DWARF

Pierre-Marie de Rodat

pmderodat@lse.epita.fr
PM @ {rezosup, freenode, geeknode, . . . }
`http://lse.epita.fr`

February 12, 2013

Plan

1 Introduction

Debugging. . .

- Debuggers generally have access to:
  - Registers
  - Virtual memory
  - Most of the time, the binary file
- They can compute:
  - The backtrace: with the frame pointer register, or with some static analysis. . .
  - Not much more :-(

Debugging. . .

- Debuggers generally have access to:
    - Registers
    - Virtual memory
    - Most of the time, the binary file
- They can compute:
    - The backtrace: with the frame pointer register, or with some static analysis. . .
    - Not much more :-(

Debugging. . .

- Debuggers generally have access to:
  - Registers
  - Virtual memory
  - Most of the time, the binary file
- They can compute:
  - The backtrace: with the frame pointer register, or with some static analysis. . .
  - Not much more :-(

Debugging. . .

- Debuggers generally have access to:
  - Registers
  - Virtual memory
  - Most of the time, the binary file
- They can compute:
  - The backtrace: with the frame pointer register, or with some static analysis. . .
  - Not much more :-(

Debugging. . .

- Debuggers generally have access to:
  - Registers
  - Virtual memory
  - Most of the time, the binary file
- They can compute:
  - The backtrace: with the frame pointer register, or with some static analysis. . .
  - Not much more :-(

Debugging. . .

- Debuggers generally have access to:
    - Registers
    - Virtual memory
    - Most of the time, the binary file
- They can compute:
    - The backtrace: with the frame pointer register, or with some static analysis. . .
    - Not much more :-(

Debugging. . .

- Debuggers generally have access to:
    - Registers
    - Virtual memory
    - Most of the time, the binary file
- They can compute:
    - The backtrace: with the frame pointer register, or with some static analysis. . .
    - Not much more :-(

. . . with DWARF (1/3)

- With no other information, debugging a high-level language source code is hard.
- Manually look at ASM and original source code.
- Understand how the program works, where expressions are evaluated, etc.
- Compilers can help producing DWARF info (among others) along with the ASM.
- At each compilation pass, maintain metadata associated with the code.
- With GCC/Clang, enabled with the -g switch.

. . . with DWARF (1/3)

- With no other information, debugging a high-level language source code is hard.
- Manually look at ASM and original source code.
- Understand how the program works, where expressions are evaluated, etc.
- Compilers can help producing DWARF info (among others) along with the ASM.
- At each compilation pass, maintain metadata associated with the code.
- With GCC/Clang, enabled with the -g switch.

. . . with DWARF (1/3)

- With no other information, debugging a high-level language source code is hard.
- Manually look at ASM and original source code.
- Understand how the program works, where expressions are evaluated, etc.
- Compilers can help producing DWARF info (among others) along with the ASM.
- At each compilation pass, maintain metadata associated with the code.
- With GCC/Clang, enabled with the -g switch.

. . . with DWARF (1/3)

- With no other information, debugging a high-level language source code is hard.
- Manually look at ASM and original source code.
- Understand how the program works, where expressions are evaluated, etc.
- Compilers can help producing DWARF info (among others) along with the ASM.
- At each compilation pass, maintain metadata associated with the code.
- With GCC/Clang, enabled with the -g switch.

. . . with DWARF (1/3)

- With no other information, debugging a high-level language source code is hard.
- Manually look at ASM and original source code.
- Understand how the program works, where expressions are evaluated, etc.
- Compilers can help producing DWARF info (among others) along with the ASM.
- At each compilation pass, maintain metadata associated with the code.
- With GCC/Clang, enabled with the `-g` switch.

. . . with DWARF (1/3)

- With no other information, debugging a high-level language source code is hard.
- Manually look at ASM and original source code.
- Understand how the program works, where expressions are evaluated, etc.
- Compilers can help producing DWARF info (among others) along with the ASM.
- At each compilation pass, maintain metadata associated with the code.
- With GCC/Clang, enabled with the `-g` switch.

