Module 1-D

Practical Assembly Language Programming

Tim Rogers 2022

Reading

- Textbook, Chapter 7,”Structured Programming”, pages 133 – 160.
    - Today’s topic.
- Textbook, Chapter 8, “Subroutines”, pages 161 – 202.
    - Also today – hopefully…

Reading Ahead

- Textbook, Chapter 10, “Mixing C and Assembly”, pages 215 – 236.
  - We’ll talk about this soon.
- Textbook, Chapter 14, “General Purpose I/O (GPIO)”, pages 341 – 372.
  - We’ll talk about this in the next lecture module.
- Textbook, Chapter 11, “Interrupts”, pages 237 – 268.
  - We’ll talk about this later on.

Learning Outcome #1

"An ability to program a microcontroller to perform various tasks"

How?

A. Architecture and Programming Model
B. Instruction Set Basics
C. Addressing modes and conditionals
D. Practical Assembly Language Programming
E. Stacks and Recursive Functions*

Assembler Directives

Directive	Definition
.cpu cortex-m0	Limit instructions to those recognized by Cortex-M0.
.fpu softvfp	We don't have a floating-point hardware.
.syntax unified	Use unified syntax.
.thumb	Use 16- and 32-bit Thumb instruction set (not 32-bit ARM instructions).
.data	Put following items in read-write memory.
.text	Put following items in read-only memory.
.equ name, replacement	Replace name with replacement. (like #define)
.byte n [, n, n, n, ...]	Reserve and initialize 1 byte of storage for each element in a list of 8-bit integers.
.hword n [, n, n, n, ...]	Reserve and initialize 2 bytes of storage for each element in a list of 16-bit integers.
.word n [, n, n, n, ...]	Reserve and initialize 4 bytes of storage for each element in a list of 32-bit integers.
.space S	Reserve S bytes of storage. Do not initialize.
.string "..."	Reserve space for the characters plus a null byte terminator.
.align B	One of either of the following:
.balign B	Align the following item on a B-byte boundary.
.b2align B	Align the following item on a 2B-byte boundary.
.global symbol	Make the given symbol visible to outside modules.

Think in C. Code in Assembly

- Much easier to understand an algorithm in C.
- We will use it as a blueprint for your assembly.

- Some examples already

- Today we will talk about structured transformations for:
  - Variables and arrays
  - All control flow
  - Subroutines (i.e. function calls)

Updating a variable in memory

```c
int x;
x = x + 1;
```

```
ldr r0, =x          // load addr of x
ldr r1, [r0]        // load value of x
adds r1, #1         // add one to value
str r1, [r0]        // store result to x
```

```
…
.balign 4           // literal pool
lit_pool:           // added automatically
     .word x
```

Automatically generated by assembler

```
.data
x: .space 4
```

If manually add the literal pool

```c
int x;
x = x + 1;
```

```
ldr r0, lit_pool// load addr of x
ldr r1, [r0] // load value of x
adds r1, #1 // add one to value
str r1, [r0] // store result to x
```

```
…
.balign 4 // literal pool
lit_pool: // if we added, manually
    .word x
```

We have to write this ourselves

```
.data
x: .space 4
```

Basic math statements + getting variables

```
x = alpha + beta * gamma;
```

```
ldr r0, =alpha // load addr of alpha
ldr r1, [r0] // load value of alpha
ldr r0, =beta // load addr of beta
ldr r2, [r0] // load value of beta
ldr r0, =gamma // load addr of gamma
ldr r3, [r0] // load value of gamma
muls r2, r3 // r2 = r2 * r3
adds r1,r1,r2 // r1 = r1 + r2
ldr r0, =x // load addr of x
str r1, [r0] // store result to x
…
.balign 4 // literal pool added automatically
alpha_addr: .word alpha
beta_addr: .word beta
gamma_addr: .word gamma
x_addr: .word x
```

```
.data
.balign 4
alpha: .space 4
beta: .space 4
gamma: .space 4
x: .space 4
```

