Есть ли у вас вопросы?

Краткое содержание предыдущей серии

- Как в ассемблере cortex m3 организована работа с памятью?
- Зачем нужны длинные команды?
- Какое бывает поведение в C?
- Что такое отступы?

Краткое содержание этой серии

- Числа со знаком
- Операции в языке C
- Ассемблерные команды, им соответствующие

Комментарии в Кейле

- Ne pishite kommentarii translitom
- Чтобы включить русский язык:
  - edit -> configuration -> encoding (UTF8)
  - edit -> configuration -> colors&fonts -> c/c++ editor -> font (courier new)
- Or just comment your code in english, that would be nice.

Двоичные числа

Допустим, у нас есть сетка из 4 разрядов. Сколько различных чисел мы можем хранить с ее помощью?

2^4 т.е. 16

Для чисел без знака диапазон выглядит вот так:

decimal	bin
15	1111
14	1110
13	1101
...	...
2	0010
1	0001
0	0000

Наименьшее число – 0
Наибольшее число – 15

Двоичные числа

Допустим, у нас есть сетка из 4 разрядов.
Но мы хотим хранить числа **со знаком.**
Сколько различных чисел мы сможем хранить?
по-прежнему 2^4 т.е. 16

Но диапазон будет **неизбежно** другой!
Например, 8 отрицательных чисел и 8 неотрицательных.

Как хранить знак?

Как хранить знак, если у вас есть только биты?

Например, назначить один бит знаковым!

А в остальных хранить модуль числа.

Этот способ называется «прямой код» - «sign and magnitude».

1	0	0	0	0	0	1	0	= -2

0	0	0	0	0	0	1	0	= 2

Прямой код

Плюсы:

- «Интуитивно понятен для человека»
- Удобен при программировании на ассемблере
- Используется в стандарте IEEE 754 – т.е. для представления чисел с плавающей точкой

Минусы:

- Два способа записи для числа 0 (+0 и -0)
- Сложная схемотехника для арифметических операций с числами разного знака
- Позиция знакового бита зависит от количества разрядов (т.е. от типа переменной).
  - Но можно сделать знаковым нулевой бит!

А как еще можно хранить знак?

Что должно быть?

- -1 + 1 должно быть равно 0
- -2 + 1 должно быть равно -1 (и далее, по индукции)

Вспоминаем о свойствах арифметики на ограниченной разрядной сетке.

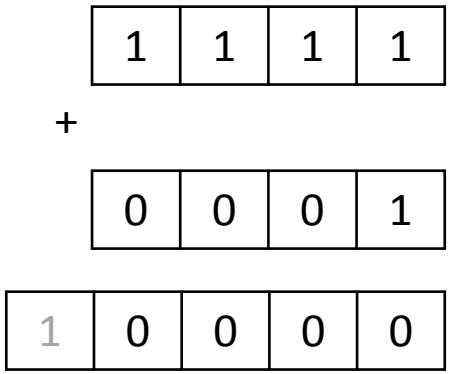
Сложение в ограниченной разрядной сетке

Допустим, что у нас есть 4 двоичных разряда. В них можно представить только 16 разных чисел.

А что будет, если мы возьмем число 15 и прибавим к нему 1?

Должно получится 16, но для этого нужен пятый разряд. А его нет. Поэтому бит просто «потеряется».

Это называется переполнение (сверху) – integer overflow.



Сложение в ограниченной разрядной сетке.

Получилось, что в сетке из четырех разрядов $15 + 1 = 0$.

Почему бы не отобрать двоичное представление числа 15 и не сказать, что так мы теперь кодируем -1?

А как представить -2? Так, чтобы -2+1 было равно -1.

По индукции, получаем следующее

1111_2 + 1 = 0 это -1 + 1

1110_2 + 1 = 1111_2 это -2 + 1

1101_2 + 1 = 1110_2 это -3 + 1

…

но нужно ведь когда-то остановится!

Удобно, если количество отрицательных и неотрицательных чисел одинаковое.

Тогда наименьшее отрицательное число будет

1000_2 – равное -8

Оставшиеся числа отдадим под неотрицательные.

Дополнительный код

decimal	bin
15	1111
14	1110
13	1101
12	1100
11	1011
10	1010
9	1001
8	1000
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000

decimal	bin
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Дополнительный код

На английском – «two’s complement» – «дополнение до двух».

