마이크로프로세서응용실험 LAB05 결과보고서

Program flow control

20181536 엄석훈

1. 목적

● 프로그램의 흐름을 변경/제어하는 명령어의 동작에 대해 이해한다.

● 일반적인 분기명령어와 branch with link 명령어의 차이점을 이해한다. 이 과정에서 link register의 역할에 대해 구체적으로 이해한다.

● 조건분기와 무조건분기를 구분한다. 분기명령어를 이용하여 반복 수행을 위한 loop을 구성하는 방법에 대해 생각해 본다.

● PC에 대하여 상대적인 주소로 분기하는 경우와 절대주소로 분기하는 방법에 대해 알아본다.

● Conditional execution을 위해 명령어에 추가되는 suffix들의 유형을 살펴보고 이러한 방식의 도입이 가져다 주는 장점들에 대해 파악해 본다.

● Table을 이용한 프로그램 흐름 변경기법에 대해 알아본다.

● Branch penalty와 이를 완화시킬 수 있는 방법 중 하나인 loop unrolling에 대해 알아본다.

2. 이론

1) Branches and loops in a program

프로그램을 수행하는 중 순차적으로 명령어를 수행하는 것이 아니라 subroutine을 들어가거나 조건에 따라 명령어를 수행하는 경우 프로그램에서 branch가 필요하다. 이때 사용하는 명령어는 무조건분기 명령어와 조건분기 명령어로 나눌 수 있다. 무조건분기 명령어는 B label과 같은 명령어로 label이 있는 주소로 PC를 바꿔 branch하게 된다. 그리고 조건분기 명령어 같은 경우는 B에 조건에 해당하는 suffix를 추가해서 사용하게 된다. 예를 들어 BNE, BEQ와 suffix를 붙여서 다음과 같이 사용하며 flag를 기준으로 해당 branch 명령어를 수행할지 말지 결정하게 된다. 다음으로 loop는 프로그램에서 매우 중요하게 사용되는 기법인데 동일한 동작을 여러 번 반복할 때 사용하

게 된다. 이때 loop를 종료하기 위해서는 종료조건을 정해두고 만족하면 종료하거나 반복횟수를 지정해서 원하는 만큼 loop를 돌고나서 loop를 종료하게 된다. 무조건분기, 조건분기, loop의 동작에 대한 그림은 아래 [그림 1]에서 확인 가능하다.

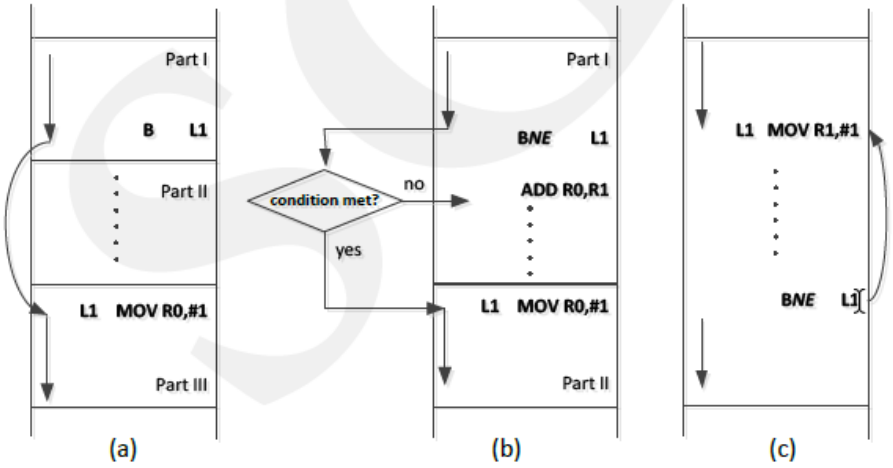


그림 1 (a) 무조건분기, (b) 조건분기, (c) loop 동작에 대한 그림

우리가 사용하는 Cortex-M3에서 branch 명령어는 아래 [그림 2]과 같은 format으로 이루어진다. 이때 PC에서 imm값 * 2만큼 더해진 곳으로 PC를 옮겨 명령어를 fetch한다. 따라서 16bit Thumb 명령어를 사용하는 경우는 offset을 8-bit로 표현 가능해야 한다. imm값에 *2를 하는 이유는 명령어의 위치는 항상 2의 배수이기 때문에 *2를 내포하고 있다고 가정하고 imm값을 지정하면 2배 큰 범위를 접근할 수 있다는 장점이 있다. 하지만 만약 32-bit brach 명령어의 offset범위보다 멀리있는 명령어로 branch하기 위해서는 LDR또는 MOV명령어를 사용해서 직접 PC의 값을 바꿔주어야 한다. 또는 BX 명령어를 사용해서 branch할 수 있다.

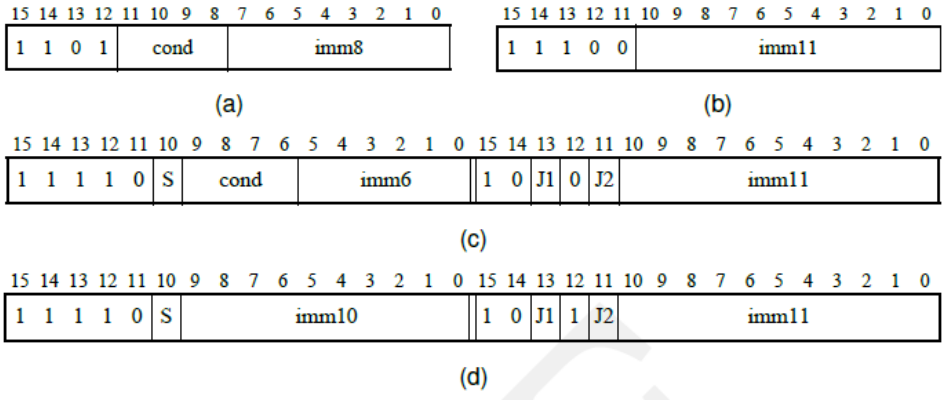


그림 2 Branch 명령어 format

2) Branch with link

일반적인 branch명령어와 달리 BL(branch with link)명령어는 원래의 barnch명령어와 더불어 PC가 가리키고 있던 다음 명령어의 주소를 LR(R14)에 저장해준다. 이렇게 함으로써 나중에 branch한 장소에서 명령어를 수행한 뒤 돌아올 명령어의 위치를 기억하고 있다는 장점이 있다. 이를 이용해 나중에 subroutine을 사용하는데 도움이 된다.

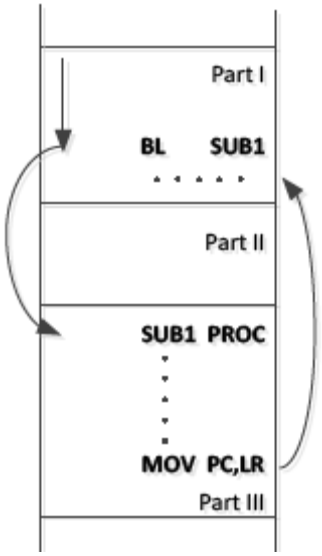


그림 3 BL 명령어의 활용 그림

3) Conditional execution

ARM에서는 분기명령어의 장점을 다른 명령어에서도 활용할 수 있도록 conditional execution이라는 것을 만들어주었다. 이를 통해 기본적인 명령어에 suffix를 붙여서 특정 조건을 만족하는 경우에만 해당 명령어를 수행하도록 해주었다. 예를 들어 ADDEQ라는 명령어는 명령어 수행 전 Z flag를 확인해 1이라면 ADD명령어를 수행하고 그렇지 않다면 명령어를 수행하지 않는다. 이를 통해서 원래라면 branch명령어를 활용해야 하는 경우에도 conditional execution을 활용해 명령어 차원에서 조건에 따라 명령어를 수행되도록 할 수 있어 branch penalty가 줄어들고 코드를 좀 더 compact하게 짤 수 있다. 따라서 코드의 메모리 공간과 수행 시간차원에서 모두 절약할 수 있게 해준다.

4) Jump tables

Jump table은 데이터를 저장하는 lookup table과 유사하게 jump할 주소들을 저장하고 있는 테이블이다. 따라서 jump table에 접근하는 명령어인 TBB와 TBH를 사용해서 base address에서 offset 만큼 이동해서 jump table에서 원하는 주소를 찾아 jump할 수 있게 도와준다. TBB명령어는 byte 크기의 offset을 지원하는데 offset에 2배를 곱해서 주소를 접근하기 때문에 2 * 255 + 4인 514의 범위를 접근할 수 있고 TBH명령어는 halfword 크기의 offset을 지원함으로 2 * 65535 +4를 해서 base주소에서 131074의 범위까지 접근할 수 있다. 아래 [그림 4]에서 TBB 명령어의 사용 예시를 확인할 수 있다.

```
    ADR.W  R0, BranchTable_Byte
    TBB    [R0, R1]          ; R1 is the index, R0 is the base address of the
                             ; branch table
Case1
; an instruction sequence follows
Case2
; an instruction sequence follows
Case3
; an instruction sequence follows
BranchTable_Byte
    DCB    0                 ; Case1 offset calculation
    DCB    ((Case2-Case1)/2)  ; Case2 offset calculation
    DCB    ((Case3-Case1)/2)  ; Case3 offset calculation
```

그림 4 TBB 명령어의 사용 예시

3. 실험 과정

1) 실험 1

STEP 1: Program 5.2를 이용하여 프로젝트를 생성한다.

```
1        area lab5_1,code
2        entry
3   __main proc
4        export __main [weak]
5   start mov r1,#8
6        mov r2,#32
7   gcd  cmp r1,r2
8        beq done
9        blt less
10       sub r1,r1,r2
11       b gcd
12  less sub r2,r2,r1
13       b gcd
14  ;    space 0x800
15  done
16       b .
17  ;    ldr r3,=start
18  ;    mov r15,r3
19       align
20       endp
21       end
```

그림 5 직접 작성한 lab5_1.s 코드

STEP 2: 프로그램의 단계별 수행 및 레지스터 r1과 r2 변경과정의 추적을 통해 GCD(greatest common divisor;최대공약수)를 구하는 과정임을 확인한다.

R1 0x00000008
R2 0x00000020

그림 6 레지스터의 변경과정 1

R1 0x00000008
R2 0x00000018

그림 7 레지스터 변경과정 2

R1 0x00000008
R2 0x00000010

그림 8 레지스터 변경 과정 3

R1 0x00000008
R2 0x00000008

그림 9 레지스터 변경 과정 4

[그림 6]부터 [그림 9]까지가 프로그램의 단계별 수행에 따른 r1, r2 레지스터 값의 변경과정이다. 먼저 최대공약수를 구하려는 두 숫자를 로드한다. 프로그램이 수행되면서 r1와 r2를 비교해서 같으면 해당 숫자가 최대공약수이다. 그리고 두 숫자가 다르면 큰 숫자에서 작은 숫자를 빼고 다시 r1과 r2를 비교하는 과정으로 돌아간다. 이 과정을 반복하면 마지막에는 최대 공약수가 남게 되고 이를 [그림 9]에서 확인할 수 있다. 즉 0x08과 0x20의 최대공약수는 0x08임을 알 수 있다.

STEP 3: *.lst 파일의 확인을 통해 Lines 8, 9, 11, 13의 assemble 결과와 그림 5.3의 명령어 format들과 비교하면서 순반향 분기와 역방향 분기의 동작 원리를 이해한다. 특이 immx(여기서 x는 8 또는 11 등과 같이 immediate field의 길이를 의미)가 분기의 방향에 따라 어떻게 번역되었는지 확인한다.

```
 7 00000008 4291     gcd     cmp             r1,r2
 8 0000000A D006             beq             done
 9 0000000C DB02             blt             less
10 0000000E EBA1 0102        sub             r1,r1,r2
11 00000012 E7F9             b               gcd
12 00000014 EBA2 0201
                     less    sub             r2,r2,r1
13 00000018 E7F6             b               gcd
14 0000001A          ;      space 0x800
15 0000001A          done
```

그림 10 lst파일 assemble 결과

branch명령어의 format은 이론 부분의 [그림 2]에서 확인할 수 있다. line 8의 명령어를 2진수로 나타내면 1101 0000 0000 0110인데 이를 [그림 2]의 (a)를 사용해서 해석해보면 imm8은 0000 0110으로 0x6이다. 따라서 branch할 목적지 주소를 계산해보면 line 8를 실행할 때 PC는 0x0E이고 여기에 0x6 * 2만큼을 더해주면 0x1A로 branch하게 되고 이는 line 15에서 볼 수 있듯이 label done의 주소이다. 다음으로 line 9의 명령어를 2진수로 나타내면 1101 1011 0000 0010이고 imm8은 0000 0010이고 0x2이다. 똑같이 계산해보면 line 9를 실행할 때 PC는 0x10이고 여기에 0x2 * 2만큼 더해주면 0x14로 branch하게 되고 이는 line 12에서 볼 수 있듯이 label less의 주소이다.

