# Project 1: Simple MIPS assembler

## Due 11:59 PM, March 26st(Fri.)

**TA: SeongTae Bang(st.bang@dgist.ac.kr), MinHo Kim (mhkim@dgist.ac.kr)**

## 1. Introduction

이 과제의 목표는 **MIPS(Big-endian) 어셈블리 코드를 바이너리 코드로 변환하는 MIPS ISA assembler를 구현해보는 것**이다. 이 과제를 통해 만들어야 하는 assembler는 linking 등의 복잡한 일을 수행하지 않는 간단한 구조이기 때문에, 최근의 컴파일러가 사용하고 있는 symbol, relocation table 등은 고려하지 않아도 된다.

## 2. Instruction Set

A. MIPS instruction set

이 assembler가 지원해야 하는 명령어는 다음과 같다.

| | | | | | | |
|---|---|---|---|---|---|---|
| ADDIU | ADDU | AND | ANDI | BEQ | BNE | J |
| JAL | JR | LUI | LW | **LA*** | NOR | OR |
| ORI | SLTIU | SLTU | SLL | SRL | SW | SUBU |
| LB | SB | | | | | |

- LB와 SB를 제외한 load와 store 명령어는 4B word만 지원하면 된다.
- Assembler는 10진수 및 16진수의 숫자를 모두 지원하여야 한다. 16진수는 숫자 앞에 (0x)가 붙어 10진수와 구별할 수 있다.
- 레지스터의 이름은 언제나 "$n"($0 \leq n \leq 31$, n은 정수)와 같은 형태이다.
- * LA(Load Address) 명령어는 실제 명령어가 아닌 의사 명령어(pseudo instruction)이다. LA는 그대로 바이너리 코드로 변환되지 않고, 하나 또는 두 개의 다른 어셈블리 명령어로 바뀐다. 다음은 그 예시이다.

➔ LA 명령어는 주소에 따라 lui와 ori instruction으로 변환된다.

```
lui $register, 상위 16bit address
```

```
ori $register, 하위 16bit address
```

➔ 0x1000 0000 주소를 load할 경우, 하위 16bit address가 0x0000 이므로 ori instruction을 사용하지 않는다. 이는 아래와 같이 동작한다.

```
lui $2, 0x1000
```

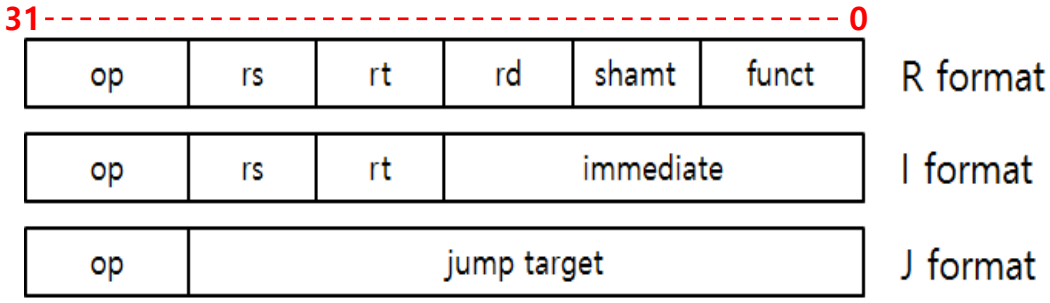
➔ 반면에 0x1000 0004 주소를 load할 때, 하위 16bit address가 0x0004이다. 이는 0

이 아닌 숫자이므로 아래와 같이 ori를 포함한 두 instruction으로 동작한다.

```
lui $2, 0x1000
ori $2, $2, 0x0004
```

B. Instruction Formats

instruction은 다음과 같이 세 가지 종류로 분류된다




- R format : 산술 명령어 형식

  e.g. ADDU, AND, JR, NOR, OR, SLTU, SLL, SRL, SUBU

| Field | Description |
|---|---|
| op | 명령어가 실행할 연산의 종류 |
| rs | 첫 번째 근원지(source) 피연산자 레지스터 |
| rt | 두 번째 근원지(source) 피연산자 레지스터 |
| rd | 목적지(destination) 레지스터. 연산 결과가 저장 됨 |
| shamt | 자리이동(shift) 량. 사용하지 않을 경우 0으로 채움 |
| funct | op필드에서 연산의 종류를 표시하고, funct 필드에서 어떤 연산인지 구체적으로 지정 |

