2 Solutions

Solution 2.1

2.1.1:

a.	`add f, g, h`
	`add f, f, i`
	`add f, f, j`

b.	`add f, h, #5` `add f, f, g`

2.1.2:

a.	3
b.	2

2.1.3:

a.	14
b.	10

2.1.4:

a.	`f = g + h`
b.	`f = g + h`

2.1.5:

a.	5
b.	5

Solution 2.2

2.2.1:

a.	`add f, f, f` `add f, f, i`
b.	`add f, j, #2` `add f, f, g`

2.2.2:

a.	2
b.	2

2.2.3:

a.	6
b.	5

2.2.4:

a.	f += h;
b.	f = 1-f;

2.2.5:

a.	4
b.	0

Solution 2.3

2.3.1:

a.	`add  f, f,  g`
	`add  f, f,  h`
	`add  f, f,  i`
	`add  f, f,  j`

b.	`add  f, f,  #2`
	`add  f, f,  #5`
	`sub  f, g,  f`

2.3.2:

a.	5
b.	2

2.3.3:

a.	17
b.	–4

2.3.4:

a.	f = h – g;
b.	f = g – f – 1;

2.3.5:

a.	1
b.	0

Solution 2.4

2.4.1:

a.	`LDR  R0,  [R7,#16]`
	`add  R0,  R0, R1`
	`add  R0,  R0, R2`

b.	`LDR  R5,  [R7,#16]`
	`LDR  R0,  [R5,#0]`
	`sub  R0,  R1, R0`

2.4.2:

a.	3
b.	3

2.4.3:

a.	4
b.	4

2.4.4:

a.	`f += g + h + i + j;`
b.	`f = A[1];`

2.4.5:

a.	no change
b.	no change

2.4.6:

a.	5 as written, 5 minimally
b.	2 as written, 2 minimally

Solution 2.5

2.5.1:

a.

Address	Data	
12	1	temp = Array[3];
8	6	Array[3] = Array[2];
4	4	Array[2] = Array[1];
0	2	Array[1] = Array[0];
		Array[0] = temp;

b.

Address	Data	
16	1	temp = Array[4];
12	2	Array[4] = Array[0];
8	3	Array[0] = temp;
4	4	temp = Array[3];
0	5	Array[3] = Array[1];
		Array[1] = temp;

2.5.2:

a.

```
LDR   R4,   [R6,#12]
LDR   R5,   [R6,#8]
STR   R5,   [R6,#12]
LDR   R5,   [R6,#4]
STR   R5,   [R6,#8]
LDR   R5,   [R6,#0]
STR   R5,   [R6,#4]
STR   R4,   [R6,#0]
```

b.

```
LDR   R4,   [R6,#16]
LDR   R5,   [R6,#0]
STR   R5,   [R6,#16]
STR   R4,   [R6,#0]
LDR   R4,   [R6,#12]
LDR   R5,   [R6,#4]
STR   R5,   [R6,#12]
STR   R4,   [R6,#4]
```

2.5.3:

a.	8 ARM instructions, +1 ARM inst. for every non-zero offset lw/sw pair (11 ARM inst.)
b.	8 ARM instructions, +1 ARM inst. for every non- zero offset lw/sw pair (11 ARM inst.)

2.5.4:

a.	305419896
b.	3199070221

2.5.5: (same as solution in MIPS version)

	Little-Endian		Big-Endian	
a.	Address	Data	Address	Data
	12	12	12	78
	8	34	8	56
	4	56	4	34
	0	78	0	12

b.	Address	Data	Address	Data
	12	be	12	0d
	8	ad	8	f0
	4	f0	4	ad
	0	0d	0	be

Solution 2.6

2.6.1:

Assuming word array.

a.	`LDR R0, [R7,#4]`
	`sub R0, R0, R1`
	`add R0, R0, R2`	

b.	`add R5, R7, R1, LSL #2`
	`LDR R5, [R5,#0]`
	`add R5, R6, R5, LSL #2`
	`LDR R0, [R5,#4]`
		`;Shift left two bits - to multiply by 4 (word array)`
		`;R5=b[g]`
		`;R5= (a[0]+b[g]*4)`

2.6.2:

a.	3
b.	4

2.6.3:

a.	4
b.	5

2.6.4:

a.	`f = 2i + h;`
b.	`f = A[g -3];`

2.6.5:

a.	`R0 = 110`
b.	`R0 = 300`

2.6.6:

a.

	Type	Opcode	Rd	Rn	Operand2	I
ADD r0, r0, r1	DP	4	R0	R0	R1	
ADD r0, r3, r2	DP	4	r0	r3	r2	
ADD r0, r0, r3	DP	4	r0	r0	r3	

b.						
SUB r6, r6, #20	DP	2	R6	R6	20	1
ADD r6, r6, r1	DP	4	r6	r6	r1	1
LDR r0, [r6,#8]	DT	24	r0	r6	8	

Solution 2.7

2.7.1:

a.	–1391460350
b.	–19629

2.7.2:

a.	2903506946
b.	4294947667

2.7.3:

a.	AD100002
b.	FFFFB353

2.7.4:

a.	01111111111111111111111111111111
b.	1111101000

2.7.5:

a.	7FFFFFFF
b.	3E8

2.7.6:

a.	80000001
b.	FFFFFC18

Solution 2.8

2.8.1:

a.	7FFFFFFF, no overflow
b.	80000000, overflow

2.8.2:

a.	60000001, no overflow
b.	0, no overflow

2.8.3:

a.	EFFFFFFF, overflow
b.	C0000000, overflow

2.8.4:

a.	overflow
b.	no overflow

2.8.5:

a.	no overflow
b.	no overflow

2.8.6:

a.	overflow
b.	no overflow

Solution 2.9

2.9.1:

a.	overflow
b.	no overflow

2.9.2:

a.	overflow
b.	no overflow

2.9.3:

a.	no overflow
b.	overflow

2.9.4:

a.	no overflow
b.	no overflow

2.9.5:

a.	1D100002
b.	6FFFB353

2.9.6:

a.	487587842
b.	1879028563

Solution 2.10

2.10.1:

a.	STR R5, [R3,#32]
b.	LDR R2, [R3,#32]

2.10.2:

a.	DT-type
b.	DT-type

2.10.3:

a.	E5953020
b.	E5823020

2.10.4:

a.	E0800005
b.	E5831004

2.10.5:

a.	DP-type
b.	DT-type

2.10.6:

a.	op=0x4, rd=0x0, rn=0x0, operand2=0x005, I=0x0
b.	op=0x24, rd=0x1, rn=0x3, offset=0x004

Solution 2.11

2.11.1:

a.	1110 0000 1000 0100 0010 0000 0000 0101two
b.	1110 0000 0100 0010 0011 0000 0000 0001two

2.11.2:

a.	3766755333
b.	3762434049

2.11.3:

a.	ADD r2, r4,r5
b.	SUB r3,r2,r1

2.11.4:

a.	DT-type
b.	DP-type

2.11.5:

a.	STR r5, [r6,#0]	
b. i)	SUB r5, r6, r0	#depending on the immediate bit set to 0 or 1
	SUB r5, r6,#0	

