

LibMCPU: архитектура машинного кода, памяти и ABI вызовов

Регистровая модель, ISA, PTR64, CALL ABI, lifecycle и runtime

Техническое справочное руководство

Нормативное техническое руководство LibMCPU. Документ описывает текущий архитектурный контракт и намеренно не привязан к номеру выпуска пакета.

Статус документа. Документ фиксирует текущий архитектурный контракт LibMCPU: register model, ISA, memory spaces, PTR64, CALL ABI, MELF lifecycle, syscall/error ABI, dynamic startup linking и подписанный HLL Runtime ABI между mcpu-ld, CRT и MCPU Libc. Документ намеренно не связан с номером выпуска пакета. Для бинарного формата MELF действует отдельное руководство `libmcpu-melf-abi-ru.pdf`.

Главный принцип. Физический регистр LibMCPU нетипизирован. REG24 задаёт только физическую ширину и байтовый адрес в едином register file. Integer/Real/Complex смысл определяется opcode; запись `c<bits>@addr` является semantic view того же физического регистра. Архитектурный PTR64 также является обычным 64-битным значением, но его декодирование и права доступа задаются ABI.

Оглавление

Оглавление	1
I Базовая архитектура, ISA и CALL ABI	4
1 Адресные пространства	5
2 Физическая адресация регистра: REG24	5
3 r и c: разные смыслы одного REG24	6
4 Канонический порядок байтов PROGRAM	6
5 Непосредственные числовые константы	6
5.1 LDI - Integer	7
5.2 LDR - Real	7
5.3 LDC - Complex	8
5.4 Почему это assembler error, а не runtime rounding	8
5.5 Conversion-class instructions сохраняют PSW	9
6 DATA, ADDR64 и NULL guard	9
6.1 Нулевой PTR64 и первые 16 байт DATA	9
6.2 .align выравнивает guest offset	9
6.3 Code pointers in DATA	9
7 Полноразмерные константы	10
8 Предопределённые математические константы	10
9 OPCODE16 и сигнатуры	10
10 Явное управление флагами PSW	11
11 Ограничения разрядности	11
12 PC, CODE64 и direct/indirect control flow	11
12.1 Зачем нужны JMPP и CALLP	12
12.2 JMPP - indirect jump	12
12.3 CALLP - indirect call	12
12.4 Atomic fault semantics CALLP	13
12.5 Assembler directive .cptr	13
12.6 Полный пример: indirect call + indirect jump	13
12.7 Jump-table pattern	14
13 STACK и SP	14
13.1 Низкоуровневые операции STACK	14
14 CALL/RET, блок аргументов (argument block) и PSW calling convention	15
14.1 Размещение блок аргументов (argument block)	15
14.2 Простые PUSH остаются допустимым сокращением	15
14.3 CALL, RET, FP и PSW	16
15 Return Value Area	16
16 PTR64: сегментированный виртуальный	16
17 Создание указателей: LEA, LEASP, LEAFP	17

17.1	LEA: полный PTR64 immediate	17
17.2	LEASP и LEAFP	17
18	Косвенная адресация: LOADP и STOREP	17
19	Pointer arithmetic	18
19.1	Набор инструкций	18
19.2	Границы и fault	18
19.3	PDIFF	19
20	FP как специальное архитектурное состояние	19
21	Кадр стека (stack frame): пролог и эпилог	19
21.1	Инструкции управления кадром (frame instructions)	20
22	Адресация относительно FP (FP-relative addressing)	20
22.1	Смещения блока аргументов после установки FP	21
23	Динамическое выделение памяти в STACK и SETSP	21
23.1	Выделение N байт во время выполнения	21
23.2	Понижение VLA (VLA-style lowering)	21
23.3	Вложенные динамические области	22
24	С-подобная модель автоматических объектов (automatic objects)	22
25	Формальный ABI аргументов	23
26	Структуры и массивы	23
26.1	Fixed local struct	23
26.2	Structure by value	23
26.3	Local array / VLA	23
26.4	Array argument	23
27	Aggregate return: hidden result pointer	23
28	Block copy: BMOV и BMOVI	24
28.1	Разрешённые memory spaces	24
28.2	Проверка ranges и atomicity	24
28.3	Lowering: assignment структуры	25
28.4	Lowering: aggregate argument by value	25
28.5	Lowering: aggregate return через hidden PTR64	25
28.6	Runtime-sized arrays и VLA	25
28.7	memcpu и memmove в C-like frontend	25
29	ABI data layout и assembler alignment	25
29.1	Natural alignment	25
29.2	Assembler не вставляет hidden alignment	26
29.3	Normal и packed aggregate layout	26

II	Memory backing, представление объектов и assembler	27
30	Host backing memory spaces и loader	28
30.1	Anonymous mmap only	28
30.2	Architectural size, capacity и HEAP limit	28
30.3	MELF loader и окончательный process image	28
31	Fault, atomicity и invariants	28
32	Byte order и architectural object representation	29
32.1	PROGRAM: canonical MSB-first	29
32.2	Data-object domain	29
32.3	Явные helper families в реализации	29
32.4	Integer, PTR64, return PC и overlapping registers	29
32.5	Real и Complex	30
32.6	Assembler data directives	30
32.7	Byte-preserving memory operations	30
32.8	Raw bytes, packed I/O и external byte order	31
32.9	Target metadata в MELF	31
32.10	Host alignment и unaligned accesses	31
33	Pointer comparison, red zone, active/reserved	31
33.1	Equality и ordering для PTR64	31
33.2	Stack red zone отсутствует	31
33.3	Active и reserved PTR64 selectors	32
33.4	BSWAP: raw byte reversal	32
33.5	In-place и snapshot semantics	32
33.6	External binary formats	33

34	Актуальное покрытие LibMMPU и семейства ISA	33
34.1	System и data-movement family	33
34.2	Integer family	33
34.3	Real family	33
34.4	Complex family	34
34.5	Control-flow и stack family	34
34.6	Явное управление arithmetic/status flags	34
34.7	PSW-neutral conversion class	34
34.8	Неинструкционные функции	34
34.9	Regression contract	34
35	Ассемблер: UTF-8 source, строки, .equ и arithmetic expressions	34
35.1	Кодировка assembler source	35
35.2	Командная строка mcpu-as	35
35.3	.utf8 - NUL-terminated UTF-8 byte string	35
35.4	.ucs2 - UTF-8 source literal в strict UCS-2 target objects	35
35.5	.byte, .integer и .int	35
35.6	.equ - абсолютные assembler symbols	36
35.7	Integer expression grammar и precedence	36
35.8	Label expressions и fixups	36
35.9	Symbolic register addresses	36
35.10	Граница mcpu-as и будущего препроцессора	36
35.11	Полный пример	37
35.12	Маркеры строк препроцессора и логические координаты diagnostics	37

III Program Model ABI, runtime и динамическая загрузка 39

36	Program Model Lifecycle ABI	40
36.1	С отдельным INIT	40
36.2	Без INIT	40
36.3	FINI, END и EXIT	40
37	HLL Runtime ABI: PROGRAM, MODEL, CRT и MCPU Libc	40
37.1	Целевая integer-модель HLL ABI	40
37.2	Две формы законченной HLL-программы	41
37.3	Автоматическая классификация в mcpu-ld	41
37.4	Распределение ответственности CRT	41
37.5	McpuStartup, argc и argv	41
37.6	Внутренние hooks MCPU Libc	42
37.7	PRE_RUNTIME и RUNNING	42
37.8	PROGRAM lifecycle	42
37.9	MODEL lifecycle	42
37.10	HLL finalization и сохранение status	43
38	McpuStartup и argc/argv	43
39	HEAP и program break	43
40	SYSCALL ABI и SYS_brk	43
41	Startup dynamic linking: архитектурный взгляд	44
41.1	GLOBAL/WEAK interposition	44
41.2	LOCAL ET_DYN runtime rebasing	44
42	mcpu-run как host-side test bench	44
42.1	Typed actions	45
42.2	Raw actions	45
42.3	Двойной дефис как разделитель	45
43	Бинарные утилиты toolchain	45
44	Инварианты реализации	45
1	Полный справочник директив mcpu-as	46
2	Полная таблица OPCODE16	47
3	PSW selectors и RCONST	50
3.1	PSW bit selectors	50
3.2	RCONST selectors	51
4	Краткая памятка ABI	52

Часть I

Базовая архитектура, ISA и CALL ABI

1. Адресные пространства

LibMPCPU имеет независимые архитектурные адресные пространства (address spaces). Register file остаётся TLS-состоянием CPU; PROGRAM содержит исполняемый код; остальные сегменты являются пространствами объектов данных (data-object spaces) и используют целевое представление объектов LibMPCPU.

Пространство	SEG	Назначение	Доступ guest
DATA	0x0	Глобальные/статические объекты и полноразмерные константы	R/W
STACK	0x1	Вызовы, аргументы, automatic objects, spills	R/W
PROGRAM	0x2	Машинный код; секции .init/.text/.fini	R + execute; запись запрещена
CONTROL	0x3	Параметры управления запуском и модельным временем	R; runtime может изменять
INPUT	0x4	Внешние возмущающие воздействия	R; runtime задаёт значения
OUTPUT	0x5	Воздействие модели на среду / текущий выходной вектор	R/W
TRACE	0x6	Текущий telemetry frame модели	R/W
HEAP	0x7	Обнулённая runtime-область динамической памяти	R/W
reserved	0x8..0xF	Будущие адресные пространства	dereference fault
Register file	—	512 KiB TLS; перекрывающиеся представления регистров	REG24

PROGRAM, DATA, STACK, CONTROL, INPUT, OUTPUT, TRACE и HEAP логически независимы и байтово адресуемы. Host backing каждого активного адресного пространства создаётся отдельным anonymous mmap(); файлы executable/image никогда не являются file-backed memory виртуального CPU.

DATA и HEAP разделены. DATA является статическим пространством данных, формируемым образом executable (image-backed data space). HEAP является отдельным runtime-пространством с SEG=0x7, не имеет assembler payload и не расширяет DATA. Будущие malloc/free/realloc/calloc будут library-level ABI поверх HEAP, а не ISA instructions.

Структуры CONTROL/INPUT/OUTPUT/TRACE не задаются CPU. ISA видит только bytes и права доступа. Форматы записей модели (model records), имена полей, типы и семантику модельного времени определит будущий C-подобный язык моделирования и runtime metadata.

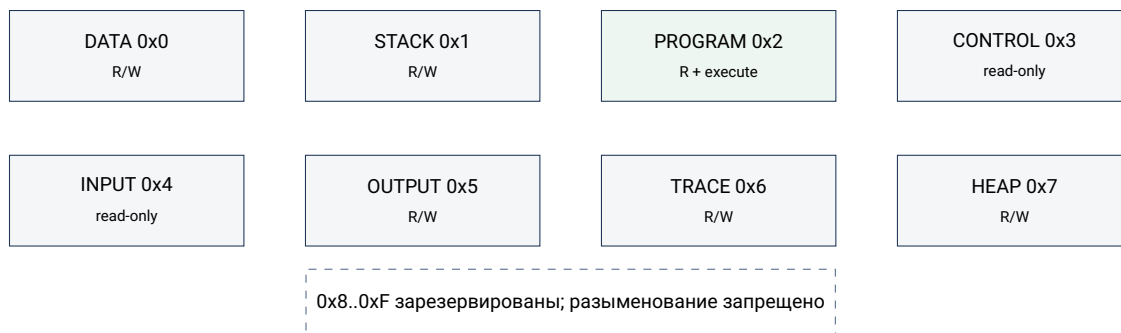


Рис. 1: Активная карта сегментов PTR64 и права доступа guest-программы.

2. Физическая адресация регистра: REG24

Машинный операнд физического регистра имеет ровно 24 бита. Четыре старших бита кодируют физическую ширину представления регистра, оставшиеся двадцать — байтовый адрес начала представления внутри единого регистрового файла.

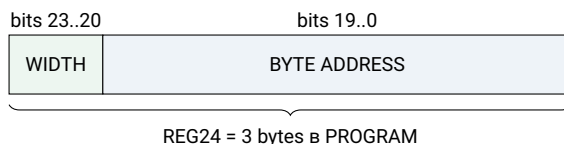


Рис. 2: Логическое устройство REG24.

Три байта PROGRAM формируются следующим образом: первый байт содержит WIDTH[3:0] и ADDRESS[19:16], второй — ADDRESS[15:8], третий — ADDRESS[7:0]. Это машинное поле имеет канонический порядок байтов PROGRAM и не является host C bit-field.

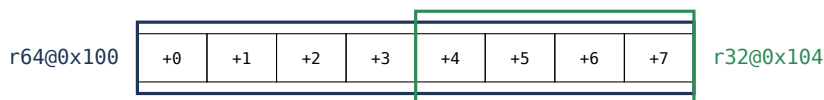
WIDTH	Физическая ширина	Размер
0x0	8 bit	1 byte
0x1	16 bit	2 bytes
0x2	32 bit	4 bytes
0x3	64 bit	8 bytes
0x4	128 bit	16 bytes
0x5	256 bit	32 bytes
0x6	512 bit	64 bytes
0x7	1024 bit	128 bytes
0x8	2048 bit	256 bytes
0x9	4096 bit	512 bytes
0xA	8192 bit	1024 bytes
0xB	16384 bit	2048 bytes
0xC	32768 bit	4096 bytes
0xD	65536 bit	8192 bytes

Register file имеет 512 KiB, поэтому ADDRESS[19] обязан быть нулём. Views имеют natural alignment. Формула архитектуры остаётся REGISTER = WIDTH + BYTE ADDRESS.

```

r8@0x00000    -> 00 00 00
r16@0x00000   -> 10 00 00
r4096@0x02000 -> 90 20 00
r65536@0x7e000 -> D7 E0 00

```



одни и те же физические байты register file

Рис. 3: Представления (views) разной ширины могут перекрываться.

3. r и c: разные смыслы одного REG24

```

r64@0x02000    ; 64 физических бита, integer/real смысл задаёт opcode
c64@0x02000    ; те же 64 физических бита, complex semantic view

```

Обе записи имеют одинаковый REG24. Для c<bits> число bits означает полную физическую ширину; каждая real-компонента имеет половинную ширину.

Complex view	Физический размер	Компоненты
c64	64 bit	2 x real32
c128	128 bit	2 x real64
c4096	4096 bit	2 x real2048
c32768	32768 bit	2 x real16384
c65536	65536 bit	2 x real32768

4. Канонический порядок байтов PROGRAM

OPCODE16, REG24, IMM64, ADDR64, CODE64 и OFF64 кодируются в PROGRAM MSB-first, независимо от host endian и target data profile. Архитектурные data objects (register file, DATA, STACK и остальные data spaces) образуют отдельный byte-order domain и используют representation, совместимое с настроенной LibMPPU. Полная нормативная модель приведена в разделе 31.

PROGRAM serialization и architectural data-object representation - разные уровни. Decoder собирает многобайтовые поля явно как MSB-first; произвольный host-cast packed C-структурой не является частью формата.

5. Непосредственные числовые константы

Inline numeric payload всегда равен 64 битам; opcode задаёт интерпретацию. Непосредственная форма предназначена только для значений, которые точно представимы в машинном immediate-формате соответствующей инструкции. Ассемблер не имеет права молча терять значащие биты/цифры при преобразовании textual literal в immediate.

Exact-immediate rule. Если textual literal не представим точно в immediate source format,

mcru-as завершает сборку с ошибкой и рекомендует создать полноразмерный объект

через .integer/.real/.complex в DATA и загрузить его инструкцией LOAD.

5.1. LDI - Integer

```
LDI r<bits>@addr, integer
```

OPCODE16 + REG24 + IMM64 = 13 bytes

IMM64 - signed 64-bit two's-complement integer. Для destination шириной 64 бит и более literal обязан точно помещаться в signed integer64. Для destination уже 64 бит literal дополнительно обязан помещаться в signed диапазон именно этой ширины: assembler не генерирует заведомо теряющее значение icvt().

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

; ошибка: +255 не является signed integer8 value
LDI r8@0x00000, 255

; ошибка: literal шире signed integer64 source format
LDI r128@0x00100, 9223372036854775808
```

Для большого целого используется DATA:

```
.data
.align 16
```

wide_integer:

```
.integer 128, 9223372036854775808
.text
LOAD r128@0x00100, wide_integer
```

Типичная диагностика:

```
mcru-as: error:
line N: LDI immediate is not exactly representable as integer64;
```

use .integer <bits>, value in DATA and LOAD

5.2. LDR - Real

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

OPCODE16 + REG24 + REAL64 = 13 bytes

Machine payload LDR всегда binary64. Но assembler применяет более строгую проверку в зависимости от destination: для r32 textual literal обязан быть точно представим как real32; для r64 и всех более широких Real-register views - точно представим как real64. После этой проверки runtime по-прежнему выполняет r_cvt() из embedded real64 к физической ширине destination.

```
; корректно: значения точно представимы
LDR r32@0x02000, -17.0
LDR r4096@0x04000, 2.0

; ошибка: 0.1 не имеет точного binary32 representation
LDR r32@0x02000, 0.1

; ошибка: high-precision decimal сначала потерял бы точность в binary64
LDR r4096@0x04000, 3.14243987129849274029174912694696912849214e-286
```

Диагностика для r32:

```
mcru-as: error:
line N: LDR immediate is not exactly representable as real32;
```

use .real <bits>, value in DATA and LOAD

Диагностика для r64 и шире:

```
mcru-as: error:
line N: LDR immediate is not exactly representable as real64;
```

use .real <bits>, value in DATA and LOAD

Правильная форма для произвольной высокоточной константы:

```
.data
.align 16
```

pi_like:

```
.text
    .real 4096, 3.14243987129849274029174912694696912849214e-286
    LOAD r4096@0x04000, pi_like
```

.real передаёт textual literal непосредственно ascii_to_real() LibMCPUs с precision конечного DATA-object; промежуточного binary64 здесь нет.

5.3. LDC - Complex

В машинном коде inline-комплекс состоит из двух компонент binary32:

REAL	IMAG
float32, 32 bits	float32, 32 bits

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

Формат инструкции: OPCODE16 + REG24 + COMPLEX64 = 13 bytes.

Минимальный complex destination - c64, то есть две real32 components. Inline LDC всегда содержит 2 x binary32, поэтому каждая textual component обязана быть точно представима как real32 независимо от ширины destination. После декодирования CPU расширяет обе real32 components через LibMCPUs к половине физической ширины c-view и вызывает c_gen_complex().

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

; ошибка: real component 0.1 не представима точно как real32
LDC c128@0x02000, 0.1r0+32.0j0

; ошибка ещё на разборе регистра: c32 запрещён
LDC c32@0x02000, 1.0r0+2.0j0
```

Диагностика потери точности:

```
mcpu-as: error:
line N: LDC immediate component is not exactly representable as real32;
```

use .complex <bits>, real, imag in DATA and LOAD

Для широкого complex object:

```
.data
.align 16
```

wide_z:

```
.text
    .complex 4096, 3.14243987129849274029174912694696912849214e-286, -0.125
    LOAD c4096@0x04000, wide_z
```

.complex формирует components непосредственно в требуемой LibMCPUs precision; LDC для таких значений использовать нельзя.

5.4. Почему это assembler error, а не runtime rounding

Immediate является частью PROGRAM image и уже после сборки не содержит исходный текст literal. Если assembler сначала округлит длинную десятичную запись до binary64/binary32, executor уже не способен узнать, что информация была потеряна. Поэтому exactness проверяется до emission bytecode: textual literal разбирается LibMCPUs с высокой precision, конвертируется в immediate precision и обратно, после чего значения сравниваются. Только exact round-trip допускается к emission.

Это правило относится к immediate-формам LDI/LDR/LDC. Полноразмерные DATA directives намеренно выполняют conversion непосредственно к объявленной ширине объекта; обычное математическое округление decimal->binary в этой целевой precision является частью создания самого объекта.

