Lecture 3: Embedded Programming and Memory

A note about homeworks

Homeworks are graded on *good-faith effort*

Low-stakes way to introduce you to course material

You are not expected to know it already!

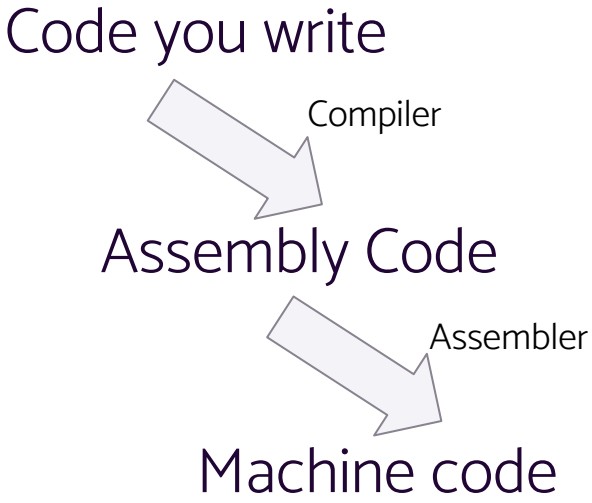
Set yourself a limit (time or emotional) and if you don't have an answer, explain why

Variety of embedded architectures - discussion

MCUs are varied

But knowing the theory of how a CPU, peripherals, memory work gives context to reading a data sheet

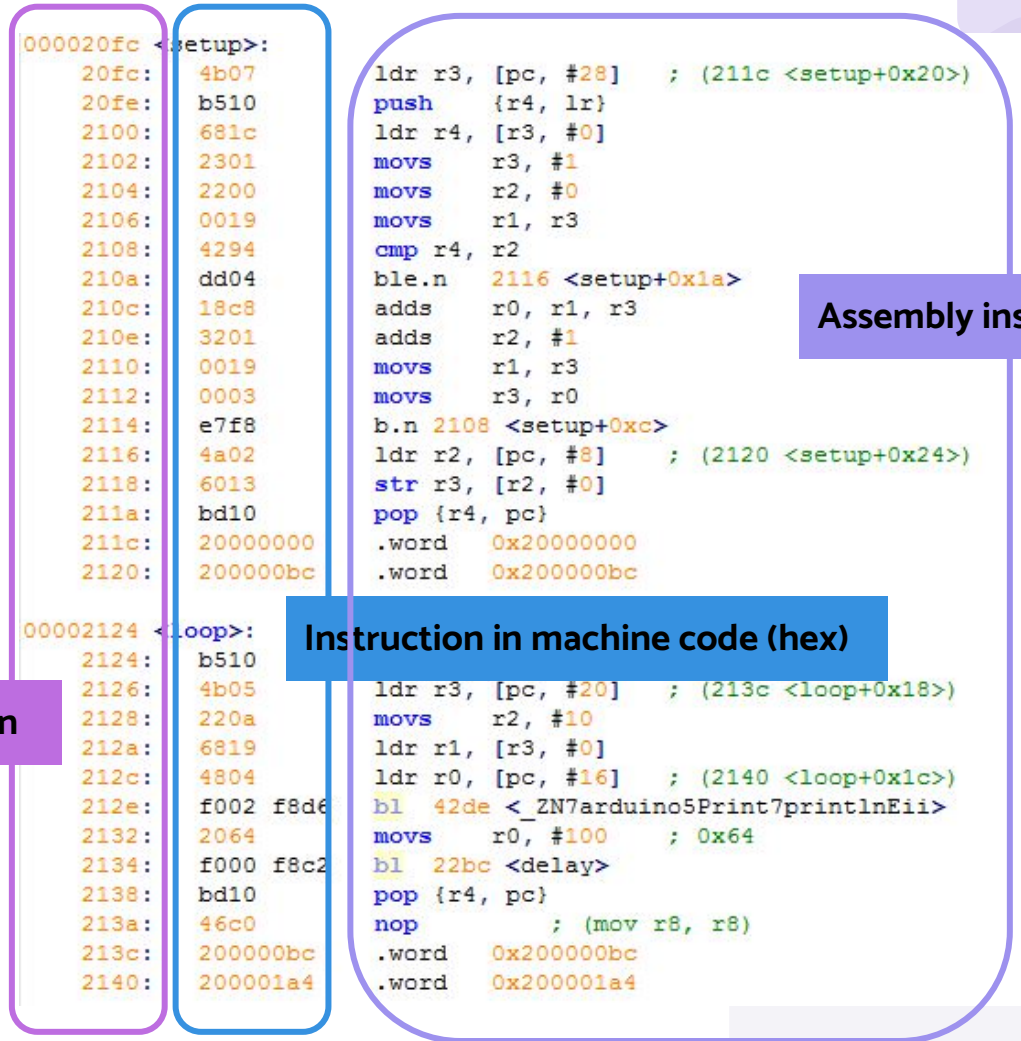
How software you write becomes code running on an MCU



