# Introduction to GCC Backend

Take a example to understand how it works

J.Liu

hellogcc

August 16, 2011

Outline

Structure

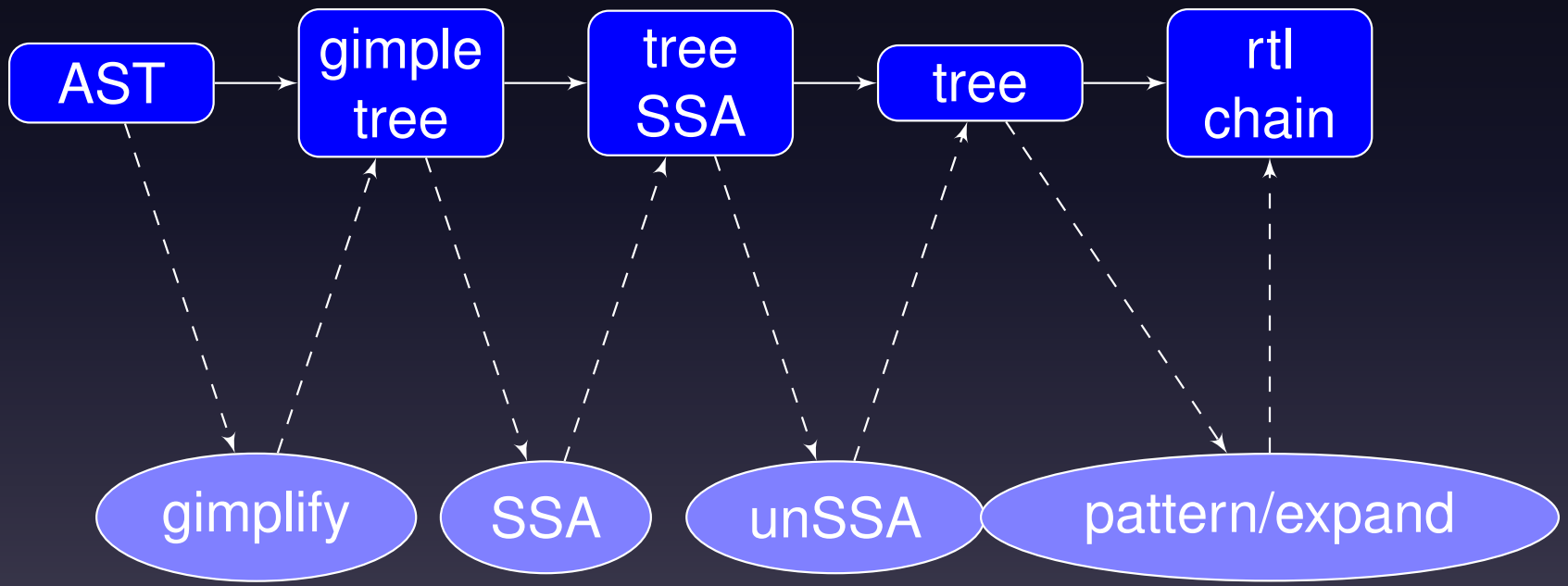
Expand

Backend

Disscution

# compile pipeline

Figure: compile pipeline



# a simple example

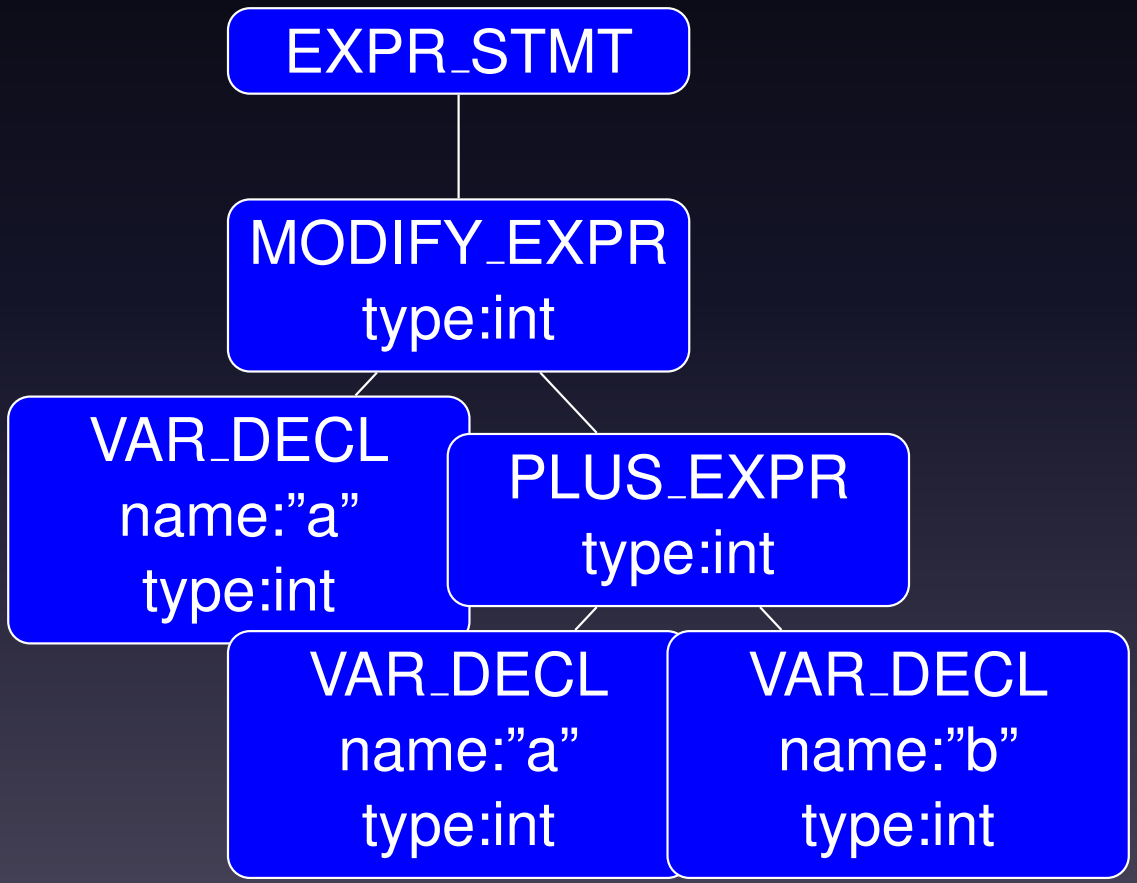
Let us take a simple example to understand how gcc works.

