COL216 Computer Architecture **Minor Test1** **05.02.2018**

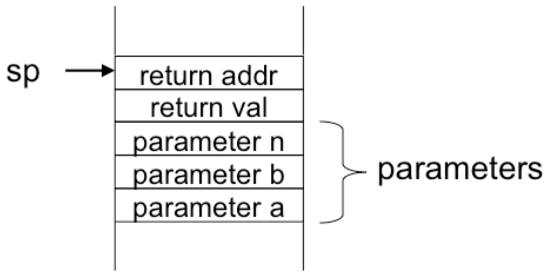
1. Write a recursive function in ARM assembly for the following C function, with a restriction that all the parameters and results have to be passed through stack.

```
 int func (int a, b, n) {

      if (n == 0) return (b);
      else
              return (func (a + b, a, n – 1));

};
```

[4]

Solution:

Stack frame



Low level C code	**Assembly code**
int func (int a, b, n) {	func: str lr, [sp, # -4]! @ save (push) lr in current frame
t0 = a;	ldr r0, [sp, # 28] @ load a from current frame
t1 = b;	ldr r1, [sp, # 24] @ load b from current frame
t2 = n;	ldr r2, [sp, # 20] @ load n from current frame
if (t2 == 0)	cmp r2, # 0
return t1;	beq Ret
else	
t1 = t0 + t1;	add r1, r0, r1
t2 = t2 - 1;	sub r2, r2, # 1
t1 = func (t1, t0, t2);	str r1, [sp, # -4]! @ push a+b into callee frame str r0, [sp, # -4]! @ push a into callee frame str r2, [sp, # -4]! @ push n-1 into callee frame sub sp, sp, # -4 @ leave space in frame for the result bl func @ recursive call ldr r1, [sp], # 4 @ pop result from callee frame add sp, sp, # 12 @ release space used for parameters

return t1;
};
	Ret: str r1, [sp, # 16] @ store result in current frame
	ldr lr, [sp], # 4 @ restore (pop) lr from current frame
	mov pc, lr @ return to caller

[1 mark for passing input parameters through stack.
1 mark for passing the result through stack.
1 mark for correct call/return and saving/restoring of lr.
1 mark for correctly translating computational part.]

2. The following ARM function takes 2 packed unsigned 7 digit BCD numbers as arguments in r0 and r1 and produces a result in r0. Registers r6 and r7 hold the constants 0x06666666 and 0x11111110, respectively. Find the result if the argument values are 0x01762493 and 0x05634882, showing the value computed at each step. What does this code do in general? Explain how you arrived at the answer. [Note: In packed BCD, each digit occupies 4 bits.]

```
func:  add r2, r0, r6
       eor r3, r2, r1
       add r2, r2, r1
       eor r3, r2, r3
       bic r3, r7, r3
       orr r3, r3, r3 LSL #1
       sub r0, r2, r3 LSR #3
       mov pc, lr
```

[5]

Solution:

Instruction	Register contents (in hex) after instruction				Explanation
	r0	r1	r2	r3	
	01762493	05634882			Initial values
(1) add r2, r0, r6			07dc8af9		6 added to each digit to get correct BCD carries
(2) eor r3, r2, r1				02bfc27b	xor of two addends
(3) add r2, r2, r1			0d3fd37b		sum before removing extra 6's
(4) eor r3, r2, r3				0fe01100	carries for all bit positions = xor of addends and sum
(5) bic r3, r7, r3				10110010	positions where there is no BCD carry
(6) orr r3, r3, r3 LSL #1				30330030	first step for removing extra 6's
(7) sub r0, r2, r3 LSR #3	07397375				sum after removing extra 6's. r3 LSR #3 is 0x06066006 here
(8) mov pc, lr					

The result (contents of r0) = 0x07397375.

