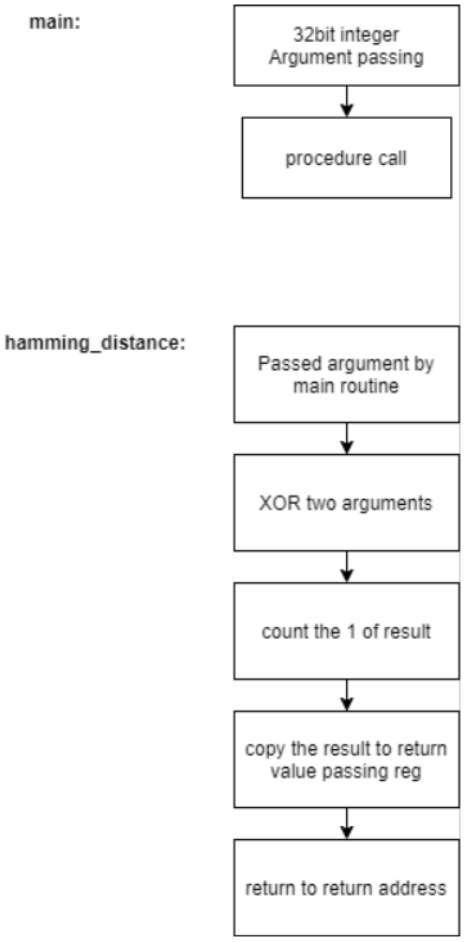
임베디드시스템 설계 및 실습 – 보고서

제출일: 2021-04-07; Lab 2
이름: 박관영(2016124099), 오한별(2018124120), 김서영(2017106007) ;4조

✓ Assignment 1

개요



Assignment 1은 Arm assembly를 통해 두 32bit integer의 hamming distance를 계산하는 것이다. Hamming distance는 두 integer에 대해 bitwise XOR 연산을 취한 뒤 그 결과의 모든 bit들을 더함으로써 구할 수 있다. Hamming distance를 계산하는 procedure와 해당 procedure를 call하고 argument를 passing하는 main routine으로 assignment program을 구현하였고, program의 결과는 main routine의 exit code로 확인할 수 있도록 하였다.

procedure의 연산 결과를 exit code로 확인하기 위해서는 APCS에서 규정하고 있는 register의 용도에 맞게 프로그램을 구현할 필요가 있다. 해당 예제에서는 return value를 passing하는 r0를 사용하여 program의 exit code로 procedure의 연산 결과를 확인할 수 있었다. Procedure call을 하는 main routine은 C 또는 assembly 로 구현할 수 있는데, main routine을 C로 구현할 경우에는 procedure의 argument를 passing하는 register도 APCS에 맞게 구현할 필요가 있다. 또한 C를 compile할 때 gcc의 -marm option을 통해서 compile함으로써 arm state로 동작하도록 할 수 있고, assembly로 구현된 procedure를 call하여 APCS에서 규정된 register를 통해서 argument를 passing하고, return value를 받을 수 있다.

진행 내용

```
.global main

main:
        ldr     r0, argument_1      @@ 32bit size argument를 passing하기 위해 ldr 명령을 사용한다.
        ldr     r1, argument_2      @@ (mov 명령의 immediate field size는 32bit보다 작기 때문에 다음과 같이
                                       구현한다.)
        b       hamming_distance    @@ procedure call
        bx      lr                  @@ Return to return address

argument_1:     .word   0x12341234
argument_2:     .word   0x43214321

----------------------------------------------------------------------asm_main_hamming_distance.s
```

```
.global hamming_distance

hamming_distance:
        eor             r2, r0, r1          @@ 두 argument에 대해 bitwise XOR 연산을 수행.

        mov             r0, #1              @@ and연산을 통해 위 연산 결과의 모든 bit를 검사하기 위한 값.
        mov             r3, #0              @@ 모든 bit를 accumulate 할 register.

CAL_DISTANCE:
        cmp             r2, #0              @@ eor 연산 결과가 0이 될 때까지 routine 반복.
        beq             RETURN              @@ 0이 되면 routine을 빠져나간다.

        and             r1, r0, r2          @@ 매 routine마다 eor 된 값의 LSB를 검사한다.
        mov             r2, r2, LSR#1       @@ 다음 bit를 검사하기 위해 logical shift right 한다.
        add             r3, r1, r3          @@ 검사한 값을 accumulate한다.

        b               CAL_DISTANCE        @@ routine 반복.

RETURN:
        mov             r0, r3              @@ ACPS의 규정을 지키기 위해 return value passing reg에 값을 mov.
        bx              lr                  @@ lr의 값으로 pc값을 update. (return)
```

