

# Mikroprocesory

## 5. Programování - od C k assembleru

Stanislav Vitek

Katedra radioelektroniky

České vysoké učení technické v Praze

## V předchozích přednáškách jsme viděli

- Co je to ISA a jaké jsou její základní rysy
- Jaká je struktura paměti procesorů ARM
- Jak lze ovládat paměťově mapované periferie
- Registry a pár základních ASM instrukcí
- Základní struktura programu pro STM32F4
  - startup kód
  - linker skript
  - vektor přerušení
  - ResetHandler

## Obsah přednášky

1. Programování vestavných zařízení
2. Preprocesor - od konstant po složité operace
3. Kompilátor - předklad a optimalizace kódu
4. Linker - spojení a umístění programu
5. Programování v ARM assembleru
6. Calling Conventions - AAPCS

# 1. Programování vestavných zařízení

## Bare-metal programování

Cílem je mít plnou kontrolu nad periferiemi, výkonem a velikostí kódu.

- nepoužívá žádný operační systém (např. FreeRTOS)
- nepoužívají se ani HAL knihovny (Hardware Abstraction Layer),
- přistupuje přímo k registrům periferií,
- využívá minimální startup kód a linker script,
- běží přímo po resetu z adresy `0x0800 0000` (flash).

# Úrovně abstrakce

## 1. Přímý přístup do registrů (tzv. register-level programming)

- Používáš přímo registry z Reference Manualu (např. `GPIOA->MODER = 0x1;` )
- Maximální výkon, minimální kód, přesná kontrola

## 2. CMSIS (Cortex Microcontroller Software Interface Standard)

- Používá základní hlavičky od ARM (např. `stm32f4xx.h`, `core_cm4.h` )
- Čitelnější kód, stále blízko železu, snadno přenositelný

## 3. LL (Low Layer) knihovny od ST

- Součást STM32Cube, např. `LL_GPIO_SetOutputPin(GPIOA, LL_GPIO_PIN_5);`
- Kompromis mezi HAL a bare-metal, velmi efektivní

## Proces kompilace

Fáze	Co dělá	Nástroj
<b>Preprocessing</b>	Rozbalí makra, vloží hlavičky, odstraní komentáře	cpp
<b>Kompilace</b>	Převede C → assembler	cc1
<b>Assembling</b>	Přeloží assembler do objektového kódu ( .o )	as
<b>Linkování</b>	Spojí všechny .o + knihovny → .elf	ld
<b>Konverze</b>	Převede .elf → .bin nebo .hex pro programátor	objcopy
<b>Nahrání</b>	Zapíše do MCU paměti (FLASH)	st-flash

## Příklad kódu pro inicializaci a ovládání GPIO

```
#include "stm32f4xx.h"

int main(void)
{
    // inicializace periferie
    RCC->AHB1ENR |= (1 << 3);    // zapnout GPIOD
    GPIOD->MODER |= (1 << 24);   // PD12 = output

    // nekonečná smyčka
    while (1)
    {
        GPIOD->ODR ^= (1 << 12); // toggle LED
    }
}
```

# Preprocessor

Preprocesor není samostatným programem, jeho výstup lze získat volbou kompilátoru

```
arm-none-eabi-gcc -E main.c -o main.i
```

## Výsledek:

- rozbalený soubor `main.i`, jen čistý C kód
- `#include` je nahrazen obsahem souboru
- `#define` je odstraněno, makra jsou expandována
- komentáře odstraněny

Např. `RCC->AHB1ENR` je přepsáno na adresu `*(volatile uint32_t *)0x40023830`).

## Kompilace do assembleru

Volbou kompilátoru je cílová platforma a ISA, která se má pro překlad použít

```
arm-none-eabi-gcc -S -mcpu=cortex-m4 -mthumb main.i -o main.s
```

Vznikne `main.s` – assemblerový zdroják.

```
ldr    r3, =0x40023830
ldr    r2, [r3]
orr    r2, r2, #8
str    r2, [r3]
```

Úlohou kompilátoru je převod na ASM instrukce a správa **ABI** (Application Binary Interface)

- jak se předávají argumenty ( `r0` – `r3` ), kam se ukládá návratová hodnota ( `r0` ), co se musí uložit na zásobník, jak se vytváří rámeček funkce ( `push {lr}` , `pop {pc}` ).

## Spojení objektových souborů, linkování

Pokud je součástí projektu více modulů a knihovny, linker je propojí. Zároveň identifikuje symboly a umístí je do reálné paměti na základě předpisu.

```
arm-none-eabi-gcc main.o startup_stm32f4xx.o \  
-T stm32f4.ld -o main.elf -Wl,-Map=main.map
```

- spojí všechny `.o` do jednoho `.elf`
- použije linker skript `stm32f4.ld` → určí, co kam do FLASH a RAM
- vytvoří `.map` soubor – přehled sekcí a symbolů

Výsledek ( `main.elf` ) je plnohodnotný spustitelný soubor s debug informacemi.

## Konverze pro programátor a programování

Z `.elf` můžeme vytvořit čistý binární obraz (bez debug dat):

```
arm-none-eabi-objcopy -O binary main.elf main.bin
```

Nebo Intel HEX formát:

```
arm-none-eabi-objcopy -O ihex main.elf main.hex
```

Tento soubor se potom nahraje do mikrokontroléru např. pomocí `st-flash`

```
st-flash write main.bin 0x08000000
```

nebo pomocí `openocd`

```
openocd -f interface/stlink.cfg -f target/stm32f4x.cfg \  
-c "program main.elf verify reset exit"
```

## **2. Preprocessor - od konstant po složité operace**

## Jednoduché konstanty a operace

Nejznámější případ:

```
#define LED_PIN    (1 << 5)
#define LED_PORT  GPIOA
```

Při překladu se pouze nahradí text, bez typové kontroly.

Makra umožňují např. přístup do registrů:

```
#define REG32(addr)    (*(volatile uint32_t *)(addr))
#define GPIOA_ODR     REG32(0x40020014)
```

## Parametrická makra

```
#define BIT(n) (1U << (n))

#define SET_BIT(REG, N) ((REG) |= BIT(N))
#define CLEAR_BIT(REG, N) ((REG) &= ~BIT(N))
#define TOGGLE_BIT(REG, N) ((REG) ^= BIT(N))
```

```
#define PERIPH_REG(BASE, OFFSET) (*(volatile uint32_t *)((BASE) + (OFFSET)))
#define RCC_BASE 0x40023800U
#define RCC_AHB1ENR PERIPH_REG(RCC_BASE, 0x30U)

#define ENABLE_GPIO(port) (RCC_AHB1ENR |= (1U << (port)))

void enable_gpioa(void)
{
    ENABLE_GPIO(0); // povolí hodiny pro GPIOA
}
```

# Operátory preprocesoru

## Stringification operátor #

Převede argument makra na řetězec (string literal):

```
#define TO_STRING(x) #x  
  
TO_STRING(hello) // expanduje na "hello"  
TO_STRING(123)  // expanduje na "123"
```

## Token pasting operátor ##

Spojí dva tokeny do jednoho:

```
#define CONCAT(a, b) a##b  
  
CONCAT(GPIO, A) // expanduje na GPIOA  
CONCAT(pin_, 5) // expanduje na pin_5
```

Tyto operátory umožňují dynamické generování názvů a textových konstant.

