Compilers - Chapter 6:

Optimisation and data-flow analysis

Part 1: Introduction to optimisation

- Lecturers:
  - Paul Kelly (p.kelly@imperial.ac.uk)

  - Naranker Dulay (n.dulay@imperial.ac.uk)

- Materials:
  - materials.doc.ic.ac.uk, Panopto
  - Textbook
  - Course web pages (http://www.doc.ic.ac.uk/~phjk/Compilers)
  - Piazza (https://piazza.com/class/kf7uelkyxk7aa)

Overview

- This introductory course has focussed so far on fast, simple techniques which generated code that works reasonably well
- We now briefly look at what *optimising* compilers do, and how they do it
- Compare "gcc file.c" versus "gcc –O file.c"
- According to the gcc manual page ("man gcc"):
  - Without `-O', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.
  - Without `-O', only variables declared "register" are allocated in registers

The plan

- To optimise or not to optimise?
- High-level vs low-level; role of analysis
- Peephole optimisation
- Local, global, interprocedural
  - Loop optimisations
  - Where optimisation fits in the compiler
    - Example: **live ranges**
      - Live ranges as a data flow problem
      - Solving the data-flow equations
      - Deriving the interference graph
    - Other data-flow analyses
    - **Loop-invariant code** and **code motion optimisations**
  - More sophisticated optimisations

This chapter

Next chapter

Optimisation: example

- Consider the loop from tutorial exercise 4:

```
void P(int i, int j)
{
  int k, tmp;
  for (k=0; k<100; k++) {
    tmp = A[i+k];
    A[i+k] = A[j+k];
    A[j+k] = tmp;
  }
}
```

- What can optimisation do here?

Without optimisation….

```
_P:
    subl $36,%esp
    pushl %ebp
    pushl %ebx
    nop
    movl $0,28(%esp)
    .align 4
L3:
    cmpl $99,28(%esp)
    jle L6
    jmp L4
    .align 4
L6:
    movl 48(%esp),%eax
    movl 28(%esp),%edx
    addl %edx,%eax
    leal 0(,%eax,4),%edx
    movl $_A,%eax
    movl (%edx,%eax),%edx
    movl %edx,24(%esp)
    movl 48(%esp),%eax
    movl 28(%esp),%ecx
    leal (%ecx,%eax),%edx
    leal 0(,%edx,4),%eax
    movl $_A,%edx
    movl 52(%esp),%ecx
    movl 28(%esp),%ebx
    addl %ebx,%ecx
    leal 0(,%ecx,4),%ebx
    movl $_A,%ecx
    movl (%ebx,%ecx),%ebx
    movl %ebx,(%eax,%edx)
    movl 52(%esp),%eax
    movl 28(%esp),%ecx
    leal (%ecx,%eax),%edx
    leal 0(,%edx,4),%eax
    movl $_A,%edx
    movl 24(%esp),%ecx
    movl %ecx,(%eax,%edx)
L5:
    incl 28(%esp)
    jmp L3
    .align 4
L4:
L2:
    popl %ebx
    popl %ebp
    addl $36,%esp
    ret
```

Without optimisation, code is large, slow, but compiles quickly and works well with the debugger

31 instructions in loop

Performance:

- 8.2ns per iteration (gcc 3.2.2, 2GHz Pentium IV)

With optimisation:

- In this extreme example, optimised code is 2-4 times faster
  - Use registers not stack
  - One jump per iteration
  - Loop-invariant offset calculation moved out
  - Array pointers incremented instead of recalculated
  - Loop control variable replaced with down-counter

```
_P: pushl %edi
    pushl %esi
    movl $99,%edi
    pushl %ebx
    movl $_A,%esi
    movl 20(%esp),%ebx
    movl 16(%esp),%ecx
    sall $2,%ebx
    sall $2,%ecx
    .align 4
L6:
    movl (%esi,%ecx),%edx
    movl (%esi,%ebx),%eax
    movl %eax,(%esi,%ecx)
    movl %edx,(%esi,%ebx)
    addl $4,%ecx
    addl $4,%ebx
    decl %edi
    jns L6
    popl %ebx
    popl %esi
    popl %edi
    ret
```

8 instructions in loop

Performance:

- 3.4ns per iteration (gcc 3.2.2, 2GHz Pentium IV)

With optimisation:

- In this extreme example, optimised code is 2-4 times faster
  - Use registers not stack
  - One jump per iteration
  - Loop-invariant offset calculation moved out
  - Array pointers incremented instead of recalculated
  - Loop control variable replaced with down-counter

```
_P:  pushl  %esi
     pushl  %ebx
     movl   12(%esp), %edx
     movl   16(%esp), %ecx
     leal   0(,%edx,4), %ebx
     subl   %edx, %ecx
     movl   %ecx, %edx
     leal   _A(%ebx), %eax
     addl   $_A+400, %ebx
L2:  movl   (%eax), %ecx
     movl   (%eax,%edx,4), %esi
     movl   %esi, (%eax)
     movl   %ecx, (%eax,%edx,4)
     addl   $4, %eax
     cmpl   %ebx, %eax
     jne    L2
     popl   %ebx
     popl   %esi
     ret
```

7 instructions in loop

- 0.7ns per iteration (gcc 5.4 –O3, 3.2GHz Intel Skylake i76600U)

With optimisation:

