Intermediate Representations

Akim Demaille Étienne Renault Roland Levillain
first.last@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

April 29, 2019

Intermediate Representations

Intermediate Representations

Compilers Structure

So many ends...

Ends:

- front end: analysis
- middle end: generic synthesis
- back end: specific synthesis

The gcc team suggests

- front end: name ("a front end").
- front-end: adjective ("the front-end interface").

Front Ends...

The front end is dedicated to analysis:

- lexical analysis (scanning)
- syntactic analysis (parsing)
- ast generation
- static semantic analysis (type checking, context sensitive checks)
- source language specific optimizations
- hir generation

... Back Ends

The back end is dedicated to specific synthesis:

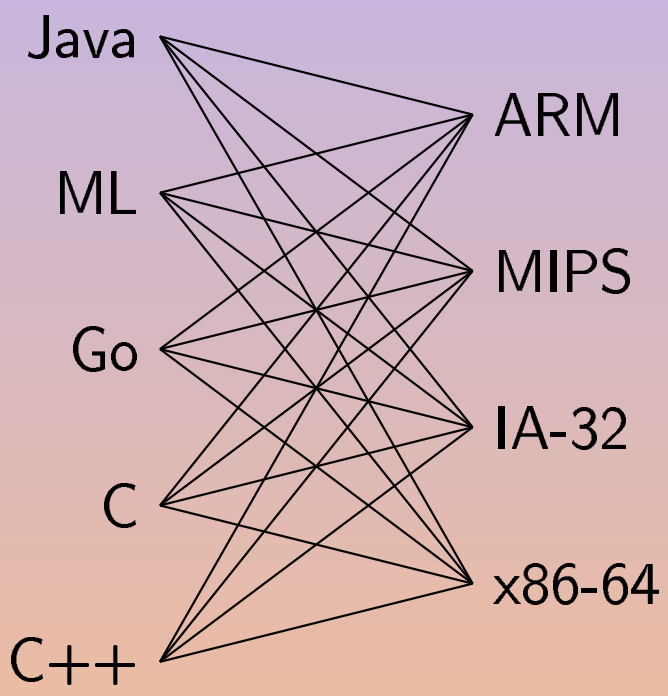
- instruction selection (mir to lir)
- register allocation
- assembly specific optimizations
- assembly code emission

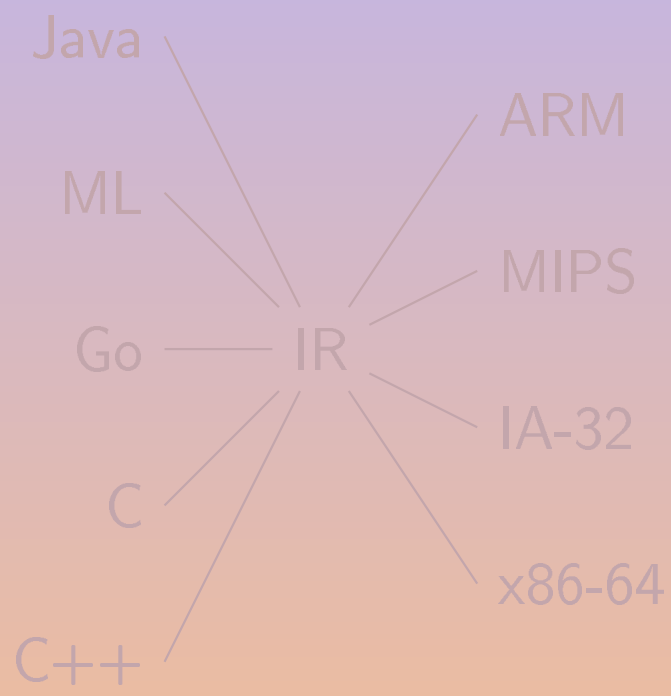
... Middle Ends...

The middle end is dedicated to generic synthesis:

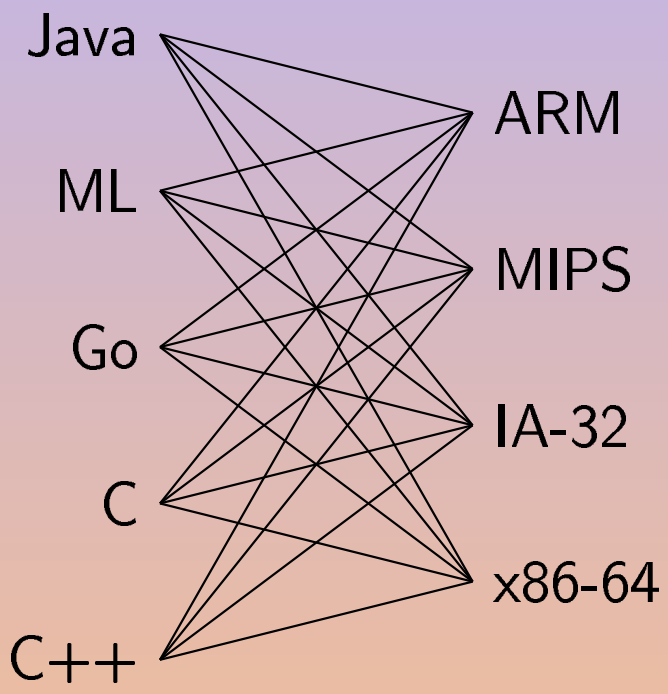
- stepwise refinement of hir to mir
- generic optimizations

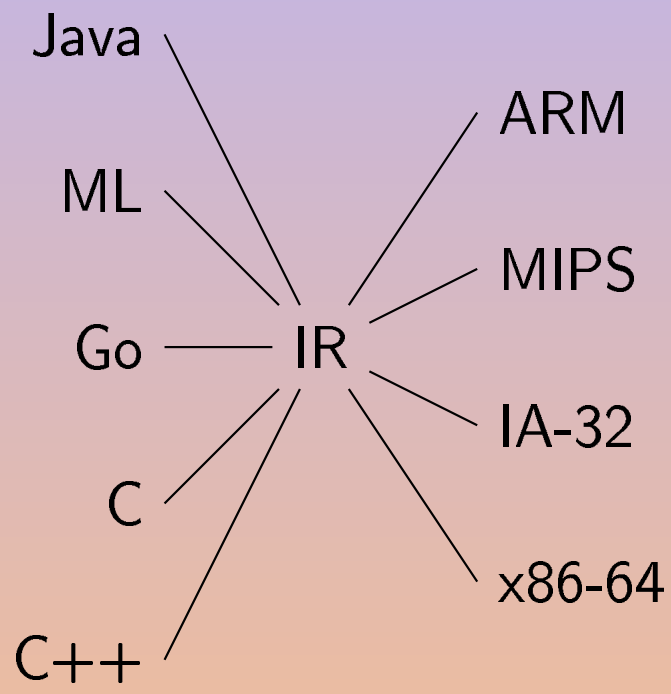
Retargetable Compilers





Retargetable Compilers





Other Compiling Strategies

- Intermediate language-based strategy: SmartEiffel, GHC
- Bytecode strategy: Java bytecode (JVM), CIL (.NET)
- Hybrid approaches: GCJ (Java bytecode or native code)
- Retargetable optimizing back ends: MLRISC, VPO (Very Portable Optimizer), and somehow C-- (Quick C--).
- Modular systems: LLVM (compiler as a library, centered on a typed IR). Contains the LLVM core libraries, Clang, LLDB, etc. Also:
  - VMKit: a substrate for virtual machines (JVM, etc.).
  - Emscripten: an LLVM-to-JavaScript compiler. Enables C/C++ to JS compilation.

Intermediate Representations (IR) are fundamental.

Intermediate Representations

Format? Representation? Language?

Intermediate representation:

- a faithful model of the source program
- "written" in an abstract language, the *intermediate language*
- may have an external syntax
- may be interpreted/compiled
  (havm, byte code)
- may have different levels
  (gcc's Tree is very much like C).

What Language Flavor?

- Imperative?
  - Stack Based? (Java Byte-code)
  - Register Based? (gcc's rtl, `tc`'s `Tree`)
- Functional?
  Most functional languages are compiled into a lower level language, eventually a simple λ-calculus.
- Other?

What Level?

A whole range of expressivities, typically aiming at making some optimizations easier:

- Keep array expressions?

  Yes: adequate for dependency analysis and related optimizations,

  No: Good for constant folding, strength reduction, loop invariant code motion, etc.

- Keep loop constructs?

What level of machine independence?

- Explicit register names?

Designing an Intermediate Representation

> *Intermediate-language design is largely an art, not a science.*
>
> — [Muchnick, 1997]

Different Levels [Muchnick, 1997]

```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

```
r1 <- [fp - 4]
r2 <- r1 + 2
r3 <- [fp - 8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp - 216
f1 <- [r7 + r6]
```

Different Levels [Muchnick, 1997]

```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

```
r1 <- [fp - 4]
r2 <- r1 + 2
r3 <- [fp - 8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp - 216
f1 <- [r7 + r6]
```

Different Levels [Muchnick, 1997]

```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

```
r1 <- [fp - 4]
r2 <- r1 + 2
r3 <- [fp - 8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp - 216
f1 <- [r7 + r6]
```

Different Levels [Muchnick, 1997]

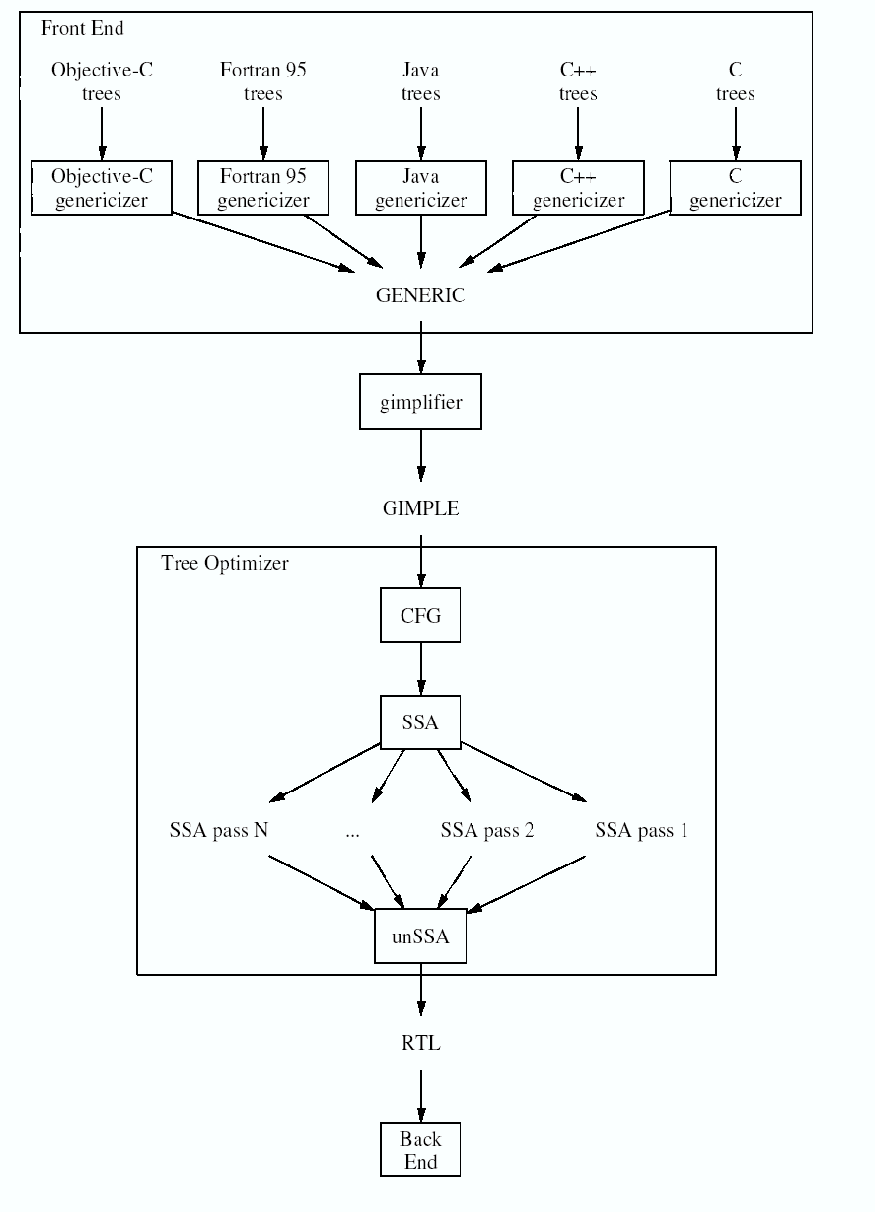
```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

```
r1 <- [fp - 4]
r2 <- r1 + 2
r3 <- [fp - 8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp - 216
f1 <- [r7 + r6]
```

Different Levels: The GCC Structure



Stack Based: Java Byte-Code [Edwards, 2003]

```
class Gcd
{
  static public int gcd(int a, int b)
  {
    while (a != b)
      {
        if (a > b)
          a -= b;
        else
          b -= a;
      }
    return a;
  }

  static public int main(String[] arg)
  {
    return gcd(12, 34);
  }
}
```

Stack Based: Java Byte-Code

```
% gcj-3.3 -c gcd.java
% jcf-dump-3.3 -c gcd
...
Method name:"gcd" public static
Signature: 5=(int,int)int
Attribute "Code", length:66,
max_stack:2, max_locals:2,
code_length:26
   0: iload_0
   1: iload_1
   2: if_icmpeq 24
   5: iload_0
   6: iload_1
   7: if_icmple 17
  10: iload_0
  11: iload_1
  12: isub
  13: istore_0
  17: iload_1
  18: iload_0
  19: isub
  20: istore_1
  21: goto 0
  24: iload_0
  25: ireturn
Attribute "LineNumberTable",
          length:22, count: 5
   line: 5 at pc: 0
   line: 7 at pc: 5
   line: 8 at pc: 10
   line: 10 at pc: 17
   line: 12 at pc: 24
...
```

Stack Based [Edwards, 2003]

Advantages

- Trivial translation of expressions
- Trivial interpreters
- No pressure on registers
- Often compact

Disadvantages

- Does not fit with today's architectures
- Hard to analyze
- Hard to optimize

Stack Based [Edwards, 2003]

Advantages

- Trivial translation of expressions
- Trivial interpreters
- No pressure on registers
- Often compact

Disadvantages

- Does not fit with today's architectures
- Hard to analyze
- Hard to optimize

Stack Based: Examples

ucode, used in hp pa-risk, and mips, was designed for stack evaluation (HP 3000 is stack based).
Today it is less adequate.
mips translates it back and forth to triples for optimization.
hp converts it into sllic (Spectrum Low Level ir) [Muchnick, 1997].

Register Based: tc's Tree

```
let function gcd(a: int, b: int) : int =
    (
      while a <> b
        do if a > b then a := a - b
                    else b := b - a;
      a
    )
in
  print_int(gcd(42, 51))
end
```

Register Based: tc's Tree (1/4)

```
/* == High Level Intermediate representation. == */
# Routine: gcd
label l0
# Prologue
move temp t0 temp fp
move temp fp temp sp
move
  temp sp
  binop sub temp sp const 12
move
mem temp fp
    temp i0
move
  mem binop add temp fp const -4
  temp i1
move
  mem binop add temp fp  const -8
  temp i2
```

Register Based: tc's Tree (2/4)

```
# Body
move temp rv
  eseq
  seq
    label l2
    cjump ne mem binop add temp fp const −4
           mem binop add temp fp const −8
           name l3 name l1
    label l3
    seq
      cjump gt mem binop add temp fp const −4
             mem binop add temp fp const −8
             name l4 name l5
      label l4
      move mem binop add temp fp const −4
          binop sub mem binop add temp fp const −4
                 mem binop add temp fp const −8
      jump name l6
```

Register Based: tc's Tree (3/4)

```
      label l5
      move mem binop add temp fp const -8
           binop sub mem binop add temp fp const -8
                     mem binop add temp fp const -4
      label l6
    seq end
    jump name l2
    label l1
  seq end
  mem  binop add temp fp const -4

# Epilogue
move temp sp temp fp
move temp fp temp t0
label end
```

Register Based: tc's Tree (4/4)

```
# Routine: _main
label main
# Prologue
# Body
seq
  sxp
    call
      name print_int
      call name 10 temp fp const 42 const 51
      call end
    call end
  sxp
    const 0
seq end
# Epilogue
label end
```

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).

- 2 address instructions? (*triples*)
- 3 address instructions? (*quadruples*)

Tree Expressions and instructions are unnamed, related to each other as nodes of trees

dag Compact, good for local value numbering, but that's all.

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).

- 2 address instructions? (*triples*)
- 3 address instructions? (*quadruples*)

Tree Expressions and instructions are unnamed, related to each other as nodes of trees

dag Compact, good for local value numbering, but that's all.

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).

- 2 address instructions? (*triples*)
- 3 address instructions? (*quadruples*)

Tree Expressions and instructions are unnamed, related to each other as nodes of trees

dag Compact, good for local value numbering, but that's all.

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).

- 2 address instructions? (*triples*)
- 3 address instructions? (*quadruples*)

Tree Expressions and instructions are unnamed, related to each other as nodes of trees

dag Compact, good for local value numbering, but that's all.

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).

- 2 address instructions? (*triples*)
- 3 address instructions? (*quadruples*)

Tree Expressions and instructions are unnamed, related to each other as nodes of trees

dag Compact, good for local value numbering, but that's all.

Quadruples vs. Triples [Muchnick, 1997]

```
L1:  i  <- i + 1

     t1 <- i + 1
     t2 <- p + 4
     t3 <- *t2
     p  <- t2
     t4 <- t1 < 10
     *r <- t3
     if t4 goto L1
```

```
(1)  i + 1
(2)  i sto (1)
(3)  i + 1
(4)  p + 4
(5)  *(4)
(6)  p sto (4)
(7)  (3) < 10
(8)  *r sto (5)
(9)  if (7), (1)
```

Quadruples vs. Triples [Muchnick, 1997]

```
L1:  i  <- i + 1

     t1 <- i + 1
     t2 <- p + 4
     t3 <- *t2
     p  <- t2
     t4 <- t1 < 10
     *r <- t3
     if t4 goto L1
```

```
(1)  i + 1
(2)  i sto (1)
(3)  i + 1
(4)  p + 4
(5)  *(4)
(6)  p sto (4)
(7)  (3) < 10
(8)  *r sto (5)
(9)  if (7), (1)
```

Register Based: gcc's rtl

```
int
gcd(int a, int b)
{
  while (a != b)
    {
      if (a > b)
        a -= b;
      else
        b -= a;
    }
  return a;
}
```

Register Based: gcc's rtl

```
;; Function gcd
(note 1 0 2 ("gcd.c") 3)
(note 2 1 3 NOTE_INSN_DELETED)
(note 3 2 4 NOTE_INSN_FUNCTION_BEG)
(note 4 3 5 NOTE_INSN_DELETED)
(note 5 4 6 NOTE_INSN_DELETED)
(note 6 5 7 NOTE_INSN_DELETED)
(insn 7 6 8 (const_int 0 [0x0]) -1 (nil)
    (nil))
```

Register Based: gcc's rtl cont'd

```
(note 8 7 9 ("gcd.c") 4)
(note 9 8 40 NOTE_INSN_LOOP_BEG)
(note 40 9 10 NOTE_INSN_LOOP_CONT)
(code_label 10 40 13 2 "" "" [0 uses])
(insn 13 10 14 (set (reg:SI 59)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
    -1 (nil) (nil))
(insn 14 13 15 (set (reg:CCZ 17 flags)
        (compare:CCZ (reg:SI 59)
            (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32]))) -1 (nil)
    (nil))
```

Register Based: gcc's rtl cont'd

```
(jump_insn 15 14 16 (set (pc)
        (if_then_else (ne (reg:CCZ 17 flags)
                (const_int 0 [0x0]))
            (label_ref 18)
            (pc))) -1 (nil)
    (nil))
(jump_insn 16 15 17 (set (pc)
        (label_ref 44)) -1 (nil)
    (nil))
(barrier 17 16 18)
(code_label 18 17 19 4 "" "" [0 uses])
(note 19 18 20 NOTE_INSN_LOOP_END_TOP_COND)
(note 20 19 21 NOTE_INSN_DELETED)
(note 21 20 22 NOTE_INSN_DELETED)
```

Register Based: gcc's rtl cont'd

```
(note 22 21 25 ("gcd.c") 6)
(insn 25 22 26 (set (reg:SI 60)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
     -1 (nil) (nil))
(insn 26 25 27 (set (reg:CCGC 17 flags)
        (compare:CCGC (reg:SI 60)
            (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32]))) -1 (nil)
    (nil))
(jump_insn 27 26 28 (set (pc)
        (if_then_else (le (reg:CCGC 17 flags)
                (const_int 0 [0x0]))
            (label_ref 34)
            (pc))) -1 (nil)
    (nil))
```

Register Based: gcc's rtl cont'd

```
(note 28 27 30 ("gcd.c") 7)
(insn 30 28 31 (set (reg:SI 61)
        (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                (const_int 4 [0x4])) [0 b+0 S4 A32])) -1 (nil)
    (nil))
(insn 31 30 32 (parallel[
            (set (mem/f:SI (reg/f:SI 53 virtual-incoming-args)
                [0 a+0 S4 A32])
                (minus:SI (mem/f:SI (reg/f:SI 53 virtual-incoming-args)
                [0 a+0 S4 A32])
                    (reg:SI 61)))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (expr_list:REG_EQUAL (minus:SI (mem/f:SI (reg/f:SI 53
                                virtual-incoming-args) [0 a+0 S4 A32])
            (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32]))
        (nil)))
(jump_insn 32 31 33 (set (pc)
        (label_ref 39)) -1 (nil)
    (nil))
(barrier 33 32 34)
(code_label 34 33 35 5 "" "" [0 uses])
```

Register Based: gcc's rtl cont'd

```
(note 35 34 37 ("gcd.c") 9)
(insn 37 35 38 (set (reg:SI 62)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
     -1 (nil) (nil))
(insn 38 37 39 (parallel[
            (set (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                        (const_int 4 [0x4])) [0 b+0 S4 A32])
                (minus:SI (mem/f:SI (plus:SI (reg/f:SI 53
                                              virtual-incoming-args)
                            (const_int 4 [0x4])) [0 b+0 S4 A32])
                    (reg:SI 62)))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (expr_list:REG_EQUAL (minus:SI (mem/f:SI (plus:SI (reg/f:SI
                                            53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32])
            (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
        (nil)))
(code_label 39 38 41 6 "" "" [0 uses])
(jump_insn 41 39 42 (set (pc)
        (label_ref 10)) -1 (nil)
    (nil))
(barrier 42 41 43)
(note 43 42 44 NOTE_INSN_LOOP_END)
```

Register Based: gcc's rtl cont'd

```
(note 45 44 46 ("gcd.c") 11)
(note 46 45 47 NOTE_INSN_DELETED)
(note 47 46 49 NOTE_INSN_DELETED)
(insn 49 47 51 (set (reg:SI 64)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32])) -1 (nil)
    (nil))
(insn 51 49 52 (set (reg:SI 58)
        (reg:SI 64)) -1 (nil)
    (nil))
(jump_insn 52 51 53 (set (pc)
        (label_ref 56)) -1 (nil)
    (nil))
(barrier 53 52 54)
(note 54 53 55 NOTE_INSN_FUNCTION_END)
(note 55 54 59 ("gcd.c") 12)
(insn 59 55 60 (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))
(insn 60 59 56 (clobber (reg:SI 58)) -1 (nil)
    (nil))
(code_label 56 60 58 1 "" "" [0 uses])
(insn 58 56 61 (set (reg/i:SI 0 eax)
        (reg:SI 58)) -1 (nil)
    (nil))
(insn 61 58 0 (use (reg/i:SI 0 eax)) -1 (nil)
```

Register Based [Edwards, 2003]

Advantages

- Suits today's architectures
- Clearer data flow

Disadvantages

- Harder to synthesize
- Less compact
- Harder to interpret

Register Based [Edwards, 2003]

Advantages

- Suits today's architectures
- Clearer data flow

Disadvantages

- Harder to synthesize
- Less compact
- Harder to interpret

Tree

Tree [Appel, 1998]

A simple intermediate language:

- Tree structure (no kidding...)
- Unbounded number of registers (`temporaries`)
- Two way conditional jump

Tree: Grammar

```
⟨Exp⟩ ::= "const" int
        | "name" ⟨Label⟩
        | "temp" ⟨Temp⟩
        | "binop" ⟨Oper⟩ ⟨Exp⟩ ⟨Exp⟩
        | "mem" ⟨Exp⟩
        | "call" ⟨Exp⟩ [{⟨Exp⟩}] "call end"
        | "eseq" ⟨Stm⟩ ⟨Exp⟩

⟨Stm⟩ ::= "move" ⟨Exp⟩ ⟨Exp⟩
        | "sxp" ⟨Exp⟩
        | "jump" ⟨Exp⟩ [{⟨Label⟩}]
        | "cjump" ⟨Relop⟩ ⟨Exp⟩ ⟨Exp⟩ ⟨Label⟩ ⟨Label⟩
        | "seq" [{⟨Stm⟩}] "seq end"
        | "label" ⟨Label⟩

⟨Oper⟩  ::= "add" | "sub" | "mul" | "div" | "mod"
⟨Relop⟩ ::= "eq" | "ne" | "lt" | "gt" | "le" | "ge"
```

Tree Samples

```
% echo '1 + 2 * 3' | tc -H -
/* == High Level Intermediate representation. == */
# Routine: Main Program
label Main
# Prologue
# Body
sxp
    binop add
        const 1
        binop mul
            const 2
            const 3
# Epilogue
label end
```

Tree Samples

```
% echo 'if 1 then print_int (1)' | tc -H -
# Routine: Main Program
label Main
# Prologue
# Body
seq
    cjump ne, const 1, const 0, name l1, name l2
    label l1
    sxp call name print_int, const 1
    jump name l3
    label l2
    sxp const 0
    label l3
seq end
# Epilogue
label end
```

Memory Management

Memory Management

Memory Hierarchy [Appel, 1998]

Different kinds of memory in a computer, with different performances:

- **Registers** Small memory units built on the cpu (bytes, 1 cycle)
- **L1 Cache** Last main memory access results (kB, 2-3 cycles)
- **L2 Cache** (MB, 10 cycles)
- **Memory** The usual ram (GB, 100 cycles)
- **Storage** Disks (100GB, TB, $>$ 1Mcycles)

Use the registers as much as possible.

Register Overflow

What if there are not enough registers? Use the main memory, but how?
Recursion:

Without Each name is bound once. It can be statically allocated a single unit of main memory. (Cobol, Concurrent Pascal, Fortran (unless `recursive`)).

With A single name can be part of several concurrent bindings. Memory allocation must be dynamic.

Register Overflow

What if there are not enough registers? Use the main memory, but how?
Recursion:

Without Each name is bound once. It can be statically allocated a single unit of main memory. (Cobol, Concurrent Pascal, Fortran (unless `recursive`)).

With A single name can be part of several concurrent bindings. Memory allocation must be dynamic.

Dynamic Memory Allocation

Depending on the persistence, several models:

Global Global objects, whose liveness is equal to that of the program, are statically allocated
(e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function
(e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

User Controlled
: `malloc/free` (C), `new/dispose` (Pascal), `new/delete` (C++) etc.

Garbage Collected
: With or without `new`
(`lisp`, Smalltalk, `ML`, Haskell, Tiger, Perl etc.).

Dynamic Memory Allocation

Depending on the persistence, several models:

Global Global objects, whose liveness is equal to that of the program, are statically allocated
(e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function
(e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

- User Controlled
  - `malloc/free` (C), `new/dispose` (Pascal), `new/delete` (C++) etc.
- Garbage Collected
  - With or without `new`
  - (`lisp`, Smalltalk, `ML`, Haskell, Tiger, Perl etc.).

Dynamic Memory Allocation

Depending on the persistence, several models:

Global Global objects, whose liveness is equal to that of the program, are statically allocated
(e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function
(e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

- User Controlled
  - `malloc`/`free` (C), `new`/`dispose` (Pascal), `new`/`delete` (C++) etc.
- Garbage Collected
  - With or without `new`
  - (`lisp`, Smalltalk, `ML`, Haskell, Tiger, Perl etc.).

Dynamic Memory Allocation

Depending on the persistence, several models:

Global Global objects, whose liveness is equal to that of the program, are statically allocated
(e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function
(e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

- **User Controlled**
  `malloc`/`free` (C), `new`/`dispose` (Pascal), `new`/`delete` (C++) etc.
- **Garbage Collected**
  With or without `new`
  (`lisp`, Smalltalk, `ML`, Haskell, Tiger, Perl etc.).

Dynamic Memory Allocation

Depending on the persistence, several models:

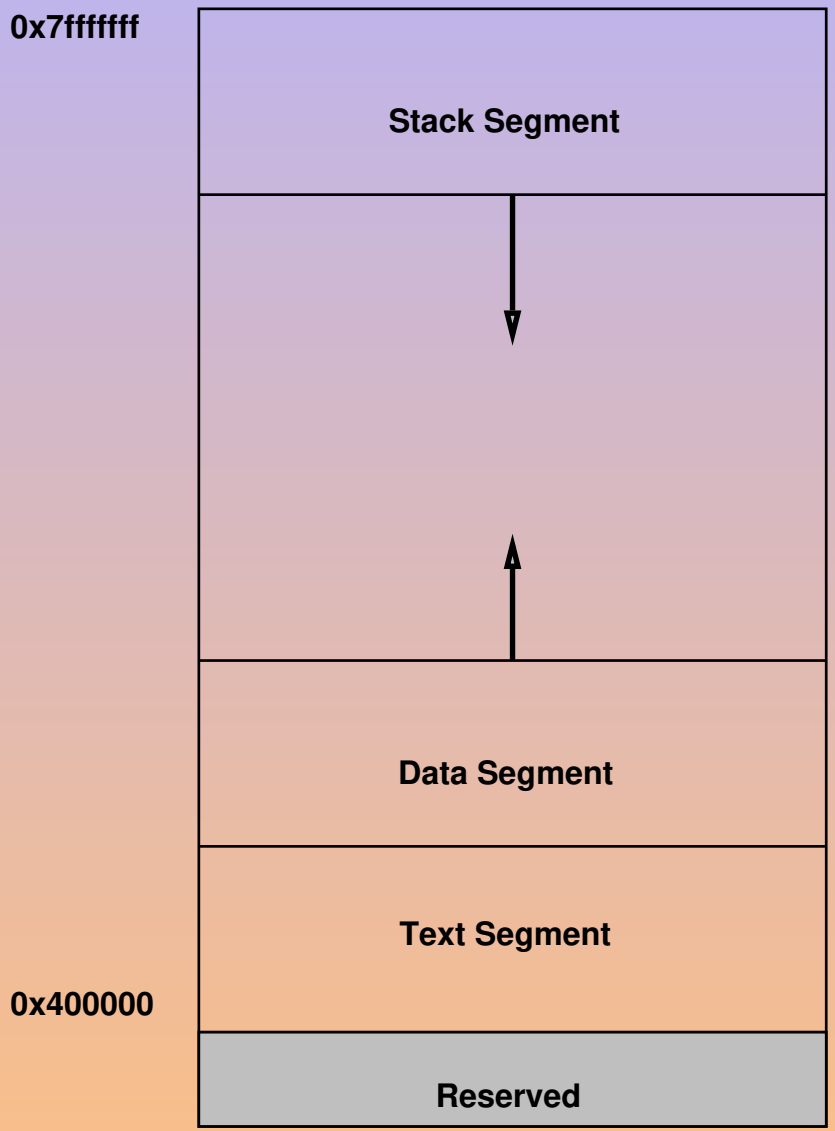
Global Global objects, whose liveness is equal to that of the program, are statically allocated
(e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function
(e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

- **User Controlled**
  `malloc`/`free` (C), `new`/`dispose` (Pascal), `new`/`delete` (C++) etc.
- **Garbage Collected**
  With or without `new`
  (`lisp`, Smalltalk, `ML`, Haskell, Tiger, Perl etc.).

spim Memory Model [Larus, 1990]



Stack Management

Function calls is a last-in first-out process, hence, it is properly represented by a stack.

Or...

"Call tree": the complete history of calls.
The execution of the program is its depth first traversal.
Depth-first walk requires a stack.

Activation Blocks

Activation Blocks

- In recursive languages, a single routine can be "opened" several times concurrently.
- An *activation* designates one single instance of execution.
- Automatic variables are bound to the liveness of the activation.
- Their location is naturally called *activation block*, or *stack frame*.

Activation Blocks

- In recursive languages, a single routine can be "opened" several times concurrently.
- An *activation* designates one single instance of execution.
- Automatic variables are bound to the liveness of the activation.
- Their location is naturally called *activation block*, or *stack frame*.

Activation Blocks

- In recursive languages, a single routine can be "opened" several times concurrently.
- An *activation* designates one single instance of execution.
- Automatic variables are bound to the liveness of the activation.
- Their location is naturally called *activation block*, or *stack frame*.

Activation Blocks

- In recursive languages, a single routine can be "opened" several times concurrently.
- An *activation* designates one single instance of execution.
- Automatic variables are bound to the liveness of the activation.
- Their location is naturally called *activation block*, or *stack frame*.

Activation Blocks Contents

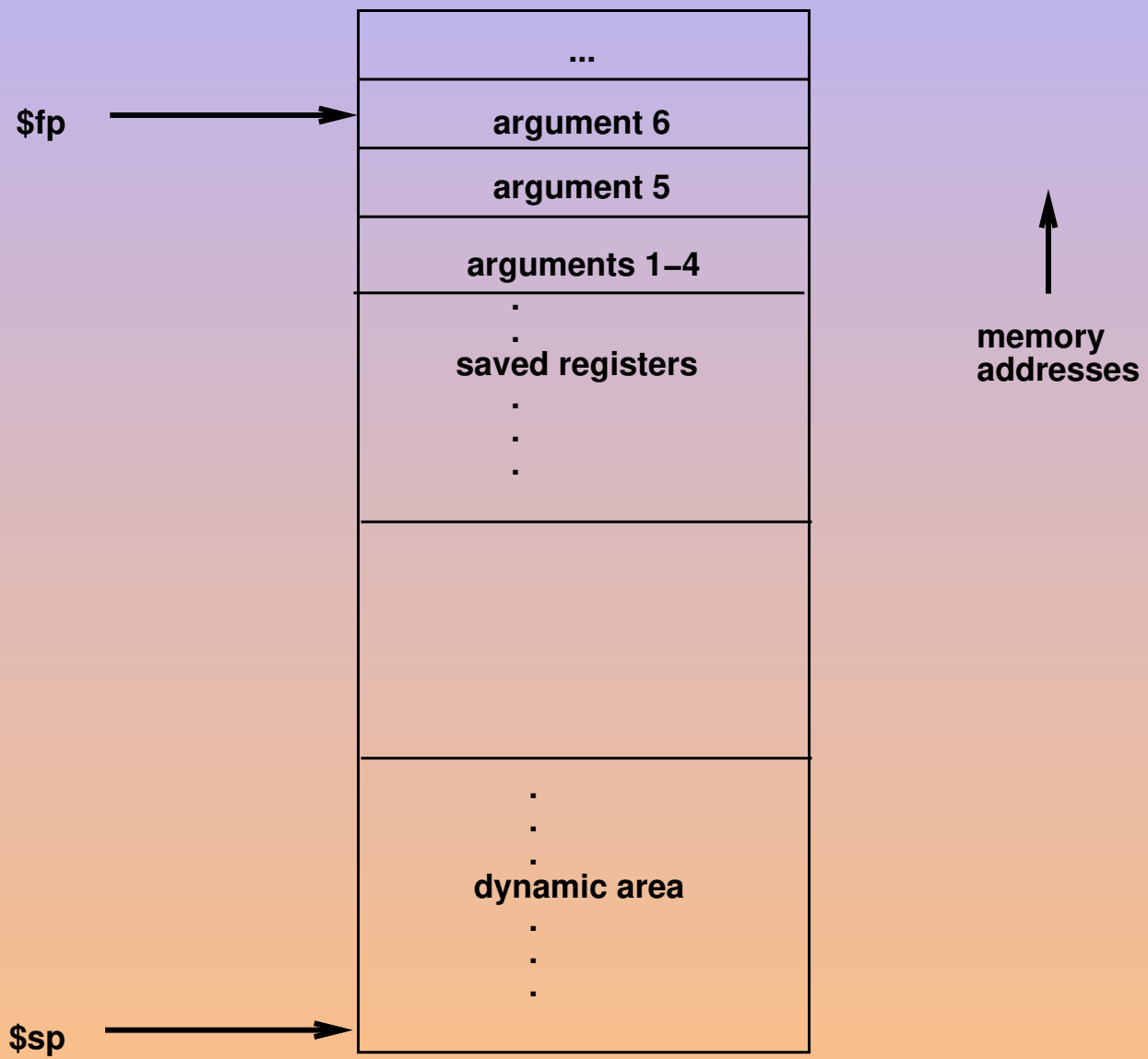
Data to store on the stack:

arguments	incoming
local variables	user automatic variables
return address	where to `return`
saved registers	the caller's environment to restore
temp	compiler automatic variables, spills
static link	when needed

Activation Blocks Layout

The layout is suggested by the constructor.
Usually the layout is from earliest known, to latest.

Activation Blocks Layout on mips [Larus, 1990]



Frame & Stack Pointers

The stack of activation blocks is implemented as an array with

frame pointer the inner frontier of the activation block

stack pointer the outer frontier

Usually the stack is represented growing towards the bottom.

Flexible Automatic Memory

`auto` Static size, automatic memory.

`malloc` Dynamic size, persistent memory.

Automatic memory is extremely convenient...

```
int
open2(char* str1, char* str2, int flags, int mode)
{
  char name[strlen(str1) + strlen(str2) + 1];
  stpcpy(stpcpy(name, str1), str2);
  return open(name, flags, mode);
}
```

Flexible Automatic Memory

malloc is a poor replacement.

```
int
open2(char* str1, char* str2, int flags, int mode)
{
  char* name
    = (char*) malloc(strlen(str1) + strlen(str2) + 1);
  if (name == 0)
    fatal("virtual memory exceeded");
  stpcpy(stpcpy(name, str1), str2);
  int fd = open(name, flags, mode);
  free(name);
  return fd;
}
```

Flexible Automatic Memory

alloca is a good replacement.

```
int
open2(char *str1, char *str2, int flags, int mode)
{
  char *name
    = (char *) alloca(strlen(str1) + strlen(str2) + 1);
  stpcpy(stpcpy(name, str1), str2);
  return open(name, flags, mode);
}
```

Advantages of `alloca` [Loosemore et al., 2003]

- Using `alloca` wastes very little space and is very fast.
  (It is open-coded by the GNU C compiler.)
- `alloca` does not cause memory fragmentation.
  Since `alloca` does not have separate pools for different sizes of block, space used for any size block can be reused for any other size.
- Automatically freed.
  Nonlocal exits done with `longjmp` automatically free the space allocated with `alloca` when they exit through the function that called `alloca`. This is the most important reason to use `alloca`.

Advantages of `alloca` [Loosemore et al., 2003]

- Using `alloca` wastes very little space and is very fast.
  (It is open-coded by the GNU C compiler.)
- `alloca` does not cause memory fragmentation.
  Since `alloca` does not have separate pools for different sizes of block, space used for any size block can be reused for any other size.
- Automatically freed.
  Nonlocal exits done with `longjmp` automatically free the space allocated with `alloca` when they exit through the function that called `alloca`. This is the most important reason to use `alloca`.

Advantages of `alloca` [Loosemore et al., 2003]

- Using `alloca` wastes very little space and is very fast.
  (It is open-coded by the GNU C compiler.)
- `alloca` does not cause memory fragmentation.
  Since `alloca` does not have separate pools for different sizes of block, space used for any size block can be reused for any other size.
- Automatically freed.
  Nonlocal exits done with `longjmp` automatically free the space allocated with `alloca` when they exit through the function that called `alloca`. This is the most important reason to use `alloca`.

Disadvantages of `alloca` [Loosemore et al., 2003]

- If you try to allocate more memory than the machine can provide, you don't get a clean error message.
  Instead you get a fatal signal like the one you would get from an infinite recursion; probably a segmentation violation.
- Some non-GNU systems fail to support `alloca`, so it is less portable. However, a slower emulation of `alloca` written in C is available for use on systems with this deficiency.

Disadvantages of `alloca` [Loosemore et al., 2003]

- If you try to allocate more memory than the machine can provide, you don't get a clean error message.
  Instead you get a fatal signal like the one you would get from an infinite recursion; probably a segmentation violation.
- Some non-GNU systems fail to support `alloca`, so it is less portable. However, a slower emulation of `alloca` written in C is available for use on systems with this deficiency.

Arrays vs. Alloca [Loosemore et al., 2003]

- A variable size array's space is freed at the end of the scope of the name of the array.
  The space allocated with `alloca` remains until the end of the function.
- It is possible to use `alloca` within a loop, allocating an additional block on each iteration.
  This is impossible with variable-sized arrays.

Arrays vs. Alloca [Loosemore et al., 2003]

- A variable size array's space is freed at the end of the scope of the name of the array.
  The space allocated with `alloca` remains until the end of the function.
- It is possible to use `alloca` within a loop, allocating an additional block on each iteration.
  This is impossible with variable-sized arrays.

Implementing Dynamic Arrays & Alloca

- Playing with `$sp` which makes `$fp` mandatory.
- An additional stack (as with the C emulation of `alloca`).

Nonlocal Variables

escapes-n-recursion

```
let function trace(fn: string, val: int) =
      (print(fn); print("("); print_int(val); print(") "))

    function one(input : int) =
    let function two() =
        (trace("two", input); one(input - 1))
    in
      if input > 0 then
        (two(); trace("one", input))
    end
in
  one(3); print("\n")
end
```

escapes-n-recursion

```
let function trace(fn: string, val: int) =
      (print(fn); print("("); print_int(val); print(") "))

    function one(input : int) =
    let function two() =
        (trace("two", input); one(input - 1))
    in
      if input > 0 then
        (two(); trace("one", input))
    end
in
  one(3); print("\n")
end
```

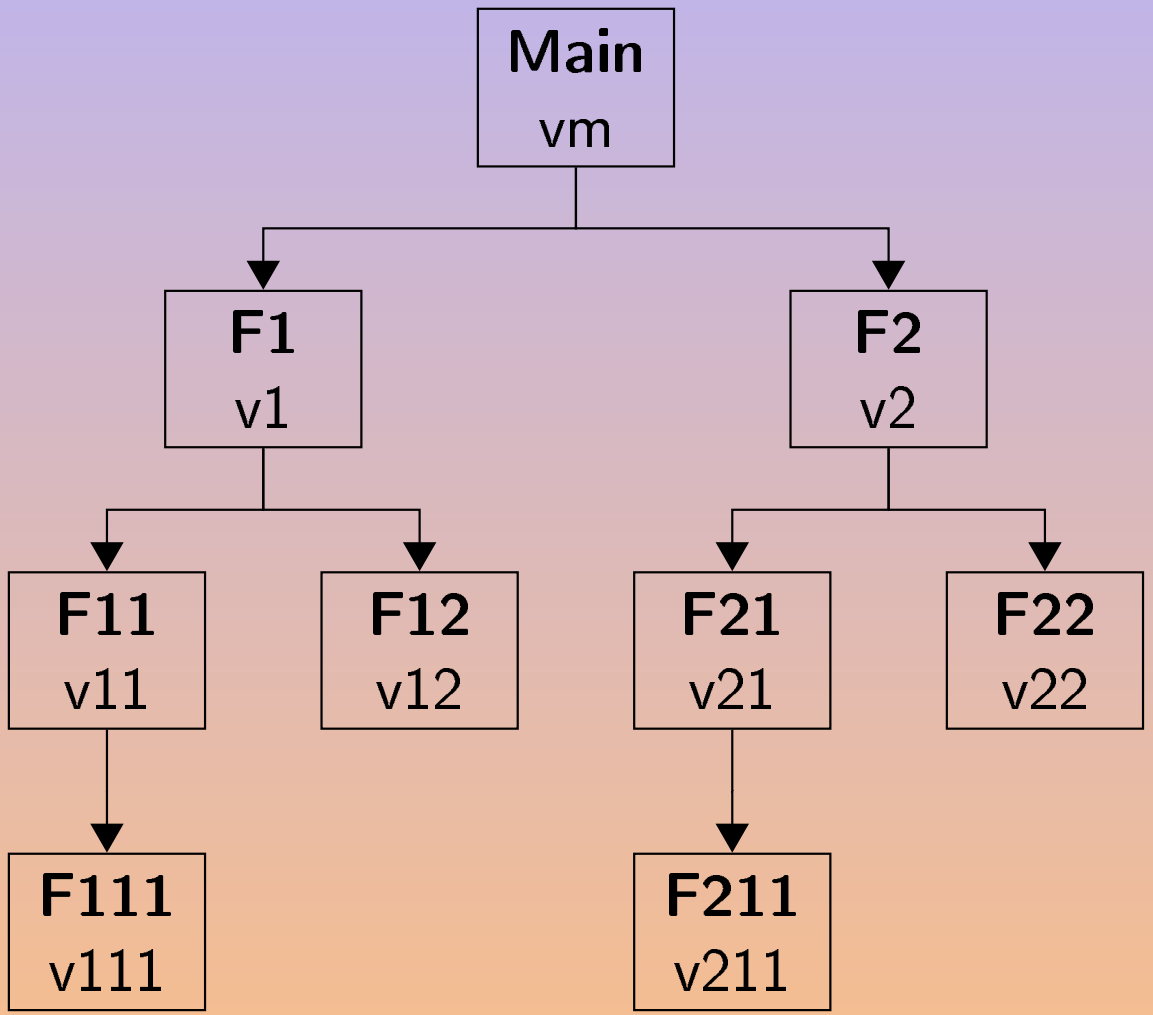
```
% tc -H escapes-n-recursion.tig > f.hir && havm f.hir
```

escapes-n-recursion

```
let function trace(fn: string, val: int) =
      (print(fn); print("("); print_int(val); print(") "))

    function one(input : int) =
    let function two() =
        (trace("two", input); one(input - 1))
    in
      if input > 0 then
        (two(); trace("one", input))
    end
in
  one(3); print("\n")
end
```

```
% tc -H escapes-n-recursion.tig > f.hir && havm f.hir
two(3) two(2) two(1) one(1) one(2) one(3)
```

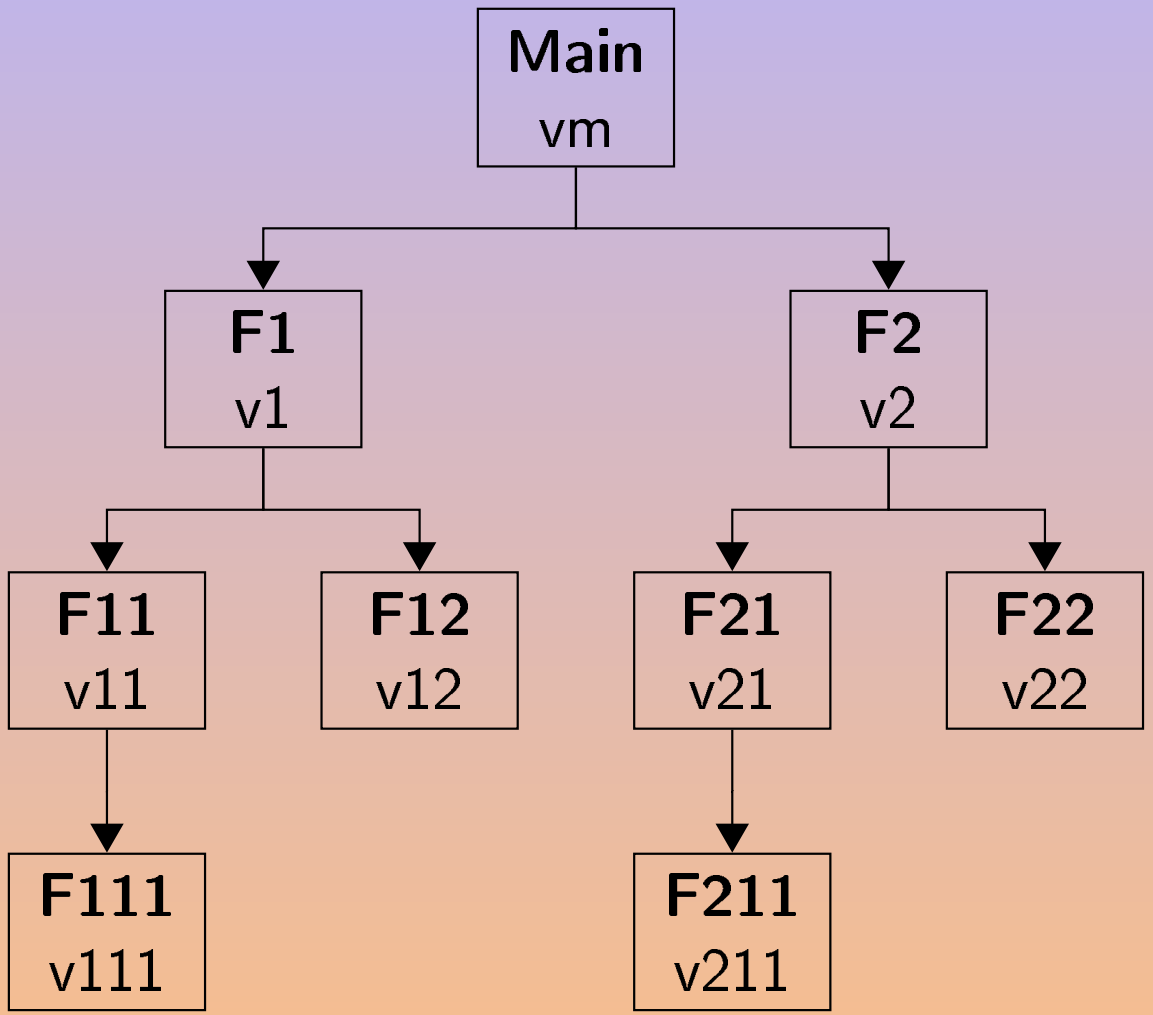
Deep Static Function Hierarchies



What if:

- Main uses vm
- Main calls F1
- F1 uses v1
- F1 uses vm, non local
- F1 calls F11
- F11 uses v11
- F11 uses v1
- F11 uses vm
- F11 calls F12
- F12 calls F1

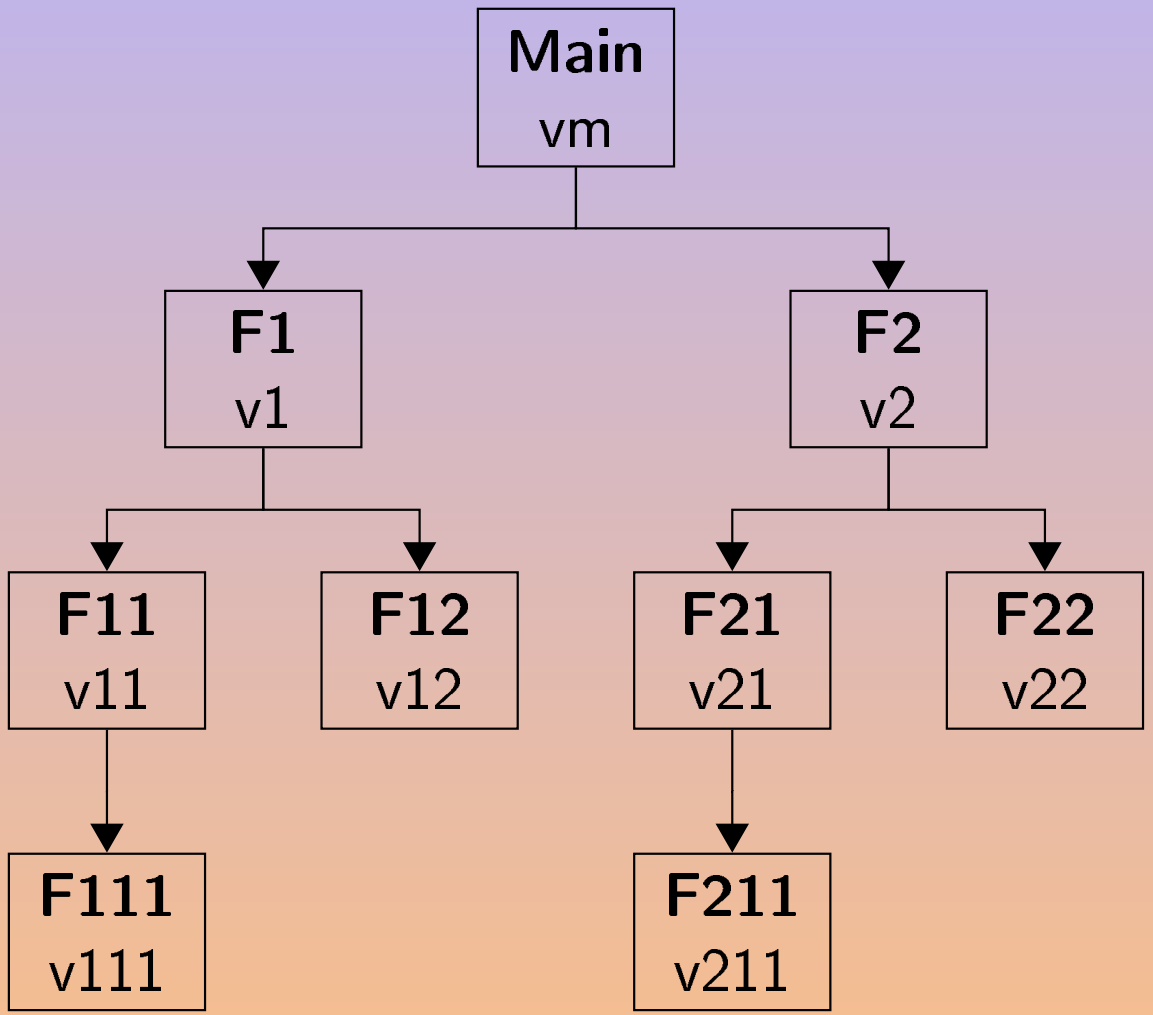
Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

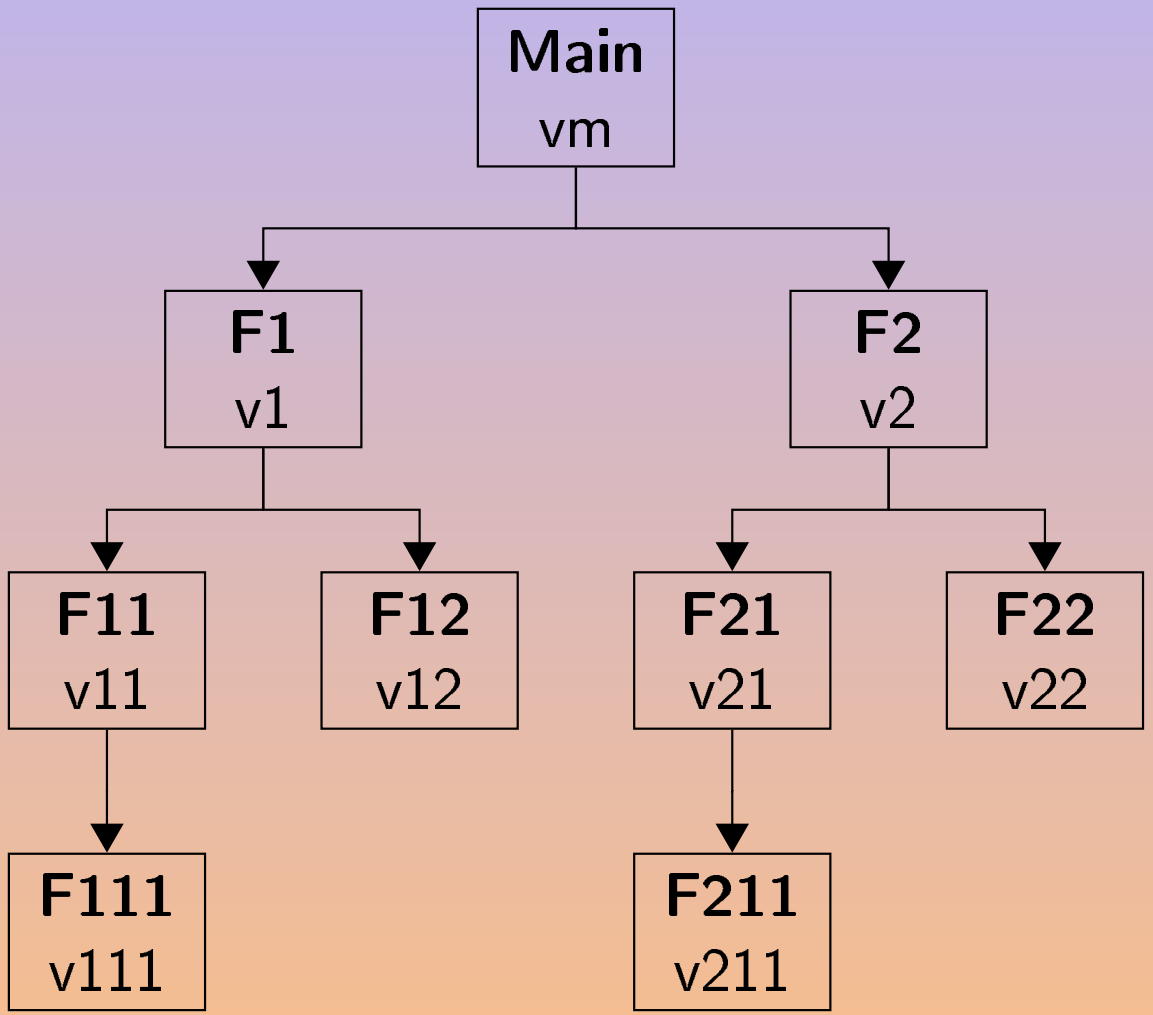
Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

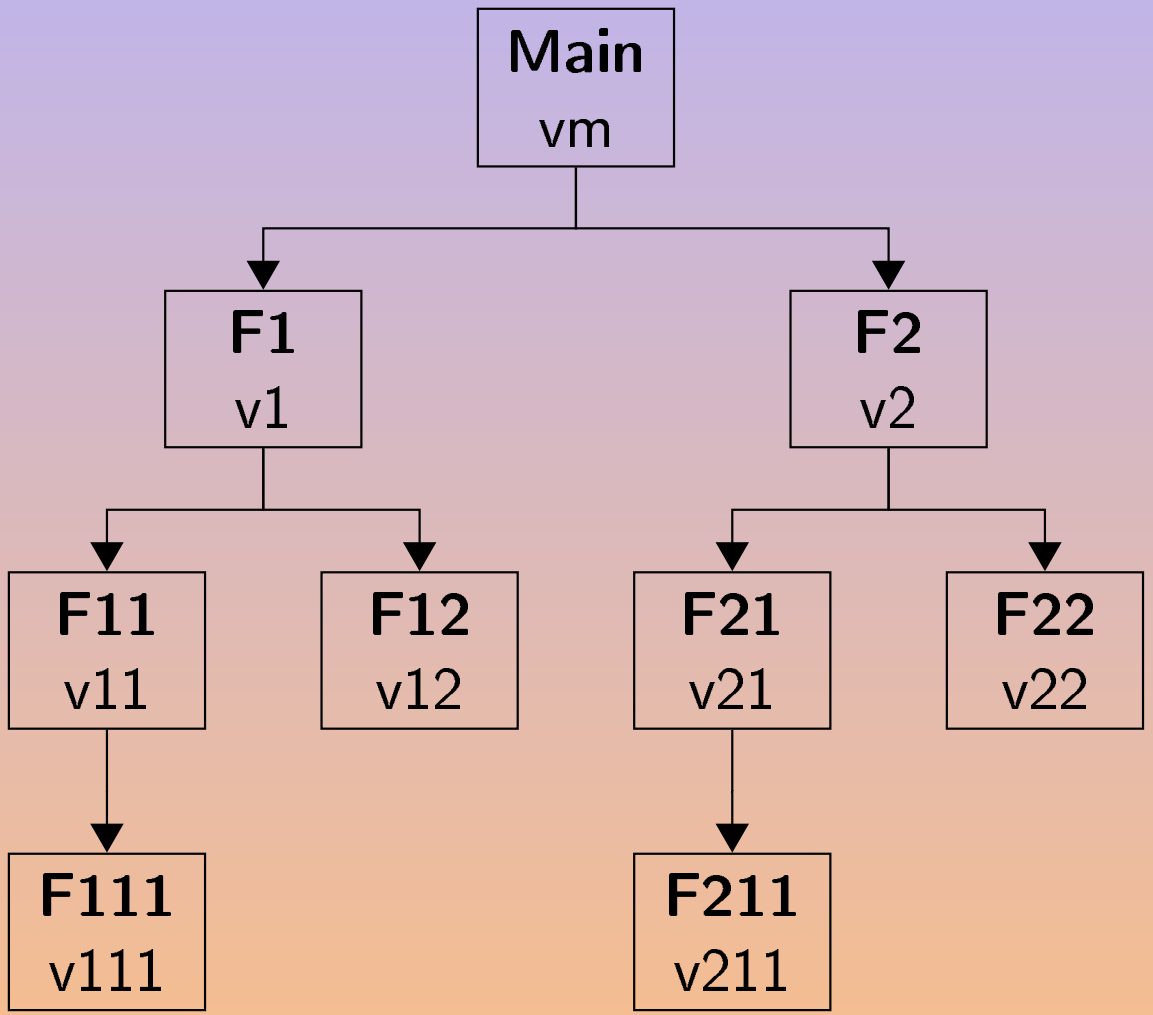
Deep Static Function Hierarchies



What if:

1. `Main` uses `vm`
2. `Main` calls `F1`
3. `F1` uses `v1`
4. `F1` uses `vm`, non local
5. `F1` calls `F11`
6. `F11` uses `v11`
7. `F11` uses `v1`
8. `F11` uses `vm`
9. `F11` calls `F12`
10. `F12` calls `F1`

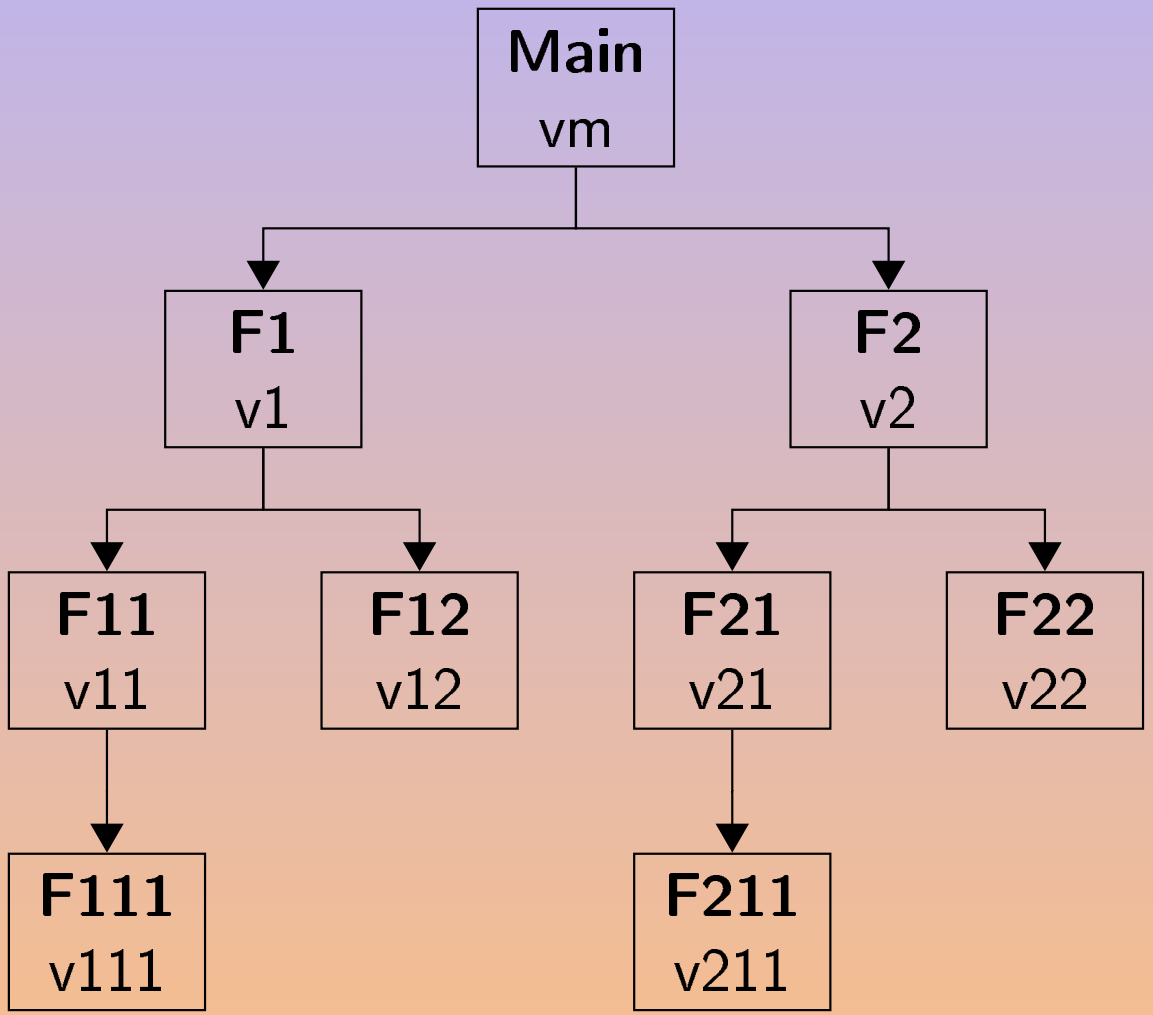
Deep Static Function Hierarchies



What if:

1. `Main uses vm`
2. `Main calls F1`
3. `F1 uses v1`
4. `F1 uses vm, non local`
5. `F1 calls F11`
6. `F11 uses v11`
7. `F11 uses v1`
8. `F11 uses vm`
9. `F11 calls F12`
10. `F12 calls F1`

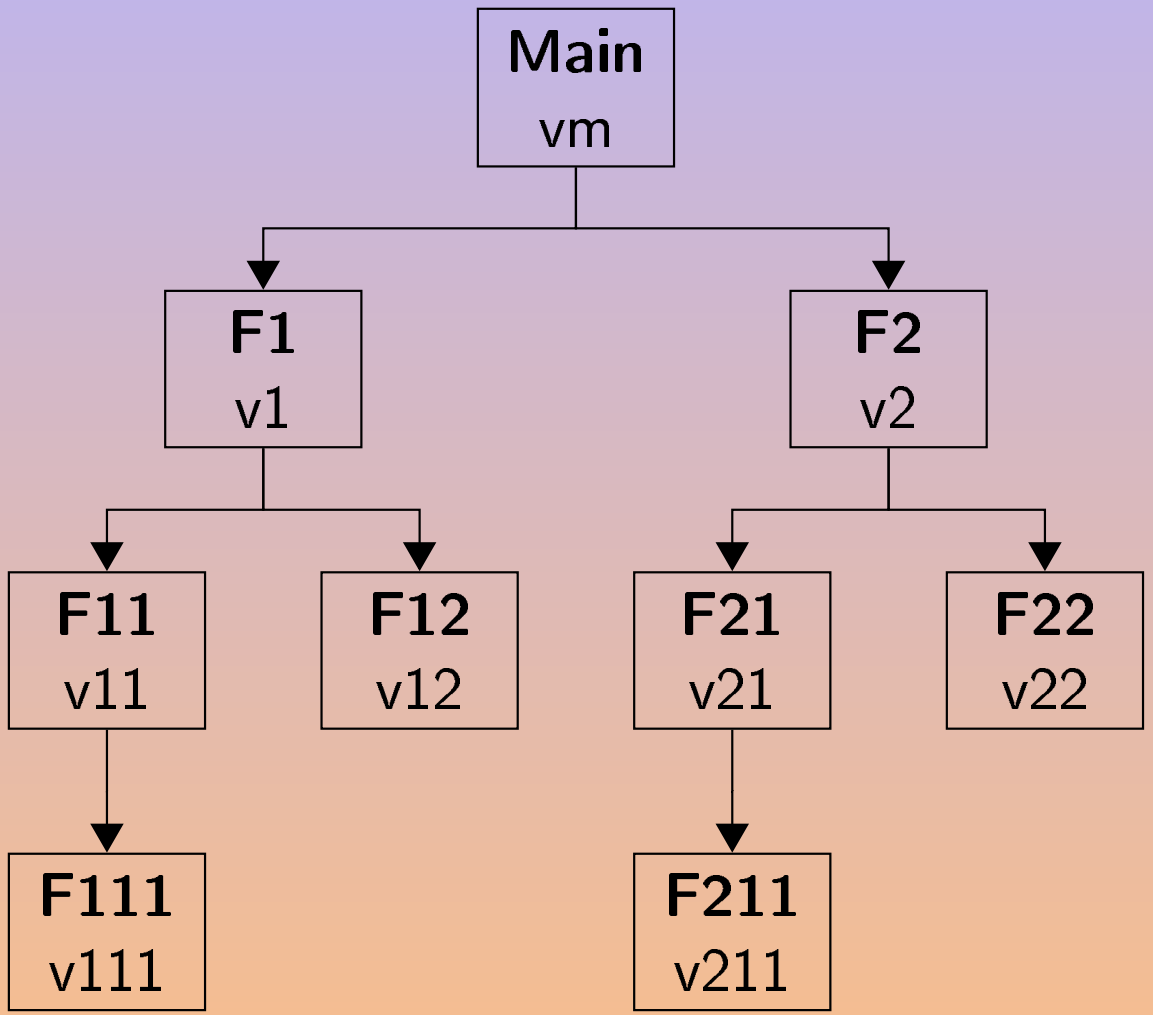
Deep Static Function Hierarchies



What if:

1. `Main` uses `vm`
2. `Main` calls `F1`
3. `F1` uses `v1`
4. `F1` uses `vm`, non local
5. `F1` calls `F11`
6. `F11` uses `v11`
7. `F11` uses `v1`
8. `F11` uses `vm`
9. `F11` calls `F12`
10. `F12` calls `F1`

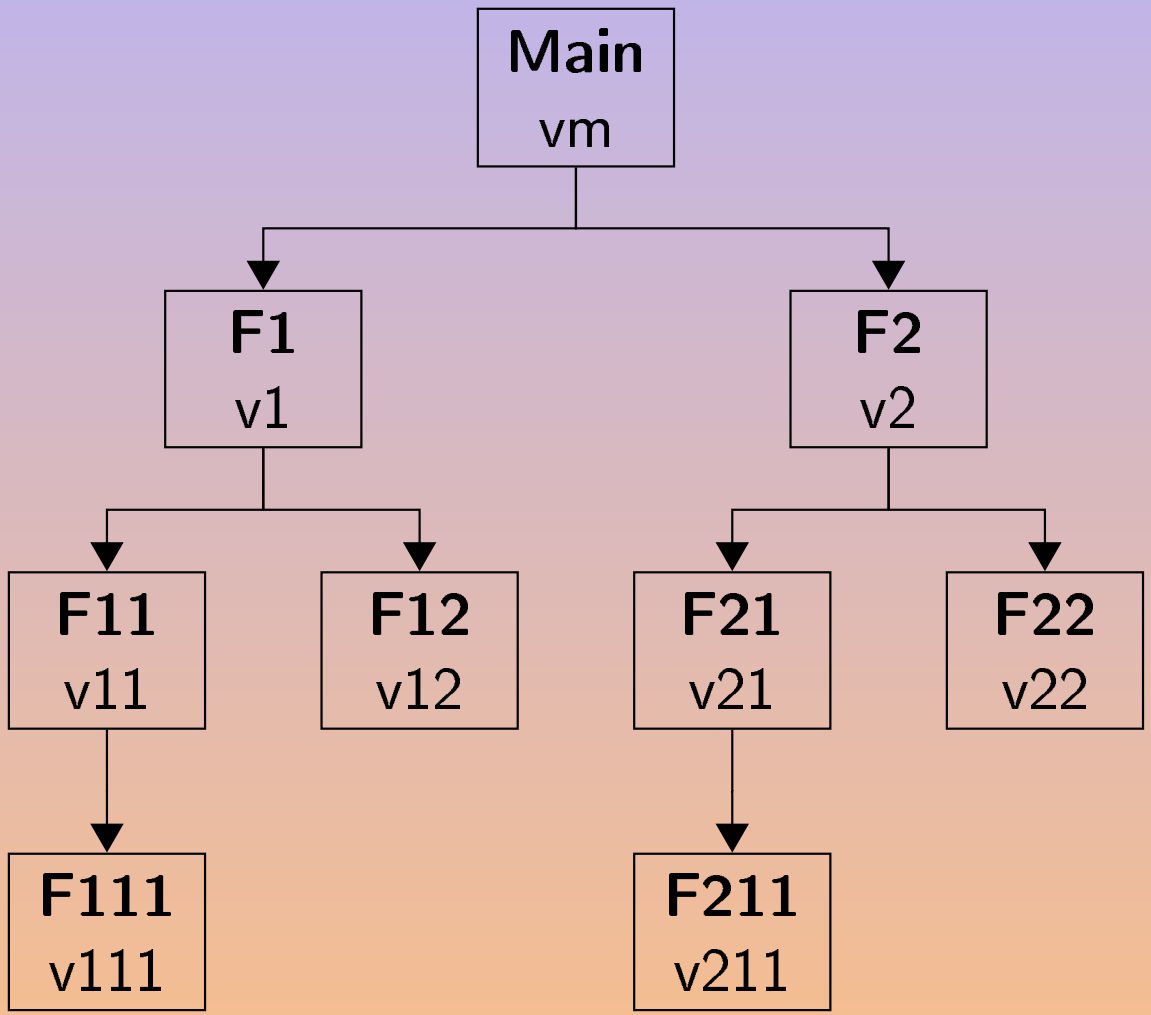
Deep Static Function Hierarchies



What if:

1. `Main` uses `vm`
2. `Main` calls `F1`
3. `F1` uses `v1`
4. `F1` uses `vm`, non local
5. `F1` calls `F11`
6. `F11` uses `v11`
7. `F11` uses `v1`
8. `F11` uses `vm`
9. `F11` calls `F12`
10. `F12` calls `F1`

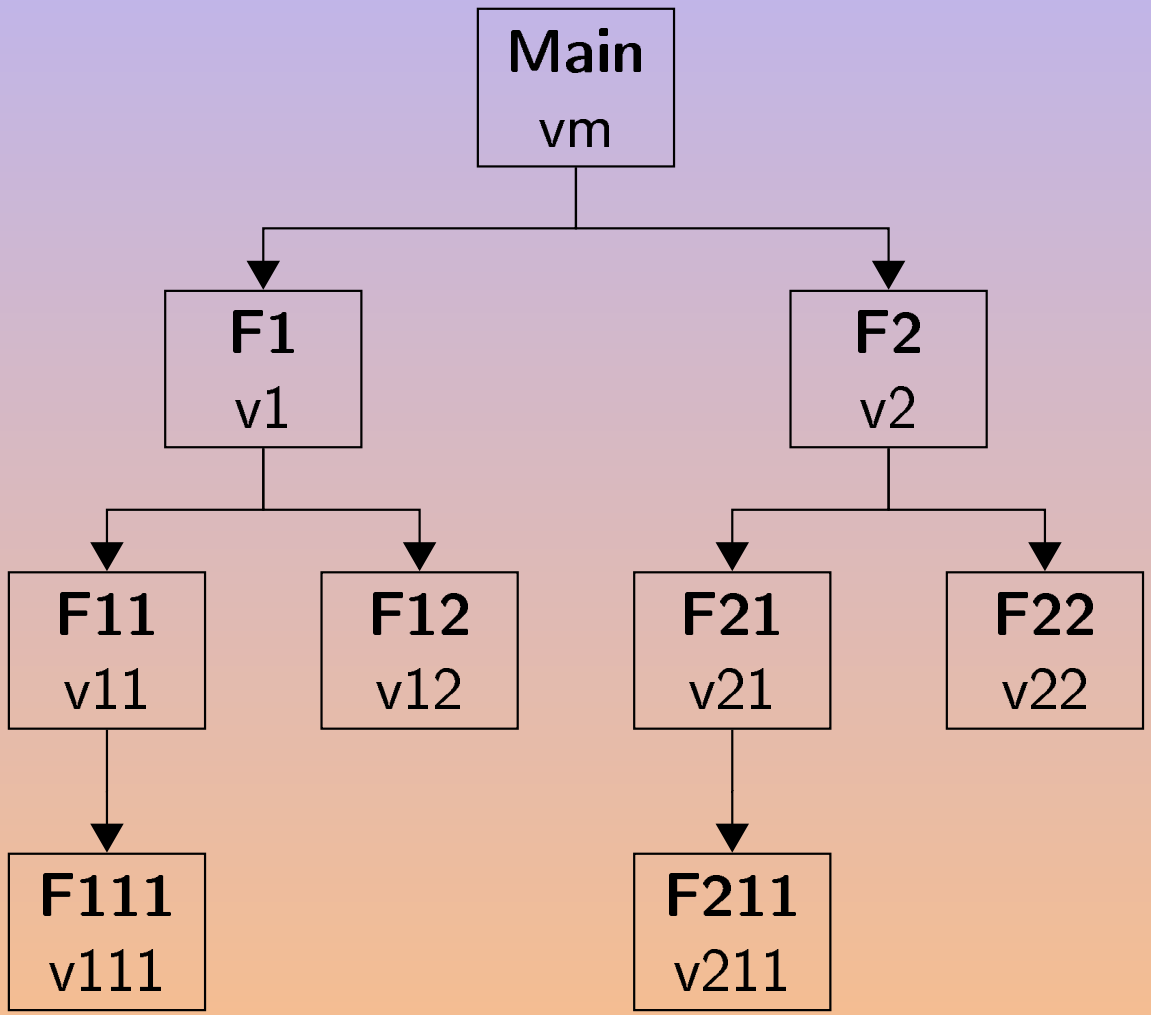
Deep Static Function Hierarchies



What if:

1. `Main` uses `vm`
2. `Main` calls `F1`
3. `F1` uses `v1`
4. `F1` uses `vm`, non local
5. `F1` calls `F11`
6. `F11` uses `v11`
7. `F11` uses `v1`
8. `F11` uses `vm`
9. `F11` calls `F12`
10. `F12` calls `F1`

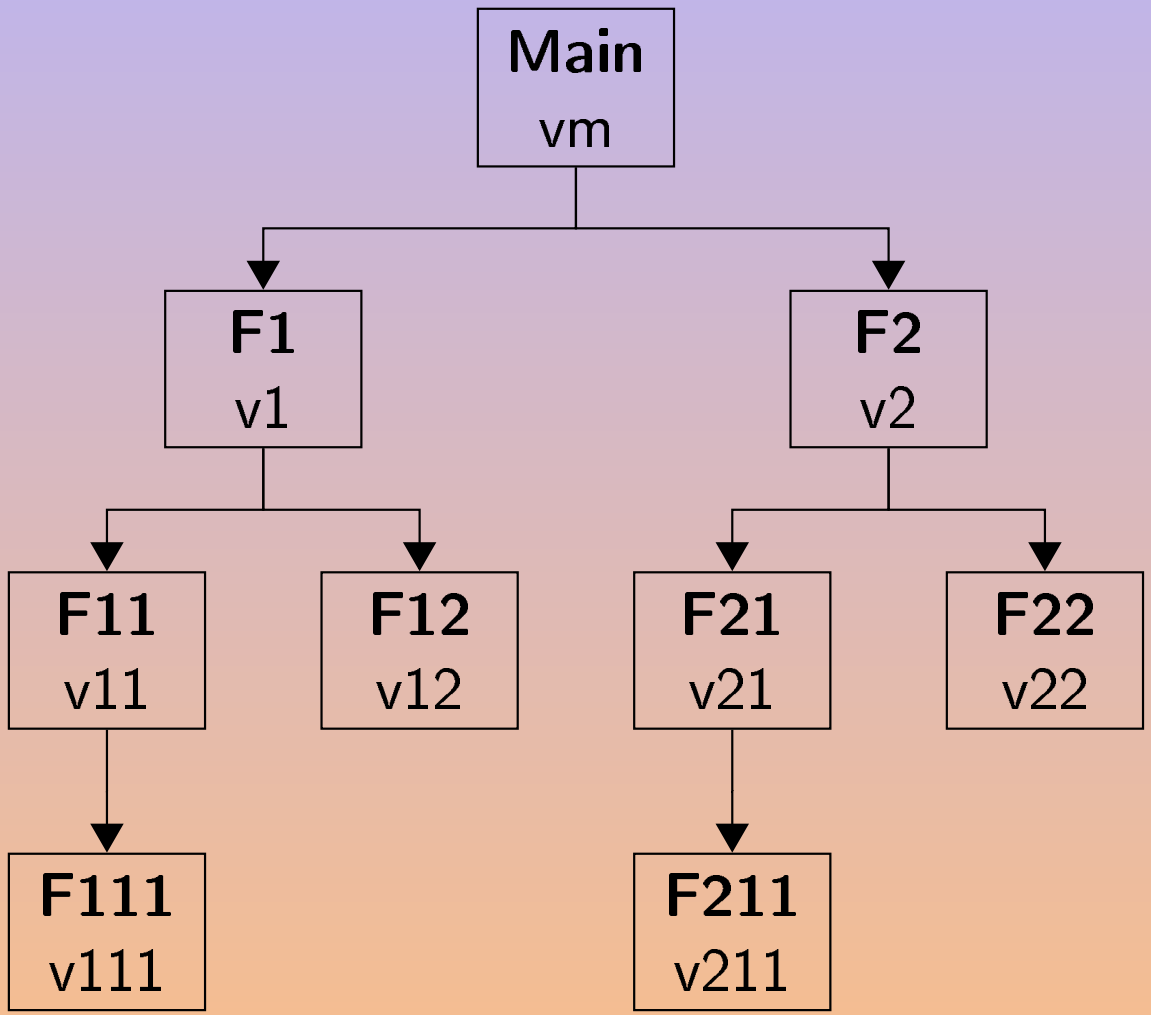
Deep Static Function Hierarchies



What if:

1. `Main` uses `vm`
2. `Main` calls `F1`
3. `F1` uses `v1`
4. `F1` uses `vm`, non local
5. `F1` calls `F11`
6. `F11` uses `v11`
7. `F11` uses `v1`
8. `F11` uses `vm`
9. `F11` calls `F12`
10. `F12` calls `F1`

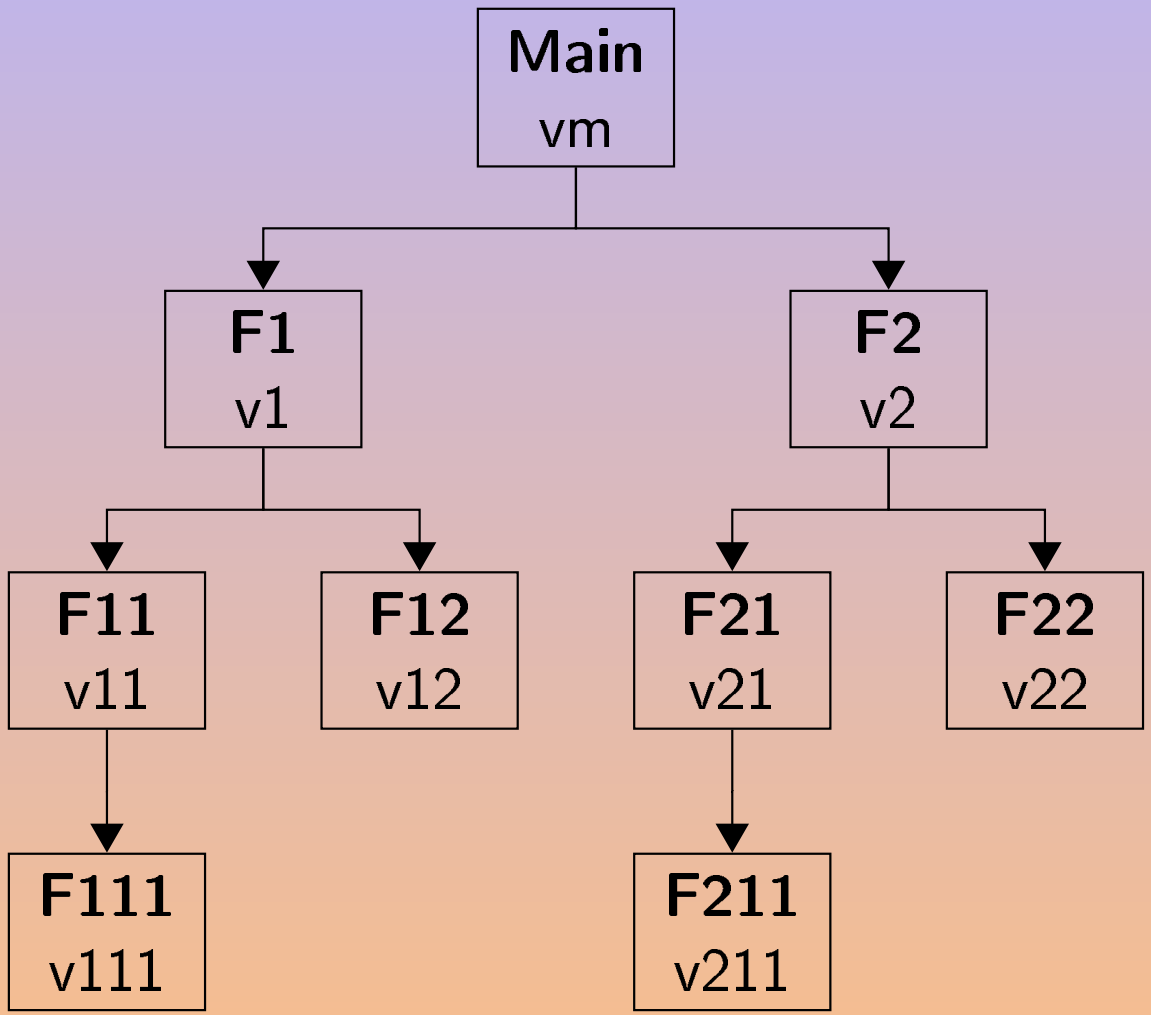
Deep Static Function Hierarchies



What if:

1. `Main` uses `vm`
2. `Main` calls `F1`
3. `F1` uses `v1`
4. `F1` uses `vm`, non local
5. `F1` calls `F11`
6. `F11` uses `v11`
7. `F11` uses `v1`
8. `F11` uses `vm`
9. `F11` calls `F12`
10. `F12` calls `F1`

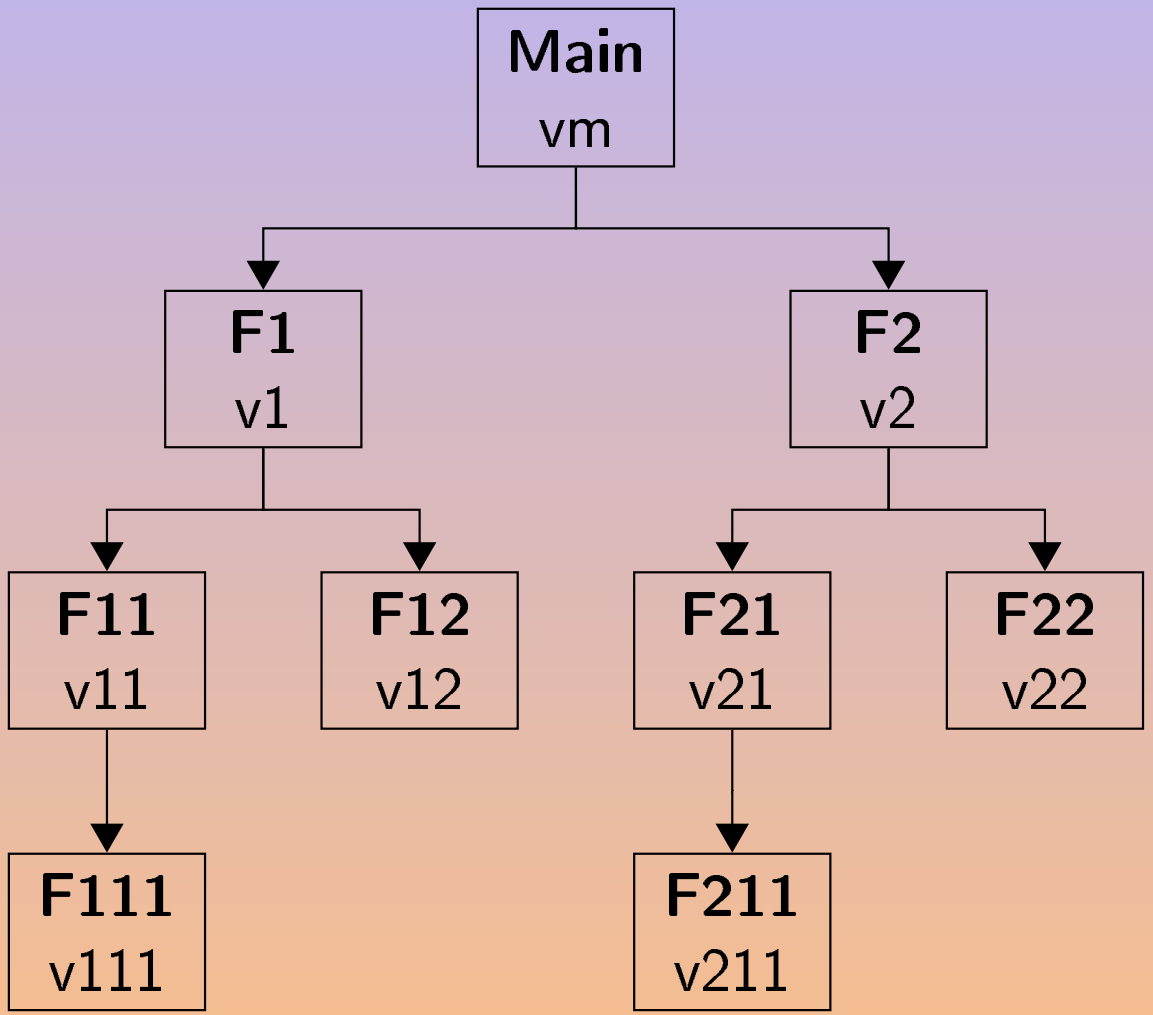
Deep Static Function Hierarchies



What if:

1. `Main` uses `vm`
2. `Main` calls `F1`
3. `F1` uses `v1`
4. `F1` uses `vm`, non local
5. `F1` calls `F11`
6. `F11` uses `v11`
7. `F11` uses `v1`
8. `F11` uses `vm`
9. `F11` calls `F12`
10. `F12` calls `F1`

Deep Static Function Hierarchies



What if:

1. `Main` uses `vm`
2. `Main` calls `F1`
3. `F1` uses `v1`
4. `F1` uses `vm`, non local
5. `F1` calls `F11`
6. `F11` uses `v11`
7. `F11` uses `v1`
8. `F11` uses `vm`
9. `F11` calls `F12`
10. `F12` calls `F1`

Deep Static Function Hierarchies

The caller must provide the callee with its static link.

Caller	Callee	Static Link
Main	F1	$\texttt{fp}_{\texttt{Main}}$ = fp
F1	F11	$\texttt{fp}_{\texttt{F1}}$ = fp
F11	F12	$\texttt{fp}_{\texttt{F1}}$ = $\texttt{sl}_{\texttt{F11}}$ = $*\texttt{fp}_{\texttt{F11}}$ = *fp
F12	F2	$\texttt{fp}_{\texttt{Main}}$ = $\texttt{sl}_{\texttt{F1}}$ = $*\texttt{sl}_{\texttt{F12}}$ = $**\texttt{fp}_{\texttt{F12}}$ = **fp
F2	F22	$\texttt{fp}_{\texttt{F2}}$ = fp
F22	F11	$\texttt{fp}_{\texttt{F1}}$ = ???

Assuming that the static link is stored at fp.

Higher Order Functions

```
let
  function addgen (a: int) : int -> int =
    let
      function res (b: int) : int =
        a + b
    in
      res
    end
  var add50 := addgen (50)
in
  add50 (1)
end
```

Translation to Intermediate Language

Calling Conventions

Calling Conventions at hir Level

You must:

- Preserve *some* registers (`fp`, `sp`)
- Allocate the frame
- Handle the static link (`i0`)
- Receive the (other) arguments (`i1`, `i2`...)