## Vícestupňová a generická makra

Taková makra se používají např. v CMSIS, FreeRTOS nebo HALu pro generování opakujících se konstrukcí.

**Příklad:** spojování symbolů

```
#define MAKE_REG(port, reg) GPIO##port##_##reg
```

**Použití:**

```
MAKE_REG(A, ODR) = 1; // -> GPIOA_ODR = 1;
```

Toto makro dokáže dynamicky generovat názvy proměnných a struktur podle parametrů — extrémně užitečné v MCU kódu, kde jsou registry pojmenovány systematicky.

## Příklad: generování ISR handlerů

```
#define DEFINE_ISR(name) \
    void name##_IRQHandler(void) { \
        handle_interrupt(#name); \
    }

DEFINE_ISR(TIM2)
DEFINE_ISR(USART1)
```

### Výsledek:

```
void TIM2_IRQHandler(void) { handle_interrupt("TIM2"); }
void USART1_IRQHandler(void) { handle_interrupt("USART1"); }
```

Tohle je zajímavý styl — makro generuje opakující se kód, s parametrizovaným jménem i obsahem. Více rozebereme na konci této sekce.

## Příklad: "pseudogenerické" makro pro registraci driverů

```
#define REGISTER_DRIVER(NAME, INITFN) \
    __attribute__((section(".drivers"))) \
    static const struct driver_entry NAME##_driver = { #NAME, INITFN }
```

### Použití:

```
REGISTER_DRIVER(spi1, spi1_init);  
REGISTER_DRIVER(uart2, uart2_init);
```

Každý zápis vytvoří položku v sekci `.drivers`, kterou může linker sesbírat.

Více rozebereme v části věnované linkeru

## Příklad: makro, které vytváří celé periferie

Takhle to dělají např. výrobci SDK:

```
#define GPIO_PIN_DEFINE(PORT, PIN) \
    static inline void GPIO##PORT##_PIN##PIN##_Set(void) \
    { GPIO##PORT->BSRR = (1 << PIN); } \
    static inline void GPIO##PORT##_PIN##PIN##_Reset(void) \
    { GPIO##PORT->BSRR = (1 << (PIN + 16)); }
```

### Použití:

```
GPIO_PIN_DEFINE(A, 5) // Vygeneruje GPIOA_PIN5_Set() a GPIOA_PIN5_Reset()
```

## Rizika a omezení

Makra jsou výkonná, ale mají:

- žádnou typovou kontrolu (vše je textová náhrada),
- složitou laditelnost (debugger ukazuje kód po expanzi),
- těžkou čitelnost při vícestupňovém vnoření,
- neočekávané chování při použití `++` nebo `--` uvnitř parametrů.

Typická chyba:

```
#define SQR(x) ((x)*(x))
int a = 2;
int b = SQR(a++); // problém: expanduje se na ((a++)*(a++))
```

## Alternativy

Modernější (a bezpečnější) přístupy:

- static inline funkce místo maker
  - kompilátor umí inline optimalizovat stejně, ale s typovou kontrolou.
- constexpr / templates v C++ (kde to MCU dovolí).
- Generátory kódu (Python, CMake, CubeMX) – místo extrémních maker.

## Generování ISR handlerů pomocí makra

- ISR (Interrupt Service Routine, obsluha přerušení) je speciální funkce, kterou procesor zavolá asynchronně při události (např. timer, UART RX, DMA done, EXTI pin).
- Má vyšší prioritu než běžný kód, přerušuje běh programu, a po dokončení se návrat děje přes instrukci `bx lr` s návratem z privilegovaného režimu.
- Proto musí ISR být:
  - co nejkratší, deterministická (neměla by se zdržovat), určitě bez rekurze
  - bez blokování (`while`, `printf`, `malloc`, `sleep`, ...),

### Je proto rozumné volat v ISR další funkci?

Ano, ale s rozumem

## Příklad: “přímá” obsluha vs. přenesení do jiné funkce

### Špatně

- ISR přímo dělá vše
- dlouhé, blokující, nedeterministické.

```
void USART1_IRQHandler(void) {  
    while (!(USART1->SR & USART_SR_RXNE));  
    uint8_t c = USART1->DR;  
    printf("RX: %c\n", c); // velký problém – printf může dlouho čekat  
}
```

## Dobře

- V ISR jen nejnütnější operace (signalizace)
- Skutečná obsluha (práce s daty) v jiné funkci

```
volatile uint8_t rx_byte;
volatile bool rx_ready = false;

void USART1_IRQHandler(void) {
    if (USART1->SR & USART_SR_RXNE) {
        rx_byte = USART1->DR;
        rx_ready = true;
    }
}

void main (void) {
    for (;;) {
        if (rx_ready) {
            rx_ready = false;
            handle_rx_byte(rx_byte);
        }
    }
}
```

## Příklad generického makra pro ISR

```
#define DEFINE_ISR(name)          \
    void name##_IRQHandler(void) { \
        handle_interrupt_##name(); \
    }

DEFINE_ISR(TIM2)
DEFINE_ISR(USART1)
```

Konkrétní handler pro USART1 :

```
static inline void handle_interrupt_USART1(void) {
    if (USART1->SR & USART_SR_RXNE) {
        uint8_t c = USART1->DR;
        ringbuffer_put(&uart_rx_buf, c);
    }
}
```

### **3. Kompilátor - předklad a optimalizace kódu**

# Fáze kompilace

## 1. Lexikální analýza

- rozdělení zdrojového kódu na tokeny ( `if` , `x` , `=` , `42` , `;` )

## 2. Syntaktická analýza (parser)

- z tokenů vzniká abstraktní syntaktický strom (AST)

## 3. Sémantická analýza

- kontrola typů, rozsahů proměnných, volání funkcí

## 4. Intermediate Representation (IR)

- AST se převede do mezikódu (např. GIMPLE / SSA v GCC)
- zde se dělají optimalizace (např. propagace konstant, inlining)

## 5. Optimalizace podle úrovně `-O0` až `-O3`

## 6. Generování assembleru

## Příklad

```
int sum(int *arr, int n) {  
    int s = 0;  
    for (int i = 0; i < n; i++) {  
        s += arr[i];  
    }  
    return s;  
}
```

## Lexikální analýza

- Kompilátor nejprve rozdělí kód na tokeny, tedy základní lexikální jednotky:

```
[int] [sum] [(] [int] [*] [arr] [,] [int] [n] [)] [{}]  
[int] [s] [=] [0] [;]  
[for] [(] [int] [i] [=] [0] [;] [i] [<] [n] [;] [i] [++] [)]  
[s] [+]= [arr] [ [ ] [i] [ ] ] [;]  
[return] [s] [;]  
[}]
```

Každý token má typ a hodnotu

- `int` → klíčové slovo, `s` → identifikátor, `=` → operátor přiřazení, `0` → literál atd.

