MA4832

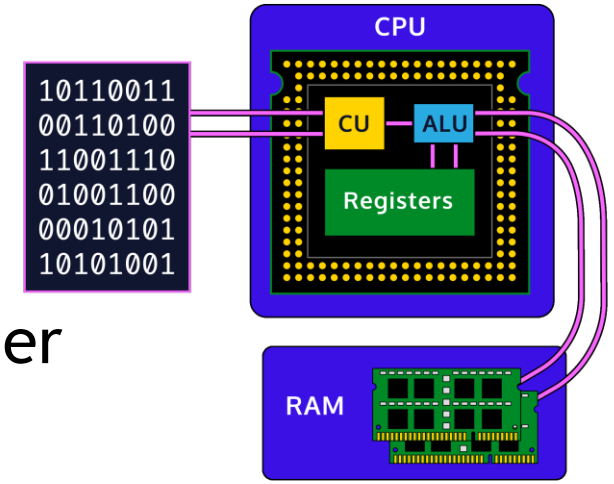
Microprocessor Systems

Xie Ming, PhD (France)

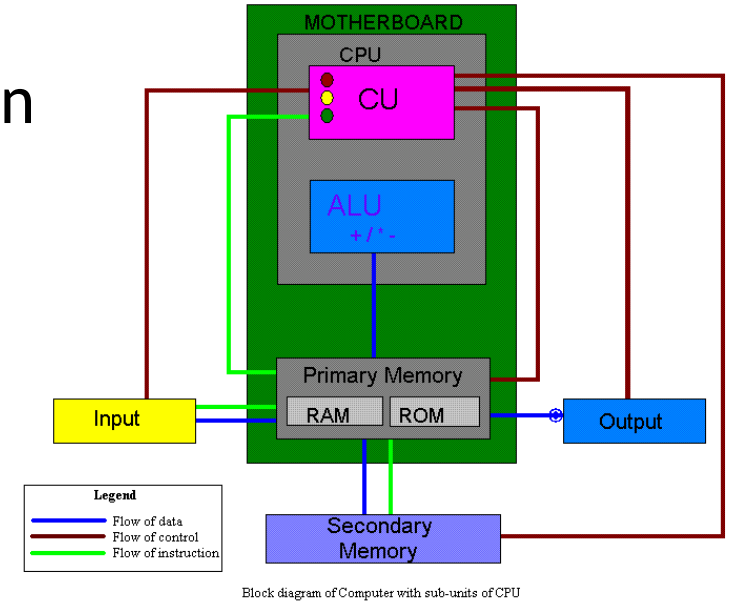
mmxie@ntu.edu.sg

http://personal.ntu.edu.sg/mmxie

Outline



- Lecture 1: Basics of ARM Microcontroller
- Lecture 2: ARM's Memories
- Lecture 3: ARM's Data Representation
- Lecture 4: ARM's Programming
- Lecture 5: ARM's Data Input/Output
- Lecture 6: ARM's Data Processing



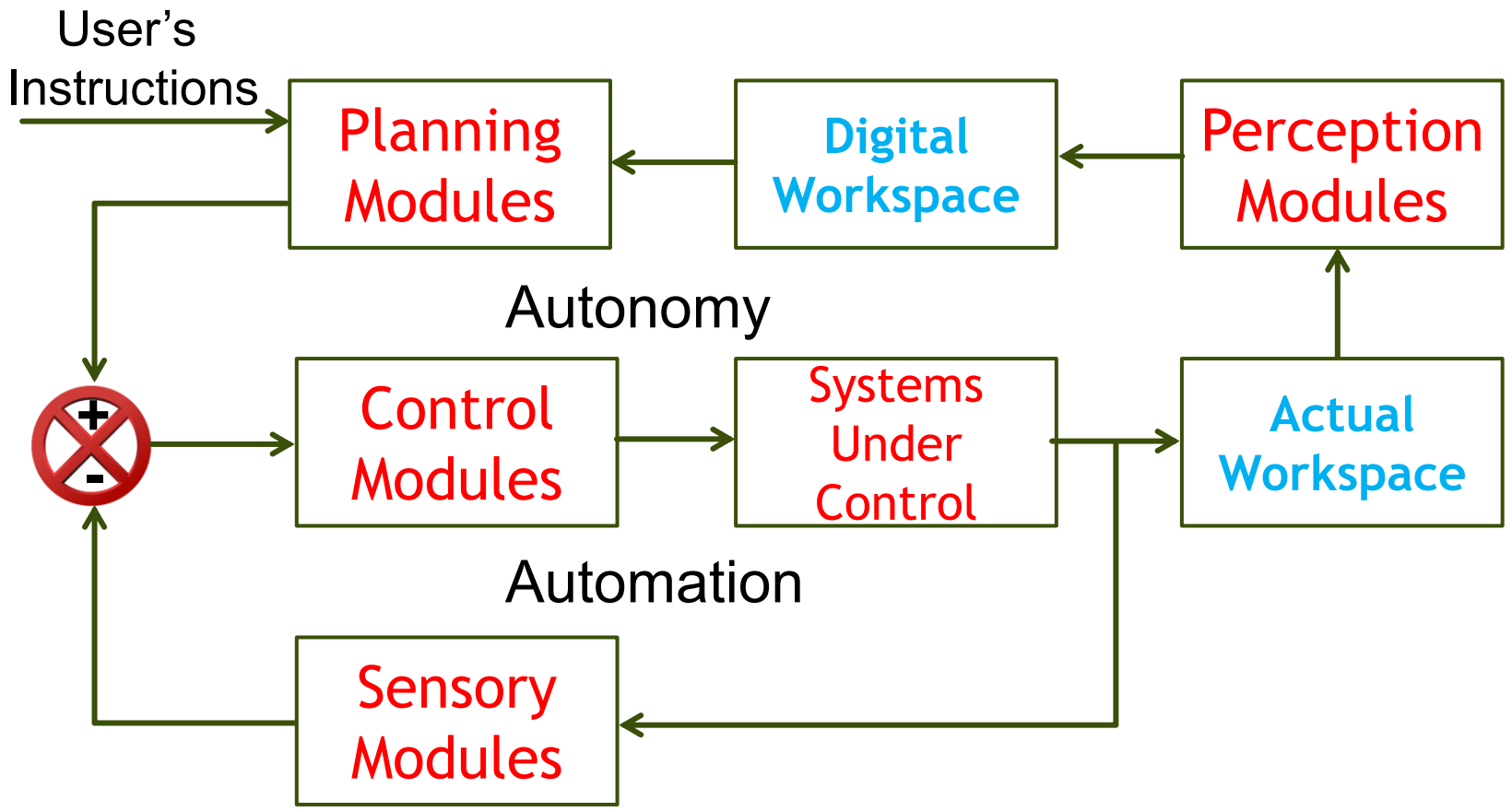
Block diagram of Computer with sub-units of CPU

Remember your mission as MAE undergraduates …

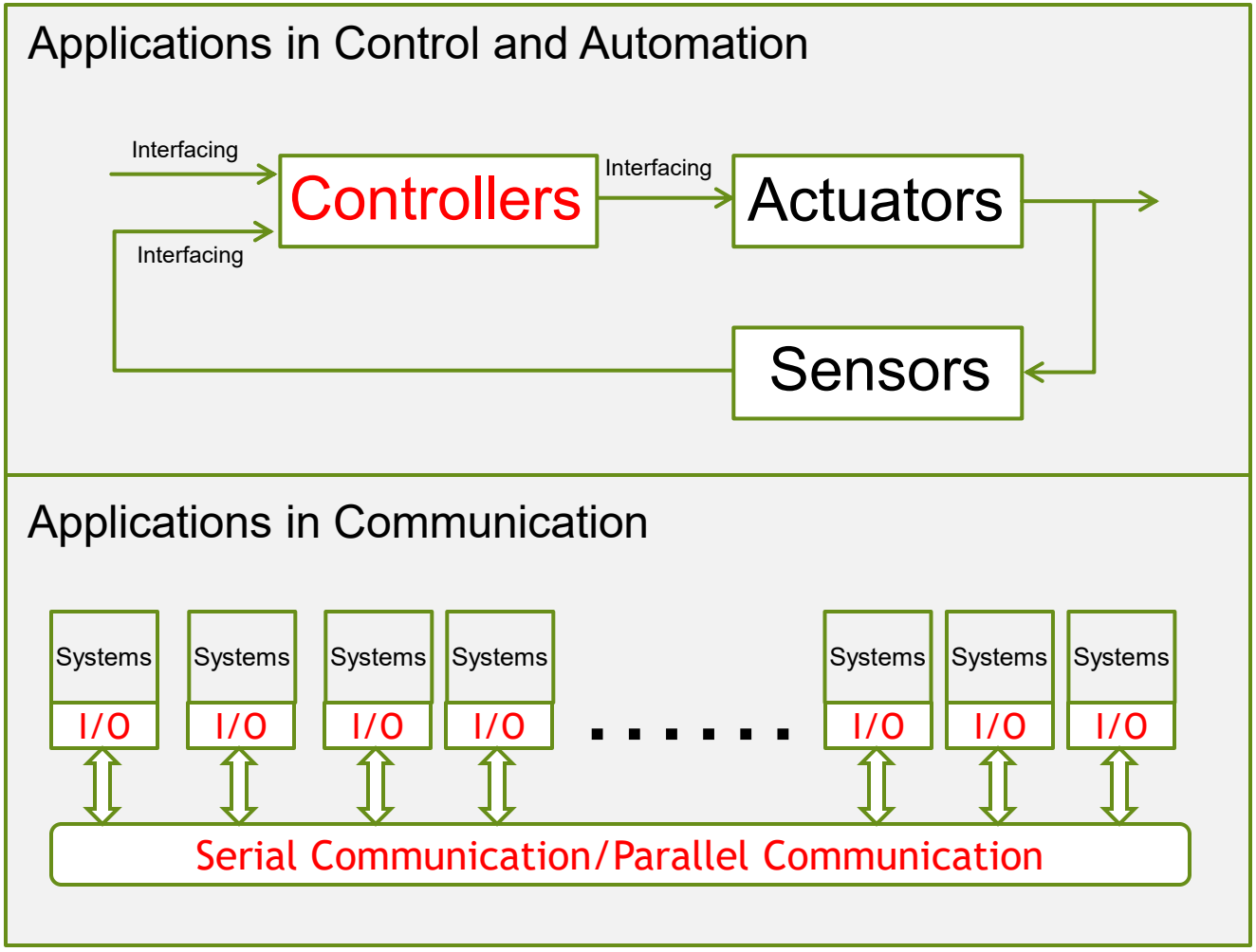
- You are here to grow your knowledge and skills so as to be able to design machines with controllable behaviors and hopefully in some intelligent ways.

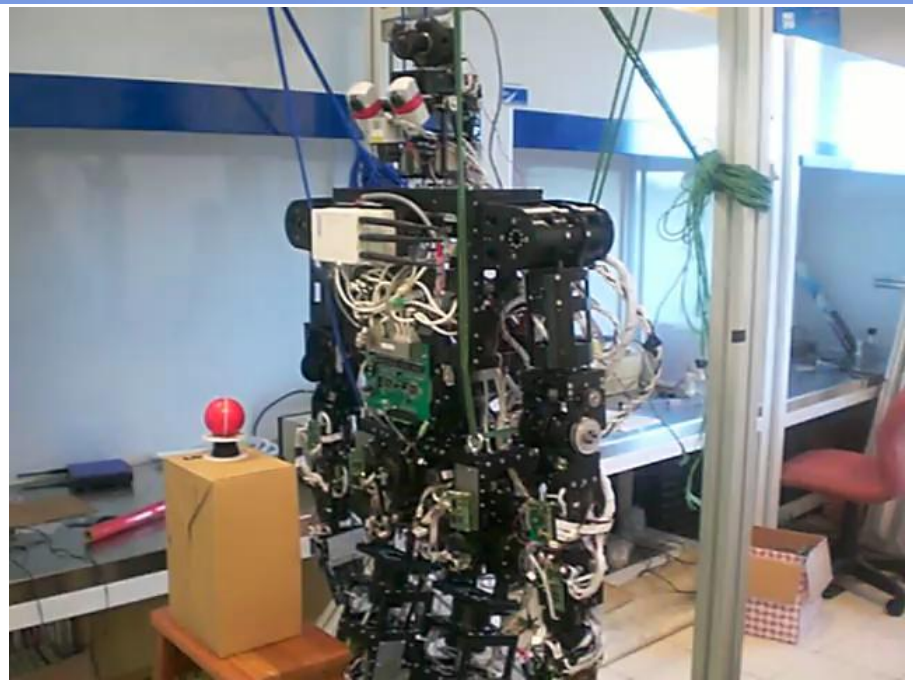
How to fulfill your mission?

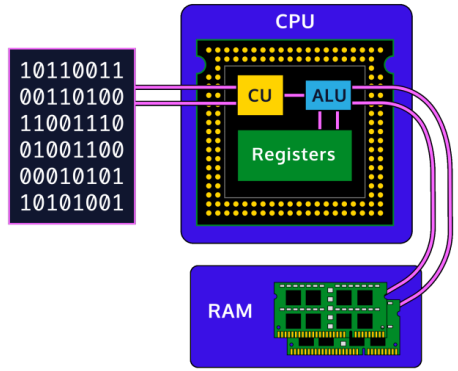
- To apply learnt knowledge and skills into the implementation of the following universal blueprint underlying all the intelligent machines or systems.

