CET 241: Day 3

Dr. Noori KIM

Software processing overview: Program execution flow - ARM ISA, assembling brief

Recap C programming

- What is a C program?
  - Simply just a text file ≠ understandable
    - Contains only human readable characters
    - Can be opened with notepad
  - Computer stores characters via ASCII code mapping
    - Every character is represented by bits

- A heart of computers (the generalization of computing process)

CU

1. Fetch: instructions
   - From memory into CU
2. Decode: split instructions
   - Opcode, Operand..
   - Pass values to ALU

ALU

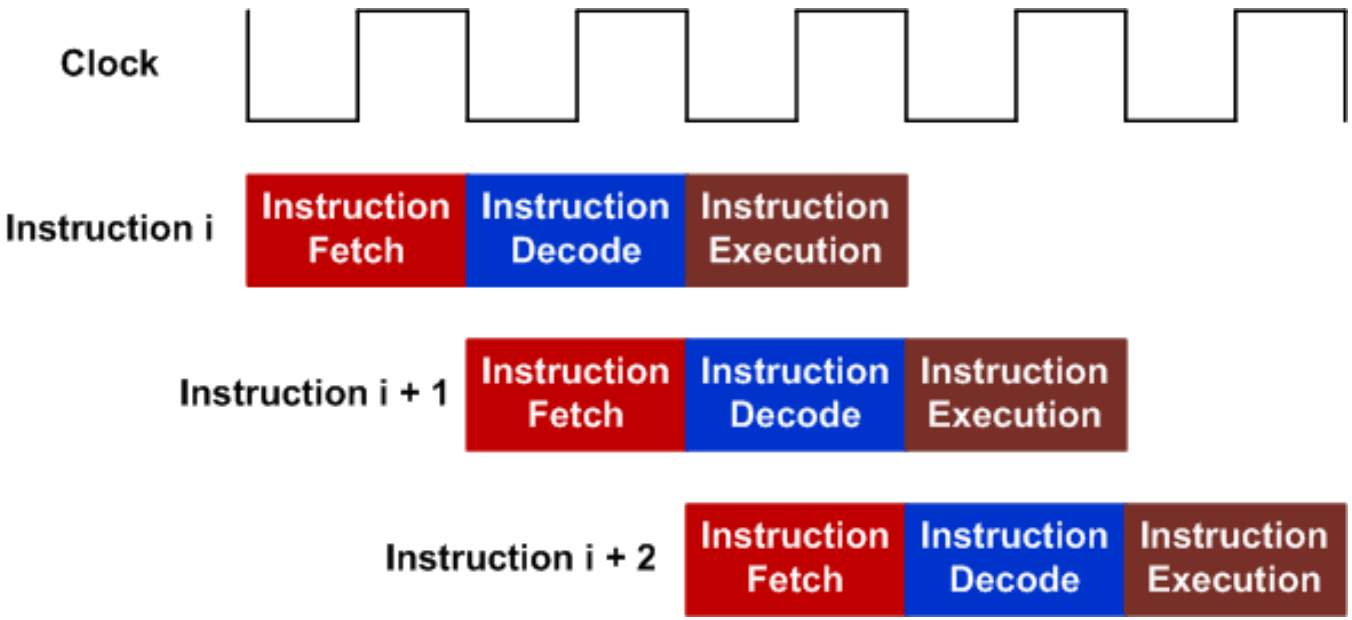
3. Execute: perform the operation

(4. Write back)

Cycle

Three-state pipeline: Fetch, Decode, Execution (Cortex-M3)

- **Pipelining** allows hardware resources to be fully utilized
- One 32-bit instruction or two 16-bit instructions can be fetched.



Pipeline of 32-bit instructions

Levels of Program Code

C Program

```
int main(void){
  int i;
  int total = 0;
  for (i = 0; i < 10; i++) {
    total += i;
  }
  while(1); // Dead loop
}
```

Compile →

Assembly Program

```
        MOVS r1, #0
        MOVS r0, #0
        B    check
loop    ADD  r1, r1, r0
        ADDS r0, r0, #1
check   CMP  r0, #10
        BLT  loop
self    B    self
```