You don't:

- Save temporaries (`havm` has magic for recursion)
- Jump to the `ra` (this is not nice feature from `havm`)

havm Calling Conventions

```
let function gcd (a: int, b: int) : int = (...)
in print_int (gcd (42, 51)) end
```

```
# Routine: gcd
label l0
# Prologue
move temp t2, temp fp
move temp fp, temp sp
move temp sp, temp sp - const 4
move mem temp fp, temp i0
move temp t0, temp i1
move temp t1, temp i2
```

```
# Body
move temp rv
     eseq
       ...
       temp t0
# Epilogue
move temp sp, temp fp
move temp fp, temp t2
label end
# Routine: Main Program
label Main
sxp call name print_int
          call name l0 temp fp
               const 42 const 51
label end
```

Clever Translations

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

It depends on the *use*:

1. if $\alpha < \beta$ then ...
2. a := $\alpha < \beta$
3. $(\alpha < \beta, ())$.

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

It depends on the *use*:

1. `if` $\alpha < \beta$ `then ...`
2. `a :=` $\alpha < \beta$
3. $(\alpha < \beta, ())$.

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

It depends on the *use*:

1. `if` $\alpha < \beta$ `then ...`
2. `a :=` $\alpha < \beta$
3. $(\alpha < \beta, ())$.

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

It depends on the *use*:

1. `if` $\alpha < \beta$ `then ...`
2. `a :=` $\alpha < \beta$
3. $(\alpha < \beta$, ()).

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)			
Cx(a < b)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)		
Cx(a < b)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	
Cx(a < b)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- Ex Expression shell, encapsulation of a proto value,
- Nx Statement shell, encapsulating a wannabe statement,
- Cx Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)	seq(sxp(a), sxp(b))		
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$($a < b$), t)	
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$(a $<$ b), t)	cjump(a $<$ b, t, f)
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$(a $<$ b), t)	cjump(a $<$ b, t, f)
Nx(s)	s		

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- Ex Expression shell, encapsulation of a proto value,
- Nx Statement shell, encapsulating a wannabe statement,
- Cx Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$(a $<$ b), t)	cjump(a $<$ b, t, f)
Nx(s)	s	???	

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$(a $<$ b), t)	cjump(a $<$ b, t, f)
Nx(s)	s	???	???

if 11 < 22 | 22 < 33 then print_int(1) else print_int(0)

```
cjump ne
    eseq seq cjump 11 < 22 name l0 name l1
             label l0  move temp t0 const 1
                       jump name l2
             label l1  move temp t0
                       eseq seq move temp t1 const 1
                                cjump 22 < 33 name l3 name l4
                                label l4
                                move temp t1 const 0
                                label l3
                           seq end
                           temp t1
                      jump name l2
             label l2
         seq end
         temp t0
    const 0
    name l5
    name l6
label l5   sxp call name print_int const 1
           jump name l7
label l6   sxp call name print_int const 0
           jump name l7
label l7
```

A Better Translation: Ix

```
  seq
    cjump l1 < 22 name l3 name l4
    label l3
      cjump 1 <> 0 name l0 name l1
    label l4
      cjump 22 < 33 name l0 name l1
  seq end
label l0
  sxp call name print_int const 1
  jump name l2
label l1
  sxp call name print_int const 0
label l2
```

Complex Expressions

Complex Expressions

- Array creation
- Record creation
- String comparison
- While loops
- For loops

While Loops

```
while condition
  do body
```

While Loops

```
while condition
  do body
```

```
test:
  if not (condition)
    goto done
  body
  goto test
done:
```

For Loops

```
for i := min to max
  do body
```

```
let i := min
    limit := max
in
  while i <= limit
  do
    (body; ++i)
end
```

For Loops

```
for i := min to max
  do body
```

```
let i := min
    limit := max
in
  if (i > limit)
    goto end
loop:
    body
    if (i >= limit)
      goto end
    ++i
    goto loop
end:
```

Additional Features

- Bounds checking
- Nil checking
- ...

The Case of the Tiger Compiler

Translation in the Tiger Compiler

Actors: The `temp` Module

`temp::Temp`	*temporaries* are pseudo-registers. Generation of fresh temporaries.
`temp::Label`	Pseudo addresses, both for data and code. Generation of fresh labels.
`misc::endo_map<T>`	Mapping from `T` to `T`. Used during register allocation.

Actors: The `tree` Module

Implementation of hir and lir.

```
/Tree/   /Exp/    Const (int)
                  Name  (temp::Label)
                  Temp  (temp::Temp)
                  Binop (Oper, Exp, Exp)
                  Mem   (Exp)
                  Call  (Exp, list<Exp*>)
                  Eseq  (Stm, Exp)
         /Stm/    Move  (Exp, Exp)
                  Sxp   (Exp)
                  Jump  (Exp, list<temp::Label>)
                  CJump (Relop, Exp, Exp, Label, Label)
                  Seq   (list<Stm *>)
                  Label (temp::Label)
```

Actors: The `tree` Module: Warnings

- `temp::Temp` is not `tree::Temp`.
  The latter aggregates one of the former.
  Similarly with `Label`.
- n-ary seq.
  (Unlike [Appel, 1998]).
- `Sxp` instead of `Exp`.

Actors: The `frame` Module

`Access` How to reach a "variable".
Abstract class with two concrete subclasses.

`frame::In_Register`

`frame::In_Frame`

`Frame` What "variables" a frame contains.

```
local_alloc(bool escapes_p) -> Access
```

Frames and (`frame::`) accesses are not aware of static links.

Actors: The `translate` Module

`Access` Static link aware version of `frame::Access`:
how to reach a variable, including non local: a `frame::Access` and a `translate::Level`.

`exp(Level use) -> Exp` Tree expression

The location of `this` Access, from the `use` point of view.

`Level` Static link aware version of `frame::Frame`:
what variables a frame contains, and *where is its parent level*.

`fp(Level use) -> Exp` Tree expression

The frame pointer of `this` Level, from the `use` point of view.
Used for `calls`, and reaching frame resident temporaries.

Actors: The `translate` Module

`translate::Exp`
: Prototranslation wrappers (`Ex`, `Nx`, `Cx`, `Ix`).

`translate/translation.hh`
: Auxiliary functions used by the `Translator`.

`translate::Translator`
: The translator.

lir: Low Level Intermediate Representation

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no "expression".
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide "jump or continue" instructions.
- Limited Number of Registers
  From `temps` to actual registers.

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no “expression”.
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide “jump or continue” instructions.
- Limited Number of Registers
  From `temps` to actual registers.

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no “expression”.
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide “jump or continue” instructions.
- Limited Number of Registers
  From `temps` to actual registers.

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no “expression”.
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide “jump or continue” instructions.
- Limited Number of Registers
  From `temps` to actual registers.

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no "expression".
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide "jump or continue" instructions.
- Limited Number of Registers
  From `temps` to actual registers.

Linearization: Principle

- eseq and seq must be eliminated (except the outermost seq).
- Similar to cut-elimination: permute inner eseq and seq to lift them higher, until they vanish.
- A simple rewriting system.

```
eseq (s1, eseq (s2, e))  ⇝ eseq (seq (s1, s2), e)
sxp (eseq (s, e))        ⇝ seq (s, sxp (e))
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3)
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es)
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
binop (+, eseq (s, e1), e2)
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
binop (+, eseq (s, e1), e2) ⇝ eseq (s, binop (+, e1, e2))
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
binop (+, eseq (s, e1), e2) ⇝ eseq (s, binop (+, e1, e2))
binop (+, e1, eseq (s, e2))
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
binop (+, eseq (s, e1), e2) ⇝ eseq (s, binop (+, e1, e2))
binop (+, e1, eseq (s, e2)) ⇝ eseq (s, binop (+, e1, e2))
```

Linearization: Incorrect Changes

```
binop (+, e1, eseq (s, e2)) ⇝ eseq (s, binop (+, e1, e2))
```

- But what if s modifies the value of e1?

```
binop (+, temp t,
          eseq (move (temp t, const 42),
                const 0))
  ⇝ eseq (move (temp t, const 42),
          binop (+, temp t, const 0))
```

- This transformation is invalid: it changes the semantics.
- How can it be solved?

Linearization: Incorrect Changes

```
binop (+, e1, eseq (s, e2)) ⇝ eseq (s, binop (+, e1, e2))
```

- But what if s modifies the value of e1?

```
binop (+, temp t,
          eseq (move (temp t, const 42),
                const 0))
  ⇝ eseq (move (temp t, const 42),
          binop (+, temp t, const 0))
```

- This transformation is invalid: it changes the semantics.
- How can it be solved?

Linearization: Incorrect Changes

```
binop (+, e1, eseq (s, e2)) ⇝ eseq (s, binop (+, e1, e2))
```

- But what if s modifies the value of e1?

```
binop (+, temp t,
          eseq (move (temp t, const 42),
                const 0))
  ⇝ eseq (move (temp t, const 42),
          binop (+, temp t, const 0))
```

- This transformation is invalid: it changes the semantics.
- How can it be solved?

Linearization: Incorrect Changes

t + (t := 42, 0)

```
binop (+,
       temp t,
       eseq (move (temp t, const 42),
             const 0))
```

Wrong

```
eseq (move (temp t,
            const 42),
      binop (+,
             temp t,
             const 0))
```

Linearization: Incorrect Changes

t + (t := 42, 0)

```
binop (+,
       temp t,
       eseq (move (temp t, const 42),
             const 0))
```

Wrong

```
eseq (move (temp t,
            const 42),
      binop (+,
             temp t,
             const 0))
```

Linearization: Incorrect Changes

t + (t := 42, 0)

```
binop (+,
       temp t,
       eseq (move (temp t, const 42),
             const 0))
```

Wrong

```
eseq (move (temp t,
            const 42),
      binop (+,
             temp t,
             const 0))
```

Right

```
eseq (seq (move (temp t0, temp t)
           move (temp t, const 42)),
      binop (+,
             temp t0,
             const 0))
```

Linearization: More Temporaries

- When "de-expressioning" fresh temporaries are needed

```
binop (+, e1, eseq (s, e2))
 ⇝ eseq (seq (move (temp t, e1), s),
         binop (+, temp t, e2))
```

- More generally

```
call (f, es1, eseq (s, e), es2)
 ⇝ eseq (seq (move (temp t1, e1),
              move (temp t2, e2),
              move (temp t3, e3),
              ...,
              s),
         call (f, ts, e, es2))
```

- This is extremely inefficient when not needed...

Linearization: Commutativity

- Save useless extra temporaries and moves.
- Problem: commutativity cannot be known statically.
  E.g., `move (mem (t1), e)` and `mem (t2)`
  commute iff `t1` $\neq$ `t2`.
- We need a *conservative* approximation,
  i.e., never say "commute" when they don't.
  E.g., "if `e` is a `const` then `s` and `e` definitely commute".

Linearization: Commutativity

- Save useless extra temporaries and moves.
- Problem: commutativity cannot be known statically.
  E.g., `move (mem (t1), e)` and `mem (t2)`
  commute iff `t1` $\neq$ `t2`.
- We need a *conservative* approximation,
  i.e., never say "commute" when they don't.
  E.g., "if `e` is a `const` then `s` and `e` definitely commute".

Linearization: Commutativity

- Save useless extra temporaries and moves.
- Problem: commutativity cannot be known statically.
  E.g., `move (mem (t1), e)` and `mem (t2)`
  commute iff `t1` $\neq$ `t2`.
- We need a *conservative* approximation,
  i.e., never say "commute" when they don't.
  E.g., "if `e` is a `const` then `s` and `e` definitely commute".

Call Normalization

Normalization of a `call` depends on the kind of the routine:

- procedure then its parent must be an `sxp`
- function then its parent must be a `move (temp t, .)`

This normalization is performed simultaneously with linearization.

Two Way Jumps

Obviously, to enable the translation of a `cjump` into actual assembly instructions, the "false" label must follow the `cjump`.
How?

Two Way Jumps: Basic Blocks

Split the long outer seq into "basic blocks":

- a single entry: the first instruction
- a single (maybe multi-) exit: the last instruction

It may require

- a new label as first instruction, to which the prologue `jumps`
- new labels after `jumps` or `cjumps`
- a new `jump` from the last instruction to the epilogue.

Two Way Jumps: Traces

Start from the initial block, and "sew" each remaining basic block to this growing "trace".

- If the last instruction is a `jump`
  - if the "destination block" is available, add it
  - otherwise, fetch any other remaining block.
- If the last instruction is a `cjump`
  - If the false destination is available, push it
  - If the true destination is available, flip the `cjump` and push it,
  - otherwise, change the `cjump` to go to a fresh label, attach this label, and finally `jump` to the initial false destination.

Two Way Jumps: Optimizing Traces

Many `jumps` should be removable, but sometimes there are choices to make.

```
label prologue
  Prologue.
jump name test
```

```
label test
cjump i <= N, body, done
```

```
label body
  Body.
jump name test
```

```
label done
  Epilogue
jump name end
```

Two Way Jumps: Optimizing Traces

```
label prologue
  Prologue
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i > N,
   done, body
```

```
label test
cjump i <= N,
   body, done
```

```
label body
  Body
jump name test
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

```
label test
cjump i <= N,
  body, done
```

```
label done
  Epilogue
jump name end
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

Two Way Jumps: Optimizing Traces

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i > N,
   done, body
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i <= N,
   body, done
```

```
label done
  Epilogue
jump name end
```

```
label body
  Body
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label body
  Body
jump name test
```

```
label test
cjump i <= N,
  body, done
```

```
label done
  Epilogue
jump name end
```

Two Way Jumps: Optimizing Traces

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i > N,
   done, body
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i <= N,
   body, done
```

```
label done
  Epilogue
jump name end
```

```
label body
  Body
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label body
  Body
jump name test
```

```
label test
cjump i <= N,
  body, done
```

```
label done
  Epilogue
jump name end
```

Two Way Jumps: Optimizing Traces

```
label prologue
  Prologue
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i > N,
   done, body
```

```
label test
cjump i <= N,
   body, done
```

```
label body
  Body
jump name test
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

```
label test
cjump i <= N,
  body, done
```

```
label done
  Epilogue
jump name end
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

Bibliography I


Appel, A. W. (1998).
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press.


Edwards, S. (2003).
COMS W4115 Programming Languages and Translators.
http://www.cs.columbia.edu/~sedwards/classes/2003/w4115/.


Larus, J. R. (1990).
SPIM S20: A MIPS R2000 simulator.
Technical Report TR966, Computer Sciences Department, University of Wisconsin–Madison.

Bibliography II

Loosemore, S., Stallman, R. M., McGrath, R., Oram, A., and Drepper, U. (2003).
The GNU C Library Reference Manual.
Free Software Foundation, 59 Temple Place – Suite 330, Boston, MA 02111-1307 USA, 0.10 edition.

Muchnick, S. (1997).
Advanced Compiler Design and Implementation.
Morgan Kaufmann Publishers.

Instruction Selection

Akim Demaille Étienne Renault Roland Levillain
first.*last*@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

April 23, 2018

Instruction Selection

Microprocessors

Instruction set architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine
IBM introducing 360 (1964)

The Instruction Set Architecture (ISA) is the part of the processor that is visible to the programmer or compiler writer.

What is an instruction set?

An instruction set specifies a processor functionality:

- what operations are supported
- what storage mechanisms are used
- how to access storage
- how to communicate program to processor

Technical aspect of instruction set

1. format: length, encoding
2. operations: data type (floating or fixed point) , number and kind of operands
3. storage:
   - internal: accumulator, stack, register
   - memory: address size, addressing modes
4. control: branch condition, support for procedures, predication

What makes a good instruction set?

An instruction set specifies a processor functionnality:

- **implementability**: support for a (high performances) range of implementation
- **programmability**: easy to express program (by Humans before 80's, mostly by compiler nowadays)
- **backward/forward compatibility**: implementability & programmability across generation

cisc – Complex Instruction Set Chip

- large number of instructions (100-250)
- 6, 8, 16 registers, some for pointers, others for integer computation
- arithmetic in memory can be processed
- two address code
- many possible effects (e.g., self-incrementation)

cisc – Pros & Cons

Pros:

- Simplified compiler: translation from IR is straightforward
- Smaller assembly code than risc assembly code
- Fewer instructions will be fetched
- Special purpose register available: stack pointer, interrupt handling ...

Cons:

- Variable length instruction format
- Many instruction require many clock for execution
- Limiter number of general purpose register
- (often) new version of cisc include the subset of instructions of the previous version

Motivations for something else!

Though the CISC programs could be small in length, but number of bits of memory occupies may not be less

The complex instructions do not simplify the compilers: many clock cycles can be wasted to find the appropriate instruction.

risc architectures were designed with the goal of executing one instruction per clock cycle.

risc – Reduced Instruction Set Chip

- 32 generic purpose registers
- arithmetic only available on registers
- 3 address code
- `load` and `store` relative to a register (`M[r + const]`)
- only one effect or result per instruction

risc – Pipeline 1/3

Pipelining is the overlapping the execution of several instructions in a pipeline fashion.

A pipeline is (typically) decomposed into five stages:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MA)
5. Write Back (WB)

risc – Pipeline 2/3

inst1:	IF	ID	EX	MA	WB			
inst2:		IF	ID	EX	MA	WB		
inst3:			IF	ID	EX	MA	WB	
inst4:				IF	ID	EX	MA	WB
inst5:					IF	ID	EX	MA	WB

The slowest stage determines the speed of the whole pipeline!

Ex introduces latency

- Register-Register Operation: 1 cycle
- Memory Reference: 2 cycles
- Multi-cycle Instructions (floating point): many cycles

risc – Pipeline 3/3

Data hazard: When an instruction depends on the results of a previous instruction still in the pipeline.

- inst1 write in $s1 during WB
- inst1 read in $s1 during ID

inst1:	IF	ID	EX	MA	WB	
inst2:		IF	ID	EX	MA	WB

inst2 must be split, causing delays...

other dependencies can appears

risc – Pros & Cons

Pros:

- Fixed length instructions: decoding is easier
- Simpler hardware: higher clock rate
- Efficient Instruction pipeline
- Large number of general purpose register
- Overlapped register windows to speed up procedure call and return
- One instruction per cycle

Cons:

- Minimal number of addressing modes: only Load and Store
- Relatively few instructions

Nowadays

- the classification pure-risc or pure-cisc is becoming more and more inappropriate and may be irrelevant
- modern processors use a calculated combination elements of both design styles
- decisive factor is based on a tradeoff between the required improvement in performance and the expected added cost
- Some processors that are classified as CISC while employing a number of RISC features, such as pipelining

ARM provides the advantage of using a CISC (in terms of functionality) and the advantage of an RISC (in terms of reduced code lengths).

Lessons to be learned

Implementability

Driven by technology: microcode, VLSI, FPGA, pipelining, superscalar, SIMD, SSE

Programmability

Driven by compiler technology

Sum-up

- Many non technical issues influence ISA's
- Best solutions don't always win (Intel X86)

Intel X86 (IA32)

- Introduced in 1978
- 8×32 bits "general" register
- variable length instructions (1–15 byte)
- long life to the king! 15 generations from Intel 8086 to Intel Kabylake

Intel's trick?

Decoder translates cisc into risc micro-operations

A Typical risc: mips

mips Registers and Use Convention [Larus, 1990]

Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0–v1	2–3	Expression evaluation and results of a function
a0–a3	4–7	Function argument 1–4
t0–t7	8–15	Temporary (not preserved across call)
s0–s7	16–23	Saved temporary (preserved across call)
t8–t9	24–25	Temporary (not preserved across call)
k0–k1	26–27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

Typical risc Instructions

The following slides are based on [Larus, 1990].

- The assembler translates pseudo-instructions (marked with † below).
- In all instructions below, `Src2` can be
  - a register
  - an immediate value (a 16 bit integer).
- The immediate forms are included for reference.
- The assembler translates the general form (e.g., `add`) into the immediate form (e.g., `addi`) if the second argument is constant.

Integer Arithmetics

Arithmetic: Addition/Subtraction

`add Rdest, Rsrc1, Src2` *Addition (with overflow)*
`addi Rdest, Rsrc1, Imm` *Addition Immediate (with overflow)*
`addu Rdest, Rsrc1, Src2` *Addition (without overflow)*
`addiu Rdest, Rsrc1, Imm` *Addition Immediate (without overflow)*

Put the sum of the integers from `Rsrc1` and `Src2` (or `Imm`) into `Rdest`.

`sub Rdest, Rsrc1, Src2` *Subtract (with overflow)*
`subu Rdest, Rsrc1, Src2` *Subtract (without overflow)*

Put the difference of the integers from `Rsrc1` and `Src2` into `Rdest`.

Arithmetic: Division

If an operand is negative, the remainder is unspecified by the mips architecture and depends on the conventions of the machine on which spim is run.

`div Rsrc1, Rsrc2` *Divide (signed)*
`divu Rsrc1, Rsrc2` *Divide (unsigned)*

Divide the contents of the two registers. Leave the quotient in register `lo` and the remainder in register `hi`.

`div Rdest, Rsrc1, Src2` *Divide (signed, with overflow)* †
`divu Rdest, Rsrc1, Src2` *Divide (unsigned, without overflow)* †

Put the quotient of the integers from `Rsrc1` and `Src2` into `Rdest`.

`rem Rdest, Rsrc1, Src2` *Remainder* †
`remu Rdest, Rsrc1, Src2` *Unsigned Remainder* †

Likewise for the the remainder of the division.

Arithmetic: Multiplication

`mul Rdest, Rsrc1, Src2` *Multiply (without overflow)* †
`mulo Rdest, Rsrc1, Src2` *Multiply (with overflow)* †
`mulou Rdest, Rsrc1, Src2` *Unsigned Multiply (with overflow)* †

Put the product of the integers from `Rsrc1` and `Src2` into `Rdest`.

`mult Rsrc1, Rsrc2` *Multiply*
`multu Rsrc1, Rsrc2` *Unsigned Multiply*

Multiply the contents of the two registers. Leave the low-order word of the product in register `lo` and the high-word in register `hi`.

Arithmetic Instructions

`abs Rdest, Rsrc` *Absolute Value* †

Put the absolute value of the integer from `Rsrc` in `Rdest`.

`neg Rdest, Rsrc` *Negate Value (with overflow)* †

`negu Rdest, Rsrc` *Negate Value (without overflow)* †

Put the negative of the integer from `Rsrc` into `Rdest`.

Logical Operations

Logical Instructions

`and Rdest, Rsrc1, Src2` — *AND*

`andi Rdest, Rsrc1, Imm` — *AND Immediate*

Put the logical AND of the integers from `Rsrc1` and `Src2` (or `Imm`) into `Rdest`.

`not Rdest, Rsrc` — *NOT* †

Put the bitwise logical negation of the integer from `Rsrc` into `Rdest`.

Logical Instructions

`nor Rdest, Rsrc1, Src2` *NOR*

Put the logical NOR of the integers from `Rsrc1` and `Src2` into `Rdest`.

`or Rdest, Rsrc1, Src2` *OR*

`ori Rdest, Rsrc1, Imm` *OR Immediate*

Put the logical OR of the integers from `Rsrc1` and `Src2` (or `Imm`) into `Rdest`.

`xor Rdest, Rsrc1, Src2` *XOR*

`xori Rdest, Rsrc1, Imm` *XOR Immediate*

Put the logical XOR of the integers from `Rsrc1` and `Src2` (or `Imm`) into `Rdest`.

Logical Instructions

`rol Rdest, Rsrc1, Src2` — *Rotate Left* †
`ror Rdest, Rsrc1, Src2` — *Rotate Right* †

Rotate the contents of `Rsrc1` left (right) by the distance indicated by `Src2` and put the result in `Rdest`.

`sll Rdest, Rsrc1, Src2` — *Shift Left Logical*
`sllv Rdest, Rsrc1, Rsrc2` — *Shift Left Logical Variable*
`sra Rdest, Rsrc1, Src2` — *Shift Right Arithmetic*
`srav Rdest, Rsrc1, Rsrc2` — *Shift Right Arithmetic Variable*
`srl Rdest, Rsrc1, Src2` — *Shift Right Logical*
`srlv Rdest, Rsrc1, Rsrc2` — *Shift Right Logical Variable*

Shift the contents of `Rsrc1` left (right) by the distance indicated by `Src2` (`Rsrc2`) and put the result in `Rdest`.

Control Flow

Comparison Instructions

`seq Rdest, Rsrc1, Src2` — *Set Equal* †

Set `Rdest` to 1 if `Rsrc1` equals `Src2`, otherwise to 0.

`sne Rdest, Rsrc1, Src2` — *Set Not Equal* †

Set `Rdest` to 1 if `Rsrc1` is not equal to `Src2`, otherwise to 0.

Comparison Instructions

`sge Rdest, Rsrc1, Src2` — *Set Greater Than Equal* †
`sgeu Rdest, Rsrc1, Src2` — *Set Greater Than Equal Unsigned* †
Set `Rdest` to 1 if `Rsrc1` $\geq$ `Src2`, otherwise to 0.

`sgt Rdest, Rsrc1, Src2` — *Set Greater Than* †
`sgtu Rdest, Rsrc1, Src2` — *Set Greater Than Unsigned* †
Set `Rdest` to 1 if `Rsrc1` $>$ `Src2`, otherwise to 0.

`sle Rdest, Rsrc1, Src2` — *Set Less Than Equal* †
`sleu Rdest, Rsrc1, Src2` — *Set Less Than Equal Unsigned* †
Set `Rdest` to 1 if `Rsrc1` $\leq$ `Src2`, otherwise to 0.

`slt Rdest, Rsrc1, Src2` — *Set Less Than*
`slti Rdest, Rsrc1, Imm` — *Set Less Than Immediate*
`sltu Rdest, Rsrc1, Src2` — *Set Less Than Unsigned*
`sltiu Rdest, Rsrc1, Imm` — *Set Less Than Unsigned Immediate*
Set `Rdest` to 1 if `Rsrc1` $<$ `Src2` (or `Imm`), otherwise to 0.

Branch and Jump Instructions

Branch instructions use a signed 16-bit offset field: jump from -2^{15} to $+2^{15}-1$) *instructions* (not bytes). The *jump* instruction contains a 26 bit address field.

`b label` *Branch instruction* †

Unconditionally branch to *label*.

`j label` *Jump*

Unconditionally jump to *label*.

`jal label` *Jump and Link*

`jalr Rsrc` *Jump and Link Register*

Unconditionally jump to *label* or whose address is in `Rsrc`. Save the address of the next instruction in register 31.

`jr Rsrc` *Jump Register*

Unconditionally jump to the instruction whose address is in register `Rsrc`.

Branch and Jump Instructions

`bczt label` *Branch Coprocessor z True*

`bczf label` *Branch Coprocessor z False*

Conditionally branch to *label* if coprocessor *z*'s condition flag is true (false).

Branch and Jump Instructions

Conditionally branch to *label* if the contents of `Rsrc1` $*$ `Src2`.

`beq Rsrc1, Src2, label`	*Branch on Equal*
`bne Rsrc1, Src2, label`	*Branch on Not Equal*
`beqz Rsrc, label`	*Branch on Equal Zero* †
`bnez Rsrc, label`	*Branch on Not Equal Zero* †

Branch and Jump Instructions

Conditionally branch to *label* if the contents of `Rsrc1` $*$ `Src2`.

`bge Rsrc1, Src2, label` — *Branch on Greater Than Equal* †
`bgeu Rsrc1, Src2, label` — *Branch on GTE Unsigned* †
`bgez Rsrc, label` — *Branch on Greater Than Equal Zero*
`bgezal Rsrc, label` — *Branch on Greater Than Equal Zero And Link*

Conditionally branch to *label* if the contents of `Rsrc` are greater than or equal to 0. Save the address of the next instruction in register 31.

`bgt Rsrc1, Src2, label` — *Branch on Greater Than* †
`bgtu Rsrc1, Src2, label` — *Branch on Greater Than Unsigned* †
`bgtz Rsrc, label` — *Branch on Greater Than Zero*

Branch and Jump Instructions

Conditionally branch to *label* if the contents of `Rsrc1` are $*$ to `Src2`.

`ble Rsrc1, Src2, label` — *Branch on Less Than Equal* †

`bleu Rsrc1, Src2, label` — *Branch on LTE Unsigned* †

`blez Rsrc, label` — *Branch on Less Than Equal Zero*

`bgezal Rsrc, label` — *Branch on Greater Than Equal Zero And Link*

`bltzal Rsrc, label` — *Branch on Less Than And Link*

Conditionally branch to *label* if the contents of `Rsrc` are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.

`blt Rsrc1, Src2, label` — *Branch on Less Than* †

`bltu Rsrc1, Src2, label` — *Branch on Less Than Unsigned* †

`bltz Rsrc, label` — *Branch on Less Than Zero*

Exception and Trap Instructions

`rfe` *Return From Exception*

Restore the Status register.

`syscall` *System Call*

Register `$v0` contains the number of the system call provided by spim.

`break n` *Break*

Cause exception *n*. Exception 1 is reserved for the debugger.

`nop` *No operation*

Do nothing.

Loads and Stores

Constant-Manipulating Instructions

`li Rdest, imm` *Load Immediate* †

Move the immediate `imm` into `Rdest`.

`lui Rdest, imm` *Load Upper Immediate*

Load the lower halfword of the immediate `imm` into the upper halfword of `Rdest`. The lower bits of the register are set to 0.

Load: Byte & Halfword

`lb Rdest, address` *Load Byte*
`lbu Rdest, address` *Load Unsigned Byte*

Load the byte at *address* into `Rdest`. The byte is sign-extended by the `lb`, but not the `lbu`, instruction.

`lh Rdest, address` *Load Halfword*
`lhu Rdest, address` *Load Unsigned Halfword*

Load the 16-bit quantity (halfword) at *address* into register `Rdest`. The halfword is sign-extended by the `lh`, but not the **lhu**, instruction

Load: Word

`lw Rdest, address` *Load Word*

Load the 32-bit quantity (word) at *address* into `Rdest`.

`lwcz Rdest, address` *Load Word Coprocessor*

Load the word at *address* into `Rdest` of coprocessor *z* (0–3).

`lwl Rdest, address` *Load Word Left*

`lwr Rdest, address` *Load Word Right*

Load the left (right) bytes from the word at the possibly-unaligned *address* into `Rdest`.

`ulh Rdest, address` *Unaligned Load Halfword* †

`ulhu Rdest, address` *Unaligned Load Halfword Unsigned* †

Load the 16-bit quantity (halfword) at the possibly-unaligned *address* into `Rdest`. The halfword is sign-extended by the `ulh`, but not the **ulhu**, instruction

`ulw Rdest, address` *Unaligned Load Word* †

Load the 32-bit quantity (word) at the possibly-unaligned *address* into `Rdest`.

Load Instructions

`la` `Rdest, address` *Load Address* †

Load computed *address*, not the contents of the location, into `Rdest`.

`ld` `Rdest, address` *Load Double-Word* †

Load the 64-bit quantity at *address* into `Rdest` and `Rdest + 1`.

Store: Byte & Halfword

`sb Rsrc, address` *Store Byte*

Store the low byte from `Rsrc` at *address*.

`sh Rsrc, address` *Store Halfword*

Store the low halfword from `Rsrc` at *address*.

Store: Word

`sw Rsrc, address` *Store Word*

Store the word from `Rsrc` at *address*.

`swcz Rsrc, address` *Store Word Coprocessor*

Store the word from `Rsrc` of coprocessor *z* at *address*.

`swl Rsrc, address` *Store Word Left*

`swr Rsrc, address` *Store Word Right*

Store the left (right) bytes from `Rsrc` at the possibly-unaligned *address*.

`ush Rsrc, address` *Unaligned Store Halfword* †

Store the low halfword from `Rsrc` at the possibly-unaligned *address*.

`usw Rsrc, address` *Unaligned Store Word* †

Store the word from `Rsrc` at the possibly-unaligned *address*.

Store: Double Word

`sd Rsrc, address` *Store Double-Word* †

Store the 64-bit quantity in `Rsrc` and `Rsrc + 1` at *address*.

Data Movement Instructions

`move Rdest, Rsrc` — *Move* †

Move the contents of `Rsrc` to `Rdest`.

The multiply and divide unit produces its result in two additional registers, `hi` and `lo` (e.g., `mul Rdest, Rsrc1, Src2`).

`mfhi Rdest` — *Move From hi*

`mflo Rdest` — *Move From lo*

Move the contents of the `hi` (`lo`) register to `Rdest`.

`mthi Rdest` — *Move To hi*

`mtlo Rdest` — *Move To lo*

Move the contents `Rdest` to the `hi` (`lo`) register.

Data Movement Instructions

Coprocessors have their own register sets. These instructions move values between these registers and the CPU's registers.

`mfcz Rdest, CPsrc` *Move From Coprocessor z*

Move the contents of coprocessor z's register `CPsrc` to CPU `Rdest`.

`mfc1.d Rdest, FRsrc1` *Move Double From Coprocessor 1* †

Move the contents of floating point registers `FRsrc1` and `FRsrc1 + 1` to CPU registers `Rdest` and `Rdest + 1`.

`mtcz Rsrc, CPdest` *Move To Coprocessor z*

Move the contents of CPU `Rsrc` to coprocessor z's register `CPdest`.

Floating Point Operations

mips Floating Point Instructions

- Floating point coprocessor 1 operates on single (32-bit) and double precision (64-bit) FP numbers.
- 32 32-bit registers `$f0`–`$f31`.
- Two FP registers to hold doubles.
- FP operations only use even-numbered registers including instructions that operate on single floats.
- Values are moved one word (32-bits) at a time by `lwc1`, `swc1`, `mtc1`, and `mfc1` or by the `l.s`, `l.d`, `s.s`, and `s.d` pseudo-instructions.
- The flag set by FP comparison operations is read by the CPU with its `bc1t` and `bc1f` instructions.

Floating Point: Arithmetics

Compute the $*$ of the floating float doubles (singles) in `FRsrc1` and `FRsrc2` and put it in `FRdest`.

`add.d FRdest, FRsrc1, FRsrc2`	*Floating Point Addition Double*
`add.s FRdest, FRsrc1, FRsrc2`	*Floating Point Addition Single*
`div.d FRdest, FRsrc1, FRsrc2`	*Floating Point Divide Double*
`div.s FRdest, FRsrc1, FRsrc2`	*Floating Point Divide Single*
`mul.d FRdest, FRsrc1, FRsrc2`	*Floating Point Multiply Double*
`mul.s FRdest, FRsrc1, FRsrc2`	*Floating Point Multiply Single*
`sub.d FRdest, FRsrc1, FRsrc2`	*Floating Point Subtract Double*
`sub.s FRdest, FRsrc1, FRsrc2`	*Floating Point Subtract Single*
`abs.d FRdest, FRsrc`	*Floating Point Absolute Value Double*
`abs.s FRdest, FRsrc`	*Floating Point Absolute Value Single*
`neg.d FRdest, FRsrc`	*Negate Double*
`neg.s FRdest, FRsrc`	*Negate Single*

Floating Point: Comparison

Compare the floating point double in `FRsrc1` against the one in `FRsrc2` and set the floating point condition flag true if they are $*$.

`c.eq.d FRsrc1, FRsrc2`	*Compare Equal Double*
`c.eq.s FRsrc1, FRsrc2`	*Compare Equal Single*
`c.le.d FRsrc1, FRsrc2`	*Compare Less Than Equal Double*
`c.le.s FRsrc1, FRsrc2`	*Compare Less Than Equal Single*
`c.lt.d FRsrc1, FRsrc2`	*Compare Less Than Double*
`c.lt.s FRsrc1, FRsrc2`	*Compare Less Than Single*

Floating Point: Conversions

Convert between (i) single, (ii) double precision floating point number or (iii) integer in `FRsrc` to `FRdest`.

`cvt.d.s FRdest, FRsrc`	*Convert Single to Double*
`cvt.d.w FRdest, FRsrc`	*Convert Integer to Double*
`cvt.s.d FRdest, FRsrc`	*Convert Double to Single*
`cvt.s.w FRdest, FRsrc`	*Convert Integer to Single*
`cvt.w.d FRdest, FRsrc`	*Convert Double to Integer*
`cvt.w.s FRdest, FRsrc`	*Convert Single to Integer*

Floating Point: Moves

`l.d FRdest, address` *Load Floating Point Double* †
`l.s FRdest, address` *Load Floating Point Single* †
Load the floating float double (single) at `address` into register `FRdest`.

`mov.d FRdest, FRsrc` *Move Floating Point Double*
`mov.s FRdest, FRsrc` *Move Floating Point Single*
Move the floating float double (single) from `FRsrc` to `FRdest`.

`s.d FRdest, address` *Store Floating Point Double* †
`s.s FRdest, address` *Store Floating Point Single* †
Store the floating float double (single) in `FRdest` at `address`.

The EPITA Tiger Compiler

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (Ultrix: 1 mips)
  - Meaningless Indication of Processor Speed
    - Meaningless Information Provided by Salesmen
    - Meaningless Information per Second
    - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (10 mips, 1 mip)
  - Meaningless Indication of Processor Speed
    - Meaningless Information Provided by Salesmen
    - Meaningless Information per Second
    - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (10 mips, 1 mip)
  - Meaningless Indication of Processor Speed
  - Meaningless Information Provided by Salesmen
  - Meaningless Information per Second
  - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (10 mips, 1 mip)
  - Meaningless Indication of Processor Speed
  - Meaningless Information Provided by Salesmen
  - Meaningless Information per Second
  - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (10 mips, 1 mip)
  - Meaningless Indication of Processor Speed
  - Meaningless Information Provided by Salesmen
  - Meaningless Information per Second
  - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (10 mips, 1 mip)
  - Meaningless Indication of Processor Speed
  - Meaningless Information Provided by Salesmen
  - Meaningless Information per Second
  - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (10 mips, 1 mip)
  - Meaningless Indication of Processor Speed
  - Meaningless Information Provided by Salesmen
  - Meaningless Information per Second
  - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (10 mips, 1 mip)
  - Meaningless Indication of Processor Speed
  - Meaningless Information Provided by Salesmen
  - Meaningless Information per Second
  - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

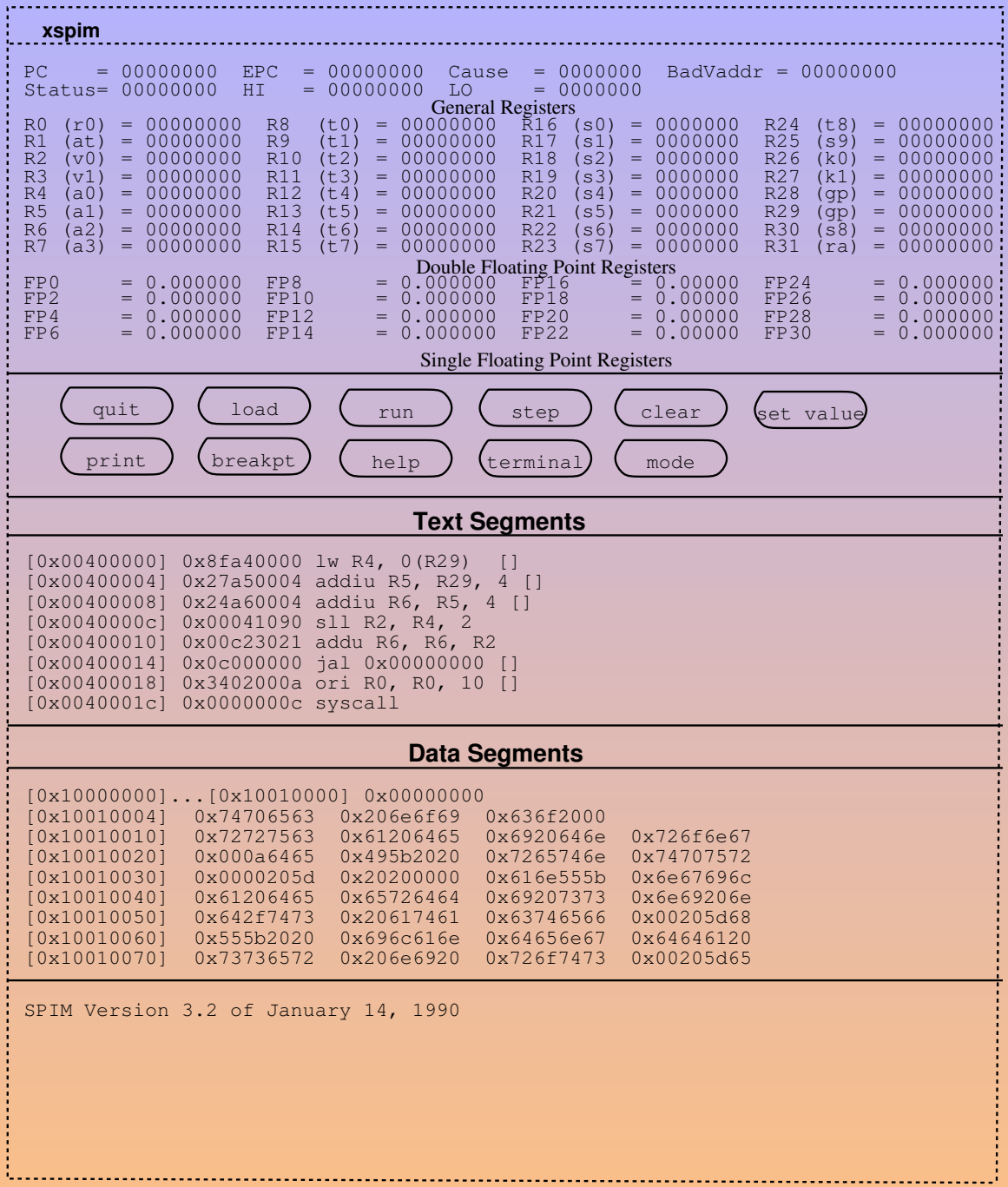
We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (10 mips, 1 mip)
  - Meaningless Indication of Processor Speed
  - Meaningless Information Provided by Salesmen
  - Meaningless Information per Second
  - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!

The EPITA Tiger Project

We aim at mips because:

- mips is a nice assembly language
- mips is more modern
- mips is meaningful:
  - Million Instructions Per Second (10 mips, 1 mip)
  - Meaningless Indication of Processor Speed
  - Meaningless Information Provided by Salesmen
  - Meaningless Information per Second
  - Microprocessor without Interlocked Piped Stages
- spim is a portable mips emulator
- spim has a cool modern gui, xspim!


Register Display
Control Buttons
User and Kernel Text Segments
Data and Stack Segments
SPIM Messages
xspim
PC = 00000000 EPC = 00000000 Cause = 0000000 BadVaddr = 00000000
Status= 00000000 HI = 00000000 LO = 0000000
General Registers
R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 0000000 R24 (t8) = 00000000
R1 (at) = 00000000 R9 (t1) = 00000000 R17 (s1) = 0000000 R25 (s9) = 00000000
R2 (v0) = 00000000 R10 (t2) = 00000000 R18 (s2) = 0000000 R26 (k0) = 00000000
R3 (v1) = 00000000 R11 (t3) = 00000000 R19 (s3) = 0000000 R27 (k1) = 00000000
R4 (a0) = 00000000 R12 (t4) = 00000000 R20 (s4) = 0000000 R28 (gp) = 00000000
R5 (a1) = 00000000 R13 (t5) = 00000000 R21 (s5) = 0000000 R29 (gp) = 00000000
R6 (a2) = 00000000 R14 (t6) = 00000000 R22 (s6) = 0000000 R30 (s8) = 00000000
R7 (a3) = 00000000 R15 (t7) = 00000000 R23 (s7) = 0000000 R31 (ra) = 00000000
Double Floating Point Registers
FP0 = 0.000000 FP8 = 0.000000 FP16 = 0.00000 FP24 = 0.000000
FP2 = 0.000000 FP10 = 0.000000 FP18 = 0.00000 FP26 = 0.000000
FP4 = 0.000000 FP12 = 0.000000 FP20 = 0.00000 FP28 = 0.000000
FP6 = 0.000000 FP14 = 0.000000 FP22 = 0.00000 FP30 = 0.000000
Single Floating Point Registers
quit
load
run
step
clear
set value
print
breakpt
help
terminal
mode
Text Segments
[0x00400000] 0x8fa40000 lw R4, 0(R29) []
[0x00400004] 0x27a50004 addiu R5, R29, 4 []
[0x00400008] 0x24a60004 addiu R6, R5, 4 []
[0x0040000c] 0x00041090 sll R2, R4, 2
[0x00400010] 0x00c23021 addu R6, R6, R2
[0x00400014] 0x0c000000 jal 0x00000000 []
[0x00400018] 0x3402000a ori R0, R0, 10 []
[0x0040001c] 0x0000000c syscall
Data Segments
[0x10000000]...[0x10010000] 0x00000000
[0x10010004] 0x74706563 0x206e6f69 0x636f2000
[0x10010010] 0x72727563 0x61206465 0x6920646e 0x726f6e67
[0x10010020] 0x000a6465 0x495b2020 0x7265746e 0x74707572
[0x10010030] 0x0000205d 0x20200000 0x616e555b 0x6e67696c
[0x10010040] 0x61206465 0x65726464 0x69207373 0x6e69206e
[0x10010050] 0x642f7473 0x20617461 0x63746566 0x00205d68
[0x10010060] 0x555b2020 0x696c616e 0x64656e67 0x64646120
[0x10010070] 0x73736572 0x206e6920 0x726f7473 0x00205d65
SPIM Version 3.2 of January 14, 1990

A Sample: fact

```
/* Define a recursive function.  */
let
  /* Calculate n! */
  function fact (n : int) : int =
    if n = 0
      then 1
      else n * fact (n - 1)
in
  print_int (fact (10));
  print ("\n")
end
```

```
# Routine: fact
l0:     sw      $fp, -8 ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 16
        sw      $ra, -12 ($fp)
        sw      $a0, ($fp)
        sw      $a1, -4 ($fp)
l5:     lw      $t0, -4 ($fp)
        beq     $t0, 0, l1
l2:     lw      $a0, ($fp)
        lw      $t0, -4 ($fp)
        sub     $a1, $t0, 1
        jal     l0
        lw      $t0, -4 ($fp)
        mul     $t0, $t0, $v0
l3:     move    $v0, $t0
        j       l6
l1:     li      $t0, 1
        j       l3
l6:     lw      $ra, -12 ($fp)
        move    $sp, $fp
        lw      $fp, -8 ($fp)
        jr      $ra
```

```
.data
l4:
        .word 1
        .asciiz "\n"
.text
# Routine: Main
t_main: sw      $fp, ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, -4 ($fp)
l7:     move    $a0, $fp
        li      $a1, 10
        jal     l0
        move    $a0, $v0
        jal     print_int
        la      $a0, l4
        jal     print
l8:     lw      $ra, -4 ($fp)
        move    $sp, $fp
        lw      $fp, ($fp)
        jr      $ra
```

Nolimips (formerly Mipsy)

- Another mips emulator
- Interactive loop
- Unlimited number of `$x42` registers!

```
# Routine: fact
l0:     sw      $a0, ($fp)
        sw      $a1, -4 ($fp)
        move    $x11, $s0
        move    $x12, $s1
        move    $x13, $s2
        move    $x14, $s3
        move    $x15, $s4
        move    $x16, $s5
        move    $x17, $s6
        move    $x18, $s7
l5:     lw      $x5, -4 ($fp)
        beq     $x5, 0, l1
l2:     lw      $x6, ($fp)
        move    $a0, $x6
        lw      $x8, -4 ($fp)
        sub     $x7, $x8, 1
        move    $a1, $x7
        jal     l0
        move    $x3, $v0
        lw      $x10, -4 ($fp)
        mul     $x9, $x10, $x3
        move    $x0, $x9
l3:     move    $v0, $x0
        j       l6
l1:     li      $x0, 1
        j       l3
l6:     move    $s0, $x11
        move    $s1, $x12
        move    $s2, $x13
        move    $s3, $x14
        move    $s4, $x15
        move    $s5, $x16
        move    $s6, $x17
        move    $s7, $x18
```

```
# Routine: fact
l0:     sw      $fp, -8 ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 16
        sw      $ra, -12 ($fp)
        sw      $a0, ($fp)
        sw      $a1, -4 ($fp)
l5:     lw      $t0, -4 ($fp)
        beq     $t0, 0, l1
l2:     lw      $a0, ($fp)
        lw      $t0, -4 ($fp)
        sub     $a1, $t0, 1
        jal     l0
        lw      $t0, -4 ($fp)
        mul     $t0, $t0, $v0
l3:     move    $v0, $t0
        j       l6
l1:     li      $t0, 1
        j       l3
l6:     lw      $ra, -12 ($fp)
        move    $sp, $fp
        lw      $fp, -8 ($fp)
        jr      $ra
```

Instruction Selection

Nolimips (formerly Mipsy)

- Another mips emulator
- Interactive loop
- Unlimited number of `$x42` registers!

```
# Routine: fact
l0:     sw      $a0, ($fp)
        sw      $a1, -4 ($fp)
        move    $x11, $s0
        move    $x12, $s1
        move    $x13, $s2
        move    $x14, $s3
        move    $x15, $s4
        move    $x16, $s5
        move    $x17, $s6
        move    $x18, $s7
l5:     lw      $x5, -4 ($fp)
        beq     $x5, 0, l1
l2:     lw      $x6, ($fp)
        move    $a0, $x6
        lw      $x8, -4 ($fp)
        sub     $x7, $x8, 1
        move    $a1, $x7
        jal     l0
        move    $x3, $v0
        lw      $x10, -4 ($fp)
        mul     $x9, $x10, $x3
        move    $x0, $x9
l3:     move    $v0, $x0
        j       l6
l1:     li      $x0, 1
        j       l3
l6:     move    $s0, $x11
        move    $s1, $x12
        move    $s2, $x13
        move    $s3, $x14
        move    $s4, $x15
        move    $s5, $x16
        move    $s6, $x17
        move    $s7, $x18
```

```
# Routine: fact
l0:     sw      $fp, -8 ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 16
        sw      $ra, -12 ($fp)
        sw      $a0, ($fp)
        sw      $a1, -4 ($fp)
l5:     lw      $t0, -4 ($fp)
        beq     $t0, 0, l1
l2:     lw      $a0, ($fp)
        lw      $t0, -4 ($fp)
        sub     $a1, $t0, 1
        jal     l0
        lw      $t0, -4 ($fp)
        mul     $t0, $t0, $v0
l3:     move    $v0, $t0
        j       l6
l1:     li      $t0, 1
        j       l3
l6:     lw      $ra, -12 ($fp)
        move    $sp, $fp
        lw      $fp, -8 ($fp)
        jr      $ra
```

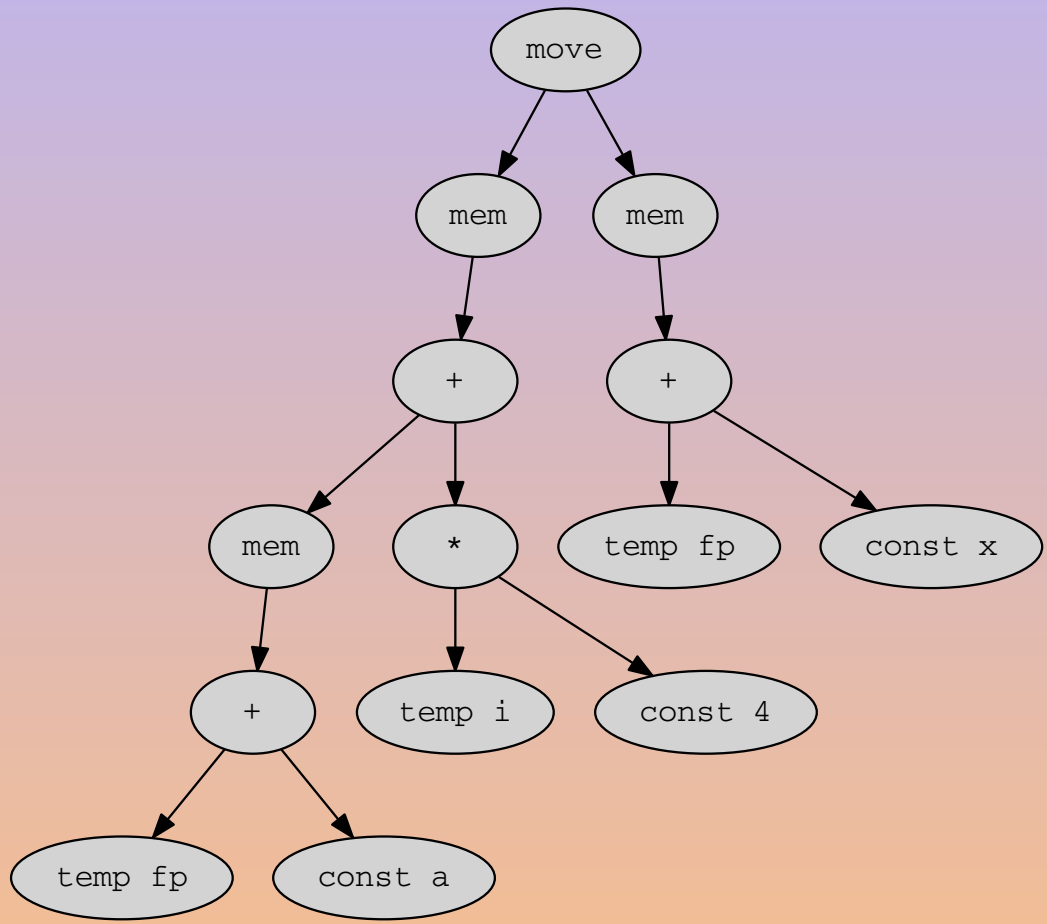
Instruction Selection

Translating a Simple Instruction

How would you translate

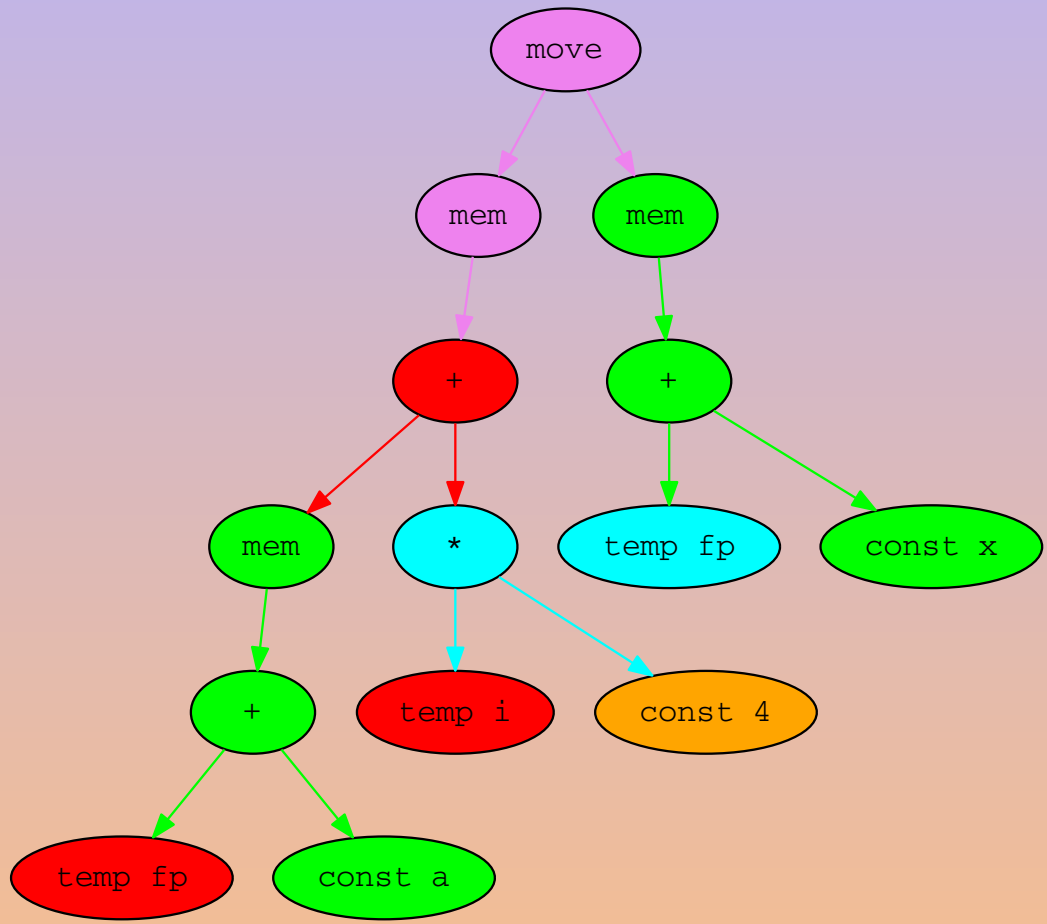
```
a[i] := x
```

where `x` is frame resident, and `i` is not? [Appel, 1998]



Simple Instruction: Translation 1

```
load  t17 <- M[fp + a]
addi  t18 <- r0 + 4
mul   t19 <- ti * t18
add   t20 <- t17 + t19
load  t21 <- M[fp + x]
store M[t20 + 0] <- t21
```



Tree Patterns

- Translation from Tree to Assembly corresponds to *parsing a tree*.
- Looking for a covering of the tree, using tiles.
- The set of tiles corresponds to the instruction set.

Tree Patterns

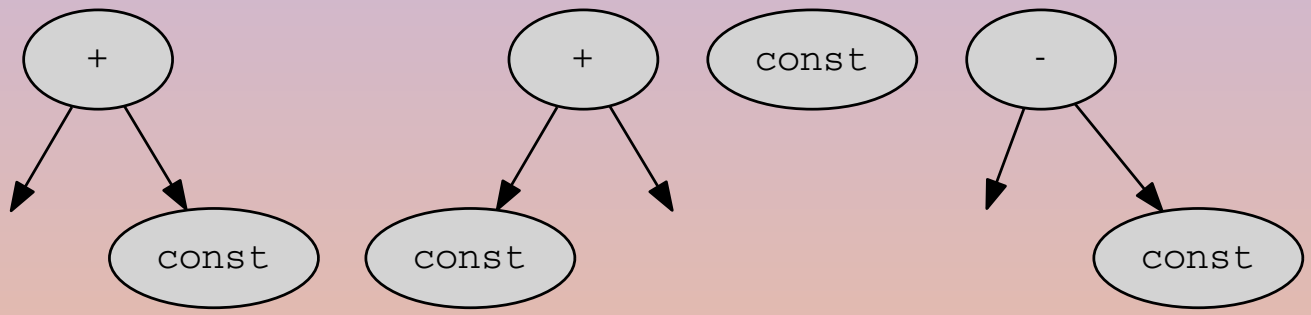
- Translation from Tree to Assembly corresponds to *parsing a tree*.
- Looking for a covering of the tree, using tiles.
- The set of tiles corresponds to the instruction set.

Tree Patterns

- Translation from Tree to Assembly corresponds to *parsing a tree*.
- Looking for a covering of the tree, using tiles.
- The set of tiles corresponds to the instruction set.

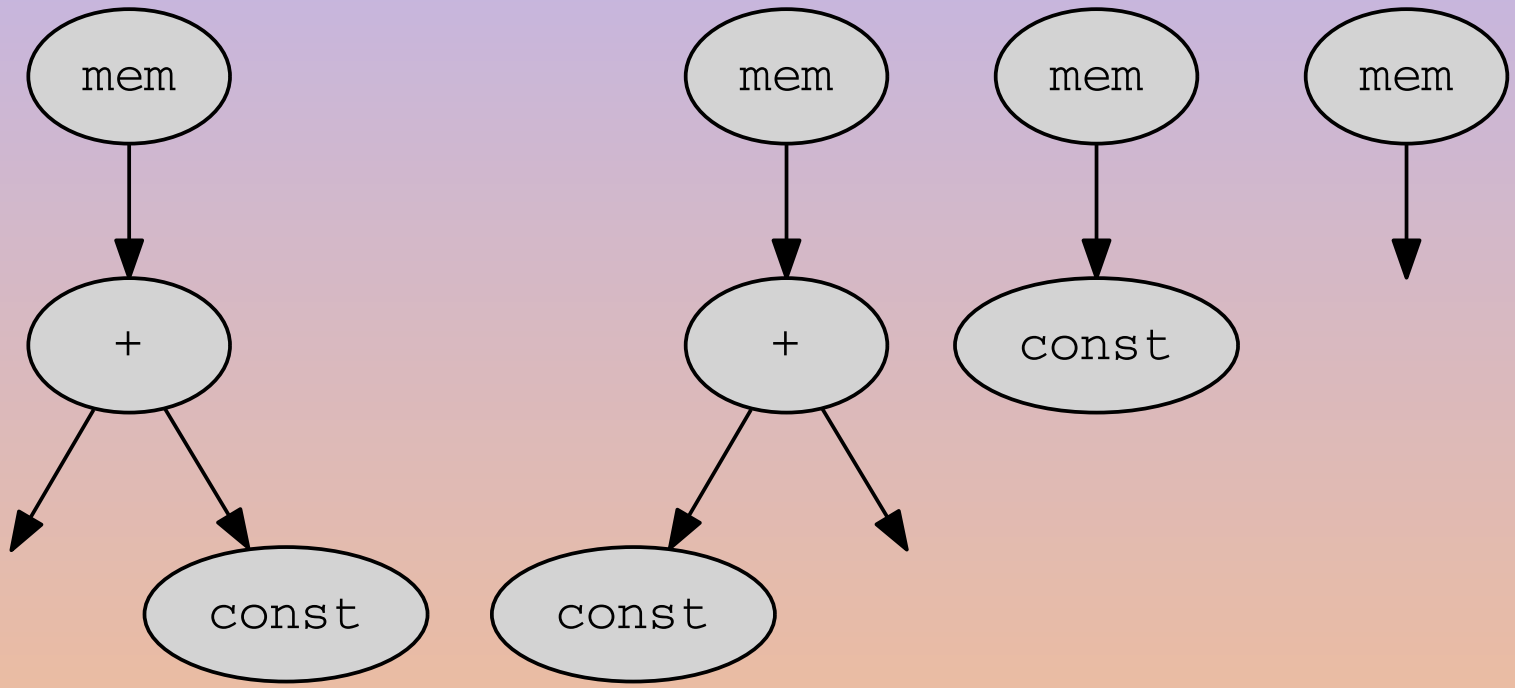
Tiles

Missing nodes are plugs for *temporaries*: tiles read from temps, and create temps.

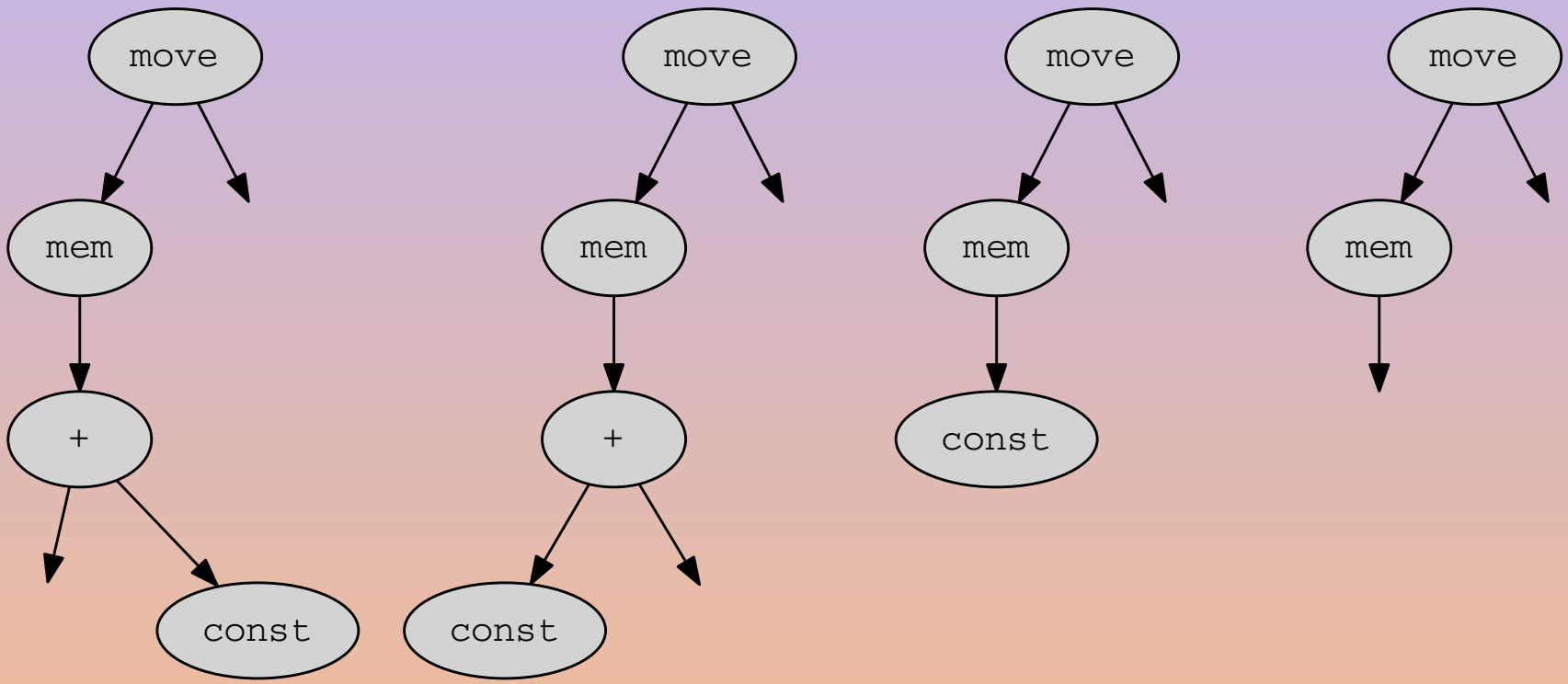


Some architectures rely on a special register to produce 0.

Tiles: Loading $\texttt{load}\ r_i \leftarrow M[r_j + c]$

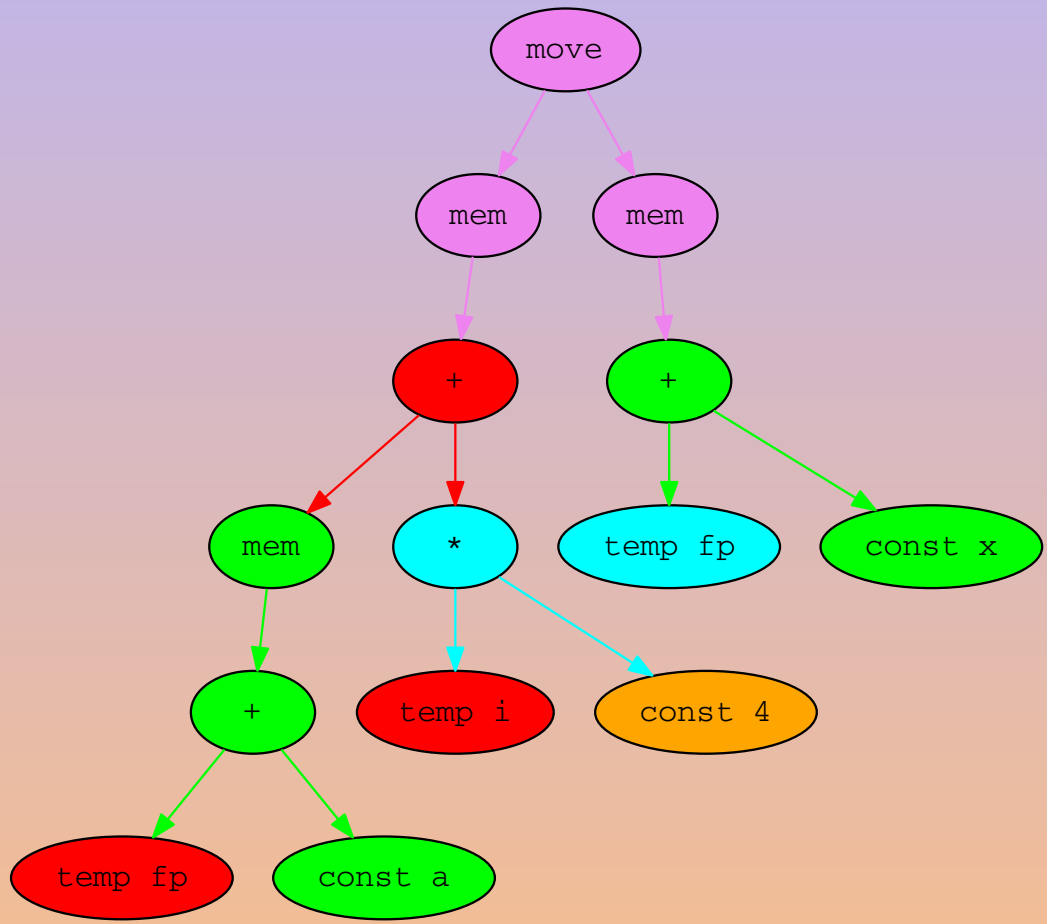


Tiles: Storing store $M[r_j + c] \leftarrow r_i$



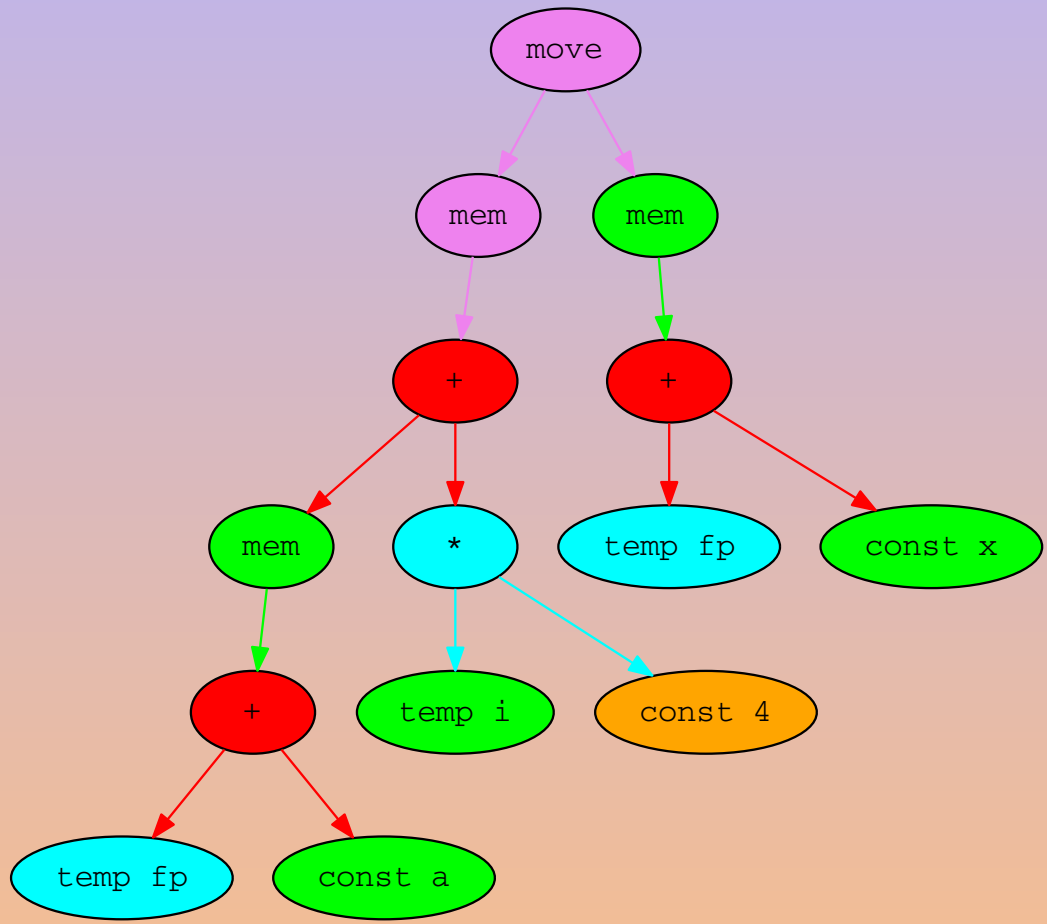
Simple Instruction: Translation 2

```
load  t17 <- M[fp + a]
addi  t18 <- r0 + 4
mul   t19 <- ti * t18
add   t20 <- t17 + t19
addi  t21 <- fp + x
movem M[t20] <- M[t21]
```



Simple Instruction: Translation 3

```
addi  t17 <- r0 + a
add   t18 <- fp + t17
load  t19 <- M[t18 + 0]
addi  t20 <- r0 + 4
mul   t21 <- ti * t20
add   t22 <- t19 + t21
addi  t23 <- r0 + x
add   t24 <- fp + t23
load  t25 <- M[t24 + 0]
store M[t22 + 0] <- t25
```



Translating a Simple Instruction

- There is always a solution
  (provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
  - some are locally better, *optimum coverings*
    (no fusion can reduce the cost),
    - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution
  (provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
  - some are locally better, *optimal coverings*
    (no fusion can reduce the cost),
  - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution
  (provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
  - some are locally better, *optimal coverings*
    (no fusion can reduce the cost),
  - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution
  (provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
  - some are locally better, *optimal coverings*
    (no fusion can reduce the cost),
  - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution
  (provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
  - some are locally better, *optimal coverings*
    (no fusion can reduce the cost),
  - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Translating a Simple Instruction

- There is always a solution
  (provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
  - some are locally better, *optimal coverings*
    (no fusion can reduce the cost),
  - some are globally better, *optimum coverings*.

Nowadays this approach is too naive:

- cpus are really layers of units that work in parallel.
- Costs are therefore interrelated.

Algorithms for Instruction Selection

Maximal Munch Find an optimal tiling.

- Top-down strategy.
- Cover the current node with the largest tile.
- Repeat on subtrees.
- Generate instructions in reverse-order after tile placement.

Dynamic Programming Find an optimum tiling.

- Bottom-up strategy.
- Assign cost to each node.
- Cost = cost of selected tile + cost of subtrees.
- Select a tile with minimal cost and recurse upward.
- Implemented by code generator generators (Twig, Burg, iBurg, MonoBURG, ...).

Algorithms for Instruction Selection

Maximal Munch Find an optimal tiling.

- Top-down strategy.
- Cover the current node with the largest tile.
- Repeat on subtrees.
- Generate instructions in reverse-order after tile placement.

Dynamic Programming Find an optimum tiling.

- Bottom-up strategy.
- Assign cost to each node.
- Cost = cost of selected tile + cost of subtrees.
- Select a tile with minimal cost and recurse upward.
- Implemented by code generator generators (Twig, Burg, iBurg, MonoBURG, ...).

Tree Matching

- The basic operation is the *pattern matching*.
- Not all the languages stand equal before pattern matching. . .

Tree Matching

- The basic operation is the *pattern matching*.
- Not all the languages stand equal before pattern matching. . .

... in Stratego

```
Select-swri :
  MOVE(MEM(BINOP(PLUS, e1, CONST(n))), e2) →
  SEQ(MOVE(r2, e2), SEQ(MOVE(r1, e1), sw-ri(r2, r1, n)))
  where <new-atemp> e1 ⇒  r1; <new-atemp> e2 ⇒  r2

Select-swr :
  MOVE(MEM(e1), e2) →  SEQ(MOVE(r2, e2), SEQ(MOVE(r1, e1), sw-r(r2, r1)))
  where <new-atemp> e1 ⇒  r1; <new-atemp> e2 ⇒  r2

Select-nop :
  MOVE(TEMP(r), TEMP(r)) →  NUL
Select-nop :
  MOVE(REG(r), REG(r)) →  NUL

Select-mover :
  MOVE(TEMP(r), TEMP(t)) →  move(TEMP(r), TEMP(t))  where <not(eq)> (r, t)
Select-mover :
  MOVE(TEMP(r), REG(t)) →  move(TEMP(r), REG(t))    where <not(eq)> (r, t)
Select-mover :
  MOVE(REG(r), TEMP(t)) →  move(REG(r), TEMP(t))    where <not(eq)> (r, t)
Select-mover :
  MOVE(REG(r), REG(t)) →  move(REG(r), REG(t))      where <not(eq)> (r, t)
```

... in Haskell: `Ir.hs` [Anisko, 2003]

```
module Ir (Stm (Move, Sxp, Jump, CJump, Seq, Label,
                LabelEnd, Literal),
           ...)
where

data Stm a =
    Move { ma :: a, lval :: Exp a, rval :: Exp a }
  | Sxp a (Exp a)
  | Jump a (Exp a)
  | CJump { cja :: a,
            rop :: Relop, cleft :: Exp a, cright :: Exp a,
            iftrue :: Exp a, iffalse :: Exp a }
  | Seq a [Stm a]
  | Label { la :: a,
            name :: String, size :: Int }
  | LabelEnd a
  | Literal { lita :: a,
              litname :: String, litcontent :: [Int] }
```

... in Haskell `Eval.hs` [Anisko, 2003]

```
module Eval (evalStm, ...)
where
import Ir
import Monad (Mnd, rfetch, rstore, rpush, rpop, ...)
import Result (Res (IntRes, UnitRes))
import Profile (profileExp, profileStm)

evalStm :: Stm Loc -> Mnd ()
evalStm stm@(Move loc (Temp _ t) e) =
    do (IntRes r) <- evalExp e
       verbose loc ["move", "(", "temp", t, ")", show r]
       profileStm stm
       rstore t r

evalStm stm@(Move loc (Mem _ e) f) =
    do (IntRes r) <- evalExp e
       (IntRes s) <- evalExp f
       verbose loc ["move", "(", "mem", show r, ")", show s]
       profileStm stm
       mstore r s
```

... in Haskell Low.hs [Anisko, 2003]

```
module Low (lowExp, lowStms)
where import ...

lowStms :: Int -> [Stm Ann] -> Mnd Bool
lowStms _ [] = return True

lowStms level
        ((CJump _ _ e f _ (Name _ s)) : (Label _ s' _) : stms)
        | s == s' =
    do a <- lowExp (level + 1) e
       b <- lowExp (level + 1) f
       c <- lowStms level stms
       return $ a && b && c

lowStms level (CJump l _ e f _ _ : stms) =
    do awarn l ["invalid cjump"]
       lowExp (level + 1) e
       lowExp (level + 1) f
       lowStms level stms
       return False
```

... in Haskell High.hs [Anisko, 2003]

```
module High (highExp, highStms)
where import ...

highStms :: Int -> [Stm Ann] -> Mnd Bool
highStms level ss =
    do a <- sequence $ map (highStm level) ss
       return (and a)

highStm :: Int -> Stm Ann -> Mnd Bool
highStm level (Move l dest src) =
    do a  <- highExp (level + 1) dest
       a' <- case dest of
               Temp _ _ -> return True
               Mem _ _  -> return True
               _        -> do awarn (annExp dest)
                                    ["invalid move destination:",
                                     show dest]
                              return False
       b <- highExp (level + 1) src
       return $ a && a' && b
```

... in C++

```
52 lines matching "switch\\|case\\|default\\|//" in buffer codegen.cc.
 28:switch (stm.kind_get ())
 30:  case Tree::move_kind :
 36:      switch (dst->kind_get ())
 38:        case Tree::mem_kind : // dst
 41:            // MOVE (MEM (...), ...)
 42:            switch (src.kind_get ())
 44:              // MOVE (MEM (...), MEM (...))
 45:              case Tree::mem_kind : // src
 55:              default : // src
 57:                  // MOVE (MEM (...) , e1)
 59:                  switch (addr->kind_get ())
 61:                    case Tree::binop_kind : // addr
 63:                        // MOVE (MEM (BINOP (..., ..., ...)) , e1)
 69:                        switch (binop.oper_get ())
 71:                          case Binop::minus:
 73:                          case Binop::plus:
 74:                            // MOVE (MEM (BINOP (+/-, e1, CONST (i))),
 77:                            // MOVE (MEM (BINOP (+/-, CONST (i), e1)) ,
 87:                          default:
 88:                            // MOVE (MEM (BINOP (...)) , e1)
```

... in C++

```
case Node::move_kind :
{
  DOWN_CAST (Move,  move, stm);
  const Exp* dst = move.dst_get (); const Exp* src = move.src_get ();
  switch (dst->kind_get ()) {
    case Node::mem_kind : { // dst
      DOWN_CAST (Mem, mem, *dst);
      // MOVE (MEM (...), ...)
      switch (src.kind_get ()) {
        // MOVE (MEM (...), MEM (...))
      case Node::mem_kind : // src
        ...
      default : { // src
        // MOVE (MEM (...) , e1)
        const Exp* addr = dst.exp_get ();
        switch (addr->kind_get ()) {
        case Node::binop_kind : { // addr
          // MOVE (MEM (BINOP (..., ..., ...)) , e1)
          DOWN_CAST (Binop, binop, *addr);
          const Exp* binop_left = binop.left_get ();
          const Exp* binop_right = binop.right_get ();
          short sign = 1;
          switch (binop.oper_get ()) {
          case Binop::minus:  sign = -1;
          case Binop::plus:
            // MOVE (MEM (BINOP (+/-, e1, CONST (i))), e2)
            if (binop_right->kind_get () == Node::const_kind)
              std::swap (binop_left, binop_right);
            // MOVE (MEM (BINOP (+/-, CONST (i), e1)) , e2)
            if (binop_left->kind_get () == Node::const_kind) {
              DOWN_CAST (Const, const_left, *binop_left);
              emit (_assembly->store_build (munchExp (src),
                                            munchExp (* binop_right),
```

... in C++

Break down long switches into smaller functions.

Twig, Burg, iBurg [Fraser et al., 1992]

```
%{ /* ... */
enum { ADDI=309, ADDRLP=295, ASGNI=53, CNSTI=21, CVCI=85,
 I0I=661, INDIRC=67 };
/* ... */
%}
%term ADDI=309 ADDRLP=295 ASGNI=53
%term CNSTI=21 CVCI=85 I0I=661 INDIRC=67
%%
/* ... */
```

Twig, Burg, iBurg [Fraser et al., 1992]

```
/* ... */
%%
stmt:   ASGNI(disp,reg) = 4 (1);
stmt:   reg = 5;
reg:    ADDI(reg,rc) = 6 (1);
reg:    CVCI(INDIRC(disp)) = 7 (1);
reg:    IOI = 8;
reg:    disp = 9 (1);
disp:   ADDI(reg,con) = 10;
disp:   ADDRLP = 11;
rc:     con = 12;
rc:     reg = 13;
con:    CNSTI = 14;
con:    IOI = 15;
%%
/* ... */
```

MonoBURG

```
binop: Binop(lhs : exp, rhs : Const)
{
  auto binop = tree.cast<Binop>();
  auto cst = rhs.cast<Const>();
  EMIT(IA32_ASSEMBLY
       .binop_build(binop->oper_get(), lhs->asm_get(),
                    cst->value_get(), tree->asm_get()));
}

binop: Binop(lhs : exp, rhs : exp)
{
  auto binop = tree.cast<Binop>();
  EMIT(IA32_ASSEMBLY
       .binop_build(binop->oper_get(), lhs->asm_get(),
                    rhs->asm_get(), tree->asm_get()));
}
```

Bibliography I


Anisko, R. (2003).
Havm.
http://tiger.lrde.epita.fr/Havm.


Appel, A. W. (1998).
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press.


Fraser, C. W., Hanson, D. R., and Proebsting, T. A. (1992).
Engineering a simple, efficient code-generator generator.
ACM Letters on Programming Languages and Systems, 1(3):213–226.


Larus, J. R. (1990).
SPIM S20: A MIPS R2000 simulator.
Technical Report TR966, Computer Sciences Department, University of Wisconsin–Madison.

Liveness Analysis

Akim Demaille Étienne Renault Roland Levillain
first.*last*@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

May 19, 2018

Liveness Analysis

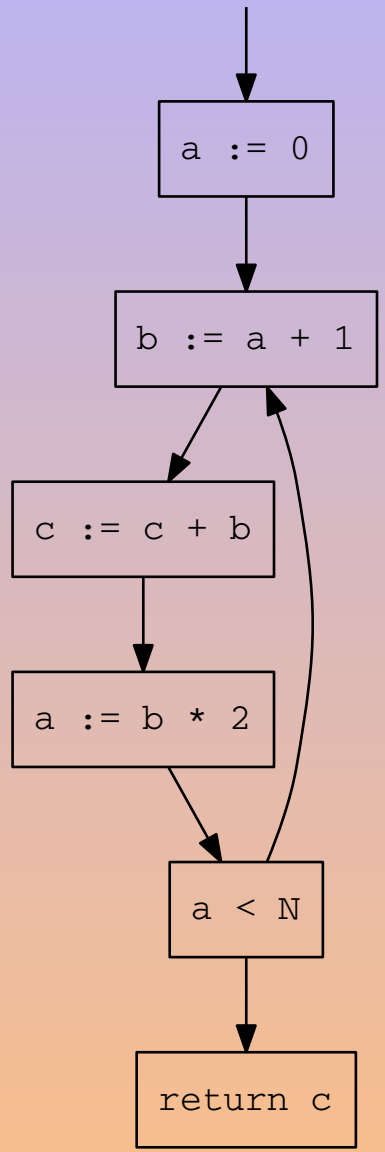
Control Flow Graph

Control Flow Graph [Appel, 1998]

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```

Control Flow Graph [Appel, 1998]

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```



7.tig

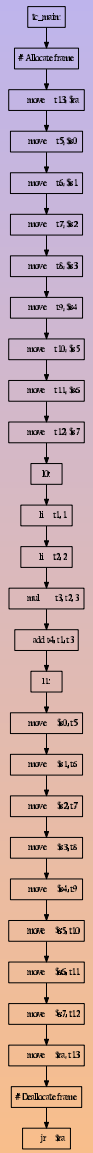
```
1 + 2 * 3
```

7's Pre-Assembly

```
tc_main:
# Allocate frame
        move    $x13, $ra
        move    $x5, $s0
        move    $x6, $s1
        move    $x7, $s2
        move    $x8, $s3
        move    $x9, $s4
        move    $x10, $s5
        move    $x11, $s6
        move    $x12, $s7
l0:
        li      $x1, 1
        li      $x2, 2
        mul     $x3, $x2, 3
        add     $x4, $x1, $x3
```

```
l1:
        move    $s0, $x5
        move    $s1, $x6
        move    $s2, $x7
        move    $s3, $x8
        move    $s4, $x9
        move    $s5, $x10
        move    $s6, $x11
        move    $s7, $x12
        move    $ra, $x13
# Deallocate frame
        jr      $ra
```

7's Flowgraph



7000.tig

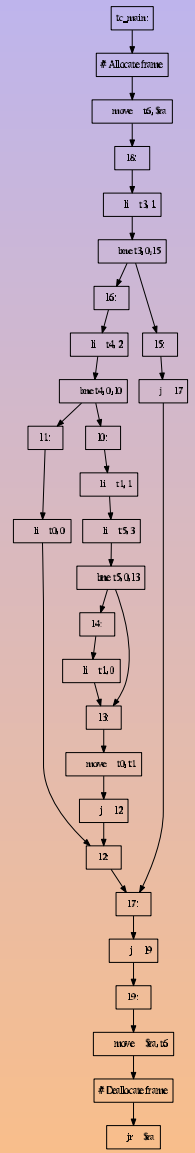
```
1 | 2 & 3
```

7000's Pre-Assembly

```
tc_main:
# Allocate frame
        move    $x6, $ra
18:
        li      $x3, 1
        bne     $x3, 0, 15
16:
        li      $x4, 2
        bne     $x4, 0, 10
11:
        li      $x0, 0
12:
17:
        j       19
```

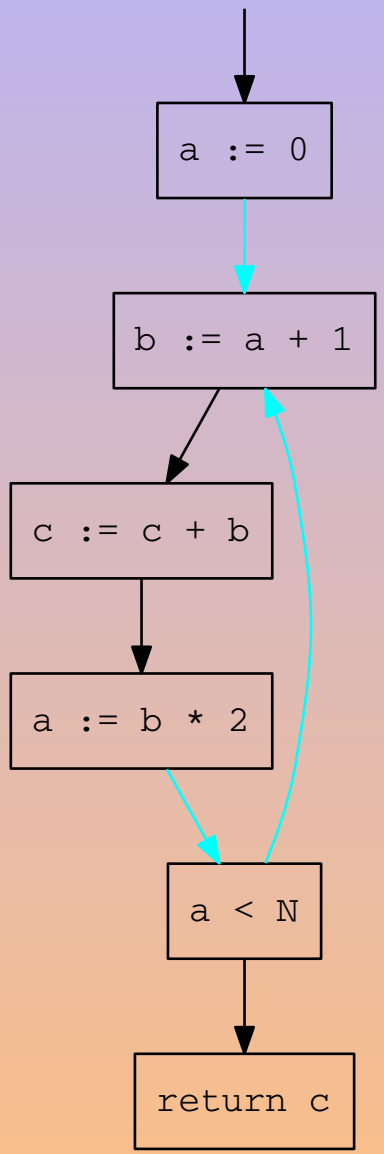
```
10:
        li      $x1, 1
        li      $x5, 3
        bne     $x5, 0, 13
14:
        li      $x1, 0
13:
        move    $x0, $x1
        j       12
15:
        j       17
19:
        move    $ra, $x6
# Deallocate frame
        jr      $ra
```

7000's Flowgraph

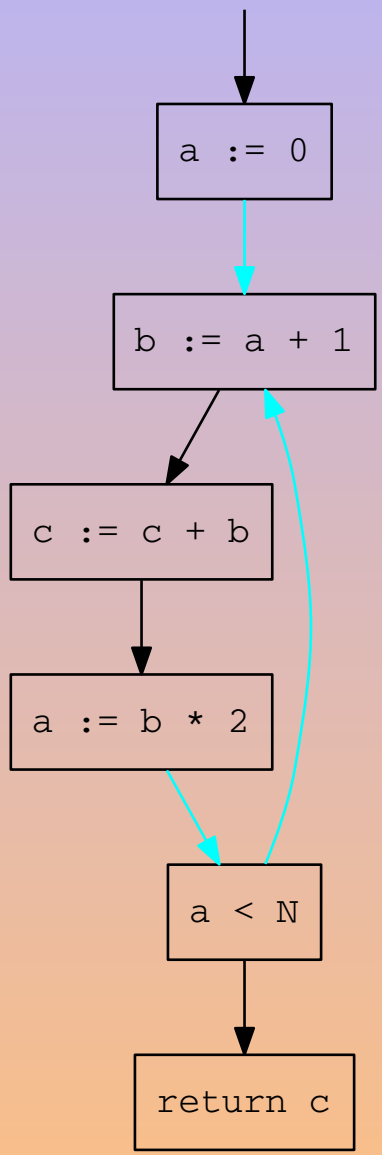


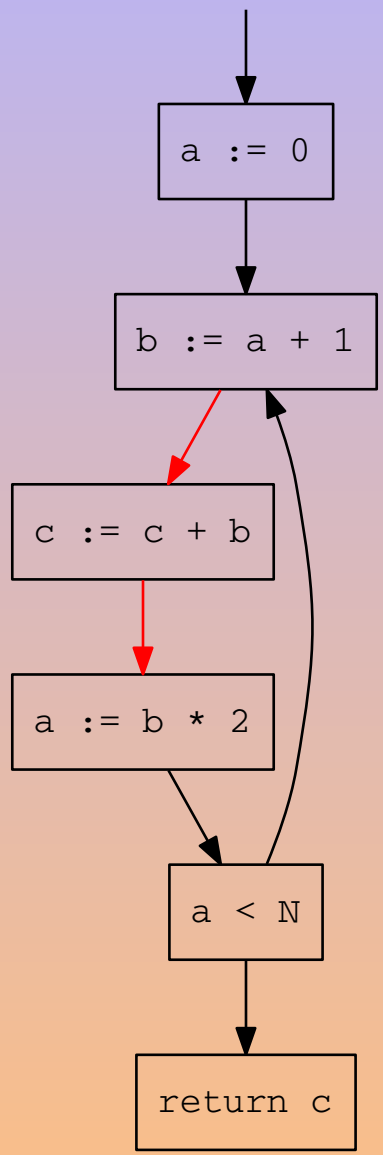
Liveness

Liveness

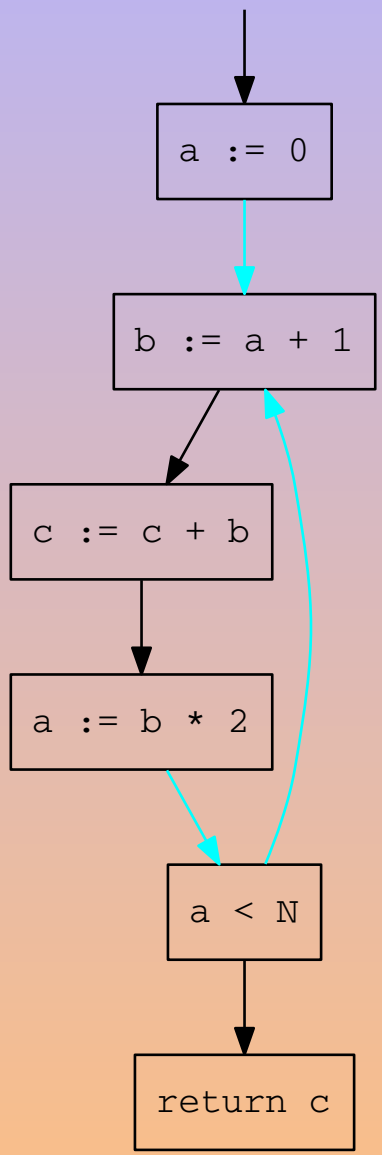


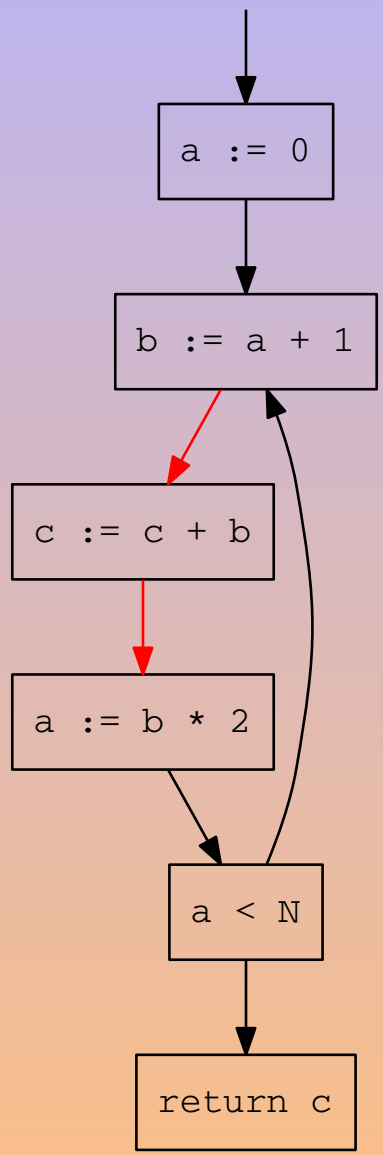
Liveness

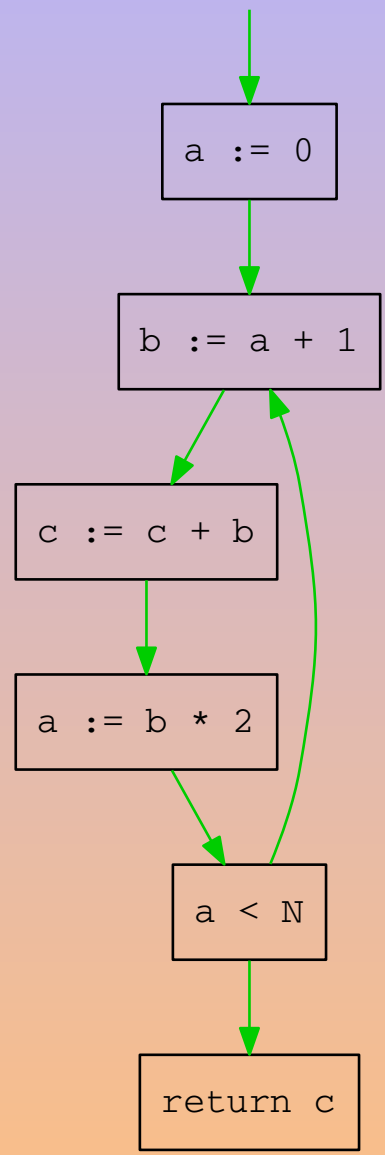




Liveness


a := 0
b := a + 1
c := c + b
a := b * 2
a < N
return c


a := 0
b := a + 1
c := c + b
a := b * 2
a < N
return c


a := 0
b := a + 1
c := c + b
a := b * 2
a < N
return c

Dataflow Equations for Liveness Analysis

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Liveness Calculation

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a								
2	a	b								
3	bc	c								
4	b	a								
5	a									
6	c									

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Liveness Calculation

1st step

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a								
2	a	b	a							
3	bc	c	bc							
4	b	a	b							
5	a		a	a						
6	c		c							

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Liveness Calculation

			1st step		2nd step					
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a				
2	a	b	a		a	bc				
3	bc	c	bc		bc	b				
4	b	a	b		b	a				
5	a		a	a	a	ac				
6	c		c		c					

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Liveness Calculation

			1st step		2nd step		3rd step			
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		
2	a	b	a		a	bc	ac	bc		
3	bc	c	bc		bc	b	bc	b		
4	b	a	b		b	a	b	a		
5	a		a	a	a	ac	ac	ac		
6	c		c		c		c			

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Liveness Calculation

			1st step		2nd step		3rd step		4th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Liveness Calculation

			1st step		2nd step		3rd step		4th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

			5th step					
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac				
2	a	b	ac	bc				
3	bc	c	bc	b				
4	b	a	bc	ac				
5	a		ac	ac				
6	c		c					

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Liveness Calculation

			1st step		2nd step		3rd step		4th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

			5th step		6th step			
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac	c	ac		
2	a	b	ac	bc	ac	bc		
3	bc	c	bc	b	bc	bc		
4	b	a	bc	ac	bc	ac		
5	a		ac	ac	ac	ac		
6	c		c		c			

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$$

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

Liveness Calculation

			1st step		2nd step		3rd step		4th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

			5th step		6th step		7th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac	c	ac	c	ac
2	a	b	ac	bc	ac	bc	ac	bc
3	bc	c	bc	b	bc	bc	bc	bc
4	b	a	bc	ac	bc	ac	bc	ac
5	a		ac	ac	ac	ac	ac	ac
6	c		c		c		c	

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$$

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

Liveness Calculation (Forward)

			1st step		2nd step		3rd step		4th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

			5th step		6th step		7th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac	c	ac	c	ac
2	a	b	ac	bc	ac	bc	ac	bc
3	bc	c	bc	b	bc	bc	bc	bc
4	b	a	bc	ac	bc	ac	bc	ac
5	a		ac	ac	ac	ac	ac	ac
6	c		c		c		c	

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$$

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

Calculation done following forward control-flow edges.

