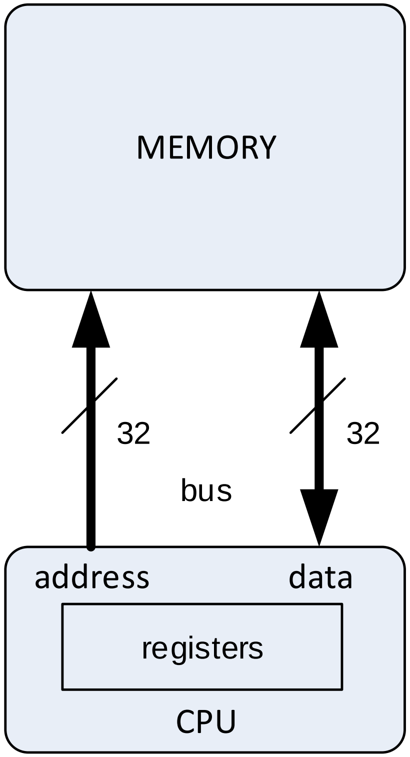
Simple Model of an ARM Microprocessor System

- comprises a central processing unit (CPU) and memory
- instructions and data are stored in memory
- the CPU reads instructions from memory (one after another) and executes them
- when the CPU executes an instruction
  - it performs operations between its registers OR
  - it reads data from memory and stores it in a register OR
  - it writes data from a register and stores it in memory



Memory

- memory comprises an array of memory locations
- each location stores a byte of data
- each location location has a unique 32 bit address 0x00000000 to 0xFFFFFFFF
- the address space, 2^{32} bytes (4GB), is the amount of memory that can be physically attached to the CPU
- the byte stored at each location may be part of an instruction (as each instruction is 4 bytes) **or** data

0xFFFFFFFF	0xFF
0xFFFFFFFE	0xEE
0xFFFFFFFD	0xDD
0xFFFFFFFC	0xCC
	⋮
0x00000005	0x05
0x00000004	0x04
0x00000003	0x11
0x00000002	0x22
0x00000001	0x33
0x00000000	0x44

memory as an array of BYTEs

Memory

- often easier to view memory as an array of WORDs (32 bits) rather than an array of BYTEs
- as each WORD location is aligned on a 4 byte boundary, the low order 2 bits of each address is 0
- making a comparison with the previous slide, the byte of data stored at memory location 0 is the least significant byte of the WORD stored in location 0
- this way of storing a WORD is termed LITTLE ENDIAN - the least significant byte is stored at the lowest address (the other way is BIG ENDIAN)
- ARM CPUs can be configured to be LITTLE ENDIAN or BIG ENDIAN (term from [Gulliver's Travels]())

0xFFFFFFFC	0xCCDDEEFF
0xFFFFFFF8	0xF8F8F8F8
	⋮
0x0000000C	0x87654321 0
0x00000008	0x8ABCDEF0
0x00000004	0x12345678
0x00000000	0x11223344

memory as an array of WORDs

ARM CPU Registers

- the ARM CPU contains 16 x 32bit registers R0 to R15
- data can be read from memory and stored in a register
- data in a register can be written to memory
- arithmetic operations can be performed between the registers

```
ADD   R0, R1, R2          ; R0 = R1 + R2
```

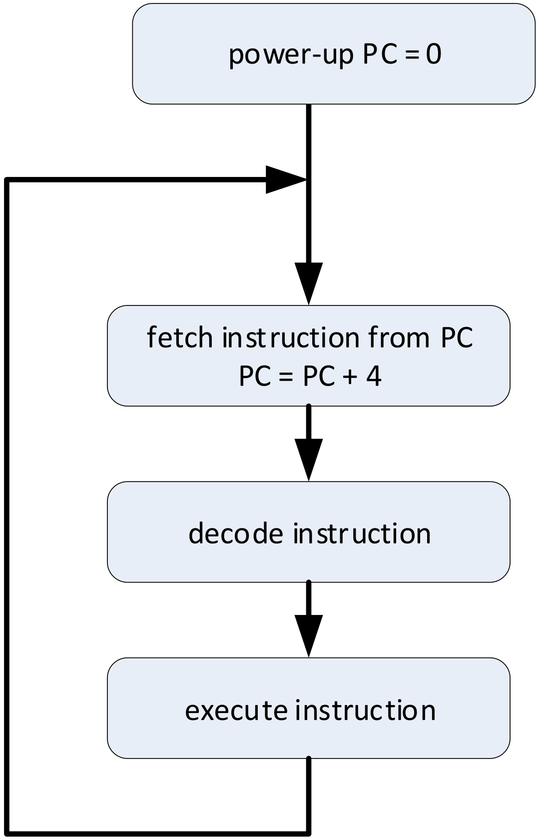
- R0 to R12 are considered general purpose registers
- R13, R14 and R15 are specialised
- registers are far quicker to access than memory