Listing 1: add case

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5     int a = 1;
6     int b = 2;
7
8     a = a + b;
9
10    return 0;
11 }
```

# AST expression

Figure: AST expression of ”a = a + b;”



# GIMPLE expression

Listing 2: gimple code

```
1  main ()
2  {
3    int D.2553;
4    int a;
5    int b;
6
7    a = 1;
8    b = 2;
9    a = a + b;
10   D.2553 = 0;
11   return D.2553;
12 }
```

# RTL pattern

Listing 3: rtl pattern

```
1  (define_insn "*add<mode>3"
2    [(set (match_operand:GPR 0 "register_operand" "=d,d")
3          (plus:GPR (match_operand:GPR 1 "register_operand" "d,d")
4                    (match_operand:GPR 2 "arith_operand" "d,Q")))]
5    "!TARGET_MIPS16"
6    "@
7  ␣␣␣␣<d>addu\t%0,%1,%2
8  ␣␣␣␣<d>addiu\t%0,%1,%2"
9    [(set_attr "type" "arith")
10    (set_attr "mode" "<MODE>")])
```

# RTL expression

Listing 4: rtl code

```
;; load imm "1" into reg 259
(insn 5 4 6 3 add.c:5 (set (reg:SI 259) (const_int 1 [0x1])) -1 (nil))
;; store it to a address
(insn 6 5 7 3 add.c:5 (set (mem/c/i:SI (reg/f:SI 253 virtual-stack-vars) [0 a+0 S4 A32])
        (reg:SI 259)) -1 (nil))
;; load imm "2" into reg 260
(insn 7 6 8 3 add.c:6 (set (reg:SI 260) (const_int 2 [0x2])) -1 (nil))
;; store it to a's address + 4
(insn 8 7 9 3 add.c:6 (set (mem/c/i:SI (plus:SI (reg/f:SI 253 virtual-stack-vars)
                (const_int 4 [0x4])) [0 b+0 S4 A32])
        (reg:SI 260)) -1 (nil))
;; load a
(insn 9 8 10 3 add.c:8 (set (reg:SI 261)
        (mem/c/i:SI (reg/f:SI 253 virtual-stack-vars) [0 a+0 S4 A32])) -1 (nil))
;; load b from a' address + 4
(insn 10 9 11 3 add.c:8 (set (reg:SI 262)
        (mem/c/i:SI (plus:SI (reg/f:SI 253 virtual-stack-vars)
                (const_int 4 [0x4])) [0 b+0 S4 A32])) -1 (nil))
;; a = a + b
(insn 11 10 12 3 add.c:8 (set (reg:SI 263)
        (plus:SI (reg:SI 261)
            (reg:SI 262))) -1 (nil))
;; store a
(insn 12 11 13 3 add.c:8 (set (mem/c/i:SI (reg/f:SI 253 virtual-stack-vars) [0 a+0 S4 A3
        (reg:SI 263)) -1 (nil))
```

# ASM code

Listing 5: asm code

