CMPSCI 201 – Fall 2006

Midterm #1 – October 23, 2006

SOLUTION KEY

Professor William T. Verts

<1> 15 Points – For each of the following 8-bit binary values, show the equivalent decimal (base 10) values for both unsigned and signed interpretations, as well as the 8-bit two's-complement negation. The sign bit is the left-most of the eight bits.

Initial Value 8-Bit Binary	Unsigned Decimal	Signed Decimal	2's Complement 8-Bit Binary
`00000000`	0	0	`00000000`
`01111011`	123	+123	`10000101`
`10000000`	128	-128	`10000000`
`11110101`	245	-11	`00001011`
`11111111`	255	-1	`00000001`

<2> 15 Points – For each of the following 8-bit binary addition problems, show the sum in binary (8 bits) and the new values, 0 or 1, of the `N` (negative), `Z` (zero), `V` (overflow), and `C` (carry) status bits as a result of the sum. The sign bit is the left-most of the eight bits.

```
      01110100             01110100             10001101
     +10000111            +00111111            +11100101
Sum=  11111011       Sum=  10110011       Sum=  01110010
N   = 1              N   = 1              N   = 0
Z   = 0              Z   = 0              Z   = 0
V   = 0              V   = 1              V   = 1
C   = 0              C   = 0              C   = 1
```

<3> 10 Points – You know that the single ARM instruction `ADD R0,R0,R0,LSL #2` will multiply the contents of `R0` by 5 in-place without using the `MUL` instruction. By using other shift amounts, or by using different op-codes (such as `MOV`, `MVN`, `SUB`, or `RSB`), you can multiply in-place by a wide variety of constants. Determine, yes or no, if each of the numbers listed below can be used as a multiplier in this fashion. If yes, write down the single ARM instruction that will multiply `R0` in-place by the specified constant. If no, explain why not.

1. 7 Yes: **`RSB R0,R0,R0,LSL #3`** (8×R0 – R0)

2. 17 Yes: **`ADD R0,R0,R0,LSL #4`** (16×R0 + R0)

3. 21 No: 21 is 10101 in binary, requiring add/subtract of multiple shifts

4. -7 Yes: **`SUB R0,R0,R0,LSL #3`** (R0 - 8×R0)

5. 6 No: 6 is 110 in binary, requiring add/subtract of multiple shifts

<4> 10 Points – A task in an x86 program is to multiply two 16-bit integer numbers together as if they were *fixed-point* numbers, where the implicit decimal point is in the middle of each number (i.e., eight bits of whole number to the left and eight bits of fraction to the right). One operand is in AX and the other is in BX. Write a code fragment (not a subroutine or complete program) in correct x86 assembly language that returns the fixed-point product in AX. Ignore problems with overflow, and don't worry about register transparency; simply return in AX the middle 16 bits of the 32-bit product. (We looked at a similar problem on the ARM of multiplying two 32-bit fixed-point numbers together.)

Just performing **MUL** (or **IMUL**) puts the 32-bit product into **DX:AX**, so we need to get the composite number shifted right by eight bits. The brute-force approach is to shift **DX** right one bit (dropping the rightmost bit into the carry bit), then rotate right **AX** one bit (rotating the carry bit into the high bit of **AX**), and do it eight times:

```
                MUL   BX
                MOV   CX,8
      Again:    SHR   DX,1
                ROR   AX,1
                LOOP  Again
```

The key insight to performing this efficiently is to recognize that while **AX** is 16 bits, each 8-bit half is independently named as **AH** and **AL**. Similarly, **DX** is **DH** and **DL**. Thus, the part that we want is the lower half of **DX** (**DL**) and the upper half of **AX** (**AH**), and those pieces need to be moved into **AX**:

```
                MUL   BX
                MOV   AL,AH
                MOV   AH,DL
```

<5> 15 Points – The value 527 is in register R0. Trace the following ARM code fragment, and show the register results (in decimal) after each corresponding instruction is executed. Write **xxx** in registers where the value is not yet known (as shown). 5 points extra credit for determining the intended function of this program fragment.

		R0	R1	R2
		527	xxx	xxx
SUB	R1,R0,#10	527	517	xxx
SUB	R0,R0,R0,LSR #2	396	517	xxx
ADD	R0,R0,R0,LSR #4	420	517	xxx
ADD	R0,R0,R0,LSR #8	421	517	xxx
ADD	R0,R0,R0,LSR #16	421	517	xxx
MOV	R0,R0,LSR #3	52	517	xxx
ADD	R2,R0,R0,LSL #2	52	517	260
SUBS	R1,R1,R2,LSL #1	52	-3	260
ADDPL	R0,R0,#1	52	-3	260
ADDMI	R1,R1,#10	52	7	260

Extra Credit: This code fragment divides register **R0** by 10, and puts the remainder into register **R1** (register **R2** is just a temporary variable).

<6> 15 Points –We have seen how to load constants into 8-bit 6502 registers, 16-bit or 32-bit x86 registers, and 32-bit ARM registers. For example, we might write `LDA #0` on the 6502, `MOV AX,0` on the 8088, `MOV EAX,0` on the 386 (and later), and `MOV R0,#0` on the ARM. ***Any*** legal constant can be part of the 6502 and x86 instructions, but only a restricted set is available on the ARM. On the back of this page write a short essay explaining why this is so. In your answer discuss CISC vs. RISC designs, instruction lengths, execution time, etc. Please write neatly.