Liveness Calculation (Backward)

	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c							
5	a							
4	b	a						
3	bc	c						
2	a	b						
1		a						

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Calculation done following *reverse* control-flow edges.

Liveness Calculation (Backward)

	use	*def*	1st step *out*	1st step *in*	*out*	*in*	*out*	*in*
6	c			c				
5	a		c	ac				
4	b	a	ac	bc				
3	bc	c	bc	bc				
2	a	b	bc	ac				
1		a	ac	c				

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Calculation done following *reverse* control-flow edges.

Liveness Calculation (Backward)

			1st step		2nd step			
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		
5	a		c	ac	ac	ac		
4	b	a	ac	bc	ac	bc		
3	bc	c	bc	bc	bc	bc		
2	a	b	bc	ac	bc	ac		
1		a	ac	c	ac	c		

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Calculation done following *reverse* control-flow edges.

Liveness Calculation (Backward)

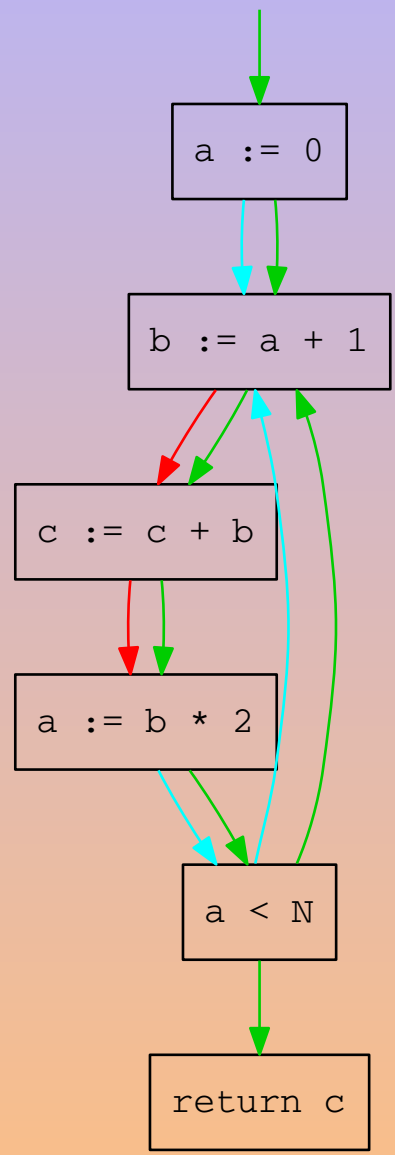
			1st step		2nd step		3rd step	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Calculation done following *reverse* control-flow edges.

Control Flow Graph [Appel, 1998]

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```



Conservative Approximation

Suppose d a variable not used in the fragment of code

Another Solution

	use	*def*	*out*	*in*
1		a		
2	a	b		
3	bc	c		
4	b	a		
5	a			
6	c			

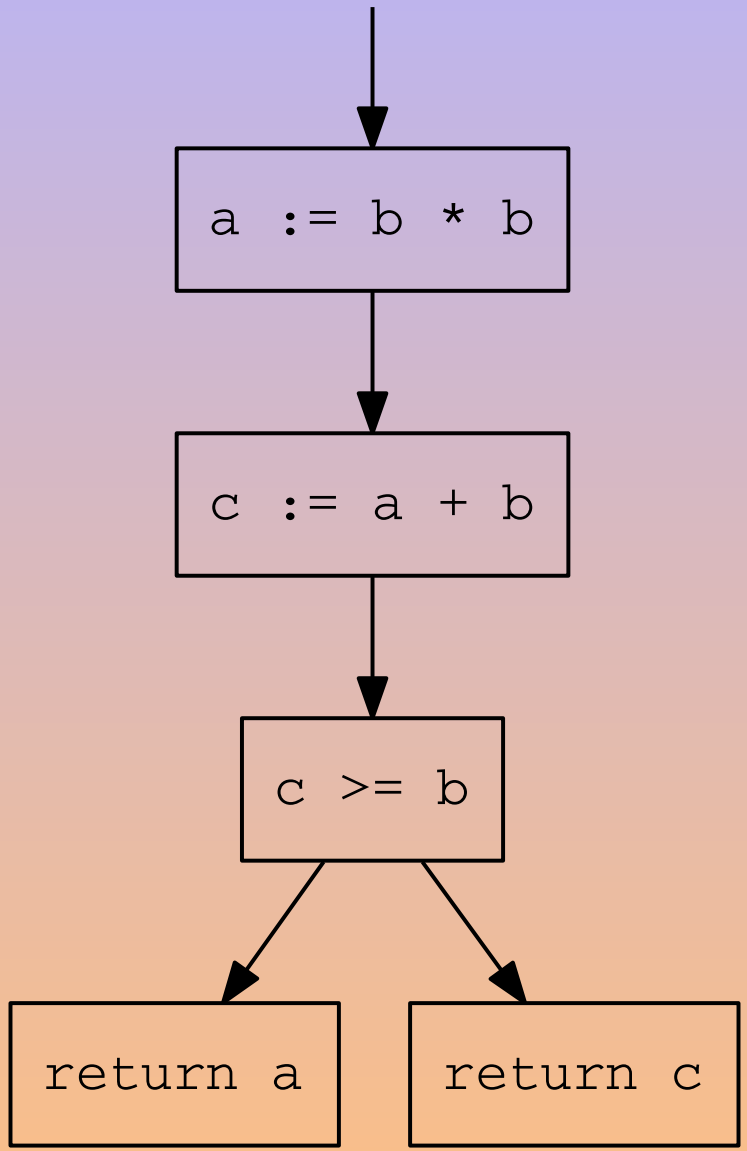
Conservative Approximation

Suppose d a variable not used in the fragment of code

Another Solution

	use	*def*	*out*	*in*
1		a	cd	acd
2	a	b	acd	bcd
3	bc	c	bcd	bcd
4	b	a	bcd	acd
5	a		acd	acd
6	c		c	

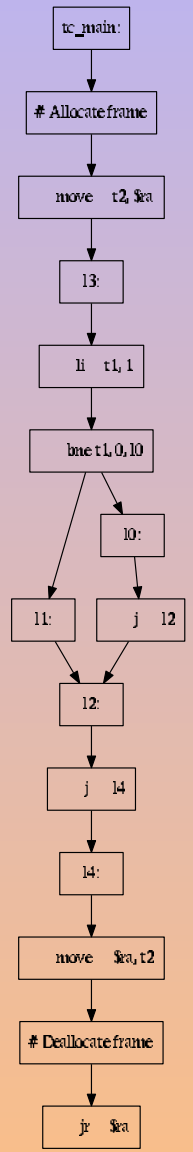
Conservative Approximation



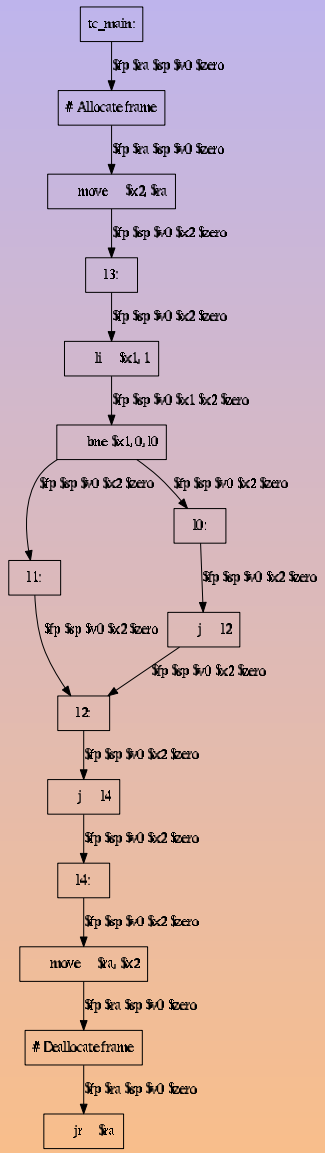
ors.tig

```
1 | 2
```

ors' Flowgraph



ors' Liveness Graph



Various Dataflow Analysis

Optimizing Compiler

- First step toward optimizing compilers
- How definitions and uses are related to each other
- What value a variable may have at a given point
- Constant propagation
- Common sub-expression elimination
- Copy propagation
- Dead Code Elimination

Constant propagation

An ambiguous definition is a statement that might or not assign a temporary t. For instance, a call may sometimes modifies t and sometimes not.

We don't have this problem for tiger due to excaping variables.

Don't loose optimisation! Consider every definiton as ambiguous

We need to define the set of definitions that reach the begining and the end of each node.

- *gen*: when enter this statement, we know that we will reach the end of it
- kills: any statement that invalidates a *gen*
- begin[n]: which statements can reach the begining of statement n
- end[n]: which statements can reach the end of statement n

Reaching definition [Appel, 1998]

```
    a := 5
    c := 1
L1: if c > a goto L2
    c := c + c
    goto L1
L2: a := c - a
    c := 0
```

	gen	*kills*	*begin*	*end*	*begin*	*end*	*begin*	*end*
1	1	6						
2	2	4,7						
3								
4	4	2,7						
5								
6	6	1						
7	7	2,4						

$$\text{end}[n] = \text{gen}[n] \cup (\text{begin}[n] \setminus \text{kills}[n])$$

$$\text{begin}[n] = \bigcup_{p \in \text{pred}[n]} \text{end}[p]$$

			1st step					
	gen	*kills*	*begin*	*end*	*begin*	*end*	*begin*	*end*
1	1	6		1				
2	2	4,7	1	1,2				
3			1,2	1,2				
4	4	2,7	1,2	1,4				
5			1,4	1,4				
6	6	1	1,2	2,6				
7	7	2,4	2,6	6,7				

$$\text{end}[n] = \text{gen}[n] \cup (\text{begin}[n] \setminus \text{kills}[n])$$

$$\text{begin}[n] = \bigcup_{p \in \text{pred}[n]} \text{end}[p]$$

			1st step		2nd step			
	gen	*kills*	*begin*	*end*	*begin*	*end*	*begin*	*end*
1	1	6		1		1		
2	2	4,7	1	1,2	1	1,2		
3			1,2	1,2	1,2,4	1,2,4		
4	4	2,7	1,2	1,4	1,2,4	1,4		
5			1,4	1,4	1,4	1,4		
6	6	1	1,2	2,6	1,2,4	2,4,6		
7	7	2,4	2,6	6,7	2,4,6	6,7		

$$\text{end}[n] = \text{gen}[n] \cup (\text{begin}[n] \setminus \text{kills}[n])$$

$$\text{begin}[n] = \bigcup_{p \in \text{pred}[n]} \text{end}[p]$$

			1st step		2nd step		3rd step	
	gen	*kills*	*begin*	*end*	*begin*	*end*	*begin*	*end*
1	1	6		1		1		1
2	2	4,7	1	1,2	1	1,2	1	1,2
3			1,2	1,2	1,2,4	1,2,4	1,2,4	1,2,4
4	4	2,7	1,2	1,4	1,2,4	1,4	1,2,4	1,4
5			1,4	1,4	1,4	1,4	1,4	1,4
6	6	1	1,2	2,6	1,2,4	2,4,6	1,2,4	2,4,6
7	7	2,4	2,6	6,7	2,4,6	6,7	2,4,6	6,7

$$\text{end}[n] = \text{gen}[n] \cup (\text{begin}[n] \setminus \text{kills}[n])$$

$$\text{begin}[n] = \bigcup_{p \in \text{pred}[n]} \text{end}[p]$$

Constant Propagation

- If we have a statement $d_1 : t := c$, with c constant, and another statement d_2 that uses t.
- t is constant
- if d_1 reaches d_2 **and** no other definition of t reaches d_2
- then we can rewrite d_2

In the previous example, only one definition of a reaches statement 3 so we can replace $c > a$ by $c > 5$.

Copy Propagation

- If we have a statement $d_1 : t := z$, with z variable, and another statement d_2 that uses t.
- t is constant
- if d_1 reaches d_2 **and** no other definition of t reaches d_2 **and** the is no definition of z in all pathes between d_1 and d_2
- then we can rewrite d_2

Good register allocator will automatically detects some such cases.

Optimizing compiler

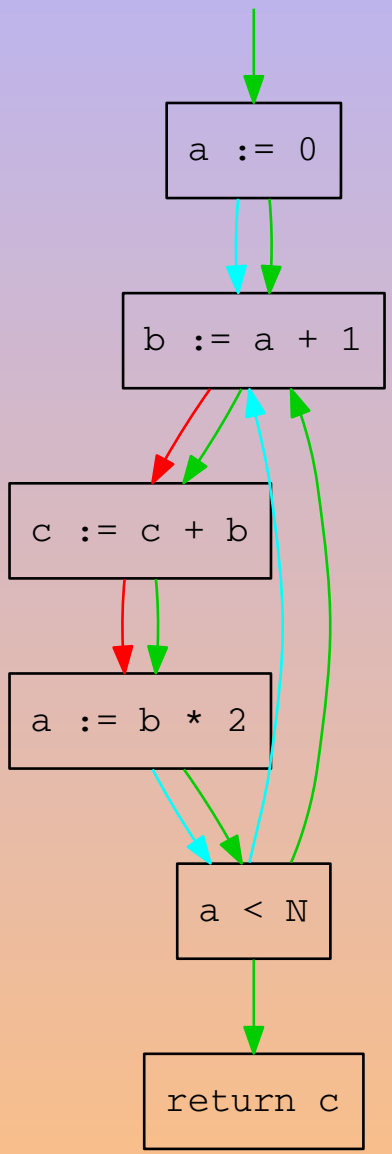
The removal of dead statements (or other optimizations) might introduce new dead statements.

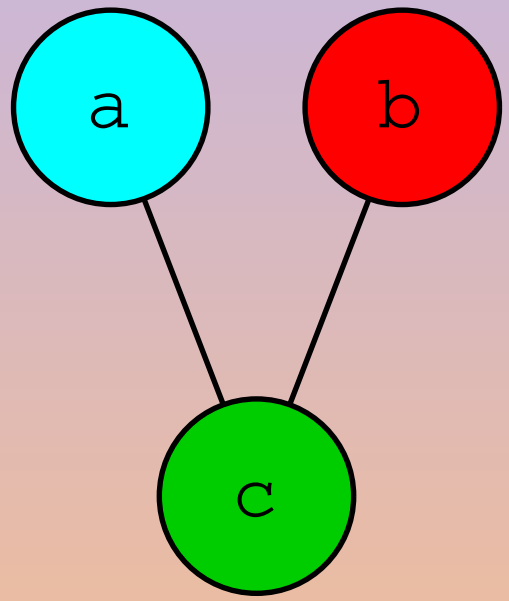
To avoid the need for repeated global calculation, several strategies exist:

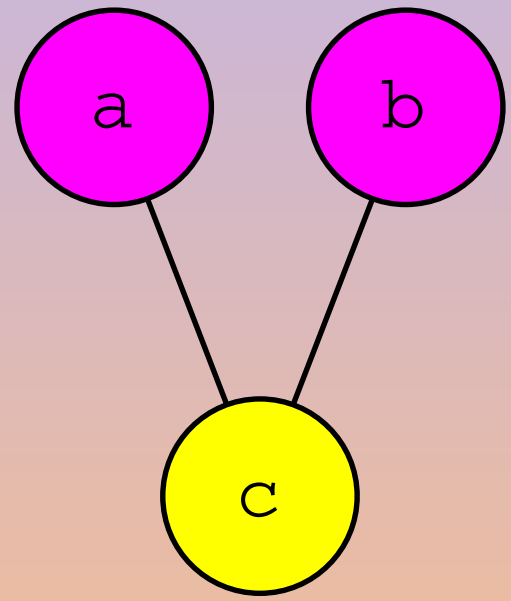
- Cutoff: perform no more than k round
- Cascading analysis: predict the cascade of effects of an optimization. Value numbering is a typical case of cascading analysis
- Incremental dataflow analysis: patch the dataflow after applying an optimization.

Interference Graph

Interference Graph

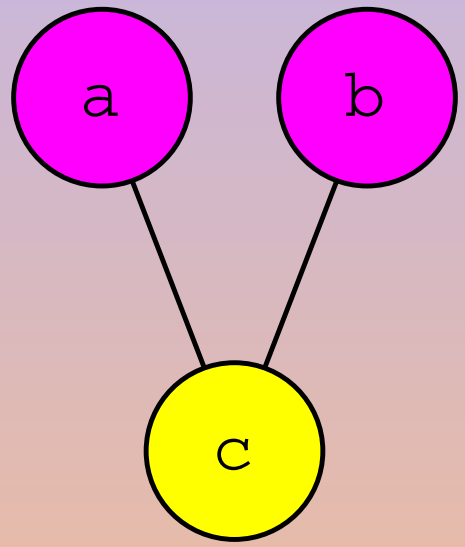






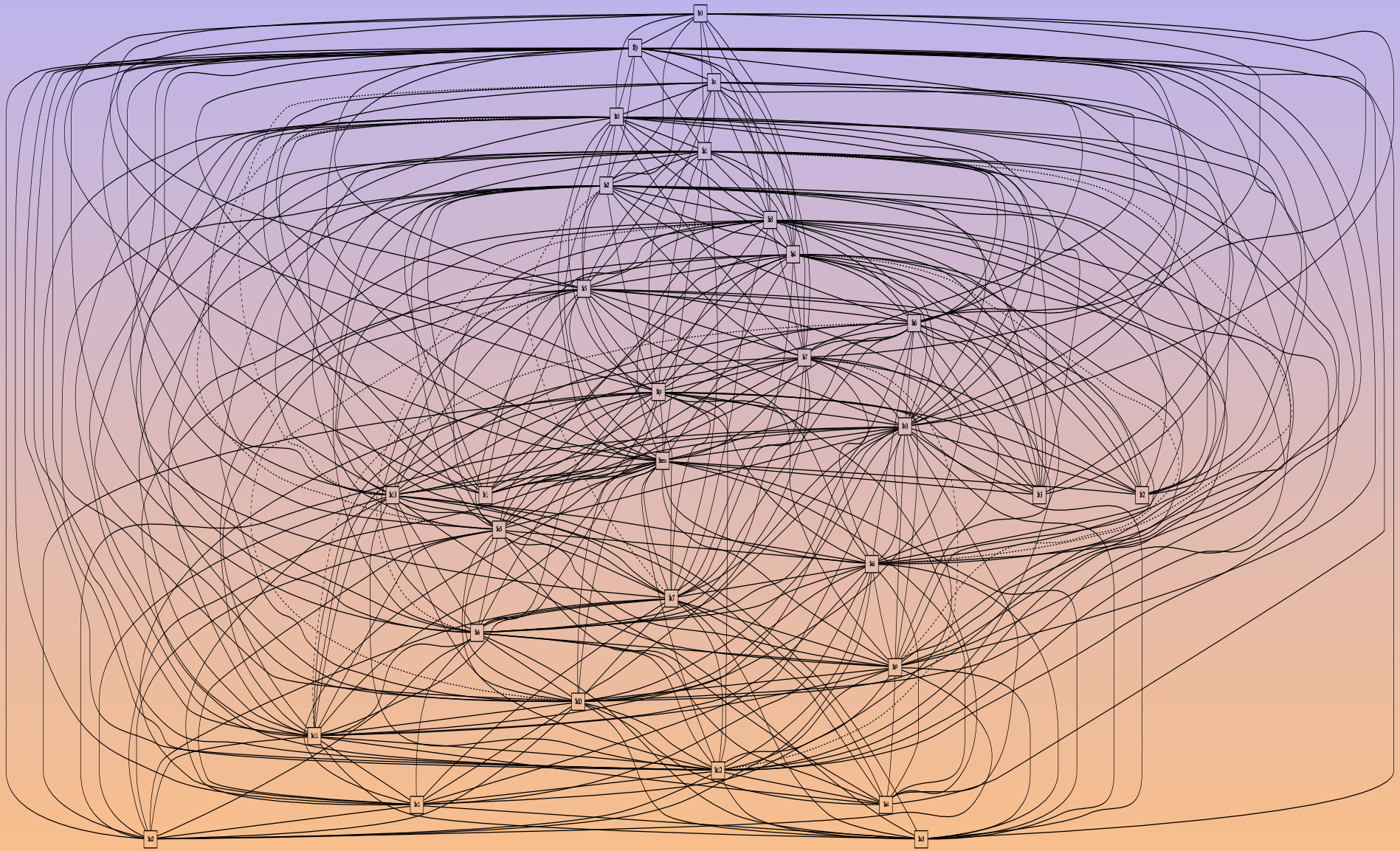
Register Allocation

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```

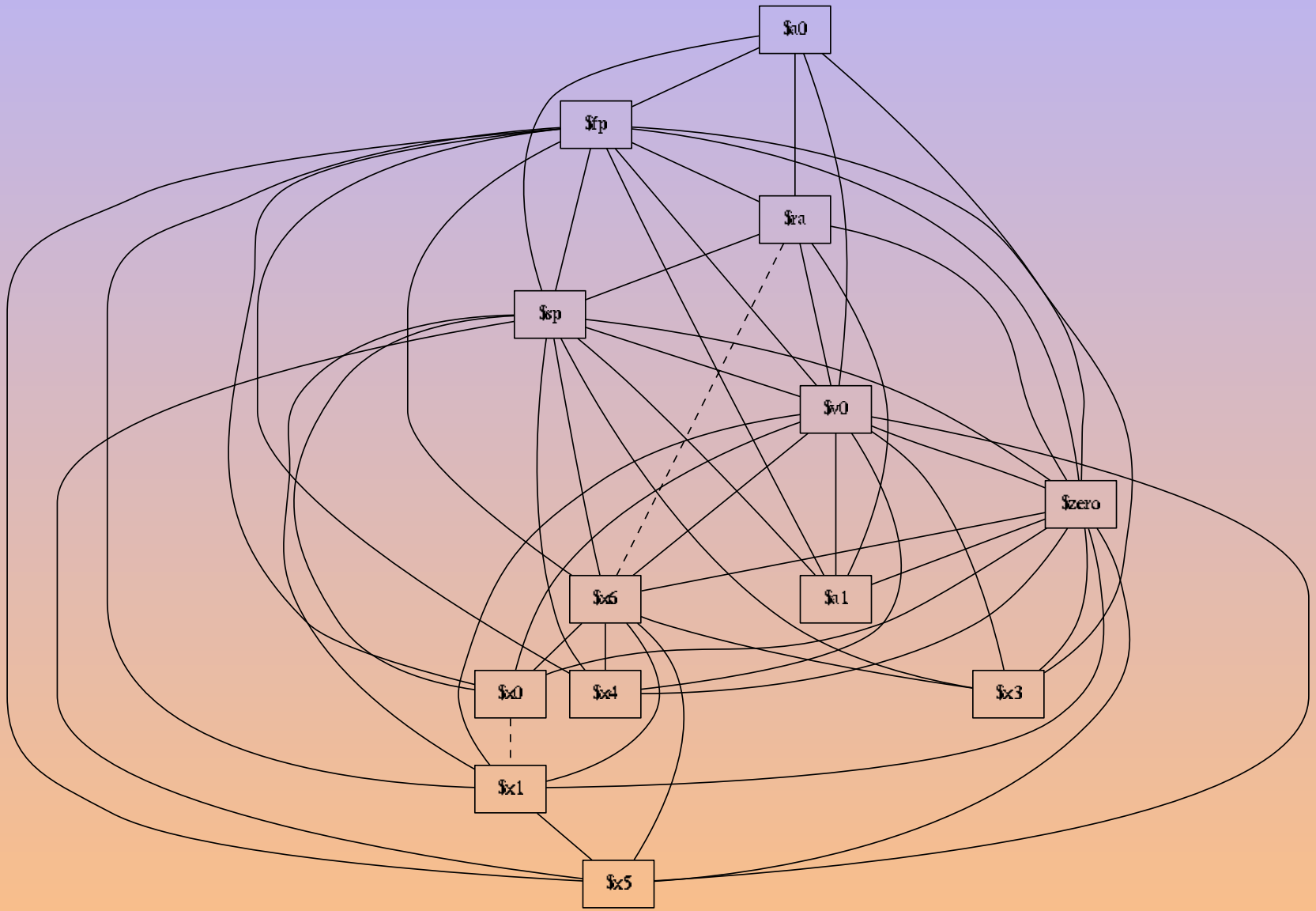



```
    r1 := 0
L1: r1 := r1 + 1
    r2 := r2 + r1
    r1 := r1 * 2
    if r1 < N goto L1
    return r2
```

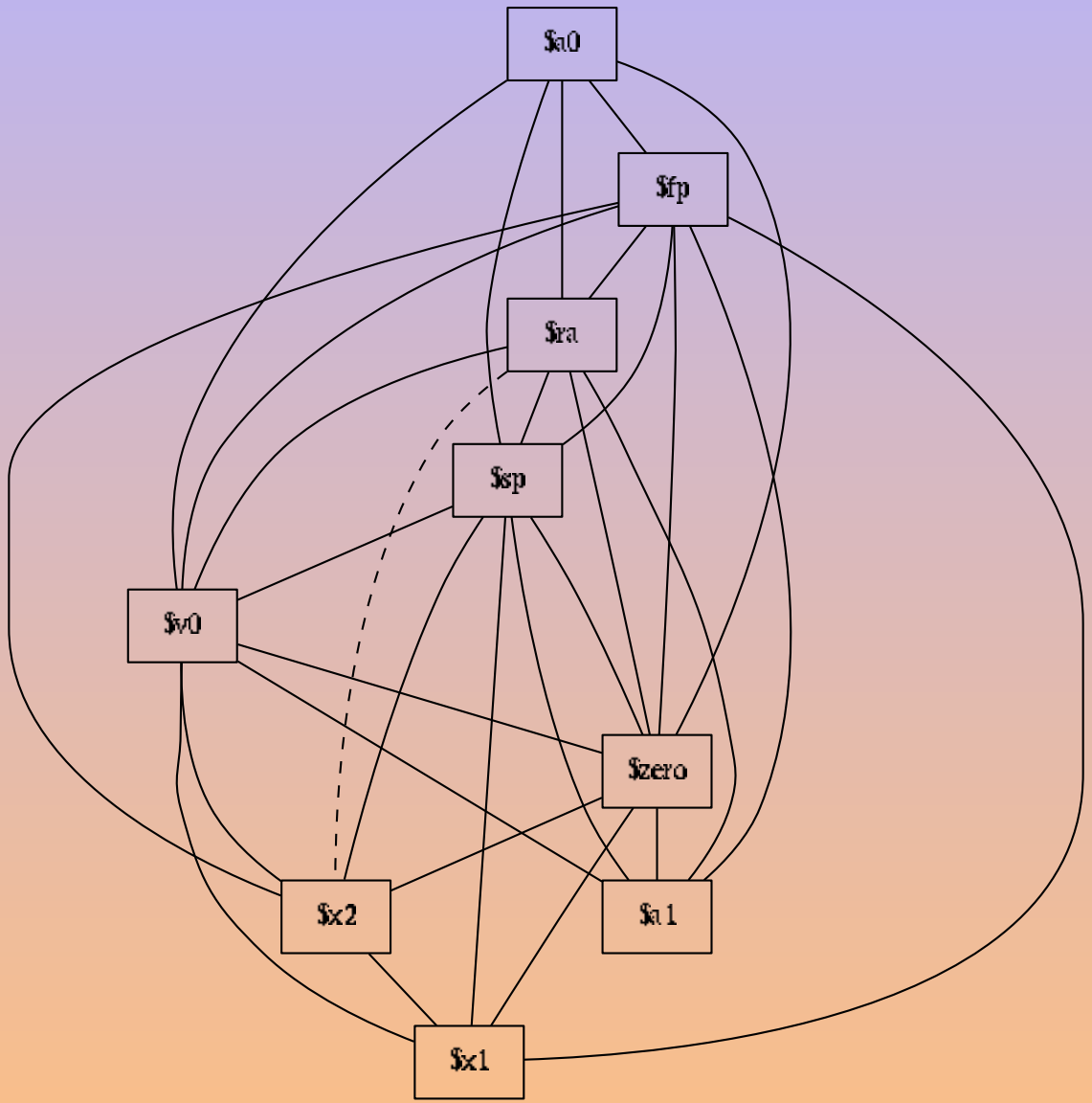
7's Interference Graph



7000's Interference Graph



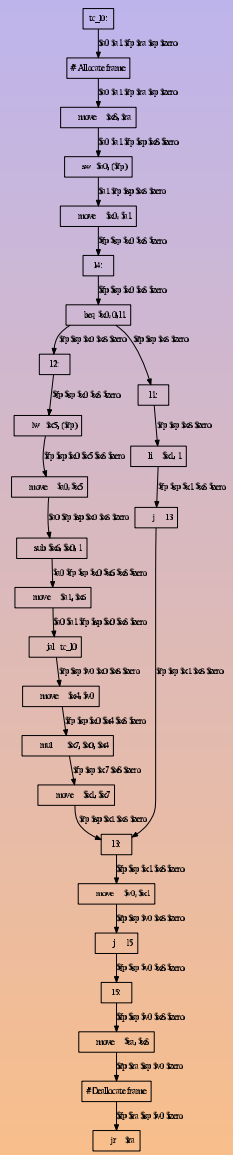
ors' Interference Graph



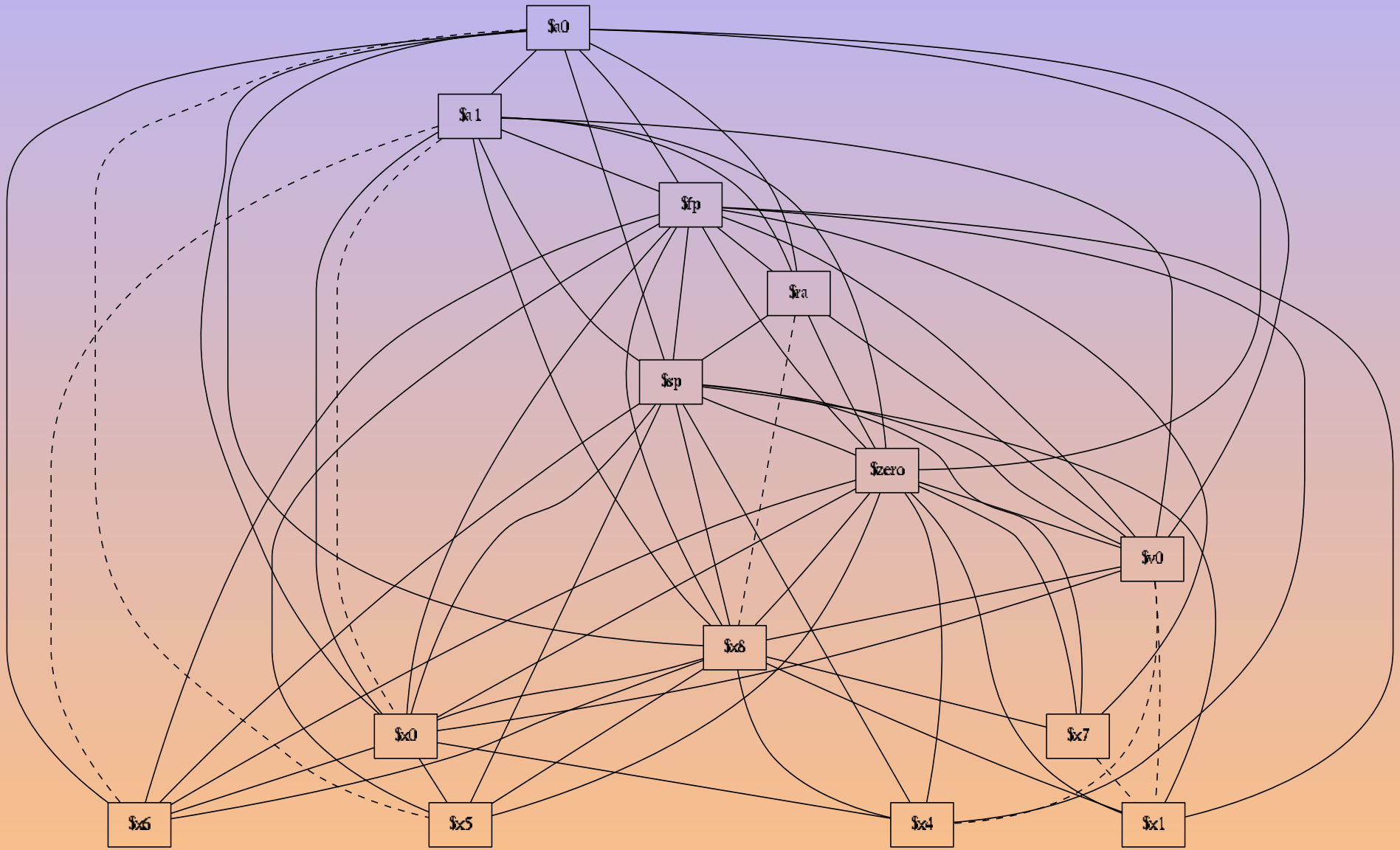
fact.tig

```
let function fact (n : int) : int =
      if n = 0 then
        1
      else
        n * fact (n - 1)
in
  fact (12)
end
```

fact's Liveness Graph



fact's Interference Graph



Bibliography I


Appel, A. W. (1998).
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press.

Register Allocation

Akim Demaille Étienne Renault Roland Levillain
first.last@lrde.epita.fr

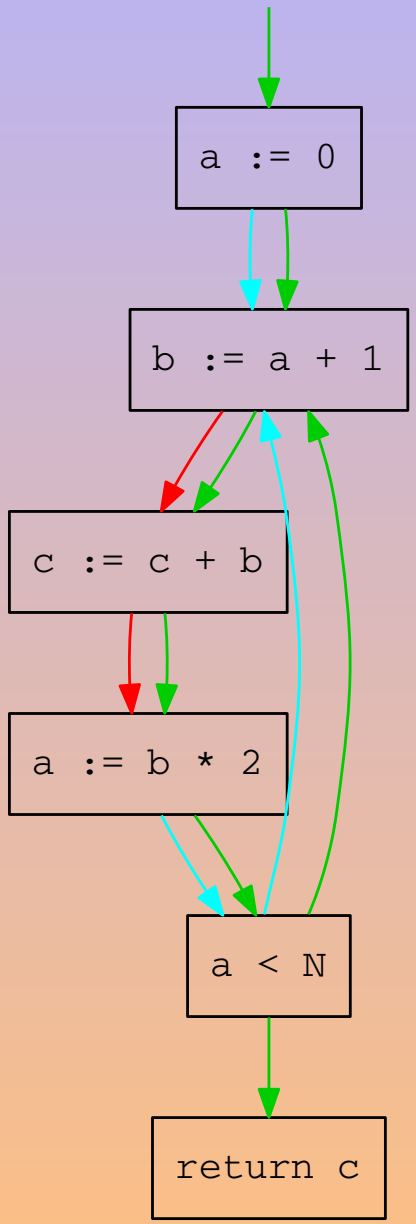
EPITA — École Pour l'Informatique et les Techniques Avancées

May 19, 2018

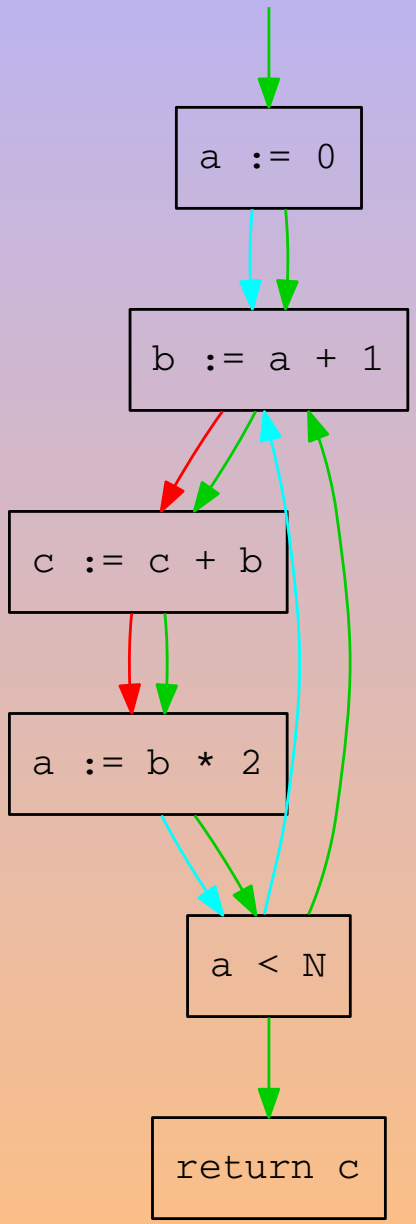
Register Allocation

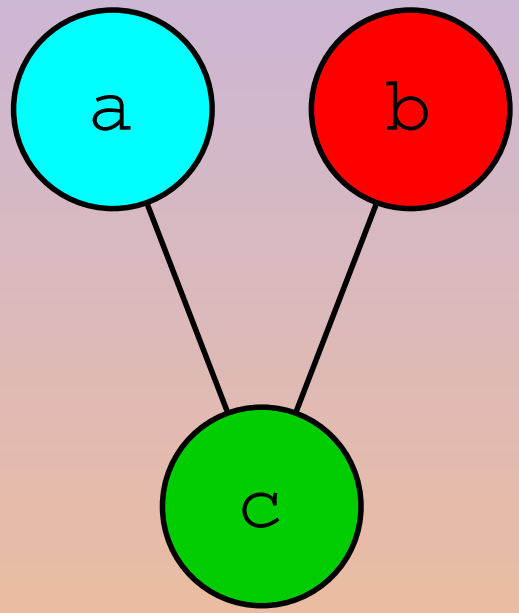
Interference Graph

Interference Graph

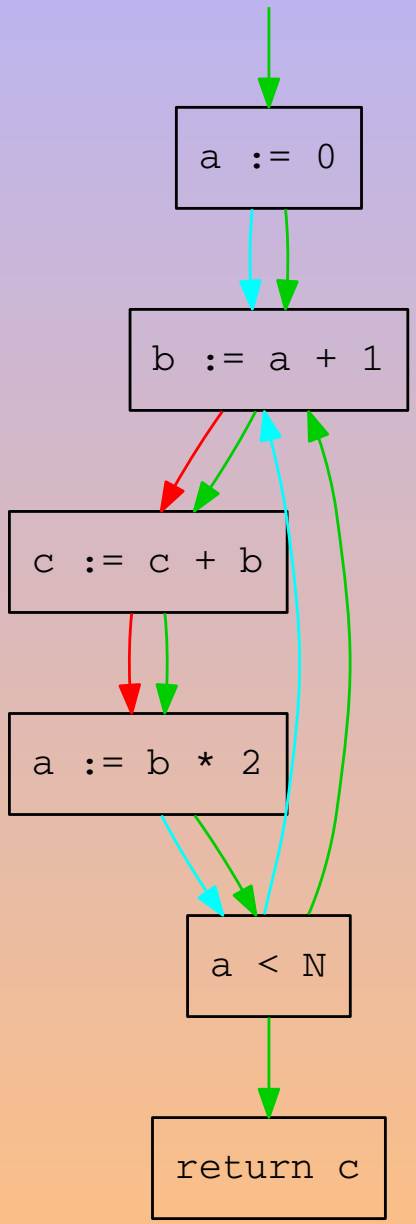


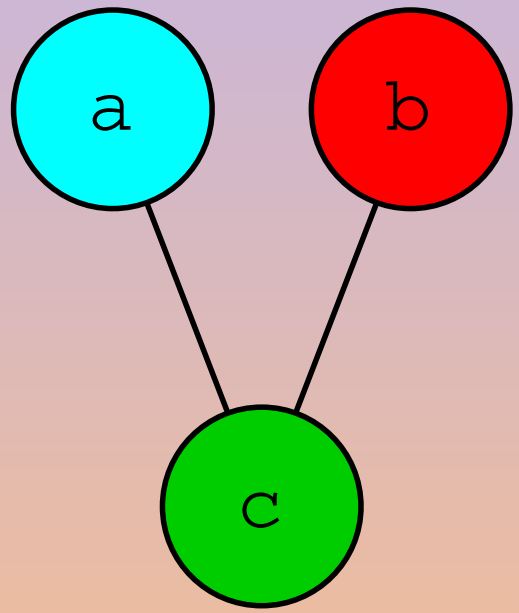
Interference Graph

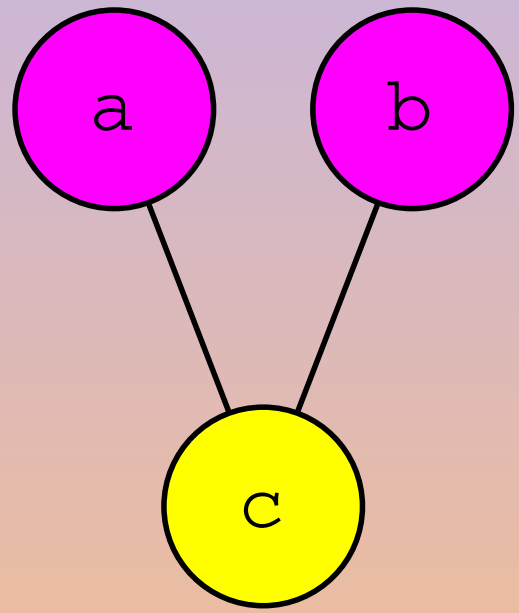




Interference Graph





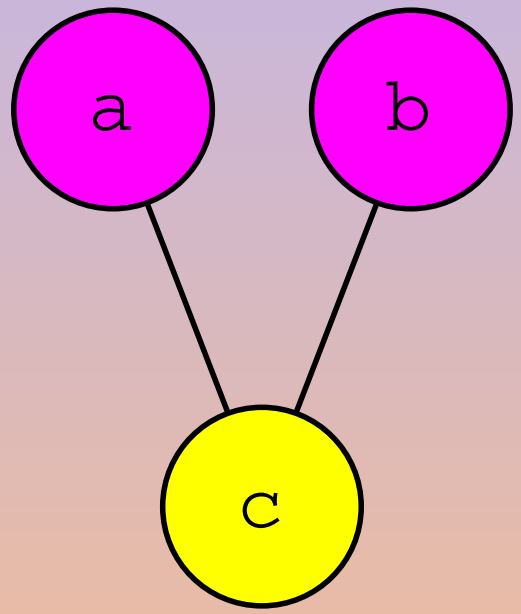


Register Allocation

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```

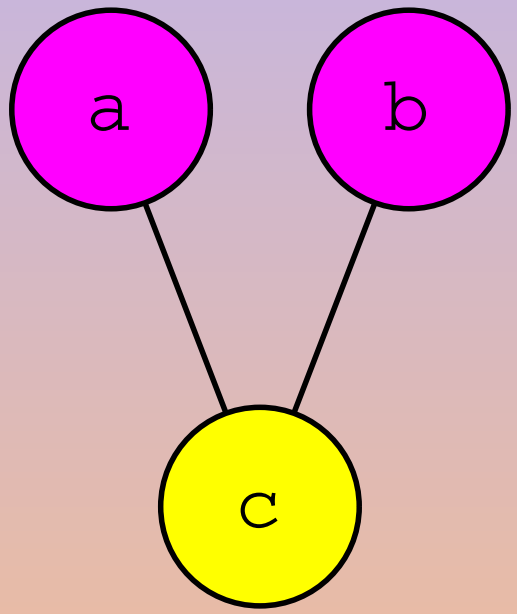
Register Allocation

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```



Register Allocation

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```



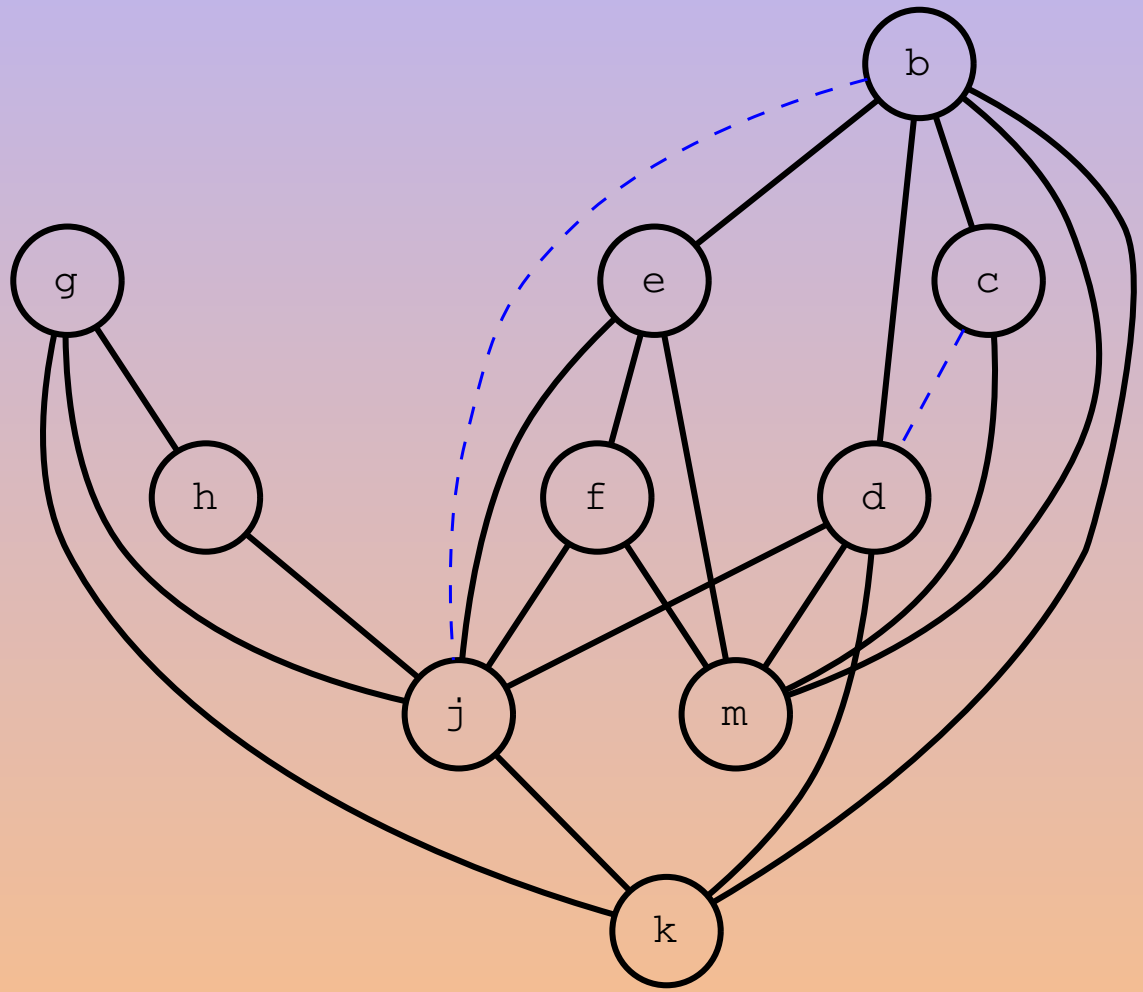

```
    r1 := 0
L1: r1 := r1 + 1
    r2 := r2 + r1
    r1 := r1 * 2
    if r1 < N goto L1
    return r2
```

Coloring by Simplification

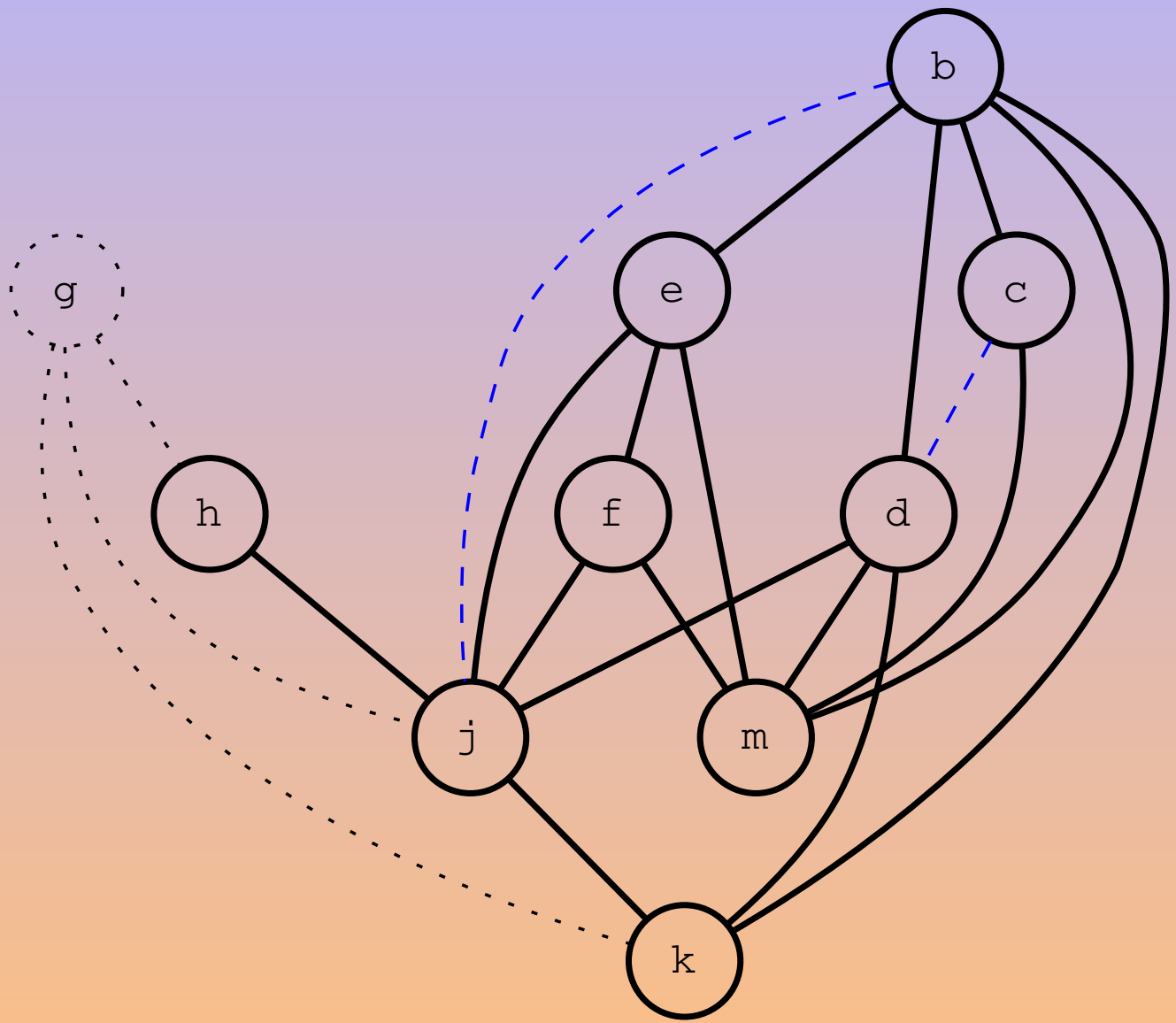
Interference Graph [Appel, 1998]

Four registers: r1, r2, r3, r4.

```
live in: k j
  g := [j + 12]
  h := k - 1
  f := g * h
  e := [j + 8]
  m := [j + 16]
  b := [f]
  c := e + 8
  d := c
  k := m + 4
  j := b
live out: d k j
```

