Lecture #7

Bit Hacking, Multiprecision Math, Peer Reviews

18-348 Embedded System Engineering

Philip Koopman

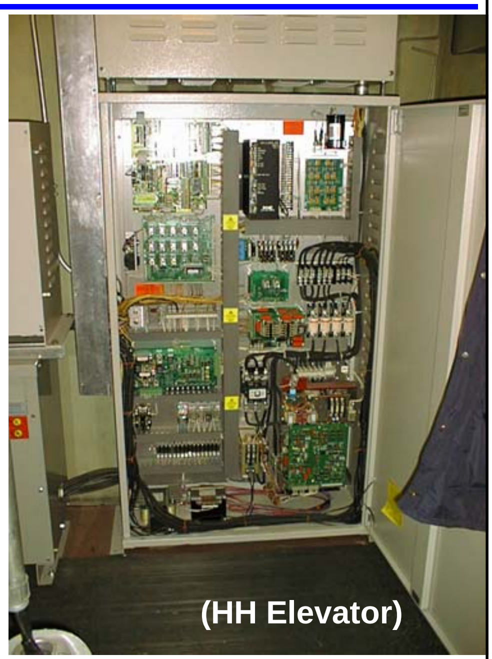
Wednesday, 3-Feb-2016

Electrical&Computer ENGINEERING

Carnegie Mellon

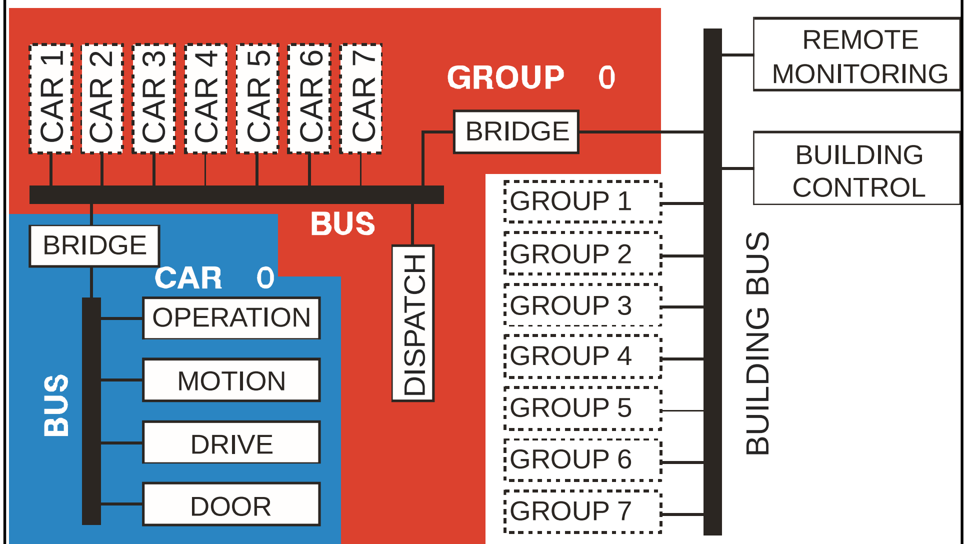
Example: Elevator Controllers

- **Motor control**
  - Firm real time
  - Safety critical
  - Mechanical interlocks
  - Fail safe
- **Door control**
  - Soft real time
  - Somewhat safety critical
  - Mechanical interlocks
  - Fail safe
- **Many other subsystems**
- **Most electronics for I/O and power (replaces relays)**



Embedded Distributed Architecture

- **Separate Control Systems for many functions**
  - (Real elevators have more than are shown here.)



Where Are We Now?

