Compacting WebAssembly

Andreas Wälchli
Supervisor: Olivier Flückiger

SGC Seminar Project
21.05.2019

What is WebAssembly?

- binary instruction format
- stack-based VM

- runs client-side in/near JS-VM at near native speed
- integrates with JS
  - call WASM function from JS and vice-versa

- many more uses coming up

What is WebAssembly?

WEBASSEMBLY

- universal compilation target
  - C, C++, Go, Java, Javascript, Kotlin, Lua, Perl, PHP, Prolog, Ruby, Rust, Swift, …
- efficient and fast
  - single-pass parsing and validation with zero lookahead
  - only 4 types (i32, i64, f32, f64)
- text format based on S-expressions

Code Example

Java

```
int f(int x, int y) {
  return x * (x – y);
}
```

.wat

```
(func $f
  (param $x i32)
  (param $y i32)
  (result i32)
  (return
    (i32.mul
      (get_local $x)
      (i32.sub
        (get_local $x)
        (get_local $y)))))
```

.wasm

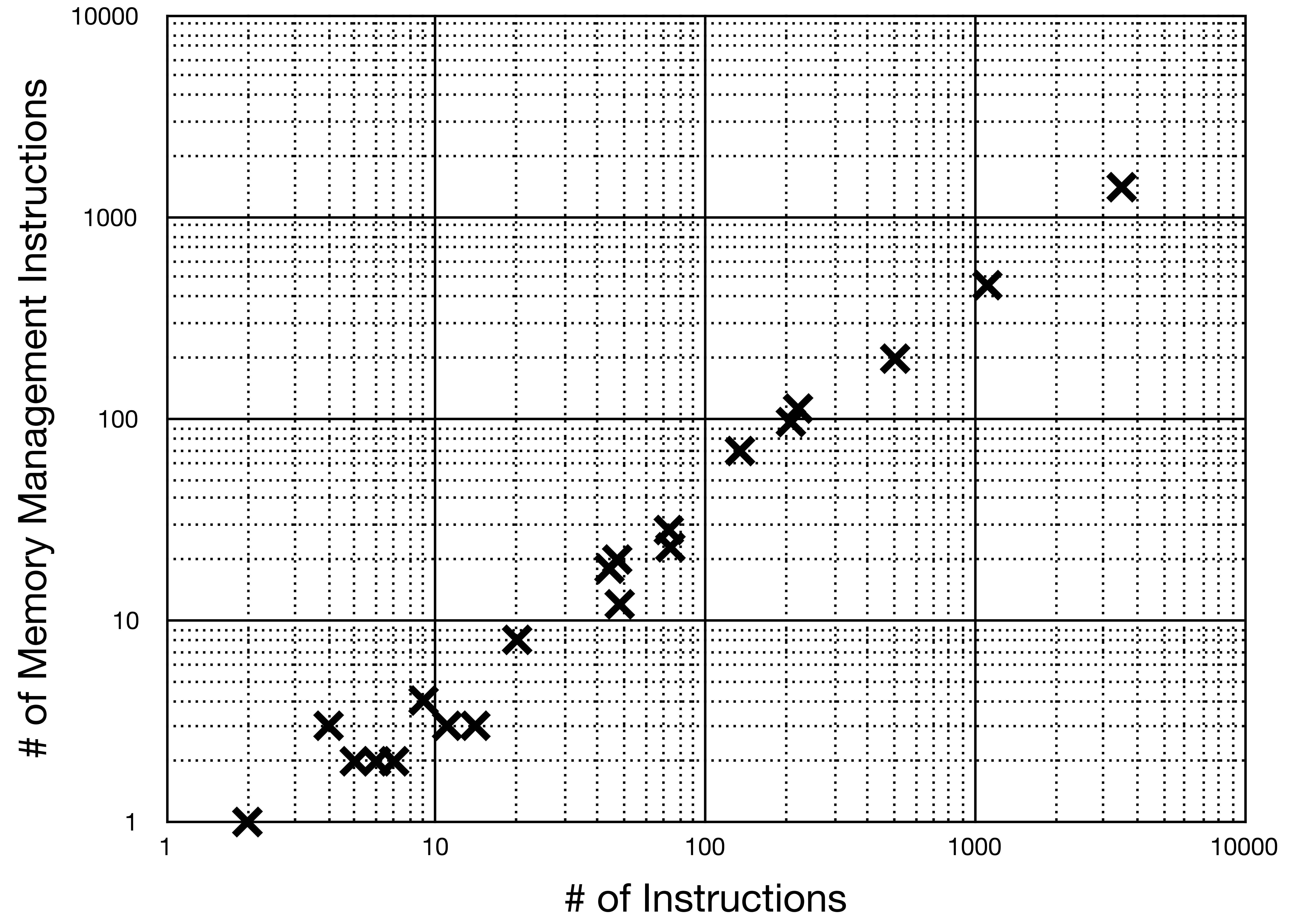
```
0x60          function signature
0x02          2 parameters
0x7f 0x7f     i32, i32
0x01          1 return type
0x7f          i32
...
0x0b          length: 11 bytes
0x00          no local variables
0x20 0x00     local.get 0
0x20 0x00     local.get 0
0x20 0x01     local.get 1
0x6b          i32.sub
0x6c          i32.mul
0x0f          return
0x0b          end
```

Motivation

- WASM is a stack machine
- source language may not be stack-based
- a lot of memory management code in binary (~40%)

Question: Can we reduce the amount of memory management?

→ reduce # of instructions, binary size


10000
1000
100
10
1
1
10
100
1000
10000
# of Memory Management Instructions
# of Instructions

Approaches

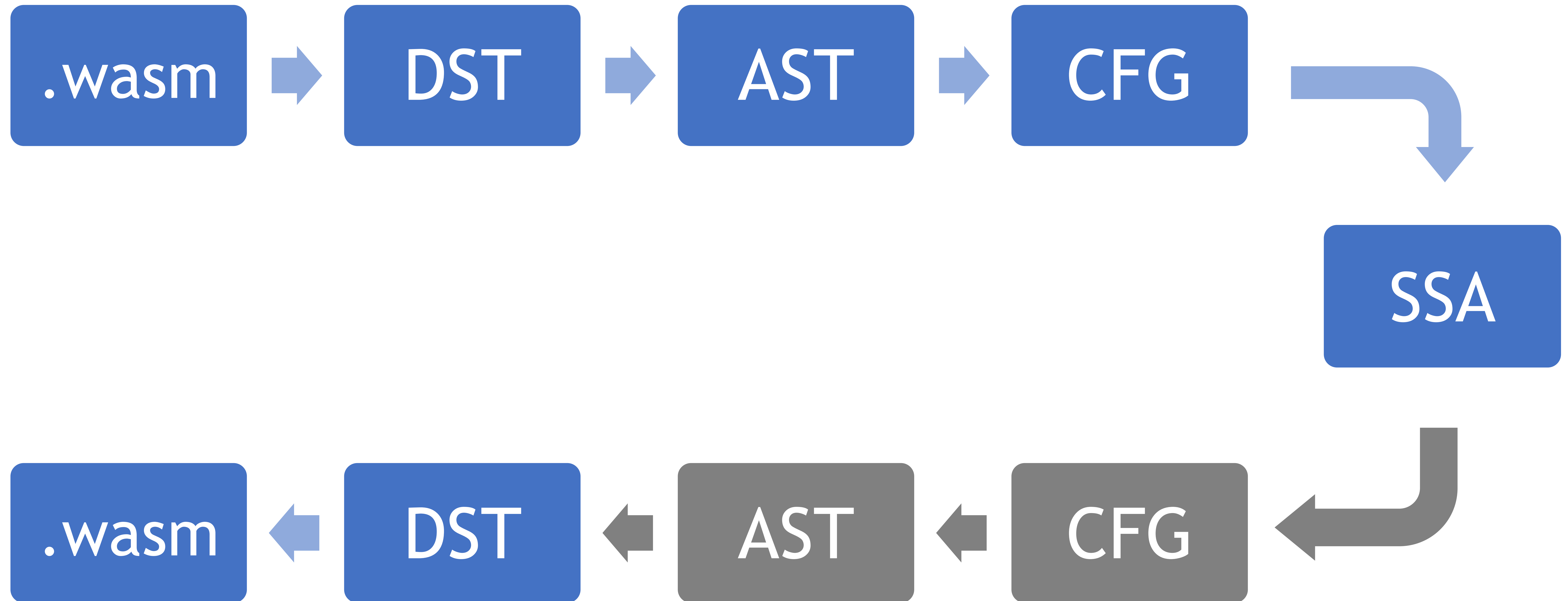
1. Improve *emscripten* (LLVM →asm.js/WASM compiler)

2. Implement stand-alone optimiser
   - applicable to pre-existing binaries
   - evaluate different WASM compilers

Constraints

- restricted stack manipulation
  - drop/pop, ~~dup/copy, swap/exch, pick/index, pull/roll~~
- classical optimisation algorithms not applicable
  - e.g. Koopman, 1992
- optimise through instruction reordering only!

Setup



AST to CFG

- produce a control-flow graph from the AST any function
- create a block for each control-flow instruction
  - *block*, *loop*, *ifelse*

- link blocks together according to implicit branching and explicit branching instructions
  - *br*, *br_if*, *br_table*, *return*, *end*

- sanitise graph

Example

Java

```
int f(int x, int y) {
  int temp = 0;
  if(y < x) {
    temp = x - y;
  } else {
    temp = y - x;
  }
  return 2 * temp;
}
```

.wasm

```
local.get 0
local.get 1
i32.lte_s
if [i32]
  local.get 0
  local.get 1
  i32.sub
else
  local.get 1
  local.get 0
  i32.sub
end
i32.const 2
i32.mul
return
end
```

Example

.wasm

```
local.get 0
local.get 1
i32.lte_s
if [i32]
  local.get 0
  local.get 1
  i32.sub
else
  local.get 1
  local.get 0
  i32.sub
end
i32.const 2
i32.mul
return
end
```

Example

.wasm

```
local.get 0
local.get 1
i32.lte_s
if [i32]
  local.get 0
  local.get 1
  i32.sub
else
  local.get 1
  local.get 0
  i32.sub
end
i32.const 2
i32.mul
return
end
```

Example

.wasm

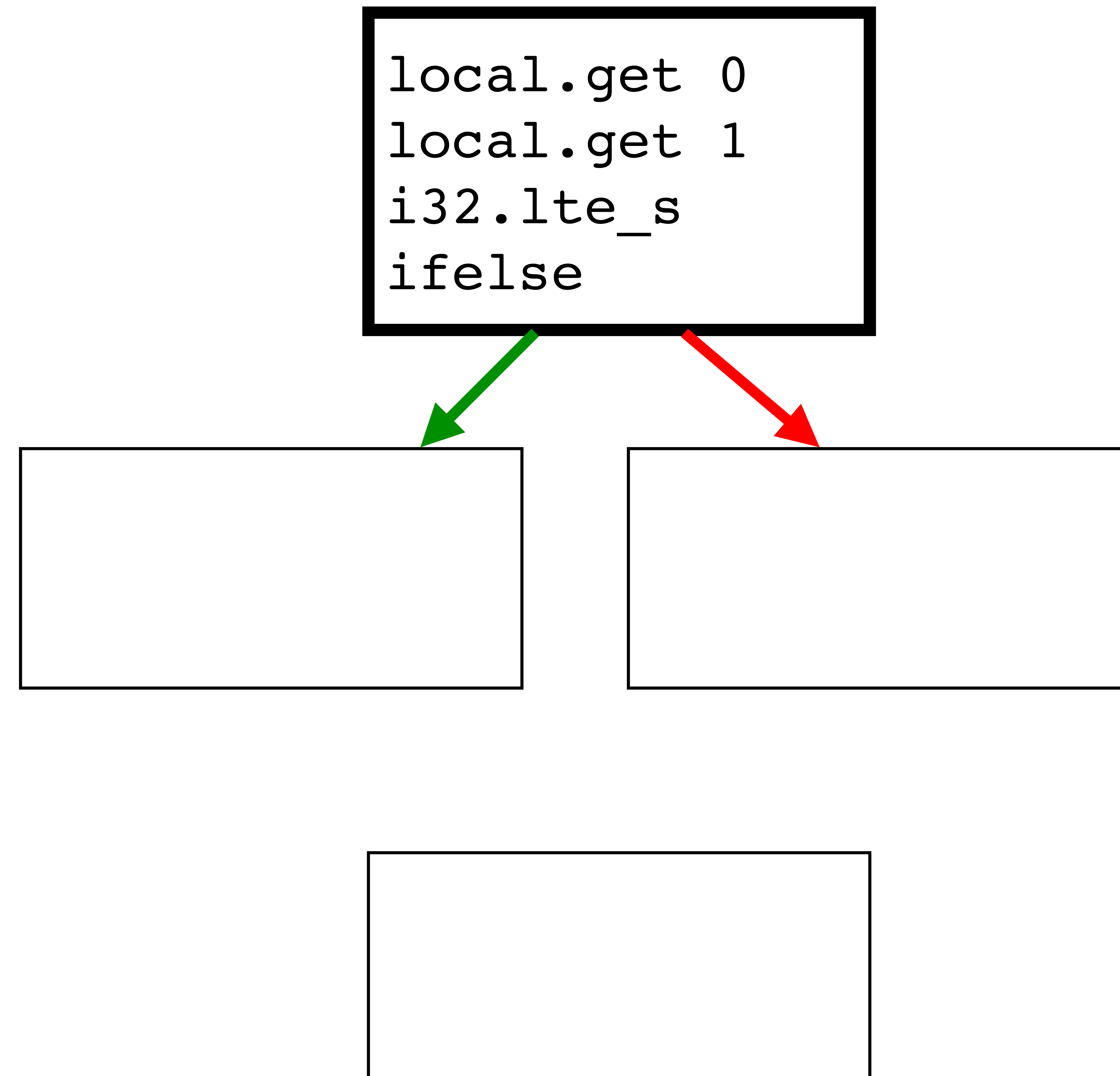
```
local.get 0
local.get 1
i32.lte_s
if [i32]
  local.get 0
  local.get 1
  i32.sub
else
  local.get 1
  local.get 0
  i32.sub
end
i32.const 2
i32.mul
return
end
```

```
local.get 0
local.get 1
i32.lte_s
```