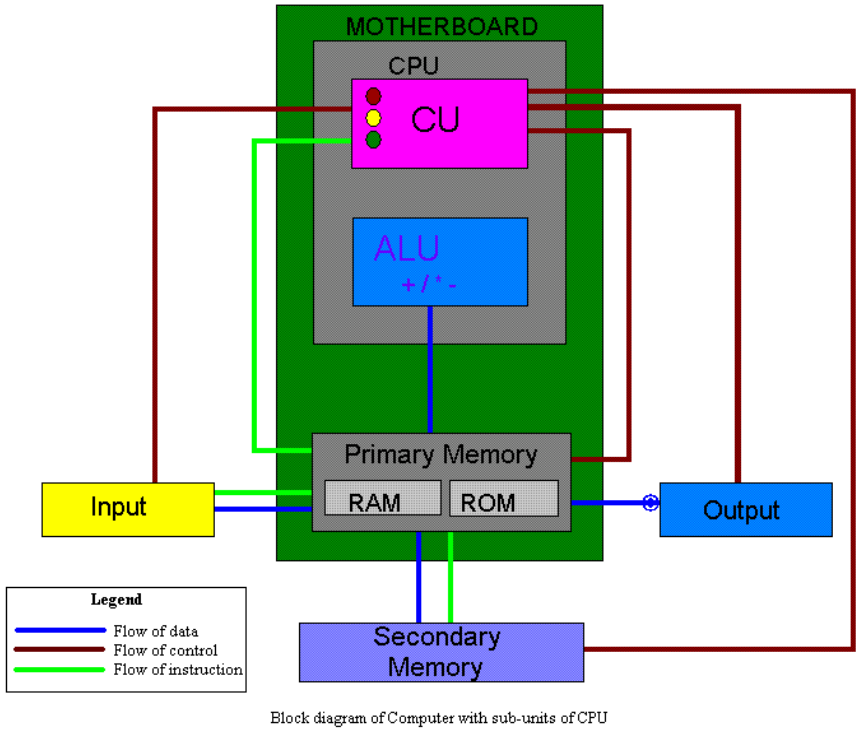


Why to study?


Applications in Control and Automation
Interfacing
Interfacing
Controllers
Interfacing
Actuators
Sensors
Applications in Communication
Systems
I/O
Systems
I/O
Systems
I/O
Systems
I/O
Systems
I/O
Systems
I/O
Systems
I/O
Serial Communication/Parallel Communication



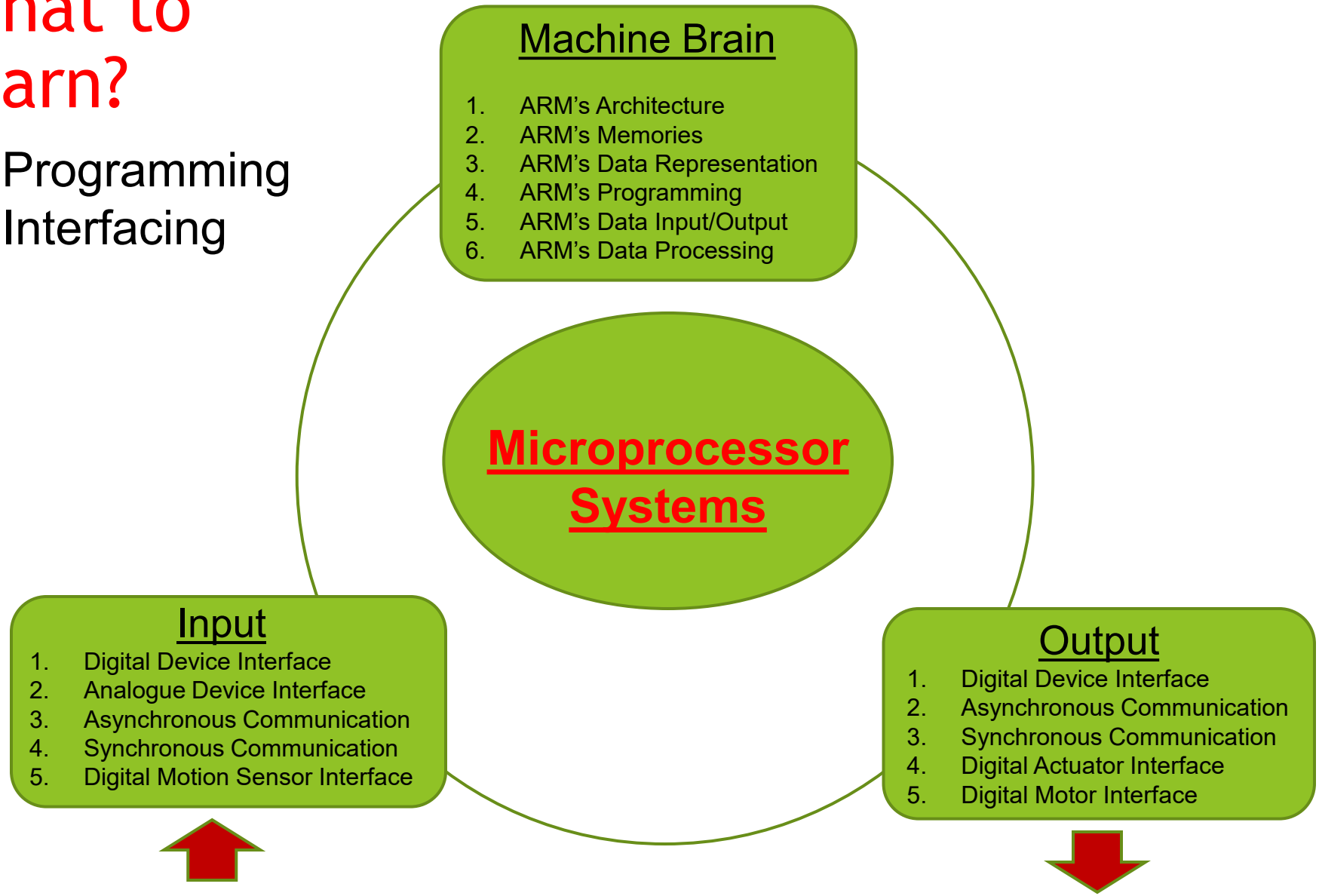




Block diagram of Computer with sub-units of CPU

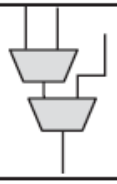
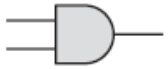
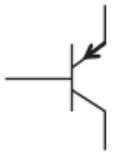
What to learn?

- Programming
- Interfacing



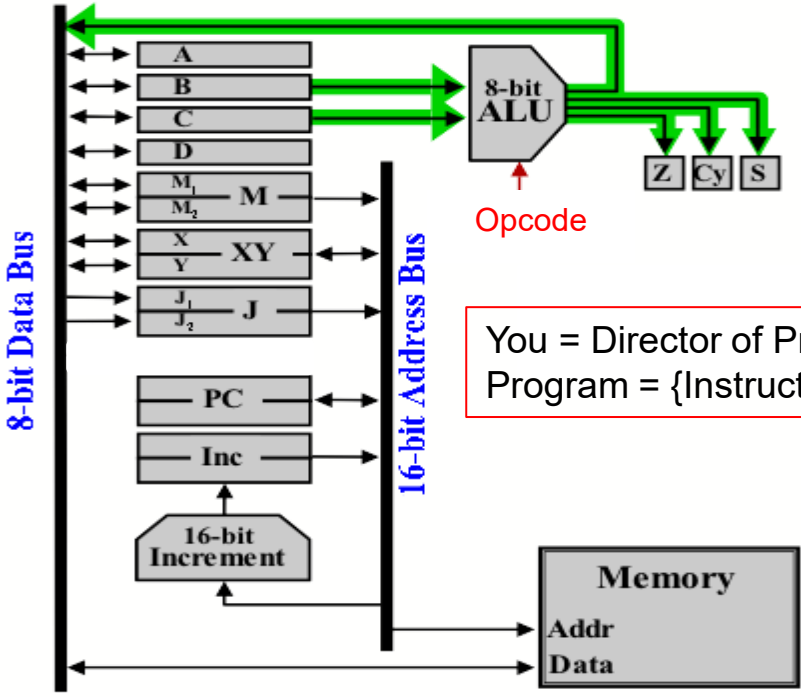
What is your role?

- Data = {Values, Symbols, Addresses, Instructions}
- Instructions = {Op Code + Addresses + Value/Symbol}

Applications/OS	
Languages	C++, Java
ISA	`EOR r3,r2,r1` `BEQ Table`
Microarchitecture	
Gates	
Transistors	

Algorithms or Solutions

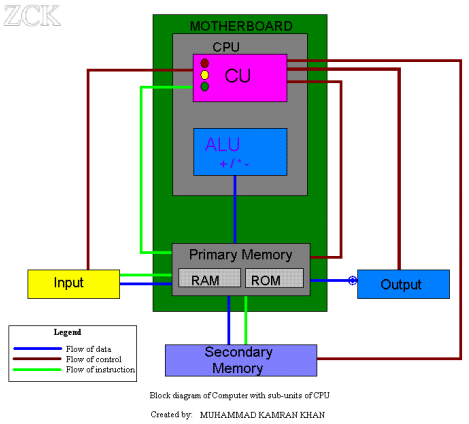
Problems to be solved



You = Director of Program
Program = {Instructions}

Memory = {Address + Data/Instruction}

How to learn?



- To understand data flows inside a microcontroller.
- To translate your solutions into data flows.
- To pay attention to <memory address> and <memory data>.
  - Memory Address: Address Label/Name and Address Value.
  - Memory Data: Data Label/Name and Data Value.

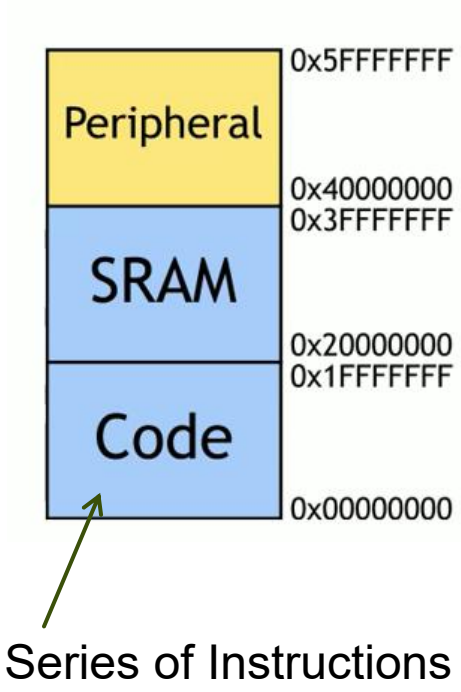
Memory Content

Data	Memory Address

- To pay attention to <memory address> and <memory code>.
  - Memory Address: Address Label/Name and Address Value.
  - Memory Code: Code Label/Name and Code Value.

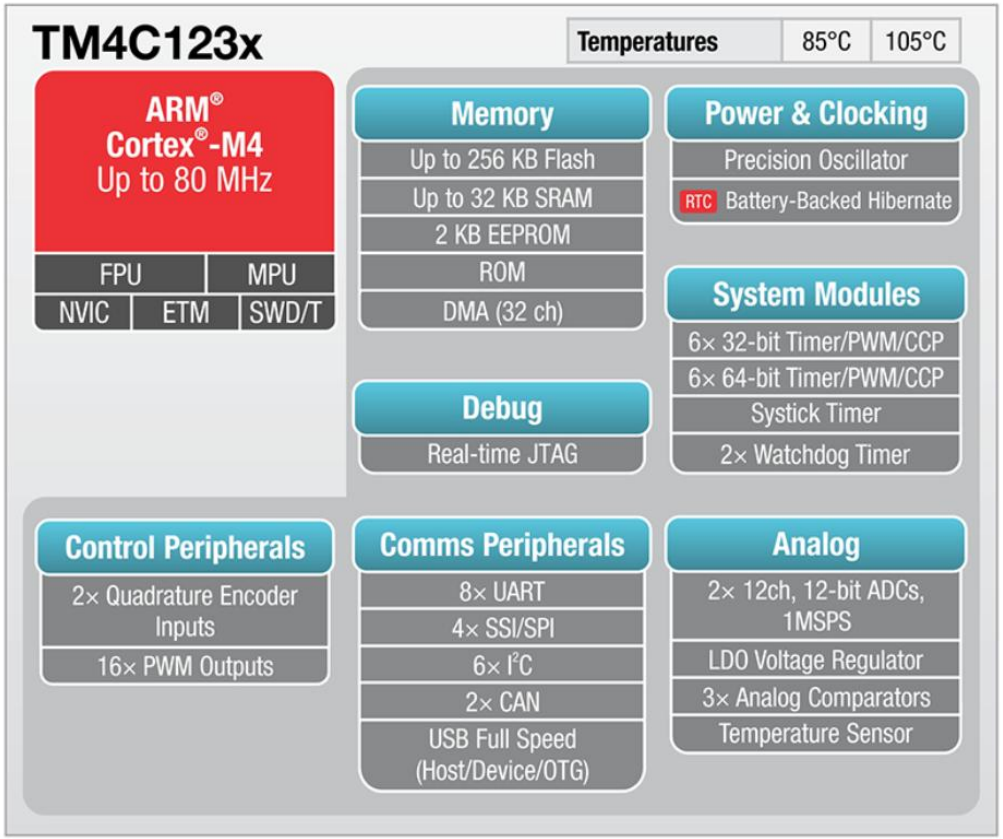
Memory Content

Instruction	Memory Address



Example of Using I/O Modules

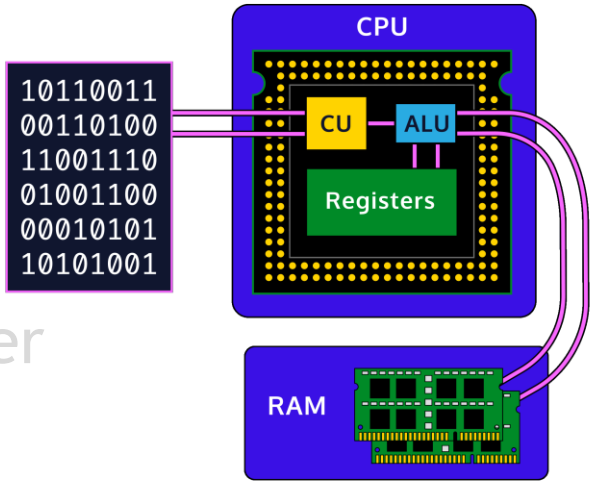
- Configure Control Registers
- Clear/Monitor Status Registers
- Read/Write Data Registers
- Instructions:
  - MOV <address of destination>, <source of value>
  - LDR <address of destination>, <source of value>
  - STR <source of value>, <address of destination>



