Глава 3

Язык ассемблера MCPU

Эта глава посвящена собственно языку ассемблера MCPU, низкоуровневому ABI и жизненному циклу исполняемых программ. Мы начнём с программной модели процессора, регистров, стека и управляющих конструкций, затем подробно разберём функции, системные вызовы, статическую и динамическую компоновку и завершим главу программными моделями и численным примером маятника. Вопросы составных типов, указателей, массивов, управления памятью, runtime и построения компилятора вынесены в следующую главу «Системное программирование». Цель главы — дать законченное низкоуровневое представление о том, что именно исполняет MCPU и какие соглашения должен соблюдать ассемблерный код.

Andrey V. Kosteltsev · октябрь 2026

Оглавление

3 Язык ассемблера MCPU **101**

3.1 Программная модель MCPU . . . 101
3.1.1 Регистровый файл и перекрывающиеся представления . . . 101
3.1.2 Абсолютные символы .equ и имена регистров . . . 103
3.1.3 PC, SP, FP и PSW . . . 104
3.1.4 Независимые адресные пространства . . . 104
3.1.5 PTR64: адрес состоит из сегмента и смещения . . . 105
3.1.6 Непосредственные данные в программе . . . 107
3.1.7 Прямая и косвенная адресация . . . 111
3.1.8 Почему нельзя упорядочивать указатели из разных сегментов . . . 112
3.1.9 Что должен запомнить программист . . . 112
3.2 Первые инструкции и наблюдение состояния процессора . . . 112
3.2.1 Минимальная программа . . . 112
3.2.2 Сборка . . . 113
3.2.3 Перекрытие можно увидеть тем же способом . . . 114
3.2.4 PSW пока остаётся рядом . . . 115
3.3 Стек . . . 115
3.3.1 Как растёт STACK . . . 115
3.3.2 PUSH и POP . . . 116
3.3.3 Первый эксперимент со стеком . . . 116
3.3.4 SP — это граница живой памяти . . . 117
3.4 Подпрограммы и соглашение о вызовах . . . 118
3.4.1 Что делает CALL . . . 118
3.4.2 Простейшая функция с двумя параметрами . . . 118
3.4.3 Где находится возвращаемое значение . . . 119
3.4.4 Один параметр всё равно требует ABI-размещения . . . 120
3.4.5 Frame pointer: когда одного SP становится мало . . . 120
3.4.6 PSW и вызов функции . . . 121
3.5 Параметры командной строки . . . 122
3.5.1 Startup area создаёт loader . . . 122
3.5.2 Почему argv является PTR64 . . . 123
3.5.3 Минимальный пример argc/argv . . . 123
3.5.4 Разберём адресацию по шагам . . . 124
3.5.5 Startup area и обычный стек функций . . . 125
3.6 Объекты данных в программе . . . 126
3.6.1 Typed objects и raw bytes . . . 126
3.6.2 Строки: .utf8 и .ucs2 . . . 128
3.6.3 Выравнивание является частью layout . . . 130
3.6.4 Нулевая память: .bss и .zero . . . 130
3.6.5 Label не является PTR64 object . . . 131
3.6.6 Указатели как объекты данных: .ptr и .cptr . . . 131
3.6.7 Один пример со всеми основными объектами . . . 133
3.6.8 Что отсюда понадобится будущему компилятору . . . 134
3.7 Наблюдение за ходом выполнения: минимальный вывод в stdout . . . 135
3.7.1 Строка в DATA и завершающий NUL . . . 135

3.7.2 Имена символов и локальные метки . . . 136
3.7.3 Как print находит длину строки . . . 136
3.7.4 Короткий предварительный взгляд на SYSCALL ABI . . . 137
3.7.5 Полный служебный пример . . . 138
3.8 PSW, сравнения и переходы . . . 140
3.8.1 ICMP выполняет мысленное a-b . . . 140
3.8.2 JSET и JCLR — фундаментальные conditional branches . . . 141
3.8.3 Равенство и неравенство . . . 142
3.8.4 Unsigned ordering . . . 142
3.8.5 Почему для signed ordering недостаточно SF . . . 143
3.8.6 Таблица отношений как контракт генератора кода . . . 144
3.8.7 PSW является временным состоянием . . . 145
3.8.8 Прямой и косвенный control transfer . . . 145
3.8.9 Первый наблюдаемый branch . . . 145
3.9 Basic blocks и граф потока управления . . . 147
3.9.1 Почему labels сами по себе ещё не CFG . . . 147
3.9.2 Entry, exit, predecessor и successor . . . 148
3.9.3 AST и CFG отвечают на разные вопросы . . . 148
3.9.4 Метки как symbolic control-flow targets . . . 149
3.9.5 Условие лучше lowering-ить в два target-a . . . 149
3.9.6 Терминатор и fall-through как ресурс генератора кода . . . 150
3.9.7 CFG — не только картинка для учебника . . . 151
3.10 Условный оператор: от семантики к assembler . . . 151
3.10.1 if без else . . . 151
3.10.2 if/else создаёт две альтернативы и join . . . 152
3.10.3 Порядок blocks не обязан повторять порядок source . . . 154
3.10.4 Цепочка else if является деревом решений . . . 154
3.10.5 Join block и значения, созданные в разных ветвях . . . 155
3.10.6 Условия с side effects требуют сохранения порядка . . . 156
3.10.7 Semantic procedure для if/else . . . 156
3.11 Логические выражения и short-circuit semantics . . . 157
3.11.1 Логическое AND как граф . . . 157
3.11.2 Логическое OR . . . 158
3.11.3 Logical NOT обычно не требует instruction . . . 158
3.11.4 De Morgan как преобразование control flow . . . 158
3.11.5 Branch context и value context . . . 159
3.11.6 Почему нельзя заменять short-circuit обычным IAND . . . 159
3.11.7 Полный пример short-circuit . . . 160
3.11.8 Compiler context для boolean lowering . . . 161
3.12 Циклы: while, do while, for, break и continue . . . 162
3.12.1 while: сначала test, потом body . . . 162
3.12.2 do while: body является entry цикла . . . 163
3.12.3 for содержит четыре логические части . . . 163
3.12.4 continue не всегда идёт в test . . . 164
3.12.5 Loop context как стек semantic state . . . 165
3.12.6 Форма условия и направление branch . . . 166
3.12.7 Минимальный runnable loop . . . 166
3.12.8 Массивный цикл и pointer induction variable . . . 167
3.12.9 Natural loop как compiler object . . . 168
3.13 switch: выбор между многими ветвями . . . 168

3.13.1 Семантика сначала, стратегия потом . . . 169
3.13.2 Линейная цепочка comparisons . . . 169
3.13.3 Дерево решений для разреженного множества . . . 171
3.13.4 Jump table для плотного диапазона . . . 171
3.13.5 Bounds check обязателен . . . 172
3.13.6 Runnable jump-table example . . . 173
3.13.7 Как генератор кода выбирает стратегию . . . 175
3.13.8 break в switch и stack control contexts . . . 175
3.13.9 От switch к следующему крупному этапу . . . 175
3.14 Функции . . . 176
3.14.1 Граница вызова . . . 176
3.14.2 Возвращаемое значение . . . 176
3.14.3 Frame Pointer: неподвижная система координат кадра . . . 176
3.14.4 Динамический локальный объект . . . 177
3.14.5 PSW и рабочие регистры через границу CALL . . . 178
3.14.6 Косвенный вызов . . . 178
3.14.7 Функция как объект межмодульного контракта . . . 178
3.15 Системные вызовы . . . 179
3.15.1 Фиксированный регистровый ABI . . . 179
3.15.2 Результат и ошибки . . . 179
3.15.3 Заголовки `include/sys/asm` . . . 180
3.15.4 Полная таблица реализованных номеров . . . 180
3.15.5 Классы системных вызовов . . . 181
3.15.6 Память guest как аргумент . . . 181
3.15.7 Дескрипторы . . . 181
3.16 Статическая и динамическая компоновка . . . 182
3.16.1 LOCAL, GLOBAL, WEAK и неопределённый символ . . . 182
3.16.2 Relocatable link . . . 182
3.16.3 Статический архив `.a` . . . 183
3.16.4 Динамическая библиотека ET_DYN . . . 183
3.16.5 Область символов и interposition . . . 183
3.16.6 Ограничения ET_DYN . . . 183
3.16.7 Инструменты проверки . . . 183
3.17 Программные модели . . . 184
3.17.1 Целые, вещественные и комплексные вычисления LibMPU . . . 184
3.17.2 Массивы численных данных и вычислительные циклы . . . 184
3.17.3 Подпрограмма численного метода . . . 185
3.17.4 PROGRAM и MODEL как две модели выполнения . . . 185
3.17.5 Модель физического маятника . . . 186
Справочник по языку ассемблера . . . 186
Литература . . . 198

3. Язык ассемблера MCPU

3.1. Программная модель MCPU

Язык ассемблера имеет смысл только тогда, когда программист понимает машину, для которой пишет программу. На языке высокого уровня многие детали можно на время забыть: где именно находится промежуточное значение, каким образом функция получает параметры и что физически представляет собой указатель. На ассемблере эти вопросы становятся частью программы. Поэтому прежде, чем рассматривать отдельные инструкции, построим в голове цельный образ MCPU как виртуального процессора.

MCPU не пытается копировать конкретный микропроцессор. Его архитектура предназначена для воспроизводимого выполнения вычислительных программ и программных моделей поверх LibMPU. Тем не менее с точки зрения программиста перед нами вполне обычная вычислительная машина: у неё есть исполняемый код, регистровый файл, специальные регистры состояния, стек, статические и динамические данные. Отличие состоит в том, что все эти части определены явно и не зависят от случайных свойств host-процессора.

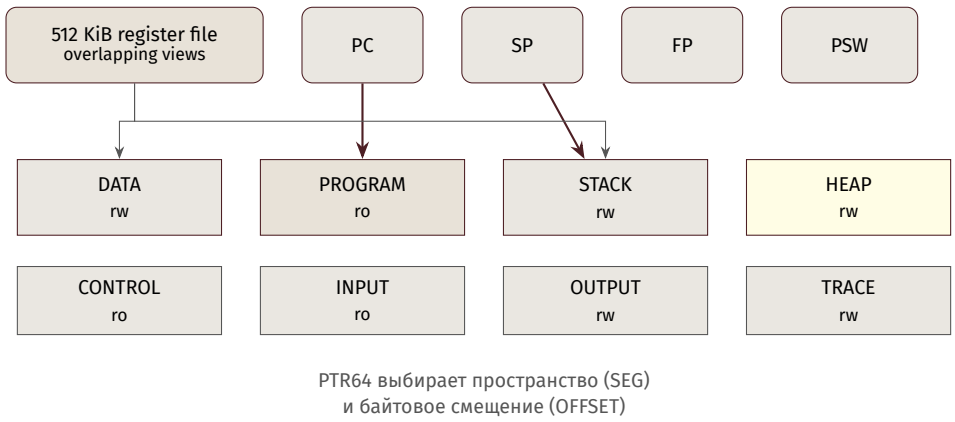


Рис. 3.1. MCPU глазами программиста: регистровый файл, специальное состояние процессора и независимые адресные пространства.

На рисунке важно видеть не набор отдельных прямоугольников, а три уровня машины. Вверху находится *состояние процессора*: PC, SP, FP, PSW и большой регистровый файл. Слева внизу находится PROGRAM — исполняемый код. Остальные пространства содержат данные различного назначения. Доступ к ним осуществляется не через единое плоское адресное пространство, а через архитектурные указатели PTR64, в которых явно записано, к какому пространству относится адрес.

3.1.1. Регистровый файл и перекрывающиеся представления

Регистровый файл MCPU имеет размер 512 KiB. Это не набор пронумерованных регистров вида `r0`, `r1`, `r2`. Регистр задаётся двумя характеристиками: **физической шириной представления** и **байтовым адресом** внутри общего регистрового файла. Поэтому запись

```
r64@0x00100
```

означает: взять 64-битное представление, начинающееся с байта `0x00100` регистрового файла.

Адресная часть регистра является обычной числовой константой языка ассемблера. Её можно записывать в десятичной, восьмеричной или шестнадцатеричной системе счисления. Поэтому операнды

```
r6416
r64@020
r64@0x10
```

обозначают один и тот же 64-битный регистр, начинающийся с байта 16 регистрового файла. Здесь `16` — десятичная запись, `020` — восьмеричная, а `0x10` — шестнадцатеричная. Число после символа `@` — это именно **байтовый адрес начала представления** внутри register file, а не порядковый номер регистра. Поэтому запись `r64@0x10` следует читать как “64-битное представление, начинающееся с байта 0x10”, а не как “регистр номер 0x10”. При чтении программы полезно сразу отделять ширину представления от адреса: во всех трёх случаях ширина равна 64 битам, а меняется только форма записи одного и того же байтового адреса.

В машинном коде такой операнд кодируется полем REG24. Старшие четыре бита REG24 задают физическую ширину, а оставшиеся двадцать — байтовый адрес начала представления. Для программиста важнее другое следствие: разные представления могут смотреть на одни и те же физические байты.

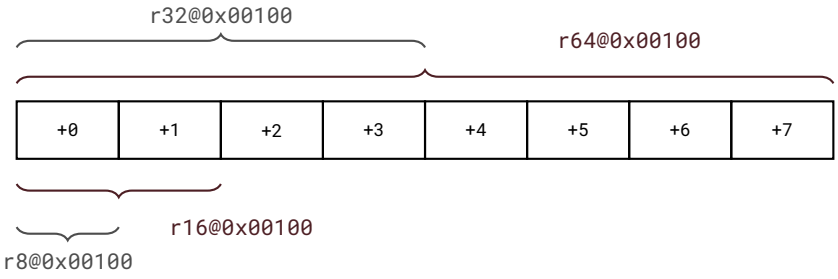


Рис. 3.2. Перекрытие представлений в едином регистровом файле.

Например, `r64@0x00100` занимает восемь байтов с адресами `0x00100...0x00107`. Представления `r32@0x00100`, `r16@0x00102` или `r8@0x00103` не создают новые места хранения. Они лишь дают другой взгляд на часть тех же физических байтов. Запись через узкое представление немедленно меняет содержимое соответствующей части более широкого представления.

Это свойство надо запомнить с самого начала. Имя регистра в MCPU описывает *view* — представление участка register file, а не автономную ячейку. Перекрытие является обычным архитектурным механизмом и не зависит от языка программирования.

Как уже отмечалось в главе 1, register file относится к домену архитектурных объектов данных и использует target representation LibMPU. Это другой byte-order domain, чем PROGRAM: машинные поля инструкций в PROGRAM имеют каноническую сериализацию, тогда как значения в register file и сегментах данных хранятся в целевом представлении MCPU/LibMPU. Поэтому пе-

рекрывающиеся регистровые представления описывают одни и те же физические байты, а их числовая интерпретация учитывает установленный для MCPU порядок байтов. Здесь мы лишь фиксируем это правило; подробнее порядок байтов уже был разобран во введении.

MCPU допускает очень широкие физические представления — от 8 до 65536 бит. В первых примерах нам понадобятся в основном `r8`, `r16`, `r32` и `r64`; широкие целые, вещественные и комплексные значения мы рассмотрим позднее, когда перейдём к типам данных и операциям LibMPU.

3.1.2. Абсолютные символы .equ и имена регистров

Постоянно писать в программе физические адреса register file вроде `0x00100`, `0x00108` и `0x00110` неудобно. Уже на уровне исходного текста ассемблера этим числам можно дать осмысленные имена. Для этого служит директива `.equ`:

```
.equ x, 0x00100
.equ a, x + 8
.equ b, a + 8
```

Директива `.equ name, expression` определяет **абсолютный символ ассемблера**. Такой символ обозначает целочисленное значение, вычисляемое во время ассемблирования. Он не создаёт объект в DATA, не занимает байты PROGRAM и не резервирует место в register file. В выражении можно использовать числовые константы, арифметические операции и уже определённые абсолютные символы. В данном примере после вычисления выражений получаются значения `x = 0x00100`, `a = 0x00108` и `b = 0x00110`.

Адресная часть REG24 допускает целочисленное выражение, поэтому абсолютный символ можно непосредственно использовать вместо числового адреса регистра:

```
r64@x
r64@a
r64@b
```

Эти записи обозначают те же физические представления register file, что и `r64@0x00100`, `r64@0x00108` и `r64@0x00110`. Никаких новых архитектурных “регистров x, a и b” директива `.equ` не создаёт: имена всего лишь делают байтовые адреса представлений читаемыми. Если изменить определение `x`, изменится и адрес всех выражений, которые от него зависят.

Именно поэтому в более длинных программах удобно сначала распределить рабочие области register file символически, например:

```
.equ left,   0x00100
.equ right,  left + 8
.equ result, right + 8
```

а затем пользоваться операндами `r64@left`, `r64@right` и `r64@result`. Такой стиль отделяет *на-*

значение регистра от его физического адреса и особенно удобен для кода, который позднее будет генерировать компилятор. При этом все правила перекрытия остаются прежними: если два выражения задают пересекающиеся диапазоны байтов, соответствующие register views действительно перекрываются. `.equ` не вводит никакой дополнительной защиты.

Абсолютный символ `.equ` важно не путать с меткой секции. `.equ` является именем вычисленного числа, тогда как метка вроде `object:` или `loop:` привязана к позиции в секции и может участвовать в relocatable-адресации. К этому различию мы вернёмся в разделе 3.6, когда начнём создавать реальные объекты DATA.

3.1.3. PC, SP, FP и PSW

Кроме обычного регистрового файла MCPU имеет специальное состояние процессора. Эти величины не являются представлениями `r64@...` и не занимают байты register file.

Имя	Физический смысл	Назначение
PC	64-битное смещение в PROGRAM	адрес следующей инструкции
SP	64-битное смещение в STACK	текущая граница живой области стека
FP	64-битное смещение в STACK	стабильная база кадра функции
PSW	32-битное слово состояния	флаги результата и состояние CPU

PC — **program counter** (*счётчик команд*). В нём находится байтовое смещение следующей выполняемой инструкции внутри PROGRAM. SP — **stack pointer** (*указатель стека*); он будет подробно рассмотрен в разделе 3.3. FP — **frame pointer** (*указатель кадра*); он становится особенно полезен при работе с локальными объектами и вложенными вызовами функций.

PSW — **processor status word** (*слово состояния процессора*). Арифметические и логические операции устанавливают в нём признаки результата, а условные переходы читают эти признаки. В младшей части PSW находятся, в частности, CF, OF, SF, ZF, а также состояния вещественного и математического доменов LibMPU. Биты HF, TF и IF относятся к архитектурному состоянию HALT, трассировки/ловушек и разрешения прерываний. Полную таблицу PSW мы введём тогда, когда начнём пользоваться сравнениями и условными переходами. Сейчас достаточно запомнить главное: результат вычисления может изменить не только регистр-приёмник, но и состояние PSW.

3.1.4. Независимые адресные пространства

У MCPU нет единственной безразмерной “памяти вообще”. Архитектура разделяет несколько независимых пространств. Каждое из них байтово адресуемо, но имеет собственное назначение и собственные права доступа guest-программы.

SEG	Пространство	Назначение	Guest-доступ
0x0	DATA	статические и глобальные данные	R/W
0x1	STACK	вызовы, аргументы, локальные объекты	R/W
0x2	PROGRAM	машинный код	R + execute
0x3	CONTROL	параметры управления запуском	R
0x4	INPUT	внешние входные воздействия	R
0x5	OUTPUT	выходные данные модели	R/W
0x6	TRACE	текущая трассировочная запись	R/W
0x7	HEAP	динамическая память runtime	R/W

DATA и HEAP намеренно различаются. DATA формируется компоновщиком как часть исполняемого образа. HEAP возникает во время выполнения и предназначен для динамической памяти. Позднее поверх HEAP могут быть построены привычные `malloc()`, `free()` и другие функции библиотеки, но сам процессор видит лишь отдельное адресное пространство и текущую границу program break.

CONTROL, INPUT, OUTPUT и TRACE особенно важны для программных моделей. CPU не приписывает байтам внутри этих пространств физический смысл: структуру записей задаёт программа и будущая среда языка моделирования. Однако права доступа уже являются частью архитектуры: например, guest-программа может читать INPUT, но не должна самовольно переписывать входное воздействие, пришедшее от supervisor-а.

3.1.5. PTR64: адрес состоит из сегмента и смещения

Для обращения к объектам разных пространств MCPU использует 64-битный архитектурный указатель PTR64. Это **не host pointer**. Старшие четыре бита задают selector адресного пространства, младшие 60 бит — байтовое смещение внутри него.

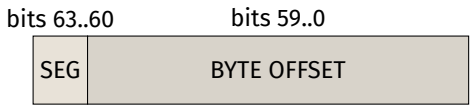


Рис. 3.3. Логическая структура PTR64.

Таким образом, адрес следует мысленно читать не как одно число, а как пару `SEG:OFFSET`.

Например, DATA:0x100, STACK:0x100 и HEAP:0x100 имеют одинаковое смещение `0x100`, но указывают в три совершенно разных пространства. Именно selector делает их различными адресами.

Нулевой указатель имеет единственное представление — все 64 бита равны нулю. Битово это совпадает с `DATA:0`. По этой причине начало *финального* DATA процесса нельзя использовать для размещения обычных данных: первые 16 байтов, то есть диапазон смещений `0...15`, образуют защитную область (**guard area** (*защитная область*)). Она отделяет нулевой указатель от первого обычного объекта DATA: даже ошибочный адрес около нуля не совпадает с адресом первой переменной программы.

Здесь важно различать два уровня представления программы. В relocatable-объекте ET_REL

вклад секции `.data` ещё не содержит этой защитной области. Его первый объект вполне может иметь section-relative offset 0. Защитную область создаёт компоновщик при построении конечного исполняемого образа: в итоговом DATA процесса он резервирует один диапазон `0...15`, а затем размещает за ним объединённые DATA-вклады входных объектов с сохранением требуемого выравнивания.

Поэтому исходный текст

```
.data
.align8
        .int 64, 0x37
```

следует мысленно рассматривать в два этапа. В соответствующем ET_REL этот объект может находиться в начале собственной секции `.data`, то есть иметь section-relative offset 0. После окончательной компоновки в ET_EXEC перед ним появляется linker-owned guard area размером 16 байтов. Если перед этим объектом нет других DATA-вкладов, а его выравнивание равно восьми байтам, то в process DATA он окажется по смещению 16:

```
ET_REL .data:             ET_EXEC / process DATA:

+0   0x37 object          +0 ... +15   NULL guard
                          +16          0x37 object
```

Именно process offset 16 видит выполняющаяся программа и именно его использует `mcpu-run`. Поэтому после запуска значение можно непосредственно посмотреть снаружи:

```
mcpu-run --int-data=64@16 -- program
```

Запись `64@16` в этой опции означает: прочитать из **DATA процесса** 64-битное целое значение, начинающееся с байтового смещения 16. В нашем примере там находится записанное директивой `.int` значение `0x37` (десятичное 55). Если же исследовать промежуточный ET_REL с помощью `mcpu-readelf` или `mcpu-objdump`, там этот же объект может иметь section-relative offset 0 — это не противоречие, а следствие того, что linker-owned guard area возникает только при формировании окончательного DATA исполняемого файла.

Такой способ наблюдения памяти будет особенно удобен в первых учебных программах: guest-коду не требуется ничего печатать, а состояние DATA можно проверить после завершения выполнения средствами `mcpu-run`.

У HEAP также существует низкая защитная область. Однако здесь важно не смешивать понятие защитной области с NULL: например, `PTR64(HEAP,0)` не является нулевым указателем, потому что его selector уже отличен от нуля. Единственным представлением NULL остаётся полностью нулевое 64-битное значение, то есть битовый образ `DATA:0`.