← 32 bits →	
R0	
R1	
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13 (SP)	stack pointer
R14 (LR)	link register
R15 (PC)	program counter

ARM CPU registers

Program Execution

- the CPU continuously fetches, decodes and executes instructions stored in memory at the address specified by the Program Counter (R15 or PC)
- on power-up, the PC is initialised to 0 so that the first instruction executed is a memory address 0
- after fetching each instruction, 4 is added to the PC so that the PC contains the address of the next sequential instruction (ALL instructions are 4 bytes)
- CPU keeps fetching, decoding and executing instructions until it is switched off



ARM data processing instructions

- consider the following ARM assembly language instructions

```
ADD  - add
SUB  - subtract
RSB  - reverse subtract
MOV – move
MUL  - multiply
```

- three address instructions, need to specify dst, src1 and src2 registers

```
ADD    R0, R1, R2    ; R0 = R1 + R2 (R0:dst R1:src1 R2:src2)
SUB    R0, R1, R2    ; R0 = R1 – R2
RSB    R0, R1, R2    ; R0 = R2 – R1
MOV    R0, R1        ; R0 = R1 (makes a copy of R1, src1 ignored)
MUL    R0, R1, R2    ; R0 = R1 * R2 (NB: dst and src1 registers cannot be the same)

ADD    R0, R0, R0    ; R0 = R0 + R0
```

Immediate src2 Operand

- the src2 operand can be a register **<u>OR</u>** a constant value
- there are limitations to the constant values that can "fit" in src2 field (these will be explained later)
- the fall-back position is to use a LDR instruction as it can load a register with any 32bit constant (also explained later)

```
ADD  R0, R1, #1        ; R0 = R1 + 1
ADD  R2, R3, #0x0F     ; R2 = R3 + 0x0F
SUB  R1, R1, #2        ; R1 = R1 – 2
MOV  R0, #3            ; R0 = 3
```

- note the # symbol – means an immediate constant
- MUL instruction is an exception, src2 cannot be an immediate constant

ARM LDR instruction

- LDR can be used to load an immediate (or constant) value into a register

```
LDR    R0, =0x1234             ; R0 = 0x1234
```

- = symbol for an immediate LDR operand, other instructions use the # symbol
- ; indicates the start of a comment
- LDR is not quite what it seems, explained in more detail later

ARM data processing example

- if x = 50, compute $x^2 + 10*x - 3$

- need to decide how best to use the registers
- compute result in R0
- use R1 to hold x
- use R2 as a temporary register for performing the computation

```
MOV  R1, #50        ; R1 = x = 50
MUL  R0, R1, R1     ; R0 = x²
MOV  R2, #10        ; R2 = 10
MUL  R2, R1, R2     ; R2 = 10x (MUL R2, R2, R1 would not work)
ADD  R0, R0, R2     ; R0 = x² + 10x
SUB  R0, R0, #3     ; R0 = x² + 10x - 3
```

- work around limitations of MUL instruction
  - dst and src1 cannot be the same register
  - src2 cannot be an immediate constant

Assembly Language => Machine Code Example

- assembly language instructions are converted into machine code by the assembler
- the CPU fetches, decodes and executes machine code instructions stored in memory
- each machine code instruction is 4 bytes (32 bits)
- the 32-bit machine code instruction encodes the operation (eg ADD) and operands