다음으로 line 11의 명령어를 2진수로 나타내면 1110 0111 1111 1001이고 이를 [그림 2]의 (b)를 이용해서 해석해 보면 imm11의 값은 111 1111 1001이고 0x7F9이고 이를 2's complement로 보면 10진수 -7이다. Line 11을 실행할 때 PC는 0x16이고 여기에 0x7F9 * 2를 하고 더한 수를 11-bit로 표시하면 000 0000 1000이고 0x08이고 이는 line 7의 gcd label이 가리키는 주소이다. 마지막으로 동일하게 line 13을 2진수로 나타내면 1110 0111 1111 0110이고 imm11의 값은 111 1111 0110이고 0x7F6이고 line 13을 실행할 때 PC는 0x1C이고 여기에 0x7F6 * 2를 하고 더한 수를 11-bit로

표시하면 동일하게 0x08이 되고 line 7의 gcd label이 가리키는 주소임을 확인할 수 있다.

이를 바탕으로 branch명령어의 동작 원리를 생각해보면 branch format에 저장된 imm숫자와 2를 곱한 뒤 명령어를 수행할 때의 PC값을 더해준 뒤 imm의 범위대로 잘르면 이동하려는 주소가 나오게 된다. 즉 순방향 분기와 역방향 분기 모두 동일하게 imm값을 이용해서 branch하려는 주소를 계산한다. 다만 한가지 확인할 수 있는 점이 있다면 순방향 분기는 imm값은 양수이기 때문에 MSB가 0이고 역방향 분기라면 imm값이 음수가 되어야 하기 때문에 MSB가 1이 되는 것을 확인할 수 있다.

STEP 4: Line 14의 주석처리를 제거한 후 프로그램을 assemble한다. 이 때 발생하는 오류 또는 경고가 있다면 그 의미를 확인한다. 이 상태에서 line 8의 beq를 beq.w로 변경한 후 다시 assemble한다. *.lst 파일을 통해 해당 명령어가 어떻게 번역되었는지 그림 5.3의 명령어 format들을 통해 확인하고, 그 의미와 앞서 설명한 오류 또는 경고가 사라진 이유를 생각해보자.

```
Build started: Project: lab5_1
*** Using Compiler 'V6.19', folder: 'C:\Keil_v5\ARM\ARMCLANG\Bin'
Build target 'Target 1'
assembling lab5_1.s...
lab5_1.s(8): error: A1762E: Branch offset 0x0000080C out of range of 16-bit Thumb branch, but offset encodable in 32-bit Thumb branch
".\Objects\lab5_1.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:00
```

그림 11 Line 14 주석 제거 후 오류

Line 14의 주석처리를 제거하면 done label과 line 8 명령어 사이에 0x800 바이트의 공간이 할당된다. 따라서 line 8에서의 PC값과 done label 사이의 거리가 0x80C임으로 offset은 406 즉 2진법으로 100 0000 0110가 되어서 offset을 표현하기 위해서는 11bit가 필요하다. 하지만 16-bit Thumb branch명령어는 8-bit의 imm을 사용할 수 있음으로 branch가 목적지에 접근할 수 없다고 오류를 낸 것이다.

다음으로 beq.w로 수정 후 assemble한 경우의 lst 파일은 아래 [그림 12]에서 확인 가능하다. beq.w는 32-bit 명령어로 0xF0008407로 번역되었다. 이를 [그림 2]의 (c)의 format을 이용해서 해석하면 0xF0008407은 2진수로 1111 0000 0000 0000 1000 0100 0000 0111임으로 imm11은 100 0000 0111로 0x407이고 0x407 * 2 + line 8에서의 PC인 0x0A + 4를 더해주면 0x81C로 branch하게 된다. 따라서 32-bit 명령어를 사용해서 offset을 나타내는 imm값의 범위가 커서 더 먼 거리를 branch할 수 있었다. 따라서 접근할 수 없었던 영역을 접근할 수 있게 되었고 에러가 사라졌다.

```
 5 00000000 F04F 0108
                    start   mov             r1,#8
 6 00000004 F04F 0220       mov             r2,#32
 7 00000008 4291     gcd    cmp             r1,r2
 8 0000000A F000 8407       beq.w           done
 9 0000000E DB02            blt             less
10 00000010 EBA1 0102       sub             r1,r1,r2
11 00000014 E7F8            b               gcd
12 00000016 EBA2 0201
                    less    sub             r2,r2,r1
13 0000001A E7F5            b               gcd
```

```
           00 00 00
           00 00            space           0x800
15 0000081C          done
```

그림 12 beq.w 사용 후 lst 파일

STEP 5: Line 16은 주석처리 하고, lines 17-18의 주석처리를 제거한 후 프로그램을 assemble한다. 이 상태에서 프로그램을 단계별로 수행하면서 line 18 명령어 동작의 의미를 해석한다. Line 18을 bx r3으로 교체한 후 이 과정을 반복한다. 이 분기 명령어와 STEP 3에서 확인한 분기 명령어의 동작상의 차이점을 A-6페이지를 통해 확인한다.

주석처리를 변경한 후 프로그램을 assemble하면 아래 [그림 13]과 같다. Line 18은 MOV pc,r3로 해석되었는데 r3에는 line 17에 의해서 start label의 주소인 0x080002C8을 가지고 있었고 이를 MOV 명령어를 통해서 pc에 0x080002C8을 로드하였다. Line 18 수행 후 pc는 0x080002C8이 되었음으로 다음으로 line 18 명령어 다음에는 line 5가 수행되었다. 이 명령어를 통해서 pc를 우리가 원하는 값으로 지정되도록 해 프로그램의 흐름을 바꿀 수 있었다.

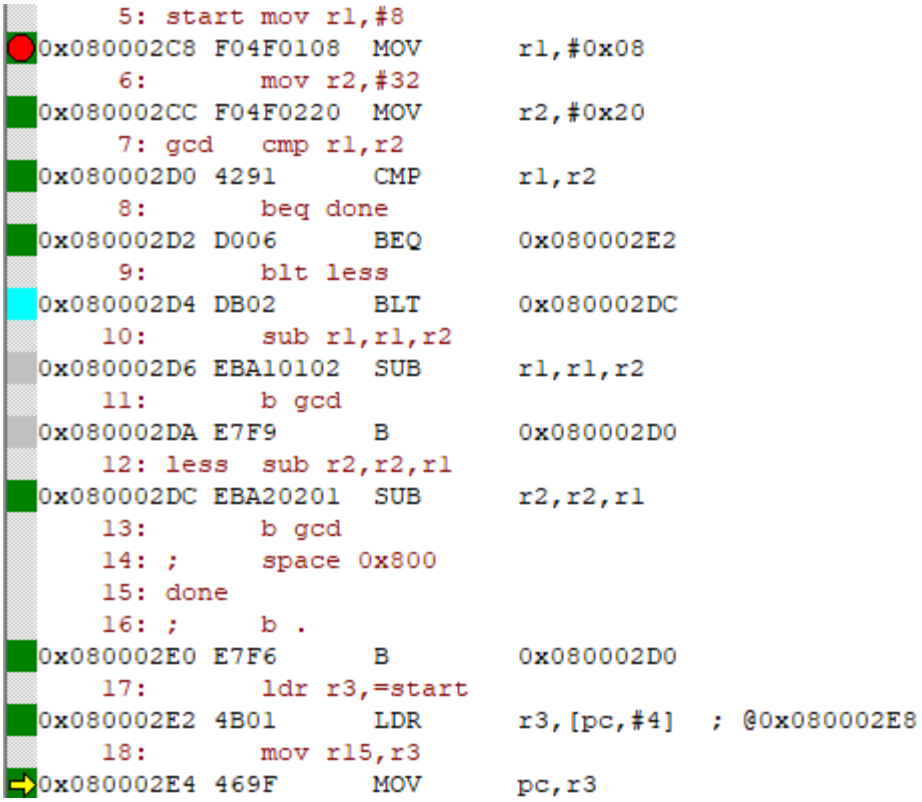


그림 13 주석 변경 후 코드

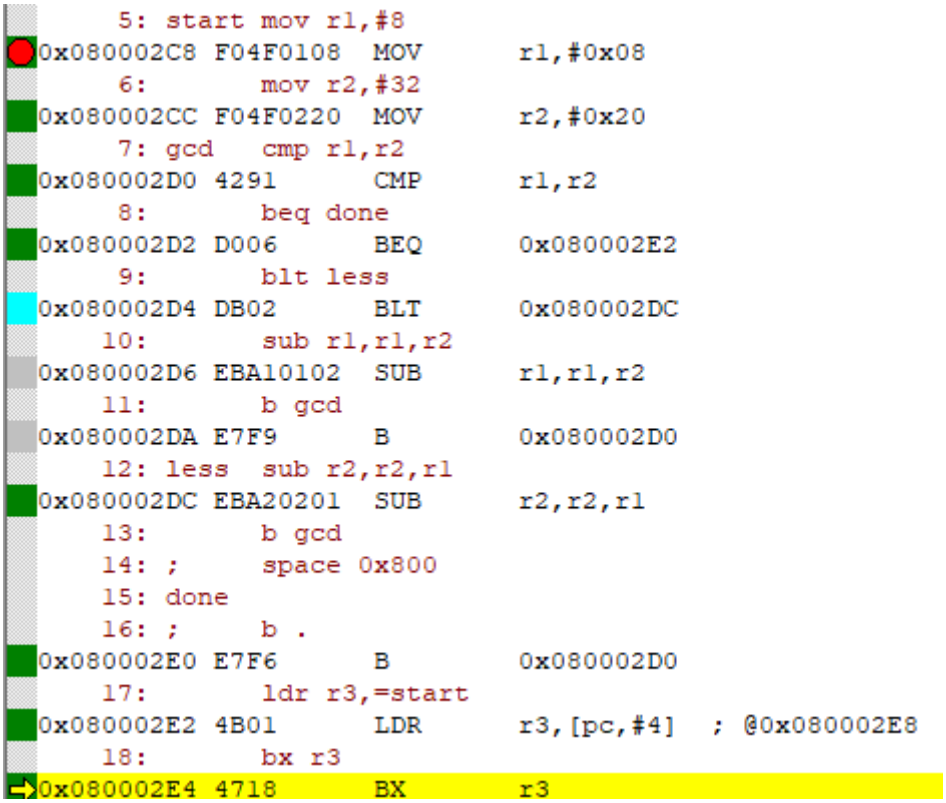


그림 14 Line 18 수정 후 코드

다음으로 line 18을 bx r3로 변경한 후 프로그램을 수행 한 결과는 mov pc,r3 명령어와 동일한 동

작을 수행하였다. STEP 3에서 사용한 B명령어와의 차이점은 B명령어는 뒤에 label이 들어와 pc-relative 표현이 필요한데 반해서 BX명령어는 레지스터에 저장되어 있는 값으로 branch하게 된다. 그리고 이때 BX 명령어를 사용할 때 레지스터에 저장된 값의 LSB는 반드시 1이어야 한다. 어차피 주소는 항상 짝수이기 때문에 LSB의 값은 무의미한데 이를 fault exception 확인을 위해서 LSB는 항상 1로 사용한다.

STEP 6: 이전 과정들을 수행하면서 임의로 변경했던 주석처리 부분들을 Program 5.2의 원래의 상태로 복원한다.

Lines 7-13을 103페이지의 gcd 코드 중 우측에 주어진 conditional execution을 사용한 것으로 변경한 후 그 동작을 확인한다.

.lst 파일을 통해 교체한 부분이 차지하는 code size와 원 프로그램의 line 7-13이 차지하는 code size를 비교하여 conditional execution이 갖는 장점을 확인한다.

또한 각 명령어의 수행시간 확인을 통해 GCD를 구하는데 소요된 시간을 구해 conditional execution 명령어의 사용에 따른 효과를 수행시간 측면에서 비교해보자.