Basic math statements + getting variables

```
x = alpha + beta * gamma;
```

Same, except manual literal creation

```
.data
.balign 4
alpha: .space 4
beta: .space 4
gamma: .space 4
x: .space 4
```

```
ldr r0, alpha_addr // load addr of alpha
ldr r1, [r0] // load value of alpha
ldr r0, beta_addr // load addr of beta
ldr r2, [r0] // load value of beta
ldr r0, gamma_addr // load addr of gamma
ldr r3, [r0] // load value of gamma
muls r2, r3 // r2 = r2 * r3
adds r1,r1,r2 // r1 = r1 + r2
ldr r0, x_addr // load addr of x
str r1, [r0] // store result to x
…
.balign 4
alpha_addr: .word alpha
beta_addr: .word beta
gamma_addr: .word gamma
x_addr: .word x
```

Basic math statements + getting variables

```
x = alpha + beta * gamma;
```

Same as before but manual creation of a single-entry literal pool when placement of all variables is known.

```
.data
.balign 4
alpha: .space 4
beta: .space 4
gamma: .space 4
x: .space 4
```

```
ldr r0, vars // load addr of alpha
ldr r1, [r0] // load value of alpha
ldr r2, [r0, #4] // load value of beta
ldr r3, [r0, #8] // load value of gamma
muls r2, r3 // r2 = r2 * r3
adds r1,r1,r2 // r1 = r1 + r2
str r1, [r0, #12] // store result to x
…
.balign 4
vars: .word alpha
```

Think of alpha, beta, gamma as entries in a struct

Basic math statements + getting variables

```
x = alpha + beta * gamma;
```

Same as before - but let assembler create literal pool

```
ldr r0, =alpha // load addr of alpha
ldr r1, [r0] // load value of alpha
ldr r2, [r0, #4] // load value of beta
ldr r3, [r0, #8] // load value of gamma
muls r2, r3 // r2 = r2 * r3
adds r1,r1,r2 // r1 = r1 + r2
str r1, [r0, #12] // store result to x
```

```
.data
.balign 4
alpha: .space 4
beta: .space 4
gamma: .space 4
x: .space 4
```

If-then-else: General form

```
if (expr) {
    then_statements;
    …
} else {
    else_statements;
    …
}
```

```
if1:
    expr
    …
    branch_if_not else1
then1:
    then_statements
    …
    b endif1
else1:
    else_statements
    …
endif1:
```

If-then else example

```c
if (x > 100) {
    x = x - 1;
} else {
    x = x + 1;
}
```

```
if1:
    ldr r0, =x
    ldr r1, [r0]
    cmp r1, #100
    ble else1
then1:
    ldr r0, =x
    ldr r1, [r0]
    subs r1, #1
    str r1, [r0]
    b endif1
else1:
    ldr r0, =x
    ldr r1, [r0]
    adds r1, #1
    str r1, [r0]
endif1:
```

“do-while” General Form

```
do {
    do_body_stmts;
    ...
} while (expr);
```

```
do1:
    do_body_stmts
    ...
while1:
    expr
    branch_if_yes do1
enddo1:
```

“do-while” Example

```c
do {
    x = x >> 1;
} while (x > 2);
```

```
do1:
    ldr r0, =x
    ldr r1, [r0]
    asrs r1, r1, #1
    str r1, [r0]
while1:
    ldr r0, =x
    ldr r1, [r0]
    cmp r1, #2
    bgt do1
enddo1:
```

“while” loop general form

```
while (expr) {
    while_body_stmts; ...
}
```

```
while1:
    expr
    branch_if_not endwhile1
do1:
    while_body_stmts; …
    b while1
endwhile1:
```

"while" loop example

```
while (x > y) {
    y = y + 1;
}
```

```
while1:
    ldr r0, =x
    ldr r1, [r0]
    ldr r0, =y
    ldr r2, [r0]
    cmp r1, r2
    ble endwhile1
do1:
    ldr r0, =y
    ldr r1, [r0]
    adds r1, #1
    str r1, [r0]
    b while1
endwhile1:
```