5.5. Conversion-class instructions сохраняют PSW

Некоторые LibMPCPU conversion helpers изменяют внутренние arithmetic/status flags как побочный эффект своей реализации. Например, `icvt()` влияет на часть integer flags. Для LibMPCPU это нежелательно: преобразование значения не должно неожиданно уничтожать condition state, вычисленное предыдущей инструкцией.

Нормативное правило. Инструкции LDI, LDR, LDC, ICVT, RCVT, ITOR, UITOR, RTOI, RTOUI, CCVT и CRTOC являются PSW-neutral. Перед вызовом LibMPCPU conversion routine executor сохраняет PSW[15:0], после завершения conversion восстанавливает прежнее значение. HF/TF/IF/MODE при этом вообще не изменяются.

```
saved = PSW[15:0]
icvt(...)      ; или r_cvt(), c_cvt(), ltor(), ...
```

PSW[15:0] = saved

Внутренняя реализация LibMPCPU намеренно не включает private header LibMPCPU и не зависит от private макро `__MPU_FLAGS`. Сохранение/восстановление делается через уже существующий public PSW facade LibMPCPU, который собирает и восстанавливает LibMPCPU flags через публичные accessor functions. Архитектурно результат эквивалентен сохранению `__MPU_FLAGS` вокруг conversion.

Это правило относится только к операциям, смысл которых - загрузить или преобразовать значение. Обычные arithmetic instructions продолжают публиковать нормальные LibMPCPU flag effects и поэтому не выполняют такого save/restore.

6. DATA, ADDR64 и NULL guard

ADDR64 остаётся 64-битным byte offset внутри DATA для прямых инструкций LOAD/STORE. Это не PTR64.

```
LOAD REG24, ADDR64      ; DATA -> register file
STORE ADDR64, REG24     ; register file -> DATA
```

Обе формы занимают 13 байт. Размер копии задаётся физической шириной REG24.

6.1. Нулевой PTR64 и первые 16 байт DATA

ABI определяет нулевой указатель как ровно `0x0000000000000000`. Битово это совпадает с комбинацией SEG=DATA, OFFSET=0. Поэтому executor отдельно распознаёт all-zero PTR64; при попытке разыменования LOADP, STOREP, BMOV и BMOVI возвращают CPU_EXEC_POINTER_FAULT.

PTR64 NULL = `0x0000000000000000`

```
DATA 0x0000 .. 0x000f    reserved NULL guard
DATA 0x0010 ..          ordinary assembler objects
```

Assembler всегда начинает DATA image с 16 нулевых guard bytes. Поэтому обычная DATA label не может иметь PTR64 value zero и первый допустимый object offset сразу кратен 16.

Прямой ADDR64 и NULL. NULL относится к PTR64. Низкоуровневые прямые LOAD/STORE технически адресуют DATA по raw ADDR64, но ABI считает DATA[0..15] зарезервированной областью и conforming assembler/program не размещает там объекты.

6.2. .align выравнивает guest offset

```
.data
.align 16
```

x:

```
.real 4096, 0.0
```

`.align N` двигает только текущий guest DATA offset до следующего кратного N значения. Директивы `.integer/.real/.complex` сами по себе не вставляют никакого padding и пишут

объект точно с текущего offset. Это принципиально: assembler способен описывать как обычные ABI-layouts, так и packed binary layouts.

6.3. Code pointers in DATA

DATA может содержать PROGRAM pointers для indirect control flow. Директива `.cptr label` занимает 8 байт, резервирует PTR64 object и после полной сборки `.text` разрешает CODE label в PROGRAM byte offset. Она не добавляет implicit alignment.

```
.data
.align 8
```

handlers:

```
.cptr handler0
.cptr handler1
```

Получившийся object можно загрузить обычным LOADP r64,... и передать в JMPP/CALLP. Forward CODE labels поддерживаются assembler fixup mechanism.

7. Полноразмерные константы

```
.data
.align 16
```

huge_integer:

```
.integer 65536, 12345678901234567890...
```

high_precision_real:

```
.real 16384, 1.234567890123456789...
```

high_precision_complex:

```
.complex 32768, 1.234567890123456789..., -2.5
```

```
.text
LOAD r65536@0x00000, huge_integer
LOAD r16384@0x02000, high_precision_real
LOAD c32768@0x08000, high_precision_complex
```

Тип, размер, internal padding/alignment C-like объектов известны assembler/compiler. CPU копирует байты и не хранит runtime type tag.

8. Предопределённые математические константы

```
RCONST r4096@0x02000, PI
RCONST r16384@0x04000, E
RCONST r128@0x08000, SQRT2
```

OPCODE16 + REG24 + CONST8 = 6 bytes

LibMCPU строит значение непосредственно с требуемой точностью. Основные группы констант перечислены в приложении: ZERO/HALF/ONE/TWO/TEN, PI/E/LN2/LN10, кратные PI, SQRT2/SQRT3/SQRTPI, GOLDENRATIO, EULERGAMMA и CATALAN.

9. OPCODE16 и сигнатуры

OPCODE16 полностью определяет количество и тип следующих операндов. Отдельного format-byte нет.

Диапазон	Назначение	Примеры
0x00xx	CPU/system и PSW status	NOP, HALT, SETF/CLRF, CLRCPUF/CLRMPIUF
0x01xx	data movement / построение адресов	MOV, LDI, LDR, LDC, LOAD, STORE, LEA, LEASP, LOADP, STOREP
0x10xx	integer	IADD, ISUB, ICMP, ...
0x20xx	real	RADD, RSIN, RLOG, ...
0x30xx	complex	CADD, CEXP, CPOW, ...
0x40xx	control flow	JMP/JMPP, JSET/JCLR, CALL/CALLP, RET
0x50xx	stack/frame	PUSH, POP, LDSP, STSP, ADJSP, PUSHFP, SETFP, ...

Код операции	Мнемоника	Формат операндов
0x0010	SETF	PSWBIT8 (AF..INXF)
0x0011	CLRF	PSWBIT8 (AF..INXF)
0x0012	CLRCPUF	—
0x0013	CLRMPIUF	—
0x0120	LOAD	R/C, ADDR64
0x0121	STORE	ADDR64, R/C
0x0130	LEA	r64, ADDR64

Код операции	Мнемоника	Формат операндов
0x0131	LEASP	r64, OFF64
0x0132	LOADP	R/C, r64(PTR64)
0x0133	STOREP	r64(PTR64), R/C
0x0140	PADD	r64(dst), r64(ptr), r64(delta)
0x0141	PADDI	r64(dst), r64(ptr), OFF64
0x0142	PSUB	r64(dst), r64(ptr), r64(delta)
0x0143	PSUBI	r64(dst), r64(ptr), OFF64
0x0144	PDIF	r64(dst), r64(ptr1), r64(ptr2)
0x0150	BMOV	r64(dst PTR64), r64(src PTR64), r64(byte_count)
0x0151	BMOVI	r64(dst PTR64), r64(src PTR64), IMM64 byte_count
0x4000	JMP	CODE64
0x4001/02	JSET/JCLR	PSWBIT8, CODE64
0x4003	JMPP	r64(PROGRAM PTR64)
0x4010	CALL	CODE64
0x4011	RET	—
0x4012	CALLP	r64(PROGRAM PTR64)
0x5000/01	PUSH/POP	R/C
0x5010/11	LDSP/STSP	R/C, OFF64 / OFF64, R/C
0x5020	ADJSP	OFF64

10. Явное управление флагами PSW

Arithmetic/status domain PSW намеренно разделён на два byte-range. PSW[7:0] — CPU flags, PSW[15:8] — MPU mathematical flags. Execution/control state начинается выше этого диапазона и не является writable через обычные flag instructions.

SETF	CF	; PSW[CF] = 1
CLRF	AF	; PSW[AF] = 0
CLRCPUF		; PSW[7:0] = 0
CLRMPUF		; PSW[15:8] = 0

SETF и CLRF принимают те же symbolic flag names AF, CF, OF, SF, PF, ZF, RF, VF, DMF, SNGF, OVFF, UDF, TLSF, PLSF, INDF, INXF, которые используются branch selector namespace, но write contract намеренно заканчивается на bit 15. Попытка SETFHF, CLRFTH или SETFIF является assembler error; decoder/executor независимо отвергает такой bytecode как invalid operand. MODE многобитовое поле и отдельного selector не имеет.

Каждая single-flag instruction изменяет ровно выбранный bit и сохраняет остальные биты PSW. CLRCPUF использует маску 0x000000ff, CLRMPUF — 0x0000ff00; обе операции сохраняют HF/TF/IF/MODE и reserved bits.

Практически это устраняет необходимость искусственно выполнять арифметическую операцию только ради получения заданного initial carry:

CLRF	CF
IADC	r64@0x00000, r64@0x00010, r64@0x00018
IADC	r64@0x00008, r64@0x00020, r64@0x00028
SETF	CF
ISBB	r64@0x00030, r64@0x00040, r64@0x00048

IADC, ISBB и rotate-through-carry instructions продолжают использовать установленную LibMPU flag semantics; SETF/CLRF не переопределяют их арифметику, а только задают входное flag state.

11. Ограничения разрядности

Класс	Простые операции	Трансцендентные
Integer	до 65536 bit	—
Real	до 65536 bit	до 16384 bit
Complex	до 65536 = 2 x real32768	до 32768 = 2 x real16384

12. PC, CODE64 и direct/indirect control flow

PC - 64-битный byte offset в PROGRAM. CODE64 - такой же raw PROGRAM offset внутри machine code, encoded canonical MSB-first. Прямые переходы получают target из bytecode, а indirect forms получают target через обычный 64-bit register, содержащий PROGRAM PTR64.

```

обычная инструкция:  PC = old_PC + instruction_length
JMP CODE64:          PC = target
JSET/JCLR taken:      PC = target
JSET/JCLR not taken:  PC = next_pc
JMPP r64(PTR64):      PC = OFFSET(ptr), если SEG=PROGRAM
CALL CODE64:          push next_pc; PC = target
CALLP r64(PTR64):     push next_pc; PC = OFFSET(ptr), если SEG=PROGRAM

```

JZ/JNZ/JC/JNC/JO/JNO/JS/JNS являются assembler aliases для JSET/JCLR над конкретными PSW bit.

12.1. Зачем нужны JMPP и CALLP

JMP и CALL кодируют конкретный CODE64 target прямо внутри инструкции. Это удобно для statically known labels. JMPP и CALLP нужны тогда, когда target выбирается во время исполнения: function pointer, callback, таблица обработчиков, virtual dispatch, jump table или runtime-selected routine.

Code/function pointer ABI. Указатель на executable code является обычным PTR64 с SEG = PROGRAM (0x2). Offset - byte address внутри текущего PROGRAM. Отдельного архитектурного типа function-pointer не существует: это 64-битное значение в обычном r64.

12.2. JMPP - indirect jump

Код операции	Source syntax	Машинный формат / длина
0x4003	JMPP r64@addr	OPCODE16 + REG24 = 5 bytes

Source register содержит полный PTR64. Executor сначала читает 8 байт register view и проверяет:

```
ptr != NULL SEG(ptr) == PROGRAM OFFSET(ptr) < program.size
```

При успехе выполняется только:

```
PC = OFFSET(ptr)
```

SP, FP, STACK, PSW и сам source register не изменяются. Ошибка segment selector, NULL или offset за пределами текущего PROGRAM даёт CPU_EXEC_POINTER_FAULT, причём PC остаётся прежним.

```

; r64@0x00100 = PTR64(PROGRAM, target_offset)
JMPP  r64@0x00100

```

Для r64@0x00100 REG24 равен 30 01 00, поэтому machine bytes инструкции:

```
40 03 30 01 00
```

Instruction boundary metadata нет. CPU проверяет только PROGRAM segment и byte range. Если runtime-created pointer указывает в середину другой инструкции, PTR64 может быть range-valid; следующая decode operation определит, образует ли последовательность bytes допустимую инструкцию. Assembler-generated code labels и

```
.cptr label естественно указывают на начало помеченной assembler instruction.
```

12.3. CALLP - indirect call

Код операции	Source syntax	Машинный формат / длина
0x4012	CALLP r64@addr	OPCODE16 + REG24 = 5 bytes

CALLP использует тот же CALL ABI, что и direct CALL CODE64. Различается только источник target. После проверки PROGRAM PTR64 и наличия места в STACK вычисляется:

```

next_pc = old_PC + 5
new_sp = old_SP - 8

```

и только после всех проверок происходит архитектурное изменение:

```

STACK[new_sp .. new_sp+7] = next_pc ; raw 64-bit CODE64 offset
SP = new_sp
PC = OFFSET(ptr)

```

Возвратный адрес не является PTR64. CALLP, как и CALL, помещает в STACK raw 64-bit PROGRAM byte offset. Поэтому существующий RET не различает direct и indirect call: он снимает тот же CODE64 и устанавливает PC.

```

; r64@0x00100 = PTR64(PROGRAM, function_offset)
CALLP  r64@0x00100
; execution resumes here after RET

```

Для того же r64@0x00100 machine bytes:

40 12 30 01 00

CALLP не меняет PSW и FP. Register file считается volatile across CALL по уже принятому ABI; это соглашение caller/callee, а не side effect самой инструкции.

12.4. Atomic fault semantics CALLP

CALLP сначала проверяет pointer, текущий PROGRAM range и возможность разместить 8-byte адрес возврата (return PC) в STACK. До успешного завершения всех проверок не изменяются PC, SP и STACK. Поэтому не существует состояния "return address уже записан, но переход не произошёл".

Условие	Результат	Архитектурное состояние
Operand не является физическим 64-bit register	WIDTH/operand fault	без изменений
NULL или любой non-PROGRAM PTR64	POINTER_FAULT	без изменений
PROGRAM offset $\geq$ program.size	POINTER_FAULT	без изменений
Недостаточно STACK для адрес возврата (return PC)	STACK_FAULT	без изменений
Успех	CPU_EXEC_BRANCH	адрес возврата (return PC) помещён в STACK; SP/PC изменены

12.5. Assembler directive .cptr

Чтобы таблицы function/code pointers можно было собирать непосредственно в DATA, assembler поддерживает:

```
.cptr CODE_LABEL
```

Директива занимает ровно 8 байт и после разрешения label создаёт:

PTR64(SEG=PROGRAM, OFFSET=byte_address_of(CODE_LABEL))

Forward references разрешены. .cptr не вставляет automatic alignment; при необходимости таблица явно начинается с .align 8 или более сильного alignment.

Каноническое spelling - .cptr. Историческое .codeptr assembler продолжает принимать как compatibility alias, но новые исходники должны использовать .cptr. Имя .label намеренно не используется: label определяется синтаксисом name:, а эта directive создаёт 8-byte code-pointer object.

Не путать с bytecode field encoding. OPCODE16/CODE64/IMM64 внутри PROGRAM имеют canonical MSB-first encoding. .cptr находится в DATA и записывает architectural object representation PTR64, то есть те же 8 байт, которые затем LOADP r64,... помещает в 64-bit register view.

12.6. Полный пример: indirect call + indirect jump

```
.data
.align 8
```

dispatch:

```
.cptr indirect_function
.cptr done

.text
LEA    r64@0x00000, dispatch
LOADP r64@0x00008, r64@0x00000
CALLP r64@0x00008

PADDI r64@0x00000, r64@0x00000, 8
LOADP r64@0x00010, r64@0x00000
JMPP  r64@0x00010

; unreachable if JMPP works
LDI   r64@0x00020, 99
HALT
```

indirect_function:

```
LDI    r64@0x00020, 42
RET
```

done:

```
HALT
```

Первый table element вызывает функцию через CALLP. RET возвращает на инструкцию после 5-byte CALLP. Затем pointer arithmetic выбирает второй element, LOADP загружает следующий PROGRAM PTR64, а JMPP переносит PC к done. Никакой специальной SWITCH/JUMP-TABLE инструкции не требуется.

12.7. Jump-table pattern

```
; index/range validation and index*8 scaling are done by surrounding code
; r64@byte_off already contains the byte offset of the selected 8-byte entry
LEA    r64@table_ptr, table
PADD   r64@entry_ptr, r64@table_ptr, r64@byte_off
LOADP  r64@target, r64@entry_ptr
JMPP   r64@target
```

Это normative low-level pattern для будущего assembler/compiler. Проверка index range и перевод element index в byte offset выполняются до вычисления/использования table entry. CPU остаётся независимым от high-level конструкции switch.

13. STACK и SP

STACK (стек) — отдельное архитектурное адресное пространство MPCPU. Оно растёт **к меньшим адресам**: при выделении памяти значение SP уменьшается, при освобождении — увеличивается. После создания пустого стека

$$SP=FP=stack_size.$$

SP хранит не PTR64, а обычный 64-битный **byte offset внутри STACK**. Поэтому указатель на объект стека получается явно через LEASP/LEAFP; сам SP не содержит selector сегмента.

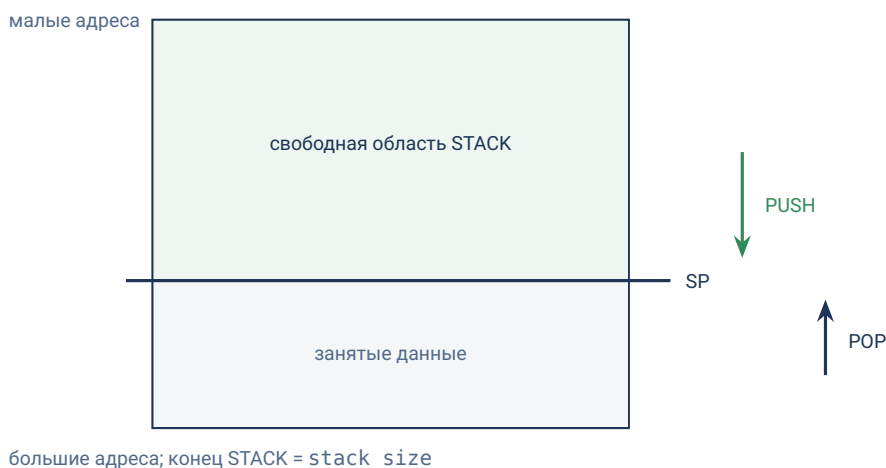


Рис. 4: Стек растёт к меньшим адресам. PUSH уменьшает SP, POP увеличивает SP.

Такое направление роста удобно для ABI вызовов: caller формирует блок аргументов (argument block), затем CALL кладёт адрес возврата (return PC) непосредственно **перед** ним, а callee может обращаться к аргументам по положительным смещениям от входного SP или от стабильного FP.

13.1. Низкоуровневые операции STACK

Инструкция	Изменение SP	Смысл
PUSHR	$SP -= \text{sizeof}(R)$	Записать представление регистра в $STACK[SP]$.
POPR	$SP += \text{sizeof}(R)$ после чтения	Прочитать значение из текущей вершины стека.
LDSPR, off	не изменяется	Прочитать объект по адресу $SP + \text{off}$.
STSPoff, R	не изменяется	Записать объект по адресу $SP + \text{off}$.
ADJSPoff	$SP += \text{signed}(\text{off})$	Явно зарезервировать или освободить область стека.

CPU не добавляет скрытого padding или alignment. ABI требует 16-byte alignment перед ABI-conforming CALL, но конкретные ADJSP, padding slots и offsets формирует assembler/compiler. Низкоуровневые операции остаются byte-addressable и допускают unaligned addresses, если конкретное ABI-правило не требует большего выравнивания.

SP — не обычная переменная. Каждое изменение SP меняет границу живой stack storage. Выделение и освобождение памяти должны быть симметричны, а любой PTR64 на освобождённую stack-область становится семантически недействительным.