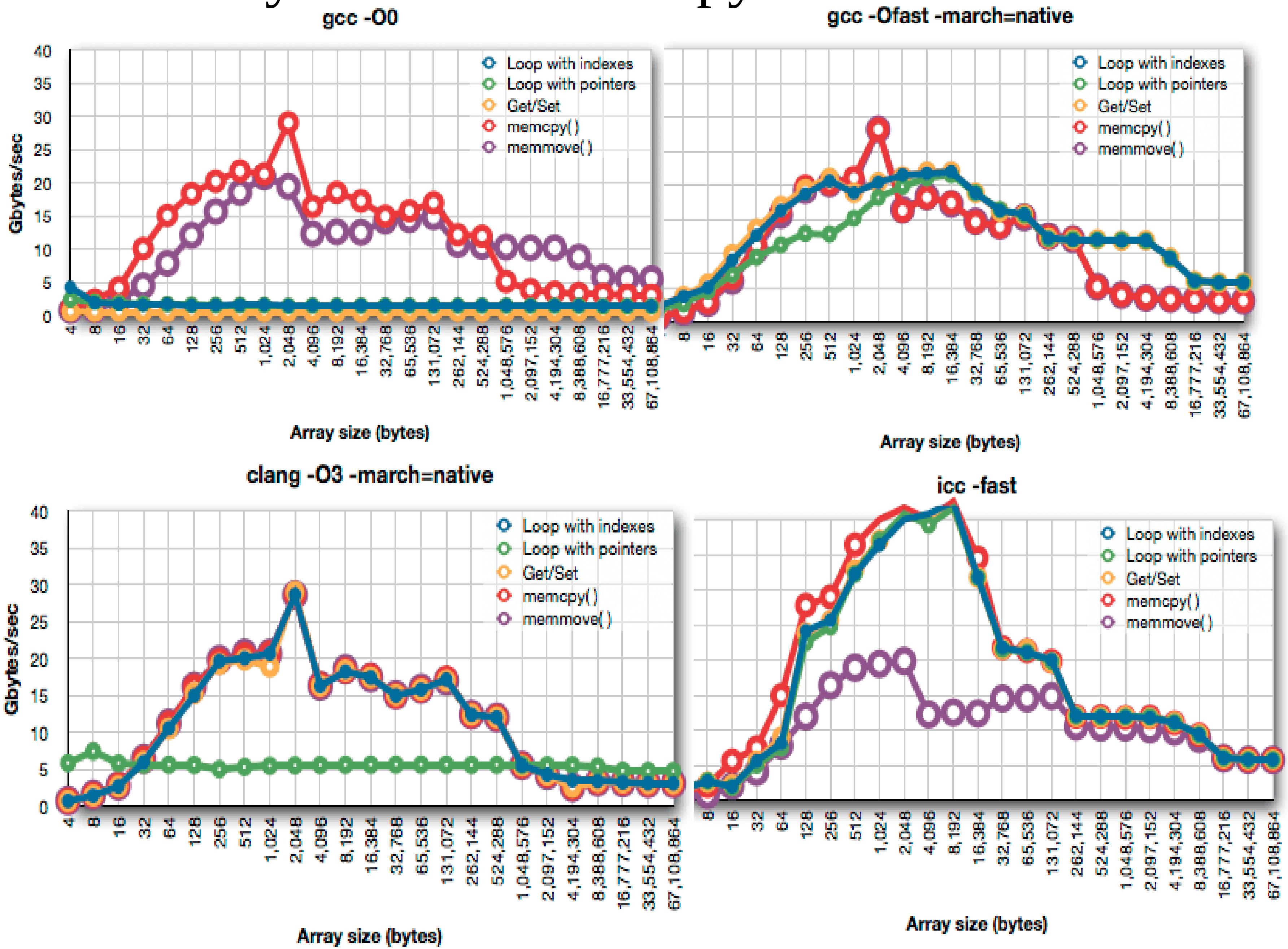
- In this code, the compiler has used vector instructions that operate on four operands at a time
- The full code is rather complicated as care is needed to check whether the memory regions overlap

- (this example goes far beyond what we can hope to cover in this course)

```
_P:  ….
     ….
.L5: movdqu  (%rdx,%rax), %xmm0
     movdqu  (%rcx,%rax), %xmm1
     movdqu  %xmm1, (%rdx,%rax)
     movdqu  %xmm0, (%rcx,%rax)
     addq    $16, %rax
     cmpq    $400, %rax
     jne     .L5
     rep ret
```

7 instructions in loop
- 0.2ns per iteration (gcc 4.8.4 –O3, –march=native, 3.2GHz Intel Skylake i7-6600U)
- Vectorised

Never write your own memcopy



http://nadeausoftware.com/articles/2012/05/c_c_tip_how_copy_memory_quickly

Optimisation principles…

- To generate really good code, need to combine many techniques, including both high-level and low-level
- High-level example: **inlining**
  - replace a call “f(x)” with the function body itself
  - Avoids call/return overheads
  - Also creates further opportunities…
  - Can we inline virtual method calls “x.f(y)”?
  - Need *static analysis* of possible types of “x”
- Low-level example: **instruction scheduling**
  - Re-order instructions so processor executes them in parallel
  - To switch order of load A[i] and store A[j], need *dependence analysis*: could i and j refer to same location?

A simple local technique – peephole optimisation

- Scan assembly code, replacing obviously inane combinations of instructions (eg mov R0,a; mov a,R0)
- Easy to implement:

```
peep :: [Instruction] -> [Instruction]
peep (Store r1 dest : Load r2 src : rest)
 | src == dest
   = Store r1 dest : (peep (Load r2 r1 : rest))
 | otherwise
   = Store r1 dest : (peep (Load r2 src : rest))
```

- Endless possibilities…
- *Phase ordering problem*: in which sequence should optimisations be applied?

Spectrum…

- Peephole optimisation works at instruction level
- The Sethi-Ullman "weights" algorithm: expressions
- "**Local**" optimisation works at the level of *basic blocks* – a sequence of instructions which has a single point of entry and a single point of exit
- "**Global**" optimisation works on a whole procedure
- **Interprocedural** optimisation works on the whole program

- Local: generally runs quickly and easy to validate
- Global: may have worse-than-linear complexity, eg O(N^2) where N is number of instructions, basic blocks, or local variables
- Interprocedural: rare – hard to avoid excessive compilation time

Some loop optimisations…

- Loop-invariant code motion
  - An instruction is **loop-invariant** if its operands can only arrive from outside the loop
  - move loop-invariant instructions into loop header
- Detection of induction variables
  - **Induction variable** is a variable which increases/decreases by a (loop-invariant) constant on each iteration
- **Strength reduction**: calculate induction variable by incrementing, instead of by multiplying other induction variables
- **Control variable selection**: replace loop control variable with one of the induction variables actually used in the loop

