ARM 平台下的反汇编

目的

作为代码插桩过程的前提，首先需要对于所提供的二进制代码进行必要的分析，了解 ELF 文件的结构以及 ARM 平台的指令编码，将二进制 01 码翻译成为用户可读的汇编代码。通过对于汇编代码的分析，用户可以得到程序应用中各个函数起始地址以及程序各个模块的流程调用等重要信息，为代码插桩提供详细的数据。经过插桩的代码最后通过再一次汇编的过程输出到目标文件。因此，正确、快速地进行平台下的反汇编工作显得十分关键。

ARM 平台介绍[1-2]

ARM（Advanced RISC Machines）是微处理器行业的一家知名企业，设计了大量高性能、廉价、耗能低的 RISC（精简指令集计算机）处理器、相关技术及软件。技术具有性能高、成本低和能耗低等特点。 经历过早期自己设计和制造芯片的不景气之后，公司自己开始不制造芯片，只将芯片的设计方案授权（licensing）给其他公司，由它们来生产，形成了较为独特的盈利模式。
RISC 结构优先选取使用频率最高的简单指令，避免复杂指令；将指令长度固定，指令格式和寻地方式种类减少；以控制逻辑为主，不用或少用微码控制等。ARM 处理器在秉承 RISC 体系优点的基础上，进行了针对嵌入式系统的功能扩展，使得指令更加灵活，处理器性能在嵌入式平台上更加突出。
ARM 微处理器的核心结构如下图所示：

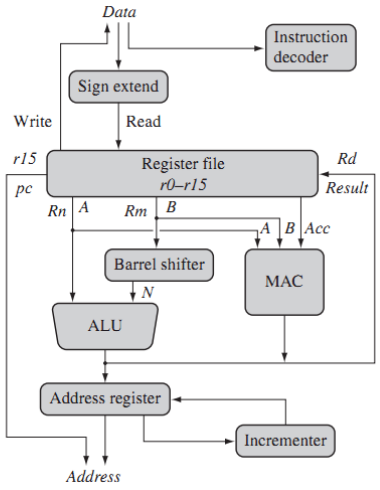

Figure 1．ARM 处理器核心结构示意图[2]

数据指令通过数据总线进入到处理器核心，然后在指令被执行之前经由指令解码器翻译。和所有精简指令集处理器一样，ARM采用了load-store架构，load指令将数据从内存拷贝到寄存器，store指令将数据从寄存器转储到内存，所有的数据处理在寄存器中完成。
ARM处理器是32位的处理器，所有的指令默认将寄存器视为32位的值，因此Sign extend会在数据写入寄存器之前将所有8位或者12位的数值转换为32位的数值。ARM指令通常有两个源寄存器: Rn, Rm 以及一个目标寄存器，操作数都是从寄存器通过内部总线读取得到。
核心的ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) 从内部总线A,B上取得操作数进行运算，然后将结果写入目标寄存器。
ARM处理器一个很大的特色是寄存器Rm可以选择性地在进入ALU运算之前在barrel shifter中进

行预处理，barrel和shifter的结合可以高效地计算计算许多复杂的表达式和地址。
对于load和store指令操作来说，最后的incrementer用来更新地址寄存器在读写下一个寄存器的值到下一个内存地址之前。

ARM 处理器的架构形成了以下几个特点：

1. 体积小、低功耗、低成本、高性能。这一特点使得 ARM 处理器在移动设备上得到广泛的应用，这也是 ARM 处理器能够占有移动设备市场 80%份额的强有力原因。
2. 支持 Thumb（16 位）/ARM（32 位）双指令集。ARM 指令集支持 ARM 核所有的特性，具有高效、快速的特点；Thumb 指令集具有灵活、小巧的特点。双指令集的特点使得 ARM 处理器能够很好地兼容 8 位/16 位器件，使得代码更加紧凑。指令长度的固定在一定程度上方便了 ELF 文件的分析以及插桩工作的进行。另外在 ARM 的指令系统中设计者还加入了一些在嵌入式系统中经常用到的一些 DSP 操作指令，使得一些系统的设计没有考虑添加额外的 DSP 模块来支持系统的功能。
3. 大量使用寄存器，指令执行速度更快，大多数数据操作都可以在寄存器中直接完成，大大增强了地址处理的执行效率。
4. 寻址方式灵活简单，执行效率高；ARM 拥有常见的寻址方式之外，另外还增加了一些非常实用的寻址方式，如：
   - 多寄存器寻址：MOV R1, {R2-R4, R6}.多寄存器寻址一次可传送几个寄存器值，允许一条指令传送 16 个寄存器的任何子集或所有寄存器。它还将寄存器之间的简单加减操作囊括于指令当中，大大精简了这些简单指令所占有的代码空间，也使得程序的可读性有所提升。
   - 寄存器移位寻址：MOV R0, R2, LSL #3. ARM 处理器中特有的 Barrel Shifter 使得数据的移位操作能够在运算单元之外进行，方便了程序代码的编写，同时也提高了代码执行的效率。
   - 堆栈寻址：LDMFD SP!, {R1-R7, PC}。 堆栈是一个按特定顺序进行存取的存储区，操作顺序为“后进先出” 。堆栈寻址是隐含的，它使用一个专门的寄存器(堆栈指针)指向一块存储区域(堆栈)，指针所指向的存储单元即是堆栈的栈顶。存储器堆栈根据增长方向可分为两种：递增和递减；根据栈顶指向的位置内容又可分为满堆栈和空堆栈。这些寻址方式的选项可以满足各种堆栈的需求，将堆栈的内容直接作为操作数参与运算。