```
1  li $2,1                    # 0x1
2  sw $2,0($fp)
3  li $2,2                    # 0x2
4  sw $2,4($fp)
5  lw $3,0($fp)
6  lw $2,4($fp)
7  addu $2,$3,$2
8  sw $2,0($fp)
```

# notice! pattern mov***mode***

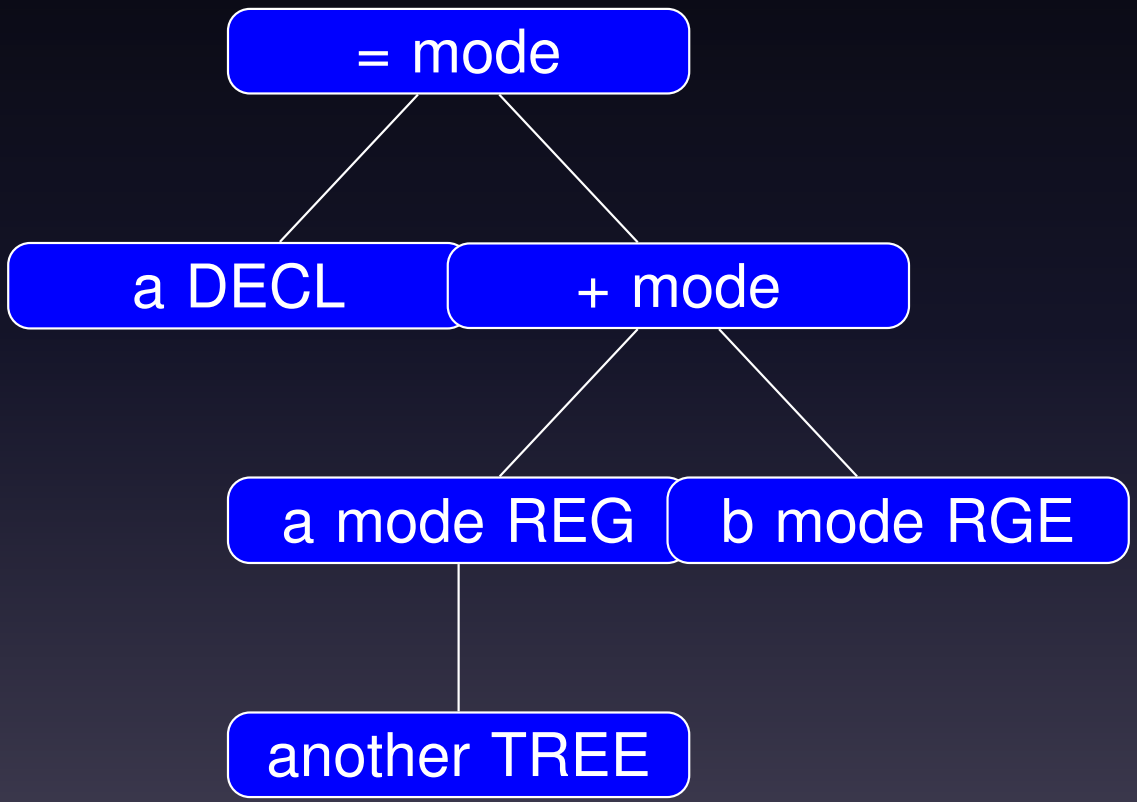
The First Pattern We Need to Write is mov***mode*** !!!
Take a RISC CPU Example:

- ***load***
  mov***mode*** to REG:***mode***from ADDR:size_of_***mode***
- ***store***
  mov***mode*** from REG:***mode*** to ADDR:size_of_***mode***
- ***move***
  mov***mode*** from REG:***mode*** to REG:***mode***

Only If mov***mode*** is ready, we are able to support other patterns. That is, firstly, we need load&store works well.

# a mov***mode*** example

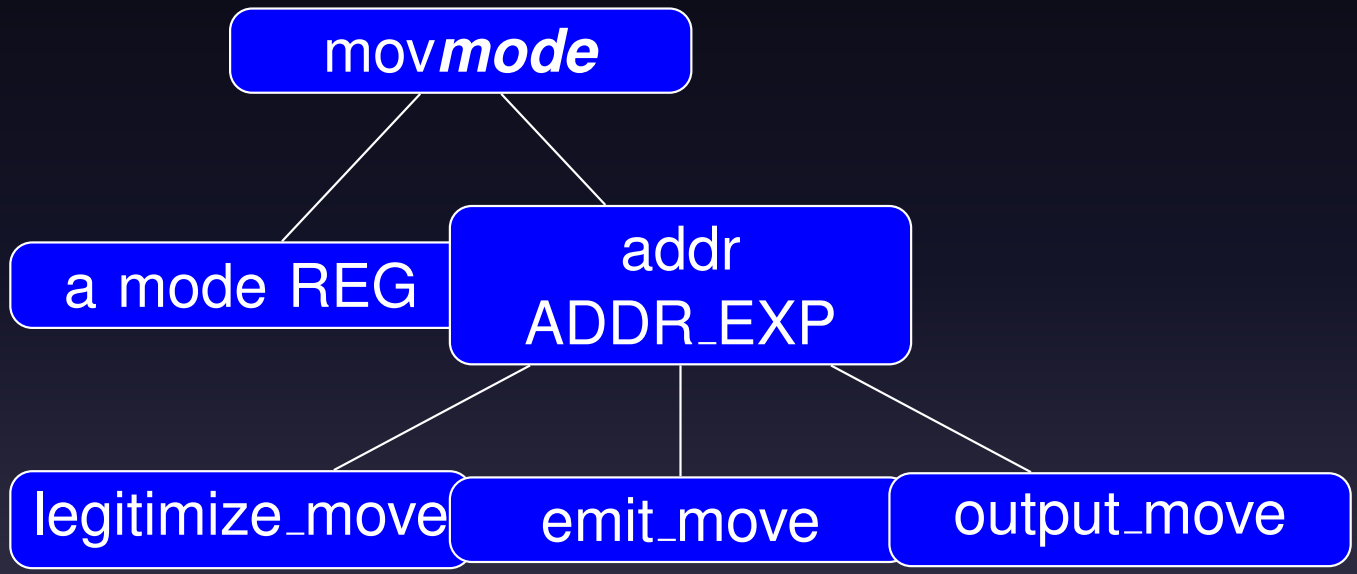
Figure: tree of ”a = a + b”




Traverse this tree, when we get ”a”, we ”jump” into another tree.
At this time, the traversal is not yet complete.

# a mov***mode*** example

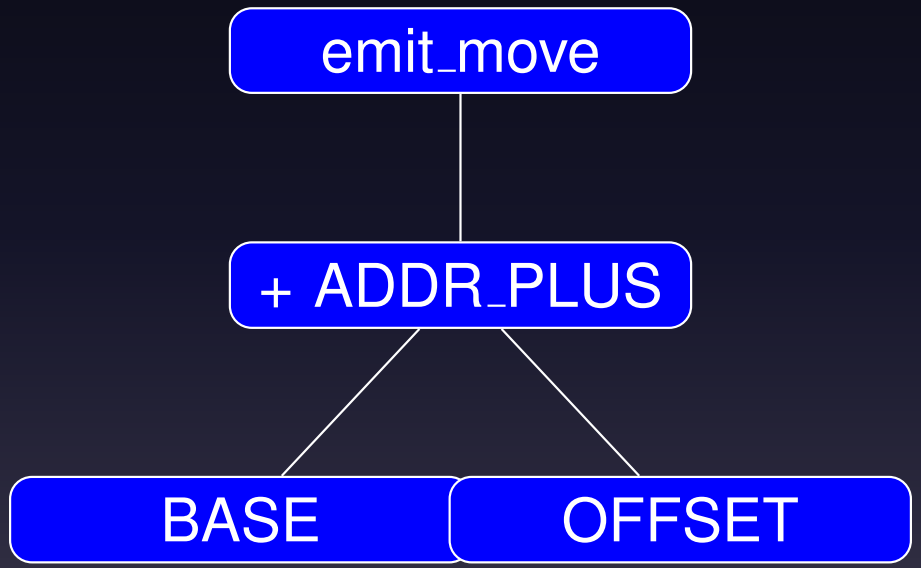
Figure: tree of ”mov***mode***”




See anything? ”define_expand ***mode***” and ”define_insn ***mode***_internal” in .md file, legitimize_move emit_move output_move in .c file.

# a mov***mode*** example

Figure: tree of ”ADDR_CALC”




It still need more compute due to your context, just a exmple here.

# tree node

tree.def

Listing 6: tree node