hamming_distance.s

```
#include<stdio.h>

int hamming_distance(int, int);                                     @@ assembly로 구현된 procedure 선언.

int main(void){
        int argument_1 = 0x12341234;                                @@ argument passing to r0
        int argument_2 = 0x43214321;                                @@ argument passing to r1
        int exit_code = hamming_distance(argument_1, argument_2);   @@ procedure 결과를 저장.
        return exit_code;                                           @@ 그 값을 exit code로 return.
}
```

main_hamming_distance.c

진행 결과

```
root@delsoclinux:~/lab2/assignment1# cat hamming_distance.s
.global hamming_distance

hamming_distance:
        eor             r2, r0, r1

        mov             r0, #1
        mov             r3, #0

CAL_DISTANCE:
        cmp             r2, #0
        beq             RETURN

        and             r1, r0, r2
        mov             r2, r2, LSR#1
        add             r3, r1, r3

        b               CAL_DISTANCE

RETURN:
        mov             r0, r3
        bx              lr
```

>>> hamming_distance.s (procedure file)

```
root@de1soclinux:~/lab2/assignment1# cat asm_main_hamming_distance.s
.global main

main:
                ldr     r0, argument_1
                ldr     r1, argument_2

                b       hamming_distance
                bx      lr

argument_1:     .word   0x12341234
argument_2:     .word   0x43214321
```

>>> main_hamming_distance.s (main routine)

➔ vim editor를 사용하여 assembly source code를 생성하였다.

```
root@de1soclinux:~/lab2/assignment1# as -o hamming_distance.o hamming_distance.s
root@de1soclinux:~/lab2/assignment1# as -o asm_main_hamming_distance.o \
>  asm_main_hamming_distance.s
```

➔ as 명령을 사용하여 두 assembly file을 assemble 하였다.

```
root@de1soclinux:~/lab2/assignment1# gcc hamming_distance.o asm_main_hamming_\
> distance.o -o asmVER
root@de1soclinux:~/lab2/assignment1# ./asmVER
```

➔ gcc의 -o option을 통해 두 assemble된 object를 link 하여 executable file을 생성한 후 실행하였다.

```
root@de1soclinux:~/lab2/assignment1# echo $?
12
```

➔ exit code를 통해 program의 결과를 확인할 수 있도록 구현하였으므로 echo $? 명령을 통해 program의 exit code를 print out 한다. 32bit integer 0x12341234와 0x43214321를 argument로 넘겨주었기 때문에 그 결과인 12가 정상적으로 출력되는 것을 확인할 수 있다.

```
root@de1soclinux:~/lab2/assignment1# cat main_hamming_distance.c
#include<stdio.h>

int hamming_distance(int, int);

int main(void){
        int argument_1 = 0x12341234;
        int argument_2 = 0x43214321;
        int exit_code = hamming_distance(argument_1, argument_2);
        return exit_code;
}
```

➔ C로 동일한 동작을 수행하는 main routine을 구현해 보았다.

```
root@de1soclinux:~/lab2/assignment1# gcc -c main_hamming_distance.c -o \
>  main_hamming_distance.o -marm
root@de1soclinux:~/lab2/assignment1# gcc main_hamming_distance.o \
> hamming_distance.o -o CVER
```

➔ gcc를 통해 c file을 compile한다. 이 때 -marm option을 사용하여 target machine (DE1-SoC board HPS) 에 대해 cross compilation을 수행한다. -marm option을 사용하지 않았을 경우 정상 동작 하지 않는 것을 확인할 수 있었고, main routine을 arm state로 동작하도록 compile 해야만 C로 구현된 main routine에서 assembly로 구현된 procedure를 call 할 수 있다는 것을 확인할 수 있었다. 이후 위와 마찬가지로 gcc의 -o option을 통해 link를 수행하여 executable file을 생성한다.

```
root@de1soclinux:~/lab2/assignment1# ./CVER
root@de1soclinux:~/lab2/assignment1# echo $?
12
```

➔ 다음과 같이 executable file을 실행 후 exit code를 통해 정상 동작하는 것을 확인할 수 있었다.

```
root@de1soclinux:~/lab2/assignment1# ls
CVER                          hamming_distance.o
asmVER                        hamming_distance.s
asm_main_hamming_distance.o   main_hamming_distance.c
asm_main_hamming_distance.s   main_hamming_distance.o
```

➔ ls 명령을 통해 현재 디렉토리에 있는 파일들을 확인할 수 있고, assembly로 작성된 code와 C로 작성된 code, 각 code의 object file과 link된 executable file이 있는 것을 확인할 수 있다.

```
root@de1soclinux:~/lab2/assignment1# ls -al asm_main_hamming_distance.o
-rw-r--r-- 1 root root 768 Jan  1  1970 asm_main_hamming_distance.o
root@de1soclinux:~/lab2/assignment1# ls -al main_hamming_distance.o
-rw-r--r-- 1 root root 984 Jan  1  1970 main_hamming_distance.o
```

