Intermediate Representations

Mooly Sagiv

http://ellcc.org/demo/index.cgi
llvm.org
https://www.cis.upenn.edu/~stevez/ CS341

Date	Lecture	Recitation	Assignment
20/10	Overview & AST	MiniJava	
27/10	Assembler & Frames	Visitor patterns	Variable&method renaming (19/11)
3/11	Simplified translation& DP	Symbol Tables	
10/11	IR+LLVM	LLVM	
17/11	LLVM Code Generation	LLVM Code Generation	Code generation (10/12)
26/11	Object Oriented Code Generation	LLVM Object Oriented Code Generation	
1/12	Semantic Analysis	Semantic Analysis and Type Checking	
8/12	Static Analysis	Static Analysis	Semantic Analysis (30/12)
15/12	Lexical Analysis	Lexical Analysis	
22/12	Top-Down Parsing	Top-Down Parsing	Lexing & Parsing (14/1)
29/12	Bottom-Up Parsing	Bottom-Up Parsing	
5/1	X86 Code Generation	X86 Code Generation & AR	
12/1	Advanced Topics	Rehearsal	

Outline

- Recap AST → X86
- IRs
- LLVM by example
- Compiling LLVM into X86
  - Register allocation
  - Instruction selection

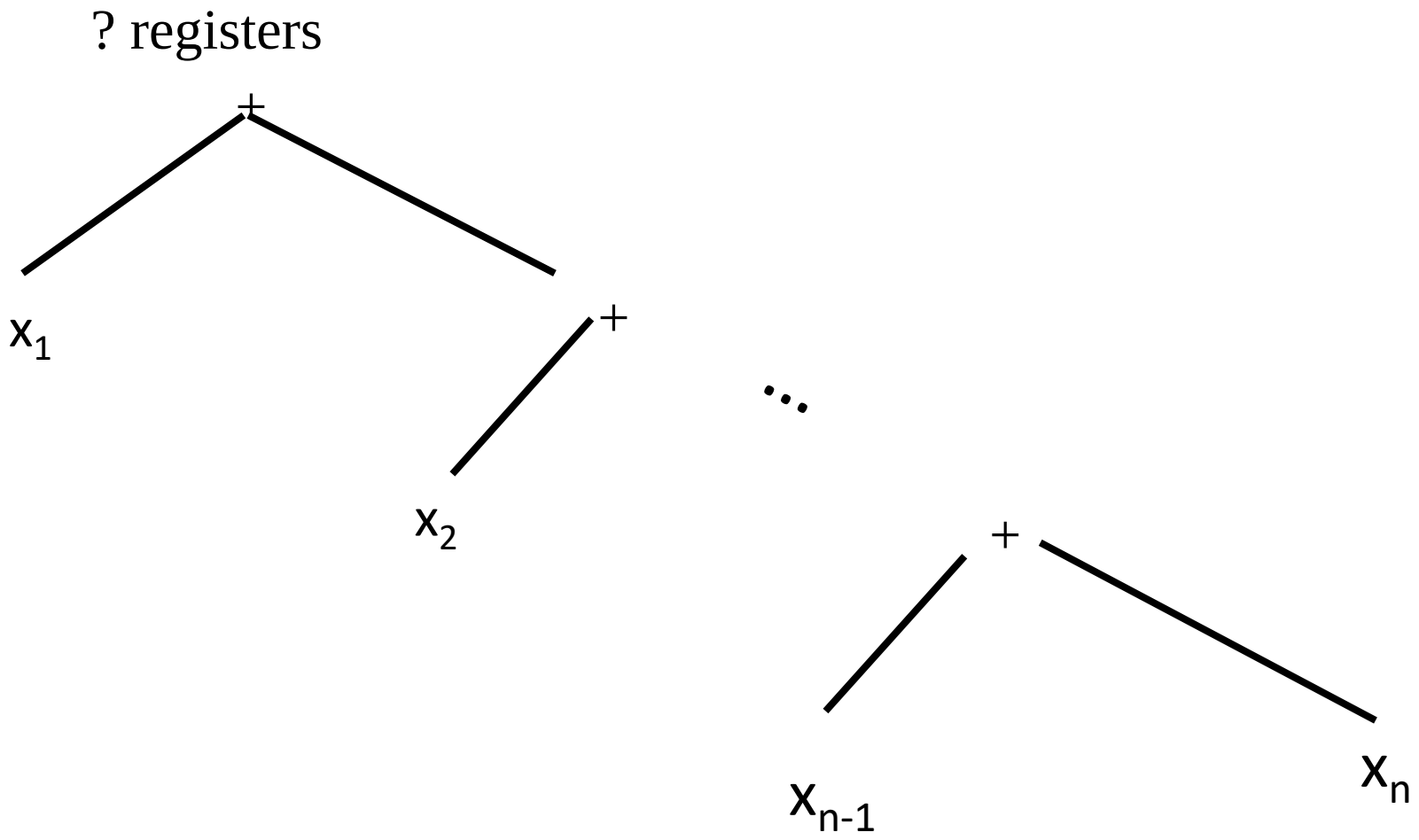
Questions

- What is the maximal number of registers required in the code generated from a given AST?
  - What kind of trees generate that?
- What is the difference between the code generated for caller/callee saved register?

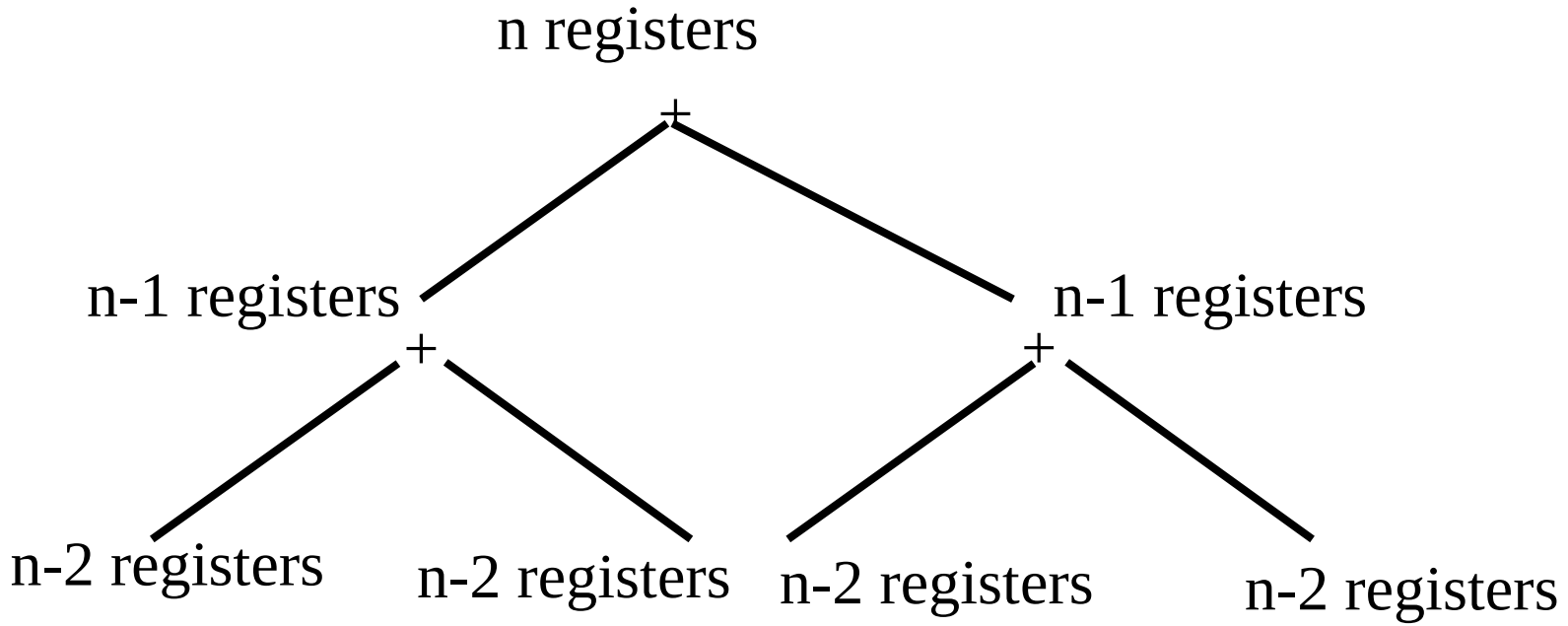
Two Phase Solution
Dynamic Programming
Sethi & Ullman (R

- Bottom-up (labeling)
  - Compute for every subtree
    - The minimal number of registers needed
    - Weight
- Top-Down
  - Generate the code using labeling by preferring “heavier” subtrees (larger labeling)
  - Can integrate spilling

“Good” tree



“Bad” tree


n registers
+
n-1 registers
+
n-1 registers
+
n-2 registers
n-2 registers
n-2 registers
n-2 registers

The need for global register allocation

```
int foo() {
  int x = 1 ;
  x = x + 1;
  x = x + 1;
 ...
 printf("%d", x);
}
```

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 1
        add     DWORD PTR [rbp-4], 1
        add     DWORD PTR [rbp-4], 1
        mov     eax, DWORD PTR [rbp-4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        nop
        leave
        ret
```

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     eax, 1
        add     eax, 1
        add     eax, 1
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        nop
        leave
        ret
```

Caller-Save and Callee-Save Registers

- callee-save-registers (MIPS 16-23, X86 r12-15, rbp, rsp)
  - Saved by the callee when modified
  - Values are automatically preserved across calls
- caller-save-registers
  - Saved by the caller when needed
  - Values are not automatically preserved
- Usually the architecture defines caller-save and callee-save registers
  - Separate compilation
  - Interoperability between code produced by different compilers/languages
- But compilers can decide when to use calller/callee registers

X86lite Registers: 16 64-bit registers

register	usage	Callee save
rax	general purpose accumulator	N
rbx	base register, pointer to data	N
rcx	counter register for strings & loops	N
rdx	data register for I/O	N
rsi	pointer register, string source register	N
rdi	pointer register, string destination register	N
rbp	base pointer, points to the stack frame	Y
rsp	stack pointer, points to the top of the stack	Y
r08-r11	General purpose registers	N
r12-15	General purpose registers	Y

Maintained Invariants: Callee Saved Registers

- Save (usually in the stack) before first use
- Restore before the call is ended
- Architecture support

Maintained Invariants: Caller Saved Registers

- Save (usually in the stack) before call if value is needed
  - Restore after call
- Architecture support requires more effort

The need for global register allocation

```
int foo() {
  int x = 1 ;
  x = x + 1;
  bar();
  x = x + 1;
 printf("%d", x);
}
```

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     eax, 1
        add     eax, 1
         ....
        add     eax, 1
         mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        nop
        leave
        ret
```

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     eax, 1
        add     eax, 1
        push eax
        call bar()
         pop eax
        add     eax, 1
         mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        nop
        leave
        ret
```

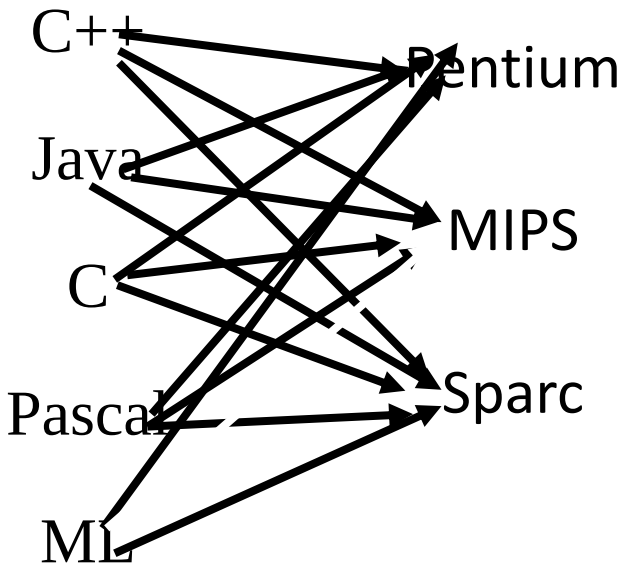
The Code Generation Problem

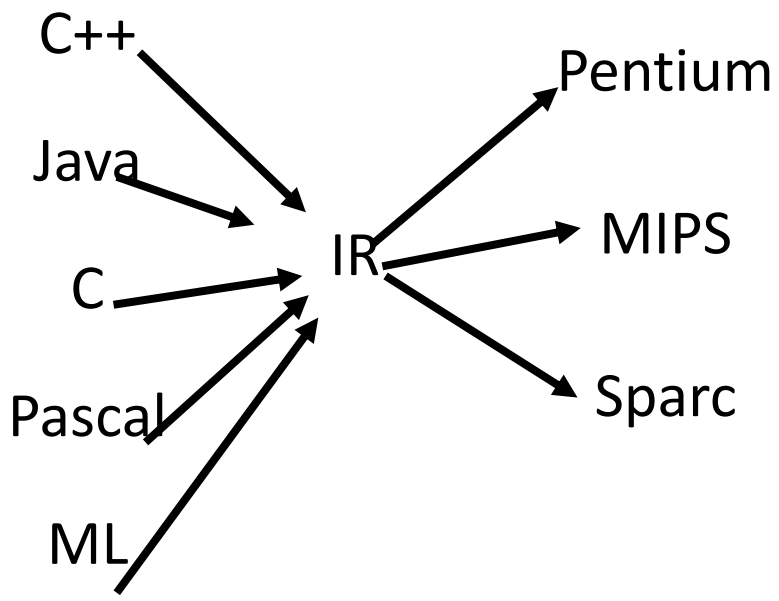
- Input: A high level program
- Output: Assembly Program
- Two related problems:
  - Instruction selection
  - Register allocation
- The problem is very hard
  - Compilers break the program into subprograms and compile them separately maintaining invariants
  - Good but not optimal code

Granularity of compilation	Complexity of optimal register allocation
Expression trees	Linear
Procedure	NP hard - Undecidable

Why intermediate representation?