   这些新的寻址方式的增加极大地方便了汇编程序的编写，当然同时也使得编译的结果更为整洁精炼。
5. 指令数据并行存取。由于数据的存取操作在指令流中占据了将近四分之一的比例。一个基本的 load 操作需要 3 个时钟周期，一个 store 操作也需要 2 个时钟周期，这大大影响到了平均 CPI——指令的执行效率，因此如何有效地优化这些指令在高速处理器中显得格外重要。在 ARM 9 处理器中采用了 Harvard Memory Architecture， 这使得指令的获取和数据的读取能够同时进行。[3]

ARM7TDMI Pipeline Operation

Fetch	Decode	Execute
Instruction Fetch	Convert Thumb to ARM Main Decode Register Address Decode	Register Read Shifter ALU Writeback

ARM9TDMI Pipeline Operation

Fetch	Decode	Execute	Memory	Writeback
Instruction Fetch	ARM Decode Reg. Address Decode Register Read Thumb Decode Reg. Address Decode Register Read	Shifter ALU	Memory Data access	ALU Result and / or Load data Writeback

Figure 2． ARM 7 与 ARM 9 指令执行流程对比图[3]

由图可见，在 ARM 9 的指令执行流程中额外增加了两步，这使得原本拥挤的执行步骤可以将任务相对平均地分布到其它步骤中去，由此增大了指令的执行频率。另外在解码阶段，在 ARM 9 中更是增设了一个 Thumb decoder，使得两种指令的解码得以并行处理，也省去了将 thumb 指令转换为 ARM 指令带来的麻烦与时间消耗。

ARM 处理器的这些设计要点使得处理器在性能、能耗方面都有很好的表现，这也正是 ARM 处理器能被广泛应用于手机等移动电子产品上的重要原因。

ARM 平台下可执行文件的反汇编

ELF 文件简介[4]

ELF (Executable and Linkable Format)是 UNIX 类操作系统中普遍采用的目标文件格式，分为三种类型：

- 可重定位文件 (Relocatable File)包含适合于与其他目标文件链接来创建可执行文件或者共享目标文件的代码和数据。
- 可执行文件 (Executable File)包含适合于执行的一个程序，此文件规定了 exec() 如何创建一个程序的进程映像。
- 共享目标文件(Shared Object File)包含可在两种上下文中链接的代码和数据。首先链接编辑器可以将它和其它可重定位文件和共享目标文件一起处理，生成另外一个目标文件。其次，动态链接器(Dynamic Linker)可能将它与某个可执行文件以及其它共享目标一起组合，创建进程映像。

目标文件全部是程序的二进制表示，目的是直接在某种处理器上直接执行。

ELF 文件剖析

ELF 文件在不同的视图下有不同的文件结构，共有两种视图：链接视图和执行视图。

Figure 3．ELF 文件内容结构示意图（链接视图、执行视图）[2]

两种结构视图的顶部都含有一些头部信息，包括文件的类型、可执行的平台信息以及一些头部表的属性信息：表的开始字节、表项大小以及表项数量。ELF 文件中有两个非常重要的头部表格：节区头部表、程序头部表。

- 节区头部表包含了文件中所有节区的基本信息，每一个节区在表中都有一个项，描述了该节区的名字、大小、偏移位置等关键信息。节区头部表在链接的过程中指导链接器将相同属性的节区合并成为一个段，因此需要进行链接的文件必须包含节区头部表。

- 可执行文件或者共享目标文件的程序头部是一个结构数组，每个结构描述了一个段或者系统准备程序执行所必需的其它信息。目标文件的“段”包含一个或者多个链接视图下的“节区”，也就是“段内容(Segment Contents)”。程序头部表仅对于可执行文件和共享目标文件有意义，它指导系统如何去创建一个进程映像。