```
1 DEFTREECODE (PLUS_EXPR, "plus_expr", tcc_binary, 2)
```

rtl.def

Listing 7: rtl node

```
1 DEF_RTL_EXPR(PLUS, "plus", "ee", RTX_COMM_ARITH)
```

# op

optab.c : init_optabs()

Listing 8: init optab

```
1    init_optab (add_optab, PLUS);
2    init_optabv (addv_optab, PLUS);
3    init_optab (ssadd_optab, SS_PLUS);
4    init_optab (usadd_optab, US_PLUS);
```

optab.c : optab_for_tree_code()

Listing 9: optab select

```
1  case PLUS_EXPR:
2    if (TYPE_SATURATING(type))
3      return TYPE_UNSIGNED(type) ?
4                          usadd_optab : ssadd_optab;
5    return trapv ? addv_optab : add_optab;
```

# op

optab.c : init_optabs()

Listing 10: optab library

```
1  add_optab->libcall_basename = "add";
2  add_optab->libcall_suffix = '3';
3  add_optab->libcall_gen = gen_int_fp_fixed_libfunc;
4  addv_optab->libcall_basename = "add";
5  addv_optab->libcall_suffix = '3';
6  addv_optab->libcall_gen = gen_intv_fp_libfunc;
7  ssadd_optab->libcall_basename = "ssadd";
8  ssadd_optab->libcall_suffix = '3';
9  ssadd_optab->libcall_gen = gen_signed_fixed_libfunc;
10 usadd_optab->libcall_basename = "usadd";
11 usadd_optab->libcall_suffix = '3';
12 usadd_optab->libcall_gen = gen_unsigned_fixed_libfunc
```

# op

genopinit.c

Listing 11: gen op init

```
1   static const char * const optabs[] =
2   {
3   "set_optab_handler
4   ␣␣␣␣␣␣␣␣(add_optab,␣$A,␣CODE_FOR_$(add$P$a3$))",
5   "set_optab_handler
6   ␣␣␣␣␣␣␣␣(addv_optab,␣$A,␣CODE_FOR_$(add$F$a3$)),\n\
7   ␣set_optab_handler
8   ␣␣␣␣␣␣␣␣(add_optab,␣$A,␣CODE_FOR_$(add$F$a3$))",
9   "set_optab_handler
10  ␣␣␣␣␣␣␣␣(addv_optab,␣$A,␣CODE_FOR_$(addv$I$a3$))",
11  "set_optab_handler
12  ␣␣␣␣␣␣␣␣(add_optab,␣$A,␣CODE_FOR_$(add$Q$a3$))",
13  "set_optab_handler
14  ␣␣␣␣␣␣␣␣(ssadd_optab,␣$A,␣CODE_FOR_$(ssadd$Q$a3$))",
15  "set_optab_handler
16  ␣␣␣␣␣␣␣␣(usadd_optab,␣$A,␣CODE_FOR_$(usadd$Q$a3$))",
```

# op

optab.h

Listing 12: optab enum

```
1 enum optab_index
2 {
3   OTI_ssadd,
4   OTI_usadd,
5   OTI_add,
6   OTI_addv,
```

Listing 13: optab define

