live Ranges and Live Intervals

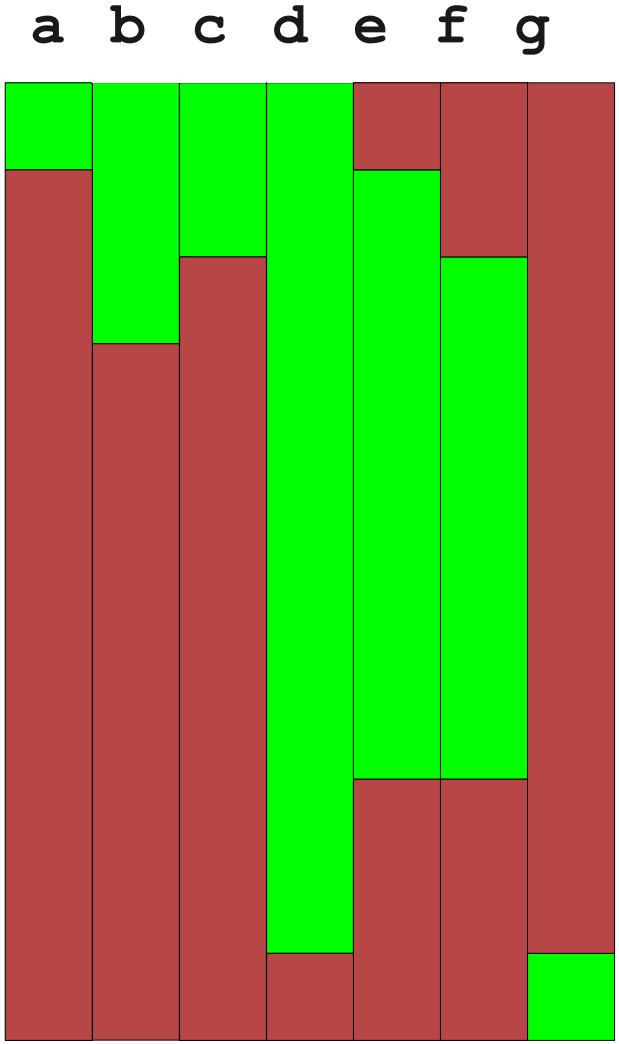
```
        e = d + a
        f = b + c
        f = f + b
        IfZ e Goto _L0
        d = e + f
        Goto _L1;
_L0:
        d = e - f
_L1:
        g = d
```



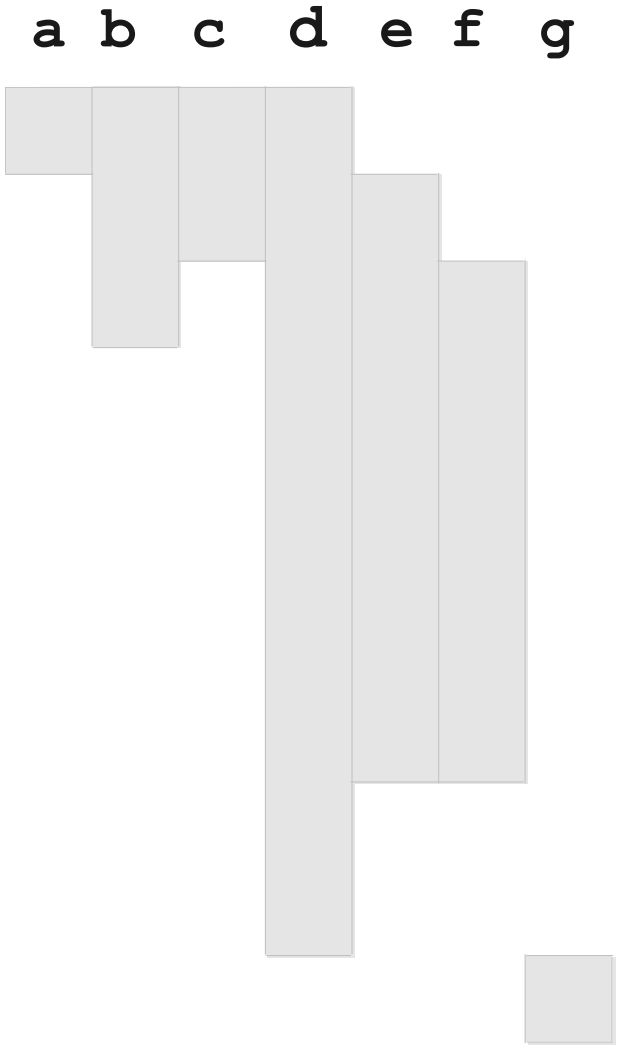


Register Allocation with Live Intervals

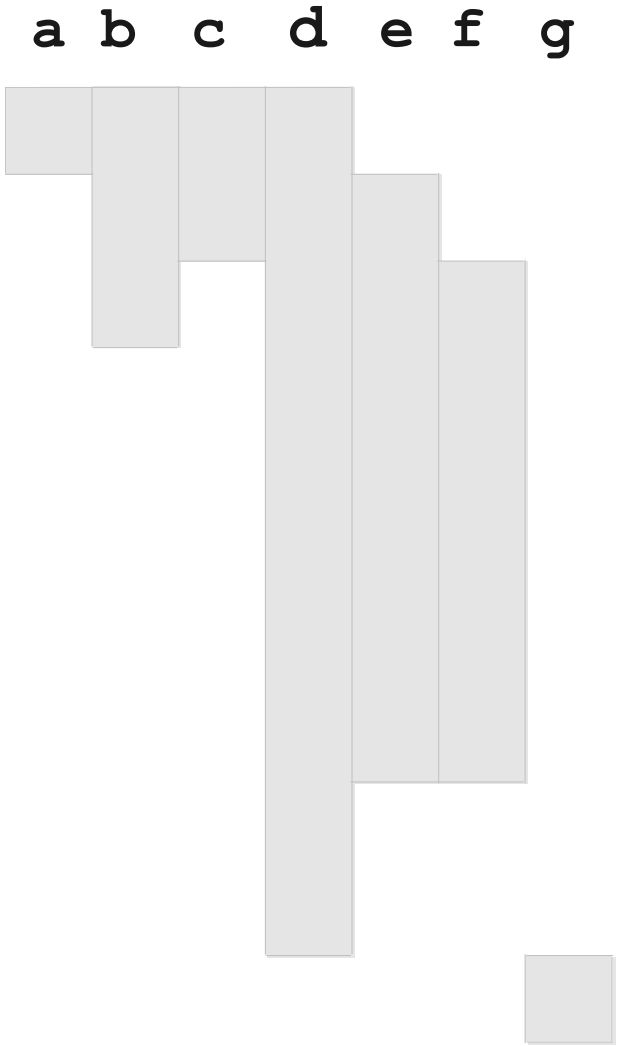
- Given the live intervals for all the variables in the program, we can allocate registers using a simple greedy algorithm.
- Idea: Track which registers are free at each point.
- When a live interval begins, give that variable a free register.
- When a live interval ends, the register is once again free.
- We can't always fit everything into a register; we'll see what do to in a minute.



