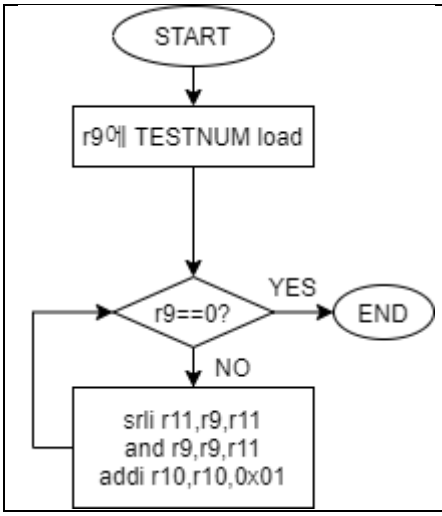
전자 HW 설계 – 실습 보고서

이름: 양해찬 (2016124145)

✓ Part I

동작 원리



TEST_NUM으로 설정한 숫자에 대해 가장 긴 1의 길이를 찾는 코드이다. 먼저 r9에 해당 숫자를 load하고 r9가 0이 될 때 까지 testnum을 right shift한 후 and를 취해 ones를 찾는다.

구현 코드 설명

```
.text
.global _start
_start:
        ldw     r9,   TEST_NUM(r0)
        mov     r10,  r0
LOOP:
        beq     r9,   r0,   END     # shift가 다 되면 r9은 0이 되기 때문에 0이 된다면 END로 분기한다.
        srli    r11,  r9,   0x01    # right shift
        and     r9,   r9,   r11     #r4와 r4를 right shift한 값을 and.
        addi    r10,  r10,  0x01    #r10=r10+1
        br      LOOP                #loop으로 br
END:    br      END
TEST_NUM:       .word     0x3fabedef    # 11111110101011111011011111011111이므로 7이 나와야 함.
                .end
```

