前言

> 请先进行x86指令集的学习再来学习本文的内容

寄存器

arm32: 17个常见寄存器

1. r11: bp
2. r13: sp
3. r14: lr 这个很特别，保存了函数的返回地址
4. pc

(aarch64)arm64: 36个常见寄存器

1. x29: bp
2. x30: lr
3. x31: sp
4. x32: pc

汇编语言

格式

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

常量

> .equ name,var; 注意逗号不要省略
> 这里指的是宏常量，而不是c语言中的const常量
> 当然，如果非要声明c语言中的常量，可以将其写在 .rodata段 中.

变量声明

> varName: varType value

```
.section .data
varStr:
    .string "helloworld"          @ 两个字符串
varInt:
    .word 123,456               @ 或者 .int
varInt64:
    .quad 1234

.section .bss
name:
    .skip, 50               @ char name[50]
intarray:
    .skip, 20               @ int intarray[5], 转换成byte
uninit:
    .space,200

intarray:
    .rept 4
    .word 0   @ 这里一定要有一个值 0 或者其他值亦可以, 但是没区别,都会变成 \x00
    .endr                 @ 第二种声明int数组的方法
```

> 还有其他的伪指令: .float, .double等等
> 上面的指令是以arm32位例子, arm64类似.

函数

调用约定: APCS

1. arm32

- 前四个参数放 r0, r1, r2, r3
- 之后的从右到左参数放栈里
- 返回值放 r0

2. aarch64

- 前8个参数放 x0--x7
- 之后的放栈里
- 返回值放 x0

函数调用指令

> bl == mov lr, pc; b target_addr
> b 类似于x86中的jmp,是无条件跳转指令

栈帧

arm32

```
@=====  第一种  有调用子函数=====
STMFD    SP!,  {R11,LR}   @==  push{r11,lr}==push  lr;push  r11
ADD      R11,  SP,  #4    @  bp  =  sp  +  4

@        局部变量
@----------------------------
@        r11(old_bp)
@----------------------------cur_bp
@        lr
@----------------------------

SUB      SP,  R11,  #4    @  sp  =  bp  -  4
LDMFD    SP!,  {R11,PC}   @==  pop{r11,pc}=pop  r11;pop  pc

@=====  第二种  没有调用子函数=====
str  r11,[sp,#-4]!   @  push  {r11}
add  r11,sp,#0

@----------------------------cur_bp
@      r11(old_bp)
@----------------------------

ldr  r11,[sp],#4    @  pop  {r11}
bx  lr
```

一个完整的arm32例子

> 32位arm程序的栈帧和x86的有些不同

```
.section  .text
.global  test1
```

```
test1:                          @ 这个函数有子函数
                                @ 所以需要将lr保存到栈里面,并且通过pop pc来返回
    stmfd sp!,{r11,lr}
    add r11,sp,#4           @ r11=sp+4

    sub sp,sp,#20
    bl test2

    sub sp,r11,#4           @ sp=r11-4
    ldmfd sp!,{r11,pc}


.global test2           @ 由于没有子函数,所以lr没有被保存到栈里面
test2:
    str r11,[sp,#-4]!   @ sp=sp-4;str r11,[sp]
    add r11,sp,#0       @ bp=sp

    sub sp,sp,#0x20     @ sp=sp-0x20

    sub sp,r11,#0       @ sp=bp
    ldr r11,[sp],#4     @ pop bp 这两句相当于x86的leave
    bx lr               @ ret


.global _start
_start:
    stmfd sp!,{r11,lr}
    add r11,sp,#4

    sub sp,sp,#0x20             @ 创建栈帧
    bl test1

    sub sp,r11,#4               @ sp=bp-4
    ldmfd sp!,{r11,pc}          @ pop bp;ret
```

aarch64(arm64)

