COMPUTER ORGANIZATION AND ARCHITECTURE Themes and Variations

ARM Processor WORKBOOK

Alan Clements

INTRODUCTION

This workbook has been written to accompany *Computer Organization and Architecture: Themes and Variations* and is designed to give students a practical introduction to the ARM processor simulator from Kiel. I have provided examples of the use of the ARM family simulator plus notes and comments in order to allow students to work together in labs and tutorials, or for individual study at home.

Before we introduce the simulator, we look at several background topics that are needed before you can begin to write assembly-language level programs.

THE INSTRUCTION SET ARCHITECTURE

An *instruction set architecture*, or ISA, is an abstract model of a computer that describes *what* it does, rather than *how* it does it. You could say that a computer's instruction set architecture is its *functional definition*. Essentially, the ISA is concerned with a computer's internal storage (its registers), the operations that the computer can perform on data (the instruction set), and the addressing modes used to access data. The term *addressing mode* is just a fancy way of expressing where the data is; for example, you can say that the data is in location 100, or you can say that it's 200 location from here, or you can say, "here's the actual data itself".

The first part of *Computer Organization and Architecture: Themes and Variations* is concerned with the instruction set architecture, and the second part is concerned with *computer organization* which described an ISA is actually implemented. Today, the term *microarchitecture* has largely replaced the *computer organization*. In this workbook, we are interested in the ISA, rather than the microarchitecture.

REGISTERS

A *register* is a storage device that holds a single data word exactly like a memory location. Registers are physically located on the CPU chip and can be accessed far more rapidly than memory. You can think of a register as a place in which data is waiting to be processed. When computers operate on data, they frequently operate on data that is in a register. For example, to perform the multiplication $A = B \times C$, you first read the values of *B* and *C* from memory into two registers. Then, you multiply the two numbers in the registers and put the result in a register. Finally, the result is transferred from a register to location *A* in memory.

In principle, there's no fundamental difference between a location in memory and a register. There are just a few registers in a computer, but millions of storage locations in memory. Consequently, you need far fewer bits to specify a register than a memory location. For example, if a computer has eight data registers, an instruction requires only three bits to select one of the eight registers to be used by an operation; that is from 000 to 111. If you specify a memory location, you need 32 bits to select one out of 2^{32} possible locations (assuming a 32-bit address space).

The size of a register (its width in bits) is normally the same size as memory locations and the size of the arithmetic and logical operations in the CPU. If you have a computer with 32-bit words, they are held in 32-bit memory locations and 32-bit registers and are processed by 32-bit adders, and so on.

There is no fundamental difference between a register and a memory location. If you could store gigabytes of high-speed memory on a CPU chip and you could use very long instruction words (i.e., with the long addresses needed to specify one individual location) then there would be no point in using registers. If you had a computer with 4 Gbytes of memory (2^{32} bytes) and wished to have an instruction that could implement C = A + B (i.e., ADD **C**, A, B) the you would require typically 16 + 32 + 32 = 112 bits (the 16 bits represent the number of bits to encode the actual operation and the three 32-bits are needed for the addresses A, B, and C). No mainstream modern computer has such a long instruction word.

PROBLEM SET 1

- 1. In your own words, explain what a register is in a computer.
- 2. How many registers does the 68K have?
- 3. How many registers does the ARM have?
- 4. What's the processor with the largest number of registers that you can find?
- 5. If a computer has 128 user-accessible general-purpose registers, how many bits are be required to access a register? That is, how many bits does it take to specify 1 out of 128?
- 6. Suppose a computer has eight registers and a 24-bit instruction length. A data processing instruction is of the ADD r1, r2, r3 which implements r1 = r2 + r3. How many bits in an instruction can be allocated to specifying an operation if there are four general-purpose registers?

IMPORTANT POINT

Never confuse the following two concepts: **value** and **address** (or location). A *memory location* holds a *value* which is the information stored in that location. The address of an item is *where* it is in memory and its value is *what* it is.

For example, suppose memory location 1234 contains the value 55. If we add 1 to 55 we get 55 + 1 which is 56. That is, we've changed the value of a variable. Now, if we add 1 to the address 1234, we get 1235. That's a different location in memory which holds a different variable.

The reason for making this point is that it is all too easy to confuse these two concepts because of the way we learn algebra at high school. We use equations like x = 4. When we write programs that use variables, the variables usually refer to the locations of data not to the values. So, when we say x = 4, we actually mean that the memory location called *x* contains the value 4.

PROBLEM SET 2

The following problems are intended to help you understand the history of the computer. These problems are intended as discussion points and don't have simple right or wrong answers. In order to do these questions you will need to read the Web-based history material that accompanies this text. You will also need to use the web as a research tool.

- 1. When did the idea of a computer first occur to people?
- 2. What is a computer?
- 3. One of the names most associated with the history of computing is *John von Neumann*. Who was von Neumann? Did he invent the computer?
- 4. When was the first microprocessor created and by whom?
- 5. What was the form of the first memory used by computers (or computing devices)?
- 6. Who said (and when) "There is a world market for maybe five computers".
- 7. What was the first hobby computer (personal computer) and when was it built?
- 8. Who was Konrad Zuse?

This warning symbol will appear whenever a particularly important or tricky concept is introduced.



 $2 \mid P a g e$

Addressing Modes

An addressing mode is simply a means of *expressing the location of an operand*. An address can be a register such as r3, or D7, or PC (program counter). An address can be a location in memory such as address 0x12345678. You can even express an address *indirectly* by saying, for example, "the address is the location whose address is in register r1". All the various ways of expressing the location of data are called collectively *addressing modes*.

Suppose someone said, "Here's ten dollars". They are giving you the actual item. This is called a **literal** or **immediate** value because it's what you actually get. Unlike all other addressing modes, you don't have to retrieve immediate data from a register or memory location.

If someone says, "Go to room 30 and you'll find the money on the table", they are telling you *where* the money is (i.e., its address is room 30). This is called an **absolute address** because expresses absolutely exactly where the money is. This addressing modes is also called direct addressing.

Now here's where the fun starts. Suppose someone says, "Go to room 40 and you'll find something to your advantage on the table". You arrive at room 40 and see a message on the table saying, "The money is in room 60". In this case we have an **indirect address** because room 40 doesn't give us with the money, but a pointer to where it is. We have to go to a second room to get the money. Indirect addressing is also called **pointer-based** addressing, because you can think of the note in room 40 as pointing to the actual data.

In real life we can't confuse a room or address in with a sum of money. However, in a computer all data is stored in binary form and the programmer has to remember whether a variable (or constant) is an address or a data value.

By the way, because there is no means of telling which operand is a *source* and which is a *destination* in a computer instruction such as MOVE A, B and different computers use different conventions, I have decided to write the destination operand in bold font to make it easier to understand the code. For example, MOVE **A**, B means that *B* is moved to *A*, because *A* is bold and therefore the destination of the result.

Let's look at three computer instructions in 68K assembly language. The operation MOVE D0, **D1** means copy the contents of register D0 into D1. The operation MOVE (A0), **D1** means copy the contents of the memory location *pointed at by register A0* into register D1. This is an example of indirect addressing because the instruction specifies register A0 as the source operand and then this value has to be read in order to access the desired operand in memory.



Here we've used 68K instructions (the 68K instruction set is given as an appendix on page 8). In ARM assembly language, which is the subject of this Workbook, indirect addressing is indicated by square brackets. For example, LDR **r0**, [r1] indicates that the contents of the memory location pointed at by register r1 is to be read and copied into register r0. Note that the ARM and 68K assembly languages specify the order of operands differently. In the assembly language we use in this course:

Immediate (literal) addressing is indicated by a '#' symbol in front of the operand (this convention is used by both the ARM and 68K). Thus, #5 in an instruction means the actual value 5. A typical ARM instruction is MOV = 0, #5 which means move the value 5 into register r0.

Absolute (direct) addressing is not implemented by the ARM processor. It is provided by the 68K and Intel IA32 processors; for example, the 68K instruction MOVE 1234, **D0** means load register D0 with the contents of memory location 1234. The ARM supports only register indirect addressing.

Indirect addressing is indicated by ARM processors by placing the pointer in square parentheses; for example, [r1]. All ARM indirect addresses are of the basic form LDR r0, [r1] or STR r3, [r6]. There are variations on this addressing mode; for example, LDR r0, [r1, #4] specifies an address that is four bytes on from the location pointed at by the contents of register r1.

Addressing Modes Example

Let's clarify addressing modes with a simple example. The memory map below gives the contents of each of the locations of a simple 16-word memory. Each of these locations contains a 4-bit binary value. We are going to look at some examples of the effect of computer operations. We adopt ARM-style assembly instructions and assume 4-bit addresses and 4-bit data.

0000	0010
0001	0011
0010	0010
0011	1010
0100	0000
0101	0010
0101	0001
0111	0011
1000	1010
1001	1111
1010	1010
1011	0011
1100	0001
1101	1000
1110	0000
1111	1010

Assume that r1 initially contains 0001 and r2 contains 1000

a.	MOV r0, #1100	Literal address	Register r0 is loaded with 1100
b.	LDR r0, [r1]	Register indirect address	Register r0 is loaded with 0011
c.	LDR r0, [r2]	Register indirect address	Register r0 is loaded with 1010
d.	LDR r0 , [r1, r2]	Register indirect address (sum of r1 and r2)	Register r0 is loaded with 1111
e.	LDR r0, [r2,#4]	Register indirect address $(r2 + 4)$	Register r0 is loaded with 0001
f.	LDR r0 , [r2, #-4]	Register indirect address $(r2 - 4)$	Register r0 is loaded with 0000

As you can see, the processor uses the address in r1 or r2 to access the appropriate memory location. ARM processors (like other processors) are able to perform limited pointer arithmetic. For example, in (d) the effective address is given as [r1,r2], which is the location pointed at by the sum of these two registers. The sum of r1 and r2 is 0001 + 1000 = 1001, so the contents of location 1001 (i.e., 1111) are loaded into r0.

Example (e) calculates an effective address by adding 4 to the contents of r2 to get 1000 + 0100 = 1100. The contents of memory location 1100 is 0001 and that value is loaded into r0. Note that example (f) is almost the same except that the constant is negative. In this case the contents of location 1000 - 0100 = 0100 (i.e., 0000) are loaded into r0. A negative offset like this accesses a location at a lower address.

EXAMPLE

A special-purpose computer has an instruction with a word-length of 24 bits. It is intended to perform operation of the type ADD r3, #24 where ADD is an operation, #24 is a literal (an actual number), and r3 is a destination register.

If there are 200 different instructions and 32 registers, what is the range of unsigned integer literals that can be supported by this computer?

SOLUTION

We know that the number of bits used to represent the instruction, plus the number of bits used to select a register, plus the number of bits used to specify a literal must be 24. There are 200 instructions. The next power of 2 greater than this is 256. Since $2^8 = 256$, we need 8 bits for the instruction. There are 32 registers and it requires 5 bits (as $2^5 = 32$) to address a register. Having allocated 8 bits to the instruction field and 5 bits to the register field, we have 24 - 8 - 5 = 11 bits left over to specify a literal (constant). Consequently, the range of literals that can be handled is 0 to 2047 (as $2^{11} = 2048$).

REGISTER TRANSFER LANGUAGE

Before we introduce computer instructions, we are going to define a notation that makes it possible to define instructions clearly and unambiguously (English language is not a good tool for defining instructions).

Register-transfer language (RTL) is an algebraic notation that describes how information is accessed from memories and registers and how it is operated on. You should appreciate that RTL is just a *notation* and not a programming language. RTL uses square brackets to indicate the *contents* of a memory location; for example, the expression

[6] = 3

is interpreted as *the contents of memory location 6 contains the value 9*. If we were using symbolic names, we might write [Time] = HoursWorked.

If you want to refer to a register, you simply use its *name* (the names of registers vary from computer to computer – the 68K has eight data registers called D0, D1, D2, ..., D7, whereas the ARM has 16 registers called r0 to r15). So, to say that register D6 contains the number 123 we write

[D6] = 123

A left or *backward* arrow \leftarrow indicates the transfer of data. The left-hand side of an expression denotes the *destination* of the data defined by the *source* of the data defined on the right-hand side of the expression. For example, the expression

 $[MAR] \leftarrow [PC]$

indicates that the contents of the *program counter*, PC, are copied into the *memory address register*, MAR. The program counter is the register that holds the location of the next instruction to be executed. The MAR is a register that holds the address of the next item to be read from memory or written to memory. Note that the contents of the PC are not modified by this operation.

The operation [3] \leftarrow [5] means copy the contents of memory location 5 to location 3.

The operation $[3] \leftarrow [5]$ tells us what's happening at the *micro* level or register-transfer level. In a high-level language this operation might be written in the rather more familiar form

x = y;

Consider the RTL expression

[PC] ← [PC] + 4

which indicates that the number in the PC is increased by 4; that is, the contents of the program counter are read, 4 is added, and the result is copied into the PC.

Suppose the computer executes an operation that stores the contents of the program counter in location 2000 in the memory. We can represent this action in RTL as

[2000] ← [PC].

Occasionally, we wish to refer to the individual bits of a register or memory location. We will do this by means of the subscript notation (p:q) to mean bits *p* to *q* inclusive; for example if we wish to indicate that bits 0 to 7 of a 32-bit register are set to zero, we write

 $[R6_{(0:7)}] \leftarrow 0.$

Numbers are assumed to be decimal, unless indicated otherwise. Computer languages adopt conventions such as 0x12AC or \$12AC to indicate hexadecimal values. In RTL we will use a subscript; that is $12AC_{16}$.

As a final example of RTL notation, consider the following RTL expressions.

a.	[20]	= 6	
b.	[20]	← 6	
c.	[20]	← [6]	
d.	[20]	← [6] + 3	١.
e.	[20]	← [[2]] ←	_

The symbol " \leftarrow " is equivalent to the assignment symbol in high-level languages. Remember that RTL is not a computer language; it is a *notation* used to define computer operations.

Example (a) states that memory location 20 contains the value 6. Example (b) states that the number 6 is *copied* or *loaded* into memory location 20. Example (c) indicates that the contents of memory location 6 are copied into memory location 20. Example (d) reads the contents of location 6, adds 3 to it, and stores the result in location 20. Example (e) is most interesting. Here, the contents of memory location 2 is read, and that value used to access memory a second time. The new value is loaded into the contents of memory location 20. This is an example of memory indirect addressing.

Consider the following examples that illustrate the assembly language of four processors and define each instruction in RTL.

Processor family	Instruction mnemonic	RTL definition
1. 68K	MOVE D0, (A5)	[[A5]] ← [D0]
2. ARM	ADD r1 , r2, r3	[r1] ← [r2] + [r3]
3. IA32	MOV ah ,6	[ah] ← 6
4. PowerPC	li r25, 10	[r25] ← 10

RTL AND ASSEMBLY LANGUAGE

Don't confuse RTL and assembly language. An assembly language is a human-readable form of a computer's binary code. It is designed to be used by programmers and may not always be logical or consistent. Some of you may notice inconsistencies in the assembly language that we learn in this course.

RTL is a formal notation that can be manipulated like any algebraic expression. It offers a means of precisely defining operations without using ambiguous English. Consider the RTL example:

Suppose that [4] = 3, [10] = 4, and [[10]] = y.

We can say that y = 3, because we can substitute y = [[10]] = [4] = 3

Similarly, [[4] + [10] + 6] = [3 + 4 + 6] = [13]

QUICK OVERVIEW OF THE ARM

Before looking at the ARM processor in detail, we provide a very brief overview. The ARM processor is classified as a 32-bit RISC (reduced instruction set processor) with a three-operand register-to-register instruction set. This is just a fancy way of saying that computer operations involve three operands in registers such as ADD r1, r2, r3. There are a few instructions that have two operands and some that have four, but that doesn't change the overall classification.

In order to get data into and out of registers (transfers between memory and registers), there are two special instructions called load and store. Load transfers data from memory to a register and store transfers data from a register to memory. These instructions have the forms LDR **r0**, [r1] and STR r0, [**r1**]. As we have seen, these instructions use register indirect (i.e., pointer-based) addressing. The location of the memory element to be accessed is held in a register and the addressing mode indicated by [r1].

The ARM uses a special instruction called ADR (load register with an address) that sets up a pointer in the first place). For example

ADR r0, List ; register r0 points at the list

Later, we will explain why this is a special instruction.

An ARM instruction like SUB r3, r2, #4 subtracts the actual value 4 (remember that the literal is indicated by the # symbol) from the contents of register r2 and puts the result in r3. Data operations implemented by ARM processors write the destination (result) operand first on the left. We write the destination operand in bold font to remind you where the result goes.

Let's create a very simple example.

MOV	r0, #2	; Put 2 in register r0
MOV	r1, #3	; Put 3 in register r1
ADD	r2 , r0, r1	; Add r0 to r1 and put the result in r2
MOV	r4, #10	; Put 10 in r4 (this is where we are going to store the result)
STR	r2,[r4]	; Store r2 in memory location 10

Note how simple all this is. You perform one primitive operation at a time.

QUICK OVERVIEW OF THE 68K

Although this text uses the ARM processor family to illustrate an instruction set architecture, we do occasionally refer to the Motorola 68K family. In brief, the Motorola 68K is a 32-bit processor first sold in 1980. The 68K family later became the ColdFire family and is now supported by Freescale because Motorola dropped out of the microprocessor market. The 68K is contemporary with Intel's IA32. Both the 68K and IA32 have classic register-to-memory architecture.

The 68K has a moderately regular instruction set in comparison with the IA32 architecture. Here, the term *regular* implies that if instruction *X* has addressing mode *Y*, then instruction *P* will also have addressing mode *Y*. The 68K's main features are:

- A 32-bit architecture with 32-bit registers.
- Separate data registers (D0 to D7) and address registers (A0 to A7). Address registers may only be used as pointer registers in generating effective addresses. A register indirect is indicated by (A0).
- All registers are 32 bits wide. However, many operations can act on the lower-order 8 bits of a data register, on the lower-order 16 bits, or on the entire 32 bits. The data size is indicated by appending .B, .W, or .L to specify an 8-bit, 16-bit, or 32-bit operation. For example MOVE .B D0, (A0).
- Data registers can take part in all data operations. Address registers can take part only in move, add, subtract, and compare operations (that is, MOVA, ADDA, SUBA, CMPA).
- Operations on data registers update the CCR register, whereas operations on address registers (apart from compare) do not affect the CCR.
- All operations on an address register yield a 32-bit result. You can perform 16-bits additions, subtractions, and loads on an address register, but the result is always sign-extended to 32 bits.
- 68K instructions are variable length. The shortest instruction is 16-bits. If a single operand is required, the length may be 16+16 or 16+32 bits. The longest instruction is 10 bytes for a move memory location to memory location such as MOVE Data1, Data2.
- The addressing modes are: literal (8-, 16-, or 32-bit constant), absolute (actual address of the operand in memory), address register based {(A0), (#offset,A0), (D0,A0)}, predecrementing -(A0), postincrementing (A0)+}
- Address register A7 is the system stack pointer and is used to store the return address after a subroutine call. The instruction RTS implements a subroutine return by popping the return address off the top of the stack and loading it in the PC.
- Program counter relative addressing is supported. For example, MOVE (PC, #offset), D0.
- The creation and deletion of stack fames is supported by LINK (create a frame) and UNLK (delete a frame).

A typical fragment of 68K code is:

MOVEA #X,A0 ;A0 points at X MOVEA #Y,A1 ;A1 points at Y MOVE #32,D1 ;32 times round the loop Loop MOVE (A0)+,D2 ;get Xi and increment pointer MOVE (A1)+,D3 ;get Yi and increment pointer MULU D2,D3 ;multiply Xi and Yi ADD D3,D0 ;update running total SUB #1,D2 ;decrement loop counter BNE Loop ;Repeat until all done		CLR	D0	;clear the total in DO
MOVEA #Y,A1 ;A1 points at Y MOVE #32,D1 ;32 times round the loop Loop MOVE (A0)+,D2 ;get Xi and increment pointer MOVE (A1)+,D3 ;get Yi and increment pointer MULU D2,D3 ;multiply Xi and Yi ADD D3,D0 ;update running total SUB #1,D2 ;decrement loop counter BNE Loop ;Repeat until all done		MOVEA	#X, AO	;A0 points at X
MOVE#32,D1;32 times round the loopLoopMOVE(A0)+,D2;get Xi and increment pointerMOVE(A1)+,D3;get Yi and increment pointerMULUD2,D3;multiply Xi and YiADDD3,D0;update running totalSUB#1,D2;decrement loop counterBNELoop;Repeat until all done		MOVEA	#Y ,A1	;Al points at Y
Loop MOVE (A0)+,D2 ;get Xi and increment pointer MOVE (A1)+,D3 ;get Yi and increment pointer MULU D2,D3 ;multiply Xi and Yi ADD D3,D0 ;update running total SUB #1,D2 ;decrement loop counter BNE Loop ;Repeat until all done		MOVE	#32 ,D1	;32 times round the loop
MOVE(A1)+,D3;get Yi and increment pointerMULUD2,D3;multiply Xi and YiADDD3,D0;update running totalSUB#1,D2;decrement loop counterBNELoop;Repeat until all done	Loop	MOVE	(A0)+, D2	;get Xi and increment pointer
MULUD2,D3;multiply Xi and YiADDD3,D0;update running totalSUB#1,D2;decrement loop counterBNELoop;Repeat until all done		MOVE	(A1)+,D3	;get Yi and increment pointer
ADDD3,D0;update running totalSUB#1,D2;decrement loop counterBNELoop;Repeat until all done		MULU	D2, D3	;multiply Xi and Yi
SUB#1,D2;decrement loop counterBNELoop;Repeat until all done		ADD	D3 ,D0	;update running total
BNE Loop ;Repeat until all done		SUB	#1, D2	;decrement loop counter
		BNE	Loop	;Repeat until all done

As you can see, this is not too far from ARM code. The significant difference is the two-operand instruction format.

68K INSTRUCTION SET

Here's a summary of the 68K operations. We give the mnemonic, name of the operation, addressing modes, and operand sizes supported (Bytes, Word, Longword).

HDCD	Add BCD with extend	Dx, Dy,	-(AX), -(AY)	В
ADD	ADD	Dn, <ea>,</ea>	<ea>,Dn</ea>	BWL
ADDA	ADD binary to An	<ea>,An</ea>		WL
ADDI	ADD Immediate	#x <ea></ea>	#<1-8> <ea></ea>	BWL.
ADDO	ADD 2 hit immediate	na, seur,	11 ×1 07, ×Cur	DWL
ADDQ	ADD 5-bit ininieulate			BVVL
ADDX	ADD extended	Dy,Dx,	-(Ay),-(Ax)	BWL
AND	Bit-wise AND	<ea>,Dn,</ea>	Dn, <ea></ea>	BWL
ANDI	Bit-wise AND with Immediate	# <data>.<ea></ea></data>		BWL
ASI	Arithmetic Shift Left	#<1-8> Dv	Dy Dy Coas	BWI
ACD	Arithmetic Shift Dight	π<1-0>,Dy,	Dx,Dy, <ea></ea>	DWL
ASK	Arithmetic Shirt Right			BWL
Bcc	Conditional Branch	Bcc <label></label>		BW
BCHG	Test a Bit and CHanGe	Dn, <ea></ea>	# <data>,<ea></ea></data>	BL
BCLR	Test a Bit and CLeaR			BL
BSET	Test a Bit and SET			BI
DCD	Dura al ta Cul Dautina	DCD alabala		DL
BSK	Branch to SubRoutine	BSK <lapel></lapel>		BW
BTST	Bit TeST	Dn, <ea></ea>	# <data>,<ea></ea></data>	BL
CHK	CHecK Dn Against Bounds	<ea>.Dn</ea>		W
CLR	CLeaR	< 63>		BWL
CMD	CoMDavo	scar Dr		DWL
CMP	COMPARE	<ea>,Dii</ea>		BVVL
CMPA	CoMPare Address	<ea>,An</ea>		WL
CMPI	CoMPare Immediate	# <data>,<ea></ea></data>		BWL
CMPM	CoMPare Memory	(Av)+(Ax)+		BWL
DBcc	Looning Instruction	DBcc Dn clabe	ls.	W
DUVC	DWide Cime d	DDCC DII, siube	15	147
DIVS	Divide Siglied	<ea>,Dii</ea>		vv
DIVU	DIVide Unsigned	<ea>,Dn</ea>		W
EOR	Exclusive OR	Dn, <ea></ea>		BWL
EORI	Exclusive OR Immediate	# <data>.<ea></ea></data>		BWL
FYC	Exchange any two registers	Ry Ry		Т
ENT	Cian EVT and	D., Ny		L
EAI	Sign EXTend	Dn		VV L
ILLEGAL	ILLEGAL-Instruction Exception			
JMP	JuMP to Affective Address	<ea></ea>		
ISR	Jump to SubRoutine	<ea></ea>		
LEA	Load Effective Address	ceas An		Т
	Alla anta Cha ala Francia	Au # adianlara		Ľ
LINK	Allocate Stack Frame	All,# <uispiace< td=""><td>inent></td><td></td></uispiace<>	inent>	
LSL	Logical Shift Left	Dx,Dy #<1-8>	,Dy <ea></ea>	BWL
LSR	Logical Shift Right			BWL
MOVE	Between Effective Addresses	<ea>.<ea></ea></ea>		BWL
MOVE	To CCR	<ea> CCR</ea>		W
		, s s s s s s s s s s s s s s s s s s s		
MOVE	To SR	<025 SR		W/
MOVE	To SR	<ea>,SR</ea>		W
MOVE MOVE	From SR	<ea>,SR SR,<ea></ea></ea>		W
MOVE MOVE MOVE	To SR From SR USP to/from Address Register	<ea>,SR SR,<ea> USP,An, An,US</ea></ea>	P <ea>,An</ea>	W W L
MOVE MOVE MOVE MOVEA	To SR From SR USP to/from Address Register MOVE Address	<ea>,SR SR,<ea> USP,An, An,US</ea></ea>	P <ea>,An</ea>	W W L WL
MOVE MOVE MOVE MOVEA MOVEM	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple	<ea>,SR SR,<ea> USP,An, An,US <register list="">,</register></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<="" td=""><td>W W L WL WL</td></register></ea></ea></ea>	W W L WL WL
MOVE MOVE MOVE MOVEA MOVEM MOVEP	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral	<ea>,SR SR,<ea> USP,An, An,US <register list="">, Dn.x(An).</register></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An).Dn</register></ea></ea></ea>	W W L WL WL
MOVE MOVE MOVE MOVEA MOVEM MOVEP MOVEO	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE Pehit immediate	<ea>,5R SR,<ea> USP,An, An,US <register list="">, Dn,x(An), #<-128 + 127</register></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn > Dn</register></ea></ea></ea>	W W WL WL L
MOVE MOVE MOVE MOVEA MOVEM MOVEP MOVEQ	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MUL kiek Simod	<ea>,5R SR,<ea> USP,An, An,US <register list="">, Dn,x(An) , #<-128,+127></register></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn</register></ea></ea></ea>	W L WL WL L W
MOVE MOVE MOVEA MOVEM MOVEP MOVEQ MULS	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed	<ea>,SR SR,<ea> USP,An, An,US <register list="">, Dn,x(An) , #<-128.+127> <ea>,Dn</ea></register></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,,Dn</register></ea></ea></ea>	W W WL WL L W
MOVE MOVE MOVEA MOVEA MOVEM MOVEP MOVEQ MULS MULU	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned	<ea>,5R SR,<ea> USP,An, An,US on,x(An), #<-128.+127: <ea>,Dn <ea>,Dn</ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn</register></ea></ea></ea>	W W L WL WL W W W
MOVE MOVE MOVEA MOVEA MOVEM MOVEQ MULS MULU NBCD	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD	<ea>,5R SR,<ea> USP,An, An,US <register list="">, Dn,x(An) , #<-128,+127> <ea>,Dn <ea>,Dn <ea></ea></ea></ea></register></ea></ea>	P <ea>,An ,<ea> ,<register list<br="">,(An),Dn >,Dn</register></ea></ea>	W W L WL WL L W W B
MOVE MOVE MOVEA MOVEM MOVEP MOVEQ MULS MULU NBCD NEG	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate	<ea>,5R SR,<ea> USP,An, An,US oregister list>, Dn,x(An), #<-128.+127: <ea>,Dn <ea>,Dn <ea> <ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,Dn</register></ea></ea></ea>	W W L WL WL U W W B B B WL
MOVE MOVE MOVEA MOVEM MOVEP MOVEQ MULS MULU NBCD NEG NEGX	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend	<ea>,SR SR,<ea> USP,An, An,US On,x(An), #<-128,+127- <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn</register></ea></ea></ea>	W W L WL WL L W W B B WL BWL
MOVE MOVE MOVEA MOVEM MOVEM MOVEQ MULS MULU NBCD NEG NEG NOP	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE 9-reipheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration	<ea>,SR SR,<ea> USP,An, An,US Dn,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn</register></ea></ea></ea>	W W L WL WL L W W B B BWL BWL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEGX NOP NOT	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127> <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,Dn</register></ea></ea></ea>	W W WL WL W W W B B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEM MOVEQ MULS MULU NBCD NEG NEG NOP NOT OD	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Dit with OD	<ea>,SR SR,<ea> USP,An, An,US Dn,x(An), #<-128.+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn</register></ea></ea></ea>	W W L WL WL L W W B BWL BWL BWL
MOVE MOVE MOVEA MOVEM MOVEQ MULS MULU NBCD NEG NEGX NOT OR	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit wise OR	<ea>,SR SR,<ea> USP,An, An,US Dn,X(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,On <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,Dn</register></ea></ea></ea>	W W L WL WL L W W W B B B WL BWL BWL BWL
MOVE MOVE MOVEA MOVEA MOVEM MOVEQ MULS MULU NBCD NEG NEG NOP NOT OR OR	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate	<ea>,SR SR,<ea> USP,An, An,US \n,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn</register></ea></ea></ea>	W W L WL WL L W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEM MOVEQ MULS MULU NBCD NEG NEGX NOP NOT OR ORI PEA	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate Push Effective Address	<ea>,SR SR,<ea> USP,An, An,US Dn,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea>,Dn #<data>,<ea></ea></data></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea></ea></register></ea></ea></ea>	W W L WL WL L W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEP MOVEQ MULS MULU NBCD NEG NEGS NOP NOT OR OR QR QRI PEA RESET	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE B-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127- <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn</register></ea></ea></ea>	W W WL WL W W W B B WL B WL B WL B WL L
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NOP NOT OR ORI PEA RESET ROL	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE 9-ripheral MOVE 9-bit immediate MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices ROtate Left	<ea>,SR SR,<ea> USP,An, An,US \n,x(An), #<-128,+127; <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx.Dy. <ea></ea></ea></register></ea></ea></ea>	W W L WL WL L W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEGS NOP NOT OR OR ORI PEA RESET ROL BOB	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices ROtate Left Potete Bight	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127> <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,Dn Dn,<ea> Dx,Dy, <ea></ea></ea></register></ea></ea></ea>	W W WL WL W W W B B WL B WL B WL B WL B
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEG NOP NOT OR OR ORI PEA RESET ROL ROR	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise Chatter and devices ROSET all external devices ROtate Left ROtate Right	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea></ea></ea></register></ea></ea></ea>	W W L WL WL L W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEM MOVEQ MULS MULU NBCD NEG NEGX NOP NOT OR ORI PEA RESET ROL ROR ROXL	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 9-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate et NEGate et NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices ROtate Left ROtate Right ROtate Left with eXtend	<ea>,SR SR,<ea> USP,An, An,US On,x(An), #<-128,+127> <ea>,Dn <ea>,Dn <ea>,Ca> <ea> <ea> <ea> <ea> <ea> <ea> <ea> <e< td=""><td>P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea></ea></ea></register></ea></ea></ea></td><td>W W WL WL W W W B B W B W B W L B W L B W L B W L B W L B W L B W L B W L</td></e<></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea></ea></ea></register></ea></ea></ea>	W W WL WL W W W B B W B W B W L B W L B W L B W L B W L B W L B W L B W L
MOVE MOVE MOVEA MOVEA MOVEM MOVEQ MULS MULU NBCD NEG NEG NEG NEG NOP NOT OR OR OR OR OR OR RESET ROL ROR ROXL ROXL	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 9-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise 0R Bit-wise 0R Bit-wise 0R Bit-wise 0R Bit-use 1 external devices ROSTA II external devices ROSTA II external devices ROTATE Left ROTATE Left with eXtend ROTATE Right with eXtend	<ea>,SR SR,<ea> USP,An, An,US On,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea></ea></ea></register></ea></ea></ea>	W W U WL WL L W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEM MOVEQ MULS MULU NBCD NEG NEG NEG NEG NOP NOT OR OR PEA RESET ROR ROR ROR ROXL ROXR RTE	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices ROtate Left ROtate Left ROtate Left with eXtend Rotate Right with eXtend Refurn from Exception	<ea>,SR SR<ea> USP,An, An,US Dn,x(An), #<-128.+127: <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea></ea></ea></register></ea></ea></ea>	W W U WL U U W W B B WL B WL B WL B WL B
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEGS NEGS NOP NOT OR OR OR OR RESET ROL ROR ROSK ROSK ROSK ROSK RTE RTE	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise 0R Bit-wise 0R Bit-wise 0R Bit-wise 0R Bit-wise 0R Bit-wise 0R Bit-use 0R Bit-use 10 external devices ROtate Left ROtate Left ROtate Right ROtate Right ROtate Right ROtate Right with eXtend Rotate Right with eXtend ReTurn from Exception ReTurn and Restore	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> =<a> =<a> =<a> =<a> =<a> =<a> =<a> =<</ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea></ea></ea></register></ea></ea></ea>	W WL WL WL W W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEG NOP NOT OR ORI PEA RESET ROL ROR ROR ROXL ROXL ROXR RTE RTR PTS	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Unsigned MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices ROtate Left ROtate Right ROtate Right ROtate Left with eXtend Rotate Right with eXtend ReTurn from Exception ReTurn and Restore	<ea>,SR SR,<ea> USP,An, An,US n,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy,<ea></ea></ea></register></ea></ea></ea>	W W U WL WL L W W W B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEP MOVEQ MULS MULU NBCD NEG NEGS NOP NOT OR OR OR OR OR OR OR RESET ROL ROS ROS ROS ROS ROS ROS ROS ROS ROS ROS	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE S-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Case Complement Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Case Complement Bit-wise OR Bit-wise OR	<ea>,SR SR,<ea> USP,An, An,US On,x(An), #<-128,+127- <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea></ea></ea></register></ea></ea></ea>	W WL WL WL W W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEG NEG NOP NOT OR OR OR PEA RESET ROL ROR ROXL ROXR RTE RTS SBCD	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 9-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Cate Eart ROtate Left ROtate Right ROtate Left ROtate Left ROtate Left ROtate Right with eXtend ROtate Right with eXtend ROtate Right with eXtend ReTurn from Exception ReTurn from Subroutine Subtract BCD with eXtend	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay)</ea></ea></register></ea></ea></ea>	W W WL WL L W W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEG NEG NOP NOT OR ORI PEA RESET ROR ROR ROR ROXL ROR ROXL ROXR RTE RTR RTS SBCD Scc	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 9-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices ROtate Left ROtate Right ROtate Left ROtate Right ROtate Left ROtate Right ROtate Left ROtate Right ROtate Right Rotate Sight with eXtend ReTurn from Exception ReTurn and Restore ReTurn for Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127- <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay)</ea></ea></register></ea></ea>	W W WL WL W W W B B WL B WL B WL B WL B
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEGC NEGC NEGC NOP NOT OR OR OR OR OR OR RESET ROL ROR ROX ROX ROX ROX ROX ROX ROX ROX ROX	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 9-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise 0R Bit-wise 0R Bit-	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> ====================================</ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay)</ea></ea></register></ea></ea></ea>	W WL WL WL W W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEGS NOP NOT OR ORI PEA RESET ROL ROR ROR ROXL ROXL ROXL ROXL ROXL SBCD SCC STOP SUB	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 8-bit immediate MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise Rester al devices ROSET all external devices ROtate Left ROtate Right ROtate Left ROtate Right ROtate Left ROtate Left ROtate Left ROtate Left ROtate Left ROtate Left ROtate Left ROtate Left ROtate Left with eXtend ReTurn from Exception ReTurn and Restore ReTurn for Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary	<ea>,SR SR,<ea> USP,An, An,US -register list>, Dn,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> === <ea> === === #<1-8>,Dy Dx,Dy <ea> #<1-8>,Dy</ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy,<ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W WL WL U U W W B WL B WL B WL B WL B
MOVE MOVE MOVEA MOVEA MOVEP MOVEQ MULS MULU NBCD NEG NEGS NOP NOT OR OR OR OR OR OR OR OR OR OR RESET ROL ROX ROX ROX ROX ROX ROX ROX ROX ROX ROX	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE S-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR Cate Left Rotate Left Rotate Left with eXtend Rotate Left with eXtend ReTurn from Exception ReTurn and Restore ReTurn from Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary SUBtract binary from An	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127- <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W WL WL W W W B B WL B WL B WL B WL B
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEG NEG NOP NOT OR OR OR OR PEA RESET ROL ROR ROXL ROXL ROXL ROXL ROXL ROXL SBCD SCC STOP SUB SUBA SUBA SUBA	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE Peripheral MOVE 8-bit immediate MULtiply Signed Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise 0R Bit-wise 0R with Immediate Push Effective Address RESET all external devices ROtate Left ROtate Right ROtate Left ROtate Right with eXtend RoTurn from Exception ReTurn from Exception ReTurn and Restore ReTurn from Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary SUBtract binary SUBtract Immediate	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W WL WL WL L W W B B WL B WL B WL B W
MOVE MOVE MOVEA MOVEA MOVEQ MOVEQ MULS MULU NBCD NEGS NEGS NOP NOT OR OR OR OR OR OR OR OR OR OR OR OR OR	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE S-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Bit-wise OR Carte Right with Immediate Push Effective Address ROtate Left ROtate Right with eXtend ROtate Left with eXtend ROtate Left with eXtend Rotate Left with eXtend Rotate Left with eXtend ReTurn from Exception ReTurn and Restore ReTurn from Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary SUBtract Dinary from An SUBtract Immediate	<ea>,SR SR,<ea> USP,An, An,US Dn,x(An), #<-128,+127> <ea>,Dn <ea>,Dn <ea>,Ca <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea< td=""><td>P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea></td><td>W W WL WL WL W W W B B WL B WL B WL B W</td></ea<></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W WL WL WL W W W B B WL B WL B WL B W
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEGC NEGC NEGC NOP NOT OR OR OR OR OR OR OR OR OR COR ROS ROS ROS ROS ROS ROS ROS ROS ROS R	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 9-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise 0R Bit-wise 0R with Immediate Push Effective Address ROTATE Address ROTATE Lettral devices ROTATE Lettral devices ROTATE Lettral devices ROTATE Left ROTATE Left with eXtend ROTATE Right with eXtend ROTATE Right with eXtend ROTATE Right with eXtend ReTurn from Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary SUBtract Linmediate SUBtract J-bit immediate	<ea>,SR SR,<ea> USP,An, An,US Dn,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W WL WL W W B B WL B WL B WL B WL B W
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEG NEG NOP NOT OR OR PEA RESET ROL ROR ROR ROXL ROR ROXL ROX ROXL ROX ROX ROX SUB SUBA SUBA SUBA SUBA	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 9-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices ROtate Right ROtate Right ROtate Right ROtate Right ROtate Right with eXtend ReTurn from Exception ReTurn and Restore ReTurn for Mubroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract Immediate SUBtract Immediate SUBtract 3-bit immediate SUBtract eXtended	<ea>,SR SR,<ea> USP,An, An,US Dn,x(An), #<-128,+127- <ea>,Dn <ea>,Ca> <ea> <ea> <ea> <ea> <ea> <ea> <ea> <e< td=""><td>P <ea>,An x(a),Dn ,Dn Dn,<ea> Dx,Dy,<ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></ea></td><td>W W WL WL WL W W W B WL B WL B WL B WL</td></e<></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An x(a),Dn ,Dn Dn,<ea> Dx,Dy,<ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></ea>	W W WL WL WL W W W B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEGC NEGC NEGC NOP NOT OR OR OR OR OR OR OR OR OR OR COR ROS ROS ROS ROS ROS ROS ROS ROS ROS R	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE B-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise 0R Bit-wise 0R Bit-	<ea>,SR SR,<ea> USP,An, An,US USP,An, An,US Dn,x(An) , #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> = = = = = = = = = = = = = = = = = = =</ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W WL WL WL W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEG NEG NOP NOT OR ORI PEA RESET ROL ROR ROR ROR ROXL ROR ROXL ROX RTE RTR RTS SBCD SCC STOP SUB SUBA SUBA SUBA SUBA SUBA SUBA SUBA	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE Peripheral MOVE 8-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices ROtate Left ROtate Right ROtate Left ROtate Right ROtate Left ROtate Right with eXtend ReTurn from Exception ReTurn from Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary SUBtract binary SUBtract S-bit immediate SUBtract 3-bit immediate SUBtract eXtended SWAP words of Dn Test & Set MSB & Set N/Z-bits	<ea>,SR SR,<ea> USP,An, An,US on,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> =<a> =<a> =<a> =<a> =<a> =<a> =<a> =<</ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W WL WL WL L W W B B WL B WL B WL B W
MOVE MOVE MOVEA MOVEA MOVEA MOVEP MOVEQ MULS MULU NBCD NEGC NEGC NEGC NOP NOT OR OR OR OR OR OR OR OR OR OR OR OR OR	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE S-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise 0R Bit-wise 0R Bit-wise 0R with Immediate Push Effective Address RESET all external devices ROtate Left ROtate Left with eXtend Rotate Left with eXtend Rotate Left with eXtend Rotate Left with eXtend ReTurn from Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary SUBtract binary from An SUBtract Immediate SUBtract 3-bit immediate SUBtract 3-bit immediate SUBtract Stended SWAP words of Dn Test & Set MSB & Set N/Z-bits Evecute TBAP Excemtion	<ea>,SR SR,<ea> USP,An, An,US Dn,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> #<data>,cea> <ea> #<1-8>,Dy Dx,Dy <ea> #<data>,cea> <ea> #<data>,Ca SD SD SD SD SD SD SD SD SD SD SD SD SD</data></ea></data></ea></ea></data></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W WL WL WL W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEG NEG NOP NOT OR OR OR OR PEA RESET ROL ROR ROXL ROR ROXL ROXL ROXL ROXL RO	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 9-bit immediate MULtiply Signed Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise 0R Bit-wise 0R with Immediate Push Effective Address ROtate Left ROtate Right ROtate Right ROtate Right with eXtend ROtate Right with eXtend ROtate Right with eXtend ROtate Right with eXtend Rotate Left ROtate Right with eXtend Rotate Left ROtate Right with eXtend ReTurn from Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary SUBtract Limmediate SUBtract Z-bit immediate SUBtract Address SUBtract Set MJZ-bits Subtract eXtended SWAP words of Dn Test & Set MJZ bit Ext	<ea>,SR SR,<ea> USP,An, An,US n,x(An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Dn <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W U WL WL WL W W B B WL B WL B WL B WL B WL B WL B WL B WL B WL B WL B WL B WL B WL B B B B B B B B B B B B B
MOVE MOVE MOVEA MOVEA MOVEQ MOVEQ MULS MULU NBCD NEGS NEGS NOP NOT OR OR OR OR OR OR OR OR OR OR OR OR OR	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE B-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise	<ea>,SR SR,<ea> USP,An, An,US Dn,x(An), #<-128,+127> <ea>,Dn <ea>,Dn <ea>,Ca <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea< td=""><td>P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea></td><td>W W U WL WL W W W B B WL B WL B WL B WL</td></ea<></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn ,,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W U WL WL W W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEGC NEGC NEGC NOP NOT OR OR OR OR OR OR OR OR OR OR OR COR ROX ROX ROX ROX ROX ROX ROX ROX ROX R	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE B-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate with eXtend No OPeration Form one's complement Bit-wise 0R Bit-wise 0R with Immediate Push Effective Address ROTATE REST all external devices ROTATE Right ROTATE Left with eXtend ROTATE Right ROTATE Left with eXtend ROTATE Right ROTATE Left with eXtend ROTATE Right with eXtend ROTATE Right with eXtend ReTurn from Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary SUBtract binary SUBtract Solve for MAN SUBtract Solve for MAN SOLVE	<ea>,SR SR,<ea> USP,An, An,US USP,An, An,US USP,An, An,US (An), #<-128,+127: <ea>,Dn <ea>,Dn <ea>,Ca <ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea< td=""><td>P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea></td><td>W WL WL WL WL W W B B WL B WL B WL B WL</td></ea<></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W WL WL WL WL W W B B WL B WL B WL B WL
MOVE MOVE MOVEA MOVEA MOVEQ MULS MULU NBCD NEG NEG NEG NEG NOP NOT OR ORI PEA RESET ROL ROR ROR ROR ROR ROR ROXL ROR ROXL ROX ROR SUB SUBA SUBA SUBA SUBA SUBA SUBA SUBA	To SR From SR USP to/from Address Register MOVE Address MOVE Multiple MOVE Peripheral MOVE 9-bit immediate MULtiply Signed MULtiply Unsigned Negate BCD NEGate NEGate with eXtend No OPeration Form one's complement Bit-wise OR Bit-wise OR with Immediate Push Effective Address RESET all external devices ROtate Left ROtate Right ROtate Left ROtate Right ROtate Left with eXtend Rotate Right ROtate Left with eXtend Rotate Right ROtate Left with eXtend ReTurn from Exception ReTurn and Restore ReTurn from Exception ReTurn and Restore ReTurn from Subroutine Subtract BCD with eXtend Set to -1 if True, 0 if False Enable & wait for interrupts SUBtract binary SUBtract binary from An SUBtract abit immediate SUBtract - bit immediate SUBtract - bit immediate SUBtract TRAP Exception TRAPV Exception if V-bit Set TeST for negative or zero Deallocate Stack Frame	<ea>,SR SR,<ea> USP,An, An,US Dn,x(An), #<-128,+127: <ea>,Dn <ea>,ea> <ea> <ea> <ea> <ea> <ea> <ea> <ea> =<a> =<a> =<a> =<a> =<a> =<a> =<a> =<</ea></ea></ea></ea></ea></ea></ea></ea></ea></ea></ea>	P <ea>,An ,<ea> <ea>,<register list<br="">x(An),Dn >,Dn Dn,<ea> Dx,Dy, <ea> -(Ax),-(Ay) <ea>,Dn</ea></ea></ea></register></ea></ea></ea>	W W WL WL WL W W B B WL B WL B WL B WL

