Compiler Construction

Code generation and optimisation

Marcin Benke

Nov 25–Dec 2, 2019

Code generation and optimisation

- Basic blocks and Control Flow Graphs
- Flow analysis
- Live variables and reaching definitions
- Code generation
- Register allocation
- Code improvement ("optimisation")
  - Constant folding
  - Common subexpression elimination
  - Dead code elimination
  - Loop optimisation

Basic block

Definition 1. A *basic block* is a sequence of instructions, where control can enter only at the beginning and leave at the end, without any branching inside. (only the last instruction can be a jump, only the first can be a jump target).

Consider the fragment:

```
[ 1]  i := m-1
[ 2]  j := n
[ 3]  t1 := 4*n
[ 4]  v := a[t1]

[ 5]  i := i+1
[ 6]  t2 := 4*i
[ 7]  t3 := a[t2]
[ 8]  if t3 < v goto (5)
```

The sequence 5–8 is a basic block, the sequence 1-8 is not.

Basic blocks

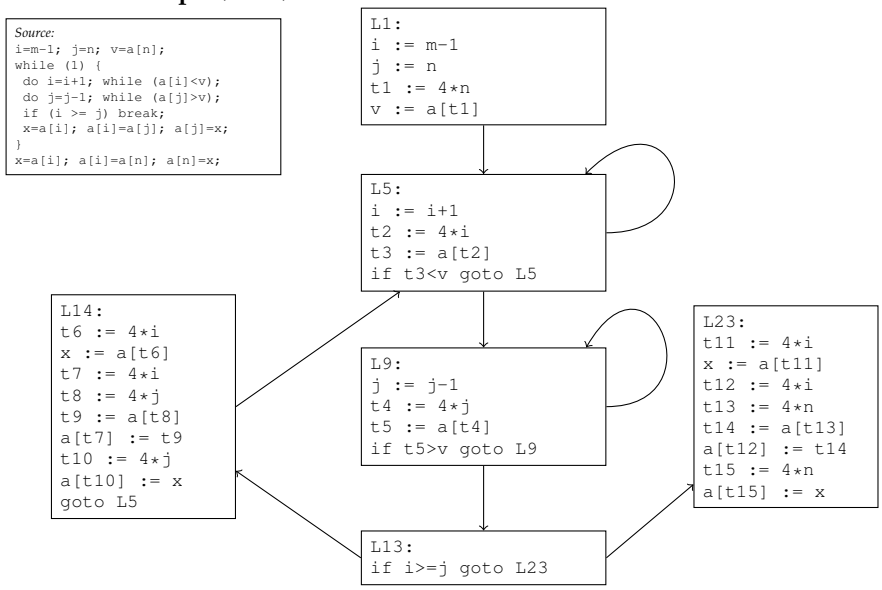
The code fragment consists of two basic blocks:

```
L1:  i := m-1
     j := n
     t1 := 4*n
     v := a[t1]
     goto L5

L5:  i := i+1
     t2 := 4*i
     t3 := a[t2]
     if t3 < v goto L5
```

The blocks can be identified by their start label, so we may talk about blocks L1 and L5.

Control Flow Graph (CFG)



Liveness analysis

- *Variable definition* — an instruction assigning value to the variable.
- *Variable use* — an instruction using the value of the variable.
- Instruction `x:=y+z` defines `x`, uses `z` and `y`

Definition 2. A variable is **alive** at a given point, if its current value can be used, i.e. there exists a path to its use, not containing any definition of this variable.

Example

At the start of the block

```
t := a
a := b
b := t
```

The variables a, b are alive. The variable t is dead, since the only path to its use contains its definition.

Reaching definitions

Definition 3. A definition reaches given point in code if no path between them contains **another** definitions of the same variable.

```
[ 2]  j := n
[ 3]  t1 := 4*n
[ 4]  v := a[t1]
[ 5]  i := i+1
[ 6]  t2 := 4*i
[ 7]  t3 := a[t2]
[ 8]  if t3 < v goto (5)
[ 9]  j := j-1
[10]  t4 := 4*j
[11]  t5 := a[t4]
```

The definition of t1 at (3) reaches (11); the definition of j at (2) does not reach (10).

Flow analysis

Flow information is usually computed via equation systems, modelling dependencies between various points in the program.

An equation (for the flow forward) has the form:

$$\mathtt{out}[S] = \mathtt{gen}[S] \cup (\mathtt{in}[S] - \mathtt{kill}[S])$$

The information available after a unit consists of information

- generated by it ($\mathtt{gen}[S]$)
- available beforehand ($\mathtt{in}[S]$)
- less the information destroyed ($\mathtt{kill}[S]$).

Flow analysis

A flow equation

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

S can be a single quadruple or a basic block

Definitions of gen and $kill$ depend on the information analysed

For liveness analysis, information flows backwards
in is computed from out:

$$in[S] = (out[S] - kill[S]) \cup use[S]$$

Example

$$in[S] = (out[S] - kill[S]) \cup use[S]$$

```
t := a  out={b,t}  kill={t}, use={a}, in={a,b}
a := b  out={t}  kill={a}, use={b}, in={b,t}
b := t  out=∅, kill={b}, use={t}, in={t}
```

Example

Bigger example:

```
{a,b,c}
t := a {b,c,t}
a := b {a,c,t}
b := t {a,b,c}
t := a-b {t,a,c}
u := a-c {t,u}
v := t+u {v,u}
d := v+u {d}
```

d live at the end

Global flow analysis

Interblock flow analysis based on incoming and outgoing edges of a block.

For liveness analysis

$$out[B_i] = \bigcup_{j \in succ(B_i)} in[B_j]$$

where $succ(B_i)$ denotes successors of B_i

For reaching definitions

$$\mathtt{in}[B_i] = \bigcup_{j \in \mathtt{pred}(B_i)} \mathtt{out}[B_j]$$

where $\mathtt{pred}(B_i)$ denotes predecessors of B_i

Cyclic CFGs leads to recursive equation systems.

Can be solved iterating until a fixpoint is reached.

Solving the data flow equations

for *each node n in CFG* **do**
  $\mathtt{in}[n] := \emptyset$; $\mathtt{out}[n] := \emptyset$;
end
repeat
  for *each node n in CFG* **do**
    $\mathtt{in}'[n] := \mathtt{in}[n]$;
    $\mathtt{out}'[n] := \mathtt{out}[n]$;
    $\mathtt{in}[n] := \mathtt{use}[n] \cup (\mathtt{out}[n] \setminus \mathtt{def}[n])$;
    $\mathtt{out}[n] = \bigcup_{s \in \mathtt{succ}(n)} \mathtt{in}[s]$;
  end
until $\mathtt{in}'[n] = \mathtt{in}[n]$ *and* $\mathtt{out}'[n] = \mathtt{out}[n]$ *for all* n;

SSA (Static Single Assignment)

Liveness analysis as well as subsequent transformations are easier if the code is in a special form: each variable has only one definition.