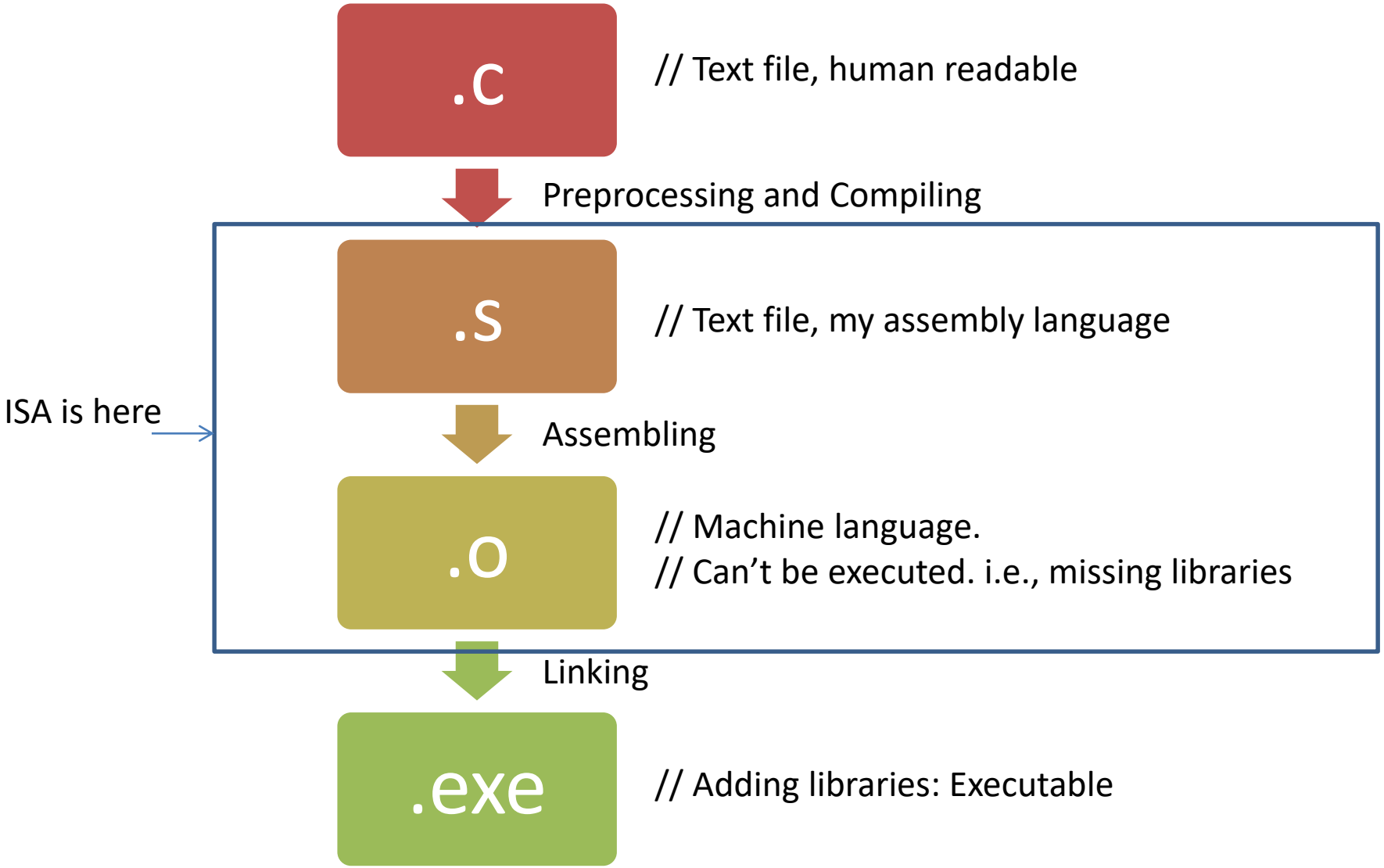
Assemble →

Machine Program

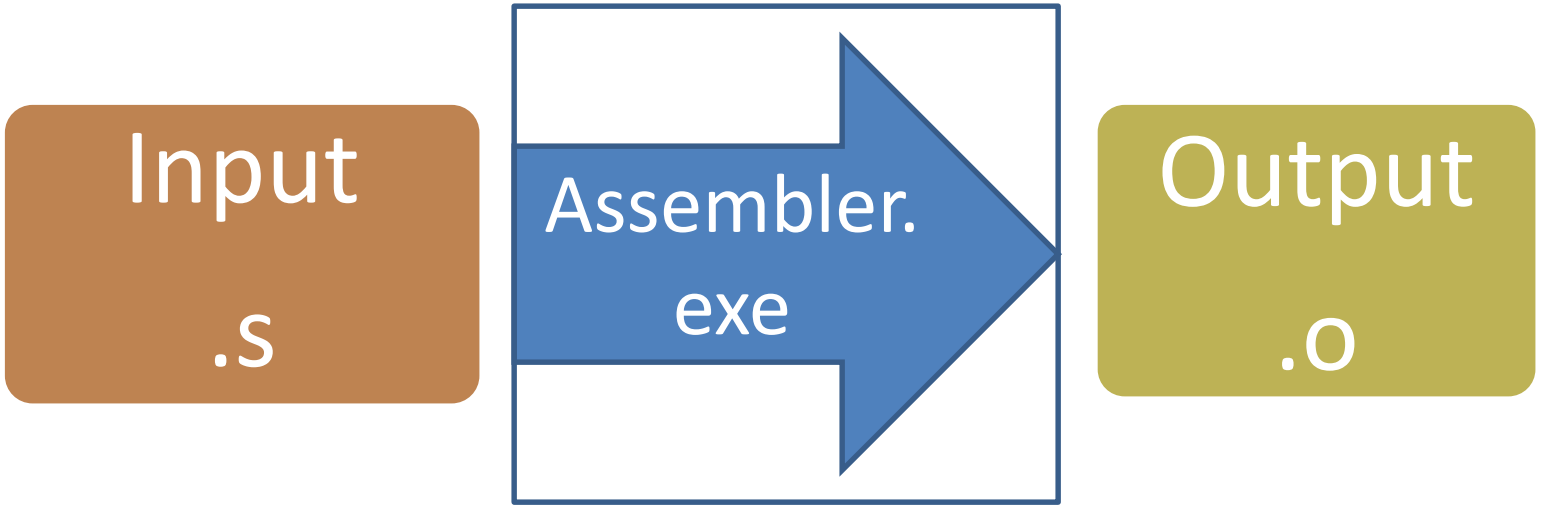
```
0010000100000000
0010000000000000
1110000000000001
0100010000000001
0001110001000000
0010100000001010
1101110011111011
1011111100000000
1110011111111110
```

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

- In short, the C program execution flow



- Each process requires F-D-E process



1. Fetch instructions
2. Decode via **ISA**
3. Execute

(4. Write back)

Instruction **S**et **A**rchitecture: **ISA**

Machine language	Assembly language
Encoding 0's and 1's: binary	Writing in textual form
Copy the value from "Register 9" into "Register 3."	
1110 0001 1010 0000 0011 0000 0000 1001	MOV R3, R9
Assembler	

- ISA: The design of the machine language encoding

4 ARM Instruction Set

This chapter describes the ARM instruction set.