전자 HW 설계 – 실습 보고서

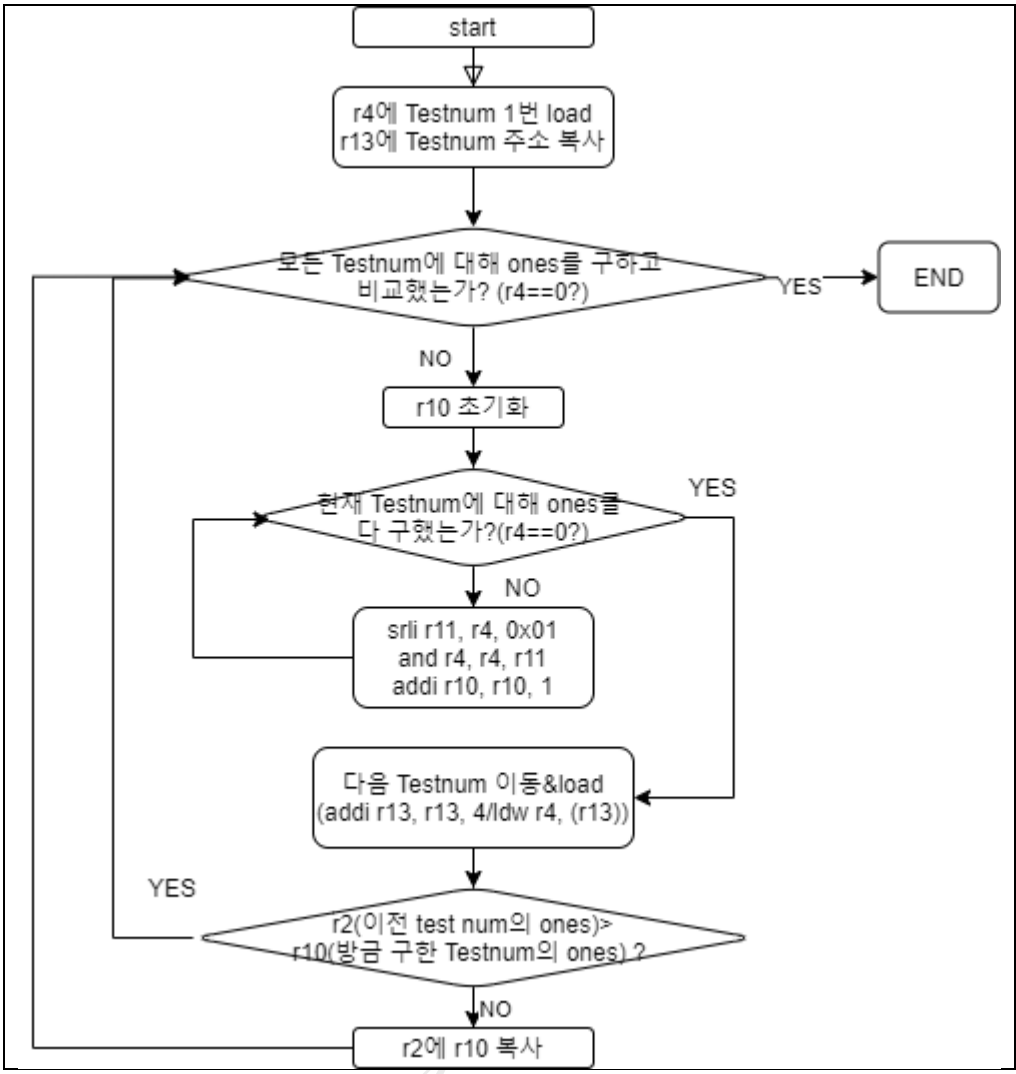
결과 및 토의

pc 0x0000001C zero 0x00000000 r1 0x00000000 r2 0x00000000 r3 0x00000000 r4 0x00000000 r5 0x00000000 r6 0x00000000 r7 0x00000000 r8 0x00000000 r9 0x00000000 r10 0x00000007 r11 0x00400000	R10에 7이 나오는 것을 볼 수 있는데 TESTNUM인 0x3fabedef는 binary로 나타내면 0011/1111/1010/1011/1110/1101/1110/1111이므로 ones는 7이다. R11은 r9가 0이 되기 직전까지 right shift 된 값인데, 0100/0000/0000/0000/0000/0000으로 16진수로 0x00400000이다.

전자 HW 설계 – 실습 보고서

✓ Part II

동작 원리



Part2는 TEST_NUM이 여러 개 주어졌을 때 그 중에서 가장 긴 연속된 1의 개수를 찾는다. Subroutine ONES를 호출하여 사용하고, TEST_NUM의 ones를 구하는 방식은 Part1과 동일하고(구하고자 하는 TEST_NUM을 right shift해 and를 취하는 것), 구한 ones를 r10에 저장해 이전에 저장했던 ones인 r2값과 비교해서 더 크다면 새로운 값을 r2에 저장하고 아니라면 다시 ONES를 호출한다. 호출 후 모든 TEST_NUM에 대해 ones를 구하고 비교했다면 마지막 word인 0을 만나게 되고, 코드는 종료하게 된다.

구현 코드 설명

```
        .text
        .global _start
_start:
        ldw     r4,     TEST_NUM(r0)    #TEST_NUM 첫 번째 word load
        mov     r2,         r0              #r2=0 초기화
        movia   r13,    TEST_NUM        #r13에 TEST_NUM 주소 복사

ONES:
        beq     r4,     r0,     END     #TEST_NUM의 끝까지 오게 되면 END
        mov     r10,    r0              #r10=0으로 초기화
        br      LOOP                    #LOOP으로 branch
```

```
ONES1:   addi   r13,   r13,   4      #다음 TEST_NUM주소로 이동
         ldw    r4,    (r13)        #r4에 다음 TEST_NUM load
         bgt    r2,    r10,   ONES   /*앞서 LOOP에서 구한 연속된 1의 개수보다 이전 TEST_NUM에 구해서 저장했던 값
이 더 크면 바로 ONES로 branch*/
         mov    r2,    r10          #아니라면 r2에 새로운 값을 저장
         br     ONES                #ONES로 branch

LOOP:    beq    r4,    r0,    ONES1   #r4가 0이면 ONES1로 branch
         srli   r11,   r4,    0x01    #r11에 r4를 right shift한 값을 넣는다
         and    r4,    r4,    r11     #r4와 r4를 right shift한 값을 and
         addi   r10,   r10,   0x01    #r10=r10+1
         br     LOOP                 #LOOP으로 branch
END:     br     END
TEST_NUM:  .word      0x000000FF, 0x00000001, 0x00000003,0x00000FAF,0x00000AAA,0x00101013,0x55555555,0xFFFFF
FFF,0x00000005,0x000A0A01,0

         .end
```

