CMPSCI 201 – Fall 2004

Midterm #1 – Answers

<1> 10 Points – Short Essay Answer – The 8088 is primarily a CISC processor design, and the ARM is primarily RISC. The 6502 is such an early design that it is difficult to place it squarely in one camp or the other. Decide whether the 6502 is more of an evolutionary ancestor of CISC designs or of RISC designs, and write a short paragraph to justify your decision, with examples where appropriate. What characteristics (if any) are similar to characteristics of the 8088, and what (if any) are similar to those of the ARM?

ANSWER: The 6502 has similarities to RISC designs in the small overall number of instructions and relative simplicity of each instruction. It has similarities to modern CISC designs in that instructions are variable length (1, 2, or 3 bytes) and that certain registers have specialized uses (the X and Y registers are not general purpose accumulators, but are primarily used as array index registers). The zero-page (8-bit) and absolute (16-bit) addressing scheme of the 6502 also resembles the segment and offset approach of the x86 line, but not to the extreme taken by Intel.

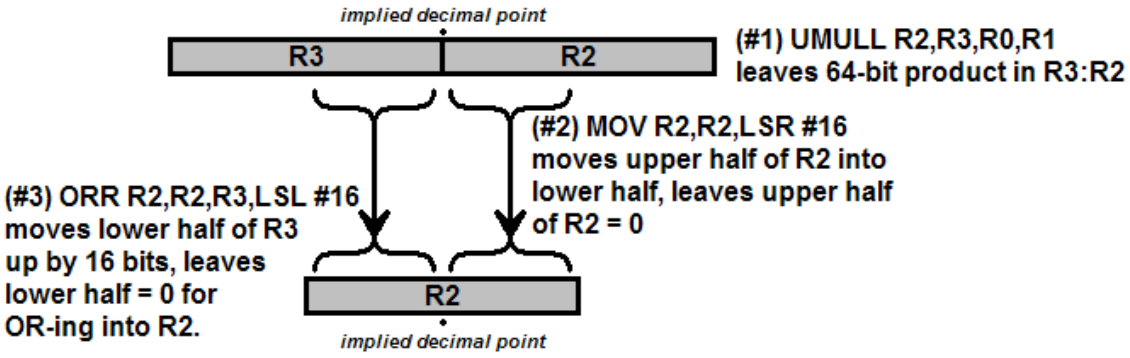
According to on-line encyclopedia Wikipedia (`http://www.wikipedia.org`) the designers of the ARM were inspired by the design of the 6502, so in some sense it is a direct ancestor of this RISC chip!

GRADING: As long as you properly support your position, I will accept either answer. There may be some valid reasons not covered here. Remove 5 points for any explanation which does not contain an explicit example to justify the decision. Remove all 10 points if there is no explanation at all.

<2> 10 Points – The ARM `UMULL` (unsigned multiplication) instruction generates a 64-bit unsigned product from two 32-bit source operands. The general form of the instruction is `UMULL` R_{DL},R_{DH},R_M,R_S where R_M and R_S represent the source operand registers, and the result of the multiplication goes into destination registers R_{DL} (low word) and R_{DH} (high word). Assuming that registers `R0` and `R1` contain the two operands represented as ***unsigned fixed point numbers***, with 16 bits to the left of the decimal point and 16 bits to the right, write a code fragment that places the normalized 32-bit fixed point product (i.e., the middle 32 bits) into register `R2`. Use as many other registers as you need, and do not worry about saving or restoring the values of those other registers.

