汇编语言与逆向技术

第7章 华为鲲鹏处理器汇编编程

王志
zwang@nankai.edu.cn

南开大学 网络空间安全学院

2024-2025学年

本章知识点

- 数据类型和寄存器
  - 重点：CPSR寄存器
- ARM指令
  - 难点：执行条件（Conditional Code）
- 寻址方式
  - 难点：多寄存器寻址、栈寻址
- 伪指令

数据类型和寄存器

数据定义

.byte	单字节定义	.byte 0x12, ‘a’, 23
.short	定义2字节数据	.short 0x1234, 65535
.long /.word	定义4字节数据	.word 0x12345678
.quad	定义8字节	.quad 0x1234567812345678
.float	定义浮点数	.float 0f3.2
.string /.asciz /.ascii	定义字符串	.ascii “abcd\0”

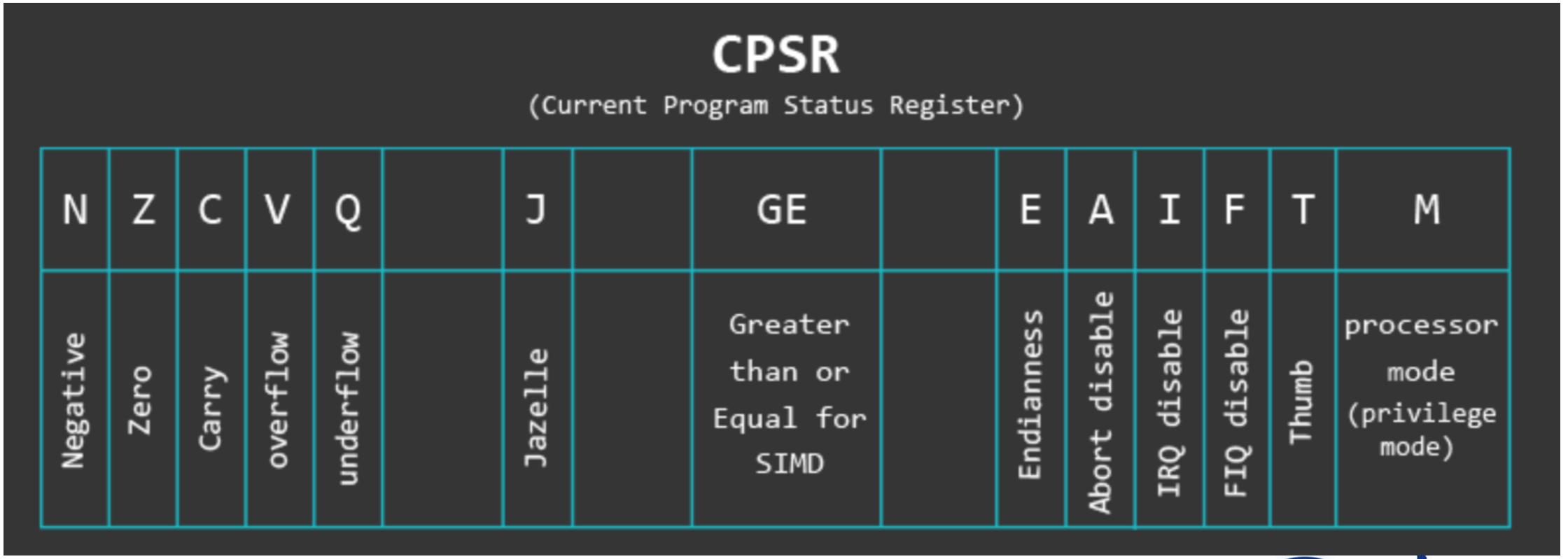
寄存器

- AArch32
  - 13个32位通用寄存器（R0-R12）
  - R13：栈指针SP
  - R14：链接寄存器LR
  - R15：程序计数器PC

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

CPSR寄存器

- 当前程序状态寄存器CPSR（Current Program Status Register）



CPSR寄存器

Flag	Description
N (Negative)	Enabled if result of the instruction yields a negative number.
Z (Zero)	Enabled if result of the instruction yields a zero value.
C (Carry)	Enabled if result of the instruction yields a value that requires a 33rd bit to be fully represented.
V (Overflow)	Enabled if result of the instruction yields a value that cannot be represented in 32 bit two's complement.
E (Endian-bit)	ARM can operate either in little endian, or big endian. This bit is set to 0 for little endian, or 1 for big endian mode.
T (Thumb-bit)	This bit is set if you are in Thumb state and is disabled when you are in ARM state.
M (Mode-bits)	These bits specify the current privilege mode (USR, SVC, etc.).
J (Jazelle)	Third execution state that allows some ARM processors to execute Java bytecode in hardware.