3.1.6. Непосредственные данные в программе

До сих пор мы говорили о том, *где* находятся данные: в register file, DATA, STACK или другом адресном пространстве. Но небольшая числовая константа может вообще не иметь отдельного объекта памяти. Она может быть записана непосредственно в машинной инструкции. Такой операнд называется **immediate** (*непосредственным*): его значение является частью байтов самой инструкции в PROGRAM.

Это различие принципиально. Например, в записи

```
LDI     r64@0x00100, 37
```

число `37` не читается из DATA и не находится в register file до выполнения инструкции. Оно закодировано непосредственно после opcode и REG24 внутри PROGRAM. Для числовых immediate-форм MCPU используется 64-битное поле данных, однако смысл этих 64 бит зависит от инструкции. Важнейшее правило ассемблера состоит в том, что такой компактный формат **не должен молча терять точность исходной константы**.

В MCPU имеются три основные инструкции загрузки непосредственных числовых значений:

Инструкция	Immediate в PROGRAM	Назначение
`LDI` `R, integer`	signed Integer64	целое значение
`LDR` `R, real`	binary64	вещественное значение
`LDC` `C, complex`	2 × binary32	комплексное значение

Целая константа: LDI.

У `LDI` источник имеет формат signed Integer64, то есть 64-битное целое в дополнительном коде. Целочисленный immediate разбирается как обычное целочисленное выражение ассемблера, поэтому литералы можно записывать в десятичной, восьмеричной и шестнадцатеричной формах: например, `16`, `020` и `0x10` обозначают одно и то же значение. Если регистр-приёмник имеет ширину 64 бита или больше, исходный литерал обязан точно помещаться в диапазон signed Integer64. Для более узкого регистра требование ещё строже: значение должно точно помещаться в signed диапазон самого регистра-приёмника.

Поэтому следующие инструкции корректны:

```
LDI     r8@0x00000, -1
LDI     r32@0x00010, 1000000
LDI     r4096@0x00100, 9223372036854775807
```

а попытка записать `255` в signed 8-битное представление отвергается:

```
LDI     r8@0x00000, 255
```

Точно так же увеличение физической ширины регистра не увеличивает ширину immediate-поля инструкции. Например,

```
LDI     r128@0x00100, 9223372036854775808
```

недопустима: значение уже не помещается в signed Integer64, хотя сам `r128` способен его хранить. Для такого числа надо создать полноразмерный объект DATA и загрузить его обычной инструкцией `LOAD`:

```
.data
.align 16
wide_integer:
        .integer 128, 9223372036854775808
.text
        LOAD    r128@0x00100, wide_integer
```

При выполнении корректной `LDI` CPU рассматривает встроенное Integer64 как исходное значение и преобразует его к физической ширине регистра средствами LibMPU. Проверка ассемблера гарантирует, что это преобразование не уничтожит значение.

Вещественная константа: LDR.

У `LDR` машинное поле непосредственного операнда всегда имеет формат binary64. Но правило точности зависит от ширины регистра-приёмника. Для `r32` исходная десятичная запись должна быть **точно** представима как real32. Для `r64` и любого более широкого вещественного представления она должна быть точно представима как real64.

Здесь особенно важно не смешивать два разных вопроса: *попадает ли число в диапазон данного вещественного формата* и *может ли этот формат представить число точно*. Обычно программист сначала думает именно о диапазоне: не слишком ли велико или не слишком ли мало значение. Но для immediate-операнда MCPU этого недостаточно. Число может без всякого переполнения помещаться в диапазон real32 и всё же требовать округления из-за ограниченной точности значащей части.

Хороший пример — десятичное число `0.1`. Оно, разумеется, находится далеко внутри диапазона real32 и real64 и может храниться в обоих форматах. Однако математически `0.1 = 1/10`, а дробь с множителем 5 в знаменателе не имеет конечного представления в системе счисления с основанием 2. Поэтому в binary32 или binary64 хранится не точное число `0.1`, а ближайшее двоичное приближение к нему. Напротив, значения `1.0`, `2.0`, `0.5`, `0.25`, `0.125` или `32.0` представляются двоично точно: они выражаются конечной суммой степеней двойки.

Точность binary32 составляет 24 значащих двоичных бита с учётом неявной старшей единицы нормализованного числа; для binary64 — 53 бита. Поэтому даже число, совершенно безопасное по диапазону, может не поместиться *точно* в значащую часть. Именно это и проверяет `mcpu-as`. Для immediate-формы вопрос звучит не “можно ли приблизительно сохранить это число?”, а “сохранится ли исходное математическое значение без изменения?”.

Например,

```
LDR     r32@0x02000, -17.0
LDR     r4096@0x04000, 2.0
```

разрешены: `-17.0` и `2.0` имеют точное двоичное представление. Для сравнения,

```
LDR     r32@0x02000, 1.0     ; представляется точно
LDR     r32@0x02004, 0.1     ; требует округления -> ошибка assembler-a
```

Вторая инструкция будет отвергнута не потому, что `0.1` “не влезает” в real32, а потому, что exact-immediate rule запрещает молчаливо заменять исходное десятичное значение его ближайшим binary32-приближением. Аналогично, высокоточная десятичная запись

```
LDR     r4096@0x04000, 3.14243987129849274029174912694696912849214e-286
```

не может быть сначала незаметно округлена до binary64 только потому, что регистр-приёмник имеет тысячи бит. В PROGRAM всё равно имеется лишь 64-битное поле непосредственного операнда. В таком случае константу надо сформировать непосредственно с требуемой точностью в DATA:

```
.data
.align 16
wide_real:
        .real 4096, 3.14243987129849274029174912694696912849214e-286
.text
        LOAD    r4096@0x04000, wide_real
```

Директива `.real` передаёт текстовое значение LibMPU непосредственно для построения объекта объявленной ширины; промежуточного binary64 здесь нет.

Комплексная константа: LDC и представления c<bits>.

Для комплексных значений MCPU использует обозначение `c<bits>@address`. Число `bits` задаёт **полную физическую ширину комплексного представления**, а не ширину одной компоненты. Поэтому минимальное представление `c64` состоит из двух 32-битных вещественных компонент, а `c128` — из двух 64-битных компонент.

Непосредственная инструкция `LDC` всегда содержит ровно две binary32-компоненты, то есть всего 64 бита непосредственных данных:

```
LDC     c64@0x02000, -17.0r0+32.0j0
LDC     c4096@0x04000, 2.0r0-0.5j0
```

Обе текстовые компоненты обязаны быть точно представимы как real32 независимо от физической ширины регистра-приёмника. Проверка выполняется для каждой компоненты незави-

симо. Поэтому здесь снова важно не говорить просто “влезает” или “не влезает”. Например, `32.0` прекрасно помещается в real32 и, более того, представляется в нём *точно*; проблема в следующем примере возникает только из-за компоненты `0.1`:

```
LDC     c128@0x02000, 1.0r0+32.0j0   ; обе компоненты точные
LDC     c128@0x02010, 0.1r0+32.0j0   ; real-компонента требует округления
```

После декодирования CPU расширяет две уже точно представленные real32-компоненты средствами LibMPU до половины физической ширины выбранного c-представления и из них формирует комплексное значение. Вторая инструкция выше отвергается ассемблером: первая, вещественная, компонента `0.1` не представима точно как real32, тогда как мнимая компонента `32.0` полностью допустима. Представления `c32` вообще не существует, поскольку минимальный комплексный регистр-приёмник — `c64`.

Для высокоточной комплексной константы действует тот же принцип, что и для широкого integer или real: полноценный объект надо создать в DATA и затем загрузить:

```
.data
.align 16
wide_complex:
        .complex 4096, 3.14243987129849274029174912694696912849214e-286, -0.125
.text
        LOAD    c4096@0x04000, wide_complex
```

Почему потеря точности считается ошибкой ассемблера.

После сборки PROGRAM уже не содержит исходную десятичную строку. Если бы ассемблер сначала округлил её до binary64 или binary32, процессор при выполнении инструкции уже не смог бы узнать, что часть информации была потеряна. Поэтому `mcpu-as` проверяет точную представимость до формирования машинного кода: текстовый литерал разбирается LibMPU с высокой точностью, преобразуется к формату immediate, затем расширяется обратно и сравнивается с исходным значением. В машинный код попадает только значение, для которого такое прямое и обратное преобразование не изменило число.

Типичная диагностика прямо подсказывает правильный выход:

```
mcpu-as: error:
line N: LDR immediate is not exactly representable as real64;
use .real <bits>, value in DATA and LOAD
```

То есть непосредственная форма предназначена не для “любого числа, которое потом как-нибудь округлится”, а только для числа, которое машинный формат непосредственного операнда способен сохранить без потерь. Всё, что требует большей точности, должно существовать как полноценный DATA-объект.

Непосредственное значение в PROGRAM не имеет порядка байтов host-машины.

Наконец, нельзя путать непосредственный операнд с объектом данных той же числовой величины. Поля машинной инструкции — OPCODE16, REG24 и 64-битное поле непосредственного операнда — принадлежат домену PROGRAM и сериализуются канонически MSB-first. Это правило не зависит от порядка байтов компьютера, на котором запущены `mcpu-as` или `mcpu-run`. Например, 64-битный непосредственный операнд `0x1122334455667788` находится в PROGRAM в порядке байтов

```
11 22 33 44 55 66 77 88
```

а 64-битный объект с тем же числовым значением в register file или DATA хранится в целевом представлении MCPU/LibMPU. В текущем little-endian целевом профиле его байты имеют порядок

```
88 77 66 55 44 33 22 11
```

Это два разных уровня архитектуры: PROGRAM представляет поток машинных инструкций, а register file и сегменты данных — объекты вычислений. Никакой неявной зависимости формата непосредственного операнда от host-процессора нет.

Внутренние преобразования `LDI`, `LDR` и `LDC` относятся к conversion-class и сохраняют status-часть PSW: загрузка константы сама по себе не должна уничтожать признаки результата, оставленные предыдущей вычислительной инструкцией.

3.1.7. Прямая и косвенная адресация

На первых шагах полезно различать два способа обращения к данным. Инструкции `LOAD/STORE` используют непосредственное 64-битное смещение внутри DATA. Такое значение называется ADDR64 и само по себе не является PTR64.

```
LOAD    r64@0x00100, global_value
STORE   global_value, r64@0x00100
```

Косвенные инструкции `LOADP/STOREP`, напротив, получают полный PTR64 из физического `r64`. Указатель можно сформировать инструкцией `LEA` по имени объекта, а для объектов стека существуют `LEASP` и `LEAFP`.

```
LEA     r64@0x00100, global_value
LOADP   r64@0x00108, r64@0x00100
```

По одному внешнему виду 64-битного числа нельзя забывать, что оно представляет адрес. CPU проверяет selector и разрешённость операции. Например, PROGRAM можно читать через `LOADP`, но запись через `STOREP` туда запрещена; INPUT и CONTROL также доступны guest-коду только для чтения.

3.1.8. Почему нельзя упорядочивать указатели из разных сегментов

Равенство PTR64 определить просто: для `==` или `!=` сравниваются все 64 бит. Поэтому DATA:0x100 и STACK:0x100 не равны.

С отношениями “меньше” и “больше” ситуация принципиально иная. Положение одного объекта относительно другого имеет смысл только внутри одного адресного пространства. Численно полное 64-битное значение DATA:0x100 может оказаться меньше или больше STACK:0x20, но архитектура не придаёт такому сравнению смысла порядка указателей.

Для вычисления расстояния между двумя PTR64 служит `PDIFF`. Эта операция сначала требует совпадения selector-ов и лишь затем вычитает 60-битные смещения. Если указатели принадлежат разным сегментам, возникает `CPU_EXEC_POINTER_FAULT`.

```
; p и q должны принадлежать одному SEG
PDIFF   r64@0x00120, r64@0x00100, r64@0x00108
```

Такое правило хорошо согласуется с привычной моделью языка C: сравнение порядка имеет содержательный смысл только для адресов, относящихся к одной области объектов. В MCPU это свойство дополнительно видно прямо в формате указателя.

3.1.9. Что должен запомнить программист

К этому месту нам ещё не нужно помнить все инструкции MCPU. Достаточно удерживать цельный образ машины. Есть единый 512-KiB register file с перекрывающимися представлениями. PC, SP, FP и PSW являются отдельным специальным состоянием. Исполняемый код живёт в PROGRAM, данные разделены на независимые пространства, а PTR64 несёт одновременно selector пространства и байтовое смещение. В следующем разделе мы впервые изменим это состояние программой и увидим результат не через печать текста, а непосредственно глазами `mcpu-run`.

3.2. Первые инструкции и наблюдение состояния процессора

Первая программа на новом ассемблере обычно печатает строку “Hello, world!”. Для MCPU полезнее начать ещё проще. Пока нам не нужны строки, системные вызовы и даже память DATA. Мы изменим несколько регистров, остановим процессор и попросим `mcpu-run` показать результат. Такой эксперимент связывает исходный текст непосредственно с программной моделью из предыдущего раздела.

3.2.1. Минимальная программа

Исходный файл `register-state.s` имеет вид:

```
; Первая программа главы: ничего не печатает, только изменяет register file.

```

```
.text
_start_life:
        LDI     r8@0x00000,   12
        LDI     r16@0x00010,  1234
        LDI     r32@0x00020,  123456
        LDI     r64@0x00028,  123456789
        HALT
```

Псевдометка `_start_life:` определяет сразу две нормативные точки входа — `_start` и `_life` — по одному адресу в секции `.text`. Для простой низкоуровневой программы без отдельной `.init` это удобная форма записи. После загрузки управление начинается именно здесь.

Инструкция `LDI` загружает непосредственную целую константу в указанное представление register file. В примере мы намеренно выбираем разные адреса, чтобы значения не перекрывались:

```
LDI     r8@0x00000,   12
LDI     r16@0x00010,  1234
LDI     r32@0x00020,  123456
LDI     r64@0x00028,  123456789
```

После выполнения `HALT` guest больше ничего не делает, но состояние процессора остаётся доступно runner-у для диагностики.

3.2.2. Сборка

В архиве примеров к книге этот пример находится в отдельном каталоге `register-state/`. Его Makefile выполняет уже знакомую нам цепочку preprocessing, assembler и linker:

```
.SUFFIXES:

MCPU_CPP = mcpu-cpp
MCPU_AS  = mcpu-as
MCPU_LD  = mcpu-ld
MCPU_RUN = mcpu-run

PROGRAM = register-state

all: $(PROGRAM)

$(PROGRAM).S: $(PROGRAM).s
        $(MCPU_CPP) -o $@ $<

$(PROGRAM).o: $(PROGRAM).S
        $(MCPU_AS) -o $@ $<

$(PROGRAM): $(PROGRAM).o
        $(MCPU_LD) --stack-size=512 --no-heap -o $@ $<
```

```
run: $(PROGRAM)
        $(MCPU_RUN) \
          --after-run=print-int-regs:r8@0x00000,r16@0x00010,r32@0x00020,r64@0x00028 \
          -- $(PROGRAM)

clean:
        rm -f $(PROGRAM).S $(PROGRAM).o $(PROGRAM)
```

Команда

```
make
```

создаёт исполняемый MELF-файл `register-state`. Теперь вместо вывода guest-программы попросим сам `mcpu-run` прочитать интересующие нас регистры после остановки:

```
mcpu-run \
  --int-reg=r8@0x00000 \
  --int-reg=r16@0x00010 \
  --int-reg=r32@0x00020 \
  --int-reg=r64@0x00028 \
  -- register-state
```

Мы увидим четыре числа, которые только что поместили в register file. Важно почувствовать разницу: программа сама ничего не печатала. Вывод сделал диагностический слой `mcpu-run`, уже после того как guest изменил состояние виртуального процессора.

3.2.3. Перекрытие можно увидеть тем же способом

Следующий небольшой опыт связывает программу с рисунком перекрывающихся представлений. Сначала обнулим широкий регистр, затем изменим один из байтов внутри него:

```
LDI     r64@0x00100, 0
LDI     r8@0x00103,  42
HALT
```

`r8@0x00103` физически принадлежит диапазону байтов `r64@0x00100`. Поэтому широкое представление после второй инструкции уже не состоит из одних нулей. Конкретное числовое значение широкого integer-view зависит от целевого представления многобайтовых объектов, но факт изменения одних и тех же физических байтов от порядка байтов не зависит.

При изучении MCPU это полезный общий приём: сначала сформулировать, какие байты или состояния должна изменить инструкция, затем выполнить минимальную программу и посмотреть на результат через диагностические средства runner-а. Мы будем пользоваться этим подходом и дальше.

3.2.4. PSW пока остаётся рядом

Не всякая инструкция меняет PSW. Например, `LDI` относится к PSW-neutral операциям: загрузка константы не должна уничтожать признаки, полученные предыдущим вычислением. Арифметические инструкции, напротив, формируют flags. Позднее это позволит писать условные переходы без отдельного “булева регистра” для каждого сравнения.

Пока достаточно помнить, что register file и PSW — разные части состояния CPU. Опция диагностики регистра показывает выбранное представление register file; условия переходов будут читать соответствующие биты PSW. В следующих разделах сначала разберём STACK и механизм функций, а затем вернёмся к PSW уже в контексте `if`, `while` и других управляющих конструкций.

3.3. Стек

До сих пор все рабочие значения жили в register file. Этого достаточно для короткого линейного фрагмента, но недостаточно для функций, локальных объектов и вложенных вычислений. Для них MCPU имеет отдельное адресное пространство STACK.

3.3.1. Как растёт STACK

STACK растёт **к меньшим адресам**. После создания пустого стека его верхняя свободная граница находится у конца выделенной области. Начальное специальное состояние равно

```
SP = FP = stack_size.
```

Когда программа резервирует место, SP уменьшается. При освобождении места SP увеличивается. Никакой скрытой “красной зоны” ниже текущего SP в ABI MCPU нет: прежде чем пользоваться байтами, программа обязана явно включить их в живую область стека.



Рис. 3.4. STACK растёт к меньшим адресам; SP отмечает границу живой области.

SP хранит не PTR64, а обычное 64-битное байтовое смещение внутри STACK. Если программе нужен настоящий указатель на объект стека, она формирует PTR64(STACK,offset) инструкцией `LEASP` или, позднее, `LEAFP`.

3.3.2. PUSH и POP

Инструкция **PUSH** R сначала уменьшает SP на размер физического представления регистра, а затем копирует байты регистра в новую вершину стека. **POP** R выполняет обратное действие: читает значение из текущей вершины и после чтения увеличивает SP.

Инструкция	Изменение SP	Смысл
PUSH R	SP -= sizeof(R)	положить значение на вершину
POP R	SP += sizeof(R)	снять значение с вершины
LDSP R, off	SP не меняется	прочитать STACK[SP+off]
STSP off, R	SP не меняется	записать значение R в STACK[SP+off]
ADJSP off	SP += off	явно изменить размер живой области

Отсутствие автоматического padding здесь принципиально. CPU выполняет ровно указанное изменение SP. Если ABI функции требует 16-байтового выравнивания, это обеспечивает программа, assembler или compiler, а не скрытая логика PUSH.

3.3.3. Первый эксперимент со стеком

Возьмём два 64-битных значения. Сначала положим в STACK число 11, затем число 22. После второго PUSH текущая вершина содержит 22, а первое значение находится по адресу SP+8.

```
; Простейшие операции со стеком.

.text
_start_life:
        LDI     r64@0x00000, 11
        LDI     r64@0x00008, 22

        ; STACK растёт вниз. После второго PUSH вершина содержит 22,
        ; а первое значение 11 находится по SP+8.
        PUSH    r64@0x00000
        PUSH    r64@0x00008
        LDSP    r64@0x00010, 8

        ; POP снимает значения в обратном порядке.
        POP     r64@0x00018
        POP     r64@0x00020

        ; Явно резервируем 16 байтов локальной области.
        ADJSP   -16
        STSP    0, r64@0x00008
        LDSP    r64@0x00028, 0
        ADJSP   16

        HALT
```

После двух PUSH стек можно мысленно представить так:

Адрес относительно SP	Содержимое
SP+0	22
SP+8	11

LDSP r64@0x00010, 8 читает 11, не меняя SP. Затем два POP снимают значения в обратном порядке. Это обычная дисциплина LIFO — **last in, first out** (*последним вошёл, первым вышел*).

Во второй части примера **ADJSP** -16 резервирует 16 байтов без помещения какого-либо значения. После этого **STSP** 0, ... и **LDSP** ..., 0 используют выделенную область как простой локальный объект. Симметричный **ADJSP** 16 освобождает её.

Собрать и проверить пример можно командами:

```
make
make run
```

Makefile находится рядом с исходником:

```
.SUFFIXES:

MCPU_CPP = mcpu-cpp
MCPU_AS  = mcpu-as
MCPU_LD  = mcpu-ld
MCPU_RUN = mcpu-run
PROGRAM  = stack

all: $(PROGRAM)

$(PROGRAM).S: $(PROGRAM).s
        $(MCPU_CPP) -o $@ $<
$(PROGRAM).o: $(PROGRAM).S
        $(MCPU_AS) -o $@ $<
$(PROGRAM): $(PROGRAM).o
        $(MCPU_LD) --stack-size=512 --no-heap -o $@ $<
run: $(PROGRAM)
        $(MCPU_RUN) --after-run=print-int-regs:r64@0x00010,r64@0x00028 -- $(PROGRAM)
clean:
        rm -f $(PROGRAM).S $(PROGRAM).o $(PROGRAM)
```

3.3.4. SP — это граница живой памяти

Полезно не воспринимать SP лишь как число, которое иногда увеличивается или уменьшается. SP определяет границу области STACK, которой программа в данный момент владеет. Если выполнить **ADJSP** -32, программа получила 32 байта. Если затем выполнить **ADJSP** 32, эти байты больше не принадлежат текущему кадру, даже если физически старые значения ещё какое-то время остаются в backing memory.

Именно поэтому PTR64 на освобождённый локальный объект семантически становится недействительным. Аппаратная проверка диапазона не обязана знать, какой именно функции

когда-то принадлежали эти байты; владение стековым объектом задаётся дисциплиной ABI.

На следующем шаге стек перестанет быть просто временным хранилищем. Мы увидим, что CALL использует его для адреса возврата, а вызывающая программа размещает там параметры функции.

3.4. Подпрограммы и соглашение о вызовах

Как только понятны STACK и SP, вызов функции перестаёт быть магией. CALL — это переход к другому участку PROGRAM с одним важным дополнительным действием: процессор сохраняет на стеке адрес, с которого выполнение должно продолжиться после RET.

3.4.1. Что делает CALL

Перед выполнением CALL вызывающая программа уже должна подготовить блок аргументов. Затем процессор вычисляет адрес следующей инструкции, уменьшает SP на 8 и записывает туда 64-битный return PC. После этого PC получает адрес вызываемой функции.

Непосредственно при входе в вызываемую функцию стек имеет следующий вид:

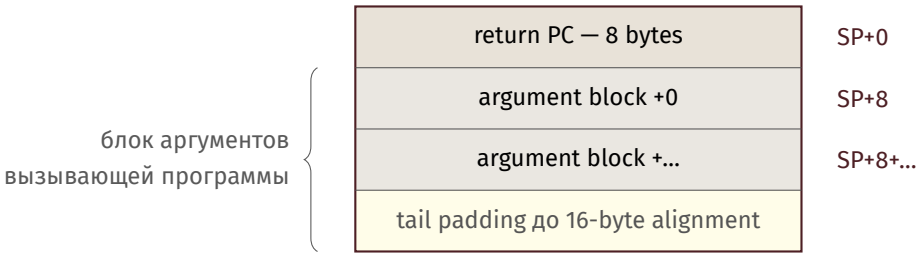


Рис. 3.5. Состояние STACK непосредственно после CALL.

