Embedded C

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

Reading Assignment

- Textbook, Chapter 10, “Mixing C and Assembly”, pages 215 – 236.
  - Talking about this in the this lecture module.
- Textbook, Chapter 15, “General-purpose Timers”, pages 373 – 414.
  - Talking about advanced timer use in the next lecture module.
- Future reading:
  - Textbook, Chapter 21, Digital-to-Analog Conversion, pp. 507 – 526.
    - You should read this first.
  - FRM, Chapter 14, Digital-to-analog converter (DAC), pp. 269 – 281.
    - Scan. Learn basics like I/O registers, enabling, use.
  - Textbook, Chapter 20, Analog-to-Digital Conversion (ADC), pp. 481 – 506.
    - Read this later.
  - FRM, Chapter 13, Analog-to-Digital converter (ADC), pp. 229 – 268.
    - Scan this later. Learn basics like I/O registers, enabling, use.

Why does the ADC
come before the DAC
in the textbook and FRM?

Mystery

Why does anyone write in C?

- Because it’s so simple to create bloated, inefficient code!
  - We have 32K of RAM and 256K of ROM.
    - More than we’ll ever need!

```c
int first(int x) {
    return x;
}
```

C compiler is good enough for me!

```
.global first
first:
    push {r7,lr}
    sub  sp, #4
    add  r7, sp, #0
    str  r0, [r7, #0]
    nop
    ldr  r2, [r7, #0]
    movs r0, r2
    mov  sp, r7
    add  sp, #4
    pop  {r7,pc}
```

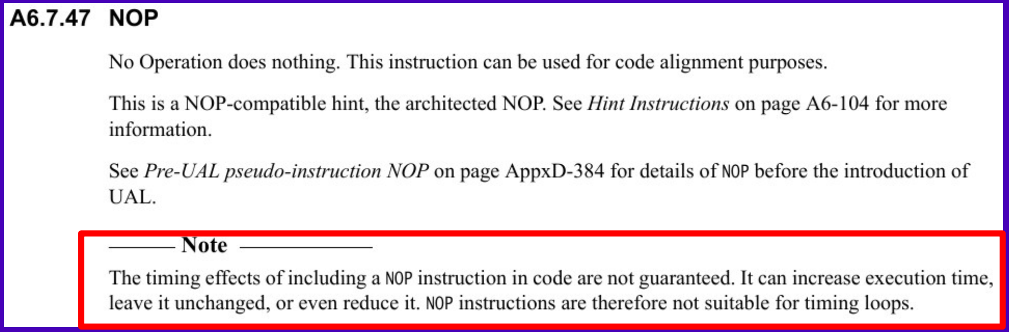
Actual student homework submission.

C is good for managing complexity

- You recognize the code is not optimal, but at least you didn’t have to understand it.
  - Your grades for the class are often correlated to how well you understand.
    - Therefore, we’re going to continue using assembly language.

With assembly language, you know exactly what is happening.

```
.text
.global micro_wait
micro_wait:
        // Total delay = r0 * (1+10*(1+3)+1+1+1+1+3)+1
        //            = r0 * 48 cycles
        // At 48MHz, this is one usec per loop pass.
        // Maximum delay is 2^31 usec = 2147.5 sec.
        movs r1, #10     // 1 cycle
loop:   subs r1, #1      // 1 cycle
        bne loop         // 3 cycles (Why?)
        nop              // 1 cycle
        nop              // 1 cycle
        nop              // 1 cycle
        subs r0, #1      // 1 cycle
        bne  micro_wait  // 3 cycles (Why?)
        bx   lr          // 1 cycle
```



A6.7.47 NOP

No Operation does nothing. This instruction can be used for code alignment purposes.

This is a NOP-compatible hint, the architected NOP. See *Hint Instructions* on page A6-104 for more information.

See *Pre-UAL pseudo-instruction NOP* on page AppxD-384 for details of NOP before the introduction of UAL.