Example

.wasm

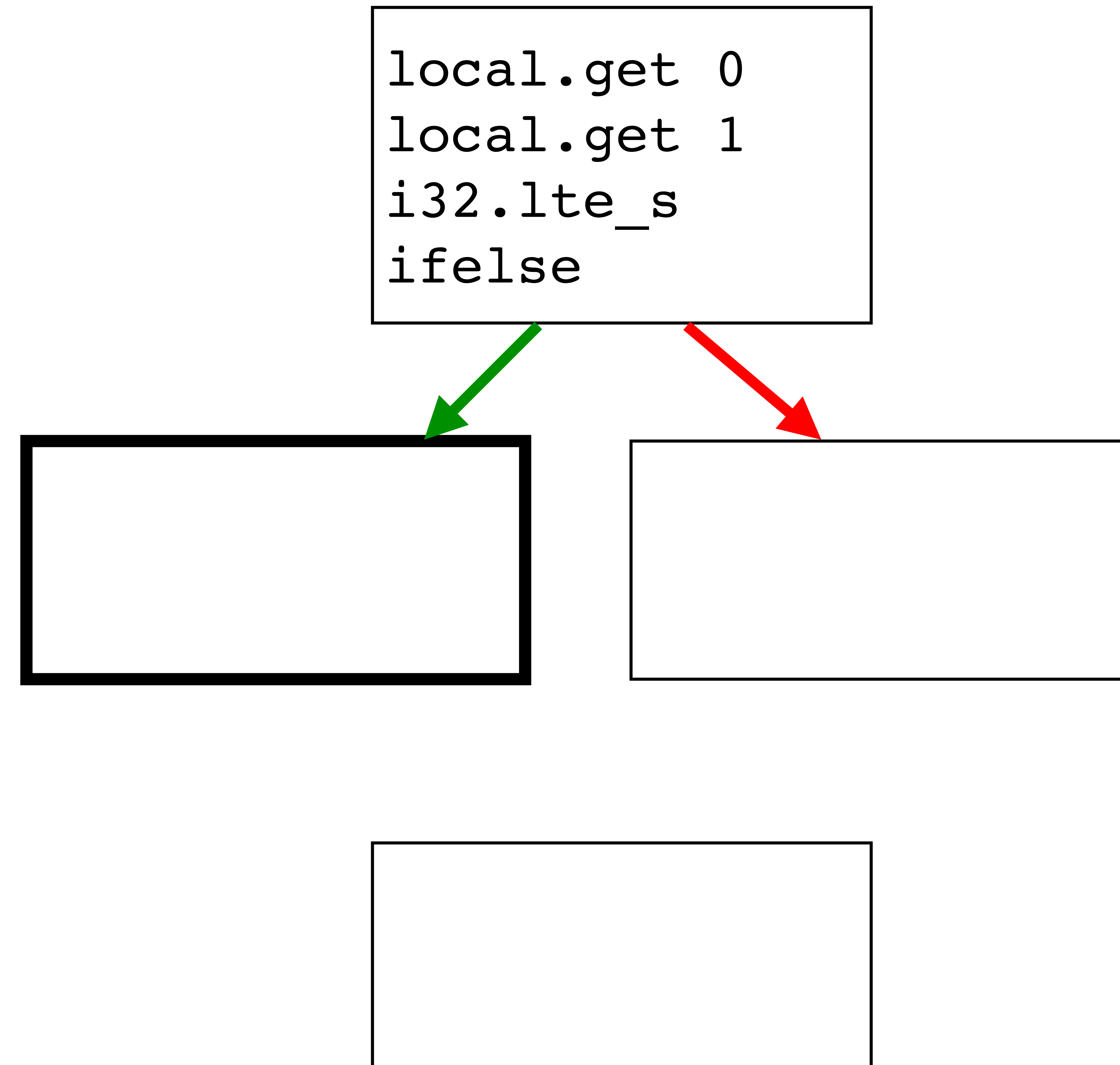
```
local.get 0
local.get 1
i32.lte_s
if [i32]
  local.get 0
  local.get 1
  i32.sub
else
  local.get 1
  local.get 0
  i32.sub
end
i32.const 2
i32.mul
return
end
```



Example

.wasm

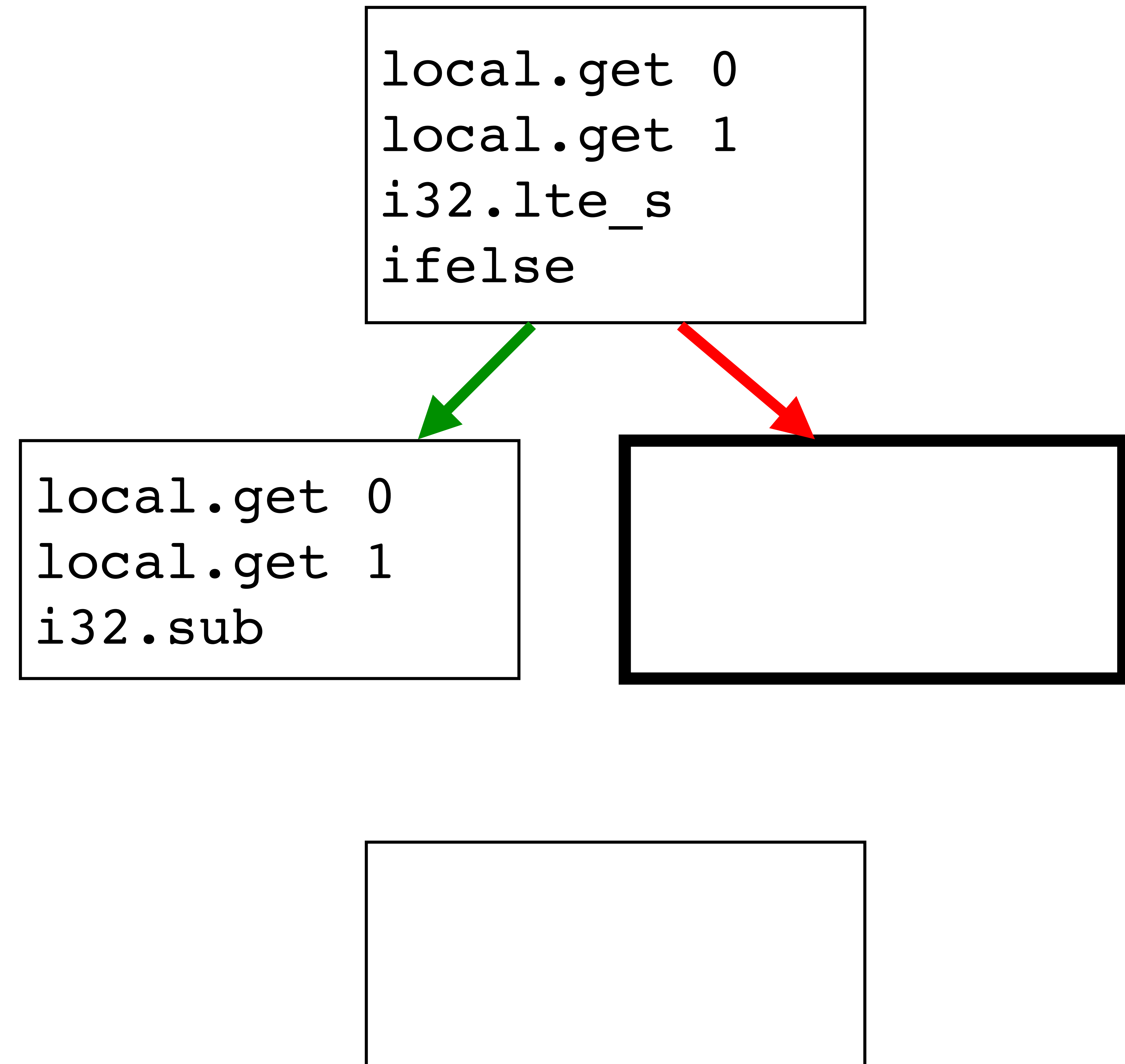
```
local.get 0
local.get 1
i32.lte_s
if [i32]
  local.get 0
  local.get 1
  i32.sub
else
  local.get 1
  local.get 0
  i32.sub
end
i32.const 2
i32.mul
return
end
```



Example

.wasm

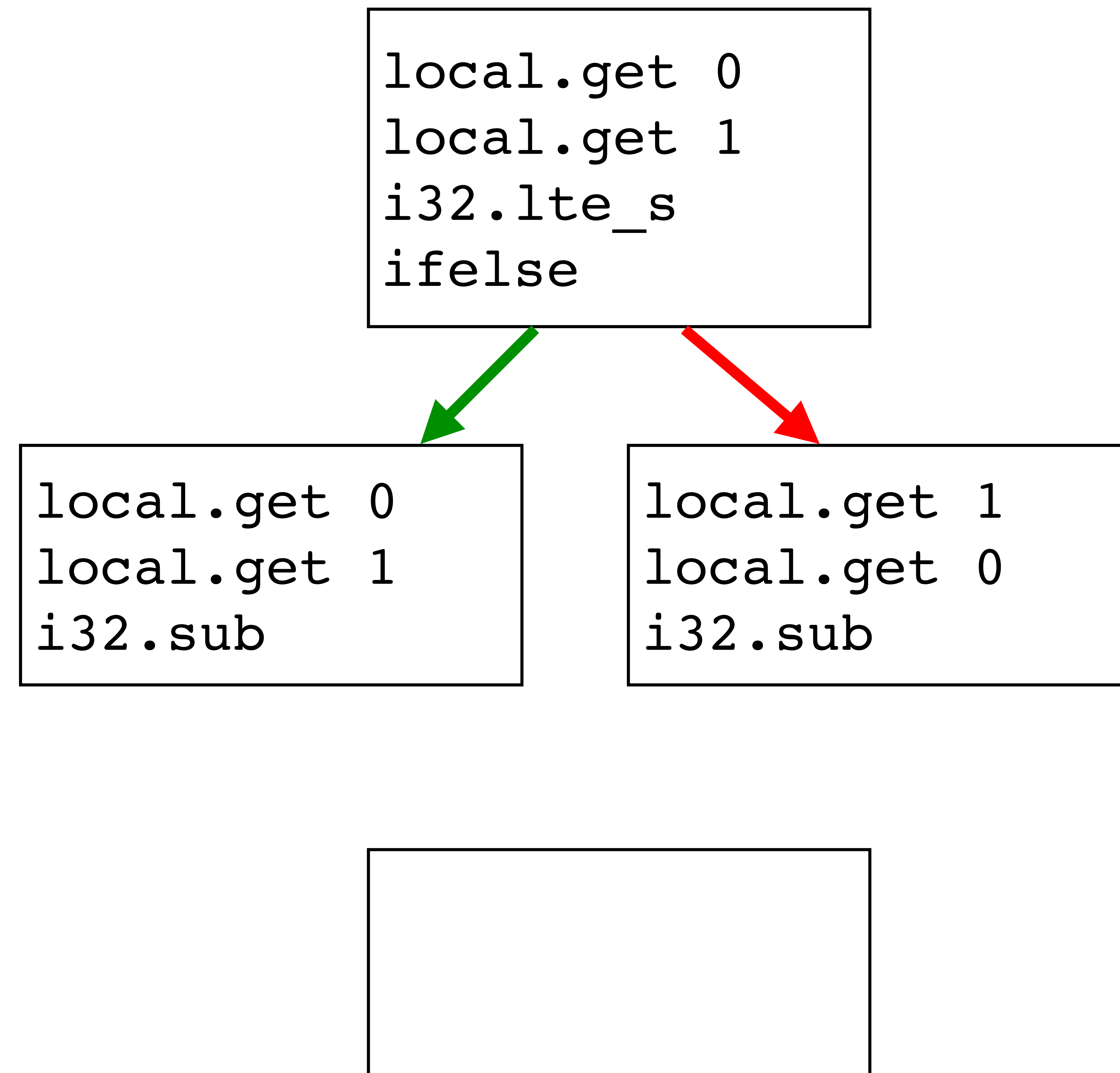
```
local.get 0
local.get 1
i32.lte_s
if [i32]
  local.get 0
  local.get 1
  i32.sub
else
  local.get 1
  local.get 0
  i32.sub
end
i32.const 2
i32.mul
return
end
```



