

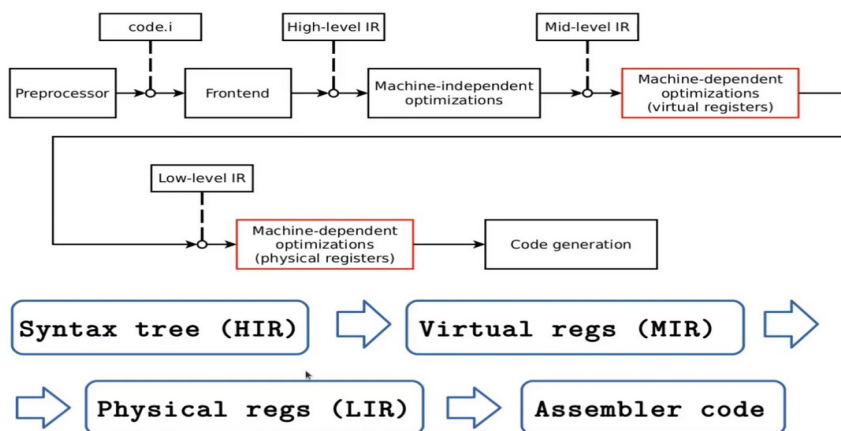
Фронтенд (упрощённая схема)



Фронтенд это небольшая и не самая сложная часть компилятора.

Что после фронтенда ? **Бекенд** компилятора, который начинается High Level Intermediate Representation, он же HIR в дальнейшем.

Что после фронтенда?



1. Machine-independent optimizations Hir

Оптимизации, который не зависят от архитектуры. Пример inline, когда мы делаем inline подстановку функции, нам все равно, как этот код будет исполняться.

Большая часть интересных оптимизаций в компиляторе, они **не** платформенно специфичны.

Например убрать недостижимый код, это **не** платформенно специфично. Оптимизации из данного уровня можно сделать для всех машин и это хорошо, чем больше мы таких оптимизаций найдем, тем меньше кода писать под конкретную архитектуру.

У компилятора много голов, но и много хвостов с конкретной архитектурой, смотри правую часть на картинке.



Машинно зависимые оптимизации, это оптимизации, которые приходится писать отдельно для каждой архитектуры.

2. Machine-dependent optimizations (virtual registers)

Грубо говоря, это представление, когда регистровый файл бесконечный и мы все наши огромные структуры данных, каки бы они не были, можем разложить в регистр.

Как я понял, здесь идет распределение виртуальных регистров в физические, перевод представления из

виртуальных в физических (оптимизация орегалок) (OREGALOC)
На данном уровне уже есть инструкции, мы можем вынести load вверх. (?)

3. Machine-dependent optimizations (*physical registers*)

Казалось бы, что тут можно еще сделать ? Регистры мы распределили, код поправили.

Однако существуют оптимизации, который можно сделать только на физических регистрах.

Самый просто пример **Scadelling**

Если у нас есть модель конвеера и мы по этой модели конвейера можем предсказывать задержки, то мы можем сделать critical path scadelling.

→ **То есть распланировать инструкции так, чтобы на критическом пути никто не стоял, не создавал задержки.**

Scadelling не имеет большого вклада на x86.

IR (Intermediate Representation) в gcc имеет несколько уровней

- GENERIC (просто деревья)
- Gimple (линеаризованные деревья, имеется в виду, что код похож на Си, но фактические это дерево лексем, написанное так, что он очень напоминает язык Си)
- Gimple SSA (Трехадресный код в SSA представлении)
- RTL (virtual regs) (Lisp подобный код, написанный для вирт регистров)
- RTL (physical regs)

Возьмём код факториала с рекурсией

```
unsigned fact(unsigned x) {  
    if (x < 2)  
        return 1;  
  
    return x * fact(x-1);  
}
```

и посмотрим какие оптимизации применяет к нему компилятор

g++ -O2 -fdump-tree-all fact.cpp -c

-fdump-tree-all — как раз ключик показывает все этапы все этапы преобразования, от дерева к файлу. Control the dumping at various stages of processing the intermediate language tree to a file.