2.11.6:

a.	0xE5965000	# Binary value 11100101100101100101000000000000
b.	0xE0465000	# I bit set to 0, 11100000010001100101000000000000
	0xE2465000	# I bit set to 1, 11100010010001100101000000000000

Solution 2.12

2.12.1:

Cond	F	I	Opcode	S	Rn	Rd	Operand2/Offset	Total
4bits	2bits	1bit	4bits	1bit	3bits	3bits	10bits	28bits
4bits	2bits	1bit	4bits	1bit	4bits	4bits	10bits	30bits

2.12.2:

Cond	F	Opcode	Rn	Rd	Operand2/Offset	Total
4bits	2bits	6bits	3bits	3bits	10bits	28bits
4bits	2bits	6bits	4bits	4bits	10bits	30bits

2.12.3:

a.	less registers → less bits per instruction → could reduce code size less registers → more register spills → more instructions
b.	smaller constants → more load instructions (for bigger values) → could increase code size smaller constants → smaller opcodes → smaller code size

2.12.4:

a.	3800043520
b.	3850375200

2.12.5:

a.	`ADD r1, r0, #0`
b.	`LDR r1, [r0,#32]`

2.12.6:

a.	DP-type, op=0x4, Rd=0x1
b.	DT-type, op=0x24, rd=0x1

Solution 2.13

2.13.1:

a.	0x57755778
b.	0xFEFFFEDE

2.13.2:

a.	0x23456780
b.	0xEADFADE0

2.13.3:

a.	0x0000AAAA
b.	0x0000BFCD

2.13.4:

a.	0x00015B5A
b.	0x00000000

2.13.5:

a.	0x5b5a0000
b.	0x000000f0

2.13.6:

a.	0xEFEFFFFF
b.	0x000000F0

Solution 2.14

2.14.1:

a.	`MOV  r1,  r0,  LSR #5` `AND  r1,  r1,  #0x0001ffff`
b.	`MOV  r1,  r0,  LSL #10` `AND  r1,  r1,  #0xffff8000`

2.14.2:

a.	`MOV  r1,  r0` `AND  r1,  r1,  #0x0000000f`
b.	`MOV  r1,  r0,  LSR #14` `AND  r1,  r1,  #0x0003c000`

2.14.3:

a.	`MOV r1, r0, LSR #28`
b.	`MOV r1, r0, LSR #14` `AND r1, r1, #0x0001c000`

2.14.4:

a.	`MOV r2, r0, LSR #11`
	`AND r2, r2, #0x0000003f`
	`AND r1, r1, #0xffffffc0`
	`OR  r1, r1, r2`

b.	`MOV  r2,  r0, LSL #3`
	`AND  r2, r2, #0x000fc000`
	`AND  r1, r1, #0xfff03fff`
	`OR  r1, r1, r2`

2.14.5:

a.	MOV r2, r0 AND r2, r2, #0x0000001f AND r1, r1, #0xfffffffe0 OR r1, r1, r2
b.	MOV r2, r0, LSL #14 AND r2, r2, #0x0007c000 AND r1, r1, #0xfff83fff OR r1, r1, r2

2.14.6:

a.	MOV r2, r0, LSR #29 AND r2, r2, #0x00000003 AND r1, r1, #0xfffffffc OR r1, r1, r2
b.	MOV r2, r0, LSR #15 AND r2, r2, #0x0000c000 AND r1, r1, #0xffff3fff OR r1, r1, r2

Solution 2.15

2.15.1:

a.	0x0000a581
b.	0x00ff5a66

2.15.2:

a.	MVN r3, r3 AND r1, r2, r3
b.	MVN r4, r2 MVN r5, r3 AND r1, r2, r3 AND r4, r4, r5 OR r1, r1, r4

2.15.3:

a.	MVN r3, r3 AND r1, r2, r3	1110 00 0 (MVN Opcode) 0 0011 0011 0000 0000 0000 1110 00 0 (AND Opcode) 0 0010 0001 0000 0000 0011
b.	MVN r4, r2 MVN r5, r3 AND r1, r2, r3 AND r4, r4, r5 OR r1, r1, r4	1110 00 0 (MVN Opcode) 0 0010 0100 0000 0000 0000 1110 00 0 (MVN Opcode) 0 0011 0101 0000 0000 0000 1110 00 0 (AND Opcode) 0 0010 0001 0000 0000 0011 1110 00 0 (AND Opcode) 0 0100 0100 0000 0000 0101 1110 00 0 (OR Opcode) 0 0001 0001 0000 0000 0100

2.15.4:

a.	0x00000220
b.	0x00001234

2.15.5: Assuming r1 = A, r2 = B, r3= base of Array C

a.	LDR r0, [r3,#0] AND r1, r2, r0

b.	CMP r1, #0
	BEQ ELSE
	MOV r1, r2
	B END
	ELSE: LDR r2, [r3,#0]
	END:

2.15.6:

This is to be worked out in a manner similar to 2.15.3.

Solution 2.16

2.16.1:

a.	r2 = 0
b.	r2 = 0

2.16.2:

a.	r2 = 0
b.	r2 = 0

2.16.3:

	CMP r0, r1
	MOVHS r2,#0
	MOVLO r2,#2

2.16.4:

a.	r2 = 0
b.	r2 = 0

2.16.5:

a.	r2 = 0
b.	r2 = 0

Solution 2.17

2.17.1:

The answer is really the same for all. All of these instructions are either supported by an existing instruction, or sequence of existing instructions. Looking for an answer along the lines of, "these instructions are not common, and we are only making the common case fast".

2.17.2:

a.	DP type
b.	DP-type

2.17.3:

a.	`ABS:  MOV  r2,#0`
	`CMP  r3,#0`
	`SUBLT  r2, r2, r3`
	`MOVGE  r2, r3`
b.	`MOV  r1,#0`
	`CMP  r2, r3`
	`MOVGT  r1,#1`

2.17.4:

a.	`20 Flags set - Z`
b.	`200 Flags set - Z`

2.17.5:

a.

```
i = 10;
do {
    temp += 2;
    i = i - 1;
} while ( i > 0)
```

b.

```
i = 10;
do {
    B = 10;
    do {
        temp += 2;
        B = B - 1;
    } while ( B > 0 )
    i = i - 1;
} while ( i > 0)
```

2.17.6:

a.	5*N + 3
b.	43*N

Solution 2.18

2.18.1:

(DIAGRAM FROM THE MIPS Version)

2.18.2:

a.	`        MOV r2, #0`
	`        B TEST`
	`LOOP:  ADD r0, r0, r1`
	`        ADD r2, r2, 1`
	`TEST:  CMP r2, #10`
	`        BLT LOOP`

b.	`LOOP:  CMP r0, #10`
	`        BGE DONE`
	`        ADD r5, r1, r0`
	`        ADD r6, r4, r0, LSL #2`
	`        STR r5, [r6, #0]`
	`        ADD r0, r0, #1`
	`        B LOOP`
	`DONE:`