Example

.wasm

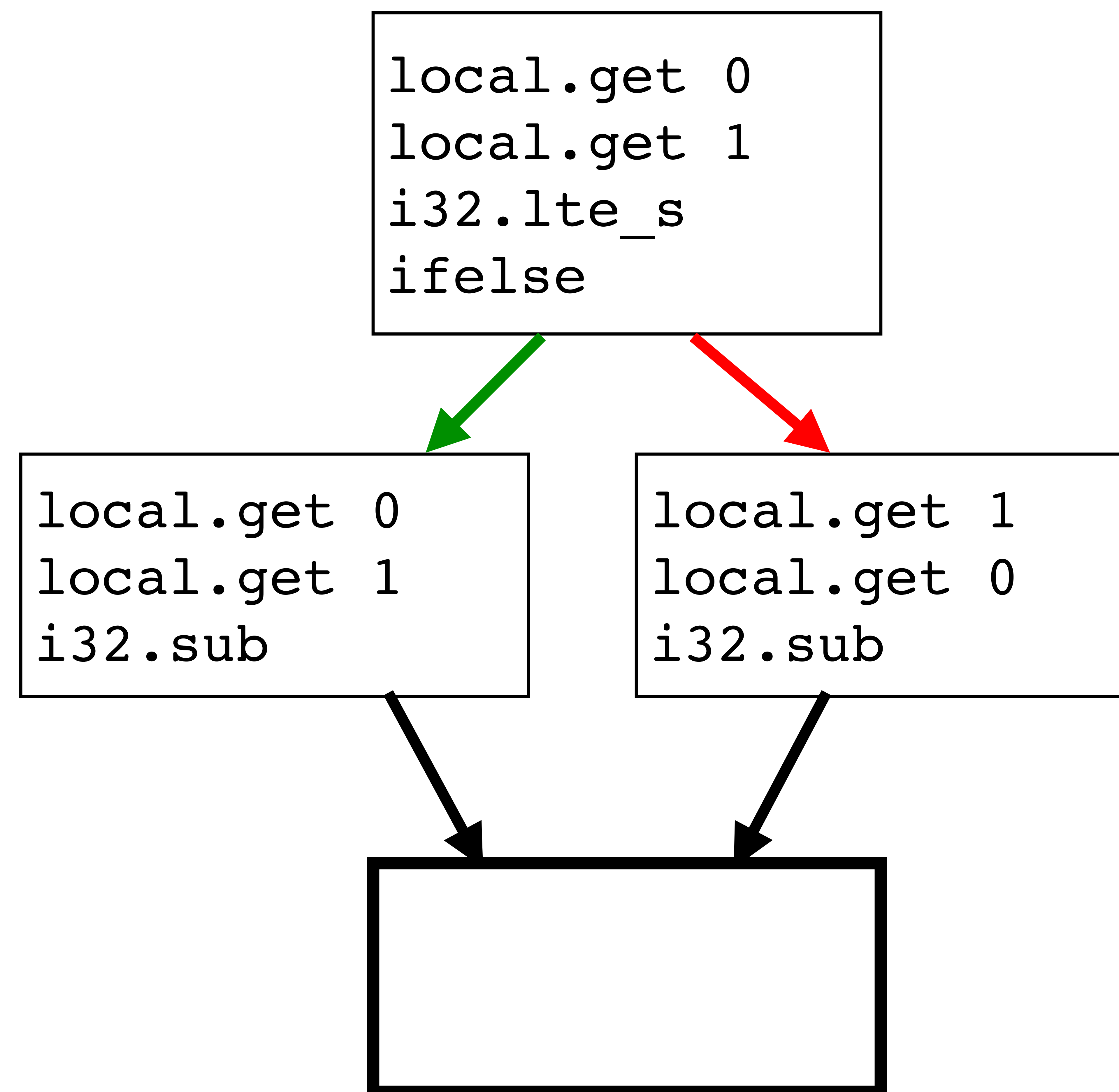
```
local.get 0
local.get 1
i32.lte_s
if [i32]
  local.get 0
  local.get 1
  i32.sub
else
  local.get 1
  local.get 0
  i32.sub
end
i32.const 2
i32.mul
return
end
```



Example

.wasm

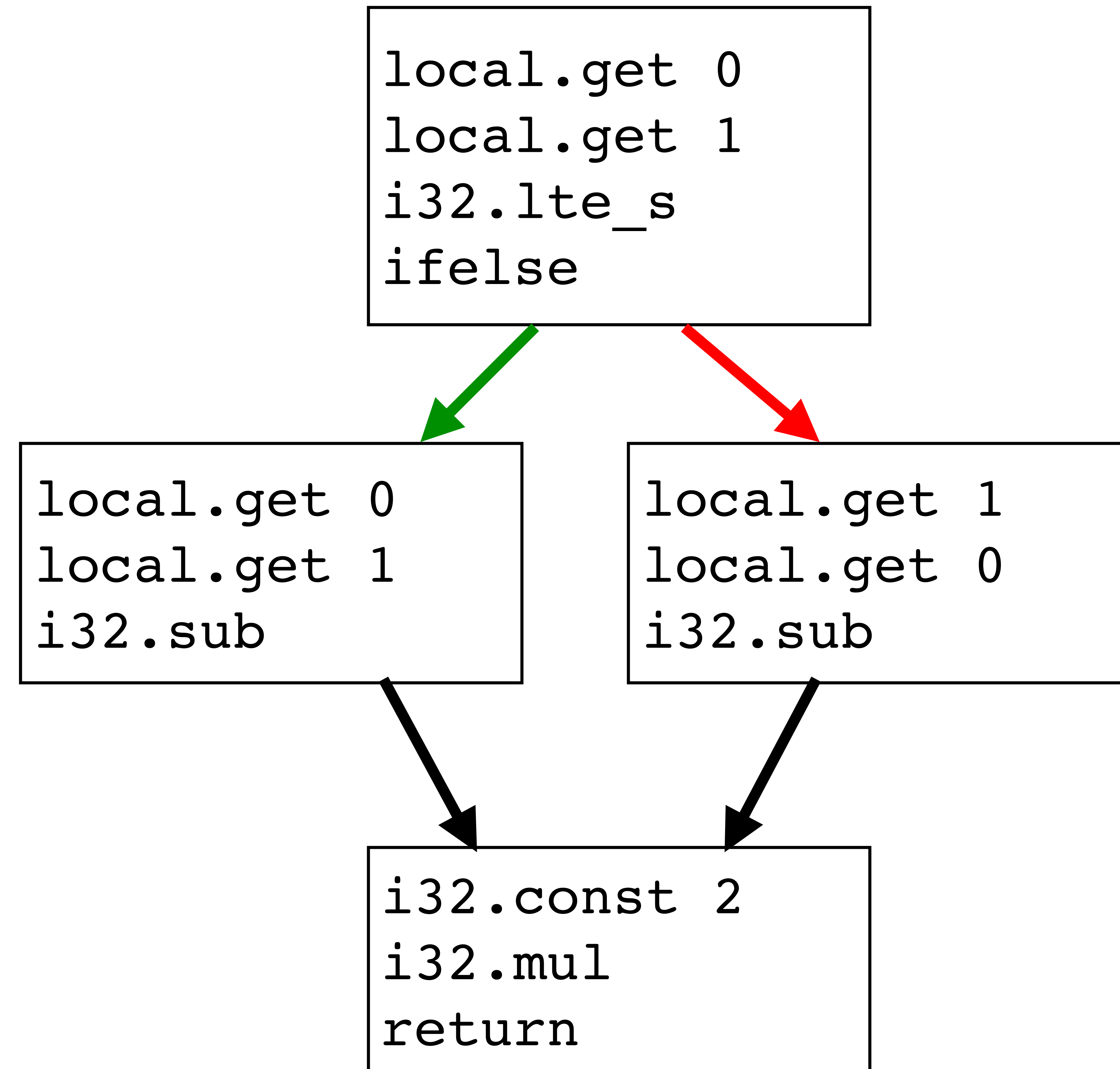
```
local.get 0
local.get 1
i32.lte_s
if [i32]
   local.get 0
   local.get 1
   i32.sub
else
   local.get 1
   local.get 0
   i32.sub
end
i32.const 2
i32.mul
return
end
```



Example

.wasm

```
local.get 0
local.get 1
i32.lte_s
if [i32]
  local.get 0
  local.get 1
  i32.sub
else
  local.get 1
  local.get 0
  i32.sub
end
i32.const 2
i32.mul
return
end
```



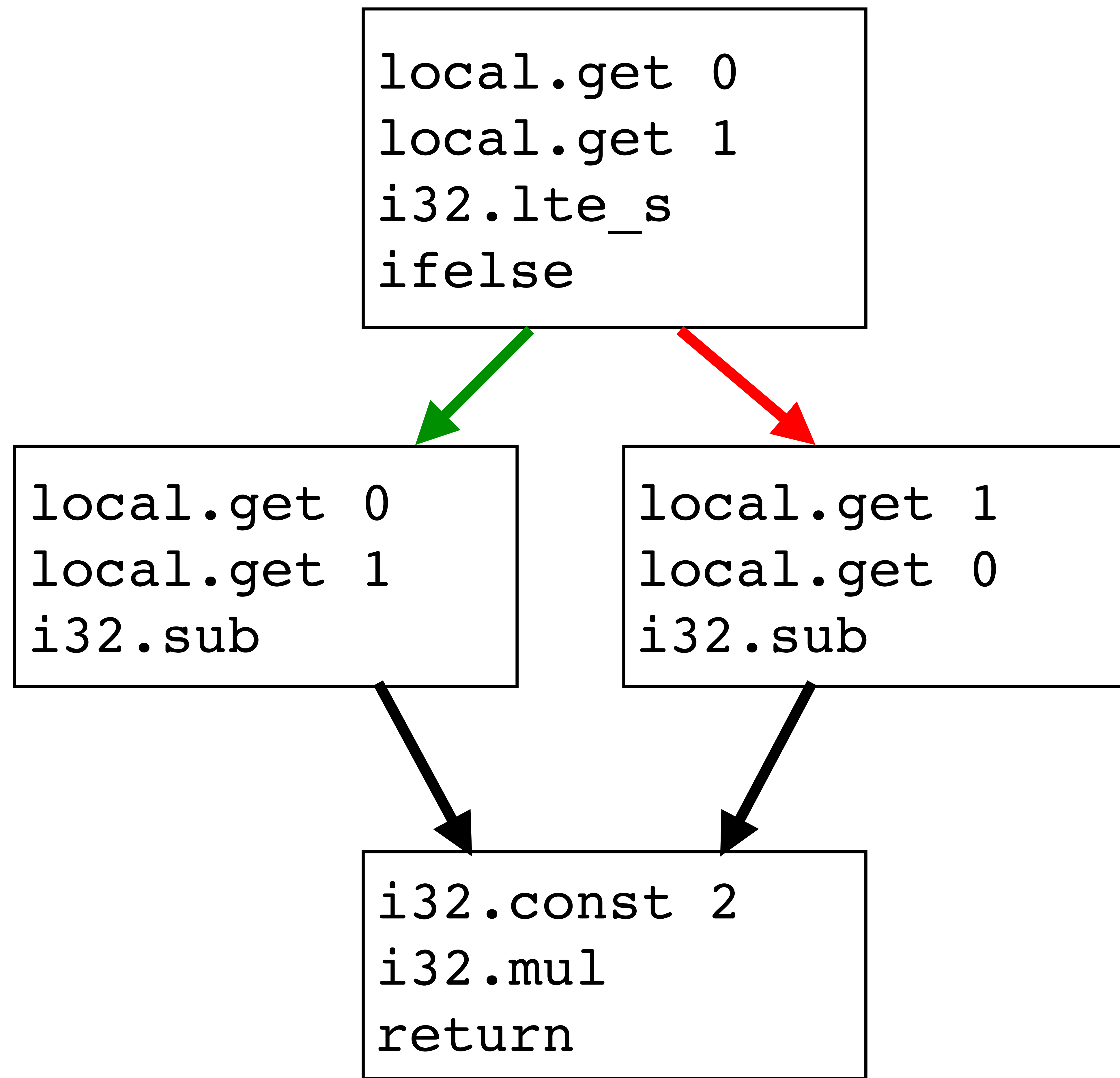
CFG to SSA

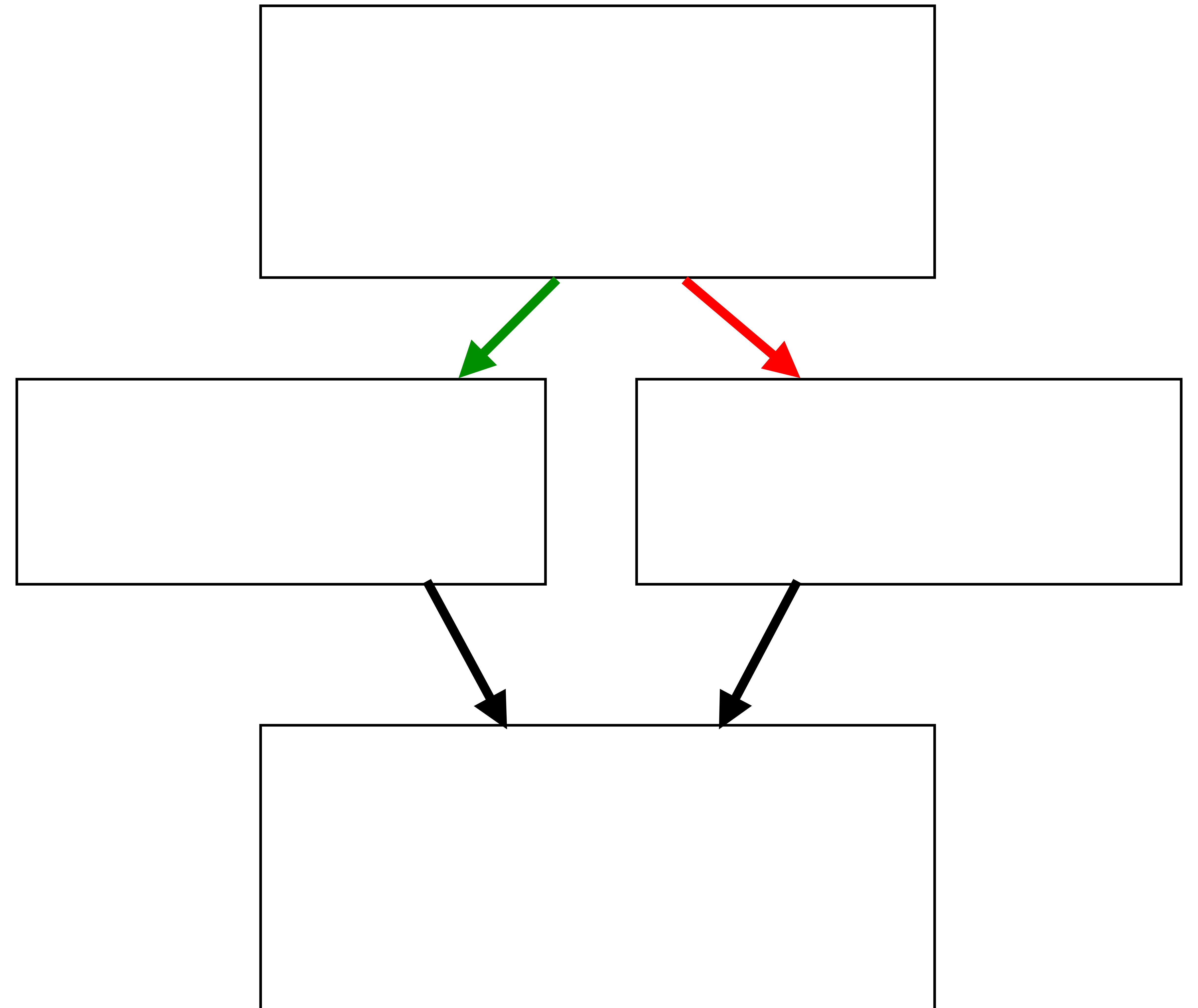
Convert the CFG into *Single Static Assignment Form* IR

