MEF UNIVERSITY
Your Freedom in Learning

EE 306 Microprocessors Spring 2016 – 2017

Tuba Ayhan

Midterm Exam

14.04.2017

1. Greatest common divisor (GCD) algorithm can be expressed in C as:

```
int gcd(int a, int b)
{
    while (a != b)
      {
        if (a > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

Below, the algorithm is implemented in ARM assembly language i. Using braches, ii. Using conditional execution of instructions.

a. (10 pnt) Fill the GCD assembly programs.
b. (2 pnt) State which register is used for the variables A and B.
c. (8 pnt) Draw the flow chart.

i. The gcd function with conditional execution of branches only:

```
gcd:    CMP     r0, r1
        B___     STOP
        BLT     less
        SUB     ____, r0, r1
        B       gcd
Less:
        ____    _____, r1, r0
        B       ____
STOP:   ___ STOP
```

ii. By using the conditional execution feature of the ARM instruction set, the gcd is implemented as:

```
gcd:    ____     r0, r1
        SUB___   r0, r0, r1
        SUB___   r1, r1, r0
        B___     gcd
```

EE 306 Microprocessors Spring 2016 – 2017

Tuba Ayhan

Midterm Exam

14.04.2017

2. (30 pnt) The instruction

```
AND R4, R4, R8
```

is stored in location 0x37C0 in the memory. At the time this instruction is fetched, registers R4 and R8 contain the values 0x1000 and 0x1004, respectively. The memory locations 0x1000 and 0x1004 contain the values 0x3214 and 0xA105.

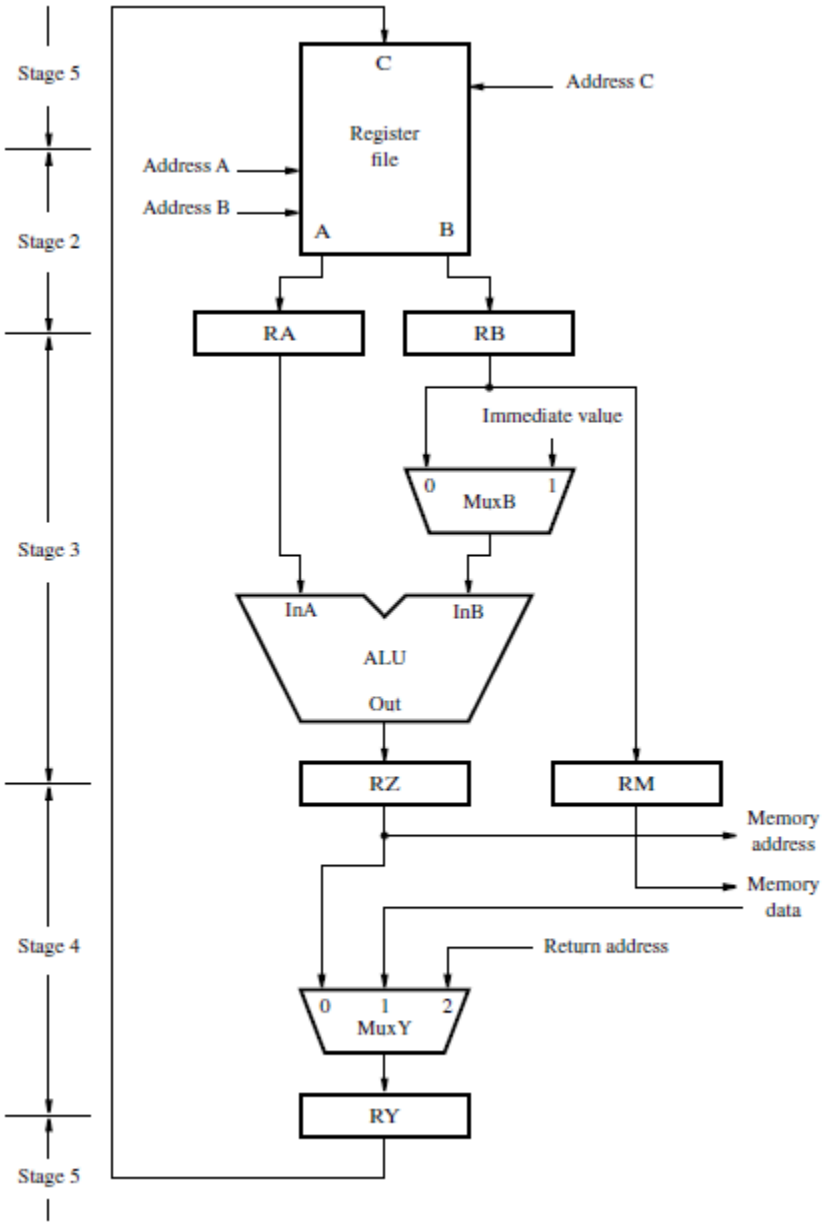
a. Write the 5 execution steps for this instruction. Give the values in registers R4, RA, RB, RM, RZ, and RY whenever they change as this instruction is executed.

b. Repeat a. for the instruction

