

---

# Fixed-Point Math and Other Optimizations

# Fixed Point Math – Why and How

Floating point is too slow and integers truncate the data

- Floating point subroutines: slower than native, overhead of passing arguments, calling subroutines... simple fixed point routines can be in-lined

Basic Idea: put the *radix point* where covers the range of numbers you need to represent

I.F Terminology

- I = number of integer bits
- F = number 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	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$

  
Radix Point

Bit Pattern	<i>Integer</i>	<i>6.2</i>	<i>1.10</i>
000 0000 0000	$0/1 = 0$	$0/4 = 0$	$0/1024 = 0$
000 0001 1100	$28/1 = 28$	$28/4 = 7$	$28/1024 = 0.0273...$
000 0110 0011	$99/1 = 99$	$99/4 = 24.75$	$99/1024 = 0.0966...$

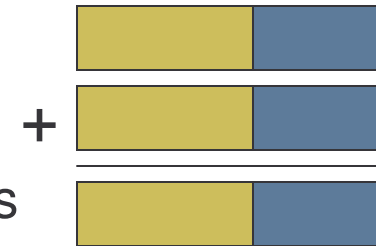


Radix Point Locations

# Rules for Fixed Point Math

## Addition, Subtraction

- Radix point stays where it started
- ...so we can treat fixed point numbers like integers



## Multiplication





- Radix point moves left by F digits
- ... so we need to normalize result afterwards, shifting the result right by F digits



$$\begin{array}{r}
 10 \\
 * 1.25 \\
 \hline
 12.5
 \end{array}$$

$$\begin{array}{r}
 \text{6.2 Format} \\
 001010.00 \\
 *000001.01 \\
 \hline
 00000000 \ 1100.1000
 \end{array}$$




# Division

		<i>Dividend</i>	7	<i>3.1 Format</i>	<i>3.2 Format</i>
÷		<i>Divisor</i>	÷ 2	111.0	111.00
		<i>Quotient</i>	3	÷ 010.0	÷ 010.00
		<i>Remainder</i>	1	0011	00011
				001.0	001.00

## Division

- Quotient is *integer*, may want to convert back to fixed point by shifting
- Radix point doesn't move in remainder

# Division, Part II

		<i>Dividend</i>		<b>3.1 Format</b>		<b>3.2 Format</b>
÷		<i>Divisor</i>	$7$	$(7*2) 1110.0$		$(7*4) 11100.00$
		<i>Quotient</i>	$\div 2$	$\div 010.0$		$\div 010.00$
			$3$	$0011.1$		$00011.10$

## Division

- To make quotient have same format as dividend and divisor, multiply dividend by  $2^F$  (shift left by F bits)
- Quotient is in fixed point format now

# Example Code for 12.4 Fixed Point Math

## Representation

- `typedef unsigned int FX_12_4;`

## Converting to and from fixed point representation

- `#define FL_TO_FX(a) (unsigned int)(a*16.0)`
- `#define INT_TO_FX(a) (a<<4)`
- `#define FX_TO_FL(a) (float)(a/16.0)`
- `#define FX_TO_INT(a) (unsigned int)(a>>4)`

## Math

- `#define FX_ADD(a,b) (a+b)`
- `#define FX_SUB(a,b) (a-b)`
- `#define FX_MUL(a,b) ((a*b)>>4)`
- `#define FX_DIV(a,b) ((a/b)<<4)`
- `#define FX_REM(a,b) ((a%b))`

# More Fixed Point Math Examples

*8.4 Format*

				$10$		$0000$	$1010.0000$
*				$+ 1.5$		$+0000$	$0001.1000$
				$11.5$		$00000000$	$1100.1000$

*4.4 Format*

				$9.0625$		$1001.0001$	
*				$* 6.5$		$*0110.1000$	
						$0011$	$1010.1110$
							$1000$

# Static Revisited

---

## Static variable

- A local variable which retains its value between function invocations
- Visible only within its module, so compiler/linker can allocate space more wisely (recall limited pointer offsets)

## Static function

- Visible only within its module, so compiler knows who is calling the functions,
- Compiler/linker can locate function to optimize calls (short call)
- Compiler can also inline the function to reduce run-time and often code size



# Volatile and Const

---

## Volatile variable

- Value can be changed outside normal program flow
  - ISR, variable is actually a hardware register
- Compiler reloads the variable from memory each time it is used

## Const variable

- const does not mean constant, but rather read-only
- consts are implemented as real variables (taking space in RAM) or in ROM, requiring loading operations (often requiring pointer manipulation)
  - A #define value can be converted into an immediate operand, which is much faster
  - So avoid them

## Const function parameters

- Allow compiler to optimize, as it knows a variable passed as a parameter hasn't been changed by the function

# More

---

## Const Volatile Variables

- Yes, it's possible
  - Volatile: A memory location that can change unexpectedly
  - Const: it is only read by the program
- Example: hardware status register

# Starting Points for Efficient Code

---

Write correct code first, optimize second.

Use a top-down approach.