多选题 1分

设置

ARM体系中，当前程序的执行状态存储在哪个寄存器中？

A SP

B LR

C PC

D CPSR

提交

南開大學 Nankai University

ARM指令

ARM指令

- MNEMONIC{S}{condition} {Rd}, Operand1, Operand2
  - MNEMONIC：指令助记符
  - {S}：是否将执行结果更新到CPSR寄存器
  - {condition} ：指令的执行条件
  - {Rd}： Register (destination) for storing the result of the instruction
  - Operand1： First operand. Either a register or an immediate value
  - Operand2： Second (flexible) operand. Can be an immediate value (number) or a register with an optional shift

ARM指令

- GNU ARM汇编语言语法格式：

[<label>:][<instruction or directive or pseudo-instruction>}@comment

instruction 指令 ： LDR R2, [R0]

direction 伪操作

pseudo-instruction 伪指令

<label>:为标号，GRM ARM汇编中，任何以冒号结尾的标识符都认为是一个标号

comment为语句的注释

ARM指令

- GNU ARM汇编语言语法格式：

[<label>:][<instruction or directive or pseudo-instruction>}@comment

注：

- ARM指令、伪指令、伪操作、寄存器名不可大小写混用，可全部大写或全部小写
- 可以在行末用“\”表示换行

GNU ARM汇编语法格式

- 特殊字符和语法
  - 代码中的注释符号：‘@’
  - 整行注释符号：‘# ’
  - 语句分离符号：‘;’
  - 立即数前缀：‘#’或‘$’

MOV指令

- MOV{S} {cond} Rd, Rm
  - ARM中，MOV指令实现寄存器之间的数据移动和将立即数写入寄存器
    - MOV R1，R0
    - MOV R2，#2
  - **MOV指令不能访问内存数据**
  - Rd是目的寄存器，Rm是源寄存器或者立即数
  - S：操作结果是否更新CPSR，MOV不更新，MOVS更新

S

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

cond

is an optional condition code.

https://developer.arm.com/documentation/dui0473/m/arm-and-thumb-instructions/mov

数据传输指令

MOV 数据传送指令

MOV R0, #0xFF000　　立即寻址，将立即数**0xFF000**装入**R0**寄存器

MOV R1, R2　　寄存器寻址，将**R2**的值存入**R1**

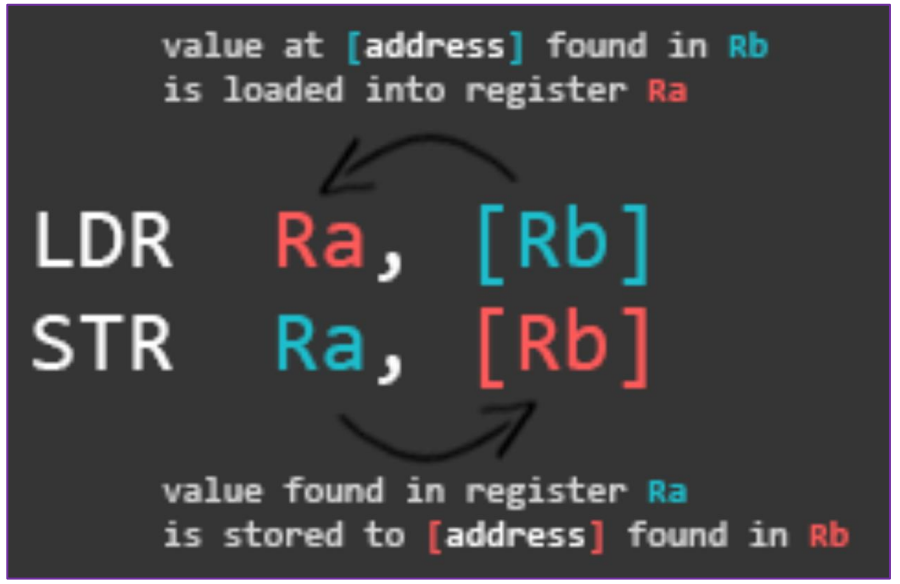
MOVS R0, R2, LSL #3　　移位寻址，**R2**的值左移**3**位，结果放入**R0**，**更新CPSR**