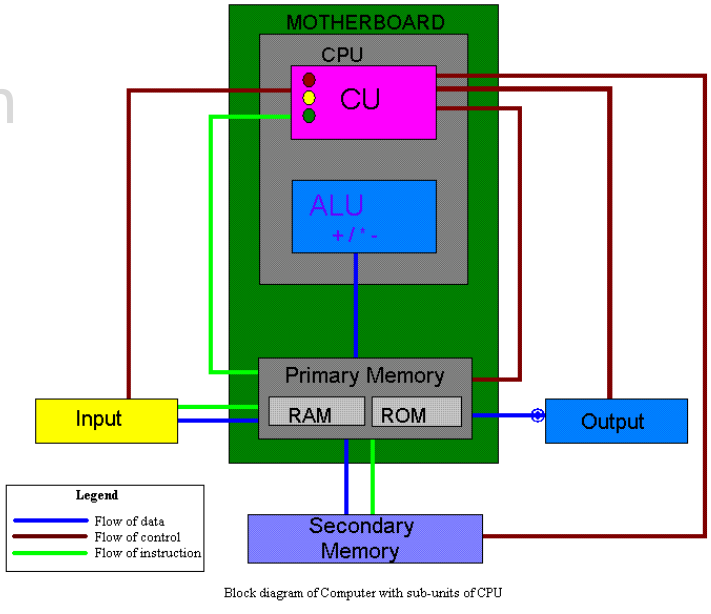

```
MOV   R0, #0x11
MOV   R1, #2560
MVN   R2, #4
MOVW  R3, #0xC0DE
MOVT  R3, #0xFEED
MOV   R4, R1
```

Word = 2 Bytes

Today's Lecture …



- Lecture 1: Basics of ARM Microcontroller
- Lecture 2: ARM's Memories
- Lecture 3: ARM's Data Representation
- Lecture 4: ARM's Programming
- Lecture 5: ARM's Data Input/Output
- Lecture 6: ARM's Data Processing



Block diagram of Computer with sub-units of CPU

NANYANG TECHNOLOGICAL UNIVERSITY

School of Mechanical & Aerospace Engineering

Design, Machine, Control and Intelligence

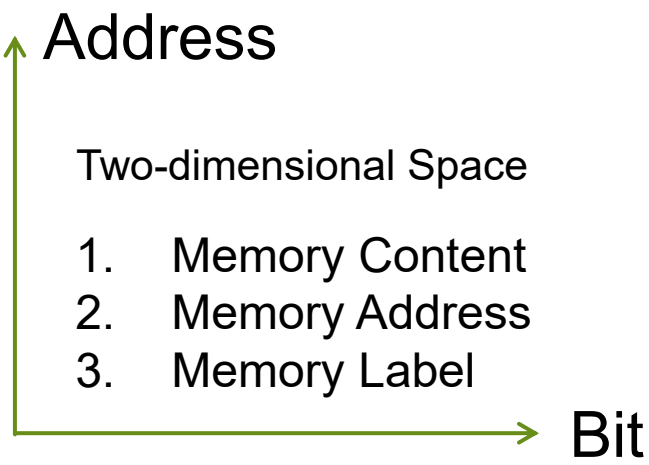
MA4832

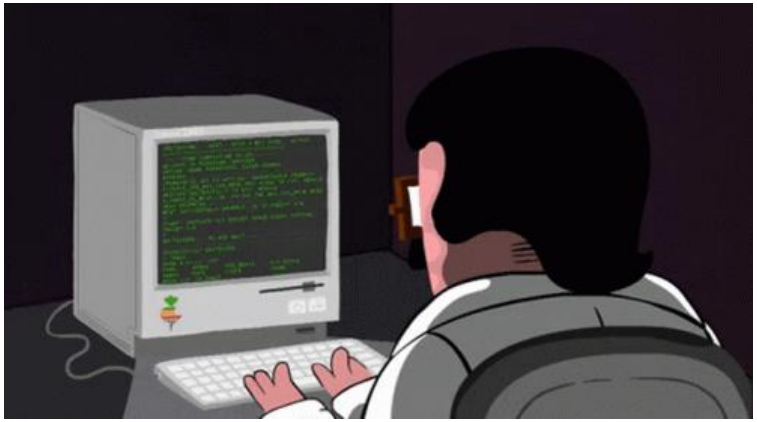
ARM's Programming

Xie Ming, PhD (France)

http://personal.ntu.edu.sg/mmxie

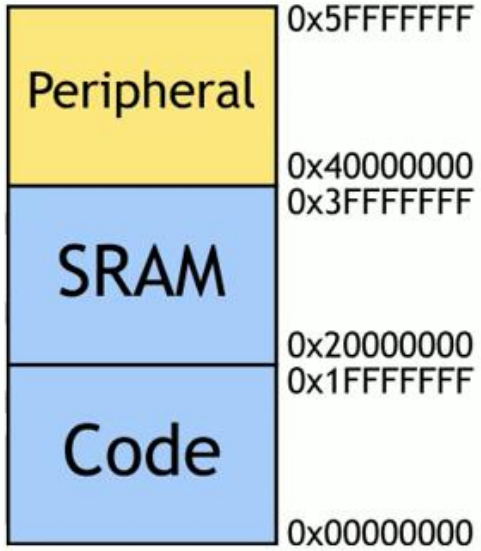
Outline



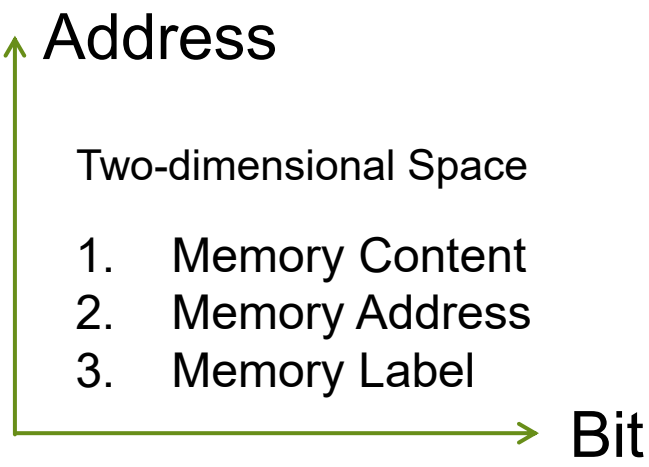


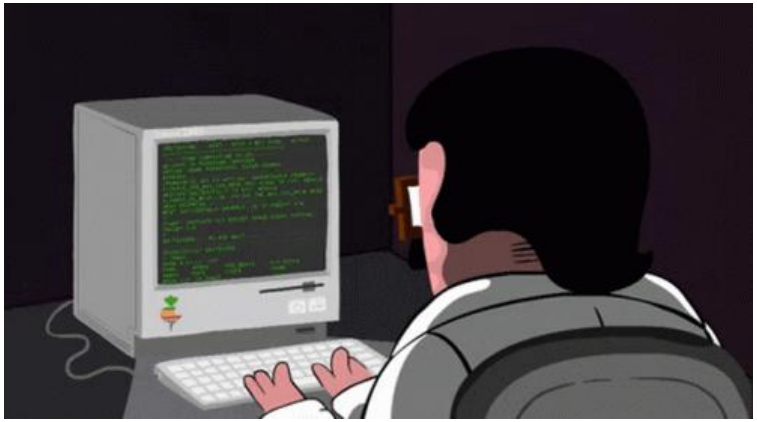
You = Director of Program
Program = {Instructions}

- Nature of Programming
- Planning of Computations in Programming
- Allocation of Memory in Programming
- ARM Programming Tools



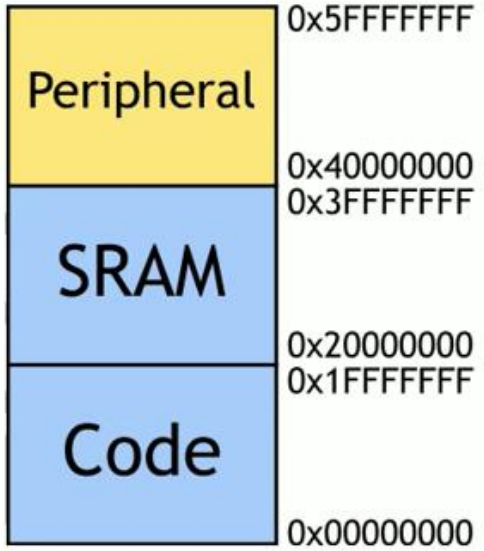
Outline





You = Director of Program
Program = {Instructions}

- Nature of Programming
- Planning of Computations in Programming
- Allocation of Memory in Programming
- ARM Programming Tools



Scenario of Knowledge Discovery and Transfer …

Authentic AI

"We are living inside an ocean of signals"

Keys of AI

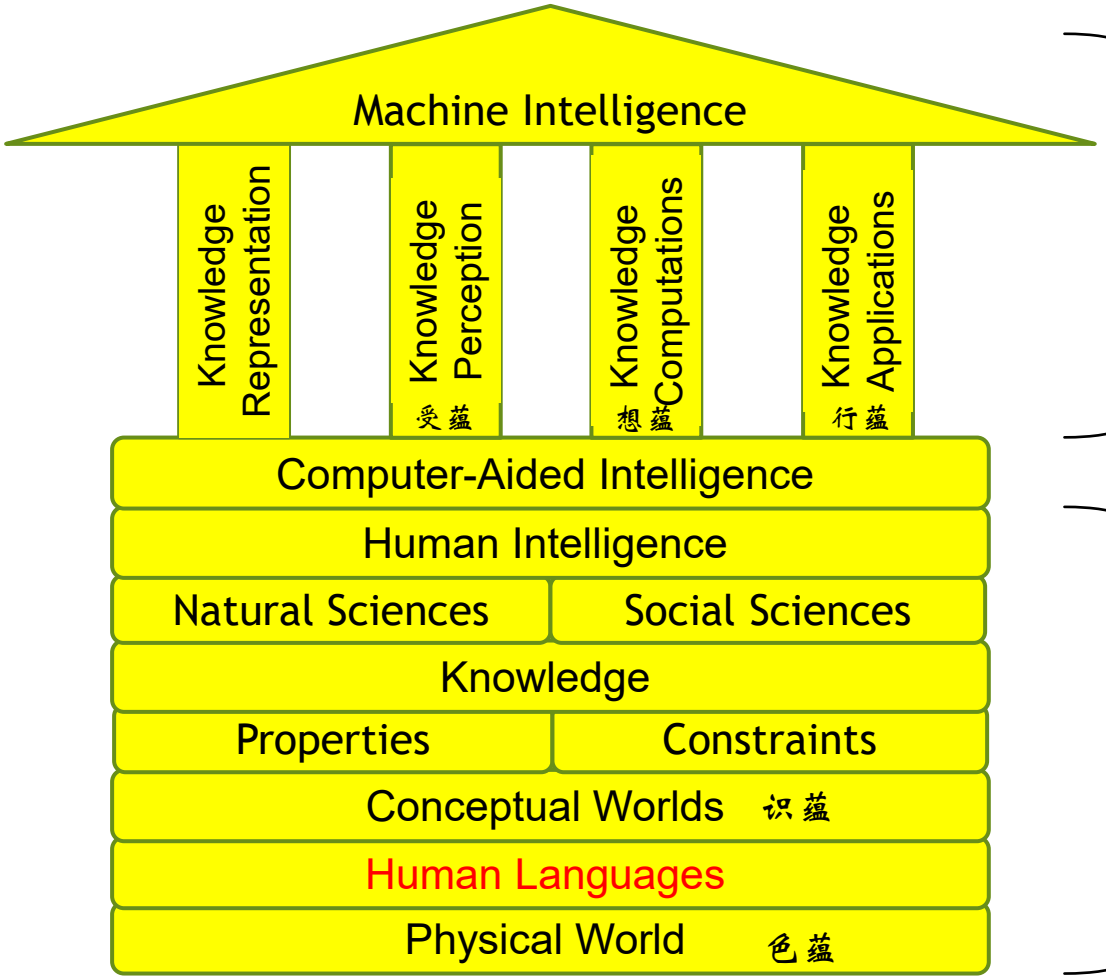
1. One Tool
2. Two Worlds
3. Three Intelligences
4. Four Pillars

Multiple AI Pipelines

Signal → Knowledge

Knowledge → Knowledge

Knowledge → Signal



Programming versus Writing

Programming

Use of Programming Languages?

- You have a solution in mind
- You organize your solution
- You compose your program
- You test your program
- You deploy your program

Use of Keil

Writing

Use of Natural Languages?