Register Allocation with Live Intervals

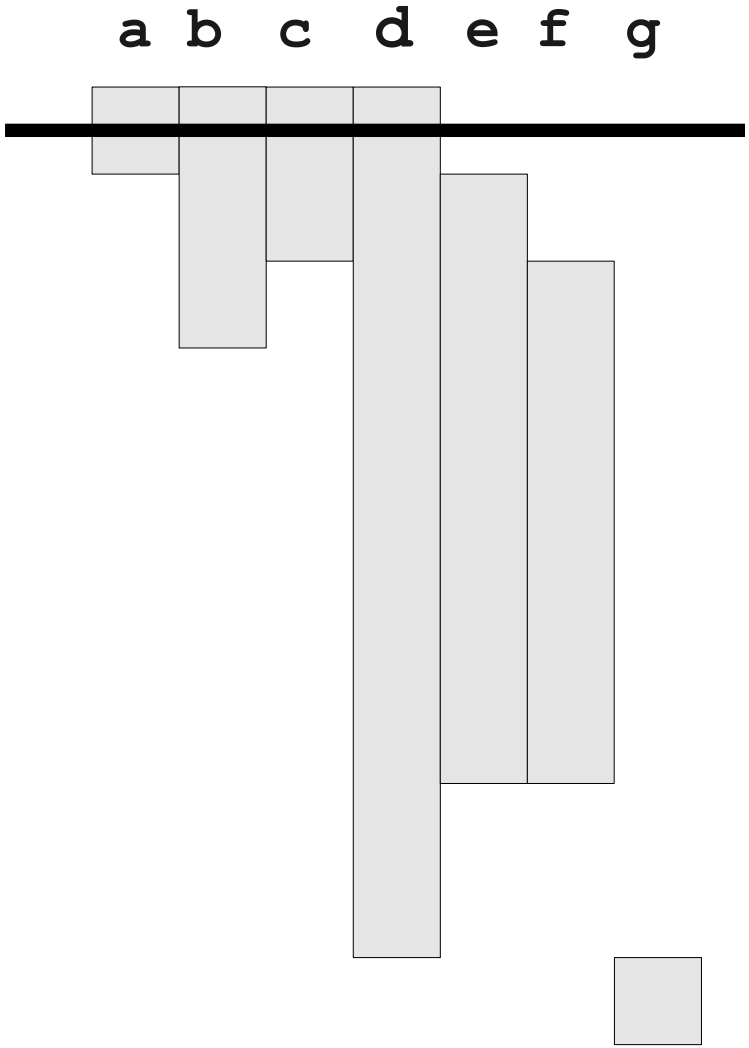


Register Allocation with Live Intervals



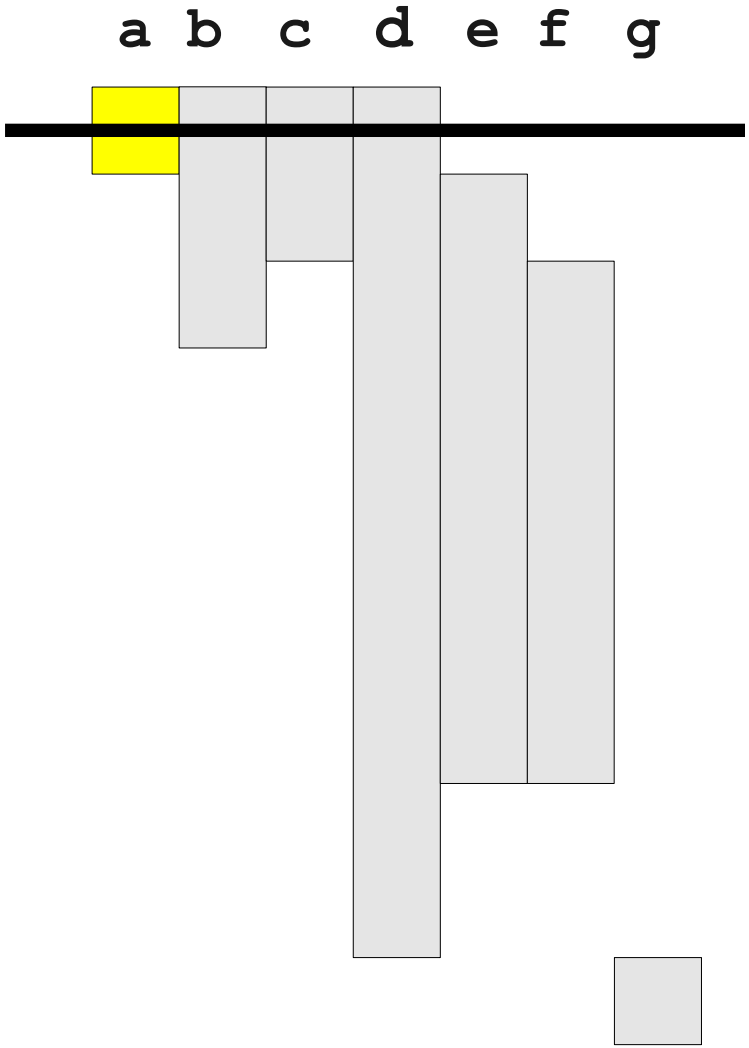
Free Registers

Register Allocation with Live Intervals



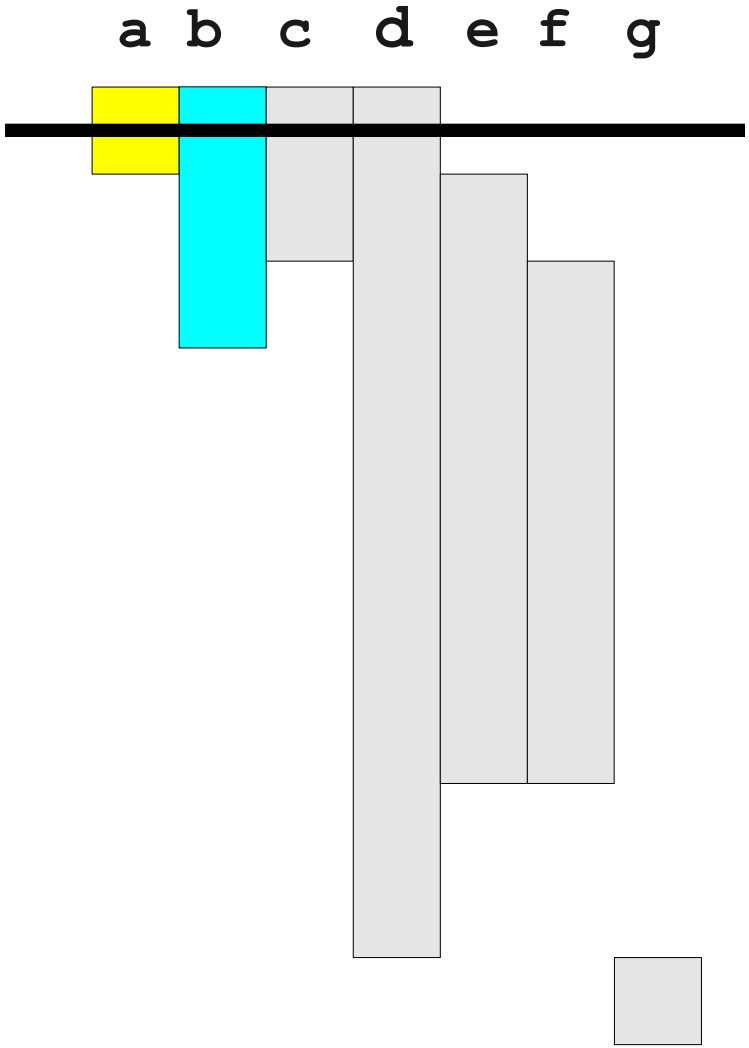
Free Registers

Register Allocation with Live Intervals



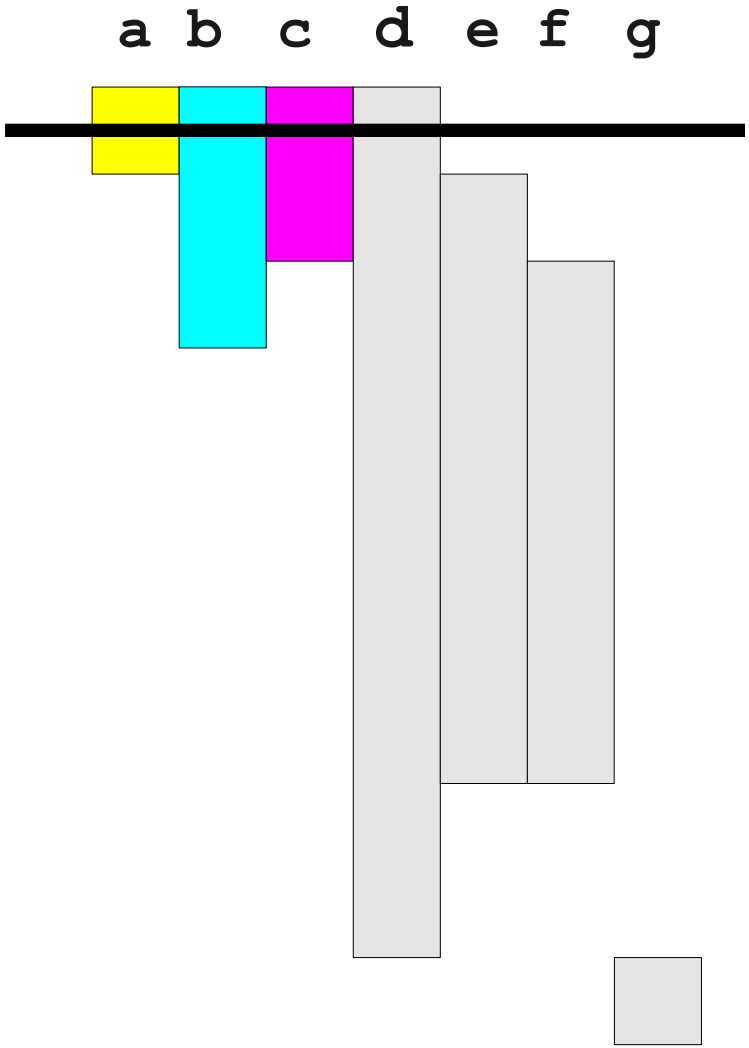
Free Registers

Register Allocation with Live Intervals



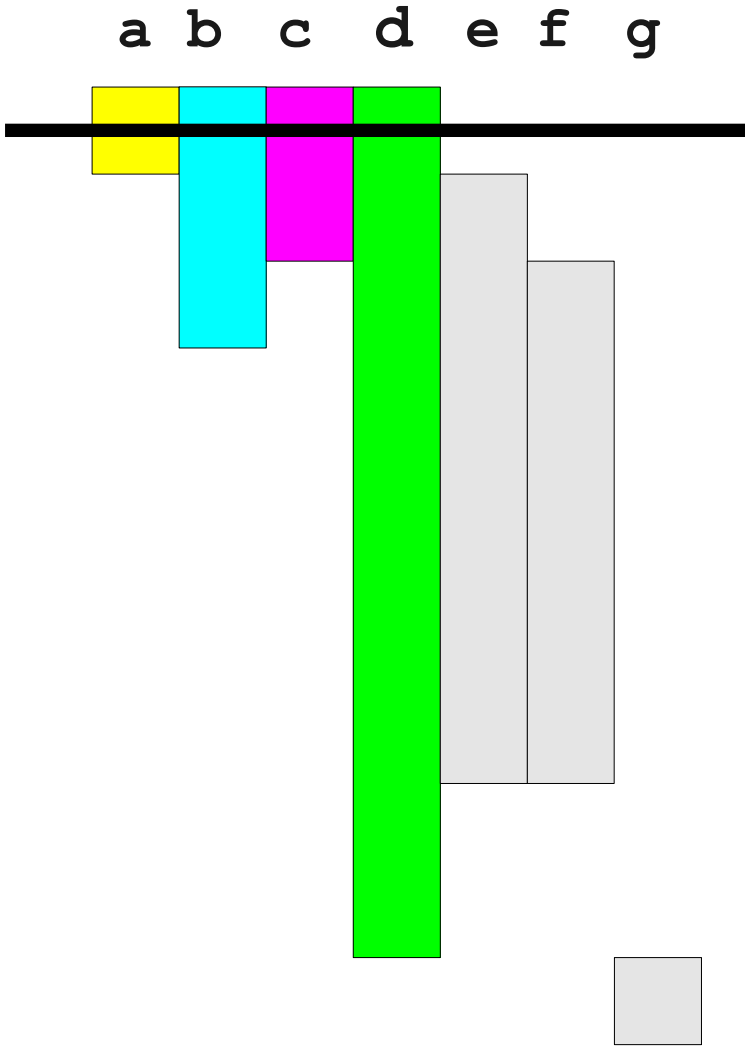
Free Registers

Register Allocation with Live Intervals



Free Registers

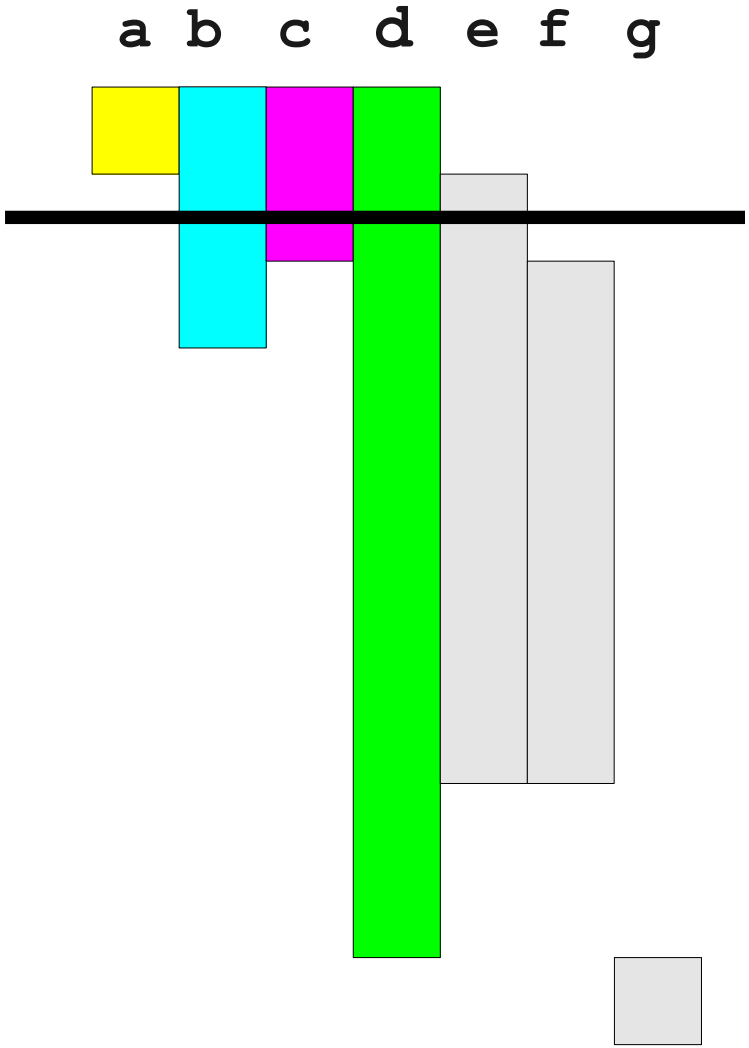
Register Allocation with Live Intervals



Free Registers

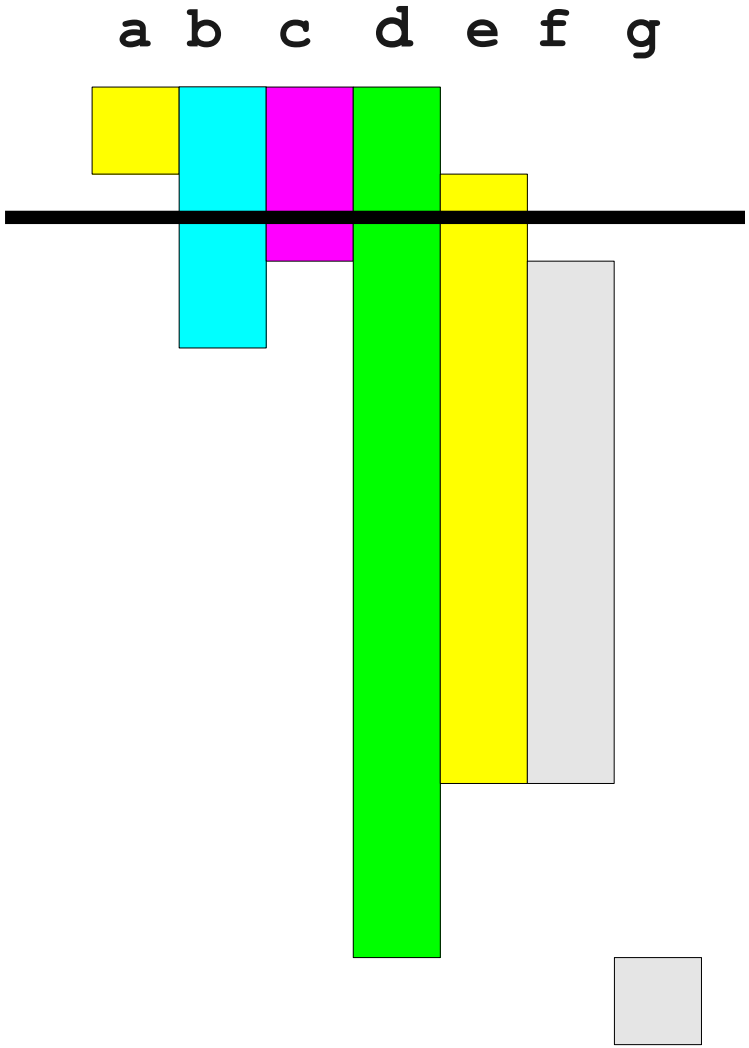
R_0	R_1	R_2	R_2

Register Allocation with Live Intervals



Free Registers

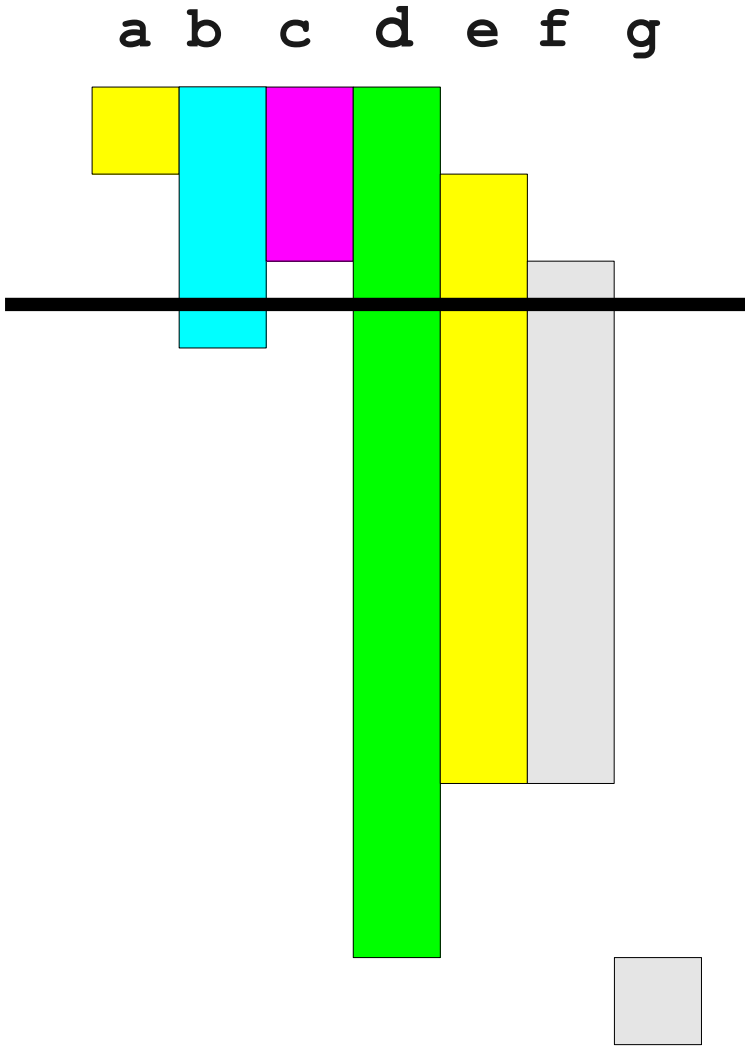
Register Allocation with Live Intervals



Free Registers

R_0	R_1	R_2	R_2

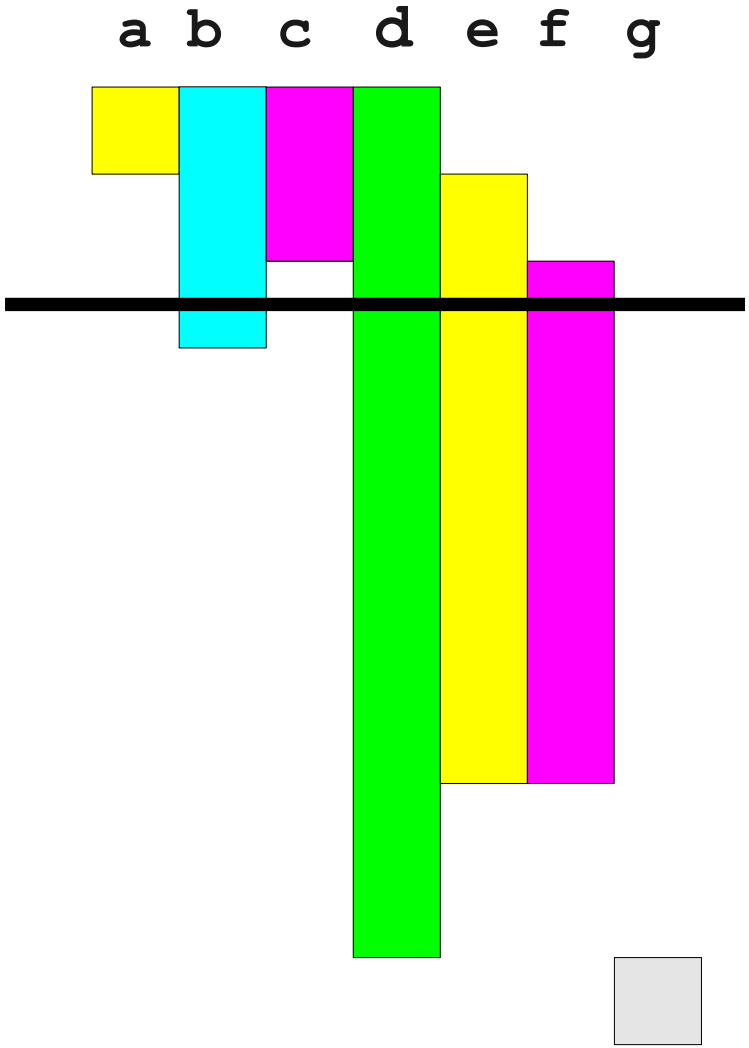
Register Allocation with Live Intervals



Free Registers

R_0	R_1	R_2	R_2

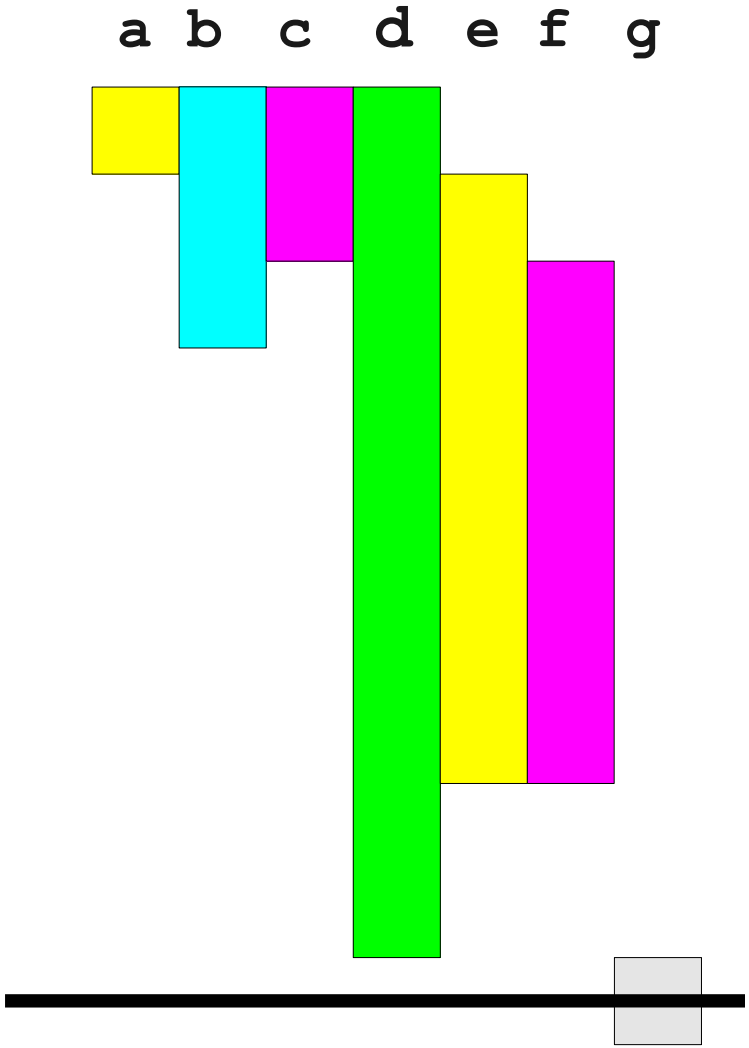
Register Allocation with Live Intervals



Free Registers

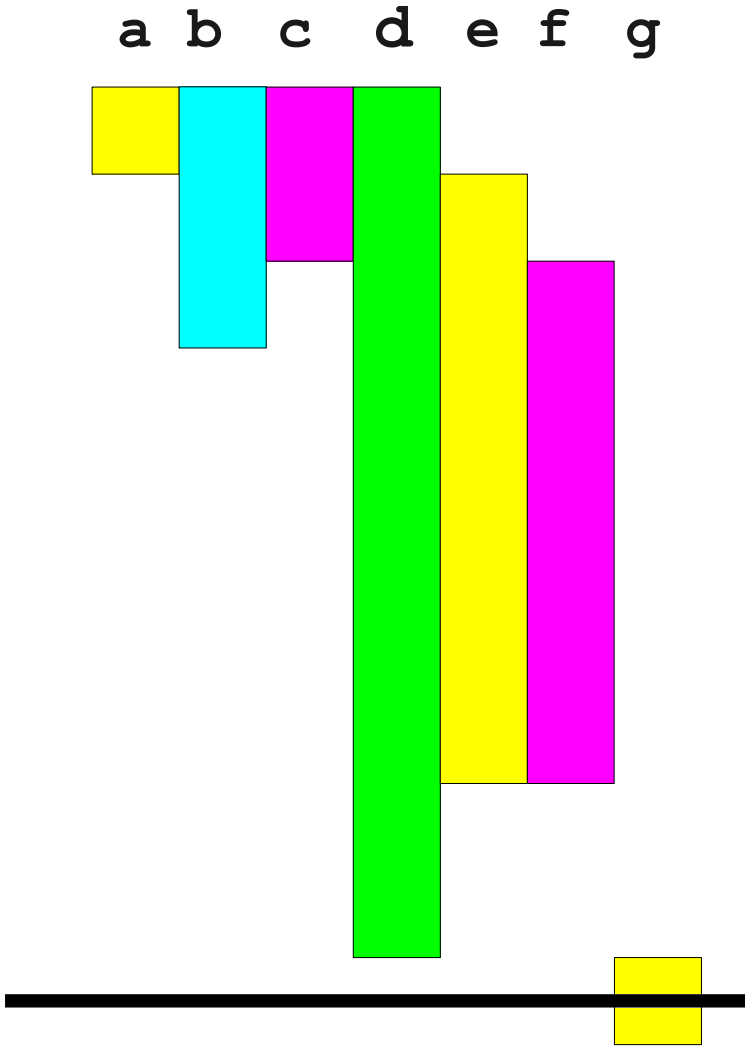
R_0	R_1	R_2	R_2

Register Allocation with Live Intervals



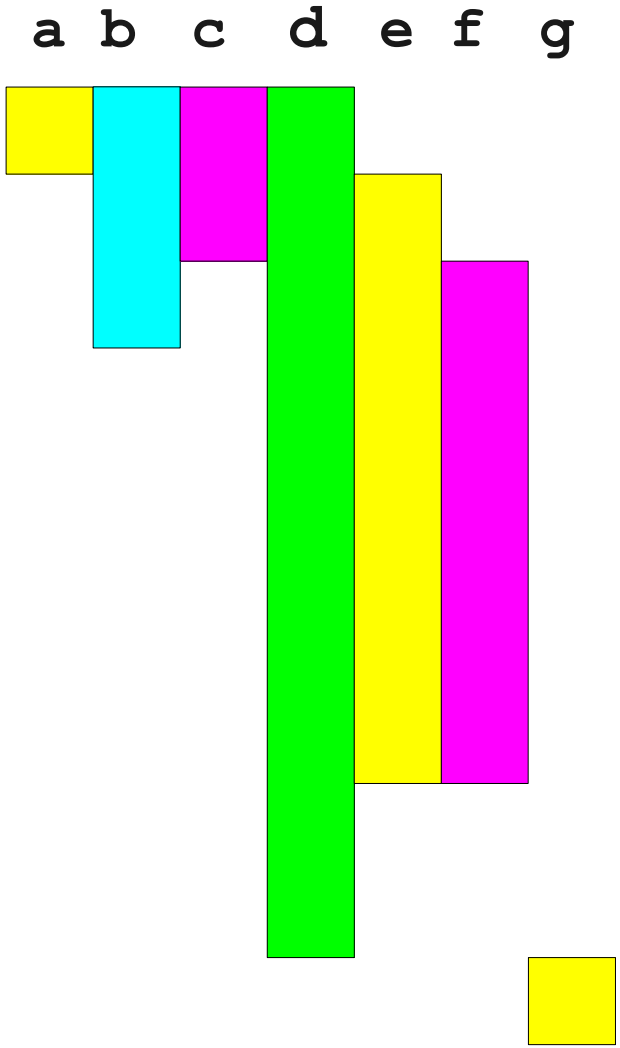
Free Registers

Register Allocation with Live Intervals



Free Registers

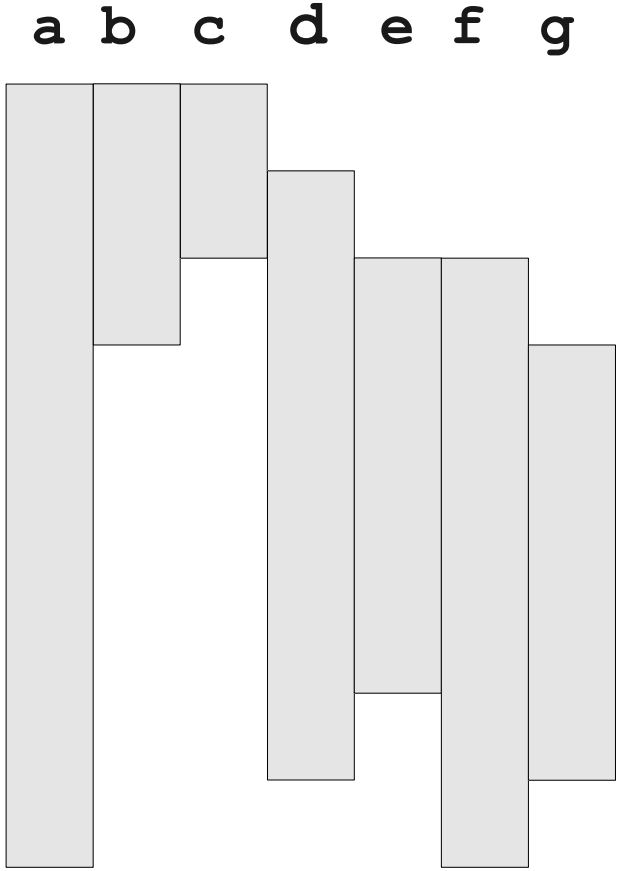
Register Allocation with Live Intervals



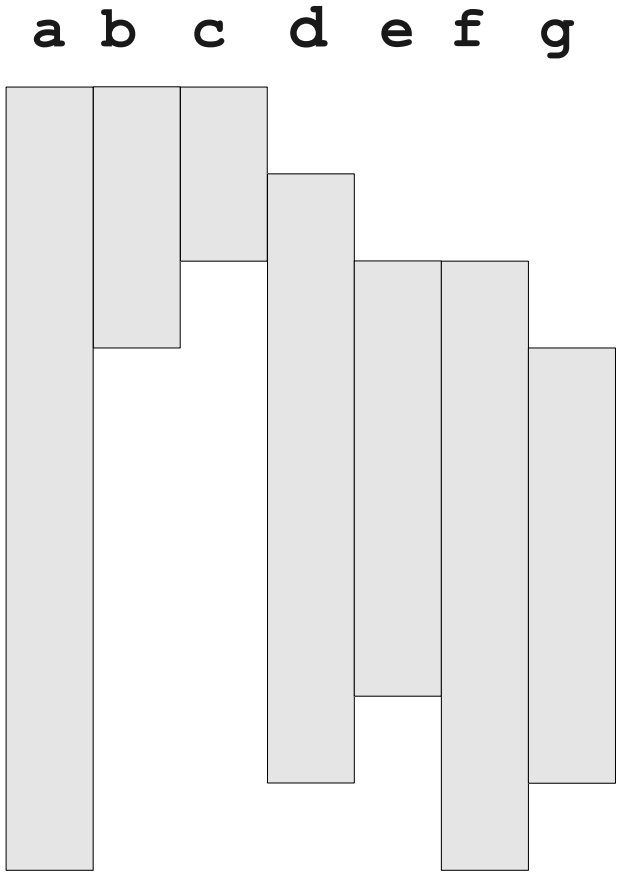
Free Registers

Another Example

Another Example

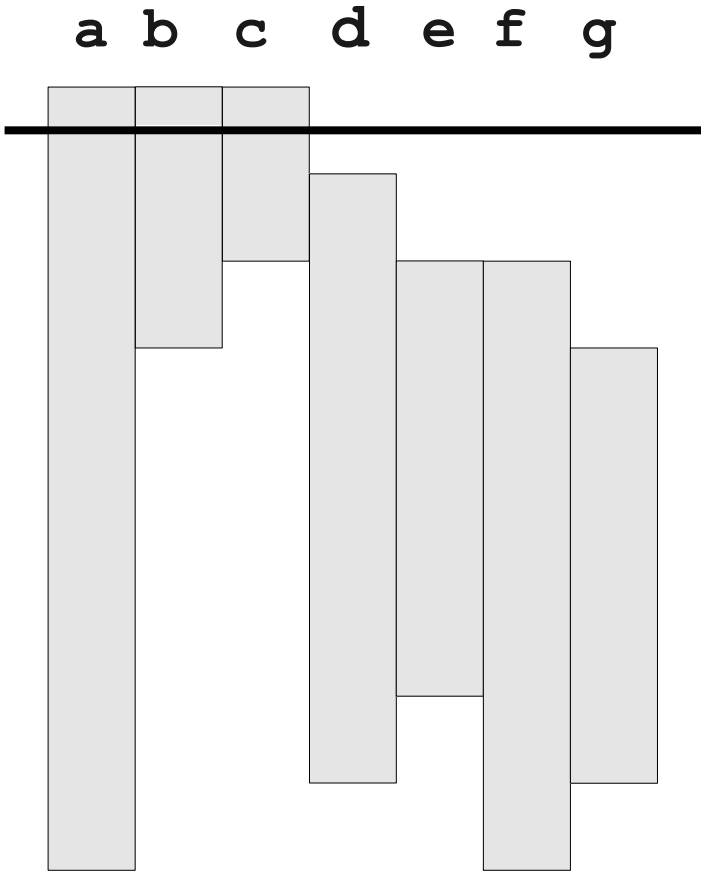


Another Example



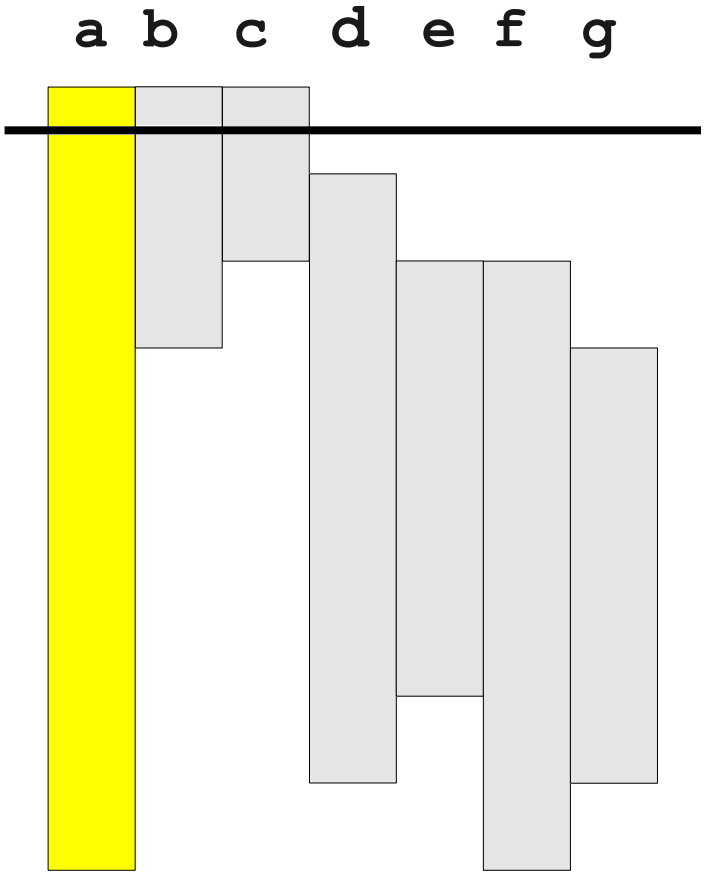
Free Registers

Another Example



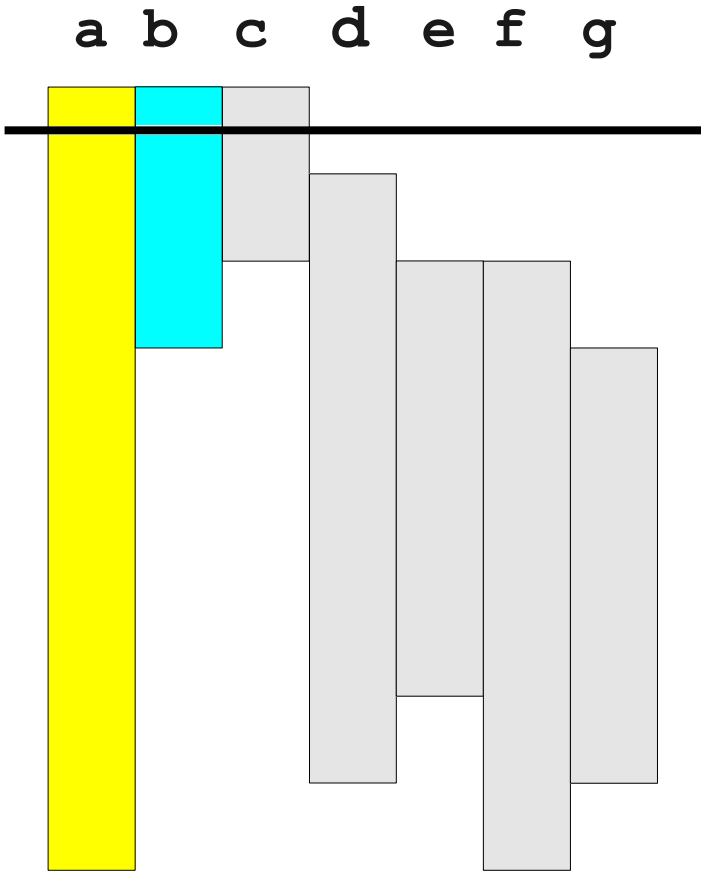
Free Registers

Another Example



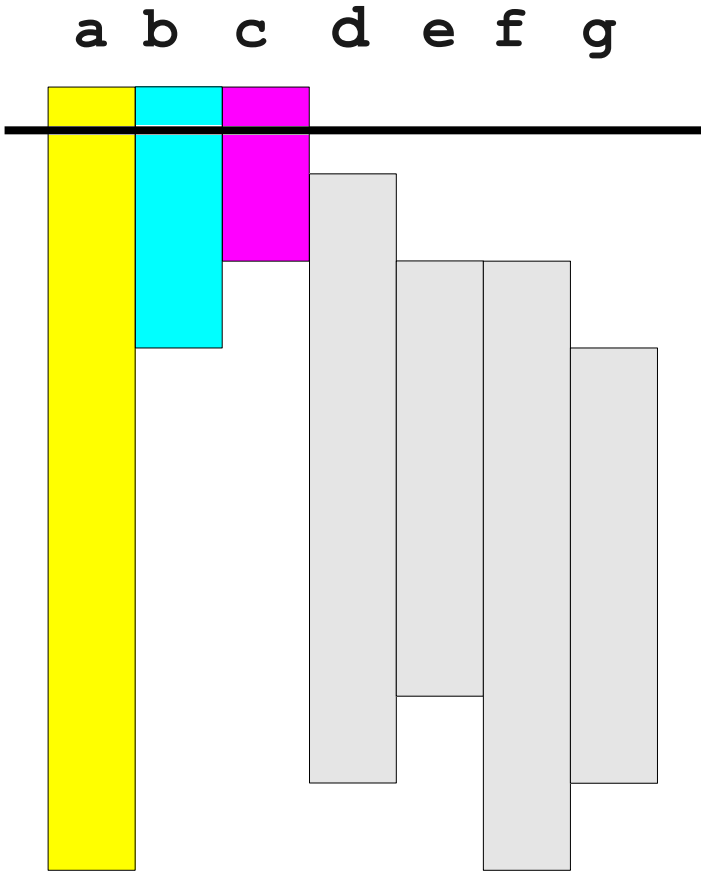
Free Registers

Another Example



Free Registers

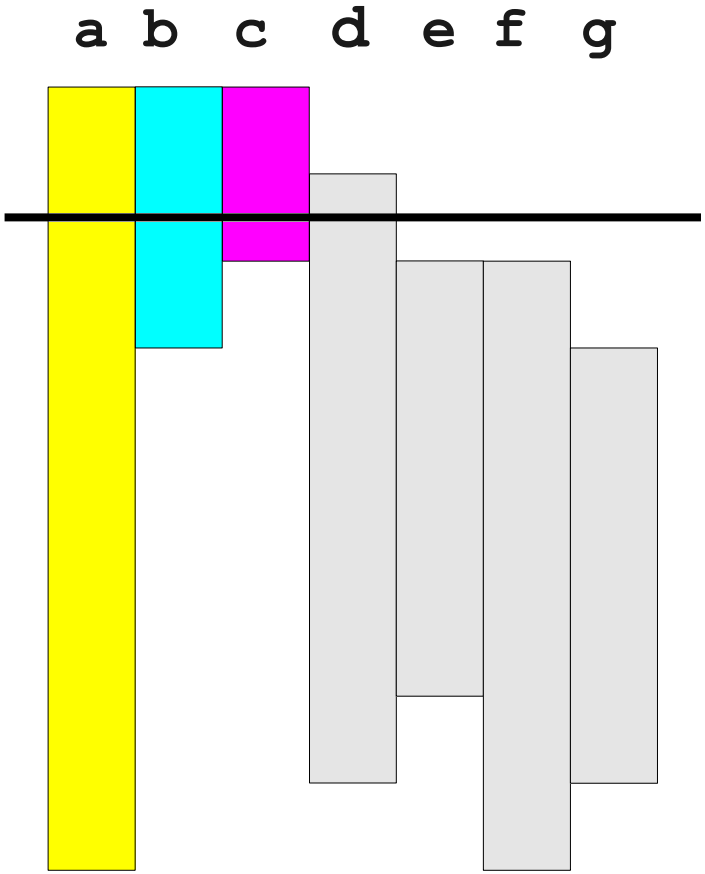
Another Example



Free Registers

R_0	R_1	R_2

Another Example



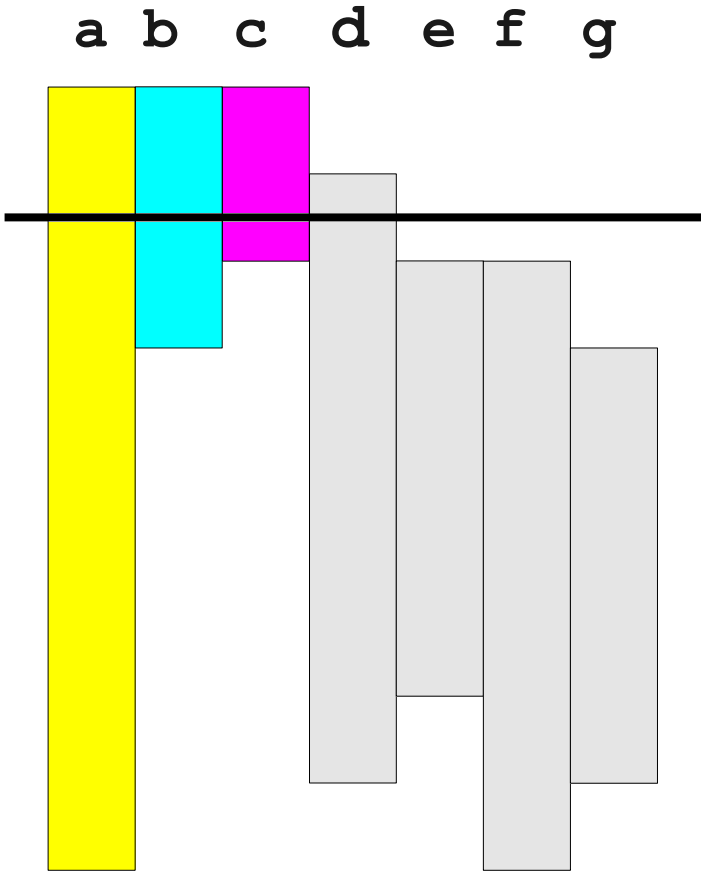
Free Registers

What do we do now?

Register Spilling

- If a register cannot be found for a variable *v*, we may need to **spill** a variable.
- When a variable is spilled, it is stored in memory rather than a register.
- When we need a register for the spilled variable:
  - Evict some existing register to memory.
  - Load the variable into the register.
  - When done, write the register back to memory and reload the register with its original value.
- Spilling is slow, but sometimes necessary.

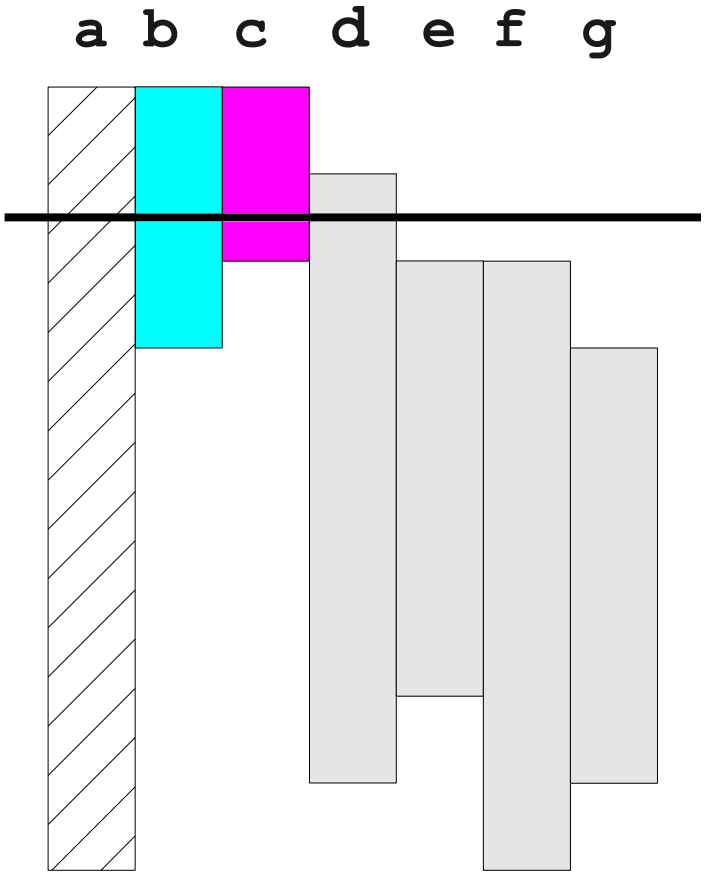
Another Example



Free Registers

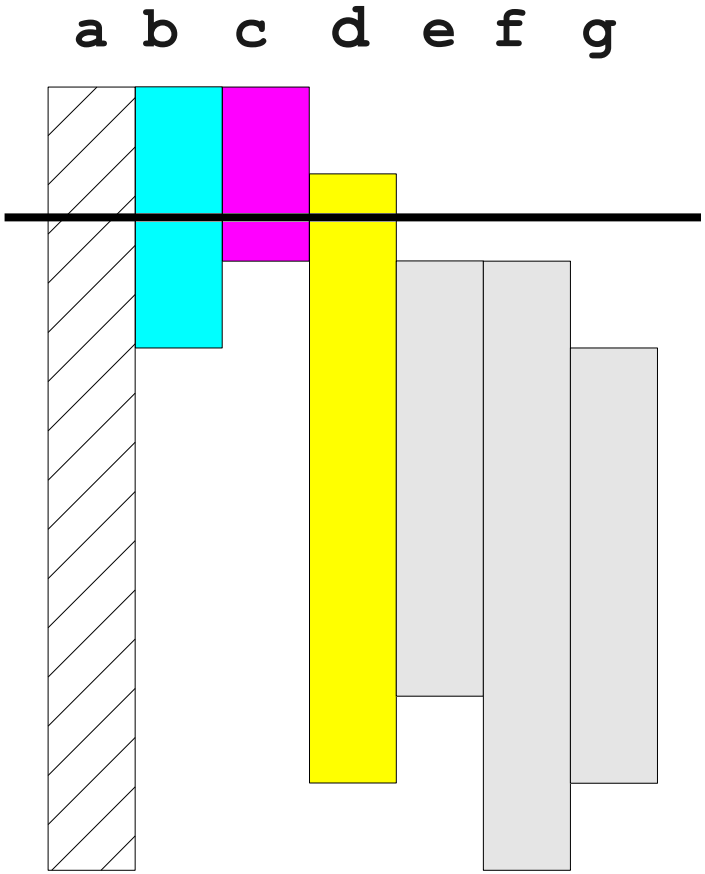
What do we do now?

Another Example



Free Registers

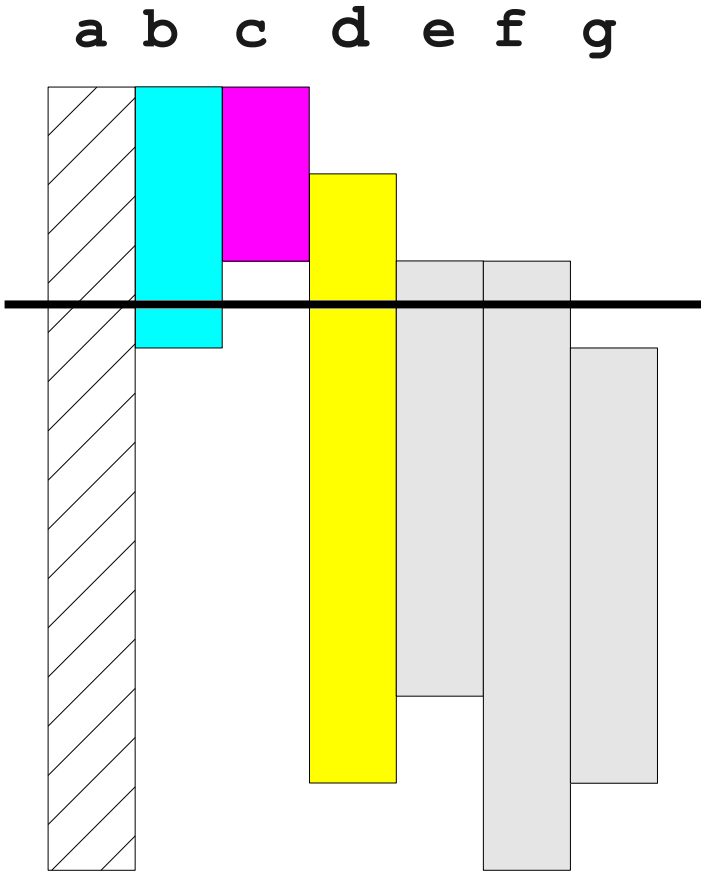
Another Example



Free Registers

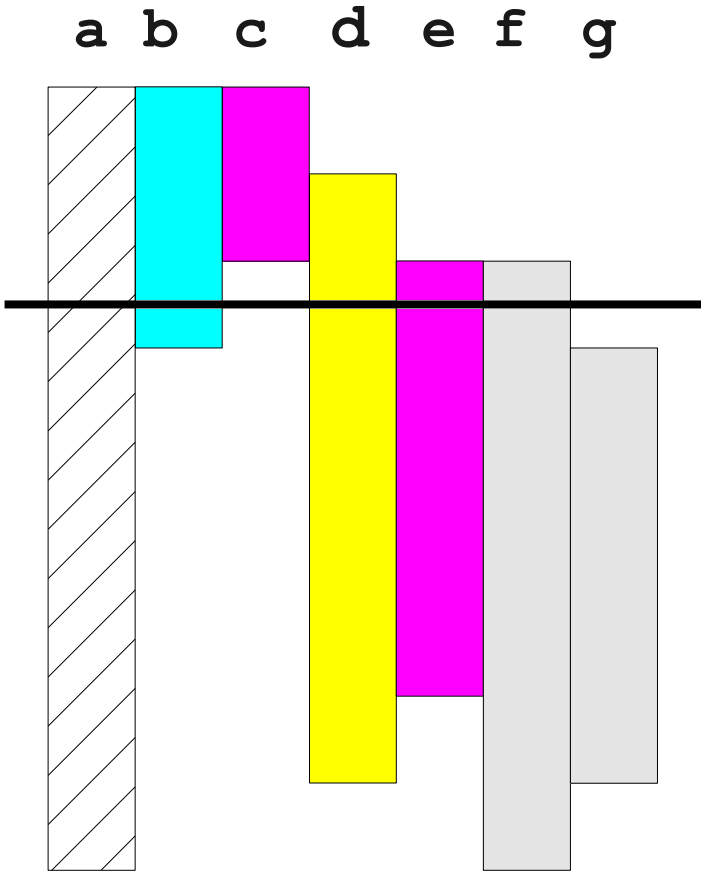
R_0	R_1	R_2

Another Example



Free Registers

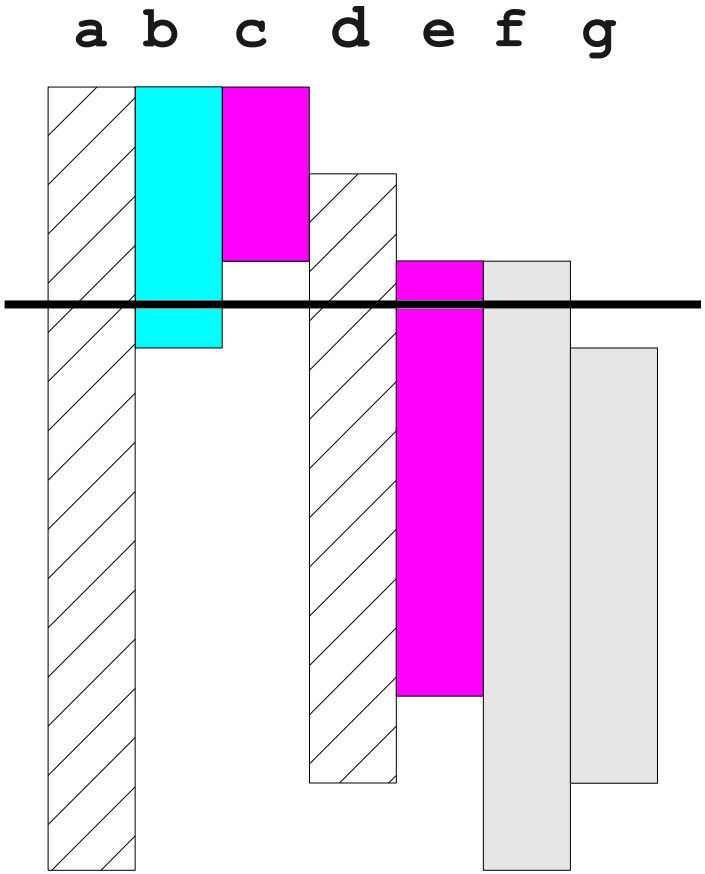
Another Example



Free Registers

R_0	R_1	R_2

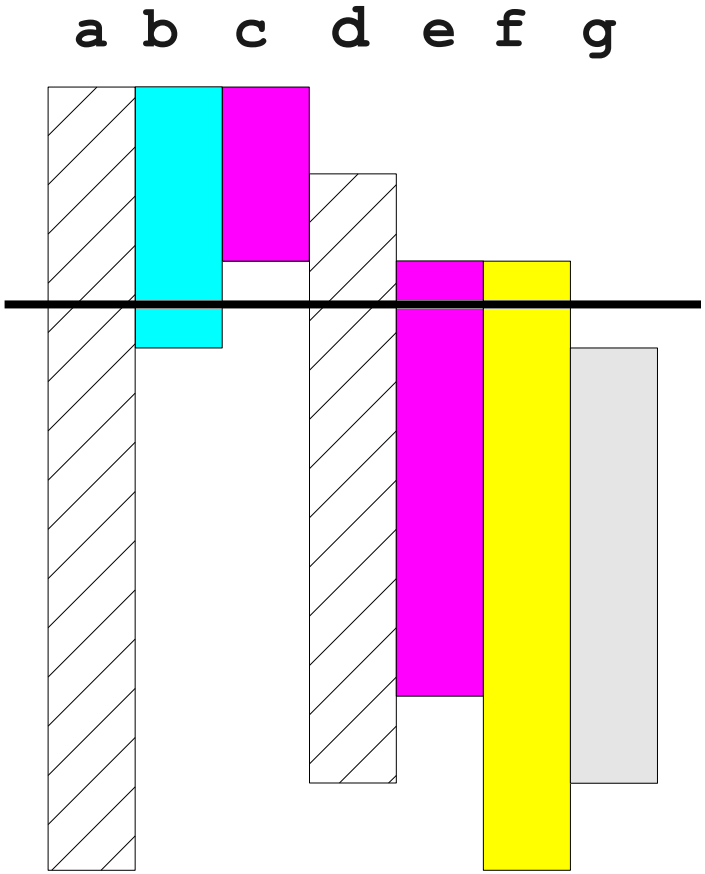
Another Example



Free Registers

R_0	R_1	R_2

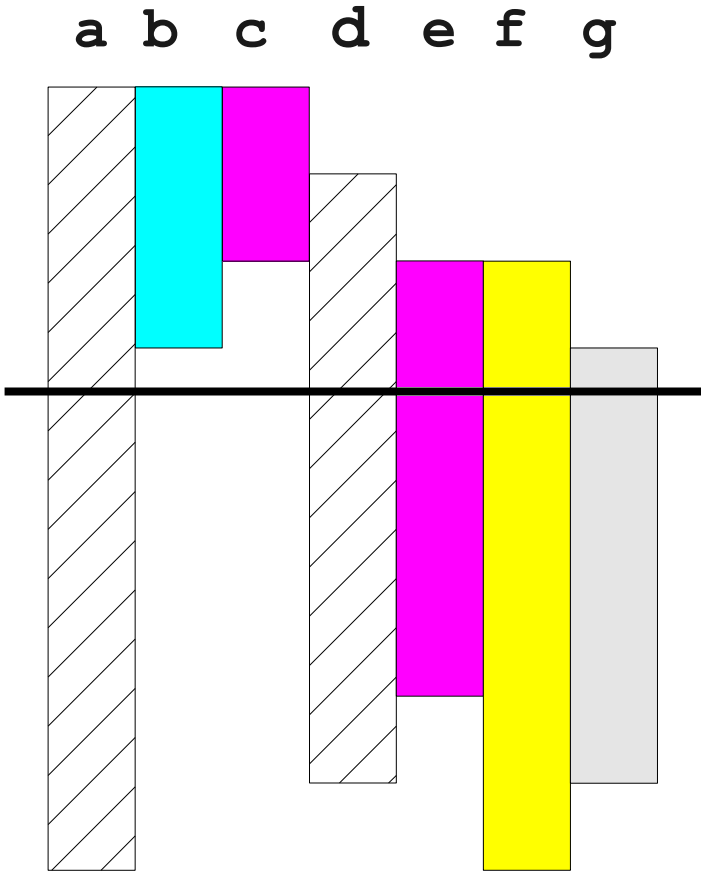
Another Example



Free Registers

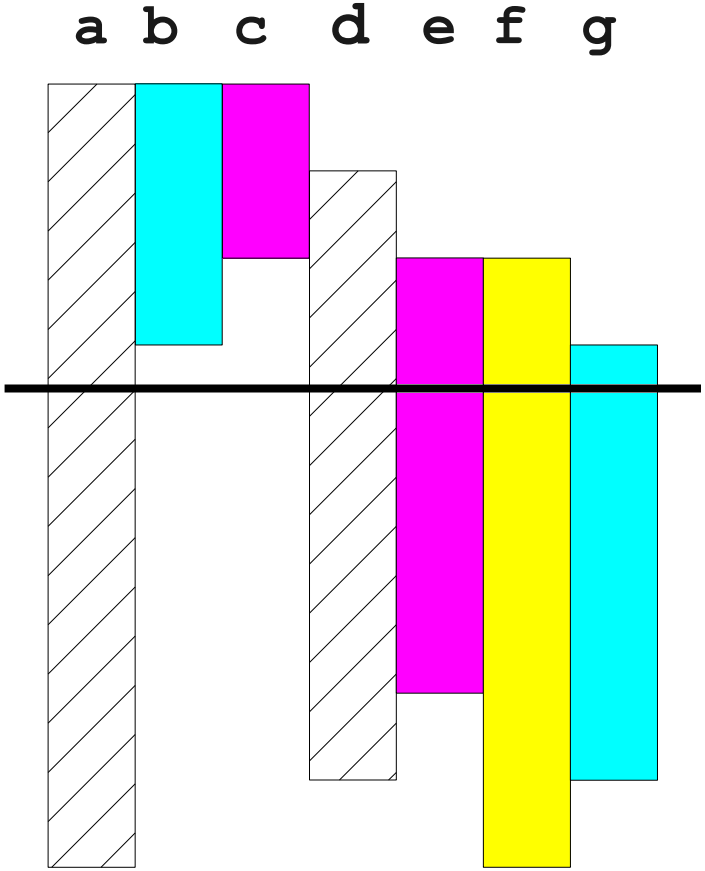
R_0	R_1	R_2

Another Example



Free Registers

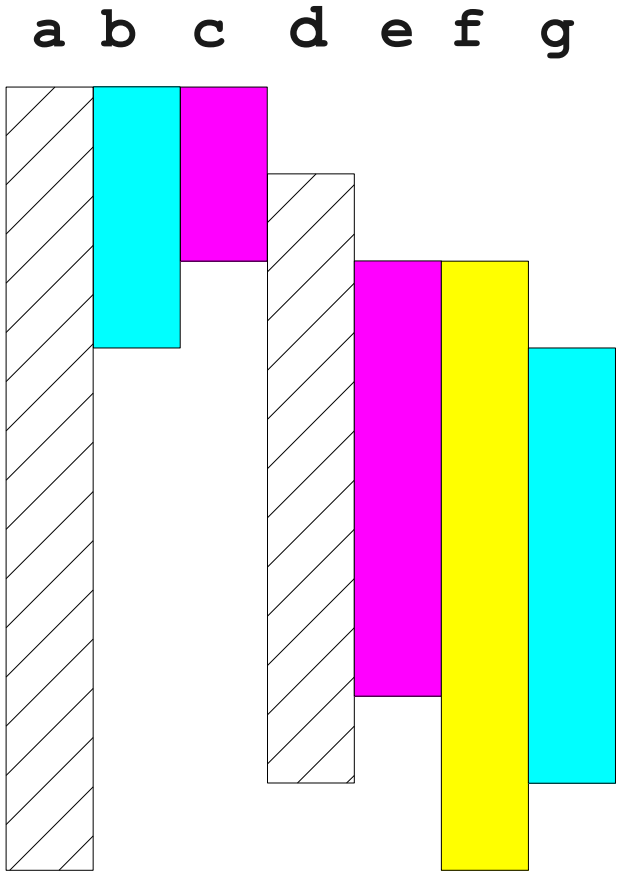
Another Example



Free Registers

R_0	R_1	R_2

Another Example



Free Registers

R_0	R_1	R_2

Linear Scan Register Allocation

- This algorithm is called **linear scan register allocation** and is a comparatively new algorithm.
- Advantages:
  - Very efficient (after computing live intervals, runs in linear time)
  - Produces good code in many instances.
  - Allocation step works in one pass; can generate code during iteration.
  - Often used in JIT compilers like Java HotSpot.
- Disadvantages:
  - Imprecise due to use of live **intervals** rather than live **ranges**.
  - Other techniques known to be superior in many cases.

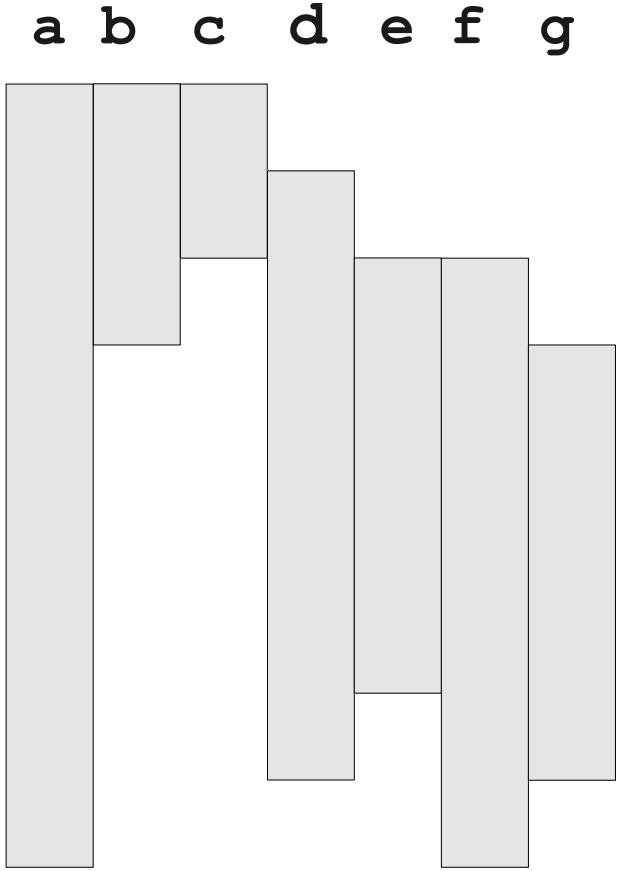
Spilling

* Keep it in memory (CISC vs RISC)
* Move back when necessary (?)

Second-Chance Bin Packing

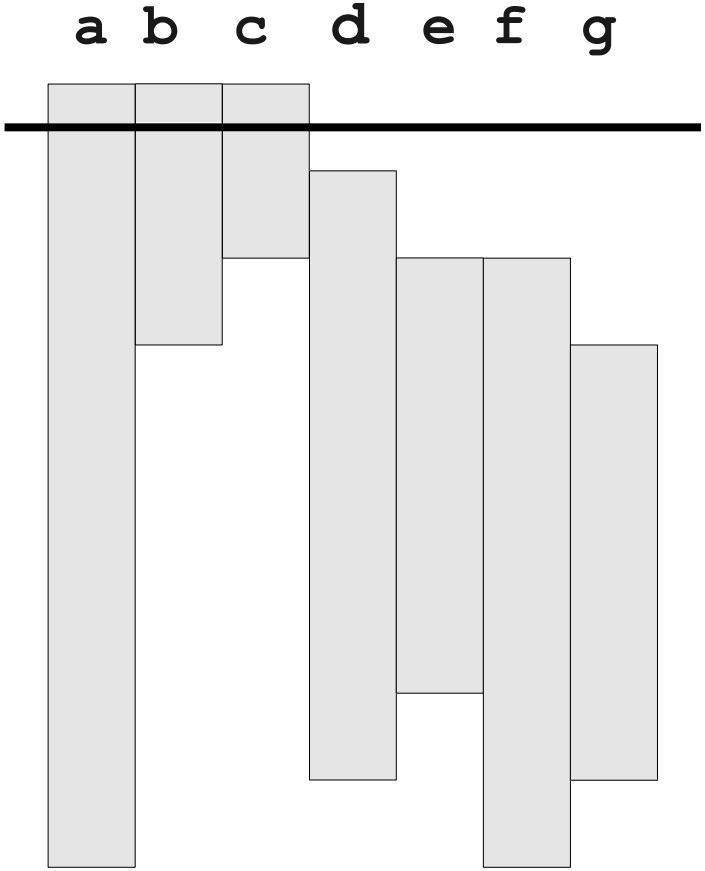
- A more aggressive version of linear-scan.
- Uses live **ranges** instead of live **intervals**.
- If a variable must be spilled, don't spill all uses of it.
  - A later live range might still fit into a register.
- Requires a final data-flow analysis to confirm variables are assigned consistent locations.
- See “Quality and Speed in Linear-scan Register Allocation” by Traub, Holloway, and Smith.

Second-Chance Bin Packing



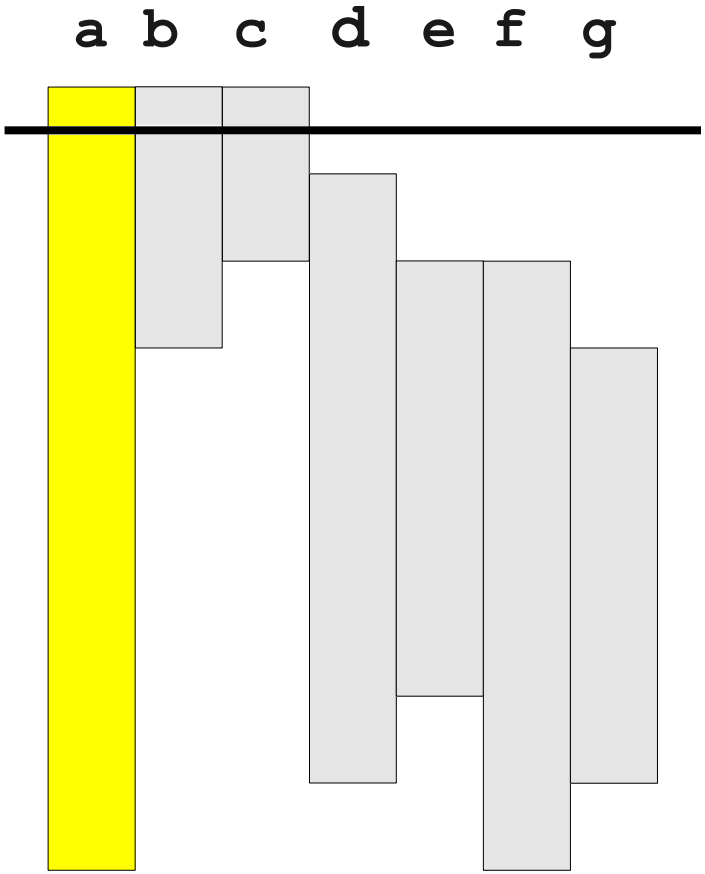
Free Registers

Second-Chance Bin Packing



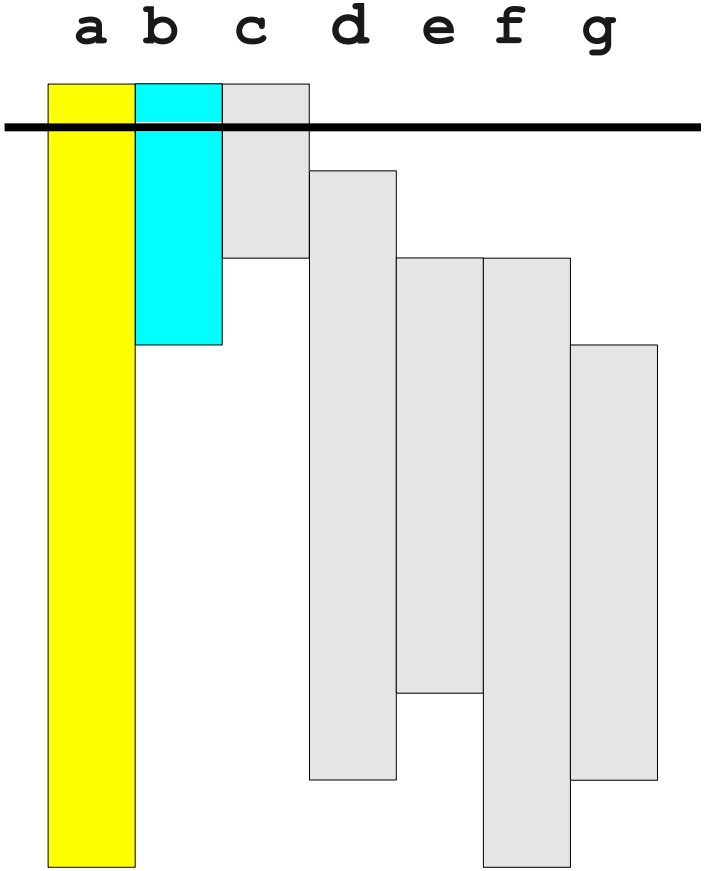
Free Registers

Second-Chance Bin Packing



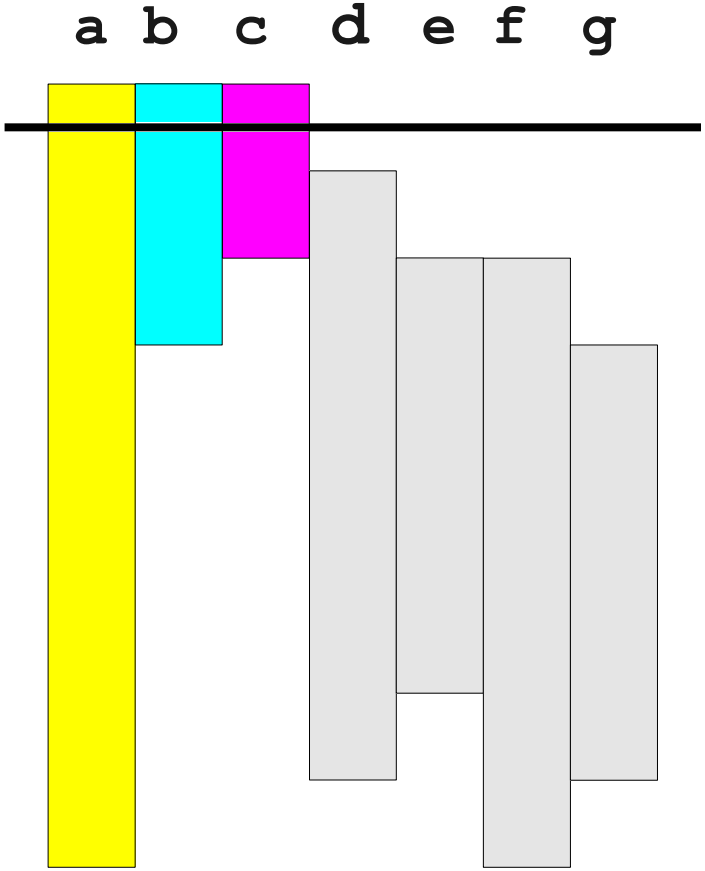
Free Registers

Second-Chance Bin Packing



Free Registers

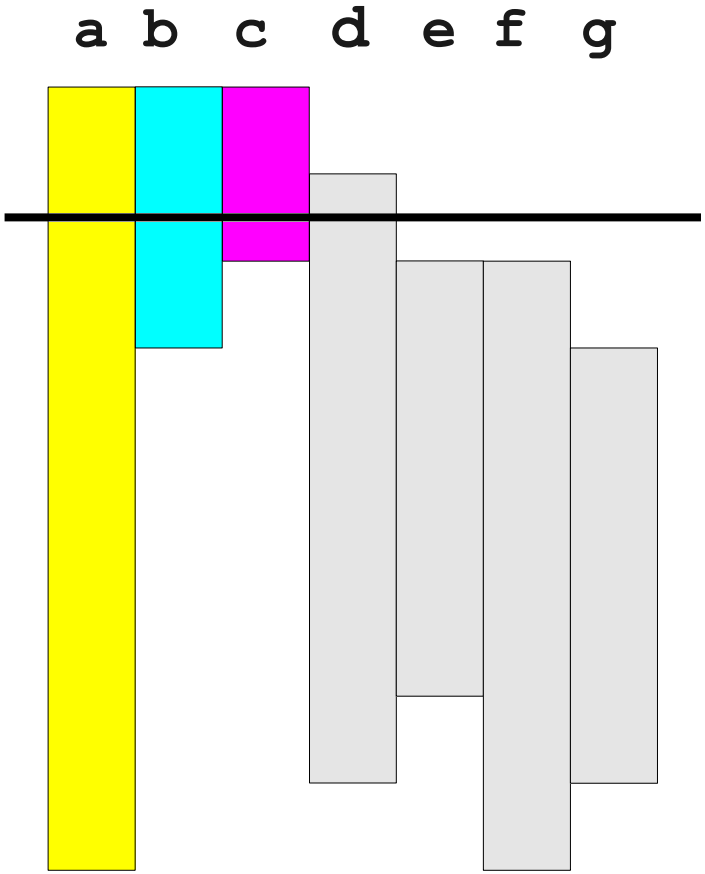
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