Program

```
int N = 12;
int fibo = 0;

void setup() {
  int f_prev = 1;
  int f = 1;

  int i = 0;

  while (i < N) {
    int f_next = f + f_prev;
    f_prev = f;
    f = f_next;
    i += 1;
  }
  fibo = f;
}

void loop() {
  Serial.println(fibo);
  delay(100);
}
```

Assembly

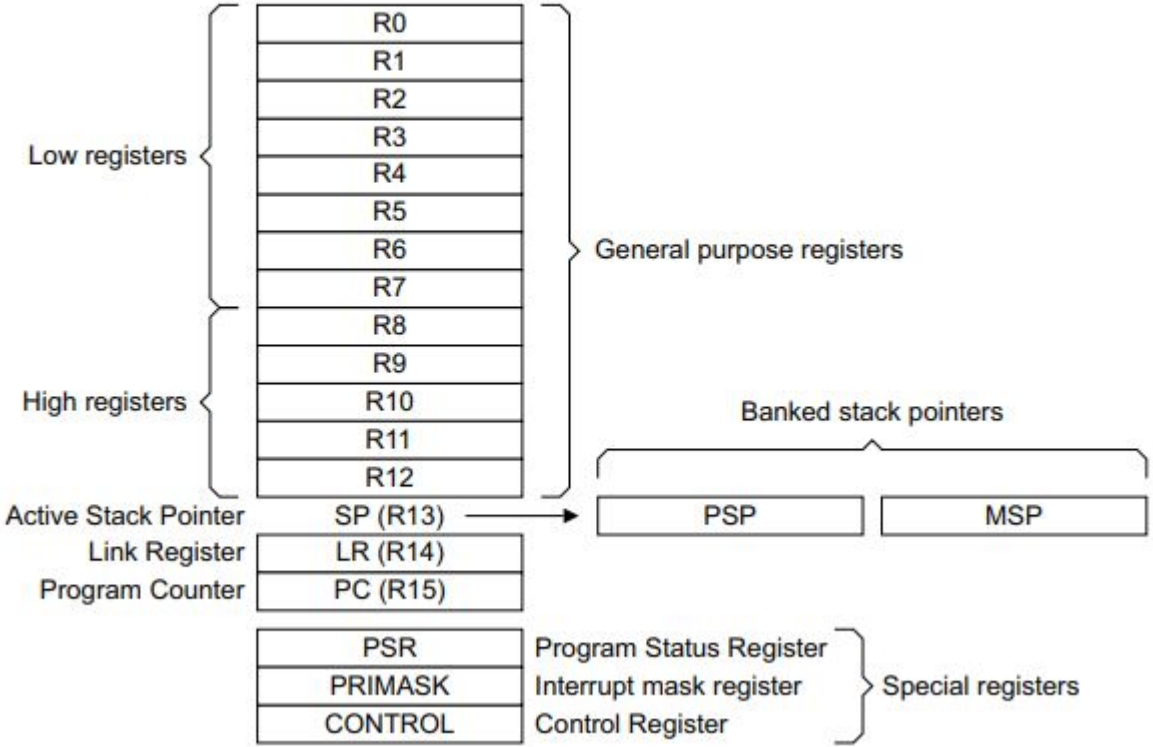

000020fc <setup>:
20fc: 4b07 ldr r3, [pc, #28] ; (211c <setup+0x20>)
20fe: b510 push {r4, lr}
2100: 681c ldr r4, [r3, #0]
2102: 2301 movs r3, #1
2104: 2200 movs r2, #0
2106: 0019 movs r1, r3
2108: 4294 cmp r4, r2
210a: dd04 ble.n 2116 <setup+0x1a>
210c: 18c8 adds r0, r1, r3
210e: 3201 adds r2, #1
2110: 0019 movs r1, r3
2112: 0003 movs r3, r0
2114: e7f8 b.n 2108 <setup+0xc>
2116: 4a02 ldr r2, [pc, #8] ; (2120 <setup+0x24>)
2118: 6013 str r3, [r2, #0]
211a: bd10 pop {r4, pc}
211c: 20000000 .word 0x20000000
2120: 200000bc .word 0x200000bc
00002124 <loop>:
2124: b510
2126: 4b05 ldr r3, [pc, #20] ; (213c <loop+0x18>)
2128: 220a movs r2, #10
212a: 6819 ldr r1, [r3, #0]
212c: 4804 ldr r0, [pc, #16] ; (2140 <loop+0x1c>)
212e: f002 f8d6 bl 42de <_ZN7arduino5Print7println Eii>
2132: 2064 movs r0, #100 ; 0x64
2134: f000 f8c2 bl 22bc <delay>
2138: bd10 pop {r4, pc}
213a: 46c0 nop ; (mov r8, r8)
213c: 200000bc .word 0x200000bc
2140: 200001a4 .word 0x200001a4
Assembly instructions
Instruction in machine code (hex)