[2 marks for correct answer. 1 mark if answer is ok except for some minor error]

In general, the function performs BCD addition of two numbers. [1 mark]

The last column of the table explains what each step does, leading to the computation of BCD sum. The basic idea is to achieve BCD addition through binary addition. Consider addition of two BCD digits A and B. The resulting sum and carry, S and C are as defined below.
if ((a + b) < 10) S = a + b; C = 0; else S = (a + b) -10; C = 1;

This can be re-written as follows.
if ((a + b + 6) < 16) S = a + b; C = 0; else S = (a + b + 6) -16; C = 1;

This is equivalent to the following sequence.

- if ((a + b + 6) < 16) S = (a + b + 6); C = 0; else S = (a + b + 6) -16; C = 1;
- if (C == 0) S = S – 6;

This is same as performing binary addition A + B + 6 (instructions 1 and 3), then subtracting 6 (instruction 7) from the digits where there was no carry. Instructions 2, 4 and 5 identify BCD carries.

[2 marks for correct explanation. 1 mark for a fair attempt.]

3. Suppose a new processor is designed which is same as ARM but has the following two changes. (a) All DP instructions are made "2-address instructions". (b) Provision for shifting the second operand is removed from all DP instructions, but four independent shift instructions (again 2-address) are introduced. How would you rewrite the code of Q2 with these changes?

[3]

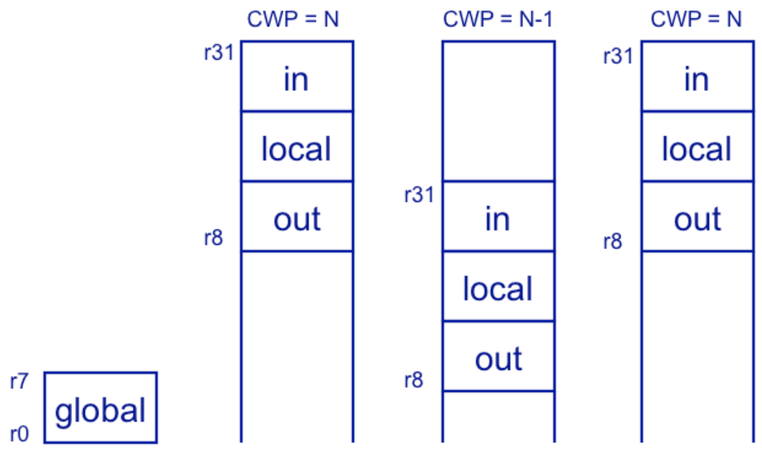
Solution:

Original 3-address code	Equivalent 2-address code	Marks
`func: add r2, r0, r6`	`func: mov r2, r0`	.5
	`add r2, r6`	
`eor r3, r2, r1`	`mov r3, r2`	.5
	`eor r3, r1`	
`add r2, r2, r1`	`add r2, r1`	.5
`eor r3, r2, r3`	`eor r3, r2`	
`bic r3, r7, r3`	`mov r4, r7`	.5
	`bic r4, r3`	
	`mov r3, r4`	
`orr r3, r3, r3 LSL #1`	`mov r4, r3`	.5
	`LSL r4, #1`	
	`orr r3, r4`	
`sub r0, r2, r3 LSR #3`	`LSR r3, #3`	.5
	`sub r2, r3`	
	`mov r0, r2`	
`mov pc, lr`	`mov pc, lr`	

4. Explain the concept of register windows. What benefit does it provide?

[3]

Solution:



[.5 marks for correct figure.]

Concept: It is a large array of registers, organized as a set of overlapping windows. At any time, one of the windows defines the set of registers that are accessible. A register called current window pointer (CWP) indicates which window is currently accessible. When a subroutine call is made, CWP moves to the next window and on return from the subroutine, it goes back to the previous window.

[1.5 marks for correct explanation. 1 mark for a fair attempt.]

Benefit: Depending upon the number of windows available, nested calls up to a certain depth can be made without any overhead of register saving and restoring. Beyond that depth, windows need to be saved (in stack) and restored.

[1 mark for correct explanation. .5 marks for a fair attempt.]