Native Integer and Fixed-Point Math

NATIVE DEVICE INTEGER MATH

Native Device Integer Math

- Basic idea: don't do unnecessary conversions
- Example: sensor which warns if temperature is at or below freezing (32°F)
  - Sensor indicates temperature with analog voltage
    - V_T = Temperature*12 mV/°F
  - Measure voltage with analog to digital converter
    - 10 bits, V_{ref} = 3.3 V
- Naïve approach
  - Measure voltage, convert to temperature
  - temperature = (ADC_result/1024.0)*3.3V / (12 mV/°F)

```
float temperature = ADC_result*0.268555;
if (temperature <= 32.0)
    Freeze_Warning();
```

- Native integer approach
  - Compare ADC result to value corresponding to freezing
  - `#define FREEZE_TEMP_CODE ((32°F * 12 mV/°F)*1024/3.3V) // is 119`

```
if (ADC_result <= FREEZE_TEMP_CODE)
     Freeze_Warning();
```

FIXED-POINT MATH

Limitations of Integer and Floating-Point Data Types

- Integers truncate the fractional part of the data

- Floating point is slow if there is no hardware support (must be emulated in software)
  - Floating point representation: S, M, E, B
    - S: indicates sign of value (0=+ or 1=-)
    - Mantissa M is scaled by base B raised to exponent E
    - Base B is 2, fixed for the format.
    - Value = $(-1)^{S}*M*B^{E}$

- Basic steps in performing floating point operation in software
  - Align mantissas by shifting them, adjusting exponents
  - Perform operation
  - Normalize result

Further Issues with Floating Point: IEEE-754

- Exponent (8 bits) is biased to allow negative exponents (small values)
  - Exponent of E is actually stored as E+127
- Mantissa (called *significand*) is 24 bits long, but only 23 bits are stored
  - Is normalized (shifted) to a value in range [1,2),
    - Now the first bit (left of binary point) is a 1.
    - Can delete that bit before storing since it will always be a 1.
  - Frees up another bit for better precision!
  - Resulting value is a fraction in range [0,1) and is used for storage
- These conversions take additional time in software.
  - Add or remove bias from exponent
  - Add or remove implicit 1 bit
- Good explanation: http://steve.hollasch.net/cgindex/coding/ieeefloat.html

Fixed Point Math – Why and How

- Basic Idea:
  - Locate the *radix point* so the values cover the **range** you need to represent with enough **resolution**
    - Range: difference between smallest and largest values which can be represented
    - Resolution: difference between two adjacent values
  - Constant number of discrete values possible (2^N)
- Naming Styles
- *qf*
  - *f* = no. of fraction bits
  - *i* = ? Implied as word size - f
- *Q i.f*
  - *i* = no. of integer bits
  - *f* = no. of fraction bits

3.34375 in a fixed point binary representation

Bit	1	1	0	1	0	1	1
Weight	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
Weight	2	1	½	¼	1/8	1/16	1/32

left: range ↓ Resolution ↓ 2 5

Radix Point

Bit Pattern	*Integer (q0)*	*q2*	*q10*
0000 0000	0/1 = 0	0/4 = 0	0/1024 = 0
0000 1110	28/1 = 28	28/4 = 7	28/1024 = 0.0273…
0000 1100	99/1 = 99	99/4 = 24.75	99/1024 = 0.0966…

Radix Point Locations

q10 *q2* *q0*

Dealing with Signed Values

- Two's complement for integers
  - Same instructions work for addition, subtraction
  - Different instruction for multiply, divide
- Two's complement for Fixed Point?
  - Possible, but more complicated
  - Many FXP implementations instead use sign-magnitude format, and convert as needed

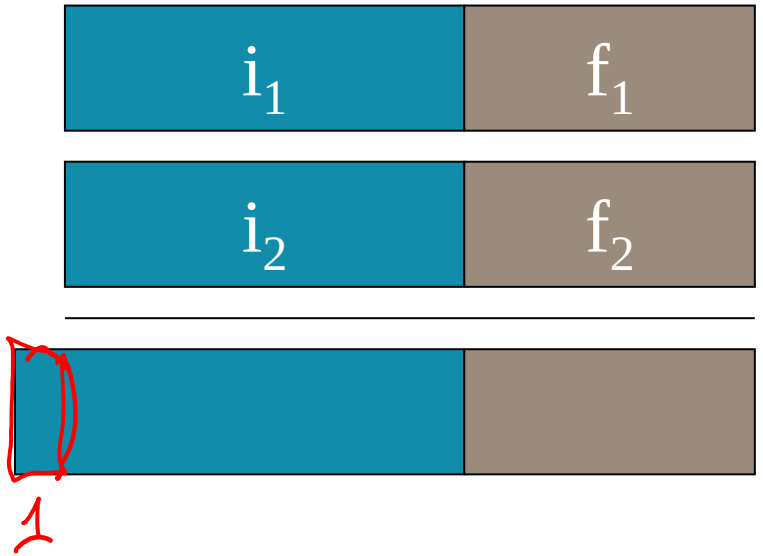
Support Operations

- Scaling
  - Shift a value to change from one (implicit) exponent to another
  - Left shift increases the number of fraction bits, right shift decreases it
  - Example: Convert from q10 to q6 by shifting right by 4 bits
  - Note: for a signed representation, this must be an arithmetic shift (MSB must remain the same)
- Normalization
  - Two values are normalized if they have the same number of fraction bits
  - Need to normalize values before addition or subtraction
- Promotion
  - Adds additional bits to improve range and/or precision
- Rounding
  - Improves accuracy by incrementing value if truncated bits are > 1/2

MATHEMATICAL OPERATIONS

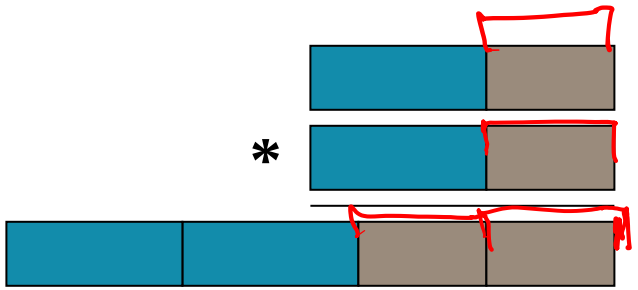
Addition, Subtraction ($qf_1 \pm qf_2$)