- I format : 전송, 조건 분기, 즉치 명령어 형식

  e.g. ADDIU, ANDI, BEQ, BNE, LUI, LW, ORI, SLTIU, SW, LB, SB

| Field | Description |
|---|---|
| op | 명령어가 실행할 연산의 종류 |
| rs | 근원지(source) 피연산자 레지스터 |
| rt | 목적지(destination) 레지스터 |
| immediate | 상수(constant) 또는 주소(address) |

- J format : 무조건 분기(jump) 명령어 형식

e.g. J, JAL

| Field | Description |
|---|---|
| op | 명령어가 실행할 연산의 종류 |
| jump target | word단위의 분기 주소 |

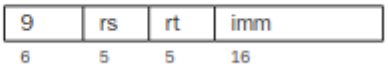
C. Instruction Details

※ Instruction의 정확한 동작은 이번 과제에서는 구현하지 않음.

- ADDIU(I형식): $rs와 sign-extended된 imm(immediate field)의 합을 $rt에 저장

**Addition immediate (without overflow)**

addiu rt, rs, imm




- ADDU(R형식): $rs와 $rt의 합을 $rd에 저장

**Addition (without overflow)**

addu rd, rs, rt


- AND(R형식): $rs와 $rt의 logical AND 연산 결과를 $rd에 저장

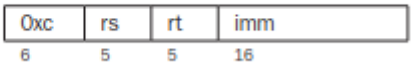
**AND**

and rd, rs, rt


- ANDI(I형식): $rs와 zero-extended된 imm(immediate field)의 logical AND연산 결과를 $rt에 저장

**AND immediate**

andi rt, rs, imm




- BEQ(I형식): $rs와 $rt의 값이 동일 할 경우, **PC+ offset * 4(instruction size)=BEQ의 address + 4 + offset * 4(instruction size)** 의 주소로 분기; e.g. PC=0x400004, offset=0x1000일 경우, 0x400004 + 0x1000 * 4 = 0x404004 또는 BEQ의 address인 0x400000 + 0x1000 * 4 + 4 = 0x404004로 계산하여 분기하면 된다.

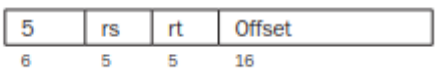
**Branch on equal**

beq rs, rt, label


- BNE(I형식): $rs와 $rt의 값이 다를 경우, **PC+ offset * 4(instruction size) = BEQ의 address + 4 + offset * 4(instruction size)**의 주소로 분기

**Branch on not equal**

bne rs, rt, label



- J(J형식): target으로 무조건 분기(jump), target은 jump 대상(address)를 4로 나눈 값으로 지정된다.

**Jump**

`j target`


- JAL(J형식): 다음 실행할 명령어의 주소를 $ra 레지스터에 저장하고, target으로 무조건 분기(jump) , target은 jump 대상(address)를 4로 나눈 값으로 지정된다.

**Jump and link**

`jal target`


- JR(R형식): $rs에 들어 있는 명령어의 주소로 무조건 분기(jump)

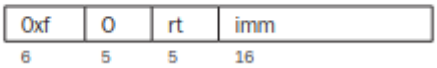
**Jump register**

`jr rs`


- LUI(I형식): imm를 $rt의 상위16bits(upper halfword)에 채우고 하위 16bits(lower bits)는 0으로 채움

**Load upper immediate**

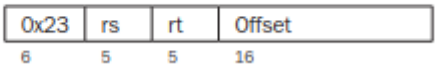
`lui rt, imm`




- LW(I형식): 해당하는 주소($rs 또는 $rs+offset)의 word(32bit) 크기 data를 $rt로 로드(load). 실제 사용될 때는 `lw rt, offset(rs)` 와 같은 형태로 사용됨.

**Load word**

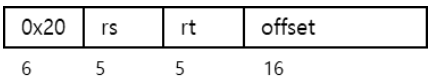
`lw rt, address`




- LB(I형식): 해당하는 주소($rs 또는 $rs+offset)의 byte(8bit) 크기 data를 $rt로 로드(load). 실제 사용될 때는 `lb rt, offset(rs)` 와 같은 형태로 사용됨.