. . . with DWARF (2/3)

- Source code structure: declarations
- Source code locations: from PC to line:column
- Variable locations: when at PC, where to look at for

```
int a;
```

- Call Frame Information: stack (un|re)winding
- Special case: exception handlers

. . . with DWARF (2/3)

- Source code structure: declarations
- Source code locations: from PC to line:column
- Variable locations: when at PC, where to look at for

```
int a;
```

- Call Frame Information: stack (un|re)winding
- Special case: exception handlers

. . . with DWARF (2/3)

- Source code structure: declarations
- Source code locations: from PC to line:column
- Variable locations: when at PC, where to look at for

  ```
  int a;
  ```

- Call Frame Information: stack (un|re)winding
- Special case: exception handlers

. . . with DWARF (2/3)

- Source code structure: declarations
- Source code locations: from PC to line:column
- Variable locations: when at PC, where to look at for

```
int a;
```

- Call Frame Information: stack (un|re)winding
- Special case: exception handlers

. . . with DWARF (2/3)

- Source code structure: declarations
- Source code locations: from PC to line:column
- Variable locations: when at PC, where to look at for

  ```
  int a;
  ```
- Call Frame Information: stack (un|re)winding
- Special case: exception handlers

. . . with DWARF (3/3)

- Full support for C, C++, Fortran, Java, ObjectiveC, . . .
- Limited support for Ada, Cobol, D, PL/I
- More if you implement it!
- Extensible: "vendor additions"

. . . with DWARF (3/3)

- Full support for C, C++, Fortran, Java, ObjectiveC, . . .
- Limited support for Ada, Cobol, D, PL/I
- More if you implement it!
- Extensible: "vendor additions"

. . . with DWARF (3/3)

- Full support for C, C++, Fortran, Java, ObjectiveC, . . .
- Limited support for Ada, Cobol, D, PL/I
- More if you implement it!
- Extensible: "vendor additions"

. . . with DWARF (3/3)

- Full support for C, C++, Fortran, Java, ObjectiveC, . . .
- Limited support for Ada, Cobol, D, PL/I
- More if you implement it!
- Extensible: "vendor additions"

DWARF in ELF

- `readelf -w <elf>`
- `objdump --dwarf[=...] <elf>`
- `.debug_info`, `.debug_abbrev`, `.debug_loc`, `...`, `.eh_frame`

DWARF in ELF

- `readelf -w <elf>`
- `objdump --dwarf[=...] <elf>`
- `.debug_info`, `.debug_abbrev`, `.debug_loc`, `...`, `.eh_frame`

DWARF in ELF

- `readelf -w <elf>`
- `objdump --dwarf[=...] <elf>`
- `.debug_info`, `.debug_abbrev`, `.debug_loc`, `...`, `.eh_frame`

Plan

2 Source code structure

Source code structure

- Kind of central knowledge about the logical layout of the program.
- Organised as a big tree.
- Tell the debugger about declarations:
  - compilation units
  - types
  - global variables
  - subprograms (plus parameters, local variables)
  - etc.

Source code structure

- Kind of central knowledge about the logical layout of the program.
- Organised as a big tree.
- Tell the debugger about declarations:
  - compilation units
  - types
  - global variables
  - subprograms (plus parameters, local variables)
  - etc.

Source code structure

- Kind of central knowledge about the logical layout of the program.
- Organised as a big tree.
- Tell the debugger about declarations:
    - compilation units
    - types
    - global variables
    - subprograms (plus parameters, local variables)
    - etc.

Source code structure

- Kind of central knowledge about the logical layout of the program.
- Organised as a big tree.
- Tell the debugger about declarations:
    - compilation units
    - types
    - global variables
    - subprograms (plus parameters, local variables)
    - etc.

Source code structure

- Kind of central knowledge about the logical layout of the program.
- Organised as a big tree.
- Tell the debugger about declarations:
  - compilation units
  - types
  - global variables
  - subprograms (plus parameters, local variables)
  - etc.

