

CS397 Mid-Term Course Evaluation

Link for Course Evaluation:

https://forms.office.com/Pages/ResponsePage.aspx?id=3VSy2p7TUEKbcYcxg_A4TQojxkhBNvhDnnEgsZH9wt9UNjhIMzk0WjJZT0FHUUhNUFVXN1dKVVNHTSQlQCNjPTEkJUAAjdD1n

Memory Map & Bit-Banding

Bit-Band Alias Regions

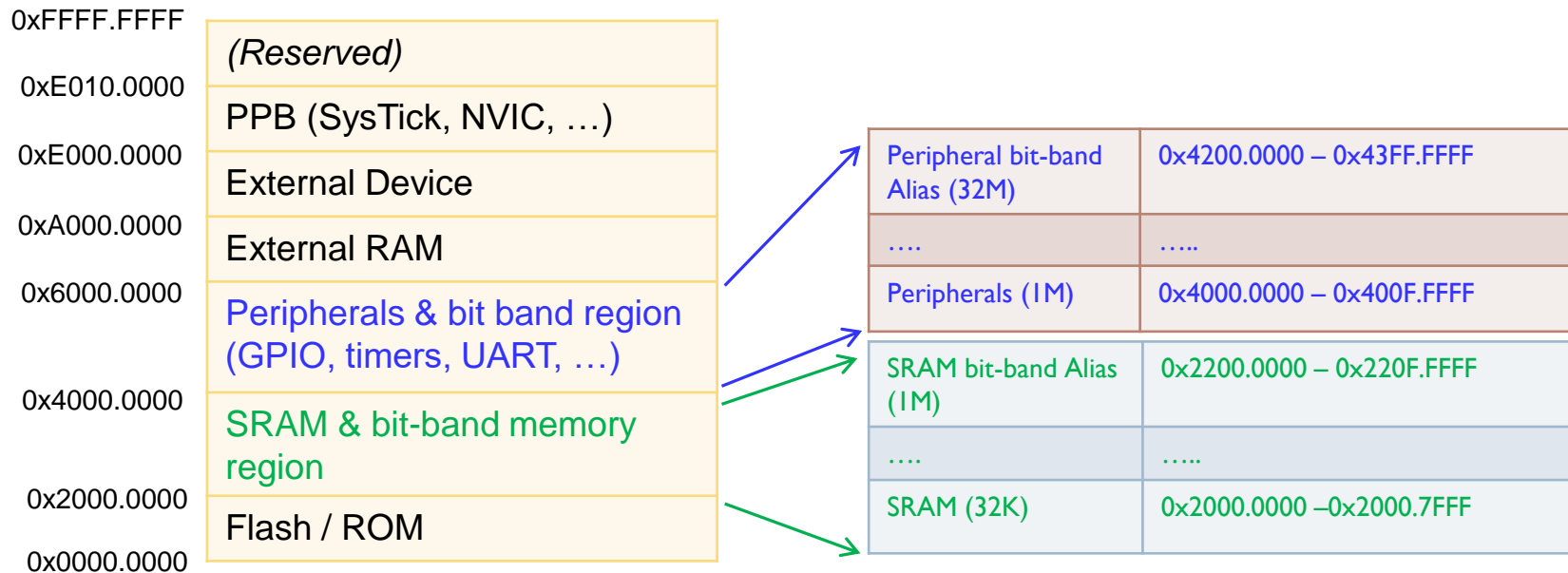
ARM Memory Map

- Total addressable memory = 4G bytes (2^{32} bytes).
- Flash ROM (program) memory starts from address 0x0000.0000.
- SRAM space starts from 0x2000.0000.
- Peripheral I/O space starts from 0x4000.000 to 0x5FFF.FFFF.
 - GPIO, Timer, UART, ADC ... registers.
- Memory map includes **TWO bit-band regions** at:
 - Lowest 1Mbyte of SRAM,
 - Lowest 1Mbyte of peripheral memory.