**Load byte**

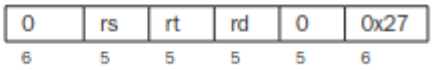
`lb rt, address`




- NOR(R형식): $rs와 $rt의 논리적 NOR 연산 결과를 $rd에 저장

**NOR**

`nor rd, rs, rt`




- OR(R형식): $rs와 $rt의 논리적 OR 연산 결과를 $rd에 저장

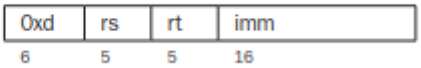
**OR**

`or rd, rs, rt`

- ORI(I형식): $rs와 zero-extended imm의 논리적 OR 연산 결과를 $rt에 저장

**OR immediate**

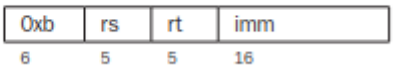
```
ori rt, rs, imm
```




- SLTIU(I형식): $rs가 sign-extended imm보다 작으면 $rt를 1로, 그렇지 않으면 0으로 설정

**Set less than unsigned immediate**

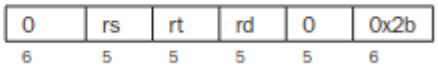
```
sltiu rt, rs, imm
```




- SLTU(R형식): $rs가 $rt보다 작으면 $rd를 1로, 그렇지 않으면 0으로 설정

**Set less than unsigned**

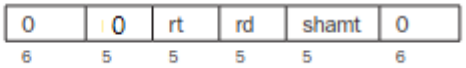
```
sltu rd, rs, rt
```




- SLL(R형식): $rt를 shamt만큼 left-shift 연산한 결과를 $rd에 저장

**Shift left logical**

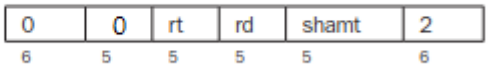
```
sll rd, rt, shamt
```




- SRL(R형식): $rt를 shamt만큼 right-shift 연산한 결과를 $rd에 저장

**Shift right logical**

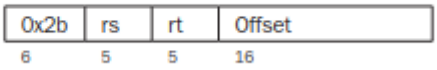
```
srl rd, rt, shamt
```




- SW(I형식): word(32bit) 크기의 $rt의 data를 해당하는 주소($rs 또는 $rs+offset)로 저장. 실제 사용될 때는 `sw rt, offset(rs)` 와 같은 형태로 사용됨.

**Store word**

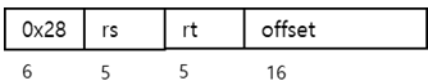
```
sw rt, address
```




- SB(I형식): byte(8bit) 크기의 $rt의 data를 해당하는 주소($rs 또는 $rs+offset)로 저장. 실제 사용될 때는 `sb rt, offset(rs)` 와 같은 형태로 사용됨.

**Store byte**

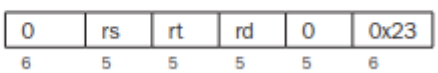
```
sb rt, address
```




- SUBU(R형식): $rs에서 $rt를 뺀 값을 $rd에 저장

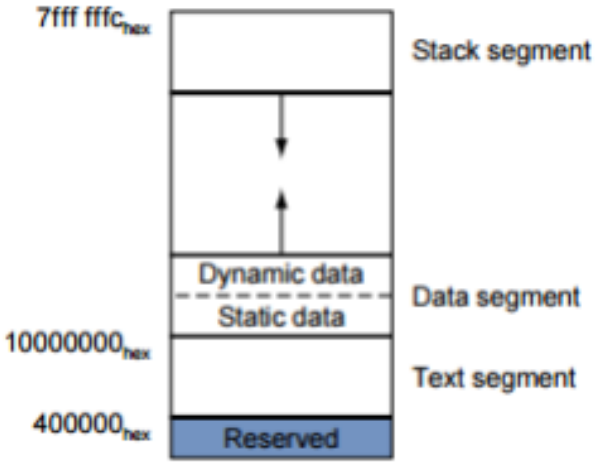
**Subtract (without overflow)**

```
subu rd, rs, rt
```



3. **Memory layout**

메모리에는 아래와 같이 영역(segment)이 나뉘며, 각 영역은 어셈블리 코드 상에서 지시자(directive)로 나타낼 수 있다. 아래 각 영역에 대한 주소는 실제 메모리의 주소(physical address)와는 상이할 수 있다.