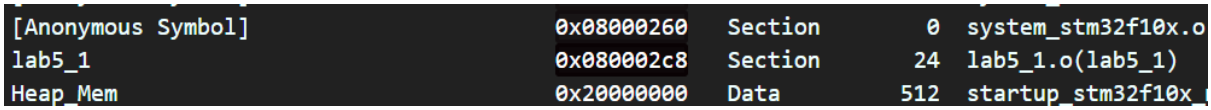
먼저 원래 코드의 code size를 확인해보기 위해서는 lst파일보다는 map파일이 더 확실히 비교할 수 있어서 map파일의 image table을 이용해서 비교를 진행하였다.

```
[Anonymous Symbol]                0x08000260   Section        0  system_stm32f10x.
lab5_1                            0x080002c8   Section       28  lab5_1.o(lab5_1)
Heap_Mem                          0x20000000   Data         512  startup_stm32f10x
```

그림 15 수정 전 코드의 코드 사이즈

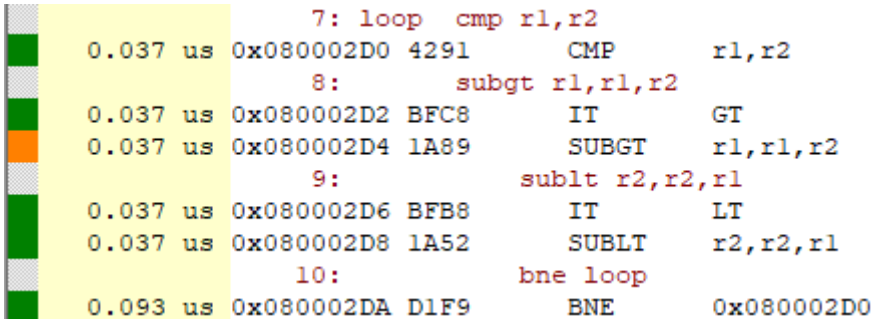
```
                        7: gcd   cmp r1,r2
    0.037 us 0x080002D0 4291      CMP      r1,r2
                        8:       beq done
    0.056 us 0x080002D2 D006      BEQ      0x080002E2
                        9:       blt less
    0.083 us 0x080002D4 DB02      BLT      0x080002DC
                       10:       sub r1,r1,r2
             0x080002D6 EBA10102  SUB      r1,r1,r2
                       11:       b gcd
             0x080002DA E7F9      B        0x080002D0
                       12: less  sub r2,r2,r1
    0.028 us 0x080002DC EBA20201  SUB      r2,r2,r1
                       13:       b gcd
                       14: ;     space 0x800
                       15: done
    0.083 us 0x080002E0 E7F6      B        0x080002D0
```

그림 16 수정 전 코드의 수행시간



[Anonymous Symbol]	0x08000260	Section	0	system_stm32f10x.o
lab5_1	0x080002c8	Section	24	lab5_1.o(lab5_1)
Heap_Mem	0x20000000	Data	512	startup_stm32f10x_

그림 17 수정 후 코드의 코드 사이즈



```
                7: loop  cmp r1,r2
    0.037 us 0x080002D0 4291      CMP       r1,r2
                8:       subgt r1,r1,r2
    0.037 us 0x080002D2 BFC8      IT        GT
    0.037 us 0x080002D4 1A89      SUBGT     r1,r1,r2
                9:           sublt r2,r2,r1
    0.037 us 0x080002D6 BFB8      IT        LT
    0.037 us 0x080002D8 1A52      SUBLT     r2,r2,r1
               10:           bne loop
    0.093 us 0x080002DA D1F9      BNE       0x080002D0
```

그림 18 수정 후 코드의 수행시간

먼저 [그림 18]을 통해 수정 후 코드가 [그림 18]과 같이 assemble된 것을 확인할 수 있다. 우리가 subgt나 sublt와 같이 코드를 작성하였는데 assemble 결과에서는 IT GT, IT LT와 같은 명령어가 추가된 것을 확인할 수 있다. 이 IT명령어는 if-then 명령어로 IT뒤에 오는 condition에 따라서 블록단위로 명령어를 묶어 동일한 condition을 가진 conditional 명령어라고 알려주는 명령어이다. 동작을 살펴보면 cmp명령어를 통해 flag를 설정하고 각 subgt와 sublt명령어를 통해 조건에 따라 뺄셈을 진행하다가 r1과 r2가 같아지면 loop를 탈출해 기존의 코드와 동일한 gcd를 구하는 동작을 수행한다.

[그림 15]와 [그림 17]의 코드 수정 전과 수정 후의 코드 사이즈를 비교해보면 28byte에서 24byte로 약간 감소한 것을 확인할 수 있다. 이 코드 사이즈에는 다른 명령어를 포함한 모든 lab5_1에서 짠 코드의 영역을 포함하고 있기 때문에 gcd코드 부분만의 정확한 비교는 힘들지만 확실히 코드의 사이즈가 감소한 것을 확인할 수 있다.

또한 [그림 16]과 [그림 18]에서 각각 명령어 별로 소요된 시간을 모두 더해보면 [그림 16]에서는 0.287us이고 [그림 18]에서는 0.278us로 수정 후 코드의 수행시간이 더 짧은 것을 확인할 수 있다. 이 차이는 코드가 커지고 더 많이 사용될수록 차이가 커지게 될 것임으로 ARM에서 지원하는 conditional execution이 코드 사이즈와 수행시간 측면에서 모두 장점을 가지는 것을 확인할 수 있다.

2) 실험 2

STEP 7: Program 5.3을 이용하여 프로젝트를 생성한다.

```
1        area lab5_2,code
2        entry
3   __main proc
4        export __main [weak]
5        mov r1,#2
6        mov r2,#8
7        bl sub1
8        mov r1,#5
9        mov r2,#3
10       bl sub1
11       b .
12       endp
13  sub1  proc
14       add r3,r1,r2, LSL #2 ; r3=r1+r2*4
15       bx lr
16       endp
17       end
```

그림 19 직접 작성한 lab5_2.s 코드

STEP 8: Line 13-16에 간단한 연산을 수행하는 procedure(일반적으로 subroutine 또는 function 이라고 불리는)가 정의되어 있음을 확인한다.

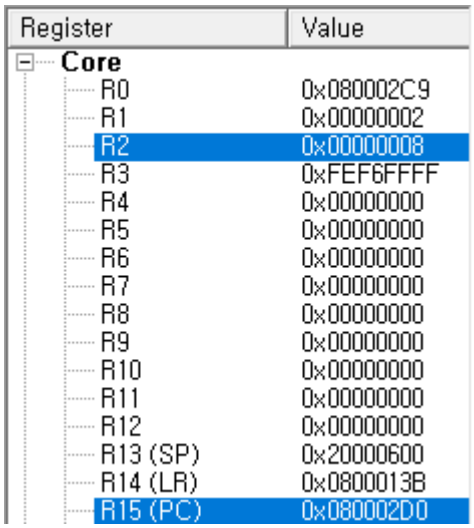
위의 [그림 19]에서 line 13을 보면 sub1 proc이라고 해서 line 3의 __main proc처럼 proc은 어셈블러에게 여기서부터 새로운 함수의 시작이라고 알려주는 directive이다. 그리고 앞에 sub1이나 __main은 해당 함수의 이름인 label이다. 즉 line 13이 sub1이라는 함수의 시작점이라고 알려주는 것이다. 그리고 밑에 sub1에서의 동작인 add r3,r1,r2, LSL #2와 bx lr이 line 14, line 15에 각각 적혀 있다. 이 함수의 동작은 r3에 r1+r2*4를 저장하고 lr에 저장된 위치로 branch하는 함수이다. 그리고 line 16에서 함수의 끝이라고 알려주는 endp라는 directive가 사용된 것을 확인할 수 있다.

STEP 9: Line 5-6 명령어를 수행한다.

Line 7 명령어의 수행 전후에 register window를 통해 r14의 내용이 어떻게 바뀌는지 확인하고 A-6페이지를 통해 이 레지스터에 저장된 내용의 의미를 해석한다. 또한 PC(r15)의 내용이 어떻게 변경되었는지 확인하고 연속적인 단계별 수행을 통해 프로그램의 흐름을 추적한다.

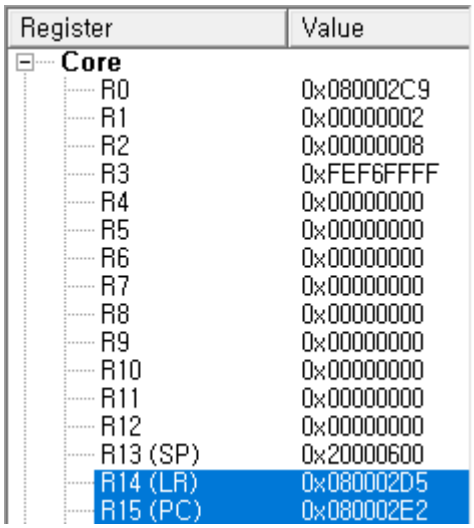
Line 15 명령어의 의미를 A-6페이지를 통해 확인한다. 여기서 lr(link register)은 r14의 별칭이다. 이 명령어의 수행을 통해 PC는 어떻게 바뀌었는지 확인한다. 이 명령어를 mov pc,lr로 교체한 후

이 단계를 반복한다.



Register	Value
Core	
R0	0x080002C9
R1	0x00000002
R2	0x00000008
R3	0xFEF6FFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000600
R14 (LR)	0x0800013B
R15 (PC)	0x080002D0

그림 20 Line 7 수행 전 레지스터 값



Register	Value
Core	
R0	0x080002C9
R1	0x00000002
R2	0x00000008
R3	0xFEF6FFFF
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000600
R14 (LR)	0x080002D5
R15 (PC)	0x080002E2

그림 21 Line 7 수행 후 레지스터 값

Line 5, line 6 명령어를 수행해 r1과 r2에 2와 8을 로드한다. 그리고 line 7 bl sub1을 수행하면 sub1의 위치로 branch하며 LR에 다시 돌아가야 할 PC값인 0x080002D5를 저장하고 PC를 sub1의 위치로 옮긴다. BL 명령어는 다시 돌아와야 할 PC값을 LR에 저장하고 일반적은 branch와 동일하게 branch하는 명령어이다. 이때 LR에 저장된 값의 LSB가 1인데 명령어는 항상 2의 배수이기 때문에 잘못 들어갔다고 생각될 수 있다. 하지만 이는 어차피 PC를 읽을 때 LSB는 무시하고 읽기 때문에 ARM mode와 / Thumb mode 사이의 변경이 일어난다는 뜻으로 LSB에 1을 붙여 둔 것이다. 만약에 강제로 LSB를 0으로 바꿔버리면 돌아올 때 fault가 발생한다.

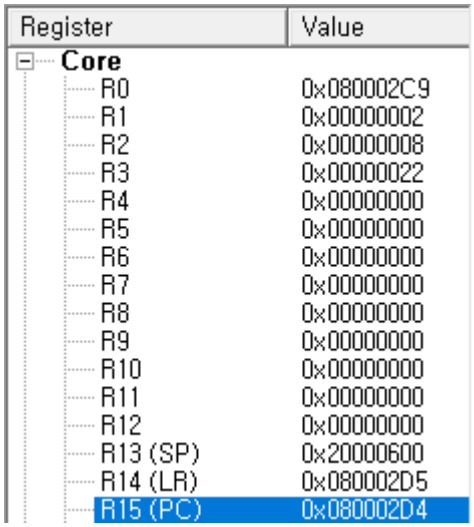


그림 22 Line 15 수행 후 레지스터 값

그리고 line 14가 수행되고 line 15는 bx lr이 수행되었다. BX 명령어는 레지스터가 가지고 있는 주소로 다시 PC를 바꿔 branch하는 명령어이다. 여기서 만약 레지스터에 저장된 값의 LSB가 0이라면 오류가 발생한다. 어차피 PC에서 LSB는 필요 없기 때문에 항상 1로 하고 실제로 가져올 때는 무시하고 가져온다. 따라서 [그림 22]에서 확인할 수 있는 것처럼 PC는 LR이 가지고 있는 값으로 branch하는데 이때 0x080002D5가 아닌 0x080002D4가 된 것을 확인할 수 있다. 마지막으로 line 15를 mov lr,pc로 코드를 수정하고 동일하게 수행한 결과 아래 [그림 23]처럼 완벽히 동일하게 동작하였다. 물론 동일하게 동작하긴 하지만 따라서 꼭 필요한 경우가 아니라면 가독성을 위해서 bx 명령어를 사용하는 것이 좋아 보인다.

