Lecture #9

# Economics, Code Optimization & Fixed Point Math

**18-348 Embedded System Engineering**

**Philip Koopman**

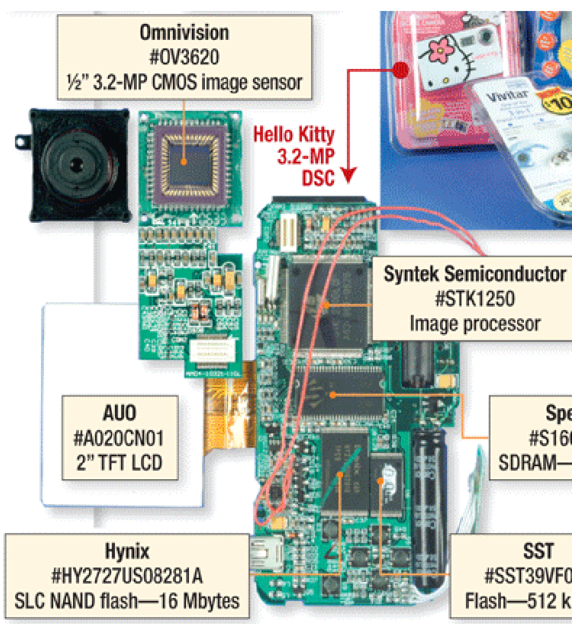
**Wednesday, 10-Feb-2016**

Electrical & Computer ENGINEERING



Carnegie Mellon

---





[Amazon.com09]

[Carey07]
Original retail $99
2 years later: sells for $45
Display $6-$7
Image sensor ~?$5
~$30 total parts cost

## Where Are We Now?

- **Where we've been:**
  - Memory bus (back to hardware for a lecture)
- **Where we're going today:**
  - Economics / General Optimization / Fixed point
- **Where we're going next:**
  - Debug & test
  - Serial ports
  - Exam #1

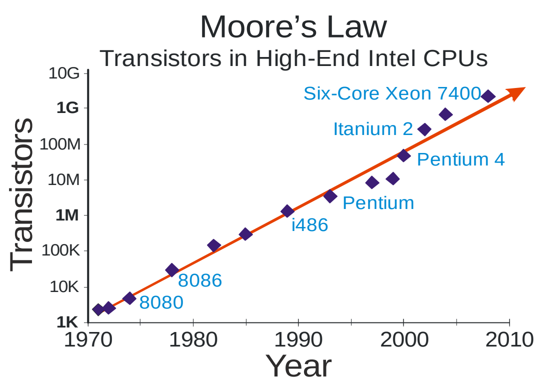


## Preview

- **Basic economics**
  - Cost vs. price
  - Recurring vs. non-recurring costs
  - How much does a line of code cost?
- **Optimizations**
  - A few very important optimization rules
  - Knowing how much and where to optimize
  - High level language optimization techniques (helping the compiler out)
    - Some 15-213 material, but we've found it doesn't stick for all students
    - Some new material
- **Fixed point math**
  - When integers aren't enough, but you can't afford floating point
  - (Yes, floating point is cheap these days, but not $.10 cheap)

## Why Aren't Embedded CPUs all 32 bits?

- **The Intel 386 was 32 bits in 1985, with 275,000 transistors**
  - Now we can build billions of transistors on a single chip!
- **First answer: fast chips are optimized for big programs, not embedded**
  - 4 MB on-chip cache for an Itanium takes 24 million transistors (assume 6T cell)
  - Many, many transistors used for superscalar + out of order execution
  - And, you can't keep it cool in many embedded systems





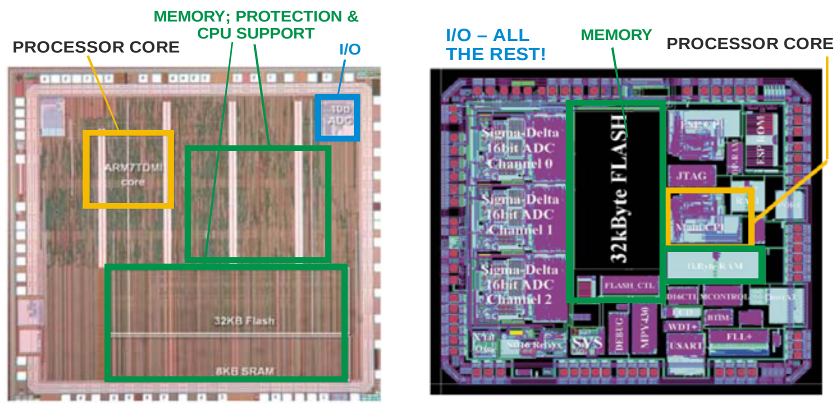

## Embedded Chips Have To Be Small(er)

- **Most embedded systems need a $1 to $10 CPU**
  - Can you afford a $500 CPU in a toaster oven?
- **This means die size is smaller than a huge CPU**
  - Smaller die takes less wafer space, meaning more raw chips per wafer
  - *And* smaller die gets better yield, meaning more good chips per wafer
  - Let's say a big CPU has 100 million transistors for $1000
    - At an arm-waving approximation perhaps you can get 2 million transistors for $10
    - This could fit an Intel 386 and 256 KB of on-chip memory, BUT no I/O
- **Embedded systems have to minimize total size and cost**
  - So real embedded systems combine CPU, memory, and I/O
  - Common to have 8K to 64K of flash memory on-chip
  - (Don't really need more than an 8-bit processor if you only have 64KB of memory and are operating on 8-bit analog inputs!)