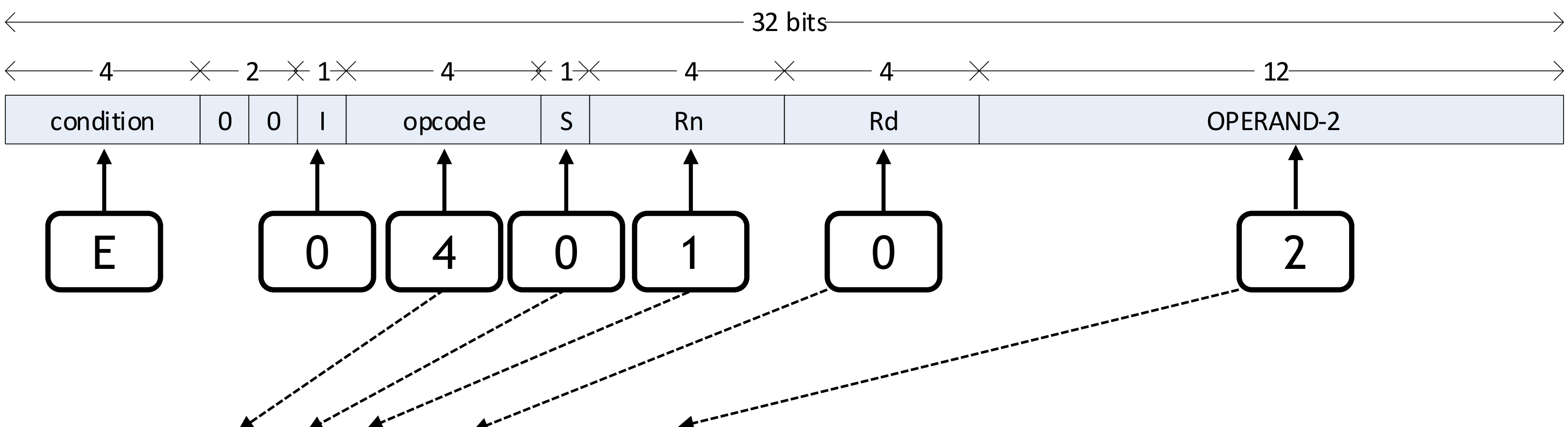
32 bits

4	2		1	4	1	4	4	12
condition	0	0	I	opcode	S	Rn	Rd	OPERAND-2

- 4 bit condition field (instruction can be conditionally executed, if value = 0xE instruction always executed)
- single I bit which determines how the OPERAND-2 field is interpreted
- 4 bit opcode field specifying the operation (16 possible operations)
- single S bit which determines if the instruction updates the condition codes