14. CALL/RET, блок аргументов (argument block) и PSW calling convention

Нормативный ABI отделяет **блок аргументов (argument block)**, принадлежащий caller-у, от 8-байтового адреса возврата (return PC), который автоматически добавляет CALL. Непосредственно перед ABI-conforming CALL значение SP кратно 16. Caller заранее резервирует целый блок аргументов (argument block), размещает в нём hidden-result pointer (если он нужен) и explicit arguments, затем выполняет CALL.

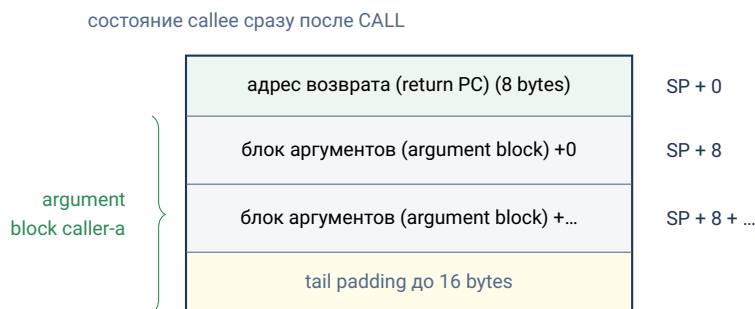
14.1. Размещение блок аргументов (argument block)

Argument block строится от меньших offsets к большим в порядке параметров функции. Если функция возвращает aggregate, hidden result PTR64 занимает slot 0. Для каждого аргумента:

```
offset = align_up(offset, alignof(argument))
argument.offset = offset
offset += sizeof(argument)

argument_block_size = align_up(offset, 16)
```

Base блок аргументов (argument block) перед CALL должен быть выровнен на 16 bytes. CPU не знает типов аргументов и не вставляет padding автоматически: это правило assembler/compiler ABI.



base блок аргументов (argument block) = caller SP перед CALL; поэтому он выровнен на 16 bytes
после PUSHFP; SETFP тот же base находится по FP + 16

Рис. 5: CALL добавляет адрес возврата (return PC) перед уже сформированным блок аргументов (argument block).

После CALL callee видит адрес возврата (return PC) по SP+0, а начало блок аргументов (argument block) — по SP+8. RET снимает **только** адрес возврата (return PC). Весь блок аргументов (argument block) после возврата освобождает caller.

14.2. Простые PUSH остаются допустимым сокращением

Старый right-to-left PUSH spelling остаётся корректным там, где он создаёт тот же layout и оставляет SP кратным 16. Например два r64 занимают ровно 16 bytes:

```
PUSH    r64@arg2
PUSH    r64@arg1
CALL    f
ADJSP   16
```

Для одного r64 блок аргументов (argument block) всё равно должен занимать 16 bytes:

```
ADJSP   -16
STSP    0, r64@arg1
CALL    f
ADJSP   16
```

14.3. CALL, RET, FP и PSW

CALL и CALLP помещают 64-bit next PC в STACK; RET снимает его обратно в PC. Return address является raw PROGRAM byte offset (CODE64), а не PTR64. Регистра RA/LR нет.

CALL/RET не изменяют FP и не сохраняют PSW автоматически. Arithmetic/status bits считаются caller-clobbered согласно принятому ABI, а persistent control bits должны быть восстановлены callee перед обычным возвратом, если callee их изменял. Сохранение FP — отдельная обязанность функций, которые выбрали frame-pointer convention.

CPU выполняет только механические действия CALL/RET. Типы аргументов, padding, hidden-result slot, владение блок аргументов (argument block) и сохранение долгоживущего состояния — контракт ABI между caller, callee и compiler.

15. Return Value Area

Верхние 8192 bytes register file зарезервированы ABI для канонического scalar return value. Это не отдельный физический регистр: результат размещается в обычном перекрывающемся register file так, чтобы **старший конец выбранного physical view совпадал с концом register file**.

View	Physical size	Канонический return register
r8	1 byte	r8@0x7ffff
r16	2 bytes	r16@0x7fffe
r32	4 bytes	r32@0x7fffc
r64	8 bytes	r64@0x7fff8
r4096	512 bytes	r4096@0x7fe00
r65536	8192 bytes	r65536@0x7e000
c64	8 bytes	c64@0x7fff8
c4096	512 bytes	c4096@0x7fe00
c65536	8192 bytes	c65536@0x7e000

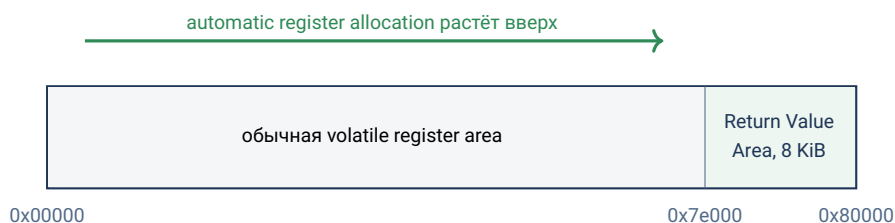


Рис. 6: Верхние 8 KiB register file зарезервированы под канонический scalar result.

Return Value Area является volatile across CALL. Если caller должен сохранить результат через следующий вызов функции, он обязан скопировать его в собственный register slot или memory object.

16. PTR64: сегментированный виртуальный

указатель PTR64 является 64-битным архитектурным указателем, но не host pointer. Старшие 4 бита выбирают segment selector, младшие 60 - byte offset внутри сегмента.

SEG bits 63..60 4 bits	BYTE OFFSET bits 59..0 60 bits
------------------------------	--------------------------------------

Каждый selector получает виртуальный диапазон 260 bytes = 1 EiB. Назначения 0x0..0x7 зафиксированы архитектурой; 0x8..0xF остаются резервом.

SEG	Имя	Назначение	Guest dereference
0x0	DATA	static/global objects	R/W
0x1	STACK	call/automatic storage	R/W
0x2	PROGRAM	machine code	R only
0x3	CONTROL	run control / configuration	R only
0x4	INPUT	environment -> model	R only

0x5	OUTPUT	model -> environment	R/W
0x6	TRACE	current telemetry frame	R/W
0x7	HEAP	runtime dynamic storage	R/W, low guard excluded
0x8..0xF	reserved	future spaces	POINTER_FAULT

$PTR64 = (SEG \ll 60) \mid OFFSET$

$SEG(ptr) = (ptr \gg 60) \& 0xF$

$OFFSET(ptr) = ptr \& 0xFFFFFFFFFFFFFFFF$

NULL остаётся единственным all-zero value. $0x0000000000000000 = PTR64(DATA, 0)$. Назначение новых selectors не меняет NULL semantics. $DATA[0..15]$ является NULL guard; $HEAP[0..15]$ является отдельным heap guard, но $PTR64(HEAP, 0)$ не является NULL.

ADDR64 и PTR64 - не одно и то же. ADDR64 в прямых LOAD/STORE остаётся raw DATA offset. PTR64 является runtime value с selector+offset и используется косвенными memory/ control-flow operations.

17. Создание указателей: LEA, LEASP, LEAFP

17.1. LEA: полный PTR64 immediate

LEA r64@dst, symbol_or_expression

machine: $OPCODE16 + REG24 + PTR64IMM = 13$ bytes runtime: r64@dst = decoded PTR64 immediate

LEA сохраняет прежний размер 13 bytes, но его 64-bit machine field теперь нормативно трактуется как полный PTR64 immediate, а не как DATA-only ADDR64. Ассемблер определяет selector по section, к которой принадлежит label:

Label section	LEA result
.data	$PTR64(DATA, offset)$
.control	$PTR64(CONTROL, offset)$
.input	$PTR64(INPUT, offset)$
.output	$PTR64(OUTPUT, offset)$
.trace	$PTR64(TRACE, offset)$
.init / .text / .fini	$PTR64(PROGRAM, offset)$
.input	

force:

```
.real 128, 0.0
.text
```

_life:

```
LEA r64@0x00100, force ; PTR64(INPUT, offset(force))
LOADP r128@0x00200, r64@0x00100
```

Чистое absolute/.equ expression без section label сохраняет историческую DATA semantics. Для DATA-label binary encoding остаётся совместимым с прежним LEA: selector DATA равен нулю, поэтому старший nibble 64-bit field не меняет bytes.

17.2. LEASP и LEAFP

```
LEASP r64@dst, signed_offset -> PTR64(STACK, SP + signed_offset)
LEAFP r64@dst, signed_offset -> PTR64(STACK, FP + signed_offset)
```

Эти instructions остаются специальными быстрыми constructors для automatic stack objects. Формирование one-past pointer допустимо при offset, равном architectural segment size; dereference всё равно требует, чтобы полный transfer range был valid.

18. Косвенная адресация: LOADP и STOREP

```
LOADP R/C, r64(PTR64) ; memory[ptr] -> register
STOREP r64(PTR64), R/C ; register -> memory[ptr]
```

machine size = $OPCODE16 + REG24 + REG24 = 8$ bytes

Pointer operand всегда является physical r64. Размер transfer задаётся шириной value-register; bytes копируются без endian conversion.

Сегмент	LOADP	STOREP	Причина политики доступа
DATA	да	да	обычная изменяемая память
STACK	да	да	автоматические объекты и кадры вызовов
PROGRAM	да	нет	код доступен для чтения, запись запрещена
CONTROL	да	нет	конфигурация принадлежит runtime/supervisor
INPUT	да	нет	входные воздействия принадлежат среде
OUTPUT	да	да	модель формирует выходные данные
TRACE	да	да	модель формирует текущий telemetry frame
HEAP	да	да	динамическая guest-память; offsets 0..15 защищены
0x8..0xF	POINTER_FAULT	POINTER_FAULT	селектор не назначен

CPU не хранит type tag у PTR64. Pointer provenance, pointee type, object lifetime, ownership и масштабирование на sizeof(T) являются ответственностью compiler/runtime ABI. CONTROL и INPUT read-only именно для guest CPU; host runtime API имеет право обновлять их в разрешённые lifecycle points.

19. Pointer arithmetic

Pointer arithmetic выполняется только над 60-битным BYTE OFFSET. Четырёхбитный SEG никогда не участвует в переносе или заёме и всегда сохраняется без изменения. Единица смещения - один байт.

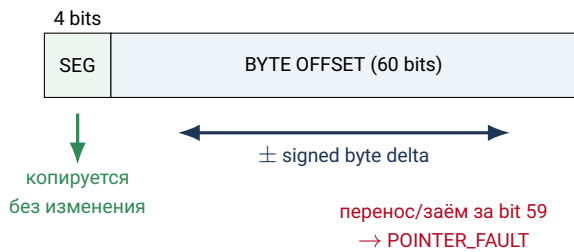


Рис. 7: PADD/PSUB изменяют только OFFSET; SEG остаётся неизменным.

19.1. Набор инструкций

Мнемоника	Семантика	Размер
PADD d,p,n	$d = p + \text{signed64}(n)$	11 bytes
PADDI d,p,k	$d = p + \text{signed0FF64}(k)$	16 bytes
PSUB d,p,n	$d = p - \text{signed64}(n)$	11 bytes
PSUBI d,p,k	$d = p - \text{signed0FF64}(k)$	16 bytes
PDIFF d,a,b	$d = \text{OFFSET}(a) - \text{OFFSET}(b)$, signed byte count	11 bytes

Во всех register-формах операнды обязаны быть физическими r64. Для PADD/PSUB третий r64 трактуется как знаковое 64-битное число в дополнительном коде. Для PADDI/PSUBI смещение кодируется существующим OFF64 в canonical MSB-first byte order. Реализация использует фиксированные типы LibMPCPU `__mpu_int64_t` и `__mpu_uint64_t`, выбранные configure с гарантированной шириной 64 бита.

```
PADD r64@dst, r64@ptr, r64@delta
PADDI r64@dst, r64@ptr, 24
PSUB r64@dst, r64@ptr, r64@delta
PSUBI r64@dst, r64@ptr, -16
PDIFF r64@bytes, r64@p2, r64@p1
```

19.2. Границы и fault

Результирующий offset обязан лежать в диапазоне 0 .. 0x0fffffffffffff. Попытка выйти ниже нуля или выше 60-битного предела вызывает CPU_EXEC_POINTER_FAULT. Destination при fault не изменяется. PSW pointer arithmetic не изменяет.

Важно: PADD/PSUB не выполняют dereference и поэтому не сравнивают offset с фактически выделенным размером DATA/STACK/PROGRAM. Они проверяют только архитектурную 60-битную границу PTR64. Фактические границы проверяются позднее, при выполнении LOADP/STOREP. Это позволяет формировать one-past и промежуточные указатели; их языковая допустимость остаётся задачей compiler.

19.3. PDIFF

PDIFF разрешён только если segment selectors обоих PTR64 совпадают. Разность двух 60-битных offsets всегда помещается в signed 64 bits и возвращается как число байтов. Разные SEG дают CPU_EXEC_POINTER_FAULT. Само значение SEG может быть и зарезервированным: арифметика его только сравнивает/сохраняет и не пытается dereference.

```
; conceptual C lowering
p + n      -> PADDI/PADD with n * sizeof(*p) bytes
p - n      -> PSUBI/PSUB with n * sizeof(*p) bytes
p2 - p1    -> PDIFF byte_delta; compiler scales by sizeof(*p)
```

ISA сознательно не содержит операции pointer + pointer. Типовая арифметика, масштабирование элементов, provenance и правила сравнения принадлежат high-level compiler.

20. FP как специальное архитектурное состояние

FP (*frame pointer, указатель кадра*) — третий специальный 64-битный координатный регистр CPU рядом с PC и SP. Он не занимает байты register file и не является PTR64. В контексте CPU FP хранит raw byte offset внутри STACK.

Состояние	Физический смысл	Назначение
PC	64-bit offset в PROGRAM	адрес следующей machine instruction
SP	64-bit offset в STACK	подвижная граница живой stack storage
FP	64-bit offset в STACK	стабильная база текущего stack frame
PSW	32-bit status/control word	FLAGS, traps/interrupt mode и status bits

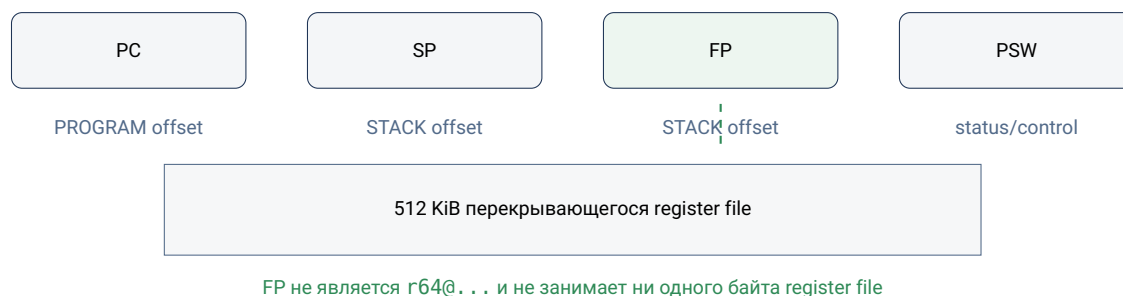


Рис. 8: FP относится к специальному состоянию CPU и не перекрывается с register file.

При создании STACK начальное состояние равно $SP=FP=stack_size$. FP является optional: leaf/simple function может вообще не устанавливать frame pointer. Compiler включает FP, когда нужна стабильная база для fixed locals, VLA/runtime-sized objects, spills, сложных frames или удобной диагностики.

21. Кадр стека (stack frame): пролог и эпилог

Сразу после обычного CALL layout определяется относительно входного SP:

Адрес	Содержимое
SP+0	адрес возврата (return PC)
SP+8	начало блок аргументов (argument block)
SP+8+k	argument slot с block offset k

Функция, использующая FP, выполняет канонический prologue:

```
PUSHFP
SETFP
ADJSP    -fixed_frame_size
```

PUSHFP сохраняет старый FP как 64-bit value и уменьшает SP на 8. SETFP фиксирует текущий SP как базу нового frame. После этого fixed locals получают стабильные отрицательные offsets от FP, а arguments — стабильные положительные offsets.

Offset от FP	Объект	Комментарий
FP+0	сохранённый FP caller-a	значение, записанное PUSHFP
FP+8	адрес возврата (return PC)	значение, записанное CALL
FP+16	блок аргументов (argument block) +0	начало блока аргументов caller-a
FP+16+k	argument slot	аргумент с block offset k
FP- ...	fixed locals/spills	offsets назначает compiler/frame layout

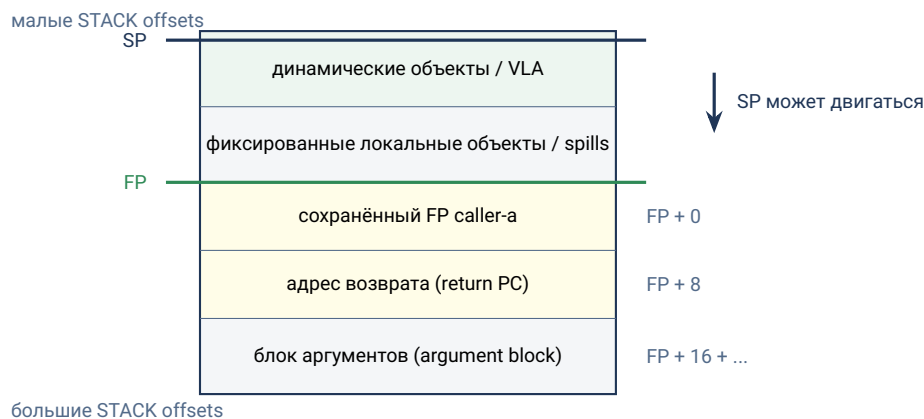


Рис. 9: Frame с FP: fixed objects имеют стабильные offsets, а SP может двигаться ниже FP.

21.1. Инструкции управления кадром (frame instructions)

Код операции	Мнемоника	Семантика
0x5030	PUSHFP	SP -= 8; STACK[SP] = FP
0x5031	SETPP	FP = SP
0x5032	RESTSP	SP = FP
0x5033	POPFP	FP = STACK[SP]; SP += 8

Канонический epilogue:

```

RESTSP    ; discard dynamic and fixed frame storage below FP
POPFP     ; restore caller FP; SP now points to адрес возврата (return PC)
RET       ; restore PC

```

CALL/RET сами FP не меняют. Сохранение и восстановление frame pointer является частью prologue/epilogue только тех функций, которые решили использовать FP. Leaf/simple functions могут работать только через SP.

Главное назначение FP — отделить стабильную геометрию frame от временных движений SP. Fixed local всегда остаётся по тому же FP - offset, даже если функция создаёт outgoing блок аргументов (argument block)s или runtime-sized temporary storage.

22. Адресация относительно FP (FP-relative addressing)

Три инструкции являются прямыми аналогами SP-relative операций, но используют стабильную базу FP:

Код операции	Мнемоника	Машинный формат	Семантика
0x5040	LDFF R, OFF64	2 + 3 + 8 = 13 bytes	R = STACK[FP + off]
0x5041	STFF OFF64, R	2 + 8 + 3 = 13 bytes	STACK[FP + off] = R
0x5042	LEAFF r64, OFF64	2 + 3 + 8 = 13 bytes	r64 = PTR64 (STACK, FP + off)

OFF64 - signed 64-bit byte displacement в canonical MSB-first encoding. Для LDFF/STFF CPU проверяет, что полный range объекта помещается в физически allocated STACK. LEAFF проверяет вычисленный offset и разрешает pointer formation на границе stack_size (one-past), но не выполняет dereference.

```
; fixed local scalar at FP-8
STFP    -8, r64@0x00100
LDFP    r64@0x00108, -8

; pointer to that local object
LEAFP   r64@0x00110, -8
LOADP   r64@0x00118, r64@0x00110
```

22.1. Смещения блока аргументов после установки FP

После PUSHFP; SETFP начало блок аргументов (argument block) всегда имеет фиксированный адрес FP+16. Но offsets individual arguments вычисляются внутри блока по их ABI alignment.

```
FP + 0      сохранный FP callera-
FP + 8      адрес возврата (return PC)
FP + 16     блок аргументов (argument block) base
FP + 16 + k аргумент at block offset k
...
```

Без aggregate return первый explicit argument имеет block offset 0. При aggregate return hidden result PTR64 занимает block offset 0, а первый explicit argument начинается с align_up(8, alignof(arg1)). Поэтому, например, для 8-byte aligned arg1 это FP+24, а для 16-byte aligned arg1 - FP+32.