Loop optimisations - example

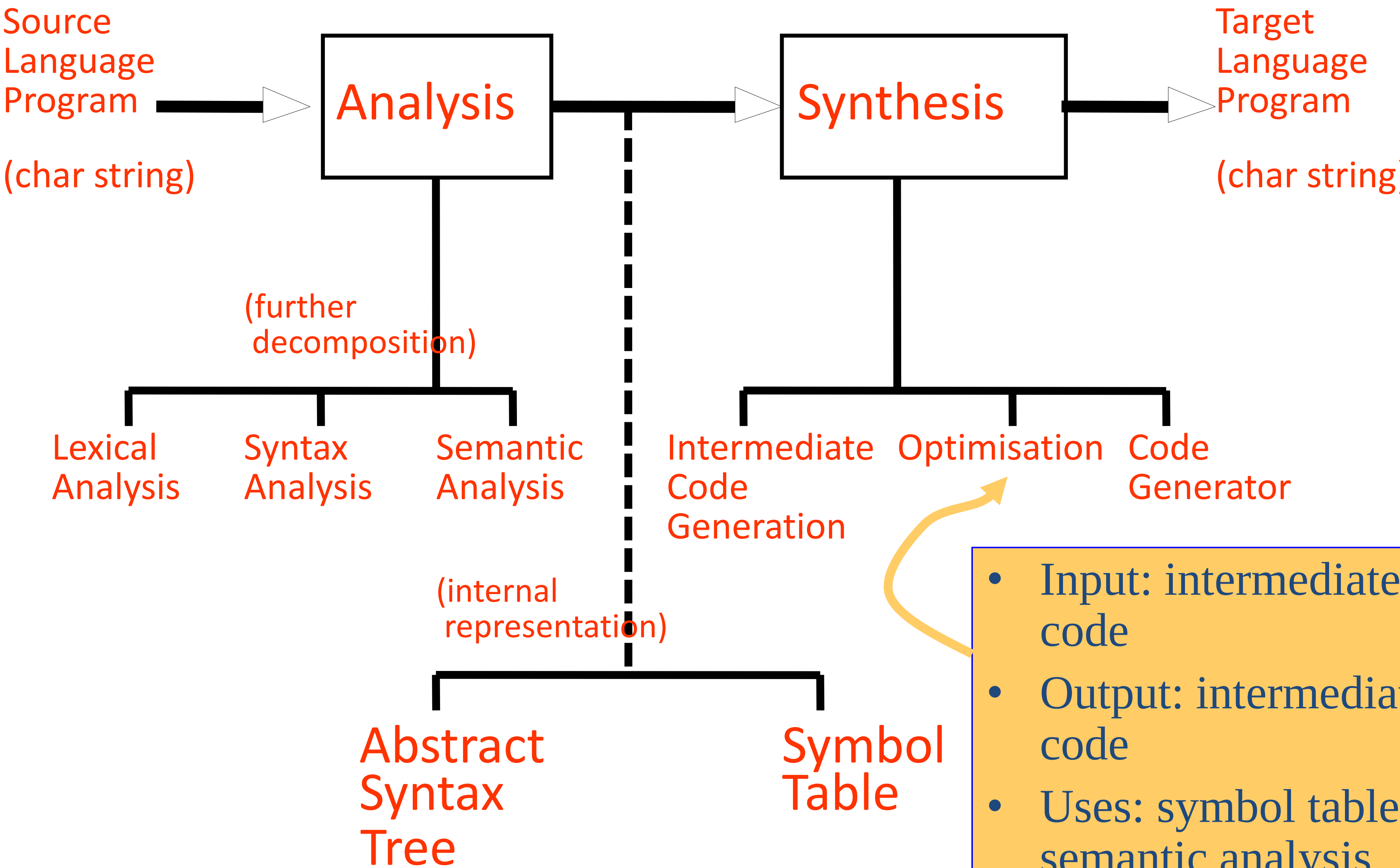
```
int P(int N, int M)
{
  int i, u, v, w, x, y;
  int z = 0;

  for (i=0; i<N; i++) {
    w = w+10;
    x = w*10;
    y = z*(w-x);
    u = w+x+y+N+M;
    v = v+u;
  }
  return v;
}
```

1. y is constant
2. w-x is dead code
3. y+N+M is loop-invariant
4. i, w and x are induction variables (so is w+x)
5. x increases by 100 each iteration
6. i is used only to control the loop, and can be omitted if convenient

1. (constant propagation Appel pg457)
2. (dead code elimination pg457,397)
3. (loop-invariant code motion pg422)
4. (induction variable recognition pg426)
5. (strength reduction ditto)
6. (rewriting comparisons, pg428)

Where does optimisation happen?

