

Архитектура ARM

Дмитрий Мельник

dm@ispras.ru

Архитектура ARM

- ARM: Advanced RISC Machine
 - Область применения: встраиваемые системы
 - Разработчик: компания ARM Holdings, лицензирует дизайн процессора производителям оборудования
- Основные особенности:
 - Энергоэффективность
 - Низкая стоимость
 - Относительно простое ядро
 - Расширяемость

Архитектура ARM

Архитектура	Семейство процессоров	Год	Примеры устройств
ARMv1	ARM1	1985	
ARMv2	ARM2, ARM3		
ARMv3	ARM6, ARM7	1992	
ARMv4	StrongARM, ARM7TDMI, ARM9TDMI	2003	iPaq 4150
ARMv5	ARM7EJ, ARM9E, ARM10E, XScale		
ARMv6	ARM11	2007	iPhone (orig, 3G)
ARMv7	Cortex A8, Cortex A9	2008	N900, iPhone (3GS, 4, 4S)
ARMv8	Пока нет	2011	

RISC vs CISC

(Reduced vs Complex Instruction Set Computer)

- Простые операции
 - Ограниченный набор простых команд (например, нет деления)
 - Команда выполняется за один такт
 - Фиксированная длина команды (простота декодирования)
- Конвейер
 - Каждая операция разбивается на однотипные простые этапы, которые выполняются параллельно
 - Каждый этап занимает 1 такт, в т.ч. декодирование
- Регистры
 - Много однотипных взаимозаменяемых регистров (могут использоваться и для данных, и для адресации)

RISC vs CISC

(Reduced vs Complex Instruction Set Computer)

- Модель работы с памятью
 - Отдельные команды для загрузки/сохранения в память
 - Команды обработки данных работают только с регистрами
- Сложность оптимизаций перенесена из процессора в компилятор
 - Производительность сильно зависит от компилятора
- Итого: более простое ядро, выше частота процессора

Режимы процессора ARM

- ARM (32 бит)
 - Режим по умолчанию
 - Доступны все команды процессора, работа с сопроцессором
- Thumb-1 (16 бит)
 - Высокая плотность кода, лучшее использование I-Cache
 - На 16-битной памяти команда передается за один такт
 - Ограниченный набор команд и регистров
- Thumb-2 (смешанный 16/32 бит)
 - Объединяет преимущества ARM и Thumb-1
 - Размер кода на 25% меньше при сравнимой производительности
- Jazelle (8 бит)
 - Аппаратно реализовано свыше 60% команд Java-байткода
 - Режим доступен только производителям оборудования

Регистры

- 16 регистров общего назначения
 - Размер: 32 бита, используются в целочисленных командах, полностью взаимозаменяемые
 - Именованное: r0 - r15
 - Некоторые регистры имеют специальные имена и назначение:
 - PC (r15) – Program Counter
 - LR (r14) – Link Register
 - SP (r13) – Stack Pointer

Регистры

- Регистры состояния
 - CPSR (Current Program Status Register): флаги, обработка прерываний
 - SPSR (Saved Program Status Register): хранит копию CPSR при обработке прерывания
 - Доступ: команды MRS/MSR
- Контрольные регистры
 - Изменения параметров кэширования, управления памятью, режимов процессора и т.п.
 - Доступ: команды MRC/MCR
- Вещественные регистры (FPU и векторного сопроцессора)

Условные флаги

Флаг	Описание	Когда устанавливается
Q	Saturation	Насыщение/переполнение в DSP-операциях
V	oVerflow	Переполнение (знаковое)
C	Carry	Перенос бита (беззнаковый)
Z	Zero	Нулевой результат (равенство)
N	Negative	Отрицательное значение (бит 31 установлен)

Флаги расположены в регистре CPSR (current program status register)



Установка флагов

Если команда ALU имеет суффикс 'S', то флаги будут установлены в соответствии с результатом команды.

