Computer Organization Assignment#1

MIPS ISS(Instruction Set Simulator)

Due : 23:59, Monday, October 8, 2012

I. Introduction

ISS를 사용하여 MIPS instruction을 익숙하게 함이 목적이다. ISS를 사용하면 각 instruction이 어떤 동작을 하며 어떻게 메모리에 접근하여 프로그램을 수행하는지를 눈으로 확인할 수 있을 뿐만 아니라, 32개의 GPR(General Purpose Register)의 역할을 확인하고 직접 컨트롤이 가능하다. 크게 3가지 과제를 수행하게 되는데 첫째로, MIPS ISS중 하나인 SPIM의 사용법을 익히고, 둘째로, 35개의 instruction에 대해 그 동작을 익힌다. 셋째로, 공부한 instruction을 바탕으로 직접 assembly coding을하고 ISS를 통하여 simulation 해 본다. ISS가 무엇인지, assembly language는 무엇인지에 대한 기본적인 개념은 본 문서 맨 뒤의 APPENDIX A에 첨부하였다.

II. Theory

1. SPIM의 사용 법

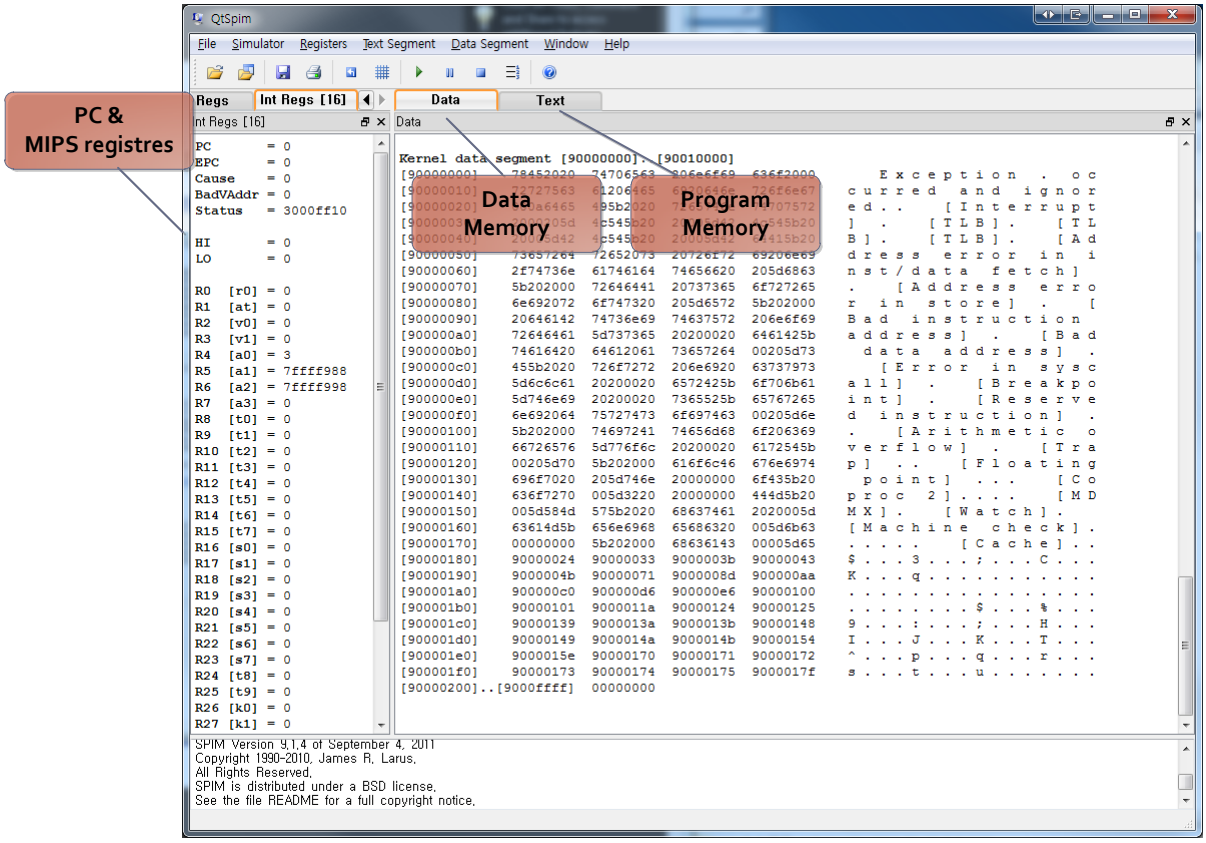


Figure 1 SPIM window layout

여타 ISS와 달리 SPIM은 단순히 instruction을 수행해 볼 수 있을 뿐만 아니라 console을 통해 가상으로 수행결과를 출력한다든지(printf)의 'systemcall'명령이 존재한다. 단순히 processor만 있고 OS가 없다면 프로세서가 수행 한 instruction의 결과를 눈으로 확인하기 어려운데, SPIM에서는 몇 가지 자체 systemcall 을 사용하여 여러 가지 기능을 구현할 수 있다. 뿐만 아니라 가상적으로 Memory map을 구성하고 있기 때문에 stack, data sement 등의 분리 된 memory 영역에 접근하고 프로그램 수행에 이용할 수 있다. 이러한 systemcall은 SPIM에서만 유효한 것이기 때문에 이를 이용하기 위한 몇 가지 약속을 알아야 한다.

SPIM example 1: add two numbers

```
# $t2 - used to hold the sum of the $t0 and $t1.
# $v0 - 이 레지스터는 어떤 system call을 수행할 것인지 명령을 받는 역할로 정해져 있다.
# $a0 - 이 레지스터는 특정 system call에서 input을 받는 경우 input parameter로 정해져 있다.
        .text # 여기서부터 code가 시작한다는 directive
main:                           # 'main' 이라는 label
        li $v0, 4               # read number into $v0
        la $a0, str1            # str1 label 주소로 $a0에 저장
        syscall                 # str1 print
        li $v0, 5               # read number into $v0
        syscall                 # make the syscall read_int
        move $t0, $v0           # move the number read into $t0
        li $v0, 4
        la $a0, str2
        syscall                 # str2 print
        li $v0, 5               # read second number into $v0
        syscall                 # make the syscall read_int
        move $t1, $v0           # move the number read into $t1
        add $t2, $t0, $t1
        li $v0, 4
        la $a0, str3
        syscall                 # str3 print
        move $a0, $t2           # move the number to print into $a0
        li $v0, 1               # load syscall print_int into $v0
        syscall                 # print $a0 to console
        li $v0, 10              # syscall code 10 is for exit
        syscall
        .data
        str1 : .asciiz "Input 1 :"
        str2 : .asciiz "Input 2 :"
```

```
str3 : .asciiz "Result   :"
```

- #는 주석을 나타낸다.
- main : 은 label을 나타낸다. jump 등을 할 때 label을 명시하면 label로 jump 한다.