MVN 数据取反传送指令（bitwise logical negate）

MVN R0, #0　　**R0=-1**

内存访问指令

- ARM指令中只有**LDR**和**STR**指令可以访问内存
  - 数据必须从内存读入寄存器之后才可以操作
- LDR 将内存中的数据读入寄存器
  - LDR R2, [R0]
- STR 将寄存器中的数据写入内存
  - STR R2, [R1]
- 注意：源操作数和目的操作数的位置不同



分支和控制指令

Mnemonic	Brief description	See
`B{cc}`	Branch {conditionally}	*B, BL, BX, and BLX*
`BL`	Branch with Link	*B, BL, BX, and BLX*
`BLX`	Branch indirect with Link	*B, BL, BX, and BLX*
`BX`	Branch indirect	*B, BL, BX, and BLX*

分支和控制指令

条件编码和对应的标志位

Condition Code	Meaning (for cmp or subs)	Status of Flags
EQ	Equal	Z==1
NE	Not Equal	Z==0
GT	Signed Greater Than	(Z==0) && (N==V)
LT	Signed Less Than	N!=V
GE	Signed Greater Than or Equal	N==V
LE	Signed Less Than or Equal	(Z==1) || (N!=V)
CS or HS	Unsigned Higher or Same (or Carry Set)	C==1
CC or LO	Unsigned Lower (or Carry Clear)	C==0

分支和控制指令

- 大部分ARM指令都可以加执行条件
  - A feature of the ARM instruction set is that nearly all instructions are conditional

```
.global main


main:
        mov     r0, #2     /* setting up initial variable */
        cmp     r0, #3     /* comparing r0 to number 3. Negative bit get's set to 1 */
        addlt   r0, r0, #1 /* increasing r0 IF it was determined that it is smaller (lower than) number 3 */
        cmp     r0, #3     /* comparing r0 to number 3 again. Zero bit gets set to 1. Negative bit is set to 0
        addlt   r0, r0, #1 /* increasing r0 IF it was determined that it is smaller (lower than) number 3 */
        bx      lr
```

分支和控制指令

- 无条件跳转指令B
- 有条件跳转指令BXX，XX是条件编码

```
main:
        mov     r1, #2      /* setting up initial variable a */
        mov     r2, #3      /* setting up initial variable b */
        cmp     r1, r2      /* comparing variables to determine which is bigger */
        blt     r1_lower    /* jump to r1_lower in case r2 is bigger (N==1) */
        mov     r0, r1      /* if branching/jumping did not occur, r1 is bigger (or the same) so store r1 into r
        b       end         /* proceed to the end */
r1_lower:
        mov r0, r2          /* We ended up here because r1 was smaller than r2, so move r2 into r0 */
        b end               /* proceed to the end */
```

算数运算指令

ADD加法指令

ADD R0, R1, #5　　@R0←R1+5

ADD R0, R1, R2　　@R0←R1+R2

ADDS R0, R1, R2, LSL #5　　@R0←R1+R2左移5位，更新CPSR

SUB减法指令

SUB R0, R1, R2　　R0←R1-R2

SUBS R0, R1, R2, LSL #5　　R0←R1-R2左移5位，更新CPSR

逻辑运算指令

AND逻辑与指令

AND R0, R0, #5　　保持R0的第0位和第2位，其余位清0

ORR逻辑或指令

ORR R0, R0, #5　　R0的第0位和第2位设置为1，其余位不变

EOR逻辑异或指令

EORS R0, R0, #5　　R0的第0位和第2位取反，其余位不变，更新CPSR

移位运算指令

指令	说明
ASR （Arithmetic Shift Right）	算数右移 >> （结果带符号）
LSL （Logical Shift Left）	逻辑左移 <<
LSR （ Logical Shift Right）	逻辑右移 >>
ROR （Rotate Right）	循环右移：头尾相连
SXTB、SXTH、 UXTB、UXTH	字节、半字、字符号/0扩展移位运算

多选题 1分

设置

以下哪些ARM指令可以访问内存?

A LDR

B MOV

C ADD

D STR

提交

南開大學 Nankai University

多选题 1分

设置

以下哪些ARM指令会影响CPSR?

A MOV

B ADD

C ADDS

D EORS

提交

主观题 10分

设置

MOV R1, #5
MOV R0, R1, LSL #2
寄存器R0的值是多少?

作答

主观题 10分

设置

```
MOV R0, #4
CMP R0, 5
ADDLE R0, R0, #3
CMP R0, 5
ADDLE R0, R0, #3
```

寄存器R0的值是多少?