ANSWER:

```
UMULL R2,R3,R0,R1        R3:R2 := R0 * R1.
MOV   R2,R2,LSR #16      Move high half of R2 to low.
ORR   R2,R2,R3 LSL #16   Combine low half of R3 into
                         high half of R2.  Op-codes
                         ADD and EOR will work too.
```



Here is a workable alternative:

```
UMULL R2,R3,R0,R1        R3:R2 := R0 * R1.
MOV   R3,R3,LSL #16      Move low half of R3 to high.
ORR   R2,R3,R2 LSR #16   Move high half of R2 into
                         low half, then combine with
                         R3.  Op-codes ADD and EOR
                         will work too.
```

In both cases the `UMULL` instruction may be written as:

```
UMULL R2,R3,R1,R0        R3:R2 := R1 * R0.
```

GRADING: Reserve 2 points for getting the `UMULL` specified correctly. Reserve 4 points for successfully moving the upper half of `R2` into the lower half. Reserve 4 points for successfully combining the lower half of `R3` into the upper half of `R2`.

<3> 10 Points – Which of the following constants can I load into a register in one `MOV` instruction, such as `MOV R0,#_____` (immediate value) or `MOV R0,#___,___` (immediate value right-rotated by an even number between 0 and 30)?

1. 5 Yes (The number is in the range [0…255] and will fit into the eight bits available for the constant part of an immediate value. The amount of right-rotation is zero.)

2. 192 Yes (The number is in the range [0…255] and will fit into the eight bits available for the constant part of an immediate value. The amount of right-rotation is zero.)

3. 259 No (The binary number is `100000001`, which will not fit into the eight bits available for the constant part of an immediate value.)

4. 768 Yes (The binary number is `1100000000`, which is the number 3 right-rotated by 24 bit positions.)

5. 1025 No (The binary number is `10000000001`, which will not fit into the eight bits available for the constant part of an immediate value.)

GRADING: Each answer (yes or no) is worth 2 points. No explanations are necessary.

<4> 10 Points – In each of the following problems you are to multiply the contents of integer register `R0` by a constant value, in **<u>one</u>** instruction, without using any other registers, and without using any explicit multiplication instruction such as `MUL`, `MLA`, or `UMULL`. In questions 1 and 2 you are to find ***two different methods*** for multiplication by +3. For question 3 you may assume that the initial value in `R0` is always positive.

1. `R0 := R0 × 3` `ADD R0,R0,R0,LSL #1` *(R0+2×R0)*

2. `R0 := R0 × 3` `RSB R0,R0,R0,LSL #2` *(4×R0−R0)*

3. `R0 := R0 × 1½` `ADD R0,R0,R0,LSR #1` *(R0+R0÷2)*

4. `R0 := R0 × 4` `MOV R0,R0,LSL #2` *(4×R0)*

5. `R0 := R0 × -3` `SUB R0,R0,R0,LSL #2` *(R0−4×R0)*

GRADING: Each answer is worth 2 points. In each case remove 1 point for a basically correct instruction containing an incorrect shift-type or amount.

<5> 10 Points – Convert the decimal number 26.25 into (a) binary scientific notation (i.e., $\pm 1.xxxx \times 2^{Y}$), and (b) the equivalent binary single-precision floating-point representation.

(a) $11010.01 = 1.101001 \times 2^{4}$

(b) Sign = 0, Biased exponent = 4+127 = $131_{10} = 10000011_2$, Mantissa = .101001

Final answer: 0 10000011 10100100000000000000000

GRADING: Part (a) is worth 4 points. Remove 1 point for a nearly correct fraction (an error in a couple of bits); 2 points if the fraction is massively wrong. Remove 2 points for an incorrect exponent.

Part (b) is worth 6 points. Remove 1 bit for an incorrect sign bit. Remove 2 points for incorrectly translating the exponent into a biased exponent. Remove 2 points for incorrectly translating the fraction into the mantissa (not removing the leading 1 digit, for example). Remove 1 point for not zero-filling the right bits of the mantissa. DO NOT REMOVE POINTS here if the numbers in part (a) are incorrect but where those numbers are correctly converted into floating point.

<6> 10 Points – For this problem you will need to use the `MUL` R_D`,`R_M`,`R_S instruction. Create a complete ARM subroutine to evaluate the integer polynomial $y = 12x^2 + 8x + 3$, where the value of x is passed in through `R0` and the result y is passed back through `R1`. You must maintain full transparency on all registers other than `R1`.