결과 및 토의

pc	0x00000044	r18	0x00000000
zero	0x00000000	r19	0x00000000
r1	0x00000000	r20	0x00000000
r2	0x00000020	r21	0x00000000
r3	0x00000000	r22	0x00000000
r4	0x00000000	r23	0x00000000
r5	0x00000000	et	0x00000000
r6	0x00000000	bt	0xFFFFFFFF
r7	0x00000000	gp	0x00000000
r8	0x00000000	sp	0x00000000
r9	0x00000000	fp	0x00000000
r10	0x00000001	ea	0x00000000
r11	0x00050500	ba	0xFFFFFFFF
r12	0x00000000	ra	0x00000000
r13	0x00000070	status	0x00000000
r14	0x00000000	estatus	0x00000000
r15	0x00000000	bstatus	0xFFFFFFFF
r16	0x00000000	ienable	0x00000000
r17	0x00000000	ipending	0x00000000
		cpuid	0x00000000

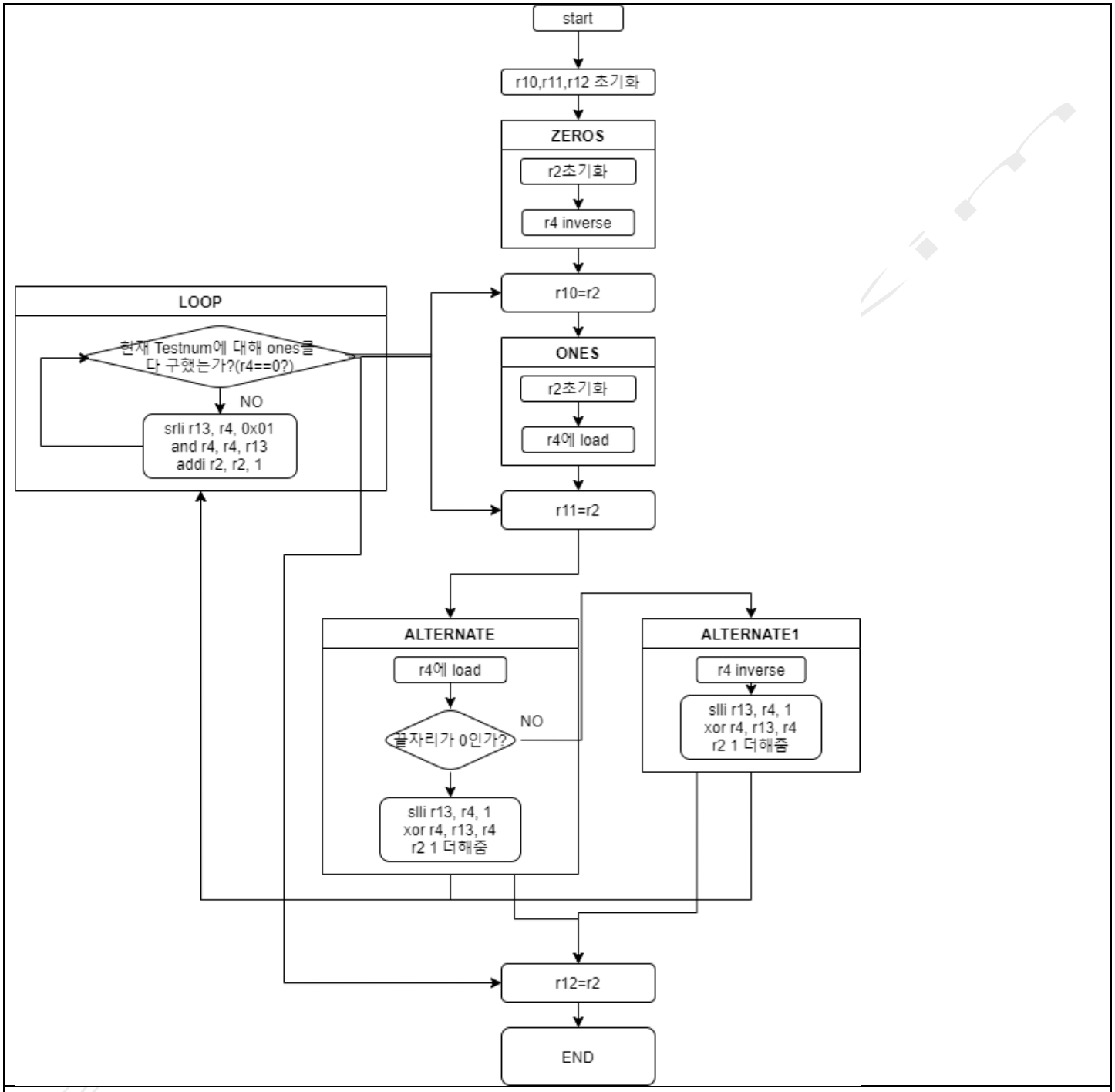
TEST_NUM 이 0x000000FF, 0x00000001, 0x00000003, 0x00000FAF, 0x00000AAA,0x00101013, 0x55555555,0xFFFFFFFF,0x00000005,0x000A0A01 인데 0xFFFFFFFF의 ONES가 32개로 가장 크기 때문에 16진수로 0x00000020이 출력되어야 한다.