**※ Directives**

A. .text

- 사용자의 텍스트 영역(user text segment)을 가리킨다. 명령어들(instructions)이 여기에 위치한다.
- 이 영역은 항상 0x00400000번지에서 시작한다.

B. .data

- 데이터 영역(data segment)을 가리키며, 전역 변수(static data) 또는 동적으로 할당 되는 변수(dynamic data)가 위치하는 영역이다.
- 이 영역은 항상 0x10000000번지에서 시작한다.

C. .word

- Data segment 내에서 32비트 데이터(word)가 위치하는 영역으로, C에서의 int 정수 배열 또는 변수로 생각하여도 된다.

## 4. Programming Guideline

A. Environment

제출된 코드는 Ubuntu 16.04 (Linux), 또는 Ubuntu 18.04, Windows Subsystem for Linux 환경에서 테스트될 것이다. 위 환경 중 하나에서 코드가 실행 가능하면 채점 가능하다. 이외의 OS에서 코딩하는 경우 OS API 등으로 인해 컴파일이 불가능한 경

우가 있을 수 있으니, 가능하다면 위 환경에서 테스트를 해보는 것을 권장한다.

B. Program Language

프로그래밍 언어는 C, C++이 허용된다. 이 언어들 중에 어떤 것을 사용하여도 좋다.

- C 혹은 C++의 경우, gcc 4.8, g++ 4.8 버전 이상의 컴파일러를 이용하여 채점을 진행하게 된다.
- C 혹은 C++의 경우 Microsoft Visual Studio를 사용하여야만 컴파일이 되는 경우, **컴파일이 불가능한 것으로 간주**한다. 또한, Microsoft Visual Studio 프로젝트 파일을 제출하는 것은 인정되지 않는다.

C. Input format

다음과 같은 형태의 sample file이 주어진다. LMS에서 파일 형태로 다운로드한 경우만 인정하며, 텍스트를 복사하여 붙여 넣는 등의 방법으로 생성된 파일은 sample file로 인정되지 않는다.

```
1           .data
2  array:   .word   3
3           .word   123
4           .word   4346
5  array2:  .word   0x12345678
6           .word   0xFFFFFFFF
7           .text
8  main:
9           addiu   $2, $0, 1024
10          addu    $3, $2, $2
11          or      $4, $3, $2
12          addiu   $5, $0, 1234
13          sll     $6, $5, 16
14          addiu   $7, $6, 9999
15          subu    $8, $7, $2
16          nor     $9, $4, $3
17          ori     $10, $2, 255
18          srl     $11, $6, 5
19          srl     $12, $6, 4
20          la      $4, array2
21          lb      $2, 1($4)
22          sb      $2, 6($4)
23          and     $13, $11, $5
24          andi    $14, $4, 100
25          subu    $15, $0, $10
26          lui     $17, 100
27          addiu   $2, $0, 0xa
```

Sample1

```
1           .data
2  var:   .word   5
3           .text
4  main:
5      la $8, var
6      lw $9, 0($8)
7      addu $2, $0, $9
8      jal sum
9      j exit
10
11 sum: sltiu $1, $2, 1
12     bne $1, $0, sum_exit
13     addu $3, $3, $2
14     addiu $2, $2, -1
15     j sum
16 sum_exit:
17     addu $4, $3, $0
18     jr $31
19 exit:
```

Sample2

Sample1의 첫 번째 열에 있는 `array:`, `array2:`, `main:`은 라벨을 의미하며 이는 해당 줄 또는 바로 다음의 인스트럭션 또는 데이터의 주소를 가리킨다. 예를 들어, 19번째 줄의 `la` 명령어의 2번째 인자인 `array2`의 경우 5번째 줄의 word의 주소를 가

리킨다. Sample2의 19번째 줄에 있는 `exit:`의 경우 다음에 아무런 인스트럭션이 없기 때문에 그 전에 있는 인스트럭션의 다음 주소(PC+4)를 가리키도록 한다. 이는 더 이상 실행할 인스트럭션이 없기 때문에 프로그램이 종료됨을 의미한다. `.data`, `.word`, `.text`는 directives를 의미한다. 1번째 줄의 `.data`는 이 아래에 적힌 것이 데이터라는 것을 의미하고, 주소의 경우 데이터의 시작 위치인 0x10000000부터 시작하게 된다. `.word`는 하나의 워드가 그 다음의 값을 가졌음을 의미한다. `.text` 부터는 명령어(instruction)가 시작된다.