0xFFFF.FFFF	(Reserved)
0xE010.0000	PPB (SysTick, NVIC, ...)
0xE000.0000	External Device
0xA000.0000	External RAM
0x6000.0000	Peripherals & bit band region (GPIO, timers, UART, ...)
0x4000.0000	SRAM & bit-band memory region
0x2000.0000	Flash / ROM
0x0000.0000	

Source: Tiva TM4C123GH6PM Data Sheet (spmu376e.pdf, p95)

Bit-Banding (SRAM & Peripherals)



Bit-banding is a feature of ARM Cortex-M3 & Cortex-M4 CPUs.

It maps each bit in the bit-band region to a word-aligned address in the bit-band alias region.

- **SRAM:** Each bit in SRAM Bit-band region (32K bytes) is mapped to an entire word (32 bits) in a 2nd memory region called the Bit-band Alias region (1M bytes = 32K bytes × 32bits) at 0x2200.0000.
- **Peripheral:** Peripheral addresses are bit-banded to Bit-band Alias region at 0x4200.0000.

Bit-Banding – How it Works

[SRAM Bit-band alias memory] – starts from 0x2200.0000

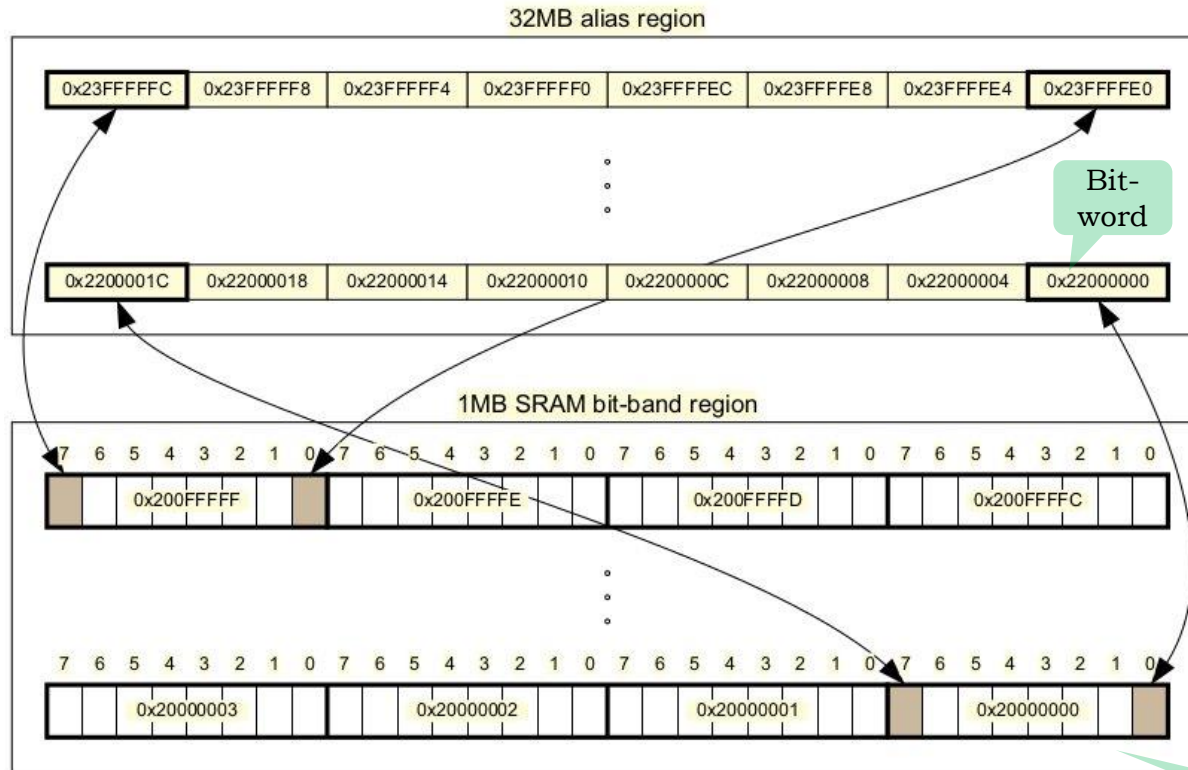


Table 6.8 Remapping of Bit-Band Addresses in the SRAM Region

Bit-Band Region	Aliased Equivalent
0x20000000 bit [0]	0x22000000 bit [0]
0x20000000 bit [1]	0x22000004 bit [0]
0x20000000 bit [2]	0x22000008 bit [0]
...	...
0x20000000 bit [31]	0x2200007C bit [0]
0x20000004 bit [0]	0x22000080 bit [0]
...	...
0x20000004 bit [31]	0x220000FC bit [0]
...	...
0x200FFFC bit [31]	0x23FFFFFFC bit [0]