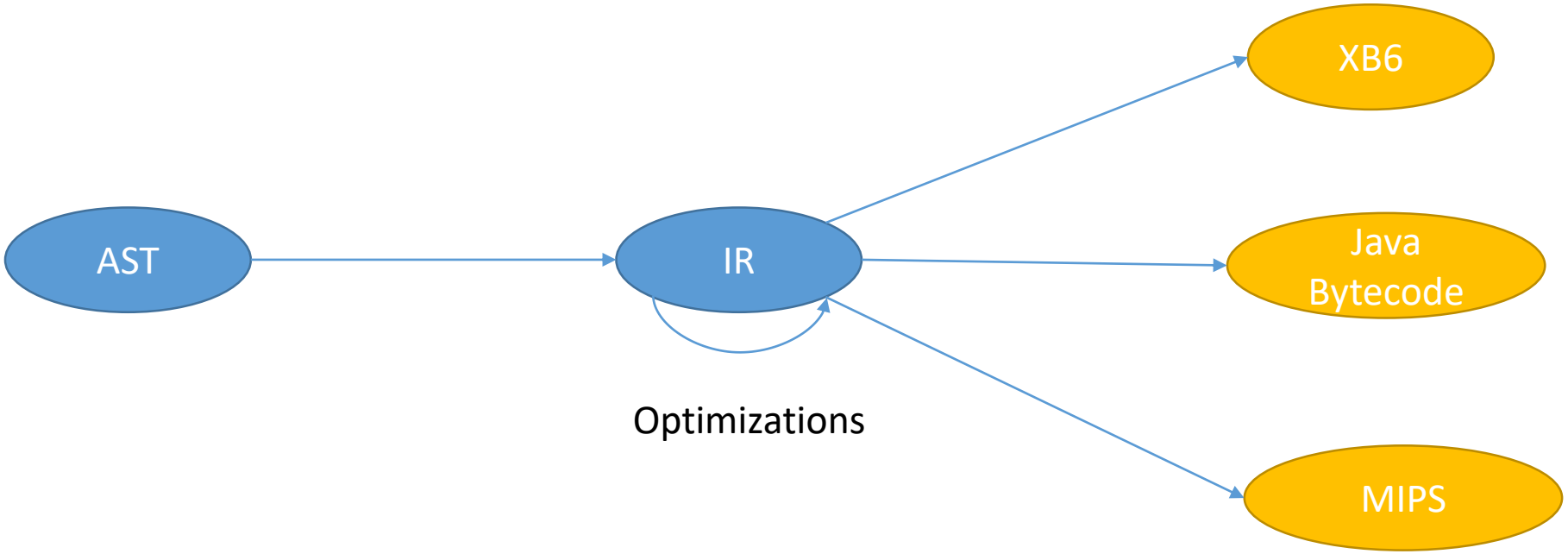
- Breaks the compilation into well understood components
  - Well tunes compilation techniques
    - Instruction selection
    - Register allocation
  - More efficient generated code
- Reuse across different machines
- Reuse across different high level languages





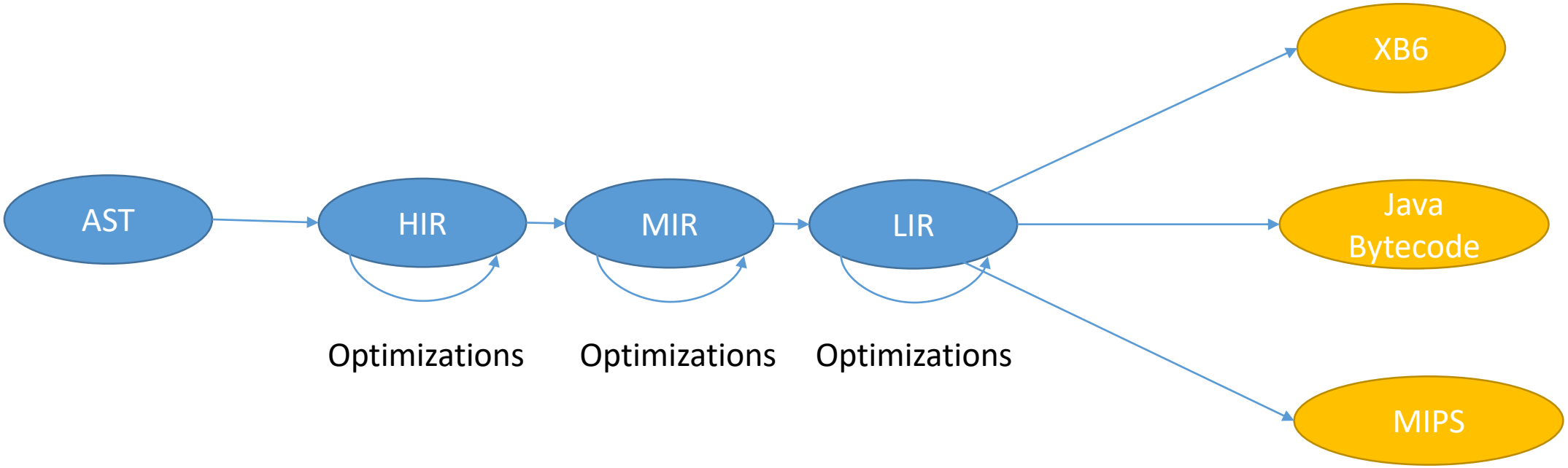
Intermediate Representations(IRs)

- Abstract machine code: hides details of the target architecture
- Allows machine independent code generation and optimization



Multiple IRs

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
- Multiple intermediate representations used for different purposes



What makes a good IR?

- Easy translation target
  - from the level above
- Easy to translate
  - to the level below
- Narrow interface
  - Fewer constructs means simpler phases/optimizations
- Example: Source language might have “while”, “for”, and “foreach” loops (and maybe more variants)
  - "for(<pre>;<cond>;<post>){<body>” ≡<pre>; while<cond>{<body>; <post>}

IR’s at the extreme

- High-level IR’s
  - AST + Extra nodes with type information
  - Normal form
    - Core language
      - “a[i]” ≡ *(a + i) ≡ *(i + a) ≡ “i[a]”
- Machine dependent assembly code
  - Extra pseudo code
    - interfacing with garbage collector or memory allocators
  - Unbounded number of registers
  - Unify certain instructions
    - General multiplications
    - Brunch/Jump

X86 with symbolic registers

```
int foo() {
  int x = 1 ;
  x = x + 1;
  x = x + 1;
 printf("%d", x);
}
```

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     s1, 1
        mov s2, s1
        add     s2, 1
        mov   s3, s2
        add     s3, 1
        mov     esi, s3
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        leave
        ret
```

Sym	Real
s1	eax
s2	eax
s3	eax

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     eax, 1
        mov eax, eax
        add     eax, 1
        mov   eax, eax
        add     eax, 1
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        leave
        ret
```

X86 with symbolic registers

```
int foo() {
  int x = 1 ;
  x = x + 1;
  x = x + 1;
 printf("%d", x);
}
```

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     s1, 1
        mov s2, s1
        add     s2, 1
        mov   s3, s2
        add     s3, 1
        mov     esi, s3
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        leave
        ret
```

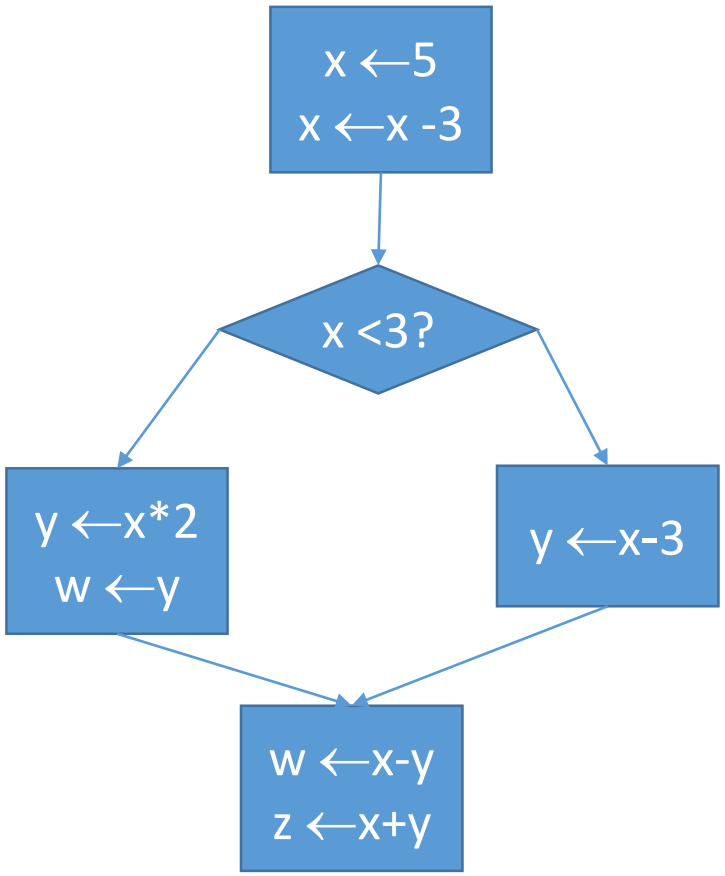
Sym	Real
s1	eax
s2	eax
s3	eax

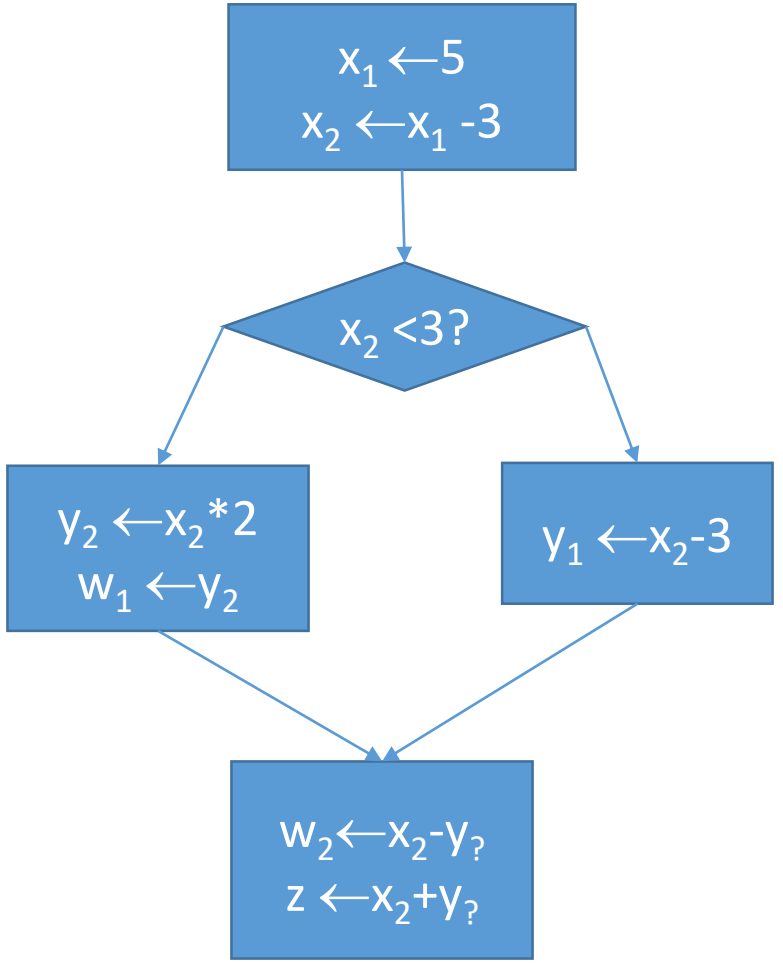
```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     eax, 1
        mov eax, eax
        add     eax, 1
        mov   eax, eax
        add     eax, 1
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        leave
        ret
```

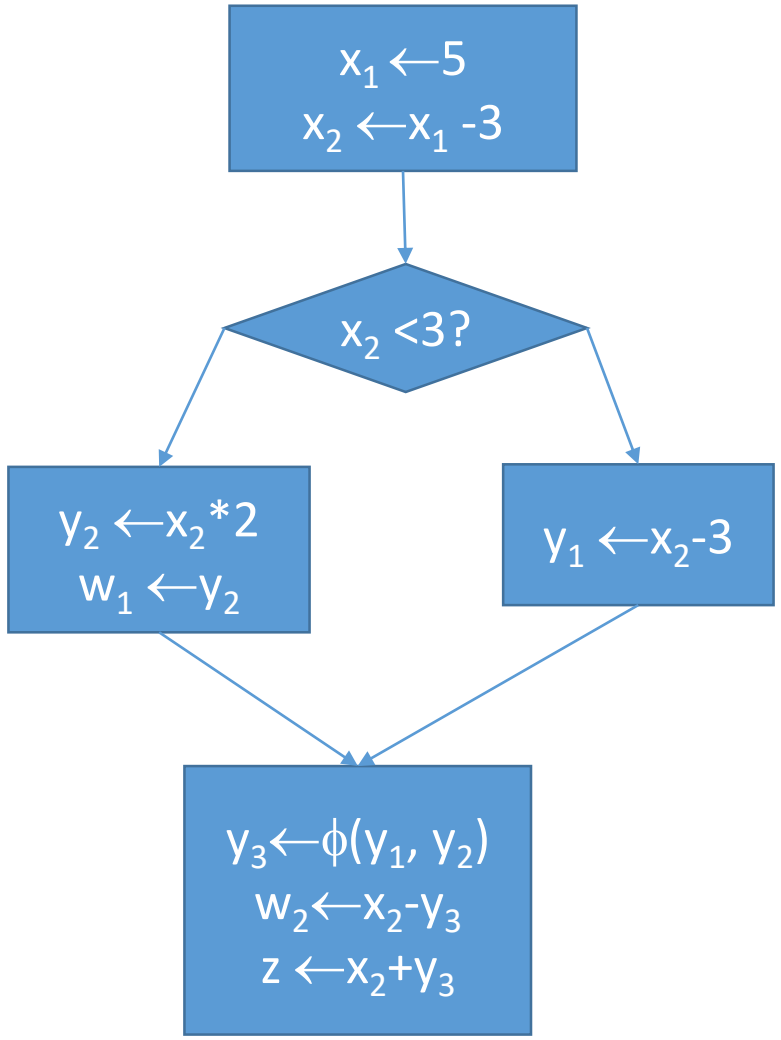
Static Single Assignment(SSA)

- Every variable has a unique assignment
  - Defined before used
- Makes the program functional
- Simplifies program reasoning

Converting to SSA







Source: Wikipedia

Mid-level IR's: Many Varieties

- Intermediate between AST (abstract syntax) and assembly
- May have unstructured jumps, abstract registers or memory locations
- Convenient for translation to high-quality machine code

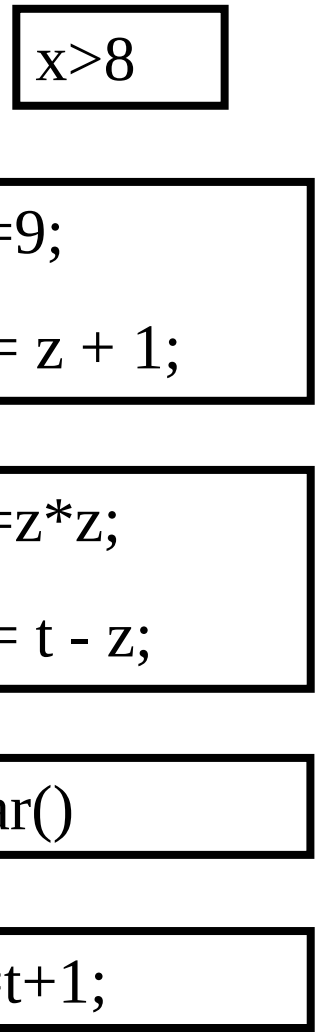
IR	Examples	Pros	Cons
Quadruples a = b op c (3-address)	RISC	Flexible	Suboptimal X86 translation
Variant of quadruples with SSA	LLVM	Facilitates optimizations	Verbose
Triples(2-addres) x = x op y		Easy to generate X86 via tiling	Register allocation may be harder
Stack based	UCODE, Java Bytecode	Easy to generate code	Hard to optimize

Basic Block

- Parts of control graph without split
- A sequence of assignments and expressions which are always executed together
- Maximal Basic Block Cannot be extended
  - Start at label or at routine entry
  - Ends just before jump like node, label, procedure call, routine exit

Example Basic Blocks

```
void foo()
{
    if (x > 8) {
        z = 9;
        t = z + 1;
        }
     z = z * z;
     t = t – z ;
     bar();
     t = t + 1;
```

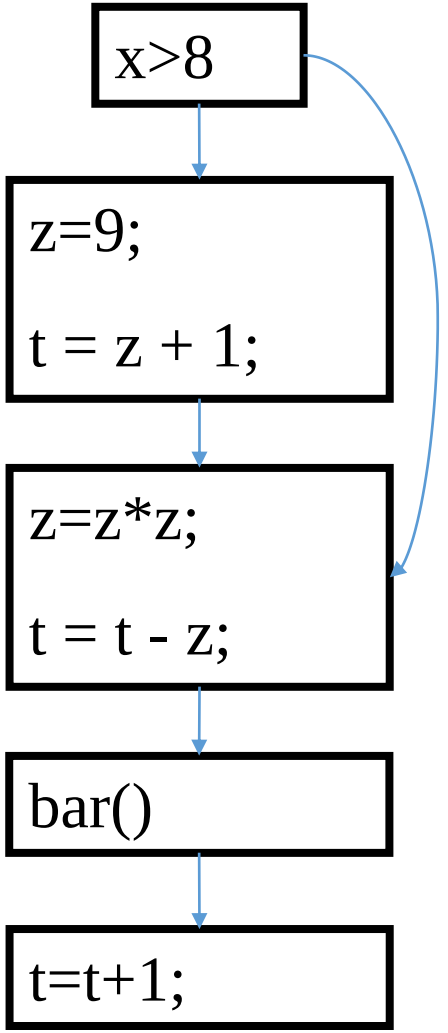


Control Flow Graph

- The compiler does not know the actual executions
- A finite directed graph conservatively represents all behaviors
  - Nodes are basic blocks
  - Edges represent immediate flow of control

Example Control Flow Graph