ARM Instruction Set

4.1 Instruction Set Summary

4.1.1 Format summary

The ARM instruction set formats are shown below.

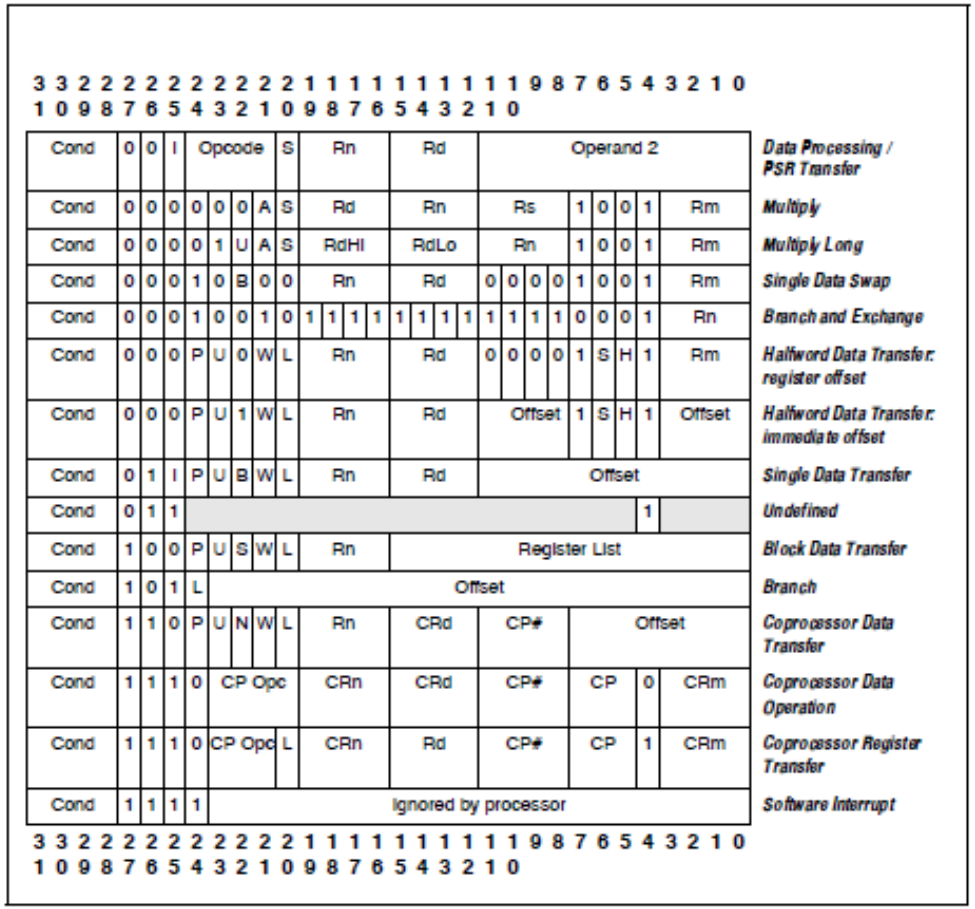


Figure 4-1: ARM instruction set formats

Note *Some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for instance a Multiply instruction with bit 6 changed to a 1. These instructions should not be used, as their action may change in future ARM implementations.*

Instruction	Binary format	HEX format
MOV R5, #0x12	$1110\_0011\_1010\_0000\_0101\_0000\_0001\_0010_2$	0xE3A0_5012

4.1.2 Instruction summary

Mnemonic	Instruction	Action	See Section:
MOV	Move register or constant	Rd : = Op2	4.5

Instruction	Binary format	HEX format
MOV R5, #0x12	$1110\_0011\_1010\_0000\_0101\_0000\_0001\_0010_2$	0xE3A0_5012

4.2 The Condition Field

In ARM state, all instructions are conditionally executed according to the state of the CPSR condition codes and the instruction's condition field. This field (bits 31:28) determines the circumstances under which an instruction is to be executed. If the state of the C, N, Z and V flags fulfils the conditions encoded by the field, the instruction is executed, otherwise it is ignored.

There are sixteen possible conditions, each represented by a two-character suffix that can be appended to the instruction's mnemonic. For example, a Branch (B in assembly language) becomes BEQ for "Branch if Equal", which means the Branch will only be taken if the Z flag is set.

In practice, fifteen different conditions may be used: these are listed in ***Table 4-2: Condition code summary***. The sixteenth (1111) is reserved, and must not be used.

In the absence of a suffix, the condition field of most instructions is set to "Always" (sufix AL). This means the instruction will always be executed regardless of the CPSR condition codes.