Compiler invariant: между SETFP и соответствующим POPFP FP рассматривается compiler-ом как стабильная база. PUSH/POP/CALL/ADJSP/SETSP не меняют FP.

23. Динамическое выделение памяти в STACK и SETSP

Dynamic objects не требуют отдельной ALLOCA-инструкции. LibMCPUs использует уже существующие PTR64 + pointer arithmetic и одну операцию установки SP:

Код операции	Мнемоника	Машинный формат	Семантика
0x5050	SETSP r64(PTR64)	2 + 3 = 5 bytes	если SEG=STACK и offset входит в allocated STACK, SP=OFFSET(ptr)

SETSP принимает только физический r64 с PTR64. DATA/PROGRAM/reserved segment дают CPU_EXEC_POINTER_FAULT; STACK offset больше фактического stack_size даёт CPU_EXEC_STACK_FAULT. При fault SP не изменяется.

23.1. Выделение N байт во время выполнения

```
; r64@0x00118 contains runtime byte count N
LEASP   r64@0x00110, 0      ; mark = PTR64(STACK, текущий SP)
PSUB    r64@0x00120, r64@0x00110, r64@0x00118
SETSP   r64@0x00120        ; reserve N bytes downward
LEASP   r64@0x00130, 0      ; &object == new stack top

; ... use object through r64@0x00130 ...

SETSP   r64@0x00110        ; release this dynamic scope
```

Pointer arithmetic работает в байтах. Масштабирование $n * \text{sizeof}(T)$ и alignment выполняет compiler. CPU не добавляет hidden padding.

23.2. Понижение VLA (VLA-style lowering)

```
; VLA/dynamic object lowering.
; Assume compiler has already computed the required aligned byte count
; into r64@0x00118 at runtime.
```

f:

```
PUSHFP
SETFP
ADJSP   -16                ; fixed part of frame, if needed

LEASP   r64@0x00110, 0      ; save stack mark before dynamic object
PSUB    r64@0x00120, r64@0x00110, r64@0x00118
SETSP   r64@0x00120
LEASP   r64@0x00130, 0      ; PTR64 to first byte of dynamic object

; ... access object through r64@0x00130 ...

RESTSP                ; releases dynamic and fixed frame storage
```

POPFP
RET

Размер dynamic object. CPU получает число байтов уже вычисленным в обычном r64. Вычисление $n * \text{sizeof}(T)$, overflow checks и align-up относятся к compiler lowering и могут использовать доступные integer operations. Механизм FP/SETSP от конкретного способа вычисления размера не зависит.

23.3. Вложенные динамические области

Runtime-sized objects удобно освобождать не по размеру, а восстановлением заранее сохранённой stack mark:

```
LEASP    r64@0x00200, 0      ; внешняя отметка
; allocate object A
...
LEASP    r64@0x00208, 0      ; внутренняя отметка
; allocate object B
...
SETSP    r64@0x00208         ; release B-scope
...
SETSP    r64@0x00200         ; release A-scope
```

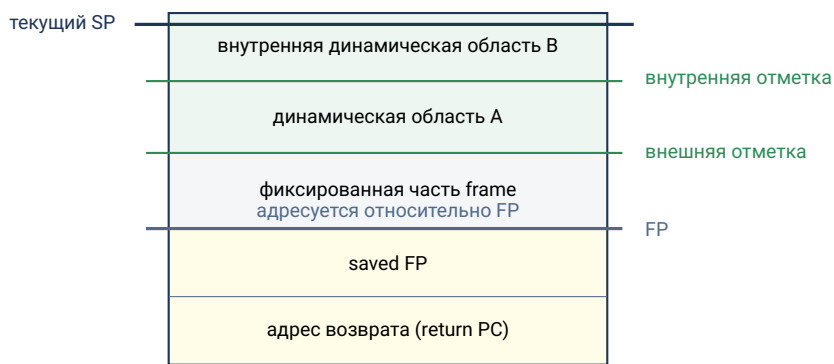


Рис. 10: Dynamic scopes меняют SP, но FP и fixed frame остаются стабильными.

Когда frame установлен, compiler не должен использовать SETSP для выхода выше FP и не должен захватывать caller-owned bytes. CPU аппаратно проверяет границы всего allocated STACK; принадлежность конкретных bytes текущему frame контролирует compiler/ABI.

24. С-подобная модель автоматических объектов (automatic objects)

STACK является полноценной памятью automatic storage duration. Локальные структуры, union, массивы, VLA и runtime-sized temporaries живут в stack frame. Scalar locals могут оставаться в register file либо spill-иться в STACK.

Объект	Рекомендуемая база	Время жизни
fixed scalar spill	FP + fixed negative offset	до RESTSP / возврата
fixed struct/array	FP + fixed negative offset	до RESTSP / возврата
VLA/runtime object	PTR64 от LEASP после SETSP	до восстановления соответствующей stack mark / RESTSP
argument	FP + positive offset; без FP — SP-relative	в пределах call

PTR64 на automatic object имеет обычный C-like lifetime: после освобождения dynamic scope или frame он становится семантически недействительным, даже если underlying STACK bytes физически ещё существуют.

Alignment не является скрытой функцией CPU. Для ABI-conforming frame/argument layout используется максимум 16 bytes natural alignment и 16-byte CALL alignment; offsets и padding явно создаются assembler/compiler.

25. Формальный ABI аргументов

Категория	ABI representation
Integer scalar	by value в блок аргументов (argument block)
Real scalar	by value в блок аргументов (argument block)
Complex scalar	by value в блок аргументов (argument block)
PTR64	8-byte scalar by value
struct/union by value	полное object representation в блок аргументов (argument block)
struct/union by pointer	PTR64 на исходный object
array parameter в C-like semantics	PTR64 на первый element

Argument block base имеет alignment 16. Каждый slot получает ABI alignment своего значения; полный размер блока округляется к 16. Padding является частью call layout, но не частью значения scalar argument.

```
; one r64 argument
ADJSP  -16
STSP   0, r64@0x00000
CALL   f
ADJSP  16
```

На входе callee блок аргументов (argument block) начинается по SP+8. Если callee выполняет PUSHFP; SETFP, тот же block начинается по FP+16. First explicit argument без hidden result имеет block offset 0; при hidden result его offset вычисляется по обычному align-up rule и не обязан быть фиксированным.

Почему это важно. Если hidden PTR64 занимает block offset 0, следующий argument с alignment 8 начнётся с +8, а argument с alignment 16 - с +16. Поэтому ABI не кодирует постоянную формулу «первый explicit argument всегда hidden+8».

26. Структуры и массивы

26.1. Fixed local struct

```
PUSHFP
SETFP
ADJSP -16

STFP  -16, r64@0x00100
STFP  -8, r64@0x00108
LEAFP r64@0x00110, -16 ; &local_struct
...
```

26.2. Structure by value

Caller создаёт exact copy complete object representation в argument area. Для memory-to-memory переноса используется BMOV/BMOVI, поэтому aggregate не обязан проходить через register file и его размер не ограничен 8192 bytes максимального register view.

```
; struct S s occupies 48 bytes in caller frame
LEAFP r64@0x00100, -96 ; src = &s
ADJSP -48 ; 48-byte блок аргументов (argument block), already 16-aligned
LEASP r64@0x00108, 0 ; dst = argument slot +0
BMOVI r64@0x00108, r64@0x00100, 48
CALL f
ADJSP 48 ; caller cleanup
```

26.3. Local array / VLA

Массив фиксированного размера получает постоянное отрицательное смещение от FP. VLA получает динамическую область STACK через LEASP/PSUB/SETSP; compiler сохраняет PTR64 на начало массива в регистре или spill-слоте, если текущий SP затем будет двигаться дальше.

26.4. Array argument

В обычной C-семантике array parameter передаётся как PTR64 на первый element. Копирование всего массива не происходит.

27. Aggregate return: hidden result pointer

Scalar integer/real/complex/pointer возвращаются через Return Value Area. Structure/union произвольного размера использует caller-provided storage и hidden PTR64.

Hidden result is argument slot 0. Caller помещает PTR64 destination в block offset 0, затем layout explicit arguments начинается после него с обычным align-up. В callee без FP block base находится по SP+8; после PUSHFP; SETFP - по FP+16.

```
; Example: hidden result PTR64 + aggregate argument aligned to 8
; block offset +0 hidden result PTR64
; block offset +8 explicit aggregate
; block is rounded up to 16-byte total size

; callee with FP
PUSHFP
SETPF
LDFP    r64@0x00100, 16      ; hidden result = block +0
; explicit offsets are determined by ABI layout, not by a universal constant
...
RESTSP
POPPF
RET
```

Callee записывает aggregate bytes через hidden PTR64. RET не копирует aggregate и не трогает Return Value Area для такого результата. Caller освобождает весь блок аргументов (argument block) одним ADJSP.

28. Block copy: BMOV и BMOVI

Block-copy переносит произвольное число байтов непосредственно между архитектурными memory spaces через PTR64. Register file используется только для самих указателей и, в форме BMOV, для runtime byte count. Содержимое копируемого объекта не проходит через REG24.

Код операции	Мнемоника	Машинный формат	Размер
0x0150	BMOV dst, src, count	OPCODE16 + REG24 + REG24 + REG24	11 bytes
0x0151	BMOVI dst, src, IMM64	OPCODE16 + REG24 + REG24 + IMM64	16 bytes

Все register operands этих двух инструкций обязаны быть физическими r64. Первые два r64 содержат PTR64 destination и source. Третий r64 у BMOV интерпретируется как unsigned 64-bit byte count (`__mpu_uint64_t`). У BMOVI byte count находится непосредственно в PROGRAM как unsigned IMM64, MSB-first.

Семантика всегда memmove. Если source и destination ranges перекрываются в одном сегменте, результат обязан быть таким же, как у C memmove(). Направление физического копирования не является частью ISA и выбирается реализацией.

```
BMOVI r64@dst_ptr, r64@src_ptr, 64
; copies exactly 64 bytes

LDI    r64@size, 4096
BMOV   r64@dst_ptr, r64@src_ptr, r64@size
; runtime-sized 4096-byte copy
```

28.1. Разрешённые memory spaces

Segment	Source	Destination
DATA (0x0)	да	да
STACK (0x1)	да	да
PROGRAM (0x2)	да	нет
CONTROL (0x3)	да	нет
INPUT (0x4)	да	нет
OUTPUT (0x5)	да	да
TRACE (0x6)	да	да
HEAP (0x7)	да, кроме low guard	да, кроме low guard
0x8..0xF	POINTER_FAULT	POINTER_FAULT

BMOV/BMOVI используют те же segment access rules, что LOADP/STOREP. CONTROL и INPUT могут быть sources, но не destinations. OUTPUT/TRACE/HEAP являются обычными writable data-object spaces. PROGRAM остаётся read-only source.

28.2. Проверка ranges и atomicity

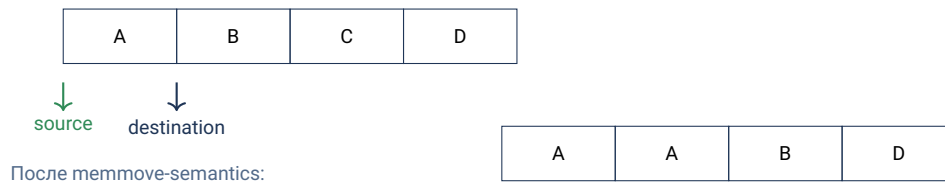
До первого изменения destination CPU обязан:

1. прочитать оба PTR64 и byte count; 2. проверить source segment и полный range [src, src+count); 3. проверить destination segment, write permission и полный range [dst, dst+count); 4. только после успешных проверок выполнить block move.

Правило атомарной ошибки: если диапазон source или destination недопустим, байты destination не изменяются. BMOV/BMOVI не меняют PSW, SP, FP и сами PTR64-регистры.

PTR64 offset может быть one-past физического сегмента только при нулевом размере. `count == 0` является корректным по-оп после проверки `segment/access policy`; запись в PROGRAM остаётся запрещённой даже для destination PROGRAM.

До `BMOVIdst=src+8, count=16`:



После `memmove-semantics`:

Рис. 11: `BMOV/BMOVI` определены как `memmove`, поэтому `overlapping ranges` обрабатываются корректно.

28.3. Lowering: assignment структуры

```
; C-like: dst = src;      sizeof(struct S) = 64
LEAFP   r64@0x00100, -64    ; &dst
LEAFP   r64@0x00108, -128   ; &src
BMOVI   r64@0x00100, r64@0x00108, 64
```

CPU копирует complete object representation, включая padding bytes. Понимание полей структуры остаётся задачей compiler.

28.4. Lowering: aggregate argument by value

```
; source aggregate S lives at FP-96, sizeof(S)=48, align <=16
LEAFP   r64@0x00100, -96
ADJSP   -48                      ; complete 16-aligned блок аргументов (argument block)
LEASP   r64@0x00108, 0
BMOVI   r64@0x00108, r64@0x00100, 48
CALL    consume_S
ADJSP   48
```

Для нескольких arguments assembler/compiler вычисляет offsets внутри цельного блок аргументов (argument block) в declaration order, соблюдая alignment каждого slot; общий размер округляется к 16.

28.5. Lowering: aggregate return через hidden PTR64

```
; callee with FP; hidden result PTR64 at FP+16
LDFP    r64@0x00100, 16          ; destination supplied by caller
LEAFP   r64@0x00108, -128        ; local result object
BMOVI   r64@0x00100, r64@0x00108, 128
RESTSP
POPPF
RET
```

RET по-прежнему не знает о структуре. BMOVI просто переносит готовый object representation в caller-provided storage.

28.6. Runtime-sized arrays и VLA

```
; dst_ptr and src_ptr are PTR64; r64@0x00110 = runtime byte count
BMOV    r64@0x00100, r64@0x00108, r64@0x00110
```

Compiler предварительно вычисляет `element_count * sizeof(T)`, выполняет требуемые overflow checks и передаёт итоговый byte count. BMOV не знает element type и не выполняет масштабирование.

28.7. memcpy и memmove в C-like frontend

Одна архитектурная операция достаточна для обеих high-level семантик. Вызов `memmove` естественно lowering-ится в `BMOV/BMOVI`. `memcpy` также может lowering-иться в ту же инструкцию: `memmove-semantics` являются более сильной гарантией и не меняют результат программы при неперекрывающихся ranges. Frontend всё равно может диагностировать нарушение своих собственных правил aliasing/overlap.

29. ABI data layout и assembler alignment

29.1. Natural alignment

Базовое правило LibMCPUCPU ABI для обычного scalar/object storage:

`natural_alignment = min(sizeof(object), 16)`

```
ABI_MAX_ALIGN    = 16
STACK/CALL_ALIGN = 16
```

Это означает, например: r8 -> align 1, r32 -> align 4, r64/PTR64 -> align 8, r128 и все более крупные numeric objects -> align 16. Размер объекта при этом не меняется: real4096 занимает 512 bytes, но ABI-natural alignment равен 16, а не 512.

29.2. Assembler не вставляет hidden alignment

.integer, .real и .complex записывают bytes непосредственно в текущую DATA position. Только явный .align N меняет position. Благодаря этому один и тот же assembler поддерживает normal и packed layout.

```
; normal ABI-like layout
.data
.align 16
```

obj:

```
.integer 8, 1      ; offset +0
.align 4
.integer 32, 2      ; next 4-byte aligned field
.align 2
.integer 16, 3
.align 4            ; trailing struct padding

; packed binary layout: no implicit padding
```

packed_obj:

```
.integer 8, 1
.integer 32, 2
.integer 16, 3
```

Explicit .align может быть больше 16 (например 64, 512 или 4096) для специальных таблиц, cache/page oriented data или внешнего формата. Такое требование относится к guest offset и не увеличивает ABI natural alignment типа.

29.3. Normal и packed aggregate layout

Для normal aggregate поля размещаются с их natural alignment, alignment aggregate равен максимуму alignment полей (не более 16), а sizeof округляется вверх до aggregate alignment. Packed aggregate имеет alignment 1, не имеет automatic internal/trailing padding и сохраняет внутренний layout вложенного normal aggregate как самостоятельного типа.

Bit-fields. LibMCPUs ABI не требует отдельного bit-field layout: необходимые операции представляются обычными integer masks/shifts. Это сохраняет binary layout явным и переносимым.

Часть II

Memory backing, представление объектов и assembler

30. Host backing memory spaces и loader

Host allocation не является частью PTR64. Активные architectural memory spaces имеют независимый anonymous mmap() backing; guest pointer содержит selector+offset и никогда не содержит host virtual address.

30.1. Anonymous mmap only

```
mmap(NULL, capacity,
     PROT_READ | PROT_WRITE,
     MAP_PRIVATE | MAP_ANONYMOUS,
     -1, 0)
```

PROGRAM, DATA, STACK, CONTROL, INPUT, OUTPUT, TRACE и HEAP получают отдельные mappings. Loader читает image bytes обычным read() в anonymous memory. PROGRAM после загрузки переводится host-side в read-only mapping; guest access policy дополнительно проверяется instruction resolver. CONTROL/INPUT сохраняются host-writable, потому что runtime должен иметь возможность менять configuration/input между фазами исполнения, но guest STOREP в них запрещён.

30.2. Architectural size, capacity и HEAP limit

Величина	Смысл
segment.size	точное число байтов, доступных guest-программе
segment.capacity	округлённый по страницам размер host mapping
heap.size	текущий guest-visible размер HEAP
heap.capacity	текущая вместимость host backing HEAP
heap.limit	настроенный неизменяемый максимальный размер HEAP

Все CPU bounds checks используют architectural size, а не host capacity. HEAP additionally reserves offsets 0..15; первый ordinary heap byte имеет offset CPU_HEAP_BEGIN = 16.

Relocatable host backing. PTR64(HEAP,offset) не зависит от host base. Поэтому будущая реализация heap growth может создать larger anonymous mapping, скопировать bytes и заменить host base, не меняя ни одного guest pointer.

30.3. MELF loader и окончательный process image

Начиная с MELF ABI, development sidcar images не являются форматом запуска. mcpu - ld создаёт ET_EXEC/ET_DYN с segment metadata, а mcpu - run валидирует target description, program-model metadata, program headers и relocation records до передачи управления guest. Для ABI v2 все preload-модули сначала полностью читаются и валидируются, затем назначаются независимые PROGRAM/DATA bases, после чего применяются retained relocations. PROGRAM после relocation phase переводится в read-only host mapping.

