어셈블리어 기말

초기값이 없는 변수 선언

1. 예약 데이터 타입 : resb = 1byte, resw : 2byte(word), resd : 4byte(dword), resq: 8byte(quad word)

```
seciton .bss
        ; 일반 변수
        a resb 4

        ; 배열의 경우
        ; 배열 선언 : <변수이름> <예약 데이터 타입> <배열 개수>
        f resb 3
        d resw 4
```

초기 값이 결정된 변수 선언

```
section .data
        ; 일반 변수
        a db 0x1
        msg1 db 'haha ', 0x00  -> 문자열 끝에 null(0x00)을 꼭 넣어야함

        ;배열의 경우
        <변수이름> <데이터타입> <배열 개수만큼의 값>
        b db 30, 60,20,70,60
        c times 5 dw 0
        a1 dw 0x1234,0x5678,10
```

지시어	목적	저장 공간
DB	Byte 정의	1바이트 할당
DW	Word 정의	2바이트 할당
DD	Doubleword 정의	4바이트 할당
DQ	Quadword 정의	8바이트 할당

mov al, [a] -> a의 저장값(1바이트) 을 a1레지스터에 저장
mov eax, a -> a의 주소값을 eax(4바이트) 레지스터에 저장
mov [a], byte 0x34 -> byte 0x34를 a주소에 복사
(값에다가 넣는거면 src 변수를 []로 감싸주기)

화면 출력 매크로(SASM에서 지원)
PRINT_HEX 바이트수, 레지스터, 변수명 -> 16진수 출력
PRINT_DEC 바이트수, 레지스터, 변수명 -> 10진수 출력
PRINT_STRING 문자열 주소 -> 문자열 출력
NEWLINE -> 줄 변경

사용자 입력 매크로
GET_HEX 입력할_바이트_수, 입력_받을_주소 -> 16진수 입력
GET_DEC 입력할_바이트_수, 입력_받을_주소 -> 10진수 입력

리턴 레지스터가 바뀌는 사칙연산

mul para
- 1 byte : ax = al * para
- 2 byte : dx:ax = ax * para
- 4 byte : edx:eax = eax * para

div para
- 1 byte : al = ax / para ah = ax % para
- 2 byte : ax = dx:ax / para dx = dx:ax % para
- 4 byte : eax = edx:eax / para edx = eax % para

- **산술 비교**
  - cmp para1, para2
    - para1-para2 연산 수행
    - eflag 레지스터 변경
    - 조건 분기 명령문과 연결되어 사용
- **분기 명령어**
  - JE/JNE label == !=
    - 같다면/같지 않다면 label로 점프
  - JG/JGE label > >=
    - 크다면/크거나 같으면 label로 점프
  - JL/JLE label < <=
    - 작다면/작거나 같으면 label로 점프
  - JMP label
    - label 로 무조건 점프

- **AND, OR, XOR, NOT 연산**
  - and/or/xor para1, para2 (para1=para1 연산 para2)
    - para1: 레지스터, 메모리
    - para2: 레지스터, 메모리, 상수값
    - 연산 결과는 모두 para1에 저장
    - para1, para2 모두 메모리 인 경우는 연산 불가
  - not para(para=not para, 0은 1로 1은 0으로 변경)
    - para: 연산대상으로 레지스터, 메모리

지정 수 만큼 레지스터 저장 값 시프트

- shr para1, para2 (오른쪽으로 비트 이동)
  - para1:작업 할 장소 (레지스터나 메모리)
  - para2: 오른쪽으로 이동할 비트 수
- shl para1, para2 (왼쪽으로 비트 이동)
  - para1:작업 할 장소 (레지스터나 메모리)
  - para2: 오른쪽으로 이동할 비트 수

- 반복 명령어
  - loop label
    - cx/ecx 값을 1 감소시키고, 만약 cx/ecx가 0이 아니라면 label로 점프
    - 만약 cx/ecx가 0이라면 점프 없이 다음 명령 수행
    - mov ecx, 반복횟수

```
라벨:
  <반복되는 내용>
loop 라벨
```

```
%include "io64.inc"

section .text
global CMAIN
CMAIN:
    mov rbp, rsp; for correct debug
    ;write your code here
    mov ax, 0
```