V 5.0

THE ARM FAMILY

We use the ARM family in this course to illustrate computer architecture for several reasons. First, it illustrates all the important elements of an instruction set architecture. Second, it is easy to understand and has a very gentle learning curve in comparison with some other processors; for example an add operation is specified by ADD r1, r2, r3 which adds register r2 to register r3 and puts the result in r1. What could be simpler? Third, the ARM has some very interesting attributes such as *predicated execution* that make it an excellent vehicle for teaching computer architecture.

THE ARM REGISTER SET

The ARM has 16 *general-purpose* 32-bit data registers, r0 to r15, that can be used by the programmer to store temporary variables. However, registers r14 and r15 that special purposes. Register r14 holds a subroutine return address after a subroutine call. Consequently, you should use r14 only to deal with return addresses.

Register r15 holds the program counter, the next instruction to be executed. You cannot use r15 for any other purpose. The ARM is highly unusual in this respect because all other microprocessor families have a dedicated program counter that is not normally directly accessible by the programmer. The ARM programmer must not use r15 as a general-purpose data register as that would crash the computer.

THE INSTRUCTION

Computer instructions are executed sequentially, one by one in turn, unless a special instruction deliberately changes the *flow* of control or unless an event called an *exception* (interrupt) takes place.

The structure of instructions varies from machine to machine. The format of an instruction running on an Intel processor is different to the format of an instruction running on a 68K or an ARM (even though the instructions might do the same thing). Instructions are classified by what they do and by the number of operands they take. The three basic instruction types are: *data movement* that copies data from one location to another, *data processing* that operates on data, and *flow control* that modifies the order in which instructions are executed. Instruction formats can take zero, one, two, three, or even four operands. Consider the following examples of instructions with zero to three operands. In these examples operands P, Q, and R may be memory locations or registers.

Operands	Instruction	Effect
Three	ADD P ,Q,R	Add R to Q and put the result in P
Two	ADD P ,Q	Add P to Q and put the result in P
One	ADD P	Add P to an accumulator
Zero	ADD	Add the top two items on the stack and push the result

A three-address computer instruction can be written

```
operation destination, source1, source2
```

where operation defines the nature of the instruction, source1 is the location of the first operand, source2 is the location of the second operand, and **destination** is the location of the result. The instruction ADD r3, r1, r7 adds r1 and r7 to get r3.

This is a little pedantic, but... When we say that r1 is added to r7, we really mean that the *contents* of r1 is added to the contents of r7. However, it gets boring being so precise so we often just use a register's name when we really mean the contents of that register.



Microprocessors don't implement three-address instructions that access memory; you can access only registers. It's not the fault of the instruction designer. It's a limitation imposed by the practicalities of computer technology. Suppose that a computer has a 32-bit address that allows a total of 2^{32} bytes of memory to be accessed. The three address fields, *P*, *Q*, and *R* needed to implement ADD **P**, Q, R would each be 32 bits, requiring $3 \times 32 = 96$ bits to specify the operands. Assuming a 16-bit operation code (allowing up to $2^{16} = 65,536$ instructions), the total instruction size would be 96 + 16 = 112 bits or 14 bytes. This instruction format is shown below.



(a) Format of a hypothetical instruction with three address fields

4	32 bits			
17 bi	ts	5 bits	5 bits	5 bits
Op-code	Control bits	Destination register	Source 1 register	Source 2 register

(b) Format of a hypothetical instruction with a register-to-register architecture

Part (b) of the above figure illustrates a typical RISC instruction format. This uses a register-to-register architecture that allows three registers to be specified. Each has a 5-bit address field which allows 32 registers.

POSSIBLE THREE-ADDRESS INSTRUCTION FORMATS

Computer technology developed when memory was very expensive indeed. Implementing a 14-byte instruction was not costeffective in the 1970s. Even if memory had been cheap, it would have been too expensive to implement 112-bit-wide data buses to move instructions from point to point in the computer. Finally, main memory is intrinsically slower than on-chip register.

The modern RISC processor allows you to specify three addresses in an instruction by providing three 5-bit operand address fields. This restriction lets you select from one of only 32 different operands that are located in registers within the CPU itself. By using on-chip registers to hold operands, the time taken to access data is minimized because no other storage mechanism can be accessed as rapidly as a register. An instruction with three 32-bit operands requires 3×5 bits to specify the operands which allows a 32-bit instruction to use the remaining 32 - 15 = 17 bits to specify the instruction.

We'll use the ADD instruction to add together four values in registers r2, r3, r4, and r5. In the following fragment of code, the semicolon indicates the start of a comment field that is not part of the executable code. This code is typical of RISC processors like the ARM.

ADD	r1, r2,r3	;r1	=	r2	+	r3							
ADD	r1 , r1, r4	;r1	=	r1	+	r4							
ADD	r1, r1,r5	;r1	=	r1	+	r5 =	= r2	+	r3	+	r4	+	r5



TWO-ADDRESS MACHINES

A CISC machine like, the 68K, has a two-address instruction format. Clearly, you can't execute $P \leftarrow Q + R$ with just two operands. You can execute $Q \leftarrow P + Q$. One operand appears *twice*, first as a source and then as a destination. The operation ADD P, Q implements $[Q] \leftarrow [P] + [Q]$. The price of a two-operand instruction format is the destruction of one of the source operands.

Most computer instructions can't directly access two memory locations. Typically, the operands are either two registers or one register and a memory location; for example, the 68K ADD instruction can be written:

Instruction	RTL definition	Mode
ADD D0, D1	[D1] ← [D1] + [D0]	Register-to-register
ADD P, D2	[D2] ← [D2] + [P]	Memory-to-register (P is a directly address memory location)
ADD D7, p	[P] ← [P] + [D7]	Register-to-memory

The 68K has seven general-purpose registers D0 to D7; there are no restrictions on the way in which you use these registers; that is, if you can use Di you can also use Dj for any i or j from 0 to 7.

ONE-ADDRESS MACHINES

A one-address machine specifies only one operand in the instruction. The second operand is a register called an *accumulator* that always takes part in the operation. For example, the one-address instruction ADD P means $[A] \leftarrow [A] + [P]$. The notation [A] indicates the contents of the accumulator. A simple high-level operation R = P + Q can be implemented by the following fragment of 8-bit code (from a 6800 8-bit processor).

LDA	P	;load accumulator with H	2
ADD	Q	;add Q to accumulator	
STA	R	;store accumulator in R	

Eight-bit machines have one-address architectures. Eight-bit code is verbose, because you have to load data into the accumulator, process it, and then store it in memory to avoid it being overwritten by the next data-processing instruction. One-address machines are still widely used in embedded controllers in low-cost systems.

DATA SIZE

I don't want to go into the details of data size here because it's a large topic. However, I do need to introduce a basic concept. If a computer can move data from *A* to *B*, or can perform an operation on data, we need to know the number of bits in a word being moved or processed.

The very first microprocessor, the Intel 4004, used a 4-bit word because the technology at that time could not economically fabricate chips capable of handling longer wordlengths. The 4004 was able to handle 4-bit values.

Very shortly after the introduction of the 4040, 8-bit microprocessors appeared. An 8-bit word is called a byte and operations in 8-bit computers are applied to bytes. You can't perform a 6-bit operation and you can't perform a 10-bit operation. Although an 8-bit word can handle text efficiently, it is unsuited to the representation of addresses or to any quantity that can have more than 256 possible values. Eight-bit processors can concatenate two 8-bit words to create a 16-bit address.

Over the years, microprocessors grew in complexity to support 16-bit, 32-bit and 64-bit words. The larger the word size, the more work you can do in an instruction. In this course we use ARM processors that have 32-bit data words and 32-bit addresses.

However, just as 4-bit and 8-bit words are too short to represent many types of data, 32-bit and 64-bit words are often too big. For example, if you use ASCII-encoded text, each character requires 8 bits. If you put an ASCII character in a 32-bit register, 24 bits are unused. This represents an inefficient use of storage. So, programmers often employ tricks to pack more than one character in a word.

SUB-WORD OPERATIONS

If you wish to access individual bytes in a 16- or 32-bit processor, you need special instructions. The 68K family deals with 8bit, 16-bit, and 32-bit data by permitting most data-processing instructions to act on an 8-, 16-, or 32-bit slice of a register; for example ADD.B D0, D1, ADD.W D0, D1 and ADD.L D0, D1 each adds the contents of data register D0 to D1 and puts the result in D1. The suffix .B specifies an 8-bit byte operation, .W specifies a 16-bit word operation, and .L specifies a 32-bit longword operation. In each case the bits taking part in the operation are the low-order bits, and bits not taking part in the operation do not change. For example, the RTL definition of ADD.W D1, D3 is

 $[D3_{(0:15)}] \leftarrow [D3_{(0:15)}] + [D1_{(0:15)}]$

RISC processors do not (generally) support 8- or 16-bit data-processing operations on 32-bit registers, but they do support 8-bit and 16-bit memory accesses. Consider the following ARM examples.

LDR	r0, [r1]	;load r0 with the 32-bit contents of memory pointed at by register r1
LDRB	r0, [r1]	;load r0 with the 8-bit contents of memory pointed at by register r1
LDRH	r0, [r1]	;load r0 with the 16-bit contents of memory pointed at by register r1

There is also a similar set of store mnemonics with the forms STR, STRB, and STRH.

In 68K terminology 8/16/32 bit values are called byte/word/longword, whereas ARM processor literature uses byte/half word/word.

Suppose a processor supports operations that act on a subsection of a register. What happens to the bits that do not take part in the operation? Assume that a register is partitioned as figure (a) demonstrates.

Figure (b) shows how some processors deal with the problem by ignoring higher-order bits. If you add the two-low-order bytes in a 32-bit word, bits 0 to 7 are added together and bits 8 to 31 remain unchanged; for example, 0x12345678 + 0x1111111 = 0x12345689. This is true of the 68K processor.

Figure (c) demonstrates an alternative arrangement. Here, the bits not taking part in an operation are automatically cleared. In this case, 0x12345678 + 0x11111111 = 0x00000089.

In (d) the bits not taking part in an operation are sign-extended. This means that if you add two bytes in a 32-bit word, the result is sign extended to 32 bits. The 68K treats the contents of address registers in this way. If you perform a 16-bit operation on an address register, the result is sigh-extended to 32 bits.



PROBLEM SET 3

These questions ask you about the role of registers in computer architecture, the role of addressing modes, and the design of computer instruction sets.

- 1. In the context of microprocessors, what is a *user-visible* register?
- 2. Modern microprocessors have more registers than previous generations of microprocessors. Why?
- 3. Registers are used in different ways by different microprocessor families (e.g., Intel IA32, Motorola 68K, ARM etc.). Describe some of the differences in the way in which registers are used and comment on the relative merits.
- 4. A special-purpose computer's instruction set is 24 bytes wide. This is a three-operand load and store (register-to-register) machine. If this computer provides 64 general-purpose registers, how many different instructions can be implemented?
- 5. A 32-bit computer with a 32-bit instruction word uses 122 different instructions. If this computer has a three-address register-to-register instruction set, how many registers can be supported?
- 6. A computer devotes only 4 bits in its instruction word to the selection of one of 16 registers. Can you suggest of ways of providing more than 16 registers while keeping the number of bits in an instruction that selects a register at 4?

- 7. What are the various groups of instruction types implemented by typical microprocessors (i.e., how are instructions classified)? Give examples of different types of instructions.
- 8. What are the relative merits of one-address, two-address, and three-address instructions?
- 9. Under what circumstances is it possible to have a zero-address computer?
- 10. Are there occasions where 4-address or even 5-address instructions could prove useful?

USING THE KEIL SIMULATOR

The processor in the PC is not a member of the ARM family. It's usually a member of Intel's IA32 family or an AMD processor. However, you can run ARM code processor on your PC using a program from KeilTM. This can be found at <u>www.keil.com</u>. The Keil package, called μ Vision4, is very sophisticated and is intended for engineers designing embedded ARM-based systems. Consequently, it includes far more facilities than we need. The demonstration version that you can download for a PC is limited to assembly-level programs smaller than 32K bytes. This restriction is not be a problem for students.

Essentially, μ Vision4 is an IDE (integrated development system) that is *project-based*; that is, each new program must be part of a project. You begin by creating a project (i.e., a container for all your files) and then select the target processor you are going to use. You create source files (in our case, these will be assembly language files) and then you build your application (i.e., create code for your chosen processor). μ Vision4 allows you to construct projects with multiple source files and files in C or C++, although we will not be using these facilities. Having built your file, you can execute it and follow the progress of its execution.

Let's step through the process of creating a program. Note that this package will continue to be upgraded during its life and there may be differences between these examples and the system you are using. However, the sequence of operations should remain substantially the same. On loading μ Vision4 you are presented with the following screen.

🔣 E: \	Cengag	jeBook	\Z_ARM	I\Chap3(Ques60.u	ιvproj - μ'	Vision	4											_	. 🗆 🗵	1
<u>F</u> ile	<u>E</u> dit	<u>V</u> iew	<u>P</u> roject	Fl <u>a</u> sh	<u>D</u> ebug	Pe <u>r</u> ipher	als <u>1</u>	<u>[</u> ools	<u>s</u> vcs	<u>W</u> indow	ı <u>H</u> elp	р									
	🚰 🖫	9	8 B)	89		P	際日	周段		罪 //言 /	k 🖄)		Ŧ	B) (*	0	۲	06	۶ 🚓	•	
1 🧼		0	LOAD	Target 1		-	*	📥 🗟	4												
																					I
																					l
																					l
																					l
																					l
																					l
																					l
																					l
																					l
																					l
																		Simul	ation		

To start, select **<u>P</u>roject** from the upper row of tabs and then **New \muVision Project** from the pull-down window. This brings up the **Create New Project** Window and you create a project name in the selected directory. I will use *FirstExample*. The development system automatically appends a file type to create **FirstExample.uvproj**.

When you hit **save**, a new window will automatically appear that invites you to select the device you are going to use. In this case it is the ARM (see the figure below). If you select the ARM pull-down window, it will offer various ARM versions. I used the **ARM7 (Big Endian)** version. Once you have elected the processor, a return is made to the basic project window.

Select Device for Target 'Target 1'		×
CPU		
Vendor: ARM Device: Toolset:		
Data base	Description:	Choose the processor you wish to simulate from this list.
ARM7 (Big Endian)	4	
	OK Cancel	Help

Selecting the target device

The next step is to create a source file. Click **File** in the main window. This will open a file window with the default name Text1. Now you can enter you source program.

After entering the program you need to save it. This is done in the normal Windows way: select **file** and then **save**. You then have to give it a name. I chose **FirstExample.asm**. Note that I used the extension **.asm** (assembly langue) because the development system does not know which type of source file you are creating. The following image shows the screen after the program has been entered and saved.

🔣 E:\Cengage	eBook\Ger	ngageWorkBo	ok\FirstExample.uvproj - μVision4	
<u>F</u> ile <u>E</u> dit <u>\</u>	<u>v</u> iew <u>P</u> roj	ect Fl <u>a</u> sh	<u>D</u> ebug Pe <u>r</u> ipherals <u>T</u> ools <u>S</u> VCS <u>W</u> indow <u>H</u> elp	
🗋 💕 🛃	🍠 🕺 🖬	à 🗈 🔊 (≥ <= ⇒ 🥐 雅 雅 雅 微 蕾 譯 /////////////////////////////////	🔹 🔜 🥐 🔍 💿 🔿 🚷 🔳 🗸
🕸 🔛	i 🖓 🔍 🖓	Target 1	💌 🔊 📥 🔁	
FirstE	xample.asn	n*	,	▼ X
01 02 03	AREA ENTRY	FirstExa	mple, CODE, READONLY	Assembler directives
04 05	MOV MOV	r1,#2 r2,#3	;load r1 with 2 ;load r2 with 3	
06	MOV MOV	r3,#4 r4,#5	;load r3 with 4 ;load r4 with 5	
08	ADD ADD	r1, r1, r2	;add r1 and r2 and put the result in r1	
10	ADD	r1, r1, r4	; add r1 and r4 and put the result in r1	
12	MOV	10,11	;put the sum in ro	
13	END			
				Simulation

This program is simple. It loads register with numbers (literals), adds them together and then moves the result into register r0. Note that there are three lines that are not part of the assembly language. These are **assembler directives** that tell the assembler things it needs to know. The first assemble directive is

AREA FirstExample, CODE, READONLY

The purpose of this AREA directive is to name the region of memory where the program will be located. In this case we've used FirstExample. The parameter CODE indicates that the data will be code rather than data. The third parameter READONLY indicates that the memory is read-only because are not going to alter its contents.

The ENTRY directive simply tells the assembler that the code is to be executed from this point. It's the starting point.

The final directive, END, indicates the end of the program and that there is no further code beyond this point.

The next step is to tell the project manager about the assembly file we've just created. Click on **Project** to select the project drop-down menu and then select **Manage**. Then select **Components, Environment, Books...** from the secondary drop-down window. The figure (below left) shows this situation. Now click <u>Add Files</u> to select your source file. You will have to change the **File of Type** box from its default C Source file (*.C) to ASM Source file. Having done that, you should have the situation below right with one file displayed. Then click OK to end the sequence.

ponests, Environment and Books	×	Components, betroment and Backs	×
ect Components Folders (Expressore Books		Project Components Folders/Estensore Books	1
Proper Targets (1) X + 4 Graue (1) X + 4 Stores (1)	fine X + 4	Proset Targets (1) X + 4 Groupe (1) X + 4 Plane Remot 1 Bases Group 1	× + 4
			The source file we
			created.
Set as Current Target	dd Res		Jata Files
DK Caret	Hela	DK Carcel	Hels

At this stage, we have a project, a processor, and a source file. Now we need to create the environment and assemble the code. Click on **Project** and then **Build target** from the drop-down window. The following image shows the screen after we have built the target.

🔣 E: \	CengageBook\G	iengageWorkB	ook\FirstExample.uvproj - µVisi	on4			
<u>F</u> ile	<u>E</u> dit <u>V</u> iew <u>P</u>	roject Fl <u>a</u> sh	<u>D</u> ebug Pe <u>r</u> ipherals <u>T</u> ools <u>S</u> \	/CS <u>W</u> indow	<u>H</u> elp		
	🚰 🛃 🗿 3	B 🗈 🤊	P ← → P B B B		2		
1 🔌	🎬 🎬 🧼 🗮	Target 1	- 🔊 📥 🗟				
	FirstExample.a	ism				▼ ×	
01	AREA	FirstExa	mple, CODE, READONLY				
02	ENTR	RY .				Note how the assemb	oler has
03						formatted your code	It uses
04	MOV	r1, #2	;load r1 with 2			formatical your code.	11 4303
05	MOV	r2,#3	;load r2 with 3			color to distinguish b	etween
06	MOV	r3,#4	;load r3 with 4			code and comments a	and it
07	MOV	r4,#5	;load r4 with 5			highlights literals	
08	ADD	r1, r1, r2	;add r1 and r2 and p	ut the res	ult in rl	mgninghts neerais.	
09	ADD	r1, r1, r3	;add r1 and r3 and p	ut the res	ult in r1		
10	ADD	r1, r1, r4	;add rl and r4 and p	ut the res	ult in rl		
	MOV	ru, ri	;put the sum in r0				
12	END					-	
		•					
Build	Output					д Х	
Bui 1	d target !T	arget 1!					
asse	mbling Firs	tExample.a	570				
link	ing ing	oblighter at					
Prog	ram Size: C	ode=32 RO-0	data=0 RW-data=0 ZI-dat	ta=0			
"Fir	stExample.a	xf" - 0 Er	cor(s), 0 Warning(s).				
						-	
•							

The **Build Output** window shows the status of the process. Here we are interested only in the magic words **0 Error(s)** that indicate all went well. Had there been any errors in the code, we would have been informed and would have had to edit the source file and then rebuild it. This cycle is repeated until there are no errors reported.

The final step is to run the program in the simulator. To do this select **Debug** from the main window and click on **Start/Stop Debug Session**. This brings up the EVALUATION MODE box that tells you are restricted to 32K. Click on OK to bring up the simulator window as shown below.



We don't need all this. Using normal Windows features we can resize and remove windows not of immediate importance to get the following image. We now have three windows. On the left there is a window, **Registers**, showing the contents of registers. All values are in hexadecimal. The other items (which have expansion boxes) are not of interest at the moment; these describe the status of the processor and the value of carry and overflow bits etc.

The window on the top right, **Disassembly**, is not necessary in this example. We could have closed it but have left it open in order to demonstrate the structure of the program in memory. This window takes code in memory and transforms it back into ARM processor op-code and mnemonics. However, should it encounter data (i.e., number or text etc.) it will produce meaningless code. In this example, you can see each of the instructions. The leftmost column is the memory address, the next column the 32-bit value (in hexadecimal) at that location. The third column contains the disassembled op-code; for example, at address 0x0000000C we have the value E3A04005 which translates into MOV **r4**, #5. Note that this window also contains the original source code instructions and the line numbers.

Version 1 [WORKBOOK FOR COMPUTER ORGANIZATION AND ARCHITECTURE: THEME AND VARIATIONS]

🖫 E:\CengageBook\GengageWorkBook\FirstExample.uvproj - µVision4	
<u>F</u> ile <u>E</u> dit <u>V</u> iew <u>P</u> roject Fl <u>a</u> sh <u>D</u> ebug Pe <u>r</u> ipherals <u>T</u> ools <u>S</u> VCS <u>W</u> indow <u>H</u> elp	
]	- 🗈 🖉 🥥 🧉 🔗 🕭 📰 - 🍕
Ĩ₩ĨЩV ₩₩₩₩ ₩ Ŷ ΣЩЩ<u>G</u>⊟₩₩▼Щ▼₩▼₩▼₩▼ X▼ :	
Registers 4 X Disassembly	<u> </u>
Register Value 4: MOV r1,#2 ;load r1 with	h 2
Current Current	-
R0 0x0000000 5: MOV r2,#3 ; load r2 with	n 3
R1 0x0000000 6: MOV r3 t4 :load r3 with	h 4
R2 0x0000000 0x0000008 F3203004 MOV P3 #0x00000004	1 1
R3 0x0000000 7: MOV r4.#5 :load r4 with	h 5
R4 0x0000000 0x000000C E3A04005 MOV R4,#0x00000005	
R5 0x0000000 8: ADD r1,r1,r2 ;add r1 and r	r2 and put the result in r1
R6 0x0000000 0x00000010 E0811002 ADD R1,R1,R2	-
9: ADD r1,r1,r3 ;add r1 and r	r3 and put the result in r1
R9 0x0000000 0x00000014 E0811003 ADD R1,R1,R3	
R10 0x0000000 10: ADD r1,r1,r4 ;add r1 and r	r4 and put the result in r1
R11 0x0000000 0x0000018 E0811004 ADD R1,R1,R4	
R12 0x0000000 11: MOV r0,r1 ;put the sum in r	r0
R13 (SP) 0x00000000 0x0000001C E1A00001 MOV R0, R1	
R14 (LR) 0x00000000	•
R15 (PC) 0x0000000	▼ X
UT APEA EirstExample CODE PEADONLY	
Em SPSR (XU0000000 01 01 AREA FIISCERAMPIC, CODE, READORDI	
rast interrupt rast inter	
US MOV r2,#3 ;load r2 with 3	
Hont O6 MOV r3,#4 ;load r3 with 4	
Undefined 07 MOV r4,#5 ;load r4 with 5	
Internal 08 ADD r1, r1, r2 ; add r1 and r2 and pt	ut the result in r1
PC \$ 0x0000000 09 ADD r1,r1,r3 ;add r1 and r3 and pt	ut the result in r1
Mode Supervisor 10 ADD r1, r1, r4 ; add r1 and r4 and pt	ut the result in r1
States 0 MOV r0, r1 ; put the sum in r0	
Sec 0.0000000 12	
13 EUD	
	Simulation

The window below is the same as above except that we've closed the disassembly window and resized to reduce clutter.

🔣 E:\	Cengag	jeBool	c\Genga	jeWorki	Book\Fir	stExam	ple.uvp	roj - µ\	/ision4									_	
<u>F</u> ile	<u>E</u> dit	<u>V</u> iew	<u>P</u> roject	Fl <u>a</u> sh	<u>D</u> ebug	Pe <u>r</u> iph	erals	<u>T</u> ools	<u>s</u> vcs	<u>W</u> indow	<u>H</u> elp								
	💕 🛃		X Pat	8 9			la lit	四四		運 //E //j	2			-		۴ 🔍	٠	් ර) 🊓
RST	e 🛛) [}	₹ 0 ()	*{}		1. 🖪	-	- 🐯	-	😼 • 🔜 •		-	🎌 - 📔	3					
Regis	ters				џх		FirstEx	ample.a	ism										• ×
Regi	ster		Value			01		AREA	Fi	rstExamp	ole, (CODE, H	READONI	LY					
(Current					02													
	R0		0x0000	00000		03													
	R1		0x0000	00000		->04		MOV	rl	,#2	;loa	d ri w	ith 2						
	R2		0x0000	00000		05		MOV	r2	, #3	;10a	d r2 w	ith 3						
	R3		0x0000	00000		06		MOV	r 3	,#4	;10a	d r3 w:	ith 4						
	R4		0x0000	00000		07		MOV	r4	,#5	;10a	dr4w:	ith 5						
	R5		0x0000	00000		08		ADD	r1	, r1, r2	;add	r1 and	d r2 ai	nd put	the	result	; in	r1	
	R6		0x0000	00000		09		ADD	r1	,r1,r3	;add	r1 and	d r3 ai	nd put	the	result	; in	r1	
	R7		0x0000	00000		10		ADD	r1	,r1,r4	;add	r1 and	d r4 an	nd put	the	result	; in	r1	
	R8		0x0000	00000		11		MOV	rO	, r1	;put	the su	um in 1	r0					
	R9		0x0000	00000		12													
	R10		0x0000	00000		13		END											
	R11		0x0000	00000															
	R12		0x0000	00000															
	R13	(SP)	0x0000	00000															
	R14	(LR)	0x0000	00000															
	R15	(PC)	0x0000	00000	-														
E Pr	oject	🗮 Regi	isters																Ð
																		Simula	itio //

The final step is to run the code. We can run it in several different ways. Here, we are going to execute it line by line so that we can observe the way in which the contents of the registers change after each instruction. In the image below, we have clicked on the *step one line icon* (highlighted in the image) and the first instruction has been executed. Note that in the register list, r1 is highlighted and it has the value 2 (because it was loaded with 2). The contents of the program counter, r15, are 4 because it now points to the second instruction.



The next image shows the screen after we have clicked the step-in button five times and have executed the first five instructions.

🖫 E:\CengageBook\GengageWorkBook\F	irstExample.uvproj - µVision4	
<u>F</u> ile <u>E</u> dit <u>V</u> iew <u>P</u> roject Fl <u>a</u> sh <u>D</u> ebu	g Pe <u>r</u> ipherals <u>T</u> ools <u>S</u> VCS <u>W</u> indow <u>H</u> elp	
🗋 😂 🛃 🍠 🕉 🏜 🖄 🤗 🕒	◇ ◇ 教 教 孝 孝 // // 🎽 📃 💌 💽 🖉	ی 🔗 د 🌢
🛿 👫 🗏 🔕 🤭 🖓 🖓 😗 🌩 🔼	◙, 🕞 🚍 🖗 - 💷 - 👳 - 🔜 - 🔜 - 🔛 - 🔛 -	
Registers 4 ×	FirstExample.asm	▼ ×
Register Value	01 AREA FirstExample, CODE, READONLY 02	
Current R0 0x00000000 R1 0x000000003 R2 0x00000004 R3 0x00000005 R4 0x0000000 R5 0x0000000 R6 0x0000000 R7 0x0000000 R8 0x0000000	02 03 04 MOV r1, #2 ; load r1 with 2 05 MOV r2, #3 ; load r2 with 3 06 MOV r3, #4 ; load r3 with 4 07 MOV r4. #5 ; load r4 with 5 08 ADD r1, r1, r2 ; add r1 and r2 and put the result 10 ADD r1, r1, r4 ; add r1 and r4 and put the result 11 MOV r0, r1 ; put the sum in r0	in r1 in r1 in r1
R9 0x0000000 R10 0x0000000 R11 0x0000000 R12 0x0000000 R13 (SP) 0x0000000 R14 (LR) 0x0000000 R15 (PC) 0x00000014	This is the situation after the ADD r1 , r1, r2 has been executed. Register r1 initially contains 2 and r2 contains 3. After the addition r1 contains 2+3 = 5	2
		Simulatio

Note that you have to click on **Debug** and then **Start/stop Debugging Session** to get out of the debug mode.