Note

The timing effects of including a NOP instruction in code are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. NOP instructions are therefore not suitable for timing loops.

- ARM Architecture Reference Manual covers many different devices,
  - some of which may be speculative, out-of-order, superscalar.
- For the CPU we’re using, nop does exactly what we expect it to.

Why Do Branches Take 3 Cycles?

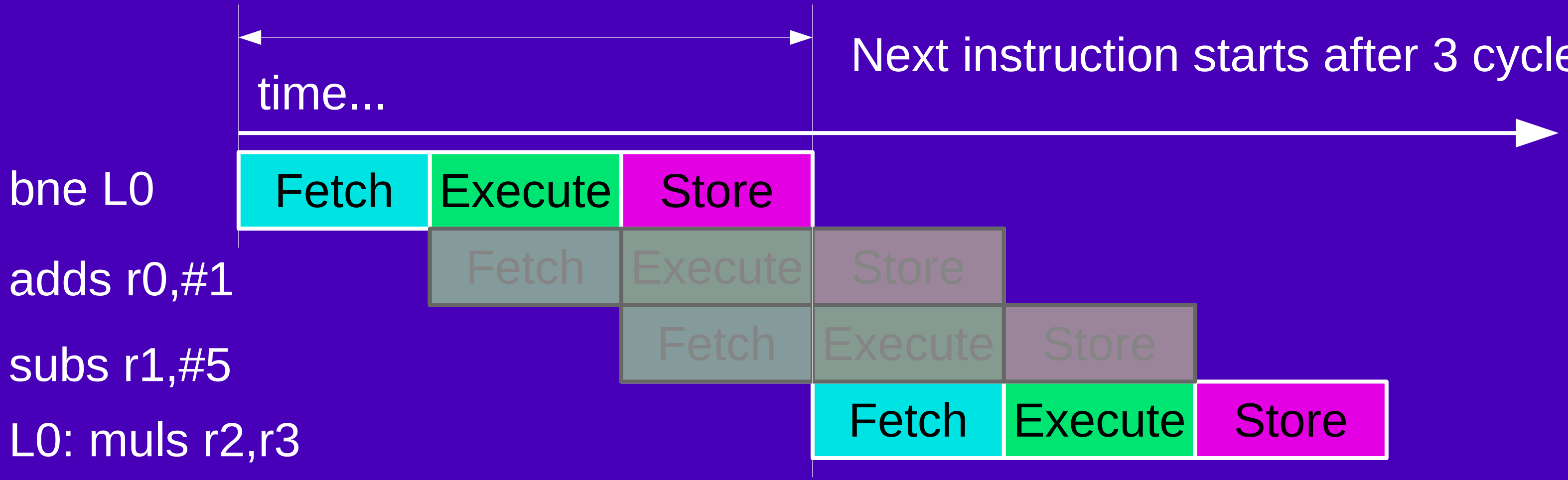
- I mentioned that the ARM Cortex-M0 had a three-stage pipeline.
  - Each instruction takes 3 cycles
  - Every instruction starts on the 2nd cycle of the previous one

time... →

Instr #1:	Fetch	Execute	Store			
Instr #2:		Fetch	Execute	Store		
Instr #3:			Fetch	Execute	Store	
Instr #4:				Fetch	Execute	Store

Why Do Branches Take 3 Cycles?

- When an instruction is a branch, the instructions immediately following it may or may not be executed.
  - If the branch is taken, we need to throw away those instructions.
  - We can’t fetch the right instructions until we update the PC!



C operators:

- Bitwise logical operators:
  - | OR
  - & AND
  - ^ XOR
- Unary logical operator:
  - ~ Bitwise NOT (different than negate)
- Shift operators:
  - << Left shift (like multiplication)
  - >> Right shift (like division)
    - An unsigned integer uses the LSR instruction.
    - A signed integer uses the ASR instruction.