По `SP+0` находится адрес возврата. Начало блока аргументов находится по `SP+8`. RET снимает только return PC и восстанавливает PC. Сам блок аргументов остаётся собственностью вызывающей программы, поэтому после возврата именно она освобождает занятое им место.

Return PC является raw 64-битным смещением внутри PROGRAM, а не PTR64. Отдельного регистра RA или LR в MCPU нет.

3.4.2. Простейшая функция с двумя параметрами

Рассмотрим функцию

```
add2(a, b) = a + b
```

для двух 64-битных целых параметров. Два `r64` занимают ровно 16 байтов, поэтому простой вариант с PUSH справа налево одновременно удовлетворяет формальному требованию ABI о 16-байтовом блоке аргументов.

```
; CALL ABI: add2(a,b) возвращает a+b в каноническом r64 return view.

.text
_start_life:
        LDI     r64@0x00000, 11
        LDI     r64@0x00008, 22

        PUSH    r64@0x00008             ; b, второй параметр
        PUSH    r64@0x00000             ; a, первый параметр
        CALL    add2
        ADJSP   16                       ; вызывающая программа освобождает блок
    аргументов

        MOV     r64@0x00010, r64@0x7fff8
        HALT

add2:
        LDSP    r64@0x00100, 8           ; a
        LDSP    r64@0x00108, 16          ; b
        IADD    r64@0x7fff8, r64@0x00100, r64@0x00108
        RET
```

Caller выполняет:

```
PUSH    r64@0x00008     ; b
PUSH    r64@0x00000     ; a
CALL    add2
ADJSP   16              ; вызывающая программа освобождает аргументы
```

На входе `add2` расположение известно точно:

Адрес	Содержимое
SP+0	return PC
SP+8	первый параметр a
SP+16	второй параметр b

Поэтому вызываемая функция читает параметры через LDSP и складывает их.

3.4.3. Где находится возвращаемое значение

Скалярный результат функции передаётся не через стек. ABI резервирует верхние 8192 байта register file как **Return Value Area** (*область возвращаемого значения*). Конкретное представление выбирается так, чтобы его старший конец совпадал с концом register file.

Тип view	Размер	Каноническое место результата
r8	1 byte	r8@0x7ffff
r16	2 bytes	r16@0x7fffe
r32	4 bytes	r32@0x7fffc
r64	8 bytes	r64@0x7fff8

Именно поэтому add2 записывает сумму в r64@0x7fff8. После RET вызывающая программа при необходимости сразу копирует результат в собственный рабочий регистр. Содержимое Return Value Area не сохраняется через CALL: следующий вызов имеет право использовать эту область заново.

3.4.4. Один параметр всё равно требует ABI-размещения

Два 64-битных PUSH в предыдущем примере дали 16 байтов автоматически. Для одного r64 нельзя просто сделать один PUSH и считать блок аргументов завершённым: перед ABI-conforming CALL SP должен быть выровнен на 16 байтов, а размер блока округляется вверх до 16.

Поэтому общий вариант выглядит так:

```
ADJSP   -16
STSP    0, r64@0x00100
CALL    f
ADJSP   16
```

CPU не знает, что лежит внутри блока: целое число, вещественное число, указатель или часть структуры. Размещение параметров, выравнивающих промежутков (padding) и скрытых ячеек (slots) — это соглашение между вызывающей программой, вызываемой функцией и будущим компилятором языка высокого уровня.

3.4.5. Frame pointer: когда одного SP становится мало

Пока функция только читает параметры и сразу возвращается, обращаться к ним относительно SP удобно. Но функция сама может делать PUSH, резервировать локальные объекты, вызывать другие функции. Тогда SP постоянно движется, и один и тот же параметр оказывается по разным текущим смещениям от SP.

Для стабильной базы служит FP. Функция, решившая использовать frame pointer, начинает с канонического пролога:

```
PUSHFP
SETFP
ADJSP   -fixed_frame_size
```

После PUSHFP ; SETFP геометрия фиксируется:

Смещение от FP	Содержимое
FP+0	сохранённый FP вызывающей программы
FP+8	return PC
FP+16	начало блока аргументов
FP-...	локальные объекты и spills

Теперь SP может двигаться ниже FP, а параметры и фиксированные локальные объекты остаются по стабильным смещениям. Инструкции LDFP, STFP и LEAFP являются FP-relative аналогами операций с SP.

Канонический эпилог выполняет обратную последовательность:

```
RESTSP
POPFP
RET
```

CALL и RET сами FP не меняют. Использовать frame pointer или обойтись только SP решает функция и соглашение компилятора.

3.4.6. PSW и вызов функции

CALL и RET сами не обязаны уничтожать PSW, но вызываемая функция свободна выполнять арифметические операции и тем самым менять флаги. Поэтому арифметические и status-биты рассматриваются как состояние, которое вызываемая функция вправе изменить. Нельзя вычислить условие, вызвать произвольную функцию, а затем рассчитывать, что прежний ZF или CF всё ещё описывает результат до вызова.

Это правило станет особенно важным в разделе об условных операторах. Пока достаточно запомнить: через границу CALL нужно сохранять не только нужные рабочие значения, но и учитывать, какое состояние ABI считает volatile.

Сборка и запуск нашего примера:

```
make
make run
```

```
.SUFFIXES:

MCPU_CPP = mcpu-cpp
MCPU_AS  = mcpu-as
MCPU_LD  = mcpu-ld
MCPU_RUN = mcpu-run
PROGRAM  = call

all: $(PROGRAM)
```

```
$(PROGRAM).S: $(PROGRAM).s
        $(MCPU_CPP) -o $@ $<
$(PROGRAM).o: $(PROGRAM).S
        $(MCPU_AS) -o $@ $<
$(PROGRAM): $(PROGRAM).o
        $(MCPU_LD) --stack-size=512 --no-heap -o $@ $<
run: $(PROGRAM)
        $(MCPU_RUN) --int-reg=r64@0x00010 -- $(PROGRAM)
clean:
        rm -f $(PROGRAM).S $(PROGRAM).o $(PROGRAM)
```

3.5. Параметры командной строки

Теперь у нас есть всё необходимое, чтобы без скачка в рассуждении понять передачу параметров командной строки. Мы уже знаем STACK, умеем читать объекты по смещениям, знаем PTR64 и понимаем, что указатель на массив и указатель на строку являются обычными значениями, указывающими в конкретное адресное пространство.

3.5.1. Startup area создаёт loader

До выполнения первой guest-инструкции загрузчик (loader) строит в верхней части STACK постоянную область запуска `McpuStartup`. `McpuStartup` — это публичная структура ABI, объявленная в установленном заголовочном файле `/usr/include/mcpu-elf.h`. Она существует всё время жизни процесса и содержит не только `argc` и `argv`, но и служебные поля версии ABI.

Заголовок занимает 32 байта:

Смещение	Поле	Смысл
`+0x00`	`uint32 version`	версия startup ABI
`+0x04`	`uint32 size`	32; размер структуры
`+0x08`	`uint64 argc`	число аргументов
`+0x10`	`PTR64 argv`	массив указателей в STACK
`+0x18`	`uint64 reserved`	ноль

Сразу после заголовка располагается массив `argv[0..argc]` из PTR64. Последний элемент массива равен NULL. Ещё дальше находятся сами NUL-terminated строки.

Начальное состояние перед первой guest-инструкцией имеет важную особенность:

```
PC = e_entry,    SP = startup_base,    FP = stack_size.
```

То есть SP указывает непосредственно на заголовок `McpuStartup`. Искусственного CALL-frame для `_start` нет.

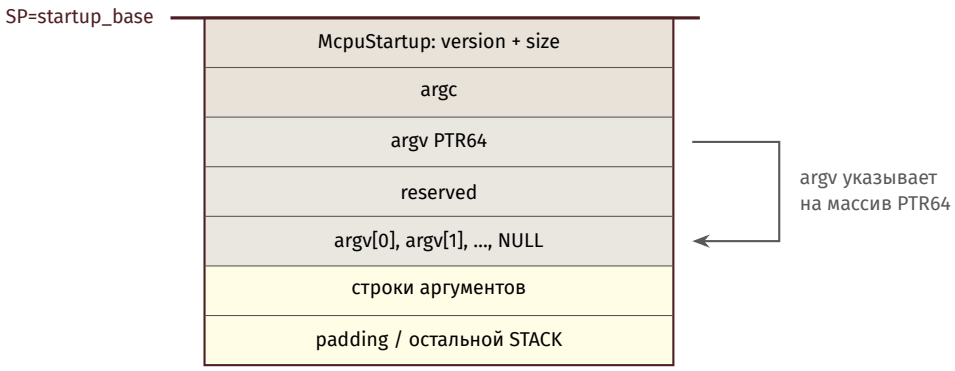


Рис. 3.6. Постоянная startup area в STACK: McpuStartup, массив argv и строки.

3.5.2. Почему argv является PTR64

Поле `argv` содержит PTR64 со selector-ом STACK. Это указатель на первый элемент массива указателей. Каждый `argv[i]` в свою очередь является PTR64(STACK,...) на начало соответствующей строки.

Получается двухступенчатая адресация:

$$\texttt{argv} \longrightarrow \texttt{argv[i]} \longrightarrow \text{байты строки.}$$

Чтобы получить `argv[1]`, надо прибавить к `argv` восемь байтов, потому что один PTR64 занимает 8 bytes. Затем `LOADP` читает сам указатель `argv[1]`. После этого к нему можно прибавлять смещения отдельных символов и снова использовать `LOADP`.

Именно здесь становится видно, зачем ранее мы отдельно разбирали pointer arithmetic: `PADDI` изменяет только 60-битный OFFSET и сохраняет SEG. Указатель остаётся указателем на STACK.

3.5.3. Минимальный пример argc/argv

Следующая программа ничего не печатает. Она оставляет результаты в регистрах, чтобы мы снова увидели их средствами `mcpu-run`. Пример считывает `argc`, затем символ `'h'` из аргумента `"–help"` и символ `'x'` из `"-x"`.

```
; argc/argv: читаем startup area и оставляем результаты в register file.

.text
_start_life:
        ; McpuStartup находится по начальному SP:
        ;   SP+8  = argc
        ;   SP+16 = argv (PTR64 на argv[0])
        LDSP    r64@0x00000, 8
        LDSP    r64@0x00008, 16

        ; argv[1][2] для строки "--help" -> 'h'
        PADDI   r64@0x00010, r64@0x00008, 8
        LOADP   r64@0x00018, r64@0x00010
        PADDI   r64@0x00018, r64@0x00018, 2
```

```
        LOADP   r8@0x00020, r64@0x00018

        ; argv[2][1] для строки "-x" -> 'x'
        PADDI   r64@0x00010, r64@0x00008, 16
        LOADP   r64@0x00018, r64@0x00010
        PADDI   r64@0x00018, r64@0x00018, 1
        LOADP   r8@0x00021, r64@0x00018

        HALT
```

Запустим программу с явным разделителем –:

```
mcpu-run \
  --int-reg=r64@0x00000 \
  --int-reg=r8@0x00020 \
  --int-reg=r8@0x00021 \
  -- argc-argv --help -x
```

Всё, что находится после первого –, больше не разбирается как опции runner-a. Первый token после separator-a является именем исполняемого файла и становится guest `argv[0]`. Поэтому программа получает:

```
argv[0] = "argc-argv"
argv[1] = "--help"
argv[2] = "-x"
```

Ожидаемые значения регистров:

Регистр	Значение	Смысл
`r64@0x00000`	3	`argc`
`r8@0x00020`	104	ASCII/UTF-8 byte `'h'`
`r8@0x00021`	120	ASCII/UTF-8 byte `'x'`

3.5.4. Разберём адресацию по шагам

В начале программы SP ещё указывает на `McpuStartup`, поэтому

```
LDSP    r64@0x00000, 8
LDSP    r64@0x00008, 16
```

читает соответственно `argc` и `argv`. Далее

```
PADDI   r64@0x00010, r64@0x00008, 8
```

формирует адрес элемента `argv[1]`. Инструкция

```
LOADP   r64@0x00018, r64@0x00010
```

читает из STACK восемь байтов PTR64 и получает указатель на строку `"–help"`. Ещё одно `PADDI ...,2` перемещает этот указатель к третьему байту строки, а `LOADP r8,...` считывает ровно один байт — букву `h`.

Второй аргумент разбирается тем же способом. Мы нигде не преобразовывали STACK address в host pointer и не обращались к памяти runner-а напрямую. Все операции выполнены внутри адресной модели MCPU.

3.5.5. Startup area и обычный стек функций

Постоянство startup area не означает, что SP навсегда останется равным `startup_base`. Как только программа начинает строить собственные кадры функций, SP будет уменьшаться и увеличиваться обычным образом. Сами байты `McpuStartup`, массив `argv` и строки при этом остаются живыми до завершения executable; указатели на них можно сохранить и использовать позднее.

Это важный переход к более крупным программам. В дальнейшем низкоуровневая `_start` может прочитать startup area, подготовить нормальный ABI-блок аргументов и вызвать функцию более высокого уровня. Именно на этом месте в будущей главе о собственной Libc и компиляторе появятся CRT и привычная функция `main(argc,argv)`. Сейчас нам достаточно понимать механическую основу этого перехода.

Makefile для примера:

```
.SUFFIXES:

MCPU_CPP = mcpu-cpp
MCPU_AS  = mcpu-as
MCPU_LD  = mcpu-ld
MCPU_RUN = mcpu-run
PROGRAM  = argc-argv

all: $(PROGRAM)

$(PROGRAM).S: $(PROGRAM).s
        $(MCPU_CPP) -o $@ $<
$(PROGRAM).o: $(PROGRAM).S
        $(MCPU_AS) -o $@ $<
$(PROGRAM): $(PROGRAM).o
        $(MCPU_LD) --stack-size=4K --no-heap -o $@ $<
run: $(PROGRAM)
        $(MCPU_RUN) \
```

```
          --after-run=print-int-regs:r64@0x00000,r8@0x00020,r8@0x00021 \
          -- $(PROGRAM) --help -x
clean:
        rm -f $(PROGRAM).S $(PROGRAM).o $(PROGRAM)
```

Команды проверки:

```
make
make run
```

На этом первая цепочка главы замыкается. Мы начали с физической модели регистра и PTR64, затем научились наблюдать состояние CPU, разобрали STACK и CALL ABI и в итоге прочитали настоящие параметры командной строки. Дальнейшие конструкции языка ассемблера теперь можно вводить поверх уже понятной модели машины, а не как набор несвязанных мнемоник.

3.6. Объекты данных в программе

До сих пор данные появлялись в наших программах в двух формах. Небольшие числовые значения мы помещали непосредственно в инструкции `LDI/LDR/LDC`, а значения, созданные во время выполнения, жили в register file или STACK. Для законченной программы этого мало. Нужны постоянные таблицы, начальные значения переменных, строки, массивы байтов и числа такой точности, которая не помещается в immediate-поле инструкции.

Эти объекты описываются в data-like sections исходного файла. В первых примерах нам достаточно секции `.data`. Важно сразу отделить три понятия, которые в исходнике стоят рядом, но относятся к разным уровням:

Понятие	Пример	Что оно задаёт
символ	`counter:`	имя места в section
директива объекта	`.int 64, 0`	физическое представление и начальное значение
адрес после link	`DATA:offset`	положение объекта в адресном пространстве процесса

Для будущего компилятора это различие фундаментально. Таблица символов отвечает на вопрос “что называется `counter`?”. Система типов отвечает на вопрос “каков размер и представление объекта?”. Layout и linker отвечают на вопрос “где объект окажется?”. Машинная инструкция уже использует результат этих решений: либо обращается к известному DATA offset, либо сначала строит PTR64.

3.6.1. Typed objects и raw bytes

Ассемблер MCPU имеет несколько директив создания данных. Они не являются “типами” в смысле языка C: CPU не хранит рядом с байтами метаданные `int`, `double` или `string`. Директива лишь говорит assembler-у, какие байты надо создать и какое target representation использовать.

`.int 64, 42`	typed integer object
`.real 64, 0.5`	typed Real object
`.complex 128, 1.0, -0.5`	two Real components
`.byte 0x4d,0x43,0x50,0x55`	raw bytes
`.utf8 "Hello"`	UTF-8 bytes + 00
`.ucs2 "Hello"`	UCS-2 u16 objects + 0000

Рис. 3.7. Разные способы формирования объектов в DATA.

`.byte` — наиболее низкий уровень. Она записывает ровно перечисленные байты, не добавляет terminator и не выполняет endian conversion:

```
header:
        .byte 0x4d, 0x43, 0x50, 0x55
```

Так удобно описывать magic, packed records, двоичные протоколы и любые участки, для которых программист сознательно задаёт физический byte layout.

`.integer` и её точный синоним `.int` создают integer objects заданной физической ширины:

```
count:
        .int 64, 42
codes:
        .integer 16, 0x0410, 0x043d, 0x0000
```

Каждое значение списка является отдельным объектом указанной ширины. Скрытого padding между элементами директива не добавляет. Байты объекта записываются в target representation MCPU/LibMPU, то есть относятся к data-object domain, а не к канонической MSB-first сериализации PROGRAM.

`.real` и `.complex` работают по той же идее, но создают полноразмерные LibMPU objects непосредственно в требуемой precision:

```
half:
        .real 64, 0.5

z:
        .complex 128, 1.0, -0.5
```

Для `.complex 128` полная ширина объекта равна 128 битам: это две Real64-компоненты. В отличие от immediate `LDR`/`LDC`, здесь нет промежуточного ограничения binary64 или двух binary32 components. Именно поэтому в предыдущем разделе мы использовали `.real 4096` и `.complex 4096` для высокоточных констант.

3.6.2. Строки: .utf8 и .ucs2

Строка полезна как первый пример объекта, который уже нельзя воспринимать одним числом. В MCPU assembler предусмотрены две разные строковые директивы.

`.utf8` получает строковый literal из UTF-8 source, записывает его как последовательность UTF-8 bytes и добавляет один завершающий byte `0x00`:

```
message8:
        .utf8 "Hello"
```

Физически для ASCII-частного случая получится:

```
48 65 6c 6c 6f 00
```

Здесь нет понятия endian: UTF-8 является последовательностью байтов.

`.ucs2` решает другую задачу. Source literal по-прежнему записан в UTF-8, но assembler преобразует Unicode characters в strict UCS-2 и сохраняет каждый code unit как отдельный 16-битный architectural data object. В конце добавляется U+0000:

```
message16:
        .ucs2 "Hello"
```

Логический набор code units равен

```
0048 0065 006c 006c 006f 0000
```

а физический порядок двух байтов внутри каждого `uint16` определяется target representation. Это принципиально отличается от UTF-16: strict UCS-2 не использует surrogate pairs, поэтому scalar values выше U+FFFF и surrogate range U+D800..U+DFFF для `.ucs2` недопустимы.

Обе строковые директивы создают C-style NUL-terminated strings. Если внутри объекта нужен произвольный нулевой byte или произвольная бинарная запись, используется `.byte`, а не строковая директива.

Escape-последовательности. В строковом литерале можно записывать не только обычные UTF-8 characters, но и управляющие characters или кодовые точки, которые неудобно помещать в source буквально. Текущий `mcpu-as` понимает следующий набор escape-последовательностей:

Запись	Значение
\\	обратная косая черта
\”	двойная кавычка
\n	перевод строки
\r	возврат каретки
\t	горизонтальная табуляция
\b	возврат на одну позицию
\f	перевод формата
\v	вертикальная табуляция
\a	звуковой сигнал
\xHH	ровно две шестнадцатеричные цифры: один byte
\uHHHH	ровно четыре шестнадцатеричные цифры: Unicode code point
\UHHHHHHHH	ровно восемь шестнадцатеричных цифр: Unicode code point

Например:

```
.utf8 ”first\tsecond\n”
.utf8 ”A=\x41”
.utf8 ”A=\u0410”
```

В форме \xHH после x должны находиться *ровно две* hexadecimal digits, то есть синтаксически задаётся значение одного byte от 0x00 до 0xff. Однако 0x00 внутри .utf8/.ucs2 запрещён, потому что завершающий NUL добавляет сама директива. Кроме того, после обработки всех escapes результат обязан оставаться корректной UTF-8 последовательностью. Поэтому одиночный произвольный byte из диапазона 0x80..0xff нельзя считать способом обойти проверку UTF-8: такие bytes допустимы только как часть корректной многобайтовой UTF-8 encoding.

Формы \uHHHH и \UHHHHHHHH задают не bytes, а Unicode scalar values. \u всегда содержит четыре hexadecimal digits, \U — восемь. Значение U+0000 запрещено как embedded NUL; surrogate range U+D800..U+DFFF также недопустим. Для .utf8 максимально допустимый Unicode scalar value равен U+10FFFF. Для .ucs2 действует дополнительное ограничение strict UCS-2: итоговая кодовая точка должна помещаться в один 16-битный code unit, то есть находиться в BMP и не быть surrogate.

Важно не переносить сюда синтаксис character/string constants языка C автоматически. В MCPU assembler \020 не является восьмеричной escape-последовательностью, а \0x20 не означает hexadecimal value. Оба варианта начинаются с \0 и отвергаются как попытка поместить embedded NUL. Для пробела следует написать, например,

```
.utf8 ”\x20”
.utf8 ”\u0020”
```

или просто обычный пробел внутри строки. Такое правило делает длину numeric escape однозначной: \x всегда читает ровно два hexadecimal digits, \u — четыре, а \U — восемь.

3.6.3. Выравнивание является частью layout

Assembler не обязан угадывать желаемое выравнивание следующего объекта. Для этого служит `.align`:

```
.data
.align 8
counter:
        .int 64, 0
```

`.align 8` передвигает текущую position section к следующей подходящей границе и при необходимости добавляет padding. Для обычной DATA это реальные байты образа; для NOBITS-подобного пространства меняется размер layout без payload bytes.

Для компилятора layout объекта поэтому определяется не только его размером. В общем случае генератор кода должен знать по крайней мере

`size`, `alignment`, `representation`.

Если позже мы введём структуры, именно эти три характеристики будут определять offsets полей, tail padding и размер массива структур.

3.6.4. Нулевая память: .bss и .zero

Не всякий объект нуждается в явном наборе начальных байтов внутри файла. Для объектов, которые к началу выполнения должны состоять из нулей, assembler поддерживает секцию `.bss`. В MELF такая секция представляется как NOBITS: она имеет размер и участвует в layout, но её нулевое содержимое не требуется хранить в файле как обычный payload.

Простейший пример:

```
.bss
.align 16
scratch:
        .zero 256
```

`.zero N` резервирует `N` нулевых байтов. В `.bss` это особенно естественно: увеличивается размер zero-initialized области без необходимости сериализовать эти байты в MELF. Та же директива допустима и в обычной data-like section, где она создаёт соответствующий нулевой участок. Скрытого автоматического выравнивания `.zero` не выполняет; если оно требуется следующему объекту, используется явная `.align`.

Для будущего компилятора различие между `.data` и `.bss` практически означает выбор способа хранения начального состояния объекта. Объект с ненулевым статическим инициализатором требует содержимого в `.data`; большой объект, начальное значение которого целиком равно нулю, естественно размещать в `.bss`.

3.6.5. Label не является PTR64 object

Запись

```
value:
        .int 64, 42
```

определяет symbol `value`, но не создаёт рядом отдельный указатель. Symbol позволяет assembler-у и linker-у сослаться на position объекта. Если программе нужен настоящий PTR64, его надо сформировать явно:

```
LEA     r64@0x00100, value
```

`LEA` помещает в `r64@0x00100` полный PTR64 с правильным selector-ом пространства и offset объекта. Напротив,

```
LOAD    r64@0x00108, value
```

не получает адрес как значение, а непосредственно читает 64-битный объект по DATA-relative адресу symbol-а.