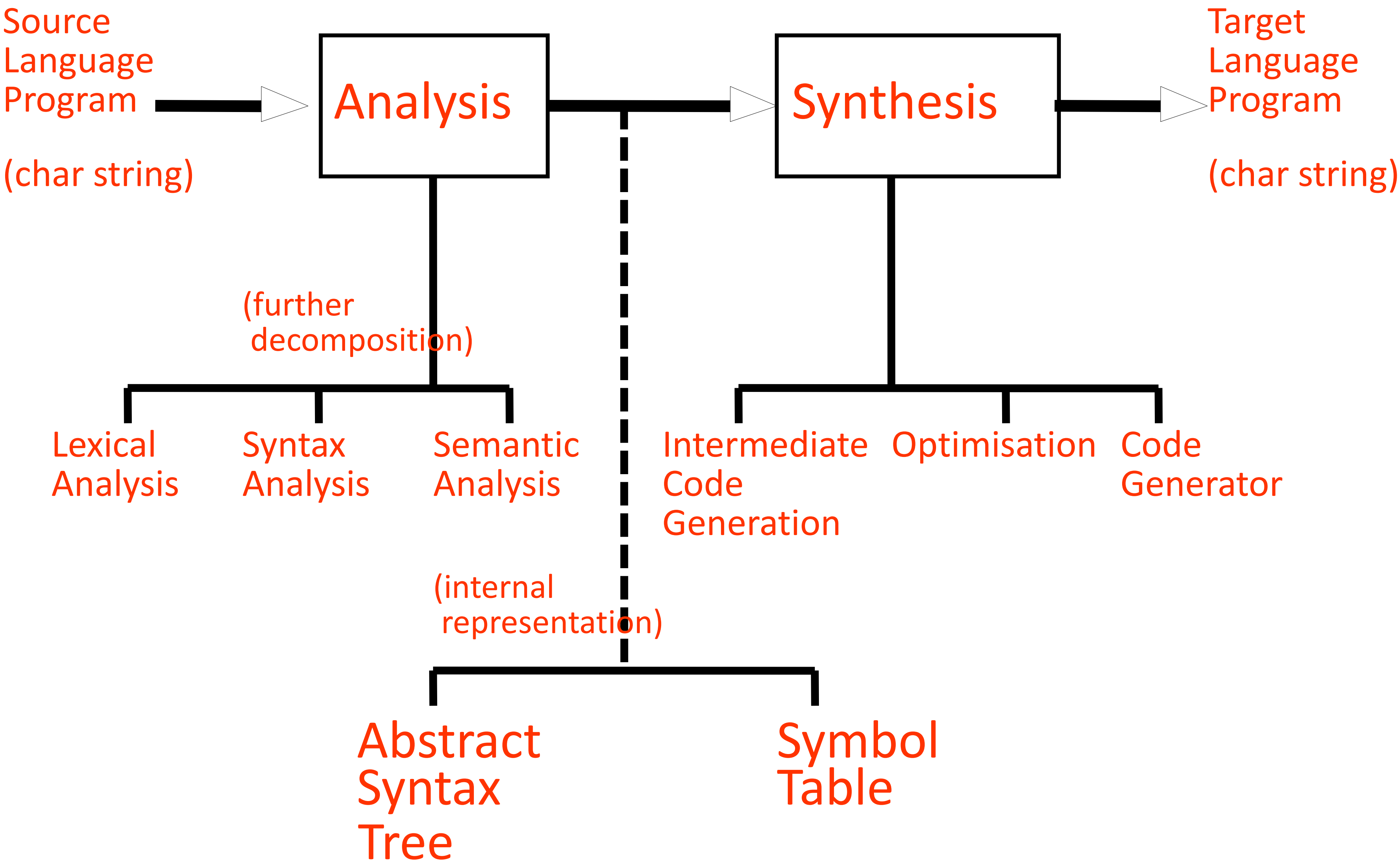


- Input: intermediate code
- Output: intermediate code
- Uses: symbol table, semantic analysis

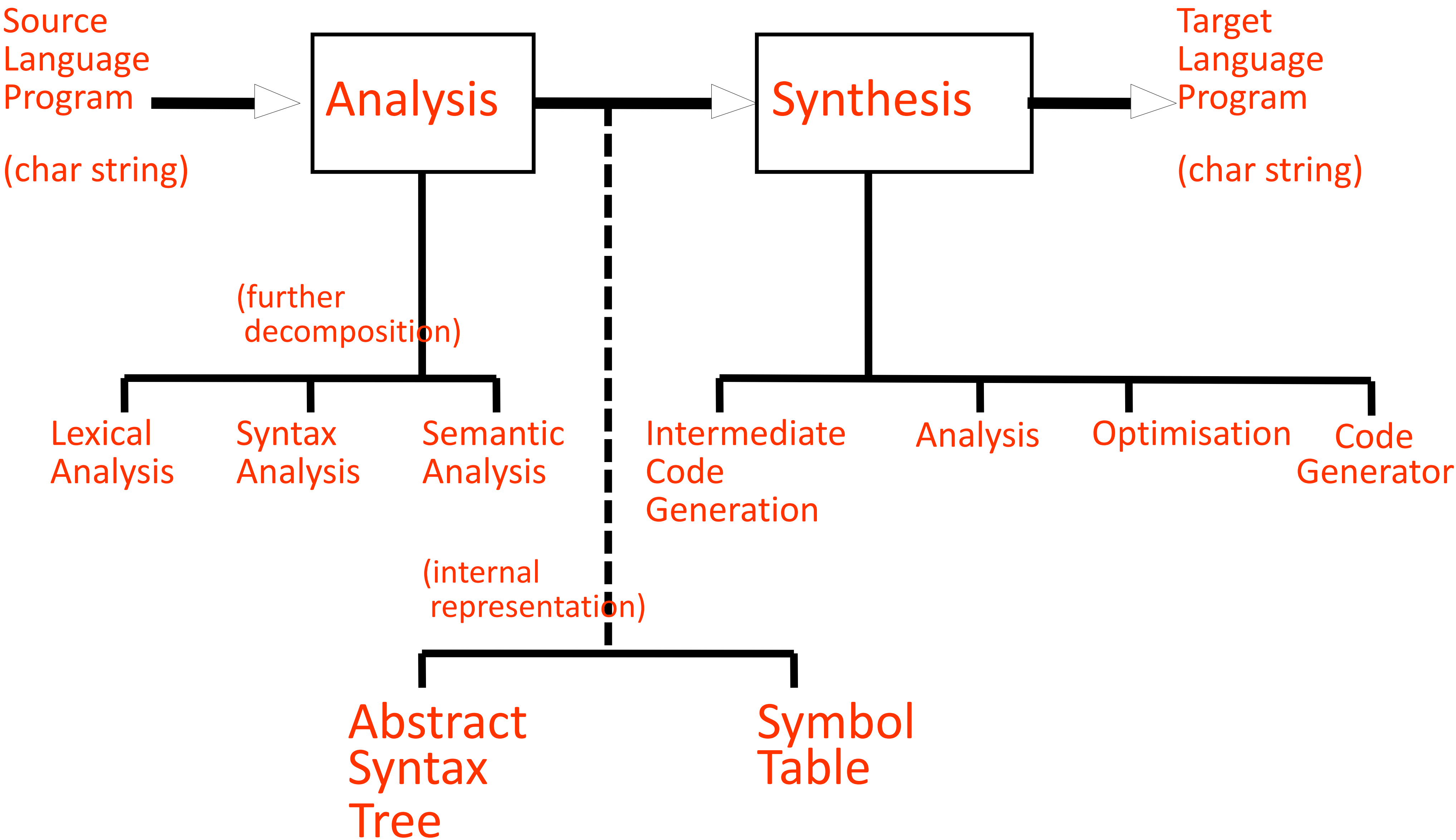
Intermediate code

- In our simple compiler, translator traverses AST and produces assembler code directly
- In optimising compiler, translator traverses AST and produces “intermediate code”
- Intermediate code is designed to
  - Represent all primitive operations necessary to execute program
  - In a uniform way, easy to analyse and manipulate
  - Independently of target instruction set
- Compiler writers argue… Appel advocates two IRs:
    - Tree: before instruction selection
    - FlowGraph: after instruction selection
- IR uses “temporaries” T0, T1, T2… instead of real registers; after optimisation, use graph colouring to assign temporaries to real registers