[SRAM] - starts from 0x2000.0000

$$\text{bit_word_offset} = (\text{byte_offset} \times 32) + (\text{bit_no} \times 4)$$

$$\text{bit_word_addr} = \text{bit_band_base} + \text{bit_word_offset}$$

SRAM base addr

Source:
<https://developer.arm.com/documentation/ddi0337/e/Memory-Map/Bit-banding>

Bit-Banding – How it Works

$$bit_word_offset = (byte_offset \times 32) + (bit_no \times 4)$$

$$bit_word_addr = bit_band_base + bit_word_offset$$

Examples:

- ✓ For SRAM address = 0x2000.0001, bit 3, we have: byte_offset = 1, bit_no = 3.
 - Bit 3 will be alias-mapped to: $0x2200.0000 + (1 * 32) + (3 * 4) = 0x2200.002C$.
- ✓ For SRAM address = 0x2000.0002, bit 5:
 - Bit-band alias address = $0x2200.0000 + (2 * 32) + (5 * 4) = 0x2200.0054$
- ✓ For SRAM address = 0x2000.7FFF, bit 7:
 - Bit-band alias address = $0x2200.0000 + (0x7FFF * 32) + (7 * 4) = 0x220F.FFFC$

Bit-Banding (SRAM & Peripherals)

- **Read-Modify-Write** : (3-step process)
 - In a conventional CPU operation, changing the value of a bit in a memory location is a 3-step process:
 - read the current value into a temporary register
 - modify the value through masking
 - write result back to memory location.
- **Bit-banding**:
 - A Write-cycle to a word in the Bit-band alias memory region performs a write to the corresponding bit in the Bit-band region.
 - Reading a word in the Bit-band alias region returns the value of the corresponding bit in the Bit-band region.
 - Above operations are requires less number of instruction cycles and are **atomic** and thus eliminate race conditions.

Bit-Banding – WRITE Operation

Writing to 0x2000.0000, bit 2

Without Bit-Band

Read 0x20000000 to register

Set bit 2 in register

Write register to 0x20000000

With Bit-Band

Write 1 to 0x22000008

Without Bit-Band

```
LDR    R0,=0x20000000 ; Setup address
LDR    R1, [R0]        ; Read
ORR.W  R1, #0x4        ; Modify bit
STR    R1, [R0] ; Write back result
```

With Bit-Band

```
LDR    R0,=0x22000008 ; Setup address
MOV    R1, #1         ; Setup data
STR    R1, [R0]       ; Write
```

[Source:](#) The definitive guide to ARM Cortex-M3 and Cortex-M4 processors, 3rd edition, Joseph Yiu

Bit-Banding – READ Operation

Reading status of 0x2000.0000, bit 2

Without Bit-Band

Read 0x20000000 to register

Shift bit 2 to LSB and mask other bits

With Bit-Band

Read from 0x22000008

Without Bit-Band

```
LDR    R0,=0x20000000 ; Setup address
LDR     R1, [R0]        ; Read
UBFX.W R1, R1, #2, #1 ; Extract bit[2]
```

With Bit-Band

```
LDR    R0,=0x22000008 ; Setup address
LDR     R1, [R0]        ; Read
```

[Source](#): The definitive guide to ARM Cortex-M3 and Cortex-M4 processors, 3rd edition, Joseph Yiu

Comparing Clock Cycles:

Bit-Band Alias vs Read-Write-Modify

```
LED_RED_ON_BB(); // with bit-band alias access (see slide 13)
```

0x00000964	2001	MOVS	r0,#0x01	1 clk
0x00000966	4916	LDR	r1,[pc,#88] ; @0x000009C0	2 clk
0x00000968	6008	STR	r0,[r1,#0x00]	2 clk
Total:				5 clk

```
LED_RED_ON(); // GPIOF->DATA |= 1<<1 - used in Labs
```