```
1 #define ssadd_optab (&optab_table[OTI_ssadd])
2 #define usadd_optab (&optab_table[OTI_usadd])
3 #define add_optab (&optab_table[OTI_add])
4 #define addv_optab (&optab_table[OTI_addv])
```

# tree to RTL

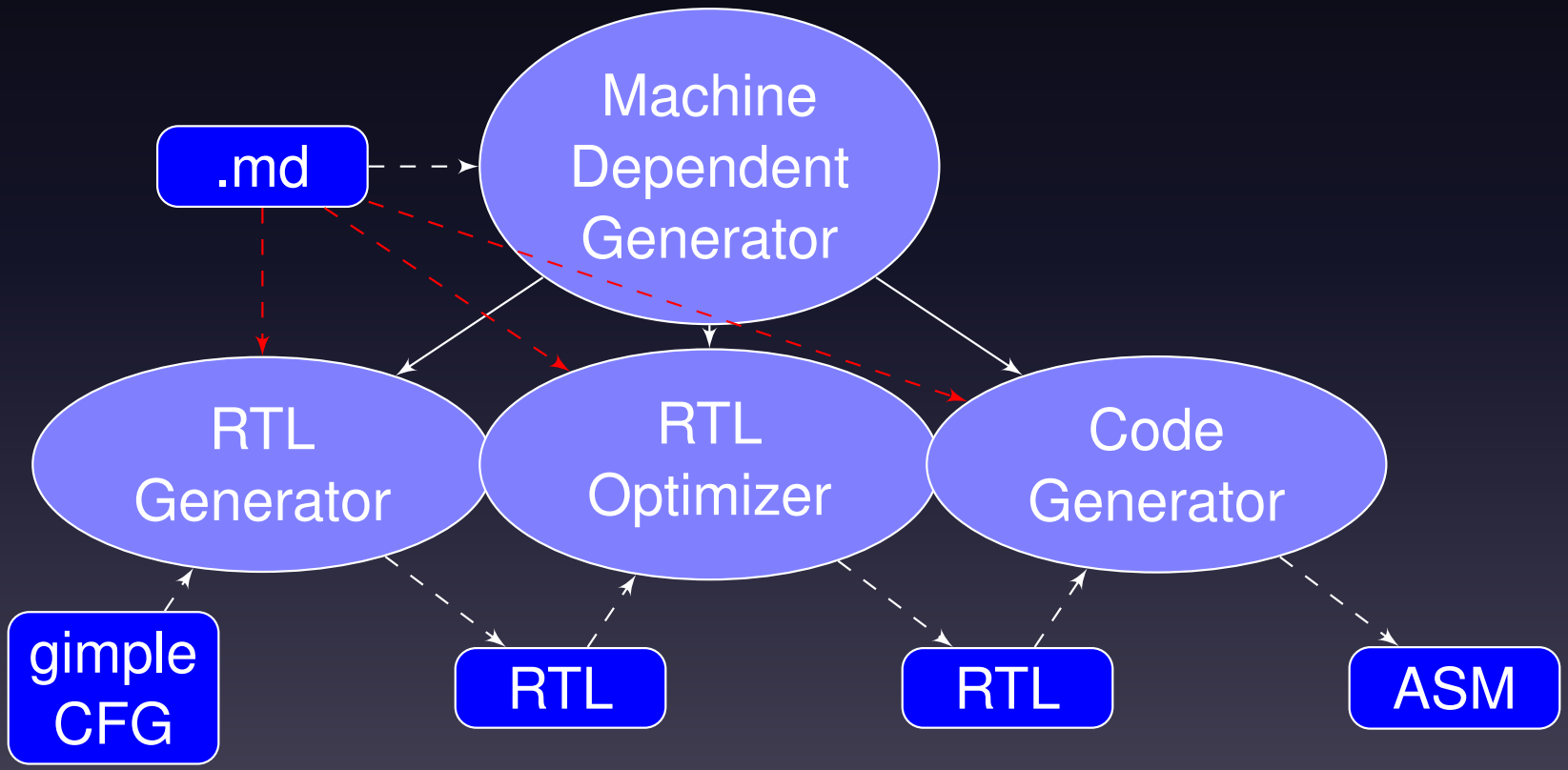
expr.c : expand_expr_real_2()

Listing 14: tree to RTL

