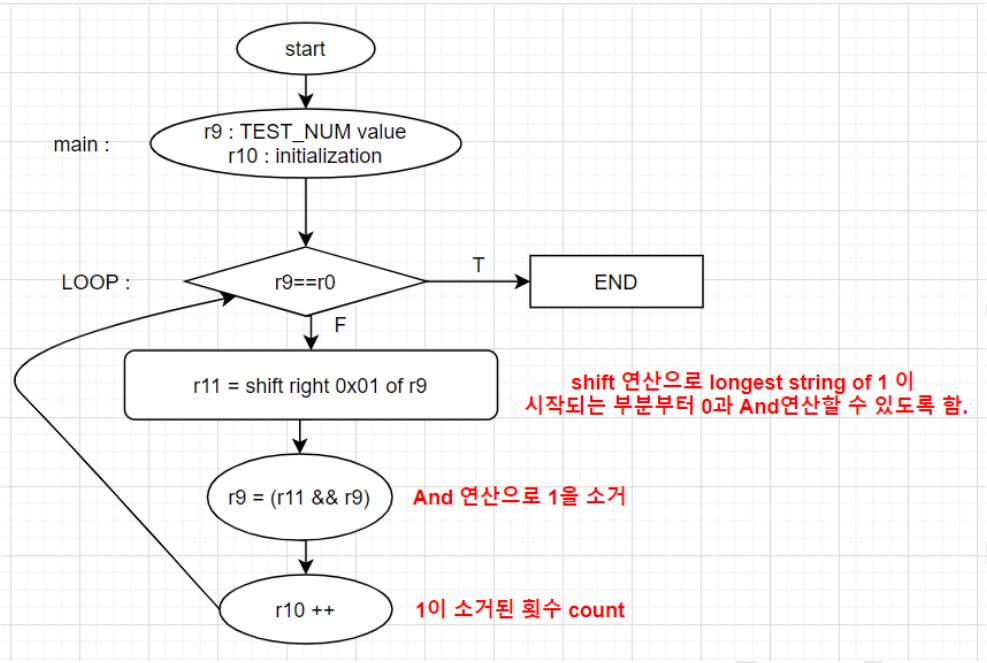
전자 HW 설계 – 실습 보고서

이름: 박관영 (2016124099)

✓ Part I

동작 원리



> > > > >

R9 : TEST_NUM value

R10 : longest string of 1s 저장

R11 : shifted value of r9

TEST_NUM에 미리 저장해 둔 숫자를 2진수로 나타냈을 때, 1이 연속되어 나오는 부분 중 가장 긴 부분의 길이를 계산하는 program이다.

<GPR setting>

R9에 TEST_NUM의 숫자를 읽어 그 값을 저장한다. 결과값(longest string of 1s)을 저장할 r10을 0으로 초기화한다.

<LOOP>

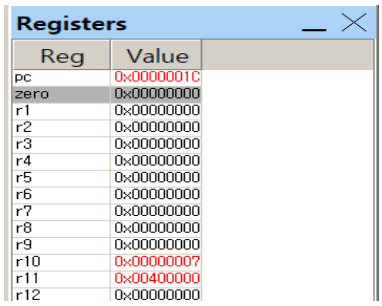
R9를 통해 전달받은 값으로 연산을 수행한다. R11에 r9의 값을 1칸 shifting하여 저장한 뒤 r9와 r11을 and 연산하여 r9에 저장한다. R9의 값이 0이 될 때까지 LOOP을 반복하다가 0이되면 r10을 통해 결과를 확인할 수 있다. 이 때 shift, &&으로 longest string of 1s을 확인할 수 있는 이유는 string of 1s가 끝나는 자릿수의 값은 0일 것이므로 shift right 후 and를 취해주면 string of 1s의 한자리가 소거된다. 또한, string of 1s가 여러 개 있을 때 가장 긴 string이 소거되어야 전체 값이 0이 될 것이므로 r9의 값이 0이 되었을 때까지의 연산이 몇 번 수행되었는지 count하여 longest string of 1s를 알아낼 수 있다.