Know your microprocessors' architecture, compiler (features and also object model used), and programming language.

Leave assembly language for unported designs, interrupt service routines, and frequently used functions.

# Floating Point Data Type Specifications

Use the smallest adequate data type, or else...

- Conversions without an FPU are very slow
- Extra space is used
- C standard allows compiler or preprocessor to convert automatically, slowing down code more

Single-precision (SP) vs. double-precision (DP)

- ANSI/IEEE 754-1985, *Standard for Binary Floating Point Arithmetic*
  - Single precision: 32 bits
    - 1 sign bit, 8 exponent bits, 23 fraction bits
  - Double precision
    - 1 sign bit, 11 exponent bits, 52 fraction bits
- Single-precision is likely all that is needed
- Use single-precision floating point specifier “f”

# Floating-Point Specifier Example

No “f”

```
float res = val / 10.0;
```

Assembler:

```
move.l -4(a6),-(sp)
jsr __ftod //SP to DP
clr.l -(sp)
move.l #1076101120,-
(sp)
jsr __ddiv //DP divide
jsr __dtof //DP to SP
move.l (s)+,-8(a6)
```

“f”

```
float res = val / 10.0 f;
```

Assembler:

```
move.l -4(a6),-(sp)
move.l #1076101120,-
(sp)
jsr __fdiv
move.l (s)+,-8(a6)
```

# Automatic Promotions

---

Standard math routines usually accept double-precision inputs and return double-precision outputs.

Again it is likely only single-precision is needed.

- Cast to single-precision if accuracy and overflow conditions are satisfied

# Automatic Promotions Example

## Automatic

```
float res =  
    (17.0f*sqrt(val)) / 10.0f;  
// load val  
// convert to DP  
// _sqrt() on DP returns DP  
// load DP of 17.0  
// DP multiply  
// load DP of 10.0  
// DP divide  
// convert DP to SP  
// save result
```

two DP loads, DP multiply, DP divide,  
DP to SP conversion

## Casting to avoid promotion

```
float res =  
    (17.0f*(float)(sqrt(val)))/10.0f;  
// load val  
// convert to DP  
// _sqrt() on DP returns DP  
// convert result to SP  
// load SP of 17.0  
// SP multiply  
// load SP of 10.0  
// SP divide  
// save result
```

two SP loads, SP multiply, SP divide

# Rewriting and Rearranging Expressions

---

Divides take much longer to execute than multiplies.

Branches taken are usually faster than those not taken.

Repeated evaluation of same expression is a waste of time.



# Examples

---

```
float res = (17.0f*(float)(sqrt(val)))/10.0f;
```

is better written as:

```
float res = 1.7f*(float)(sqrt(val));
```

```
D = A / B / C;   as: D = A / (B*C);
```

```
D = A / (B * C);   as: bc = B * C;
```

```
E = 1/(1+(B*C));   D = A / bc;
```

```
E = 1/(1+bc);
```

## Algebraic Simplifications and the Laws of Exponents

Original Expression	Optimized Expression
$a^2 - 3a + 2$ $2^*, 1-, 1+$	$(a - 1) * (a - 2)$ $1^*, 2-$
$(a - 1)*(a + 1)$ $1^*, 1-, 1+$	$a^2-1$ $1^*, 1-$
$1/(1+a/b)$ $2/, 1+$	$b/(b+a)$ $1/, 1+$
$a^m * a^n$ $2 ^, 1^*$	$a^{m+n}$ $1 ^, 1+$
$(a^m)^n$ $2 ^$	$a^{m*n}$ $1 ^, 1^*$

# Literal Definitions

---

#defines are prone to error

```
float c, r;
```

```
r = c/TWO_PI;
```

```
#define TWO_PI 2 * 3.14
```

- $r = c/2*3.14$  is wrong! Evaluates to  $r = (c/2) * 3.14$

```
#define TWO_PI (2*3.14)
```

- Avoids a divide error
- However,  $2*3.14$  is loaded as DP leading to extra operations
  - convert c to DP
  - DP divide
  - convert quotient to SP

```
#define TWO_PI (float)(2.0*3.14)
```

- Avoids extra operations

# The Standard Math Library Revisited

---

Double precision is likely expected by the standard math library.

Look for workarounds:

- `abs()`

```
float output=fabs(input);
```

could be written as:

```
if(input<0)
```

```
    output=-input;
```

```
else
```

```
    output=input;
```

# Functions: Parameters and Variables

---

Consider a function which uses a global variable and calls other functions

- Compiler needs to save the variable before the call, in case it is used by the called function, reducing performance
- By loading the global into a local variable, it may be promoted into a register by the register allocator, resulting in better performance
  - Can only do this if the called functions don't use the global

Taking the address of a local variable makes it more difficult or impossible to promote it into a register

# More about Functions

---

Prototype your functions, or else the compiler may promote all arguments to ints or doubles!

Group function calls together, since they force global variables to be written back to memory

Make local functions static, keep in same module (source file)

- Allows more aggressive function inlining
- Only functions in same module can be inlined