V 5.0

USING THE KEIL SIMULATOR: A SECOND EXAMPLE

Let's now look at a more realistic example of the use of the simulator that includes both a loop and an example register indirect addressing. We are going to add together five numbers stored in memory.

```
AREA Pointers, CODE, READONLY
        ENTRY
Start
        ADR
              r0,List
                            ;register r0 points to List
        MOV
              r1,#5
                            ; initialize loop counter in r1 to 5
        MOV
              r2,#0
                            ; clear the sum in r3
              r3, [r0]
r0, r0, #4
r2, r2, r3
r1, r1, #1
              r3,[r0]
                            ; copy the element pointed at by r0 to r3
Loop
        T,DR
                            ; point to the next element in the series
        ADD
        ADD
                            ;add the element to the running total
        SUBS r1, r1, #1
                            ;decrement to the loop counter
                            ;repeat until all elements added
        BNE
              Loop
Endless B
              Endless
                            ; infinite loop
List
        DCD
               3,4,3,6,7
                            ;the data (five 32-bit words)
        END
```

A COMMENT ON PROGRAM LAYOUT

When writing an assembly language program, column one is reserved for a user-defined name that allows us to refer to that line (more specifically, it corresponds to the address of that line in the program when it's been assembled into machine code). In this example, the four labels are Start, Loop, Endless, and List. Actually, Start, is a dummy label in the sense we never refer to it. I added it simply to demonstrate that we can label a line just for the programmer (this indicates the start of the program).

Anywhere after column one, we can write an instruction. The only rule is that there must be at least one space following the mnemonic, and that parameters must be separated by commas. Spaces after a comma are optional; for example, we can write

ADD r1, r2, r3 or ADD r1, r2, r3

Finally, we can append a comment to the right. The assembler we are using requires a semicolon to separate it from the code. Although we don't have to write a program in columns as we've done above, it makes the program easier to read.

The *executable* code consists of three parts. The first part beginning with the label Start sets up the environment. The instruction ADR **r0**, List is a pseudo instruction that loads the 32-bit value of List into register r0. List is a label that refers to the five items of data in memory. What is the value of List? That's something the programmer doesn't have to worry about; the assembler's job is to convert labels into their actual values. However, in this case it is easy. The assembler begins at address zero and each instruction occupies four bytes. The code consists of eight instructions which occupy $8 \times 4 = 32$ bytes. Consequently List refers to location 0x0000020.

The two move instructions initialize a loop counter (we are going to go round five times), and set the initial total to zero.

The body of the code which we've printed in blue performs the actual addition. The LDR **r3**, [r0] instruction loads register r3 with the contents of the memory location pointed at by r0. Since we initialized r0 to point to List, we will first access the

value 3. Then, we increment the pointer in r0 to point to the next word in memory to be added. This lets us step through the sequence of five numbers.

The next step is to add the value of r3 we've just read to the running total in register r2 using ADD r2, r2, r3.

Finally, the instruction SUBS r1, r1, #1 subtracts 1 from the loop counter in r1, which goes from 5 to 4 on the first cycle round the loop. The S on the end of SUB tells the processor to update the condition codes (carry, zero, negative, overflow) at the end of the subtract operation. The next instruction, BNE Loop, tests for the zero condition that we get when the loop count goes from 1 to 0. If the loop count is not 0, the BNE (branch on not zero) forces a jump to the line labeled by Loop and this block of code is executed again. If the loop count is 0, we have finished the loop and fall through to the next instruction.

There's nothing for us left to do, so we "jam" the computer by inserting the B Endless instruction. This is an unconditional branch (jump) to the line labeled by Endless. Because a jump is made to this line, the operation is repeated endlessly. This is a classic way of stopping a simulation.

Following the executable code, the assembler directive DCD (define constant data) allows you to preload data into memory before the program runs. In this case, the values 1, 4, 3, 6, and 7 are each loaded into memory as a 32-bit value.

The following snapshot demonstrates the state of the program after it has been loaded the **project Build target** function used to perform the assembly.

🔣 E:\C	engageBoo	k\Genga	geWorki	Book\Se	condExample.	.uvproj -	μVisio	n4		_ 🗆 ×	
<u>F</u> ile	<u>E</u> dit <u>V</u> iew	<u>P</u> roject	Fl <u>a</u> sh	<u>D</u> ebug	Pe <u>r</u> ipherals	Tools	<u>s</u> vcs	<u>W</u> indow	<u>H</u> elp		
1 🗋 🛛	j 🔒 🧊	3 🖬 (2 9	6	P 12	四限		i //# //#	2		
1 🔮 🕻	🖺 🔛 🧼 🗄		Target 1	L	- 🕺	(† 🛃 🗄	3				1
	SecondExa	mple.asm						Click Start/	on the Del Stop Debu	g Session comma	nd, and
01 02		AREA F	ointe	rs, CO	DE, READON	NLY		then c	lick on it. window.	This will display	the
03 04 05 06	Start	ENTRY ADR MOV MOV	r0,Li: r1,#5 r2 #0	st	;regista ;initial	er r0 j lize l	point oop c	s to Li counter	st r1 to 5		
07 08 09	Loop	LDR ADD ADD	r3, [r0 r0, r0, r2, r2	0] ,#4 ,r3	;copy el ;point t ;add the	lement to nex e elem	poin t ele ent t	ted at i ment in to the r	by r0 t series unning	o r3 total	
10 11 12	Endless	SUBS BNE B	r1,r1, Loop Endles	,#1 ss	;decreme; ;repeat ;infinit	ent to until te looj	the all p	loop co element	unter s added		
13 14 15	List	END	3,4,3	,6,7	;the dat	ta (fi	ve 32	-bit wo	rds)		
	<u></u>										

The next snapshot shows the result of entering the debug mode. For clarity, we have removed some of the windows that are not needed.

				Some	of the		D'	1
		110 10		debug	gging		Disassemble	d
E:\CengageBool	k\GengageWorkBo	ok\SecondEx	ample.uvproj -	Comm	nands.		code window	v. <u>uux</u>
<u>File Edit V</u> iew	Project Flash	<u>D</u> ebug Pe <u>r</u> ipl	herals <u>T</u> ools	<u>s</u> vcs <u>v</u>				
	る陶竈(2)		門際開設	連連 //編 //編	<u> </u>	• 🔊 🖉	° 🔍 🗕 🖓 🔗	🚓 🖃 🔸
	} () +() +() +() +()		💻 🦾 🚑 🕶 🛙	🛛 - 🛛 - 🛛	- 🔝 -	🛠 • 😑		
Registers	џ)	Disassembly	/					₽ ×
Register	Value	4	:Start /	ADR r0,List	;re	gister r0 points t	to List	
Current	-		0000 E28F0	01C ADD	RO, PC,	#0x0000001C		
R0	0x00000000	0*0000	: 0004 E2701	MOV r1,#	5 D1 #0w	;initialize loop	counter r1 to	5
R1	0x00000000	6		MOV r2.#	0	clear the sum it	N 73	
R2	0x00000000	0x0000	0008 E3A02	2000 MOV	R2.#0x	00000000		
R3	0x00000000	7	: Loop I	DR r3,[r0]	; co	py element pointed	i at by r0 to 1	r3
R4	0x00000000	0x0000	000C E5903	3000 LDR	R3,[R0]		
R6	0x00000000	8	: 4	ADD r0,r0,#4	;po	int to next elemer	nt in se <mark>l Instr</mark>	uction generated by the
R7	0x00000000	0x0000	0010 E2800	0004 ADD	R0,R0,	#0x0000004	pseu	do operation
R8	0x00000000	9	: 1	ADD r2,r2,r3	;ad	d the element to t	the runn ADR	r0. List.
R9	0x00000000	0x0000	0014 E0822	2003 ADD	R2,R2,	K3		
R10	0x00000000	010000	: 0019 82511	SUBS TI,T	1,#1 D1 D1	;decrement to	o the Id	
R11	0x00000000	11	·	BNE Loop	KI,KI,	<pre>:repeat until all</pre>	l elements adde	be
R12	0x00000000	0x0000	001C 1AFFE	FFA BNE	0x0000	000C		
R13 (SP)	0x00000000	12	: Endless H	B Endless	;in	finite loop		
R 14 (LR)	0x00000000	0x0000	0020 EAFFE	FFFE B	0x0000	0020		The constants
E CPSR	0x000000003	0x0000	0024 00000	003 ANDEQ	R0,R0,	R3		created by DCD.
	0x00000000	0x0000	0028 00000	0004 ANDEQ	R0,R0,	R4		created by 202.
		0x0000	002C 00000	003 ANDEQ	RO,RO,	R3		
E Fast Interrupt		0x0000	0030 00000	0006 ANDEQ	RO,RO,	R6		
+ Interrupt		0x0000	0034 00000	1007 ANDEQ	R0, R0,	<u>K7</u>		
Abort		Se	condExample.as	m				▼ ×
		01						
PC \$	0x0000000	02	AREA	Pointers, COD	E. READO	NLY		
Mode	Supervisor	03	ENTRY					
States	0		art ADR	r0,List	;regist	er r0 points to Li	ist	
Sec	0.00000000	05	MOV	r1,# 5	;initia	lize loop counter	r1 to 5	
		06	MOV	r2,# 0	;clear	the sum in r3		
		07 Lo	op LDR	r3,[r0]	;copy e	lement pointed at	by r0 to r3	
		08	ADD	r0,r0,#4	;point	to next element in	1 series	
		10	SUBS	r1, r1, #1	; decrem	ent to the loop of	ounter	
		11	BNE	Loop	;repeat	until all element	ts added	
		12 En	dless B	Endless	;infini	te loop		
		13 Li	st DCD	3,4,3,6,7	;the da	ta (five 32-bit wo	ords)	
L		14	END					_ _
🔃 Project 🛛 🧱 Reg	isters							
Command								† ×
*** Error: 'C	:\Keil\ARM\BI	N\DARMO.DI	LL' not fou	nd				
Running with	Code Size Lim	uit: 32K	1110					
Load "E:\\Cen	gageBook\\Gen	lgageWorkBo	ok/\Second	Example.AXF"				
*** Restricte	d Version wit	h 32768 B	te Code Si	ze Limit				-1
*** Currently	used: 56 Bvt	es (0%)	for code of	DC DIMIO				-
<u> </u>		- , ,						F
>								
ASSIGN BreakD	isable BreakE	nable Brea	kKill Brea	kList BreakSet	: BreakAc	cess COVERAGE DEF	INE DIR Displa	y Enter
							Simulat	ion
							Sinitiat	

There are two notable features. First, the pseudo operation ADR r0, List has been translated into the actual instruction ADD r0, PC, #0x000001C. A pseudo operation uses existing instruction to create an operation that loads a 32-bit value into a register. The assembler might generate different code on different occasions.

A second interesting feature also appears in the disassembly window. You can easily recognize the program in this window. However, the data values stored by the assembler in memory (i.e., 5, 5, 6, 7) are read by the disassembler and translated into instructions. In this case the first value, 5, corresponds to the code for ANDEQ r0, r0, r3). Of course, this code is nonsense. However, the disassembler does not know this; it just translates anything in memory into an instruction. We are now going to create a memory window. To do this, click on **View** then select **Memory Window** and then **Memory 1**. Enter the starting address in the **Address** box in the memory window (which is 0) and hit return. This produces the display as shown below. In the memory window we can see both the code and data. Note that I have resized the memory window (it may have a different number of bytes per line on your system). You can drag the edge of the memory window to display as many or as few bytes per line as you require.

	1.4.1					
tes 0 X Dissembly						
	0 ×					
ter Vela 4: Start ADR r0,List /register r0 points to List						
Current contraction contractio						
- RD BACCOCCC SI HOV FI, FS JINITIALIZE LOOP COUNTER FI to S						
R2 B-05000000 Select 'View' tab then felser the sum in r3						
R3 0.0000000 memory then memory 1 to copy element pointed at by r0 to r3						
R0]						
- R6 0-00000000 Select memory window. point to next element in series						
-R8 0-0000000 9: ADD r2,r2,r3 radd the element to the running total						
- R9 0x0000000 101 E0822003 ADD B2,R2,R3						
H10 0x0000000 0x000000018 E2511001 SUBS R1,R1,#0x00000001						
R12 0x0000000 11: BNE Loop :repeat until all elements added						
R13(SP) 040000000 12: Endless B Endless ; infinite loop						
- R15(PC) 0x00000000 EAFFFFFE B 0x00000020						
CPSR 0x0000003 0x0000025 00000004 ANDEQ R0,R0,R4						
User/System 0x000002C 00000003 ANDEQ R0,R0,R3						
Fast Interrupt 0x00000034 00000007 AMELEQ R0, R0, R0	-1					
Supervisor	2					
Rot SecondExample.asm	• ×					
Internal 01	E					
PC 5 0x000000 02 AREA Pointers, CODE, READONLY	-					
- Sales 0						
Sec 00000000 05 MOV r1.45 vinitialize loop counter r1 to 8						
07 Loop LDR r3,[r0] roupy element pointed at by r0 to r3						
08 ADD r0,r0,#4 /point to next element in series						
10 SUBS r1,r1,\$1 /decrement to the loop counter						
11 BNE Loop repeat intil all elements added						
13 List DCL 3,4,3,6,7 (the data (five 32-bit words)						
14 ENE	-					
	and the second sec					
rajier an weginers 1	-					
	4 ×					
and integrated [12]	1 1 1					
and mixegneen [15]] sand ning with Code Size Limit: 32K i "E:\\CengageBook\\GeogageWorkBook\\SecondExample.AXF*	• ×					
and integrated [12]	• *					
and integrated [11]	• ×					
and integrated [13]	* * *					
Anne (meregeneen) (151) wand ning with Code Size Limit: 32K 4 "E:\\CengageBook\\GeogageWorkBook\\SecondExample.AXF* Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (0%) IGU BreakDisable BreakEndle BreakEntl BreakList BreakSet BreakAppeas COVERAGE DEFINE DIR Display Ent	***					
Ander	* ×					
And	******					
And	******			×		
Anne and and a size limit: 32M of "E:\\CengageBook\\GeograpeBook\\SecondExample.AXF" Restricted Version with 32766 Byte Code Size Limit Currently used: 56 Bytes (0%) IGN BreakDisable BreakEnable BreakEill BreakList BreakSet BreakAccess COVERAGE DEFINE DIR Display Ent Semulation	***			×		
and ming with Code Size Limit: 32K 4 "E:\\CengageBook\\GeogageWorkBook\\SecondExample.AXK* Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (0%) IGW BreakDisable BreakEnable BreakEill BreakList BreakSet BreakAccess COVERAGE DEFINE DIR Display Ent Semulation Semulation	*****			×		
and and ning with Code Size Limit: 32M 4 "E:\\CengageBook\\GeogageWorkBook\\SecondExample.AXF* Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (0%) IGN BreakDisable BreakEnable BreakEill BreakList BreakSet BreakAccess COVERAGE DEFINE DIR Display Ent Semulation Semulation	**			×		
and and ining with Code Size Limit: 32F 4 "E:\\CengageBook\\GeogageWorkBook\\SecondExample.AXF* Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (0%) IGN BreakDisable BreakEnable BreakEill BreakList BreakSet BreakAppeas COVERAGE DEFINE DIR Display Ent Benutation Benutati	**************************************	0 82	20 03	×		
Number Init: Number sand ning with Code Size Limit: 32F 1 "S:\\CengageBook\\SecondExample.AXF" Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (0%) IGN BreakDisable BreakEnable BreakEill BreakList BreakSet BreakAppeas COVERAGE DEFINE DIR Display Ent Semory 1 ddress: 00000000 000000000: E2 8F 00 1C E3 A0 10 05 E3 A0 20 00 E5 90 30 00 60 00	0 04 E	0 82	20 03 00 03	×		
Interview IIII mand ning with Code Size Limit: 32% 1 "E:\\CengageBook\\GengageWorkBook\\SecondExample.AXF* Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (0%) IGW BreakDisable BreakEnable BreakEill BreakList BreakSet BreakAppease COVERAGE DEFINE DIR Display Ent Semory 1 ddress: 00000000 E2 8F 00 1C E3 A0 10 05 E3 A0 20 00 E5 90 30 00 E2 80 0 000000001 E2 E31 10 01 1A FF FF FA EA FF FF FE 00 00 00 30 00 00 00 00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00	0 04 E 0 04 O	0 82 0 00 0 00	20 03 00 03 00 00	×	emory	win
And and ning with Code Size Limit: 32% 4 "E:\\CengageBook\\GeogageWorkBook\\SecondExample.AXF* Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (0%) IGH BreakDisable BreakEnable BreakEill BreakList BreakBet BreakAppeas COVERAGE DEFINE DIR Display Ent Browney 1 Second 2 000000000 E E2 &F 00 1C E3 A0 10 05 E3 A0 20 00 E5 90 30 00 E2 80 0 000000018: E2 31 10 01 1A FF FF FA EA FF FF FE 00 00 00 30 00 00 00000030: 00 00 00 00 00 00 00 00 00 00 00 00 0		0 82 0 00 0 00 0 00	20 03 00 03 00 00 00 00	×	emory	wine
Interview IIII mand ning with Code Size Limit: 32% 1 "E:\\CengageBook\\GengageWorkBook\\SecondExample.AXF* Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (0%) IGW BreakDisable BreakEnable BreakEill BreakList BreakIet BreakAppease COVERAGE DEFINE DIR Display Ent Semory 1 ddress: 00000000 E2 8F 00 1C E3 A0 10 05 E3 A0 20 00 E5 90 30 00 E2 80 0 000000001 E2 8F 00 1C E3 A0 10 05 E3 A0 20 00 00 00 00 00 00 00 00 00 00 000000030: 00 00 00 06 00 00 07 00 00 00 00 00 00 00 00 00 00		0 82 0 00 0 00 0 00 0 00	20 03 00 03 00 00 00 00 00 00	×	emory	wina
Internation IIII mand ning with Code Size Limit: 32% 1 "E:\\CengageBook\\GengageWorkBook\\SecondExample.AXF* Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (0%) IGW BreakDisable BreakEnable BreakEill BreakList BreakIet BreakAccess COVERAGE DEFINE DIR Display Ent Semory 1 ddress: 00000000 E2 8F 00 1C E3 A0 10 05 E3 A0 20 00 E5 90 30 00 E2 80 0 000000001 E2 8F 00 1C E3 A0 10 05 E3 A0 20 00 00 00 00 00 00 00 00 00 000000030: 00 00 00 06 00 00 00 07 00 00 00 00 00 00 00 00 00		0 82 0 00 0 00 0 00 0 00 0 00 0 00	20 03 00 03 00 00 00 00 00 00 00 00	M	emory	wind
<pre>move</pre>		0 82 0 00 0 00 0 00 0 00 0 00 0 00 0 00	20 03 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00	M	emory	wind
<pre>mand into code Size Limit: 32% i "E:\\CengageBook\\GengageWorkBook\\SecondExample.AXF* Restricted Version with 32766 Byte Code Size Limit Currently used: 56 Bytes (0%) IGU BreakDisable BreakEnable BreakEill BreakList BreakBet BreakBooese COVERAGE DEFINE DIE Display Ent</pre>		0 82 0 00 0 00 0 00 0 00 0 00 0 00 0 00	20 03 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00		emory	wind
<pre>mining with Code Size Limit: 32# ining with Code Size Limit: 32# i "E:\\CengageBook\\GengageWorkBook\\SecondExample.AXF" Sestricted Version with 32766 Byte Code Size Limit Currently used: 56 Bytes (0%)</pre>		0 82 0 00 0 00 0 00 0 00 0 00 0 00 0 00	20 03 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00		emory	wind
Internation IIII axad and and gith Code Size Limit: 32K d "5:\\CengageBook\\GengageNorkBook\\SecondExample.AXF* Bestricted Version with 32766 Byte Code Size Limit Currently used: 56 Bytes (0%) IGB BreakDissble BreakEnable BreakEill BreakList BreakAccess COVERAGE DEFINE DIE Display Ent Second 00000000: E2 SF 00 1C E3 A0 10 05 E3 A0 20 00 E5 90 30 00 E2 80 0 00000001: E2 SF 00 1C E3 A0 10 05 E3 A0 20 00 00 00 00 00 00 00 00 00000001: E2 SF 00 0 C E3 A0 10 05 E3 A0 20 00 0 E5 90 30 00 E2 80 0 00000018: E2 SI 10 01 1A FF FF FA EA FF FF FE 00 00 00 00 00 00 00 00 00 00000048: 00 00 00 00 00 00 00 00 00 00 00 00 00		0 82 0 00 0 00 0 00 0 00 0 00 0 00 0 00	20 03 00 03 00 00 00		emory	wind
Interception Intil arang ning with Code Size Limit 32% d "S:\\CengageBook\\GengageWorkBook\\SecondExample.AXF* Restricted Version with 32760 Byte Code Size Limit Currently used: 56 Bytes (0%) IGN BreakDiseble BreakEnable BreakEill BreakList BreakIde BreakBocess COVERACE DEFINE DIR Display Ent Second Control (1) 00000000: E2 BF 00 1C E3 A0 10 05 E3 A0 20 00 E5 90 30 00 E2 80 0 000000018: E2 1 10 01 1A FF FF FA EA FF FF FE 00 00 00 00 00 00 00 00 00000003: E2 1 10 01 1A FF FF FA EA FF FF FE 00 00 00 00 00 00 00 00 00 00 00 00 00	0 04 E 0 04 E 0 04 C 0 00 0 0 00 0 0 00 0 0 00 0 0 00 0 0 00 0	0 82 0 00 0 00 0 00 0 00 0 00 0 00 0 00	20 03 00 03 00 00 00	M s byte	emory	wine 0000
Image with Code Size Limit: 32K d "E:\\CengageBook\GengageWorkBook\SecondExample.AXF" Restricted Version with 32768 Byte Code Size Limit Currently used: 56 Bytes (04) EGN BreakDisable BreakEmable BreakEill BreakList BreakAccess COVERAGE DEFINE DIR Display Ent Sendetion mory 1 ddress: 00000000 E2 8F 00 1C E3 A0 10 05 E3 A0 20 00 E5 90 30 00 E2 80 0 000000001 E2 81 10 01 1A FF FF FA EA FF FF E 00 00 00 03 00 00 00 000000048: 00 00 00 00 00 00 00 00 00 00 00 00 00	0 04 E 0 04 E 0 04 0 0 00 0 0 00 0 0 00 0 0 00 0 0 00 0 0 00 0	0 82 0 00 0 00 0 00 0 00 0 00 0 00 0 00	20 03 00 03 00 00 00 000 00 00 00	M s byte te is at	emory is 0x00	wine 0000

Let's now run through a debug session with this program. The snapshot below shows the screen after we have executed the first three instructions. You can see that r0 is loaded with 0x24 (the start of the data area), r1 contains 5, and r2 contains 0 (note, we have to clear r2 to 0 in the code because, in a real system, r2 will probably not contain 0 at the start of a block of code). Failure to initialize registers is proably the most common error that students make when writing assembly language programms.

The next instruction to be executed is highlighted in both the program and disassembly windows.

🔣 E:	\Cenga	geBool	k\GengageWorkE	look\Sec	ondExample	e.uvproj -	µVision4	
<u>F</u> ile	<u>E</u> dit	<u>V</u> iew	<u>P</u> roject Fl <u>a</u> sh	<u>D</u> ebug	Pe <u>r</u> ipherals	Tools	<u>SVCS</u> <u>W</u> indow	<u>H</u> elp
	💕 🖵		み 山田 タ		-> la l	自己	# # //# //#	🛛 🕐 🔍 💌 🔜 🖑 🔍 🔶 🖉 💼
RST	E.	₩ €	} {} {} {} {} {} {} {} {} {} {} {} {} {}	> 2	à 🖪 🗏 🤅	à 💹 • [🔲 • 🛃 • 🔜 •	· 💷 🕶 🕎 🕶 🖳
Regis	sters		д×	Disasse	mbly			т х
Reg	ister		Value		4: Sta	rt AD	R r0,List	;register r0 points to List
	Current			0x0	0000000	E28F00	1C ADD	R0, PC, #0x000001C
	R0		0x00000024		5:		MOV r1,	5 ; initialize loop counter r1 to 5
	R1		0x00000005	0x0	0000004	E3A010	05 MOV	R1,#0x0000005
	R2		0x00000000		6:		MOV r2,	0 ; clear the sum in r3
	R3		0x00000000	0x0	0000008	E3A020	00 MOV	R2,#0x0000000
	R4		0x00000000		7: Loo	p LD	R r3,[r0]	; copy element pointed at by r0 to r3
	R5		0x00000000		000000C	E59030	00 LDR	R3,[R0]
	R6		0x00000000		8:	AD.	D r0,r0,#4	; point to next element in series
	R7		0x00000000	UX0	0000010	E28000	04 ADD	RU, RU, #UXUUUUUU4
	R8		0x00000000	00	9:	AD.	D 12,12,13	add the element to the running total
	R9		0x00000000	UX0	10.	E08220	US ADD	KZ, KZ, KJ
	R10		0x00000000	0.00	10:	F25110	OI CUBS FI,I	P1 P1 #0:00000001
	R11		0x00000000	0.00	11.	E20110	DI SUBS	RI,RI,#0X0000000
	R12		0x00000000	0.00	0000010	175555	ENE LOOF	owooooooc
	R13	(SP)	0x00000000		12. End	IAFFFF	FA DNE Fndless	infinite loop
	R14	(LR)	0x00000000	0x0	0000020	TATTA	FF B	0x00000020
	R15	(PC)	0x0000000C		0000020	DALLE		•
	+ CPS	R	0x000000D3					
 	± SPS User/Sv	K stem	0×0000000		SecondExa	mple.asm		▼ ×
	Fast Inte	mupt		01				
±	Interrupt			02		AREA P	ointers, COD	E, READONLY
	Superv	isor		03		ENTRY		
	Abort			04	Start	ADR :	r0,List	;register r0 points to List
÷	Undefine	ed		05		MOV	r1,# 5	;initialize loop counter r1 to 5
<u> </u> ⊡	Internal			06		MOV	r2,# 0	;clear the sum in r3
	PC	s	0x000000C	->07	Loop	LDR :	r3,[r0]	;copy element pointed at by r0 to r3
	Mod	e	Supervisor	08		ADD :	r0,r0,# 4	;point to next element in series
	State	es	3	09		ADD :	r2,r2,r3	;add the element to the running total
	Sec		0.00000000	10		SUBS	r1,r1,# 1	;decrement to the loop counter
				11		BNE	Loop	;repeat until all elements added
				➡12	Endless	B 1	Endless	;infinite loop
				13	List	DCD :	3,4,3,6,7	;the data (five 32-bit words)
				14		END		
💷 P	roject	🗮 Reg	isters					
								Simulation

The next snapshot shows the state of the simulator after we have nearly completed one trip round the loop and are at the last instruction, the branch to Loop on not zero. The value of r0 is 28 (i.e., 24 + 4) because we are pointing at the next data item. The value of r1 (the loop counter) is 4 because we've decremented it on this trip. The value of r3 is 3 because we've loaded the first number, and the value of r2 is three because the sum contains only one number so far.

The final snapshot for this example just shows some of the registers and code. Rregister r2 now holds the sum of the five numbers in memory. The value of r0 contains 0x38 which the next location after the five numbers $(24 + 5 \times 4 = 38 \text{ using hexadecimal arithmetic})$.

Version 1 WORKBOOK FOR COMPUTER ORGANIZATION AND ARCHITECTURE: THEME AND VARIATIONS

🔣 E:\CengageBook	¢\GengageWork₿	ook\Sec	ondExample	e.uvproj -	µVision4						
<u>F</u> ile <u>E</u> dit <u>V</u> iew	<u>P</u> roject Fl <u>a</u> sh	<u>D</u> ebug	Pe <u>r</u> ipherals	<u>T</u> ools	<u>SVCS</u> <u>W</u> indow	<u>H</u> elp					
I 🗋 🎯 📮 🗿 🗌	と時間の		in the state	97 PT 2		🔊 🔹 🔊 🔕 🧉	0 🔗 🚓 🔚				
				· ··· · · · · · · · · · · · · · · · ·							
🛛 RST 🖽 🕥 🕅	0 0 0 0		K 16 📰 🖗								
Registers	ф ×	Disasse	mbly				<u> </u>				
Register	Value		4: Sta	rt AD	R r0,List	;register r0 points to List	_				
		0x0	0000000	E28F00	1C ADD	R0,PC,#0x0000001C					
R0	0x00000028	0	5:	E23010	MOV FI,	5 ;initialize loop counter ri P1 #0w00000005	to 5				
<mark>R1</mark>	0x00000004	UXU	6.	LJAUIU	NOV	RI,#0X00000005					
R2	0x0000003	0.00	0.00000	E27020	MOV 12,	D ;Clear the sum in rs					
R3	0x0000003	U.X.	7. 100	ESAUZU	00 MOV D m2 [m0]	K2,#0X0000000	- - 2				
R4	0x00000000	0.00	000000C	БЕ0030 h TD	A IS,[IU]	;copy element pointed at by ro to	0 13				
R5	0x00000000			E39030	D 20 20 #	KS, [KU]					
R6	0x00000000	0.00	0.000010	E29000	04 ADD	point to next element in series					
R7	0x0000000		0000010	E20000	04 ADD D x2 x2 x	add the element to the running	total				
R8	0x0000000	0.00	2.	E08220	03 ADD		LULAI				
R9	0x0000000	U.A.	10.	200220	SIIBS v1	1 #1 decrement to the loop of	ounter				
R10	0x00000000	0.20	0000018	F25110	01 SUBS	P1 P1 #0v0000001	Juncer				
R11	0x0000000	0.00	11.	625110	BNE Loc	<pre>repeat until all elements a/</pre>	habb				
R12	0x0000000		0000010	17474	FA BNF	oxooooooc	uucu				
R13 (SP)	0x0000000		12: End	less B	Endless	infinite loop					
R14 (LR)	0x00000000	0x0	0000020	EAFFFF	FE B	0x0000020					
R15 (PC)	0x0000001C										
E CPSR	0x200000D3										
I ± SPSR	0x00000000		SecondExa	mple.asm			▼ ×				
User/System											
Fast Interrupt				70F7 D	ointona CO	E RENDONI V					
		02		ENTRY	oincers, co	L, READONLI					
		03	Start	ADD	m0 Tigt						
Abort		04	Start	MOU	r0,1150	initializa lean counter mi to 5					
		00		MON	n 2 # 0	counter ri to s					
	0.0000010	00	Loop		$r_{2, \pm 0}$	clear the sum in ro					
PC S	Concercione	07	цоор	ADD	$r_0 r_0 \#_4$	propy element pointed at by ro to ro					
Node	Supervisor	00			r_{2} r_{2} r_{3}	add the element to the murping total	,				
States	3	10		SUBS	$r1 r1 \pm 1$	degreement to the loop counter	£				
380	0.0000000	10		BNE	Loon	repeat until all elements added					
		12	Endless	B	Endless	infinite loop					
		13	List	DCD -	3.4.3.6.7	the data (five 32-hit words)					
		14		END	-,-,0,0,7	, one data (LIVE SZ-DIE FOLDS)					
Project Project	isters										
Indents selected tout	left one tab ston						imulation				
indents selected text	dents selected text left one tab stop Simulation										



Let's look another program that uses pointer-based addressing to access memory. The snapshot below illustrates a program that adds together pairs of elements of two vectors X and Y, and puts the result in Z; that is, it performs $z_i = x_i + y_i$ for i = 0 to 3.

🔣 E:\(Cengagel	Book\Gen	gageWorkBo	ok\ThirdExample.uvproj - µVision4							
<u>F</u> ile	<u>E</u> dit <u>V</u> i	ew <u>P</u> roje	ct Fl <u>a</u> sh	<u> 2ebug Per</u> ipherals <u>T</u> ools <u>S</u> VCS <u>W</u> indow <u>H</u> elp							
1 🗋 🛛	i 🖬 🖥	1 % B	a 🗈 🤊	> <> > 🥐 黎 豫 徽 譯 譯 /// /版 🌌	• 🗟 🥐 🍳						
i 🧇 i	🏥 🎬 🍯) 🚉 🔤	Target 1	👻 🌋 📥 🎨 Clear All Bookmarks (Ctrl+Shift+F2)							
] ThirdEx	cample.asn	n	Clear all bookmarks in the current document	▼ ×						
01	01 AREA VectorAddition, CODE, READONLY										
02		ENTRY									
03	Start	MOV	r0, #4	;four elements to add r0 is the counter							
04		ADR	r1,X	;r1 points to X							
05		ADR	r2,Y	;r2 points to Y							
06		ADR	r3,Z	r3 points to Z							
07	Loop	LDR	r4,[r1],	4 ;Repeat get element of X and update pointer							
08		LDR	r5,[r2],	4 ; get element of Y and update pointer							
09		ADD	r4,r4,r5	; add the two lelements							
10		STR	r4,[r3],	4 ; store result in Z and uddate pointer							
11		SUBS	r0,r0,#1	; decrement loop counter							
12		BNE	Loop	;Until all elements done							
13	Me	В	Me	;infinite loop							
14											
15		AREA V	ectorAdd	tion, DATA, READWRITE							
16	х	DCD	1,2,3,4	;data for array X							
17	Y	DCD	3,4,5,6	;data for array Y							
18	Z	DCD	0,0,0,0	;dummy data for array Z							
19		END									

We've defined two data areas: AREA VectorAddition, CODE, READONLY where the program code is located, and AREA VectorAddition, DATA, READWRITE. The parameters CODE and DATA refer to regions of memory that contain code or data, and the parameters READONLY and READWRITE indicate that the region of memory space can only be read (as in the case of the program), or can be both read from or written to (using parameter READWRITE).

Once the program is ready to run, you select **Debug** and **Start/Stop Debug Session** in the normal way. We then have to perform an additional step to indicate that the data memory is writable. Click the **Debug** tab and then the **Memory Map** tab.

The Memory Map below shows the situation with the address range 0x00 to 0x5B defined as both executable and readable memory. We need to define locations 0x38 to 0x5B as writable locations. To do this, enter the values in the **Map Range box** and tick **<u>Read</u>** and <u>**Write**</u>.



The final three snapshots for this example show, in order, the initial memory map, the state of the system during execution, and the final memory map.

The initial memory map shows the code from 0x00000000 to 0x0000001B, and the data area starting at 0x0000002C. The snapshot is taken at the end of the first cycle of iteration. The three pointer registers are loaded with addresses that are four bytes greater than the start of the three vectors, because auto-incrementing is used and the pointer is increased after it has been used. The final memory map shows the source data in read and the data written back to memory in green.

Memory 1																							x
Address: 000000	000																						
0x00000000:	E3	AO	00	04	E2	8F	10	20	E2	8F	20	2C	E2	8F	30	38	E4	91	40	04	E4	92	
0x00000016:	50	04	E0	84	40	05	E4	83	40	04	E2	50	00	01	1A	FF	FF	F9	EA	FF	FF	FE	
0x0000002C:	00	00	00	01	00	00	00	02	00	00	00	03	00	00	00	04	00	00	00	03	00	00	
0x00000042:	00	04	00	00	00	05	00	00	00	06	00	00	00	00	00	00	00	00	00	00	00	00	
0x0000058:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0x000006E:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0x0000084:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0x000009A:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	•

🔣 E: \	🖫 E:\CengageBook\GengageWorkBook\ThirdExample.uvproj - µVision4												
<u>F</u> ile	<u>E</u> dit <u>V</u> iew	<u>P</u> roject Fl <u>a</u> sh	Deb	ug Pej	ipherals	Tools	<u>S</u> VCS <u>W</u> indow	Help					
	🞽 🛃 🧭	るも氏の	6	$\langle \dots \rangle$	PB	内内	k ⊈ ≢ //≞ //j	- 🔊 🔍 🔍 🖉 🐨 🔍					
RST	1 🛛 🕹	Ŋ ()→ ()→ *() 4		0. 1	3 📰 🕼	5 🐺 •	🔲 🔹 😒 🕶	• 💷 • 📓 • 🏃 • 🖳					
Regist	ters		φ×) ThirdEx	ample.a	sm	▼ X					
Regi	ster	Value		01		AREA	VectorAdditi	on, CODE, READONLY					
	Current		- 1	02		ENTRY	2						
	R0	0x0000003		03	Start	MOV	r0,# 4	;four elements to add r0 is the counter					
	R1	0x0000030		04		ADR	r1,X	;r1 points to X					
	R2	0x0000040		05		ADR	r2,Y	;r2 points to Y					
	R3	0x0000050		06		ADR	r3,Z	;r3 points to Z					
	R4	0x0000004		07	Loop	LDR	r4,[r1],# 4	;Repeat get element of X and update pointer					
	R5	0x0000003		08		LDR	r5,[r2],# 4	; get element of Y and update pointer					
	R6	0x00000000		09		ADD	r4,r4,r5	; add the two lelements					
	R7	0x00000000		10		STR	r4,[r3],# 4	; store result in Z and uddate pointer					
	R8	0x00000000		11		SUBS	r0,r0,#1	; decrement loop counter					
	R9	0x00000000		 →12		BNE	Loop	;Until all elements done					
	R10	0x00000000		13	Me	в	Me	;infinite loop					
	R11	0x00000000		14									
	····· R12	0x00000000		15		AREA	VectorAdditi	on, DATA, READWRITE					
	······ R13 (SP)	0x00000000		16	х	DCD	1,2,3,4	;data for array X					
	······ R14 (LR)	0x00000000		17	Y	DCD	3,4,5,6	;data for array Y					
	R15 (PC)	0x00000024		18	Z	DCD	0,0,0,0	;dummy data for array Z					
	CPSR	0x200000D3	-	19		END							
🖭 Pr	oject 🛛 🚟 Re	gisters											
								Simulation					

Memory 1																							×
Address: 000000	00																						
0x0000000:	E3	A 0	00	04	E2	8F	10	20	E2	8F	20	2C	E2	8F	30	38	E4	91	40	04	E4	92	
0x00000016:	50	04	E0	84	40	05	E4	83	40	04	E2	50	00	01	1A	FF	FF	F9	EA	FF	FF	FE	
0x0000002C:	00	00	00	01	00	00	00	02	00	00	00	03	00	00	00	04	00	00	00	03	00	00	
0x00000042:	00	04	00	00	00	05	00	00	00	06	00	00	00	04	00	00	00	06	00	00	00	08	
0x0000058:	00	00	00	AO	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0x000006E:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0x0000084:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0x0000009A:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	•

V 5.0

PROBLEM SET 4

- 1. What is an assembler?
- 2. What is a cross-assembler?
- 3. What is a (CPU) simulator?
- 4. Does a simulator run as fast as the *native* or *target* processor being simulated? For example, does ARM processor code being executed by a simulated ARM processor on a PC run faster or slower than the same code being run on a real ARM processor?
- 5. What is the wrong with ADD **r0**, r16, r1?
- 6. Is there anything wrong with ADD r0, r0, r0?
- 7. Why is ADD **r1**, #5, r2 wrong?
- 8. What is the difference between a **syntax** error and a **semantic** error?
- 9. What is the difference between an instruction and an assembler directive?
- 10. What is the effect of ADR **r0**, 1234?
- 11. ADR **r0**, #1234 is known as a pseudo instruction. What is a pseudo instruction and what is its purpose?
- 12. What's wrong, if anything, with ADD **r15**, r2, r3?