구현 코드 설명

```
.text          # 다음 segment가 program이라는 assembler directive
.global _start
_start:
        ldw r9, TEST_NUM(r0)   # TEST_NUM에 있는 word를 r9으로 load.
        mov r10, r0            # 결과값 저장할 r10을 초기화
```

```
LOOP: beq r9, r0, END          # r9가 0이 될때까지 LOOP 반복.
      srli r11, r9, 0x01       # r9 1자리 shift하여 r11에 저장
      and r9, r9, r11          # and연산 수행
      addi r10, r10, 0x01      # LOOP 반복횟수 count
      br LOOP

END: br END

TEST_NUM: .word 0x3fabedef
          .end
```

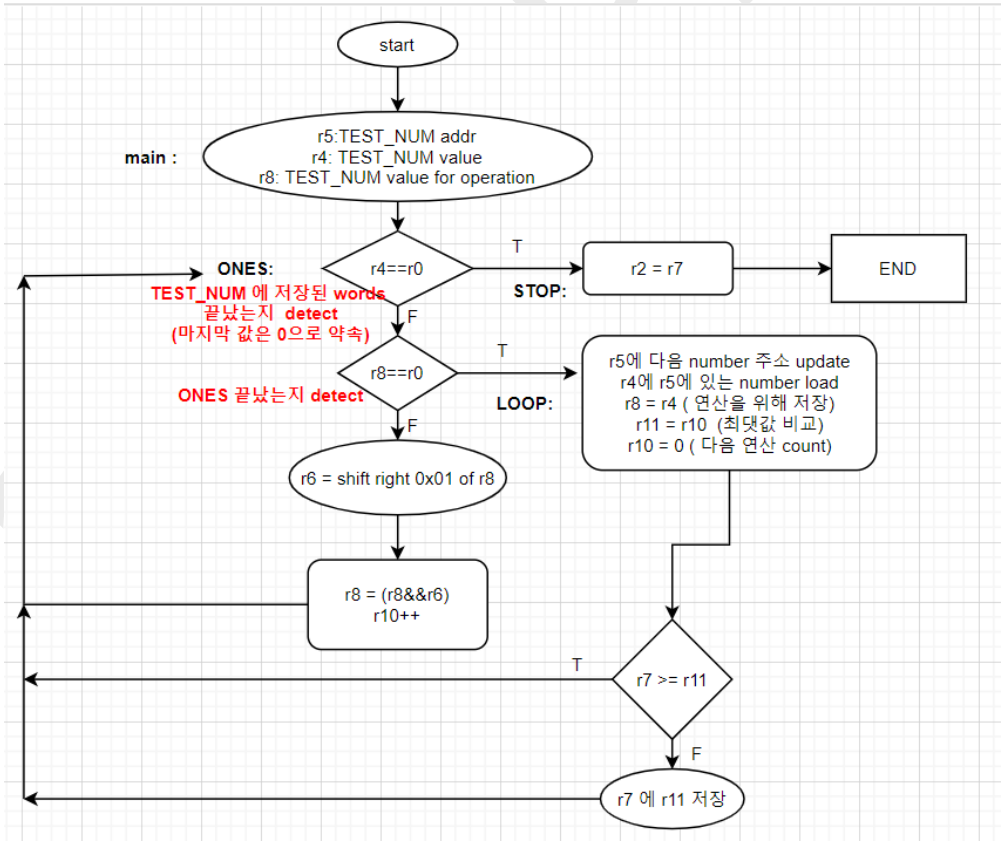
결과 및 토의



R10에 저장된 값으로 TEST_NUM에 저장된 값의 longest string of 1s를 알 수 있다.

✓ Part II

동작 원리



>>>>>

R2 : return value

R4: TEST_NUM value

R5: TEST_NUM addr

R6: shifted value (at ONES)

R7: max value of longest string of 1s

R8: TEST_NUM value for operation

R10: longest string of 1s

R11: longest string of 1s for compare to max value

Part1과 동일하게 longest string of 1s를 구하는 program이다. 여러 words의 값을 구하고 비교해야 하므로 비교를 위한

register가 더 필요하다. 또한 longest string of 1s를 계산하는 subroutine ONES를 반복하면서 LOOP를 통해 각각의 결과를 최댓값과 비교, 저장할 수 있었다. R4, r8을 통해 argument passing을 하였고, r2를 return value reg로 설정하였다.

<GPR setting>

TEST_NUM 주소를 r5에 저장하고, 각 주소의 값을 r4, r8에 저장하여 argument passing하였다. R4에 저장된 값으로 TEST_NUM의 마지막 words를 detect하고 r8에 저장된 값으로 shift, && 연산을 수행할 수 있다. R8이 0이 된다면 한번의 연산이 끝난 것을 detect할 수 있다.