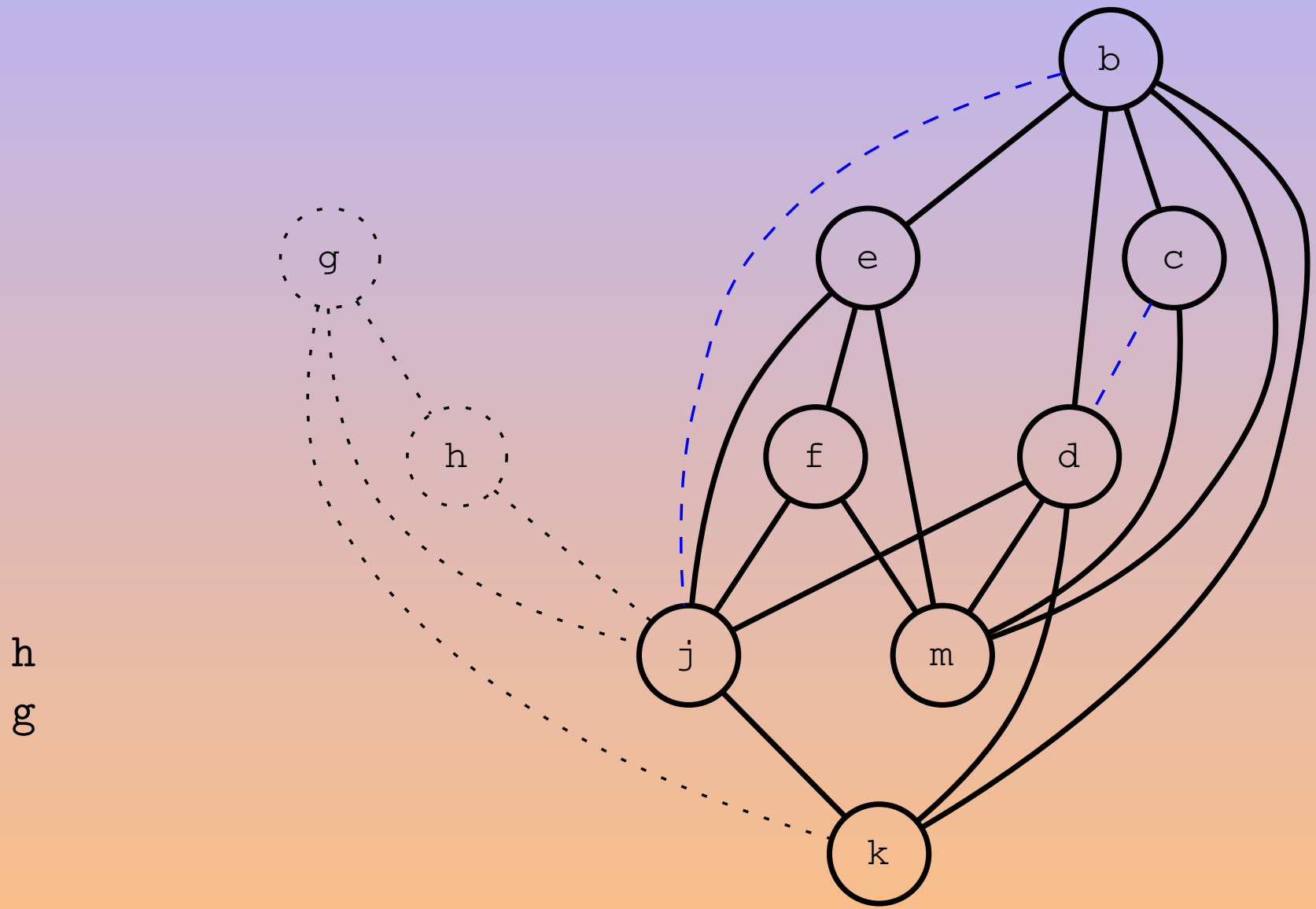


Interference Graph: Simplify 0



g

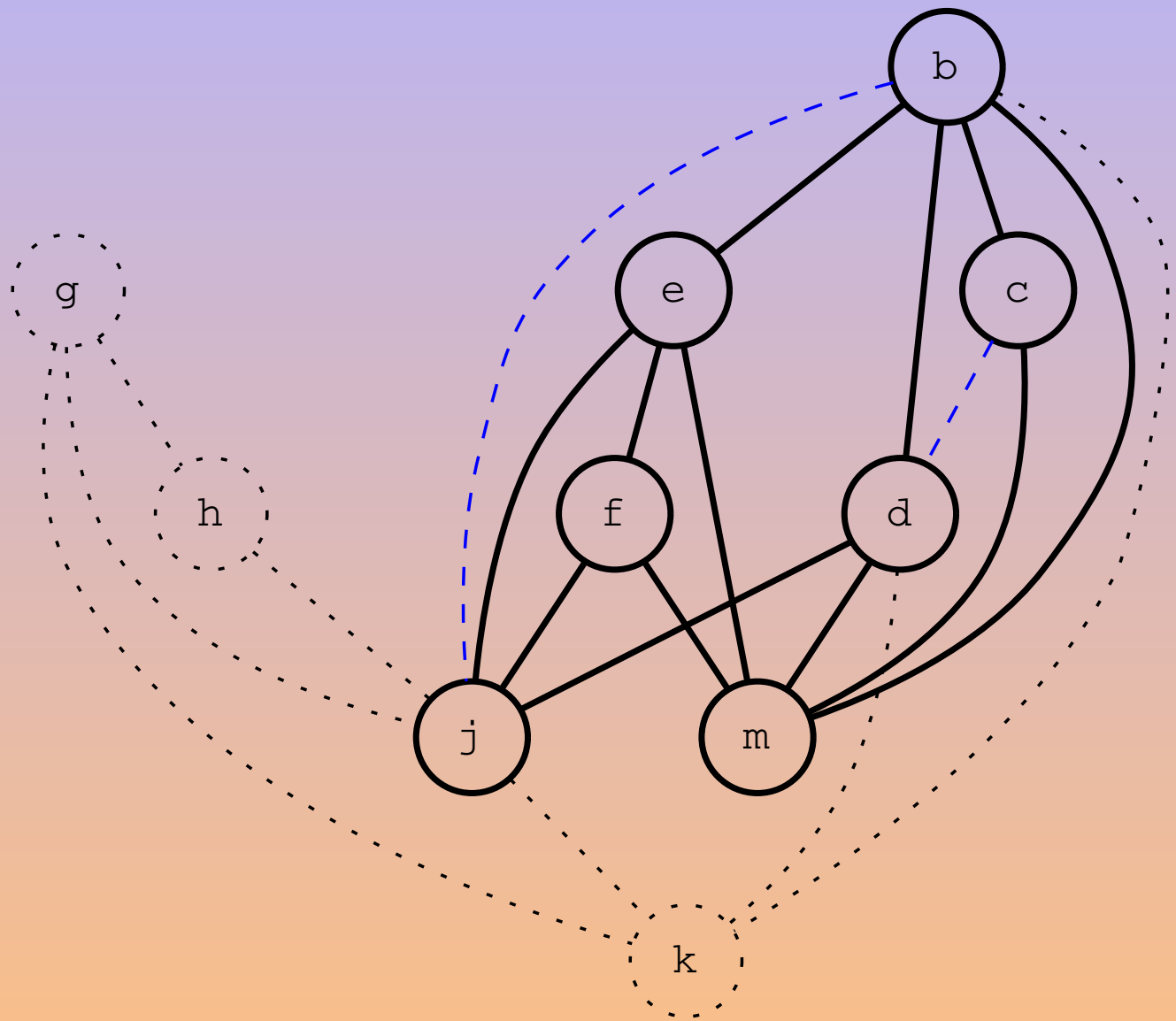
Interference Graph: Simplify 1



h
g

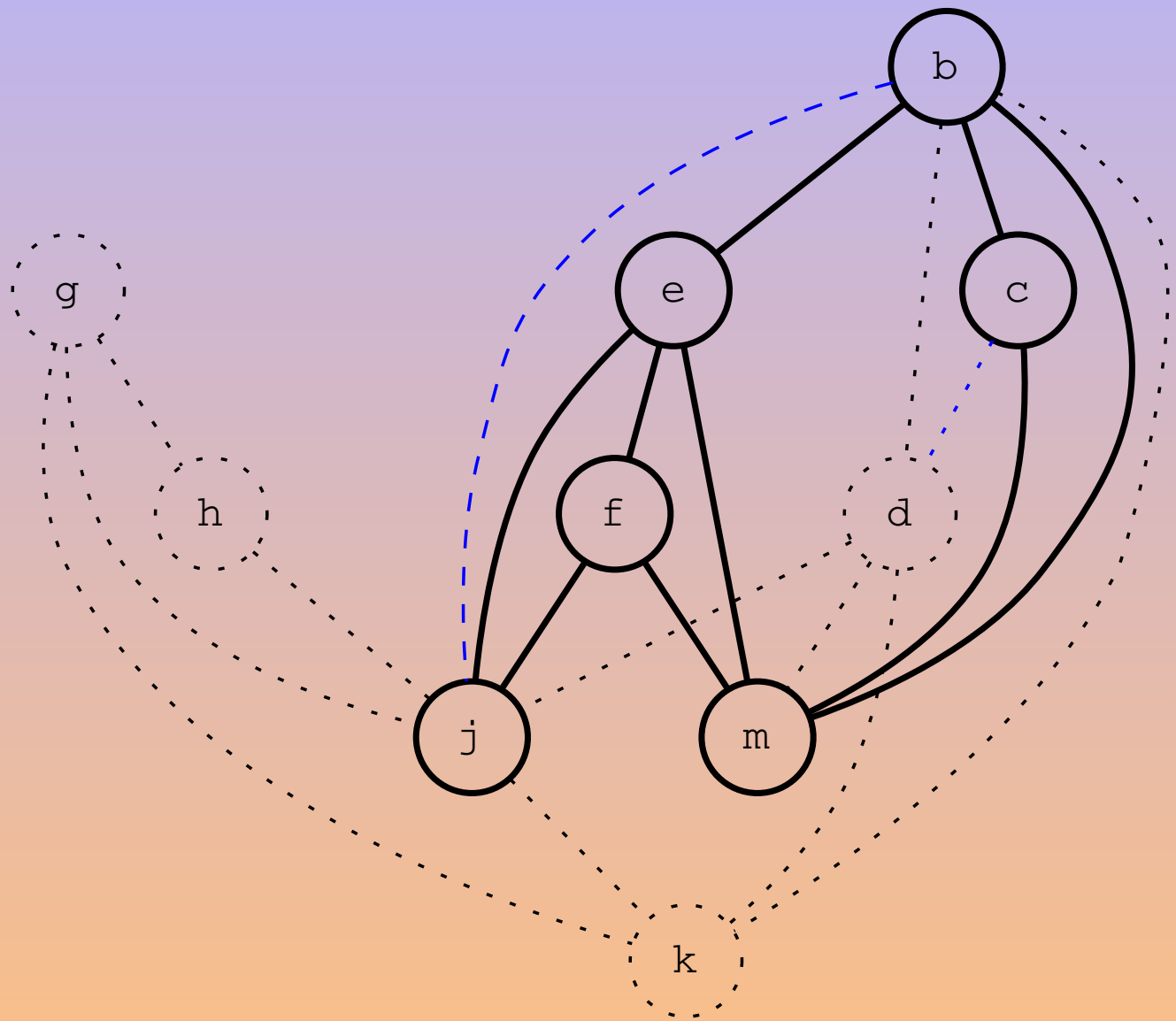
Interference Graph: Simplify 2

k
h
g



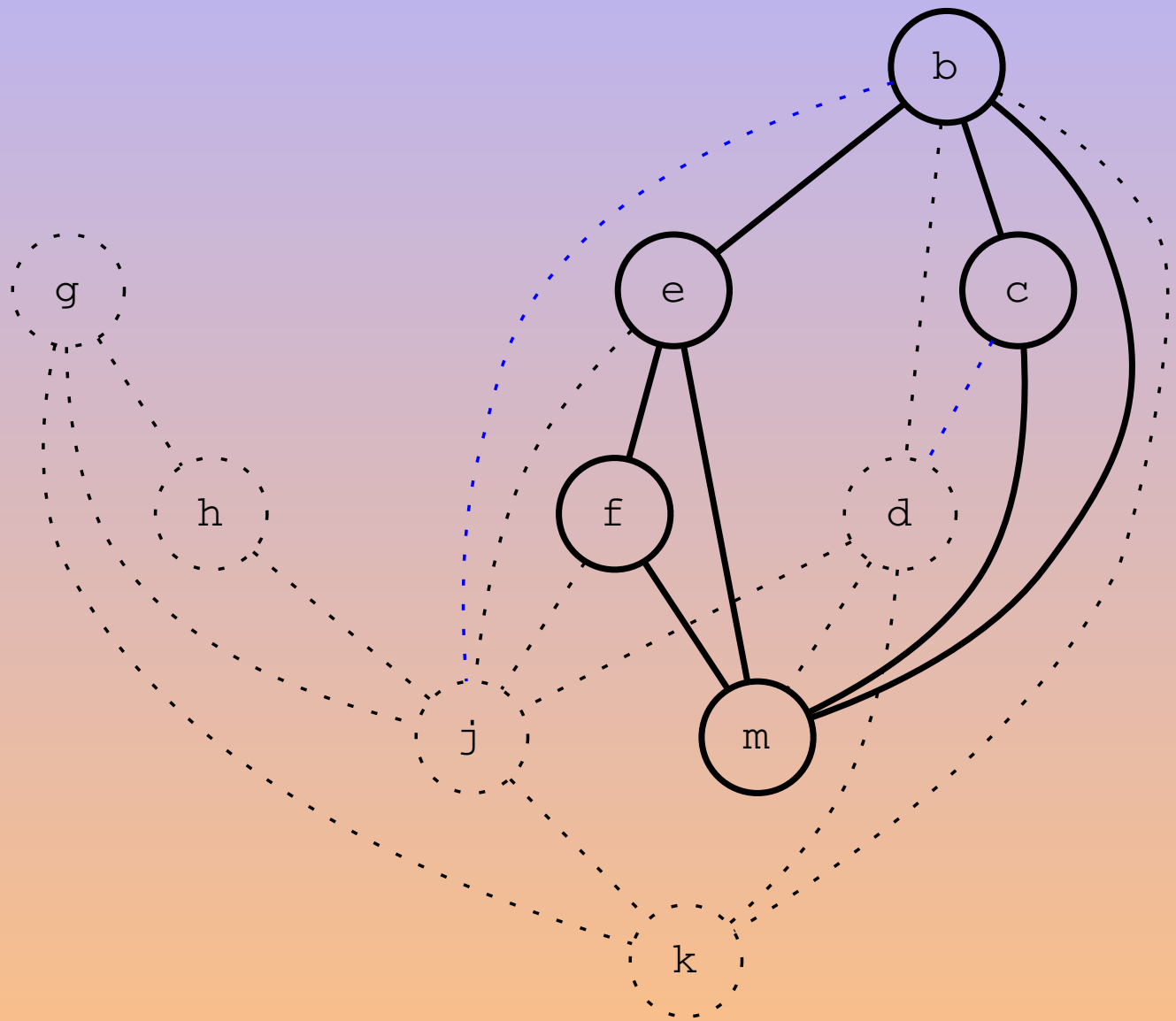
Interference Graph: Simplify 3

d
k
h
g



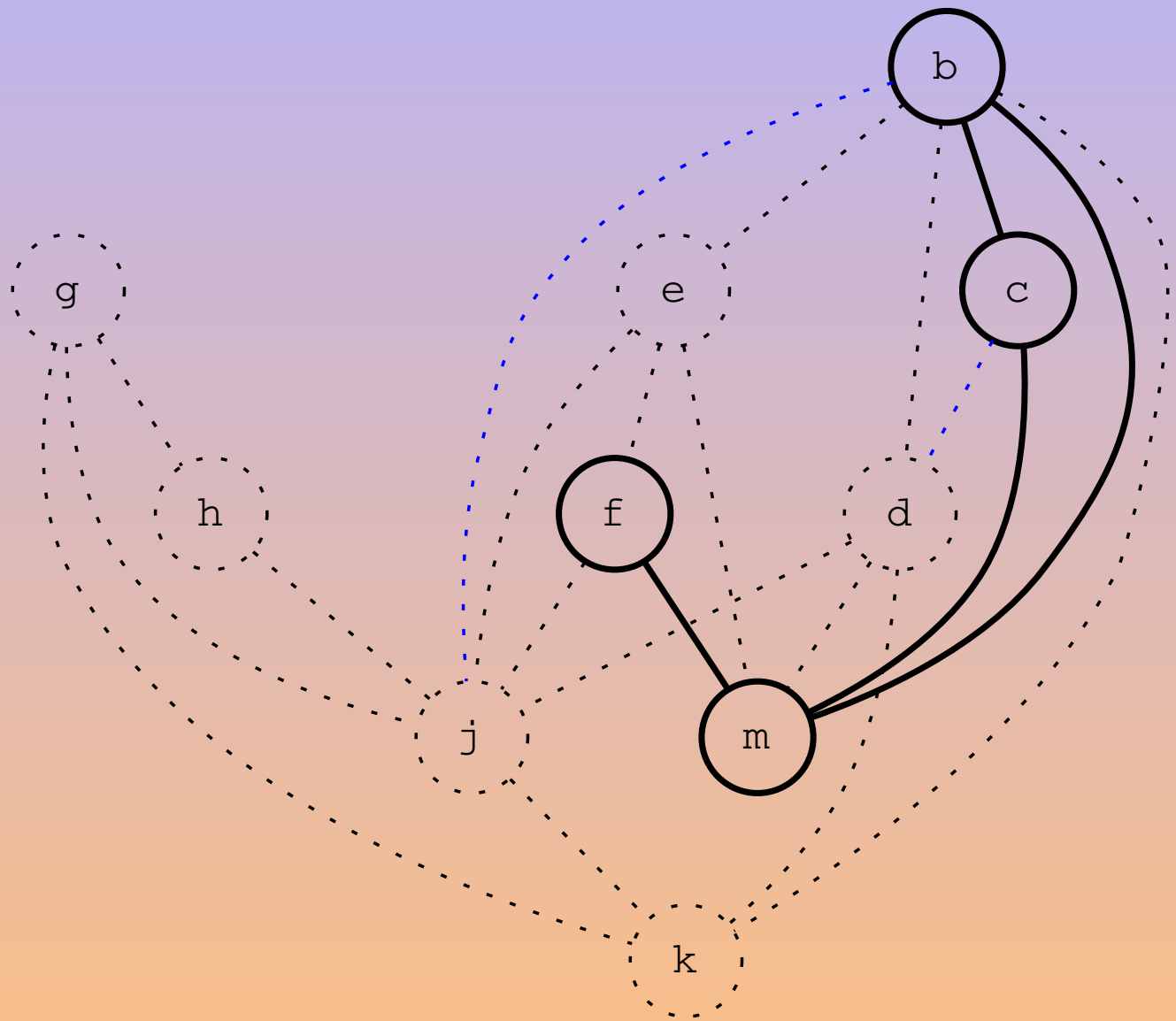
Interference Graph: Simplify 4

j
d
k
h
g



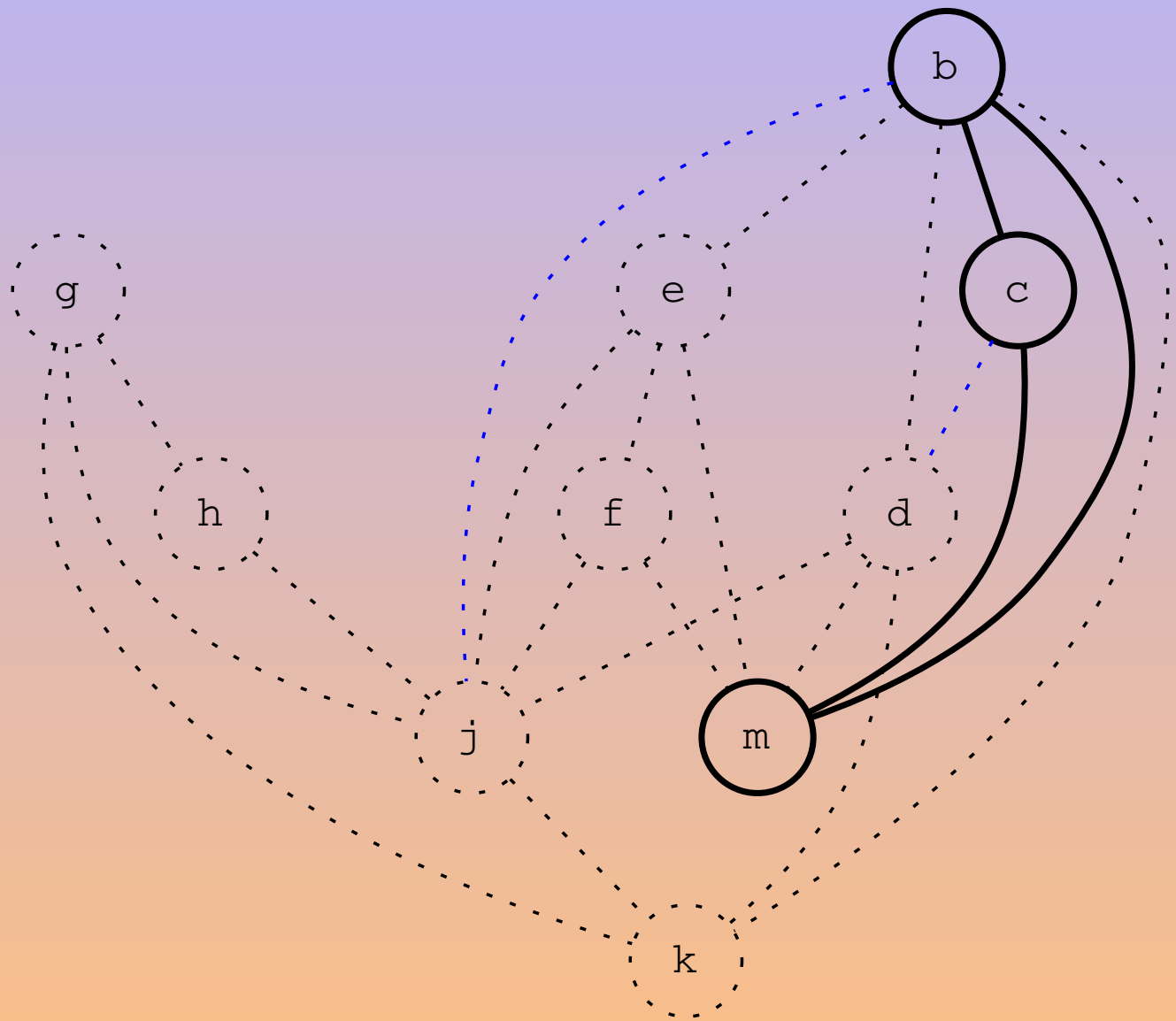
Interference Graph: Simplify 5

e
j
d
k
h
g



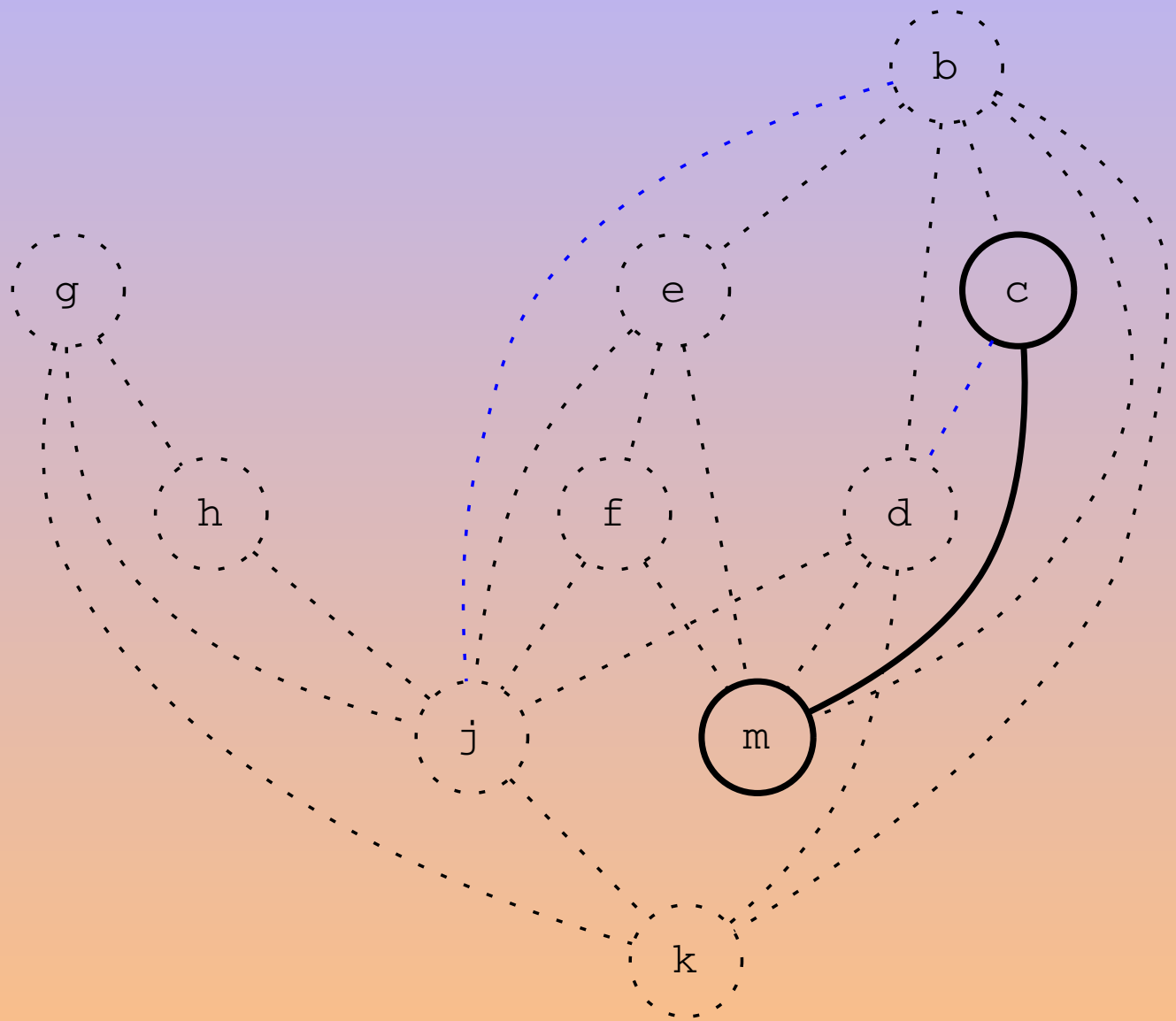
Interference Graph: Simplify 6

f
e
j
d
k
h
g



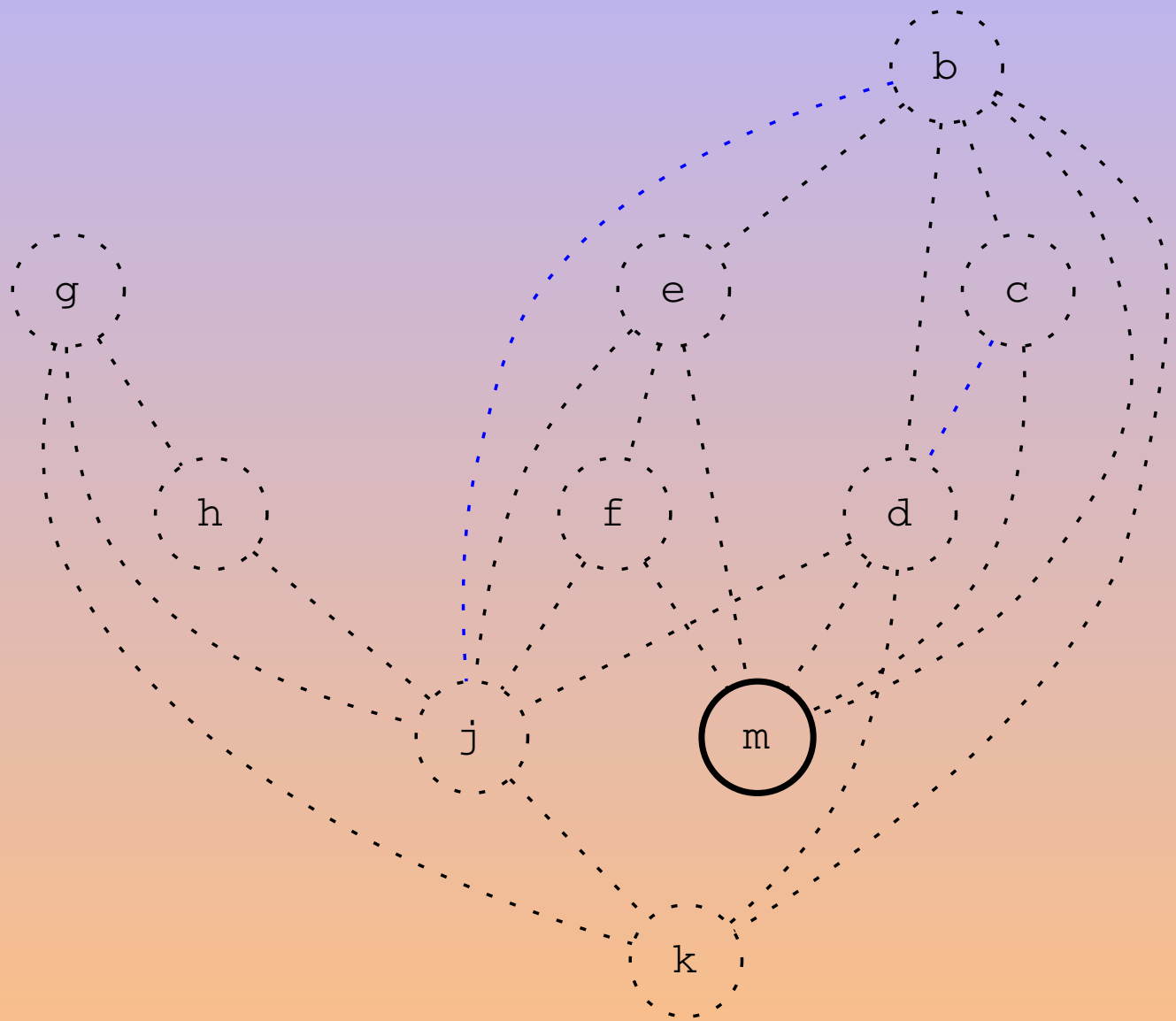
Interference Graph: Simplify 7

b
f
e
j
d
k
h
g



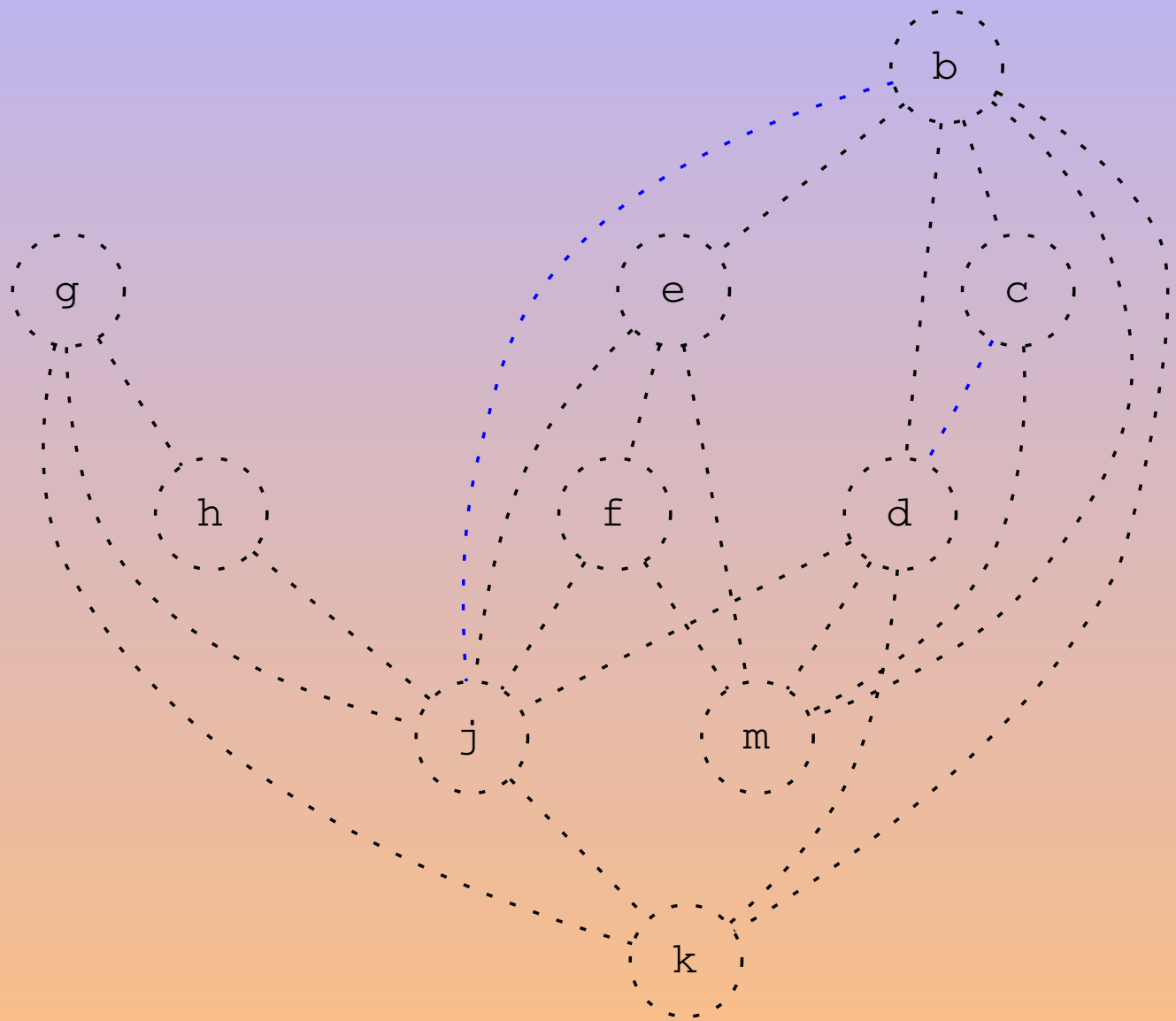
Interference Graph: Simplify 8

c
b
f
e
j
d
k
h
g



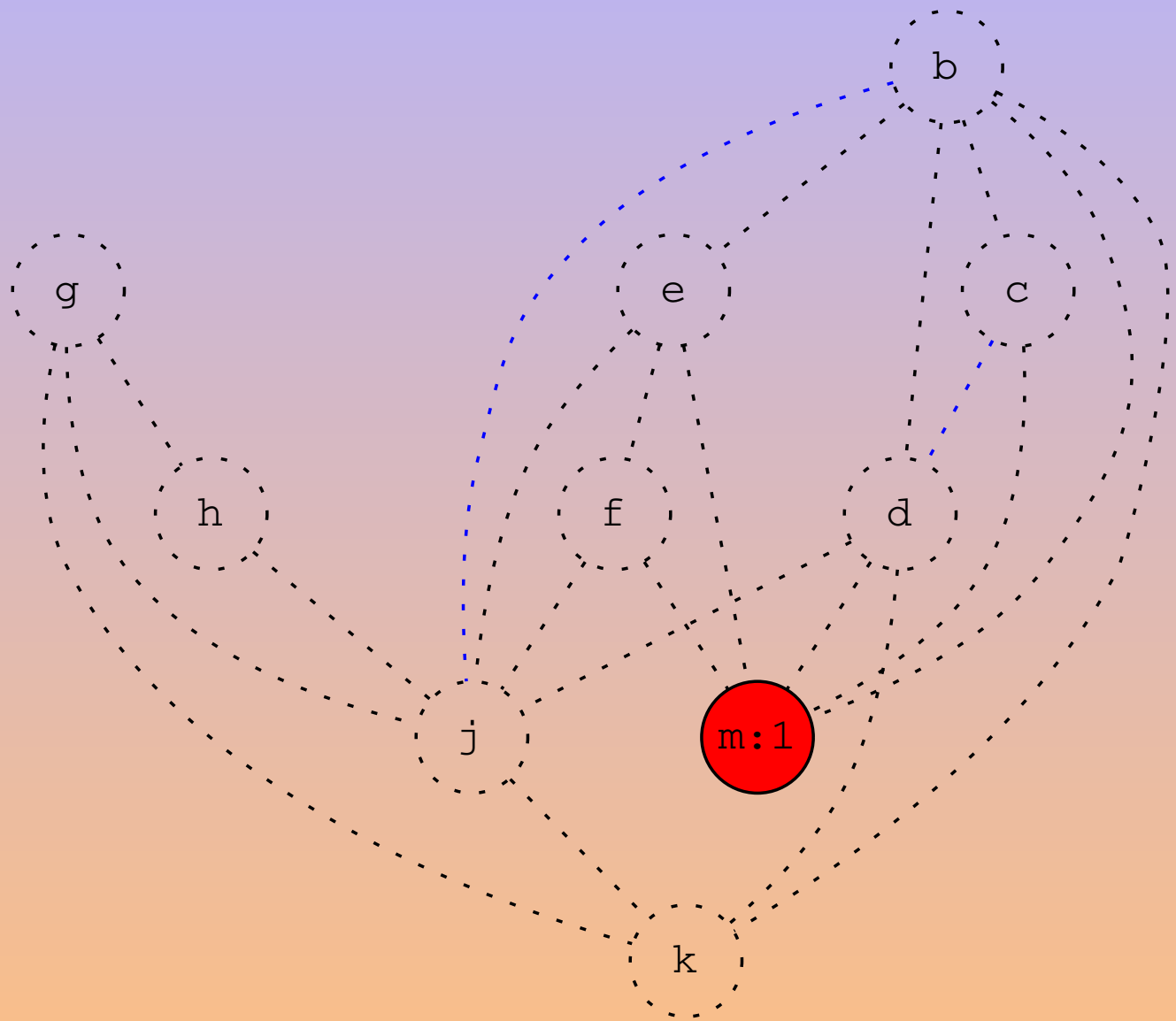
Interference Graph: Simplify 9

m
c
b
f
e
j
d
k
h
g



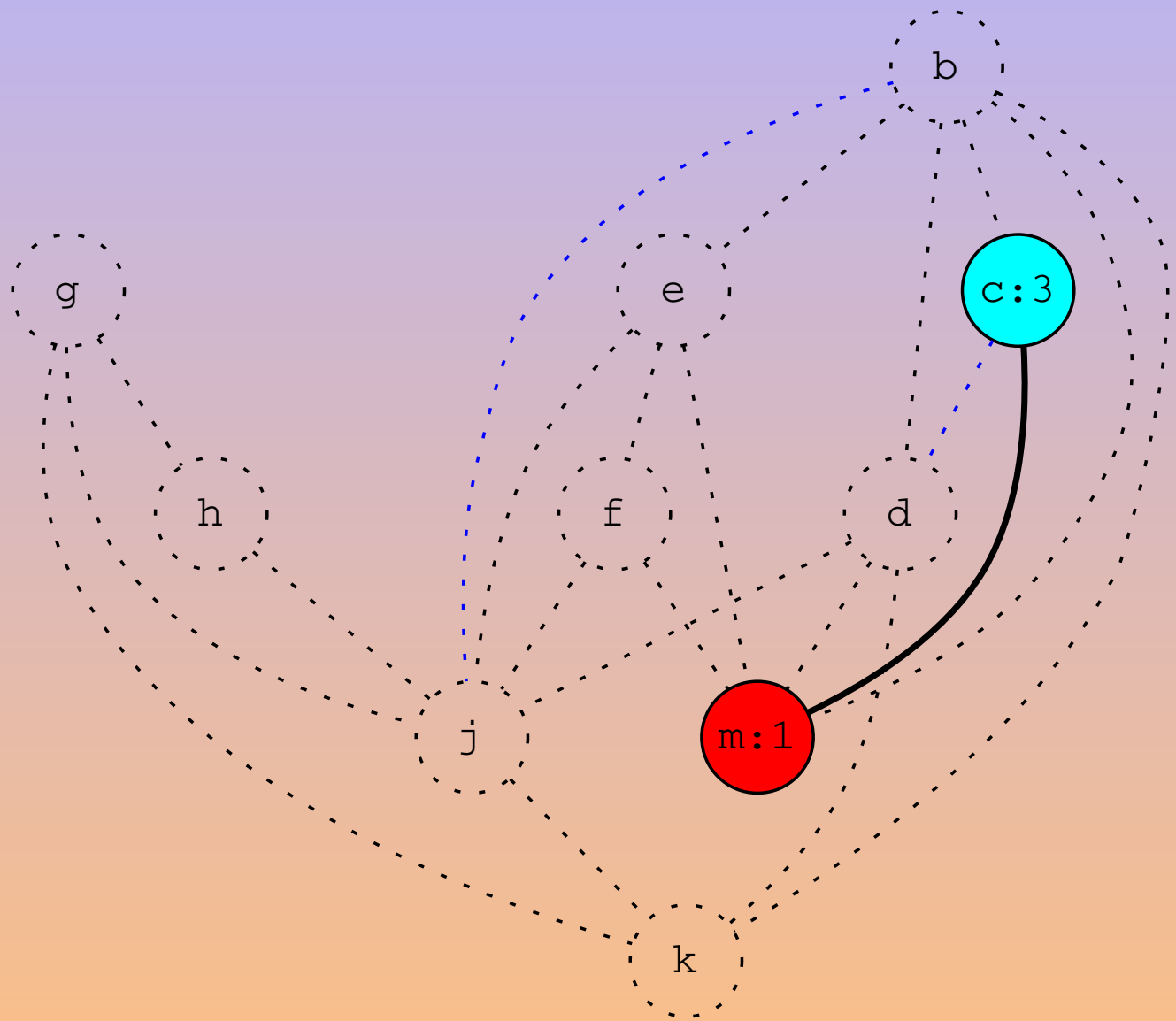
Interference Graph: Color 9

m
c
b
f
e
j
d
k
h
g



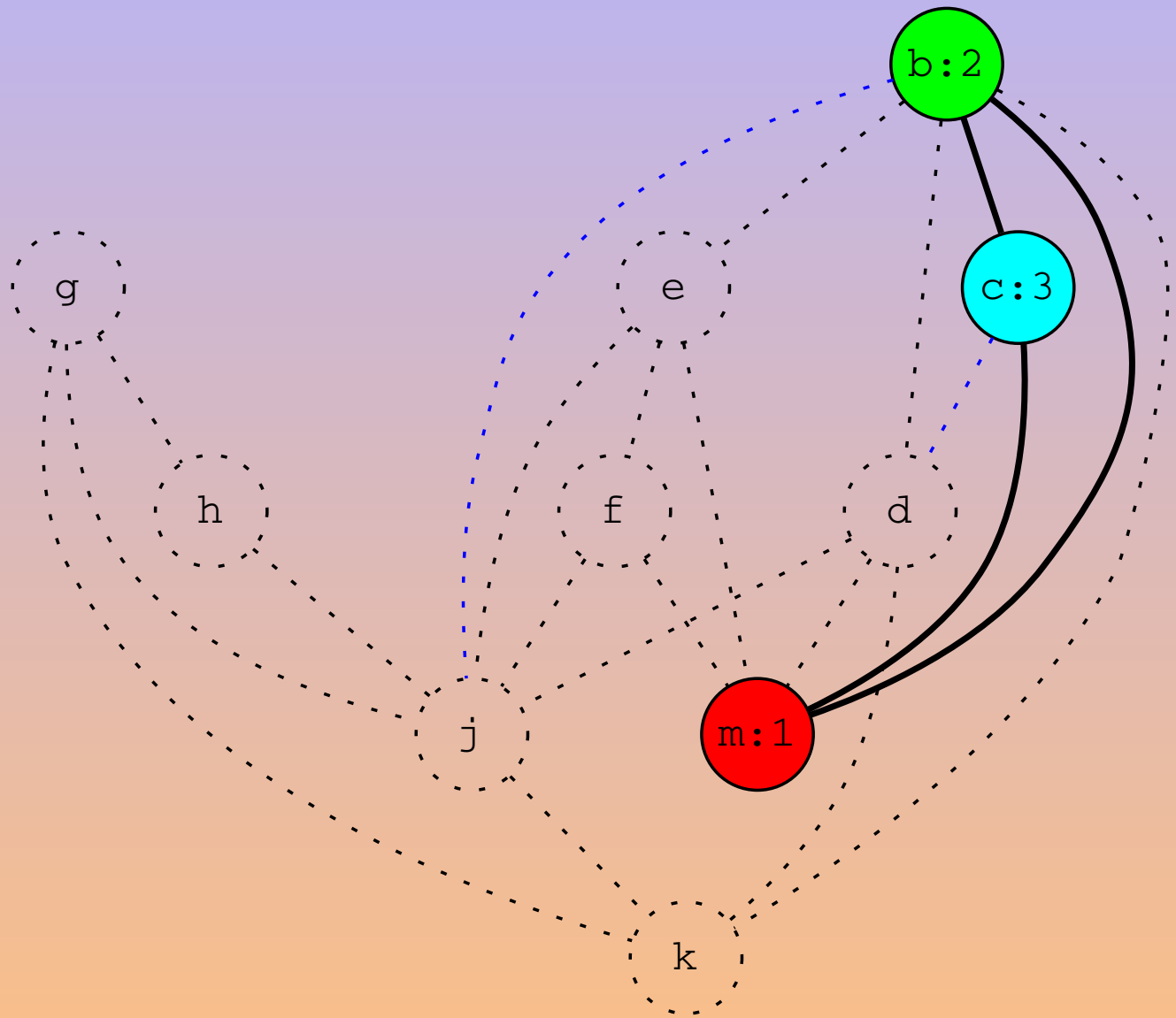
Interference Graph: Color 8

c
b
f
e
j
d
k
h
g



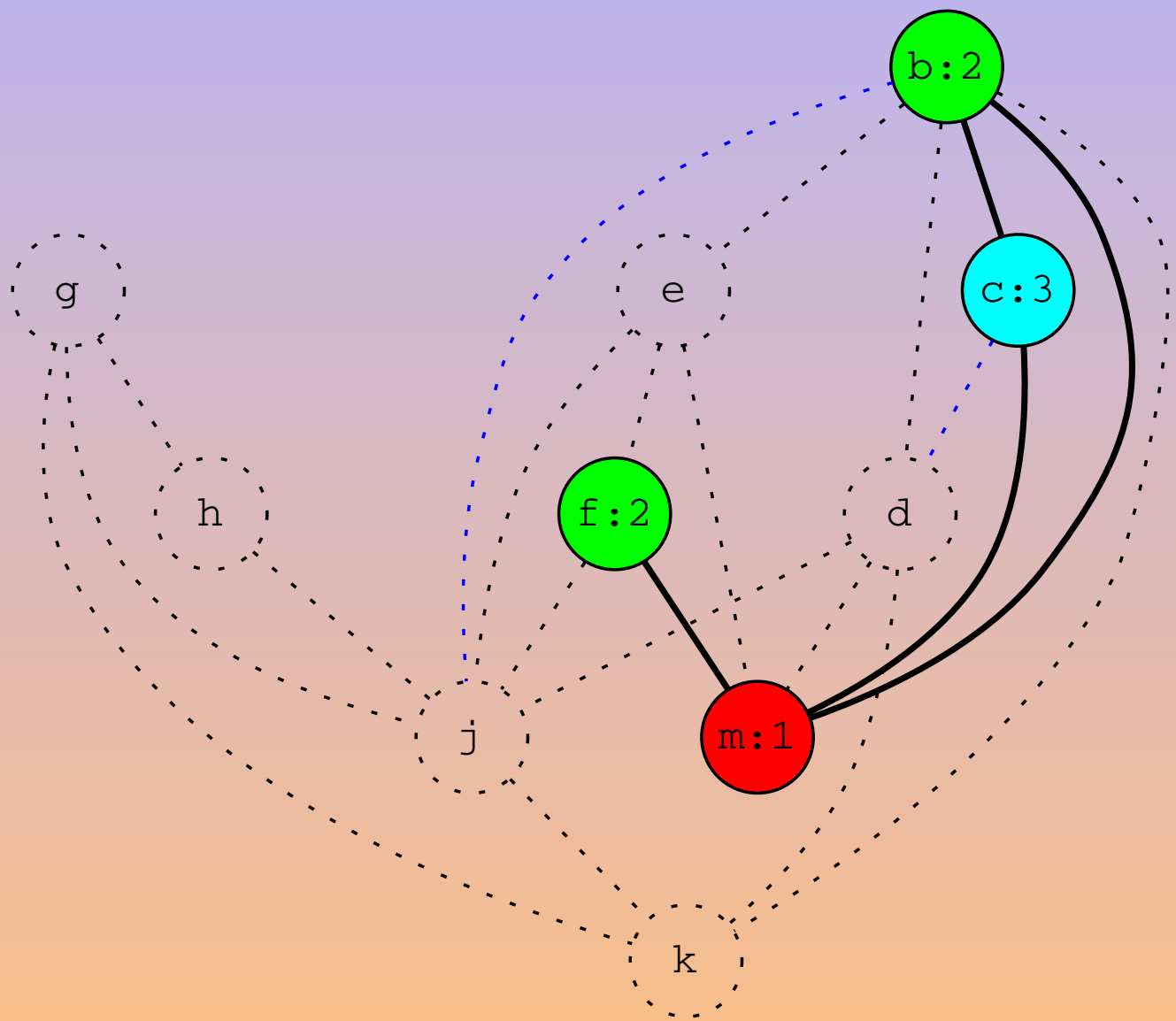
Interference Graph: Color 7

b
f
e
j
d
k
h
g



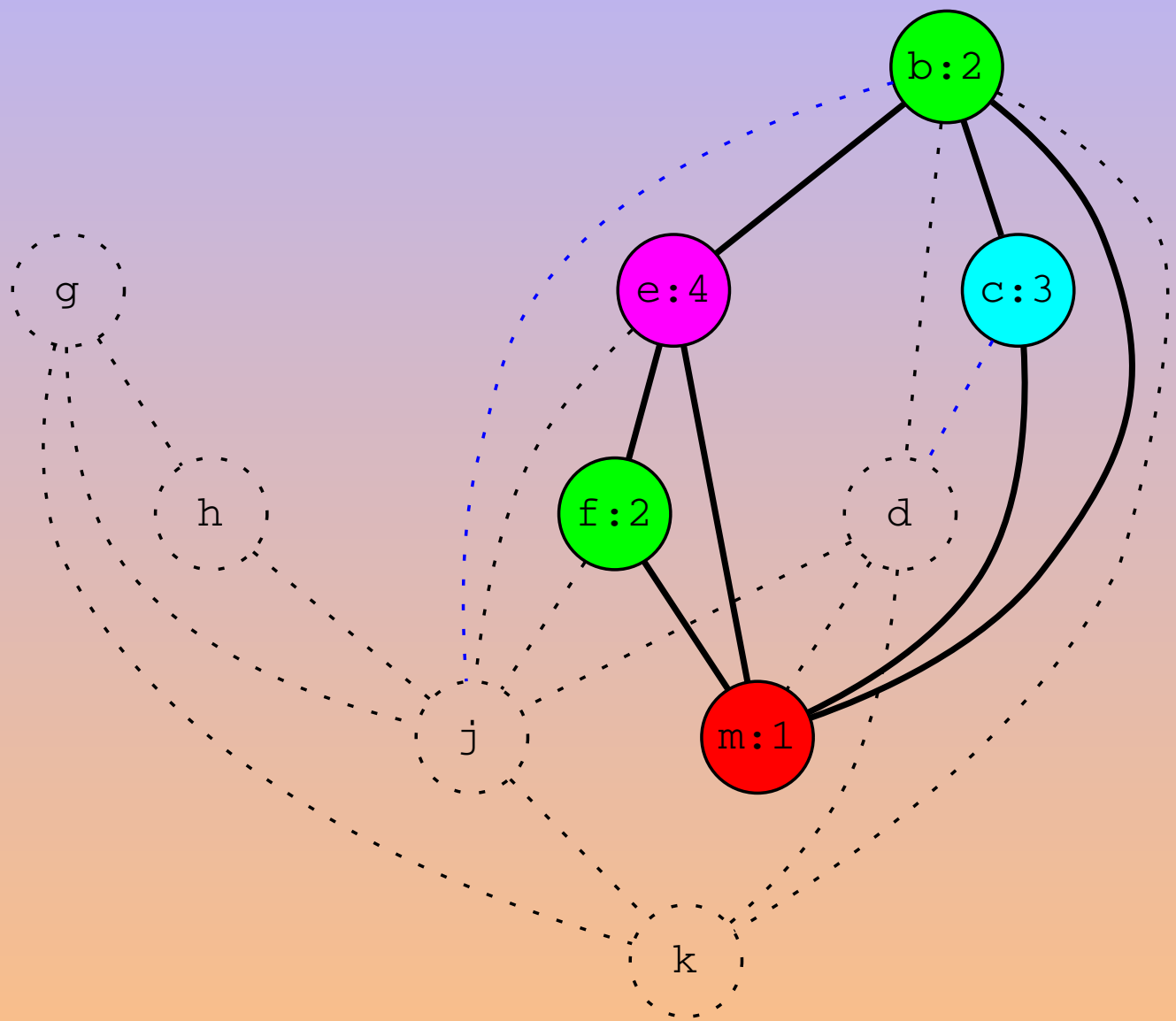
Interference Graph: Color 6

f
e
j
d
k
h
g



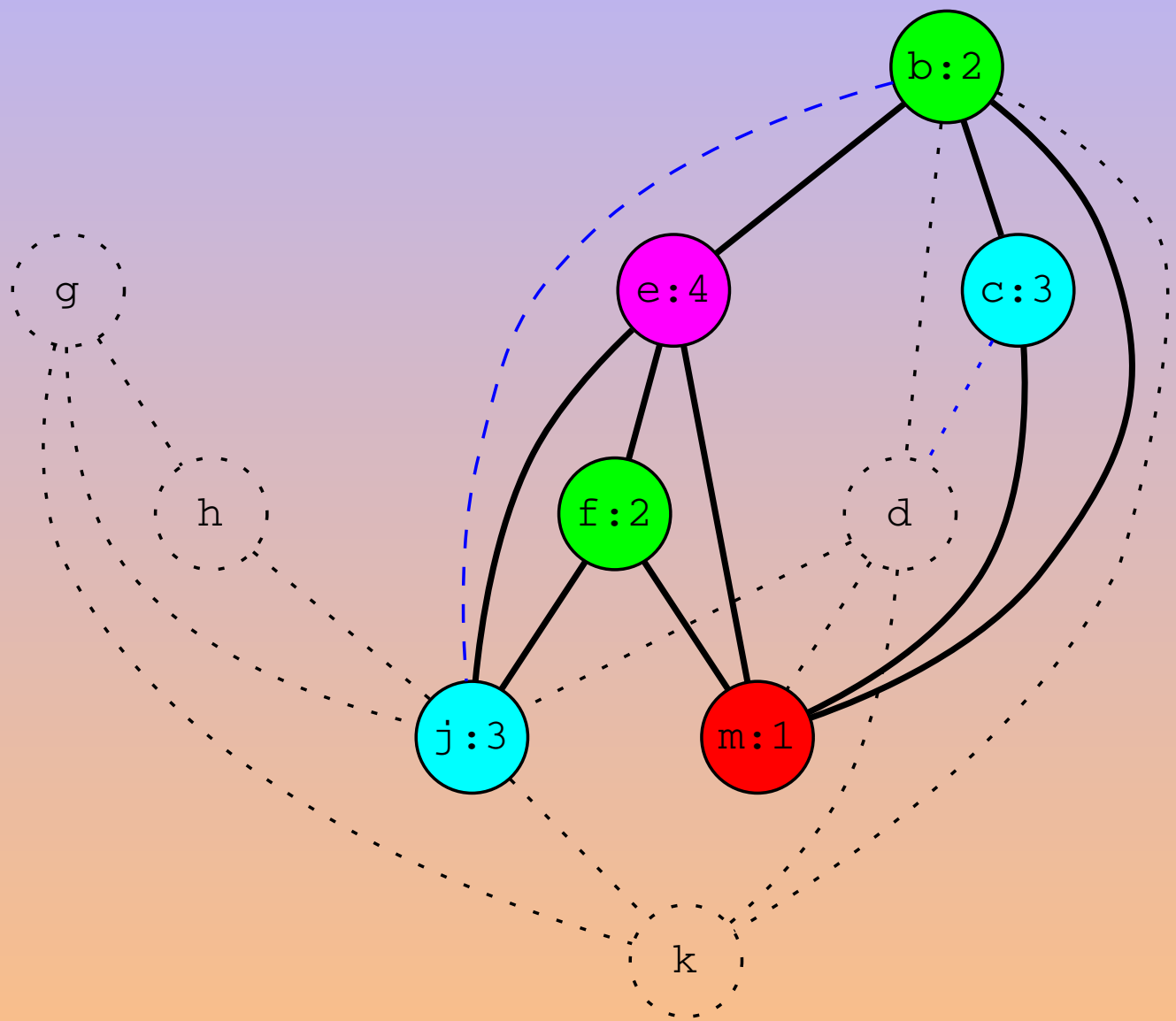
Interference Graph: Color 5

e
j
d
k
h
g



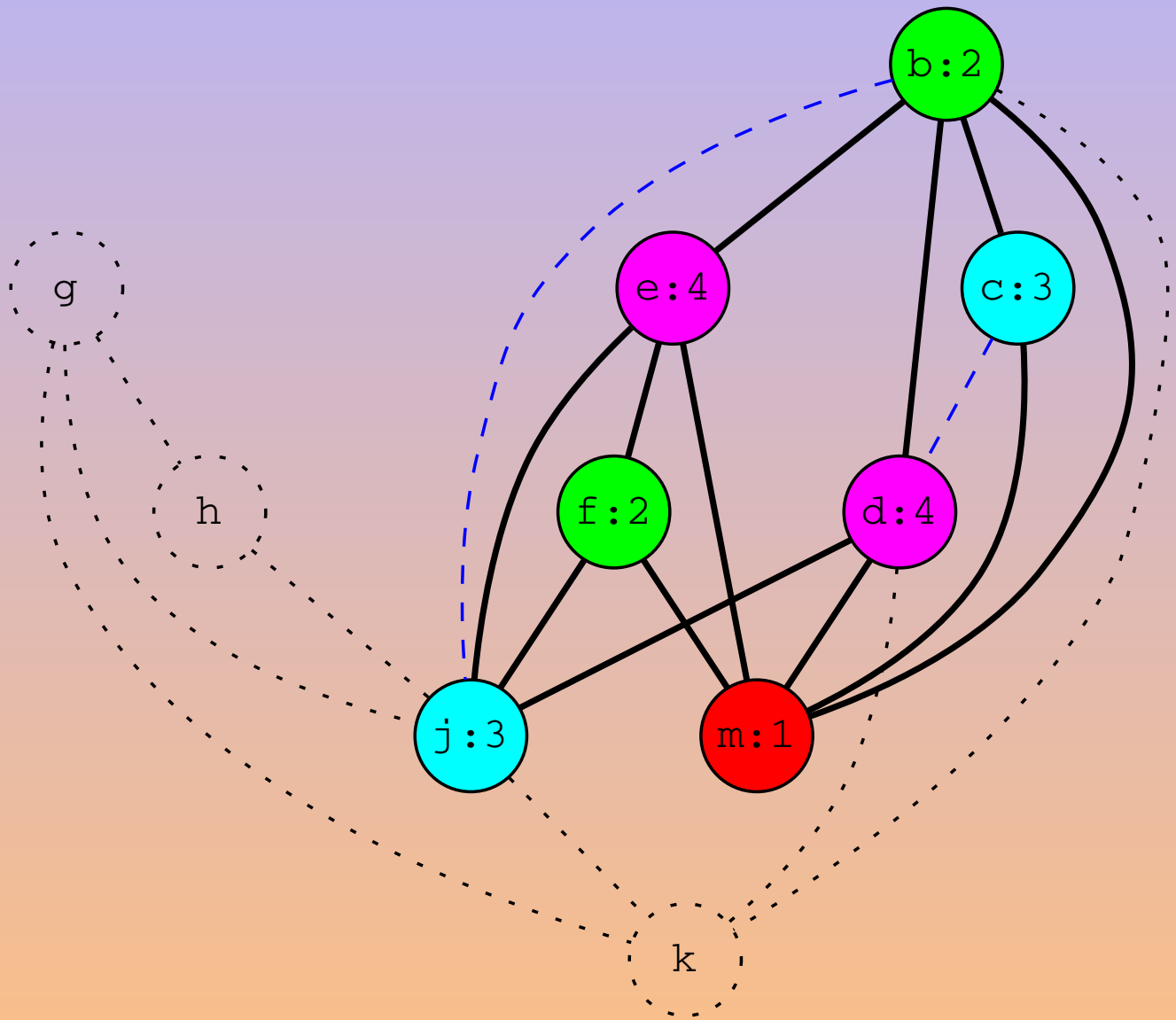
Interference Graph: Color 4

j
d
k
h
g

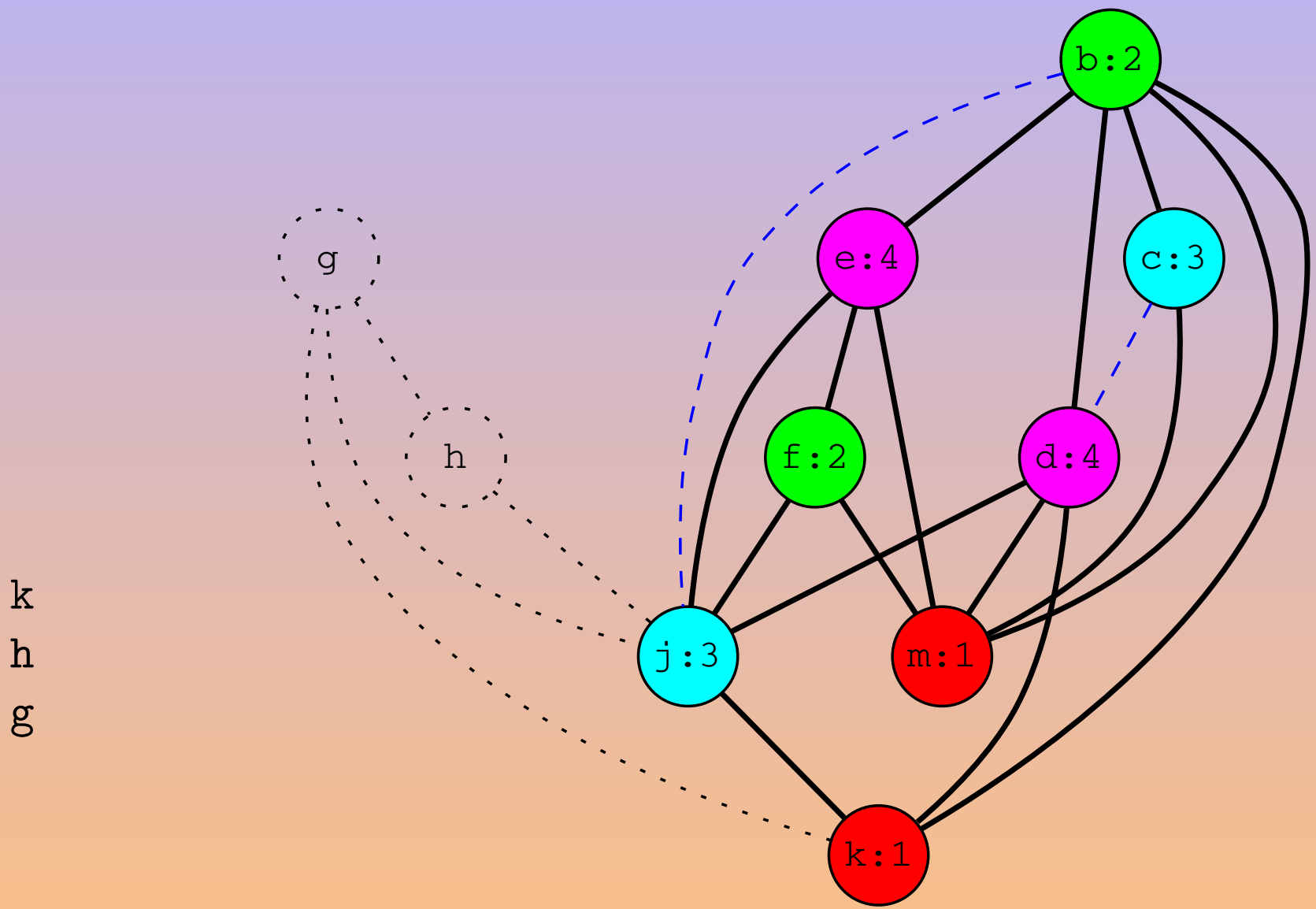


Interference Graph: Color 3

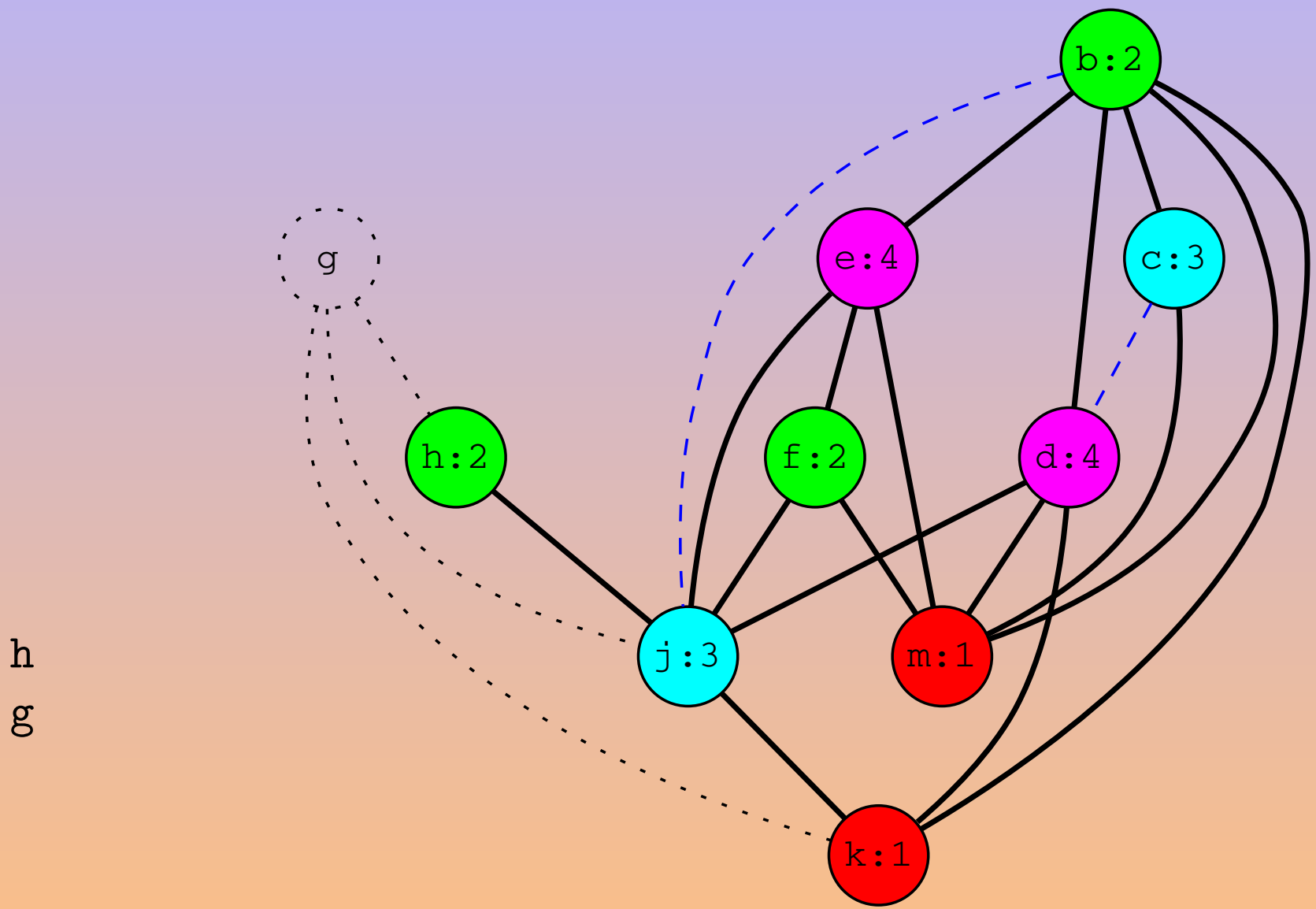
d
k
h
g



Interference Graph: Color 2

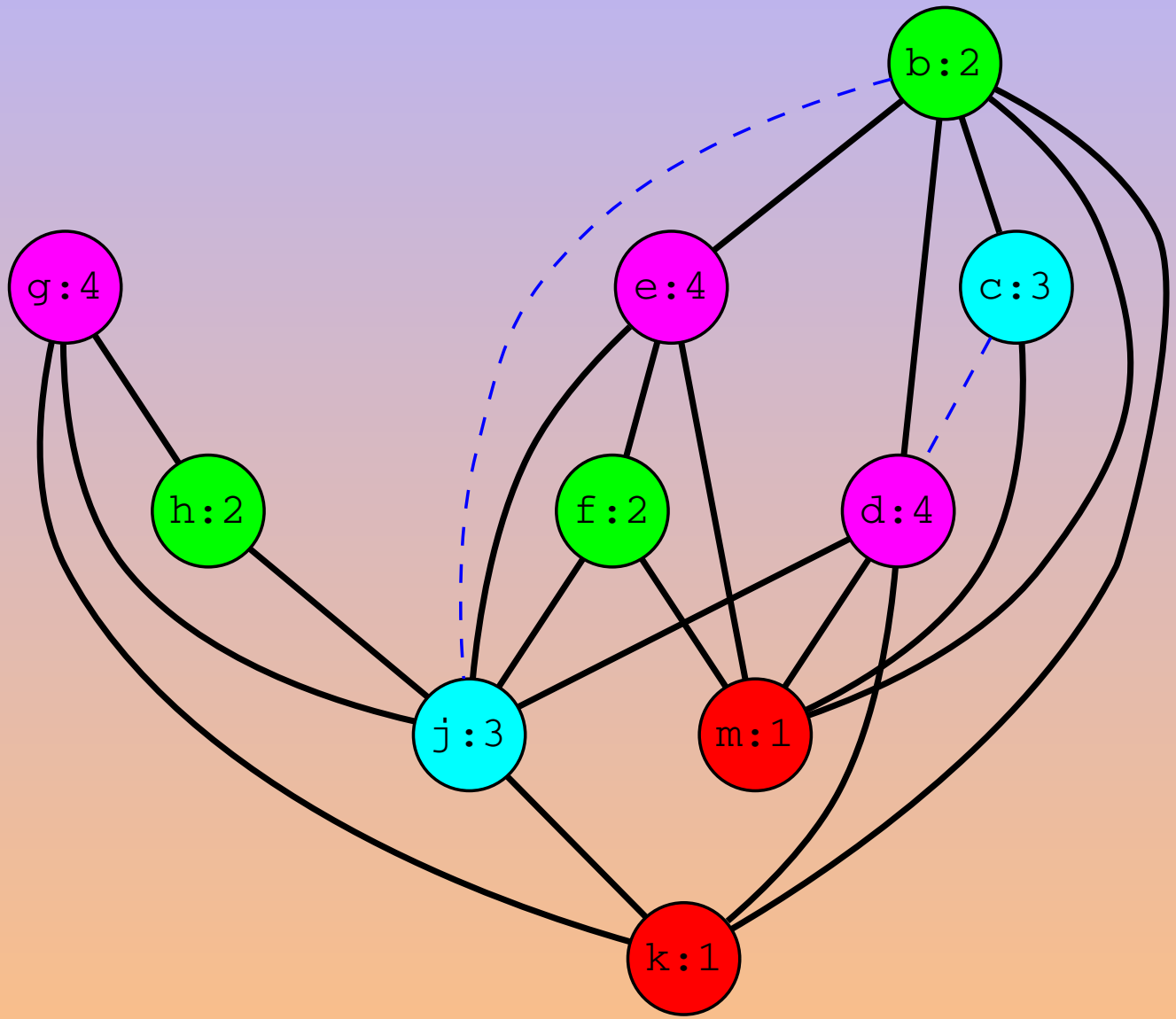


Interference Graph: Color 1



h
g

Interference Graph: Color 0



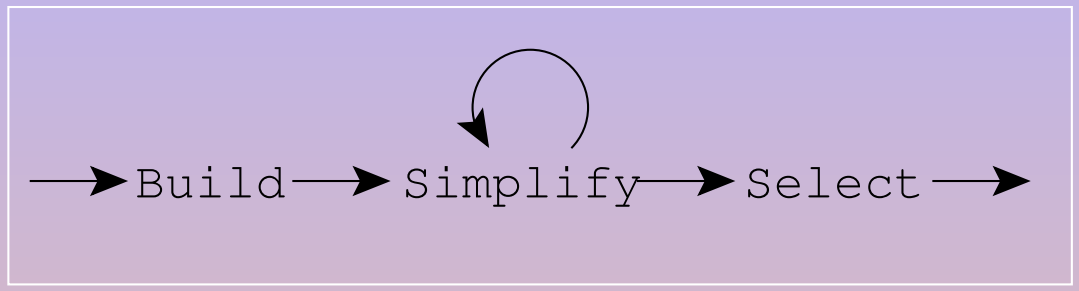
g

Result

```
live in: k j
  g := [j + 12]
  h := k - 1
  f := g * h
  e := [j + 8]
  m := [j + 16]
  b := [f]
  c := e + 8
  d := c
  k := m + 4
  j := b
live out: d k j
```

```
live in: r1 r3
  r4 := [r3 + 12]
  r2 := r1 - 1
  r2 := r4 * r2
  r4 := [r3 + 8]
  r1 := [r3 + 16]
  r2 := [r2]
  r3 := r4 + 8
  r4 := r3
  r1 := r1 + 4
  r3 := r2
live out: r4 r1 r3
```

Simple Register Allocation

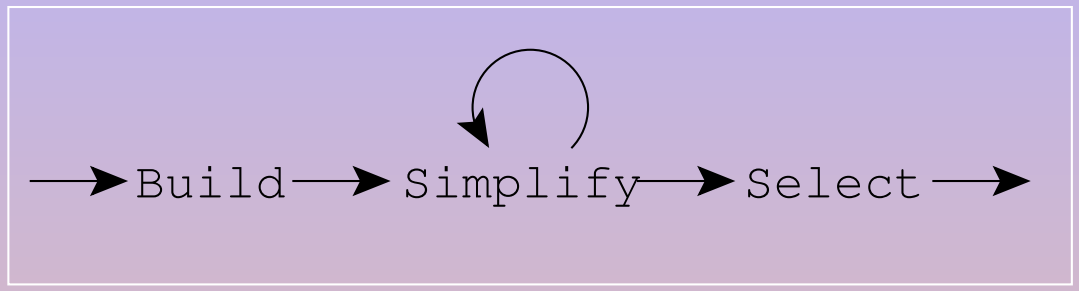


build the conflict graph from the program

simplify the nodes with insignificant degree

select (or color) while rebuilding the graph.

Simple Register Allocation

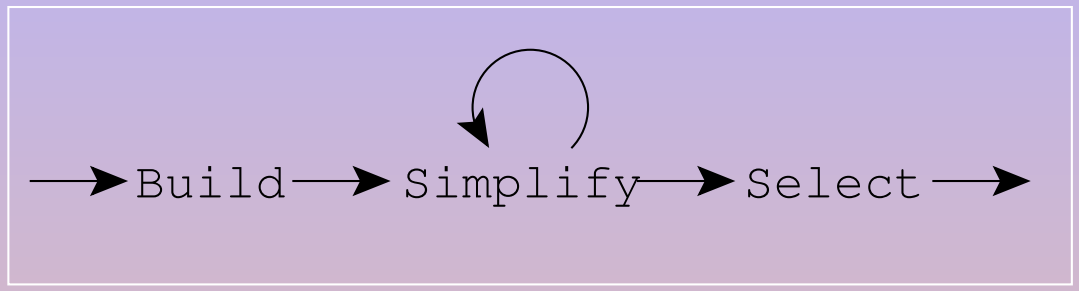


build the conflict graph from the program

simplify the nodes with insignificant degree

select (or color) while rebuilding the graph.

Simple Register Allocation

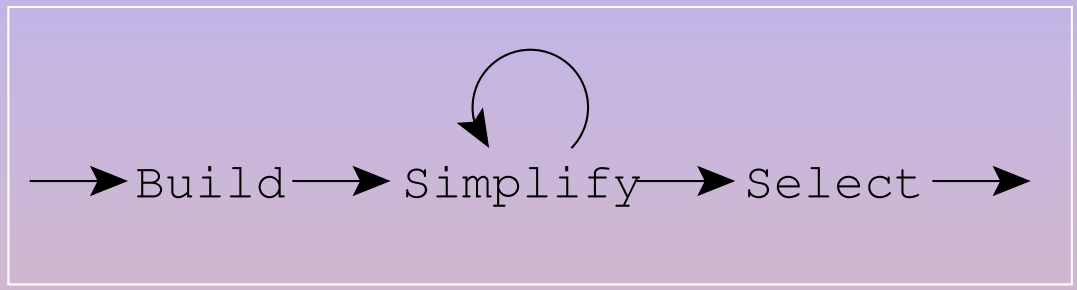


build the conflict graph from the program

simplify the nodes with insignificant degree

select (or color) while rebuilding the graph.

Simple Register Allocation



build the conflict graph from the program

simplify the nodes with insignificant degree

select (or color) while rebuilding the graph.

Based on:

> *A.B. Kempe. On the Geographical problem of the four colors, Am. J. Math 2, 193–200, 1879.*

[Appel, 1998, Matz, 2003]

Yes, but What Color? [Matz, 2003]

- Usually, first-fit (registers are ordered).
- Trying caller save first helps.
- **Biased Coloring.** [Briggs, 1992]
  Use a color already unavailable to our neighbors.

Yes, but What Color? [Matz, 2003]

- Usually, first-fit (registers are ordered).
- Trying caller save first helps.
- **Biased Coloring.** [Briggs, 1992]
  Use a color already unavailable to our neighbors.

Yes, but What Color? [Matz, 2003]

- Usually, first-fit (registers are ordered).
- Trying caller save first helps.
- **Biased Coloring.** [Briggs, 1992]
  Use a color already unavailable to our neighbors.

Spilling

Spilling

A map can always be colored with 4 colors. . .

Spilling

A map can always be colored with 4 colors...

But for graph coloring, there is no reason for:

- this simple heuristics to always find a solution,
- a solution to always exist...

Spilling

A map can always be colored with 4 colors...

But for graph coloring, there is no reason for:

- this simple heuristics to always find a solution,
- a solution to always exist...

Spilling

- Not enough registers
  `t1 := t1 + t2`
- So use the stack
  `[sp + 4] := [sp + 4] + [sp + 8]`
- But use temporaries to do so!
  ```
  t12 := [sp + 4]
  t13 := [sp + 8]
  t12 := t12 + t13
  [sp + 4] := t12
  ```
- Why should it solve the problem?

Spilling

- Not enough registers
  `t1 := t1 + t2`
- So use the stack
  `[sp + 4] := [sp + 4] + [sp + 8]`
- But use temporaries to do so!
  ```
  t12 := [sp + 4]
  t13 := [sp + 8]
  t12 := t12 + t13
  [sp + 4] := t12
  ```
- Why should it solve the problem?

Spilling

- Not enough registers

```
t1 := t1 + t2
```

- So use the stack

```
[sp + 4] := [sp + 4] + [sp + 8]
```

- But use temporaries to do so!

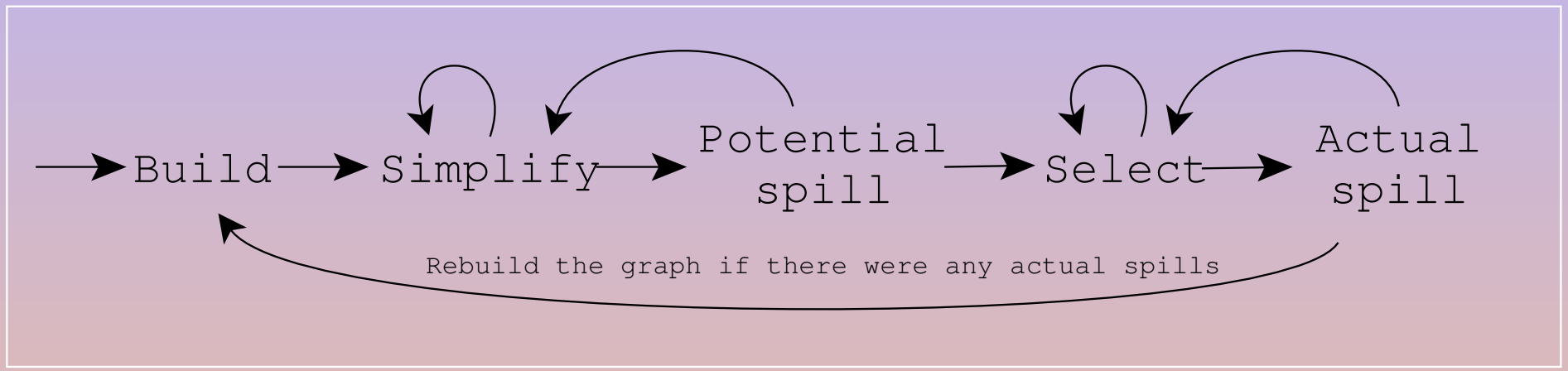
```
t12 := [sp + 4]
t13 := [sp + 8]
t12 := t12 + t13
[sp + 4] := t12
```

- Why should it solve the problem?

Spilling

- Not enough registers
  ```
  t1 := t1 + t2
  ```
- So use the stack
  ```
  [sp + 4] := [sp + 4] + [sp + 8]
  ```
- But use temporaries to do so!
  ```
  t12 := [sp + 4]
  t13 := [sp + 8]
  t12 := t12 + t13
  [sp + 4] := t12
  ```
- Why should it solve the problem?

Register Allocation with Spills



spill when one cannot simplify, the (uses of the) temporary must be rewritten using the stack.

rebuild but then, the conflict graph is to be rewritten

[Appel, 1998, Matz, 2003]

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

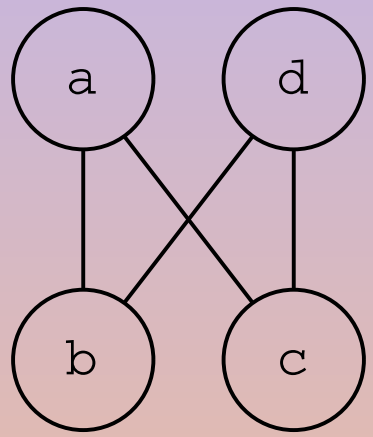
- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Optimistic Coloring

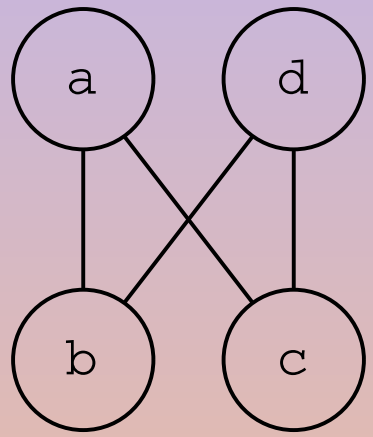
- We miss many opportunities to avoid the stack



- Handle spills as if they were simplified (*potential spills*)
- then try to color them
- There might not be *actual spills*

Optimistic Coloring

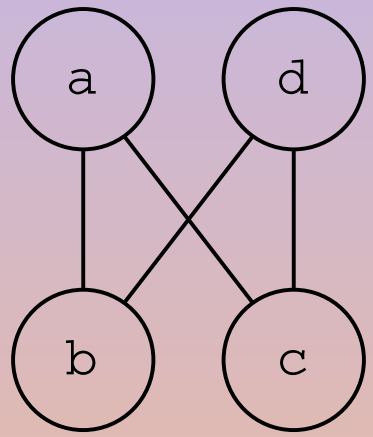
- We miss many opportunities to avoid the stack



- Handle spills as if they were simplified (*potential spills*)
- then try to color them
- There might not be *actual spills*

Optimistic Coloring

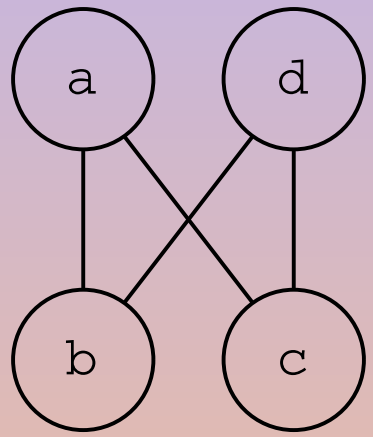
- We miss many opportunities to avoid the stack



- Handle spills as if they were simplified (*potential spills*)
- then try to color them
- There might not be *actual spills*

Optimistic Coloring

- We miss many opportunities to avoid the stack



- Handle spills as if they were simplified (*potential spills*)
- then try to color them
- There might not be *actual spills*

Coalescing

Coalescing

- Some low-level form of *copy propagation*
- While building traces we tried to remove `jumps`
- While allocating registers, we try to remove `moves`

live-in: `t2`

```
t1 := ...
t2 := t1 + t2
t3 := t2
t4 := t1 + t3
t2 := t3 + t4
t1 := t2 - t4
```

live-out: `t1`

Coalescing

- Some low-level form of *copy propagation*
- While building traces we tried to remove `jumps`
- While allocating registers, we try to remove `moves`

live-in: `t2`

```
t1 := ...
t2 := t1 + t2
t3 := t2
t4 := t1 + t3
t2 := t3 + t4
t1 := t2 - t4
```

live-out: `t1`

Coalescing

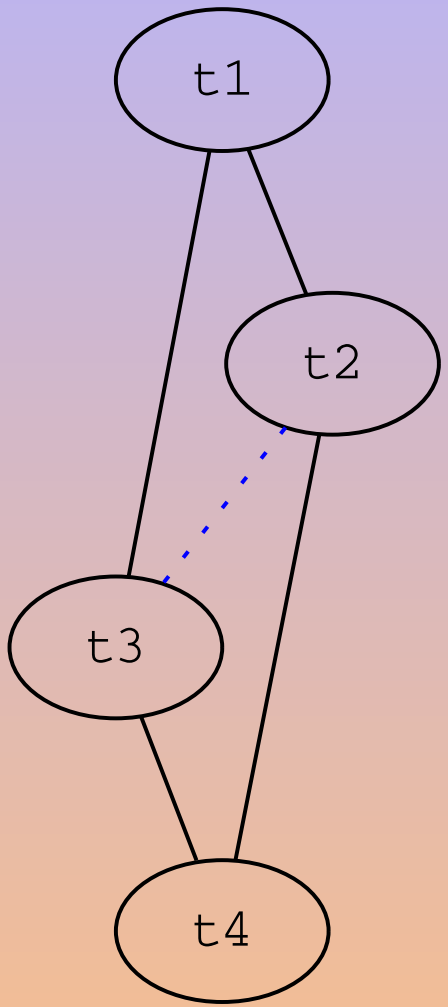
- Some low-level form of *copy propagation*
- While building traces we tried to remove `jumps`
- While allocating registers, we try to remove `moves`

  live-in: `t2`

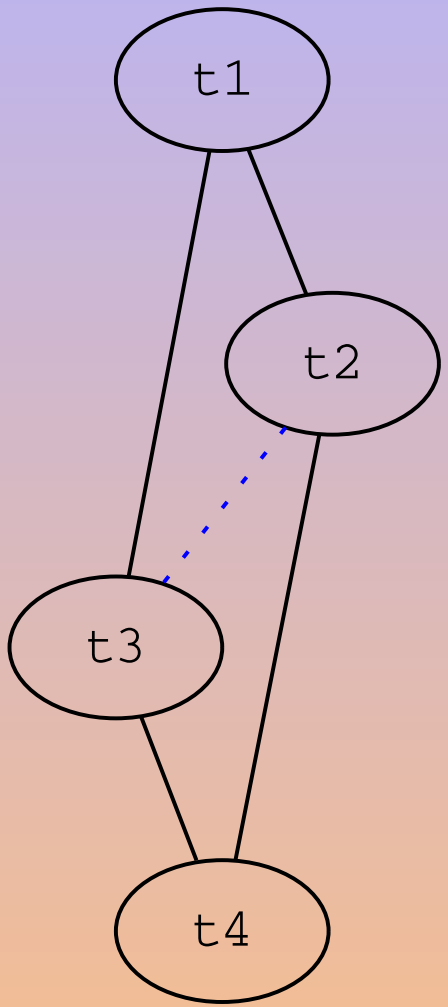
```
t1 := ...
t2 := t1 + t2
t3 := t2
t4 := t1 + t3
t2 := t3 + t4
t1 := t2 - t4
```

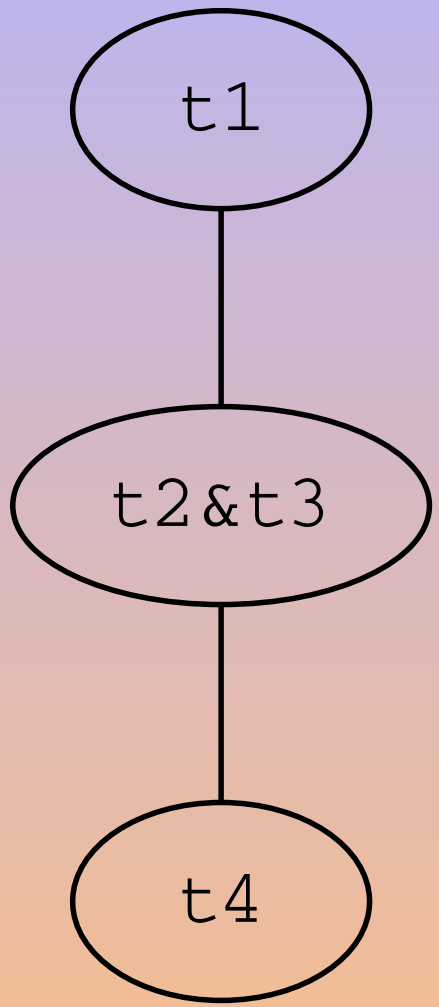
  live-out: `t1`

Coalescing Improves the Coloralibility

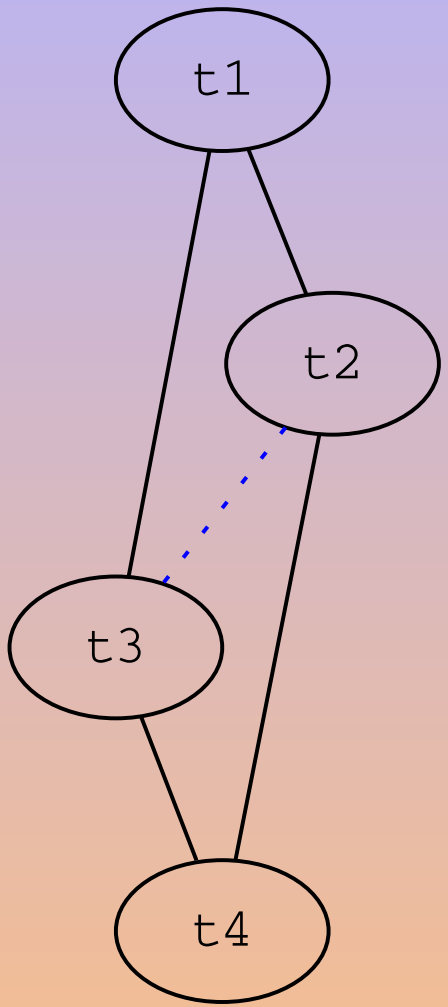


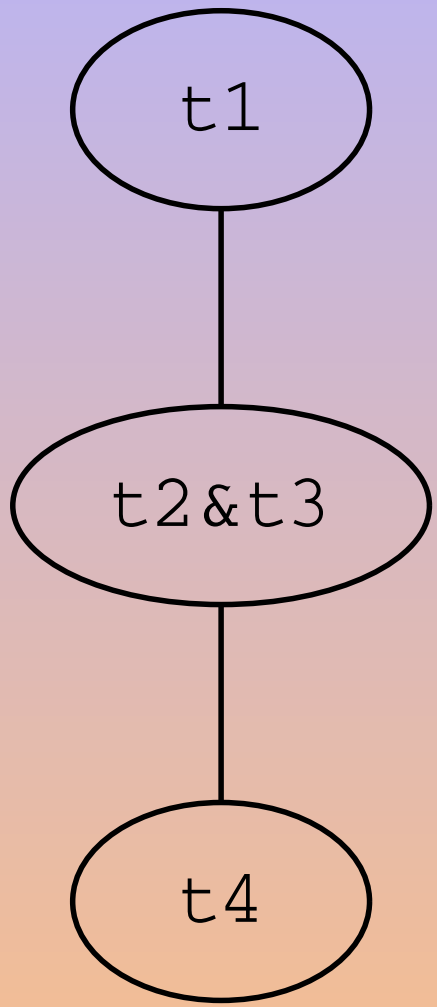
Coalescing Improves the Coloralibility





Coalescing Improves the Coloralibility





t1 and t4 have one neighbor less!

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if
  - Briggs: ab has fewer than k neighbors of significant degree
  - George: every neighbor of a is
    - of insignificant degree
    - already interfering with b
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if
  - Briggs ab has fewer than k neighbors of significant degree.
  - George every neighbor of a is
    - of insignificant degree
    - already interfering with b
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if

  Briggs ab has fewer than k neighbors of significant degree.

  George every neighbor of a is
  - of insignificant degree
  - already interfering with b

- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if

  Briggs ab has fewer than k neighbors of significant degree.
  George every neighbor of a is
  - of insignificant degree
  - already interfering with b
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if

  Briggs ab has fewer than k neighbors of significant degree.
  George every neighbor of a is
  - of insignificant degree
  - already interfering with b
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if

  Briggs ab has fewer than k neighbors of significant degree.
  George every neighbor of a is
  - of insignificant degree
  - already interfering with b
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

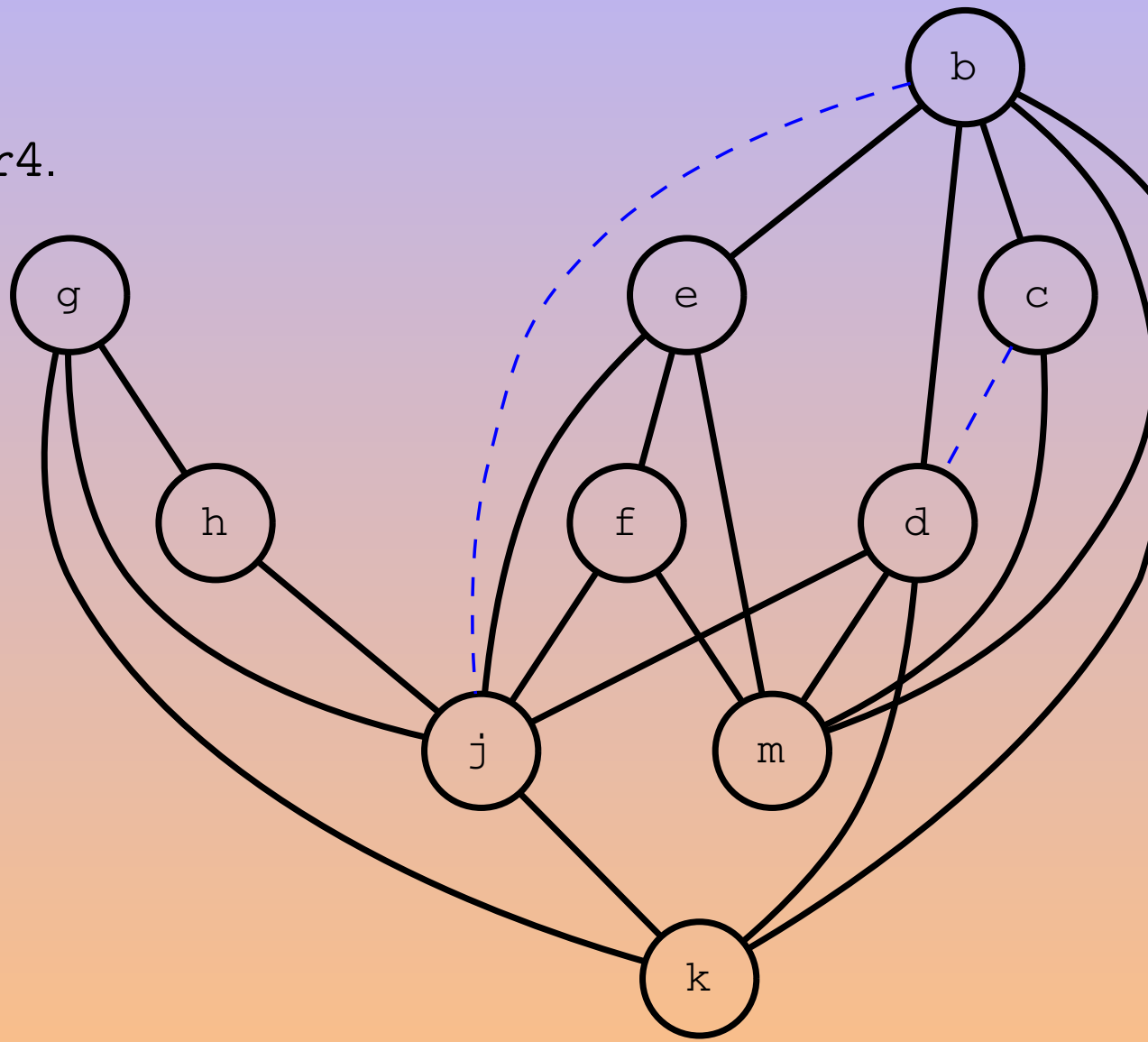
- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if

  Briggs ab has fewer than k neighbors of significant degree.
  George every neighbor of a is
  - of insignificant degree
  - already interfering with b
- George's criterion is well suited for real registers

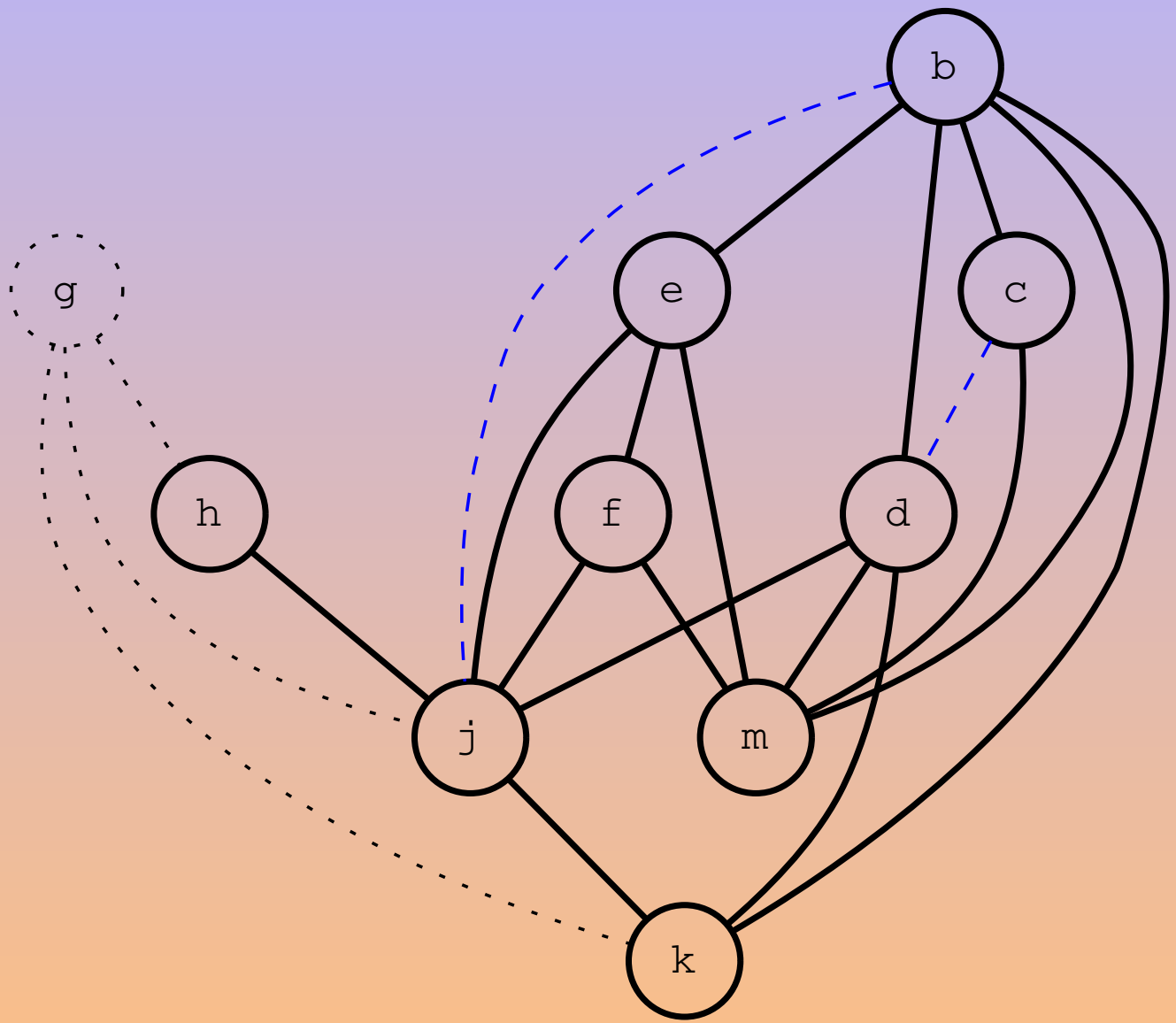
Interference Graph [Appel, 1998]

Four registers: r1, r2, r3, r4.

```
live in: k j
  g := [j + 12]
  h := k - 1
  f := g * h
  e := [j + 8]
  m := [j + 16]
  b := [f]
  c := e + 8
  d := c
  k := m + 4
  j := b
live out: d k j
```

