ARM Assembly Language Guide

ARM is an example of a Reduced Instruction Set Computer (RISC) which was designed for easy instruction pipelining. ARM has a "Load/Store" architecture since all instructions (other than the load and store instructions) must use register operands. ARM has 16 32-bit "general purpose" registers (r0, r1, r2, ... , r15), but some of these have special uses (see ARM Register Conventions table on page 4).

Common ARM Instructions (and **psuedo-instructions**)		
Type of Instruction	**ARM Assembly Language**	**Register Transfer Language Description**
Memory Access (Load and Store)	LDR r4, Mem	[r4] ← [Mem] ; Mem is a global variable label
	STR r4, Mem	[Mem] ← [r4]
	LDR r4, [r3]	[r4] ← [[r3]] ; *register indirect*
	STR r4, [r3, #4]	[[r3] + 4] ← [r4] ; *register indirect with offset*
Move	MOV r4, r2	[r4] ← [r2]
	MOV r4, #10	[r4] ← 10 ; 8-bit literal, but can be shifted
Load Address	**ADR** r4, Mem	[r4] ← load address of label Mem
Arithmetic Instruction (reg. operands only, except last operand can be an 8-bit integer)	ADD r4, r2, r3	[r4] ← [r2] + [r3]
	MUL r4, r2, r3	[r4] ← [r2] * [r3] (32-bit product)
	SUB r4, r2, r3	[r4] ← [r2] - [r3]
Compare (sets condition codes)	CMP r4, r2	Sets condition codes by r4 - r2
Conditional Branch	BGT LABEL (BGE, BLT, BLE, BEQ, BNE)	Branch to LABEL if r4 > r2 if it follow above CMP
Unconditional Branch	B LABEL	Always Branch to LABEL

A simple ARM assembly language program to sum the elements in an array A is given below:

```
      ; ARM Example that sums an array via the algorithm:
      ; SUM = 0                         (uses r6 for sum)
      ; for I = 0 to LENGTH - 1 do      (uses r1 for I)
      ;      SUM = SUM + ARRAY[I]       (uses r3 for address of A[I])
      ; end for
           AREA SUMARRAY, CODE, READONLY
           ENTRY                  ; Always needed to indicate where to start pgm
           LDR r2, LENGTH
           SUB r2, r2, #1         ; r2 contains (LENGTH-1)
           MOV r6, #0             ; r6 sum set to 0
FOR_INIT   MOV  r1, #0            ; r1 index I set to 0
           ADR r3, ARRAY          ; start r3 with address of A[0]
FOR_CMP    CMP r1, r2             ; compare I and (LENGTH-1)
           BGT END_FOR            ; drop out of loop if I < (LENGTH-1)
           LDR r4, [r3],#4        ; load r4 with A[I] then walk r3 down ARRAY
           ADD r6, r6, r4         ; update sum with A[I]
           ADD r1, r1, #1         ; increment I
           B FOR_CMP              ; loop back to for-loop check
END_FOR
           STR r6, SUM            ; store result in SUM
STOP       B STOP

           AREA SUMARRAY, DATA, READWRITE
           ALIGN
SUM        DCD 0XFFFFFFFF
ARRAY      DCD 5, 10, 15, 20, 30, 40, 50
LENGTH     DCD 7

           END                    ; Needed to stop assembly
```

ARM Logical Instructions		
AND r4, r2, r3	[r4] ← [r2] (bit-wise AND) [r3]	
AND r4, r2, #0xFF000000	[r4] ← [r2] (bit-wise AND) $FF000000_{16}$	
ORR r4, r2, r3	[r4] ← [r2] (bit-wise OR) [r3]	
EOR r4, r2, r3	[r4] ← [r2] (bit-wise XOR) [r3]	
BIC r4, r2, r3	[r4] ← [r2] (bit-wise AND) (NOT [r3])	BIt Clear - clear bits set in r3
MOVN r4, r2	[r4] ← (NOT) [r2]	Flip all the bits