Source code structure

- Kind of central knowledge about the logical layout of the program.
- Organised as a big tree.
- Tell the debugger about declarations:
  - compilation units
  - types
  - global variables
  - subprograms (plus parameters, local variables)
  - etc.

Example 1 — Source code

```c
#include <stdio.h>

void put_hello_world()
{
    puts("Hello, world!");
}

int main(void)
{
    put_hello_world();
    return 0;
}
```

Example 1 — objdump --dwarf=info (1/2)

```
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
   <c>   DW_AT_producer    : (indirect string, offset: 0x43):
                               GNU C 4.7.2
   <10>   DW_AT_language    : 1        (ANSI C)
   <11>   DW_AT_name        : (indirect string, offset: 0x59):
                               simple.c
   <15>   DW_AT_comp_dir    : (indirect string, offset: 0x0):
                               /tmp
   <19>   DW_AT_low_pc      : 0x4004fc
   <21>   DW_AT_high_pc     : 0x400521
   <29>   DW_AT_stmt_list   : 0x0
<1><2d>: Abbrev Number: 2 (DW_TAG_base_type)
   <2e>   DW_AT_byte_size   : 8
   <2f>   DW_AT_encoding    : 7        (unsigned)
   <30>   DW_AT_name        : (indirect string, offset: 0x87):
                               long unsigned int
<1><34>: Abbrev Number: 2 (DW_TAG_base_type)
   <35>   DW_AT_byte_size   : 1
   <36>   DW_AT_encoding    : 8        (unsigned char)
   <37>   DW_AT_name        : (indirect string, offset: 0x62):
                               unsigned char
```

Example 1 — objdump --dwarf=info (2/2)

```
<1><73>: Abbrev Number: 4 (DW_TAG_subprogram)
   <74>   DW_AT_external     : 1
   <75>   DW_AT_name         : (indirect string, offset: 0x20):
                              put_hello_world
   <79>   DW_AT_decl_file    : 1
   <7a>   DW_AT_decl_line    : 3
   <7b>   DW_AT_low_pc       : 0x4004fc
   <83>   DW_AT_high_pc      : 0x40050c
   <8b>   DW_AT_frame_base   : 0x0      (location list)
   <8f>   DW_AT_GNU_all_tail_call_sites: 1
<1><90>: Abbrev Number: 5 (DW_TAG_subprogram)
   <91>   DW_AT_external     : 1
   <92>   DW_AT_name         : (indirect string, offset: 0x82):
                              main
   <96>   DW_AT_decl_file    : 1
   <97>   DW_AT_decl_line    : 8
   <98>   DW_AT_prototyped   : 1
   <99>   DW_AT_type         : <0x57>
   <9d>   DW_AT_low_pc       : 0x40050c
   <a5>   DW_AT_high_pc      : 0x400521
   <ad>   DW_AT_frame_base   : 0x60     (location list)
   <b1>   DW_AT_GNU_all_tail_call_sites: 1
```

Example 2 — Source code

```c
#include <stdlib.h>

struct my_list
{
    unsigned value;
    struct my_list *next;
};


unsigned my_list_max(struct my_list *l)
{
    unsigned max = 0;
    while (l != NULL)
    {
        if (l->value > max)
            max = l->value;
        l = l->next;
    }
    return max;
}
```

Example 2 — objdump --dwarf=info (1/2)

```
 <1><49>: Abbrev Number: 4 (DW_TAG_structure_type)
    <4a>   DW_AT_name          : (indirect string, offset: 0x6c): my_list
    <4e>   DW_AT_byte_size     : 16
    <4f>   DW_AT_decl_file     : 1
    <50>   DW_AT_decl_line     : 3
    <51>   DW_AT_sibling       : <0x72>
 <2><55>: Abbrev Number: 5 (DW_TAG_member)
    <56>   DW_AT_name          : (indirect string, offset: 0x58): value
    <5a>   DW_AT_decl_file     : 1
    <5b>   DW_AT_decl_line     : 5
    <5c>   DW_AT_type          : <0x72>
    <60>   DW_AT_data_member_location: [...] (DW_OP_plus_uconst: 0)
 <2><63>: Abbrev Number: 5 (DW_TAG_member)
    <64>   DW_AT_name          : (indirect string, offset: 0x5e): next
    <68>   DW_AT_decl_file     : 1
    <69>   DW_AT_decl_line     : 6
    <6a>   DW_AT_type          : <0x79>
    <6e>   DW_AT_data_member_location: [...] (DW_OP_plus_uconst: 8)
[<0x72> = unsigned int]
 <1><79>: Abbrev Number: 6 (DW_TAG_pointer_type)
    <7a>   DW_AT_byte_size     : 8
    <7b>   DW_AT_type          : <0x49>
```

