**Barrel Shifter Worksheet (Part of Homework 3)**

**Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**1. Logical Shift Left (LSL) / Logical Shift Right (LSR)**

So to shift binary digits left or right “logically” just means that the bits move X spaces left or right. Any bits that fall off the edge vanish (possibly into the carry bit), and any bits coming in from the other edge are always zeroes. If R0 holds 10011100, then doing a MOV R0,R0,LSL #2 will result in R0 = 01110000.

For all these questions, given the 8-bit pattern and the amount to shift left or right, write the result.

|  |  |  |
| --- | --- | --- |
| Question # | Question | Answer: |
| 00001 | 00111100 LSL #1 | 01111000 |
| 00010 | 00111100 LSR #1 |  |
| 00011 | 00111100 LSL #2 |  |
| 00100 | 00111100 LSR #2 |  |
| 00101 | 10000000 LSL #1 |  |
| 00110 | 10000000 LSR #1 |  |
| 00111 | 11110000 LSL #1 |  |
| 01000 | 11110000 LSL #4 |  |
| 01001 | 11110000 LSR #1 |  |
| 01010 | 11110000 LSR #4 |  |

|  |  |  |
| --- | --- | --- |
| 01011 | 10101010 LSL #1 |  |
| 01100 | 10101010 LSL #2 |  |
| 01101 | 10101010 LSL #8 | 00000000 |
| 01110 | 10101010 LSR #1 |  |
| 01111 | 10101010 LSR #2 |  |
| 10000 | 10101010 LSR #8 |  |
| 10001 | 00000000 LSL #1 |  |
| 10010 | 00000000 LSR #1 |  |
| 10100 | 11111111 LSL #3 |  |
| 10101 | 11111111 LSR #6 |  |

What is 00000111 in decimal? \_\_\_\_\_\_\_\_\_ What is 00000111 LSL #1? \_\_\_\_\_\_\_\_\_\_\_\_

What is 00000111 LSL #2? \_\_\_\_\_\_\_\_\_\_ What is 00000111 LSL #3? \_\_\_\_\_\_\_\_\_\_\_\_

What is 00000111 LSR #1? \_\_\_\_\_\_\_\_\_\_ What is 00000111 LSR #2? \_\_\_\_\_\_\_\_\_\_\_\_

**2. Arithmetic Shift Right (ASR)**

As we can see, a shift left is the same thing as x2, and a shift right is the same thing as a division by 2. This works fine as long as the numbers are unsigned, but what if we’re doing 2’s compliment? The simple answer is if we mean the right shift to really be a division by 2, we have to do something a little different: if the leftmost bit is a 1, fill in the gap with a 1. Otherwise, it’s just like a LSR.

|  |  |  |  |
| --- | --- | --- | --- |
| Question # | Binary | Original Value in Decimal: | Result in Decimal: |
| 10110 | 11111100 ASR #1 |  |  |
| 10111 | 11111000 ASR #1 |  |  |
| 11000 | 10000000 ASR #7 |  |  |
| 11001 | 01010101 ASR #1 |  |  |
| 11010 | 00000000 ASR #3 |  |  |
| 11011 | 11110000 ASR #1 |  |  |
| 11100 | 01111111 ASR #1 |  |  |
| 11101 | 11110000 ASR #2 |  |  |
| 11110 | 01111111 ASR #2 |  |  |
| 11111 | 10000000 ASR #4 |  |  |
| 00000 | 00011111 ASR #4 |  |  |

Note: there is no arithmetic shift left, since we only care about preserving negativity and division on the internet. I mean, in binary math.

**3. Rotate Right (ROR)**

This is one of those commands we can do (for free!) in assembly that cannot be done easily in C++. A rotate right fills in the leftmost bit with whatever bit falls of the edge of the register to the right. So 00001111 ROR #1 would be 10000111. The rightmost bit becomes the leftmost one, and so forth.

There is no rotate left. If you want to rotate left, just do a rotate right with a bigger number of bits.

|  |  |  |
| --- | --- | --- |
| Question # | Question | Answer: |
| 00001 | 00111100 ROR #1 |  |
| 00010 | 00111101 ROR #1 |  |
| 00011 | 00111100 ROR #2 |  |
| 00100 | 00111101 ROR #2 | 01001111 |
| 00101 | 10000000 ROR #1 |  |
| 00110 | 10000001 ROR #1 |  |
| 00111 | 11110011 ROR #1 |  |
| 01000 | 11110011 ROR #4 |  |
| 01001 | 11110111 ROR #1 |  |
| 01010 | 11110111 ROR #4 |  |

There’s also a rotate right extended (RRX) that uses the carry bit as a bonus bit that gets rotated around, but you guys are probably sick of all this grunt work, so we’ll just leave it there.

The neat thing about these shift operations is that they can be used with most flexible second operands as basically a free multiply or divide. They’re never done in ARM32 as a separate instruction.

For example, instead of doing something like this:

ADD R0,R0,R1 //x = x + y

ADD R0,R0,R1 //x = x + y again (i.e. x = 2 \* y total)

We could do this in one step:

ADD R0,R0,R1, LSL #1 //x = x + 2\*y. Free multiply by 2!

Exceptions: There’s a couple exceptions you can’t combine the shifter with, the only one that’s important is MUL (multiply).