> aarch64 一进函数作用域，第一条指令便创建了局部变量所需要的全部栈帧，并且局部变量所在位置比bp所指向的位置要高

```
@===== 第一种 =====
stp x29,x30,[sp, #-0x30]! @ push {x29,x30} == push x30; push x29
mov x29,sp

 - List item

@---------------------------------------cur_bp,sp
@           x29(old_bp)
@-------------------------------------
@           x30(lr)
@-------------------------------------
@           局部变量(局部变量在这里,这是和x86很不同的地方)
@-------------------------------------

ldp x29,x30,[sp],#0x30    @ pop {x29,x30} == pop x29; pop x30; sp=sp+16
ret

@===== 第二种 =====
sub sp,sp,0x10      @ sub的大小取决于栈帧需要多大; 没有保存fp, 因为可以通过sp寻址

add sp,sp,0x10
ret             @ bx lr
```

一个完整的aarch64例子

```
.global  .text

.global  test1
test1:
     stp  x29,x30,[sp,#-0x10]!           @  没有局部变量
    mov  x29,sp

    bl  test2

     ldp  x29,x30,[sp],#0x10
     ret

.global  test2
test2:
    sub  sp,sp,0x30

    add  sp,sp,0x30
     ret

.global  _start
_start:
     stp  x29,x30,[sp,#-0x30]!            @  push  x29(bp);push  x30(lr)  ,不一定是#0x30,  由局部所需的空间决定
    mov  x29,sp                          @  x29(bp)=sp

     ldp  x29,x30,[sp],#0x30
     ret                                 @  bx  x30(lr)
```

细节

> 1. 函数的返回方式有两种，取决于进入函数后有没有将 lr 保存到栈里
> 2. 而且在64位的aarch架构中，x29(bp)不一定会保存在栈里面(这似乎取决于函数有没有子函数)，因为子函数可以通过sp寻址，这也算是和x64有区别的地方
> 3. 不管是arm32还是arm64，他们的栈帧变化都和x86有区别
> 4. 在aarch64中，由于其栈帧的特别之处，对栈溢出的长度要求要更高直至可以覆盖上一个函数的返回地址

系统调用

1. arm32

- 系统调用号: r7
- 参数: r0 -- r6
- swi 0x0

2. aarch64(arm64)

- 系统调用号: x8
- 参数: x0 -- x5
- svc #0

> ubuntu下可以使用 man syscall 查看

arm立即数

> arm或者aarch64(arm64)是定长指令集，每一条指令都是32bits，所以在指令中不可能存在32bits的立即数，但是，在某些情况下，我们可能需要获得一个32bits或者64bits的立即数

获得地址

1. arm32

```
adrl  <reg>,  label      @  可以将label的地址写入寄存器
```

2. aarch64(arm64)

```
adrp  <reg>,  label      @  可以将label的地址写进寄存器
```

> ldr 也可以加载地址，但是有范围限制,所以ldr一般使用寄存器寻址
> 实际上adrl和adrp也是有范围限制的
> 可以使用 ldr <reg>, =<imm> 来获得立即数

获得其他立即数

> 通过间接获得
> 将一个数放到寄存器，然后不断加减乘除即可得到想要的立即数

arm/aarch64 独特之处

arm与x86(包括x86,x86-64)不同的小细节

1. arm中一个word是32bit的
2. 虽然arm32中也有push,pop指令，但似乎保存和pc,lr等环境信息使用的是stm,ldm等指令
3. aarch64(arm64)没有push，pop，stm,ldm，取而代之的是 ldp 和 stp
4. arm32和aarch64(arm64)的栈帧变化和x86不同，具体请看 函数调用-栈帧
5. aarch64(arm64)不一定会把x29(即bp寄存器)保存到栈里，这取决于函数有没有子函数

arm小细节

1. 不是所有的函数都会把 lr 保存到栈里面的，只有在该函数有子函数(或者需要处理异常的时候)才会也才需要将 lr 保存到栈里面
2. arm的指令似乎都有一个共同的特点：左边的操作数是寄存器或者把寄存器当指针用；右边的操作数是内存单元或者寄存器
3. 由于指令长度的限制，如果要对内存进行读写操作，一定要将其地址写进寄存器，然后通过寄存器寻址