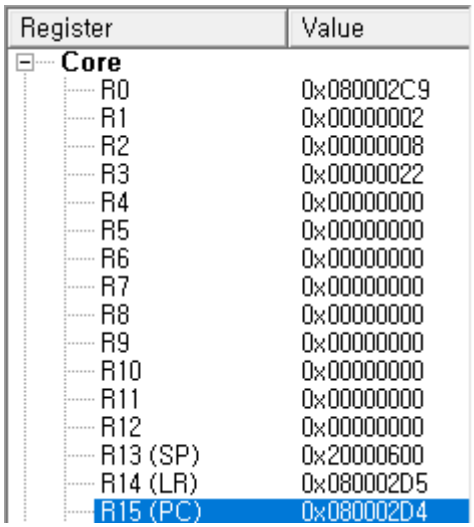
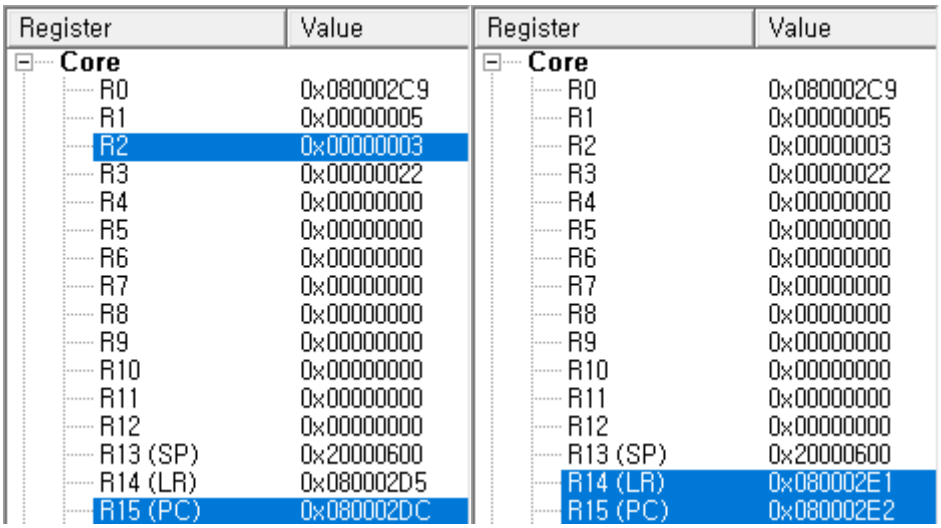


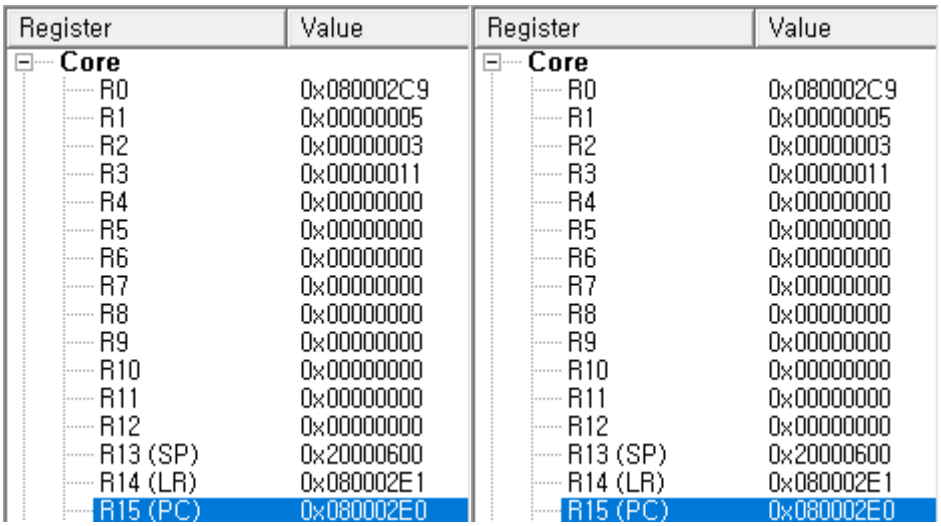
그림 23 Line 15 수정 후 수행 후 레지스터 값

STEP 10: Lines 8-10 명령어들에 대해 위 과정을 반복한다.



Register	Value	Register	Value
Core		Core	
R0	0x080002C9	R0	0x080002C9
R1	0x00000005	R1	0x00000005
R2	0x00000003	R2	0x00000003
R3	0x00000022	R3	0x00000022
R4	0x00000000	R4	0x00000000
R5	0x00000000	R5	0x00000000
R6	0x00000000	R6	0x00000000
R7	0x00000000	R7	0x00000000
R8	0x00000000	R8	0x00000000
R9	0x00000000	R9	0x00000000
R10	0x00000000	R10	0x00000000
R11	0x00000000	R11	0x00000000
R12	0x00000000	R12	0x00000000
R13 (SP)	0x20000600	R13 (SP)	0x20000600
R14 (LR)	0x080002D5	R14 (LR)	0x080002E1
R15 (PC)	0x080002DC	R15 (PC)	0x080002E2

그림 24 Line 10 수행 (왼쪽)전, (오른쪽)후 레지스터 값



Register	Value	Register	Value
Core		Core	
R0	0x080002C9	R0	0x080002C9
R1	0x00000005	R1	0x00000005
R2	0x00000003	R2	0x00000003
R3	0x00000011	R3	0x00000011
R4	0x00000000	R4	0x00000000
R5	0x00000000	R5	0x00000000
R6	0x00000000	R6	0x00000000
R7	0x00000000	R7	0x00000000
R8	0x00000000	R8	0x00000000
R9	0x00000000	R9	0x00000000
R10	0x00000000	R10	0x00000000
R11	0x00000000	R11	0x00000000
R12	0x00000000	R12	0x00000000
R13 (SP)	0x20000600	R13 (SP)	0x20000600
R14 (LR)	0x080002E1	R14 (LR)	0x080002E1
R15 (PC)	0x080002E0	R15 (PC)	0x080002E0

그림 25 Line 15 수정 (왼쪽)전, (오른쪽)후 수행 후 레지스터 값

위의 STEP 11과 동일하게 동작하는 것을 알 수 있다. [그림 24]에서는 line 10에서 bl 명령어 실행 전 후 레지스터 값을 보면 수행 후에는 LR에 돌아가야 할 PC값이 LSB가 1이 되도록 저장되고 PC는 sub1의 위치로 branch한 것을 알 수 있다. 그리고 [그림 25]에서는 역시나 bx lr 명령어와 mov pc,lr 명령어가 동일하게 동작하는 것을 확인할 수 있다.

STEP 11: 이 프로그램을 통해 앞서 살펴본 b 명령어와 bl 명령어의 결정적인 차이는 무엇인가? 그림 5.2(d)를 그림 5.2의 나머지 경우들과 비교하여 설명해 보자.

이 프로그램을 통해 확인할 수 있는 b 명령어와 bl 명령어의 결정적인 차이는 bl 명령어는 LR에

돌아올 PC값을 저장하고 branch하는 반면 b 명령어는 그냥 바로 branch한다. 교재 그림 5.2에서 (d)만 돌아올 주소를 직접 저장하고 있기 때문에 branch를 해서 프로그램에서 다른 부분으로 점프했다가 subroutine을 수행하고 다시 원래의 흐름대로 돌아올 수 있다. 즉 나머지 경우는 같이 사용할 다른 branch가 없는데 비해서 bl은 bx 또는 mov와 함께 활용되어 좀 더 안전하고 편하게 프로그램 사이를 이동할 수 있다.

3) 실험 3

STEP 12: Program 5.4를 이용하여 프로젝트를 생성한다.

```
1         area lab5_3,code
2         entry
3   __main proc
4         export __main [weak]
5         mov r3,#2 ; comp type - 0:add, 1:sub, 2:mul, 3:div
6         mov r1,#8
7         mov r2,#3
8         adr r4,jtable
9         tbb [r4,r3]
10  doadd add r0,r1,r2
11        b svr0
12  dosub sub r0,r1,r2
13        b svr0
14  domul mul r0,r1,r2
15        usat r0,#8,r0
16        b svr0
17  dodiv udiv r0,r1,r2
18  svr0  ldr r5,=result
19        str r0,[r5]
20        b .
21        align
22  jtable
23        dcb 0
24        dcb ((dosub - doadd)/2)
25        dcb ((domul - doadd)/2)
26        dcb ((dodiv - doadd)/2)
27        endp
28        area lab5data,data
29  result dcd 0
30        end
```

그림 26 직접 작성한 lab5_3.s 코드

STEP 13: Memory window를 통해 lines 22-26에 해당하는 위치에 어떠한 내용이 저장되었는지를 확인하고 그 의미를 파악한다.

```
0x080002F8: FE E7 00 00 00 03 06 0B
```

그림 27 Lines 22-26에 해당하는 memory

Lines 22-26은 jump table에 대한 부분이다. TBB명령어를 통해서 jump table에 접근해서 우리가 원하는 곳으로 branch하게 된다. [그림 27]과 [그림 26]의 코드를 보면 jump table에는 00 / 03 / 06 / 0B가 각각 저장되어 있는데 이는 branch할 때 필요한 offset이다. doadd는 TBB 바로 다음 주소에 있음으로 offset이 0으로 되어있고 그 다음 주소인 dosub는 PC가 6만큼 증가해야 하는데 어차피 명령어는 2의 배수로 되어있음으로 좀 더 큰 범위를 접근하기 위해서 offset을 0x3으로 2로 나눈 값을 저장하였다. 마찬가지로 domul은 offset을 0x6으로 저장하고 dodiv는 offset을 0xB로 저장하였다. 즉 메모리에는 branch할 때 필요한 offset의 절반에 해당하는 값이 저장되어 있다고 생각할 수 있다.

STEP 14: Line 9의 tbb 명령어의 동작을 A-34페이지를 통해 확인한다. 프로그램을 수행하면서 이 명령어의 동작을 확인한다. 이 과정에서 r4와 r3의 역할을 파악한다. 이를 위해 r3에 저장되는 내용을 변경하며 수행해본다.

TBB 명령어는 table branch byte 명령어로 TBB [Rn, Rm]의 형태로 명령어를 사용하며 Rn에는 table의 주소를 가지고 있고 Rm은 index값을 가지고 있다. 즉 Rn + Rm의 주소로 가서 해당 테이블에 저장되어 있는 offset을 확인한 뒤 현재 PC에서 offset * 2한 주소로 branch를 한다.

TBB를 수행할 때 r4에는 [그림 26]에서 확인할 수 있듯이 jtable의 시작 주소가 저장되어 있고, r3에는 index인 2가 저장되어 있다. 그리고 tbb [r4,r3]의 결과는 [그림 29]를 참고하면 r4 + r3인 0x080002FE를 접근하고 해당 주소에는 [그림 27]에서 확인할 수 있는 것처럼 0x06이 저장되어 있다. 따라서 TBB 명령어를 수행할 때의 PC인 0X080002DA + 0x06 * 2 = 0x080002E6가 되어 [그림 29]에서 확인할 수 있는 것처럼 PC가 0x080002E6가 된 것을 확인할 수 있다. 이 과정을 통해서 r4는 tbb가 참조하려는 table의 시작 주소를 가지고 있고 r3는 table에서 몇 번째 index를 참조할 것인지에 대한 정보를 가지고 있다. 이를 통해 table에 접근해 branch할 주소의 offset을 구한 뒤 PC값에 offset의 2배만큼을 더한 주소로 branch한다. 추가적으로 아래 [그림 30]처럼 r3의 값을 0, 1, 3으로 변경한 뒤 line 9의 수행 후의 레지스터 값을 확인해본 결과 r3가 0인 경우 PC는 그대로 0x080002DA가 되고, r3가 1인 경우 0x080002DA + 0x3 * 2 = 0x080002E0, r3가 3인 경우 0x080002DA + 0xB * 2 = 0x080002F0로 예상한 대로 branch한 것을 확인할 수 있다.

```
     9:        tbb [r4,r3]
0x080002D6 E8D4F003  TBB       [r4,r3]
    10: doadd add r0,r1,r2
0x080002DA EB010002  ADD       r0,r1,r2
    11:        b svr0
0x080002DE E009      B         0x080002F4
    12: dosub sub r0,r1,r2
0x080002E0 EBA10002  SUB       r0,r1,r2
    13:        b svr0
0x080002E4 E006      B         0x080002F4
    14: domul mul r0,r1,r2
0x080002E6 FB01F002  MUL       r0,r1,r2
```

그림 28 TBB 관련 코드 부분

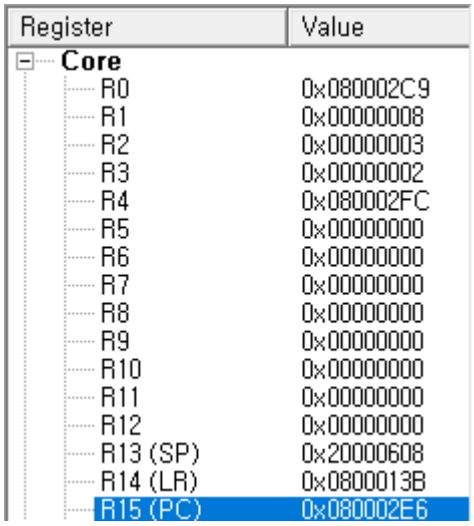


그림 29 line 15 수행 후 레지스터 값

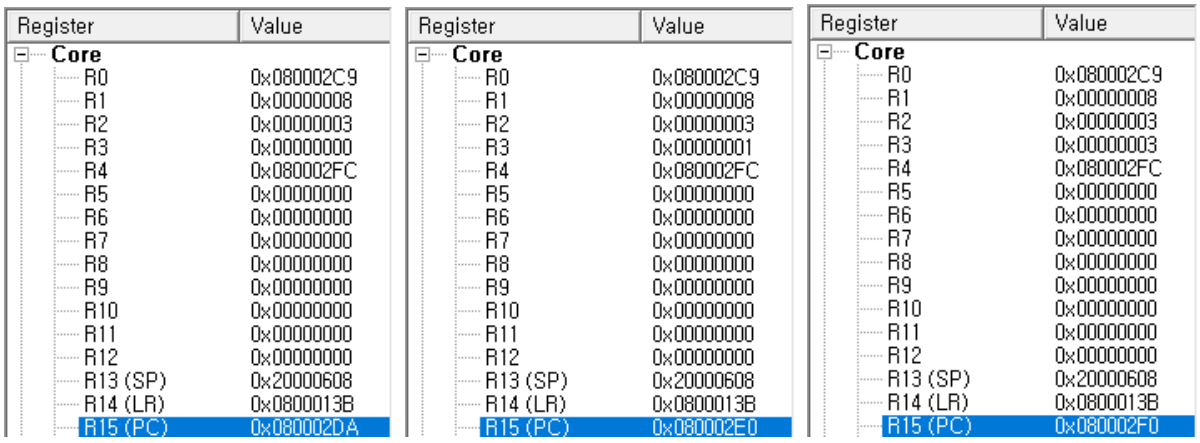


그림 30 r3값을 왼쪽부터 0, 1, 3으로 수정 후 line 9 실행 후 레지스터 값

STEP 15: Line 5에서 r3에 저장되는 내용의 변경이 프로그램 수행에 어떻게 영향을 미치는지 확인한다. Line 19 명령어의 수행결과를 Memory window를 통해 확인한다.