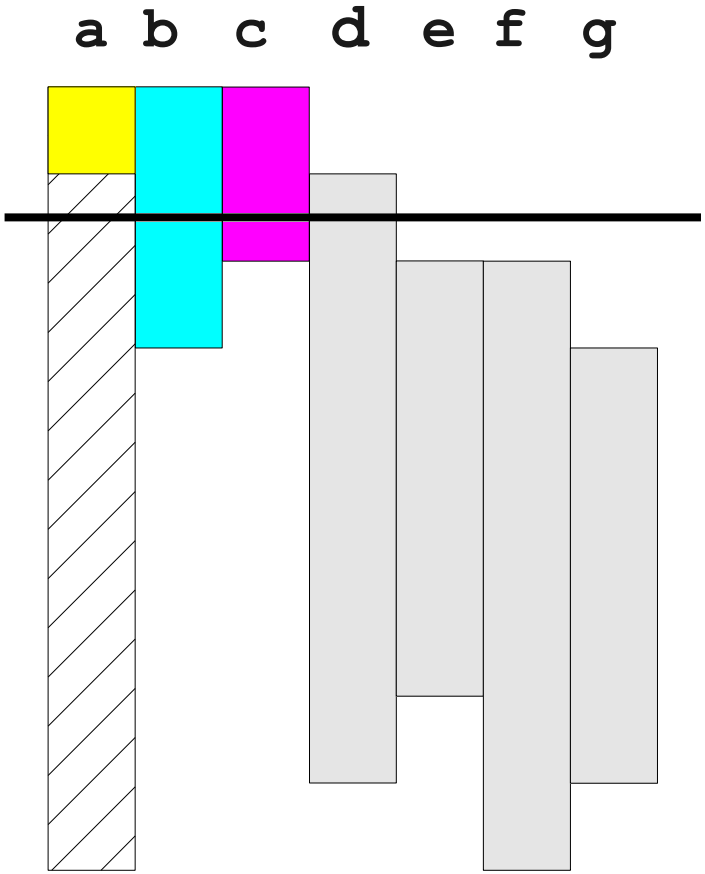
Second-Chance Bin Packing



Free Registers

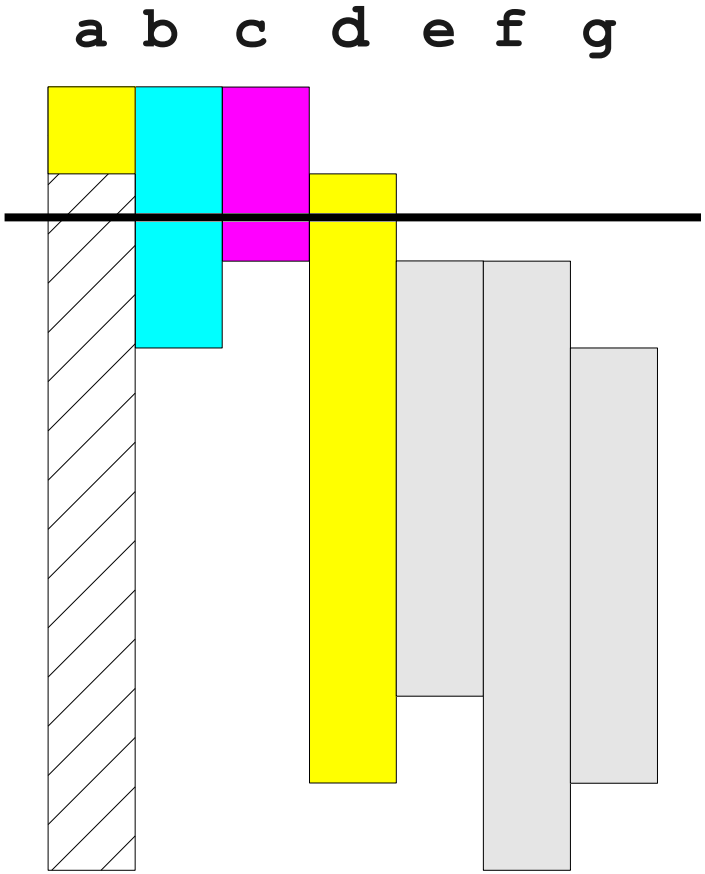
R_0	R_1	R_2

Second-Chance Bin Packing



Free Registers

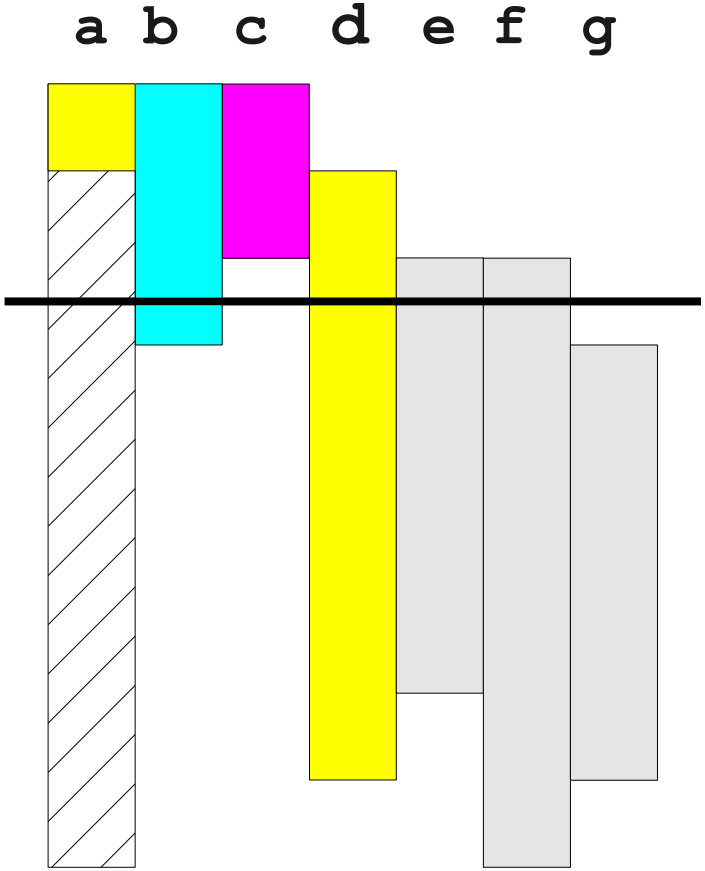
Second-Chance Bin Packing



Free Registers

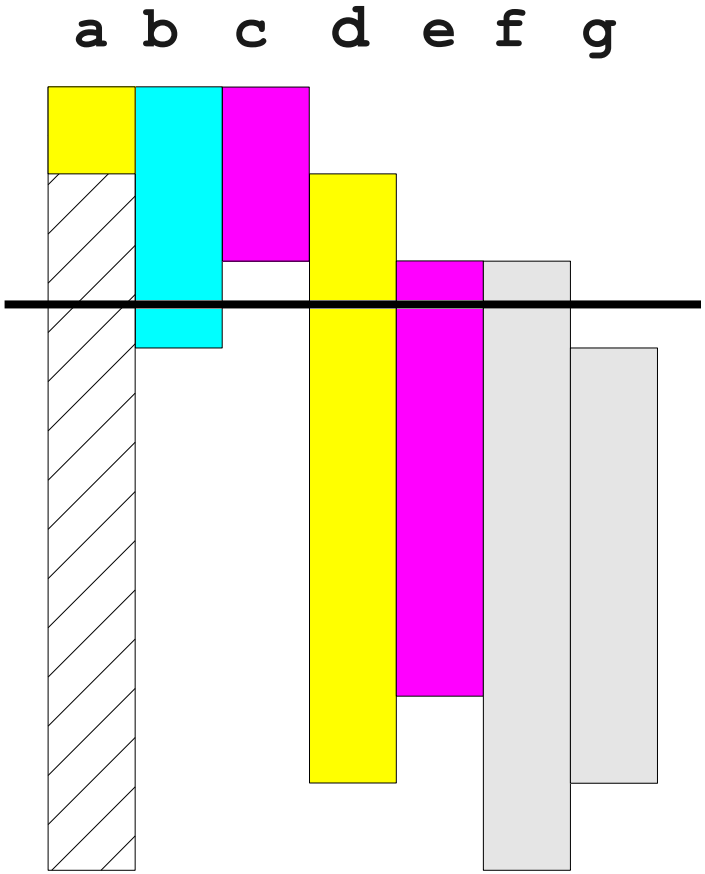
R_0	R_1	R_2

Second-Chance Bin Packing



Free Registers

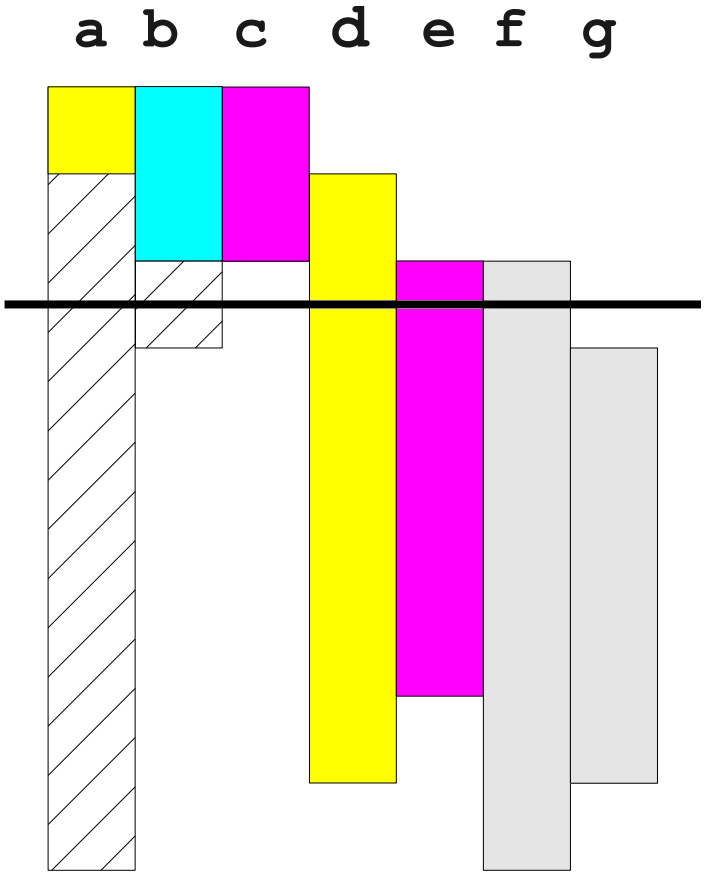
Second-Chance Bin Packing



Free Registers

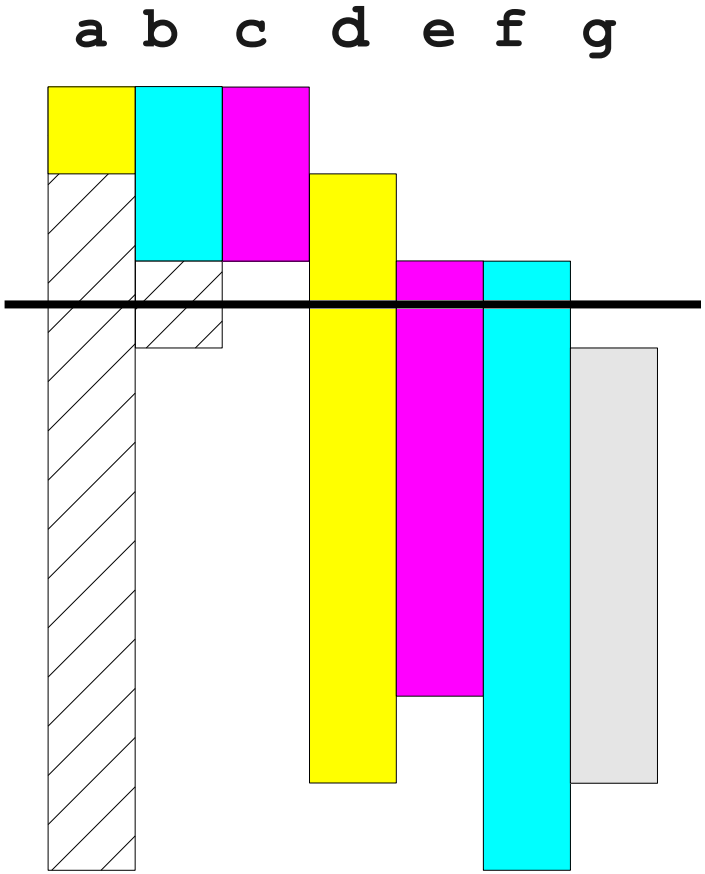
R_0	R_1	R_2

Second-Chance Bin Packing



Free Registers

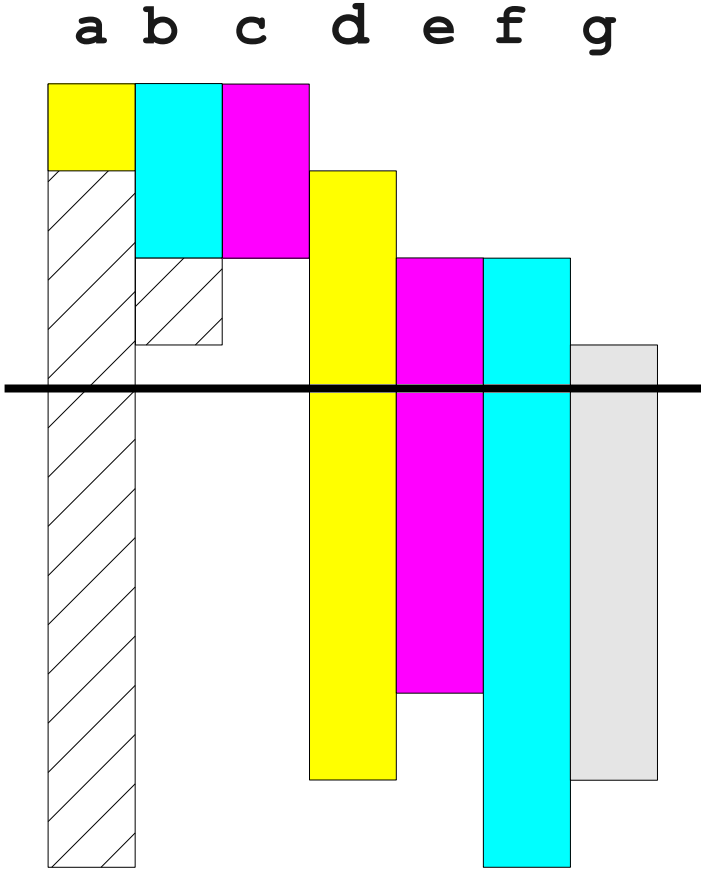
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

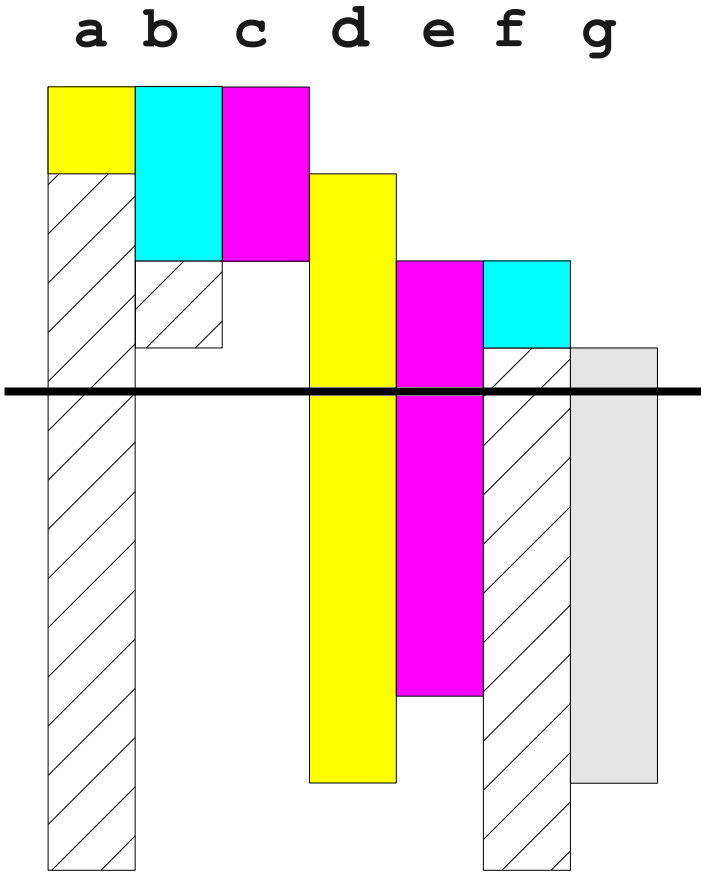
Second-Chance Bin Packing



Free Registers

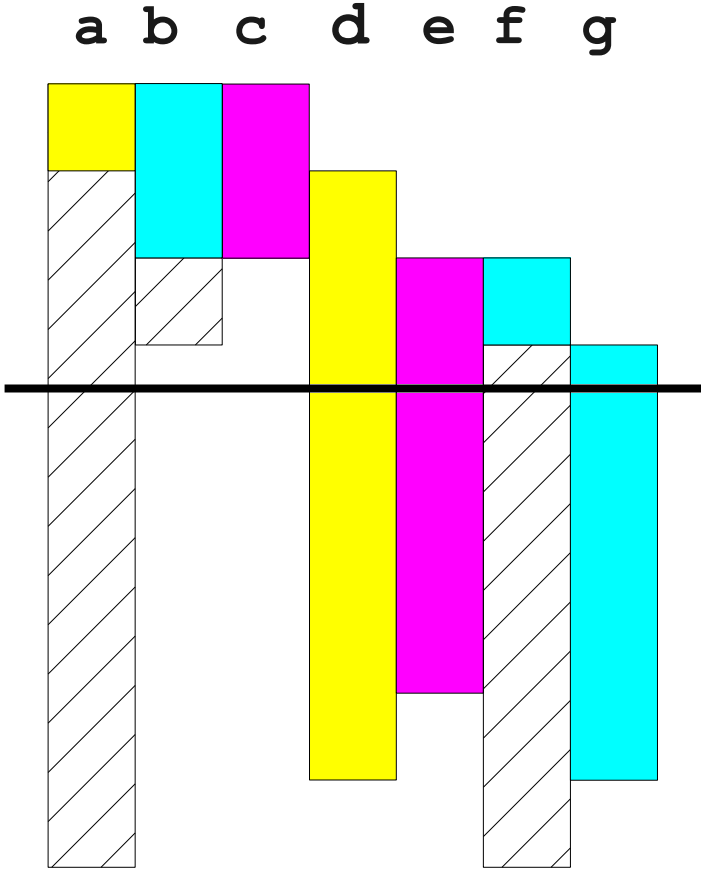
R_0	R_1	R_2

Second-Chance Bin Packing



Free Registers

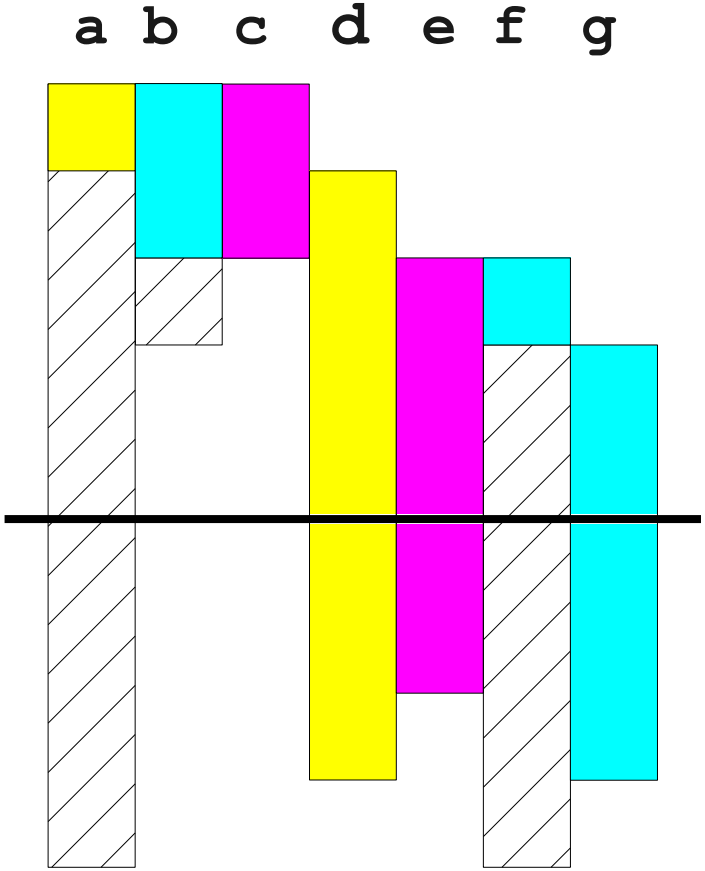
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

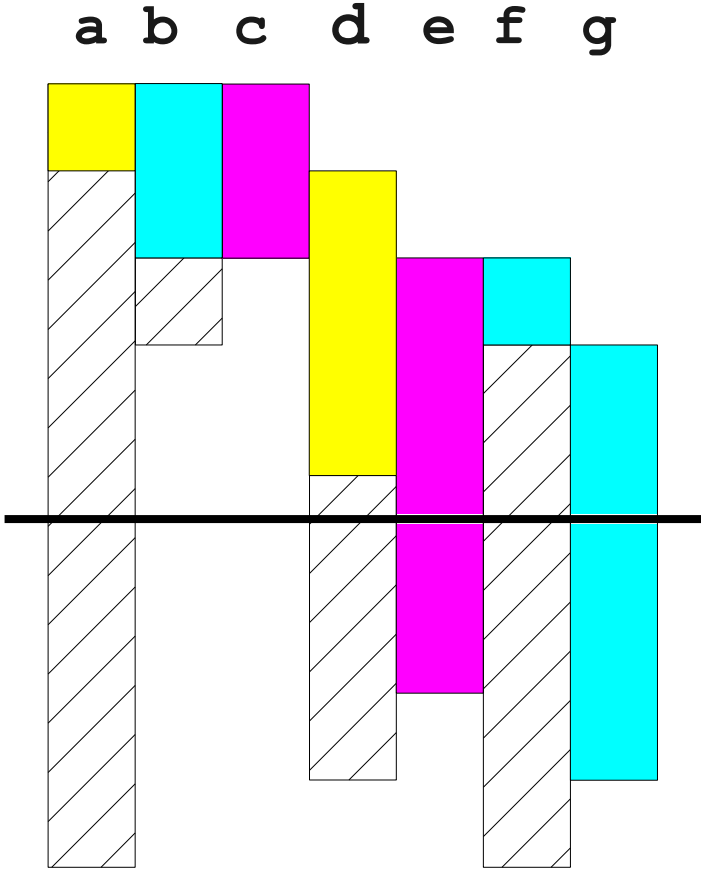
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

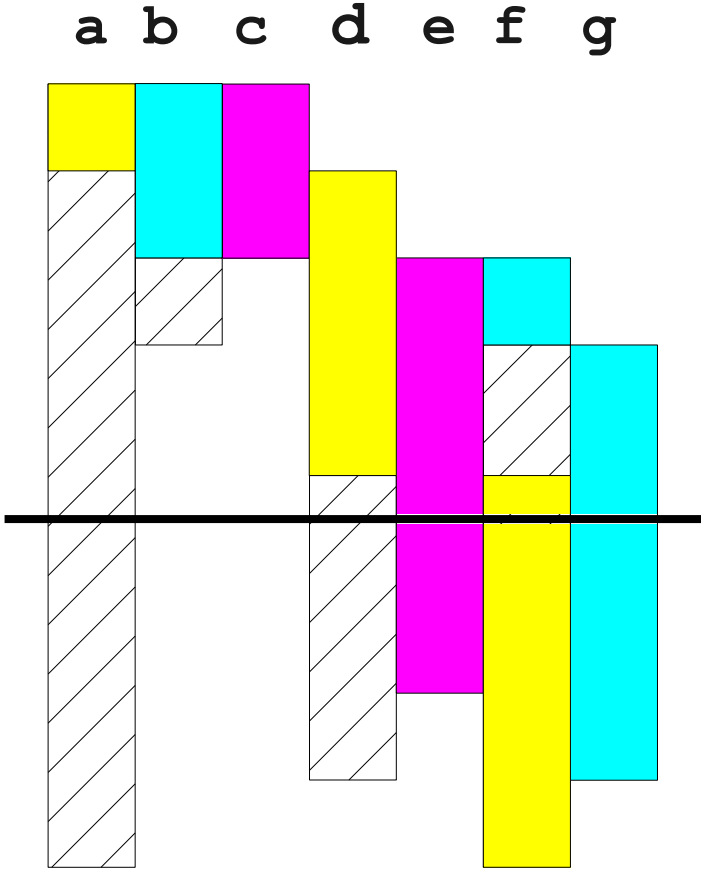
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

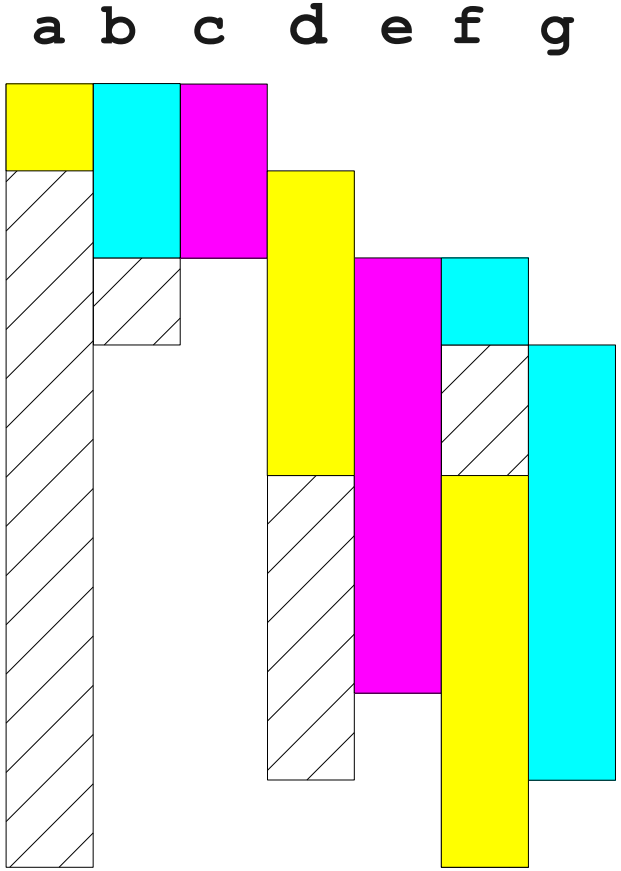
Second-Chance Bin Packing



Free Registers

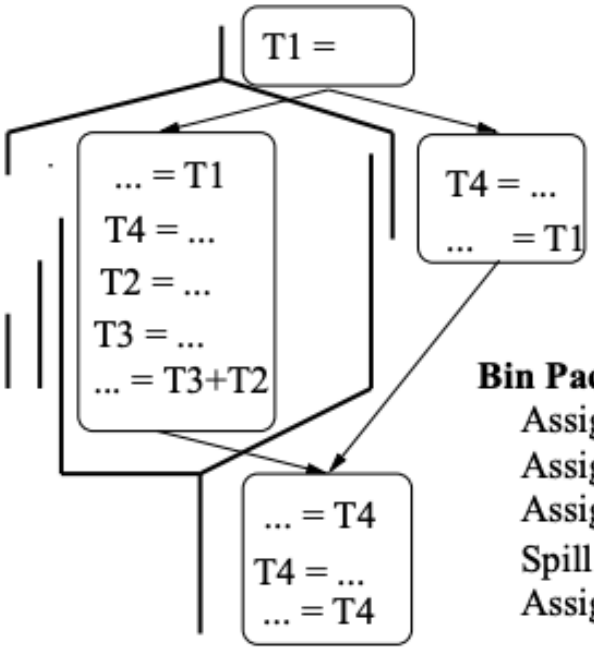
R_0	R_1	R_2

Second-Chance Bin Packing



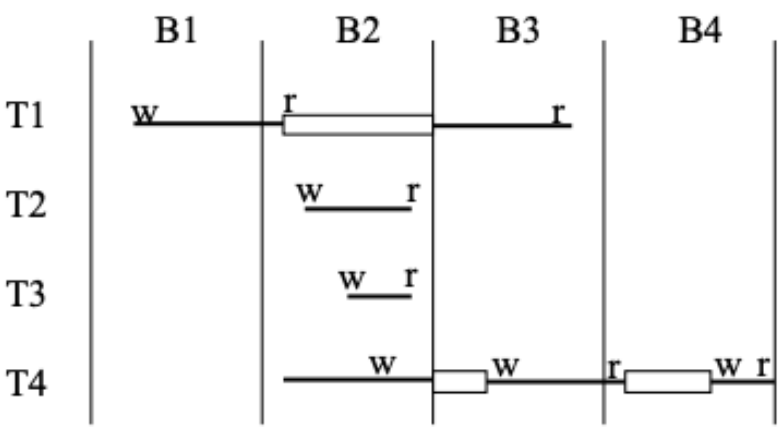
Free Registers

R_0	R_1	R_2



Bin Packing

Assign T1, R1
Assign T4, R2
Assign T2, R1
Spill T4
Assign T3, R2

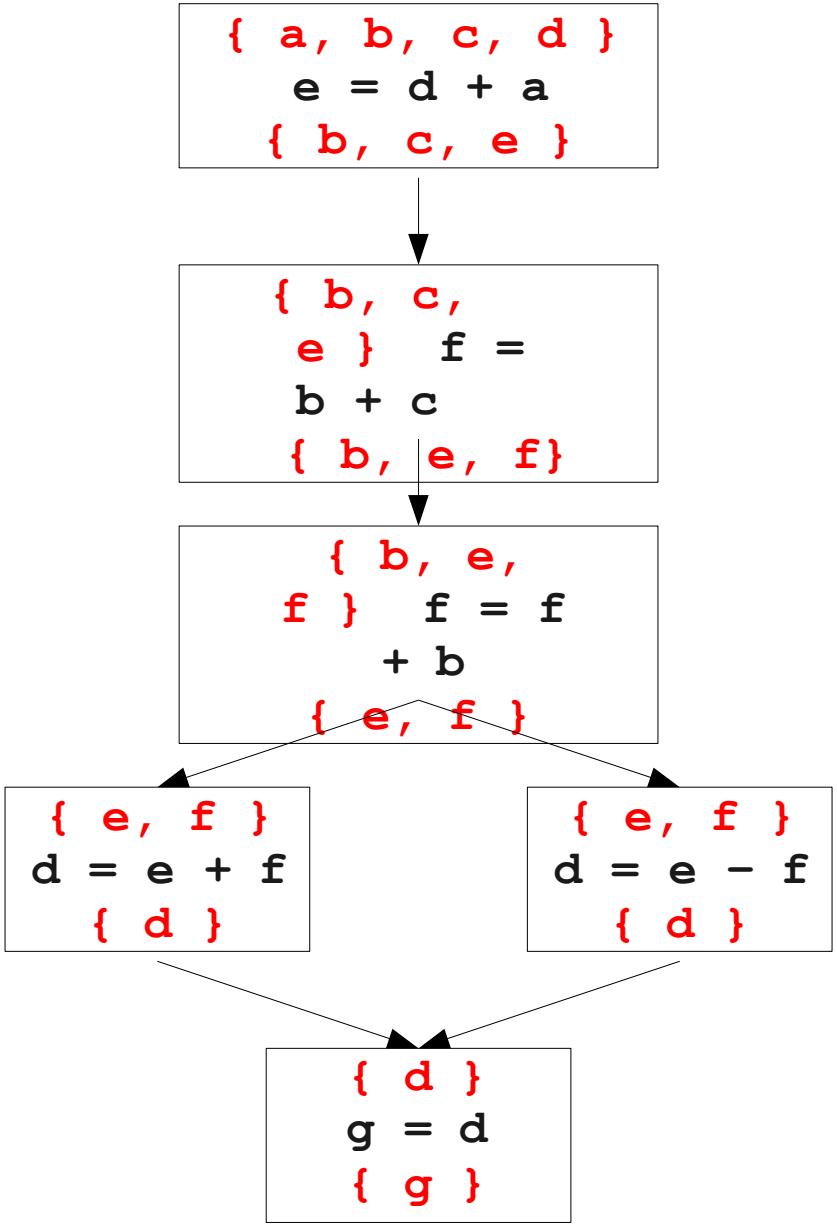


Give temporaries numerous chances to get a register (live range splitting)

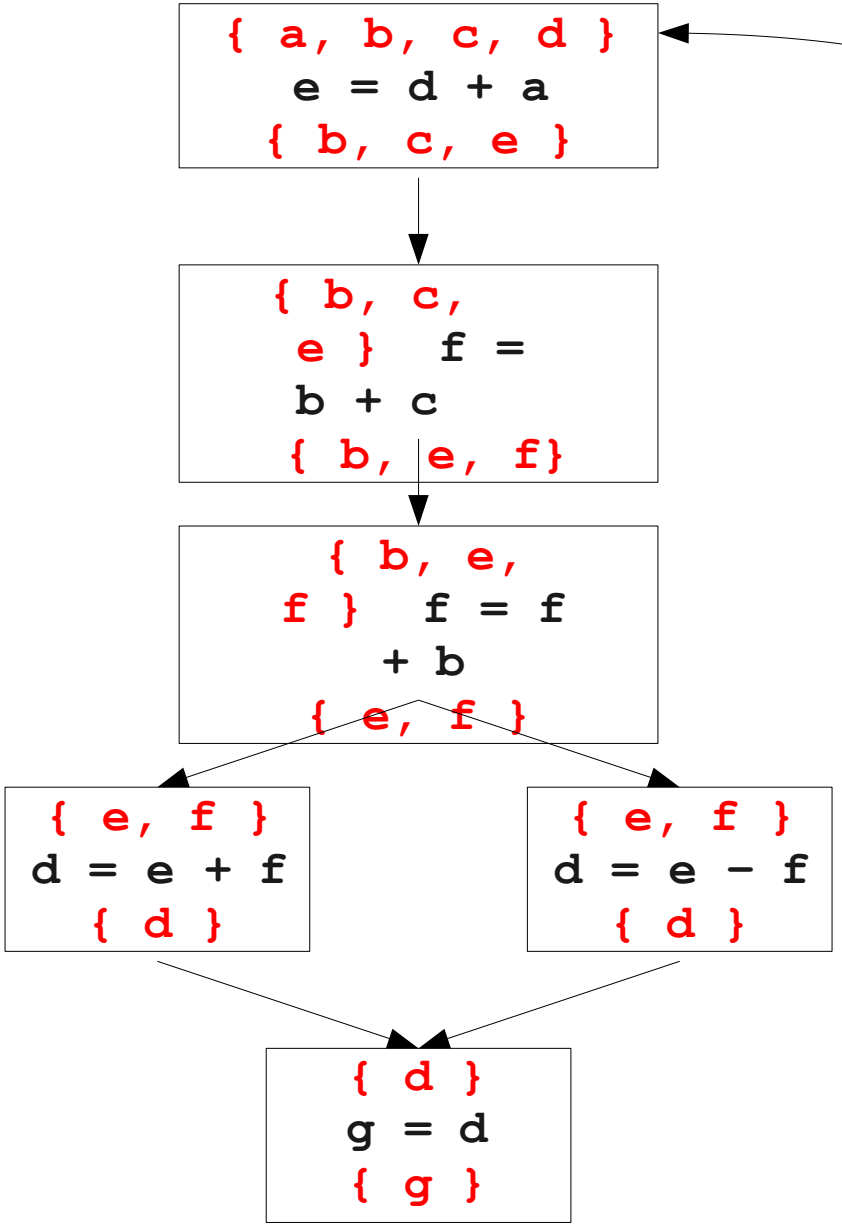
```
for each instruction in linear order
    for each temporary t
        if (t currently in register r)
            rewrite reference
        else // beginning of live range or spilled
            if (∃ r with large enough hole)
                assign t to r
            else spill lowest cost candidate
    end for
end for
for each edge in control flow graph
    resolve conflicting location assumptions
end for
```

An Entirely Different Approach

An Entirely Different Approach



An Entirely Different Approach

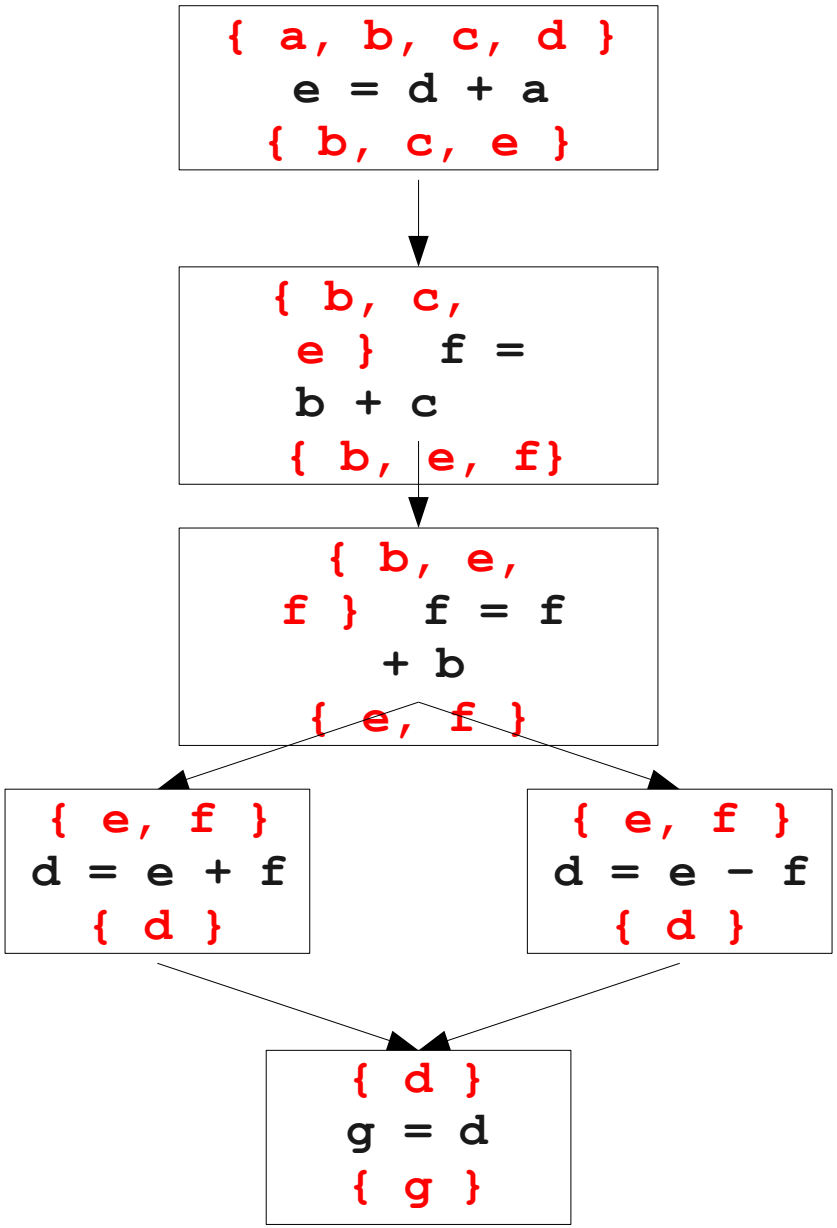


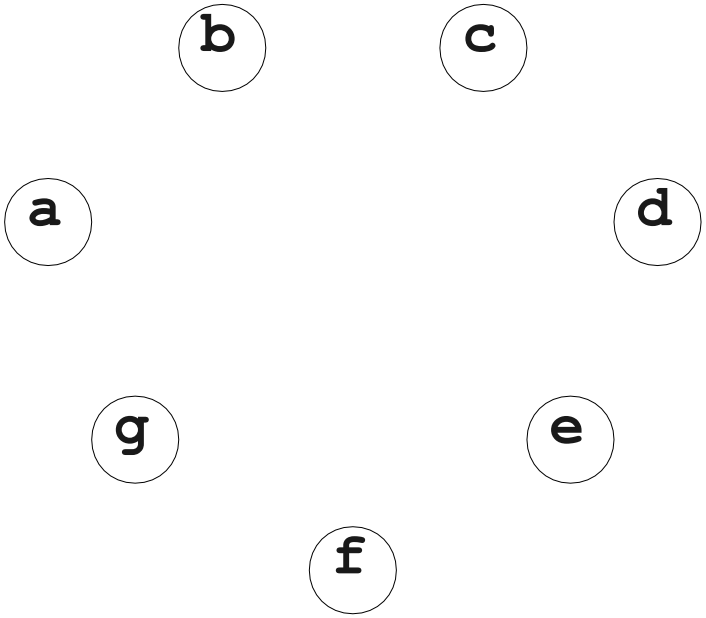
What can we infer from all these variables being live at this point?

The Register Interference Graph

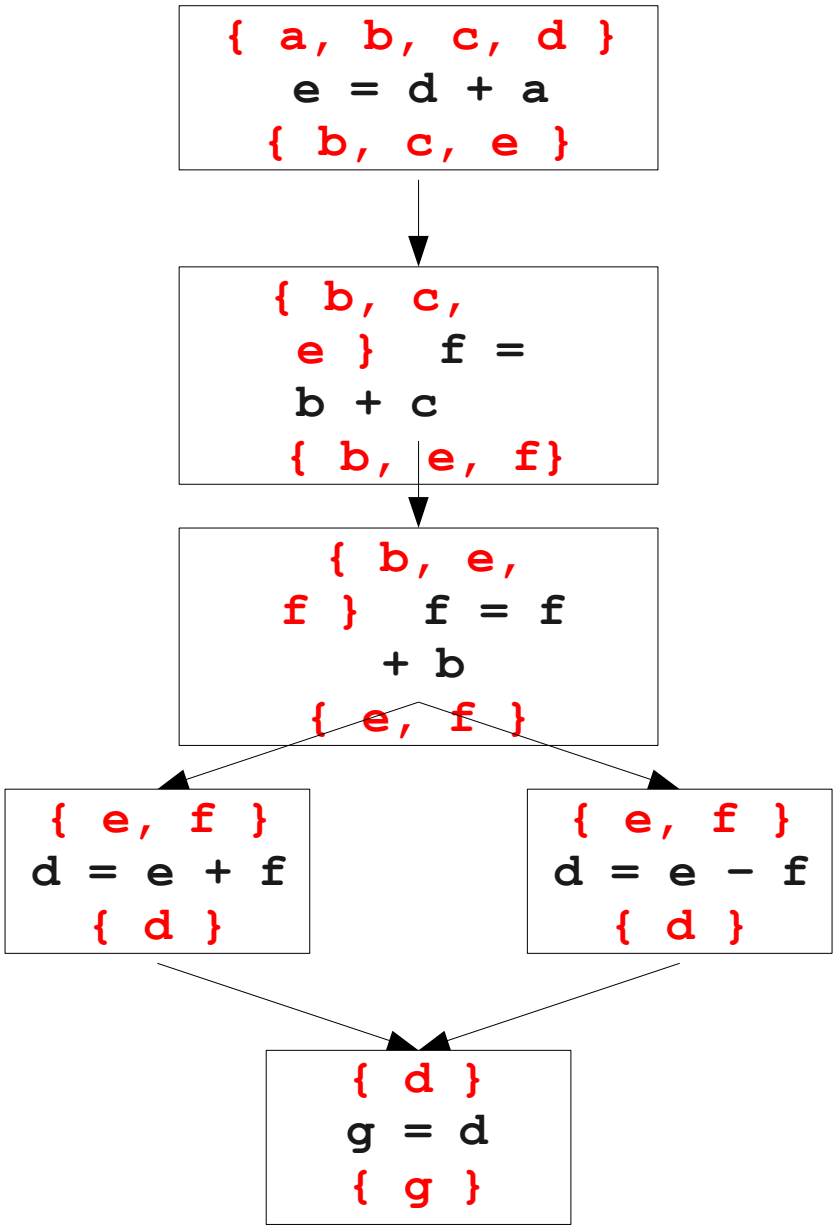
- The **register interference graph** (RIG) of a control-flow graph is an undirected graph where
  - Each node is a variable.
  - There is an edge between two variables that are live at the same program point.
- Perform register allocation by assigning each variable a different register from all of its neighbors.
- There's just one catch...

