Assembly Language Programming

ECE 362
https://engineering.purdue.edu/ece362/

Reading you should have done

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

Upcoming reading

- 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.

Assembler Directives

• .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 2^B-byte boundary.
• .global symbol	Make the given symbol visible to outside modules.

Don't forget comma.

Think in C. Code in Assembly.

- We’re going to start writing major programs in assembly language.
- The easiest way to do that (that I know of) is to think about what you want in C, and then translate it into assembly language.
- C becomes the modeling language.
- There are three things we need to understand:
  - How to represent variables and arrays
  - How to represent control flow (if, while, do, for)
  - How to represent subroutines

C-to-Assembly Translation: Variable Update Via Register

```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

.data
x: .space 4
```

C-to-Assembly Translation: Variable Update Via Register

```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

.data
x: .space 4
```

C-to-Assembly Translation: Statements

```
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
.data                        …
.balign 4                    .balign 4 // literal pool added automatically
alpha:  .space 4             alpha_addr: .word alpha
beta:   .space 4             beta_addr:  .word beta
gamma:  .space 4             gamma_addr: .word gamma
x:      .space 4             x_addr:     .word x
```

C-to-Assembly Translation: Statements

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

Same as before
but manual creation
of the literal pool...

```
.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
```

C-to-Assembly Translation: Statements

```
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
```

C-to-Assembly Translation: Statements

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

Same as before
but automatic creation
of the 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, =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
```

C-to-Assembly Translation: "if-then-else"

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

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

C-to-Assembly Translation: "if-then-else" example

```
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:
```

Branch is always based on the *opposite* of the condition tested for.

C-to-Assembly Translation: "do-while"

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

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

C-to-Assembly Translation: "do-while" example

```
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:
```

C-to-Assembly Translation: "while"

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

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

C-to-Assembly Translation: "while" 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:
```

C-to-Assembly Translation: "for"

```
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:
```

C-to-Assembly Translation: "for" example

```
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]
        b forcond1
```

```
fordone1:
        ldr r0, =n
        ldr r1, [r0]
        ldr r0, =r
        str r1, [r0]
```

Simple Division

- The ARM Cortex-M0 has no divide instruction.
  - You should feel lucky that it has a multiply.
- Simple way to divide one number by another:
  - 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.

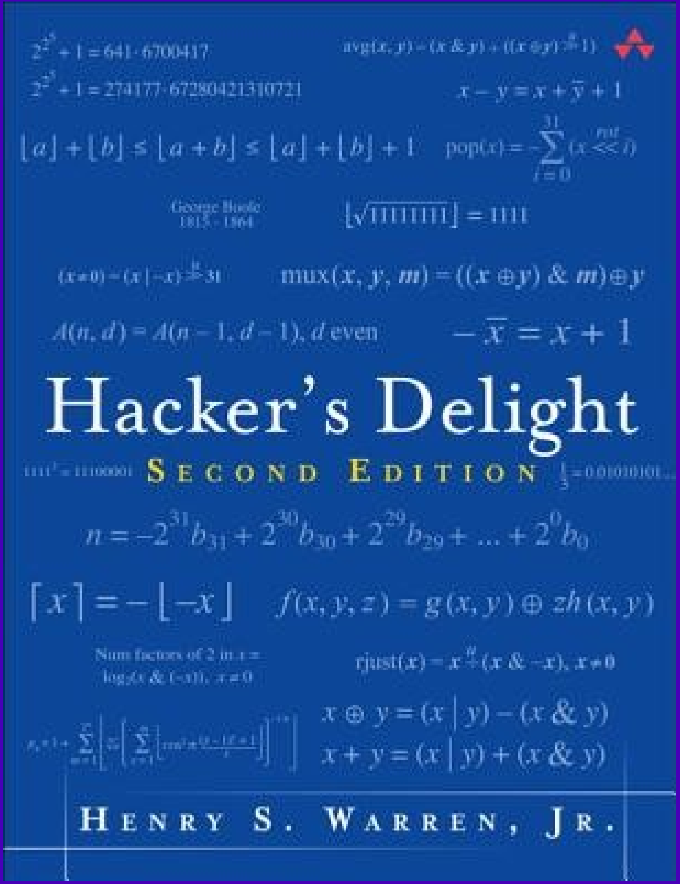
```
for (q=0; n>=d; q++) {
    n = n - d;
}
r = n;
```

Division Example:

- 90 / 12: quotient = 0
- 78 / 12: quotient = 1
- 66 / 12: quotient = 2
- 54 / 12: quotient = 3
- 42 / 12: quotient = 4
- 30 / 12: quotient = 5
- 18 / 12: quotient = 6
- 6 / 12: quotient = 7
- Done. Quotient = 7, Remainder = 6

"Can’t we do this any faster?"

Yes.



Consider a simple program:

- Let’s use subroutines this time.
  - How do we even do this?

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

void empty_subroutine(void) {
}
```

The Stack: Subroutines

- The Cortex-M0 has a stack pointer (SP), but no JSR instruction.
  - Recall that JSR on the simple computer:
    - Decremented (subtracted one from) SP
    - Wrote PC of next instruction to [SP]
    - Jumped to immediate argument.
  - Cannot have all that complexity in a RISC CPU.
  - To invoke a subroutine with a Cortex-M0 CPU, we:
    - save pointer to next instruction in Link Register (LR)
    - branch to the destination address
    - Instruction is BL (Branch with Link)
    - It is the callee’s responsibility to save the LR

BL: (branch with link)

“Put the current PC into the LR register and branch to the given label.”

Example of BL

```
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
```

"Wait, you didn't even use the stack there."

True. This is the smallest example possible.

Example 0 on lecture notes page

BX: Branch and Exchange

"Put the contents of the specified register into the PC."

Saving LR on the Stack

- 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.
  - 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 A6.7.50

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 Functions

- Consider this C program:
  - main calls first
    - first calls second
    - first calls second again
  - main calls first again
    - first calls second
    - first calls second again
- first and second must know where to return to!

By the way, what's the final value of x?