Modifying bits of a word

- OR bits into a word (turn ON bits):

  x = x | 0x0a0c shorthand: x |= 0x0a0c

  Turns ON the bits with ‘1’s in 0000 1010 0000 1100

- AND bits into a word (mask OFF ‘0’ bits):

  x = x & 0x0a0c shorthand: x &= 0x0a0c
  Turns OFF all <u>except</u> bits with ‘1’s in 0000 1010 0000 1100

- AND inverse bits of a word (mask OFF ‘1’ bits):

  x = x & ~0x0a0c shorthand: x &= ~0x0a0c

  Turns OFF bits with ‘1’s in 0000 1010 0000 1100

- XOR bits of a word (toggle the ‘1’ bits)

  x = x ^ 0x0a0c shorthand: x ^= 0x0a0c

  Inverts the bits with ‘1’s in 0000 1010 0000 1100

Shifting values

- Shift a value left by 5 bits:

  x = x << 5   shorthand: x <<= 5

- Shift a value right by 8 bits:

  x = x >> 8   shorthand: x >>=8

- There is no rotate operator in C.
  - Improvise by combining shifts/ANDs/ORs.
  - rotate a 32-bit number right by 4:
    - x = (x >> 4) & 0x0fffffff | (x << 28) & 0xf0000000
- No way to directly set or check flags in C.

Combining operators

- Turn on the nth bit of the ODR:

  ODR |= 1<<n

- Turn off the nth bit of the ODR:

  ODR &= ~(1<<n)

- Note that either side of the << can be a constant or a variable.
- Warning: These look a lot easier than assembly, but remember that they are not ***atomic***.
  - If an interrupt occurs in the middle of the bloated code to implement the statement, strange things could happen.
  - Use of BRR/BSRR are just as effective in C as they are in assembly language and they are still ***atomic***.

Pointers

- If you did not take an advanced C class, you might not have studied pointers.
- That's OK. You've been using addresses. It's the same thing (but with types):

```
int x, *p; ….
x = *p     is similar to:   ldr r0,[r1]
x = p[3]   is similar to:   ldr r0,[r1,#12]
p += 1     is similar to:   adds r1, #4
```

Type casts

- In an embedded system, we usually know where everything is in memory.
- To create a pointer to the RCC_AHBENR register, we might say:

  int *rcc_ahbenr = **(int *)** 0x40021014;

  - Here, the **type cast** says "Trust me that this is really a pointer to an integer."

Structures

- In C, we can define hierarchical types to organize information that goes together.
  - The grouping is called a **struct**.
  - Each element within a struct is called a **field**.
  - We access fields with a dot (.) operator.
  - For a pointer to a struct, we access a field with an arrow (->) operator.

Example of a struct

```
struct Student {
    char name[128];
    unsigned int age;
    int km_north_of_equator;
    int hours_of_sleep;
};
```

Type declaration for <u>struct Student</u>

```
void Adjust_Schedule(void) {
    struct Student s = {
        "Typical ECE 362 student",
        20,
        4495,
        8,
    };

    for(month=1; month<=5; month+=1) {
        s.hours_of_sleep -= 1;
    }
}
```

And here we allocate space for one.
The variable is named "s".

The dot (.) operator is really just an offset into the memory space of the struct.

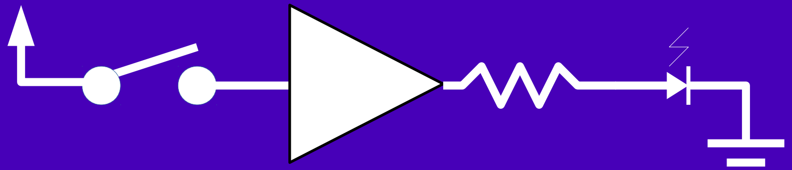
More useful example: GPIOx

```
struct GPIOx_type {
    unsigned int MODER;        // Offset 0x0
    unsigned int OTYPER;       // Offset 0x4
    unsigned int OSPEEDR;      // Offset 0x8
    unsigned int PUPDR;        // Offset 0xc
    unsigned int IDR;          // Offset 0x10
    unsigned int ODR;          // Offset 0x14
    unsigned int BSRR;         // Offset 0x18
    unsigned int LCKR;         // Offset 0x1c
    unsigned int AFR[2];       // Offset 0x20
    unsigned int BRR;          // Offset 0x28
};

struct GPIOx_type *GPIOA = (struct GPIOx_type *) 0x48000000;
struct GPIOx_type *GPIOB = (struct GPIOx_type *) 0x48000400;
struct GPIOx_type *GPIOC = (struct GPIOx_type *) 0x48000800;
```