- **Where we've been:**
  - Embedded programming techniques
  - Linking C to assembly
- **Where we're going today:**
  - More Embedded programming techniques
    - Some 15-213 material, but we've found it doesn't stick for all students
  - Multi-precision math
  - Design reviews
- **Where we're going next:**
  - Memory bus
  - Economics / general optimization
  - Debug & Test
  - Serial ports

  - Exam #1

Preview

- **REMINDER: Pre-Labs are Individual Effort! (like "homework")**

- **Bit-based optimizations**
  - Bit masking
  - Division and multiplication via shifting
  - Counting bits

- **Multiple precision math – doing math bigger than CPU size**
  - Addition/subtraction
  - Multiplication
  - A little division

- **Peer Reviews**
  - Reviews are very effective at finding defects
  - Basic good review practices

Buffer Wrap-Around Trick

- **Let's say you have a buffer that is 256 bytes long**
  - Want to fill bytes 0 … 255, then wrap-around to 0
  - Slow and really obvious code:

    x[i] = z;   i = (i + 1) % 256;

    - » But, division is slow!
  - "Obvious" faster code:

    x[i++] = z;   if (i==256) {i = 0;}
  - "Tricky" code:

    x[i++] = z;   i = i & 0xFF;

- **In general, i = i & (2^N-1) for a buffer size of 2^N**
  - For example, what hex value do you AND with for a 1024-element buffer?
  - What if buffer size isn't an even power of two?

Optimization: "Strength Reduction"

- **Convert division and multiplication into shifting**
  - Can be faster than multiply if chip doesn't have a hardware multiplier
  - Usually only worth doing for small integers or numbers near power of 2
  - Works for both signed and unsigned numbers

- **Multiply by shifting:**
  - Multiply by 2^N by shifting left N bits

  - A = A * 8; => A = A << 3;
  - B = B * 512 => B = B << 9;

- **Complex multiply by selective shift & add:**
  - A = A * 9 => A = (A<<3) + A;
  - A = A * 15 => A = (A)+(A<<1)+(A<<2)+(A<<3);
  - A = A * 15 => A = (A<<4)-(A);

Division Via Shifting

- **Unsigned division is easy**
  - unsigned int n;
  - n = n / 8; => n = n >> 3;

- **But, signed division is more difficult!**
  - Integer division rounds toward zero (symmetric around zero)
  - Shifting rounds down (asymmetric at zero)

- **For signed division, answer isn't quite right!**
  - (How to work around this is a lab topic)
  - Convince yourself you have the right answer via a test program

i	i>>2	i/4
-8	-2	-2
-7	-2	-1
-6	-2	-1
-5	-2	-1
-4	-1	-1
-3	-1	0
-2	-1	0
-1	-1	0
0	0	0
1	0	0
2	0	0
3	0	0
4	1	1
5	1	1
6	1	1
7	1	1
8	2	2

The Carry Bit

- **Remember that the basic purpose of a carry bit is multi-precision math**
  - Example: 16-bit addition done with 8-bit operations:

```
LDAA     X_lo  ; add low byte   Z = X + Y
ADDA     Y_lo  ; generate carry for high byte
STAA     Z_lo

LDAA     X_hi  ; add high byte  Z = X + Y
ADCA     Y_hi  ; incorporate carry from low byte
STAA     Z_hi
```

- **Can generalize to as many bits as you want**
  - Lowest "chunk" is ADDA; all other chunks are ADCA
    OR
  - Use all ADCA and make sure to use CLC before first set of adds
  - Subtract is similar – use subtract with borrow ("borrow" is the carry bit)

Using The Carry Bit As An Error Return Flag

- **Robust code needs error handling**
  - But, how do you know an error happened?
  - Sometimes can use an "illegal" return value (e.g., null pointer)
  - But what if all values are legal?
    Use an "out of band" value … such as the carry bit

```
  …
   JSR   MyRoutine
   BCS   error_handler
  …
… … … … … … … … … … … … … … … … … … …
MyRoutine:
   …   ; do processing
   CLC          ; normal return here
   RTS
ErrorRtn:
   STC          ; set carry bit as error flag
   RTS
```

Flag Manipulation Via Carry Bit

- **Convert a "dirty" flag to a clean flag in an integer**
  - Clean flag in C is 1=True or 0=False
    - "Dirty" non-zero flag is non-zero, but could be anything
    - C compilers may waste a lot of code cleaning flags to conform to the standard
  - On some CPUs, especially with high branch penalties this is a major win; on HC12 it's not worth doing

```
    ; assume starting value is in A (5 bytes / 3 cycles)
    ; simple way with a branch
    TSTA          ; sets flags based on contents of A
    BEQ    zero_val
    LDAA   #1     ; load a clean carry bit; result in A
Zero_val:
    … - - - - - - - alternate code here- - - - - - - - - - - -
    ; tricky way with carry bit  (result in B)
    LDAB   #1     ; default value is true
    DECA           ; false value is now $FF instead of $00
    ADDA   #1     ; generates carry-out if false
    SBCB   #0     ; subtract one only if false
    ; 7 bytes total / 4 cycles, but no branch
```


