CSE 378
Quiz Section #2
Coding in MIPS Assembly

Let's translate this C function into assembly:

```
int arr[10] = {1,2,3,4,5,6,7,8,9,10};

int compute_smth() {
      int i, sum=0;
      for (i = 0; i < 10; i++) {
            sum = sum + arr[i];
      }
      printf("result: %d",sum);
      return sum;
}
```

So, now that we have our program, let us convert it to something that would be a bit easier to translate into assembly. Here is the reworked version:

```
int arr[10] = {1,2,3,4,5,6,7,8,9,10};

void main() {
      int i, sum;
      i = 0;
      sum = 0;
      while(i < 10) {
            sum = sum + arr[i];
            i = i+1;
      }
      printf("result: ");
      printf("%d",sum);
}
```

To begin with, you should think about which variables/constants in the program are global and which are local. Global variables will need to be declared in a static data segment, for which you use the ".data" directive. This is typically written before your code, which goes into the ".text" segment. Which variables are global here, and which are local? Integer array arr is global. Static string "result: " is also global – it's a global constant which has to be declared in the .data segment (there's no way to include strings in MIPS instructions!). So how do we actually declare globals? The directive ".data" specifies to the assembler that space for some data needs to be allocated in the "static data" portion of the program memory. Well, great, but how do we actually declare what data it is? Some keywords are helpful:

```
.asciiz "hello" # reserves space for this string, and places
                # 'hello' into it
.space 50       # reserves space for 50 bytes, for general use
.word 17,27     # reserves space for two words (each 4 bytes),
                # and initializes them to 17 and 27.
```

Some others: .byte, .half, .double…

Before any declaration, you can include labels, which will serve as addresses to the data (example in a second).

Note that ALL global variables, and NO non-global variables go into the "static data" section. A local variable, such as i or sum, does not belong here. So, for our global array, we could use the "space" keyword

to reserve enough memory. Since we also want to initialize it to 1-10 (e.g. for testing, etc.), we can also use ".word" and list the ten values separated by commas. Now, let's worry about the string "result: " in the first printf(). That can also be placed into the global data section, using the ".asciiz" keyword. Making the appropriate changes, our program now begins with

```
.data
arr:  .word 1,2,3,4,5,6,7,8,9,10
msg:  .asciiz "result: "

// main function will follow
```

What if we wanted to declare another word below msg? We need to be careful; what is wrong with saying:

```
msg:  .asciiz "asdfg"
num:  .word 42
```

num won't be aligned! ¾ chance you will crash (depending on length of preceding string). Fix? Use ".align" keyword. .align N will align the next data item at a 2^N boundary. So here, you want to use:

```
msg:  .asciiz "asdfg"
      .align 2    # align at the next 4 byte (2^2) boundary
num:  .word 42
```

Next, let us specify (to SPIM) where the main() code is. We need to use the `.text` directive to begin the "code" segment. Everything that comes after it will be the actual executable code, and will get placed into memory as the "text" segment of the program. Note that ALL the code goes into this section. So, our function will now look like this:

```
.data
arr:  .word 1,2,3,4,5,6,7,8,9,10
msg:  .asciiz "result: "

.text
.globl main # declare main function as global, allow calls to it
main:
      int i, sum;
      i = 0;
      sum = 0;
      while(i < 10) {
            sum = sum + arr[i];
            i = i+1;
      }
      printf("result: ");
      printf("%d",sum);

.end main
```

Well, what about the local variables, like i and sum? Where would they go? Simple – we can keep them in temporary-value registers ($t0 through $t7, or $8 through $15). One important rule to always adhere to: when programming in assembly, always always write down what each register represents. If you don't, careless errors WILL happen, and painful debugging will ensue. So, it is a good idea to create something like this somewhere in the file:

```
# variable assignments:
# t0 = i
# t1 = sum
# t2 = constant 10, for comparisons (we need to check when to break out of the while loop)
# t3 = address of array elements (for loading from array)
# t4 = temporary values
```