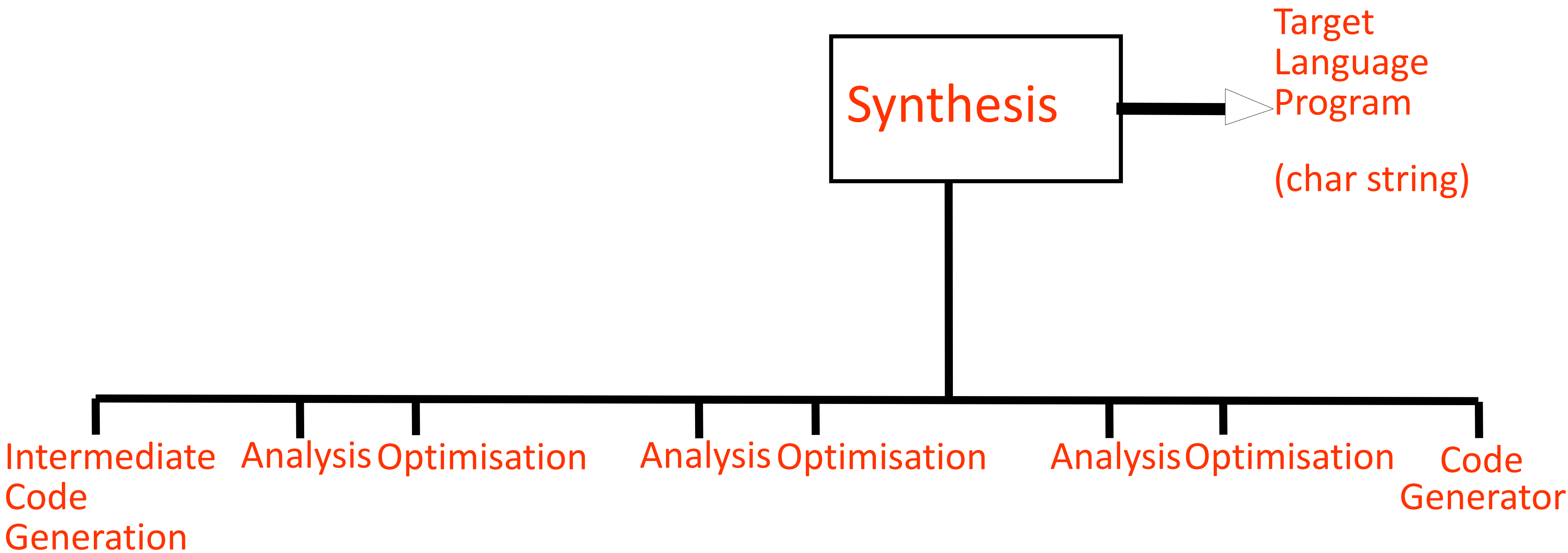
Where does optimisation happen?



Where does optimisation happen?



Where does optimisation happen?



Intermediate representations

Intermediate Code Generation | Analysis | Optimisation | Analysis | Optimisation | Analysis | Optimisation | Code Generator

Successively-lowered representations

For example in GCC:

- *"GENERIC": a tree representation common to all GCC front-end languages*
- *"GIMPLE": three-address-code tree-based representation*
- *"Low-level GIMPLE": linear control flow, explicit exceptions*
- *"SSA GIMPLE": Static-single-assignment – variables are renamed so that uses are reached by exactly one definition*
- *"Register-Transfer Language": low-level representation from which instructions are selected*

To see, try "gcc -fdump-tree-all file.c"
Or on Compiler Explorer:
https://godbolt.org/z/78qd4r for GIMPLE
https://godbolt.org/z/7WW4vT for RTL

(for interest – beyond examinable scope of the course

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

With `-O`, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

`-O` turns on the following optimization flags:

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time
```

Extract from gcc's documentation showing which optimisations are activated by the "-O" flag

(for interest – beyond examinable scope of the course

https://gcc.g

With -O, the compiler tries to reduce code … on time.

-O turns on the following optimization flag…

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time
```

github.com/gcc-mirror/gcc/blob/master/gcc/opts.c

```
static const struct default_options default_options_table[] =
  {
    /* -O1 and -Og optimizations.  */
    { OPT_LEVELS_1_PLUS, OPT_fcombine_stack_adjustments, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fcompare_elim, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fcprop_registers, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fdefer_pop, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fforward_propagate, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fguess_branch_probability, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fipa_profile, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fipa_pure_const, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fipa_reference, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fipa_reference_addressable, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fmerge_constants, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fomit_frame_pointer, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_freorder_blocks, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fshrink_wrap, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fsplit_wide_types, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_builtin_call_dce, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_ccp, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_ch, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_coalesce_vars, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_copy_prop, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_dce, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_dominator_opts, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_fre, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_sink, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_slsr, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_ter, NULL, 1 },

    /* -O1 (and not -Og) optimizations.  */
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fbranch_count_reg, NULL, 1 },
#if DELAY_SLOTS
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fdelayed_branch, NULL, 1 },
#endif
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fdse, NULL, 1 },
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fif_conversion, NULL, 1 },
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fif_conversion2, NULL, 1 },
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_finline_functions_called_once, NULL, 1 },
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fmove_loop_invariants, NULL, 1 },
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fssa_phiopt, NULL, 1 },
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fipa_modref, NULL, 1 },
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_ftree_bit_ccp, NULL, 1 },
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_ftree_dse, NULL, 1 },
    { OPT_LEVELS_1_PLUS_NOT
    { OPT_LEVELS_1_PLUS_NOT
```