Example 2 — objdump --dwarf=info (2/2)

```
<1><7f>: Abbrev Number: 7 (DW_TAG_subprogram)
   <80>   DW_AT_external      : 1
   <81>   DW_AT_name          : [...] my_list_max
   <85>   DW_AT_decl_file     : 1
   <86>   DW_AT_decl_line     : 10
   <87>   DW_AT_prototyped    : 1
   <88>   DW_AT_type          : <0x72>
   <8c>   DW_AT_low_pc        : 0x0
   <94>   DW_AT_high_pc       : 0x3d
   <9c>   DW_AT_frame_base    : 0x0      (location list)
   <a0>   DW_AT_GNU_all_call_sites: 1
<2><a1>: Abbrev Number: 8 (DW_TAG_formal_parameter)
   <a2>   DW_AT_name          : l
   <a4>   DW_AT_decl_file     : 1
   <a5>   DW_AT_decl_line     : 10
   <a6>   DW_AT_type          : <0x79>
   <aa>   DW_AT_location      : [...] (DW_OP_fbreg: -40)
<2><ad>: Abbrev Number: 9 (DW_TAG_variable)
   <ae>   DW_AT_name          : max
   <b2>   DW_AT_decl_file     : 1
   <b3>   DW_AT_decl_line     : 12
   <b4>   DW_AT_type          : <0x72>
   <b8>   DW_AT_location      : [...] (DW_OP_fbreg: -20)
```

Beyond

- .debug_info works with .debug_abbrev
- Other data structures have their own constructs (union: DW_TAG_union_type, C++ class: DW_TAG_class_type)
- There is support for various language pecularities (artificial object this pointer, static link, etc.)
- Langage with "too advanced" features can use basic constructs to encode information.
- Or they can define their own entries and attributes.

Beyond

- `.debug_info` works with `.debug_abbrev`
- Other data structures have their own constructs (`union`: `DW_TAG_union_type`, C++ `class`: `DW_TAG_class_type`)
- There is support for various language pecularities (`artificial` object this pointer, static link, etc.)
- Langage with "too advanced" features can use basic constructs to encode information.
- Or they can define their own entries and attributes.

Beyond

- `.debug_info` works with `.debug_abbrev`
- Other data structures have their own constructs (`union`: `DW_TAG_union_type`, C++ `class`: `DW_TAG_class_type`)
- There is support for various language pecularities (`artificial` object `this` pointer, static link, etc.)
- Langage with "too advanced" features can use basic constructs to encode information.
- Or they can define their own entries and attributes.

Beyond

- `.debug_info` works with `.debug_abbrev`
- Other data structures have their own constructs (`union`: `DW_TAG_union_type`, C++ `class`: `DW_TAG_class_type`)
- There is support for various language pecularities (`artificial` object `this` pointer, static link, etc.)
- Langage with "too advanced" features can use basic constructs to encode information.
- Or they can define their own entries and attributes.

Beyond

- `.debug_info` works with `.debug_abbrev`
- Other data structures have their own constructs (`union`: `DW_TAG_union_type`, C++ `class`: `DW_TAG_class_type`)
- There is support for various language pecularities (`artificial` object `this` pointer, static link, etc.)
- Langage with "too advanced" features can use basic constructs to encode information.
- Or they can define their own entries and attributes.