- remove memory management with symbolic names
- insert φ-statements to resolve conflicts
- reduce unnecessary φ's and assignments

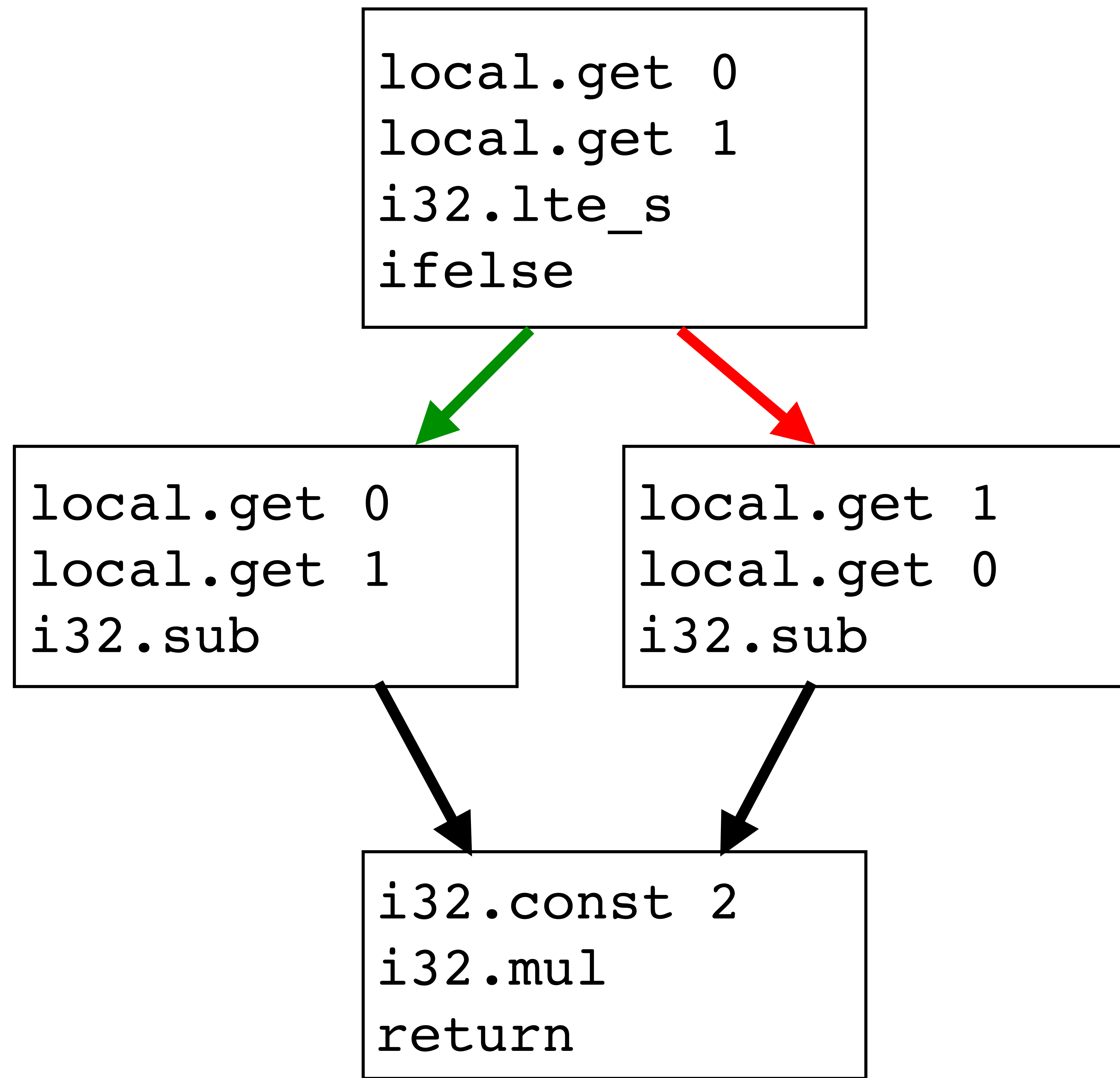
y := φ(x, x) → y := x   y := x, z := f(y) → z := f(x)