## How Embedded Microcontrollers Spend Transistors

- **32-bit & 64-bit processors: optimize for speed – often \$5 - \$100**
- **8- & 16-bit processors: optimize for I/O integration**
  - Small memory, no operating system – often \$0.50 - \$10
- **Low-end CPUs spend chip area to lower total system cost**

32-bit ARM CPU — 8/16-bit TI CPU

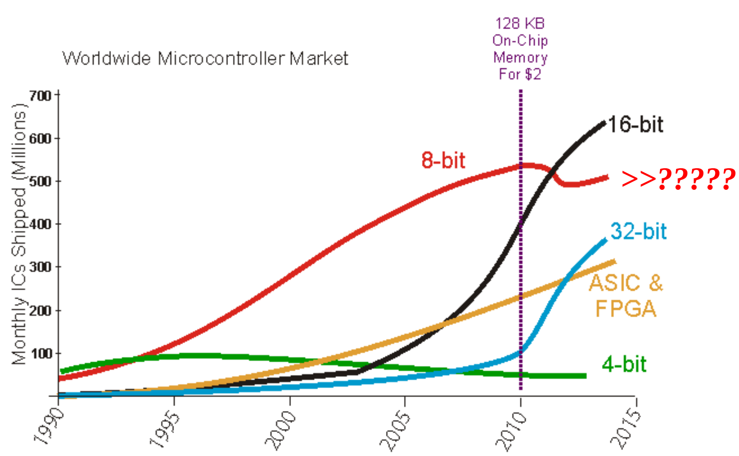



[Cravotta05]

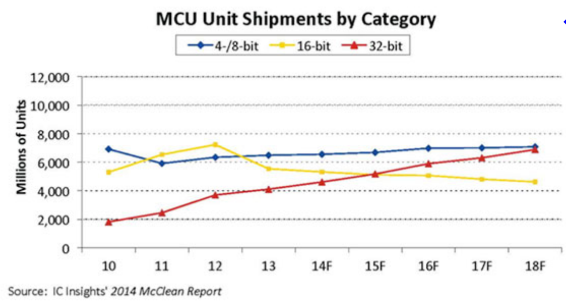


## CPU Size Trends – A prediction from 10 years ago

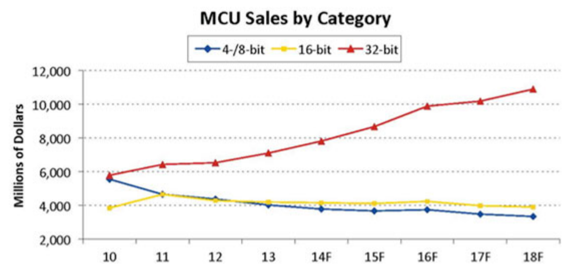
- **Most of the market (by # units) is low cost; so small CPUs dominate**
- **16-bit crossover started when:**
  - 128 KB+ of flash is small enough to leave room for I/O
  - Cost of chip is about \$2
  - Example: Nov 2009: NXP 32-bit ARM chip with good I/O; only 32KB flash; \$1
- **16-bit CPU life has been extended by compilers that do large memories**



# Market Data From 2014




- But, many of these 32-bit chips also have multiple on-chip 8/16-bit CPUs as helpers (e.g., smart peripherals)




http://eecatalog.com/8bit/2014/01/31/microcontroller automotive-and-internet-of-things/#



Charles Murray, Design News, Sept 2015

# Why 8-Bit MCUs Refuse to Go Away

New peripherals are paving the way for the continued success of the 8-bit microcontroller.

| TABLE 1: Microcontroller Market, By Revenue | | |
|---|---|---|
| 1. | 8-bit | 39.7% |
| 2. | 32-bit | 38.5% |
| 3. | 16-bit | 21.8% |
| (Source: Gartner, Inc.) | | |

The answer is that 8-bit's success has little to do with it being, well, 8 bits. For many engineers, it's more about familiarity. That's why white goods makers employ it in refrigerators, freezers, washing machines, dryers and dishwashers. It's why automakers use it in window lifts, door locks, mirrors, seat motors, and interior lighting. It's why tens of millions of smart cards incorporate it, and why countless low-end motors are controlled by it.

"It's about legacy," Jim McGregor, founder and principal analyst for Tirias Research, told *Design News*. "The engineers have been using it so long that they just don't want to switch their software base."



http://www.designnews.com/author.asp?doc_id=278431

## Bill of Materials (BOM)

- **BOM is a list of all components in system**
  - "17 pieces 1K Ohm 5% ¼ watt resistor"
  - "3 pieces 74LS374"
  - One circuit board
  - Power supply
  - ….
  - Software image rev 8.71.3
  - …

- **What's the cost of this system?**
  - BOM component costs
  - Cost of assembly, manufacture, test
  - Cost for engineering and software

  - There are inherent differences – some are per unit and some are per project



## Software Costs

- **"Firmware is the most expensive thing in the universe"**
  **– Jack Ganssle**
  - $/per pound; but amortized over 1 million units it might be nearly free

- **Typical embedded firmware costs $20 - $50 per line of code**
  - Defense work with documentation is $100/line
  - Space shuttle code perhaps $1000/line
  - 10,000 lines of code is $150K - $1M for embedded or defense work!
  - Includes all the engineering process, not just hacking "student-quality" demos

- **Lines of code often cost the same, independent of language**
  - One line of C cost = one line of assembly code cost…
    ***BUT***, one line of C does about 4x to 5x as much…
    ***SO***, assembly programs are about 4x -5x (or more) times expensive