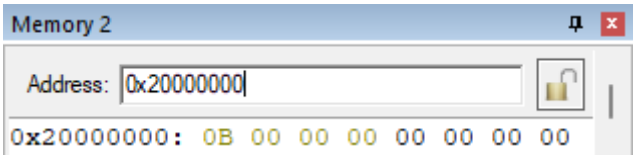


그림 31 r3 = 0, line 19 수행 후 레지스터 값

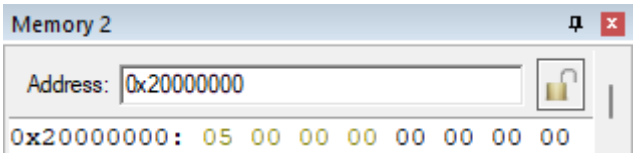


그림 32 r3 = 1, line 19 수행 후 레지스터 값

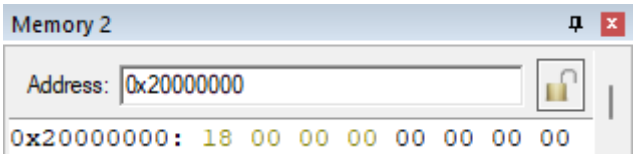


그림 33 r3 = 2, line 19 수행 후 레지스터 값

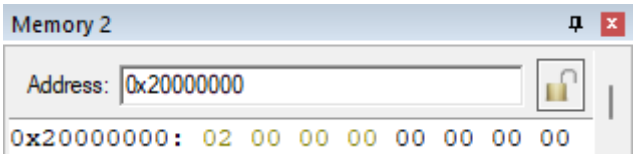


그림 34 r3 = 3, line 19 수행 후 레지스터 값

r3에 저장되는 값을 0부터 3까지 바꿔가면서 프로그램을 실행하여 line 19 수행 후 메모리에 저장된 데이터를 [그림 31] 부터 [그림 34]에 걸쳐서 확인할 수 있다. r3값은 위의 STEP 14에서 확인했듯이 tbb명령어의 index로 사용되어 jump table에서 offset을 가져올 때 영향을 준다. 따라서 r3값을 바꿔서 프로그램에서 수행 될 명령어를 선택할 수 있었고 r3 = 0인 경우는 r1 + r2의 값을 메모리에 저장하고, r3 = 1인 경우는 r1 – r2의 값을 메모리에 저장하고, r3 = 2인 경우는 r1 * r2의 값을 메모리에 저장하고, r3 = 4인 경우는 r1 / r2의 몫을 메모리에 저장하였다. 이를 통해서 상황에 따라서 다른 명령어가 수행할 수 있도록 branch의 기능을 확인할 수 있었다.

4) 실험 4

STEP 16: Program 5.5를 이용하여 프로젝트를 생성한다. 이 프로그램에 의해 수행결과를 메모리 영역에서 변경된 내용을 통해 확인한다.

```
1        area lab5_4,code
2        entry
3   __main proc
4        export __main [weak]
5        mov r3,#0x10
6        ldr r1,=s_data
7        ldr r2,=d_data
8   lp   ldrb r0,[r1],#1
9        strb r0,[r2],#1
10  ;    ldrb r0,[r1],#1
11  ;    strb r0,[r2],#1
12       subs r3,#1
13       bne lp
14  here  b here
15  s_data dcb "Sogang University"
16       endp
17       align
18       area lab5data,data
19  d_data space 16
20       end
```

그림 35 직접 작성한 lab5_4.s 코드

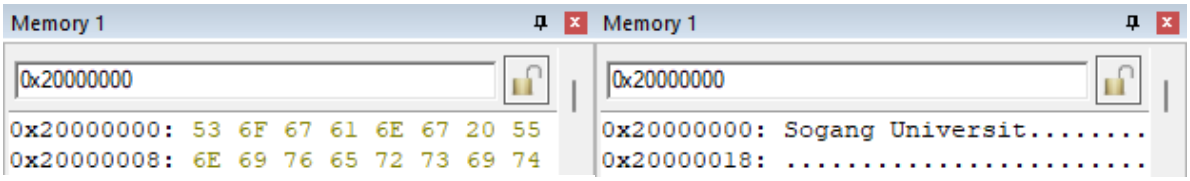


그림 36 프로그램 수행 후 메모리 값

[그림 35]와 같이 프로그램을 작성하고 프로그램을 끝까지 수행한 결과는 [그림 36]과 같다. r3에 0x10을 저장하고 s_data의 내용을 하나씩 복사하며 r3의 값을 하나씩 감소하다가 r3가 0이되면 loop를 탈출한다. 이를 통해 s_data에 저장된 데이터가 16byte만큼 메모리에 복사되었다. 그 결과를 살펴보면 문자열이 ascii 값으로 변환되어 메모리에 저장되었는데 [그림 36]의 왼쪽처럼 숫자로만 보면 무슨 뜻인지 알기 힘들기 때문에 Memory window에서 ascii값으로 변환해서 [그림 36]의 오른쪽과 같이 살펴볼 수 있다.

STEP 17: Lines 8-13 수행하는데 소요된 시간을 측정한다.

[아래 그림 37]과 같이 loop를 모두 수행한 뒤 loop문의 수행시간을 합치면 1.266us가 소요된 것을 확인할 수 있다.

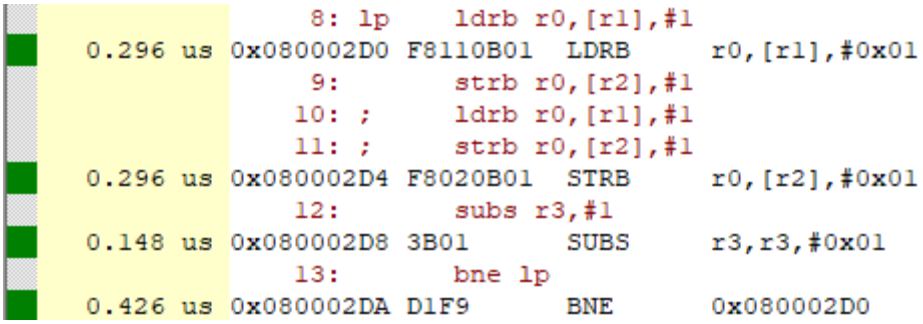


그림 37 Line 8 - 13 수행 시간

STEP 18: Line 5에서 0x10을 0x08로 변경하고, lines 10-11의 주석처리를 제거한다. 이렇게 변경한 프로그램의 동작에 따른 처리결과가 변경 전의 프로그램과 동일한지 프로그램 해석을 통해 확인한다. 변경된 프로그램을 이용해 위 처리시간 측정 과정을 반복한다.

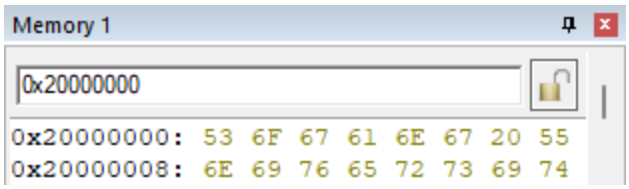


그림 38 코드 수정 후 프로그램 수행 후 메모리 값

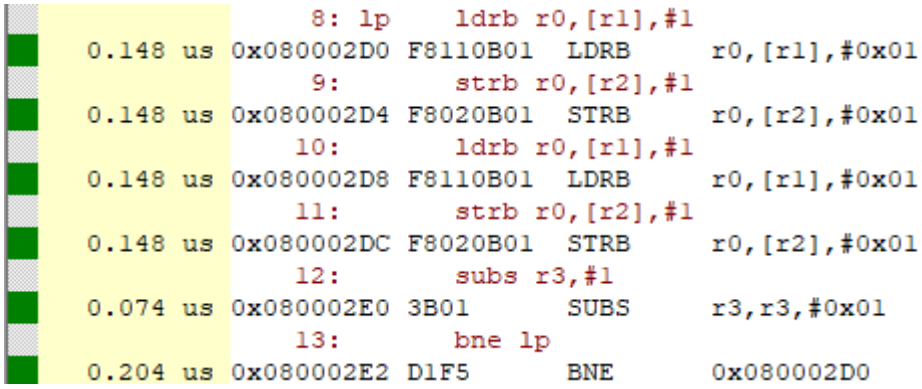


그림 39 코드 수정 후 line 8 - 13 수행 시간

코드수정 후 프로그램을 실행한 결과 [그림 38]과 같이 이전과 동일한 결과가 나왔다. 코드 수정 전에는 loop문을 16번 돌면서 매 loop마다 1byte씩 데이터를 로드하고 메모리에 저장하였는데 코드 수정 후에는 loop문을 8번 돌면서 loop문 한번에 1byte씩 두 번 데이터를 로드하고 메모리에 저장하여 똑같이 총 16byte를 메모리에 저장하였음으로 프로그램 수행 결과는 동일하다.

다만 [그림 39]에서 확인할 수 있듯이 loop문 전체의 수행시간을 더해보면 0.870us로 코드 수정 전에 비해서 수행 시간이 많이 감소한 것을 확인할 수 있다.

STEP 19: 위 두 과정을 통해 측정된 처리시간에 차이가 있다면 그 이유는 무엇때문이라고 해석할 수 있는가? Branch penalty와 연관지어 설명해보다.

[그림 37]과 [그림 39]을 살펴보면 각 코드에서 명령어는 모두 동일한 횟수로 수행되었는데 각 코드의 line 13의 branch 명령어의 수행 시간이 다른 명령어에 비해서 가장 긴 것을 확인할 수 있다. 이는 branch를 할 때 발생하는 branch penalty때문이다. 수정 전 코드에서는 branch 명령어가 총 16번 수행되고 수정 후 코드에서는 8번 수행됨으로 branch 명령어가 더 많이 수행되는 수정 전 코드에서 branch penalty가 더 많이 발생하고 따라서 수행 시간이 더 길게 나오는 것이다.

branch 명령어의 수행시간이 다른 명령어에 비해서 긴 것은 branch를 하게 되면 branch penalty가 발생하기 때문이다. branch penalty란 원래 cpu는 파이프라이닝을 통해서 한번에 하나의 동작만 하는 것이 아니라 각 파이프라인 fetch, decode, execute에 맞춰서 각 단계별로 하나씩 명령어를 수행하게 된다. 하지만 cpu 입장에서는 branch여부를 생각하지 않고 무조건 코드상 다음 명령어를 fetch하고 decode하게 된다. 하지만 이때 execute한 명령어의 결과 branch를 하게 되면 그 전에 fetch, decode하고 있던 명령어는 버려지게 되고 branch한 곳에서 새로운 명령어를 다시 fetch하게 된다. 요약하면 branch를 하게 되면 이전에 파이프라인에서 작업하고 있던 명령어를 버리고 새로운 명령어를 파이프파인에 넣고 작업하기 때문에 버려지는 cycle이 존재하게 되고 이를 branch penalty라고 한다.

4. Exercises

1) 분기 명령어들의 분기 목적지 주소지정은 PC에 대해 상대적으로 계산하는 방식과 절대번지를 사용하는 방식으로 구분할 수 있다. 이러한 주소 지정방식을 프로그램의 re-location과 연관시켜 생각해보자.

분기 명령어에서 b label과 같이 사용하는 경우는 label의 주소가 현재 PC에 대해서 상대적인 위치로 나타나 branch를 한다. 어셈블러에 의해 명령어가 배치될 때 현재 명령어와 label의 상대적인 주소를 계산해서 pc-relative 표현이 들어간다. 반면 bx rm과 같은 분기 명령어는 rm에 들어있는 절대주소 자체로 branch를 하는 명령어이다.

프로그램이 ram에 로드되어 수행될 때 실제 메모리 주소가 명령어가 relocation되는데 이때 각 명령어는 상대적인 주소에서 절대적인 주소로 바뀌어 가지게 되는데 따라서 프로그램을 작성할 당시에는 해당 프로그램의 특정 명령어가 어떤 절대주소를 가지게 될 지 알 수 없다. 따라서 b와 같이 일반적인 분기 명령어를 사용할 때 PC에 상대적인 주소로 계산해서 명령어를 사용하게 된다. 하지만 bx와 같은 명령어는 프로그램을 수행하는 도중 branch해서 돌아가야 할 주소를 lr에 저장하거나, IO장치와 같이 메모리 맵에 의해 고정된 주소를 접근할 때 주로 사용함으로 절대주소를 사용하는 방식을 취한다.