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Table 4-2: Condition code summary

Instruction	Binary format	HEX format
MOV R5, #0x12	$1110\_0011\_1010\_0000\_0101\_0000\_0001\_0010_2$	0xE3A0_5012

1	1	1	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1	0

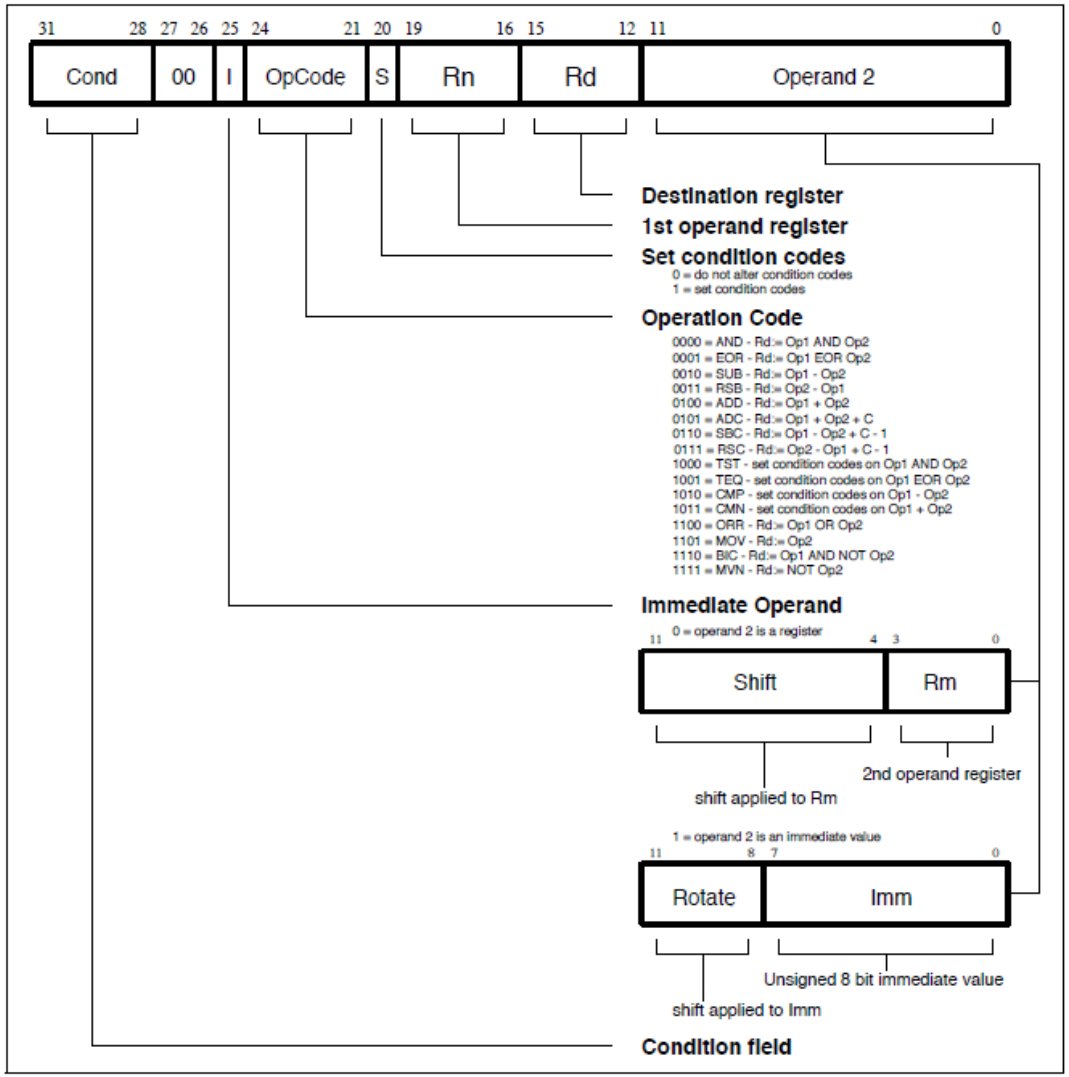


Figure 4-4: Data processing instructions

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

Registers

Register	Value
Core	
R0	0x10000000
R1	0x00000019
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000

Disassembly

```
0x00000104 4[illegible]      LDR     r0,[pc,[illegible]]   ; @0x00000128
0x00000106 F04F0119  MOV     r1,#0x19
0x0000010A 6001      STR     r1,[r0,#0x00]
0x0000010C F04F010A  MOV     r1,#0x0A
0x00000110 F04F0005  MOV     r0,#0x05
0x00000114 B500      PUSH    {lr}
0x00000116 F000F803  BL.W    0x00000120
0x0000011A F85DEB04  LDR     lr,[sp],#0x04
0x0000011E E7FE      B       0x0000011E
0x00000120 EB000201  ADD     r2,r0,r1
0x00000124 46F7      MOV     pc,lr
0x00000126 BF00      NOP
0x00000128 0000      DCW     0x0000
0x0000012A 1000      DCW     0x1000
0x0000012C 0000      DCW     0x0000
```

Addr	Contents
0x2000.0000	.
.	.
.	.
.	
0x2009.C040	1 byte
0x2009.C041	.
0x2009.C042	.
0x2009.C043	.
.	.
.	.
.	.
.	.
0x2009.C014	
0x2009.C015	
0x2009.C016	
0x2009.C017	

Check the following concepts from this picture

- ISA (assembly → machine code)
- ARM instruction (32 bits, 4 bytes)
- Thumb instruction (16 bits, 2 bytes)
- Byte addressable memory
- Etc …..

Instruction **S**et **A**rchitecture: **ISA**

An ISA specifies,

1. Data types
2. Syntax
3. Registers (general purpose, special…)
4. Supported instruction: Opcode
5. Memory access mechanisms: LDR and STR

ISA: provides a manual of assembly programming for a specific micro-processor/controller

ISA and Instructional Set summary

https://en.wikipedia.org/wiki/Instruction_set_architecture

http://www.keil.com/support/man/docs/armasm/armasm_dom1361289850509.htm

1. Data types

- In ARM® Cortex®-M3,4 system, all data are categorized into the following data types:
  - A 32-bit data item is called a **Word** and its length is 4 bytes.
  - A 16-bit data item is called a **Half-Word** and its length is 2 bytes.
  - An 8-bit data item is called a **Byte** and its length is 1 byte.

