Introduction to ARM Assembly Language Programs

EE3002/ IM2003 Microprocessor Part 1

Chapter 3, Appendix B of textbook

Why Assembly Language?

- It is a course requirement to know assembly language programming.
- Learning assembly language will help you understand the ARM processor architecture better.
- Assembly language is a low-level language
- It allows you to write the most compact and efficient code.

Low Level vs High Level languages

- Application level language

  e.g. Matlab, Labview etc.
- Procedural language

  e.g. Fortran, Cobol, Basic, C++, etc

High Level
User Friendly

- Assembly language

  e.g. MOV r2, r1 ; MOV the contents of r1 to r2
- Machine Code

  e.g. 0xE1A02001 ??? Is it an instruction or data?

Low Level
Machine Friendly

Low Level Languages

- Low level languages are processor dependent while high level languages are processor independent.
- The first low level language is machine code which consist of binary numbers (or hexadecimal numbers). It is nearly impossible to decipher the program and very tedious to write.
- The assembly language is then invented. It uses English abbreviations to describe the operations. For example ADD, MOV, SUB, MUL to represent operations like addition, move, subtract and multiply respectively.
- There is a one-to-one mapping between assembly language and machine code.

High Level Language

- High level languages are not designed for a particular processor.
- A program written in high level language only requires minor changes (if any) to run on different platforms. An assembly language program need to be completely rewritten.
- Use of high-level languages makes programming easier as the user can concentrate on the logic of the problem to be solved instead of the intricacies of the processor architecture and other hardware details
- A single statement in high level language can correspond to many instructions in assembly language.

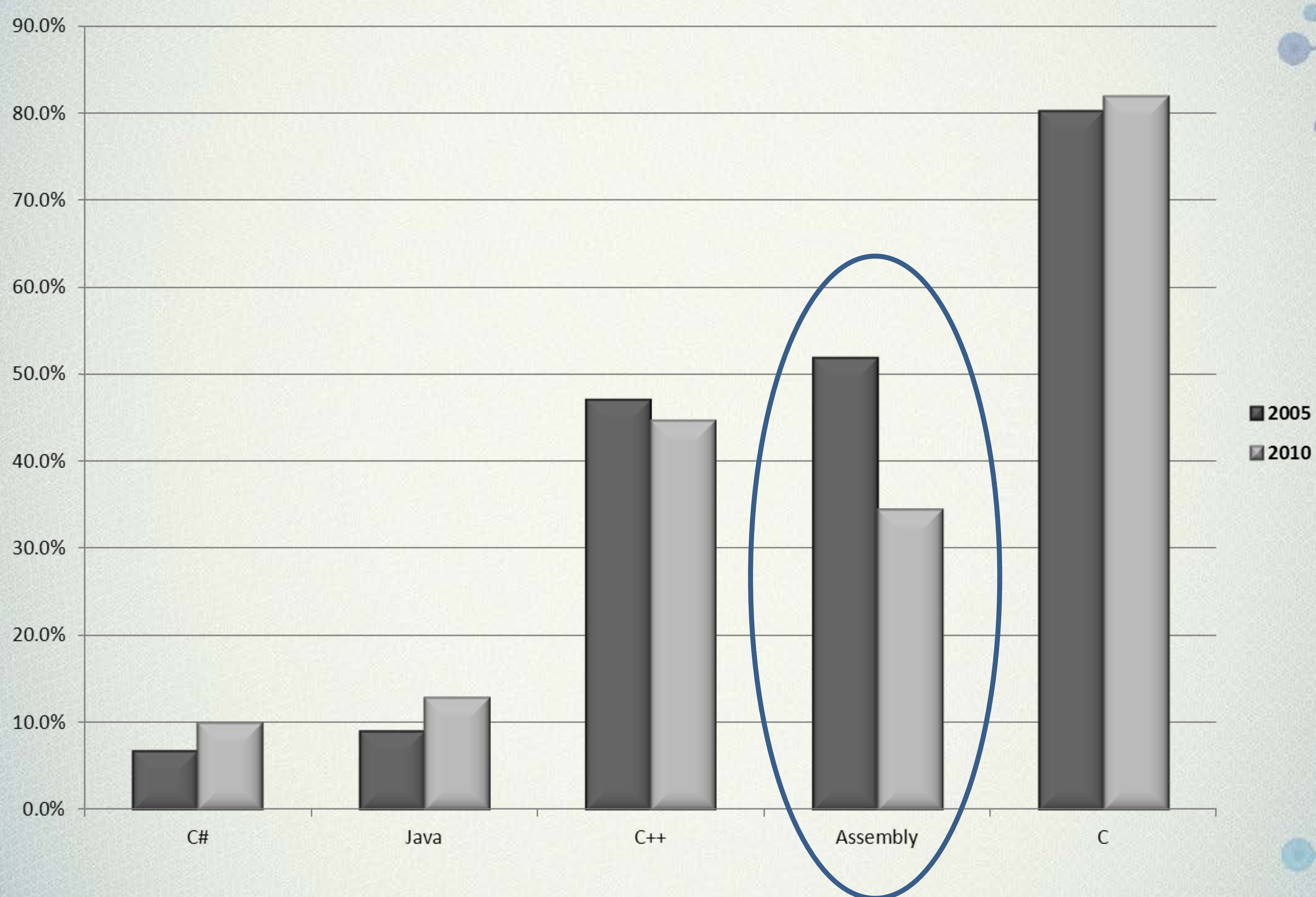
Advantages of Assembly language

- It allows you to have the most compact code size. (Refers to the executable file and NOT the source file). This is important when you are programming a microcontroller which has limited code space.
- It allows you to have the fastest or most efficient code. Writing in assembly language allows you to optimize the code according to the processor architecture. Sometimes this is important for real-time control.