Great, we're set! Now let's actually begin translating the code. Let's initialize the local variables to their necessary values:

```
.data
arr:   .word 1,2,3,4,5,6,7,8,9,10
msg:   .asciiz "result: "

.text
.globl main # declare main function as global, allow calls to it
main:

       addi $t0, $0, 0           # clear i
       addi $t1, $0, 0           # clear sum
       ori  $t2, $0, 10          # Initializing t2 to its constant value 10
       la   $t3, arr             # load address of array into t4, pseudoinstr

       while(i < 10) {
              sum = sum + arr[i];
              i = i+1;
       }
       printf("result: ");
       printf("%d",sum);

.end main
```

Notice how we initialized $t3 by using the "arr" label – the (base) address of our array. Now, let us write the loop in assembly. Remember, that we will want to break out of it when $t0 is no longer less than $t2.

Important: pay attention on how we walk through the array. We need to compute current element address for each iteration (for loading). Elements are separated by 4 bytes (easy to screw up)! There are a few ways to do this, this one is most efficient (another (see lecture) involves multiplication of i by 4 and adding to base – here, we just add 4 to last element's address to get the current element address).

Also, at the very end of the function, we will want to return back to the caller. Therefore, we will need to jump to where we were called from – information stored in $ra.

```
       .data
       arr: .word 1,2,3,4,5,6,7,8,9,10
       msg: .asciiz      "Result: "

       .text
       .globl main

       # variable assignments:
       # t0 = i
       # t1 = sum
       # t2 = constant 10, for comparisons
       # t3 = address of array elements
       # t4 = temporary values

       main:
       addi $t0, $0, 0   # clear i
       addi $t1, $0, 0   # clear sum
       ori  $t2, $0, 10  # Initializing t2 to its constant value 10
       la   $t3, arr     # load address of array into t4

       loop:
       slt $t4, $t0, $t2 # compare, $t4 = i < sum ? 1 : 0
       beq $t4, $0, end  # if i is not < 10, exit the loop
```

```
lw  $t4, 0($t3)   # load current array element into t4
add $t1, $t1, $t4 # add it to sum
add $t0, $t0, 1   # increment i
add $t3, $t3, 4   # increment current array element pointer
j loop

end:

printf("result: ");
printf("%d",sum);

jr $ra
.end main
```

What about those printfs? How do we get rid of those? We use the `syscall` instruction to perform these operations. To do so, we insert the system call number into register $v0, and its arguments into the $a-registers. And then we issue the "syscall" instruction. More specifically, to print an integer, we use system-call number 1, and set $a0 to the integer we want to print. To print a string, we use system-call number 4, and set $a0 to contain the base address of the string. To read in an integer, we use system-call number 5. The result is returned in register $v0 (and the error code in $v1, so both v0 and v1 are destroy by any syscall!). These are far from the only system calls available – to find out more about them, consult the full listings in the book. So, after applying these changes to our program, it becomes…

```
arr: .word 1,2,3,4,5,6,7,8,9,10
msg: .asciiz      "Result: "

.text
.globl main

# variable assignments:
# t0 = i
# t1 = sum
# t2 = constant 10, for comparisons
# t3 = address of array elements
# t4 = temporary values

main:
addi $t0, $0, 0   # clear i
addi $t1, $0, 0   # clear sum
ori  $t2, $0, 10  # Initializing t2 to its constant value 10
la   $t3, arr     # load address of array into t4

loop:
slt $t4, $t0, $t2 # compare, $t4 = i < sum ? 1 : 0
beq $t4, $0, end  # if i is not < 10, exit the loop
lw  $t4, 0($t3)   # load current array element into t4
add $t1, $t1, $t4 # add it to sum
add $t0, $t0, 1   # increment i
add $t3, $t3, 4   # increment current array element pointer
j loop

end:

addi $v0, $0, 4   # Now we print out result: string
la $a0, msg
syscall
```

```
addi $v0, $0, 1   # followed by the actual sum (which is in t1)
add $a0, $t1, $0
syscall

jr $ra
.end main
```

And this is the full assembly version of our simple C program above.