作答

寻址方式

ARM寻址方式

- ARM的寻址方式：

1. 立即数
2. 寄存器
3. 寄存器间接寻址
4. 基址寻址
5. 多寄存器寻址
6. 栈寻址

立即数

- 立即数是操作数
- 若操作数为常量，用#表示常量
- 0x或&表示16进制数，否则表示十进制数

例：

MOV R0, #0xFF000　　将立即数0xFF000装入R0寄存器

SUB R0, R0, #64　　R0减去64，结果放入R0

主观题 10分

设置

masm32位汇编语言，16进制数如何表示?

作答

寄存器

- 根据寄存器编码获取寄存器内存储的操作数

例：

MOV R1, R2　　将R2的值存入R1

SUB R0, R1, R2　　将R1的值减去R2的值，结果保存到R0中

寄存器间接寻址

- 操作数从寄存器所指向的内存中取出，寄存器存储的是内存地址

例：

LDR R1, [R2]　　将R2指向的存储单元的数据读出，存入R1

STR R1, [R2]　　将R1的值写入到R2所指向的存储单元

SWP R1, R1, [R2]　　将R1中的值和R2所指向的存储单元的内容交换

注：LDR（Load Register）指令用于读取内存数据

STR（Store Register）指令用于写入内存数据

SWP Rd, Rm, [Rn]，将[Rn]读入Rd，将Rm写入[Rn]

基址寻址

- 基址寄存器的内容与指令中的偏移量相加，得到有效操作数的地址后访问该地址空间

1. 前索引

```
LDR R0, [R1, #4]          // R0 ← mem32[R1 + 4]
```

R1中保存的地址+4，新地址中的值存入R0

基址寻址

- 基址寄存器的内容与指令中的偏移量相加，得到有效操作数的地址后访问该地址空间

2. 自动索引

LDR R0, [R1, #4]! // R0 ← mem32[**R1 + 4**]；R1 ← R1 + 4

在前索引的基础上，**新地址回写进R1**

注：！表示回写地址

基址寻址

- 基址寄存器的内容与指令中的偏移量相加，得到有效操作数的地址后访问该地址空间

3. 后索引

LDR R0, [R1], #4 // R0 ← mem32[R1]；R1 ← R1 + 4

R1中保存地址的内容写进R0，R1中保存的地址+4再写进R1

多寄存器寻址类型

- 多寄存器寻址也称为块拷贝寻址
- LDM（Load Multiple）多数据加载
  - 将多个地址上的值加载到多个寄存器上
- STM（Store Multiple）多数据存储
  - 将多个寄存器上的值存到多个地址上

多寄存器寻址格式

- LDM{cond} mode Rn{!}, reglist{^}
- STM{cond} mode Rn{!}, reglist{^}
  - Rn: 基址寄存器，存储传输数据的起始地址，Rn不能为R15
  - ！：表示最后的地址写回到Rn中
  - reglist：寄存器列表，用“，”隔开，如{R1，R2，R3}
    - 最多16个寄存器
  - ^：不允许在用户模式和系统模式下运行

多寄存器寻址模式

- IA(Increase After)每次传送后地址加4，寄存器从左到右执行
  - STMIA R0, {R1, LR} 先存R1，再存LR
- IB(Increase Before)每次传送前地址加4
- DA（Decrease After）每次传送后地址减4
- DB（Decrease Before）每次传送前地址减4

多寄存器寻址实例

- MOV R1, #0x10000000 //数据传输起始地址0x10000000
- LDMIB R1！，{R0, R4-R6} // 从左向右加载
  - R1地址加4，LDR R0, #0x10000004
  - R1地址加4， LDR R4, #0x10000008
  - R1地址加4， LDR R5, #0x1000000C
  - R1地址加4， LDR R6, #0x10000010
  - MOV R1, #0x10000010 // “!”，最后的地址写回R1寄存器

主观题 10分

设置

LDR R1, #0x10000000 //数据传输起始地址0x10000000
LDMIA R1！，{R0, R4-R6}
请写出LDMIA指令执行之后R1寄存器的值？R6寄存器存储数据的内存地址？

作答

多寄存器寻址实例

- LDR R1, #0x10000000 //数据传输起始地址0x10000000
- LDMIA R1！，{R0, R4-R6} // 从左向右加载
  - LDR R0, #0x10000000， R1地址加4
  - LDR R4, #0x10000004， R1地址加4
  - LDR R5, #0x10000008， R1地址加4
  - LDR R6, #0x1000000C， R1地址加4
  - LDR R1, #0x10000010 // “!”，最后的地址写回R1寄存器