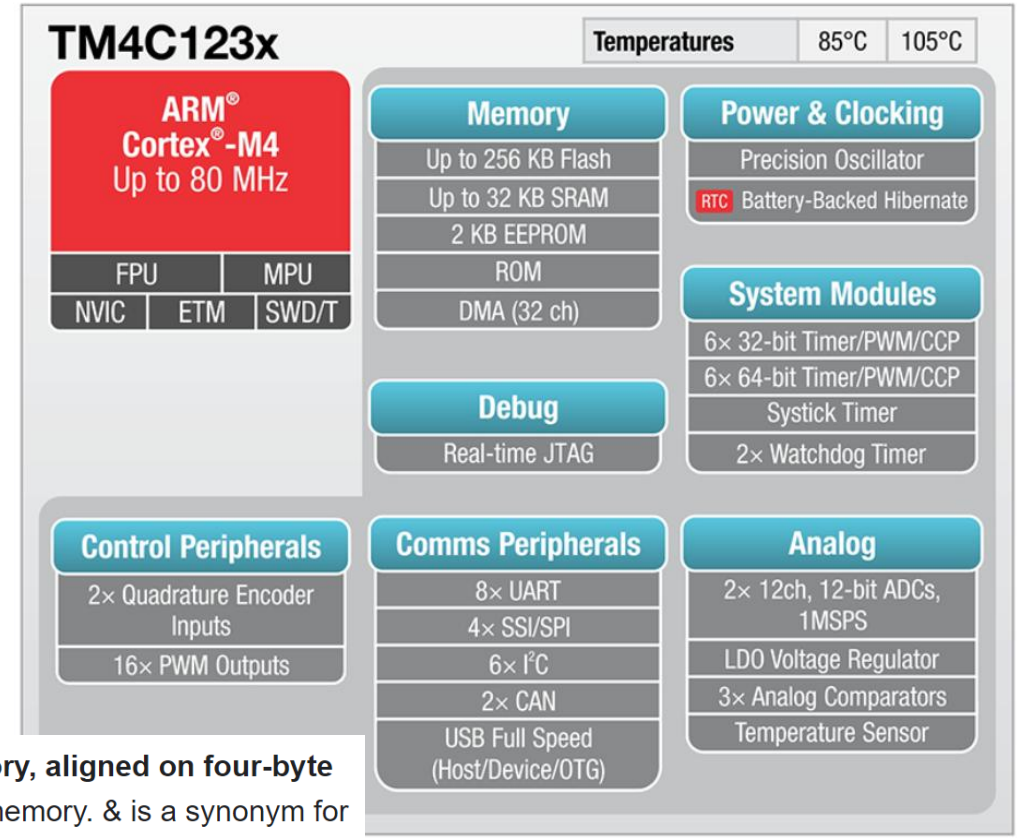
- You have a story in mind
- You organize your story
- You compose your texts
- You proof-read your texts
- You publish your texts

Use of LaTex, Word

How to organize your solution?

FPU: Floating-Point Unit
MPU: Multi-core Processing Unit

- Step 1: To plan arithmetic and/or logical computations underlying your solution
- Step 2: To allocate memory resources which are indispensable for the implementation of the planned computations.



The DCD **directive allocates one or more words of memory, aligned on four-byte boundaries**, and defines the initial runtime contents of the memory. & is a synonym for DCD . DCDU is the same, except that the memory alignment is arbitrary.

Example of doing x = (a + b) - c

```
                EXPORT Start
                AREA    progA, CODE, READONLY
var_a   DCD     5       ; a = 5    4 byte words
var_b   DCD     6       ; b = 6
var_c   DCD     7       ; c = 7
var_x   DCD     0       ; x = 4

                        ; x = (a + b) – c
Start           ADR r4,var_a    ; get address for a
                LDR r0,[r4]     ; get value of a
                ADR r4,var_b    ; get address for b, reusing r4
                LDR r1,[r4]     ; get value of b
                ADD r3,r0,r1    ; compute a + b
                ADR r4,var_c    ; get address for c
                LDR r2,[r4]     ; get value of c
                SUB r3,r3,r2    ; complete computation of x
                ADR r4,var_x    ; get address for x
                STR r3,[r4]     ; store value of x
stop    B       stop
                END
```

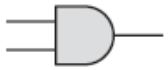
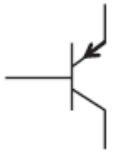
Allocation of Memory

Planning of Computations

ADR loads address to a register

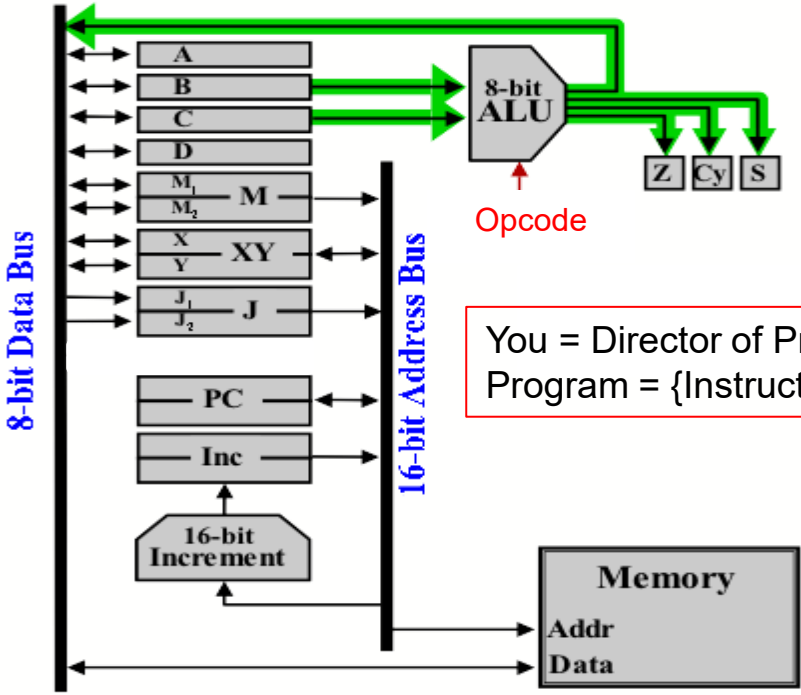
Role of Programmers: Planning and Allocation

- Data = {Values, Symbols, Addresses, Instructions}
- Instructions = {Op Code + Addresses + Value/Symbol}

Applications/OS	
Languages	C++, Java
ISA	EOR r3,r2,r1 BEQ Table
Microarchitecture	
Gates	
Transistors	

Algorithms or Solutions

Problems to be solved

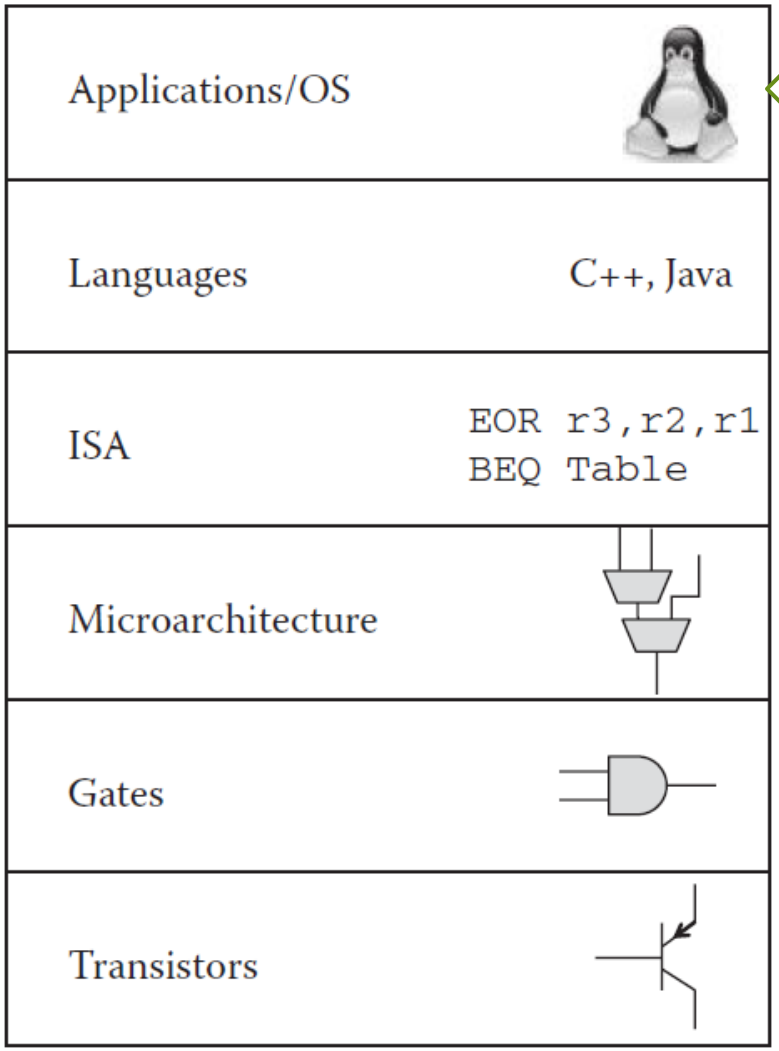


You = Director of Program
Program = {Instructions}

Memory = {Address + Data/Instruction}

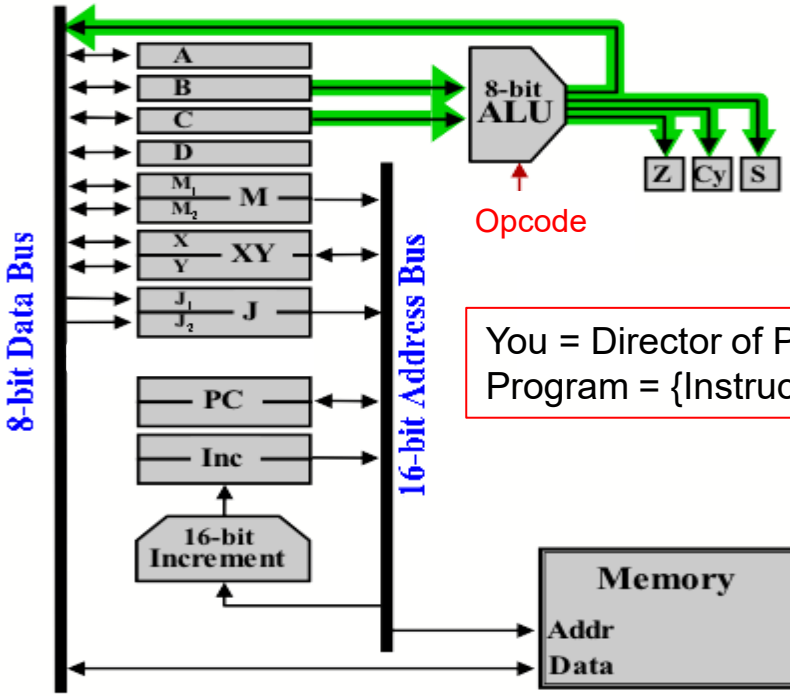
Role of Microprocessors: Do Cycle-by-Cycle Executions

- Data = {Values, Symbols, Addresses, Instructions}
- Instructions = {Op Code + Addresses + Value/Symbol}



Algorithms or Solutions

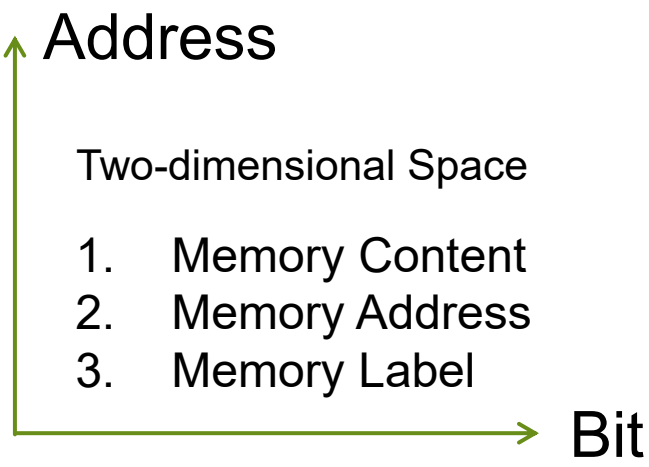
Problems to be solved

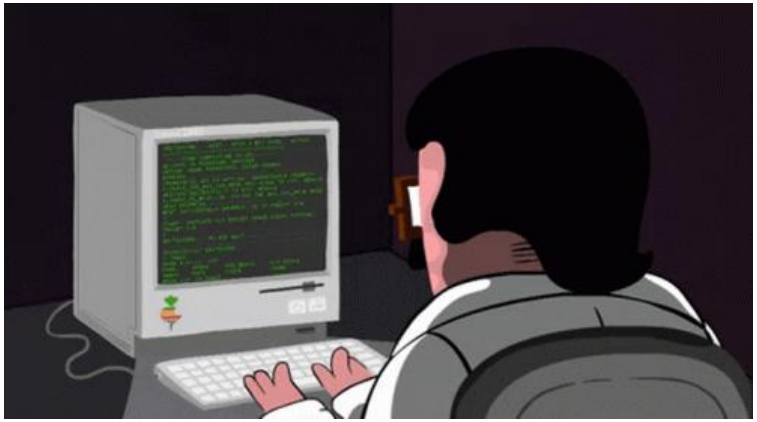


You = Director of Program
Program = {Instructions}

Memory = {Address + Data/Instruction}

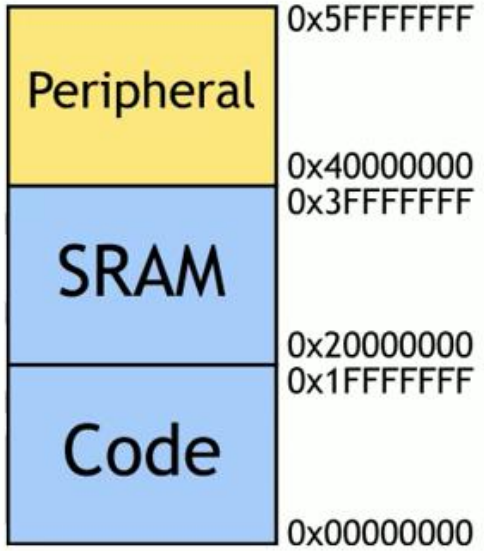
Outline





You = Director of Program
Program = {Instructions}

- Nature of Programming
- Planning of Computations in Programming
- Allocation of Memory in Programming
- ARM Programming Tools



Procedure of Planning Computations

- Step 1: Describe your solution in the form of mathematics or logics.
- Step 2: Transform your solution into algorithms which consist of series of arithmetic and/or logical computations.
- Step 3: Draw flowcharts which interconnect arithmetic and/or logical computations in terms of their inputs and outputs.
- Step 4: Describe flowcharts with the use of a programming language.