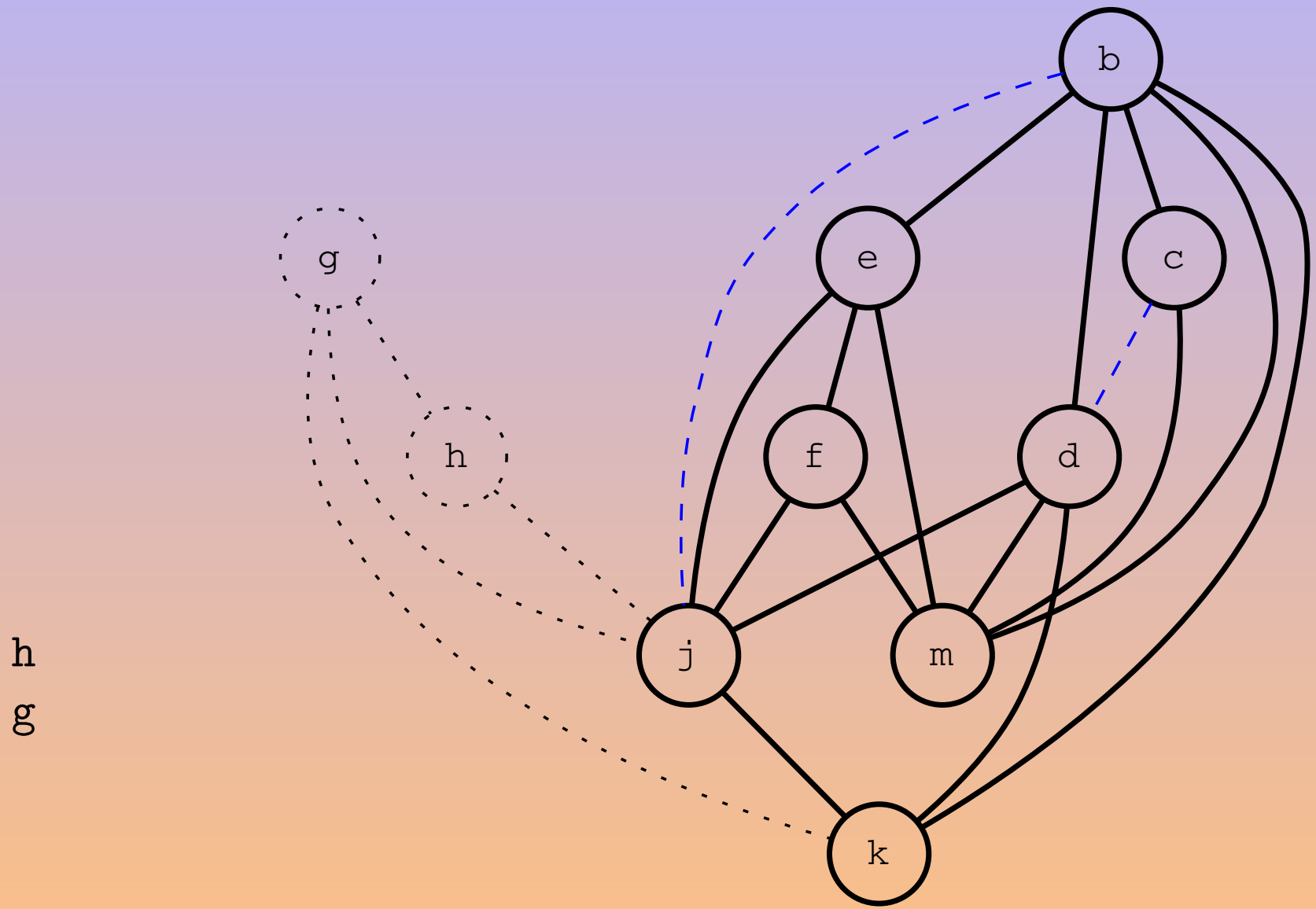


Interference Graph: Simplify 0



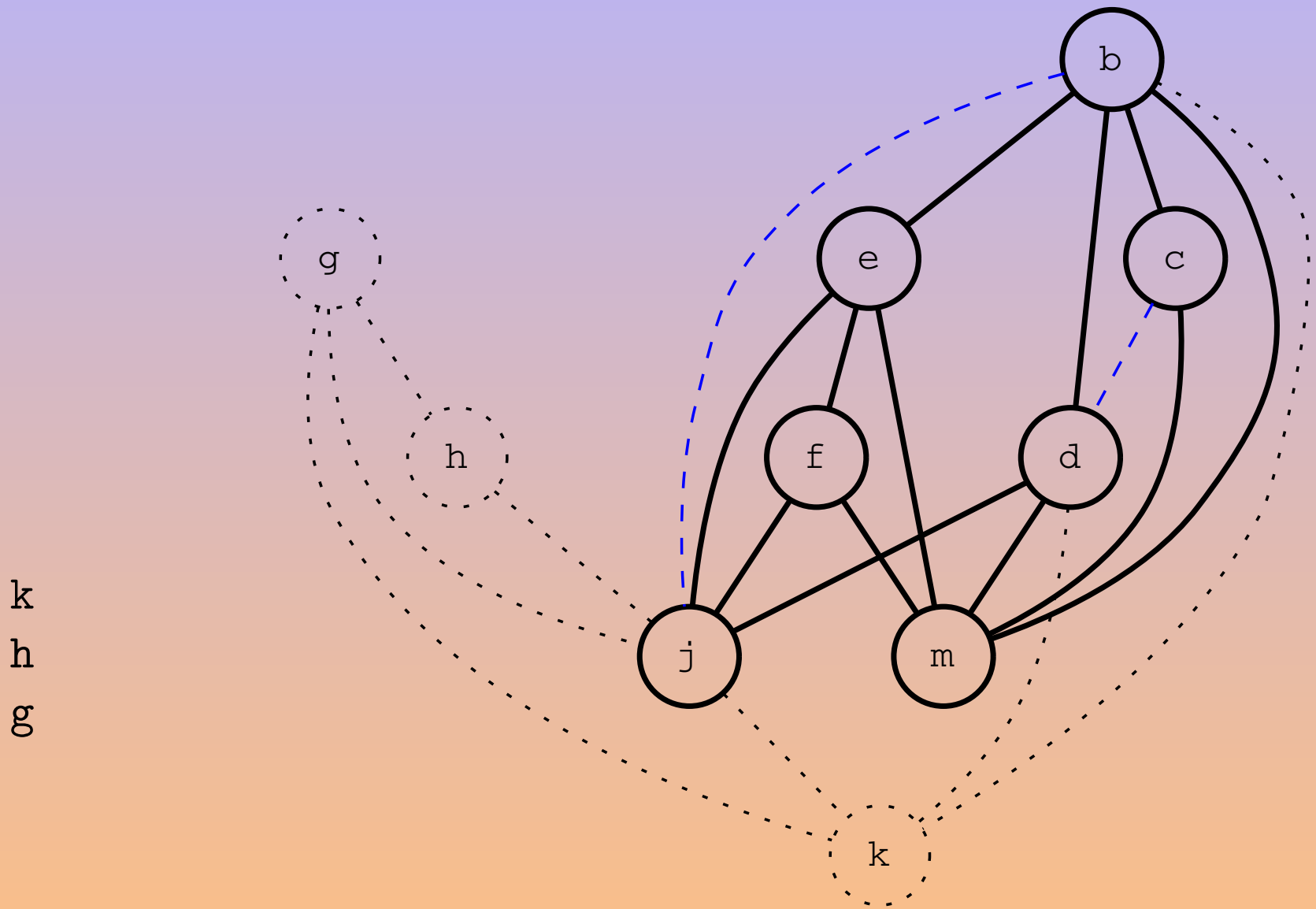
g

Interference Graph: Simplify 1



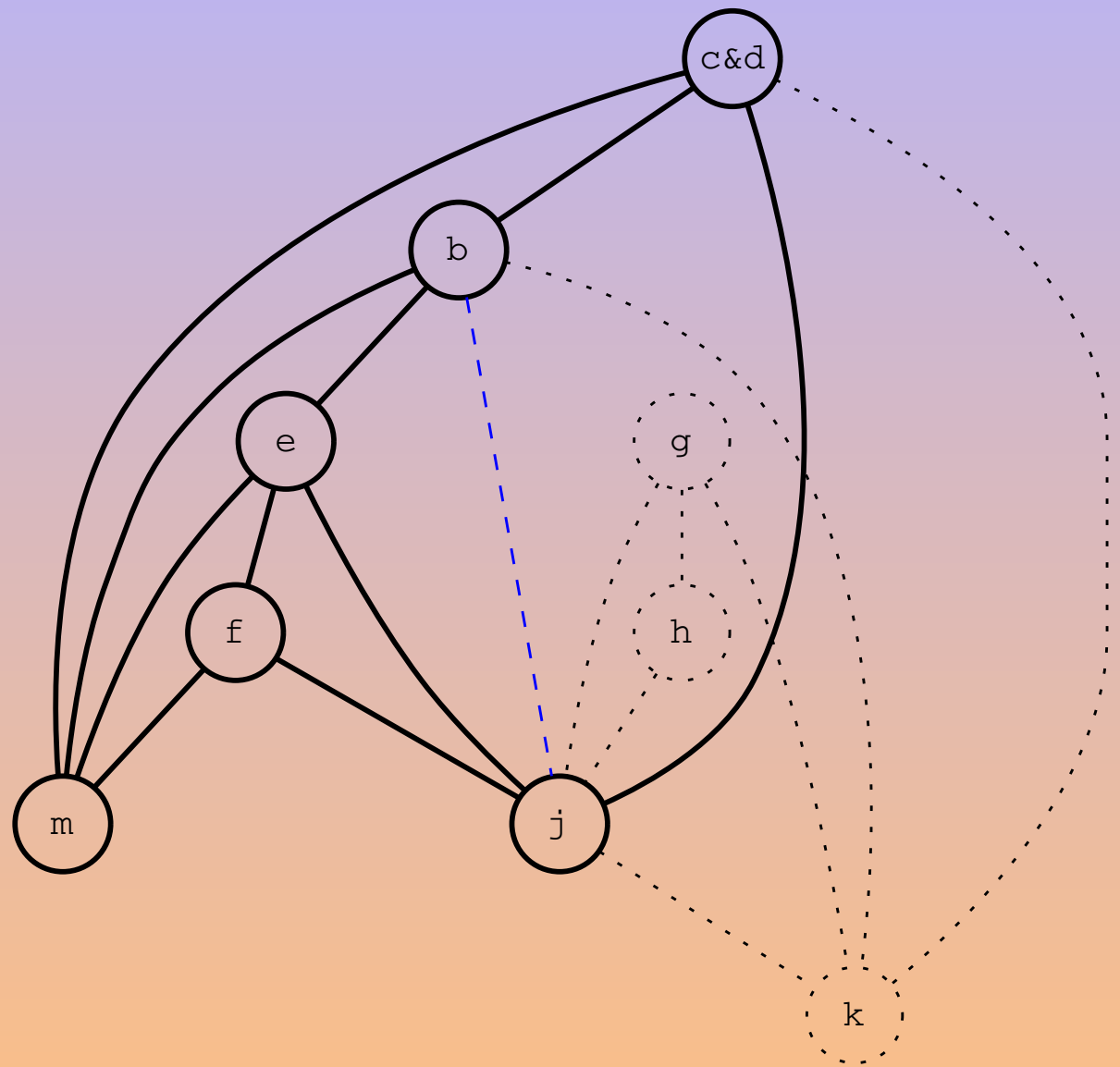
h
g

Interference Graph: Simplify 2



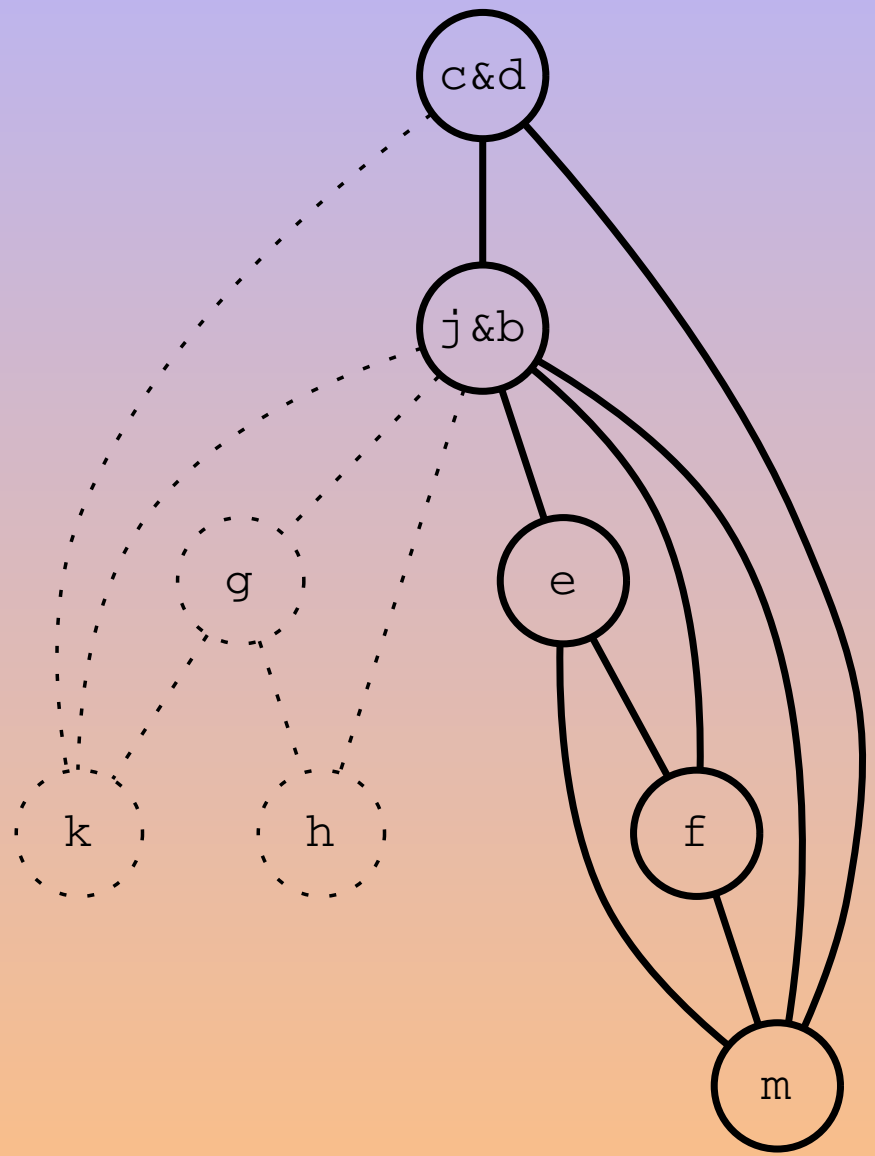
Interference Graph: Simplify 3

c&d
k
h
g



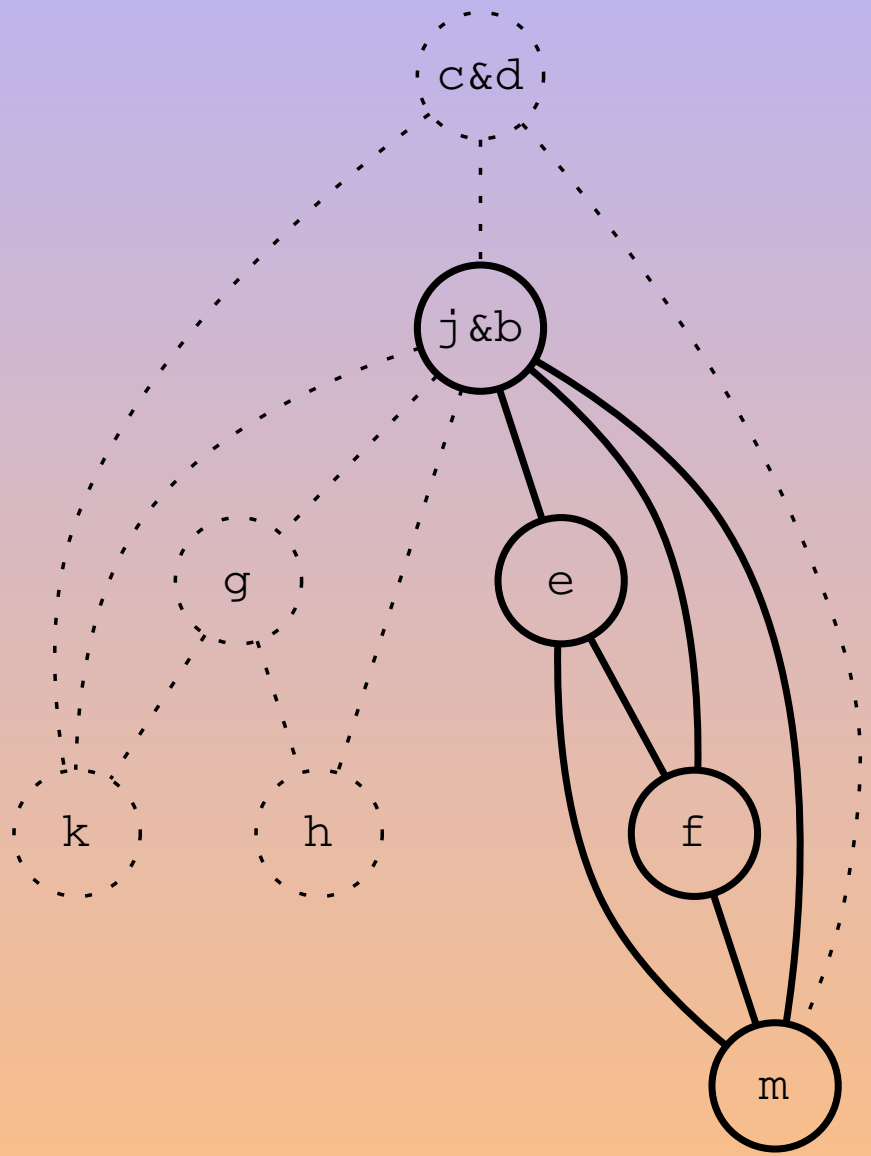
Interference Graph: Simplify 4

j&b
c&d
k
h
g



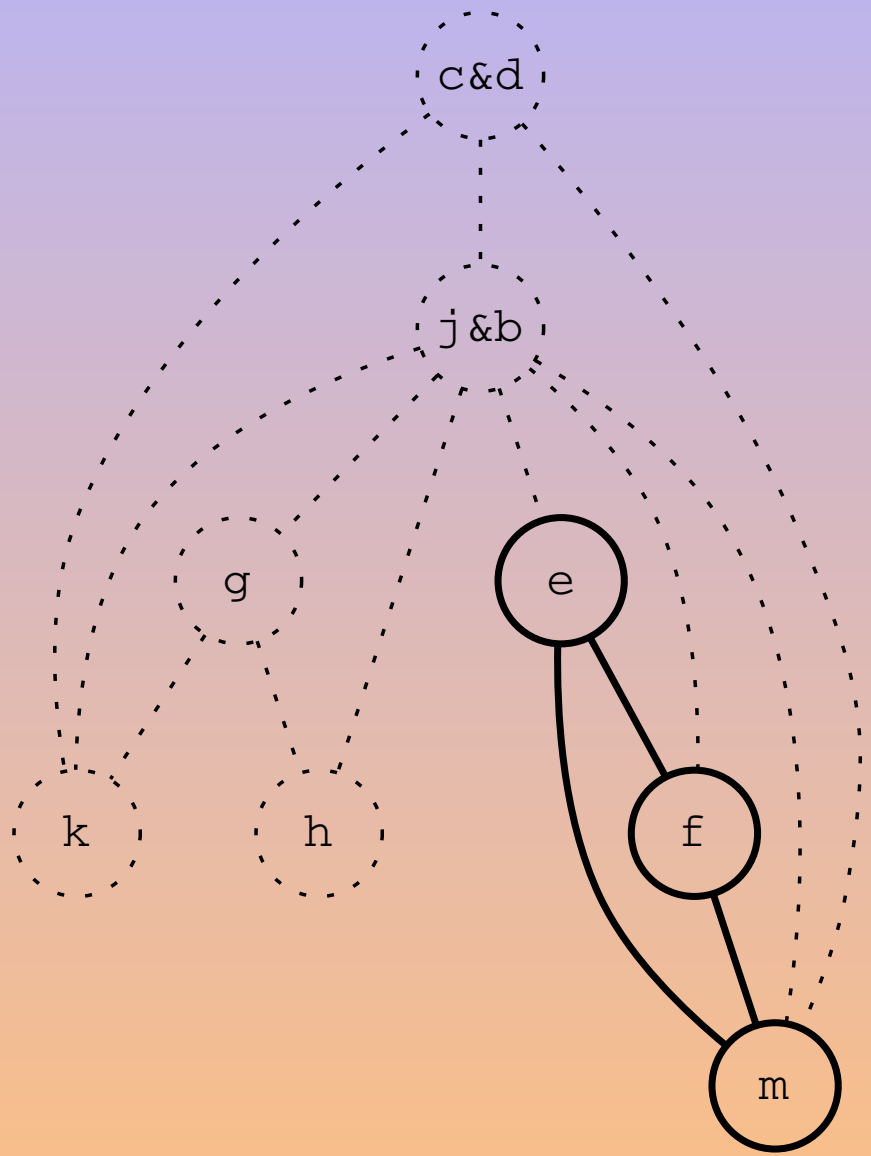
Interference Graph: Simplify 5

c&d
j&b
c&d
k
h
g



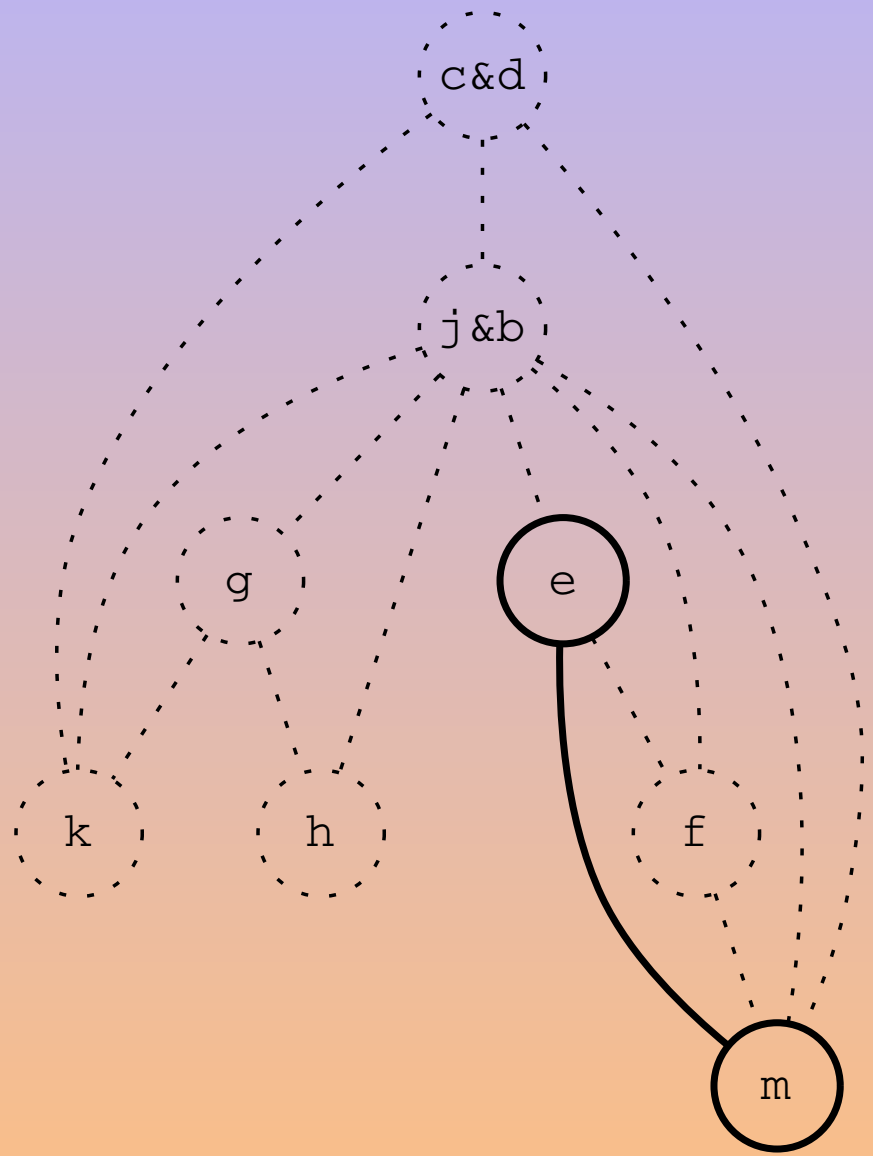
Interference Graph: Simplify 6

j&b
c&d
j&b
c&d
k
h
g



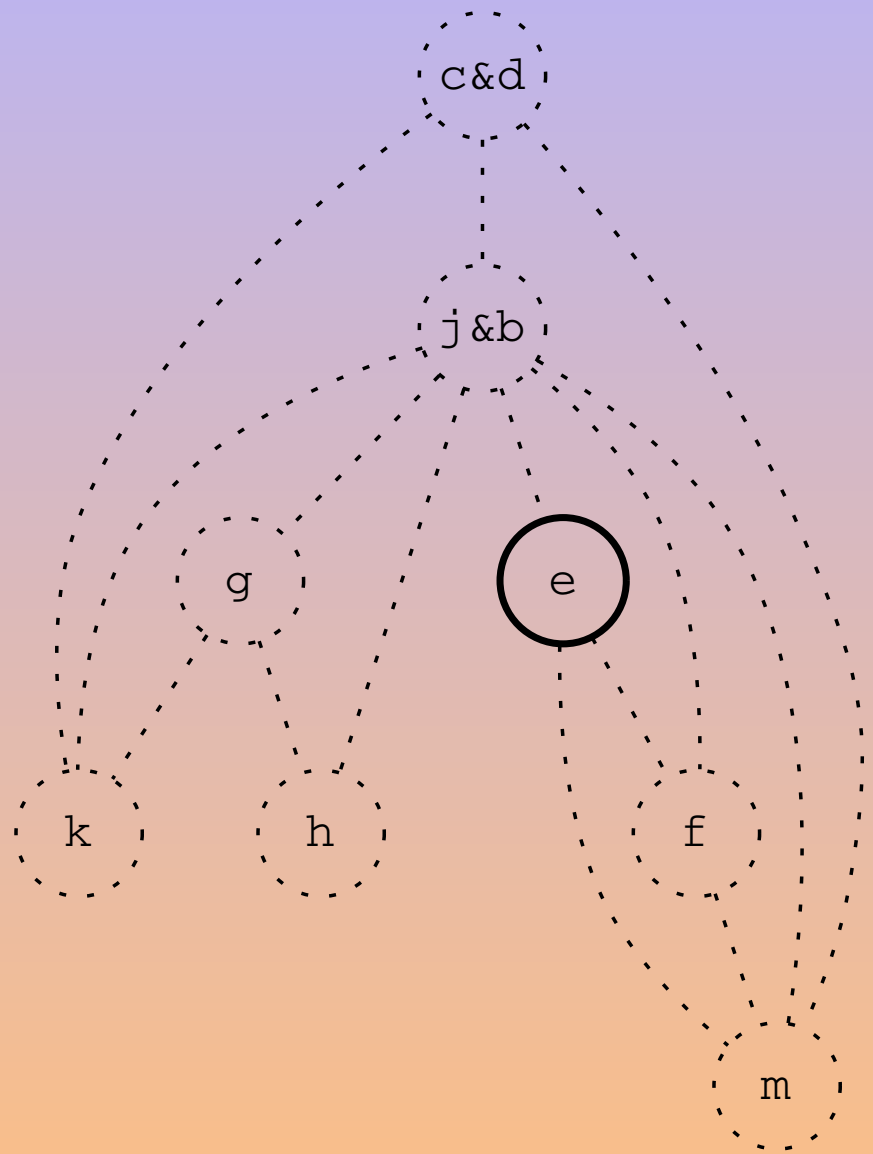
Interference Graph: Simplify 7

f
j&b
c&d
j&b
c&d
k
h
g



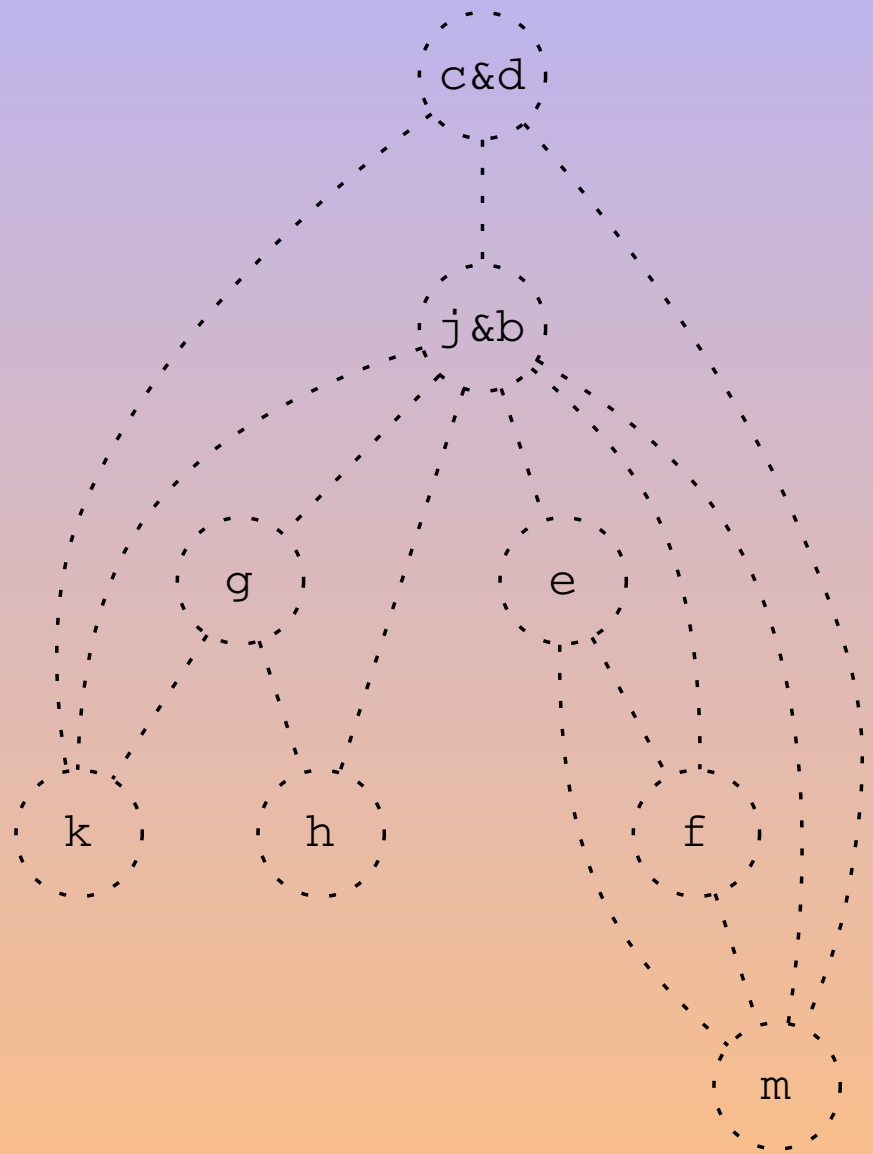
Interference Graph: Simplify 8

m
f
j&b
c&d
j&b
c&d
k
h
g



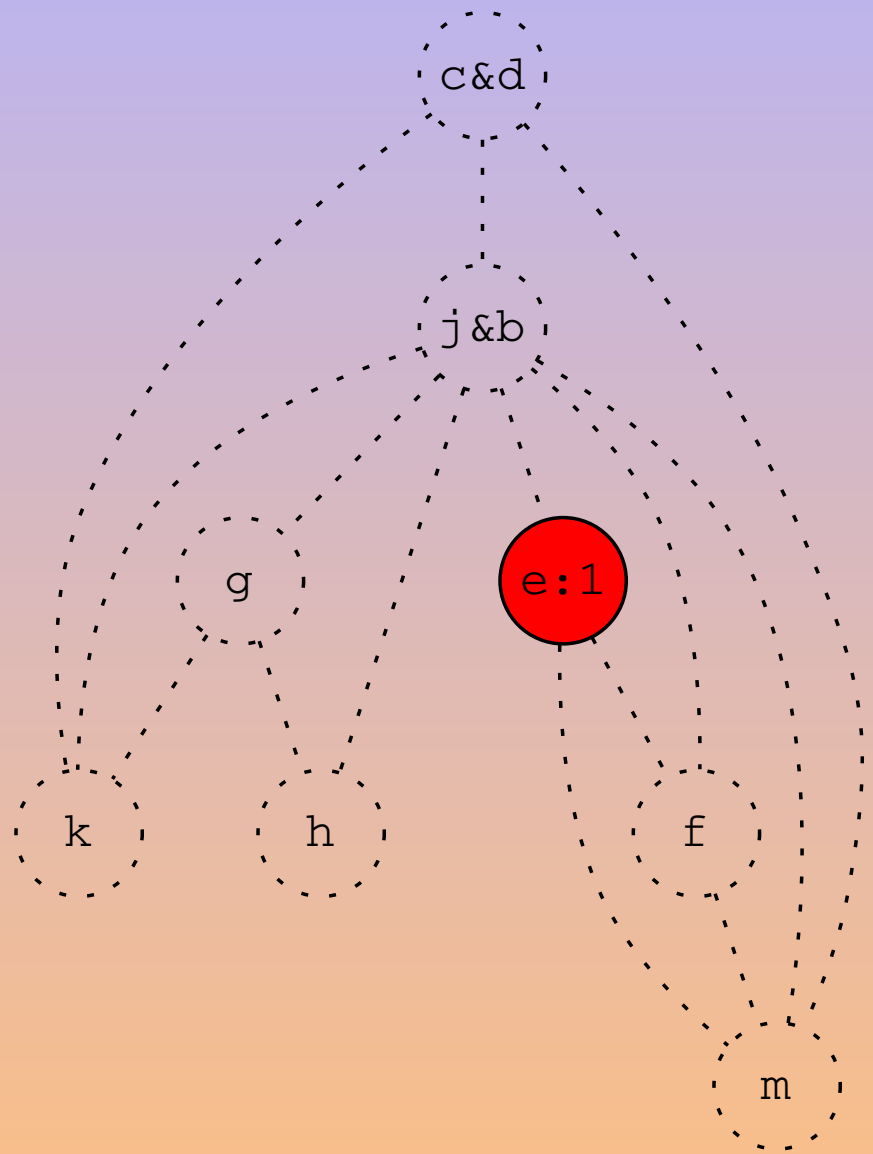
Interference Graph: Simplify 9

e
m
f
j&b
c&d
j&b
c&d
k
h
g



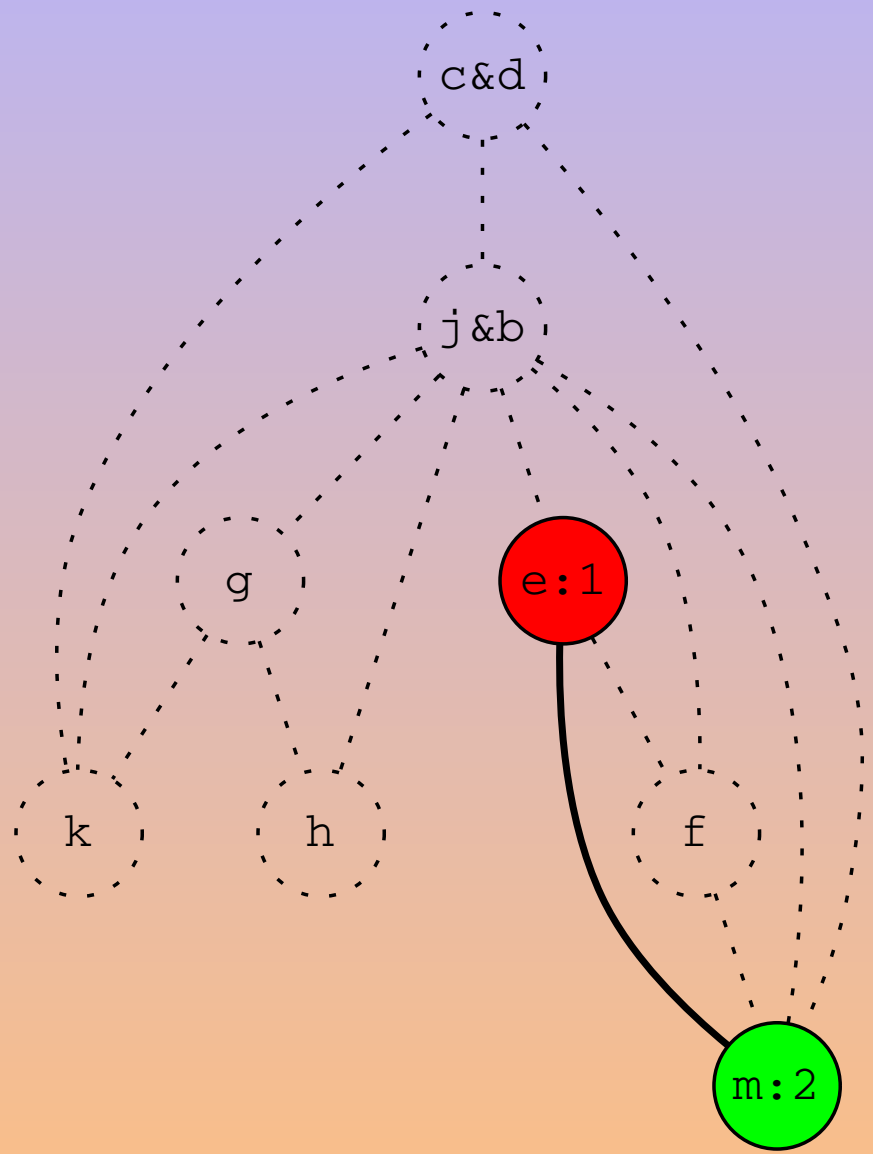
Interference Graph: Simplify 9

e
m
f
j&b
c&d
j&b
c&d
k
h
g



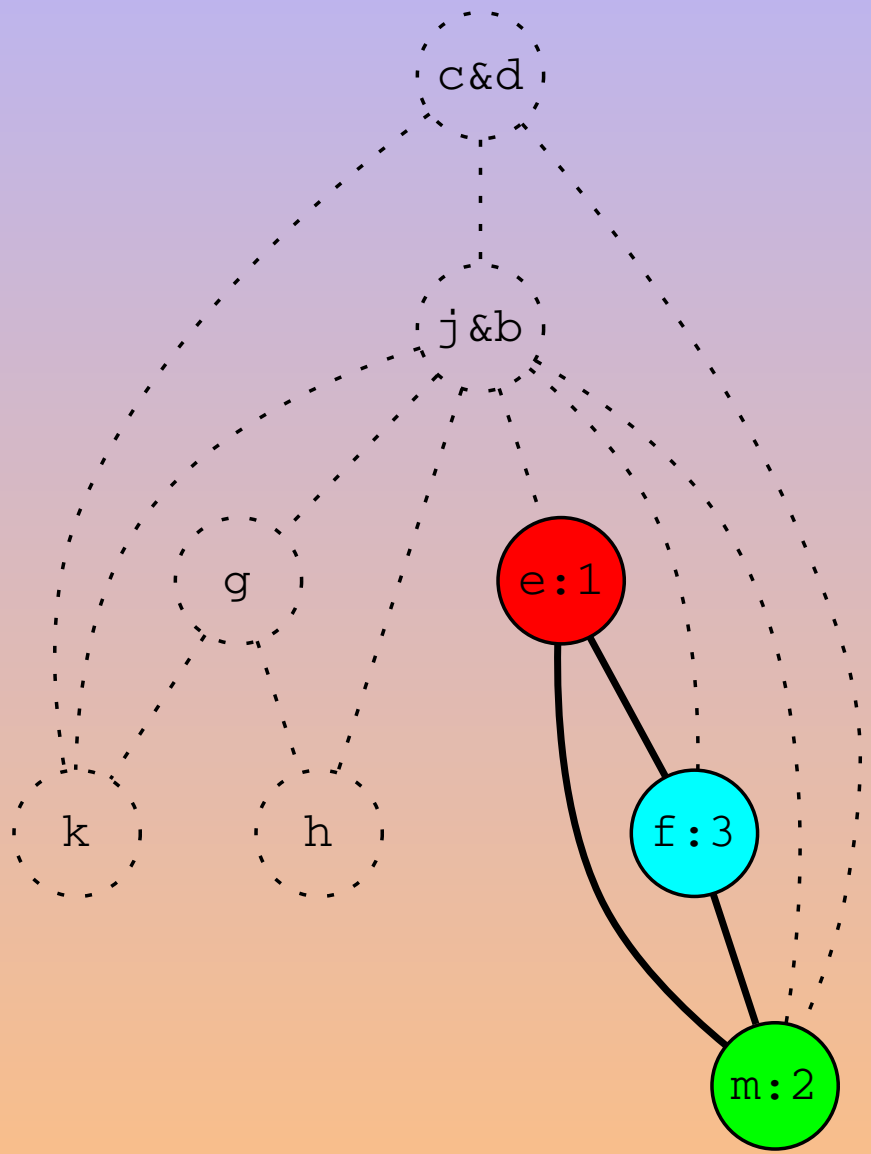
Interference Graph: Simplify 8

m
f
j&b
c&d
j&b
c&d
k
h
g



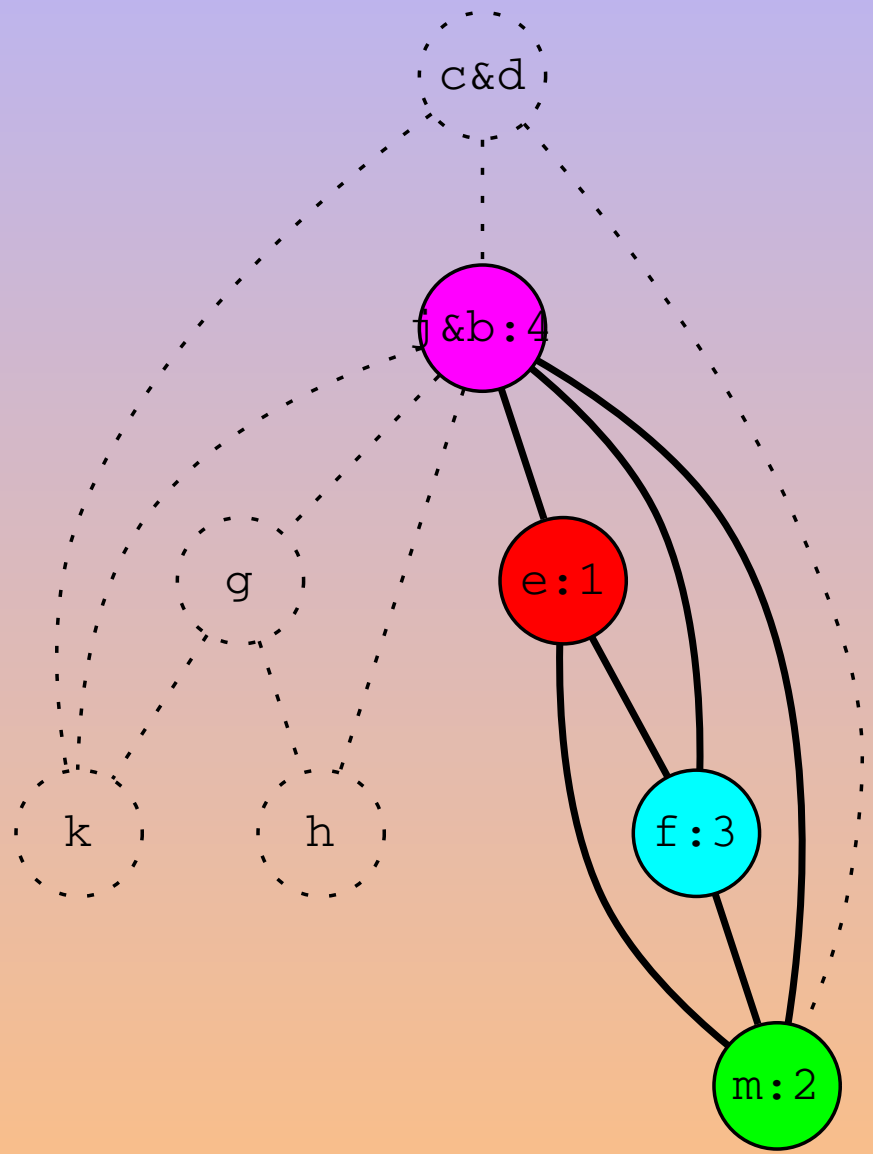
Interference Graph: Simplify 7

f
j&b
c&d
j&b
c&d
k
h
g



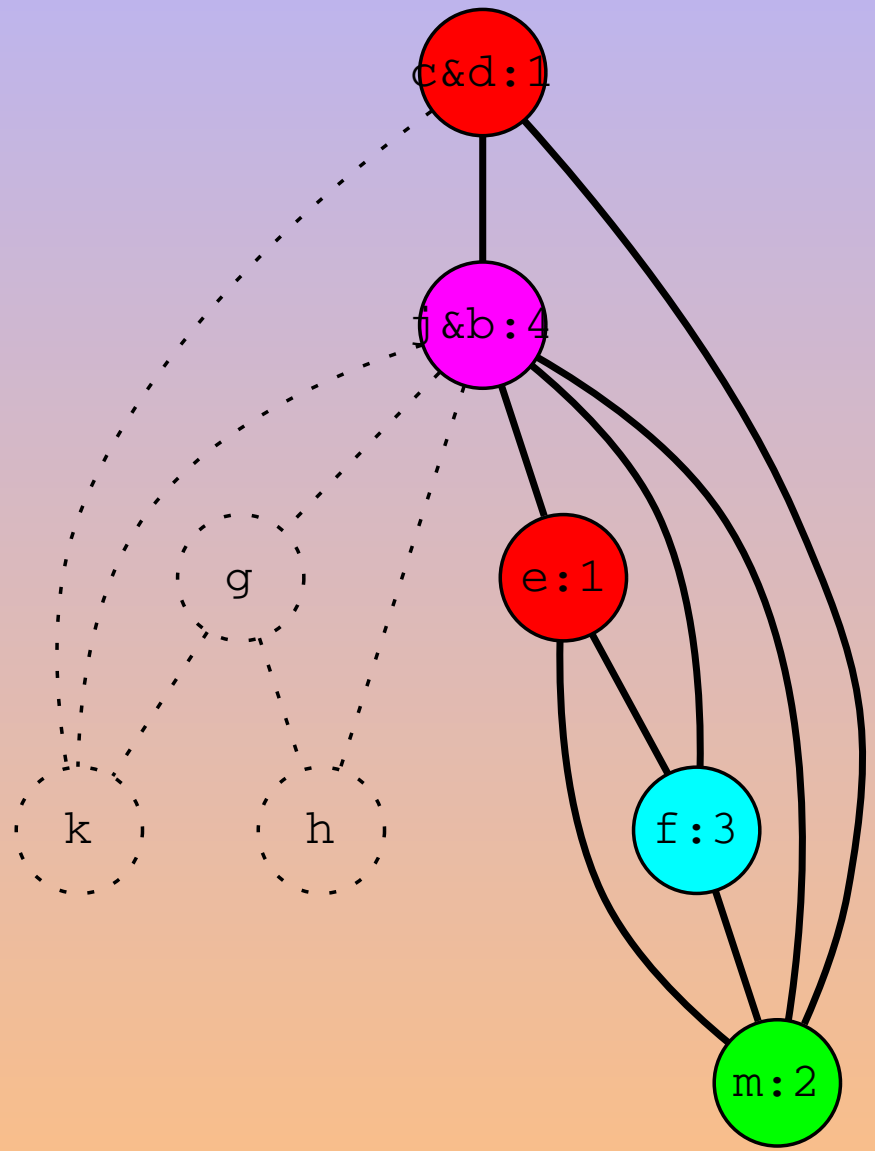
Interference Graph: Simplify 6

j&b
c&d
j&b
c&d
k
h
g



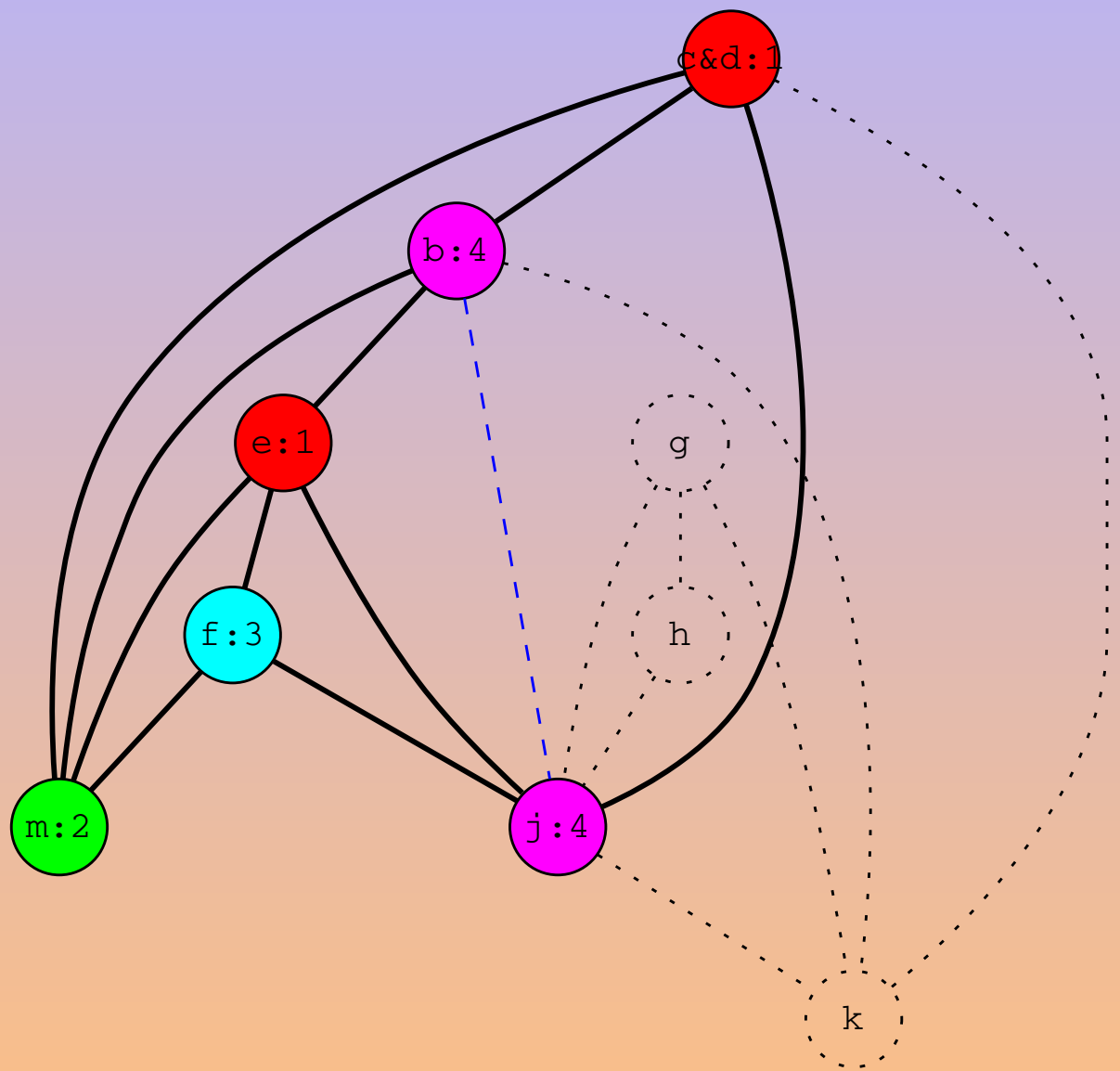
Interference Graph: Simplify 5

c&d
j&b
c&d
k
h
g



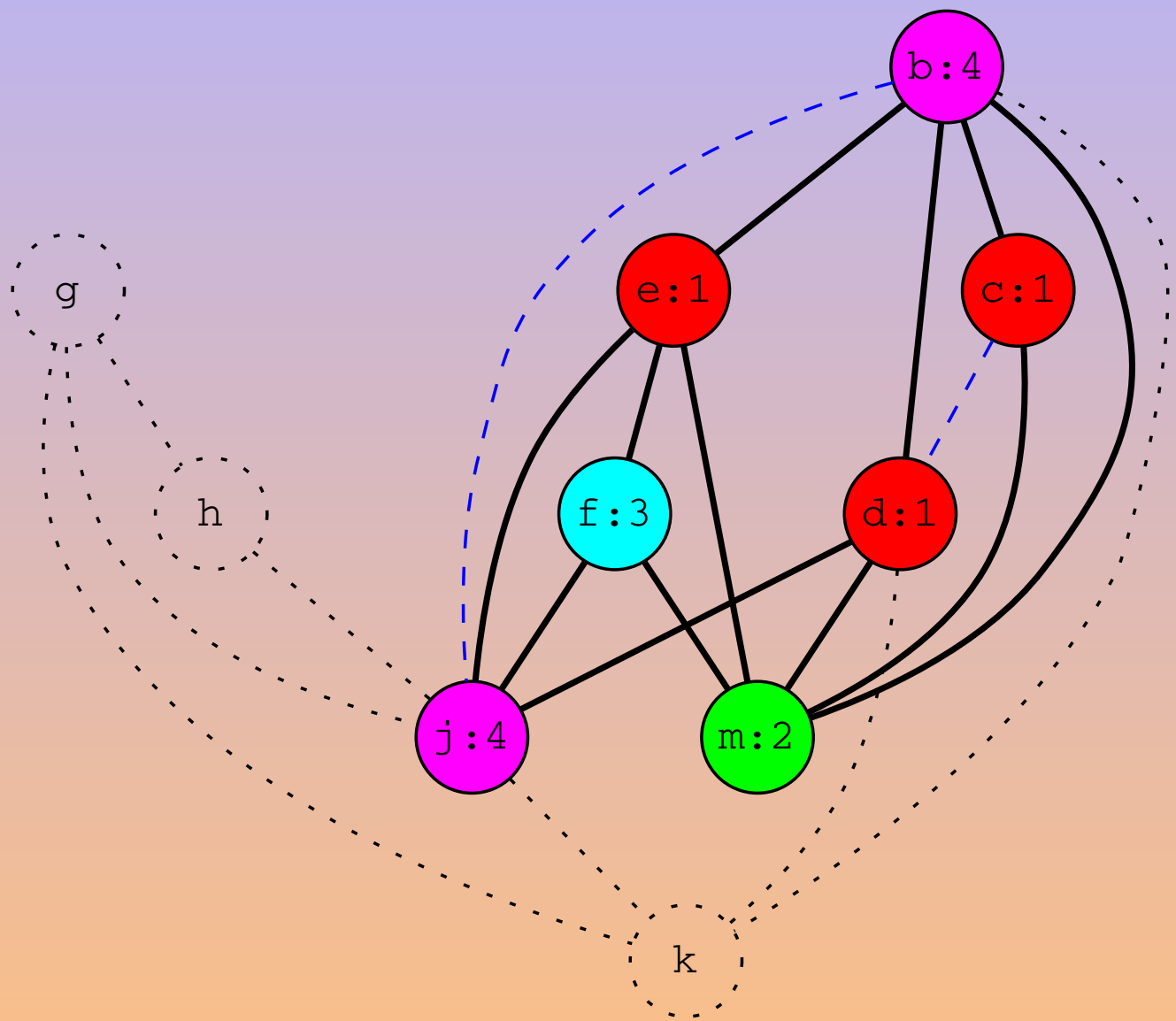
Interference Graph: Simplify 4

j&*b*
c&*d*
k
h
g

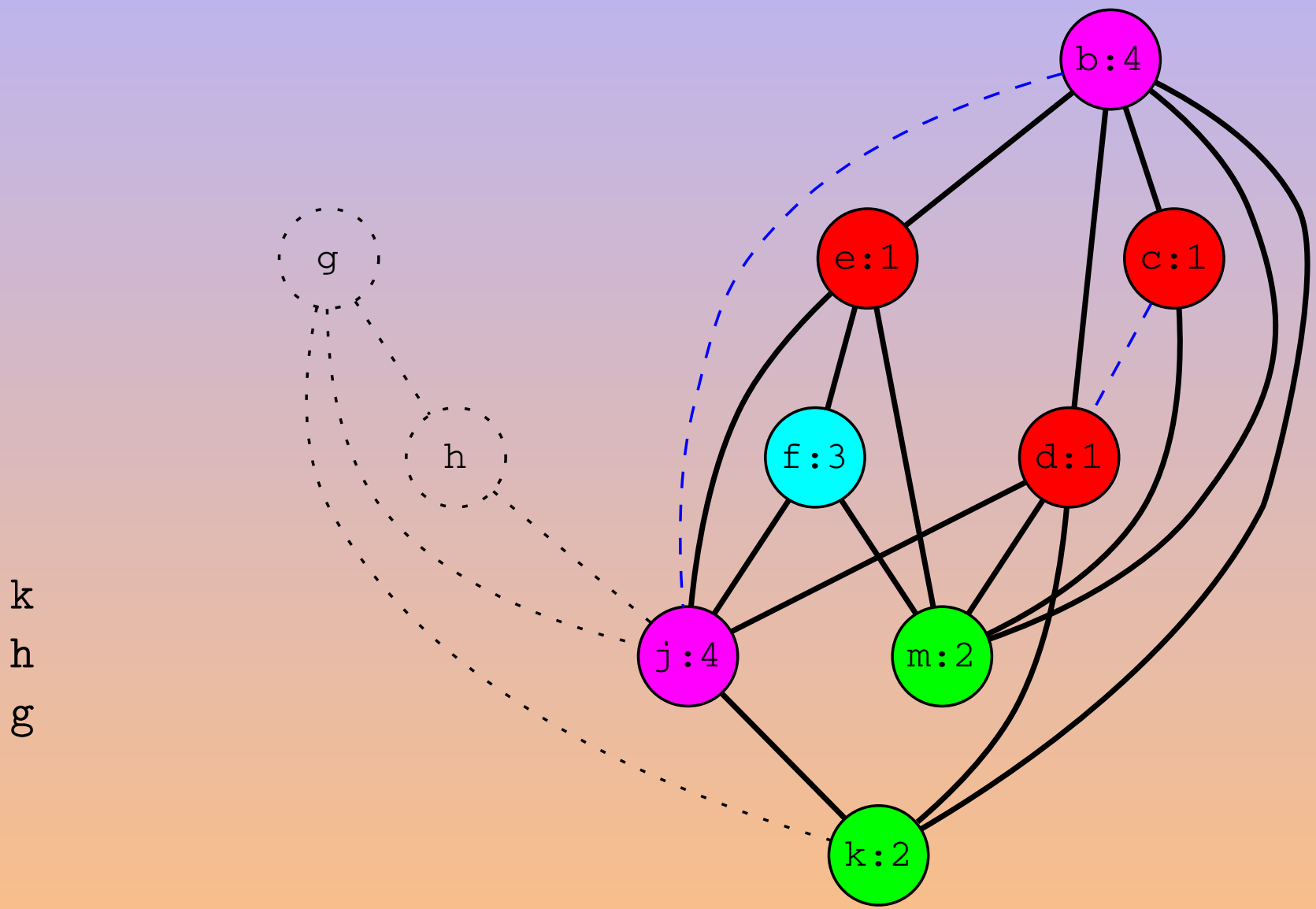


Interference Graph: Simplify 3

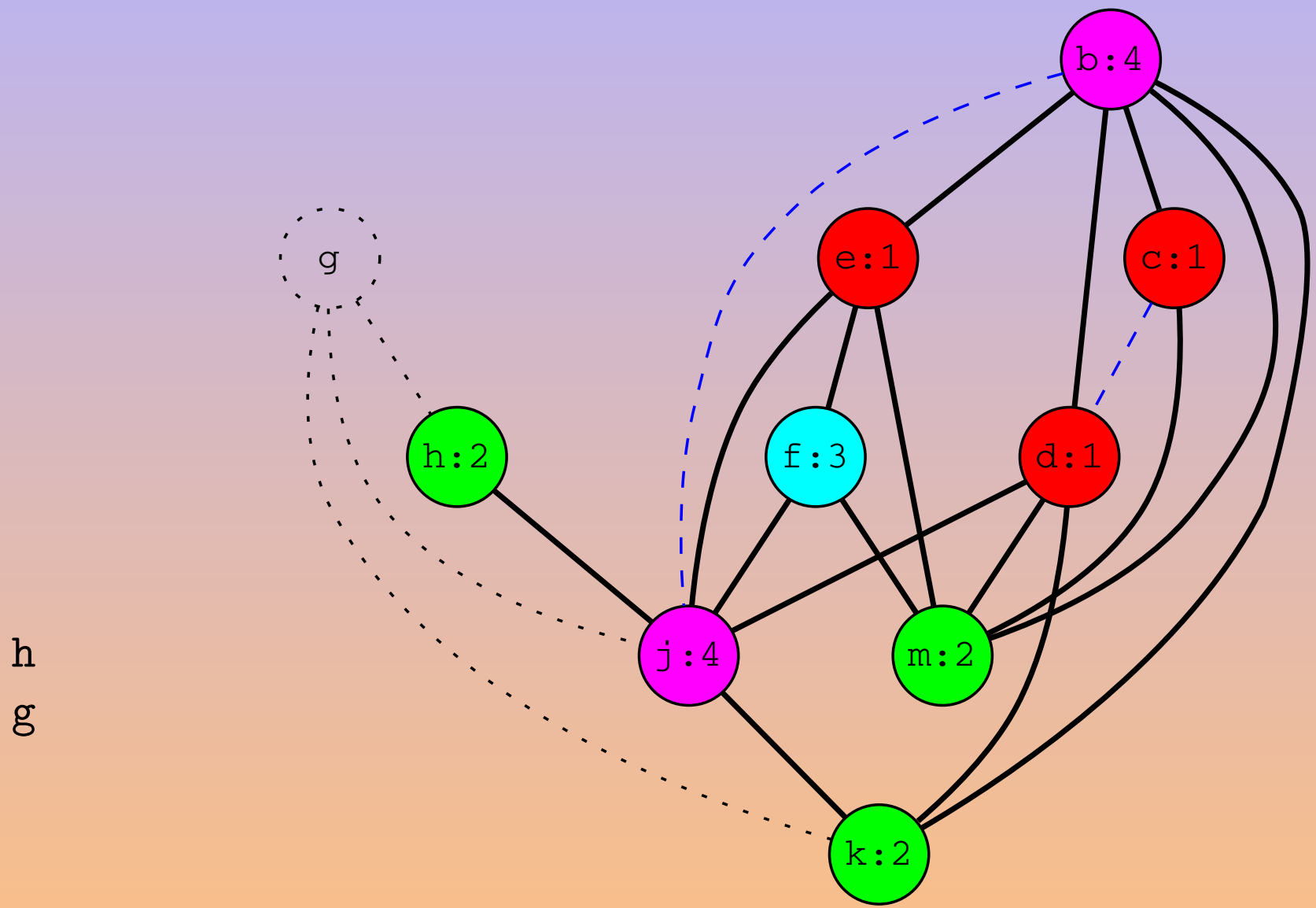
c&*d*
k
h
g



Interference Graph: Simplify 2

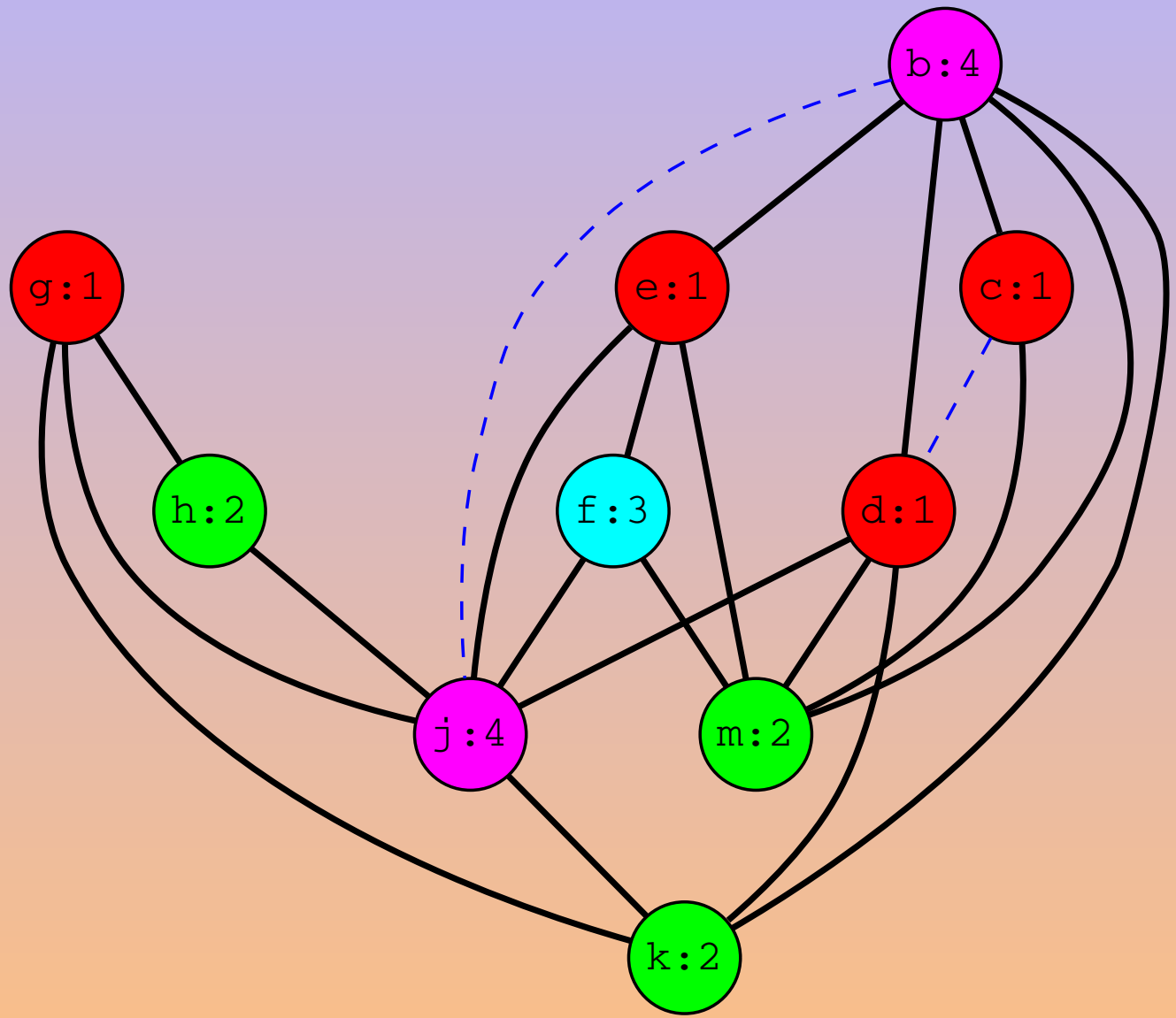


Interference Graph: Simplify 1



h
g

Interference Graph: Simplify 0



g

Interference Graph: Result

```
live in: k j
  g := [j + 12]
  h := k - 1
  f := g * h
  e := [j + 8]
  m := [j + 16]
  b := [f]
  c := e + 8
  d := c
  k := m + 4
  j := b
live out: d k j
```

```
live in: r2 r4
  r1 := [r4 + 12]
  r2 := r2 - 1
  r3 := r1 * r2
  r1 := [r4 + 8]
  r2 := [r4 + 16]
  r4 := [r3]
  r1 := r1 + 8
# r1 := r1
  r2 := r2 + 4
# r4 := r4
live out: r1 r2 r4
```

Precolored Nodes

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Callee & Caller Save Registers

- It just rocks!
  - Caller Save Def'd by `calls`.
  - Callee Save Def'd at entry, used at exit of functions.
- Register pressure will push temporaries live across calls into callee save.

Callee & Caller Save Registers

- It just rocks!
  - Caller Save Def'd by `calls`.
  - Callee Save Def'd at entry, used at exit of functions.
- Register pressure will push temporaries live across calls into callee save.

Callee & Caller Save Registers

- It just rocks!

  Caller Save Def'd by `calls`.

  Callee Save Def'd at entry, used at exit of functions.

- Register pressure will push temporaries live across calls into callee save.

Callee & Caller Save Registers

- It just rocks!

  Caller Save Def'd by `calls`.

  Callee Save Def'd at entry, used at exit of functions.

- Register pressure will push temporaries live across calls into callee save.

Conflicts

Minimize the conflicts ("pressure") with hard regs. Source and sink.

```
# Routine: fact
l0:                     # def $s0, $s1...
  move $x11, $s0        # def: $x11 use: $s0
  move $x12, $s1        # def: $x12 use: $s1
  ...
l6:
  move $s0, $x11        # def: $s0 use: $x11
  move $s1, $x12        # def: $s1 use: $x12
  ...
                        # use: $fp, $ra, $sp,
                        # ...  $v0, $zero
```

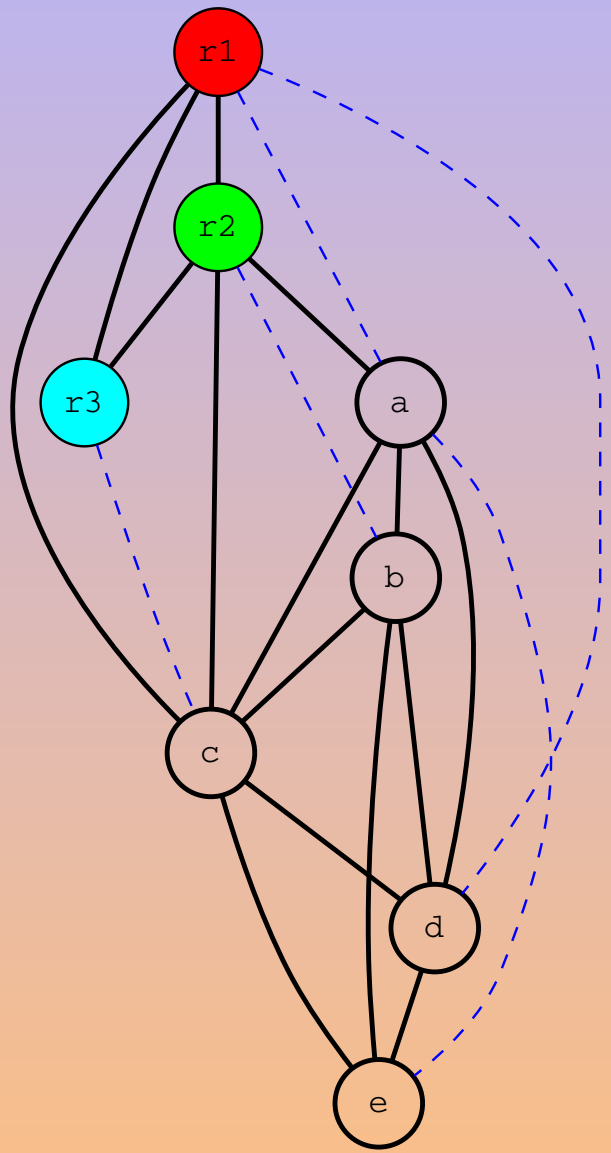
Example [Appel, 1998]

```
int
f (int a, int b)
{
  int d = 0;
  int e = a;
  do
  {
    d += b;
    --e;
  } while (e > 0);
  return d;
}
```

```
enter:
  c := r3
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  r3 := c
  return
# liveout: r1, r3
```

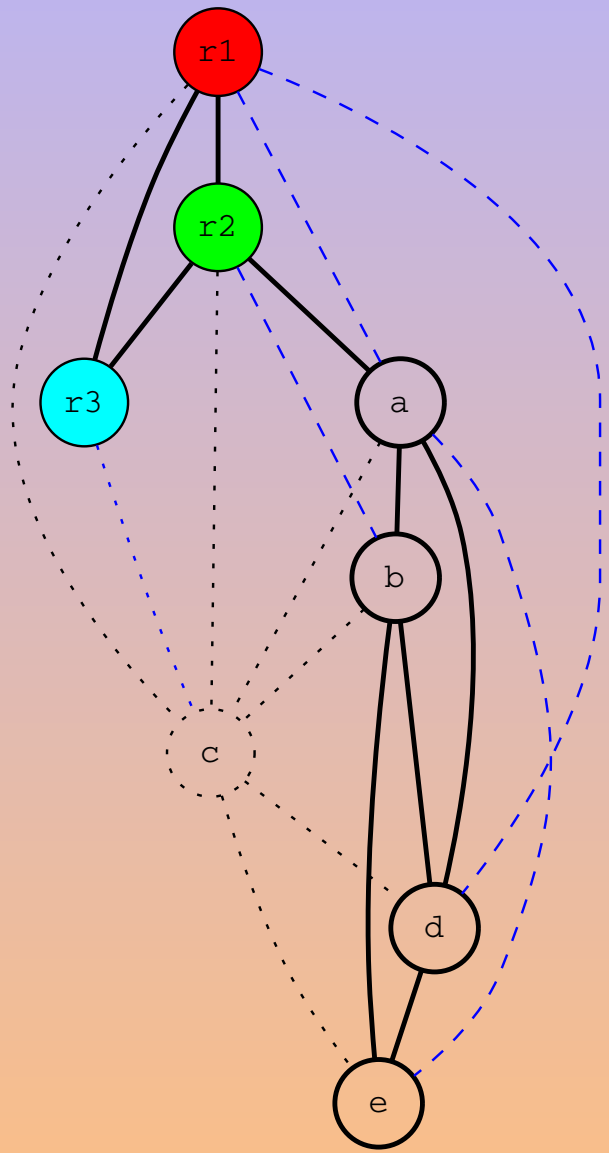
Example

```
enter:
  c := r3
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  r3 := c
  return
# liveout: r1, r3
```



Interference Graph: Simplify 0

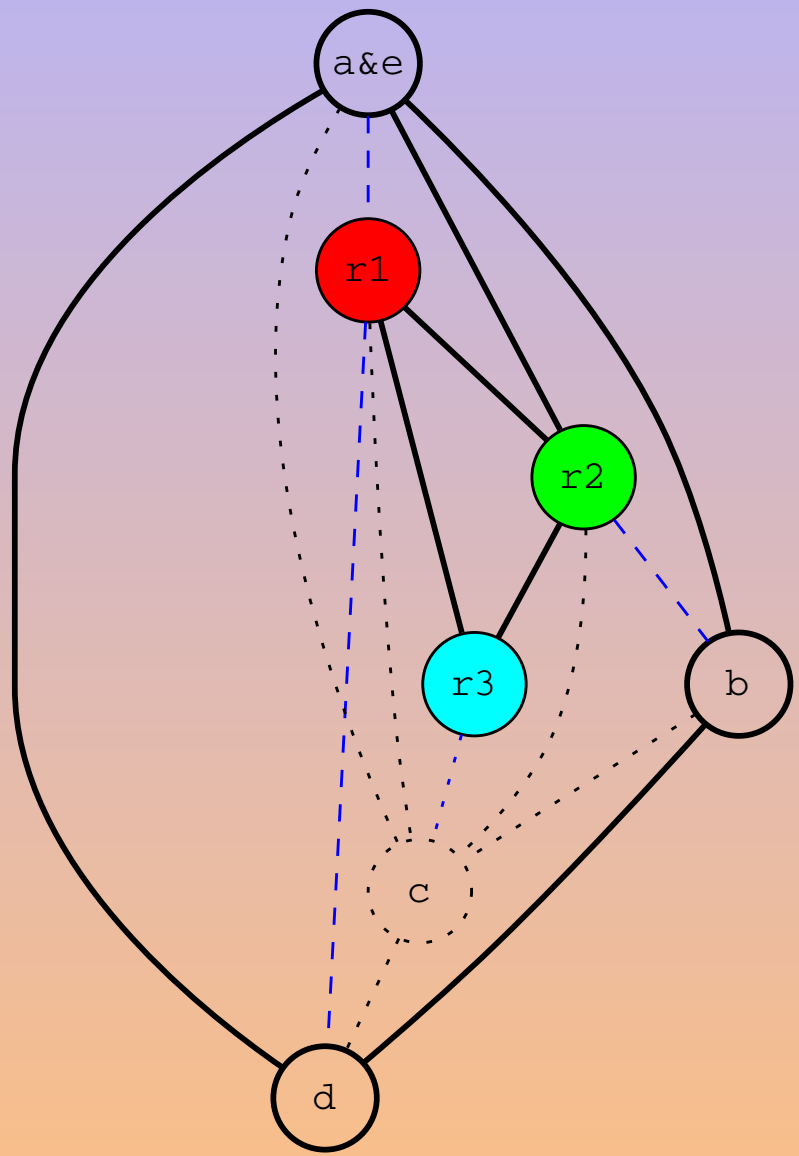
c



Interference Graph: Simplify 1

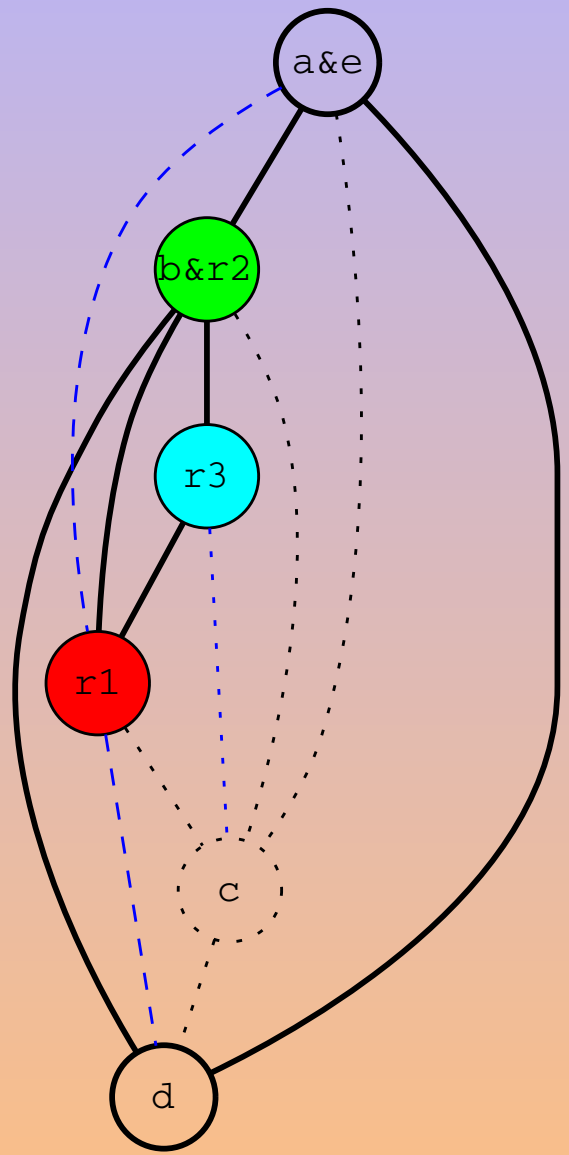
a&*e*

c



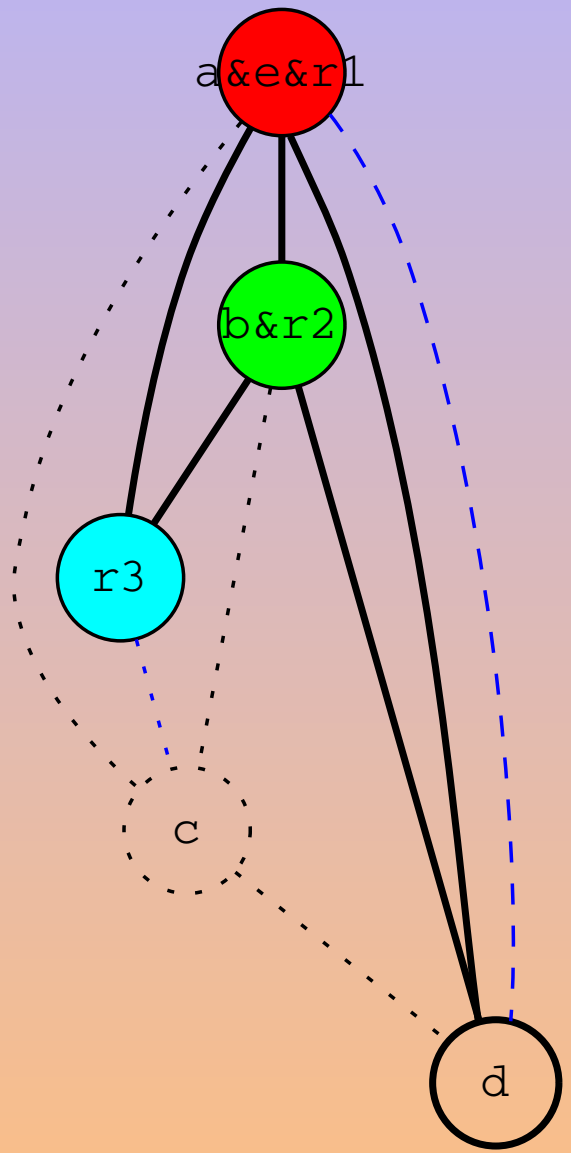
Interference Graph: Simplify 2

b&r2
a&e
c



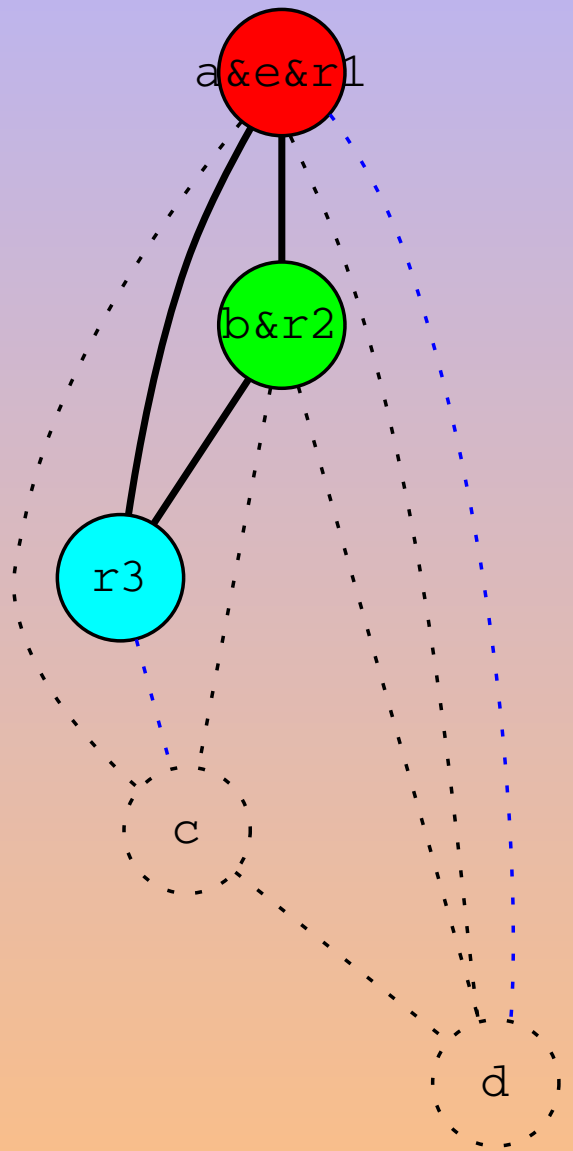
Interference Graph: Simplify 3

a&*e*&*r*1
b&*r*2
a&*e*
c



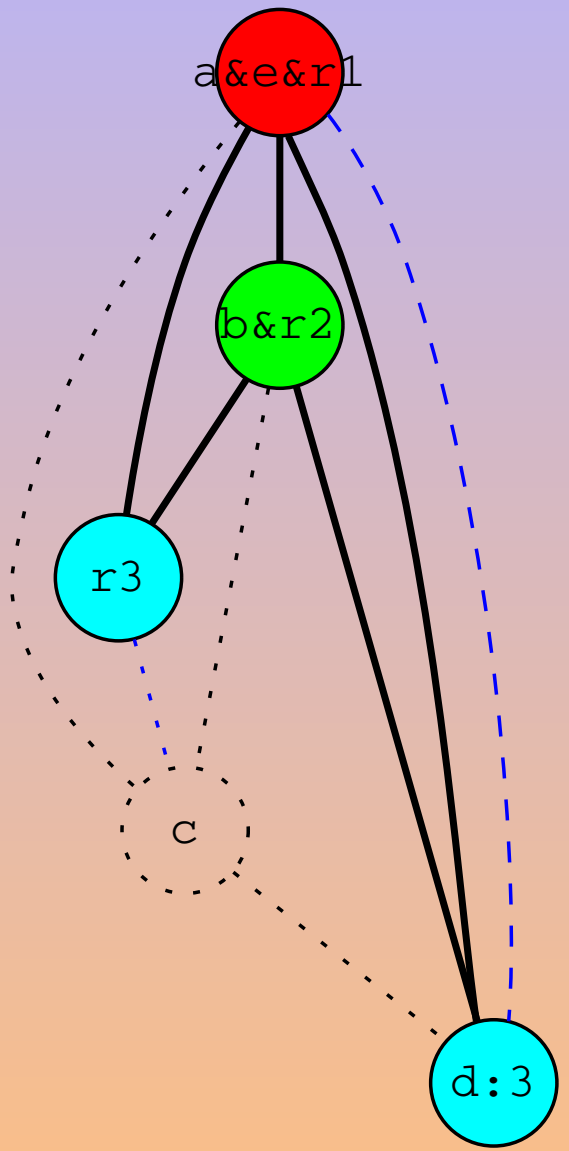
Interference Graph: Simplify 4

d
a&*e*&*r*1
b&*r*2
a&*e*
c



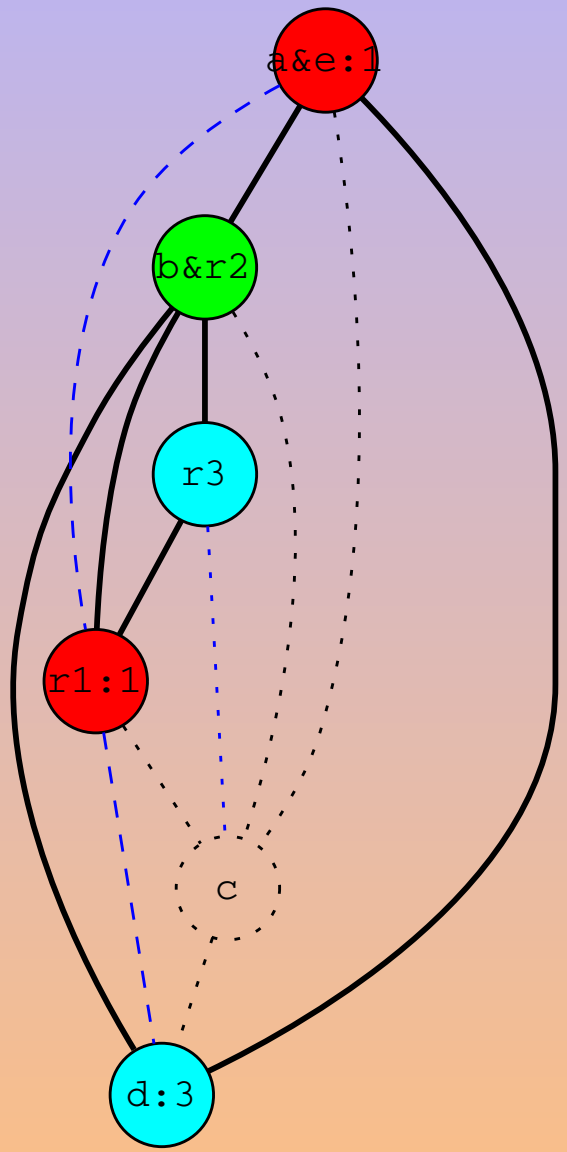
Interference Graph: Simplify 4

d
a&*e*&*r*1
b&*r*2
a&*e*
c



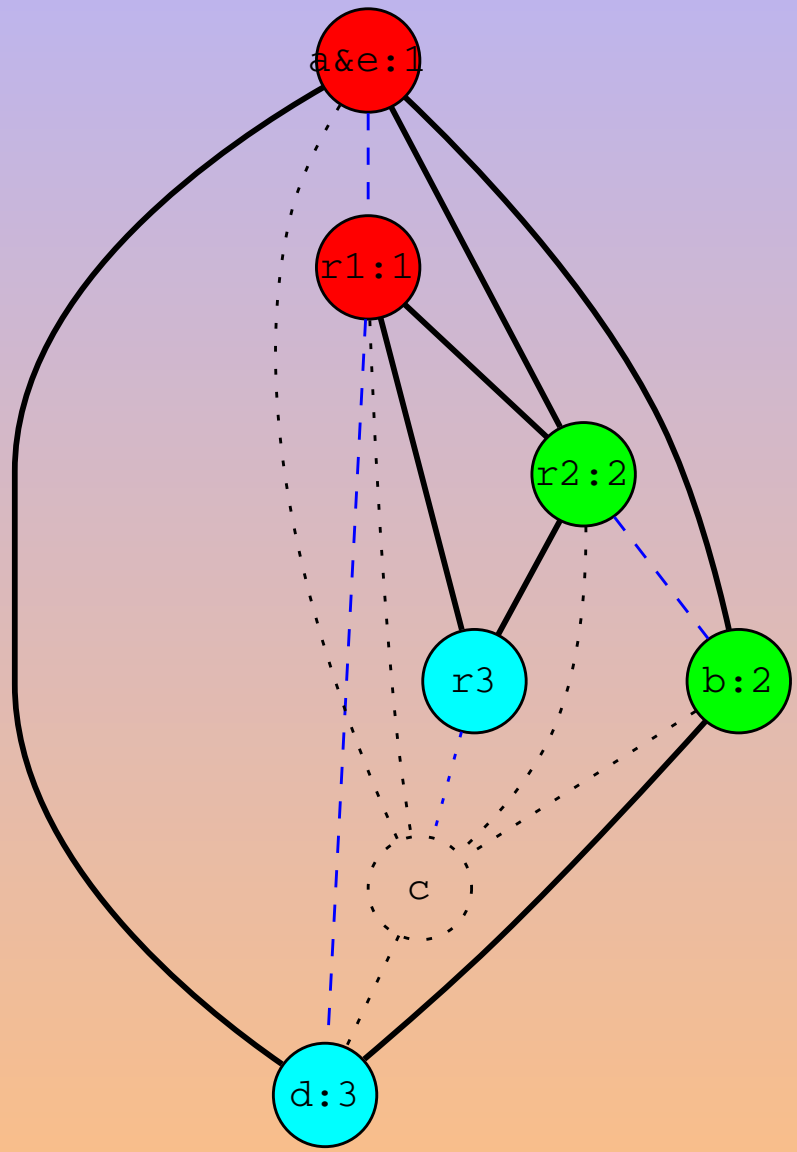
Interference Graph: Simplify 3

$a\&e\&r1$
$b\&r2$
$a\&e$
c



Interference Graph: Simplify 2

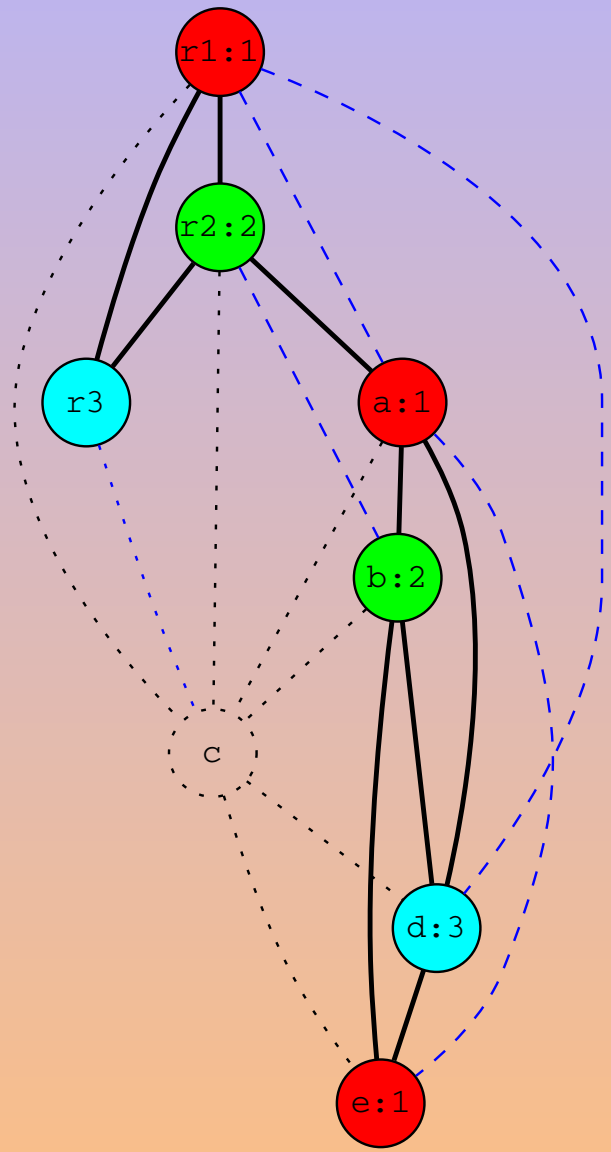
b&*r*2
a&*e*
c



Interference Graph: Simplify 1

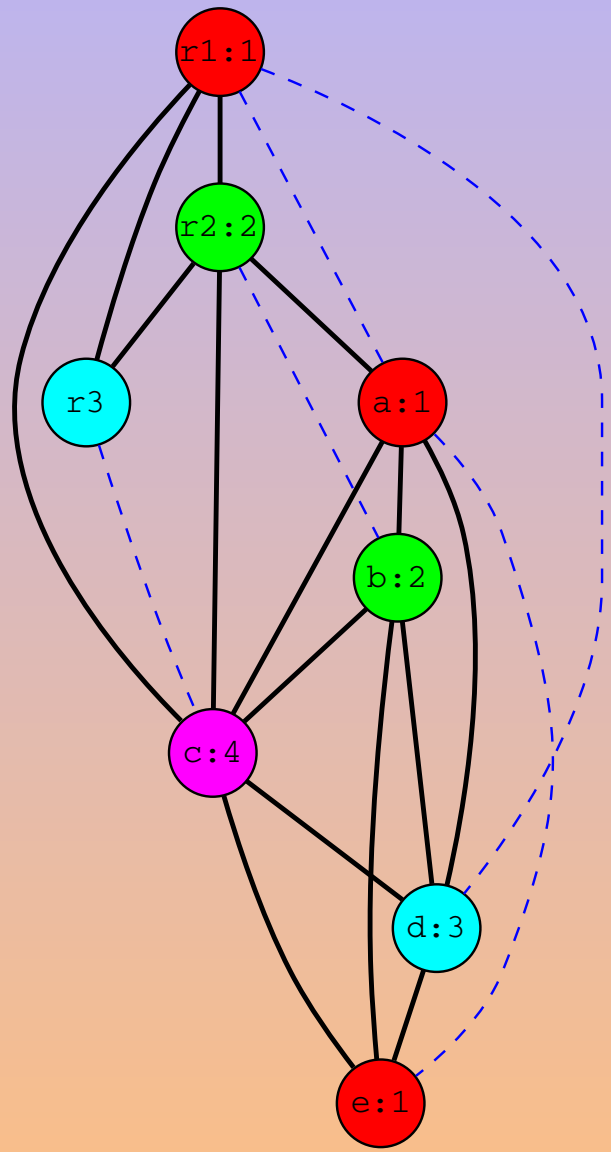
$a \& e$

c



Interference Graph: Simplify 0

c



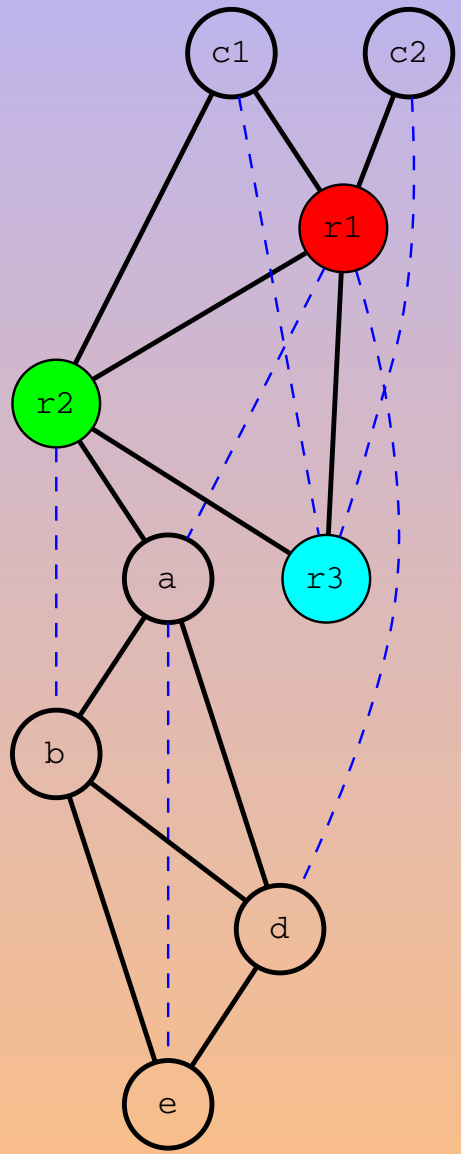
Spilling

```
enter:
  c := r3
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  r3 := c
  return
# liveout: r1, r3
```

```
enter:
  c1   := r3
  [sp+8] := c1
  a    := r1
  b    := r2
  d    := 0
  e    := a
loop:
  d    := d + b
  e    := e - 1
  if e > 0
     goto loop
  r1     := d
  c2     := [sp+8]
  r3     := c2
  return
# liveout: r1, r3
```

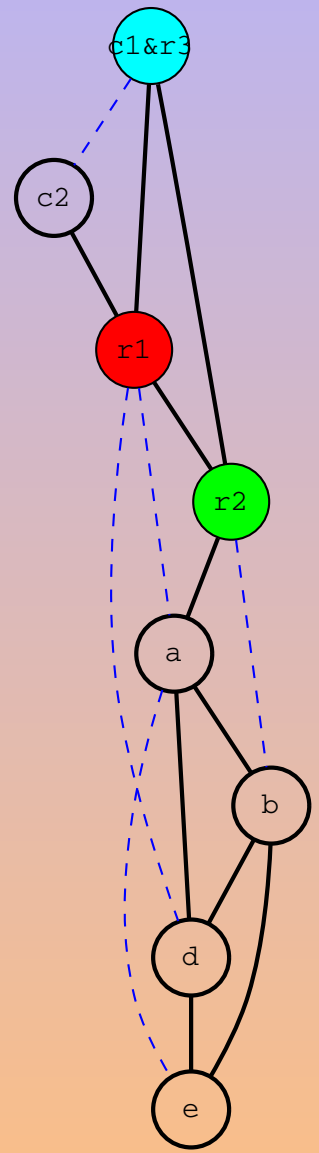
Example

```
enter:
  c1   := r3
  [sp+8] := c1
  a    := r1
  b    := r2
  d    := 0
  e    := a
loop:
  d    := d + b
  e    := e - 1
  if e > 0
     goto loop
  r1   := d
  c2   := [sp+8]
  r3   := c2
  return
# liveout: r1, r3
```



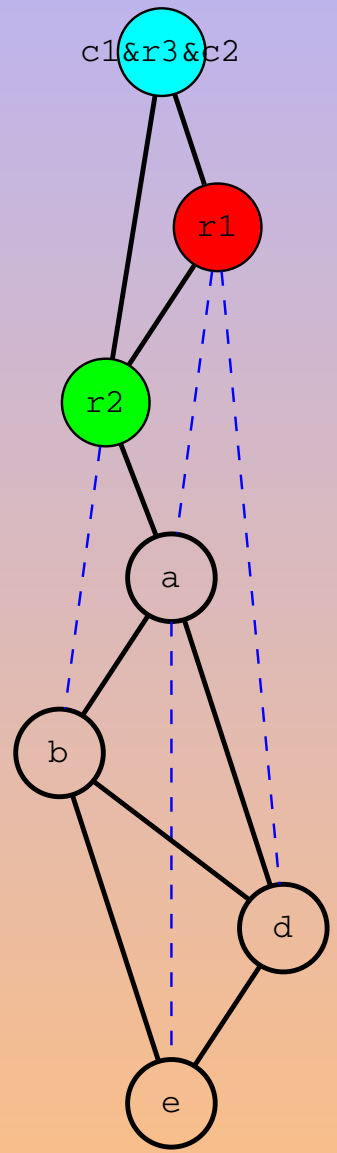
Interference Graph: Simplify 0

$c1\&r3$



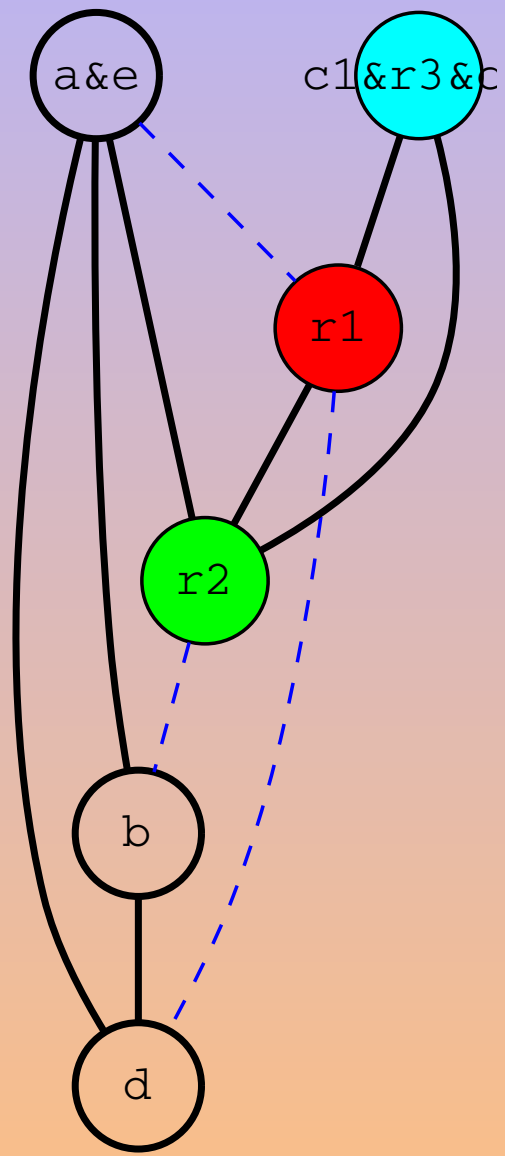
Interference Graph: Simplify 1

$c1\&r3\&c2$
$c1\&r3$



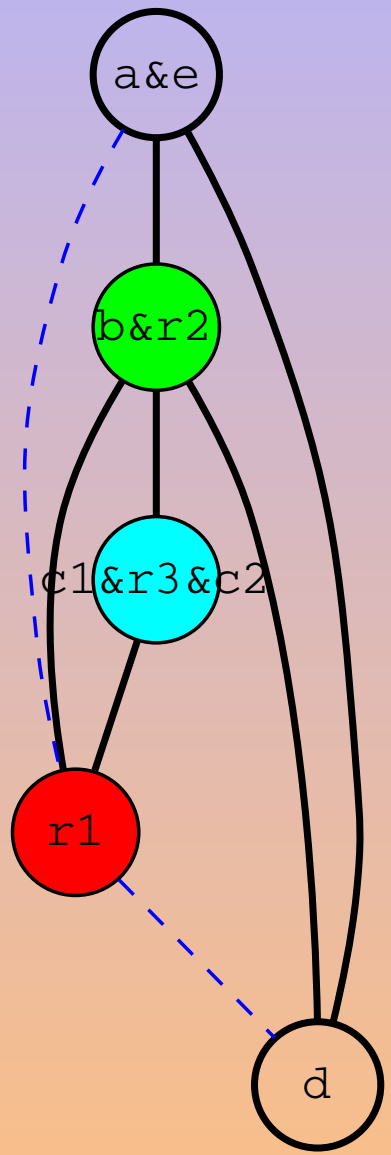
Interference Graph: Simplify 2

$a\&e$
$c1\&r3\&c2$
$c1\&r3$



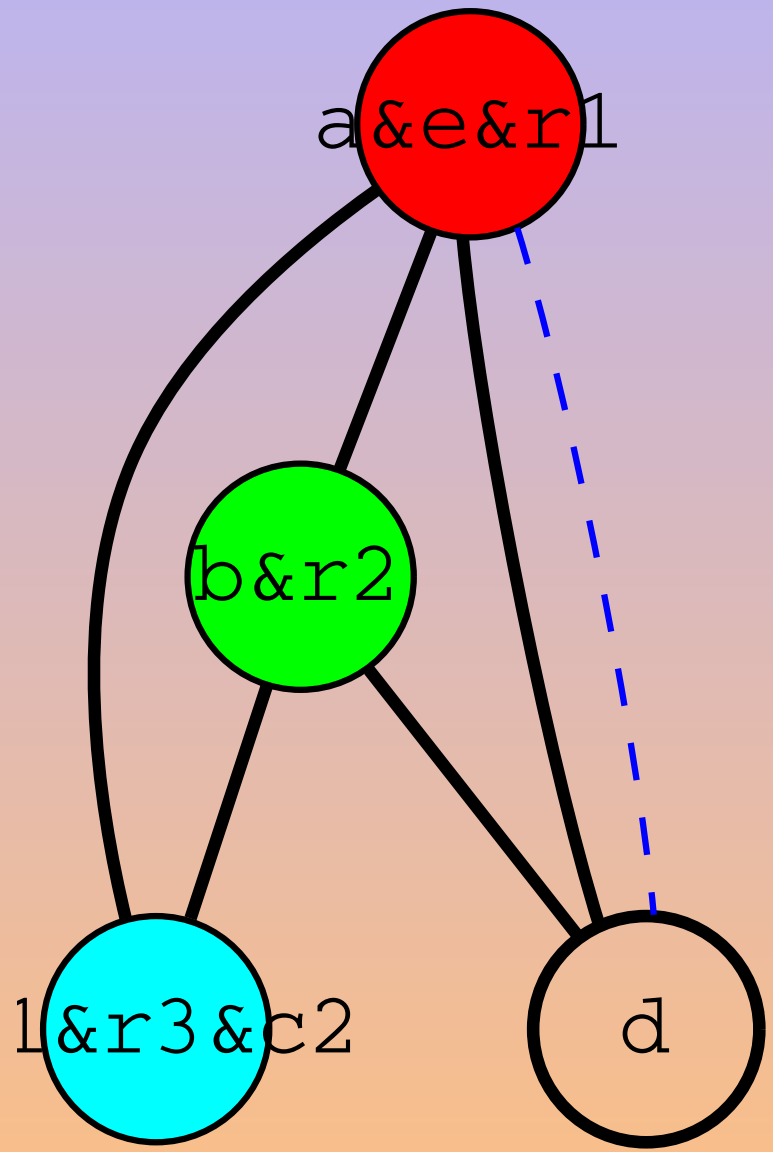
Interference Graph: Simplify 3

b&*r*2
a&*e*
*c*1&*r*3&*c*2
*c*1&*r*3



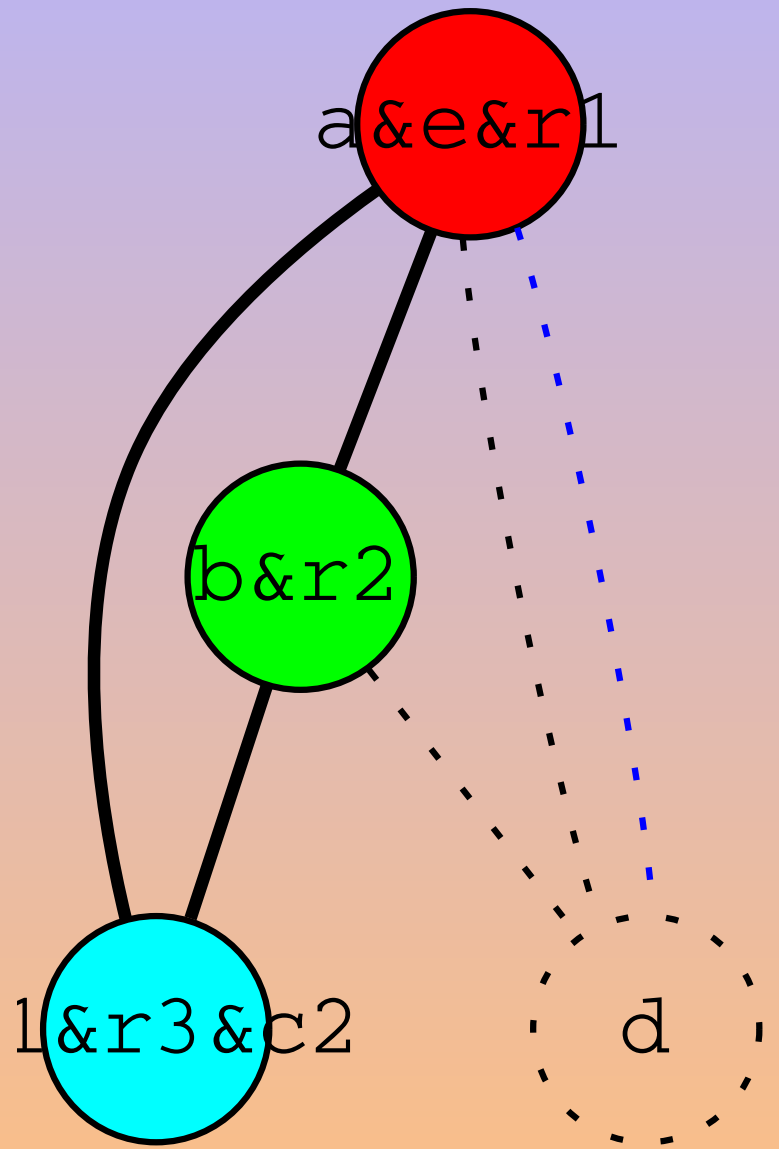
Interference Graph: Simplify 4

a&*e*&*r*1
b&*r*2
a&*e*
*c*1&*r*3&*c*2
*c*1&*r*3



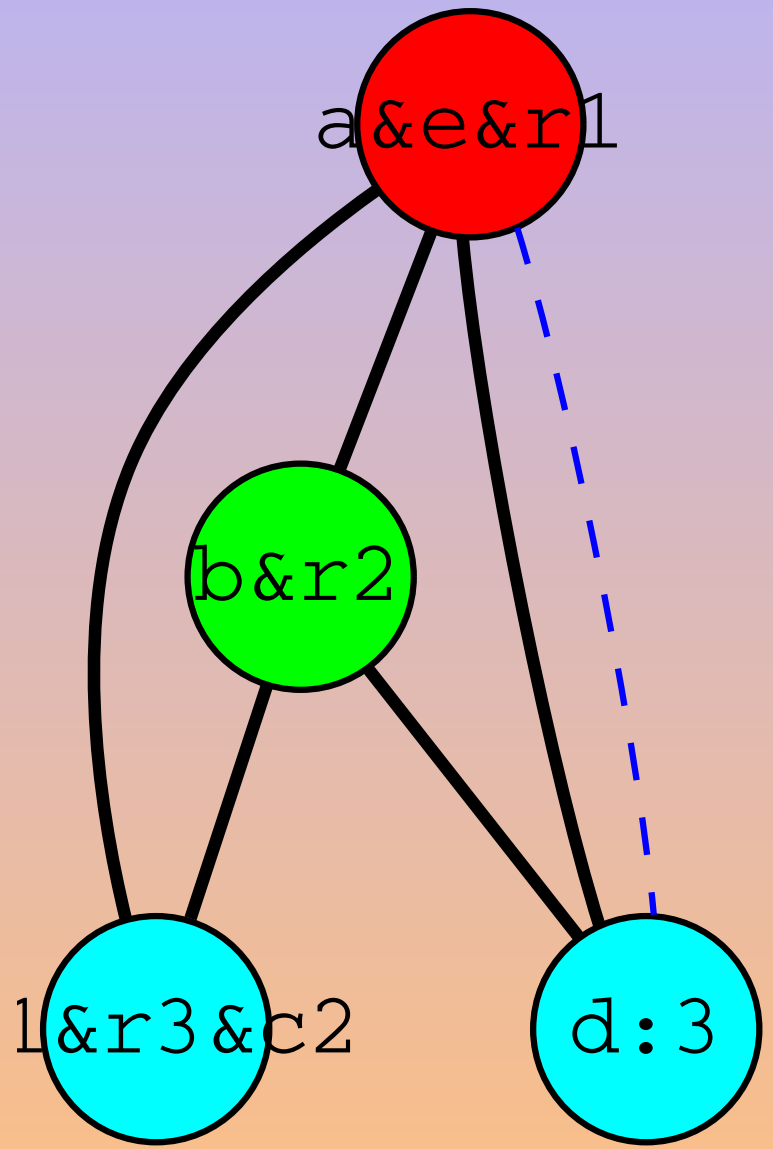
Interference Graph: Simplify 5

d
$a\&e\&r1$
$b\&r2$
$a\&e$
$c1\&r3\&c2$
$c1\&r3$



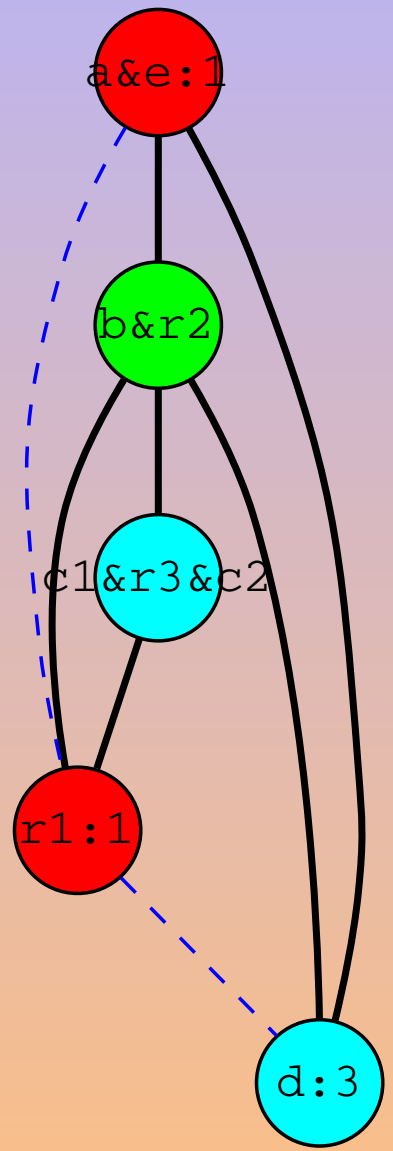
Interference Graph: Simplify 5

d
$a\&e\&r1$
$b\&r2$
$a\&e$
$c1\&r3\&c2$
$c1\&r3$



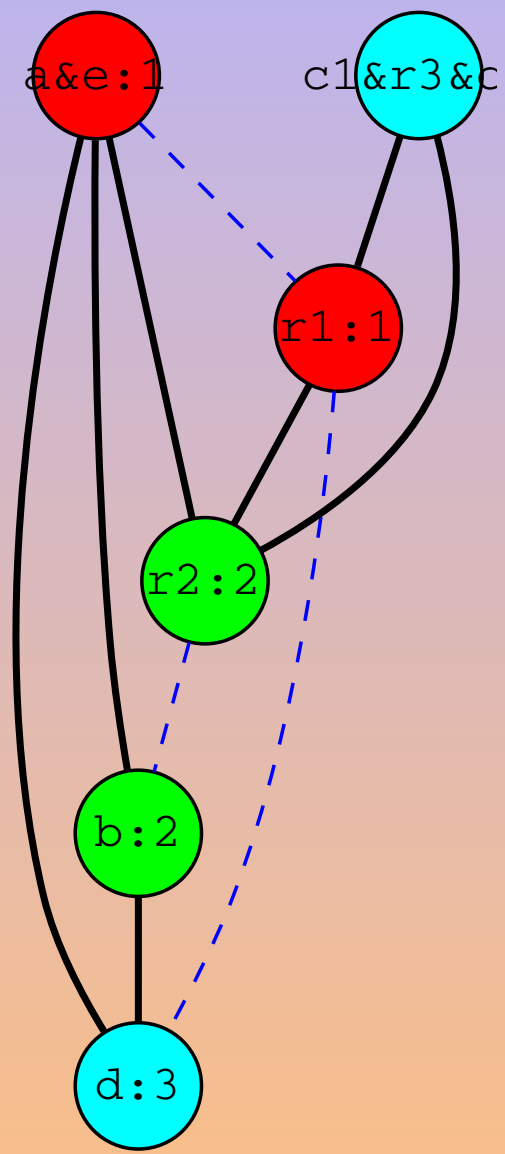
Interference Graph: Simplify 4

$a\&e\&r1$
$b\&r2$
$a\&e$
$c1\&r3\&c2$
$c1\&r3$



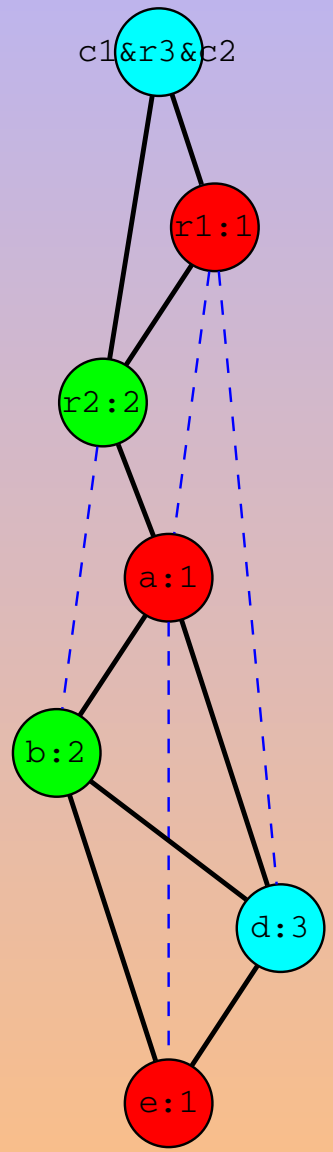
Interference Graph: Simplify 3

b&*r*2
a&*e*
*c*1&*r*3&*c*2
*c*1&*r*3



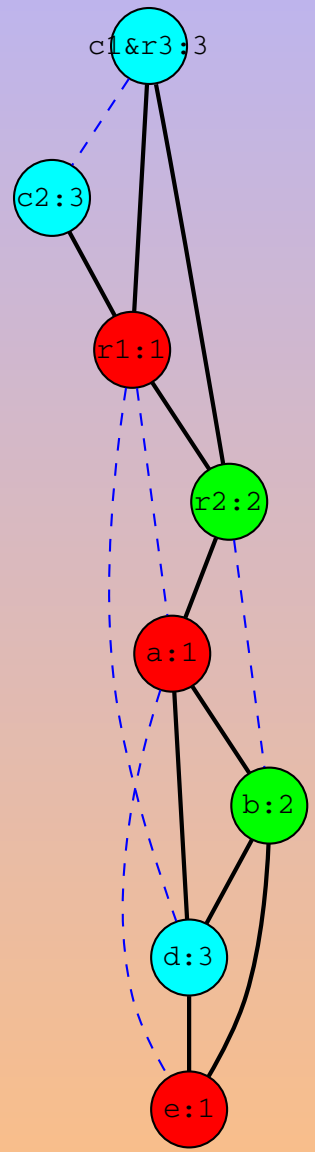
Interference Graph: Simplify 2

$a\&e$
$c1\&r3\&c2$
$c1\&r3$



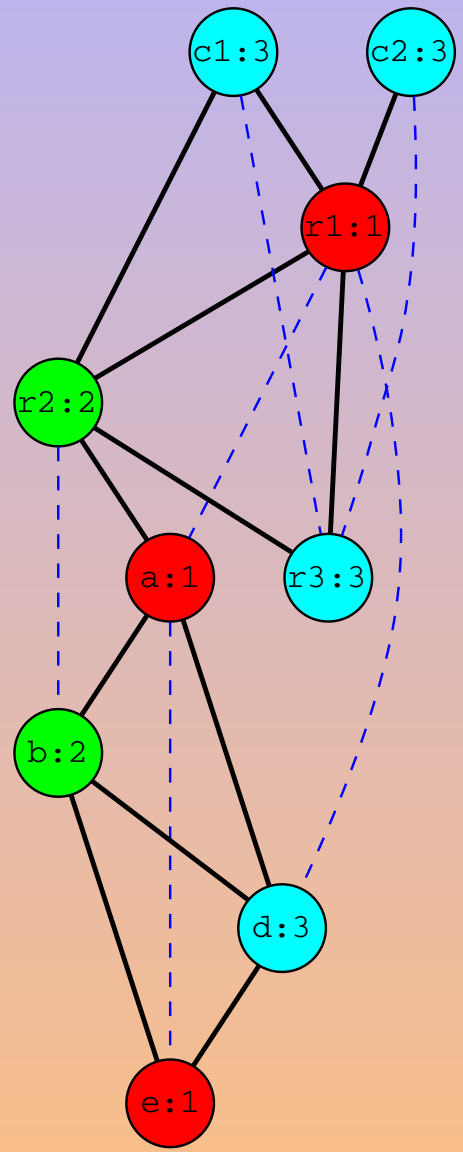
Interference Graph: Simplify 1

$c1\&r3\&c2$
$c1\&r3$



Interference Graph: Simplify 0

$c1\&r3$



Result

```
enter:
  c1   := r3
  [sp+8] := c1
  a    := r1
  b    := r2
  d    := 0
  e    := a
loop:
  d    := d + b
  e    := e - 1
  if e > 0
     goto loop
  r1   := d
  c2   := [sp+8]
  r3   := c2
  return
# liveout: r1, r3
```

```
enter:
  r3   := r3
  [sp+8] := r3
  r1   := r1
  r2   := r2
  r3   := 0
  r1   := r1
loop:
  r3   := r3 + r2
  r1   := r1 - 1
  if r1 > 0
     goto loop
  r1   := r3
  r3   := [sp+8]
  r3   := r3
  return
# liveout: r1, r3
```

```
enter:

  [sp+8] := r3


  r3   := 0

loop:
  r3   := r3 + r2
  r1   := r1 - 1
  if r1 > 0
     goto loop
  r1   := r3
  r3   := [sp+8]

  return
# liveout: r1, r3
```

Implementation

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.
- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix
- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.
- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.
- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.

- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.

  Use both!
- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.

  Use both!
- For more information, see [Appel, 1998].

Alternatives to Graph Coloring

Register Allocation for Trees

Can be done during instruction selection with maximal munch

```
function SimpleAlloc (t)
  for each nontrivial tile u child of t
    SimpleAlloc (u)
  for each nontrivial tile u child of t
    n := n - 1
  n := n + 1
  assign rn to (the root of) t
```

[Appel, 1998]

Bibliography I


Appel, A. W. (1998).
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press.


Briggs, P. (1992).
Register Allocation via Graph Coloring.
PhD thesis, Rice University, Houston, Texas.


Matz, M. (2003).
Design and Implementation of a Graph Coloring Register Allocator for gcc.
pages 151–169.

Instruction scheduling

Akim Demaille, Etienne Renault, Roland Levillain

May 19, 2018

Table of contents

Dependencies analysis 1/2

Two instructions are independent they can be permuted without altering the consistency

Dependencies analysis 1/2

Two instructions are independent they can be permuted without altering the consistency

- The 3 following instructions are independent

$$\begin{aligned} \text{inst}_1 : &\quad a \leftarrow 42 \\ \text{inst}_2 : &\quad b \leftarrow 51 \\ \text{inst}_3 : &\quad c \leftarrow 0 \end{aligned}$$

Dependencies analysis 1/2

Two instructions are independent they can be permuted without altering the consistency

- The 3 following instructions are independent

$$\begin{array}{ll} \text{inst}_1 : & a \leftarrow 42 \\ \text{inst}_2 : & b \leftarrow 51 \\ \text{inst}_3 : & c \leftarrow 0 \end{array}$$

- inst_1, inst_2 and $inst_3$ can then be reordered

$$\begin{array}{ll} \text{inst}_1 : & a \leftarrow 42 \\ \text{inst}_2 : & b \leftarrow 51 \\ \text{inst}_3 : & c \leftarrow 0 \end{array} \parallel \begin{array}{ll} \text{inst}_1 : & a \leftarrow 42 \\ \text{inst}_3 : & c \leftarrow 0 \\ \text{inst}_2 : & b \leftarrow 51 \end{array} \parallel \begin{array}{ll} \text{inst}_3 : & c \leftarrow 0 \\ \text{inst}_1 : & a \leftarrow 42 \\ \text{inst}_2 : & b \leftarrow 51 \end{array}$$

$$\begin{array}{ll} \text{inst}_1 : & c \leftarrow 0 \\ \text{inst}_2 : & b \leftarrow 51 \\ \text{inst}_3 : & a \leftarrow 42 \end{array} \parallel \begin{array}{ll} \text{inst}_1 : & b \leftarrow 51 \\ \text{inst}_3 : & c \leftarrow 0 \\ \text{inst}_2 : & a \leftarrow 42 \end{array} \parallel \begin{array}{ll} \text{inst}_3 : & b \leftarrow 51 \\ \text{inst}_1 : & a \leftarrow 42 \\ \text{inst}_2 : & c \leftarrow 0 \end{array}$$

Dependencies analysis 2/2

Two instructions are dependent if the first one needs to be executed before the second one.

Dependencies analysis 2/2

Two instructions are dependent if the first one needs to be executed before the second one.

- The 3 following instructions are dependent, i.e. no reordering is possible!

$$\begin{aligned} \text{inst}_1 : &\quad a \leftarrow 42 \\ \text{inst}_2 : &\quad b \leftarrow a + 51 \\ \text{inst}_3 : &\quad c \leftarrow b \times 12 \end{aligned}$$

Dependencies analysis 2/2

Two instructions are dependent if the first one needs to be executed before the second one.

- The 3 following instructions are dependent, i.e. no reordering is possible!

$$\begin{aligned} \text{inst}_1 : &\quad a \leftarrow 42 \\ \text{inst}_2 : &\quad b \leftarrow a + 51 \\ \text{inst}_3 : &\quad c \leftarrow b \times 12 \end{aligned}$$

- Two kind of dependencies:
  - **Data dependencies**: the instruction manipulates a "variable" computed by another instruction.
  - **Instruction dependencies**: the instruction is a "cjump", the next instruction depends of the "cjump".

Read after Write (RAW)

An instruction reads from a location after an earlier instruction has written to it.

Read after Write (RAW)

An instruction reads from a location after an earlier instruction has written to it.

inst_1 : `lw $2, 0($4)`
inst_2 : `addi $6, $2, 42`

Read after Write (RAW)

An instruction reads from a location after an earlier instruction has written to it.

inst_1 : `lw $2, 0($4)`
inst_2 : `addi $6, $2, 42`

inst_1 and inst_2 cannot be permuted, otherwise inst_2 would read an old value for $2

Write after Read (WAR)

An instruction writes to a location after an earlier instruction has read from it.

Write after Read (WAR)

An instruction writes to a location after an earlier instruction has read from it.

inst_1 : `lw   $2, 0($4)`
inst_2 : `addi $4, $12, 42`

Write after Read (WAR)

An instruction writes to a location after an earlier instruction has read from it.

inst_1 : `lw $2, 0($4)`
inst_2 : `addi $4, $12, 42`

inst_1 and inst_2 cannot be permuted, otherwise inst_1 would read a new value for $4

Write after Write (WAW)

An instruction writes to a location after an earlier instruction has written to it.

Write after Write (WAW)

An instruction writes to a location after an earlier instruction has written to it.

$inst_1$: `add $1, $2, $3`
$inst_2$: `add $1, $5, $6`

Write after Write (WAW)

An instruction writes to a location after an earlier instruction has written to it.

$inst_1$: `add $1, $2, $3`
$inst_2$: `add $1, $5, $6`

$inst_1$ and $inst_2$ cannot be permuted, otherwise $inst_1$ would write an old value in \$1

Why and When reordering?

We would like to reorder the instructions within each basic block in a way which:

- preserves the dependencies between those instructions (and hence the correctness of the program)
- achieves the minimum possible number of pipeline stalls, i.e. two instructions simultaneously in the pipeline manipulates same data, registers, etc.

The two problems can be addressed separately (whew!).

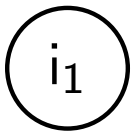
Preserving and computing dependencies?

We construct a directed acyclic graph (DAG) to represent the dependencies between instructions:

- For each instruction in the basic block, create a corresponding vertex in the graph
- For each dependency between two instructions, create a corresponding (annotated) edge in the graph. Note that this edge is annotated.

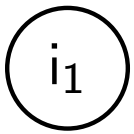
Computing the dependency graph

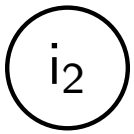
i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



Computing the dependency graph

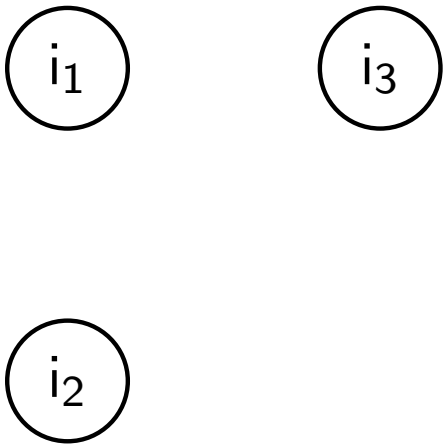
i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	





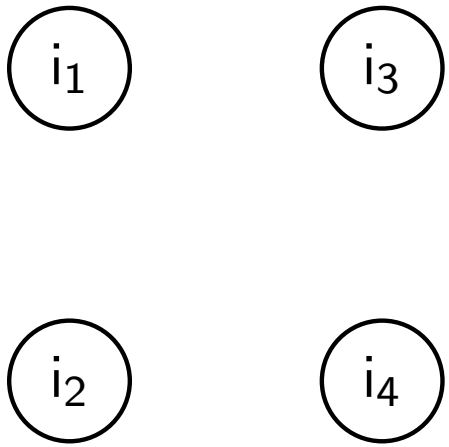
Computing the dependency graph

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



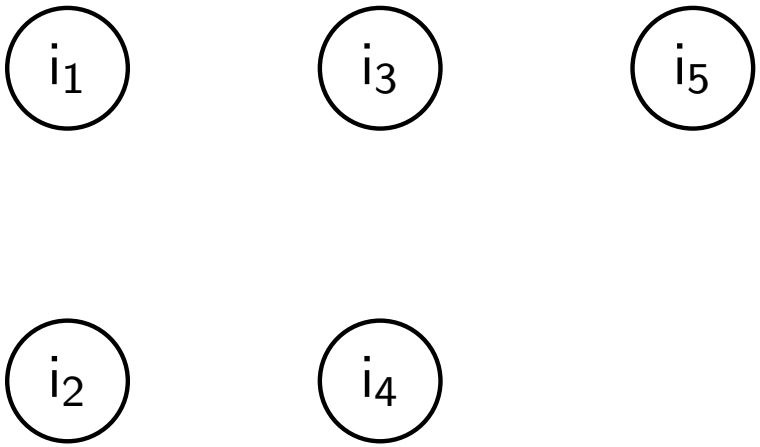
Computing the dependency graph

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



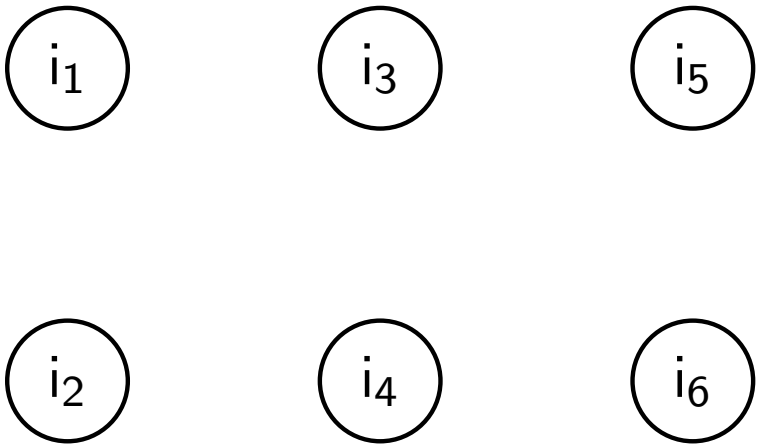
Computing the dependency graph

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



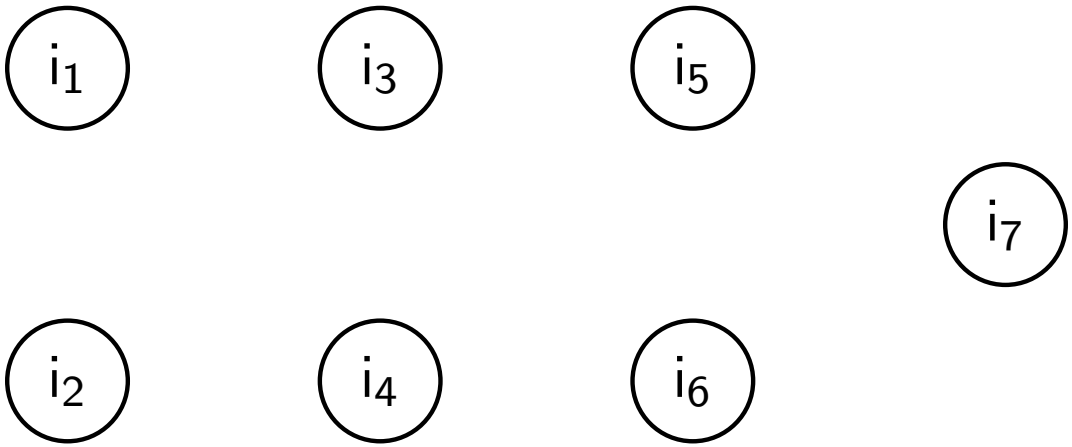
Computing the dependency graph

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



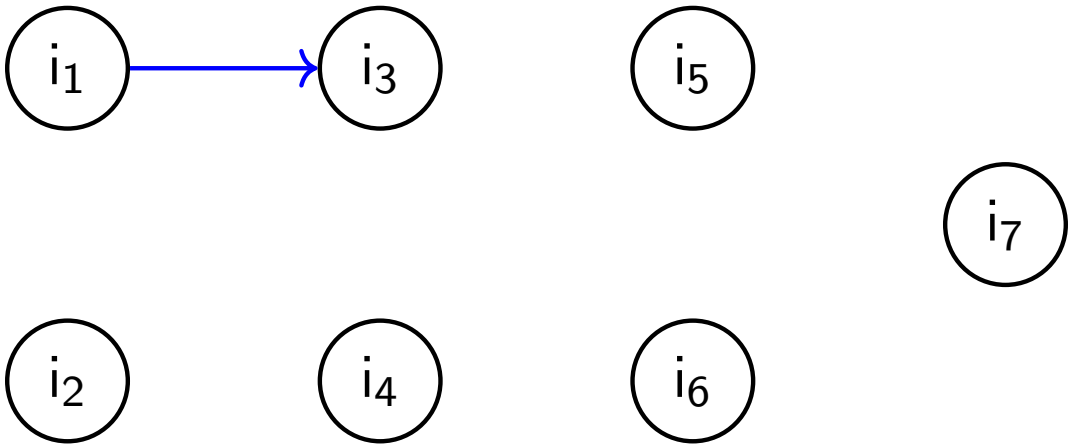
Computing the dependency graph

```
i1 :  lw  $1,0($10)  |  i4 :  sw  $3,12($10)  |  i7 :  sw  $3,16($10)
i2 :  lw  $2,4($10)  |  i5 :  lw  $4,8($10)   |
i3 :  add $3,$1,$2   |  i6 :  add $3,$1,$4    |
```



Computing the dependency graph

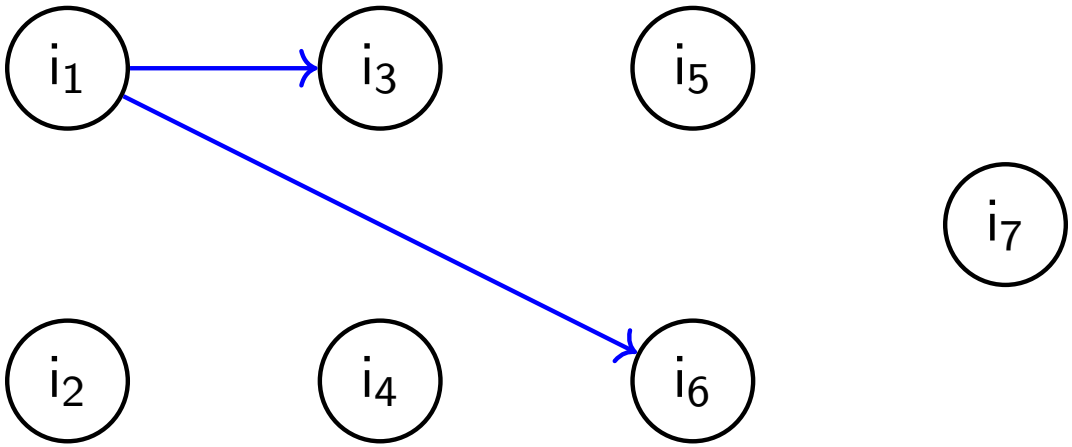
```
i1 :  lw   $1,0($10)  |  i4 :  sw   $3,12($10)  |  i7 :  sw   $3,16($10)
i2 :  lw   $2,4($10)  |  i5 :  lw   $4,8($10)   |
i3 :  add  $3,$1,$2   |  i6 :  add  $3,$1,$4    |
```



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

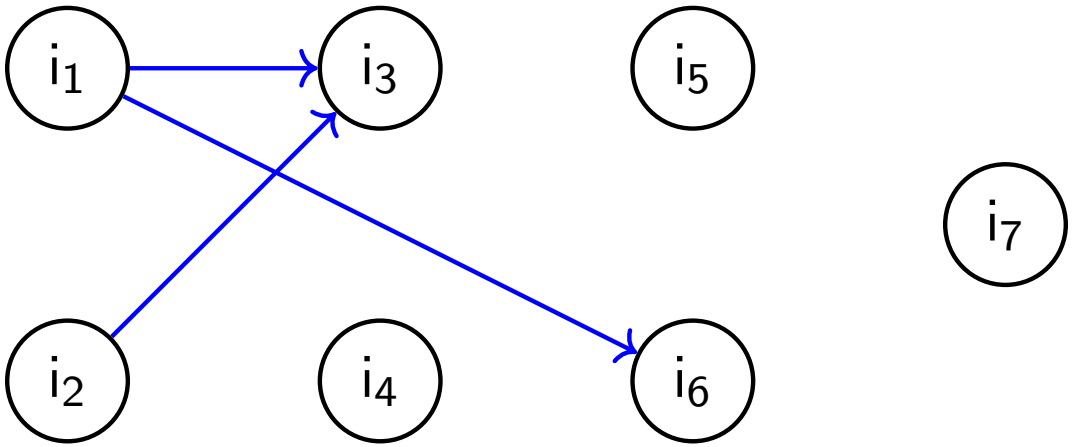
i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

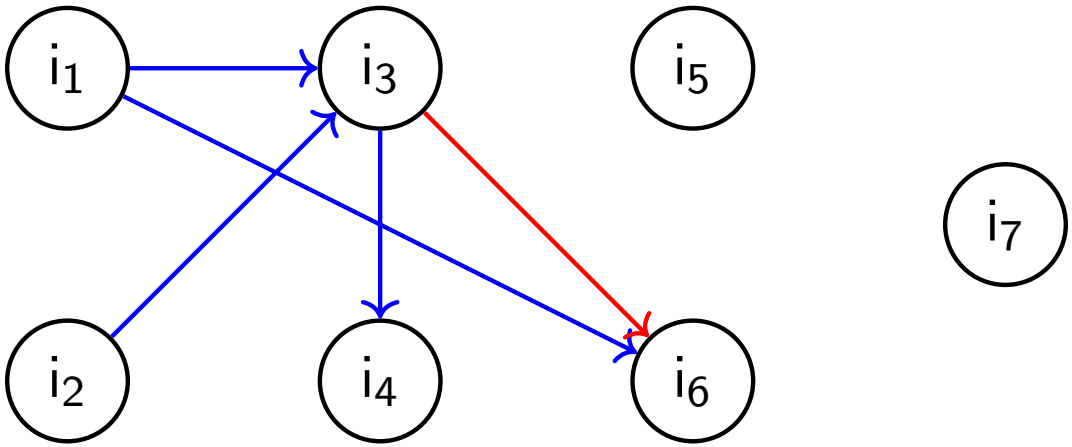
i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

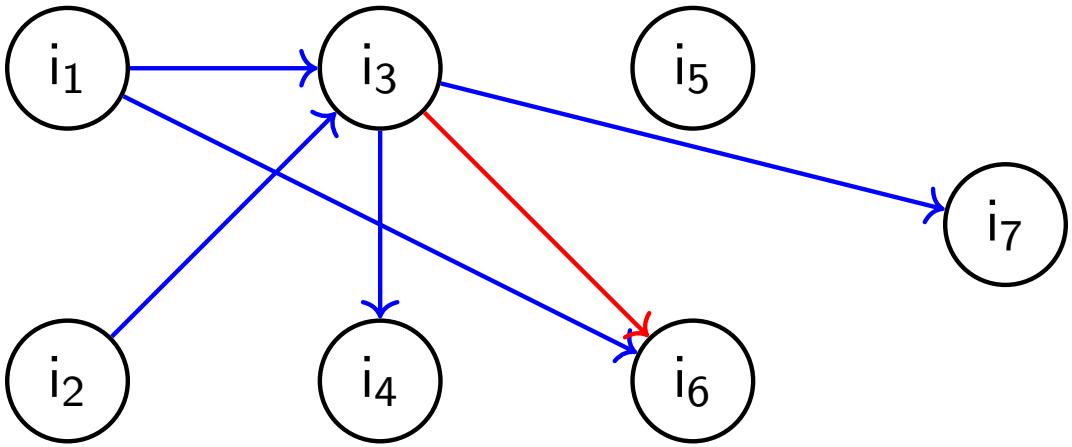
```
i1 :  lw   $1,0($10)  |  i4 :  sw   $3,12($10)  |  i7 :  sw   $3,16($10)
i2 :  lw   $2,4($10)  |  i5 :  lw   $4,8($10)   |
i3 :  add  $3,$1,$2   |  i6 :  add  $3,$1,$4    |
```



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

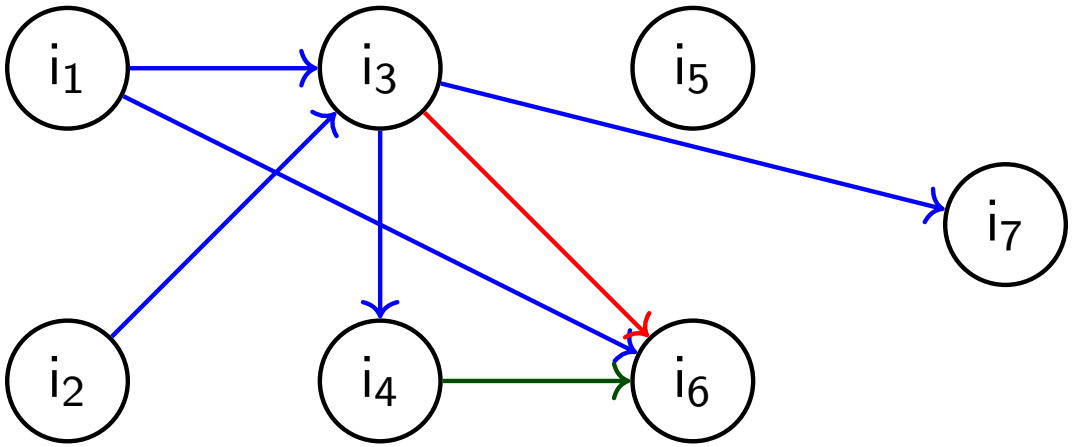
```
i1 :  lw   $1,0($10)  |  i4 :  sw   $3,12($10)  |  i7 :  sw   $3,16($10)
i2 :  lw   $2,4($10)  |  i5 :  lw   $4,8($10)   |
i3 :  add  $3,$1,$2   |  i6 :  add  $3,$1,$4    |
```



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

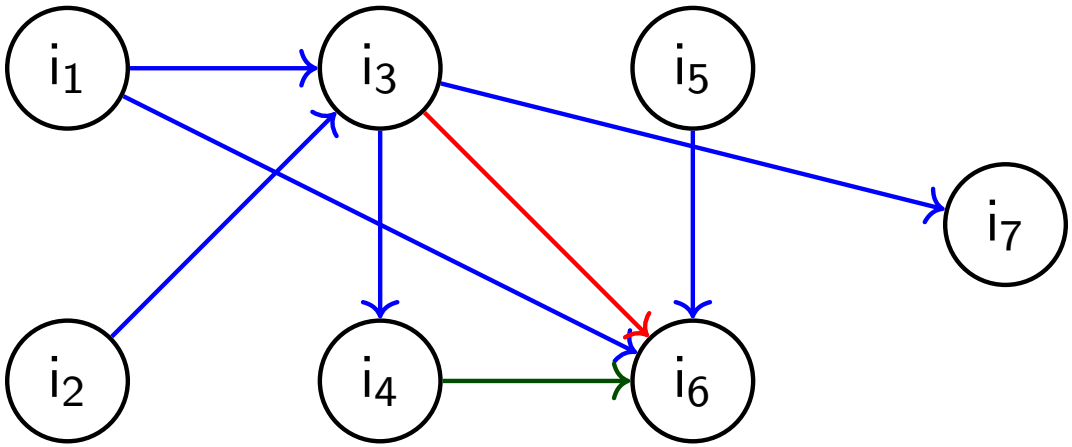
```
i1 :  lw   $1,0($10)  |  i4 :  sw   $3,12($10)  |  i7 :  sw   $3,16($10)
i2 :  lw   $2,4($10)  |  i5 :  lw   $4,8($10)   |
i3 :  add  $3,$1,$2   |  i6 :  add  $3,$1,$4    |
```



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

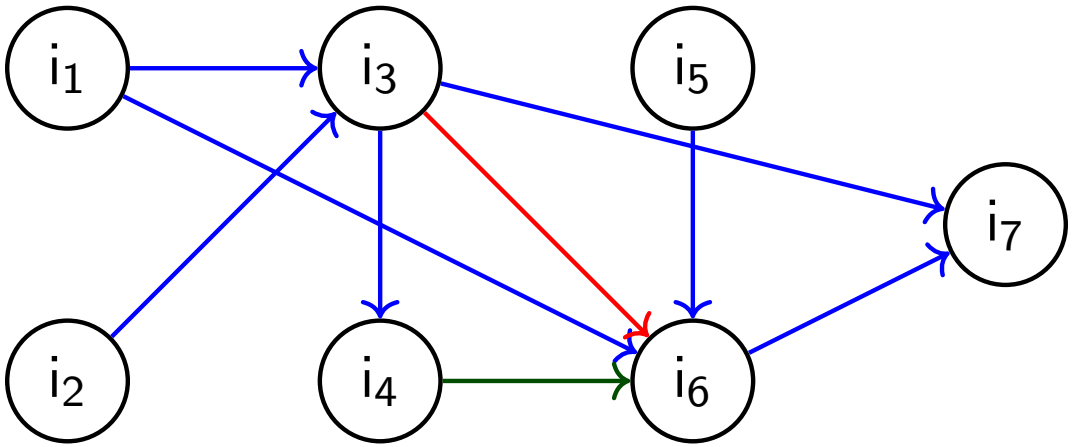
```
i1 :  lw   $1,0($10)  |  i4 :  sw   $3,12($10)  |  i7 :  sw   $3,16($10)
i2 :  lw   $2,4($10)  |  i5 :  lw   $4,8($10)   |
i3 :  add  $3,$1,$2   |  i6 :  add  $3,$1,$4    |
```



Type of dependency: RAW, WAW, WAR

Computing the dependency graph

```
i1 :  lw   $1,0($10)  |  i4 :  sw   $3,12($10)  |  i7 :  sw   $3,16($10)
i2 :  lw   $2,4($10)  |  i5 :  lw   $4,8($10)   |
i3 :  add  $3,$1,$2   |  i6 :  add  $3,$1,$4    |
```



Type of dependency: RAW, WAW, WAR

Preserving dependencies: Critical Path 1/2

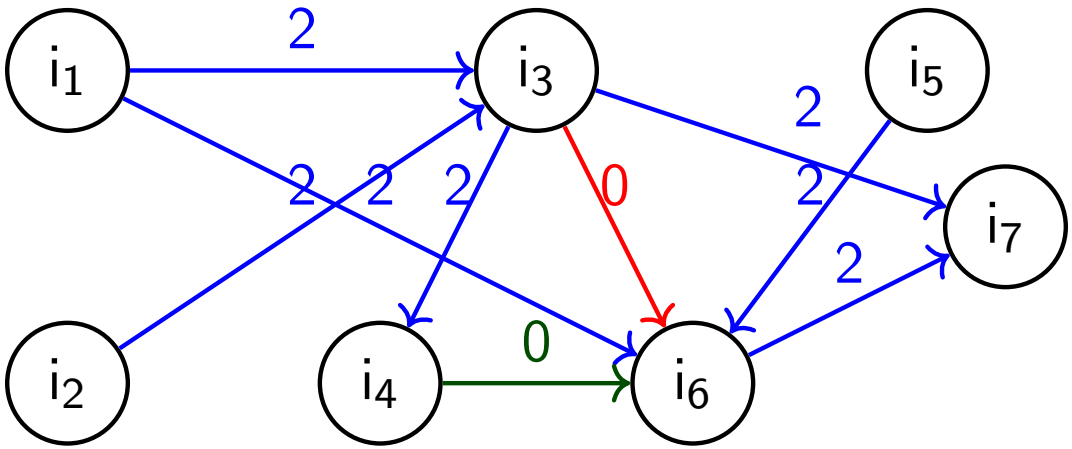
The critical path represents the longest path between two nodes. We add **delays** (weights) to edges:

- 0 for WAW and WAR dependencies
- 2 for RAW dependencies with memory access
- 1 for other RAW dependencies

Preserving dependencies: Critical Path 1/2

The critical path represents the longest path between two nodes. We add **delays** (weights) to edges:

- 0 for WAW and WAR dependencies
- 2 for RAW dependencies with memory access
- 1 for other RAW dependencies



Preserving dependencies: Critical Path 2/2

Any (reverse) topological sort of this DAG (i.e. any linear ordering of the vertices which keeps all the edges "pointing forwards") will maintain the dependencies and hence preserve the correctness of the program.

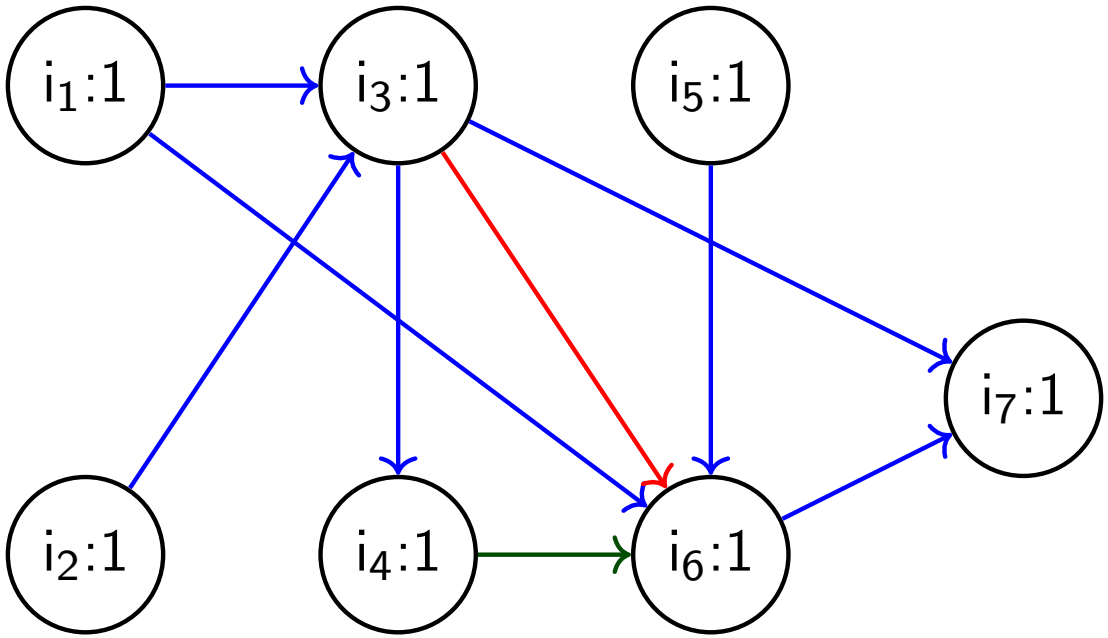
Algorithm:

- Associate a weight 1 to all "instruction node"
- For all nodes n_i in topological postorder
  - If n_i is not a leaf
    - ★ For all nodes n_j in succ(n_i) do
      n_i.weight $\leftarrow$ max (n_i.weight, n_j.weight+ delay(n_i, n_j))

Remember "important" edges during computations, they will form the critical path.

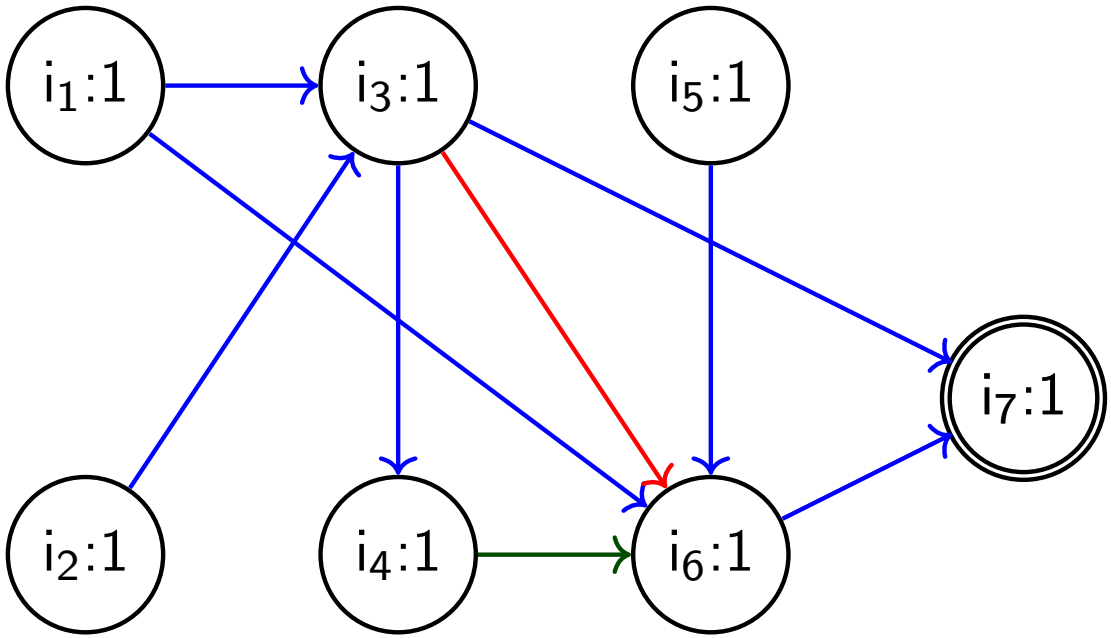
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

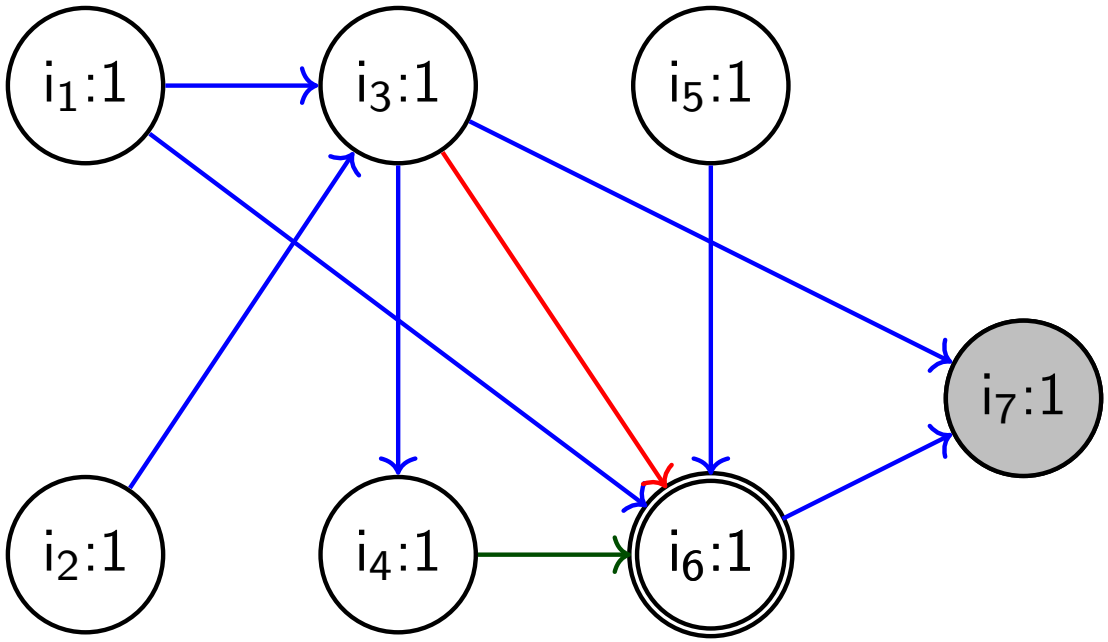
Delays: blue arrows 2, red and green 0



i_7 doesn't have successors, skip it!

Computing the critical path

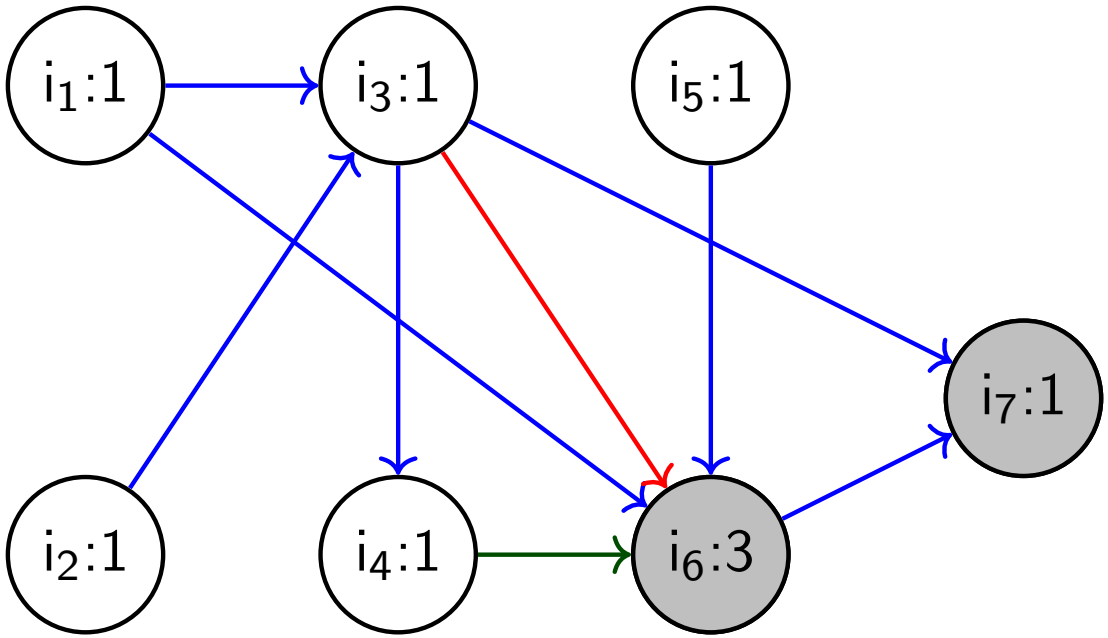
Delays: blue arrows 2, red and green 0



delay(i_6, i_7)=2 $>$ 1, change i_6 weight!

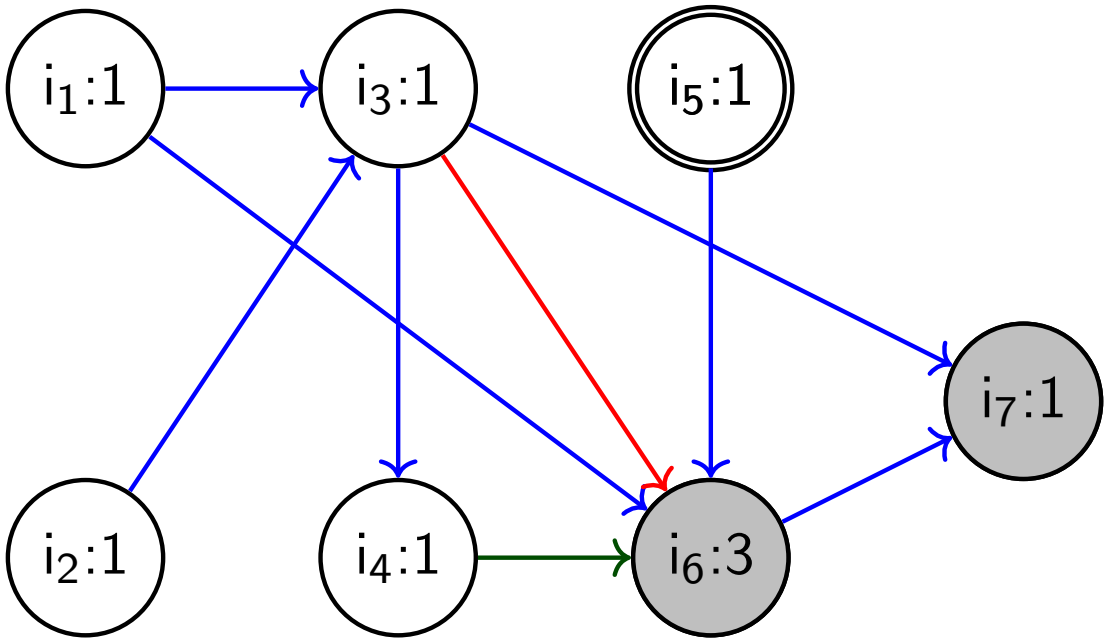
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

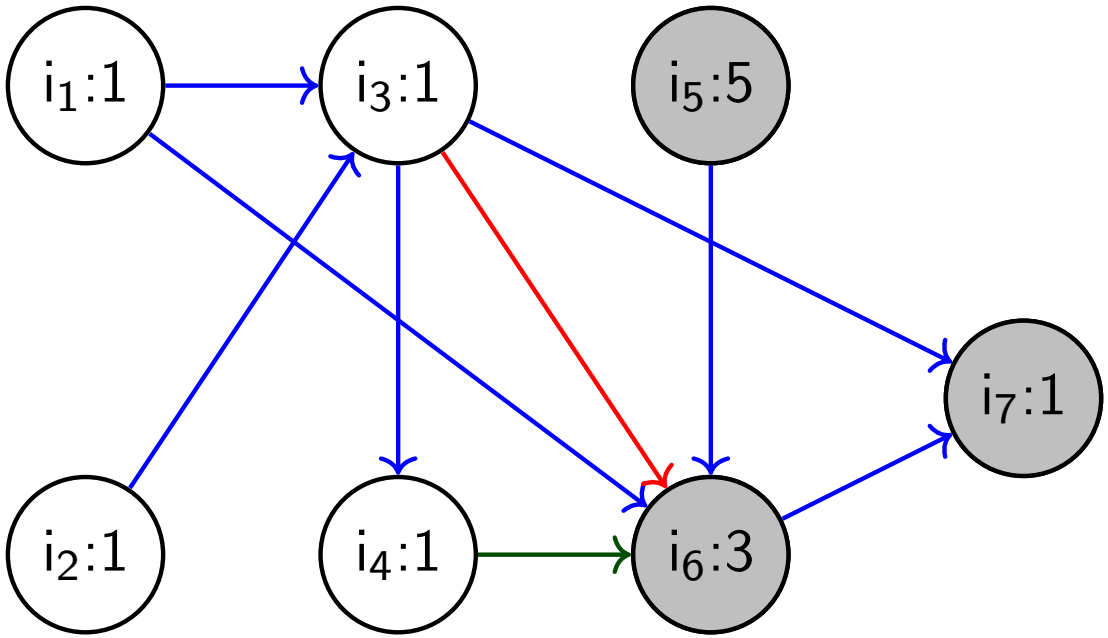
Delays: blue arrows 2, red and green 0



delay(i_5, i_6)=2 $>$ 1, change i_5 weight!

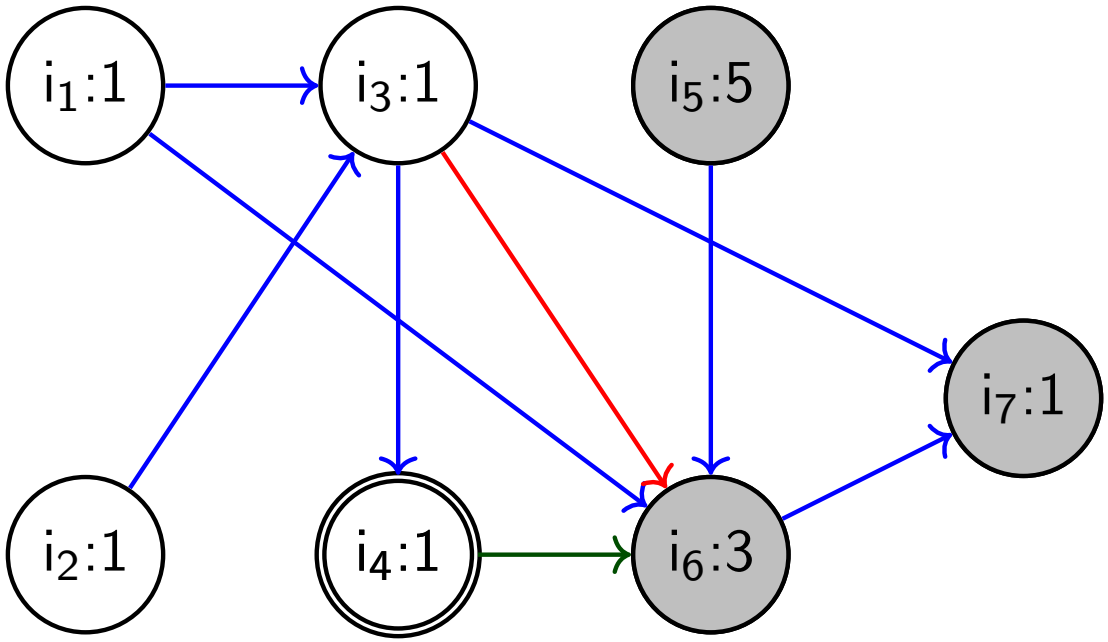
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

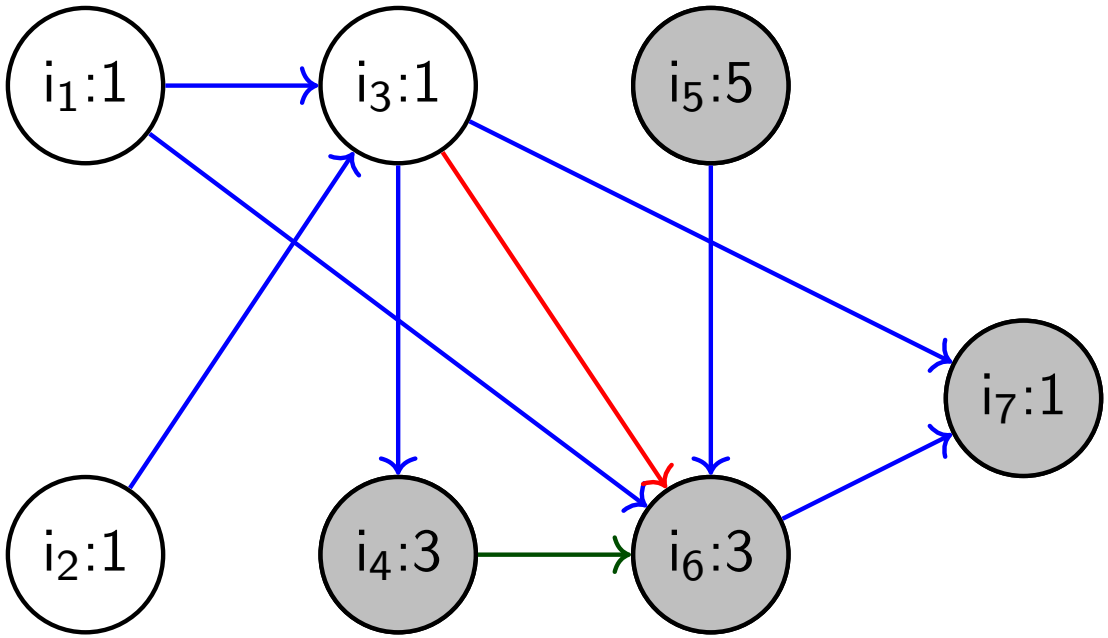
Delays: blue arrows 2, red and green 0



i_6.weight=3 $>$ 1, change i_4 weight!

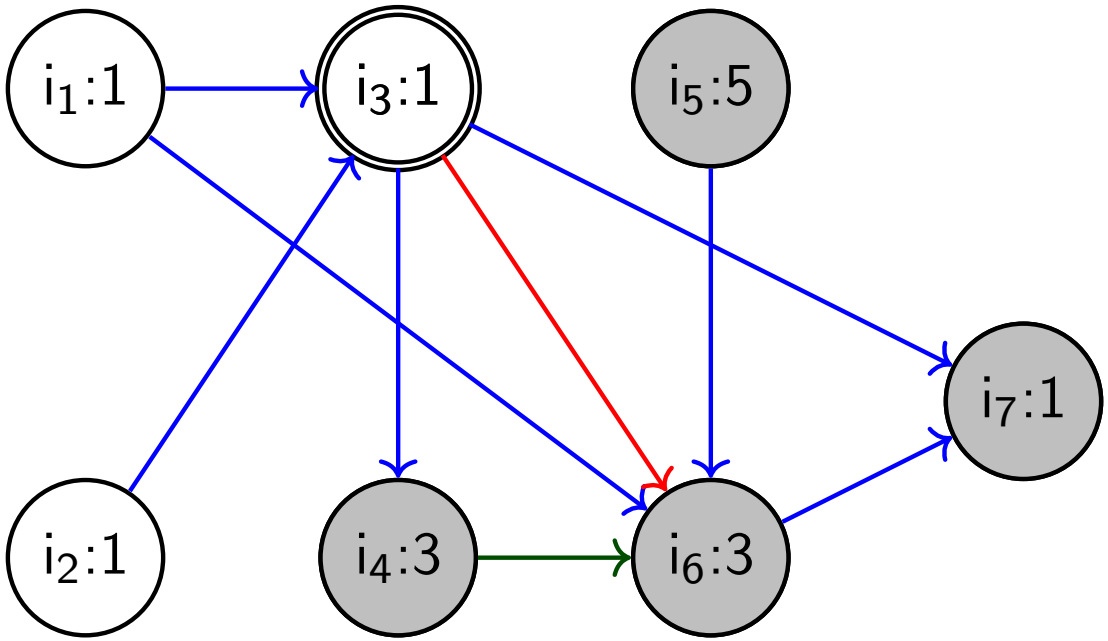
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

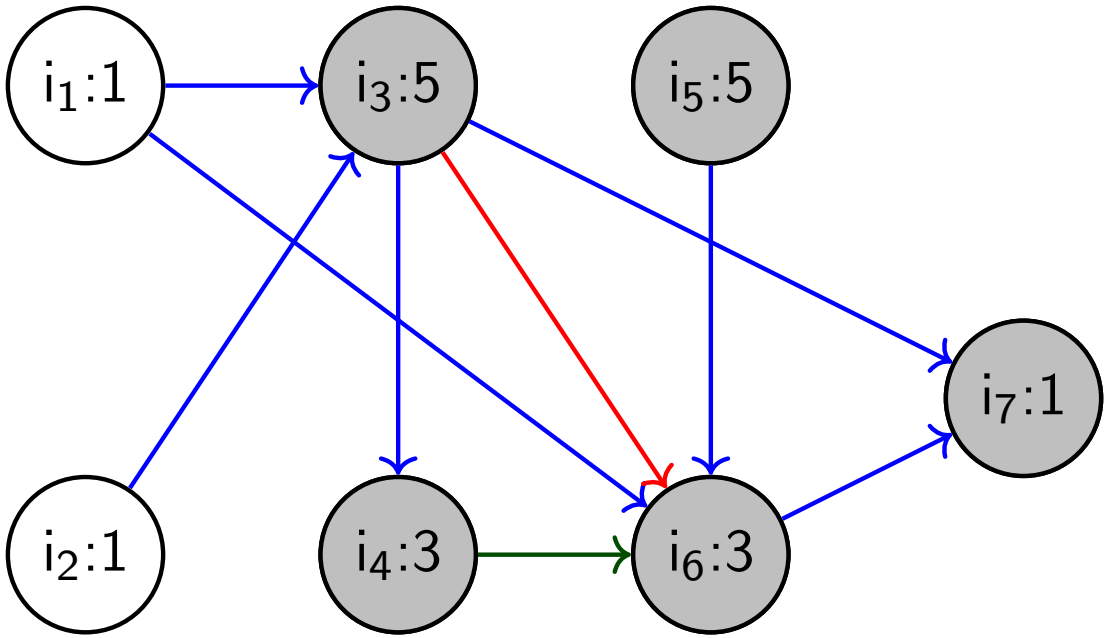
Delays: blue arrows 2, red and green 0



delay(i_3, i_4) + i_4.weight=3 $>$ 1, change i_3 weight!

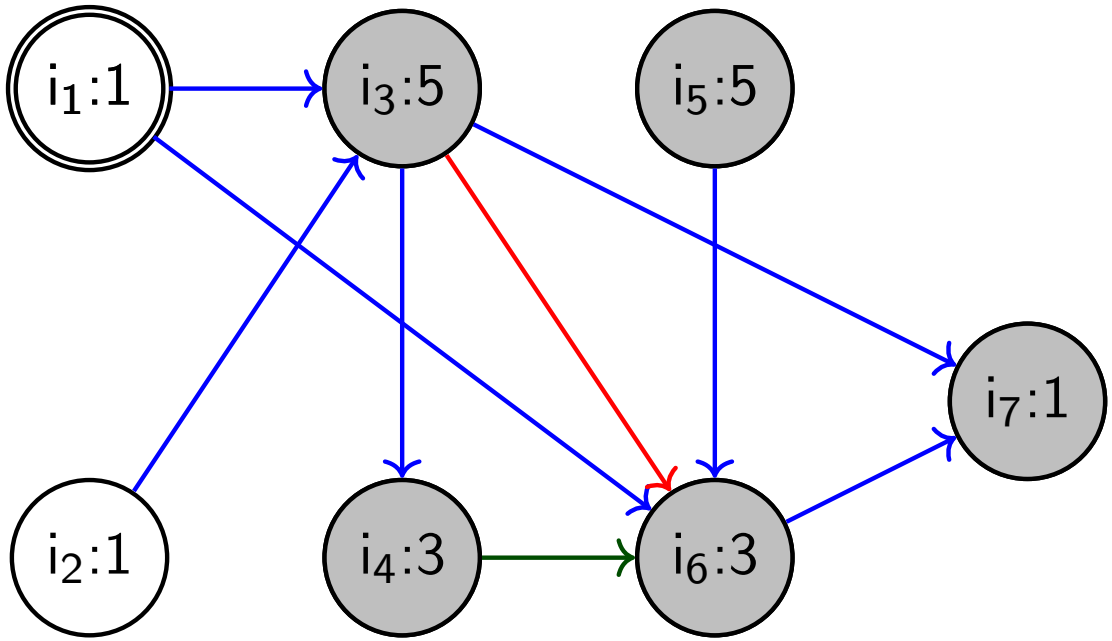
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

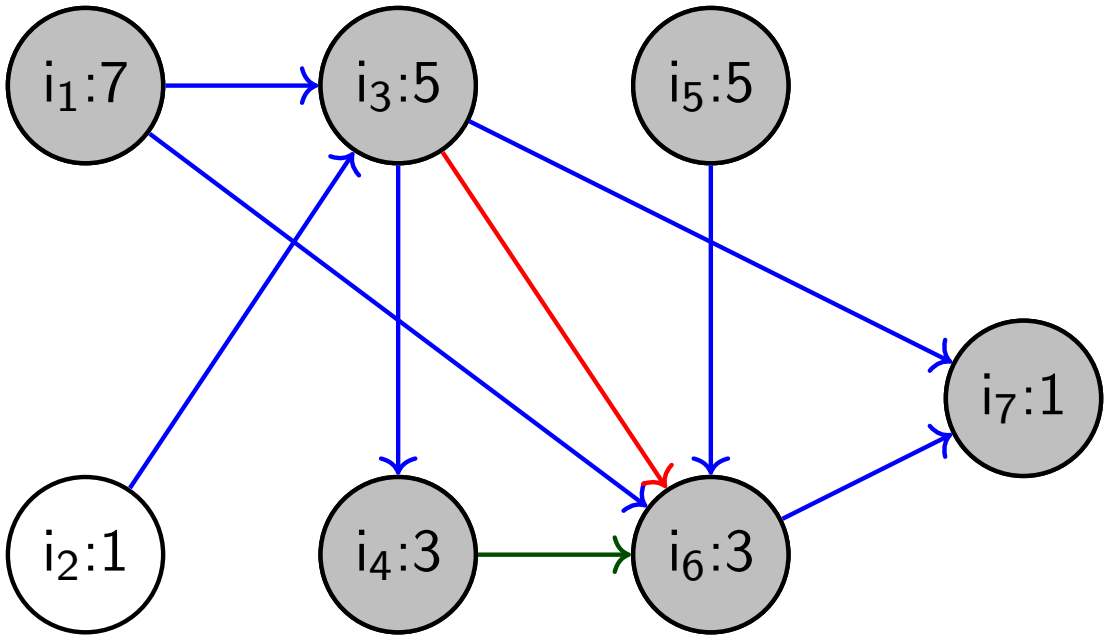
Delays: blue arrows 2, red and green 0



delay(i_1, i_3) + i_3.weight=7 $>$ 1, change i_1 weight!

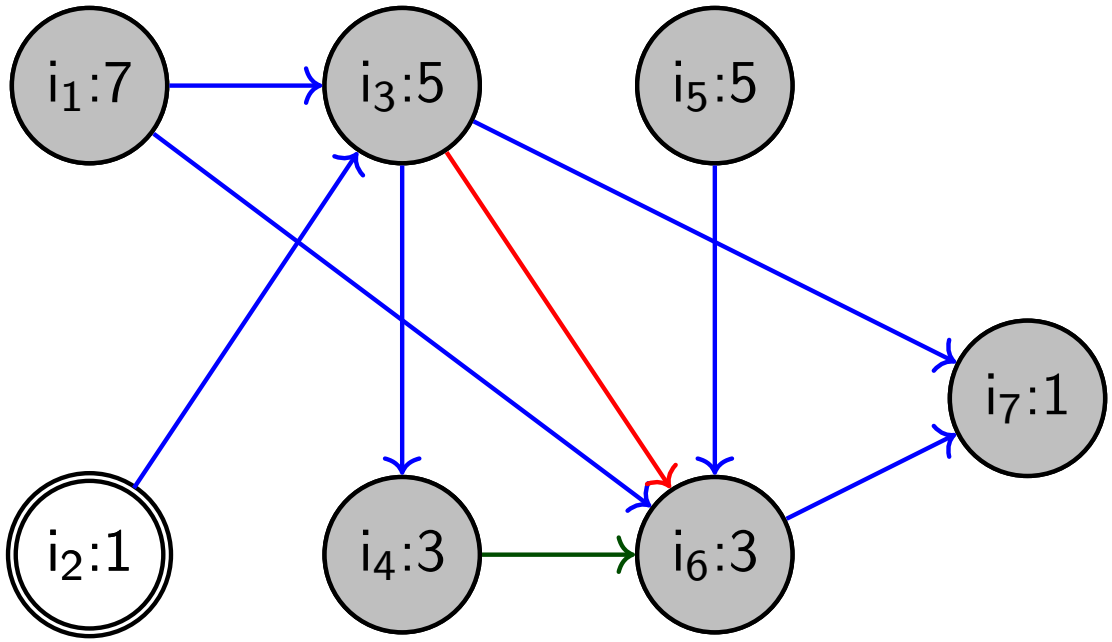
Computing the critical path

Delays: blue arrows 2, red and green 0



Computing the critical path

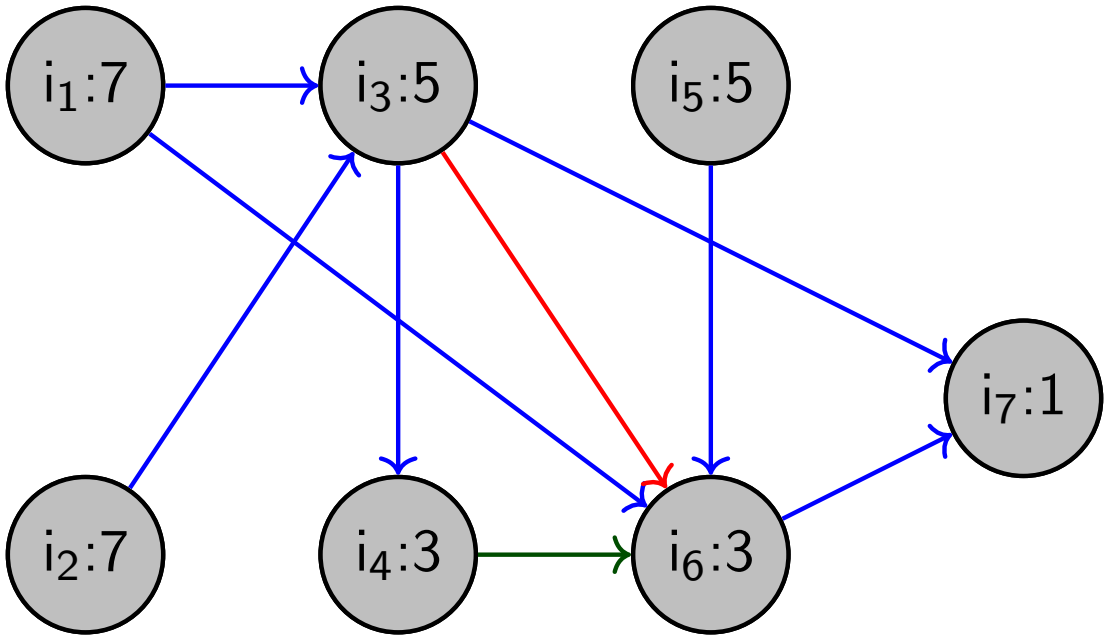
Delays: blue arrows 2, red and green 0



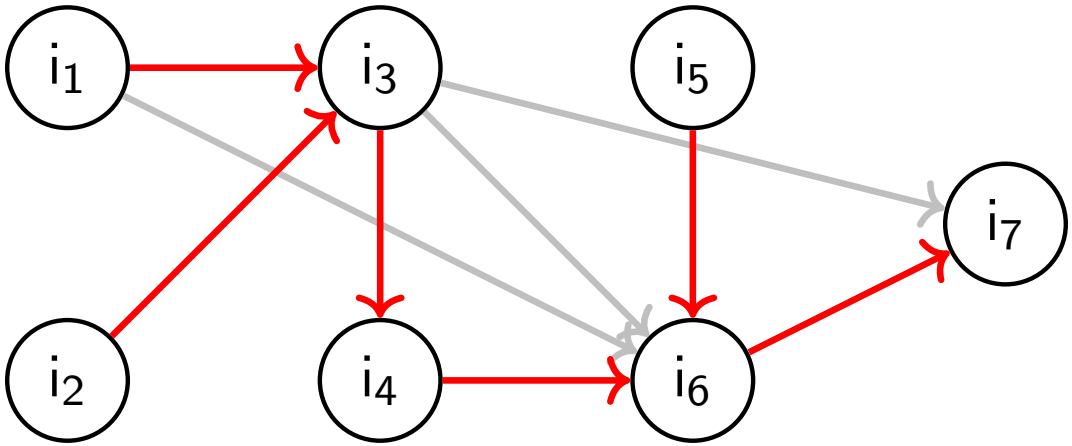
delay(i_2, i_3) + i_3.weight=7 $>$ 1, change i_2 weight!

Computing the critical path

Delays: blue arrows 2, red and green 0

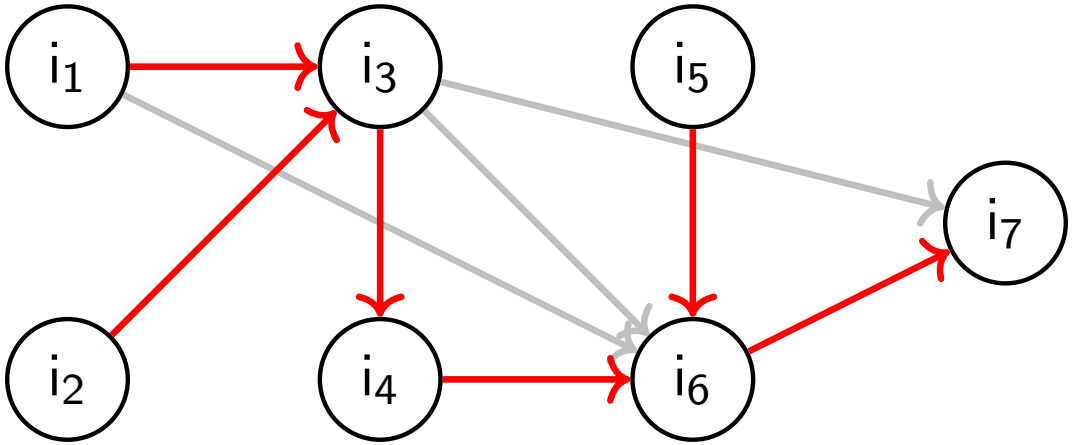


So many orders . . . with one critial path



$i_1,i_2,i_3,i_4,i_5,i_6,i_7$
$i_1,i_2,i_5,i_3,i_4,i_6,i_7$
$i_1,i_2,i_3,i_5,i_4,i_6,i_7$
$i_2,i_1,i_5,i_3,i_4,i_6,i_7$
$i_5,i_1,i_2,i_3,i_4,i_6,i_7$
$i_2,i_1,i_3,i_5,i_4,i_6,i_7$
$i_1,i_5,i_2,i_3,i_4,i_6,i_7$
$i_5,i_2,i_1,i_3,i_4,i_6,i_7$
$i_2,i_1,i_3,i_4,i_5,i_6,i_7$
$i_2,i_5,i_1,i_3,i_4,i_6,i_7$

So many orders . . . with one critial path



$i_1,i_2,i_3,i_4,i_5,i_6,i_7$
$i_1,i_2,i_5,i_3,i_4,i_6,i_7$
$i_1,i_2,i_3,i_5,i_4,i_6,i_7$
$i_2,i_1,i_5,i_3,i_4,i_6,i_7$
$i_5,i_1,i_2,i_3,i_4,i_6,i_7$
$i_2,i_1,i_3,i_5,i_4,i_6,i_7$
$i_1,i_5,i_2,i_3,i_4,i_6,i_7$
$i_5,i_2,i_1,i_3,i_4,i_6,i_7$
$i_2,i_1,i_3,i_4,i_5,i_6,i_7$
$i_2,i_5,i_1,i_3,i_4,i_6,i_7$

All these permutations respect dependencies
but is there a best instruction scheduling?

Performances and Pipeline

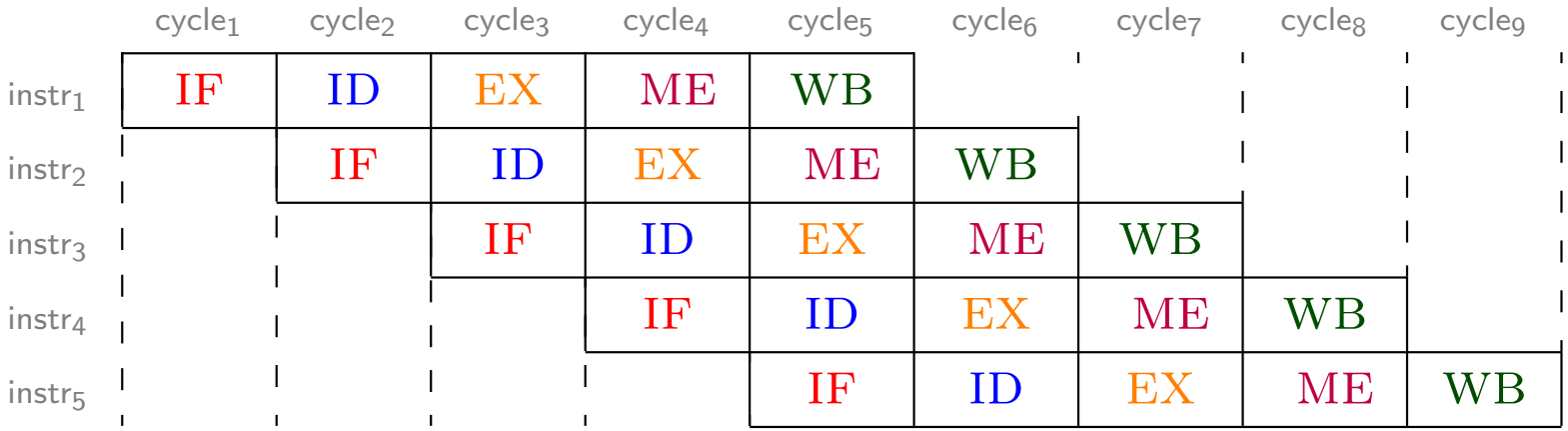
Not all orders are equivalents!

- Some dependencies can bring hazards that slow down performances inside of the pipeline
- Hazard occurs when:
  - 1 instruction requires the previous instruction has finished
  - 2 instructions need the same data at the same time: one of the two is blocked

Instructions Pipeline

The microprocessor (MIPS) contains 5 stages:

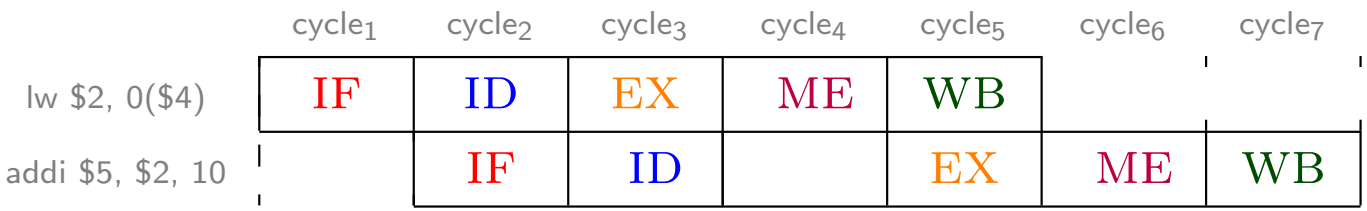
- IF: Instruction Fetch
- ID: Instruction Decode. Read operands from registers, compute the address of the next instruction
- EX Execute instructions requiring the ALU
- ME Read/write into Memory
- WB Write Back. Results are written into registers.



Hazard: RAW dependencies 1/2

Some instruction requires a result computed by a previous one!

Consider the following example:

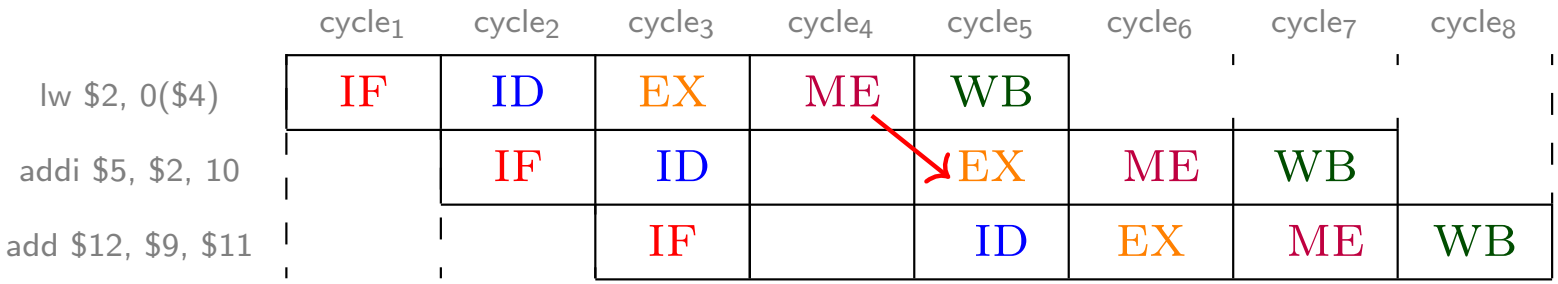


- `lw` produces its result into $2 during the ME stage
- ADDI requires $2 for the EX stage
- In this example, 1 stall (cycle 4)

The goal of `risc` architectures is to produce one per cycle!

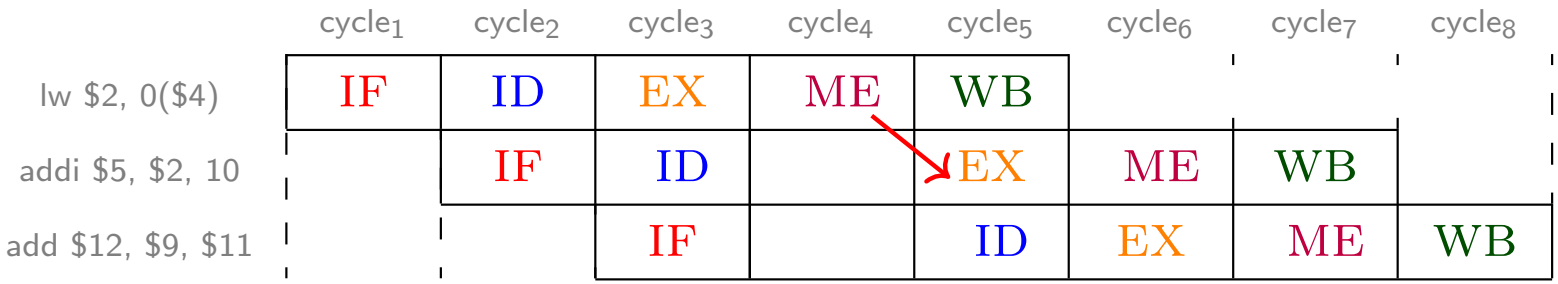
Hazard: RAW dependencies 2/2

Consider now the following example:



Hazard: RAW dependencies 2/2

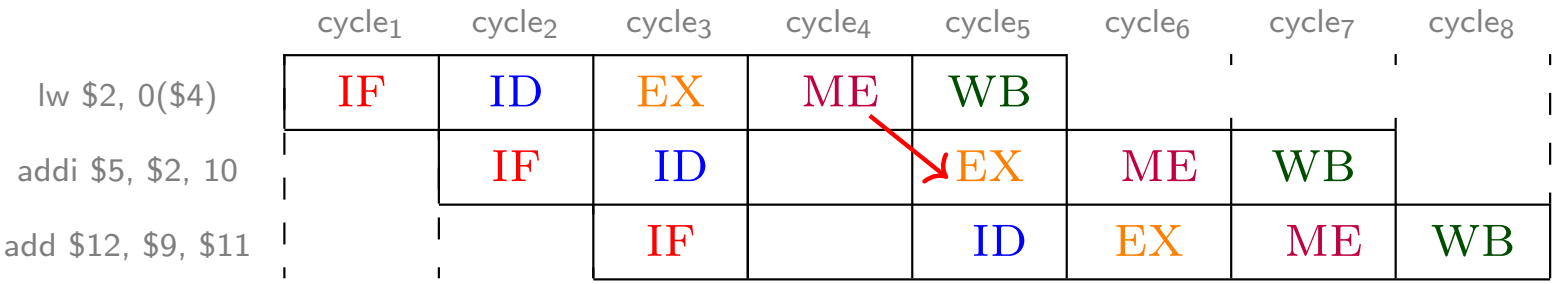
Consider now the following example:



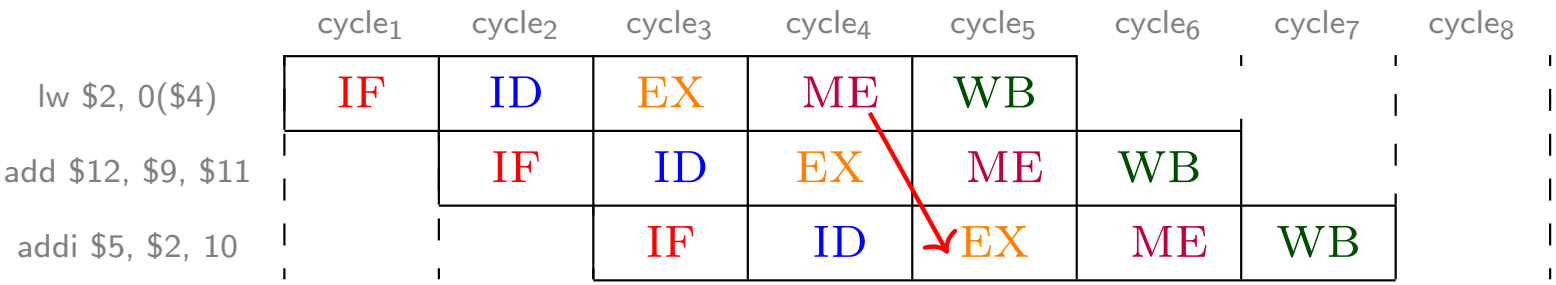
Let's look . . . instruction 3 is independent from the others

Hazard: RAW dependencies 2/2

Consider now the following example:



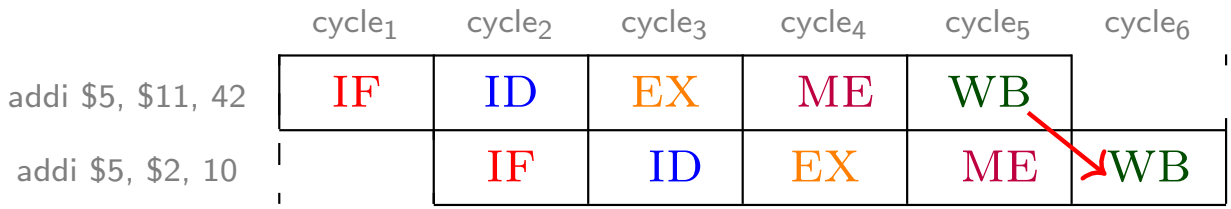
Let's look . . . instruction 3 is independent from the others so we can change the order!



Hazard: WAW dependencies

Two instructions write in the same register!

Consider the following example:

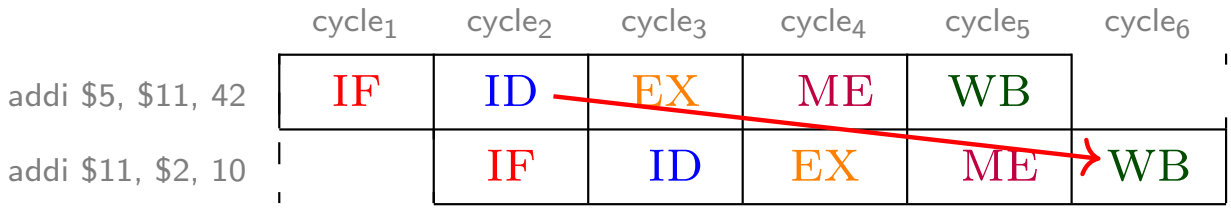


WAW do not produce stalls !
(even when writing in the same memory address)

Hazard: WAR dependencies

One instruction writes where a previous one reads!

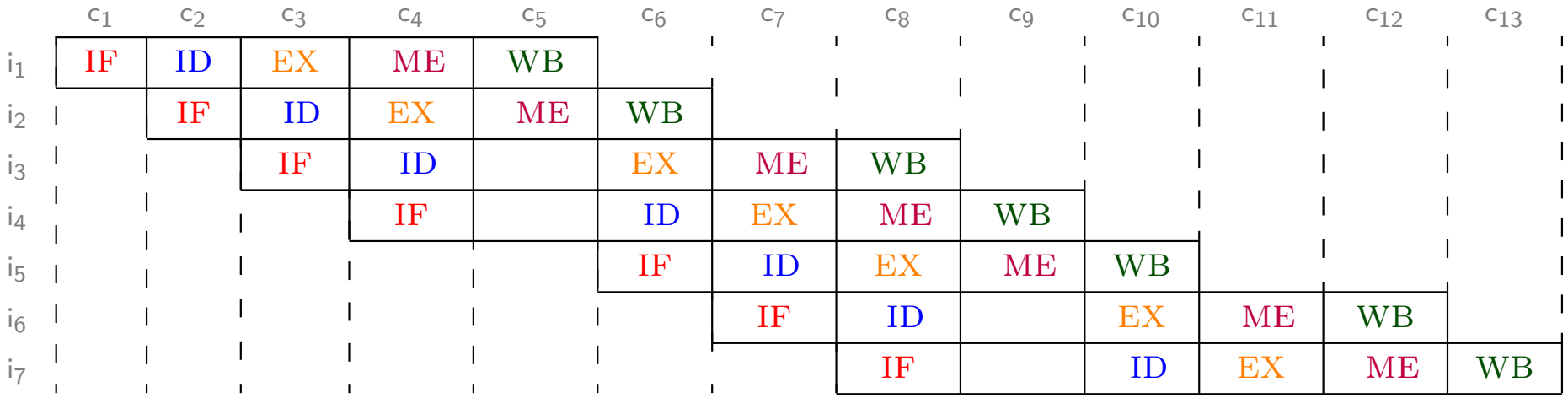
Consider the following example:



WAR do not produce stalls !

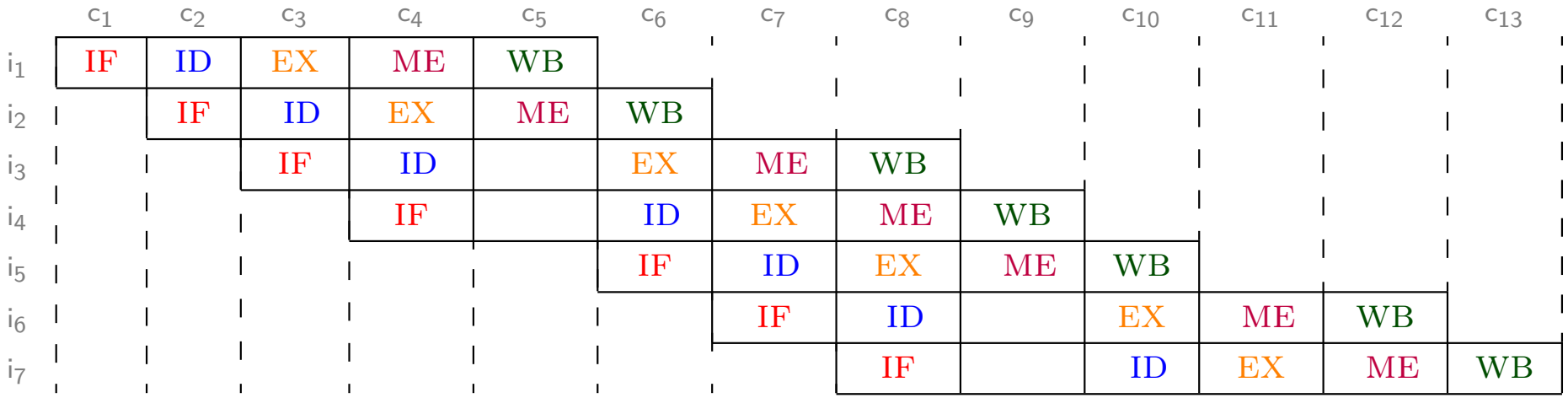
Back to the example – without scheduling

```
i1 :  lw  $1,0($10)  |  i4 :  sw  $3,12($10)  |  i7 :  sw  $3,16($10)
i2 :  lw  $2,4($10)  |  i5 :  lw  $4,8($10)   |
i3 :  add $3,$1,$2   |  i6 :  add $3,$1,$4    |
```



Back to the example – without scheduling

```
i1 :  lw   $1,0($10)  |  i4 :  sw   $3,12($10)  |  i7 :  sw   $3,16($10)
i2 :  lw   $2,4($10)  |  i5 :  lw   $4,8($10)   |
i3 :  add  $3,$1,$2   |  i6 :  add  $3,$1,$4    |
```



Without scheduling: 2 dependencies, 2 stalls, 13 cycles!

