EE306 - Microprocessors

Lab 1
Using Logic Instructions with the ARM Processor

February 17, 2020

Erdal Sidal Dogan
#041702023

Alp Gokcek
#041701014

1 Longest Alternating Strings

Our approach for this problem was to utilize the *XOR* operation. We XORed the input string with another string that consist of alternating 1's and 0's. Given that XOR of two different bits always results in 1, when we XOR the input with the alternating string, longest sequence of 1's in the output should give the longest sequence of alternating bits in the input also. The problem is that we didn't knew wheter the first bit of longest alternating string was 1 or 0. This was a problem because the answer was either the length of longest sequence of 0's or 1's inthe output string, depending on the first bit of alternating sequence in input string.

$$\begin{array}{rr} & 101101010001 \\ \text{XOR} & 101010101010 \\ \hline & 000111111011 \end{array} \qquad \begin{array}{rr} & 101101010001 \\ \text{XOR} & 010101010101 \\ \hline & 111000000100 \end{array}$$

Above you can see the input string 101101010001 is being XORed with two alternating strings, one of them starts with 1 and other with 0. Using two alternating strings is to address the problem mentioned in previous paragraph, unknown start bits of the longest alternating string. What we did in order to solve this problem is, we used XOR operation for two different alternating strings. We know that the lenght of the longest sequence of 1's in the result of XOR operation is equal to the lenght of longest alternating bits in the input. We count length of longest sequence of 1's in both XOR outputs, the greater number is our answer.

Listing 1: Assembly Code for finding longest string of alternating 1's and 0's

```
.global _start
_start:
       LDR R1, TEST_NUM // load the data word into R1
       LDR R3, CONSECUTIVE_ALTERNATING // load the alternating word into R3
       MOV R0, #0 // R0 will hold the result and first loop's result
       MOV R4, #0 // R0 will hold the second loop's result
       EOR R1, R1, R3 // XORed data word with alternating word
LOOP:
       CMP R1, #0 // loop until the data contains no more 1's
       MOV R4, R0
       BEQ INIT_LOOP1 // branch to initialize the second loop's words
       LSR R2, R1, #1 // perform SHIFT, followed by AND
       AND R1, R1, R2
       ADD R0, #1 // count the string lengths so far
       B LOOP
INIT_LOOP1:
       LDR R1, TEST_NUM // load the data word into R1
       LDR R3, CONSECUTIVE_ALTERNATING1 // load the second alternating word
       EOR R1, R1, R3 // XORed the data word with second alternating word
       B LOOP1
LOOP1:
       CMP R1, #0 // loop until the data contains no more 1's
       BEQ END
       LSR R2, R1, #1 // perform SHIFT, followed by AND
       AND R1, R1, R2
       ADD R4, #1 // count the string lengths so far
       B LOOP1
COMPARE_RESULTS:
       CMP R0, R4 // compare first loop's result with second loop's result
       MOVLT R0, R4 // if second is greater than first, then select second as result
       MOV R4, #0 // clear register after use
```

```
END:
       B END
TEST_NUM: .word 0x103fe05f
CONSECUTIVE_ALTERNATING: .word 0xAAAAAAAA
CONSECUTIVE_ALTERNATING1: .word 0x55555555
.end
```