```
1  case PLUS_EXPR:
2    if (TREE_CODE (treeop0) == PLUS_EXPR
3        && TREE_CODE (TREE_OPERAND (treeop0, 1))
4                                          == INTEGER_CST
5        && TREE_CODE (treeop1) == VAR_DECL
6        && (DECL_RTL (treeop1) == frame_pointer_rtx
7              || DECL_RTL (treeop1) == stack_pointer_rtx
8              || DECL_RTL (treeop1) == arg_pointer_rtx))
9    {
10     tree t = treeop1;
11
12     treeop1 = TREE_OPERAND (treeop0, 0);
13     TREE_OPERAND (treeop0, 0) = t;
14   }
15   ...
```

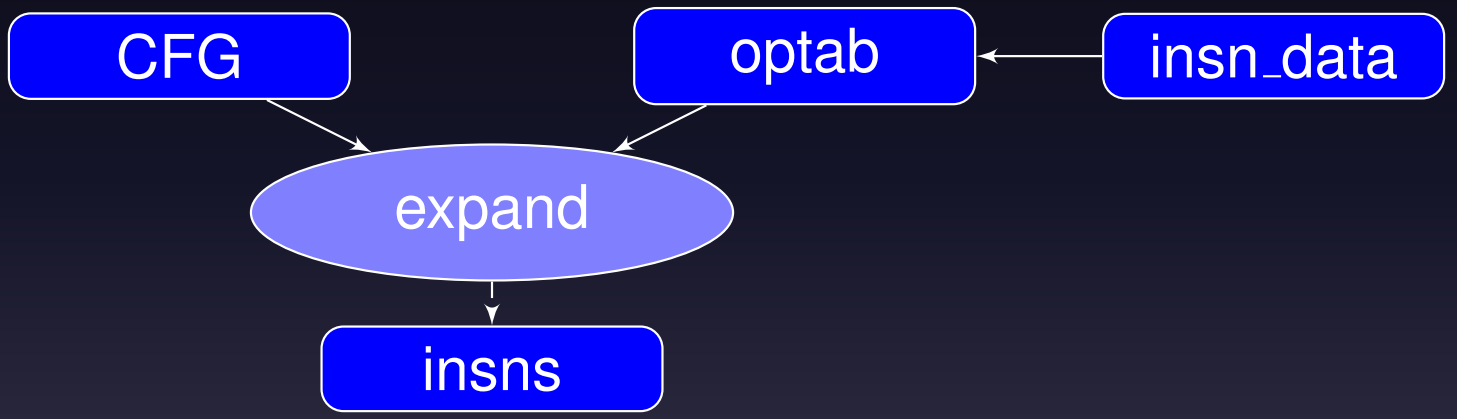
# how does Backend work

Figure: Backend pipeline



# CFG expander

Figure: CFG expander




expander read CFG, find suitable opcode from insn_data in optab.

# pattern type

- define_insn generate one RTL
- define_expand multiple RTL generation
- define_split plit a complex insn into several simpler insns
- define_insn_and_split when define_split exactly matches a define_insn
- define_peephole2 RTL to RTL peephole optimizers
- define_attr defines attributes and a set of values for each
- define_cond_exec define conditional execution, or predication
