Homework Assignment 3.1

Due: Wed, 16 Sep 2020 23:59:59 (approximately 85 days ago)
[Score: 10 / 10 points possible]
Weight = 0.5

The goal of this homework is translate meaningful C subroutines and functions into equivalent ARM Cortex-M0 assembly language. We often refer to this as "hand compliation". And the expectation is that you will do this by hand. Although it is possible to use GCC to compile these programs into assembly language, it will be immediately obvious to a grader that you've done so. You won't get credit for such submissions. Beyond that, these exercises will be graded on whether or not they work. We will assemble them and run them. They must do what they are supposed to to get the points they are worth. No partial credit.

The Application Binary Interface

Keep in mind the three basic rules about the ABI:

- When a subroutine is called, the expectation is that the first four arguments will be in R0, R1, R2, and R3. (If the subroutine accepts more than four arguments, then arguments five and beyond are pushed onto the stack before the subroutine is called. We'll try that in later exercises.)
- When a subroutine returns, the value it returns is expected to be in R0.
- A subroutine may overwrite R0, R1, R2, R3, and R12, but the calling subroutine demands that all the other registers are the same upon return as they were just before calling the subroutine. If you intend to change the value of R4 in a subroutine, you should use **`PUSH {R4,LR}`** at the beginning of the subroutine when you push LR. You can restore it and return to the caller with **`POP {R4,LR}`**.

Most of the following exercises are independent of the others. You can write all of your functions in a single .s file in System Workbench or in the simulator and test them there. If you want to test them to make sure they work when they are called from a C function, you can create a main() function in a C program in System Workbench. Your translated subroutines must still be in a separate .s file. I.e., you cannot put assembly language directly into a C file. Each subroutine should be prefaced with a *.global subroutine_name* to make it visible to other modules. For instance:

Example	Your Translation	You can call it like this:

`void nothing(void) {`
`}`
	`.global nothing`
	`nothing:`
	`  push {lr} // let's use push...`
	`  pop  {pc} // ...and pop for this example`
		`void nothing(void);`
		`int main(void) {`
		`  nothing();`
		`  return 0;`
		`}`

`int always_six(void) {`
`  return 6;`
`}`
	`.global always_six`
	`always_six:`
	`  push {lr}`
	`  movs r0,#6`
	`  pop  {pc}`
		`int always_six(void);`
		`int main(void) {`
		`  int check = always_six();`
		`  // you can examine 'check' to`
		`  // tell that the result is 6.`
		`  return 0;`
		`}`

`int add3(int x) {`
`  return x + 3;`
`}`
	`.global add3`
	`add3:`
	`  adds r0,#3`
	`  // as long as we don't call anything`
	`  // else in a subroutine, we can use`
	`  // the bx lr instruction instead of`
	`  // push {lr} and pop {pc}.`
	`  bx lr`
		`int add3(void);`
		`int main(void) {`
		`  int check = add3(5);`
		`  // you can examine check to`
		`  // tell that the result is 8.`
		`  return 0;`
		`}`

`int max(int a, int b) {`
`  if (a > b)`
`  return a;`
`  return b;`
`}`
	`.global max`
	`max:`
	`  cmp r0,r1`
	`  ble returnb`
	`  // we return r0.  That's already the max.`
	`  bx lr // so just return`
	`returnb:`
	`  movs r0,r1 // return value of r1 in r0`
	`  bx lr`
		`void max(int,int);`
		`int main(void) {`
		`  int check = max(12,42);`
		`  // you can examine check to`
		`  // tell that the result is 42.`
		`  return 0;`
		`}`

`void zeroptr(int *x) {`
`  *x = 0;`
`}`
	`.global zeroptr`
	`zeroptr:`
	`  movs r1,#0 // It's OK to overwrite r1`
	`  str r1,[r0]`
	`  bx lr`
		`void zeroptr(int *);`
		`int main(void) {`
		`  int value = 42;`
		`  zeroptr(&value);`
		`  // you can examine value to`
		`  // check that it is now zero.`
		`  return 0;`
		`}`

For each translated subroutine, you need only provide the assembly language for it, much like is shown in the "Your Translation" column in the table above.

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.