Lexikální analyzátor už tady odstraní mezery a komentáře, ale neví nic o významu.

## Syntaktická analýza

Parser vytvoří abstraktní syntaktický strom (AST) kde každý příkaz, výraz, nebo operátor má svůj uzel.

```
FunctionDecl "sum"
├── Param arr : int*
├── Param n   : int
├── CompoundStmt
│   ├── VarDecl s : int = 0
│   ├── ForStmt
│   │   ├── Init: VarDecl i : int = 0
│   │   ├── Condition: (BinaryOperator '<', Identifier i, Identifier n)
│   │   ├── Increment: (UnaryOperator '++', Identifier i)
│   │   └── Body:
│   │       └── CompoundStmt
│   │           └── BinaryOperator '+=', Identifier s, ArraySubscript(arr, i)
│   └── ReturnStmt(Identifier s)
```

## Syntaktická analýza

Možná detekce chyb: špatné závorky, chybějící středník, neplatné výrazy atd.

## Sémantická analýza

Na AST se provádí kontroly významu:

- odpovídají typy? (např. `arr[i]` je typu `int` )
- proměnné jsou deklarovány?
- návratová hodnota odpovídá `int` ?

Pokud všechno sedí, pokračuje se do další fáze.

## Intermediate Representation

Kód se přepíše do abstraktního mezikódu, kde

- už nejsou složitosti jazyka C (typy, složené výrazy, scoping...),
- ale ještě to není specifické pro konkrétní procesor (ARM, RISC-V...).

```
s_1 = 0;
i_2 = 0;
L1:
  if (i_2 >= n_0) goto L2;
  s_1 = s_1 + arr_0[i_2];
  i_2 = i_2 + 1;
  goto L1;
L2:
  return s_1;
```

Kompilátor zde provádí optimalizace, např. rozbalování smyček, eliminaci mrtvého kódu, ...

## Generování assembleru (pro ARM Cortex-M, -00)

Bez optimalizace (zachována struktura C):

```
sum:
    push    {r4, r5, r6, lr}
    movs   r3, #0
    movs   r4, #0
.L2:
    cmp    r4, r1
    bge   .L3
    ldr    r5, [r0, r4, lsl #2]
    add   r3, r3, r5
    adds  r4, r4, #1
    b     .L2
.L3:
    mov   r0, r3
    pop  {r4, r5, r6, pc}
```

## Prolog – vytvoření rámce

```
push {r4, lr}
```

Na začátku funkce ukládáme registry `r4` a `lr` (link register) na zásobník.

- `lr` obsahuje návratovou adresu (kam se má např. `bx lr` vrátit).
- `r4` se ukládá, protože se používá jako "callee saved" registr (ARM AAPCS říká, že funkce, která `r4` použije, ho musí po sobě obnovit.)
  - GCC to dělá automaticky na `-O0` vždy, když použije `r4`.

Pozn.: Cortex-M má registr konvenci:

- `r0 – r3` → argumenty a dočasné proměnné (caller-saved),
- `r4 – r11` → zachovávané mezi funkcemi (callee-saved).

## Inicializace lokálních proměnných

```
movs    r3, #0    ; s = 0
movs    r4, #0    ; i = 0
```

- r3 drží akumulátor s
- r4 drží index i

Na `-00` si GCC dává velmi záležet, aby bylo možné v debuggeru sledovat proměnné přesně podle kódu v C — i kdyby to znamenalo méně efektivní ASM.

Proto vytváří oddělené registry pro s i i, místo aby použil jeden.

## Tělo smyčky

```
L2:
    cmp    r4, r1           ; if (i >= n)
    bcs   .L3             ; break
    ldr   r2, [r0, r4, lsl #2] ; a[i]
    adds  r3, r3, r2       ; s += a[i]
    adds  r4, r4, #1       ; i++
    b     .L2             ; repeat
L3:
```

r0 → pointer a (1. argument)

r1 → délka n (2. argument)

r4 → index i

r3 → akumulátor s

ldr [r0, r4, lsl #2] → načti a[i] (32b int → i\*4 )

bcs (branch if carry set) → větvení, když r4 >= r1

## Epilog – návrat z funkce

```
mov    r0, r3    ; návratová hodnota  
pop    {r4, pc}  ; obnoví r4 a skočí na uloženou adresu (ret)
```

r0 → návratový registr podle ABI,

pop {r4, pc} → „restore and return“

- elegantní trik: při obnově pc se procesor vrátí na adresu z lr .

## -O1 — základní optimalizace

```
sum:
    movs    r2, #0
    movs    r3, #0
.L2:
    cmp     r3, r1
    bcs     .L3
    ldr     r12, [r0, r3, lsl #2]
    adds   r2, r2, r12
    adds   r3, r3, #1
    b      .L2
.L3:
    mov     r0, r2
    bx     lr
```

- odstraněn nepotřebný push/pop rámeček,
- návrat přes `bx lr`,
- menší počet registrů.

## -O2 — optimalizace výkonu

```
sum:
    cmp     r1, #0
    movs   r2, #0
    beq    .L3
.L2:
    ldr    r3, [r0], #4
    subs   r1, r1, #1
    adds   r2, r2, r3
    bne    .L2
.L3:
    mov    r0, r2
    bx    lr
```

- počítadlo jde dolů (místo `i < n` -> `while(n--)`), což zkracuje testovací instrukce,
- post-inkrementace ukazatele (`ldr [r0], #4`),
- žádný zbytečný registr pro index.