- define_constants using literal constants make .md file more understabdable

# Machine Dependent Generator

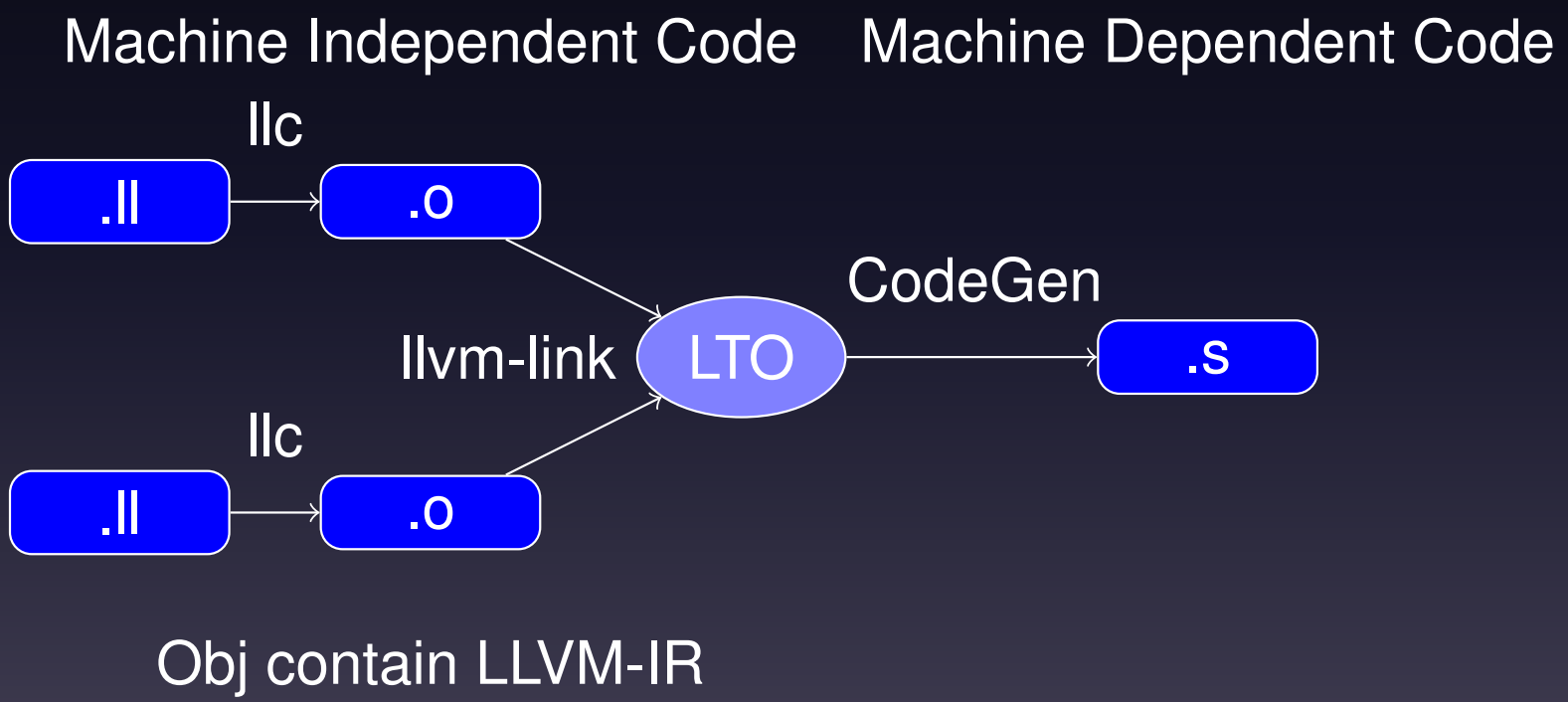
- genattr
  insn-attr.h, generate attributes.
- genattrtab
  insn-attrtab.c, compute attributes.
- genautomaata
  insn-automata.c, generate code for pipeline description.
- gencodes
  insn-codes.h, generate insn_code_number values for CODE_FOR_XXX.
- genconstants
  insn-constants.h, generate constant for #define in MD file.
- genemit
  insn-enit.c, emit insns as rtl.
- genextract
  insn-extract, extract operands from insn as rtl.

# Machine Dependent Generator

- genmodes
  insn-modes.h, generate modes fr machine.
- genopinit
  insn-opinit.c, generate initialize optabs.
- genoutput
  insn-output.c, output assembler from rtl.
- genpeep
  insn-peep.c, peephole.
- genrecog
  insn-recog.c, recognize if rtl is valid.

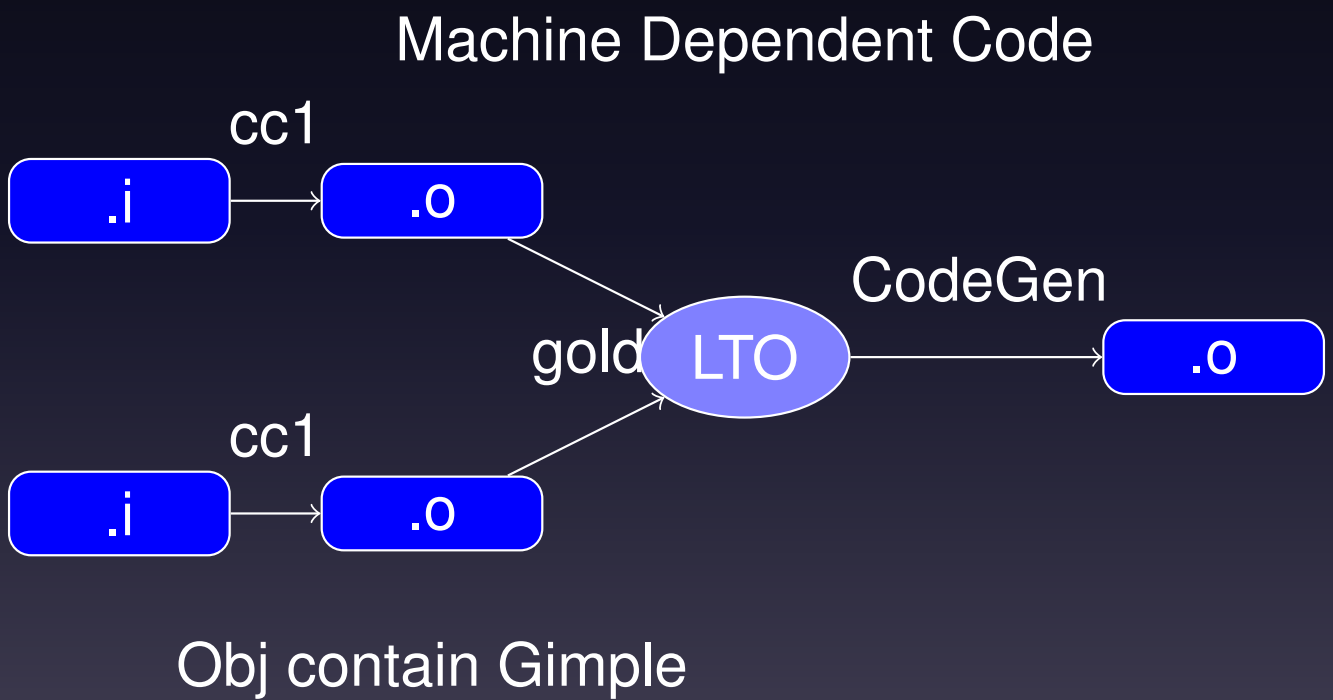
# LLVM LTO pipeline

Figure: LLVM LTO pipeline



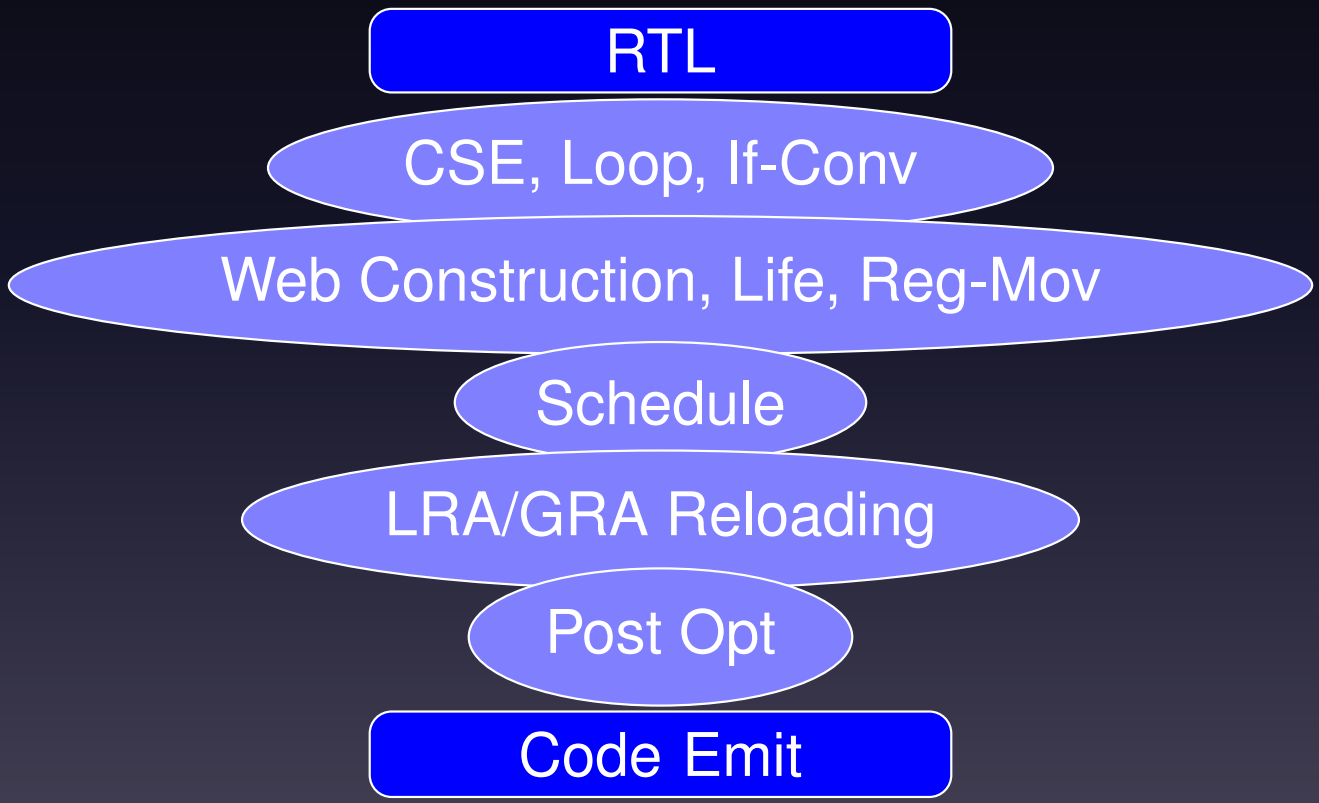
# GCC LTO pipeline

Figure: GCC LTO pipeline



# RTL Backend

Figure: RTL Backend



# compare backend

- GCC:machine.h
