Programming at Different Levels: A Teaching Module for Undergraduate Computer Architecture Course

Xuejun Liang, Loretta A. Moore, and Jacqueline Jackson
Department of Computer Science, Jackson State University, Jackson, MS, USA

Abstract - *Computers can be programmed by using the high-level programming languages, such as C, C++, Java, Python, etc., and the low-level assembly programming languages, such as MIPS, IA-32, Accumulator, etc.. In this teaching module, we will use some simple computing examples to illustrate how to solve these same problems by using different computer programming languages. This work will expose students with different computer architectures and programming languages. It will show the similarities and differences among these architectures and programming languages. So, it will help students to get a better understanding of fundamentals of computer architectures and programming languages, and to enhance their problem solving skills with using computers.*

Keywords: Computer Architecture, Assembly Language, High-Level Programming Language, Problem Solving

1 Introduction

Computing with using computers can be carried out at different levels. At the application level, computer users can use any computer software for their application. For example, users can use the Microsoft Office Excel to organize and compute their data. At the high-level programming language level, computer programmers can use high-level programming languages, such as C, C++, Java, Python, etc., to write a computer program to solve their problems. At the low-level programming language level, programmers can use different machine assembly programming languages, such as MIPS, IA-32, Accumulator, etc., to write a computer program for the corresponding computer. It is quite difficult for a new computer science student to understand the similarities and differences among high-level programming languages and among low-level programming languages, and relationship between high-level and low-level programming languages. But, the above knowledge is important for students to have a deeper understanding of computer systems, to write correct and efficient computer programs, and to debug computer codes.

In this teaching module, we will use some simple examples to illustrate how to solve these computational problems using different means with computers. This work will help students to have a better understanding of fundamentals of computer architectures and programming languages. More importantly, this work will show how to design and implement a computer algorithm for a particular computer architecture or using a particular programming language. Therefore, it is highly expected that students will enhance their problem solving ability using computers significantly after they complete this teaching module.

In this teaching module, three simple computing problems are studied: (1) evaluate arithmetic expressions, (2) sort an array of integers, and (3) get an array of integer from the keyboard, sort the array, and display the sorted array on the screen. Three high-level programming languages are utilized: (1) Python, (2) Java, and (3) C++ (and C). Four computer architectures (assembly languages) are considered: (1) MIPS, (2) IA-32, (3) Accumulator-based machine, and (4) Stack-based machine. Eclipse is used for both C++ and Java programming. Real assembly languages are used for MIPS and IA-32 architectures, while the accumulator-based machine and stacked-based machine are simulated using Python, Java, and C++ for evaluating arithmetic expressions.

In the rest of the paper, the brief instructions and references for software download and installation are given first in Section 2. So, students can setup their computer for compiling and running the given codes and their own codes for the study. Then, the three computing problems and the solutions are discussed in Sections 3, 4, and 5, respectively. Finally, the conclusion and future work are given in Section 6.

2 Software Download and Installation

The software packages listed below are needed for running programs discussed in this paper. They all have been installed and tested on the Windows 7 platform.

2.1. **Python**: Download Python 2.7 release for Windows from http://www.python.org/download/releases/2.7/ [1] and select Windows x86 MSI Installer (2.7). Then run the downloaded program Python-2.7 for the installation.

2.2. **SPIM Simulator**: This simulator will be used for the MIPS assembly language programming. Download it from http://sourceforge.net/projects/spimsimulator/files/ [2] and select PCSpim_9.1.9.zip. Unzip it and run the setup program for the installation.

2.3. **MinGW**: (Minimal GNU for Windows): MinGW will provide a native Windows port of the GNU Compiler Collection (GCC), which will be used for linking IA-32 object code with C library I/O functions. Download MinGW from http://sourceforge.net/projects/mingw/files/ and click Installer and then select mingw-get-setup.exe. Then run the setup program for the installation.