Computing Parity

- **Parity of a number is xor of all the bits**
  - Parity of 8-bit value = x0 ⊕ x1 ⊕ x2 ⊕ x3 ⊕ x4 ⊕ x5 ⊕ x6 ⊕ x7

- **Brute force way (k operations for k bits):**

  x = ((x) ^ (x>>1) ^ (x>>2) ^ (x>>3) ^ (x>>4) ^ (x>>5) ^ (x>>6) ^ (x>>7)) & 1;

- **Better way (log2(k) operations for k bits):**

  x = x ^ x>>4; // parity for 8 bits
  x = x ^ x>>2;
  x = (x ^ x>>1) & 1;

  x = x ^ x>>16;// parity for 32 bits
  x = x ^ x>>8;
  x = x ^ x>>4;
  x = x ^ x>>2;
  x = (x ^ x>>1) & 1;

Parity for 8 bits;	X ➔ bits =	J K L M N P Q R
x = x ^ x>>4;	bits =	J K L M (N^J) (P^K) (Q^L) (R^M)
x = x ^ x>>2;	bits =	- - - - - - (Q^L^N^J) (R^M^P^K)
x = x ^ x>>1;	bits =	- - - - - - - (R^M^P^K^Q^L^N^J)
x = x & 1;	bits =	0 0 0 0 0 0 0 (parity)

- **How many steps (lines of code according to above style) for 128 bits?**

Counting Bits

- **Sometimes you need to know how many 1 bits are in a word**
  - Some mainframes actually had "bit count" instructions!
  - Some companies ask this question as a job interview question
- **Simple way (16-bit example)**

```
// input in integer "value"
int count = 0;
for (int i = 0 ; i < 16; i++)
   { if ( (value >> i) & 1) count++;}
```

- **Usually more efficient to shift value as well and avoid "if":**

```
int count = 0;
for (int i = 0 ; i < 16; i++)
   { count +=  value & 1;
     value = value >> 1;
   }
```


Handy Tool – Lookup Table

- **Lookup table is a precomputed set of answers stored for later use**
  - At compile time or when program starts, do the computation once
  - Store results in an array
  - Instead of computing again at run time, just look up the answer
  - More advanced: store samples from continuous data (e.g., trig functions) and interpolate
- **Example: pre-computing a lookup table for bit counting**

```
uint8 count_table[….] = {0,       // 0x00   zero bits
                         1,       // 0x01   1 bit set
                         1,       // 0x02   1 bit set
                         2,       // 0x03   2 bits set
                         1,       // 0x04
                         2,       // 0x05
                         2,       // 0x06
                         3,       // 0x07
                         1,       // 0x08
                         2,       // 0x09
                         2,       // 0x0A
```

- **Usage: bitCount = table[0x0A]; // get number of bits=2 in hex value 0x0A**

Counting Bits – Better Ways

- **In assembler, can use shifting and carry**
  - Shift bits into CY bit
  - Do ADC into sum
  - Stop not after 16 iterations, but when residual word is zero
- **Even better is to use lookup tables**
  - One 256-entry lookup table for 8 bit value (preload with counts)
  - But if memory is tight, use a 16-entry table

```
uint8 count_table[16] = {0, 1, 1, 2,    1, 2, 2, 3,    1, 2, 2, 3,    2, 3, 3, 4 };
// input in integer "value"
count = count_table[value & 0xF] + count_table[(value>>4) & 0xF];
```

  - (similarly, can use a 4-bit or 8-bit table for 16 and 32 bit data words)
  - On Pentium-III, an 8-bit table was faster than a 16-bit table because 8-bit table *fits completely into L1 cache*

Multi-Precision Math

- **What happens if you need big integers and you have a small CPU?**
  - 16-bit+ math on an 8-bit CPU
  - 32-bit+ math on a 16-bit CPU
  - 64-bit math on a 32-bit CPU
- **To do this, you need multi-precision math**
  - Most embedded engineers end up implementing multiprecision math sometime
  - Some C compilers have it, some don't
    - CW tools don't support 64 bit integers
    - Sometimes you need it for assembly language
    - For every new CPU, *someone* needs to write the math routines
  - Does this really happen – yes!
    - Cryptographic operations (e.g., mod function on 128 bits)
    - There are 31,557,600 seconds in a year – that won't fit in 16 bits
      - » Number of microseconds in a year won't fit either!