앞의 예제는 $t0 와 $t1에 들어갈 값을 사용자로부터 console창을 통해 입력 받아서 그 합을 출력하는 assemply code이다. 여기서 .text, syscall, move, li 등 실제 MIPS instruction에는 포함되지 않은 psudo code와 유사한 개념으로 assemply 코딩을 쉽게 하도록 돕고 있다. 이에 대한 설명을 아래에 보충한다.

아래의 Table1은 SPIM에서 사용되는 directives로서 assemply 코딩을 할 때 필요하다. 사용 예는 example1을 참조한다.

Table 1 SPIM directives

.data	start data segment
.ascii str	store the string str in memory without '\0'
.asciiz str	idem, with '\0'
.byte 3,4,16	store 3 byte values
.double 3.14, 2.72	store 2 doubles
.float 3.14, 2.72	store 2 floats
.word 3,4,16	store 3 32-bit quantities
.space 100	reserve 100 bytes
.text	start text segment

Table 2 syscall services

Service	명령	INPUT	동작 설명
print_int	$v0 = 1	$a0 = integer to print	prints $a0 to standard output
print_float	$v0 = 2	$f12 = float to print	prints $f12 to standard output
print_double	$v0 = 3	$f12 = double to print	prints $f12 to standard output
print_string	$v0 = 4	$a0 = address of first character	prints a character string to standard output
read_int	$v0 = 5		integer read from standard input placed in $v0
read_float	$v0 = 6		float read from standard input placed in $f0
read_double	$v0 = 7		double read from standard input placed in $f0
read_string	$v0 = 8	$a0 = address to place string, $a1 = max string length	reads standard input into address in $a0
sbrk	$v0 = 9	$a0 = number of bytes required	$v0= address of allocated memory Allocates memory from the heap
exit	$v0 = 10		
print_char	$v0 = 11	$a0 = character (low 8 bits)	
read_char	$v0 = 12		$v0 = character (no line feed) echoed
file_open	$v0 = 13	$a0 = full path (zero terminated string with no line feed),	$v0 = file descriptor

		$a1 = flags, $a2 = UNIX octal file mode (0644 for rw-r--r--)	
file_read	$v0 = 14	$a0 = file descriptor, $a1 = buffer address, $a2 = amount to read in bytes	$v0 = amount of data in buffer from file (-1 = error, 0 = end of file)
file_write	$v0 = 15	$a0 = file descriptor, $a1 = buffer address, $a2 = amount to write in bytes	$v0 = amount of data in buffer to file (-1 = error, 0 = end of file)
file_close	$v0 = 16	$a0 = file descriptor	

Table 3 SPIM instruction

Instruction	Meaning
li $t0, 44	register $t1의 값을 register $t0에 복사한다. 실제로 MIPS ISA에 존재하는 instruction이 아니므로 컴파일러가 자동으로 addi $t0, $0, 44 등으로 바꾼다.
la $v0, L1	'L1 :' 이라는 label이 있을 때, 그 label이 있는 주소를 v0에 넣는다.
move $t0, $t1	register $t1의 값을 register $t0에 복사한다. 컴파일러가 자동으로 addu $t0, $0, $t1 등으로 바꾼다.
lui $t0, 0x4355	Set the upper 16 bits or register $t0 to the value 0x4355.

III. Requirements

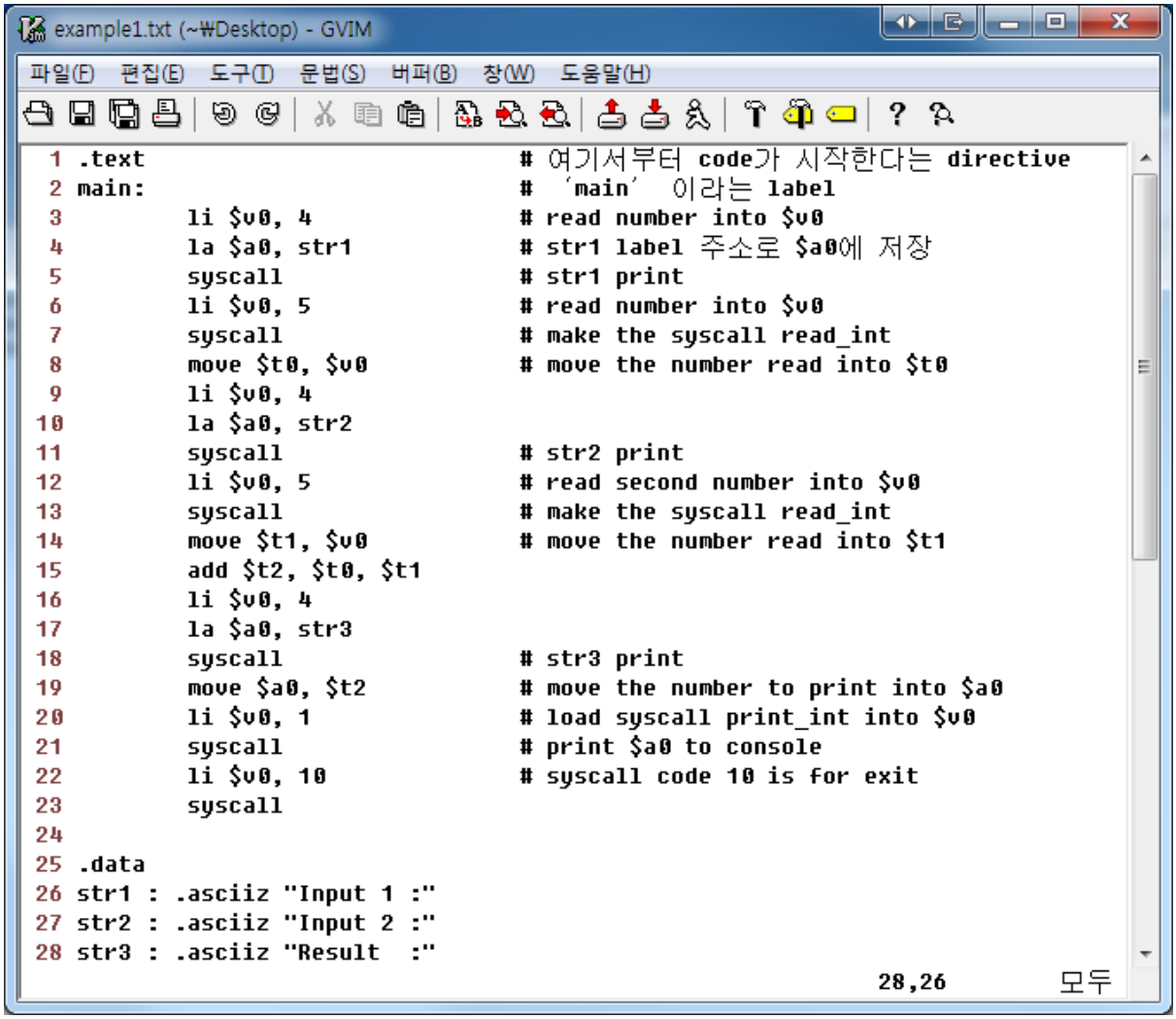
1. SPIM(MIPS ISS)의 사용 법 익히기
2. I-type, R-type, J-type 각각의 Instruction에 대한 format과 기능 익히기
3. assembly 코딩