0x0000098C	480E	LDR	r0,[pc,#56] ; @0x000009C8	2 clk
0x0000098E	6800	LDR	r0,[r0,#0x00]	2 clk
0x00000990	F0400002	ORR	r0,r0,#0x02	1 clk
0x00000994	490D	LDR	r1,[pc,#52] ; @0x000009CC	2 clk
0x00000996	F8C103FC	STR	r0,[r1,#0x3FC]	2 clk
Total:				9 clk

Bit-band access utilizes less clock cycles than normal Read-Modify-Write access
- more efficient!

Accessing Bit-Band Alias Memory

- Writing a word (32-bits) to the alias region updates a **single bit** in the bit-band region.
- When writing to the alias regions **bit 0** of the 32 bit word is used to set the value at the bit-banding region. – **only bit 0 is used.**
- Reading from the alias address will return the value from the bit-band region in bit 0 and the other bits will be cleared.
- Bits [31:1] of the alias word have NO effect on the bit-band bit.
 - Writing 0x01 to alias address has the same effect as writing 0xFF → bit 0 = 1.
 - Writing 0x00 to alias address has the same effect as writing 0x0E → bit 0 = 0.
 - When reading a word in the alias region, 0x0000.0000 indicates that the targeted bit in the bit-band region is clear & 0x0000.0001 indicates that the targeted bit in the bit-band region is set.

Accessing Bit-Band Alias (SRAM)

```
#define SRAM_BASE          0x20000000 // SRAM base address
#define BITBAND_SRAM_BASE  0x22000000

#define BITBAND_SRAM(addr,bit_no) ((BITBAND_SRAM_BASE+(addr-SRAM_BASE)*32+(bit_no*4)))

#define SRAM1_ADDR          0x20004000 // SRAM memory location
#define SRAM1_ADDRP *((volatile unsigned int *) (SRAM1_ADDR))

#define SRAM1_B7 *((volatile unsigned int *) (BITBAND_SRAM(SRAM1_ADDR,7)))
#define SRAM1_B0 *((volatile unsigned int *) (BITBAND_SRAM(SRAM1_ADDR,0)))

/** program statements */
static volatile uint32_t tmp;
SRAM1_ADDRP = 0x0; // initialize SRAM1 location to 0
SRAM1_B7 = 1;      // set SRAM1 bit 7
tmp = SRAM1_ADDRP; // tmp = 0x80
SRAM1_B0 = 1;      // set SRAM1 bit 0
tmp = SRAM1_ADDRP; // tmp = 0x81
SRAM1_B0 = 0xFF;   //
tmp = SRAM1_ADDRP; // tmp = 0x81
```

Accessing Bit-Band Alias (Peripheral)

```
#define PERI_BASE    0x40000000    // Cortex-M4 peripheral base addr
#define GPIOF_DATA  0x400253FC    // GPIOF data register addr; Note: 0xFC, not 0x00
#define GPIOA_DATA  0x400043FC    // GPIOA data register addr

#define BITBAND_PERI_BASE    0x42000000 // start of peripheral bit-band alias region
#define BITBAND_PERI(addr,bit_no) ((BITBAND_PERI_BASE+(addr-PERI_BASE)*32+(bit_no*4)))

// define GPIO port accesses
#define GPIOF_B1 *((volatile unsigned int *) (BITBAND_PERI(GPIOF_DATA,1))) // RED LED
#define GPIOF_B2 *((volatile unsigned int *) (BITBAND_PERI(GPIOF_DATA,2))) // BLUE LED
#define GPIOA_B4 *((volatile unsigned int *) (BITBAND_PERI(GPIOA_DATA,4))) // BUZZER

#define LED_RED_ON_BB() (GPIOF_B1 = 1)
#define LED_RED_OFF_BB() (GPIOF_B1 = 0)

/**  program statements  */
LED_RED_ON_BB();    // Red LED on
GPIOF_B1 = 1;        // Red LED on (same as above)
GPIOF_B2 = 0;        // Blue LED off
GPIOA_B4 = 1;        // Buzzer on
```

Bit-Banding

Why would we want to use Bit-Banding?