The 6502 and x86 processors have instructions that are of variable length, so they can string along enough bytes to embed the entire constant into the instruction. This takes extra fetches, and correspondingly extra execution time, depending on the length of the constant. Having instructions of variable length and execution time is fairly typical of CISC designs.

(So, for the 6502 the "`LDA` immediate" op-code tells the processor to fetch one more byte, on the 8088 the "`MOV AX,____`" op-code tells the processor to fetch two more bytes, and on the 386 and later the "`MOV EAX,____`" op-code tells the processor to fetch four more bytes. Similarly, the "`MOV AH,____`" instructions tells the 8088 to fetch only one more byte. The 6502 is primitive enough that it has characteristics of both RISC and CISC, and was in large part an inspiration for the ARM design team).

In contrast, the ARM instructions are always 32 bits in length. Those 32 bits must reserve space for the op-code, conditional execution codes, destination register, etc., leaving only a few bits remaining to encode constants. In particular, ARM instructions have only 12 bits for encoding constants, which allows only 4096 distinct values. Since instructions are fixed length, they take a fixed amount of time to fetch from memory and decode. Since the instruction lengths are known, it is easy for the processor to be executing one instruction as it is decoding the next (already fetched) instruction and fetching the instruction after that. This is typical of RISC designs.

(Pre-fetching instructions on a variable-length instruction set is much more difficult to perform efficiently.)

<7> 15 Points – In this problem you are to create an ARM subroutine to print out the 32-bit value passed in through register R0 as eight hexadecimal characters. Since we have not covered subroutine mechanisms on the ARM, nor how to use the stack, I have outlined the framework for the subroutine below. You do not have to worry about register transparency. The software tool that prints characters on the platform we will use expects its ASCII argument in the rightmost eight bits of R0, and is called by the ARM instruction SWI &0 (this is a software interrupt). You may not call other subroutines.

```
Print_Hex                                  ; Subroutine entry point
                    MOV    R1,R0
                    MOV    R2,#28
Loop                MOV    R0,R1,LSR R2
                    AND    R0,R0,#&0000000F
                    CMP    R0,#9
                    ADDGT  R0,R0,#'A'-10
                    ADDLE  R0,R0,#'0'
                    SWI    &0
                    SUBS   R2,R2,#4
                    BGE    Loop
```

-or-

```
                    MOV    R1,R0

                    MOV    R0,R1,LSR #28        ; Copy #1
                    AND    R0,R0,#&0000000F
                    CMP    R0,#9
                    ADDGT  R0,R0,#'A'-10
                    ADDLE  R0,R0,#'0'
                    SWI    &0

                    MOV    R0,R1,LSR #24        ; Copy #2
                    AND    R0,R0,#&0000000F
                    CMP    R0,#9
                    ADDGT  R0,R0,#'A'-10
                    ADDLE  R0,R0,#'0'
                    SWI    &0
                    …
                    MOV    R0,R1,LSR #0         ; Copy #8
                    AND    R0,R0,#&0000000F
                    CMP    R0,#9
                    ADDGT  R0,R0,#'A'-10
                    ADDLE  R0,R0,#'0'
                    SWI    &0

                    MOV    PC,LR    ; Return from subroutine
```

<8> 5 Points – The following x86 code fragment computes the integer square-root of the unsigned 16-bit value in `AX`, and returns that value in `AX` (overwriting the initial value).

```
                XOR  BX,BX          EOR  R1,R1,R1
                XOR  CX,CX          EOR  R2,R2,R2
                MOV  DX,1           MOV  R3,#1

SQRT_Loop:      ADD  CX,DX          ADD  R2,R2,R3
                CMP  CX,AX          CMP  R2,R0
                JA   SQRT_Done      BHI  SQRT_Done
                INC  BX             ADD  R1,R1,#1
                ADD  DX,2           ADD  R3,R3,#2
                CMP  BX,255         CMP  R1,#255
                JL   SQRT_Loop      BLT  SQRT_Loop
SQRT_Done:      MOV  AX,BX          MOV  R0,R1
```

Translate this fragment into the equivalent ARM code, where the argument is passed in and the result is passed back through register `R0`. Note that the x86 unsigned conditional jump `JA` (jump-if-above) is equivalent to the ARM instruction `BHI` (branch-if-higher).

The instructions can be translated essentially 1-for-1 from x86 into ARM code. It would be an even closer match had the x86 instructions been written for the 386 or later (using `XOR EBX,EBX` instead of `XOR BX,BX` for example). By mechanically writing `R0` for `AX`, `R1` for `BX`, `R2` for `CX`, and `R3` for `DX`, the instructions need only be replaced with the equivalent op-codes and register formats required by the ARM. One minor syntactic change is that the commas must be omitted from the jump-labels. The `EOR R2,R2,R2` instruction could be written as `MOV R2,#0` which takes up the same space and has the same effect.