  - Optimized code is more expensive than unoptimized code
    - It is trickier to write
    - It has more bugs and requires more maintenance

## Recurring & Non-Recurring Costs

- **Recurring Expenses (RE)**
  **– directly related to each unit produced**
  - Raw materials
  - Manufacturing labor
  - Shipping
- **Non-Recurring Expenses (NRE)**
  **– "one-time" costs to produce the first unit**
  - Engineering time
  - Semiconductor masks
  - Capital equipment (assuming equipment bought up-front)
  - Software



## Cost of Goods

- **Cost of goods general calculation:**
  - Assumes amortization of NRE over number of items produced

$$\boxed{CostPerItem = RE + \left(\frac{NRE}{\#\,Items}\right)}$$

- **Example:**
  - # Items: 100,000 units
  - NRE: 5000 lines of source code @ $25/line = $125K + $50K other = $175K
  - RE: $1 CPU + $2 other electronics + $1 housing + $1 other costs = $5
  - COST PER ITEM = $5 + ($175,000/100,000) = $5 + $1.75 = $6.75
    $6.75 * 2x wholesale markup * 2x retail markup => price = $27 retail
  - Note that software cost can't (shouldn't be) ignored!

## Cost vs. Price

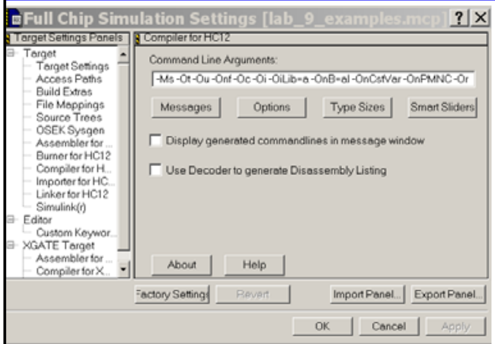
- **Goods are sold with a "mark up" from cost, yielding a "margin"**
  - "Mark up" is amount you add to cost to get price
  - "Margin" is fraction of price that is the mark up
  - Let's say BOM hardware is $10 and labor is $5; total = $15
  - If you mark up $12, price is $15+12 = $27
  - Margin is $12/$27 = 44.4% (i.e., 44.4% of price is mark up)
- **Is that all profit?**
  - Not at all … you still have to pay for:
    - Engineering and research
    - Cost of sales (sales commissions, marketing)
    - Shipping
    - Warranty returns
    - Overhead (offices, lights, the CEO's salary, …..)
  - Computation of margin varies depending on assumptions
    - What's included or excluded from the cost
  - Retailers often buy goods at 50% discount from retail
    - $10 cost with 50% wholesale margin => $20 wholesale => $40 retail(!)
    - How much can you pay for a CPU in a $25 product?

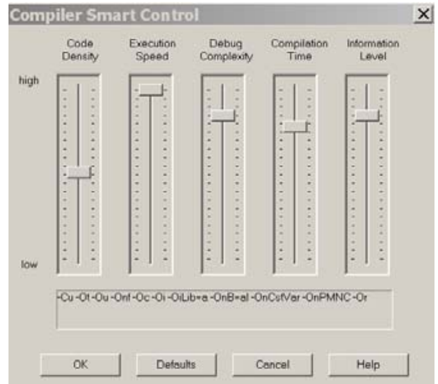


## Optimization – Getting Better Code

- **"To define it rudely but not inaptly, engineering . . . Is the art of doing that well with one dollar, which any bungler can do with two after a fashion"**
  - Arthur Mellen Wellington, 1847-1895, U.S. engineer, *The Economic Theory of the Location of Railways* (6th ed., 1900) [asme.org]
- **Optimize for:**
  - Speed – fewer clocks
  - Space – fewer bytes
  - Cost – less effort to write (e.g., automatic code generators)
  - Least likely to have defects (e.g., simple, traceable, and defensive code)
- **Step one:**
  - Ask the compiler to optimize for you (use the –O flags)

# Optimization Rule #1 – Turn On The Optimizer!






# Optimization Rule #2: Optimize What Matters

- **Speed**
  - Find the routines that take all the time, and optimize those first
  - Find sequences of operations used everywhere that are slow, optimize them
- **Size**
  - Find the biggest routines and work on them
  - Find bulky code structures that are used in many places, and improve them
- **Cost**
  - Find tools that will generate most of the code for you
  - Find “bug farms” (lots of defects) and improve those first

# Amdahl's Law

$$SPEEDUP = \frac{1}{(1 - FRACTION_{ENHANCED}) + \left(\frac{FRACTION_{ENHANCED}}{SPEEDUP_{ENHANCED}}\right)}$$

- **Originally applied to parallel computation, but applies elsewhere**
  - What if you speed up half the computation by a factor of 10?

$$SPEEDUP = \frac{1}{(1 - 0.5) + \left(\frac{0.5}{10}\right)} = 1.82 \text{ times faster}$$

- **Insight: zero execution time on loop doesn't help with rest of program!**
  - Optimizing a loop that is 10% of program, at most, improves total time by 10%

- **Optimization Corollary (rule 2.5): Make the common case fast**
  - But after a while it won't be so common (in terms of time consumed)...
  - … so optimizing is a game of diminishing returns with effort



# How Much Do You Optimize?

- **Usually it makes no sense for everything to be optimized**
  - Don't write code that is seldom executed in assembly language!