- 4 bit Rn field specifying src1 register (R0 .. R15)
- 4 bit Rd field specifying dst register (R0 .. R15)
- 12 bit src2 field (if I bit = 0 interpreted as a register or if I = 1 as an immediate value)
- fields will be described in more detail later in module

Assembly Language => Machine Code

- what is the machine code for ADD R0, R1, R2



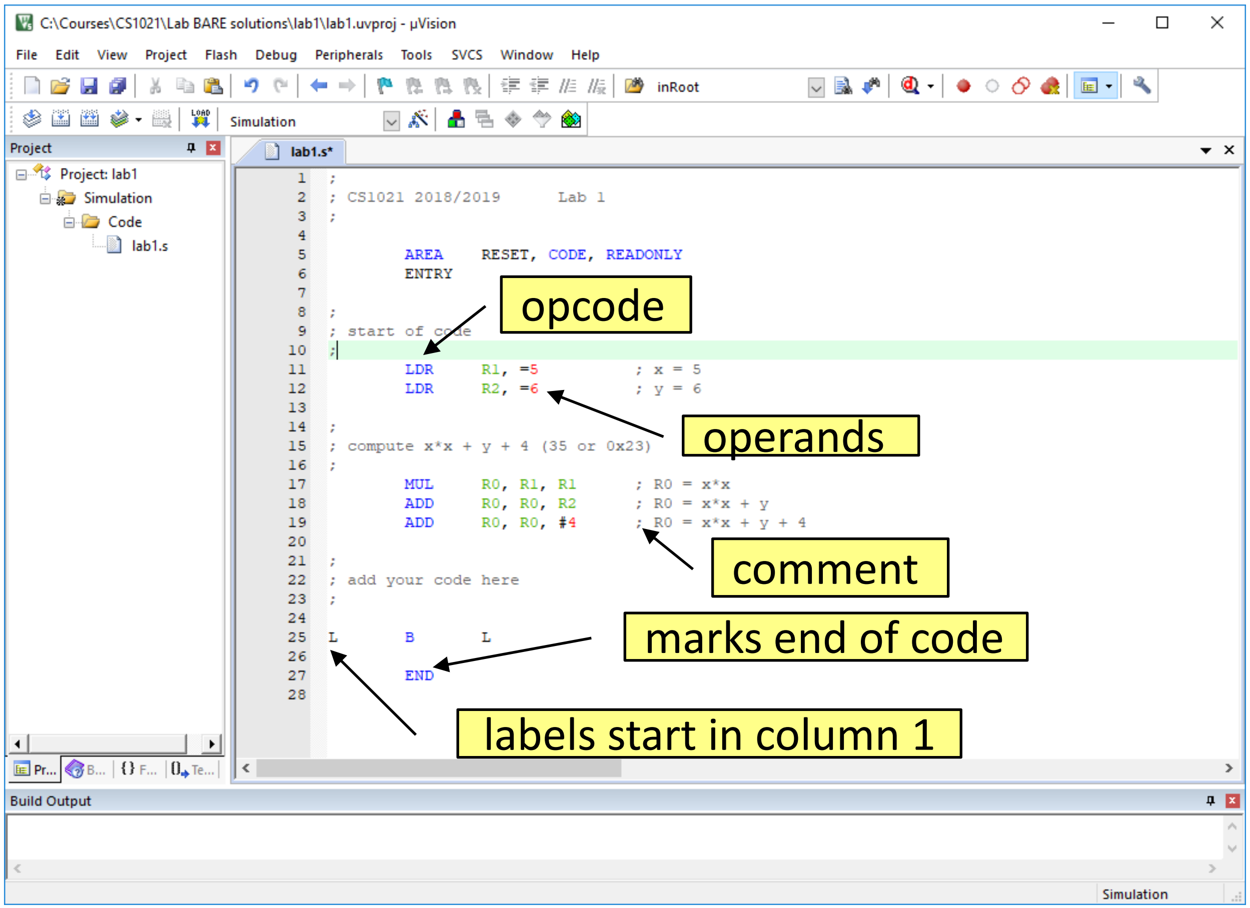
- $1110\ 0000\ 1000\ 0001\ 0000\ 0000\ 0000\ 0010_2$
- 0xE0810002