2.4. **NASM**: (the Netwide Assembler): It is an assembler targeting the Intel x86 series of processors. It can get object codes from IA-32 assembly codes. Download NASM from http://www.nasm.us/pub/nasm/releasebuilds/2.10.09/win32/ and select nasm-2.10.09-win32.zip. Unzip it into a directory (for example, C:\NASM) where you want to install NASM. Then add the directory (for example, C:\NASM\nasm-2.10.09) where the nasm executable is stored into the Path environment variable.

To add C:\NASM\nasm-2.10.09 into the Path environment variable in Windows 7, from the Window's Start, select in turn the following: Control Panel, System and Security, System, Advanced system settings. Then, the System Properties dialogue window shows up, click the Environment Variables button. Under System variables list, find and select the variable Path, and click the Edit... button. Then type "C:\NASM\nasm-2.10.09;" at the end of the variable value. When you are done, click OK.

To compile and run x86 assembly language program, you will type `cmd` in Window's search programs and files textbox to get a command line window first. Then compile and link your program, say, exprX86.asm, by typing

```
>nasm -f win32 exprX86.asm
>gcc exprX86.obj -o exprX86
```

Finally, run the program by typing

```
>exprX86
```

2.5. **Eclipse IDE for Java Developers**: Download and unzip the file eclipse-java-indigo-SR2-win32-x86_64.zip from http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/indigo/SR2/eclipse-java-indigo-SR2-win32-x86_64.zip. Then, create a shortcut of eclipse.exe on your Desktop.

2.6. **Eclipse IDE for C++ Developers**: Download and unzip eclipse-cpp-indigo-SR2-incubation-win32-x86_64.zip from http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/indigo/SR2/eclipse-cpp-indigo-SR2-incubation-win32-x86_64.zip. Then, create a shortcut of eclipse.exe on your Desktop.

3 Evaluate Arithmetic Expressions

Consider the following arithmetic expression:

$$d = (a+b) * (c-b) + (b-c) + a \qquad (1)$$

Assume initial values a=32, b=45, and c=23. Now, we will evaluate this expression using the following programming languages and architectures.

3.1. **Python**: Figure 1 lists the Python code to evaluate (1).

```
a = 32
b = 45
c = 23
d = (a+b) * (c-b) + (b-c) + a
print d
```

Figure 1: Python Code to Evaluate the Expression (1)

3.2. **Java**: Figure 2 lists the Java code to evaluate (1).

```
public class ComputeExpression {
  public static void main(String[] args) {
    int a = 32;
    int b = 45;
    int c = 23;
    int d = (a+b)*(c-b)+(b-c)+a;
    System.out.println("The answer is  " + d);
  }
}
```

Figure 2: Java Code to Evaluate the Expression (1)

3.3. **C++**: Figure 3 lists the C++ code to evaluate (1).

```
#include <iostream>
int main() {
  int a = 32;
  int b = 45;
  int c = 23;
  int d = (a+b)*(c-b)+(b-c)+a;
  std:: cout << "The answer is " << d << endl;
  return 0;
}
```

Figure 3: C++ Code to Evaluate the Expression (1)

3.4. **MIPS**: Figure 4 lists the MIPS code to evaluate (1), where a, b, c, and d are assumed to be stored in memory locations A, B, C, and D, respectively. Note that the result display is omitted. The SPIM provides system calls to display data in registers or character strings stored in memory. From the code, we can see that MIPS is a load/store architecture (or register-to-register arithmetic) and there are three operands per ALU instructions.

```
  .text              # text segment
  .globl main
main:
  lw   $t0, A        # $t0 = a
  lw   $t1, B        # $t1 = b
  lw   $t2, C        # $t2 = c
  sub  $t3, $t2, $t1 # $t3 = c-b
  sub  $t4, $t0, $t3 # $t4 = a-(c-b) = (b-c)+a
  add  $t5, $t0, $t1 # $t5 = a+b
  mul  $t6, $t5, $t3 # $t6 = (a+b)*(c-b)
  add  $t7, $t6, $t4 # $t7 = (a+b(c-b)+(b-c)+a
  sw   $t7, D
  ......
  li $v0,10
  syscall            # au revoir...

  .data              # data segment
A:    .word 32
B:    .word 45
C:    .word 23
D:    .word 0
```