<ONES>

Part1의 LOOP와 동일한 구조의 연산을 수행한다. 그러나 여러 숫자의 연산을 하게 되므로 r4를 통해서 모든 TEST_NUM을 비교했는지 판별하는 instruction이 추가되었다.

<LOOP>

TEST_NUM에 저장된 각 words의 longest string of 1s를 비교하고 그 최댓값을 저장한다. R5에 다음 word의 주소를 update, r4, r8에 각 word의 값을 update한다. 또한, r10의 결과값과 최댓값의 비교를 위해 r11에 결과값을 저장하고 r10은 다음 연산 counting을 위해 0으로 초기화 한다. R11과 r7 (최댓값 저장 reg)을 비교한 후 r11의 값이 더 크면 r7을 r11의 값으로 update 하게 된다.

<STOP>

TEST_NUM의 모든 word를 연산하게 되면 subroutine을 종료한다. 이 때 subroutine의 return value는 r2를 통해 passing한다. 즉 r7의 값을 r2에 copy하여 결과 확인을 용이하게 한다.

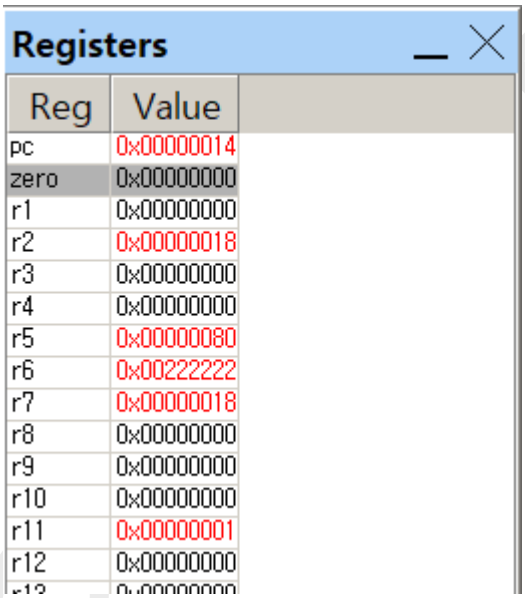
구현 코드 설명

```
.text
.global _start
_start:
        movia r5, TEST_NUM     # r5 - addr of TEST_NUM
        ldw r4, (r5)           # r4 - value of TEST_NUM
        mov r8, r4             # r8 - argument reg of ONES r4와 같은 값.  Shift/and연산 수행을 위해 저장한다.
        call ONES

END:   br END

ONES: beq r4, r0, STOP      # TEST_NUM 값이 0 이면 STOP
      beq r8, r0, LOOP      # 연산결과가 0이면
      srli r6, r8, 0x01     # r6 - shift value
      and r8, r8, r6
      addi r10, r10, 0x01   # r10 - longest string of 1s
      br ONES

LOOP:   addi r5, r5, 4      # to read each word of TEST_NUM
```

```
        ldw   r4, (r5)          # r4 update
        mov r8, r4              # r8 update
        mov r11, r10             # r11 - to compare with max
       mov r10, r0             # r10 다음 연산 count를 위해 초기화
        bge   r7, r11, ONES      # r7 - max value store
        mov r7, r11             # max보다 큰 값이 나타나면 저장
        br ONES

STOP:   mov r2, r7              # r2로 최댓값 return
        ret

TEST_NUM: .word 0x00aaaaaa, 0x00bbbbbb, 0x00cccccc, 0x00dddddd, 0x00eeeeee
            .word 0x00ffffff, 0x00111111, 0x00222222, 0x00333333, 0x00444444, 0
            .end
```