链接视图下保留的节区有.text(代码段), .data(初始化数据段), .bss(未初始化数据段), .symtab(符号表，包含用来定位、重定位程序中符号定义和引用的信息), .strtab(字符串表，包括头部表或程序中字符串的引用映射)等等，非保留的节区中包括用户自定义的函数等。用户可以在编译的时候增加选项，根据需求指定这些特殊节区的起始位置、对齐方式等。
值得注意的是，符号表中的所包含的通常是一些静态变量或者函数的定义位置，但是由于通常文件与文件之间存在函数和变量的调用关系，在文件进行链接生成可执行文件之前，符号表结构体中的值是未定义的，必须通过链接器确定函数和变量在输出可执行文件中的确切偏移位置。

ELF 文件的查看工具

现今已存在一些 ELF 文件查看工具，如通用的 objdump 以及经常的 readelf，通过这些工具我们可以查看 ELF 文件的具体内容，方便进行文件的分析。

objdump

objdump 是以一种可阅读的格式让你更多地了解二进制文件带有的信息的工具。objdump 借助 BFD，更加通用一些, 可以应付不同文件格式，它提供反汇编的功能。
由于本项目中所要解析的是在 ARM 平台下可执行的 ELF 文件，gcc 自带的 objdump 工具不支持 arm 平台下的编译文件。因此需要构建 linux 下交叉编译环境，在基于 ARM 的嵌入式系统开发中，常常用到交叉编译的 GCC 工具链有两种：arm-linux-*和 arm-elf-*，两者区别主要在于使用不同的 C 库文件。arm-linux-*使用 GNU 的 Glibc，而 arm-elf-*一般使用 uClibc/uC-libc 或者使用 REDHAT 专门为嵌入式系统的开发的 C 库 newlib。两类交叉编译环境的构建比较繁琐，一般用户可以下载构建好的交叉编译工具，这样用户可以直接使用对应的 arm-linux-objdump 或 arm-elf-objdump 查看相应 ELF 文件的可读反汇编结果。使用者可以通过 arm-*-objdump –D *.elf 直接查看反汇编的结果：

```
shell.elf:     file format elf32-littlearm

Disassembly of section .text:

a0000000 <_fclose_r>:
a0000000:       e92d4070        push    {r4, r5, r6, lr}
a0000004:       e2515000        subs    r5, r1, #0      ; 0x0
a0000008:       e1a06000        mov     r6, r0
a000000c:       01a04005        moveq   r4, r5
a0000010:       0a00002b        beq     a00000c4 <_fclose_r+0xc4>
a0000014:       eb0000d5        bl      a0000370 <__sfp_lock_acquire>
a0000018:       e3560000        cmp     r6, #0  ; 0x0
a000001c:       0a000002        beq     a000002c <_fclose_r+0x2c>
a0000020:       e5963038        ldr     r3, [r6, #56]
a0000024:       e3530000        cmp     r3, #0  ; 0x0
a0000028:       0a000027        beq     a00000cc <_fclose_r+0xcc>
a000002c:       e1d540fc        ldrsh   r4, [r5, #12]
a0000030:       e3540000        cmp     r4, #0  ; 0x0
a0000034:       0a000021        beq     a00000c0 <_fclose_r+0xc0>
a0000038:       e1a00006        mov     r0, r6
a000003c:       e1a01005        mov     r1, r5
a0000040:       eb000030        bl      a0000108 <_fflush_r>
a0000044:       e595302c        ldr     r3, [r5, #44]
a0000048:       e3530000        cmp     r3, #0  ; 0x0
a000004c:       e1a04000        mov     r4, r0
a0000050:       0a000005        beq     a000006c <_fclose_r+0x6c>
a0000054:       e1a00006        mov     r0, r6
```

Figure 4．Arm-linux-objdump 反汇编结果

当然用户还可以通过参数的设定有选择地进行文件内容信息的查看。

readelf

另一个常用的查看工具是 readelf，readelf 则并不借助 BFD，而是直接读取 ELF 格式文件的信息，得到的信息也略细致一些。通过指令 readelf –a *.elf 用户可以看到文件的具体结构和内容对照。

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            ARM
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0xa000c920
  Start of program headers:          52 (bytes into file)
  Start of section headers:          358640 (bytes into file)
  Flags:                             0x2, has entry point, GNU EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         35
  Section header string table index: 32