## -O3 — agresivní optimalizace, unrolling (1/2)

```
sum:
    cmp     r1, #0
    beq     .L3
    movs    r2, #0
.L4:
    ldr     r3, [r0], #4
    subs    r1, r1, #1
    adds    r2, r2, r3
    bne     .L4
    mov     r0, r2
    bx     lr
```

V tomto případě je kód optimalizovaný -O3 podobný kódu -O2 , protože smyčka je krátká.

## -O3 — agresivní optimalizace, unrolling (2/2)

Co kdyby `n` bylo známé konstantně - např. `sum(a, 4)` ?

GCC by smyčku rozbalil (unrolled):

```
ldr r2, [r0]
ldr r3, [r0, #4]
ldr r12, [r0, #8]
ldr r1, [r0, #12]
adds r0, r2, r3
adds r0, r0, r12
adds r0, r0, r1
bx lr
```

Loop unrolling patří mezi smyčkové optimalizace (loop optimization), která zrychluje vykonávání kódu na úkor jeho velikosti.

Cílem je minimalizace instrukcí porovnávání a manipulace s řídicí proměnnou.

## Další úrovně optimalizace

### -Os (optimize for size)

```
sum:
    cbz    r1, .L3        ; compare and branch if zero
    movs   r2, #0
.L2:
    ldr    r3, [r0], #4
    subs   r1, r1, #1
    add    r2, r2, r3     ; bez 's' suffix (kratší instrukce)
    bne    .L2
.L3:
    mov    r0, r2
    bx    lr
```

- Minimalizuje velikost kódu, důležité pro MCU s omezenou programovou pamětí
- Používá kratší instrukce (např. `cbz` místo `cmp` + `beq` )
- Kompromis mezi rychlostí a velikostí

## -Og (optimize for debugging)

- Zachovává strukturu kódu podobnou `-O0`, ale s menšími optimalizacemi
- Lepší pro debugging než `-O1`, ale efektivnější než `-O0`
- Proměnné zůstávají v paměti/registrech, kde je debugger očekává
- Užitečné při vývoji, když potřebuješ debugovat, ale `-O0` je příliš pomalé

### Kdy použít kterou úroveň:

- `-O0` : vývoj, plné možnosti ladění
- `-Og` : vývoj s laděním, ale potřeba vyššího výkonu
- `-O1/-O2` : produkční build, vyvážený výkon/velikost
- `-O3` : maximální výkon, větší kód
- `-Os` : minimální velikost, embedded systémy s omezenou programovou pamětí

## Další příklady, kde se úroveň optimalizace dramaticky projeví

### 1. Podmíněné větvení s konstantou

Např. `if (x > 0) return 1; else return 0;` se přeloží na `cmp` + `movgt` (conditional move).

### 2. Volání malých funkcí

`-O2` a `-O3` často funkci vkládají jako inline - žádný `bl`, žádné zásobníkové operace.

### 3. Optimalizace `memcpy`

Pro malé délky GCC generuje vlastní kód místo volání knihovny.

## Optimalizace vložením inline assembleru

Inline assembler ( `__asm__` nebo `asm( )` ) se používá, když je potřeba:

- využít speciální instrukci (např. `WFI` , `BKPT` , `REV16` ),
- optimalizovat smyčku nebo aritmetiku, kterou kompilátor nepozná,
- přistupovat k registrům procesoru (např. `PSR` , `CONTROL` , `PRIMASK` ).

### Základní syntaxe

```
asm [volatile] ("assembly code"  
                : output operands  
                : input operands  
                : clobbered registers);
```

## Modifikátory pro vkládání inline assembleru

`r` → general purpose register

`m` → operand v paměti, přístup do RAM/periferie

`i` → konstanta známá při překladu, např. posun, maska

`I` → malá konstanta 0-255, typicky pro `immediate` v ARM instrukcích

`J` → immediate -4095 to 4095, např. offset při adresaci

`=r` → output register, kompilátor přidělí registr a zajistí bezkoliznost

`+r` → input/output register, in-place operace ( `x++` )

## Příklad 1 - přímé vložení instrukce

```
__asm__("wfi");
```

"Wait For Interrupt" – uspí jádro, dokud nepřijde přerušení.

## Příklad 2 – čtení CPU registru

```
uint32_t get_psr(void) {  
    uint32_t result;  
    __asm__ volatile ("mrs %0, psr" : "=r"(result));  
    return result;  
}
```

`mrs` čte speciální registr `PSR` (Program Status Register). Kompilátor to neumí napsat přímo v C.

### Příklad 3

```
int x = 5, y;
__asm__ ("adds %0, %1, #1"
        : "=r" (y)      // výstup
        : "r" (x)      // vstup
        : "cc");       // mění příznakový registr
```

### Příklad 4

```
void increment_by_five(int *value) {
    asm("ldr r0, %1\n\t"      // Load value
        "add r0, r0, #5\n\t" // Add 5
        "str r0, %0"        // Store back
        : "=m" (*value)     // output: memory location
        : "m" (*value)     // input: memory location
        : "r0");           // clobbered register
}
```

## Příklad 5 – zrychlení bitového počítání (1/2)

```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v

for (c = 0; v; v >>= 1)
{
    c += v & 1;
}
```

Naivní verze, která projde cyklus tolikrát, kolik bitů má `v`

```
for (c = 0; v; c++)
{
    v &= v - 1; // clear the least significant bit set
}
```

Verze Briana Kernigana, vyžaduje tolik iterací, kolik je v čísle jedniček

## Příklad 5 – zrychlení bitového počítání (2/2)

```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v

__asm__("popcnt %0, %1" : "=r"(c) : "r"(v));
```

ARM M4 má instrukci `popcnt` (v rámci DSP rozšíření). Instrukce spočítá bity v jediném cyklu.

Kompilátor ji obvykle nepoužije, pokud výslovně není nastavena optimalizace pro DSP

```
gcc -mfpv4-sp-d16 -mfloat-abi=hard
```

## Příklad 6 – čekací smyčka

```
static inline void delay_cycles(uint32_t n) {  
    __asm__ volatile (  
        "1: subs %[n], %[n], #1 \n"  
        "   bne 1b          \n"  
        : [n] "+r"(n)  
    );  
}
```

- `volatile` zajistí, že se smyčka nesmaže,
- `subs` a `bne` se vykonají přesně n-krát,
- žádný overhead C funkce, žádné přepisy registrů.

Typicky o 15–30 % rychlejší než totéž v C, protože kompilátor často přidává kontrolu a overhead při volání funkce.