Plan

3 Source code locations

Source code locations

- Goal: associate statements locations (line, filename) to PC values (both ways).
- Can be a very huge table for big compilation units.
- DWARF way: create a VM to build the table!
- Also contain other PC-dependant data (ARM instruction set, . . .)
- Located in `.debug_line`

Source code locations

- Goal: associate statements locations (line, filename) to PC values (both ways).
- Can be a very huge table for big compilation units.
- DWARF way: create a VM to build the table!
- Also contain other PC-dependant data (ARM instruction set, . . .)
- Located in `.debug_line`

Source code locations

- Goal: associate statements locations (line, filename) to PC values (both ways).
- Can be a very huge table for big compilation units.
- DWARF way: create a VM to build the table!
- Also contain other PC-dependant data (ARM instruction set, . . .)
- Located in `.debug_line`

Source code locations

- Goal: associate statements locations (line, filename) to PC values (both ways).
- Can be a very huge table for big compilation units.
- DWARF way: create a VM to build the table!
- Also contain other PC-dependant data (ARM instruction set, . . .)
- Located in .debug_line

Source code locations

- Goal: associate statements locations (line, filename) to PC values (both ways).
- Can be a very huge table for big compilation units.
- DWARF way: create a VM to build the table!
- Also contain other PC-dependant data (ARM instruction set, . . .)
- Located in `.debug_line`

Example — Source code

```c
unsigned my_list_max(struct my_list *l)
{
    unsigned max = 0;
    while (l != NULL)
    {
        if (l->value > max)
            max = l->value;
        l = l->next;
    }
    return max;
}
```

Example — Line number program

```
objdump --debug=rawline

[...]
The File Name Table:
 Entry Dir     Time     Size     Name
 1     0       0        0        lesssimple.c

Line Number Statements:
 Extended opcode 2: set Address to 0x0
 Advance Line by 10 to 11
 Copy
 Special opcode 118: advance Address by 8 to 0x8 and Line by 1 to 12
 Special opcode 104: advance Address by 7 to 0xf and Line by 1 to 13
 Special opcode 35: advance Address by 2 to 0x11 and Line by 2 to 15
 Special opcode 160: advance Address by 11 to 0x1c and Line by 1 to 16
 Special opcode 132: advance Address by 9 to 0x25 and Line by 1 to 17
 Extended opcode 4: set Discriminator to 1
 Special opcode 169: advance Address by 12 to 0x31 and Line by -4 to 13
 Special opcode 109: advance Address by 7 to 0x38 and Line by 6 to 19
 Special opcode 48: advance Address by 3 to 0x3b and Line by 1 to 20
 Advance PC by 2 to 0x3d
 Extended opcode 1: End of Sequence
```

Example — Line number table

```
objdump --debug=decodeline

CU: lesssimple.c:
File name      Line        Starting
               number      address
lesssimple.c       11              0

lesssimple.c       12            0x8
lesssimple.c       13            0xf
lesssimple.c       15           0x11
lesssimple.c       16           0x1c
lesssimple.c       17           0x25
lesssimple.c       13           0x31
lesssimple.c       19           0x38
lesssimple.c       20           0x3b
```

```
0000000000000000 <my_list_max>:
   0: push    %rbp
   1: mov     %rsp,%rbp
   4: mov     %rdi,-0x18(%rbp)
   8: movl    $0x0,-0x4(%rbp)
   f: jmp     31 <my_list_max+0x31>
  11: mov     -0x18(%rbp),%rax
  15: mov     (%rax),%eax
  17: cmp     -0x4(%rbp),%eax
  1a: jbe     25 <my_list_max+0x25>
  1c: mov     -0x18(%rbp),%rax
  20: mov     (%rax),%eax
  22: mov     %eax,-0x4(%rbp)
  25: mov     -0x18(%rbp),%rax
  29: mov     0x8(%rax),%rax
  2d: mov     %rax,-0x18(%rbp)
  31: cmpq    $0x0,-0x18(%rbp)
  36: jne     11 <my_list_max+0x11>
  38: mov     -0x4(%rbp),%eax
  3b: pop     %rbp
  3c: retq
```