Raghuram Selvaraj 

Implement the following C subroutines [10 points]

```
.cpu cortex-m0
```

1. nine [1 point]

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

```
// Just return the value 9.
int nine(void) {
        return 9; // Remember: return value in R0
}
```

2. first [1 point]

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

```
// Return the single argument passed in.
int first(int x) { // Remember: first argument is in R0
        return x;
}
```

3. add4 [1 point]

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

```
// Add the four arguments passed in.
int add4(int a, int b, int c, int d) {
        return a + b + c + d;
}
```

4. or_into [1 point]

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

If you have not seen a pointer in C before, try not to panic here. A pointer is just an address, and you know how to load from and store to addresses. If the point is in R0, then you can load its value into R2 with the instruction ldr r2,[r0], and you can store the value of r2 into that address with str r2,[r0].

```
// OR the bits of x into the pointed-to-value.
void or_into(int *ptr, int x) {
        *ptr = *ptr | x;
}
```

5. getbit [1 point]

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

```
// Return 1 if bit n of the pointed-to-value is set.  Else return 0.
int getbit(int *ptr, int n) {
        // Shift right by n bits, then AND with 1.
        return (*ptr >> n) & 1;
}
```

6. setbit [1 point]

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

```
// Set bit n of pointed-to-value to 1.
void setbit(int *ptr, int n) {
        // Shift '1' left by n bits, then OR with value.
        *ptr = *ptr | (1 << n);
}
```

7. clrbit [1 point]

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.global nine
nine:
    push {lr}
    movs r0, #9
    pop {pc}

.global first
first:
    push {lr}
    pop {pc}

.global add4
add4:
    push {lr}
    adds r2, r3
    adds r1, r2
    adds r0, r1
    pop {pc}

.global or_into
or_into:
    push {lr}
    ldr r2, [r0]
    orrs r2, r1
    str r2, [r0]
    pop {pc}

.global getbit
getbit:
    push {lr}
    ldr r2, [r0]
    asrs r2, r1
    movs r0, #1
    ands r0, r2
    pop {pc}

.global setbit
setbit:
    push {lr}
    ldr r2, [r0]
    movs r3, #1
    lsls r3, r1
    orrs r2, r3
    str r2, [r0]
    pop {pc}

.global clrbit
clrbit:
    push {lr}
    ldr r2, [r0]
    movs r3, #1
    lsls r3, r1
    bics r2, r3
    str r2, [r0]
    pop {pc}

.global inner
inner:
    push {lr}
    adds r0, #9
    pop {pc}

.global outer
outer:
    push {r4, lr}
    movs r4, r0
    lsls r0, r0, #1
    bl inner
    adds r0, r4
    pop {r4, pc}

.global set2
set2:
    push {lr}
    movs r1, #2
    bl setbit
    pop {pc}

.global get4
get4:
    push {lr}
    movs r1, #4
    bl getbit
    pop {pc}


// We put this down here just in case
// you forget to return from a subroutine.
// If you hit this, you'll know you forgot.
bkpt
```

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

```
// Clear bit n of the pointed-to-value.
// Leave the rest of the bits the same!
void clrbit(int *ptr, int n) {
        // Shift '1' left by n bits,
        // invert the shifted 1 << n,
        // then AND with value.
        *ptr = *ptr & ~(1 << n);
}
```

---

8. inner/outer [1 point]

Translate ("hand compile") the following C functions into assembly language. Note that you will need to use PUSH/POP for the "outer" function, below. You should also push an extra register to save argument 1. Remember the rules: A subroutine is always allowed to modify R0 - R3, and you should expect it to do so. A subroutine must always return with the same values for R4 - R7 that it was called with. Otherwise, the caller may not function properly.

```
// This function is called by outer().
int inner(int x) {
        return x + 9;
}
int outer(int x) {
        return x + inner(x*2);
}
```

---

9. set2 [1 point]

Translate ("hand compile") the following C subroutine into assembly language. (It uses **setbit**, which you write above .)

```
// Use setbit() to set bit 2 of *ptr.
void set2(int *ptr) {
        setbit(ptr, 2);
}
```

---

10. get4 [1 point]

Translate ("hand compile") the following C function into assembly language. (It uses **getbit**, which you write above .)

```
// Use getbit() to get bit 4 of *ptr.
int get4(int *ptr) {
        return getbit(ptr, 4);
}
```

You can try the code you wrote in the simulator using the link at the bottom of the page. Basic testcases are built in to the simulator. When all tests are passed, the numbers

```
10 09 08 07 06 05 04 03 02 01
```

should appear in the upper right area of the Memory Browser. If any the first ten memory bytes are shown as '00', the testcase for that problem failed.

Remember to copy things back into the textbox if you make modifications to your code in the simulator.

Save first. Then click here to try it in the simulator