Плюсы:

- Удобная арифметика – вычитание через сложение!
- Единственная запись числа 0
- Простая смена знака схемотехникой (инвертировать все биты и прибавить 1)
- Единица в старшем бите означает, что число отрицательное

Минусы:

- Не очень-то удобно для человека
- Отрицательных чисел на 1 больше чем *положительных*
- Запись одинаковых чисел зависит от разрядной сетки!

Отрицательные числа

Но это не единственные способы!

- Обратный код
- Нега-двоичная система (по основанию -2)
- ...

Сюрприз:

Стандарт языка C ***не*** описывает, как именно хранятся числа со знаком!

Арифметические операции в C

- +, -, *, / и %
- их комбинации с = (+=, -+ и т.д.)
- ++ и -- (пост- и пред-)
- сравнения
- индекс (a[3]), который на самом деле сложение и разыменование

% - взятие остатка от деления

Арифметика в ассемблере

- Целочисленная (в Cortex M3 «родная»)
- С плавающей точкой (в Cortex M3 организована программно)

- Векторная (в Cortex M3 отсутствует)
- SIMD (single instruction multiple data) (в Cortex M3 отсутствует)

Дальше речь только о целочисленной арифметике, для которой есть специализированные инструкции

Арифметика в ассемблере
Сложение

- ADD r0, r1, r2 – сложение с переполнением (overflow) (короткая и длинная версии)
- ADD**W** – длинная версия, поддерживает 12-битовый непосредственный операнд (**w**ide)
- ADD**S** – сложение с обновлением регистра состояния (**S**tatus); вообще S – это постфикс
- AD**C** – сложение с учетом флага **C**arry (переноса)
- AD**CS** - ?
- ADD**WS** и ADD**WC** - ... отсутствуют.

Арифметика в ассемблере
Что еще за флаг Carry?

Пусть мы складываем два десятичных числа из 3 цифр. Сколько цифр нам понадобиться (в худшем случае), чтобы записать результат?

4.

Почему?

Потому что в худшем случае: 999 + 999 = 1998

Это работает и в двоичном коде, ведь 1+1 = 10.

Арифметика в ассемблере
Что еще за флаг Carry?

Флаг Carry (он же бит переноса) и есть этот дополнительный двоичный разряд при сложении.

Флаг Carry находится в регистре состояний.

Префикс S у команды означает, что «команда влияет на регистр статуса» – в том числе, может установить флаг Carry

Т.е. ADDS r0, r1,r2 – складывает содержимое двух регистров и может установить флаг переноса. Происходит **точное** сложение, без выхода за разрядную сетку!

Арифметика в ассемблере
Что еще за флаг Carry?

ADC – сложение **с учетом** бита переноса (он просто прибавляется к слагаемым).

И зачем это нужно?

Чтобы складывать 64-битные (или еще более длинные) числа!

Сначала складываются младшие 32 бита (с выставлением бита переноса), потом складываются старшие 32 бита с учетом переноса!

Арифметика в ассемблере
Какие еще есть флаги?

- C – флаг Carry (перенос)
- N – флаг Negative (отрицательный результат)
- Z – флаг Zero (результат 0)
- V – флаг oVerflow (знаковое переполнение, «неверная» смена знака)

Есть и другие, но к арифметике они не относятся

Примеры

Для простоты, пусть у нас есть регистры из 4 бит

Сложение без переноса (ADD):

	1	1	1	1
+	0	0	0	1

1	0	0	0	0

Результат из пяти бит, регистр из 4 – последний бит просто теряется.
Это называется «переполнение» - overflow.
Постфикса S в команде нет – регистр флагов не обновляется

В данном случае получается, что 15+1 = 0

В некоторых случаях, это совершенно нормально (сложение по модулю 16)

Примеры

Для простоты, пусть у нас есть регистры из 4 бит

Сложение с обновлением регистра состояний (ADDS):

	1	1	1	1
+	0	0	0	1
1	0	0	0	0

Флаги			
C	N	Z	V
1	0	1	0

Результат из пяти бит, регистр из 4. Так как команда с постфиксом S – обновляется флаг Carry (и в нем, фактически, хранится пятый бит).

Так как в результат в регистре получился нулевой – выставляется и флаг Zero

Потери данных нет, четыре бита результата в регистре, пятый – флаг C

Примеры

Пусть у нас есть регистры из 4 бит, но мы хотим складывать 8-битные числа.