D. Output format

프로그램의 출력으로 저장할 바이너리 파일의 형식은 다음과 같다.

```
<text section size>
<data section size>
<instruction 1>
...
<instruction n>
<value 1>
...
<value m>
```

처음 두 줄은 각각 text section, data section의 byte size를 나타내는 32bit 정수가 와야 한다. 각 명령어와 데이터들은 4Byte단위로 저장이 된다. Text section와 data section이 동일하게 25개 일 경우 크기는 4Byte*25로 100Byte로 첫번째 줄과 두번째 줄에 0x64의 값이 쓰여야 한다. 이후 나오는 instruction들은 한 줄마다 text section에 있는 32bit instruction들이 들어간다. 따라오는 value 들은 data section에 있는 32bit 정수 값이 와야 한다. 첫번째 data section가 Var: .word 5일 경우 value1에는 5에 대응하는 값을 나타내야 한다.

위와 같은 형식을 갖춘 바이너리 데이터를 16진수로 변환하여 text 형식으로 출력하도록 한다. 예를 들어, "addiu $2, $0, 1024"를 binary로 나타내면 "00100100000000100000010000000000"이고, 이를 16진수로 나타내면 "0x24020400"이다. 이 "0x24020400" 형태로 나타내어진 instruction 값을 text 형식으로 출력하도록 한다. 이 값을 10진수인 604111872와 같은 형태로 출력하여서는 안된다. 다른 section의 데이터도 같은 방식으로 출력하도록 한다. 16진수 알파벳의 경우 소문자로 출력하도록 하며 16진수의 자리 수는 고정하지 않아야 한다. (e.g. 0x0000003a가 아닌 0x3a로 표기)

E. Execution command

완성된 프로그램은 리눅스 콘솔창에서 다음 명령어 중 하나로 실행 되어야 한다.

`$ ./runfile <assembly file>`

위 명령어를 통해 assembly file (*.s) 을 input 으로 받고, input 파일을 바이너리 파일로 변환한 object file (*.o)을 output 파일로 출력하여야 한다.
e.g. sample.s (어셈블리 파일) -> sample.o (바이너리 파일)

## 5. Report

자신이 수행한 과제에 대한 간략한 설명을 기재하고, 과제의 **컴파일/실행 방법, 환경**이 들어있는 보고서를 작성하여 소스 파일과 함께 제출하도록 한다. 보고서는 ".pdf" 형식으로 변환하여 제출한다. 보고서 내에 소스 코드를 기재할 필요는 없다.

## 6. Submission

**프로젝트를 통해 작성한 소스 파일(.c, .cpp, .h) 및 보고서를 zip 형식으로 압축**하여 반드시 두 조교 **모두**에게 이메일로 제출하도록 한다. 제출한 형식이 위와 다를 경우, 감점의 요인이 되므로 주의해야 한다.

E-mail: mhkim@dgist.ac.kr, st.bang@dgist.ac.kr

**압축 파일의 이름은 학번_이름.zip 형식으로 양식에 맞게 하여 제출한다.**

e.g. 201711999 김철수 학생의 경우, **201711999_김철수.zip** 형식으로 제출.

## 7. Notice

A. Due date: **23:59, March 26st(Fri.)**

B. Penalty

- Late due

| Case | Penalty |
|---|---|
| 1일 지연 | 점수 10% 감점 |
| 2일 지연 | 점수 30% 감점 |
| 3일 지연 | 점수 50% 감점 |
| 4일 이상 지연 | 0점 처리 |

- Cheating(Peer code, senior code, internet code) 적발 시 모든 프로그래밍 과제 0점 처리 및 최종 학점에서 D-처리.
- Compile 혹은 실행이 안 될 경우 조교가 학생에게 연락하여 코드 수정 요청. 마지막 Late due까지 compile이 안 될 경우 0점 처리.

C. TA Contact & Office Hour

TA에게 질문 사항 또는 기타 용건이 있다면, 두 명의 TA 모두에게 질문 메일로 문의할 것. TA를 직접 만나서 이야기하고 싶다면, 메일로 미리 약속을 잡을 것.

- TA 방성태: st.bang@dgist.ac.kr
- TA 김민호: mhkim@dgist.ac.kr