Example: 16-bit Add & Subtract For 8-bit CPU

```
X: DS.B 2
Y: DS.B 2
Z: DS.B 2
```

- (be sure to get x,y order right for subtract!)

◆ Z = X + Y

```
LDAA X+1
ADDA Y+1
STAA Z+1

LDAA X
ADCA Y
STAA Z
```

◆ Z = X - Y

```
LDAA X+1
SUBA Y+1
STAA Z+1

LDAA X
SBCA Y
STAA Z
```


32-Bit Add For 8-Bit CPU

```
X: DS.B 4    ; assume this byte order:
Y: DS.B 4    ;  (hi) 0, +1, +2, +3 (lo)
Z: DS.B 4
```

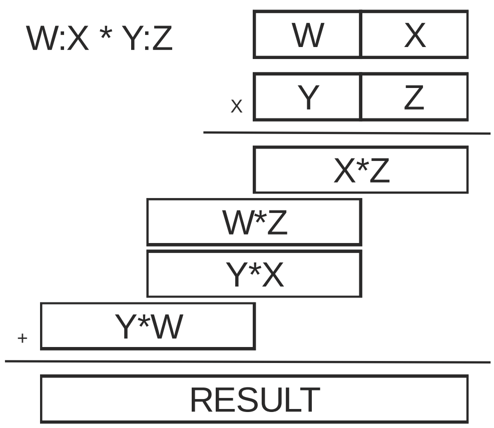
◆ Z = X + Y (subtract is similar)

```
LDAA X+3
ADDA Y+3
STAA Z+3

LDAA X+2
ADCA Y+2
STAA Z+2
```

```
LDAA X+1
ADCA Y+1
STAA Z+1

LDAA X
ADCA Y
STAA Z
```

Multiplication

- **Remember how we do unsigned decimal multiplication?**
  - Binary multiplication is the same process, but “digits” are 8 bits each



  - 16 * 16 bits => 8-bit “digits”; 16-bit partial products and 32-bit result
  - Also works for 16-bit “digits” making 64-bit result

Multiplication – Simple C Code

```
// 16x16 gives 32-bit unsigned multiply
uint16 a, b;
uint32 w, x, y, z; // 32-bits for
  compiler
uint32 result;
w=((a>>8)&0xFF);   x = (a & 0xFF);
y=((b>>8)&0xFF);   z = (b & 0xFF);
result =                        x*z;
result +=          (w*z)<<8;
result +=          (y*x)<<8;
result +=(y*w)<<16;
```

- uint32 used for w,x,y,z so compiler properly maintains 32-bit sums

Multiplication – Lookup Table Method

- **Use a lookup table with 8x8 = 16 bit products**
  - Instead of individual multiply instructions – some CPUs don't have a MULT instruction!
  - 8x8=>16 bit table is 128KB (too big for many embedded CPUs)
  - 4x4=>8 bit table is only 256 bytes

```
// 8x8 gives 16-bit unsigned multiply/tables
uint8 prod_table[256];  // init with products
#define table_mult(a,b)  prod_table[(a&0F)<<4|b&0F]

uint8 a, b;
uint16 w, x, y, z; // 16-bits for compiler
uint16 result;
w=((a>>4)&0xF);   x = (a & 0xF);
y=((b>>4)&0xF);   z = (b & 0xF);
result =                   table_mult(x,z);
result +=            table_mult(w,z)<<4;
result +=            table_mult(y,x)<<4;
result += table_mult(y,w)<<8;
```