Figure 4: MIPS Code to Evaluate the Expression (1)

3.5. **IA-32**: Figure 5 lists IA-32 code to evaluate (1), where a, b, c, and d are assumed to be stored in memory locations A, B, C, and D, respectively. Note that the result display is omitted. The NASM allows to use C library function `_printf` to display data. Programmers for IA-32 can simply call it in the assembly code after pushing its actual parameters onto the stack. From the code, we can see that IA-32 is not a load/store architecture (or register-to-register arithmetic) and there are two operands per ALU instructions. So, the output operand of ALU instructions is the same with the first input operand, and only the second input operand can be a memory location.

```
   global  _main
   extern  _printf

   section .text    ; Text section
 _main:
   mov  eax, [A]    ; eax = a;
   mov  ebx, [B]    ; ebx = b;
   add  eax, ebx    ; eax = a+b;
   sub  ebx, [C]    ; ebx = b-c;
   imul ebx         ; edx:eax = (a+b)*(b-c);
   add  ebx, [A]    ; ebx = (b-c)+a;
   sub  ebx, eax    ; ebx = (a+b)*(c-b)+(b-c)+a;
   mov  [D], ebx    ; [D] = (a+b)*(c-b)+(b-c)+a;
   ......
   ret

   section .data    ; Data section
 A:       dd  32
 B:       dd  45
 C:       dd  23
 D:       dd  0
```

Figure 5: IA-32 Code to Evaluate the Expression (1)

3.6. **Accumulator**: Assume that accumulator-based machine has the following five instructions (functions), where *acc* is the accumulator, as shown in Table 1. Then, Figure 6 lists the accumulator code to evaluate (1). Note that the accumulator machine has only one operand and the rest of operands is *acc*.

Table 1: Instructions of Accumulator Machine

Instruction	Meaning
ld(A)	acc = memory[A]
add(A)	acc = acc + memory[A]
mul(A)	acc = acc * memory[A]
sub(A)	acc = acc - memory[A]
st(A)	memory[A] = acc

```
 # Accumulator code
 ld(a)     #acc = a
 add(b)    #acc = a+b
 st(d)     #d = a+b
 ld(c)     #acc = c
 sub(b)    #acc = c-b
 mul(d)    #acc = (a+b)*(c-b)
 add(b)    #acc = (a+b)*(c-b)+b
 sub(c)    #acc = (a+b)*(c-b)+(b-c)
 add(a)    #acc = (a+b)*(c-b) +(b-c)+a
 st(d)     #d = (a+b)*(c-b) + (b-c) + a
```

Figure 6: Accumulator Code to Evaluate the Expression (1)

In order to run the code shown in Figure 6, we can define and use the instructions (or functions) in Table 1 using high-level programming languages and then run the code in simulation.

3.6.1. **Python Simulation**: Figure 7 lists the Python code to simulate the accumulator machine and to evaluate the expression (1).

```
acc = 0

Def ld(A):
  global acc
  acc = A
def add(A):
  global acc
  acc = acc + A
def mul(A):
  global acc
  acc = acc * A
def sub(A):
  global acc
  acc = acc - A
def st(A):
  global acc
  A[0] = acc

a = 32
b = 45
c = 23
d = [0]

# Accumulator code

print d
```

Figure 7: Python Code to Simulate the Accumulator Machine

From the code in Figure 7, we can see that *acc* is implemented as a global variable and the array type parameter is used for passing the result back from the function `st(A)` because the Python function uses the call-by-value only.

