ARM Instructions

ARM Instructions

- Data
  - Base
  - Positive & Negative
  - Floating Point
- Processor Mode
- ARM Registers
  - Banked Registers
  - CISC & RISC
  - Thumb Instructions
- arm instructions

Data

Base

1. 通过进制表示, 以下面的前缀开头:
    1. `0b` Base-2.
    2. `0` or 空 Base-10.
    3. `0x` Base-16.
2. 整数进制转换:
    1. 十进制转OTHER:
        1. 整除法
    2. OTHER转十进制:
        1. 对应位置的次幂:

           $0b10110 = 2^4 + 2^2 + 2^1 = 22$
3. 浮点数进制转换:
    1. 十进制转OTHER:
        1.
    2. OTHER转十进制:
        1. $0b101.11 = \frac{0b10111}{0b100} = \frac{23}{4} = 5.75$
        2. $12.5 = \frac{125}{10} = \frac{0b1111101}{0b1010} = 0b1100.1$

Positive & Negative

Two's complement

```
int num;

-num = ~num + 1
```

> *Two's complement* 的运算比直接使用最高位表示符号好做. 直接相加即是答案. 并且靠最高位也依然能看出符号.

Floating Point

> 对于这个问题, 已经问过很多次, 但是都没有找到一个能让我满意的答案, 现在在建军的课上我觉得差不多找到了.

浮点数之所以叫做浮点数, 是因为还是有定点数 *fix point*. 他们都是用于表示小数*或者说是分数*.

1. 定点数:

   原理就是 *+101101.10010*, 整数部分与小数部分分别表示出来. 组合在一起.

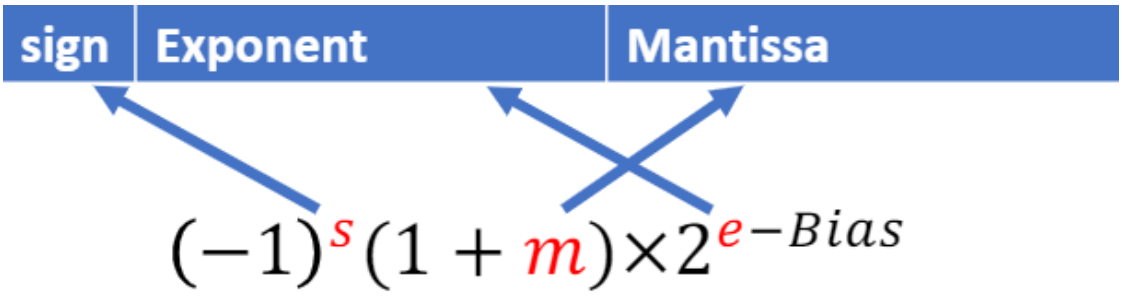
   

   使用的 *int* 与 *long* 就是没有 *fraction* 的形式而已. 负数也是使用 *Two's complement*

2. 浮点数:

   原理: 二进制版的科学计数法.

   之所以叫*浮点数*是因为相对于*定点数*来讲, 他的小数点在哪里是不确定的. 或者说, 不是依靠小数点来确定数值的.