PROBLEM SET 5

Here we provide an introduction to the Keil ARM processor development system.

1. Write a simple program to perform: $Z = A + B + C - (D \times E)$

The instructions you may use are ADD, SUB, and MUL. Assume that the data is in registers r0 to r4 (representing A to E) and the result is put in r5.

Enter your program into the Keil simulator and run it. You can use move instruction to load data into registers. Do you get the expected answer?

- 2. Now assume A, B, C, D and E are 16-bit values in memory. You can load them by using a DCD directive. Remember that you use a label to define the first memory location and you can put successive values on the same line by separating them by commas. However, since each data item needs its own name, you are going to have to use one directive per element; that is:,
 - A DCD 4 B DCD 12
 - C DCD -2

Enter the program, compile (build) it and test it.

3. Write a program that includes deliberate syntax errors. Enter it in the development system, assemble (build) it and then debug it.

POINTER-BASED ADDRESSING MODES

We have already introduced addressing modes. Here we discuss the ARM processor's register indirect addressing mode that supports several variations. This is an important topic because it's essential to the efficient manipulation of data structures such as tables, arrays, matrixes, and vectors.



Let's first review the basic concept of register indirect, or *pointer-based*, addressing. Indirect addressing specifies a *pointer* to the actual operand which is invariably in a register. For example, the instruction, LDR r0, [r1], first reads the contents of register r1 to obtain the pointer that gives you the address of the actual operand in memory. It then reads the memory location specified by the pointer in r1 to get the required data. This addressing mode requires *three* memory accesses; the first is to read the instruction to identify the register containing the pointer, the second is to read the contents of the register to get the pointer. And the third is to get the desired operand at the location specified by the pointer.

You can easily see why this addressing mode is called **indirect** because the address register specifies the operand indirectly by telling you *where* it is, rather than *what* it is. This is the only form of addressing that the ARM processor can use to access memory. The box below describes three variations on this addressing mode and gives their assembly language forms, defines the addressing mode (using RTL), and gives them names. The naming of addressing modes is not always consistent in computer science and manufacturers sometimes use different names for the same addressing mode.

1.	LDR	r1, [r0]	r0 points at the operand
			[r1] ← [[r0]]
			Base register addressing
2.	LDR	r1 ,[r0,#4]	The operand is 4 bytes on from the location pointed at by r0
			$[r1] \leftarrow [[r0 + 4]]$
			Pre-indexed addressing
3.	LDR	r1 ,[r0,#4]!	The operand is 4 bytes on from the location pointed at by r0. After loading
			the operand, the pointer register is incremented by 4
			$[r1] \leftarrow [[r0 + 4]]$
			$[r0] \leftarrow [r0] + 4$
			Pre-indexed addressing with writeback
			Autoincrementing preindexed addressing
4.	LDR	r1, [r0],#4	The operand is pointed at by r0. After making the access, r0 is updated by 4.
			[r1] ← [[r0]]
			$[r0] \leftarrow [r0] + 4$
			Post-indexed addressing

The following figure describes the basic register indirect (sometimes called indexed or base addressing). The instruction specifies an address register and that register points at the actual location of the data in memory.



PLAYING WITH POINTERS

In principle, you don't need any addressing mode other than the simple register indirect [r0]. In practice, computer design is very much about the tradeoff between computational efficiency, complexity, and cost. Most computers provide variations on the basic register indirect addressing mode in order to reduce the size of the code and speed up its execution. In the 1980s, this was taken to extremes by the 68020 microprocessor that had truly complex addressing modes that could perform amazing operations with a single instruction. However, such addressing modes were so complex that compilers could not handle them optimally, and they took up a big chunk of the silicon chip. They were, at best, used infrequently. And they were slow.

Consider the operation LDR r1, [r0, #8]. This is only a slight modification of ARM's plain vanilla register indirect addressing. The difference is the literal within the square brackets. The address of the operand is found at [r0] + 8; that is, the operand is 8 bytes on from the location pointer at by r0. This addressing mode is sometimes called **pre-indexed addressing**. Executing LDR r1, [r0, #8] Pointer register r0 n The operand loaded into r1 is 8 bytes from the location pointed at by r0.

The offset, in this case 8, is not added to the contents of the pointer in the register. The contents of r0 are fed to an adder, the offset oddad and the regult used to access memory. The

added, and the result used to access memory. The contents of register r0 do not change.

A typical application of pre-indexed addressing is in accessing a table. Consider a table in memory containing 12 entries corresponding to January to December. Register r0 points at the start of the table (i.e., January). The following operations have the effect:

LDR r1, [r0]	;Load r1 with the January data
LDR r2, [r0,#8]	;Load r2 with the March data
LDR r3, [r0, #28]	;Load r3 with the August data
STR r4,[r0,#44]	;Store the data in r4 in August's location



Why are the offsets 12, 32, and 44? The wordlength of an ARM processor is 32 bits or 4 bytes. If r0 points at January, the data occupies locations [r0], [r0] + 4, [r0] + 8, [r0] + 12, etc. For example, February is the second month and its data is at [r0] + 4.

The offset for the first month is 0, and the offset of month *i* is (i - 1 * 4); e.g., May is month 5 and its offset is (5 - 1) * 4 = 16.

In practice, programmers rarely used literal numeric offsets. The EQU (equate) directive assembler directive allows you to replace any number by a name; for example,

Hastings EQU 1066

This assembler directive causes the assembler to substitute 1066 for Hastings whenever it sees it. It doesn't matter whether you write ADR **r0**, Hastings or ADR **r0**, 1066, it has the same effect. The example below demonstrates how we can use assembler directives with pre-indexed addressing to access an array of days, add two values together, and store the result. The memory data shows that the final value (4 + 7 = 11 = 0x0B) has been correctly stored.

🔣 E:\CengageBoo	ok\GengageWorkBool	k\Assei	nblerDire	ctiveEQ	U.uvproj - µVision4		
<u>F</u> ile <u>E</u> dit <u>V</u> iew	<u>P</u> roject Fl <u>a</u> sh <u>D</u> e	bug F	e <u>r</u> ipherals	<u>T</u> ools	<u>S</u> VCS <u>W</u> indow <u>H</u> elp	2	
i 🗋 💕 🖬 🥥	る 単路 タウ	<	> la l	2月1	👷 💷 //# //# 🖄	•	• 🔜 🖉 鱼 o 🔗 🍓 🔳 🔌
👫 🗐 🛞 🕇	ት 🖓 🖓 🖓	> 🔍	i 🔁 📰 🌡	à 🚚 •	- 🔜 - 🔜 -	• 🖪 • 🔆 • 📃 👘	
Registers	ą ×		EQUte	st.asm			▼ ×
Register	Value 🔺		_	AREA	EOUtest, CODE, R	EADONLY	
E Current		02		ENTR	Y		
R0	0x0000004	03	Mon	EQU	0	;Monday offset	
R1	0x00000018	04	Tue	EQU	4	1	There we use Fri as the
R2	0x0000007	05	Wed	EQU	8	; /	offset rether then the literal
R3	0x000000B	06	Thu	EQU	12	; /	onset famer man me meral
R4	0x00000000	07	Fri	EQU	16	;	value16 (i.e., $(5-1) \times 4$)
R5	0x0000000	08	Sat	EQU	20	; /	
R6	0x0000000	09	Sun	EQU	24	;Sunday offset	
R7	0x0000000	10	l .				
R8	0x0000000	11		ADR	r1,Week	;r1 points to Weel	k
R9	0x0000000	→12		LDR	r0,[r1,#Mon]	get Monday's data	a
R10	0x0000000	13		LDR	r2,[r1,#Fri]	;get Friday's dat	a
R11	0x0000000	→14		ADD	r3,r0,r2	;add thes values	
R12	0×0000000	15		STR	r3,[r1,#Sun]	;and put the resu	lt in Sunday's slot
······ R13 (SP)	0×0000000	5>16	Me	в	Me	parking loop;	
R14 (LR)	0x00000000	17					
R15 (PC)	0x0000014	18		AREA	EQUTEST, DATA,	READWRITE	
± CPSR	0x00000D3	19	week	DCD	4,6,2,5,7,9,7	;set up some dumm	y data for testing
± SPSR	0x0000000	20				; these values V1.	The result
User/System		21		END		; seven consectui	ve word locations
		22		END			/ in memory
	-	2.3					
🔚 Project 🛛 🚟 Re	gisters	Ī					
Memory 1							д х
Address: 0x18							
x0000018: 0	0 00 00 04 00 0	0 00	06 00	00 00	02 00 00 00 05 00	00 00 07 00 00 00	09 00 00 00 0B 00 00 00 00
x0000038: 0	0 00 00 00 00 0	0 00	00 00	00 00	00 00 00 00 00 00	00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 🔳
							Simulation

Suppose we didn't have pre-indexing. What would we do? We'd have to write something like:

VON	r1, r0	;Save pointer r0 in r1.
ADD	r1, r1, #4	;Create a new temporary pointer in r1
LDR	r2, [r1]	;Read the required data from memory
		; LDR r2 , [r0, #4] Does this in one instruction and doesn't tie up a second register

Simple pre-indexing is useful for accessing elements at a specified offset. However, it does not change the pointer. Sometimes, we are stepping through a data structure and we need to permanently update the pointer each time it's used; for example:

	MOV	r2, #64	;Set up loop count for 64 elements
	MOV	r3, #0	;Clear the sum
	ADR	<pre>r0,Table</pre>	;Point to the table of data elements to be summed
Next	LDR	r1, [r0]	;Repeat: Read an element
	ADD	r0, r0 , #4	; Update the pointer
	ADD	r3, r3,r1	; Add a new element to the total
	SUBS	r2, r2,#1	; Decrement loop counter
	BNE	Next	;Repeat until all done

There's nothing new here. After accessing an element we update the pointer ready for the next cycle of iteration (in blue). Fortunately, ARM processors provide a **post-indexed** addressing mode. The offset is provided after the pointer, as the example demonstrates.

LDR **r1**, [r0], #4

In this case, the operand is accessed at the address pointer at by r0, and then r0 is incremented by 4. This addressing mode saves an instruction without incurring a time penalty. We can now write.

	MOV	r2, #64	;Set up loop count for 64 elements
	MOV	r3, #0	;Clear the sum
	ADR	<pre>r0,Table</pre>	;Point to the table of data elements to be summed
Next	LDR	r1 , [r0],#4	;Repeat: Read an element and update the pointer
	ADD	r3, r3,r1	; Add a new element to the total
	SUBS	r2, r2, #1	; Decrement loop counter
	BNE	Next	;Repeat until all done

OVERVIEW OF THE ARM PROCESSOR'S INSTRUCTIONS

We now look at the type of operations ARM processors can carry out. In general, all computer instructions fall into a small number of groups. The main groups are:

- Data movement: These are all the operations that move data from one place to another and often account for about 70% of all the instructions in a program.
- Data processing: These are instructions that operate on data; that is, change its value. This group is often subdivided into arithmetic operations, logical operations (also called Boolean or bitwise), and shift operations. The following table describes there instructions.

Arithmetic	Arithmetic instructions perform operations on data in numeric form.	
Logical	A logical operation treats data as a string of bits and performs a Boolean operation on these bits; for example 11000111 AND 10101010 yields 10000010.	
Shift	Shift instructions move the bits in a register one or more places left or right; for example shifting 00000111 one place left yields 00001110.	
Bit	A bit instruction acts on an individual bit in a register, rather than the entire contents of a register. Bit instructions allow you to test a single bit in a word (for 1 or 0), to set a bit, to clear a bit to 0, or to flip a bit into its complementary state.	
Compare	Compare instructions compare two operands and set the processor's status flags accordingly; for example, a compare operation allows you to carry out the test $(X < Y)$ or $(x = y)$.	

Flow control: This group is concerned with modifying the sequence in which instructions are executed. There are three main subgroups: the unconditional branch that forces a jump to a specific point in a program, the conditional branch that forces a jump to a point in a program, if and only if a specified condition is met, and the subroutine call and return. The terms *branch* and *jump* are used largely interchangeably in computing.

STATUS FLAGS

The processor status register records the *outcome* of an instruction and implements conditional behavior by selecting one of two courses of action. Some processors call this register a *condition code* register. Conditional behavior lets us implement high-level language operations such as

if (x == 4) then

or

V 5.0

(i = 0; i < 20; i++).

A processor status register contains at least four bits, Z, N, C and V, whose values are set or cleared after an instruction has been executed. These four flags (i.e., status bits) are:

Z-bit Set if the result of the operation is zero

- N-bit Set if the result is negative in a two's complement sense; that is the leftmost bit is zero.
- C-bit Set if the result yields a carry-out; that is, if the C-bit is 1.
- V-bit Set if the result is out-of-range in a two's complement sense.

Typical CISC processors update these flags after each operation. Most RISC processors like the ARM require you to explicitly update the condition codes. This makes sense because you can update the condition codes at one point in the program and test them later as long as you haven't performed a second update. ARM processors require you to append an S after an instruction in order to force an update. Compare instruction do not need the S because, by definition, they update condition codes. Consider the following example:



ADD **r0**, r1, r2 ;add r1 to r2 SUB **r0**, r0, r3 ;subtract r3 to get r1+ r2 - r3 SUBS **r0**, r0, #5 ;subtract 5 to get r1+ r2 - r3 - 5 and update condition codes.

Consider the following example using 8-bit arithmetic. Suppose r0 contains 00110101_2 and r1 contains 01100011_2 , the effect of adding these two values together with ADD **r1**, r0, r1 would result in

 $\begin{array}{r} 00110101_{2} \\
+01100011_{2} \\
10011000_{2} \\ \end{array}$

The result is 10011000_2 which is deposited in r1. If we interpret these numbers as twos complement values, we have added two positive values to yield a negative result. Consequently, the V-bit is set to indicate arithmetic overflow. The result is not zero, so the Z-bit is cleared. The carry out is 0. The most-significant bit is 1 and the N-bit is set. Consequently, after this operation C = 0, Z = 0, N = 1, V = 1.

DATA MOVEMENT INSTRUCTIONS

Although the most frequently executed computer operation is *data movement*, it is incorrectly named because the one thing it does not do is to *move* data. Data movement instructions *copy* data; for example, the instruction MOV **r1**,r0 copies the contents of r0 to r1, but does not modify the value of r1. You could say that a data movement instruction is a *data propagate* or *data copy* instruction. You can also move a literal; for example, MOV **r1**,#12.

ARM processors have one highly unusual move instruction, the MVN, *move negative*, that takes the bits of one register, inverts them, and then copies them to the destination register; that is,

MVN **r0**, r1 has the effect $[r0] \leftarrow 0xFFFFFFF \oplus r1$ (performing an exclusive OR with 1 inverts a bit). This is not the two's complement of the register; it is the inverted bits of the register and differs from the two's complement by 1.

Like other RISC processors, the ARM has special-purpose data load and store instruction that copy data to and from memory; that is LDR and STR. We have encountered these instructions many times. CISC processors generally allow combined memory access and data operations. For example, the 68K instruction ADD **D2**, address that adds the contents of memory location address to register D2 and puts the sum in D2. This is a two-address instruction.

ARM processors do, in fact, implement a special *swap memory with register* (SWP) instruction that copies a memory location to a register and a register to the memory location. This operation is *atomic* and cannot be split up or interrupted (i.e., both the memory transfer to the register and from the register must take place without any interruption). You don't have to worry about this instruction because we will not be using it. It's intended for signaling between distributed processes.

ARITHMETIC INSTRUCTIONS

Arithmetic operations are those that act on numeric data (i.e., signed and unsigned integers).

ADD r2 ,r1,r0	Add	[r2] 🗲 [r1] + [r0]
ADC r2 ,r1,r0	Add with carry	[r2] ← [r1] + [r0] + C
SUB r2 ,r1,r0	Subtract	[r2] 🗲 [r1] - [r0]
SBS r2 ,r1,r0	Subtract with borrow	[r2] 🗲 [r1] - [r0] - C
RSB r2 ,r1,r0	Reverse subtract	[r2] 🗲 [r0] - [r1]
RSC r2 ,r1,r0	Reverse subtract with carry	[r2] ← [r0] - [r1] - C
MUL r2 ,r1,r0	Multiply (unsigned)	[r2] ← [r1] × [r0]
Add The ADD instruction adds the contents of two operands and deposits the result in the destination operand. One operand may be in memory. All addition and subtraction instructions update the contents of the condition code register unless the destination operand is an address register.

Add with Carry The *add with carry* instruction, ADC, is almost the same as the ADD. The difference is that ADC adds the contents of two registers together with the carry bit; that is ADC r2, r1, r0 performs $[r2] \leftarrow [r1] + [r0] + C$, where C is the carry bit generated by a previous operation.

This instruction is used in extended arithmetic. Suppose you wish to add two 64-bit numbers using a 32-bit ARM processor. Assume that the most-significant 32 bits of X are in r0 and the least-significant 32 bits in r1. The most-significant 32 bits of Y are in r2 and the least-significant bits in r3. We can perform the 64-bit addition X + Y by

ADD **r5**,r3,r1 ;Add the low-order 32 bits, update carry flag ADC **r4**,r2,r0 ;Add the high-order 32 bits plus any carry

We use ADD to add the two low-order 32-bit words. An addition records and carry bit generated by the addition and moves it to the C-bit. The ADC adds the high-order words together with any carry that was generated by adding the low-order words. The figure below demonstrates Z = X + Y where X, Y and Z are 64-bit values and the addition is to be performed with 32-bit arithmetic. Each of the operands is divided into an upper and a lower 32-bit word.



Subtract The subtract instruction subtracts the first source operand from the second source operand and puts the result in the destination. SUB r2, r1, r0 performs $[r2] \leftarrow [r1] - [r0]$. A subtract with borrow, SBC r2, r1, r0 performs $[r2] \leftarrow [r1] - [r0]$. A subtract with borrow, SBC r2, r1, r0 performs $[r2] \leftarrow [r1] - [r0]$. C (the carry bit is also subtracted from the result). This is entirely analogous to the corresponding add with carry instruction and is used in the same way to perform extended arithmetic.

The ARM processor has a most unusual variant of the subtract instruction, RSB (reverse subtract) that performs a reverse subtraction in which the operands are reversed; that is,

SBC r2, r1, r0 performs $[r2] \leftarrow [r1] - [r0]$ (normal subtraction) RSB r2, r1, r0 performs $[r2] \leftarrow [r0] - [r1]$ (reverse subtraction)

At first sight, this instruction seems pointless. After all, if you want to reverse the order of the operands, you can just write them the other way round and write SUB r2, r1, r0 or SUB r2, r0, r1 as required. However, ARM instructions that specify a literal operand are always of the form ADD r1, r2, #12 and the position of the literal cannot be changed. Therefore, the reverse subtraction allows you to perform the operation, say, 123 - r0 by writing RSB r0, r0, #123.

Multiplication All members of the ARM processor family have a basic multiplication instruction, MUL, that multiplies two 32bit words together and keeps the 32-bit lower-order word of the 64-bit result. Its format is:

MUL $\mathbf{r}_{d}, \mathbf{r}_{n}, \mathbf{r}_{m}$ which performs $[\mathbf{r}_{d(0:31)}] \leftarrow [\mathbf{r}_{n(0:15)}] \times [\mathbf{r}_{m(0:15)}]$

There are other multiplication instructions that are implemented by other members of the ARM family. We do not deal with these variations.

Members of the ARM processor family include an interesting and powerful instruction, the *multiply and accumulate* (MLA). When you multiply two numbers, what do you do with the product? More often than not, you add it to a running total. This is a fundamental operation in signal processing, image processing, and a range of other applications. For example, if **A** and **B** are vectors, then their *inner product* is defined as $s = \Sigma a_i b_i$ for i = 0 to n - 1.

The operation MLA \mathbf{r}_{d} , \mathbf{r}_{n} , \mathbf{r}_{m} , \mathbf{r}_{a} is defined as $\mathbf{r}_{d} = \mathbf{r}_{a} + \mathbf{r}_{n} \times \mathbf{r}_{m}$. This is unusual because it is a four-operand instruction. Suppose we want to form the inner product of two four-component vectors, we can write:

	MOV	r0, #4	;Set up loop count for 4 components
	MOV	r1, #0	;Clear the inner product
	ADR	r2,VecA	;r2 points to vector A
	ADR	r3,VecB	;r3 points to vector B
Next	LDR	r4, [r2],#4	;Repeat: Read an element from vector A
	LDR	r5, [r3],#4	; Read an element from vector B
	MLA	r1 , r4, r5, r1	; Multiply a pair of components and add to the total
	SUBS	r0, r0 , #1	; Decrement loop counter
	BNE	Next	:Repeat until all done



This code is very similar to that we used before, except that we have two pointers, one to each of the vectors. As an exercise, convert this into a program and run it on the Keil simulator. Provide your own data (by means of a DCD directive and debug the program. Ensure that the result is correct by evaluating it yourself and comparing it with the result from the simulator.

Division Most members of the ARM family have very few division instructions. In fact, none at all. If you wish to perform division you have to write a routine to perform the division using other operations. Fortunately, division is a surprisingly infrequent operation.

COMPARE INSTRUCTIONS

V 5.0

High-level language provide conditional constructs of the form

if $(x == y) \{a = b * c\};$

We examine how these constructs are implemented later. At this stage we are interested in the comparison part of the above construct, (x = y), that tests two variables for equality. We can also test for greater than or less than. The operation that performs the test is called *comparison*.

The ARM processor provides a compare instruction CMP r0, r1, that evaluates [r0] - [r1] and updates the bits of the bits in the condition code register accordingly, Consider the examples,

rO	r1	Operation	Processor status flags
10101010	10101010	CMP r0,r1	Z = 1, $C = 1$, $N = 0$, $V = 0$
10101010	00000000	CMP r0,r1	Z = 0, C = 1, N = 1, V = 0
10101010	11000001	CMP r0,r1	Z = 0, C = 0, N = 1, V = 0
10101010	01000001	CMP r0,r1	Z = 0, C = 1, N = 0, V = 1
01101010	10101010	CMP r0,r1	Z = 0, C = 0, N = 1, V = 1

A compare instruction is inevitably followed by a branch instruction that chooses one of two courses of action depending only on the outcome of the comparison. Here we demonstrate a compare followed by a branch.

Consider the high-level construct if $(x == 5) \{x = x + 10\}$;

We can write the following fragment of code.

```
r1, [r0]
                      ; get X in r1 (we assume that r0 is pointing at X in memory)
     LDR
     CMP
          r1,#5
                      ; is X == 5?
          Exit
                      ; if not equal then go to 'exit'
     BNE
          r1, r1, #10
     ADD
                      ;else add 10 to X
          r1,[r0]
                      ;restore X to memory
     STR
Exit
                      ;
```

In this example the branch instruction BNE Exit forces a branch (jump) to the line labeled by Exit if the outcome of the compare operation yields not zero.

LOGICAL INSTRUCTIONS

Logical operations allow you to directly manipulate the individual bits of a word. When a logical operation is applied to two 32-bit values, the logical operation is applied (in parallel) to each of the 32 *pairs* of bits; for example, a logical AND between words **A** and **B** would perform $c_i = a_i \cdot b_i$ for all values of bit *i*.

Mnemonic	Operation	Definition	Example
AND r2 , r1, r0	Logical AND	[r2] ← [r1] · [r0]	11110000 AND 10101010 = 10100000
ORR r2 ,r1,r0	Logical OR	[r2] ← [r1] + [r0]	11110000 OR 10101010 = 11111010
EOR r2 ,r1,r0	Exclusive OR	[r2] ← [r1] ⊕ [r0]	11110000 EOR 10101010 = 01011010
NOT r2 , r1	Logical NOT	[r2] ← [r1]	11110000 = 00001111
MVN r2 ,r1	Move negated	[r2] ← [r1]	11110000 = 00001111
BIC r2 , r1, r0	Logical AND NOT	[r2] ← [r1] · [r0]	11110000 AND $10101010 = 01010000$

The AND operation is *dyadic* and is applied to two source operands. Bit *i* of the source is ANDed with bit *i* of the destination and the result is stored in bit *i* of the destination. If $[r1] = 11001010_2$, the operation AND **r1**, #2_11110000 results in $[r1] = 11000000_2$. Remember that the symbol # indicates a literal or actual operand, and the prefix 2_ indicates a binary value.

The AND operation *masks* the bits of a word. If you AND bit x with bit y, the result is 0 if y = 0, and x if y = 1. A typical application of the AND instruction is to strip the parity bit off an ASCII-encoded character. That is,

AND **r2**, r1, #2 01111111

clears bit 7 of r1 to zero, and leaves bits 0 to 6 unchanged.

The OR operation is used to set one or more bits of a word to 1. ORing a bit with 0 has no effect, and ORing the bit with 1 sets it. For example, if $[r1] = 11001010_2$, the operation

ORR **r2**, r1, #2 11110000

results in $[r2] = 11111010_2$.

The exclusive OR, EOR, operation is used to toggle (i.e., invert) one or more bits of a word. EORing a bit with 0 has no effect, and EORing it with 1 inverts it. For example, if $[r1] = 11001010_2$, the operation

EOR **r2**, r1, #2 11110000

results in $[r1] = 00111010_2$.

V 5.0

By using the NOT, AND, OR, and EOR instructions, you can perform any logical operations on a word. Suppose you wish the clear bits 0, 1, and 2, set bits 3, 4, and 5, and toggle bits 6 and 7 of the byte in r0. You could write:

AND **r2**, r1, #2_11111000 ; Clear bits 0, 1, and 2 ORR **r2**, r1, #2_00111000 ; Set bits 3, 4, and 5 EOR **r2**, r1, #2_11000000 ; Toggle bits 6 and 7

If [r1] initially contains 01010101_2 , its final contents will be 10111000_2 . We will look at a more practical application of bit manipulation after we have covered branch operations in a little more detail.

ARM processors lack a NOT instruction that takes the logical complement of a word. However, the MVN, move negated instruction inverts the bits of the data being moved, so that MVN **r1**, r1 is the same as NOT r1.

ARM processors have a *bit clear instruction*, BIC, that performs a combined AND with a negation. The effect of a BIC is to AND the first operand with the negated second operand; that is, if the operands are *A* and *B*, then $C = A \cdot \overline{B}$. This instruction is used as a mask to selectively clear bits; for example, the mask word 00001111, can be used to clear the four lower-order bits of the source operand. Consider,

BIC **r2**, r1, #2 00001111 ; If r1 contains 00111010 the value of r2 is 00110000

SHIFT INSTRUCTIONS

A shift operation moves a group of bits one or more places left or right as the table below demonstrates.

Source	After shift left	After shift right
00110011	01100110	00011001
11110011	11100110	01111001
10000001	00000010	01000000

Shift operations are used to multiply or divide by a power of 2, rearrange the bits of a word, and access bits in a specific location of a word. Suppose 11001010_2 is shifted one place right. A logical shift right operation introduces a 0 into the leftmost bit position vacated by the shift, and the new value is 01100101_2 .

Although there are only two shift directions, left and right, there are several variations on the basic shift operation, depending on whether we are treating the value being shifted as an integer or a signed value, and whether we include the carry bit in the shifting.

All microprocessors have a set of shift operations that move the bits of a word one or more places left or right. However, the ARM processor is unique because it doesn't have an *explicit* shift operation. Instead, shift operations are incorporated in all data processing operations as an option. The second operand can be shifted before it takes part in an operation.



THE FOUR CLASSES OF SHIFT INSTRUCTION

Arithmetic shifts treat the data shifted as a signed two's complement value. The sign-bit is propagated by an arithmetic shift right. The number $1100101_2 = -54$ is negative, and an ASR gives 11100101_2 (i.e., -27).

When a word is shifted right arithmetically, the old least-significant bit is copied into the carry flag bit. An arithmetic shift left is equivalent to multiplication by 2, and an arithmetic shift right is equivalent to division by 2.

The number of bits to be shifted can be a *constant* defined in the program and the shift instruction always executes the same number of shifts. Some computers let you specify the number of bits to be shifted as the contents of a register. This allows you to implement *dynamic* shifts because you can change the contents of the register that specifies the number of shifts. The following figure graphically illustrates the various forms of shift.



A circular shift operation treats the data being shifted as a ring with the most-significant bit adjacent to the least-significant bit. Circular shifts result in the most-significant bit being shifted into the least-significant bit position (left shift), or vice versa for a right shift. No data is lost during a circular shift. Consider the following examples.

Shift type	Before circular shift	After circular shift
Rotate left, ROL	11001110	10011101
Rotate right, ROR	11001110	01100111

The last type of shift operation is called *rotate through carry*. The carry bit is treated as part of the word to be shifted. A circular shift is performed with the old carry bit being shifted into the register, and the bit lost from the register being shifted into the carry bit. Suppose that the carry bit is currently 1 and that the 8-bit value 11110000_2 is to be shifted one place right through carry. The final result is 11111000_2 and the carry bit is 0.

The ARM processor's shift options are:

- LSL #n The operand is shifted left by $0 \le n \le 31$ places. The vacated bits at the least-significant end of the operand are filled with zeros.
- LSR #n The operand is shifted right by $1 \le n \le 32$ places. The vacated bits at the most-significant end of the operand are filled with zeros.
- ASR #n The operand is shifted right by $1 \le n \le 32$ places. The vacated bits at the most-significant end of the operand are filled with zeros if the original operand was positive, or with 1s if it was negative (i.e., the sign-bit is replicated). This divides a number by 2 for each place shifted.
- ROR #n The operand is rotated right by $1 \le n \le 31$ places. The bit shifted out of the least-significant end is copied into the most-significant end of the operand. This shift preserves all bits.
- RRX The operand is rotated right by one bit. The bit shifted out of the least-significant end of the operand is shifted into the C-bit. The old value of the C-bit is copied into the most-significant end of the operand; that is, shifting takes place over 33 bits (i.e., the operand plus the C-bit).

Note that there should be *ten* versions if all possibilities are included (2 directions \times 5 modes). However, the missing operations can be synthesized from the existing operations; for example, an arithmetic shift left is identical to a logical shift left, and a rotate left can be achieved by rotating right (e.g., one shift left is the same as 31 shifts right).

If you want to perform a simple shift, you can apply it to a MOV instruction; for example,

MOV	r2, r1, LSL #4	; this will perform a 4-bit logical shift left
		; on the contents of r1 and copy the result to r2.

Let's look at another example. Consider the addition operation.

ADD	r2 ,r1,r0,	LSL #2	; this will perform a 2-bit logical shift left on the contents
			; of r0, add the result to r1, and put the sum in r2; that is
			; $[r2] \leftarrow [r1] + [r0] \times 4$

In this case, r0 is shifted left twice which is equivalent to multiplying by 4. Consequently, this forms the sum of r1 plus 4 r0. Such an operation is often used in calculating the value of addresses in array accesses and pointer manipulation.

BRANCH INSTRUCTIONS

A *branch instruction* modifies the flow of control and causes the program to continue execution at the *target address* specified by the branch. The simplest branch instruction is the *unconditional* branch instruction, B target, that always forces a jump to the instruction at the target address. In the following fragment of code, the 'B Here' instruction forces the ARM processor to execute next the instruction on the line with the labeled by Here.

	В	Here	; jump to the line that begins 'Here'	^
	•			
	•			111
Here	ADD	r1 , r1, r0		

In the next example, execution continues sequentially from instruction 1 to instruction 8, which is B 2000 (branch to instruction *N* at location 2000₁₆). The address of the first instruction is 1000_{16} and each instruction takes 4 bytes. Execution then continues with the instruction at location *N*. Instruction N + 5 is B 1040 (branch to instruction 17 at location 1040₁₆.) and a change of flow takes place again. Note that in reality, the ARM processor's branch instruction does not use an absolute

address but a relative address giving the distance to branch from the current instruction. We've used an absolute address here for convenient. In practice, the programmer uses a symbolic name (the line to branch to) and the assembler works out the appropriate relative offset.

1000 1004 1008 100C 1010 1014 1018 101C	Instruction 1 Instruction 2 Instruction 3 Instruction 4 Instruction 5 Instruction 6 Instruction 7 B 2000	The branch instruction at 2 executed next.	ction forces 2000 to be
1020	Instruction 9		> 2000 Instruction N
1024	Instruction 10		2004 Instruction N+1
1028	Instruction 11		2008 Instruction N+2
102C	Instruction 12		200C Instruction N+3
1030	Instruction 13	This block of code	2010 Instruction N+4
1034	Instruction 14	is not executed.	2014 B 1040
1038	Instruction 15		2017 2 1040
103C	Instruction 16		
1040	Instruction 17	The branch B '	1040 instruction
1044	Instruction 18	forces a jump t	o 1040.

We have already used a simple unconditional branch in ARM programs when we wrote Here B Here when we wanted to force the computer into an infinite loop at the end of a program.

The most important feature of any computer is its ability to implement *conditional behavior* by carrying out a test and then branching on the result of the text. The next example demonstrates the flow of control with a conditional branch.

Let's look at this conditional behavior in high-level language. Consider the following example of the high-level construct

if (x == 3) then y = 4.

We can translate this construct into the following ARM processor code.

```
CMP r1,#3 ; (x == 3)?

BNE exit ; if x is not 3 then leave

MOV r2,#4 ; if x is 3 then y = 4

exit ...
```

The instruction CMP r1, #3 compares the contents of register r1 with the literal 3 by evaluating [r1] - 3 and setting the status flags. If the result of the operation is zero, the Z-bit is set to 1. If the result is not zero (i.e., r1 does not contain 3), the Z-bit is set to 0.

The key instruction is BNE exit, which means 'branch on not zero to the instruction labeled exit'. The effect of this instruction is to test the Z-bit of the status flags and then branch to the instruction with the label 'exit' if Z = 0 (i.e., r1 is not 3). If r1 is 3, Z = 1, the branch is not taken and the MOV **r2**, #4 instruction is executed.

ARM processors provide 16 branch instructions of the form Bcc where the suffix cc defines the branch condition. Some of these 16 conditions are described below.

Version 1 [WORKBOOK FOR COMPUTER ORGANIZATION AND ARCHITECTURE: THEME AND VARIATIONS]

Mnemonic	Condition	Flags
BEQ	equal	Z = 1
BNE	not equal (not zero)	$\mathbf{Z} = 0$
BCS/BHS	carry set/higher or same	C = 1
BCC/BLO	carry clear/lower	$\mathbf{C} = 0$
BMI	negative	N = 1
BPL	positive or zero	$\mathbf{N} = 0$
BVS	overflow set	$\mathbf{V} = 1$
BVC	overflow clear	$\mathbf{V} = 0$

CONDITIONAL BRANCH EXAMPLE

Let's look at a simple application of conditional branching. You can implement a loop construct in the following way

Next	MOV	r0, #20	;load the loop counter r0 with 20 ;body of loop
	SUBS BNE	r0 ,r0,#1 Next	;decrement loop counter and set status flags ;repeat until loop count = zero

Let's look at another example of the use of branching. Suppose we have a number in r0 and we wish to set r1 to 1 if the number is odd, set r1 to 2 if the number is divisible by 4, and set r2 to 1 if it is greater than 200. This can be expressed as

r1 = 0; r2 = 0; if (r0 > 200) then r2 = 1 if (r0%2 == 1) then r1 = 1 //%2 is modulus 2 if (r0%4 == 0) then r1 = 2 //%4 is modulus 4

We can translate this into ARM processor code as

	MOV	r1, #0	;clear r1
	MOV	r 2,#0	;clear r2
	CMP	r 0,#200	;is r0 > 200
	BLE	Next	; if not then do next test
	MOV	r2, #1	; if it is, then set r2 to 1
Next	MOVS	r3 , r0, ROR #1	;dummy rotate right (and update status). R3 is a temp reg
	BCC	Next1	; if carry clear then try next test
	MOV	r1, #1	; if set, number odd, then set r1 to 1
	В	Exit	;and leave this block
Next1	BICS	<pre>r3, r0, #0xFFFF</pre>	FFFC ;clear all bits except 2 least sig and update status
	BNE	Exit	; if not zero then exit
	MOV	r1, #2	; if zero, number divisible by 4, then set r1 to 2
Exit			

As you can see, the code consists of tests and the actions or branches round actions. Note the way we test for divisibility by 4. The effect of BICS r3, r0, #0xFFFFFFC is to perform a logical AND between the contents of r0 and the logical inverse of the literal, which is 000...11. This operation masks r0 down to the two least-significant bits 000...bb. In order for the number to be divisible by 4, bb must be 00. Therefore, if we test for zero and the result is zero, the number was divisible by 4.

Note that in the testing we end up with some dummy values. In these cases we use r3 as a dummy register.

PREDICATED EXECUTION

The ARM processor is unusual in the sense that it provides a *conditional* (or *predicated*) execution mode that very few other processors support. When an instruction is read from memory, the processor checks its associated condition. If the condition is true, it is executed. If the condition is false, it is simply ignored and the next instruction in sequence dealt with. That is, instruction execution can be squashed.

