Student Name: William Browne Student Number: 09389822

Explanation of how the original program generates Hamming Codes

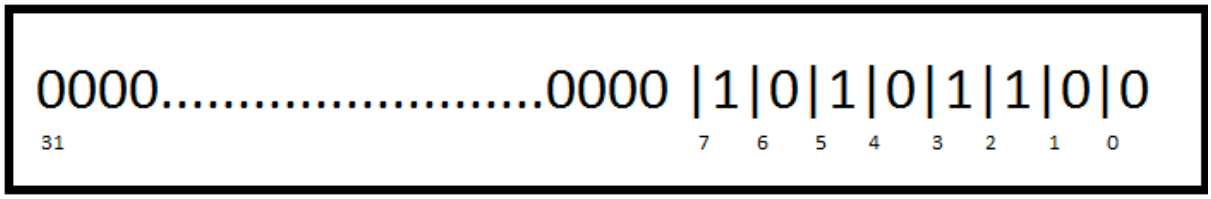
Hamming codes => error correcting codes

Parity bits => used as the simplest form of error detecting code

By starting with an 8-bit value, we are required to expand it to a 12-bit value (in order to insert 4 check bits that generate even parity).

In the case of the first program, the four check bits that are added (referred to as c0, c1, c2, c3) are placed in bit positions 0, 1, 3, and 7. The original bit positions will be referred to as d0 – d31.

Below is the original 8-bit value in R1 -> 0XAC or binary 10101100



When we have chosen our 8-bit value, and loaded it into a register, we need to expand it so it becomes a 12-bit value (so we can then enter in our four check bits).

The way in which we expand the 8-bit value is by separately and individually clearing all the bits apart from the ones we will use for the check bits. Firstly, we clear all the bits in the value except the LSB (least significant bit) or d0, then the result we get we shift left two bit positions. Next, we clear all bits in the value r0 except d1, d2, and d3. The result we get is combined with d0 using an ORR operation. In the last step, the same procedure is followed but with the positions d4 – d7.

Note: The value in R1 is not changed in the above procedure

Once we have our new 12-bit value, we can begin inputting the check bit values into their appropriate positions. This is done by generating a parity bit

for each check bit (using EOR), clearing all but that check bit, and finally, adding the check bit to the 12-bit in r0 using the ORR operation.

```
; Generate check bit c0

EOR R2, R0, R0, LSR #2   ; Generate c0 parity bit using parity tree
EOR R2, R2, R2, LSR #4   ; ... second iteration ...
EOR R2, R2, R2, LSR #8   ; ... final iteration

AND R2, R2, #0x1         ; Clear all but check bit c0
ORR R0, R0, R2       ; Combine check bit c0 with result
```

<u>The even parity generated by the check bits</u>

<u>{obtained from Jonathan's lecture slides}</u>

C0, in bit position 0, is calculated to produce even parity for bit positions

2, 4, 6, 8, 10

C1 in bit position 1, is calculated to produce even parity for bit positions

2, 5, 6, 9, 10

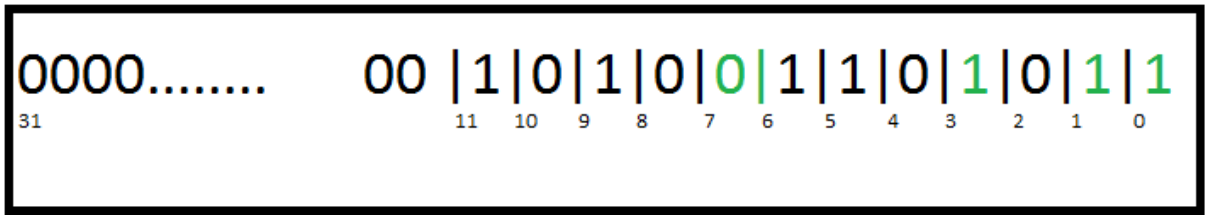
C2 in bit position 3, is calculated to produce even parity for bit positions