Примеры:

```
subs r1, r2, #1
```

```
lsls r1, r2, #5
```

```
cmp r2, #1
```

Коды условий

Предикат	Отрицание	Флаги	
EQ	NE	Z	==
CS / HS	CC / LO	C	>= (беззнаковое)
MI	PL	N	< 0
VS	VC	V	переполнение
HI	LS	zC	> (беззнаковое)
GE	LT	NV nv	>= (знаковое)
GT	LE	NzV nzv	> (знаковое)
AL	-	-	Безусловная команда

Условное выполнение

Почти все команды ARM могут быть записаны в *условной форме*. В этом случае условие приписывается после команды, и она будет выполнена, только если условие истинно.

Примеры:

```
addge r1, r1, r1
```

вычисление модуля $|r1 - r2|$:

```
subs r1, r1, r2  
rsblt r1, r1, #0
```

Модуль сдвига

Второй аргумент ALU-команд может быть представлен в виде

$ARG2 = R \text{ shift_op } B$, где

R – регистр

B – величина сдвига (0-31)

$shift_op$ – один из *LSL*, *ASL*, *RSL*, *ROR* или *RRX*

Пример:

```
add r1, r1, r1 lsl #2 // r1 = r1 + r1 * 4 = r1 * 5
```

Представление констант

Второй аргумент ALU-команд может быть также константой, которая кодируется 12 битами (8 бит значение, 4 бита величина сдвига):

$$CONST_32 = CONST_8 \ll (2 * N), \quad 0 \leq N < 16$$

Примеры правильных и неправильных констант:

```
and r1, r1, #255
and r1, r1, #510
and r1, r1, #0xff00ff00
and r1, r1, #0xff000000
```

Представление констант

Второй аргумент ALU-команд может быть также константой, которая кодируется 12 битами (8 бит значение, 4 бита величина сдвига):

$$CONST_32 = CONST_8 \ll (2 * N), \quad 0 \leq N < 16$$

Примеры правильных и неправильных констант:

```
and r1, r1, #255
```

```
and r1, r1, #510 // 510 = 255 << 1 – нечетный сдвиг
```

```
and r1, r1, #0xff00ff00 // значение «шире» 8 бит
```

```
and r1, r1, #0xff000000
```

Кодировка команд

ADDGE r0, r1, r2 asr #31

Действие:

```
if (flags_match (GE))
```

```
    r0 = r1 + r2 >> 31;
```

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										

GE ALU ADD 0 1 0 0 31 ASR 2
(w/reg)

Действие:

```
r0 = r1 & ~0xff000000;
```

```
set_flags();
```

BICS r0, r1, #FF000000

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															

Режимы процессора ARM

	ARM	Thumb-1	Thumb-2
Размер команды	32 бит	16 бит	16/32 бит
Кол-во команд	~60	30	~60
Условное выполнение	Почти все команды	Нет	С помощью IT-блоков
Сдвиг аргументов	Во всех командах ALU	Отдельные команды	В 32-битных командах
Доступные регистры	15 общего назначения + pc	8 общего назначения + 7 специальных + pc	15 общего назначения + pc
Примеры команд	<pre>add r1, r2, r3 asl #2</pre> <p>(A)</p>	<pre>add r1, r2</pre> <p>(B)</p>	<p>Обе формы допустимы; Размер (A) – 16 бит, (B) – 32 бит</p>

Задания

1. Что делают команды

```
adds    r1, r0, #42  
bicge   r0, r0, r0, asr #31
```

(Эквивалент на C: `if (x >= -42) { ... }`)

2. Вычислить $X = X * 81$ в две команды без использования MUL
($81 = 5 * 16 + 1$ или $(8 + 1)^2$)

3. Записать в одну команду выражение

$X = (X \geq 0) ? X : X - 1$

(Использовать ADD, ASR)

Команды работы с памятью

Команды загрузки/сохранения LDR / STR:

```
ldr r1, [r2, #+/-imm12]  
ldr r1, [r2, +/-r3, shift imm5]
```