3.6.2. **Java Simulation**: Figure 8 lists Java code to simulate the accumulator machine and to evaluate the expression (1).

```
public class ExprAccumulator {
  static int acc;

  static void ld(int A){
    acc = A;
  }
  static void add(int A){
    acc = acc + A;
  }
  static void sub(int A){
    acc = acc - A;
  }
  static void mul(int A){
    acc = acc * A;
  }
  static void st(int [] A){
    A[0] = acc;
  }

  public static void main(String[] args) {
    int a = 32;
    int b = 45;
    int c = 23;
    int[] d = new int[1];

    //# Accumulator code

    System.out.println("The answer: " + d[0]);
  }
}
```

Figure 8: Java Code to Simulate the Accumulator Machine

From the code in Figure 8, we can see that *acc* and all the functions are static. The array type parameter is used for passing the result back from the function `st(int[]A)` because the Java function uses the call-by-value only.

3.6.3. **C++ Simulation**: Figure 9 lists C++ code to simulate the accumulator machine and to evaluate the expression (1).

```
#include <iostream>
using namespace std;

int acc;

void ld(int A){
  acc = A;
}
void add(int A){
  acc = acc + A;
}
void mul(int A){
  acc = acc * A;
}
void sub(int A){
  acc = acc - A;
}
void st(int &A){
  A = acc;
}

int main()  {
  int a = 32;
  int b = 45;
  int c = 23;
  int d = 0;

  //# Accumulator code

  std::cout << "The answer: " << d << endl;
  return 0;
}
```

Figure 9: C++ Code to Simulate the Accumulator Machine

From the code in Figure 9, we can see that *acc* is implemented as a global variable. The C++ function `st(int &A)` now uses the call-by-reference for passing the result.

3.7. **Stack-based**: Assume that the stack-based machine has the following five instructions (functions) as shown in Table 2. Then, Figure 10 lists the stack-based code to evaluate (1).

Table 2: Instructions of Stack-based Machine

Instruction	Meaning
push(A)	tos++; S[tos] = memory[A]
pop(A)	memory[A] = S[tos]; tos--
add	S[tos-1] = S[tos-1]+S[tos]; tos--
sub	S[tos-1] = S[tos-1]-S[tos]; tos--
mul	S[tos-1] = S[tos-1]*S[tos]; tos--

```
#Stack-based code
push(a)
push(b)
add()      # a+b
push(c)
push(b)
sub()      # c-b
mul()      # (a+b)*(c-b)
push(b)
push(c)
sub()      # b-c
add()      # (a+b)*(c-b)+(b-c)
push(a)
add()      # (a+b)*(c-b)+(b-c)+a
pop(d)
```

Figure 10: Stack-Based Code to Evaluate the Expression (1)

Note that all operands of ALU instructions are located in the operand stack. Therefore, ALU instruction has zero operand in the stack-based machine. The code shown in Figure 10 can be obtained from the postfix notation of Expression (1), which is

$$d = a\ b + c\ b - *\ b\ c - + a + \qquad (2)$$

In order to run the code shown in Figure 10, we can define and use the instructions (or functions) in Table 2 using high-level programming languages and then run the code in simulation. We can use an array to simulate the operand stack as shown in Table 2. At the mean time, we can use the stack data structure in high-level language or library to simulate the operand stack. These codes are omitted here for the space.

4 Sorting Array

Consider sorting the integer array: 68, 86, 65, 75, 67, 76, 62, 79, 98, 67 in an ascending order.

