Hw3

Saturday, February 5, 2022 2:07 PM

Homework Assignment 3

Due: Fri, 04 Feb 2022 17:30:00 (approximately 20 hours ago)
[12 points possible] [10 penalties possible] [0 penalties graded so far]
Weight = 0.50

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 in the simulator. 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,PC}`**.

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.

Tzu Yu Chen

Implement the following C subroutines [12 points]

1. twentyone [1 point]

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

```
// Just return the value 21.
int twentyone(void) {
        return 21; // 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. comp4 [1 point]

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

```
// Compute an expression for the four arguments passed in.
int comp4(int a, int b, int c, int d) {
        return a * b - a * c + a * 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]

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 - 4;
}
int outer(int x) {
        return x * inner(x+3) * inner(x+5);
}
```

9. set4 [1 point]

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

```
// Use setbit() to set bit 4 of *ptr.
void set4(int *ptr) {
        setbit(ptr, 4);
}
```

10. get6 [1 point]

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

```
// Use getbit() to get bit 6 of *ptr.
int get6(int *ptr) {
        return getbit(ptr, 6);
}
```

11. largest_weird [1 point]

Translate ("hand compile") the following C functions into assembly language. Remember to obey the ABI rules for this and all other exercises. In particular, if largest_weird() changes the values of R4-R7 that were set by the caller, it's incorrect. Also, be sure to create two functions for this exercise. The largest_weird function must call the weird function. It's certainly possible to write largest_weird without using two functions, but that's not what we're asking for here, and it won't get credit.

```
int weird(int x, int y) {
        return (x*3) & ~(y*5);
}
int largest_weird(int x, int y) {
        // You will need lots of registers to implement this function.
        // Use PUSH to save R4,R5,R6,R7 along with LR.
        // Use R4,R5,R6,R7 for a, b, y, and big.
        // Upon return, POP R4,R5,R6,R7 along with PC.
        int big = 0;
        for(int a=x; a <= y; a++)
                for(int b=a; b <= y; b++) {
                        int temp = weird(a,b);
                        if (temp > big)
                                big = temp;
                }
        return big;
}
```

12. mulvec [1 point]

Translate ("hand compile") the following C function into assembly language. This function multiplies one "vector" (b) by a constant (m) and writes the result to another vector (a). Both vectors are of size n. The sum of all the vector entries is returned. You may not modify the elements of b[] because it is declared as "const". This array may be in flash memory, and an attempt to modify it would cause a HardFault.

```
int mulvec(int a[], const int b[], int n, int m) {
        int sum = 0;
        for(int i=0; i < n; i++) {
                a[i] = m * b[i];
                sum += a[i];
        }
        return sum;
}
```

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

```
12 11 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 'ff', 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