  LLVM:MReg.td MCallingConv.td
- GCC:rtl pattern
  LLVM:MInstrInfo.td
- GCC:genautomata
  LLVM:MipsSchedule.td
- GCC:define_expand
  LLVM: DAGToDAG/Lowering
- GCC:construct a DFA to match pattern
  LLVM:SelectionDAG

# Personal Views

- GCC&LLVM both are complete compilers, both have their own FE ME and BE.
- LLVM Backend is similar with GCC Backend.
- GCC LTO implemented as a FE, FE Tech is still important. Ian write a go FE for GCC, can you do that?
- GCC LTO learn from LLVM.
- LLVM Schedule learn from GCC SMS&automata.
- A lot of nice people working on GCC and LLVM at the same time.

# Can We Generate Code From Gimple?

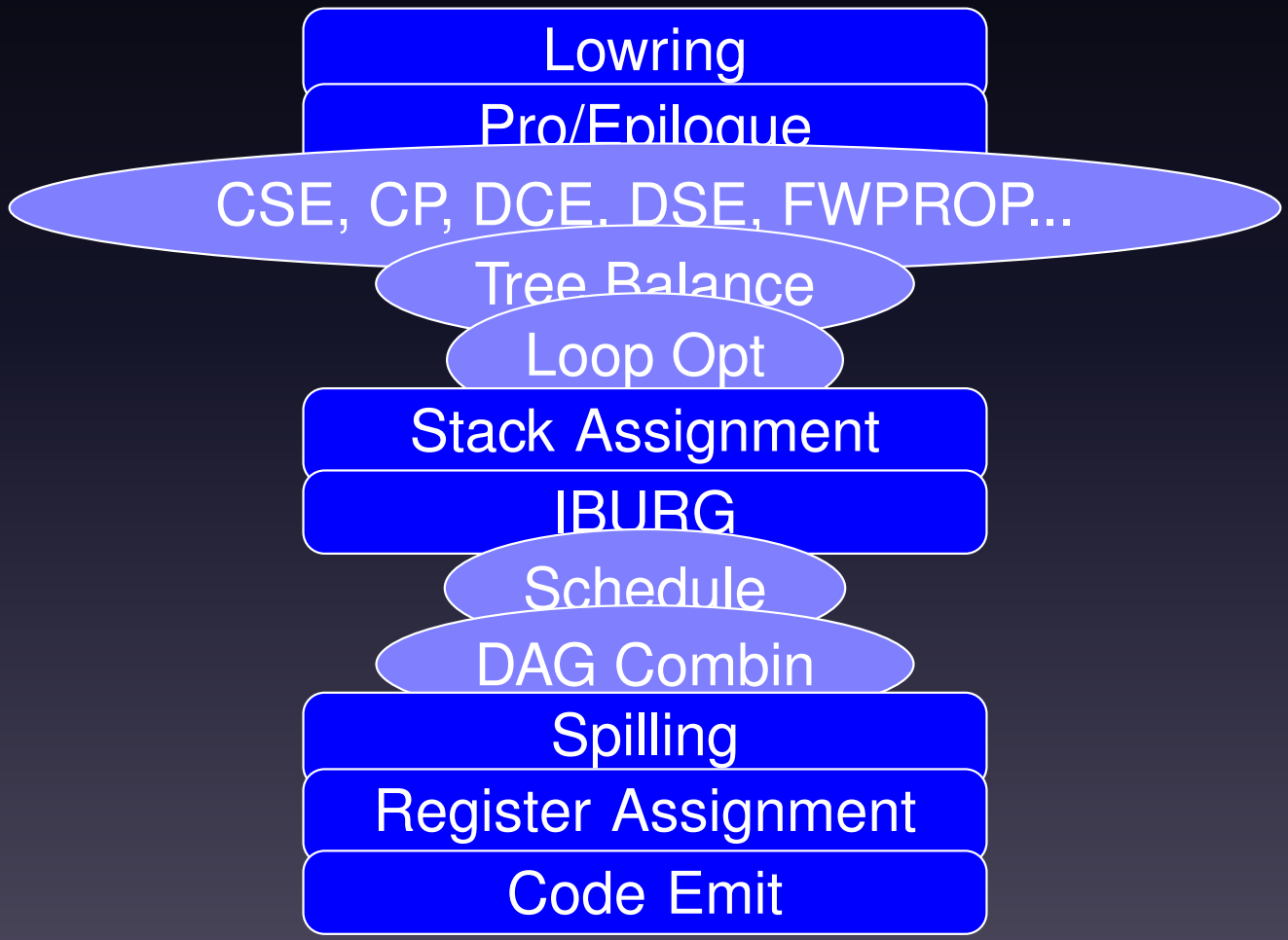
fuse: RA

- Can NOT intervene RA, reload specifically, for its complexity.
- DSP have kinds of strange load/store.
- RTL backend is very consummate, but not perfect, if you write a wrong pattern, it will still be matched and generate wrong RTL.
- gimple-pass is easier to write than RTL-pass.

People always do not like the things that they do not understand.
I do NOT understand ***RTL Backend***.

# Gimple Backend

Figure: Gimple backend



# Discussion

Discussion!