4.1 **Using Built-in Sorting Function**: Figures 11, 12 and 13 list the Python code, Java code, and C++ code that use their built-in array sorting function to sort the array, respectively. It can be noticed that the codes are all quite simple. The Python code are only three lines! Note that printing an integer array needs to have a loop structure in C++, but the loop is not needed in Python and Java.

```
a = [68,86,65,75,67,76,62,79,98,67]
a.sort()
print a
```

Figure 11: Python Code to Sort an Array

```
import java.util.Arrays;

public class Sorting {
  public static void main(String[] args) {
    int[]a ={68,86,65,75,67,76,62,79,98,67};
    Arrays.sort(a);

    //print results
    System.out.println(Arrays.toString(a));
  }
}
```

Figure 12: Java Code to Sort an Array

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
  const int len = 10;
  int a[len]={68,86,65,75,67,76,62,79,98,67};
  sort(a, a+len);

  //print results
  for (int i=0; i< len; i++)
    cout << a[i] << " ";
  cout << endl;

  return 0;
}
```

Figure 13: C++ Code to Sort an Array

4.2 **Using the Bubble Sort Algorithm**: Consider putting the elements in a vertical array. The bubble sort algorithm will scan the array elements from bottom to up repeatedly until the array is sorted. In each scan, every adjacent pair of elements are compared. They are swapped if they are not in right order. So, after the first scan, the first element in the array is already in its right place. Similarly, the second scan will make the second element in the array to be in its right place, and so on.

4.2.1. **Python, Java, and C++**: Figure 14 lists Python code that uses the bubble sort algorithm to sort the array. As you can guess, the bubble sort implementations in Java and C++ will be identical, and they are different from Python only in syntax. Java and C++ use { } to include a block of statements, while Python use : and the indentation to indicate a block of statements. Variables should have an explicit data type in Java and C++, but not in Python. Therefore, Java and C++ codes that use the bubble sort algorithm are omitted here.

```
a = [68,86,65,75,67,76,62,79,98,67]
len = len(a)

current = 0;
while (current < len - 1):
  index = len-1
  while (index > current):
    if (a[index] < a[index-1]):
      tmp = a[index]
      a[index] = a[index-1]
      a[index-1] = tmp
    index = index - 1
  current = current + 1

print a
```

Figure 14: Python Code: Bubble Sort

4.2.2. **MIPS and IA-32**: After the bubble sort algorithm has been implemented using a high-level programming language, a better way to implement it in an assembly language again is to consider how to translate branch and loop statements from high-level languages to low-level languages. In Table 3, the second column lists corresponding constructs in assembly languages for branches and loops in high-level language.

Table 3: Branch and Loop in Assembly Language

High Level Language	Assembly Language

`if(x<y)`
`  A;`
`C;`
	`     if(x>=y) goto c`
	`     A`
	`c:   C`

`if(x<y)`
`  A;`
`else`
`  B;`
`C;`
	`     if(x>=y) goto b`
	`     A`
	`     goto c`
	`b:   B`
	`c:   C`

`While(x<y)`
`  A;`
`C;`
	`a:   if(x >= y)goto c`
	`     A`
	`     goto a`
	`c:   C`

Figures 15 and 16 list MIPS code and IA-32 code that both use bubble sort algorithm to sort the array, respectively. One key point in assembly language programming is how to access array elements, i.e. how to compute the memory addresses of array elements. Simply speaking, the address of an array element can be computed by adding the array base address and the array element offset, which is the product of the array element index and the element size. As the element size in the code is four, the element offset should be multiple of four.

```
  .text
  .globl main
main:
  move $t0, $0       # $t0 = current
  lw   $t1, count    # $t1 = 10
  sll  $t1, $t1, 2   # $t1 = 40
  addi $t1, $t1, -4  # $t1 = 36
loop1:
  beq  $t0, $t1, done
  move $t2, $t1      # $t2 = index
loop2:
  beq  $t2, $t0, next
  addi $t3, $t2, -4
  lw   $t4, A($t2)
  lw   $t5, A($t3)
  bge  $t4, $t5, continue
  sw   $t5, A($t2)
  sw   $t4, A($t3)
continue:
  addi $t2, $t2, -4
  b    loop2
next:
  addi $t0, $t0, 4
  b    loop1
done:
  li   $v0,10
  syscall            # au revoir...

