Homework Assignment 3.2

Due: Wed, 23 Sep 2020 23:59:59 (approximately 78 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.

We're getting to the point in the semester when a sense of desperation sets in for some students, and the decision is made to copy solutions from other students. This is a difficult assignment for the purpose of ensuring that you are prepared for a subsequent exam that will seem easy by comparison. If you need help, ask a teaching assistant rather than another student. In any case, a zero on (half of) a homework score is preferable to a letter-grade drop for the class.

Raghuram Selvaraj ✓

Implement the following C subroutines [10 points]

11. sumsq [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 sumsq() 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 sumsq function must call the sq function. It's certainly possible to write sumsq without using two functions, but that's not what we're asking for here, and it won't get credit.

```
int sq(int x) {
        return x*x;
}
int sumsq(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 x, y, sum, and n.
        // Upon return, POP R4,R5,R6,R7 along with PC.
        int sum = 0;
        for(int n=x; n <= y; n++)
                sum += sq(n);
        return sum;
}
```

12. log2 [1 point] ✓

Translate ("hand compile") the following C function into assembly language. The log2 function returns the value $\lfloor \log_2(n) \rfloor$. It's not the most efficient way to do this.

```
int log2(unsigned int n) {
        int lg = -1; // same thing as 0xffffffff
        while(n > 0) {
                n = n >> 1;
                lg = lg + 1;
        }
        return lg;
}
```

13. divide [1 point] ✓

Translate ("hand compile") the following C function into assembly language. This division algorithm takes ***unsigned*** numerator and denominator values and returns the unsigned integer quotient. It's also not the very fastest way to do this, but still much better than the extremely slow algorithm presented in the lecture notes. Note that it relies on the `log2` function described above.

```
unsigned int divide(unsigned int n, unsigned int d) {
        if (d == 0)
                return 0xffffffff; // Let this represent divide-by-zero error.
        if (d > n)      // CAREFUL: unsigned comparison
                return 0;
        int nbits = log2(n);
        int dbits = log2(d);
        int shift = nbits - dbits;
        d = d << shift; // shift the denominator over to size of numerator
        unsigned int q = 0;   // q is the quotient
        for (  ; shift >= 0; shift -- ) {
                // If you look carefully, you'll recognize this as the
                // standard way of dividing by trial subtraction.
                q = q << 1;
                if (d <= n) { // CAREFUL: unsigned comparison
                        n = n - d;
                        q = q | 1;
                }
                d = d >> 1;
        }
        return q;
}
```

14. itersqrt [1 point] ✓

Translate ("hand compile") the following C function into assembly language. This function is an iterative algorithm to compute the integer square root, $\lfloor \sqrt{n} \rfloor$. For obvious reasons, it only

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

.global sq
sq:
    push {lr}
    movs r1, r0
    muls r0, r1
    pop {pc}

.global sumsq
sumsq:
    push {r4-r7, lr}
    movs r4, r0
    movs r5, r1

    movs r6, #0         // sum
    for1:
        movs r7, r4     // n
    check1:
        cmp r7, r5
        bgt endfor1
    body1:
        movs r0, r7
        bl sq
        adds r6, r0
    next1:
        adds r7, #1
        b check1
    endfor1:
    movs r0, r6
    pop {r4-r7, pc}

.global log2
log2:
    push {lr}
    ldr r1, =0xffffffff
    while1:
        cmp r0, #0
        bls endwhile1
    do1:
        lsrs r0, r0, #1
        adds r1, #1
        b while1
    endwhile1:
        movs r0, r1
    pop {pc}

.global divide
divide:
    push {r4-r7, lr}
    cmp r1, #0
    beq divzero

    cmp r1, r0
    bhi retzero

    movs r4, r0             // n
    movs r5, r1             // d
    bl log2
    movs r6, r0             // nbits
    movs r0, r5
    bl log2
    movs r7, r0             // dbits

    movs r0, r6
    subs r0, r7             // shift

    lsls r5, r5, r0
    movs r1, #0             // q

    for2:
    check2:
        cmp r0, #0
        blt endfor2
    body2:
        lsls r1, r1, #1
        if1:
            cmp r5, r4
            bgt endif1
        then1:
            subs r4, r4, r5
            movs r2, #1
            orrs r1, r2
        endif1:
        lsrs r5, r5, #1
    next2:
        subs r0, #1
        b check2
    endfor2:
        movs r0, r1
```

accepts unsigned integers. (Warning: Don't spend too much time trying to understand *how* this works unless you're genuinely curious.)

```
unsigned int itersqrt(unsigned int n) {
        if (n < 2) // This is the same as: if (n <= 1)
                return n;
        int shift = log2(n) + 1;
        shift += shift & 1; // round up to next multiple of 2
        int result = 0;
        do {
                shift -= 2;
                result = result << 1;
                result = result | 1;
                if ((result * result) > (n >> shift))   // CAREFUL: unsigned
                        result = result ^ 1;
        } while (shift > 0);
        return result;
}
```

15. recsqrt [1 point]

Translate ("hand compile") the following C function into assembly language. This is a recursive version of **itersqrt()**. If you were not interested in understanding how itersqrt worked, staring at this one too long may cause blindness.

This function is, at least, pleasantly short. This demonstrates the compact representation of many recursively-specified functions.

This is an example of a subroutine where you will need to save R0 in a separate register, recursively invoke itself, and multiply the result of the recursion by the previously saved value of R0.