ARM Shift and Rotate Instructions	
MOV r4, r5, LSL #3	r4 ← logical shift left r5 by 3 positions. (Shift in zeros)
MOV r4, r5, LSL r6	r4 ← logical shift left r5 by the number of positions specified in register r6
MOV r4, r5, LSR #3	r4 ←logical shift right r5 by 3 positions. (Shift in zeros)
MOV r4, r5, ASR #3	r4 ← arithmetic shift right r5 by 3 positions. (Shift with sign-extend)
MOV r4, r5, ROR #3	r4 ← rotate right r5 by 3 positions. (Circulate bits)
AND r4, r5, r6, LSL #2	Shifts can operate on 3rd register operand of arithmetic or logical instruction, e.g., r4 ← r5 AND (logical shift left r6 by 8 positions)

Common usages for shift/rotate and logical instructions include:

1. To calculate the address of element array[i], we calculate (base address of array) + i * 4 for an array of words. Since multiplication is a slow operation, we can shift i's value left two bit positions. For example:

```
ADR r3, ARRAY          # load base address of ARRAY into r3 (ARRAY contains 4-byte items)
LDR r2, I              # load index I into r2
MOV r4, r2, LSL #2     # logical shift i's value in r2 by 2 to multiply its value by 4
ADD r5, r3, r4         # finish calculation of the address of element array[i] in r5
LDR r4, [r5]           # load the value of array[i] into r4 using the address in r5
```

Alternatively, we can perform this same address calculation with a single ADD:

```
ADD r5, r3, r2, LSL #2  # calculate address of array[i] in r5 with single ADD
LDR r4, [r5]            # load the value of array[i] into r4 using the address in r5
```

Alternatively, ARM has some nice addressing modes to speedup array item access:

```
LDR r4, [r3,r2,LSL #2]        # load the value of array[i] into r4
```

2. Sometimes you want to manipulate individual bits in a "*string of bits*". For example, you can represent a set of letters using a bit-string. Each bit in the bit-string is associated with a letter: bit position 0 with 'A', bit position 1 with 'B', ..., bit position 25 with 'Z'. Bit-string bits are set to '1' to indicate that their corresponding letters are in the set, and '0' if not in the set. For example, the set { 'A', 'B', 'D', 'Y' } would be represented as:

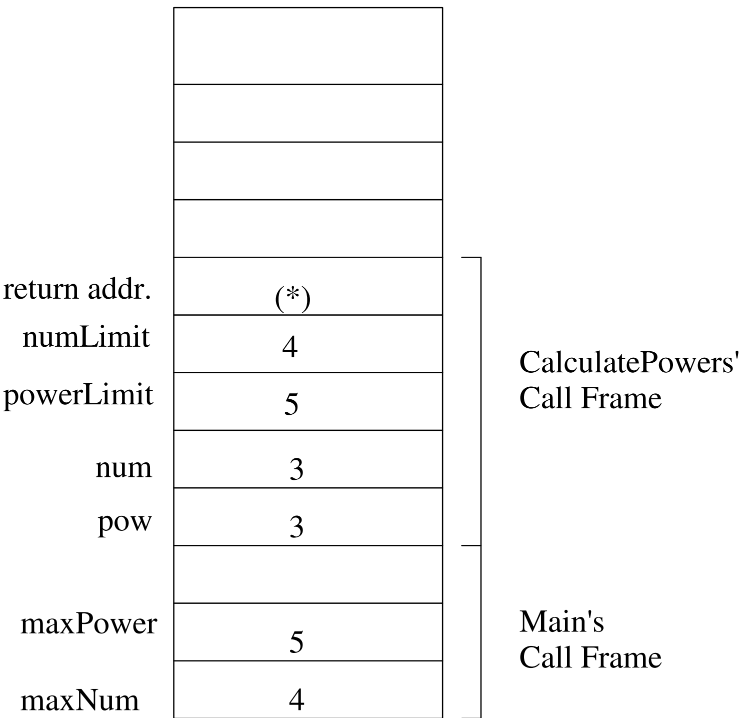
	unused	'Z'	'Y'	'X'	. . .	'E'	'D'	'C'	'B'	'A'
{ 'A', 'B', 'D', 'Y' } is	0 0 0 0 0 0	0	1	0		0	1	0	1	1
bit position:		25	24	23		4	3	2	1	0

To determine if a specific ASCII character, say 'C' (67_{10}) is in the set, you would need to build a "*mask*" containing a single "1" in bit position 2. The sequence of instructions "MOV r3, #1" followed by "MOV r3, r3, LSL #2" would build the needed mask in r3. If the bit-string set of letters is in register r5, then we can check for the character 'C' using the mask in r3 and the instruction "AND r6 r5, r3". If the bit-string set in r5 contained a 'C', then r6 will be non-zero; otherwise r6 will be zero.

