SCC.131 – Week 15 quiz

- SCC.131 quiz questions will cover **all** material from weeks <u>10,11,12,13,14</u>.
- We will **not** use code runner questions.
- We will use C/C++ and assembly code in our questions.
- The quiz will be designed to last 30 min.

- Come to your normal lab slot.
- No calculators.
- Bring pens, we will provide paper.

Recap Questions

- How do you load the value 0x11223344 to register R0?

```
mov R0, #0x3344
movt R0, #0x1122
```

- Assume R0 = 4 , R1 = 2. What is the result value of R0, after the execution of the instruction *sub R0, R1.*

```
R0 = 2 ; R0 = R0 – R1
```

- Implement the following arithmetic operations: R1 = 16*R0 + R1/8.

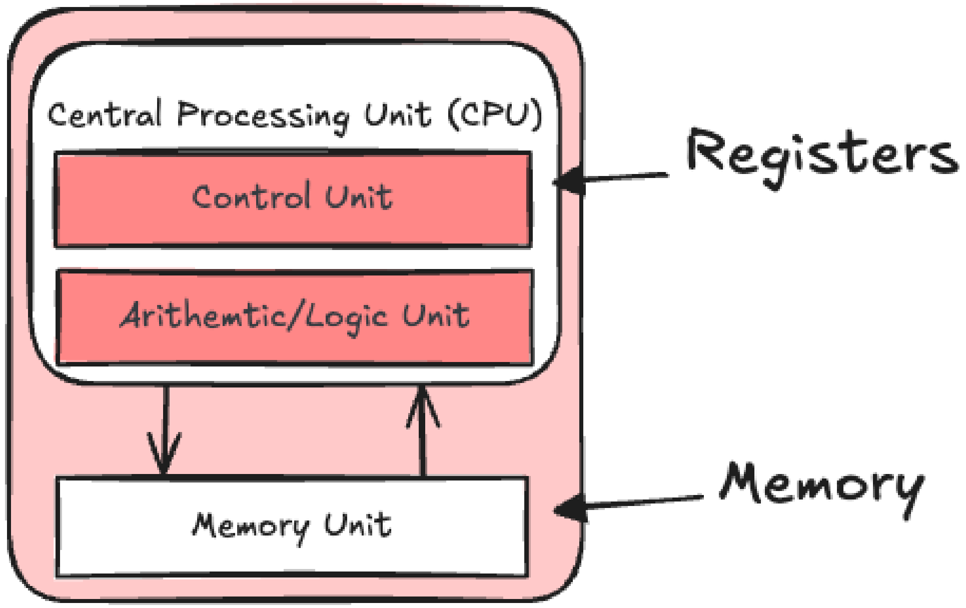
```
lsl R0, R0, 4
lsr R1, R1, 3
add R1, R0
```