```
void foo()
{
    if (x > 8) {
        z = 9;
        t = z + 1;
        }
     z = z * z;
     t = t – z ;
     bar();
     t = t + 1;
```



Constructing Basic Blocks

- Applied for each function body
- Scan the statement list from left to right
- Whenever a LABEL is found
  - a new block begins (and the previous block ends)
- Whenever JUMP or BRANCH are found
  - the current block ends (and the next block begins)
- When a block ends without JUMP or BRANCH}
  - JUMP to the following LABEL
- When a block does not start with a LABEL
  - Add a LABEL
- At the end of the function body jump to the beginning of the epilogue

What is the LLVM Project?

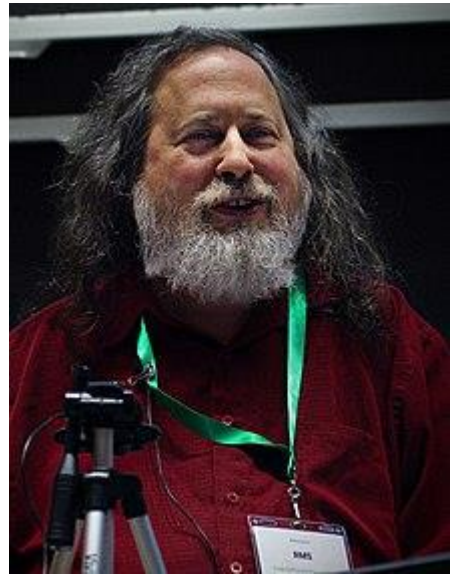
- Collection of industrial strength compiler technology
- Optimizer and Code Generator
  - llvm-gcc and Clang Front-ends
  - MSIL and .NET Virtual Machines
  - Started as a PhD by Chris Lattner
    - University of Illinois Urbana-Champaign
    - ACM Software System Award

LLVM Vision and Approach

- build a set of modular compiler components
- Reduces the time & cost to construct a particular compiler
- Components are shared across different compilers
- Allows choice of the right component for the job

GNU Compiler Collection(gcc)

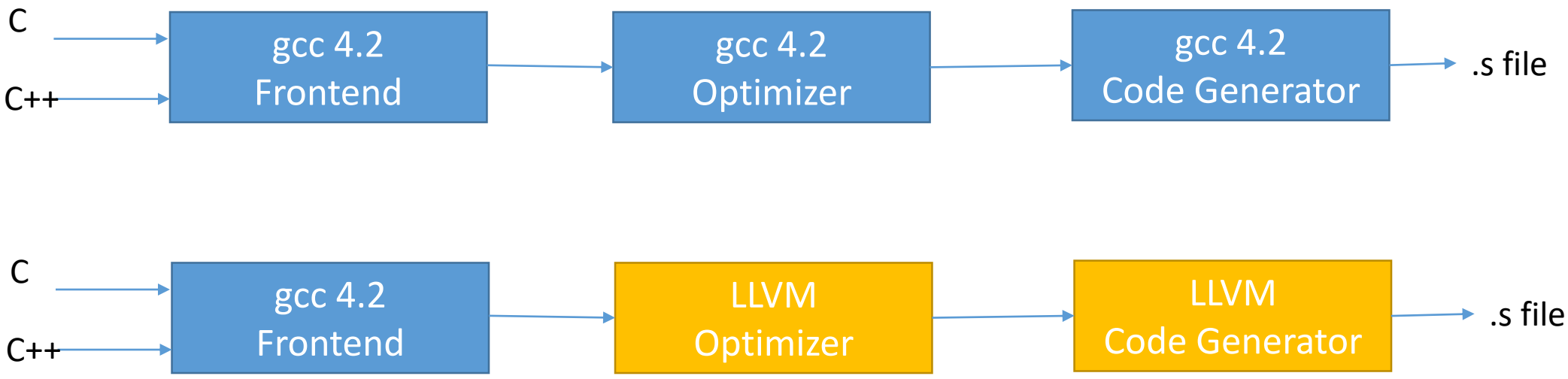
- GNU Project
- 1987—now
- Various programming languages and architectures
- Highly optimized
- 7,348,239 lines of code
- Hard to extend



Richard Stallman

LLVM gcc4.2 Design

- Reuses gcc optimizer and code generation with LLVM
  - Reuses parser and runtime libraries



Compiling factorial

```
int factorial(int num) {
   if (num == 1) return 1 ;
   else return num * factorial(num -1 );
}
```

```
define i32 @fact(i32) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  %4 = load i32, i32* %3, align 4
  %5 = icmp eq i32 %4, 1
  br i1 %5, label %6, label %7
; <label>:6:                                       ; preds = %1
  store i32 1, i32* %2, align 4
  br label %13
; <label>:7:                                       ; preds = %1
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call i32 @fact(i32 %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %13
; <label>:13:                                      ; preds = %7, %6
  %14 = load i32, i32* %2, align 4
  ret i32 %14
}
```

Compiling LLVM to X86

- Instruction selection
  - Map sequences of LLVM instructions into X86
  - Compile 3-address into 2-address
    - “%10 = **sub** nsw i32 %9, 1” ≡ mov %10, %9 ; sub32 %10, 1

- Register allocation
  - Allocate physical registers to symbolic
  - Increase stack frame if necessary

Instruction Selection

- Every instruction has a cost
- “Tile” every LLVM instruction with an appropriate sequence
- Can deploy dynamic programming

Register Allocation

- Map symbolic registers into physical
- Chose between caller= and callee-save registers
- Reuse machine registers
- Avoid store/loads
- Sometimes eliminate mov
  - Allocate the same register to source and target

A Simple Example

```
L0:     a ← 0
L1:     b ← a + 1
        c ← c + b
        a ← b * 2
        if c < N goto L1
        return c
```

```
L0:     r1← 0
L1:     r1← r1 + 1
        r2 ← r2 + r1
        r1 ← r1 * 2
        if r2 < N goto L1
        return r2
```

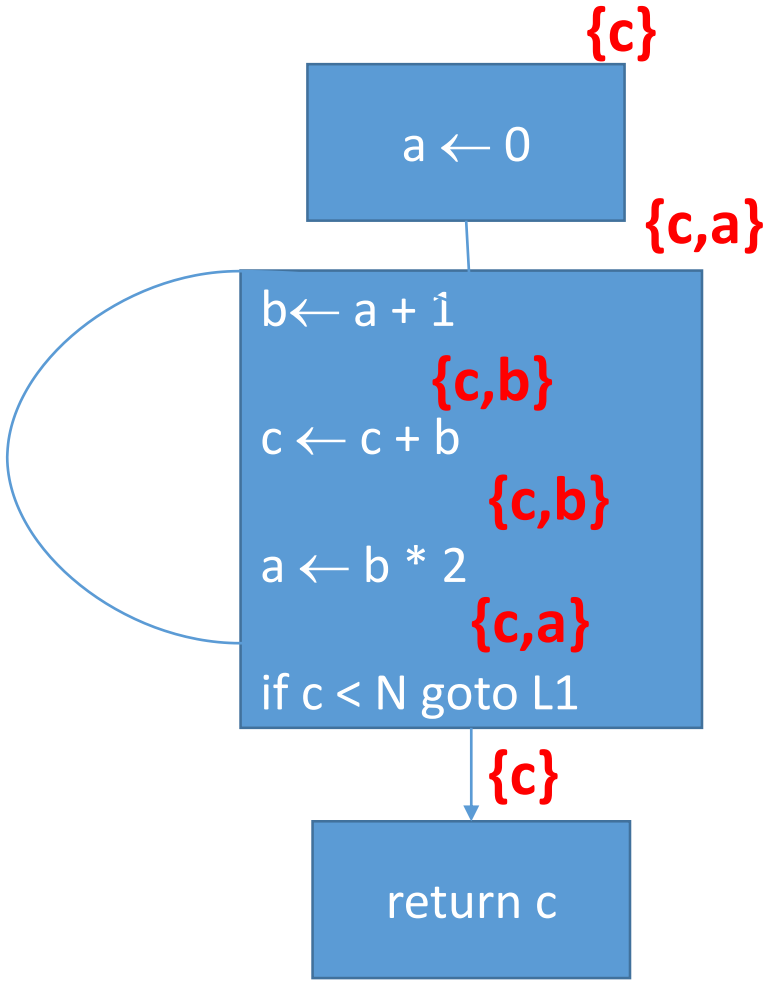
Can this be implemented in a machine with two registers?

Live symbolic registers

- A symbolic register is live at a program point if it may be used before set on some path from this point
- A symbolic register is not live (dead) at a program point if it is not used on all paths from this point

Liveness in the example

```
L0:     a ← 0
L1:     b ← a + 1
        c ← c + b
        a ← b * 2
        if c < N goto L1
        return c
```



Which variables are live at the entry to the procedure?

```
void foo()
{
    if (x > 8) {
        z = 9;
        t = z + 1;
        }
     z = z * z;
     t = t – z ;
     bar();
     t = t + 1;
```

Live symbolic registers

- A symbolic register is live at a program point if it may be used before set on some path from this point
- A symbolic register is not live (dead) at a program point if it is not used on all paths from this point
- The problem of computing liveness is undecidable

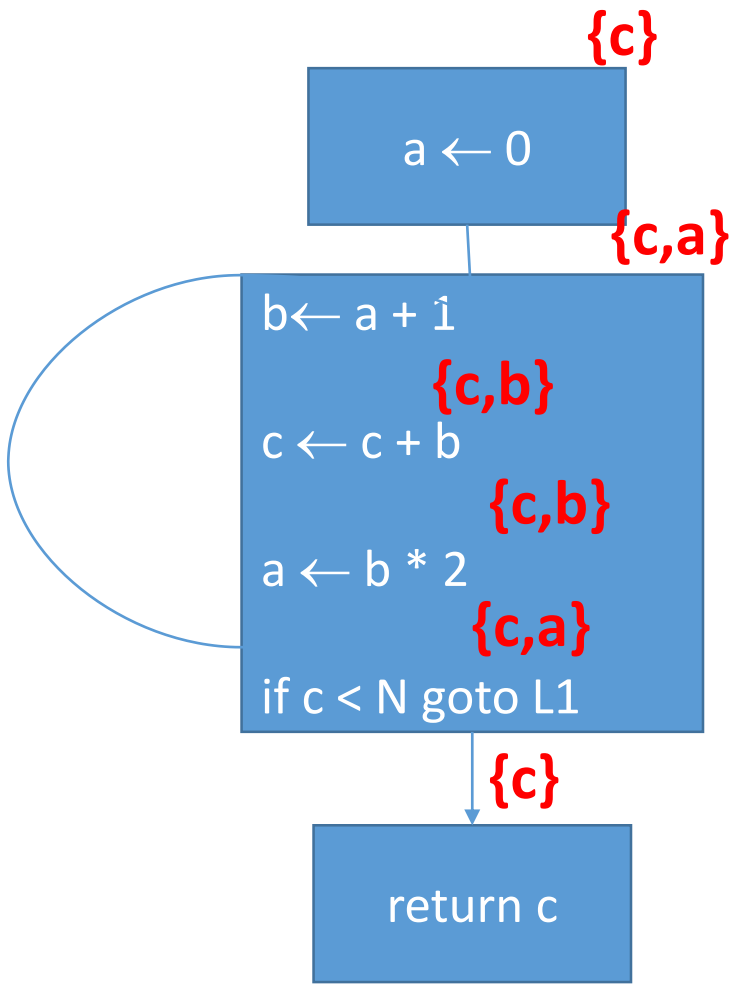
```
x = 5;
foo();
y = x;
```

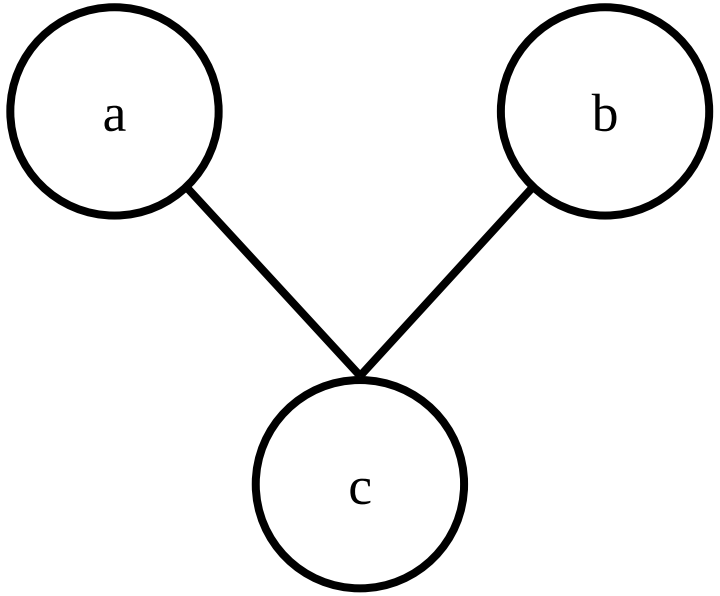
- But the compiler can over-approximate
  - Every live variable is detected

Using Liveness information

- Symbolic Registers which are not live together can share the same symbolic register

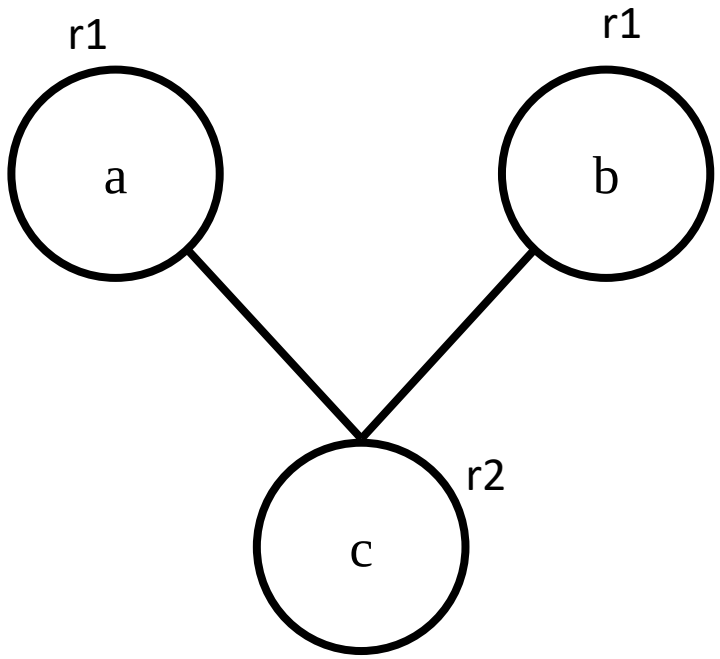
Using Liveness Information





Coloring the graph

```
L0:     a ← 0
L1:     b ← a + 1
        c ← c + b
        a ← b * 2
        if c < N goto L1
        return c
```