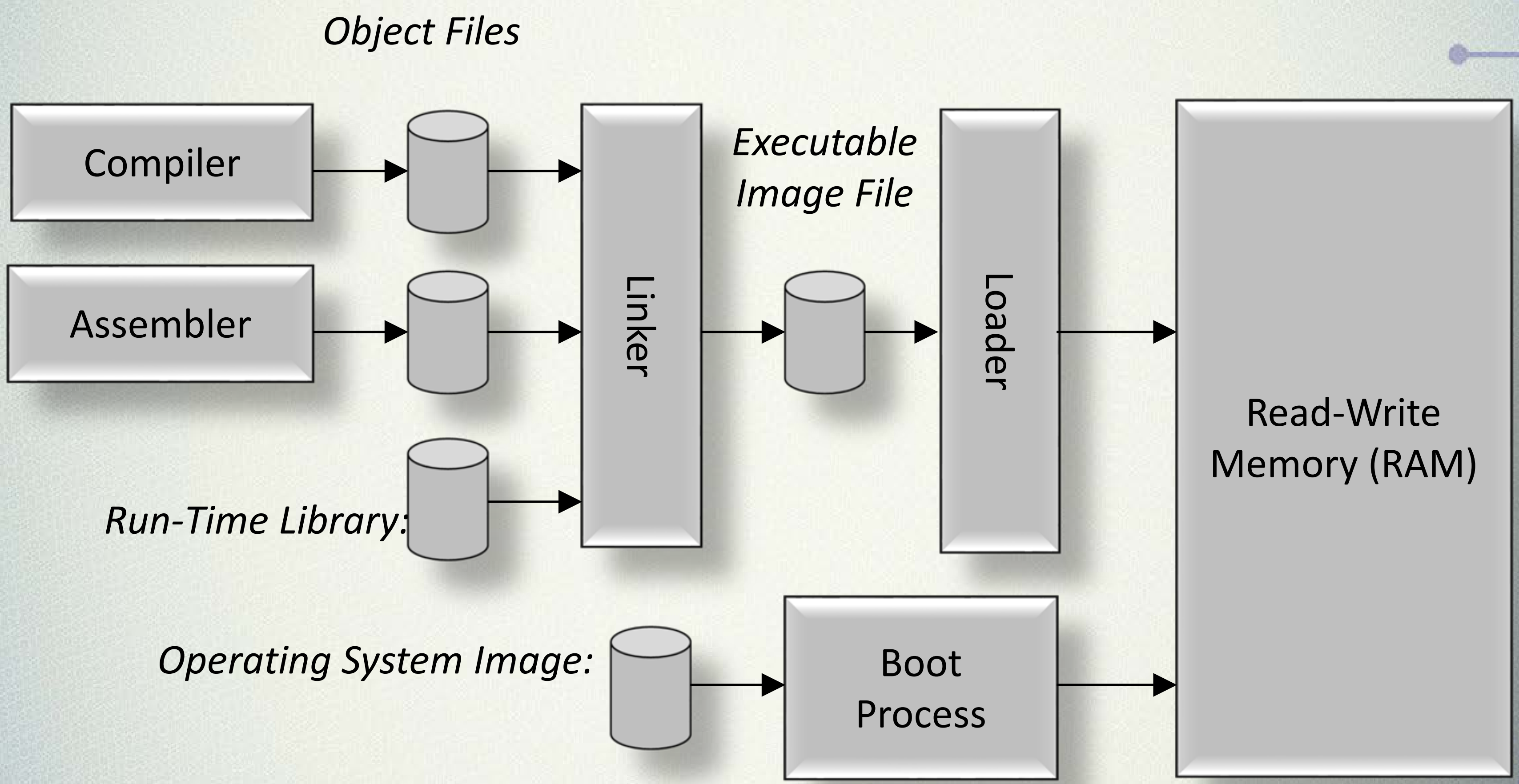
Disadvantages of Assembly Language

- It is difficult to write
  - You need to know the processor architecture well.
  - You have to translate your thoughts into action by using fixed assembly instructions, and sometimes many instructions are necessary for a simple task.
  - You have to rewrite the program if a different processor is used. Platform dependent.
- It is also difficult to read and understand.
  - Good assembly language programs normally has lots of comment lines to aid understanding.

