Есть ли у вас вопросы?

Краткое содержание предыдущей серии

- Что такое CISC и RISC?
- В чем их основные отличия?
- Что это такое и что здесь что?

```
0x08000330 6011   STR r1,[r2,#0x00]
```

- Что такое адресация?
- Какие способы адресации вы помните?

Краткое содержание этой серии

- Типизация в языках программирования
- Команды загрузки и сохранения
- Подробнее о длинных и коротких командах Thumb-2
- Чудеса языка C
- О стиле написания кода

Типизация в языках программирования

О чем речь?

Типизация – способ задания *типа* объекта.

А что такое тип?

Простым языком тип – это смысл, который несет объект.

Тип ограничивает область допустимых значений и допустимых действий над объектом.

Более формально см. «теория типов»

Виды типизации

По наличию типизации языки программирования бывают:

- Типизированные (C, C++, Java, Python...) – работа с «объектами»
- Бестиповые (Ассемблер, Brainfuck..) – работа с памятью напрямую

Типизация бывает:

- Статическая и динамическая.
- Сильная (строгая) и слабая (нестрогая).
- Явная и неявная (и утиная).

Все это разные свойства. Т.е. типизация в одном языке может быть статическая, строгая и явная, например.
Подробнее http://habrahabr.ru/post/161205/

По времени определения типа

Название	Статическая	Динамическая
Когда определяется тип?	Во время компиляции	Во время выполнения
Языки	C, Java, C++	Javascript, Python, Ruby

Но в языках со статической типизацией можно руками сделать динамическую.
Иногда, можно и наоборот.

По силе (строгости)

Название	Сильная (строгая)	Слабая (нестрогая)
Как происходит преобразование типов?	По приказу программиста – явное приведение типа (explicit type cast)	Автоматически – неявное приведение типа (impicit type conversion, coercion)
Языки	Java, Python, Haskell	C, C++, Javascript, PHP

Обычно автоматически приводятся не любые типы к любым типам. Например, приведение числа к массиву весьма неоднозначно (см. “javascript wat”)

В строго-типизированных языках иногда разрешается «расширение» (promotion) – например, short автоматически приводится к int.

По «явности»

Название	Явная	Неявная	Неявная утиная (duck typing)
Нужно ли задавать тип при объявлении переменной?	Нужно: int a = 5;	Не нужно: var a = 5;	Не нужно, лишь бы крякало и летало
Языки	Java, C, Pascal	Javascript, Python, Lua	Python, шаблоны в C++

В языке с явной типизацией бывает «неявная типизация по выбору»
И наоборот тоже бывает.

Утиный тест: если что-то выглядит, как утка, крякает как утка и летает, как утка – это утка

Бестиповой ассемблер – это как?

- Никаких переменных нет, есть только память
- Но на память можно «смотреть» по-разному (как на int или как на char)
- В RISC «смотреть» в память можно только командами загрузки и сохранения

Memory Map в STM32

- Процессор 32-битный, адресное пространство тоже 32-битное.
- Значит, адреса в памяти меняются от 0 до 2^{32}-1
- И максимальный адресуемый диапазон памяти – 4 Гб.
- Но в контроллере всего лишь 128 Кб Flash-памяти и 20 Кб ОЗУ.

Memory Map в STM32 (упрощенная)

... 0xE000 0000	Периферийные регистры ядра Cortex M3
... 0x4000 0000	Периферийные регистры
... ... 0x2000 0000	ОЗУ
... ...	
0x1FFF F7FF ... 0x1FFF F000	Системное ПЗУ
... ...	
0x0801 FFFF ... 0x0800 0000	Flash-память
..... 0x0000 0000	Сюда отображается Системное ПЗУ или Flash-память (при обычной работе - Flash)

- Код выполняется прямо из Flash-памяти
- Переменные хранятся в ОЗУ

Команды загрузки и сохранения

- Команда загрузки – load – LDR – загружает значение из памяти в регистр

```
LDR r0,[pc,#28]
```

- Команда сохранения – store – STR – сохраняет значение из регистра в память

```
STR r1,[r0,#0x0C]
```

И команды работы со стеком

- PUSH – поместить значение регистров в стек

```
PUSH  {r4-r6,lr}
```

- POP – вынуть значения из стека и положить в регистры

```
POP   {r4,pc}
```

Стек находится в той же оперативной памяти.

Эти команды эквивалентны STM и LDM от Stack Pointer’a

Где «тип» же в командах?