“for” loop general form

```
for (init; check; next_stmt) {
    for_body_stmts; ...
}
```

```
init;
while (check) {
    for_body_stmts; …
    next_stmt;
}
```

```
for1:
    init
forcond1:
    check
    branch_if_not fordone1
forbody1:
    for_body_stmts; …
fornext1:
    next_stmt
    b forcond1
fordone1:
```

“for” loop example

```c
for (q=0; n>=d; q++) {
      n = n - d;
}
r = n;
```

```
for1:
    ldr r0, =q
    movs r1, #0
    str r1, [r0]
forcond1:
    ldr r0, =n
    ldr r1, [r0]
    ldr r0, =d
    ldr r2, [r0]
    cmp r1, r2
    blt fordone1
forbody1:
    ldr r0, =n
    ldr r1, [r0]
    ldr r0, =d
    ldr r2, [r0]
    subs r1, r1, r2
    ldr r0, =n
    str r1, [r0]
fornext1:
    ldr r0, =q;
    ldr r1, [r0]
    adds r1, #1
    str r1, [r0]
fordone1:
    ldr r0, =n
    ldr r1, [r0]
    ldr r0, =r
    str r1, [r0]
```

Surprise! This “for” loop actually did something useful.

Note: This is not an efficient way to implement division! But good for a simple example

Quotient (q)

$$\frac{5 \leftarrow \text{Dividend } (n)}{2 \leftarrow \text{Divisor } (d)} = 2 + 1$$

Remainder (r)

Dividend = Quotient * Divisor + Remainder

– Initialize the quotient to be zero.
– While numerator >= denominator,
  - Subtract the divisor (denom.) from the numerator
  - Increment the quotient
– The dividend (numerator) is now the remainder.

- Implements a simple division algorithm.
- Remember: no DIV instruction on Cortex-M0

```
for (q=0; n>=d; q++) {
    n = n - d;
}
r = n;
```

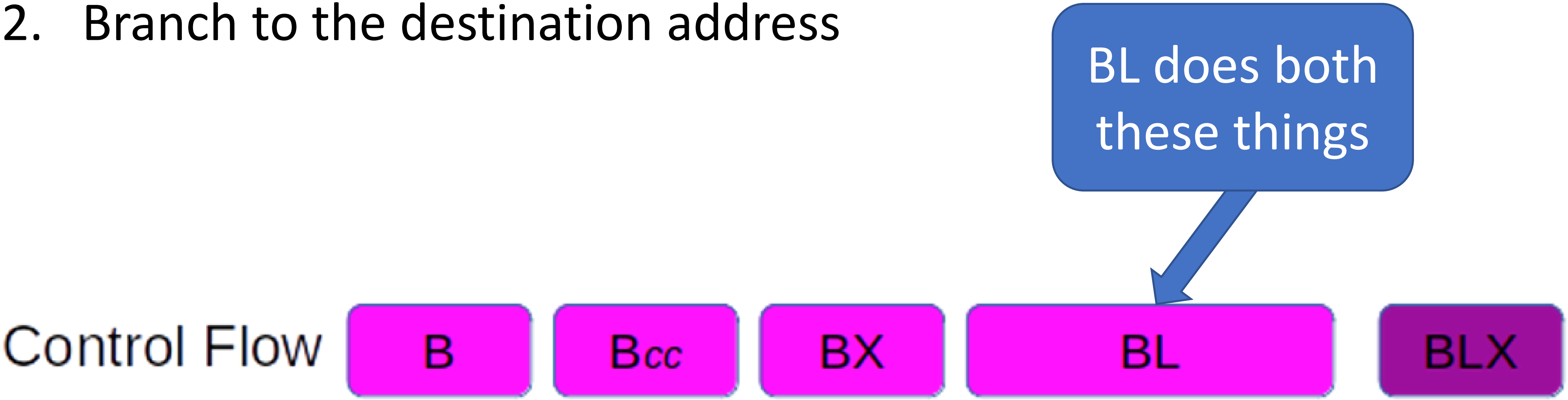
Want to implement functions (aka subroutines)

```c
int x;
int main(void) {
    x = x + 1;
    empty_subroutine();
    x = x - 2;
    ...
}

void empty_subroutine(void) {
}
```