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/CHDIJJHE.html

https://developer.arm.com/docs/dui0553/latest/the-cortex-m4-processor/programmers-model/data-types

2. Syntax

ARM Instruction Syntax

- In linguistics, syntax (/ˈsɪnˌtæks/) is the set of rules, principles, and processes that govern the structure of sentences in a given language, **specifically word order**.
- Assembly language instructions format have four fields separated by spaces or tabs

Label	Opcode	Operands	Comment
Func	**MOV**	**R0, #100**	**; this sets R0 to 100**
	BX	**LR**	**; this is a function return**

- **Label Field**: A label is an identifier to provide a symbolic memory reference (i.e, branch instruction's address or a symbol for a constant)
- **Operation (op code) Field**: Instructions
- **Operands Field:** The operands field contains the data or an address for its corresponding instruction to be operated or performed
- **Comment Field:** This field enables users to place some comments

- A list of **symbols** to describe **assembly instructions**

Rd Rn represent registers
{Rd,} represents an optional destination register
#imm12 represents a 12-bit constant, 0 to 4095
{cond} represents an optional logical condition
{S} sets the condition code bits

- i.e., the general description of the addition

```
ADD{cond} {Rd,} Rn,   #imm12
ADD            R0,   R1,  #10        ; R0=R1+10
```

- **{ @#$%%}: optional (inside of parenthesis)**

```
ADD{cond} {Rd,} Rn,   #imm12
ADD            R0,   R1,  #10        ; R0=R1+10
```

Examples: Variants of the ADD instruction

```
ADD r1, r2, r3     ; r1 = r2 + r3
ADD r1, r3         ; r1 = r1 + r3
ADD r1, r2, #4     ; r1 = r2 + 4
ADD r1, #15        ; r1 = r1 + 15
```

- And we have condition codes to be used with

0. EQ	equal	Z
1. NE	not equal	!Z
2. CS or HS	carry set / unsigned higher or same	C
3. CC or LO	carry clear / unsigned lower	!C
4. MI	minus / negative	N
5. PL	plus / positive or zero	!N
6. VS	overflow set	V
7. VC	overflow clear	!V

8. HI	unsigned higher	C && !Z
9. LS	unsigned lower or same	!C || Z
10. GE	signed greater than or equal	N == V
11. LT	signed less than	N != V
12. GT	signed greater than	!Z && (N == V)
13. LE	signed less than or equal	Z || (N != V)
14. AL or omitted	always	true

- The condition code is usually followed at the end of the opcode

```
ADDEQ      R12,   R1,  #10      ; if Z=1, then R12=R1+10
```

- You will this condition flags soon (in PSR)

Ex: Add the numbers (1-10)

C code

```
int total;
int i;

total = 0;
for (i = 10; i > 0; i--) {
   total += i;
}
```

Assembly code supported by ARM ISA

```
        MOV   R0, #0        ; R0 accumulates total
        MOV   R1, #10       ; R1 counts from 10
                            ; down to 1
again   ADD   R0, R0, R1
        SUBS  R1, R1, #1
        BNE   again
halt    B     halt          ; infinite loop
                            ; to stop computation
```

3. Registers (general purpose, special…)

- Recap) Cortex-M4's core register structure

Size: 32bits

- 13 GP registers
- SP
- LR
- PC

R/W

- + special registers

R only

Special registers

PSR
PRIMASK
FAULTMASK
BASEPRI
CONTROL

Program status register

Exception mask registers

CONTROL register

General purpose registers

R0	
R1	
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13 (MSP)	R13 (PSP)
R14 (LR)	
R15 (PC)	

Stack pointer

Link register

Program counter

MSP – MAIN STACK POINTER

PSP – PROCESS STACK POINTER

Assembly Instruction types

- Arithmetic and logic
  - Add, Subtract, Multiply, Divide, Shift, Rotate
- Data movement
  - Load, Store, Move
- Compare and branch
  - Compare, Test, If-then, Branch, compare and branch on zero
- Miscellaneous
  - Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier

LDR and STR

- ONLY these two special instructions do this job
- Can access memory address

IMPORTANT CONCEPT to be checked

- To assign (represent) a number (constant) in a computer we need two parameters
  - Value (content)
  - Address

Addr	Contents
0x2000.0000	.
.	.
.	.
.	
0x2009.C040	1 byte
0x2009.C041	.
0x2009.C042	.
0x2009.C043	.
.	.
.	.
.	.
.	.
0x2009.C014	
0x2009.C015	
0x2009.C016	
0x2009.C017	

A memory map

Addr	Contents
0x2000.0000	.
.	.
.	.
.	
0x2009.C040	1 byte
0x2009.C041	.
0x2009.C042	.
0x2009.C043	.
.	.
.	.
.	.
.	.
0x2009.C014	
0x2009.C015	
0x2009.C016	
0x2009.C017	

- A structure of data that indicates how memory is laid out

```
Value1 EQU 0x2009.C040
LDR   R1, = Value1     (1)
LDR   R0,  [R1]        (2)
```

PC Relative Addressing

(1) R1

0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0

0x2009.C040 Value1 is loaded to R1

- The number loaded on R1 ***is interpreted as*** **<u>an address</u>** on **the memory map**

(2) [R1] : memory access instructions (LDR, STR) with square brackets

- Search for the specific address on **the memory map**, grab the values on the address
- [R1]: a value that R1 is pointing to on **the memory map**
- Therefore LDR R0, [R1] will bring us…

R0

?	?	?	?	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	?	?	?	?	?	?