Given the previous definitions

```
// Read bit from PA0.  Write bit to PC8.
//

for(;;)
    (*GPIOC).ODR = (*GPIOC).ODR & ~0x00000100 |
                   ((*GPIOA).IDR & 0x00000001) << 8;
```

Same thing with arrows



```
// Read bit from PA0.  Write bit to PC8.
//

for(;;)
      GPIOC->ODR =   GPIOC->ODR & ~0x00000100 |
                   (  GPIOA->IDR & 0x00000001) << 8;
```

```
     ldr  r0, =GPIOA     // Load, into R0, base address of GPIOA
     ldr  r1, =GPIOC     // Load, into R1, base address of GPIOC
again:
     ldr  r2, [r0, #IDR] // Load, into R2, value of GPIOA_IDR
     movs r3, #1
     ands r2, r3         // Look at only bit zero.
     lsls r2, #8         // Translate bit 0 to bit 8
     ldr  r4, [r1, #ODR] // Load value of ODR
     ldr  r3, =0xfffffeff// Load mask
     ands r4, r3         // AND off bit with mask
     orrs r4, r2         // OR in the new value
     str  r4, [r1, #ODR] // Store value to GPIOC_ODR
     b    again
```

STM32 Standard Firmware

- C structure definitions for control registers is provided with the standard firmware.
  - Advantage: You don’t have to look up the addresses and offsets for things.
  - The C code on the previous slide actually compiles and does what it looks like it does.
  - More about this in a few slides when we talk about CMSIS.

Storage class of variables

- Variables in C can be given a storage class which defines how the compiler can access them.
  - One of these classes is ***const***.
  - Const says that a variable must not be mutated after its initial assignment.
  - Compiler is free to put such a "variable" in the text segment where it cannot be modified.

Example of const

```
int var = 15;
const int one = 1;  // The value of "one" cannot be changed.

// Since "one" is a global const variable,
// it is placed in the text segment.

int main(void) {
    int x;
    for(x=0; x<20; x += one)
        var += one;

    one = 2;    // not allowed.  Fault handler!
    return 0;
}
```

Other examples of const

```
// Most commonly, "const" is used to describe pointers
// whose memory region a program is not allowed to modify.

const int *GPIOA_IDR = (const int *) 0x48000010;

...

int x = *GPIOA_IDR;           // fine
*GPIOA_IDR = 5;               // compiler error
* ((int *) GPIOA_IDR) = 5;    // do it anyway

// const is a reminder that something should not be modified.
// You can still get around it.
```

What about this code?

```
int count = 0;

int myfunc(void) {
    int begin = count;
    while (count == begin)
        ;
    return count;
}
```

Compiler sees this code and believes that the value of count never changes.

<u>So it never checks.</u>

What if something outside this code (that the compiler doesn't know about) may modify count at any time?

Example

```c
#include <stdio.h>
#include <signal.h>

int count = 0;

void handler(int sig) {
  count += 1;
  printf("\nChanged to %d\n", count);
}

int main(void) {
    signal(SIGINT, handler);    // Set up a <ctrl>-C handler.

    int begin = count;          // Copy value of count.
    while (count == begin)      // Check to see if count changed.
        ;
    printf("Count changed.\n");
    return count;
}
```

Compile this on a normal machine (i.e. Unix) or a nearly normal machine (i.e. MacOS).

