어셈블리 프로그래밍 설계 및 실습

보고서

실험제목: Block Data Transfer & Stack

실험일자: 2018년 10월 11일 (목)

제출일자: 2018년 10월 25일 (목)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 화 5, 수 6,7

학 번: 2015722025

성 명: 정용훈

1. 제목 및 목적

A. 제목

Block Data Transfer & Stack

B. 목적

레지스터와 메모리간 블록단위로 정보를 저장하고 불러오는 것에 대하여 이해하며, 어셈블리에서의 함수 사용법과 Conditional execution과의 효율성에 대하여 배운다 또한 Stack에 대하여 이해하고 여러가지 Stack명령어를 익힌다.

2. 설계 (Design)

A. Pseudo code

Problem1

(1) sp에 값을 저장할 Memory의 주소 값을 저장한다.

(2) r0~r7에 0~7의 값을 순차적으로 저장한다.

(3) 레지스터의 값을 바꾸기 위해 기존의 레지스터의 값을 주소에 순차적으로 저장한다. (STMEA)

(4) 저장되어 있는 값을 원하는 레지스터에 넣기 위하여 순차적으로 값을 가져온다. r5->r4->r3->r7->r2->r0->r6->r1 (Stack Pointer가 주소 뒤쪽에 있는 것을 주의)

Problem2

(1) sp에 기존 레지스터의 값을 저장하기 위한 주소 값 저장

(2) r10에 최종 값을 저장하기 위한 주소 값 저장

(3) r0~r7까지 10~17의 기본 값 저장

(4) sp메모리에 r0~r7의 값을 저장

(5) doRegister()함수 실행: (각 레지스터 값에 레지스터의 인덱스 값 더함)

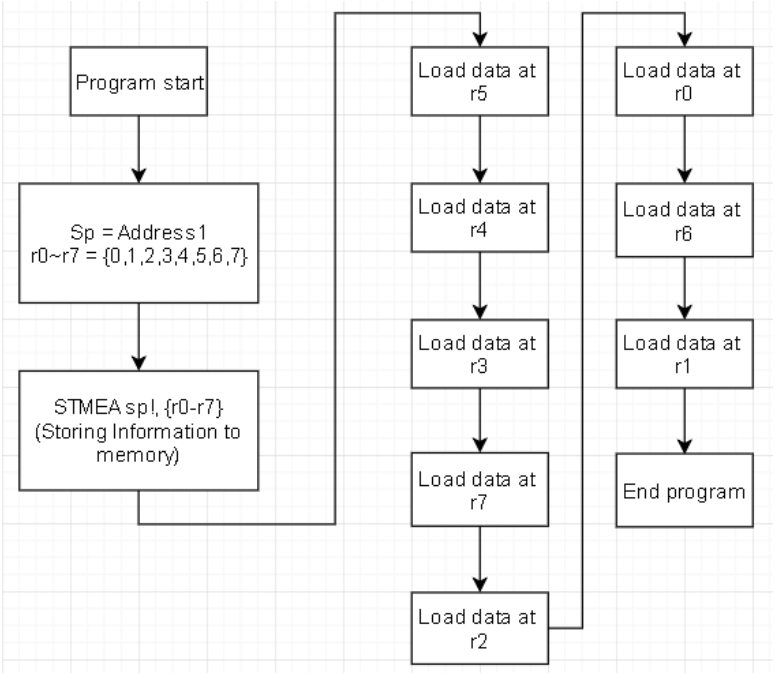
(6) doGCD()함수 실행: 레지스터의 현재 저장된 값과 160의 최대 공약수를 구하고 공약수를 최종 메모리 가장 앞에 저장한다.

(7) 최종 메모리에 기존 레지스터의 값과 doRegister()함수를 실행시킨 레지스터의 값을 더하여 저장한다.

반복적인 덧셈 연산과 값을 불러오는 자세한 명령어는 Pseudo code에서 생략했습니다.