- In bit-banding, a write to a word in the Alias region also performs a write to the corresponding bit in the bit-band region.
- Similarly, reading a word in the Alias region gives the value of the corresponding bit in the bit-band region.
- Bit-band operations allows for fast (efficient) read/write of individual bits.
- Bit-band operations are **atomic**.
 - Operation cannot be interrupt by other activities. Once it has started, it will go to completion.
 - Thus, the operation is deterministic and there is no data conflict.

Bit-Banding

Potential uses of Bit-Banding

- Serial data transfers from GPIO ports to serial devices.
- Simplify branch decisions: if a branch is carried out based on a single bit status in a register.
 - Instead of, (*without bit-banding*)
 - Reading the whole register
 - Masking the unwanted bits
 - Comparing & branching
 - We could, (*with bit-banding*)
 - Reading the status bit via it's bit-band alias (gets a 0 or 1)
 - Comparing & branching.

Bit-Band vs Bit-Bang

In computer engineering and electrical engineering, **bit banging** is slang for any method of data transmission that employs software as a substitute for dedicated hardware to generate transmitted signals or process received signals. Software directly sets and samples the states of GPIOs (e.g., pins on a microcontroller), and is responsible for meeting all timing requirements and protocol sequencing of the signals.

In contrast to bit banging, dedicated hardware (e.g., UART, SPI interface) satisfies these requirements and, if necessary, provides a data buffer to relax software timing requirements. Bit banging can be implemented at very low cost, and is commonly used in embedded systems.

Bit banging allows a device to implement different protocols with minimal or no hardware changes. In some cases, bit banging is made feasible by newer, faster processors because more recent hardware operates much more quickly than hardware did when standard communications protocols were created.

Source: https://en.wikipedia.org/wiki/Bit_banging

In short,

- Bit-banging implements a protocol through SW and GPIOs; is flexible and suited for proprietary, non-standard protocols and low-cost solutions.
- Bit-band is a feature of the Cortex-M architecture to improve efficiency (execution time) of bit-operations.

Performance-related Topics

Performance-Related: Conditional Branches

- Conditional Branches based on multiple or complex conditions can be optimized.
- Example of program to extract prime numbers from 1 to 31:

```
void is_a_prime_number(unsigned int i)
{
    if ((i==2) || (i==3) || (i==5) || (i==7) ||
        (i==11) || (i==13) || (i==17) || (i==19) ||
        (i==23) || (i==29) || (i==31)) {
        printf ("- %d\n", i);
    }
    return;
}
```

C code

Assembly code

```
is_a_prime_number
0x080002ca: 2802      .(  CMP    r0,#2
0x080002cc: d013      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002ce: 2803      .(  CMP    r0,#3
0x080002d0: d011      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002d2: 2805      .(  CMP    r0,#5
0x080002d4: d00f      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002d6: 2807      .(  CMP    r0,#7
0x080002d8: d00d      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002da: 280b      .(  CMP    r0,#0xb
0x080002dc: d00b      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002de: 280d      .(  CMP    r0,#0xd
0x080002e0: d009      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002e2: 2811      .(  CMP    r0,#0x11
0x080002e4: d007      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002e6: 2813      .(  CMP    r0,#0x13
0x080002e8: d005      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002ea: 2817      .(  CMP    r0,#0x17
0x080002ec: d003      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002ee: 281d      .(  CMP    r0,#0x1d
0x080002f0: d001      .. BEQ    0x80002f6 ; branch_simple + 44
0x080002f2: 281f      .(  CMP    r0,#0x1f
0x080002f4: d103      .. BNE    0x80002fe ; branch_simple + 52
0x080002f6: 4601      .F  MOV    r1,r0
0x080002f8: a01e      .. ADR    r0,{pc}+0x7c ; 0x8000374
0x080002fa: f000b93d ..=. B.W   __2printf ; 0x8000578
0x080002fe: 4770 pG   BX    lr
```

Performance-Related: Conditional Branches

- Prime numbers program can be optimized/shortened through encoding the prime numbers by its bit position & using it for branch decision.

