Лекция 5
Программирование на ассемблере ARM

Отображение Си в ARM

- В качестве языка высокого уровня выберем язык Си
- Будем рассматривать отображение основных конструкций Си в конструкции ассемблера
- Си удобен для этого (более-менее прозрачное отображение)

Signed vs unsigned char

- На ARM по умолчанию тип char unsigned, на x86 тип char signed
- Регистр процессора — 32-битный, при загрузке 8-битного значения необходимо его преобразование в 32-битное
- Для беззнаковых значений — старшие 24 разряда заполняются 0 (инструкция ldrb)
- Для знаковых значений — старшие 24 разряда заполняются битом знака (инструкция ldrsb)

Пример

```
unsigned char c1;
signed char c2;
int i3;

i3 = c1;
i3 = c2;
```

```
.text
ldr    r0, c1ptr
ldrb   r0, [r0]
ldr    r1, i3ptr
str    r0, [r1]
ldr    r0, c2ptr
ldrsb    r0, [r0]
str    r0, [r1]
// …
c1ptr:   .int   c1
c2ptr:   .int   c2
i3ptr:    .int   i3
    .data
c1:   .byte    0
c2:   .byte    0
i3:    .int 0
```

Signed/unsigned в сравнениях

```
int r0, r1;

                                    cmp     r0, r1
if (r0 < r1) r0 = r1;               movlt   r0, r1

unsigned r0, r1;

if (r0 < r1) r0 = r1;               cmp     r0, r1
                                    movlo   r0, r1
```

[unsigned] long long

- Размер типа [unsigned] long long — 64 бита — два машинных слова
- Для чтения из памяти или записи в память нужны две операции работы со смежными словами в памяти или инструкции ldm/stm
- Для умножения и деления могут вызываться специальные подпрограммы runtime-библиотеки компилятора (libgcc)

Long long сложение/вычитание

- Выполняется за две операции, используем флаг 'C' после первой операции
- Пусть r0, r1 — первый операнд, r2, r3 — второй операнд, r4, r5 — результат
- Сложение:
  ```
  adds    r4, r0, r2
  adc     r5, r1, r3
  ```
- Вычитание:
  ```
  subs    r4, r0, r2
  sbc     r5, r1, r3
  ```
- Сравнение (игнорируем результат в r4, r5)
  ```
  subs    r4, r0, r2
  sbcs    r5, r1, r3
  ```

Загрузки long long на регистр

- Пусть адрес находится в r2, загружаем в { r0, r1}
  ldmia r2, { r0, r1}
- Для сохранения в память
  stmia r2, { r0, r1}

Стековые операции и память

- Stmfd == stmdb
- Stmfa == stmia
- Stmed == stmda
- Stmea == stmib

Условный оператор if

```
if (r0 >= r1) {
    …
}
```

```
cmp     r0, r1
blt     doneif

…

doneif:
```

Оператор if

```
if (r0 > 5) {
    ...
} else {
    ...
}
```

```
    cmp     r0, #5
    ble     elselab
    ...
    b       donelab
elselab:
    ...
donelab:
```

Логические операции

- Логические операции && и || представляются в виде последовательности вложенных if
- Например:
  if (i < n && a[i] != 0)
- Эквивалентно следующей конструкции:

```
if (i < n) {
    if (a[i] != 0) {
    }
}
```

Цикл while

```
while (r0 != r1) {
    // …
}
```

```
cmp     r0, r1
beq     noloop
loop:
…
cmp     r0, r1
bne     loop
noloop:
```

switch

```
switch (expr) {
    case VAL1: CODE1;
        break;
    case VAL2: CODE2;
        break;
    default:
        break;
}
```

- В зависимости от множества значений VAL компилятор генерирует несколько вариантов:
  - Линейный поиск со сравнением
  - Поиск по дереву со сравнением
  - Переход по таблице

switch

- Если множество меток сгруппировано плотно (например, последовательно), используется переход по таблице
- Пусть множество значений: { 0, 1, 2, 3 }
- Адреса фрагментов, обрабатывающих варианты, помещаются в таблицу TABLE, например: { LAB0, LAB1, LAB2, LAB3 }, LAB — метки
- Выполняется переход по таблице

```
add  r1, pc, #TAB - . - 8
ldr        pc, [r1, r0, lsl #2]
```

Работа со стеком

- Локальные переменные размещаются в стеке или на регистрах
- Если берется адрес переменной, то только на стеке
- Память под локальные переменные выделяется при входе в подпрограмму и освобождается перед выходом из подпрограмму