## Příklad 7 - přístup k GPIO

```
#define GPIO_BASE 0x40020000
#define GPIOA_ODR (GPIO_BASE + 0x14)

void set_gpio_pin(int pin) {

    volatile uint32_t *gpio_odr = (uint32_t*)GPIOA_ODR;

    asm volatile("ldr r0, %1\n\t" // Load current value
                 "orr r0, r0, %2\n\t" // Set bit
                 "str r0, %0" // Store back
                 : "=m" (*gpio_odr) // output
                 : "m" (*gpio_odr), "r" (1 << pin) // inputs
                 : "r0" // clobbered
                 );
}
```

## Vliv direktivy `volatile`

```
#define STATUS_REG (*(uint32_t*)0x40000000)

void wait_ready(void) {
    while ((STATUS_REG & 0x1) == 0) {
        // čekáme na bit 0 = 1
    }
}
```

Kompilátor vidí, že `STATUS_REG` je obyčejná proměnná a během smyčky se nemění.

```
wait_ready:
    ldr    r3, =0x40000000
    ldr    r2, [r3]
    tst    r2, #1      ; provede AND mezi operandy, nastaví příznakový registr
    beq    .L2
.L2:
    bx    lr          ; smyčka odstraněna jako "nekonečná"
```

## Opravená verze

```
#define STATUS_REG (*(volatile uint32_t*)0x40000000)
```

Pak GCC ví, že se může hodnota měnit mimo kód, a vždy znovu čte z paměti.

```
wait_ready:  
    ldr    r3, =0x40000000  
.L1:  
    ldr    r2, [r3]  
    tst    r2, #1  
    beq    .L1  
    bx    lr
```

## Příklad — čtení senzoru nebo periferie

```
uint16_t read_adc(void) {  
    while (!(ADC1->SR & (1 << 1))) { } // čekáme na EOC (end of conversion)  
    return ADC1->DR;  
}
```

Bez `volatile` na struktuře `ADC_TypeDef` (nebo jejích polích) kompilátor:

- přečte `ADC1->SR` jednou,
- uloží si to do registru,
- a smyčku nikdy neopustí.

Proto ST všechny periferní registry v CMSIS definuje jako `volatile uint32_t`.

## Příklad - flag při přerušení

Časté řešení obsluhy přerušení, ISR nastaví globální proměnnou:

```
int flag = 0;

void main (void) {
    while (!flag) {
        // čekáme na přerušení
    }
}
```

Obslužná rutina přerušení nastavuje řídicí proměnnou `flag`

```
void EXTI0_IRQHandler(void) {
    flag = 1;
}
```

Bez `volatile` zrychlí při optimalizaci kompilátor kód tak, že uloží hodnotu `flag` do registru (zde `r3`) a ten už neaktualizuje.

```
ldr r3, =flag      ; načte adresu flag
ldr r3, [r3]       ; načte flag do r3 (např. hodnota 0)
loop:
  cmp r3, #0
  beq loop         ; stále 0 => stále běžíme
```

S `volatile` aktualizuje hodnotu registru v každé iteraci.

```
loop:
  ldr r3, =flag
  ldr r3, [r3]     ; znovu načte z paměti
  cmp r3, #0
  beq loop
```

## Co přesně **volatile** říká kompilátoru?

Deklarace:

```
volatile uint32_t x;
```

znamená:

Tahle proměnná se může změnit kdykoli mimo kontrolu kompilátoru — např. hardwarem nebo přerušením — a proto musí být při každém čtení znovu načtena z paměti a při každém zápisu zapsána přímo do paměti.

**Zjednodušeně:**

- Kompilátor se nesmí spoléhat na žádné mezivýsledky, cache nebo registry týkající se této hodnoty.

## Praktický příklad — vliv na výkon

```
volatile uint32_t *GPIOA_ODR = (uint32_t*)0x40020014;

void toggle_led(void) {
    *GPIOA_ODR ^= (1 << 5);
}
```

Každé zavolání udělá:

1. čtení registru z periférie,
2. XOR s maskou,
3. zápis zpět.

Protože je konstanta `volatile`, kompilátor neoptimalizuje — ani při volání ve smyčce:

```
for (int i = 0; i < 1000; i++) toggle_led();
```

## Výsledek:

- Každý průchod smyčky = plný přístup přes sběrnici AHB,
- CPU čeká na periférii (latence, bus stall),
- žádná možnost zrychlení, slučování, unrollingu, nic.

Bez `volatile` by to kompilátor zoptimalizoval třeba takto:

```
uint32_t tmp = *GPIOA_ODR;  
tmp ^= (1 << 5);  
*GPIOA_ODR = tmp;
```

A pokud by volání bylo ve smyčce, dokonce by to unrollnul nebo přemístil instrukce tak, aby běžely v pipeline co nejrychleji.

Ale s `volatile` to má kompilátor zakázané.

## Co z toho plyne?

`volatile` je bezpečnostní brzda, ne optimalizační nástroj.

Programátor má jistotu, že přistupuje opravdu k HW registrům, ale při zbytečném použití se zabije výkon i flexibilita překladače.

Proto se často dělá kompromis:

- HW registry → `volatile` vždy (to je správné)
- kritické úseky → `inline asm (asm volatile)` místo označování celé proměnné

`volatile` nebrání přeuspořádání instrukcí vůči jiným `volatile` přístupům

- cacheovaná data nebo sdílené proměnné → třeba zabránit přeuspořádání instrukcí
- je potřeba použít paměťovou bariéru

## Paměťová bariéra

Kompilátor C, pokud má pocit, že to nijak nezmění chování programu, může volně:

- přesouvat čtení a zápisy do paměti, slučovat přístupy, zahazovat redundantní R/W:

```
asm volatile ("" ::: "memory");
```

Tento kód je tzv. „compiler memory barrier“ a říká kompilátoru, že „paměť se mohla změnit“:

- všechny dřívější zápisy do paměti musí být dokončeny před bariérou,
- všechny následující čtení musí číst z aktuální paměti,
- a kompilátor nesmí tyto operace přeuspořádat přes bariéru.

```
// CMSIS  
__DSB(); // Data Synchronization Barrier  
__ISB(); // Instruction Synchronization Barrier
```

```
ready = 1;
asm volatile ("" ::: "memory"); // zajistí pořadí
if (ready) start_dma();
```

Bariéra zajistí, že `ready = 1` skutečně proběhne před tím, než se vyhodnotí `if`.

```
REG_CTRL = ENABLE;
asm volatile ("" ::: "memory");
REG_START = 1;
```

Bez bariéry by mohl kompilátor zápis při optimalizaci přehodit (protože mu to z pohledu jazyka C připadá jedno, přitom na pořadí operací záleží).

```
memcpy(buffer, data, len);
asm volatile ("" ::: "memory");
DMA->START = (uint32_t)buffer;
```

Bariéra zajistí, že CPU nejdříve dokončí zápisy do bufferu, než spustí DMA.

