Лекция 6
Программирование на ассемблере ARM

Позиционная зависимость

```
        .text
        ldr     r0, varadr
        ldr     r0, [r0]
        // …
varadr: .int var
        .data
var:    .int 0
```

- Ячейка 'varadr' содержит адрес в памяти, по которому размещается ячейка var

Позиционная зависимость

```
        add    r0, pc, #table - . - 8
        ldr    pc, [r0, r1, lsl #2]
table: .int    L0, L1, L2, L3
L0:
L1:
L2:
L3:
```

- Таблица 'table' содержит адреса точек в программе, на которые будет выполнен переход

Позиционная зависимость

- При компоновке программы одним из параметров является базовый адрес
  - Одноименные секции программы из разных объектных файлов сливаются в одну
  - Все секции программы размещаются в некотором порядке последовательно
  - Все секции программы (вся программа) рассматривается как одно целое и размещается, начиная с базового адреса

Позиционная зависимость

- Базовый адрес зависит от платформы:
  - Linux ARM: 0x10000
  - Linux x86: 0x8048000
  - Linux x64: 0x400000
- Может меняться произвольным образом на одной платформе
  -Wl,-Ttext-segment=ADDR
- Исполняемые файлы будут различаться из-за разных адресов в памяти
- Исполняемый файл пригоден только для загрузки в память с определенного адреса

Позиционная независимость

```
add    r0, pc, #table - . - 8
bne    label   // кодируется смещением от pc
bl     sub     // смещение отн. pc
ldr    r0, varaddr // смещ. отн. Pc
ldr    r0, [pc, #varaddr - . - 8] // то же самое
```

- Программа рассматривается как единое целое, только базовый адрес может меняться
- Бинарное представление этих инструкций не меняется

Статическая компоновка

- Компоновщик собирает в исполняемый файл все используемые подпрограммы
  - Строится список зависимостей (начиная с подпрограмм в основных объектных файлах, подбираются необходимые модули из статических библиотек, пока все зависимости не будут разрешены)
  - Собирается единый исполняемый файл
  - Все подпрограммы, в том числе библиотечные, настраиваются на работу с базового адреса данного исполняемого файла
- Одна и та же подпрограмма (например printf) в разных статически скомпонованных файлах может находиться по разным адресам

Статическая компоновка

- Статическая компоновка — единственный вариант в embedded системах (программа работает без операционной системы вообще)
- Статически скомпонованная программа не зависит от версий библиотек, установленных в данной конкретной системе
- Должна работать везде, если не изменился интерфейс системных вызовов

Динамическая компоновка

- Исполняемый файл содержит таблицы импортов, в которых указывается имя подпрограммы и имя библиотеки
- При загрузке программы на выполнение загрузчик (напр. /lib/ld-linux.so.2) подгружает в адресное пространство требуемые библиотеки и обрабатывает импорты

Динамическая компоновка

- Недостатки
  - Для запуска программы на выполнение требуется поддержка операционной системы и сложная программа-загрузчик
  - Программа запускается медленнее, чем при статической компоновке
  - Программа требует больше виртуального адресного пространства — динамическая библиотека подгружается в адресное пространство целиком, а не только то, что требуется

Динамическая компоновка

- Достоинства:
  - Уменьшение размера исполняемых файлов
  - Возможность обновления библиотеки без перекомпиляции изменяемых файлов (например, для устранения уязвимостей)
  - Возможность нескольким процессам использовать одну и ту же копию динамической библиотеки в памяти (то есть виртуальные адреса соответствующие библиотеке отображаются на одни и те же физические страницы ОЗУ)

Динамическая компоновка

- Практически невозможно добиться, чтобы в разных программах одна и та же динамическая библиотека всегда отображалась по одним и тем же адресам
  - Список библиотек заранее неизвестен, их много — неизбежны конфликты
  - Для повышения защищенности системы применяется рандомизация размещения библиотек (ASLR)

Динамическая компоновка

- Динамическая библиотека в разных процессах будет размещаться по разным базовым адресам
- Варианты реализации:
  - Fixup tables (Win32 DLL)
  - Position-Independent Code (ELF SO)

Позиционно-независимый код

- Позиционно-независимый код (position-independent code, PIC) — секция .text (кода) и другие секции, которые отображаются в режиме «только чтение» не зависят от базового адреса
- При необходимости адреса, требующие настройки, помещаются в секцию Global Offset Table (GOT)

```
gcc -fPIC prog.c
```

Global Offset Table

- Секция .got (Global offset table) содержит адреса глобальных переменных
- Секция .got модифицируется загрузчиком при загрузке программы на выполнение
- Секция .got доступна на модификацию и находится рядом с .data
- В коде используется смещение к адресу переменной относительно начала .got

Глобальные переменные

- На ARM обращение к обычной глобальной переменной «дорогое» (две инструкции вместо одной и одна вспомогательная ячейка памяти)
- В PIC-коде обращение к глобальной переменной еще дороже — 5 инструкций и две вспомогательных ячеек памяти
- Глобальные переменные — плохо!