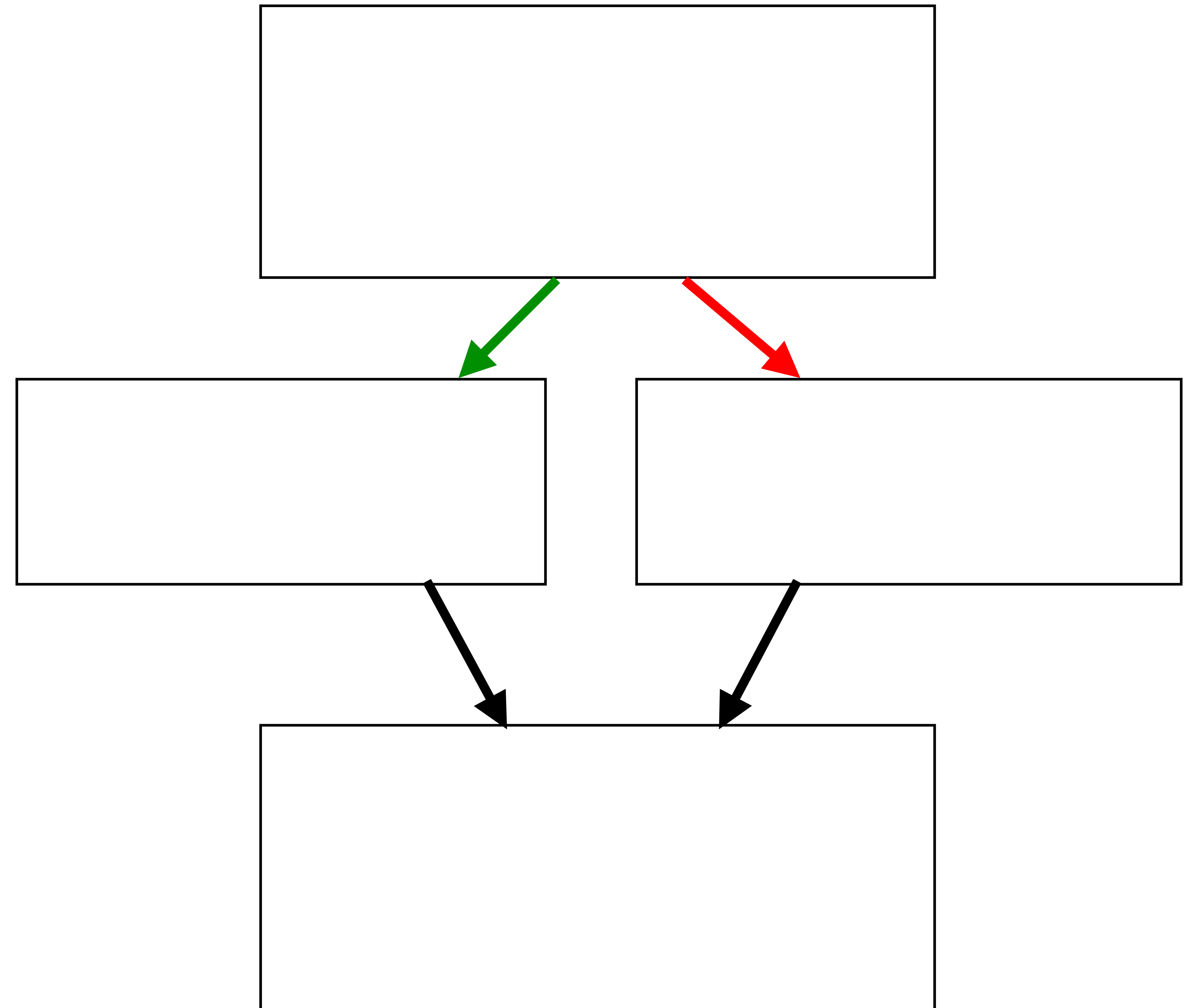
Example



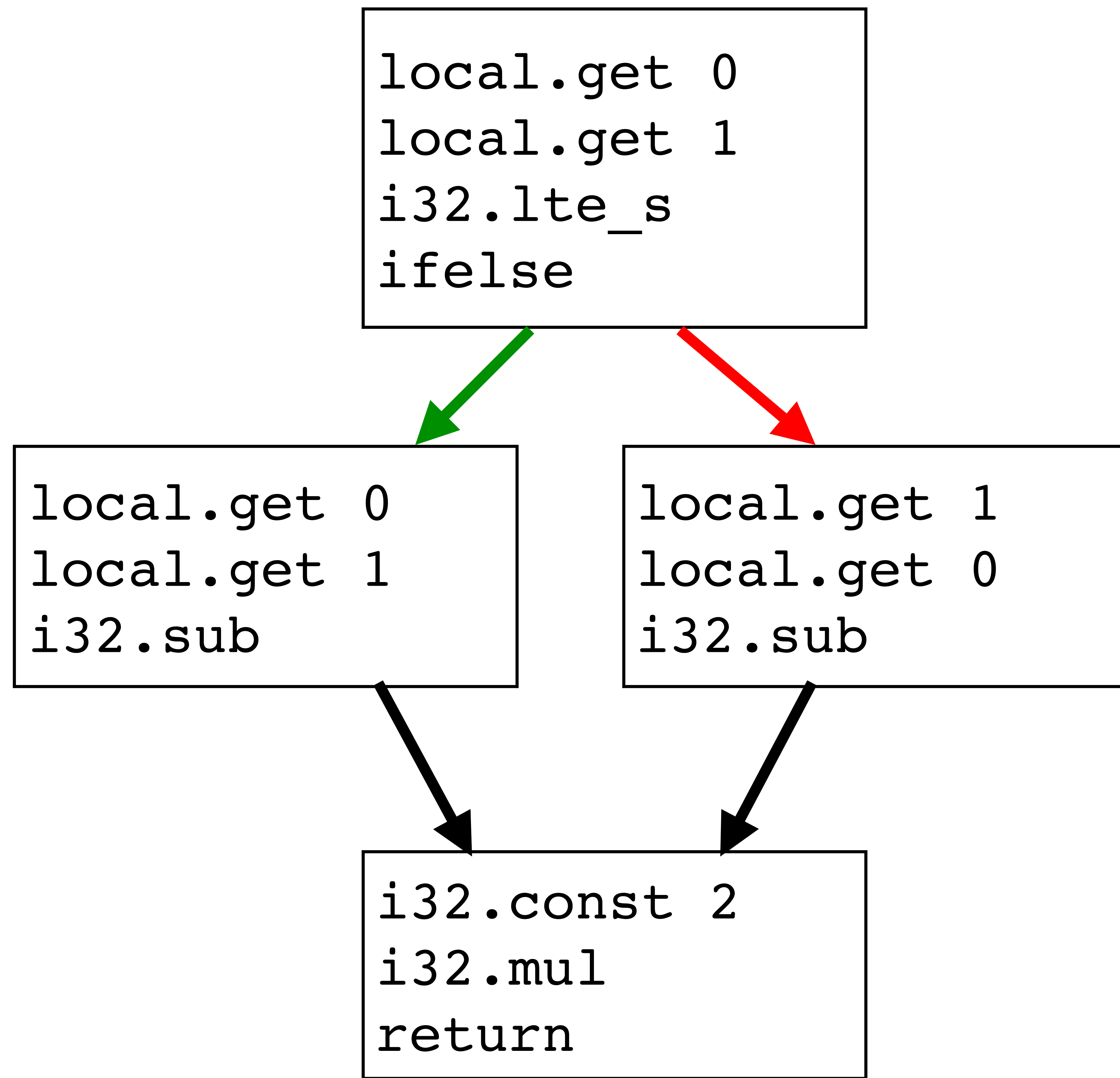


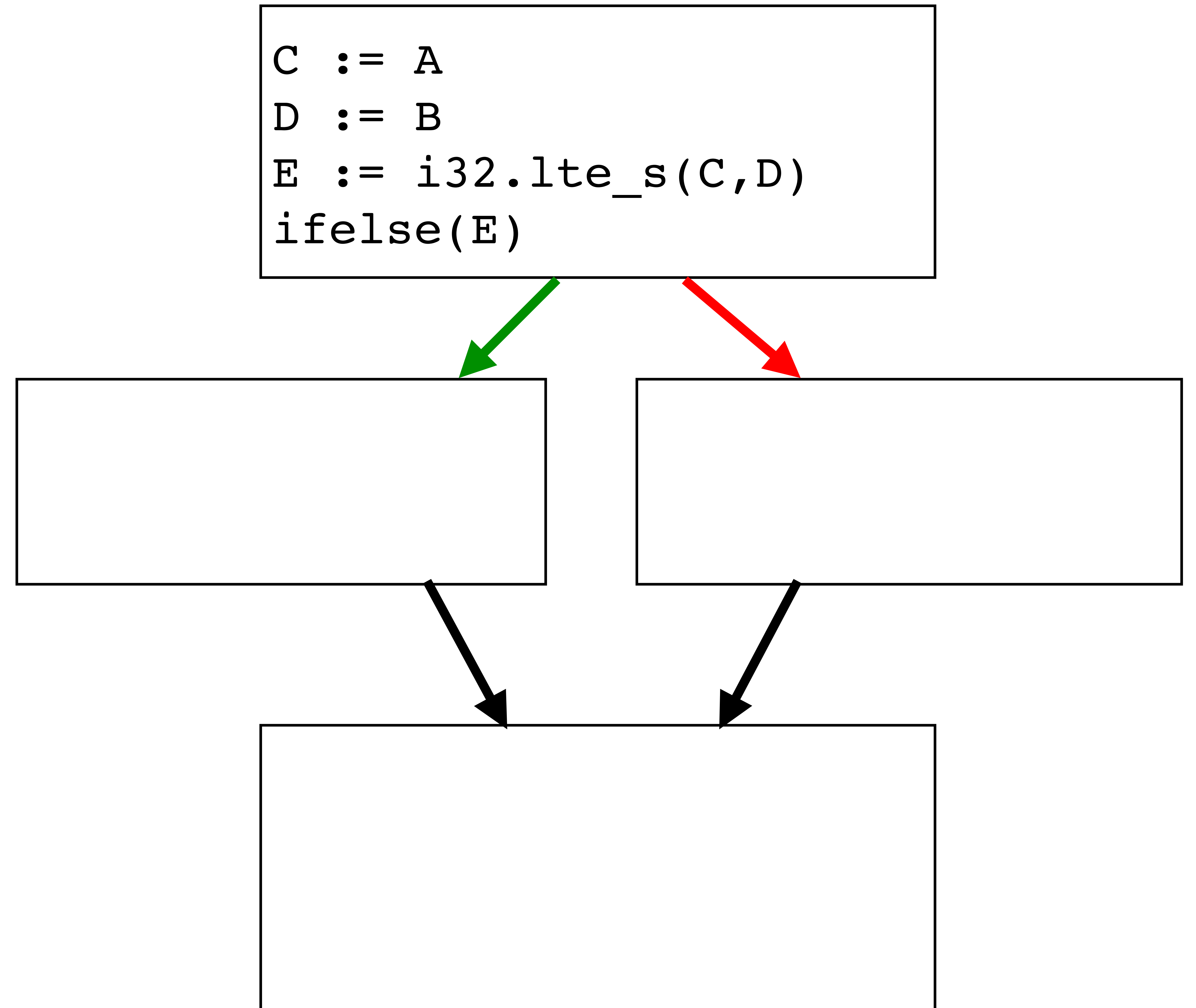
Example



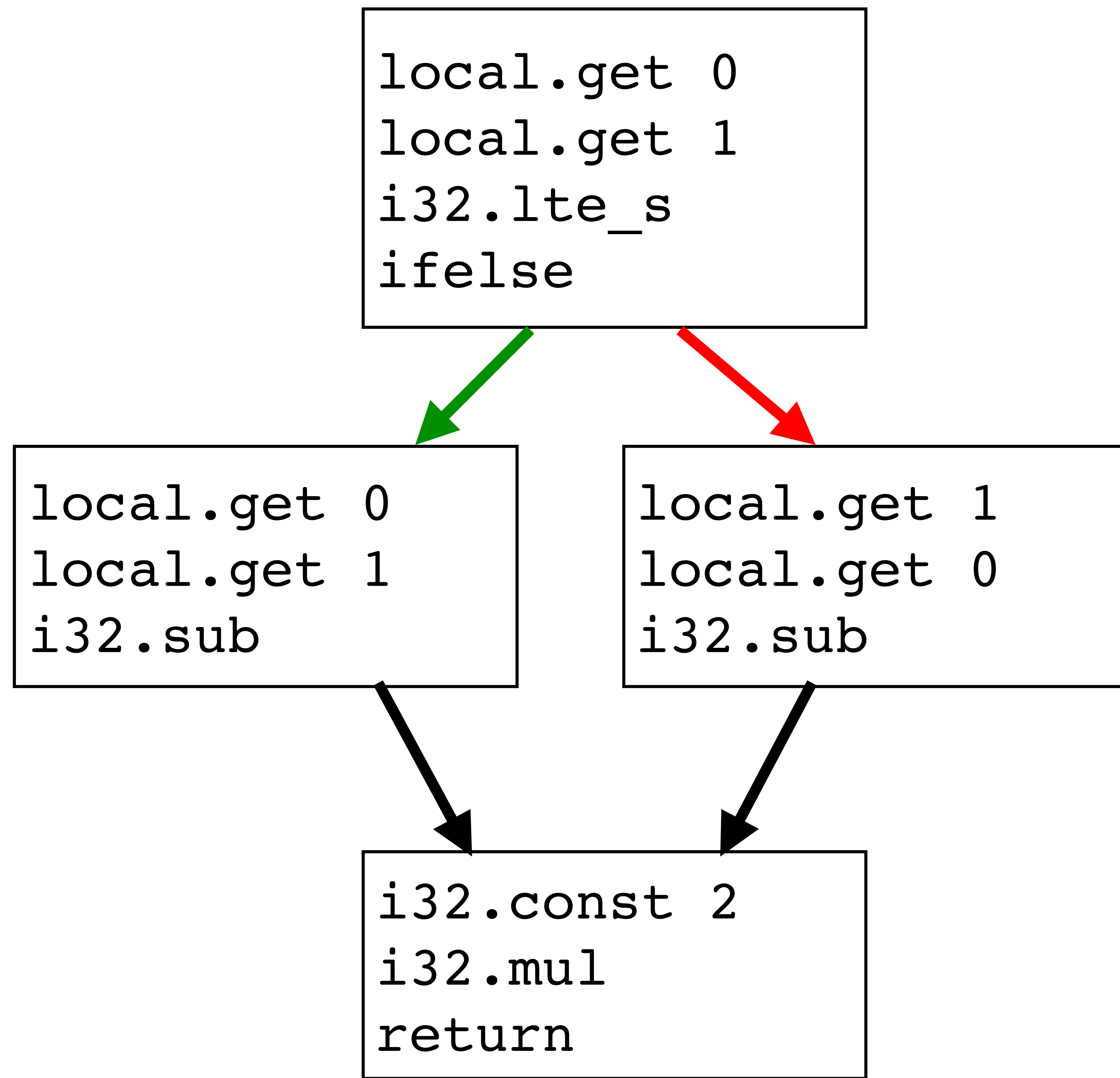


Example





Example

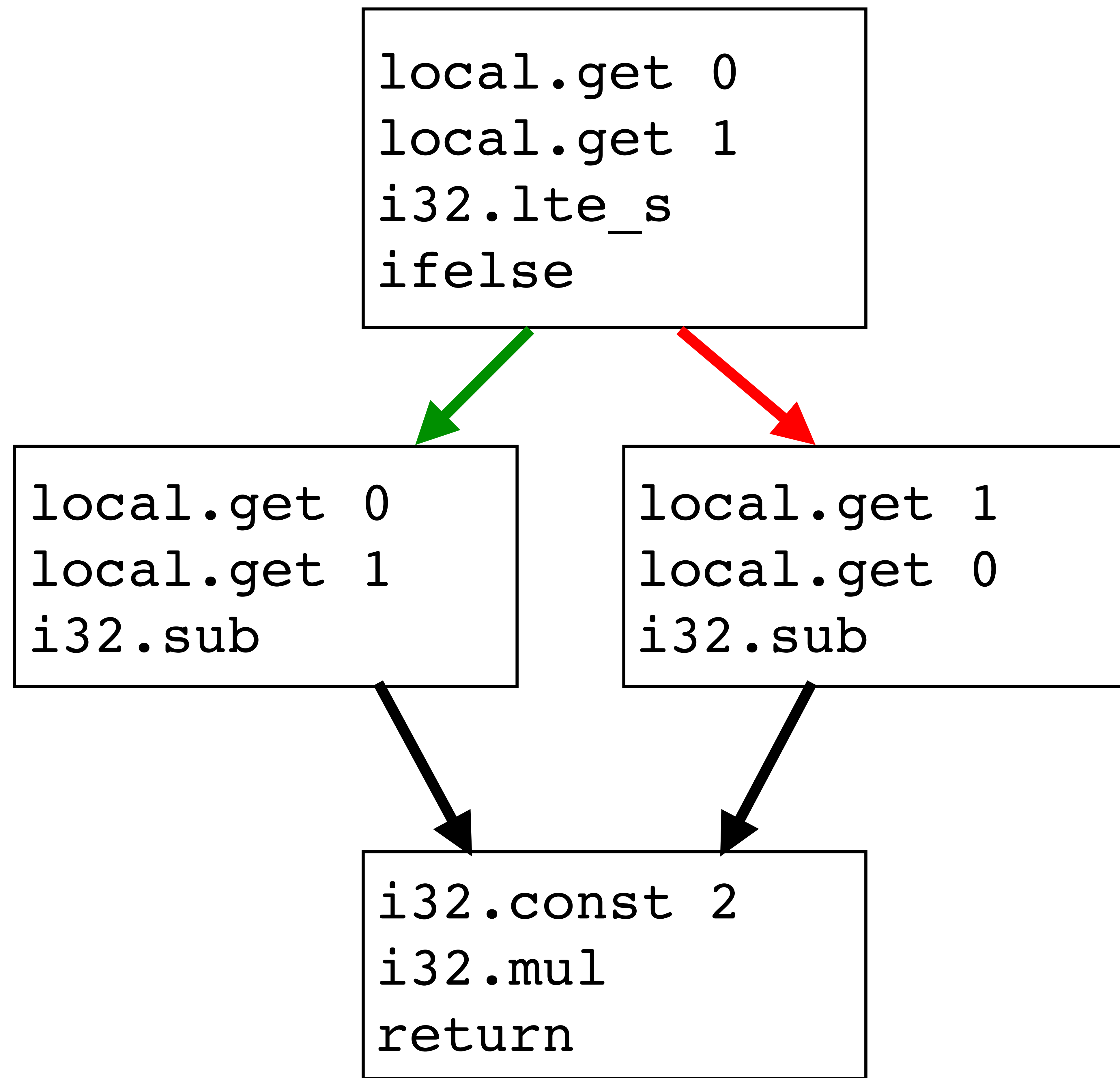



```
A := local[0]
B := local[1]
```

```
C := A
D := B
E := i32.lte_s(C,D)
ifelse(E)
```

```
F := A
G := B
H := i32.sub(F,G)
```

Example



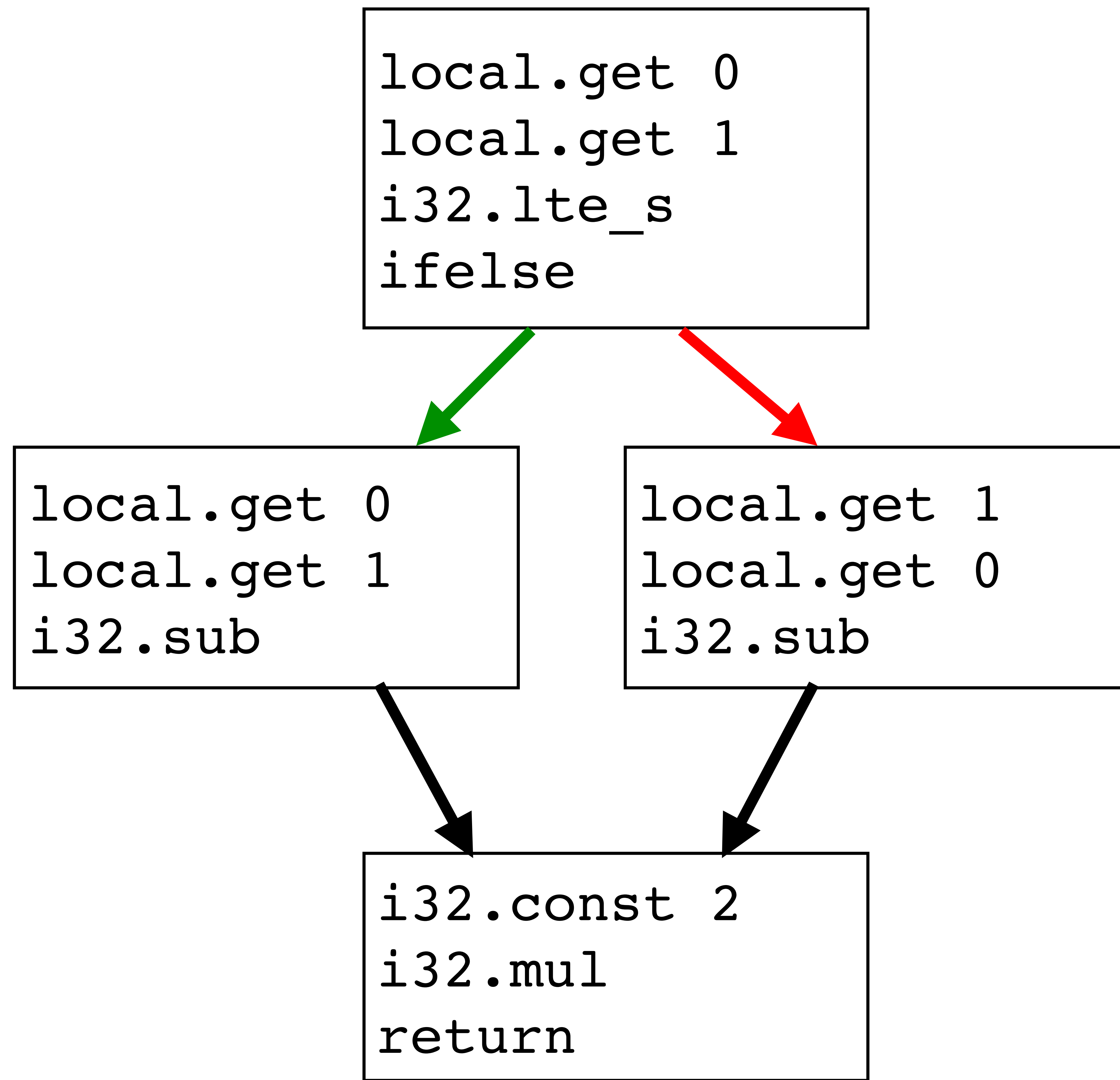

```
A := local[0]
B := local[1]
```

```
C := A
D := B
E := i32.lte_s(C,D)
ifelse(E)
```

```
F := A
G := B
H := i32.sub(F,G)
```

```
I := B
J := A
K := i32.sub(I,J)
```