2.18.3:

a.	6 instructions to implement and 44 instructions executed
b.	7 instructions to implement and 2 instructions executed

2.18.4:

a.	601
b.	401

2.18.5:

a.	`for ( i = 100; i> 0; i- ) {`
	`      result += Mem Array [s0];`
	`      s0 += 1;`
	`}`

b.	`for ( i= 0; i<100; i+=2 ) {`
	`       result += Mem Array [s0 + i];`
	`       result += Mem Array [s0 + i + 1];`
	`}`

2.18.6:

a.

```
        MOV r1, #100
LOOP:   LDR r3, [r2, #0]
        ADD r4, r4, r3
        ADD r2, r2, #4
        SUBS r1, r1, #1
        BNE LOOP
```

b.

```
        ADD r1, r2,  #400
LOOP:   LDR r3, [r2, #0]
        ADD r4, r4, r3, LSL #1
        ADD r2, r2, #8
        CMP r1, r2
        BNE LOOP
```

Solution 2.19

2.19.1:

a.

```
compare:
          SUB sp, sp, #4
          STR lr, 0[sp,#0]
          ;#r0,r1 has input, r2 has ouput
          BL sub
          CMP r2,#0
          MOVGE r2,#1
          MOVLT r2,#0
          LDR lr, [sp,#0]
          ADD sp, sp, #4
          MOV pc,lr
          SUB r2, r0, r1
sub:      MOV pc,lr
```

b.

```
fib_iter:
          SUB sp, sp, #16
          STR lr, [sp,#12]
          STR r4, [sp,#8]
          STR r5, [sp,#4]
          STR r6, [sp,#0]
          MOV r4, r0
          MOV r5, r1
          MOV r6, r2
          MOV r3, r1  ;# r3 is used as output register..
          CMP r2 , #0
          BEQ exit
          ADD r0, r4, r5
          MOV r1, r4
          SUB r2, r6, #1
          BL fib_iter
exit:     LDR r6, [sp,#0]
          LDR r5, [sp,#4]
          LDR r4, [sp,#8]
          LDR lr, [sp,#12]
          ADD sp, sp, 16
          MOV pc,lr
```

2.19.2:

a.

```
compare:
        SUB r2, r0, r1
        #r0,r1 has input r2
        CMP r2,#0
        MOVGE r2,#1
        MOVLT r2,#0
        MOV pc,lr
```

b. Due to the recursive nature of the code, not possible for the compiler to in-line the function call.

2.19.3:

a.

```
after calling function compare:
old sp ->       0x7ffffffc      ???
sp->    -4      contents of register lr
after calling function sub:
old sp ->       0x7ffffffc      ???
        -4      contents of register lr
sp->    -8      contents of register lr  #return to compare
```

b.

```
after calling function fib_iter:
old sp ->       0x7ffffffc      ???
        -4      contents of register lr
        -8      contents of register r4
        -12     contents of register r5
sp->    -16     contents of register r6
```

2.19.4:

a.

```
f:   SUB   sp,sp,#8
     STR   lr,[sp,#4]
     STR   r4,[sp,#0]
     MOV   r4,r2 ; #r2 is scratch register so it is saved in r4
     BL    func
     MOV   r0,r3
     MOV   r1,r4
     BL    func
     LDR   lr,[sp,#4]
     LDR   r4,[sp,#0]
     ADD   sp,sp,#8
     MOV   pc,lr
```

b.

```
f:  SUB   sp,sp,#12
    STR   lr,[sp,#8]
    STR   r5,[sp,#4]
    STR   r4,[sp,#0]
    MOV   r4,r1     ; #r1 and r2 are saved in r4 and r5
    MOV   r5,r2
    BL    func
    MOV   r0,r4
    MOV   r1,r5
    MOV   r4,r3     ; #r3 is saved in r4
    BL    func
    ADD   r3,r3,r4
    LDR   lr,[sp,#8]
    LDR   r5,[sp,#4]
    LDR   r4,[sp,#0]
    ADD   sp,sp,#12
    MOV   pc,lr
```

2.19.5:

a.	We can use the tail-call optimization for the second call to func, but then we must restore lr and sp before that call. We save only one instruction.
b.	We can NOT use the tail call optimization here, because the value returned from f is not equal to the value returned by the last call to func.

2.19.6:

Register lr is equal to the return address in the caller function, registers sp and r4,r5 have the same values they had when function f was called, and registers r0,r1 and r2 can have an arbitrary value. For registers r0,r1 and r2, note that although our function f does not modify it, function func is allowed to modify it so we cannot assume anything about these registers after function func has been called.

Solution 2.20

2.20.1:

A.

```
FACT:   SUB  SP,SP,#8
        STR  LR,[SP,#4]
        STR  R0,[SP,#8]
        MOV  R2,R0
        CMP  R0,#2
        BGE  L1
        MOV  R1,#1
        ADD  SP,SP,#8
        MOV  PC,LR
L1:     SUB  R0,R0,#1
        BL   FACT
        MUL  R1,R2,R1
        LDR  R0,[SP,#4]
        LDR  LR,[SP,0]
        ADD  SP,SP,#4
```

b.

```
FACT:   SUB  SP,SP,#8
        STR  LR,[SP,#4]
        STR  R0,[SP,#8]
        MOV  R2,R0
        CMP  R0,#2
        BGE  L1
        MOV  R1,#1
        ADD  SP,SP,#8
        MOV  PC,LR
L1:     SUB  R0,R0,#1
        BL   FACT
        MUL  R1,R2,R1
        LDR  R0,[SP,#4]
        LDR  LR,[SP,0]
        ADD  SP,SP,#4
        MOV  PC,LR
```

2.20.2:

A.

16 instructions to execute the reccursive one and 13 instructions to execute the non reccursive one.

```
FACT:   SUB  SP,SP,#4
        LDR  LR,[SP,#4]
        MOV  R2,R0
        MOV  R3,#1
LOOP:   CMP  R2,#2
        BLT  DONE
        MUL  R3,R2,R3
        SUB  R2,R2,1
        B    LOOP
DONE:   MOV  R0,R3
        LDR  LR,[SP,#4]
        ADD  SP,SP,4
        MOV  PC,LR
```

b.

16 instructions to execute the reccursive one and 13 instructions to execute the non reccursive one.

```
FACT:   SUB  SP,SP,#4
        LDR  LR,[SP,#4]
        MOV  R2,R0
        MOV  R3,#1
LOOP:   CMP  R2,#2
        BLT  DONE
        MUL  R3,R2,R3
        SUB  R2,R2,1
        BLOOP
DONE:   MOV  R0,R3
        LDR  LR,[SP,#4]
        ADD  SP,SP,4
        MOV  PC,LR
```

2.20.3:

a.