Вызов внешних подпрограмм

- При динамической компоновке необходимо привязать внешние подпрограммы к их фактическим адресам в разделяемых библиотеках
- Lazy Bindind — привязка адреса при первом вызове
- Внешние подпрограммы вызываются не напрямую, а через PLT (procedure linkage table)

PLT

- Каждая внешняя подпрограмма имеет соответствующую запись в GOT, в которой вначале хранится адрес подпрограммы связывания
- При первом вызове вызывается подпрограмма связывания, которая заменяет запись в GOT на действительный адрес подпрограммы в разделяемой библиотеке
- Последующие вызовы переходят с помощью GOT напрямую по адресу в разделяемой библиотеке

Кодирование инструкций

- В исполняемом коде каждая поддерживаемая процессором инструкция должна быть закодирована в двоичное представление
- Декодирование должно быть однозначным
- Пример: кодирование x86 — переменная длина инструкции (от 1 до 16 байт)

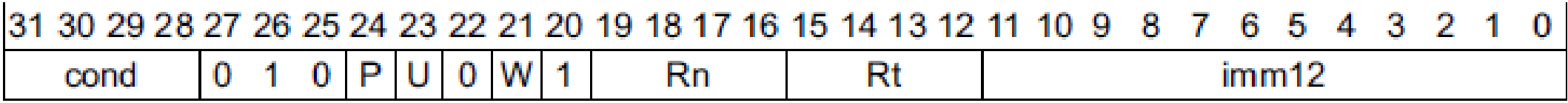
Кодирование ARM

- Процессоры ARM поддерживают два режима кодирования: ARM и Thumb
  - ARM — любая инструкция — 32 бита
  - Thumb — длина закодированной инструкции 16 или 32 бита, поддерживается не все инструкции
    - Примерно 25% выше плотность кода
    - Медленнее
- Между режимами ARM и Thumb можно переключаться во время работы программы
  blx subroutine
- Если младший бит == 1, переключение в thumb

LDR

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>, [<Rn>{, #+/-<imm12>}]
LDR<c> <Rt>, [<Rn>], #+/-<imm12>
LDR<c> <Rt>, [<Rn>, #+/-<imm12>]!
```

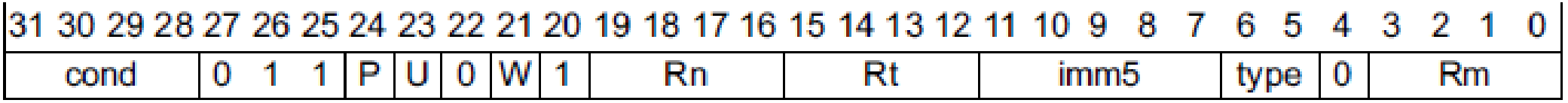


31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	0	W	1	Rn	Rt	imm12

- Cond — condition code
- Imm12 — беззнаковое 12-битное смещение
- Если U == 0, то смещение меняет знак
- Режимы адресации
  - P == 1, W == 0 : [Rn, +/- Imm12]
  - P == 1, W == 1 : [Rn, +/- Imm12]!
  - P == 0, W == 0 : [Rn], +/- Imm12]

LDR

```
LDR<c> <Rt>, [<Rn>,+/-<Rm>{, <shift>}]{!}
LDR<c> <Rt>, [<Rn>],+/-<Rm>{, <shift>}
```



31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	P	U	0	W	1	Rn	Rt	imm5	type	0	Rm

- Type — тип сдвига
- Imm5 — счетчик сдвига

ADD

```
ADD{S}<c> <Rd>, <Rn>, #<const>
```

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0	0	1	0	1	0	0	S	Rn	Rd	imm12

```
ADD{S}<c> <Rd>, <Rn>, <Rm>{, <shift>}
```

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0	0	0	0	1	0	0	S	Rn	Rd	imm5	type	0	Rm

- S — установить биты флагов
- Imm5 — счетчик сдвигов
- Type — тип сдвига