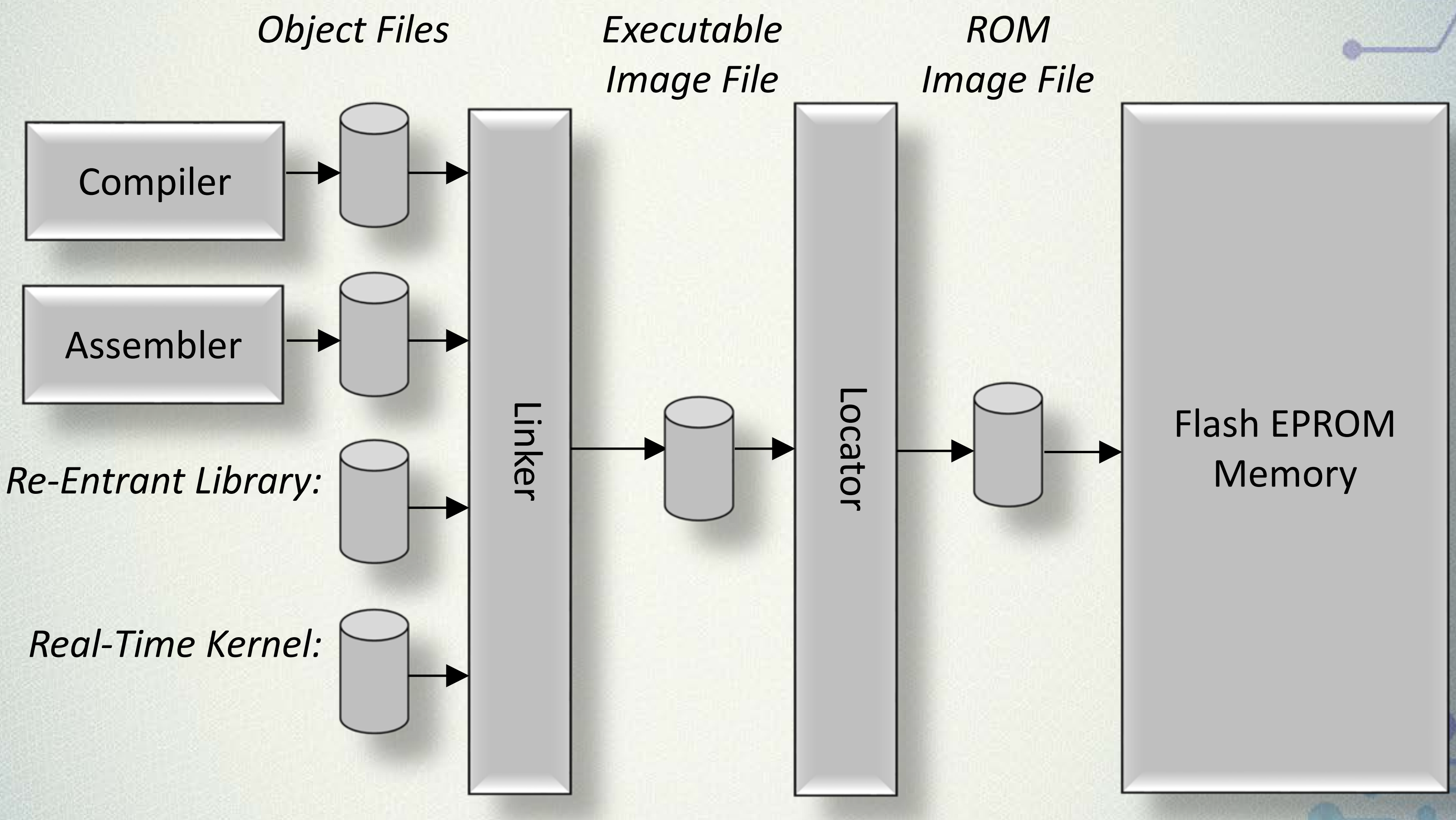
Languages used in embedded systems



Desktop Application Development



Embedded Application Development



Keil µVision 4

- Keil µVision 4 is a Integrated Development Environment (IDE) tool.
- It allows you to edit, assemble, debug and test your code.
- The best thing about it is it has a freeware version.
- You can download the program from the Edventure course site (recommended) or download the program from www.keil.com.
- If you use one of the older versions, you may have to change the linker settings otherwise you will encounter linker error.

First Assembly Program (Shifting Data)

```
AREA Prog1, CODE, READONLY
ENTRY
MOV   r0,  #0x11          ; load initial value
MOV   r1,  r0, LSL #1     ; shift 1 bit left
MOV   r2,  r1, LSL #1     ; shift 1 bit left
stop Bstop
END
```

Assembler Directives vs. Microprocessor Instructions

- Those Blue wordings are Assembler Directives which are instructions to the Assembler (a piece of software). These instructions are not converted into machine code and will never be executed by the processor. We will cover more of it later.
- The Green wordings are ARM microprocessor instructions. They are converted to machine code during assembly and are executed by the microprocessor when the program is run.

Assembler Directives

- AREA – for the assembler to create a block of code (in this case) in memory.
- CODE – The block created is a set of instructions (or program)
- READONLY – The block is read-only, which means it cannot be written into.
- AREA Prog1, CODE, READONLY means the assembler will create a block of code called Prog1 (just a name) which is readonly.
- ENTRY – it is the point where the program starts execution.
- END – End of program, the assembler will ignore all other instructions after the END statement.

