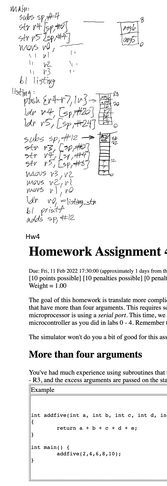

Hw4

Homework Assignment 4

Due: Fri, 11 Feb 2022 17:30:00 (approximately 1 days from the time this page was loaded)
[10 points possible] [10 penalties possible] [0 penalties graded so far]
Weight = 1.00

The goal of this homework is translate more complicated C functions and subroutines that contain stack-allocated data into ARM Cortex-M0 assembly language. You will also experiment with functions that have more than four arguments. This requires some additional understanding of how the stack works as well as how to reserve and free space on the stack. One of the easiest ways to interact with a microprocessor is using a *serial port*. This time, we give you some initialization code (stdio.s) that you can use to configure a serial port of the STM32. Connect the USB-to-serial adapter to your microcontroller as you did in labs 0 - 4. Remember that detailed instructions on how to use serial communication can be found on the serial port setup page.

The simulator won't do you a bit of good for this assignment since we don't have a serial port or terminal emulator for it.

More than four arguments

You've had much experience using subroutines that take at most four arguments that are passed in R0 - R3. Subroutines with more than four arguments are still called with the first four arguments in R0 - R3, and the excess arguments are passed on the stack. For example, consider the following function that accepts five arguments, and a main function that calls it:

Example	Assembly Translation	Call it like this:

`int addfive(int a, int b, int c, int d, int e)`
`{`
`  return a + b + c + d + e;`
`}`
 	
`int main() {`
`  addfive(2,4,6,8,10);`
`}`
	`.global addfive`
	`addfive:`
	`  adds r0, r1  // a += b`
	`  adds r0, r2  // a += c`
	`  adds r0, r3  // a += d`
	`  ldr  r1, [sp,#0]  // Load e from the stack.`
	`  adds r0, r1  // a += e`
	`  bx lr  // Return`
		`.global main`
		`main:`
		`  movs r0,#10  // Argument 5 is 10`
		`  sub  sp,#4  // Allocate 4 bytes on stack.`
		`  str r0,[sp,#0] // Store 5 on the stack.`
		`  movs r0,#2  // Argument 1 is 2`
		`  movs r1,#4  // Argument 2 is 4`
		`  movs r2,#6  // Argument 3 is 6`
		`  movs r3,#8  // Argument 4 is 8`
		`  bl addfive  // Call addfive()`
		`  // IMPORTANT: Deallocate 4 bytes from stack.`
		`  add  sp,#4`
		`  wfi`

You know that PUSH and POP make the stack grow downward and shrink upward, respectively. The amount of the change is proportional to the number of registers saved and restored—four bytes per register. We can do the same with subroutines, but must do so with one rule: Any called subroutine must restore the stack to the same it was when it was called. Similarly, the caller may allocate space on the stack before calling a function as it did in the example above.

This manner of passing more than four arguments is completely compatible with saving and restoring registers as well. We only need to keep track of how many additional space is allocated on the stack with the saved registers. For example, we could rewrite the example, above, with PUSH and POP, and we can (uselessly) save R4 – R7:

Example	Assembly Translation	Call it like this:

`int addfive(int a, int b, int c, int d, int e)`
`{`
`  return a + b + c + d + e;`
`}`
 	
`int main() {`
`  addfive(2,4,6,8,10);`
`}`
	`.global addfive`
	`addfive:`
	`  push {r4-r7,lr}`
	`  adds r0, r1  // a += b`
	`  adds r0, r2  // a += c`
	`  adds r0, r3  // a += d`
	`  ldr  r1, [sp,#20]  // Load e from the stack.`
	`  adds r0, r1  // a += e`
	`  pop  {r4-r7,pc}  // Return`
		`.global main`
		`main:`
		`  movs r0,#10  // Argument 5 is 10`
		`  sub  sp,#4  // Allocate 4 bytes on stack.`
		`  str r0,[sp,#0] // Store 5 on the stack.`
		`  movs r0,#2  // Argument 1 is 2`
		`  movs r1,#4  // Argument 2 is 4`
		`  movs r2,#6  // Argument 3 is 6`
		`  movs r3,#8  // Argument 4 is 8`
		`  bl addfive  // Call addfive()`
		`  // IMPORTANT: Deallocate 4 bytes from stack.`
		`  add  sp,#4`
		`  wfi`

Allocating space on the stack for local variables

A caller allocates space on the stack for extra arguments. A subroutine can allocate space on the stack for a *local variable* such as an array. To do so, we need only keep track of how much space was allocated and deallocate it before returning. For example, consider the following example:

Example	Assembly Translation	Call it like this:

`int addfive(int a, int b, int c, int d, int e)`
`{`
`  int x;`
`  int sum = 0;`
`  int array[100];`
`  for(x=0; x < 100; x++)`
`    array[x] = x;`
`  for(x=a; x <= b; x++)`
`    sum += array[x];`
`  sum += a + b + c + d + e;`
`  return sum;`
`}`
 	