Примеры:

```
ldr r1, [pc, #256]  
ldr r1, [sp, r2, asl #2]  
L1:  
ldr r1, L1 + 248  
ldr pc, [r1, r2, asl #2] // table-jump для switch
```

Команды работы с памятью

Преинкремент / постинкремент адресного регистра:

```
ldr r1, [r2, +/-r3, shift imm5]!  
ldr r1, [r2], +/-r3, shift imm5
```

Множество регистров:

```
ldmia r1, {r1-r16}  
ldmia r1!, {r1-r16} - с обратной записью в адресный регистр
```

Суффиксы:

ia – increment after

db – decrement before

Множество регистров: любое подмножество
в порядке возрастания номеров (например,
{r0, r1-r10, pc})

Команды работы со стеком

Работа со стеком реализуется через команды
`stmdb/ldmia`

```
push {r0-r15}
```

```
stmdb sp!, {r0, r15}
```

```
pop {r0-r15}
```

```
ldmia sp!, {r0, r15}
```

<code>sp -> 0x100</code>	Последнее доступное значение
<code>0x0FC</code>	<code>r0</code>
<code>0x0F8</code>	<code>r1</code>
<code>...</code>	<code>...</code>
<code>0x0C0 <- sp (после)</code>	<code>r15</code>

Вызов функций

Вызов функций выполняется при помощи команды `bl` – *branch with link*

Текущее значение регистра `pc` сохраняется в регистре `lr`

Возврат из функции выполняется с помощью команды `bx lr`

```
some_func:                               bl some_func
    push {lr}
    ...
    pop {pc}
```

```
some_func:
    bx lr
```

Конвейер современного ARM

- Cortex-A8
 - Конвейер из 13 стадий
 - 2 АЛУ устройства, 1 Load/Store, 1 Multiply
 - Может выполнять до 2-х команд за такт
 - Из них должно быть не более одной Load/Store и Multiply, причем умножение должно идти первым
- Cortex-A9: добавлено динамическое переупорядочение команд

Примеры описания времени выполнения команд на конвейере

- Этапы конвейера обозначаются E1, E2, ..., E5
- Перед началом каждого этапа команда может требовать готовности операндов в регистрах, а после – выдавать готовые операнды
- ADD r1, r2, r3
 - r2, r3: требуются перед E2
 - r1: после E2
- MOV r1, r2 asl #const
 - r2: требуется перед E1
 - r1: готов после E1
- Следствие 1: `mov r2, r1; add r3, r2, r1` могут начать выполняться одновременно
- Следствие 2: `add r1, r2, r3; mov r2, r1` (в обратном порядке) имеют задержку в 2 такта между командами

Селективный планировщик

-O2

```
.L11:
      ldr ip, [r3, r7]
      ldr r4, [r3, r6]
      add r3, r3, #4
      cmp r3, r5
      mla r2, r4, ip, r2
      mul r1, r2, r1
      bne .L11
```

2 cycles { ldr, ldr }
5 cycles { mla, mul }

До:
12 тактов

Source

```
int foo(int N) {
    int i;
    int a=0, b=1;
    for (i=0; i<N; i++)
    {
        a += x[i] * y[i];
        b *= a;
    }
    return a+b;
}
```

Selective Scheduling

```
Prologue { ldr r4, [r3, r7]
           mov ip, #1
           ldr r5, [r3, r8]
           mov r2, r3
.L15:
Kernel { mla r2, r5, r4, r2
         cmp r1, r6
         mov r3, r1
         ldrne r4, [r3, r7]
         addne r1, r3, #4
         ldrne r5, [r3, r8]
         mul ip, r2, ip
         bne .L15
```

После:
8 тактов

Ускорение
8%

Селективный планировщик выполняет конвейеризацию циклов

- «Скрывает» задержки у долгих команд, перенося их на предыдущую итерацию цикла
- Помогает использовать параллелизм на уровне команд
- Среднее ускорение 1-3%