보고서에 첨부해야 할 구체적인 내용은 IV. Implementation Details의 각 항목 마지막에 Requirement #1, #2, #3으로 표시하였다.

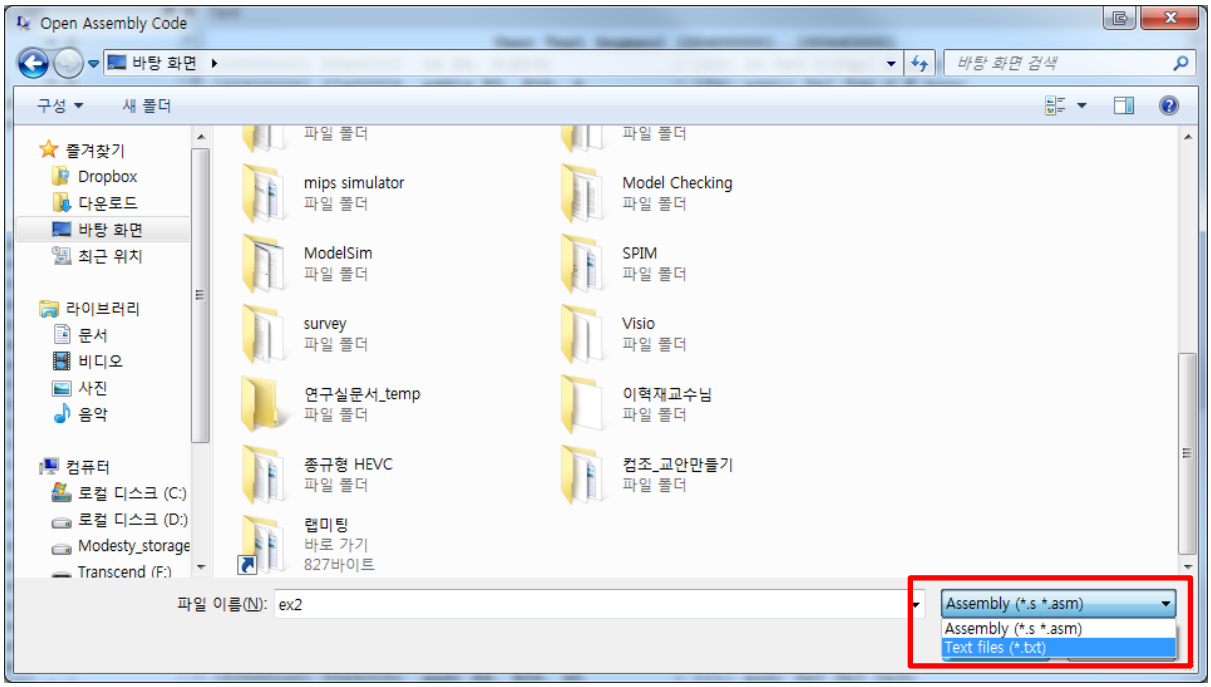
IV. Implementation Details

1. SPIM simulation

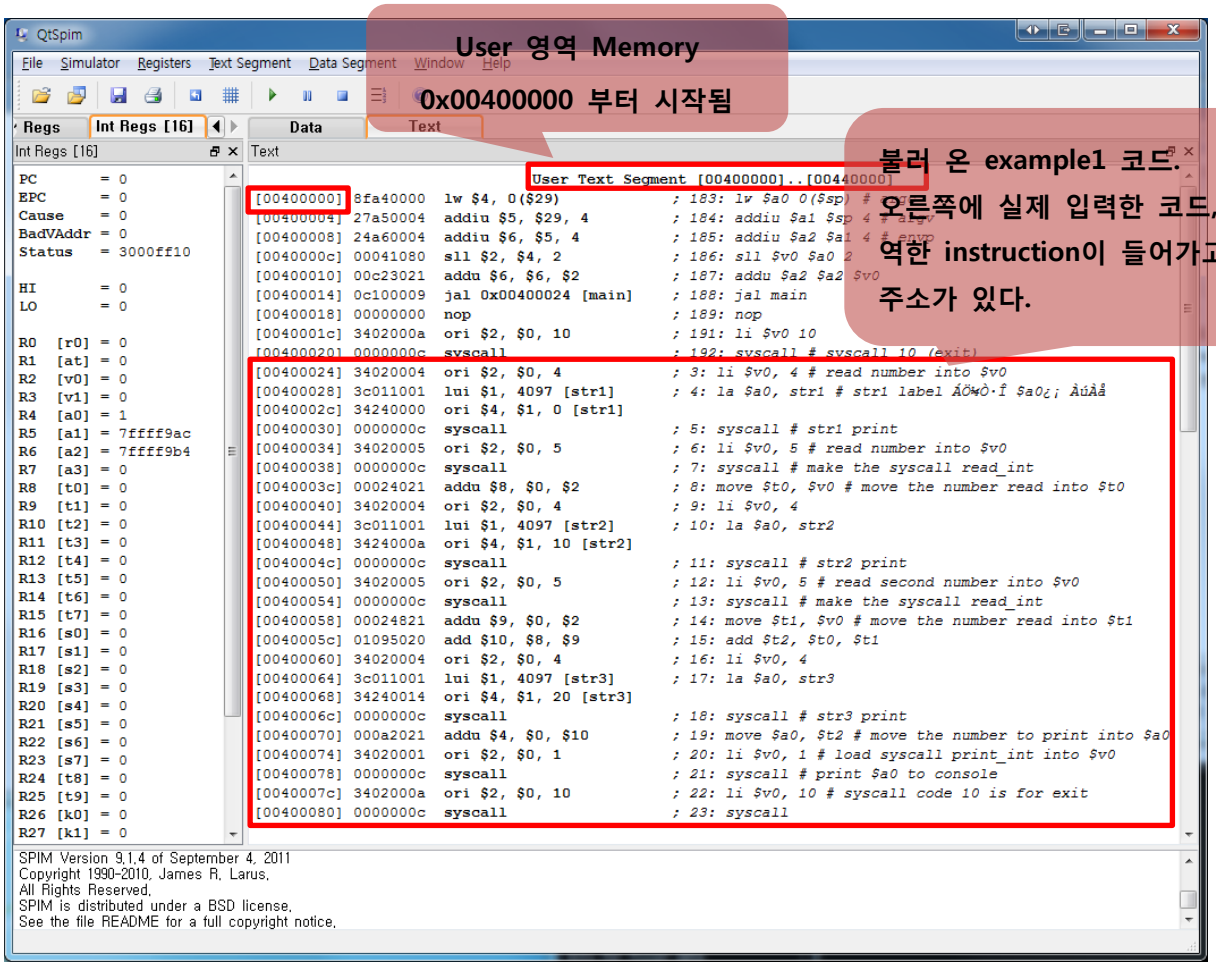
앞서 살펴 본 Example1을 SPIM을 통해 시뮬레이션 해 보자.



메모장이나 VIM 등 텍스트편집툴에 example1의 assemply code를 복사하고 저장한다.



SPIM의 File 메뉴 → LoadFile 을 클릭하여 파일 이름 옆의 종류를 Text files(*.txt)로 바꾸고 저장한 example1.txt를 불러온다.



파일을 열어보면 기대와 달리 우리가 불러온 assemply code만 나오는 것이 아니라 복잡하게 다른 instruction들이 보인다. 이는 OS가 없이도 console 등을 사용하여 가상으로 machine code를 테스트 하기 위한 내용이므로 일일이 이해할 필요는 없다.

또한 메모리 영역이 자동으로 할당된다. Figure5의 Memory map에 나타난 것 처럼 .text로 표시한 0x00400000 주소부터 코드가 저장되고, stack은 0x7fffffff에서부터 아래로 자라는 형식이 된다.