`int main() {`
`  addfive(2,4,6,8,10);`
`}`
	`.global addfive`
	`addfive:`
	`  push {r4-r7,lr}`
	`  sub  sp,#400  // Allocate 100 integers`
	`  mov  r7,sp  // R7 is the beginning od "array"`
	`  movs r5,#0  // Let R5 be "sum"`
	`  movs r4,#0  // Let R4 be "x"`
	`for1:`
	`  cmp  r4,#100`
	`  beq done1`
	`  lsls r6,r4,#2  // R6 is 4*x`
	`  str  r4,[r7,r6]  // array[x] = x`
	`  adds r4,#1  // x++`
	`  b for1`
	`done1:`
	`  movs r4, r0  // x = a`
	`for2:`
	`  cmp  r4, r1  // x <= b?`
	`  bgt done2`
	`  lsls r6,r4,#2  // r6 = r4 * 4`
	`  ldr  r6,[r7,r6]  // r6 = array[x]`
	`  adds r5,r6  // sum += array[x]`
	`  adds r4,#1  // x++`
	`  b for2`
	`done2:`
	`  adds r5, r0  // sum += a`
	`  adds r5, r1  // sum += b`
	`  adds r5, r2  // sum += c`
	`  adds r5, r3  // sum += d`
	`  ldr  r1, [sp,#420] // Load e from the stack.`
	`  adds r5, r1  // a += e`
	`  movs r0, r5  // Want to return value in r5`
	`  add  sp,#400  // Deallocate 100 integers`
	`  pop  {r4-r7,pc}  // Return`
		`.global main`
		`main:`
		`  movs r0,#10  // Argument 5 is 10`
		`  sub  sp,#4  // Allocate 4 bytes on stack.`
		`  str r0,[sp,#0] // Store 5 on the stack.`
		`  movs r0,#2  // Argument 1 is 2`
		`  movs r1,#4  // Argument 2 is 4`
		`  movs r2,#6  // Argument 3 is 6`
		`  movs r3,#8  // Argument 4 is 8`
		`  bl addfive  // Call addfive()`
		`  // IMPORTANT: Deallocate 4 bytes from stack.`
		`  add  sp,#4`
		`  wfi`

One important thing to keep in mind with stack allocation is that the SP register can only be incremented or decremented by 4. If you wanted to make space for an array of characters such as `char buffer[13];` then you would need to round **up** the size of the allocation to **16**. The fundamental reason for this is that the lower two bits of the SP register are always zero. The special SUB and ADD instructions that work with SP also only work with values divisible by four.

Standard I/O on the STM32

One of the best reasons to use the serial port is that the STM32 Standard Peripheral firmware allows you to use it with the C Standard I/O library calls. The stdio.s functions set these up to work with the serial port. As you know, one of the most common C functions to call is **`printf()`**. Now, we can call it in assembly language. For instance:

```
.global main
main:
        bl serial_init
        ldr r0,=greeting
        bl printf
        wfi
greeting:
        .string "Hello, World.\n"
        .balign 2 // Align anything after this.
```

Of course, the primary reason we use **`printf()`** is so that we can print the values of variables in a formatted manner. The firmware for the STM32 is no less capable:

```
.global main
main:
        bl serial_init
        ldr r0,=format
        movs r1,#14
        ldr r2,=0xdeadfabd
        bl printf
        wfi
format:
        .string "The value of x=%03d.  Beverage preference is 0x%08x!\n"
        .balign 2
```

Each of the following exercises are independent of the others, but you should write and test all of your functions in a single .s file in System Workbench. A code skeleton has been provided for you in the **hw4.s** file. These subroutines will be called by a C program, so you should **definitely** be sure that you don't overwrite R4 - R11 without saving them first. Each subroutine should be prefaced with a *.global subroutine_name* to make it visible to other modules. An example C that calls each function is provided for you in the **main.c** file. This time, when you are done, you can submit the entire **hw4.s** file that contains all of your subroutines.

You should assume, for each subroutine, that **`serial_init`** has already been called for you by **`main()`**. You don't have to do that in your code.

Q1: hello [1 point]

Translate ("hand compile") the following C function into assembly language. First, let's just make sure that Standard I/O is working and you have your serial port wired correctly. What we expect to see here is an assembly language subroutine that sets up the first and second arguments as strings and does a **`bl printf`**.

When you implement this, you should put the **`login`** string, as well as the format string (the first argument to printf), in the .text segment. Be sure to put a .balign 2 between any string and any assembly language instructions. Character strings can start on any address and have any length, but each instruction must reside at an address that is evenly divisible by 2.

```
const char login[] = "xyz";
void hello(void) {
  printf("Hello, %s!\n", login);  // Here, printf is given two arguments
}
```

Q2: showsub2 [1 point]

Translate ("hand compile") the following C function into assembly language. More difficult this time. The first argument given to printf() is the string with the format statements. The second argument will be the value of the 'a' variable (which was passed in as argument 1). The fourth argument will be the product of the 'a' and 'b' variables. Make sure that the spacing in the format string exactly the way it is presented below.

```
void showsub2(int a, int b)
{
  printf("%d - %d = %d\n", a, b, a-b); // Here, printf is given four args
}
```