```
L0:     r1← 0
L1:     r1← r1 + 1
        r2 ← r2 + r1
        r1 ← r1 * 2
        if r2 < N goto L1
        return c
```

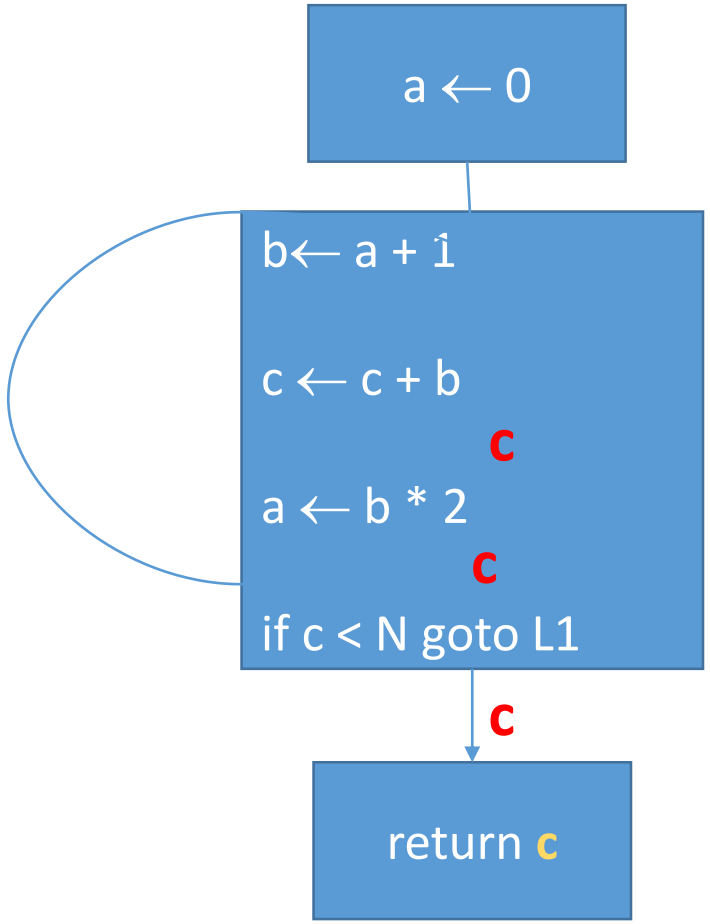
Remaining Problems

- Compute liveness information
- Color the graph

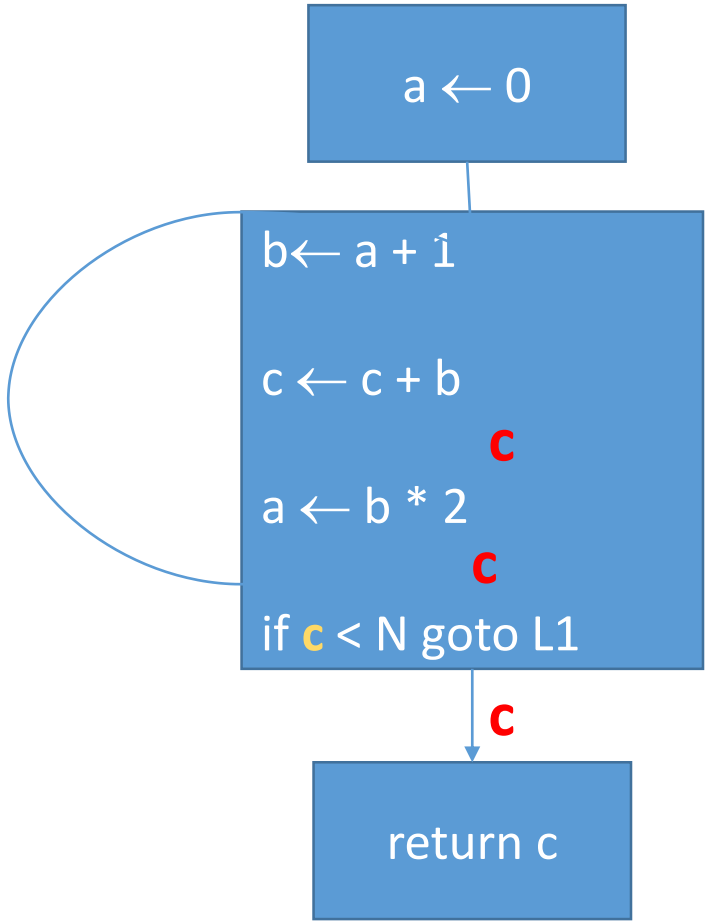
Computing Liveness(Simple Algorithm)

- Reverse the control flow graph
- Every variable is live from its use until the first assignment
- Can be computed via Depth First Search
  - Cycles do not matter

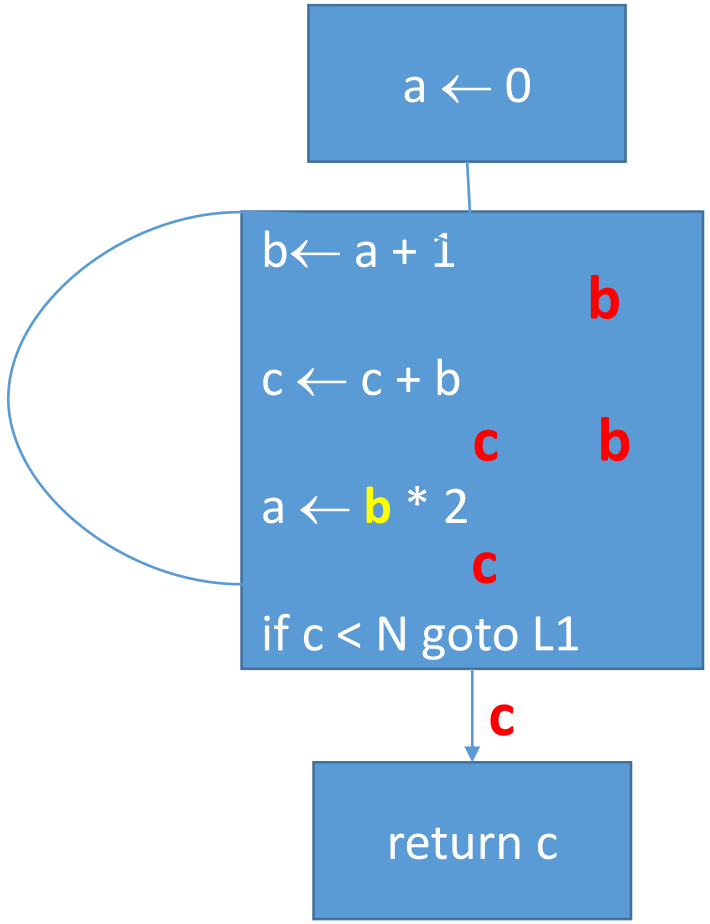
Computing Liveness via DFS(1)



