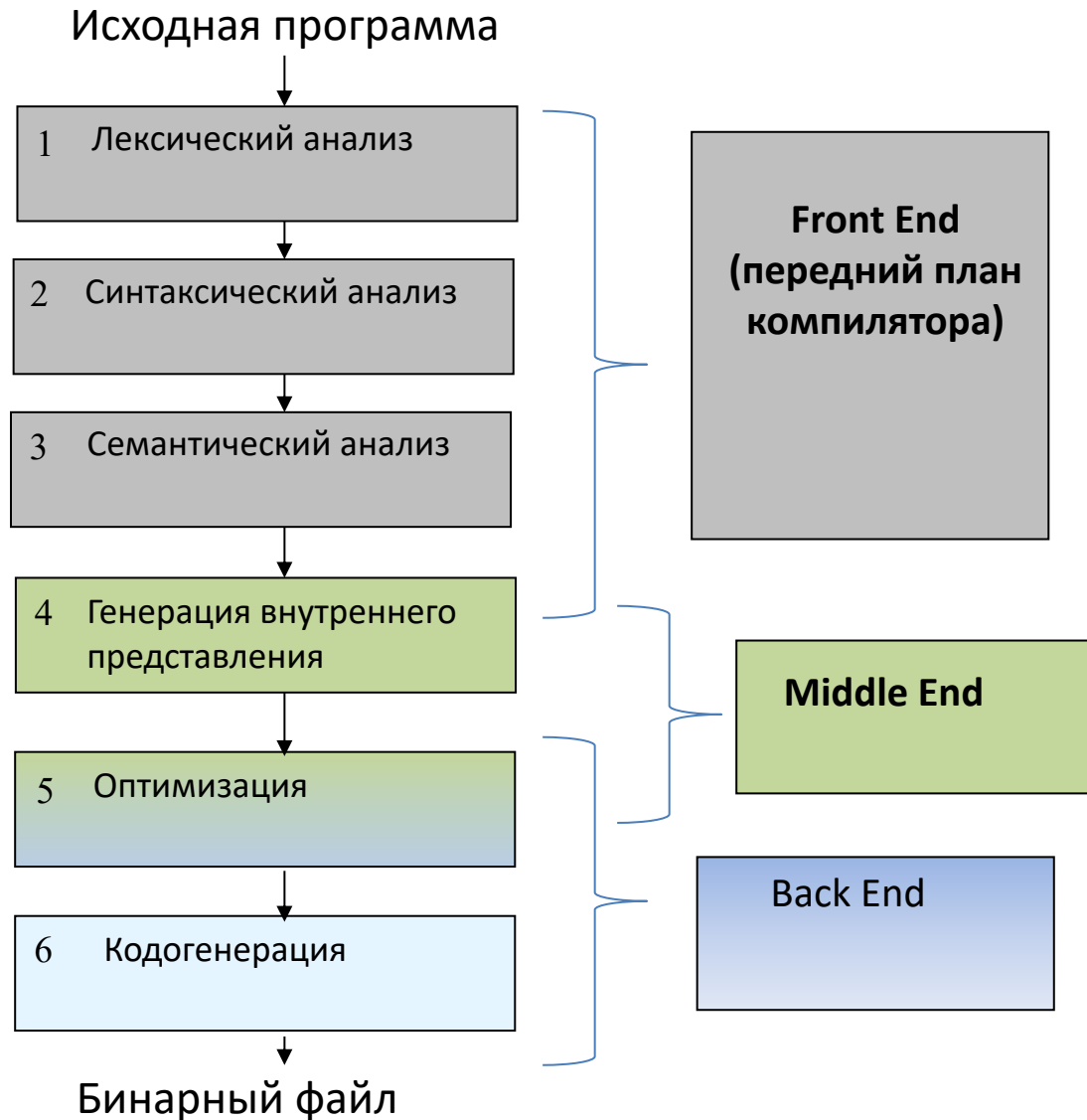


**13. Введение в
машинно-зависимые
оптимизации.
Архитектура процессора
ARM.**

Фазы компиляции



Внутреннее представление

- Трехадресное представление:

```
dest = OP arg1, arg2
```

- Неограниченное число псевдорегистров

- LLVM:

```
%3 = load i64, i64* %1  
%4 = load i64, i64* %2  
%5 = add nsw i32 %3, %4
```