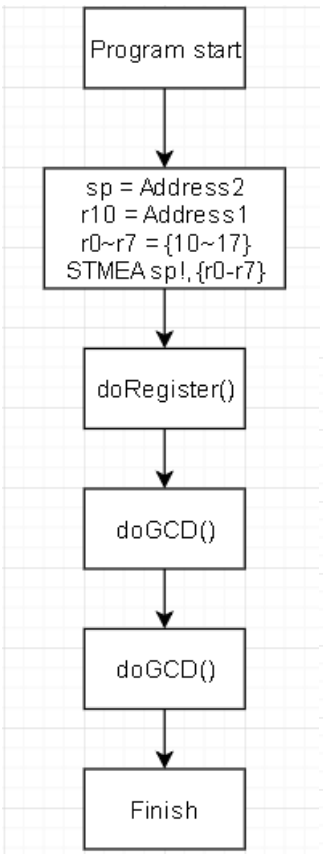
B. Flow chart 작성

Problem1

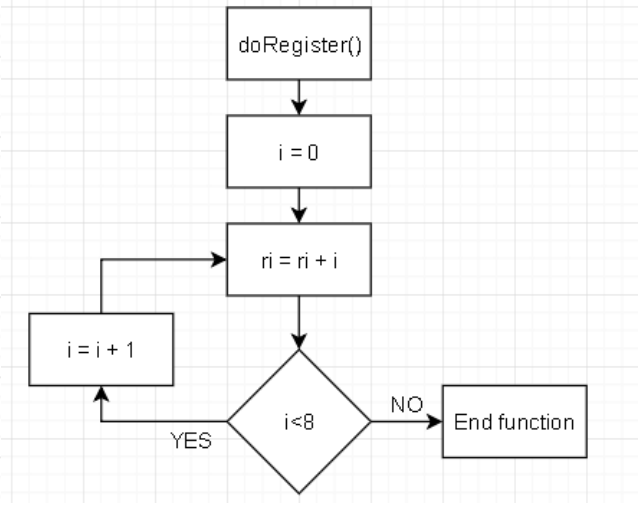


Problem2

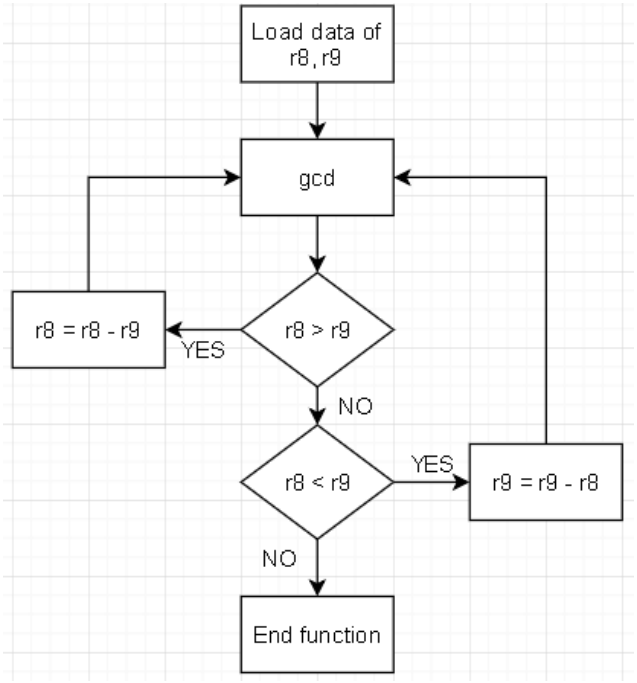
전체적인 순서



doRegister()



gcd


Load data of r8, r9
gcd
r8 > r9
YES
r8 = r8 - r9
NO
r8 < r9
YES
r9 = r9 - r8
NO
End function

C.Result

Problem1

R0	0x00000000
R1	0x00000001
R2	0x00000002
R3	0x00000003
R4	0x00000004
R5	0x00000005
R6	0x00000006
R7	0x00000007

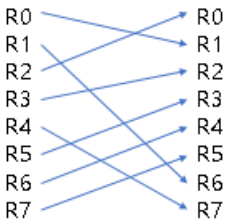
해당 이미지는 Register의 초기값입니다. 다음 이미지는 해당 값을 STMEA를 통하여 메모리에 저장한 모습입니다.

```
0x00040000: 00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00 06 00
0x0004001A: 00 00 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

STMEA의 명령으로 Stack pointer가 정보를 먼저 채우고 앞으로 이동하는 방식입니다. 해당 메모리를 보면 앞쪽부터 채워진 것을 확인 할 수 있습니다. 그리고 다음 이미지는 Stack Pointer를 고려하여 과제에서 제시한 알맞은 Register에 저장한 결과입니다.

R0	0x00000002
R1	0x00000000
R2	0x00000003
R3	0x00000005
R4	0x00000006
R5	0x00000007
R6	0x00000001
R7	0x00000004

(과제서 제시된 값 옮김)>>



Problem2

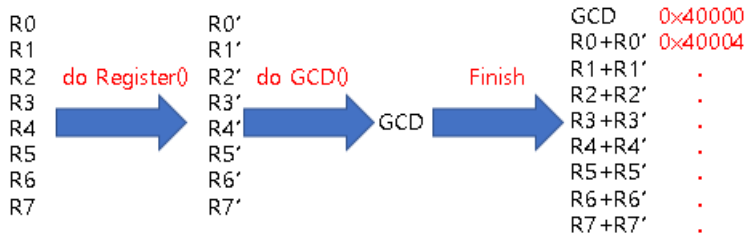
두번째 문제는 레지스터에 초기 값을 저장 후 구현된 함수를 거쳐 최종적으로 메모리에 값을 저장하는 것이 목적입니다. 처음으로 레지스터의 초기값을 저장한 모습입니다.

R0	0x0000000A
R1	0x0000000B
R2	0x0000000C
R3	0x0000000D
R4	0x0000000E
R5	0x0000000F
R6	0x00000010
R7	0x00000011

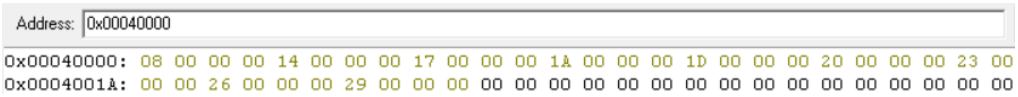
```
Address: 0x00041000
0x00041000: 0A 00 00 00 0B 00 00 00 0C 00 00 00 0D 00 00 00 0E 00 00 00 0F 00 00 00 10 00
0x0004101A: 00 00 11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