This form is called SSA: Static Single Assignment — **statically** each variable is assigned only once (though it can be executed many times at run time, e.g. in a loop)

For *strict* SSA, every use of a variable must be dominated by its definition.

Liveness for strict SSA can be computed by finding all definition-use paths.

LLVM requires strict SSA form with respect to its registers.

Transforming into SSA — basic block

Within a single basic block, transforming to SSA is easy: every variable definition is replaced by a definition of its numbered variant, e.g.

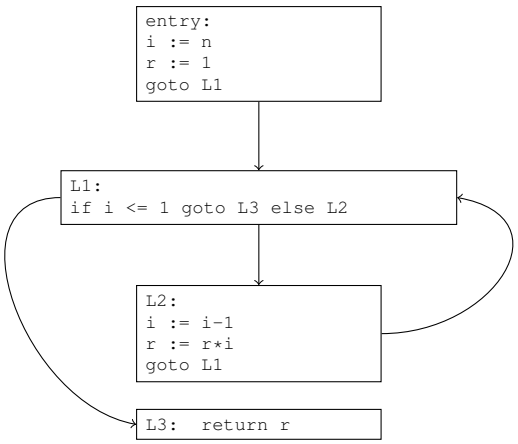
```
i := n
r := 1
r := r * i
i := i - 1
return r
```

gets replaced by

```
i1 := n
r1 := 1
r2 := r1 * i1
i2 := i1 - 1
return r2
```

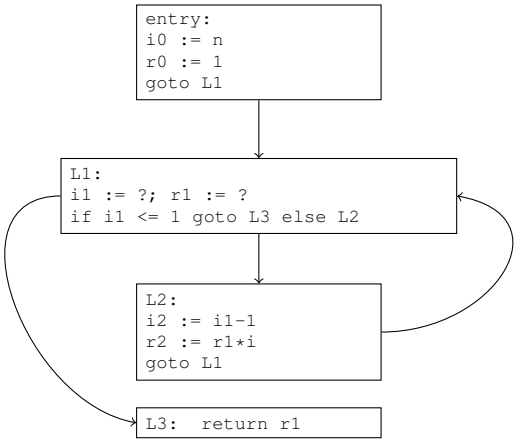
Transforming into SSA — Control Flow Graph

The nontrivial case is when the same variable is defined on two joining paths in the CFG, e.g.



Transforming into SSA — Control Flow Graph

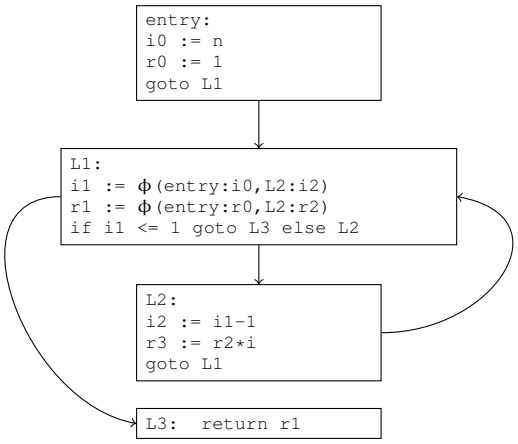
Numbering variables causes a problem



Values of the variables in the L1 block depend on the path the control reaches it.

Transforming into SSA — ϕ functions

We can encode this knowledge (and defer this problem) using a hypothetical function ϕ which chooses the proper variant of a variable:



ϕ nodes in LLVM

`phi type` $[val_1,inedge_1]\ldots[val_n,inedge_n]$

- may refer to current block (loops!), may choose `undef` for some edges
- must be at the start of a block
- order does not matter
- must have exactly one entry for each predecessor

Inserting ϕ nodes

A simple algorithm:

1. for each block with predecessors $B_1,\ldots,B_n$ $(n > 1)$
2. for each live variable v
3. at rthe start of the block
4. insert $v = \phi(B_1 : v,\ldots,B_n : v)$ (sic!)
5. systematically renumber variables

Transforming into SSA — ϕ functions in the LLVM

We could try writing our example thus:

```
entry:
        %i.0 = i32 %n
        %r.0 = i32 1
        br label %L1
L1:
        %i.1 = phi i32 [%r.0, %entry], [%i.2, %L2]
        %r.1 = phi i32 [%i.0, %entry], [%r.2, %L2]
        %c0 = icmp sle i32 %i.1, 1
        br i1 %c0, label %L3, label %L2
L2:
        %r.2 = mul i32 %r.1, %i.1
        %i.2 = sub i32 %i.1, 1
        br label %L1
L3:
        ret i32 %r.1
```

This is not correct yet — we shall see the proper code later.

Generating machine code

- Machine state: contents of registers, stack and memory
- Basing technique: simulating behaviour of the target machine (state sequences)
- We use description of the resources (mostly registers) as well as description of variables and values.

Descriptions

Register description

- State: free, occupied, reserved, etc.
- Contents (possibly many values)

Value description

- type, size
- Locations (possibly many)
- Aliases (a variable can be accessible both directly and via a pointer or reference)
- Value description only for live variables.

Assumptions about the target machine

We assume that the target machine has n general purpose registers $R_0, \ldots, R_{n-1}$, and instructions

- LOAD a, Ri or MOV a, Ri) — load the value at address a to R_i
- STORE Ri, a — store the contents of R_i at the (memory) address a
- Addresses of the form:
  - constant (e.g. global variable address)
  - constant+Rj (array element),
  - constant+FP (a local variable or a parameter, though we usually use just symbolic names here)
- op Ri, Rj — meaning Rj := Ri op Rj, where op — arithmetical operation
- op a, R — analogously for memory location a
- op $c, Ri — analogously for constant c

An aside — theory vs reality

The machine described above is close to the x86 architecture.

In reality, we cannot make even such modest assumptions about the common denominator for various processors, e.g.

- RISC architectures do not have LOAD a, Ri — only LOAD [Rj], Ri
- On x86 we can perform addition on (almost) any registers, while division only with dividend in EDX:EAX, quotient EAX and remainder in EDX (RISCs often don't have a division instruction as such)
- Most RISCs have instructions like ADD/SUB Ri, Rj, Rk. On x86, there is an instruction that can be used for a similar addition, but not substraction (guess which instruction is it?)
- etc etc

Code generation for a basic block

1. Determine variables alive at the end of the block
2. Compute *next use* for the result and each argument of every quadruple.
3. Generate code for subsequent quadruples, allocating registers "on the fly", deferring memory writes as long as possible.
4. At the end of the block write all live, but yet unwritten values.