Don’t know the contents in Value1, but this instruction will grab data in that address

LDR R1, = Value1

R1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | . | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Value1 0x2009.C040

LDR R0, [R1]

R1: address on memory map

[R1] : values allocated in the address

Memory map

Addr	Contents
0x2000.0000	.
.	.
.	.
.	.
0x2009.C040	1 byte
0x2009.C041	.
0x2009.C042	.
0x2009.C043	.
.	
.	
.	
.	.
0x2009.C014	
0x2009.C015	
0x2009.C016	
0x2009.C017	

R0 | ? | ? | ? | ? | ? | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | ? | ? | ? | ? | ? | ?

Don't know the contents in Value1, but this instruction will grab data in that address

LDR R0, [R1]

When we store it back

STR R0, [R1]

R0: Updated value

[R1] : values allocated in the address

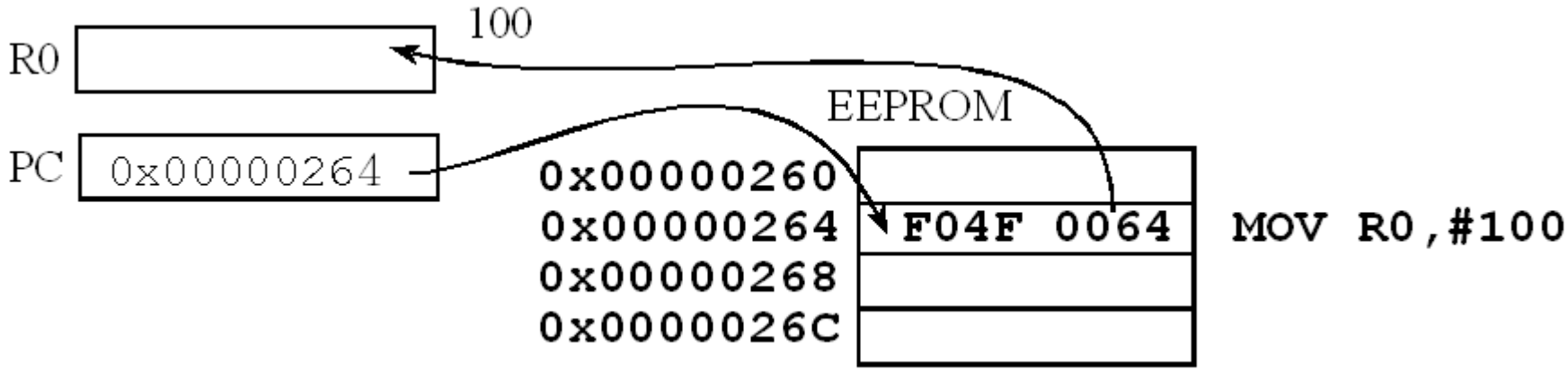
R0

Addr	Contents
0x2000.0000	.
.	.
.	.
.	.
0x2009.C040	1 byte
0x2009.C041	.
0x2009.C042	.
0x2009.C043	.
.	.
.	.
.	.
.	.
0x2009.C014	
0x2009.C015	
0x2009.C016	
0x2009.C017	

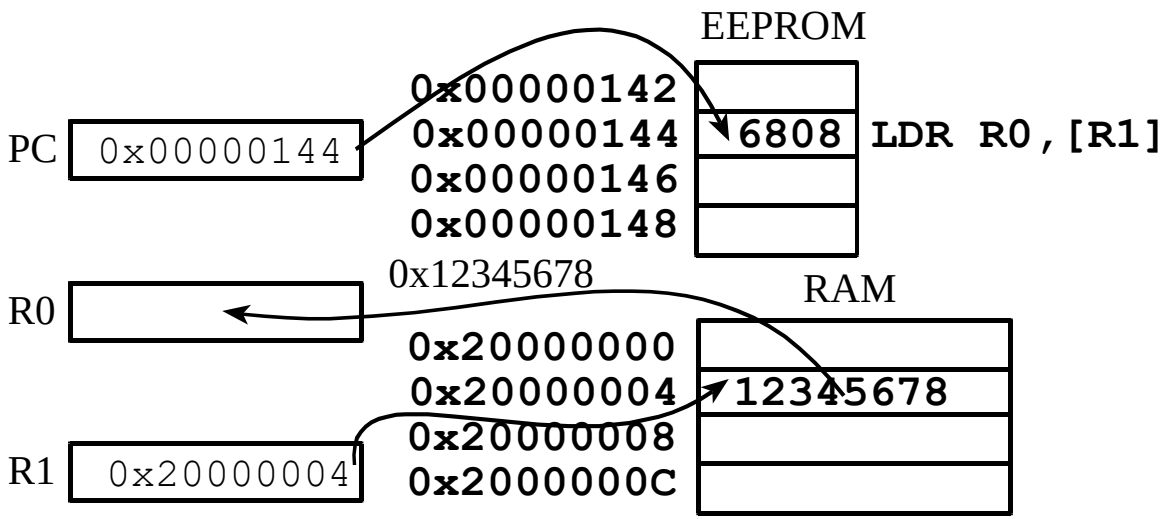
```
LDR   R0,  [ R1 ]

STR   R0,  [ R1 ]
```

Test differences

```
MOV   R0,   #100        ; R0=100, immediate addressing
```




```
LDR   R0,   [R1]        ; R0= value pointed to by R1
```