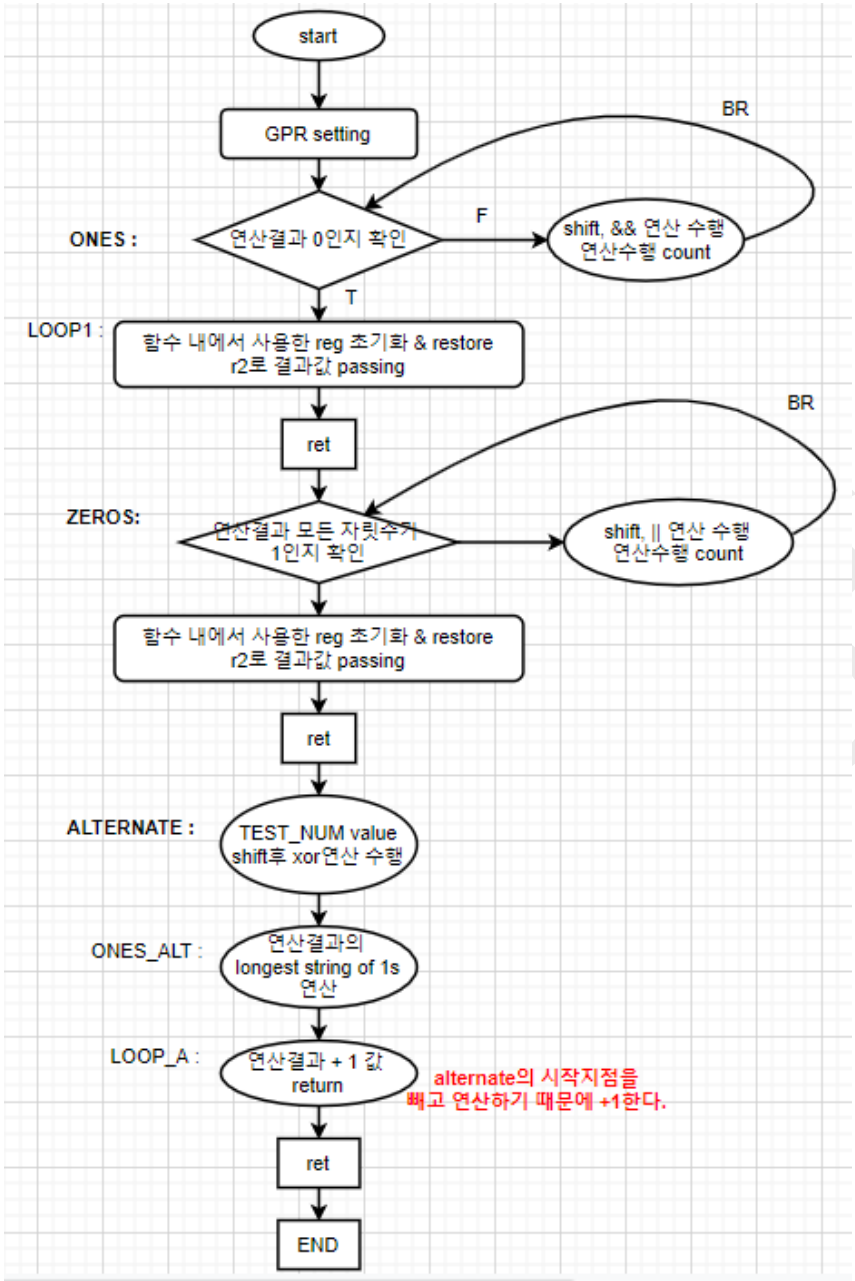
결과 및 토의

R2를 통해 return value를 passing하므로 r2의 값을 통해 TEST_NUM에 저장된 10개의 수들의 longest string of 1s를 확인할 수 있다. 이 때 r2의 값이 32보다 큰 값을 가진다면 code를 만들 때 오류가 있었음을 알 수 있다. Word는 32bit이므로 longest string of 1s는 32보다 큰 값을 가질 수 없기 때문이다. Part 2 program은 subroutine에서 다시 subroutine call을 하는 경우가 없으므로 subroutine 내부에서 사용한 register를 초기화 하지 않았다. 따라서 r11을 통해 마지막 숫자의 연산결과를 확인할 수 있고, r7에는 r2와 같은 값이 저장되어 있는 것을 확인할 수 있다.



✓ Part III

동작 원리



>>>>>

R2: return value

R4: TEST_NUM addr

R5: TEST_NUM value

R10: longest string of 1s

R11: longest string of 0s

R12: longest string of alternating 0s, 1s

Other registers - subroutine 내에서 사용 후 초기화

Part1의 program처럼 하나의 word에 대해서 longest string을 계산한다. 다른 점은 longest string of 1s 이외에 0s와 alternating 1s, 0s 을 구하는 subroutine도 구현해야 한다는 점이다. R4, r5를 통해 TEST_NUM의 addr과 value를 argument passing하고 r2를 통해 return value passing한다. R10에는 ONES의 return value, r11에는 ZEROS, r12에는 ALTERNATE의 return value를 저장하게 된다.

<GPR setting>

Argument passing을 위한 r4, r5를 TEST_NUM의 주소와 word로 setting한다.

<ONES>

Part 1의 LOOP와 같은 구조의 subroutine이다. 하나의 word에 대해서만 연산을 수행하면 되므로 각 register의 number외에는 모든 동작이 일치한다. 그러나 subroutine 내 사용하였던 regisiter를 다른 subroutine에서 재사용하기 위해 초기화하는 과정이 추가된다.