Example of Transforming Solution into Algorithm

Solution:

$$ax^2 + bx + c = 0$$

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Algorithm:

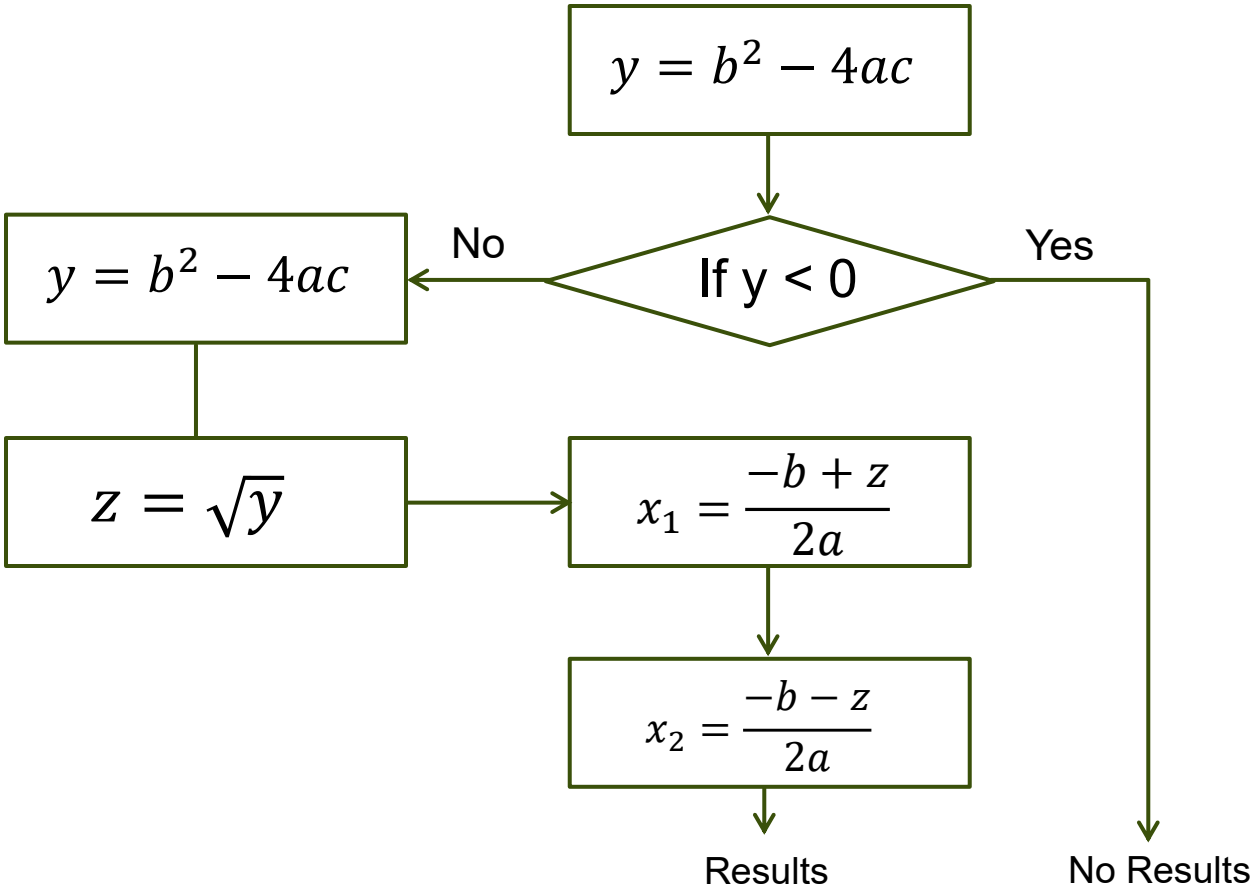
First do: $y = b^2 - 4ac$

Then do: $z = \sqrt{y}$

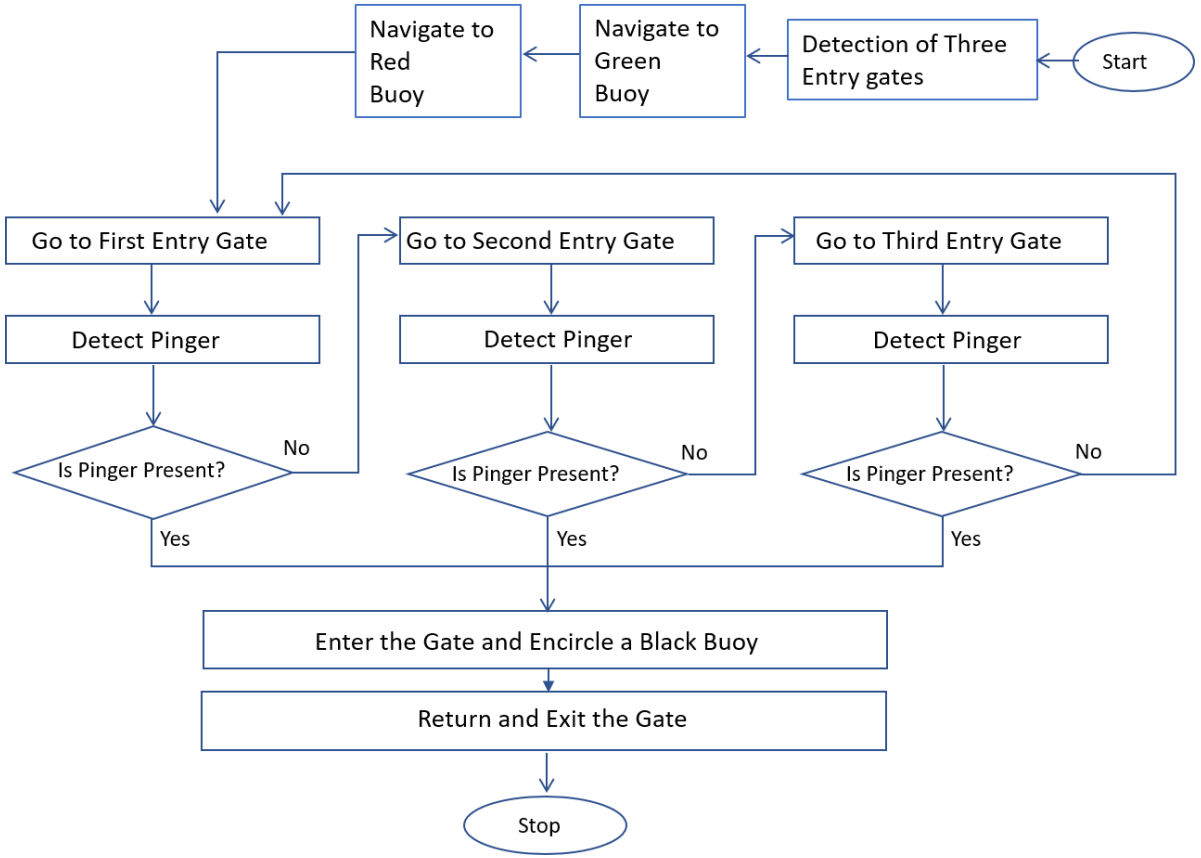
Then do: $x_1 = \frac{-b + z}{2a}$

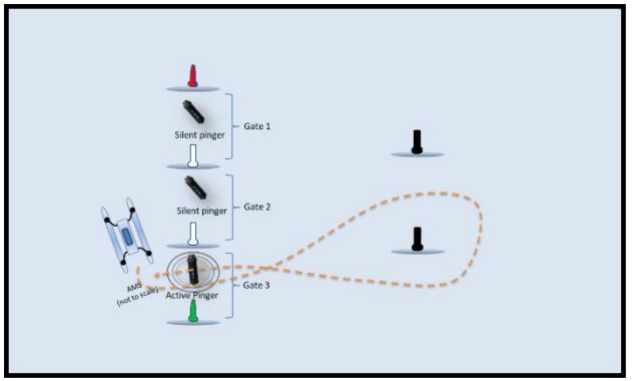
Then do: $x_2 = \frac{-b - z}{2a}$

Example of Flowchart Corresponding to Algorithm



More Example of Flowchart Corresponding to Algorithm



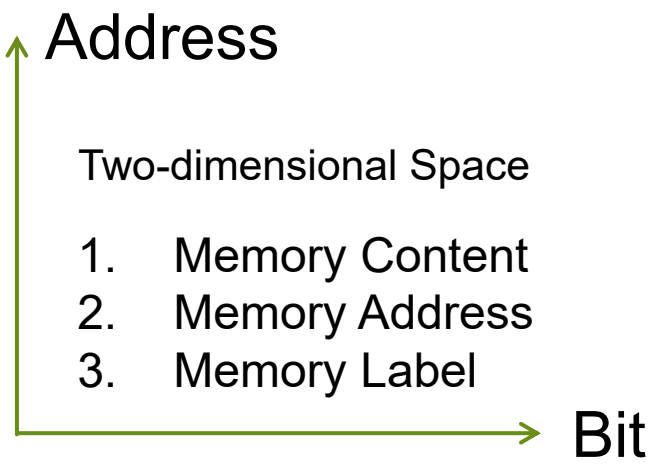


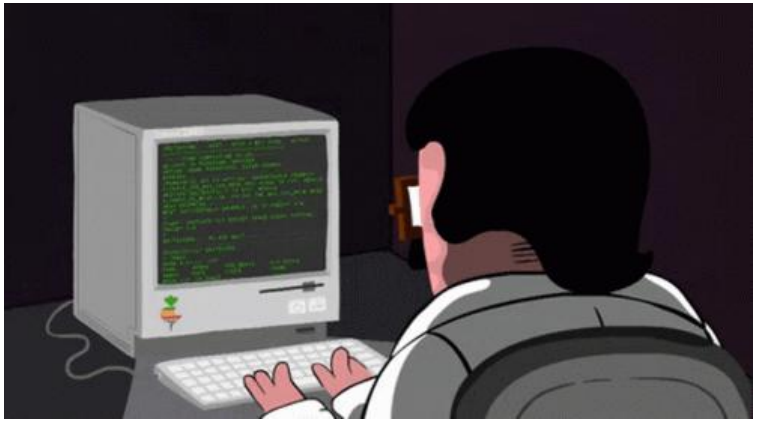
Example of Translation Flowchart into a Sequence of Instructions …

Burger Shop

```
void setup() {
    cleanTheShop()
    prepareIngredient()
    openTheStore()
}
void loop() {
    welcomeCustomer()
    receiveOrder()
    makeFood()
    deliverFood()
}
```

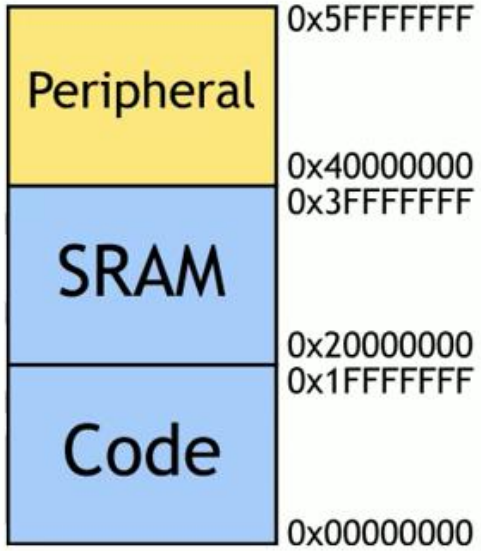
Outline





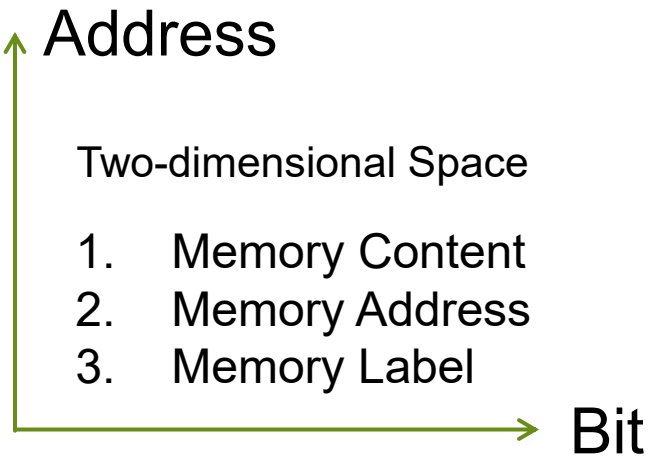
You = Director of Program
Program = {Instructions}

- Nature of Programming
- Planning of Computations in Programming
- Allocation of Memory in Programming
- ARM Programming Tools



Motivation

- The hardware could support the concurrent running of multiple application programs.
- The ALU is being shared among these programs. This means that an application program has the full access to the ALU during the time when ALU is allocated to it.
- However, memory units are not shared in general.
- Hence, each application program must take care of memory allocation explicitly.