Recursive version

```
FACT:       SUB  SP,SP,#8
            STR  LR,[SP,#4]
            STR  R0,[SP,#8]
            MOV  R2,R0
HERE:       CMP  R0,#2
            BGE  L1
            MOV  R1,#1
            ADD  SP,SP,#8
            MOV  PC,LR
L1:         SUB  R0,R0,#1
            BL   FACT
            MUL  R1,R2,R1
```

```
            LDR R0,[SP,#4]
            LDR LR,[SP,0]
            ADD SP,SP,#4
            MOV PC,LR

at label HERE, after calling function FACT with input of 4:

old SP ->   0xnnnnnnnn          ???
            -4                  contents of register LR
SP->        -8                  contents of register R0

at label HERE, after calling function FACT with input of 3:

old SP ->   0xnnnnnnnn          ???
SP->        -4                  contents of register LR
            -8                  contents of register R0
            -12                 contents of register LR
SP->        -16                 contents of register R0

at label HERE, after calling function FACT with input of 2:

old SP ->   0xnnnnnnnn          ???
            -4                  contents of register LR
            -8                  contents of register R0
            -12                 contents of register LR
            -16                 contents of register R0
            -20                 contents of register LR
SP->        -24                 contents of register R0

at label HERE, after calling function FACT with input of 1:

old SP ->   0xnnnnnnnn          ???
            -4                  contents of register LR
            -8                  contents of register R0
            -12                 contents of register LR
            -16                 contents of register R0
            -20                 contents of register LR
            -24                 contents of register R0
            -28                 contents of register LR
SP->        -32                 contents of register R0
```

b.

```
Recursive
version

FACT:       SUB  SP,SP,#8
            STR  LR,[SP,#4]
            STR  R0,[SP,#8]
            MOV  R2,R0
HERE:       CMP  R0,#2
            BGE  L1
            MOV  R1,#1
            ADD  SP,SP,#8
            MOV  PC,LR
L1:         SUB  R0,R0,#1
            BL  FACT
            MUL  R1,R2,R1
            LDR  R0,[SP,#4]
            LDR  LR,[SP,0]
            ADD  SP,SP,#4
            MOV  PC,LR
```

```
at label HERE, after calling function FACT with input of 4:
old SP ->  0xnnnnnnnn          ???
           -4                  contents of register LR
SP->       -8                  contents of register R0
at label HERE, after calling function FACT with input of 3:
old SP ->  0xnnnnnnnn          ???
           |-4                 contents of register LR
SP->       -8                  contents of register R0
           |-12                contents of register LR
SP->       -16                 contents of register R0
at label HERE, after calling function FACT with input of 2:
old SP ->  0xnnnnnnnn          ???
           -4                  contents of register LR
           -8                  contents of register R0
           -12                 contents of register LR
           -16                 contents of register R0
           -20                 contents of register LR
SP->       -24                 contents of register R0
at label HERE, after calling function FACT with input of 1:
old SP ->   0xnnnnnnnn         ???
           -4                  contents of register LR
           -8                  contents of register R0
           -12                 contents of register LR
           -16                 contents of register R0
           -20                 contents of register LR
           -24                 contents of register R0
           -28                 contents of register LR
SP->       -32                 contents of register R0
```

2.20.4:

```
FIB:    SUB   sp,sp,#12
        STR   lr,[sp,#8]
        STR   r2,[sp,#4]
        STR   r1,[sp,#0]
        CMP   r1,#1
        BGE   L1
        MOV   r0,r1
        B     EXIT
L1:     SUB   r1,r1,#1
        BL    FIB
        MOV   r2,r0
        SUB   r1,r1,#1
        BL    FIB
        ADD   r0,r0,r2
EXIT:   LDR   r1,[sp,#0]
        LDR   r2,[sp,#4]
        LDR   lr,[sp,#8]
        ADD   sp,sp,#12
MOV     pc,lr
```

2.20.5:

a.

```
        23 ARM instructions to execute non-recursive vs. 73 instructions
        to execute (corrected version of) recursion
        Non-recursive version:
FIB:    SUB  sp, sp, #4
        STR  lr, [sp,#0]
        MOV  r4, #1
        MOV  r5, #1
LOOP:   CMP  r1, #3
        BLT  EXIT
        MOV  r3, r4
        ADD  r4, r4, r5
        MOV  r5, r3
        SUB  r1, r1, #1
        B    LOOP
EXIT:   MOV  r0, r4
        LDR  lr, [sp,#0]
        ADD  sp, sp, #4
        MOV  pc,lr
```

2.20.6:

```
FIB:        SUB         sp,sp,#12
            STR         lr,[sp,#8]
            STR         r2,[sp,#4]
            STR         r1,[sp,#0]
HERE:       CMP         r1,#1
            BGE         L1
            MOV         r0,r1
            B           EXIT
L1:         SUB         r1,r1,#1
            BL          FIB
            MOV         r2,r0
            SUB         r1,r1,#1
            BL          FIB
            ADD         r0,r0,r2
EXIT:       LDR         r1,[sp,#0]
            LDR         r2,[sp,#4]
            LDR         lr,[sp,#8]
            ADD         sp,sp,#12
MOV         pc,lr
at label HERE, after calling function FIB with input of 4:
old sp ->   0xnnnnnnnn  ???
            -4          contents of register lr
            -8          contents of register r2
sp->        -12         contents of register r1
```

Solution 2.21

2.21.1:

a.

```
after entering function main:
old sp ->          0x7fffffc         ???
sp->               -4                contents of register lr
after entering function leaf_function: (for the first time)
old sp ->          0x7fffffc ???
                   -4                contents of register lr
sp->               -8                contents of register lr (return
                                     to main)
```

The stack pointer will grow further down for subsequent recursive calls.

b.

```
after entering function main:
old sp ->          0x7fffffc         ???
sp->               -4                contents of register lr
after entering function my_function:
old sp ->          0x7fffffc         ???
                   -4                contents of register lr
sp->               -8                contents of register lr (return
                                     to main)
global pointers:   100               my_global
0x10008000
```

2.21.2:

a.

```
MAIN:   SUB     sp,   sp,       #4
        STR     lr,   [sp,#0]
        MOV     r0,   #1
        BL      LEAF
        LDR     lr,   [sp,#0]
        ADD     sp,   sp, #4
        MOV
        pc,lr
LEAF:   SUB     sp,   sp,       #8
        STR     lr,   [sp,#4]
        STR     r4,   [sp,#0]
        ADD     r4,   r0,       #1
        CMP     r0,   #5
        BGT     DONE
        MOV     r0,   r4
        BL      LEAF
DONE:   MOV     r1,   r4
        LDR     r4,   [sp,#0]
        LDR     lr,   [sp,#4]
        ADD     sp,   sp,       #8
        MOV     pc,   lr
```

b.

```
MAIN:   SUB     sp,   sp,       #4
        STR     lr,   [sp,#0]
        MOV     r0,   #10
        MOV     r5,   #20
        LDR     r1,   [r4,#0];  #assume r4 has global variable base
        BL      FUNC
        MOV     r6,   r2
        LDR     lr,   [sp,#0]
        ADD     sp,   sp,       #4
        MOV     pc,   lr
FUNC:   SUB     r2,   r0,       r1
        MOV     pc,   lr
```

** 2.21.3 question does not exist

2.21.4:

a.	Register r4 is used to hold a temporary result without saving r4 first. To correct this problem, r0-r3 scratch registers can be used in place of r4 in the first two instructions. Note that a sub-optimal solution would be to continue using r4, but add code to save/restore it.
b.	The ADD and SUB instructions move the stack pointer in the wrong direction. Note that the ARM calling convention requires the stack to grow down. Even if the stack grew up, this code would be incorrect because lr and r4 are saved according to the stack-grows-down convention.