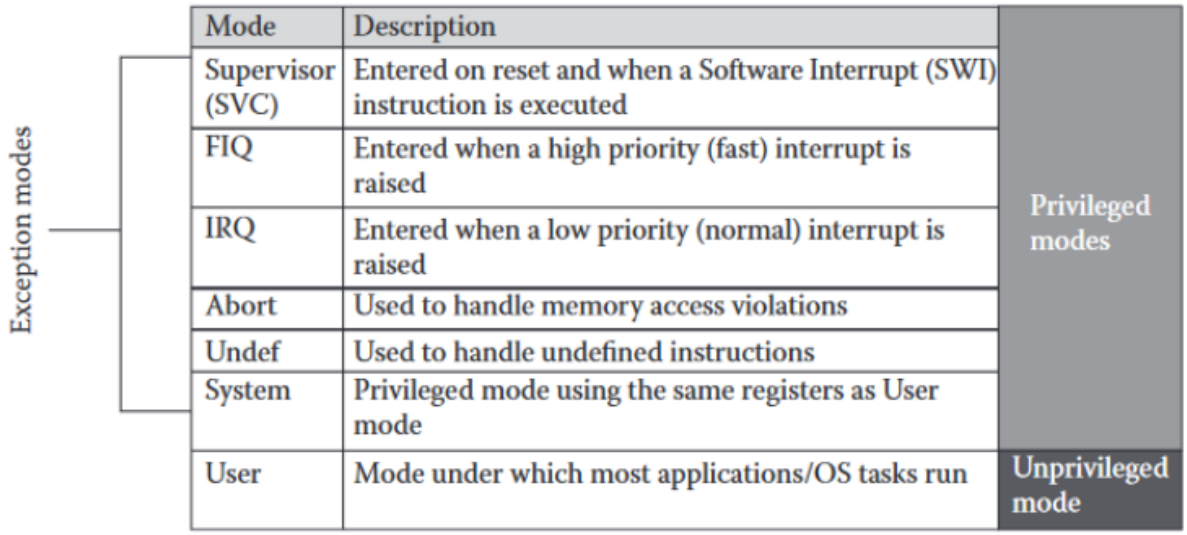
   

   1. Mantissa: 科学计数法的小数部分.
   2. Exponent: 科学计数法的指数部分.
   3. sign: 符号.
   4. Bias: 指数的偏移值, 因为如果没有偏移值的话, 这个数就太大了, 不能表示很小的部分. 加上偏移值之后, 幂的取值范围也是对半分的了.

在 32 位浮点数中，总共是8位的 $exponent$，$bias = 2^{8-1} - 1 = 127$，所以实际的指数区间是 $[-126, 127]$.

Processor Mode

7种进程模式.



	Mode	Description	
Exception modes	Supervisor (SVC)	Entered on reset and when a Software Interrupt (SWI) instruction is executed	Privileged modes
	FIQ	Entered when a high priority (fast) interrupt is raised	
	IRQ	Entered when a low priority (normal) interrupt is raised	
	Abort	Used to handle memory access violations	
	Undef	Used to handle undefined instructions	
	System	Privileged mode using the same registers as User mode	
	User	Mode under which most applications/OS tasks run	Unprivileged mode

一般情况下, 只有 *user* 和 *supervisor* 会被使用到, 在外部的中断下, 比如键盘敲击指令会引起其他模式.

ARM Registers

- 13 general-purpose registers R0-R12.
- R11: Frame Pointer (FP).
- R12: 1 Intra-Procedure-call scratch register (IP).
- R13: 1 Stack Pointer (SP).
- R14: 1 Link Register (LR).
- R15: 1 Program Counter (PC).
- 1 Current Program Status Register (CPSR).

Banked Registers

> 算是被存下来的 *register*. 在比较紧急的 *mode* 下才能使用, 正常的*mode*下是不会使用的.

User	IRQ
R0	R0
R1	R1
R2	R2
R3	R3
R4	R4
R5	R5
R6	R6
R7	R7
R8	R8
R9	R9
R10	R10
R11	R11
R12	R12
R13	R13_IRQ
R14	R14_IRQ
PC	PC
CPSR	CPSR
	SPSR_IRQ

这些 *register* 的效果是一样的, 只不过在不同的 *mode* 下会使用别的 *register* 以快速应对外部变化. 不需要清空原有寄存器里面的东西.

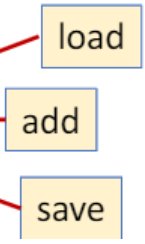
CISC & RISC

RISC CPU 只能处理在 *registers* 中的数据, 但是 *CISC* 可以直接处理在内存中的数据.

在 *RISC* 中, data必须先从内存中 load 进 register, 在处理过之后, 再 *store* 进内存.