  .data
A:     .word 68,86,65,75,67,76,62,79,98,67
count: .word 10
```

Figure 15: MIPS Code: Bubble Sort

```
  global  _main
  section .text
_main:
  sub  edi, edi      ; edi = current
  mov  ecx, [count]  ; ecx = 10
  shl  ecx, 2        ; ecx = 40
  sub  ecx, 4        ; ecx = 36
loop1:
  cmp  edi, ecx
  je   done
  mov  eax, ecx      ; eax = index
loop2:
  cmp  eax, edi
  je   next
  mov  ebx, [eax+A]
  mov  edx, [eax+A-4]
  cmp  ebx, edx
  jge  continue
  mov  [eax+A], edx
  mov  [eax+A-4], ebx

continue:
  sub  eax, 4
  jmp  loop2
next:
  add  edi, 4
  jmp  loop1
done:
  ret

  section .data
A:      dd  68,86,65,75,67,76,62,79,98,67
count:  dd  10
```

Figure 16: IA-32 Code: Bubble Sort

From Figures 15 and 16, it can be seen that both codes have the same program structure with the Python code. Registers $t0, $t1, and $t2 in MIPS code and registers edi, ecx, and eax in IA-32 code have the values of current*4, (len-1)*4, and index*4, respectively, where current, len-1, and index are used in the Python code.

5 Problem Decomposition

In this section, we will consider the problem: get an array of integers from the keyboard, sort the array, and display the sorted array on the screen. This task is composed by three subtasks: (1) input an array, (2) sort an array, and (3) print an array. We want to use a subroutine for array sorting, while the input and the output will remain in the main routine.

5.1 **Python, Java, C++, and C**: Utilizing a subroutine in high-level language is easy as we see in subsections 3.6.1, 3.6.2, and 3.6.3. Printing an integer array in high-level language is also simple. We will only show how to accept an integer array from the keyboard. These are shown in Figures 17-20.

```
a = []
length= int(raw_input("Enter the array length: "))
for i in range(0, length):
  if( i = 0):
    a.append(int(raw_input(Enter your integer array)))
  else
    a.append(int(raw_input()))
```

Figure 17: Python Code: Accept Input Array

```
public class SortBubble {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.print("Enter the array length:  ");
    int length = in.nextInt();
    int [] a = new int[length];
    System.out.print("Enter your integer array:  ");
    for(int i = 0; i < length; i++)
      a[i] = in.nextInt();
    }
}
```

Figure 18: Java Code: Accept Input Array

```
int main() {
  int length;
  cout << "Enter the array length: " << endl;
  cin >> length;
  int a[length];
  cout << "Enter your integer array: " << endl;
  for(int i = 0; i < length; i++)
    cin >> a[i];
  return 0;
}
```

Figure 19: C++ Code: Accept Input Array

```
int main() {
  int length;
  printf("Enter the array length: " );
  scanf("%d", length);
  int a[length];
  printf( "Enter your integer array:" );
  for(int i = 0; i < length; i++)
    scanf(%d", a[i]);
  return 0;
}
```

Figure 20: C Code: Accept Input Array

Note that Figure 20 shows the C code that uses printf and scanf functions to do the I/O. We will need this knowledge to do the I/O for IA-32 assembly programs with using NASM.

5.2 **MIPS and IA-32**: Firstly, how to accept input array from the keyboard and print it on the screen is shown in Figure 21 and 22 for MIPS and IA-32, respectively. Both codes get the array length first and then get array elements using a loop. Both codes also print the array using a loop. The array length is stored at count and the array is stored starting from A. In MIPS code in Figure 21, the system calls are used for both read an integer from keyboard and print an integer on screen. In IA-32 code in Figure 22, _scanf is called for reading and _printf is called for printing. Note that before calling them, we need to push the arguments onto the stack.

