CS1022 Exercise Set #4
Lab Exercise: Game Clock

1 Exercise Set 1

1.1 Addressing Modes

(a) (i) Load a word from memory at address 0xA1000080 + 8 = 0xA1000088 into register R0.

(ii) Load a byte from memory at address 0xA1000080 + 1 = 0xA1000081 into register R0. R1 is updated with 0xA1000081.

(iii) Load a word from memory at address 0xA1000080. R1 is updated with 0xA1000080 + 4 = 0xA1000084.

(iv) Store a word from R0 to memory at 0xA1000080 + (0x42 <<2) = 0xA1000188

1.2 Arrays

(a) (i)

```
        LDRH    R4, [R10, R5, LSL #1]
```

(ii)

```
        ADD     R8, R6, #2
        LDRH    R7, [R10, R8, LSL #1]
        STRH    R7, [R10, R6, LSL #1]
```

(iii)

```
        MOV     R7, R5, LSL #4          ; idx = i * 16
        ADD     R7, R7, R6              ; idx += j
        LDR     R8, [R11, R7, LSL #2]

        MOV     R7, R6, LSL #4          ; idx = j * 16
        ADD     R7, R7, R5              ; idx += i
        LDR     R9, [R11, R7, LSL #2]

        MUL     R4, R8, R9
```

(b) Remove

```
last = size - 1;
current = remove;
while (current < last) {
    arr[current++] = arr[current];
}
```

```
        SUB     R4, R2, #1
        MOV     R5, R1
whDel
        CMP     R5, R4
        BHS     eWhDel
        LDR     R6, [R0, R5, LSL #2]
        ADD     R5, R5, #1
        STR     R6, [R0, R5, LSL #2]
        B       whDel
eWhDel
```

(c) Matrix Multiplication

```
        MOV     r4, #0                  ; i = 0;
wh1
        CMP     r4, r3                  ; while (i < N)
        BHS     endwh1                  ; {
        MOV     r5, #0                  ;   j = 0;
wh2
        CMP     r5, r3                  ;   while (j < N)
        BHS     endwh2                  ;   {
        MOV     r7, #0                  ;     r = 0;
        MOV     r6, #0                  ;     k = 0;
wh3
        CMP     r6, r3                  ;     while (k < N)
        BHS     endwh3                  ;     {
        MUL     r8, r4, r3              ;       idx = (i * N)
        ADD     r8, r8, r6              ;               + k;
        LDR     r9, [r1, r8, LSL #2]    ;       tmpA = Memory.Byte[pA + (idx * 4)];
        MUL     r8, r6, r3              ;       idx = (k * N)
        ADD     r8, r8, r5              ;               + j;
        LDR     r10, [r1, r8, LSL #2];          tmpB = Memory.Byte[pB + (idx * 4)];
        MUL     r9, r10, r9             ;       tmpA = tmpA * tmpB;
        ADD     r7, r7, r9              ;       r = r + tmpA;
        ADD     r6, r6, #1              ;       k = k + 1;
        B       wh3                     ;     }
endwh3
        MUL     r8, r4, r3              ;     idx = (i * N)
        ADD     r8, r8, r5              ;             + j;
        STR     r7, [r0, r8, LSL #2]    ;     Memory.Byte[pZ + (idx * 4)];
        ADD     r5, r5, #1              ;     j = j + 1;
        B       wh2                     ;   }
endwh2
        ADD     r4, r4, #1              ;   i = i + 1;
        B       wh1                     ; }
endwh1
```

2 Exercise Set 2

2.1 Stacks

2.1.1 Stack Operations with LDR and STR

(a)

```
        STR     R4, [SP, #-4]!
        STR     R5, [SP, #-4]!
        STR     R6, [SP, #-4]!
```

(b) Diagram of memory showing stack state after above operations

(c) (Note the order – reverse of above)

```
        LDR     R6, [SP], #4
        LDR     R5, [SP], #4
        LDR     R4, [SP], #4
```

(d) Diagram of memory showing stack state after above operations

2.1.2 Stack Operations with LDM and STM

(a)

```
        STMFD SP!, {R4-R6}
```

(b) Note, the LDM instruction takes care of the order of registers for us. The rule is the lowest numbered register is always loaded/stored from/to the lowest address. So, the order of registers in the list doesn't matter in either LDM or STM!

```
        LDMFD SP!, {R4-R6}
```

2.1.3 Understanding LDM and STM

(1) `STMFD SP!, R4, R6, R8-R11`

Diagram of memory showing stack state after above operation

(2) `LDMFD SP, R10, R11`

Note – no !

Diagram of memory showing stack state after above operation

(3) `LDMFD SP!, R8, R10, R11, R4, R6`

Note – order doesn't matter, we will still reverse the PUSH performed by the first instruction!!