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        a0000000 008000 00c920 00  AX  0   0  4
  [ 2] .text.main        PROGBITS        a000c920 014920 000390 00  AX  0   0  4
  [ 3] .text.cprintf     PROGBITS        a000ccb0 014cb0 000090 00  AX  0   0  4
  [ 4] .text.pt_cread    PROGBITS        a000cd40 014d40 000110 00  AX  0   0  4
```

Figure 5．Readelf 执行结果

由图可见，readelf 将 ELF 文件信息按照字节顺序将一些重要的信息以可读的形式显示出来。但是 readelf 本身不具备反汇编的功能，由于本项目需要对于汇编指令进行分析和插桩工作，还需要对于汇编代码进行流程的分析，现有的 ELF 文件查看工具显然不能够满足这些具体的需求，因此需要在一些开源工具的基础上进行功能的扩展或者根据 ELF 文件描述文档独立开发出新的带有插桩功能的反汇编工具。

代码控制流程图的生成

通常代码插桩的位置位于函数的出入口、循环的头尾等地方，为了保证插桩的准确性和全面行，插桩程序需要对于汇编程序的代码块绘制详细的控制流程图，清楚地了解各个代码块之间的调用跳转关系。Diablo 就是这样一个满足这类需求的工具。

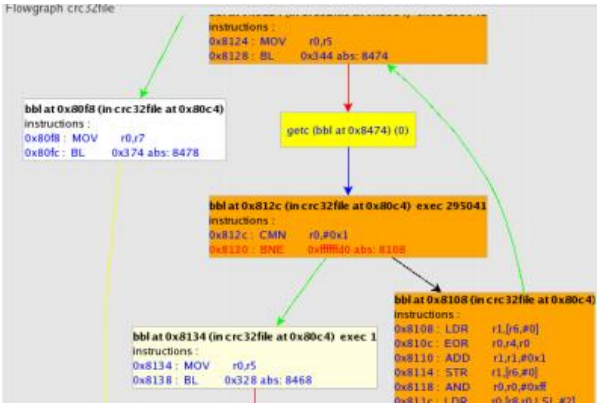

Figure 6．程序指令流程控制图

Diablo 是一个二进制代码编辑的架构，它支持 ARM,X86 等多种目标平台下可执行文件格式。Diablo 的核心和后台提供了读写目标文件、汇编反汇编代码、创建控制流程图(Interprocedural Control Flow Graph) 等功能，同时它的数据结构分为架构相关和架构无关两个部分，方便架构依赖和架构无关的代码分析。[5-6]

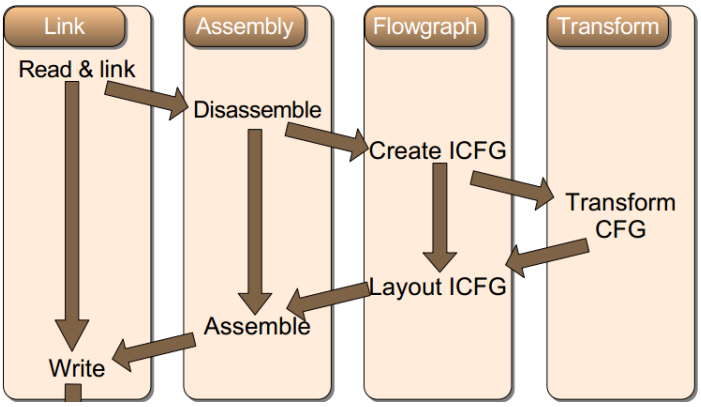

Figure 7． Diablo 执行流程

Diablo 的工作过程分为 4 个流程，链接、编译、流程图绘制以及流程图转换。它分析链接后反编译的代码绘制出流程之间的控制逻辑关系，根据调用关系以及实际的需求进行代码的插桩和优化工作。由于 Diablo 是开源的，可以利用它源代码中流程控制图的部分，生成流程控制图，继而进行代码的插桩工作。

小结

在代码插桩的过程中，反汇编和流程控制图的生成是极为基础，也是关键的两个前提步骤。这需要程序员对于所用平台的二进制文件的结构和内容都有全面的了解，充分利用现有的工具和开放源码，对文件进行必要的解析和流程分析，根据用户定制的代码插桩要求进行插桩工作。

参考资料

1. 百度百科. *ARM*. Available from: http://baike.baidu.com/view/11200.htm.
2. Seal, D., *ARM Architecture Reference Manual*. 2000.
3. *The ARM9 Family - High Performance Microprocessors for Embedded Applications*, in *Proceedings of the International Conference on Computer Design*. 1998, IEEE Computer Society. p. 230.
4. 滕启明, *ELF 文件格式分析*, in *JBEOS-TN-03-005*. 2003, 北京大学信息科学技术学院操作系统实验室.
5. Bus, B.D., et al., *The design and implementation of FIT: a flexible instrumentation toolkit*, in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2004, ACM: Washington DC, USA. p. 29-34.
6. Ghent University, B. *Diablo Manual*. Available from: http://diablo.elis.ugent.be/manual_main.

[6]