Code generation for a single quadruple

Consider a quadruple `A := B op C`

1. Choose an instruction
2. Choose a register L to perform the operation.
3. Using descriptions, look up B's locations and if necessary emit `MOV B, L`
4. Choose C′ — on of the locations of C (preferably a register)
5. Emit `OP C', L`
6. Adjust the descriptions of A and L (value of A is only in L now).
7. If C is dead, adjust descriptions, possibly freeing a register

Sample simulation with register and value descriptions

Assume d is live at the end of the block

Value descriptions need to be kept for live variables only

Quad	Code	R0	R1	a	b	c	d	t	u	v
		-	-	a	b	c	-	-	-	-
t = a				a	b	c		a		
a = b				b	b	c		a		
b = t				b	a	c		-		
t = a-b	`R0 := b`	a		R0,b	a	c				
	`R0 -= a`	t		b	a	c		R0		
u= a-c	`R1 := b`	t	a	R1,b		c		R0		
	`R1 -= c`	t	u			c		R0	R1	
v = t+u	`R0 += R1`	v	u			-			R1	R0
d = v+u	`R0 += R1`	d	u				R0			
	`d := R0`	d	u				R0,d			

What if there is no free register (spilling)

Usually the number of registers needed to compute the program (called *register pressure*) exceeds the number of available registers.

Some values must be *spilled* to memory.

Belady's strategy (originally for VM page replacement in operating systems [Belady 1966])

1. Choose a register holding a value whose use lies farthest in the future and free it;
2. prefer clean values to dirty values;
3. if no clean value, send a dirty one to memory (spilling);
4. update value descriptions, e.g. if the register held a, b and we spill it to a, now a holds a,b.

Global register allocation

Global = across the basic blocks, usually for a whole function.

1. Allocate a pool of r registers; in a program fragment choosen values will be kept in registers permanently.
2. Build the collision graph: its vertices are variables; if at the point of definition of a, a variable b is alive, we add a collision edge (a, b) (meaning a and b can't use the same register).
3. Colour the graph with r colours.
4. The problem is NP-hard in general, hence use heuristics.
5. Can be solved in polynomial time for the code in SSA form [Pereira 2005] (though special care must be taken when deconstructing ϕ functions — [Hack,Grund,Goos 2006; Pereira,Palsberg 2009])
6. For SSA there is also a global variant of Belady's algorithm [Braun,Hack 2009]

Greedy graph colouring

Input: $G = (V, E)$ and an ordered sequence $v_1, \ldots, v_n$ of nodes. Output: Assignment $col : V \to 0, \ldots, K$.

For every v_i in the order given, choose $col(v_i)$ be the lowest colour not used in $N(v_i)$.

For every graph, there is some order for which the greedy algorithm produces the optimal colouring.

For interference graphs of strict SSA programs, this order can be computed efficiently.

Chordal graphs

A graph is called *chordal* if every cycle of length $\geqslant 4$ has a chord (shortcut), i.e. an edge that connects two vertices on the cycle, but is not a part of the cycle.

The graph on the left is chordal; the one on the right is not.

The edge 1–3 is a chord for the 1–2–3–4 cycle.

Simplicial ordering

A vertex is called simplicial if its neighbourhood is a clique.

on the left, 2 and 4 are simplicial; on the right none are

An ordering $x_1, \ldots, x_n$ of the vertices is a simplicial elimination scheme, if for every $i \in [1, n-1]$, x_i is a simplicial vertex in $G[x_{i+1}, \ldots, x_n]$.

Every chordal graph has a simplicial ordering scheme;

For chordal (SSA) interference graphs and simplicial ordering, greedy algorithms gives optimal colouring.

Maximum Cardinality Search

Every chordal graph has a simplicial ordering scheme; it can be computed using the Maximum Cardinality Search Algorithm

Algirithm MCS

1. input: $G = (V, E)$
2. output: a simplicial ordering $\sigma = v_1, \ldots, v_n$
3. Set $w(v) = 0$ for all $v \in V$
4. for i in $[1..|V|]$
   (a) choose $v \in V$ st $w(v)$ maximal in V
   (b) set $\sigma(i) = v$
   (c) for every u being a neighbour of v set $w(u) += 1$
   (d) remove v from G

Register nodes and precolouring

Some values may be tied to particular registers (e.g. division, function result)

One way to handle this is the following:

- add nodes representing registers to the interference graph
- precolour them with their colours
- precolour value nodes as needed (e.g. division result with EAX)
- add interference edges as needed (e.g. all other values live after division interfere with EDX)

Caller-save and callee-save registers

So far, we have treated all registers as (more or less) equal.
Some of them are however caller-save, others callee-save.

Callee-save registers need to be saved at the start of the fuction, but survive calls.

Caller save registers do not need to be saved up front, but before every function call they are to survive.

A simple way to limit register saves:

- prefer callee-save registers for value whose lifetime spans across function calls;
- prefer caller-save registers for other values.

Target independent optimisations

- Constant folding/propagation
- Common subexpression elimination
- Dead code elimination
- Loop optimisation

Constant folding/propagation

- Constant expressions can be replaced by their values
- Assignment of a constant to a variable can be removed by replacing all occurrences of the variable by the constant.

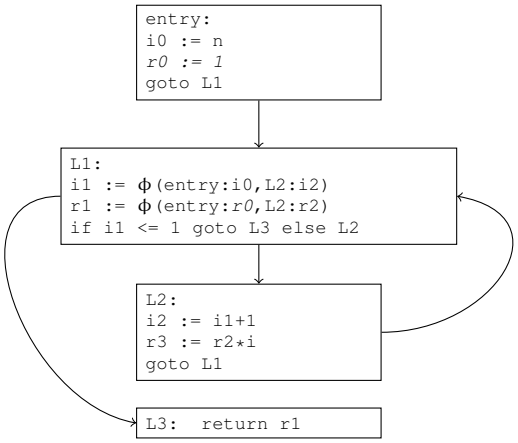
For example the sequence

```
t1 := 7
t2 := t1 - 1
t3 := t2 * t2
a := b + t3
```