- GCC:

```
(set (mem:SI (plus:SI (reg:SI 3 bx) (const_int 4)))  
      (reg:SI 85))
```

(Представление для операции записи в память [bx + 4] = @reg85)

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

Архитек- тура (ISA)	Семейство процессоров	Год	Примеры устройств	Разряд- ность
ARMv1	ARM1	1985		32 бит
ARMv2	ARM2, ARM3			
ARMv3	ARM6, ARM7	1992		
ARMv4	StrongARM, ARM7TDMI, ARM9TDMI	1993- ~2001	Nintendo DS, Nokia 6110, iPod, iPaq 4150	
ARMv5	ARM7EJ, ARM9E, ARM10E, XScale			
ARMv6	ARM11	2002	iPhone (orig, 3G)	
ARMv7	Cortex-A8, A9, A15, A7 Cortex-M	2005+	iPhone (3GS, 4, 5), iPad 1-4, Samsung Galaxy S1-S4, PS Vita; Today: IoT, MCUs	
ARMv8	Cortex-A57, A53, ...	2012+	iPhone 5S и выше	64 бит

Классические подходы RISC и CISC

(Reduced vs Complex Instruction Set Computer)

- Простые операции
 - Ограниченный набор простых команд (например, нет целочисленного деления – оно появилось только в ARMv7s)
 - 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

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

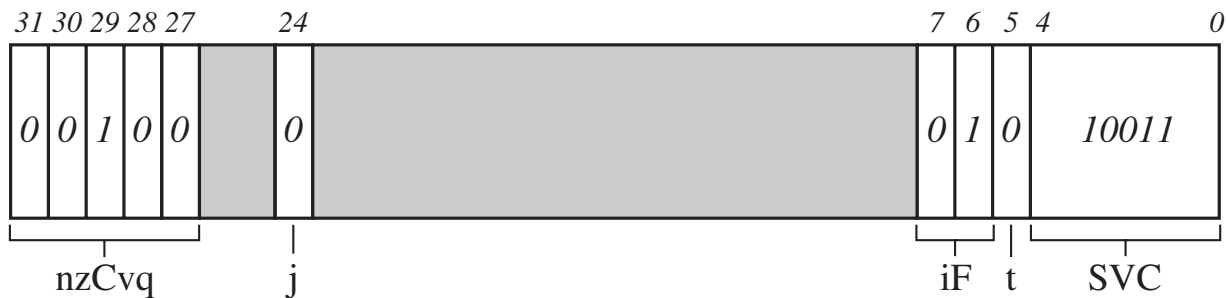
Регистры

- Вещественные регистры (FPU и векторного сопроцессора)
- 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

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)



Условные флаги / предикаты

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)

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

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
```

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
```

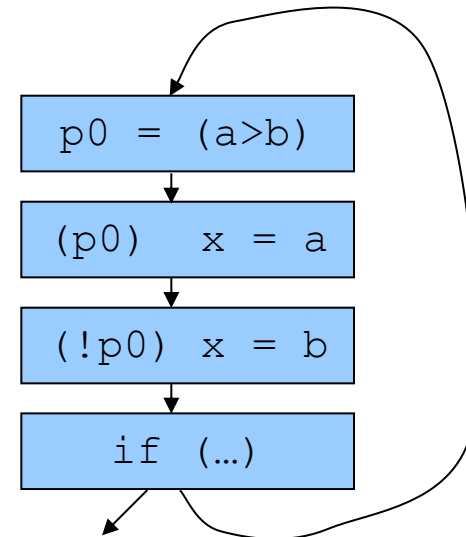
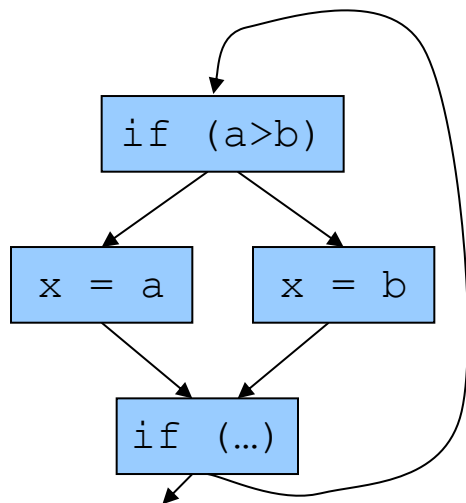
вычисление модуля $|r1 - r2|$:

```
subs r1, r1, r2
```

```
rsblt r1, r1, #0
```