ARM Assembly – Memory

Haris Rotsos

Recap and Outline

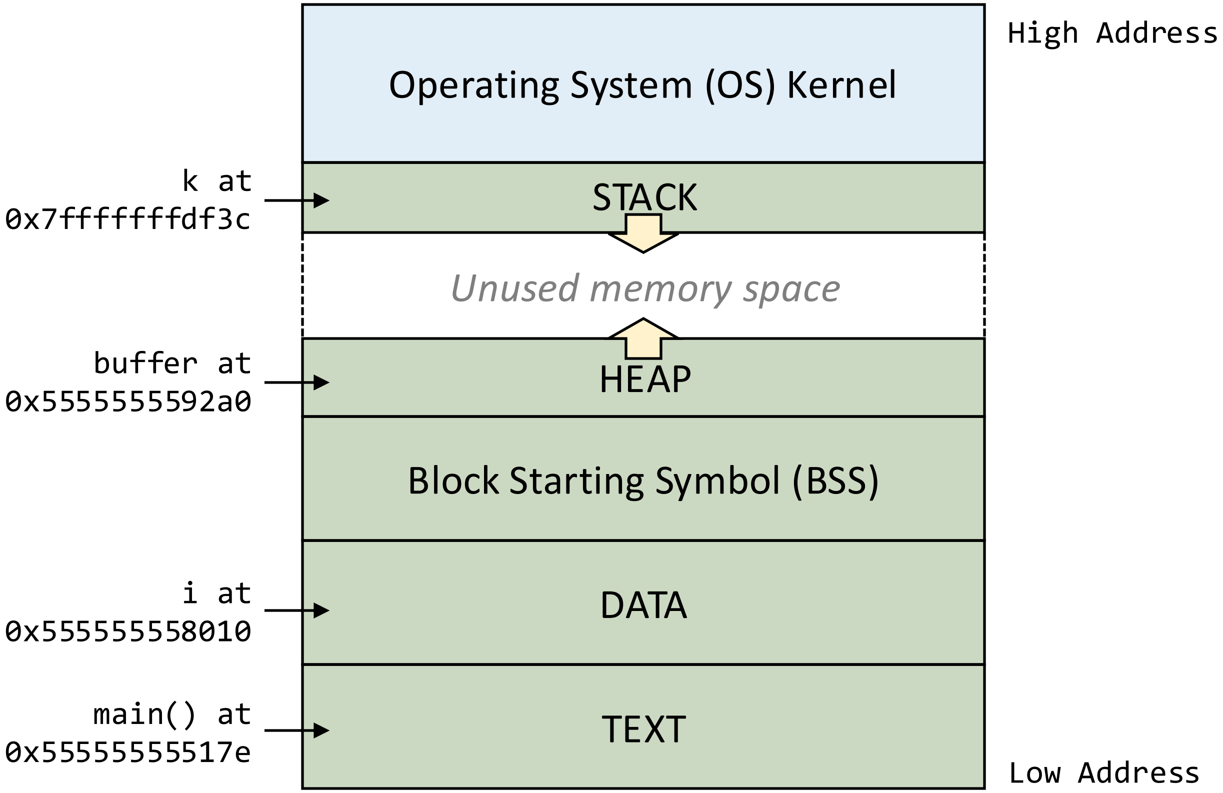
- New programming paradigm: ARM assembly.
  - Instruction Set Architecture
  - Arithmetic operations
- Based on what you know, you are able to develop programs that use 16 variables.
  - What happens if we want more?
  - How do we access and write in memory?



Memory layout – Bringing it all together

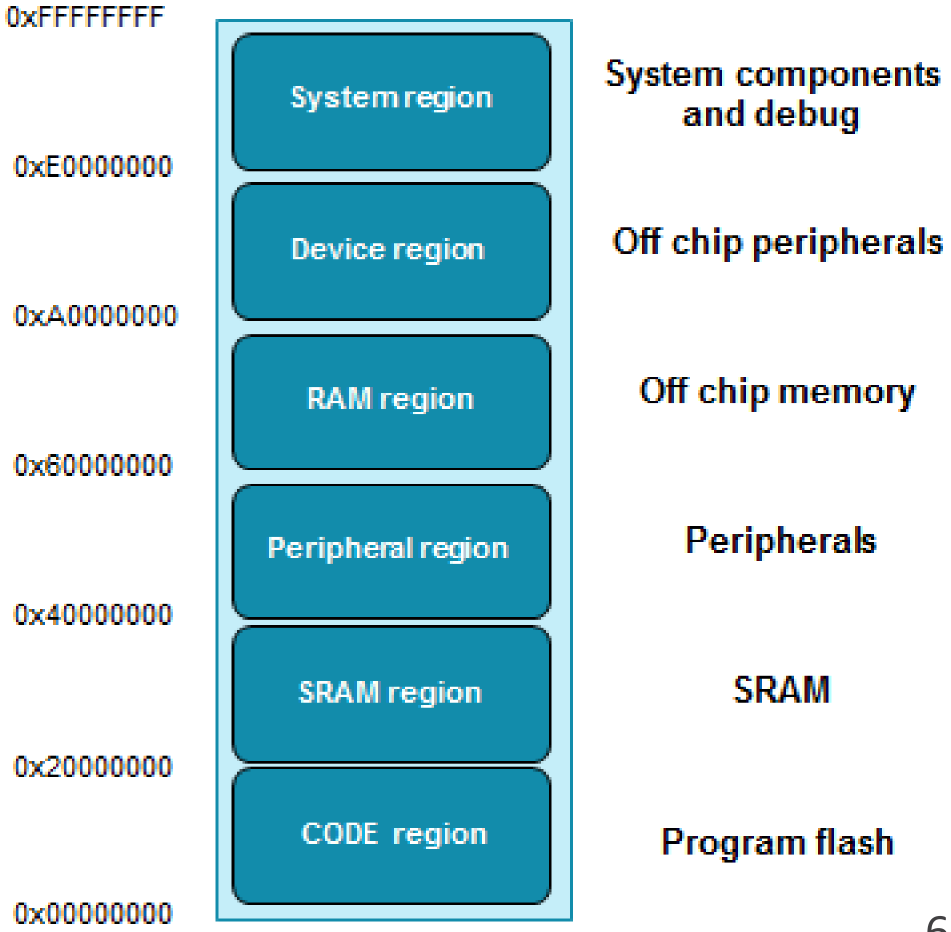
SCC131_example3.c

```
#include <stdio.h>
#include <stdlib.h>

int i = 5;

int func1(int a)
{
    int k = 5;
}

int main()
{
    char * buffer;
    buffer = (char*)malloc(i+1);
    if (buffer == NULL) exit(1);
    func1(i);
    return 0;
}
```



