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

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*)
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
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 OR 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).

\[
\begin{align*}
\text{ADD} & \quad R0, R1, \#1 \quad ; \quad R0 = R1 + 1 \\
\text{ADD} & \quad R2, R3, \#0x0F \quad ; \quad R2 = R3 + 0x0F \\
\text{SUB} & \quad R1, R1, \#2 \quad ; \quad R1 = R1 - 2 \\
\text{MOV} & \quad R0, \#3 \quad ; \quad R0 = 3 \\
\end{align*}
\]

- 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 $x^2 + 10x - 3$
- 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 = $x^2$
MOV R2, #10 ; R2 = 10
MUL R2, R1, R2 ; R2 = 10x (MUL R2, R2, R1 would not work)
ADD R0, R0, R2 ; R0 = $x^2 + 10x$
SUB R0, R0, #3 ; R0 = $x^2 + 10x - 3$
```

- 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

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

\[
\begin{array}{cccccccc}
\text{condition} & 0 & 0 & 1 & \text{opcode} & S & \text{Rn} & \text{Rd} & \text{OPERAND-2} \\
\hline
E & 0 & 4 & 0 & 1 & 0 & 0 & 2 \\
\end{array}
\]

- 1110 0000 1000 0001 0000 0000 0010
- 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

- Labels start in column 1
- Marks end of code

**Opcode**

**Operands**

**Comment**

**Marks end of code**

**Labels start in column 1**
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

address
machine code
listing window
program stopped here
assembly language window
Breakpoint (click here to set or remove)
registers
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 into 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

- ..\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