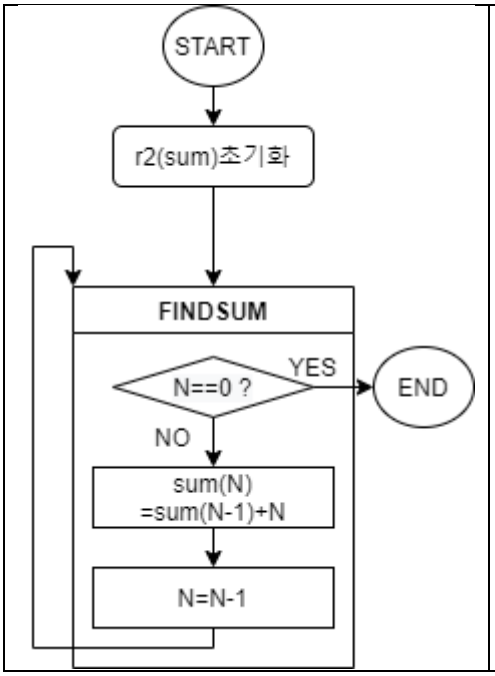
전자 HW 설계 – 실습 보고서

이름: 양해찬 (2016124145)

✓ Part I

동작 원리



1부터 N까지의 수의 합을 출력하는 코드이다.
먼저 r2(sum)을 0으로 초기화 하고, 지정한 N에 맞게 더하기를 반복한다.
FINDSUM에서 r4(N)을 1식 빼 주며 loop을 도는데, r4가 0이 되면 FINDSUM을 call 한 바로 뒤인 END로 분기한다.

예를 들어, 만약 N이 9라고 가정하면 먼저 N이 9인 상태로 sum에 더해 sum의 초기값은 0이므로 9가 된다. 다음으로 N에 1을 뺀 후 다시 sum에 더한다. 위와 같은 내용을 N이 0이 될 때 까지 반복하는 과정을 간략히 나타내면
N:9, sum:0 -> N:8, sum:9 -> N:7, sum:16 -> N:6, sum:23 -> N:5, sum:29
-> N:4, sum:34 -> N:3, sum:38 -> N:2, sum:41 -> N:1, sum:44 -> N:0, sum:45

구현 코드 설명

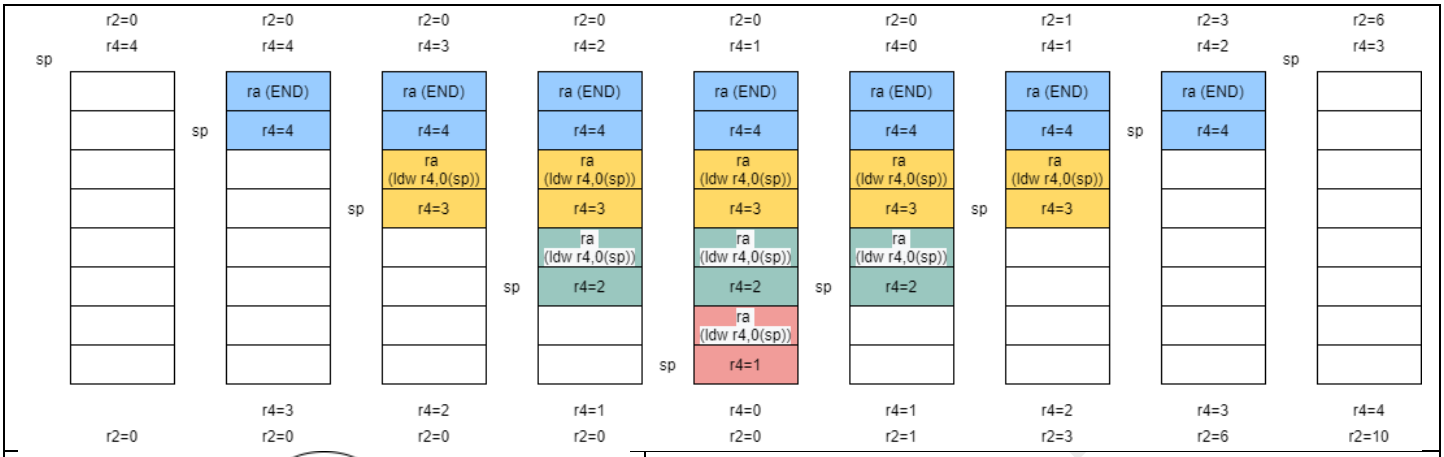
```
        .text
        .global _start

_start: mov     r2,     r0          #r2(sum저장할 레지스터)초기화
        movia   r13,    N           #r13에 N주소값 저장
        ldw     r4,     0(r13)      #r4에 N에 저장된 숫자 load
        call    FINDSUM             #FINDSUM 호출

END:    br      END

FINDSUM: bne    r4,     r0,     LOOP    #r4가 0이 아니라면 LOOP으로 br
        ret                             #call FINDSUM 뒤로 return
LOOP:   add     r2,     r2,     r4      #r2=r2+r4(sum(N)=sum(N-1)+N)
        subi    r4,     r4,     1       #N=N-1
        br      FINDSUM                 #FINDSUM 반복

N:      .word   9
        .end
```