Diagram of memory showing stack state after above operation

2.1.4 Value to Decimal String (using a stack)

We will assume that the value to convert to a decimal ASCII string is stored in R1. We will also assume that space has been set aside for the string at the address in R0. We will use the system stack to store the ASCII characters as they are produced.

```
start
        LDR     R0, =deststr     ; buffer = deststr;
        LDR     R1, =365         ; val = 365;

        MOV     R4, #0           ; push NULL character on to stack so
        STMFD   SP!, {R4}        ;   we know when to stop popping below!

                                 ; NOTE: We will push and pop word size
                                 ;   values. Not strictly necessary here
                                 ;   but it's good practice when using
                                 ;   the system stack

        MOV     R4, R1           ; let's not destroy the original input

doDigits                         ; do {

        MOV     R5, #0           ;   eventual quotient

                                 ;   (INEFFICIENTLY!!) DIVIDE BY 10
whDiv
        CMP     R4, #10          ;   while (remainder >= 10)
        BLO     eWhDiv           ;   {
        SUB     R4, R4, #10      ;     remainder = remainder - 10;
        ADD     R5, R5, #1       ;     quotient++;
        B       whDiv            ;   }

                                 ;   STORE CHARACTER ON STACK
eWhDiv
        ADD     R6, R4, #'0'     ;   char = remainder + '0';
        STMFD   SP!, {R6}        ;   push(char);
        MOV     R4, R5           ;   remainder = quotient;
        CMP     R4, #0           ; while (remainder != 0);
        BNE     doDigits

doPop                            ; do {
        LDMFD   SP!, {R4}        ;   char = pop();
        STRB    R4, [R0], #1     ;   Memory.Byte[buffer++] = char;
        CMP     R4, #0           ; } while (char != NULL);
        BNE     doPop

stop    B       stop
```

2.2 Subroutines

2.2.1 Subroutine Basics

The program will not work because, when we branch to sub1 (`BL sub1`), the processor stores the return address in LR. Then, when we branch to sub2 (`BL sub2`) the processor will overwrite the sub1 return address.

We can return from sub2 but when we attempt to return from sub1, the return address has been lost (overwritten) and we will actually return to the point from which we called sub2!!!

The solution is to save the LR on the system stack either on entry to sub1 or, at least, before we

`BL sub2`.

2.2.2 Subroutines and the System Stack

Stack illustration(s)

2.2.3 Subroutine Interfaces

(a) (i) zeroMemory

```
; zeroMemory - zero a range of addresses in memory
; Parameters:
;   R0: start address of range to zero
;   R1: length number of bytes to zero (unsigned word)
```

(ii) factorial

```
; factorial - compute x!
; Parameters:
;   R0: x (unsigned word)
; Return value:
;   R0: x!
```

(iii) power

```
; power - compute x^y
; Parameters:
;   R0: x (signed word)
;   R1: y (unsigned word)
; Return value:
;   R0: x^y
```

(iv) quadratic

```
; quadratic - evaluate a quadratic function of the form f(x) = ax^2+bx+c
; Parameters:
;   R0: a (signed word)
;   R1: b (signed word)
;   R2: c (signed word)
;   R3: x (signed word)
; return value:
;   R0: f(x) (signed word)
```

(b) (i) zeroMemory (e.g. zero 1024 bytes starting at the address in R4)

```
        MOV     R0, R4
        LDR     R1, =1024
        BL      zeroMemory
```

(ii) factorial (e.g. compute 6!)

```
        MOV     R0, #6
        BL      factorial
```

(iii) power (assume R4, R7, R9 have some meaning in a wider context)

```
        MOV     R0, R4
        MOV     R1, R7
        BL      power
        MOV     R9, R0
```

(iv) quadratic (assume R4, R5, R7, R9, R11 have some meaning in a wider context)

```
        MOV     R0, R4
        MOV     R1, R7
        MOV     R2, R9
        MOV     R3, R11
        BL      quadratic
        MOV     R5, R0
```

2.2.4 Writing Subroutines

```
; divide - divide a dividend by a divisor
; Parameters:
;   R2: dividend [in]
;         R3:   divisor [in]
; Return value:
;         R0:   quotient [out]
;         R1:   remainder [out]
;
; NOTE: above interface is not AAPCS-compliant!!!
divide
        STMFD   sp!, {LR, R2, R3}

        MOV     R0, #0
        MOV     R1, R2

wh1     CMP     R1, R3
        BLO     endwh1
        SUB     R1, R1, R3
        ADD     R0, R0, #1
        B       wh1

endwh1
        LDMFD   sp!, {PC, R2, R3}
```

3 Exercise Set 3

3.1 Floating-Point Numbers

(a) (i) 4.56789×10^5

(ii) 4.25×10^{-6}

(b) (i) 1.5

(ii) 8.3125

(iii) 0.9375

(c) (i) 1111.01

(ii) 1010.0011

(iii) $1000.11100110011\underline{0011}$

(iv) 0.00010100011110101110 ...

(d) (i) 1.11101×2^3

(ii) 1.0100011×2^3

(iii) $1.0001110011001100110011 \ldots \times 2^3$

(iv) $00001.0100011110101110 \ldots \times 2^{-4}$

(e) (i) 0x41740000

(ii) 0x41430000

(iii) 0x410e6666

(iv) 0x3da3d70a

3.2 Floating-Point Arithmetic

(i) $1.25 + 0.75 = 2$ (0x40000000)

(ii) $24.5 + 18.75 = 43.25$ (0x422d0000)

(iii) $20.75 + 0.1875 = 20.9375$ (0x41a78000)