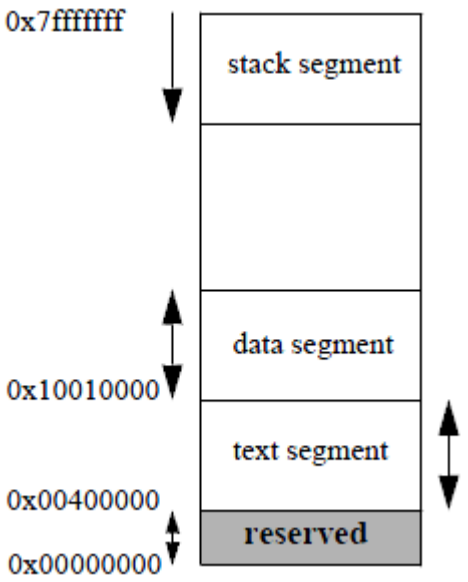
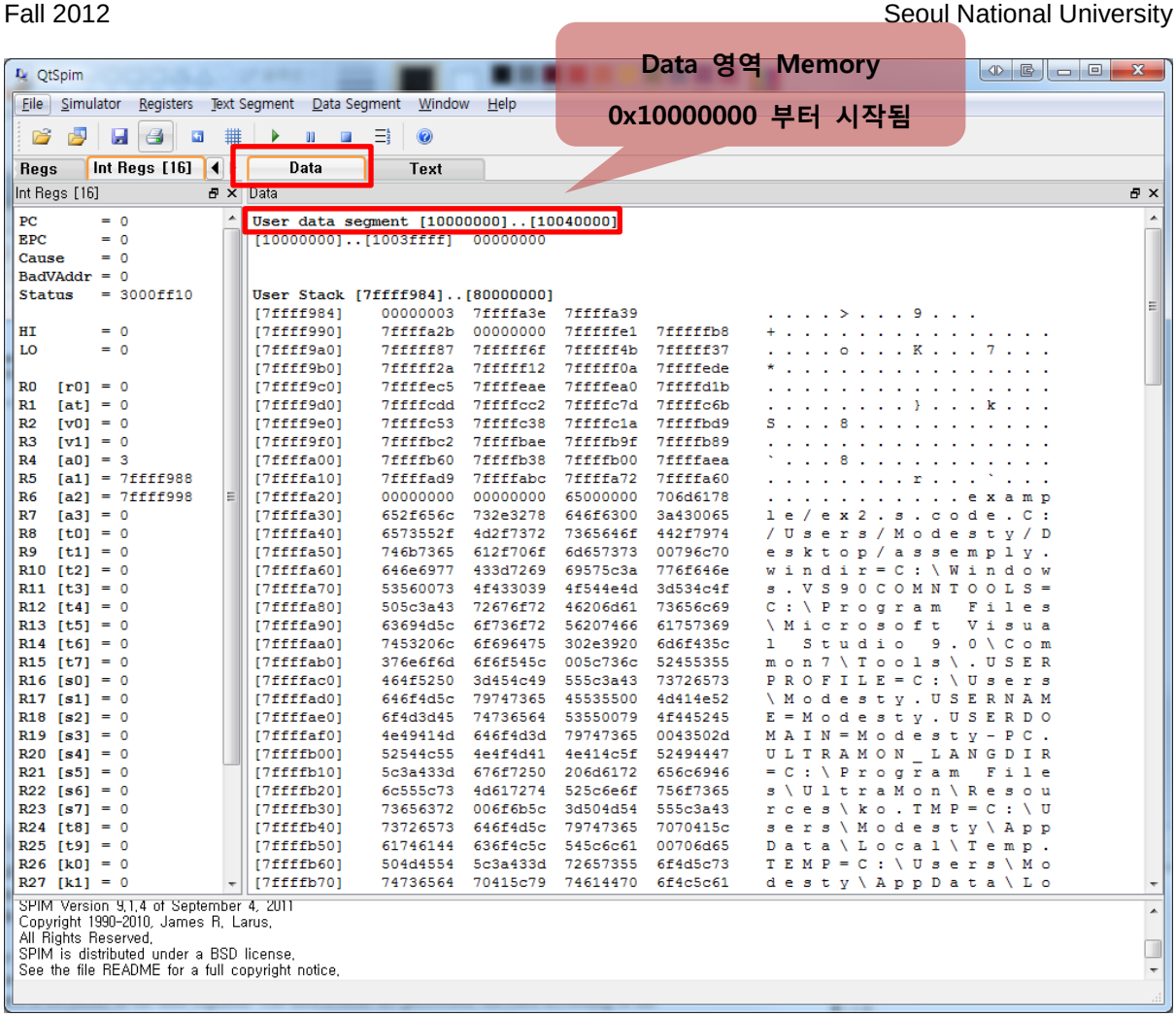
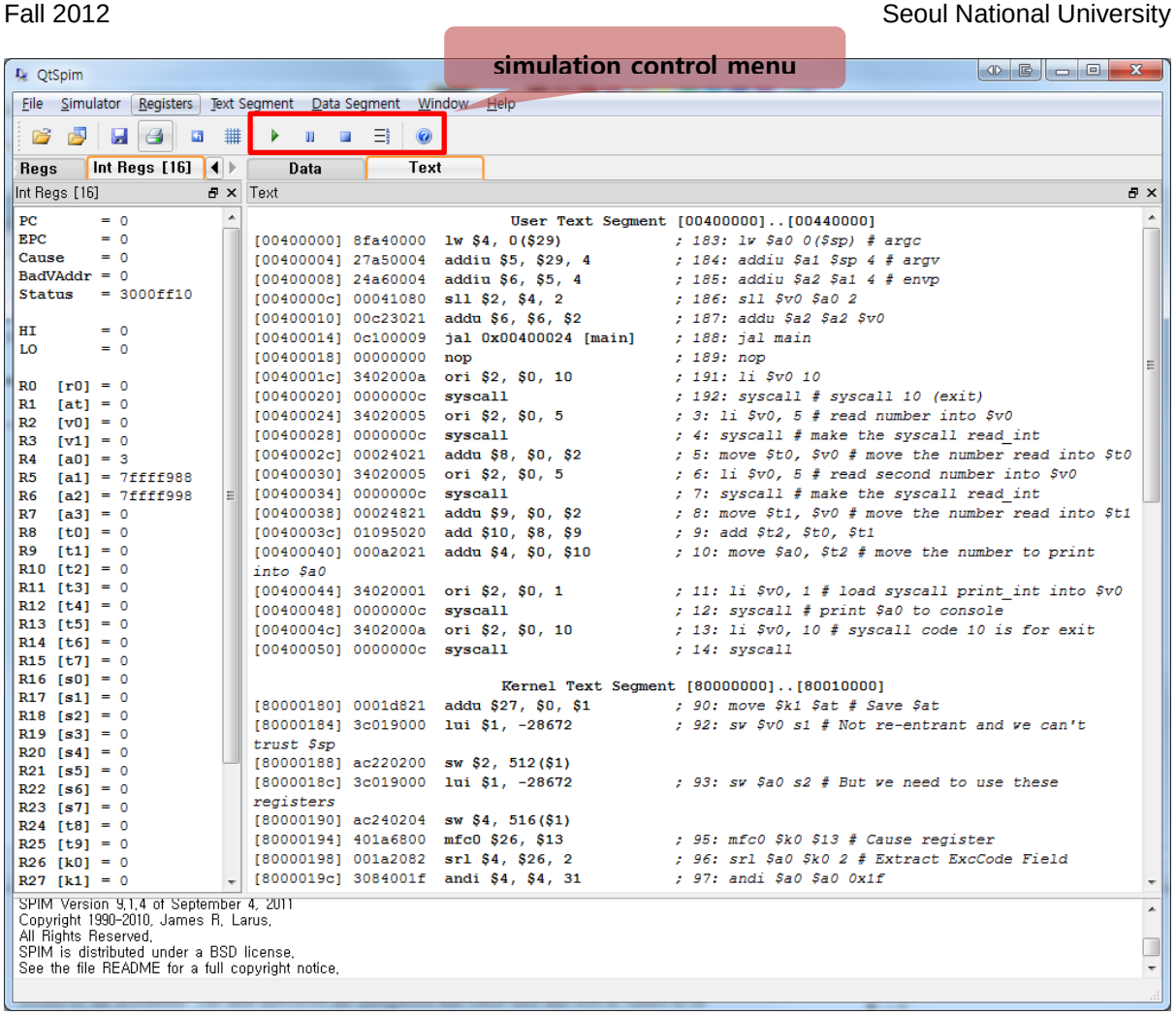


Figure 2 SPIM memory map



위 그림처럼 Data tap을 클릭하고 스크롤을 올려보면 제일 위에 User data segment라고 해서 load, store instruction 수행 시 데이터를 메모리에 저장하고 불러올 때 쓰는 메모리 영역이 바로 여기가 된다.