최종 결과값을 저장하는 r2에 저장된 값을 보면 0x20으로 정확한 출력 값이 나오는 것을 볼 수 있다.

r10에 저장되어 있는 값은 1인데, 이것은 가장 마지막에 구한 TEST_NUM의 ONES이므로 0x000A0A01의 ONES인 0x1이 맞는 값이다.

✓ Part III

동작 원리.



8 byte TEST_NUM에 대해서 가장 긴 연속되는 1의 개수를 찾는 ONES, 가장 긴 연속되는 0의 개수를 찾는 ZEROS 그리고 가장 긴 1과0이 반복적으로 연속되는 ALTERNATE를 구하는 것이다. 공통적으로 LOOP을 통해 연속되는 1의 개수를 찾고, LOOP의 동작원리는 Part2와 동일하다.

ONES: LOOP로 분기해서 ones를 센 후 r2를 리턴 한다.

ZEROS: TEST_NUM을 0과 nor을 취해 inverse 한 후 ONES와 동일하게 동작한다.

ALTERNATE,ALTERNATE1: TEST_NUM을 left shift 한 후 shift 전과 xor을 취해서 연속된 1의 개수를 찾는 방법으로 구했다. 하지만 마지막 자리가 1인 경우, left shift 하게 되면 alternate 자리가 한자리가 늘어나기 때문에 ALTERNATE1이라는 서브루틴을 하나 더 만들어 주어 마지막 자리가 0이 아닌 경우, ALTERNATE1으로 분기해서 TEST_NUM을 0과 nor을

취해 inverse 해준 후 같은 방법으로 alternate를 구하게 된다. 두 서브루틴 모두 xor을 구한 후 LOOP으로 분기해서 xor의 연속된 1의 개수를 구하는 방식이다. 그리고 alternate 를 구할 때 시작 지점을 포함해야 하므로 1을 더한 후 구했다.

구현 코드 설명

```
        .text
        .global _start

_start:  ldw r4, TEST_NUM(r0)
        mov r10, r0     #r10,r11,r12 0으로 초기화
        mov r11, r0
        mov r12, r0
        call ZEROS      #zeros 호출
        mov r10, r2     #return값을 r10에 저장
        call ONES       #ONES 호출
        mov r11, r2     #return값을 r10에 저장
        call ALTERNATE  #ALTERNATE 호출
        mov r12, r2     #return값을 r10에 저장
        br      END

ZEROS:   mov    r2,    r0        #r2 초기화
        nor    r4,    r4,    r0  #ONES와 같은 방식으로 구하기 위해 inverse를 해줌
        br     LOOP              #LOOP으로 branch

ONES:    mov    r2,    r0        #r2 초기화
        ldw    r4, TEST_NUM(r0)  #r4에 다시 TEST_NUM load
        br     LOOP              #LOOP으로 branch

LOOP:    beq    r4,    r0,    return  #ones(zeros)를 모두 구했다면 return
        srli   r13,   r4,    0x01    #r4 를 right shift
        and    r4,    r4,    r13     #shift한 값과 and
        addi   r2,    r2,    0x01    #return할 값에 +1
        br     LOOP                  #반복

ALTERNATE:  ldw    r4, TEST_NUM(r0)          #r4에 다시 TEST_NUM load
        slli   r14,   r4,    31          #r4를 31byte left shift한다.
        bne    r4,    r0,    ALTERNATE1  #shift했을 때 값(원래 값의 마지막 byte자리)이 0이 아니라면 ALTERNATE1
로 branch
        slli   r13,   r4,    0x01        #r4를 1byte만큼 left shift
        xor    r4,    r13,   r4          #shift된 값과 xor을 취한다
        movi   r2,    0x01               #시작지점부터 구해야 하므로 1을 더해준다
        br     LOOP                      #xor취한 값의 ones를 구해야 하므로 loop로 branch
ALTERNATE1: /*slli하게되면 남는 자리는 0으로 채워지게 되므로 끝자리가 1이라면 alternate가 하나 더 생기는 것이므
로 inverse해서 구해야 한다.*/
```

```
        nor     r4,     r4,     r0      #r4를 inverse
        slli    r13,    r4,     0x01    #이하 ALTERNATE와 동일
        xor     r4,     r13,    r4
        movi    r2,     0x01
        br      LOOP

END:        br  END
return:     ret
TEST_NUM: .word 0x3FABEDEF /*an example*/
.end
```