多寄存器寻址实例

- MOV R1, #0x10000000 //数据传输起始地址0x10000000
- MOV R4, #0x10
- MOV R5, #0x20
- MOV R6, #0x30
- STMIB R1, {R4-R6} //从左向右加载

- 相当于
  - R1地址加4，STR R4, 0x10000004
  - R1地址加4，STR R5, 0x10000008
  - R1地址加4，STR R6, 0x1000000C
  - R1的值是#0x10000000

主观题 10分

设置

MOV R1, #0x1000000C //数据传输起始地址0x10000000
MOV R4, #0x10
MOV R5, #0x20
MOV R6, #0x30
STMDB R1！{R4-R6}
请写出STMDB指令执行之后R1寄存器的值？R5寄存器的值写入的内存地址？

作答

栈寻址

- 存储空间中的栈与寄存器组之间的批量数据传输
  - 使用R13（SP）作为栈指针，指向栈顶
  - 先进后出FILO方式访问
- 基本指令：LDM/STM

栈寻址

- 栈指针位置
  - FULL：栈指针指向栈顶元素（即最后一个入栈的数据元素）时称为FULL栈
  - EMPTY：栈指针指向与栈顶元素相邻的一个可用单元时称为EMPTY栈
- 栈增长方向
  - Descending：当数据栈向内存地址减小的方向增长时，称为Descending栈
  - Ascending：当数据栈向内存地址增加的方向增长时，称为 Ascending栈

栈寻址

- 栈的4种管理方式
  - FD （Full Descending Stack）：满递减栈，每次传送后栈指针减4，对应DA
  - FA（Full Ascending Stack）：满递增栈，每次传送后栈指针加4，对应IA
  - ED （Empty Descending Stack）：空递减栈，每次传送前栈指针减4，对应DB
  - EA （Empty Ascending Stack）：空递增栈，每次传送前栈指针加4，对应IB

多选题 1分

设置

ARMv8支持4种栈的生长（管理）方式：FA、FD、EA、ED。大家回忆一下，x86 CPU支持哪种栈的生长方式？

A FA

B FD

C EA

D ED

提交

栈寻址

- LDMFD SP!, {R2-R4}

<Rn>

The base register. If it is the SP and ! is specified, the instruction is treated as described in *POP*.

  - **数据出栈**，将数据从内存写入**R2-R4寄存器**
  - R2←mem32[SP]，SP地址加4，
  - R3←mem32[SP]，SP地址加4，
  - R4←mem32[SP]，SP地址加4，
  - 最后的地址再写回SP

https://developer.arm.com/documentation/ddi0403/d/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-ARMv7-M-Thumb-instructions/LDM--LDMIA--LDMFD

主观题 10分

设置

SP寄存器的值是0x10000010，指令STMFD SP!, {R2-R4}执行之后，SP寄存器的值是多少？

作答

栈寻址

- STMFD SP!, {R2-R4}
  - SP的初始值0x10000010
  - 数据入栈，将R2-R4寄存器值写入内存
  - SP地址减4， 0x1000000C，mem32[SP] ←R2
  - SP地址减4，0x10000008，mem32[SP] ←R3
  - SP地址减4，0x10000004，mem32[SP] ←R4
  - 最后的地址再写回SP
    - 执行之后SP的值是0x10000004

<Rn>

The base register. If it is the SP and ! is specified, the instruction is treated as described in *PUSH*.

https://developer.arm.com/documentation/ddi0403/d/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-ARMv7-M-Thumb-instructions/STMDB--STMFD

PC相对寻址

- 由程序计数器（PC）提供基准地址，指令中的地址码字段作为偏移量，二者相加后得到的地址则为目标操作数的有效地址
- LDR R0, [PC, #8]
  - 将PC的当前值加上8作为目标地址
  - 从目标地址加载数据到寄存器R0中

寄存器移位寻址

- 在执行数据处理指令时动态地对寄存器中的数据进行移位或旋转操作，作为操作的源操作数
- ADD R2, R1, R0, LSL #2
  - 将R0寄存器中的值左移2位
  - 左移结果与R1寄存器中的值相加
  - 最终结果存储到R2寄存器中