- **General procedure ("Pareto approach" – start with biggest payoff)**
  1. Measure system to find part that matters the most (speed, size)
  2. Optimize that part only (e.g., rewrite C code; move to assembly language)
  3. If good enough, stop; else go to step 1

  - Note: this approach isn't necessarily optimal, but it is usually good enough

- **Rest of lecture will concentrate on speed**
  - That's the usual, and more difficult, optimization goal

## How Do You Know What Matters?

- **Basic idea – profiling tool**
  - Measure program execution (simulated or otherwise)
  - Find the "hot spots" where program spends all its time
  - Create a "profile" (bar chart of time spent in each loop, routine, etc.)
  - Work on the highest bar of the profile chart first
  - Example – *gprof* for Unix systems

- **General approaches**
  - Simulation
    - Have simulator record each instruction executed
  - Instrumentation
    - Automatically add code everywhere to record execution
  - Statistical:
    - Periodically interrupt execution
    - Record where Program Counter happened to be
    - Repeat until enough samples are taken to be representative



## How Small A Profiling Bin?

- **Depends on situation**
  - Per routine – usually easy
  - Per loop – often loops are where time is spent
  - Per basic block (code with no branch in; no branch out) – usually good
  - Per instruction – usually overkill

- **Do it yourself profiling is sometimes required on small systems**

```
… do some stuff …
if (x > 17)
{  pcount[29]++;
   … do the if part …
} else
{  pcount[30]++;
   … do the else part …
}
// pcount track # of executions (usually "long long int")
```

## An Auxiliary Profiling Method – The NOP Trick

- **You think you know the hot spot – but you want to be sure**
  - You could optimize the code and see how much faster it gets
  - Alternative – add nops and see how much slower it gets overall

  - Saving one clock cycle is about the same time as adding a wasted cycle
    - If you add a nop and can't see a speed difference, saving a clock cycle similarly won't matter

```
     LDAA #$FF
Start_loop: … do stuff …
     NOP       ; time with a couple no-ops
     NOP       ; see how much slower it goes
     DBNE A,Start_loop
```



## Now You Know The Hot Spots – What Next?

- **Optimization RULE NUMBER 3:**
  **A better algorithm (almost) always beats tighter code**

- **Example: searching in a 1024-page dictionary**
  - Sequential search – on average 512 pages — $O(N)$
  - Binary subdivision search – 10 pages — $O(\log_2 N)$

- **Example: sorting one thousand 8-bit integer values**
  - "Bubble Sort" – 1000 elements takes ~1,000,000 operations — $O(N^2)$
  - "Quick Sort" – 1000 elements takes ~ 10,000 operations — $O(N \log_2 N)$
  - "Radix Sort" – 1000 elements takes ~ 1000 operations — $O(N)$

- **Want to know more?**
  - Take an algorithms course – a good investment for writing faster code

## High Level Code Optimization

- **If possible, optimize your C code – don't write assembly code**
  - **Optimization Rule 4: Write the least assembly language possible**
  - Assembly code is 400% – 500%+ as expensive – and not portable
  - Optimized C code will run (perhaps slowly) on another processor

- **In fantasy land … all compilers optimize everything perfectly**
  - but we don't live in a fantasy land!

- **Every compiler has optimization strengths and weaknesses**
  - To write fast code, find out what your compiler "likes" to compile
  - For other things, you get to play "human optimizer"
  - Example: our class compiler likes pointers and doesn't like subscripts (this is very common for embedded compilers)

- **To learn more about these tricks take a course on compilers**
  - Concentrating on optimizations and "back-ends" more than formal languages
  - This is in part a review of some 15-213 content



## Common Subexpression Elimination

- **Find a common partial result and save instead of duplicating:**

```
          a = (b*c*d) + (b*c*e);
⇒         a = (b*c)*(d + e);
```

  - watch out for numeric overflow etc… but usually works OK

- **Also works on memory addressing and other places**

```
          a = x[i+j+1];  b = y[i+j+1];
⇒         temp = i+j+1;
          a = x[temp];  b = y[temp];
```

- **Many compilers do some of this automatically**
  - But sometimes they need help
  - CW does OK at this

# Common Subexpression Example

**From CW compiler:**

```
  21:   for (i = 0; i < MAX-10; i++)
 0004 6981          [2]      CLR    1,SP
  22:      { for (j = 0; j < MAX-10; j++)
 0006 6980          [2]      CLR    0,SP
  23:        { a = v[i+j+3];
 0008 e681          [3]      LDAB   1,SP
 000a eb80          [3]      ADDB   0,SP  ; Breg = i+j
 000c ce0000        [2]      LDX    #v:3
 000f a6e5          [3]      LDAA   B,X ; v[i+j+3]
 0011 6a83          [2]      STAA   3,SP
  24:          b = w[i+j+7];
 0013 ce0000        [2]      LDX    #w:7
 0016 a6e5          [3]      LDAA   B,X ; w[i+j+7]
 0018 6a82          [2]      STAA   2,SP
```

*Why are there zeros for LDX values? – linker changes them later*



# Subroutine Inlining

- **Substitute a small piece of code in-line**
  - a = average (b,c);

  …

  inline uint8 average (uint8 a, uint8 b) {  return((a+b)/2); }

```
38:     result = usaverage(a,b);
 … main code …
  0034 a684          [3]      LDAA   4,SP   ; get a
  0036 ab80          [3]      ADDA   0,SP   ; get b
  0038 6a83          [2]      STAA   3,SP   ; store result
  003a 6483          [3]      LSR    3,SP   ; result >>= 1
 … main code …
```

- (Note that the compiler also knows the >>1 trick for unsigned numbers)

## Strength Reduction

- **From previous lecture – use simple operation instead of complex one**
  - `A = A * 3;` ➔ `A = A + (A<<1);`
  - `A = A / 2;` ➔ `A = A >> 1;    // only for unsigned`