Memory address of instruction

Resources used in this presentation

ARM Cortex M0+ devices generic user guide

ARMv6-M Architecture reference manual

The processor core registers are:



A2.3.1 ARM core registers

There are thirteen general-purpose 32-bit registers, R0-R12, and an additional three 32-bit registers that have special names and usage models:

SP Stack Pointer, used a pointer to the active stack. For usage restrictions see *Use of 0b1101 as a register specifier* on page A5-83. This is preset to the top of the Main stack on reset. See *The SP registers* on page B1-211 for more information. SP is sometimes referred to as R13.

LR Link Register stores the Return Link. This is a value that relates to the return address from a subroutine that is entered using a Branch with Link instruction. The LR register is also updated on exception entry, see *Exception entry behavior* on page B1-224. LR is sometimes referred to as R14.

Note

LR can be used for other purposes when it is not required to support a return from a subroutine.

PC Program Counter, see *Use of 0b1111 as a register specifier* on page A5-82 for more information. The PC is loaded with the Reset handler start address on reset. PC is sometimes referred to as R15.

Stack

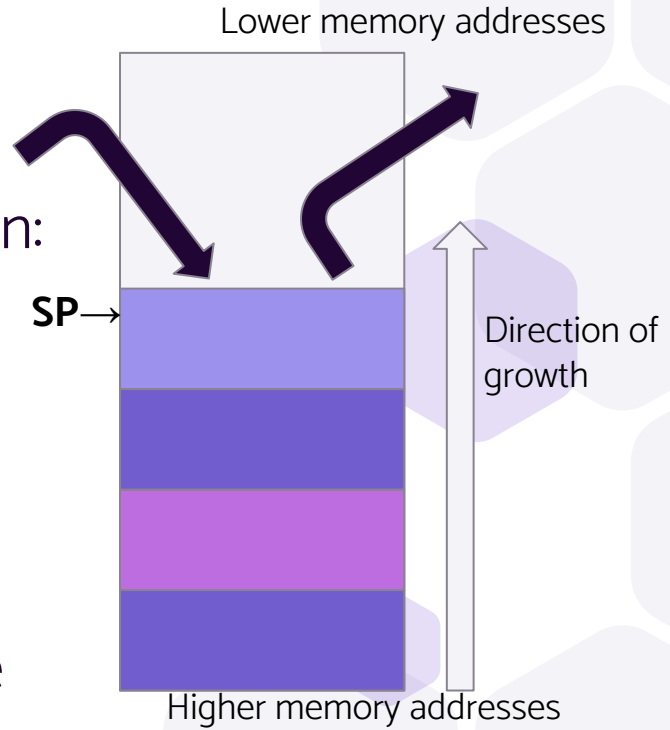
LIFO (last-in, first-out) data structure