Это различие соответствует двум классическим операциям генератора кода языка высокого уровня:

Нужно получить	Типичная машинная форма
значение объекта	`LOAD` или `LOADP`
адрес объекта	`LEA`, `LEASP`, `LEAFP`

Уже здесь появляется важная для компилятора категория **lvalue** (*вычисляемое местоположение объекта*). Выражение может требовать не само значение переменной, а её адрес. На уровне MCPU это различие становится буквальным: одна последовательность строит PTR64, другая разыменовывает его.

3.6.6. Указатели как объекты данных: .ptr и .cptr

`LEA` строит PTR64 во время выполнения программы, но иногда сам указатель должен уже находиться в статическом объекте данных: например, в таблице адресов объектов, таблице функций или таблице переходов. Для этого служат директивы `.ptr` и `.cptr`. Обе создают ровно один 64-битный architectural PTR64 object и, если окончательный адрес символа ещё неизвестен в ET_REL, оставляют компоновщику соответствующую relocation.

Общая форма `.ptr` создаёт PTR64 на допустимый статически адресуемый объект или кодовую метку:

```
.data
.align 8
value:
        .int 64, 42
p_value:
        .ptr value
```

После компоновки `p_value` содержит не число `42`, а полный PTR64, указывающий на `value`. В текущем MELF ABI статическая relocation `.ptr` может ссылаться на image-backed пространства DATA, PROGRAM, CONTROL, INPUT, OUTPUT и TRACE. На STACK и HEAP она ссылаться не может: в relocatable image нет статических symbols, обозначающих объекты этих runtime-пространств.

`.cptr` — более строгая специализированная форма для указателя на код. Её target обязан разрешиться именно в PROGRAM:

```
.data
.align 8
handler:
        .cptr .L_handler

.text
.L_handler:
        RET
```

Такой объект удобно загружать в `r64` и передавать инструкциям `CALLP` или `JMPP`. Именно поэтому позднее `.cptr` естественно появится в jump table для `switch`. Историческое имя `.codeptr` assembler продолжает принимать как совместимый синоним, но в новых исходниках следует использовать каноническое `.cptr`.

Есть ещё одна важная деталь представления. `.ptr` и `.cptr` находятся в data-object domain. Их восемь байтов записываются в target representation обычного 64-битного архитектурного объекта. Это не то же самое, что 64-битное поле адреса внутри инструкции PROGRAM, которое сериализуется канонически MSB-first. И `.ptr`, и `.cptr` сами по себе не добавляют выравнивание, поэтому перед массивом таких указателей обычно явно пишут `.align 8`.

Таким образом, полезно различать три близких по виду механизма:

Форма	Что получается
`LEA r64@p, object`	PTR64 вычисляется во время выполнения и помещается в регистр
`.ptr object`	PTR64 хранится как статический объект данных
`.cptr label`	статический PTR64, причём target обязан принадлежать PROGRAM

Это различие особенно важно для генератора кода: адрес может понадобиться как временное вычисленное значение в регистре или как часть статически инициализированного объекта, который должен быть создан ещё при компоновке.

3.6.7. Один пример со всеми основными объектами

В каталоге `examples/data-objects` находится программа, в которой рядом размещены integer, Real, Complex, raw bytes, UTF-8 и UCS-2 strings:

```
; Объекты данных разных видов в секции DATA.

.data
.align 8
answer:
        .int 64, 42
half:
        .real 64, 0.5
z:
        .complex 128, 1.0, -0.5
header:
        .byte 0x4d, 0x43, 0x50, 0x55
message8:
        .utf8 "Hello"
message16:
        .ucs2 "Hello"

.text
_start_life:
        LOAD    r64@0x00000, answer
        LOAD    r64@0x00008, half
        LOAD    c128@0x00010, z
        LEA     r64@0x00020, message8
        LEA     r64@0x00028, message16
        HALT
```

После link первый пользовательский объект DATA располагается уже после linker-owned NULL guard, о котором мы говорили в §3.1. Внутри ET_REL его section-relative position может быть другой: имена объектов намеренно используются вместо предположений о конечных offsets.

Собрать пример можно обычной цепочкой:

```
.SUFFIXES:

MCPU_CPP = mcpu-cpp
MCPU_AS  = mcpu-as
MCPU_LD  = mcpu-ld
MCPU_RUN = mcpu-run
PROGRAM  = data-objects

all: $(PROGRAM)

$(PROGRAM).S: $(PROGRAM).s
        $(MCPU_CPP) -o $@ $<
$(PROGRAM).o: $(PROGRAM).S
        $(MCPU_AS) -o $@ $<
```

```
$(PROGRAM): $(PROGRAM).o
        $(MCPU_LD) --stack-size=1K --no-heap -o $@ $<
run: $(PROGRAM)
        $(MCPU_RUN) --after-run=print-int-regs:r64@0x00000 --after-run=print-real-regs:
    r64@0x00008 -- $(PROGRAM)
clean:
        rm -f $(PROGRAM).S $(PROGRAM).o $(PROGRAM)
```

Команда `make run` показывает integer и Real, загруженные из DATA:

```
42
5.000000000000000E-1
```

Это ещё один полезный образ: директива данных не “исполняется” процессором. Она формирует объект в executable image. Только последующая инструкция `LOAD` переносит этот объект в register file.

Полный source contract assembler-a содержит также `.control`, `.input`, `.output` и `.trace`. Они создают contributions специальных пространств программной модели MCPU и потому будут подробно разбираться вместе с model/runtime ABI. Директивы `.global`/`.globl`, `.extern` и `.weak` относятся уже не к физическому представлению объекта, а к symbol binding и компоновке; их естественное место — дальнейшее обсуждение relocatable objects и linker-a. Таким образом, для базовой модели объектов данных в этом разделе мы уже имеем полный необходимый набор механизмов.

3.6.8. Что отсюда понадобится будущему компилятору

Даже такой небольшой набор директив уже позволяет сформулировать обязанности семантического анализатора[1] и генератора кода. Для каждой декларации высокого уровня компилятор должен решить:

- создаёт ли декларация объект хранения или только имя/псевдоним;
- какова физическая ширина объекта;
- какое требуется выравнивание;
- является ли инициализатор константой времени компиляции;
- можно ли закодировать значение непосредственным операндом или нужен объект в DATA;
- требуется ли программе значение объекта или его адрес;
- какая релокация понадобится, если инициализатор содержит адрес другого символа.

В дальнейшем эти решения усложнятся массивами, структурами и указателями, но принцип уже виден. Ассемблерная директива — это конечный шаг цепочки, которая в языке высокого уровня начинается с типа, декларации данных и инициализатора.

[1] Классическое изложение задач семантического анализа см., например: А. Ахо, М. Лам, Р. Сети, Дж. Ульман. «Компиляторы: принципы, технологии и инструментарий».

3.7. Наблюдение за ходом выполнения: минимальный вывод в stdout

До сих пор мы специально избегали вывода из guest-программы. Такой подход был полезен: значения регистров и DATA можно было наблюдать средствами `mcpu-run`, не вводя раньше времени системные вызовы. Но для большого количества ветвлений этот способ быстро становится неудобным. Если программа имеет десять возможных путей, хочется видеть не только конечное число, но и сам путь: какой переход был выбран, сколько раз выполнен цикл, какой вариант `case` достигнут.

Поэтому сейчас мы немного забежим вперёд и введём одну маленькую служебную функцию `print(ptr)`. Она принимает единственный PTR64, указывающий на завершаемую нулём строку, созданную директивой `.utf8`, сама находит завершающий NUL, вычисляет длину строки и выводит полезные байты в stdout. Эта функция понадобится как *инструмент наблюдения*. Полный syscall ABI, файловые дескрипторы, ошибки и будущие библиотечные оболочки мы разберём позже в отдельном системном разделе.

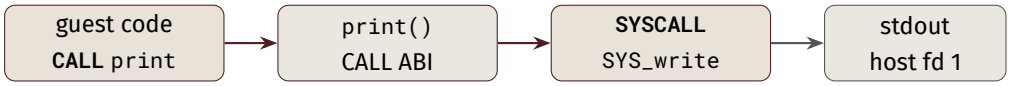


Рис. 3.8. Минимальный путь строки из guest-программы в stdout.

Важно, чтобы здесь не возникло скрытой магии. `print` остаётся обычной MCPU-функцией CALL ABI. Единственное новое действие внутри неё — инструкция `SYSCALL`; её полный контракт мы пока сознательно не раскрываем.

3.7.1. Строка в DATA и завершающий NUL

Начнём со строки:

```
.data
hello:
        .utf8 "Hello from MCPU\n"
```

Директива `.utf8` помещает в DATA байты UTF-8 и автоматически добавляет завершающий нулевой байт. Поэтому вызывающей программе не требуется отдельно хранить длину строки. Достаточно получить PTR64 на первый байт:

```
LEA     r64@0x00000, hello
```

а затем передать этот указатель функции `print`:

```
PUSH    r64@0x00000
CALL    print
```

```
ADJSP   8
```

Здесь есть полезный для будущей библиотеки образ: контракт функции определяется не только физическим типом аргумента PTR64, но и смыслом объекта, на который он указывает. Для `print` этим объектом является именно NUL-terminated UTF-8 string. Если передать указатель на область, в которой завершающего нуля нет, функция будет продолжать чтение до первого встретившегося нулевого байта или до ошибки доступа. Проверка границ пока не является частью этого учебного контракта.

3.7.2. Имена символов и локальные метки

В следующем листинге впервые появляются имена вроде `.L_find_nul` и `.L_write_string`. Точка в начале имени не является специальной командой assembler-а. Это допустимый первый символ идентификатора и одновременно удобное соглашение: префиксом `.L_` мы будем обозначать внутренние локальные метки, которые нужны реализации функции и не являются частью её внешнего интерфейса.

Текущий `mcpu-as` принимает имя символа по следующим правилам. Первый символ может быть латинской буквой, знаком подчёркивания `_`, точкой `.` или знаком доллара `$`. Начиная со второго символа разрешены также десятичные цифры. Поэтому допустимы, например:

```
print
.L_find_nul
.L_write_string
_loop2
$temporary
```

а имя, начинающееся с цифры, меткой быть не может. Имена чувствительны к регистру: `Loop` и `loop` — разные символы.

Важно отличать *синтаксис имени* от *связывания символа*. Сам префикс `.L_` не делает символ локальным особым механизмом. Обычная метка MCPU и без того получает LOCAL binding, пока программист явно не объявит её через `.global`/`.globl` или `.weak`; lifecycle symbols имеют отдельные правила ABI. Поэтому `.L_` — прежде всего читаемая договорённость между автором assembler-кода, будущим компилятором и человеком, который будет изучать сгенерированный листинг.

Для генератора кода компилятора такое соглашение особенно удобно. Генератор кода может систематически создавать имена `.L_if_17`, `.L_loop_test_4`, `.L_switch_default_2`, не смешивая их с функциями и объектами, названными программистом.

3.7.3. Как print находит длину строки

После `CALL` единственный 8-байтовый аргумент находится по адресу `SP+8`, потому что по `SP+0` уже лежит return PC. Функция загружает исходный указатель и создаёт его рабочую копию:

```
print:
        LDSP    r64@0x00100, 8
        MOV     r64@0x00108, r64@0x00100
```

Регистр `r64@0x00100` сохраняет адрес начала строки, а `r64@0x00108` движется по строке. На каждой итерации `LOADP` читает один байт, после чего `ICMP` сравнивает его с нулём:

```
.L_find_nul:
        LOADP   r8@0x00110, r64@0x00108
        CLR     r8@0x00111
        ICMP    r8@0x00110, r8@0x00111
        JZ      .L_write_string
        PADDI   r64@0x00108, r64@0x00108, 1
        JMP     .L_find_nul
```

Если очередной байт ненулевой, `PADDI` увеличивает PTR64 на один байт и цикл продолжается. Когда найден NUL, рабочий указатель указывает уже на terminator, а исходный всё ещё указывает на первый символ. Поэтому длина строки получается непосредственно как разность двух указателей:

```
PDIFF   r64@0x00118, r64@0x00108, r64@0x00100
```

Для строки `"Hello from MCPU\n "` результат равен 16. Сам завершающий NUL в эту длину не входит и в stdout не передаётся.

3.7.4. Короткий предварительный взгляд на SYSCALL ABI

Инструкция `SYSCALL` не имеет операндов в PROGRAM. Номер системного вызова и его аргументы передаются через фиксированную область в верхней части register file. Сейчас нам нужны только поля:

Роль	Представление	Значение для write
NARGS	`r32@0x7fff8`	3
SYSNO	`r32@0x7fffc`	`__NR_write`
ARG0	`r64@0x7fff0`	`STDOUT_FILENO`
ARG1	`r64@0x7ffe8`	PTR64 строки
ARG2	`r64@0x7ffe0`	длина в байтах
RET	`r64@0x7fff8`	результат после `SYSCALL`

NARGS и RET намеренно перекрываются: до инструкции там находится число аргументов, после неё — результат. Номер `__NR_write` публикуется target header-ом `<sys/asm/syscall.h>`, а `STDOUT_FILENO` — header-ом `<sys/asm/unistd.h>`; оба имени подставляются препроцессором.

Полное тело служебной функции получается таким:

```
print:
        ; Первый 8-байтовый аргумент находится по SP+8:
        ; CALL уже положил return PC по SP+0.
        LDSP    r64@0x00100, 8
        MOV     r64@0x00108, r64@0x00100

.L_find_nul:
        LOADP   r8@0x00110, r64@0x00108
        CLR     r8@0x00111
        ICMP    r8@0x00110, r8@0x00111
        JZ      .L_write_string
        PADDI   r64@0x00108, r64@0x00108, 1
        JMP     .L_find_nul

.L_write_string:
        PDIFF   r64@0x00118, r64@0x00108, r64@0x00100

        ; write(STDOUT_FILENO, string, length)
        LDI     r32@0x7fff8, 3
        LDI     r32@0x7fffc, __NR_write
        LDI     r64@0x7fff0, STDOUT_FILENO
        MOV     r64@0x7ffe8, r64@0x00100
        MOV     r64@0x7ffe0, r64@0x00118
        SYSCALL

        RET
```

Мы пока сознательно не разбираем отрицательные errno, частичную запись, повтор системного вызова и различие между raw syscall и будущей Libc. Для последующих примеров достаточно одного договора: `print(ptr)` принимает PTR64 на NUL-terminated UTF-8 string и выводит её полезные байты в stdout.

3.7.5. Полный служебный пример

Полный исходник лежит в `examples/print-string`:

```
#include <sys/asm/syscall.h>
#include <sys/asm/unistd.h>

.data
hello:
        .utf8 "Hello from MCPU\n"

.text
_start_life:
        LEA     r64@0x00000, hello
```

```
        ; print(ptr): один PTR64 параметр -> 8-byte argument block.
        PUSH    r64@0x00000
        CALL    print
        ADJSP   8
        HALT

; print(const char *ptr)
; ptr указывает на NUL-terminated UTF-8 string.
print:
        ; Первый байтовый8- аргумент находится по SP+8:
        ; CALL уже положил return PC по SP+0.
        LDSP    r64@0x00100, 8
        MOV     r64@0x00108, r64@0x00100

.L_find_nul:
        LOADP   r8@0x00110, r64@0x00108
        CLR     r8@0x00111
        ICMP    r8@0x00110, r8@0x00111
        JZ      .L_write_string
        PADDI   r64@0x00108, r64@0x00108, 1
        JMP     .L_find_nul

.L_write_string:
        PDIFF   r64@0x00118, r64@0x00108, r64@0x00100

        ; write(STDOUT_FILENO, string, length)
        LDI     r32@0x7fff8, 3
        LDI     r32@0x7fffc, __NR_write
        LDI     r64@0x7fff0, STDOUT_FILENO
        MOV     r64@0x7ffe8, r64@0x00100
        MOV     r64@0x7ffe0, r64@0x00118
        SYSCALL
        RET
```

Makefile также не вводит ничего нового:

```
.SUFFIXES:
MCPU_CPP = mcpu-cpp
MCPU_AS  = mcpu-as
MCPU_LD  = mcpu-ld
MCPU_RUN = mcpu-run
PROGRAM  = print-string
all: $(PROGRAM)
$(PROGRAM).S: $(PROGRAM).s
        $(MCPU_CPP) -o $@ $<
$(PROGRAM).o: $(PROGRAM).S
        $(MCPU_AS) -o $@ $<
$(PROGRAM): $(PROGRAM).o
        $(MCPU_LD) --stack-size=1K --no-heap -o $@ $<
run: $(PROGRAM)
        $(MCPU_RUN) -- $(PROGRAM)
```

```
clean:
        rm -f $(PROGRAM).S $(PROGRAM).o $(PROGRAM)
```

Команда `make run` создаёт и запускает пример. Результат:

```
Hello from MCPU
```

С этого места в control-flow examples короткие строки `"A\n"`, `"less\n"`, `"loop\n"` будут использоваться как наблюдаемые маркеры пути. Это не меняет основной предмет следующего раздела: мы изучаем не ввод-вывод, а то, почему управление попало именно в этот basic block.

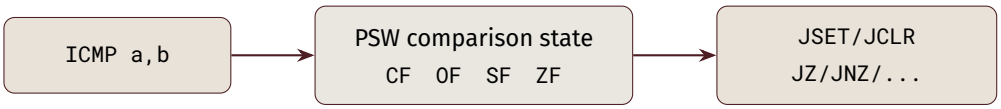
3.8. PSW, сравнения и переходы

До этого момента PC почти всегда двигался линейно: после одной инструкции выполнялась следующая. Даже CALL и RET мы рассматривали прежде всего как механизм функций. Теперь программа впервые начинает принимать решение о том, какой участок PROGRAM выполнять дальше.

На языке высокого уровня условие выглядит как выражение:

```
if (a < b)
        ...
```

Но MCPU не обязан создавать для каждого такого сравнения отдельный boolean object со значением 0 или 1. Для integer comparison существует более низкий уровень: `ICMP` формирует состояние PSW, а branch instruction интерпретирует это состояние.



сравнение не создаёт boolean object;
переход интерпретирует сохранённое состояние

Рис. 3.9. Сравнение формирует состояние PSW, а переход использует его.

Это различие особенно важно для генератора кода компилятора. В контексте `if` значение выражения часто вообще не требуется как объект. Требуется только выбрать одну из двух control-flow edges.

3.8.1. ICMP выполняет мысленное a-b

Инструкция

```
ICMP    r64@a, r64@b
```

не записывает арифметический результат в register file. Семантически она выполняет сравнение как subtraction `a-b`, оставляя оба operands неизменными, и обновляет integer comparison flags в PSW.

Для построения обычных relational conditions прежде всего важны четыре флага:

Flag	Смысл после ICMP a,b	Типичное применение
ZF	результат `a-b` равен нулю	equality / inequality
CF	возник unsigned borrow	unsigned ordering
SF	sign bit результата	часть signed ordering
OF	signed overflow subtraction	коррекция signed ordering

`AF` и `PF` также принадлежат integer status domain и могут быть проверены через универсальные branch selectors, но для lowering обычных C-like relations они обычно не нужны.

Здесь полезно заметить важный архитектурный факт: integer register view не имеет встроенной метки “signed” или “unsigned”. Одни и те же физические биты могут интерпретироваться обоими способами. Поэтому различие signed/unsigned comparison является решением программы или компилятора, а не свойством `r64`.

3.8.2. JSET и JCLR — фундаментальные conditional branches

На уровне ISA существуют две базовые conditional instructions:

```
JSET    PSW_BIT, target
JCLR    PSW_BIT, target
```

`JSET` переходит на `target`, если выбранный branchable PSW bit установлен. `JCLR` делает то же при очищенном бите. Target прямого перехода является CODE64 address в PROGRAM.

Для наиболее употребительных integer flags assembler предоставляет aliases:

Alias	Эквивалент	Смысл
JZ	`JSET ZF,target`	zero / equal
JNZ	`JCLR ZF,target`	nonzero / not equal
JC	`JSET CF,target`	carry/borrow set
JNC	`JCLR CF,target`	carry/borrow clear
JO	`JSET OF,target`	overflow set
JNO	`JCLR OF,target`	overflow clear
JS	`JSET SF,target`	sign set
JNS	`JCLR SF,target`	sign clear

Эти aliases не являются отдельными opcodes. Поэтому генератор кода может оперировать либо удобными mnemonic names, либо более общей моделью “branch if PSW bit has value V”.

Безусловный прямой переход задаётся **JMP**:

```
JMP     target
```

Если branch не сработал, PC продолжает выполнение со следующей instruction. Такой путь называется **fall-through** (*естественное продолжение управления без явного перехода*).

3.8.3. Равенство и неравенство

Самые простые relations используют только ZF:

```
ICMP    r64@0x00000, r64@0x00008
JZ      .L_equal
```

Для `a != b` достаточно заменить `JZ` на `JNZ`. Здесь signedness вообще не имеет значения: equality сравнивает одинаковые bit patterns одинаковой физической ширины.

Именно поэтому equality PTR64 тоже можно проверять обычным 64-bit `ICMP`: полные pointer representations либо совпадают, либо нет. Но такой приём не даёт meaningful ordering для указателей разных SEG, что мы уже обсуждали.

3.8.4. Unsigned ordering

После subtraction `a-b` установленный CF означает borrow, то есть для unsigned operands `a<b`. Отсюда непосредственно следуют relations:

Высокий уровень	Условие PSW	Простейший lowering
`a < b`	CF=1	`JC target`
`a >= b`	CF=0	`JNC target`
`a <= b`	CF=1 или ZF=1	две проверки
`a > b`	CF=0 и ZF=0	две проверки

Например, unsigned `a <= b` можно выразить так:

```
ICMP    r64@a, r64@b
JC      .L_true
JZ      .L_true
JMP     .L_false
```

A `a > b`:

```
ICMP    r64@a, r64@b
JC      .L_false
JZ      .L_false
JMP     .L_true
```

Генератор кода не обязан буквально генерировать именно эти последовательности: в реальном CFG один из targets часто является fall-through, поэтому последний JMP можно убрать. Но логическое условие остаётся тем же.

3.8.5. Почему для signed ordering недостаточно SF

На первый взгляд может показаться, что signed a<b означает просто “после a-b установлен SF”. Это неверно при signed overflow.

Представим fixed-width subtraction, в котором математический результат вышел за signed range. Старший bit физического результата тогда уже не соответствует математическому знаку. Флаг OF именно и сообщает, что sign interpretation была искажена overflow-ом.

Поэтому стандартная relation для signed comparison имеет вид

$$a < b \iff SF \mathrel{!}= OF.$$

Соответственно:

Relation	Condition after ICMP a,b
a < b	SF != OF
a >= b	SF == OF
a <= b	ZF == 1 || SF != OF
a > b	ZF == 0 && SF == OF

MCPU сейчас не имеет отдельного opcode JL или JGE, который внутри одной instruction проверял бы комбинацию SF/OF. Поэтому генератор кода компилятора должен разложить relation на primitive branches.

Например, signed a<b можно реализовать так:

```
        ICMP    r64@a, r64@b
        JS      .L_sf_set
        JO      .L_less          ; SF=0, OF=1
        JMP     .L_not_less
.L_sf_set:
        JNO     .L_less          ; SF=1, OF=0
        JMP     .L_not_less
```

На первый взгляд последовательность длиннее привычного ISA с compound condition codes. Но для compiler construction она даже полезна педагогически: логическое определение signed relation не скрыто внутри mnemonic-a.