초기값을 두번째 함수를 거칠 때 사용되어야 하므로 메모리에 따로 값을 저장합니다.

R0 0x0000000A
R1 0x0000000C
R2 0x0000000E
R3 0x00000010
R4 0x00000012
R5 0x00000014
R6 0x00000016
R7 0x00000018 다음의 값은 초기 레지스터의 값을 기반으로 doRegister() 함수를 거친 레지스터의 값입니다. doRegister의 기능은 원래 있던 값에 레지스터의 인덱스 번호를 더해주는 기능을 합니다. (ex r4의 값이 3이면 3+4의 값으로 초기화) 다음으로는 두번째 프로그램의 마지막 함수인 doGCD()함수입니다. 함수의 기능은 doRegister()함수를 거친 레지스터의 모든 합과 160의 최대공약수를 구하는 함수입니다. doGCD()함수는 저장된 수를 더하는 기능과 최대공약수를 구하는 part로 나뉘게 됩니다. 마지막으로 프로그램을 마무리하는 단계는 해당 함수들을 거쳐 최대공약수를 순수데로 가장처음 초기화한 레지스터의 값과 doRegister()함수를 통과한 값을 합쳐 순차적으로 메모리에 저장하는 기능입니다. 동작 순서는 아래와 같습니다.



최종적으로 메모리에 저장되는 값은 아래와 같습니다.



3. 고찰 및 결론

A. 고찰

과제를 진행하면서 Stack명령어의 다양한 종류가 있어 많이 헷갈리는 과제였습니다. 과제를 해결하기 위한 명령어로는 저장하는 명령어와 불러오는 명령어를 하나씩 알면 되지만 각각의 명령어의 동작이 조금씩 다르므로 명령어를 정확히 이해하고 있는 것이 중요했습니다. 처음 프로그램을 작성할 때는 STMEA로 저장을 하고 LDMFA로 정보를 불러오려고 하면 제대로 불리지 않는 문제점이 있었습니다. 이는 이론을 다시 보게 되는 계기가 되었으며 STMEA로 정보를 저장하면 LDMEA, STMFA로 저장하면 LDMFA로 명령어를 실행하면 Stack pointer가 정상적으로 모든 저장된 정보를 순차적으로 불러오는 것을 확인 할 수 있었습니다. Full과 Empty의 차이로는 Stack pointer에서 차이가 있었으며 꼭 같은 Full, Empty로 맞추어 주지 않아도 상황에 따라서 사용자가 Stack Pointer의 위치에 맞게 Store명령과 Load명령을 적절하게 사용하면 데이터의 저장과 불러오는 것을 효율적으로 할 수 있다고 생각됐습니다.

B. 결론

과제에서 가장 눈에 띄는 명령어는 데이터를 Block단위로 저장하고 불러올 수 있는 명령어인 STMEA, STMFA, STMED, STMFD, LDMEA, LDMFA, LDMED, LDMFD 등이 있습니다. 평소 레지스터의 값을 메모리에 저장하고 불러오기 위해서는 STR명령과 LDR명령을 레지스터 하나마다 각각 써주어야 하지만 위에 나열한 명령어를 쓰면 데이터를 Block단위로 저장하고 불러올 수 있는 장점이 있습니다. 해당 명령을 사용하게 되면 code size와 state를 눈에 띄게 많이 줄일 수 있는 장점이 있습니다. 블록 단위로 정보를 이동시키는 조건으로는 레지스터의 인덱스 값의 오름차순과 내림차순 규칙으로 정보가 저장되기 때문에 원하는 정보가 정렬이 되어있지 않다면 명령어와 레지스터를 각각 불러야 하는 좋지 않은 case가 있을 수 있습니다. 하지만 단순히 STR과 LDR을 사용할 때와 state와 code size는 같기 때문에 정보의 규칙성만 잘 인지하면 훨씬 효율적으로 프로그램을 작성할 수 있다고 생각됩니다. 또한 Stack Pointer를 활용하면 사용자에 따라 프로그램의 performance를 좋아지게 할 수 있다고 생각됩니다.

4. 참고문헌

이준환/ARM instruction set - Block data transfer & stacks/광운대학교/2018