"-O1" flags enables selected optimisation passes

(for interest – beyond examinable scope of the course

Small extract from gcc's "passes.def", which defines which analyses and optimisations are activated, in what order

https://gcc.g

With -O, the compiler tries to reduce code

-O turns on the following optimization flag

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time
```

github.com/gcc-mirror/gcc/blob/mast

```
static const struct default_options default_options_t
  {
    /* -O1 and -Og optimizations.  */
    { OPT_LEVELS_1_PLUS, OPT_fcombine_stack_adjustmer
    { OPT_LEVELS_1_PLUS, OPT_fcompare_elim, NULL, 1 }
    { OPT_LEVELS_1_PLUS, OPT_fcprop_registers, NULL,
    { OPT_LEVELS_1_PLUS, OPT_fdefer_pop, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fforward_propagate, NULL
    { OPT_LEVELS_1_PLUS, OPT_fguess_branch_probabilit
    { OPT_LEVELS_1_PLUS, OPT_fipa_profile, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fipa_pure_const, NULL, 1
    { OPT_LEVELS_1_PLUS, OPT_fipa_reference, NULL, 1
    { OPT_LEVELS_1_PLUS, OPT_fipa_reference_addressab
    { OPT_LEVELS_1_PLUS, OPT_fmerge_constants, NULL,
    { OPT_LEVELS_1_PLUS, OPT_fomit_frame_pointer, NUL
    { OPT_LEVELS_1_PLUS, OPT_freorder_blocks, NULL, 1
    { OPT_LEVELS_1_PLUS, OPT_fshrink_wrap, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_fsplit_wide_types, NULL,
    { OPT_LEVELS_1_PLUS, OPT_ftree_builtin_call_dce,
    { OPT_LEVELS_1_PLUS, OPT_ftree_ccp, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_ch, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_coalesce_vars, NUL
    { OPT_LEVELS_1_PLUS, OPT_ftree_copy_prop, NULL, 1
    { OPT_LEVELS_1_PLUS, OPT_ftree_dce, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_dominator_opts, NU
    { OPT_LEVELS_1_PLUS, OPT_ftree_fre, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_sink, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_slsr, NULL, 1 },
    { OPT_LEVELS_1_PLUS, OPT_ftree_ter, NULL, 1 },

    /* -O1 (and not -Og) optimizations.  */
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fbranch_count_
#if DELAY_SLOTS
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fdelayed_branc
#endif
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fdse, NULL, 1
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fif_conversio
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fif_conversion
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_finline_functi
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fmove_loop_inv
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fssa_phiopt, N
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_fipa_modref, N
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_ftree_bit_ccp,
    { OPT_LEVELS_1_PLUS_NOT_DEBUG, OPT_ftree_dse, NUL
    { OPT_LEVELS_1_PLUS_NOT
    { OPT_LEVELS_1_PLUS_NOT
```

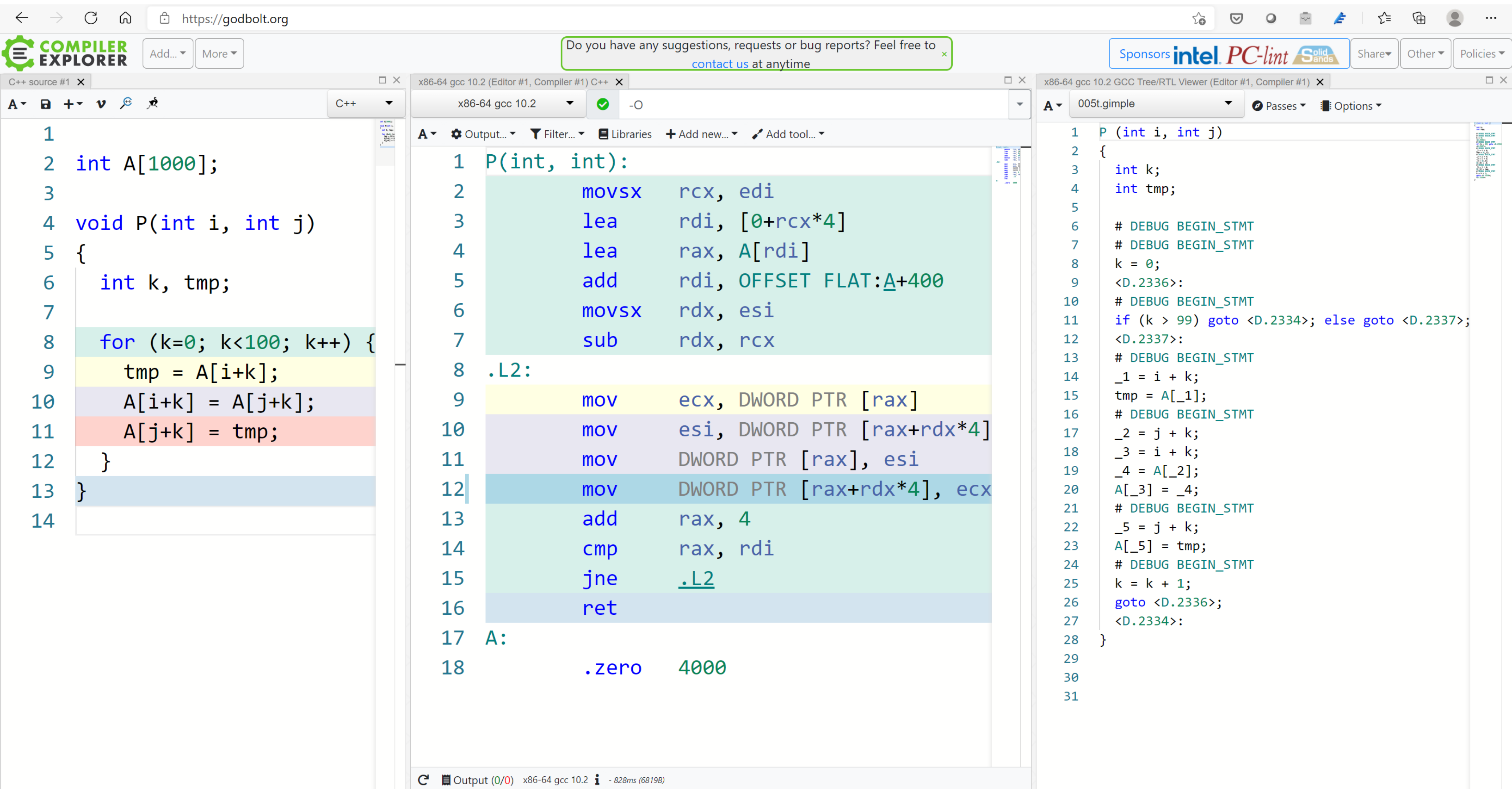
https://github.com/gcc-mirror/gcc/blob/master/gcc/passes.def