ALU is shared while memory is to be reserved

Scenario of Memory Allocation

- Scenario 1: Allocation of memory for housing a program itself.
- Scenario 2: Allocation of memory for housing constants, parameters and variables of a program.

Address

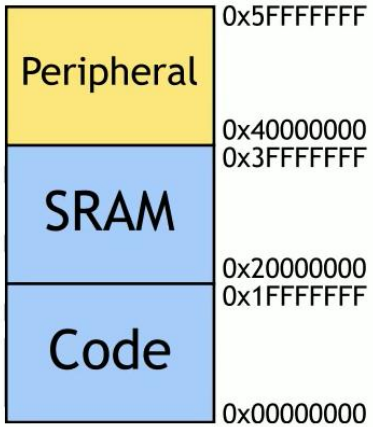
Two-dimensional Space

1. Memory Content
2. Memory Address
3. Memory Label

Bit

Cortex-M Memory

On-chip Memory Space



GiB = Giga Byte

- On-chip code, data, and I/O are located in the first 1.5 GiB of memory space
- Each is allocated 0.5 GiB
- May use physically separate buses for each space

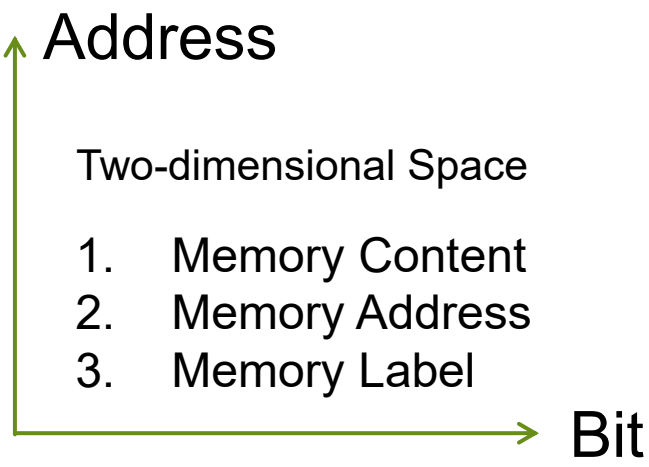
mem32[address]: consider the memory as a vector

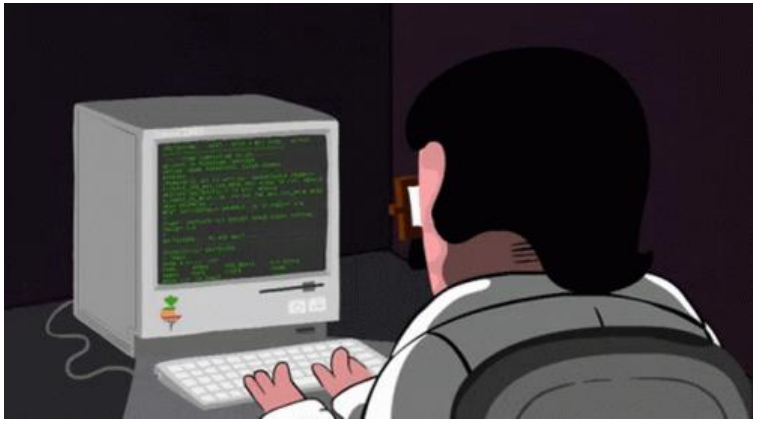
Example of Memory Allocation

```
                EXPORT Start
                AREA   progB, CODE, READONLY
var_a   DCD     1       ; a = 1    (DCD creates 4 byte variables)
var_b   DCD     15      ; b = 15
var_z   DCD     0       ; z = 5

Start                   ; z = (a << 2) | (b & 15)
        ADR r4,var_a    ; get address for a
        LDR r0,[r4]     ; get value of a
        MOV r0, r0, LSL #2   ; perform shift  ---> r0 = (a<<2)
        ADR r4, var_b   ; get address for b
        LDR r1, [r4]    ; get value of b
        AND r1, r1, #15 ; perform AND  ---> r1 = (b & 152)
        ORR r1, r0, r1  ; perform OR   ---> r1 = (a<<2) | (b & 152)
        ADR r4, var_z   ; get address for z
        STR r1, [r4]    ; store value for z
stop    B       stop
        END
```

mem32[address]: consider the memory as a vector

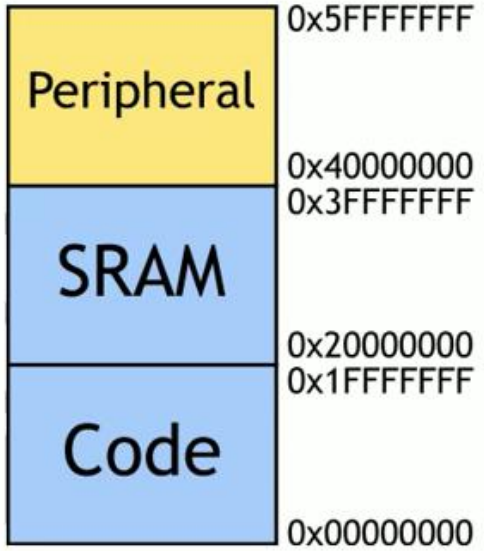
Outline





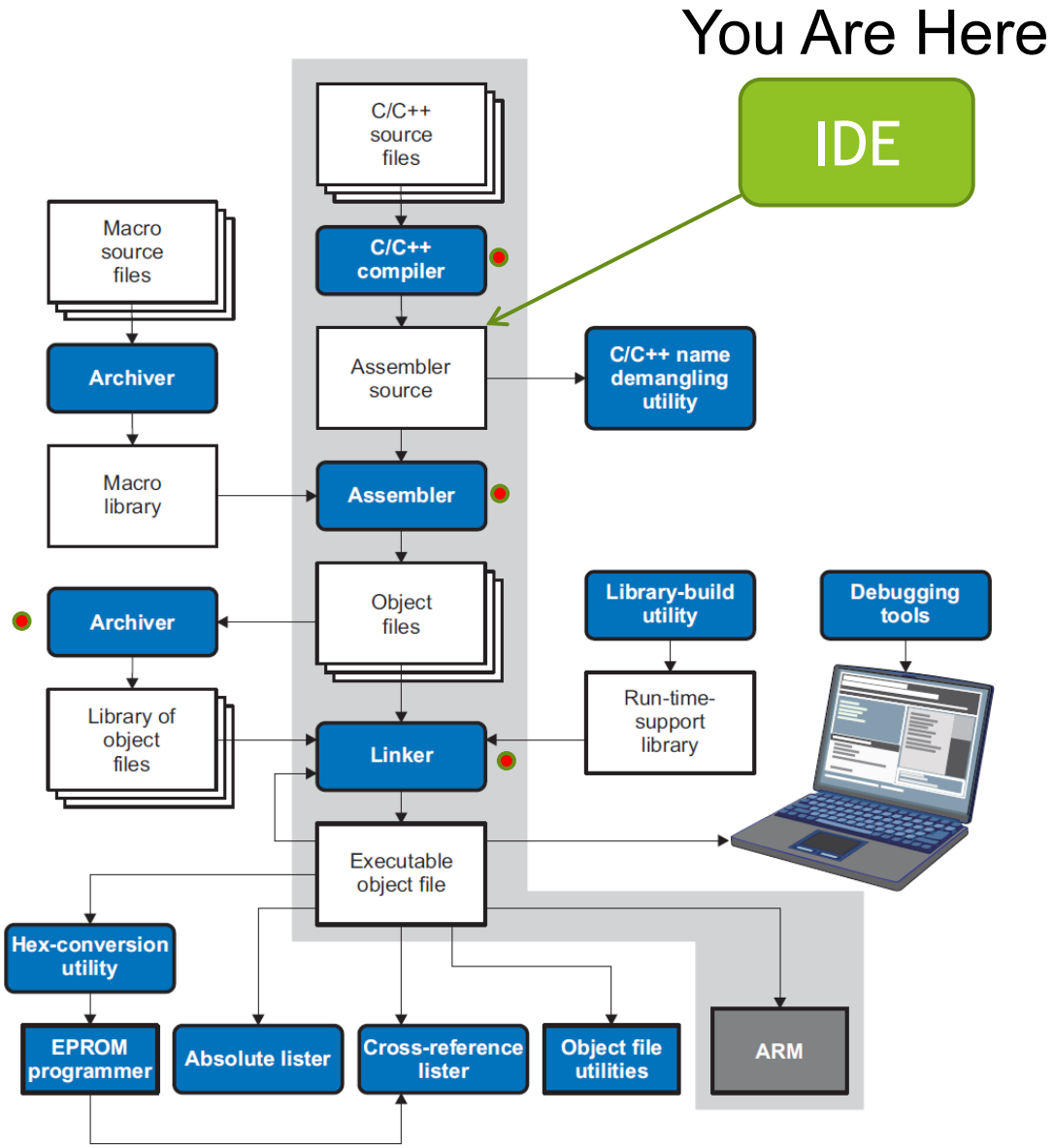
You = Director of Program
Program = {Instructions}

- Nature of Programming
- Planning of Computations in Programming
- Allocation of Memory in Programming
- ARM Programming Tools



Overview

- Compiler:
  - From .c files to .s files
- Assembler:
  - From .s files to .o files
- Linker:
  - From .o files to .exe files
  - May be linked with .a files

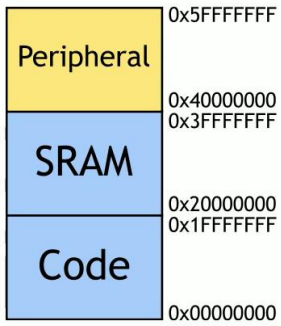


Assembler Directives for Memory Destination

Mnemonic and Syntax	Description
.bss *symbol, size in bytes*[, *alignment* [, *bank offset*]]	Reserves *size* bytes in the .bss (uninitialized data) section
.data	Assembles into the .data (initialized data) section
.sect "*section name*"	Assembles into a named (initialized) section
.text	Assembles into the .text (executable code) section
symbol **.usect** "*section name*", *size in bytes* [, *alignment*[, *bank offset*]]	Reserves *size* bytes in a named (uninitialized) section

Cortex-M Memory

On-chip Memory Space



- On-chip code, data, and I/O are located in the first 1.5 GiB of memory space
- Each is allocated 0.5 GiB
- May use physically separate buses for each space

SRAM: Data Only
CODE: Instructions/Data

- Short-term memory
- Working memory

Examples

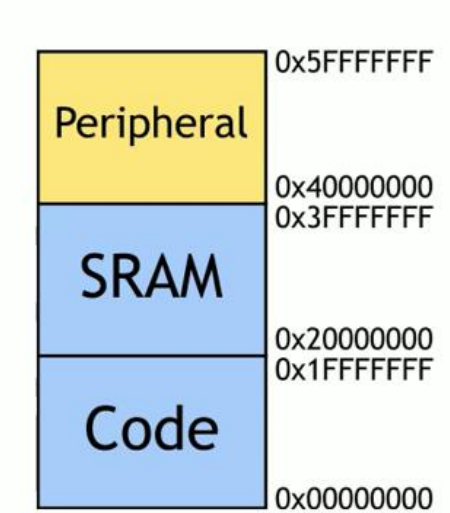
.space reserves a block of bytes

In this example, code is assembled into the .data and .text sections.

Machine Readable

```
                    ************************************************
                    **          Reserve space in .data.           **
                    ************************************************
00000000                    .data
00000000                    .space  0CCh

                    ************************************************
                    **          Assemble into .text.              **
                    ************************************************
00000000                    .text           ; Constant into .data
         00000000   INDEX   .set    0
00000000 E3A00000           MOV     R0, #INDEX

                    ************************************************
                    **          Assemble into .data.              **
                    ************************************************
000000cc            Table:  .data
000000cc FFFFFFFF           .word   -1      ; Assemble 32-bit
                                            ; constant into .data.

000000d0 FF                 .byte   0FFh    ; Assemble 8-bit
                                            ; constant into .data.

                    ************************************************
                    **          Assemble into .text.              **
                    ************************************************
00000004                    .text
00000004 000000CC   con:    .field  Table, 32
00000008 E51F100C           LDR     R1, con
0000000c E5912000           LDR     R2, [R1]
00000010 E0802002           ADD     R2, R0, R2
```

Human Readable



Assembler Directives for Data's Memory Allocation

Mnemonic and Syntax	Description
.cstring {*expr*$_1$|"*string*$_1$"}[,... , {*expr*$_n$|"*string*$_n$"}]	Initializes one or more text strings
.double *value*$_1$[, ... , *value*$_n$]	Initializes one or more 64-bit, IEEE double-precision, floating-point constants
.field *value*[, *size*]	Initializes a field of *size* bits (1-32) with *value*
.float *value*$_1$[, ... , *value*$_n$]	Initializes one or more 32-bit, IEEE single-precision, floating-point constants
.half *value*$_1$[, ... , *value*$_n$]	Initializes one or more 16-bit integers (halfword)
.int *value*$_1$[, ... , *value*$_n$]	Initializes one or more 32-bit integers
.long *value*$_1$[, ... , *value*$_n$]	Initializes one or more 32-bit integers
.short *value*$_1$[, ... , *value*$_n$]	Initializes one or more 16-bit integers (halfword)
.string {*expr*$_1$|"*string*$_1$"}[,... , {*expr*$_n$|"*string*$_n$"}]	Initializes one or more text strings
.ubyte *value*$_1$[, ... , *value*$_n$]	Initializes one or more successive unsigned bytes in the current section
.uchar *value*$_1$[, ... , *value*$_n$]	Initializes one or more successive unsigned bytes in the current section
.uhalf *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 16-bit integers (halfword)
.uint *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 32-bit integers
.ulong *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 32-bit integers
.ushort *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 16-bit integers (halfword)
.uword *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 32-bit integers
.word *value*$_1$[, ... , *value*$_n$]	Initializes one or more 32-bit integers