- Are values aligned ($f_1 == f_2$)?
  - We can treat fixed point numbers like integers
  - Radix point stays where it started
- Otherwise we need to align radix points (normalize)



- General Operation
  - Ensure operands are normalized (scale by f_1-f_2)
  - Add or subtract operands
  - Handle overflow
  - Set sign of result

Multiplication ($qf_1 * qf_2$)



10 * 1.25 = 12.5

6.2 Format

```
  001010.00
* 000001.01
-----------
00000000 1100.1000
```

- Operands do not have to be normalized
- Radix point moves **left**, so we need to normalize result afterwards, shifting the result **right**

- Operation
  - Multiply terms
  - Handle overflow
  - Scale result from $q(f_1+f_2)$ into desired format

Overflow

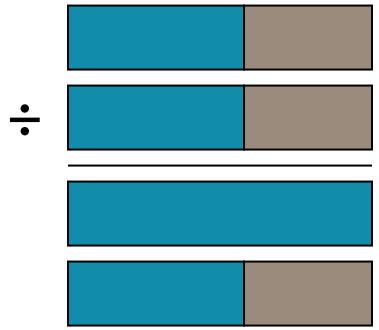
- Causes
  - Result of operation doesn't fit into representation
  - Adding two N bit values has an N+1 bit result
  - Multiplying two N bit values has a 2N bit result
    - C language discards upper N bits of integer multiplication
- Prevention
  - Promote operands to larger representation before performing operation
  - Scale operands down to have fewer fraction bits
- Compensation
  - Detect overflow after operation, then correct the result
- Saturation
  - Replace overflowing value with closest valid available value in that representation
  - May be provided in special instructions

Division

- Option 1: Multiply by the reciprocal of the divisor
  - Practical for constants
  - Also useful if processor has hardware reciprocal instruction but no divide

- Option 2: Use C integer division and remainder operations (/ and %, or div (stdlib.h))
  - /: Result is quotient of integer division, is integer (*q0*). Fraction bits have been truncated
  - %: Result is remainder of integer division,
  - div(): returns structure of type div_t (int quot, int rem)
- Option 3: Assembly language integer division instruction
  - Typically produces two results: integer quotient (*q0*) and remainder

Using Integer Division

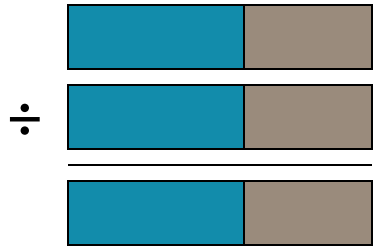


		q1 Format	q2 Format
Dividend	7	111.0	111.00
Divisor	÷ 2	÷ 010.0	÷ 010.00
Quotient	3	0011	00011
Remainder	1	001.0	001.00

- Perform integer divide on two fixed point operands with formats qf_1 and qf_2
- How many fraction bits are in the quotient?
  - If f_1=f_2, quotient is *integer*
  - Otherwise quotient has f_1-f_2 fraction bits
- Remainder has f_1 fraction bits

- Note: This example doesn't address signed numbers

Using Integer Division, Part II



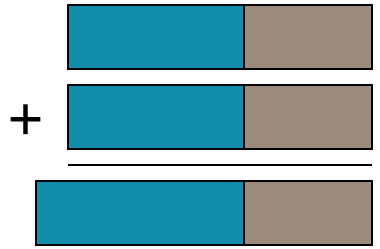
Dividend
Divisor
Quotient

	q1 Format	q2 Format
Dividend	7	(7*2) 1110.0	(7*4) 11100.00
Divisor	÷ 2	÷ 010.0	÷ 010.00
Quotient	3.5	0011.1	00011.10

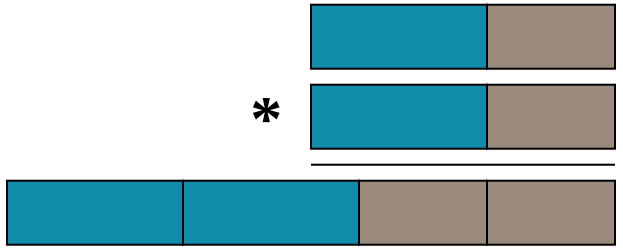
- What if we want result to have fraction bits?
- What if we want to use just the C integer divide operation (and not the modulo (remainder, %) operation)?

- Can scale dividend or divisor so quotient will have fraction bits
- To make quotient have same format as dividend and divisor (qf)
  - Multiply dividend by 2^{f2} (shift left by f_2 bits). Need to keep the extra bits or detect overflow!
  - Quotient is in qf fixed point format now

More Fixed Point Math Examples



		q4 Format
	10	0000 1010.0000
+	1.5	+0000 0001.1000
	11.5	00000000 1011.1000



		q4 Format
	9.0625	1001.0001
*	6.5	*0110.1000
	58.90625	0011 1010.1110 1000

Example Code: 28.4 Fixed Point Math

- This particular code is for unsigned numbers only! Must be tweaked to support signed numbers.
- Representation
  - `typedef int FX_28_4;`
- Converting to and from fixed point representation
  - `#define Y_BITS (4)`
  - `#define SCALE (1<<Y_BITS)`
  - `#define FL_TO_FX(a) (int)((a)*SCALE)`
  - `#define INT_TO_FX(a) ((a)*SCALE)`
  - `#define FX_TO_FL(a) ((a)/((float)SCALE)`
  - `#define FX_TO_INT(a) (int)((a)/SCALE)`
- Math
  - `#define FX_ADD(a,b) ((a)+(b))`
  - `#define FX_SUB(a,b) ((a)-(b))`
  - `#define FX_MUL(a,b) (((a)*(b))/SCALE)`
  - `#define FX_DIV(a,b) (((a)/(b))*SCALE)`
  - `#define FX_REM(a,b) (((a)%(b))`

incomplete

(((a)*SCALE)/(b))