Multiplication – Shift And Add

- **Binary multiplication instead of "decimal" or base 256 multiplication**
  - This is what is used by most low-end hardware
  - Complexity proportional to #bits (1 clock cycle per bit + overhead)
  - E.g., HC12 multiply is 12 clocks for 8x8 unsigned multiply
    - 4 clocks overhead plus one clock per bit for 8 bits

```
// code to multiply two uint8 values
uint8 a;   uint16 b;  // b starts with 8 bits
uint16 result;
uint8 i;
result = 0;  // zero shift+add accumulator
for (i=0; i<8; i++)
{ if (a & 1) result += b;
  a = a >> 1;     b = b << 1;
}
```

Better Shift And Add Multiply

- **Trick – careful analysis saves space**
  - Shift one operand out to the right in low part of variable
  - Accumulate result in high part of variable
  - Very useful if you only have one register with both operand and result
  - Cuts from two shifts per iteration to one shift (CY bit used too!)
  - ***BUT only works for positive integers (top bit 0) unless you catch carry out bit in asm!!***

```
uint32 w, x;  // but only hold 16 bit values
uint32 result;
result = x;  // high is 0; low is x
w = w << 16;  // align with high byte for adds
for (uint8 i=0; i<16; i++)
{ if (result & 1) {result = (result+w)>>1;}
  else            {result = result>>1;}
} // note: loop loses carry-out of result+w!
  // (use ROR if in assembly language)
```


RESULT = X * W

START:

W3	W2	W1	W0	0	0	0	0	*W*
0	0	0	0	X3	X2	X1	X0	*RESULT*

Test lowest bit of result (X0); assume it's 1 for this example
Add W:

cy=0	W3	W2	W1	W0	X3	X2	X1	X0	*RESULT*

Shift right:

cy=0	0	W3	W2	W1	W0	X3	X2	X1	*RESULT*

Test lowest bit of result (X1); assume it's 0 for this example
Shift right:

cy=0	0	0	W3	W2	W1	W0	X3	X2

*If rest of bits of X are all zero, when you're done you get W * 1 = W*

0	0	0	0	W3	W2	W1	W0

Example (4 x 4 => 8 bits) For Assembly Language

W * *X* = *1110* * *1101* = *10110110* = *RESULT*

		1	**1**	**1**	**0**	0	0	0	0	*W*
Start		0	0	0	0	*1*	*1*	*0*	*1*	*RESULT*
+ W	cy=0	1	1	1	0	*1*	*1*	*0*	*1*	*RESULT*
> > 1	cy=0	0	1	1	1	0	*1*	*1*	*0*	*RESULT*
> > 1	cy=0	0	0	1	1	1	0	*1*	*1*	*RESULT*
+ W	cy=1	0	0	0	1	1	0	1	1	*RESULT*
> > 1	cy=0	1	0	0	0	1	1	0	1	*RESULT*
+ W	cy=1	0	1	1	0	1	1	0	1	*RESULT*
> > 1	cy=0	1	0	1	1	0	1	1	0	*RESULT*

Signed Multiply

- **There are really tricky algorithms (usually with special hardware)**
  - Usually what you do in software is:
    1. Compute sign of result (sign A ⊕ sign B)
    2. Negate any negative inputs (2's complement) to give absolute values
    3. Multiply absolute values (works fine with tricky shift and add – top bit is zero!)
    4. Negate result if sign of result was negative
- **Other multiplication notes**
  - Don't forget that size of product is twice size of operands!
  - Signed and unsigned multiply are the same if you truncate to low order bits
    - 16-bit x 16-bit => low 16-bits of result is same whether signed or unsigned
  - In CPU hardware or microcode, 1 clock per bit
    - Can perform conditional add and shift in one clock with the right data path

Division

- **Division is also a shift-and-add operation**
  - Similar to pencil-and-paper division, but one bit at a time
  - Generally also performed on positive integers
  - Use the same registers for dividend, quotient, remainder; similar to multiply
    1. Subtract divisor (e.g., 16 bits) from dividend (e.g., 32 bits)
    2. If result is negative, add it back in; if not, record a “1” in quotient
    3. Shift everything one bit to the right (32-bit shift)
    4. Iterate to step 1 for all bits
    5. When done, get a remainder (16 bits) and a quotient (16 bits)
  - Details beyond scope of this class
    - There is also a “nonrestoring” division which is more efficient; but tricky
      - In hardware, works in one clock per bit (e.g., ~16 clocks for 32/16 => 16 bits)
    - Wikipedia division article has algorithms
      - But they look way more complicated than they need to be for assembly language
- **We expect you to know how to multiply**
  - Division is purely bonus material
  - If you have questions about doing division, see Prof. Koopman at office hours

Getting It Right Matters!