루프 전, ex, ecx값 셋팅

For 구문의 구현

- 예제

```
ax=0;
bx=1;
for(cx=10 ; 0<cx ; cx--)
{
      ax = ax + bx;
      bx ++;
}
print(ax)
```

```
%include "io64.inc"

section .text
global CMAIN
CMAIN:
    ;write your code here
    mov ax, 0
    mov bx, 1
    mov cx, 10
L1:
    add ax, bx
    inc bx
    loop L1

    PRINT_DEC 2, ax
    NEWLINE

    xor rax, rax
    ret
```

do-while 구문의 구현

- 예제

```
ax=0;
bx=0;
do
{
      ax = ax + bx
      bx++;
}while(bx<=10);
```

```
%include "io64.inc"

section .text
global CMAIN
CMAIN:
    mov rbp, rsp; for co
    ;write your code her
    mov ax, 0
    mov bx, 0
L1:
    add ax, bx
    inc bx
    cmp bx, 10
    jle L1

    PRINT_DEC 2, ax
    NEWLINE

    xor rax, rax
    ret
```

while 구문의 구현

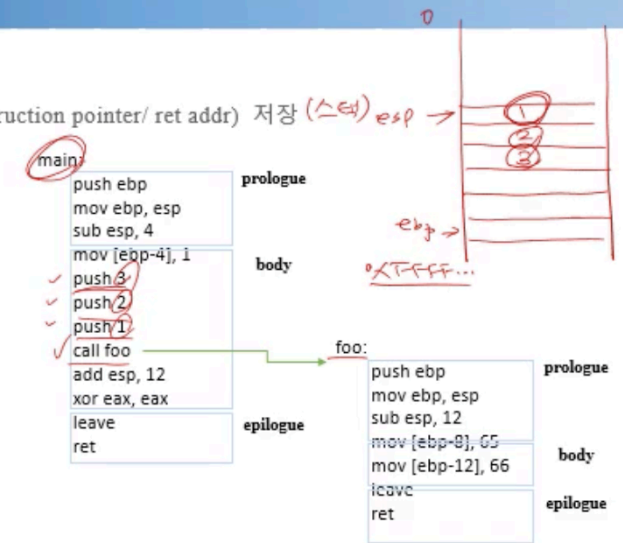
- 예제

```
ax=0;
bx=0;
while(bx<=10){
        ax = ax + bx
        bx++;
}
```

```
1   %include "io64.inc"
2
3   section .text
4   global CMAIN
5   CMAIN:
6       mov rbp, rsp; for co
7       ;write your code her
8       mov ax, 0
9       mov bx, 0
10  L1:
11      cmp bx, 10
12      jg L2
13      add ax,bx
14      inc bx
15      jmp L1
16  L2:
17      PRINT_DEC 2, ax
18      NEWLINE
19
20      xor rax, rax
21      ret
22
```

IA32 리마인딩

- 함수 호출전
  - 파라미터와 RIP (return instruction pointer/ ret addr) 저장
- 함수 프롤로그
  - 스택 프레임 생성
- 함수 바디
  - 함수 코드
- 함수 에필로그
  - 이전 스택 프레임 복원
  - 저장 ret addr로 점프



안드로이드와 달빅 바이트코드 (1)

source -> Dex (DVM에서 해석가능한 Dalivk ByteCode 내포) --(interpret)--> 기계어

레지스터는 기본 32-비트 사이즈, 두개의 연속 레지스터 쌍으로 64-비트 크기 값 표현

달빅 바이트 - 객체에 대한 타입이 정해져있음
- 변수타입
    - **I** : int , **J** : long, **Z** : boolean, **D** : double, **F** : float, **S** : short, **C** : char, **V** :void
- 패키지명 (**L패키지명**/../../Classname;)
- 배열타입
    - **[Type** --ex-> int Array = **[I**
    - 이중 배열 -> 대괄호 두개

명령어 규칙
- 32bit 데이터 처리시 -> 접미사 없음
- 64bit 데이터 처리시 -> -wide 접미사
- 타입 지정이 필요한 opCode는 뒤에 접미사로 타입을 표현한다.

“move-wide/from16, vAA, vBBBB”
-> move : Opcode
-> wide : 오퍼랜드 64bit?

-wide는 인덱싱??? (레지스터 a, a+1를 의미하는가?)