C code

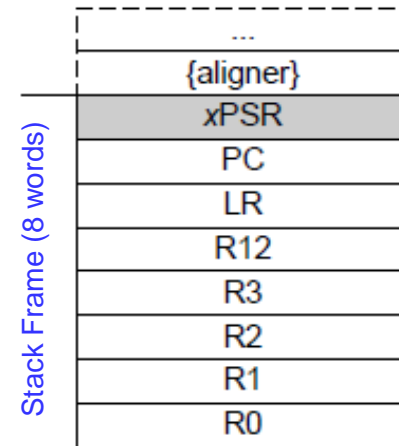
```
void branch_method1(unsigned int i)
{
    /* Bit pattern is
       31:0 - 1010 0000 1000 1010 0010 1000 1010 1100 = 0xA08A28AC */
    if ((1<<i) & (0xA08A28AC)) {
        printf ("- %d\n", i);
    }
    return;
}
```

Assembly
code

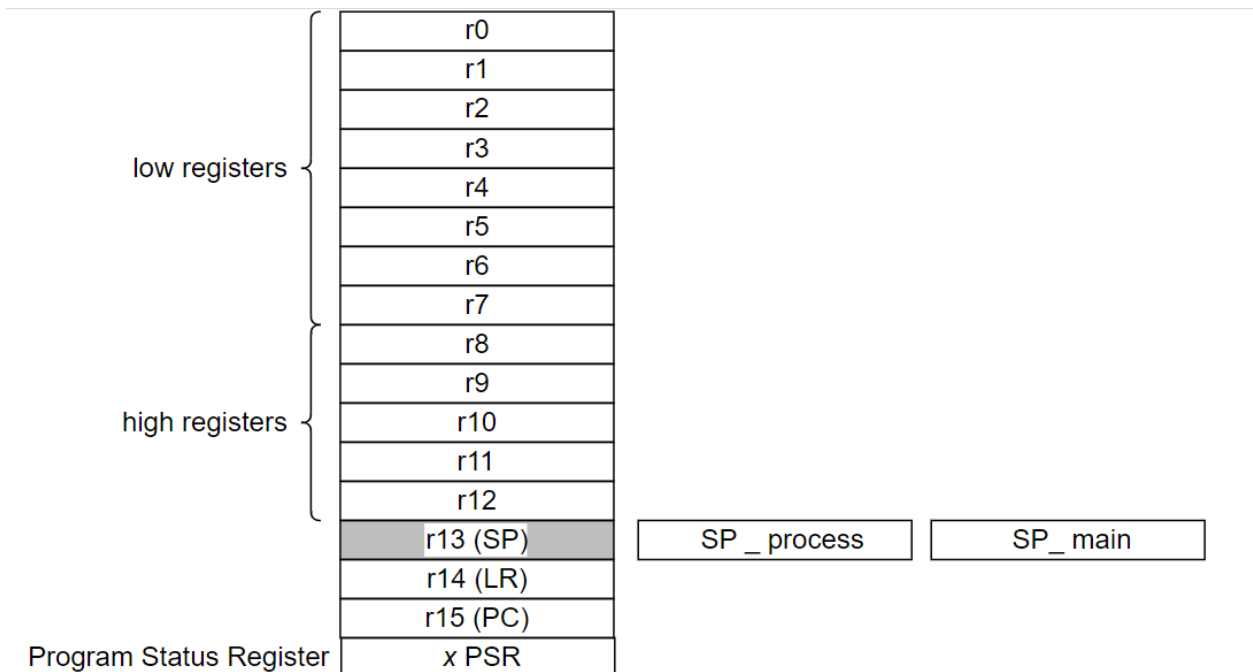
```
branch_method1
0x080003c0: 2101      .!    MOVS r1,#1
0x080003c2: 4a24      $J    LDR r2,[pc,#144] ; [0x8000454] = 0xa08a28ac
0x080003c4: 4081      .@    LSLS r1,r1,r0
0x080003c6: 4211      .B    TST r1,r2
0x080003c8: bf08      ..    IT EQ
0x080003ca: 4770      pG    BXEQ lr
0x080003cc: 4601      .F    MOV r1,r0
0x080003ce: a01f      ..    ADR r0,{pc}+0x7e ; 0x800044c
0x080003d0: f000b950 ..P. B.W __2printf ; 0x8000674
```