<ZEROS>

ONES의 구현에서 아이디어를 얻어 연속된 1을 찾는 방법을 반대로 적용하여 구현하였다. 연속된 0이 있을 경우 shift right 후 or연산을 취해주면 연속된 0s의 첫 자리가 소거된다. 모든 0이 소거되면 연산 결과값을 2진수로 나타냈을 때 모든 자리가 1이 된다. 그 후 연산 결과값에 1을 더한 값과 원래 연산 값을 AND연산 취해주면 0이 된다. Shift 후 or 연산을 수행한 후 결과 값과 그 값에 1을 더한 값을 and연산 취해주어 0이 될 때까지 반복한다. 또한 매 회 반복할 때마다 counting을 하여 longest string of 0s를 계산할 수 있다. ONES와 구현이 거의 비슷하여 사용하는 레지스터의 개수도 비슷하지만, addi 연산을 통해 1을 더한 값을 저장하는 instruction이 추가되어야 하므로 subroutine 내 레지스터 개수가 하나 더 많게 된다. ONES와 마찬가지로 subroutine내에서 사용하였던 register들은 모두 0으로 초기화한다.

<ALTERNATE>

1과 0이 반복되는 가장 긴 부분을 찾는 subroutine이다. 매 자리가 바뀌는 부분을 찾는 것이므로 xor연산이 적절하다. Shift 한 후 원래 값과 xor연산을 취해주면 1,0이 반복되는 부분이 1로 되고 나머지 부분은 0의 값을 가질 것이다. 따라서 한번의 shift, xor연산을 취한 후 ONES에서와 마찬가지로 longest string of 1s를 계산하면 된다. 그러나 이 경우 alternating string의 시작지점을 계산하지 못하게 된다. alternating string은 1과 0이 반복되는 부분이므로 앞뒤로 적어도 한 digit은 같은 수가 나오거나 word의 시작 혹은 끝부분이 될 것이다. Shift 후 xor연산을 하게 되었을 때 alternating string의 첫 자리는 detect되지 못한다. 따라서 결과값에 1을 더해주어 longest alternating string을 계산한다.

구현 코드 설명

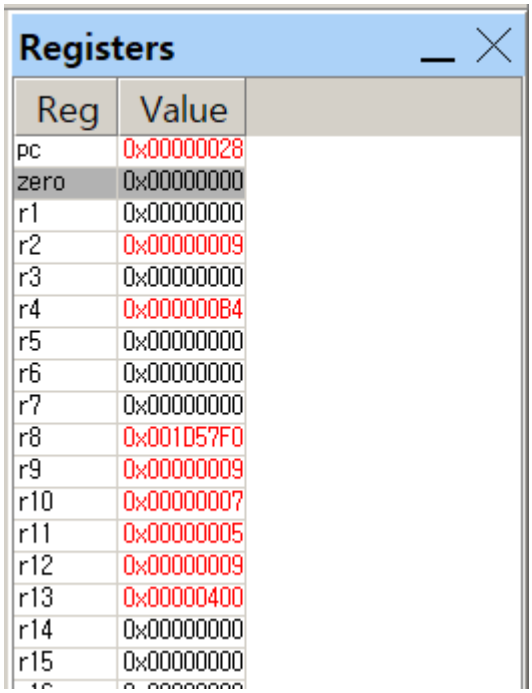
```
.text
.global _start
_start:
        movia r4, TEST_NUM      # r4 - TEST_NUM addr
        ldw r5, (r4)            # r5 - TEST_NUM value
        mov r2, r0              # return value
        call ONES
        mov r10, r2             # r10 - longest 1 string
        call ZEROS
        mov r11, r2             # r11 - longest 0 string
        call ALTERNATE
        mov r12, r2             # r12 - longest alternate string

END: br END

ONES:   beq r5, r0, LOOP1       # 연산결과가 0이 되었다면 종료
        srli r8, r5, 0x01       # r8 - shifted value store
        and r5, r5, r8
        addi r9, r9, 0x01        # r9 - counter of string 1s
```

```
            br ONES

LOOP1:      ldw r5, (r4)            # TEST_NUM value restore
            mov r2, r9              # return value
            mov r8, r0              # 사용한 reg 초기화
            mov r9, r0              # 사용한reg 초기화
            ret

ZEROS:      addi r13, r5, 0x01      # r13 - detect when r5's all digit 1
            and r13, r13, r5
            beq r13, r0, LOOP0      # r13 값이 0이 되었다면 r5의 모든 digit이 1이므로 연산 종료
            srli r8, r5, 0x01       # r5 - TEST_NUM value, r8- shifted value
            or r5, r5, r8           # 0인 부분 소거
            addi r9, r9, 0x01       # r9 - count string of 0s
            br ZEROS

LOOP0:      ldw r5, (r4)            # TEST_NUM value restore
            mov r2, r9              # return value
            mov r8, r0              # 사용한 reg 초기화
            mov r9, r0              # 사용한 reg 초기화
            mov r13, r0
            ret

ALTERNATE:  srli r8, r5, 0x01       # shift,
            xor r5, r5, r8          # xor연산 수행
            br ONES_ALT

ONES_ALT:   beq r5, r0, LOOP_A      # ONES와 동일한 역할 수행. 내부 사용 reg만 바뀜
            srli r13, r5, 0x01      # r13 - shifted value store
            and r5, r5, r13
            addi r9, r9, 0x01       # r9 - counter of string 1
            br ONES_ALT

LOOP_A: addi r9, r9, 0x01   # detect for start point
        mov r2, r9          # return value
        ret

TEST_NUM: .word 0b11101010101111111100000
          .end
```