В результате нам насыпает целый ворох файлов, где мы видим как меняется код от одного этапа оптимизации к другому.

```
fima@fima-laptop:~/fima_git/KV_Lectures/toolchain/lect2/code$ ls  
fact.cpp                                fact.cpp.037t.frel      fact.cpp.103t.forwprop2  fact.cpp.125t.copyprop2  fact.cpp.154t.ldist      fact.cpp.199t.fab1  
fact.cpp.005t.original                  fact.cpp.038t.evrp      fact.cpp.104t.objsz2     fact.cpp.126t.isolate-paths  fact.cpp.156t.copyprop3  fact.cpp.200t.widening_mul  
fact.cpp.006t.gimple                    fact.cpp.039t.mergephi1 fact.cpp.105t.alias      fact.cpp.127t.dse2          fact.cpp.169t.cunroll    fact.cpp.201t.store-merging  
fact.cpp.009t.omplower                  fact.cpp.040t.dse1      fact.cpp.106t.retslot    fact.cpp.128t.reassoc1      fact.cpp.174t.ivopts     fact.cpp.202t.tailc  
fact.cpp.010t.lower                     fact.cpp.041t.cddce1     fact.cpp.107t.fre3       fact.cpp.129t.dce3          fact.cpp.175t.lim4       fact.cpp.203t.dce7  
fact.cpp.012t.ehopt                     fact.cpp.042t.phiopt1    fact.cpp.108t.mergephi2  fact.cpp.130t.forwprop3     fact.cpp.176t.loopdone   fact.cpp.204t.crited1  
fact.cpp.013t.eh                        fact.cpp.043t.modref1    fact.cpp.109t.thread1    fact.cpp.131t.phiopt3      fact.cpp.180t.veclower21  fact.cpp.206t.uncprop1  
fact.cpp.015t.cfg                       fact.cpp.044t.tailr1     fact.cpp.110t.vrp1       fact.cpp.132t.ccp3         fact.cpp.181t.switchlower1 fact.cpp.207t.local-pure-const2  
fact.cpp.017t.ompexp                     fact.cpp.045t.iftoswitch fact.cpp.111t.dce2       fact.cpp.133t.sincos       fact.cpp.183t.reassoc2    fact.cpp.208t.modref2  
fact.cpp.022t.fixup_cfg1                 fact.cpp.046t.switchconv fact.cpp.112t.stdarg      fact.cpp.134t.bswap        fact.cpp.184t.slsr       fact.cpp.241t.resx  
fact.cpp.023t.ssa                       fact.cpp.048t.profile estimate fact.cpp.113t.cdce       fact.cpp.135t.laddress     fact.cpp.187t.fre5       fact.cpp.242t.nrv  
fact.cpp.025t.nothrow                   fact.cpp.049t.local-pure-const1 fact.cpp.114t.cselim     fact.cpp.136t.lim2         fact.cpp.188t.thread3    fact.cpp.243t.isel  
fact.cpp.027t.fixup_cfg2                 fact.cpp.050t.fnsplit    fact.cpp.115t.copyprop1  fact.cpp.137t.walloca2     fact.cpp.189t.dom3       fact.cpp.244t.optimized  
fact.cpp.028t.local-fnsummary1          fact.cpp.051t.release_ssa fact.cpp.116t.ifcombine  fact.cpp.138t.pre          fact.cpp.190t.strlen1    fact.cpp.332t.statistics  
fact.cpp.029t.einline                   fact.cpp.052t.local-fnsummary2 fact.cpp.117t.mergephi3  fact.cpp.139t.sink         fact.cpp.191t.thread4    fact.cpp.333t.earlydebug  
fact.cpp.030t.early_optimizations        fact.cpp.092t.fixup_cfg3 fact.cpp.118t.phiopt2    fact.cpp.143t.dce4         fact.cpp.192t.vrp2       fact.cpp.334t.debug  
fact.cpp.031t.objsz1                    fact.cpp.097t.adjust_alignment fact.cpp.119t.tailr2     fact.cpp.144t.fix_loops    fact.cpp.193t.copyprop5  fact.o  
fact.cpp.032t.ccp1                       fact.cpp.098t.ccp2       fact.cpp.120t.ch2        fact.cpp.145t.loop         fact.cpp.194t.wrestrict  main.cpp  
fact.cpp.033t.forwprop1                  fact.cpp.099t.post_ipa_warn1 fact.cpp.121t.cplxlower1 fact.cpp.146t.loopinit     fact.cpp.195t.dse4         
fact.cpp.034t.ethread                    fact.cpp.100t.cunrolli   fact.cpp.122t.sra        fact.cpp.148t.sccp         fact.cpp.196t.cddce3       
fact.cpp.035t.esra                       fact.cpp.101t.backprop  fact.cpp.123t.thread2    fact.cpp.152t.cddce2       fact.cpp.197t.forwprop4    
fact.cpp.036t.ealias                     fact.cpp.102t.phiprop    fact.cpp.124t.dom2       fact.cpp.153t.ivcanon      fact.cpp.198t.phiopt4
```