Улучшения кодогенерации для ARM Advanced SIMD (NEON)

- **Better support for NEON instructions**

Например, поддержка комбинирования $|a - b|$ в одну команду NEON:

vsub
vabs \longrightarrow vabd

Производительность: **+2.5%** (x264)
Уменьшение размера: **0.1%** (x264)

- **Улучшение распределения регистров для векторов констант**

Запрет на размещение векторных констант в обычных ARM регистрах

Производительность: **+3%** (evas)

- **Поддержка преобразования из float в int в векторизации**

Циклы, в которых используется преобразование типов, теперь могут быть векторизованы с использованием команды `vcvt`

Производительность: **+9%** (libmp3lame)

Улучшение оптимизации GCSE и комбинирования команд в GCC

Исходный пример

```
if (x)
  a = b + c << 2;
else
  d = e + c << 2;
```

Сейчас в GCC (неверно)

```
mov r1, r2, asl #2
cmp r3, #0
bne L1
add r4, r5, r1
b .L2
.L1:
add r6, r7, r1
```

После исправления

```
cmp r3, #0
bne L1
add r4, r5, r2, asl #2
b .L2
.L1:
add r6, r7, r2, asl #2
```

Проблема:

- Оптимизация GCSE (Global Common Subexpression Elimination) «не знает» о возможности ARM barrel shifter
- Комбинирование команд (которое о нем «знает») работает только в пределах базового блока

Решение:

- Исправить GCSE
- Доработать комбинирование

Результаты:

- Сокращение размера кода: **3.6 Кб** на SPEC 2K INT (**0.1%**)
- Используется меньше регистров

Улучшения обработки условных команд Thumb-2

Problems:

- Команда внутреннего представления RTL `if-then-else` раскрывается напрямую в ассемблер слишком поздно
- Слишком ранняя оптимизация коротких Thumb-2 команд мешает последующему преобразованию в условную форму
- IT блоки могут быть разделены в планировщике

Solutions:

- Раньше раскрывать `if-then-else` в RTL
- Изменить порядок оптимизаций в GCC
- Отдавать приоритет командам с тем же предикатом в планировщике

RTL псевдо-код

```
a = (x == 0) ? 1 : 2;  
b = (x == 0) ? 3 : 4;
```

До

```
cmp r1, #0  
ite eq  
moveq r2, #1  
movne r2, #2  
ite eq  
moveq r3, #3  
movne r3, #4
```

После

```
cmp r1, #0  
itete eq  
moveq r2, #1  
movne r2, #2  
moveq r3, #3  
movne r3, #4
```

Улучшения обработки условных команд Thumb-2

Другие улучшения:

- Преобразовывать максимум 4 команды, чтобы не увеличивать код
- Не преобразовывать команды в условную форму, если вероятность одной дуги значительно выше другой (напр. 90% к 10%)

Исходный пример

```
cmp r1, #0
bne .L2
mov r1, #1
mov r2, #2
mov r3, #3
mov r4, #4
mov r5, #5
.L2:
...
```

Преобразованный в условную форму

```
cmp r1, #0
itttt eq
moveq r1, #1
moveq r2, #2
moveq r3, #3
moveq r4, #4
it eq
moveq r5, #5
...
```

Условный переход удаляется, один IT блок - ОК

Преобразование более 5 команд «стоит» лишней IT-команды

Улучшения обработки условных команд Thumb-2

Другие улучшения:

- Преобразовывать максимум 4 команды, чтобы не увеличивать код
- **Не преобразовывать команды в условную форму, если вероятность одной дуги значительно выше другой (напр. 90% к 10%)**

Исходный пример

```
cmp r1, #0
bne .L2
mov r1, #1
mov r2, #2
mov r3, #3
mov r4, #4
mov r5, #5
.L2:
...
```