To match P.59 example

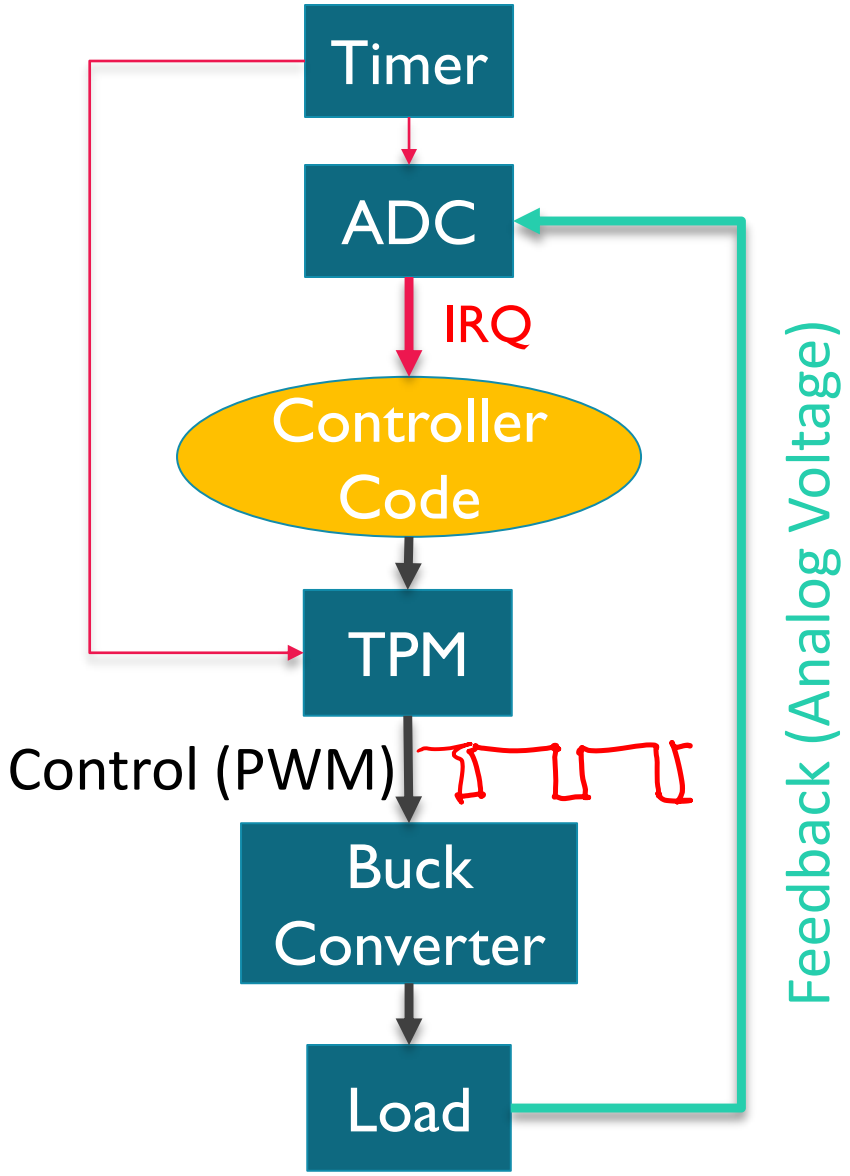
FIXED-POINT UPDATE_PID FUNCTION

Closed-Loop Control System Overview

- Provide closed-loop control of buck converter for correct and accurate output current control
- Sequence of activities

  - Periodic timer triggers ADC conversion with hardware signal
  - ADC conversion complete signal triggers ADC interrupt
  - ADC interrupt handler contains closed-loop controller code, which updates TPM with new duty cycle of PWM output

- Quality of control depends on control frequency f_c
- Control frequency f_c limited by
  - Overhead of responding to interrupt
  - Duration of controller code



Floating-Point PID Controller Implementation

```
typedef struct {
  float dState; // Last position input
  float iState; // Integrator state
  float iMax, iMin; // Maximum and minimum allowable integrator state
  float iGain, // integral gain
        pGain, // proportional gain
        dGain; // derivative gain
} SPid;
```

```
SPid plantPID = {0, // dState
  0, // iState
  LIM_DUTY_CYCLE, // iMax
  -LIM_DUTY_CYCLE, // iMin
  I_GAIN_FL, // iGain
  P_GAIN_FL, // pGain
  D_GAIN_FL  // dGain
};
```

```
float UpdatePID(SPid * pid, float error, float position){
  float pTerm, dTerm, iTerm;

  // calculate the proportional term
  pTerm = pid->pGain * error;
  // calculate the integral state with appropriate limiting
  pid->iState += error;
  if (pid->iState > pid->iMax)
    pid->iState = pid->iMax;
  else if (pid->iState < pid->iMin)
    pid->iState = pid->iMin;
  iTerm = pid->iGain * pid->iState; // calculate the integral term
  dTerm = pid->dGain * (position - pid->dState);
  pid->dState = position;

  return pTerm + iTerm - dTerm;
}
```

- Design philosophy
  - Start with easy version (floating point) and get it working
  - Then switch it over to fixed point

Fixed-Point Update PID Controller Function

- UpdatePID_FX called by ADC interrupt handler to determine new control signal (PWM duty cycle)
- Multiply operations expected to take much longer than add or subtract operations
- Can use timing debug output bit to evaluate progress through code

```
FX16_16 UpdatePID_FX(SPidFX * pid, FX16_16 error_FX, FX16_16 position_FX){
  FX16_16 pTerm, dTerm, iTerm, diff, ret_val;

  // calculate the proportional term
  pTerm = Multiply_FX(pid->pGain, error_FX);

  // calculate the integral state with appropriate limiting
  pid->iState = Add_FX(pid->iState, error_FX);
  if (pid->iState > pid->iMax)
    pid->iState = pid->iMax;
  else if (pid->iState < pid->iMin)
    pid->iState = pid->iMin;

  iTerm = Multiply_FX(pid->iGain, pid->iState); // calculate the integral term
  diff = Subtract_FX(position_FX, pid->dState);
  dTerm = Multiply_FX(pid->dGain, diff);
  pid->dState = position_FX;

  ret_val = Add_FX(pTerm, iTerm);
  ret_val = Subtract_FX(ret_val, dTerm);
  return ret_val;
}
```