Важно сказать, что до 244 optimized происходят машинно независимые **Machine independent** оптимизации.

Сравним два кода из самого **начала** из самого **конца** применения оптимизаций

cat cat fact.cpp.006t.gimple

cat fact.cpp.244t.optimized

```

unsigned int fact (unsigned int x)
{
    unsigned int D.2351;

    if (x <= 1) goto <D.2349>; else goto <D.2350>;
    <D.2349>:
    D.2351 = 1;
    // predicted unlikely by early return (on trees)
    predictor.
    return D.2351;
    <D.2350>:
    _1 = x + 4294967295;
    _2 = fact (_1);
    D.2351 = x * _2;
    return D.2351;
}

```

Видно, что начальное gimple представление не сильно отличается от оригинального кода и очень похоже на сишный код, это не сишный код, это деревья, но написаны они так, чтобы напоминать сишный код.

```

;; Function fact (_Z4factj, funcdef_no=0, decl_uid=2347,
cgraph_uid=1, symbol_order=0)

```

```

Removing basic block 5
Removing basic block 6
Removing basic block 7
Removing basic block 8
unsigned int fact (unsigned int x)
{
    unsigned int _1;
    unsigned int mult_acc_6;
    unsigned int mult_acc_10;
    unsigned int mult_acc_11;

```

```

<bb 2> [local count: 118111600]:

```

```

if (x_2(D) <= 1)
    goto <bb 3>; [11.00%]
else
    goto <bb 4>; [89.00%]

```

```

<bb 3> [local count: 118111600]:

```

```

# mult_acc_10 = PHI <mult_acc_6(4), 1(2)>
return mult_acc_10;

```

```

<bb 4> [local count: 955630225]:

```

```

# x_9 = PHI <_1(4), x_2(D)(2)>
# mult_acc_11 = PHI <mult_acc_6(4), 1(2)>
_1 = x_9 + 4294967295;
mult_acc_6 = x_9 * mult_acc_11;
if (_1 == 1)
    goto <bb 3>; [11.00%]
else
    goto <bb 4>; [89.00%]