Если вероятность перехода высокая, то лучше полагаться на branch prediction

Преобразованный в условную форму

```
cmp r1, #0
itttt eq
moveq r1, #1
moveq r2, #2
moveq r3, #3
moveq r4, #4
it eq
moveq r5, #5
...
```

Иначе конвейер может быть заполнен кодом, который никогда не выполняется

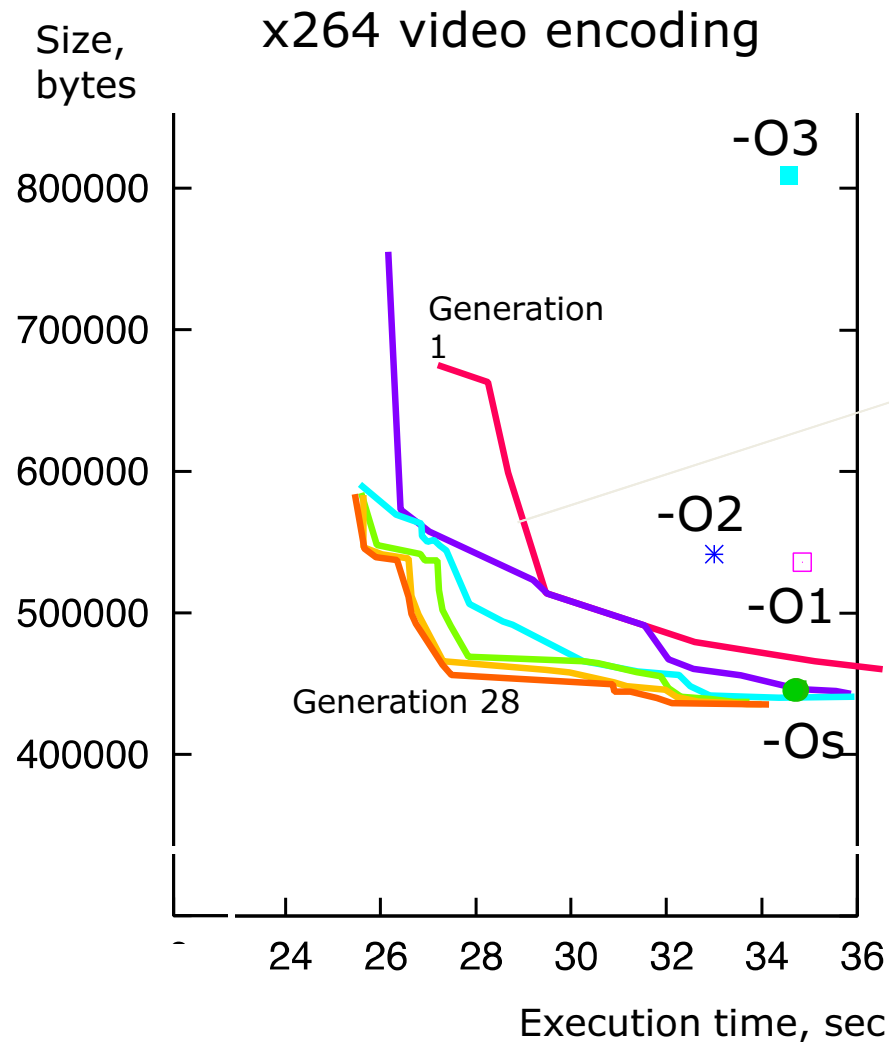
Улучшения обработки условных команд Thumb-2

Итого:

- Раньше раскрывать `if-then-else` в RTL
- Изменить порядок оптимизаций в GCC
- Отдавать приоритет командам с тем же предикатом в планировщике
- Преобразовывать максимум 4 команды, чтобы не увеличивать код
- Не преобразовывать команды в условную форму, если вероятность одной дуги значительно выше другой (напр. 90% к 10%)