High-level Language Programmer's View

main:	**CalculatePowers**(**In:** integer numLimit, integer powerLimit)	integer **Power**(**In:** integer n, integer e)

maxNum = 4
maxPower = 5
 	
CalculatePowers(maxNum, maxPower)
(*)
. . .
end main
	integer num, pow
	
	for num := 2 to numLimit do
	for pow := 1 to powerLimit do
	
	print num " raised to " pow " power is " Power(num, pow)
	end for pow
	end for num
		integer result
		if e = 0 then
		result = 1
		else if e = 1 then
		result = n
		else
		result = Power(n, e - 1)* n
		end if
		return result
		end Power



Compiler uses registers to avoid accessing the run-time stack in memory as much as possible. Registers can be used for local variables, parameters, return address, function-return value.

```
AL code for subprogram "caller"

  <code using some registers>

 call subprogram

  <wants used registers to be unchanged>
```

When a subprogram is called, some of the register values might need to be saved ("spilled") on the stack to free up some registers for the subprogram to use.

Standard conventions for spilling registers:
1) caller save - before the call, caller saves the register values it needs after execution returns from the subprogram
2) callee save - subprogram saves and restores any register it uses in its code
3) some combination of caller and callee saved (USED BY ARM)

ARM Register Conventions (APCS - Application Procedure Call Standard)			
Reg. #	**APCS Name**	**Role in Procedure Calls**	**Comments**
r0 - r3	a1 - a4	First 4 arguments into a procedure/Scratch pad/Return result(s) from a function (not preserved across call)	Caller-saved registers - subprogram can use them as scratch registers, but it must also save any needed values before calling another subprogram.
r4 - r8	v1 - v5	Register Variables (preserved across call)	Callee-saved registers - it can rely on an subprogram it calls not to change them (so a subprogram wishing to use these registers must save them on entry and restore them before it exits)
r9	sb/v6	Static base / Register Variable (preserved across call)	Callee-saved register - pointer to static base in memory
r10	sl/v7	Static base / Register Variable (preserved across call)	
r11	fp	Frame pointer (if used) / Register Variable (preserved across call)	Callee-saved register - pointer to bottom of call-frame
r12	ip	Intra-procedure call scratch register (not preserved across call)	Caller-saved register - used by linker as a scratch register. It can be used by a routine as a scratch register
r13	sp	Stack pointer - points to the top of the stack	
r14	lr	Link register - holds the return address	Receives return address on *BL* call to procedure
r15	pc	Program counter	

Using ARM Calling Convention	
Caller Code	**Callee Code**
. . . 1) save on stack (callee-saved regs) a1-a4/ip that are needed upon return 2) place arguments to be passed in a1- a4 with additional parameters pushed onto the stack 3) BL ProcName # saves return address in lr 4) restore any callee-saved registers a1-a4/ip from stack	. . . 1) allocate memory for frame by subtracting frame size from sp 2) save old fp on stack and set new fp (if fp is being used) 3) callee-saved registers (v1 - v7) if more registers than scratch registers (a1-a4, ip) are needed 4) save lr and any needed (a1-a4, ip) if another procedure is to be called **. . . code for the callee procedure** 5) for functions, place result(s) to be returned in a1-a4 6) restore any callee-saved registers (v1 - v7) from step (2) above 7) restore lr and fp if it was saved on the stack in step (3) 8) pop stack frame by adding frame size to sp 9) return to caller by moving lr into pc

main:	**CalculatePowers**(**In:** integer numLimit, integer powerLimit)	integer **Power**(**In:** integer n, integer e)
maxNum = 4		integer result
maxPower = 5	integer num, pow	if e = 0 then
		result = 1
CalculatePowers(maxNum, maxPower)	for num := 2 to numLimit do	else if e = 1 then
(*)	for pow := 1 to powerLimit do	result = n
. . .		else
end main	print num " raised to " pow " power is "	result = Power(n, e - 1)* n
	Power(num, pow)	end if
	end for pow	return result
	end for num	**end Power**
	end CalculatePowers	

a) Using the ARM register conventions, what registers would be used to pass each of the following parameters to CalculatePowers:

maxNum	maxPower

b) Using the ARM register conventions, which of these parameters ("numLimit", "powerLimit", or both of them) should be moved into v-registers?
(**NOTE: Use an v-register for any value you still need after you come back from a subprogram/function/procedure call, e.g., call to "Power"**)

c) Using the ARM register conventions, what registers should be used for each of the local variables:

num	pow

d) Using the ARM register conventions, what registers would be used to pass each of the following parameters to Power:

num	pow

e) Using the ARM register conventions, which of these parameters ("n", "e", or both of them) should be moved into v-registers?