```

```

}

```

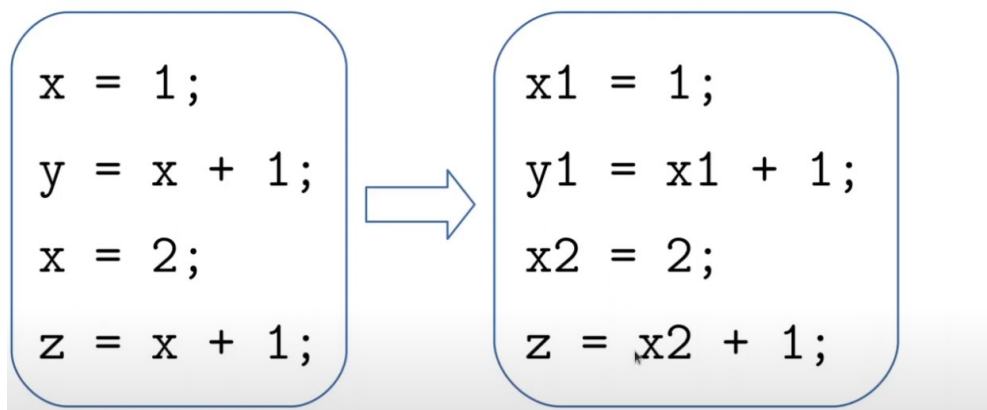
Данный код тоже не сильно отличается от базового fact.cpp.006t.gimple, код разве что подготовлен к ассемблеру путем добавления goto. Еще предлагаю обратить внимание на то, что компилятор, для своего удобства добавил несколько temporary переменных. Если компилятору нужно добавить переменных он их добавит, если будет нужно он их уберет. Поэтому всякие «гениальные» однострочные кода не имеют смысла. Если добавить лишние temp переменные компилятор их уберёт. Так же важно заметить, что fact.cpp.244t.optimized это результат всего цикла машинно независимых преобразований. И мы видим некоторую магию здесь мы видим, что нашу изначально функцию с хвостовой рекурсией а, компилятор превратил рекурсию в цикл, ого !

Так же компилятор добавил вероятности того, по какой дуге пойдёт код ! Для такой магии есть специальное преобразование fact.cpp.044t.tailr1

SSA — Static Single Assignment form.

Это такая форма преобразования кода, которая позволяет добавить номер версии к переменной, грубо говоря. Например при создании переменная будет называться x_1, когда мы будем использовать переменную это все еще будет x_1, но когда мы переменной x (которая была создана и в ssa уже была x_1) присвоим новое значение, она поменяется, имя будет тем же, но значение другим, то в ssa это будет уже x_2. Данное представление позволяет компилятору проводить оптимизации.

SSA : static single assignment



Пример трансформации в ssa представление нашего кода с факториалом

cat fact.cpp.023t.ssa

```
;; Function fact (_Z4factj, funcdef_no=0, decl_uid=2347, cgraph_uid=1,
symbol_order=0)
```

```
unsigned int fact (unsigned int x)
```

```
{
  unsigned int _1;
  unsigned int _2;
  unsigned int _3;
  unsigned int _8;
  unsigned int _9;
  unsigned int _10;
```

```
<bb 2> :
  if (x_5(D) <= 1)
    goto <bb 3>; [INV]
  else
    goto <bb 4>; [INV]

<bb 3> :
  _10 = 1;
  // predicted unlikely by early return (on trees) predictor.
  goto <bb 5>; [INV]
```

```
<bb 4> :
  _1 = x_5(D) + 4294967295;
  _8 = fact (_1);
  _2 = _8;
  _9 = x_5(D) * _2;
```

```
<bb 5> :
  # _3 = PHI <_10(3), _9(4)>
  return _3;
```

```
}
```

Видно, что компилятор создал несколько промежуточных переменных, для перевода кода в ssa. Так же обращаем внимание на фи функции

```
_3 = PHI <_10(3), _9(4)>
```

Так, решается проблема с if ветвлением, потому в момент if переменная может измениться, а может и не измениться. В скобка обозначается номер basic block или bb в ssa представлении.

Чтобы получить дампы IR используются опции g++/gcc

-fdump-tree-all-<options>

-fdump-rtl-all-<options>

Последний GIMPLE dump это fact.cpp.244t.optimized и у нас происходит первый lowering (переход на более низкий уровень представления компилятора, так он называется в llvm, в gcc это зовётся expand)

Теперь генерируем RTL представление

→ **g++ -O2 -fdump-rtl-all fact.cpp -c**

Нам насыпает, где то 100 дополнительных преобразований.