결과 및 토의

pc	0x00000088	r18	0x00000000
zero	0x00000000	r19	0x00000000
r1	0x00000000	r20	0x00000000
r2	0x00000007	r21	0x00000000
r3	0x00000000	r22	0x00000000
r4	0x00000000	r23	0x00000000
r5	0x00000000	et	0x00000000
r6	0x00000000	bt	0xFFFFFFFF
r7	0x00000000	gp	0x00000000
r8	0x00000000	sp	0x00000000
r9	0x00000000	fp	0x00000000
r10	0x00000002	ea	0x00000000
r11	0x00000007	ba	0xFFFFFFFF
r12	0x00000007	ra	0x00000024
r13	0x00020000	status	0x00000000
r14	0x80000000	estatus	0x00000000
r15	0x00000000	bstatus	0xFFFFFFFF
r16	0x00000000	ienable	0x00000000
r17	0x00000000	ipending	0x00000000
		cpuid	0x00000000

TEST_NUM이 0x3FABEDEF이므로 ZEROS는 2, ONES는 7, ALTERNATE는 7이 나와야 한다.(ALTERNATE의 경우 시작점과 끝점까지 count ex)1101 -> 3개)

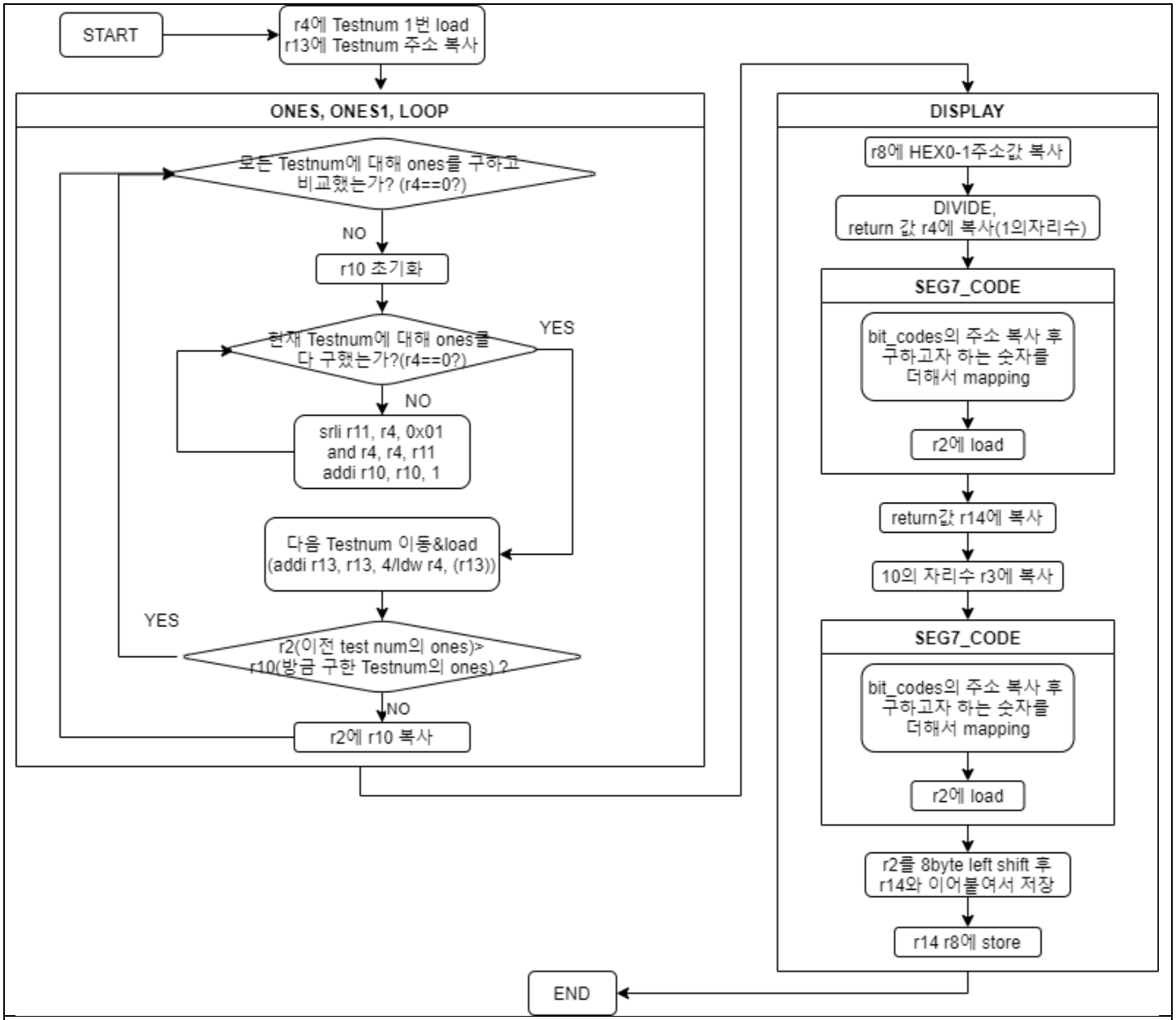
r10은 ZEROS값, r11은 ONES, r12는 ALTERNATE값이 출력되어야 하는데 각각 2, 7, 7으로 정확한 출력 값이 나오는 것을 볼 수 있다.