```
LDR R4, [R8]
```

Note: This is a 16-bit processor with RISC instruction set. Datapath flow is given in the figure. AND instruction performs bitwise logical and on the sources (sr1, sr2) and the result is given in the destination (de):

```
AND de, sr1, sr2.
```



Ref: ARM Assembly Language_ Fundamentals and Techniques (2nd ed.) [Hohl & Hinds 2014-10-20]

1. Assembly example - FACTORIAL CALCULATION

$$n! = \prod_{i=1}^{n} i = n(n-1)(n-2)\ldots(1)$$

For a given value of n, the algorithm iteratively multiplies a current product by a number that is one less than the number it used in the previous multiplication. The code continues to loop until it is no longer necessary to perform a multiplication, that is, when the multiplier is equal to zero.

After a comparison instruction (CMP), flags in the CPSR are set and can be combined so that we might say one value is less than another, greater than another, etc. In order for one signed value to be greater than another, the Z flag must be clear, and the N and V flags must be equal. From a programmer's viewpoint, you simply write the condition in the code, e.g., GE for greater-than-or-equal, LT for less-than, or EQ for equal.

1. Write an assembly program to calculate n!. Test it for n = 10. Keep the result in R7.
2. Complete the following:

```
        MOV __, __// load n into r6
        MOV __, __ // hold the result in a register. if n=0, at least n!=1
loop     ___ __, #0 // did you multiply n numbers? If not, use conditional
                                                    instructions below:
        MUL__ __, __, __  // multiply temporary result with next number: n
        SUB__ __, __, #1 // decrement n
        B__ loop ; do another mul if counter!= 0
End     B End ; stop program
```

2. Assembly example – Logic instructions

```
LDR r0, =0xF631024C // load some data
LDR r1, =0x17539ABD // load some data
EOR r0, r0, r1 //
EOR r1, r0, r1 //
EOR r0, r0, r1 //
```

Explain the code piece above. What does it do?

3. Addressing modes – LDR

Syntax

```
LDR{type}{T}{cond} Rt, [Rn {, #offset}]
LDR{type}{cond} Rt, [Rn, #offset]!
LDR{type}{T}{cond} Rt, [Rn], #offset
LDR{type}{cond} Rt, [Rn, +/-Rm {, shift}]
LDR{type}{cond} Rt, [Rn, +/-Rm {, shift}]!
LDR{type}{T}{cond} Rt, [Rn], +/-Rm {, shift}
```

where:

type can be any one of:

- B – unsigned Byte. (Zero extend to 32 bits on loads.)
- SB – signed Byte. (Sign extend to 32 bits.)
- H – unsigned Halfword. (Zero extend to 32 bits on loads.)
- SH – signed Halfword. (Sign extend to 32 bits.)

or omitted, for a Word load.

Describe the contents of registers R1-R4 after the following instructions complete. Remember, the processor is byte addressable.

```
            LDR R0,=Test_num
            LDRSB r1, [r0]
            LDRSH r2, [r0]
            LDR r3, [r0]
            LDRB r4, [r0]
Test_num: .word 0xFF03FC86
```

4. Adressing modes

Calculate the **effective address** of the following instructions if register

```
r3 = 0x00000040 and register r4 = 0x00000020:
a. STRH r9, [r3, r4]
b. LDRB r8, [r3, r4, LSL #3]
c. LDR r7, [r3], r4
d. STRB r6, [r3], r4, ASR #2
```

Calculate the value of r3 after the instructions are executed.

5. Adressing modes

Assume register r3 contains 0x8000. What would the register contain after executing the following instructions, sequentially?

