Lab 2 – Data types, Loops and Function Call in Assembly

Overview

In this lab, you will convert some simple C programs into MIPS assembly and simulate their execution. Through this lab you are expected to learn how high-level programming language constructs like datatype, loops, functions, etc. are interpreted into assembly instructions and corresponding machine executable codes.

Part I

In this part, you will get familiar with the implementations of loops and datatypes in assembly language. Below, there are three versions of the same program that basically subtracts all elements of an array from corresponding elements of another array and stores the results in a third array. Functionally they are equivalent but the data interpretations are different in the different versions. "Int's" are words (32 bits), 'shorts' are half words (16 bits), and 'unsigned shorts' are unsigned halfwords (16 bits). The basic operation being performed result[i]= var1[i] - var2[i] is the same in all three programs. The difference between the operand types used, affects how the operands are laid out in memory, how the operands are encoded within the registers and the instructions you will use to perform the subtraction. These are the key elements you should focus on when you write your programs.

Version 1. ints

```
int main(void)
{
  int var1[4] ={7, 18, 11, 3};
  int var2[4]= {12, 14, 7, 18};
  int result[4]={0};

  for (int i=0 ; i< 4; i++)
    result[i] = var1[i] - var2[i];

}
```

Version 2. shorts

```
int main(void)
{
  short var1[4] ={7, 18, 11, 3};
  short var2[4]= {12, 14, 7, 18};
  short result[4]={0};

  for (int i=0 ; i< 4; i++)
    result[i] = var1[i] - var2[i];

}
```

Version 3. unsigned shorts

```
int main(void)
{
  unsigned short var1[4] ={7, 18, 11, 3};
  unsigned short var2[4]= {12, 14, 7, 18};
  unsigned short result[4]={0};

  for (int i=0 ; i< 4; i++)
    result[i] = var1[i] - var2[i];

}
```

Loop Structures

High-level programming language loop structures can be represented using following flow charts. In the Do-While-Loop structure, loop instructions are executed first and then the break-condition is checked. In the While-Loop structure, the break-condition is checked first and then loop instructions are executed if the condition passes. For the tasks of this part, a Do-While-Loop with a counter is more suitable.

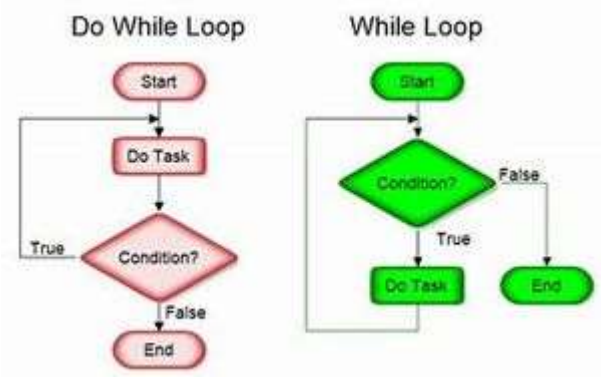

Fig. 1. Loop structure flow charts (image from Wikipedia)

Lab Report materials

The lab report should contain these items for all three versions.

- Show your assignment of registers for variables.
- Show the memory addresses where each operand (in all three arrays) is stored in memory.
- Show the results that are put into the result[] array in memory. Are they all the same?
- Show screen shots of the assembly code and final results in data memory for all three programs.

Part II

In this part, you will gain experience in writing assembly code to call, execute, and return from a function. You will use the `jal` and `jr` instructions to get to and return from a function. You will need to study the registers and adhere to the standards of which registers are saved by the caller and callee. The register conventions are attached on the last page for your convenience. You will also gain experience in the use of the stack. Some points to keep in mind while writing function calls are,

- You will need to save any of the `$tx` registers on the stack that you want to preserve after the function returns.
- Remember to pop those `$tx` back from the stack immediately after the calling function returns.
- Remember to push any of `$s` registers onto the stack you might want to use them inside your function.
- You will then need to pop those `$s` registers back before you execute the `jr $ra` to return.
- You may also need to push and pop the `$ra` register if your function calls yet another function. i.e., if your function is not a "leaf" function.

Program 1

In this first program you can use registers to pass arguments and results. Conventionally, you should use the `$a0…$a4` registers to pass your function arguments.

```
int distance (int a, int b)
{
   int temp;

   if (b > a){
      temp = a;
      a=b;
      b=temp;
      }

  return (a-b)
}

void  main(void)
{
  int var1 =30;
  int var2= 210;

  result = distance (var1, var2);

  return;
}
```

Program 2

In this second program you should pass arguments using the stack. Push the operands to the stack and then pop them back in registers the callee function as required.

```
void swap (int a, int b)
{
   int temp=a;
   a=b;
   b=temp;
}


int distance (int a, int b)
{
   if (b > a)
      swap (a,b);

  return (a-b)
}

void  main(void)
{
  int var1 =30;
  int var2= 210;

  result = distance (var1, var2);

  return;
}
```

Lab Write-up

The lab report should contain following items for this part.

- Your assemble code for both programs
- Screen shots of the SPIM simulator highlighting the points you are referring to in the text

Register Conventions

Register Number	Alternative Name	Description
0	zero	the value 0
1	$at	(**a**ssembler **t**emporary) reserved by the assembler
2-3	$v0 - $v1	(**v**alues) from expression evaluation and function results
4-7	$a0 - $a3	(**a**rguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	$t0 - $t7	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(**s**aved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	$t8 - $t9	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler
28	$gp	**g**lobal **p**ointer. Points to the middle of the 64K block of memory in the static data segment.
29	$sp	**s**tack **p**ointer Points to last location on the stack.
30	$s8/$fp	**s**aved value / **f**rame **p**ointer Preserved across procedure calls
31	$ra	**r**eturn **a**ddress