4, 5, 6, 11

C3 in bit position 7, is calculated to produce even parity for bit positions

8, 9, 10, 11

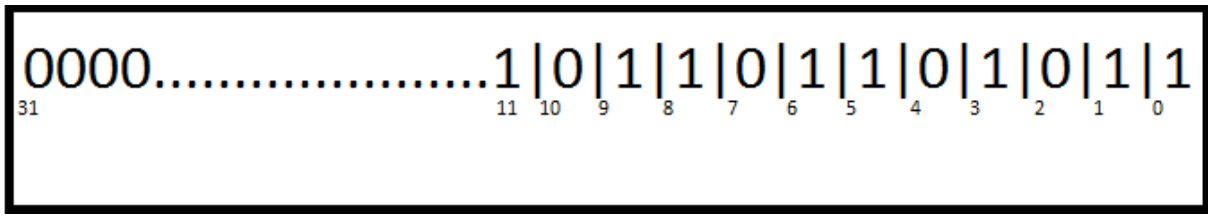
Value with check bits generated:



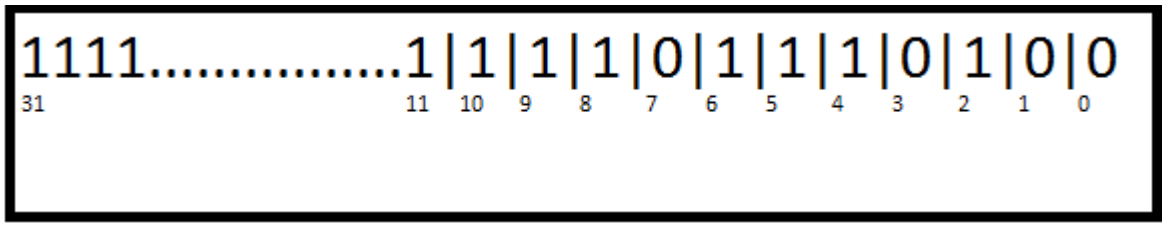
My proposed approach to the problem

After bit-8 was flipped in the original 12-bit value (causing an error), I needed to form a solution.

12-bit value with error stored in R0:



I began my solution by creating a register(R3) of value 0XFFFFFF74 (below) that I could use to clear the check bits in the original R0 value by using an AND operation. However I would store the result in a new register, leaving r0 still the-12 bit value, with an error and all check bits.