3.8.6. Таблица отношений как контракт генератора кода

Здесь и далее мы будем использовать имена с префиксом `emit_` для *семантических процедур этапа генерации кода*. Это не функции LibMCPU и не обязательный интерфейс будущего компилятора. Такая запись нужна как компактная учебная нотация для процедур, которые получают уже распознанное и типизированное внутреннее представление конструкции языка высокого уровня и порождают соответствующий MCPU assembler text. В реальном компиляторе между исходным текстом и такой процедурой обычно находятся лексический и синтаксический анализ, построение внутреннего представления программы, таблицы символов, проверка типов и другие стадии семантического анализа. Затем генератор кода переводит это представление в последовательность машинно-зависимых операций или, как в рассматриваемой нами модели, в готовый текст программы на MCPU Assembler.[2]

Поэтому запись

```
emit_relation(a, b, relation, signedness, L_true, L_false)
```

следует читать не как вызов конкретной библиотечной функции, а как описание обязанности генератора кода: получить два операнда, вид отношения, сведения о signed/unsigned semantics и две цели потока управления, после чего породить правильную MCPU-последовательность.

При проектировании компилятора удобно вообще не связывать semantic operator `<` с одной конкретной instruction. Семантический анализ определяет типы и смысл operands, а процедура `emit_relation(...)` уже на этапе генерации кода выбирает правильную комбинацию `ICMP` и branches.

Тогда source expressions

```
signed_a   < signed_b
unsigned_a < unsigned_b
ptr_a      == ptr_b
```

могут иметь одинаковый AST operator, но различный semantic type и различный lowering. Это хороший пример того, почему генератор кода должен получать уже типизированное дерево, а не угадывать signedness по внешнему виду operands.

[2] О семантических процедурах и построении интерпретаторов и компиляторов см. также: Костельцев А. В. «Построение интерпретаторов и компиляторов: Использование программ BIZON, BYACC, ZUBR». СПб.: Наука и Техника, 2001. См. [3] в списке литературы главы 2.

3.8.7. PSW является временным состоянием

Flags не являются вечным свойством operands. Они описывают результат последней инструкции, которая соответствующие flags изменила. Поэтому такая последовательность потенциально ошибочна:

```
ICMP    r64@a, r64@b
IADD    r64@x, r64@y, r64@z
JZ      .L_equal
```

`IADD` публикует собственные arithmetic flags и уничтожает comparison state, которое ожидал `JZ`. Между producer-ом flags и их потребителем генератор кода должен либо не вставлять flag-clobbering instructions, либо заново выполнить comparison.

Именно здесь становится полезным ранее установленное правило PSW-neutral conversions. `LDI`, `LDR`, `LDC` и определённые conversion instructions сохраняют incoming status flags, но обычная арифметика этого не обещает.

Для compiler scheduler-а это означает настоящую data dependency, хотя значение не хранится в register file:

$$\texttt{ICMP} \longrightarrow \texttt{PSW} \longrightarrow \texttt{branch}.$$

Если промежуточное представление compiler-а умеет моделировать flags как implicit value, такая зависимость становится явной для оптимизатора.

3.8.8. Прямой и косвенный control transfer

`JMP target` содержит прямой CODE64 target. Имя label может быть forward reference; assembler создаёт fixup/relocation, а окончательное разрешение выполняется на соответствующем этапе сборки и link.

Для косвенного перехода существует `JMPP`:

```
JMPP    r64@0x00100
```

Регистр обязан содержать валидный PTR64 с selector-ом PROGRAM. Такой механизм позже станет основой jump table для `switch`, function pointers и других форм computed control flow.

Важно различать CODE64 и PTR64. Direct `JMP` хранит PROGRAM address field как часть instruction encoding. `JMPP` получает архитектурный pointer object из register file и во время выполнения проверяет его selector и range.

3.8.9. Первый наблюдаемый branch

Пример `examples/branch-basic` сравнивает два signed integer и корректно реализует relation `a<b` через `SF != OF`. Выбранная ветвь печатает короткую строку:

```
#include <sys/asm/syscall.h>
#include <sys/asm/unistd.h>

.data
msg_less:     .utf8 "less\n"
msg_not_less: .utf8 "not less\n"

.text
_start_life:
        LDI     r64@0x00000, -5          ; a
        LDI     r64@0x00008, 7           ; b

        ; signed a < b  <=>  SF != OF after ICMP a,b
        ICMP    r64@0x00000, r64@0x00008
        JS      .L_sf_set
        JO      .L_less                  ; SF=0, OF=1
        JMP     .L_not_less
.L_sf_set:
        JNO     .L_less                  ; SF=1, OF=0
        JMP     .L_not_less

.L_less:
        LEA     r64@0x00020, msg_less
        JMP     .L_print

.L_not_less:
        LEA     r64@0x00020, msg_not_less

.L_print:
        PUSH    r64@0x00020
        CALL    print
        ADJSP   8
        HALT

print:
        ; Первый байтовый8- аргумент находится по SP+8:
        ; CALL уже положил return PC по SP+0.
        LDSP    r64@0x00100, 8
        MOV     r64@0x00108, r64@0x00100

.L_find_nul:
        LOADP   r8@0x00110, r64@0x00108
        CLR     r8@0x00111
        ICMP    r8@0x00110, r8@0x00111
        JZ      .L_write_string
        PADDI   r64@0x00108, r64@0x00108, 1
        JMP     .L_find_nul

.L_write_string:
        PDIFF   r64@0x00118, r64@0x00108, r64@0x00100

        ; write(STDOUT_FILENO, string, length)
```

```
        LDI     r32@0x7fff8, 3
        LDI     r32@0x7fffc, __NR_write
        LDI     r64@0x7fff0, STDOUT_FILENO
        MOV     r64@0x7ffe8, r64@0x00100
        MOV     r64@0x7ffe0, r64@0x00118
        SYSCALL
        RET
```

Команда

```
make run
```

даёт:

```
less
```

Теперь у нас есть все primitive building blocks для конструкции `if`. Но прежде чем переводить синтаксис языка высокого уровня строка за строкой, полезно сменить уровень абстракции и посмотреть на программу как на граф потока управления.

3.9. Basic blocks и граф потока управления

Человек, читающий небольшой assembler source, способен проследить выполнение глазами: вот label, вот comparison, вот branch. Компилятору этого недостаточно. Чтобы систематически преобразовывать конструкцию высокого уровня в машинный код, полезно представить программу как **control-flow graph** (*граф потока управления, CFG*).

Узлами такого графа являются **basic blocks** (*базовые блоки*). В простейшем определении basic block — это максимальная последовательность instructions с одним входом в начале и без внутреннего ветвления: если управление попало на первую instruction блока, оно последовательно пройдёт все instructions до terminator-а или fall-through конца блока.

3.9.1. Почему labels сами по себе ещё не CFG

Рассмотрим assembler fragment:

```
B0:
        ICMP    r64@a, r64@b
        JNZ     B2
B1:
        LDI     r64@x, 1
        JMP     B3
B2:
        LDI     r64@x, 2
```

```
B3:
        ...
```

Text labels обозначают addresses, но semantic structure видна лучше в графе:

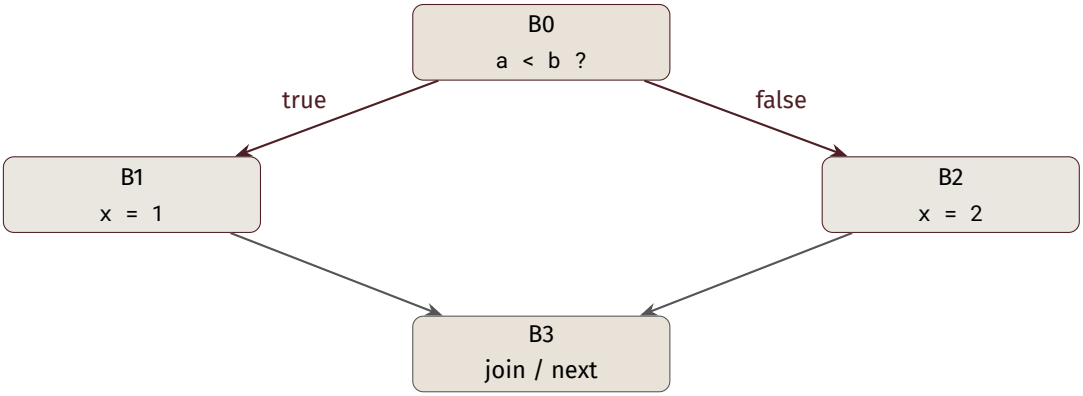


Рис. 3.10. Типичный CFG для конструкции с двумя альтернативами и общей точкой продолжения.

У блока `B0` два возможных successor-а. У `B1` один explicit successor через `JMP`. У `B2` successor достигается fall-through. `B3` является **join block** (*блоком слияния*): туда сходятся обе альтернативы.

Именно edges, а не положение строк в source, задают control semantics программы.

3.9.2. Entry, exit, predecessor и successor

Для дальнейшего разговора полезно зафиксировать минимальный словарь:

Термин	Смысл
entry	точка входа в graph или construct
exit	точка, после которой construct завершён
predecessor	block, из которого существует edge в данный block
successor	block, куда может перейти управление из данного block
fall-through	successor без explicit jump
branch target	explicit destination conditional/unconditional branch
join	block, в который сходятся несколько paths

Для начинающего это просто удобные названия. Для compiler writer это уже основа data-flow analysis: чтобы знать, какие definitions достигают block-а, какие values live на выходе и можно ли переставить instruction, надо знать predecessors и successors.

3.9.3. AST и CFG отвечают на разные вопросы

Абстрактное синтаксическое дерево удобно выражает structure исходного языка. Например,

```
if (a < b)
        x = 1;
else
        x = 2;
```

естественно представляется AST node `If(condition, then, else)`. Но AST не говорит напрямую, какой machine block станет fall-through, где расположится join и сколько branches потребуется.

CFG, наоборот, уже почти не интересуется исходной пунктуацией. Он отвечает на вопросы:

- откуда может прийти управление;
- куда оно может уйти;
- какие blocks исполняются условно;
- где paths снова сходятся;
- где образуется back edge цикла.

Поэтому типичная compiler pipeline содержит переход от syntax/typed AST к более низкому representation, в котором control flow уже явный.

3.9.4. Метки как symbolic control-flow targets

При генерации ассемблерного текста генератор кода не обязан заранее знать numeric PROGRAM offset будущего блока. Он может создать symbolic label:

```
        JNZ    .L_else
        ...
.L_else:
        ...
```

Assembler и linker разрешат target позднее. Это является простейшей формой **backpatching** (*отложенного связывания ещё неизвестной конечной позиции перехода*).

Исторически компиляторы могли действительно хранить списки незаполненных branch addresses и позже patch-ить их. При генерации symbolic assembler source роль такого patch list во многом принимает на себя система labels/relocations. Но semantic compiler всё равно должен знать, *какой* target нужен каждой edge.

3.9.5. Условие лучше lowering-ить в два target-а

Очень полезная абстракция для генератора кода выглядит не как

```
bool_value = emit_boolean(expr)
```

а как

```
emit_condition(expr, L_true, L_false)
```

То есть условие сразу получает два control destinations. Для узла отношения генератор кода может сгенерировать `ICMP` и branches. Для `&&` и `||` эта abstraction естественно расширяется short-circuit semantics, которую мы разберём отдельно.

Преимущество видно уже для простого comparison. Source expression

```
a < b
```

может использоваться в двух разных contexts:

```
if (a < b) ...      ; нужен control-flow decision
x = (a < b);        ; нужен materialized value 0 или 1
```

В первом случае создавать boolean register часто бессмысленно. Во втором без materialization уже не обойтись. Один и тот же AST node получает разный lowering в зависимости от context.

3.9.6. Терминатор и fall-through как ресурс генератора кода

Удобно считать, что basic block завершается instruction, определяющей следующий control flow: unconditional branch, conditional branch, indirect jump, return, halt/lifecycle boundary. Но в линейном assembler layout один successor может не иметь отдельной instruction: он достигается fall-through.

Это важно для качества generated code. Если у condition два successors, генератор кода обычно старается расположить один из них непосредственно после test block-а:

```
        ; false path is fall-through
        ICMP    ...
        JZ      .L_true
        ; false block
        ...
        JMP     .L_join
.L_true:
        ...
.L_join:
```

Иногда выгоднее инвертировать condition, чтобы fall-through был наиболее вероятным или наиболее простым path. Даже без branch prediction это уменьшает количество unconditional jumps и делает layout яснее.

3.9.7. CFG — не только картинка для учебника

Когда позднее появятся optimization passes, CFG станет рабочей структурой данных compiler-а. На нём естественно определяются:

- достижимость blocks;
- dominators и post-dominators;
- natural loops и back edges;
- liveness и data-flow equations;
- удаление unreachable code;
- упрощение branches и слияние blocks;
- размещение blocks перед окончательным emission.

В этой главе мы не будем превращать каждый пример в курс теории компиляторов. Но будем сознательно показывать CFG там, где он объясняет machine code лучше, чем последовательность строк. Это позволит позднее использовать тот же материал непосредственно при разработке собственного генератора кода MCPU.

3.10. Условный оператор: от семантики к assembler

Теперь можно вернуться к привычному `if`. Но полезно не воспринимать его как “особую инструкцию высокого уровня”. В генераторе кода это шаблон CFG, который затем раскладывается на comparison, conditional edge, fall-through и при необходимости unconditional jump.

3.10.1. if без else

Начнём с конструкции

```
if (a == b)
        x = 1;
```

Семантика проста: если condition ложна, body пропускается; после body и после ложного пути выполнение продолжается в одной точке. Поэтому нужен один join target:

```
        ICMP    r64@a, r64@b
        JNZ     .L_end

        LDI     r64@x, 1

.L_end:
        ...
```

Здесь condition инвертировано: source говорит “если равно — выполнить”, а machine branch говорит “если не равно — перепрыгнуть body”. Такая инверсия очень типична. Она позволяет true path сделать fall-through и не вводить отдельный jump в начало body.

Процедуру генератора кода можно записать почти буквально:

```
emit_if_without_else(cond, body):
        L_end = new_label()
        emit_condition_false_branch(cond, L_end)
        emit(body)
        place(L_end)
```

Это не syntax будущего конкретного compiler-a, а описание lowering algorithm.

3.10.2. if/else создаёт две альтернативы и join

Теперь рассмотрим:

```
if (x == 0)
        message = "zero";
else
        message = "nonzero";
```

Обе ветви должны привести к общей точке, в которой уже можно использовать выбранное значение. Типичный layout:

```
        ICMP    r64@x, r64@zero
        JNZ     .L_else

        ; then
        ...
        JMP     .L_join

.L_else:
        ; else
        ...

.L_join:
        ...
```

Один branch выбирает альтернативу, а дополнительный `JMP` нужен только потому, что then block физически расположен перед else block и не должен провалиться в него.

Полный runnable example находится в `examples/if-else`:

```
#include <sys/asm/syscall.h>
#include <sys/asm/unistd.h>

.data
```

```
msg_zero:    .utf8 "zero\n"
msg_nonzero: .utf8 "nonzero\n"

.text
_start_life:
        LDI     r64@0x00000, 0          ; x
        CLR     r64@0x00008             ; zero

        ; if (x == 0) ... else ...
        ICMP    r64@0x00000, r64@0x00008
        JNZ     .L_else

        LEA     r64@0x00020, msg_zero
        JMP     .L_join

.L_else:
        LEA     r64@0x00020, msg_nonzero

.L_join:
        PUSH    r64@0x00020
        CALL    print
        ADJSP   8
        HALT

print:
        ; Первый байтовый8- аргумент находится по SP+8:
        ; CALL уже положил return PC по SP+0.
        LDSP    r64@0x00100, 8
        MOV     r64@0x00108, r64@0x00100

.L_find_nul:
        LOADP   r8@0x00110, r64@0x00108
        CLR     r8@0x00111
        ICMP    r8@0x00110, r8@0x00111
        JZ      .L_write_string
        PADDI   r64@0x00108, r64@0x00108, 1
        JMP     .L_find_nul

.L_write_string:
        PDIFF   r64@0x00118, r64@0x00108, r64@0x00100

        ; write(STDOUT_FILENO, string, length)
        LDI     r32@0x7fff8, 3
        LDI     r32@0x7fffc, __NR_write
        LDI     r64@0x7fff0, STDOUT_FILENO
        MOV     r64@0x7ffe8, r64@0x00100
        MOV     r64@0x7ffe0, r64@0x00118
        SYSCALL
        RET
```

При x=0 результат:

```
zero
```

Если заменить initial value на ненулевое, будет выбран второй block.

3.10.3. Порядок blocks не обязан повторять порядок source

Source construct фиксирует семантику, но не обязует генератор кода располагать machine blocks exactly в том же порядке. Эти два варианта эквивалентны по смыслу:

```
; layout A: then is fall-through
ICMP    ...
JNZ     .L_else
; then
JMP     .L_join
.L_else:
; else
.L_join:
```

и

```
; layout B: else is fall-through
ICMP    ...
JZ      .L_then
; else
JMP     .L_join
.L_then:
; then
.L_join:
```

Выбор зависит от формы condition, размера blocks, ожидаемой частоты paths и последующих optimization passes. Уже здесь видно полезное правило compiler construction: **AST определяет semantics, CFG определяет control flow, block layout определяет физический порядок emission**.

3.10.4. Цепочка else if является деревом решений

Исходник

```
if (x < 0)
        A;
else if (x == 0)
        B;
else
        C;
```

не требует особой machine instruction “else-if”. Это последовательность decision blocks:

```
        ; test x < 0
        ... branch to A or next_test

next_test:
        ; test x == 0
        ... branch to B or C
```

Каждая новая ветвь появляется только на false path предыдущего condition. Все завершённые alternatives обычно сходятся к одному join.

Генератору кода удобно понижать такую source form рекурсивно: `else` сам может содержать другой `if`. Frontend даже может представить `else if` именно как nested AST node без отдельного syntactic kind.

3.10.5. Join block и значения, созданные в разных ветвях

Если обе ветви только печатают строку, join прост. Но если они вычисляют значение:

```
if (cond)
        x = 10;
else
        x = 20;
use(x);
```

генератор кода должен обеспечить, чтобы к join block значение `x` имело единое доступное представление. В прямом assembler это может быть один заранее выбранный register:

```
        ... condition ...
        JZ      .L_else
        LDI     r64@x, 10
        JMP     .L_join
.L_else:
        LDI     r64@x, 20
.L_join:
        ; r64@x содержит выбранное значение
```

Если промежуточное представление compiler-а использует SSA, та же ситуация обычно описывается phi-node в join block. MCPU ISA не имеет instruction `PHI`: phi является compile-time construct и должен исчезнуть при lowering/register allocation, превратившись в согласованные copies или выбранные locations.

Такой пример полезен тем, что показывает границу между compiler IR и ISA. Не вся операция, существующая в optimizer-е, обязана иметь одноимённую machine instruction.

3.10.6. Условия с side effects требуют сохранения порядка

На уровне простого comparison генератор кода свободно инвертирует branches и переставляет blocks, пока semantics не меняется. Но source condition может содержать вычисления с observable effects:

```
if (f() != 0)
        ...
```

или позднее:

```
if (p != NULL && *p == 42)
        ...
```

Тогда compiler обязан сохранить language-defined evaluation order и short-circuit rules. Оптимизация control flow не означает право выполнять выражение раньше, позже или лишний раз, если это меняет observable behavior.

Это особенно важно помнить при переходе от “рисования CFG” к реальному генератору кода компилятора: CFG должен выражать semantics, а не просто внешнюю форму source.

3.10.7. Semantic procedure для if/else

Полезный общий шаблон можно записать так:

```
emit_if(cond, then_stmt, else_stmt):
        L_then = new_block()
        L_else = new_block()
        L_join = new_block()

        emit_condition(cond, L_then, L_else)

L_then:
        emit(then_stmt)
        jump L_join

L_else:
        emit(else_stmt)
        jump/fall-through L_join

L_join:
```

После построения CFG отдельный layout pass может решить, какой block поставить рядом с test и какой explicit jump удалить за счёт fall-through. Это чище, чем пытаться одновременно решать semantics, register allocation и final instruction order в одной рекурсивной функции parser-a.

3.11. Логические выражения и short-circuit semantics

Условный оператор становится по-настоящему интересным, когда condition само состоит из нескольких частей. В языках семейства C операторы `&&` и `||` имеют не только логический результат, но и строго определённую **short-circuit** (*сокращённую*) семантику: правая часть вычисляется только тогда, когда без неё ещё нельзя определить результат.

Это означает, что boolean expression часто правильнее lowering-ить сразу в CFG, а не сначала вычислять все подвыражения в 0/1 и затем комбинировать их арифметически.

3.11.1. Логическое AND как граф

Рассмотрим

```
if (a != 0 && b < 10)
        A;
else
        B;
```

Если `a==0`, значение всего `&&` уже ложно. Проверять `b<10` не только бесполезно: в общем случае это может быть семантически запрещено, если вычисление `b` связано с функцией, volatile object или разыменованием указателя.

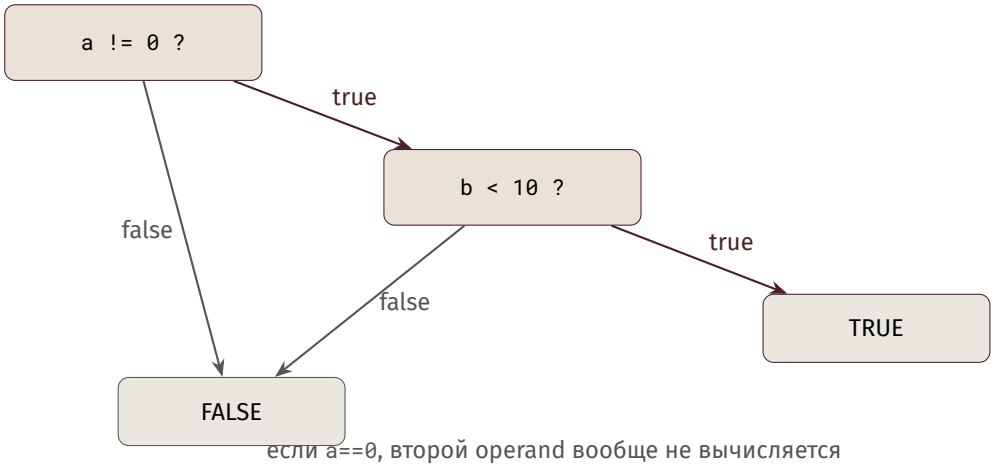


Рис. 3.11. Short-circuit CFG для `a != 0 && b < 10`.

Процедуру генератора кода выражается очень естественно:

```
emit_condition(A && B, L_true, L_false):
        L_rhs = new_block()
        emit_condition(A, L_rhs, L_false)
L_rhs:
        emit_condition(B, L_true, L_false)
```

Обратите внимание: здесь не возникло ни одного обязательного boolean register. Значением expression является сам выбор control-flow edge.

3.11.2. Логическое OR

Для `A || B` симметрия другая. Если A истинно, весь результат уже истинен; B требуется только на false path:

```
emit_condition(A || B, L_true, L_false):
        L_rhs = new_block()
        emit_condition(A, L_true, L_rhs)
L_rhs:
        emit_condition(B, L_true, L_false)
```

Именно такая recursive definition делает abstraction `emit_condition(expr,L_true,L_false)` особенно удобной. Relational leaf знает, как породить comparison и branch. Logical node знает, как соединить control destinations. Parser и semantic analyzer при этом не должны знать детали PSW.

3.11.3. Logical NOT обычно не требует instruction

В control context выражение

```
!A
```

часто lowering-ится вообще без arithmetic instruction:

```
emit_condition(!A, L_true, L_false):
        emit_condition(A, L_false, L_true)
```

То есть достаточно поменять targets местами. Это хороший пример оптимизации, которая следует прямо из semantic representation: logical negation в branch context может быть операцией над графом, а не над register value.