2) Conditional execution의 도입에 따른 장점들을 임베디드 시스템 구현관점에서 파악해보자.

임베디드 시스템을 구현할 때는 자원이 한정되어 있기 때문에 코드의 사이즈나 효율성을 최대한 확보해아한다. 이때 conditional execution을 도입하면 branch의 횟수가 감소함으로 branch penalty가 발생하는 빈도가 감소하고 따라서 프로그램의 수행 시간도 감소한다. 또한 원래라면 branch 명령어와 일반적인 명령어로 구성되야 할 동작을 conditional execution이 가능한 명령어 하나로 대체해서 사용할 수 있기 때문에 코드의 사이즈도 감소하게 된다.

3) Branch with link 명령어(bl)가 일반적인 명령어(b와 그의 조건분기 변형들)와 어떻게 다른지 확인한다. 이 과정에서 lr 레지스터로 사용되는 r14의 역할 및 활용방법에 대해 알아보자.

bl 명렁어는 다른 branch 명령어와 다르게 branch할 때 lr에 다음에 수행 할 명령어의 주소 + 1을 r14(lr)에 저장한다. 예를 들어 현재 명령어가 32-bit 명령어로 주소가 0x08000200이었다면 다음 명령어의 주소는 0x08000204임으로 lr에는 0x08000205를 저장한다. 뒤에 1을 붙여서 저장하는 이유는 나중에 bx나 mov와 같은 명령어로 lr의 값을 PC에 다서 로드할 때 LSB가 1이어야 오류가 발생하지 않기 때문이다. 즉 lr 레지스터는 branch를 하고나서 명령어를 수행하다가 다시 원래의 흐름으로 돌아와야 할 경우를 고려해서 돌아올 주소를 저장하는 역할을 한다. 이를 활용하면 subroutine을 만들어서 main 프로그램을 수행하다가 subroutine으로 branch 했다가 다시

main 프로그램을 수행하려 할 때 lr을 활용해서 돌아올 주소를 기억하는 등 활용할 수 있다.

4)프로그램을 작성할 때 그 흐름을 변경하는 명령어를 사용하지 않을 수 없다. 이러한 명령어의 사용이 프로세서의 동작에 부정적으로 영향을 미치는 요소(branch penalty)들이 있는데, 일반적으로 branch penalty를 어떻게 정의하는지 알아보자. 또한 loop unrolling의 채택에 따른 trade-off를 정래해보자.

프로그램에서 branch명령어를 사용해서 흐름을 변경할 수 있다. 이때 CPU에서 파이프라이닝을 통해서 명령어를 수행하기 위해서 여러 단계를 거칠 수 있다. 우리가 사용하는 ARM cortex-m3에서는 3단계의 파이프라인이 존재한다. 명령어를 fetch하고, decode하고, execute하는 단계로 이루어진다. 만약 명령어가 branch라면 명령어를 수행하고 나서 다음 명령어가 바로 execute되는 것이 아니라 fetch부터 다시 해야 함으로 일부 cycle이 낭비되는데 이를 branch penalty라고 한다. Loop unrolling은 이러한 branch횟수를 줄여 branch penalty를 줄이기 위한 기법이다. 간단히 설명하면 loop문에서 branch를 통해 동일한 동작을 여러 번 수행하는 대신 코드를 길게 쭉 나열해서 동일한 동작을 여러 번 수행하도록 하는 것이다. 이 방법을 사용하면 branch 횟수는 줄어들기 때문에 branch penalty는 감소한다. 하지만 코드의 사이즈가 길어지고 가독성이 떨어지며 코드를 유지보수 하는 것이 어려워지는 단점이 있다.

5) Program 5.5의 line 13에서 사용된 bne 명령어가 조건을 만족해서 lp로 분기하는 경우와 조건을 만족하지 않아 그 다음 명령어로 진행되는 경우 수행 시간이 다른데 각각 얼마로 측정되는가? 측정한 결과를 뒷받침하는 근거를 이 명령어의 처리시간을 소개하는 문헌을 통해 찾아보자.

bne 명령어가 조건을 만족해서 lp로 분기하는 경우는 0.028us가 소요되며 3cycle이 걸린 것을 알 수 있다. 반면 bne 명령어가 조건을 만족하지 않아 바로 다음 명령어가 수행되는 경우는 0.009us가 소요되며 1cycle이 걸린 것을 측정할 수 있었다. 이에 대한 근거는 ARM에서 발간한 기술문서인 ARM7TDMI Technical Reference Manual의 6-4쪽에서 가져온 아래 [그림 40]에서 확인할 수 있다. 이 문서에 의하면 branch 명령어가 첫번째 cycle에서 branch할 목적지를 계산하게 된다. 그리

고 두번째 cycle에서 branch 목적지에서 새로운 명령어를 fetch하게 되고 세번째 cycle에서 목적지에서 다음 명령어는 fetch하고 목적지의 명령어는 decode한다. 그리고 그 다음 사이클에서는 branch 목적지의 명령어가 수행됨으로 branch 명령어를 수행할 때 총 3 cycle이 소모되는 것을 알 수 있다. 하지만 bne 명령어의 조건을 만족하지 않은 경우는 새로운 명령어를 fetch할 필요 없이 이미 fetch한 명령어를 바로 이어서 수행하면 되기 때문에 1 cycle만 소모되는 것이다.

Table 6-1 Branch instruction cycle operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
1	pc+2L	i	0	(pc+2L)	0	0	0
2	alu	i	0	(alu)	0	1	0
3	alu+L	i	0	(alu+L)	0	1	0
	alu+2L						

그림 40 Branch 명령어의 cycle

5. 추가 실험

1) Literal Pool

아래 [그림 41]에서 확인 가능한 것처럼 비분기 명령어 사이에 literal pool이 발생하는 경우에는 프로그램을 수행하다 PC가 다음 명령어를 찾지 못해서 프로그램의 동작이 멈추게 된다. 아래 [그림 41]에서도 화살표가 가리키고 있는 명령어에서 다음 명령어를 찾지 못해 더 이상 프로그램이 수행되지 못하는 문제가 발생하였다.

```
    12:                 ; Danger Section
0x080002D4 EBA00301  SUB      r3,r0,r1
    13:                 ldr.n r7, =0xE7FE
    14:                 ltorg
    15:
    16:                 space 4096
    17:
0x080002D8 4F00      LDR      r7,[pc,#0]   ; @0x080002DC
0x080002DA 0000      DCW      0x0000
0x080002DC E7FE      DCW      0xE7FE
0x080002DE 0000      DCW      0x0000
0x080002E0 0000      DCW      0x0000
```

그림 41 비분기 명령에 사이의 literal pool

또한 만약 ltorg라는 directive가 없어서 line 13 명령어의 pseudo instruction을 위한 0xE7EE가 저

장될 literal pool이 end 명령어 뒤에 등장하게 된다. 하지만 이때 space 4096으로 명령어와 접근하려는 literal pool 사이에 4KB가 넘는 거리가 발생하기 때문에 아래 [그림 42]에서와 같이 오류가 발생하는 것을 확인할 수 있다.

```
Build started: Project: lab5_5
*** Using Compiler 'V6.19', folder: 'C:\Keil_v5\ARM\ARMCLANG\Bin'
Build target 'Target 1'
assembling lab5_5.s...
lab5_5.s(13): error: A1284E: Literal pool too distant, use LTORG to assemble it within 4KB
".\Objects\lab5_5.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:01
```

그림 42 ltorg 제거 후 발생한 에러

이처럼 literal pool이나 다른 공간 할당에 의해 PC가 다음 명령어를 찾지 못하는 경우가 발생할 수 있다. 또는 literal pool이 너무 멀어서 명령어가 literal pool에 접근하지 못하는 경우도 발생할 수 있다. 따라서 이 두가지 문제를 모두 해결하기 위해서는 우선적으로 literal pool이 명령어 가까이 존재해야 하는데 이를 위해서는 bl로 branch를 한 뒤 해당 영역에서 원하는 동작을 수행한 뒤 bx명령어로 돌아오도록 구성하며 이때 bx 명령어 바로 뒤에 ltorg를 통해 literal pool이 가까이 있을 수 있도록 해준다. 그리고 아래 코드의 space 4096처럼 명령어 중간에 공간이 할당되어 명령어가 끊어져 있다면 동일하게 branch를 통해서 다음 명령어의 주소로 찾아갈 수 있도록 b 명령어를 사용해준다. 그리고 만약 거리가 멀다면 32-bit branch 명령어를 사용해준다. 이를 모두 해결한 코드와 수행 결과는 아래 [그림 43], [그림 44]와 같다. 이번 추가실험을 통해서 긴 코드 사이에서 데이터를 원활히 접근하고 사용할 수 있도록 branch를 활용하는 방법에 대해서 익혔다.

```
1     area lab5_5,code
2     entry
3  __main proc
4     export __main [weak]
5  start
6           ldr r0, =0x12
7           ldr r1, =0x33
8           ldr r2, =0xEE
9           sub r3, r0, r1
10          ; Danger Section
11          bl func1
12          b.w func2
13          space 4096
14
15 func1    ldr.n r7, =0xE7FE
16          bx lr
17          ltorg
18 func2    ldr r0, =0xAA
19          ldr r1, =0xBB
20          ldr r2, =0xCC
21          sub r3, r0, r1
22          b .
23    endp
24    align
25 dummy1 dcd 0xFFFFEEEE
26 dummy2 dcd 0x12341234
27 dummy3 dcd 0xABCABCDD
28    end
```

그림 43 Branch를 활용해 오류를 수정한 코드

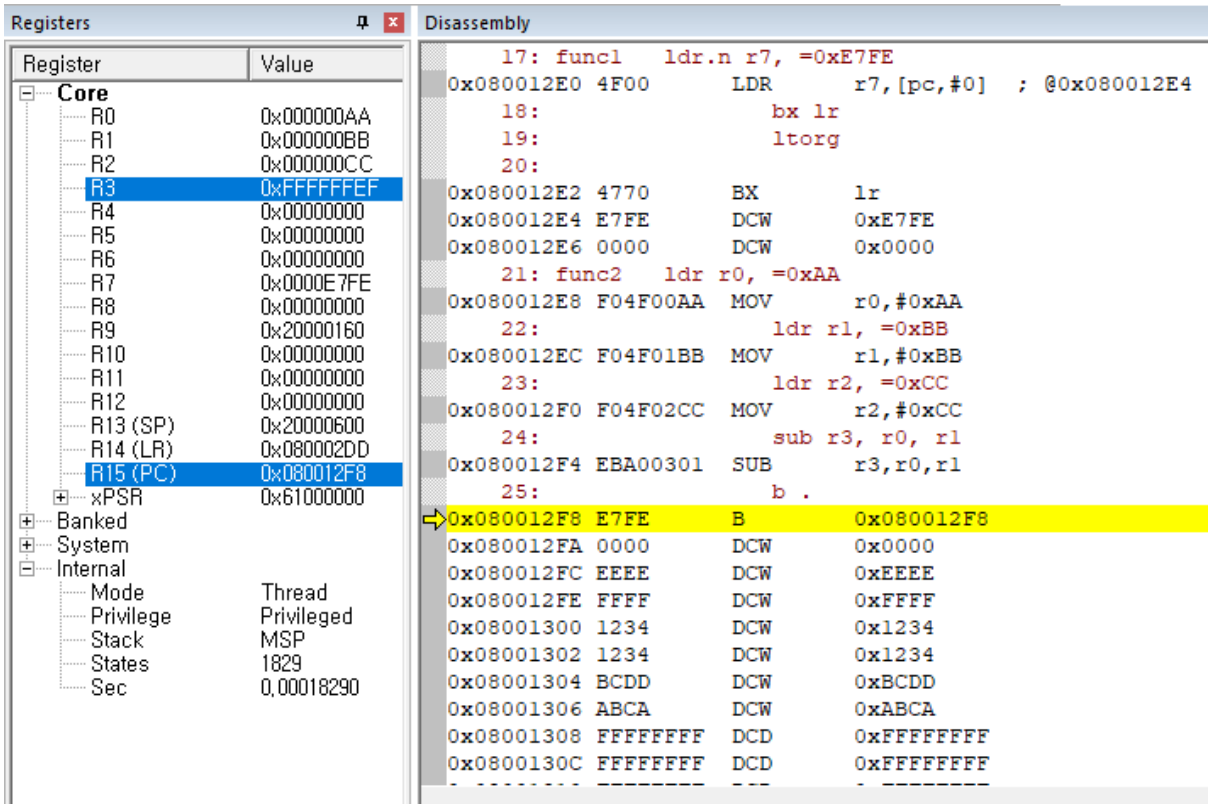


그림 44 프로그램 수행 완료된 모습

2) Bubble Sort

정렬은 실행활이나 컴퓨터에서 매우 중요한 문제이다. loop문을 이용하면 bubble sort를 이용해서 간단하게 정렬할 수 있다. Bubble sort를 오름차순을 기준으로 간단히 설명하면 맨 앞에서부터 양 옆의 숫자를 비교해서 큰 숫자가 오른쪽에 오도록 해준다. 이를 모든 숫자에 대해서 수행하고 나면 맨 오른쪽에는 가장 큰 숫자가 정렬된다. 이제 다음으로는 맨 오른쪽 숫자를 제외하고 동일하게 하나씩 비교하며 두번째로 가장 큰 숫자가 오른쪽에서 두번째가 되도록 정렬한다. 이를 모든 숫자가 정렬될 때까지 반복하게 하면 모든 숫자가 정렬되는 알고리즘이다. 어셈블리에서 branch와 conditional execution을 활용해서 아래 [그림 45]와 같이 bubble sort 코드를 작성할 수 있다. 그리고 [그림 46]에서와 같이 오름차순으로 정렬된 것을 확인할 수 있다.

```
1     area lab5_6,code
2     entry
3  __main proc
4     export __main [weak]
5  start ldr r0,=data2
6        adr r1,data1
7        ldm r1,{r2-r6}
8        stm r0,{r2-r6}
9        mov r1, #5
10
11 loop1 mov r2,#0
12       mov r3,#0
13 loop2 add r4,r2,#1
14       cmp r4,r1
15       beq check
16       ldr r5,[r0,r2,lsl #2]
17       ldr r6,[r0,r4,lsl #2]
18       cmp r5,r6
19       strgt r6,[r0,r2,lsl #2]
20       strgt r5,[r0,r4,lsl #2]
21       addgt r3,#1
22       mov r2,r4
23       b loop2
24 check cmp r3,#0
25       subgt r1,#1
26       bgt loop1
27
28         b .
29         align
30 data1 dcd 0x00000007, 0x00000004, 0x00000005, 0x00000001, 0x00000003
31    endp
32    align
33    area lab5data,data
34 data2 space 16
35    end
```