ARM Instructions

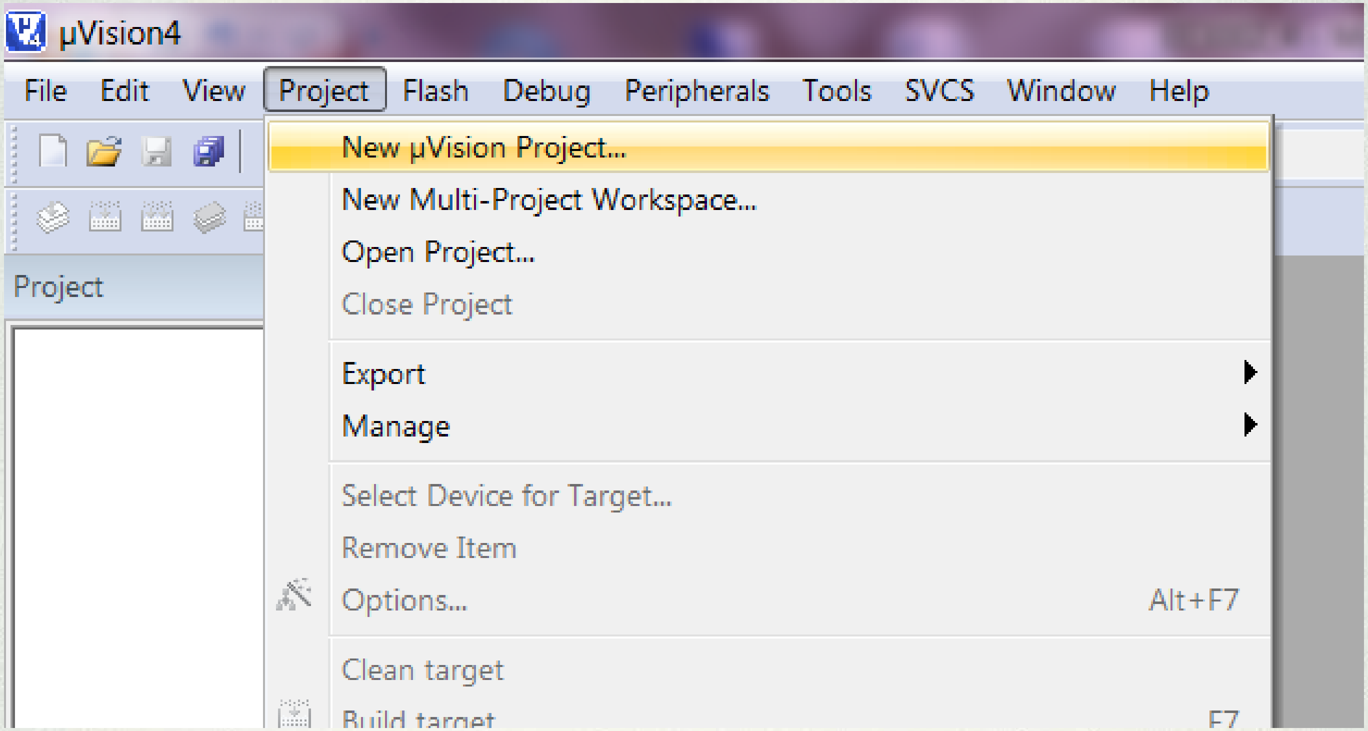
- MOV r0, #0x11
  - The # means that the constant is an immediate operand
  - This instruction MOVes 11 hexadecimal into register r0.
  - Now register r0 contains 0x11
  - Or binary 0000 0000 0000 0000 0000 0000 0001 0001
- MOV r1, r0, LSL #1
  - LSL stands for Logical Shift Left, the instruction will shift the contents of r0 1 bit to the left and place the result in register r1.
  - Now register r1 contains 0x22
  - Or binary 0000 0000 0000 0000 0000 0000 0010 0010
- What is the result in r2 after the last instruction ?