이제 시뮬레이션을 실행 해 보자. 위 그림의 메뉴를 사용하면 되는데, 재생버튼을 누르면 모든 instruction이 모두 수행되어 버리므로 중간과정을 관찰할 수 없다. 그래서 가장 오른쪽 메뉴인 single step을 클릭하면(또는 단축키 F10) 0x00400000부터 PC값이 4씩 올라가면서 instruction의 위치를 가리키고, 해당 instruction이 수행되면서 변하는 register의 값이라든지 Data 영역 memory의 변화 등을 관찰할 수 있다.

한 번 simulation을 실행하고 나면 register와 메모리 값, PC 등을 모두 초기화하고 다시 시작해야 하므로 SPIM의 **File 메뉴 → Reinitialize and LoadFile** 을 클릭하여 다시 example1.txt를 불러와서 실행한다. 이번에는 **Simulator 메뉴 → Run/Continue (단축키 F5)**를 클릭하여 실행하고 console창에 숫자입력 <enter>, 숫자입력 <enter> 하면 두 입력한 숫자의 합이 출력된다.

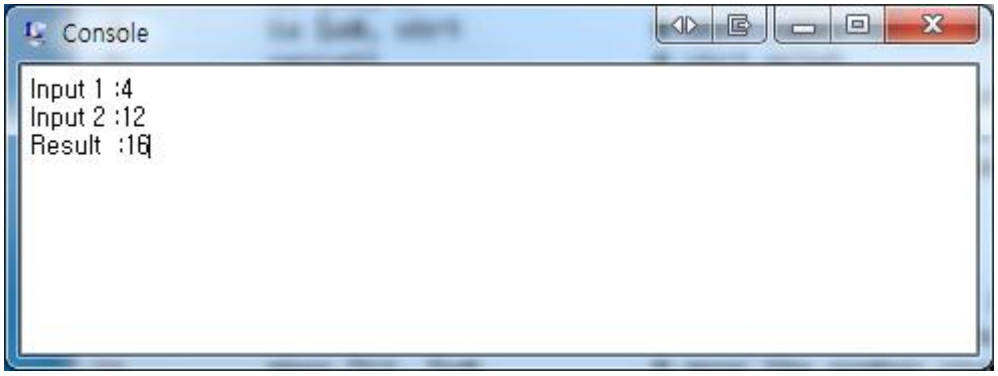


Figure 3 example1의 출력화면

Requirement #1
첨부한 **example2, 3, 4, 5**를 위의 example1에서 처럼 SPIM을 통해 수행한 결과를 보고서에 첨부하라. 또한 모든 instruction에 대한 설명을 주석으로 달아서 보고서에 첨부하라.

2. Instruction

ADD	ADDU	ADDI	ADDIU	SUB	SUBU
SLT	SLTU	SLTI	SLTIU	AND	ANDI
OR	ORI	XOR	XORI	NOR	SRL
SRA	SLL	SRLV	SRAV	SLLV	LW
SW	BEQ	BNE	BLTZ	BLEZ	BGTZ
BGEZ	LUI	J	JAL	JR	

위의 35개 instruction은 프로젝트2부터 구현할 MIPS에서 지원하게 될 Instruction 목록이다. 이 instruction들에 대한 datapath를 구현하려면 instruction의 format, 기능 등을 분류하고 정확히 알고 있어야 한다. 그렇지 않으면 프로젝트 2에서 상당히 고생하게 될 것이다. R-type/I-type/J-type으로 분류하고 각각을 아래의 예시 에서 처럼 정리한다.

예시
R-type **1. ADD** Description: Adds two registers and stores the result in a register Operation: $d = $s + $t Syntax: add $d, $s, $t Encoding: 0000 00ss ssst tttt dddd d000 0010 0000
I-type

1. ADDI

Description	Adds a register and a sign-extended immediate value and stores the result in a register
Operation	$t = $s + imm
Syntax	addi $t, $s, imm
Encoding	0010 00ss ssst tttt iiii iiii iiii iiii

J-type

1. J

Description	Jumps to the calculated address
Operation	PC = (PC & 0xf0000000) | (target << 2)
Syntax	j target
Encoding	0000 10ii iiii iiii iiii iiii iiii iiii

Requirement #2

35개의 instruction에 대해 R-type/I-type/J-type으로 분류하고 위의 양식에 맞춰 그 내용을 기술하여 보고서에 첨부한다. 내용이 정확하면서 본인이 알아보기 쉽도록 하는 것이 중요하다.

3. Assembly coding

지금까지 익힌 SPIM의 사용법과 Instruction에 대한 이해를 바탕으로 직접 assembly coding을 해 본다. 아래의 C코드에 대응하는 assembly 코드를 작성하고 SPIM을 사용하여 테스트 해 본다.

```
#include <stdio.h>

void main() {
  int a[7] = {-36, 20, -27, 15, 1, -62, -41};
  int n = 7;
  int i;
  int npos=0;
  int nneg=0;

  printf("Computer Organization, Fall 2011\n");
  printf("SPIM simulation\n");
  printf("Athor : 본인이름, StudentNum : 학번\n");

  for (i = 0; i < n; i++){
     if(a[i] > 0)
        npos++;
     else
```

```
            nneg++;
        }
         printf("Number of pos : %d\n", npos);
         printf("Number of neg : %d\n", nneg);

    }
```

Requirement #3
위의 C코드에 상응하는 assembly 코드를 작성하고 주석을 달아 보고서에 첨부한다. 작성한 코드 파일 또한 따로 제출한다.

Additional Requirement
필수는 아니지만 성의 있게 수행하면 가산점을 받을 수 있다. Requirement#2에서 조사한 35개의 instruction을 최대한 많은 수를 활용하여 자유주제로 assembly 코드를 작성 해 본다. 프로그램의 수준은 상관없고 단순히 instruction을 사용한 수가 많을수록 좋은 평가를 받는다.