Plan

4 Variable locations

Variable locations

- Knowing what can be accessed is good.
- *How* to access it?
- There is almost no rule!
- DWARF way: create a VM to evaluate *location expressions*!
- Located in `.debug_loc`

Variable locations

- Knowing what can be accessed is good.
- *How* to access it?
- There is almost no rule!
- DWARF way: create a VM to evaluate *location expressions*!
- Located in `.debug_loc`

Variable locations

- Knowing what can be accessed is good.
- *How* to access it?
- There is almost no rule!
- DWARF way: create a VM to evaluate *location expressions*!
- Located in `.debug_loc`

Variable locations

- Knowing what can be accessed is good.
- *How* to access it?
- There is almost no rule!
- DWARF way: create a VM to evaluate *location expressions*!
- Located in `.debug_loc`

Variable locations

- Knowing what can be accessed is good.
- *How* to access it?
- There is almost no rule!
- DWARF way: create a VM to evaluate *location expressions*!
- Located in `.debug_loc`

Example (1/2)

```
 <2><63>: Abbrev Number: 5 (DW_TAG_member)
    <64>   DW_AT_name        : (indirect string, offset: 0x5e): next
    <68>   DW_AT_decl_file   : 1
    <69>   DW_AT_decl_line   : 6
    <6a>   DW_AT_type        : <0x79>
    <6e>   DW_AT_data_member_location: 2 byte block: 23 8
                                        (DW_OP_plus_uconst: 8)
[...]
 <2><ad>: Abbrev Number: 9 (DW_TAG_variable)
    <ae>   DW_AT_name        : max
    <b2>   DW_AT_decl_file   : 1
    <b3>   DW_AT_decl_line   : 12
    <b4>   DW_AT_type        : <0x72>
    <b8>   DW_AT_location    : 2 byte block: 91 6c
                                 (DW_OP_fbreg: -20)
```

Example (2/2)

```
objdump --debug=info
```

```
 <1><7f>: Abbrev Number: 7 (DW_TAG_subprogram)
    <80>   DW_AT_external    : 1
    <81>   DW_AT_name        : [...] my_list_max
    <85>   DW_AT_decl_file   : 1
    <86>   DW_AT_decl_line   : 10
    <87>   DW_AT_prototyped  : 1
    <88>   DW_AT_type        : <0x72>
    <8c>   DW_AT_low_pc      : 0x0
    <94>   DW_AT_high_pc     : 0x3d
    <9c>   DW_AT_frame_base  : 0x0      (location list)
    <a0>   DW_AT_GNU_all_call_sites: 1
```

```
objdump --debug=loc
```

```
Offset   Begin            End              Expression
00000000 0000000000000000 0000000000000001 (DW_OP_breg7 (rsp): 8)
00000000 0000000000000001 0000000000000004 (DW_OP_breg7 (rsp): 16)
00000000 0000000000000004 000000000000003c (DW_OP_breg6 (rbp): 16)
00000000 000000000000003c 000000000000003d (DW_OP_breg7 (rsp): 8)
00000000 <End of list>
```

Plan

5 Call Frame Information / Exception handlers

Call Frame Information

- Debugging involves inspecting the whole stack.
- At one point, direct access to most recent call frame.
- To access other ones: stack unwinding.
- Located in `.eh_frame` (`.debug_frame`?)

Call Frame Information

- Debugging involves inspecting the whole stack.
- At one point, direct access to most recent call frame.
- To access other ones: stack unwinding.
- Located in `.eh_frame` (`.debug_frame`?)

Call Frame Information

- Debugging involves inspecting the whole stack.
- At one point, direct access to most recent call frame.
- To access other ones: stack unwinding.
- Located in `.eh_frame` (`.debug_frame`?)

Call Frame Information

- Debugging involves inspecting the whole stack.
- At one point, direct access to most recent call frame.
- To access other ones: stack unwinding.
- Located in `.eh_frame` (`.debug_frame`?)

Call Frame Information

- PC range for the targetted subprogram.
- Call Frame Address: stack pointer at the subprogram call site.
- A set of register used by the current subprogram (and then the values they contained is saved somewhere).