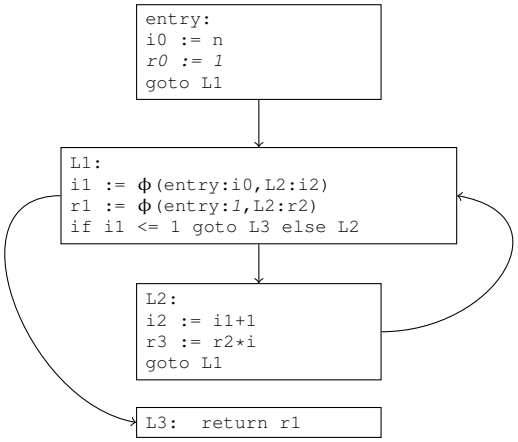
can be replaced by

```
a := b + 36
```

Constant propagation

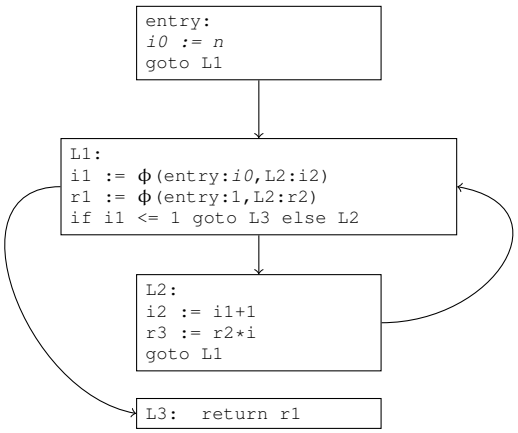


Constant propagation

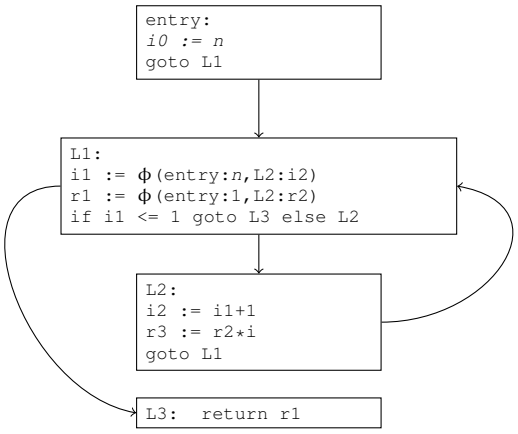


Copy propagation

Similarly, copy statement $x = y$, can be removed by replacing all uses of x reached by this definition by y (SSA helps)



Copy propagation



LLVM code

Hence the LLVM code in one of previous lectures:

```
define i32 @fact(i32 %n) {
entry: br label %L1

L1:
        %i.1 = phi i32 [%n, %entry], [%i.2, %L2]
        %r.1 = phi i32 [1, %entry], [%r.2, %L2]
        %c0 = icmp sle i32 %i.1, 1
        br i1 %c0, label %L3, label %L2

L2:
        %r.2 = mul i32 %r.1, %i.1
        %i.2 = sub i32 %i.1, 1
```

```
        br label %L1
L3:
        ret i32 %r.1
}
```

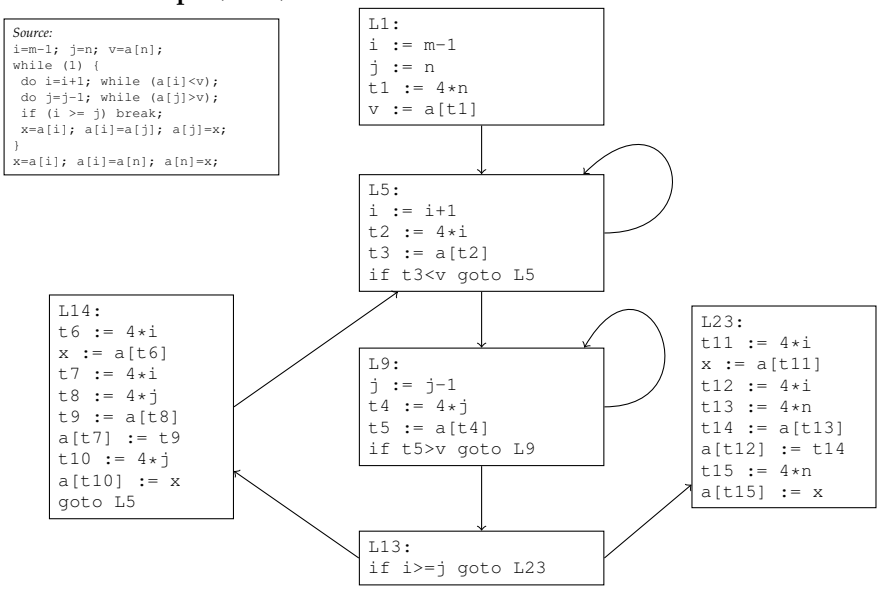
Running example

```
void quicksort(int m, int n)
{
    int i,j;
    int v,x;
    if (n<=m) return;
// === Interesting part  ===
    i=m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x =a[i]; a[i] = a[j]; a[j] = x;
   }
   x = a[i]; a[i] = a[n]; a[n] = x;
// === E N D ===
quicksort(m,j);  quicksort(i+1,n);
}
```

Quadruples

```
[ 1]  i := m-1                [16]  t7 := 4*i
[ 2]  j := n                  [17]  t8 := 4*j
[ 3]  t1 := 4*n               [18]  t9 := a[t8]
[ 4]  v := a[t1]              [19]  a[t7] := t9
[ 5]  i := i+1                [20]  t10 := 4*j
[ 6]  t2 := 4*i               [21]  a[t10] := x
[ 7]  t3 := a[t2]             [22]  goto (5)
[ 8]  if t3 < v goto (5)      [23]  t11 := 4*i
[ 9]  j := j-1                [24]  x := a[t11]
[10]  t4 := 4*j               [25]  t12 := 4*i
[11]  t5 := a[t4]             [26]  t13 := 4*n
[12]  if t5 > v goto (9)      [27]  t14 := a[t13]
[13]  if i >= j goto (23)     [28]  a[t12] := t14
[14]  t6 := 4*i               [29]  t15 := 4*n
[15]  x := a[t6]              [30]  a[t15] := x
```

Control Flow Graph (CFG)



Local Common Subexpression Elimination

Local — within a basic block

If many quadruples share a RHS, we can compute it just once:

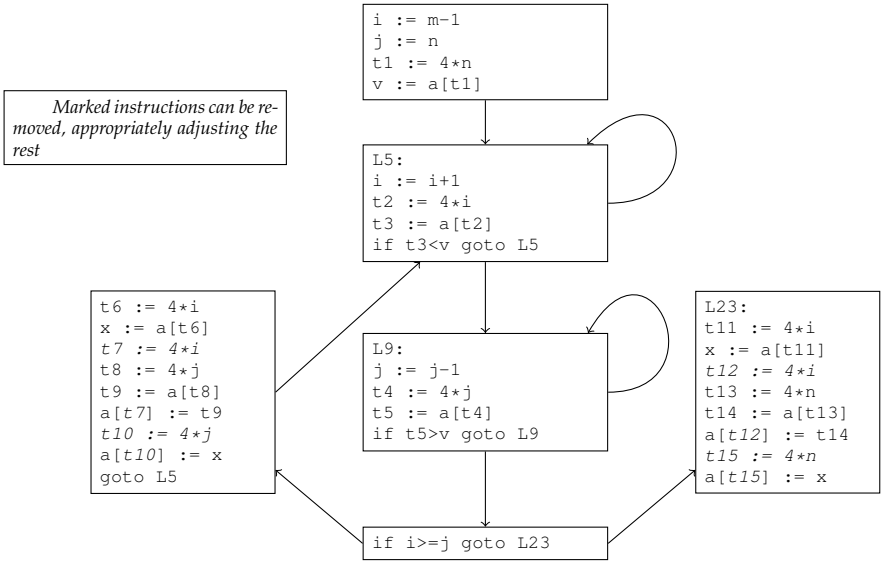
```
   t6 := 4*i
x := a[t6]
t7 := 4*i
a[t7] := y
```

can be replaced by

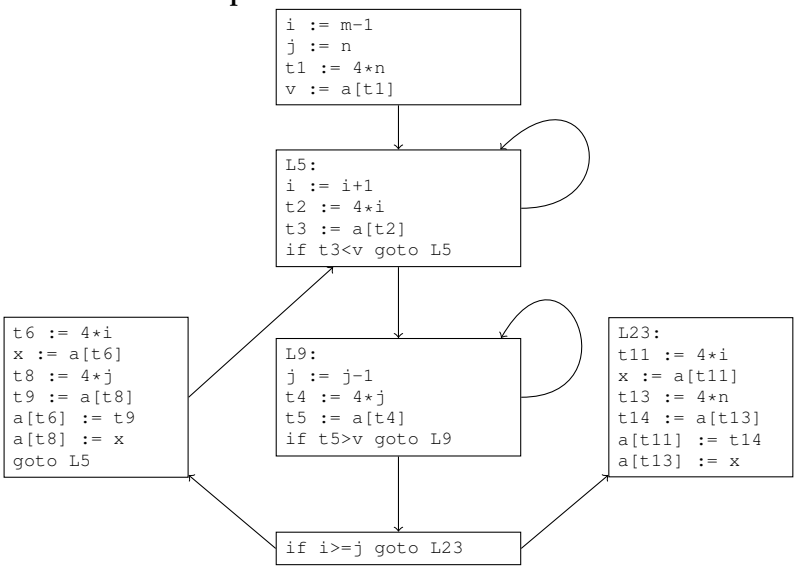
```
   t6 := 4*i
x := a[t6]
t7 := t6
a[t7] := y
```

Important that neither `t6` nor `i` change inbetween.