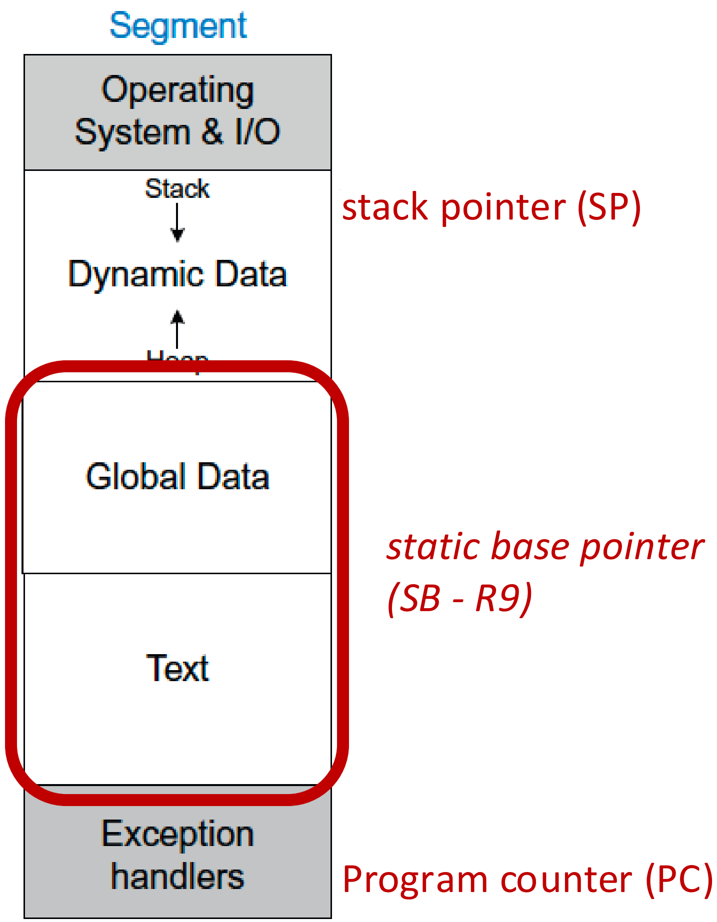
Memory Map

- The Cortex-M3 processor is a memory-mapped system.
  - Fixed linear memory map of 4 gigabytes of addressable memory space
  - Dedicated address ranges for code (code space), SRAM(memory space), external memories/devices and internal/external peripherals.
- Unaligned data access in a single core access.
  - Can read single bytes or half-world (16-bit) in a clock cycle.
  - Unaligned transfers are converted into multiple aligned transfers.
  - Remain transparent to application programmers.