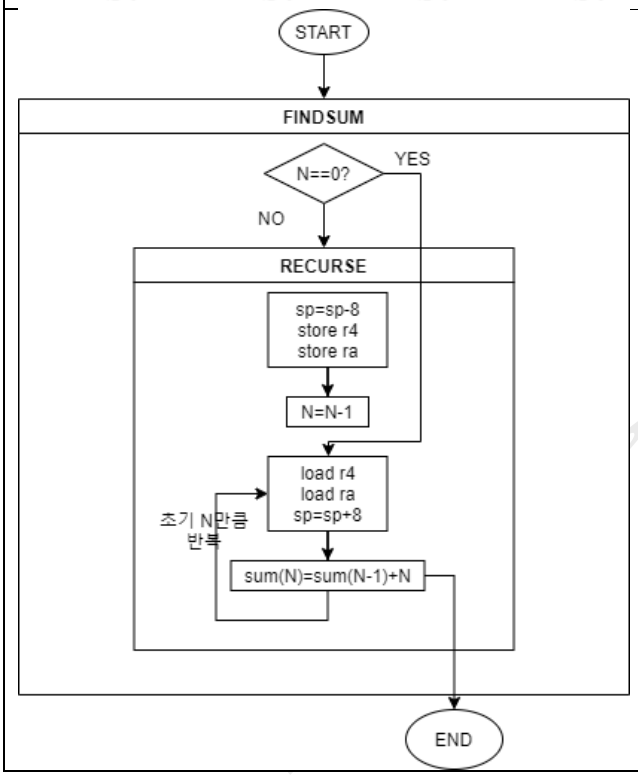
pc	0x00000014	r18	0x00000000	코드에서 N은 9이므로 Sum=9+8+7+6+5+4+3+2+1=45이다. Sum이 저장된 r2를 보면 0x2D로 값이 잘 나온 것을 볼 수 있다. r4를 보면 0이 출력되는데, r4가 0이 될 때까지 loop를 돌았기 때문이다. r13에는 N의 주소값, ra에는 END의 주소값이 출력되고 있다.
zero	0x00000000	r19	0x00000000	
r1	0x00000000	r20	0x00000000	
r2	0x0000002D	r21	0x00000000	
r3	0x00000000	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	0x00000000	ea	0x00000000	
r11	0x00000000	ba	0xFFFFFFFF	
r12	0x00000000	ra	0x00000014	
r13	0x0000002C	status	0x00000000	
r14	0x00000000	estatus	0x00000000	
r15	0x00000000	bstatus	0xFFFFFFFF	
r16	0x00000000	ienable	0x00000000	
r17	0x00000000	ipending	0x00000000	
r18	0x00000000	cpuid	0x00000000	

전자 HW 설계 – 실습 보고서

✓ Part II

동작 원리.