결과 및 토의

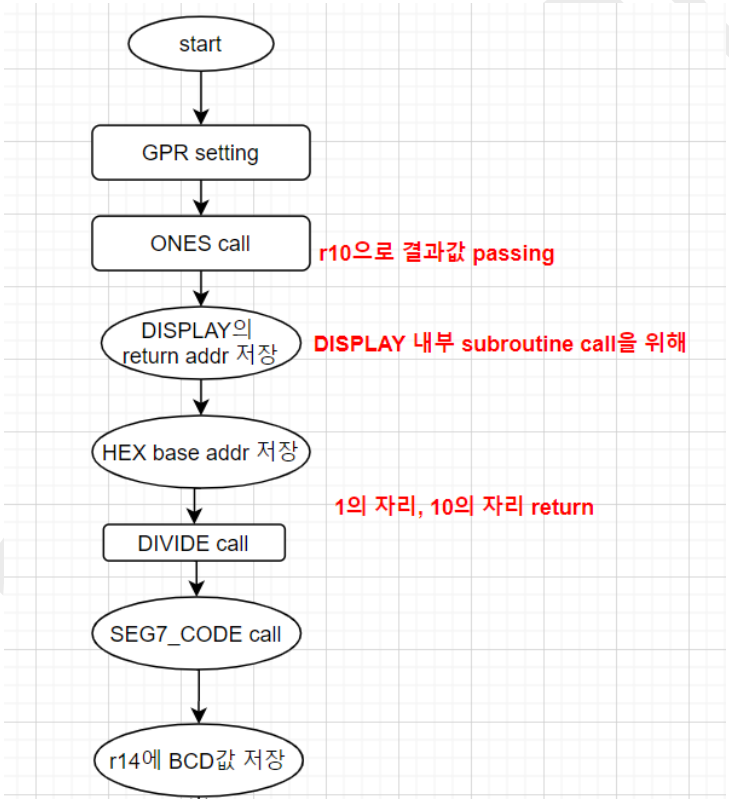
ONES를 구현한 아이디어를 기반으로 ZEROS, ALTERNATE를 구현하였다. 각 내부에서 사용한 register는 subroutine이 끝날 때 초기화 후 다음 subroutine에서 재사용하였으므로 마지막으로 call된 subroutine에 의한 값을 저장하고 있을 것이다. 따라서 각 subroutine의 return value를 저장하고 있는 r10~r12를 중점으로 확인하여 결과값이 제대로 나왔는지를 확인할 수

있었다.