예를 들어 `move-wide/from16 vAA, vBBBB` 명령어에서

- move는 기본 연산을 나타내는 기본 opcode입니다(레지스터의 값 이동).
- wide는 너비가 64비트인 데이터에서 연산됨을 나타내는 이름 접미사입니다.
- from16`'은 16비트 레지스터 참조를 소스로 갖는 변형을 나타내는 opcode 접미사입니다.
- vAA는 대상 레지스터(연산에 나타남. 즉, 대상 인수가 항상 먼저 오도록 규칙이 적용됨)이고, `v0`~`v255` 범위 내에 있어야 합니다.
- vBBBB는 소스 레지스터이고 v0~v65535 범위 내에 있어야 합니다

안드로이드와 달빅 바이트코드 (2)

메소드 반환값
- move-result vAA
    - 리턴값을 레지스터 A에 저장
        - -wide 64bit데이터
        - -object src가 object Type

배열 조작
- new-array vA, vB, type@CCCC
    -
- filled-new-array {vC, vD, vE, vF, vG}, type@BBBB
    - vC, vD, vE, vF, vG의 값이 담긴 배열
- fill-array-data vAA, +BBBBBBBB
    - 생성하는건 아니고, new-array를 통해 만들어진 배열에 값을 채우는것 (+BBBBBBBB는 주소의 오프셋값)

인스턴스 생성
- new-instance vAA, type@BBBB
    - 대부분 class type의 인자가 들어옴
    - new String()
        - ex -> **new-instance V0, LJava/Lang/String**
            -> 패키지 이름은 L로 시작함

무조건 분기문
- goto +AA (최대 offset 8bit내)
- goto/16 +AAAA (최대 offset 16bit내)
- goto/32 +AAAAAAAA (최대 offset 32bit내)

조건 분기문
- if-test vA, vB, +CCCC
    - test 결과가 참이면 지정 주소(라벨)로 분기
    - test : eq, ne, lt, ge, gt, le이 있음

비교 연산
- cmp**kind** vAA, vBB, vCC
    - kind : cmpl-float (lt bias), cmpg-flaot (gt bias) .....
    - B와 C의 비교 결과를 A에 저장
        1) 레지스터 B == 레지스터 C→레지스터 A에 0 저장
        2) 레지스터 B > 레지스터 C→레지스터 A에 1 저장
        3) 레지스터 B < 레지스터 C→레지스터 A에 -1 저장

클래스 멤버 필드 값 읽기/쓰기
sadda i*instanceop* 추가추가추가추각바재ㅑ 겆배ㅑ ㅓ애ㅑ ㅁ저애ㅑ ~~

배열 요소 값 읽기/쓰기
arrayop vAA, vBB, vCC
aarraypo은 다음과 같음
- 읽기 : aget, aget-wide, aget-object, aget-boolean, aget-byte, aget-char, aget-short
    - 레지스터 vBB + vCC(*offset*)의 값을 vAA에 저장
- 쓰기 : aput, aput-wide, aput-object, aput-boolean, aput-byte, aput-char, aput-short
    - 레지스터 vBB + vCC(*offset*)에 vAA의 값을 저장

정적 필드 값 읽기/쓰기
: static 변수는 한메모리로 저장해서 새로운 instance가 만들어져도 기존 메모리로 접근해서 값이 똑같을 수 밖에 없음.
- **s**staticop vAA, field@BBBB(정적 필드)

가상(Virtual) 메소드 호출
- invoke-virtual {**vC**, vD, vE, vF, vG}, meth@BBBB
- invoke-virtual/range {**vCCCC** .. vNNNN}, meth@BBBB
    - 지정된 인자 값을 활용하여 인스턴스 메소드 호출

- 첫번째 인자는 반드시 인스턴스 레퍼런스 = 오브젝트를 가르키는 인덱스 값 (참조 주소)
- 호출 시, Virtual method table 내 메소드 접근

*가상의 메서드를 사용하는 이유?

정적 (Static) 메소드 호출

- invoke-static {vC, vD, vE, vF, vG}, meth@BBBB
- invoke-static/range {vCCCC .. vNNNN}, meth@BBBB
  - 지정된 인자 값을 활용하여 정적 메소드 호출
  - “this ” 인자가 첫번쨰 인자로 지정되지 않음 (staticop와 같은 원리!)

Private 메소드 혹은 Constructor 메소드 호출 (일반적인 호출)

- invoke-direct {this변수, 인자 인자 인자}
- invoke-direct/range {this변수, 인자 인자 인자}
  - 비 static, private 메서드 호출… 일반적일걸?

Super 메소드 호출

상속받은 부모 메서드 호출

Interface 메소드 호출

달빅 바이트코드 맨 밑에 예제문제 나올 거같음

알면 좋은 것들?