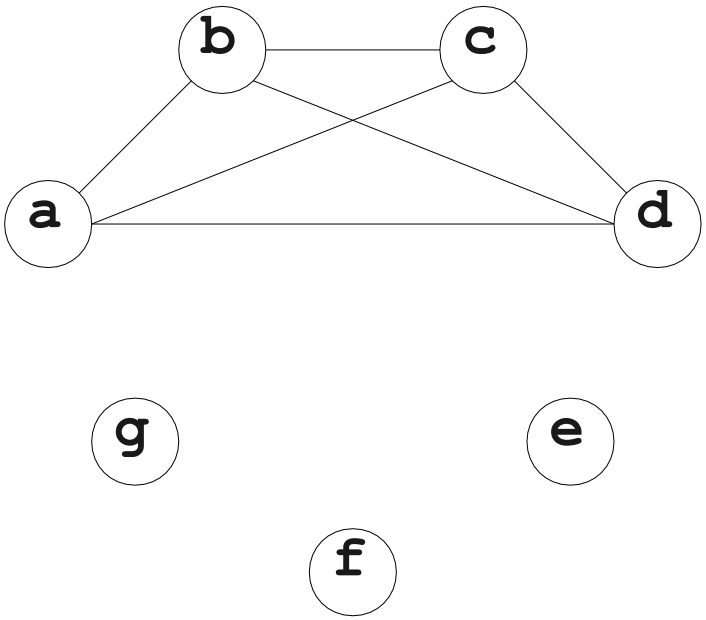
An Entirely Different Approach



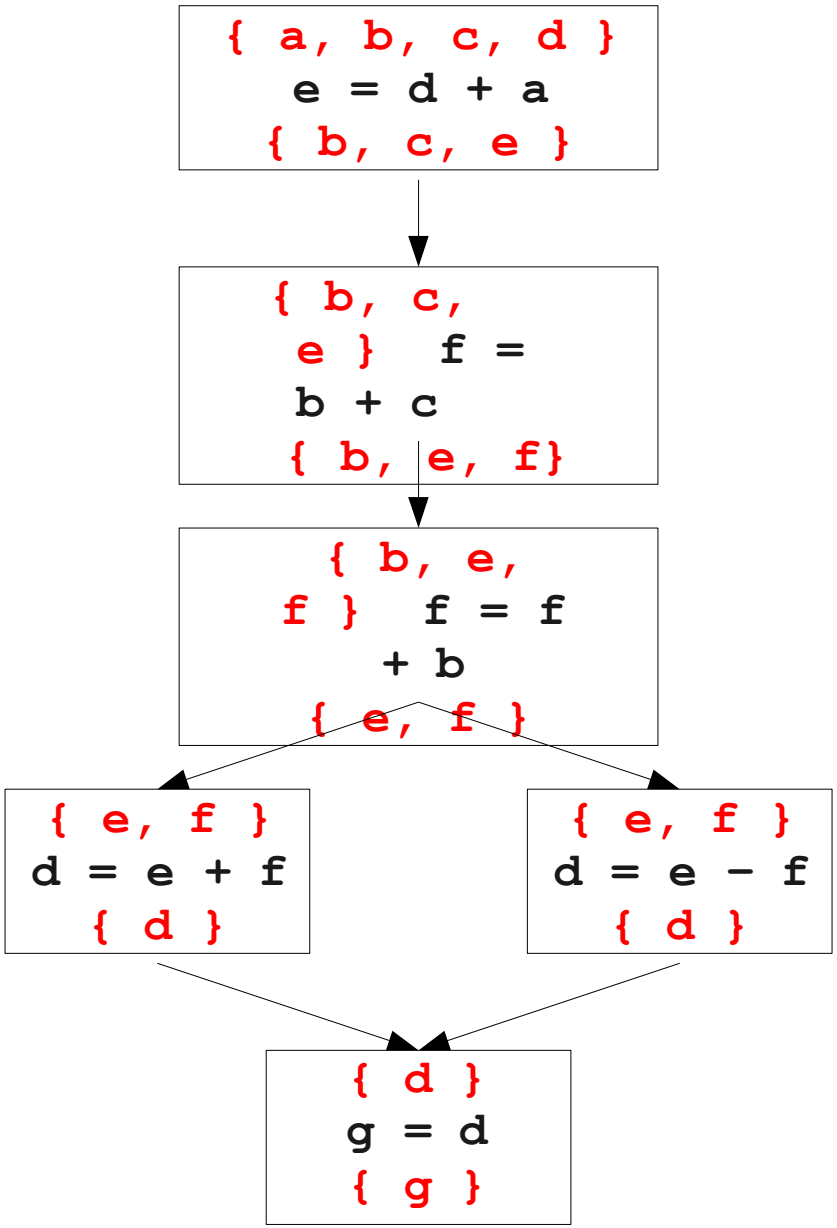


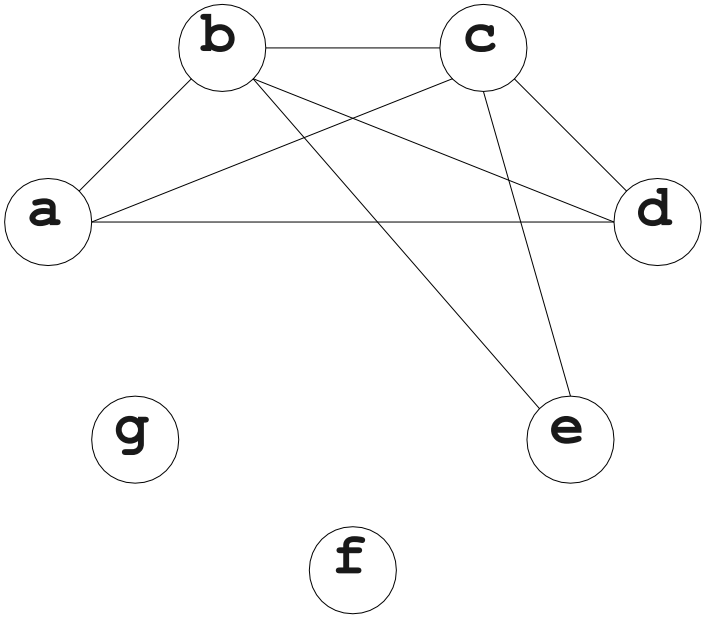
An Entirely Different Approach



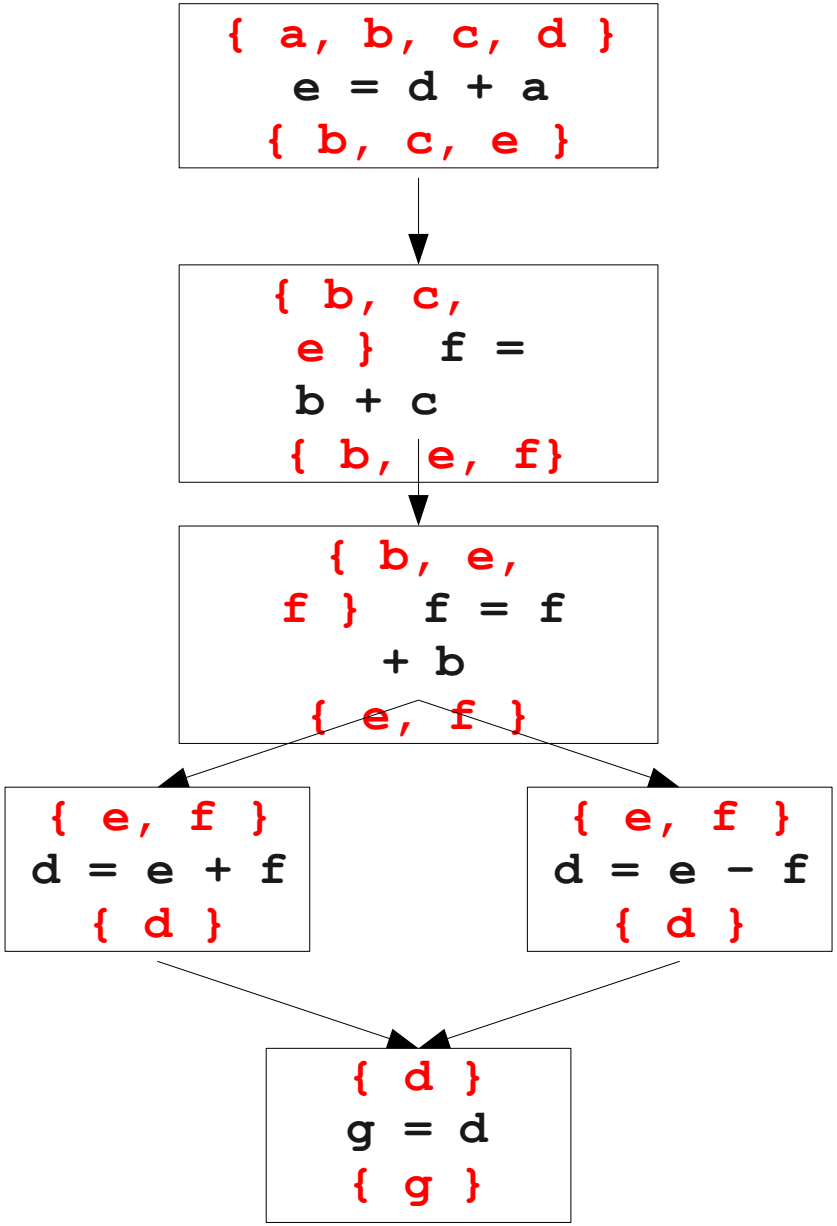


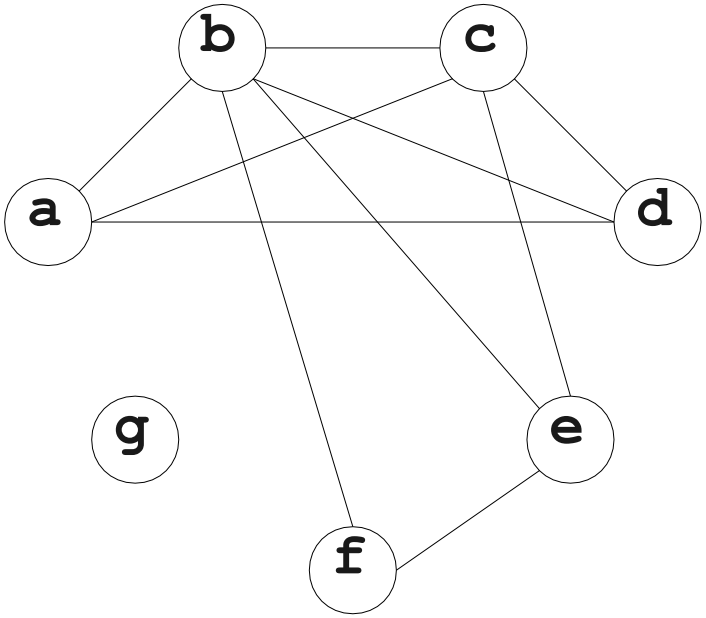
An Entirely Different Approach



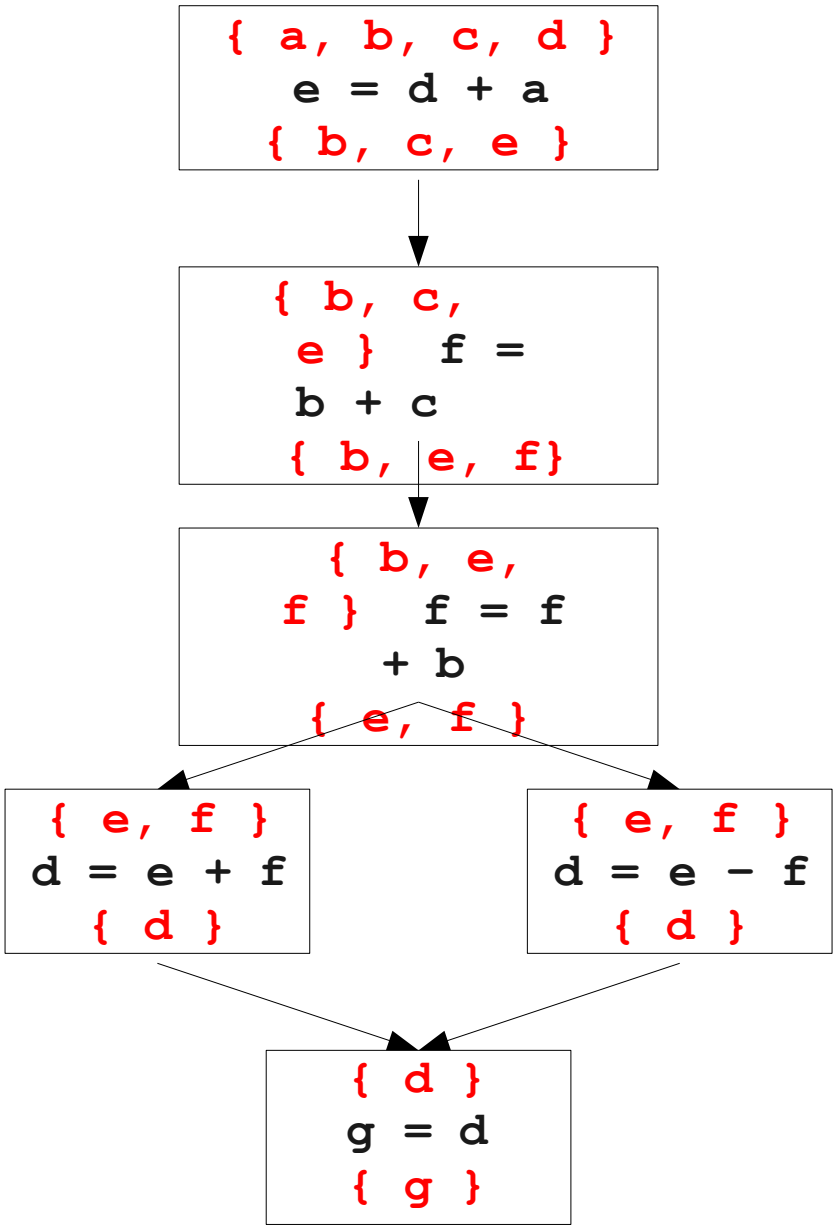


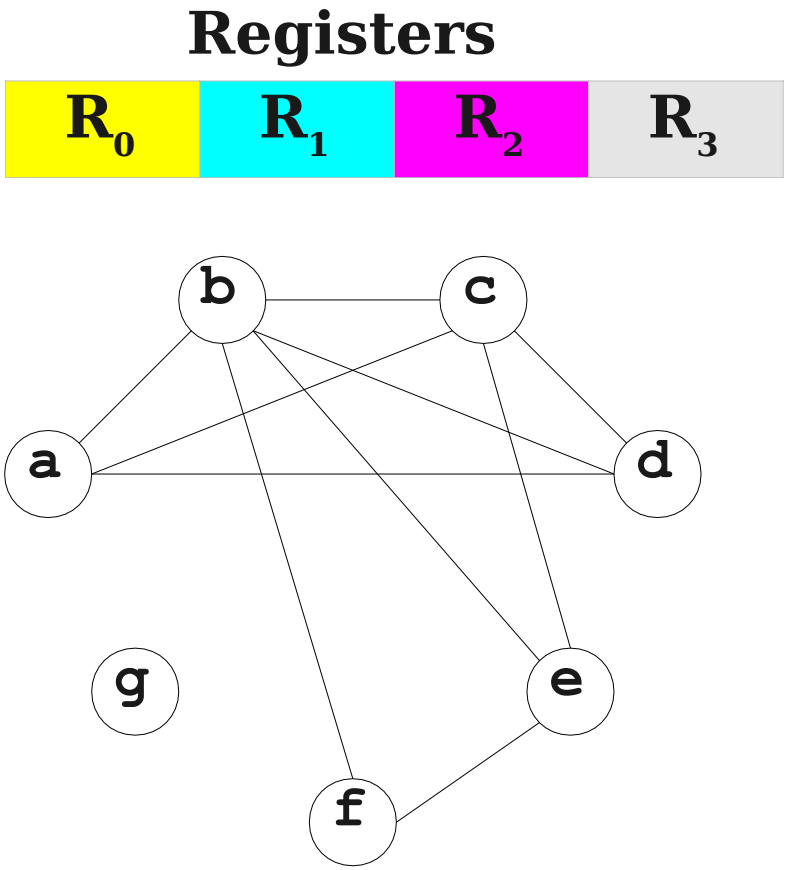
An Entirely Different Approach



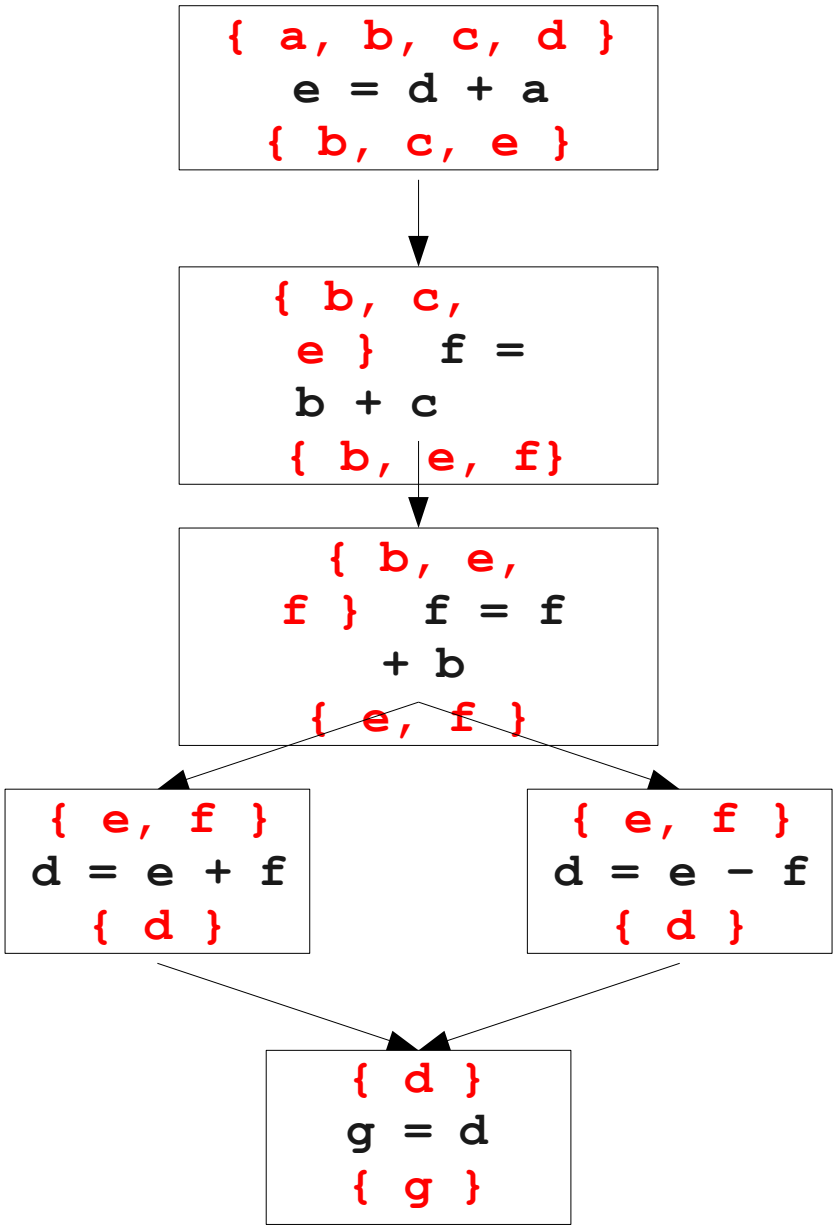


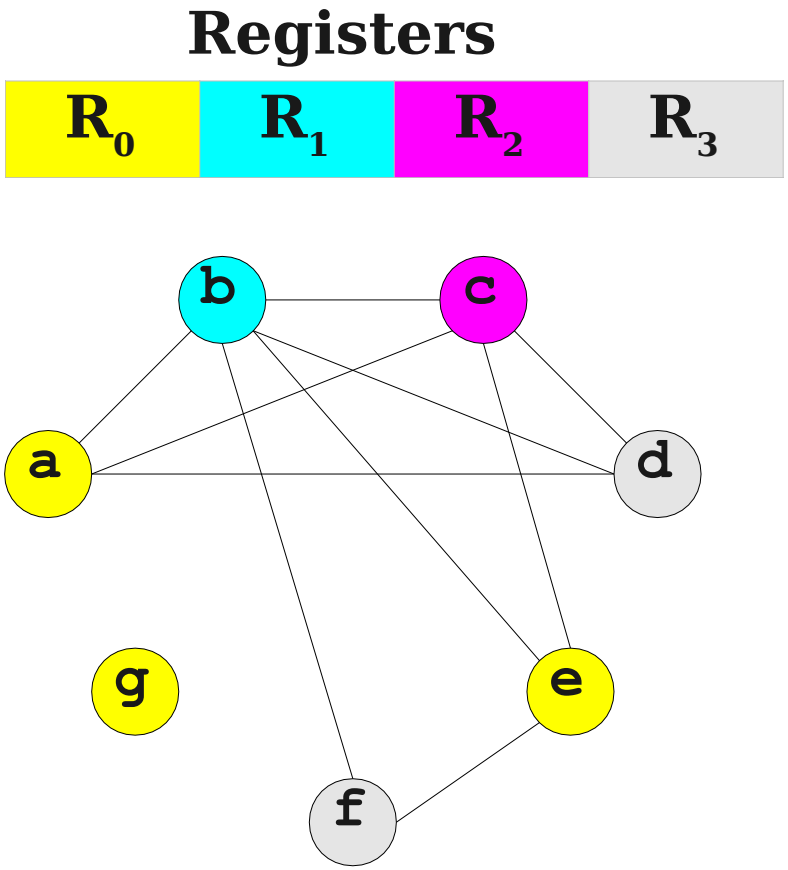
An Entirely Different Approach





An Entirely Different Approach





The One Catch

- This problem is equivalent to **graph-coloring**, which is **NP-hard** if there are at least three registers.
- No good polynomial-time algorithms (or even good approximations!) are known for this problem.
- We have to be content with a heuristic that is good enough for RIGs that arise in practice.

The One Catch to The One Catch

If you can figure out a way to assign registers to arbitrary RIGs, you've just proven **P = NP** and will get a **$1,000,000 check** from the Clay Mathematics Institute.

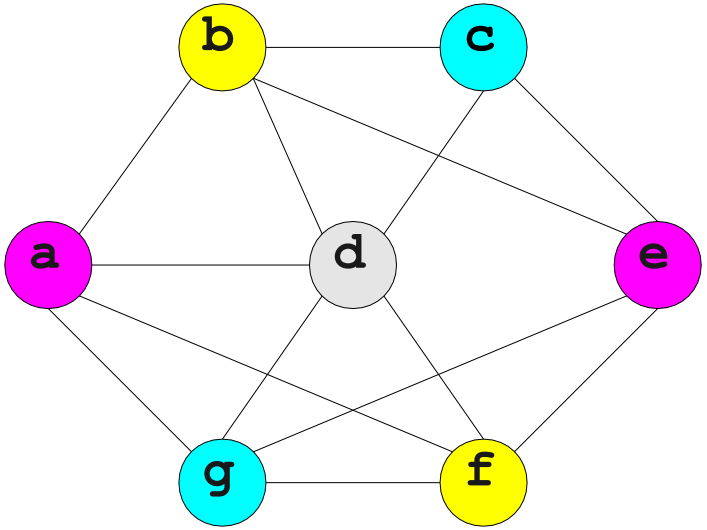
Battling **NP**-Hardness

Chaitin's Algorithm

- Intuition:
  - Suppose we are trying to k-color a graph and find a node with fewer than k edges.
  - If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in.
  - Reason: With fewer than k neighbors, some color must be left over.

- Algorithm:
  - Find a node with fewer than k outgoing edges.
  - Remove it from the graph.
  - Recursively color the rest of the graph.
  - Add the node back in.
  - Assign it a valid color.

Chaitin's Algorithm



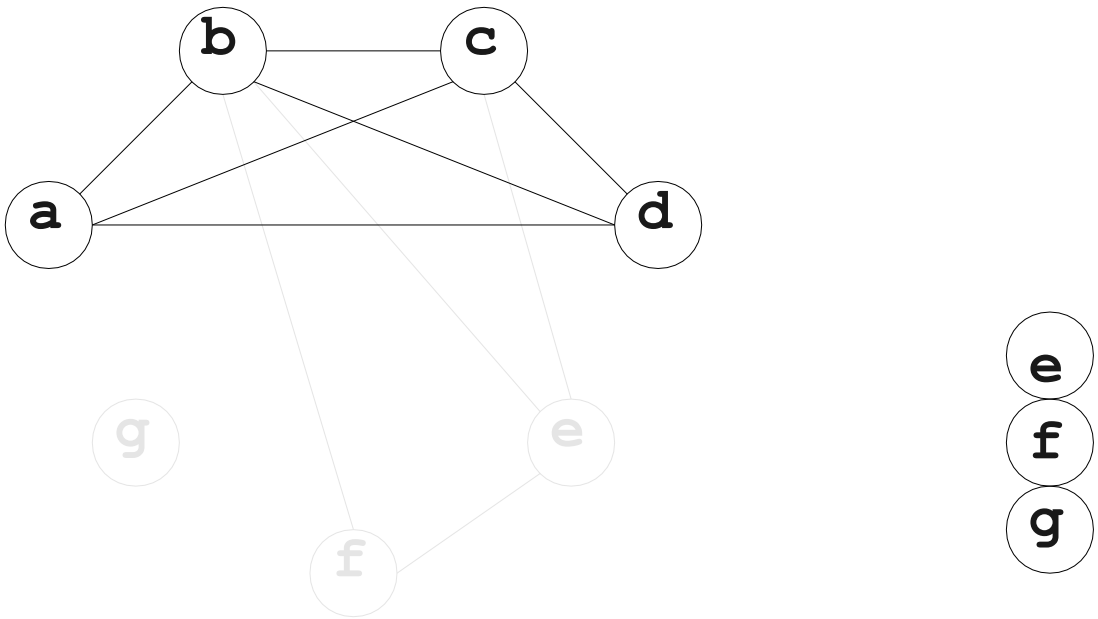
Registers

R_0	R_1	R_2	R_3

One Problem

- What if we can't find a node with fewer than k neighbors?
- Choose and remove an arbitrary node, marking it “troublesome.”
  - Use heuristics to choose which one.
- When adding node back in, it may be possible to find a valid color.
- Otherwise, we have to spill that node.

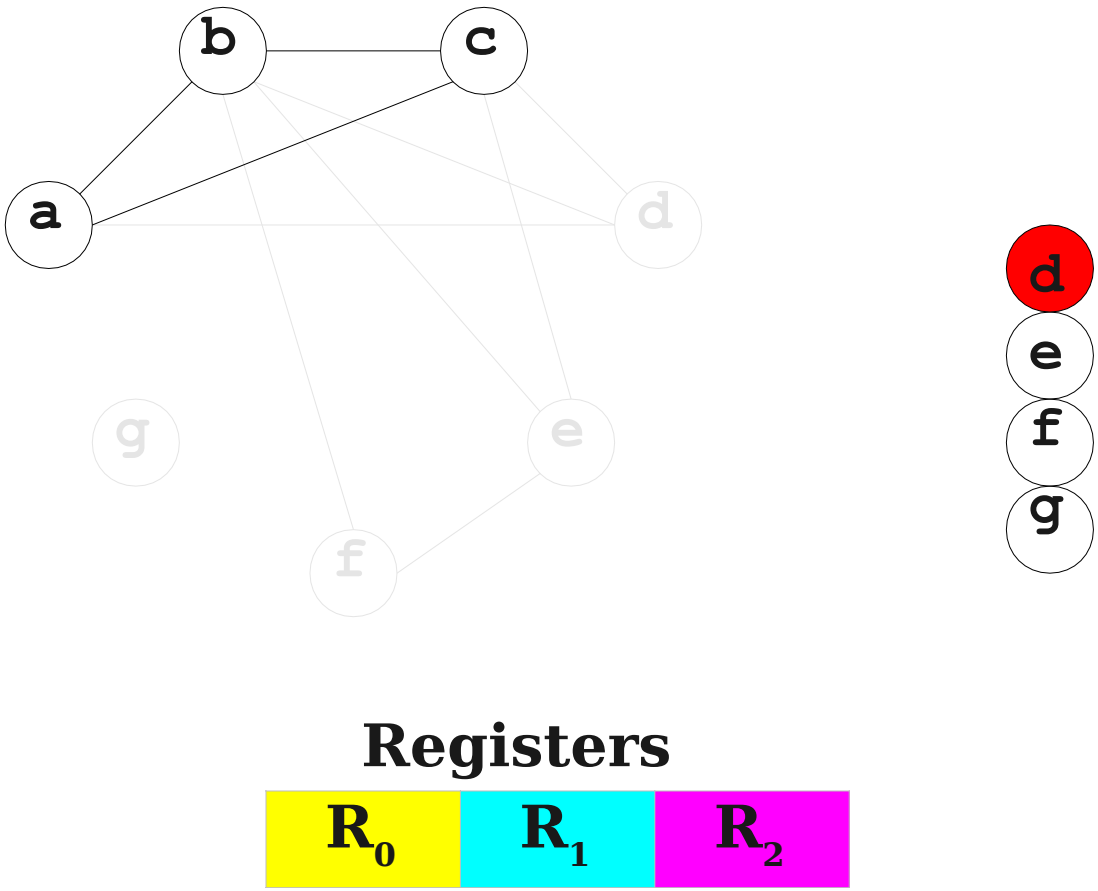
Chaitin's Algorithm Reloaded



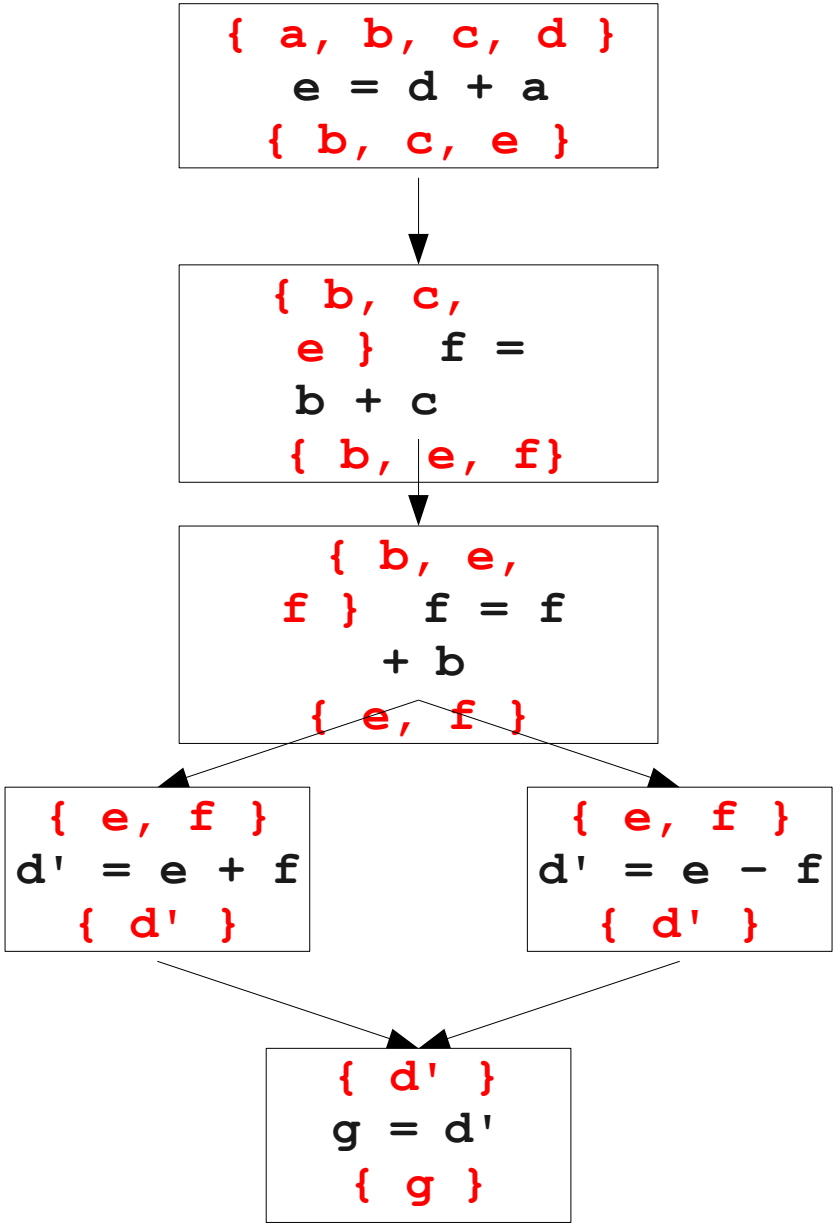
Registers

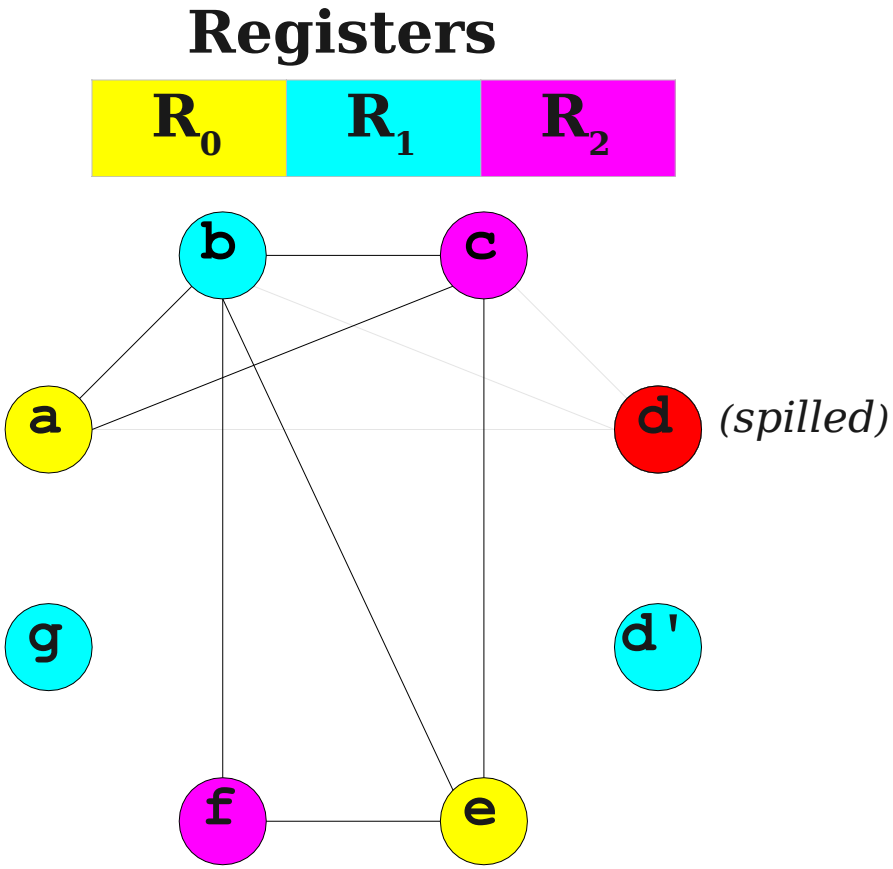
R_0	R_1	R_2

Chaitin's Algorithm Reloaded



A Smarter Algorithm





Chaitin's Algorithm

- Advantages:
  - For many control-flow graphs, finds an excellent assignment of variables to registers.
  - When distinguishing variables by use, produces a precise RIG.
  - Often used in production compilers like GCC.
- Disadvantages:
  - Core approach based on the NP-hard graph coloring problem.
  - Heuristic may produce pathologically worst-case assignments.

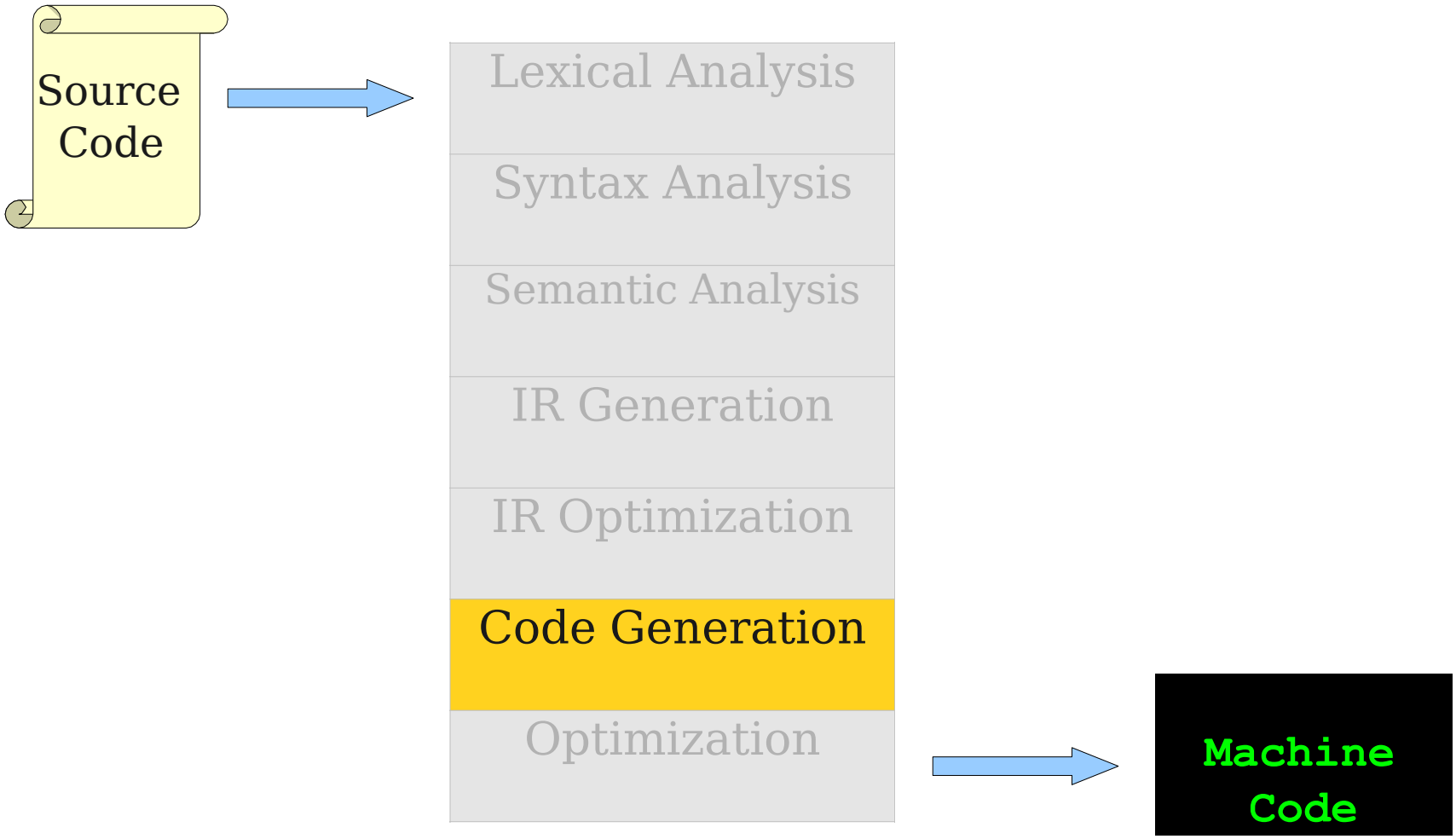
Improvements to the Algorithm

- Choose what to spill intelligently.
  - Use heuristics (least-commonly used, greatest improvement, etc.) to determine what to spill.
- Handle spilling intelligently.
  - When spilling a variable, recompute the RIG based on the spill and use a new coloring to find a register.

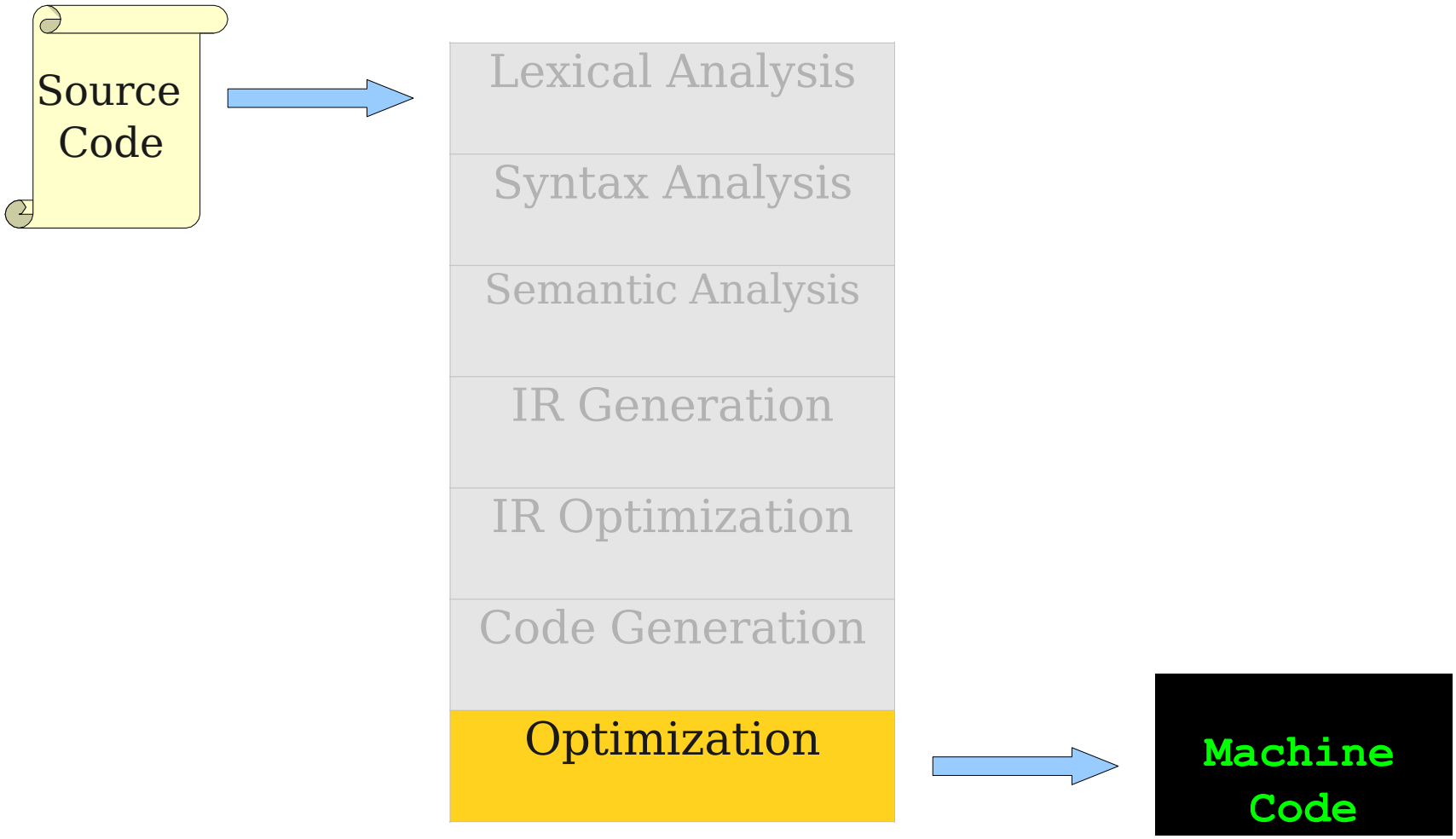
Summary of Register Allocation

- Critical step in all optimizing compilers.
- The **linear scan** algorithm uses **live intervals** to greedily assign variables to registers.
  - Often used in JIT compilers due to efficiency.
- **Chaitin's algorithm** uses the **register interference graph** (based on **live ranges**) and **graph coloring** to assign registers.
  - The basis for the technique used in GCC.

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Final Code Optimization

- **Goal**: Optimize generated code by exploiting machine-dependent properties not visible at the IR level.
- Critical step in most compilers, but often very messy:
  - Techniques developed for one machine may be completely useless on another.
  - Techniques developed for one language may be completely useless with another.