Call Frame Information

- PC range for the targetted subprogram.
- Call Frame Address: stack pointer at the subprogram call site.
- A set of register used by the current subprogram (and then the values they contained is saved somewhere).

Call Frame Information

- PC range for the targetted subprogram.
- Call Frame Address: stack pointer at the subprogram call site.
- A set of register used by the current subprogram (and then the values they contained is saved somewhere).

Call Frame Information (1/2)

```
objdump --debug=frames

00000000 00000014 00000000 CIE
  Version:               1
  Augmentation:          "zR"
  Code alignment factor: 1
  Data alignment factor: -8
  Return address column: 16
  Augmentation data:     1b

  DW_CFA_def_cfa: r7 (rsp) ofs 8
  DW_CFA_offset: r16 (rip) at cfa-8
  DW_CFA_nop
  DW_CFA_nop
```

Call Frame Information (2/2)

```
objdump --debug=frames
```

```
00000118 00000024 000000ec FDE cie=00000030
    pc=fffffffffffed2d0..fffffffffffed352
  DW_CFA_advance_loc: 10 to fffffffffffed2da
  DW_CFA_offset: r3 (rbx) at cfa-40
  DW_CFA_offset: r6 (rbp) at cfa-32
  DW_CFA_advance_loc: 13 to fffffffffffed2e7
  DW_CFA_offset: r12 (r12) at cfa-24
  DW_CFA_offset: r13 (r13) at cfa-16
  DW_CFA_advance_loc: 7 to fffffffffffed2ee
  DW_CFA_def_cfa_offset: 48
  DW_CFA_advance_loc1: 99 to fffffffffffed351
  DW_CFA_def_cfa_offset: 8
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
```

Exception handlers

- `.eh_frame` is inteded to be loaded in the process' memory.
- Zero-cost exceptions: do nothing particular in the fast path.
- When throwing an exception, call some runtime library.
- The runtime library uses DWARF information to unwind the stack until finding an exception handler.
- Language specific: not in the DWARF specification. . .

Exception handlers

- `.eh_frame` is inteded to be loaded in the process' memory.
- Zero-cost exceptions: do nothing particular in the fast path.
- When throwing an exception, call some runtime library.
- The runtime library uses DWARF information to unwind the stack until finding an exception handler.
- Language specific: not in the DWARF specification. . .

Exception handlers

- `.eh_frame` is inteded to be loaded in the process' memory.
- Zero-cost exceptions: do nothing particular in the fast path.
- When throwing an exception, call some runtime library.
- The runtime library uses DWARF information to unwind the stack until finding an exception handler.
- Language specific: not in the DWARF specification…

Exception handlers

- .eh_frame is inteded to be loaded in the process' memory.
- Zero-cost exceptions: do nothing particular in the fast path.
- When throwing an exception, call some runtime library.
- The runtime library uses DWARF information to unwind the stack until finding an exception handler.
- Language specific: not in the DWARF specification. . .

Exception handlers

- `.eh_frame` is inteded to be loaded in the process' memory.
- Zero-cost exceptions: do nothing particular in the fast path.
- When throwing an exception, call some runtime library.
- The runtime library uses DWARF information to unwind the stack until finding an exception handler.
- Language specific: not in the DWARF specification. . .

Plan

6 Conclusion

Conclusion

- The DWARF format itself is quite simple (once big tables are compressed :-)
- Producing DWARF (and debug information in general) is not straightforward.
- Even if it's architecture and language independant, there is a need to add support for it both to the compiler and to the debugger.

Conclusion

- The DWARF format itself is quite simple (once big tables are compressed :-)
- Producing DWARF (and debug information in general) is not straightforward.
- Even if it's architecture and language independant, there is a need to add support for it both to the compiler and to the debugger.

Conclusion

- The DWARF format itself is quite simple (once big tables are compressed :-)
- Producing DWARF (and debug information in general) is not straightforward.
- Even if it's architecture and language independant, there is a need to add support for it both to the compiler and to the debugger.