Examples

Example 1 This example uses the .int directive to initialize words.

```
1 00000000                      .space 73h
2 00000000                      .bss   PAGE, 128    ; Reserve 128 bytes for pointer PAGE
3 00000080                      .bss   SYMPTR, 4    ; Reserve 4 bytes for pointer SYMPTR
4 00000074 E3A00056  INST:  MOV    R0, #056h
5 00000078 0000000A         .int   10, SYMPTR, -1, 35 + 'a', INST, "abc"
  0000007c 00000080
  00000080 FFFFFFFF
  00000084 00000084
  00000088 00000074
  0000008c 00000061
  00000090 00000062
  00000094 00000063
```

Example 2 This example shows how the .long directive initializes words. The symbol DAT1 points to the first word that is reserved.

```
1 00000000 0000ABCD  DAT1:  .long   0ABCDh, 'A' + 100h, 'g', 'o'
  00000004 00000141
  00000008 00000067
  0000000c 0000006F
2 00000010 00000000         .long   DAT1, 0AABBCCDDh
  00000014 AABBCCDD
3 00000018           DAT2:
```

Example 3 In this example, the .word directive is used to initialize words. The symbol WORDX points to the first word that is reserved.

```
1 00000000 00000C80  WORDX:  .word   3200, 1 + 'AB', -0AFh, 'X'
  00000004 00004242
  00000008 FFFFFF51
  0000000c 00000058
```

Assembler Directives for Controlling Conditional Computations

Mnemonic and Syntax	Description
.if *condition*	Assembles code block if the *condition* is true
.else	Assembles code block if the .if *condition* is false. When using the .if construct, the .else construct is optional.
.elseif *condition*	Assembles code block if the .if *condition* is false and the .elseif condition is true. When using the .if construct, the .elseif construct is optional.
.endif	Ends .if code block
.loop [*count*]	Begins repeatable assembly of a code block; the loop count is determined by the *count*.
.break [*end condition*]	Ends .loop assembly if *end condition* is true. When using the .loop construct, the .break construct is optional.
.endloop	Ends .loop code block

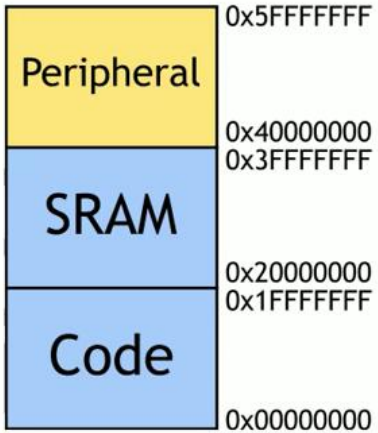
Example

This example shows conditional assembly:

Memory for Data

Memory for Code

```
 1                00000001  SYM1     .set    1
 2                00000002  SYM2     .set    2
 3                00000003  SYM3     .set    3
 4                00000004  SYM4     .set    4
 5
 6                          If_4:    .if     SYM4 = SYM2 * SYM2
 7  00000000  04                     .byte   SYM4              ; Equal values
 8                                   .else
 9                                   .byte   SYM2 * SYM2       ; Unequal values
10                                   .endif
11
12                          If_5:    .if     SYM1 <= 10
13  00000001  0A                     .byte   10                ; Less than / equal
14                                   .else
15                                   .byte   SYM1              ; Greater than
16                                   .endif
17
18                          If_6:    .if     SYM3 * SYM2 != SYM4 + SYM2
19                                   .byte   SYM3 * SYM2       ; Unequal value
20                                   .else
21  00000002  08                     .byte   SYM4 + SYM4       ; Equal values
22                                   .endif
23
24                          If_7:    .if     SYM1 = SYM2
25                                   .byte   SYM1
26                                   .elseif SYM2 + SYM3 = 5
27  00000003  05                     .byte   SYM2 + SYM3
28                                   .endif
```



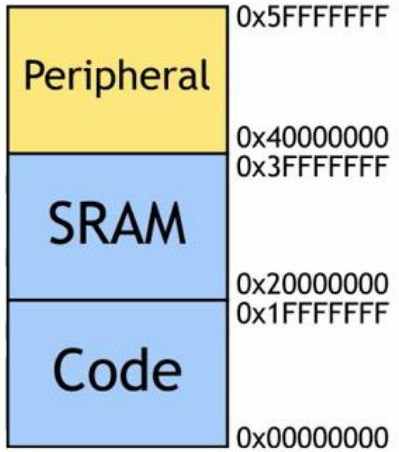
Directive for Creating Reusable Block of Codes (i.e. MACRO)

A macro definition is a series of source statements in the following format:

```
macname   .macro   [parameter1 ] [, ... , parametern ]
          model statements or macro directives
          [.mexit]
          .endm
```

macname	names the macro. You must place the name in the source statement's label field. Only the first 128 characters of a macro name are significant. The assembler places the macro name in the internal opcode table, replacing any instruction or previous macro definition with the same name.
.macro	is the directive that identifies the source statement as the first line of a macro definition. You must place .macro in the opcode field.
parameter $_1$, *parameter* $_n$	are optional substitution symbols that appear as operands for the .macro directive.
.mexit	is a directive that functions as a *goto .endm*. The .mexit directive is useful when error testing confirms that macro expansion fails and completing the rest of the macro is unnecessary.
.endm	is the directive that terminates the macro definition.

Example: x = a + b + c



Macro definition: The following code defines a macro, add3, with four parameters:

```
 1                      *
 2
 3                      *       add3
 4                      *
 5                      *       ADDRP = P1 + P2 + P3
 6
 7                      add3    .macro P1, P2, P3, ADDRP
 8
 9                              ADD     ADDRP, P1, P2
10                              ADD     ADDRP, ADDRP, P3
11                              .endm
```

Create a macro

Macro call: The following code calls the add3 macro with four arguments:

```
12
13 00000000                    add3 R1, R2, R3, R0
```

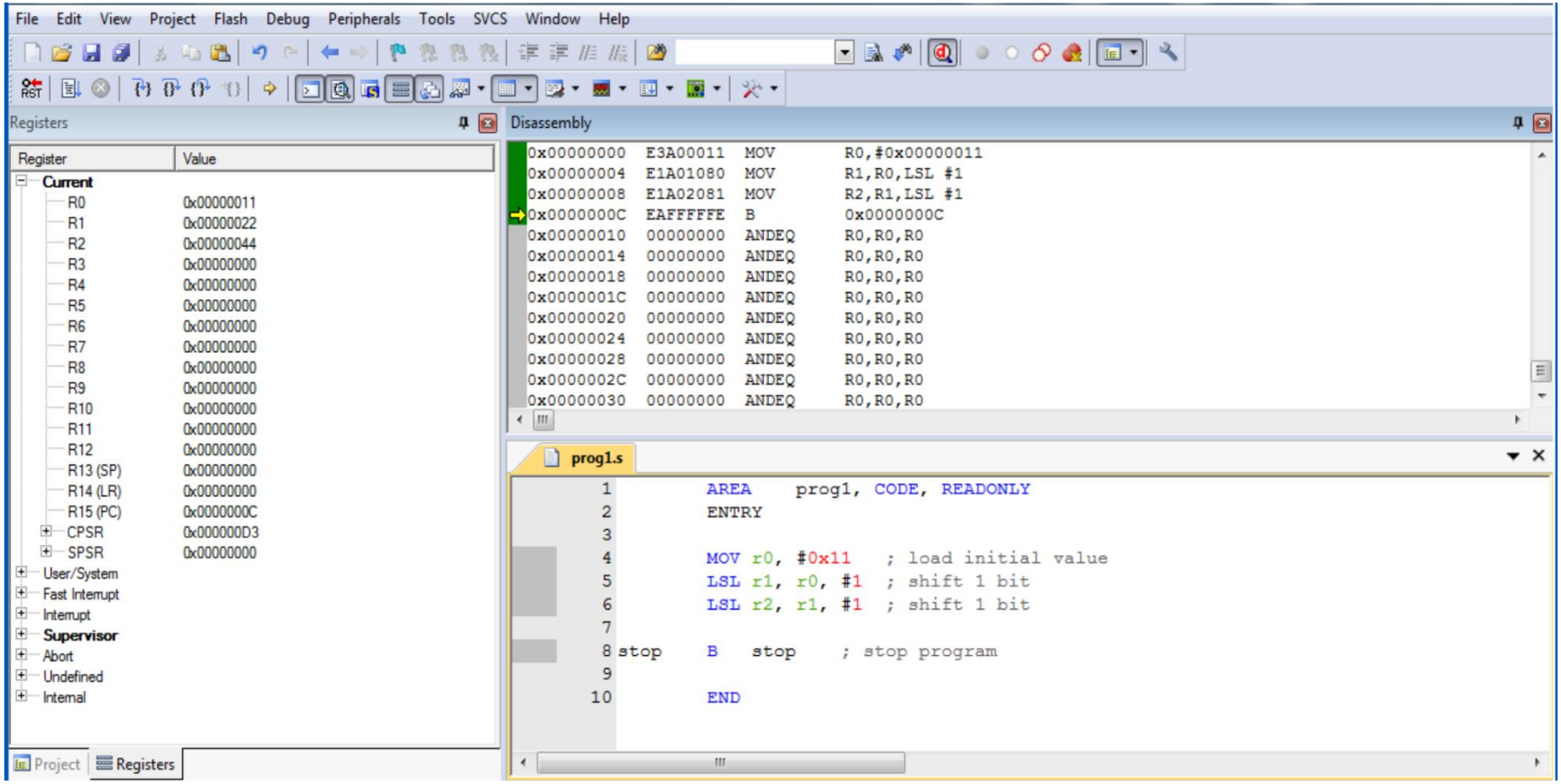
Invoke or call a macro

Macro expansion: The following code shows the substitution of the macro definition for the macro call. The assembler substitutes R1, R2, R3, and R0 for the P1, P2, P3, and ADDRP parameters of add3.

```
1
1       00000000 E0810002              ADD     R0, R1, R2
1       00000004 E0800003              ADD     R0, R0, R3
```

Equivalent instructions

Good News: Integrated Development Environment (Keil)

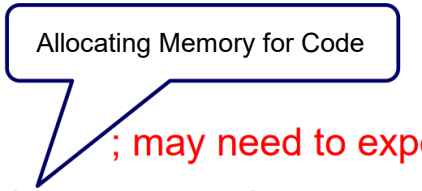


Keil Directives for Allocating Memory to House Instructions/Data

- Format:

  { label } {instruction | directive | pseudo-instruction} { ; comment }

- Example:




```
                EXPORT Start          ; may need to export start
                AREA ARMex, CODE, READONLY     ; Assembler Directive
                                      ; Assembler directive – 1st instruction to
                                      ; execute with standalone program
Start   MOV   r0, #10   ; label & processor instruction
        MOV   r1, #3    ; another processor instruction
stop    B     stop      ; label & another processor instruction
        END             ; End of source file – assembler directive
```

Keil Directive for Reusable Instructions (i.e. MACRO and MEND)

- The MACRO directive marks the start of the definition of a macro. Macro expansion terminates at the MEND directive.

- Format:
  - MACRO
  - {$label} macroname{$cond} {$parameter{,$parameter}...}
  - ; code
  - MEND

- Example:

```
MACRO
        ; mcro definition
        ; vara=8 * (varb + varc + 6)

$Label_1 AddMul $vara, $varab, $varac

$Label_1
        Add $vara, $varab, $varac
        Add $vara, $vara  , #6
        LSL  $vara, $vara  , #3
        MEND
```

Invocation of MACRO

```
MACRO
        ; mcro definition
        ; vara=8 * (varb + varc + 6)

$Label_1 AddMul $vara, $varab, $varac

$Label_1
        Add $vara, $varab, $varac
        Add $vara, $vara , #6
        LSL  $vara, $vara , #3
        MEND
```

- Invoke Macro

  AddMul r0, r1, r2

- Resultant code after

  ADD r0, r1, r2

  ADD r0, r1, #6

  LSL r0, r1, #3

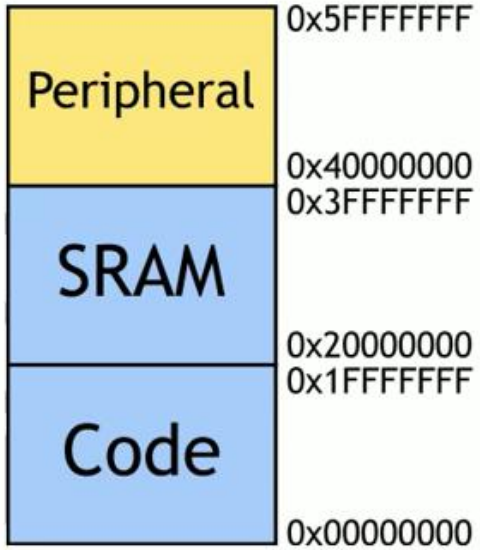
More Example

```
 ; macro definition
                MACRO                   ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                    ; end macro definition
```

```
 ; macro invocation
abc             xmac    subr1,de        ; invoke macro
                ; code                  ; this is what is
abcloop1        ; code                  ; is produced when
                ; code                  ; the xmac macro is
                BGE     abcloop1        ; expanded
abcloop2        ; code
                BL      subr1
                BGT     abcloop2
                ; code
                ADR     de
                ; code
```