2.21.5:

a.	void f(int a, int b, int c, int d){
	  int x;
	  x= 2*(a–d)+c–b;
	}

b.	int f(int a, int b, int c){
	  return g(a, b)+c;
	}

2.21.6:

a.	The function returns 842 (which is 2*(1–30)+1000–100)
b.	The function returns 1500 (g(a, b) is 500, so it returns 500+1000)

Solution 2.22

2.22.1:

a.	65 20 98 121 116 101
b.	99 111 109 112 117 116 101 114

2.22.2:

a.	U+0041,	U+0020,	U+0062,	U+0079,	U+0074,	U+0065		
b.	U+0063,	U+006f,	U+006d,	U+0070,	U+0075,	U+0074,	U+0065,	U+0072

2.22.3:

a.	add
b.	shift

Solution 2.23

2.23.1:

a.

```
MAIN:   SUB     sp,   sp,     #12
        STR     lr,   [sp,#8]
        STR     r4,   [sp,#4]
        STR     r5,   [sp,#0]
        MOV     r4,   #0x30;  # '0'
        MOV     r5,   #0x39;  # '9'
        MOV     r1,   #0;     output register
        MOV     r3,   r0;     #copy of address so that
                              incoming address is not
                              changed.
LOOP:   LDR     r2,   [r3,#0]
        CMP     r2,   #0
        BE DONE
        CMP     r2,   r4,
        BLT nput_error
        CMP     r5,   r2
        BGT input_error
        SUB     r2,   r2,     r4
        CMP     r1,   #0
        BEQ FIRST
        MUL     r1,   r1,     #10
FIRST:  ADD     r1,   r1,     r2
        ADD     r3,   r3,     #1
```

```
          B LOOP
input_
error:
          MOV       r1,     #-1
DONE:     LDR       r5,     [sp,#0]
          LDR       r4,     [sp,#4]
          LDR       lr,     [sp,#8]
          ADD       sp,     sp,     #12
          MOV       pc,     lr
```

b.

```
MAIN:     SUB       sp,     sp,     #20
          STR       lr,     [sp,#16]
          STR       r4,     [sp,#12]
          STR       r5,     [sp,#8]
          STR       r6,     [sp,#4]
          STR       r7,     [sp,#0]
          MOV       r4,     $0, 0x41    ;# 'A'
          MOV       r5,     $0, 0x46    ;# 'F'
          MOV       r6,     $0,     0x30        ;# '0'
          MOV       r7,     $0,     0x39        ;# '9'
          MOV       r1,     #0
          MOV       r3,     r0;     #copy of address so that incoming
                                    address is not changed.
LOOP:     LDR       r2,     [r3,#0]
          CMP       r2,     #0
          BE DONE
          CMP       r2,     r6
          BLT input_error
          CMP       r7,     r2
          BGT HEX
          SUB       r2,     r2,     r6
          B DEC
HEX:      CMP r2,   r4
          BLT input_error
          CMP       r5,     r2
          BGT input_error
          SUB       r2,     r2,     r4
```

```
        ADD     r2,   r2,   #10
DEC:    CMP     r1,   #0
        BEQ FIRST
        MUL     r1,   r1,   #10
FIRST:  ADD     r1,   r1,   r2
        ADD     r3,   r3,   #1
        B LOOP
        Input_error :
        MOV     r1,   #0
DONE:   LDR     r7,   [sp,#0]
        LDR     r6,   [sp,#4]
        LDR     r5,   [sp,#8]
        LDR     r4,   [sp,#12]
        LDR     lr,   [sp,#16]
        ADD     sp,   sp,   20
        MOV     pc,   lr
```

Solution 2.24

2.24.1:

a.	@1000 0010	0	12	ff	ff
b.	@1000 0010	12	ff	ff	ff

2.24.2:

a.	@1000 0010	0	80	00	0
b.	@1000 0010	80	0	0	0

2.24.3:

a.	@1000 0010	0	11	55	55
b.	@1000 0010	11	55	55	55

Solution 2.25

2.25.1:

a.

```
MOV     r1,#4013          # 1111 (Rotate Immediate) 10101101(constant)
MOV     r2,#3073          # 0110 (Rotate Imm) 000000001 (constant)
OR      r1,r2,r1, LSL #2
OR      r1,r1,#2
```

b.

```
MVN     r1,#0
```

2.25.2:

Initialize the given constants into a register as shown in 2.25.1 and ADD or MOV it to the PC

2.25.3:

The 8-bit Operand2 field in the DP format can be subdivided into 2 fields: a 5-bit constant field on the right and a 3-bit rotate right field. This latter field rotates the 5-bit constant to the right by 4 times the value in the rotate field.

2.25.4:

Using MVN instruction.

2.25.5:

a.	The two MOV instructions								
	Cond	F	I	Opcode	S	Rn	Rd	Operand2/Offset	
	14	0	1	13	0	0	1	15	173
	14	0	1	13	0	0	2	6	1
	4bits	2bits	1bit	4bits	1bit	4bits	4bits	4bits	8bits
b.	Same as general instructiuon, nothing specific to Operand2/Offset field								

Solution 2.26

2.26.1:

(Refer to section 2.10)

a.	LDR	r0,[r1,r2]
b.	LDR	r2,[r0,r3, LSL#3]
c.	LDR	r3,[r1,#5]! Or LDR r3,[r1,r2]!
d.	STR	r0,[r2],#2 Or STR r0,[r2],r1

2.26.2:

All instructions are primarily DT type.

2.26.3:

The register offset modes are useful in indexing arrays.
The pre-indexed and post-indexed modes provide a compact format for accessing subsequent elements of an array. The compiler can generate compact code.

The flags are not set after the addition or subtraction operations with the post and pre-indexed instruction. In loops it might not be possible to use this when the result of the addition or subtraction is needed for deciding the branch.

2.26.4:

r0=0x0000000	r1=0x00009004	r2=0x05050509	r3=0x00000000

2.26.5:

No change in given memory locations except:

```
mem[0x05050505]= 0x00000000
```

Solution 2.27

2.27.1:

(Refer to section 2.10)

a.	`ADD r2, r0, #5`
b.	`ADD r2, r0, r1, LSL, #2`
c.	`LDR r2, [r0, r1, LSL, #2]!`
d.	`BEQ 4000`

2.27.2:

All instructions are DP type except last one which is BR type.