```
int x = 0;

int main() {
    first();
    first();
    ...
}

void first() {
    second();
    second();
}

void second() {
    x += 1;
}
```

Don’t always need to save LR

```
.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
```

Example 1 on lecture notes page

first() must save LR <u>before</u> calling second()

first() returns with pop {pc}

second() is a “leaf function”. It calls no other functions, so LR will not be overwritten. No need to push it.

```
int x = 0;

int main() {
    first();
    first();
    ...
}

void first() {
    second();
    second();
}

void second() {
    x += 1;
}
```

We can save more registers with PUSH/POP than just LR/PC

Example 2 (or something similar) on lecture notes page

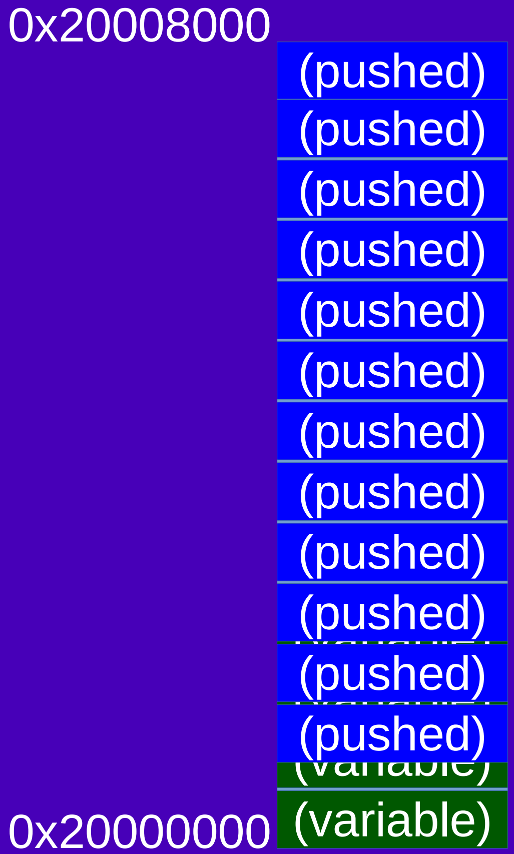
```
.syntax unified
.cpu cortex-m0
.thumb

.global main
main:
    movs    r0, #25           // initialize r0 with 25
    bl      incr0             // "call" incr0.  LR is next instruction.
    nop
    bkpt                      // What is value of r0 when we get here?

incr0:
    push    {r1-r4,r6,lr}     // Save lots of registers to the stack
    adds    r0, #1            // Add 1 to r0
    pop     {r1-r4,r6,pc}     // Pop several words from stack
```

r1-r4 means R1 ***<u>through</u>*** R4 (R1,R2,R3,R4)

Be Careful With Stack Memory

- Your development board has 32K (32768) bytes of SRAM (static random-access memory).
  - `0x20000000 - 0x20007fff`
- SP is initialized to 0x20008000.
  - (It is decremented by 4 before writing a word to memory.)
- Things like variables in the data segment start at 0x20000000.
- It’s really all the same memory.
- If you keep pushing things onto the stack,

  you will eventually overwrite variables.



Consider a recursive subroutine:

```c
int total = 0;

int sum(int x) {
    if (x > 0) {
        total += x;
        sum(x-1);
    }
}

int main() {
    sum(3);
}
```

```
sum:    push {lr}        // save link register
if1:    cmp  r0,#0       // x > 0
        ble  endif1      // no? goto endif1
        ldr  r1,=total   // get addr of total
        ldr  r2,[r1]     // load from total
        adds r2,r0       // total = total + x
        str  r2,[r1]     // store total
        subs r0,#1       // x - 1
        bl   sum         // sum(x - 1)
endif1: nop
        pop  {pc}        // return

.global main
main:
        movs r0, #3
        bl   sum
        wfi
.data
total   .word 0
```

Example 3 on lecture notes page

Application Binary Interface

- Every platform defines an ABI for functions and subroutines.
- Ours looks like this:
  - The call<u>er</u> passes the first four parameters in R0-R3.
    - Any more parameters go on the stack.
  - The call<u>ee</u> is allowed to modify R0-R3 and R12.
    - If any other registers are changed, they must first be saved to the stack.
  - The return value from a function is in R0.
- These are the rules if you want your assembly language functions to get along with C functions.
  - We can do anything we want if we're writing our own code in assembly language, but a C compiler maintains certain expectations. When we ***<u>link in</u>*** C code, we have to play by the rules.

Why have an ABI?

- Clearly defines who needs to save registers and when.
- As long as everyone adheres to the ABI, you know that fn() will not mess with R4 – R11.
- calc() saves and restores R4 so that any caller can trust it not to modify R4.

```
int calc(int x) {
    return x + fn(3);
}
```

```
calc:
    push {r4,lr}
    movs r4,r0
    movs r0, #3
    bl fn
    adds r0,r4
    pop {r4,pc}
```

Consequences of the ABI

- You almost never want to save and restore R0.
  - Don’t push or pop that.
- You almost never need to save and restore R1 – R3 or R12.
- Generally, if you need to use R4 – R11, save them at the beginning of your subroutine and restore them at subroutine exit.

Check the “Bonus Lectures”

- There are many nuances of subroutines that are difficult for students to understand.
- The course textbook has good coverage of these topics.
- Otherwise, go through the bonus lectures on subroutines and recursive flow control.