V. Submission

제출과제
1. 보고서(이메일과 hardcopy 모두 제출)
① Requirement#1 example2~5 simulation 결과 캡쳐 첨부, 주석 달린 코드 내용을 문서에 첨부 ② Requirement#2 35개의 instruction 정리한 내용 첨부 ③ Requirement#3 본인이 작성한 Assembly code에 주석 달아서 보고서에 첨부, simulation 결과 첨부
2. Requirement #3에서 본인이 작성한 assemply code 파일(이메일 제출)

제출방법
1. 조교 **이메일** modesty@sdgroup.snu.ac.kr 로 보고서, 코드파일을 압축하여 제출. 2. 보고서 **Hardcopy**는 각자 출력해서 10월 9일 수업시간 전에 제출
※ 이메일 제출은 10/8일 월요일 저녁 11시 59분 까지 제출분만 인정(메일 도착 시간 기준)

APPENDIX A Background knowledge

1. Assembly & machine language

C언어로 프로그래밍코드를 작성하고 컴파일하고 실행파일을 만들어 application을 수행하는 일련의 프로그래밍 경험은 다들 있을 것이다. 여기에는 여러 가지 전제가 있는데, 우선 대부분의 PC에서 사용하고 있는 Processor는 Intel architecture를 사용하고 있다. 즉 ISA(Instruction Set Architecture)가 Intel target이라면 이 ISA에서 지원하는 instruction으로 소스코드가 컴파일 된다. 즉 보통 우리가 사용하고 있는 컴파일러는 정확히 말하면 Inter의 x86 ISA용 instruction을 사용하여 C코드를 x86 ISA가 알아들을 수 있는 machine language로 번역해 주는 툴 이라고 할 수 있다. 이 컴파일러를 통해 나온 실행파일은 Intel processor에서만 실행 가능하다. 실제 MIPS ISA용 컴파일러로 C코드를 컴파일 하여 실행하면 잘못 된 비트파일이라는 메시지가 나온다.

다시 말해 어떤 소스코드를 우리가 설계하는 MIPS ISA를 타겟으로 테스트하기 위해서는 첫째로 MIPS ISA를 지원하는 **cross-compiler**가 필요하고, 둘째로 컴파일 된 machine code를 실행 할 **MIPS 프로세서**가 필요하다.

gcc는 GNU에서 만든 컴파일러 이다. GNU에 대해서는 각자 자료조사를 해 보시길 바란다. gcc를 사용한 컴파일과정은 아래와 같다.

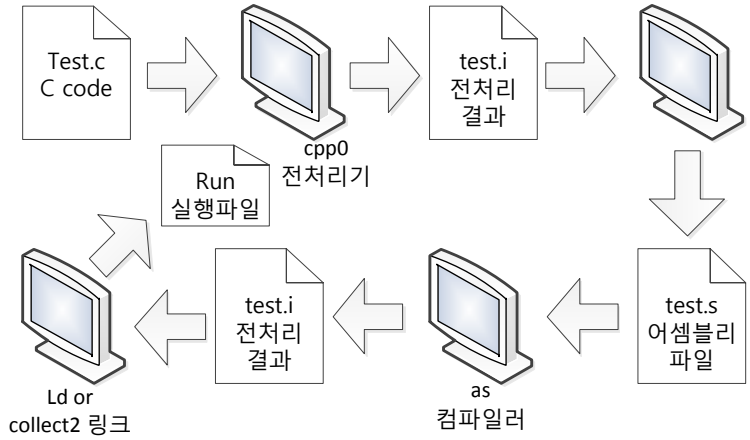


Figure 1 gcc 컴파일 과정[1]

실제로 mipsel-linux-gcc 라는 MIPS ISA 타겟의 cross-compiler를 설치하고 컴파일한 결과를 살펴보자. 아래의 명령을 수행하면 –v 옵션에 의해 컴파일 과정이 화면에 출력되고, --save-temps 옵션에 의해 컴파일 과정에서 발생하는 중간 파일들(test.s, test.i 등)이 모두 저장된다. 실제로 테스트 해 볼 간단한 c source에 대해 machine language code 생성과정을 수행하면 다음과 같은 결과를 얻을 수 있다.

[1]백창우,『유닉스·리눅스 프로그래밍 필수 유틸리티 : vi, make, gcc, gdb, cvs, rpm』,한빛미디어㈜,1999, p.164.

*명령어 : **mipsel-linux-gcc –v –-save-temps –o run test.c**

```
[modesty@bunjee ~/mips_test]$ mipsel-linux-gcc -v --save-temps -o run test.c
Reading specs from /home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/specs
Configured with: ./configure --target=mipsel-linux --prefix=/usr --program-prefix=mipsel-linux- --with-headers=/project/toolchain_mips_le/kernel-
2.6-x/fedora-core-x/linux/include --enable-languages=c,c++ : (reconfigured) ./configure --target=mipsel-linux --prefix=/usr --enable-languages=c
--without-headers --with-gnu-ld --with-gnu-as --disable-shared --disable-threads : (reconfigured) ./configure --target=mipsel-linux --prefix=/usr
 --program-prefix=mipsel-linux- --with-headers=/project/toolchain_mips_le/kernel-2.6-x/fedora-core-x/linux/include --enable-languages=c,c++
Thread model: posix
gcc version 3.4.4
 /home/modesty/usr/bin/../libexec/gcc/mipsel-linux/3.4.4/cc1 -E -quiet -v -iprefix /home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/ test.c -o
 test.i
ignoring nonexistent directory "/usr/lib/gcc/mipsel-linux/3.4.4/include"
#include "..." search starts here:
#include <...> search starts here:
 /home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/include
 /home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/../../../../mipsel-linux/sys-include
 /home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/../../../../mipsel-linux/include
 /usr/lib/gcc/../../mipsel-linux/sys-include
 /usr/lib/gcc/../../mipsel-linux/include
End of search list.
 /home/modesty/usr/bin/../libexec/gcc/mipsel-linux/3.4.4/cc1 -fpreprocessed test.i -quiet -dumpbase test.c -auxbase test -version -o test.s
GNU C version 3.4.4 (mipsel-linux)
        compiled by GNU C version 3.4.2 20041017 (Red Hat 3.4.2-6.fc3).
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
test.c: In function `main':
test.c:1: warning: return type of 'main' is not `int'
 /home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/../../../../mipsel-linux/bin/as -EL -no-mdebug -32 -v -KPIC -o test.o test.s
GNU assembler version 2.16.1 (mipsel-linux) using BFD version 2.16.1
 /home/modesty/usr/bin/../libexec/gcc/mipsel-linux/3.4.4/collect2 --eh-frame-hdr -EL -dynamic-linker /lib/ld.so.1 -o run /home/modesty/usr/bin/..
/lib/gcc/mipsel-linux/3.4.4/../../../../mipsel-linux/lib/crt1.o /home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/../../../../mipsel-linux/lib/
crti.o /home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/crtbegin.o -L/home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4 -L/home/modesty/usr/b
in/../lib/gcc -L/home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/../../../../mipsel-linux/lib -L/usr/lib/gcc/../../mipsel-linux/lib test.o -lg
cc --as-needed -lgcc_s --no-as-needed -rpath-link /lib:/usr/lib -lc -lgcc --as-needed -lgcc_s --no-as-needed /home/modesty/usr/bin/../lib/gcc/mip
sel-linux/3.4.4/crtend.o /home/modesty/usr/bin/../lib/gcc/mipsel-linux/3.4.4/../../../../mipsel-linux/lib/crtn.o
```