«Как мы смотрим» на память задается с помощью постфиксов:

- LDR – загрузить слово (word, 4 байта)
- LDR**H** – загрузить полуслово (halfword, 2 байта)
- LDR**B** – загрузить байт (byte)
- LDR**S**B – загрузить знаковый байт (signed byte)
- LDR**SH** – загрузить знаковое полуслово (signed halfword)
- LDR**D** – загрузить двойное слово (double word, 8 байт)
- LDR**M** – загрузить много байт (multiple)

Для STR аналогично.

Есть еще постфиксы, которые к типу не относятся.

У POP и PUSH «типовых» постфиксов нет.

Типы C и ассемблер

- Между целочисленными типами и командами соотношение очевидное

- А как же float и double?

  float – 4 байта, специализированных регистров для него нет – поэтому просто LDR

  double – 8 байт – просто LDRD

Длинные и короткие команды

Как процессор узнает, длинную команду нужно выполнить или короткую?

- Раньше (в ARMv5) было просто два режима
- В ARMv7 хитрее – в коде команды написано, длинная она или нет

Длинные и короткие команды

- Процессор считывает 16 бит по адресу, указанному в PC.

- Если биты с 15 по 11 это:
  - 11101 или
  - 11110 или
  - 11111 – то команда длинная. И нужно считать еще 16 бит.

- Иначе: команда короткая

Поэтому коды длинных команд часто начинаются на F

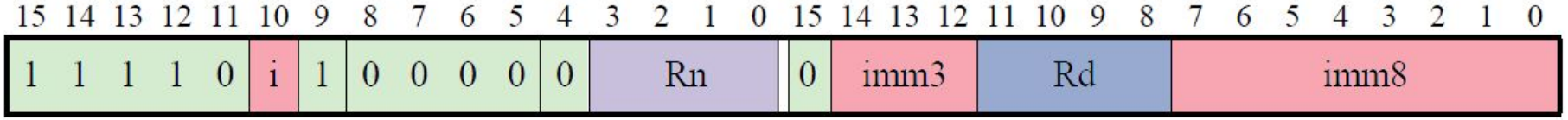
Структура короткой команды на примере `MOVS r0,#0x05`

MOV (1) (Move) moves a large immediate value to a register.

- 0x2005 = 0010 0000 0000 0101
- Пять старших бит (opcode) показывают что это, собственно, команда mov
- Биты 10, 9 и 8 задают номер регистра
- Биты с 7 по 0 задают непосредственный операнд

Это кодирование T2

Структура длинной команды (ADDW)



ADDW Rn, Rd, Imm -> Rd = Rn + Imm12

- Зеленое – opcode (в частности показывает, что команда длинная)
- Rn – регистр-слагаемое
- Rd – регистр-приемник результата
- Imm12 – число, собираемое из i:imm3:imm8 – 12 бит

Это кодирование T4

Пример ADDW

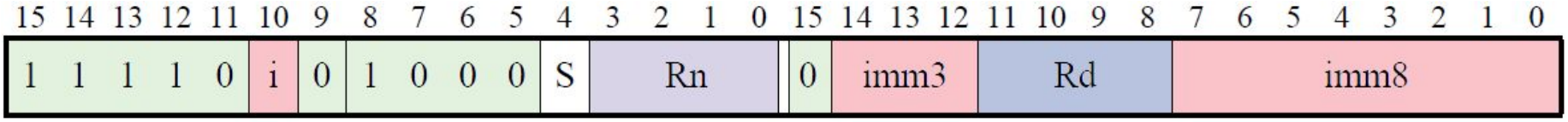
```
F600401A  ADDW      r0,r0,#0xC1A
```

C1A = 1100 0001 1010

F600401A =

15	11	7	3 0	15	11	7	3 0
1111	0110	0000	0000	0100	0000	0001	1010

Структура длинной команды (ADD)

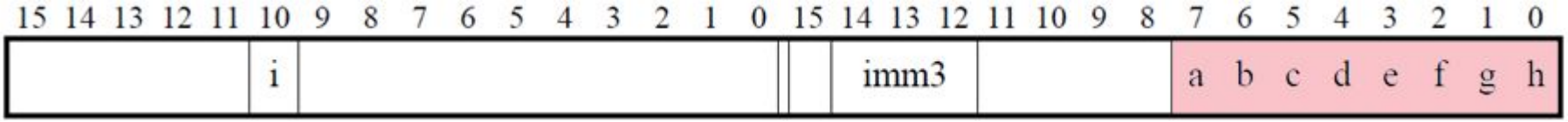


ADD Rn, Rd, Imm -> Rd = Rn + Imm32

- Зеленое – opcode (в частности показывает, что команда длинная)
- Rn – регистр-слагаемое
- Rd – регистр-приемник результата
- S – опциональный бит (обновлять ли флаги состояния)
- Imm32 – число, задаваемое i:imm3:imm8 – хитрым образом!