ANSWER:

```
Poly      STR  R2,SaveR2          Save R2
          MUL  R2,R0,R0           R2 := X^2
          MOV  R1,R2,LSL #2       R1 := 4×R2 = 4X^2
          ADD  R1,R1,R1,LSL #1    R1 := 3×R1 = 3×4X^2 = 12X^2
          ADD  R1,R1,R0,LSL #3    R1 := 12X^2 + 8X
          ADD  R1,R1,#3           R1 := 12X^2 + 8X + 3
          LDR  R2,SaveR2          Restore R2
          MOV  PC,LR              Return

SaveR2    DCD  0                  Reserve space for R2
```

You can also use push `R2` onto the stack (`STR R2,[SP,#-4]!`) and pop it from the stack (`LDR R2,[SP],#4`) instead of storing into and loading from an explicit memory location.

It is also possible to load 12 into `R2` (but as shown in the appendix of the textbook you cannot multiply by an immediate constant).

Here is a legal variation:

```
Poly    STR  R2,[SP,#-4]!        Push R2
        MUL  R1,R0,R0            R1 := X^2
        MOV  R2,#12              R2 := 12
        MUL  R1,R2,R1            R1 := R1×R2 = 12X^2
        ADD  R1,R1,R0,LSL #3     R1 := 12X^2 + 8X
        ADD  R1,R1,#3            R1 := 12X^2 + 8X + 3
        LDR  R2,[SP],#4          Pop R2
        MOV  PC,LR               Return
```

GRADING: Accept any form that is functionally correct. Remove 1 point for each minor syntax error, including multiplying by a constant (as in `MUL R1,R2,#12`). Remove 2 points for not maintaining transparency for registers such as `R2` (there is no need to save or restore `LR` since there are no nested subroutine-calls).

Note that `MUL R1,R2,R1` is legal, but `MUL R1,R1,R2` is not (the first operand register cannot be the same as the destination register). Since we have not covered this in class, no penalty will be incurred for the illegal form.

<7> 10 Points – Translate the following high-level pseudo-code fragment into ARM assembly code, using as few instructions as possible.

```
R0 := 0
R1 := 20
Repeat
     If R1 is Even Then R0 := R0 + R1
     R1 := R1 - 3
Until R1 <= 0
```

ANSWER:

```
        MOV   R0,#0          R0 := 0
        MOV   R1,#20         R1 := 20
RP1     TST   R1,#1          Set Z if low bit = 0 (even)
        ADDEQ R0,R0,R1       If Z=1 Then R0 := R0 + R1
        SUBS  R1,R1,#3       R1 := R1 - 3, set flags
        BGT   RP1            If Result > 0 Then GoTo RP1
```

It is legal to use `AND R2,R1,#1` here instead of the `TST` instruction, but this approach is inefficient as it burns one more register unnecessarily. It is also inefficient to use the following code for the loop termination as it uses one too many instructions:

```
SUB R1,R1,#3
CMP R1,#0
BGT RP1
```

-alternative approach-

A smart compiler would unroll the loop since it is controlled by a constant, then recognize that there are only a handful of values added into `R0` (20, 14, 8, and 2) and that the value in `R1` always ends up at -1. Thus, the entire block of code could legally be replaced by the two-instruction sequence:

```
MOV    R0,#44
MOV    R1,#-1
```

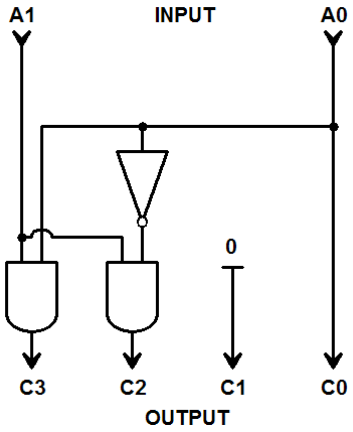
You can't get away with this trick if the initial value of `R1` was derived from a variable or another register, instead of from a constant!