  MOV r2, r1, LSL #1

Stopping the program

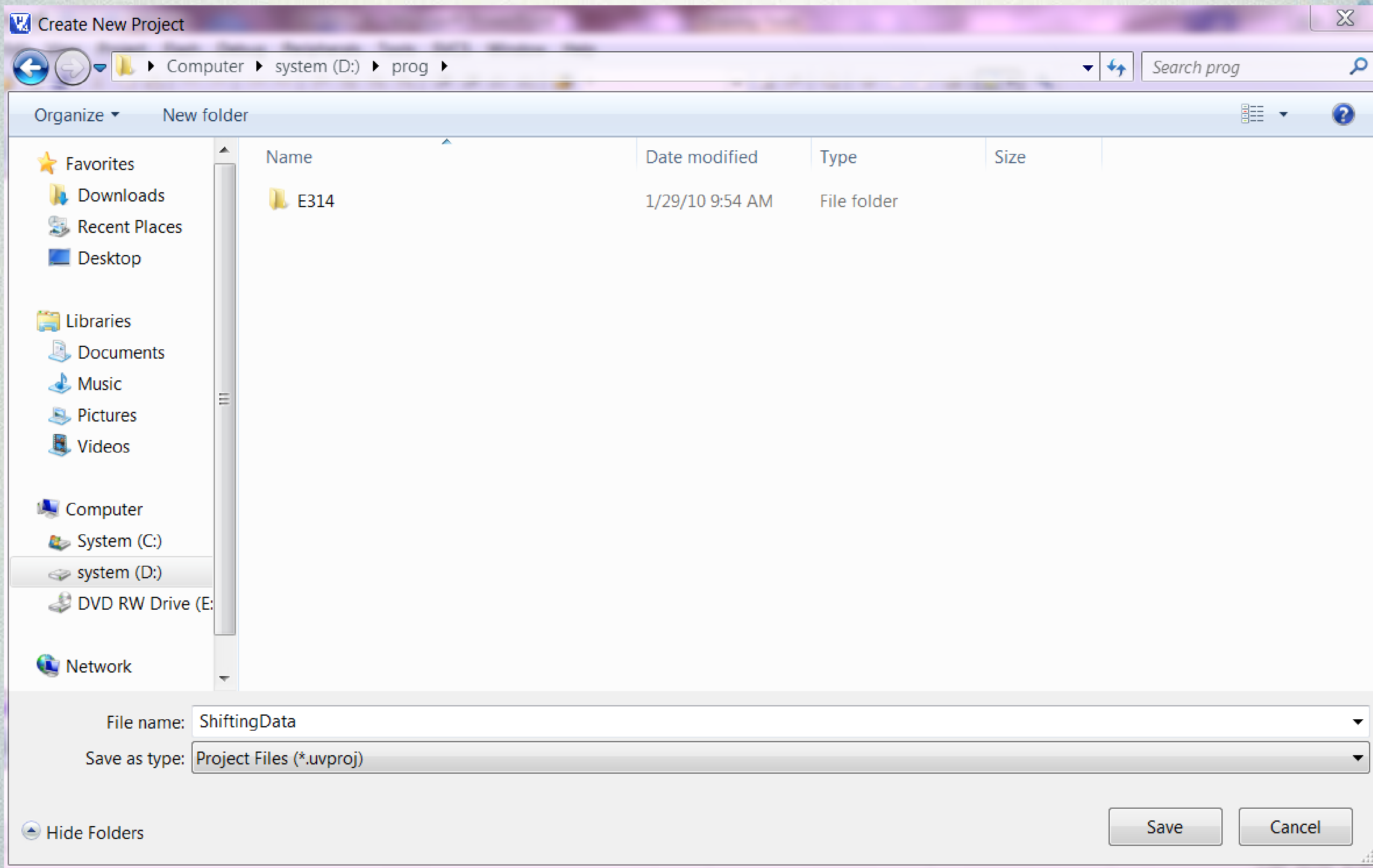
- stop   B   stop
  - stop is the name of a label, you can call it any other name, it will still work.
  - B is short for Branch
  - The instruction actually tells the processor to branch to the branch instruction itself, which puts the code into an infinite loop.
  - Very crude method but convenient as it allows us to terminate the simulation easily by choosing Start/Stop Debug Session from the Debug menu or clicking the Halt button in our Keil tools