Преобразование команд в условную форму

- Команды условного перехода заменяются командами условного выполнения
- Количество ветвлений сокращается, это позволяет:
 - Избежать сбоя конвейера при неправильном предсказании перехода
 - Выполнять конвейеризацию циклов (см. далее)



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 //  $r1 = r1 + r1 * 4 = r1 * 5$`

Примеры

1. Вычислить $X = X * 81$ в две команды без использования MUL

2. Записать в одну команду выражение:

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

Примеры

1. Вычислить $X = X * 81$ в две команды без использования MUL

Решение:

$81 = 9^2 = (8 + 1)^2$ [или $81 = 5 * 16 + 1$]

`add r1, r0, r0 asl #3` *// $r1 = r0 * 9 = r0 + r0 * 8 = r0 + (r0 \ll 3)$*

`add r2, r1, r1 asl #3` *// $r2 = r0 * 81 = r1 * 9$*

2. Записать в одну команду выражение:

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

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

Решение:

`add r0, r0, r0 asr #31`

Примеры

3. Какое значение будет в *r0* после выполнения команд:

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

Примеры

3. Какое значение будет в *r0* после выполнения команд:

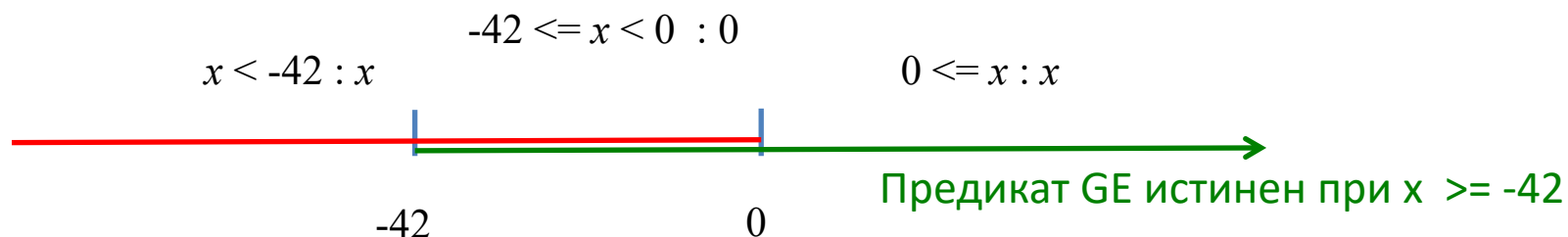
```
add $\mathbf{s}$     r1, r0, #42           ;; (1)
```

```
bic $\mathbf{ge}$     r0, r0, r0 asr #31    ;; (2)
```

Решение:

(1) $x = r0, y = r1; y = x + 42; y \geq 0$ при $x \geq -42$

(2) Команда `bic $\mathbf{ge}$  r0, r0, r0 << 31` по условию GE и при $x < 0$ обнуляет r0



`asr #31` = арифметический сдвиг на 31 бит вправо, т.е. копирование знакового бита
`bic` = битовый «and not», т.е. все биты, установленные во 2-м аргументе, будут сброшены

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      // по L1 + 248 находится таблица адресов  
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
```

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 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												

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
```

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 командах длиной 32 бит
Доступные регистры	15 общего назначения + pc	8 общего назначения + 7 специальных + pc	15 общего назначения + pc
Примеры команд	<code>add r1, r2, r3</code> <code>asl #2</code> (A)	<code>add r1, r2</code> (B)	Обе формы допустимы; Размер (B) – 16 бит, (A) – 32 бит