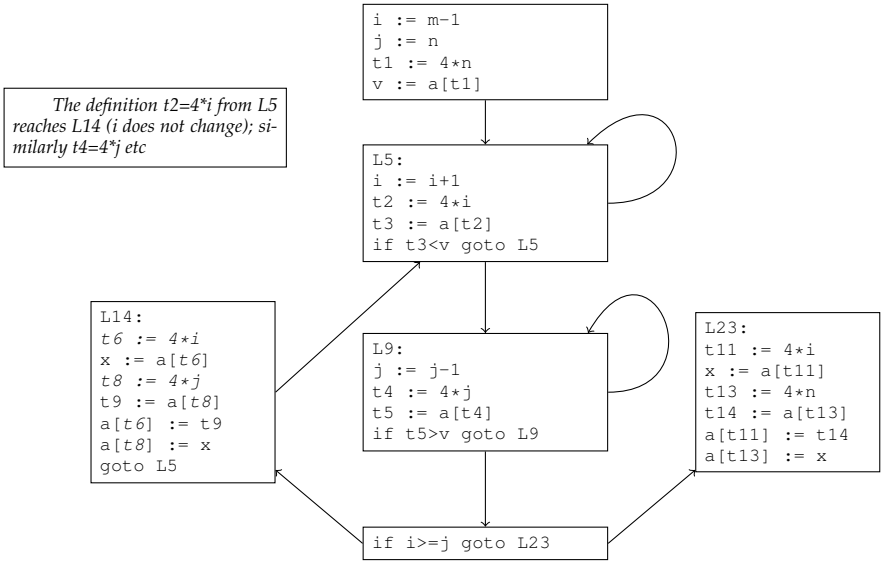
Local Common Subexpression Elimination



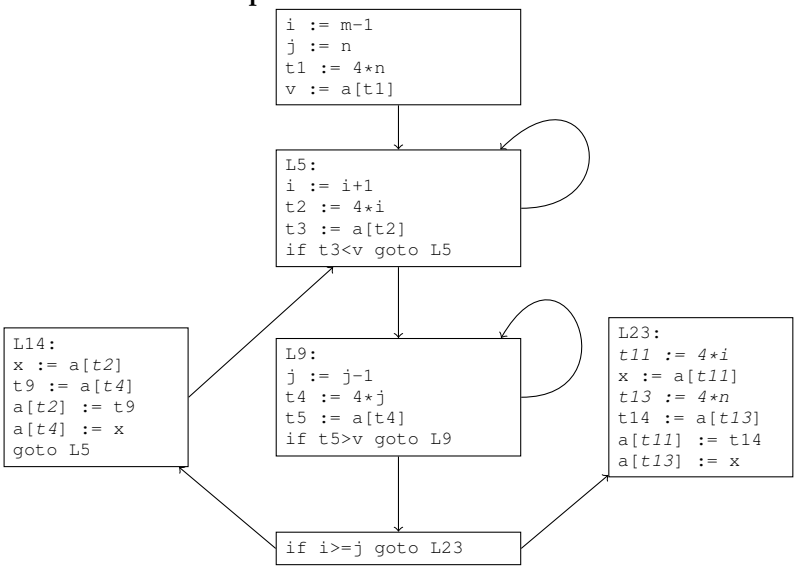
Local Common Subexpression Elimination



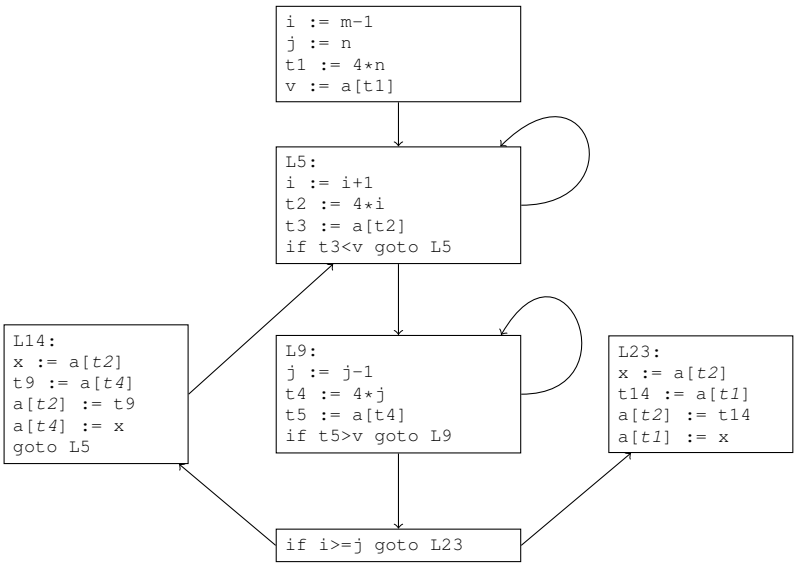
Global Common Subexpression Elimination



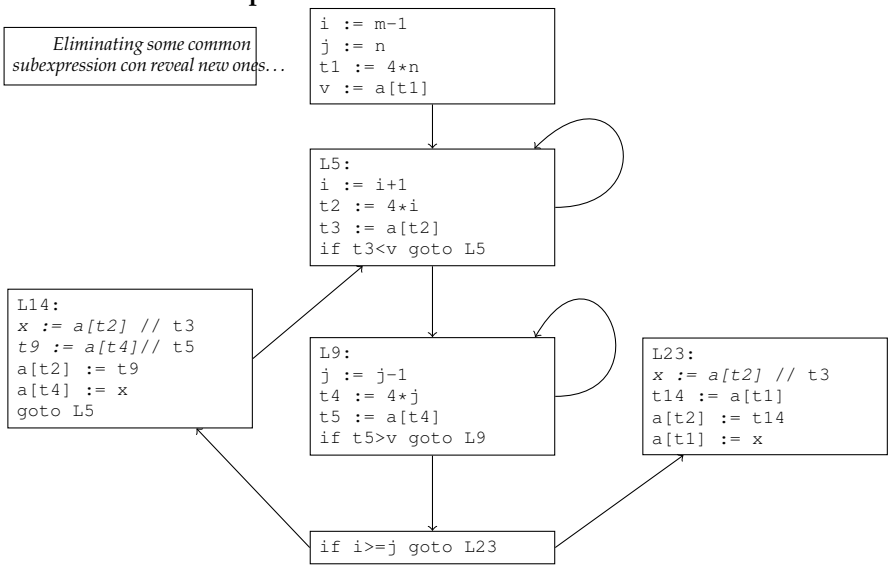
Global Common Subexpression Elimination 2



Global Common Subexpression Elimination 2



Global Common Subexpression Elimination 3



Can we use v instead of t14?

Aliasing

Between `v := a[t1]` and `t14 := a[t1]` neither `v` or `t1` change.

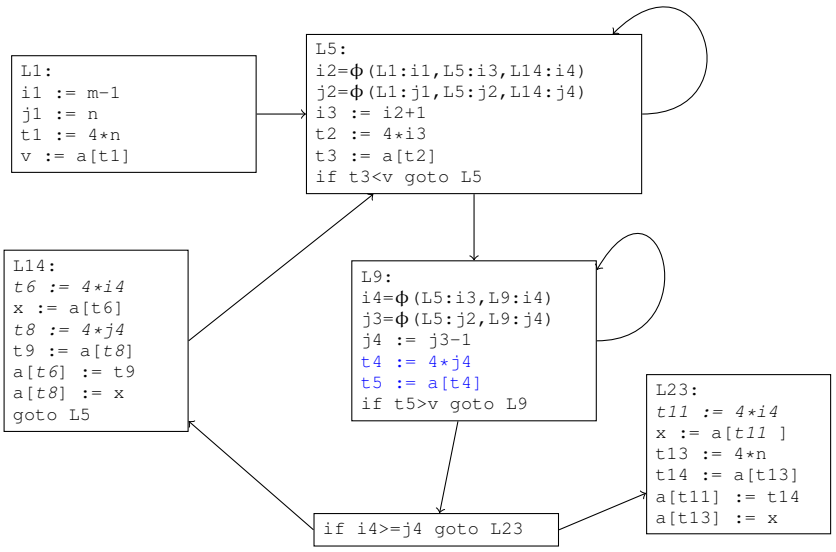
Does that mean `a[t1]` is also unchanged?

We might have accessed this cell via another index.
E.g. in `L14` there is `a[t2] := t9`

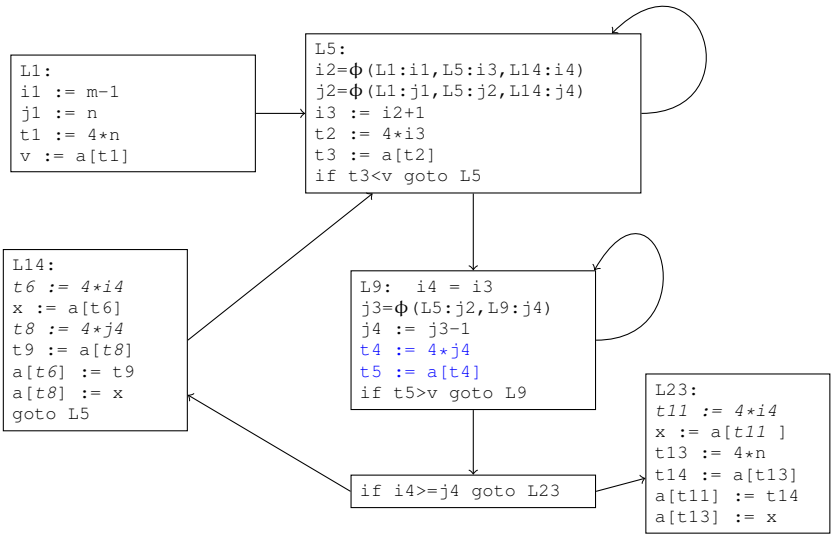
This is called *aliasing*: multiple access paths (aliases) for one memory cell.

Makes code analyses and improvement difficult.