Performance-Related: Function Parameters

- **AAPCS**: Procedure Call Standard for ARM Architecture.
 - Part of the larger 'Application Binary Interface (ABI) for ARM Architecture.
- AAPCS defines how subroutines can be separately written, separately compiled & separately assembled, and work together.
- Implication of AAPCS:
 - C function can modify R0 to R3, R12, R14 (LR), PSR (these are part of ARM Stack Frame).
 - If R4 to R11 is used, these registers should be saved to stack memory.
- R0 to R3, R12, LR, PSR are called '**caller-saved registers**' – program code that calls the subroutine needs to save these registers (in the stack) before execution is transferred to subroutine.
- R4 to R11 are called '**callee-saved registers**' – the subroutine needs to save & restore these registers.



Performance-Related: Function Parameters



Full Set of ARM Registers.

Performance-Related: Function Parameters

- A function/subroutine uses R0 to R3 for passing parameters & R0 & R1 are used to pass the return result.
- Implication:
 - Passing up to 4 word-sized parameters can be done very efficiently as they can all be placed in registers.
 - If a function requires more than 4 parameters, there is no extra registers for use and the parameters are placed on the stack (by the subroutine).
 - Extra instructions are required; program is less efficient.
- For performance optimization:
 - Limit function/subroutine parameters to maximum of 4.
 - If more than 4 parameters are needed, consider defining a data structure and pass the pointer to the data structure.

Performance-Related: Function Parameters

Function with 2 parameters

```
uint32_t function2 (uint32_t a, uint32_t b)
{
    uint32_t tmp;
    tmp=a+b;
    return(tmp);
}
```

```
MOV        r2,r0
ADDS       r0,r2,r1
BX         lr
```


- Parameters **a** stored in R0, **b** in R1.

Function with 6 parameters

```
uint32_t function6 (uint32_t a, uint32_t b,
uint32_t c, uint32_t d, uint32_t e, uint32_t f)
{
    uint32_t tmp;
    tmp=a+b;
    return(tmp);
}
```

Pushed to Stack
by subroutine

```
PUSH      {r4-r7,lr}
MOV        r4,r0
LDRD       r5,r6,[sp,#0x14]
ADDS       r0,r4,r1
POP       {r4-r7,pc}
```



- Parameters **e** stored in R4, **f** in R5.
- Contents of R4 to R7 pushed & popped by callee-function.

- ✓ Functions with up to 4 parameters uses R0 to R3 (part of the stack frame) to store parameters.
- ✓ Functions with than 4 parameters requires the callee-program (subroutine) to PUSH-POP the additional parameters → additional program overhead incurred.

Performance-Related: Function Parameters

- **AAPCS** rule: double-word (2×32 -bit) parameters must be passed in an even-odd (word-aligned) registers pair.
 - Double word parameter can be stored in register pairs R0:R1 or R2:R3 but not in R1:R2.
- Examples:

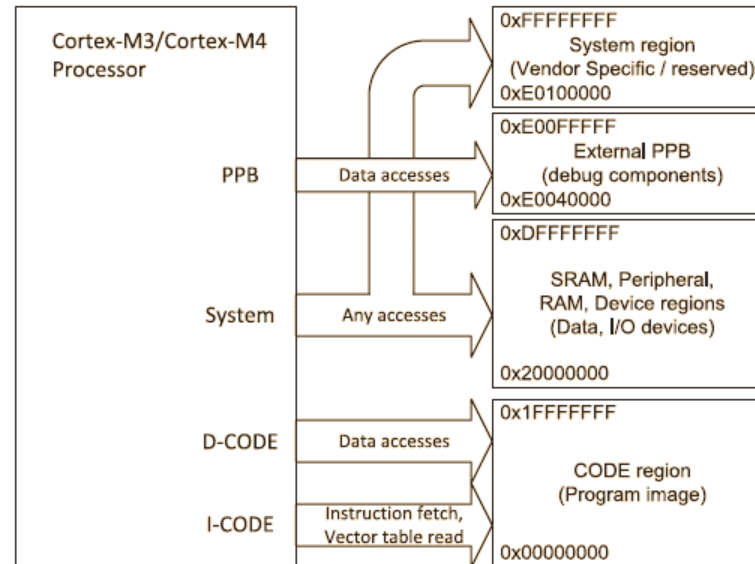
```
void function1(uint32_t a, uint64_t b, uint32_t c);
```