- don't have to remember the machine code, but looking at how instructions are encoded can help with figuring out what the instruction can do

Writing Assembly Language Programs

- writing programs using machine code is possible, but NOT practical
- much easier to write programs using assembly language
  - instructions are written using mnemonics (ADD instead 0x04, R2 instead of 0x2, …)
- assembly language translated into machine code by the assembler, stored in memory and then executed by CPU
- ARM assembly opcodes and operands NOT case sensitive
- one assembly language instruction per line
- labels start in column 1, otherwise leave blank except if a comment
- opcode followed by operands (separated by commas)
- comments start with a semicolon

Writing Assembly Language Programs

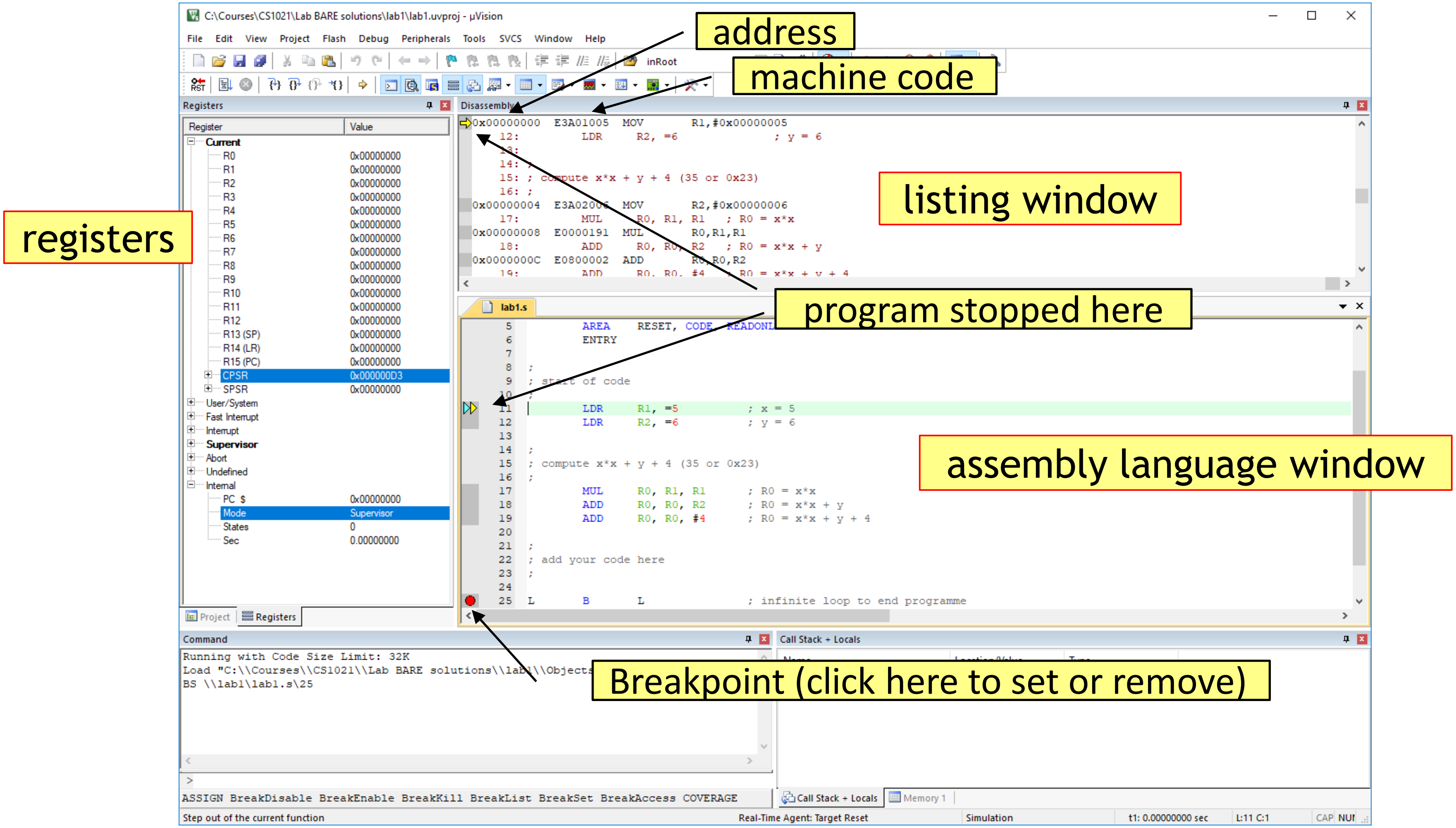


Executing/Debugging Assembly Language Programs

- build target [Project][Build Target]
- correct any assembly language errors and **REBUILD**

- run program [Debug][Start/Stop Debug Session]
- programs stops before first instruction executed

Executing/Debugging Assembly Language Programs



Executing/Debugging Assembly Language Programs

- press F11 or [Debug][Step] to single step one instruction at a time
- check instruction execution by examining register contents (remember register contents in hexadecimal)

- set breakpoints (red circle) by clicking on assembly language instruction (left hand side of assembly language window)
- press F5 or [Debug][Run] to run to next breakpoint (or forever if no breakpoint hit)
- check instruction execution by examining register contents

- break program in to sections and get each section working before moving on to next section