r2의 value가 7인 이유는 마지막으로 r2를 사용한 것이 ALTERNATE(or ALTERNATE1) 서브루틴인데 여기서의 리턴 값인 7이 저장되어 있기 때문이다.

전자 HW 설계 – 실습 보고서

✓ Part IV

동작 원리



Part4는 Part2에서의 결과를 DE1-SOC 보드의 7-segment로 출력하는 코드이다.
Ones를 구하고 divide 하는 과정은 이전의 실습과 동일하게 동작한다.
7-segment로 출력하기 위해서는 먼저 DIVIDE를 이용해 10의 자리와 1의 자리로 분리를 해주어야 한다.
HEX별 LED의 점등 유무 정보가 담겨있는 BIT_CODES를 이용해 SEG7_CODE에서 DIVIDE에서 return받은 값인 1의자리와 10의 자리 숫자의 정보에 맞게 r2와 r3에 load 한다.
10의 자리 숫자는 HEX1, 1의 자리 숫자는 HEX0에 출력하는 방식이다. 각 HEX는 8byte씩 할당되어 있는데, SEG7_CODE의 return값을 이어 붙이면 총 16byte로 HEX0과 HEX1에 해당되는 주소 값에 store하면 출력할 수 있다.

.

```
        .text
        .global _start
_start:  ldw    r4,    TEST_NUM(r0)   #testnum load
        mov    r2,       r0          #r2 초기화
        movia  r13,   TEST_NUM       #r13에 TEST_NUM주소 복사
        call   ONES                  #ONES 호출
        call   DISPLAY               #DISPLAY 호출

ONES:     beq    r4,    r0,    return  #TEST_NUM의 끝까지 오게 되면 END
        mov    r10,   r0             #r10=0으로 초기화
        br     LOOP                  #LOOP으로 branch

ONES1:    addi   r13,   r13,   4      #다음 TEST_NUM주소로 이동
        ldw    r4,    (r13)          #r4에 다음 TEST_NUM load
        bgt    r2,    r10,   ONES   /*앞서 LOOP에서 구한 연속된 1의 개수보다
        이전 TEST_NUM에 구해서 저장했던 값이 더 크면 바로 ONES로 branch*/
        mov    r2,    r10            #아니라면 r2에 새로운 값을 저장
        br     ONES                  #ONES로 branch

LOOP:     beq    r4,    r0,    ONES1  #r4가 0이면 ONES1로 branch
        srli   r11,   r4,    0x01   #r11에 r4를 right shift한 값을 넣는다
        and    r4,    r4,    r11    #r4와 r4를 right shift한 값을 and
        addi   r10,   r10,   0x01   #r10=r10+1
        br     LOOP                  #LOOP으로 branch

# 숫자별로 HEX LED에 불이 ON인지 OFF인지를 binary 로 나타낸 것
BIT_CODES:  .byte   0b00111111,0b00000110,0b01011011,0b01001111,0b01100110
        .byte   0b01101101,0b01111101,0b00000111,0b01111111,0b01100111
        .skip   2   #숫자가 총 10개이므로 10byte이다. word단위로 만들어 주기 위해 2byte skip
SEG7_CODE:  movia   r15,BIT_CODES   #BIT_CODES 주소 복사
        add r15,r15,r4          #r4의 숫자에 맞게 주소 값에 더해서 해당 bitcode 주소를 가질 수 있게 한다.
        ldb r2,(r15)            #r2에 load
        br  return              #return
DIVIDE:
      movi    r5,10