- **What does the CW compiler do with signed integer division by two?**

```
 44:     r2 = (m + n) / 2;

  004e b764          [1]      TFR    Y,D
  0050 8480          [1]      ANDA   #128
  0052 2605          [3/1]    BNE    *+7 ;abs = 0059
  0054 b764          [1]      TFR    Y,D
  0056 49            [1]      LSRD      ; shift if pos
  0057 2009          [3]      BRA    *+11 ;abs = 0062
  0059 ce0002        [2]      LDX    #2
  005c b764          [1]      TFR    Y,D
  005e 1815          [12]     IDIVS     ; divide if neg
  0060 b751          [1]      TFR    X,B
  0062 6b85          [2]      STAB   5,SP
```



## Can We Help Division By Two In C?

```
inline int8 mydiv2(int8 a)
{ if (a & 0x80) { a++; }     // or could use a<0
  return(a>>1);
}
```

- Note: “>>” is undefined in C standard for neg numbers; check your compiler

- **The CW compiler doesn’t know the whole “divide by 2” trick**
  - Avoids 12-clock signed division for negative number – better is:

```
 66:     r2 = mydiv2(m);
  00a6 a682          [3]      LDAA   2,SP    ; load m
  00a8 6a83          [2]      STAA   3,SP
  00aa 8480          [1]      ANDA   #128    ; test hi bit
  00ac 2702          [3/1]    BEQ    *+4 ;abs = 00b0
  00ae 6283          [3]      INC    3,SP    ; inc if neg
  00b0 a683          [3]      LDAA   3,SP
  00b2 47            [1]      ASRA           ; shift right
  00b3 6a80          [2]      STAA   0,SP
```

## Loop Unrolling

- **Do multiple iterations of loop as in-line code**
  - To reduce per-loop overhead (e.g., do two iterations at once; halves overhead)
  - To eliminate loop overhead for a small constant number of loops
  - CW does this one

```
 71:     for (i = 1; i < 3; i++)
 72:     { v[a+b+i] = w[a+b+i];
00ba 1806           [2]      ABA    ;  compute a+b
00bc ce0000         [2]      LDX    #w:1  ; i=1
00bf e6e4           [3]      LDAB   A,X
00c1 ce0000         [2]      LDX    #v:1
00c4 6be4           [2]      STAB   A,X
00c6 ce0000         [2]      LDX    #w:2  ; i=2
00c9 e6e4           [3]      LDAB   A,X
00cb ce0000         [2]      LDX    #v:2
00ce 6be4           [2]      STAB   A,X
 73:     }   ;  no loop overhead at all!
```



## Code Hoisting

- **Sometimes there is a computation in a loop that is redundant**
  - Move it (“hoist it”) to before start of loop
  - Think of it as common subexpression elimination to outside of loop
  - CW compiler misses this one: (33 clocks per loop)

```
77:     { v[a+b+c] += w[a+b+c];          // why recompute
  00dd e682           [3]      LDAB   2,SP ; a+b+c for each loop
  00df eb83           [3]      ADDB   3,SP
  00e1 eb8d           [3]      ADDB   13,SP
  00e3 ce0000         [2]      LDX    #v
  00e6 a6e5           [3]      LDAA   B,X
  00e8 cd0000         [2]      LDY    #w
  00eb abed           [3]      ADDA   B,Y
  00ed 6ae5           [2]      STAA   B,X
  00ef 6284           [3]      INC    4,SP
  00f1 e684           [3]      LDAB   4,SP
  00f3 e182           [3]      CMPB   2,SP
  00f5 25e6           [3/1]    BCS    *-24 ;abs = 00dd
   78:     }
```

## Code Hoisting Example

- **Rewrite as:** **d = a + b + c;**
  **for (i = 1; i < a; i++)**
  **{ v[d] += w[d]; }**

  **(25 clocks per loop)**

```
81:     d = a + b + c;
  ; compute d outside loop
  00f6 e682   [3]     LDAB  2,SP
  00f8 eb83   [3]     ADDB  3,SP
  00fa eb87   [3]     ADDB  7,SP
  00fc 6b88   [2]     STAB  8,SP

  ; loop initialization
   82:    for (i = 1; i < a; i++)
  00fe c601   [1]     LDAB  #1
  0100 6b84   [2]     STAB  4,SP
  0102 2010   [3]     BRA   *+18
   ;abs = 0114
```

```
; main loop body
83:     { v[d] += w[d];
  0104 e688   [3]     LDAB  8,SP
  0106 ce0000 [2]     LDX   #v
  0109 a6e5   [3]     LDAA  B,X
  010b cd0000 [2]     LDY   #w
  010e abed   [3]     ADDA  B,Y
  0110 6ae5   [2]     STAA  B,X
  0112 6284   [3]     INC   4,SP
  0114 e684   [3]     LDAB  4,SP
  0116 e182   [3]     CMPB  2,SP
  0118 25ea   [3/1]   BCS   *-20
   ;abs = 0104
   84:     }
```



## Use Pointers Instead Of Arrays

- **C compilers sometimes favor pointers instead of arrays**
  - Maps more cleanly into index registers
  - Lots of legacy code already uses pointers, so compilers concentrate on that
- **Sometimes the CW compiler switches to pointers**
  - But usually only for simple loops over static arrays
  - Usually, using pointers generates faster code

```
int8 x[100];                 int8 x[100];
int8 a;                      int8 a;  int8 *p;
a = x[17];          ➔         p = &x[17];
                              a = *p;
```