All ARM processor instructions are conditional. So far, we have ignored this because the default condition is always execute. If you wish to attach an explicit condition, you simply add a condition suffix to the end of an instruction. Exactly the same suffixes used by conditional branches; for example EQ. Consider the following example,

ADDEQ **r0**, r1, r2

This is a conditional version of the ADD. If the Z-bit (zero) is true, this instruction will be executed. Otherwise, it will be ignored. Let's look at the previous example again.

We can translate this into ARM processor code using conditional instructions.

```
MOV
      r1, #0
                         ; r1 = 0
                         ; r^2 = 0
MOV
      r2,#0
      r0, #200
                         ; if (r0 > 200) then r2 = 1
CMP
MOVGE r2, #1
                         ; if (r0\%2 == 1) then r1 = 1 //\%2 is mod 2
MOVS r3, r0, ROR #1
MOVCS r1, #1
BICS r3, r0, #0xFFFFFFC; if (r0\%4 == 0) then r1 = 2 //\%4 is mod 4
                         ; if zero, number divisible by 4, then set r1 to 2
MOVEQ r1, #2
```

Notice how much more compact the code it. All the branch instructions have gone. We perform a test and then a predicated operation. There's nothing to stop us doing multiple operations; for example,

CMP	r1, #123	;	if	r1	==	= 12	23	
ADDEQ	r3 , r3, #1	;		rЗ	=	r3	+	1
SUBEQ	r4 ,r4,#5	;		r4	=	r4	+	1

In this case, two operations are conditional and they are both predicated on outcome of the test on r1. We can also make tests themselves predicated in order to test compound conditions; for example.

if (r0 > 200) && (r2 == 4) then r2 = 1

CMP	r0, #200	;	if	r0	> 200)		
CMPGT	r3 , r3, #4	;		if	r3 =	r3	+	1
MOVEQ	r2, #2	;			r4 =	r4	+	1

Here, we do a test (CMP r0, #200) and then a second test if the outcome is true. The third instruction is executed only if the previous two tests were true.

BRANCH AND LINK

The ARM processor includes a *branch with link* instruction that executes a branch and saves the return address. This allows you to call a subroutine and then return to the calling point. The form of the instruction is BL target, where BL is the opcode and target the address of the point at which execution is to continue. The branch with link instruction stores the return address in the link register r14. Consequently, programmers should not use r14 as a general-purpose register. If you use a second BL instruction you will overwrite the previous address in the link register.

Consider the following example.

	MOV	r1, #4	;	put parameter in r1
	MOV	r2, #3	;	put second parameter in r2
	BL	TestSub	;	call the subroutine
			;	return here
TestSub	ADD	r3 , r1, r2	;	very simple subroutine to do addition
	MOV	PC,lr	;	<pre>same as MOV r15,r14 (forces jump back)</pre>

THE MAXIMUM SEQUENCE COUNTER

For our next example we return to the problem of the sequence counter we introduced in Chapter 1 of Computer Organizatrion and Architecture. Our problem is to take a sequence of digits, one by one, and determine the longest run in a sequence of digits as the following figure demonstrated. The figure below is taken from the text and shows a string of digits where the longest sequence is 4.



The pseudocode we developed to solve this problem is expressed as follows.

```
1.
    Read the first digit in the string and call it New Digit
2.
    Set the Current_Run_Value to New_Digit
    Set the Current Run Length to 1
3.
    Set the Max Run to 1
4.
    REPEAT
5.
      Read the next digit in the sequence (i.e., read New Digit)
6.
7.
      IF its value is the same as Current Run Value
             THEN Current Run Length = Current Run Length + 1
8.
9.
             ELSE {Current Run Length = 1
                  Current Run Value = New Digit}
10.
11.
      IF Current Run Length > Max Run
             THEN Max Run = Current Run Length
12.
    UNTIL The last digit is read
13.
```

This code can be converted into ARM assembly language in the following way.

```
AREA
                 RunLength, CODE, READWRITE ; find the longest run in a sequence
                                                 ; r9 points to the sting
          ADR
                 r9,
                                                 ; r0 is i (1 initially)
         MOV
                  r0,#1
                                                 ; r1 is New_Digit (initially the first element in the string
          LDR
                  r1,[r9]
                                                 ; r2 is the Current_Run_Value
         MOV
                  r2,r1
         MOV
                  r3,#1
                                                 ; r3 is the Current_Run_Length (set to 1)
         MOV
                                                 ; r4 is the Max_Run_Length (set to 1)
                  r4,#1
                                                 ; Repeat: point to next element
Repeat
         ADD
                  r9,r9,#4
          LDR
                  r1,[r9]
                                                          Read next digit
                                                 ;
                                                          Compare New_Digit and Current_Digit
         CMP
                  r2,r1
                                                 ;
         ADDEQ
                  r3,r3,#1
                                                          IF same THEN Current_Length=Current_Length+1
                                                 ;
         MOVNE
                  r3,#1
                                                                  ELSE Current_Run_Length = 1
                                                 ;
         MOVNE
                  r2,r1
                                                                  Current_Run_Value = New_Digit
                                                 ;
                                                          IF Current_Run_Length > Max_Run
                  r3,r4
          CMP
                                                 ;
         MOVPL
                  r4,r3
                                                                  THEN Max_Run = Current_Run_Length
                                                 ;
                                                          increment digit counter
          ADD
                  r0,r0,#1
                                                 ;
          CMP
                  r0,#18
          BNE
                                                 ; until all digits tested
                  Repeat
Park
          B
                  Park
                                                 ; parking loop
String
          DCD
                  2,2,2,2,2,3,6,6,8,6,4,2,2,3,2,2,2 ; the string
         END
```

The interesting part of this code is in red. Instead of using a conventional test and branch operation (e.g., CMP r1, r2 followed by BEQ abc) we make use of conditional or predicated execution. Consider the code fragment:

CMP	r2,r1	;	Compare New_Digit and Current_Digit
ADDEQ	r3,r3,#1	;	IF same THEN Current_Length=Current_Length+1
MOVNE	r3,#1	;	ELSE Current_Run_Length = 1
MOVNE	r2,r1	;	Current_Run_Value = New_Digit

Initially, r2 is compared with r1 which sets the zero and negative flags. The ADDEQ instruction is executed if r1 and r2 were equal. The next two instructions are predicated by NE (not equal or not zero). If r1 is not equal to r2 then both these instructions are executed. Both parts of the IF THEN ELSE clause are mutually exclusive and we do not need branch instructions.

The following snapshot shows the execution of the code in the Keil simulator at the end of the program (note that this example uses a different sequence of digits to the one in the figure above). Register r4 contains the length of the longest run which is 5.



THE STACK

The stack is a *last-in-first-out* (LIFO) data structure. It is a queue with only one end; that is, new items enter at the same point as old items leave. Items leave a stack in the reverse order in which they arrive. A LIFO queue is the same as a stack in conventional English. If you pile books on top of each other and then remove them from the top, it behaves exactly like a stack.

A stack can be used in many ways. However, we are interested in the following three applications of the stack:

- 1. Storing subroutine return addresses
- 2. Passing parameters from a program to a subroutine
- 3. Providing temporary storage (local workspace) in a subroutine.

The following diagram illustrates one possible stack structure (there are four variations that are determined by the way in which the stack grows). The stack can be located in any region of memory. This stack grows up towards low addresses; that is, the address of an item at the top of the stack is lower than the address of an item at the bottom of the stack.

Address register r13 is used as the *stack pointer* by convention. It should not be used for any other purpose. When an item enters the stack it is said to be *pushed* on the stack. When an item leaves the stack, it is said to be *pushed* off the stack.





In this stack, the stack pointer points to the item at the top of the stack. This item is the last element pushed on the stack and will be the first item pulled off the stack (hence the term LIFO or *last-in first-out*).

Suppose you have an item in register r0 and wish to push it on the stack. Since the stack pointer points at the top of the stack, the pointer must be moved up (i.e., decremented) before the item is moved to the location now pointed at. We can do this by

SUB **r13**, r13, #4 ;decrement the stack pointer to move it up STR r0, [**r13**] ;now put the item on the stack

Fortunately, you can combine these two operations together by using the ARM processor's auto-decrementing addressing mode

```
STR r0, [r13, #-4]!
```

This instruction stores the contents of r0 at an address -4 bytes from r13; that is, 4 bytes above it. The contents of r13 are then decremented by 4.

To pull (pop) a word off the stack, we perform the inverse operation; that is, we read the item currently at the top of the stack pointed at by r13 and then increment r13 to point to the new item at the top of the stack. We can do this by:

LDR	r0,[r13]	;read the item at the top of the stack
ADD	r13, r13, #4	;increment the stack pointer

Once again, you can combine these two operations together by using the ARM processor's auto-incrementing addressing mode

LDR **r0**, [r13], #4

The next figure shows the state of the stack after pushing r0 and then r1 on the stack by executing STR r0, [r13, #-4]! and STR r0, [r13, #-4]!. Note that we've used the sp synonym for r13.



The next step is to look at the subroutine and demonstrate how subroutines use the stack to handle return addresses, pass parameters, and create space for local variables required by a subroutine during its life.

SUBROUTINE CALLS

A subroutine is a piece of code that is called, executed and a return is made to the calling point. Subroutines are very important because they implement the *function* or *procedure* at the high-level language level. At this point, we are interested only in the principle of the subroutine call and return.



This figure demonstrates the subroutine call. Code is executed sequentially until a subroutine call is encountered. The current place in the code sequence is saved and control is then transferred to the subroutine; that is, the first instruction in the subroutine is executed and the processor continues executing instructions in the subroutine until a *return* instruction is encountered. Then, control is transferred back to the point immediately after the subroutine call by retrieving the saved return address.

Consider a simple subroutine called ABC that calculates the value of $2x^2$ (where x is a 16-bit value passed in r0). This subroutine is called by the instruction BL ABC (branch to

subroutine) that jumps and saves a copy of the return address in the link register, r14. A return back to the calling point is made by copying the return address from the link register to the program counter, r15. Note that typical CISC processors like the Intel IA32 family automatically use the stack to store the return address and employ an RTS (return from subroutine) instruction to return to the calling point. A typical ARM processor call and return routine is:

	BL 	XYZ	;call XYZ ;return here
XYZ			;The subroutine ;
	MOV	<pre>pc,lr</pre>	;copy saved address to PC to return

Let's create a simple example. Consider a subroutine that calculates the value of $x^2 + 1$, where x is in register r0 and the result is returned in r0.

Loop	MOV	r0, #4	;set up a dummy parameter
	BL	SQR1	;call SQR1
	MOV	r3, r2	;do something with the result
	B	Loop	;stay here
SQR1	MUL	r1 , r0, r0	;Calculate x^2 (note – can't use source register as destination)
	ADD	r0 , r1, #1	;Add 1 to get $x^2 + 1$
	MOV	pc , lr	;Return

The following snapshots show the state of this program at the point the subroutine has been called. Note that r14 (the link register) contains the return address 0x00000008 (this is the third instruction MOV **r3**, r2)

E:\Ce	engageBook	\GengageWorkB	ook\Ex	ample7	.uvproj	- µVision4				
<u>F</u> ile <u>E</u>	dit <u>V</u> iew	<u>P</u> roject Fl <u>a</u> sh	<u>D</u> ebug	Pe <u>r</u> ip	herals	<u>T</u> ools <u>S</u> VC	S <u>W</u> indow	<u>H</u> elp		
I 🗋 🖆	i 🖬 🎒 🕴	み 単語 🧐			門際	内内目	= i= //± //≵	2	- 🗟 🥙 🞑	ه 📀
RST	EL 🛞 🥐	∂ {} ³ * ()		Q. 🖪		🐺 • 📖 ·	• 📴 • 🔜 •	🔜 🔹	📓 • 🎘 • 🖳	
Register	s		τ×		Subrou	itineEg1.asm	۱			▼ ×
Registe	r	Value		01		AREA	Subroutin	eTest,	CODE, READONLY	
Cu	rrent			02		MOV	r0,# 4	;set	; up a dummy parameter	
	- R0	0x00000004		03		BL	SQR1	;cal	.1 SQR1	
	·R1	0x0000000		04		MOV	r3, r2	;do	something with the result	
	· R2	0x0000000		05	Loop	в	Loop	;sta	y here	
	· R3	0x0000000		06						
	· R4	0x0000000		->07	5QR1	MUL	r1,r0,r0	;Cal	culate x2	
	· R5	0x0000000		08		ADD	r0,r1,#1	;Add	l 1 to get x2 + 1	
	- R6	0x0000000		09		MOV	pc,lr	;Ret	urn	
	· R7	0x0000000		10		END				
	· R8	0x0000000		11						
	· R9	0x0000000								
	· R10	0x00000000								
	·R11	0x0000000								
	R12	0x0000000								
	R13 (SP)	0x0000000	_							
	R14 (LR)	0x0000008								
	R15 (PC)	0x00000010								
+····	CPSR	0x00000D3								
+	SPSR	0x0000000								
	er/System		-	L						
🖭 Proje	ect 🛛 🧮 Regi	sters								F
										Simula

This subroutine mechanism has two flaws. First, because the multiply instruction can't use the same register for source and destination, we have to use r1 to receive the result. This means that r1 is used by the subroutine and any data in it will be overwritten. Second, this subroutine can't call another subroutine or be reused because the return address is in r13, the link register, and another subroutine call would overwrite it.

One way of solving these problems is to save the link register at the beginning of a subroutine and then restore it at the end. Where should it be saved? The stack is the best place to save registers because the stack grows upward, and all data is placed on top and not removed or overwritten as new data is added. We can also save other registers on the stack. We can now rewrite the previous subroutine as:

SQR1	STR	lr ,[sp,#-4]!	;Save link register on the stack
	STR	r1 ,[sp,#-4]!	;Save register r1 on the stack
	MUL	r1, r0,r0	;Calculate x^2 (remember that we can't use source register as destination with MUL)
	ADD	r0, r1,#1	;Add 1 to get $x^2 + 1$
	LDR	r1 ,[sp] ,#4	;Restore register r1 from the stack
	LDR	1 r ,[sp],#4	;Restore link register from the stack
	MOV	<pre>pc,lr</pre>	;Return

The detailed code is as follows. Note the markers.

	AREA	SubroutineTest,	CODE, READWRITE ; make readwrite because we have the stack in this area
	ADR	sp ,Base	; point to the base of the stack
	MOV	r1, #0xAB	; dummy value for r1
	MOV	lr, #0x11	; dummy value for link register, r14
	MOV	r0, #4	; set up a dummy parameter in r0
	BL	SQR1	; call SQR1
	MOV	r3 , r0	; do something with the result which is in r0
Loop	В	Loop	; stay here
SQR1	STR	lr,[sp,#-4]!	; Save link register on the stack
	STR	r1,[sp,#-4]!	; Save register r1 on the stack
	MUL	r1 , r0, r0	; Calculate x ² (note - can't use source register as destination)
	ADD	r0 , r0, #1	; Add 1 to get x ² + 1
	LDR	r1 ,[sp],#4	; Restore register r1 on the stack
	LDR	lr ,[sp],#4	; Restore link register on the stack
	MOV	<pre>pc,lr</pre>	; Return
	DCD	0x89ABCDEF,0,0,0),0x12345678 ; stack area
Base	DCD END	Охаааааааа	; stack base and dummy data

Too Fast Tieth.	graject Figuh	Debug Perjoherals Tools SVI	DAC2 Minigon Refs	
	10.01	- PERAS	7 7 7 7 7 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1	
10 U I	0 11 11 10]•]• ■• □• ■• [ו].	
egisters	8.2	Disassembly		8 X
Register	Value	2: ADR sp.	p.Base spoint to the base of the stack	
Current	100000000	20x00000000 E28FD044	4 ADD R13, FC, #0x00000044	
RD	0x00000000	0x00000004 E3A010AB	8 MOV 81.#0x000000AB	
- R1	0x00000000	4: MOV 1r.	r,#0x11 /dummy value for link register, r14	
H2	0x0000000	0x00000008 E3A0E011	1 NOV R14, #0x00000011	
- RA	0x00000000	5: MOV ±0,	0,84 :set up a dummy parameter in r0	
- 85	0x00000000	0x0000000C E3A00004	4 MOV R0, #0x0000004	
- 86	Gx00000000	6: BL SQI	QR1 (call SQR1	
-R7	0x00000000	0x0000010 EB000001	1 BL 0x0000001C	
Ra	0x00000000	71 BOV IS,	3,FD Ido something with the result which is in FO	
- R9	0-00000000	St Lorp B Lor		
R10	Gx00000000	9:	oop shear nere	
R11	0x00000000	Cox00000018 EAFFFFE	E B 0x00000018	
B12 - B13 (CP)	0x0000000	10: SQR1 STR 1r,	r, [sp, #-4]! ;Save link register on the stack	
- R14 (LP)	0x0000000	0x0000001C E52DE004	4 STR R14, [R13, #-0x0004]!	
- B15 (PC)	0x00000000	11: STR #1,	1,[sp.#-H]) :Save register il on the stack	
H-CPSR	0x00000003	20x00000020 E52D1004	4 BIR R1,[R13, #-0x0004]!	
IE SPSR	0x00000000	12: MUL F1,	1,r0,r0 (Calculate x^2 (can't use source register as destination	n)
H Deen/System		0x0000024 £0010090	O BOL RI, KU, KU	
Fast Interrupt		0x00000028 #2800001	1 ADD B0 B0 #0+00000001	
e kterupt		14: LDS +1.	1.[sn].#4 (Restore register r) from the stack	
Supervisor		0x0000002C E49D1004	4 LDR R1,[R13],#0x0004	
Finite Aport		15: LDR 1r,	r, [ap],#4 :Restore link register from the stack	
E kternal		0x00000030 E49DE004	4 LDR R14, [R13], #0x0004	
PC 8	0x00000000	16: MOV pc,	c, 1r ;Return	
Mode	Supervisor	0x00000034 E1A0P00E	E MOV PC, R14	
States	0	0x00000038 SSABCDEF	F SIMMIIB R11:, (RO-R3, RS-R8, R10-R11, R14-PC)	
Sec.	0.00000000	C	a simon as as as	
				1
		Example8.asm		+ ×
		01 AREA Subrout	utineTest, CODE, READWRITE /readwrite because the stack is in this area	-
		⇔02 ADB sp,Base	se /point to the base of the stack	1
		03 MOV r1,#GRA	sab rdummy walue for ri	
		of NOV 11, FUEL	Ril Fainny Value for link register, ri4	
		00 80 9001	/set up a commy parameter in ry	
		07 MOV r3.r0	the sensitive with the result which is in rd	
		00 Loop B Loop	/stay here	
		09		
		10 SQR1 STR lr, [sp,	p,#-fl! sSave link register on the steck	
		11 STR r1, (sp.	p.#-4]1 /Save register rI on the stack	
		12 MOL r1, r0, r	,r0 (Calculate x*2 (can't use source register as destination)	
		13 ADD r0,r0,#	.#1 /Add 1 to get x"# + 1	
		14 LDR r1, (sp)	p), #4 /Restore register ri from the stark	
		15 LDR IT, [sp]	p],#t :Restore link register from the stack	
		17 NUV pc, Ir -)	- And Corn	
		18 DCD DxS9ARC	SCDEF. 0. 0. 0 x12045678 retack area	
		19 Base DCD OxAAAAA	AAAAA /stack base and dummy data	
		20 ENE	and the second	
Project Real	isters			20
Direjest Regi	isters			<u>ম</u>
Biogent Biogen	isters			ک ده ۲۵۰
D Project EReg lemory 1 Address: 0	isters			کر د ه 1911
Dibagent = Régi temony 1 Address: 0 x000000000 ; E3	2 88 D0 44	A0 10 A5 E3 A0 E0 11 1	E3 A0 00 04 E8 00 00 01 E1 A0 30 00 EA FF FF FE E5 2D E0 04 E5 2D 10 04	80 01 00
Addess 0 x00000005: E3 x00000005: E3 x00000027: 53	2 0F D0 44 0 E2 80 00 a ab 00 00	A0 10 A5 E3 A0 E0 11 1 E4 5D 10 04 E4 5D E0 00	E3 A0 00 04 E8 00 00 01 E1 A0 30 00 EA FF FF FE E5 2D E0 04 E5 2D 10 04 04 E1 A0 F0 DE E5 AB CD EF 00 00 00 00 00 00 00 00 00 00 12 34 56 00 00 00 00 00 00 00 00 00 00 00 00 00	1 00 00 00
☐ Project ■ Reg emony 1 Address: 0 x00000000000000000000000000000000000	2 0F D0 44 0 E2 85 00 A AA 00 0D	A0 10 A8 E3 A0 E0 11 1 E4 9D 10 04 E4 9D 20 0 00 00 00 00 00 00 00	E3 A0 00 04 E8 00 00 01 E1 A0 30 00 EA FF FF FE E5 2D E0 04 E5 2D 10 04 04 E1 A0 F0 02 89 A5 CD EF 00 00 00 00 00 00 00 00 00 00 00 00 00	80 01 00 75 AA AA 00 00 00

The next snapshot shows the situation at the point the subroutine ${\tt SQR1}$ is called.

Version 1 [WORKBOOK FOR COMPUTER ORGANIZATION AND ARCHITECTURE: THEME AND VARIATIONS]

E:\CengageBoo	k\CengageWurl	xbook (Example8.uvproj - pVision4	_ [D] ×
Ere Edit Yiew	Project Fight	Debug Pelipherals Tools SVCS Window Help	
	(A 40-01)		
18 国 Q (1 0 11 11		
Registers		K Duassembly	8 X
Register	Value	2: ADR sp. Sase :point to the base of the stack	*
Current	1.0000000	3. MOV of 1004B and Kis, PG, FOX00000044	
RD	0x00000004	CX0000004 E3A010AB MOV R1.#0x000000AB	
- P2	0x00000048	4: MOV 1r, #Owll /dummy value for link register, r14	
- 83	0x00000000	0x0000008 E3A0E011 MOV R14,#0x00000011	
- Fi4	0x00000000	5: MOV r0,#4 ratt up a dummy parameter in r0	
- 85	Gx00000000	Storesource Esacoso4 Nov Rd, Fox000000004	
86	Ex00000000	Cadonoodio Ebiopool BL 0x000001C	
-00	0x0000000	7: MOV r3,r0 ido something with the result which is in r0	
- 89	0x00000000	0x00000014 E1R03000 MOV R3,R0	
-R10	0x00000000	D: Loop B Loop Istay bere	
-R11	0x00000000		
R12	0+00000000	10: SQR1 STR 1r, [sp, #-5]! JSave link repister on the stack	
H13 (SP)	0x0000004C	Cox0000001C #52DE004 STR R14,[#13,#-0x0004]1	
R15(BC)	0:0000010	11: STR f1,[sp,#-4]! :Save register f1 on the stack	
H-CPSR	0x00000003	Dx00000020 E52D1004 STR R1,[R13,#-0x0004]!	
I SPSR	0x00000000	12: MUL F1,F0,F0 (Calculate x'2 (can't use source register as destination)	
H User/System		131 ADD TOTOLE 1 Jadd 1 to get $x^2 + 1$	
Fast Interrupt		0x00000028 E2900001 ADD R0,R0,#0x00000001	
B Supervisor		14: LDS rl,[sp],#4 :Restore register rl from the stack	
H Abot		D0x000002C E49D1004 LDR R1,[R13],\$0x0004	
Undefined		15: LDR ir, sp),#6 ;Restore linx register from the stack	
🖂 – Internal		lei MOV pe, lr ikturn	
PC \$	0x9000001C	0x00000034 EIA0F00E MOV PC, R14	
States	7	0x00000038 S9ABCDEF STMMIIB R11!, (R0-R3, R5-R5, R10-R11, R14-PC)	
Sec.	0.00000000	0x000003C 00000000 ANDEQ R0,R0	
1000			
		1) Example8.asm	+ ×
		O AREA SubroutineTest, CODE, READWRITE /readwrite because the stack is in this area	
		02 ADR sp,Base /point to the base of the stack	-
		03 MOV r1,#0xAB ydunny value for r1	
		04 MOV ir, tixil schemp value for link register, r14	
		05 BUY F1.F1 755 Up a dumty parameter in r0	
		07 MOV r3.r0 rds senting with the result which is in r0	
		08 Loop B Loop /stay here	
		09	
		File StR in (sp.3-1) - Save link register on the stack	
		12 MOL r1.r0.r0 (Calculate x*2 (can't use concer residere as destination)	
		13 ADD TO, TO, 41 radd 1 to get s'a + 1	
		14 LDR r1. (ap), #4 /Restore register r1 from the stark	
		15 LDR Ir, [sp], #4 :Restore link register from the stack	
		16 NOV pc, ir sReturn	
		18 DCE DXSSABCDEF. 0. 0. 0. 0912345578 Starts	
		19 Base DCD OxAAAAAAAA ystack base and dummy data	_
		20 ENE	-1
Project Rec	pisters		- 2Ê
Henen I			
menory a			
Address: 0			4
0x00000000; #	2 88 00 44	E3 A0 10 AB E3 A0 E0 11 E3 A0 00 04 EB 00 00 01 E1 A0 30 00 EA FF FF FE E5 2D E0 04 E5 2D 10 04 E0	01 00
Dx00000027: 5	0 E2 80 00	01 E4 9D 10 04 E4 9D E0 04 E1 A0 F0 0E 89 AB CD EF 00 00 00 00 00 00 00 00 00 00 00 12 34 56 78	aa aa
0x0000004E: A	A AA 00 0D	00 00 00 00 00 00 00 00 00 00 00 00 00	• 00 00
Call Stack + Lint	It's Memory I		
		Simulation into approximation	ec III
		Churchen The Annual Contraction	01 /