```
  .text
  .globl main
main:
  #Read count
  li    $v0, 5
  syscall           # read an integer
  sw    $v0, count

  # Read integer array
  move  $t0, $0
  lw    $t1, count
  sll   $t1, $t1, 2
loop1:
  beq   $t0, $t1, done1
  li    $v0,5
  syscall           # read an integer
  sw    $v0, A($t0)
  addi  $t0, $t0, 4
  b     loop1
done1:

  # print the integer array
  move  $t0, $0
  lw    $t1, count
  sll   $t1, $t1, 2
loop2:
  beq   $t0, $t1, done2
  lw    $a0, A($t0)
  li    $v0,1
  syscall            # print an integer
  la    $a0, sp
  li    $v0,4
  syscall            # print the space
  addi  $t0, $t0, 4
  b loop2
done2:

  li    $v0,10
  syscall            # au revoir...

  .data
A:     .word 0:20
count: .word 0
sp:    .asciiz " "
```

Figure 21: MIPS Code: Accept Input Array and Print It

Secondly, as shown in Figure 23 and Figure 24, the subroutine bubbleSort is designed to have two input arguments. The first one is the array starting address and is passed to the subroutine via $a0 and via edi for MIPS and IA-32, respectively. The second one is the array length and is passed via $a1 and via ecs for MIPS and IA-32, respectively. The sorted array will remain in the same memory locations.

```
  global  _main
  extern  _printf
  extern _scanf

  section .text
_main:
  ;read count
  push  count
  push  rfmat
  call  _scanf       ;read integer
  add   esp, 8

  ;read integer array
  mov   ecx, [count];
  mov   edi, A
read:
  push  ecx
  push  edi
  push  rfmat
  call  _scanf       ;read integer
  add   esp, 8
  add   edi, 4
  pop   ecx
  loop  read

  ;print integer array
  mov   edi, A
  mov ecx, [count]
print:
  push  ecx
  mov   ebx, [edi]
  push  ebx
  push  wfmat
  call  _printf ;print integer
  add   esp, 8
  add   edi, 4
  pop   ecx
  loop  print
  ret

  .section .bss
A:      resd 20
count:  resd 1

  .section .data
rfmat:  db  '%d', 0
wfmat:  db  '%6d', 0
end:    db  10, 0
```

Figure 22: IA-32 Code: Accept Input Array and Print It

```
  .text
  .globl main
main:
  # Read count
  # Read integer array and store at A

  la    $a0, A
  lw    $a1, count

  jal   bubbleSort

  # print the result

  li $v0,10
  syscall          # au revoir...

bubbleSort:
  ......
  jr   $ra
```

Figure 23: MIPS Code with Subroutine Call

```
  global  _main
  extern  _printf
  extern  _scanf

  section  .text
_main:
  ;Read count
  ;Read integer array and store at A

  mov   edi, A
  mov   ecx, [count]

  call  bubbleSort

  ;Print the sorted integer array:
  ret

bubbleSort:
  ......
  ret
```

Figure 24: IA-32 Code with Subroutine Call

Note that Figure 23 and 24 show the whole program skeletons. the subroutine `bubbleSort` codes are omitted here in Figure 23 and 24 because they have almost identical codes with those shown in Figure 15 and 16. The codes of reading an input integer array and printing the output array can be found from Figure 21 and 22.

6 Conclusion and Future Work

The teaching module was taught last semester and students were given a related assignment as bonus. Most students are very interested in this material and like the assignment. Some future work can be as follows.

- Illustrate the stack-based machine by using the Java byte code (or Java Virtual Machine) directly.
- Illustrate implementations of basic data type such as two-dimensional array and class at assembly language level.
- Provide some applications that can be more effectively and simply solved by using assembly languages than high-level programming languages.
- Multi-core processor programming and architecture.
- GPU (graphics processing unit) programming model and architecture.

7 References

[1] Python 2.7 Release, Available at http://www.python.org/download/releases/2.7/

[2] Sourceforge, *Spim MIPS simulator,* Available at http://sourceforge.net/projects/spimsimulator/files/

[3] Sourceforge, *MinGW-Minimal GNU for Windows,* Available at http://sourceforge.net/projects/mingw/files/