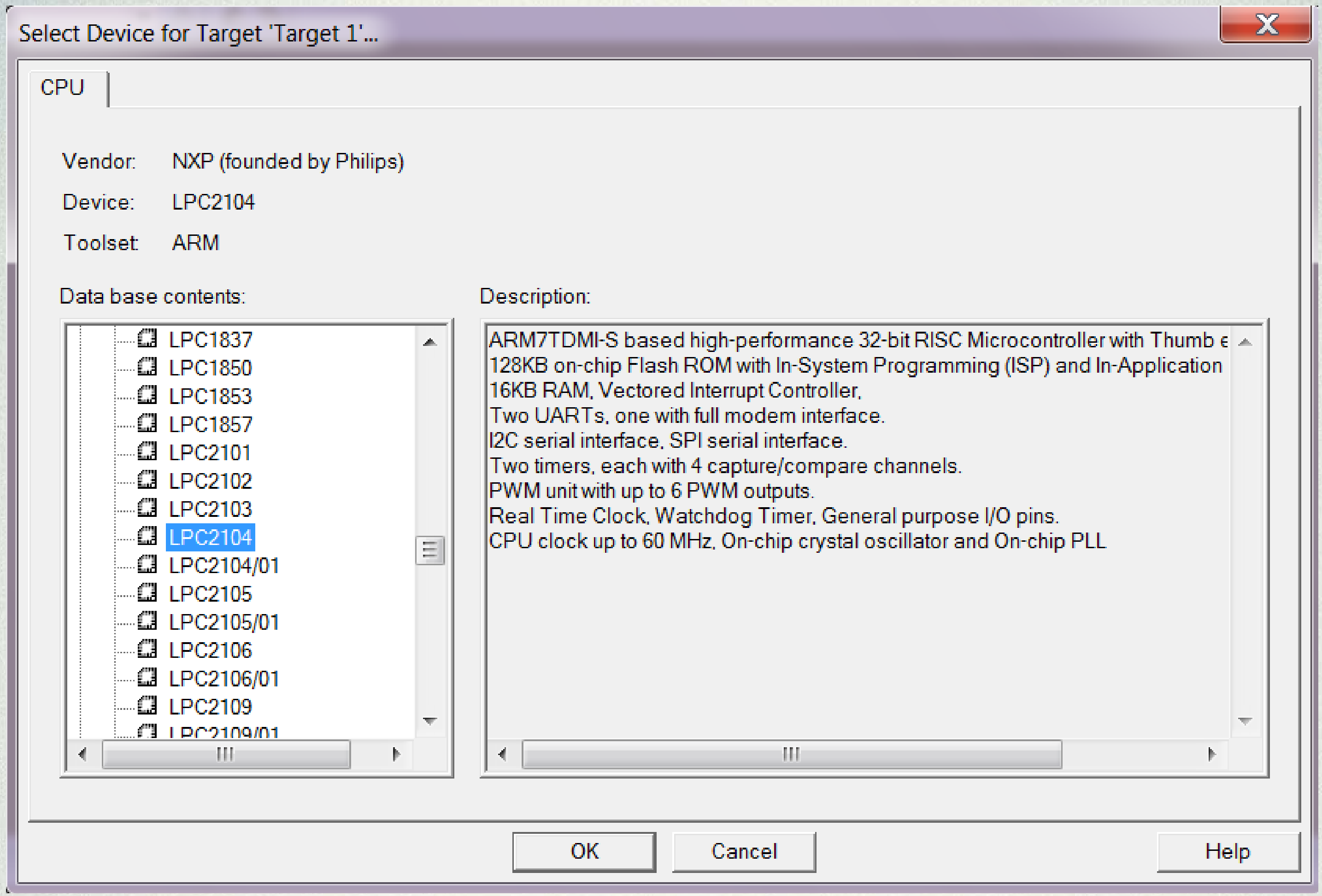
Getting Started with Keil µVision4



- Launch the Keil µVision 4 program
- Choose New µVision Project from Project menu.

Save The Project Titled : Shiftingdata





- Choose LPC2104 from NXP.
- When you click Ok, a dialog box will appear asking if you want to include startup code for this device. Click NO.

Creating Application Code

- From the **File** menu, choose **New** to create your assembly file with the editor.
- Type in the shifting data program given earlier on slide 12.
- Make sure that except for the stop label, all the other lines must have a space (gap) on the left.

The program should look as follows:

prog1.s

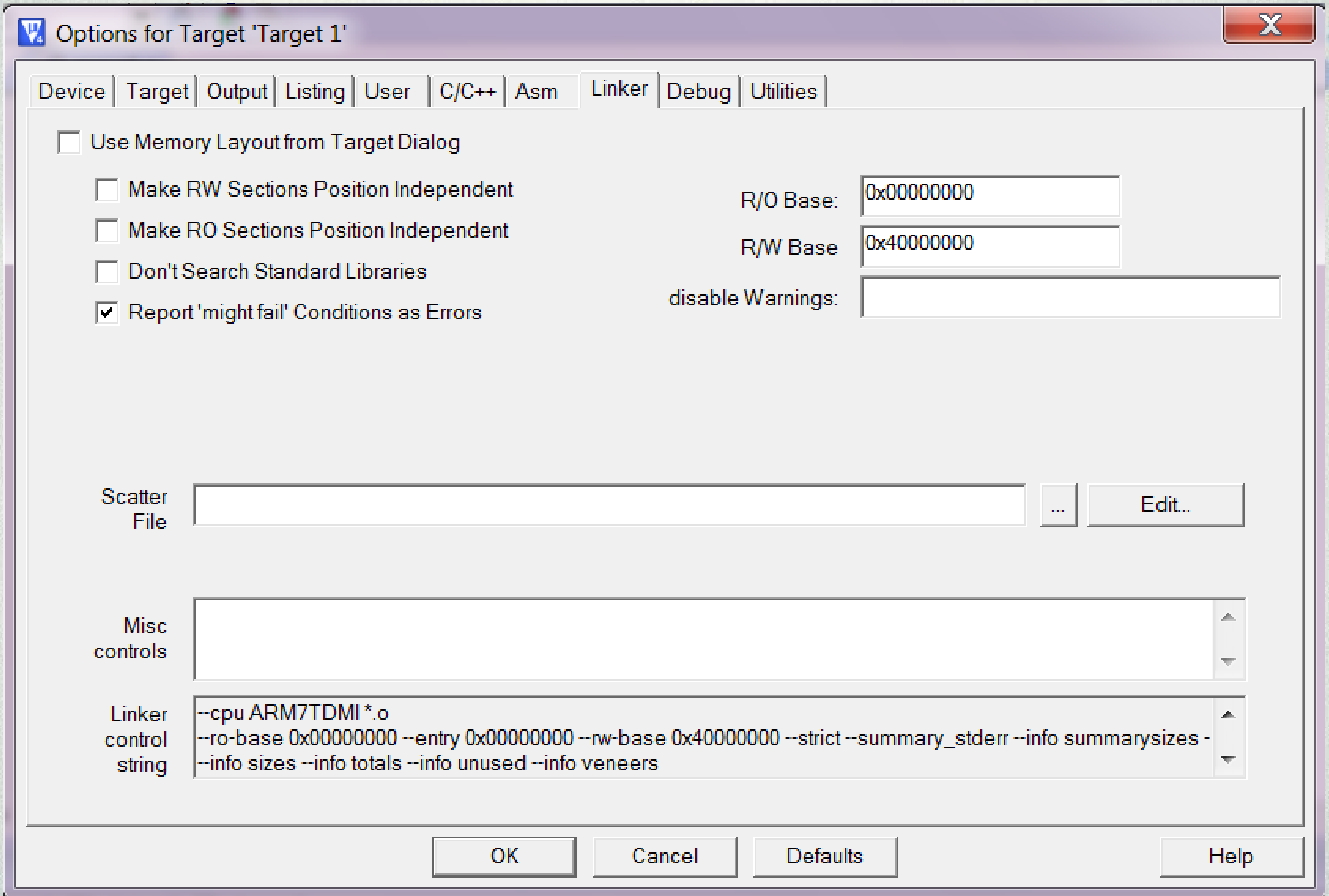
```
    AREA Prog1, CODE, READONLY
    ENTRY

    MOV r0, #0x11          ;load initial values
    MOV r1, r0, LSL #1     ;Shift 1 bit left
    MOV r2, r1, LSL #1     ;Shift 1 bit left

stop B stop                ;Stop Program
    END
```