Fixed-Point PID Controller Implementation: Types, + and -

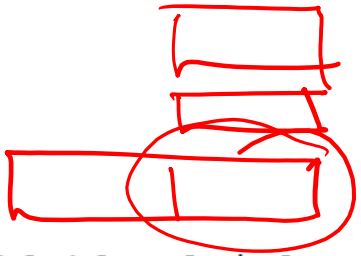
```
typedef int32_t FX16_16;

#define FL_TO_FX(x)  ((FX16_16)((x)*65536.0))
#define INT_TO_FX(x)  ((FX16_16)((x)*65536))
#define FX_TO_INT(x)  ((int32_t)((x)/65536))
#define FX_TO_FL(x)  ((float)((x)/65536.0))
```

why not shifts?

```
typedef struct {
  FX16_16 dState; // Last position input
  FX16_16 iState; // Integrator state
  FX16_16 iMax, iMin; // Maximum and minimum allowable integrator state
  FX16_16 iGain, // integral gain
          pGain, // proportional gain
          dGain; // derivative gain
} SPidFX;

FX16_16 Add_FX(FX16_16 a, FX16_16 b) {
  FX16_16 p;
  // Add. This will overflow if a+b > 2^16
  p = a + b;
  return p;
}

FX16_16 Subtract_FX(FX16_16 a, FX16_16 b) {
  FX16_16 p;
  p = a - b;
  return p;
}
```

- Simple implementation
  - Can use native 32-bit words
  - Simple because we ignore overflows and rounding!

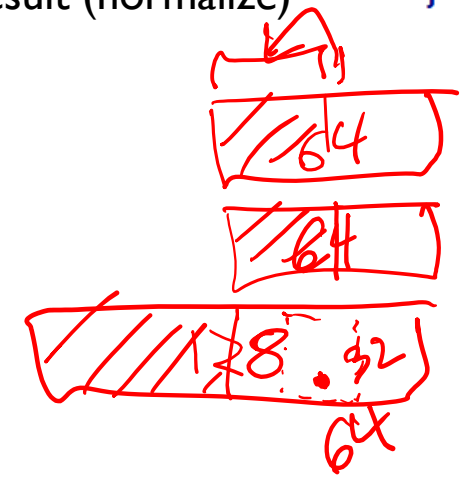
Signed 16.16 * 16.16

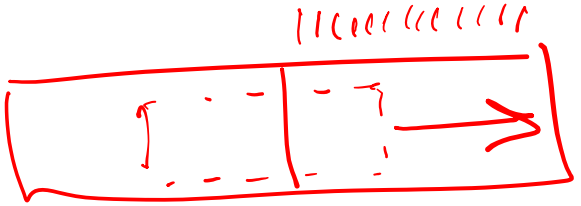


- Result of 32x32 multiply should be 64 bits long
  - C multiply of 32-bit integers just returns lower 32 bits of result
- Solution
  - Promote arguments to 64 bits
  - Multiply 64x64 to get 64-bit product
  - Process the result (normalize)

```
FX16_16 Multiply_FX(FX16_16 a, FX16_16 b) {
    int64_t p, pa, pb;
    // Long multiply first.
    pa = a;
    pb = b;
    p = pa * pb;
    // Should check for overflow!
    // Normalize after multiplication
    p >>= 16;
    return (FX16_16)(p&0xffffffff);
}
```

```
PUSH    {r4,lr}
ASRS    r4,r0,#31
ASRS    r3,r1,#31
MOV     r2,r1
MOV     r1,r4
BL      __aeabi_lmul
LSLS    r1,r1,#16
LSRS    r0,r0,#16
ORRS    r0,r0,r1
POP     {r4,pc}
```