31. Fault, atomicity и invariants

- REG24 проверяет width, natural alignment и границы register file.
- JMP/CALLP требуют PROGRAM PTR64: NULL, другой SEG или offset вне program.size дают POINTER_FAULT до изменения PC/SP/STACK.
- CALLP сначала валидирует target и stack space; return PC записывается только после успешной полной проверки.
- All-zero PTR64 является NULL и даёт POINTER_FAULT при indirect dereference.
- Assembler резервирует DATA[0..15]; ordinary DATA labels начинаются не раньше 0x10.
- CODE64 target должен ссылаться внутрь PROGRAM.
- STACK operations сначала проверяют bounds, затем меняют SP/FP/memory.
- PUSHFP проверяет место и корректность текущего FP до записи.
- POPFP сначала читает saved value и проверяет saved_fp <= stack_size; при fault ни FP, ни SP не меняются.
- SETFP требует allocated STACK и корректный SP.
- RESTSP требует FP <= stack_size.
- LD/STFP/LEAF/STFP используют signed OFF64 относительно FP и выполняют bounds checking.
- SETSP принимает только STACK PTR64; wrong segment -> pointer fault, out-of-range offset -> stack fault.
- Ни одна FP/SP management instruction не меняет PSW.
- CALL/CALLP/RET сами PSW-neutral; normal-function ABI считает PSW[15:0] volatile, TF/IF/MODE persistent control state, HF отдельным execution state.
- PADD/PADDI/PSUB/PSUBI проверяют 60-bit PTR64 offset, но не object lifetime.
- BMOV/BMOVI сначала проверяют complete source и destination ranges; при fault destination memory не изменяется. Overlap разрешён и имеет memmove semantics.
- CPU не обеспечивает ownership текущего frame: правило SP <= FP внутри FP-frame и stack-scope discipline обеспечиваются compiler/ABI.

Нельзя хранить долгоживущий указатель на automatic object после release scope/frame. PTR64 остаётся битово корректным, но

high-level program нарушает lifetime. Это семантическая ошибка compiler/program, а не PTR64 encoding fault.

32. Byte order и architectural object representation

LibMCPUCPU намеренно разделяет два независимых byte-order domain. Это один из фундаментальных ABI contracts: PROGRAM является переносимым сериализованным instruction stream, а register file и memory segments содержат runtime objects в target representation, совместимом с LibMPCPU.

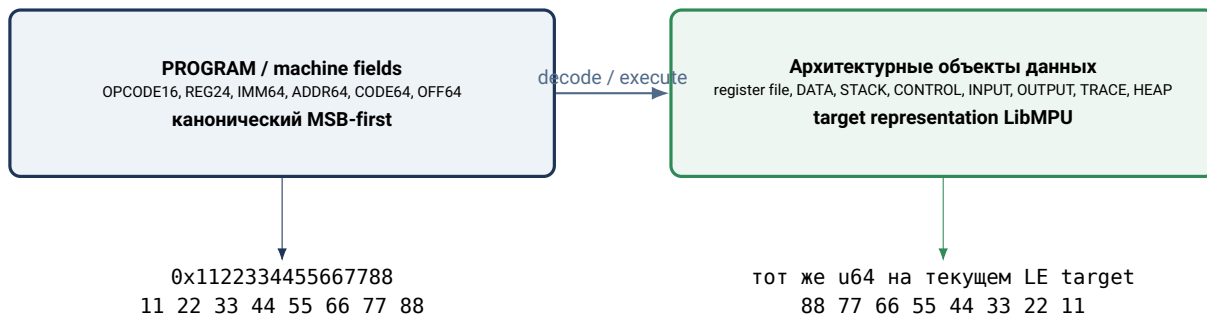


Рис. 12: Сериализация PROGRAM и целевое представление объектов данных являются разными byte-order domains.

32.1. PROGRAM: canonical MSB-first

Все multi-byte machine fields внутри PROGRAM кодируются старшим байтом вперёд. Это правило не зависит от host CPU, LibMPCPU byte order или будущего object-file container.

value = 0x1122334455667788

IMM64 / CODE64 / ADDR64 / OFF64 bytes: 11 22 33 44 55 66 77 88

REG24 также является canonical machine field: его три байта формируются из WIDTH и ADDRESS по уже определённому битовому layout. Decoder не делает C-cast на packed host structure.

32.2. Data-object domain

Register file, DATA, STACK, CONTROL, INPUT, OUTPUT, TRACE и HEAP используют одно target object representation. Для integer-like objects byte order берётся из target profile LibMCPUCPU, который совпадает с configured LibMPCPU:

CPU_DATA_BYTE_ORDER	=	MPU_BYTE_ORDER
CPU_DATA_WORD_ORDER	=	MPU_WORD_ORDER
CPU_DATA_BYTE_ORDER_*	=	corresponding LibMPCPU settings
CPU_DATA_WORD_ORDER_*	=	corresponding LibMPCPU settings

На текущей little-endian конфигурации:

u64 value 0x1122334455667788 address+0 88 address+1 77 address+2 66 address+3 55 address+4 44 address+5 33 address+6 22 address+7 11

Общее representation. Если один object скопирован между register file или любыми data-object spaces без числового преобразования, его bytes не переставляются. Endian conversion не является скрытым side effect memory instructions.

32.3. Явные helper families в реализации

Чтобы source code executor/assembler визуально не смешивал два domain, используются две группы accessors:

__cpu_code_get_u16 / __cpu_code_put_u16 __cpu_code_get_u32 / __cpu_code_put_u32 __cpu_code_get_u64 / __cpu_code_put_u64
 __cpu_data_get_u16 / __cpu_data_put_u16 __cpu_data_get_u32 / __cpu_data_put_u32 __cpu_data_get_u64 / __cpu_data_put_u64

code-helpers явно собирают/разбирают MSB-first fields. data-helpers работают с target object representation через byte-safe copy semantics. Слово data здесь означает architectural data-object domain, а не только PTR64 segment DATA=0x0; эти helpers применимы также к register-file и STACK bytes.

Именно поэтому выбрано имя __cpu_data_get_u*, а не __cpu_object_get_u*: в исходниках сразу виден контраст code vs data. Во избежание путаницы с segment DATA это значение термина закреплено в ABI documentation.

32.4. Integer, PTR64, return PC и overlapping registers

Signed integer использует two's-complement; unsigned - binary modulo 2^N . Их физический byte/word order является target data profile. PTR64 логически остаётся 4-bit SEG + 60-bit OFFSET, но когда он находится в r64, DATA field, STACK argument или structure member, это обычный 64-bit data object.

PTR64(PROGRAM, 0x1234) = 0x20000000000001234

current LE target bytes: 34 12 00 00 00 00 00 20

Return PC, который CALL/CALLP помещают в STACK, тоже является 64-bit STACK data object. В самой CALL instruction CODE64 target encoded MSB-first; pushed return PC хранится в target data representation. RET читает его как data object и устанавливает логический PC.

Overlapping register views физически наблюдают одни и те же bytes. Поэтому endian-sensitive assembler code, который разбирает широкое integer value через меньшие overlapping views, является target-specific low-level code. Это допустимо и сознательно является частью ABI.

32.5. Real и Complex

LibMCPUs не вводит отдельный формат floating/complex objects поверх LibMPUs. Для real<bits> применяется external object representation соответствующего LibMPUs real type при текущем target profile. Для complex<bits> используется LibMPUs complex representation двух real components половинной ширины.

Complex physical half order не фиксируется независимым правилом LibMCPUs. Он следует configured LibMPUs word-order profile. Нельзя считать универсальной истиной, что первые bytes complex object всегда являются real component, а вторые - imaginary component. Complex arithmetic должен использовать C-opcodes/LibMPUs representation либо явно target-specific byte code.

32.6. Assembler data directives

Directive inside .data создаёт target data object либо явно заданную raw byte sequence, а не canonical machine field:

```
.data
.align 8
```

x:

```
.integer 64, 0x1122334455667788
```

text16:

```
.ucs2 Андрей""
```

text8:

```
.utf8 Андрей""
```

raw:

```
.byte 0x11, 0x22, 0x33
```

```
.align 8
```

jump:

```
.cptr target
```

```
.text
```

target:

```
HALT
```

На текущем LE target bytes объекта x идут как 88 77 66 55 44 33 22 11. Но literal того же значения в LDI находится в PROGRAM как canonical 11 22 33 44 55 66 77 88. .integer/.int, .real и .complex генерируют target/LibMPUs object representation. .ucs2 пишет target-representation 16-bit code units, тогда как .utf8 и .byte являются raw byte sequences без endian conversion. .cptr генерирует target-representation PTR64(PROGRAM,label/ expression).

.cptr является каноническим mnemonic directive. Исторический .codeptr принимается assembler как compatibility alias. .label не используется: запись name: уже определяет label, тогда как .cptr создаёт объект-указатель. Полные правила строк, .equ и expressions приведены в разделе 34.

32.7. Byte-preserving memory operations

```
LOAD / STORE
LOADP / STOREP
PUSH / POP
LDSP / STSP
LDFP / STFP
BMOV / BMOVI
```

Эти операции не знают endian semantics конкретного объекта. Они переносят physical bytes. В частности:

DATA source –LOAD–> register A DATA source –BMOV–> DATA/STACK destination –LOAD–> register B

bytes(register A) == bytes(register B)

Точно так же structure/array/packed object может быть скопирован BMOV без type metadata и без byte swap. Padding bytes, если они являются частью complete object representation, тоже сохраняются.

32.8. Raw bytes, packed I/O и external byte order

char, byte arrays, UTF-8 и arbitrary binary blobs являются последовательностями байтов и сами по себе endian не имеют. packed означает отсутствие layout padding, но не означает network/ big-endian или little-endian protocol representation.

Если внешний файл/протокол требует конкретный byte order, assembler/program должен выполнить явное преобразование. Автоматическая endian conversion при LOAD/STORE запрещена этой ABI-моделью. Для явного разворота физического порядка байтов LibMCPUCPU предоставляет универсальную инструкцию BSWAP, подробно определённую в разделе 32. Она не подменяет понятие external byte order: решение о необходимости swap принимает программа или инструмент.

32.9. Target metadata в MELF

MELF уже содержит нормативную target metadata: byte order/container encoding, machine id и MCPUCPU target description. Loader не пытается эвристически перекодировать DATA/STACK-compatible objects при несовместимом profile. Образ либо совместим с текущей LibMCPUCPU target representation, либо отвергается до исполнения. PROGRAM machine fields при этом сохраняют отдельное canonical MSB-first кодирование.

32.10. Host alignment и unaligned accesses

Byte order не меняет ранее принятые alignment rules. Guest memory остаётся byte-addressable; unaligned accesses разрешены ISA. Реализация не должна читать произвольную guest memory через alignment-sensitive или strict-aliasing-unsafe casts. __cpu_data_get_u*/put используют

byte-safe copy semantics, а большие arbitrary objects перемещаются memcpu/memmove или LibMCPUCPU routines.

33. Pointer comparison, red zone, active/reserved

PTR64 и BSWAP Этот раздел фиксирует четыре низкоуровневых соглашения CPU/assembler ABI, необходимые до появления object/executable format: сравнение PTR64, отсутствие stack red zone, поведение активных/зарезервированных segment selectors и универсальное преобразование порядка физических байтов.

33.1. Equality и ordering для PTR64

PTR64 остаётся обычным 64-битным архитектурным значением. Для равенства/неравенства сравниваются все 64 бита: selector и 60-bit offset. Поэтому существующего integer compare достаточно:

```
ICMP    r64@p, r64@q
JZ      equal
JNZ     not_equal
```

Это правило корректно и для разных selector: DATA:0x100 и STACK:0x100 не равны, поскольку их полные PTR64 различаются. Обычный ICMP может также выдавать signed integer flags для двух pointer bit patterns, однако ABI не придаёт этому результату смысла pointer ordering.

Pointer ordering определяется только внутри одного segment selector и выполняется через PDIFF. Если selectors различны, PDIFF возвращает CPU_EXEC_POINTER_FAULT и destination остаётся неизменным.

```
PDIFF   r64@delta, r64@p, r64@q    ; signed byte difference
ITEST   r64@delta, r64@delta
JS      p_less_than_q
JZ      p_equal_q
```

Разность двух 60-bit offsets всегда помещается в signed 64-bit диапазон. CPU не знает границ high-level объектов/массивов: same-segment ordering является низкоуровневой адресной операцией, а object-domain restrictions будущего языка остаются задачей compiler/program.

33.2. Stack red zone отсутствует

```
CPU_ABI_STACK_RED_ZONE_SIZE = 0
```

STACK растёт вниз. Если текущий SP = X, байты ниже X не считаются принадлежащими функции как скрытая scratch-area. Функция должна сначала явно изменить SP через ADJSP или валидированный SETSP, а уже затем использовать выделенную область.

Важно: ISA может физически позволить SP-relative address с отрицательным OFF64, если он попадает в allocated STACK. Это не превращает область ниже текущего SP в ABI red zone. Использование невыделенных bytes является ABI violation, а не обязательным

аппаратным fault.

```
; неправильно как normal ABI scratch
STSP    -8, r64@tmp

; правильно
ADJSP   -16
STSP    0, r64@tmp
...
ADJSP   16
```

Отсутствие red zone упрощает будущие traps/interrupts: обработчик может строить собственный frame ниже текущего SP, не уничтожая скрытые temporaries leaf-функции.

33.3. Active и reserved PTR64 selectors

Selectors 0x0..0x7 имеют назначенную address-space semantics. Selectors 0x8..0xF остаются структурно допустимыми PTR64 values, но не имеют memory semantics. Это сохраняет возможность будущего расширения без изменения PTR64 encoding.

Операция над reserved PTR64	Правило
(0x8..0xF)	
MOV / SWAP / byte storage	разрешено: bit pattern можно копировать и хранить
ICMP equality / inequality	разрешено как full 64-bit comparison
PADD/PADDI/PSUB/PSUBI	разрешено; selector сохраняется, arithmetic только по 60-bit offset
PDIFF	разрешено только при одинаковом selector
LOADP / STOREP / BMOV / BMOVI	CPU_EXEC_POINTER_FAULT
JMPP / CALLP	CPU_EXEC_POINTER_FAULT
SETSP	CPU_EXEC_POINTER_FAULT

HEAP теперь является активным selector 0x7. Его low guard 0..15 недоступен для dereference, но это не делает HEAP:0 значением NULL. All-zero DATA:0 остаётся единственным NULL PTR64.

33.4. BSWAP: raw byte reversal

BSWAP имеет opcode 0x0103 и 8-byte machine form:

```
BSWAP r<bits>@dst, r<bits>@src
```

OPCODE16 + REG24(dst) + REG24(src)

2 bytes + 3 bytes + 3 bytes = 8 bytes

Оба operands обязаны иметь одинаковую физическую ширину. Ассемблер использует именно r<bits> syntax, подчёркивая raw/non-complex interpretation. Допустимы все существующие widths от r8 до r65536.

```
source:    b0 b1 b2 ... b(N-2) b(N-1)
```

destination: b(N-1) b(N-2) ... b1 b0

r8 является identity. Для r65536 разворачиваются все 8192 bytes. BSWAP не знает, является ли содержимое integer, PTR64, real, complex или частью packed record: операция определена только над физическими байтами.

PSW invariant. BSWAP не изменяет ни volatile status flags, ни TF/IF/MODE/HF. Width mismatch возвращает CPU_EXEC_WIDTH_FAULT до изменения destination.

33.5. In-place и snapshot semantics

Архитектурно BSWAP определяется как чтение полного source snapshot и только затем запись reversed bytes в destination. Поэтому in-place форма полностью определена:

```
BSWAP r64@0x00100, r64@0x00100
```

При текущем natural alignment equal-width register views не могут частично перекрываться иначе, чем полностью совпасть. Тем не менее snapshot formulation является нормативной и не зависит от внутренних оптимизаций исполнителя.

33.6. External binary formats

BSWAP не означает «to little endian» или «to big endian». Она лишь разворачивает bytes. Решение о swap принимает программа/tool, сравнивая external format с текущим CPU_DATA_BYTE_ORDER.

```
; native target-order value
LDI    r64@0x00000, 42

; физически reversed representation например( wire order, если он противоположен target)
BSWAP  r64@0x000008, r64@0x000000

; обратное преобразование
BSWAP  r64@0x000010, r64@0x000008
```

Для packed file/protocol records это даёт явное и предсказуемое преобразование. Packed layout и byte order остаются независимыми понятиями: packed убирает padding, BSWAP меняет порядок bytes конкретного field.

34. Актуальное покрытие LibMMPU и семейства ISA

Вычислительное ядро ISA не является минимальным proof-of-concept. Оно содержит полноценные integer, Real и Complex families, а также отдельные группы управления потоком, стеком, указателями и runtime boundary. Этот раздел фиксирует именно архитектурную группировку; точные значения всех OPCODE16 приведены в приложении В и генерируются из public cpu_orcode в include/libmcpu.h.

34.1. System и data-movement family

Системная группа содержит NOP, HALT, lifecycle instruction END, graceful EXIT, zero-operand SYSCALL и отдельную PSW/status subgroup: SETF, CLRF, CLRCPUF, CLRMPIF. Data-movement/address family содержит MOV, CLR, SWAP, raw-byte BSWAP, immediate forms LDI/LDR/LDC, direct DATA access LOAD/STORE, pointer construction/dereference LEA/LEASP/LOADP/STOREP, pointer arithmetic PADD/PADDI/PSUB/PSUBI/PDIFF и block move BMOV/BMOVI.

Тип operands задаёт opcode. REG24 не содержит integer/Real/Complex tag. Поэтому один и тот же physical register может последовательно участвовать в разных domains, но assembler проверяет semantic view там, где signature требует c<bits> либо конкретный r64 carrier. Executor независимо повторяет width/signature validation и не доверяет исходному тексту assembler.

34.2. Integer family

Integer operations работают с physical widths от 8 до 65536 bits. Базовые arithmetic/logical operations включают IADD/IADC/ISUB/ISBB, IAND/IOR/IXOR/INOT, INEG, compare/test, increment/decrement и exchange. Shift/rotate представлены как single-bit forms и variants с count в r64: logical/arithmetic shifts, ordinary rotates и rotates through carry.

Width-changing operations намеренно разделены на unsigned copy ICPU и signed conversion ICVT. Full multiply возвращает product расширенной ширины через IMUL/ISMUL; division имеет отдельные unsigned/signed четырехоперандные формы IDIV/ISDIV с quotient и remainder. IBITSLEN выдаёт длину integer в bits в r64.

Модульная арифметика LibMMPU (например modular multiply/power и GCD-related higher-level routines) намеренно не превращена в CPU orcodes. Она остаётся library-level functionality: ISA предоставляет primitives, а не дублирует всю API-поверхность LibMMPU.

34.3. Real family

Real arithmetic работает непосредственно с LibMMPU objects требуемой physical precision. Простые operations (copy, conversion, add/sub/mul/div, sign/abs, decomposition, predicates) поддерживают архитектурно допустимые Real widths. Transcendental family ограничена тем же верхним пределом, что и настроенная LibMMPU math backend; в проверенной конфигурации проекта это real16384.

В ISA присутствуют:

- arithmetic и square root;
- trigonometric/inverse trigonometric functions;
- logarithms, exponentials, hyperbolic и inverse hyperbolic functions;
- pow, hypot, atan2;
- floor/ceil/round, frexp/ldexp/logb/modf/remainder;
- signed/unsigned integer-to-Real и Real-to-integer conversions;
- classification/predicate family для sign, IND, NaN classes, infinity и signed zero;
- RCONST, создающий predefined values непосредственно в precision destination.

Широкий Real register не проходит через host double. Там, где instruction содержит immediate, exactness проверяется assembler-ом до emission; для произвольной высокой точности значение хранится как target Real object в data-like section и

загружается byte-preserving instruction LOAD/LOADP.

34.4. Complex family

Complex register view имеет полный physical width от c64 до c65536, а каждая компонента занимает половину. Arithmetic CADD/CSUB/CMUL/CDIV использует LibMPCPU complex primitives. Math family содержит CEXP/CSIN/CCOS/CSINH/CCOSH/CLOG/CSQRT/CP0W; для функций, требующих math backend, предел component precision соответствует Real math limit (то есть c32768 при real16384 components).

Отдельно представлены extraction/construction/conversion operations: CREAL/CIMAG, CRT0C, CGEN, CPOLAR, conjugation, absolute value/norm/argument, equality predicates и copy/precision conversion. Для mixed-domain signatures Real component register обязан иметь ровно половинную physical width соответствующего Complex object.