2.27.3:

Immediate addressing is useful for dealing with small constants.
Scaled register addressing allows barrel shift operations on one of the operands – giving raise to a shift and add type of instruction – useful for a multiply and accumulate kind of operation.
Pre-indexing and post-indexing help generate compact code.
PC-relative addressing provides flexibility in accessing constants, relative jumps etc.

Solution 2.28

2.28.1:

Number of instructions: 4

2.28.2:

This code using lock accesses a shared memory location and increments it.

1) During unlocking R3 may not be set to 0.

2) R2 is used in both SWP instruction and pointer in LDR instruction

2.28.3:

try:	MOV R3,#1
	SWP R4,R3,[R1,#0]
	CMP R4,#1
	BEQ try
	LDR R4,[R2,#0]
	ADD R3,R4,#1
	STR R3,[R2,#0]
	MOV R3,#0
	STR R3,[R1,#0]

2.28.4:

a.

Processor 1	Processor 2	Cycle	Processor 1		Mem	Processor 2	
			r2	r3	(r1)	r2	r3
		0	1	2	99	30	40
SWP R2,R3,[R1,#0]		1	0	99	2	30	40
	SWP R2,R3,[R1,#0]	2	0	99	40	0	2

b.

Processor 1	Processor 2	Cycle	Processor 1		Mem	Processor 2	
			r2	r3	(r1)	r2	r3
		0	2	3	1	10	20
try:	MOV R3,#1	1	2	3	1	10	1
try: MOV R3,#1	SWP R2,R3,[R1,#0]	2	2	1	1	0	1
SWP R2,R3,[R1,#0]		3	0	1	1	0	1
CMP R2,#1		4	0	1	1	0	1
BEQ try	CMP R2,#1	5	0	1	1	0	1
	BEQ try	6	0	1	1	0	1

Solution 2.29

2.29.1:

a.

```
try:    MOV R3,#1
        SWP R2,R3,[R1,#0]
        CMP R2,#1
        BEQ try
        LDR R2,[R4,#0]
        ADD R2,R2,R5
        STR R2,[R4,#0]
        MOV R3,#0
        STR R3,[R1,#0]
```

b.

```
try:    MOV R3,#1
        SWP R2,R3,[R1,#0]
        CMP R2,#1
        BEQ try
        LDR R2,[R4,#0]
        CMP R2,R5
MOVGE   R2,R5
        STR R2,[R4,#0]
        MOV R3,#0
        STR R3,[R1,#0]
```

2.29.2:

a.

```
Try:    LDR R3,[R4,#0]
        ADD R6,R3,R5
        SWP R2,R6,[R4,#0]
        CMP R2,R3
        BEQ DONE
        STR R2, [R4,#0]
        B TRY
DONE:
```

b.

```
try:    LDR R3,[R4,#0]
        MOV R6, R3
        CMP R6,R5
        MOVGE R6,R5
        SWP R2,R6,[R4,#0]
        CMP R2,R3
        BEQ DONE
        STR R2, [R4,#0]
        B TRY
DONE:
```

2.29.3:

The Best case performance of the codes are:

a.	2.29.1 requires 9 clock cyles whereas 2.29.2 requires only 5 clock cycles.
b.	2.29.1 requires 10 clock cyles whereas 2.29.2 requires only 6 clock cycles.

2.29.4:

a.	When two processors try to execute their swp instructions after loading the current value of shvar, only one succeeds with the swp, making the other processor to restore, loop back and load the fresh shvar value and continue swapping.
b.	Similar argument can be done here. Difference is that, the new value of shvar is loaded and minimum is calculated until the swp instruction succeeds.

2.29.5:

The address of shvar variable is only used and not its value in a register because, it is a shared variable.The value of the variable x is used and not its address because, it is local to a processor.

2.29.6:

Atomic operation of two shared variable is not possible using a swp instruction because, the swp instruction is a register to memory operation. That is, it does not support a memory to memory swp operation.

Solution 2.30

2.30.1 :

The pseudo instruction LDR r0, #constant can be implemented by
(i) loading from memory (PC-relative addressing),
(ii) using MOV or MVN instructions for small constants.

2.30.2 :

A Mov or MVN with shift would be appropriate for such constants.

MOV r0,#fff, LSL #4

Solution 2.31

2.31.1:

a.

	Text Size	0x440
	Data Size	0x90
Text	Address	Instruction
	0x00400000	LDR R0, [R3,#0x8000]
	0x00400004	BL 0x0400140
	...	...

	0x00400140	STR R1, [R3,#0x8040]
	0x00400144	BL 0x0400000
	...	...
Data	0x10000000	(X)
	...	...
	0x10000040	(Y)

b.

	Text Size	0x440
	Data Size	0x90
Text	Address	Instruction
	0x00400000	LDR R0, [R3,#0x1000]
	0x00400004	ORR R1,R0,#0
	0x00400008	BL 0x0400140
	...	...
	0x00400140	STR R0, [R3,#0x8040]
	0x00400144	B 0x04002C0
	...	...
	0x004002C0	MOV PC,LR
	...	...
Data	0x10000000	(X)
	...	...
	0x10000040	(Y)

2.31.2:

0x8000 data, 0xFC00000 text. However, because of the size of the beq immediate field, 218 words is a more practical program limitation.

Solution 2.32

2.32.1:

a.

```
swap:   ADD    R4,R0,R1, LSL#2
        LDR    R6,[R4,#0]
        ADD    R5,R0,R2, LSL #2
        LDR    R7,[R5,#0]
        STR    R7,[R4,#0]
        STR    R6,[R5,#0]
        MOV    PC,LR
```

b.

```
swap:  LDR    R4,[R0,#0]
       LDR    R5,[R0,#4]
       STR    R5,[R0,#0]
       STR    R4,[R0,#4]
       MOV    PC,LR
```

2.32.2:

a. Pass j+1 as a third parameter to swap. We can do this by adding an "ADD R2,R1,#1" instruction right before "BL swap".

b. Pass the address of v[j] to swap. Since that address is already in R12 at the point when we want to call swap, we can replace the two parameter-passing instructions before "BL swap" with a simple "mov R0,R12".