Это кодирование T3

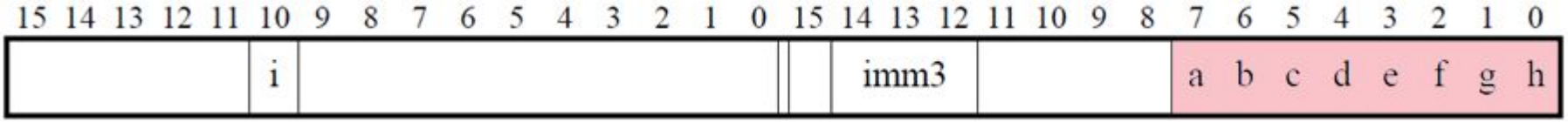
Непосредственный операнд при кодировании T3



Часть первая:

i	imm3	a	Результат (32 бита)
0	000	-	00000000 00000000 00000000 abcdefgh
0	001	-	00000000 abcdefgh 00000000 abcdefgh
0	010	-	abcdefgh 00000000 abcdefgh 00000000
0	011	-	abcdefgh abcdefgh abcdefgh abcdefgh

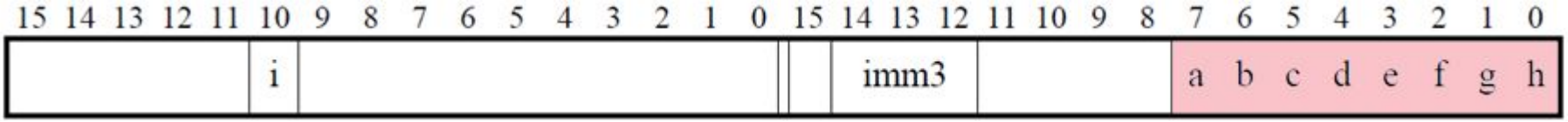
Непосредственный операнд при кодировании T3



Часть вторая:

i	imm3	a	Результат (32 бита)	Позиция младшего бита 32 - i:imm3:a
0	100	0	1bcdefgh 00000000 00000000 00000000	32 – 8 = 24
0	100	1	01bcdefg h0000000 00000000 00000000	32 – 9 = 23
0	101	0	001bcdef gh000000 00000000 00000000	32 – 10 = 22
0	101	1	0001bcde fgh00000 00000000 00000000	32 – 11 = 21
	...			
1	110	1	00000000 00000000 00001bc defgh000	32 – 29 = 3
1	111	0	00000000 00000000 000001b cdefgh00	32 – 30 = 2
1	111	1	00000000 00000000 0000001 bcdefgh0	32 – 31 = 1

Непосредственный операнд при кодировании T3



Часть вторая:

i	imm3	a	Результат (32 бита)	Позиция младшего бита 32 - i:imm3:a
0	100	0	1bcdefgh 00000000 00000000 00000000	32 – 8 = 24
0	100	1	01bcdefg h0000000 00000000 00000000	32 – 9 = 23
0	101	0	001bcdef gh000000 00000000 00000000	32 – 10 = 22
0	101	1	0001bcde fgh00000 00000000 00000000	32 – 11 = 21
	...			
1	110	1	00000000 00000000 00001bc defgh000	32 – 29 = 3
1	111	0	00000000 00000000 000001b cdefgh00	32 – 30 = 2
1	111	1	00000000 00000000 0000001 bcdefgh0	32 – 31 = 1

Пример команды ADD с кодированием T3

F5001090 ADD r0,r0,#0x120000

0x12 = 10010

0x12 0000 = 1001 (и 17 нулей)

Т.е. позиция значащего бита – 17? Не факт. Закодировать это число можно по-разному

F5001090 =

15	11	7	3 0	15	11	7	3 0
1111	0101	0000	0000	0001	0000	1001	0000

i = 1, imm3 = 1, a = 1 значит позиция младшего бита = $32 - 10011_2 = 13$

Значит, imm32 = 1001 0000 << 13.

Проверяем.