```
PUSH_INSERT_PASSES_WITHIN (pass_rest_of_compilation)
    NEXT_PASS (pass_instantiate_virtual_regs);
    NEXT_PASS (pass_into_cfg_
    NEXT_PASS (pass_jump);
    NEXT_PASS (pass_lower_sub
    NEXT_PASS (pass_df_initia
    NEXT_PASS (pass_cse);
    NEXT_PASS (pass_rtl_fwpro
    NEXT_PASS (pass_rtl_cprop
    NEXT_PASS (pass_rtl_pre);
    NEXT_PASS (pass_rtl_hoist
    NEXT_PASS (pass_rtl_cprop
    NEXT_PASS (pass_rtl_store
    NEXT_PASS (pass_cse_after
    NEXT_PASS (pass_rtl_ifcvt
    NEXT_PASS (pass_reginfo_i
    /* Perform loop optimizat
       sooner, but we want th
       efficiently.  */
    NEXT_PASS (pass_loop2);
    PUSH_INSERT_PASSES_WITHIN (pass_loop2)
        NEXT_PASS (pass_rtl_loop_init);
        NEXT_PASS (pass_rtl_move_loop_invariants);
        NEXT_PASS (pass_rtl_unroll_loops);
        NEXT_PASS (pass_rtl_doloop);
        NEXT_PASS (pass_rtl_loop_done);
    POP_INSERT_PASSES ()
    NEXT_PASS (pass_lower_subreg2);
    NEXT_PASS (pass_web);
    NEXT_PASS (pass_rtl_cprop);
    NEXT_PASS (pass_cse2);
    NEXT_PASS (pass_rtl_dse1);
    NEXT_PASS (pass_rtl_fwprop_addr);
    NEXT_PASS (pass_inc_dec);
    NEXT_PASS (pass_initialize_regs);
    NEXT_PASS (pass_ud_rtl_dce);
```

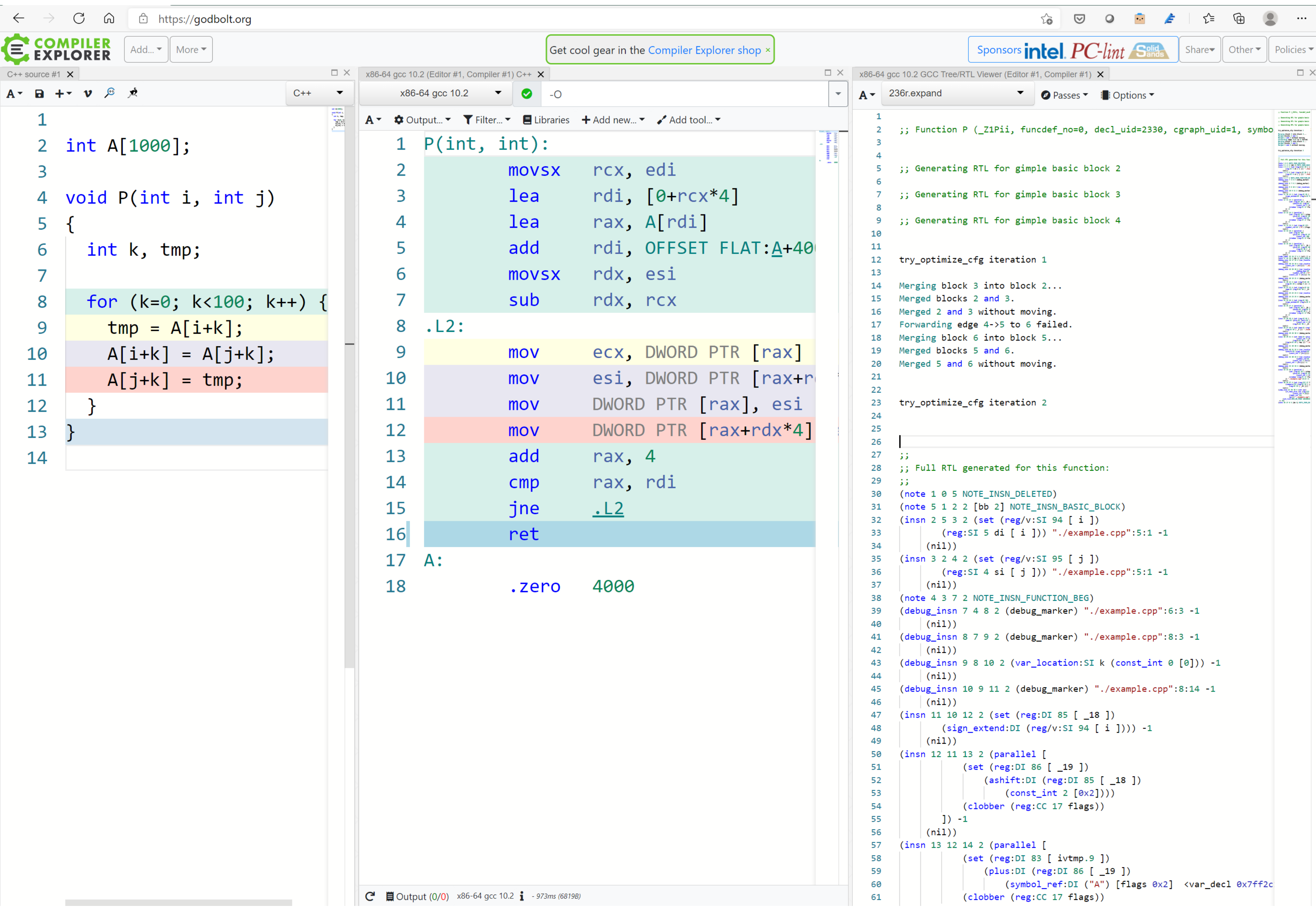
(for interest – beyond examinable scope of the course

Summary

- Optimisations consist of analyses and transformations
- Key optimisations include common sub-expression elimination, loop-invariant code motion, induction variable selection, strength reduction, dead code elimination (there are many more)
- Low-level optimisations: instruction selection, instruction scheduling, register allocation
- High-level optimisations: function inlining, loop unrolling – often *enable* other optimisations
  - The ***phase ordering problem*** is the challenge of finding the right order in which to apply optimisations
- ***Intermediate representations*** (IRs) are designed to make analyses and optimisations easy
- Compilers successively ***lower*** high-level IR to low-level IR
- Optimisation algorithms that work at the function level may have worse-than-linear time complexity
  - But inter-procedural, whole-program ("link time") optimisations need to be O(n)



To see GCC's intermediate representations for yourself, try "gcc -fdump-tree-all file.c"
Or on Compiler Explorer:
https://godbolt.org/z/78qd4r for GIMPLE (shown above)
https://godbolt.org/z/7WW4vT for RTL (next slide)


https://godbolt.org
COMPILER EXPLORER
Add...
More
Get cool gear in the Compiler Explorer shop
Sponsors intel PC-lint
Share
Other
Policies
C++ source #1
C++
int A[1000];

void P(int i, int j)
{
  int k, tmp;

  for (k=0; k<100; k++) {
    tmp = A[i+k];
    A[i+k] = A[j+k];
    A[j+k] = tmp;
  }
}
x86-64 gcc 10.2 (Editor #1, Compiler #1) C++
x86-64 gcc 10.2
-O
Output...
Filter...
Libraries
Add new...
Add tool...
P(int, int):
movsx rcx, edi
lea rdi, [0+rcx*4]
lea rax, A[rdi]
add rdi, OFFSET FLAT:A+40
movsx rdx, esi
sub rdx, rcx
.L2:
mov ecx, DWORD PTR [rax]
mov esi, DWORD PTR [rax+r
mov DWORD PTR [rax], esi
mov DWORD PTR [rax+rdx*4]
add rax, 4
cmp rax, rdi
jne .L2
ret