Keeps track of information for execution:

- Local variables
- Return pointers

Grows “downward”

Stack Pointer (SP) points to latest value



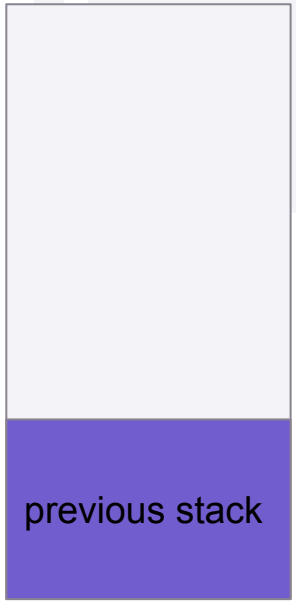
Cortex M0+ stack operations

push *reglist* - push the registers in *reglist* onto the stack (highest value registers pushed first), decrements stack pointer

pop *reglist* - pop the values on the stack into the registers in *reglist* (lowest value registers popped first)

if SP is in *reglist*, branch to where SP is pointing after pop

```
000020fc <setup>:
    20fc:   4b07        ldr r3, [pc, #28]   ; (211c <setup+0x20>)
    20fe:   b510        push    {r4, lr}
    2100:   681c        ldr r4, [r3, #0]
    2102:   2301        movs    r3, #1
    2104:   2200        movs    r2, #0
    2106:   0019        movs    r1, r3
    2108:   4294        cmp r4, r2
    210a:   dd04        ble.n   2116 <setup+0x1a>
    210c:   18c8        adds    r0, r1, r3
    210e:   3201        adds    r2, #1
    2110:   0019        movs    r1, r3
    2112:   0003        movs    r3, r0
    2114:   e7f8        b.n 2108 <setup+0xc>
    2116:   4a02        ldr r2, [pc, #8]    ; (2120 <setup+0x24>)
    2118:   6013        str r3, [r2, #0]
    211a:   bd10        pop {r4, pc}
    211c:   20000000    .word   0x20000000
    2120:   200000bc    .word   0x200000bc

00002124 <loop>:
    2124:   b510        push    {r4, lr}
    2126:   4b05        ldr r3, [pc, #20]   ; (213c <loop+0x18>)
    2128:   220a        movs    r2, #10
    212a:   6819        ldr r1, [r3, #0]
    212c:   4804        ldr r0, [pc, #16]   ; (2140 <loop+0x1c>)
    212e:   f002 f8d6   bl  42de <_ZN7arduino5Print7println Eii>
    2132:   2064        movs    r0, #100    ; 0x64
    2134:   f000 f8c2   bl  22bc <delay>
    2138:   bd10        pop {r4, pc}
    213a:   46c0        nop         ; (mov r8, r8)
    213c:   200000bc    .word   0x200000bc
    2140:   200001a4    .word   0x200001a4
```



R0	
R1	
R2	
R2	
R3	
R4	
R5	
R6	
R7	

LR	

Passing parameters?

Multiple conventions

- Pass on stack
- Pass as registers
- Combination (In gcc: first four arguments passed in registers, then stack)

What does the code that we looked at do?

Why learn about assembly when compilers exist?