Figure 2 compile 과정

```
 1 main:
 2     addiu   $sp,$sp,-24
 3     sw  $fp,16($sp)
 4     move    $fp,$sp
 5     sw  $0,8($fp)
 6     sw  $0,12($fp)
 7     sw  $0,8($fp)
 8 $L2:
 9     lw  $2,8($fp)
10     slt $2,$2,10
11     beq $2,$0,$L1
12     lw  $3,12($fp)
13     lw  $2,8($fp)
14     addu    $2,$3,$2
15     sw  $2,12($fp)
16     lw  $2,8($fp)
17     addiu   $2,$2,1
18     sw  $2,8($fp)
19     b   $L2
20 $L1:
21     move    $sp,$fp
22     lw  $fp,16($sp)
23     addiu   $sp,$sp,24
24     j   $31
```

② test.s (assembly language code)

```
 1 void main(){
 2
 3     int i = 0;
 4     int j = 0;
 5
 6     for(i = 0; i < 10; i++)
 7         j += i;
 8 }
```

① test.c (c source file)

```
 1 00100111101111011111111111101000
 2 10101111101111100000000000010000
 3 00000011101000001111000000100001
 4 10101111110000000000000000001000
 5 10101111110000000000000000001100
 6 10101111110000000000000000001000
 7 10001111110000100000000000001000
 8 00101000010000100000000000001010
 9 00010000010000000000000000001000
10 10001111110000110000000000001100
11 10001111110000100000000000001000
12 00000000011000100001000000100001
13 10101111110000100000000000001100
14 10001111110000100000000000001000
15 00100100010000100000000000000001
16 10101111110000100000000000001000
17 00010000000000001111111111110101
18 00000011110000001110100000100001
19 10001111101111100000000000010000
20 00100111101111010000000000011000
21 00000011111000000000000000001000
```

③ machine language code

Figure 3 MIPS machine code 생성 과정

2. ISS(Instruction Set Simulator)

소스코드를 우리가 설계하는 MIPS ISA를 타겟으로 테스트하기 위해 첫째로 **cross-compiler**, 둘째로 컴파일 된 machine code를 실행 할 **MIPS 프로세서**가 필요하다고 했다. 그런데 우리는 지금 MIPS 프로세서가 없기 때문에 MIPS instruction으로 컴파일 된 machine code가 있어도 실행을 해 볼 수가 없다. 이럴 때 ISS가 필요한 것이다. 실제로는 프로세서를 설계할 때 먼저 ISS를 설계하는 것이 순서이다. 왜냐하면 하드웨어로 프로세서를 설계했을 때 이 것이 제대로 동작하는지 확인하기 위해 비교해야 할 대상이 필요한데 바로 ISS가 비교 대상이 된다. 게다가 Verilog 시뮬레이터를 이용하여 회로의 동작을 확인하는 것은 속도가 너무도 느리다. 그래서 ISS는 주로 HDL(Hardware Description Language)가 아닌 C, C++ 등의 상위 언어로 모델링하는 것이 보통이다. 아래 내용은 IDEC newsletter에 KAIST 배영돈 박사님(donny.ics.kaist.ac.kr)께서 연재하신 '마이크로프로세서 설계 무작정 따라하기'에 실린 내용의 일부이다.

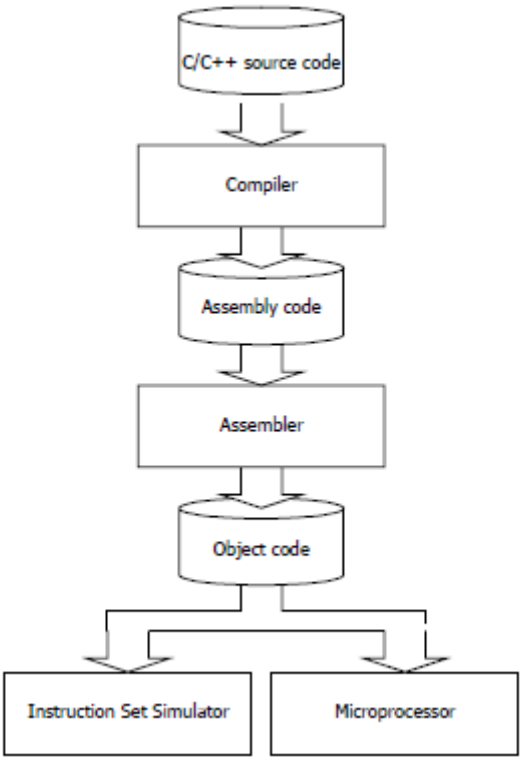


Figure 4 Microprocessor 개발 환경

C코드를 설계하고자 하는 processor 타겟으로 컴파일하여 최종 object code까지 만들어지면 이것을 설계한 Microprocessor와 C언어 등으로 모델링 된 ISS에 동시에 넣어 두 결과를 비교함으로써 설계한 Microprocessor가 제대로 동작하는지 검증할 수 있다.

이번 프로젝트에서는 C언어 등을 사용하여 ISS를 설계하지는 않고 MIPS용으로 많이 사용되는 SPIM이라는 ISS를 사용하여 Instruction을 test하고 assemply coding도 직접 해 본다.