How to call a function on the Cotex-M0

1. Save pointer to next instruction in Link Register (LR)
   - **It is the callee's responsibility to save the LR**
2. Branch to the destination address



Special branch instructions BL / BX

Example 0

```c
int x;
int main(void) {
    x = x + 1;
    empty_subroutine();
    x = x - 2;
    ...
}

void empty_subroutine(void) {
}
```

```
ldr r0,=x
ldr r1,[r0]
adds r1,r1,#1
str r1,[r0]
bl empty_subroutine
ldr r0,=x
ldr r1,[r0]
subs r1,r1,#2
str r1,[r0]
bkpt
…
empty_subroutine:
    bx lr

.data
x: .word 1
```

BX: Branch and Exchange "Put the contents of the specified register into the PC."

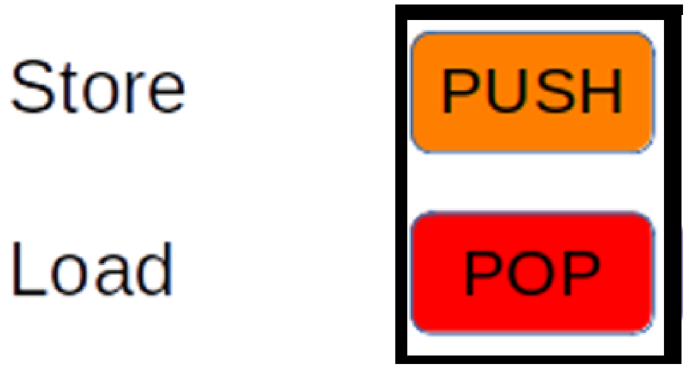
Empty subroutine too simple. Real function calls need a stack (FILO data structure)

- Cortex-M0 has PUSH and POP instructions:
- PUSH:
  - Decrement SP (multiple times).
  - Write multiple registers into memory pointed to by SP.
    - One of those registers can be LR.

See ARMv6-M Architecture Reference Manual Section A6.7.50

- POP:
  - Read multiple registers from memory pointed to by SP.
    - One of those registers can be PC.
    - Increment SP (multiple times).

See ARMv6-M Architecture Reference Manual Section A6.7.49



A real subroutine

```
.syntax unified
.cpu cortex-m0
.thumb
.global main
main:
    movs r0, #25 // initialize r0 with 25
    bl incr0 // "call" incr0. LR is next instruction.
    bkpt // What is value of r0 when we get here?
incr0:
    push {lr} // Save the LR “on the stack”
    adds r0, #1 // Add 1 to r0
    pop {pc} // Pop one word “from the stack” to PC.
```

More complex calling

Aside: What is the final value of x?
A: 1
B: 2
C: 3
D: 4
E: 5

```
int x = 0;
int main() {
    first();
    first();
...
}
void first() {
    second();
    second();
}
void second() {
    x += 1;
}
```

- main() calls:
  - First once
  - First again
- First calls:
  - Second once
  - Second again
- Each time we call the functions, the return address will be different

Saving LR not always needed

Example 1

```c
int x = 0;
int main() {
    first();
    first();
...
}
void first() {
    second();
    second();
}
void second() {
    x += 1;
}
```

```asm
.data
x: .word 0
.text
.global main
main:
    bl first
    bl first
    bkpt
first:
    push {lr}
    bl second
    bl second
    pop {pc}
second:
    ldr r0, =x
    ldr r1, [r0]
    adds r1, #1
    str r1, [r0]
    bx lr
```