.catch java/io/IOException from 1300 to 1306 using 130a
- 1300~1306라벨에 대해서 IOException 예외는 별도 처리 (try)

move-exception vAA
- A라는 익셉션 래퍼런스 벨류 Catch. 이후 코드에서 catch(vAA){ 가 시작됨

(1) EAX 레지스터에 십진수 3, EBX 레지스터에 십진수 5를 저장하고,
두 레지스터의 값을 교환(SWAP)하여 각 레지스터에 다시 저장하는 코드를 작성하시오.
(mov 명령어 만을 이용, 레지스터는 모두 사용 가능)

```
    mov eax, 3
    mov ebx, 5

    mov edx, eax
    mov eax, ebx
```

(3) EAX 레지스터에 이미 저장되어진 값이 십진수 10과 동일한지 비교하여
10과 같을 때 EBX 레지스터에 십진수 30을 저장하고,
10과 다를 때 EBX 레지스터에 십진수 50을 저장하는 어셈블리 코드를 작성하시오

```
    cmp eax, 10
    jne NOT_EQUAL_TEN
    je EQUAL_TEN
NOT_EQUAL_TEN:
    mov ebx, 50
    jmp END
EQUAL_TEN:
    mov ebx, 30
    jmp END
END:
    PRINT_DEC 4, ebx
    xor rax, rax
    ret
```

(4) AX 레지스터에 십진수 10을 저장하고, BL 레지스터에 십진수 3을 저장한 뒤,
AX에 저장된 10을 BL 내 3으로 나눈 몫과 나머지를 DL레지스터와 DH레지스터에 각각 저장하는
어셈블리어 코드를 작성하시오

```
mov AX, 10
mov BL, 3
div bx
;AX = AX / BL
;DX = AX % BL
PRINT_DEC 2, AL
NEWLINE
PRINT_DEC 2, AH  tODOTODOTODO
```

(5) sub 함수 아래의 의사코드와 같이 정이돼 있을 때,
sub(1, 10)을 호출하는 어셈블리 코드를 완성하시오 (오른쪽 ??? 빈칸 채우기)

C-style pseudo code

```
int main(){
    sub(1,10);
}

int sub(int a, int b){
    return b-a;
}
```

Assembly Code

```
    ??? -> push 10
    ??? -> push 1
    call sub
    mov esp, sbp
    pop ebp
    ret

sub:
    … … …
    … … …
```

(6) nop 명령어에 대해 설명하시오

프로그램이 작동중 NOP 명령어를 만났을 시
해당 명령어를 수행하지 않고 무시하고 다음 명령어로 넘어가게 된다.

(7) 다음의 **leave** 명령어 코드의 수행결과와 동일한 연산이 이러지도록 하는 코드를 구현하시오

```
mov ESP, EBP
pop EBP
```

(8) 다음의 pop 명령어 코드의 수행결과와 동일한 연산이 이러지도록 하는 코드를 구현하시오

```
pop EIP
jmp EIP
```

(9) 아래와 같이 Foo라는 함수(프로시저)가 정의 되어있을때,
sub esp, 12 어셈블리 코드가 의미하는 바가 무엇인지 설명하시오
(힌트 : 스택프레임과 연관지어 설명)

```
Foo:
    push ebp
    mov ebp, esp
    sub esp, 12   <<< 여기
    mov [ebp-8], 65
    mov [ebp-12], 66
    leave
    ret
```

foo 함수가 실행됨에 따라
foo의 스택프레임을 만들어야한다.

esp를 12만큼 뺌에 따라
foo함수에서 12만큼의 로컬 변수를
할당 할 수 있다.

(10) 아래와 같이 Foo라는 함수(프로시저)가 정의 되어 있을 때,
push ebp라는 어셈블리 코드가 함수에서 처음 실행되는 이유는 무엇인지 설명하시오

```
Foo:
    push ebp      <<< 여기
    mov ebp, esp
    sub esp, 12
    mov [ebp-8], 65
    mov [ebp-12], 66
    leave
    ret
```

스택프레임을 만들기 위해서이다.
또한 함수가 끝났을때 pop ebp를 통해
이전 코드로 복귀 할 수 있다.

(11) 리버싱을 통해 add eax, 1과 sub ebx, 2의 코드가 실행 될 수 있도록 하는 방법 3가지를 각각 설명하시오 (디버깅 환경에서 조작)

```
   cmp eax, ebx
   je label1
   add eax, 1
   sub ebx, 2
label1:
   ret
```

(1) cmp 이후 ZF를 0로 수정한다.
(2) EIP를 add eax, 1 주소로 수정한다.
(3) je label1을 NOP으로 인라인 코드 패치 한다.