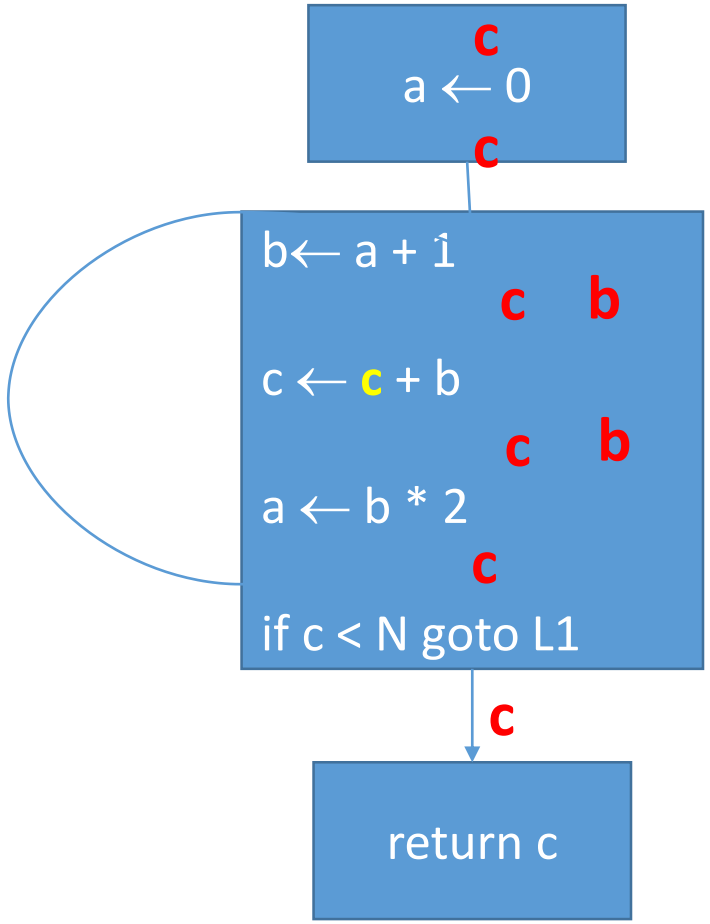
Computing Liveness via DFS(2)



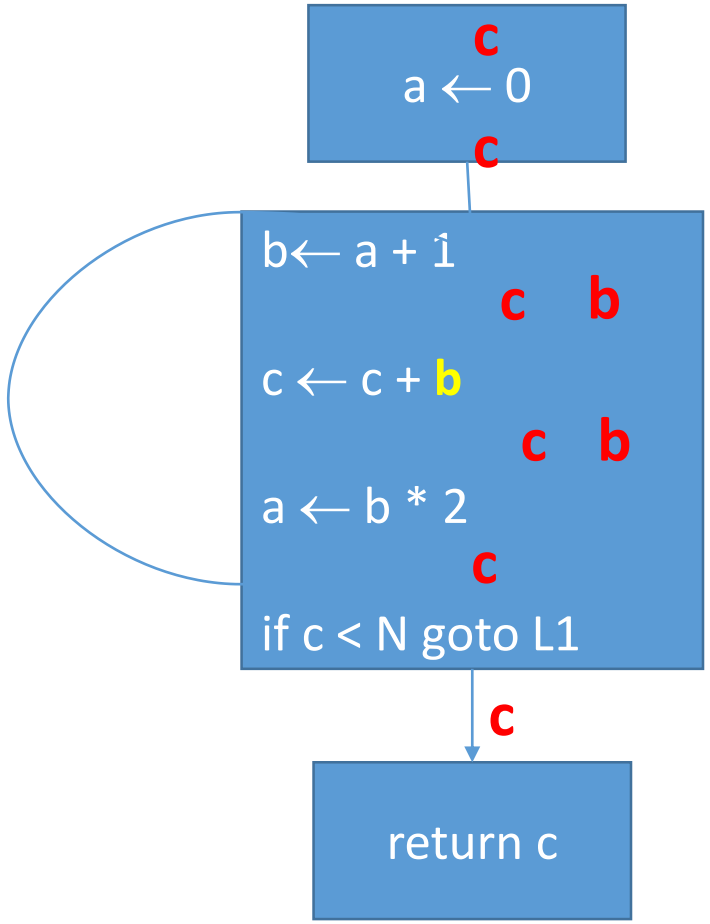
Computing Liveness via DFS(3)



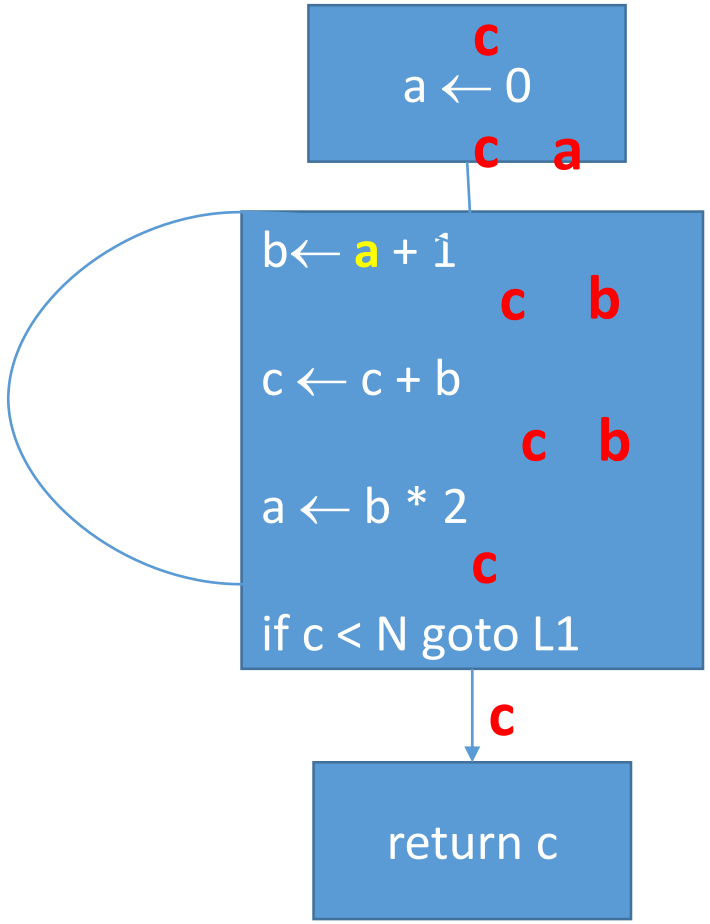
Computing Liveness via DFS(4)



Computing Liveness via DFS(5)



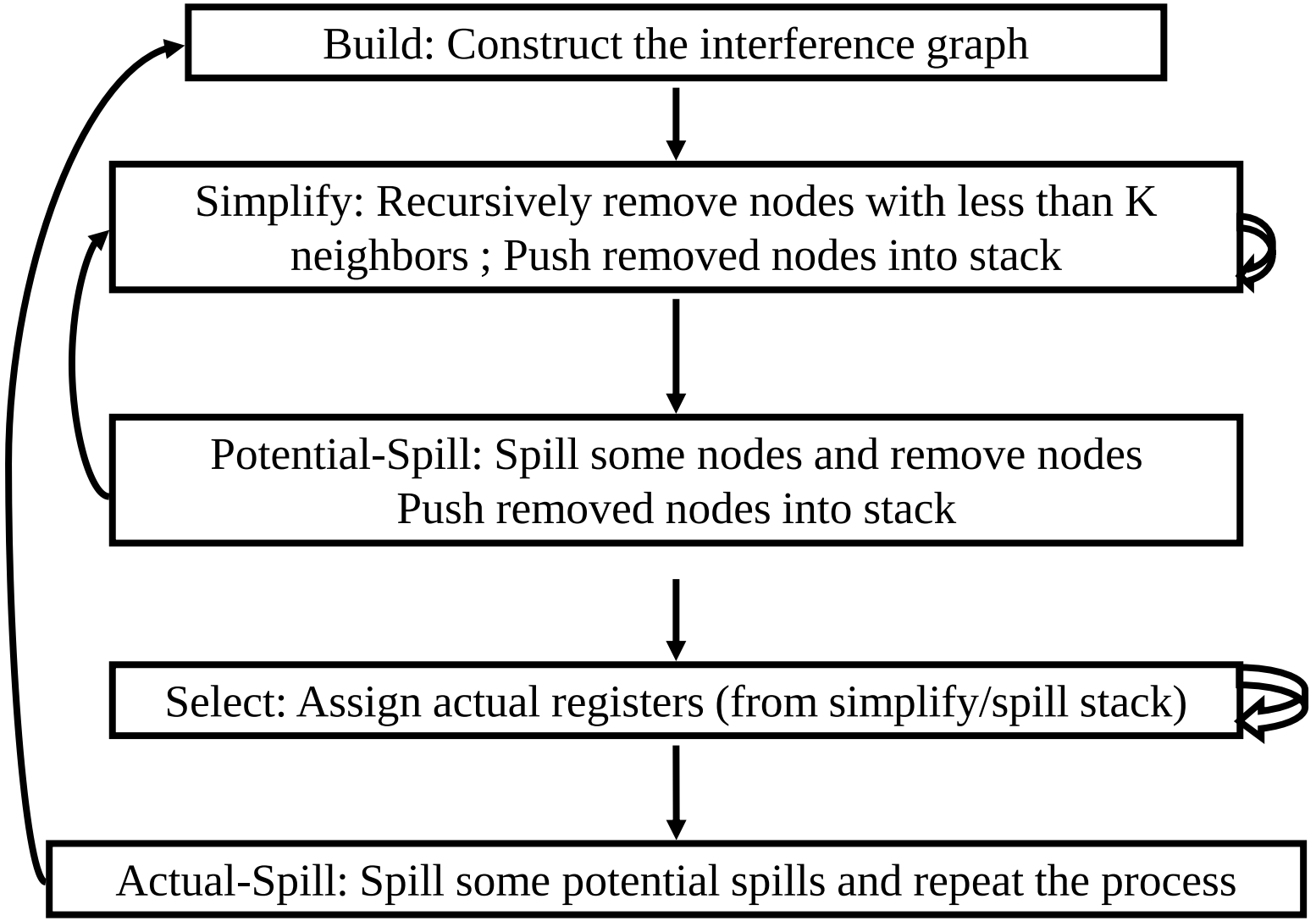
Computing Liveness via DFS(6)



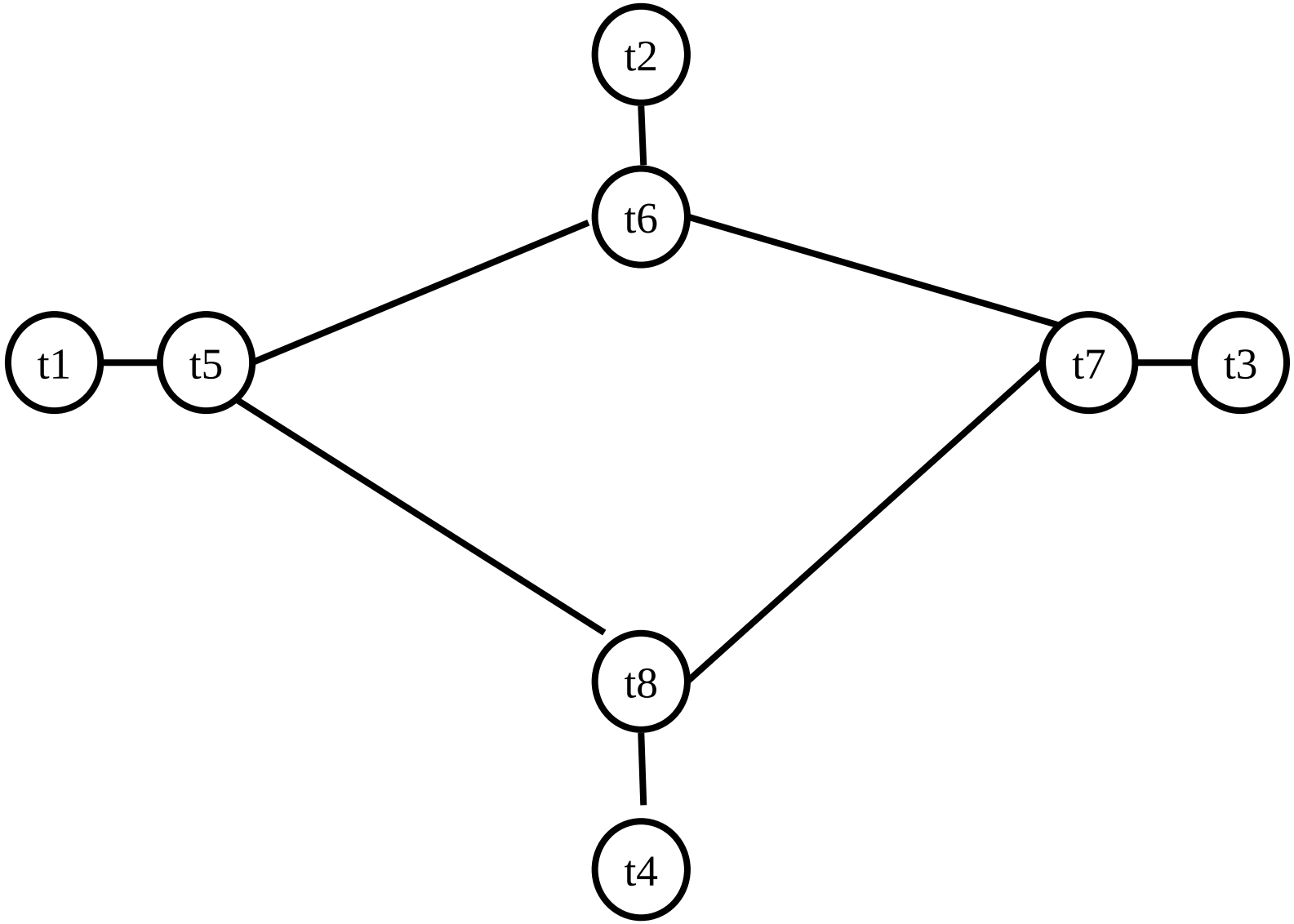
Coloring by Simplification [Kempe 1879]

- K
  - the number of machine registers
- G(V, E)
  - the interference graph
- Consider a node v $\in$V with less than K neighbors:
  - Color G – v in K colors