	Eicher alfin	Debug Peripheral	I Jooli SVCS Window	. Refe	
) 🥥 🖬 🟈	(A 10-01)	- KR123	2月3日第第6月	1 🙋	
	1 07 11 11 4		N 🖓 • 🔝 • 🦉 • 🗯	 ・回・図・ 次・ 1 	
gisters	a x	Depassembly			8.2
legister	Value	0x00000000	E28FD044 ADD	R15, PC, #0x00000044	1
Current	Contractory (3:	MOV 11, #0xAB	Jdummy value for rl	1
RD	0x00000005	4:	MOV 1r.#0x11	:dummy value for link register, r14	
- 82	0x00000048	80000000x0	ESAGEO11 MOV	R14, #0x00000011	
- 83	0x00000000	51	MOV ±0,#4	fact up a dummy parameter in r0	
-Fi4	0x00000000	Cx0000000C	E3A00004 MOV	R0,#0x00000004	
R5	0x00000000	01000000000	EB000001 BL	0x0000003C	
86	Bx00000000	7:	MOV 13,10	rdo something with the result which is in r0	
-89	0x0000000	0x00000014	E1A03000 MOV	R3,R0	
- R9	0x00000000	S: Loo	p B Loop	catay here	
R10	0x00000000	97 Rovootoonte	ENFERTER B	0+00000010	
R11	0x00000000	10: 3QR	1 578 lr. (sp. 4-4)	1 :Save link register on the stack	
- 812 - Etta (GP)	0-00000000	0x0000001C	ESZDE004 STR	R14, [R13, #-0x0004]!	
- R14 (LR)	Bx00000014	11:	STR r1,[sp, #-4]	Save register r1 on the stack	
R15(PC)	6x00000034	Cx00000020	E52D1004 5TR	R1, [R13, #-0x0004] !	
H-CPSR	0x00000003	Ecv0000024	F0010090 MUL	(calourace x = (dan t use source register as destination)	
III SPSR	0400000000	13:	ADD 10,10,01	sAdd 1 to get x^2 + 1	
Fast Interruct		0x00000028	E2800001 ADD	R0, R0, #0x0000001	
- Interrupt		14:	LDR r1,[sp],#4	sRestore register ri from the stack	
Supervisor		0x0000002C	E49D1004 LDR	R1, [R13], #0x0004	
Abot		B0x00000030	E49DE004 LDR	R14. [R13].#0x0004	
Undefined		14:	MOV po, 1r ; Retur	EN	
PC 8	0x00000034	-00x0000034	EIAOFOGE MOV		
PC 8 Mode	0x00000034 Supervisor	0x00000034	EIAOFDOE NOV S9ABCDEF STNHIIB	FC,R14 R11!, (RO-R3,R5-R6,R10-R11,R14-PC)	
PC 8 Mode States	0x00000034 Supervisor 20	Cx00000034	EIAOFDOE NOV S9ABCDEF STNHIIB ODODODOO ANDEQ ODODODOO ANDEQ	FC,R14 R11!, (R0-R3,R5-R6,R10-R11,R14-PC) R0,R0,R0 R0 R0 R0	
PC 8 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 0x00000035 0x0000003C 0x0000003C	ELAOFOGE NOV S9ABCDEF STMHIIB OCCODOGO ANDEQ OCCODOGO ANDEQ	FC,R14 R11!, (RO-R3,R5-R6,R10-R11,R14-PC) R0,R0,R0 R0,R0 	
PC 8 Mode States Sec	0x200000034 Supervisor 20 0.00000000	Cx00000034 Cx00000038 Cx0000003C Cx00000050	ELAOPDOE NOV 99ABCDEF STNHIIB 000000000 ANDEQ 000000000 ANDEQ	FC,R14 R11!, (RO-R3,R5-R6,R10-R11,R14-PC) R0,R0,R0 R0,R0,R0	ъŝ
PC 8 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 0x00000035 0x0000003C 0x00000050 4[] 1 Example8	ZIAOFOGE NOV S9ABCDEF STNHIIB OCCODOCO ANDEQ OCCCODOC ANDEQ SSM	FC,R14 R11!, (RO-R3,R5-R6,R10-R11,R14-PC) R0,R0,R0 R0,R0,R0	म • `
- PC 8 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 0x0000038 0x000003C 0x0000030 Cx00000040 Cx00000040 Cx00000040 Cx00000040 Cx00000040 Cx00000032 Cx00000000 Cx0000000 Cx0000000 Cx0000000 Cx000000 Cx000000 Cx000000 Cx000000 Cx000000 Cx000000 Cx000000 Cx000000 Cx000000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx0000	ZIADEGOE MOV S9ABCDEF STNHILD OCCODOC ANDEQ OCCODOC ANDEQ ANDEQ ANDEQ SSM F sp, Dase	PC,R14 R111, (RO-R3,R5-R6,R10-R11,R14-PC) R0,R0,R0 R0,R0,R0 Point to the base of the stack	1 1 1
- PC 8 - Mode - States - Sec	0x00000034 Supervitor 20 0.00000000	Cx00000034 0x0000038 0x000003C 0x0000040 Cx0000040 Cx0000040 Cx0000040 Cx0000040 Cx0000040 Cx0000040 Cx00000032 Cx00000032 Cx00000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx0000038 Cx00000038 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx000000000 Cx00000000 Cx00000000 Cx0000000000	ZIÃOFOCE MOV S9ABCDEF STNHIB OCCODOC ANDEQ OCCODOC ANDEQ OCCODOC ANDEQ SSM SSM 8 SD, Base /; V r1, f0xAB /; V l f0xAB /;	PC.R14 R11:,(R0-R3,R5-R6,R10-R11,R14-PC) R0,R0,R0 R0,R0,R0 Point to the base of the stack dummy value for ti	ม์ • •
- PC 8 - Mode - States - Sec	0x00000034 Supervitor 20 0.0000000	Cx00000034 0x0000035 0x0000035 0x0000050 Cx0000050 0x0000050 0x0000050 0x0000050 0x0000050 0x00000054 0x00000034 0x00000034 0x00000034 0x00000034 0x00000034 0x00000035 0x00000035 0x00000035 0x00000035 0x00000035 0x00000035 0x00000035 0x00000035 0x00000035 0x00000035 0x00000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x0000035 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x000000000 0x000000000 0x0000000000	ZIÃOFOCE MOV SPARCEF STMHIB OCCODOC ANDEQ COCCODOC ANDEQ COCCODOC ANDEQ SEM SE SP,Dase 25 V r1,‡0xAB 70 V r1,‡0xAB 70 V r0,24	PC.R14 R11:,(R0-R3,R5-R6,R10-R11,R14-PC: R0,R0,R0 R0,R0,R0 R0,R0,R0 R0,R0,r0 R0,R0,r0 R0,R0,r0 R0,R0,r0 R0,R0,r0 R0,R0 R0 R0,R0 R0 R0,R0 R0 R0,R0 R0 R0,R0 R0 R0 R0,R0 R0 R0 R0 R0 R0 R0 R0 R0 R0 R0 R0 R0 R	ม์ •
- PC 8 - Mode - States - Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 0x0000035 0x0000035 0x0000040 (1) Cx0000040 (2) Cx0000040 (3) Cx0000040 (4) Cx0000040 (4) Cx00000035 (4) Cx00000035 (4) Cx00000035 (4) Cx00000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000035 (4) Cx0000005 (4) Cx0000035 (4) Cx0000035 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx000005 (4) Cx000005 (4) Cx000005 (4) Cx000005 (4) Cx000005 (4) Cx000005 (4) Cx000005 (4) Cx00005 (4) Cx0005 (4) Cx0005 (4) Cx0	ZIAOFOGE MOV S9ABCDEF STMHIB ODCODDOC ANDEQ COCODDOC ANDEQ COCODDOC ANDEQ Sam Sam F sp,Base // V r1,#0xAB // V r1,#0xA1 // SQB1 //	PC,R14 R11:,(R0-R3,R5-R6,R10-R11,R14-PC: R0,R0,R0 R0,R0,R0 references point to the base of the stack dummy value for r1 dummy value for r1 dummy value for link register, r14 set up a dummy parameter in r0 cell SURM	म • •
menta PC 3 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000054 0x0000036 0x00000050 (1) 0x00000050 (1) 02 02 03 03 04 00 04 04 00 05 04 00 05 00 06 05 00 06 05 00 07 00 07	ZIAOFOGE MOV S9ABCDEF STMHIIB OD000000 ANDEQ O0000000 ANDEQ O0000000 ANDEQ V r1, #0x88 70 V r1, #0x88 70 V r1, #0x81 74 V r0, #4 73 SQR1 75 V r3, r0 75	PC,R14 R11:,(R0-R3,R5-R6,R10-R11,R14-PC: R0,R0,R0 R0,R0,R0 references and the stack dummy value for r1 dummy value for r1 dummy value for link register, r14 set up a dummy parameter in r0 ceil SQR1 do something with the result which is in r0	भ भ
PC 3 PC 3 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000054 0x00000036 0x00000050 Cx00000050 Cx00000050 Cx00000050 Cx00000050 Cx0000050 Cx0000050 Cx0000050 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000054 Cx00000056 Cx000000056 Cx00000056 Cx00000056 Cx00000056 Cx00000056 Cx00000056 Cx00000056 Cx00000056 Cx00000056 Cx00000056 Cx00000056 Cx00000056 Cx000000000 Cx000000000 Cx000000000000	EXADEGOE HOV S9ABCDEF STNHIB OCCODDCO ANDEQ OCCODDCO ANDEQ Internet Internet ssm Internet F sp, Base Internet V r1, FOXAB Internet V r0, #1 Internet SQM1 Internet Internet V r0, #1 Internet SQM1 Internet Internet Loop Internet Internet	PC.R14 R11:,(R0-R3,R5-R6,R10-R11,R14-PC: R0,R0,R0 R0,R0,R0 	म • • •
PC 3 PC 3 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 Cx0000035 Cx0000035 Cx0000035 Cx0000050 Cx0000050 Cx0000050 Cx0000050 Cx0000050 Cx00000050 Cx00000055 Cx000000054 Cx000000054 Cx000000054 Cx000000054 Cx00000055 Cx00000055 Cx00000055 Cx00000055 Cx00000055 Cx00000055 Cx00000055 Cx0000055 Cx0000055 Cx0000055 Cx0000055 Cx0000055 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx00005 Cx00005 Cx000005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx05 Cx05	ZIÃOFOGE MOV S9ABCDEF STNHIB OCCODOC ANDEQ OCCODOC ANDEQ OCCODOC ANDEQ SSM SSM SSM SSM SSM SSM SSM SS	PC.R14 R11: (RO-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 root to the base of the stark dummy value for r1 dummy value for r1 dummy value for r1 hk register, r14 set up a dummy parameter in r0 cell SQRA do Something with the result which is in r0 stay here	1 1 1
PC 3 PC 3 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 0x0000035 0x0000035 0x0000035 0x0000050 0x0000050 0x0000050 0x0000050 0x0000050 0x0000005 0x00000035 0x00000034 0x00000034 0x00000034 0x00000034 0x00000034 0x00000035 0x0000005 0x00000005 0x00000005 0x00000005 0x00000000 0x0000000 0x0000000 0x0000000 0x0000000 0x0000000 0x0000000 0x0000000 0x0000000 0x0000000 0x00000000	ZIÃOFOGE MOV S9ABCDEF STNHIB OCOCODOC ANDEQ OCOCODOC ANDEQ OCOCODOC OCOCOCO OCOCOCO OCOCOCODOC OCOCODOC OCOCO OCOCOCOCO O	PC.R14 R11: (R0-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 rows have an ex- point to the base of the stack dummy value for r1 dummy value for r1 dummy value for r1 dummy value for r1 dummy parameter in r0 cell SQR1 do something with the result which is in r0 stay here Save link register on the stack	<u>ب</u> ب
PC 3 PC 3 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 Ox0000035 Ox0000035 Ox0000035 Ox0000040 (1) Cx0000040 Ox0000040 Ox0000040 Ox000004 Ox000004 Ox00000034 Ox00000034 Ox00000034 Ox00000034 Ox00000034 Ox00000034 Ox00000034 Ox00000035 Ox0000035 Ox0000035 Ox00000035 Ox00000035 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox000005 Ox000005 Ox000005 Ox0000005 Ox0000005 Ox0000005 Ox000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox000005 Ox000005 Ox000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox0000005 Ox00000005 Ox0000005 Ox0000005 Ox00000005 Ox00000005 Ox000000005 Ox00000005 Ox0000005 Ox00000005 Ox00000005 Ox0000005 Ox0000000000 Ox0000000000 Ox0000000000	ZIAOFOGE MOV S9ABCDEF STMHIB ODGODODO ANDEQ 00000000 ANDEQ 0000000 ANDEQ 00000000 ANDEQ 00000000 ANDEQ 00000000 ANDEQ 0000000 ANDEQ 00000000 ANDEQ 000000000000000000000000000000000000	PC.R14 R11: (R0-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 R0, R0, R	بر ب
PC 3 PC 3 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 0x0000035 0x0000035 0x0000036 (1) Cx0000046 (2) Cx0000046 (2) Cx0000046 (3) Cx0000046 (4) Cx0000035 (4) Cx00000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0000005 (4) Cx0005 (4) Cx000005 (4) Cx0005 (4) Cx0005 (4) Cx05	ZIADFOGE MOV S9ABCDEF STNHIIB OCCODDC ANDEQ OCCODDC ANDEQ COCCODC ANDEQ COCC	PC,R14 R11:,(R0-R3,R5-R6,R10-R11,R14-PC: R0,R0,R0 R0,R0,R0 R0,R0,R0 R0,R0,R0 R0,R0,R0 R0,R0,R0 R0,R0,R0 R0,R0,R0 R0 R0,R0 R0 R0 R0 R0 R0 R0 R0 R0 R0 R0 R0 R0 R	1 •
PC 3 PC 3 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 Cx00000036 Cx0000035 Cx0000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx00005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx00005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx00005 Cx000005 Cx00005 Cx00005 Cx0005 Cx0005 Cx0005 Cx	EXAMPNOE HOV S9ABCDEF STNHIIB OCCODDOC ANDEQ OCCODDOC ANDEQ Internet Internet SSM Internet # Sp, Base Internet V Int, FOXAB Internet V Int, Fox, FO Internet B Int, Fox, FO Internet D Int, Fox, FO Internet	PC.R14 R11: (RO-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 root to the base of the stack dummy value for r1 dummy value for r1 dummy value for r1 dummy value for r1 set up a dummy parameter in r0 ref1 SyR1 do Something with the result which is in r0 stay here Save link register on the stack Save register r1 on the stack Save register r1 on the stack Calculate x*2 (can't use source register as destination) Add 1 to get x*2 + 1 Restore register r1 from the stack	भ • •
PC 3 PC 3 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 Cx0000035 Ox000003C Ox000003C Ox0000050 Cx0000050 Cx0000050 Cx0000050 Cx0000050 Cx0000050 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000003 Cx0000000 Cx0000000 Cx0000000 Cx000000 Cx00000 Cx000000 Cx000000 Cx000000 Cx000000 Cx000000 Cx00000 Cx000000 Cx00000 Cx00000 Cx000000 Cx000000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx00000 Cx0000 Cx000000 Cx000000 Cx000000 Cx000000 Cx0000000 Cx0000000 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx000000000 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx0000000 Cx0000000 Cx0000000 Cx0000000 Cx0000000 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx00000000 Cx000000000 Cx00000000 Cx00000000 Cx00000000 Cx000000000 Cx00000000 Cx0000000000	ZIÃOFOGE MOV S9ABCDEF STMHIE OCCODOCO ANDEQ OCCODOCO ANDEQ SSM 8 SP, Dase /; V 11, #0xAB /; SQR1 /; SQR1 /; L 000 /; R 11, [sp, ±-4]! /; L 11, r0, r0 /; L 01, r0, #1 /; R 11, [sp], ±4 /; R 11, [sp	PC.R14 R11: (R0-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0 R0, R0 roote to the base of the stack dummy value for r1 dummy value for r1 dummy value for r1 dummy value for r1 dummy value for r1 for for r1 set up a dummy parameter in r0 ref1 SQR1 do something with the result which is in r0 stay here Save link register on the stack Save rejister r1 on the stack Calculate x^2 (cen't use source register as destination) Add 1 to get x^2 + 1 Restore register from the stack Restore risk register from the stack	1
PC 3 PC 3 Mode States Sec	0x0000004 Supervisor 20 0.00000000	Cx00000034 Ox00000035 Ox00000035 Ox00000050 Cx00000050 Cx00000050 Cx00000050 Cx00000050 Cx00000050 Cx00000035 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx0000000 Cx0000005 Cx0000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx000000005 Cx00000005 Cx00000005 Cx00000005 Cx0000005 Cx0000005 Cx0000005 Cx000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx005	ZIÃOFOGE MOV S9ABCDEF STMHIIB OCCODOCO ANDEQ OCCODOCO ANDEQ OCCODOCO ANDEQ OCCODOCO ANDEQ OCCODOCO ANDEQ OCCODOCO ANDEQ OCCODOCO ANDEQ OCCODOCO ANDEQ OCCODOCO ANDEQ SSM V r1, #0xAB V r1, #0xAB V r0, #1 SQR1 V r0, #1 SQR1 V r3, r0 E r1, [sp, #-4]1 L r1, r0, r0 E r1, [sp], #4 V r0, #1 ZI r1, [sp], #4 V r0, IT SREturn	PC.R14 R11: (RO-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 rooten to the base of the stark dummy value for r1 dummy parameter in r0 cell SQR1 do something with the result which is in r0 stay here Save link register on the stack Save register r1 on the stack Save register r1 on the stack Calculate x-2 + 1 Restore register r1 from the stack Restore link register from the stack	1
PC 3 PC 3 Mode States Sec	0x00000034 Supervisor 20 0.00000000	Cx00000034 Ox00000035 Ox00000035 Ox00000036 Ox00000040 (1) Cx00000040 (2) Cx0000040 (3) Cx00000035 Cx00000005 Cx000005 Cx000005 Cx0000005 Cx0000005 Cx0000005 Cx000 Cx000005 Cx0000005 Cx0000005 Cx000005 Cx0000005 Cx0000005 Cx000005 Cx000005 Cx000005 Cx0000005 Cx0000005 Cx0000005 Cx000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx000005 Cx000005 Cx00005 Cx00005 Cx00005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx005	ZIADFOCE MOV S9ABCDEF STMHIB OCCODDOC ANDEQ OCCODDOC ANDEQ OCCODDOC ANDEQ OCCODDOC ANDEQ OCCODDOC ANDEQ OCCODDOC ANDEQ OCCODDOC ANDEQ OCCODDOC ANDEQ SSM F S9,Base 25 V r1, ±0xAB 22 V r1, ±0xAB 22 V r0, ±1 22 Lcop 22 B r1, [sp, #-4]! 22 B r1, [sp, #-4]! 22 B r1, [sp, #-4]! 22 D r0, r0, #1 22 B I1, [sp], #4 22 V po,Ir SMENTER	<pre>PC.R14 R11: (R0-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 r0 for r1 dummy value for r1 dummy parameter in r0 call SQRI do something with the result which is in r0 stay here Save register r1 on the stack Save register r1 on the stack Save register r1 on the stack Save register r1 on the stack Rastore register r1 from the stack Restore register r1 from the stack Restore link register from the stack</pre>	1
riteria - PC 3 - Mode - States - Sec	0x0000004 Supervisor 20 0.00000000	Cx00000034 0x0000035 0x0000035 0x0000036 (1) (2) 0x0000040 (1) (2) 0x0000040 (2) 0x0000040 (3) 0x0000040 (4) 00 00 00 00 00 00 00 00 00 0	EXADEGGE MOV S9ABCDEF STNHIB OCCODDE ANDEQ OCCODDE ANDEQ OCCODDE ANDEQ OCCODDE ANDEQ OCCODDE ANDEQ OCCODDE ANDEQ OCCODDE ANDEQ OCCODDE ANDEQ OCCODE ANDEQ OCCODE O	<pre>PC.R14 R11:,(R0-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 rows have of the stack dummy value for r1 dummy value for r1 dummy walue for r1 dummy walue for ink register, r14 set up a dummy parameter in r0 cell SQRI do something with the result which is in r0 stay here Save link register on the stack Save register r1 on the stack Restore register r1 from the stack Restore register r1 from the stack Restore register r1 from the stack Restore link register from the stack</pre>	1 •
PC 3 PC 3 Mode States Sec	0x0000004 Supervisor 20 0.00000000	Cx00000034 Cx0000035 Cx000005 Cx000005 Cx000005 Cx000005 Cx0000005 Cx00005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx00005 Cx000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx00005 Cx000005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx005 Cx0	EXAMPLATE NOV S9ABCDEF STNHIIB OCCODDOC ANDEQ OCCODDC ANDEQ COCCODDC ANDEQ COCCODC ANDEQ COCCODDC ANDEQ COCCODC ANDEQ COCCOD	<pre>PC.R14 R11: (RO-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 root to the base of the stack dummy value for r1 dummy value for r1 dummy value for r1 dummy present r1 r0 cell SQRA do Something with the result which is in r0 stay here Save link register on the stack Save register r1 on the stack Save register r1 on the stack Calculate x*3 (can't use source register as destination) Add 1 to get x*3 + 1 Restore register r1 from the stack Restore register r1 from the stack Associated and stack 0,0x12345678 iscack area stack hase and dummy date</pre>	1) 1
Ficeraa PC 3 Mode States Sec	0x00000034 Supervisor 20 0.000000000	Cx00000034 Cx0000035 Cx00000035 Cx00000035 Cx00000035 Cx0000000 Cx0000000 Cx0000000 Cx00000000 Cx00000000 Cx000000000 Cx000000000 Cx0000000000	ZIÁDEGOE MOV S9ABCDEF STNHIIB OCCODODO ANDEQ OCCODOC ANDEQ 	<pre>PC.R14 R11:,(R0-R3,R5-R6,R10-R11,R14-PC: R0,R0,R0 R0,R0 roose roose</pre>	1
Pograt EReg	0x00000034 Supervice 20 0.000000000	Cx00000034 Cx0000035 Cx00000035 Cx00000035 Cx00000035 Cx00000035 Cx00000035 Cx000000035 Cx00000000 Cx000000000 Cx000000000 Cx0000000000	ZIÁDEGOE MOV S9ABCDEF STNHIIB OCCODOCO ANDEQ OCCODOCO ANDEQ 	<pre>PC.R14 R11, (RO-R3, R5-R6, R10-R11, R14-PC) R0, R0, R0 R0, R0, R0 root to the base of the stark dummy value for r1 dummy value for r1 dummy value for r1 dummy parameter is r0 cell SQR1 do Something with the result which is in r0 stay here Save link register on the stack Save register r1 on the stack Save register r1 on the stack Calculate x*2 (cen't use source register as destination) Add 1 to get x*2 * 1 Restore register r1 from the stack Restore link register from the stack Restore link register from the stack 0,0x12945678 /stack area stark hase and dummy date</pre>	1
PC 3 PC 3 Mode States Sec	okd0000034 Supervice 20 0.00000000	Cx00000034 0x0000035 0x000005 0x000005 0x000005 0x000005 0x000005 0x000005 0x000005 0x000005 0x000005 0x000005 0x000005 0x000005 0x00005 0x0000005 0x0000005 0x0000005 0x0000005 0x0000005 0x0000005 0x0000005 0x0000005 0x0000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x000000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x0000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000005 0x00000000 0x00000000 0x000000000 0x00000000	ZIÃOFOGE MOV S9ABCDEF STNHIIB OCCODODO ANDEQ OCCODODO ANDEQ 	<pre>PC.R14 R11, (R0-R3, R5-R6, R10-R11, R14-PC) R0, R0, R0 R0, R0 r0, r0, r0, r0 r0, r0, r0, r0, r0 r0, r0, r0, r0, r0, r0, r0, r0, r0, r0,</pre>	1 •
PC 3 PC 3 Mode States Sec Project Reg mory 1	6x00000034 Supervice 20 0.000000000	Cx00000034 Cx0000035 Cx000005 Cx0000035 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx00005 Cx000005 Cx00005 Cx000005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx00005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx05	23ADETOLE MOV 39ABCDEF STNHIIB OCCODDOC ANDEQ OCCODDOC ANDEQ 3500 5 SP, Base 7 11, FOXAB 7 11, FOXAB 7 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,	PC.R14 R11:,(RO-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 root to the base of the stark dummy value for r1 dummy value for r1 dummy value for r1 dummy parameter in r0 cell SQRA do Something with the result which is in r0 stay Aere Save link register on the stack Save register r1 on the stack Save register r1 on the stack Calculate x*2 (cen't use source register as destination) Add 1 to get x*2 *1 Restore register r1 from the stack Restore link register from the stack 0.0xt2345678 istack area stark hase and dummy date	1 - -
Project Reg PC 3 Mode States Sec Sec Sec Sec Sec Sec Sec Sec	0x00000000 Supervice 20 0.000000000	Cx00000034 Ox0000035 Ox0000035 Ox0000035 Ox0000035 Ox0000035 Ox0000035 Ox0000035 Ox000035 Ox000035 Ox000035 Ox000005 Ox000005 Ox000005 Ox000005 Ox000005 Ox000005 Ox000005 Ox000005 Ox000005 Ox000005 Ox000005 Ox000005 Ox00000005 Ox000005 Ox000005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox00005 Ox0005 Ox00005 Ox005 Ox005	EXAMPLAGE MOV S9ABCDEF STNHIIB OCCODDOC ANDEQ OCCODDOC ANDEQ 	PC.R14 R11, (RO-R3, R5-R6, R10-R11, R14-PC) R0, R0, R0 R0, R0, R0 root to the base of the stark dummy value for r1 dummy value for r1 dummy value for r1 dummy parameter in r0 cell SQRA do Something with the result which is in r0 stay here Save link register on the stack Save register r1 on the stack Save register r1 on the stack Calculate x*2 (cen't use source register as destination) Add 1 to get x*2 * 1 Restore register r1 from the stack Restore link register from the stack 0,0x12345678 istack area stark hase and dummy date	
IProjett EReg mony 1 doeson 000000000000000000000000000000000000	0x0000004 Supervice 20 0.00000000	Cx00000034 0x00000035 0x0000035 0x0000036 0x0000046 Cx0000046 Cx0000046 Cx0000046 Cx0000046 Cx0000046 Cx0000046 Cx0000046 Cx00000035 Cx000000035 Cx00000035 Cx000000035 Cx00000035 Cx00000035 Cx00000035 Cx00000035 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx00000005 Cx00000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx000005 Cx000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx00000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx0000005 Cx000005 Cx000005 Cx000005 Cx000005 Cx00005 Cx00005 Cx00005 Cx00005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx0005 Cx005	ZIÃOFOGE MOV S9ABCDEF STNHIIB OCCODOCO ANDEQ OCCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ SSM F SP, Base /; V 11, #0xAB /; V 11, #0xAB /; V 11, #0xAB /; V 11, #0xAB /; SQR1 /; SQ	PC.R14 R11:,(R0-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 rooteness of the stack dummy value for r1 dummy value for r1 dummy value for r1 dummy value for r1 dummy value for r1 for r0 r1 star parameter in r0 rei1 SQR1 do something with the result which is in r0 stay here Save link register on the stack Save register r1 on the stack Save register r1 on the stack Calculate x*2 (can't use source register as destination) Add 1 fo get x*2 * 1 Restore link register from the stack Restore link register from the stack Save has end dummy date 0, 0x12945678 /stack sizes stark hase end dummy date	
PC 3 PC 3 Mode States Sec	0x0000000 20 0.00000000 0.00000000 20 0.00000000	CX00000034 OX0000035 OX0000035 OX0000035 OX0000040 CX0000040 CX0000040 CX0000040 CX0000040 CX0000040 OX0000040 OX0000040 OX0000040 OX0000040 OX0000040 OX000004 OX0000004 OX00000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX0000004 OX00000004 OX00000004 OX0000004 OX0000004 OX0000004 OX00000004 OX000000000 OX000000000 OX00000000 OX0000000000	ZIÃOFOGE MOV S9ABCDEF STNHIIB OCCODOCO ANDEQ OCCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ COCODOCO ANDEQ SSM S S9, Base /; V r1, #0xAB /; V r1, #0xAB /; V r0, #1 /; SQR1 /; SQR	PC.R14 R11:,(R0-R3, R5-R6, R10-R11, R14-PC: R0, R0, R0 R0, R0, R0 re-re-tro-res point to the base of the stack dummy value for r1 dummy value for r1 dummy value for r1 dummy value for r1 dummy parameter is r0 cell 5QR1 do something with the result which is in r0 stay here Save link register on the stack Save register r1 on the stack Save register r1 on the stack Calculate x-2 + 1 Restore link register from the stack Restore link register from the stack 0, 0x12345476 istack sizes stark base and dummy date 04 EB 00 00 01 E1 A0 30 00 EA FF FF EE S2D E0 04 E5 2D 10 04 E0 F0 02 E5 AS CD EF 00 00 00 00 00 00 00 00 00 00 00 00 00	

The final screen shows the situation immediately before the return that is made by copying the link register to the PC.

Version 1 [WORKBOOK FOR COMPUTER ORGANIZATION AND ARCHITECTURE: THEME AND VARIATIONS]

E\Cengagefloo	&\GengageWork	iBook \Example&uvgiruj - pVision4	
Fie Fae Xiew	graject Figuri	Beond Lethouse and Table and Lethouse and Le	
	A STOLL		
	9 (0· () · () · ()		100000
Registers	ů ×	C Disasembly C	1 ×
Register	Value	2: ADS sp.Base : point to the base of the stack Decorpood Kingtonia ADD Bill PC Exception	크
- Current	0-0000005	S: MOV r1,#0xAB (dummy value for r1	
RI	0x00000014	2000000004 E3A010AB MOV R1,#0x00000AB	
- R2	0x00000000	Store concerned a state of the state of the sector of the state of the	
84	0x0000000	51 MOV r0,#4 ret up a parameter to pass to SQB1	
RS	0x00000000	0x0000000C E3A00004 MOV R0,#0x0000004	
R5	0x00000000	C: DL SQMI CCALISQMI	
88	0x00000000	71 HOV r3,r2 /do something with the result	
R9	0x00000000	0x00000014 E1A03002 MOV R3,R2	
R10	Gx00000000	CiLoop B Loop interview	
R11 R12	0x0000000	9: SQR1 MUL r1,r0,r0 (Calculate x**2 (note - can't use source register as destination	on)
R13 (SP)	0x0000004C	CAUCOCOCCLE ECOLOGEO MUL RI,RG,RO	
- R14 (LR)	0+00000010	AVI SIR IF, SP, +-41 Josev int register on the stack	
H15(HC) (H-CPSR	0-00000003	11: SIR rl,[sp,‡-4]: /Save register rl on the stack	
E SPSR	0x00000000	2x00000024 E52D1004 STR R1,[R13,#-0x0004]!	
User/System		CARDONDER X280000 ADD R0.80.404000001	
E Interupt		13: LDR lr, [sp], #4 ;Restore link register from the stack	
Supervisor		2x0000002C E49DE004 LDR R14, [R13], #0x0004	
Abort		TAT IN TATATOPIAN AND A CONTRACT AND	
Condefined		15: HOV pc.lz /Return	-
PC \$	0x00000034	DOMADOODSH EIAGEDDE MOV PC,RI4	
Mode	Supervisor	0x00000035 5WABCLEF STRATS K17, NO-85, 85-85, KIO-RIT, KI+-PC)	
Sec	29 0.0000000	0x00000040 00000000 ANDEQ R0,R0,R0	
		0x0000014 00000010 ANDEQ R0,R0,R0,LSL R0	
		Oxfordonote labababa Bigs OxFRABAFC	1.00
		Approximate southers an as as	- 10
			* ^
		0 Akra Subroutinetest, CODE, HADMAITE Jake readyrite because we have the stack in this block and the stack in this base of the stack.	5 420
		03 HOV T1, #OxAB /downy value for r1	
		04 MOV 1r,#Oxl1 rdumy value for link register, r14	
		(5 MW F0.84 / rest up a parameter to pass to SgRI	
		0 HOV r3.r2 and senthing with the result	
		08 Loop B Loop rstay here	
		100 SQR1 MUL r1,r0,r0 //Calculate #**3 (note - can't use source register as destination) th STP is less solid (can be be a stark)	
		11 STR r1, [sp.8-4]: /Save register r1 on the stack	
		12 ADD r0,r0,#1 :Add 1 to get x**3 + 1	
		13 LDS 1r, spj. 44 /Restore link register from the stack	
		2)15 MOV pc.1r Jaturn	
		16 DCD 0x89ABCDEF,0,0,0,0x12345678 intack area	12
		17 Base DCD OxAAAAAAA rstack base and dumny date	-
Project ER	pisters	10 Enu	1
Mamora 1			
and many a			(Date)
Address 0			11
0x00000000; E	12 8F DO 44 1	E3 A0 10 AB E3 A0 E0 11 E3 A0 00 04 E5 00 00 01 E1 A0 30 02 EA FF FF FE E0 01 00 90 E5 2D E0 04 E5 2D	10
0x000000271 8	14 E2 80 00	01 E4 9D E0 04 E4 9D 10 04 E1 A0 F0 0E 85 AB CD EF 00 00 00 00 00 00 00 00 00 00 10 00 00	AA
UXDODUGO4E: A	A AA 00 00 0	ao a	
Call State.	Memory 1		
		Simulation t1: 0.0000000 set	

MULTIPLE SUBROUTINE CALLS

We next extend the example by demonstrating a multiple call. Here, we've used a typical CISC instruction BSR ABC to implement the call. A branch to subroutine instruction automatically saves the return address on the stack (unlike the ARM that saves it in the link register). Because subroutine return addresses are stacked, you can call subroutines from within a subroutine (nesting). In the following figure, the main body of the code calls subroutine ABC. At the end of the subroutine, a return instruction makes a return to the point immediately following the call. In this example, the subroutine is called from two different places and yet a return is made to the correct point in each case.

In order to achieve this objective with the ARM processor, we can use the ARM's *block move instructions* that copy multiple registers to and from the stack.



USING BLOCK MOVE INSTRUCTIONS

In practice, programmers don't use the simple code we've written above to save registers on the stack and to retrieve them. Traditionally, RISC processors provide simple, regular instructions that take one cycle (in principle) to execute. The ARM processor family is different because it has a set of instructions that perform multiple actions. These instructions are called block move operations and are able to copy the contents of several registers to or from memory.

When you first encounter ARM's block move instruction you are likely to be overwhelmed by their apparent complexity. In fact, they are not complex; it's just that there are several options to choose from. So, to keep things simple, we will just discuss one option here. These two block move instructions we are going to use are:

STMFD	;Push a group of registers on the stack
LDMFD	Pull a group of registers off the stack

Couldn't be simpler. The STMFD mnemonic stands for *store multiple registers full descending*. The expression "full descending" tells you two things. The term *full* means that the stack points at the top item on the stack. The term *descending* tells you that the stack grows towards lower addresses as items are pushed. This is exactly the same type of stack we've already described. When we wish to store data on the system stack, we have to use r13 which we can write as sp. We also have to write sp! or r13! to tell the assembler that we want to use automatic indexing. Finally, we have to create a register list by enclosing the registers to be moved between braces; that is, {r0,r1,r7} specifies registers r0, r1 and r7, We can use a dash to denote a sequence of registers; for example {r0-r5,r8,r11} indicates the register list r0, r1, r2, r3, r4, r5, r8, and r11.

To push r0 and r1 on the stack, we write STMFD sp!, {r0,r1}. Similarly, to pull r1 and r2 off the stack, we write LDMFD sp!, {r0,r1}

Suppose we use a different register list for the store and retrieve multiple register operations. What would happen if we execute $STMFD \ sp!, \{r0, r1\}$ then LDMFD $\ sp!, \{r5, r7\}$? Well, we push r0 and r1 and then we pull their values off the stack and transfer them to registers r5 and r7. In other words, we've copied one group of registers into another group.

Let's demonstrate these block move instructions in action.

```
BlockMove, CODE, READWRITE ; make readwrite because we have the stack in this area
      AREA
                                     ; point to the base of the stack
      ADR
             sp,Base
                                     ; dummy value for r0
      MOV
             r0, #0xAB
                                     ; dummy value for r1
      MOV
             r1, #0xCD
                                     ; dummy value for link register, r14
      MOV
             lr,#0xDE
                                     ; call Test
      ΒL
             SOR1
                                     ; stay here
Loop B
             Loop
                                     ; save r0, r1, Ir on the stack
Test STMFD sp!, {r0, r1, lr}
                                     ; let's do something pointless
      MOV
             r0, #0x11
                                     ; let's do something pointless
      MOV
             r1, #0x22
                                     ; let's change the link register
      MOV
             r14, #0x22
             r3, r0, r1
                                     ; ladd r0 and r1 and put the result in r3
      ADD
                                     ; pull r0, r1, Ir off the stack
      LDMFD sp!, {r0,r1,pc}
            0x89ABCDEF,0,0,0,0x12345678 ; stack area
      DCD
Base DCD
            0xaaaaaaaa
                                     ; stack base and dummy data
      END
```

This code is built on the previous example and uses the same basic format and stack structure. We use markers in memory like 0xAAAAAAA and register values like 0xAB so that we can see the data in memory when we come to debug the code. We're going to run this example and examine the state of the registers and memory at three points.

The next snapshot shows the situation immediately after the program has been loaded.

Elle Edit View	Core7\Desktop\Cer Eroject Flash D	ngageBook\Ge sbug Padphar	igageWork an Ioola	Scok/Example8.uvprg - pVision4 SVCS Window Belp		
200 F	. २ ८ १ । स । 10 २ ।		5 @ • []	******		
Registers		Disasteritati				÷ 🖸
Register	Value	(1) BA	ckMove.a	sm		▼ ×
 ■ Current B0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 (SP) R14 (LR) R15 (PC) ■ CPSR ♦ SPSR 	0x50000000 0x5000000 0x5000000 0x5000000 0x5000000 0x5000000 0x50000000 0x50000000 0x50000000 0x50000000 0x50000000 0x50000000 0x50000000 0x50000000 0x50000000 0x50000000 0x50000000 0x50000000 0x500000000	01 01 02 03 04 05 05 05 05 05 05 05 05 10 11 12 13 ♥14 16 15 15 15 15 15 15 15 15 15 15	AREA ADR MOV MOV BU BL P B STMFD MOV MOV MOV ADD LONFD CDR = DCD	BlockMove,CODE, READNRITE sp,Base r0,#0xAB r1,#0xCD Test Loop sp!,(r0,r1,lr) r0,#0x11 r1,#0x22 r14,#0x22 r3,r0,r1 sp!,(r0,r1,pq) 0x89ABCDEF,0,0,0,0x1234567 0xADADADA	<pre>/make readwrite because we /point to the base of the s /dummy value for r1 /dummy value for r1 /dummy value for link regis /call Test /stay here /stay here /stay here /stay bare /let's do something pointle /let's do something pointle /let's do something else po /let's change the link regi /add r0 and s1 and put the /pull r0, r1, ir off the st #/stack area /stack base and dummy data</pre>	have the stack in this area tack ter, s14 ck sm intless ster result in r3 ack
Project	spinters	18	END		Marker for the	<u>.</u>
Memory 1 Address 0	/	First data memory a	stored i fter the	n code.	base of the stack.	• E
0x00000000; H3 0x00000030; 89 0x00000060; 00 0x00000060; 00	BT DO 3C A0 AS CD EF 00 00 OO OO OO OO OO OO OO OO OO OO OO OO OO	00 AB E3 A0 00 D0 00 00 00 D0 00 00 00 D0 00 00	55 CD E3 55 50 50 55 50 50 55 50 50 55 50 50	AD 80 DE EB 00 D0 00 EA FF FF 73 00 00 00 12 34 56 78 AA AA AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	19 17 40 03 83 A0 01 11 83 A0 12 23 00 00 01 00 00 01 00 01 02 00 01 02 03 </td <td>E3 A0 E0 22 E0 60 30 01 E8 80 00 00 0 00 00 00 00 00 00 00 00 00 00 00 00 0 0 00 00 00 00 00 00 00 00 00 00 00 0 0 00 00 00 00 00 00 00 00 00 00 00 0 0 00 00 00 00 00 00 00 00 00 00 00 0 0 0 0</td>	E3 A0 E0 22 E0 60 30 01 E8 80 00 00 0 00 00 00 00 00 00 00 00 00 00 00 00 0 0 00 00 00 00 00 00 00 00 00 00 00 0 0 00 00 00 00 00 00 00 00 00 00 00 0 0 00 00 00 00 00 00 00 00 00 00 00 0 0 0 0
					Simulation	11: 0,00000000 sec 1:14 Ci69

C\Users\AlanCore7\Desktop\CengageBi	ook\GengageWorkSo	ok/Example8.uvproj - pVision4	-				×
File Edit View Project Flash Debug J	Parlochargh Tools 9	VCS Window Help					
	1 have been	Note that the stack pointer is pointing at	• • • •	e 🗔 🔹			
Registers # Cine	an an ibiy	location 0x44, the base					4 8
Register Value +	1 BlockMove.am	of the stack.					* *
B Carrent 0 </td <td>ARE B AVE S ACCV I HOCV I BCV I BCV I BL 7 LOOD B I HOCV I</td> <td>lockMove,CODE,READWRITE p,Base 0,fUsAB 1,f0xCD r,f0xDE est AOD p!,(r0,r1,lr) 0,f0x11 1,f0x22 14,f0x2 14,f0</td> <td>/make readwrite be /point to the base /dummy value for i /dummy value for i /dummy value for i /sall Test /stay beze /save r0, r1, ir c /let's do somethin /let's do somethin /let's change the /add r0 and z1 and /pull r0, r1, ir c</td> <td>cause we have to o of the stack of iink register, i on the stack of pointless of else pointles link register i put the result off the stack</td> <td>ne stack 1 14 :s</td> <td>n this area</td> <td></td>	ARE B AVE S ACCV I HOCV I BCV I BCV I BL 7 LOOD B I HOCV I	lockMove,CODE,READWRITE p,Base 0,fUsAB 1,f0xCD r,f0xDE est AOD p!,(r0,r1,lr) 0,f0x11 1,f0x22 14,f0x2 14,f0	/make readwrite be /point to the base /dummy value for i /dummy value for i /dummy value for i /sall Test /stay beze /save r0, r1, ir c /let's do somethin /let's do somethin /let's change the /add r0 and z1 and /pull r0, r1, ir c	cause we have to o of the stack of iink register, i on the stack of pointless of else pointles link register i put the result off the stack	ne stack 1 14 :s	n this area	
+ CPSR 0x00000003	DCD Rx	89ABCDEF,0,0,0,0x123456	Heistack area				-
* Use/System	Base DCB 0x	лллллл	;stack base and du	mmy data			10
Proset BRegisters	Here's the initi	al					2
Memory 1	link register						4
Address 0	lilik register.						1
x000000000; EJ ST D0 3C EJ A0 00 AB 3 x00000030; EJ AS CD EF 00 00 00 00 x00000050; D0 00 00 00 00 00 00 00 x00000050; D0 00 00 00 00 00 00 00	E3 A0 55 CD E3 A0 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00	E0 DE EB 00 00 00 EA FF FF FE 00 00 12 34 56 78 AA AA AA AA 00 00 00 00 00 00 00 00 00 00 00 00 00	E9 JD 40 03 E3 A0 00 13 00 00 00 00 00 00 00 00 00 00 00 00 00	E3 A0 10 22 E3 A0 0 00 00 00 00 00 00 00 0 00 00 00 00 0	ED 22 ED 80 5 00 00 00 00 0 00 00 00 00 0	00 00 00 00 00 00 00 00 00 00 00 00 00	03
			Simula	ztion.	t1: 0,00000000 s	ec Li6 Cil	

The next snapshot shows the state immediately before the branch with link instruction.

The next snapshot shows the state after we have called the subroutine and executed the first instruction.



Ele Edit Vew	ereject Flas	h Debu	igeBoo	k\Geng ipheras	ageWork Tools	Sook\Examp SVC5 Wind	es.uvpro	- µVis	Hon4																	-
0000	1 4 15 1	i I		1 n	10.00	建建用 1	1 10							0.0	9.4		•	5								
新旧の一行 legisters	[0 ¹ 11 1 0	0 2 9 1	Regi cont	ister r ains r	<mark>3 now</mark> 0 + r1.	- #	• 🛛 •	3 + 1	*•																	
Register	Value	-/	1.1	Block	Move.as	275																				• :
E Current RD R1 R2 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13(SP) R14(LR) R15(SP) R14(LR)	0x000000000000000000000000000000000000		01 02 03 04 05 06 07 08 07 08 07 08 07 08 07 08 07 01 11 12 13 14	Loop Test	AREA ADR MOV MOV BOV BL BL STMED MOV MOV MOV MOV ADD LDMED	BlockMov #p.Base r0,#DxAF r1.#6xC1 lr,#6xC1 rest Loop sp!,(r0, r0,#6x11 r1,#0x22 r14,#0x2 r3,r0,r1 sp!,(r0,	r1, lr)	S, REA	LINNEI	TE	/mak /dum	e re ny v ny v l Te r lo r s d r s d r s d r s d r s d	edwi e the elue elue et est est est est est est est est est	ite be be for for for for for for for for for for	bec ise r ri r li r li ning ing ing ing r of	ause of t nk i the pol ink put t ti	the regi the the the the the	hav ster ster ess dint free took	e t. k , r les r ult	ne s 14 s 1n	Hero dum link	in e's t my reg	thi he in valu ister	nitia ie of	the	
+ CPSR + SPSR Use(System Football	0x000000003 0x000000000	•	15 15 17 18	Base	DCD (DCD (END	іх99АВСОВ Іхдалала	ε,0,0, Λ	0,0x	1234	5678	;sta ;sta	ck a ck b	194 130	and	dua	пу (iata								0	ب
Aerony 1																										4
Address 0		-																							1	8
000000000: E2 00000000: 89 00000060: 00	EF DO 3C E3 AS CD EF 00 00 00 00 00 00 00 00 00	A0 00 00 00 00 00 00 00	AB E3 D0 00 D0 00	A0 15 00 00 00 00	CD ES 2 A5 00 0 00 00 0		00 00 00 00 00 00	AA 00 00 00 00 00	FF FF AA AA 00 00 00 00	FE 8	41 8 00 0 00 0	40 03 00 00 00 00	ES 3 00 0 00 0	00 00 00 00	11 1 00 0 00 0	13 A0	10 1 00 0 00 0	22 E3 10 00 00 00	A0 00 00	ED 22 00 00 00 00	ED 8 00 0 00 0	00 0 00 0 00 0	01 E 00 0 00 0	8 8D 0 00 10 00 0 00	80 0 00 0 00 0	0.000
														Sin	iulati	pri.			1	11:0,0	000000	00 580	373	1.144	D:C	

The next snapshot shows the state immediately before we execute the last subroutine instruction and return.

The final final shows the state after we have executed the last instruction in the subroutine and have returned to the calling program.

C\Users\Alan	Core7\Desktop\Ceng	ageBoo	k\Geng	ageWork	Sook\Example8		uVision4															-	×	
Bie Edit View	Broject Flash Deb	ug Pag	pherais	Tools	SVC5 Window	Нер			_		_	_		_	-	-								
0040	1309-1	1		2 21	定定用 (1)	10				10	6	0.0	103	14										
	TRACK AND AND PA	1221 -	-	110.000	THE OWNER AND A	-	NA LANC	ST	1000	1		1.1	-	-										
en la un n		1381-10		H1+1-		19.9	-1 × -	1.1																
Registers	A 14	Disaste	embly.																				4	2
Register	Value +	1 1	Block	Move.as	1275																		*	×
 Current Part 1 Part 1	Control Control Contro Control Control Control Contro	03 02 03 04 05 06 \$07 08 05 10 11 12 13	koop Test	AREA ADR MOV MOV MOV BL B STMFD MOV MOV ADD LONFD	<pre>BlockMove sp.Base r0.#0xAB r1.#0xCD lr.#0xDE Test Loop sp!,(r0.r r0.#0x11 r1.#0x22 r3.r0.r1 sp',(r0.r r0.#0x11</pre>	, CODE, 1 1, lr;	READWR	ITE	/make /poin /dumm /dumm /dumm /call /stay /stay /let' /let' /let' /iet'	reach t to t y valu y valu rest here r0, n s do n s do n s do s s chan t0 and	rite the f the f t	r bec ase r ri r li r li hing hing he l and	ause of : nk : the poi ink put	se he i egii sta sta regi the	hav Ftac ster ster int int int ser	e t k , : les s ult	ne : 14 s	r3	t in	th.	2 45	Tex		
	0x00000000 0x00000000 x000000000 x000000	15 15 17 18 •	Base	DCD DCD END CD EX 0 0 00 00	0x99ABCDEF 0xAAAAAAAA ab 80 CK 88 00 00 CD 00 00 00 00 00	, 0, 0, 0, , 0, 0, 0, 00 00 00	0x123	45678 FF FE E BA AA 0 00 00 0	; stac ; stac ; stac ; stac ; stac ; stac	t ares t bars t bars t bars	a and a and a a a a a a a a a a a a	(dum 0 11 1 0 00 (my 0 E3 A0 50 00	10 2 00 0	2 83	A0 00 00	ED 2 00 0	2 20 00 00 00	80 01 00 01	00 00	E3 80 00 00 00 00	00 1 00 1	* *	The second se
#000000965; DD	00 00 00 00 00 00	00 00	00 00	00 00	00 00 00 00	00 00 00	00 00	00.00.0	0 00 0	0 00 00	00.0	0.0.0	00 00	00.0	0 00	0.0	00 0	0.00	00 00	00	00 00	00 0	00	
		_	_	_		_	_	_	_	_	3	mujetk	211.	_	_		115.0%	0000	JUU SR	S	11/0	1	-	l

V 5.0

PASSING A PARAMETER TO A SUBROUTINE

When you call a subroutine, you often have to pass parameters to the subroutine. In a high-level language you might call subroutine XYZ with parameters P and Q by XYZ (P, Q). In a low-level language, you can push the parameters on the stack immediately before calling the subroutine. Of course, you don't have to pass parameters via the stack; for example, if there are a very few, you can transfer them via registers.