Why

Break

Assembly to hex

```
20fe:    b510
```

Encode an instruction like b510

Stages of a microprocessor

Fetch - fetch next instruction from memory

Decode - decode instruction

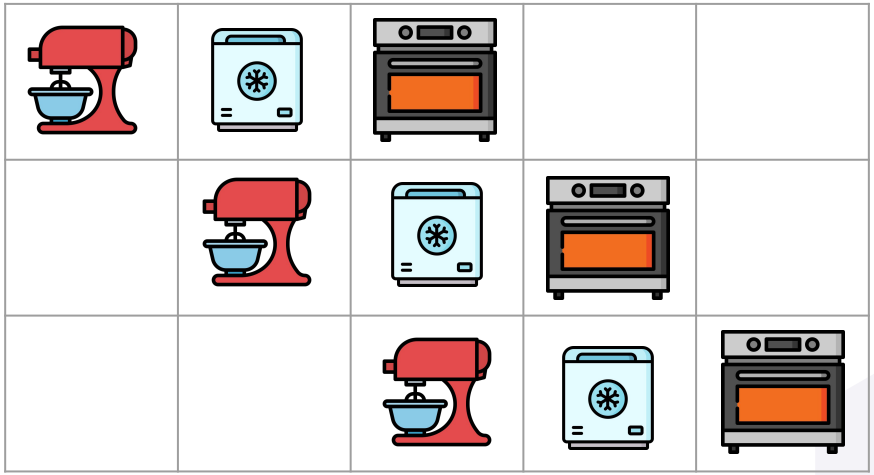
Execute - perform computation

- ALU (arithmetic logic unit): add, subtract, negate, bit operations
- Shift: used in multiplication/division

Memory access - read or write registers

Pipelining 

vs



images: Flaticon.com

Pipeline hazards (dependencies)

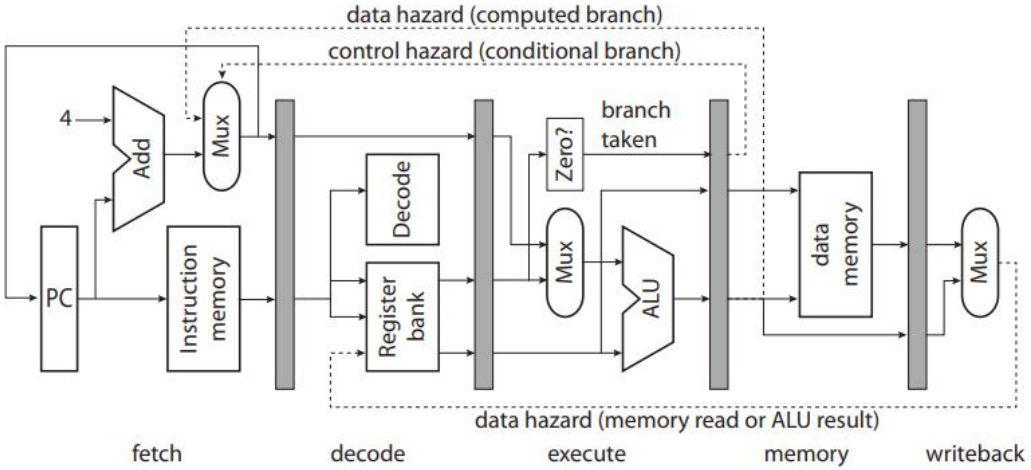


Figure 8.2: Simple pipeline (after Patterson and Hennessy (1996)).