- Choose **Save As** from the **File** menu and give it a name, such as **Prog1.s**
- The assembly file must be added to the project. In the Project Workspace window on the left, click on the plus sign to expand the Target 1 folder.
- Right click on the Source Group 1 folder, than choose **Add Files to Group 'Source Group 1'** to add **Prog1.s**

Change The Default Linker Settings(v4.6)

- This step is unnecessary if you are using the Edventure version.
- Right click **Target 1** and then **options for Target 1** and then click on the **Linker** Tab.
- Uncheck the box next to **Use Memory Layout from Target dialog** and delete the scatter file.

Linker Settings



Building the Project and Running Code

- Select **Rebuild all target files** from the **Project** menu.
- Now that the executable file has been produced, use the debugger for simulation.
- From the **Debug** menu, choose Start/Stop Debug Session.
- In **Debug** session, new windows such as Disassembly window, Register window and Memory window will be seen.
- Single-step through the code by clicking on the **Step Into** button or choose **Step** from the **Debug** menu or use the F11 key. Observe the changes in the register window.
- When finished, choose **Start/Stop Debug Session** again from the **Debug** menu

Disassembly Window

- It shows the machine code of the ARM instructions and their memory location.
- Note that the **stop** label had been converted into its memory address (0x0000000C)

Disassembly

```
      6:          MOV      r0, #0x11          ;load initial value
0x00000000  E3A00011  MOV        R0,#0x00000011
      7:          MOV r1, r0, LSL #1        ;shift 1 bit left
0x00000004  E1A01080  MOV        R1,R0,LSL #1
      8:          MOV r2, r1, LSL #1        ;shift 1 bit left
      9:
0x00000008  E1A02081  MOV        R2,R1,LSL #1
     10: stop B stop                       ;stop program
0x0000000C  EAFFFFFE  B          0x0000000C
0x00000010  00000000  ANDEQ      R0,R0,R0
0x00000014  00000000  ANDEQ      R0,R0,R0
0x00000018  00000000  ANDEQ      R0,R0,R0
0x0000001C  00000000  ANDEQ      R0,R0,R0
```

Register, Memory Windows

- In the Register Window, you can see the values of all the registers and watched the values change as you step through the program.
- In the Memory window, you can see the contents of the memory. Observe how the machine code of the ARM instruction is stored in the memory. ARM default uses little-endian which means that less significant byte is stored in lower address. More about endianess will be covered later.
- Learn to use breakpoints in the debug mode also.

Factorial Program

- Factorial of n or n! = n(n-1)(n-2)…(1)
- This program will introduce the topics of

– Conditional execution: The multiplication may or may not be performed, depending on the result of another instruction.

– Setting flags: The "S" suffix on an instruction directs the processor to update the flags in the CPSR based on the result of the operation

– Change-of-flow instructions: A branch will load a new address, called a branch target, into the Program Counter, and execution will resume from this new address.

```
      AREA Prog2,  CODE,  READONLY
      ENTRY
      MOV   r6, #10; load 10 into r6
      MOV   r4, r6        ;  copy n into a temp register
loop  SUBS  r4, r4, #1; decrement next multiplier
      MULNE  r7, r6, r4  ;perform multiply
      MOV    r6, r7
      BNE    loop  ;go again if not complete
stop  B      stop  ;stop program
      END
```

Explanation Of The Program

- The instruction "**MOV r6, #10**" moves the decimal (default) value of 10 into r6. The program computes the factorial of 10 and store the result in r6.
- The next **MOV** instruction copies the contents of r6 into r4, which also serves as the multiplier which is reduced by one every iteration until it reaches one.
- The next instruction "**SUBS r4, r4, #1**", performs r4 = r4 – 1 operation. The "**S**" at the end of the **SUB** instruction means the condition code flags will be set at the end of the instruction. If the result is 0, the Z (Zero) flag will be set (=1).