- [Debug][Start/Stop Debug Session] to exit debug session

Some Assembly Language Programming Guidelines

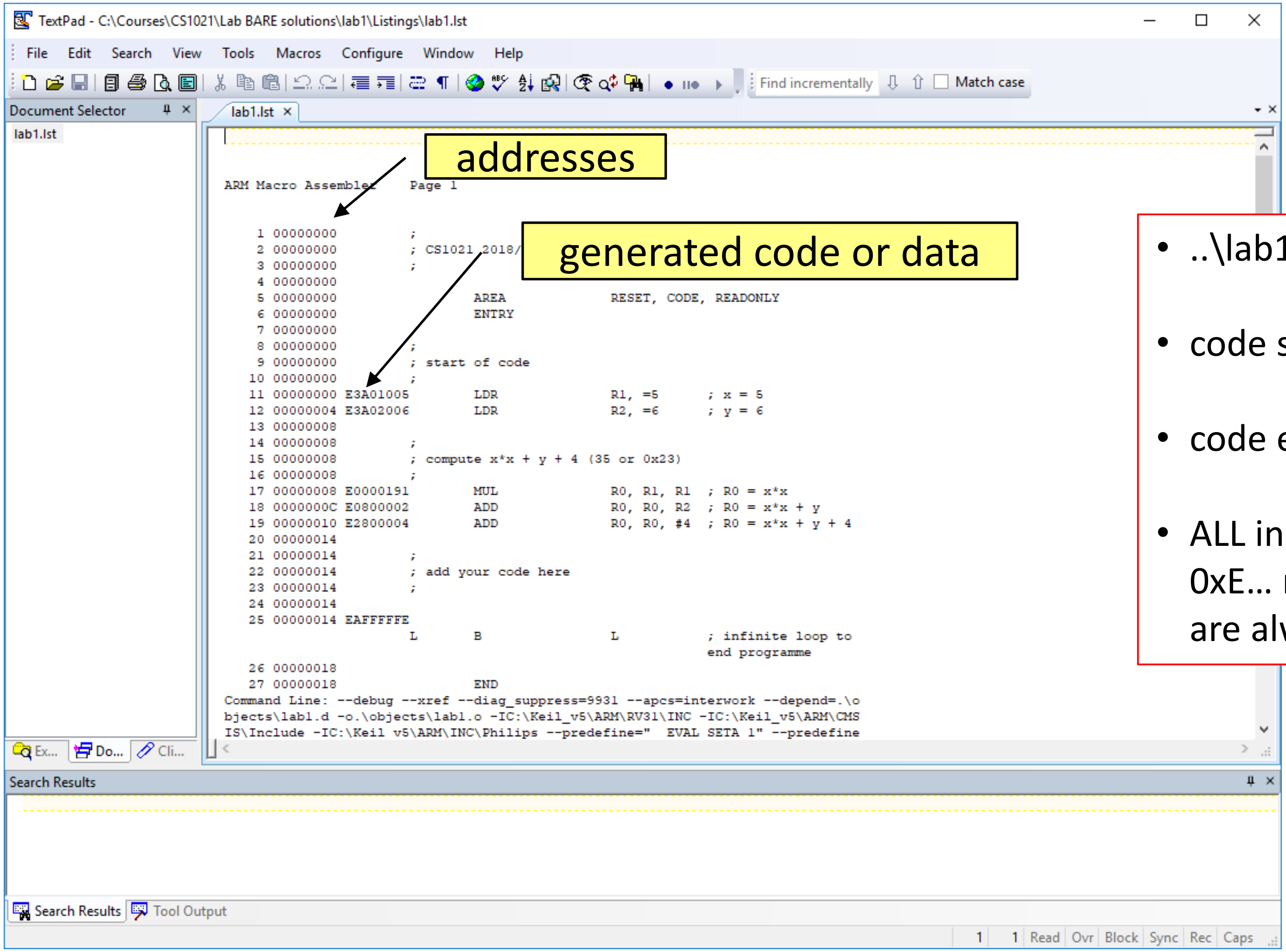
- comment every line of code with a helpful comment
- assume someone else may be reading your code

```
ADD  R0, R0, R2        ; R0 = R0 + R2              - poor
ADD  R0, R0, R2        ; R0 = x*x + 10x + 3        - better
```

- break your programs into small sections, separated by blank lines or comments
- try to keep your programs simple and easy to follow
- use TABs to align operator, operands and comments into columns
- tidy code = tidy mind

- remember to initialise values in registers and memory
- don't assume everything is set to zero when you start or switch on

Assembly Language Listing




```
ARM Macro Assembler    Page 1


    1 00000000          ;
    2 00000000          ; CS1021 2018/
    3 00000000          ;
    4 00000000
    5 00000000                  AREA            RESET, CODE, READONLY
    6 00000000                  ENTRY
    7 00000000
    8 00000000          ;
    9 00000000          ; start of code
   10 00000000          ;
   11 00000000 E3A01005         LDR             R1, =5      ; x = 5
   12 00000004 E3A02006         LDR             R2, =6      ; y = 6
   13 00000008
   14 00000008          ;
   15 00000008          ; compute x*x + y + 4 (35 or 0x23)
   16 00000008          ;
   17 00000008 E0000191         MUL             R0, R1, R1  ; R0 = x*x
   18 0000000C E0800002         ADD             R0, R0, R2  ; R0 = x*x + y
   19 00000010 E2800004         ADD             R0, R0, #4  ; R0 = x*x + y + 4
   20 00000014
   21 00000014          ;
   22 00000014          ; add your code here
   23 00000014          ;
   24 00000014
   25 00000014 EAFFFFFE
                        L       B               L           ; infinite loop to
                                                            end programme
   26 00000018
   27 00000018                  END
Command Line: --debug --xref --diag_suppress=9931 --apcs=interwork --depend=.\o
bjects\lab1.d -o.\objects\lab1.o -IC:\Keil_v5\ARM\RV31\INC -IC:\Keil_v5\ARM\CMS
IS\Include -IC:\Keil v5\ARM\INC\Philips --predefine="  EVAL SETA 1" --predefine
```

- ..\lab1\Listings\lab1.lst
- code starts at 0x00000000
- code ends at 0x00000017
- ALL instructions start with 0xE… meaning that they are always executed

This week's Tutorial and Lab

- will put this week's lecture notes on CS1021 web site today
- look at the this week's notes before Thursday's tutorial
- lab1 this Friday - you'll write your first assembly language program
- will put the lab question on the CS1021 web site on Thursday, so take some time to look at it in advance so you know what you have to do