GRADING: As before, accept any answer that is functionally correct. Remove 1 point for each minor syntax error, or for any instruction which ***must*** be changed in order to bring the code into compliance with the high-level pseudo-code. In future exams any parts of the answer denoted above as "inefficient" will have points deducted, but not here.

<8> 10 Points –

8 points – Trace the following gate circuit and show the output for all possible input behaviors.

A1	A0	C3	C2	C1	C0
0	0	0	0	0	0
0	1	0	0	0	1
1	0	0	1	0	0
1	1	1	0	0	1



GRADING: ½ point each answer. Round to the nearest whole point (i.e., both -½ and -1 both are treated as -1).

2 points – What is the purpose of this circuit?

ANSWER: The binary number in $C_3C_2C_1C_0$ is the **square** of the number in A_1A_0.

GRADING: All or nothing. The answer must describe C as the square of A.

<9> 10 Points – Write a complete, correct, ARM assembly language subroutine to print isosceles triangles made out of stars and blanks (spaces). The size of the triangle to print is given by input parameter `N`, which is to be passed in through the `R0` register. Here are some sample values for `N` and the triangles printed out as a result:

```
N=1     N=2     N=3       N=4         ...
*         *       *         *
        * *     * *       * *
              * * *     * * *
                      * * * *
```

Three ASCII-based subroutines are available, called `Print_Blank`, `Print_Star`, and `Print_LF` (remember that line-feed = ASCII 10), that may be called by your subroutine; all three are completely transparent. I will be looking for efficiency in your code, so pay particular attention to the overall number of instructions, execution time, register usage, etc. As always, your subroutine must be completely transparent with respect to its register usage, but the only `LDR/STR` instructions you are allowed to use are for saving and restoring registers.

ANSWER: In problems such as this one it often makes sense to write the desired routine in high-level pseudo code before writing any assembly language. For this problem, the pseudo code (in Pascal notation) would be:

```
For Row := 1 To N Do
     Begin
          For Col := 1 To N-Row Do
               Begin
                    Print_Blank ;
               End ;
          For Col := 1 To N Do
               Begin
                    Print_Star ;
                    Print_Blank ;
               End ;
          Print_LF ;
     End
```

Considering that in Pascal FOR-loops terminate automatically if the starting value is greater than the stopping value, this pseudo code works correctly for any value of `N`, including 0 and negative numbers (i.e., no output). The structure of the code shows that we need two loop counters, one for the outer loop and another for the two inner loops (which are independent from one another). In assembly language each loop counter will be kept in a register. We will also need a third register to hold the value of `N-Row` (the number of leading blanks for each row of output).

Remember that `N` is in `R0`. Let us keep `Row` in `R1`, `Col` in `R2`, and `N-Row` in `R3`. It is trivial to convert this structure into a series of equivalent While-loops. Rewriting the pseudo code in terms of the registers instead of variables and expressions, and While-loops instead of For-loops gives us the following:

```
R1 := 1 ;
While R1 <= R0 Do
     Begin
          R3 := R0 - R1 ;

          R2 := 1 ;
          While R2 <= R3 Do
               Begin
                    Print_Blank ;
                    R2 := R2 + 1 ;
               End ;

          R2 := 1 ;
          While R2 <= R0 Do
               Begin
                    Print_Star ;
                    Print_Blank ;
                    R2 := R2 + 1 ;
               End ;

          Print_LF ;

          R1 := R1 + 1 ;
     End
```