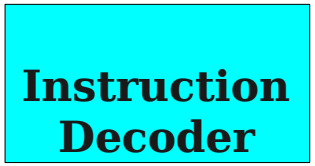
Optimizations for Pipelining

Processor Pipelines

Processor Pipelines

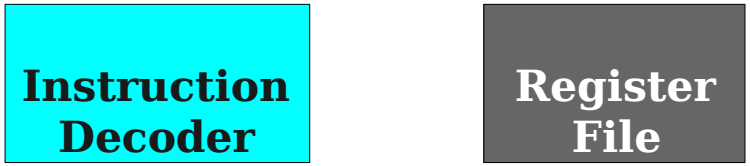
```
add   $t2,  $t0, $t1     # $t2 = $t0 + $t1
add   $t5,  $t3, $t4     # $t5 = $t3 + $t4
add   $t8,  $t6, $t7     # $t8 = $t6 + $t7
```

Processor Pipelines



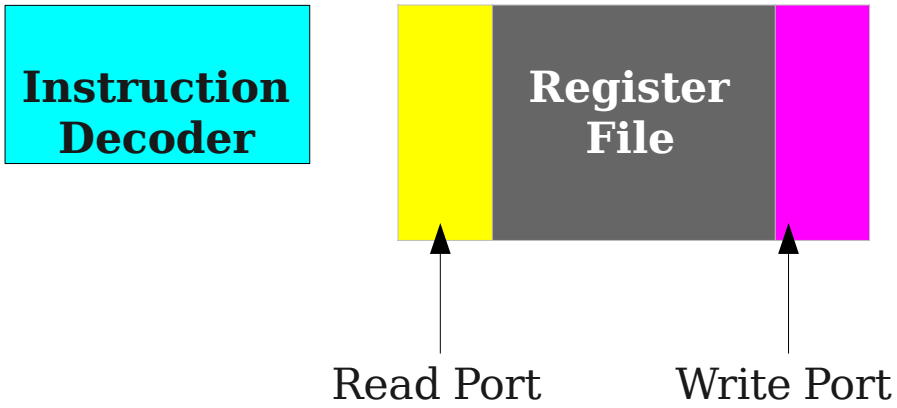

```
add   $t2,   $t0, $t1     # $t2  =  $t0  +  $t1
add   $t5,   $t3, $t4     # $t5  =  $t3  +  $t4
add   $t8,   $t6, $t7     # $t8  =  $t6  +  $t7
```

Processor Pipelines



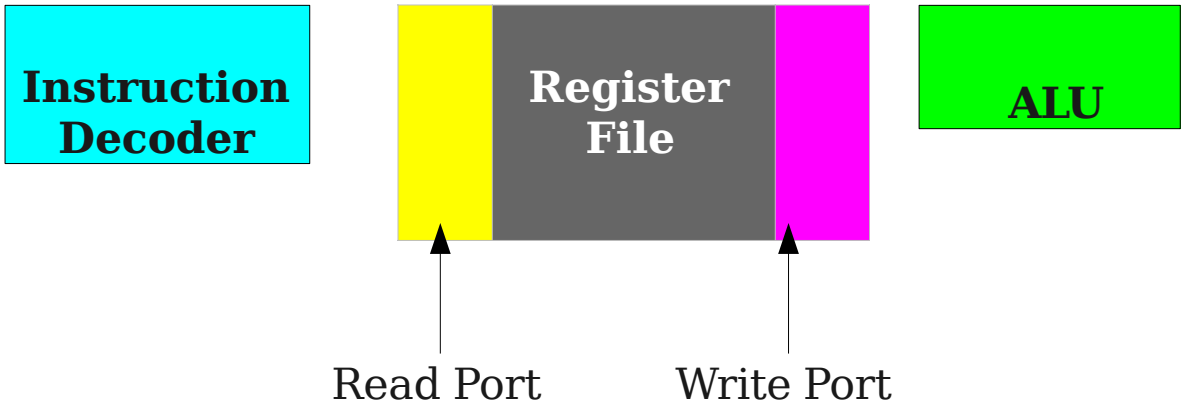

```
add   $t2,   $t0, $t1     # $t2  =  $t0  +  $t1
add   $t5,   $t3, $t4     # $t5  =  $t3  +  $t4
add   $t8,   $t6, $t7     # $t8  =  $t6  +  $t7
```

Processor Pipelines



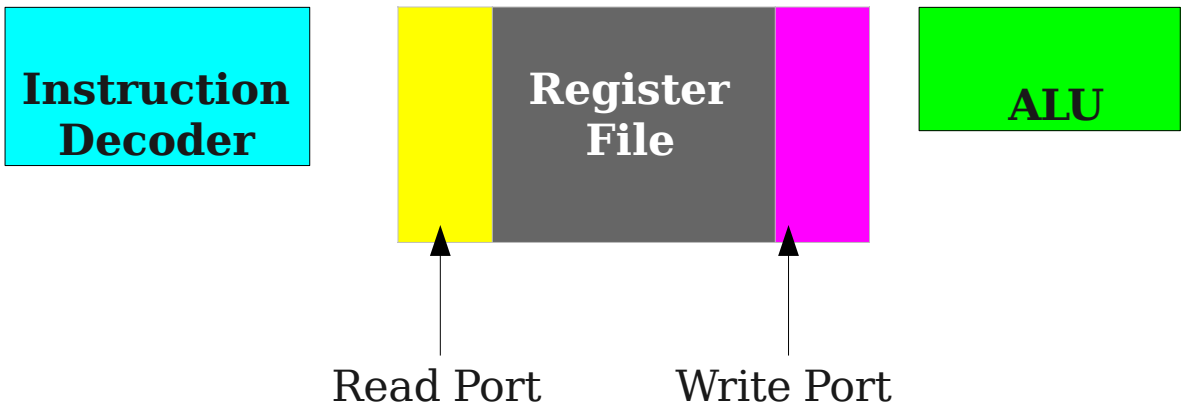

```
add   $t2,   $t0,  $t1     # $t2  =  $t0  +  $t1
add   $t5,   $t3,  $t4     # $t5  =  $t3  +  $t4
add   $t8,   $t6,  $t7     # $t8  =  $t6  +  $t7
```

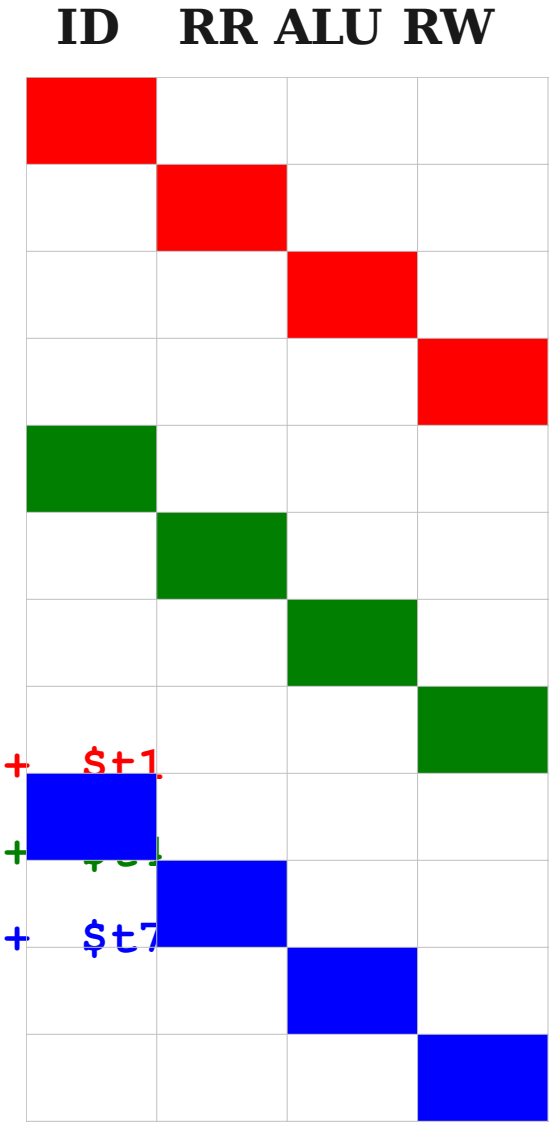
Processor Pipelines




```
add   $t2,   $t0,  $t1     # $t2  =  $t0  +  $t1
add   $t5,   $t3,  $t4     # $t5  =  $t3  +  $t4
add   $t8,   $t6,  $t7     # $t8  =  $t6  +  $t7
```

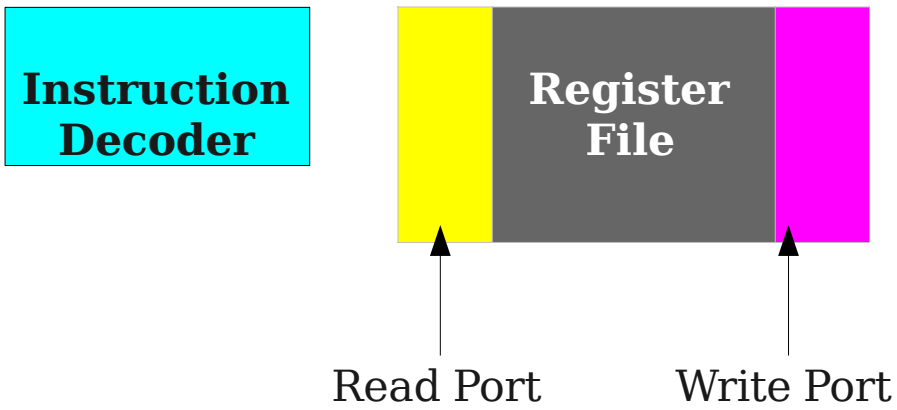
Processor Pipelines





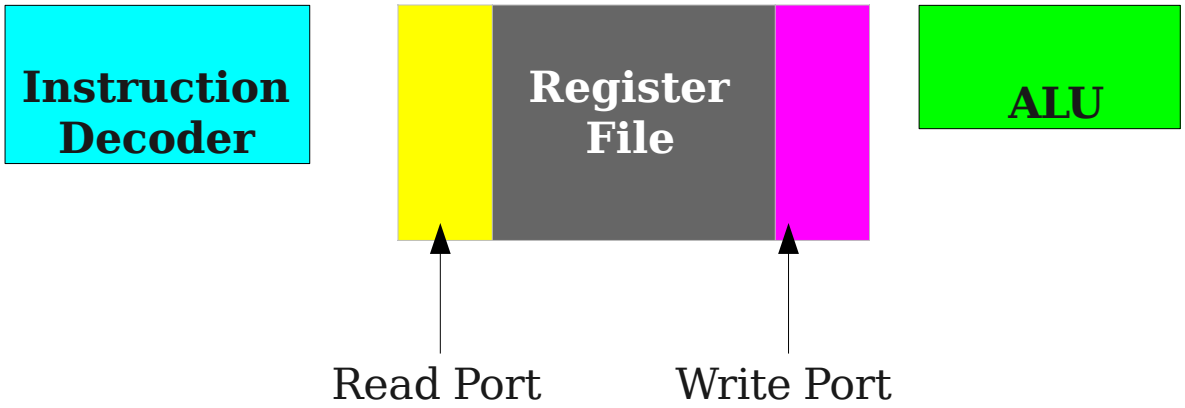

```
add   $t2,   $t0,   $t1      # $t2  =  $t0  +  $t1
add   $t5,   $t3,   $t4      # $t5  =  $t3  +  $t4
add   $t8,   $t6,   $t7      # $t8  =  $t6  +  $t7
```

Processor Pipelines



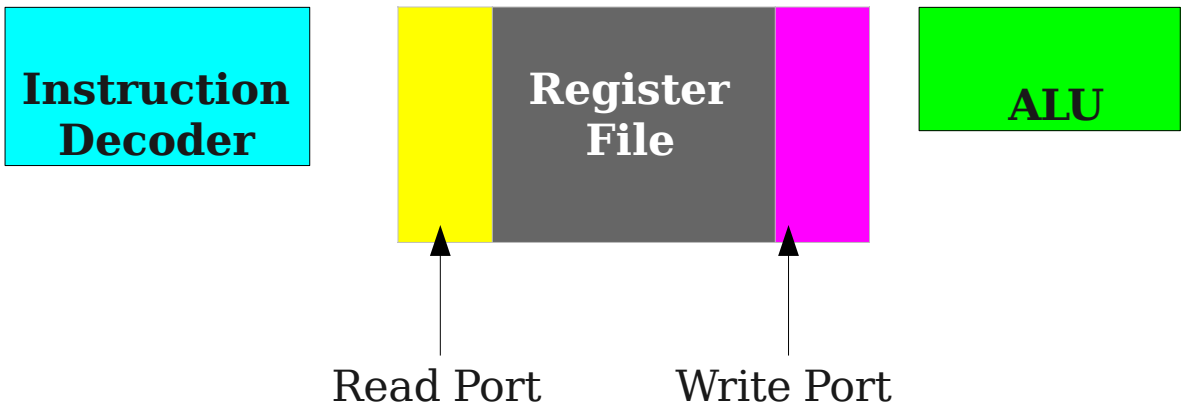

```
add   $t2,   $t0,   $t1      # $t2  =  $t0  +  $t1
add   $t5,   $t3,   $t4      # $t5  =  $t3  +  $t4
add   $t8,   $t6,   $t7      # $t8  =  $t6  +  $t7
```

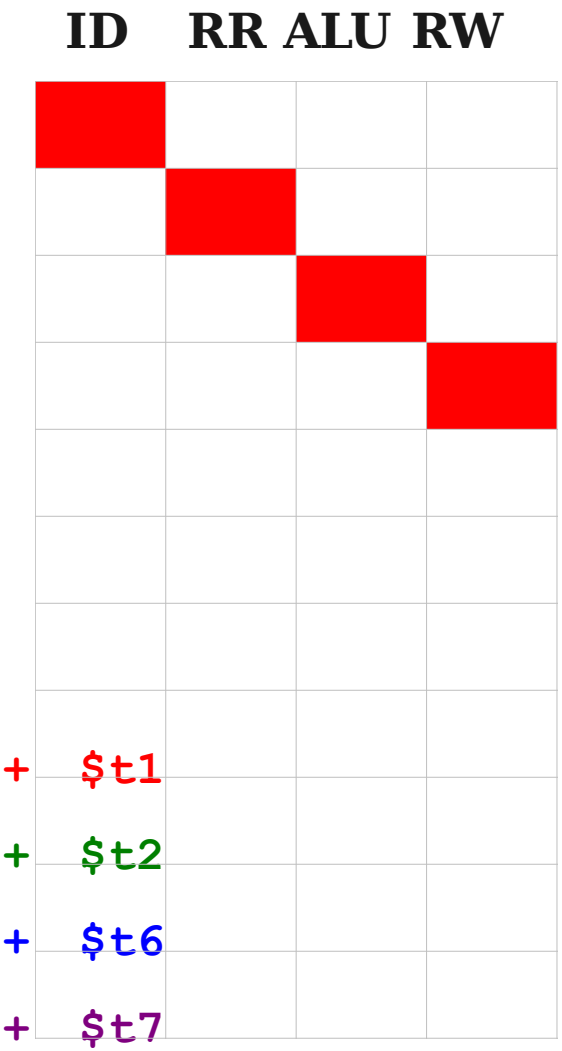
Pipeline Hazards




```
add   $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add   $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add   $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add   $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

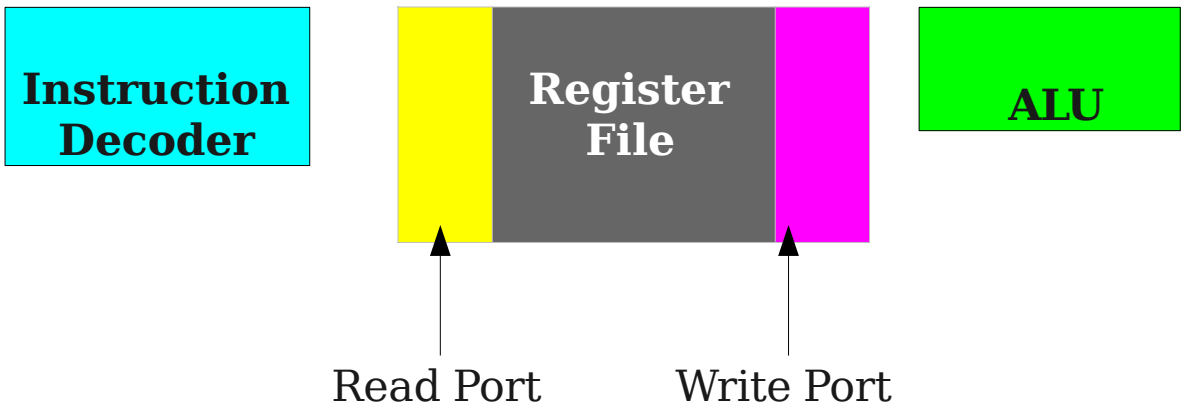
Pipeline Hazards






```
add  $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add  $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add  $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add  $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

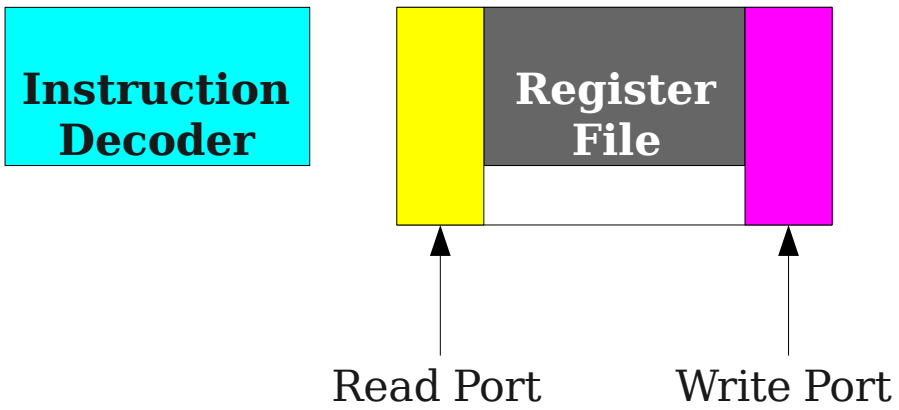
Pipeline Hazards

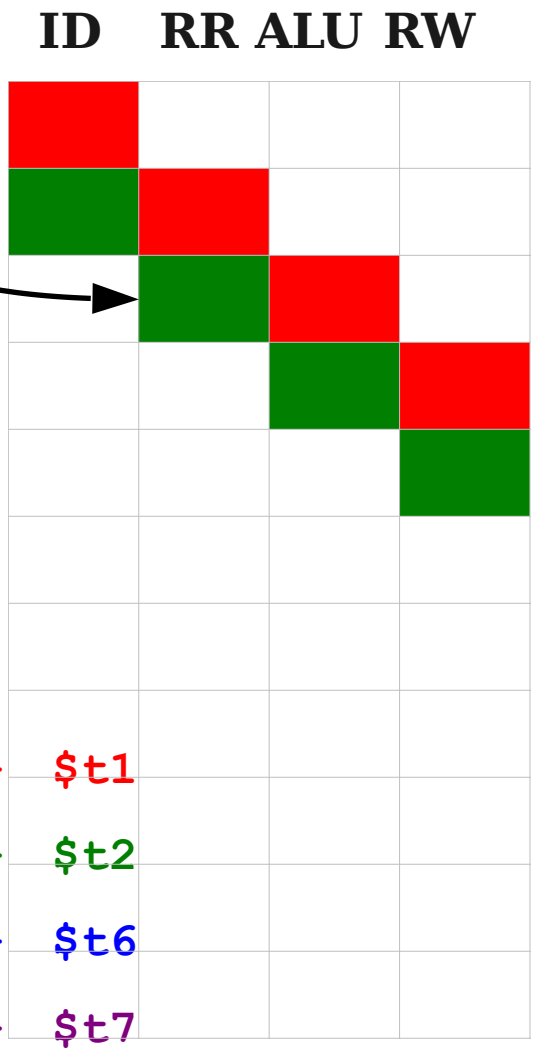



```
add   $t2,   $t0,   $t1      # $t2  =  $t0  +  $t1
add   $t4,   $t3,   $t2      # $t5  =  $t3  +  $t2
add   $t7,   $t5,   $t6      # $t7  =  $t5  +  $t6
add   $t0,   $t0,   $t7      # $t0  =  $t0  +  $t7
```

Pipeline Hazards

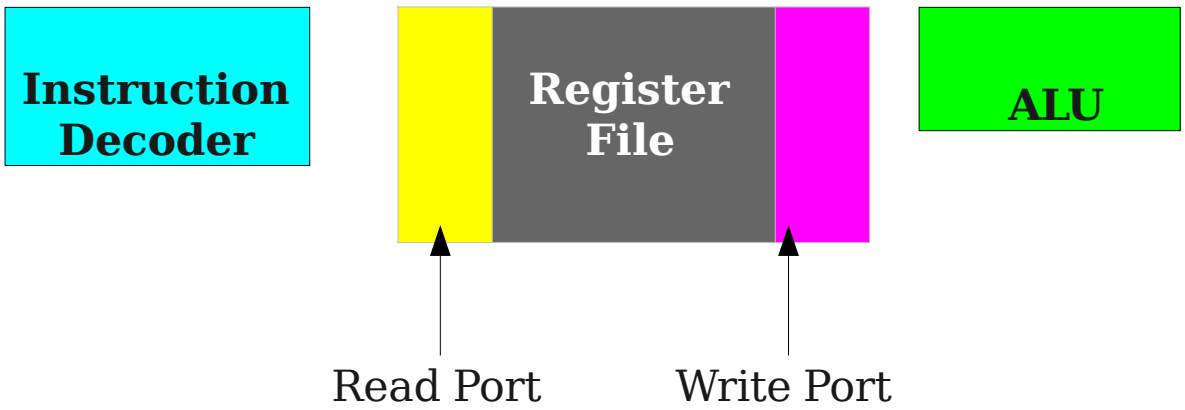
This value isn't ready yet!





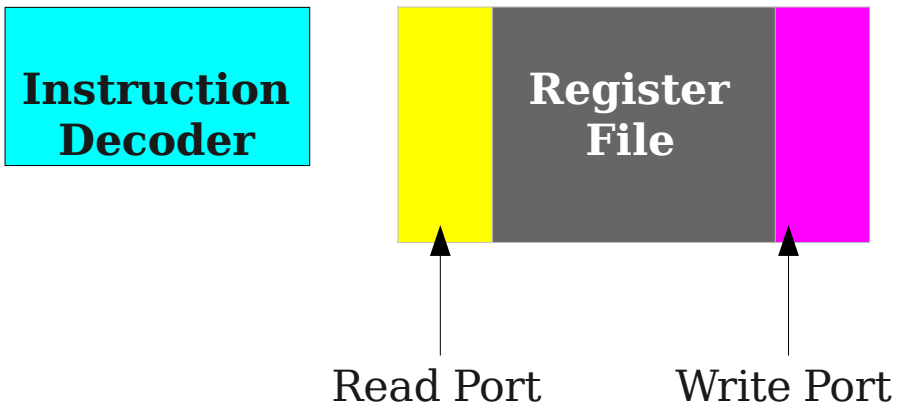

```
add  $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add  $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add  $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add  $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

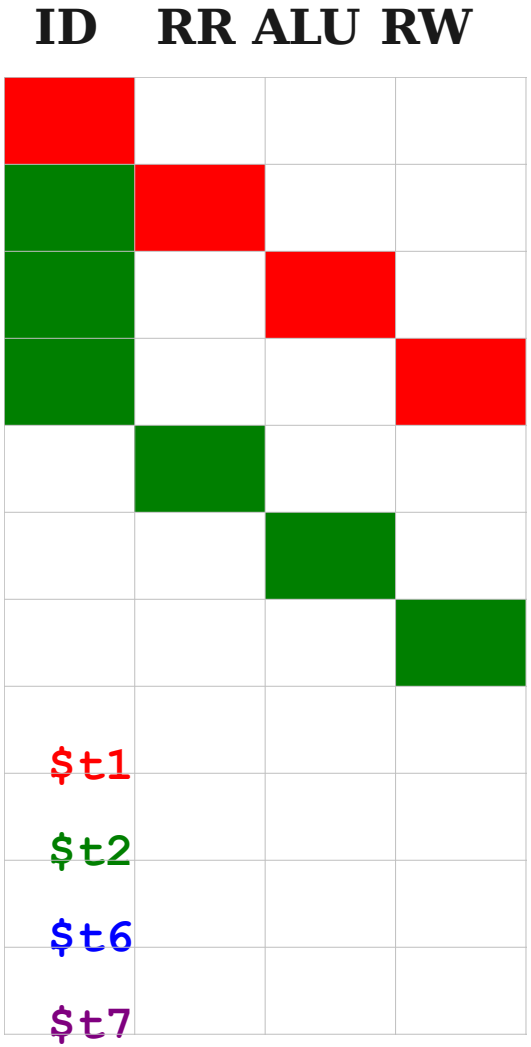
Pipeline Hazards




```
add   $t2,   $t0,   $t1      # $t2  =  $t0  +  $t1
add   $t4,   $t3,   $t2      # $t5  =  $t3  +  $t2
add   $t7,   $t5,   $t6      # $t7  =  $t5  +  $t6
add   $t0,   $t0,   $t7      # $t0  =  $t0  +  $t7
```

Pipeline Hazards

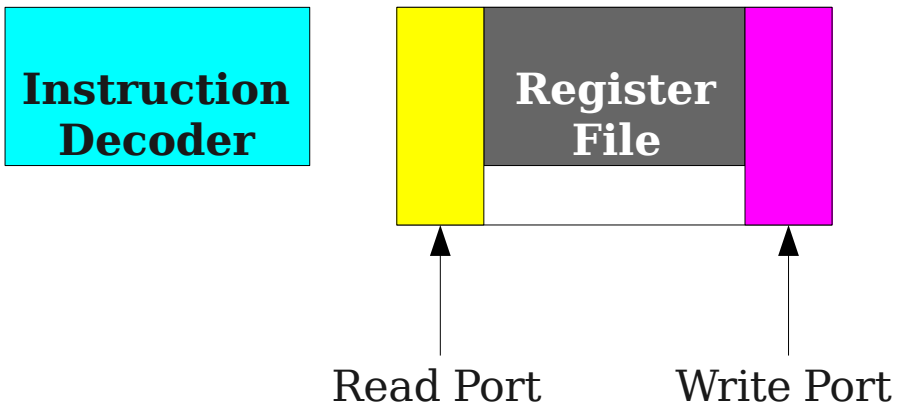


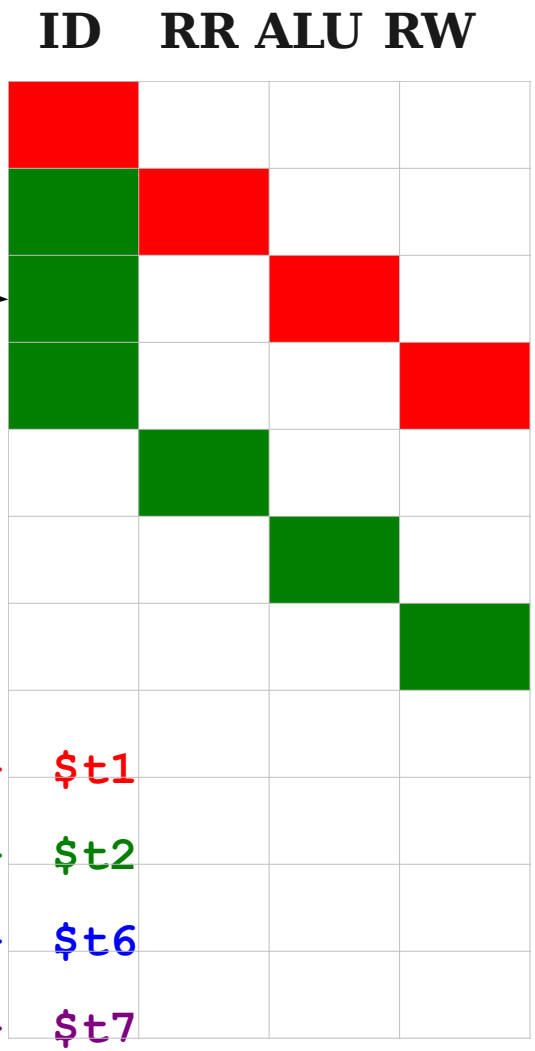



```
add   $t2,   $t0,   $t1      # $t2  =  $t0  +  $t1
add   $t4,   $t3,   $t2      # $t5  =  $t3  +  $t2
add   $t7,   $t5,   $t6      # $t7  =  $t5  +  $t6
add   $t0,   $t0,   $t7      # $t0  =  $t0  +  $t7
```

Pipeline Hazards

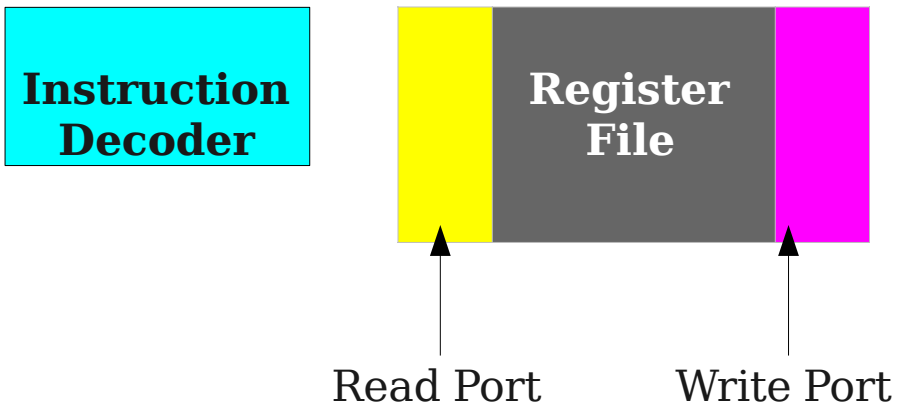
Stall pipeline until value is ready

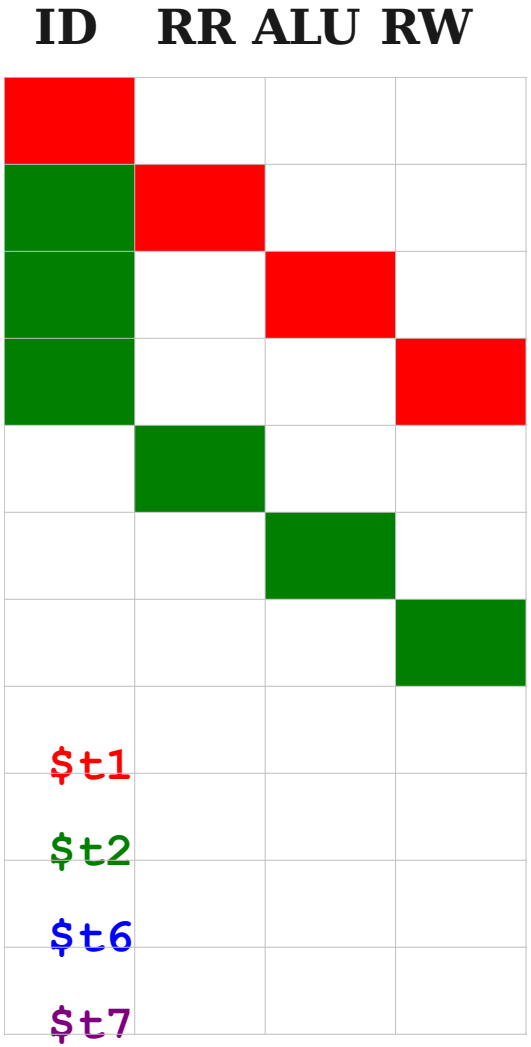





```
add  $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add  $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add  $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add  $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

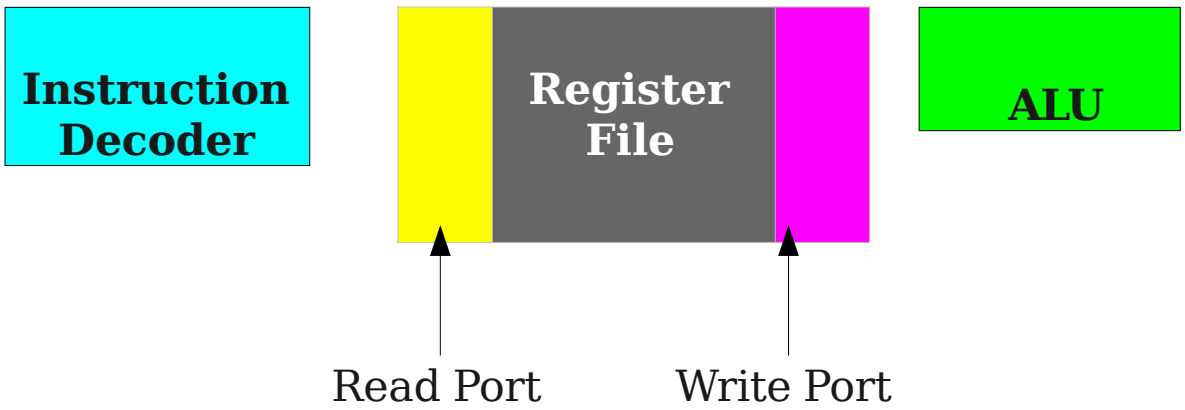
Pipeline Hazards

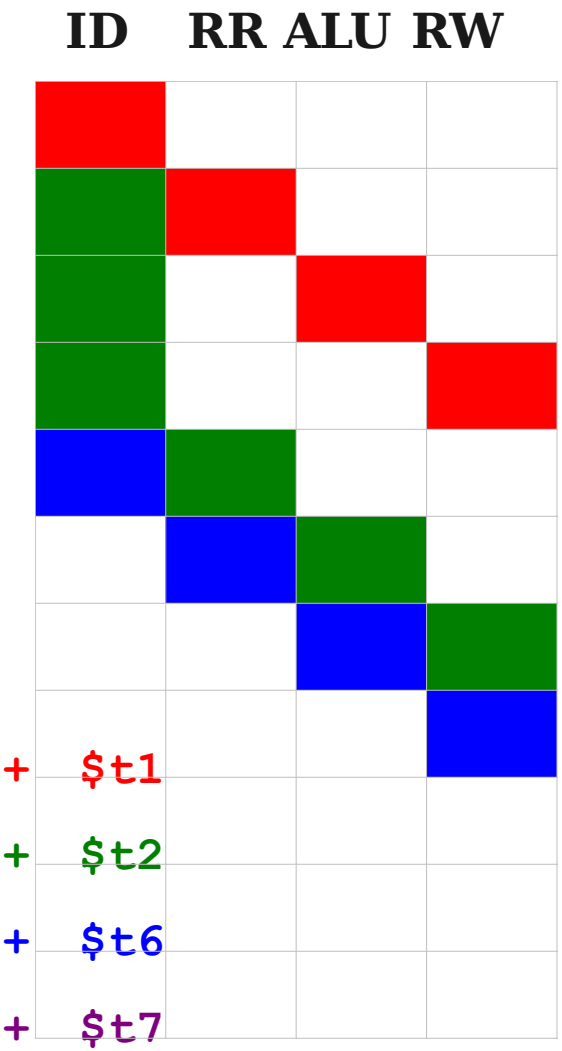





```
add   $t2,   $t0,   $t1      # $t2  =  $t0  +  $t1
add   $t4,   $t3,   $t2      # $t5  =  $t3  +  $t2
add   $t7,   $t5,   $t6      # $t7  =  $t5  +  $t6
add   $t0,   $t0,   $t7      # $t0  =  $t0  +  $t7
```

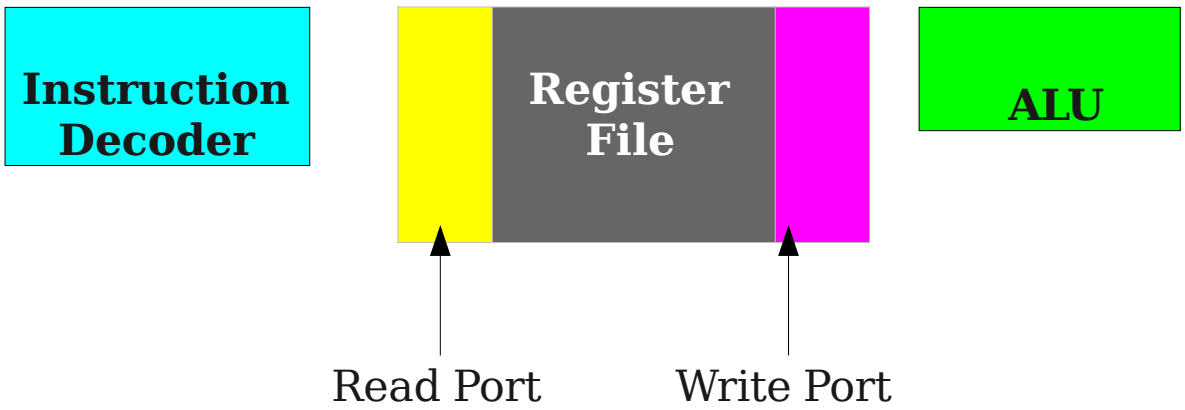
Pipeline Hazards

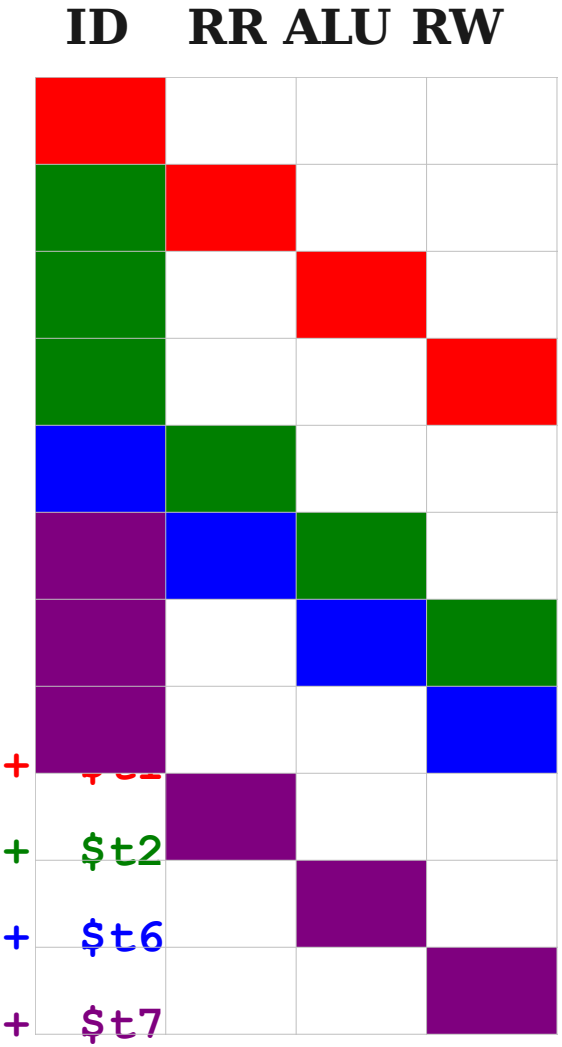





```
add  $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add  $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add  $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add  $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

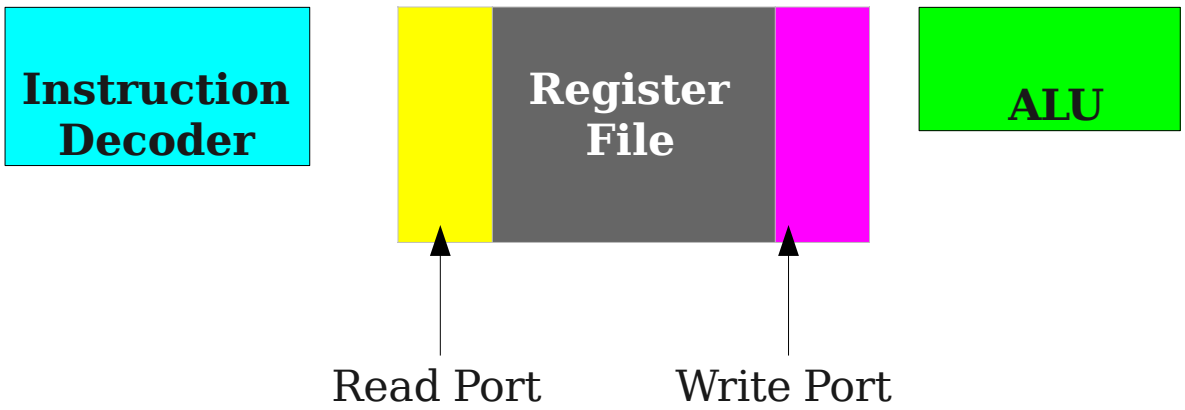
Pipeline Hazards






```
add   $t2,   $t0,   $t1      # $t2 = $t0 + $t1
add   $t4,   $t3,   $t2      # $t5 = $t3 + $t2
add   $t7,   $t5,   $t6      # $t7 = $t5 + $t6
add   $t0,   $t0,   $t7      # $t0 = $t0 + $t7
```

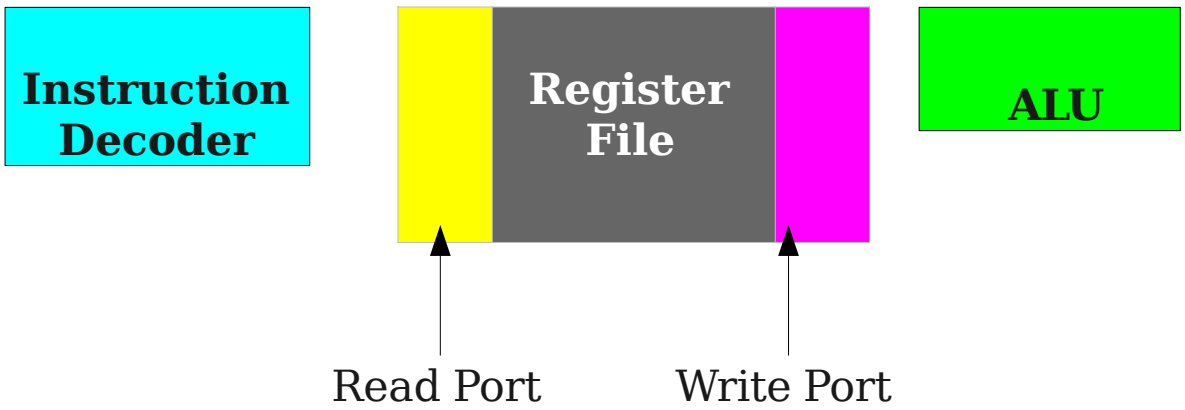
Pipeline Hazards



ID	RR	ALU	RW

```
add  $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add  $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add  $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add  $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