34.5. Control-flow и stack family

Прямые targets представлены CODE64; indirect targets - PROGRAM PTR64 в r64. Branch family состоит из JMP, JSET/JCLR, JMPP; call family - CALL, CALLP, RET. Условия JZ/JNZ/JC/JNC/. . . являются assembler aliases над PSW bits, а не отдельными opcodes.

Stack family содержит object push/pop, SP-relative и FP-relative load/store, signed ADJSP, canonical frame operations PUSHFP/SETFP/RESTSP/POPFP, LEAFP и validated SETSP. Эти instructions работают с raw STACK offsets/PTR64 и не вводят скрытого frame metadata в CPU.

34.6. Явное управление arithmetic/status flags

PSW разделён на два writable byte domains: CPU flags PSW[7:0] и MPU flags PSW[15:8]. Инструкции SETFflag и CLRFFflag изменяют ровно один именованный bit из диапазона AF.INXF и сохраняют все остальные bits PSW. CLRCPUF очищает только PSW[7:0], а CLRMPUF — только PSW[15:8].

HF, TF, IF и многобитовое поле MODE не принадлежат writable arithmetic/status domain. SETF/CLRF обязаны отвергать selectors HF/TF/IF; bulk-clear instructions также не изменяют PSW bits 16 и выше. JSET/JCLR при этом сохраняют более широкий read-only selector contract и могут проверять HF/TF/IF.

Явная установка CF предназначена, в частности, для подготовки входного carry/borrow state перед IADC/ISBB и rotate-through-carry family. Сами integer instructions сохраняют установленную LibMPCPU semantics; новые flag instructions только дают assembler-программе прямой способ подготовить их входное состояние.

34.7. PSW-neutral conversion class

Conversion-class instructions сохраняют incoming PSW[15:0], даже если внутренний LibMPCPU helper меняет library flags. К этой категории относятся непосредственные numeric loads и width/domain conversions, в частности LDI, LDR, LDC, ICVT, RCVT, ITOR, UITOR, RT0I, RT0UI, CCVT, CRT0C. Arithmetic instructions, напротив, публикуют обычные LibMPCPU status effects.

Это различие важно для compiler backend: преобразование representation не должно разрушать condition state предыдущего сравнения, тогда как arithmetic operation по определению может менять flags.

34.8. Неинструкционные функции

Text parsing/formatting, format introspection, diagnostics и library context management не являются CPU operations. Они принадлежат assembler, LibMPCPU, будущей guest library либо host runtime. Такая граница сохраняет ISA вычислительной и не привязывает bytecode к host textual API.

34.9. Regression contract

Нормативное поведение instruction family проверяется не только unit tests decoder-a. Regression suite сравнивает CPU execution с соответствующими LibMPCPU primitives, проверяет malformed widths/operands, exact-immediate diagnostics, PSW preservation для conversions, pointer/stack faults и no-partial-write invariants. Отдельный пакет model examples дополнительно проверяет композицию ISA + CALL ABI + MELF + loader на длинном execution path.

35. Ассемблер: UTF-8 source, строки, .equ и arithmetic expressions

Низкоуровневый mcpu-as обязан уметь описывать binary images data-like sections, labels и relocation-like address expressions без обязательного текстового препроцессора. Текстовая метапрограмма (#define, #include, conditional compilation, parameterized macros) остаётся отдельным будущим слоем. В assembler входят только механизмы, непосредственно связанные с machine/data representation и разрешением символов.

35.1. Кодировка assembler source

Исходный файл `mcru-as` всегда UTF-8. Результат сборки не зависит от `LANG`, `LC_CTYPE` или другой locale host-системы. Перед разбором assembler проверяет source как well-formed UTF-8 через LibMPUIO.

Это особенно важно для строковых директив: одинаковый `.s` обязан создавать одинаковые image bytes на любых build hosts при одинаковом LibMCPUs target profile.

35.2. Командная строка mcru-as

Ассемблер всегда создаёт один перемещаемый объект MELF ET_REL, поэтому отдельный переключатель режима сборки не нужен. Выходной файл задаётся обычной UNIX-опцией `-o`, а единственный исходный файл является позиционным аргументом:

```
mcru-as -o file.o file.s
mcru-as file.s -o file.o
```

Обе формы полностью эквивалентны: порядок `-o` и входного файла свободный. Ровно один входной файл и ровно один выходной файл обязательны. Опции `-h` и `--help` печатают справку. Историческая форма `mcru-as -s input.s -o output.o` удалена: `-s` больше не является опцией, поскольку другого режима работы у assembler нет.

35.3. .utf8 - NUL-terminated UTF-8 byte string

```
.data
```

```
name_utf8:
```

```
.utf8 Андрей""
```

`.utf8` записывает UTF-8 payload буквально как bytes и добавляет один завершающий `0x00`. Никакого target endian преобразования и никакой locale conversion нет.

"Андрей" -> D0 90 D0 BD D0 B4 D1 80 D0 B5 D0 B9 00

Поддерживаются escapes `\\`, `\"`, `\n`, `\r`, `\t`, `\b`, `\f`, `\v`, `\a`, byte escape `\xHH`, а также Unicode scalar escapes `\uXXXX` и `\UXXXXXXXX`. Результирующая последовательность снова проверяется как UTF-8.

Встроенный NUL внутри `.utf8/.ucs2` намеренно не поддерживается: эти директивы определяют C-style NUL-terminated strings. Для произвольного binary layout с внутренними нулями используется `.byte` или `.integer/.int`.

35.4. .ucs2 - UTF-8 source literal в strict UCS-2 target objects

```
.data
```

```
name_ucs2:
```

```
.ucs2 Андрей""
```

Assembler сначала получает UTF-8 literal, затем использует strict LibMPUIO UTF-8 -> UCS-2 conversion. Каждый UCS-2 code unit записывается как отдельный 16-bit architectural data object с помощью target data byte order; в конце добавляется `U+0000`.

U+0410 U+043D U+0434 U+0440 U+0435 U+0439 U+0000

current little-endian target bytes:

```
10 04 3D 04 34 04 40 04 35 04 39 04      00 00
```

Strict UCS-2 допускает scalar values `U+0000..U+FFFF`, кроме surrogate range `U+D800..U+DFFF`. UTF-8 characters выше `U+FFFF` (например emoji) не раскладываются на UTF-16 surrogate pairs, а дают assembler error. Таким образом `.ucs2` точно соответствует принятой в LibMCPUs/LibMPUIO модели UCS-2, а не UTF-16.

35.5. .byte, .integer и .int

`.byte` создаёт raw bytes без implicit terminator, alignment или endian conversion:

```
.byte 0x4c, 0x43, 0x50, 0x55
.byte 1, 2+3, BLOCK_SIZE/512
```

Каждый элемент должен находиться в диапазоне `0..255`. `.integer` сохраняет прежний смысл typed integer object, а `.int` является полным синонимом. Обе директивы принимают список значений одной ширины:

```
.integer 16, 0x0410, 0x043d, 0x0000
.int      8, 65, 66, 0
```

Каждое значение формирует отдельный object указанной width непосредственно в текущей data-like section position; hidden padding отсутствует.

35.6. .equ - абсолютные assembler symbols

```
.equ table_ptr, 0x03000
.equ block_size, 4 * 512
.equ next_reg, table_ptr + 8
```

.equ section-independent и не создаёт bytes. Имена symbols case-sensitive. Все .equ definitions собираются до emission, поэтому разрешены forward references между .equ symbols.

.equ является absolute constant symbol. В текущем low-level assembler выражение

справа может ссылаться на integer literals и другие .equ, но не на DATA/CODE labels. Это сохраняет различие между absolute assembler symbols и section-relative labels/relocations MELF.

Имя label не может совпадать с именем .equ.

35.7. Integer expression grammar и precedence

primary := integer | symbol | '(' expression ')'

unary := ['+' | '-'] unary | primary

product := unary { ('*' | '/' | '%') unary } expression := product { ('+' | '-') product }

Операции имеют обычный arithmetic precedence: unary, затем multiply/divide/remainder, затем add/subtract. Division by zero, overflow internal expression domain и значение вне требуемой width дают assembler error.

Expressions используются для absolute numeric operands: register byte address, LDI, stack/pointer signed offsets, BMOVI count, .align, .byte, integer widths/values и других соответствующих immediate fields.

35.8. Label expressions и fixups

Address expressions дополнительно получают labels тех sections, которые допустимы для конкретного operand class. Поэтому assembler непосредственно поддерживает:

```
LEA    r64@table_ptr, input_vector+8    ; selector follows label section
LOAD   r64@next_reg, data_object+16     ; direct LOAD remains DATA-relative
JMP     loop+0                          ; PROGRAM CODE64
CALL    function
.cptr   handler+0
```

LEA принимает labels из DATA/CONTROL/INPUT/OUTPUT/TRACE и code sections и создаёт полный PTR64 selector+offset. Direct LOAD/STORE остаются в namespace DATA labels + .equ. CODE64 operands и .cptr используют PROGRAM labels + .equ. Forward labels работают через fixups. CPU по-прежнему не хранит instruction-boundary metadata: arithmetic CODE expression обязано использоваться программистом осмысленно.

35.9. Symbolic register addresses

REG24 byte address может быть integer expression из absolute .equ symbols:

```
.equ table_ptr, 0x03000
.equ temp_ptr, table_ptr + 8

LEA    r64@table_ptr, table
LDI     r64@temp_ptr, 42
LDI     r64@(table_ptr + 16), 7
```

После вычисления expression assembler выполняет обычную проверку register width, natural alignment и границ 512 KiB register file. DATA/CODE labels в register-address expression не участвуют: label address и register-file byte address являются разными namespaces.

35.10. Граница mcpu-as и будущего препроцессора

Механизм	Уровень	Причина
.init/.text/.fini/.data/.control/.input/.output/.trace, labels, .align	mcpu-as	определяют layout executable/data images
.integer/.int/.real/.complex/.byte/.utf8/.ucs2/.cptr	mcpu-as	создают data-like objects/bytes

<code>.equ, arithmetic expressions, symbol fixups</code>	<code>mcru-as</code>	необходимы для адресов и будущих relocations
<code>#define/#undef/#include/#if/#ifdef</code>	отдельный preprocessor	текстовая metaprogramming не является machine encoding
parameterized macros / repetition	отдельный preprocessor	source transformation до mcru-as

Таким образом обычный symbolic assembler остаётся самодостаточным, а будущая схема `.S -> cru-cpp -> mcru-as` может добавляться независимо и не усложняет binary semantics cru-as.

35.11. Полный пример

<pre>.equ table_ptr, 0x03000 .equ next_reg, table_ptr + 8 .equ block_size, 4 * 512 .data name16: .ucs2 Андрей"" name8: .utf8 Андрей"" header: .byte 0x4c, 0x43, 0x50, 0x55, block_size/512 letters: .int 16, 0x0410, 0x043d, 0x0434, 0 .align 16 answer: .integer 64, 42 .text start: LEA r64@table_ptr, name16 LOAD r64@next_reg, answer HALT</pre>

Здесь ни один identifier не требует textual macro substitution: `table_ptr/next_reg` - absolute assembler symbols, `name16/answer` - DATA labels, а arithmetic expression вычисляется самим assembler.

35.12. Маркеры строк препроцессора и логические координаты diagnostics

После появления `mcru - cpp assembler` не использует собственную директиву `#line` для управления source coordinates. Вместо неё `mcru - as` распознаёт маркеры строк, оставляемые препроцессором в выходном потоке:

<pre># digit-sequence # digit-sequence "filename" # digit-sequence "filename" flags...</pre>
--

`digit-sequence` содержит как минимум одну decimal digit и всегда разбирается в основании 10, даже при leading zeroes. Допустимый диапазон - 0..2147483647. Значение задаёт logical line number для physical source line, непосредственно следующей за marker. Filename optional; при его отсутствии сохраняется предыдущий logical filename. Всё после закрывающей кавычки filename считается полем flags препроцессора и полностью игнорируется assembler.

Marker не создаёт bytes, symbols, `__LINE__` или `__FILE__`, не участвует в expression evaluation и влияет только на source coordinates последующих diagnostics. Empty и comment-only ordinary lines увеличивают logical counter обычным образом.

Строки `#pragma` и `#lang` предназначены для других участников toolchain: `mcru - as` полностью игнорирует их и не увеличивает logical line counter. Любая другая строка, начинающаяся с `#`, также удаляется из assembler token stream, но учитывается как одна

ordinary logical source line.

Историческая форма #line. Запись `#lineN"file"` больше не является source-location directive `mcpu-as`. Она обрабатывается как обычная неизвестная строка `#...`: игнорируется как assembler syntax и учитывается при продвижении logical counter. Координаты source задаются только preprocessing line marker формой `#N"file"flags...`. Это diagnostics contract и не часть MELF ABI.

Часть III

Program Model ABI, runtime и динамическая загрузка

36. Program Model Lifecycle ABI

Каждый исполняемый MELF-файл имеет обязательные lifecycle symbols `_start` и `_life`; `e_entry` всегда указывает на `_start`. Секция `.init` и символ `_fini` необязательны. Эти точки входа принадлежат MELF/runtime ABI и не должны смешиваться с функциями будущего языка высокого уровня.

36.1. С отдельным INIT

```
.init
_start:
    ... one-time startup ...
    END

.text
_life:
    ... resumable LIFE phase ...
    END
```

Тогда `e_init_entry=_start`, а `e_life_entry=_life`. Первый `END` завершает `INIT`; runtime устанавливает PC в `_life` и начинает `LIFE`. Последующие `LIFE END` возвращают управление supervisor-у, а следующий resume продолжается с инструкции после соответствующего `END`.

36.2. Без INIT

```
.text
_start_life:
    ... startup / ordinary program trampoline ...
```

Псевдометка `_start_life`: эквивалентна двум последовательным labels `_start`: и `_life`: по одному адресу. В final executable `e_init_entry=0` и `e_entry==e_life_entry`. Эта форма особенно естественна для обычной PROGRAM, которая заканчивается через `EXIT` и не создаёт `LIFE boundaries`.

36.3. FINI, END и EXIT

`END` завершает текущую управляемую фазу и возвращает управление supervisor-у. `Graceful EXIT(status)` завершает normal execution; если имеется `.fini/_fini`, runtime сохраняет status, входит в `FINI` и требует завершить `FINI` инструкцией `END`. Второй `EXIT` внутри `FINI` не является нормальным контрактом. Diagnostic stop по `- - steps` не симулирует graceful `EXIT` и потому сам по себе `FINI` не запускает.

37. HLL Runtime ABI: PROGRAM, MODEL, CRT и MCPUs Libc

Этот раздел фиксирует подписанный HLL Runtime ABI. Он описывает границу между объектами, которые в будущем генерирует `mcru - cc`, существующим `mcru - ld`, CRT, MCPUs Libc и уже действующим MELF lifecycle. Контракт не зависит от номера выпуска LibMCPUs.

Разделение уровней. HLL-функции `main()`, `init()`, `life()` и `fini()` являются обычными функциями CALL ABI и завершаются `RET`. Машинные точки `_start`, `_life`, `_fini`, а также lifecycle-инструкции `END/EXIT` принадлежат CRT и MELF runtime. Пользовательская HLL-функция не должна исполнять raw `END` или `EXIT` из открытого CALL frame.

37.1. Целевая integer-модель HLL ABI

HLL ABI использует фиксированные целевые типы `int8/int16/int32/int64`, соответствующие unsigned-типы и `PTR64`. Исторические C-имена `char/short/int/long` не определяют бинарный ABI. `PTR64` — обозначение бинарного pointer object, а не синоним `int64*`.

Канонические compiler-visible пределы публикуются препроцессором целевой среды. Для жизненного цикла важны определения:

```
#define LIFE_END      0
#define LIFE_CONTINUE (-__INT32_MAX__ - 1)
```

Таким образом минимальное значение signed int32 выражается непосредственно через target predefined macro `__INT32_MAX__`.

37.2. Две формы законченной HLL-программы

Исходный compilation unit определяет ровно одну из двух форм:

Форма	Функции HLL ABI	Назначение
PROGRAM	<code>int32main(int64argc, PTR64argv);</code> <code>optional source-level voidfini(void)</code>	однократная обычная программа
MODEL	<code>int32init(int64argc, PTR64argv);</code> <code>int32life(void); optional source-level voidfini(void)</code>	модель с INIT и повторяющимися LIFE-фазами

Frontend нормализует optional source-level `fini`: если программист её не определил, compiler создаёт пустую strong GLOBAL/FUNC `fini`. Поэтому defining ET_REL всегда содержит:

- PROGRAM: strong GLOBAL/FUNC `main` и `fini`;
- MODEL: strong GLOBAL/FUNC `init`, `life` и `fini`.

Все complete-program definitions одной формы принадлежат **одному** defining ET_REL. Другие translation units могут их объявлять, но не определять. `main` несовместим с парой `init/life`; неполная или разделённая между несколькими ET_REL форма является ошибкой final link.

37.3. Автоматическая классификация в mcpu-ld

Для final ET_EXEC mcpu-ld классифицирует обычные входные ET_REL до разрешения библиотек. Никаких ключей - - program или - - model нет:

Определения defining ET_REL	Режим
<code>main+fini</code>	HLL PROGRAM
<code>init+life+fini</code>	HLL MODEL
<code>_start/_life[_fini]</code> без HLL roots	low-level MELF executable; automatic CRT не добавляется

Смешение raw lifecycle definitions с HLL complete-program roots запрещено. Символы из `-l...` не используются для классификации и потому библиотека не может случайно превратить link в PROGRAM или MODEL.

После классификации linker автоматически вставляет CRT из активного MCPU system root:

```
PROGRAM: <root>/lib/mcpu/crt0.o crt1.o <user objects/libraries> crtn.o
MODEL:   <root>/lib/mcpu/crt1.o crt1.o <user objects/libraries> crtn.o
```

Отсутствие любого обязательного CRT object является fatal link error. Переход к другому root или деградация в low-level link не выполняются.

37.4. Распределение ответственности CRT

Канонические исходники принадлежат LibMCPU и устанавливаются как target runtime objects:

<code>crt0.o</code>	PROGRAM adapter; формирует <code>_start==_life</code> в <code>.text</code> и вызывает <code>main</code> .
<code>crt1.o</code>	MODEL adapter; формирует <code>_start</code> в <code>.init</code> и <code>_life</code> в <code>.text</code> , вызывает <code>init/life</code> .
<code>crti.o</code>	общая startup state/validation и вызовы начала MCPU Libc.
<code>crtn.o</code>	обязательная HLL <code>_fini</code> , PRE_RUNTIME/RUNNING branch и graceful finalization.

HLL ET_EXEC всегда получает `_fini`, даже если source-level `fini()` была опущена и нормализована frontend-ом в пустую функцию.

37.5. McpuStartup, argc и argv

Loader строит McpuStartup один раз до первой guest instruction. Заголовок, массив `argv[argc+1]` и строки аргументов остаются действительными до конца процесса. CRT проверяет version/size/representation, затем проверяет, что `argc` не превышает `__INT64_MAX__`, и преобразует unsigned startup field в target signed `int64`. Существующий `PTR64argv` переиспользуется; MCPU Libc не копирует и не владеет `argc/argv`.

Для `main()` и `init()` CRT создаёт обычный 16-байтовый CALL argument block:

До CALL	Объект	После push return PC callee видит
SP+0	int64 argc	SP+8
SP+8	PTR64 argv	SP+16

Возврат int32 из main/init/life находится в канонической Return Value Area r32@0x7fffc.