Consider the following example where we have a very simple subroutine that adds two numbers P and Q and returns their result S = P + Q. Using pseudocode, we can write the following sequence of actions that describes the passing of the two parameters and the receiving of the result.

Push P Push Q Call ADD Pull S Adjust the stack

We push the two parameters on the stack and call the subroutine. The subroutine reads the two parameters off the top of the stack, and replaces one by the result. Note that we have to adjust the stack to take account of the fact that we have pushed two parameters but pulled only one. The stack must always be balanced with equal numbers of push and pull operations.

The next diagram shows the effect of pushing a parameter on the stack before calling a subroutine. State (a) demonstrates the situation immediately before the subroutine is called. State (b) shows the situation in which both parameters have been pushed. State (c) shows the situation in which a subroutine has been called and the return address is saved on the stack (typical of CISC processors).







As you can see, the stack grows as parameters are pushed and the subroutine called. Then the stack declines as a return made and the two items on the stack removed. Now, let's look at this process in detail using an ARM processor.

USING THE STACK - AN ARM EXAMPLE

The following code sets up an environment and carries out the actions we have described.

	AREA	ParamTest,CODE,READWRITE	; make readwrite because of the stack
	ADR	sp ,Base	; point to the base of the stack
	MOV	r0 ,#0xAB	; dummy value for P in r0
	MOV	r1 ,#0xCD	; dummy value for Q in r1
	STR	r0,[sp,#-4]!	; push P
	STR	r1,[sp,#-4]!	; push Q
	BL	ADDR	; call the adder
	LDR	r2 ,[sp],#4	; pull S off the stack
	ADD	sp , sp, #4	; adjust the stack pointer
Loop	В	Loop	; park here
ADDR	STR	lr,[sp,#-4]!	; push the link register on the stack
	LDR	r5 ,[sp,#8]	; get P (buried under the return address and Q)
	LDR	r6 ,[sp,#4]	; get q (buried under the return address)
	ADD	r5, r5 , r6	; do the addition
	STR	r5 ,[sp,#4]	; save result on the stack under return address (overwrite Q)
	LDR	pc, [sp],#4	; pull return address off the stack
	DCD	0,0,0,0,0	; stack area
Base	DCD END	ΟΧΑΑΑΑΑΑΑ	; stack base and dummy data as marker

The following snapshot demonstrates the situation when the program has been loaded.

Registers		a ×	1	ParamTesti au	m			
Recenter	Viela	-	01	ARFA	FaranTesti, CODE, READWRITE		anse of the stark	
Register □ Current ■ R0 #1 ■ R2 #3 ■ R4 #3 ■ R4 #6 ■ R5 #6 ■ R7 #8 ■ R9 #10 ■ R12 #13 (SP) ■ R14 (LR) #15 (PC) ■ CPSR ■ SPSR ■ User/System ■ Fast Interupt	Velue 0x90000000 0x90000000 0x90000000 0x0000000 0x00000000 0x00000000 0x000000000 0x000000000 0x00000000000000000000000000000000000		07 02 04 05 06 07 09 10 11 12 13 14 15 16 17 18 19 20 21 10 11 15 15 16 17 18 19 20 10 10 10 10 10 10 10 10 10 1	AREA ADB MOV MOV STB STB BL LDB ADD LCOP B ADD STB LDB LDB ADD STB LDB LDB ENC ENC	<pre>ParanTest1, CODE, READWRITE sp,Base r0, #0xAB r1, #0xCD r0, [sp,#-4]! ADDS r2, [sp,#-4]! ADDS r2, [sp],#4 sp,sp,#4 Loop lr, [sp,#4] r5, [sp,#4] r6, [sp,#4] r6, [sp,#4] pd, [sp],#4 0, 0, 0, 0, 0 0xAAAAAAAA</pre>	make read/write bec point to the base of pdummy value for P i pdumny value for Q i push Q point Q point Q point C point S off the stack park here push link register park here push link register park here push link register park defend the addition raeve result on the spoll return address pstack area pstack base and dumm	ause of the stack f the stack n r0 n r1 k on the stack the return address a the return address) stack under return ad , working registers o y data as marker	nd D) dress Ef stack
Project Reg	isters	1	11.1	-				2
Memory 1			-					9
Address 0								21
x00000000 : E x00000024: E x0000004: 0 x0000006: 0 x000006: 0 x000006: 0 x000006: 0 x000006: 0 x000006: 0 x0000006: 0 x000000000 x000000000 x0000000000 x00000000	2 8F D0 48 5 2D E0 04 0 00 00 00 0 00 00 00 0 00 00 00 1 00 100 1	E3 3 E5 5 00 0 00 0	10 00 9D 50 90 00 90 00	AB E3 A0 1 08 <u>E5 90 6</u> 00 <u>AA AA A</u> 00 00 00 00	0 CD ES 2D 00 04 ES 2D 10 0 0 04 E0 E5 50 06 ES ED 50 0 A 2A 00 00 00 00 00 00 00 00 00 00 0 00 00 00 00 00 00 00 00 00 00 This is toward stack i	M Em OG OO OZ E4 9D 2 M E4 9D PO 04 00 <t< td=""><td>20 04 E2 BD D0 04 EA F 20 00 00 00 00 00 00 00 20 00<td>F FF FE 0 00 00 0 00 00 0 00 00 tti 0.000000 row up on the</td></td></t<>	20 04 E2 BD D0 04 EA F 20 00 00 00 00 00 00 00 20 00 <td>F FF FE 0 00 00 0 00 00 0 00 00 tti 0.000000 row up on the</td>	F FF FE 0 00 00 0 00 00 0 00 00 tti 0.000000 row up on the

The first five lines set up the stack pointer, put some data (the parameters P and Q) into registers r0 and r1 and then push the parameters on the stack using pre-indexing with auto decrementing; that is, the stack pointer is moved up by one word (4 bytes) and then the data stored at that location.

ADR	sp, Base	; point to the base of the stack
MOV	r0, #0xAB	; dummy value for P in r0
MOV	r1, #0xCD	; dummy value for Q in r1
STR	r0,[sp,#-4]!	; push P
STR	r1,[sp,#-4]!	; push Q

The following snapshot demonstrates the situation before calling the subroutine (i.e., we are about to execute the branch with link instruction).



on the stack

The snapshot of the system below shows situation in the subroutine after reading the two parameters and pushing the return address. We have called a subroutine and loaded r14, the link register, with the return address, and then executed the following code:

ADDR STR	lr,[sp,#-4] !	; push the link register on the stack
LDR	r5 ,[sp,#8]	; get P (buried under the return address and Q)
LDR	r6 ,[sp,#4]	; get Q (buried under the return address)

This code first pushes the link register on the stack and then reads the two parameters off the stack. You will see that registers r5 and r6 contain the same parameters are r0 and r1, and that the contents of the link register are now the topmost element on the stack.

E:\Cengagelloo	k\GengageWork	Book	Paran	nTesta	vproj -	pVisio	n4																		-15	IX
Ele Edit View	Project Flash	Det	bug F	Periphen	ds Ic	ols §	vcs w	indow	Help																	
1 🗋 🧉 🕼 🕼 🖉	A LEAST				12.7	1.1.1	深深	11=11	1				3	• []	10	1		0	0	•	1		5			
125 00 7	0 0 0	÷ 1	10	6		3+0	•	- # -			20	• 23														
Registers	4	×	1	Paran	TestL	sm		-																		×
Register	Value	•	01		AREJ	Fai	canTes	t1,00	DE,R	CADWRI	TE ;	make	1.10	ad/s	(hit)	e be	1ceu	8 R -	of t	cha	ata	168				-
E Current	1000000000		02		ADB	sp	Base				1	poir	12 2	o th	te bi	1,5 E	20	the	ste	ack.						-
- R0	0x000000AB		03		MOV	r0	#0xAB				7	duna	Q, In	alue	: fo:	r P	in .	20								
- F1	0x00000CD		04		MOV	r1	FOXCD				1	dum	th as	alue	10	r 9	10	<i>r</i> 1								
R2	0×00000000		05		STH	ru	(sp. #	-411			1	pusi	P													
-R3	0x00000000		00		STR	11	(sp.#	-a[:			1	pust	2	100	1.0											
H4	0x00000000		00		1 TVD	-1	feel	÷4			1	CHI	C.D.	E. 80	toer.	1	in the									
170	0-00000000		100		ADD	800	an. #4					- aid-ia		- 1.4	at a	-2	C.A									
-97	0x0000000		10	Loop	B	Los	DD					mari	he	200	a La	1										
- 88	0+00000000		11				- P					part														
- 89	0x00000000		12	ADDR	STR	11	[sp. #				1	pus?	If	nk z	regi	ster	00	Th.	+ 5	taci	÷					
- R10	0x00000000		13		LDR	r5	Isp, #	81				get	P fi	buri	ied :	unde	r t	he	reto	irn	add	iress	and	01		
-R11	0x00000000		14		LDB	r6	[sp, #	41			-	get	9 0	buri	ied i	inde	r t	he	retu	177	edd	iress	()			
-R12	0x00000000		€)15		ADD	r5.	r5, r6				4	do 1	ihe i	eddi	tio	n										
R13 (SP)	0x00000044		16		STR	r5	[sp, #	41			3	dev:	10	sult	100	the	1 25	ack	une	der.	ret	0.50	addr	888		
-R14 (LF)	0x00000018		17		LDB	pe	[sp],	#4			-	pull	ret	1227	i ada	ires	18,	For	king	t n	egi.	ters	622	aba	cà .	
R15 (PC)	0x00000030		-18																							
III CPSR	0x00000D3		19	and the second	DCD	0,0	,0,0,0				1	star	ik al	rea												
III SPSR	0x00000000		20	Base	DCD	0xA	1999999	A			3	stac	the Da	d.se	and	dun	my.	dat	a a:	s m	erke	5.E'				
E User/System			21		ENC																					
11 Fast Interupt																										-
I Project Req	picters	1	1																						-2	ſ
Memory 1																									3	×
Address 0	1																								1	-
0x00000001 E	2 8F D0 48	K3 A	0 00	AB ES	A0 .	10 CD	E5 21	00 0	14 ES	2D 10	0 04	22	0 0	0.03	2 84	9D	20	04	E2	6D .	DO I	04 E2	L FF	FF 2	£	
0x00000024; E	5 2D E0 04	£5 9	D 50	08 E5	9D	60 04	E0 85	50 0	06 E5	6D 50	0 04	£4	9D F	0.0	4 00	00	00	00	00	00	00 1	00 00		00 1	.0	
0x00000048: 0	0 00 00 CD	00 00	0 00	AB AA	AA I	hh hh	00 00	00 0	00 00	00 00	0 00	00 (0 00	0 0	0 00	00	00	00	00	00	00	00 00	0	00 0	0	1.1
0x0000006C: 0	00 00 00 00	00 0	0 00	00 00	00	00.00	00 00	00 0	0 00	00.00	0.00	00	0.0	0.0	0 00	00	00	00	00	00	00.1	00.00	00	00 0	0	
Call Stack 🛄	Memory 1																					/				
																		. Sir	nulat	ion		/		11:0.00	00000	10
															Tł	ne lin	nk r	egis	ster,	/	7					
															r1	4, sa	aved	l on	the		1					
															sta	ack					1					

The next snapshot shows the situation immediately before the return from subroutine. We have just executed

ADD	r5, r5,r6	;do the addition
STR	r5 ,[sp,#4]	; save result on the stack under return address

These instructions perform the addition of the parameters in registers r5 and r6 and then store the result at [sp] + 4 which is one word below the top of the stack; that is, the location of parameter Q. The following memory map shows that Q (in memory) has changed from 0x000000CD to 0x00000178.



Version 1 [WORKBOOK FOR COMPUTER ORGANIZATION AND ARCHITECTURE: THEME AND VARIATIONS]

The final snapshot shows the situation at the end of the program when we have executed the following code.

	LDR	pc, [sp],#4	; pull return address off the stack (last line of subroutine)
	LDR ADD	r2 ,[sp],#4 sp ,sp,#4	; pull S off the stack (first operation after the subroutine) ; adjust the stack pointer
Loop	В	Loop	; park here

Note that this code is rewritten in execution order rather than program order; that is, the first line is the last operation in the subroutine and the second line is the first instruction at the return point.

A return is made by pulling the link register off the stack and putting it in the program counter. In the calling routine, the top of the stack is pulled (i.e., the result) and put in r2. Finally, the stack pointer is incremented by 4 to restore it to its original value.



IMPROVING THE CODE

Few programmers would write the code we used in the previous example. A more reasonable approach is:

	AREA	ParamTest1,CODE,READWRITE	; make readwrite because we locate the stack in this area
	ADR	sp ,Base	; point to the base of the stack
	MOV	r0, #0xAB	; dummy value for P in r0
	MOV	r1, #0xCD	; dummy value for Q in r1
	STMFD	sp!, {r0, r1}	; push P and Q
	BL	ADDR	; call the addition subroutine
	LDMFD	sp!,{r0,r2}	; pull S and P off the stack
Loop	В	Loop	; park here
ADDR	STMFD	sp! , {r5, r6, lr}	; push the link register and working registers
	LDR	r5 ,[sp,#16]	; get P (buried under the return address and Q)
	LDR	r6 ,[sp,#12]	; get Q (buried under the return address)
	ADD	r5 ,r5,r6	; do the addition
	STR	r5,[sp,#16]	; save result on the stack under the return address
	LDMFD	sp!,{ r5,r6,pc }	; pull return address and working registers
	DCD (<pre>DxFFFFFFFF,0,0,0,0,0</pre>	; stack area
Base	DCD (END	АААААААА	; stack base and dummy data

We need to look at some of the features of this program in greater detail.



67 | P a g e

Now we can execute this code in debug mode and trace its execution. The next snapshot shows the situation after the code has been loaded and simulation is about to begin.

E:\Ceogagefloo	k\CengageWorkbo	ok\Para	nstest	lod uvpre	nj - pWsian4				_								-101
Eile Edit View	Project Flash 1	Depog	Pepiphe	sais Ioc	als SVCS Window Help												
0040	- 00				入 建汞田市 🥶			101	0			5.	2				
ST BOT	• (P () *) ↔		a			· 1	R . 2		-			-	-				
legisters	0 ×		Paran	TestMod	Lasm												+ >
Register	Value .	11		AREA	ParamIest1, CODE, REA	DWRITE	make read	iviite b	ecauar	ve 1	poate	the	ataci	in	this a	rea	
Current	2	207	T	ADR	sp,Base		rpoint to	the bas	# 11E -	the st	ack						
RD	Dx00000000	.01	10	NOV	r0,#0xAB		rdimey val	ue for	Pisi	0							
-R1	Gh00000000	04		MOV	r1,#0xCD		adutny val	ter for	Q 10 1	11							
R2	0x00000000	.05		STMPD	sp1.(e0.e1)		rpush F an	nd g	1000								
- R3	0x00000000	05		81.	ADDR		reall the	additio	n sub	Pootin	0						
R4	0x00000000	07		LDMPD	sp1, (r0, r2)		rpull 5 at	ut P off	the .	Woma'r							
RS	0x00000000	80	Loop	B	Loop		mark here										
RS	0x0000000	09															
87	0x00000000	10	ADDR	STMPD	ap1.(r5.r6.1r)		znuch the	tink m	diste	bns -	enekis		tieti	175			
- RR	0-00000000	11		LDR	r5, [ap. #16]		aget P (h)	tried un	der ti	te net	orn ac	idres	-	1 179			
69	0-00000000	12		LDR	r6. [ep. #12]		reat 0 /bs	reight un	der ri	te yet	Den ad	(drag	1.00				
- R10	0x00000000	17		ADD	15.15.16		rdo the at	Hition	10.01				124				
PIT	2x02000000	14		STR	r5. [sp. \$16]		-Aave -aat	IF ON P	he at	inter inter	-		-	and do	et. er let		
1212	0+00000000	15		LDMPD	ant. (r5, r6, nc)		erest11 made	the address		d ber	vine -		harr				
D11 (CD)	2-0000000	16					spost tott	ITTE GOVER	000 81	10 9 9 9 1		Cy as	1001.0				
Pita (ar)	0-00000000	17		DCD	OWFFFFFFFF 0 D 0	0.0	serack and										
D35 (00)	3-0000000	10	Bare	nen	Ownandababab		antack bas	a sed d	-	1							
HIDDO	5-0000000	10	nenc	P'NTT	- Anthenenenen		TATATA THE	a and a		101.4							
in Gran	akovovou (*)	1.7	1	Plat.													
E Project E Rec	oters .	1141	-													_	1
temory 1																	.0
Address 0																	15
*00000001 E	2 87 D0 44 E3	A0 00	AB E	3 80 1	CD E9 2D 00 03 EB	00.00 (01 EE 5D 00	05 KA	FT TT	FE ES	2D 40	60	g5 90	50.3	10 25	9D 60	0C
x000000281 E	0 85 55 06 E5	8D 50	10 E	B BD BI	0 60 FF FF FF FF 00	00 00 0	00 00 00 00	00 00	00 00	00 00	00 00	00	00 00	00	AA DO	AA AA	AA
x000000501 0	00 00 00 00 00	00 00	00 0	0 00 0	00 00 00 00 00 00 00	00 00 (00 00 00 00	00 00	00 00	00 00	00 00	00	00 00	00	00 00	00 00	00
										Semi	noite			11:00	00000000	sec	

The next snapshot shows the situation after the code has been executed up to the beginning of the subroutine.

E:\CengageRoo	k\CengageWorkbo	ok\Paran	testHod uvp	roj - pVisiond	x lot
Eile Edit View	Project Flash	Debog P	egiphenais Io	ols SVCS Window Help	
	100		P. 0.0	1月建建用作 🥶	- 2 · Q • O 🚓 💷 - 🔧
1第 BO T	9 (P () 4) 🔶	00			8. 3
Registers	0.00		ParamTestMo	d.asm	+ x
Register	Value	01	ARZA	PeramTest1, CODE, READWRITE	/make readvrite because we locate the stack in this area
Current	Ware constant	02	ADB	sp,Base	spoint to the base of the stack
RD	BA0000004B	03	MOV	r0, #0xAB	rdummy value for P in r0
R1	0x000000CD	04	MOV	F1, #0xCD	adunny value for g in ri
10	0x0000000	05	STREET	ADOD	rpush e and y
P.0	00000000	05	1 DMPT	an! (rf) r2)	roull f and B off the stark
	0x00000000	86	LOOD B	Loon	mark here
RS	0x00000000	09			- Para sere
- 87	Dx00000000	10	ADDR STMPL	ap!,(r5,r6,1r)	ypush the link redister and working redisters
	0+00000000 -	11	LDR	r5,[ap,#16]	(get P (buried under the return address and Q)
R9	0x00000000	12	LDR	r6,[sp.#12]	rget Q (buried under the return address)
R10	0x00000000	13	ADD	r5, r5, r6	ado the addition
R11	0x000000x0	14	STR	r5,[sp,#16]	reave result on the stack under the return address
R12	0x00000000	15	LDMPI	sp1, (r5, r6, pc)	spull return address and working registers
R13 (SP)	0x00000044	16	0.00		The Alexandream and the A
R14 (LR)	0x00000014	17	DCD	OXFFFFFFFF, 0, 0, 0, 0, 0	VSTack area
H15(PC)	0-0000000	10	page Net	OXANAAAAAA	raters mape and minuty data
I In Cras	activities in the	1.7	Line		
and a second second		<u>u a</u>	-		
Memory 1			100		
Address 0					11
0x000000001 E	2 87 D0 44 E3	A0 00	AB E3 A0 1	0 CD E9 2D 00 03 EB 00 00	01 ES BD 00 05 EA FF FF FE ES 2D 40 60 E5 9D 50 10 E5 9D 60 0C
0x000000281 E	0 85 56 06 E5	SD SD	10 E8 BD 8	0 60 FF FF FF FF 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000501 0	0 00 00 00 00	00 00	00 00 00 0	0 00 00 00 00 00 00 00 00 00	
					Simulation 11: 0.0000000 sec
					Hore are peremeters P and O on the steels
					Here are parameters F and Q on the stack.
					Note that there are above 0xAAAAAAAA
					that we have used as a marker to show the
					base of the stack
					base of the stack.

The next snapshot shows the situation immediately before the subroutine return.

E:\CengageBook	\CengageWork	book\Para	instestMod uvpr	oj - pMsiani			
le Edit View	Project Flash	Qebog	Periphenais Io	ols SVCS Window Help			
040	40.0		1000	N 3 R R II 10 3	- 2 - 2	00000	
	Th. 1. 111	- min			Sel S		
গ এন প	u. 11 11 1	111 198		- Constraint a series of the	02.02		
piters	. ņ .	× .	ParamTestMod	Lasm			+ ×
gister	Value	• 02	ADR	sp, Dase	spoint to the base	of the stack	-
Current		03	MOM	r0, #OxAB	rdunny value for P	in r0	
RD	BADD0000AB	04	VOM	r1, #0xCD	rdummy value for Q	17 11	
R1	0x000000CD	05	STMPD	sp!, (r0, r1)	/push P and Q		
R2	0+00000000	06	BL	ADDR	reall the addition	subroutine	
R3	0x00000000	07	(Deta)	sp1, (r0,r2)	spull 5 and P off t	he stack	
R4	5x00000000	08	Loop B	Loop	spark hars		
RS	0x00000178	89	Contration and				
RIS	0x000000AB	10	ADDR STMPD	sp!, (r5, r6, 1r)	spush the link regi	ater and working registers	
- B7	Dx00000000	11	LDR	r5,[sp,#16]	/get F (Duried unde	r the return address and Q)	
- R#	0x00000000 -	- 12	LDR	r6,[sp,#12]	iget 2 (buried unde	r the return address)	
RB	0x00000000		ADD	r5, r5, r6	rdo the addition		
R10	0x00000000	14	STB	r5,[sp,#16]	coave result on the	stack under the return addres	4
R11	0x00000000	\$215	LDMPP	sp!,(r5,r6,pc)	spull seturn addres	a and working registers	
R12	0x00000000	16					
R13 (SP)	Dx00000038	17	DCD	OXEFFFFFFF, 0, 0, 0, 0, 0	/stack area		
R14 (LR)	0x00000014	18	Base DCD	OXRAAAAAAA	/stack hase and dum	my data	
R15 (PC)	0x0000030	19	END				5.00
IE CPSR	0x00000003	- 20					T
haject 📰 Regi	iters	1.1					15
iony 1							9 ×
ess 0			_				6 - C
00000001 82	87 D0 44 K	3 80 00	AN E3 A0 1	0 CD K9 2D 00 03 KB 00 00	01 KE BD 00 05 KA FF	FF FE ES 2D 40 60 E5 9D 50 10	\$5 9D 60 0C
00000281 E0	85 55 06 E	5 8D 50	10 E8 BD 8	0 60 FF FF FF FF DO DO DO	00 00 00 00 00 00 00	OR 14 90 00 00 A1 00 00 D1 78	AA AA AA AA
00000501 00	00 00 00 0	0 00 00	00 00 00 0	0 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00	00 00 00 00
						Simulation 11:0.00	00000 sec U15 1
					\sim		
					\backslash		\backslash
				Here are registers r5 r6	and r14 that we	This is the result 0x00	000178 that has h
				fiere are registers 15, 10,	and 114 that we	This is the result, 0x00	oboli 70, that has b
				1 1 1 1			

The final snapshot shows the situation at the end of the program after the data has been pulled off the stack.



PASSING A PARAMETER BY ITS ADDRESS

Some languages let you pass parameters by reference rather than by value; that is, you send the *address* of the parameter to a subroutine. The 68K processor has a *push effective address* instruction, PEA that pushes a 32-bit address on the stack. ARM programmers have to use conventional memory store instructions.

When you retrieve a parameter passed by reference (address), you have to pull the address off the stack (or read it from the stack) and then access the parameter by means of address register indirect addressing. Consider the following fragment of code that pushes an address (initially in register r0), call a subroutine, and then retrieves the actual parameter (i.e., its *value*) in the subroutine..

	STR BL	r0 ,[sp,#-4]! ABC	; Push the address of parameter P on the stack (address is in r0) ; Call subroutine ABC and save the link register
ABC	STR	lr,[sp,#-4]!	; Save the return address on the stack
	LDR	r1 , [sp, #4]	; Read the address of parameter P under the return address
	LDR	r2 ,[r1]	; Get the value of parameter P
	LDMFD	sp!, {pc}	; Return by loading the PC with the return address from the stack

Retrieving a parameter by reference is a two-step operation. The first part is to get the parameter's address, and the second part is to get the value pointed at by that address. In this case we first load the address of P using LDR r1, [sp, #4] to get the *address* of P in r1 and then use LDR r2, [r1] to get the *value* of P in r2. We have put these two lines in blue to highlight their importance.

Let's use this code in an actual program. Below, we use subroutine ABC to perform P + 1. The effect of this program should be to add 1 to P's initial value 0x12345678 to give 0x12345679 in the memory location defined as P. Since there are 11 instructions before this location, the address of P is 0x0000002C (i.e., 11×4 expressed in hexadecimal).

	AREA	PassByRef,CODE,READWRITE	; Make readwrite because we locate the stack in this area
	ADR	sp ,Base	; Point to the base of the stack
	ADR	r0 , P	; Load r0 with the address of parameter P
	STR	r0,[sp,#-4]!	; Push the address of parameter P on the stack (address is in r0)
	BL	ABC	; Call subroutine ABC and save the link register
Moi	В	Moi	; Infinite loop to end the program
ABC	STR	lr,[sp,#-4] !	; Save the return address on the stack
	LDR	r1 ,[sp,#4]	; Read the address of parameter P under the return address
	LDR	r2, [r1]	; Get the value of parameter P
	ADD	r2 , r2, #1	; Add 1 to P
	STR	r2,[r1]	; Save the parameter in the calling environment
	LDMFD	sp!, { pc }	; Return by loading the PC with the return address from the stack
P	DCD	0x12345678	; Location of parameter P and its value
	DCD	0xFFFFFFFF,0,0,0,0,0	; Stack area
Base	DCD END	Охааааааа	; Stack base and dummy data

The first instruction, ADR **sp**, Base, loads the stack pointer with the initial base of the stack, and the second instruction, ADR **r0**, P, loads r0 with the address of P. It is important to stress here that we are loading the address of P (0x0000002C) and not it's value (0x12345678).

The following snapshot demonstrates the situation immediately after the program has been loaded. We've highlighted the data area and the stack.

Cengrantiook	CengageWorkboo	ok (Pasa)	uddress.	evproj - p	Mauni		
Elle Edit View	fraject Figin D	Jepup 8	Pepphera	n Jook	SAC2 Window Helb		
Dena	1 - n5 -		17	12 13 7	**********	· · · · · · · · · · · · · · · · · · ·	
27 140 71	0-0-11 -	DA		3.0.	3	· ·	
Registers	0 ×		Panally	Ref.asm			* X
Regular	Webum .	01	-	114/2011			
Current		02		AREA	PassByRef, CODE, READWRITE	/make readwrite because we locate the stack in this area.	-
- B0	0x00000000	5 83	1	ADB	sp,Base	spoint to the pase of the stack	
	0+00000000	04		ADB	r0, F	sload s0 with the address of parameter P	
- R2	0x00000000	05		STR	r0, [sp.#-4]]	(Fush the address of parameter P on the stack (address is in r0)	
83	0~00000000	96		BL	ABC	sCall subroutine ABC and save the link redister	
- H4	0+00000000	97	Mol	в	Mo1	rinfinite loop	
- 115	0-20002000	88					
- BE	0x00000000	09	ABC	STR	1r. (ap. 8-91)	Save the return address on the stack	
87	0-00000000	10		1.08	r1. (ap. 25)	Read the address of navameter P under the return address	
	0.00000000	11		LDB	#9. [#1]	Cat the using of manager 2	
- 09	A-000000000	12		ADD	r2, r2, #1	the star of the particular of the star of	
D10	0-2002000	12		670	-2 1-11	there the revealetion is the solution endlangert	
700	0.0000000	12		LINNED	and (not)	There has reaching the Construction and the second se	
711	0.00000000	45		Ladres of	who is the l	Augustin ph roading me so are the larger and as a scale	
F112	0.00000000	10	e	here'	0412045570	Construction of the anticonstruction of the second s	
H(13(5P)	0x0000000	16	E	DUD	VAL2313ETB	Frontion of Dermitian & ending for welle	
PITA (LPS	040000000	17	2000	0.0	VX2FFFFFFFF,0,0,0,0,0	ratice arms	
H15(MG)	0420000000	18	DASE	DUL.	OXANAAAAAA	Jutack base and unuty data	
IN OPSR	0x00000003 +	19		ENL			-
Project. Regi	sters	11					2
Aenory 1							4 ×
Address: 0			-1				10-
0×0000000. E	OF DO 40 TO		20 55	20.00	04 FR 00 00 00 FR FF FF FF	F FE 3D FO DA FE GD 10 DA FE DI 30 00 F1 03 30 01 FE DI 30 00 F0 BD 00 00 13 3	4 55 70
0x00000000		00.00	00.00	00.00	NO DO DO DO DO DO DO DO		0 00 00
avonoonee- o	0.00 00 00 00	00 00	00 00	00 00	AC DO DO DO DO DO DO DO DO		0 00 00 1
The second se		94.94	0.0.0.0	90.04	<u></u>		
STORE DUNK + LINE	- Witmony 1						
/						Simulation It. 0.0000000 sec	VCRU QUALRYE
							1 6 5
E. 1 . 6 1				-		Stack base marker This is the	e value of P
End of code a	na data			The	e space we've	(0x12345	678) at mer
marker				1eft	for the stack	000004C	
				len		location (x0000002C

The next snapshot shows the state of the system up to the start of the subroutine. You can see that r0 contains the address of parameter P (i.e., 0x0000002C). The stack pointer has been moved up from its initial value of 0x00000040 to 0x00000044.


The next snapshot traced execution to the point at which the address of P has been read off the stack into r1, but the value of P has not yet been loaded into r2.



The final snapshot shows the sitiation at the end of the program. The value of P in memory has been updated.



The Stack Frame and Low-Level Support for High-Level Languages

We now look at how a low-level language provides support for local variables in subroutines, and discuss how parameters are passed to and from procedures in greater detail.

In addition to the parameters passed between a subroutine and its calling program, a subroutine sometimes needs local workspace for its temporary variables. Each time the subroutine is called, a new workspace must be assigned to it.

Suppose task A is using a subroutine and workspace has been allocated for use by the subroutine's variables. Assume a task switch takes place while task A is executing the subroutine and task B uses the same subroutine. Clearly, task B must be allocated new workspace for its own variables, if it is not to corrupt task A's variables. The stack provides a convenient mechanism for implementing the dynamic allocation of workspace. This storage allocation is dynamic because it is allocated to variables when they are created and then de-allocated when the variables are no longer required.

Two items closely associated with dynamic storage techniques are the stack frame (SF) and the frame pointer (FP). The stack frame is a region of temporary storage at the top of the current stack. The frame pointer, which is in an address register, points to the bottom of the stack frame. Figure (a) illustrates the state of the stack after a subroutine call and figure (b) illustrates the

stack frame that has been created on top of the subroutine's return address.

A stack frame can exist in several forms. It is, of course, programmer, dependent. Figure (b) shows a stack frame with a stack that grows towards low addresses. Note that, in this example, the frame pointer points to the empty base of the frame above the return address on the stack.





a. The state of the stack immediately after a subroutine call

Let's consider the creation of a simple stack frame as figure (b) above demonstrates. We look at a more realistic example later. First we need to move the stack pointer up by one word to point at the empty base of the frame. We can do this by SUB **sp**, sp, #4. The next step is to make the frame pointer, fp, point at the base of the stack, which we can do with MOV **fp**, sp; that is, we copy the stack pointer into the frame pointer. A stack-frame is then created by moving the stack pointer up by *d* locations at the start of a subroutine. For example, reserving 16 bytes of memory is achieved by executing sub **sp**, sp, #-16. Once the stack frame has been created, local variables can be accessed via the stack pointer and a suitable offset. Consider the following code:

b. The state of the stack after creating a stack frame

		Version 1	[WORKBOOK FOR COMPUTER ORGANIZATION AND ARCHITECTURE: THEME AND VARIATIONS]
AnySub	SUB MOV	sp, sp,#4 fp, sp	; Move the stack pointer up one word past the return address on the stack ; Set up the frame pointer to point to the top of the stack
	SUB	sp, sp , #16	; Move the stack pointer to the top of the stack frame (we'll allocate 16 bytes)
	•		; The subroutine proper (i.e., the code goes here)
	ADD MOV	sp,sp,#20 pc, lr	; Collapse the stack frame (i.e., 16 + 4) ; and return from subroutine

Before a return from subroutine is made, the stack frame must be collapsed by an ADD sp, sp, #20 instruction. This simply moves the stack pointer down. In practice, this code would not be used, because it doesn't preserve the old frame pointer; that is, the frame pointer is *destroyed* by this code.

A better way of implementing a stack frame is to save the old frame pointer on the stack before creating the frame itself; that is,

AnySub	SUB	sp ,sp,#4	; Move the stack pointer up to create space for the old frame pointer
	STR	fp ,[sp]	; Save the old (existing) frame pointer on the stack
	MOV	fp, sp	; Set up the frame pointer to point to the base of the stack
	SUB	sp, sp, # 16	; move the stack pointer to the top of the stack frame
	•		
	•		; The subroutine proper
	•		
	MOV	sp, fp	; Restore the stack pointer and collapse the frame
	LDR	fp ,[sp],#4	; Restore the old (existing) frame pointer on the stack
	ADD	sp,sp,#4	; Move the stack pointer down to point to the return address
	MOV	pc, lr	; and return from subroutine

In practice the code would be more compact with the ARM's facilities (e.g., auto incrementing and decrementing addressing modes) being better used. Consider the following example. In this case consider the following example where a subroutine is called using a BL instruction (branch with link). In this case the return address is not saved on the stack.

	BL	ABCD	; Call subroutine ABCD
			;
			;
ABCD	STR	fp ,[sp,#-4]!	; Save the old frame pointer on the stack (pre-indexing)
	MOV	fp , sp	; Set up the frame pointer to point to the base of the stack
	SUB	sp ,sp,#16	; Move the stack pointer to the top of the stack frame
	•		; The subroutine proper
	•	_	
	MOV	sp, fp	; Restore the stack pointer and collapse the frame
	LDR	fp ,[sp],#4	; Restore the old frame pointer on the stack and post-increment the stack
	MOV	pc,lr	; Return

The following snapshot of the simulator demonstrates this fragment of code in the simulator using some dummy data to keep track of register values.

Cengagefion	k\GengageWorkt	Sook VI.	nample	12.uvpraj	- p/Misian	14	A REAL PROPERTY AND INCOME.	alQi X
file hat gien	Project Fight	Debu	g Per	ipherali]	poka SI	CS Window Help		
	1-10-			四 九		王王田 四 四	· · · · · · · · · · · · · · · · · · ·	
1 0 1 0 W	0 0 0 9	> 1	10.1		a. 🗆	- 3	M - 2 - 1	
Registers		0 X	1	Stackfra	meSimple	Asm.asm		* ×
Register.	Value		01		AREA	StackFraneSimp	1e, CODE, READWRITE /create a stack frame	- 27
Current			02		LDB	lr,=0x12345676	idinary value for Ir	-
- RØ	0x00000000		03		LDR	fr. =0xAAAAAAAA	/dummy value for frame pointer	
- R1	0+00000000		04		ADR	sp, stack	/stack pointer points to base of stack	
- R2	0x00000000		05		BL	ABCD	Call subrouting ABCD	
- R3	Dx00000000		05	Lane -	ADD	r1, r1, #0xEE	/dunny operation	
- R4	0x00000000		07	Again	E	Again	condless loop	
- R5	0x00000000		08				2) - Contraction of the second s	
- 85	0x00000000		09				1	
- R7	0x00000000		10	ABCD	STR	fp,[sp,#-4]!	(Save the old frame puinter on the stack (pre-indexing)	
- F18	0x00000000		11		MOV	fp, sp	Alet up the frame pointer to point to the base of the stack	
- R9	Gx00000000		12		SUB	sp, sp, #16	Move the stack pointer to the top of the stack frame	
- R10	0x00000000		13		#25			
B11	(baaaaaaaa		14		2.1		/The subroutine proper	
- F12	0x00000000		15		1:00			
- R13-(SP)	Dx00000044		16		MOV	sp,fp	(Restore the stack pointer and collapse the frame	
R14 (LR)	0x00000010	- 16	17.		LDB	fp,[sp].#4	Restore the old frame pointer on the stack and post-increment	the stac
R15(PC)	0x00000010		16		MOA	pc, lr	/Beburn	
CPSR	Be00000003		19					
IEI- SPSR	0x00000000		20	a server	DCD	DK0, 0, 0, 0, 0K0	Istack eres	
to User/System		-	21	stack	DCD	ONFEFEEFEE	rstank base and dumny data (marker)	
Project Reg	isters		1.1		. FRE			2
	10					1 2	Simulation et: 0.0000000 sec	#

Up to now, we've demonstrated simple examples of stack frames. The next step is to provide a more realistic (albeit simple) example. This example will demonstrate various aspects of machine-level programming; for example, the use of registers (global and local), the use of temporary storage (stack frames), and parameter passing.

PASSING PARAMETERS TO AND FROM A STACK FRAME

We are going to use a subroutine that is called by pushing the return address on the stack. We pass two parameters to the stack; one by value and one by reference. Let's assume that the stack performs $B = A^2 + B$, where A is passed by reference and B by value.