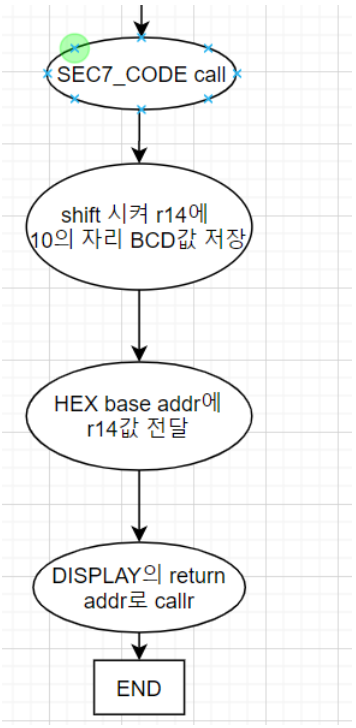


✓ Part IV

동작 원리



➔



>>>>>

R2: return value

R4: TEST_NUM value/ argument passing

R5: TEST_NUM addr

R6: TEST_NUM value for operation

R10: longest string of 1s

R14: BCD value of result

R16: DISPLAY return address

R17: base addr of HEX

Other registers - subroutine 내에서 사용 후 초기화

Part4의 프로그램은 part2의 결과를 HEX에 표시하는 프로그램이다. LAB1에서 사용하였던 DIVIDE subroutine과 LAB2의 part2에서 사용한 여러 words의 longest string of 1s 중 max 값을 찾아내는 ONES subroutine을 다시 사용하였다. 또한 DIVIDE의 결과값을 BCD코드로 바꾸는 subroutine도 사용하였다.

<GPR setting>

Part2와 마찬가지로 TEST_NUM의 주소와 word를 r4, r5를 통해 argument passing하고, word의 마지막을 detect하기 위해 r6에도 r4와 같은 값을 저장하여 passing한다.

<ONES>

Part2와 똑같이 구현하였지만, 이후 ONES를 call하는 부분에서 r10으로 return value를 받아 사용하므로, 결과값(max value)를 r10을 통해 passing 한다. 또한 내부에서 사용한 register의 이후 재사용을 위해 초기화 하는 과정이 추가되었다.

<DIVIDE>

LAB1과 똑같이 구현하였지만, 내부 register number가 조금 바뀌었고 r4를 통해 argument가 pass되고 return value가 r2(일의자리), r3(10의 자리) 두개를 통해 pass된다.

<SEG7_CODE>

미리 저장된 0-9 BCD값을 return 하는 subroutine이다. R4를 통해 argument가 pass되고 그 값에 해당하는 만큼 주소를 이동하여 적절한 BCD값을 r2를 통해 return 하게 된다. BCD는 8자리 2진수이므로 byte단위로 저장하였고, 10개의 값이 저장되어 있으므로, .skip 2를 통해 word alignment를 유지해준다.

<DISPLAY>

Subroutine 내에서 또다른 subroutine을 call하게 되면 ra에 저장되는 return address가 subroutine 내부에서 call된 subroutine의 return address로 고정되어 subroutine을 call한 subroutine의 return address는 사라지게 된다. 따라서 DISPLAY의 return address를 r16에 따로 저장하여 DISPLAY가 종료된 후 r16의 값으로 return하게 하였다. R17에 HEX의 base address를 저장하여 hex에 원하는 BCD 값을 전달할 수 있게 하였다. 이후 ONES의 결과값을 받아 DIVIDE, SEG7_code를 차례로 call하여 ONES의 결과값을 BCD값으로 바꿔주었다. 이 때, 10의 자리 값은 저장 후 8bit shift하여 1의 자리 값을 저장한 reg와 or연산을 취해 한reg에 (10의 자리 bcd | 1의 자리 bcd) 와 같이 저장될 수 있게 하였다. 이 값을 r17(hex base addr)에 저장할 때는 cache를 사용하게 되므로 stwio instruction을 사용하였다.