- **Lab involves changing a loop from indices to pointers.**

# Loop Optimization

- **Some MCUs have special instructions and addressing modes**

- **For example, count-down loops**
  - "for (i = 100; i >0; i--)"
    - Might compile into a decrement and test for zero assembly instruction
    - DBNE instruction does this, right?
  - Thus, it is often faster than: "for (i = 1; i <=100; i++)"
    - Requires increment and compare



# Use Minimal Data Types

- **Don't use a 16-bit int when an 8-bit int will do!**
  - This assumes the CPU "likes" 8 bit data values, which is true of our CPU
  - Memory size aside, often get best speed by matching data sizes to hardware word size

- **… we've already discussed data types, but don't forget to do this! …**
  - int8 uint8
  - int16 uint16

## A Word About Compiler Bugs(!)

- **Many compilers have bugs …
  and many of those bugs show up in infrequently used features …
  such as:**
  - Extended precision arithmetic (e.g., long long shifting on some workstations)
    - Or anything that is used infrequently in production code
  - Very high optimization levels (e.g., "-O4" optimization)
  - That having been said, the CW tools are *remarkably* clean
- **If you have strange problems with your software …**
  - … try reducing optimizations and see if problems go away
  - Alternately, check the compiled output and see if it is correct



## Optimization Via Special Hardware

- **DSP – Digital Signal Processor chip**
  - Has hardware multiplier & hardware multi-bit shift (barrel shifter)
    - (These might be the same array of AND gates used two ways)
  - Often has hardware support for FFT butterfly operand access
  - Used for signal processing
  - Traditionally integer, but newer ones have floating point
- **FPGA – Field Programmable Gate Array**
  - Can program chip to have any hardware you like (Verilog => HW synthesis)
  - Can implement a CPU in a large FPGA plus other logic
  - Can have a fixed CPU (smaller die area) with FPGA around it
  - Much more expensive per gate than ASIC or ASSP
- **ASIC – Application Specific IC = your own custom chip**
- **ASSP – Application Specific Standard Product**
  - Someone else's idea of a chip tailored to your application area
  - Standard product, but with hardware support (e.g., CRC hardware; Fuzzy logic support)

# Fixed Point Math

- **Floating point math is very expensive!**
  - Usually no hardware for floating point on small microcontrollers
  - Software support is big (lots of code space) and slow (lots of clock cycles)
- **General approach to reduce cost: use fixed point math**
  - Use an integer with some digits of a fraction already put in
    - E.g., for 16-bit machine value can interpret as 8 bits integer and 8 bits fraction


  - Or, change units fractional units “1/10 of one degree” for temperature
    - $807_{10}$ = 80.7 degrees, etc
    - But, usually math is more efficient if you use binary radix
      can use shift instead of divide to align results of * and /
- **Addition and subtraction easy – just use integer add subtract**
- **Division and multiplication difficult – need to do “scaling” to line up decimal**

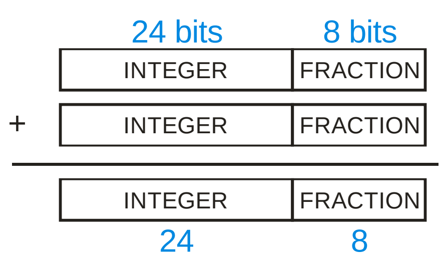


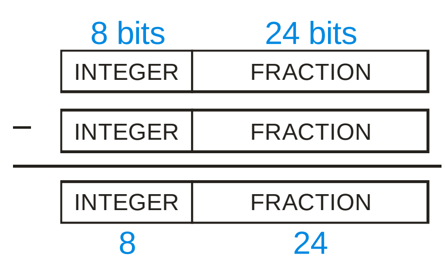
# Fixed Point Add and Subtract

- **Implementation: no different than multi-precision add/subtract**
  - Radix point stays in same position in result as in operands
  - Two’s complement still works as it does for integers

```
  244.6
+ 125.3
---------
  369.9
```

```
  1.A6B
- 3.2FC
---------
 -1.891
---------
  E.76F
```



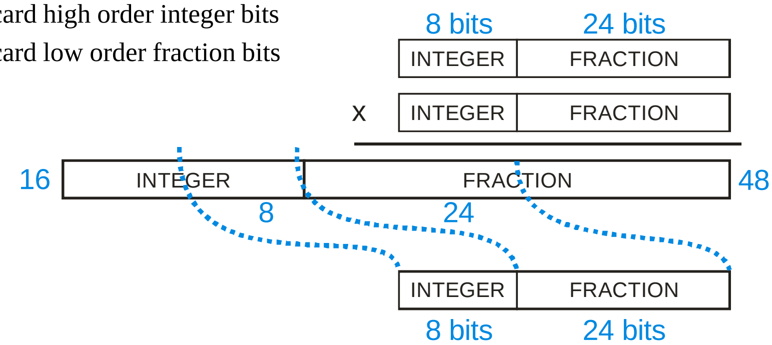



# Fixed Point Multiply

- **Basic multiplication is same as for integers**
  - Radix point shifts to the left
  - Same number of total bits to right and left as <u>sum</u> of bits in operands
  - E.g.: 8.24 x 8.24 => 16.48 bits
- **Result alignment option #1:**
  - Re-align radix point
  - Discard high order integer bits
  - Discard low order fraction bits

```
  2.4A6
x 1.C53
---------
04.0E09D2

4.0E0
```