Lee/Seshia 2017

Cortex-M0+

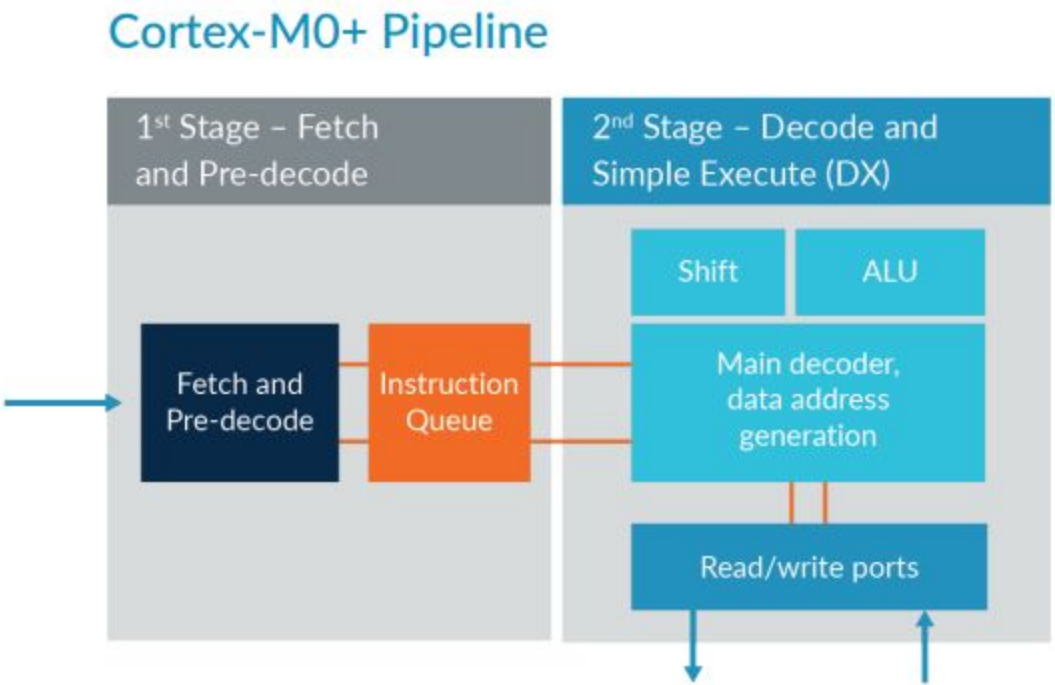


Image source

Memory

Information stored in memory:

- Code memory
- Stack memory
- Program data
- Heap (dynamically allocated data)
- Register file

```
Disassembly of section .data:

20000000 <N>:
20000000:       0000000c
```

Types of memory

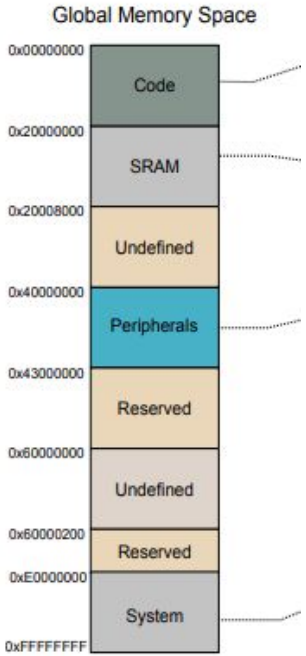
Volatile - Gets erased when power gets turned off

- RAM (DRAM, SRAM)

Non-volatile - Persists when power gets turned off

- Flash
- ROM (sometimes rewritable, like EEPROM)

Memory layout of SAMD 21 chip



Peripherals

Timers, ADCs, GPIO, etc

Controlled by special registers (**different** from CPU registers!)

“Memory-mapped”: from CPU perspective, just like writing to any other memory address

- From MCU perspective, need a controller to send data to the right place

You will see this in lab!

Specifications

x-bit processor:

- Data registers, data busses, *words* are that size

memory address may not be that size

- Common for 8-bit CPUs to have 16-bit addresses (**why?**)
- **What are the implications for atomicity?**

Harvard Architecture - code has separate memory space from data (common in MCUs)

- vs. Von Neumann - shared memory space (SAMD21 is Von Neumann)

How information gets onto an MCU

Bootloader

- Firmware on the board that can interface with the computer
- Copies memory on upload

Hardware programmer

- Special piece of hardware that connects to pins directly and transfers using a protocol

Why do we need to know about the type and layout of memory on an MCU?