4% сокращение размера на SPEC2K INT

Automatic Tuning

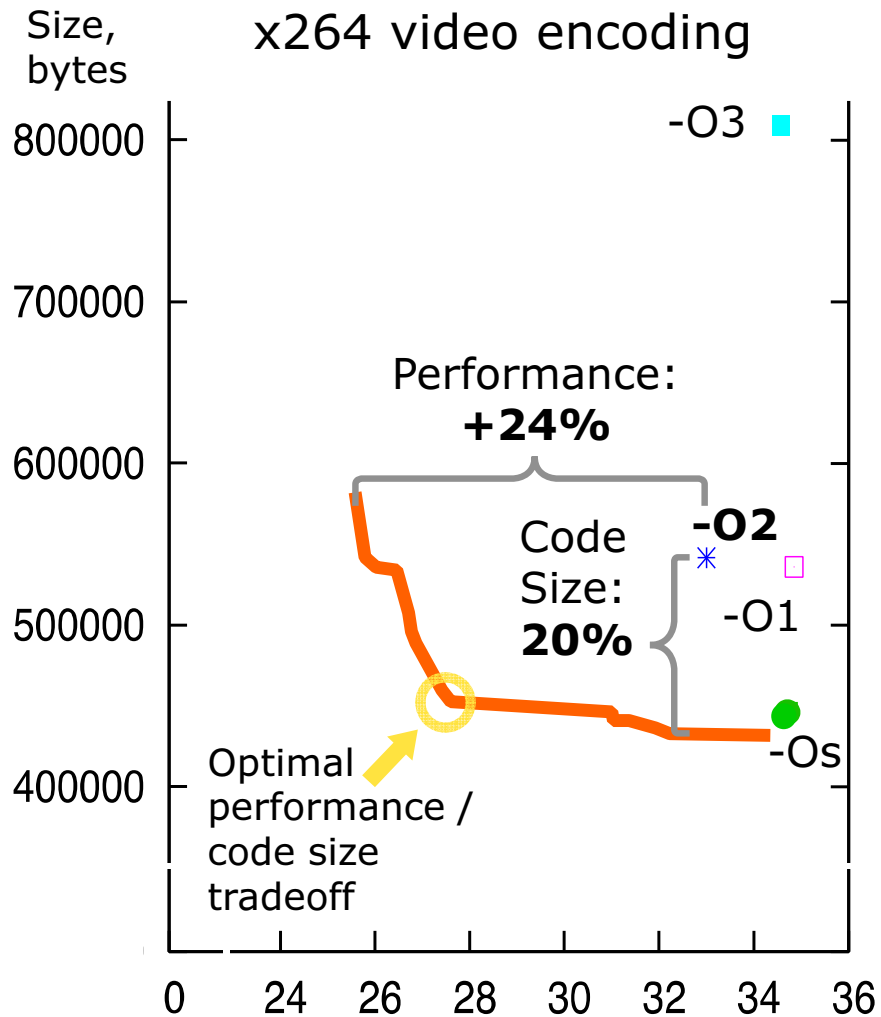


Each point on the graph corresponds to some set of compiler options (can contain up to ~200 options), e.g.:

```
-O2 -ftree-vectorize
-fno-gcse -fprefetch-
loop-arrays ...
```

- Better than GCC's -Os, -O2, -O3
- Gives control over performance / code size tradeoff

Automatic Tuning



- Better than GCC's -Os, -O2, -O3
- Gives control over performance / code size tradeoff

LTO & Visibility Tool

- **LTO (Link Time Optimization)**
 - new feature in GCC that enables its optimizations to work at inter-module scope
- **Current LTO problems:**
 - Applications bundle wrong libtool version
 - Applications often do not set correct function visibility, so cross-module optimization doesn't work

Our tool prepares shared libraries for building with LTO:

- Diagnoses problems in build scripts and tools
- Automatically corrects function visibility, making sure only library API functions are exported

Corrected visibility results in substantial improvements in both code size and performance due to reduced symbol table size and better inlining opportunities

Code size reduction: **5%** average (up to **13%**)
Performance: up to **7-16%** on selected apps

Проект по оптимизации GCC под ARM