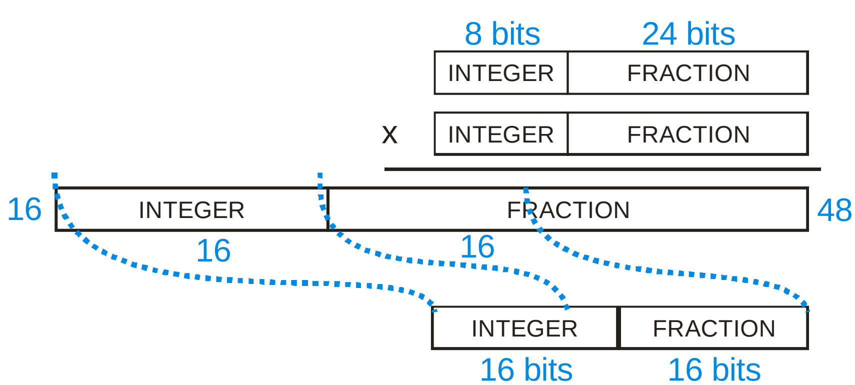



# Fixed Point Multiply – 2

- **Result alignment option #2:**
  - Keep integer bits and as many fraction bits as will fit
  - Discard all low order bits
  - Whether you do this depends on how many significant integer bits you predict you will have

```
  2.4A6
x 1.C53
---------
04.0E09D2

04.0E
```

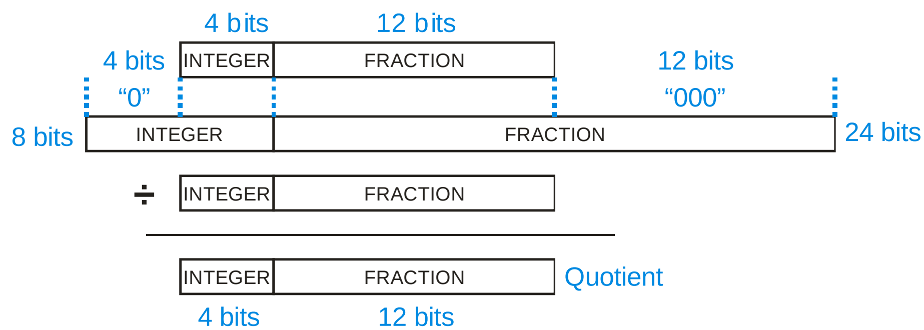


# Fixed Point Divide

- **Create Dividend with twice as many bits before & after radix point**
  - Then, execute normal integer division
  - Quotient will have correct format
  - Think of formatting as the reverse of multiply

```
  7.4A6  →  07.4A6000
÷ 1.5BA       1.5BA
            ---------
              5.5E7
```

  - Non-negative example below:






# Keeping Track of the Radix Point

- **Main practical differences between fixed & floating point:**
  - Fixed point is faster in absence of floating point hardware
    - Bit-by-bit alignment is expensive in hardware (requires a barrel shifter)
  - More digits of precision (don't "waste" bits on exponent)
  - Programmer has to manually keep track of the radix point and align as needed
    - Arguments to fixed point math need not have homogeneous radix point formats

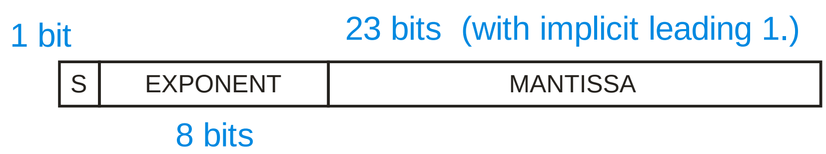
```
  244.6             24A.6            24.A6
+   1.446         x 1.B4C          x .1B4C
-----------    ---------------   --------------------
  245.A          03E8.6348        0 3 E 8 6 3 4 8

                                       .
                                      ???
```

# How Is Floating Point Different?

- **Uses scientific notation (exponent plus mantissa)**
- **Single precision is:**
  - 1 bit sign (applies to sign of number, not sign of exponent)
  - 8 bit exponent (range -126 to +127); ~ $10^{37}$
  - 24 bit mantissa, aligned with "1" in first bit, which is implicit; ~ 7 decimal digits
  - A number of special bit patterns, e.g.:
    - NaN = "not a number" – result of numerical error propagated to outputs
    - Infinity
- **Double precision is 64 bits – bigger exponent; bigger mantissa**

IEEE Floating Point Format
Single Precision: 32 bits total






# Floating Point Pitfalls #1 & #2 – Comparisons

- **Besides being slow/expensive, there are times when floating point can burn you!**

- **Problem #1: comparisons might not be meaningful**
  - What is wrong with this code fragment?

```
if (MyFloatA == MyFloatB) . . .
```

- **Problem #2: sometimes comparisons fail**
  - Consider, for example, a speed limit on a system
    - Simple control loop: if speed is too fast, reduce commanded speed by 10%
    - (For example, perhaps you are going down a hill and picking up speed from gravity)

  - When will this code NOT work as expected?

```
#define SPEEDLIMIT 3.0
double SpeedCommand, SpeedActual;
. . .
if (SpeedActual > SPEEDLIMIT) {SpeedCommand *= 0.9;}
```

## Floating Point Pitfall #3 – Roundoff

- **What output does this program produce?**

```
#include <stdio.h>

int main(void)
{
  union  { float fv;  int iv; } count;
  long i;

  for (i = 0; i < 0x00FFFFF8; i++)
  { count.fv += 1;
  }

  for (i = 0; i < 16; i++)
  { count.fv += 1;
    printf(" + 1 = %8.0f   0x%08X\n", count.fv, count.iv);
  }
 return;
}
```