```
a. STR r6, [r3, #12]
b. STRB r7, [r3], #4
c. LDRH r5, [r3], #8
d. LDR r12, [r3, #12]!
```

6. Logic and arithmetic:

Write a compare routine to compare two 64-bit unsigned values A and B. Higher and lower half of A is kept in r3 and r4, respectively. A = [r3 r4] and similarly, B = [r5 r6]. Compare A and B, then write the greater number in [r8 r9].

7. Branch

Code the following IF-THEN statement using ARM instructions:

```
if (r2 != r7)
    r2 = r2 - r7;
  else
    r2 = r2 + r4;
```

8. Branch

Write a routine to reverse the word order in a block of memory. The block contains 32 words of data.

9. Subroutines and stacks

Study the STM and LDM instructions below. Then answer the questions.

STM (Store Multiple registers) writes one or more registers to consecutive addresses in memory to an address specified in a base register.

Syntax

```
STM{addr_mode}{cond} Rn{!},reglist{^}
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted.
- IB – Increment address Before each transfer (ARM only).
- DA – Decrement address After each transfer (ARM only).
- DB – Decrement address Before each transfer.

LDM (Load Multiple registers) loads one or more registers from consecutive addresses in memory at an address specified in a base register.

Syntax

```
LDM{addr_mode}{cond} Rn{!},reglist{^}
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted
- IB – Increment address Before each transfer (ARM only)
- DA – Decrement address After each transfer (ARM only)
- DB – Decrement address Before each transfer.

a. if register r6 holds the address 0x8000, what address holds the value in register r0, r4, r7, and the link register after `STMIA r6, {r0, r4, r7, lr}` is executed.

b. Assume that memory and registers r0 through r3 appear as in the table →

Describe the memory and register contents after executing the instruction

`LDMIA r3!, {r0, r1, r2}`

Address			Register
0x8010	0x00000001	0x13	r0
0x800C	0xFEEDDEAF	0xFFFFFFFF	r1
0x8008	0x00008888	0xEEEEEEEE	r2
0x8004	0x12340000	0x8000	r3
0x8000	0xBABE0000		

Solutions

Midterm1 :

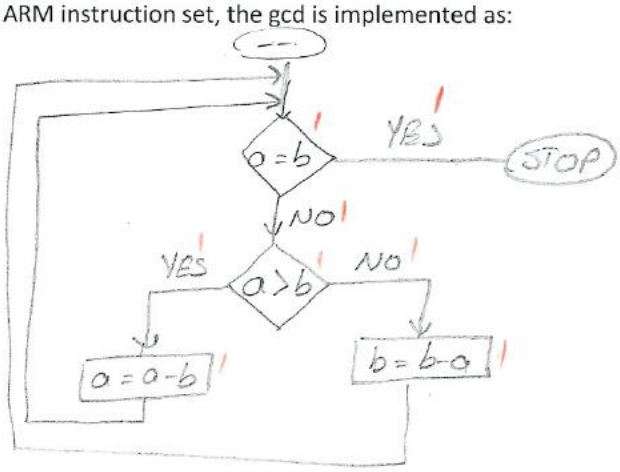
i. The gcd function with conditional execution of branches only:

```
gcd:    CMP     r0, r1
        BEQ     STOP
        BLT     less
        SUB     r0, r0, r1
        B       gcd
Less:
        SUB     r1, r1, r0
        B       gcd
STOP:   B STOP
```

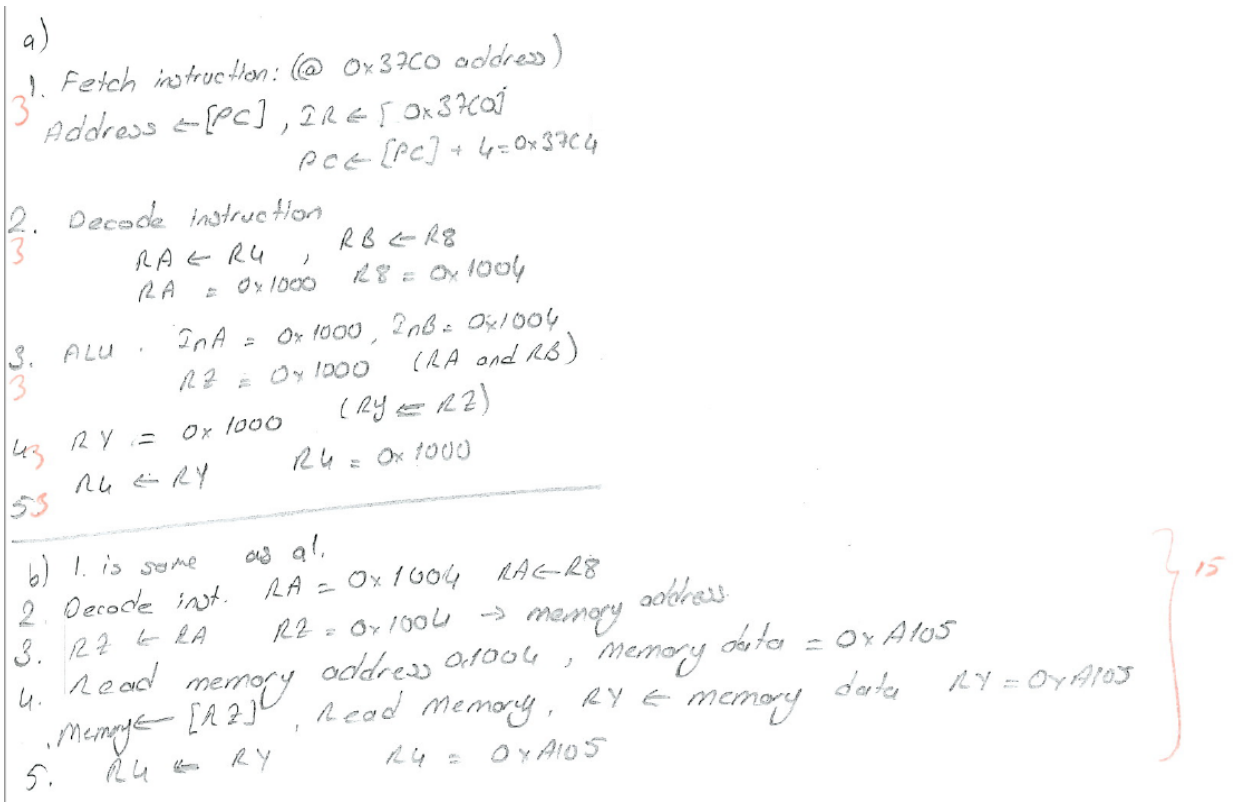
r0: A 1pnt
r1: B 1pnt

ii. By using the conditional execution feature of the ARM instruction set, the gcd is implemented as:

```
gcd:    CMP     r0, r1
        SUBGT   r0, r0, r1
        SUBLT   r1, r1, r0
        BNE     gcd
```



Midterm 2:



1. Solution Factorial Calculation:

```
        MOV R6, #10// load n into r6
        MOV R7, #1 // hold the result in a register. if n=0, at least n!=1
loop    CMP  R6, #0 // did you multiply n numbers? If not, use conditional
                                                    instructions below:
        MULGT R7, R7, R6  // multiply temporary result with next number: n
        SUBGT R6, R6, #1 // decrement n
        BGT loop ; do another mul if counter!= 0
End     B End ; stop program
```

2. Solution Logic instructions

Swapping register contents.

3. Solution addressing modes - LDR

R1: 0xFFFFFF86, R2: 0xFFFFFC86, R3: 0xFF03FC86, R4: 0x00000086

4. Solution addressing modes

a. 0x60, b. 0x140, c. 0x40, d. 0x40. At the end: R3 = 0x68

5. Solution addressing modes

a. 0x8000, b. 0x8004, c.0x800c, d.0x8018

6. Logic and arithmetic

```
mov r3,#0 A = 0x0000000F
mov r4,#15
mov r5,#0 B = 0x0000000C
```

```
mov r6,#10
cmp r3,r5
cmpeq r4,r6
movge r8,r3
movge r9,r4
movlt r8,r5
movlt r9,r6
```

7. Logic and arithmetic

```
cmp r2,r7
subne r2,r2,r7
addeq r2,r2,r4
```

8. Logic and arithmetic

```
mov r0,#32 // number of words in the list
ldr r1, =list // r1: first address
ldr r2, =listson // r2: last address: think about different ways to
get the last address.

loop: cmp r1,r2
bgt end
ldr r3, [r1] //read the first number in orig. list
ldr r4, [r2] //read the last number in orig list.
str r4,[r1],#4  //first and last numbers swap then update first
address
str r3,[r2],#-4 //first and last numbers swap then update last
address
B loop

end: b end
list: .word
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29,30,31
listson: .word 32
```

9. Subroutines and stacks

a. 0x8000 holds r0, 0x8004 holds r4, 0x8008 holds r7, and 0x800c holds LR.
b. R0 = 0xBABE0000, R1 =0x12340000, R2 = 0x00008888.