Memory Layout

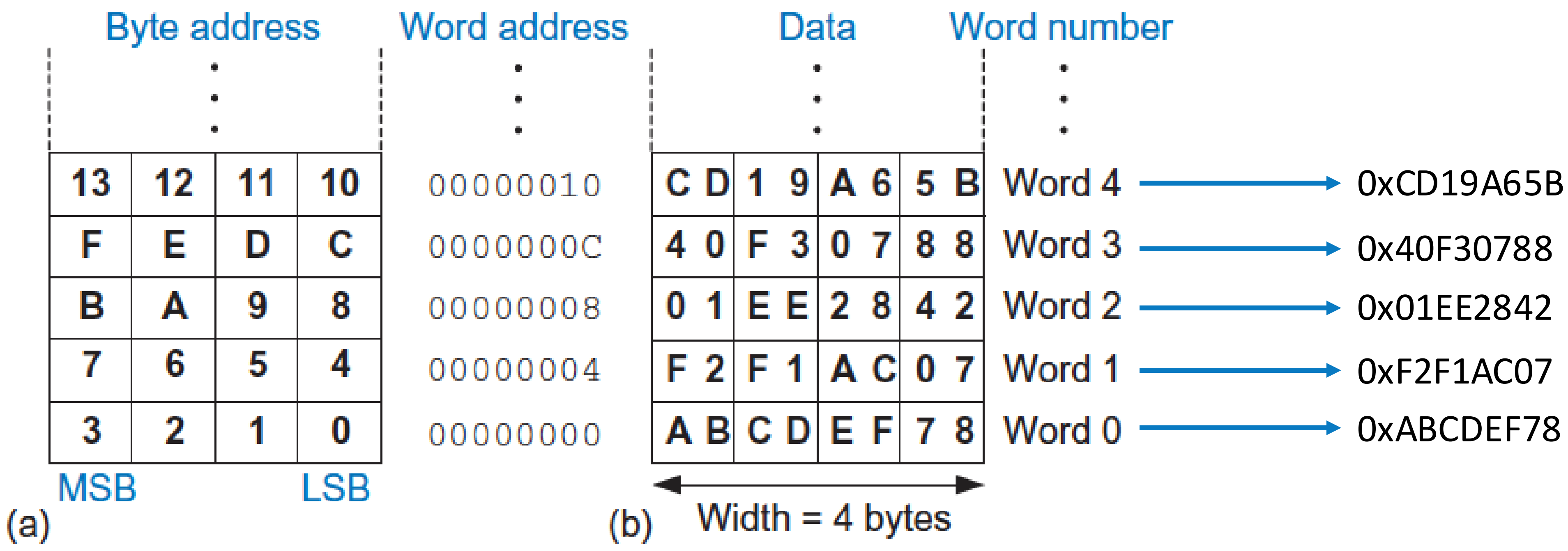
- The *stack (SRAM region)* stores all local variables and function arguments.
  - The stack pointer register, points at the bottom.
- The *heap (SRAM region)* stores data allocated during runtime (e.g. malloc).
- The *global data (SRAM region)* area stores global variables
  - ARM uses R9 as the static base pointer (SB).
- Global contains the bss area (uninitialized global data).
- The *text segment (CODE region)* stores the machine language program.
  - PC register: points to the next instruction
  - needs to be modified to change flow of program execution!



Memory in ARM ISA

ARM uses a byte-addressable memory (Each byte has a unique address.)