구현 코드 설명

```
.text
.global _start
_start: movia r5, TEST_NUM
        ldw r4, (r5)
        mov r6, r4
        call ONES
        call DISPLAY

END: br END

ONES:    beq r4, r0, STOP
         beq r6, r0, LOOP
         srli r8, r6, 0x01    # r8 - shift value
        and r6, r6, r8
        addi r7, r7, 0x01    # r7 - longest string of 1's
        br ONES

LOOP:    addi r5, r5, 4       # to read each word of TEST_NUM
        ldw r4, (r5)
         mov r6, r4
         mov r9, r7           # r9 - to compare with max value
         mov r7, r0
        bge r10, r9, ONES     # r10 - max value store
         mov r10, r9

         br ONES


STOP:  mov r5, r0       # 사용한 reg 초기화
       mov r7, r0
       mov r8, r0
       mov r9, r0
       ret

DIVIDE:  mov r5, r4      # r4를 통해 argument passed
         movi r7, 10     # r7 - divider
         movi r8, 0      # r8 초기화
         movia r9, RESULT    # r9 - 결과 저장
        mov r2, r0
        mov r3, r0
        movi r11, 0

CONT:  blt r5, r7, LOOP2  # divider보다 작을 때 까지 뺄셈 수행
```

```
        sub r5, r5, r7          # 나머지 계산
        addi r8, r8, 1          # 몫 계산
        br CONT

LOOP2:  stb r5, (r9)        # 계산된 나머지(n번째 자리의 숫자) r9에 저장
        addi r9, r9, 1  # 다음자리 저장을 위해 주소+1
        mov r5, r8      # 다음 계산을 위해 몫 넘겨줌
        mov r11, r8     # divider보다 큰지 비교하기 위해 r11에도 몫 저장
        movi r8, 0      # 다음 몫 계산을 위해 0으로 초기화
        blt r11, r7, LOOP3    # divider보다 크다면 연산 다시 수행
        br CONT
LOOP3:  stb r11, (r9)     # 마지막 연산의 결과 저장
        ldb r3, (r9)      # r9 주소에 있는 값을 ldb하여 r3에 10의 자리 load
        ldb r2, RESULT(r0)  # 결과값이 저장된 RESULT의 처음주소를 ldb하여 r2에 1의 자리 load

DIV_END:  mov r5, r0        # 사용한 reg 초기화
          mov r7, r0
          mov r8, r0
          mov r9, r0
          mov r11, r0
          ret

SEG7_CODE: movia r15, BIT_CODES    # starting address of bit codes
           add r15, r15, r4       # index into the bit codes
           ldb r2, (r15)         # read the bit code needed for our digit
           ret

BIT_CODES: .byte 0b00111111, 0b00000110, 0b01011011, 0b01001111, 0b01100110
           .byte 0b01101101, 0b01111101, 0b00000111, 0b01111111, 0b01100111
           .skip  2  # pad with 2 bytes to maintain word alignment

DISPLAY: mov r16, ra              # DISPLAY의 return address 저장
         movia r17, 0xff200020     # HEX의 base address 저장
         mov r4, r10              # display r10 on HEX1−0
         call DIVIDE             # ones digit will be in r2; tens digit in r3
         mov r4, r2              # pass ones digit to SEG7_CODE
         call SEG7_CODE
         mov r14, r2             # save bit code
         mov r4, r3              # retrieve tens digit, pass to SEG7_CODE
         call SEG7_CODE
         slli r2, r2, 8          # 10의 자리 표현을 위해 shift
         or r14, r14, r2         # bit code for tens|bit code for ones
         stwio r14, (r17)        # store at hex base addr, cache를 사용하므로 stwio 사용

FIN: callr r16        # return to return address of DISPLAY ( main )
```

```
RESULT: .skip 4

TEST_NUM: .word 0x00aaaaaa, 0x00bbbbbb, 0x00cccccc, 0x00dddddd, 0x00eeeeee
          .word 0x00ffffff, 0x00111111, 0x00222222, 0x00333333, 0x00444444, 0
          .end
```

결과 및 토의

Part4에서는 이전에 사용했던 subroutine을 사용하므로 argument passing, return value passing만 신경 써서 subroutine을 구현했다. 또한 subroutine 내에서 또다른 subroutine을 call할 경우에는 return address를 또다른 register에 저장한 후 call을 해야 했다. Part2의 결과를 bcd 값으로 바꾸어 출력하므로 16진수의 결과값이 10진수의 값으로 HEX에 출력되는 것을 확인할 수 있고, part2의 결과와 비교하여 그 값이 맞는 지를 확인할 수 있다. 설정해준 TEST_NUM의 결과는 24로 출력됨을 알 수 있었다. 또한 hex의 base address에 값을 넘겨줄 때는 cache를 거쳐 저장해야 했다. -io를 붙여 instruction을 구현했다. R10의 결과값이 10진수로 hex0-1에 표현된 것을 확인할 수 있다. 이 값은 part2의 결과값과 일치한다.

Registers

Reg	Value
pc	0x00000018
zero	0x00000000
r1	0x00000000
r2	0x00005B00
r3	0x00000002
r4	0x00000002
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000018
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00005B66
r15	0x000000EE
r16	0x00000018
r17	0xFF200020
r18	0x00000000
r19	0x00000000
r20	0x00000000
r21	0x00000000
r22	0x00000000
r23	0x00000000
et	0x00000000
bt	0xFFFFFFFF
gp	0x00000000
sp	0x00000000
fp	0x00000000
ea	0x00000000
ba	0xFFFFFFFF
ra	0x00000130