«Святые писания»

- Cortex-M3 Devices Generic User Guide:
  - Программная модель ядра процессора
  - Описание синтаксиса и семантики инструкций процессора
  - Периферия уровня ядра
- ARMv7-M Architecture Reference Manual:
  - Подробное описание ядра процессора
  - Кодирование инструкций
- STM32F10x Reference Manual:
  - Подробное описание всех периферийных устройств семейства микроконтроллеров
- STM32F103x8 Datasheet:
  - Электрические характеристики конкретной модели МК
  - Распиновка, габариты и т.д.

«Святые писания»

- Errata sheet:
  - Описание всех известных производителю ошибок в конкретном МК или серии МК

- Спецификация микроконтроллеров Миландр серии 1986ВЕ9х:
  - Единственный документ о МК Миландр
  - **Страницы 51-141 являются плохим переводом трех глав Cortex-M3 User Guide!**

Переходим к чудесам!

Краткий экскурс в историю

- Язык C появился в ~1973 году.
- Компьютеры были очень разные.
- C – «кроссплатформенный ассемблер».
- Поэтому **очень много** вещей в C зависят от платформы, для которой программа написана.
- Очень много вещей – это наследство от старых времен

Типы в языке C

- Типизация: статическая слабая и неявная.
- «Простые» типы:
  - _Bool (bool) – начиная с C99 - #include <stdbool.h>
  - char, signed char, unsigned char
  - short
  - int
  - long
  - long long
  - float, double, long double
  - void
  - указатели
- Композитные: ~~массивы~~, структуры, объединения
- Функции
- Выражения

Как искать чудеса?

- Читать Стандарт языка (C89, C99, C03)
- Искать на stackoverflow.com
- Просто писать программы! Чудо рано или поздно найдет вас.

Чудо первое: размеры типов

- Оператор sizeof возвращает размеры в размерах типа char
- Количество бит в одном char задается в макросе CHAR_BITS
- Для всех остальных типов задаются минимальные диапазоны

Чудо первое: размеры типов

Тип	**Размер в битах**
char	не менее 8
short	не менее 16
int	не менее 16
long	не менее 32
long long	не менее 64

Как жить с этим чудом? Использовать типы с известной длиной!

```
#include <stdint.h>
```

char – чтобы хранить символы (а не числа)
int8_t – знаковый 8 бит
uint8_t – беззнаковый 8

int16_t – знаковый 16 бит
uint16_t – беззнаковый 16 бит и т.д. вплоть до 64

Чудо второе: поведение

Поведение	Его смысл	Пример
Определенное (well defined)	Происходящее известно и однозначно описано в стандарте	`{ int a = 5; }`
Неуточняемое (unspecified)	Возможно конечное число вариантов, описанных в стандарте	`int a = pow( f1(), f2() );`
Платформозависимое (implementation-defined)	Поведение выбирается компилятором	`#pragma`
Неопределенное (Undefined)	NASAL DEMONS (описано в стандарте)	`int buf[5];` `buf[10] = 34;`
Не описанное	NASAL DEMONS 80 lvl (в стандарте не описано)	`int b = -1 << 30; // в C89`

Чудо второе: поведение

- Компиляторы *не всегда* предупреждают
- Компиляторы *не всегда* следуют стандарту

Как жить с этим чудом?

- Не выключать предупреждения (warning)
- **ЧИТАТЬ** предупреждения! Ценить, что они вообще есть.
- Помнить, что отсутствие предупреждений не означает, что все хорошо.

Чудо третье, неожиданное

Если поведение определенное, это еще не значит, что оно очевидное.

Примеры:

- Правила неявного преобразования типов
- Приоритеты операторов
- Арифметика с плавающей точкой (не только в C!)
- Макросы
- Указатели, указатели, указатели..

Компиляция с оптимизацией усиливают чудеса!

Чудо третье, неожиданное

Как жить с чудесами?

Учиться, учиться и еще раз учиться.
Или искать другой язык.

Но в каждом языке есть свои чудеса!

Немножко о стиле кода

- Пишите комментарии о *смысле* происходящего
- Если у вас больше трех переменных – называйте их *осмысленно*.
- **Ставьте отступы!**

Что такое отступы

```
void TimingDelay_Decrement(void)
{
  if (TimingDelay != 0x00)
  {
    TimingDelay--;
  }
}
```

Отступ – символ табуляции или 2 пробела или 4 пробела после { и до соответствующей }

Keil -> Edit -> Configuration -> Auto Indent = Block

Код без отступов читать очень неприятно