RTL преобразования напоминают чистый Lisp я тут уже не сильно понимаю, весь RTL выглядит вот так и все это будет выглядеть так до самой кодогенерации. Тут мои полномочия все, привожу кусок кода.

```
;;
;; Full RTL generated for this function:
;;
(note 1 0 6 NOTE_INSN_DELETED)
(note 6 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 2 6 3 2 (set (reg/v:SI 84 [ x ])
  (reg:SI 5 di [ x ])) "fact.cpp":1:28 -1
  (nil))
(note 3 2 10 2 NOTE_INSN_FUNCTION_BEG)
(insn 10 3 11 2 (set (reg:CC 17 flags)
  (compare:CC (reg/v:SI 84 [ x ])
    (const_int 1 [0x1]))) "fact.cpp":2:2 -1
  (nil))
(jump_insn 11 10 30 2 (set (pc)
  (if_then_else (gtu (reg:CC 17 flags)
    (const_int 0 [0]))
    (label_ref:DI 32)
    (pc))) "fact.cpp":2:2 806 {*jcc}
  (int_list:REG_BR_PROB 955630228 (nil))
-> 32)
.....
```

GCC: как читать RTL дампы

`_6 = x_7 + 4294967295`



```
(insn 14 13 0 (parallel [
  (set (reg:SI 89 [ _6 ])
    (plus:SI (reg/v:SI 91 [ x ])
      (const_int -1 [0xffffffffffffffff])))
  (clobber (reg:CC 17 flags))
]) fact.c:7 -1
(nil))
```

Что тут написано ? :)

Давайте разбираться. Поскольку мы в RTL с виртуальными регистрами, у нас не было еще распределения по физическим, а значит в данный момент у нас на всех регистров хватает на всех.

Переменной **_6** соответствует регистр **89**, а **x_7** соответствует регистр **91**, оно присваивает их как хочет. Важно то, что не трогаются маленькие числа, потому что они зарезервированы под физические регистры.

Дальше происходит **set** в **89** регистр (две скобки открываются, закрывается только одна) выражения результата выполнения выражения (**91** регистр + `unsigned_int(-1)`).

Параллельно (видимо слово **parallel** в самом начале) идет **clobber** (clobber = непредсказуемо изменяет) регистра флагов, регистра **CC**.

Здесь присутствует работа с физическим регистром флагов. Машинно зависимое представление, как раз работает с 17 регистром флагов для x86.

fact.cpp.294r.ira — Interprocedure Register Allocator, помент покраски графа, **REG_ALLOC**

fact.cpp.295r.reload Замена виртуальных регистров на физические и выгрузка виртуальных регистров в память, если мы не можем сопоставить все виртуальные регистры физическим регистрам, последующая загрузка обратно.

fact.cpp.297r.postreload Последующая чистка за процедурой **reload**

Переходя к ассемблеру, есть команда для g++/gcc, которая показывает, какая rtl инструкция привела к ассемблерному коду

g++ -O2 fact.cpp -dP -S -masm=intel

А что такое Ассемблер ?

Слово Ассемблер семантически перегружено.

1. Есть язык ассемблера, который традиционно очень близок к машинному уровню. До некоторой степени ассемблер это весёлое представление двоичного кода.