GCSE with SSA

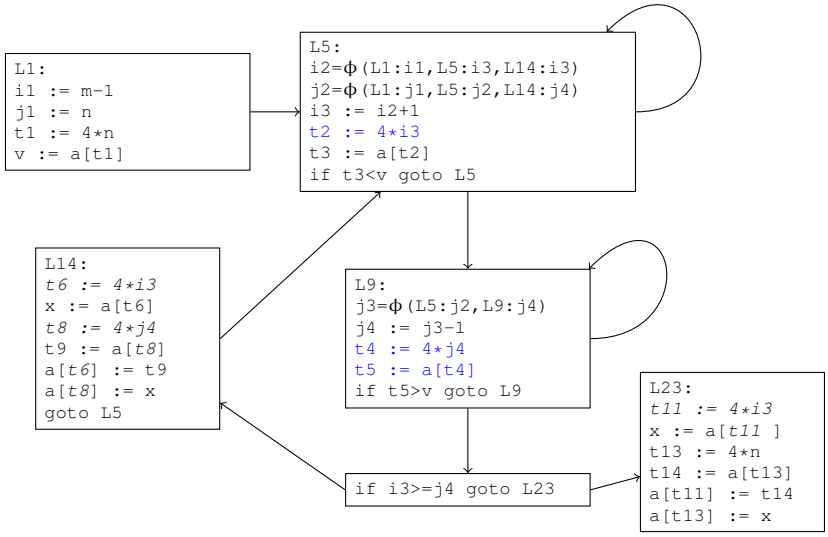


GCSE with SSA



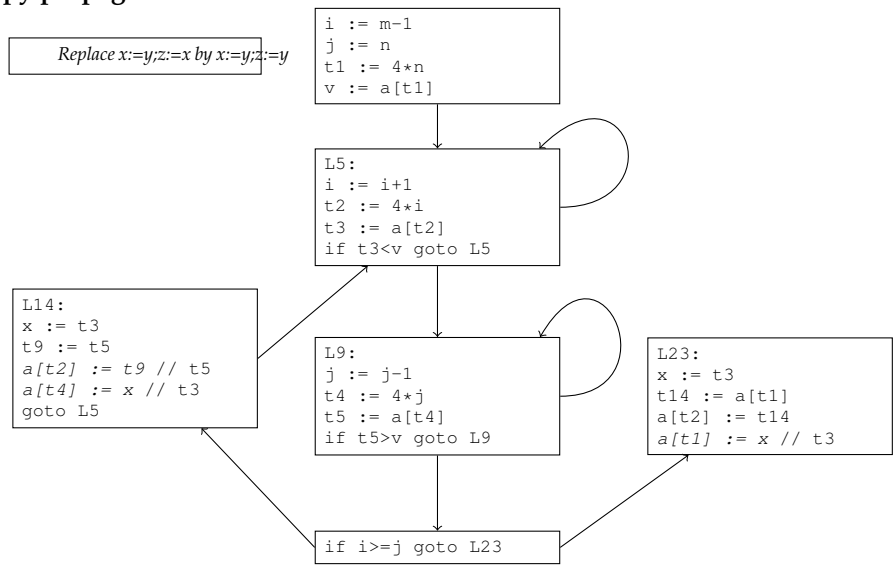
Eliminate unnecessary ϕ:

GCSE with SSA

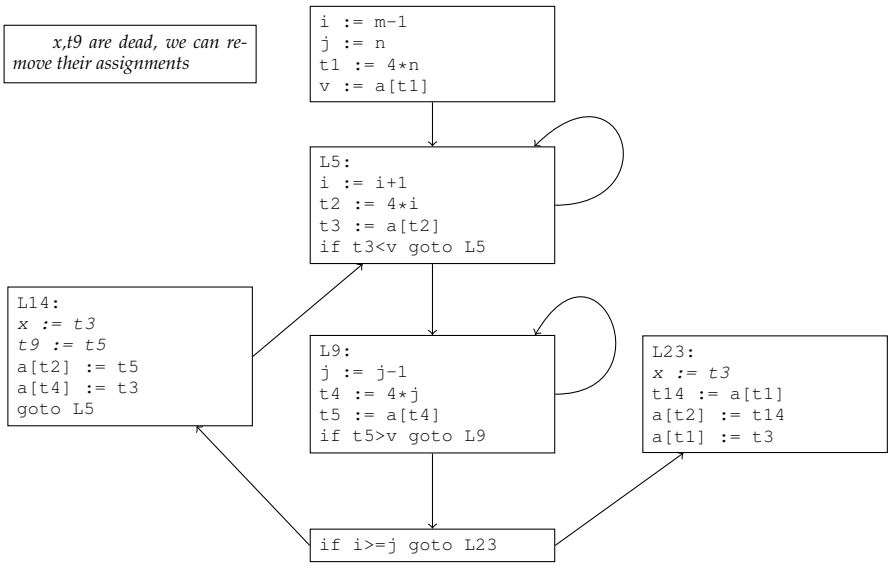


Copy propagation

Copy propagation



Dead Code Elimination



Moving code out of the loop

Loop invariant expressions (with no side-effects) can be computed before the loop, e.g.

```
while(i<=n-3) {
  s += a[i] ^ n;
  i++;
}
```

can be replaced by

```
t = n-3;
while(i<=t) {
  s += a[i] ^ n;
  i++;
}
```

Problem: register pressure may increase. If we are out of registers, we can perform an invers transform: rematerialization (trade memory read for recomputation).

Strength reduction and induction variables

- Strength reduction consists of replacing a more expensive operation (e.g. multiplication) by a cheaper one (addition)
- It's possible and useful with respect to so called *induction variables* — ones being incremented (or decremented) by the same value with each iteration.

Example

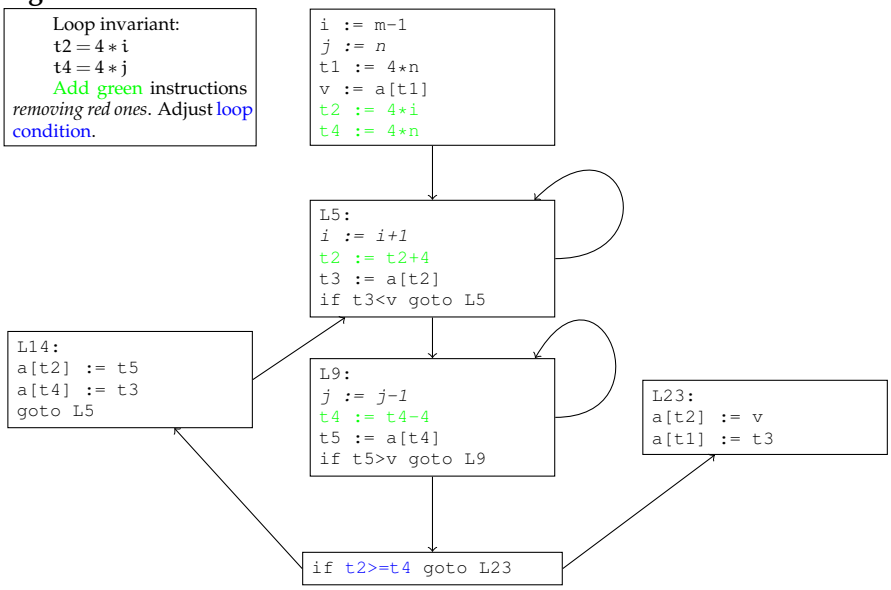
The code

```
    i := 0;
    t2 := 4*i
    goto L2
L1: i := i+1
    t2 := 4*i
    t3 := a[t2]
    s := s + t3
L2: if(a[t2]<=k) goto L1
```

can be replaced by

```
    t2 := 0;
    goto L2
L1: t2 := t2 + 4
    t3 := a[t2]
    s := s + t3
L2: if(a[t2]<=k) goto L1
```

Strength reduction and induction variables



Conclusions

- We set out with 30 quadruples, ended up with 20 (potentially cheaper).
- After code generation more optimisations (e.g. peephole) can be applied

- We get smaller and faster code.
- Price: bigger and slower compiler.
- Optimisation is error-prone (and bugs hard to find).