Now it remains to convert the pseudo code into assembly language. Along with the `LR` register, `R1`, `R2`, and `R3` are the only other registers we will change; these are the ones that need saving and restoring for transparency purposes. We need not save `R0` as its value does not change. As you should see from the pseudo code above, translating each statement into the equivalent ARM assembly language is a straightforward, mechanical process.

```
Triangle  STR  LR,SaveLR      ; Save all registers
          STR  R1,SaveR1      ;
          STR  R2,SaveR2      ;
          STR  R3,SaveR3      ;

          MOV  R1,#1          ; R1 := 1
While1    CMP  R1,R0          ; While R1 <= N Do
          BGT  End1           ;    Begin

          SUB  R3,R0,R1       ;         R3 := N - Row
          MOV  R2,#1          ;         R2 := 1
While2    CMP  R2,R3          ;         While R2 <= R3 Do
          BGT  End2           ;              Begin
          BL   Print_Blank    ;                Print_Blank
          ADD  R2,R2,#1       ;                R2 := R2 + 1
          B    While2         ;
End2                          ;              End

          MOV  R2,#1          ;         R2 := 1
While3    CMP  R2,R0          ;         While R2 <= N Do
          BGT  End3           ;              Begin
          BL   Print_Star     ;                Print_Star
          BL   Print_Blank    ;                Print_Blank
          ADD  R2,R2,#1       ;                R2 := R2 + 1
          B    While3         ;
End3                          ;              End

          BL   Print_LF       ;         Print_LF
          ADD  R1,R1,#1       ;         R1 := R1 + 1

          B    While1         ;
End1                          ;    End

          LDR  R3,SaveR3      ; Restore all registers
          LDR  R2,SaveR2      ;
          LDR  R1,SaveR1      ;
          LDR  PC,SaveLR      ; Return

SaveLR    DCD  0              ;
SaveR1    DCD  0              ;
SaveR2    DCD  0              ;
SaveR3    DCD  0              ;
```

This is the expected code result for this problem. It is allowed to save registers to and restore registers from the stack, as long as the code is written correctly.

If the assumption is made that `N` is always greater than zero (which in class was stated as something you *cannot* assume), the first and third While-loops can be optimized into Repeat-loops. Furthermore, changing a requirement that the number of leading blanks is in the range `[0…N-1]` into the range `[1…N]` allows us to rewrite the second While-loop as a Repeat-loop as well. The two inner Repeat-loops can now be rewritten as count-down instead of count-up loops, gaining some efficiency there by using tests against zero instead of test against positive integers. Here is the new pseudo code:

```
R1 := 1 ;
Repeat
     R3 := R0 - R1 + 1 ;

     R2 := R3 ;
     Repeat
          Print_Blank ;
          R2 := R2 - 1 ;
     Until R2 <= 0 ;

     R2 := R0 ;
     Repeat
          Print_Star ;
          Print_Blank ;
          R2 := R2 - 1 ;
     Until R2 <= 0 ;

     Print_LF ;

     R1 := R1 + 1 ;
Until R1 > R0 ;
```