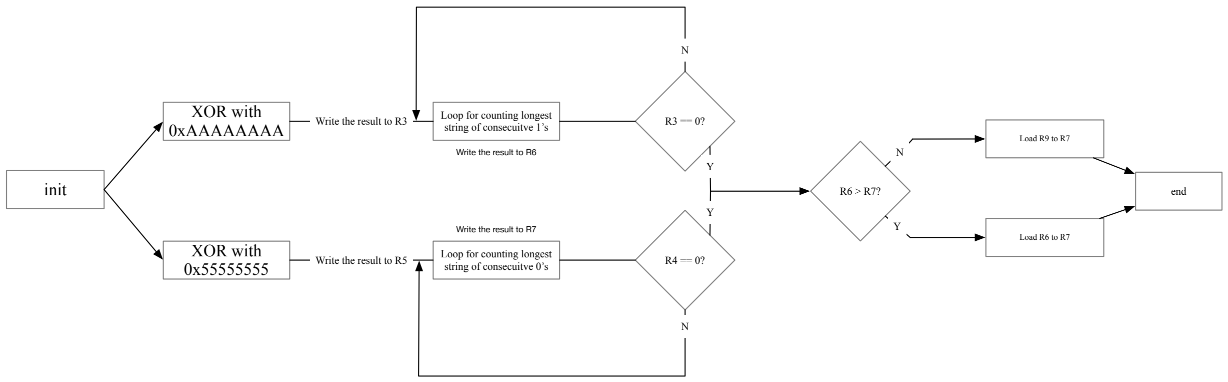

Figure 1: Longest Sequnce of Alternating 1's and 0's Flowchart

2 Parity Bit

Our approach for this problem was to sum all the bits in the binary number and check whether it is odd or even. In the first part, we have calculated the bitwise sum of all bits in the given number. We have taken the bitwise *AND* operation of the input string with hexadecimal number 0x1 and stored the result in a register. Given that AND operation of 1 with another bit always results in the other bits value, when we AND the input with the 1, we get the value of LSB (least significant bit). Then we shifted the input string to the right by 1 bit which cleared the LSB. In the final part of the calculating bitwise sum of all bits in the input string, we add the result of *AND* operation that stored in a register with the current value of the sum of bits. This process continues until the input string is equal to hexadecimal number 0x0. In the last part, we checked whether the sum of all bits LSB is 1 or 0. If it is 1 then the sum is odd which makes the even parity bit 1 and odd parity bit 0. If it is 0 then the sum is even which makes the even parity bit 0 and the odd parity bit 1. After checking last bit we did clear the MSB (most significant bit) and LSB and calculated the logical *OR* operation with hexadecimal number 0x80000000 if the even parity bit is 1, else we ORed with hexadecimal number 0x00000001.

Listing 2: Assembly code for finding the odd and even parity bits

```
.global _start
_start:
       LDR R1, TEST_NUM // load the data word into R1
       LDR R8, RESULT // r8 will hold the result
       LDR R11, OR_NUM1 //odd parity bit = 1, even parity bit = 0
       LDR R12, OR_NUM2 //odd parity bit = 0, even parity bit = 1
       MOV R0, #0 // initialized accumulator
LOOP:
       CMP R1, #0 // loop until the data contains no more 1's
       BEQ CONT
       AND R3, R1, #1 // take the last bit
       LSR R2, R1, #1 // perform SHIFT, followed by AND
       ADD R0, R0, R3 // add the current bit to accumulator
       B LOOP
```

```
EVEN:
       ORR R8, R8, R11 // put OR_NUM1 as the result
       B END
CONT:
       ANDS R3, R0, #1 // last bit indicates whether sum is even or odd
       BEQ EVEN // if sum is even then branch
       ORR R8, R8, R12 // put OR_NUM2 as the result
END:
       B END
TEST_NUM: .word 0xAAAAAAAB
RESULT: .word 0x00000000
OR_NUM1: .word 0x80000000 //odd parity bit = 1
OR_NUM2: .word 0x00000001 //even parity bit = 1
.end
```

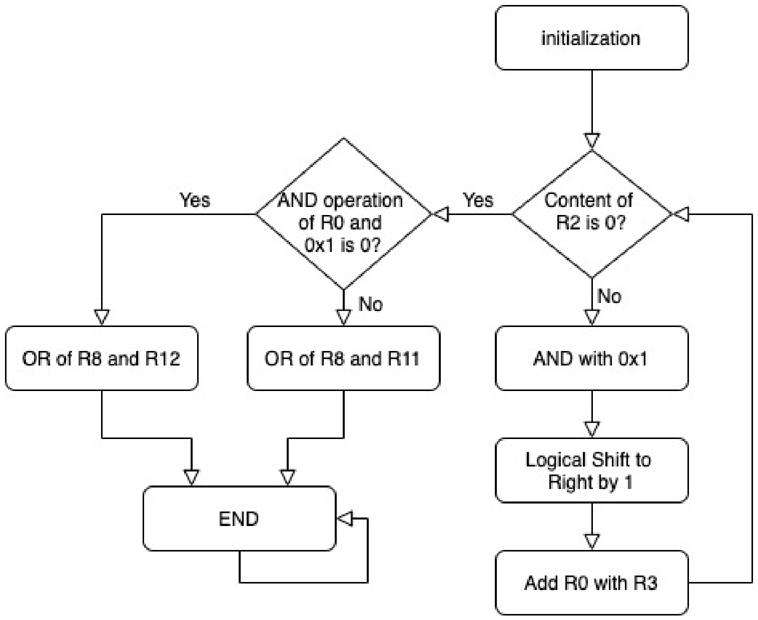

Figure 2: Parity Bit detection flowchart