3.11.4. De Morgan как преобразование control flow

По той же причине законы De Morgan полезны генератору кода не только как абстрактная логика:

$$!(A \;\&\&\; B) \equiv !A \;||\; !B,$$
$$!(A \;||\; B) \equiv !A \;\&\&\; !B.$$

Но compiler не обязан буквально переписывать AST. Если branch API принимает true/false targets, многие такие преобразования получаются простой перестановкой edges.

3.11.5. Branch context и value context

Теперь сравним два source fragments:

```
if (a < b)
        f();
```

и

```
x = (a < b);
```

В первом случае достаточно направить управление в true/false blocks. Во втором language semantics требует получить обычное value, которое можно сохранить, передать функции или использовать позже.

Один из простейших способов materialization:

```
        ; condition branches to two value blocks
        ...
.L_false:
        LDI     r64@result, 0
        JMP     .L_join
.L_true:
        LDI     r64@result, 1
.L_join:
```

Здесь boolean value строится control flow-ом. Позднее optimizer может заменить такой diamond более эффективной ISA-specific sequence, если архитектура предоставляет подходящую predicate/set instruction. Но correctness сначала должна быть выражена независимо от optimization.

3.11.6. Почему нельзя заменять short-circuit обычным IAND

Наивный compiler мог бы попытаться сделать так:

```
va = materialize(A)
vb = materialize(B)
IAND result, va, vb
```

Для чистых выражений результат 0/1 может совпасть. Но evaluation semantics уже другая: B вычисляется всегда. Например,

```
p != NULL && *p == 42
```

при p==NULL обязано завершиться ложью без dereference. Правильный CFG делает pointer check dominator-ом блока с LOADP. Это уже не стилистика, а условие memory safety semantics исходного языка.

Аналогично:

```
index < n && array[index] == key
```

должно сначала доказать bounds condition, и только затем формировать address элемента и выполнять load. Хорошо построенный CFG буквально фиксирует этот порядок.

3.11.7. Полный пример short-circuit

Пример examples/short-circuit реализует

```
if (a != 0 && b < 10)
        print("A\n");
else
        print("B\n");
```

Здесь b рассматривается как unsigned integer, поэтому relation b<10 использует CF:

```
#include <sys/asm/syscall.h>
#include <sys/asm/unistd.h>

.data
msg_true:  .utf8 "A\n"
msg_false: .utf8 "B\n"

.text
_start_life:
        LDI     r64@0x00000, 1          ; a
        LDI     r64@0x00008, 7          ; unsigned b
        CLR     r64@0x00010             ; zero
        LDI     r64@0x00018, 10         ; limit

        ; if (a != 0 && b < 10)
        ICMP    r64@0x00000, r64@0x00010
        JZ      .L_false                ; short circuit: b is not evaluated

        ICMP    r64@0x00008, r64@0x00018
        JNC     .L_false                ; unsigned b >= 10

.L_true:
        LEA     r64@0x00020, msg_true
        JMP     .L_print
```

```
.L_false:
        LEA     r64@0x00020, msg_false

.L_print:
        PUSH    r64@0x00020
        CALL    print
        ADJSP   8
        HALT

print:
        ; Первый байтовый8- аргумент находится по SP+8:
        ; CALL уже положил return PC по SP+0.
        LDSP    r64@0x00100, 8
        MOV     r64@0x00108, r64@0x00100

.L_find_nul:
        LOADP   r8@0x00110, r64@0x00108
        CLR     r8@0x00111
        ICMP    r8@0x00110, r8@0x00111
        JZ      .L_write_string
        PADDI   r64@0x00108, r64@0x00108, 1
        JMP     .L_find_nul

.L_write_string:
        PDIFF   r64@0x00118, r64@0x00108, r64@0x00100

        ; write(STDOUT_FILENO, string, length)
        LDI     r32@0x7fff8, 3
        LDI     r32@0x7fffc, __NR_write
        LDI     r64@0x7fff0, STDOUT_FILENO
        MOV     r64@0x7ffe8, r64@0x00100
        MOV     r64@0x7ffe0, r64@0x00118
        SYSCALL
        RET
```

При `a=1`, `b=7` программа печатает:

```
A
```

Если поставить `a=0`, branch после первого `ICMP` сразу уйдёт в false block; второй comparison не будет выполнен.

3.11.8. Compiler context для boolean lowering

К этому месту полезно разделить интерфейс будущего генератора кода хотя бы концептуально:

```
emit_value(expr) -> location
emit_address(expr) -> PTR64 location
emit_condition(expr, L_true, L_false)
```

Одна AST node может поддерживать несколько contexts. Variable в value context загружается; в address context даёт lvalue address. Relational expression в branch context порождает edges; в value context materializes 0/1. Logical `&&`/`||` в branch context строят short-circuit CFG.

Такое разделение значительно лучше масштабируется, чем попытка сначала превратить каждое выражение в register value, а затем уже разбираться, зачем оно понадобилось.

3.12. Циклы: while, do while, for, break и continue

Цикл — это первый construct, в котором CFG получает **back edge** (*ребро, возвращающее управление к уже существующему блоку*). Именно back edge превращает одноразовое ветвление в повторение.

На уровне assembler нет отдельной инструкции `while` или `for`. Есть labels, comparisons и jumps. Но для генератора кода компилятора полезно различать формы циклов, потому что у них разные entry, test position и continue target.

3.12.1. while: сначала test, потом body

Высокоуровневая форма

```
while (cond) {
        body;
}
```

имеет естественный CFG:

```
.L_test:
        ; cond
        ... branch false -> .L_exit

.L_body:
        ; body
        ...
        JMP     .L_test

.L_exit:
```

Condition проверяется до первой итерации, поэтому body может не выполниться ни разу.

Алгоритм генератора кода:

```
emit_while(cond, body):
        L_test = new_block()
        L_body = new_block()
        L_exit = new_block()

        jump/fall-through L_test
L_test:
        emit_condition(cond, L_body, L_exit)
L_body:
        emit(body)
        jump L_test
L_exit:
```

После layout один из explicit jumps может исчезнуть как fall-through, но semantic graph остаётся таким.

3.12.2. do while: body является entry цикла

В конструкции

```
do {
        body;
} while (cond);
```

первая итерация выполняется без предварительного test. Поэтому CFG другой:

```
.L_body:
        ... body ...
.L_test:
        ... cond ...
        ; true -> .L_body
        ; false -> .L_exit
.L_exit:
```

Это различие нельзя потерять при lowering. Попытка mechanically преобразовать обе формы в один и тот же шаблон с test в начале меняет semantics для false initial condition.

3.12.3. for содержит четыре логические части

Классическая форма

```
for (init; cond; step) {
        body;
```

```
}
```

лучше всего понимается как CFG из четырёх частей:

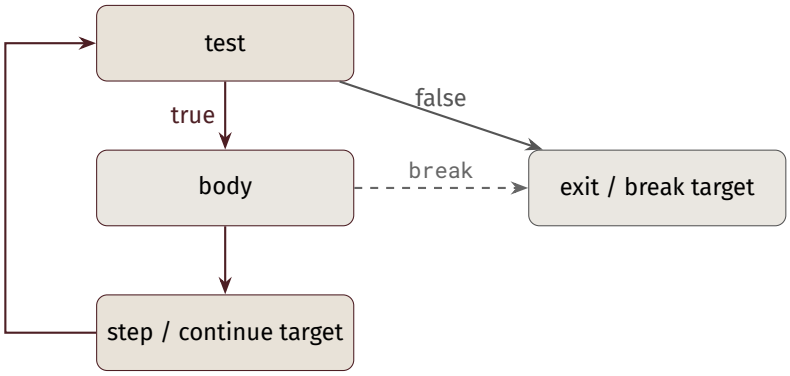


Рис. 3.12. Обобщённый CFG цикла; для `for` step является отдельным continue target.

Полный lowering:

```
emit(init)

.L_test:
        emit_condition(cond, .L_body, .L_exit)

.L_body:
        emit(body)

.L_step:
        emit(step)
        JMP     .L_test

.L_exit:
```

Если `cond` отсутствует, он трактуется как true. Если `step` пуст, блок всё равно полезно сохранить как logical continue target до последующего CFG simplification.

3.12.4. continue не всегда идёт в test

Это одна из деталей, которую легко упустить в самодельном compiler-е.

Для

```
while (cond) {
        ...
        continue;
}
```

`continue` действительно передаёт управление к test следующей итерации.

Но для

```
for (init; cond; step) {
        ...
        continue;
}
```

сначала обязан выполниться `step`, и только затем новый test. Поэтому targets различаются:

Loop kind	continue target	break target
`while`	test block	exit block
`do while`	bottom test block	exit block
`for`	step block	exit block

Это отличный аргумент в пользу compiler-side loop context вместо ad-hoc поиска ближайшего label.

3.12.5. Loop context как стек semantic state

Во время lowering удобно хранить структуру вида

```
LoopContext:
        break_target
        continue_target
```

При входе во вложенный цикл новый context помещается поверх предыдущего. При выходе снимается. Тогда source `break` и `continue` всегда относятся к ближайшему enclosing loop.

Например:

```
while (outer) {
        for (...) {
                if (x)
                        break;
        }
        continue;
}
```

Первый `break` использует exit внутреннего `for`. Последний `continue` после завершения `for` относится уже к внешнему `while`. Стек contexts выражает это напрямую.

Позднее `switch` добавит ещё один вид breakable context, и тогда semantic stack должен различать loop continue target и generic break target.

3.12.6. Форма условия и направление branch

Для loop test особенно важен выбор fall-through. Типичный `while` удобно layout-ить так, чтобы true path входил в body без дополнительного jump:

```
.L_test:
        ICMP    ...
        JZ      .L_exit      ; false condition
        ; body begins as fall-through
        ...
        JMP     .L_test
.L_exit:
```

Но если body огромен, exit очень короток или profiling information говорит, что loop почти никогда не выполняется, block layout pass может выбрать другую форму. Semantic lowering не должен быть привязан к одному физическому порядку blocks.

3.12.7. Минимальный runnable loop

Пример `examples/loops` держит counter в `r64`, печатает `"loop\n"` три раза, затем `"done\n"`:

```
#include <sys/asm/syscall.h>
#include <sys/asm/unistd.h>

.data
msg_loop: .utf8 "loop\n"
msg_done: .utf8 "done\n"

.text
_start_life:
        LDI     r64@0x00000, 3           ; i
        CLR     r64@0x00008              ; zero

.L_test:
        ICMP    r64@0x00000, r64@0x00008
        JZ      .L_done

        LEA     r64@0x00020, msg_loop
        PUSH    r64@0x00020
        CALL    print
        ADJSP   8

        IDEC    r64@0x00000, r64@0x00000
        JMP     .L_test

.L_done:
        LEA     r64@0x00020, msg_done
        PUSH    r64@0x00020
        CALL    print
```

```
        ADJSP   8
        HALT

print:
        ; Первый байтовый8- аргумент находится по SP+8:
        ; CALL уже положил return PC по SP+0.
        LDSP    r64@0x00100, 8
        MOV     r64@0x00108, r64@0x00100

.L_find_nul:
        LOADP   r8@0x00110, r64@0x00108
        CLR     r8@0x00111
        ICMP    r8@0x00110, r8@0x00111
        JZ      .L_write_string
        PADDI   r64@0x00108, r64@0x00108, 1
        JMP     .L_find_nul

.L_write_string:
        PDIFF   r64@0x00118, r64@0x00108, r64@0x00100

        ; write(STDOUT_FILENO, string, length)
        LDI     r32@0x7fff8, 3
        LDI     r32@0x7fffc, __NR_write
        LDI     r64@0x7fff0, STDOUT_FILENO
        MOV     r64@0x7ffe8, r64@0x00100
        MOV     r64@0x7ffe0, r64@0x00118
        SYSCALL
        RET
```

Ожидаемый вывод:

```
loop
loop
loop
done
```

Здесь intentionally используется явный `JMP .L_test`, хотя некоторые конкретные loops можно записать короче. Нам важен узнаваемый CFG pattern.

3.12.8. Массивный цикл и pointer induction variable

Позднее, после подробного раздела о массивах, типичный цикл

```
for (i = 0; i < n; ++i)
        sum += a[i];
```

может lowering-иться не только через integer index. Генератор кода вправе преобразовать его в

pointer induction variable:

```
p = &a[0]
end = &a[n]
while (p != end) {
        sum += *p
        p += sizeof(element)
}
```

На MCPU это естественно использует PTR64 и `PADDI/PADD`. Такая optimization становится корректной только после анализа типа элемента, размера, границ массива и pointer semantics. Поэтому сейчас мы фиксируем control-flow skeleton, а address arithmetic вернём в следующем крупном блоке о compound data.

3.12.9. Natural loop как compiler object

Для optimizer-а цикл уже не обязан совпадать с source construct. После inline, simplification или goto-like control flow CFG может содержать back edge, который образует natural loop независимо от исходного слова `while`. Поэтому зрелый compiler обычно имеет два уровня:

- frontend lowering знает source semantics `while/for/do`;
- CFG analysis позднее обнаруживает loops как graph structure.

Это разделение позволяет loop optimizations работать и с кодом, который возник после других transformations.

3.13. SWITCH: ВЫБОР МЕЖДУ МНОГИМИ ВЕТВЯМИ

Конструкция `switch` интересна для compiler writer тем, что один и тот же source construct не имеет единственного очевидного machine pattern. Для двух или трёх case разумна линейная цепочка comparisons. Для большого разреженного множества может оказаться лучше дерево решений. Для плотного диапазона — jump table.

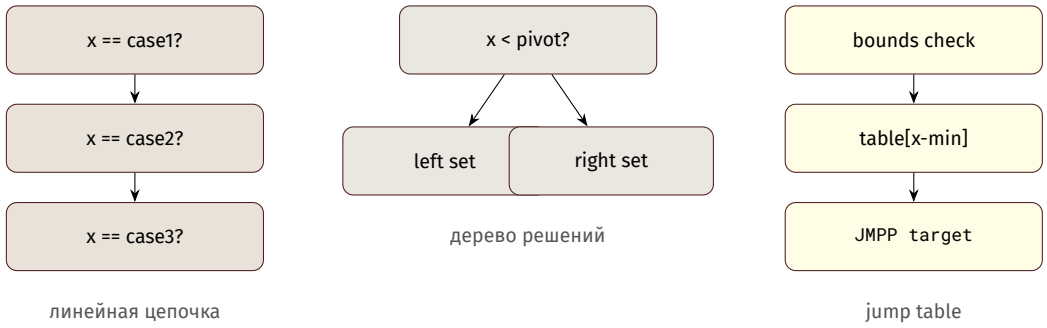


Рис. 3.13. Три типичных стратегии lowering конструкции switch.

Поэтому `switch` — хорошее место, чтобы окончательно отделить semantics от code-selection policy.

3.13.1. Семантика сначала, стратегия потом

Пусть source имеет вид:

```
switch (x) {
case 2:
        A;
        break;
case 3:
        B;
        break;
default:
        C;
}
```

Semantic analysis должен определить набор case values, проверить их допустимость и отсутствие duplicates, найти default и понять source fall-through между case labels. Только после этого генератор кода выбирает способ диспетчеризации.

Важно не путать два разных значения слова fall-through:

- *source switch fall-through* — намеренное продолжение из одного case body в следующий case body при отсутствии `break`;
- *machine block fall-through* — переход к следующему физически расположенному basic block без explicit jump.

Первое является частью semantics языка. Второе — свойство layout generated code. Compiler может сохранить source fall-through при совершенно другом физическом порядке blocks, если CFG edges остаются правильными.

3.13.2. Линейная цепочка comparisons

Для малого числа cases самый простой lowering:

```
        ICMP    r64@x, r64@case2
        JZ      .L_case_2
        ICMP    r64@x, r64@case3
        JZ      .L_case_3
        JMP     .L_default
```

Стоимость растёт линейно с количеством cases, но code size мал, branch structure очевидна и не требуется таблица данных.

Полный пример находится в `examples/switch-chain`:

```
#include <sys/asm/syscall.h>
#include <sys/asm/unistd.h>
```

```
.data
msg_2:       .utf8 "case 2\n"
msg_3:       .utf8 "case 3\n"
msg_default: .utf8 "default\n"

.text
_start_life:
        LDI     r64@0x00000, 3           ; x
        LDI     r64@0x00008, 2
        LDI     r64@0x00010, 3

        ICMP    r64@0x00000, r64@0x00008
        JZ      .L_case_2
        ICMP    r64@0x00000, r64@0x00010
        JZ      .L_case_3
        JMP     .L_default

.L_case_2:
        LEA     r64@0x00020, msg_2
        JMP     .L_print
.L_case_3:
        LEA     r64@0x00020, msg_3
        JMP     .L_print
.L_default:
        LEA     r64@0x00020, msg_default
.L_print:
        PUSH    r64@0x00020
        CALL    print
        ADJSP   8
        HALT

print:
        ; Первый байтовый8- аргумент находится по SP+8:
        ; CALL уже положил return PC по SP+0.
        LDSP    r64@0x00100, 8
        MOV     r64@0x00108, r64@0x00100

.L_find_nul:
        LOADP   r8@0x00110, r64@0x00108
        CLR     r8@0x00111
        ICMP    r8@0x00110, r8@0x00111
        JZ      .L_write_string
        PADDI   r64@0x00108, r64@0x00108, 1
        JMP     .L_find_nul

.L_write_string:
        PDIFF   r64@0x00118, r64@0x00108, r64@0x00100

        ; write(STDOUT_FILENO, string, length)
        LDI     r32@0x7fff8, 3
        LDI     r32@0x7fffc, __NR_write
        LDI     r64@0x7fff0, STDOUT_FILENO
```

```
        MOV     r64@0x7ffe8, r64@0x00100
        MOV     r64@0x7ffe0, r64@0x00118
        SYSCALL
        RET
```

При x=3 он выводит:

```
case 3
```

Для небольшого switch такая реализация может быть оптимальной даже при наличии indirect jump instruction.

3.13.3. Дерево решений для разреженного множества

Представим cases

```
1, 7, 31, 400, 9000, 1000000
```

Jump table от 1 до 1000000 была бы абсурдно разреженной. Линейная цепочка в худшем случае требует шесть comparisons. Если набор достаточно велик, генератор кода может отсортировать case values и строить binary decision tree:

```
if (x < 400)
        search {1,7,31}
else if (x > 400)
        search {9000,1000000}
else
        case 400
```

На MCPU signedness controlling expression определяет, какая relation нужна для tree branches. Для unsigned switch используется CF/ZF logic, для signed — SF/OF/ZF logic, рассмотренная ранее.

Compiler heuristic может учитывать не только число cases, но и profile probabilities. Частый case иногда выгодно проверить первым, даже если идеальное сбалансированное дерево имело бы другую форму.

3.13.4. Jump table для плотного диапазона

Если case values покрывают небольшой плотный диапазон, полезна таблица PROGRAM pointers. Общая схема:

```
index = x - min_case
if (index >= table_size)
```

```
        goto default

target = jump_table[index]
goto *target
```

На MCPU target является PTR64(PROGRAM,...), поэтому естественный indirect branch — `JMPP`.

Таблица может быть описана директивой `.cptr`:

```
.data
.align 8
jump_table:
        .cptr   .L_case_0
        .cptr   .L_case_1
        .cptr   .L_case_2
```

Каждая `.cptr` создаёт 8-byte architectural PTR64 object, указывающий в PROGRAM. Это именно DATA object в target representation, а не CODE64 field внутри instruction. Если target address окончательно неизвестен в ET_REL, assembler/linker сохраняют необходимую relocation information.

После bounds check код формирует адрес элемента таблицы, загружает PTR64 и выполняет indirect jump:

```
        ; r64@index содержит 0..2
        ; offset = index * 8
        IADD    r64@offset, r64@index, r64@index
        IADD    r64@offset, r64@offset, r64@offset
        IADD    r64@offset, r64@offset, r64@offset

        LEA     r64@table, jump_table
        PADD    r64@entry, r64@table, r64@offset
        LOADP   r64@target, r64@entry
        JMPP    r64@target
```

В рабочем генераторе кода умножение на 8 может быть реализовано более подходящей shift/strength-reduction sequence. Здесь три `IADD` выбраны только для прозрачности: мы буквально строим 2x, 4x, 8x.

3.13.5. Bounds check обязателен

Jump table нельзя индексировать непроверенным source value. Если `x` находится вне case range, вычисленный address может выйти за таблицу и загрузить произвольные восемь байтов как code pointer.

Для unsigned index и table size 3:

```
        LDI     r64@limit, 3
        ICMP    r64@index, r64@limit
        JNC     .L_default       ; index >= 3
```

После этого доказано `0 <= index < 3`, и address calculation безопасна относительно логического размера таблицы.

Если исходный switch signed и `min_case` не ноль, генератор кода обычно сначала нормализует значение `x-min_case`, затем выполняет такой unsigned range check. Это классический приём: после subtract диапазон `min..max` превращается в `0..range`, а отрицательные исходные значения после unsigned interpretation автоматически оказываются выше допустимого range.

3.13.6. Runnable jump-table example

Полный пример `examples/switch-table` использует три entries и `JMPP`:

```
#include <sys/asm/syscall.h>
#include <sys/asm/unistd.h>

.data
.align 8
jump_table:
        .cptr   .L_case_0
        .cptr   .L_case_1
        .cptr   .L_case_2
msg_0:       .utf8 "case 0\n"
msg_1:       .utf8 "case 1\n"
msg_2:       .utf8 "case 2\n"
msg_default: .utf8 "default\n"

.text
_start_life:
        LDI     r64@0x00000, 2           ; unsigned x
        LDI     r64@0x00008, 3           ; number of table entries

        ; Bounds check: x >= 3 -> default.  For unsigned compare, CF=0 means x>=3.
        ICMP    r64@0x00000, r64@0x00008
        JNC     .L_default

        ; byte_offset = x * 8
        IADD    r64@0x00010, r64@0x00000, r64@0x00000
        IADD    r64@0x00010, r64@0x00010, r64@0x00010
        IADD    r64@0x00010, r64@0x00010, r64@0x00010

        LEA     r64@0x00018, jump_table
        PADD    r64@0x00018, r64@0x00018, r64@0x00010
        LOADP   r64@0x00020, r64@0x00018
        JMPP    r64@0x00020
```

```
.L_case_0:
        LEA     r64@0x00028, msg_0
        JMP     .L_print
.L_case_1:
        LEA     r64@0x00028, msg_1
        JMP     .L_print
.L_case_2:
        LEA     r64@0x00028, msg_2
        JMP     .L_print
.L_default:
        LEA     r64@0x00028, msg_default

.L_print:
        PUSH    r64@0x00028
        CALL    print
        ADJSP   8
        HALT

print:
        ; Первый байтовый8- аргумент находится по SP+8:
        ; CALL уже положил return PC по SP+0.
        LDSP    r64@0x00100, 8
        MOV     r64@0x00108, r64@0x00100

.L_find_nul:
        LOADP   r8@0x00110, r64@0x00108
        CLR     r8@0x00111
        ICMP    r8@0x00110, r8@0x00111
        JZ      .L_write_string
        PADDI   r64@0x00108, r64@0x00108, 1
        JMP     .L_find_nul

.L_write_string:
        PDIFF   r64@0x00118, r64@0x00108, r64@0x00100

        ; write(STDOUT_FILENO, string, length)
        LDI     r32@0x7fff8, 3
        LDI     r32@0x7fffc, __NR_write
        LDI     r64@0x7fff0, STDOUT_FILENO
        MOV     r64@0x7ffe8, r64@0x00100
        MOV     r64@0x7ffe0, r64@0x00118
        SYSCALL
        RET
```

При x=2 вывод:

```
case 2
```

Этот пример связывает сразу несколько ранее разрозненных механизмов: DATA objects, `.cptr`, PTR64 arithmetic, `LOADP`, indirect control transfer и bounds check.