  - Color v in a color different than its (colored) neighbors

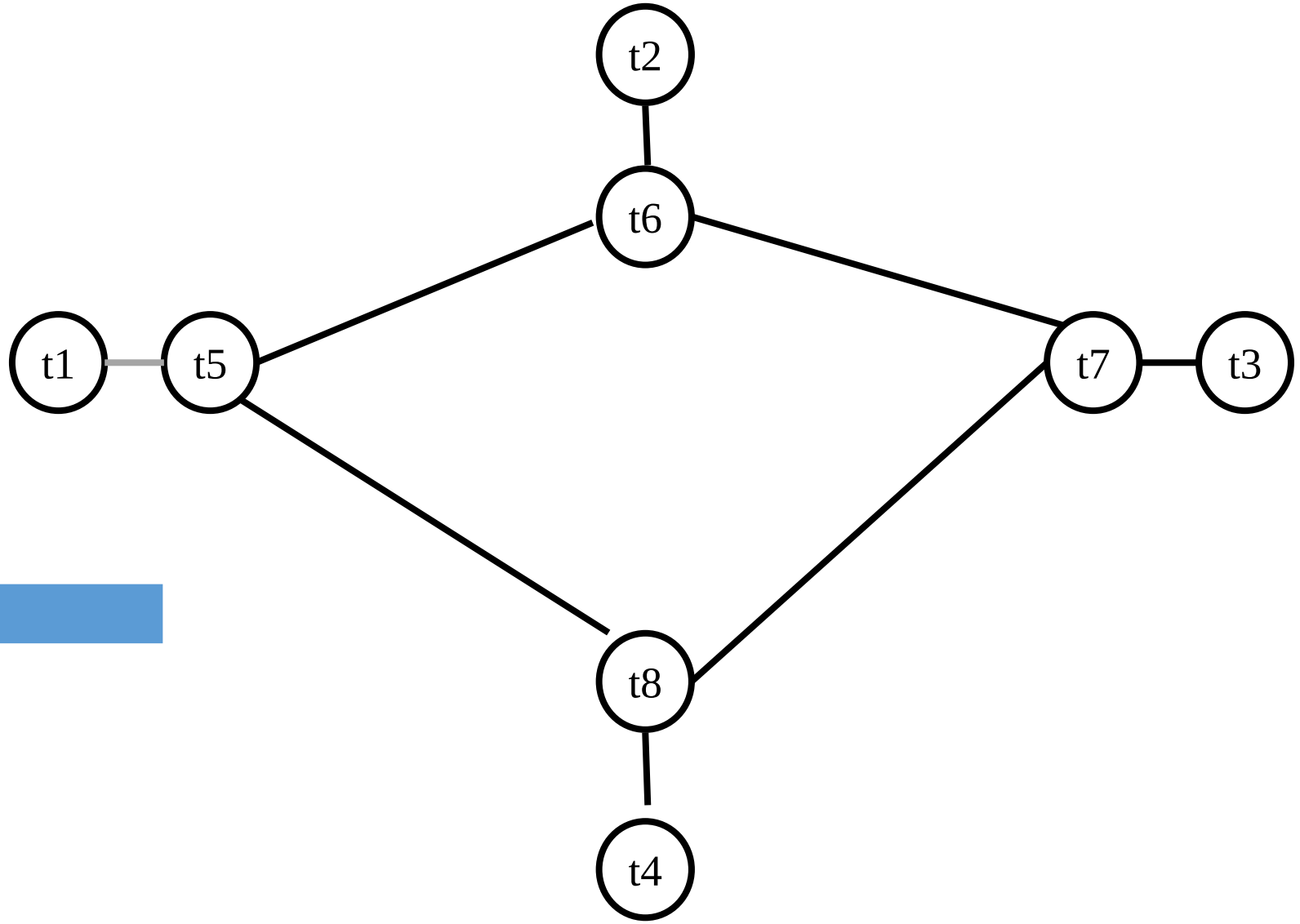
Graph Coloring by Simplification



Artificial Example K=2

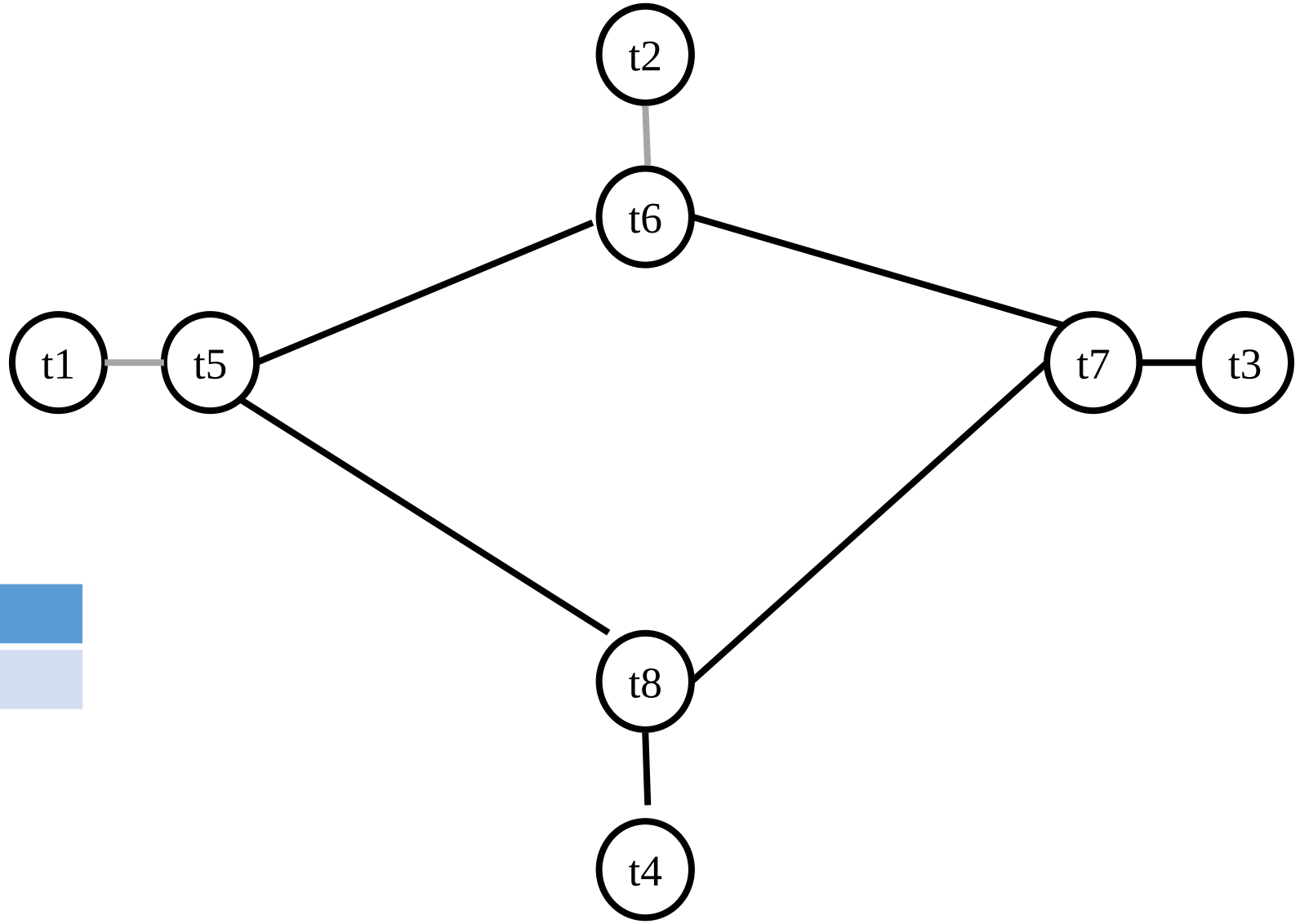


Artificial Example K=2



t1

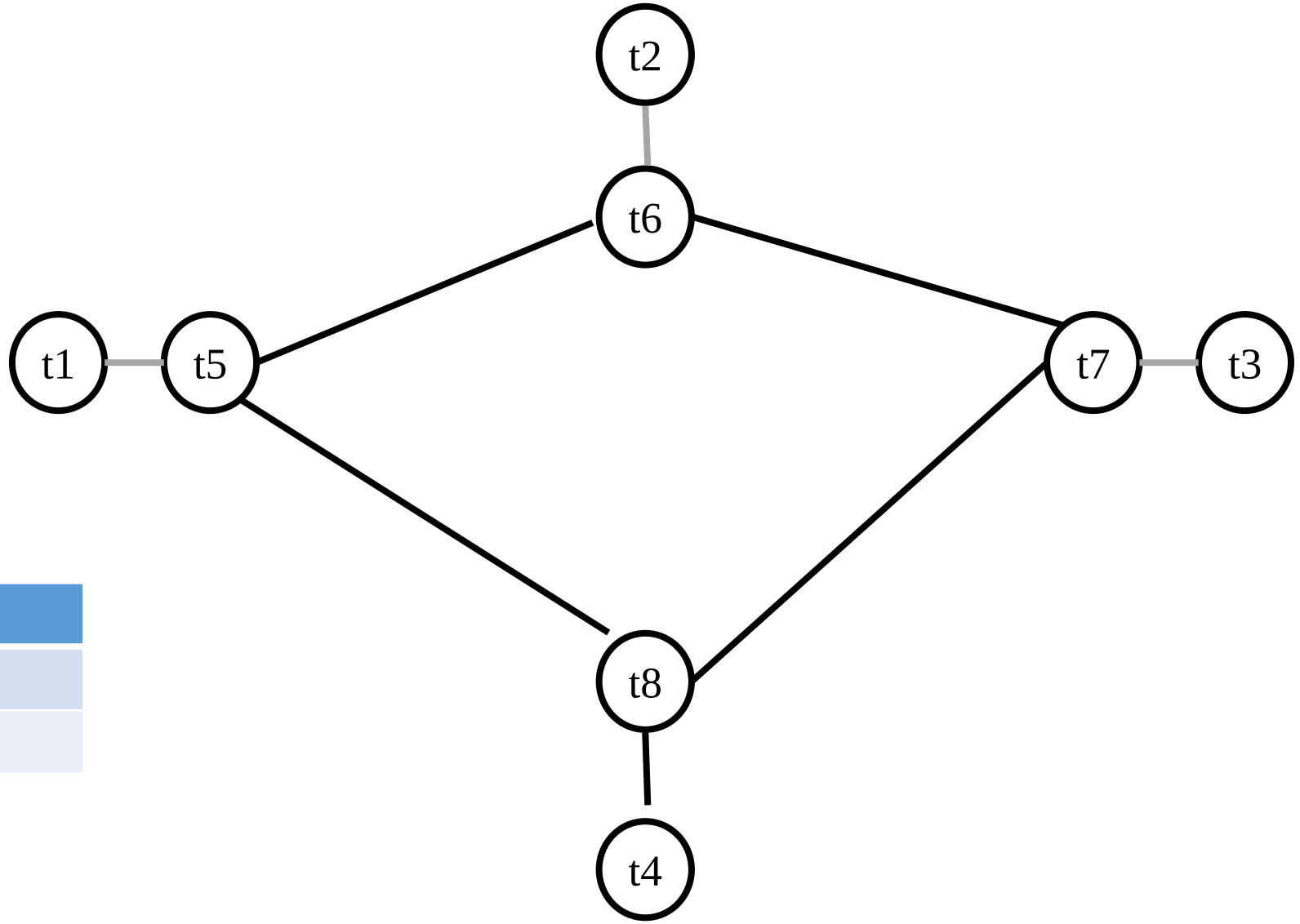
Artificial Example K=2



t2

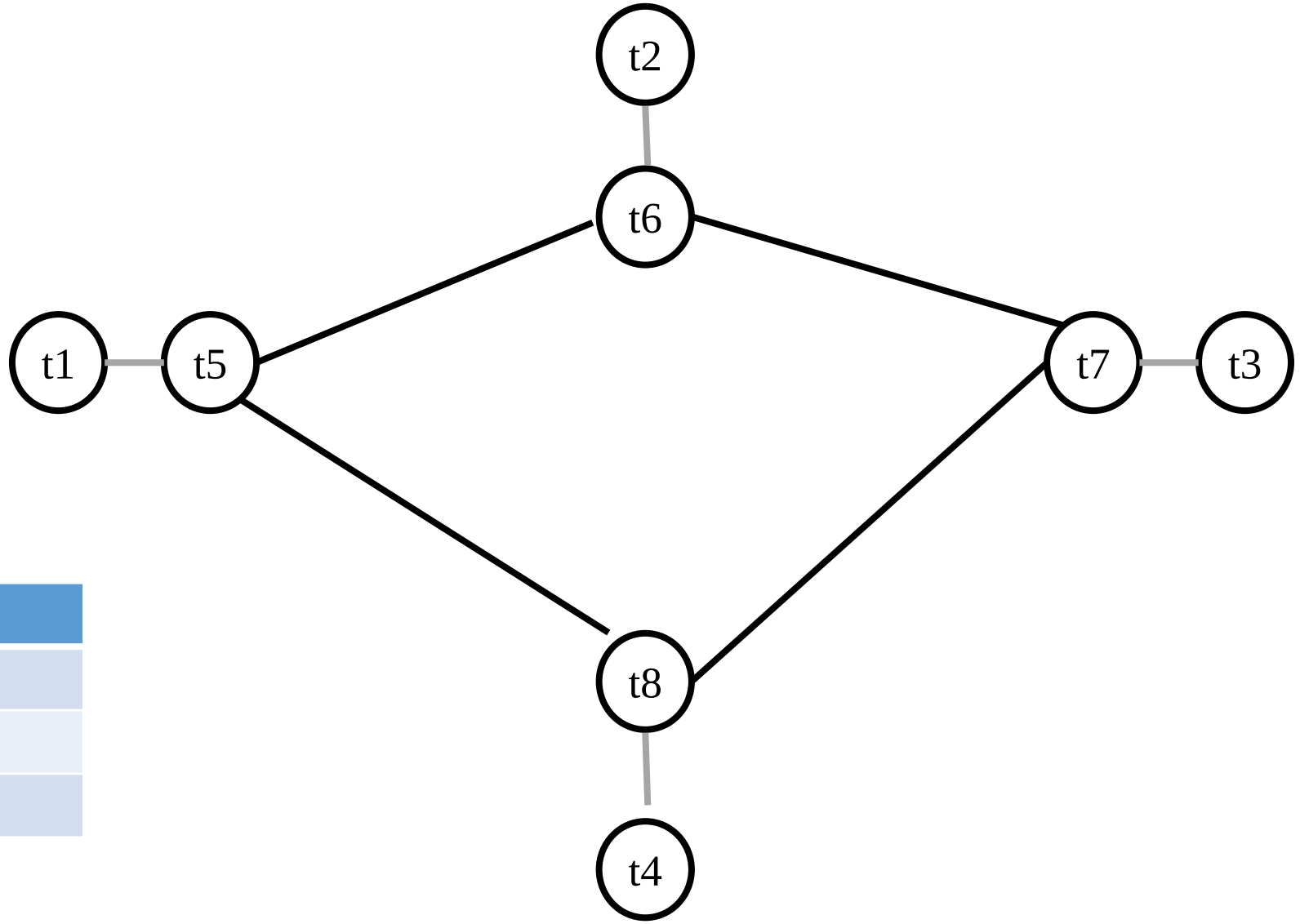
t1

Artificial Example K=2



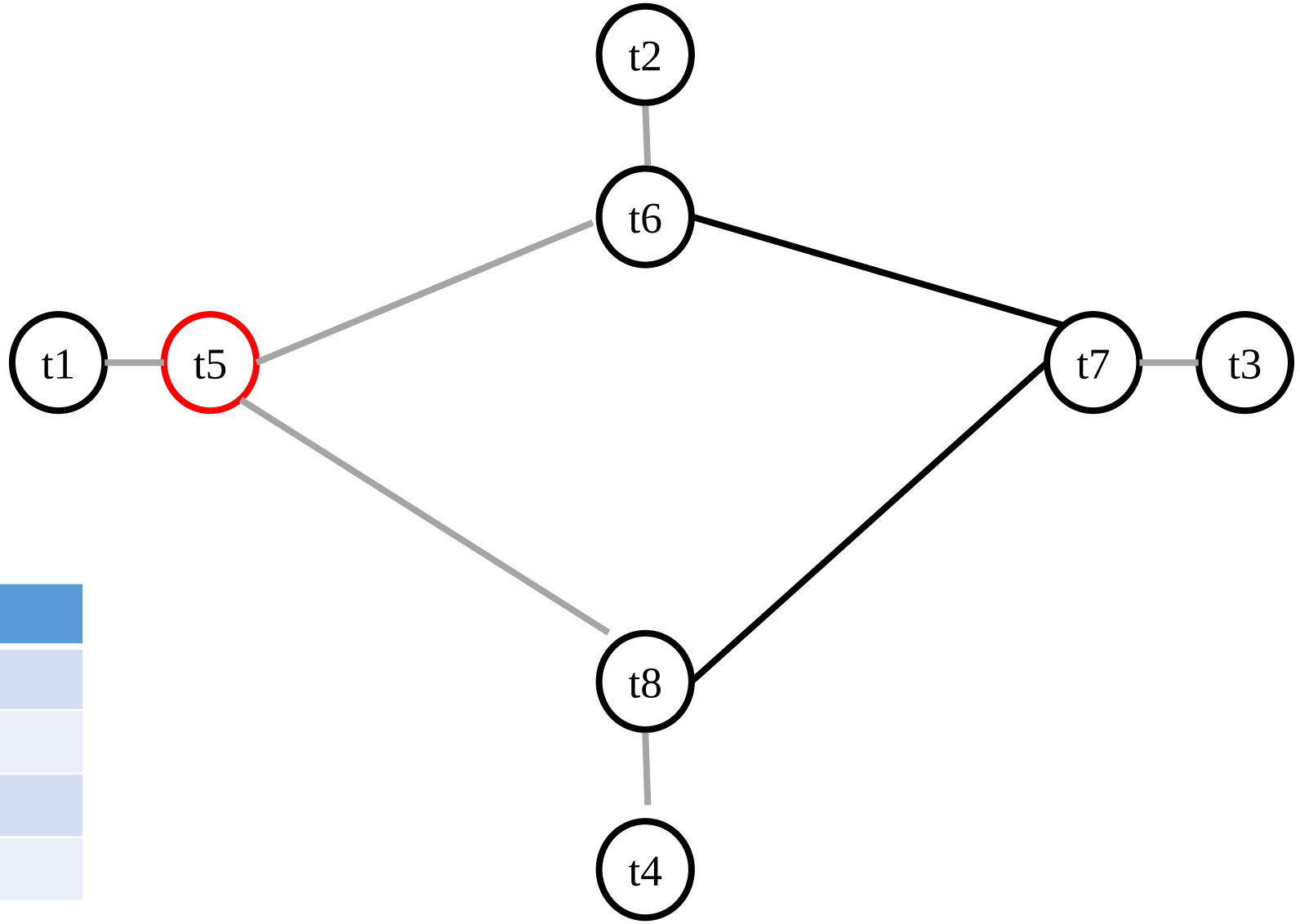
t3
t2
t1

Artificial Example K=2



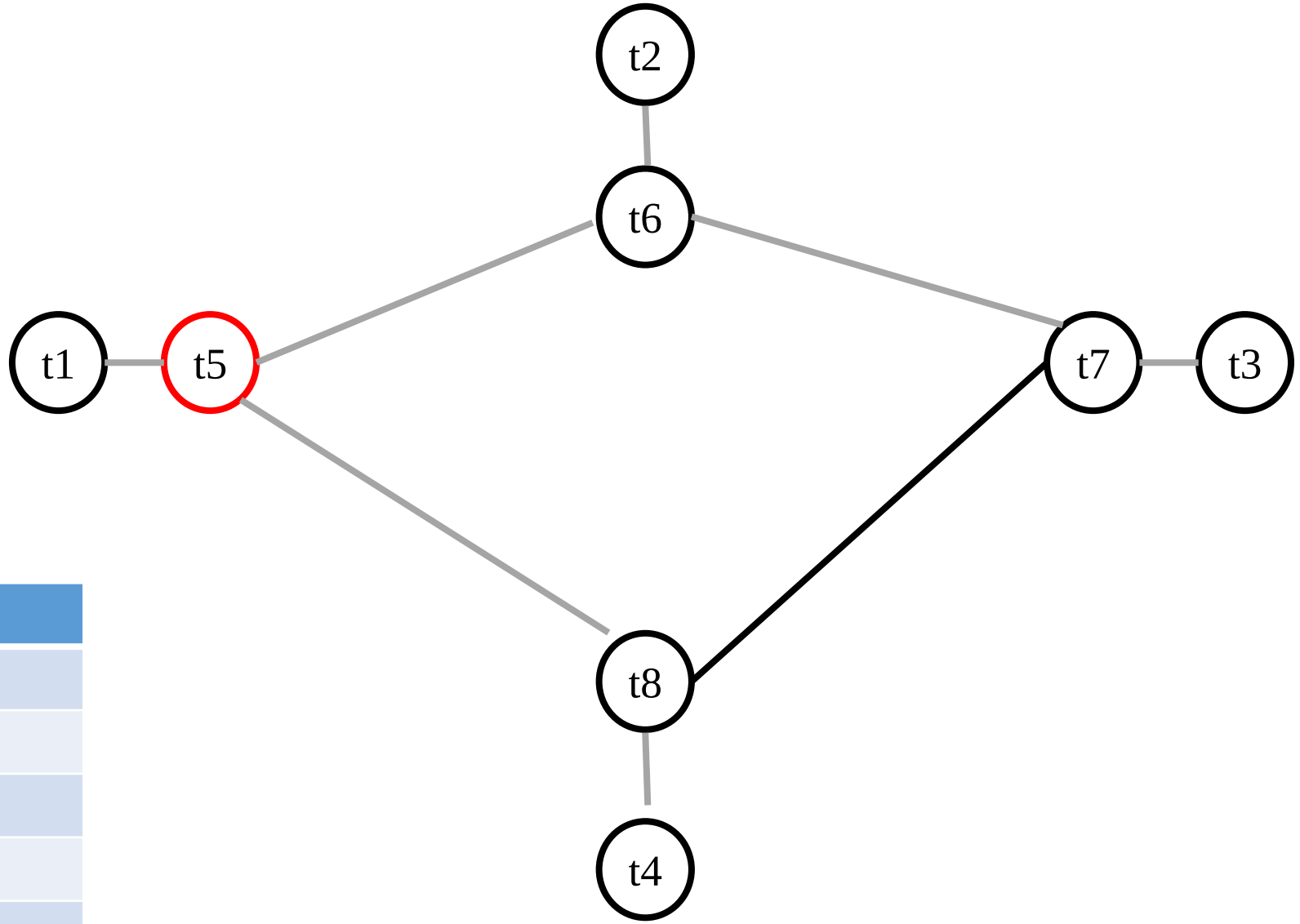
t4
t3
t2
t1

Artificial Example K=2