      movi    r3,0       #십의 자리 수

CONT:   blt r2,r5,return   #r2가 10보다 작으면 return, r2는 1의 자리 수
      sub r2,r2,r5       #r2에 10을 뺄 때마다 r3에 더함으로써 r3이 10의 자리 수를 가질 수 있다.
      addi    r3,r3,1
      br  CONT

return:    ret
```

```
DISPLAY:    movia   r8,0xFF200020      #r8에 HEX1-0 주소 값을 복사
            call    DIVIDE              #DIVIDE 호출
            mov     r4,     r2          #r4에 r2 복사(1의 자리 수)
            call    SEG7_CODE           #1의 자리 수에 맞게 mapping
            mov     r14,    r2          #return받은 값을 r14에 복사
            mov     r4,     r3          #r4에 r3 복사(10의 자리 수)
            call    SEG7_CODE           #10의 자리 수에 맞게 mapping
            slli    r2,     r2,     8
            or      r14,    r14,    r2  #r2와 r3에 저장된 값을 r14에 하나로 이어서 저장
            stw     r14,    (r8)        #r8에 store
END:        br      END
TEST_NUM:   .word       0x000000FF, 0x00000001, 0x00000003,0x00000FAF,0x00000AAA,0x00101013,0x55555555,0xFFFFFFFF,0x00000005,0x000A0A01,0

            .end
```

결과 및 토의

pc	0x000000B4	r19	0x00000000
zero	0x00000000	r20	0x00000000
r1	0x00000000	r21	0x00000000
r2	0x00004F00	r22	0x00000000
r3	0x00000003	r23	0x00000000
r4	0x00000003	et	0x00000000
r5	0x0000000A	bt	0xFFFFFFFF
r6	0x00000000	gp	0x00000000
r7	0x00000000	sp	0x00000000
r8	0xFF200020	fp	0x00000000
r9	0x00000000	ea	0x00000000
r10	0x00000001	ba	0xFFFFFFFF
r11	0x00050500	ra	0x000000A8
r12	0x00000000	status	0x00000000
r13	0x000000E0	estatus	0x00000000
r14	0x00004F5B	bstatus	0xFFFFFFFF
r15	0x0000004F	ienable	0x00000000
r16	0x00000000	ipending	0x00000000
r17	0x00000000	cpuid	0x00000000

TEST_NUM이 0x000000FF, 0x00000001, 0x00000003,0x00000FAF, 0x00000AAA,0x00101013,0x55555555,0xFFFFFFFF,0x00000005,0x000A0A01 이므로 ONES는 0xFFFFFFFF의 32 즉 0x20이 출력되어야 한다.

r2가 0x4F00인 이유는 DIVIDE를 통해 구한 ONES의 10의 자리 수인 0에 해당하는 bit code가 0b01001111즉, 0x4F인데 이것이 1의 자리 수에 해당하는 bit code와 연결하기 위해 8byte만큼 left shift 했기 때문이다.

r3는 DIVIDE를 통해 구한 ONES의 10의 자리 수 이므로 3이다.

r14가 0x004F5B인 이유는 1의 자리 수에 해당하는 2의 bit code 0b01011011 즉, 0x5B와 10의 자리 수인 3에 해당하는 bit code가 0b01001111즉, 0x4F을 8byte만큼 left shift 한 값 0x4F00과 합쳤기 때문이다.

r8에는 HEX 주소 값인 0xFF200020이 들어가 있다.

실제 보드의 HEX1에는 ONES의 10의 자리 수인 3, HEX0에는 ONES의 1의 자리 수인 2가 출력 되고 있다.