2.32.3:

a.

```
swap:
ADD     R4, R4, R0    ; No LSL
LDRB    R6, [R4,#0]    ; Byte-sized load
ADD     R5, R5, R0    ; No LSL
LDRB    R7, [R5,#0]
STRB    R7, [R4,#0]    ; Byte-sized store
STRB    R6, [R5,#0]
MOV     PC, LR
```

b.

```
swap:
LDRB   R4, [R0,#0]
LDRB   R5, [R0,#1]    ; Offset is 1, not 4
STRB   R5, [R0,#0]
STRB   R4, [R0,#1]
MOV    PC, LR
```

Solution 2.33

2.33.1:

a.

```
find:       MOV    R9,#0
loop:       CMP    R9, R1
            BEQ    done
            ADD    R4, R0, R9, LSL #2
            LDR    R4,[R4,#0]
            CMP    R4,R2
            BNE    skip
            MOV    R0,R9 ; the return value is put to R0
            MOV    PC,LR
skip:       ADD    R9,R9,#1
            B      loop
done:       MOV    R0,-1
            MOV    PC,LR
```

b.

```
count:    MOV    R9,#0
          MOV    R4,#0
loop:     CMP    R4,R1
          BEQ    done
          ADD    R5,R0,R4, LSL #2
          LDR    R5,[R5,#0]
          CMP    R5,R2
          BNE    skip
          ADD    R9,R9,#1
skip:     ADD    R4,R4,#1
          B      loop
done:    MOV     R0,R9 ; the return value is put to R0
         MOV     PC,LR
```

2.33.2:

a.

```
int find(int *a, int n, int x){
    int *p;
    for(p=a;p!=a+n;p++)
       if(*p==x)
       return p-a;
    return -1;
}
```

b.

```
int count(int *a, int n, int x){
    int res=0;
    int *p;
    for(p=a;p!=a+n;p++)
        if(*p==x)
        res=res+1;
    return res;
}
```

2.33.3:

a.

```
find:      MOV    R4,R0
           ADD    R5,R0, R1, LSL #2
loop:      CMP    R4,R5
           BEQ    done
           LDR    R6,[R4,#0]
           CMP    R6,R2
           BNE    skip
           SUB    R9,R4,R0
           MOV    R0,R9, LSR #2
           MOV    PC,LR
skip:      add    R4,R4,#4
           b      loop
done:      MOV    R0,#-1
           MOV    PC,LR
```

b.	`find:  MOV  R9,#0`
	`  mov  R4,R0`
	`  add  R4,R0,R1, LSL #2`
	`loop:  CMP  R4,R5`
	`  BEQ  done`
	`  LDR  R6,[R4,#0]`
	`  CMP  R6,R2`
	`  BNE  skip`
	`  add  R9,R9,#1`
	`skip:  add  R4,R4,#4`
	`  b  loop`
	`done:  MOV  R0,R9`
	`  MOV  PC,LR`

2.33.4:

	Array-based	Pointer-based
a.	8	7
b.	8	7

2.33.5:

	Array-based	Pointer-based
a.	5	7
b.	6	7

Solution 2.34

2.34.1:

a.	`  addi  $s0,  $0,  10`
	`LOOP:  add  $s0,  $s0,  $s1`
	`  addi  $s0,  $s0,  -1`
	`  bne  $s0,  $0,  LOOP`

b.	`  sll  $s1,  $s2,  28`
	`  srl  $s2,  $s2,  4`
	`  or  $s1,  $s1,  $s2`

2.34.2:

a.	ADD, ADDI - all MIPS register-register(R) instruction format BNE - an MIPS jump (J) instruction format
b.	SLL,SRL, OR - an MIPS register-register instruction format

2.34.3:

a.	`CMP  r0,  r1` `BMI  FARAWAY`
b.	`ADD  r0,  r1,  r2`

2.34.4:

a.	CMP - an ARM DP type format BMI - an ARM branch instruction format
b.	ADD - an ARM DP type format

Solution 2.35

2.35.1:

a.	register operand
b.	register + offset and update register

2.35.2:

a.	`lw $s0, ($s1)`

b.	`lw $s1, ($s0)`
	`lw $s2, 4($s0)`
	`lw $s3, 8($s0)`

2.35.3:

a.	`addi $s0, $0, TABLE1`
	`addi $s1, $0, 100`
	`xor $s2, $s2, $s2`
	`ADDLP: lw $s4, ($s0)`
	`addi $s2, $s2, 4`
	`addi $s0, $s0, 4`
	`addi $s1, $s1, -1`
	`bne $s1, $0, ADDLP`

b.	`sll $s1, $s2, 28`
	`srl $s2, $s2, 4`
	`or $s1, $s1, $s2`

2.35.4:

a.	8 ARM vs. 8 MIPS instructions
b.	1 ARM vs. 3 MIPS instructions

2.35.5:

a.	ARM 0.67 times as fast as MIPs
b.	ARM 2 times as fast as MIPs

Solution 2.36

2.36.1:

a.	`sll $s1, $s1, 3` `add $s3, $s2, $s1`

b.	`sll $s4, $s1, 29`
	`srl $s1, $s1, 3`
	`or $s1, $s1, $s4`
	`add $s3, $s2, $s1`

2.36.2:

a.	`addi $s3, $s2, 64`
b.	`addi $s3, $s2, 64`

2.36.3:

a.	`sll $s1, $s1, 3` `add $s3, $s2, $s1`

b.	`sll $s4, $s1, 29`
	`srl $s1, $s1, 3`
	`or $s1, $s1, $s4`
	`add $s3, $s2, $s1`

2.36.4:

a.	ADD r3, r2, #1
b.	ADD r3, r2, 0x8000

Solution 2.37

2.37.1:

a.	`mov edx, [esi+4*ebx]`		`edx=memory(esi+4*ebx)`

b.	`START:`	`mov ax, 00101100b`
		`mov cx, 00000011b`
		`mov bx, 11110000b`
		`and ax, bx`
		`or ax, cx`
			`char ax = 00101100b;`
			`char bx = 11110000b;`
			`char cx = 00000011b;`
			`ax = ax && bx;`
			`ax = ax || cx;`

2.37.2:

a.		`add R8, R8, R6, LSL #2` `LDR R7, [R8,#0]`

b.	`START:`	`MOV R4, #0x2c`
		`MOV R6, 0x03`
		`MOV R5, 0xf0`
		`and R4, R4, R5`
		`or R4, R4, R6`

2.37.3:

a.	`mov edx, [esi+4*ebx]`	6, 1, 1, 8, 8
b.	`add eax, 0x12345678`	4, 4, 1, 32

2.37.4:

a.	`MOV R4, #2`
	`ADD R0, R1, R0, LSL R4`
	`LDR R0, [R0,#0]`

b.	`LDRH R0, #0x1234`
	`MOV R1, R0, LSL #16`
	`ORR R1, #0x5678`
	`ADD R2, R1`

Solution 2.38

2.38.1:

a.	This instruction copies ECX bytes from an array pointed to by ESI to an array pointer by EDI. An example C library function that can easily be implemented using this instruction is memcpy.
b.	This instruction copies ECX elements, where each element is 4 bytes in size, from an array pointed to by ESI to an array pointer by EDI.