Segway recalls scooters for injury risk

AP Associated Press

AP Photo: In an undated file photo released by Segway, Inc. is the Segway i2 Personal Transporter.Segway...

By MICHAEL P. REGAN, AP Business Writer

Thu Sep 14, 2:19 PM ET 2006

NEW YORK - Segway Inc. is recalling all 23,500 of the self-balancing scooters it has shipped because of a software glitch that can make its wheels unexpectedly reverse direction, throwing off the rider and in at least one incident, break some teeth.

The U.S. Consumer Product Safety Commission on Thursday said consumers should stop using the vehicles immediately. Segway is cooperating on the voluntary recall.

Segway has received six reports of problems with the Personal Transporter, resulting in head and wrist injuries.

Segway is offering its customers, which include more than 150 police departments around the world, a free software upgrade that will fix the problem. The upgrades will be done at Segway's 100 dealerships and service centers around the world, according to Segway spokeswoman Carla Vallone. The Bedford, N.H., company will pay to ship the devices to the appropriate center if need be.

It is the second time the scooters, which sell for about $4,000 to $5,500, have been recalled since they first went on sale in 2002. The 2003 recall involved the first 6,000 of the devices sold, and involved a problem that could cause riders to fall off the device when was the battery depleted.

Segway Chief Technology Officer Doug Field, who has been involved with the development of the device since its earliest days, said the problem that sparked the latest recall was found while the company was testing its new model. He said a very unusual and specific set of conditions can cause the problem.

http://news.yahoo.com/s/ap/20060914/ap_on_bi_ge/segway_recall

How Common Are Coding Defects?

- **2012 Coverity scan of open source software results:**
  - Control flow issues: 3,464 errors
  - Null pointer dereferences: 2,724
  - Resource leaks: 2,544
  - Integer handling issues: 2,512
  - Memory – corruptions : 2,264
  - Memory – illegal accesses: 1,693
  - Error handling issues: 1,432
  - Uninitialized variables: 1,374
  - Unintialized members: 918

http://www.embedded.com/electronics-blogs/break-points/4415338/Coverity-Scan-2012?cid=Newsletter+-+Whats+New+on+Embedded.com

REVIEWS

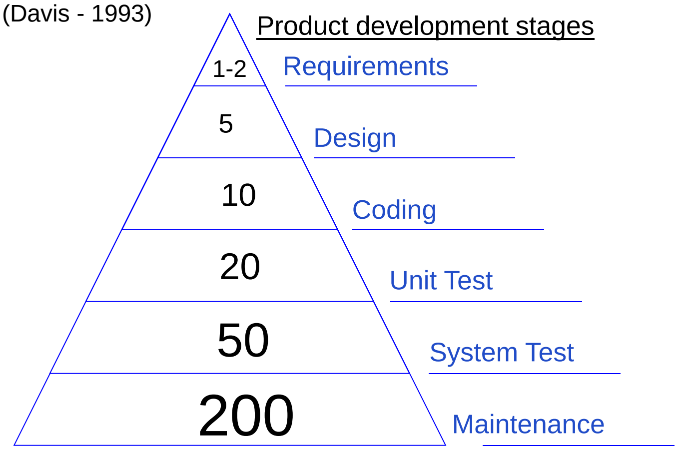
- **Design reviews are the most cost effective way to eliminate defects**
  - Review early and often
  - Review everything
  - Reviews are ***MORE EFFICIENT*** than testing for catching bugs!
- **In the context of the course:**
  - When your lab partner does something, double-check it *thoroughly!*
  - When you do something, ask your lab partner to check it!
  - In class, you're also reviewing my slides (and finding occasional slips)
  - The TAs and I review labs before release