그림 45 직접 작성한 bubble sort 코드

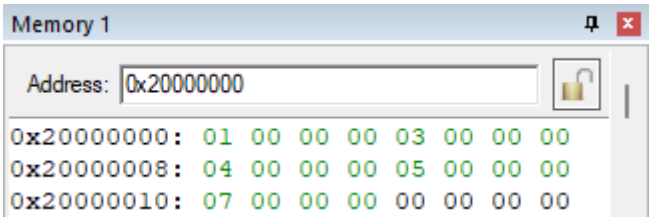


그림 46 Bubble sort 수행 후 결과

[그림 45]의 코드에 대해 간략이 설명하자면 코드 앞부분에서 data1의 데이터를 전부 r2-r6로 가져와서 메모리 공간에 저장해주었다. 먼저 r1은 정렬해야 할 배열의 크기를 저장하고 r2는 loop2에서 비교하는 숫자가 몇 번째 element인지를 나타내고 r6는 swap 횟수를 기록한다. loop2에 들어가면 r2와 그 옆인 r4를 비교해서 r2가 크다면 스왑하고 아니라면 r4를 r2에 넣어 비교할 숫자를 옆으로 한 칸 이동한다. 계속해서 loop2를 반복하다가 r4가 r1과 같아져 배열의 끝에 도달하면 check로 branch해서 배열의 크기인 r1을 하나 줄여주고 만약 loop2에서 swap이 한 번도 일어나지 않아서 r3 = 0이었다면 그대로 sort를 종료한다. 이 과정을 거치게 되면 bubble sort가 완료된다.

3) Additional Instruction – CBZ, CBNZ

```
 1                          area lab5_7,code
 2                          entry
 3                        __main proc
 4                          export __main [weak]
 5    0.009 us            start mov r0,#0
 6    0.009 us                  cmp r0,#0
 7    0.028 us                  beq j1
 8                              mov r1,#0
 9    0.009 us            j1    mov r1,#1
10    0.028 us                  cbz r0,j2
11                              mov r2,#0
12    0.009 us            j2    mov r2,#1
13    0.009 us                  mov r0,#5
14    0.009 us                  cmp r0,#0
15    0.028 us                  bne j3
16                              mov r3,#0
17    0.009 us            j3    mov r3,#1
18    0.028 us                  cbnz r0,j4
19                              mov r4,#0
20    0.009 us            j4    mov r4,#1
21    0.028 us                  b .
22                              endp
23                              end
```

그림 47 직접 작성한 lab5_7.s 코드와 수행시간

CBZ와 CBNZ 명렁어의 사용방법과 동작을 알아보기 위해서 [그림 47]과 같이 코드를 작성하였다. CBZ Rn, label과 같은 방식으로 명령어를 사용하며 Rn의 데이터가 0이라면 label로 branch하고 0이 아니라면 그대로 진행하는 명령어이다. 그리고 CBNZ Rn, label은 유사하게 Rn의 데이터가 0이 아니라면 label로 branch하고 0이면 그대로 진행하는 명령어이다. [그림 47]의 코드를 보면 line 10과 line 18에서 모두 조건에 따라서 정상적으로 branch한 것을 확인할 수 있다.

여기서 line 6-7의 cmp, beq 2줄의 명령어와 line 10의 cbz명령어가 동일한 동작을 수행하는 것을 알 수 있는데 수행시간 측면에서 본다면 0.037us와 0.028us임으로 cbz명령어로 줄일 수 있다면 무조건 줄이는 것이 코드의 사이즈나 수행시간 측면에서 모두 효율적이다. 동일하게 line 14-15의 cmp, bne 2줄의 명령어와 line 18의 cbnz명령어도 동일한 동작을 수행하며 수행시간과 코드 사이즈 측면에서 모두 효율적인 것을 알 수 있다.

다만 한가지 차이점이라면 cmp명령어는 flag를 업데이트 하는데 반해서 cbz와 cbnz명령어는 flag를 업데이트 하지 않는다는 특징이 있다. 따라서 만약 flag의 업데이트가 꼭 필요한 상황이라면 cbz, cbnz명령어 대신 cmp와 beq 또는 bne 명령어를 사용해야 한다.

4) Day Calculator

```
area lab5_8,code
        entry
__main proc
        export __main [weak]
start ldr r0,data1
              mov r2,#10000
              mov r8,#0
              udiv r1,r0,r2
              sub r1,#1
year1 cmp r1,#400
```

```
            blt year2
        subs r1,#400
            addmi r1,#400
            b year1
year2 cmp r1,#300
        addge r8,#1
            subge r1,#300
            bge year3
            cmp r1,#200
            addge r8,#3
            subge r1,#200
            bge year3
            cmp r1,#100
            addge r8,#5
            subge r1,#100
year3 add r8,r1
            mov r2,#4
        udiv r3,r1,r2
            add r8,r3


        mov r2,#100
            udiv r1,r0,r2
            mov r2,#10000
            udiv r3,r0,r2
            mov r2,#100
            mul r3,r2
```

```
                sub r1,r3 ;r1 month
                adr r4,table
                sub r2,r1,#1
                tbb [r4,r2]
jan     add r8,#0
                b day
feb     add r8,#3
                b day
mar     add r8,#3
                b leap
apr     add r8,#6
                b leap
may     add r8,#8
                b leap
jun     add r8,#11
                b leap
jul     add r8,#13
                b leap
aug     add r8,#16
                b leap
sep     add r8,#19
                b leap
oct     add r8,#21
                b leap
nov     add r8,#24
                b leap
```

```
dec     add r8,#26
            b leap
leap    mov r2,#10000
            udiv r3,r0,r2
loop    mov r2,#4
            udiv r4,r3,r2
            mul r4,r2
            cmp r4,r3
            bne day
            add r8,#1
            mov r2,#100
            udiv r4,r3,r2
            mul r4,r2
            cmp r4,r3
            bne day
            sub r8,#1
            mov r2,#400
            udiv r4,r3,r2
            mul r4,r2
            cmp r4,r3
            bne day
            add r8,#1
day     mov r2,#100
            mul r1,r2
            mov r2,#10000
            udiv r3,r0,r2
```

```
            mul r3,r2
            sub r0,r1
            sub r0,r3
            add r8,r0
dodiv mov r2,#7
            udiv r7,r8,r2
            mul r7,r2
            sub r8,r7
            b .
            align
table dcb 0
        dcb ((feb - jan)/2)
            dcb ((mar - jan)/2)
            dcb ((apr - jan)/2)
            dcb ((may - jan)/2)
            dcb ((jun - jan)/2)
            dcb ((jul - jan)/2)
            dcb ((aug - jan)/2)
            dcb ((sep - jan)/2)
            dcb ((oct - jan)/2)
            dcb ((nov - jan)/2)
            dcb ((dec - jan)/2)
        endp
data1 dcd 19450815
        end
```

이번 추가실험에서는 특정 날짜가 주어졌을 때 요일을 계산해주는 프로그램을 만들어 보았다. 코

드의 마지막에 있는 data1에 yyyymmdd의 꼴로 숫자를 입력하면 r8에 요일을 계산해주는 프로그램이다. r8에 저장된 결과가 0이면 일요일, 1이면 월요일 이런 식으로 숫자가 하나 증가할 때 마다 요일이 하나씩 증가하며 즉 6은 토요일이다.

코드를 전부 설명하기는 어렵지만 흐름으로만 보면 먼저 년도를 보고 요일을 계산해준다. 공식이 도출된 과정은 1년 1월 1일이 일요일인 것을 기준으로 해서 400년 마다 요일이 같아지고 100년, 200년, 300년일 때의 각 요일을 계산해주고 그 나머지는 윤년을 계산해서 년도의 1월 1일 요일을 계산해주었다. 이 과정에서 branch를 이용해서 400으로 계속 빼주고 100년, 200년, 300년에 따라 case문처럼 구성해주었다. 다음으로 각 월에 대해 table을 만들어서 월마다 더해주어야 할 요일을 미리 계산하고 jump table을 이용해서 branch해주었다. 그리고 마지막에는 윤년 여부를 계산해서 요일을 보정해주었다. 마지막으로 지금까지 구한 요일에 일을 더해서 요일을 구한 뒤 보기 좋게 7로 나눈 나머지를 r8에 저장하였다.

코드에서 conditional execution이나 branch, jump table을 사용하여 나름 편하게 코드를 작성할 수 있었다. 물론 나중에 subroutine을 배운다면 조금 더 코드를 정리할 수 있겠지만 우선은 정확하게 돌아가는 요일 계산기를 완성할 수 있었다. 몇 가지 날짜에 대한 프로그램 수행 결과는 아래 [그림 48], [그림 49에, [그림 50]서 확인 가능하다.