Q3: showsub3 [1 point]

Translate ("hand compile") the following C function into assembly language. Since there are five arguments, you'll need to put one of them in stack memory.

```
void showsub3(int a, int b, int c)
{
  printf("%d - %d - %d = %d\n", a, b, c, a-b-c); // five args
}
```

Q4: listing [1 point]

Translate ("hand compile") the following C function into assembly language.

```
void listing(const char *school, int course, const char *verb,
             int enrollment, const char *season, int year)
{
  // You need to allocate space on the stack to hold some of the
  // seven parameters to printf().  Then move the arguments around
  // to call printf() with the proper arguments.
  printf("%s %03d %s %d students in %s, %d\n",
         school, course, verb, enrollment, season, year);
}
```

Q5: trivial [1 point]

Translate ("hand compile") the following C function into assembly language. This function requires you to allocate a 100-word on the stack.

```
int trivial(unsigned int n)
{
  int tmp[100]; // a stack-allocated array of 100 words
  for(int x=0; x < sizeof tmp / sizeof tmp[0]; x += 1)
    tmp[x] = x+1;
  if (n >= sizeof tmp / sizeof tmp[0])
    n = sizeof tmp / sizeof tmp[0] - 1;
  return tmp[n];
}
```

Q6: depth [1 point]

Translate ("hand compile") the following C function into assembly language. The **`depth()`** subroutine is an example of a *recursive* function. It calls itself. Although it is possible to create a version of this function that is not recursive, we want you to create a faithful recursive assembly language translation.

There are two important things that must be done when making a recursive function:

- There must be a decision mechanism that stops the recursion from going on forever. Here, the first argument given to depth() is decremented by one each time it calls itself. When it is called with a zero, it returns immediately without making further calls to itself.
- Since registers like R0 may be modified by each call to a recursive function, it is important to save the values still needed after the call returns. For instance, in the call below, R0 is replaced by the return value of each recursive invocation. It is also replaced by the call to **`strlen()`**. It is important to save R0, which is the a argument value. One way of saving it is to put it in a high register such as R4 and be sure to push/pop it on each invocation. Another way is to allocate space on the stack and store it there.

```
int depth(int x, const char *s)
{
  int len = strlen(s);
  if (x == 0)
    return len;
  puts(s);
  return len + depth(x-1, s);
}
```

Q7: collatz [1 point]

Translate ("hand compile") the following C function into assembly language. This function counts the steps of the Collatz Conjecture. This famous problem has many other names. It is impossible to prove a closed-form solution, but it is easy to compute. Don't call this with zero.

```
unsigned int collatz(unsigned int n) {
  if (n == 1)
    return 0;
  if (in & 1) == 0)
    return 1 + collatz(n/2);
  return 1 + collatz(3*n+1);
}
```

Q8: permute [1 point]

Translate ("hand compile") the following C function into assembly language. This is difficult because you must store two of the arguments on the stack, and it is recursive, so the stack is always growing.

```
int permute(int a, int b, int c, int d, int e, int f) {
    if (a <= 0)
        return f + e + d + c + b + a;
    return permute(f-1,a,b,c,d,e) + 1;
}
```

Q9: bizarre [1 point]

Translate ("hand compile") the following C function into assembly language. Here, a pointer to a function is being passed into **`qsort()`**. The **`qsort()`** function exists in the standard library -- like **`printf()`**. It sorts an array of n entries, each of which are m bytes in size, given a *comparison function* That means that **`qsort()`** will call it using a **`bx`** instruction at some point. The address must be an odd number so that it is regarded as Thumb code.

```
int compare(const void *a, const void *b) {
    return *(int*)a - *(int*)b;
}
int bizarre(int base, int nth) {
    int array[200];
    int x;
    for(x=0; x < 200; x++)
        array[x] = ((base+x*1) * 255) & 0xff;
    qsort(array, 200, 4, compare);
    return array[nth];
}
```

Hint: Here's the **`compare()`** function for you:

```
.type compare, %function  // You know why you need this line, right?
compare:
    ldr  r0,[r0]
    ldr  r1,[r1]
    subs r0,r1
    bx lr
```

Q10: easy [1 point]

Translate ("hand compile") the following C function into assembly language. This subroutine is easy. That's it's name. Break it down into its components, and implement one piece at a time.

It's not really that difficult. It's just tedious, and very good practice.

```
int easy(int a, int b, int shift, int mask, int skip)
{
    int save[64];
    int result = 0;
    save[0] = 1;
    for(int x=1; x < sizeof save / sizeof save[0]; x += 1)
        save[x] = (save[x-1] * (a & mask)) << shift;
    for(int x=a; x!=b; x = (x * 5 + skip) & 63)
        result ^= save[x];
    return result;
}
```

Academic Honesty Statement [0 ... -10 points]

By typing my name, below, I hereby certify that the work on this homework is my own and that I have not copied the work of any other student (past or present) while completing it. I understand that if I fail to honor this agreement, I will receive a score of zero for the assignment, a one letter drop in my final course grade, and be subject to possible disciplinary action.

Tzu Yu Chen

Save