Android Code

```java
private void swap(int array[], int i){
   int temp = array[i];
   array[i] = array[i+1];
   array[i+1] = temp;
}
```

Smali Code

```
.method private swap ([II)V
;.limit registers 5 내 맘대러 주석처리 해버림
; parameter 1 (int array) = v1
; parameter 2 (int) = v2
; temp = v3
    aget v3, v1 , v2 ; temp = array[i]
    add-int/lit8 v4, v2 ; v4 = i+1
    aget v5, v1, v4 ; v5= array[i+1]
    aput v5, v1, v2 ; array[i] = v5 = array[i+1]
    aput v3, v1, v4 ; array[i+1] = temp
    return-void
.end method
```

Android Code

```java
private void sort( int array[] ){
    boolean swapped;
    do {
        swapped = false;
        for (int i = 0; i < array.length - 1; ++i){
            if (array[i] > array[i+1]){
                swap ( array, i )
                swapped = true;
            }
        }
    } while (swapped);
}
```

Smali Code

```
.method private sort([I)V
; this: v6 (Ltest10;)
; parameter[0] : v7 ([I)
          const/4 v5, 1              ; v5=1
          const/4 v4, 0              ; v4=0
12c4 :    move v0, v4                ; v0=v4
          move v1, v4                ; v1=v4
12c8 :    array-length v2, v7        ; v2=v7.length
          sub-int/2addr v2, v5       ; v2=v2-v5
          if-ge v0, v2, 12ee         ; if(v0>=v2) -> 12ee
          aget v2, v7, v0            ; v2=v7[v0]
          add-int/lit8 v3, v0, 1     ; v3=v0+1
          aget v3, v7, v3            ; v3=v7[v3]
          if-le v2, v3, 12e8         ; if(v2<=v3) -> 12e8
          invoke-direct {v6, v7, v0), Test10/swap
          move v1, v5                ; v1=v5
12e8 :    add-int/lit v0, v0, 1      ; v0=v0+1
          goto 12c8                  ; -> 12c8
12ee :    if-nez v1, 12c4            ; if(v1 != 0) -> 12c4
          return-void
```

Android Code

```
int array[] = {
4, 7, 1, 8, 10, 2, 1, 5
};

instance.sort(array);
```

Smali Code

```
const/16            v1, 8
new-array           v1, v1, [I ; v1을 [v1]개의 int array
fill-array-data     v1, 1288 ; v1에 1288을 넣음
invoke-direct       {v0, v1}, Test10/sort
......
1288 :    data-array ; 배열 데이터의 시작
                0x04, 0x00, 0x00, 0x00 ; #0
                0x07, 0x00, 0x00, 0x00 ; #1
                0x01, 0x00, 0x00, 0x00 ; #2
                0x08, 0x00, 0x00, 0x00 ; #3
                0x0A, 0x00, 0x00, 0x00 ; #4
                0x02, 0x00, 0x00, 0x00 ; #5
                0x01, 0x00, 0x00, 0x00 ; #6
                0x05, 0x00, 0x00, 0x00 ; #7
          end data-array ; 배열 데이터의 끝
```

const - 상수(숫자) 값을 이동함(값 : 32비트)
const/4 - “ (값 : 4비트 int)
const/16 - “ (값 : 16비트 int)

new-array vA, vB, type@CCCC
- A에 B개의 C타입 어레이를 만듭니다.

fill-array-data vA, vB
- 배열 A에 B(Data-Array 분기점)를 넣음

안드로이드 달빅 기반의 실행파일 오브젝트인 classes.dex 파일은 상수 풀 공유 (Shared Constant Pool)를 이용하여 여러 자바의 클래스 파일(.class 파일)을 하나로 통합 관리 한다. 상수 풀 공유 방법과 그 장점을 설명하시오

방법 : 플라이웨이 패턴 방식으로 동작한다

장점 : 상수 풀 공유를 통해서 동일 요소 반복 저장을 방지 하여,
메모리를 효율 적으로 사용할 수 있다.

클래스(Class)와 오브젝트(Object)의 차이를 기술하시오

클래스는 객체를 만들어 내기 위한 **설계도** 혹은 틀이고,
오브젝트는 틀(클래스)로 만든 실제 구현체이다.