## Globální / statické proměnné

V C existují dva hlavní typy statických proměnných:

### Globální statické proměnné

```
static int counter = 0; // viditelné jen v tomto souboru
```

Paměť je alokována v datové sekci ( `.data` nebo `.bss` ) při startu MCU.

### Lokální statické proměnné

```
void foo(void) {  
    static int counter = 0; // zachovává hodnotu mezi voláními  
    counter++;  
}
```

Paměť alokována v datové sekci (ne zásobník), hodnota se neztrácí po opuštění funkce.

## Specifika pro embedded / STM32

- sekce `.data` → inicializované proměnné, kopírovány z Flash do RAM při startu.
- sekce `.bss` → neinicializované statické proměnné, na `0` nastavena při startu.
- RAM footprint je důležitý → statické proměnné zůstávají po celou dobu běhu MCU.
- pokud se ke statické proměnné přistupuje z ISR, musí být volatile nebo chráněna.

## Statické proměnné vs MISRA

8.5 Statická proměnná musí být inicializována před použitím

8.8 Lokální statické proměnné musí být použity v rámci funkce

8.10 Nepoužívané globální statické proměnné (nebo okomentovat důvod)

8.12 Lokální statické proměnné musí být inicializovány konstantou

9.1/9.3 Proměnné přístupné z ISR musí být správně označeny volatile

## 4. Linker - spojení a umístění programu

Linker spojuje všechny objektové soubory a knihovny do spustitelného obrazu programu.

- Sloučí sekce kódu a dat ( `.text` , `.data` , `.bss` , `.rodata` ,...)
- Vyřeší odkazy mezi moduly – propojí volání funkcí a přístupy k proměnným
- Rozmístí paměť podle linker skriptu – přesně určí adresy v FLASH a RAM
- Vytvoří výstupní formát – např. ELF, BIN, HEX

V embedded kontextu

- Linker musí respektovat hardwarové adresy periférií, paměťové mapy a vektory přerušení
- Správné rozmístění sekcí rozhoduje o spolehlivosti i velikosti kódu
- Umožňuje tvorbu bootloaderů, driver sekcí, custom memory regionů

## Příklad linker scriptu pro STM32F4

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512K
  RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 128K
}

SECTIONS
{
  .text : {
    KEEP(*( .isr_vector)) /* Vektor přerušení na začátek */
    *(.text*) /* Kód programu */
    *(.rodata*) /* Konstanty (read-only data) */
  } > FLASH

  .data : {
    _sdata = .; /* Start .data v RAM */
    *(.data*)
    _edata = .; /* Konec .data */
  } > RAM AT> FLASH /* V RAM, ale kopírováno z FLASH */

  .bss : {
    _sbss = .;
    *(.bss*)
    _ebss = .;
  } > RAM /* Neinicializovaná data */
}
```

## Co libc nebo jiná systémová knihovna?

`libc` je implementace základních funkcí jazyka C (např. `memcpy`, `printf`, `malloc`, `strlen`, atd.).

Na běžných počítačích je to např. `glibc`, ale ta je pro mikrokontroléry nepoužitelná — potřebuje OS.

Pro bare-metal se používají minimalistické a přenositelné varianty, které:

- nevyžadují souborový systém ani OS,
- umožňují vlastní implementaci systémových volání (např. `_write`, `_sbrk`),
- a jsou kompatibilní s `arm-none-eabi-gcc`.

# Hlavní varianty libc pro ARM bare-metal

## newlib

- GNU toolchain
- Plná implementace libc (včetně stdio, malloc, ...)
- Kompatibilní, snadno dostupná
- Velká (100–200 kB), nutno implementovat „syscalls“

## newlib-nano

- Součást arm-none-eabi
- Zjednodušená verze newlib
- Malá velikost (~30 kB), printf ořezaný
- Bez podpory float v printf, chybí některé funkce

# Hlavní varianty libc pro ARM bare-metal

## **picolibc**

- Nová, sloučenina newlib + avr-libc
- Optimalizovaná pro MCU, menší footprint
- Aktivní vývoj, lepší než newlib-nano
- Méně rozšířená

## **microlib**

- ARM/Keil
- Vlastní minimalistická knihovna ARM
- Extrémně malá (desítky bajtů)
- Jen v Keil toolchainu

## Jak se libc propojuje se zbytkem systému

Při použití např. `printf`, `malloc` nebo `open` libc očekává, že existují tzv. systémová volání:

```
int _write(int fd, const void *buf, size_t count);  
void *_sbrk(ptrdiff_t incr);  
int _read(int fd, void *buf, size_t count);
```

Ty je třeba v bare-metal projektu implementovat (nebo nahradit dummy verzí).

Příklady implementace na následujícím slide.

```

int _write(int fd, const void *buf, size_t count) {
    const uint8_t *ptr = buf;
    for (size_t i = 0; i < count; i++) {
        ITM_SendChar(ptr[i]); // nebo přes UART
    }
    return count;
}

void *_sbrk(ptrdiff_t incr) {
    extern char _end; // z linker scriptu
    static char *heap_end;
    char *prev_heap_end;

    if (heap_end == 0) heap_end = &_end;
    prev_heap_end = heap_end;
    heap_end += incr;
    return (void *)prev_heap_end;
}

```

# Volba libc při kompilaci

Při kompilaci pomocí `arm-none-eabi-gcc` si lze vybrat variantu `libc` pomocí linker prepínačů:

## 1. standardní (newlib)

```
arm-none-eabi-gcc main.c -o main.elf -lc -lnosys
```

## 2. odlehčená verze (newlib-nano)

```
arm-none-eabi-gcc main.c -o main.elf -lc -lnosys --specs=nano.specs
```

## 3. bez libc úplně

```
arm-none-eabi-gcc main.c -o main.elf -nostdlib -nostartfiles
```

`-lnosys` přidává stuby systémových volání, které vrací chybové kódy místo skutečných funkcí.

## Kompilace a linkování bez `-lnosys`

```
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

`newlib` očekává, že někdo implementuje syst. volání `_write`, `_sbrk`, `_read`, ...

```
arm-none-eabi-gcc main.c -o main.elf

/usr/lib/gcc/arm-none-eabi/.../libgcc.a(_write.o): in function `_write_r':
undefined reference to `_write'
/usr/lib/gcc/arm-none-eabi/.../libc.a(lib_a-sbrkr.o): in function `_sbrk_r':
undefined reference to `_sbrk'
collect2: error: ld returned 1 exit status
```

## Co udělá `-lnosys`

S volbou `-lnosys` nebo variantou `-specs=nosys.specs` :

```
arm-none-eabi-gcc main.c -o main.elf -lnosys
```

Linker připojí knihovnu `libnosys.a` , která obsahuje minimální implementace:

```
_ssize_t _write(int file, const void *ptr, size_t len) {  
    (void)file; (void)ptr; (void)len;  
    return -1; // žádné zařízení k zápisu  
}  
  
void _exit(int status) {  
    while (1) { } // nikdy se nevrátí  
}
```

Linker je spokojený - všechny symboly existují.