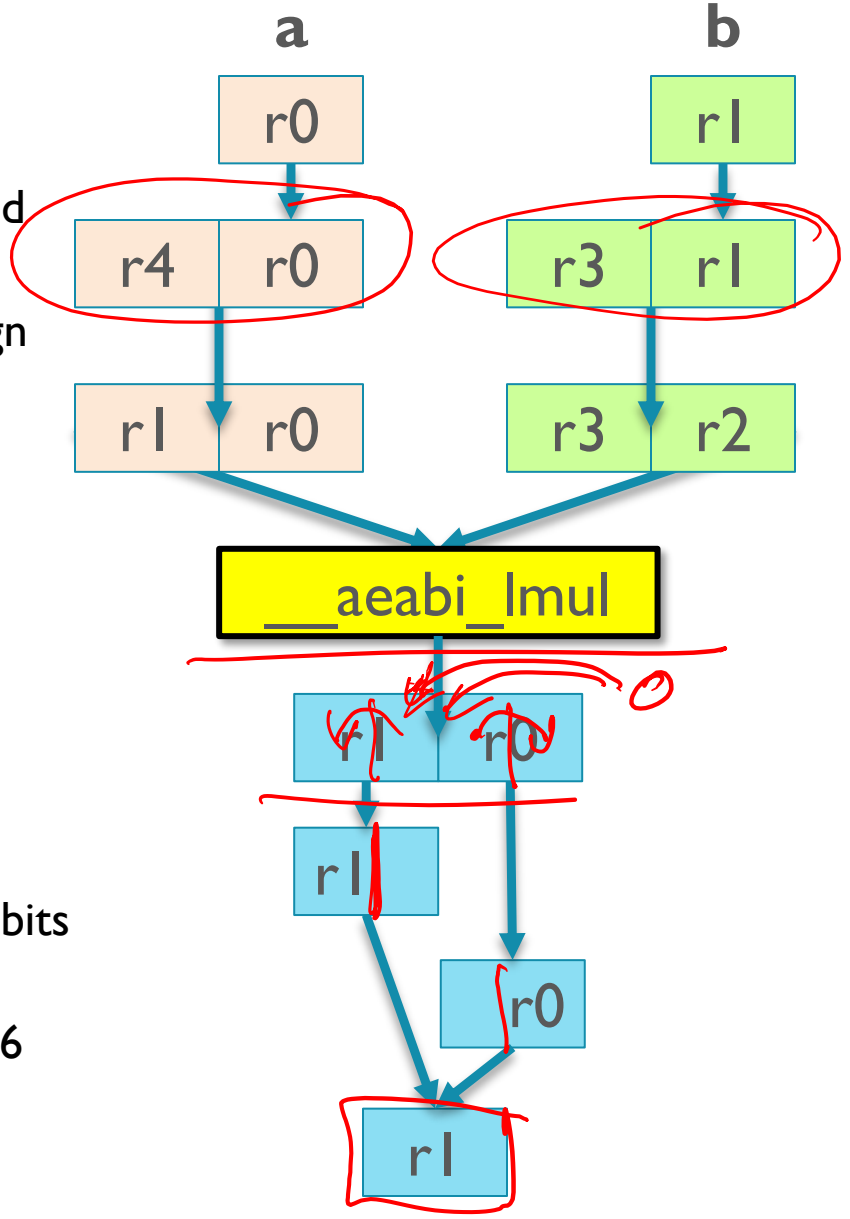




Signed 16.16 * 16.16 Explained

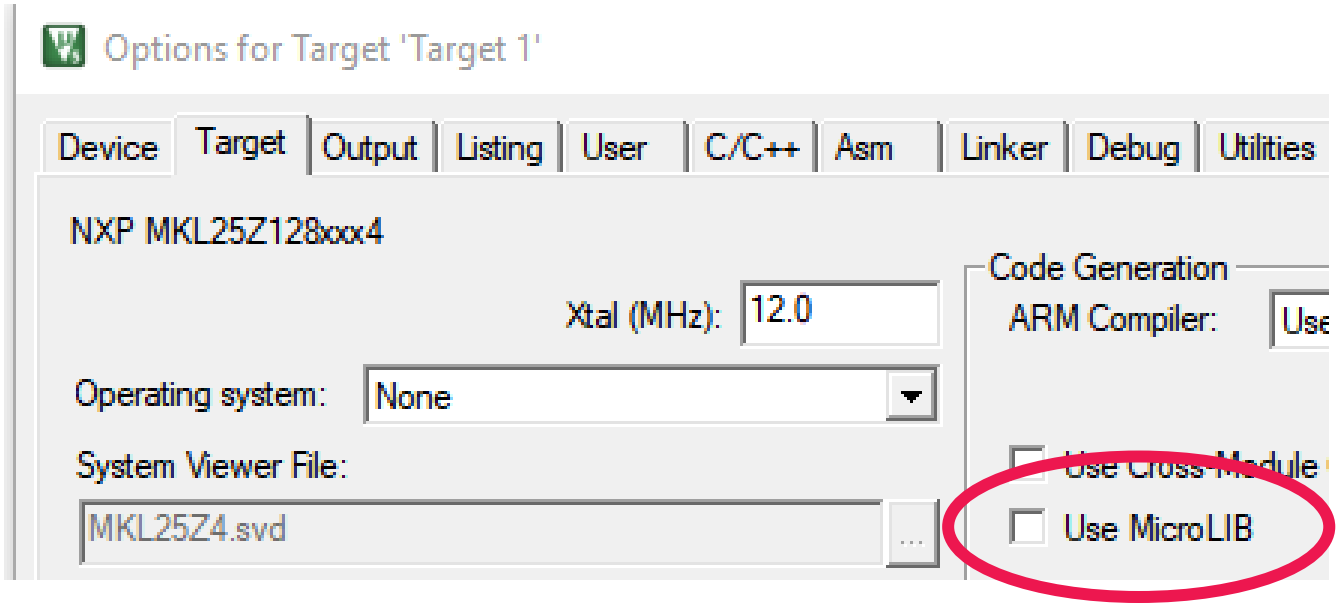
```
PUSH    {r4,lr}
ASRS    r4,r0,#31
ASRS    r3,r1,#31


MOV     r2,r1
MOV     r1,r4

BL      __aeabi_lmul

LSLS    r1,r1,#16
LSRS    r0,r0,#16
ORRS    r0,r0,r1

POP     {r4,pc}
```

- Sign-extension to 64 bits
  - a (r0) and b (r1) to 64 bits pa (r4:r0) and pb (r3:r2)
  - ASRS: arithmetic shift right performs sign extension by setting all of upper word's sign bits to match lower word's sign
- Move pa and pb into argument registers (r1:r0 and r3:r2)
- Call __aeabi_lmul for long multiply
- Extract middle 32 bits of result
  - LSLS: logical shift left extracts lower 16 bits of r1
  - LSRS: logical shift right extracts upper 16 bits of r0
  - ORRS: merges together middle 32 bits



Speed?

- MicroLIB version is much slower than standard C library version! Why?