Explanation “Cont’d”

- The **MULNE** instruction multiplies r6 by r4 and put the result in r7, but only if (subtraction result in this case) not equal to zero. Note: when the subtraction result in zero, Z flag is set or Z flag =1, otherwise Z flag is 0 or not set).
- The third **MOV** instruction places the product into r6, which will become the final result eventually.
- The **BNE** works with a label that we placed above it called loop (can be replaced by any name, it is not a reserved word). When executed, the branch instruction changes the program counter to the address of its branch target but only if the result from the subtraction is non-zero (by testing the Z flag).

Exchange Register Contents

- This program exchanges the contents of two registers in a very elegant and efficient manner without the need of a temporary storage.
- Suppose two values **A** and **B** are to be exchanged. The following algorithm can be used:

A = A EOR B

B = A EOR B

A = A EOR B

where EOR denotes the Exclusive Or Operation

Exclusive Or Operation

- Revision of EOR
- A EOR A = 0
- A EOR 0 = A
- A EOR 1 = NOT(A)

A	B	A EOR B
0	0	0
0	1	1
1	0	1
1	1	0

Exchange Program

```
AREA Prog3,  CODE, READONLY
ENTRY
LDR    r0, =0xF631024C     ;load some data
LDR    r1, =0x17539ABD     ;load some data
EOR    r0, r0, r1          ; r0 = r0 EOR r1
EOR    r1, r0, r1          ; r1 = r0 EOR r1
EOR    r0, r0, r1          ; r0 = r0 EOR r1
stop   B   stop            ; stop program
       END
```

EOR Calculations

- 0xF631024C:
  – 1111 0110 0011 0001 0000 0010 0100 1100
- 0x17539ABD:
  – 0001 0111 0101 0011 1001 1010 1011 1101
- 0xF631024C EOR 0x17539ABD:
  – 1110 0001 0110 0010 1001 1000 1111 0001
- 0xE16298F1 (Result)

Explanation

- In this program two Pseudo-Instructions**, LDR** (load) are used to load the initial values into the registers. More will be covered later.
- Convince yourself by working through the example by hand that it really works.

ARM Instruction Set Summary (1/4)

Mnemonic	Instruction	Action
ADC	Add with carry	Rd:=Rn+Op2+Carry
ADD	Add	Rd:=Rn+Op2
AND	AND	Rd:=Rn AND Op2
B	Branch	R15:=address
BIC	Bit Clear	Rd:=Rn AND NOT Op2
BL	Branch with Link	R14:=R15 R15:=address
BX	Branch and Exchange	R15:=Rn T bit:=Rn[0]
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags:=Rn+Op2
CMP	Compare	CPSR flags:=Rn-Op2

ARM Instruction Set Summary (2/4)

Mnemonic	Instruction	Action
EOR	Exclusive OR	Rd:=Rn^Op2
LDC	Load Coprocessor from memory	(Coprocessor load)
LDM	Load multiple registers	Stack Manipulation (Pop)
LDR	Load register from memory	Rd:=(address)
MCR	Move CPU register to coprocessor register	CRn:=rRn{<op>cRm}
MLA	Multiply Accumulate	Rd:=(Rm*Rs)+Rn
MOV	Move register or constant	Rd:=Op2
MRC	Move from coprocessor register to CPU register	rRn:=cRn{<op>cRm}
MRS	Move PSR status/flags to register	Rn:=PSR
MSR	Move register to PSR status/flags	PSR:=Rm

ARM Instruction Set Summary (3/4)

Mnemonic	Instruction	Action
MUL	Multiply	Rd:=Rm*Rs
MVN	Move negative register	Rd:=~Op2
ORR	OR	Rd:=Rn OR Op2
RSB	Reverse Subtract	Rd:=Op2-Rn
RSC	Reverse Subtract with Carry	Rd:=Op2-Rn-1+Carry
SBC	Subtract with Carry	Rd:=Rn-Op2-1+Carry
STC	Store coprocessor register to memory	address:=cRn
STM	Store Multiple	Stack manipulation (Push)

ARM Instruction Set Summary (4/4)

Mnemonic	Instruction	Action
STR	Store register to memory	<address>:=Rd
SUB	Subtract	Rd:=Rn-Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd:=[Rn] [Rn]:=Rm
TEQ	Test bitwise equality	CPSR flags:=Rn EOR Op2
TST	Test bits	CPSR flags:=Rn AND Op2

Summary

- You should be able to edit, debug, step, run a simple assembly program in Keil µVision 4.
- Understand the pros and cons of writing in assembly language.
- Know the difference between Assembly directives and ARM instructions.
- Have some idea of the ARM instruction format.