Assembly example

Code Area

```
        AREA string_copy, CODE, READONLY
        EXPORT  __main
        ALIGN
        ENTRY
__main  PROC

strcpy  LDR   r1, =srcStr       ; Retrieve address of the source string
        LDR   r0, =dstStr       ; Retrieve address of the destination string
loop    LDRB  r2, [r1], #1      ; Load a byte & increase src address pointer
        STRB  r2, [r0], #1      ; Store a byte & increase dst address pointer
        CMP   r2, #0            ; Check for the null terminator
        BNE   loop              ; Copy the next byte if string is not ended
stop    B     stop              ; Dead loop. Embedded program never exits.

        ENDP
```

Data Area

```
        AREA myData, DATA, READWRITE
        ALIGN

srcStr  DCB   "The source string.",0       ; Strings are null terminated
dstStr  DCB   "The destination string.",0  ; dststr has more space than srcstr

        END
```

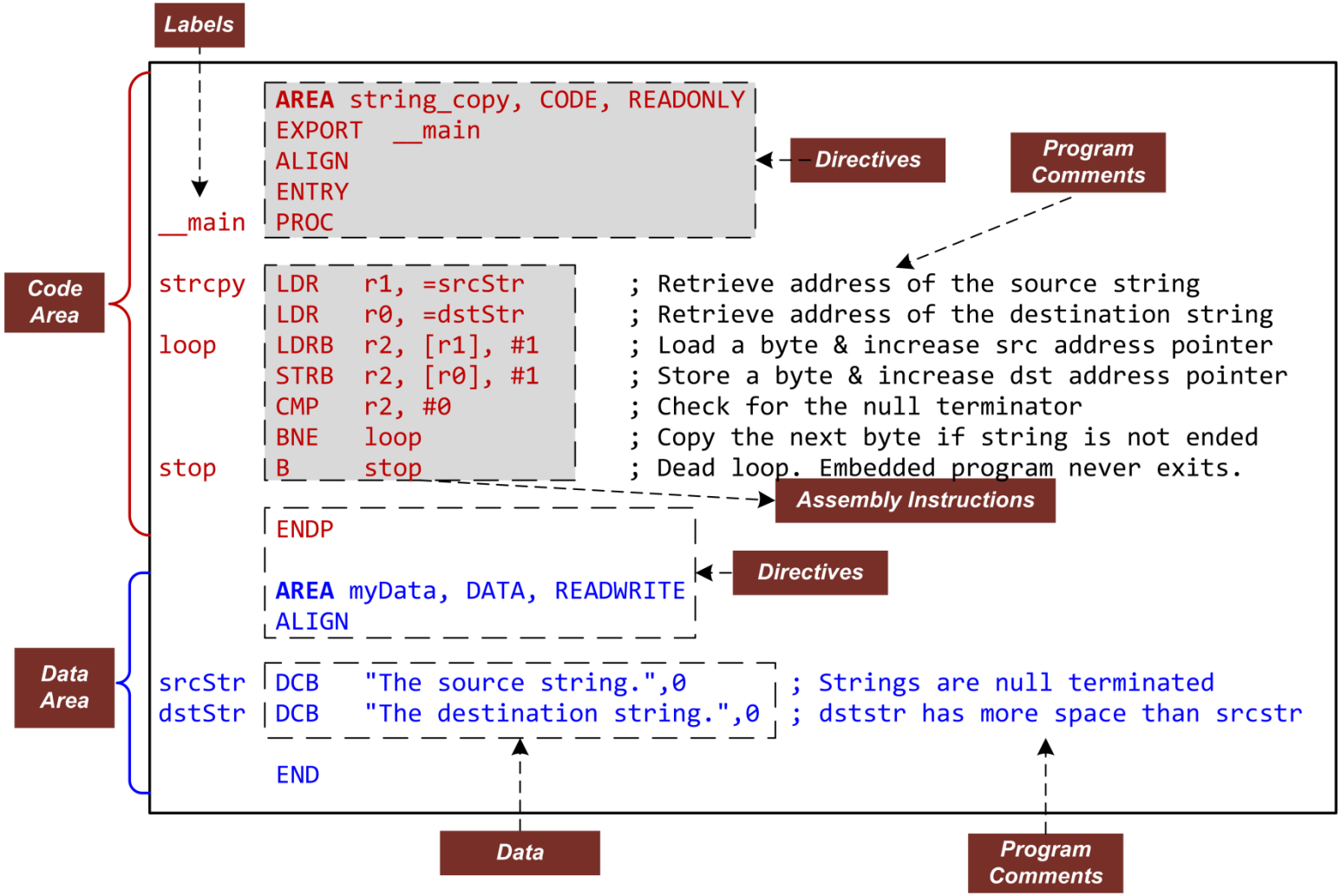
Code Area

Data Area

Program Comments

```
        AREA string_copy, CODE, READONLY
        EXPORT  __main
        ALIGN
        ENTRY
__main  PROC

strcpy  LDR   r1, =srcStr      ; Retrieve address of the source string
        LDR   r0, =dstStr      ; Retrieve address of the destination string
loop    LDRB  r2, [r1], #1     ; Load a byte & increase src address pointer
        STRB  r2, [r0], #1     ; Store a byte & increase dst address pointer
        CMP   r2, #0           ; Check for the null terminator
        BNE   loop             ; Copy the next byte if string is not ended
stop    B     stop             ; Dead loop. Embedded program never exits.

        ENDP

        AREA myData, DATA, READWRITE
        ALIGN

srcStr  DCB   "The source string.",0        ; Strings are null terminated
dstStr  DCB   "The destination string.",0   ; dststr has more space than srcstr

        END
```