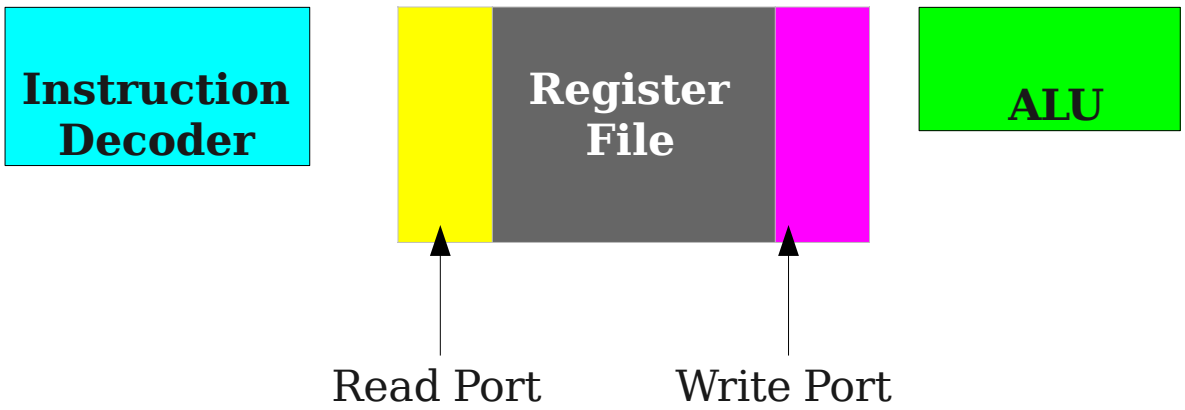
Pipeline Hazards

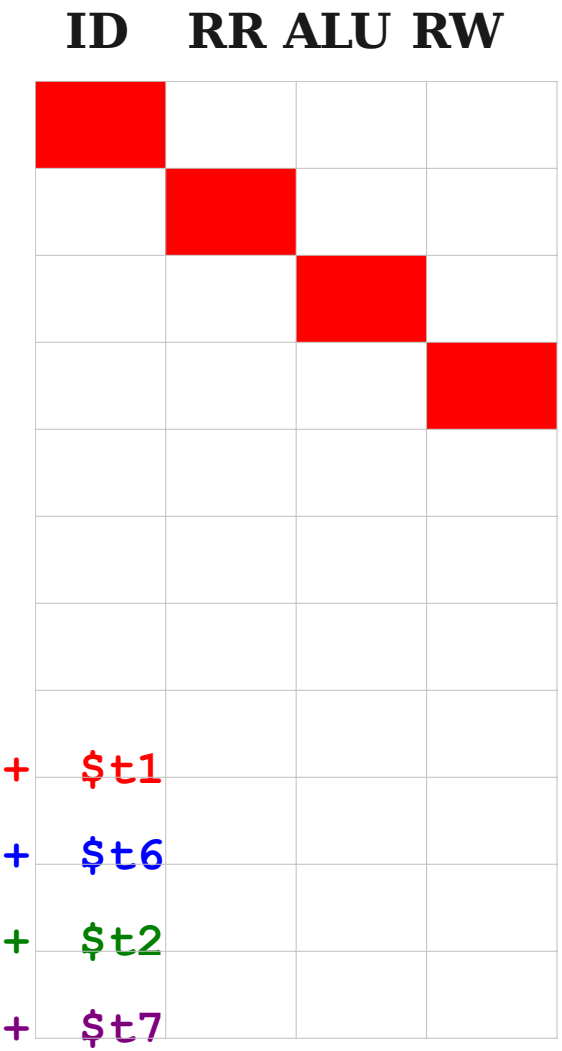


ID	RR	ALU	RW

```
add  $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add  $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add  $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add  $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

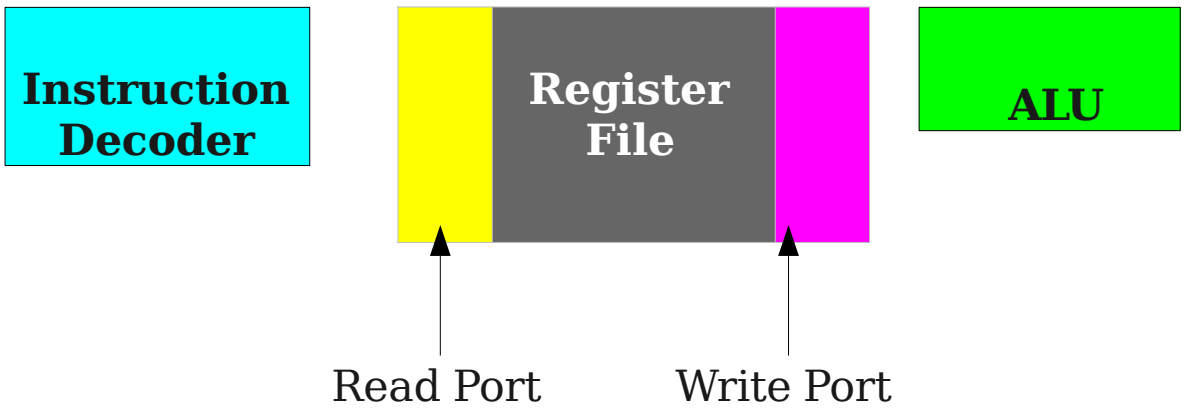
Pipeline Hazards

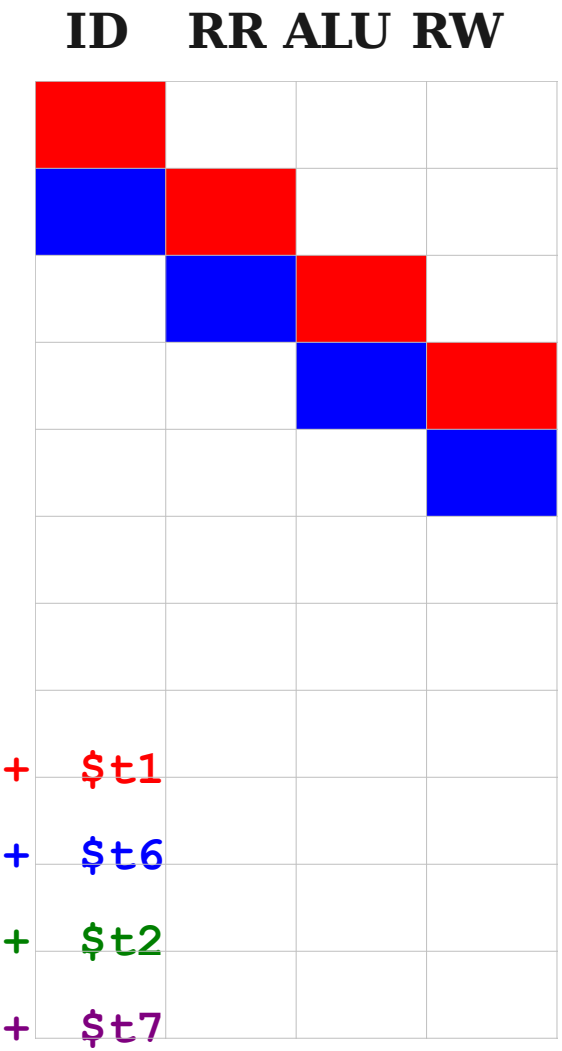





```
add  $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add  $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add  $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add  $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

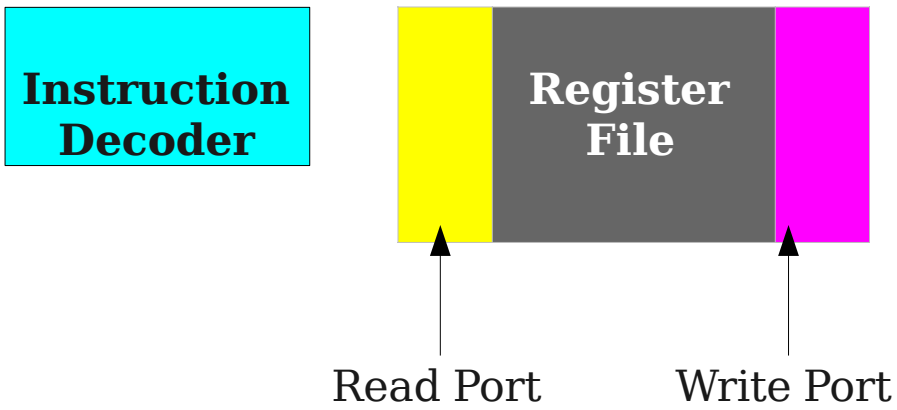
Pipeline Hazards

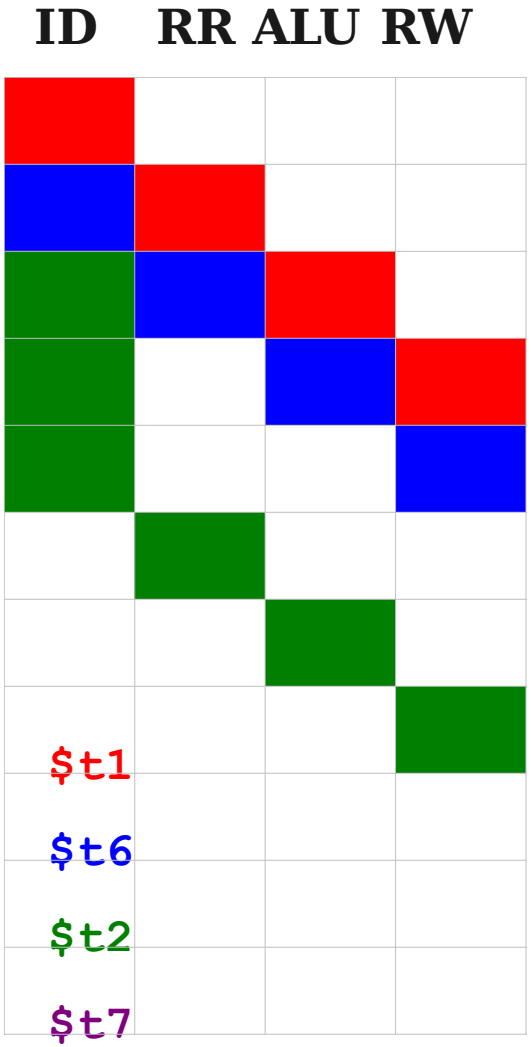





```
add   $t2,   $t0,   $t1      # $t2 = $t0 + $t1
add   $t7,   $t5,   $t6      # $t7 = $t5 + $t6
add   $t4,   $t3,   $t2      # $t5 = $t3 + $t2
add   $t0,   $t0,   $t7      # $t0 = $t0 + $t7
```

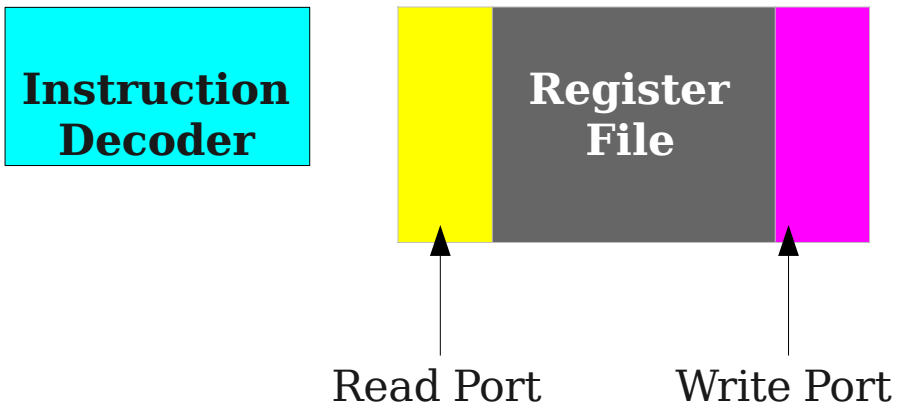
Pipeline Hazards

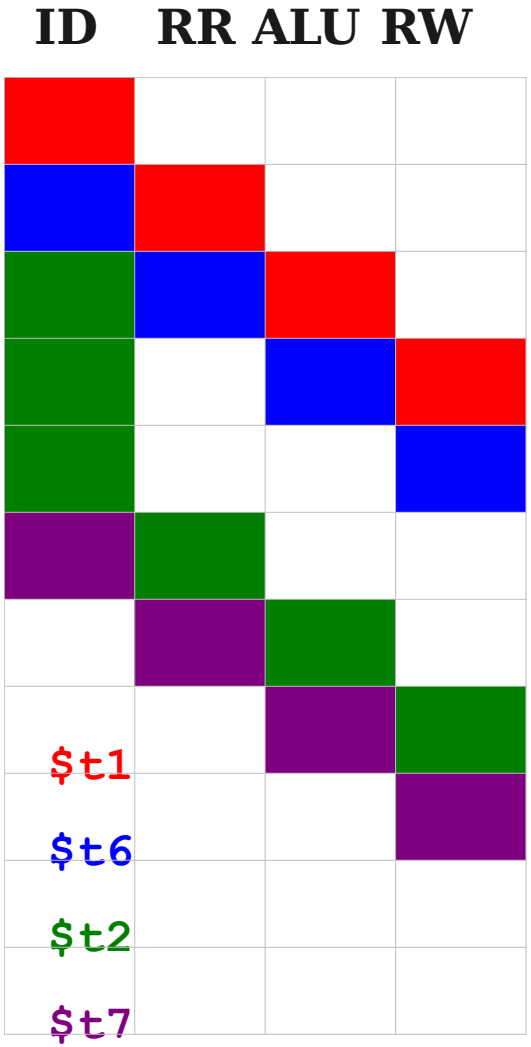





```
add  $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add  $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add  $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add  $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

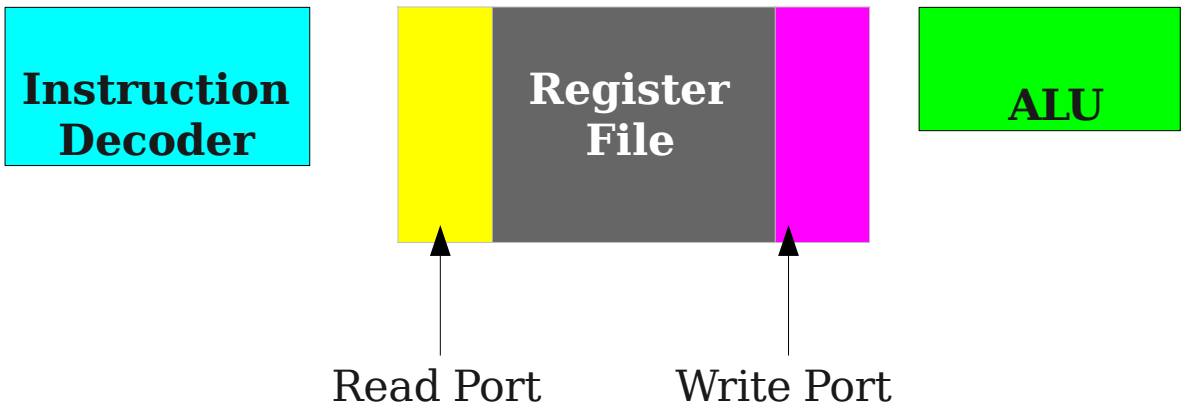
Pipeline Hazards






```
add  $t2,  $t0,  $t1     # $t2 = $t0 + $t1
add  $t7,  $t5,  $t6     # $t7 = $t5 + $t6
add  $t4,  $t3,  $t2     # $t5 = $t3 + $t2
add  $t0,  $t0,  $t7     # $t0 = $t0 + $t7
```

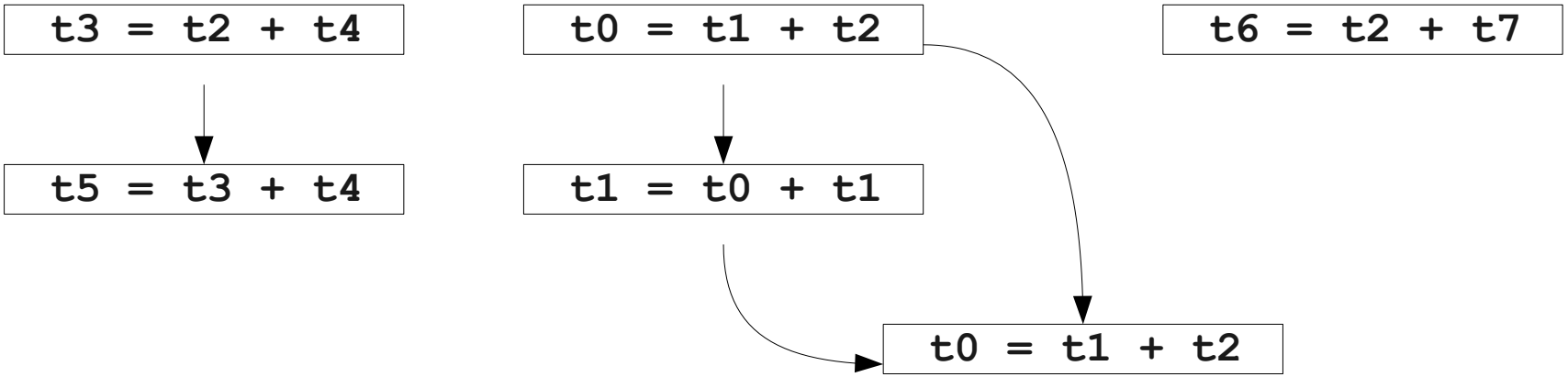
Pipeline Hazards




```
add   $t2,   $t0,   $t1     # $t2 = $t0 + $t1
add   $t7,   $t5,   $t6     # $t7 = $t5 + $t6
add   $t4,   $t3,   $t2     # $t5 = $t3 +
add   $t0,   $t0,   $t7     # $t0 = $t0 +
```

Two clock cycles faster!

Instruction Scheduling



Instruction Scheduling

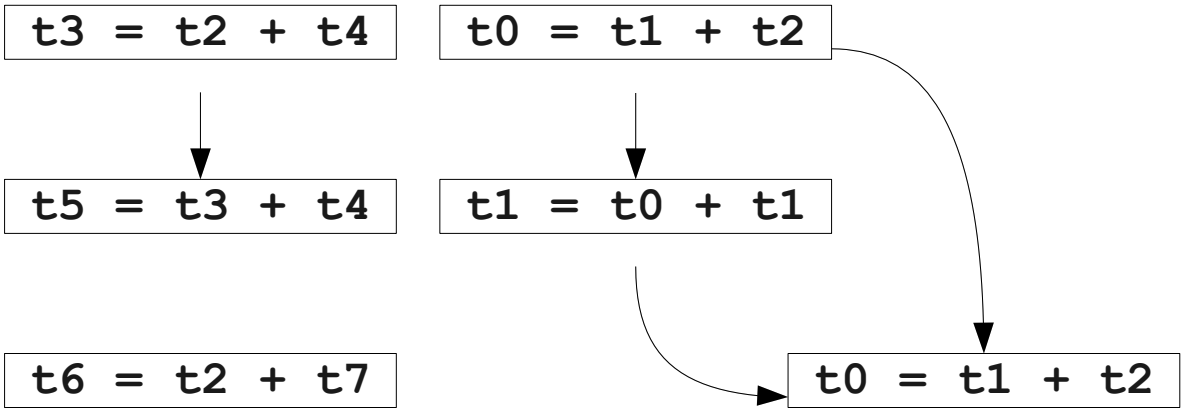

```
t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2
t1 = t0 + t1
t0 = t1 + t2
t6 = t2 + t7
```

```
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4
t0 = t1 + t2
```

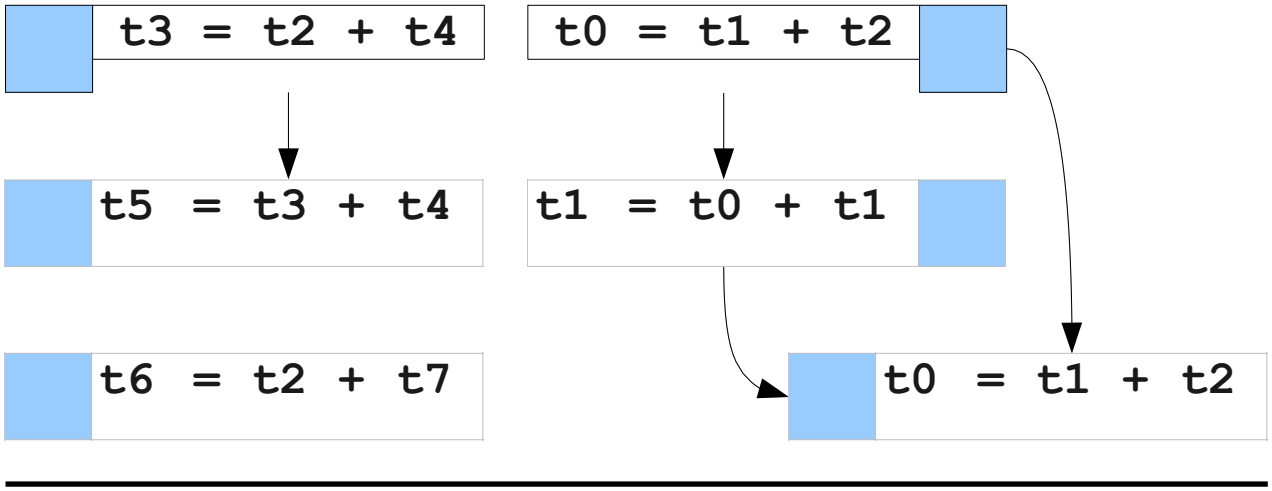
One Small Problem

- There can be many valid topological orderings of a data dependency graph.
- How do we pick one that works well with the pipeline?
- In general, finding the fastest instruction schedule is known to be **NP-hard**.
  - Don't expect a polynomial-time algorithm anytime soon!
- Heuristics are used in practice:
  - Schedule instructions that can run to completion without interference before instructions that cause interference.
  - Schedule instructions with more dependents before instructions with fewer dependents.

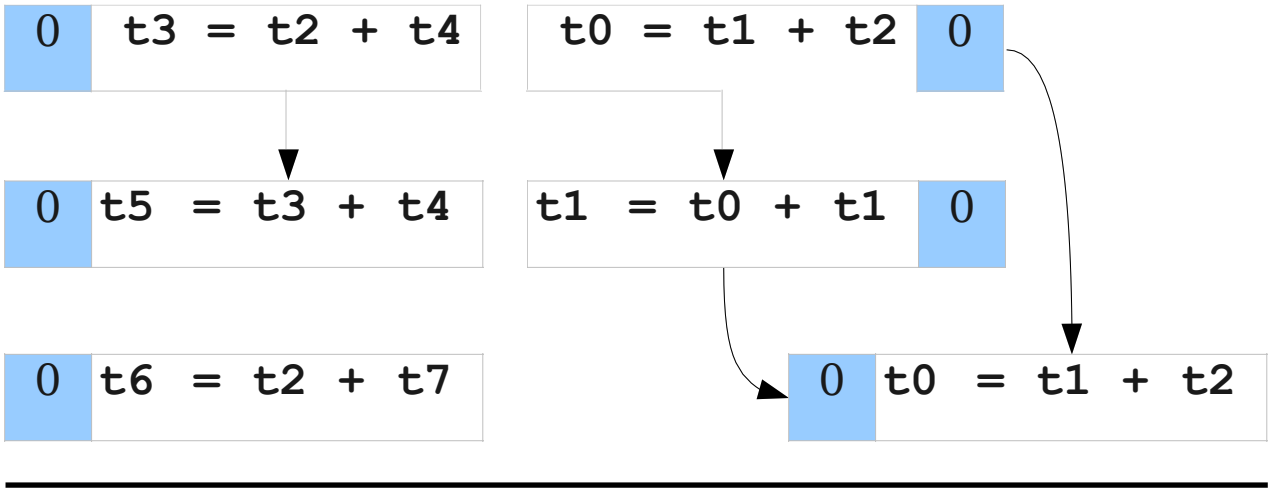
Instruction Scheduling



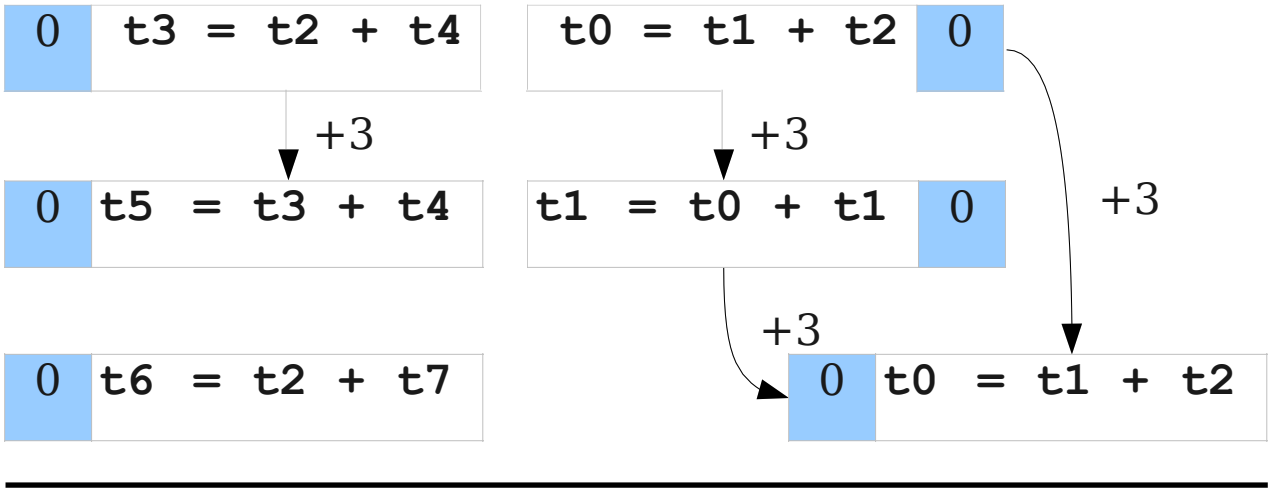
Instruction Scheduling



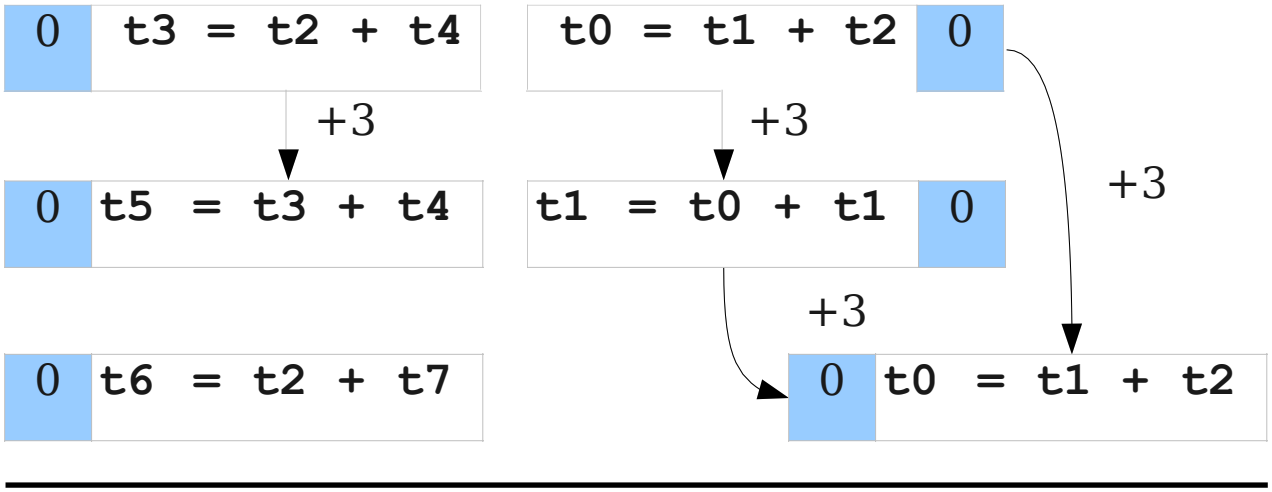
Instruction Scheduling



Instruction Scheduling

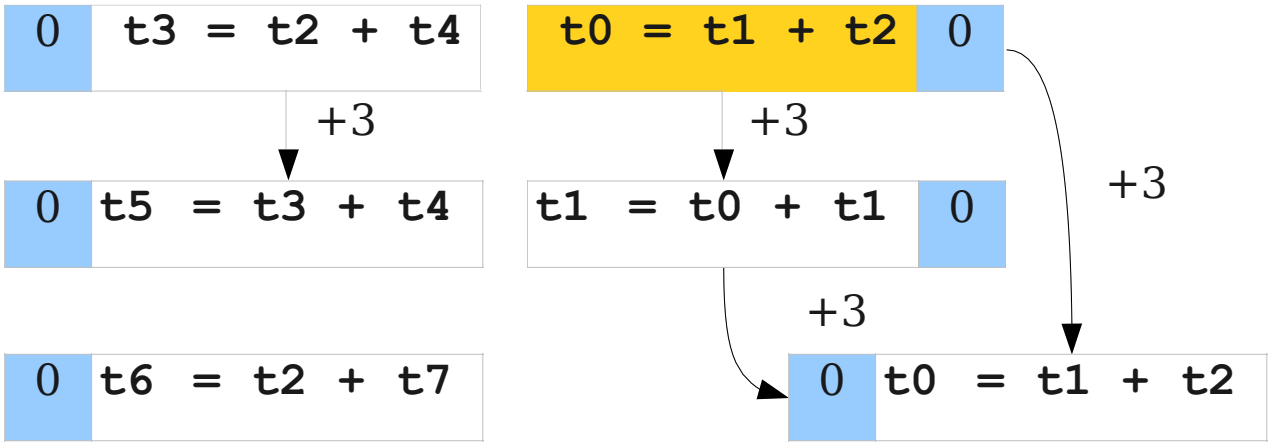


Instruction Scheduling



ID	RR	ALU	RW

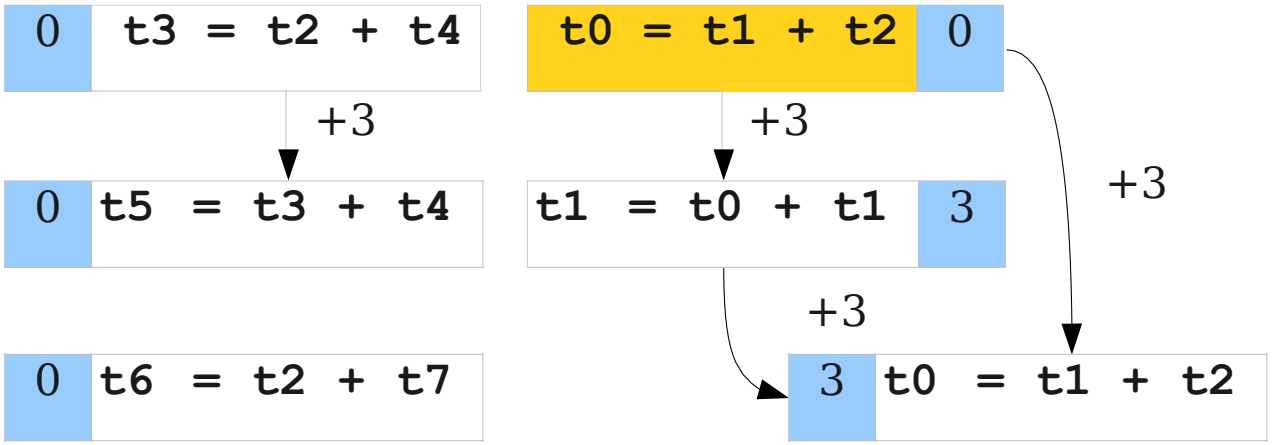
Instruction Scheduling




```
t0 = t1 + t2
```

ID	RR	ALU	RW

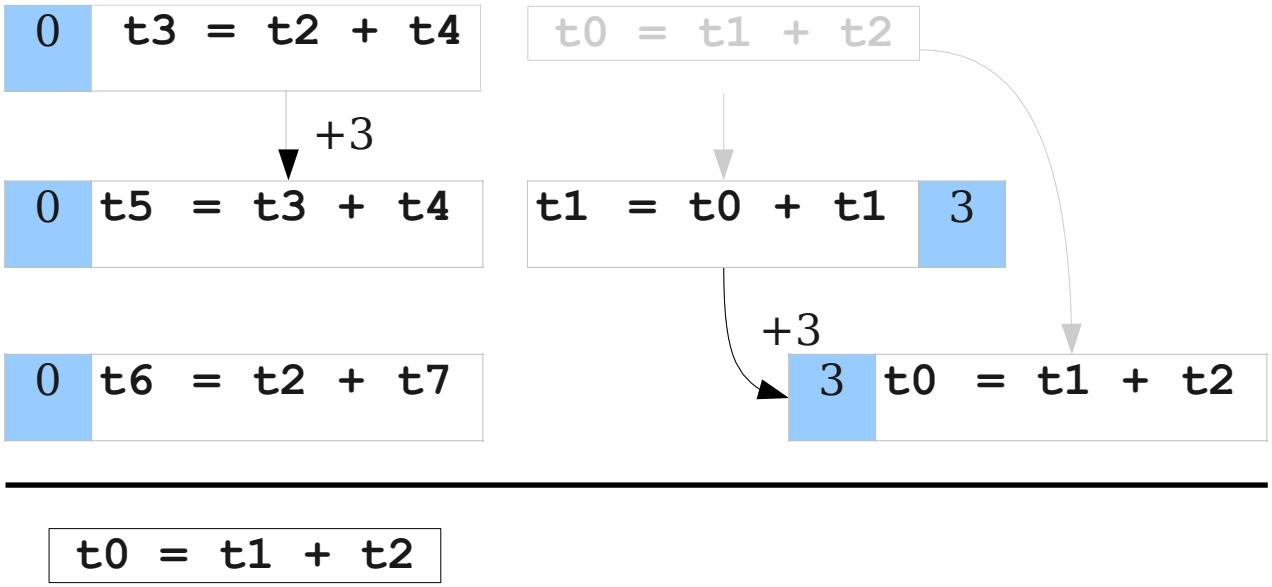
Instruction Scheduling




```
t0 = t1 + t2
```

ID	RR	ALU	RW

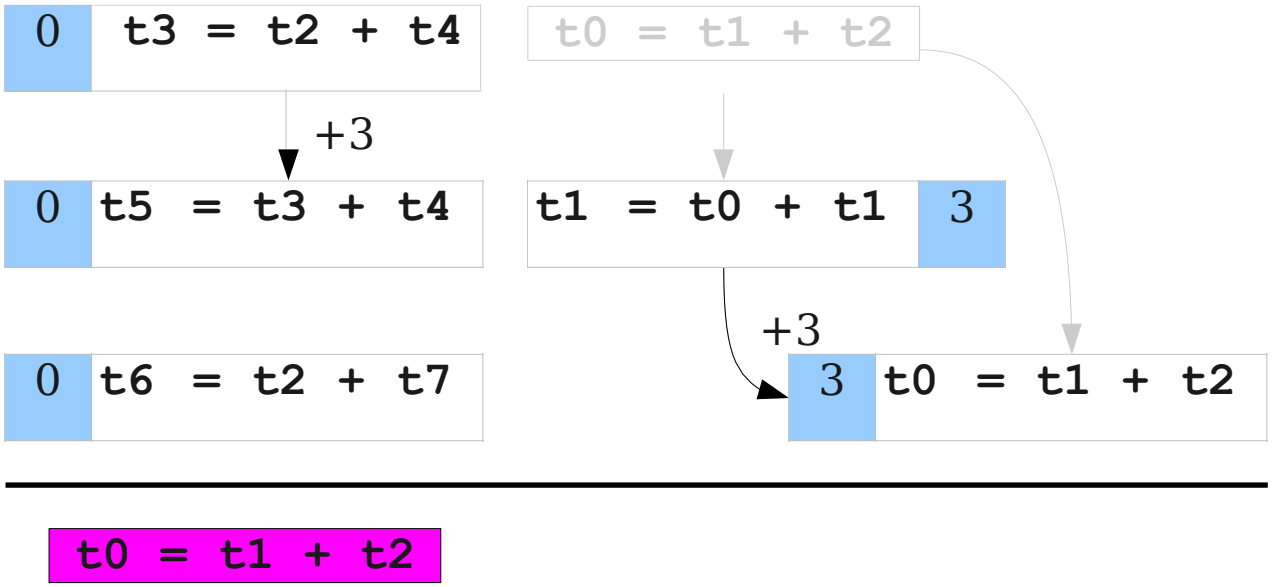
Instruction Scheduling



ID	RR	ALU	RW

```
t0 = t1 + t2
```

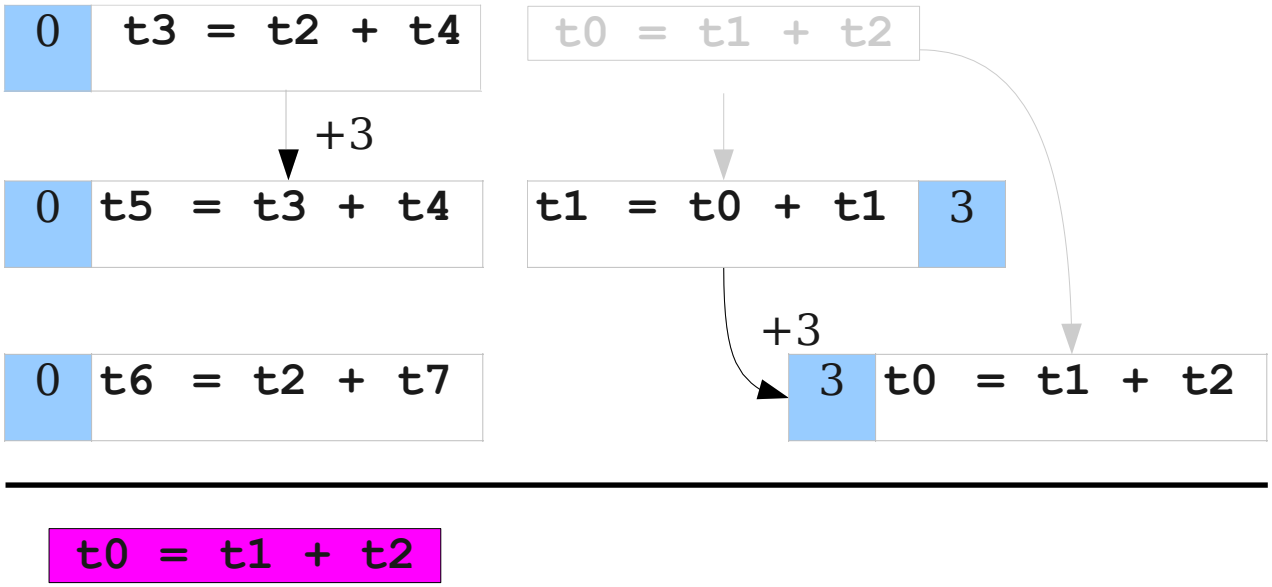
Instruction Scheduling

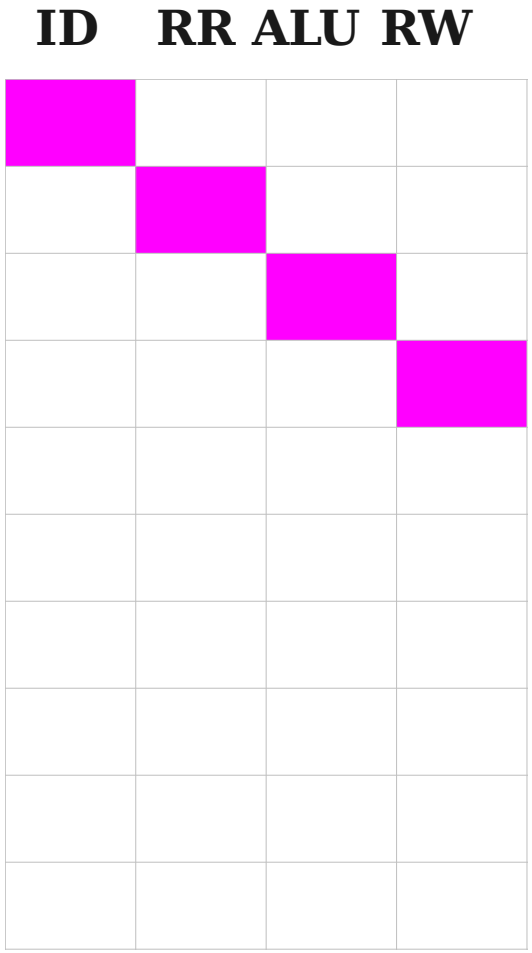


ID	RR	ALU	RW