Object Code for __aeabi_lmul in MicroLib C Library

```
__aeabi_lmul
_ll_mul
    0x000000fc:    b5f0        ..      PUSH    {r4-r7,lr}
    0x000000fe:    b41f        ..      PUSH    {r0-r4}
    0x00000100:    b086        ..      SUB     sp,sp,#0x18
    0x00000102:    2000        .       MOVS    r0,#0
    0x00000104:    9000        ..      STR     r0,[sp,#0]
    0x00000106:    9001        ..      STR     r0,[sp,#4]
    0x00000108:    9002        ..      STR     r0,[sp,#8]
    0x0000010a:    9806        ..      LDR     r0,[sp,#0x18]
    0x0000010c:    9906        ..      LDR     r1,[sp,#0x18]
    0x0000010e:    b280        ..      UXTH    r0,r0
    0x00000110:    9004        ..      STR     r0,[sp,#0x10]
    0x00000112:    9807        ..      LDR     r0,[sp,#0x1c]
    0x00000114:    0c09        ..      LSRS    r1,r1,#16
    0x00000116:    0402        ..      LSLS    r2,r0,#16
    0x00000118:    1400        ..      ASRS    r0,r0,#16
    0x0000011a:    4311        .C      ORRS    r1,r1,r2
    0x0000011c:    9007        ..      STR     r0,[sp,#0x1c]
    0x0000011e:    2700        .'      MOVS    r7,#0
    0x00000120:    9d08        ..      LDR     r5,[sp,#0x20]
    0x00000122:    9806        ..      LDR     r0,[sp,#0x24]
    0x00000124:    9106        ..      STR     r1,[sp,#0x18]
    0x00000126:    4646        FF      MOV     r6,r7
    0x00000128:    4644        DF      MOV     r4,r7
    0x0000012a:    9003        ..      STR     r0,[sp,#0xc]
    0x0000012c:    9803        ..      LDR     r0,[sp,#0xc]
    0x0000012e:    b2a9        ..      UXTH    r1,r5
    0x00000130:    0402        ..      LSLS    r2,r0,#16
    0x00000132:    0c00        ..      LSRS    r0,r0,#16
    0x00000134:    9003        ..      STR     r0,[sp,#0xc]
    0x00000136:    9804        ..      LDR     r0,[sp,#0x10]
    0x00000138:    0c2d        -.      LSRS    r5,r5,#16
    0x0000013a:    4315        .C      ORRS    r5,r5,r2
    0x0000013c:    4348        HC      MULS    r0,r1,r0
    0x0000013e:    2100        .!      MOVS    r1,#0
    0x00000140:    4622        "F      MOV     r2,r4
    0x00000142:    f000f8f1    ....    BL      __aeabi_llsl ; 0x328
    0x00000146:    19c7        ..      ADDS    r7,r0,r7
    0x00000148:    4171        qA      ADCS    r1,r1,r6
    0x0000014a:    3410        .4      ADDS    r4,r4,#0x10
    0x0000014c:    460e        .F      MOV     r6,r1
    0x0000014e:    2c40        @,      CMP     r4,#0x40
    0x00000150:    dbce        ..      BLT     0x12c ; __aeabi_lmul + 48
    0x00000152:    4638        8F      MOV     r0,r7
    0x00000154:    9a02        ..      LDR     r2,[sp,#8]
    0x00000156:    f000f8e7    ....    BL      __aeabi_llsl ; 0x328
    0x0000015a:    9a00        ..      LDR     r2,[sp,#0]
    0x0000015c:    9b01        ..      LDR     r3,[sp,#4]
    0x0000015e:    1880        ..      ADDS    r0,r0,r2
    0x00000160:    9000        ..      STR     r0,[sp,#0]
    0x00000162:    4159        YA      ADCS    r1,r1,r3
    0x00000164:    9802        ..      LDR     r0,[sp,#8]
    0x00000166:    9101        ..      STR     r1,[sp,#4]
    0x00000168:    3010        .0      ADDS    r0,r0,#0x10
    0x0000016a:    9002        ..      STR     r0,[sp,#8]
    0x0000016c:    2840        @(      CMP     r0,#0x40
    0x0000016e:    dbcc        ..      BLT     0x10a ; __aeabi_lmul + 14
    0x00000170:    9800        ..      LDR     r0,[sp,#0]
    0x00000172:    b00b        ..      ADD     sp,sp,#0x2c
    0x00000174:    bdf0        ..      POP     {r4-r7,pc}
```

What's going on here?

Two nested loops?

Did the compiler forget about the MULS instruction?

Could use Ghidra to make sense of function using control flow graph.

Object Code for __aeabi_lmul in Regular C Library

- Much shorter code!
- Extended precision integer math computes partial products
- Library code computes product with 6 multiplies
  - Are 6 really needed?
- Could you optimize this for the fixed point PID controller knowing it has limited input data ranges?

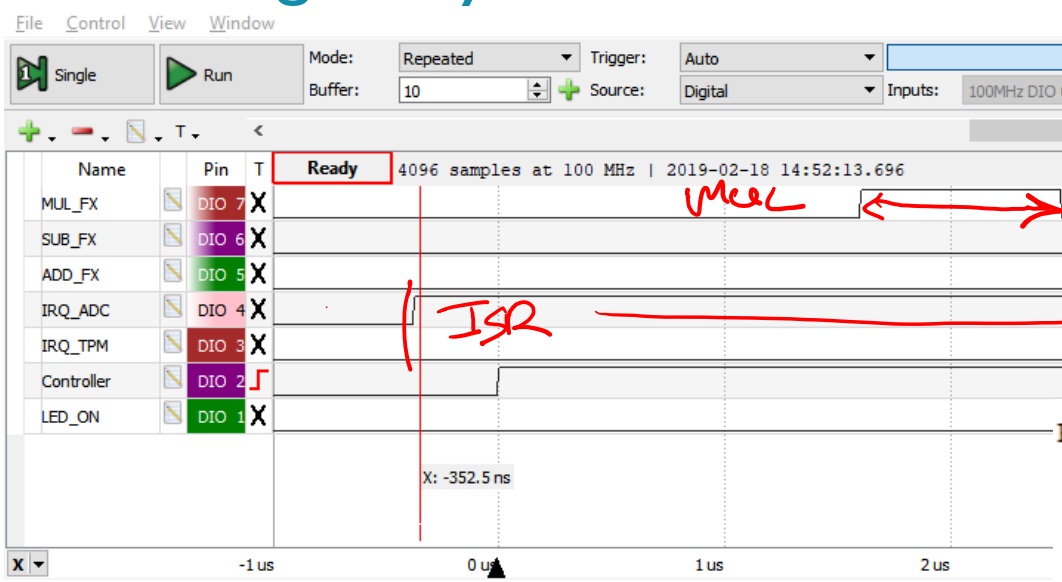
```
__aeabi_lmul
__ll_mul
    0x000001a4:    4343        CC      MULS    r3,r0,r3
    0x000001a6:    4351        QC      MULS    r1,r2,r1
    0x000001a8:    b530        0.      PUSH    {r4,r5,lr}
    0x000001aa:    185c        \.      ADDS    r4,r3,r1
    0x000001ac:    0c01        ..      LSRS    r1,r0,#16
    0x000001ae:    0c13        ..      LSRS    r3,r2,#16
    0x000001b0:    460d        .F      MOV     r5,r1
    0x000001b2:    b292        ..      UXTH    r2,r2
    0x000001b4:    435d        ]C      MULS    r5,r3,r5
    0x000001b6:    b280        ..      UXTH    r0,r0
    0x000001b8:    4351        QC      MULS    r1,r2,r1
    0x000001ba:    192c        ,.      ADDS    r4,r5,r4
    0x000001bc:    4605        .F      MOV     r5,r0
    0x000001be:    4355        UC      MULS    r5,r2,r5
    0x000001c0:    0c0a        ..      LSRS    r2,r1,#16
    0x000001c2:    0409        ..      LSLS    r1,r1,#16
    0x000001c4:    194d        M.      ADDS    r5,r1,r5
    0x000001c6:    4162        bA      ADCS    r2,r2,r4
    0x000001c8:    4358        XC      MULS    r0,r3,r0
    0x000001ca:    0c01        ..      LSRS    r1,r0,#16
    0x000001cc:    0400        ..      LSLS    r0,r0,#16
    0x000001ce:    1940        @.      ADDS    r0,r0,r5
    0x000001d0:    4151        QA      ADCS    r1,r1,r2
    0x000001d2:    bd30        0.      POP     {r4,r5,pc}
```