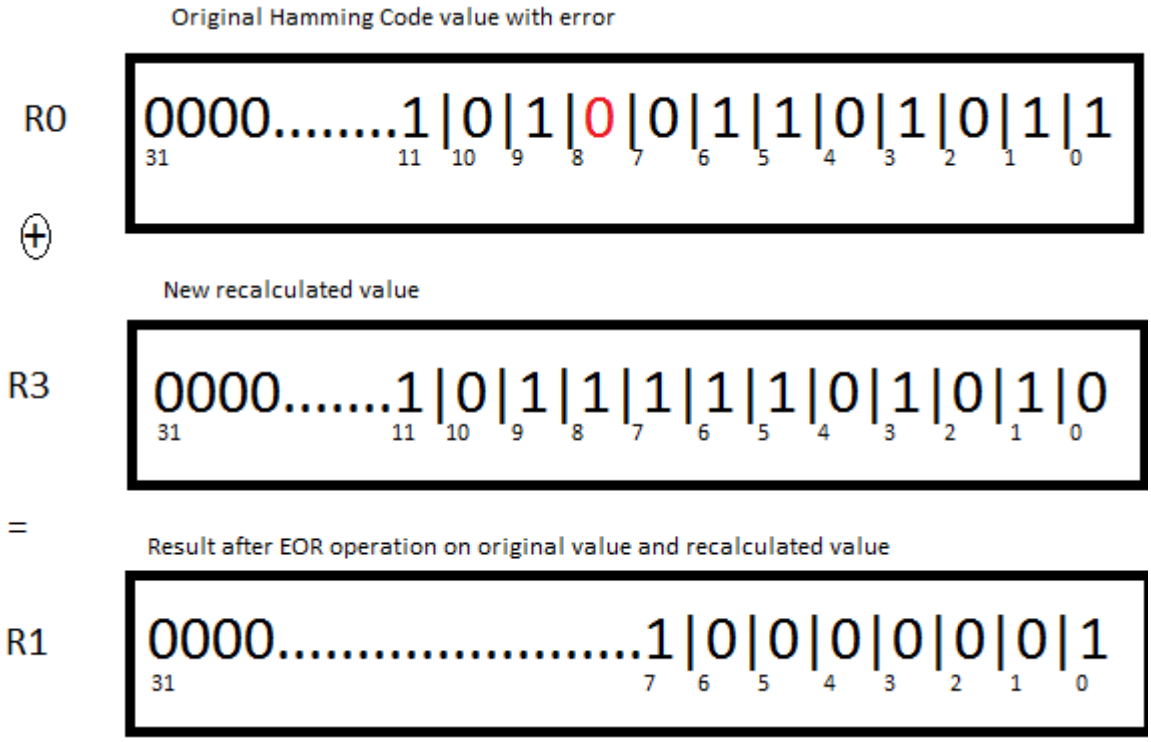


After check bits 0, 1, 3, and 7 were cleared, I began generating the check bits (just as done before in the original program).

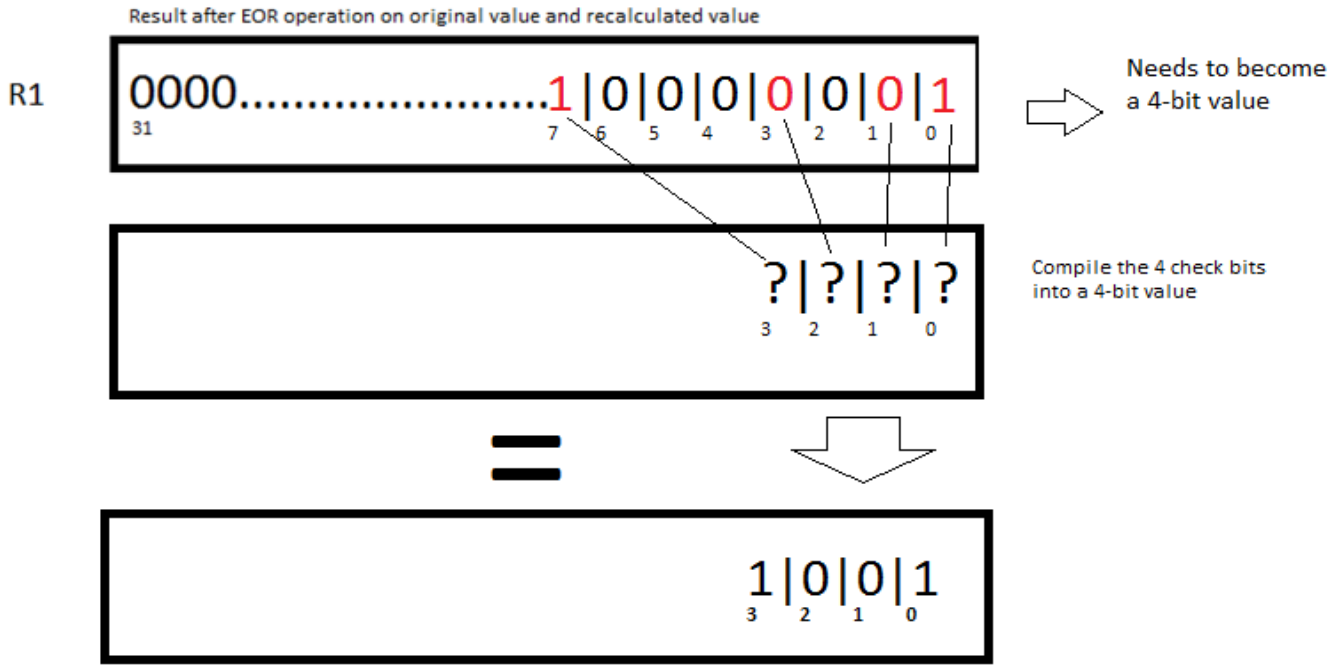
Example:

```
093     ; Generate check bit c0
094
095     EOR R2, R3, R3, LSR #2  ; Generate c0 parity bit using parity tree
096     EOR R2, R2, R2, LSR #4  ; ... second iteration ...
097     EOR R2, R2, R2, LSR #8  ; ... final iteration
098
099     AND R2, R2, #0x1        ; Clear all but check bit c0
100     ORR R3, R3, R2          ; Combine check bit c0 with result
101
```

With the original value in r0 and my recalculated value in r3, I compared the two values using EOR and stored the result in a new register.