37.6. Внутренние hooks MCPU Libc

CRT использует ровно четыре reserved internal hooks:

```
void __mcpu_libc_init(void);
void __mcpu_init_array(void);
void __mcpu_fini_array(void);
void __mcpu_libc_fini(void);
```

Это обычные no-argument CALL ABI functions с RET. Startup order после успешной validation:

```
__mcpu_libc_init()
__mcpu_init_array()
main(...) или init(...)
```

Graceful HLL finalization в состоянии RUNNING:

```
fini()
__mcpu_fini_array()
__mcpu_libc_fini()
END
```

Physical ABI init/fini arrays и constructor priorities могут развиваться отдельно; наличие hooks уже фиксирует границу CRT и Libc.

Обычная ABI-v3 DT_NEEDED библиотека MCPU Libc имеет право предоставить эти четыре symbols. Explicit preload не имеет права их подменять. Complete-program roots main/init/life/fini, lifecycle symbols и CRT-private names также не являются пользовательским interposition API.

37.7. PRE_RUNTIME и RUNNING

CRT хранит private logical state PRE_RUNTIME / RUNNING; его физическое представление не является public ABI. До успешного завершения __mcpu_libc_init() и __mcpu_init_array() состояние остаётся PRE_RUNTIME.

Если validation startup area или диапазона argc завершается ошибкой, CRT выполняет graceful EXIT(startup_failure_status). Supervisor входит в _fini, но PRE_RUNTIME branch выполняет только END: пользовательская fini, fini-array и libc-fini не вызываются. Namespace startup-failure status не выводится из host errno.

После startup hooks CRT устанавливает RUNNING. С этого момента normal graceful EXIT всегда приводит к полному HLL finalization. Поэтому ненулевой возврат init() уже находится в RUNNING и вызывает пользовательскую fini(); MODEL fini должна допускать частично инициализированную модель.

37.8. PROGRAM lifecycle

PROGRAM не имеет отдельной .init. _start==_life находится в .text. После validation и startup hooks CRT вызывает

```
int32 main(int64 argc, PTR64 argv);
```

и после RET выполняет EXIT(main_status). Дополнительного LIFE END после main() нет.

37.9. MODEL lifecycle

MODEL использует _start в .init и _life в .text. В INIT CRT вызывает

```
int32 init(int64 argc, PTR64 argv);
```

Возврат 0 создаёт ровно одну INIT boundary через END. Ненулевой status выполняет EXIT(status) и переходит к полной HLL finalization, поскольку runtime уже RUNNING.

Каждая LIFE-фаза является отдельным обычным вызовом:

```
int32 life(void);
```

Return namespace фиксирован:

int32 return	Имя	Действие CRT
- __INT32_MAX__ - 1	LIFE_CONTINUE	выполнить ровно один LIFE END; после resume снова вызвать life()
0	LIFE_END	EXIT(0) без дополнительного END
1..__INT32_MAX__	error status	EXIT(status) без дополнительного END
- __INT32_MAX__..-1	reserved	не использовать без будущего расширения ABI

Это гарантирует, что естественное завершение модели не создаёт фиктивной последней LIFE boundary и, например, лишней строки TRACE.

37.10. HLL finalization и сохранение status

В RUNNING каноническая `_fini` концептуально выполняет:

```
CALL fini
CALL __mcpu_fini_array
CALL __mcpu_libc_fini
END
```

Original EXIT status сохраняется runtime через всю finalization sequence. Второй EXIT из `_fini` не выполняется. Диагностическая остановка `mcpu-run - - steps=...` не является graceful exit и не синтезирует HLL finalization.

Главный итог. HLL-программа определяет только обычные functions. `mcpu-ld` распознаёт нормализованную форму ET-REL и выбирает CRT автоматически. CRT владеет MELF lifecycle, MCPU Libc владеет библиотечным runtime, а loader владеет persistent startup area. Ни один слой не дублирует ответственность соседнего.

38. McpuStartup и argc/argv

Перед первой guest-инструкцией загрузчик создаёт постоянную область запуска (startup area) в верхней части STACK. Заголовок имеет размер 32 байта и выравнивание 16 байт:

Смещение	Поле	Размер	Значение
0x00	version	4	MCPU_STARTUP_VERSION_1
0x04	size	4	32
0x08	argc	8	число аргументов guest-программы
0x10	argv	8	PTR64(STACK, массив argv)
0x18	reserved	8	ноль

После заголовка расположен массив `argv[0..argc]` из PTR64, завершающий NULL, затем NUL-terminated строки аргументов. Эта область остаётся действительной всё время жизни executable. Начальное состояние: `PC=e_entry`, `SP=startup_base`, `FP=stack_size`.

39. HEAP и program break

HEAP — отдельный runtime-сегмент `SEG=0x7`. Он не расширяет DATA и не имеет assembler payload. Определяемый компоновщиком неподменяемый символ `_end` равен `PTR64(HEAP,16)`; это начальный и минимальный program break.

Runtime хранит `_heap_brk`, `_heap_size`, `_heap_capacity`, `immutable_heap_limit`. Запрос ниже 16 или выше limit отклоняется и возвращает прежний break. Shrink не обязан уменьшать host mapping/capacity; освобождённый range немедленно zeroes. Regrow в уже существующий backing снова открывает deterministic zero memory. При backing relocation guest pointers остаются valid, потому что содержат `segment+offset`.

40. SYSCALL ABI и SYS_brk

Zero-operand instruction SYSCALL использует fixed overlapping register area в верхней части register file.

Role	Address/view	Meaning
SYSNO	r32@0x7fff8	syscall number
NARGS	r32@0x7fff8	argument count
RET	r64@0x7fff8	return value; overlaps NARGS
ARG0..ARG5	r64@0x7fff0 ... r64@0x7ffc8	up to six arguments

Первый syscall — SYS_brk, number 10. Он требует один argument: NULL означает query, HEAP PTR64 requests absolute new break. Общее правило syscall ABI использует стабильные host-independent MCPU errno: неуспех возвращает отрицательное -MCPU_errno, а host errno преобразуется на sysdeps boundary и наружу не выходит. SYS_brk является специальным исключением: он возвращает фактический/current HEAP PTR64 и при accepted, и при rejected request. Unsupported syscall number возвращает -ENOSYS из MCPU errno namespace.

```
; query current break
MOV    r32@0x7fff8, 1
LDI    r32@0x7fff8, 10
CLR    r64@0x7fff0
SYSCALL
; result: r64@0x7fff8
```

41. Startup dynamic linking: архитектурный взгляд

Текущий normal dynamic contract — frozen MELF ABI v3. mcpu-ld--dynamic создаёт ABI-v3 ET_EXEC, а mcpu-ld--shared--soname=NAME — ABI-v3 ET_DYN. В отличие от исторического ABI v2, normal dependencies записываются в canonical non-ALLOC .dynamic как DT_NEEDED, а shared object имеет обязательный DT_SONAME. ABI v2 остаётся читаемым как historical explicit-preload input format.

mcpu-run обнаруживает NEEDED graph до исполнения guest-кода, проверяет SONAME identity, загружает dependencies breadth-first/left-to-right и завершает symbol resolution/RELA до первой инструкции _start. Явный --ldpreload и MCPU_LD_PRELOAD остаются override/interposition mechanisms. PLT/GOT, lazy binding, dlopen(), dlsym() и runtime lookup после старта отсутствуют.

ET_DYN содержит только загружаемое PROGRAM/DATA содержимое .text/.data/.bss плюс non-ALLOC metadata; lifecycle, CONTROL/INPUT/OUTPUT/TRACE, STACK и HEAP принадлежат главному executable.

41.1. GLOBAL/WEAK interposition

Сильный GLOBAL имеет приоритет над WEAK. Для equal-strength definitions highest scope имеет explicit preload с установленным rightmost priority, затем main executable, затем normal NEEDED graph в breadth-first/left-to-right order. Lifecycle symbols _start/_life/_fini остаются частными для executable и не участвуют в interposition.

41.2. LOCAL ET_DYN runtime rebasing

LOCAL symbol остаётся private и non-preemptible, но ссылка на него внутри ET_DYN **не может быть окончательно превращена в process address при static link**: final PROGRAM/DATA base shared object ещё неизвестна. Поэтому такая ссылка сохраняется как LOCAL runtime RELA. Startup linker разрешает её строго против originating module и добавляет runtime segment base. LOCAL symbols не попадают в GLOBAL/WEAK winner namespace.

Правило возникло как исправление runtime-address contract ещё в ABI v2 и является нормативной частью frozen ABI v3. Оно не открывает LOCAL symbol для interposition.

42. mcpu-run как host-side test bench

Помимо one-shot inspection options, runtime поддерживает четыре lifecycle stages:

Опция	Граница выполнения
--before-run= <action>	after load/preload/relocations/startup, before first guest instruction
--after-init= <action>	after INIT END; при отсутствии .init — один раз на logical empty INIT boundary
--on-step=<action>	after every LIFE END
--after-run= <action>	after final normal stop; graceful FINI уже завершён

Stage-опции можно повторять; они выполняются слева направо. Фильтры по номеру шага намеренно отсутствуют.

42.1. Typed actions

```
print-int-regs:r64@0x100,r64@0x108
print-real-regs:r128@0x200,r256@0x300
set-int-regs:r64@0x100=123,r64@0x108=-17
set-real-regs:r128@0x200=1.25,r128@0x220=-2.5

print-int-<memory>:64@16,64@24
print-real-<memory>:128@32,128@48
set-int-<memory>:64@16=123,64@24=456
set-real-<memory>:128@32=1.25,128@48=2.5
```

<memory> — data, control, heap, input, output, trace. Один scalar print-action печатает comma-list одной строкой через spaces. Set values разбираются через LibMPIO `mpu_sscanf()`; whole action валидируется до первой записи, а guest PSW сохраняется вокруг host-side numeric parsing.

42.2. Raw actions

`print-raw-<memory>:<bytes>@<addr>` read-only и описывает ровно один dump. Comma-list запрещён; для нескольких dumps повторяется stage option. `set-raw-*` не существует. Size принимает raw bytes и K/KiB, M/MiB, G/GiB с binary multipliers. DATA raw view ограничен linked .data и не перелезает в .bss; HEAP host view ограничен prepared backing, тогда как guest access ограничен current break.

42.3. Двойной дефис как разделитель

Preferred CLI:

```
mcru-run [runtime-options] -- executable [program-args...]
```

После separator `--` tokens не интерпретируются как runner options. Первый token становится executable spelling и guest `argv[0]`. Legacy form без separator сохранена.

43. Бинарные утилиты toolchain

Системные инструменты MELF используют единый public format contract и не вводят альтернативного представления машинного кода.

Команда	Назначение
<code>mcru - readelf</code>	Структурный read-only просмотр MELF headers, sections, symbols, RELA, dynamic и MCPU metadata.
<code>mcru - ar</code>	Deterministic UNIX static archives из MELF ET_REL с System-V/GNU-style symbol index.
<code>mcru - nm</code>	Symbol-table inspection для объектов, executables, shared objects и archives.
<code>mcru - objdump</code>	Section/relocation inspection и дизассемблирование через тот же decoder, который используется CPU runtime.
<code>mcru - strip</code>	Консервативное удаление только non-runtime symbol metadata; link/runtime-critical sections сохраняются.
<code>mcru - objcopy</code>	MELF-to-MELF structural copy и safe editing non-ALLOC metadata; runtime-visible image защищён.

MELF уже является native object/executable container виртуальной MCPU. Поэтому `mcru - objcopy` не имеет режимов flat binary, Intel HEX, S-record, VMA/LMA или подготовки firmware image для SPI NOR. Metadata-only edit обязан сохранять guest-visible allocated image и frozen ABI v3 execution semantics.

Английские и русские manual pages устанавливаются как section 1 для каждой команды; обзор архитектуры и toolchain доступен как `libmcru(7)`.

44. Инварианты реализации

1. PROGRAM machine fields всегда MSB-first; data representation не смешивается с code representation.
2. Register address и PTR64 offset не являются host pointers.
3. Bounds/permission validation precedes guest-visible writes, где instruction contract требует atomicity.
4. DATA имеет один process-owned 16-byte NULL guard; ET_DYN не получает собственного guard.
5. HEAP имеет immutable configured limit и minimum break 16; released bytes zeroed on shrink.
6. ABI-v1 static executable полностью linked; ordinary loader его не relink-ит.

7. ABI-v2/v3 startup linking завершается до `_start`; после старта symbol lookup отсутствует.
8. LOCAL ET_DYN references runtime-rebased, но никогда не interpose.
9. Lifecycle boundaries определяются guest END/EXIT; stage actions лишь наблюдают/модифицируют state на host side.

1. Полный справочник директив `mcru-as`

Этот appendix перечисляет assembler directives и специальные формы, которые являются частью текущего source contract. Детальная грамматика выражений и ограничения sections приведены в основной части документа.

Директива	Форма	Назначение
<code>.init</code>	без аргументов	Переключает текущую code section на INIT. Для final low-level ET_EXEC здесь определяется <code>_start</code> ; ET_DYN такую секцию иметь не может.
<code>.text</code>	без аргументов	Основная PROGRAM code section. Здесь находится <code>_life</code> , а PROGRAM без отдельного INIT имеет <code>_start==_life</code> .
<code>.fini</code>	без аргументов	FINI code section; low-level entry symbol — <code>_fini</code> . HLL finalization создаётся CRT автоматически.
<code>.data</code>	без аргументов	Инициализированные writable DATA objects.
<code>.bss</code>	без аргументов	Zero-initialized DATA, сериализуется как NOBITS.
<code>.control</code>	без аргументов	Model CONTROL space contribution; guest read-only во время исполнения. Только main ET_EXEC.
<code>.input</code>	без аргументов	Model INPUT space contribution. Только main ET_EXEC.
<code>.output</code>	без аргументов	Model OUTPUT space contribution. Только main ET_EXEC.
<code>.trace</code>	без аргументов	Model TRACE space contribution. Только main ET_EXEC.
<code>.align</code>	<code>.alignN</code>	Выравнивает следующую позицию текущей section до степени/границы, проверяемой assembler-ом; добавляет padding либо увеличивает BSS size.
<code>.zero</code>	<code>.zeroN</code>	Резервирует N нулевых байтов; особенно естественна в <code>.bss</code> .
<code>.byte</code>	<code>.byteexpr[, ...]</code>	Записывает raw bytes 0..255 без terminator, alignment и endian conversion.
<code>.integer, .int</code>	<code>.intbits, expr[, ...]</code>	Создаёт целые target objects заданной ширины. <code>.int</code> — полный синоним <code>.integer</code> .
<code>.real</code>	<code>.realbits, value</code>	Создаёт target Real object указанной physical width через LibMPCPU representation.
<code>.complex</code>	<code>.complexbits, real, imag</code>	Создаёт target Complex object; bits задаёт physical width всего Complex object.
<code>.utf8</code>	<code>.utf8"text"</code>	Записывает проверенную UTF-8 NUL-terminated byte string. Embedded NUL не допускается.
<code>.ucs2</code>	<code>.ucs2"text"</code>	Преобразует UTF-8 source literal в strict UCS-2 target code units и добавляет U+0000. Surrogate range и scalar values выше U+FFFF запрещены.
<code>.cptr, .codeptr</code>	<code>.cptrcode-expr</code>	Создаёт PROGRAM/CODE pointer object с relocation при необходимости. <code>.codeptr</code> — синоним.
<code>.ptr</code>	<code>.ptrpointer-expr</code>	Создаёт полный PTR64 на допустимый target segment/label с relocation при необходимости.
<code>.equ</code>	<code>.equname, expr</code>	Определяет absolute assembler symbol, не создающий bytes. Допускает forward references между <code>.equ</code> ; section labels в таком выражении не используются.
<code>.global, .globl</code>	<code>.globalname</code>	Делает symbol GLOBAL. <code>.globl</code> — синоним.
<code>.extern</code>	<code>.externname</code>	Объявляет внешний undefined symbol для последующей компоновки.
<code>.weak</code>	<code>.weakname</code>	Объявляет/помечает symbol как WEAK согласно MELF symbol binding.

Особые синтаксические формы. `_start_life`: — assembler pseudo-label, эквивалентный последовательным `_start:` и `_life:` по одному адресу в `.text`. Он не является директивой и не создаёт отдельный MELF symbol `_start_life`. Preprocessor line markers `#N"file"flags...` меняют только logical coordinates diagnostics; `#pragma` и `#lang` игнорируются assembler-ом как строки, принадлежащие другим стадиям toolchain.

2. Полная таблица OPCODE16

Таблица перечисляет все значения `public cpu_opcode`. Каждый opcode однозначно задаёт число и тип последующих operands; PROGRAM machine fields сериализуются MSB-first. Operand spellings показаны в терминах assembler ABI.

Mnemonic	OPCODE16	Operands	Назначение
NOP	0x0000	- - -	Нет операции; PC переходит к следующей инструкции.
HALT	0x0001	- - -	Устанавливает machine halt state; используется как низкоуровневая остановка CPU.
END	0x0002	- - -	Завершает текущую externally managed INIT/LIFE/FINI phase и возвращает управление supervisor-у.
EXIT	0x0003	r32status	Graceful завершение программы/модели с signed int32 status; при наличии FINI runtime переходит к <code>_fini</code> .
SYSCALL	0x0004	- - -	Выполняет системный вызов через фиксированную register ABI area.
SETF	0x0010	PSWBIT8	Устанавливает ровно один writable arithmetic/status flag в диапазоне PSW[15:0]; HF/TF/IF недопустимы.
CLRF	0x0011	PSWBIT8	Очищает ровно один writable arithmetic/status flag в диапазоне PSW[15:0]; HF/TF/IF недопустимы.
CLRCPUF	0x0012	- - -	Очищает CPU flag byte PSW[7:0], сохраняя PSW[15:8] и control/execution state.
CLRMPUF	0x0013	- - -	Очищает MPU flag byte PSW[15:8], сохраняя PSW[7:0] и control/execution state.
MOV	0x0100	Rdst, Rsrc	Копирует объект между регистрами одинаковой физической ширины.
CLR	0x0101	Rdst	Заполняет выбранный регистр нулями.
SWAP	0x0102	Ra, Rb	Обменивает содержимое двух регистров одинаковой ширины.
BSWAP	0x0103	Rdst, Rsrc	Разворачивает порядок байтов source object; raw byte reversal без числовой интерпретации.
LDI	0x0110	Rdst, imm64	Загружает signed integer immediate с приведением к ширине integer register; conversion PSW-neutral.
LDR	0x0111	Rdst, realliteral	Загружает exactly representable immediate Real32/Real64; более широкие constants следует хранить в DATA.
LDC	0x0112	Cdst, complexliteral	Загружает immediate Complex64 как две точно представимые Real32 components.
LOAD	0x0120	Rdst, DATA64	Копирует объект из DATA по прямому DATA-relative адресу в регистр.
STORE	0x0121	DATA64, Rsrc	Копирует регистр в DATA по прямому DATA-relative адресу.
LEA	0x0130	r64dst, PTR64	Формирует полный segment-aware PTR64 на label/target object.
LEASP	0x0131	r64dst, OFF64	Формирует STACK PTR64 как SP плюс signed byte offset.
LOADP	0x0132	R/Cdst, r64ptr	Читает объект через PTR64 после проверки segment rights и bounds.
STOREP	0x0133	r64ptr, R/Csrc	Записывает объект через PTR64 после проверки segment rights и bounds.
PADD	0x0140	r64dst, r64ptr, r64delta	Прибавляет signed byte delta из r64 к PTR64 с сохранением segment selector.
PADDI	0x0141	r64dst, r64ptr, OFF64	Прибавляет signed immediate byte offset к PTR64.
PSUB	0x0142	r64dst, r64ptr, r64delta	Вычитает signed byte delta из PTR64.
PSUBI	0x0143	r64dst, r64ptr, OFF64	Вычитает signed immediate byte offset из PTR64.
PDIFF	0x0144	r64dst, r64a, r64b	Вычисляет signed разность byte offsets двух совместимых PTR64.
BMOV	0x0150	r64dst, r64src, r64count	Перемещает count байтов между PTR64 ranges с memmove-семантикой и полной предварительной проверкой.
BMOVI	0x0151	r64dst, r64src, IMM64count	То же, но byte count задан immediate.
IADD	0x1000	Idst, Ia, Ib	Беззнаковое/битовое integer сложение выбранной ширины с публикацией integer flags.