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

Уровни параллелизма в процессоре

- **Конвейер**
 - Позволяет параллельно исполнять инструкции, находящиеся на разных стадиях исполнения
- **Параллелизм функциональных устройств**
 - Несколько конвейеров / функциональных устройств

Микроархитектура ARM (32-bit)

- Cortex-A8, A7
 - 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, A15, ... : используется 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: готов 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

Примеры расписания выполнения команд на конвейере

- **ADD r1, r2, r3**
 - r2, r3: требуются перед E2
 - r1: готов после E2
- **MOV r1, r2 asl #const**
 - r2: требуется перед E1
 - r1: готов после E1
- Такие две команды могут начать выполняться на одном такте:

	E1	E2
mov r1, r2	R2 R1	
add r2, r1, r2		R1, R2 R2

← Стадии конвейера

- При выдаче этих команд в обратном порядке появится задержка в 2 такта:

	E1	E2	E3
add r2, r1, r2	--	R1, R2 R2	
mov r1, r2	Ожидание готовности R2		R2 R1

Обозначения:

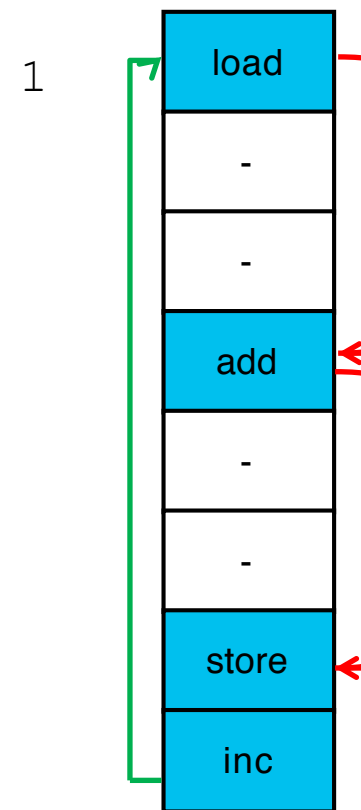
Rn – Регистр требуется в начале стадии выполнения E_i

Rk – Записывается в конце стадии E_i

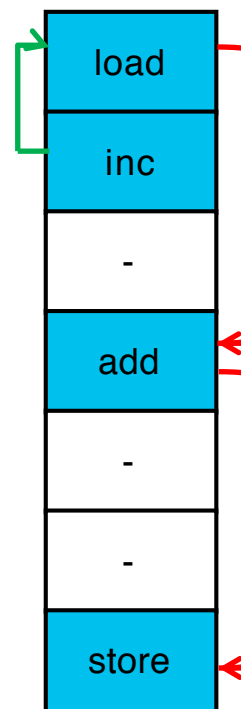
Планирование команд

```
loop:  
  arr[i] = arr[i] + 1  
  i++  
  goto loop
```

Задача: требуется минимизировать длину расписания так, чтобы зависимости по данным между командами были удовлетворены



