Assembly Language Programming

ECE 362
https://engineering.purdue.edu/ee362/
Reading you should have done

• Textbook, Chapter 7, ”Structured Programming”, pages 133 – 160.
  - Today’s topic.

  - Also today.
Upcoming reading

• Textbook, Chapter 10, “Mixing C and Assembly”, pages 215 – 236.
  - We’ll talk about this next lecture.

• Textbook, Chapter 14, “General Purpose I/O (GPIO)”, pages 341 – 372.
  - We’ll talk about this next lecture.

  - We’ll talk about this next week.
Logical Shift Left

- We can shift a 32-bit value in a register “left” by any number of bits.
  - This is the same as the C language operator: `<<`
  - Bits that are shifted out the left end (MSB) of the register are moved into the APSR Carry flag, so we call it “LSLS”.
  - Zeros are shifted in on the right side (LSB).

Example: `LSLS R0, R0, #3`

Initial R0: 10111001110001001100110100001111111

New R0: 110011100100110011010100110011111100
We can use LSL to test bits

- Shift “leftmost” bit into the carry flag.
- How do we test if the carry flag is set?
  - BCS: “Branch if Carry Set”

```assembly
// Count the number of ‘1’ bits in R2.
// Store result in R0.
        movs r0, #0
more:   lsls r2, r2, #1
        bcs one
        beq done
        b more
one:    adds r0, #1
        b more
done:
```
Other Shift & Rotate Instructions

Logical Shift Right: LSRS R0, R0, #2
0 1011 1001 1100 0010 0011 001101000011111
Initial R0
New R0

Arithmetic Shift Right: ASRS R0, R0, #2
1011 1001 1100 0010 0011 001101000011111
Initial R0
New R0

Rotate Right: RORS R0, R0, #6
10111001 1100 0010 0011 001101000011111
Initial R0
New R0
Bitwise Logical Instructions

**ANDS R0, R1**

<table>
<thead>
<tr>
<th>Initial R0</th>
<th>AND</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10111001110001001100110101000111111</td>
<td>(mask)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001000000010011001100000111100</td>
</tr>
</tbody>
</table>

**ORRS R0, R1**

<table>
<thead>
<tr>
<th>Initial R0</th>
<th>OR</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10111001110001001100110101000111111</td>
<td>(new bits to set)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10111001110001111111111101000111111</td>
</tr>
</tbody>
</table>
Bitwise Logical Instructions

EORS R0, R1

Initial R0

AND

R1

New R0

(MVNS R0, R1  // Also known as NOT

Initial R1

New R0

(bits to toggle)
Assembler Directives

- .cpu cortex-m0
- .fpu softvfp
- .syntax unified
- .thumb
- .data
- .text
- .equ name, replacement
- .byte X
- .hword X
- .word X
- .space S
- .string "...
- .align B
- .global symbol

Limit instructions to those recognized by Cortex-M0.
We don't have a floating-point hardware.
Use unified syntax.
Use 16-bit Thumb instructions (not 32-bit ARM instructions).
Put following items in read-write memory.
Put following items in read-only memory.
Replace name with replacement. (like #define)
Reserve 1 byte of storage. Initialize with X.
Reserve 2 bytes of storage. Initialize with X.
Reserve 4 bytes of storage. Initialize with X.
Reserve S bytes of storage. Do not initialize.
Reserve space for the characters plus a null byte terminator.
Align the following item on a B-byte boundary.
Make the given symbol visible to outside modules.
C-to-Assembly Translation: Variable Update Via Register

```plaintext
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

... .align 4  // literal pool
lit_pool:
   .word x  // added automatically

.data
x: .space 4
```
C-to-Assembly Translation: Variable Update Via Register

```
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

...  // literal pool
lit_pool:       // added automatically
    .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
```

```
.align 4 // literal pool added automatically
alpha_addr: .word alpha
beta_addr:  .word beta
 gamma_addr: .word gamma
   x_addr:     .word x
```
C-to-Assembly Translation: Statements

\[
x = \alpha + \beta \times \gamma;
\]

```assembly
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 \times r3
adds r1,r1,r2           // r1 = r1 + r2
ldr r0, x_addr          // load addr of x
str r1, [r0]             // store result to x

.align 4
alpha_addr: .word alpha
beta_addr:  .word beta
gamma_addr: .word gamma
x_addr:     .word x
```

Same as before but manual creation of the literal pool...
C-to-Assembly Translation: Statements

x = alpha + beta * gamma;
ldr r0, vars
ldr r1, [r0]
ldr r2, [r0, #4]
ldr r3, [r0, #8]
muls r2, r3
adds r1,r1,r2
str r1, [r0, #12]

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

.data
.align 4
alpha: .space 4
beta: .space 4
gamma: .space 4
x: .space 4
C-to-Assembly Translation: "if-then-else"

if (expr) {
    then_statements; ...
} else {
    else_statements; ...
}

if1:
    expr
    ...
    branch_if_not else1
then1:
    then_statements
    ...
    b fi1
else1:
    else_statements
    ...
fi1:
C-to-Assembly Translation: "if-then-else" example

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

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

do {  do1:
    do_body_stmts; ...   do_body_stmts
  ...   ...
} while (expr);   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

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

```assembly
while1:
    ldr r0, =x
    ldr r1, [r0]
    ldr r0, =y
    ldr r2, [r0]
    cmp r1, r2
    b! `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;
}
C-to-Assembly Translation: "for" example

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

```assembly
for1:
    ldr r0, =q
    movs r1, #0
    str r1, [r0]

check1:
    ldr r0, =n
    ldr r1, [r0]
    ldr r0, =d
    ldr r2, [r0]
    cmp r1, r2
    blt endfor1

body1:
    ldr r0, =n
    ldr r1, [r0]
    ldr r0, =d
    ldr r2, [r0]
    subs r1, r1, r2
    ldr r0, =n
    str r1, [r0]

next1:
    ldr r0, =q;
    ldr r1, [r0]
    adds r1, #1
    str r1, [r0]
    b check1

endfor1:
    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 from the numerator
    • Increment the quotient
  – The dividend is now the remainder.

```c
for (q=0; n>=d; q++) {
    n = n - d;
}
r = n;
```
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
Consider a simple program:

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

```c
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:
“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]
...
```

```
empty_subroutine:
    bx lr
```

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

True. This is the smallest example possible.

bx lr:

"Put the LR register into the PC."
Saving LR on the Stack

- Cortex-M0 has PUSH and POP instructions:
  - PUSH:
    - Decrement SP (multiple times).
    - Write multiple registers to 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 A6.7.49
See ARMv6-M Architecture Reference Manual Section A6.7.50
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.
```
We can save more registers with PUSH/POP

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

r0-r4 means R0 through R4
About Stack Memory

- Your development board has 8192 bytes of SRAM (static random-access memory).
  - 0x20000000 – 0x20001fff
- SP is initialized to 0x20002000.
  - (It is decremented before writing to memory.)
- Things in the data segment start at 0x20000000.
- 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);
}
```

```c
int total = 0;

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

```c
.int total = 0;
```

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

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

```c
.int total = 0;
```
Application Binary Interface

• Every platform defines an ABI for functions and subroutines.
• Ours looks like this:
  - The caller passes the first four parameters in R0-R3.
    • Any more parameters go on the stack.
  - Callee 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 link in C code, we have to play by the rules.
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, you should save only R4 – R7.