```
t0 = t1 + t2
```

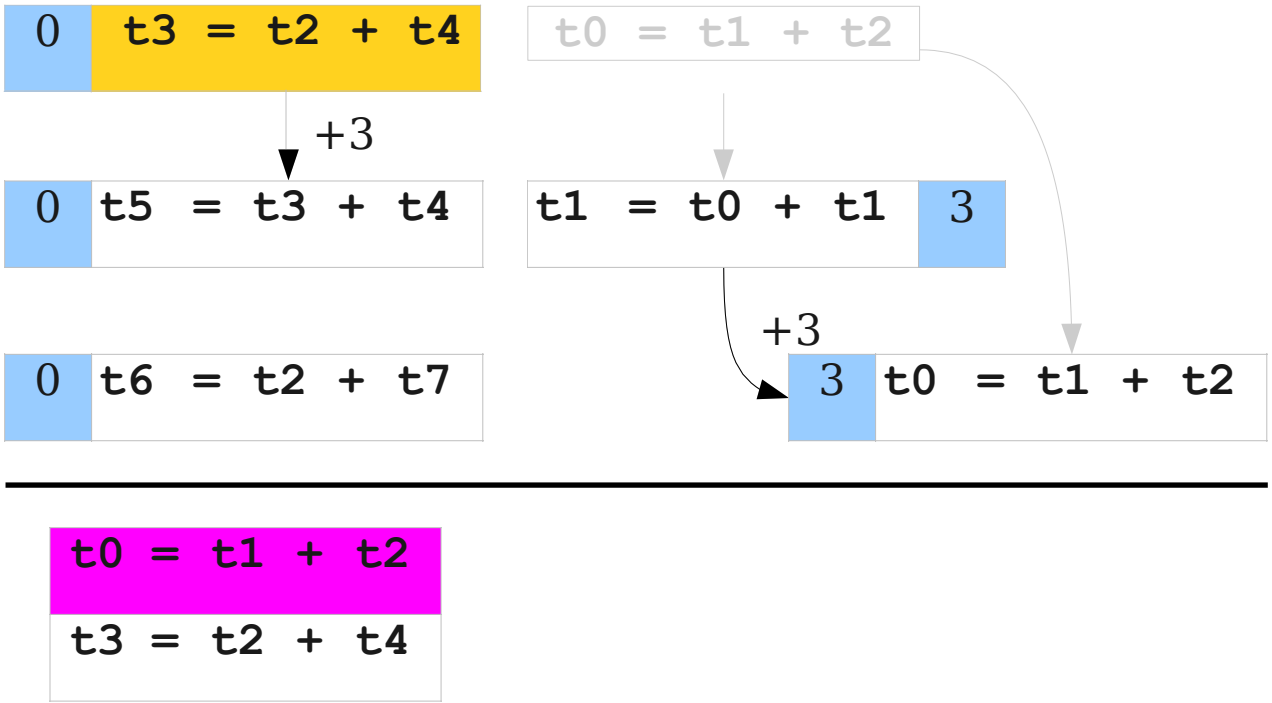
Instruction Scheduling

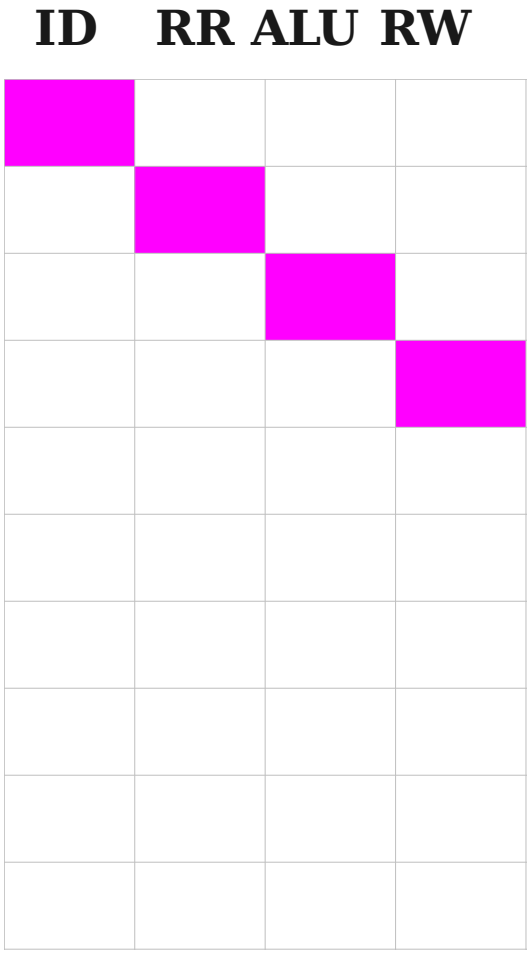




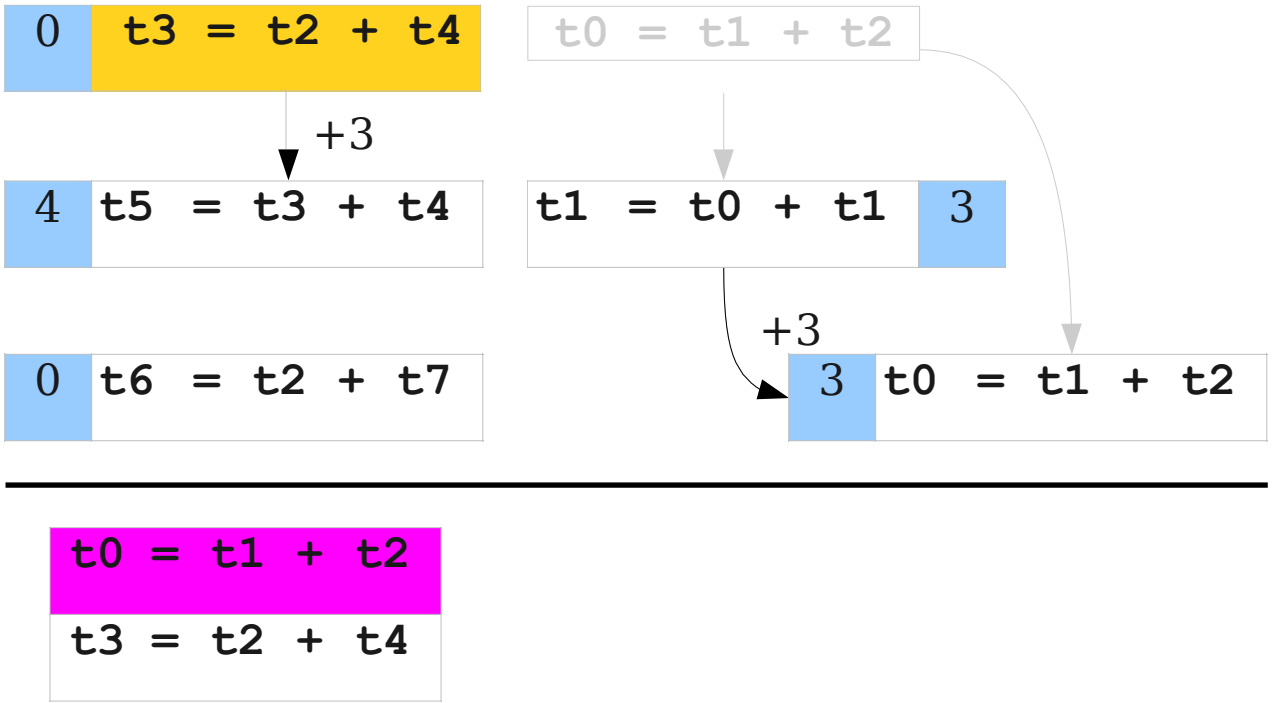
t0 = t1 + t2

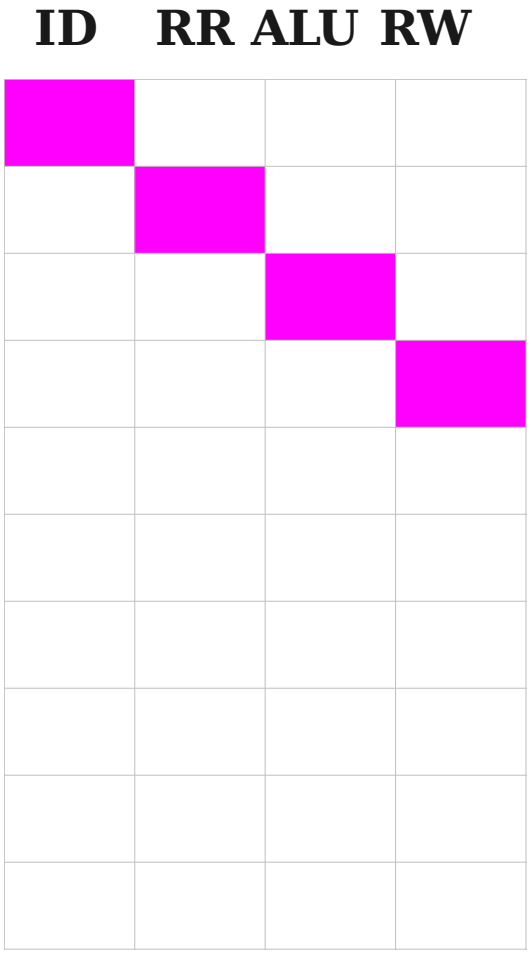
Instruction Scheduling



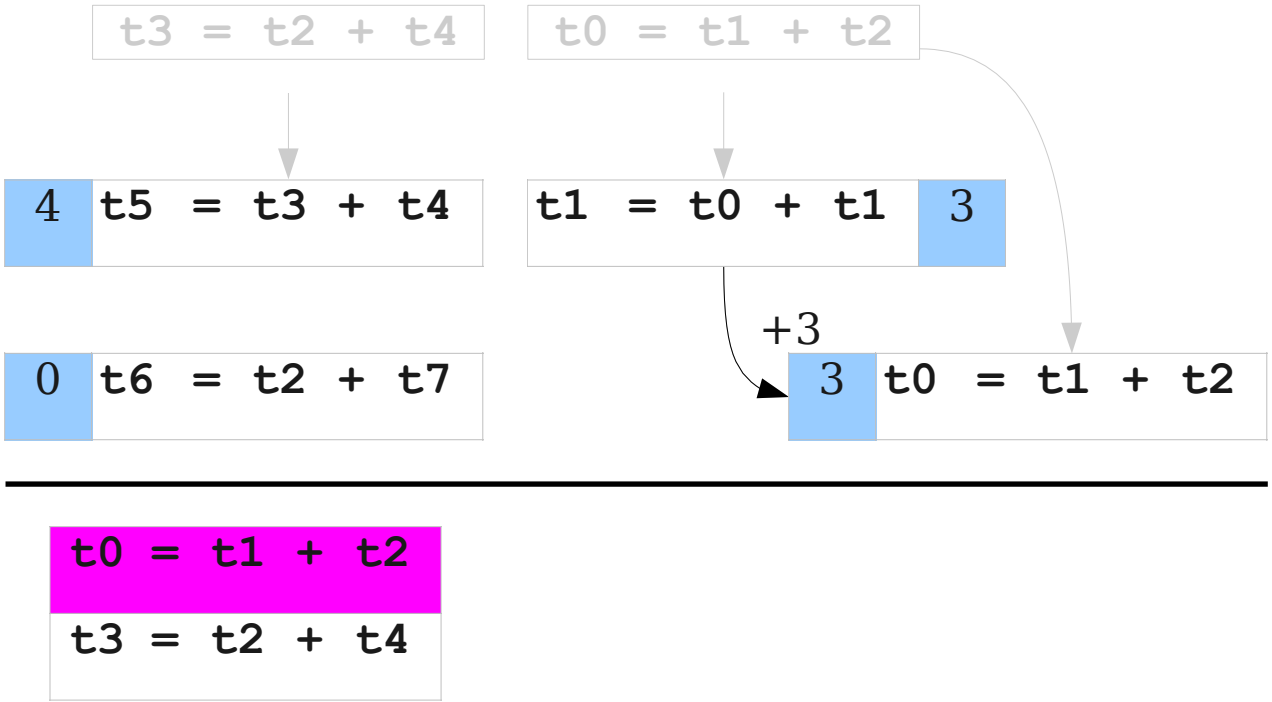


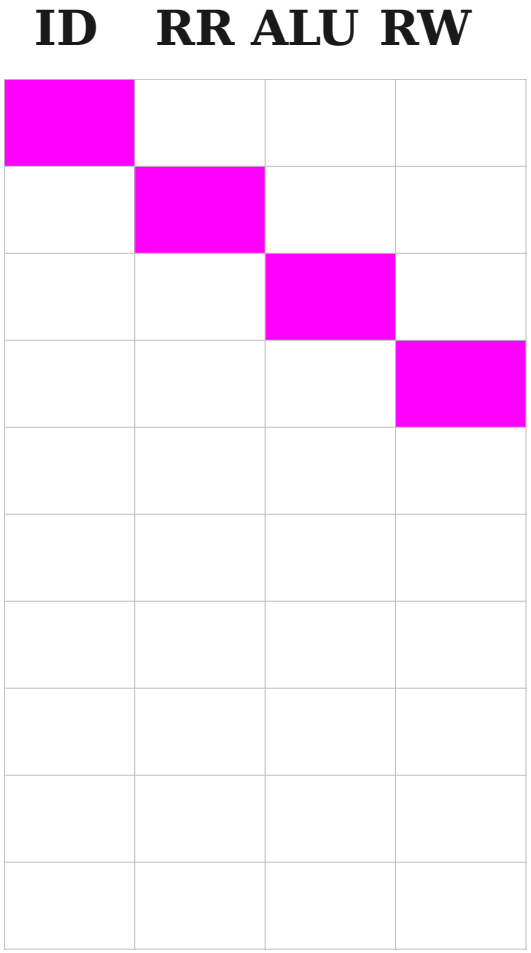
Instruction Scheduling



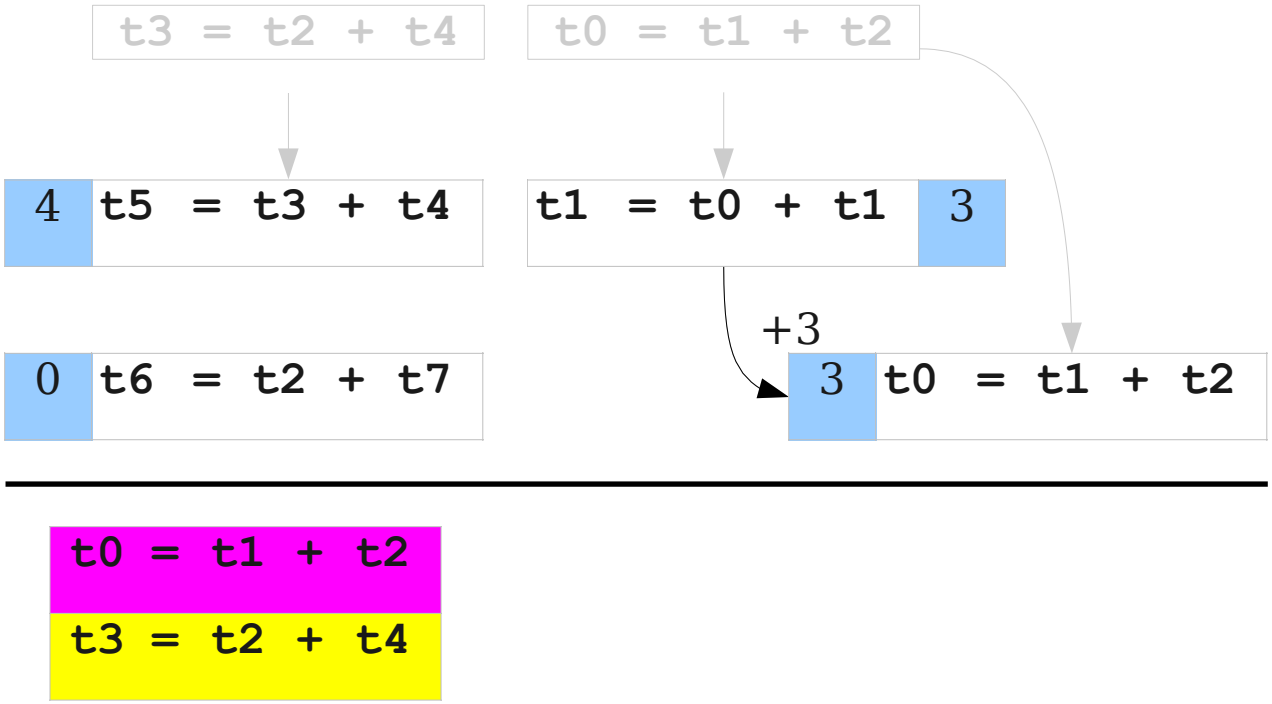


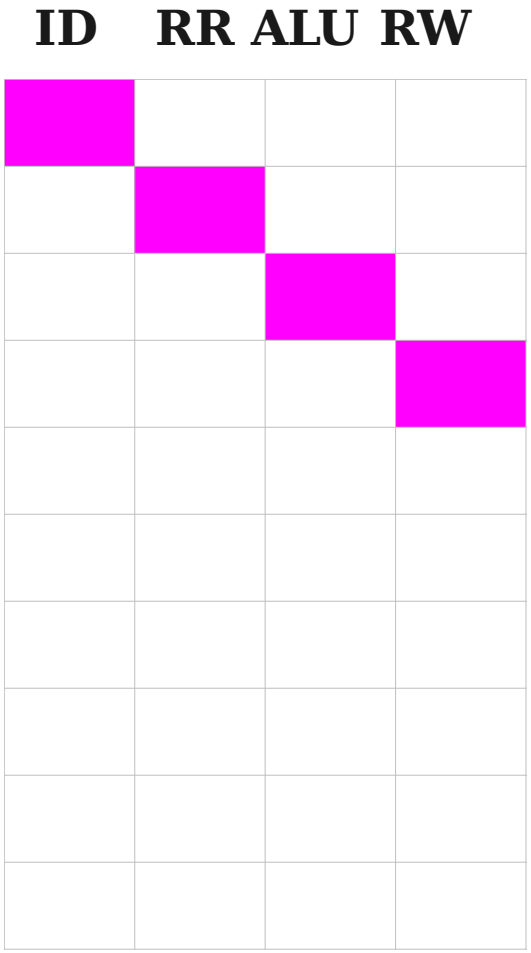
Instruction Scheduling



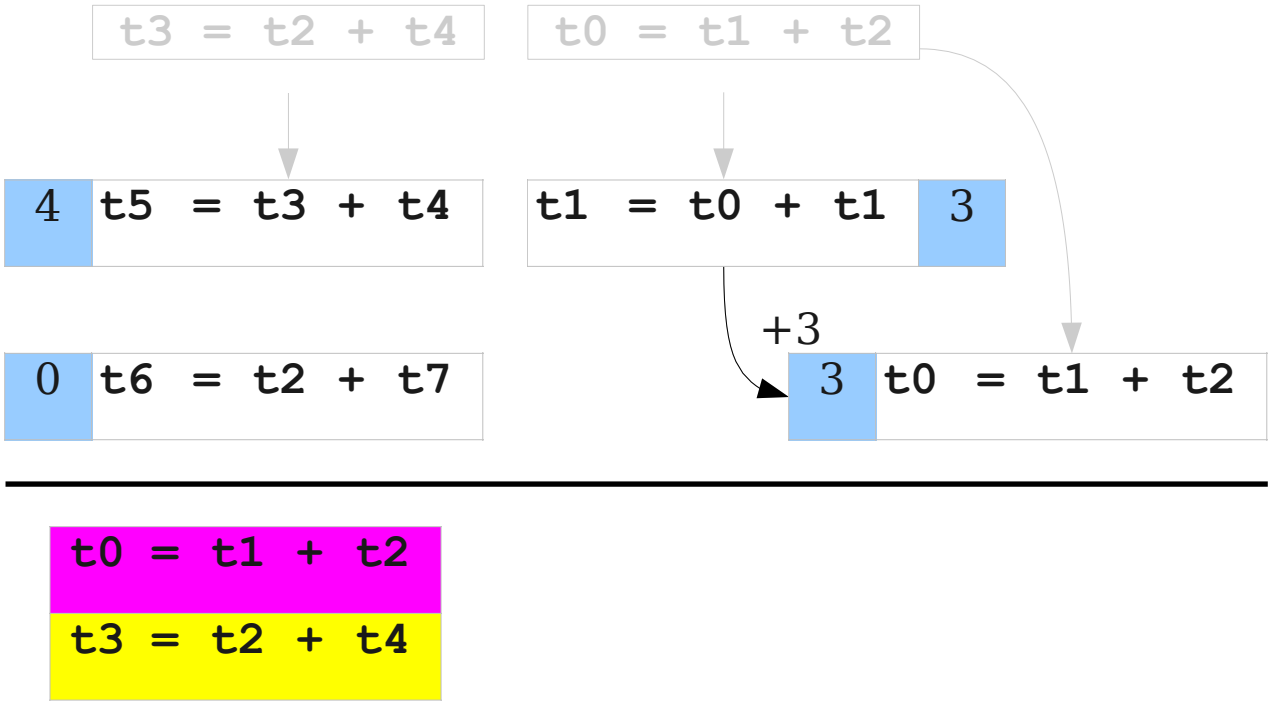


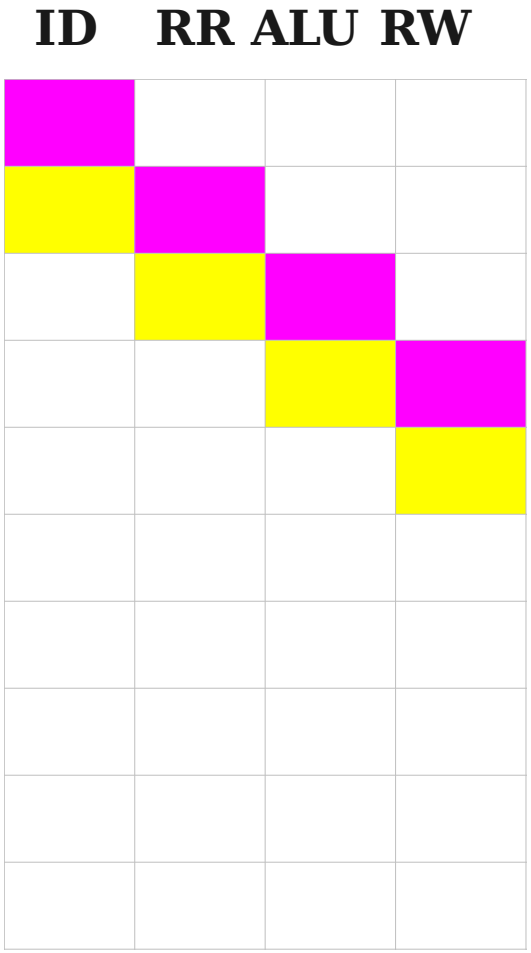
Instruction Scheduling



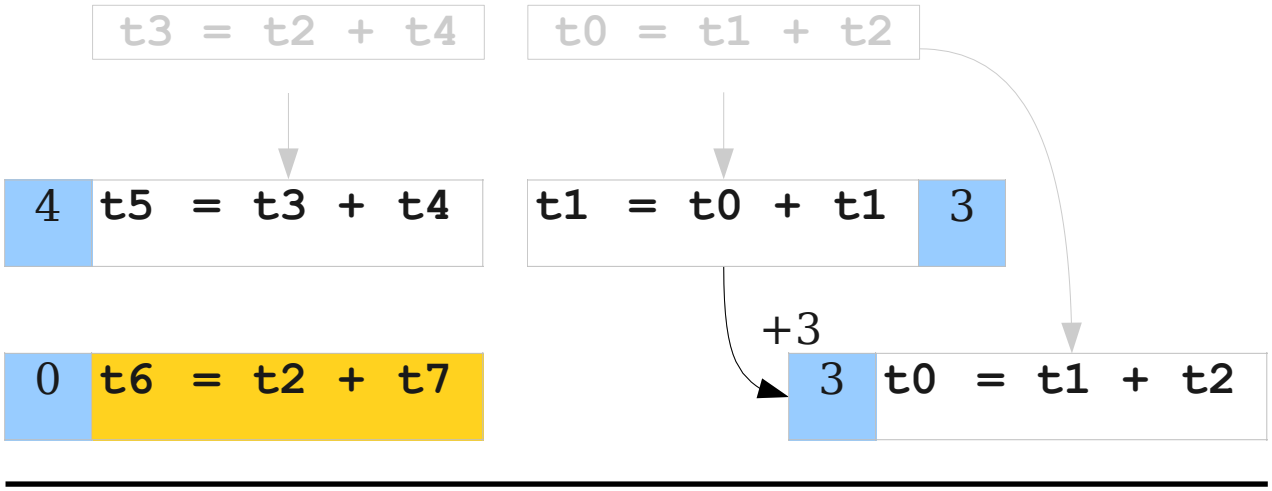


Instruction Scheduling

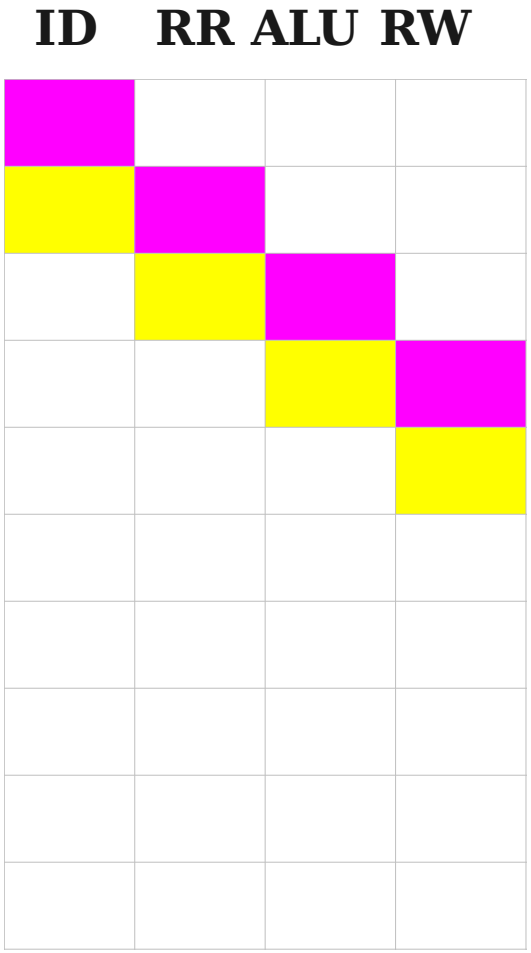




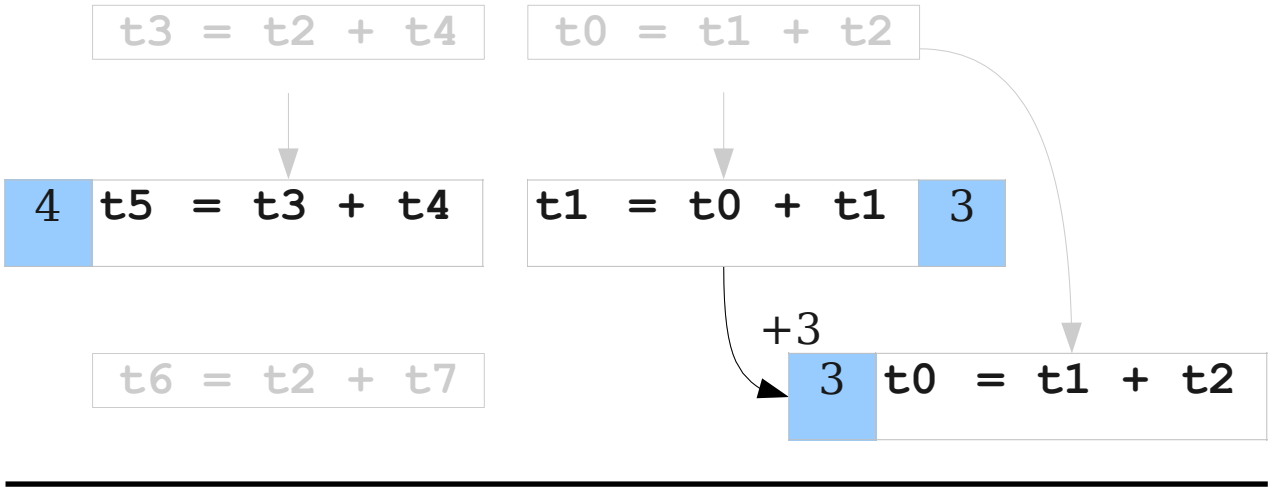
Instruction Scheduling



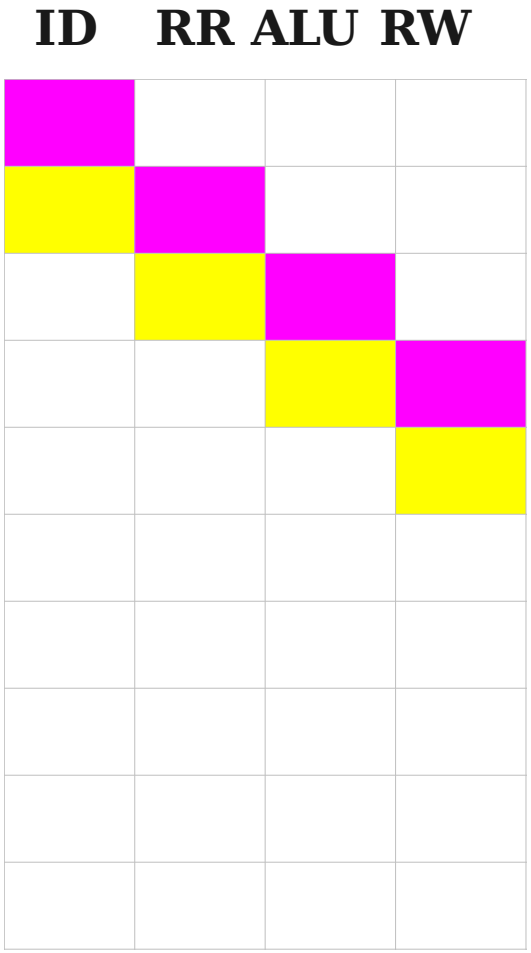

```
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
```



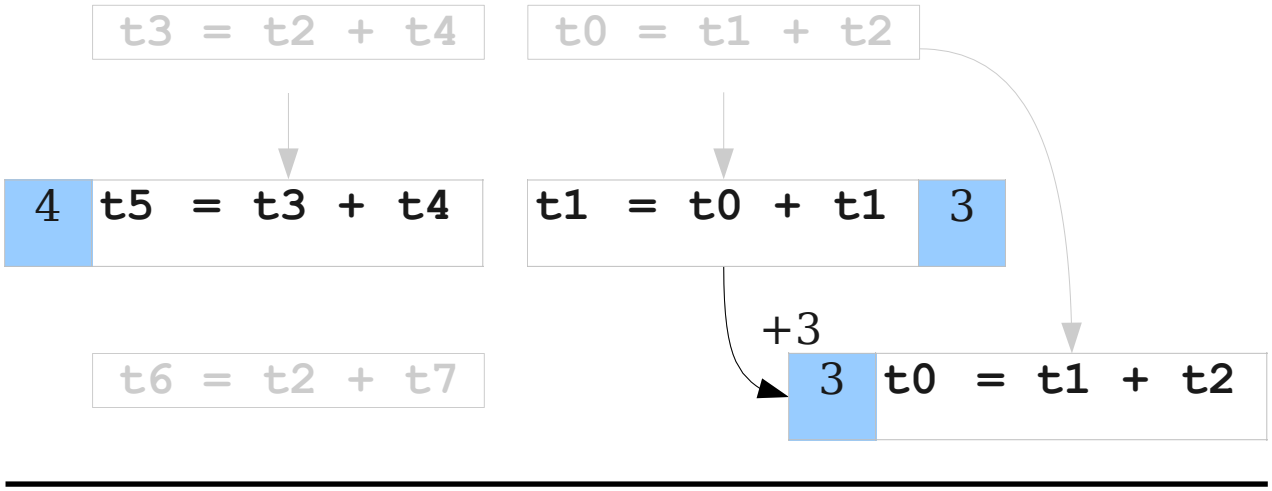
Instruction Scheduling

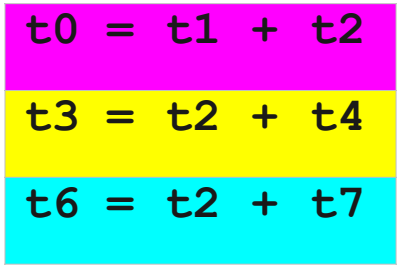


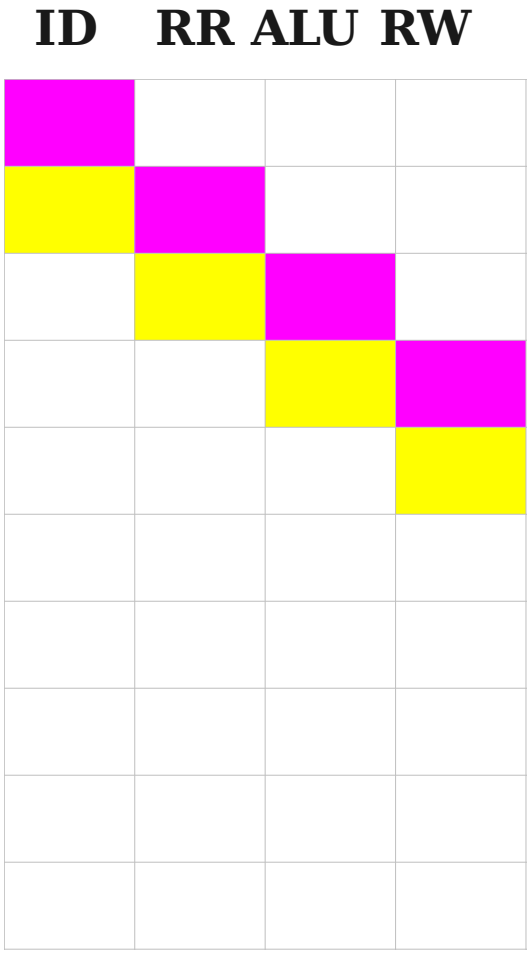