Minimizing Stalls – First approach

Each time we emit the next instruction, we should try to choose one which

- P_1 does not conflict with the previous emitted instruction
- P_2: is most likely to conflict if first of a pair (e.g. prefer `lw` to `add`)
- P_3: is as far away as possible (along paths in the DAG) from an instruction which can validly be scheduled last

Minimizing Stalls – First approach

Each time we emit the next instruction, we should try to choose one which

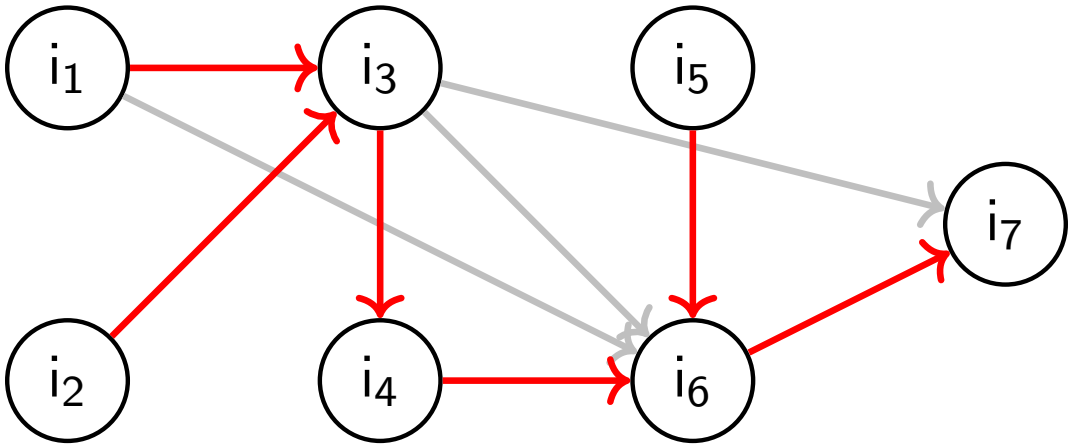
- P_1 does not conflict with the previous emitted instruction
- P_2: is most likely to conflict if first of a pair (e.g. prefer `lw` to `add`)
- P_3: is as far away as possible (along paths in the DAG) from an instruction which can validly be scheduled last

Algorithm:

- Compute the dependency graph
- While the list of candidate instructions is not empty
  - If one instruction satisfies P_1, P_2, and P_3: remove it from the list and emit it.
    - ★ Remove the instruction from the DAG and insert the newly minimal elements into the candidate list.
  - Otherwise emit a `nop` instruction

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

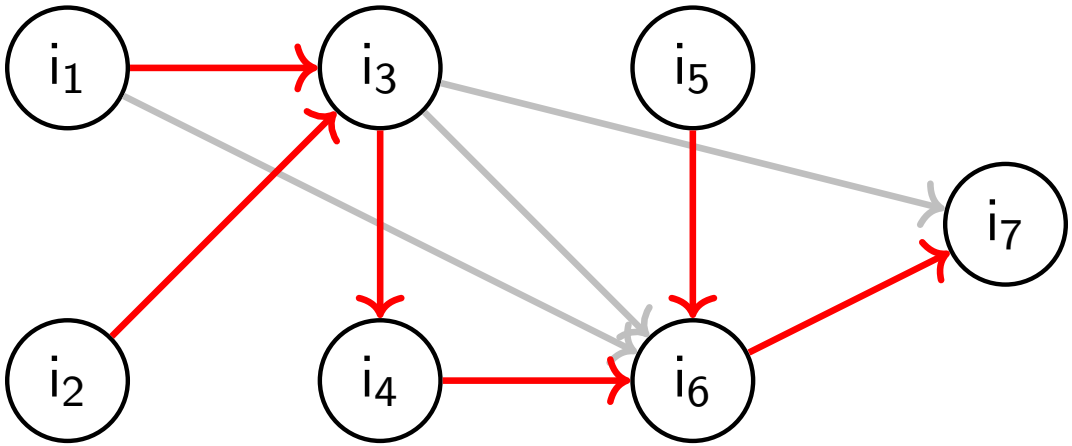


Candidates = $\{i_1, i_2, i_5\}$

Final Order =

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



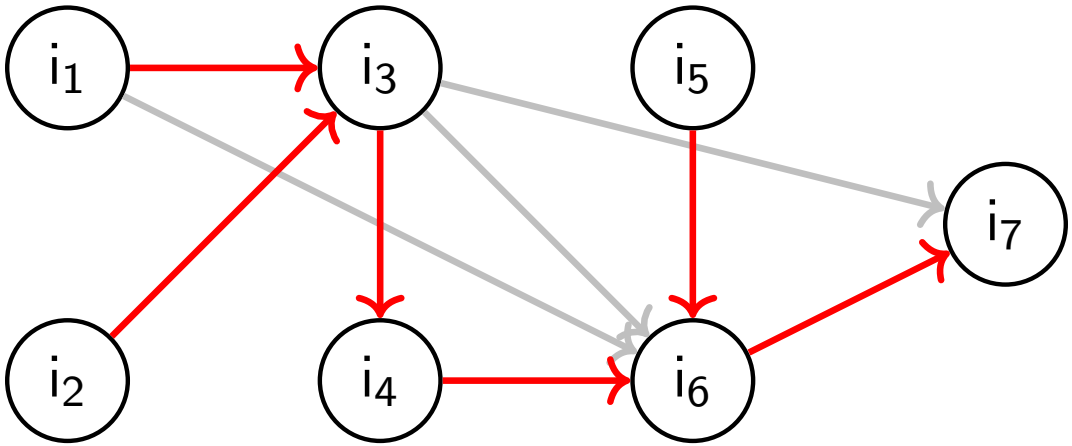
Candidates = $\{i_1, i_2, i_5\}$

Final Order =

Choose i_1 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



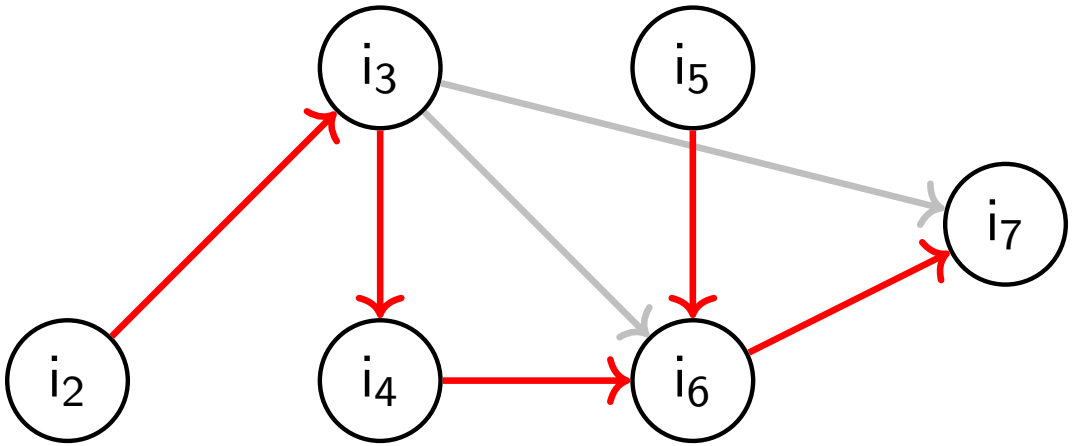
Candidates = $\{i_1, i_2, i_5\}$

Final Order = i_1

Choose i_1 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



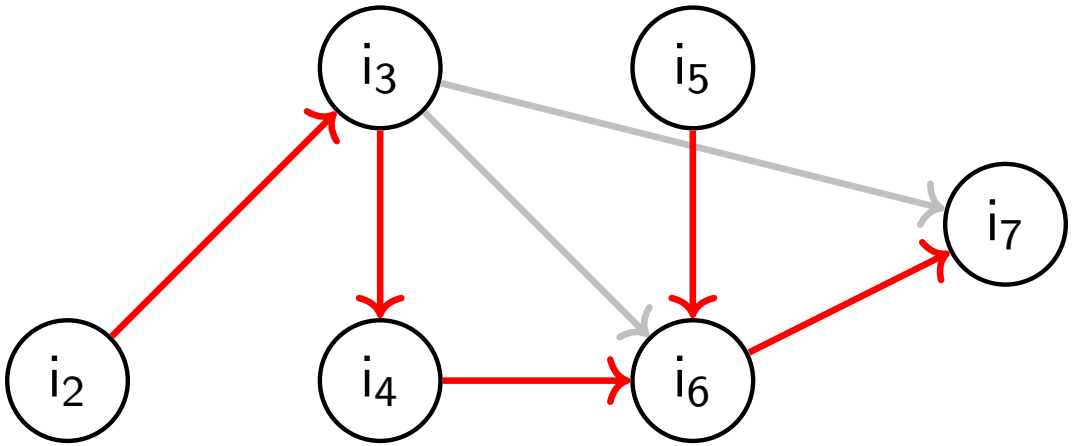
Candidates = $\{i_1, i_2, i_5\}$

Final Order = i_1

Choose i_1 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

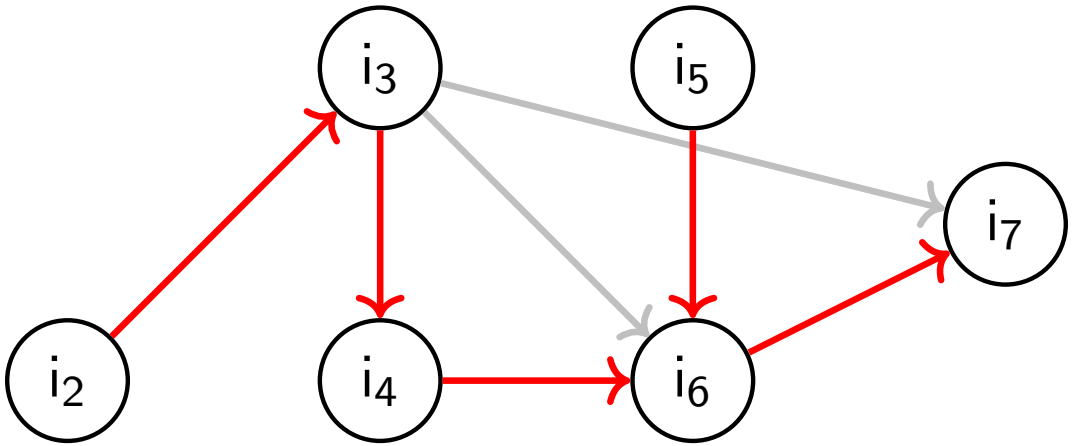


Candidates = $\{i_2, i_5\}$

Final Order = i_1

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



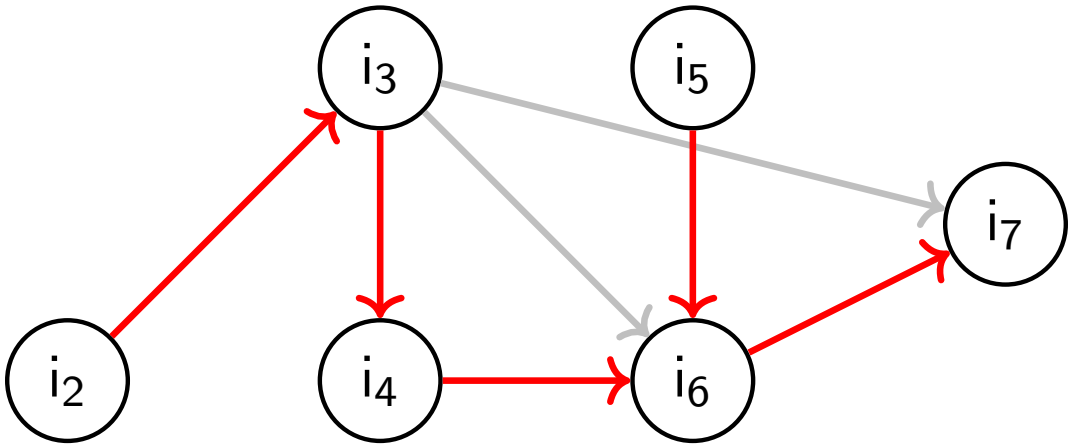
Candidates = $\{i_2, i_5\}$

Final Order = i_1

Choose i_2 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



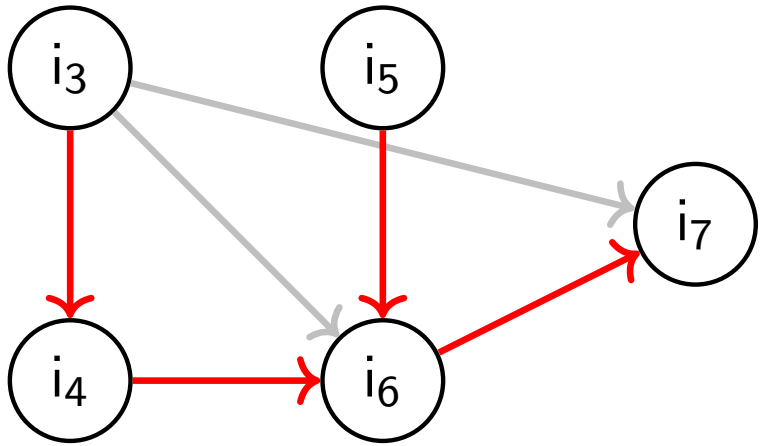
Candidates = $\{i_2, i_5\}$

Final Order = i_1, i_2

Choose i_2 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



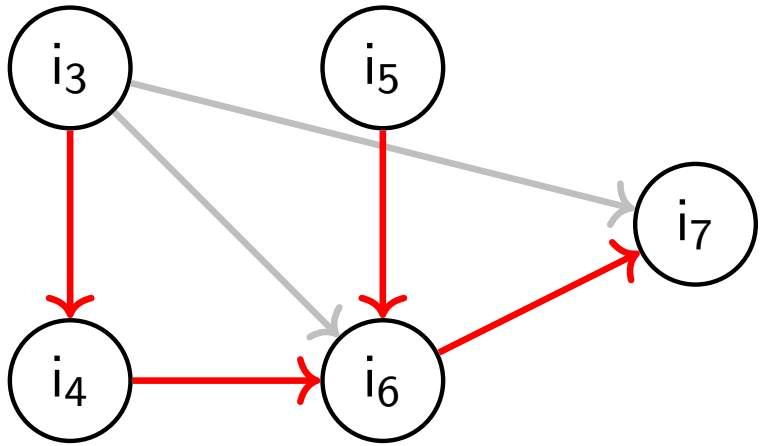
Candidates = $\{i_2, i_5\}$

Final Order = i_1, i_2

Choose i_2 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

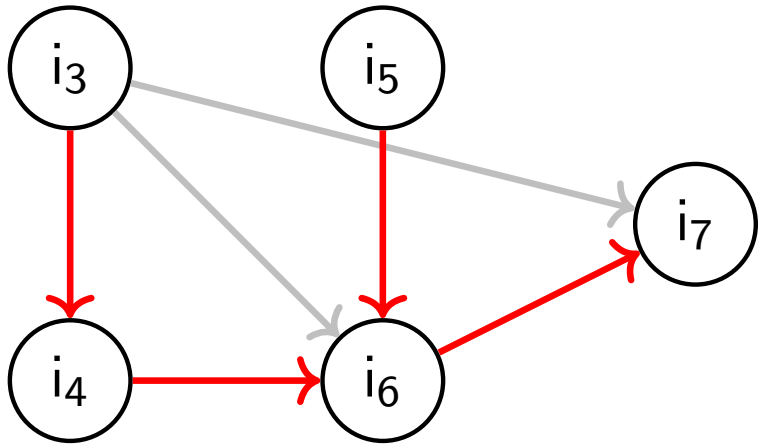


Candidates = $\{i_5, i_3\}$

Final Order = i_1, i_2

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

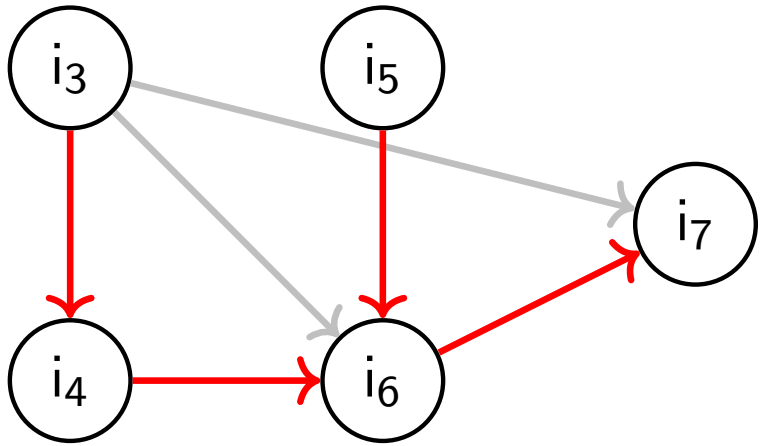


Candidates = $\{i_5, i_3\}$
Final Order = i_1, i_2

Choose i_5 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



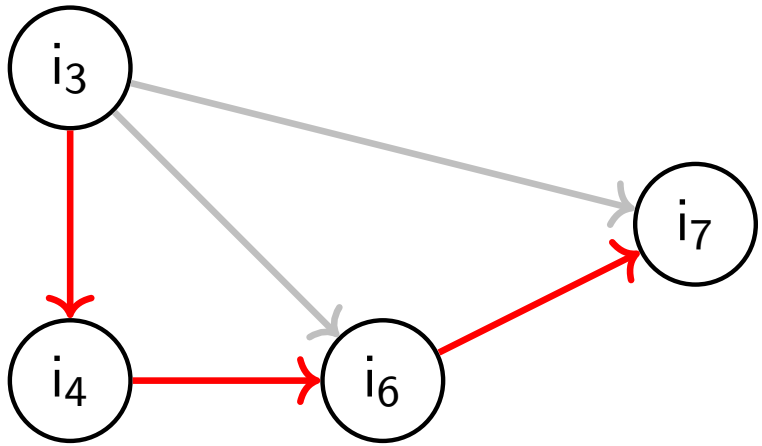
Candidates = $\{i_5, i_3\}$

Final Order = i_1, i_2 , i_5

Choose i_5 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



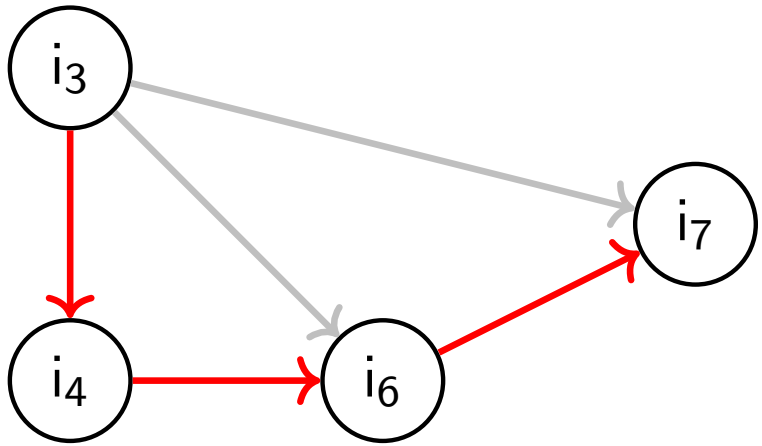
Candidates = $\{i_5, i_3\}$

Final Order = i_1, i_2 , i_5

Choose i_5 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

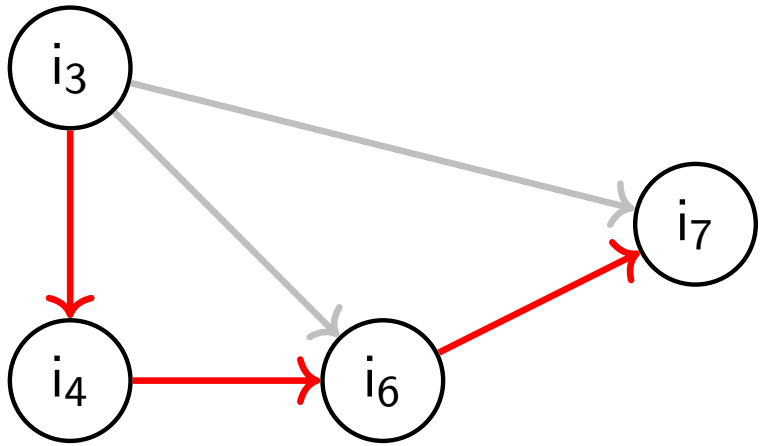


Candidates = $\{i_3\}$

Final Order = i_1, i_2, i_5

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

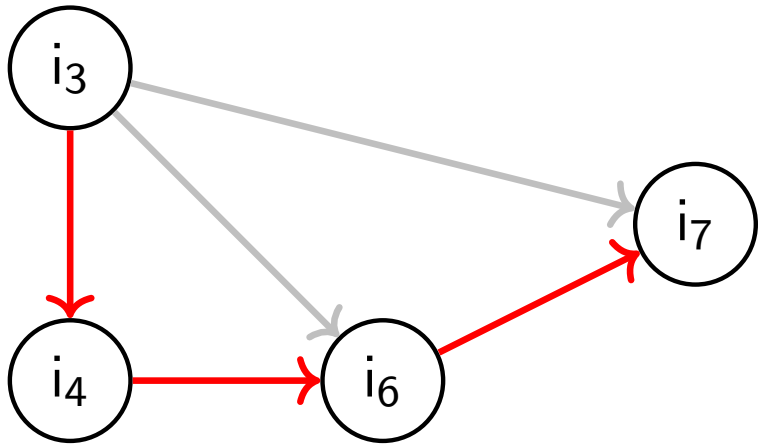


Candidates = $\{i_3\}$
Final Order = i_1, i_2, i_5

Choose i_3 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	



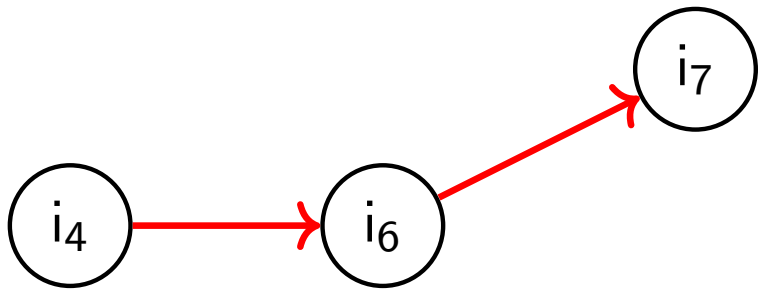
Candidates = $\{i_3\}$

Final Order = i_1, i_2, i_5, i_3

Choose i_3 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



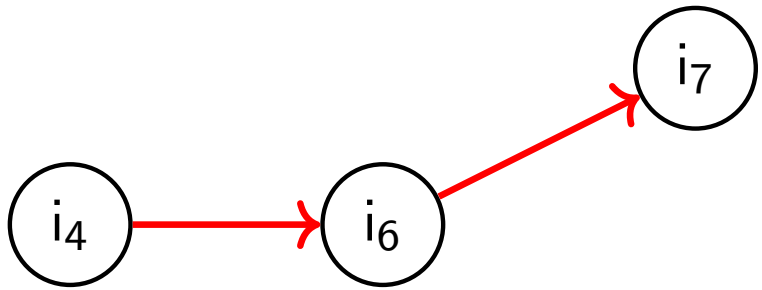
Candidates = $\{i_3\}$

Final Order = i_1, i_2, i_5, i_3

Choose i_3 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

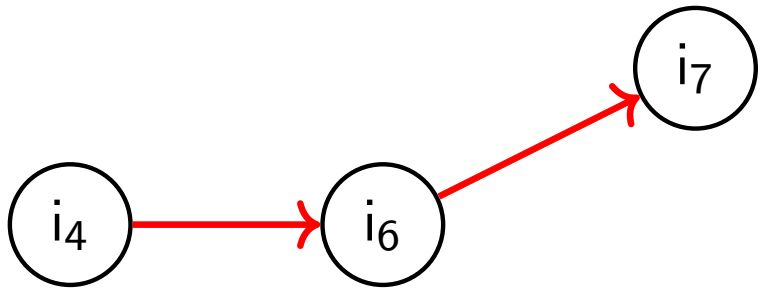


Candidates = $\{i_4\}$

Final Order = i_1, i_2, i_5, i_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



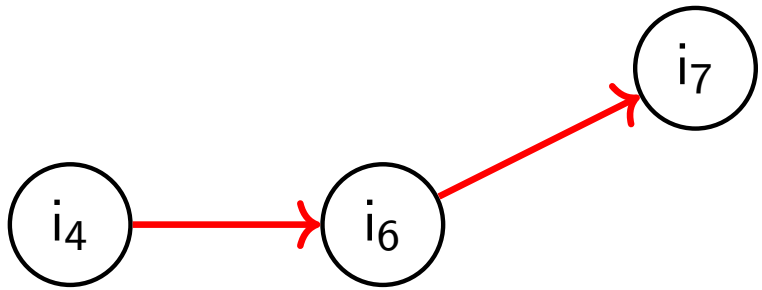
Candidates = $\{i_4\}$

Final Order = i_1, i_2, i_5, i_3

Choose i_4 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	lw	$1,0($10)	i_4 :	sw	$3,12($10)	i_7 :	sw	$3,16($10)
i_2 :	lw	$2,4($10)	i_5 :	lw	$4,8($10)			
i_3 :	add	$3,$1,$2	i_6 :	add	$3,$1,$4			



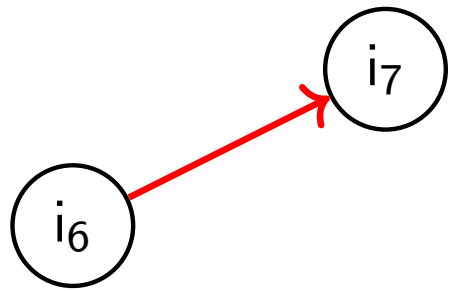
Candidates = $\{i_4\}$

Final Order = i_1, i_2, i_5, i_3, i_4

Choose i_4 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



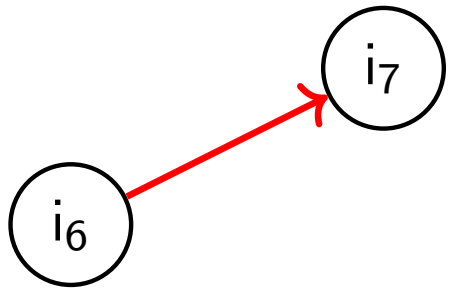
Candidates = $\{i_4\}$

Final Order = i_1, i_2, i_5, i_3, i_4

Choose i_4 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : `lw $1,0($10)`	i_4 : `sw $3,12($10)`	i_7 : `sw $3,16($10)`
i_2 : `lw $2,4($10)`	i_5 : `lw $4,8($10)`	
i_3 : `add $3,$1,$2`	i_6 : `add $3,$1,$4`	

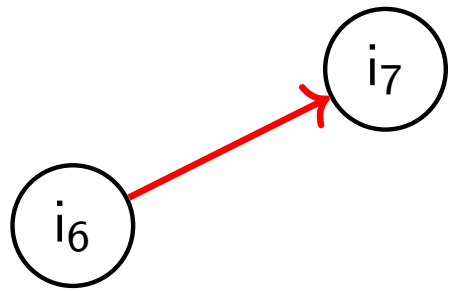


Candidates = $\{i_6\}$

Final Order = i_1, i_2, i_5, i_3, i_4

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



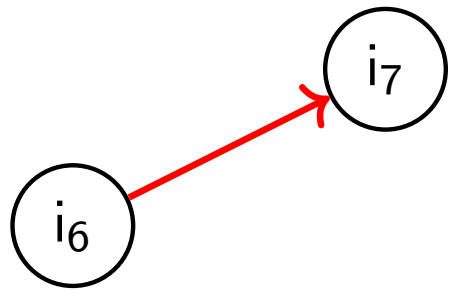
Candidates = $\{i_6\}$

Final Order = i_1, i_2, i_5, i_3, i_4

Choose i_6 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	



Candidates = $\{i_6\}$

Final Order = $i_1, i_2, i_5, i_3, i_4, i_6$

Choose i_6 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

Candidates = $\{i_6\}$

Final Order = $i_1, i_2, i_5, i_3, i_4, i_6$

Choose i_6 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

Candidates = $\{i_7\}$

Final Order = $i_1, i_2, i_5, i_3, i_4, i_6$

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

Candidates = $\{i_7\}$

Final Order = $i_1, i_2, i_5, i_3, i_4, i_6$

Choose i_7 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

Candidates = $\{i_7\}$

Final Order = $i_1, i_2, i_5, i_3, i_4, i_6, i_7$

Choose i_7 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

Candidates = $\{i_7\}$

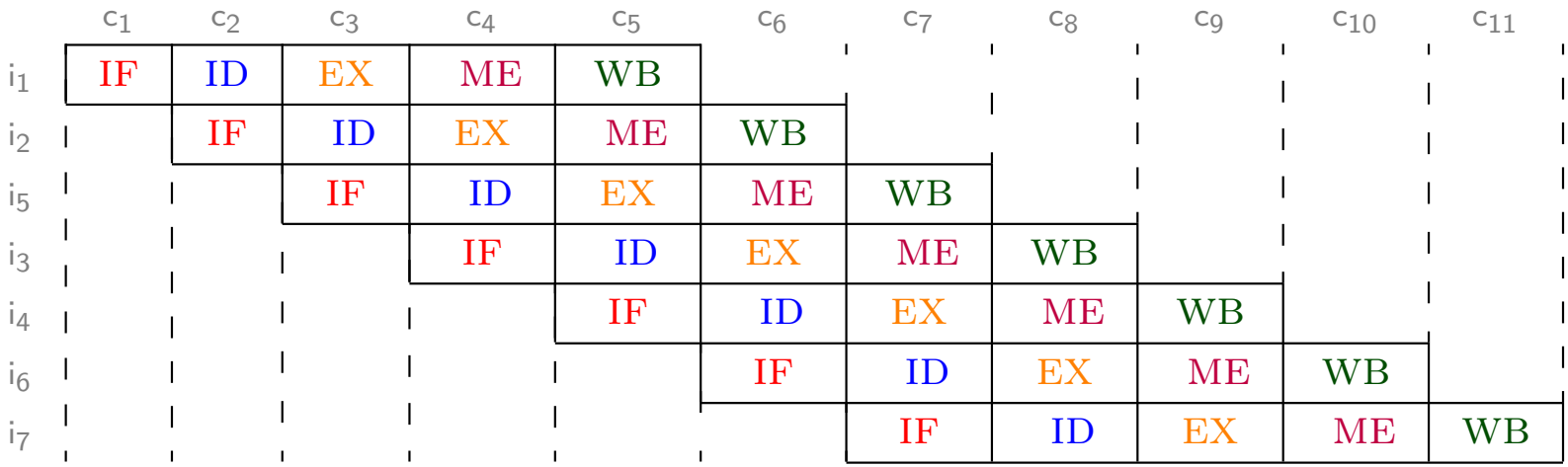
Final Order = $i_1, i_2, i_5, i_3, i_4, i_6, i_7$

Choose i_7 since it satisfies P_1, P_2 and P_3

Applying scheduling algorithm to the example

i_1 :	`lw`	`$1,0($10)`	i_4 :	`sw`	`$3,12($10)`	i_7 :	`sw`	`$3,16($10)`
i_2 :	`lw`	`$2,4($10)`	i_5 :	`lw`	`$4,8($10)`			
i_3 :	`add`	`$3,$1,$2`	i_6 :	`add`	`$3,$1,$4`			

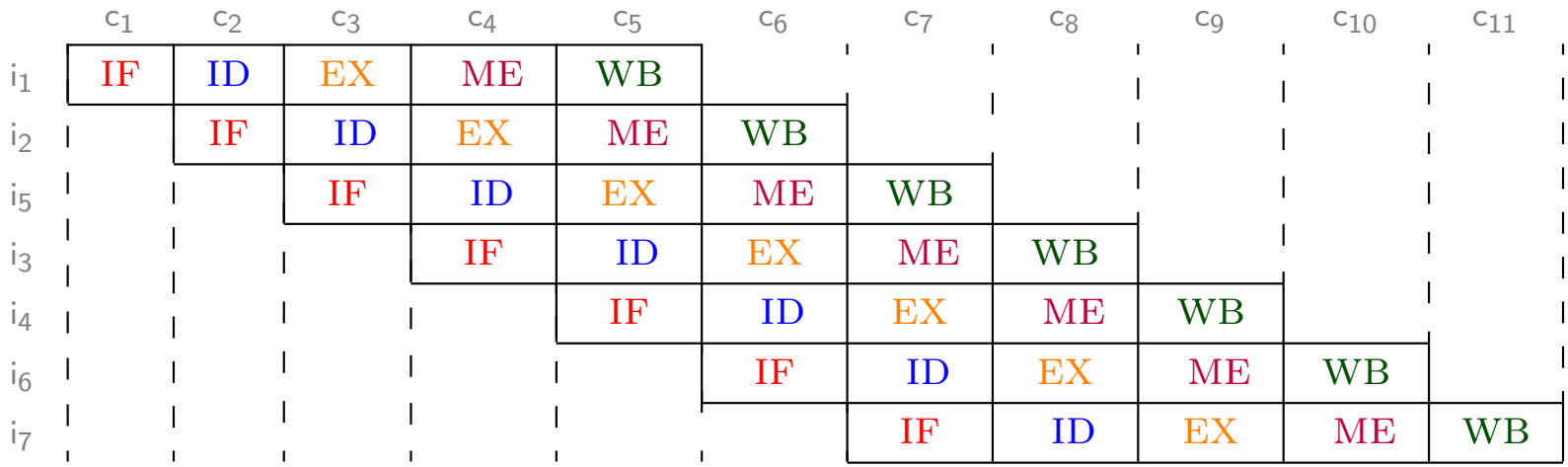
Final Order = i_1, i_2, i_5, i_3, i_4, i_6, i_7



Applying scheduling algorithm to the example

i_1 : lw $1,0($10)	i_4 : sw $3,12($10)	i_7 : sw $3,16($10)
i_2 : lw $2,4($10)	i_5 : lw $4,8($10)	
i_3 : add $3,$1,$2	i_6 : add $3,$1,$4	

Final Order = i_1, i_2, i_5, i_3, i_4, i_6, i_7



With scheduling: still 2 dependencies but 0 stalls and 11 cycles!

A word on scheduling strategies

- Sometimes we cannot avoid some stalls
- Computing the critical path can be smarter:
  - Rather than attributing 1 as weight to every instruction, we can adjust according to the real time of executing the instruction
  - We can take advantages of the number of successors
  - ... many yet-to-be-define heuristics!
- Computing the DAG of dependencies can be done in $O(n^2)$ by scanning backwards through the basic block and adding edges as dependencies arise

A word on performances

We can statically compute instructions per cycle IPC$=\frac{\text{nb instructions}}{\text{nb cycles}}$, to evaluate 2 possible scheduling.

In the previous example:

- without scheduling IPC$=\frac{7}{13} = 0.53$
- with scheduling IPC$=\frac{7}{11} = 0.63$ (better!)

We can also statically compute cycle per instructions: CPI $= \frac{1}{\text{IPC}}$.
The CPI lower bound is $\frac{\sum \alpha \times \beta}{\text{nb instructions}}$, avec α is the number of instructions for a given instruction type and β the associated cost.

Can we do better?

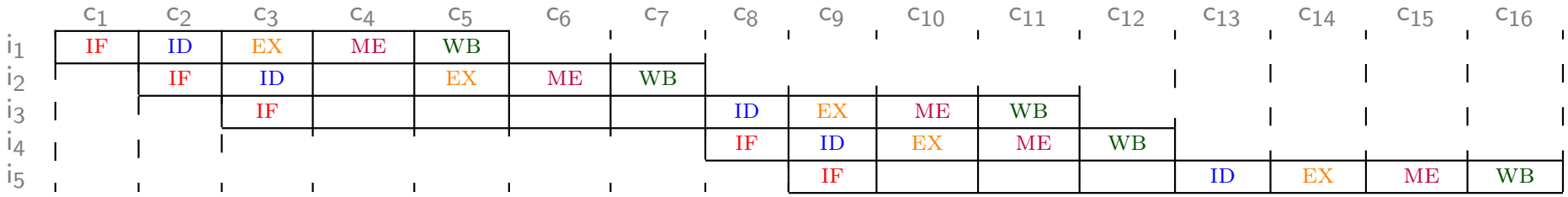
Consider the following code (representing a basic block):

```
i1:  Loop:  lw    $t0, 0($s1)     # t0=array element
i2:         addu  $t0, $t0, $s2   # add scalar in s2
i3:         sw    $t0, 0($s1)     # store result
i4:         addi  $s1, $s1,-4     # decrement pointer
i5:         bne   $s1, $0, Loop   # branch s1!=0
```

Can we do better?

Consider the following code (representing a basic block):

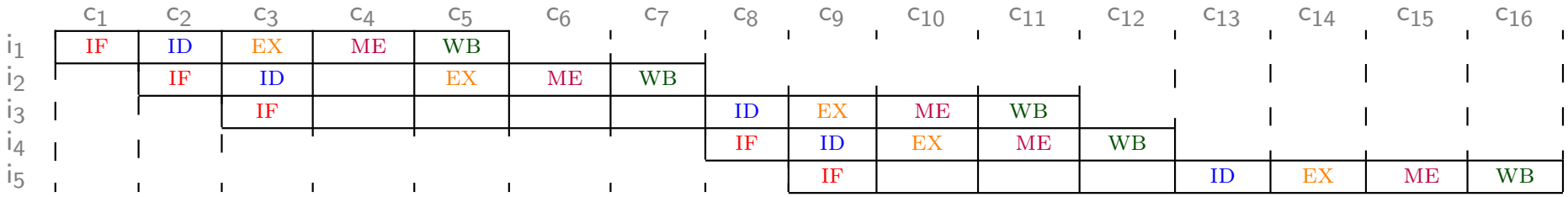
```
i1:  Loop:  lw    $t0, 0($s1)     # t0=array element
i2:         addu  $t0, $t0, $s2   # add scalar in s2
i3:         sw    $t0, 0($s1)     # store result
i4:         addi  $s1, $s1,-4     # decrement pointer
i5:         bne   $s1, $0, Loop   # branch s1!=0
```



Can we do better?

Consider the following code (representing a basic block):

```
i1:  Loop:  lw    $t0, 0($s1)     # t0=array element
i2:         addu  $t0, $t0, $s2   # add scalar in s2
i3:         sw    $t0, 0($s1)     # store result
i4:         addi  $s1, $s1,-4     # decrement pointer
i5:         bne   $s1, $0, Loop   # branch s1!=0
```



16 cycles for 5 instructions that are all dependent!

IPC = 0.31

Loop Unrolling

- Replicate loop body to expose more parallelism
- Reduces loop-control overhead

Loop Unrolling

- Replicate loop body to expose more parallelism
- Reduces loop-control overhead

At high level, it can be seen as following:

Without Loop Unrolling	With Loop Unrolling
int i;	int i;
for (i = 0; i < 100; ++i)	for (i = 0; i < 100; i+=5)
tab[i] = tab[i] +42;	tab[i] = tab[i] +42;
	tab[i+1] = tab[i+1] +42;
	tab[i+2] = tab[i+2] +42;
	tab[i+3] = tab[i+3] +42;
	tab[i+4] = tab[i+4] +42;

Loop Unrolling

- Replicate loop body to expose more parallelism
- Reduces loop-control overhead

At high level, it can be seen as following:

Without Loop Unrolling	With Loop Unrolling
int i;	int i;
for (i = 0; i < 100; ++i)	for (i = 0; i < 100; i+=5)
tab[i] = tab[i] +42;	tab[i] = tab[i] +42;
	tab[i+1] = tab[i+1] +42;
	tab[i+2] = tab[i+2] +42;
	tab[i+3] = tab[i+3] +42;
	tab[i+4] = tab[i+4] +42;

Special care must be taken for pre and post loops operations (as well as intra-loop dependencies)

Loop Unrolling – back to the example

```
i1:   Loop:  lw    $t0, 0($s1)    # t0=array element
i2:          addu  $t0, $t0, $s2  # add scalar in s2
i3:          sw    $t0, 0($s1)    # store result
i4:          addi  $s1, $s1,-4    # decrement pointer
i5:          bne   $s1, $0, Loop  # branch s1!=0
i6:   Loop:  lw    $t0, 0($s1)    # t0=array element
i7:          addu  $t0, $t0, $s2  # add scalar in s2
i8:          sw    $t0, 0($s1)    # store result
i9:          addi  $s1, $s1,-4    # decrement pointer
i10:         bne   $s1, $0, Loop  # branch s1!=0
i11:  Loop:  lw    $t0, 0($s1)    # t0=array element
i12:         addu  $t0, $t0, $s2  # add scalar in s2
i13:         sw    $t0, 0($s1)    # store result
i14:         addi  $s1, $s1,-4    # decrement pointer
i15:         bne   $s1, $0, Loop  # branch s1!=0
```

First duplicate N times the the body of the loop!

Loop Unrolling – back to the example

```
i1:   Loop:  lw    $t0, 0($s1)     # t0=array element
i2:          addu  $t0, $t0, $s2   # add scalar in s2
i3:          sw    $t0, 0($s1)     # store result
i4:          addi  $s1, $s1,-4     # decrement pointer
i6:          lw    $t0, 0($s1)     # t0=array element
i7:          addu  $t0, $t0, $s2   # add scalar in s2
i8:          sw    $t0, 0($s1)     # store result
i9:          addi  $s1, $s1,-4     # decrement pointer
i11:         lw    $t0, 0($s1)     # t0=array element
i12:         addu  $t0, $t0, $s2   # add scalar in s2
i13:         sw    $t0, 0($s1)     # store result
i14:         addi  $s1, $s1,-4     # decrement pointer
i15:         bne   $s1, $0, Loop   # branch s1!=0
```

Remove redundant labels and jump
(by supposing that we are able to do it!)

Loop Unrolling – back to the example

```
i1:   Loop:  lw    $t0, 0($s1)     # t0=array element
i2:          addu  $t0, $t0, $s2   # add scalar in s2
i3:          sw    $t0, 0($s1)     # store result
i4:          addi  $s1, $s1,-4     # decrement pointer
i6:          lw    $t1, 0($s1)     # t0=array element
i7:          addu  $t1, $t1, $s2   # add scalar in s2
i8:          sw    $t1, 0($s1)     # store result
i9:          addi  $s1, $s1,-4     # decrement pointer
i11:         lw    $t2, 0($s1)     # t0=array element
i12:         addu  $t2, $t2, $s2   # add scalar in s2
i13:         sw    $t2, 0($s1)     # store result
i14:         addi  $s1, $s1,-4     # decrement pointer
i15:         bne   $s1, $0, Loop   # branch s1!=0
```

Use other temporaries name when possible!

Loop Unrolling – back to the example

```
i4:   Loop:  addi   $s1, $s1,-12     # decrement pointer
i1:          lw     $t0, 0($s1)      # t0=array element
i2:          addu   $t0, $t0, $s2    # add scalar in s2
i3:          sw     $t0, 0($s1)      # store result
i6:          lw     $t1, 4($s1)      # t0=array element
i7:          addu   $t1, $t1, $s2    # add scalar in s2
i8:          sw     $t1, 4($s1)      # store result
i11:         lw     $t2, 8($s1)      # t0=array element
i12:         addu   $t2, $t2, $s2    # add scalar in s2
i13:         sw     $t2, 8($s1)      # store result
i15:         bne    $s1, $0, Loop    # branch s1!=0
```

Grab redundant operation and merge them **carefully**!

Loop Unrolling – back to the example

```
i1:   Loop:  addi   $s1, $s1,-12     # decrement pointer for N=3
i2:          lw     $t0, 0($s1)      # t0=array element
i3:          lw     $t1, 4($s1)      # t1=array element
i4:          lw     $t2, 8($s1)      # t2=array element
i5:          addu   $t0, $t0, $s2    # add scalar in s2
i6:          addu   $t1, $t1, $s2    # add scalar in s2
i7:          addu   $t2, $t2, $s2    # add scalar in s2
i8:          sw     $t0, 0($s1)      # store result
i9:          sw     $t1, 4($s1)      # store result
i10:         sw     $t2, 8($s1)      # store result
i11:         bne    $s1, $0, Loop    # branch s1!=0
```

Schedule the instructions and renumber them (and update comments)!

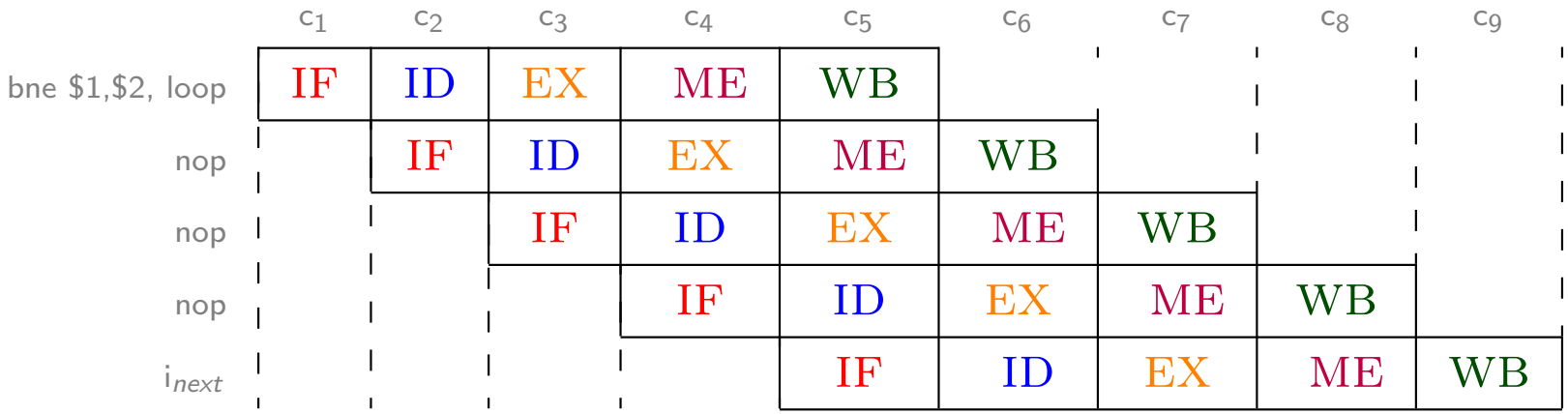
Pros & Cons

- We avoid a lot of conditional jumps (and many stall hence)
- We require 19 cycles for 11 instructions: IPC=0.57
  (a lot better than the previous 0.31)
- This trick allows to have more independent instructions to insert, and thus, less stalls!
- But we have now a prologue and an epilogue: i.e., two more basic blocks
- Require more temporaries: register allocation will be harder!
- Try it by yourself in gcc `-funroll-loops`

A very last word on Branch Hazards 1/2

- Conditional jumps often introduce delays since we cannot pre-fetch instrcutions
  - Branch Outcome and Branch Target Address are ready at the end of the EX stage (3th stage)
  - Conditional branches are solved when PC is updated at the end of the ME stage (4th stage)
- Can we avoid them?

We only know i_{next} at cycle 5!



A very last word on Branch Hazards 2/2

- X delayed slot: the X instructions after a branch are systematically executed
- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches
- We need branch prediction... but nowadays, most of processors do it for us (and use `slt`...)!
- Some architectures have bypass between stages to avoid stalls

Avoid as possible floating points and jumps!

A very last word on Branch Hazards 2/2

- X delayed slot: the X instructions after a branch are systematically executed
- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches
- We need branch prediction... but nowadays, most of processors do it for us (and use `slt`...)!
- Some architectures have bypass between stages to avoid stalls

Avoid as possible floating points and jumps!

"Do you program in `mips`?" she asked. "`nop`", he said.

Stalls due to caches

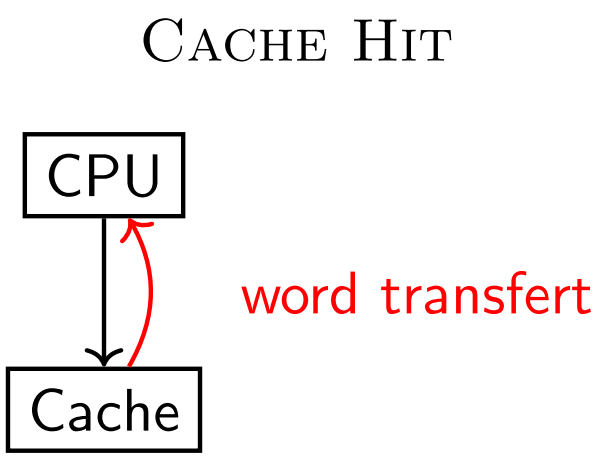
When the processor processor needs to access a data:

- If data is in cache: with a cost of 3 cycles
- Otherwise: with a cost of 100 cycles

Stalls due to caches

When the processor processor needs to access a data:

- If data is in cache: with a cost of 3 cycles
- Otherwise: with a cost of 100 cycles

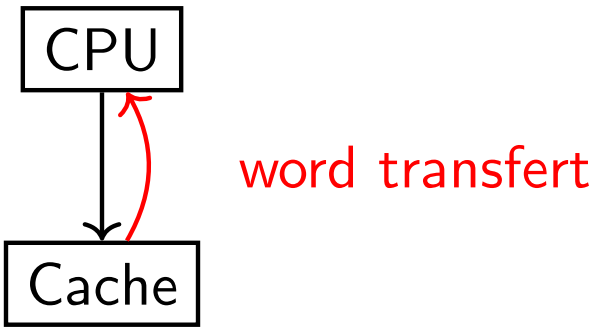


Stalls due to caches

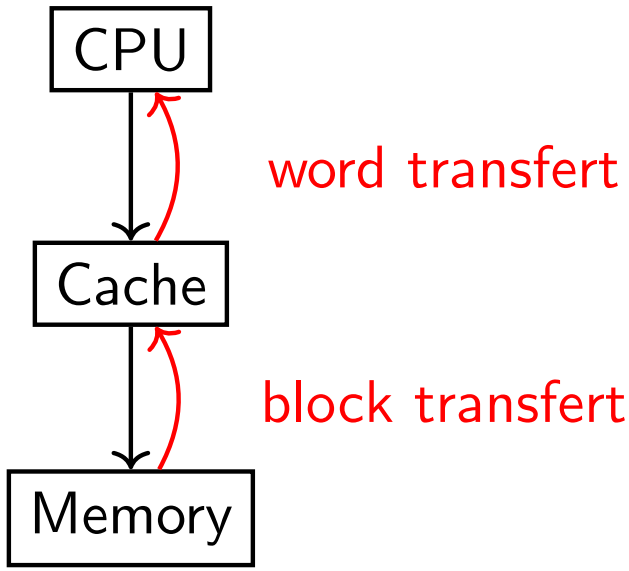
When the processor processor needs to access a data:

- If data is in cache: with a cost of 3 cycles
- Otherwise: with a cost of 100 cycles

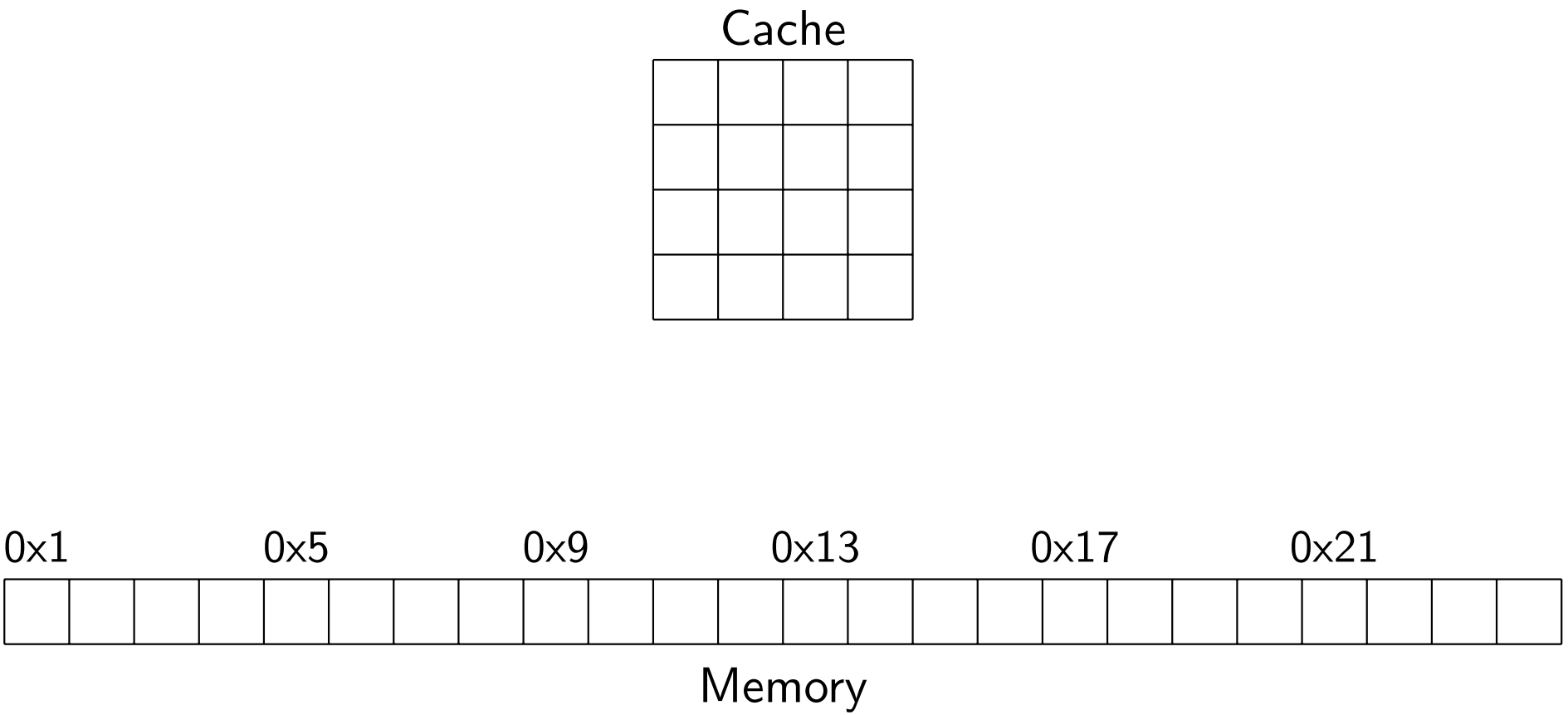
Cache Hit



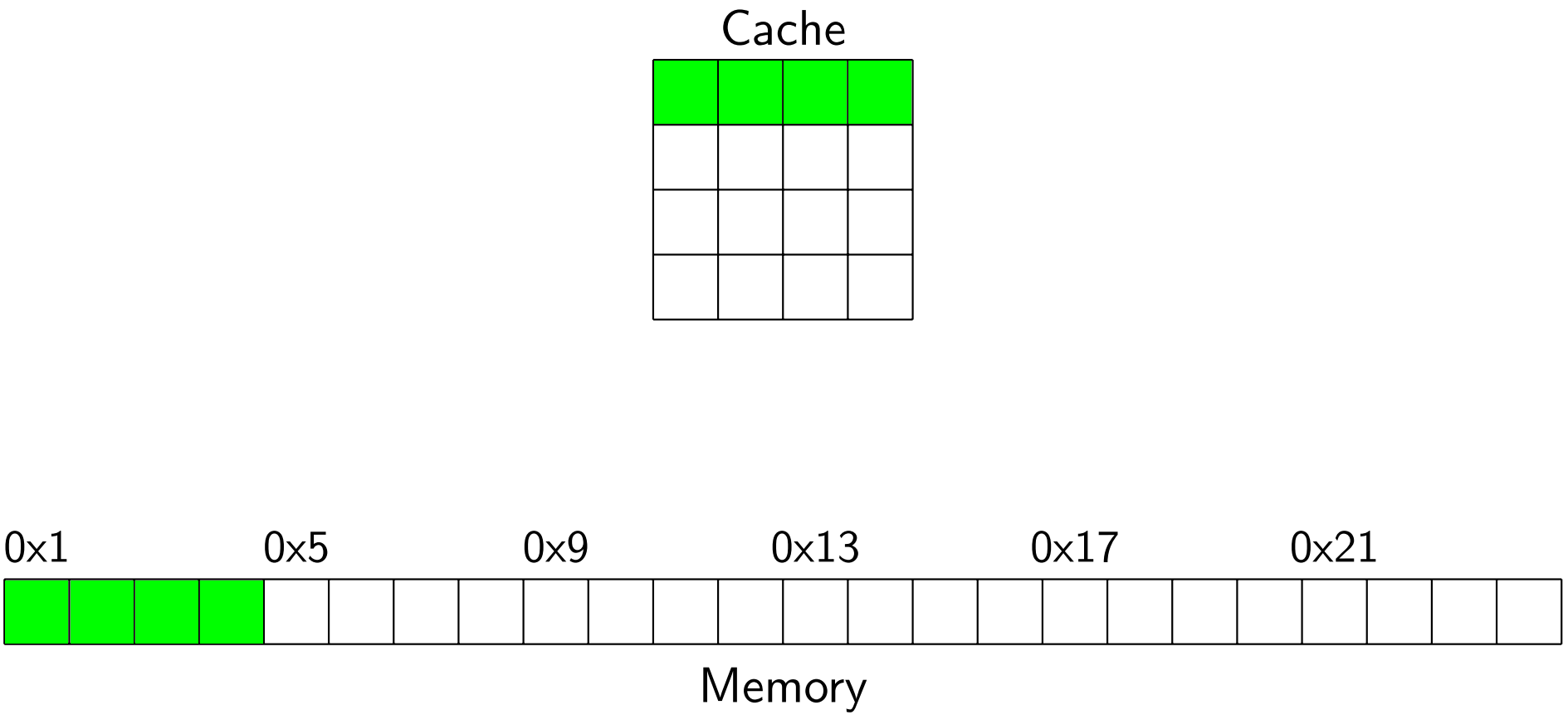
Cache Miss



Cache Fundamentals 1/2

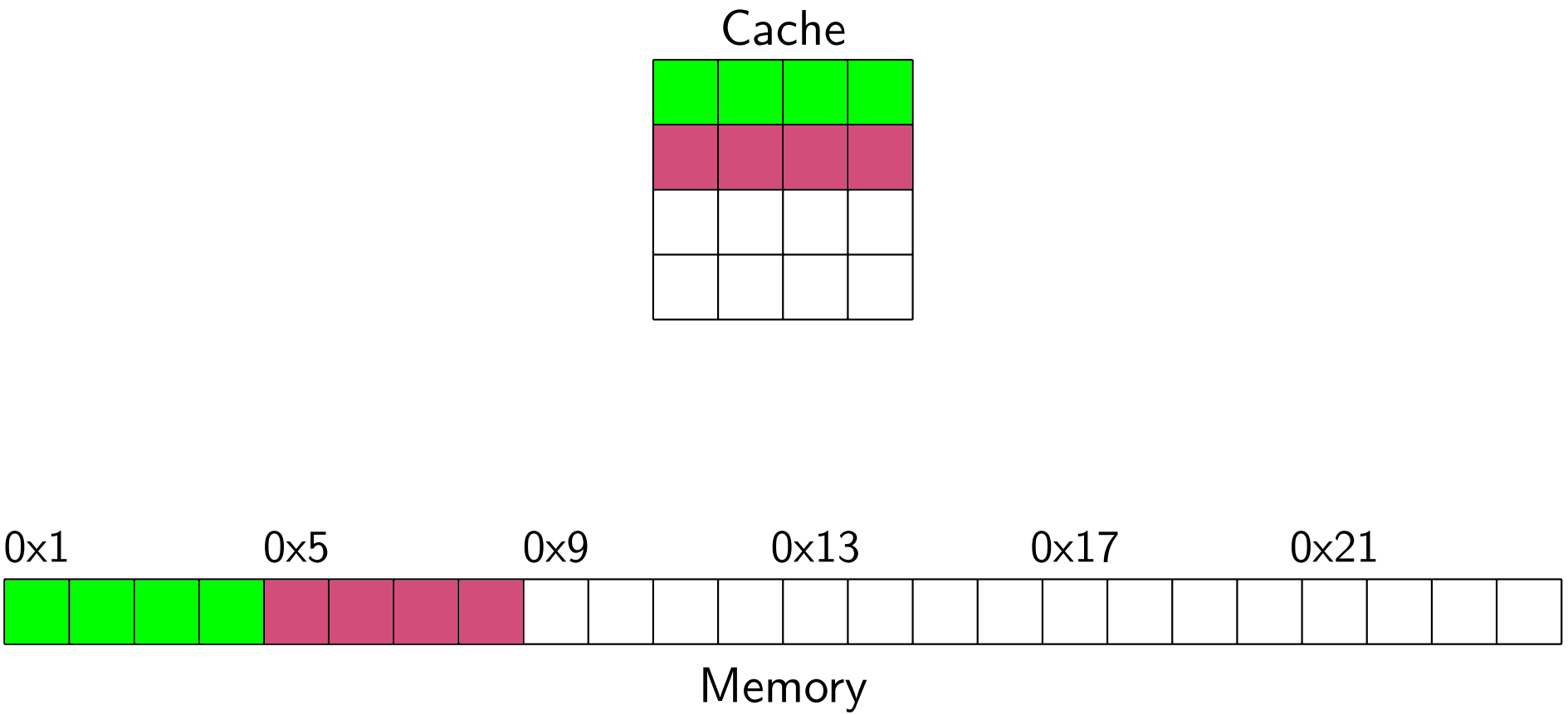


Cache Fundamentals 1/2



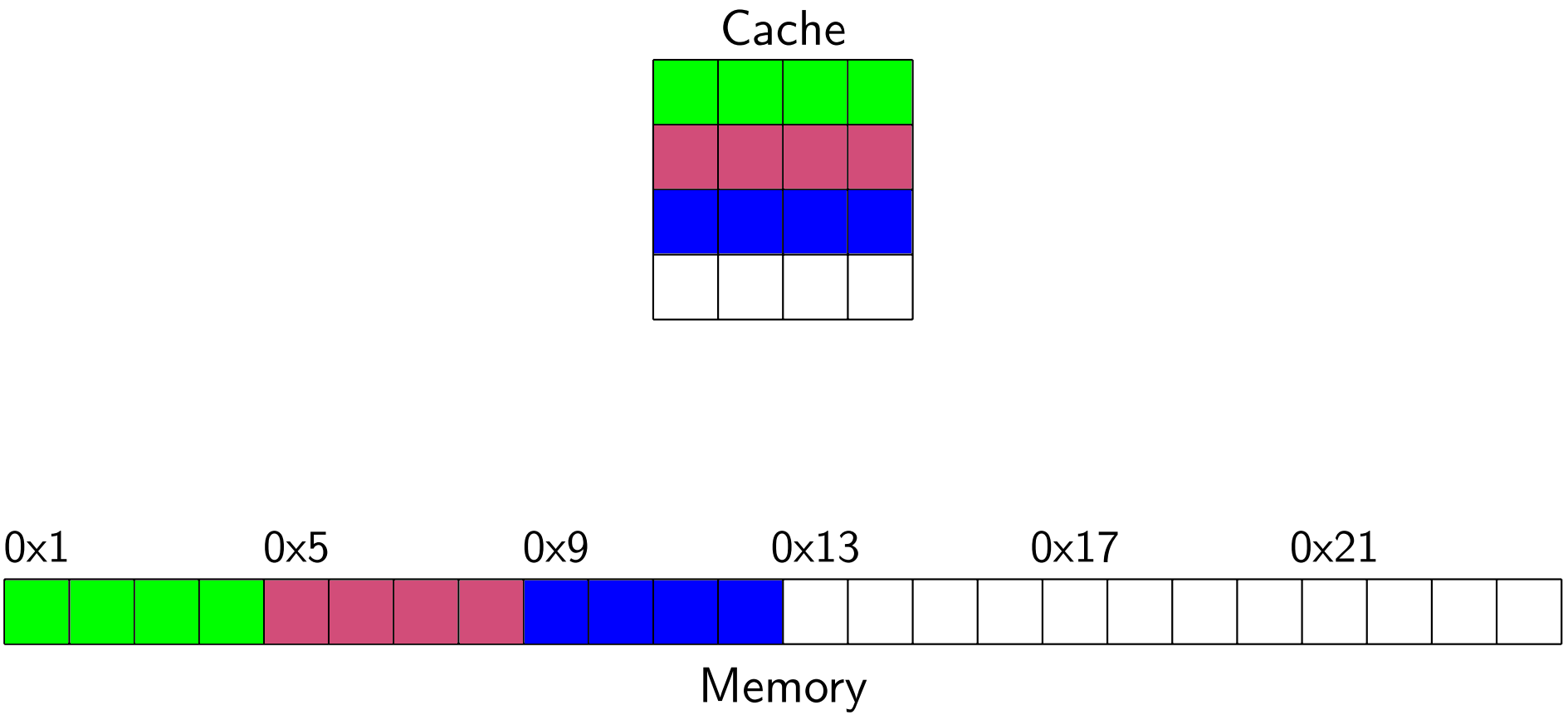
Access to adress 0x1, 4 words are fetched

Cache Fundamentals 1/2



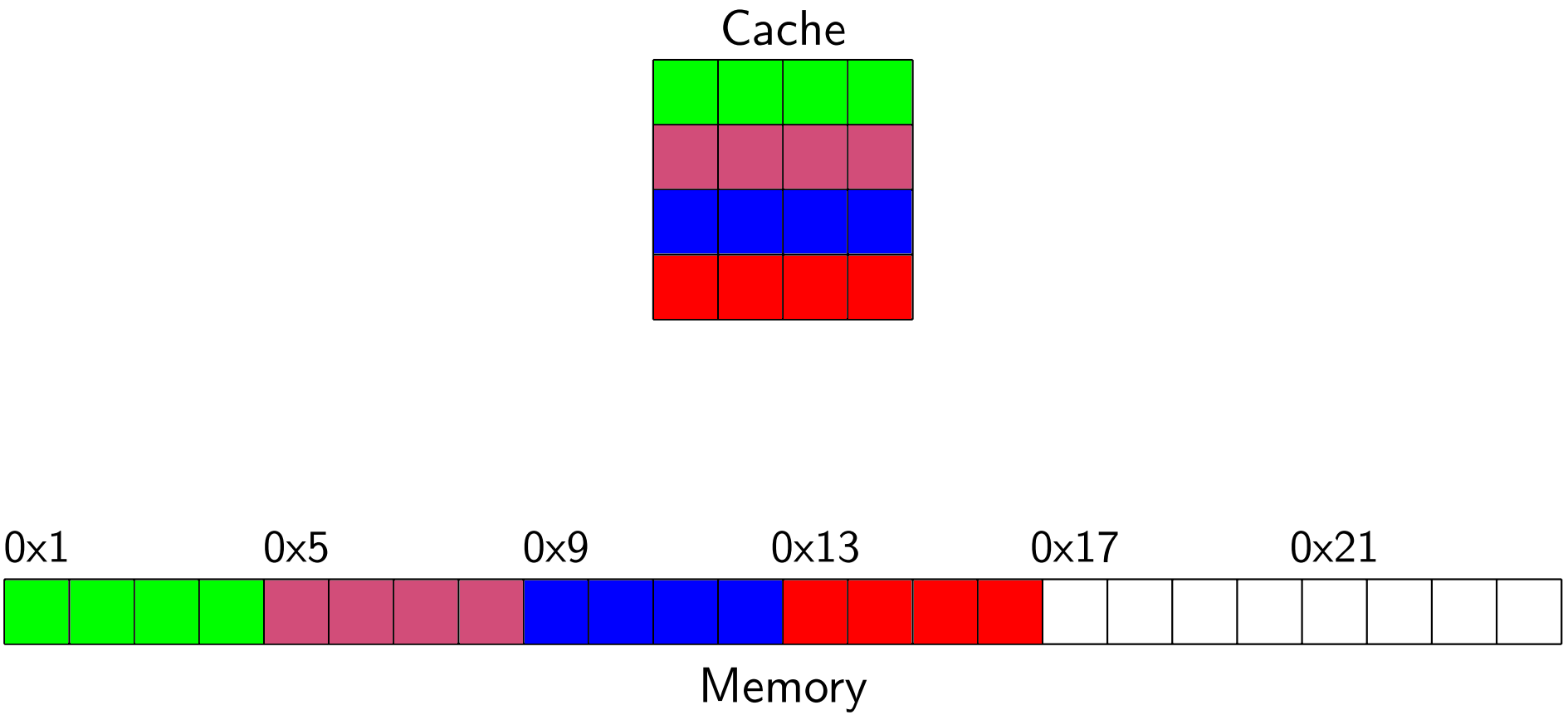
Access to adress 0x5, 4 words are fetched

Cache Fundamentals 1/2



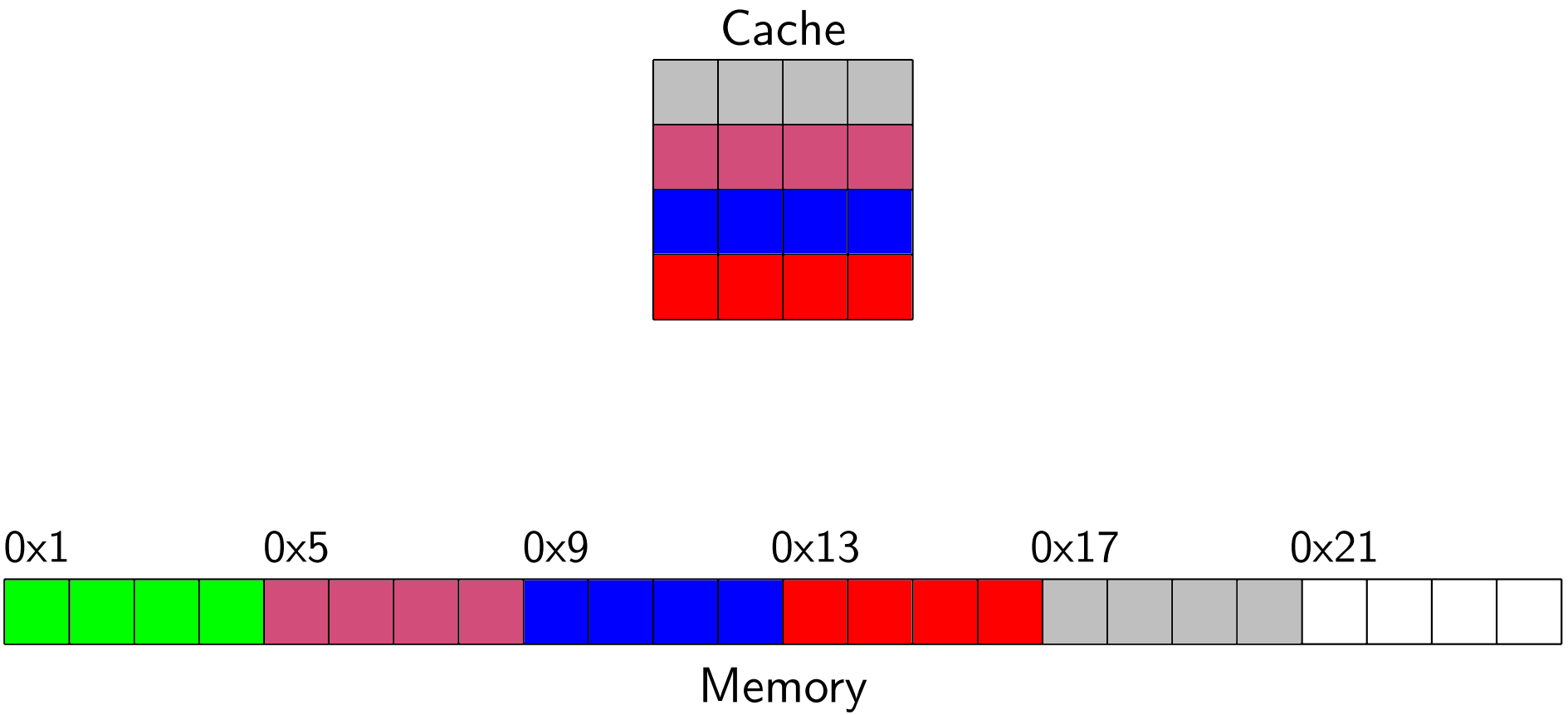
Access to adress 0x9, 4 words are fetched

Cache Fundamentals 1/2



Access to adress 0x13, 4 words are fetched

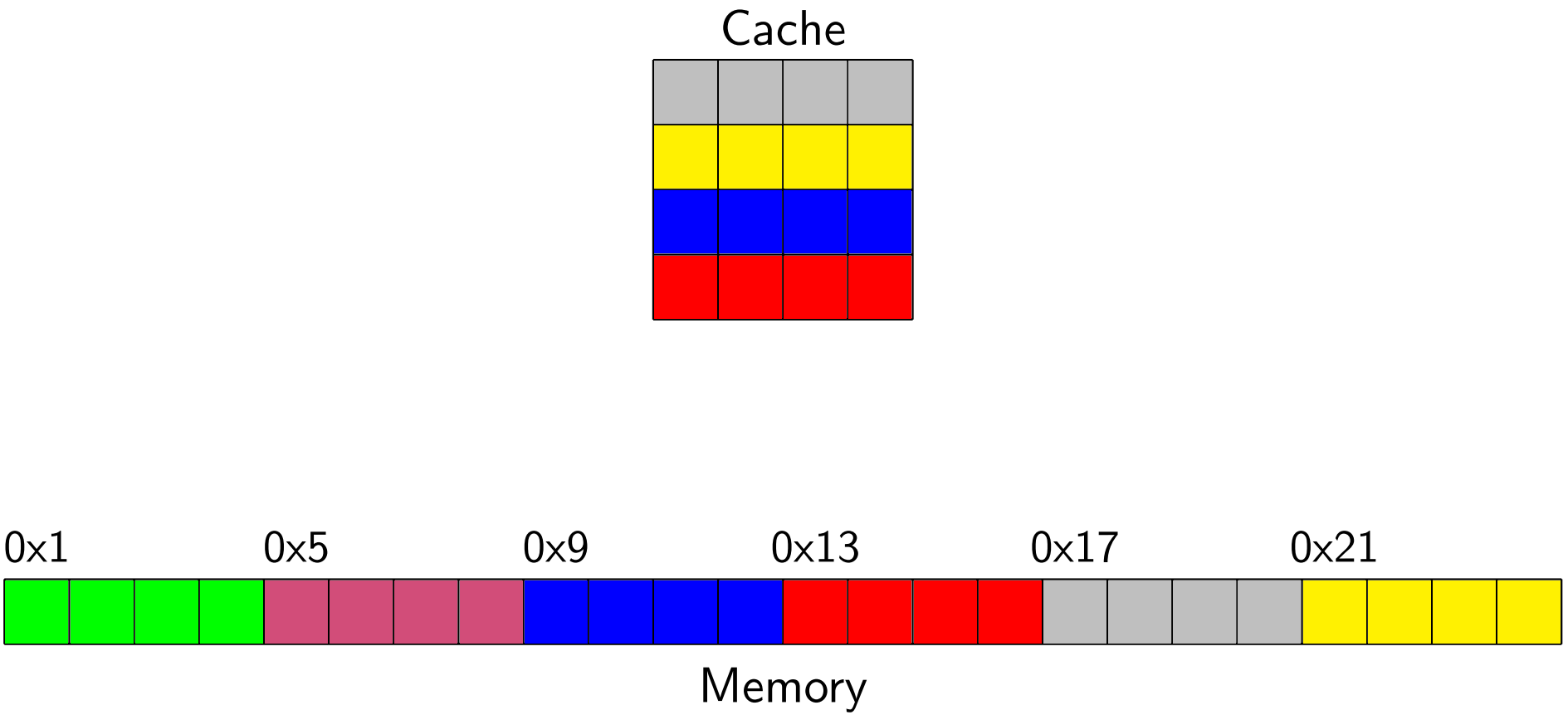
Cache Fundamentals 1/2



Access to adress 0x17, 4 words are fetched

First line of cache is replaced!

Cache Fundamentals 1/2



Access to adress 0x21, 4 words are fetched

Second line of cache is replaced!

Cache Fundamentals 1/2

Many strategies to put data into the cache:

- Direct Mapping:
  - The address is decomposed in 3 parts: `tag` (8b), `line` (22b), and `word`(2b)
  - Each block of main memory maps to only one cache line, i.e. block-size = cache-line-size
  - Simple, Inexpensive, and fixed location for given block
- Associative Mapping:
  - A main memory block can load into any line of cache
  - Memory address is interpreted as tag and word
  - Tag uniquely identifies block of memory
  - Each block of main memory maps to only one cache line, i.e. block-size = cache-line-size
  - Complex, Expensive, and no-fixed location for given block

Prefetching

Fetch the data before it is needed (i.e. pre-fetch) by the program

- Eliminate cache misses
- Involves predicting which address will be needed in the future (as for branch prediction)
- In contrast to branch prediction:
  - incorrect prefetched data will simply not be used
  - there is no need for state recovery

Locality

- Locality is the principle that future memory accesses are near past accesses
- Memories take advantage of two types of locality
  - Temporal locality, i.e. near in time: we will often access the same data again very soon
  - Spatial locality, i.e. near in space/distance: our next access is often very close to our last access (or recent accesses)

Locality

- Locality is the principle that future memory accesses are near past accesses
- Memories take advantage of two types of locality
  - Temporal locality, i.e. near in time: we will often access the same data again very soon
  - Spatial locality, i.e. near in space/distance: our next access is often very close to our last access (or recent accesses)

Some Instruction Set Architecture (ISA) allows to pre-fetch some data: i.e., Humans or compilers has to insert (take advantage) of these instructions

Loops optimisations

We have already seen loops-unrolling to avoid stalls inside of the processor. Other techniques exist to avoid stalls due to cache:

- Loop Fission
- Loop interchanging
- Tabular Grouping
- Loop blocking
- Loop reversal
- Loop tiling
- ...

Loop Fission 1/2

Consider the following code, and direct mapping strategy:

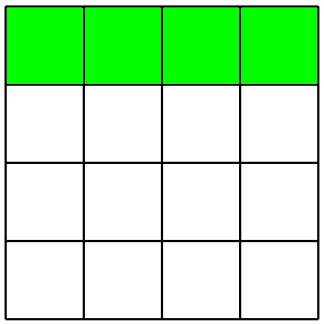
```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
   A[i] = B[i];
   C[i] = C[i-1] + 1;
}
```

Loop Fission 1/2

Consider the following code, and direct mapping strategy:

```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
   A[i] = B[i];
   C[i] = C[i-1] + 1;
}
```

Fetch A[i], A[$i+1$], A[$i+2$] and A[$i+3$]

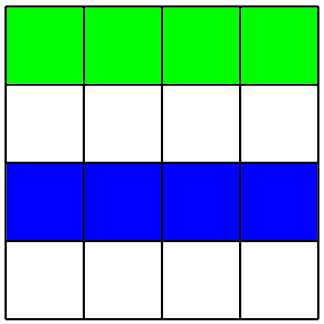


Loop Fission 1/2

Consider the following code, and direct mapping strategy:

```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
   A[i] = B[i];
   C[i] = C[i-1] + 1;
}
```

Fetch B[i], B[$i+1$], B[$i+2$] and B[$i+3$]

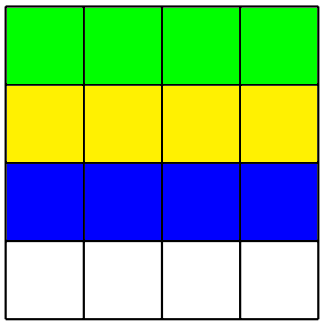


Loop Fission 1/2

Consider the following code, and direct mapping strategy:

```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
   A[i] = B[i];
   C[i] = C[i-1] + 1;
}
```

Fetch C[i], C[$i+1$], C[$i+2$] and C[$i+3$]

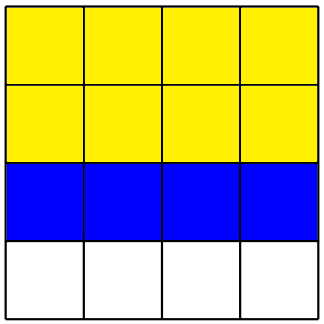


Loop Fission 1/2

Consider the following code, and direct mapping strategy:

```
int A[1024]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i) {
   A[i] = B[i];
   C[i] = C[i-1] + 1;
}
```

Fetch C[$i-1$] will probably conflict



- Hopefully A[i], B[i] and C[i] will not conflict in the cache
- but ... C[i-1] will probably!

Loop Fission 2/2

Solution

Divide the loop into two:

- Less pressure on cache
- We can now insert padding to avoid conflicts

```
int A[1024]; padding[xx]; int B[1024]; int C[1024];
for (int i = 1; i<1024; ++i)
  A[i] = B[i];
for (int i = 1; i<1024; ++i)
  C[i] = C[i-1] + 1;
```

Try it by yourself in gcc -ftree-loop-distribution

Loop interchanging 1/2

Consider the following code, and direct mapping cache:

```
int A[1024][1024];
for (int j = 1; j<1024; ++j)
   for (int i = 1; i<1024; ++i)
      A[j][i] = A[j][i] * 42;
```

In Fortran, the elements of an array are stored in memory contiguously by column, and the original loop iterates over rows, potentially creating at each access a cache miss

A	B	C
D	E	F

is stored

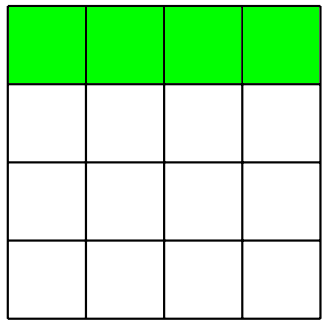
A	D	B	E	C	F

Loop interchanging 1/2

Consider the following code, and direct mapping cache:

```
int A[1024][1024];
for (int j = 1; j<1024; ++j)
   for (int i = 1; i<1024; ++i)
      A[j][i] = A[j][i] * 42;
```

Fetch $A[j][i]$, $A[j+1][i]$, $A[j+2][i]$, and $A[j+3][i]$



In Fortran, the elements of an array are stored in memory contiguously by column, and the original loop iterates over rows, potentially creating at each access a cache miss

A	B	C
D	E	F

is stored

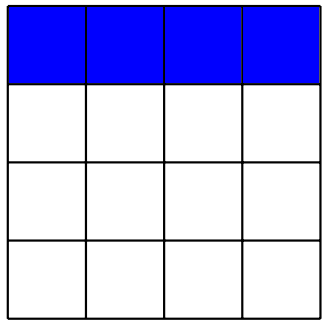
A	D	B	E	C	F

Loop interchanging 1/2

Consider the following code, and direct mapping cache:

```
int A[1024][1024];
for (int j = 1; j<1024; ++j)
   for (int i = 1; i<1024; ++i)
      A[j][i] = A[j][i] * 42;
```

Fetch $A[j+1][i]$, $A[j+2][i]$, $A[j+3][i]$, and $A[j+4][i]$



In Fortran, the elements of an array are stored in memory contiguously by column, and the original loop iterates over rows, potentially creating at each access a cache miss

A	B	C
D	E	F

is stored

A	D	B	E	C	F

Loop interchanging 2/2

Solution

This transformation switches the positions of one loop that is tightly nested within another loop.

```
int A[1024][1024];
for (int i = 1; i<1024; ++i)
   for (int j = 1; j<1024; ++j)
      A[j][i] = A[j][i] * 42;
```

Legal if the outermost loop does not carry any data dependence

Try it by yourself in gcc -floop-interchange

Tabular Grouping 1/2

Consider the following code, and direct mapping cache:

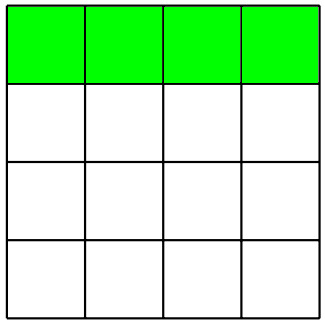
```
int A[1024]; int B[1024];
for (int j = 1; j<1024; ++j)
  A[j] = B[j] * 42;
```

Tabular Grouping 1/2

Consider the following code, and direct mapping cache:

```
int A[1024]; int B[1024];
for (int j = 1; j<1024; ++j)
  A[j] = B[j] * 42;
```

Fetch B[i], B[$i+1$], B[$i+2$] and B[$i+3$]

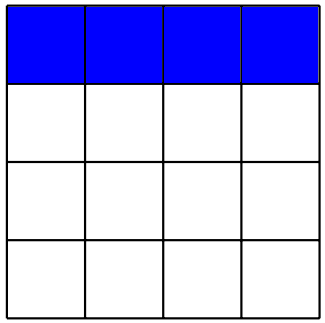


Tabular Grouping 1/2

Consider the following code, and direct mapping cache:

```
int A[1024]; int B[1024];
for (int j = 1; j<1024; ++j)
  A[j] = B[j] * 42;
```

Fetch A[i], A[$i+1$], A[$i+2$] and A[$i+3$]

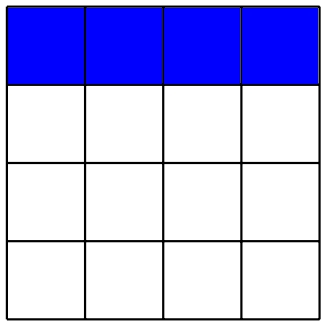


Tabular Grouping 1/2

Consider the following code, and direct mapping cache:

```
int A[1024]; int B[1024];
for (int j = 1; j<1024; ++j)
  A[j] = B[j] * 42;
```

Fetch A[i], A[$i+1$], A[$i+2$] and A[$i+3$]



Dynamic allocation does not allow padding. In the worst case, two miss per iterations

Tabular Grouping 2/2

Solution

Group the two tabular into one

```
struct twoval{int A; int B};
struct twoval R[1024];
for (int j = 1; j<1024; ++j)
  R[j].A = R[j].B * 42;
```

Avoid a lot of caches miss!

Very hard for compiler to detect such cases

Loop Blocking

Consider the code below.

```
int A[1024][1024]; int B[1024][1024];
for (int i = 1; i<1024; ++i)
   for (int j = 1; j<1024; ++j)
      A[i][j] = B[i][j];
```

- If A and B are aligned we may encounter problems.
- Similar problems occur when processing images: A[i][j] = B[i-1][j-1] + B[i-1][j] + B[i-1][j+1] + B[i][j-1] + B[i][j] + B[i][j+1] + B[i-1][j+1] + B[i+1][j] + B[i+1][j+1] ;
- In this latter case, padding is complicated...

Loop Blocking

Solution

Try to work with data that fit in memory!

```
int A[1024][1024]; int B[1024][1024];
for (int i = 1; i<1024; i += B)
  for (int j = 1; j<1024; j += B)
    for (int ii = 1; ii<min(1024, ii+B-1); ii += B)
      for (int jj = 1; jj< min(1024, ii+B-1); jj += B)
        A[i][j] = B[i][j];
```

Summary

- stalls in the processor can come from many reasons
  - from data dependencies between instructions
  - from instruction dependencies
  - from cache and memory
- modern compiler hardly try to reduce them
  - by using Instruction Level Parallelism (i.e, to have a lot of independent instructions)
  - all these optimization must occur before register allocation (which is the final step)
  - When writing a compiler, you must know the target processor by heart!
- Caches can be shared between many processors!

Garbage Collection

Akim Demaille, Etienne Renault, Roland Levillain

June 4, 2019

Table of contents

Garbage Collection 1/2

- Fisrt apparition in LISP, 1959, McCarthy
- Garbage collection is the automatic reclamation of computer storage (heap) at **runtime**
- Automatic memory management
  - New/malloc doesn't need delete/free anymore
  - Necessary for fully modular programming.
    Otherwise some modules are responsible for allocation while others are responsible for deallocation.
  - No more memory leaks
  - Avoid dangling-pointers/references.
    Reclaiming memory too soon is no more possible

Garbage Collection 2/2

- Quite expensive relative to explicit heap management
  - Slow running programs down by (very roughly) 10 percent...
  - ... But sometime cheaper or competitive
  - Fair comparison is difficult since explicit deallocation affects the structure of programs in ways that may themselves be expensive
- Possible reduction of heap fragmentation
- Functional and logic programming languages generally incorporate garbage collection because their unpredictable execution patterns
- D, Python, Caml, Effeil, Swift, C#, Go, Java, Haskell, LISP, Dylan, Prolog, etc.

What is Garbage?

- An object is called garbage at some point during execution if it will never be used again.
- What is garbage at the indicated points?

```
int main() {
  Object x, y;
  x = new Object();
  y = new Object();
  /* Point A */
  x.doSomething();
  y.doSomething();
  /* Point B */
  y = new Object();
  /* Point C */
}
```

Approximating Garbage

- In general, it is undecidable whether an object is garbage
- An object is reachable if it can still be referenced by the program.