C/Java Code	x86 assembly	arm7tdmi

x = x + 5;	addl $5, -4(%rbp)	ldr r3, [fp, #-8]
		add r3, r3, #5
		str r3, [fp, #-8]



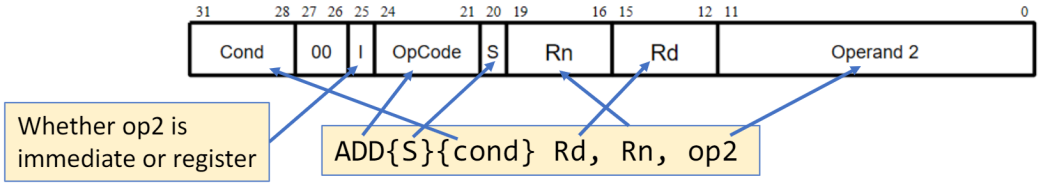
Thumb Instructions

> 从直觉上来讲, 是简化版的*arm instruction*, 目的是使用更少的内存与更少的 *registers*

7个寄存器与16位处理器.

arm instructions

Syntax 句式:



1. *cond:* is an optional condition code.
2. *Rd*: is the destination register. 是存 *result* 的 *register*.
3. *Rn*: is the register holding the first operand.
4. *Operand2*: is a flexible second operand.

> 在 {} 中的内容是可以省略的.

这是32位的 CPU, 指令也是32位长的. 64位就是64位长的指令了.

VALUE 值

在汇编里面, 这个叫做立即值 (immediate value)

用 `#` 放在数值前面, 就代表着这个数.

OPERATION 运算

Instructions	Operation	Meaning
`AND Rd, Rn, op2`	位运算*和*	Rd := Rn AND op2
`EOR Rd, Rn, op2`	位运算*异或*	Rd := Rn XOR op2
`ORR Rd, Rn, op2`	位运算 *或*	Rd := Rn OR op2
`BIC Rd, Rn, op2`	位运算 CLEAR	Rd := Rn AND NOT op2
`MVN Rd, Rn, op2`	位运算取反	Rd := NOT op2
`SUB Rd, Rn, op2`	减法	Rd := Rn - op2
`RSB Rd, Rn, op2`	反向相减	Rd := op2 - Rn
`ADD Rd, Rn, op2`	相加	Rd := Rn + op2
`ADC Rd, Rn, op2`	带进位相加	Rd := Rn + op2 + C
`SBC Rd, Rn, op2`	带进位相减	Rd := Rn - op2 + C -1
`RSC Rd, Rn, op2`	反向带进位相减	Rd := op2 - Rn + C -1
`MOV Rd, Rn, op2`	移动	Rd := op2
`MUL Rd, Rn, op2`	乘法	Rd := Rn * op2
`LSL Rd, Rm, Rs`	逻辑左移	直接左移。空位被清除。
`LSR Rd, Rm, Rs`	逻辑右移	直接右移。空位被清除。
`ASR Rd, Rm, Rs`	算数右移	寄存器内容被视为二进制补码整数。符号位被复制到空位中
`ROR Rd, Rm, Rs`	右旋	从寄存器的右端移出的位将旋转回左端。

MEMORY 与内存交互

Instructions	Operation	Meaning
`DCD`	在内存中声明一个 *WORD*	
`EQU`	在内存中声明一个常量	
`FILL`	在内存中声明一个**空的** *WORD*	
`LDR Rd, [Rn]`	从内存中读取	从 *Rn* 中得到的值当成指针, 把指针指向的内存里的值放进 *Rd*.
`STR Rd, [Rn]`	把值存在内存中	把 *Rd* 中的值存在 *Rn* 显示的内存地址中.
`MOV Rd, Rn`	把 *Rn* 里的东西直接放到 *Rd* 中.	
`ADR r3, label`	读取*label*	把label的内存地址放在r3中

> *[]* 是间接取地址的的符号, 也就是把 *Rn* 中存的值当成指针.

例子:

R6	R11	0x00004000
	0x00004000	*0xF97D5EC5*

LDR r6, [r11]

R6	R11	0x00004000
0xF97D5EC5	*0x00004000*	*0xF97D5EC5*

OFFSET 偏移

Instructions	Operation	Meaning
`LDR r6, [r11, #12]`	pre load	取出r11后面12bits的内存中的值
`LDR r6, [r11, #12]!`	pre load	先r11加上12, 之后取出r11内存中的值,
`LDR r6, [r11], #12`	post load	先取出r11内存中的值, 后r11加上12,

> 如果是 **`LDRB`** 就是以 bytes 为单位. 就不用以 4 为基准

LITTLE OR BIG ENDIAN 大小端

人类的读写方式是小端.