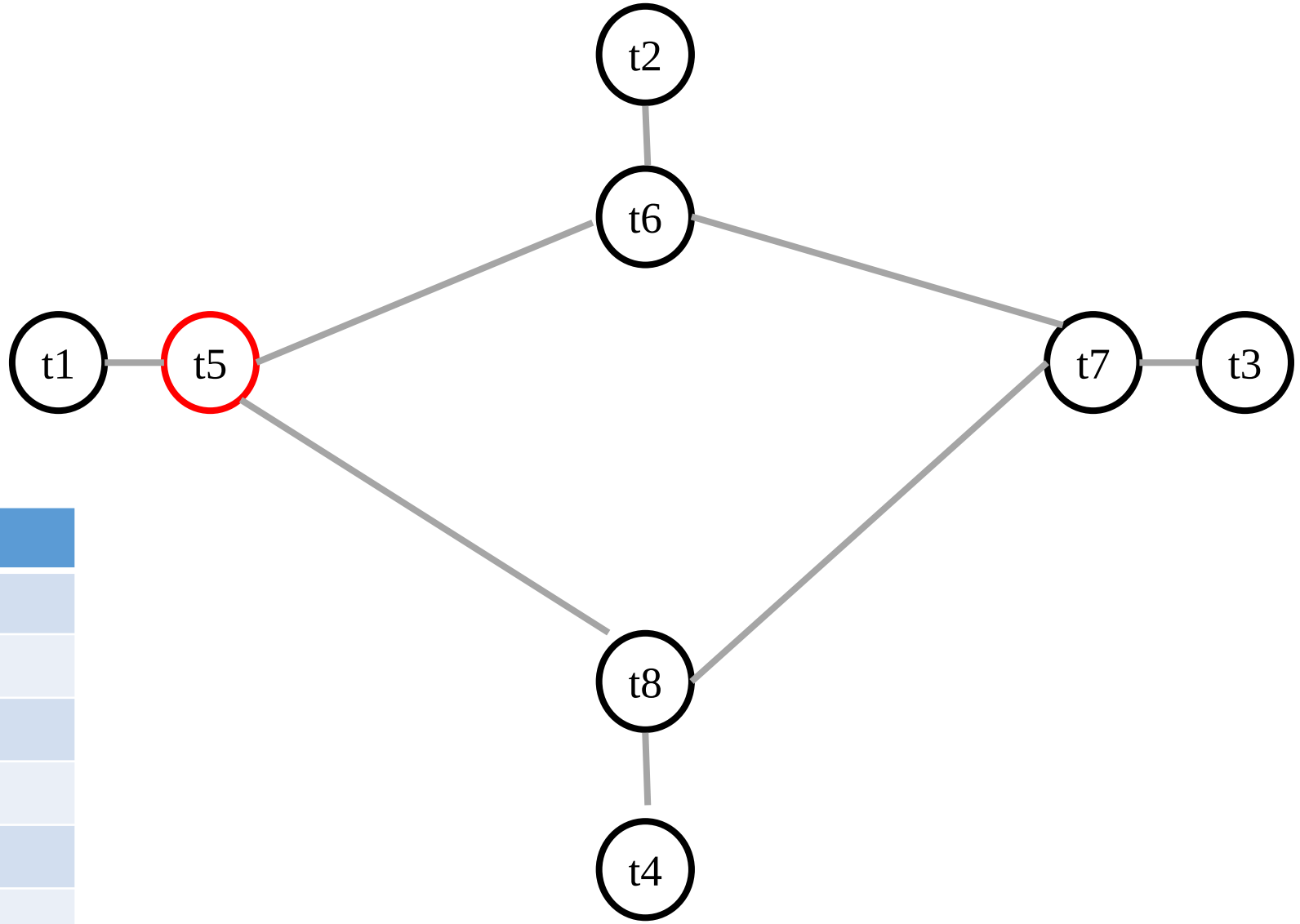
t5
t4
t3
t2
t1

Artificial Example K=2



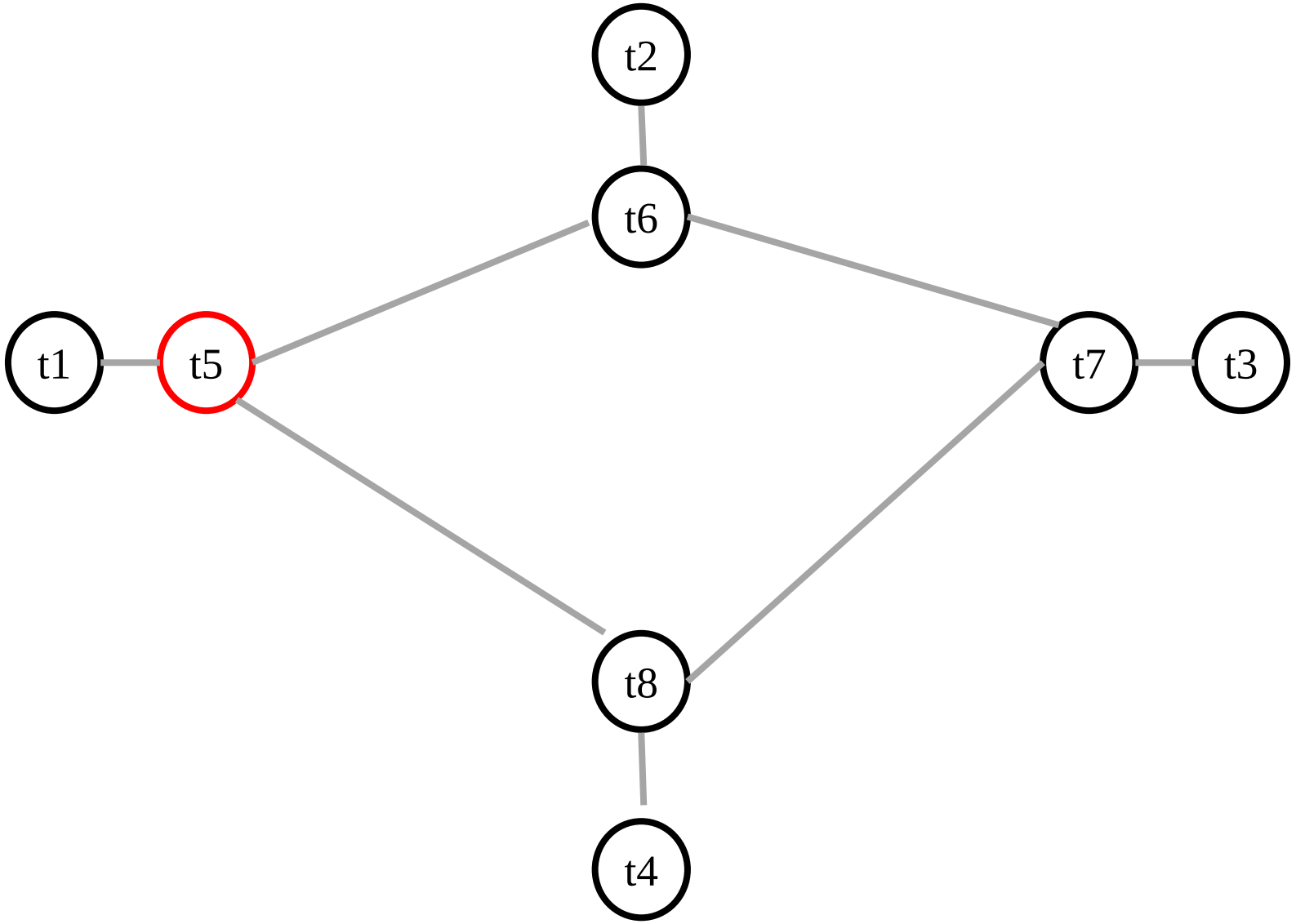
t6
t5
t4
t3
t2
t1

Artificial Example K=2



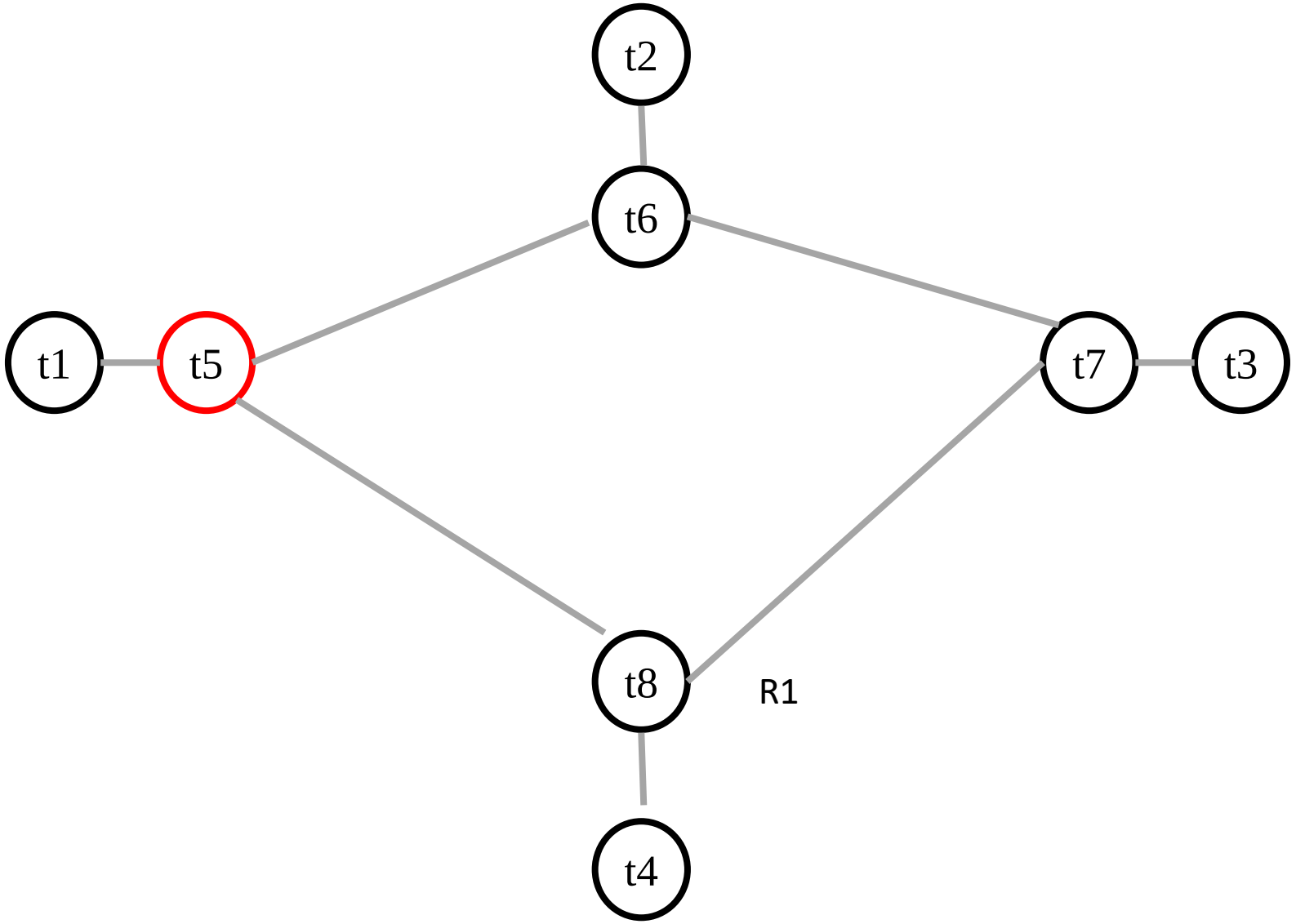
t7
t6
t5
t4
t3
t2
t1

Artificial Example K=2



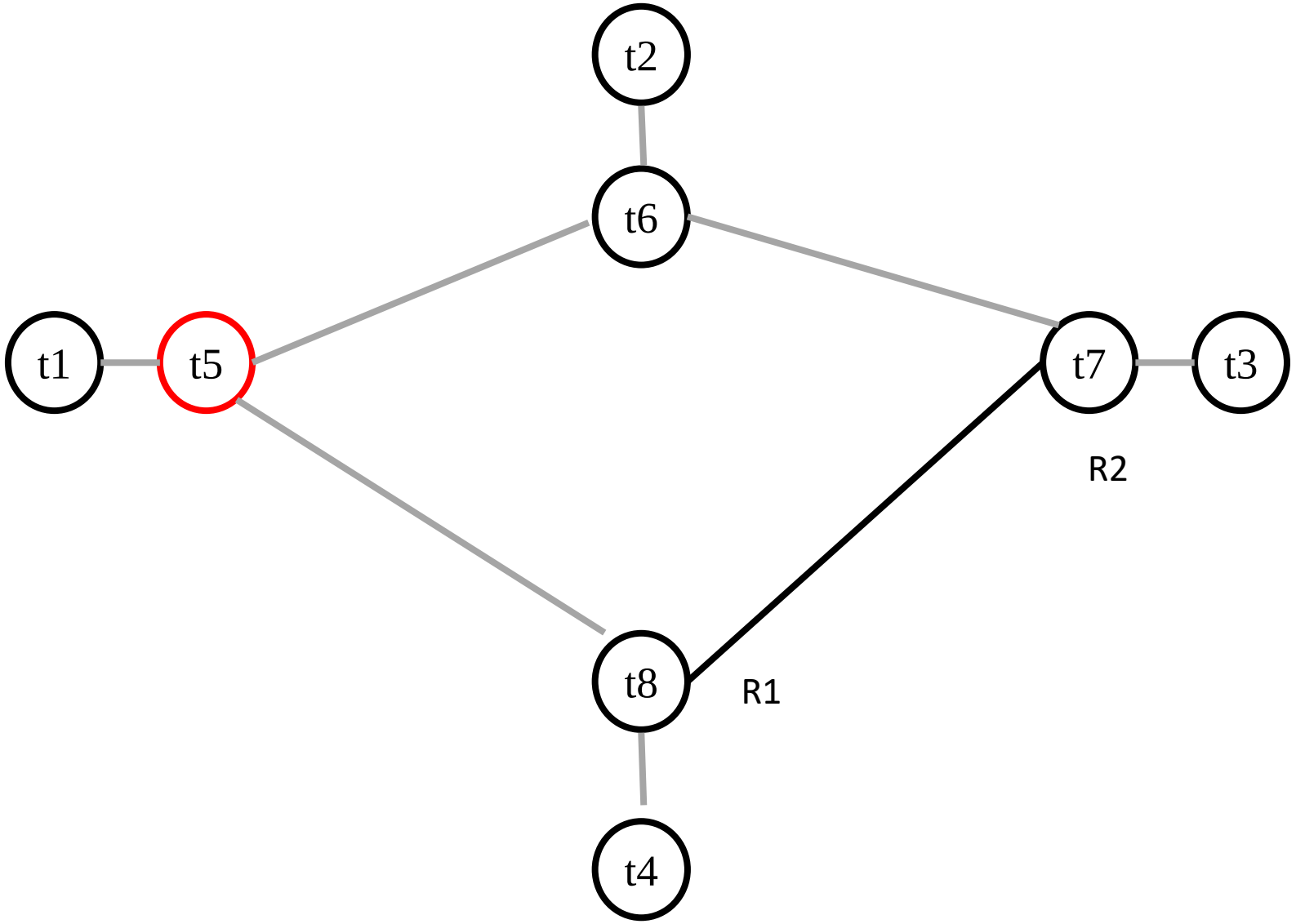
t8
t7
t6
t5
t4
t3
t2
t1

Artificial Example K=2



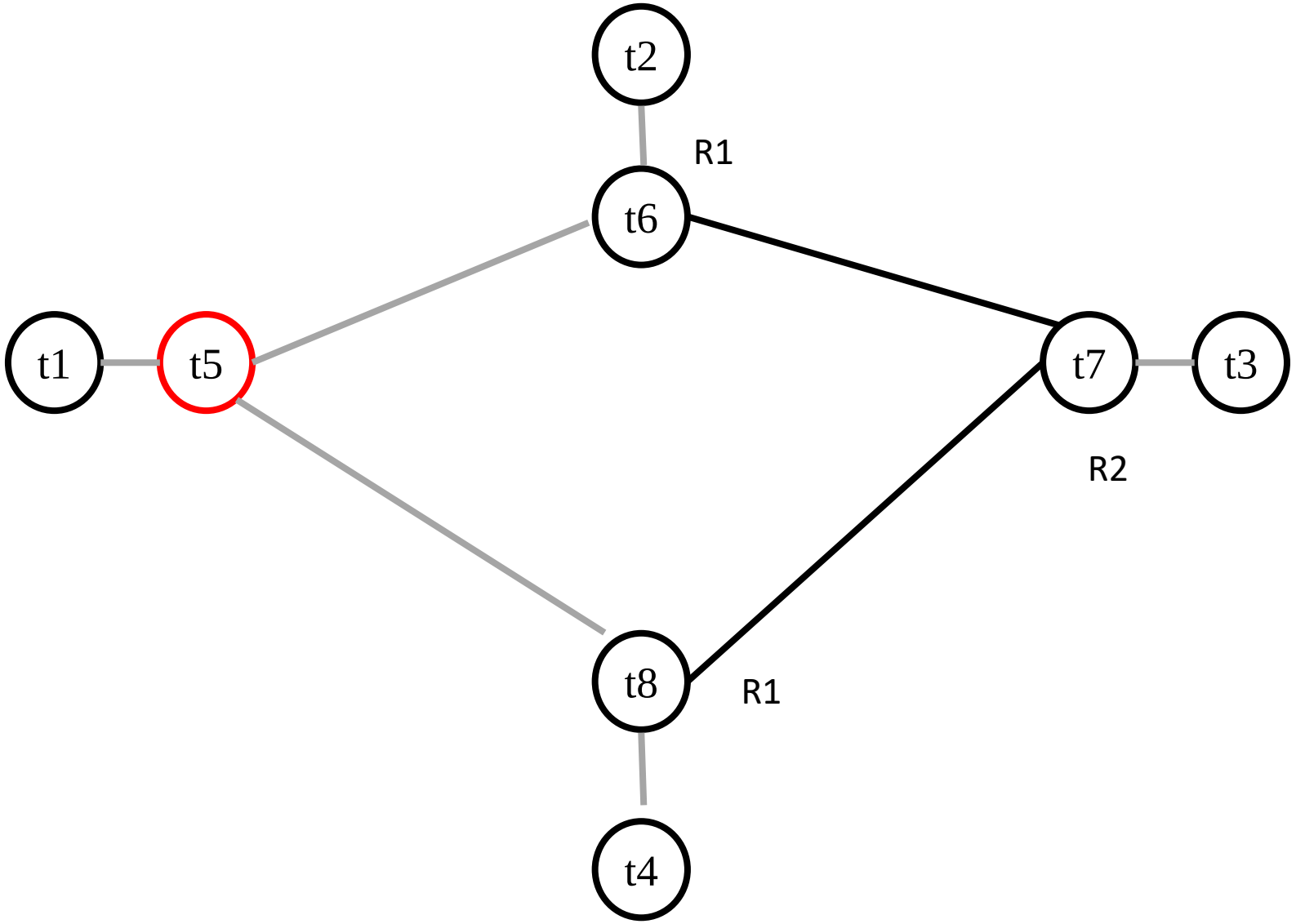
t7
t6
t5
t4
t3
t2
t1

Artificial Example K=2



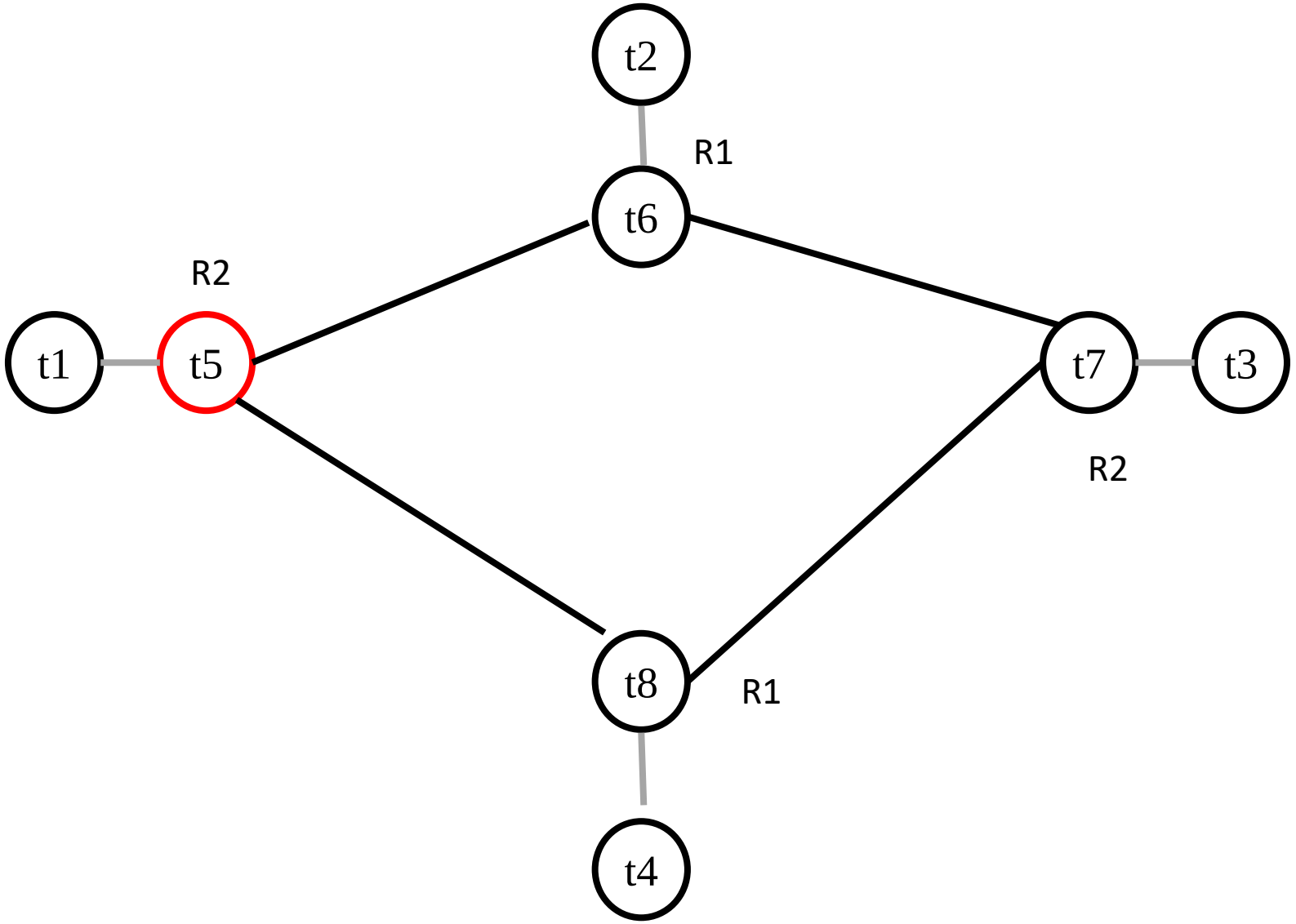
t6
t5
t4
t3
t2
t1

Artificial Example K=2



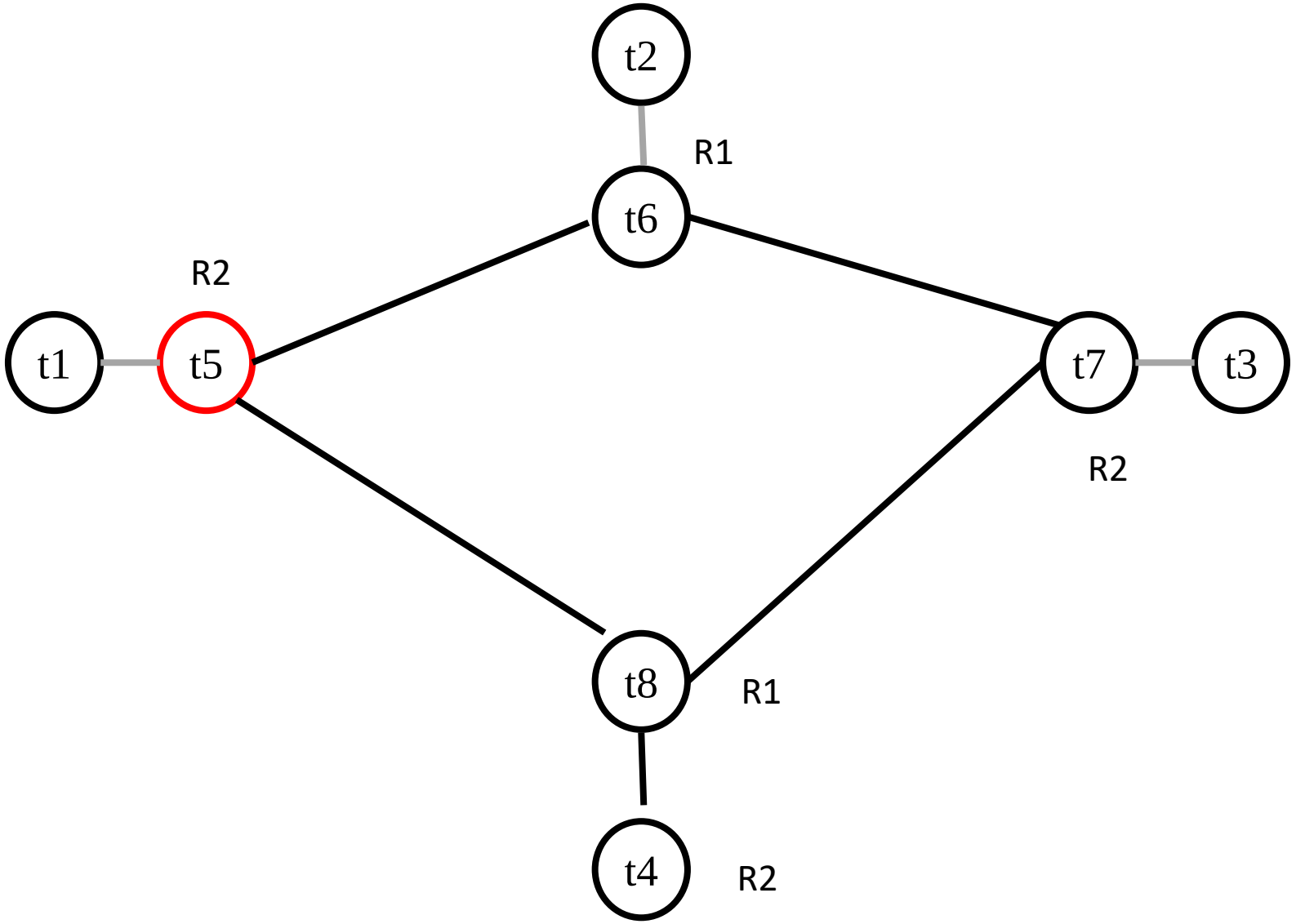
t5
t4
t3
t2
t1