Пример локальной переменной

```
void f()
{
    int x;
    long long y;
    scanf(«%d», &x);
    scanf(«%lld», &y);
    ...
}
```

```
f:
    stmfd   sp!, { r4, lr }
    sub  sp, sp, #16
    // r0 = &x
    mov r0, sp
    // r0 = x
    ldr    r0, [sp]
    // r0 = &y
    add  r0, sp, #8
    // { r0, r1 } = y
    ldr    r0, [sp, #8]
    ldr    r1, [sp, #12]
    // эпилог
    add  sp, sp, #16
    ldmfd   sp!, { r4, pc }
```

Стековые фреймы

- Часто требуется, чтобы во время работы программы возможно было проследить динамическую цепочку вызовов от самого вложенного до вызова main
- Например, при выполнении программы была сделана цепочка вызовов: main → f1 → f2 → f2 → f2 → f3
- При исполении подпрограммы f3 требуется найти, какие подпрограммы были вызваны выше по стеку, то есть f2, f2, f2, f1, main

Стековые фреймы

- Применение: обработка исключений в Си++ - требуется пройти по динамической цепочке вызовов, чтобы понять, обрабатывается ли данное исключение и где обрабатывается
- Применение: отладка (печать трассы стека)
- Применение: локальные массивы переменного размера (размер локальных переменных заранее не известен)

Стековые фреймы

- Простейший пролог не позволяет прослеживать цепочку вызовов или размещать на стеке массивы переменного размера
  - Неизвестно, сколько стека занято локальными переменными и сохраненными регистрами в каждой функции
- Данные в стеке должны быть специальным образом организованы!
- Структура данных на стеке для обеспечения возможности прослеживания цепочки вызовов — стековый фрейм

Пролог функции

- Обозначим r12 как ip (invocation pointer), r11 как fp (frame pointer)
- Стандартный пролог:

```
Mov     ip, sp          // сохраняем sp
stmfd   sp!, { fp, ip, lr, pc } // r11, r12, r14, r15
sub     fp, ip, #4
```

R15 (pc) — это адрес точки входа в подпрограмму + #12	← FP, для обращения: [fp]
R14 (lr)	[fp, #-4]
R12 (ip) — original SP	[fp, #-8]
R11 (fp)	← SP, для обращения [fp, #-12]

Пролог функции

- Обозначим r12 как ip (invocation pointer), r11 как fp (frame pointer)
- Стандартный пролог:

```
Mov     ip, sp        // сохраняем sp
stmfd   sp!, { fp, ip, lr, pc } // r11, r12, r14, r15
sub     fp, ip, #4
stmfd   sp!, { r4, r5, r6, r7 } // сохраняем регистры
sub     sp, sp, #16 // выделяем область под лок.
    //переменные
```

Пролог функции

Для нашего примера:

- [fp, #-28], [fp, #-24], [fp, #-20], [fp, #-16] — сохраненные регистры r4, r5, r6, r7
- [fp, #-44], [fp, #-40], [fp, #-36], [fp, #-32] — локальные переменные
- Работа с локальными переменными ведется относительно регистра fp

Стандартный эпилог

Для нашего примера:

```
// устанавливаем sp на начало области сохраненных регистров (если значение sp неизвестно)
ldr     ip, [fp, #-8]
sub     sp, ip, #32
// восстанавливаем регистры
ldmfd   sp!, { r4, r5, r6, r7 }
// восстанавливаем fp, sp, pc
ldmfd   sp, { fp, sp, pc}
```

Стандартный эпилог

Для нашего примера:

```
// если sp не менялось, просто уничтожаем
// локальные переменные
add      sp, sp, #16
// восстанавливаем регистры
ldmfd   sp!, { r4, r5, r6, r7 }
// восстанавливаем fp, sp, lr
ldmfd   sp, { fp, sp, pc}
```

Навигация по фреймам

- В текущей подпрограмме [fp, #-8] хранит предыдущее значение fp
  ```
  ldr    r0, [fp, #-12] // загружаем его в r0
  ldr    r1, [r0]       // в r1 сохраненный pc
  sub    r1, r1, #12    // в r1 — адрес начала подпрограммы
  ```
- Можем перейти еще на уровень вверх
  ```
  ldr    r0, [r0, #-12]// загружаем fp предыдущего уровня
  ```
- Еще на уровень вверх
  ```
  ldr    r0, [r0, #-12]
  ```
- Пока значение в r0 не станет 0