Why

Break

Embedded programming

Reasons embedded programming differs from general-purpose computing:

- Cannot assume portability
- Parallelism from interrupts
- Limited by hardware
- Care more about scheduling/deadlines
- Safety-critical applications

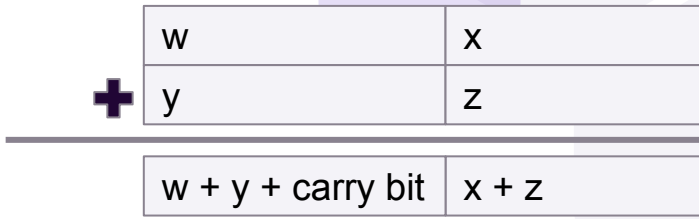
Portability

Word size

- `int` will mean different things on an 8-bit CPU vs a 32-bit CPU

Tip: be specific about size

- `int8`, `uint16`, etc



What if you need to emulate a 16-bit int on a 8-bit CPU?

- Fake it with multi-precision math!

Keywords for sharing data

static

- Value of local variable will persist between function calls
- Useful in a function like loop() when you don't want to declare a global variable

volatile

- Means variable can change outside of main execution (e.g. by an ISR)
- Tells compiler not to make certain optimizations

Floating point is often avoided in MCU applications. Why?

Why

Fixed point

Represent fractional values with implicit *fixed* divisor

- Decimal example: if fixed divisor were 1000, we would represent 0.04 as “40” (e.g. counting by milliseconds instead of seconds)
- In binary, we use powers of two as divisors
- Write format as “x.y”, with x digits before ~~decimal point~~ mantissa and y after

Fixed point example

Interpret the bits “01010110” in different formats:

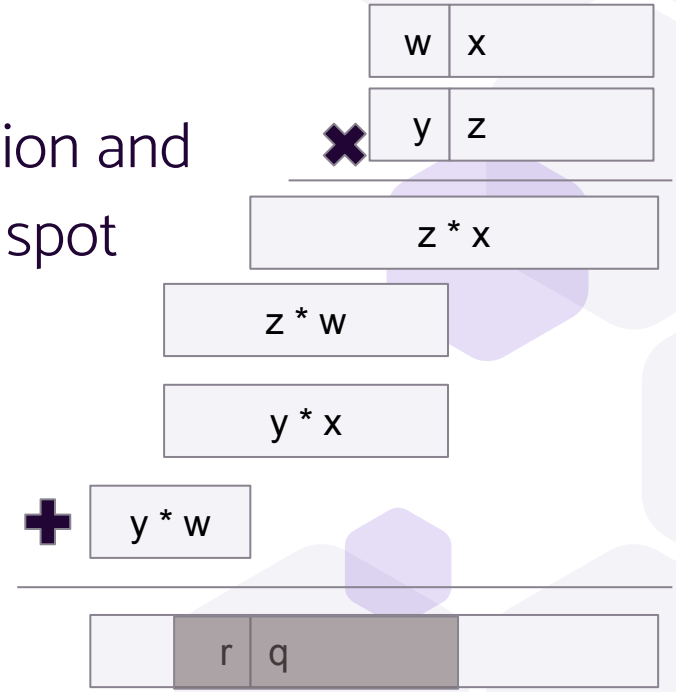
format	**regular/int**	**1.7**	**4.4**	**5.3**
divisor	n/a	2^7 = 128		
Interpreted value	86			

Fixed point math

Addition/subtraction work as usual

Let the CPU perform the computation and interpret the mantissa at the same spot

Mulitiplication: need to truncate



Summary

- Your code gets turned into assembly gets turned into machine code
- Machine code is executed on the CPU
- Data for programs is stored in different areas of memory
- Because of these architectures, embedded programming has some unique considerations

Project

Teams of 4 (make posts on Ed)

Propose a project that:

- Uses PWM, ADC, or DAC
- Has at least one interrupt service routine
- Has a watchdog timer (doesn't count as your ISR)

Uses at least one of:

- Serial communication
- Wifi
- Timer/counter

Final writeup will be required to have process and modeling & verification documentation

Send me your team/high-level idea by next Friday (Oct 1)

Proposal due two weeks from now (Oct 8)

Project ideas

- Games
  - Electronic whack-a-mole
- Music
  - Capacitive touch synthesizer
- Controllers
  - Keyboard, game controller, etc prototype
- Other
  - Plant moisture/light monitor