Пусть мы хотим сложить 0011 1111 + 0001 0001 (63+17).

Складывать придется по частям. Сначала младшие биты, потом старшие

Складываем младшие биты через ADDS, получаем младшие биты результата:

	1	1	1	1
+	0	0	0	1
	1	0	0	0	0

Флаги			
C	N	Z	V
1	0	1	0

Теперь складываем старшие биты с учетом флага carry (ADC) – с переносом:

	0	0	1	1
+	0	0	0	1
+				C
	0	1	0	1

Результат: 0101 0000 (80) в двух регистрах

Арифметика в ассемблере
Вычитание

- SUB r0, r1, r2 – вычитание, короткая и длинная версии (r0 = r1-r2) с переполнением снизу (underflow)
- SUB**W** – длинная версия с 12-битовым непосредственным операндом
- SUB**S** – вычитание с обновлением регистра состояния
- SB**C** – вычитанием с учетом Carry (если carry = 0 – вычесть еще 1)
- RSB r0,r1,r2 –> r0 = r2-r1 (вычитание наоборот).
- SB**CS**, RSB**S** – понятно
- RSBC, RSBW, SUBWS.. - отсутствуют

Арифметика в ассемблере
Сложение и вычитание

А где в сложении и вычитании учитывался знак?

А нигде. Но почему?

Потому что целые отрицательные числа в архитектуре ARMv7 хранятся в дополнительном коде!

А числа в дополнительном коде можно складывать и вычитать, не обращая внимания на знак.

Арифметика в ассемблере
Умножение

Допустим, мы умножаем 2 трехзначных числа. Сколько потребуется цифр, чтобы хранить результат?

6, к сожалению

Почему?

Потому что 999*999 = 998 001

Вывод: int32+int32 поместится в int32 (и бит carry)

int32*int32 поместится только в int64

Арифметика в ассемблере
Умножение

Операция	Смысл	Результат
MUL r0, r1, r2	r0 = r1*r2	32 бита
MLA r0, r1, r2, r3	r0 = r1*r2 + r3	32 бита
MLS r0, r1, r2, r3	r0 = r3 – r1*r2	32 бита
UMULL r0,r1,r2,r3	r0,r1 = r2*r3	беззнаковые 64 бита (в r0 младшие, в r1 старшие 32)
UMLAL r0,r1,r2,r3	r0,r1 = (r0,r1)+r2*r3	беззнаковые 64 бита (аналогично)
SMULL r0,r1,r2,r3	r0,r1 = r2*r3	знаковые 64 бита (в r0 младшие, в r1 старшие 32)
SMLAL r0,r1,r2,r3	r0,r1 = (r0,r1)+r2*r3	знаковые 64 бита (аналогично)

Арифметика в ассемблере
Деление

Делить в дополнительном коде, не обращая внимания на знак, к сожалению, нельзя.

На ноль тоже делить нельзя...?

Но при делении можно не беспокоится, что результат не влезет в разрядную сетку.. правда?

К сожалению, нельзя:

```
int16_t a = -32768;
int16_t b = -1;

a = a/b; // ???
```

Арифметика в ассемблере
Деление

- UDIV r0, r1, r2 r0 = r1/r2 (беззнаковое деление)
- SDIV r0,r1,r2 r0 = r1/r2 (знаковое деление)

На ноль делить нельзя.

Деление целочисленное, поэтому 1/2 = 0.

А как же сделать операцию % ?

a % b эквивалентно temp = a/b; result = a - temp*b; (так вот зачем нужен MLS!)

Деление в C

Помните:

- целочисленное деление на ноль – undefined behavior
- % (остаток от деления) для отрицательных чисел может быть неожиданным для вас
  - `30 % 4 = 2`
  - `-30 % -4 = -2`
  - `-30 % 4 = -2`
  - `30 % -4 = 2`

Стандарт определяет его как implementation-defined.

Сравнения в ассемблере

- CMP r0, r1 temp = r0 – r1, обновить регистр состояний, отбросить temp (аналогично SUBS temp, r0,r1)
- CMN r0,r1 temp = r0 + r1, обновить регистр состояний, отбросить temp (аналогично ADDS temp, r0,r1)
- TEQ r0,r1 (test equality), аналог ==, компилятором используется редко