Early Detection and Removal of Defects -

Peer Reviews - remove defects early and efficiently

Relative Cost to Fix Requirements Errors

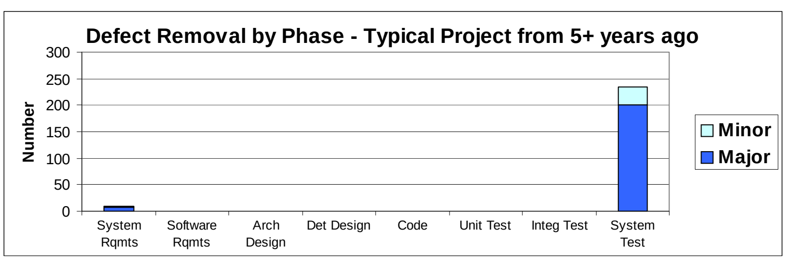
(Davis - 1993)

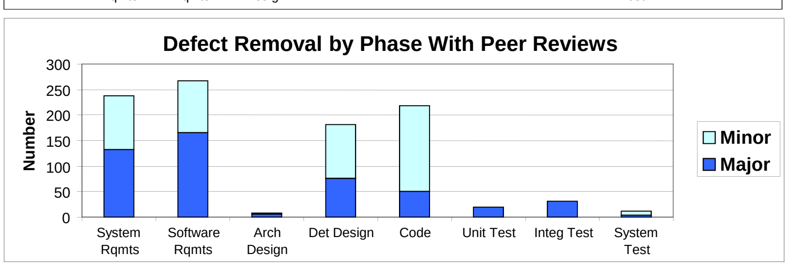


Defect Management – Then vs. Now

(Real data from embedded industry)

Defects are removed earlier





(Source: Roger G., Aug. 2005)

Good Review Practices

- **Check your egos at the door**
  - Nobody writes perfect code. Get over it.
- **Critique the Products**
  - Don't attack the author. Your turn in the hot seat will come soon enough!
- **Find problems**
  - Don't try to fix them; just identify them.
- **Limit meetings to two hours**
  - 150-200 lines per hour; two hours max
- **Avoid style "religious" debates**
  - Concentrate on substance. Ideally, use a style guideline and conform to that
  - For lazy people, it is easier to argue about style than find real problems
- **Inspect, early, often, and as formally as you can**
  - Expect reviews to take a while to provide value
  - Keep records so you can document the value

[Wiegers95]

Honesty & Review Reports

- **The point of reviews is to find problems!**
  - If we ask you to document a review, we expect that you found (and then fixed after the review!) problems
  - If you say you did a review and found no problems, that is a little fishy
    - Perfect code is possible, but rare. (But if you get lucky, don't worry.)
    - We do not penalize you for bugs found in a review!
  - If you say you did 10 reviews and found no problems then one of:
    1. You didn't do very careful reviews
    2. Your lab partner walks on water and should immediately get a job for huge $$$
    3. You didn't actually do the reviews

Most likely, it is bin #1 or bin #3; please avoid those bins.

(No, we don't want you to make up problems just to report them)

Issue Log For Course Reviews

- **Include the following information:**
  - Developer name
  - Reviewer name
  - File name
  - Date of review
  - How long the review took   and length of file (non-blank lines)
  - List of defects found
  - Metrics:
    - Lines reviewed per hour   (varies depending on complexity of code)
    - Defects found per hour   (varies depending on complexity and defect density)
- **Real reviews are more formal and involve larger chunks of things**
  - We're just trying to give you a feel for how these things work

Review

- **Bit-based optimizations**
  - Strength reduction
  - Counting bits
  - Parity
- **Multiple precision math**
  - Addition/subtraction
  - Multiplication in gory detail
  - Division (just a general idea)
- **Reviews**
  - Reviews are very effective at finding defects
  - Basic good review practices

Lab Skills

- **Fast arithmetic**
  - Shifts as division

- **Multi-precision arithmetic**
  - Base lab: multi-precision addition/subtraction
    multi-precision multiplication
  - Bonus: division
  - Non-restoring division with a single shift-chain for dividend, quotient, and remainder is likely to make your brain hurt.
    - If you find multiplication easy you should give it a try, but…
    - Don't say we didn't warn you!

- **Each partner review code before hand-in**
  - Use the review report format
    - Yes, these are "toy" reviews. The point is for you to figure out the process without spending a lot of time on the mechanics