Mnemonic	OPCODE16	Operands	Назначение
IADC	0x1001	Idst, Ia, Ib	Integer сложение с входным carry CF.
ISUB	0x1002	Idst, Ia, Ib	Integer вычитание выбранной ширины.
ISBB	0x1003	Idst, Ia, Ib	Integer вычитание с borrow/carry state.
IAND	0x1004	Idst, Ia, Ib	Побитовое AND.
IOR	0x1005	Idst, Ia, Ib	Побитовое OR.
IXOR	0x1006	Idst, Ia, Ib	Побитовое XOR.
INOT	0x1010	Idst, Isrc	Побитовое дополнение integer object.
INEG	0x1011	Idst, Isrc	Двухкомплементарное отрицание integer object.
ICMP	0x1020	Ia, Ib	Сравнивает integer operands одинаковой ширины и обновляет comparison flags без сохранения результата.
ITEST	0x1021	Ia, Ib	Выполняет AND только для формирования integer condition flags.
ISHL	0x1030	Idst, Isrc	Logical shift left на один bit.
ISHR	0x1031	Idst, Isrc	Logical shift right на один bit.
ISAL	0x1032	Idst, Isrc	Arithmetic shift left на один bit.
ISAR	0x1033	Idst, Isrc	Arithmetic shift right на один bit с сохранением знака.
IROL	0x1034	Idst, Isrc	Rotate left на один bit.
IROR	0x1035	Idst, Isrc	Rotate right на один bit.
IRCL	0x1036	Idst, Isrc	Rotate left through carry CF на один bit.
IRCR	0x1037	Idst, Isrc	Rotate right through carry CF на один bit.
ISHLN	0x1040	Idst, Isrc, r64n	Logical shift left на n bits.
ISHRN	0x1041	Idst, Isrc, r64n	Logical shift right на n bits.
ISALN	0x1042	Idst, Isrc, r64n	Arithmetic shift left на n bits.
ISARN	0x1043	Idst, Isrc, r64n	Arithmetic shift right на n bits.
IROLN	0x1044	Idst, Isrc, r64n	Rotate left на n bits.
IRORN	0x1045	Idst, Isrc, r64n	Rotate right на n bits.
IRCLN	0x1046	Idst, Isrc, r64n	Rotate left through carry на n bits.
IRCRN	0x1047	Idst, Isrc, r64n	Rotate right through carry на n bits.
IINC	0x1050	Idst, Isrc	Увеличивает integer value на единицу.
IDEC	0x1051	Idst, Isrc	Уменьшает integer value на единицу.
IXCHG	0x1052	Ia, Ib	Обменивает integer values одинаковой ширины.
ICPY	0x1053	Idst, Isrc	Unsigned width-changing copy: zero-extension/truncation по integer object rules.
ICVT	0x1054	Idst, Isrc	Signed width-changing conversion с sign extension/truncation; conversion PSW-neutral.
IMUL	0x1060	Iprod, Ia, Ib	Полное unsigned integer произведение; destination имеет product width, допустимую ISA.
ISMUL	0x1061	Iprod, Ia, Ib	Полное signed integer произведение.
IDIV	0x1062	Iquot, Irem, Inum, Iden	Unsigned integer division с отдельными quotient и remainder.
ISDIV	0x1063	Iquot, Irem, Inum, Iden	Signed integer division с отдельными quotient и remainder.
IBITSLEN	0x1064	r64dst, Isrc	Возвращает число значащих bits integer source.
RADD	0x2000	Rdst, Ra, Rb	Сложение двух Real одинаковой physical width.
RSUB	0x2001	Rdst, Ra, Rb	Вычитание Real.
RMUL	0x2002	Rdst, Ra, Rb	Умножение Real.
RDIV	0x2003	Rdst, Ra, Rb	Деление Real.
RSQRT	0x2010	Rdst, Rsrc	Квадратный корень Real.
RSIN	0x2020	Rdst, Rsrc	Синус Real.
RCOS	0x2021	Rdst, Rsrc	Косинус Real.
RTAN	0x2022	Rdst, Rsrc	Тангенс Real.
RASIN	0x2023	Rdst, Rsrc	Аркасинус Real.
RACOS	0x2024	Rdst, Rsrc	Аркакосинус Real.
RATAN	0x2025	Rdst, Rsrc	Арктангенс Real.
RLOG	0x2030	Rdst, Rsrc	Натуральный логарифм Real.
RLOG10	0x2031	Rdst, Rsrc	Десятичный логарифм Real.
RLOG2	0x2032	Rdst, Rsrc	Двоичный логарифм Real.
REXP	0x2033	Rdst, Rsrc	Экспонента e^x .
RSINH	0x2040	Rdst, Rsrc	Гиперболический синус.
RCOSH	0x2041	Rdst, Rsrc	Гиперболический косинус.
RTANH	0x2042	Rdst, Rsrc	Гиперболический тангенс.

Mnemonic	OPCODE16	Operands	Назначение
RPOW	0x2050	Rdst, Ra, Rb	Возводит Real a в степень Real b.
RHYPOT	0x2051	Rdst, Ra, Rb	Вычисляет $\text{hypot}(a,b)$ в Real domain.
RATAN2	0x2052	Rdst, Ra, Rb	Вычисляет $\text{atan2}(y,x)$ в Real domain.
RNEG	0x2060	Rdst, Rsrc	Изменяет знак Real.
RABS	0x2061	Rdst, Rsrc	Модуль Real.
RCMP	0x2062	r64dst, Ra, Rb	Сравнивает Real a/b и возвращает classification {-2,-1,0,+1} в r64.
RCPY	0x2063	Rdst, Rsrc	Копирует Real object совместимой physical width.
RCVT	0x2064	Rdst, Rsrc	Преобразует Real между поддерживаемыми precision widths; conversion PSW-neutral.
RFLOOR	0x2070	Rdst, Rsrc	Округляет Real вниз.
RCEIL	0x2071	Rdst, Rsrc	Округляет Real вверх.
RROUND	0x2072	Rdst, Rsrc	Округляет Real по правилам LibMPCPU.
RFREXP	0x2073	Rfrac, Iexp, Rsrc	Разлагает Real на significand и integer exponent.
RLDEXP	0x2074	Rdst, Rsrc, Iexp	Формирует Real как source, умноженный на степень двойки с integer exponent.
RLOGB	0x2075	Idst, Rsrc	Возвращает integer exponent/classification для Real source.
RMODF	0x2076	Rfrac, Rint, Rsrc	Разделяет Real на fractional и integral parts.
RREMAIN	0x2077	Rdst, Ra, Rb	IEEE/LibMPCPU-style remainder для двух Real operands.
RLOG1P	0x2080	Rdst, Rsrc	Вычисляет $\log(1+x)$ с Real semantics.
REXPM1	0x2081	Rdst, Rsrc	Вычисляет $\exp(x)-1$.
RASINH	0x2082	Rdst, Rsrc	Обратный гиперболический синус.
RACOSH	0x2083	Rdst, Rsrc	Обратный гиперболический косинус.
RATANH	0x2084	Rdst, Rsrc	Обратный гиперболический тангенс.
ITOR	0x2090	Rdst, Isrc	Преобразует signed integer в Real; conversion PSW-neutral.
UITOR	0x2091	Rdst, Isrc	Преобразует unsigned integer в Real; conversion PSW-neutral.
RTOI	0x2092	Idst, Rfrac, Rsrc	Преобразует Real в signed integer destination и отдельно возвращает fractional remainder.
RTOUI	0x2093	Idst, Rfrac, Rsrc	Преобразует Real в unsigned integer destination и отдельно возвращает fractional remainder.
RISNEG	0x20a0	r64dst, Rsrc	Возвращает r64 predicate: является ли Real отрицательным.
RSIGN	0x20a1	r64dst, Rsrc	Возвращает signed Real classification code {-3..+1} в r64.
RISIND	0x20a2	r64dst, Rsrc	Возвращает predicate для indeterminate Real representation.
RISNAN	0x20a3	r64dst, Rsrc	Возвращает predicate для NaN.
RISNANMAX	0x20a4	r64dst, Rsrc	Возвращает predicate для NaN max-class representation.
RISNANMIN	0x20a5	r64dst, Rsrc	Возвращает predicate для NaN min-class representation.
RISINF	0x20a6	r64dst, Rsrc	Возвращает predicate для infinity.
RISSIGNULL	0x20a7	r64dst, Rsrc	Возвращает predicate для signed null/zero classification.
RCONST	0x2100	Rdst, CONST8	Загружает предопределённую математическую Real constant по CONST8 selector.
CADD	0x3000	Cdst, Ca, Cb	Сложение Complex.
CSUB	0x3001	Cdst, Ca, Cb	Вычитание Complex.
CMUL	0x3002	Cdst, Ca, Cb	Умножение Complex.
CDIV	0x3003	Cdst, Ca, Cb	Деление Complex.
CEXP	0x3020	Cdst, Csrc	Комплексная экспонента.
CSIN	0x3021	Cdst, Csrc	Комплексный синус.
CCOS	0x3022	Cdst, Csrc	Комплексный косинус.
CSINH	0x3023	Cdst, Csrc	Комплексный гиперболический синус.
CCOSH	0x3024	Cdst, Csrc	Комплексный гиперболический косинус.
CLOG	0x3025	Cdst, Csrc	Комплексный натуральный логарифм.
CSQRT	0x3026	Cdst, Csrc	Комплексный квадратный корень.
CPOW	0x3050	Cdst, Ca, Cb	Комплексное возведение в степень.
CREAL	0x3060	Rdst, Csrc	Извлекает real component Complex object.
CIMAG	0x3061	Rdst, Csrc	Извлекает imaginary component.
CRTOC	0x3062	Cdst, Rsrc	Преобразует Real в Complex с нулевой imaginary component; conversion PSW-neutral.

Mnemonic	OPCODE16	Operands	Назначение
CGEN	0x3063	Cdst, Rre, Rim	Создаёт Complex из real и imaginary components.
CPOLAR	0x3064	Cdst, Rrho, Rtheta	Создаёт Complex по polar representation rho/theta.
CCONJ	0x3065	Cdst, Csrc	Комплексное сопряжение.
CABS	0x3066	Rdst, Csrc	Возвращает абсолютную величину Complex.
CNORM	0x3067	Rdst, Csrc	Возвращает квадрат нормы Complex.
CARG	0x3068	Rdst, Csrc	Возвращает аргумент/phase Complex.
CISEQ	0x3070	r64dst, Ca, Cb	Возвращает predicate равенства двух Complex.
CISNE	0x3071	r64dst, Ca, Cb	Возвращает predicate неравенства двух Complex.
CCPY	0x3072	Cdst, Csrc	Копирует Complex object совместимой physical width.
CCVT	0x3073	Cdst, Csrc	Преобразует Complex между precision widths; conversion PSW-neutral.
JMP	0x4000	CODE64target	Безусловный прямой переход на PROGRAM address.
JSET	0x4001	PSWBIT8, CODE64	Переходит, если выбранный branchable PSW bit установлен.
JCLR	0x4002	PSWBIT8, CODE64	Переходит, если выбранный PSW bit очищен.
JMPP	0x4003	r64ptr	Косвенный переход через PROGRAM PTR64.
CALL	0x4010	CODE64target	Кладёт address следующей инструкции в STACK и передаёт управление target.
RET	0x4011	- - -	Извлекает return PC из STACK и возвращается из CALL.
CALLP	0x4012	r64ptr	Косвенный CALL через PROGRAM PTR64 с push return PC.
PUSH	0x5000	R/Csrc	Кладёт register object в STACK и изменяет SP согласно stack ABI.
POP	0x5001	R/Cdst	Извлекает register object из STACK.
LDSP	0x5010	R/Cdst, OFF64	Читает object по адресу SP + signed byte offset.
STSP	0x5011	OFF64, R/Csrc	Записывает object по адресу SP + signed byte offset.
ADJSP	0x5020	OFF64	Изменяет SP на signed byte offset с проверкой STACK bounds/alignment policy.
PUSHFP	0x5030	- - -	Сохраняет текущий 64-bit FP в STACK.
SETFP	0x5031	- - -	Устанавливает FP = SP.
RESTSP	0x5032	- - -	Восстанавливает SP = FP.
POPFP	0x5033	- - -	Извлекает 64-bit FP из STACK.
LDFP	0x5040	R/Cdst, OFF64	Читает object по адресу FP + signed byte offset.
STFP	0x5041	OFF64, R/Csrc	Записывает object по адресу FP + signed byte offset.
LEAFP	0x5042	r64dst, OFF64	Формирует STACK PTR64 на FP + signed byte offset.
SETSP	0x5050	r64ptr	Устанавливает SP из валидного STACK PTR64.

3. PSW selectors и RCONST

3.1. PSW bit selectors

Selector	Bit	Смысл для JSET/JCLR и PSW
AF	0	CPU arithmetic auxiliary/carry state; writable by SETF/CLRF
CF	1	CPU carry/borrow flag; writable by SETF/CLRF
OF	2	CPU integer overflow flag; writable by SETF/CLRF
SF	3	CPU sign flag; writable by SETF/CLRF
PF	4	CPU parity/auxiliary result classification flag; writable by SETF/CLRF
ZF	5	CPU zero flag; writable by SETF/CLRF
RF	6	CPU-visible LibMMPU arithmetic status flag; writable by SETF/CLRF
VF	7	CPU-visible LibMMPU value/validity status flag; writable by SETF/CLRF
DOMF	8	MMPU math domain error state; writable by SETF/CLRF
SNGF	9	MMPU singularity error state; writable by SETF/CLRF
OVFF	10	MMPU floating/math overflow state; writable by SETF/CLRF
UDFF	11	MMPU floating/math underflow state; writable by SETF/CLRF
TLSF	12	MMPU total-loss-of-significance state; writable by SETF/CLRF
PLSF	13	MMPU partial-loss-of-significance state; writable by SETF/CLRF
INDF	14	MMPU invalid/indeterminate state; writable by SETF/CLRF
INXF	15	MMPU inexact state; writable by SETF/CLRF
HF	16	HALT-related execution state; readable by JSET/JCLR, not writable by SETF/CLRF

Selector	Bit	Смысл для JSET/JCLR и PSW
TF	17	trace/trap control state; readable by JSET/JCLR, not writable by SETF/CLRF
IF	18	interrupt-enable control state; readable by JSET/JCLR, not writable by SETF/CLRF

Поле MODE является многобитным и намеренно не имеет selector для JSET/JCLR. Архитектурно PSW[7:0] образует CPU flag byte, PSW[15:8] — MPU flag byte. SETF/CLRF разрешают запись только selectors 0..15; CLRCPUF очищает PSW[7:0], CLRMPUF — PSW[15:8]. HF/TF/IF/MODE этими инструкциями не изменяются. Точные правила изменения PSW конкретными арифметическими и conversion instructions описаны в основной части руководства.

3.2. RCONST selectors

Константа	CONST8	Смысл
ZERO	0x00	Точный ноль.
HALF	0x01	$1/2$.
ONE	0x02	1.
TWO	0x03	2.
TEN	0x04	10.
MTEN	0x05	-10 .
32	0x06	32.
PI	0x10	Число π .
E	0x11	Основание натурального логарифма e .
1_LN2	0x12	$1/\ln 2$.
LN2	0x13	$\ln 2$.
1_LN10	0x14	$1/\ln 10$.
LN10	0x15	$\ln 10$.
1_LG2	0x16	$1/\log_{10} 2$.
LG2	0x17	$\log_{10} 2$.
PI_2	0x20	$\pi/2$.
PI_3	0x21	$\pi/3$.
PI_4	0x22	$\pi/4$.
PI_5	0x23	$\pi/5$.
PI_6	0x24	$\pi/6$.
1_PI	0x25	$1/\pi$.
2_PI	0x26	$2/\pi$.
3_PI	0x27	$3/\pi$.
4_PI	0x28	$4/\pi$.
5_PI	0x29	$5/\pi$.
2PI	0x2a	2π .
3PI	0x2b	3π .
1_2PI	0x2c	$1/(2\pi)$.
1_3PI	0x2d	$1/(3\pi)$.
1_4PI	0x2e	$1/(4\pi)$.
1_5PI	0x2f	$1/(5\pi)$.
1_6PI	0x30	$1/(6\pi)$.
3PI_4	0x31	$3\pi/4$.
SQRTPI	0x40	$\sqrt{\pi}$.
1_SQRTPI	0x41	$1/\sqrt{\pi}$.
2_SQRTPI	0x42	$2/\sqrt{\pi}$.
SQRT2	0x43	$\sqrt{2}$.
1_SQRT2	0x44	$1/\sqrt{2}$.
SQRT3	0x45	$\sqrt{3}$.
1_SQRT3	0x46	$1/\sqrt{3}$.
DEGREE	0x47	Один градус в радианах, $\pi/180$.
1_DEGREE	0x48	Число градусов в одном радиане, $180/\pi$.
GOLDENRATIO	0x49	Золотое сечение $(1 + \sqrt{5})/2$.
EULERGAMMA	0x4a	Постоянная Эйлера–Маскерони γ .
CATALAN	0x4b	Постоянная Каталана G .
EPSILON	0x50	LibMPU Real epsilon для выбранной физической ширины результата.

Константа	CONST8	Смысл
IND	0x51	Специальное indeterminate-значение LibMPU.
PNAN	0x52	Положительное NaN-представление LibMPU.
NNAN	0x53	Отрицательное NaN-представление LibMPU.
PNANMAX	0x54	Положительное NaN максимального класса LibMPU.
NNANMAX	0x55	Отрицательное NaN максимального класса LibMPU.
PNANMIN	0x56	Положительное NaN минимального класса LibMPU.
NNANMIN	0x57	Отрицательное NaN минимального класса LibMPU.
PINF	0x58	Положительная бесконечность.
NINF	0x59	Отрицательная бесконечность.
PREALMIN	0x5a	Минимальное положительное конечное Real-значение выбранной ширины.
NREALMIN	0x5b	Минимальное по модулю отрицательное конечное Real-значение выбранной ширины.
PREALMAX	0x5c	Максимальное положительное конечное Real-значение выбранной ширины.
NREALMAX	0x5d	Максимальное по модулю отрицательное конечное Real-значение выбранной ширины.
PZERO	0x5e	Положительный signed zero LibMPU.
NZERO	0x5f	Отрицательный signed zero LibMPU.

4. Краткая памятка ABI

Register file	512 KiB, overlapping REG24 views
Return Value Area	top 8192 bytes, scalar result at 0x80000 - size
PTR64	SEG4:OFF60; DATA=0, STACK=1, PROGRAM=2, CONTROL=3, INPUT=4, OUTPUT=5, TRACE=6, HEAP=7
CALL alignment	16 bytes; CALL pushes 8-byte return PC; caller owns argument block
PSW	bits 0..15 volatile across call; TF/IF/MODE preserved if changed
DATA guard	process offsets 0..15 zero
HEAP	begin/break minimum 16; _end=HEAP:16
Startup	32-byte McpuStartup + argv PTR64 array + strings; persistent
Low-level lifecycle	mandatory _start, _life; optional _fini; e_entry=_start
HLL PROGRAM	defining ET_REL: main+fini; automatic crt0.o+crti.o+crtn.o
HLL MODEL	defining ET_REL: init+life+fini; automatic crt1.o+crti.o+crtn.o
Dynamic	frozen normal ABI v3 with SONAME/NEEDED; ABI v2 remains historical explicit-preload input format