Allocation of Constants or Parameters in Keil

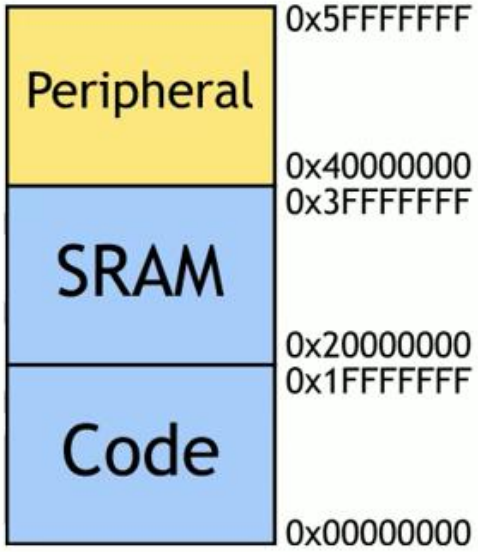
- Format with SETS Directive:
  - {name} SETS value
- Example:
  - mytext SETS “This is my text” ;
- Format with EQU Directive:
  - {name} EQU value {, type}
- Example:
  - temperature EQU 25.6 ;



Formats of Numbers in Keil

Number formats

- 123 : decimal number
- 0x3f : hexadecimal number
- n_xxx n : base (2 to 9) , xxx : number
  - 8_12 : octal number 12
- 'A' : single character constant
- "string" : string constants



Frequently Used Keil's Directives

Assembler User Guide: Directives Reference (keil.com)

Keil Directive	Uses
AREA	Defines a block of code or data
RN	Can be used to associate a register with a name
EQU	Equates a symbol to a numeric constant
ENTRY	Declares an entry point to your program
DCB, DCW, DCD	Allocates memory and specifies initial runtime contents
ALIGN	Aligns data or code to a particular memory boundary
SPACE	Reserves a zeroed block of memory of a particular size
LTORG	Assigns the starting point of a literal pool
END	Designates the end of a source file

The END directive informs the assembler that it has reached the end of a source file.

AREA

- The `AREA` directive instructs the assembler to assemble a new code or data section.

AREA *sectionname {,attr} {, attr}…*

AREA Name, CODE, READONLY

Valid Section Attributes (Keil Tools)

ALIGN = *expr*	This aligns a section on a 2^{expr}-byte boundary (note that this is different from the ALIGN directive); e.g., if *expr* = 10, then the section is aligned to a 1KB boundary.
CODE	The section is machine code (READONLY is the default)
DATA	The section is data (READWRITE is the default)
READONLY	The section can be placed in read-only memory (default for sections of CODE)
READWRITE	The section can be placed in read-write memory (default for sections of DATA)

RN Allocation of (Additional) Memory Label

- The `RN` directive defines a name for a specified register.

- Format:

  name RN expr
  - *name – name to assign to the register*
  - *expr – takes values between 0 to 15*

- Example:

```
coeff1      RN      8       ; coefficient 1
coeff2      RN      9       ; coefficient 2
dest        RN      0       ; register 0 holds the pointer to
                            ; destination matrix
```

EQU Allocation of Memory Label

- The `EQU` directive gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value.

- Format: Name EQU expr{, type}
  - *name* is the symbolic name to assign to the value,
  - *expr is an address or integer constant*
  - *Type is optional and can be:* ARM, THUMB, CODE16, CODE32, DATA

- Example:

```
SRAM_BASE   EQU    0x04000000   ; assigns SRAM a base address
abc         EQU    2            ; assigns the value 2 to the symbol abc
xyz         EQU    label+8      ; assigns the address (label+8)
                                ; to the symbol xyz
fiq         EQU    0x1C, CODE32 ; assigns the absolute address
                                ; 0x1C to the symbol fiq, and marks it
                                ; as code
```

ENTRY

- The `ENTRY` directive declares an entry point to a program.

A program must have an entry point. You can specify an entry point in the following ways:
•Using the `ENTRY` directive in assembly language source code.
•Providng a `main()` function in C or C++ source code.

- Example:

```
        AREA    ARMex, CODE, READONLY
        ENTRY       ; Entry point for the application.
        EXPORT ep1  ; Export the symbol so the linker can find it
ep1                 ; in the object file.
        ; code
        END
```

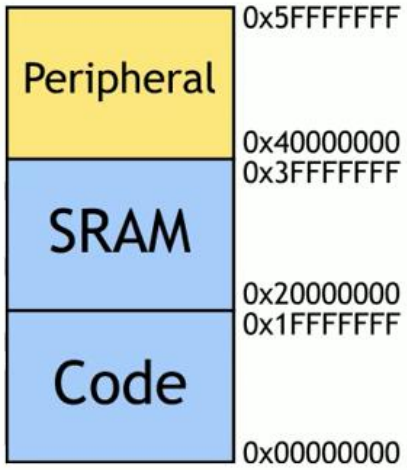
DCB (i.e. Declare 8-bit bytes)

- The `DCB` directive allocates one or more bytes of memory and defines the initial runtime contents of the memory.

- Format:
  - {label} DCB expression {, expression} …

- Example:

```
C_string DCB "C_string",0
```



Last Byte

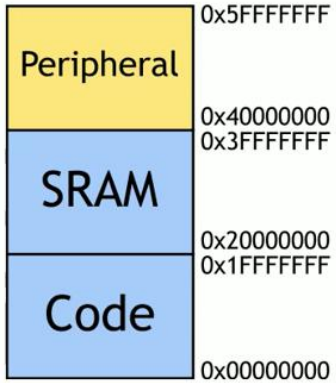
Address		ASCII equivalent
0x4000	43	C
0x4001	5F	_
0x4002	73	s
0x4003	74	t
0x4004	72	r
0x4005	69	i
0x4006	6E	n
0x4007	67	g
0x4008	00	

DCW (i.e. Declare 16-bit words)

- The `DCW` directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory. `DCWU` is the same, except that the memory alignment is arbitrary.

- Format:
  - {label} DCW{U} expression {, expression} …

- Example:



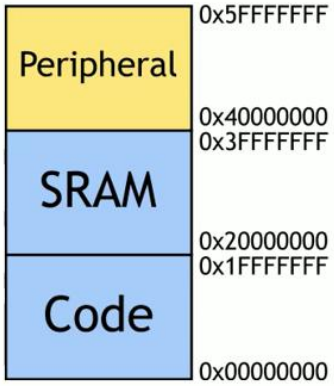

```
data    DCW     -225,2*number   ; number must already be defined
        DCWU    number+4
```

DCD (i.e. Declare 32-bit words)

- The `DCD` directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory. `DCDU` is the same, except that the memory alignment is arbitrary.

- Format:
  - {label} DCD{U} expression {, expression} …

- Example:




```
data1   DCD     1,5,20          ; Defines 3 words containing
                                ; decimal values 1, 5, and 20
data2   DCD     mem06 + 4       ; Defines 1 word containing 4 +
                                ; the address of the label mem06
        AREA    MyData, DATA, READWRITE
        DCB     255             ; Now misaligned ...
data3   DCDU    1,5,20          ; Defines 3 words containing
                                ; 1, 5 and 20, not word aligned
```

ALIGN (i.e. Change offset of alignment)

- The `ALIGN` directive aligns the current location to a specified boundary.

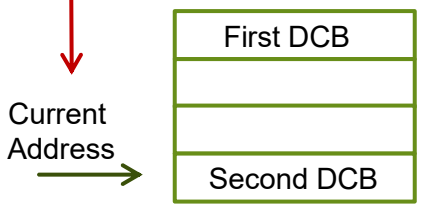
- Format: ALIGN { expr { , offset } }

  expr – a numeric expression evaluating to any power of 2^0 to 2^{31}

  Current location is defined by: offset + n * *expr. If not defined , location is the next word.*

- Example

```
        AREA    cacheable, CODE, ALIGN=3
rout1   ; code          ; aligned on 8-byte boundary
        ; code
        MOV     pc,lr   ; aligned only on 4-byte boundary
        ALIGN   8       ; now aligned on 8-byte boundary
rout2   ; code
```




```
AREA    OffsetExample, CODE
DCB     1       ; This example places the two bytes in the first
ALIGN   4,3     ; and fourth bytes of the same word.
DCB     1       ; The second DCB is offset by 3 bytes from the
                ; first DCB.
```

ALIGN 4, 3 means: data size = 4 bytes, offset between two consecutive data = 3 bytes

SPACE and FILL (i.e. allocate bytes)

- The `SPACE` directive reserves a zeroed block of memory. The `FILL` directive reserves a block of memory to fill with a given value.

- Format:
  - {label} SPACE expression
  - {label} FILL expression {, value {, value-size in terms of byte}}

- Example:

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255         ; defines 255 bytes of zeroed store
data2   FILL    50,0xAB,1 ; defines 50 bytes containing 0xAB
```

LTORG (i.e. Lookup-Table Organized)

(Literals mean "fixed and unchanging values")

- The LTORG directive instructs the assembler to assemble the current literal pool immediately. The assembler assembles the current literal pool at the end of every code section. The end of a code section is determined by the AREA directive at the beginning of the following section, or the end of the assembly.

In computer science, and specifically in compiler and assembler design, a literal pool is a lookup table used to hold literals during assembly and execution.

The assembler uses **literal pools** to store some constant data in code sections.

Working memory of a function

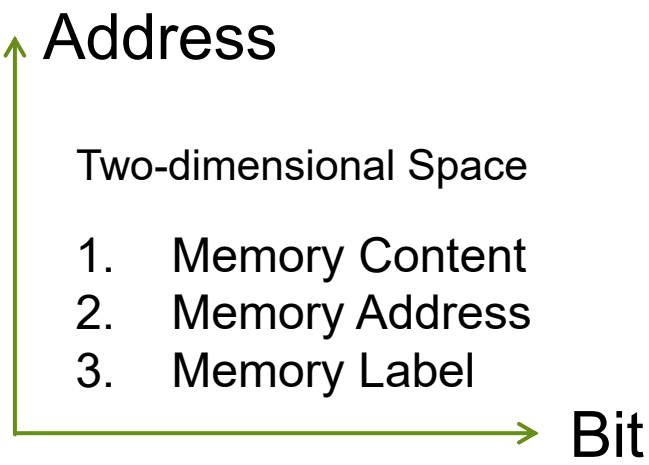
- Example:

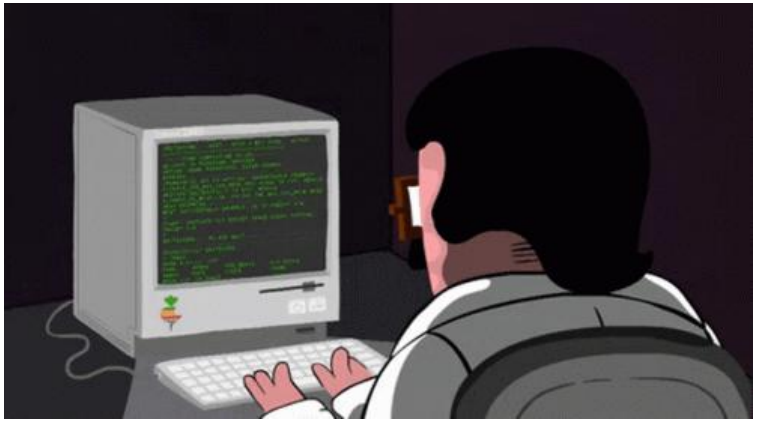
```
        AREA    Example, CODE, READONLY
start   BL      func1
func1                           ; function body
        ; code
        LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 1]
        ; code
        MOV     pc,lr           ; end function
        LTORG                   ; Literal Pool 1 contains literal &55555555.
data    SPACE   4200            ; Clears 4200 bytes of memory starting at current location.
        END                     ; Default literal pool is empty.
```

Other Operations Offered by Keil

	Keil Tools
A modulo B	A:MOD:B
Rotate A left by B bits	A:ROL:B
Rotate A right by B bits	A:ROR:B
Shift A left by B bits	A:SHL:B or A << B
Shift A right by B bits	A:SHR:B or A >> B
Add A to B	A + B
Subtract B from A	A – B
Bitwise AND of A and B	A:AND:B
Bitwise Exclusive OR of A and B	A:EOR:B
Bitwise OR of A and B	A:OR:B

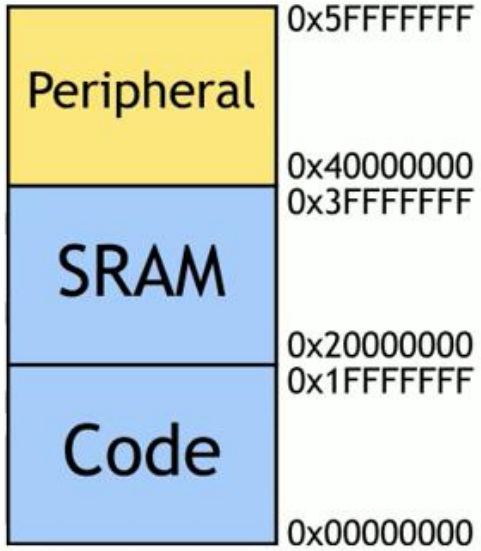
Summary





You = Director of Program
Program = {Instructions}

- Nature of Programming
- Planning of Computations in Programming
- Allocation of Memory in Programming
- ARM Programming Tools



"Ask not what your country can do for you – ask what you can do for your country," - John F. Kennedy

"Do not think that you are needy – think that you are needed in the world", - Manis Friedman

"Study will make you knowledgeable, resourceful, and hence more needed", - Xie Ming

Thank You for Listening!