1 word = 32 bits = 4 bytes



Little Endian: 78EFCDAB Big Endian: ABCDEF78

Accessing Memory

```
ldr r0, [r2, #100]
```

"load register"

destination register

Base address

offset

```
str r0, [r2, #100]
```

"store register"

source register

Target Memory Address = r2 + 100

Addressing Mode

```
ldr r0, [r1, #20]
```

- *reg+const mode:* use a register and a constant to compute the read/write address.

```
ldr r0, [r1, r2]
```

- *reg+reg mode:* use the sum of 2 registers to compute the read/write address.

```
ldr r0, [r1, r2, LSL #2]
```

- *reg+reg<<scale:* shift the second register parameter and add to the first register, to compute the read/write address.

```
int R0, *R1;
R0 = R1[5]; //int array
```

```
int R0, R2, *R1;
R0 = R1[R2];
```

```
int R0, R2, *R1;
R0 = R1[R2 << 2];
```

Memory layout and directives

- `.bss`
  - Contains statically allocated variables that are declared but have not been assigned a value yet.
- `.data`
  - Contains initialized static variables, i.e., global variables and static local variables set to a value.
- .rodata
  - Contains initialized static variables that constant.
- `.text`
  - Contains the executable instructions.

Memory Types

The region to store data

optional data type to help the assembler to allocate appropriate space

```
.data
    label: .type value
```

A name to reference the data

Optional value to initialize memory

- Data type directives:
  - .word: 4 byte integer.
  - .byte: 1 byte integer.
  - .ascii: quote enclosed string.
  - .asciz: null-terminated *string.*
  - .fill repeat, size, value: fills memory with a repeated value of size bytes.
  - .zero size: fill memory with zeroes.

Memory Addresses (pseudoinstruction)

ldr r0, =label

“load register”

label

destination register

```
r0 = &label
```

Accessing Memory

- LDRB (load byte): access individual byte in memory, zero extend.
- LDRSB (load signed byte): access individual byte in memory, sign extend.
- STRB (store byte): stores the least significant byte of the 32-bit register into the specified byte address in memory.

- LDRH (load halfword): access 16-bits from memory, zero extend.
- LDRSH (load signed halfword): access 16-bits in memory, sign extend.
- STRH (store halfword): stores the least significant 16-bits of the 32-bit register into the specified byte address in memory.

Accessing Memory
One operand in Memory

```
g = h + A[8];
```

```
ldr r0,[r3, #32]
add r1,r2,r0
```

32 = 4*8

register mapping

```
g: r1
h: r2
A: r3 (int[])
temp: r0
```

r3 is a “pointer”

Accessing Memory
Two operands in Memory

A[12] = h + A[10];

```
ldr r0,[r3, #40]
add r0,r2,r0
str r0,[r3, #48]
```

<u>register mapping</u>

h: r2
A: r3 (int[])
temp: r0

Sample Code (1)

```
.section .data
    test1: .word 0x11223344
    test2: .byte 0x80332211
    test3: .ascii "hello"
```

test1 test2

- 0x20000000: 0x11 0x22 0x33 0x44 0x11 0x22 0x33 0x80
- 0x20000008: 0x68 0x65 0x6c 0x6c 0x6f 0xd0 0x20 0xcc

test3

Sample Code (2)

```
ldr r3, =test1       @ Base address of test1 in r3
ldr r0, [r3]         @ copy the value in register r0
mov r0, #0x2211      @ load value 0x44332211
movt r0, #0x4433     @ into r0 and store in test1
str r0, [r3]

ldr r3, =test2       @ Base address of test2 in r3
mov r0, #0x2211
movt r0, #0x8033
str r0, [r3, #0]     @ Store 0x80332211 in test2
ldrb r0, [r3, #3]    @ r0 = 128 (0x80)
ldrsb r0, [r3, #2]   @ r0 = 51 (0x33)
ldrsb r0, [r3, #3]   @ r0 = -128 (0xffffff80)
```

```
ldr r3, =test3     @ Store string "hello"
mov r0, ‘h’        @ in array test3
strb r0, [r3, #0]
mov r0, 'e'
strb r0, [r3, #1]
mov r0, 'l'
strb r0, [r3, #2]
mov r0, 'l'
strb r0, [r3, #3]
mov r0, 'o'
strb r0, [r3, #4]
```

Conclusions

Memory Access

Memory Layout

Next: branching