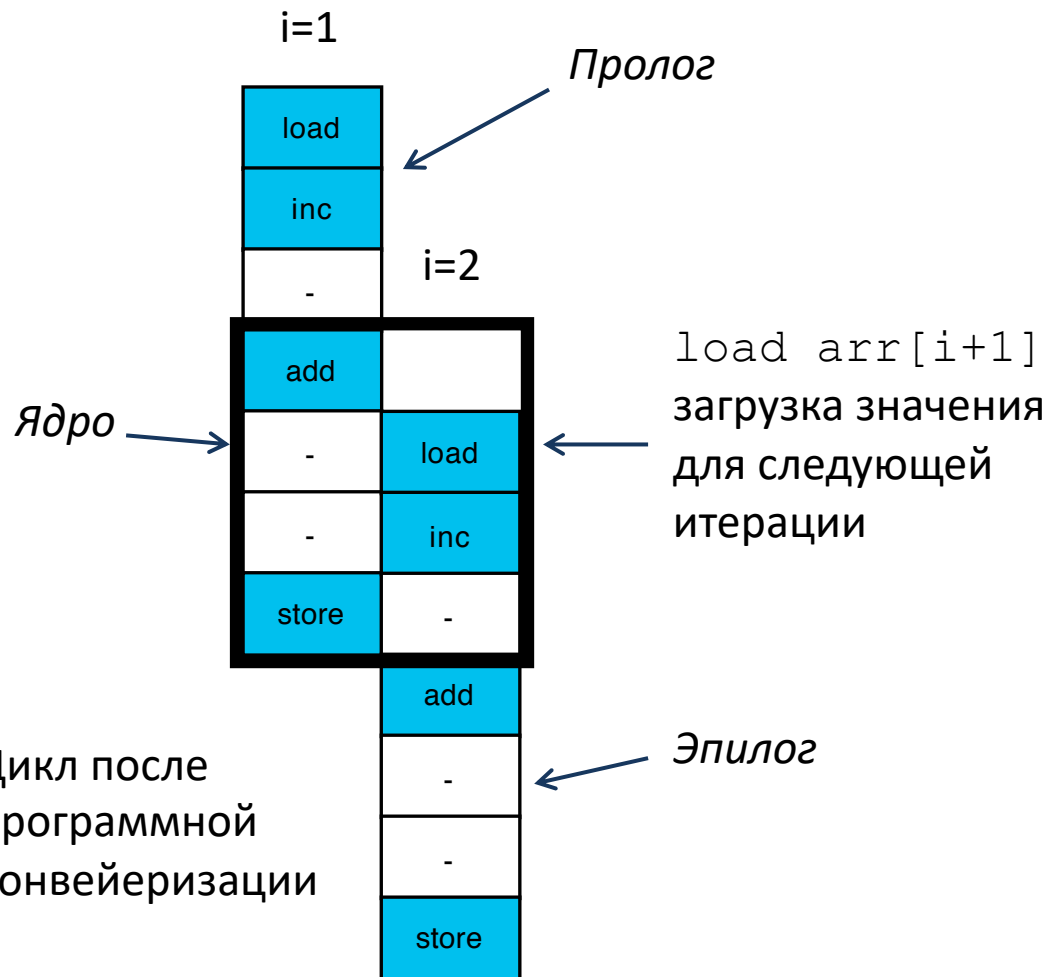
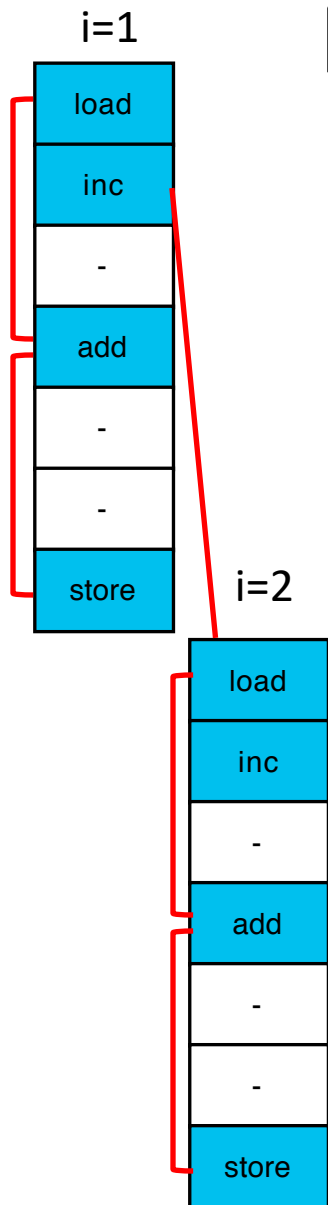
Исходный цикл



Цикл после планирования команд

зависимости по данным в пределах одного базового блока (**read-after-write** и **write-after-read**)

Программная конвейеризация ЦИКЛОВ



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
5 cycles

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:
    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%

Улучшение оптимизации GCSE и комбинирования команд в GCC

Исходный пример

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

Было 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 Kb** на SPEC 2K INT (**0.1%**)
- Используется меньше регистров

Улучшения обработки условных команд Thumb-2

Проблемы:

- 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

Решения:

- 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%)**

Если вероятность перехода высокая, то лучше полагаться на branch prediction

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
...
```

Иначе конвейер может быть заполнен кодом, который никогда не выполняется