2. Ассемблер как программа, например GAS (GNU Assembler), это программа проводит ассемблирование,

fact.cpp.294r.ira

```
.file 0
"/home/fima/fima_git/KV_Lectures/Toolchain/Lect2/code" "fact.cpp"
```

```
.p2align 4
.globl _Z4factj
.type _Z4factj, @function
```

```
_Z4factj:
```

```
.LVL0:
```

```
.LFB0:
```

```
.file 1 "fact.cpp"
.loc 1 1 28 view -0
.cfi_startproc
.loc 1 1 28 is_stmt 0 view .LVU1
endbr64
```

```
.loc 1 2 2 is_stmt 1 view .LVU2
```

```
mov eax, 1
```

```
cmp edi, 1
```

```
jbe .L1
```

```
.LVL1:
```

```
.p2align 4,,10
```

```
.p2align 3
```

```
.L2:
```

```
.loc 1 5 2 view .LVU3
```

```
mov edx, edi
```

```
.loc 1 5 17 is_stmt 0 view .LVU4
```

```
sub edi, 1
```

```
imul eax, edx
```

```
.loc 1 2 2 is_stmt 1 view .LVU5
```

```
cmp edi, 1
```

```
jne .L2
```

```
.L1:
```

```
.loc 1 6 1 is_stmt 0 view .LVU6
```

```
ret
```

```
.cfi_endproc
```

та на языке ассемблера, в программу на машинном языке.

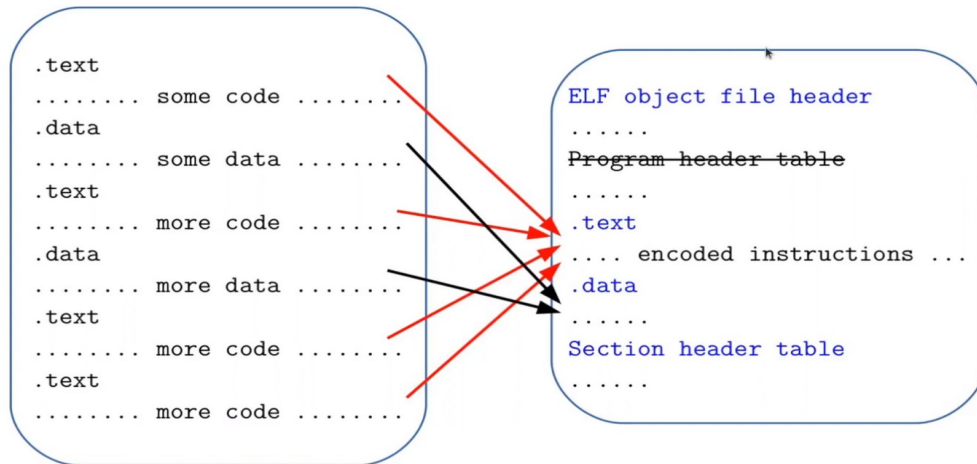
(executable and linkable), дворф и эльф :)

DWARF содержит кучу информации, например, в каком мы файле, на какой строчке (Вторая цифра в **.loc**), где начинаются и заканчиваются наши функции.

Поэтому все что собрано с дебаг опцией **-g** занимает очень много памяти. При сборке с **-g** в ассемблере появляется метка **.Ldebug_info**.

Так же метки **.cfi_startproc** и **.cfi_endproc**, это начало и функции, что позволяет нам натравливаться на нее через gdb

Ассемблирование: сборка секций



Суть ассемблирования — сборка секций. Мы берем секцию text слева, в секции text есть что то, что то бинарное. Мы берем это что то и кодируем и собираем инструкции в единую секцию text нашего объектного файла. Поскольку ассемблер порождает **объектный код**, то при вызове ассемблера у нас рождается EFL.

```
g++ -O2 fact.cpp -c
```

-c → опция означает Compile or assemble the source files, but do not link.

В заголовке нашего файла уже будет EFL.

То есть в нашем случае мы получаем объектный файл **fact.o**, но это ещё не исполняемый файл, потому что мы собрали только **объектный код**.

Как ни странно **объектный код**, это чуть больше, чем **исполняемый файл**, потому что **объектный код**, содержит чуть больше информации, в **исполняемом файле**, эта информация уже подставлена, в **объектном коде** описанная информация лежит в виде метаданных или в других местах. Поэтому часто это может затруднять ДИСассемблирование.