Example




```
A := local[0]
B := local[1]
```

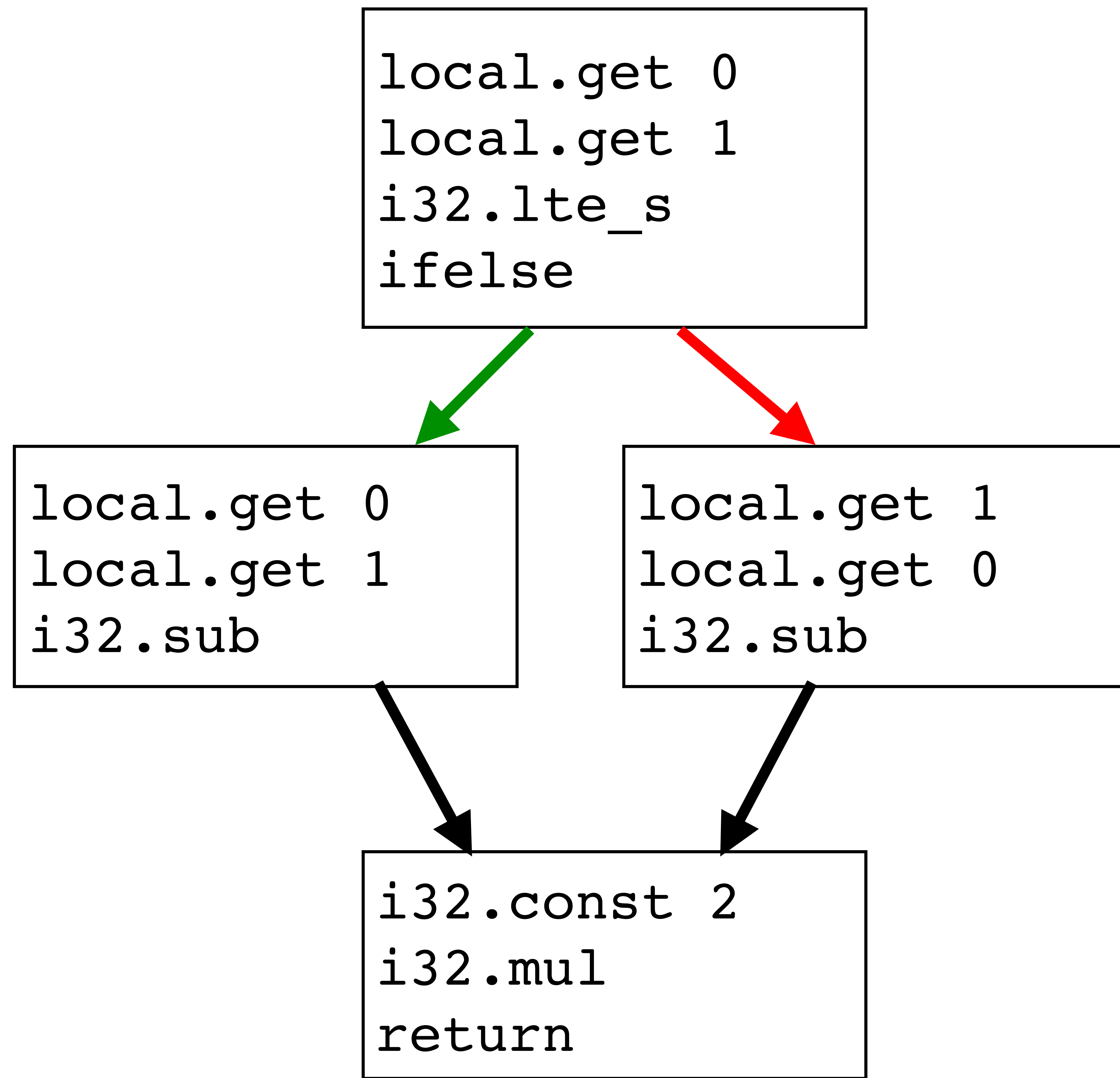
```
C := A
D := B
E := i32.lte_s(C,D)
ifelse(E)
```

```
F := A
G := B
H := i32.sub(F,G)
```

```
I := B
J := A
K := i32.sub(I,J)
```

```
L := φ(H,K)
```

Example




```
A := local[0]
B := local[1]
```

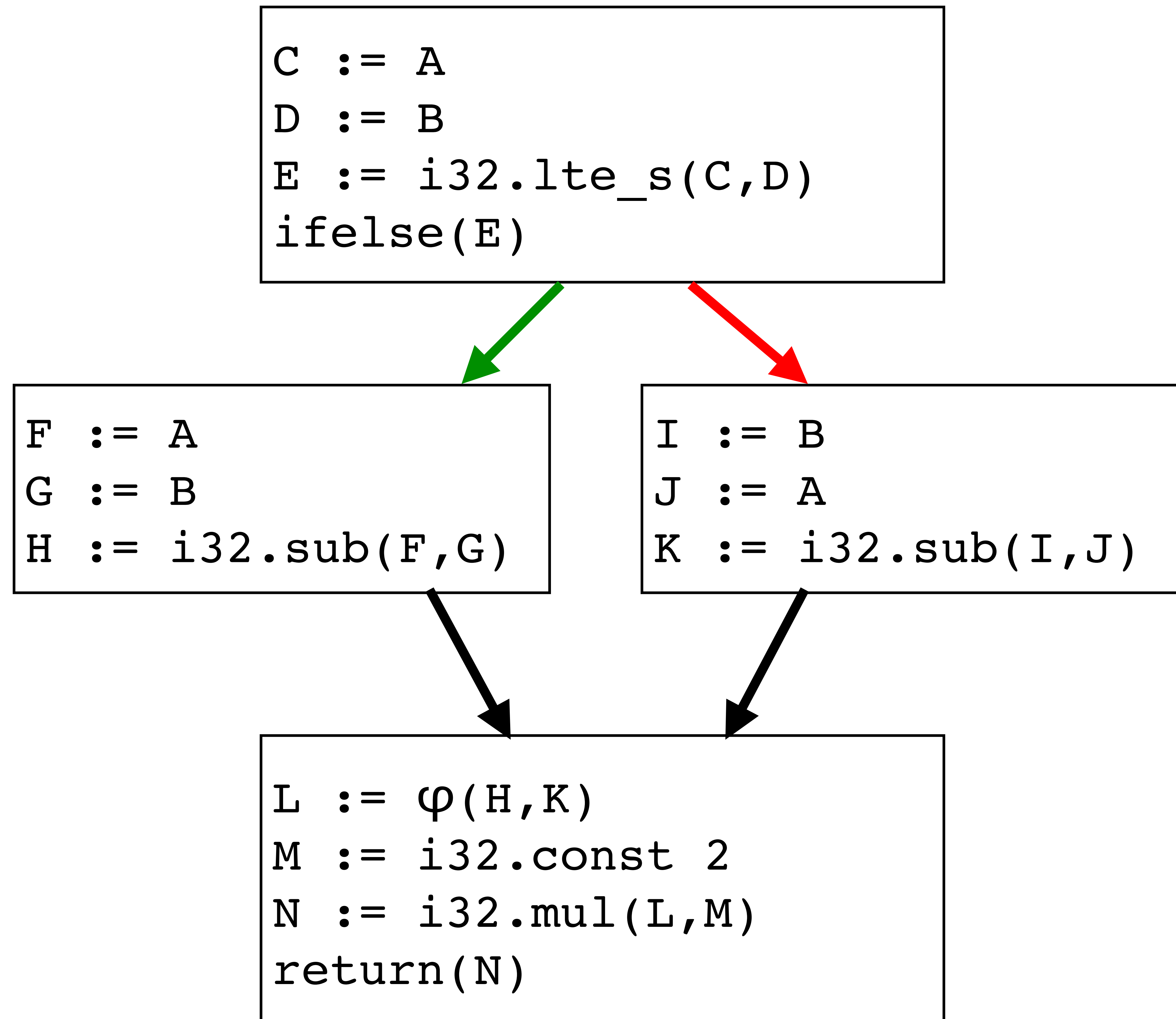
```
C := A
D := B
E := i32.lte_s(C,D)
ifelse(E)
```

```
F := A
G := B
H := i32.sub(F,G)
```

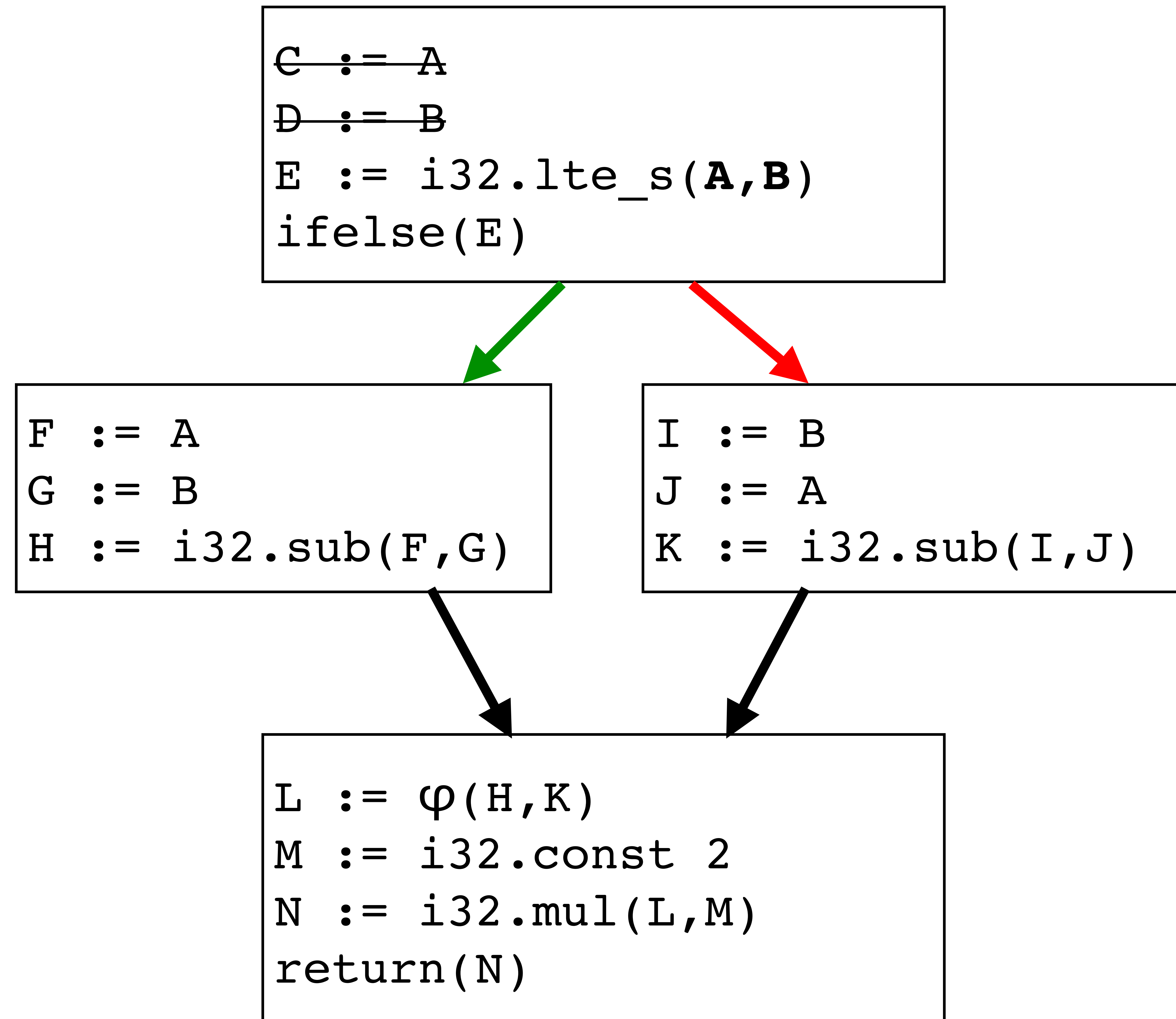
```
I := B
J := A
K := i32.sub(I,J)
```

```
L := φ(H,K)
M := i32.const 2
N := i32.mul(L,M)
return(N)
```