Program Comments


Labels
Code Area
Directives
Program Comments
Assembly Instructions
Directives
Data Area
Data
Program Comments
AREA string_copy, CODE, READONLY
EXPORT __main
ALIGN
ENTRY
__main PROC
strcpy LDR r1, =srcStr ; Retrieve address of the source string
LDR r0, =dstStr ; Retrieve address of the destination string
loop LDRB r2, [r1], #1 ; Load a byte & increase src address pointer
STRB r2, [r0], #1 ; Store a byte & increase dst address pointer
CMP r2, #0 ; Check for the null terminator
BNE loop ; Copy the next byte if string is not ended
stop B stop ; Dead loop. Embedded program never exits.
ENDP
AREA myData, DATA, READWRITE
ALIGN
srcStr DCB "The source string.",0 ; Strings are null terminated
dstStr DCB "The destination string.",0 ; dststr has more space than srcstr
END

Directive

- In computer **programming**, a **directive** pragma (from "pragmatic") is **a language construct** that specifies how a compiler (or assembler or interpreter) should process its input.
- http://www.keil.com/support/man/docs/armasm/armasm_dom1361290002364.htm
- Find 7 directives used in our given example: EQU, AREA, SPACE, GLOBAL, ALIGN, ENTRY, END
  - I can show you some examples

- Directives are **NOT** instruction. Instead, they are used to provide key information for assembly.

AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a particular memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more half-words (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
SPACE	Allocate a zeroed block of memory with a particular size
FILL	Allocate a block of memory and fill with a given value.
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file

Directive: AREA

```
value1 EQU 0x05                ;
    AREA MyData, DATA, READWRITE
value2  SPACE   4

    GLOBAL __main               ; Glob
    AREA Main, CODE, READONLY ; Area
    ALIGN   2                  ; Align
    ENTRY
```

- Indicating to the assembler the start of a new data or code section.
- Areas are the basic independent and indivisible unit processed by the linker.
- Each area is identified by a name
  - Areas within the same source file cannot share the same name.
- An assembly program must have at least one code area.
  - By default, a code area can only be read (RO)
  - A data area can read from and written to (RW)

Data Area

Code Area

```
value1 EQU 0x05                ;
    AREA MyData, DATA, READWRITE
value2  SPACE   4

    GLOBAL __main                ; Glob
    AREA Main, CODE, READONLY ; Area
    ALIGN   2                   ; Align
    ENTRY

__main
    LDR  R0, =value2        ; Load "add
    LDR  R1, =25            ; Load the
    STR  R1, [R0]           ; Store the
    MOV  R1, #2_1010        ; Move the
    MOV  R0, #value1        ; Move valu

    BL   __add_register     ; Branch W
                            ; Saves the

__loop
    B    __loop              ; Branch t

__add_register          ; "Function" I
    ADD  R2, R0, R1         ; R2 = R0 +
    MOV PC, LR

    NOP ; No OPeration, just use NOP
    END ; End of file
```

ENTRY

```
value1 EQU 0x05                ;
    AREA MyData, DATA, READWRITE
value2  SPACE   4

    GLOBAL __main               ; Glob
    AREA Main, CODE, READONLY ; Area
    ALIGN   2                  ; Align
    ENTRY
```

- Marking the first instruction to be executed within an application.
- There must be one and only one entry directive in an application, no matter how many source files the application has.

END

- Indicating the end of a source file.
- Each assembly program must end with this directive.

```
29          END ; End of file
```

Data Allocation: SPACE

- Defines Zeroed Bytes (but actually not… Please test it)

EQU

- Associating a symbolic name to a numeric constant.
- Similar to the use of #define in a C program, the EQU can be used to define a constant (as well as address) in an assembly code.

Visit the following to see summary ISA

- http://www.keil.com/support/man/docs/armasm/armasm_dom1361289850509.htm

v.keil.com/support/man/docs/armasm/armasm_dom1361289850509.htm

Product Manuals

Document Conventions

Assembler User Guide

- Preface
- Overview of the Assembler
- Overview of the ARM Architecture
- Structure of Assembly Language Modules
- Writing ARM Assembly Language
- Condition Codes
- Using the Assembler
- Symbols, Literals, Expressions, and Operators
- VFP Programming
- Assembler Command-line Options
- ARM and Thumb Instructions
  - **ARM and Thumb instruction summary**
  - Instruction width specifiers
  - Flexible second operand (Operand2)
  - Syntax of Operand2 as a constant
  - Syntax of Operand2 as a register with optional shi
  - Shift operations
  - Saturating instructions
  - Condition code suffixes
  - ADC
  - ADD
  - ADR (PC-relative)
  - ADR (register-relative)
  - ADRL pseudo-instruction
  - AND
  - ASR

Different ARM architectures support different sets of ARM and Thumb instructions.

The following table gives a summary of the availability of ARM and Thumb instructions in different versions of the ARM architecture:

Table 10-1 Summary of ARM and Thumb instructions

Mnemonic	Brief description	*Arch.*
`ADC`	Add with Carry	All
`ADD`	Add	All
`ADR`	Load program or register-relative address (short range)	All
`ADRL` pseudo-instruction	Load program or register-relative address (medium range)	x6M
`AND`	Logical AND	All
`ASR`	Arithmetic Shift Right	All
`B`	Branch	All
`BFC`	Bit Field Clear	T2
`BFI`	Bit Field Insert	T2
`BIC`	Bit Clear	All
`BKPT`	Breakpoint	5
`BL`	Branch with Link	All
`BLX`	Branch with Link, change instruction set	T
`BX`	Branch, change instruction set	T

In class assembly programming practice

- Type and execute a working program
- Discuss section by section together