A:
.zero 4000
Output (0/0) x86-64 gcc 10.2 - 973ms (6819B)
x86-64 gcc 10.2 GCC Tree/RTL Viewer (Editor #1, Compiler #1)
236r.expand
Passes
Options
;; Function P (_Z1Pii, funcdef_no=0, decl_uid=2330, cgraph_uid=1, symbo
;; Generating RTL for gimple basic block 2
;; Generating RTL for gimple basic block 3
;; Generating RTL for gimple basic block 4
try_optimize_cfg iteration 1
Merging block 3 into block 2...
Merged blocks 2 and 3.
Merged 2 and 3 without moving.
Forwarding edge 4->5 to 6 failed.
Merging block 6 into block 5...
Merged blocks 5 and 6.
Merged 5 and 6 without moving.
try_optimize_cfg iteration 2
;;
;; Full RTL generated for this function:
;;
(note 1 0 5 NOTE_INSN_DELETED)
(note 5 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 2 5 3 2 (set (reg/v:SI 94 [i])
(reg:SI 5 di [i])) "./example.cpp":5:1 -1
(nil))
(insn 3 2 4 2 (set (reg/v:SI 95 [j])
(reg:SI 4 si [j])) "./example.cpp":5:1 -1
(nil))
(note 4 3 7 2 NOTE_INSN_FUNCTION_BEG)
(debug_insn 7 4 8 2 (debug_marker) "./example.cpp":6:3 -1
(nil))
(debug_insn 8 7 9 2 (debug_marker) "./example.cpp":8:3 -1
(nil))
(debug_insn 9 8 10 2 (var_location:SI k (const_int 0 [0])) -1
(nil))
(debug_insn 10 9 11 2 (debug_marker) "./example.cpp":8:14 -1
(nil))
(insn 11 10 12 2 (set (reg:DI 85 [_18])
(sign_extend:DI (reg/v:SI 94 [i]))) -1
(nil))
(insn 12 11 13 2 (parallel [
(set (reg:DI 86 [_19])
(ashift:DI (reg:DI 85 [_18])
(const_int 2 [0x2])))
(clobber (reg:CC 17 flags))
]) -1
(nil))
(insn 13 12 14 2 (parallel [
(set (reg:DI 83 [ivtmp.9])
(plus:DI (reg:DI 86 [_19])
(symbol_ref:DI ("A") [flags 0x2] <var_decl 0x7ff2c
(clobber (reg:CC 17 flags))

Feeding curiosity

- The idea of automatically deriving the instruction selector from the definition of the instruction set dates back to a landmark paper by Susan Graham and Stephen Glanville, "A new method for compiler code generation" (POPL78, https://dl.acm.org/doi/10.1145/512760.512785). The algorithm works as a bottom-up (shift-reduce) parser – using the table contruction ideas you have learned about.
- There is a wonderful book "20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection" full of good things (https://dblp.org/db/conf/pldi/pldi2004best.html) including:
  - "Automatic generation of peephole optimizations" (Davidson and Fraser, https://dl.acm.org/doi/10.1145/989393.989407): peephole optimisers don't have to be ad-hoc. You can use the automatic instruction selection mechanism to translate instruction sequences *back* to IR, and *regenerate* them – and then use this to generate peephole optimisation rules. See also Souper (https://github.com/google/souper).
    - If you've formalised the ISA, you should be able to *prove* the correctness of peephole optimisations – see "Provably correct peephole optimizations with ALIVE"(Nuno Lopes et al, PLDI'15, https://dl.acm.org/doi/10.1145/2737924.2737965)
  - "Global register allocation at link time" (David Wall, https://dl.acm.org/doi/10.1145/989393.989415). Instead of having a fixed ABI to determine which registers can be used in each function, look at the whole program to find all the call sites.