- Parameters:
 - a stored in R0
 - b stored in R2:R3
 - c stored in R4 → *not desirable since it is 'Callee-saved'.*

```
void function1(uint64_t b, uint32_t a, uint32_t c);
```

- Parameters:
 - b stored in R0:R1
 - a stored in R2
 - c stored in R3 → *desirable as only R0 to R3 used.*

Performance-Related: Multiple Buses



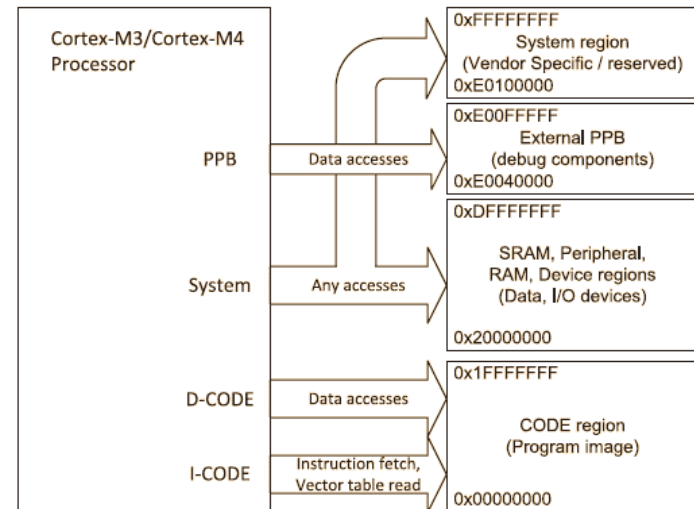
Cortex-M4 implements **parallel buses**:

- **Code region** (0x0 to 0x1FFF.FFFF): Program memory.
 - **I-Code Bus:** Instruction & vector fetch
 - **D-Code Bus:** Data & debugger accesses.
- **System Bus:** (0x2000.0000 – 0xFFFF.FFFF) – AHB/APB specifications
 - **RAM** (0x2000.000 – 0x3FFF.FFFF)
 - **Peripherals** (0x4000.000 onwards)
 - **PPB** (0xE000.000 onwards) NVIC, SysTick, FPU, ITM, ...

Performance-Related: Multiple Buses

Take advantage of multiple bus design of Cortex-M4:

- Program code execution from the CODE region (0x0000.0000 – 0x1FFF.FFFF) is faster than from system bus (0x2000.000 & above).
→ Place program code in I-Code region while data access via System bus.
- Place interrupt vector table to CODE region (lowest region at 0x0000.0000) & stack to SRAM (0x2000.0000 or above).
→ Different busses are used for program/interrupt vector & stack. Interrupt vector fetch & stacking can then be carried out at the same time with instruction fetch.



Performance Considerations:

Summary

Points to note to improve program execution:

- Program execution from the CODE region (0x0 – 0x1FFF.FFFF) is faster than from system bus (0x2000.000 & above).
→ *Program code should be in I-Code region while data access via System bus.*
- Place interrupt vector table to CODE region & stack to SRAM (0x2000.0000 or above).
→ *Different busses are used. Interrupt vector fetch & stacking can then be carried out at the same time.*
- Limit function calls to 4 input parameters or fewer.
- If there are more than 4 parameters, group data to a structure & pass pointer to data structure to reduce number of parameters.
- Use bit-banding where possible for SRAM & peripheral bit operations for faster code execution.
- Avoid using unaligned data transfers which requires more bus cycles.
 - Do not pack C data structures. (`__attribute__((packed))`)
 - Use ALIGN directive in assembly language to ensure data location is aligned.