Do We Really Need a Full 64x64 Multiply?

- Need extended precision math
  - Must compute partial products with native precision (32 bits)
- Desired result: p = pa*pb: 64-bit arguments pa & pb, 64-bit return value p (upper 64 bits truncated)

- W = weight of single 32-bit register = 2^{32}
- p = $(pa_h*W + pa_l)*(pb_h*W + pb_l)$
- p = $pa_h*W*pb_h*W + pa_h*W*pb_l + pb_h*W*pa_l + pa_l*pb_l$
  - Upper 64 bits are not needed, since truncated to fit 64-bit return value
- $p_{Low64} = pa_h*W*pb_l + pb_h*W*pa_l + pa_l*pb_l$
  - Five multiplies and two adds
    - Replace multiply by W with using upper word register
  - Three multiplies and two adds
- Opportunity for optimization!

Timing Analysis and Room for Further Improvement



- Evaluate time spent in code
  - Is 900 ns reasonable for Multiply_FX?
- Where else can we trim time?

0.9.48 ≈ 43 cycles

```
FX16_16 UpdatePID_FX(SPidFX * pid, FX16_16 error_FX, FX16_16 position_FX)
  FX16_16 pTerm, dTerm, iTerm, diff, ret_val;

  // calculate the proportional term
  pTerm = Multiply_FX(pid->pGain, error_FX);

  // calculate the integral state with appropriate limiting
  pid->iState = Add_FX(pid->iState, error_FX);
  if (pid->iState > pid->iMax)
    pid->iState = pid->iMax;
  else if (pid->iState < pid->iMin)
    pid->iState = pid->iMin;

  iTerm = Multiply_FX(pid->iGain, pid->iState); // calculate the integral
  diff = Subtract_FX(position_FX, pid->dState);
  dTerm = Multiply_FX(pid->dGain, diff);
  pid->dState = position_FX;

  ret_val = Add_FX(pTerm, iTerm);
  ret_val = Subtract_FX(ret_val, dTerm);
  return ret_val;
}
```