2.38.2:

a.	`loop: LDRB R4,[R2,#0]`
	`STRB R4,[R1,#0]`
	`SUB R0,R0,#1`
	`ADD R1,R1,#1`
	`ADD R2,R2,#1`
	`CMP R0,#0`
	`BNE loop`

b.	`loop: LDR R4,[R2,#0]`
	`STR R4,[R1,#0]`
	`SUB R0,R0,#1`
	`ADD R1,R1,#4`
	`ADD R2,R2,#4`
	`CMP R0,#0`
	`BNE loop`

2.38.3:

	x86	ARM	Speedup
a.	5	7	1.4
b.	5	7	1.4

2.38.4:

	ARM code	Code size comparison
a.	f: ADD R0,R0,R1 MOV PC,LR	ARM: 2*4=8 bytes x86: 11 bytes

b.	f: LDR R4,[R0,#0]
	LDR R5,[R1,#0]
	add R4,R4,R5
	STR R4,[R0,#0]
	STR R4,[R1,#0]
	MOV PC,LR
		ARM: 6*4=24 bytes
		x86: 19 bytes

2.38.5: In ARM, we can fetch the next two consecutive instructions by reading the next 8 bytes from the instruction memory. In x86, we only know where the second instruction begins after we have read and decoded the first one, so it is more difficult to design such a processor that executes multiple instructions in parallel. Or, in x86 we need to read the maximum number of bytes corresponding to two instructions, and use more complex decode circuitry.

2.38.6: Under these assumptions, using x86 leads to a significant slowdown (the speedup is well below1):

	ARM Cycles	x86 Cycles	Speedup
a.	2	11	0.18
b.	6	19	0.32

Solution 2.39

2.39.1:

a.	0.86 seconds
b.	0.78 seconds

2.39.2: Answer is no in all cases. Slows down the computer.

CCT = clock cycle time
ICa = instruction count (arithmetic)
ICls = instruction count (load/store)
ICb = instruction count (branch)
new CPU time = 0.75* old ICa * CPIa * 1.1 * oldCCT

+ oldICls * CPIls * 1.1 * oldCCT
+ oldICb * CPIb * 1.1 * oldCCT

The extra clock cycle time adds sufficiently to the new CPU time such that it is not quicker than the old execution
time in all cases.

2.39.3:

a.	113.16%	121.13%
b.	106.85%	110.64%

2.39.4:

a.	3
b.	2.65

2.39.5:

a.	0.6
b.	1.07

2.39.6:

a.	0.2
b.	0.716666667

Solution 2.40

2.40.1:

a.	In the first iteration R5 is 0 and the LDR fetches a[0]. After that R5 is 1 and the LDR instruction uses a non-aligned address.
b.	In the first iteration R4 and R5 point to a[0] ,b[0], so the LDR and STR instructions access a[0], b[0], and then a[0] as intended. In the second iteration R4 and R5 point to the next byte in a[0] and b[1], respectively, instead of pointing to a[1] and b[1]. Thus the first LDR uses a non-aligned address. Note that the computation for R6 (address of a[n]) does not cause any error because that address is not actually used to access memory.

2.40.2:

a.	Yes
b.	Yes.

2.40.3:

a.

```
f:   MOV R4, #0         ; ret=0
     MOV R5, #0         ; i=0
L:   ADD R6, R5, R0     ; &(a[i])
     LDR R6, [R6, #0]   ; read a[i]
     CMP R6, R2         ; if( a[i]==x)
     BNE S
```

	ADD R4, R4, #1 ; ret++
	S: ADD R5, R5, #4 ; i=i+4 Incrementing 'i' by 4 to align to word
	CMP R5, R1 ; repeat if i!=n
	BNE L
	MOV R0, R4
	MOV PC, LR ; return ret

b.	f: MOV R4, R0 ; p=a
	MOV R5,R1 ; q=b
	ADD R6, R0, R2, LSL #2 ; We must multiply n by 4 to get address of end of 'a'
	L: LDR R7,[R4, #0] ; read *p
	LDR R8, [R5, #0] ; read *q
	ADD R7, R7 ,R8 ; *p + *q
	STR R7, [R4, #0] ; *p = *p + *q
	ADD R4, R4, #4 ; p=p+4
	ADD R5, R5, #4 ; q = q + 4
	CMP R4, R6, #1 ; repeat if p!= &(a[n])
	BNE L
	MOV PC, LR ; return

2.40.4: At the exit from my_alloc, the SP register is moved to "free" the memory that is returned to main. Then my_init() writes to this memory to initialize it. Note that neither my_init nor main() access the stack memory in any other way until sort() is called, so the values at the point where sort() is called are still the same as those written by my_init:

a.	0, 0, 0, 0, 0
b.	5, 4, 3, 2, 1

2.40.5: In main, register R9 becomes 5, then my_alloc is called. The address of the array v "allocated" by my_alloc is 0xfe8, because in my_alloc SP was saved at 0xffc, and then 20 bytes (4*5) were reserved for array arr (SP was decremented by 20 to yield 0xfe8). The elements of array v returned to main are thus a[0] at 0xfe8, a[1] at 0xfec, a[2] at 0xff0, a[3] at 0xff4, and a[4] at 0xff8. After my_alloc returns, SP is back to 0x1000. The value returned from my_alloc is 0xfe8 and this address is placed into the R10 register. The my_init function does not modify SP, R9 or R10. When sort() begins to execute, SP is 0x1000, R1 is 5, R0 is 0xffe7. The sort() procedure then changes SP to 0xfec (0x1000 minus 20), and writes R2 to memory at address 0xffec (this is where a[1] is, so a[1] becomes value of R2), writes R3 to memory at address 0xfff0 (this is where a[2] is, so a[2] becomes value of R3), writes R6 to memory address 0xfff4 (this is where a[3] is, so a[3] becomes value of R6),

writes R7 to memory address 0xfff8 (this is where a[4] is, so a[4] becomes value of R7), and writes the return address to 0xfffc, which does not affect values in array v. Now the values of array v are:

a.	0 value of R2 value of R3 value of R6 value of R7
b.	5 value of R2 value of R3 value of R6 value of R7

Note: Since values of registers r2, r3, r6, r7 are not known before first iteration, we do not specify the values in the answer.

2.40.6: When the sort() procedure enters its main loop, the elements of array v are sorted without any interference from other stack accesses. The resulting sorted array is

a.	0, 1, 5, 7, 0xffe8
b.	1, 5, 5, 7, 0xffe8

Unfortunately, this is not the end of the chaos caused by the original bug in my_alloc. When the sort() function begins restoring registers, LR is read the (luckily) unmodified location where it was saved. Then R2 is read from memory at address 0xffec (this is where a[1] is), R3 is read from address 0xfff0 (this is where a[2] is), R6 is read from address 0xfff4 (this is where a[3] is), and R7 is read from address 0xfff8 (this is where a[4] is). When sort() returns to main(), registers R2 and R3 are supposed to keep n and the address of array v. As a result, after sort() returns to main(), n and v are:

a.	n=1, v=5 So v is a 1-element array of integers that begins at address 5
b.	n=5, v=5 So v is a 5-element array of integers that begins at address 5

If we were to actually attempt to access (e.g. print out) elements of array v in the main() function after this point, the first LDR would result in a non-aligned address memory access. If ARM were to tolerate non-aligned accesses, we would print out whatever values were at the address v points to (note that this is not the same address to which my_init wrote its values).