## Alternativa: vlastní implementace systémových volání

Pokud je potřeba, aby `printf` fungovala (např. přes UART), lze implementovat vlastní funkci `_write()`:

```
int _write(int file, char *ptr, int len) {
    for (int i = 0; i < len; i++) {
        uart_send_char(ptr[i]); // napojení na periférii
    }
    return len;
}
```

V tu chvíli už `-lnosys` není potřeba — linker použije tvoji implementaci.

## 5. Programování v ARM assembleru

## 32-bit ARM Instructions:

[31:28] Condition | [27:20] Instruction Type | [19:0] Operands

Příklad: `0xE2800004` = `ADD R0, R0, #4`  
`1110 0010 1000 0000 0000 0000 0000 0100`  
EQ     ADD     R0,R0     immediate #4

## 16-bit Thumb Instructions:

[15:10] Instruction | [9:0] Operands |

Příklad: `0x3004` = `ADD R0, #4`  
`001100 0000000100`  
ADD     R0     #4

## Thumb-2 (Mixed 16/32-bit):

- 16b instrukce pro běžné operace, 32b instrukce pro komplexní operace
- Lepší hustota kódu než čistý ARM, kompatibilní s Cortex-M procesory

## Praktický příklad

```
int global_var = 42;
void add_to_global(int x) {
    global_var += x;
}
```

```
.global global_var
.data
global_var: .word 42

.text
.global add_to_global
add_to_global:
    ldr r1, =global_var      ; Načti adresu global_var
    ldr r2, [r1]             ; Načti hodnotu global_var
    add r2, r2, r0           ; Přičti x (v registru r0)
    str r2, [r1]            ; Ulož zpět do global_var
    bx lr                   ; Návrat z funkce (bez návratové hodnoty)
```

## Instukce pro práci s daty

### Aritmetické operace:

```
; Základní aritmetika
add r0, r1, r2          ; r0 = r1 + r2
sub r0, r1, r2          ; r0 = r1 - r2
mul r0, r1, r2          ; r0 = r1 * r2 (32-bit result)
umull r0, r1, r2, r3    ; r1:r0 = r2 * r3 (64-bit result)

; S immediate operandy
add r0, r1, #42         ; r0 = r1 + 42
sub r0, r1, #0x100      ; r0 = r1 - 256

; S condition flags
adds r0, r1, r2         ; r0 = r1 + r2, nastaví NZCV flags
subs r0, r1, r2         ; r0 = r1 - r2, nastaví NZCV flags
```

## Logické operace

```
and r0, r1, r2      ; r0 = r1 & r2
orr r0, r1, r2      ; r0 = r1 | r2
eor r0, r1, r2      ; r0 = r1 ^ r2 (XOR)
bic r0, r1, r2      ; r0 = r1 & (~r2) (bit clear)
mvn r0, r1          ; r0 = ~r1 (NOT)
```

## Shift operace

```
lsl r0, r1, #2      ; r0 = r1 << 2 (logical shift left)
lsr r0, r1, #2      ; r0 = r1 >> 2 (logical shift right)
asr r0, r1, #2      ; r0 = r1 >> 2 (arithmetic shift right)
ror r0, r1, #2      ; r0 = rotate right o 2 bity
```

## Kombinované operace

```
add r0, r1, r2, lsl #2 ; r0 = r1 + (r2 << 2)
sub r0, r1, r2, ror #8  ; r0 = r1 - rotate_right(r2, 8)
```

# Load/Store instrukce a módy adresování

## Základní Load/Store:

```
ldr r0, [r1]           ; r0 = *r1
str r0, [r1]           ; *r1 = r0
ldrb r0, [r1]          ; r0 = *(uint8_t*)r1
strb r0, [r1]          ; *(uint8_t*)r1 = r0
ldrh r0, [r1]          ; r0 = *(uint16_t*)r1
strh r0, [r1]          ; *(uint16_t*)r1 = r0
```

## Módy adresování:

### 1. Immediate offset:

```
ldr r0, [r1, #4]       ; r0 = *(r1 + 4)
str r0, [r1, #8]       ; *(r1 + 8) = r0
```

## 2. Register offset:

```
ldr r0, [r1, r2]           ; r0 = *(r1 + r2)
ldr r0, [r1, r2, lsl #2]; r0 = *(r1 + (r2 << 2))
```

## 3. Pre-indexed:

```
ldr r0, [r1, #4]!         ; r1 = r1 + 4; r0 = *r1
str r0, [r1, #-4]!       ; r1 = r1 - 4; *r1 = r0
```

## 4. Post-indexed:

```
ldr r0, [r1], #4         ; r0 = *r1; r1 = r1 + 4
str r0, [r1], #-4       ; *r1 = r0; r1 = r1 - 4
```

# Instrukce větvení a podmíněné vykonávání

## Condition Flags (CPSR)

```
N = Negative (bit 31)
Z = Zero (bit 30)
C = Carry (bit 29)
V = Overflow (bit 28)
```

## Instrukce větvení

```
b label           ; Unconditional branch
bl function       ; Branch with link (function call)
bx lr             ; Branch exchange (return)

; Conditional branches
beq label         ; Branch if equal
bne label         ; Branch if not equal
blt label         ; Branch if less than
bgt label         ; Branch if greater than
```

## Conditional Codes

Code	Meaning	Flags	Code	Meaning	Flags
EQ	Equal	Z=1	GT	Greater Than	Z=0 AND N=V
NE	Not Equal	Z=0	GE	Greater Equal	N=V
LT	Less Than	N≠V	HI	Higher (unsigned)	C=1 AND Z=0
LE	Less Equal	Z=1 OR N≠V	LS	Lower Same (unsigned)	C=0 OR Z=1

## Pomíněné vykonávání

```
cmp r0, #0
addgt r1, r1, #1      ; if (r0 > 0) r1++
movle r1, #0         ; else r1 = 0
```

**Pozor:** Podmíněné vykonávání není dostupné v Thumb/Thumb-2.

## Praktický příklad - if-else konstrukce

```
int max(int a, int b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

## ARM assembler s podmíněným vykonáváním