CORTEX-M0+ AND CMSIS-DSP FIXED POINT MATH SUPPORT

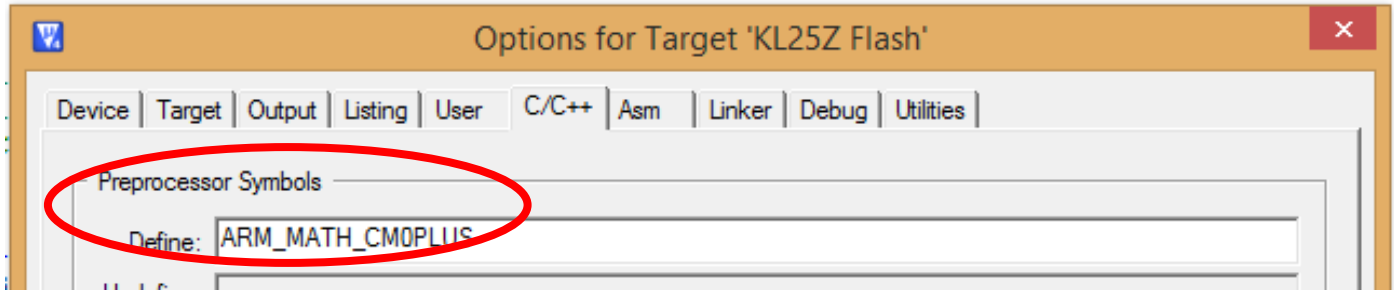
Cortex M0+ CPU Core Support

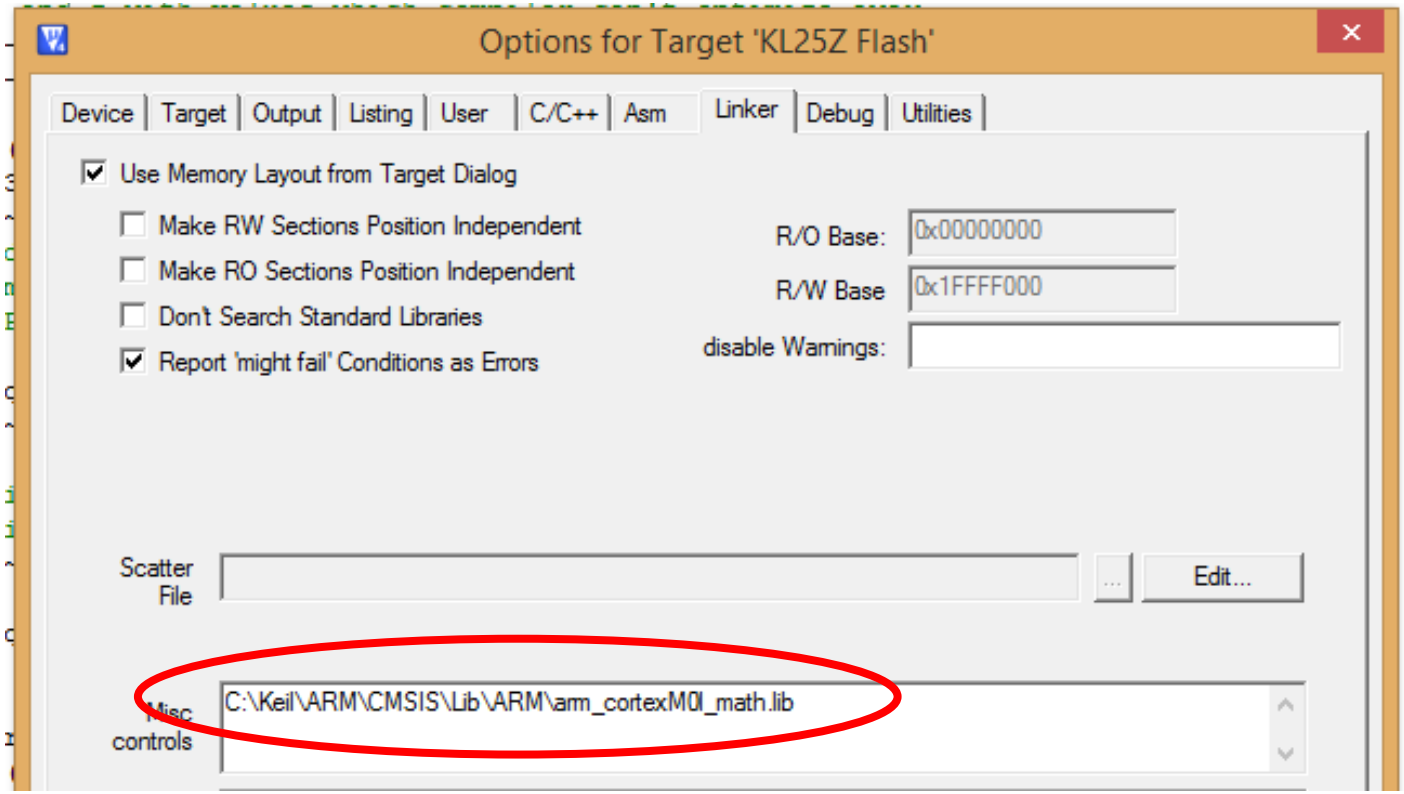
- 32-bit data in registers
- Add, Subtract
  - ADDS, ADCS, SUBS, SBCS
- Multiply: MULS
  - Signed multiply
  - Returns lower 32 bits of 64 bit product
  - Updates N, Z condition flags in APSR
- Shift
  - Logical shift left, right – does not preserve sign
  - Arithmetic shift right – preserves sign
- Rotate
  - Rotate right
- Extend (sign, zero)
  - SXTH, SXTB
  - UXTH, UXTB

CMSIS-DSP Support for Fixed Point Math

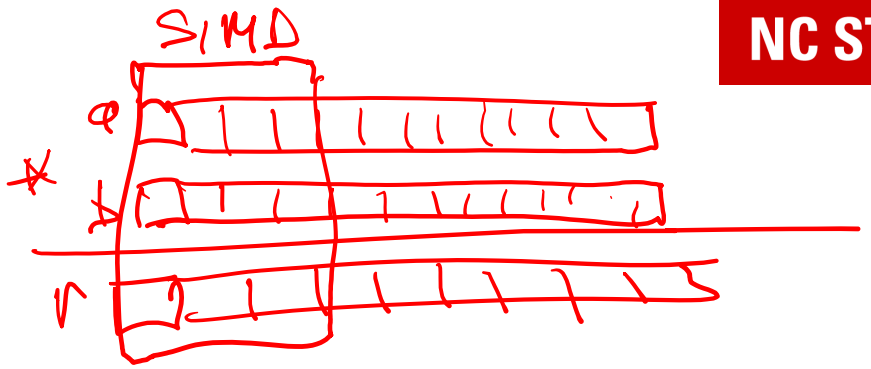
- Three fractional fixed point data types supported
  - q31_t: signed, 1 integer bit, 31 fraction bits
  - q15_t: signed, 1 integer bit, 15 fraction bits
  - q7_t: signed, 1 integer bit, 7 fraction bits. Not supported by all functions.
- Fractional: range is -1 to +1 (almost)

- To use CMSIS-DSP library
  - #include <arm_math.h>
  - C/C++ Tab: Define preprocessor symbol ARM_MATH_CM0PLUS
  - Linker Tab: Specify ARM math library to use, with location





CMSIS-DSP Software Library



- Fast functions and macros for digital signal processing and other math
  - Basic math – abs, add, sub, multiply, negate, offset, scale, shift (all support vector operations)
  - Fast math – sin, cos, sqrt
  - Complex math
  - Filters – IIR, FIR, convolution, correlation, FIR LMS, interpolator
  - Matrix
  - Transforms – FFT, DCT
  - Motor control – PID control, Clarke & Park (and inverse) transforms
  - Statistical – min, max, mean, power, rms, standard deviation, variance
  - Support – vector fill & copy, convert values
  - Interpolation – linear, bilinear
- Multiple data types supported
  - 32-bit floating point
  - 32-bit integer/fixed point
  - 16-bit integer/fixed point
  - 8-bit integer/fixed point
- Different versions optimized for core
  - M0, M0+, M1, M3, M4, M7, M23, M33
  - Vector functions use SIMD instructions on M4, M7, M33 (else use loop)
- Detailed documentation available in MDK
  - Help->Open Books Window, then select Tool User's Guide-> CMSIS Documentation

Example Code – Pythagorean Theorem

- $c = \sqrt{a^2 + b^2}$
- Equivalent C code
  - c = sqrtf((a*a) + (b*b));
- To use fixed point functions, we need to break expression into individual operations
  - a_2 = a*a;
  - b_2 = b*b;
  - sum = a_2 + b_2;
  - c = sqrtf(sum);
- Note that not all arguments are passed as pointers

```
q31_t pyth(q31_t a, q31_t b) {
  q31_t a_2, b_2, sum, result;

  arm_mult_q31(&a, &a, &a_2, 1);
  arm_mult_q31(&b, &b, &b_2, 1);
  arm_add_q31(&a_2, &b_2, &sum, 1);
  arm_sqrt_q31(sum, &result);
  return result;
}
```

length 1

Performance Evaluation

- How much faster than floating point math is the fixed point math?

- It depends… measure it with your data