f) Using the ARM register conventions, what register should be used for the local variable:

result

g) Write the code for main, CalculatePowers, and Power in ARM assembly language.

```
main:


integer scores [100];
integer n;  // # of elements


InsertionSort(scores, n)

(*)
. . .

end main
```

```
InsertionSort(numbers - address to integer array,
              length - integer)

    integer firstUnsortedIndex
    for firstUnsortedIndex = 1 to (length-1) do
        Insert(numbers, numbers[firstUnsortedIndex],
               firstUnsortedIndex-1);
    end for

end InsertionSort
```

```
Insert(numbers - address to integer array,
       elementToInsert - integer,
       lastSortedIndex - integer) {
    integer testIndex;
    testIndex = lastSortedIndex;
    while (testIndex >=0) AND
          (numbers[testIndex] > elementToInsert ) do
        numbers[ testIndex+1 ] = numbers[ testIndex ];

        testIndex =  testIndex - 1;
    end while
    numbers[ testIndex + 1 ] = elementToInsert;
end Insert
```

a) Using the ARM register conventions, what registers would be used to pass each of the following parameters to InsertionSort:

scores	n

b) Using the ARM register conventions, which of these parameters ("numbers", "length", or both of them) should be moved into v-registers?

c) Using the ARM register conventions, what registers should be used for the local variable "firstUnsortedIndex"?

d) Using the ARM register conventions, what registers would be used to pass each of the following parameter values to Insert:

numbers	numbers[firstUnsortedIndex]	firstUnsortedIndex-1

e) Using the ARM register conventions, which of these parameters ("numbers", "elementToInsert", or "lastSortedIndex") should be moved into v-registers?

f) Using the ARM register conventions, what registers should be used for the local variable "testIndex"?

g) Write the code for main, InsertionSort, and Insert in ARM assembly language.

```
                  AREA CALCULATE_POWERS_EXAMPLE, CODE, READONLY
                  ; Calculate Powers example.  Calculates
                  ENTRY
MAIN              ADR sp, STACK_START
                  MOV a1, #4
                  MOV a2, #5
                  BL CALCULATE_POWERS

STOP              B STOP

CALCULATE_POWERS
                  STMFD sp!, {v1,v2,v3,v4,lr}
                  ;v1 holds numLimit
                  ;v2 holds powerLimit
                  ;v3 holds num
                  ;v4 holds pow
                  MOV v1, a1        ; SAVE PARAMETERS TO V-REGS.
                  MOV v2, a2
FOR_INIT_1
                  MOV v3, #2
FOR_CMP_1         CMP v3, v1
                  BGT END_FOR_1

FOR_INIT_2
                  MOV v4, #1
FOR_CMP_2         CMP v4, v2
                  BGT END_FOR_2

                  MOV a1, v3        ; CALL POWER FN
                  MOV a2, v4
                  BL POWER

                  NOP; PRINT WOULD BE HERE

                  ADD v4, v4, #1
                  B FOR_CMP_2
END_FOR_2
                  ADD v3, v3, #1
                  B FOR_CMP_1
END_FOR_1
                  LDMFD sp!, {v1,v2,v3,v4,pc}

POWER             STMFD sp!, {v1,lr}
                  MOV v1, a1
IF_1              CMP a2, #0
                  BNE ELSE_IF_1
                  MOV a1, #1
                  B    END_IF_1
ELSE_IF_1         CMP a2, #1
                  BNE ELSE_1
                  ; a1 already contains n
                  B END_IF_1
ELSE_1            SUB a2, a2, #1
                  BL POWER
                  MUL ip, a1, v1
                  MOV a1, ip
END_IF_1
                  LDMFD sp!, {v1,pc}

                  AREA CALCULATE_POWERS_EXAMPLE, DATA, READWRITE

STACK_END         SPACE 0x00000FF
                  ALIGN
STACK_START       DCD 0
DUMMY             DCD   0x12345678
                  END
```