3.13.7. Как генератор кода выбирает стратегию

Нет универсального порога “после N cases всегда jump table”. Реальный compiler может оценивать:

- количество cases;
- диапазон `max-min`;
- плотность `count/(max-min+1)`;
- стоимость table bytes;
- стоимость comparisons и branches;
- наличие profile information;
- ограничения relocation/PIC model;
- предпочтение code size или execution speed.

Поэтому semantic IR для `switch` лучше некоторое время хранить как “value + mapping case->block + default”, а конкретную dispatch strategy выбирать на более позднем этапе генерации кода.

3.13.8. break в switch и stack control contexts

`break` внутри `switch` передаёт управление к switch exit. Это похоже на loop break, но `continue` относится только к enclosing loop.

Удобная compiler model:

```
BreakContext:
        break_target

LoopContext extends BreakContext:
        continue_target
```

Тогда вложенная конструкция

```
while (...) {
        switch (x) {
        case 1:
                break;      ; выходит только из switch
        }
        continue;           ; относится к while
}
```

обрабатывается стеком contexts без специальных поисков по source text.

3.13.9. От switch к следующему крупному этапу

После этого раздела мы уже умеем строить нетривиальный CFG и выбирать между direct и indirect control transfer. Следующий естественный объект исследования — данные сложнее одного scalar: массивы, pointer indexing, структуры, alignment, block copy и строки как объекты алгоритма.

Это важная граница главы. До неё мы в основном изучали *куда* идёт управление. После неё будем одновременно рассуждать о control flow и layout данных, а затем уже подойдём к полноценному syscall/runtime layer.

3.14. Функции

В предыдущих разделах мы уже использовали `CALL`, `RET` и простые блоки аргументов. Теперь соберём эти отдельные правила в единое соглашение о вызовах. Для ассемблерной программы соглашение о вызовах — не синтаксическое украшение, а договор о том, кто создаёт стековый кадр, где лежат параметры, где появляется результат и какое состояние после вызова нельзя считать сохранённым.

3.14.1. Граница вызова

Перед `CALL` вызывающая программа резервирует и заполняет блок аргументов. Размер блока округляется вверх до 16 байтов, а SP перед вызовом удовлетворяет 16-байтовому выравниванию. `CALL` дополнительно помещает на STACK 64-битный адрес возврата. Поэтому при входе в функцию первый явный аргумент находится по `SP+8`. После `RET` освобождение блока аргументов остаётся обязанностью вызывающей программы.

```
ADJSP   -16
STSP    0, r64@arg
CALL    function
ADJSP   16
```

Это правило одинаково для целых, вещественных и указательных параметров: CPU переносит байты, а смысл и размер ячейки определяются соглашением вызывающей программы и вызываемой функции.

3.14.2. Возвращаемое значение

Скалярный результат передаётся через верхний конец register file. Каноническая 64-битная форма — `r64@0x7fff8`; 32-битная — `r32@0x7fffc`, 16-битная — `r16@0x7fffe`, 8-битная — `r8@0x7ffff`. Более широкий результат занимает соответствующее представление, прижатое старшим концом к концу register file. Область результата является изменяемой через вызов: следующий `CALL` вправе использовать её заново.

Для агрегатного результата удобна иная схема: вызывающая программа выделяет объект результата и передаёт скрытый PTR64 на него первым аргументом. Такой приём позднее станет естественным для структур и больших численных объектов.

3.14.3. Frame Pointer: неподвижная система координат кадра

Если функция только читает параметры и немедленно возвращается, относительных адресов от SP достаточно. Но как только она резервирует локальные объекты, выполняет `PUSH`, со-

здаёт динамические области или вызывает другие функции, SP начинает двигаться. FP нужен как неподвижная система координат одного вызова.

Канонический пролог:

```
PUSHFP
SETFP
ADJSP   -fixed_frame_size
```

После `PUSHFP; SETFP` сохранённый FP лежит по `FP+0`, адрес возврата — по `FP+8`, первый явный аргумент — по `FP+16`; фиксированные локальные объекты обычно размещаются по отрицательным смещениям.



Рис. 3.14. Стековый кадр функции с Frame Pointer.

Доступ к фиксированным объектам выполняют `LDFP`, `STFP` и `LEAFP`. Даже если SP временно сдвинут ниже, адрес `FP-8` продолжает обозначать тот же локальный объект.

Канонический эпилог:

```
RESTSP
POPFP
RET
```

`RESTSP` уничтожает всю динамическую часть ниже FP одним присваиванием `SP=FP`; `POPFP` восстанавливает FP вызывающей программы; `RET` снимает адрес возврата.

3.14.4. Динамический локальный объект

Следующий фрагмент показывает главное достоинство FP. Фиксированный локальный объект остаётся доступным по `FP-8`, хотя SP дополнительно сдвинут на 24 байта:

```
PUSHFP
SETFP
ADJSP   -16
STFP    -8, r64@value

LEASP   r64@mark, 0
LDI     r64@size, 24
PSUB    r64@tmp, r64@mark, r64@size
SETSP   r64@tmp

LDFP    r64@fixed, -8

SETSP   r64@mark
RESTSP
POPFP
RET
```

Здесь `mark` играет роль отметки вложенной динамической области. Подобная техника нужна для временных массивов, областей переменного размера и реализации некоторых конструкций языка высокого уровня.

3.14.5. PSW и рабочие регистры через границу CALL

Сам `CALL` сохраняет адрес возврата и не является арифметической операцией, однако вызываемая функция свободна менять рабочие регистры и флаги PSW. Поэтому нельзя вычислить условие, вызвать неизвестную функцию и после возврата продолжить использовать прежний ZF, CF или OF. Значение, необходимое после вызова, вызывающая программа должна сохранить в памяти или в области, которую её собственное соглашение считает сохранной.

3.14.6. Косвенный вызов

`CALLP r64@ptr` вызывает функцию по PTR64 пространства PROGRAM. Это машинная основа указателей на функции, таблиц диспетчеризации и callback-функций. Стековый ABI после `CALLP` тот же, что после прямого `CALL`: адрес возврата находится по `SP+0`, аргументы начинаются с `SP+8`.

3.14.7. Функция как объект межмодульного контракта

Когда функция используется только внутри одного объекта, её символ может оставаться LOCAL. Для вызова из другого ET_REL символ экспортируется через `.global`; вызывающий модуль объявляет внешний символ через `.extern`. Само соглашение о вызове при этом не меняется. Таким образом CALL ABI связывает не только две функции, но и отдельно собранные объектные файлы, статические архивы и динамические библиотеки; к этому мы вернёмся в §3.16.

3.15. Системные вызовы

Инструкция SYSCALL является низкоуровневой границей между guest-кодом MCPU и supervisor. Она не принимает операнды в тексте инструкции: номер вызова, число аргументов и сами аргументы заранее размещаются в фиксированной области register file.

3.15.1. Фиксированный регистровый ABI

Перед SYSCALL используются следующие представления:

Назначение	Представление	Смысл
NARGS	r32@0x7fff8	число аргументов
SYSNO	r32@0x7fffc	номер системного вызова
ARG0	r64@0x7fff0	первый аргумент
ARG1	r64@0x7ffe8	второй аргумент
ARG2	r64@0x7ffe0	третий аргумент
ARG3...	ниже	последующие 64-битные ячейки ABI
RET	r64@0x7fff8	результат после SYSCALL

NARGS и RET физически перекрываются: до вызова младшие 32 бита области содержат число аргументов, после вызова вся 64-битная область содержит результат. Поэтому NARGS нельзя считать сохранённым состоянием.

Пример write(STDOUT_FILENO, string, length):

```
#include <sys/asm/syscall.h>
#include <sys/asm/unistd.h>

LDI     r32@0x7fff8, 3
LDI     r32@0x7fffc, __NR_write
LDI     r64@0x7fff0, STDOUT_FILENO
MOV     r64@0x7ffe8, r64@string_ptr
MOV     r64@0x7ffe0, r64@length
SYSCALL
```

3.15.2. Результат и ошибки

Обычный системный вызов при успехе возвращает своё неотрицательное или типоспецифичное значение, а при ошибке — отрицательный код стабильного MCPU errno ABI. Это именно TARGET-значения, а не номера errno host-системы. Заголовок <sys/asm/errno.h> фиксирует пространство ошибок; host-ошибка преобразуется внутри LibMCPU на границе sysdeps.

Особое исключение — SYS_brk: он возвращает фактический текущий PTR64 пространства HEAP и при невозможном запросе оставляет прежнюю границу.

3.15.3. Заголовки `include/sys/asm`

Низкоуровневая программа обычно проходит через `mcpu-cpp` и получает публичный ABI из target sysroot. Канонический набор включает:

- `syscall.h` — номера системных вызовов;
- `unistd.h` — стандартные дескрипторы, `SEEK_*`, `R_OK/W_OK/X_OK/F_OK`;
- `fcntl.h` — `O_*`, `F_*`, режимы доступа;
- `types.h` — фиксированные ширины внешнего ABI;
- `stat.h` — размер и смещения полей MCPU stat;
- `dirent.h` — layout записи каталога;
- `select.h` — фиксированное множество файловых дескрипторов;
- `time.h` — `timeval`, `timespec`, clock IDs;
- `errno.h` — системные ошибки MCPU;
- `mpu-errno.h` — независимое состояние ошибок Math Processor Unit.

Важно не смешивать два пространства ошибок: отрицательный результат `SYSCALL` относится к системному ABI, а `mpu-errno.h` описывает математическое состояние LibMPU.

3.15.4. Полная таблица реализованных номеров

Таблица ниже соответствует публичному `sys/asm/syscall.h` текущего окружения книги.

Номер	Имя	Группа
0x00a	`__NR_brk`	управление границей HEAP
0x01c	`__NR_gettimeofday`	время
0x021	`__NR_clock_getres`	время
0x022	`__NR_clock_gettime`	время
0x030	`__NR_nanosleep`	ожидание
0x031	`__NR_clock_nanosleep`	ожидание
0x100	`__NR_read`	дескрипторный ввод
0x101	`__NR_write`	дескрипторный вывод
0x102	`__NR_lseek`	позиция файла
0x103	`__NR_open`	открытие файла
0x104	`__NR_creat`	создание файла
0x105	`__NR_close`	закрытие дескриптора
0x106	`__NR_link`	жёсткая ссылка
0x107	`__NR_unlink`	удаление имени
0x108	`__NR_chdir`	текущий каталог
0x109	`__NR_mkdir`	создание каталога
0x10b	`__NR_chmod`	права доступа
0x10c	`__NR_chown`	владелец
0x10f	`__NR_access`	проверка доступа
0x111	`__NR_rename`	переименование
0x112	`__NR_rmdir`	удаление каталога
0x113	`__NR_symlink`	символическая ссылка

Номер	Имя	Группа
0x114	`__NR_readlink`	чтение ссылки
0x115	`__NR_stat`	состояние файла
0x116	`__NR_fstat`	состояние по fd
0x117	`__NR_lstat`	состояние ссылки
0x119	`__NR_fcntl`	операции над fd
0x11a	`__NR_pipe2`	канал
0x11d	`__NR_getdents`	каталог
0x11e	`__NR_select`	ожидание fd
0x120	`__NR_fsync`	синхронизация
0x121	`__NR_truncate`	изменение размера
0x122	`__NR_ftruncate`	изменение размера по fd
0x123	`__NR_fchmod`	права по fd
0x124	`__NR_fchown`	владелец по fd
0x125	`__NR_fchdir`	каталог по fd
0x126	`__NR_dup`	дублирование fd
0x127	`__NR_dup2`	дублирование в заданный fd

3.15.5. Классы системных вызовов

Практически таблицу удобнее запоминать не по номерам, а по задачам: управление HEAP; время и ожидание; потоковый ввод-вывод; работа с файлами и каталогами; метаданные; управление дескрипторами; мультиплексирование. В каталоге `mcpu-examples/examples/syscalls/` каждому такому классу соответствуют минимальные программы и скрипты запуска. README этого каталога является подробным пользовательским описанием ABI и хорошим дополнением к настоящему разделу.

3.15.6. Память guest как аргумент

Указатель системного вызова всегда является PTR64 MCPU. Supervisor проверяет пространство, направление доступа и диапазон до обращения к host API. Поэтому `read` требует доступную для записи guest-область, а `write` — доступную для чтения. Host pointer, host `struct stat` или host `time_t` через ABI не передаются.

3.15.7. Дескрипторы

Файловый дескриптор guest также не является host fd. LibMCPU поддерживает отдельное пространство guest descriptors и отображает их на внутренние host-ресурсы. Поэтому значения `0`, `1`, `2` определены ABI как stdin/stdout/stderr, но последующие guest fd нельзя трактовать как номер дескриптора процесса `mcpu-run`.

3.16. Статическая и динамическая компоновка

До сих пор большинство примеров помещалось в одном исходном файле. Реальная программа почти всегда состоит из нескольких единиц трансляции. MCPU использует MELF ET_REL для объектных файлов, архивы `.a` для статических библиотек и MELF ET_DYN для динамических библиотек.

3.16.1. LOCAL, GLOBAL, WEAK и неопределённый символ

Метка по умолчанию локальна для объектного файла. Директива `.global` (синоним `.globl`) публикует определение. `.extern` объявляет, что определение должно быть найдено при компоновке. `.weak` создаёт слабое определение, которое может уступить сильному GLOBAL.

```
; add-one.s
.global add_one
.text
add_one:
        LDSP    r64@x, 8
        LDI     r64@one, 1
        IADD    r64@0x7fff8, r64@x, r64@one
        RET
```

```
; main.s
.extern add_one
.text
_start_life:
        ...
        CALL    add_one
        ...
```

Ассемблер не обязан знать конечный адрес `add_one`. В ET_REL остаётся relocation, а окончательное значение назначает `mcpu-ld`.

3.16.2. Relocatable link

Команда `mcpu-ld -r` объединяет несколько ET_REL в новый ET_REL, не создавая исполняемую программу. Это удобно для укрупнения модулей и предварительной компоновки:

```
mcpu-as -o main.o main.s
mcpu-as -o add-one.o add-one.s
mcpu-ld -r -o combined.o main.o add-one.o
```

Символы и relocations, которые ещё нужны будущей окончательной компоновке, сохраняются.

3.16.3. Статический архив .a

mcpu-ar создаёт обычный deterministic UNIX archive из ET_REL и индексирует экспортируемые символы. При поиске -lchoice linker извлекает из архива только те members, которые нужны для разрешения текущих неопределённых символов.

```
mcpu-ar rcs libchoice.a add-one.o add-two.o
mcpu-ld --static -L. -o program main.o -lchoice
```

Опция –static делает выбор библиотеки однозначно статическим, если в каталоге одновременно лежат libchoice.a и libchoice.so.

3.16.4. Динамическая библиотека ET_DYN

Shared object строится из подходящего ET_REL:

```
mcpu-ld --shared --soname=libmath.so.1 -o libmath.so.1.0.0 math.o
ln -s libmath.so.1.0.0 libmath.so.1
ln -s libmath.so.1 libmath.so
mcpu-ld --dynamic -L. -o program main.o -lmath
```

В executable появляется зависимость NEEDED и SONAME библиотеки. При запуске loader разрешает библиотеку через активные пути поиска. mcpu-run –library-path=. позволяет добавить каталог перед системным путём.

3.16.5. Область символов и interposition

Для одинаковой силы определения порядок области видимости динамического linker-а определяет, какое GLOBAL будет выбрано. Явные preload-библиотеки имеют специальную область interposition; среди preload одинаковой силы более правое определение имеет более высокий приоритет. WEAK уступает GLOBAL.

Этот механизм полезен для подмены библиотечных функций, но не должен превращаться в скрытую зависимость программы: нормальный executable обязан иметь корректный набор NEEDED и работать без отладочного preload.

3.16.6. Ограничения ET_DYN

Динамическая библиотека владеет обычными кодом и данными: .text, .data, .bss. Ресурсы жизненного цикла исполняемой программной модели — .init, .fini, .control, .input, .output, .trace, а также _start/_life/_fini — принадлежат главному ET_EXEC. Поэтому, например, таблицы коэффициентов RK4 в librk4.so располагаются в .data.

3.16.7. Инструменты проверки

После компоновки важно не гадать о результате, а смотреть объект:

```
mcpu-readelf -a program
mcpu-nm program
mcpu-objdump -d program
mcpu-readelf -d program
```

`mcpu-readelf` показывает sections, symbols, RELA и dynamic metadata; `mcpu-nm` удобен для таблицы символов; `mcpu-objdump` связывает символы с дизассемблированным PROGRAM.

3.17. Программные модели

MCPU допускает два низкоуровневых способа жизни исполняемого ET_EXEC. Обычная программа может начать и закончить работу одним непрерывным потоком, а программная модель может иметь отдельную фазу INIT, повторяемые фазы LIFE и FINI. В этом разделе речь идёт только о низкоуровневых символах `_start`, `_life`, `_fini` и секциях MELF; автоматический CRT языков высокого уровня будет рассмотрен в следующей главе.

3.17.1. Целые, вещественные и комплексные вычисления LibMPU

Register file MCPU допускает представления произвольной поддерживаемой ширины, а арифметика LibMPU работает с Integer, Real и Complex. Поэтому численная программа не обязана сводить вычисления к native host `double`. Например, `r128`, `r4096` и более широкие Real представления являются обычными операндами MCPU.

Для точных известных математических значений используется `RCONST`: значение строится непосредственно в точности регистра назначения. Это особенно важно для `PI`, `E`, `SQRT2` и других констант, где промежуточное округление к binary64 было бы нежелательно.

```
RCONST  r128@x, PI_6
RCONST  r128@y, ZERO
RSIN    r128@s, r128@x
RMUL    r128@z, r128@s, r128@y
```

Комплексные инструкции используют представления `c<bits>`; полный физический размер complex-объекта содержит две Real-компоненты половинной ширины.

3.17.2. Массивы численных данных и вычислительные циклы

Ассемблер не имеет отдельного runtime-типа «массив». Непрерывная последовательность `.real 128` — это просто область DATA, а индексирование реализуется арифметикой PTR64. Для Real128 шаг одного элемента равен 16 байтам.

```
LEA     r64@p, vector
LDI     r64@step, 16
```

```
LOADP   r128@x, r64@p
PADD    r64@p, r64@p, r64@step
```

Такой pointer-walking часто выгоднее повторного вычисления `base+i*16`. Именно этот стиль используется в численной библиотеке RK4.

3.17.3. Подпрограмма численного метода

Законченный пример книги использует классический явный Runge–Kutta четвёртого порядка. Библиотека экспортирует число стадий `NST_RK4` и функцию интегрирования `dsmint`. Состояние `f`, временные стадии `X` и callback правой части передаются через обычный CALL ABI.

```
; dsmint(h, f, X, part_right, N, NS)
ADJSP   -64
STSP    0,  r128@h
STSP    16, r64@fptr
STSP    24, r64@xptr
STSP    32, r64@rhs
STSP    40, r64@n
STSP    48, r64@ns
CALL    dsmint
ADJSP   64
```

Адрес `part_right` является PROGRAM PTR64; внутри integrator-a callback вызывается через `CALLP`. Тем самым один и тот же численный метод не знает физический закон конкретной модели.

3.17.4. PROGRAM и MODEL как две модели выполнения

Если секции `.init` нет, `_start` и `_life` находятся по одному адресу в `.text`. Удобная assembler-запись — `_start_life:`. Это обычная программа: первое выполнение начинается там же, где находится её LIFE-точка.

Для модели с отдельной инициализацией:

```
.init
_start:
        ; подготовка модели
        END

.text
_life:
.L_life_loop:
        ; один шаг
        END
        JMP     .L_life_loop
```

```
.fini
_fini:
        ; завершение
        END
```

END в INIT возвращает управление supervisor-у на границе фазы. Следующий запуск LIFE начинается с _life; после LIFE-END исполнение в следующий раз продолжается с инструкции после этого END, поэтому явный JMP обычно замыкает цикл жизненного шага. EXIT объявляет завершение executable и при нормальном жизненном цикле переводит выполнение к FINI, если она существует.

3.17.5. Модель физического маятника

Каталог mcpu-examples/examples/pendulum/ содержит законченный низкоуровневый пример. Он не использует функции init()/life()/main() языка высокого уровня и не зависит от автоматического выбора CRT: весь lifecycle выражен непосредственно через _start, _life, _fini.

CONTROL содержит параметры эксперимента. INIT копирует необходимые параметры в DATA, вычисляет коэффициенты ν и Ω, задаёт начальное состояние и публикует первый TRACE. LIFE увеличивает номер шага, вызывает RK4, обновляет время и публикует следующую строку телеметрии. FINI устанавливает диагностический marker.

Особенно важна граница ответственности: после INIT динамика использует снимок параметров из DATA, поэтому позднее изменение CONTROL не может незаметно изменить уже инициализированную модель.

Внешняя библиотека librk4.so демонстрирует реальный динамический контракт: её таблицы Бутчера лежат в DATA ET_DYN, наружу экспортируются только интерфейсные символы, а внутренние helpers остаются локальными.

При n=20 получается 21 TRACE-состояние: начальное состояние из INIT и двадцать результатов LIFE. Этот пример объединяет почти все низкоуровневые темы главы: Real128, PTR64, массивы данных, CALL/CALLP, динамическую библиотеку, CONTROL, TRACE и lifecycle.

Справочник по языку ассемблера

Этот раздел предназначен как настольная памятка к материалу главы. Он не заменяет нормативное описание ISA и MELF ABI, но позволяет быстро восстановить форму инструкции, назначение директивы, смысл флага или имя математической константы, не разыскивая соответствующее место в основном тексте.

Обозначения операндов

Обозначение	Смысл
R	целочисленное или вещественное представление r<bits>@addr
C	комплексное представление c<bits>@addr
r64(ptr)	64-битное представление регистра, содержащее архитектурный PTR64
ADDR64	адрес/смещение, допустимое конкретным классом инструкции
PTR64	сегментированный указатель SEG4:OFF60
CODE64	прямой адрес инструкции в PROGRAM
OFF64	знаковое 64-битное байтовое смещение
CONST8	селектор предопределённой константы RCONST
PSWBIT8	селектор отдельного бита PSW для JSET/JCLR

В таблицах ниже Rdst/Cdst обозначают регистр результата, Rsrc/Csrc — источник, а Ra, Rb — два операнда одинакового класса. Реальные ограничения ширины и допустимости конкретной комбинации остаются частью ISA-контракта.

Системные инструкции и перемещение данных

Мнемоника	Форма	Назначение
NOP	—	не выполняет действий и переходит к следующей инструкции.
HALT	—	останавливает нормальное выполнение процессора.
END	—	завершает текущую внешне управляемую фазу INIT/LIFE/FINI.
EXIT	r32	завершает программу или модель со знаковым 32-битным статусом.
SYSCALL	—	выполняет системный вызов через фиксированный регистровый ABI.
MOV	Rdst, Rsrc	копирует значение между совместимыми представлениями регистров.
CLR	R	обнуляет весь объект выбранного регистра.
SWAP	Ra, Rb	меняет местами два совместимых регистровых объекта.
BSWAP	Rdst, Rsrc	копирует байты источника в обратном порядке.
LDI	R, imm64	загружает знаковую целую непосредственную константу Integer64.
LDR	R, real64	загружает вещественную непосредственную константу с полем binary64.
LDC	C, re32, im32	создаёт комплексную непосредственную константу из двух binary32-компонент.
LOAD	R/C, DATA	загружает объект непосредственно из пространства DATA.
STORE	DATA, R/C	сохраняет регистровый объект непосредственно в DATA.