FLAGS

FLAG	Meaning	PS
Z	Zero	
N	Negative	
C	Carry	
V	Overflow	

- 后面有 *s* 的表示这个 *operation* 需要引起 *flag* 的变化, 不然一般情况下不会有 *flag* 的变化.
- *flag* 会保持到下一个 *s-suffix* 执行.
- 另一些指令例如 `CMP` 也会 *set flag*.
  - 用 rd 与 rn 相减;
  - 负数: N -> 1;
  - 零: Z -> 1;
  - 溢出: V -> 1;
  - 进位: C -> 1;

例子:

1. `0xffffffff - 0x00000001 -> N and C` 在汇编中没有减法, 都是把数做 *2's implement* 之后相加的, 所以相当于是 `-1 + (-1) = 0xffffffff + 0xffffffff`, 所以既是负数又是进位.

CONDITIONAL BEANCH 条件支路

Instructions	Operation	Meaning	PS
B{cond} label	导向 label 支路	无条件直接跳转	
BNE	Branch not equal	当 flag Z == 0 的之后会跳转	
BEQ	Branch equal	当 flag Z == 1 的之后会跳转	
BL	Branch with Link	把跳转点位置存在 *R14 (LR)* 中	可通过在子程序的尾部添加 `mov pc, lr` 返回
BX	Branch and Exchange		

Branch-mnemonics: 是每一个branch的label, 当条件符合的时候就会自动跳转到这个label上来.

下面是 *conditions*

Code	Meaning (for CMP or SUBS)	Flags Tested
eq	eqaul	Z==1
ne	Not equal	Z==0
cs or hs	Unsigned higher or same (or carry set).	C==1
cc or lo	Unsigned lower (or carry clear)	C==0
mi	Negative (minus)	N==1
pl	Positive or zero. (plus)	N==0
vs	Signed overflow. (V set)	V==1
vc	No signed overflow. (V clear)	V==0
hi	Unsigned higher	(C==1) && (Z==0)
ls	Unsigned lower or same	(C==0) || (Z==1)
ge	Signed greater than or equal	N==V
lt	Signed less than	N!=V
gt	Signed greater than	(Z==0) && (N==V)
le	Signed less than or equal	(Z==1) || (N!=V)
al (or omitted)	Always executed	None tested

SUBROUTINE 子程序

`r14` 可以被写为 `lr` *link register*, 是一种简便的记忆方法, 是存放跳转地址 *return address* 的.

为了从子程序中跳出, 或者说从子程序中到主程序中, 用 `MOV pc, lr`, 全拼就是 `MOV r15, r14`

但是要是从一个子程序跳转到另一个子程序的时候就会有问题.

NESTING 嵌套

子程序内还有另一个子程序.

这种时候就不能只用 `lr`, 因为 `lr` 里面只能放一个. 所以就要用到 *栈*.

栈顶指针就是指向栈顶的.

STACK 栈

Ascending and descending stacks 升序栈与降序栈:

1. 栈顶的位置大 -> 升序 ascending
2. 栈顶的位置小 -> 降序 descenting

在 *ARM7* 中两种都可以, 但是不要从一种转化成另一种.

栈顶指针叫做 `r13`, 或者是小名 `sp` *stack pointer*

```
; To push the link register value onto a full descending stack
; 把 link register 记录的位置放到栈顶
STMFD sp!, {lr}
; To pop a value from a full descending stack back into the link register
; 把栈顶记录的地址放到 link register 中
LDMFD sp!, {lr}
```

在一般情况下, 子程序的第一个指令就是:

```
; 把返回主程序的地址放到栈顶. 之后就能调用别的程序了.
STMFD sp!, {lr}
```

从这个子程序中*return*

```
; LDMFD sp!, {lr}
; MOV pc, lr
LDMFD sp!, {pc}
```