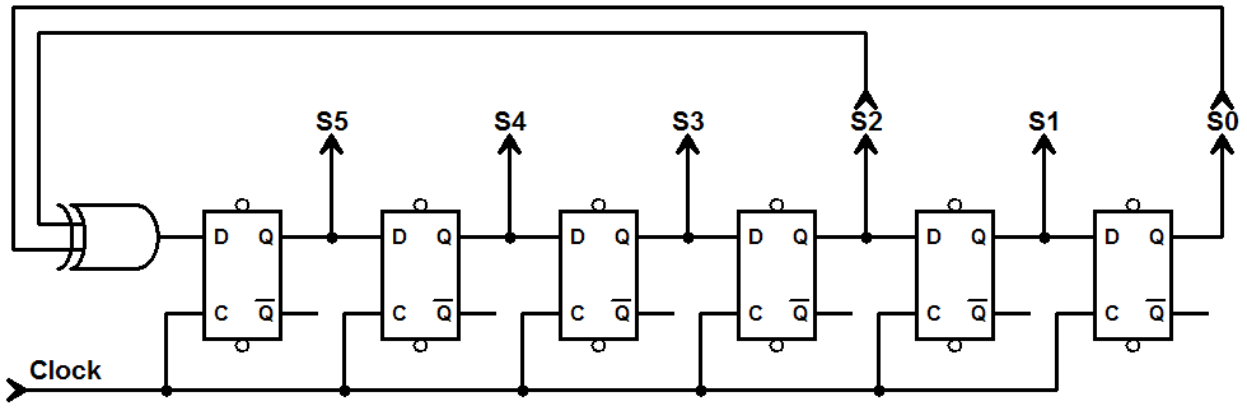
Beware of the temptation to do this, unless changing the problem definition is acceptable. In this case, these changes result in much shorter and more efficient assembly language code than before, but for the *wrong problem!* Yes, the code will print out triangles, but only those where the value of `N` was positive, and all triangles will have one extra space at the beginning of the line. You will get the same result for `N<=0` as for `N=1`.

```
Triangle  STR  LR,SaveLR      ; Save all registers
          STR  R1,SaveR1      ;
          STR  R2,SaveR2      ;
          STR  R3,SaveR3      ;

          MOV  R1,#1          ; R1 := 1
RP1       SUB  R3,R0,R1       ; Repeat
          ADD  R3,R3,#1       ;    R3 := N - R1 + 1
                              ;
          MOV  R2,R3          ;    R2 := R3
RP2       BL   Print_Blank    ;    Repeat Print_Blank
          SUBS R2,R2,#1       ;         R2 := R2 - 1
          BGT  RP2            ;    Until R2 <= 0

          MOV  R2,R0          ;    R2 := R0
RP3       BL   Print_Star     ;    Repeat Print_Star
          BL   Print_Blank    ;         Print_Blank
          SUBS R2,R2,#1       ;         R2 := R2 - 1
          BGT  RP3            ;    Until R2 <= 0

          BL   Print_LF       ;    Print_LF

          ADD  R1,R1,#1       ;    R1 := R1 + 1
          CMP  R1,R0          ; Until R1 > R0
          BLE  RP1            ;

          LDR  R3,SaveR3      ; Restore all registers
          LDR  R2,SaveR2      ;
          LDR  R1,SaveR1      ;
          LDR  PC,SaveLR      ; Return

SaveLR    DCD  0              ;
SaveR1    DCD  0              ;
SaveR2    DCD  0              ;
SaveR3    DCD  0              ;
```

GRADING: As before, accept any answer that works correctly. Remove 1 point for each minor syntax error. Remove 1 point for not correctly maintaining transparency among used registers. Remove 2 points overall for using REPEAT structures instead of WHILE structures that *change the definition* of the problem. Remove 5 points for answers that contain correct sections of code but are obviously incomplete.

<10> 10 Points – The following 6-bit shift-register circuit gets its input from the exclusive-OR of bits 0 and 2. Track the values in the six bits of the shift register over ten complete clock cycles.

Clock	S5	S4	S3	S2	S1	S0
0	1	1	1	1	0	1
1	0	1	1	1	1	0
2	1	0	1	1	1	1
3	0	1	0	1	1	1
4	0	0	1	0	1	1
5	1	0	0	1	0	1
6	0	1	0	0	1	0
7	0	0	1	0	0	1
8	1	0	0	1	0	0
9	1	1	0	0	1	0
10	0	1	1	0	0	1



After each clock cycle, the output values for S0 through S4 will become the previous values from S1 through S5 (i.e., S5 is copied to S4 at the same time that S4 is copied to S3, S3 to S2, and so on). The new value in S5 will become the XOR of the old values of bits S2 and S0, but the old value of S0 is then lost.

This circuit is actually useful in real life, as the pattern does not repeat until 63 (2^6-1) clock pulses have occurred. The sequence of values then becomes useful for a "pseudo random" number generator. The longer the shift register, the longer time will elapse before the sequence will repeat (but more XOR taps may be necessary). Note that the shift register should never be reset to zero, as the value will never change after that.

GRADING: Reserve 1 point for each line of answers. Remove 1 point for each error in the `S5` part of the line. In addition, remove 2 points overall for any case where the values in `S5...S1` of one line are not correctly transferred to `S4...S0` of the next line (but do not go below zero points).