➔ ls 명령의 -al option을 통해서 두 object file의 용량을 비교한 결과 216 byte정도 차이나는 것을 확인할 수 있었다. 단순히 procedure call 만을 수행하는 main routine 이지만 assembly로 구현된 file이 비교적 더 작은 용량을 차지한다는 것을 확인할 수 있었다.

```
00000000 <main>:
   0:   e59f0008        ldr     r0, [pc, #8]    ; 10 <argument_1>
   4:   e59f1008        ldr     r1, [pc, #8]    ; 14 <argument_2>
   8:   eafffffe        b       0 <hamming_distance>
   c:   e12fff1e        bx      lr

00000010 <argument_1>:
  10:   12341234        .word   0x12341234

00000014 <argument_2>:
  14:   43214321        .word   0x43214321
```

➔ assembly main routine object의 disassemble 결과이다. Instruction count는 6 이다.

```
Disassembly of section .text:

00000000 <main>:
   0:   e92d4800        push    {fp, lr}
   4:   e28db004        add     fp, sp, #4
   8:   e24dd010        sub     sp, sp, #16
   c:   e3013234        movw    r3, #4660       ; 0x1234
  10:   e3413234        movt    r3, #4660       ; 0x1234
  14:   e50b3010        str     r3, [fp, #-16]
  18:   e3043321        movw    r3, #17185      ; 0x4321
  1c:   e3443321        movt    r3, #17185      ; 0x4321
  20:   e50b300c        str     r3, [fp, #-12]
  24:   e51b0010        ldr     r0, [fp, #-16]
  28:   e51b100c        ldr     r1, [fp, #-12]
  2c:   ebfffffe        bl      0 <hamming_distance>
  30:   e50b0008        str     r0, [fp, #-8]
  34:   e51b3008        ldr     r3, [fp, #-8]
  38:   e1a00003        mov     r0, r3
  3c:   e24bd004        sub     sp, fp, #4
  40:   e8bd8800        pop     {fp, pc}
```

➔ C main routine object의 disassemble 결과이다. Instruction count는 17이다. C로 프로그램을 구현하면 생산적으로 코딩할 수 있지만, 다음과 같이 assembly로 직접 코딩한 것과 비교했을 때 비교적 많은 용량을 차지하는 것을 확인할 수 있다. 따라서 간단한 프로그램을 assembly로 바로 구현하게 되면 프로그램의 용량을 줄일 수 있게 된다.

결과 분석 및 팀원 간 토의 사항

Arm assembly를 통해 두 integer의 hamming distance를 계산하는 프로그램을 구현해 보았다. 프로그램의 동작은 32bit size 두 인수에 대해 EOR 연산을 수행하고, 그 결과의 1을 count하는 방식으로 구현하였다. Procedure call을 할 때 APCS에 규정되어 있는 것과 같이 register를 용도에 맞게 구현하는 것이 중요한 요소였다. 이번 Assignment와 같이 exit code로 결과를 확인하는 프로그램의 경우 기본 return value passing register에 저장되어 있는 값이 결과 값으로 도출된다. 따라서 정상적으로 프로그램을 구현하였더라도 APCS의 용도에 맞지 않게 register를 사용하게 되면 올바른 결과를 확인할 수 없게 된다. 위의 프로그램의 경우에도 결과를 return value register에 copy하는 과정이 포함되어 있다.

임의대로 register를 사용하더라도 procedure call을 한 main routine에서 그 register를 통해 return value를 받는다면 문제없이 동작할 수도 있겠지만, 다른 사용자가 사용하게 될 경우 바람직하지 못하다. 또한 C로 구현된 main routine에서 assembly로 구현된 procedure를 call할 경우 APCS에 규정되어 있는 방식대로 argument, return value passing을 한다는 것을 확인할 수 있었다. 즉 APCS에 정해진 register에 argument 값이 저장되고, 정해진 register에 있는 값을 return value로써 읽어오게 된다.

```
-mthumb
-marm
    Select between generating code that executes in ARM and Thumb
    states.  The default for most configurations is to generate
    code that executes in ARM state, but the default can be
    changed by configuring GCC with the --with-mode=state
    configure option.

    You can also override the ARM and Thumb mode for each
    function by using the "target("thumb")" and "target("arm")"
    function attributes or pragmas.
```

또한, C file과 assembly file을 link할 경우 다음과 같이 gcc의 -marm option을 사용하지 않으면 정상 동작하지 않는 것을 확인하였다. 이를 통해 C file을 arm 기반 target machine에 대해서 compile하여야 arm assembly로 작성된 file과 link 되어 정상적으로 executable file이 만들어지게 됨을 확인할 수 있었다.