Адресация, PTR64 и блочные операции

Мнемоника	Форма	Назначение
LEA	r64, address	формирует полный PTR64 по символическому адресу объекта или кода.
LEASP	r64, off64	формирует PTR64(STACK, SP+off64).
LOADP	R/C, r64(ptr)	загружает объект по PTR64, находящемуся в 64-битном регистре.
STOREP	r64(ptr), R/C	сохраняет объект по PTR64.
PADD	r64dst, r64ptr, r64delta	прибавляет знаковое байтовое смещение из регистра к PTR64.
PADDI	r64dst, r64ptr, off64	прибавляет непосредственное знаковое байтовое смещение к PTR64.
PSUB	r64dst, r64ptr, r64delta	вычитает знаковое байтовое смещение из PTR64.
PSUBI	r64dst, r64ptr, off64	вычитает непосредственное знаковое байтовое смещение из PTR64.
PDIFF	r64dst, r64a, r64b	вычисляет разность двух совместимых PTR64 в байтах.
BMOV	r64dst, r64src, r64count	копирует count байтов между областями памяти с семантикой memmove.
BMOVI	r64dst, r64src, imm64	копирует непосредственное число байтов между областями памяти.

Целая арифметика и работа с битами

Мнемоника	Форма	Назначение
IADD	Rdst, Ra, Rb	целое сложение.
IADC	Rdst, Ra, Rb	сложение с входным переносом CF.
ISUB	Rdst, Ra, Rb	целое вычитание.
ISBB	Rdst, Ra, Rb	вычитание с заёмом через CF.
IAND	Rdst, Ra, Rb	побитовое И.
IOR	Rdst, Ra, Rb	побитовое ИЛИ.
IXOR	Rdst, Ra, Rb	побитовое исключающее ИЛИ.
INOT	Rdst, Rsrc	побитовое отрицание.
INEG	Rdst, Rsrc	целое арифметическое отрицание.
ICMP	Ra, Rb	сравнивает как мысленное Ra-Rb, не сохраняя результат и формируя флаги.
ITEST	Ra, Rb	формирует флаги для побитового теста без сохранения результата.
ISHL	Rdst, Rsrc	логический сдвиг влево на один бит.
ISHR	Rdst, Rsrc	логический сдвиг вправо на один бит.
ISAL	Rdst, Rsrc	арифметический сдвиг влево на один бит.
ISAR	Rdst, Rsrc	арифметический сдвиг вправо на один бит.
IROL	Rdst, Rsrc	циклический сдвиг влево на один бит.
IROR	Rdst, Rsrc	циклический сдвиг вправо на один бит.
IRCL	Rdst, Rsrc	циклический сдвиг влево через CF на один бит.

Мнемоника	Форма	Назначение
IRCR	Rdst, Rsrc	циклический сдвиг вправо через CF на один бит.
ISHLN	Rdst, Rsrc, r64count	логический сдвиг влево на число битов из r64.
ISHRN	Rdst, Rsrc, r64count	логический сдвиг вправо на число битов из r64.
ISALN	Rdst, Rsrc, r64count	арифметический сдвиг влево на число битов из r64.
ISARN	Rdst, Rsrc, r64count	арифметический сдвиг вправо на число битов из r64.
IROLN	Rdst, Rsrc, r64count	циклический сдвиг влево на число битов из r64.
IRORN	Rdst, Rsrc, r64count	циклический сдвиг вправо на число битов из r64.
IRCLN	Rdst, Rsrc, r64count	циклический сдвиг влево через CF на число битов из r64.
IRCRN	Rdst, Rsrc, r64count	циклический сдвиг вправо через CF на число битов из r64.
IINC	Rdst, Rsrc	увеличивает целое значение на единицу.
IDEC	Rdst, Rsrc	уменьшает целое значение на единицу.
IXCHG	Ra, Rb	обменивает два целых объекта одинаковой ширины.
ICPY	Rdst, Rsrc	копирует беззнаковое целое с изменением ширины: расширение нулями, при сужении отбрасываются старшие биты.
ICVT	Rdst, Rsrc	выполняет знаковое преобразование ширины: расширение знаком, при сужении отбрасываются старшие биты.
IMUL	Rprod, Ra, Rb	полное беззнаковое произведение; регистр результата не уже удвоенной ширины источника.
ISMUL	Rprod, Ra, Rb	полное знаковое произведение.
IDIV	Rquot, Rrem, Rnum, Rden	беззнаковое деление с одновременным получением частного и остатка.
ISDIV	Rquot, Rrem, Rnum, Rden	знаковое деление с одновременным получением частного и остатка.
IBITSLEN	r64dst, Rsrc	возвращает число значащих битов целого объекта.

Вещественная арифметика LibMPU

Мнемоника	Форма	Назначение
RADD	Rdst, Ra, Rb	вещественное сложение.
RSUB	Rdst, Ra, Rb	вещественное вычитание.
RMUL	Rdst, Ra, Rb	вещественное умножение.
RDIV	Rdst, Ra, Rb	вещественное деление.
RSQRT	Rdst, Rsrc	квадратный корень.
RSIN	Rdst, Rsrc	синус.
RCOS	Rdst, Rsrc	косинус.
RTAN	Rdst, Rsrc	тангенс.
RASIN	Rdst, Rsrc	арксинус.
RACOS	Rdst, Rsrc	арккосинус.
RATAN	Rdst, Rsrc	арктангенс.
RLOG	Rdst, Rsrc	натуральный логарифм.
RLOG10	Rdst, Rsrc	десятичный логарифм.

Мнемоника	Форма	Назначение
RLOG2	Rdst, Rsrc	двоичный логарифм.
REXP	Rdst, Rsrc	экспонента e^x.
RSINH	Rdst, Rsrc	гиперболический синус.
RCOSH	Rdst, Rsrc	гиперболический косинус.
RTANH	Rdst, Rsrc	гиперболический тангенс.
RPOW	Rdst, Ra, Rb	возведение вещественного числа в вещественную степень.
RHYPOT	Rdst, Ra, Rb	вычисляет $\sqrt{a^2 + b^2}$ средствами LibMPU.
RATAN2	Rdst, Ry, Rx	двухаргументный арктангенс.
RNEG	Rdst, Rsrc	меняет знак вещественного числа.
RABS	Rdst, Rsrc	абсолютное значение.
RCMP	r64dst, Ra, Rb	сравнивает Real и возвращает целый код результата -2,-1,0,+1.
RCPY	Rdst, Rsrc	копирует Real-объект совместимой ширины.
RCVT	Rdst, Rsrc	преобразует Real между различными физическими ширинами.
RFLOOR	Rdst, Rsrc	округляет вниз.
RCEIL	Rdst, Rsrc	округляет вверх.
RROUND	Rdst, Rsrc	округляет к целому согласно семантике LibMPU.
RFREXP	Rsignif, Iexp, Rsrc	разлагает число на мантиссу и двоичный показатель.
RLDEXP	Rdst, Rsrc, Iexp	умножает Real на степень двойки, заданную целым показателем.
RLOGB	Idst, Rsrc	получает показатель порядка в целый регистр.
RMODF	Rfrac, Rint, Rsrc	разделяет число на дробную и целую вещественные части.
RREMAIN	Rdst, Ra, Rb	вычисляет вещественный остаток.
RLOG1P	Rdst, Rsrc	вычисляет log(1+x) с численно устойчивой семантикой.
REXPM1	Rdst, Rsrc	вычисляет exp(x)-1 с численно устойчивой семантикой.
RASINH	Rdst, Rsrc	обратный гиперболический синус.
RACOSH	Rdst, Rsrc	обратный гиперболический косинус.
RATANH	Rdst, Rsrc	обратный гиперболический тангенс.
ITOR	Rdst, Isrc	преобразует знаковое целое в Real.
UITOR	Rdst, Isrc	преобразует беззнаковое целое в Real.
RTOI	Idst, Rfrac, Rsrc	выделяет знаковую целую часть и оставляет вещественную дробную часть.
RTOUI	Idst, Rfrac, Rsrc	беззнаковый вариант преобразования Real в целую и дробную части.
RISNEG	r64dst, Rsrc	возвращает предикат отрицательности.
RSIGN	r64dst, Rsrc	возвращает код знака/класса, определённый LibMPU.
RISIND	r64dst, Rsrc	предикат неопределённого значения IND.
RISNAN	r64dst, Rsrc	предикат NaN.
RISNANMAX	r64dst, Rsrc	предикат специального NANMAX.
RISNANMIN	r64dst, Rsrc	предикат специального NANMIN.

Мнемоника	Форма	Назначение
RISINF	r64dst, Rsrc	предикат бесконечности.
RISSIGNULL	r64dst, Rsrc	предикат знакового нуля.
RCONST	Rdst, CONST8	строит предопределённую математическую или специальную константу непосредственно в точности регистра назначения.

Комплексная арифметика LibMPU

Мнемоника	Форма	Назначение
CADD	Cdst, Ca, Cb	комплексное сложение.
CSUB	Cdst, Ca, Cb	комплексное вычитание.
CMUL	Cdst, Ca, Cb	комплексное умножение.
CDIV	Cdst, Ca, Cb	комплексное деление.
CEXP	Cdst, Csrc	комплексная экспонента.
CSIN	Cdst, Csrc	комплексный синус.
CCOS	Cdst, Csrc	комплексный косинус.
CSINH	Cdst, Csrc	комплексный гиперболический синус.
CCOSH	Cdst, Csrc	комплексный гиперболический косинус.
CLOG	Cdst, Csrc	комплексный натуральный логарифм.
CSQRT	Cdst, Csrc	комплексный квадратный корень.
CPOW	Cdst, Ca, Cb	комплексное возведение в степень.
CREAL	Rdst, Csrc	извлекает вещественную компоненту.
CIMAG	Rdst, Csrc	извлекает мнимую компоненту.
CRTOC	Cdst, Rsrc	создаёт Complex из Real с нулевой мнимой частью.
CGEN	Cdst, Rreal, Rimag	создаёт Complex из декартовых компонент.
CPOLAR	Cdst, Rrho, Rtheta	создаёт Complex по модулю и аргументу.
CCONJ	Cdst, Csrc	комплексное сопряжение.
CABS	Rdst, Csrc	модуль комплексного числа.
CNORM	Rdst, Csrc	норма комплексного числа.
CARG	Rdst, Csrc	аргумент комплексного числа.
CISEQ	r64dst, Ca, Cb	предикат равенства комплексных чисел.
CISNE	r64dst, Ca, Cb	предикат неравенства.
CCPY	Cdst, Csrc	копирует Complex-объект.
CCVT	Cdst, Csrc	преобразует Complex между физическими ширинами.

Переходы и вызовы

Мнемоника	Форма	Назначение
JMP	CODE64	безусловный прямой переход.
JSET	PSWBIT8, CODE64	переходит, если выбранный бит PSW установлен.
JCLR	PSWBIT8, CODE64	переходит, если выбранный бит PSW сброшен.
JMPP	r64(PROGRAM PTR64)	косвенный переход по PROGRAM PTR64.

Мнемоника	Форма	Назначение
CALL	CODE64	прямой вызов; помещает адрес возврата в STACK.
RET	—	извлекает адрес возврата из STACK и продолжает выполнение вызывающей программы.
CALLP	r64(PROGRAM PTR64)	косвенный вызов по PROGRAM PTR64.

Стек и кадр функции

Мнемоника	Форма	Назначение
PUSH	R/C	помещает регистровый объект в STACK.
POP	R/C	извлекает объект из STACK.
LDSP	R/C, off64	загружает объект по адресу SP+off64.
STSP	off64, R/C	сохраняет объект по адресу SP+off64.
ADJSP	off64	изменяет SP на знаковое байтовое смещение.
PUSHFP	—	сохраняет текущий FP в STACK.
SETFP	—	устанавливает FP=SP.
RESTSP	—	восстанавливает SP=FP.
POPFP	—	восстанавливает FP из STACK.
LDFP	R/C, off64	загружает объект по адресу FP+off64.
STFP	off64, R/C	сохраняет объект по адресу FP+off64.
LEAFP	r64, off64	формирует PTR64(STACK, FP+off64).
SETSP	r64(STACK PTR64)	устанавливает SP из указателя на STACK после проверки пространства.

Условные псевдомнемоники assembler-а

Мнемоника	Форма	Назначение
JZ	JSET ZF, target	переход при ZF=1.
JNZ	JCLR ZF, target	переход при ZF=0.
JC	JSET CF, target	переход при CF=1.
JNC	JCLR CF, target	переход при CF=0.
JO	JSET OF, target	переход при OF=1.
JNO	JCLR OF, target	переход при OF=0.
JS	JSET SF, target	переход при SF=1.
JNS	JCLR SF, target	переход при SF=0.

PSW и селекторы условных переходов

Нижние шестнадцать бит PSW образуют изменяемое арифметическое/математическое состояние и считаются volatile через обычный CALL. TF, IF и поле MODE относятся к сохраняемому управляющему состоянию; HF — отдельное состояние исполнения. Универсальные JSET/JCLR адресуют только однобитовые селекторы.

Имя	Бит	Назначение
AF	0	вспомогательный/арифметический флаг; изменяемый статус.
CF	1	перенос/заём; используется беззнаковыми сравнениями и арифметикой с переносом.
OF	2	знаковое переполнение.
SF	3	знак результата.
PF	4	флаг чётности/статуса целой арифметики.
ZF	5	нулевой результат / равенство.
RF	6	состояние арифметики LibMPU.
VF	7	состояние арифметики LibMPU.
DOMF	8	ошибка области определения.
SNGF	9	сингулярность.
OVFF	10	вещественное переполнение.
UDFF	11	вещественное исчезновение порядка.
TLSF	12	полная потеря точности.
PLSF	13	частичная потеря точности.
INDF	14	неопределённый результат.
INXF	15	неточный результат.
HF	16	состояние остановки выполнения; отдельный домен состояния исполнения.
TF	17	управление trap/single-step; сохраняется обычной функцией.
IF	18	управление разрешением прерываний; сохраняется обычной функцией.

Поле MODE занимает биты 19–20 и не имеет отдельного PSWBIT8-селектора: это многобитовое поле, а не булев флаг.

Краткая памятка: какие операции могут изменить PSW

Класс	Практическое правило
целая арифметика, сравнения, сдвиги	формируют/изменяют целочисленное состояние; результат ICMP/ITEST надо потреблять до следующей изменяющей флаги операции
вещественные и комплексные операции	могут публиковать математические status-флаги LibMPU в PSW[15:0]
LDI, LDR, LDC и PSW-neutral преобразования	сохраняют входное состояние условий согласно своему контракту
CALL, CALLP, RET как инструкции	сами PSW не изменяют, но normal-function ABI считает PSW[15:0] изменяемыми вызываемой функцией через вызов

Директивы mcpu-as

Директива	Форма	Назначение
`.text`	`.text`	выбирает секцию PROGRAM обычного кода.
`.init`	`.init`	выбирает INIT-секцию lifecycle.
`.fini`	`.fini`	выбирает FINI-секцию lifecycle.
`.data`	`.data`	выбирает инициализированные данные.
`.bss`	`.bss`	выбирает нулевые данные NOBITS.
`.control`	`.control`	выбирает пространство CONTROL программной модели.
`.input`	`.input`	выбирает INPUT.
`.output`	`.output`	выбирает OUTPUT.
`.trace`	`.trace`	выбирает TRACE.
`.align`	`.align N`	выравнивает текущую позицию секции на N байтов.
`.zero`	`.zero N`	резервирует N нулевых байтов.
`.byte`	`.byte expr,...`	создаёт последовательность байтов.
`.integer / .int`	`.integer bits, expr,...`	создаёт один или несколько целых объектов указанной физической ширины.
`.real`	`.real bits, value,...`	создаёт Real-объекты заданной физической ширины.
`.complex`	`.complex bits, real, imag`	создаёт Complex-объект указанной полной физической ширины.
`.utf8`	`.utf8 "text"`	создаёт UTF-8 строку с завершающим NUL.
`.ucs2`	`.ucs2 "text"`	создаёт строку strict UCS-2 с завершающим нулевой кодовой единицей.
`.ptr`	`.ptr symbol`	создаёт 8-байтовый PTR64 объект в целевом представлении.
`.cptr`	`.cptr code-symbol`	создаёт 8-байтовый PTR64(PROGRAM,...) объект; исторический псевдоним .codeptr допустим только для совместимости.
`.equ`	`.equ name, expression`	определяет абсолютный символ ассемблера и не создаёт байтов.
`.global / .globl`	`.global name`	задаёт глобальную привязку GLOBAL для символа.
`.extern`	`.extern name`	объявляет внешний неопределённый символ.
`.weak`	`.weak name`	задаёт слабую привязку WEAK.

Специальные lifecycle-символы

Имя	Смысл
`_start`	обязательная низкоуровневая точка входа ET_EXEC; `e_entry` указывает на неё
`_life`	обязательная точка входа LIFE; при отсутствии `.init` совпадает по адресу с `_start`
`_fini`	необязательная точка входа FINI; при отсутствии соответствующее поле lifecycle равно нулю
`_start_life:`	синтаксическое сокращение ассемблера, эквивалентное последовательным `_start:` и `_life:` по одному адресу при отсутствии `.init`

Выражения ассемблера

Абсолютные целочисленные выражения ассемблера используются в адресах регистрового файла, непосредственных целых операндах, смещениях стека/указателей, размерах и директивах данных. Основная грамматика имеет обычный приоритет арифметики:

```
primary    := integer | symbol | '(' expression ')'
unary      := ['+' | '-'] unary | primary
product    := unary { ('*' | '/' | '%') unary }
expression := product { ('+' | '-') product }
```

Целые литералы допускают десятичную запись, шестнадцатеричный префикс `0x`, двоичный префикс `0b`; ведущий ноль в обычном целочисленном выражении может обозначать восьмеричную запись. Это правило относится к выражениям ассемблера и не распространяется на `digit-sequence` в маркерах строк препроцессора: там число всегда десятичное.

`.equ` задаёт абсолютный символ и не создаёт объект в секции. В адресных выражениях дополнительно допустимы метки тех секций, которые разрешены конкретным классом операнда; ссылки вперёд на метки обслуживаются механизмом исправлений (fixup) ассемблера.

Имена, строки и логические координаты исходного текста

Имена символов чувствительны к регистру. Первый символ имени может быть ASCII-буквой, `_`, `.` или `$`; последующие позиции дополнительно допускают цифры. Префикс `.L_` является соглашением об именовании внутренних меток, а не специальным механизмом локальности.

Escape-последовательности `.utf8`/`.ucs2` подробно разобраны в §3.6.2. Для диагностики после препроцессора текущий `mcpu-as` распознаёт маркеры строк, оставленные `mcpu-cpp`:

```
# digit-sequence
# digit-sequence "filename"
# digit-sequence "filename" flags...
```

`digit-sequence` всегда десятичная и задаёт логический номер следующей физической строки; допустимый диапазон — 0..2147483647. Имя файла необязательно. Поле flags после имени файла ассемблер игнорирует. Эти маркеры не создают байтов или символов и влияют только на координаты последующих сообщений об ошибках. Историческая запись `##line ...` в текущем инструментарии не является управляющей директивой координат `mcpu-as`; эту роль выполняют именно маркеры препроцессора.

Предопределённые константы RCONST

`RCONST` не хранит готовый binary64 шаблон: LibMPU строит значение непосредственно для физической Real-ширины регистра назначения.

Имя	CONST8	Смысл
ZERO	0x00	0
HALF	0x01	1/2
ONE	0x02	1
TWO	0x03	2
TEN	0x04	10
MTEN	0x05	-10
32	0x06	32
PI	0x10	π
E	0x11	e
1_LN2	0x12	1/ln(2)
LN2	0x13	ln(2)
1_LN10	0x14	1/ln(10)
LN10	0x15	ln(10)
1_LG2	0x16	1/lg(2)
LG2	0x17	lg(2)
PI_2	0x20	π/2
PI_3	0x21	π/3
PI_4	0x22	π/4
PI_5	0x23	π/5
PI_6	0x24	π/6
1_PI	0x25	1/π
2_PI	0x26	2/π
3_PI	0x27	3/π
4_PI	0x28	4/π
5_PI	0x29	5/π
2PI	0x2a	2π
3PI	0x2b	3π
1_2PI	0x2c	1/(2π)
1_3PI	0x2d	1/(3π)
1_4PI	0x2e	1/(4π)
1_5PI	0x2f	1/(5π)
1_6PI	0x30	1/(6π)
3PI_4	0x31	3π/4

Имя	CONST8	Смысл
SQRTPI	0x40	√π
1_SQRTPI	0x41	1/√π
2_SQRTPI	0x42	2/√π
SQRT2	0x43	√2
1_SQRT2	0x44	1/√2
SQRT3	0x45	√3
1_SQRT3	0x46	1/√3
DEGREE	0x47	π/180
1_DEGREE	0x48	180/π
GOLDENRATIO	0x49	золотое сечение
EULERGAMMA	0x4a	постоянная Эйлера-Маскерони
CATALAN	0x4b	постоянная Каталана
EPSILON	0x50	машинное epsilon выбранной Real-ширины
IND	0x51	неопределённое значение
PNAN	0x52	положительный NaN
NNAN	0x53	отрицательный NaN
PNANMAX	0x54	положительный NANMAX
NNANMAX	0x55	отрицательный NANMAX
PNANMIN	0x56	положительный NANMIN
NNANMIN	0x57	отрицательный NANMIN
PINF	0x58	+∞
NINF	0x59	-∞
PREALMIN	0x5a	положительное минимальное Real
NREALMIN	0x5b	отрицательное минимальное Real
PREALMAX	0x5c	положительное максимальное Real
NREALMAX	0x5d	отрицательное максимальное Real
PZERO	0x5e	+0
NZERO	0x5f	-0

Имена P*/N* для специальных значений обозначают знак, а не порядок байтов host-системы. EPSILON, предельные Real-значения, NaN-подобные формы и знакового нуля зависят от выбранной физической Real-ширины и генерируются средствами LibMPU.

Инструменты MCPU

Команда	Назначение
`mcpu-cpp`	препроцессор assembler-исходников .S: include, macro и условная компиляция.
`mcpu-as`	assembler MCPU: текст программы -> MELF ET_REL.
`mcpu-ld`	компоновщик: relocatable/static/dynamic linking и построение ET_EXEC/ET_DYN.
`mcpu-ar`	создание и обслуживание детерминированных UNIX-архивов .a из ET_REL.
`mcpu-readelf`	структурный просмотр MELF headers, sections, symbols, RELA и dynamic metadata.
`mcpu-nm`	просмотр таблиц символов объектов, исполняемых файлов, shared objects и архивов.
`mcpu-objdump`	просмотр секций/релокаций и дизассемблирование PROGRAM.
`mcpu-strip`	консервативное удаление non-runtime symbol metadata.
`mcpu-objcopy`	структурное копирование MELF и безопасное редактирование non-ALLOC metadata.
`mcpu-run`	загрузчик/supervisor и среда исполнения MCPU с наблюдением lifecycle и памяти.

Эта таблица описывает роли инструментов, а не все их параметры командной строки. Для точного набора параметров следует использовать `–help` соответствующей утилиты и её manual page.

Литература

1. Костельцев А. В. *Построение интерпретаторов и компиляторов: Использование программ BIZON, BYACC, ZUBR*. — СПб.: Наука и Техника, 2001.
2. Aho A. V., Lam M. S., Sethi R., Ullman J. D. *Compilers: Principles, Techniques, and Tools*. 2nd ed. — Addison-Wesley, 2006.
3. Документация LibMCPU: *Архитектура машинного кода и ABI*; *MELF ABI*. Нормативные описания сопровождают текущую реализацию LibMCPU.