If you compile without optimization, it will exit when you press <ctrl>-C.

If you compile with optimization (-O3), it will never exit no matter how many times you press <ctrl>-C.

How can we tell the compiler?

- Sometimes, we want to tell the compiler that a variable might change in ways that it cannot possibly know.
- We give it a storage class of ***volatile***.
- Add volatile to any type like this:

  volatile int count = 0;

- Tells the C compiler to always check it rather than holding its value in a register as an optimization.

Mixed storage classes

- Can something be both const and volatile?
  - Yes.
  - This is a read-only variable that changes in ways that the compiler cannot understand.
  - e.g.:

```
const volatile int *gpioa_idr = (const volatile int *)0x48000010;
```

If I gave you the following...

```
.equ RCC, 0x40021000
.equ AHBENR, 0x14
.equ IOPCEN, 0x80000
.equ GPIOC, 0x48000800
.equ MODER, 0x0
.equ ODR, 0x14
.equ BSRR, 0x18
.equ BRR, 0x28
.equ PIN8_MASK,    0xfffcffff
.equ PIN8_OUTPUT,  0x00010000

.global main
main:
    // OR IOPCEN bit into RCC AHBENR
```

```
// Enable pin 8 as an output
```

...could you complete it?

Assembly language to blink pc8

```
.equ RCC, 0x40021000
.equ AHBENR, 0x14
.equ IOPCEN, 0x80000
.equ GPIOC, 0x48000800
.equ MODER, 0x0
.equ ODR, 0x14
.equ BSRR, 0x18
.equ BRR, 0x28
.equ PIN8_MASK,    0x00030000
.equ PIN8_OUTPUT,  0x00010000

.global main
main:
    // OR IOPCEN bit into RCC AHBENR
    ldr  r0, =IOPCEN
    ldr  r1, =RCC
    ldr  r2, [r1,#AHBENR]
    orrs r2, r0
    str  r2, [r1,#AHBENR]
```

```
    // Enable pin 8 as an output
    ldr  r0, =PIN8_MASK
    ldr  r1, =GPIOC
    ldr  r2, [r1,#MODER]
    bics r2, r0
    ldr  r0, =PIN8_OUTPUT
    orrs r2, r0
    str  r2, [r1,#MODER]
forever:
    ldr  r0, =0x100
    str  r0, [r1,#BSRR] // pin 8 on
    ldr  r0, =1000000
    bl   micro_wait
    ldr  r0, =0x100
    str  r0, [r1,#BRR]  // pin 8 off
    ldr  r0, =1000000
    bl   micro_wait
    b    forever
```

C code to blink pc8

```
#include "stm32f0xx.h"

void micro_wait(int);

int main(void)
{
    RCC->AHBENR  |=  RCC_AHBENR_GPIOCEN;
    GPIOC->MODER &= ~GPIO_MODER_MODER8;
    GPIOC->MODER |=  GPIO_MODER_MODER8_0;

    for(;;) {
        GPIOC->BSRR = GPIO_ODR_8;
        micro_wait(1000000);
        GPIOC->BRR = GPIO_ODR_8;
        micro_wait(1000000);
    }
}
```

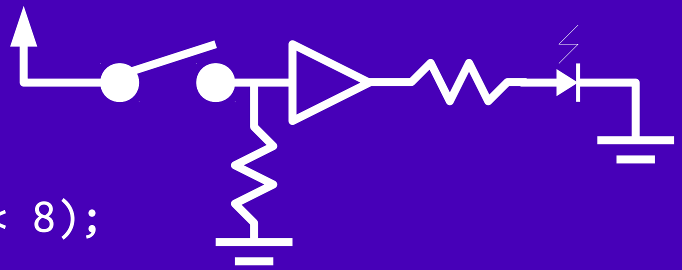
CMSIS

- Cortex Microcontroller Software Interface Standard
  - Definitions for registers and values with which to modify them
  - Usable with C structure mechanisms (->)
  - Need to build a project with Standard Peripheral firmware
  - Definitions are from provided header files.
- Open a project and look at the file:

  CMSIS/device/stm32f0xx.h

C code to copy pa0 to pc8

```
#include "stm32f0xx.h"

int main(void)
{
    RCC->AHBENR  |=  RCC_AHBENR_GPIOAEN|RCC_AHBENR_GPIOCEN;
    GPIOC->MODER &= ~GPIO_MODER_MODER8;
    GPIOC->MODER |=  GPIO_MODER_MODER8_0;
    GPIOA->PUPDR &= ~GPIO_PUPDR_PUPDR0;
    GPIOA->PUPDR |=  GPIO_PUPDR_PUPDR0_1;

    for(;;) {
        int status = (GPIOA->IDR & 1);
        GPIOC->BSRR = ((1<<8)<<16) | (status << 8);

        // I could have said, instead:
        // GPIOC->BSRR = (0x0100<<16) | ((GPIOA->IDR & 1) << 8);
    }
}
```



Putting assembly language inside a C function

- Called "inline assembly language."
- Different with every compiler.
- We're using GCC, and it works like this:
  - The asm() statement encapsulates assembly language (with labels, if you like) in a string.
  - Colon-separated definitions allow you to supply input and output arguments to the instructions, and a list of registers that are modified (clobbered) as side-effects.
  - The statement does not produce a value.
    - i.e. you can't say x=asm("…");

Inline assembly syntax

```
asm("instruction
       instruction
     label:
       instruction
       instruction"
    : <output operand list>
    : <input operand list>
    : <clobber list>);
```

Inline assembly example

```
int main(void) {
    int count = 0;
    for(;;) {
        count += 1;
        asm("nop");
        count += 1;
    }
}

// I'll bet you think this is too trivial.
// There is a subtle problem.
// The compiler might reorder the asm()
// statement if it thinks it's a good idea.
```

Inline assembly example

```c
int main(void) {
    int count = 0;
    for(;;) {
        count += 1;
        asm volatile("nop");
        count += 1;
    }
}

// volatile tells the compiler that it is
// important to leave the asm statement
// exactly where we put it.
```

Inline assembly example

```
void mywait(int x) {
    asm volatile("      mov r0, %0\n"
                 "again:\n"
                 "      nop\n"
                 "      nop\n"
                 "      nop\n"
                 "      nop\n"
                 "      sub r0, #1\n"
                 "      bne again\n"
                 :  : "r"(x) : "r0", "cc");
}

int main(void) {
    for(;;) {
        mywait(1000000);
    }
}
```

Reference to the operand zero.

List of all registers that are clobbered.

Empty output operand list

One input operand, x.
"r" means "put it in register r0-r12"

sub r0, #1 ?

- Why isn’t that subs r0,#1 ?
- Because inline assembly does not use unified syntax. Things are a little bit strange.

Inline assembly example

```
// Remember that C has no rotate operator.

int main(void) {
    int x = 1;
    for(;;) {
        asm volatile("ror %0,%1\n" : "+l"(x) : "l"(1) : "cc");
    }
}
```

%0 is a reference to the first operand.

%1 is a reference to the second operand.

X is an output operand.

+ means it is used as an input and an output.

`l` means use a Lower register (R0 – R7).

1 is an input operand. `l` says put it in a lower register.

Inline assembly constraints

- The "r" and the "`l`" and the "`+l`" are operand constraints.
  - They are different for every architecture, and you will have to look them up every time you use them.
    - Even if you are an expert with GCC.
  - Look up “GCC inline assembly contraints” for the complete story on this.

Register constraints

- You can force a variable to use a register with the register keyword:

```
register int x = 5;
```

- You can force a variable to use a specific register (in GCC) like this:

```
register int x asm("r6");
```