```
unsigned int recsqrt(unsigned int n) {
        if (n < 2) // This is the same as: if (n <= 1)
                return n;
        int smaller = recsqrt(n >> 2) << 1;
        int larger = smaller + 1;
        if (larger * larger > n)
                return smaller;
        else
                return larger;
}
```

16. fun16 [1 point]

Translate ("hand compile") the following C function into assembly language. You'll need to save R0 again.

```
unsigned int fun16(unsigned int n) {
        if (n < 3) // This is the same as: if (n <= 2)
                return n - 1;
        return n * fun16(n-1) - 1;
}
```

17. doublerec [1 point]

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

```
unsigned int doublerec(unsigned int n) {
        if (n < 3)      // This part is easy.
                return n;
        // And this part is not easy...
        // Now let's think about this:
        // You need to call doublerec twice,
        // save the result of the first call
        // so it's not overwritten by the second,
        // and then add the two results together.
        // Do it like this:
        // Push R4 and R5, then copy n into R4 and
        // save the result of the first call in R5.
        // Pop R4 and R5 when you're done.
        return doublerec(n>>1) + doublerec(n>>2);
}
```

18. vectoradd [1 point]

```
        b returndiv

    divzero:
        ldr r0, =0xffffffff
            b returndiv
    retzero:
        movs r0, #0
        b returndiv
    returndiv:
        pop {r4-r7, pc}

.global itersqrt
itersqrt:
    push {r4, lr}

    cmp r0, #1
    bls retitsq

    movs r4, r0
    bl log2
    adds r0, #1

    movs r2, #1
    movs r1, r0
    ands r1, r2
    adds r0, r1                 // shift

    movs r1, #0                 // result
    do2:
        movs r2, #1
        subs r0, #2
        lsls r1, r1, #1
        orrs r1, r2
        if2:
            movs r3, r1
            muls r3, r1
            movs r2, r4
            lsrs r2, r0
            cmp r3, r2
            bls endif2
        then2:
            movs r2, #1
            eors r1, r2
        endif2:
    while2:
        cmp r0, #0
        bls endwhile2
        b do2
    endwhile2:
        movs r0, r1

    retitsq:
        pop {r4, pc}

.global recsqrt
recsqrt:
    push {r4, lr}

    cmp r0, #1
    bls retrecsq

    movs r4, r0
    lsrs r0, r0, #2
    bl recsqrt
    lsls r0, r0, #1

    movs r1, r0
    adds r1, #1

    if3:
        movs r2, r1
        muls r2, r1
        cmp r2, r4
        bls else3
    then3:
        b retrecsq
    else3:
        movs r0, r1
        b retrecsq
    endif3:

    retrecsq:
        pop {r4, pc}

.global fun16
fun16:
    push {r4, lr}
    if4:
        cmp r0, #3
        bhi endif4
    then4:
        subs r0, #1
        b retfun16
    endif4:
        movs r4, r0
        subs r0, #1
        bl fun16
```

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

```
// In a subroutine argument 'int a[]' means the same thing as 'int *a'.
void vectoradd(int count, int z[], const int x[], const int y[]) {
        for(int n=0; n < count; n++)
                z[n] = x[n] + y[n];
}
```

---

19. updatesum [1 point]

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

```
int updatesum(int x[], int count) {
        int sum = 0;
        for(int n=0; n < count; n++) {
                sum += x[n];
                x[n] += 1;
        }
        return sum;
}
```

---

20. globalsum [1 point]

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

```
const int global_array[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
int globalsum(int count) {
        int sum = 0;
        for(int n=0; n < count; n++) {
                sum += global_array[n];
        }
        return sum;
}
```

```
        muls r0, r4
        subs r0, #1
    retfun16:
        pop {r4, pc}

.global doublerec
doublerec:
    push {r4, r5, lr}
    if5:
        cmp r0, #3
        bhi endif5
    then5:
        b retdorec
    endif5:
        movs r4, r0
        lsrs r0, r0, #1
        bl doublerec
        movs r5, r0
        movs r0, r4
        lsrs r0, r0, #2
        bl doublerec
        adds r0, r5
    retdorec:
        pop {r4, r5, pc}

.global vectoradd
vectoradd:
    push {r4-r7, lr}
    for3:
        movs r4, #0                     // n
    check3:
        cmp r4, r0
        bge endfor3
    body3:
        movs r7, r4
        lsls r7, r7, #2
        ldr r5, [r2, r7]
        ldr r6, [r3, r7]
        adds r5, r6
        str r5, [r1, r7]
    next3:
        adds r4, #1
        b check3
    endfor3:
    pop {r4-r7, pc}

.global updatesum
updatesum:
    push {r4, r5, lr}
    movs r2, #0
    for4:
        movs r3, #0
    check4:
        cmp r3, r1
        bge endfor4
    body4:
        movs r5, r3
        lsls r5, r5, #2
        ldr r4, [r0, r5]
        adds r2, r4
        adds r4, #1
        str r4, [r0, r5]
    next4:
        adds r3, #1
        b check4
    endfor4:
        movs r0, r2
    pop {r4, r5, pc}

.global globalsum
globalsum:
    push {r4, r5, lr}
    movs r1, #0
```

Save first, then click here to try it in the simulator

Remember to copy any changes you make in the simulator back into the textbox on this page.