Goals

Detect and reclaim unreachable objects

Basics of a Garbage Collector

1. Distinguishing the live objects from the garbage ones
2. Reclaiming the garbage object' storage

Basics of a Garbage Collector

1. Distinguishing the live objects from the garbage ones
2. Reclaiming the garbage object' storage

We focus on built-in garbage collectors so that:

- allocation routines performs special actions
  - reclaim memory
  - emit specific code to recognize object format
  - *etc.*
- explicit calls to the deallocator are unnecessary
  - the allocator will call it on-time
  - the objects will be automatically destroyed

Different kind of GC

- Incremental techniques:
  - allow garbage collection to proceed piecemeal while application is running
  - my provide real-time garantees
  - can be generalized into concurrent collections
- Generationnal Schemes
  - improve efficiency/locality by garbage collecting a smaller area more often
  - avoid overhead due to long time objects
  - rely on pause to collect data

Table of contents

Reference Counting

Intuition

Reference Counting

Intuition

- Maintain for each object a counter to the references to this object

Reference Counting

Intuition

- Maintain for each object a counter to the references to this object
- Each time a reference to the object is created, increase the pointed-to object's counter

Reference Counting

Intuition

- Maintain for each object a counter to the references to this object
- Each time a reference to the object is created, increase the pointed-to object's counter
- Each time an existing reference to an object is eliminated, the counter is decremented

Reference Counting

Intuition

- Maintain for each object a counter to the references to this object
- Each time a reference to the object is created, increase the pointed-to object's counter
- Each time an existing reference to an object is eliminated, the counter is decremented
- When the object counter equals zero, the memory can be reclaimed

Deallocation

Caution

When an object is destructed:

Transitive reclamation can be deferred by maintaining a list of freed objects

Deallocation

Caution

When an object is destructed:

- examines pointer fields

Transitive reclamation can be deferred by maintaining a list of freed objects

Deallocation

Caution

When an object is destructed:

- examines pointer fields
- for any references R contained by this object, decrement reference counter of R

Transitive reclamation can be deferred by maintaining a list of freed objects

Deallocation

Caution

When an object is destructed:

- examines pointer fields
- for any references R contained by this object, decrement reference counter of R
- If the reference counter of R becomes 0, reclaim memory

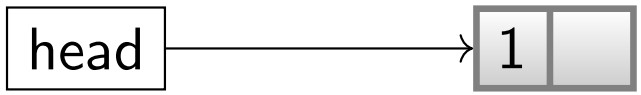
Transitive reclamation can be deferred by maintaining a list of freed objects

Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {




}
```

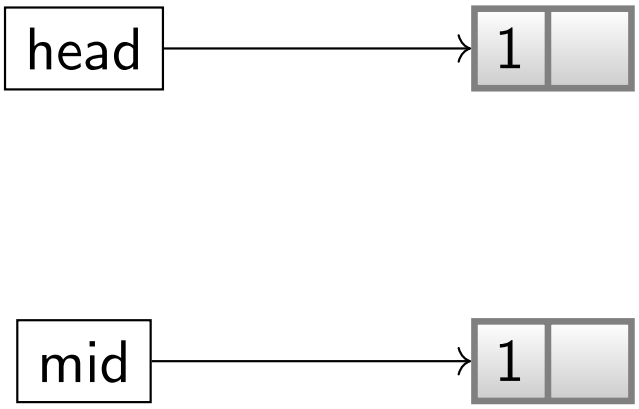
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;









}
```



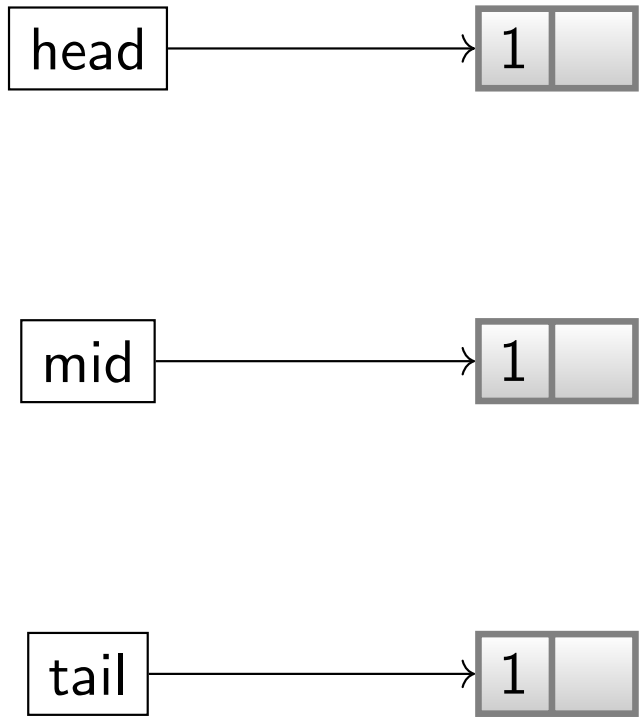
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;




}
```



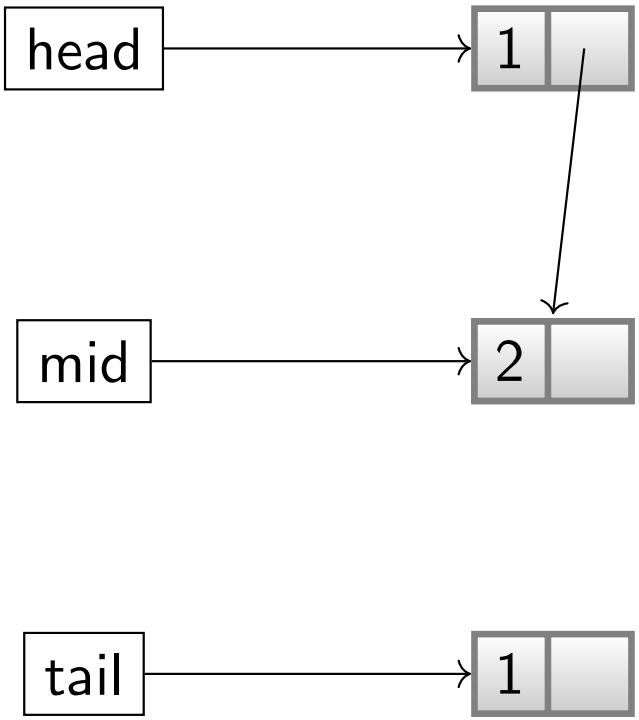
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;




}
```



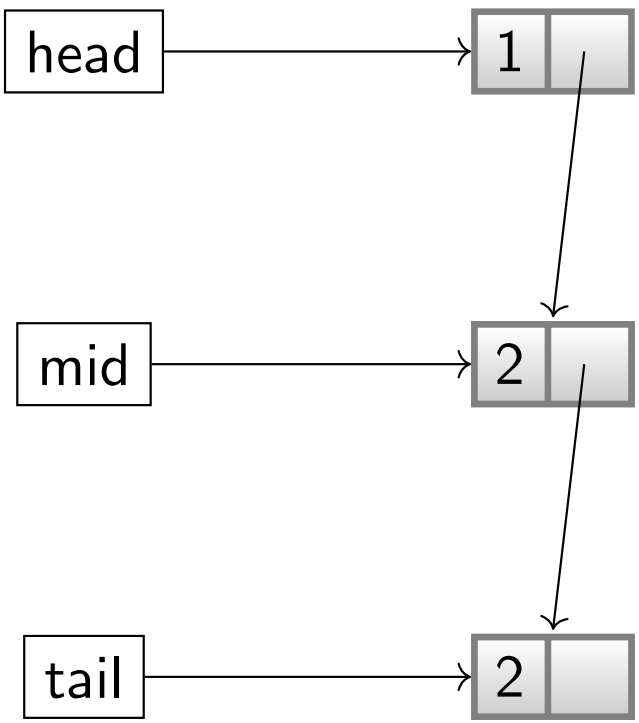
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;




}
```



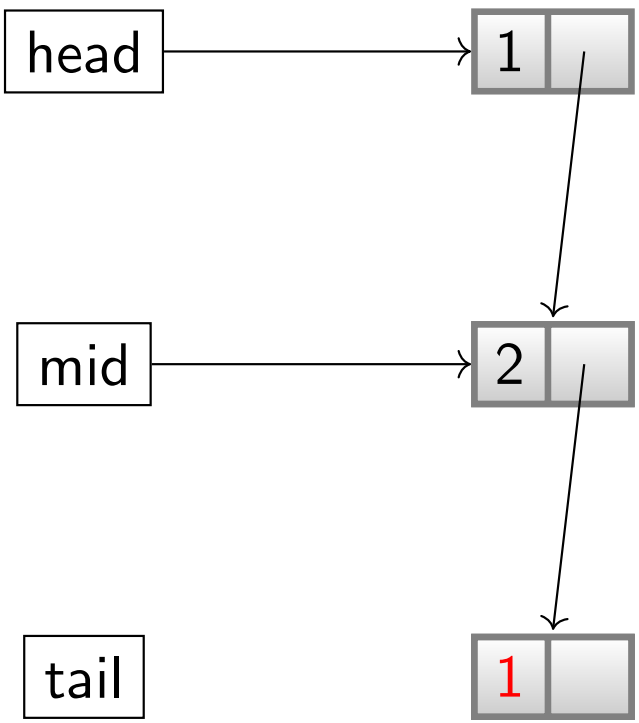
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;



}
```



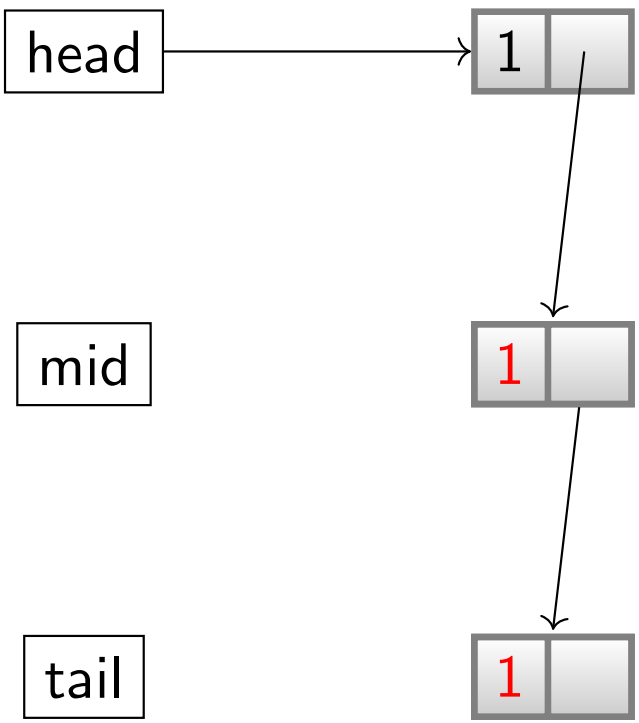
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;


}
```



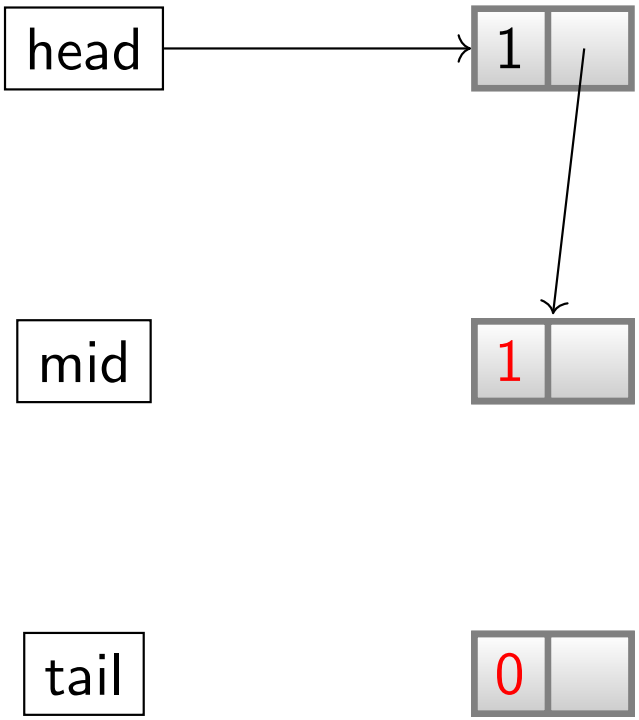
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;


}
```



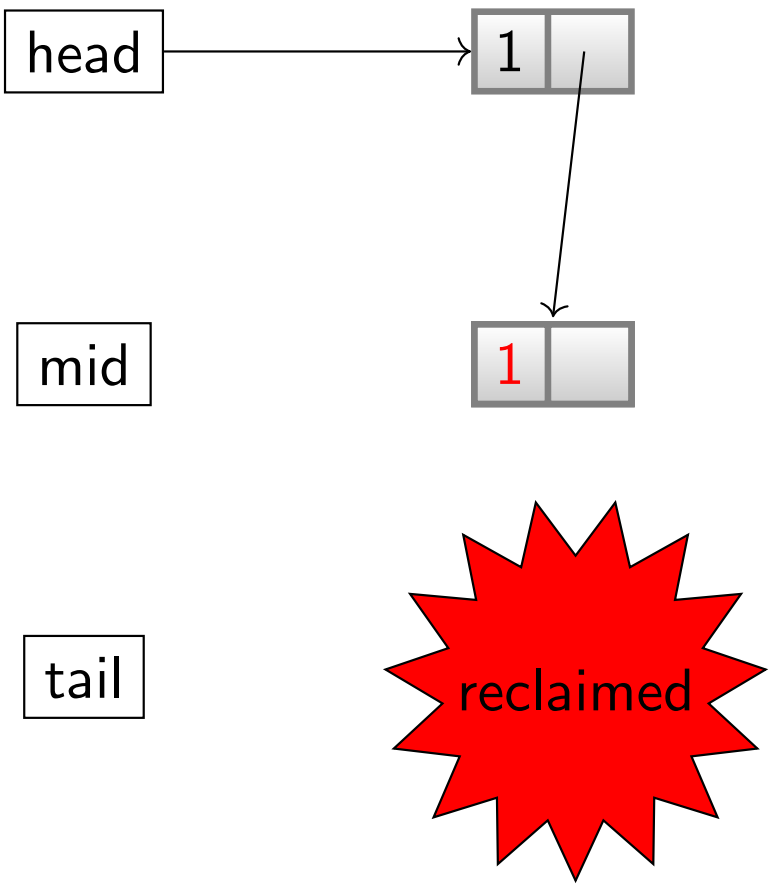
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;

}
```



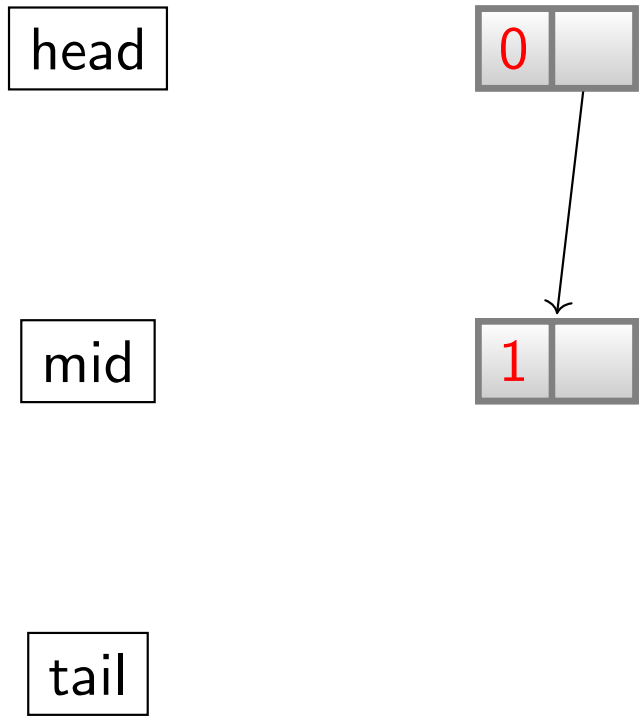
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;

}
```



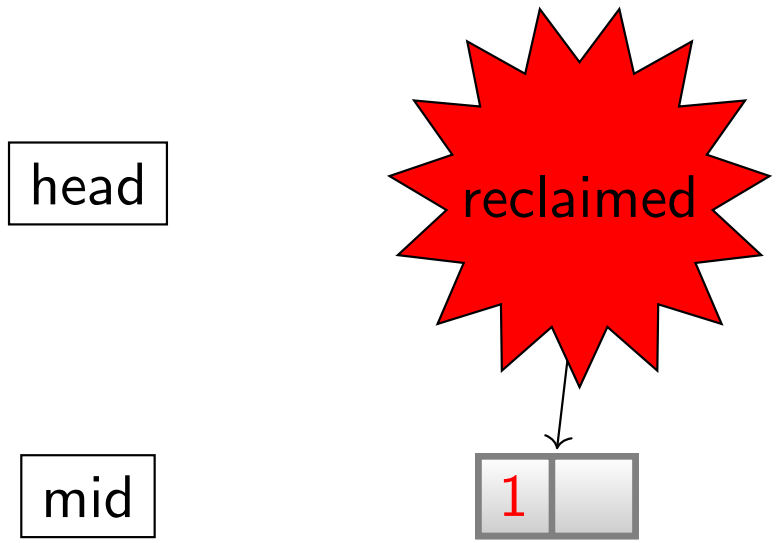
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;
 head = null;
}
```



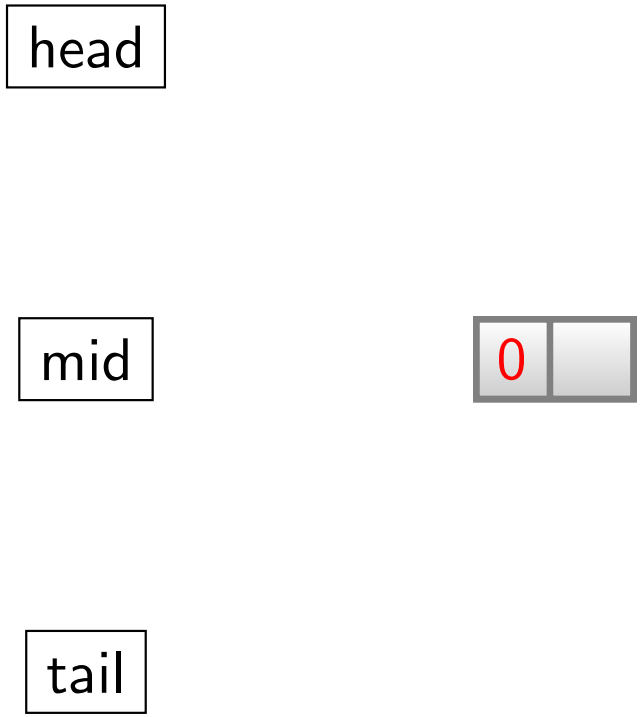
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;
 head = null;
}
```



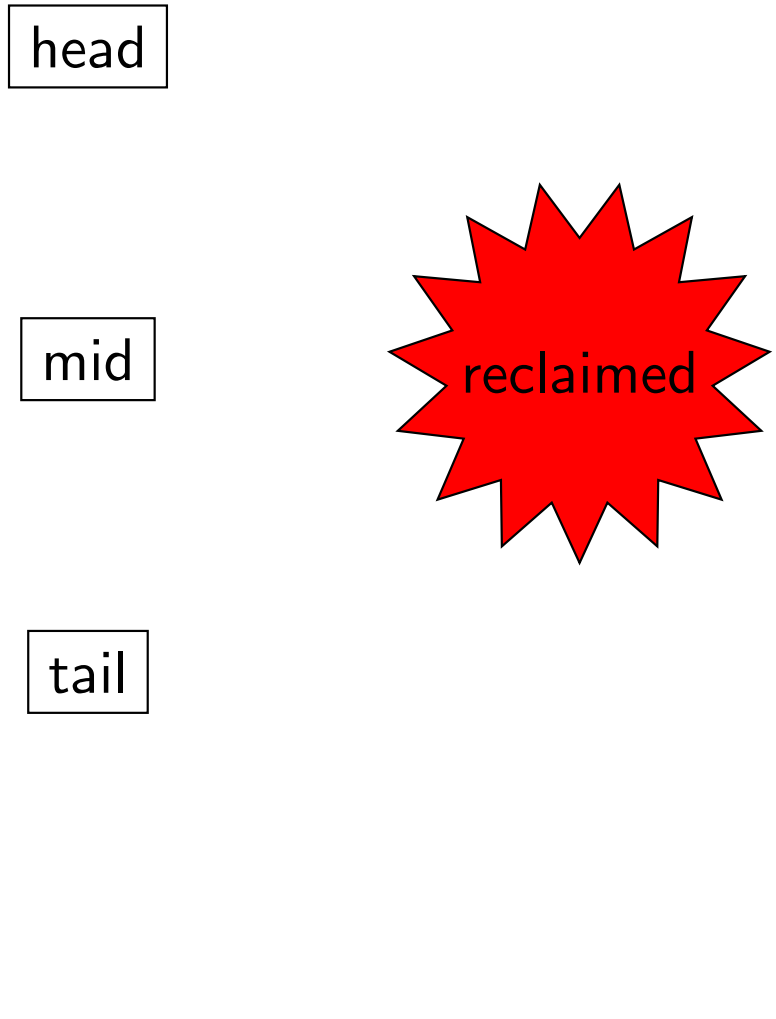
Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;
 head = null;
}
```



Exemple

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 mid = tail = null;
 head.next.next = null;
 head = null;
}
```



What about cyclic references 1

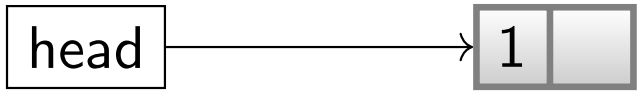
If the objects create a directed cycle, the objects references counters will never reduced to zero.

```
class LinkedList {
  LinkedList next = null;
}
int main() {









}
```

What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

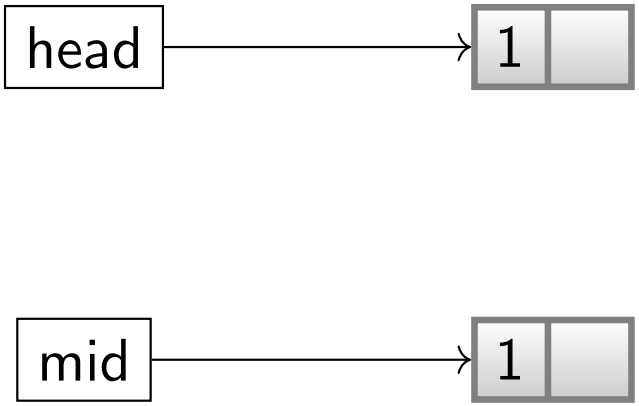
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;









}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

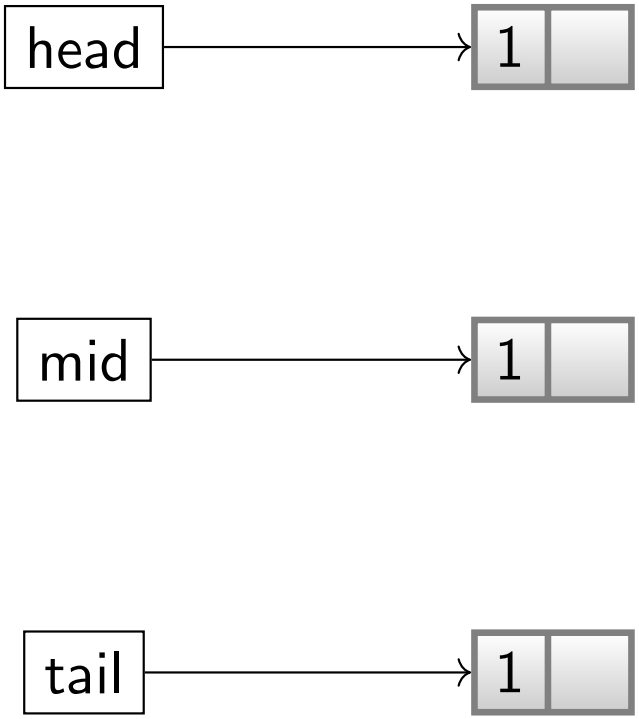
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;




}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

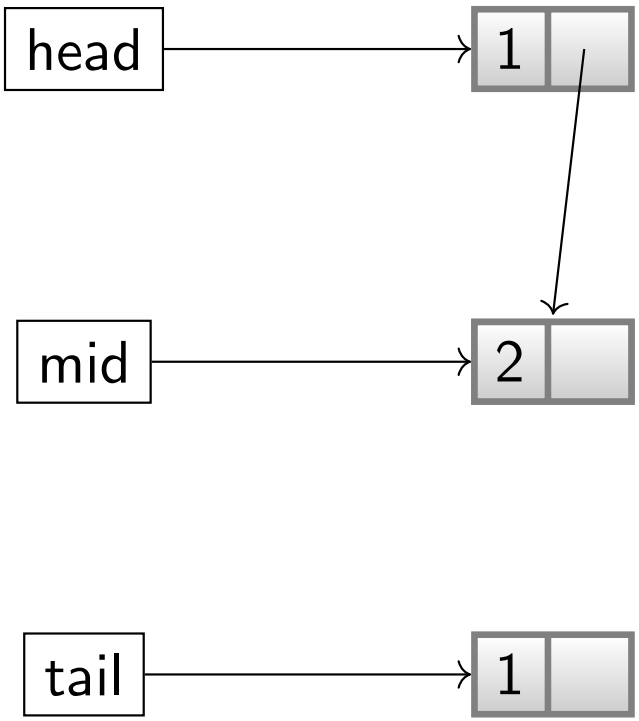
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;









}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

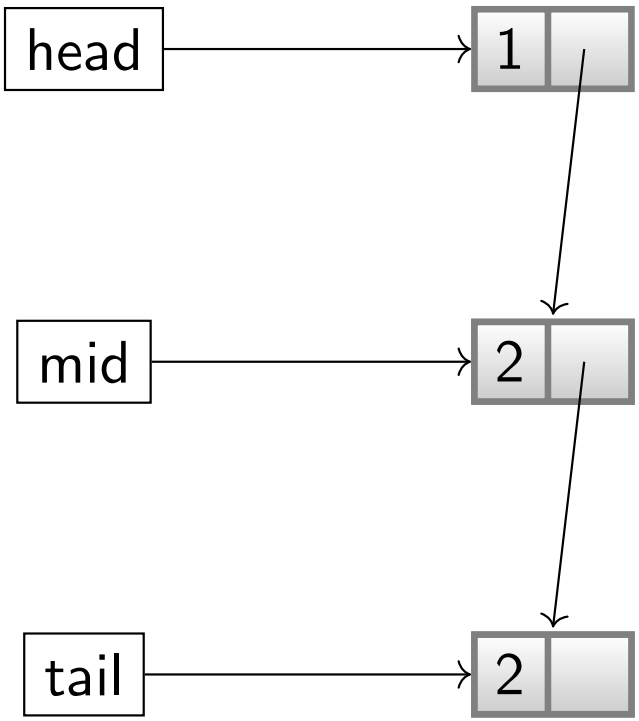
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;




}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

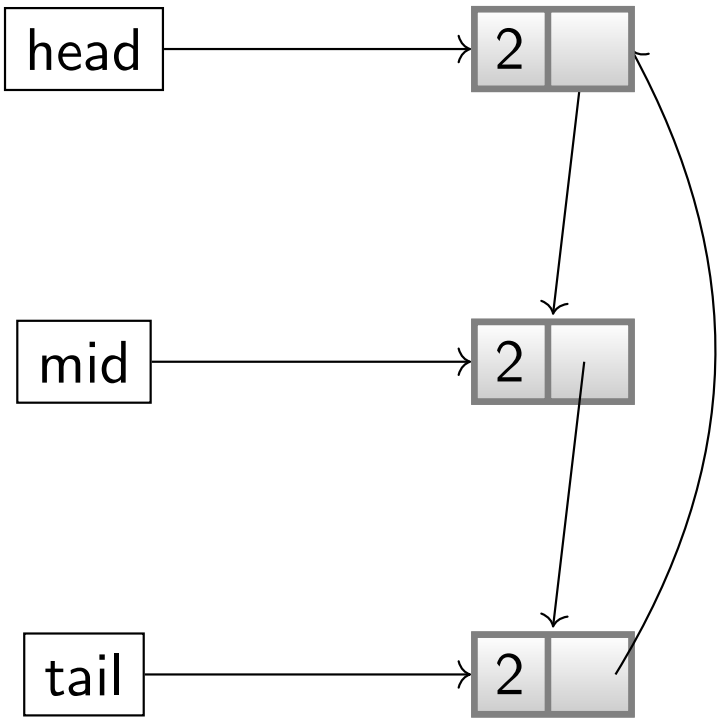
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;




}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

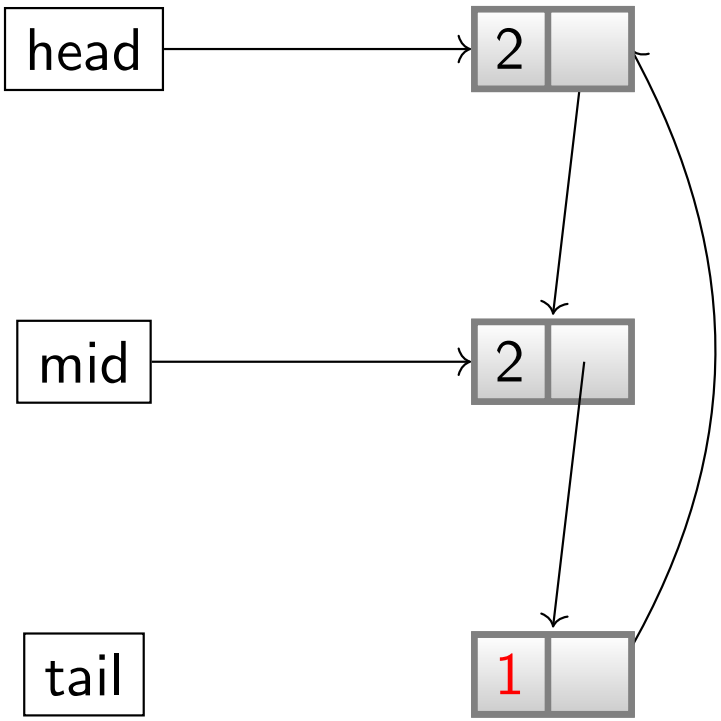
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 tail.next = head;



}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

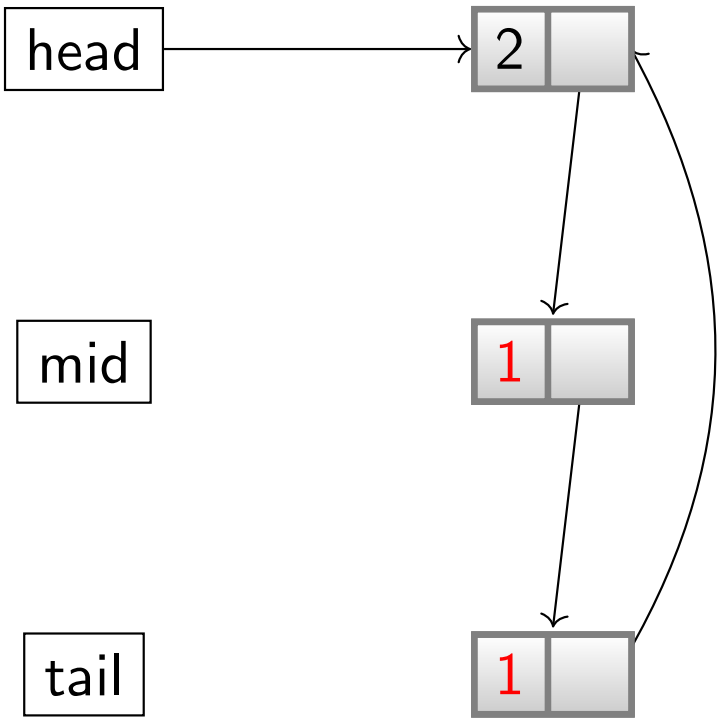
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 tail.next = head;
 tail = null;


}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

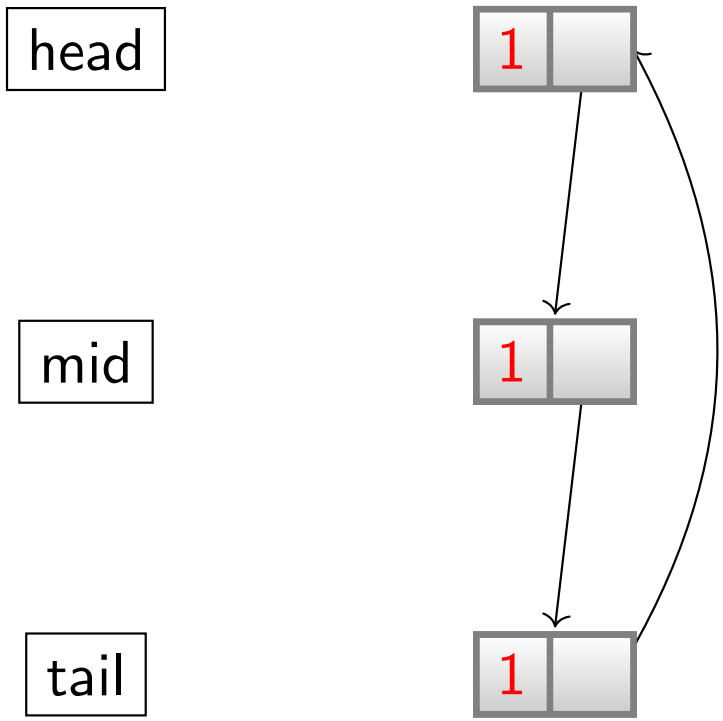
```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 tail.next = head;
 tail = null;
 mid = null;

}
```



What about cyclic references 1

If the objects create a directed cycle, the objects references counters will never reduced to zero.

```
class LinkedList {
  LinkedList next = null;
}
int main() {
 LinkedList head = new LinkedList;
 LinkedList mid = new LinkedList;
 LinkedList tail = new LinkedList;
 head.next = mid;
 mid.next = tail;
 tail.next = head;
 tail = null;
 mid = null;
 head = null;
}
```



Pros and Cons

Pros:

- Easy to implement: perl, Firefox
- Can be implemented on top of explicit memory management librairies (shared_ptr)
- Interleaved with running time
- Small overage per unit of program execution
- Transitive reclamation can be deferred by maintaining a list of freed objects
- Real-time requierements: no halt of the system.
  Necessary for application where response-time is critical

Cons:

- A whole machine word per object
- When the number of references to an object overflows, the counter is set to the maximum and the memory will never be reclaimed
- Problem with cycles
- Efficiency: cost relative to the running program

Table of contents

Analysis

- Reference counting tries to find unreachable objects by finding objects without incoming references
- These references have been forgotten !

Analysis

- Reference counting tries to find unreachable objects by finding objects without incoming references
- These references have been forgotten !

We have to trace the lifetime of objects

Intuition

Given knowledge of what's immediately accessible, find everything reachable in the program

The root set is the set of memory locations in the program that are known to be reachable

Graph Problem

Simply do a graph search starting at the root set:

- Any objects reachable from the root set are reachable
- Any objects not reachable from the root set are not reachable

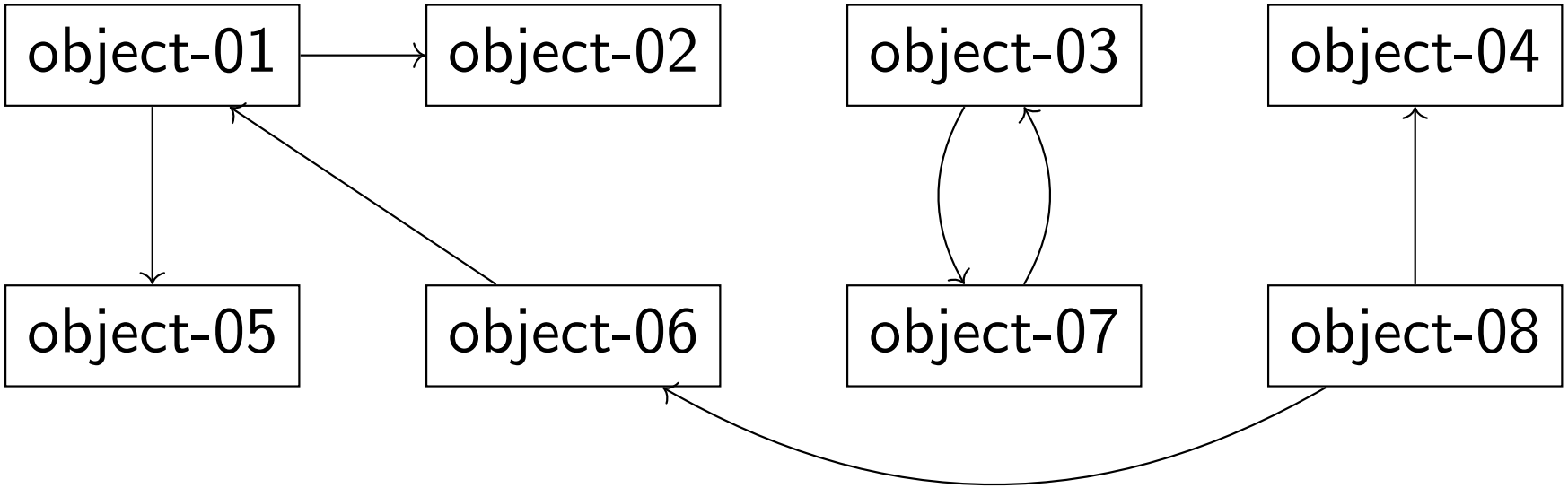
How to obtain the root set?

- static reference variables
- references registered through librairies (JNI, for instance)
- For each threads:
  - local variables
  - current method(s) arguments
  - stack
  - etc.

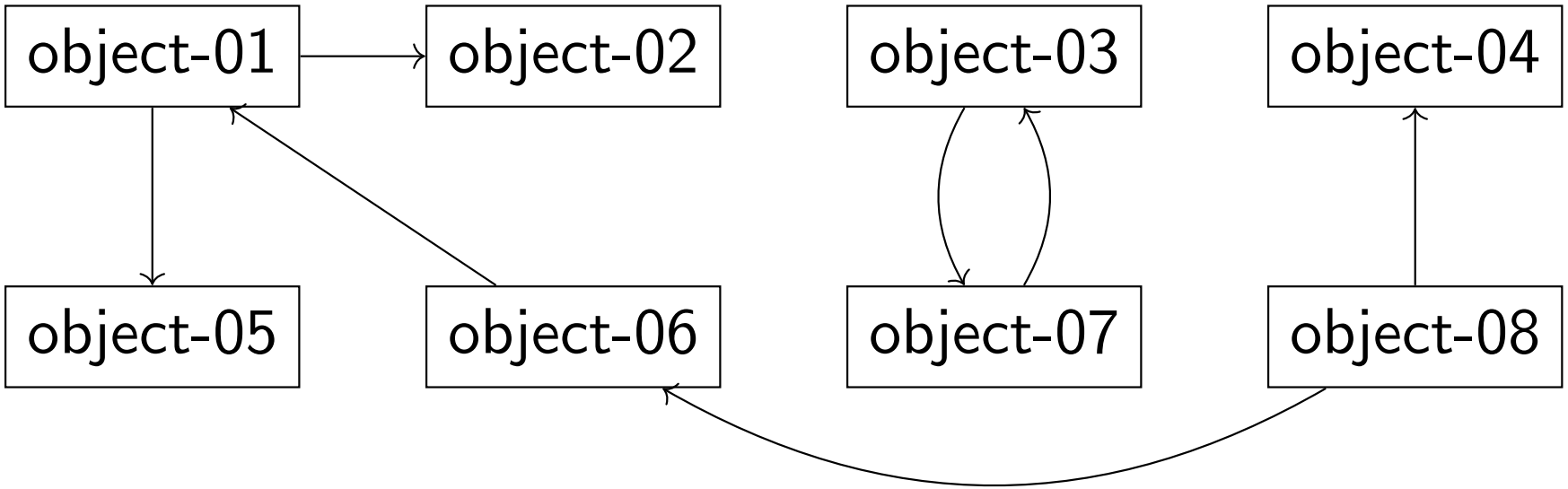
Mark-and-Sweep: the Algorithm

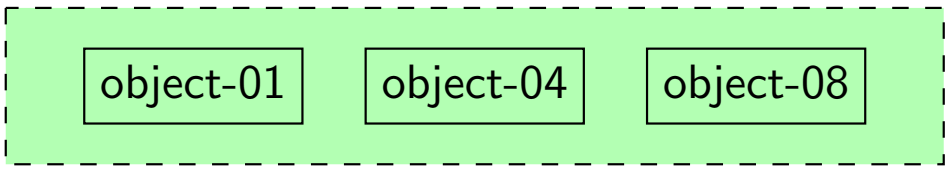
1. Marking phase: Find reachable objects
   - Add the root set to a worklist
   - While the worklist isn't empty
     - ★ Remove an object from the worklist
     - ★ If it is not marked, mark it and add to the worklist all objects reachable from that object
2. Sweeping phase: Reclaim free memory
   - If that object isn't marked, reclaim its memory
   - If the object is marked, unmark it

Example


object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08

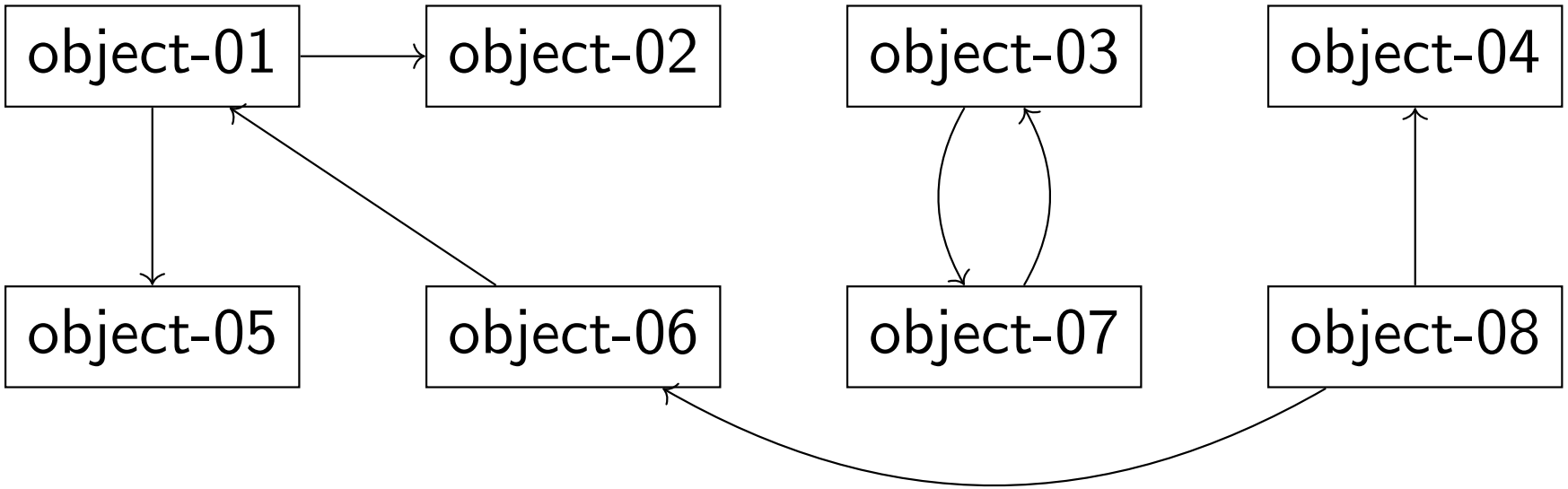
Example

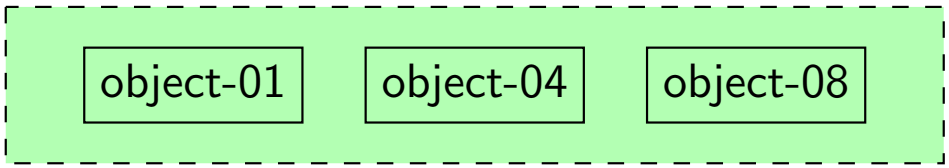




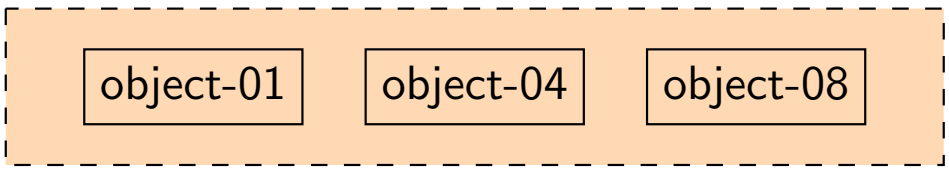
Root Set

Example



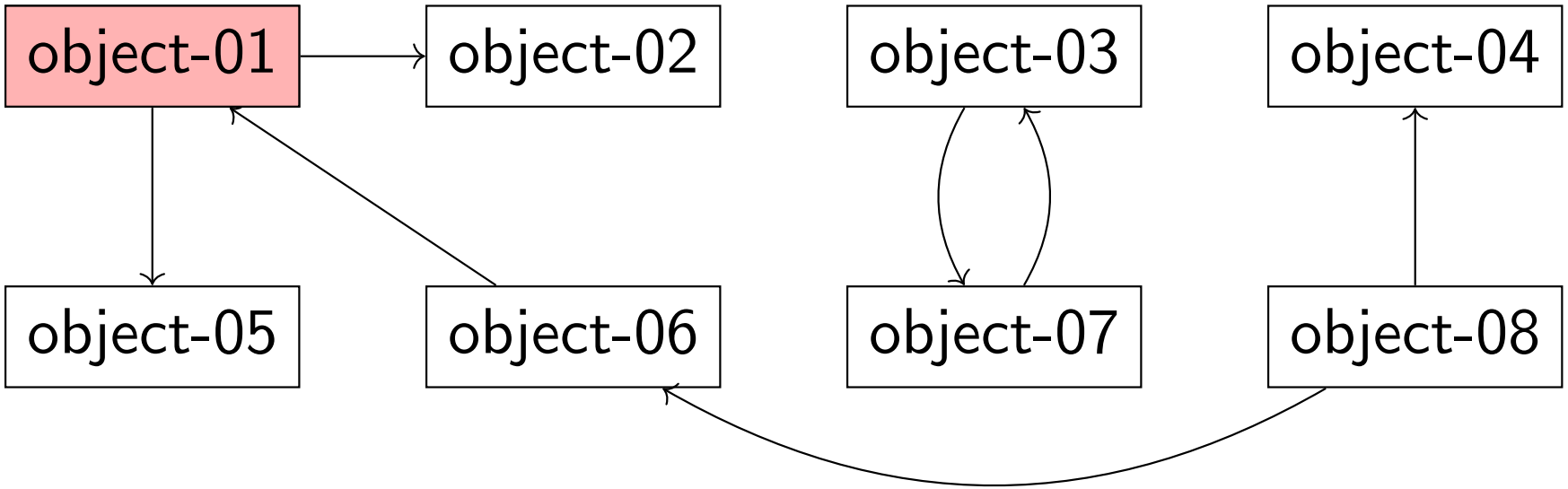


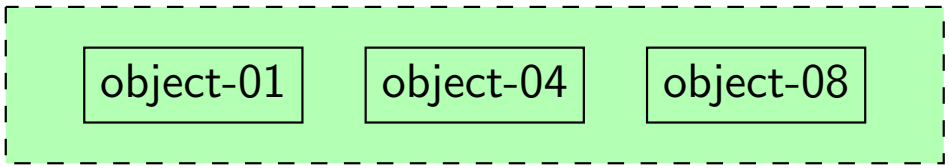
Root Set



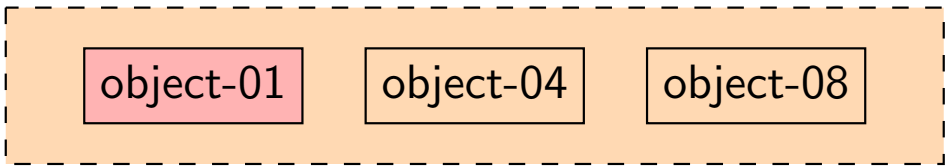
Working Set

Example



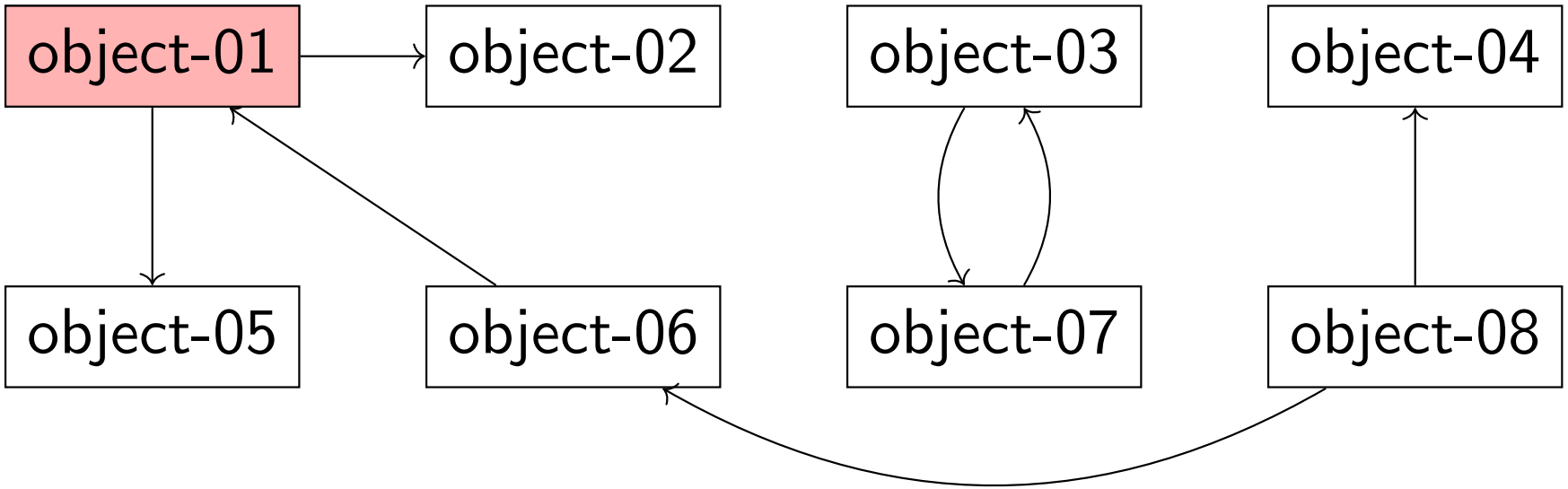


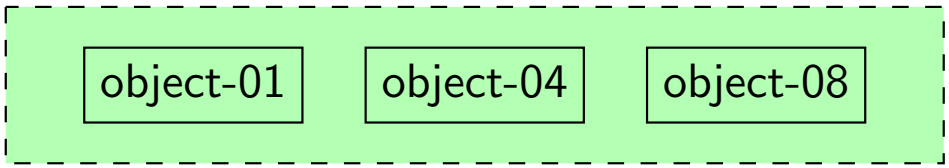
Root Set



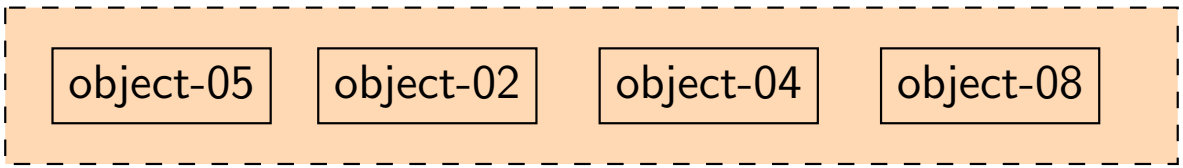
Working Set

Example



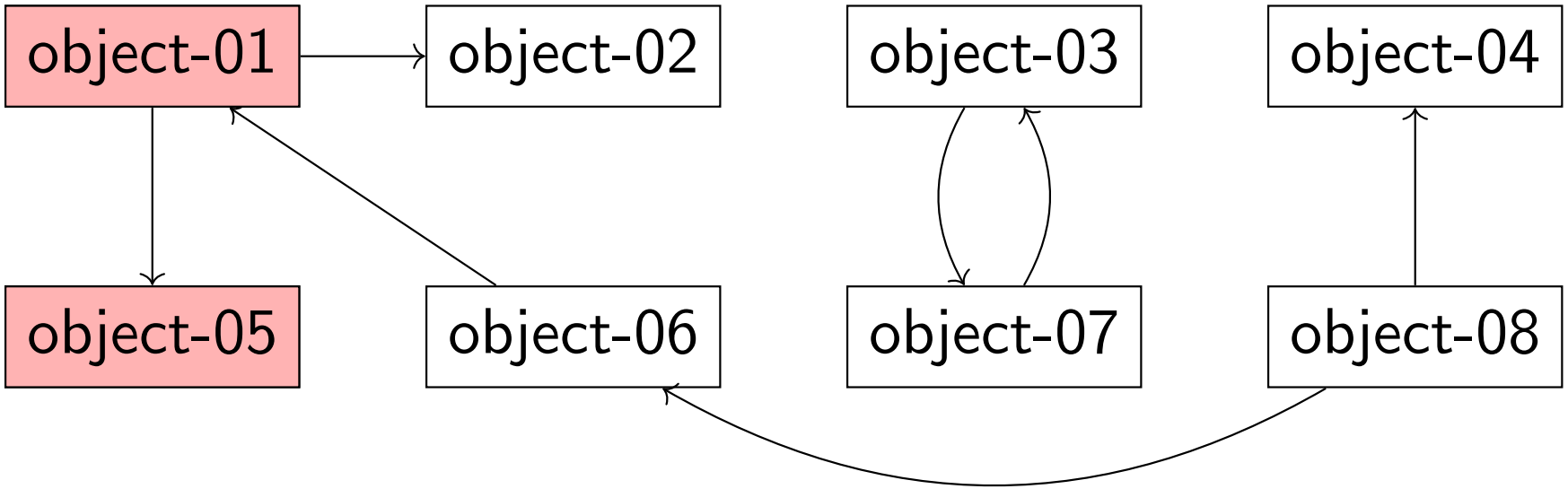


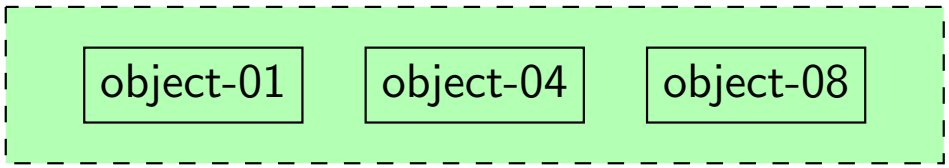
Root Set



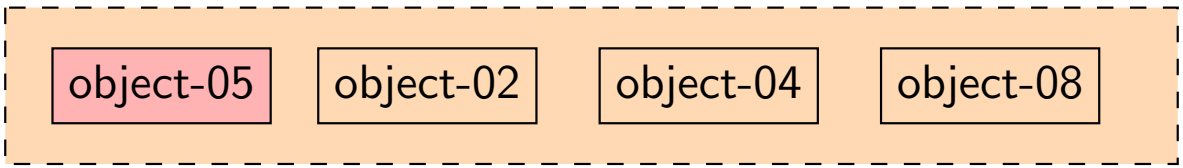
Working Set

Example



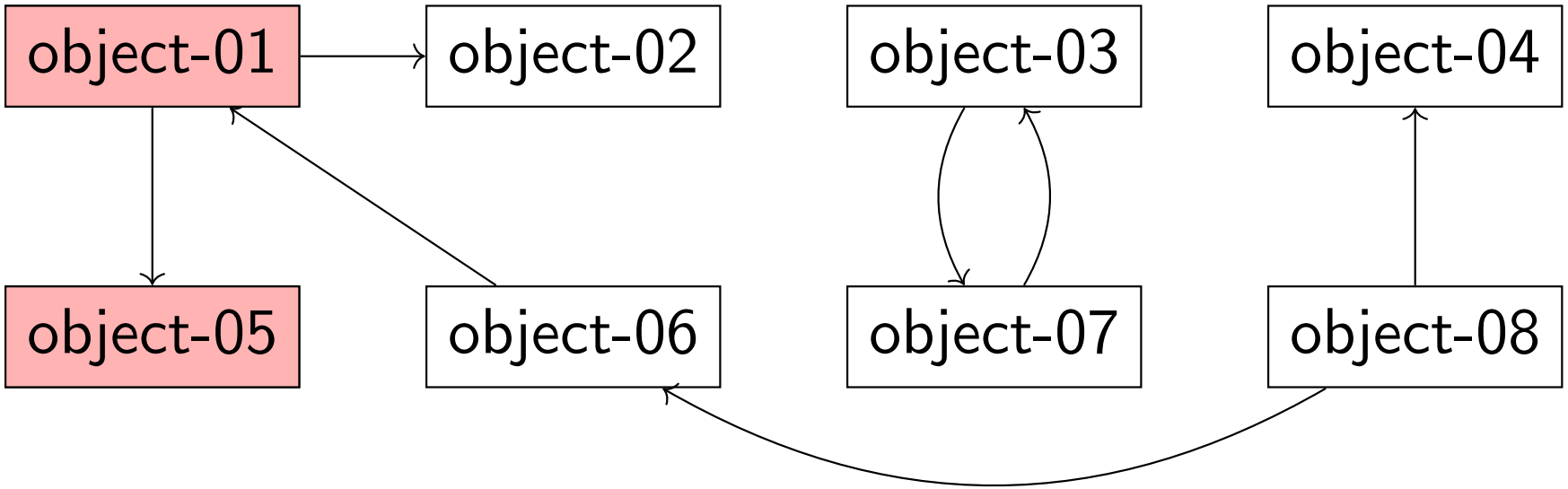


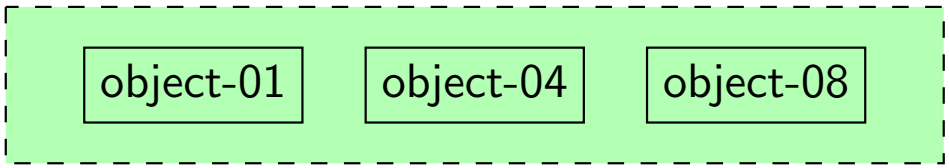
Root Set



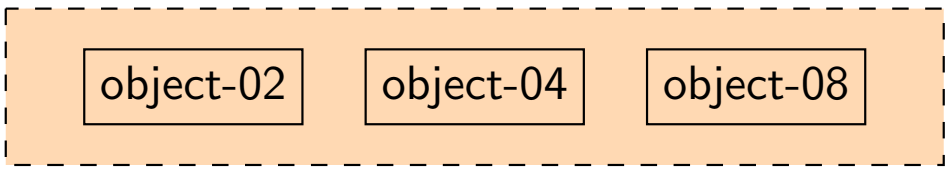
Working Set

Example



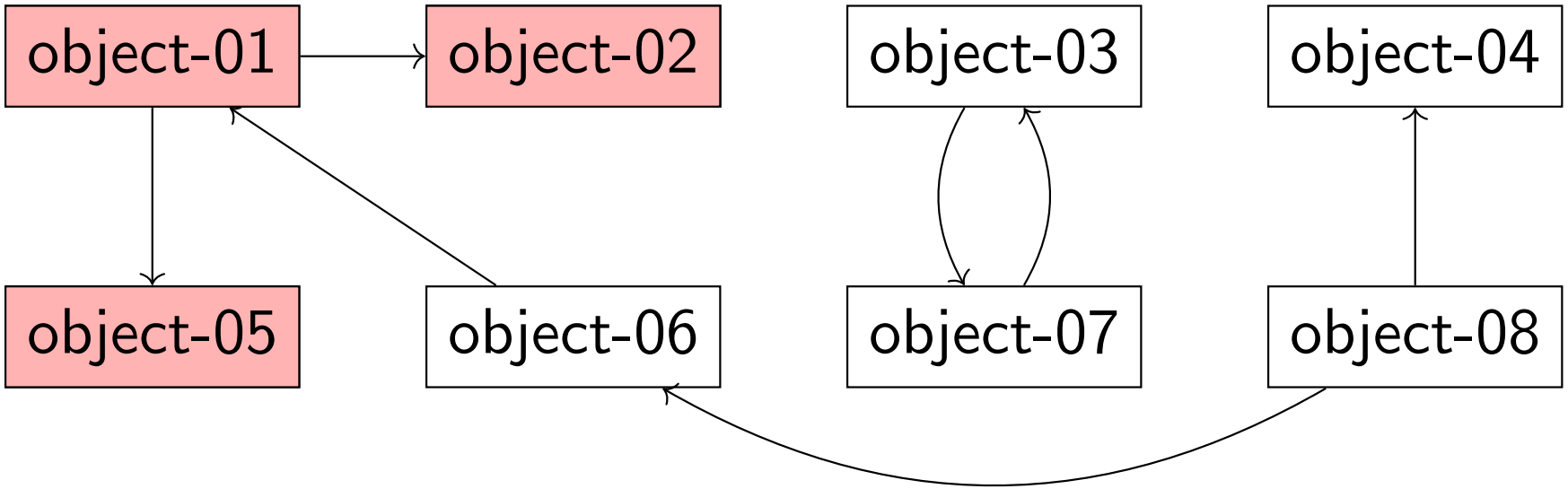


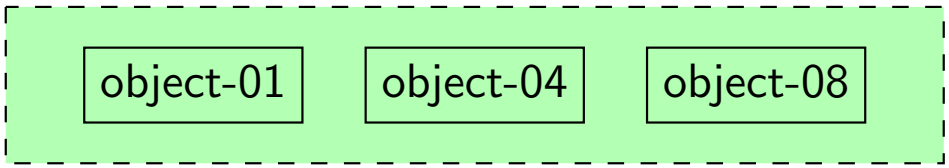
Root Set



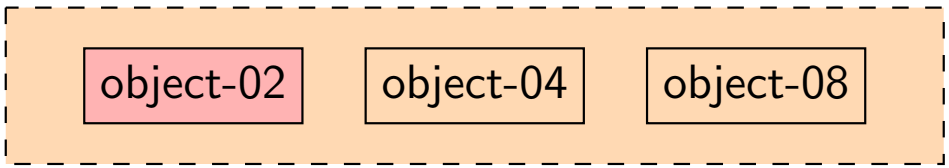
Working Set

Example



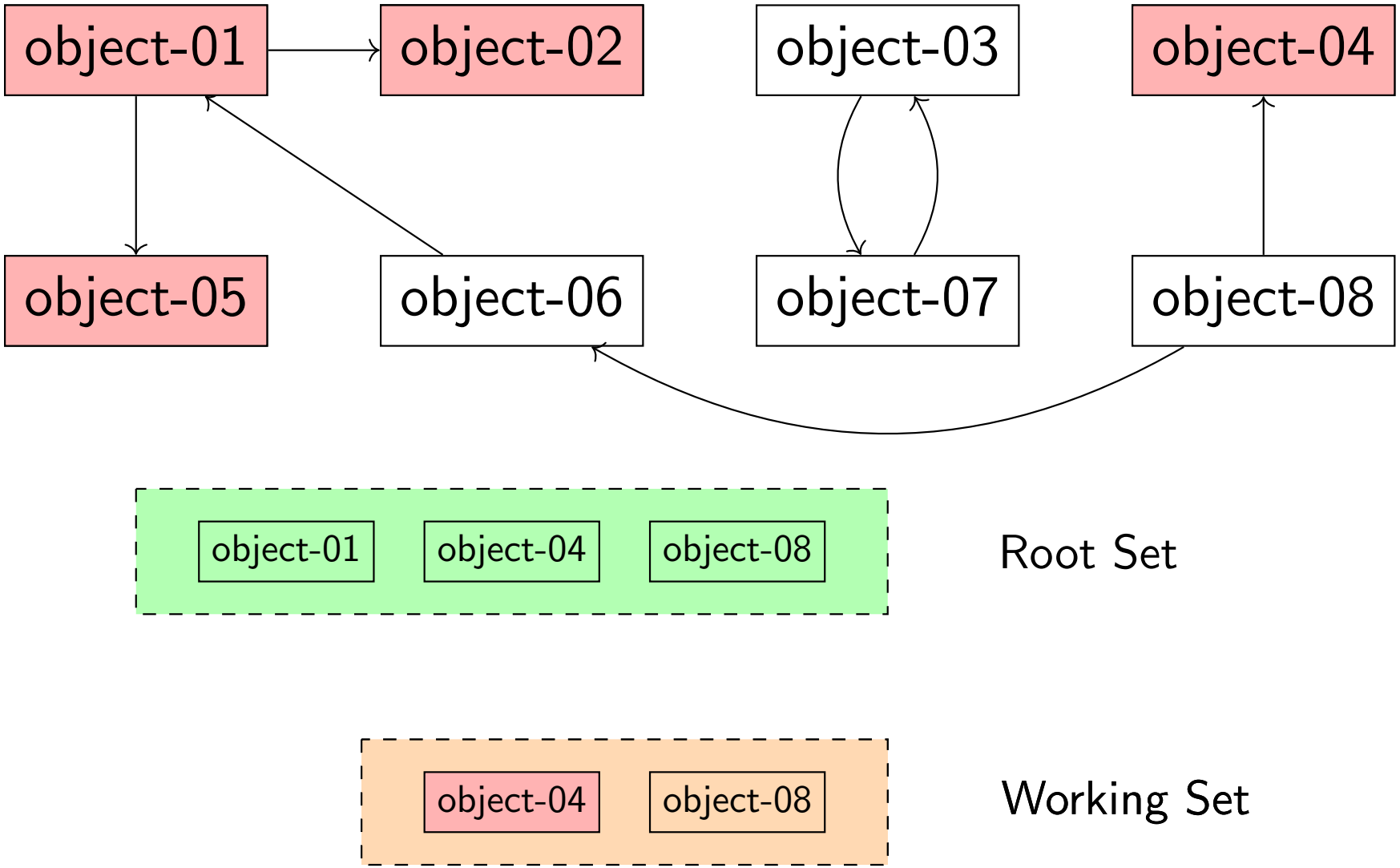


Root Set

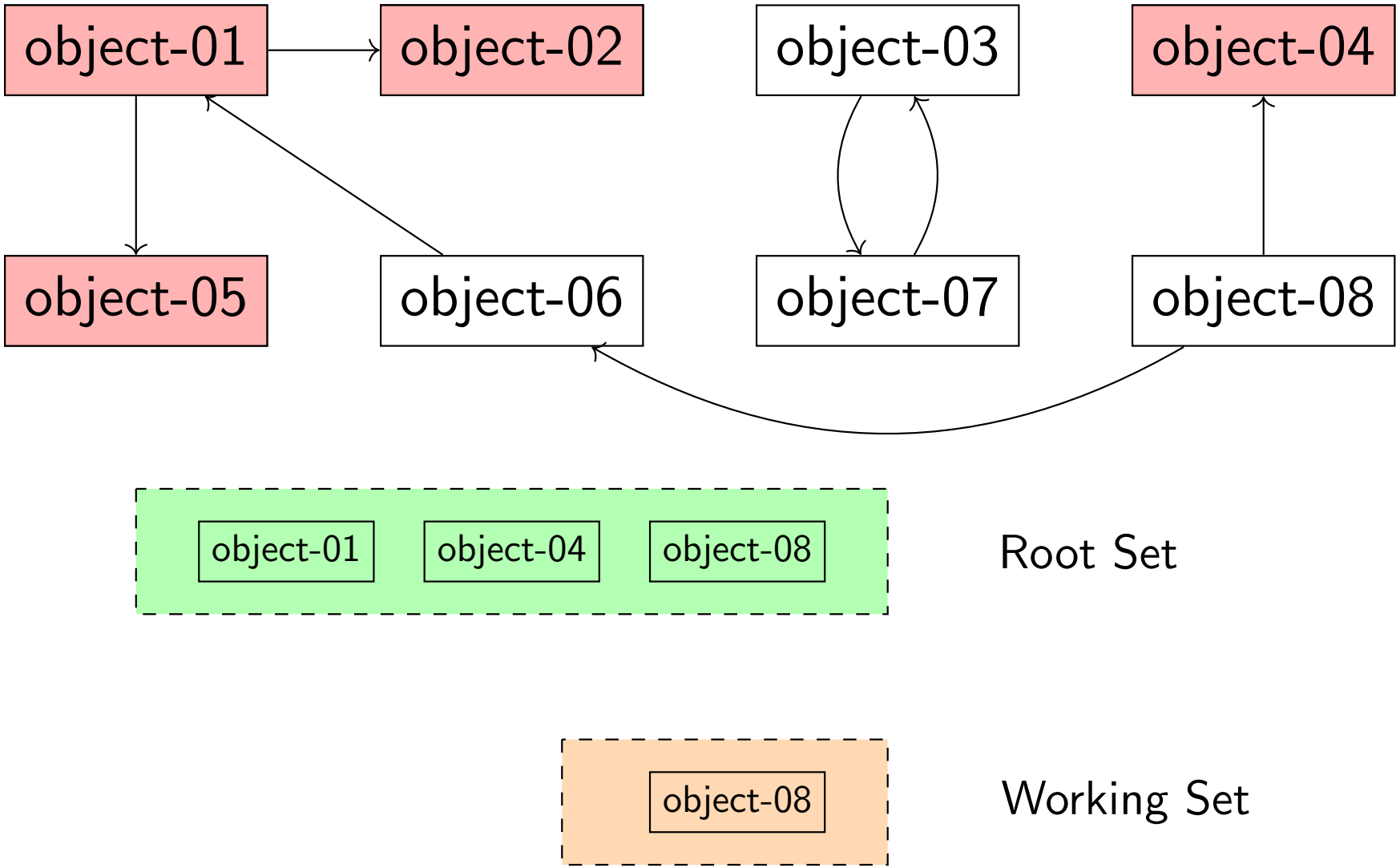


Working Set

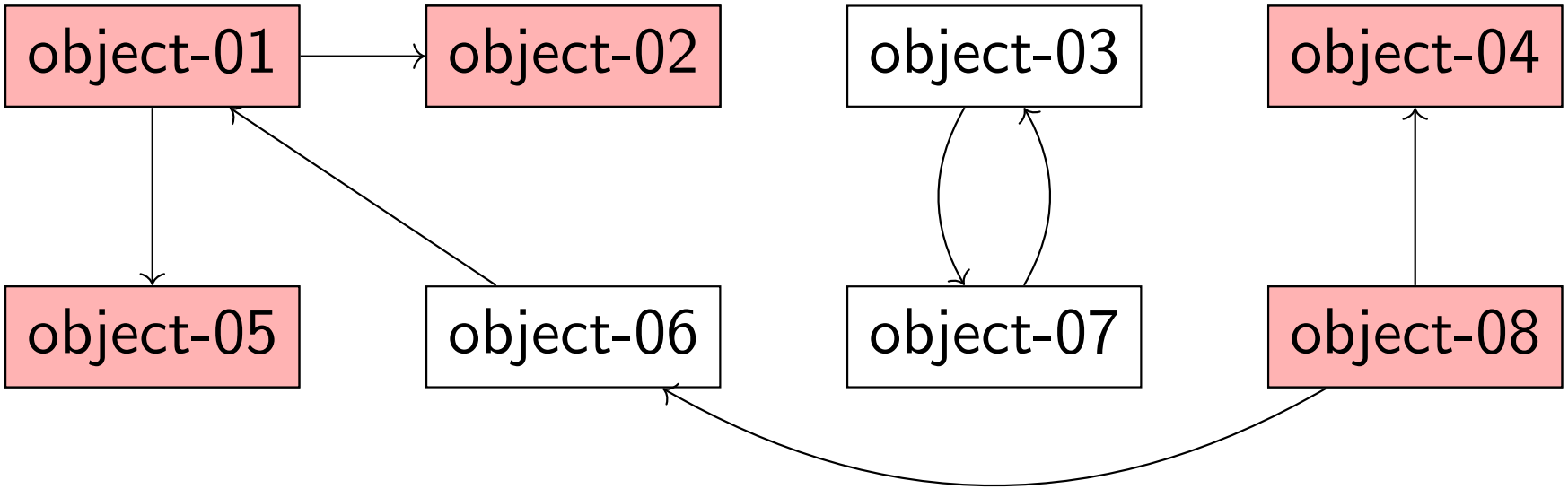
Example

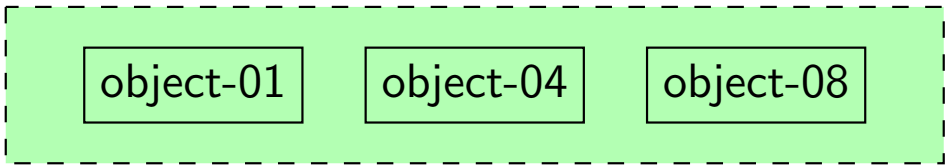

object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set
object-04
object-08
Working Set

Example

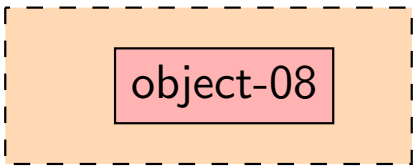

object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set
object-08
Working Set

Example



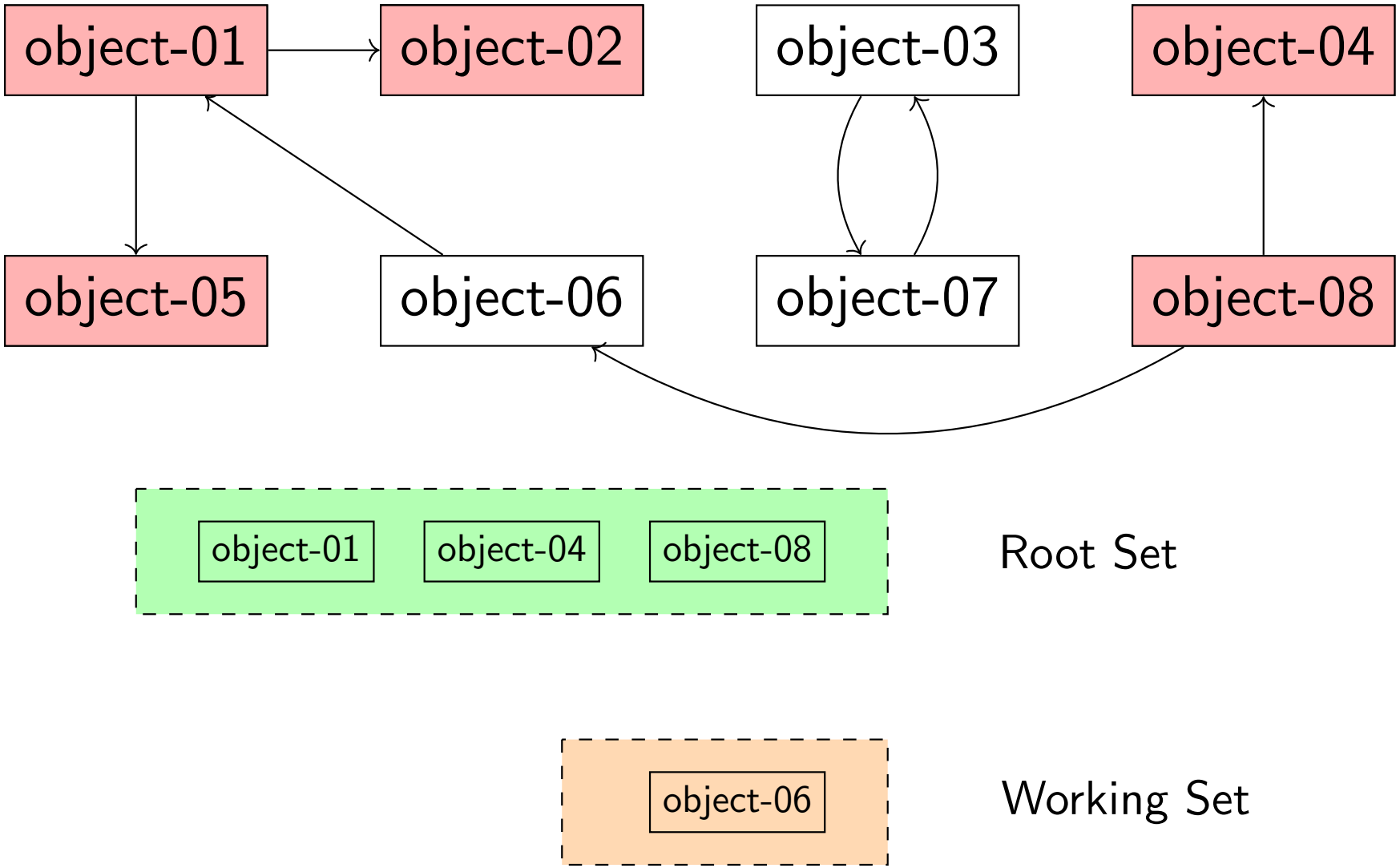


Root Set

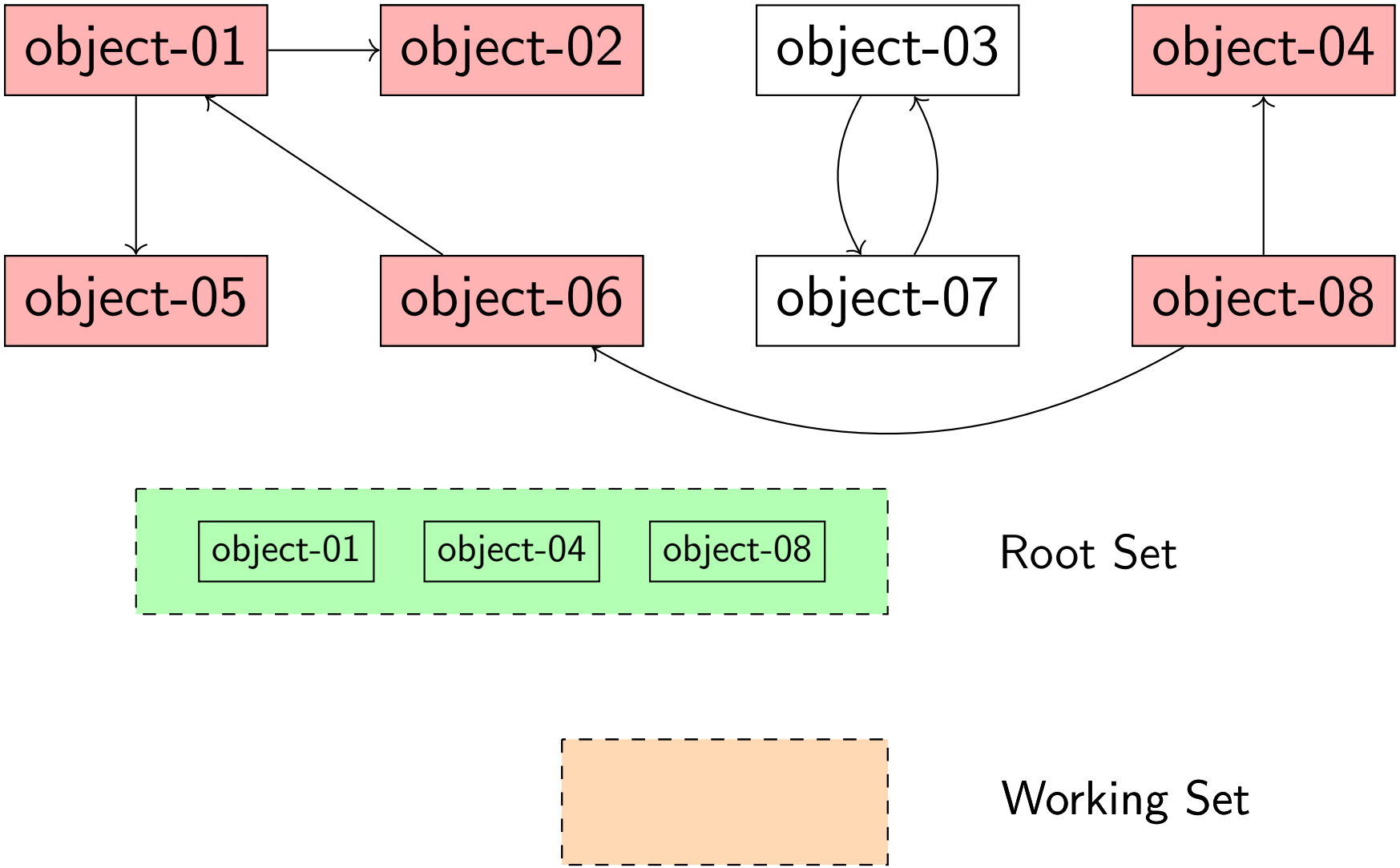


Working Set

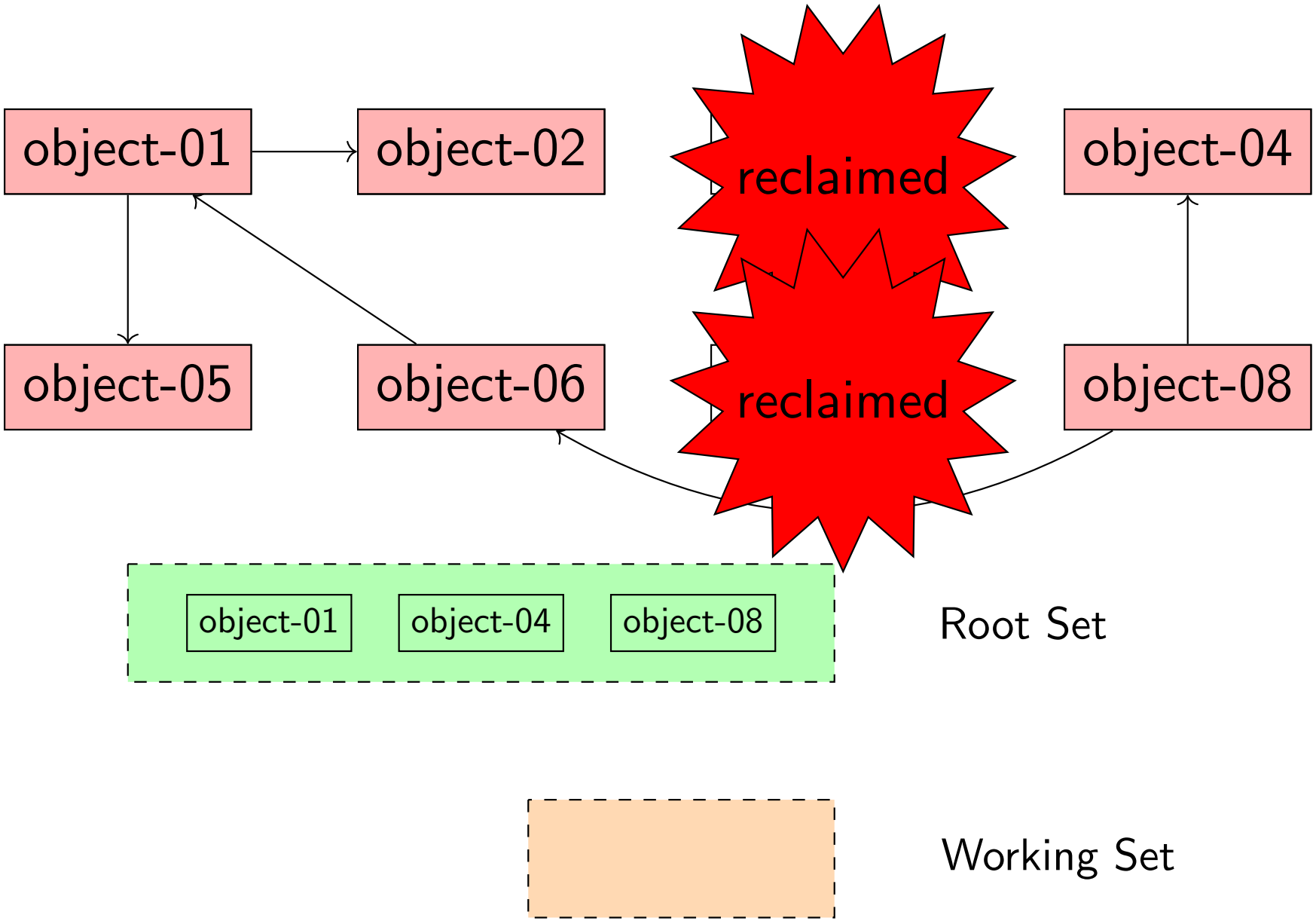
Example


object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set
object-06
Working Set

Example


object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08
object-01
object-04
object-08
Root Set
Working Set

Example


object-01
object-02
reclaimed
object-04
object-05
object-06
reclaimed
object-08
object-01
object-04
object-08
Root Set
Working Set

How to sweep?

Sweeping requires to know where are unreacheable objets !

Heap :

object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08

How to sweep?

Sweeping requires to know where are unreacheable objets !

Heap :

object-01
object-02
object-03
object-04
object-05
object-06
object-07
object-08

Just remove from the heap all non-marked objects

Problems

- Runtime proportional to number of allocated objects
  - Sweep phase visits all objects to free them or clear marks
- Work list requires lots of memory
  - Amount of space required could potentially be as large as all of memory
  - Can't preallocate this space

Pros and Cons

Pros:

- Can free cyclic references
- 1 bits per state
- Runtime can be proportional to the number of reachable objects (Baker's algorihtm)

Cons:

- Stop the world algorithm with possibly huge pauses times
- Memory Fragmentation
- Need to walk the whole heap

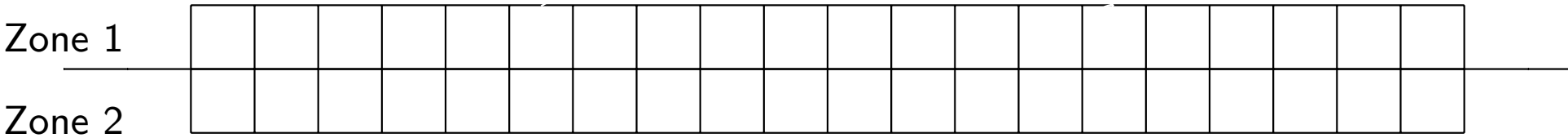
Table of contents

Analysis

- Locality can be improved
  - After garbage collection, objects are no longer closed in memory
- Allocation speed can be improved
  - After garbage collection, the free list of the allocator must be walked.

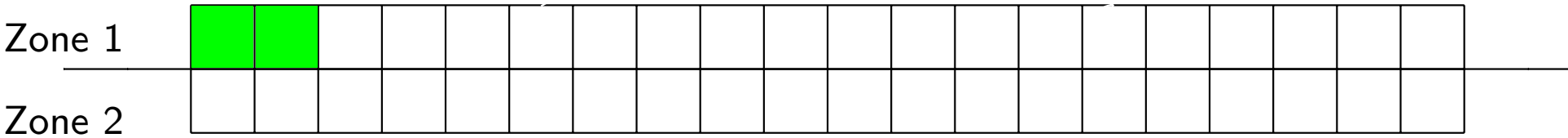
The Sweep Phase can be improved

Exemple



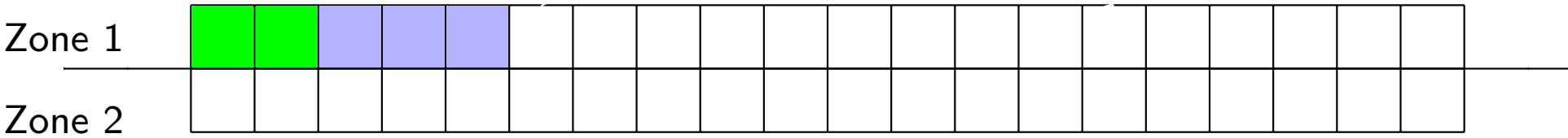
- Split memory in two pieces

Exemple



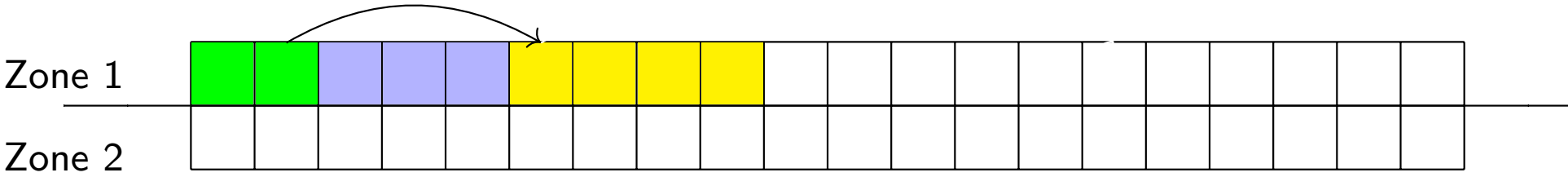
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



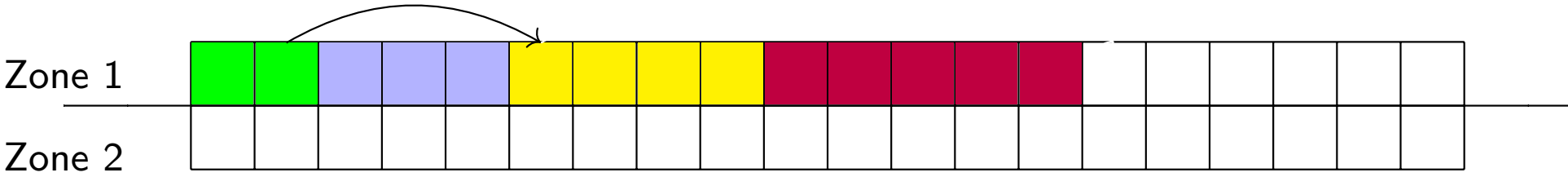
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



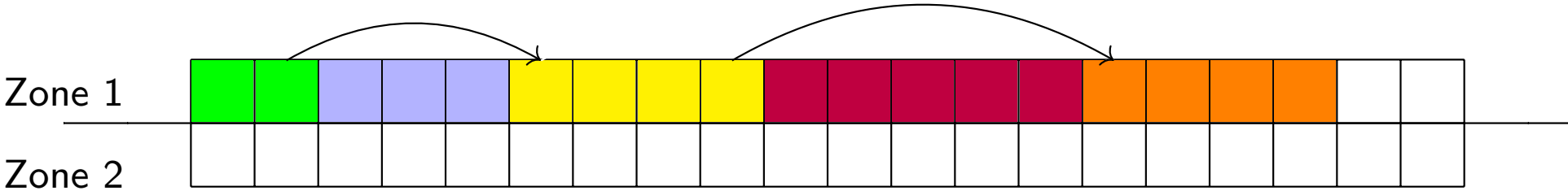
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



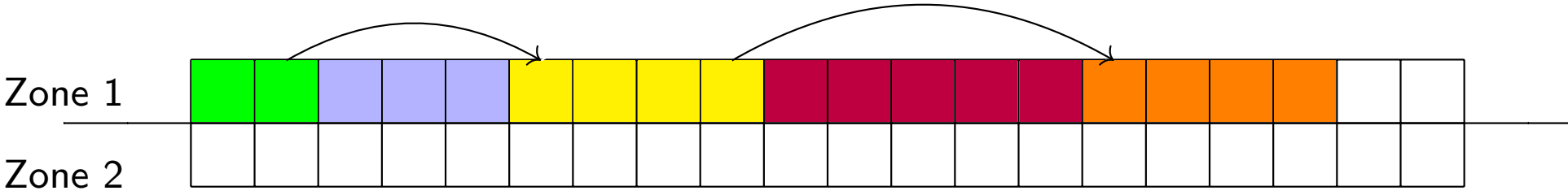
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



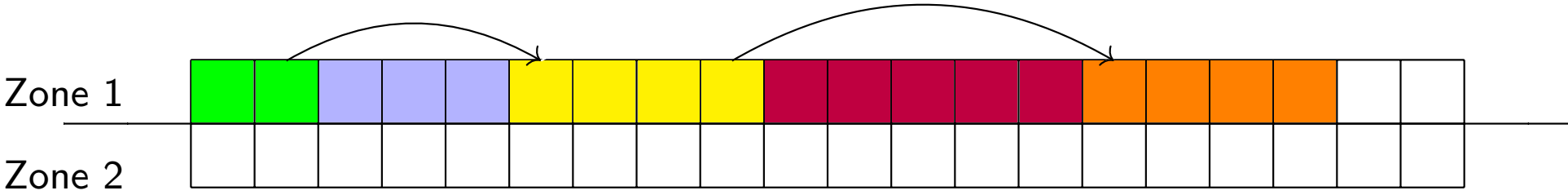
- Split memory in two pieces
- Allocate memory in the first zone

Exemple



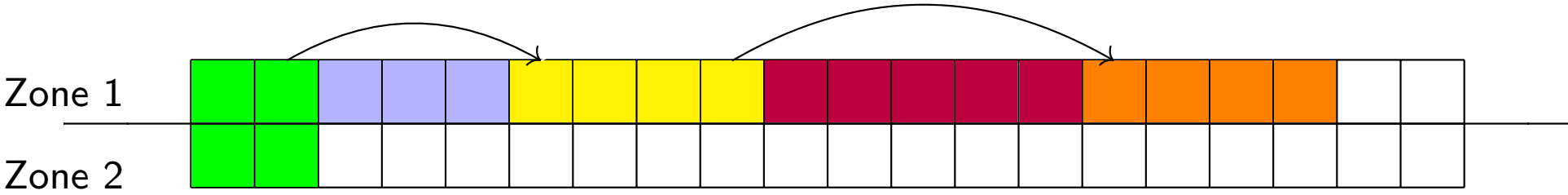
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!

Exemple



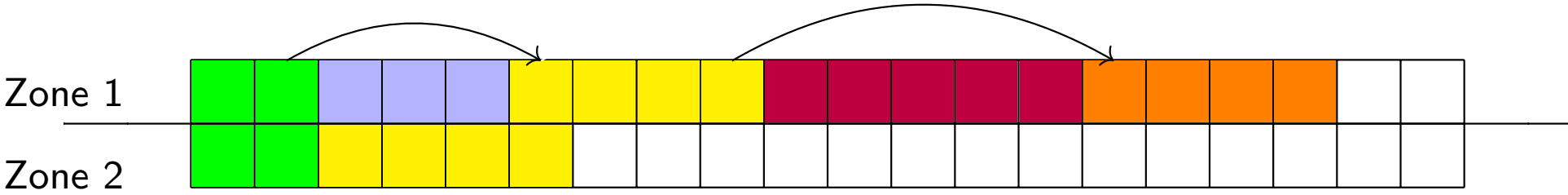
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)

Exemple



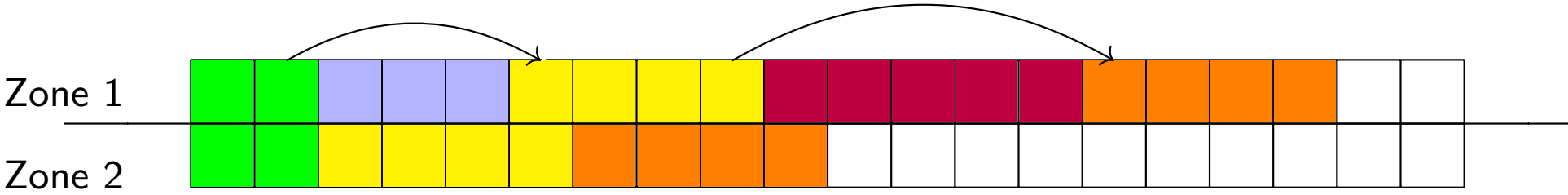
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects

Exemple



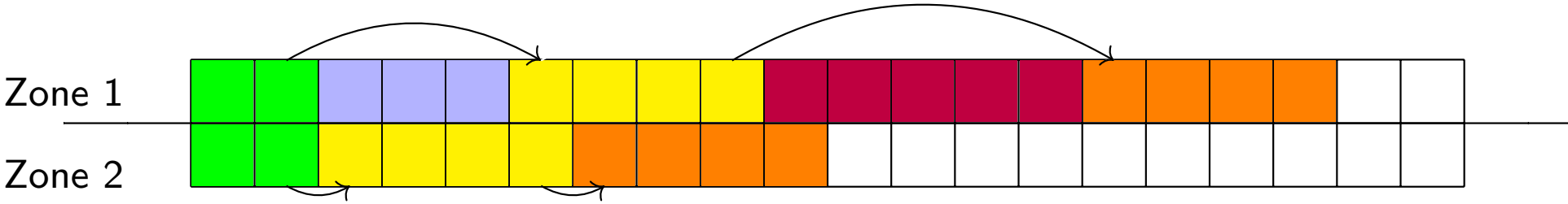
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects

Exemple



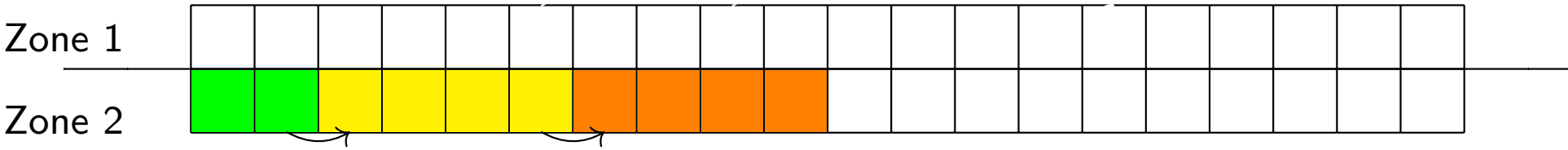
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects

Exemple



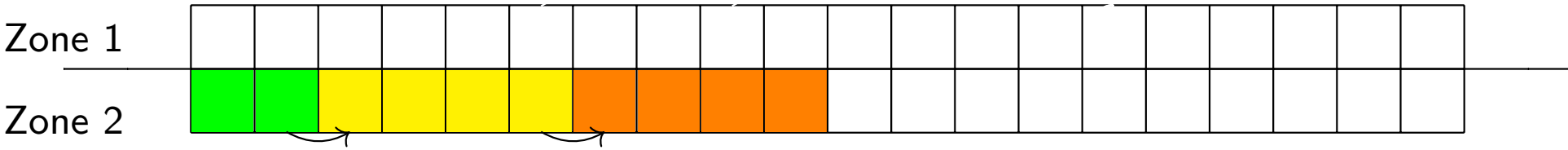
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects
- Update References & Root set

Exemple



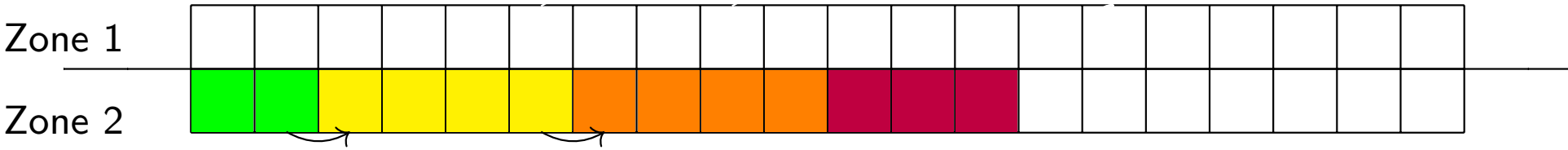
- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects
- Update References & Root set
- Clean zone 1 (Constant time)

Exemple



- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects
- Update References & Root set
- Clean zone 1 (Constant time)
- Swap zone 1 and 2 (Now allocation will happen in zone 2)

Exemple



- Split memory in two pieces
- Allocate memory in the first zone
- When running out-of-space in the first zone: Garbage Collect!
- Explore only reachable references from the root set (here only green object)
- Copy objects
- Update References & Root set
- Clean zone 1 (Constant time)
- Swap zone 1 and 2 (Now allocation will happen in zone 2)
- Allocate the object that have provoqued the GC

Implementation

- Partition memory into two regions: the old space and the new space.
- Keep track of the next free address in the new space.
- To allocate n bytes of memory:
- If n bytes space exist at the free space pointer, use those bytes and advance the pointer.
- Otherwise, do a copy step. To execute a copy step:
- For each object in the root set:
  - Copy that object over to the start of the old space.
  - Recursively copy over all objects reachable from that object.
- Adjust the pointers in the old space and root set to point to new locations.
- Exchange the roles of the old and new spaces.

Problems

How to adjust pointers in the copied objects correctly?

Problems

How to adjust pointers in the copied objects correctly?

1. Have each object contain a extra space for a forwarding pointer
2. First, do a complete bitwise copy of the object
3. Next, set the forwarding pointer of the original object to point to the new object
   - Follow the pointer to the object it references
   - Replace the pointer with the pointee's forwarding pointer

Pros and Cons

Pros:

- Compact the Heap
- Allocation only increments a pointer
- No sweep

Cons:

- Smaller Heap
- Copy
- Reference adjusting

Table of contents

Analysis

The best garbage collectors in use today are based on a combination of smaller garbage collectors

Objects Die Young

Most objects have extremely short lifetimes

Optimize garbage collection to reclaim young objects rapidly while spending less time on older objects

Generational Garbage Collector

- Partition memory into several generations
- Objects are always allocated in the first generation.
- When the first generation fills up, garbage collect it.
  - Runs quickly; collects only a small region of memory.
- Move objects that survive in the first generation long enough into the next generation.
- When no space can be found, run a full (slower) garbage collection on all of memory.

Garbage Collection in Java

1. Split the Heap in 3 zones: eden, survivors and tenured
2. New objects are allocated using a modified stop-and-copy collector in the Eden space.
3. When Eden runs out of space, the stop-and-copy collector moves its elements to the survivor space.
4. Objects that survive long enough in the survivor space become tenured and are moved to the tenured space.
5. When memory fills up, a full garbage collection (perhaps mark-and-sweep) is used to garbage-collect the tenured objects

Garbage Collection in C

- Boehm GC
- Mark and Sweep
- Conservative
- Consider all program variables as root set
- Easy to combine with C

Bibliography

- Uniprocessor Garbage Collection , Paul R. Wilson