I then isolated the results of the EOR operation, and used the bits that corresponded with the check bits to create a 4-bit value (ie bit positions 0, 1, 3 and 7)



Creating the 4-bit value was done by clearing all bits except the check bits, and shifting each check bit left however many positions it needed to. IE check bit 7 needed to be shifted 4 bit positions right in order to be in bit position 3, and check bit 3 needed to be shifted 1 position right in order to be in bit position 2 and so on.

Example:

```
140  ;Clearing all bits apart from c7 and shifting bit 4 positions right
141  LDR R4, =0X80
142  AND R4, R4, R1
143  MOV R4, R4, LSR #4
144
```

I used different registers to represent each check bit value (r4 for check bit 7 above for instance), and then I used an ADD operation so that they would combine to become a 4-bit value.

```
153      ;Adding the 4 registers together
154      ADD R1, R4, R5
155      ADD R1, R1, R6
```

By subtracting 1 from the 4-bit value, we can determine the bit position of the error!

```
159     ;Subtracting 1 from R1 to determine the bit position of the error
160     SUB R1, R1, #1
161
```

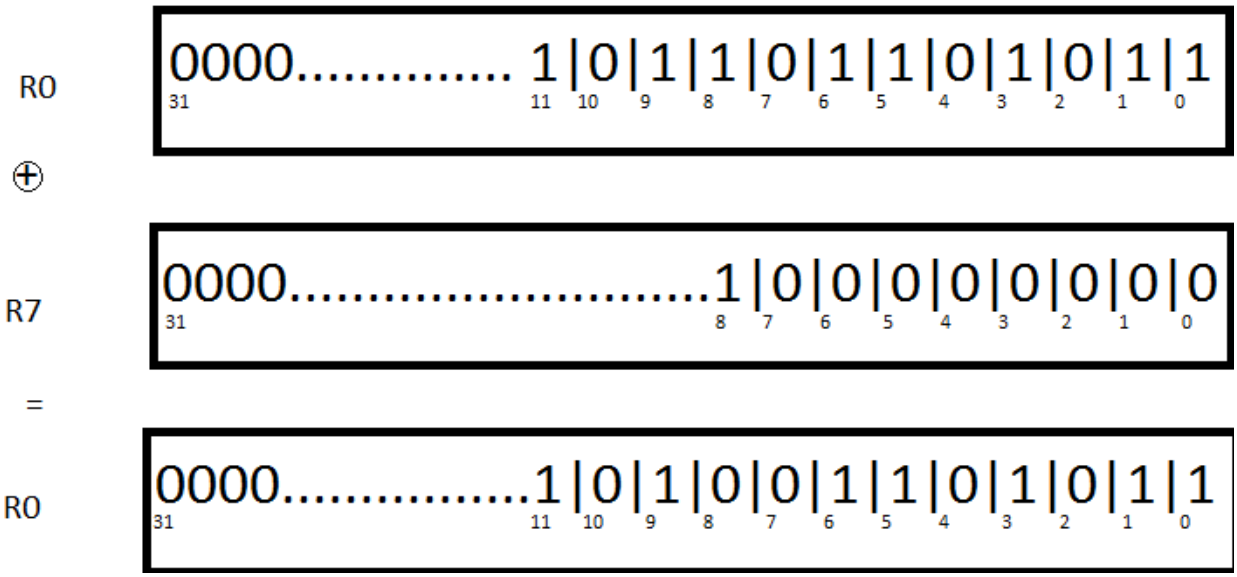
Since r1 contained binary 1001, when we subtract 1 (or binary 0001) it becomes 1000 (or 8 in decimal)

Now that we know the position of the error (bit position 8), it is time to correct it!

My approach to correcting the error

I then made a temporary register with the value of binary 1, and used a MOV operation to shift the value left using “LSL R1”(it shifts left as many bit positions as there are bits in the register. In this instance, it shifts it left 8 positions)

An EOR operation of this value with the original value in r0 is then used to flip the value in bit position 8. This is the operation that fixes the error.



The result I ended up with was 0x00000A6B