```
max:  
    cmp r0, r1           ; Compare a and b  
    movgt r0, r0         ; if (a > b) return a (NOP)  
    movle r0, r1        ; else return b  
    bx lr
```

**Pozor:** Podmíněné vykonávání není dostupné v Thumb/Thumb-2.

## 6. Calling Conventions - AAPCS

## Využití a účel registrů

```
R0–R3   : Argument/result registers (caller-saved)
R4–R11  : Variable registers (callee-saved)
R12 (IP): Intra-procedure-call scratch register
R13 (SP): Stack pointer
R14 (LR): Link register
R15 (PC): Program counter
```

### Pravidla pro předávání argumentů

1. První 4 argumenty → R0, R1, R2, R3
2. Další argumenty → Stack (FIFO pořadí)
3. Return value → R0 (32-bit), R0+R1 (64-bit)
4. Struct return → R0 obsahuje adresu
5. Stack alignment → Musí být zarovnan na 8 bajtů při volání funkce

## Zarovnání zásobníku

AAPCS vyžaduje, aby Stack Pointer (SP) byl zarovnán na 8 bajtů při vstupu do veřejné funkce:

```
// Správně – SP zarovnán na 8 bajtů
push {r4, lr}           // 8 bajtů (4+4)
sub sp, sp, #8         // alokace lokálních proměnných

// Špatně – SP není zarovnán
push {r4}              // pouze 4 bajty – porušení AAPCS
```

Kompilátor automaticky zajišťuje zarovnání, ale při ručním psaní ASM je třeba dávat pozor.

## Příklad - volání funkce

```
int complex_func(int a, int b, int c, int d, int e, int f)
{
    return a + b + c + d + e + f;
}

int main()
{
    return complex_func(1, 2, 3, 4, 5, 6);
}
```

## Implementace v assembleru

```
main:
    push {lr}           ; Save link register
    mov r0, #1          ; a = 1
    mov r1, #2          ; b = 2
    mov r2, #3          ; c = 3
    mov r3, #4          ; d = 4

    ; e=5, f=6 jdou na stack
    mov r4, #6          ; f = 6
    push {r4}           ; Push f
    mov r4, #5          ; e = 5
    push {r4}           ; Push e

    bl complex_func    ; Call function
    add sp, sp, #8      ; Clean up stack (2 args * 4 bytes)
    pop {pc}            ; Return
```

## Implementace v assembleru

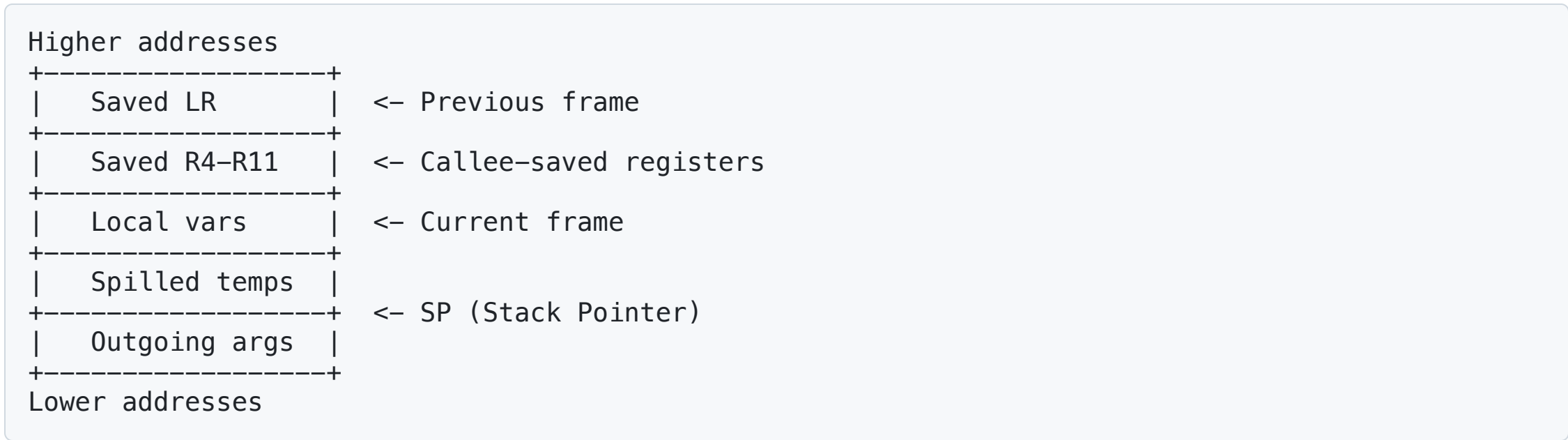
```
complex_func:
; R0=a, R1=b, R2=c, R3=d
; Stack: [SP+0]=e, [SP+4]=f
ldr r4, [sp, #0]      ; Load e
ldr r5, [sp, #4]      ; Load f

add r0, r0, r1        ; a + b
add r0, r0, r2        ; + c
add r0, r0, r3        ; + d
add r0, r0, r4        ; + e
add r0, r0, r5        ; + f
bx lr                 ; Return (result in R0)
```

# Organizace zásobníku

## Rámeček zásobníku

Rámecem nazýváme část zásobníku, která je určena pro aktuálně vykonávanou funkci.



## Prolog / Epilog funkce:

```
function:
; PROLOGUE
push {r4-r11, lr}      ; Uložení callee-saved registrů
sub sp, sp, #local_size ; Alokace lokálních registrů

; FUNCTION BODY
; ... function implementation ...

; EPILOGUE
add sp, sp, #local_size ; Dealokace lokálních proměnných
pop {r4-r11, pc}       ; Obnova registrů a return
```

## Příklad s lokálními proměnnými:

```
int factorial(int n) {
    int result = 1;
    int i;
    for (i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

```
factorial:
    push {r4, r5, lr}      ; Save registers
    sub sp, sp, #8        ; Allocate locals (result, i)

    mov r1, #1             ; result = 1
    str r1, [sp, #4]       ; Store result on stack
    mov r2, #1             ; i = 1
    str r2, [sp, #0]       ; Store i on stack
```

```

loop:
    ldr r2, [sp, #0]        ; Load i
    cmp r2, r0             ; Compare i with n
    bgt end_loop           ; if (i > n) exit loop

    ldr r1, [sp, #4]        ; Load result
    mul r1, r1, r2          ; result *= i
    str r1, [sp, #4]        ; Store result

    add r2, r2, #1         ; i++
    str r2, [sp, #0]        ; Store i
    b loop

end_loop:
    ldr r0, [sp, #4]        ; Load result for return
    add sp, sp, #8          ; Deallocate locals
    pop {r4, r5, pc}       ; Restore and return

```