## Floating Point Roundoff Error

- **If you increment floating point, at some point it stops incrementing(!)**
  - This happens a lot sooner than you might think
  - Effective size of mantissa is only 24 bits = 16777216
  - <u>**Always use an int or long for time!**</u>

```
$ ./float
 + 1 = 16777209   0x4B7FFFF9
 + 1 = 16777210   0x4B7FFFFA
 + 1 = 16777211   0x4B7FFFFB
 + 1 = 16777212   0x4B7FFFFC
 + 1 = 16777213   0x4B7FFFFD
 + 1 = 16777214   0x4B7FFFFE
 + 1 = 16777215   0x4B7FFFFF
 + 1 = 16777216   0x4B800000
 + 1 = 16777216   0x4B800000
 + 1 = 16777216   0x4B800000
 + 1 = 16777216   0x4B800000
 + 1 = 16777216   0x4B800000
 + 1 = 16777216   0x4B800000
 + 1 = 16777216   0x4B800000
 + 1 = 16777216   0x4B800000
 + 1 = 16777216   0x4B800000
```

IEEE Floating Point Format
Single Precision: 32 bits total

1 bit — 23 bits (with implicit leading 1.)

| S | EXPONENT | MANTISSA |
|---|---|---|

8 bits

# Floating Point Pitfall #3 part II – Float32 Time

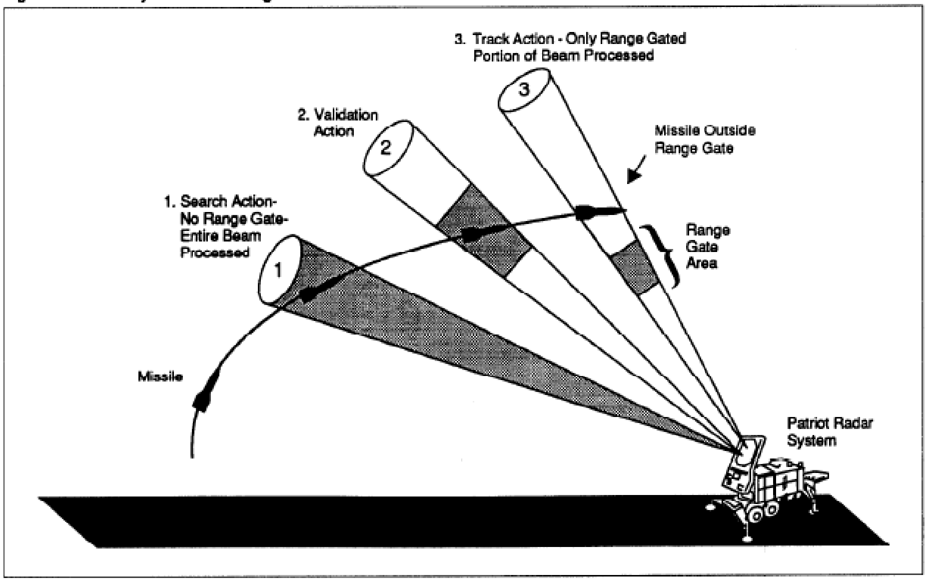
- **Say you are counting 1/100th of seconds as a time tick**
  - 32-bit count rolls over in about 16 months
  - So, let's use 32-bit floating point instead (bad idea, but why?)
- **Floating point format: 8 bit exponent 24 bit mantissa**
  - Increment number by 1/100th for every time tick
  - First problem 1/100th is an imprecise number in floating point – roundoff error
  - But, might still work OK for a while
  - As number gets bigger, roundoff error for increment gets bigger
    - Fewer of the fractional bits in 1/100 actually "count" in the additions
    - By $2^{24}$ / 100 seconds (47 hours) – the time won't increment at all!
    - With 32-bit floating point $2^{24} + 1 = 2^{24}$ (the +1 is lost in rounding error)



# Would Anyone Use Float Time?

- **Patriot Missile incident**
  - 1991: Scud kills 28 American (Desert Storm)
  - http://www.fas.org/spp/starwars/gao/im92026.htm
    "after about 20 hours, the inaccurate time calculation becomes sufficiently large to cause the radar to look in the wrong place"
    - "Range gate" used to look where target is predicted to be next
    - Target track is lost if range gate is wrong, resulting in a miss
    - The incident happened 100 hours after the last system reset
- **What was the root cause mistake?**
  - Scud missiles travel at Mach 5 (3750 mph) – Patriot designed to track aircraft
  - Time was represented in 10ths of a second as an integer
    - Then converted to 24-bit fractional value for calculation
    - 0.1 seconds is not an "even number" = 0.00011001100110011001100**11001100…**
    - At 100 hours, resultant round-off is 0.000000095 decimal
      [http://www.ima.umn.edu/~arnold/455.f96/disasters.html]
  - Even that small round-off error when doing distance = velocity * time with large base time and high velocity leads to a failure
    - After 100 hours error was 0.344 seconds = 697 meters error (per GAO report)

Figure 5: Incorrectly Calculated Range Gate




[GAO/IMTEC-92-26]



# Review

- **Basic economics**
  - Markup, margin
  - NRE vs. RE
  - How much does firmware cost per line?
- **Optimization**
  - Optimization Rules – memorize them (there are only 4 ½ of them)
    - Numbered: 1, 2, 2.5, 3, 4
  - Amdahl's law
    - Be able to apply (know the formula, but not required to write it down)
  - Profiling techniques
    - Know different profiling strategies
  - Basic optimization techniques –
    if we give you some C code, can you apply a technique we tell you to apply?
- **Fixed point**
  - Understand how to put the radix point in the right place in operands and result
  - Understand floating point pitfalls