Artificial Example K=2



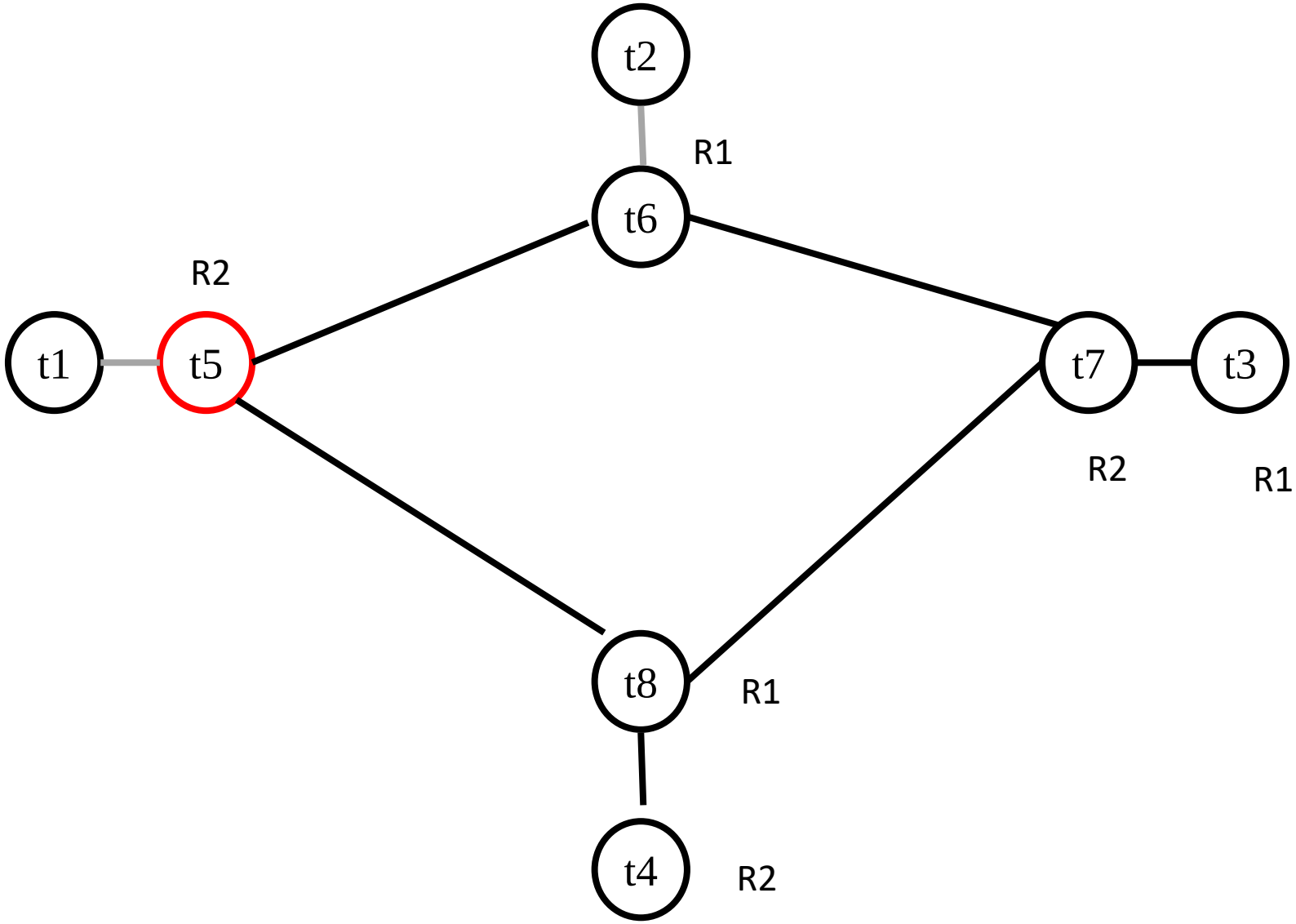
t4
t3
t2
t1

Artificial Example K=2



t3
t2
t1

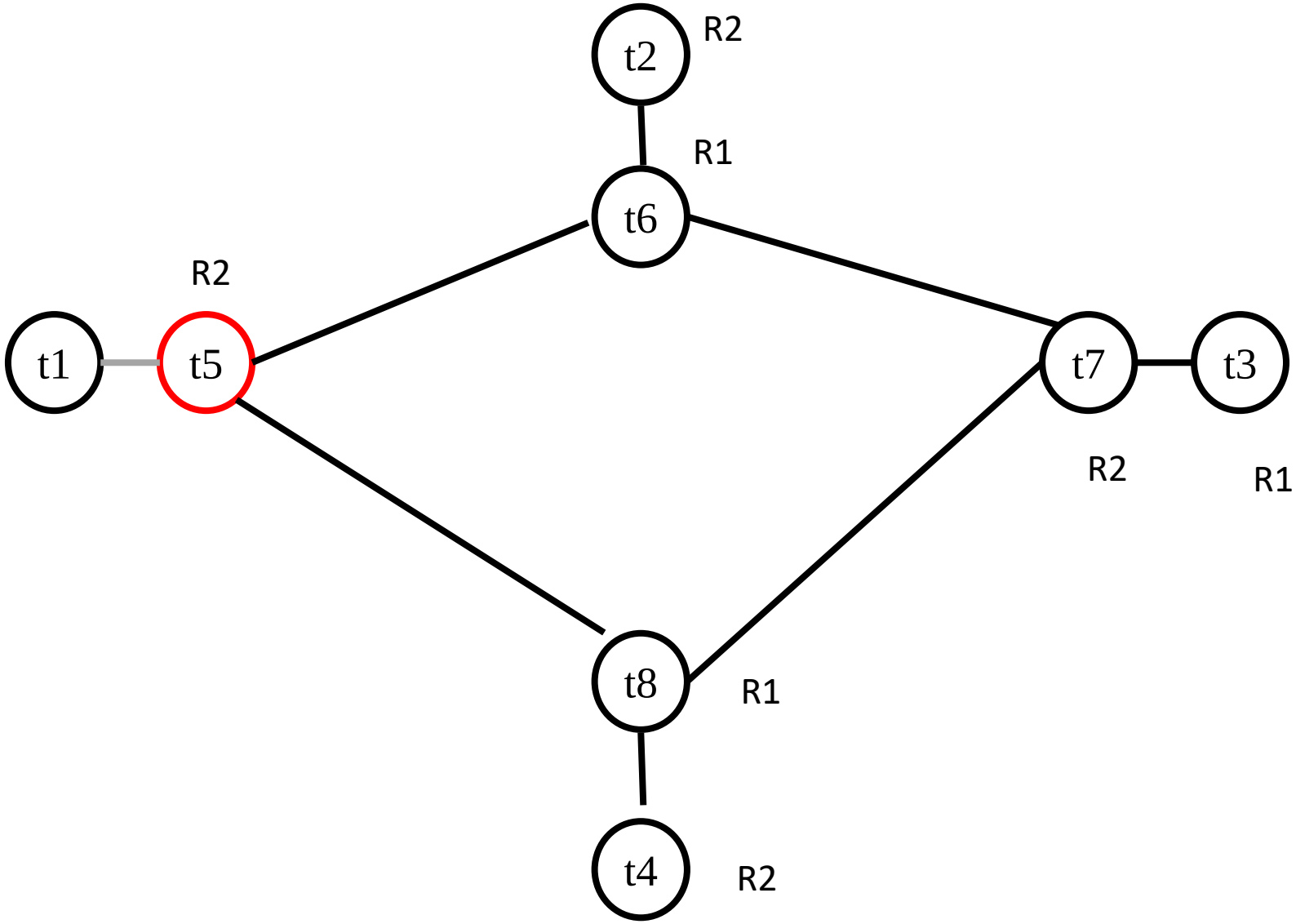
Artificial Example K=2



t2

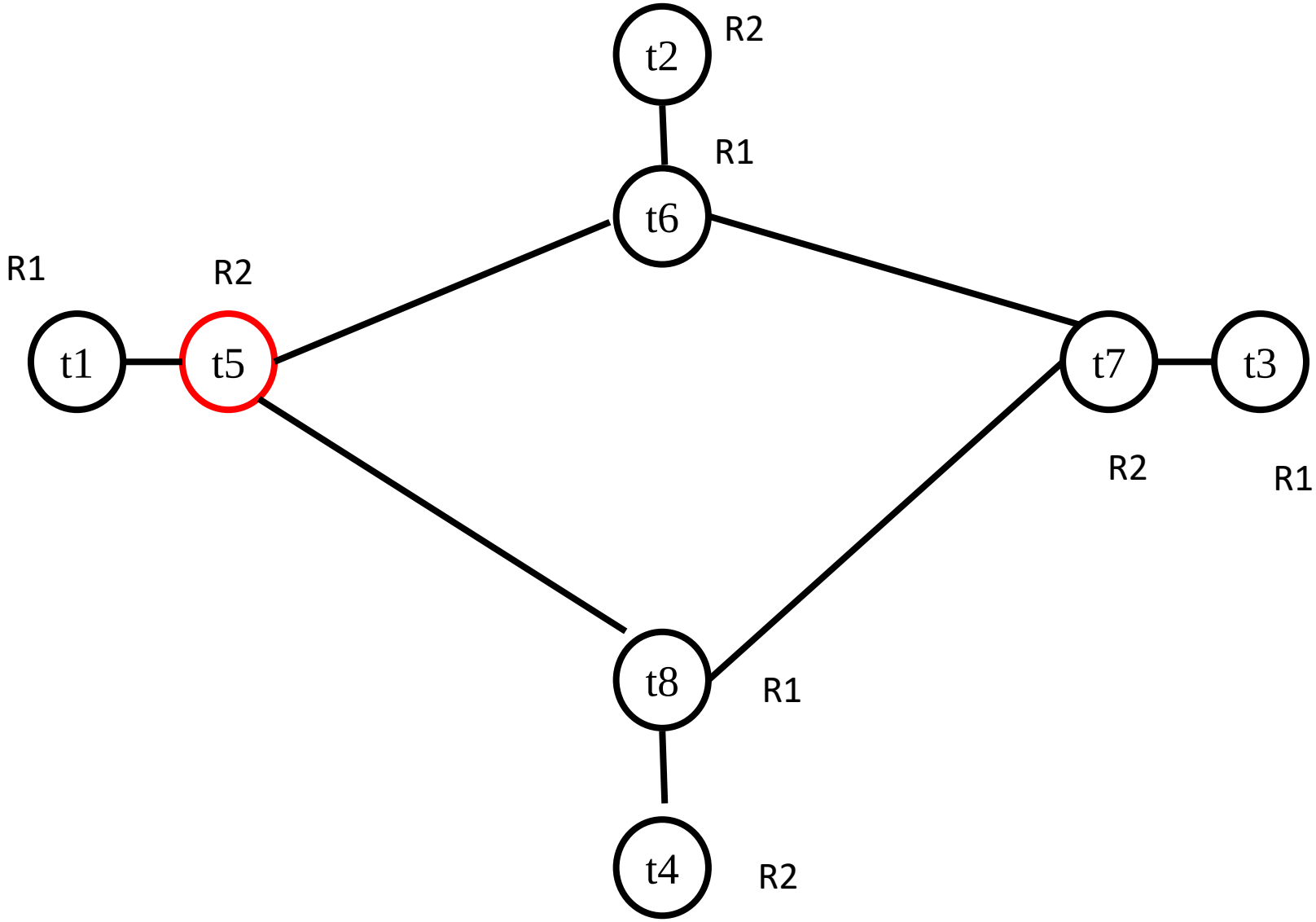
t1

Artificial Example K=2



t1

Artificial Example K=2



Extensions

- Spilling heuristics
- MOV nodes
- Caller-/Callee Save Registers

Summary

- Intermediate Languages simplifies compilation
- Two related problems:
  - Instruction selection
  - Register allocation
- LLVM provides a reusable software platform to implement both