In this example, we use two registers in the subroutine, r1 and r2, that are saved on the stack at the start of the subroutine by a store multiple registers and then retrieved at the end of the subroutine by a load multiple registers. One register, r0, is a global scratchpad and does not have to be preserved by the subroutine. Finally, we create a stack frame for one variable in the subroutine.

The code for this example is given below. We have created initial dummy values for registers so you can see them when they are saved in memory and used 0xFFFFFFF as the stack base in order to make the stack visible in the memory map.

	AREA	FrameParams, COD	DE, READWRITE
	ADR	sp, Stack	; set up the stack pointer
	LDR	<pre>fp,=0xAAAAAAAA</pre>	; dummy value for fp
	LDR	r1 ,=0x11111111	; dummy value for r1
	LDR	r2 ,=0x22222222	; dummy value for r2
	ADR	r3 , A	; r3 is a pointer to A
	LDR	r4 ,[r3]	; get parameter A
	STR	r4,[sp,#-4]!	; push the value of A on the stack
	ADR	r5, B	; get the address of B
	STR	r5,[sp,#-4]!	; push the address of B on the stack
	BL	SumSq	; call the subroutine
	LSR	r0 ,[r5]	; if it worked, r0 should contain 7
in	В	Again	; parking loop

Aga

SumSq	STMDB STR MOV SUB LDR MUL STR LDR LDR LDR ADD STR MOV LDR LDRIA	<pre>sp!, {r1,r2,lr} fp, [sp,#-4]! fp, sp sp, sp, #4 r1, [fp, #20] r2,r1,r1 r2, [fp, #-4] r2, [fp, #16] r1, [r2] r0, [fp, #-4] r1,r1,r0 r1, [r2] sp, fp fp, [sp], #4 sp!, {r1,r2,pc}</pre>	 ; save registers on the stack ; Save the old frame pointer on the stack (pre-indexing) ; Set up the frame pointer to point to the base of the stack ; Move the stack pointer to the top of the one-word stack frame ; Get the value of A off the stack in r1 ; Square A ; Store the value of A squared in the stack frame ; Get the address of B off the stack in r2 (reuse r2) ; Get the value of A squared in r0 ; Add B to A squared ; Restore the stack pointer and collapse the frame ; Restore the old frame pointer on the stack and post-increment the stack
A	DCD	2	; dummy value for A
В	DCD	3	; dummy value for B
	SPACE	16	; reserve 16 bytes for the stack
Stack	DCD END	Oxfffffff	; dummy data for the base of the stack

The next simulator snapshot shows the simulator window when the program is first loaded.

E:\CeogageBoo	k\GengageWork	Book\Exa	mple13.ovproj	- JiVision4		ad X
ile fait Liew	Project Flash	Deprid	Pegipherais I	ools SAC2 Wandow	Help	-
	1 10 11 11 1		a minet st			2
esistem	0 U 17 4		A 41 0-14			- X
Renater	155614		AUFA	Francisco Corona Ci	ODS DESIDED TTY	
Corned	Value	100	ADR	en. Stack	cast on the stark mothter	
Di	0-00000000	13	1.08	fr. =Oxidiadias	source up the state pointer	
Rt	0x00000000	04	LDR	F1.=0x11111111	rdumny value for rt	
- E2	0+00000000	05	LDR	¥2,=0x222222222	idumay value for rg	
83	0+00000000	05	ADR	r3.A	ard is a pointer to A	
R4	0x00000000	07	LDR	r4, [r3]	Just parameter &	
RS	0x00000000	08	STR	r4. [sp.#-4]!	spush the value of A on the stack	
RIS	0x00000000	09	ADR	r5,B	ight the address of B	
R7	0x00000000	10	STR	r5,[sp,#-4]!	spush the address of B on the stack	
- RB	0x00000000	11	BL	SunSq	rcall the subroutine	
R9	0+00000000	12	LDR	r0,[r5]	wif it worked, r0 should contain 15 or 0xF	
R10	0x00000040	/13	Again B	Again	/parking loop	
R11	0x00000000	14				
R12	0x00000000	15	SumSq STHD	<pre>9 sp!,(r1,r2,1r)</pre>	reave registers on the stack	
R13 (SP)	0x00000000	16	STR	fp,[sp,#-4]:	shave the old frame pointer on the stack (pre-1	ndexing)
R14 (LR)	9x00000000	17	MOV	fp,sp	:Set up the frame pointer to point to the base of	f the steck
R15 (PC)	0x00000000	18	SUB	sp,sp,‡4	sMove the stack pointer to the top of the one-wo	rd stack frame
IT CPSR	0+00000003	19	LDR	r1, [fp,#20]	/Get the value of A off the stack in rl	
i ≝ SPSR	0+00000000	20	MOL	r2, r1, r1	15quare A	
User/System		27	STR	r2,[rp,#-4]	Store the value of A squared in the stack frame	
Fast Interrupt		22	LDR	r2, [fp,#16]	yGet the address of 8 off the stack in r2 (reuse	(22)
interrupt		23	LDR	r1, [r2]	rGet the value of B in r1 (reuse r1)	
Supervisor		2	LDB	r0,[rp,‡-4]	rGet the value of A squared in ro	
Abot		2	ADD	F1, F1, F0	rAdd I to A squared	
Undefined		20	SIR	F1, [F2]	ineturn the result to the calling environment	
Internal	5-0000000	20	100	for family did.	Rescore the stark pointer and collapse the right	IP
PC S	00000000	100	LOWT	ip, [op], et	INCIDENT CAR DIA IFARE POINTEF ON THE SCHOK AND	post increment the state
Protect	Supervisor	30	LUTAL	a sylviciitziper	Jistors registers and return	
Car	0.00000000	11	a non	2	returned and how form &	
	- sorrore	22	8 DCD	3	rdimmit malue For B	
		33	SPAC	2 9.2	reserve 32 bytes for the stack	-
		34	Stack DCD	C×FFFFFFFF	rdummy data for the base of the stack	
Trainet The	untern .	Ŧ	END			التم.
TAURIER	Files	in er				
emory 1		_				
Addreem: 0						
\$:00000000 s	2 SF DO SC 8	15 9F E	O BC ES SF	10 BC 25 9F 20 BC	E2 8F 30 54 E5 93 40 00 E5 2D 40 04 E2 8E 50 4C 1	E5 2D 50 04 E8 00 00 01
0000028; Z	5 95 00 00 5	IA FF I	T FE E9 2D	40 06 E5 ZD B0 04	E1 A0 B0 0D E2 4D D0 04 E5 9B 10 14 E0 02 01 91 0	E5 08 20 04 E5 98 20 10
00000050: E	S 92 10 00 E	CS 18 C	0 04 E0 81	10 00 E5 82 10 00	EL AO DO OB E4 9D BO 04 E8 BD 80 06 20 00 02 02	20 00 00 03 00 00 00 06
00000781	0 00 00 00 00	00 00 0	0 00 00 00	00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 FF FF	AA AA AA AA 11 11 11 11
00000040:2	2 22 22 22 0	00 00 0	0 00 00 00	00 00 00 00 00 00		
00000008: 0	0 00 00 00 0	00 00 0	0 00 00 00	00 00 00 00 00 00		
000000f0: 0	00 00 00 00 0	10 00 0	0 00 00 00	00 00 00 00 00 00		
Call State	a cata: Memory	y1				/
Command						/
0155000000000111					In the second se	11. 0.0000000 at
					1 Januarian	u. owwwww.ec
					/	/
					Here's the base of the stack	Here's literals that we load in
					which will grow upwards	registers initially. Remember
					towards lower addresses	ARM processor can create 2
					towards lower addresses.	Antivi processor can create 3
						literals by storing them in m
						pool of constants and then us
						pointer-based addressing to
						pointer-based addressing to I
						tnem

The next memory map demonstrates the situation immediately before the subroutine call. You can see that registers r1 and r2 have been loaded with the markers 0x11111111 and 0x2222222. Register r4 contains the parameter A (i.e., 2) and register r5 contains the address of parameter B (i.e., 0x00000070).

the y as a ve

The stack pointer, sp or r13, contains the value 0x0000008C and is pointing at the last value pushed on the stack; that is, the address of B. Finally, the frame pointer, contains the marker 0xAAAAAAA.

E:\CengageBook	\GengageWorkI	Debug	ample 13.ovpraj - j	Wision4	dete		
Die Tour Ziem	Erder Hight	Senad	vehibusian Too	R R // //	(B) + 1		
B RO R	10 11 10	575			1		
rgisters	4 ×	1	ParamPassionFra	me asm			• ×
Recipter	Value	m	AREA	FrameParana, CO	DE. BEADWRITE		
Current	1.000	02	ADR	sp, Stack	/set up the stack point	6F	-
RD	0x000000000	113	LDR	fp,=0xAAAAAAAA	rdummy value for rf		
Rt	0x11111111	04	LDR	r1,=0x11111111	rdunny value for rl		
- FI2	0x22222222	05	LDR	¥2,=0x22222222	/dummy value for rg		
- F3	0+0000006C	06	ADR	r3,A	Jr3 is a pointer to A		
R4	0x00000002	17	LDR	r4,[r3]	jget parameter A	The state of	
102	0x00000070	105	316	24 [20] [2 - 3] [spush the address of P	the stack	
105	0x0000000	10	572	r5, fen. 2-411	rough the address of B	on the start	
- DR	0x0000000	B	BL	SunSa	scall the subrochine	PIC LINE STREET	
89	0+00000000	12	LDR	r0, [r5]	wif it worked, r0 should	d contain IF or OwF	
R10	0+00000000	/13	Again B	Again	/parking loop		
RIT	GRAAAAAAAA	14		1.976.0 C			
R12	0x00000000	15	SumSq STHD®	sp1,(r1,r2,1r)	rsave registers on the	atack	
R13 (SP)	0x000009C	16	STR	fp,[sp,#-4]:	slave the old frame poi	nter on the stack (pre-indexing	D
R14 (LR)	0000000000	17	MOV	fp, sp	:Set up the frame point	er to point to the base of the s	teck
R15(PC)	0:00000024	18	SUB	ap, sp, #4	Move the stack pointer	to the top of the one-word star	s frame
E CPSR	0+00000003	13	LDR	r2 r1 r1	Just the value of A off	the SIECK IN II	
in Srah	\$40000000	20	570	r2. (fr. 4-41	afrone the value of 1 c	manad in the stack fame.	
East Internet		22	LDR	r2. [fp. #16]	Get the address of B p	of the stack in r2 (reuse r2)	
- interrupt		23	LDR	r1, [r2]	sGet the value of B in	r1 (reuse r1)	
Supervisor		24	LDR	r0, [fp, #-4]	:Get the value of A ago	ared in r0	
Abot		25	ADD	r1, r1, r0	sAdd 1 to A squared		
Undefined		26	STR	r1, [r2]	rreturn the result to t	he calling environment	
lemetra - E		27	MOA	ep, fp	Restore the stack point	ter and collapse the frame	
PC S	0x00000024	28	LDB	fp,[sp],#4	Restors the old frame	pointer on the stack and post-in	Crement the stack
Mode	Supervisor	29	LDBLA	sp:,(r1,r2,pc)	prestore registers and	recurn	
Scatters	13	30	a non	8	and the state of t		
Dec	2 0000000	20	B DCD	3	educative relies for R		
		13	SPACE	22	rreserve 32 bytes for t	he stack	-
		34	Stack DCD	CAFFFFFFFF	rdummy data for the bas	e of the stack	
The second second		王	END				التر
uffithed minkeg	iteri	11-1					21
imory 1							× ×
Addrees: 0							
x00000000: E	SF DO SC E	5 9F I	10 BC ES 9F 10	0 8C E5 9F 20 8C	E2 HF 30 54 E5 93 40 00	E5 2D 40 04 E2 SE 50 4C E5 2D 5	0 04 28 00 00 01
x00000028; Z	98 00 00 E	A FF 1	T TE E9 2D 4	0 06 E5 ZD B0 04	E1 AO BO OD E2 4D DO 04	E5 98 10 14 KO 02 01 91 K5 08 20	0 04 85 98 20 10
x00000050: E	92 10 00 E	5 1B (00 04 E0 81 10	0 00 E5 82 10 00	E1 A0 D0 08 E4 90 B0 04	IS BD 85 06 00 00 00 82 00 00 0	0 03 00 00 00 00
000000781 0	10 00 00 0	0.00 0				DO	
000000CB: 0	00 00 00 0	0 00 0	00 00 00 00 00	0 00 00 00 00 00	00 00 00 00 00 00 00 00		0 00 00 00 00 00
*000000F0: 0	0 00 00 00 0	0 00 0	0 00 00 00 00	0 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 0	0 00 00 00 00 00 -1
Call State 1	Call Memory	4			/ /		
Command							
ep one line						Simulation	0.00000000 141
the sure sent				/		1.2eroennon line	and the second sec
						\backslash	\backslash
		The	stack pointer				
		con	tains 0x00000	08C	/	This is parameter	A mini
		and	is pointing he	ere L	lere is the value of	with the value 3 at	This is parameter
		at th	ne address of I	Bon	arameter A (i.e. 2)	mamany logation	the value 3 at mem
		au	10 address 01 1	p on p	arameter A (I.e., 2)	memory location	location 0x000000
		41	sto alr			0.00000000	10Cation 0x000000
		the	stack.	o	n the stack at	0x000006C.	
		the	stack.	a	n the stack at ddress 0x00000090.	0x0000006C.	

The next memory map shows the situation in the subroutine after saving r1, r2, and the link register on the stack.

10314	Doject Flash	Debug Pe	sipherais Too	Nision4	icio
	a model -			n in mum l	
ROF	0-0-11-0	जना हिंदू त	amo		
inters	0 X	E Pa	aramPassingFra	measm	• ×
spieler	Value	UT UT	AREA	FrameParans, CO	DE, READWRITE
Current		02	ADR	sp. Stack	/ast up the stack pointer
- R0	9x000000000	113	LDR	fp.=0xAAAAAAAA	rdummy value for ri
R1	0x11111111	04	LDR	r1,=0x11111111	rdumny value for ri
FI2	0+22222222	05	LDR	¥2,=0x22222222	/dummy value for rg
F3	0+00000060	06	ADR	r3, A	ard is a pointer to A
R4	0x00000002	07	LDR	r4, [r3]	Jget parameter &
RS	0x00000070	08	STR	r4.[sp.#-4]!)push the value of A on the stack
RIS	0x00000000	03	ADR	r5,B	ight the address of B
R7	0x00000000	10	STR	r5, [sp, #-4] !	rpush the address of B on the stack
RB	0x00000000	31	BL	SunSq	scall the subroutine
- R9	0+00000000	12	LDR	r0,[r5]	rif it worked, rf should contain IS or OxF
R10	0x00000000	/13 Ag	main B	Again	/parking loop
R11	(HAAAAAAAAA	14			
R12	0x00000000	15 50	mSq STHDD	sp1, (r1, r2, 1r)	reave registers on the stack
R13 (SP)	0x00000050	15	STR	fp,[sp,#-4]!	/Save the old frame pointer on the stack (pre-indexing)
-R14 (LR)	5x00000028	17	MOV	fp, sp	:Set up the frame pointer to point to the base of the stack
R15 (PC)	0x00000034	18	SUB	sp, sp, #4	Move the stack pointer to the top of the one-word stack frame
IE CPSR	0+00000003	19	LDR	r1, [fp, #20]	JGet the value of A off the stack in rl
E SPSR	0x00000000	20	MOL.	r2, r1, r1	/Square &
User/System	CTOCOLLEGICO,	23	STR	r2,[fp,#-4]	Store the value of A squared in the stack frame
Fast Interrupt		22	LDR	r2, [fp, #16]	(Get the address of 8 off the stack in r3 (reuse r3)
- interrupt		23	LDR	r1, [r2]	:Get the value of B in ri (reuse ri)
Supervisor		24	LDR	r0, [fp, #-4]	:Get the value of A squared in r0
Abot		25	ADD	r1, r1, r0	rAdd I to A squared
Undefined		26	STR	r1, [r2]	return the result to the calling environment
1 A 4 1 A 4		27	MOM	ep, fp	Restore the stack pointer and collapse the frame
and entropy in the	5x00000034	28	LDB	fp, [sp],#4	Restore the old frame pointer on the stack and post-increment the stack
PC \$	and a second	and the second se	LDMTA	sp1, {r1, r2, pc}	prestore registers and return
PC 5 Mode	Supervisor	29	1010010101010101010101010101010101010101		
PC 5 Mode States	Supervisor 26	30			
PC 5 Mode States Soc	Supervisor 26 9.00000000	30 31 A	DCD	2	rdunmy value for A
PC 5 Mode States Sec	Supervisor 26 9.00000000	29 30 31 A 32 B	DCD	2	rdummy value for &
PC S Mode States Sec	Supervisor 26 9.00000000	29 30 31 A 32 B 33	DCD DCD SPACE	2 3 92	rdummy value for A rdummy value for B reserve 32 bytes for the stack
PC S Mode States Sac	Supervisor 26 9.00000000	29 30 31 A 32 B 33 34 St	DCD DCD SPACE ack DCD	2 3 52 0×FFFFFFF	rdummy value for & rdummy value for B reserve 52 bytes for the stack rdummy data for the base of the stack
PC 5 Mode States Sec	Supervisor 26 9 0000000	29 30 31 A 32 B 33 34 St	DCD DCD SPACE ack DCD END	2 3 52 0×FFFFFFFF	rdummy value for & rdummy value for B rreserve 32 bytes for the stack rdummy data for the base of the stack
PC 5 Mode Sates Sec	Supervisor 26 9 00000000	29 30 31 Å 32 B 33 34 St 14	DCD DCD SPACE ack DCD END	2 3 52 C×FFFFFFF	rdummy value for & rdummy value for B reserve 52 bytes for the stack rdummy data for the base of the stack
Project Reg	Supervisor 26 9.00005000	29 30 31 A 32 B 33 34 St 15	DCD DCD SPACE ack DCD END	2 3 92 0×FFFFFFF	rdummy value for & rdummy value for B resource 52 hytes for the stack rdummy data for the base of the stack
Project Regimery 1	Supervisor 26 9.00000000	29 30 37 A 32 B 33 34 St <u>1</u>	DCD DCD SPACE ack DCD END	2 3 52 0×FFFFFFF	rdummy value for & rdummy value for B resource 52 bytes for the stack rdummy data for the base of the stack
Project Region Project Region Project Region Project Region Region Project Region Region Region Project Region Reg	Supervisor 26 2 00000000 atem	29 30 31 A 32 B 33 34 3t 14 1	DCD DCD SPACE ack DCD END DC ES 9F 1	2 3 52 0xFFFFFFFF 0 8C 85 9F 20 8C	cdummy value for A rdummy value for B reserve 32 bytes for the stack rdummy data for the base of the stack rdummy data for the base of the stack
Project Regimery 1	Supervisor 26 9 0000000 Inten 7 5F D0 9C 8 5 95 00 00 E	23 30 31 A 32 B 33 34 St 14 14 5 SF BO A FF FF	DCD SPACE ack DCD END CD END END END END END END END END END EN	2 3 52 0×FFFFFFF 0 6C E5 9F 20 8C 0 6C E5 9F 20 8C	cdummy value for A rdummy value for B rreserve 32 bytes for the stack rdummy data for the base of the stack
PC 5 Mode Sates Sec Finject Emery 1 Concologe Conco	Supervisor 26 9 00000000 atters 2 8F D0 8C 8 5 95 00 90 E 5 92 10 00 E	29 30 31 A 32 B 33 34 St 15 15 15 5 9F BO A FF FF 5 1B 00	DCD SPACE ack DCD END J BC ES 9F 1 FE E9 2D 4 04 <u>F0 81 1</u>	2 32 0×FFFFFFFF 0 SC E5 9F 20 SC 0 06 E5 2D 86 20 00	cdummy value for A cdummy value for B interpret 32 bytes for the stack ;dummy data for the base of the stack :dummy data for the stack :dummy data for the base of the st
Project Regional	Supervisor 26 0 00000000 atten 2 5F D0 8C E 5 95 00 00 E 5 92 10 00 E 5 92 10 00 E	23 30 31 A 32 B 33 34 St 14 1 • 1 • 1 •	DCD SPACE SPACE ENE SC E5 9F 1 FE E9 2D 4 04 FD 51 1 04 FD 51 1	2 32 0xFFFFFFFF 0 8C 85 9F 20 8C 0 9C 85 2D 80 04 0 00 85 82 10 00 1 12 22 22 20 20	cdummy value for A rdummy value for B reserve 32 bytes for the stack rdummy data for the base of the stack E2 BF 30 54 E5 93 40 00 E5 2D 40 04 E3 BF 50 4C E5 2D 50 04 EB 00 00 01 E1 A0 50 0D E2 4D D0 04 E5 95 10 14 E0 02 01 91 E5 08 20 04 E5 95 20 10 E1 A0 50 0D E2 4D D0 04 E5 95 10 14 E0 02 01 91 E5 08 20 04 E5 95 20 10 E1 A0 50 00 02 20 00 00 00 00 00 01 FF FF FF AA AA AA 11 11 11 11
Project ERep Project ERep Project D0000000 % % 100000000 % % 100000000 % % 10000000 % % 10000000 % % 10000000 % % 10000000 % % 10000000 % % 1000000 % % 1000000 % % 1000000 % % 100000 % 10000 % 100000 % 10000 % 10000 % 10000 % 100000 % 10000 % 100	Supervisor 26 26 0000000 atten 2 8F D0 8C E 5 95 00 00 E 5 92 10 00 E 5 92 10 00 E 5 92 22 22 0	23 30 31 A 32 B 33 34 3t 15 14 5 9F 80 A FF FF 5 1B 00 6 00 00 6 00 00	DCD SPACE SPACE EXPL C BC E5 9F 1 FE E9 2D 4 04 E0 81 1 00 00 00 00 00	2 3 52 0×FFFFFFFF 0 SC E5 9F 20 SC 0 06 E5 2D B0 04 0 00 E5 82 10 00 1 11 22 22 22 22 0 00 00 00 00	cdummy value for A rdummy value for B reserve 32 bytes for the stack rdummy data for the base of the stack rdummy data for the stack rdummy data for the base of the stack rdummy data for the stack rdumy data for the stack rdumy data for the stack rdummy data fo
Traject PC 5 Mode Sates Sec Traject Employ1 Concologe Rep 1 00000001 8 1 00000001 8 1 00000001 8 1 0000001 8 1 0000001 8 1 0000001 8 1 0000001 8 1 0000001 8 1 0000001 8 1 0000001 8 1 0000001 8 1	Supervisor 26 9 00000000 atters 2 8F D0 8C 8 5 95 00 90 8 5 92 10 00 8 2 00 00 00 0 2 2 2 2 2 0 0 00 00 00 0	23 30 31 Å 32 B 33 34 3t 35 34 5t 5 9F 80 A FF FF 5 1B 00 6 00 00 6 00 00 6 00 00	DCD SFACE SFACE END J BC ES 9F 1 FE E9 2D 4 04 F0 81 1 00 11 11 1 00 00 00 00	2 32 0×FFFFFFF 0 0C E5 9F 20 8C 0 0C E5 2D 80 00 0 0C E5 2 10 00 1 11 22 22 32 22 0 00 00 00 00 00	cdummy value for A rdummy value for B resource 32 bytes for the stack rdummy data for the base of the stack for the base of the stack resource 12 bytes for the base of t
Project Regimer, 1 Project Regimer, 1 Project Regimer, 1 Project Regimer, 1 Project	Supervisor 26 9 00000000 sters 2 8F D0 8C 8 9 90 00 00 82 9 92 10 00 00 8 9 92 10 00 00 0 2 22 22 20 0 00 00 00 00 0 0 00 00 00 00 0	23 30 31 32 33 34 35 34 35 34 35 36 37 37 34 35 36 37 36 37 37 37 37 37 37 37 37 37 37	BC E5 9F 1 EE E5 9F 1 EE E5 2D 5 04 F0 61 11 20 05 00 00 00	2 3 52 CxFFFFFFFF 0 6C 25 9F 20 8C 0 06 25 2D 80 04 0 00 25 22 10 00 1 11 20 22 22 22 0 00 00 00 00 00 0 00 00 00 00 00	Columny value for A rolumny value for B reserve 52 bytes for the stack rolumny data for the base of the stack rolumny data for the base of the stack E2 8F 30 54 55 93 40 00 55 2D 40 04 52 8F 50 4C 55 2D 50 04 E5 00 00 01 E1 A0 50 0D E2 4D D0 04 25 9B 10 14 20 02 01 91 25 08 20 04 25 9B 20 10 E1 A0 50 0D E2 4D D0 04 25 9B 10 14 20 02 01 91 25 08 20 04 25 9B 20 10 E1 A0 50 0D E2 4D D0 04 25 9B 10 14 20 02 00 00 02 00 00 00 00 00 00 00 E1 A0 50 00 02 21 00 00 00 70 00 00 00 00 00 00 00 00 00
Frend PC 5 Mode Sates Soc Soc Frend Energy 0000000000 8: 00000000000 8: 0000000000 8: 0000000000 8: 000000000 8: 0000000000 8: 0000000000 10: 000000000000000000000000000000000000	Supervisor 26 9 00000000 sters 2 8F D0 8C 8 9 90 00 00 8 9 20 00 00 0 2 22 22 0 0 00 00 00 0 0 00 0 00	23 30 31 32 33 34 35 5 9F 80 6 00 00 6 00 00 0 00 00 1	BC E5 9F 1 FE E9 2D 4 00 00 00 00 00 00 00 00	2 3 52 C×FFFFFFFF 0 6C 25 9F 20 8C 0 06 25 2D 80 04 0 00 25 21 00 04 0 00 25 82 10 00 1 1 20 22 22 22 22 0 00 00 00 00 00 0 00 00 00 00 00	<pre>cdummy value for A rdummy value for B reserve 32 bytes for the stack rdummy data for the base of the stack rdum data for the base</pre>
Termel PC 5 Mode States Sec Project Project Project Project	Supervisor 26 2 00000000 attens 2 8F D0 8C 8 5 95 00 00 8 5 92 10 00 8 5 92 10 00 8 2 2 22 20 0 00 00 00 2 2 22 22 0 0 00 00 00 0 00 00 0 00 00 0 00 00 0 00 00 0 00	23 30 31 32 33 34 35 34 35 34 35 36 37 37 38 39 39 39 34 35 36 37 36 37 37 37 37 37 37 37 37 37 37	BC E5 9F 1 FE E9 2D 4 00 00 00 0	2 3 52 CxFFFFFFFF 0 &C E5 9F 20 &C 0 &C E5 2D &C 04 0 00 E5 22 30 00 11 22 22 22 22 22 0 00 00 00 00 00 0 00 00 00 00 00	cdummy value for A rdummy value for B reserve 32 bytes for the stack rdummy data for the base of the stack rdummy data for the base of the stack rdummy data for the base of the stack rdum y data for the base of the stack pointer contains for the
Project Reg Project Reg proy 1 deem 0 00000001: 21 00000001: 21 0000001: 21 00000001: 21 00000001: 21 00000001: 21 00000001: 21 00000001: 21 00000001: 21 00000001: 21 00000001: 21 00000001: 21 0000000001: 21 000000001: 21 00000001: 21 000000001: 21 000000001: 21 00000001: 21 00000001: 21 00000001: 21 00000001: 21 00000000001: 21 000000001: 20 00000001: 20 00000001: 20 00000001: 20 00000001: 20 0000001: 20 00000001: 20 00000001: 20 00000001: 20 00000001: 20 00000001: 20 00000001: 20 00000001: 20 00000001: 20 00000000000000000000000000000000000	Supervisor 26 2 00000000 Inters 2 8F D0 8C E 5 95 00 90 E 5 92 10 00 E 5 92 10 00 E 0 00 00 00 0 0 00 00 00 0 0 00 00 00 0 12 22 22 0 0 00 0 12 22 20 0 0 0 0 0 12 22 20 0 12 20 0 12 20 0 12 20 0 12 20 0 12 20 0 10	5 9F 80 A FF FF 5 18 00 0 00 00 0 00 00 1	DCD SPACE SPACE EXPD SPACE EXPD EXPD EXPD EXPD EXPC EXPD EXPC EXPC EXPC EXPC EXPC EXPC EXPC EXPC	2 3 52 0×FFFFFFFF 0 SC E5 9F 20 SC 0 06 E5 2D B0 04 0 06 E5 2D B0 04 1 1 22 22 22 22 0 00 00 00 00 00 0 00 00 00 00 00 0 00 00 00 00 00	cdummy value for A rdummy value for B reserve 32 bytes for the stack rdummy data for the base of the stack rdummy data for the base

The following figure demonstrates the structure of the stack in this example. Note that addresses on the left are given with respect to the frame pointer. This helps to relate the stack to the offsets in the above code.

		Memory	1						
	fp 4		Low memory						
Stack pointer	ip - 4	Top of stack	The stack frame						
Frame pointer pointer	fp 🕨	0xAAAAAAA							
	fp + 4	r1 = 0x111111111	↑						
	fp + 8	r2 = 0x22222222	Saved on entry to the subroutine	1					
	fp + 12	lr (r13) = 0x00000028	¥						
	fp + 16	0x70 Address of B	Parameters pushed	Direction of					
	fp + 20	0x2 Value of A	on the stack	are added					
		0xFFFFFFF	The stack base						
			J						

.

The following figure shows the memory map after squaring A and putting it in the stack frame.



The next memory map shows the situation after storing the result in the calling environment and before cleaning up the stack frame.



In the final snapshot of memory we show the memory map at the end of the program. A return has been made to the calling program and we have completed the program and are in a parking loop. All the registers have been reset to their original values except r13, the stack pointer, the program counter, and r0 which was a global scratchpad register.

E:\CengageBoo	k\GengageW	orkBook\E	ample13	uvpraj - j	/Vision4	10																			-	
Eile Edit View	Bioject Fig	ish Debug	Pegiphe	say Io	on svc	5 <u>W</u> W	idow)	Help																		
	1. 1.1.5				117	× * /		3				- 3		1	2 4		0	a I	•	3	1					
1 0 6 禄	() () ()	0 II	0.0	40 60	- 0	3.	H -	B • 1	-	2.																
Registers	.0	×) Parami	assingFra	measm																					• ×
Register	Value	13		LDR	fp.=	OxAAA	алала	7dui	ny v	a1:0e	fe	r rl														-
Current	97	84		LDR	r1,-	0#111	11111	2 du	_y. v	alue	fo	r r1														-
RQ	040000000	11 II I		LDR	r2,=	08222	22222	2:211	my v	alue	fa	2 22														
81	041111111	05		1 DE	23.A	193		28.3	28.4	por	nce	F DO	*													
	0-0000006	- No		STR	r4. 1	m.t.	411	2000	the state		1.0	100.0	i ne l	- 2.4		- 14										
- R4	0+0000000	2 03		ADR	r5.B			Joet	t the	add	ires	5 01	B													
RS	0x0000007	10		STR	T5. [sp, f-	411	2003	in th	6.44	dre	45 01	E o	n th	= =1	ia ch										
RŚ	0x0000000	11		BL	Sun5	Q		2083	I. ch	# .#U	bru	uting	Ē													
R7	9x0000000	12	1	LDR	r0,[r5]		742	it y	orke	d.	r0 s1	bluo	con	tair	: 15	0.0	OXF								- 11
RB	0+0000000	0 013	Again	В	Agai	n.		7041	rking	100	1															
R9	0=0000000		0.000	ommo	-		1.1.1																			
P11	0-000000	15	amad	STR	fr. f	RD . 8-	411	1501	ne rh	g150	015	Pane I	ne a	LOCE	0= 1	2.4	stad	a - 2	niria.	114	awin.	11				
Rt2	0+0000000	17		MOV	fu.s	0	20.0	1000	00	the	(re	De oc	inte	r to	201	nt	tig t	-	ase	65	the	stac	6			
R13 (SP)	Gr0000008	18		SUB	sp, s	p,#4		:No	w th		auk	poir	ter	to t	he t	top:	of t	he c	D#-1	ard	ata	ik I	rane			
R14 (LR)	9x0000002	8 19		LDR	r1, [fp,#2	01	(GH)	t the	va1	12.88	05 A	130	the	stac	1 1	n 11									
R15 (PC)	0,0000002	20		MUL.	r2. r	1, r1		750	isre-	A ···																
IE CPSR	0+0000000	3 21		STR	r2, [fp,#-	4]	(5t.0	ne c	he v	win	e of	1 20	are:	d 11	1 th	e st	ack -	fran							
± SPSR	0+0000000	22		LDR	12.1	CE+#1	41	VGet	: the	add	res	s of	B of	r th	e 31	ack.	2.0	12 (reur	ne i	27					
User/System		4		LUB	-1.1	rej Frito	41	1083	COS .	Va.	00	or s	TH P	1 11	1226	1 11	2.									
T intert at				ADD	r1. r	1.00	11	2.44	1 . 1	1.4		ared	a qua	Less.	414 4											
Supervisor		28		STR	r1.[r21		7.5 11	turn.	the.	198	ult t	is th		1210	ir e	nvir	onte	30							
Abot		27		MOV	sp,f	p		7RH	store	the	art:	ack g	wint	er s	nd a	1100	spse	the	fre	ine						
E Undefined		23		LDR	fp.I	*p],#	4	1Res	store	the	ol.	d fra	ine p	oint	er d	in t	he s	teck.		i po	st-1	ncem	tesp	the		ch
E Internel		29		LDHIA	eg:1,	(rl,r	2, pc]	1201	store	reg	ist	ers a	ind - r	ecur	n											
PC S	6x0000002	; 30			- 2																					
Mode	Supervisor	31		DCD				2 due	my v	4108	IC.	5 A														
Scatters	53	- 46	-	SPACE	32			rau	-Y V	4108	and the	T 0	+ + 1		ant.											
Dec	2 0000000	34	Stack	DCD	OMTE	FFFFF	F	rdim	and d	ata	for	the	Dage	of	the	sta	ck.									
		35		END						1.1																
		36																								-
Reg	inters	1.1																								1
Memory 1	80																									ų ×
Address: 0																									1	1-
0x00000000: E	2 SF DO S	C 85 9F	80 8C 8	5 9F 1	0 80 8	5 91	20.80	82 8	F 30	54.3	c5 9	3 40	00 8	5 21	40	0.5	82 5	F 50	+C	85	ZD 5	0 04	EB	00 0	10 01	
0x0000028: Z	5 95 00 0	EA FF	FF FE E	9 2D 4	0 06 2	S ZD	80 04	El A	0 50	0D 1	12 4	D D0	04 2	5 92	10	14	20 0	2 01	. 91	55	08.2	0.04	25	98 2	10 10	2
9x00000050: E	S 92 10 0	D ES 1B	00 04 E	0 81 1	0 00 E	5 82	10 00	EI A	o DG	08 3	14 9	HD 30	04 E	8 80	3 80	0.6	00.0	0 00	92	00	00 0	0.07	00	00 0	10.00	2
0x0000078: 0	0 00 00 0	AA AA	AA AA I	1 11 1	1 11 2	2 22	22 22	00.0	0 00	28 (0 00	0 00	78 0	0 00	00	0.2	FF I	F FI	FF	AA	AA A	à 22	11	11 1	1 11	
0x000000A0: 2	2 12 22 2	00 00	00 00 0	0 00 0	0 00 0	00 00	00 00	00 0	0 00	00 1	00.0	00 00	00 0	0 00	00	00	00 0	0 00	00	00	00 0	0 00	00	00 0	0 00	2
OxOODOOCE: 0	0 00 00 0	00 00 0	00 00 0	0 00 0	0 00 0	00 00	00 00	00 0	0 00	00 0	0.00	00 00	00 0	0 00	00	00	00 0	0 00	00	00	00 0	0.00	00	00 0	10 00	-
Call State 201	ocate Me	nory 1	00 00 0	0 00 0	9 00 0	0.00	90.00	00 0	0.00	00.1	-0.0	10 QU	00.0	0.00	. 00	00	90.0	0.01	- 44	00	00.0	9.95	00		10 01	
Command	-	10																								
and reconstructed in								1								51	nulati	in			ti.	0.000	00000	iet	1	E

The point of this example was to demonstrate the stack frame and passing parameters both be reference and value.

This is both a good example and a bad example. It is good in the sense that it is relatively simple. It is bad in the sense that no one would write this code because a stack frame is not necessary because there are enough registers for the local storage.

However, this example does illustrate how much overhead is associated with accessing data in memory.

APPENDIX

ARM Mnemonics

This appendix provides brief details of the part of the ARM's instruction set. We haven't included instructions that operate on the coprocessor.

ADC	Add with carry	$Rd \leftarrow Rn + Op2 + Carry$
ADD	Add	Rd ← Rn + Op2
AND	AND	Rd ← Rn AND Op2
В	Branch	$R15 \leftarrow address$
BIC	Bit Clear	Rd ← Rn AND NOT Op2
BL	Branch with Link	R14 \leftarrow R15, R15 \leftarrow address
ВΧ	Branch and Exchange	$R15 \leftarrow Rn, T bit \leftarrow Rn[0]$
CMN	Compare Negative	CPSR flags ← Rn + Op2
CMP	Compare	CPSR flags ← Rn - Op2
EOR	Exclusive OR	Rd ← Rn ⊕ Op2
LDM	Load multiple registers	
LDR	Load register from memory	$Rd \leftarrow [address]$
MLA	Multiply Accumulate	Rd := (Rm · Rs) + Rn
MOV	Move register or constant	Rd ← Op2
MRS	Move PSR status/flags to Register	Rn ← PSR
MSR	Move register to PSR	status/flags PSR \leftarrow Rm
MUL	Multiply	$Rd \leftarrow Rm \cdot Rs$
MVN	Move negative	register Rd \leftarrow 0xFFFFFFFF EOR Op
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	$Rd \leftarrow Rn - Op2 - 1 + Carry$
STM	Store Multiple	
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 \leftarrow Rn EOR Op2
TST	Test bits	CPSR flags \leftarrow Rn AND Op2