Part2는 Part1과 같은 내용을 재귀적으로 구현하는 코드이다. Stackpointer sp에 0x20000값을 저장하고, FINDSUM을 호출한다. FINDSUM에서 r4 (N)이 0이 되기 전까지 RECURSE를 호출한다. RECURSE에서는 N이 0이 될 때까지 FINDSUM을 호출한 뒤 정확한 위치로 return 하기 위해서 마지막으로 call 한 뒤의 주소값 ra를 먼저 sp에 store해서 안전하게 저장해 둔다. 그리고 return 하는 과정에서 덧셈 연산이 제대로 수행될 수 있도록 현재의 N값 또한 sp에 store한다. r4가 0이 되면 r2(sum)을 0으로 만든 후 차례차례 sp에서 이전에 저장했던 r4(N) 값들을 꺼내 r2와 더한다. 저장했던 ra값 또한 불러와서 해당 부분으로 return하며 반복한다. 마지막 ra값인 END로 return하고 코드는 끝이 난다.

구현 코드 설명

```
        .text
        .global _start

_start:   movia   r13,    N              #r13에 N주소값 복사
          movia   sp,     0x20000          #sp에 주소값으로 0x20000복사
          ldw     r4,     0(r13)         #r4에 N에 저장된 숫자 load
          call      FINDSUM              #FINDSUM호출

END:        br      END
```

```
FINDSUM:    bne     r4,     r0,     RECURSE     #r4(N)이 0이아니면 RECURSE로 br
            mov     r2,     r0                  #r4(N)이 0이면 r2에 0복사
            ret                                 #RECURSE내 FINDSUM호출 뒤로 return

RECURSE:    subi    sp,     sp,     8           #sp에 r4와 ra를 저장하기위해 8byte를 뺀다
            stw     r4,     0(sp)               #sp에 r4 store
            stw     ra,     4(sp)               #sp에 ra store
            subi    r4,     r4,     1           #N=N-1
            call    FINDSUM                     #FINDSUM 호출
            ldw     r4,     0(sp)               #sp에 저장했던 r4값 load
            ldw     ra,     4(sp)               #sp에 저장했던 ra값 load
            addi    sp,     sp,     8           #저장했던 값을 다시 load했으므로 뺀 공간만큼 더해서 없애줌
            add     r2,     r4,     r2          #SUM(N)=N+SUM(N-1)
            ret                                 #맨처음 FINDSUM 호출한 뒤로 return


N:          .word   4
            .end
```