伪指令

伪指令

- 伪指令是编译器支持的指令，而非硬件芯片支持的指令
  - 编译器把伪指令转化成对应芯片支持的指令
  - 伪操作：

    数据定义伪操作、汇编控制伪操作、杂项伪操作……
  - 伪指令：

    ADR伪指令、ADRL伪指令、LDR伪指令……

数据定义伪操作

.byte	单字节定义	.byte 0x12, ‘a’, 23
.short	定义2字节数据	.short 0x1234, 65535
.long /.word	定义4字节数据	.word 0x12345678
.quad	定义8字节	.quad 0x1234567812345678
.float	定义浮点数	.float 0f3.2
.string /.asciz /.ascii	定义字符串	.ascii “abcd\0”

汇编控制伪操作

.if .else .endif

——类似C语言里的条件编译，汇编控制伪操作用于控制汇编程序的执行流程。.if、.else、.endif伪指令可以嵌套使用。

.macro . mend

——类似C语言里的宏函数。.macro伪操作可以将一段代码定义为一个整体，称为宏指令，在程序中通过宏指令可以多次调用该段代码。宏操作可以使用一个或多个参数。

汇编控制伪操作

.macro .mend示例

```
MACRO                                  @宏定义
CALL $Function,$dat1,$dat2             @宏名称为CALL,带3 个参数
IMPORT $Function                       @声明外部子程序 宏开始
MOV R0,$dat1                           @设置子程序参数,R0=$dat1
MOV R1,$dat2
BL $Function                           @调用子程序 宏最后一句
MEND                                   @宏定义结束
CALL FADD1,#3,#2                       @宏调用，后面是三个参数
```

杂项伪操作

伪操作	语法	说明
.arm	.arm	定义以下代码使用ARM指令集编译
.thumb	.thumb	定义以下代码使用Thumb指令集编译
.section	.section expr	定义一个段。expr可以是.text .data .bss
.text	.text {subsection}	将定义符开始的代码编译到代码段
.data	.data {subsection}	将定义符开始的代码编译到数据段，初始化数据段
.bss	.bss {subsection}	将变量存放到.bss段，未初始化数据段

ADR伪指令

- 获得标签内存地址的伪指令
  - ADR{cond} register, label

```
Start
            BL      func                ; Branch to subroutine
stop        MOV     r0, #0x18           ; angel_SWIreason_ReportException
            LDR     r1, =0x20026        ; ADP_Stopped_ApplicationExit
            SWI     0x123456            ; ARM semihosting SWI
            LTORG                       ; Create a literal pool
func        ADR     r0, Start           ; => SUB r0, PC, #offset to Start
            ADR     r1, DataArea        ; => ADD r1, PC, #offset to DataArea
            ; ADR   r2, DataArea+4300   ; This would fail because the offset
                                        ; cannot be expressed by operand2
                                        ; of an ADD
            ADRL    r2, DataArea+4300   ; => ADD r2, PC, #offset1
                                        ;    ADD r2, r2, #offset2
            MOV     pc, lr              ; Return
DataArea    SPACE   8000                ; Starting at the current location,
                                        ; clears a 8000 byte area of memory
```

ADRL伪指令

- ADR的地址范围是+/-1020字节
- ADRL的地址范围是+/-64KB字节（没有字对齐），+/-256KB（字对齐）
  - The range of an ADRL pseudo-instruction is ±64KB for a non word-aligned address and ±256KB for a word-aligned address

```
func        ADR     r0, Start               ; => SUB r0, PC, #offset to Start
            ADR     r1, DataArea            ; => ADD r1, PC, #offset to DataArea
            ; ADR   r2, DataArea+4300       ; This would fail because the offset
                                            ; cannot be expressed by operand2
                                            ; of an ADD
            ADRL    r2, DataArea+4300       ; => ADD r2, PC, #offset1
                                            ;    ADD r2, r2, #offset2
            MOV     pc, lr                  ; Return
DataArea    SPACE   8000                    ; Starting at the current location,
```

多选题 1分

设置

哪些是ARM伪指令?

A ADR

B ADRL

C MOVLE

D STMFD

提交

南開大學 Nankai University

鲲鹏CPU示例程序

- 思考：同样的代码在x86平台是否能够运行，为什么？

主观题 10分

设置

思考：同样的华为鲲鹏处理器程序在x86平台是否能够运行，为什么？

作答

汇编语言与逆向技术

第7章 华为鲲鹏处理器汇编编程

王志
zwang@nankai.edu.cn

南开大学 网络空间安全学院

2024-2025学年