```
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
```



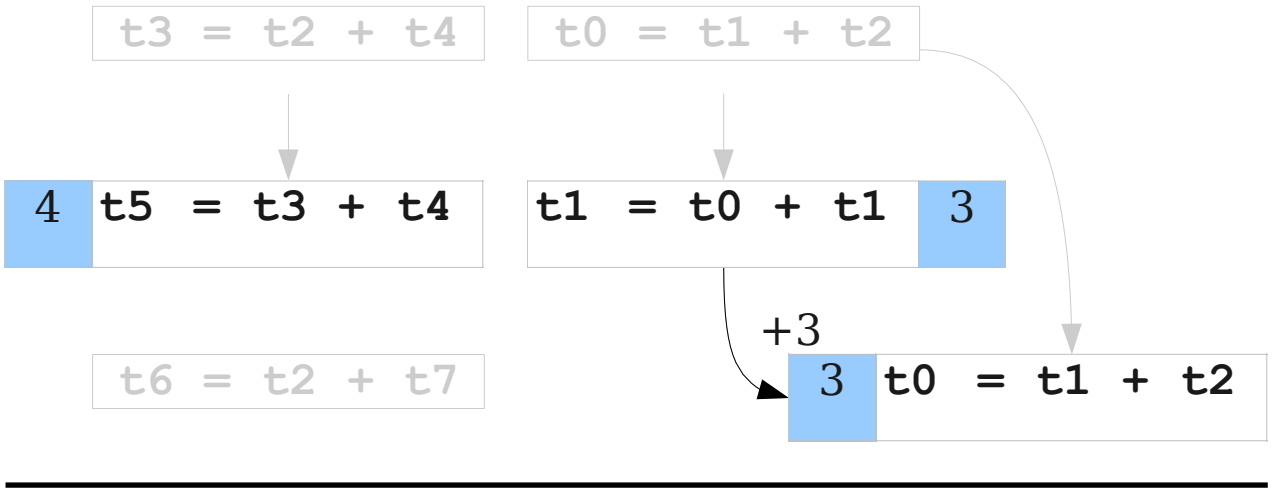
Instruction Scheduling

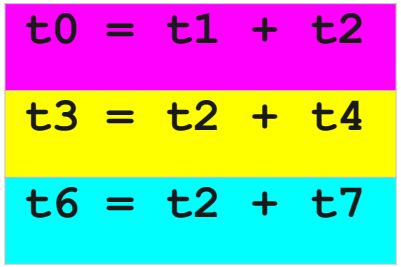


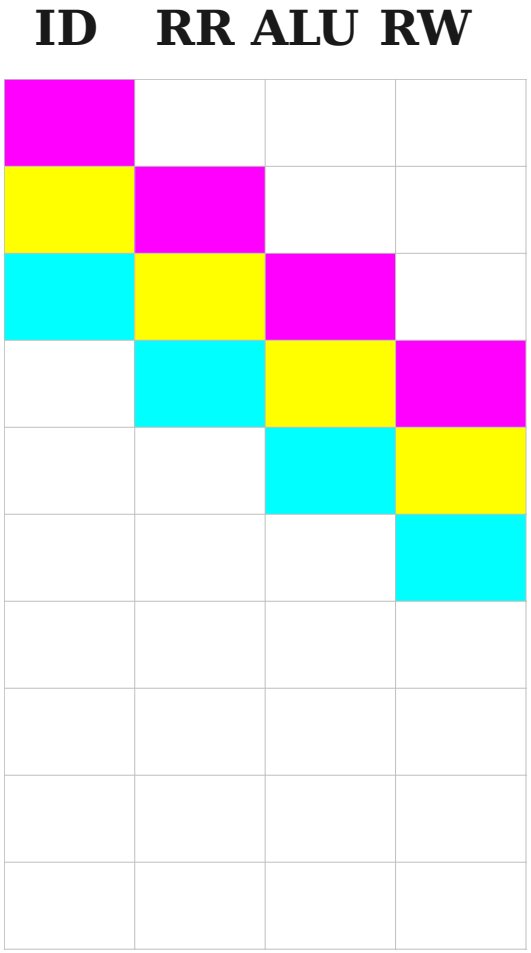




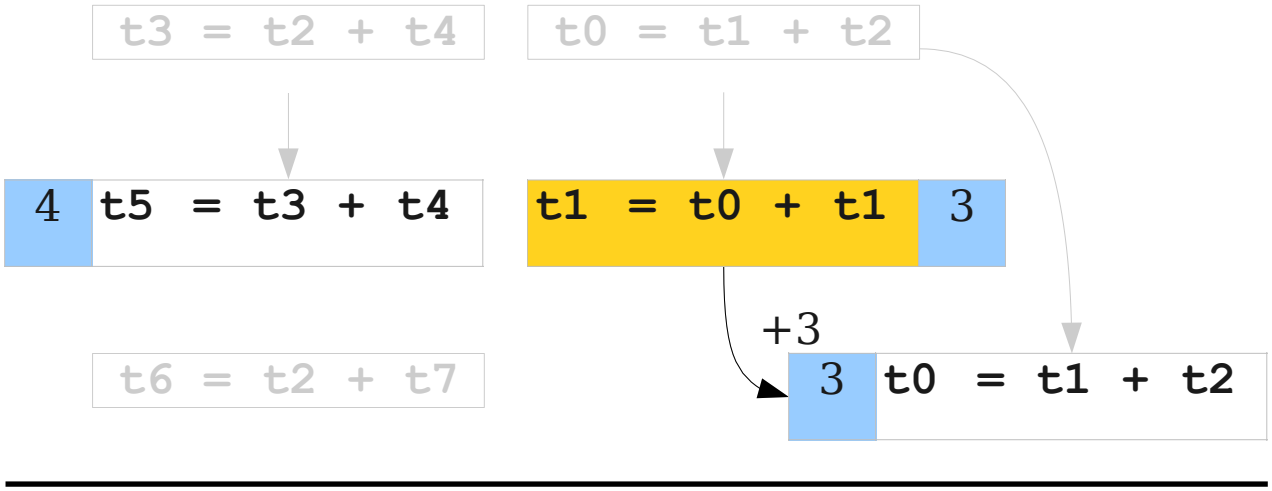
Instruction Scheduling

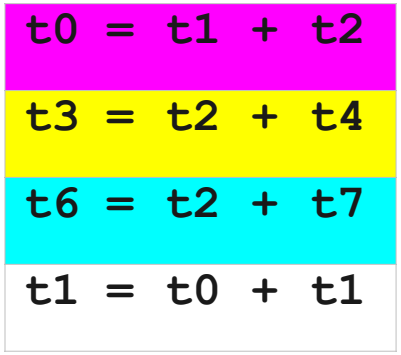


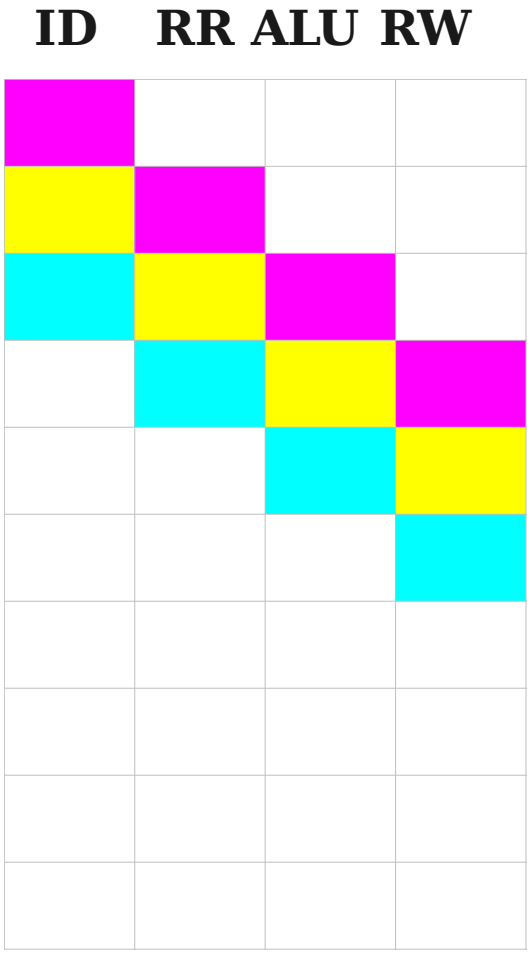




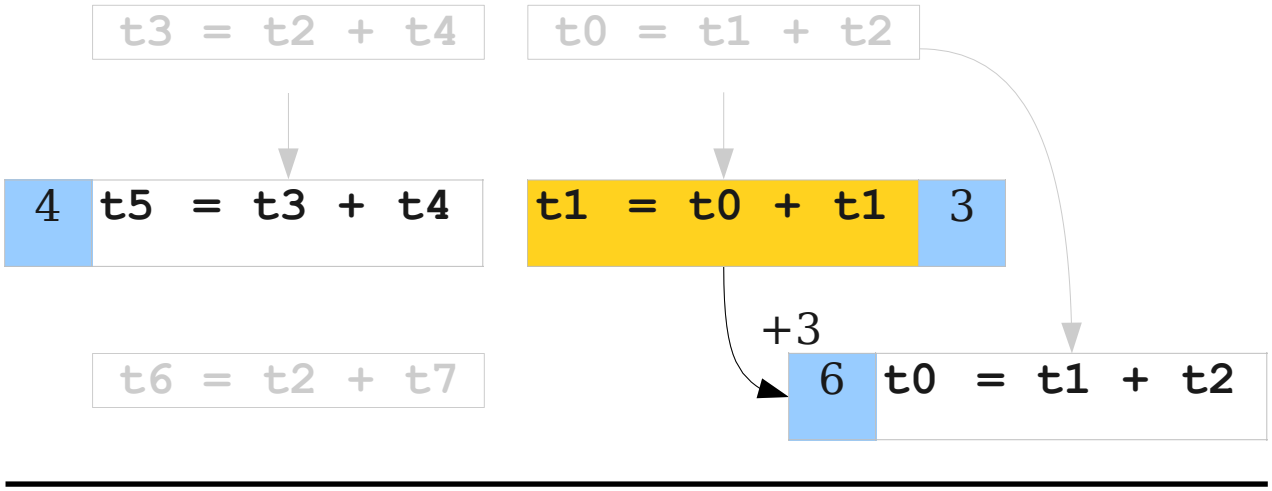
Instruction Scheduling

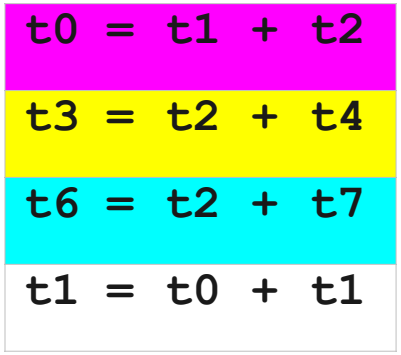


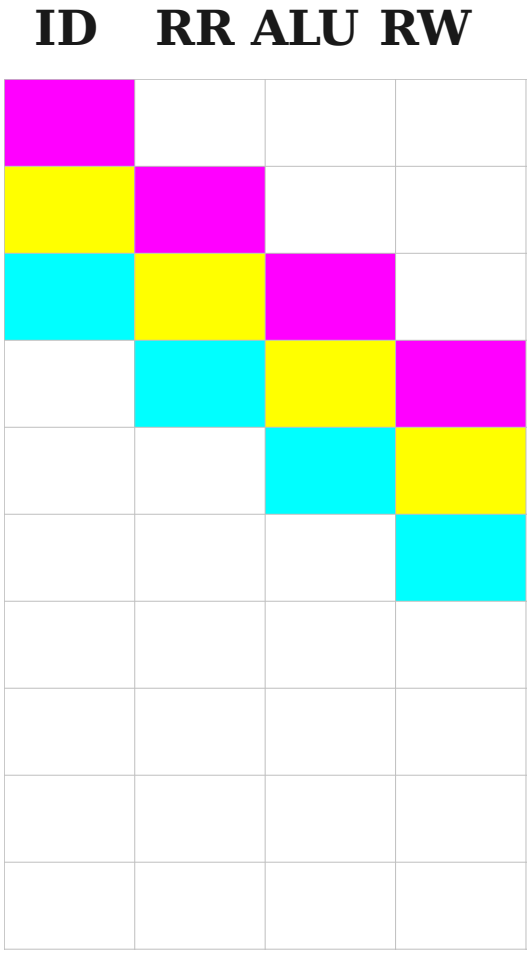




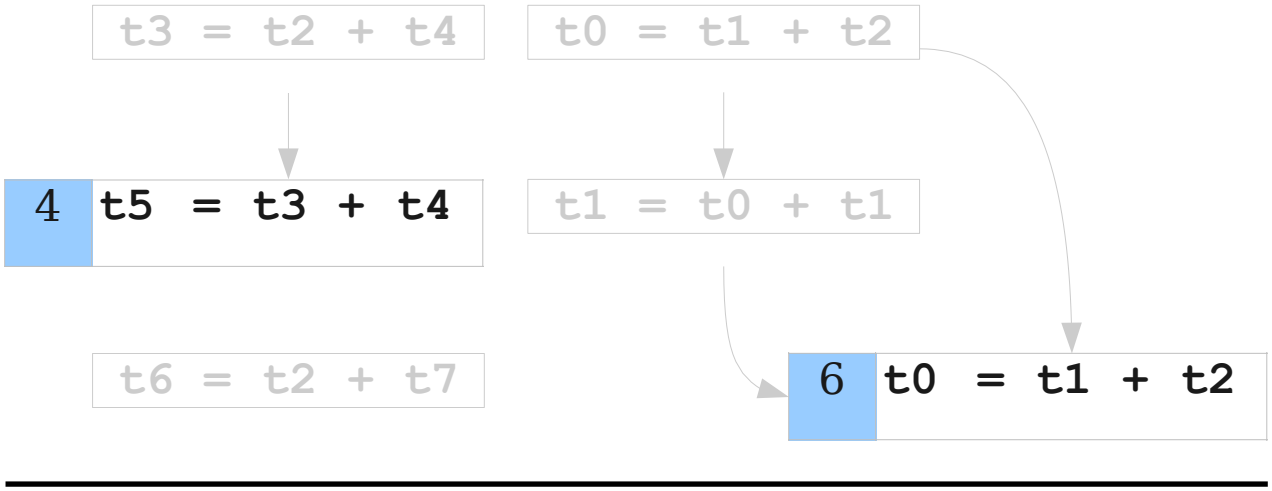
Instruction Scheduling

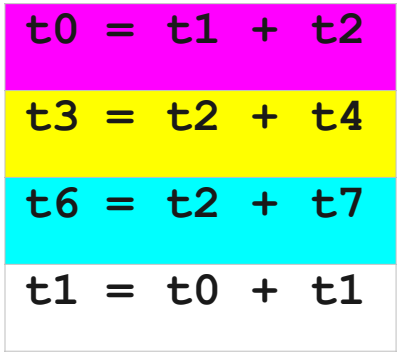


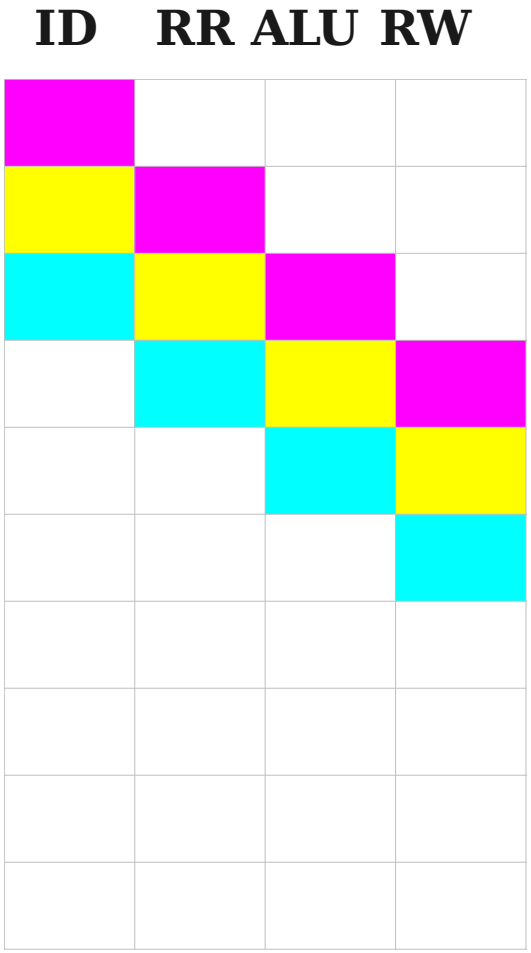




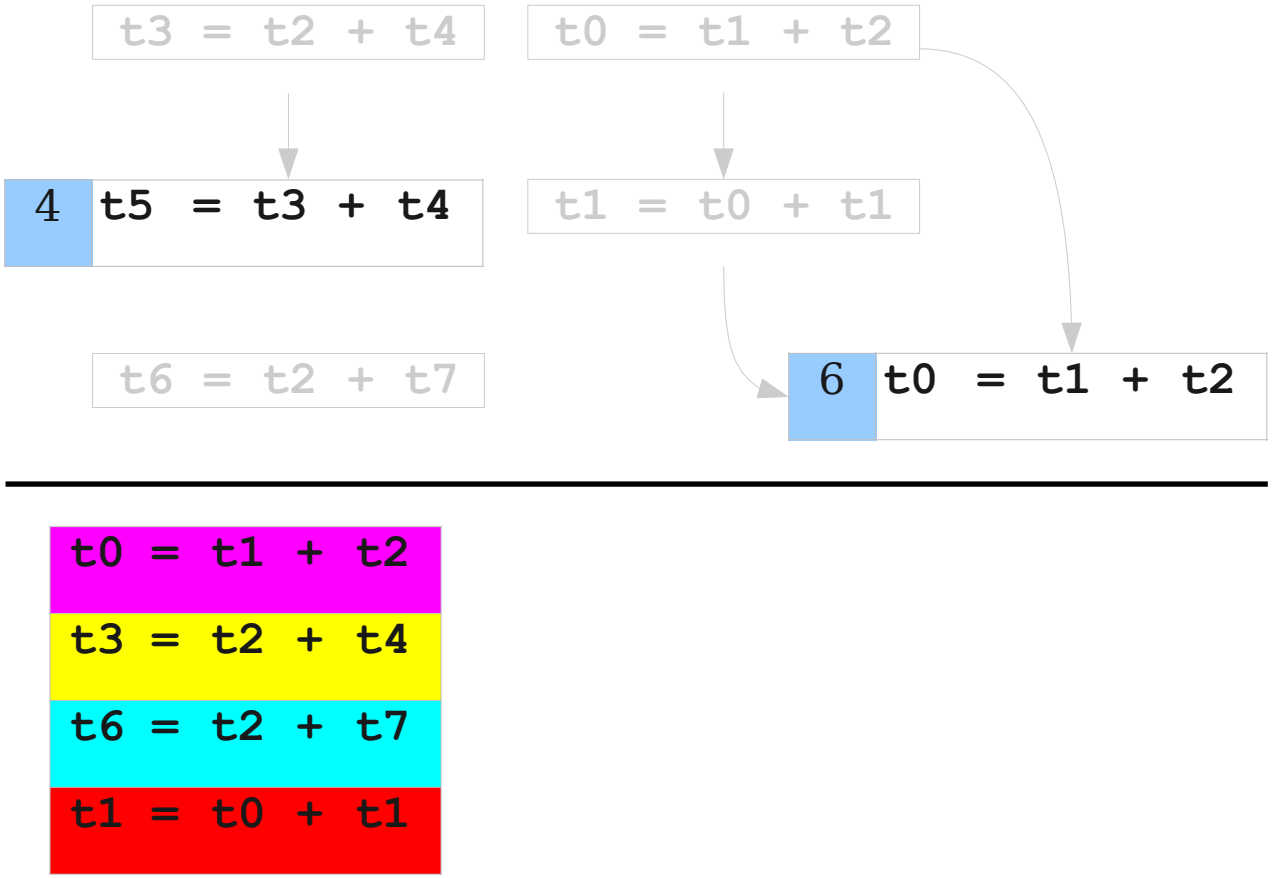
Instruction Scheduling

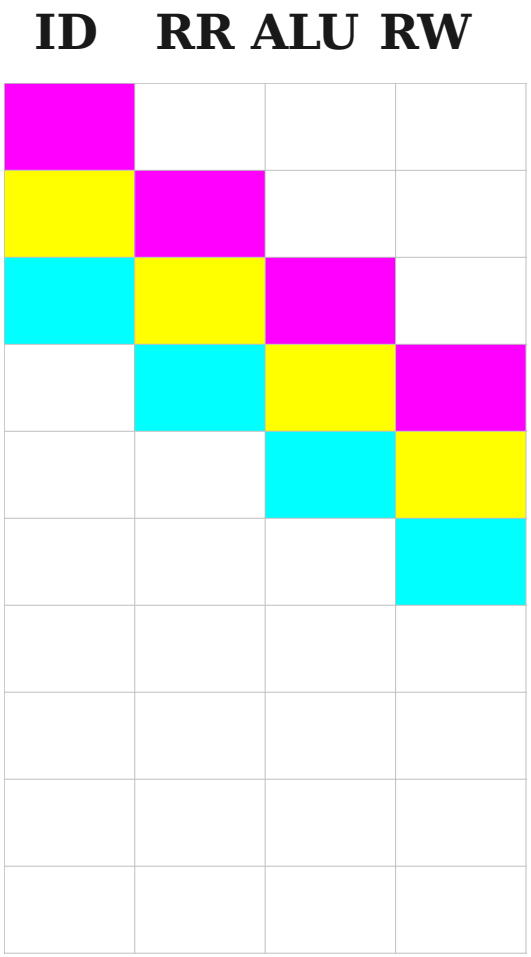




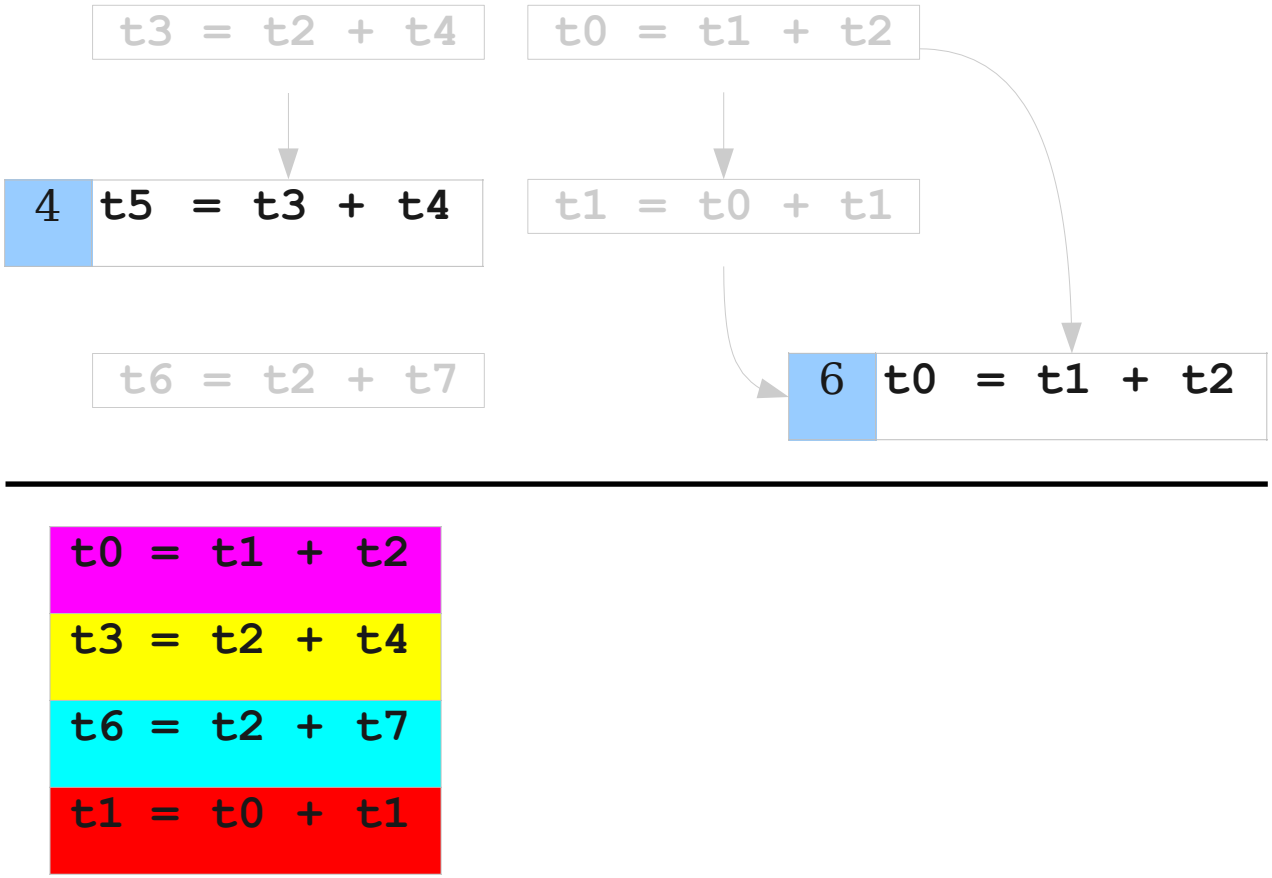


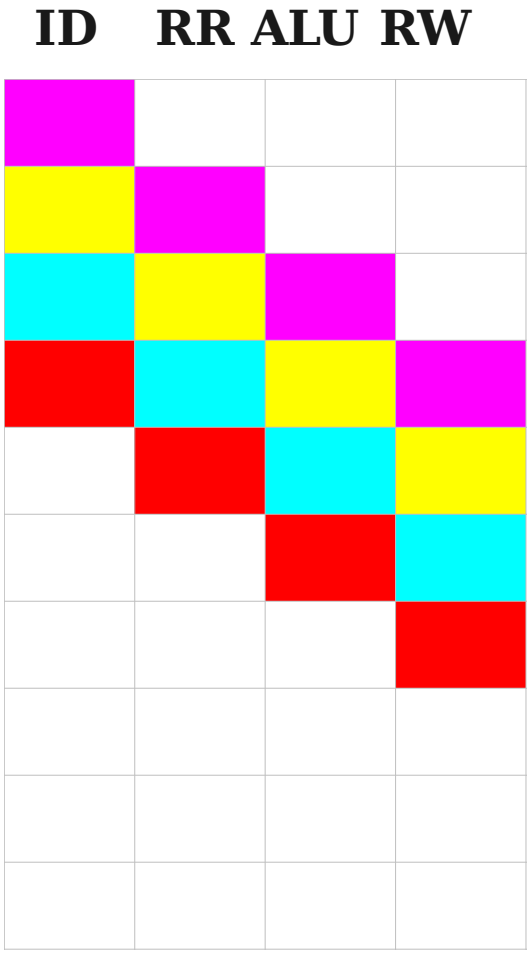
Instruction Scheduling



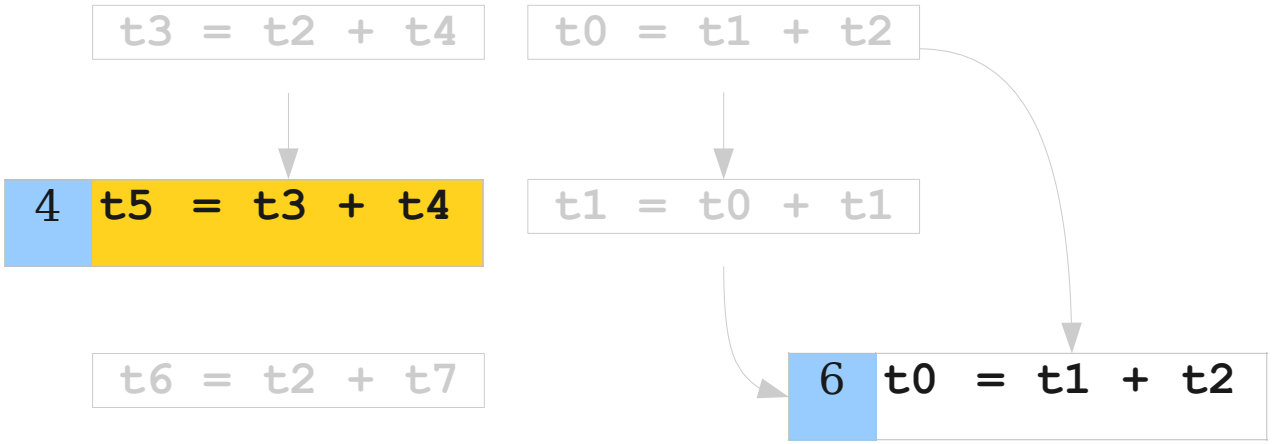


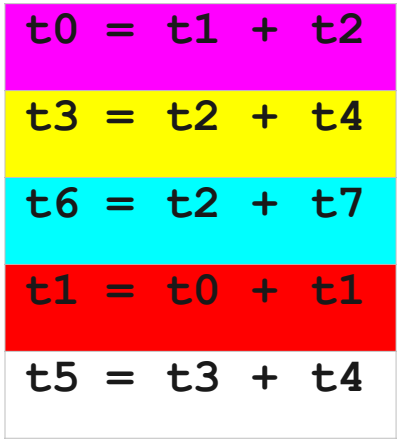
Instruction Scheduling

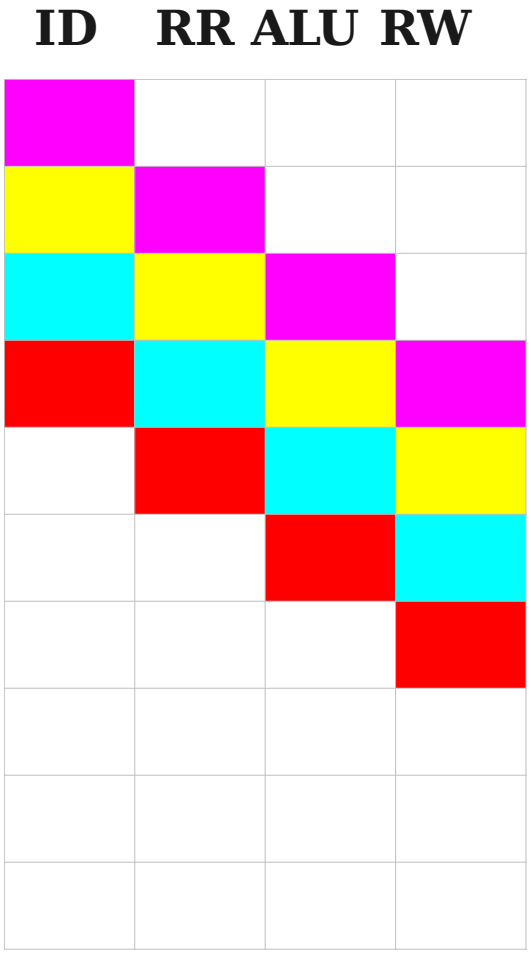




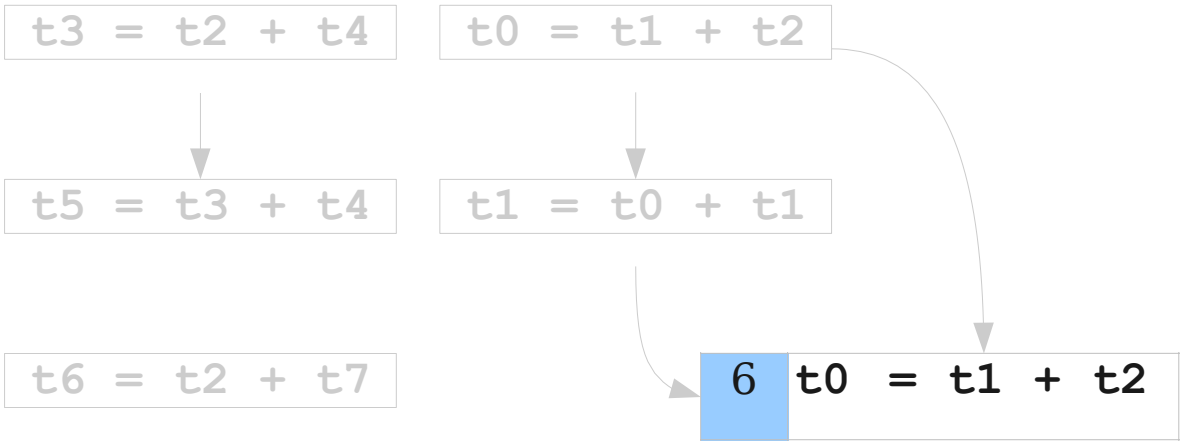
Instruction Scheduling

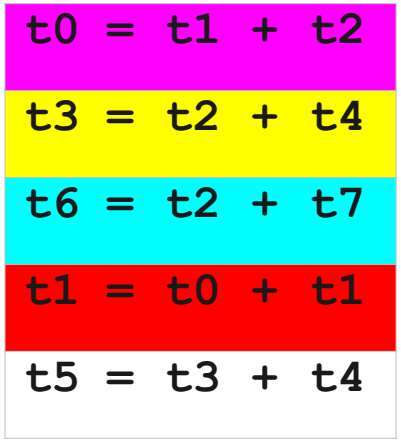


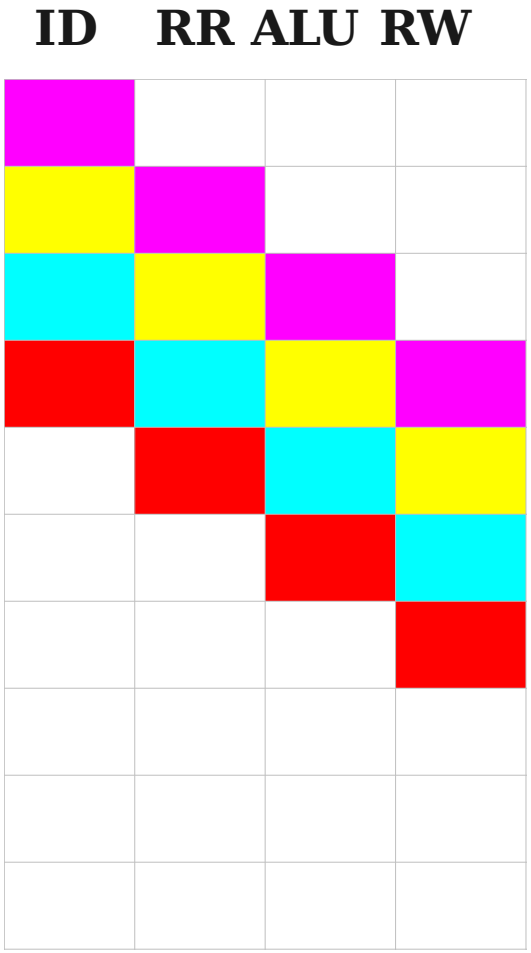




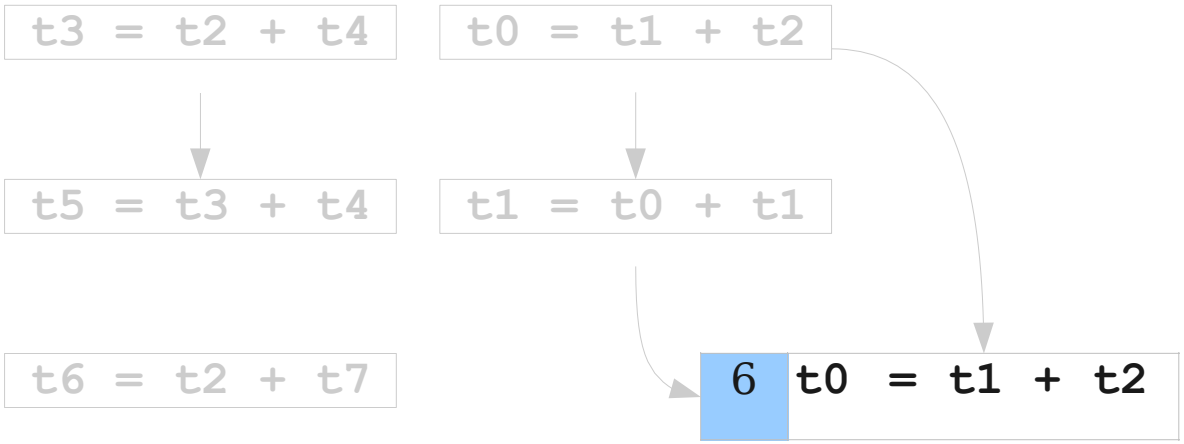
Instruction Scheduling

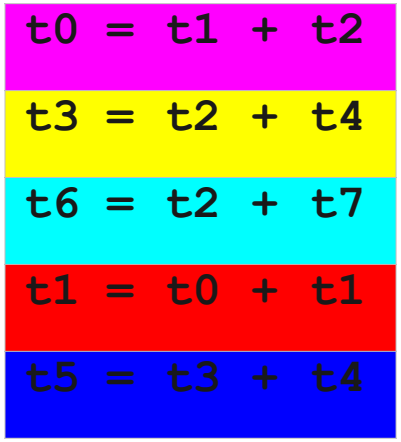


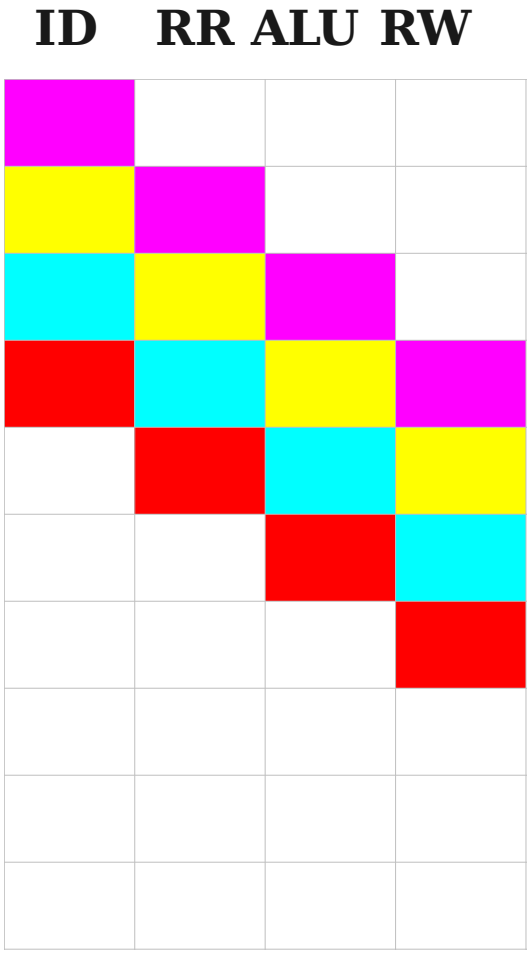




Instruction Scheduling

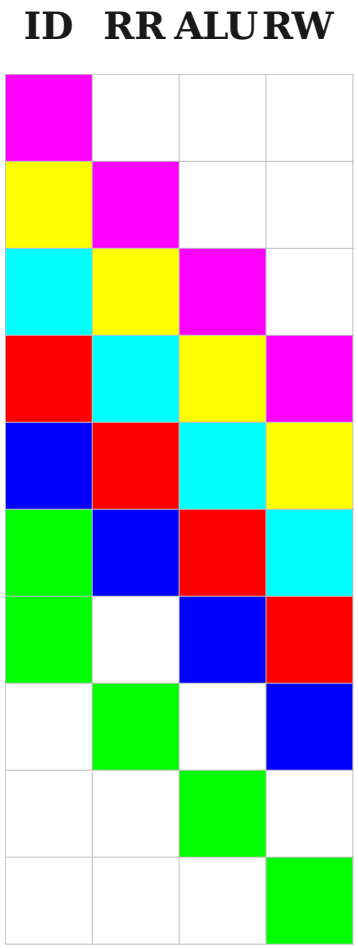




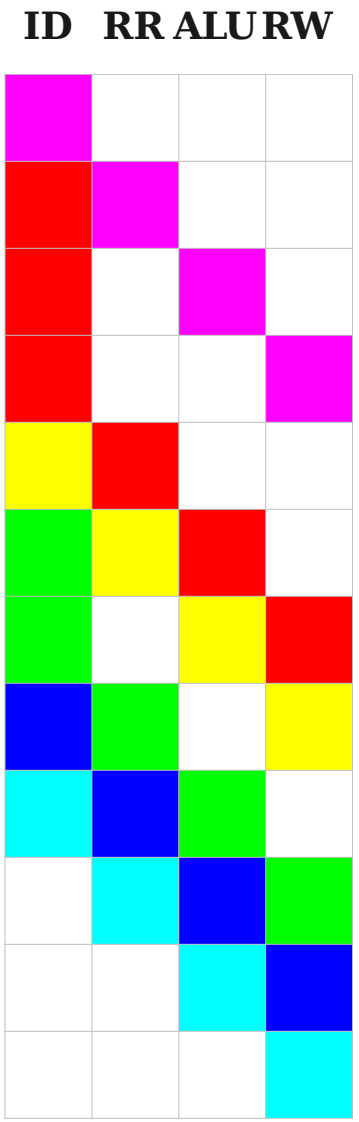


For Comparison

```
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4
t0 = t1 + t2
```



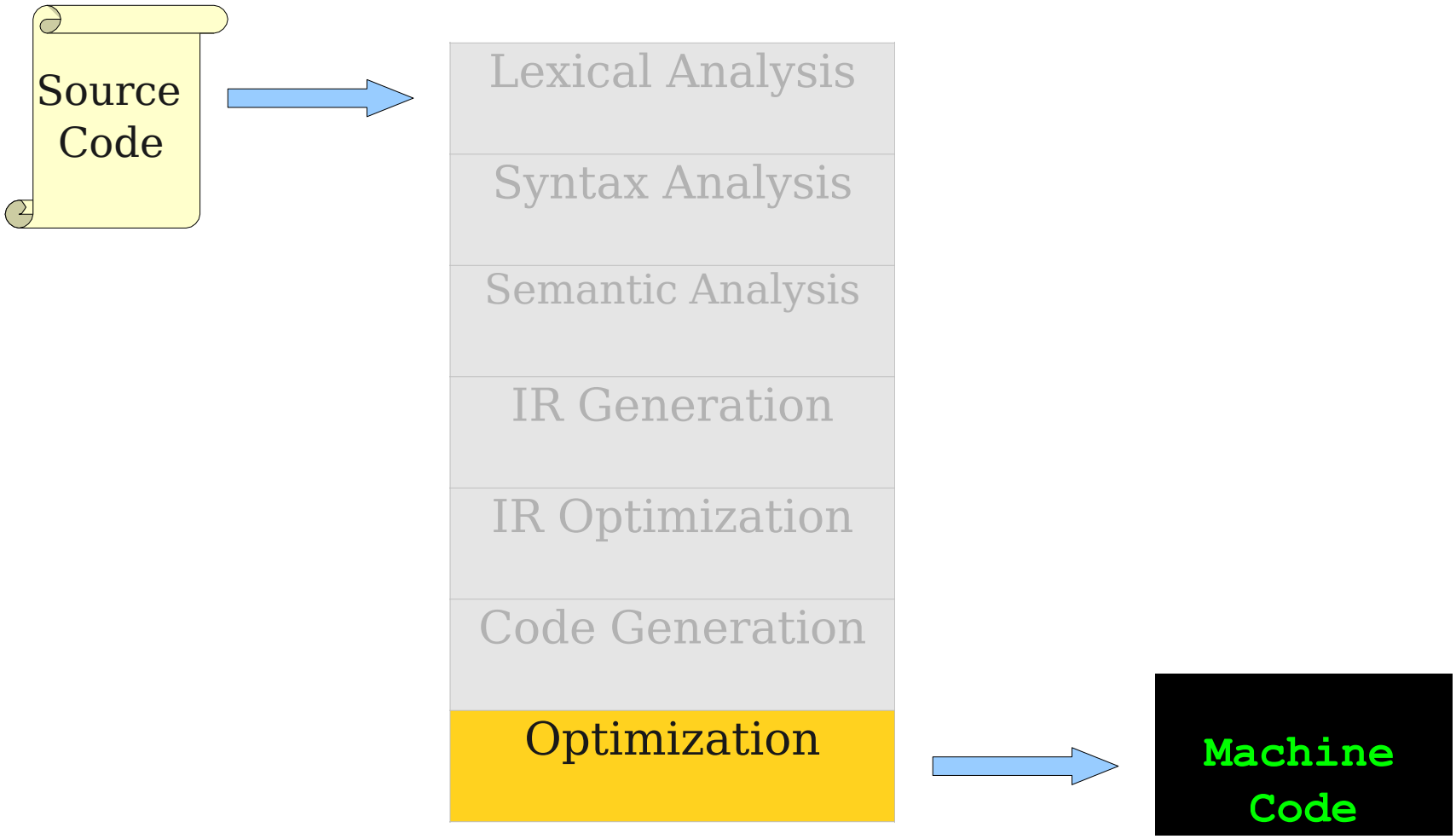

```
t0 = t1 + t2
t1 = t0 + t1
t3 = t2 + t4
t0 = t1 + t2
t5 = t3 + t4
t6 = t2 + t7
```



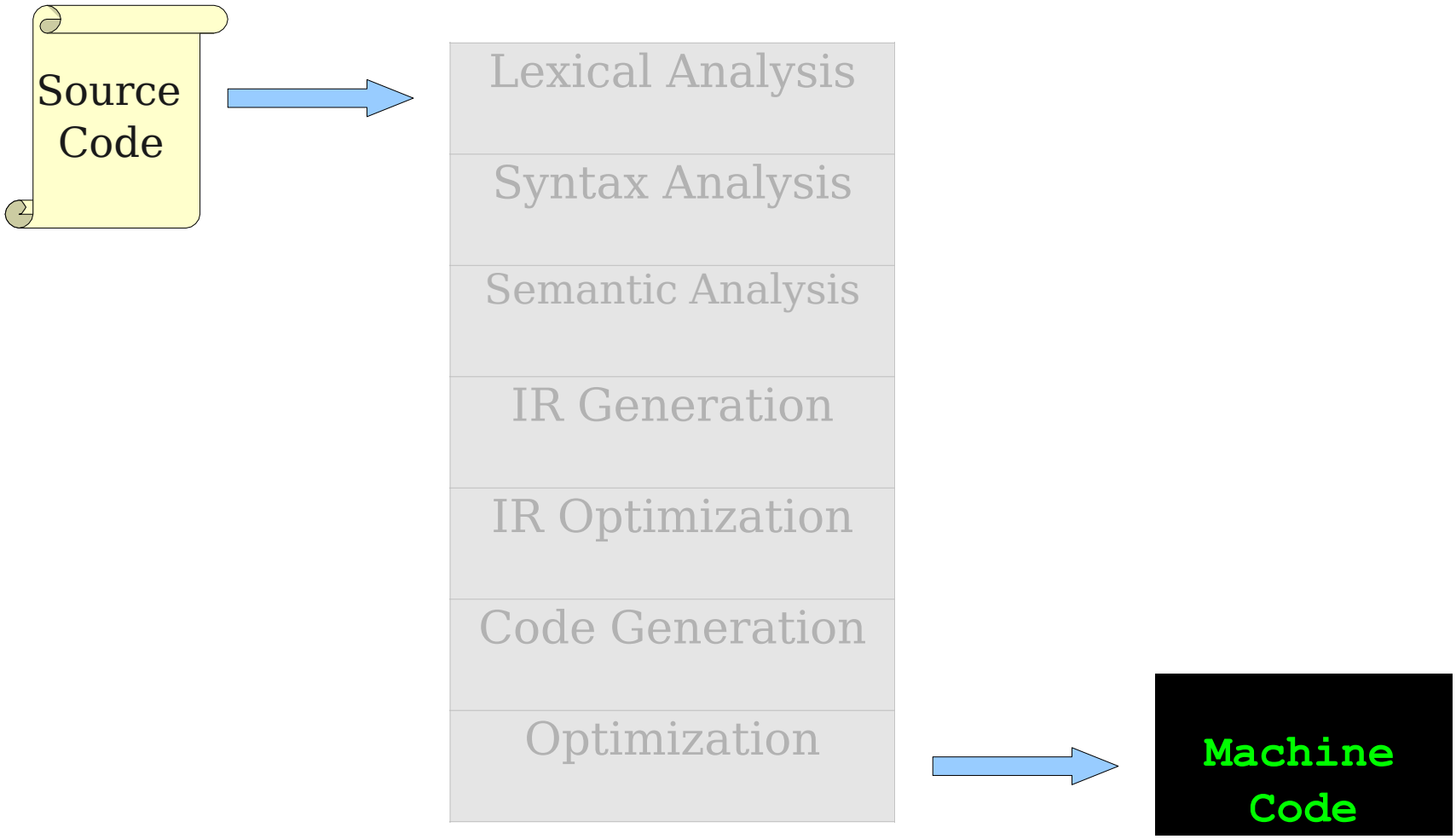
More Advanced Scheduling

- Modern optimizing compilers can do far more aggressive scheduling to obtain impressive performance gains.
- One powerful technique: **Loop unrolling**
  - Expand out several loop iterations at once.
  - Use previous algorithm to schedule instructions more intelligently.
  - Can find pipelining-level parallelism across loop iterations.
- Even more powerful technique: **Software pipelining**
  - Loop unrolling on steroids; can convert loops using tens of cycles into loops averaging two or three cycles.

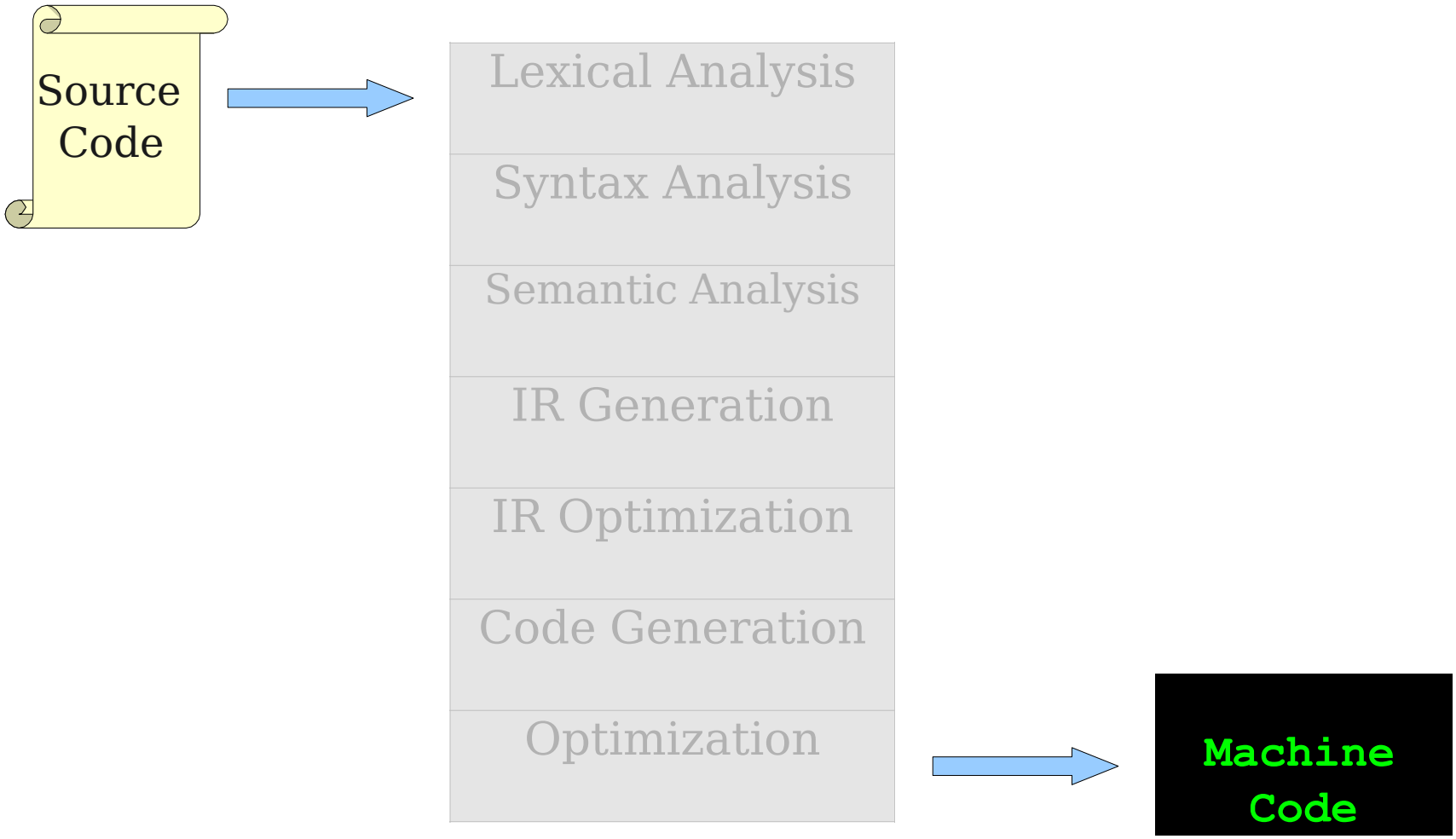
Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where We've Been



Where We've Been

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

- Regular Expressions
- Finite Automata
- Maximal-Munch
- Subset Construction
- **`flex`**
-

Where We've Been

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

IR Optimization

Code Generation

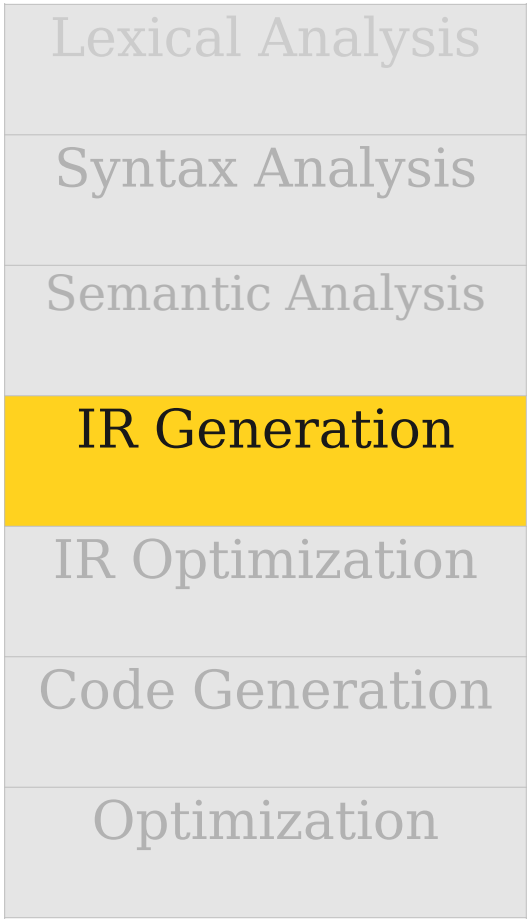
Optimization

- Syntax Graph
- Context-Free Grammars
- Parse Trees
- ASTs
- Leftmost DFS
- LL(1)
- Handles
- LR(0)
- SLR(1)
- LR(1)
- LALR(1)
-

Where We've Been

- SDT
- Scope-Checking
- Spaghetti Stacks
- Function Overloading
- Type Systems
- Well-Formedness
- Null and Error Types

Where We've Been

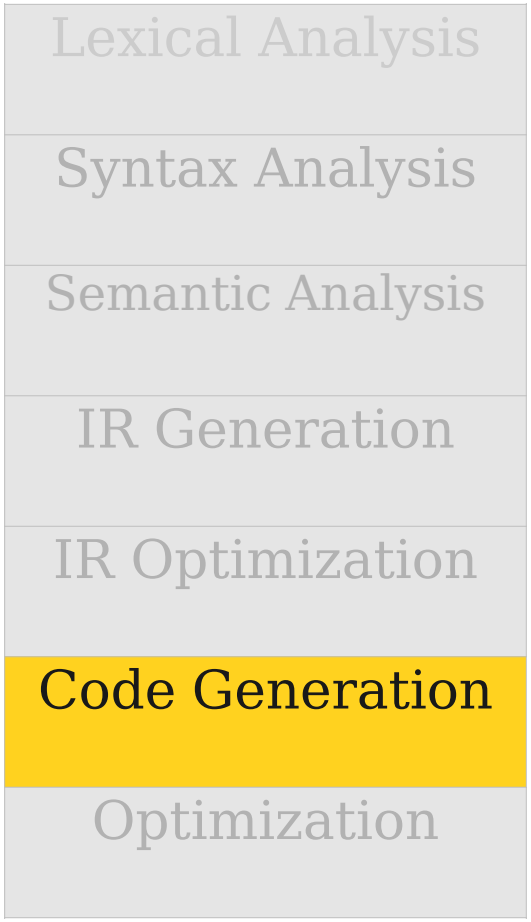


- Runtime Environments
- Function Stacks
- Closures
- Coroutines
- Parameter Passing
- Object Layouts
- Vtables
- Inline Caching
- TAC

Where We've Been

- Basic Blocks
- Control-Flow Graphs
- Common Subexpression Elimination
- Copy Propagation Dead
- Code Elimination
- Arithmetic Simplification
- Constant Folding
- Meet Semilattices
- Transfer Functions
- Global Constant Propagation
- Partial Redundancy Elimination
-

Where We've Been



- Memory Hierarchies
- Live Ranges
- Live Intervals
- Linear-Scan Register Allocation
- Register Interference Graphs
- Chaitin's Graph-Coloring Algorithm
- ~~Reference Counting~~
- ~~Mark-and-Sweep Collectors~~
- ~~Baker's Algorithm~~
- ~~Stop-and-Copy Collectors~~
- ~~Generational Collectors~~
-

Where We've Been

- Instruction Scheduling
- ~~Loop Reordering~~
- ~~Structure Peeling~~
- ~~Automatic Parallelization~~
-

Why Study Compilers? (Recap)

- Build a **large, ambitious software system**.
- See theory **come to life**.
- Learn how to **build programming languages**.
- Learn **how programming languages work**.
- Learn **tradeoffs in language design**.

Where to Go from Here