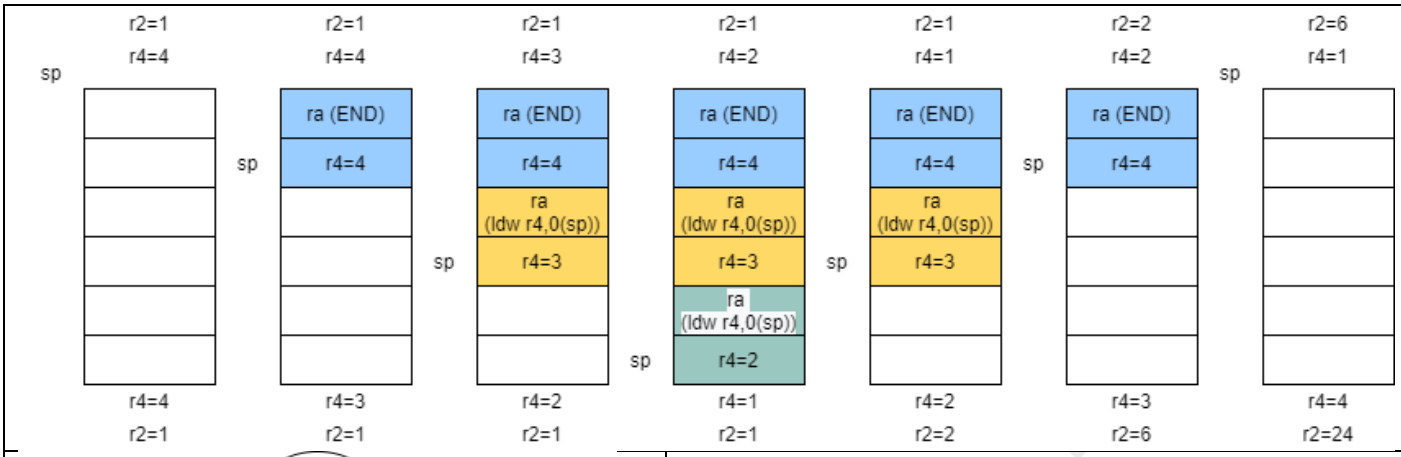
결과 및 토의

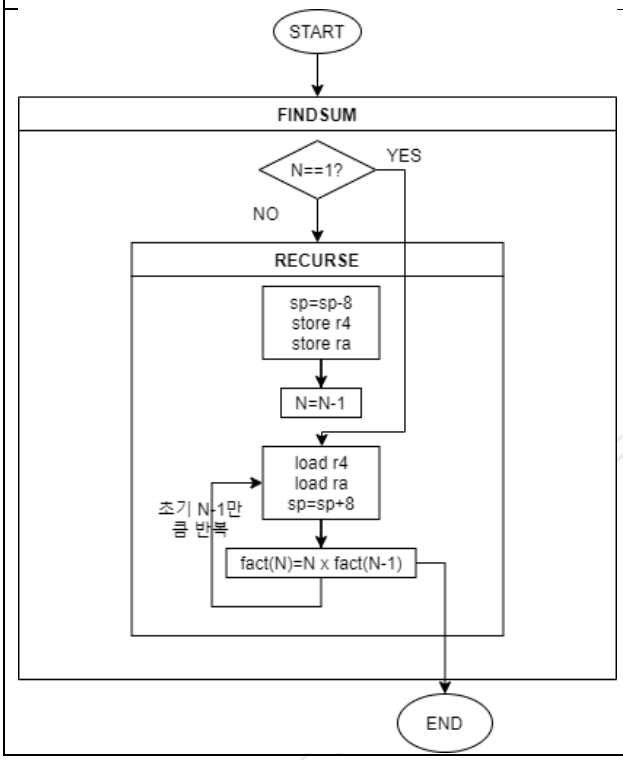
pc	0x00000018	r19	0x00000000
zero	0x00000000	r20	0x00000000
r1	0x00000000	r21	0x00000000
r2	0x0000000A	r22	0x00000000
r3	0x00000000	r23	0x00000000
r4	0x00000004	et	0x00000000
r5	0x00000000	bt	0xFFFFFFFF
r6	0x00000000	gp	0x00000000
r7	0x00000000	sp	0x00020000
r8	0x00000000	fp	0x00000000
r9	0x00000000	ea	0x00000000
r10	0x00000000	ba	0xFFFFFFFF
r11	0x00000000	ra	0x00000018
r12	0x00000000	status	0x00000000
r13	0x00000050	estatus	0x00000000
r14	0x00000000	bstatus	0xFFFFFFFF
r15	0x00000000	ienable	0x00000000
r16	0x00000000	ipending	0x00000000
r17	0x00000000	cpuid	0x00000000
r18	0x00000000		

코드에서 N이 4이므로
sum=4+3+2+1=10 인데, sum이 저장된 r2를 보면 0xA로 값이 잘 나오는 것을 볼 수 있다.
r4가 4인 이유는 stackpointer는 stack방식으로 동작하는데, 가장 먼저 들어간 것이 가장 마지막에 나오는 방식이다. 즉 가장 먼저 들어간 r4인 4가 가장 마지막에 다시 r4로 load되어 4가 출력되고 있는 것이다.
r13에는 N의 주소값이 들어가 있다.
sp에는 초기에 설정했던 주소값인 0x20000에서 factorial을 모두 수행 한 후 다시 0x20000으로 되돌려 놓았기 때문에 해당 값이 출력 되고 있다.
ra에는 r4와 같은 이유로 가장 처음 저장되었던 값인 END에 해당하는 주소값 0x18이 출력되고 있다.