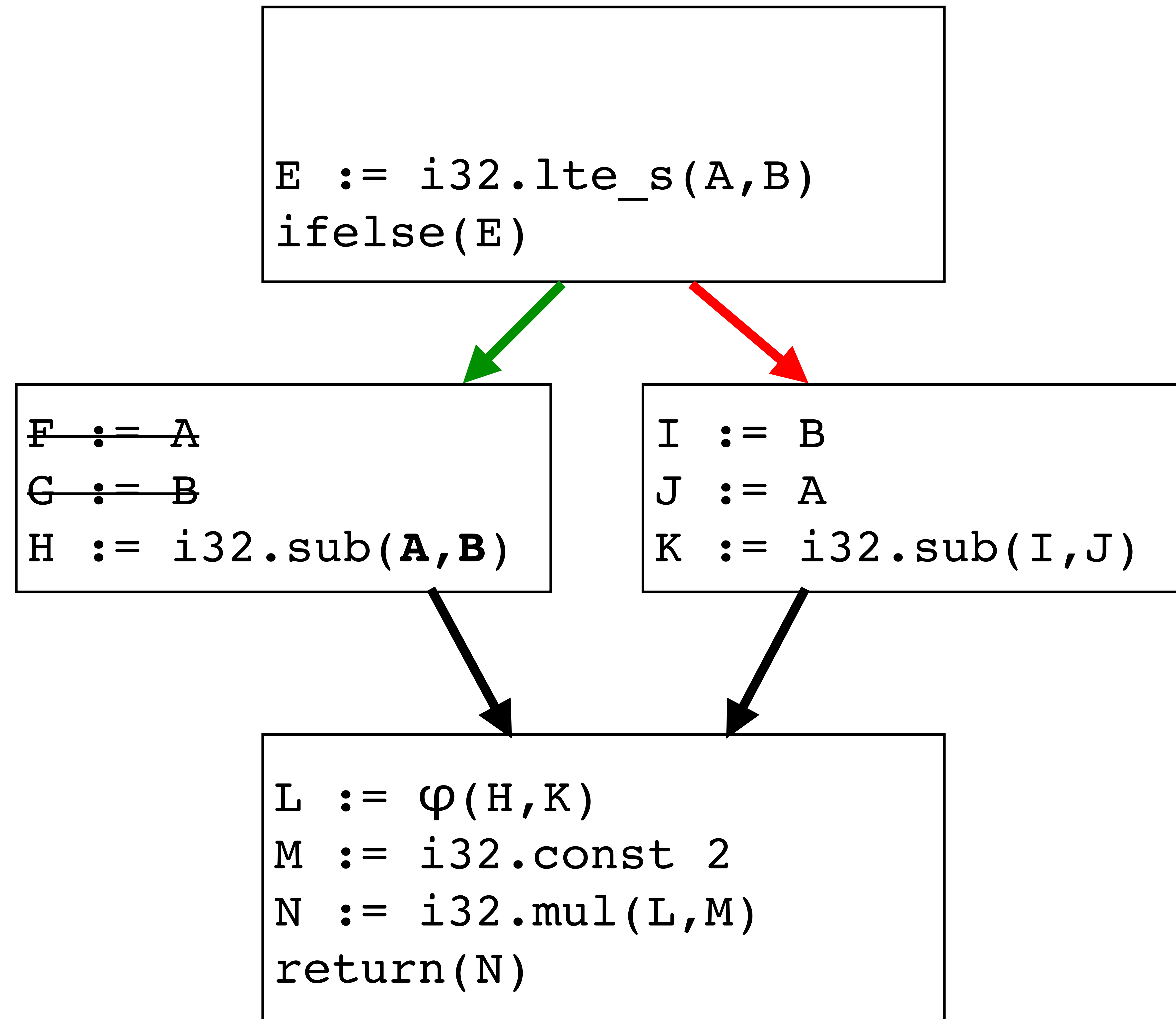
Example



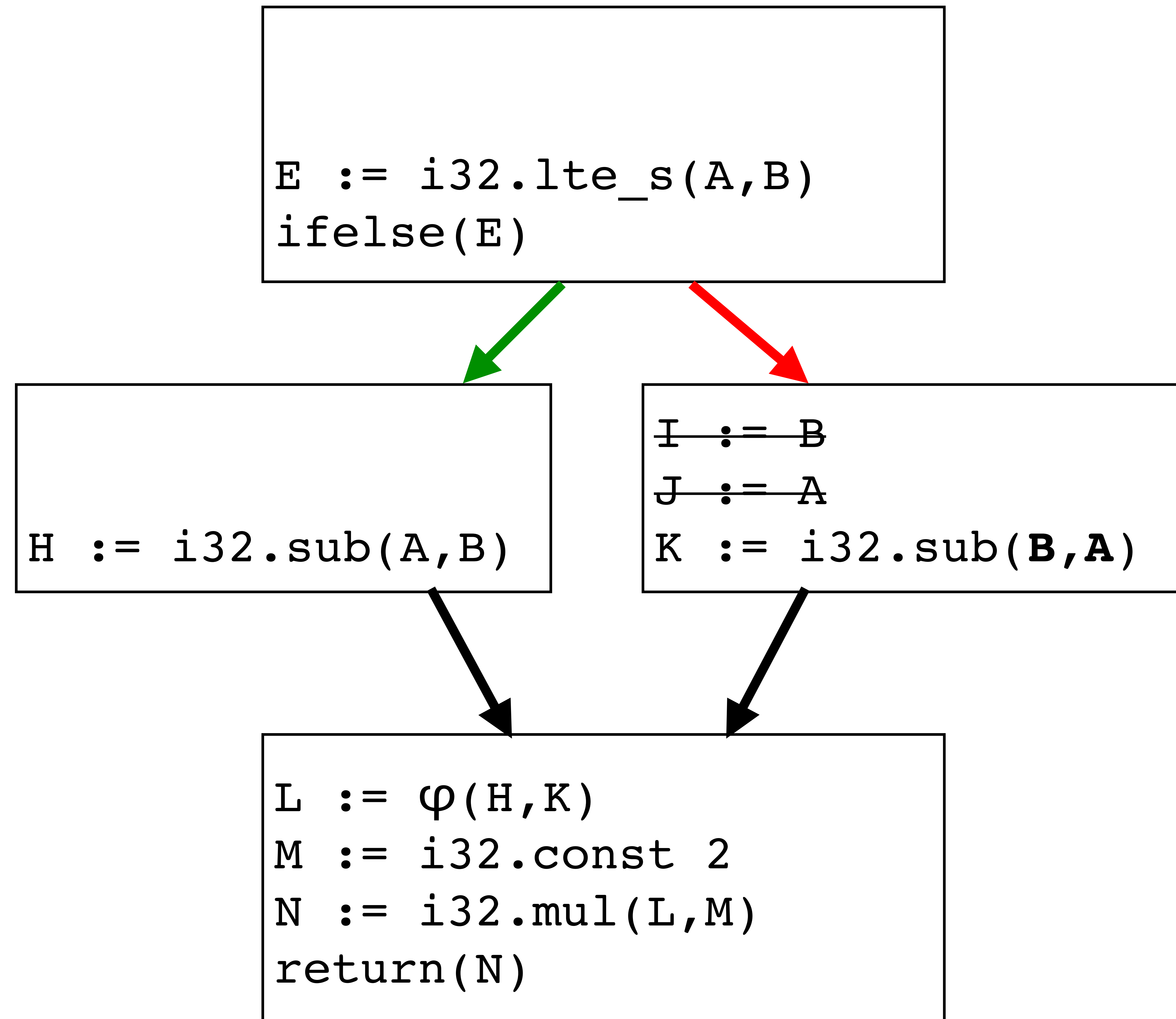
Example



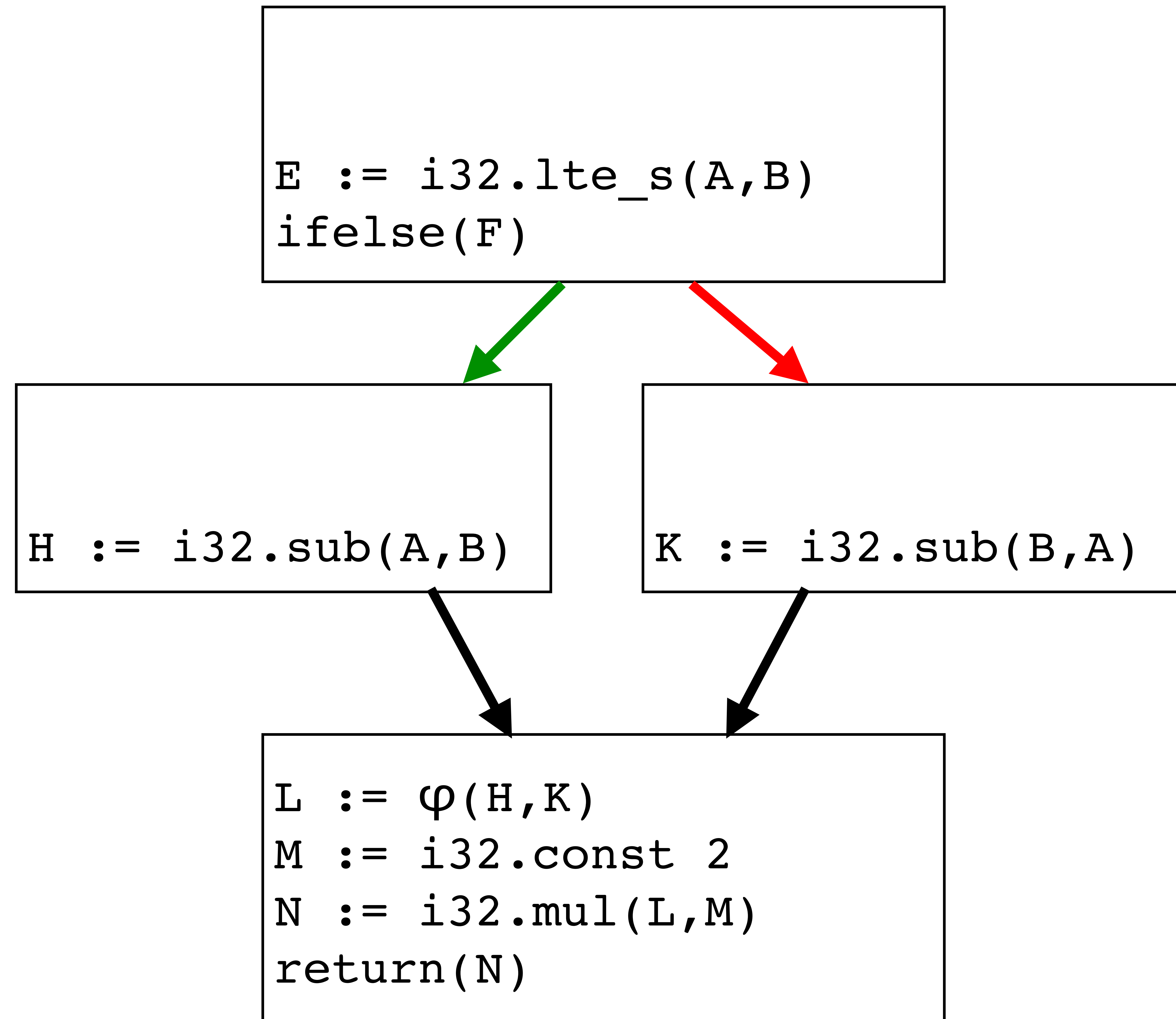
Example



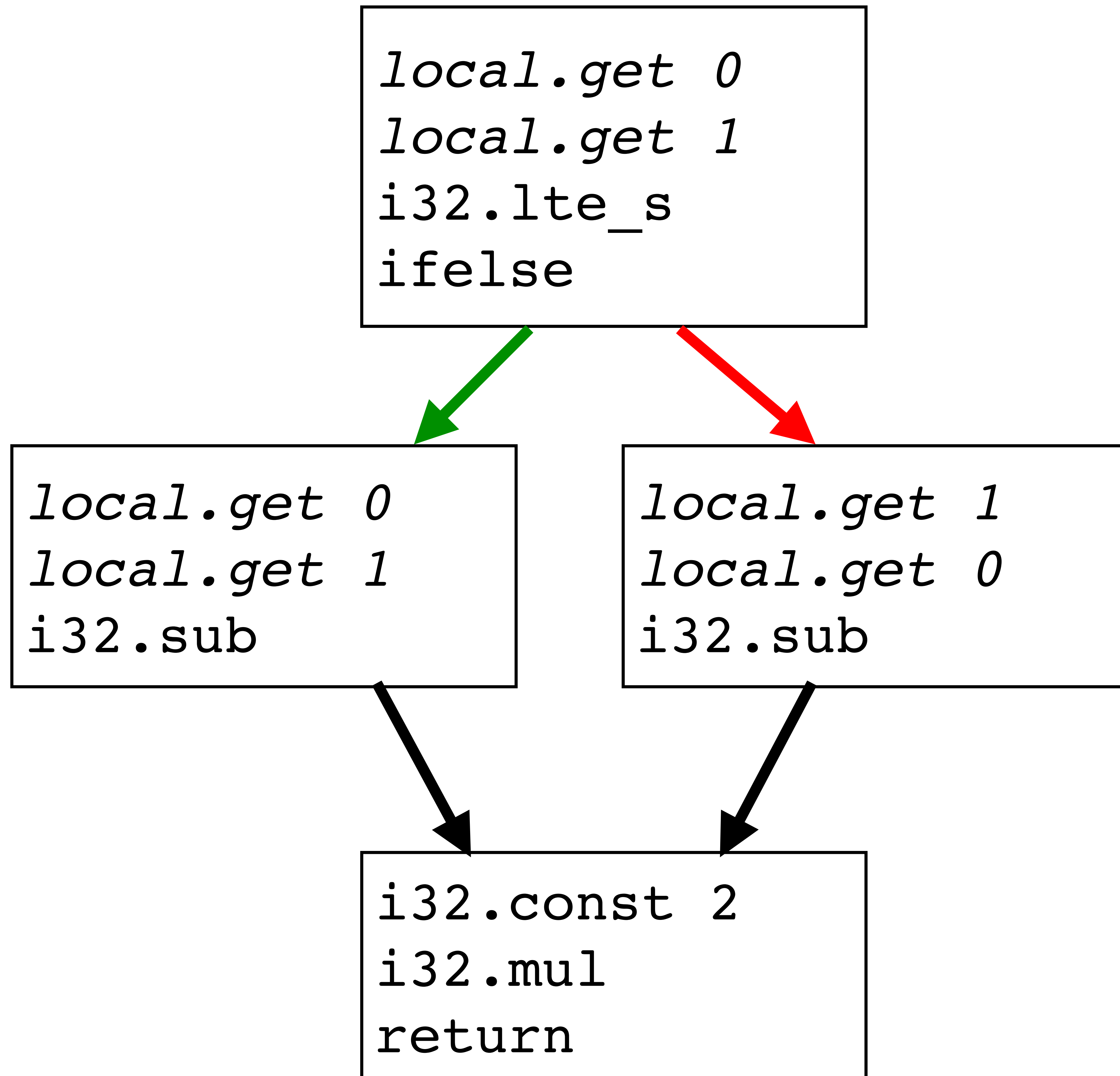
Example

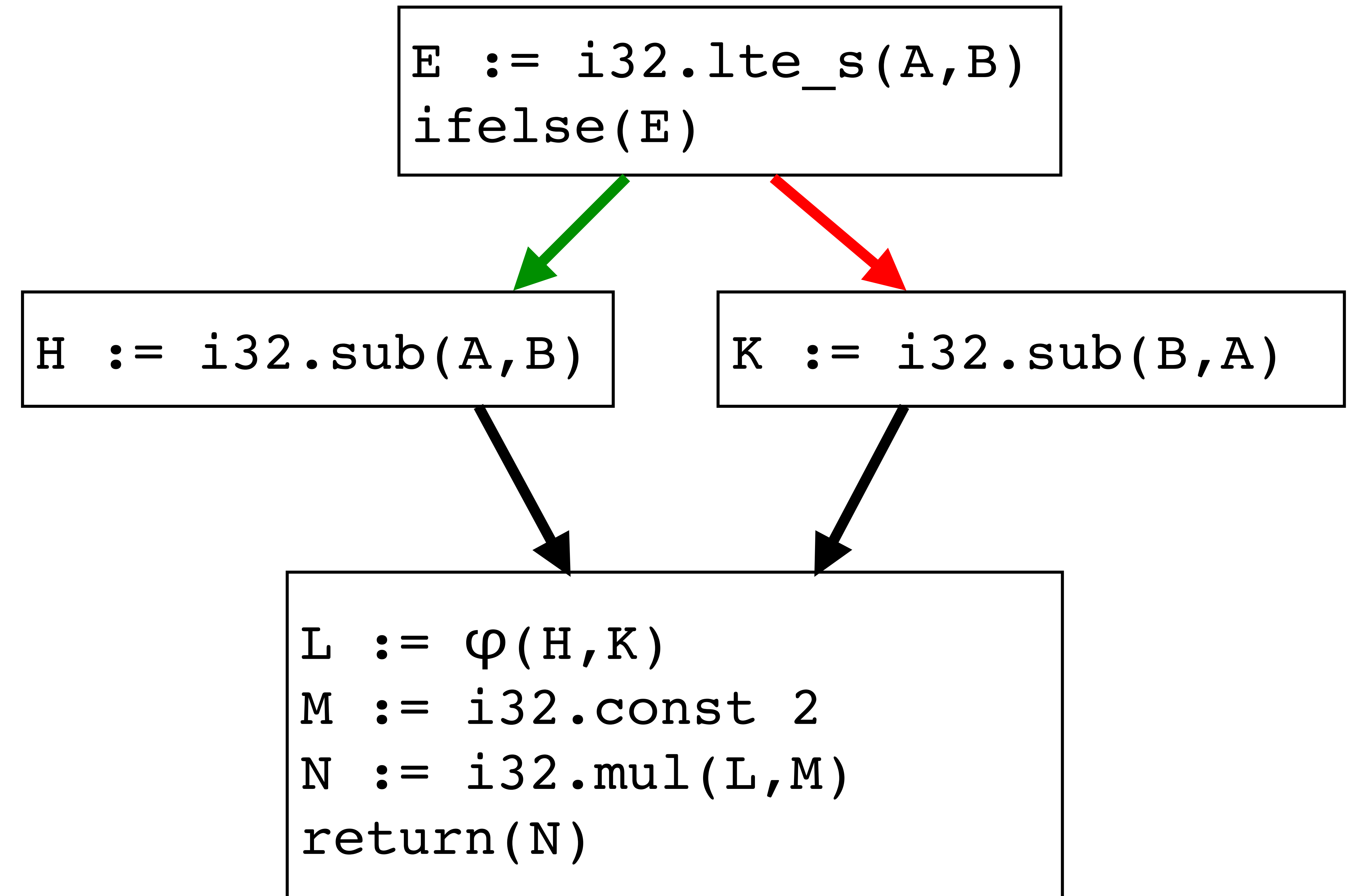


Example



Example





Future Work

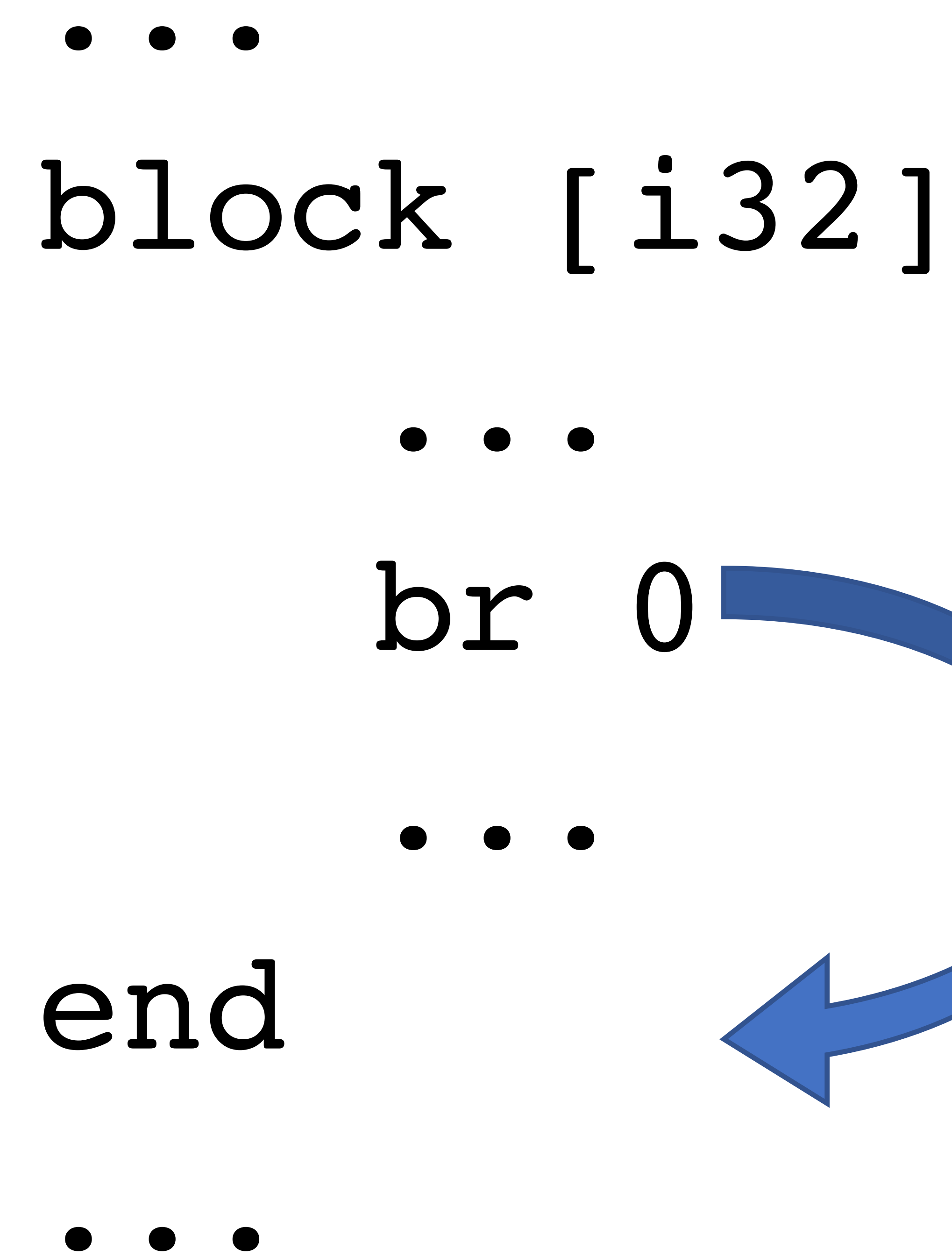
- pipeline completion
- evaluation of different compilers
- performance improvements

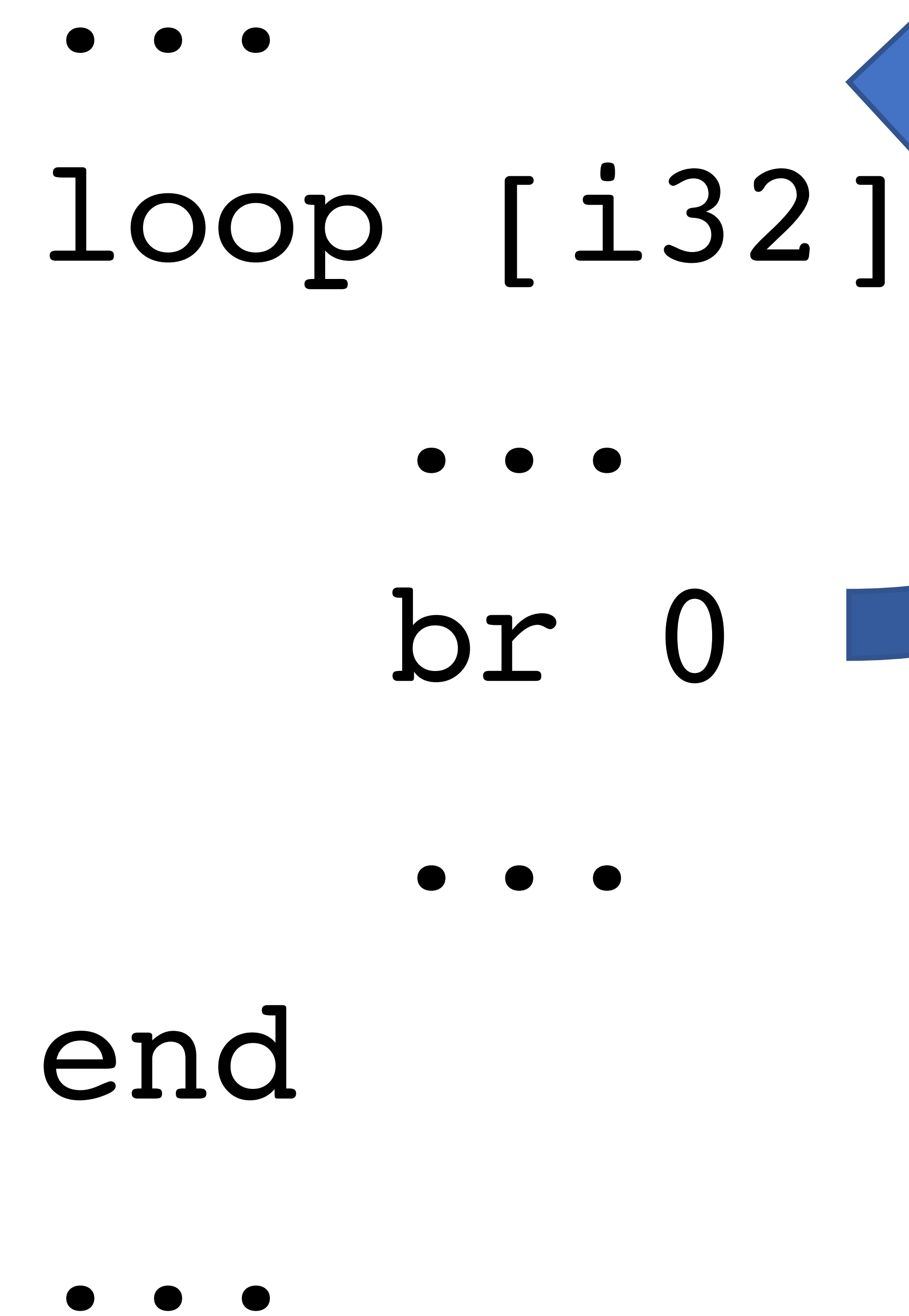
Demo

Conclusion

- WASM is a really interesting format
- WASM is easy to understand
- Restrictions make optimisations hard
    - A few more stack operations would really help

Excursion: Branching Instructions





Excursion: Branching Instructions

- branch target is always relative
- branching behaviour depends on branch target

$$\mathsf{label}_n\{instr^*\}\ B^l[val^n\ (\mathsf{br}\ l)]\ \mathsf{end} \quad\hookrightarrow\quad val^n\ instr^*$$

- labels are created when entering blocks or loops

$$\mathsf{block}\ [t^n]\ instr^*\ \mathsf{end} \quad\hookrightarrow\quad \mathsf{label}_n\{\epsilon\}\ instr^*\ \mathsf{end}$$

$$\mathsf{loop}\ [t^?]\ instr^*\ \mathsf{end} \quad\hookrightarrow\quad \mathsf{label}_0\{\mathsf{loop}\ [t^?]\ instr^*\ \mathsf{end}\}\ instr^*\ \mathsf{end}$$

- only branches targeting loops ignore return types