Peephole optimisation

- Define a set of patterns — short code sequences that can be easily improved, e.g. in the sequence

  ```
  MOV Ri, a
  MOV a, Ri
  ```

  the latter instruction is superfluous and can be removed.
- Move along the code with a small window (usually just a few instructions). If a code in the window fits one of the patterns — improve it.

Example

An assignment like `x = x + 7` can lead to the JVM code

```
iload locx
bipush 7
iadd
istore locx
```

(where `locx` is the location of x)

A peephole optimiser can recognise such a fragment and replace it with

```
iinc x 7
```

One could have avoided generating such code in the first place, but it is usually just a nuisance — peephole optimisation is simple in implementation and fast (usually just pattern matching).

Tail call optimisation

```
int factorial(int n) {
  return _factorial(n, 1);
}
int _factorial(int n, int result) {
  if (n <= 0)
    return result;
  else
     return _factorial(n - 1, n * result);
}
```

If the last instruction is a function call, it can be replaced by a jump.

If the jump is to the same function (doesn't need to be!), this is so called tail recursion.

gcc -O1

```
_factorial:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp

        movl    8(%ebp), %edx  ; edx = n
        movl    12(%ebp), %eax ; eax = result
        testl   %edx, %edx
        jle     .L2            ; if n <= 0
        imull   %edx, %eax     ; eax = n * result
        movl    %eax, 4(%esp)  ; onto the stack
        leal    -1(%edx), %eax ; eax = n-1
        movl    %eax, (%esp)   ; onto the stack
        call    _factorial
.L2:
        leave  ; restore the stack and frame pointers
        ret
```

gcc -O1 -foptimize-sibling-calls

```
        pushl   %ebp
        movl    %esp, %ebp

        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        testl   %edx, %edx
        jle     .L3
.L6:
        imull   %edx, %eax
        subl    $1, %edx
        jne     .L6
.L3:
        popl    %ebp
        ret
```

One more example of tail calls

```
int even(int n)
{
  if(!n) return 1; else return odd(n-1);
}

int odd(int n)
{
```

```
  if(n==1) return 1; else return even(n-1);
}
```

The tail calls of this kind are hard to eliminate when generating code for the JVM (whereas tail recursion can be in some cases replaced by iteration)

gcc -O1

```
even:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %edx
        movl    $1, %eax
        testl   %edx, %edx
        je      .L9
        leal    -1(%edx), %eax
        movl    %eax, (%esp)
        call    odd
.L9:
        leave
        ret
```

gcc -O1 -foptimize-sibling-calls

```
even:
        pushl    %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        testl    %eax, %eax
        je      .L12
        subl    $1, %eax
        movl    %eax, 8(%ebp)
        popl    %ebp
        jmp     odd
.L12:
        movl    $1, %eax
        popl    %ebp
        ret
```

...

```
-falign-functions[=n] -falign-jumps[=n] -falign-labels[=n] -falign-loops[=n]
-fassociative-math -fauto-inc-dec -fbranch-probabilities -fbranch-target-load-optimize
-fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcaller-saves -fcheck-data-deps
-fcprop-registers -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fcx-fortran-rules
-fcx-limited-range -fdata-sections -fdce -fdce -fdelayed-branch -fearly-inlining
-fdelete-null-pointer-checks -fdse -fexpensive-optimizations -ffast-math
-ffinite-math-only -ffloat-store -fforward-propagate -ffunction-sections
-fgcse -fgcse-after-reload -fgcse-las -fgcse-lm -fgcse-sm -fif-conversion -fif-conversion2
```

```
-finline-functions -finline-functions-called-once -finline-limit=n -finline-small-functions
-fipa-cp -fipa-marix-reorg -fipa-pta -fipa-pure-const -fipa-reference -fipa-struct-reorg
-fipa-type-escape -fivopts -fkeep-inline-functions -fkeep-static-consts
-fmerge-all-constants -fmerge-constants -fmodulo-sched -fmodulo-sched-allow-regmoves
-fmove-loop-invariants -fmudflap -fmudflapir -fmudflapth -fno-branch-count-reg
-fomit-frame-pointer -foptimize-register-move -foptimize-sibling-calls
-fpeel-loops -fpredictive-commoning -fprefetch-loop-arrays -freciprocal-math
-fregmove -frename-registers -freorder-blocks -freorder-blocks-and-partition
-freorder-functions -frerun-cse-after-loop -freschedule-modulo-scheduled-loops
-frounding-math -frtl-abstract-sequences -fsched2-use-superblocks -fsched2-use-traces
-fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns-dep[=n]
-fsched-stalled-insns[=n] -fschedule-insns -fschedule-insns2 -fsection-anchors -fsee
-fsignaling-nans -fsingle-precision-constant -fsplit-ivs-in-unroller
-fsplit-wide-types -fstack-protector -fstack-protector-all
-fstrict-aliasing -fstrict-overflow -fthread-jumps -ftracer -ftree-ccp
-ftree-ch -ftree-copy-prop -ftree-copyrename -ftree-dce
-ftree-dominator-opts -ftree-dse -ftree-fre -ftree-loop-im -ftree-loop-distribution
-ftree-loop-ivcanon -ftree-loop-linear -ftree-loop-optimize
-ftree-parallelize-loops=n -ftree-pre -ftree-reassoc -ftree-sink -ftree-sra
-ftree-store-ccp -ftree-ter -ftree-vect-loop-version -ftree-vectorize -ftree-vrp
-funit-at-a-time -funroll-all-loops -funroll-loops -funsafe-loop-optimizations
-funsafe-math-optimizations -funswitch-loops -fvariable-expansion-in-unroller
-fvect-cost-model -fvpt -fweb -fwhole-program
-O  -O0  -O1  -O2  -O3  -Os
```