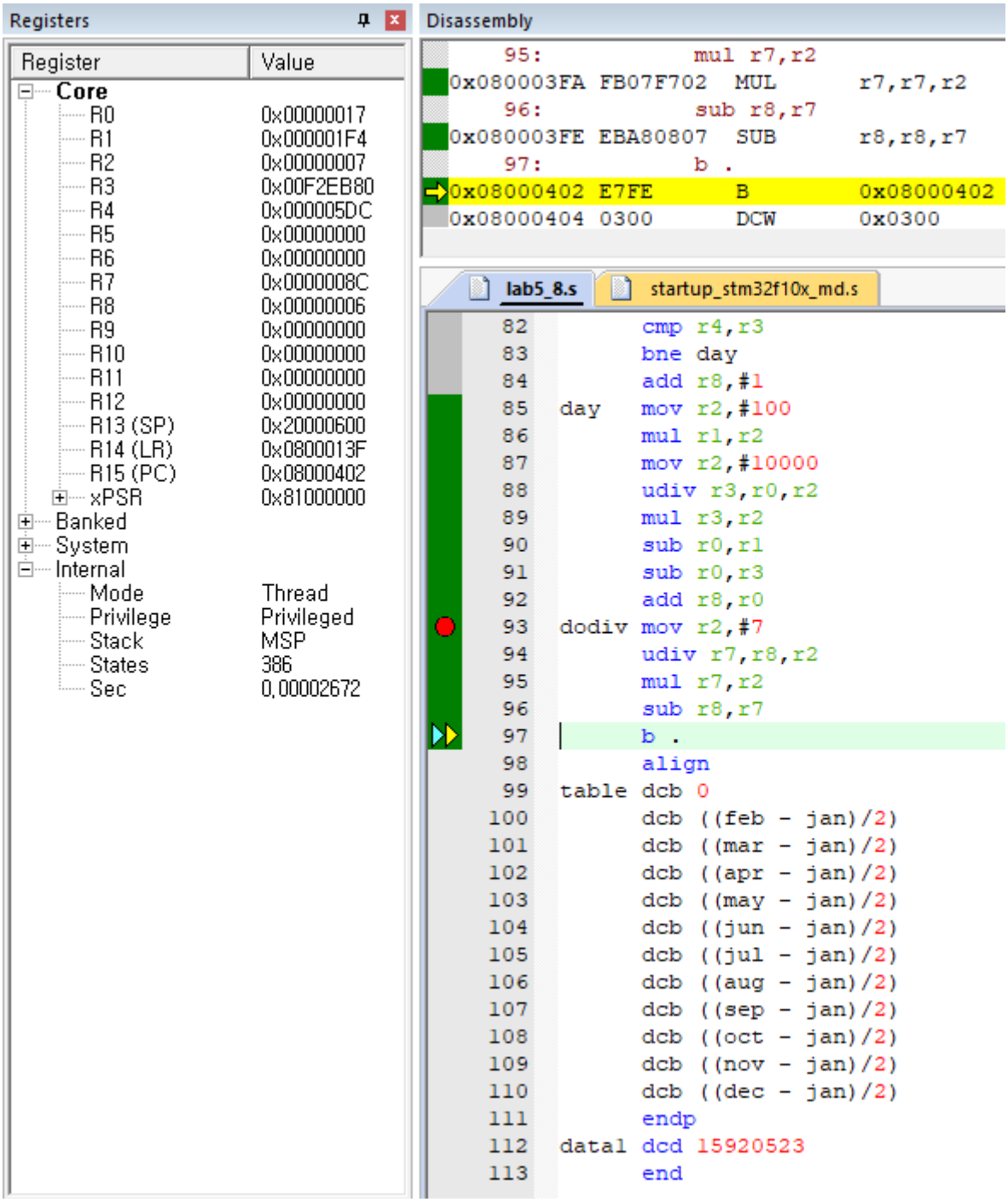


그림 48 임진왜란 날짜 계산 결과 – 토

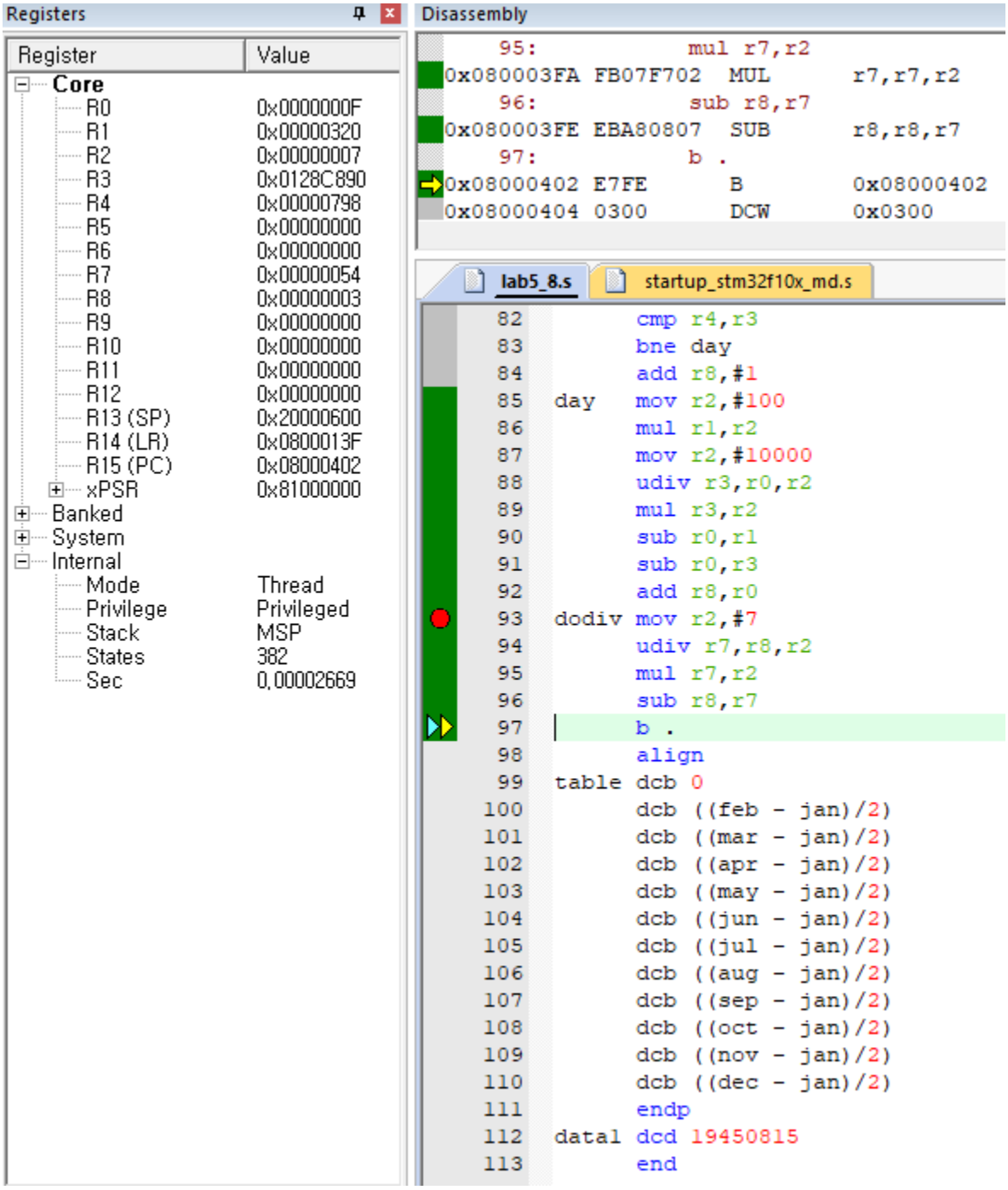


그림 49 광복절 날짜 계산 – 수

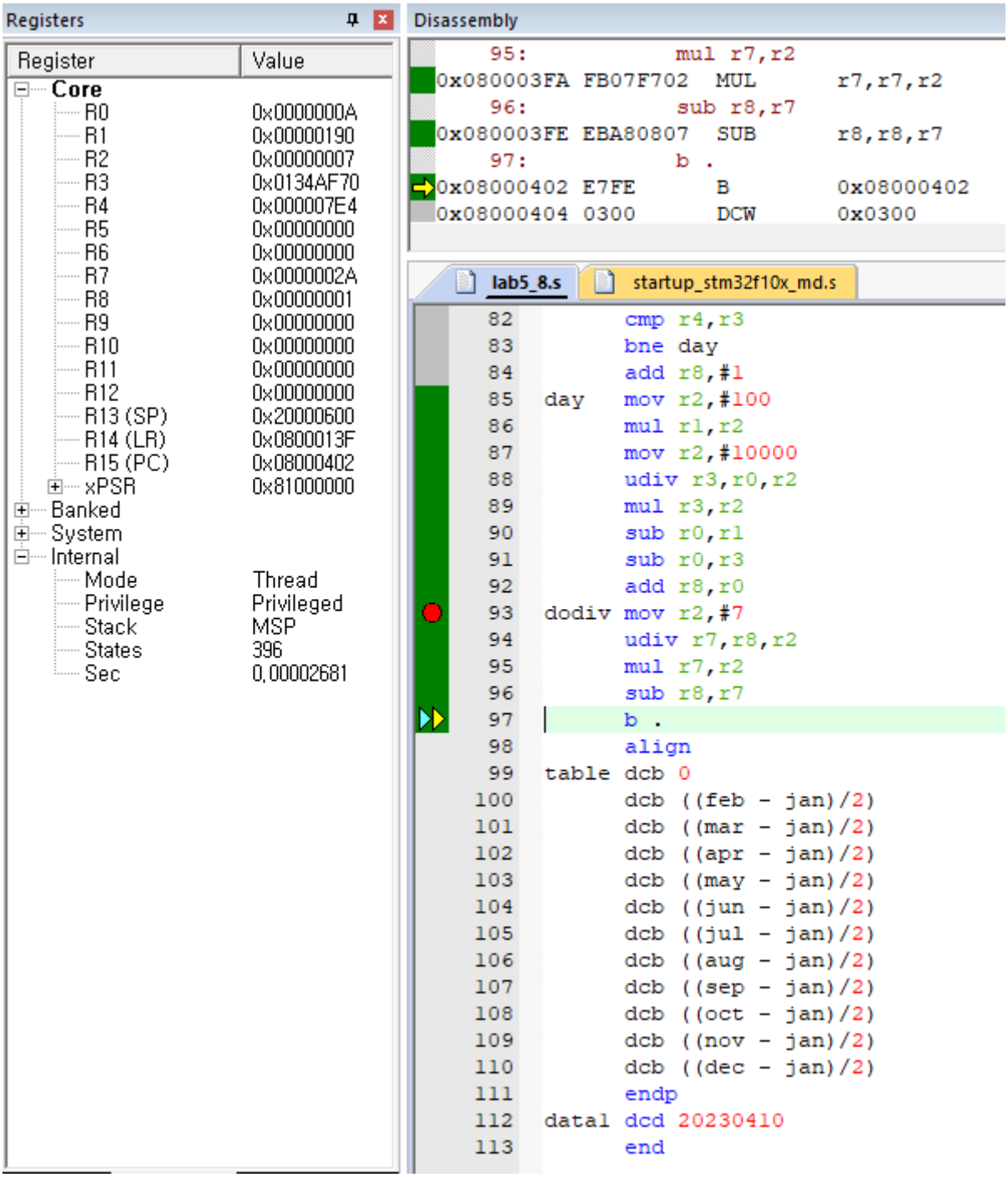


그림 50 보고서 제출일 날짜 계산 - 월

5) HardFault

```
1     area lab5_9,code
2     entry
3  __main proc
4     export __main [weak]
5  start bl func1
6        b .
7  func1 sub lr,#1
8        bx lr
9        endp
10       end
```

그림 51 직접 작성한 lab5_9.s 코드

```
130  ; Reset handler
131  Reset_Handler    PROC
132                   EXPORT  Reset_Handler             [WEAK]
133       IMPORT   __main
134       IMPORT   SystemInit
135                   LDR     R0, =SystemInit
136                   BLX     R0
137                   LDR     R0, =__main
138                   BX      R0
139                   ENDP
140
141  ; Dummy Exception Handlers (infinite loops which can be modified)
142
143  NMI_Handler     PROC
144                  EXPORT  NMI_Handler               [WEAK]
145                  B       .
146                  ENDP
147  HardFault_Handler\
148                  PROC
149                  EXPORT  HardFault_Handler         [WEAK]
150                  B Reset_Handler
151                  B       .
152                  ENDP
```

그림 52 직접 작성한 HardFault_Handler

추가실험5 에서는 bx명령어를 사용할 때 만약 LSB가 1이 아니라 0이면 어떻게 될지 궁금해서 실험을 진행했다. 이를 확인하기 위해서 [그림 51]과 같이 간단하게 코드를 작성하였다. bl명령어를 사용해 func1의 위치로 branch하고 그 내부에서 lr에 저장된 값에서 1을 빼서 LSB를 0으로 만들었다. 그리고 다시 bx lr명령어로 원래 위치로 돌아가려고 코드를 작성하였더니 HardFault가 발생하며 강제로 HardFault_Handler로 branch하였다.

HardFualt란 arm에서 발생하는 다양한 exception handling도중 에러가 발생하거나 다른 exception에 의해 처리될 수 없는 exception이 발생한 경우 HardFault가 발생하여 handler로 들어가서 동작

을 수행한다. 아직 exception에 대해서 자세히 아는 바가 없어서 가장 간단하게 [그림 52]와 같이 handler코드를 작성하였다. 프로그램을 처음 수행하면 Reset_Handler로 들어가는데 이를 참고하여 HardFault가 발생하면 프로그램을 reset 시키기 위해 B Reset_Handler로 프로그램이 리셋되도록 하였다.

이번 추가실험을 통해서 만약 bx명령어를 사용할 때 LSB가 1이면 어떻게 에러가 발생하는지 확인하였고 HardFault_Handler를 간단히 작성하여 에러가 발생하였을 때의 처리를 하였다.

6. 결론

이번 실험을 통해서 branch라는 매우 중요한 기능을 다루고 배워볼 수 있었다. branch는 프로그램을 순차적으로만 실행되지 않고 조건이나 상황에 따라 우리가 원하는 흐름으로 프로그램이 동작할 수 있게 해주는 매우 유용한 기능임을 확인할 수 있었다. 또한 ARM에서 지원해주는 conditional execution이 얼마나 유용한 기능인지 확인하고 branch를 잘 활용하기 위한 다양한 명령어의 활용과 jump table의 사용법, 마지막으로 branch에 따른 단점과 이를 약간이나마 해결하도록 도와줄 수 있는 loop unrolling까지 배워볼 수 있었다. 이를 통해 조건에 따라 동작을 다르게 하거나 반복적인 동작을 수행하는 loop문을 구성하는 방법을 익힐 수 있었다. 또한 추가실험에서는 branch를 활용해서 기존에는 해볼 수 없었던 유용한 기능을 프로그램도 직접 만들어 볼 수 있었다.

7. 참고문헌

서강대학교 전자공학과 (2023). 마이크로프로세서개론. p.97~109. Appendix A

STMicroelectronics (2017). STM32F10xxx/20xxx/21xxx/L1xxx Cortex-M3 Programming Manual (Rev 6)

ARM. (2001). ARM7TDMI Technical Reference Manual (Rev 3).

Yiu, J. (2010). The Definitive Guide To The ARM Cortex-M3 (2nd Edition)

히언. (2021). EMBEDDED RECIPES.