✓ Part III

동작 원리





Part3는 Factorial을 재귀적으로 구현한 코드이다.
Stackpointer sp에 0x20000값을 저장하고, FINDFACT을 호출한다. FINDFACT에서 r4 (N)이 1이 되기 전까지 RECURSE를 호출한다. RECURSE에서는 N이 1이 될 때까지 FINDSUM을 호출한 뒤 정확한 위치로 return 하기 위해서 마지막으로 call 한 뒤의 주소값 ra를 먼저 sp에 store해서 안전하게 저장해 둔다. 그리고 return 하는 과정에서 곱셈 연산이 제대로 수행될 수 있도록 현재의 N값 또한 sp에 store한다. r4가 0이 되면 r2(fact)을 1로 만든 후 차례차례 sp에서 이전에 저장했던 r4(N) 값들을 꺼내 r2와 곱한다. 저장했던 ra값 또한 불러와서 해당 부분으로 return하며 반복한다. 마지막 ra값인 END로 return하고 코드는 끝이 난다.

.

구현 코드 설명

```
        .text
        .global _start

_start:  movi   r5,    1          #FINDFACT에서 조건문에서 사용할 1
         movia  r13,   N           #r13에 N주소값 복사
         movia  sp,    0x20000     #sp에 주소값으로 0x20000복사
         ldw    r4,    0(r13)      #r4에 N에 저장된 숫자 load
         call   FINDFACT          #FINDFACT호출
```

```
END:        br      END

FINDFACT:   bne     r4,     r5,     RECURSE     #r4(N)가 1이 아니면 RECURSE호출
            movi    r2,     1                   #r2=1
            ret                                 #recurse 내에서 FINDFACT호출한 뒤로 return

RECURSE:    subi    sp,     sp,     8           #sp에 r4와 ra를 저장하기위해 8byte를 뺀다
            stw     r4,     0(sp)               #sp에 r4 store
            stw     ra,     4(sp)               #sp에 ra store
            subi    r4,     r4,     1           #N=N-1
            call    FINDFACT                    #FINDFACT 호출
            ldw     r4,     0(sp)               #sp에 저장했던 r4값 load
            ldw     ra,     4(sp)               #sp에 저장했던 ra값 load
            addi    sp,     sp,     8           #저장했던 값을 다시 load했으므로 뺀 공간만큼 더해서 없애줌
            mul     r2,     r4,     r2          #FACT(N)=N*FACT(N-1)
            ret                                 #맨처음 FINDFACT 호출한 뒤로 return


N:          .word   4
            .end
```

결과 및 토의

pc	0x0000001C	r19	0x00000000
zero	0x00000000	r20	0x00000000
r1	0x00000000	r21	0x00000000
r2	0x00000018	r22	0x00000000
r3	0x00000000	r23	0x00000000
r4	0x00000004	et	0x00000000
r5	0x00000001	bt	0xFFFFFFFF
r6	0x00000000	gp	0x00000000
r7	0x00000000	sp	0x00020000
r8	0x00000000	fp	0x00000000
r9	0x00000000	ea	0x00000000
r10	0x00000000	ba	0xFFFFFFFF
r11	0x00000000	ra	0x0000001C
r12	0x00000000	status	0x00000000
r13	0x00000054	estatus	0x00000000
r14	0x00000000	bstatus	0xFFFFFFFF
r15	0x00000000	ienable	0x00000000
r16	0x00000000	ipending	0x00000000
r17	0x00000000	cpuid	0x00000000
r18	0x00000000		

코드에서 N이 4이므로
fact=4x3x2x1=24 인데, sum이 저장된 r2를 보면 0x18로 값이 잘 나오는 것을 볼 수 있다.
r4가 4인 이유는 stackpointer는 stack방식으로 동작하는데, 가장 먼저 들어간 것이 가장 마지막에 나오는 방식이다. 즉 가장 먼저 들어간 r4인 4가 가장 마지막에 다시 r4로 load되어 4가 출력되고 있는 것이다.
r13에는 N의 주소값이 들어가 있다.
sp에는 초기에 설정했던 주소값인 0x20000에서 factorial을 모두 수행 한 후 다시 0x20000으로 되돌려 놓았기 때문에 해당 값이 출력 되고 있다.
ra에는 r4와 같은 이유로 가장 처음 저장되었던 값인 END에 해당하는 주소값 0x1C이 출력되고 있다.