Goals

Function is correct
Source code is concise, readable, maintainable
Time-critical sections of program run fast enough
Object code is small and efficient

Basically, optimize the use of three resources:
- Execution time
- Memory
- Development/maintenance time

Like Writing English

You can say the same thing many different ways
and mean the same thing.

There are many different ways to say the same thing.
The same thing may be said different ways.
There is more than one way to say it.
Many sentences are equivalent.
Be succinct.

Arithmetic

Integer Arithmetic Fastest
Floating-point arithmetic in hardware Slower
Floating-point arithmetic in software Very slow

Simple benchmarks

```c
for (i = 0 ; i < 10000 ; ++i)
    /* arithmetic operation */
```

On my desktop Pentium 4 with good hardware floating-point support,

<table>
<thead>
<tr>
<th>Operator</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (int)</td>
<td>1</td>
</tr>
<tr>
<td>* (int)</td>
<td>5</td>
</tr>
<tr>
<td>/ (int)</td>
<td>12</td>
</tr>
<tr>
<td>sqrt</td>
<td>28</td>
</tr>
<tr>
<td>sin</td>
<td>48</td>
</tr>
<tr>
<td>pow</td>
<td>275</td>
</tr>
</tbody>
</table>

On my Zaurus SL 5600, a 400 MHz Intel PXA250 Xcale (ARM) processor:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (int)</td>
<td>1</td>
</tr>
<tr>
<td>+ (double)</td>
<td>140</td>
</tr>
<tr>
<td>* (int)</td>
<td>1</td>
</tr>
<tr>
<td>* (double)</td>
<td>110</td>
</tr>
<tr>
<td>/ (int)</td>
<td>7</td>
</tr>
<tr>
<td>/ (double)</td>
<td>220</td>
</tr>
<tr>
<td>sqrt</td>
<td>500</td>
</tr>
<tr>
<td>sin</td>
<td>3300</td>
</tr>
<tr>
<td>pow</td>
<td>820</td>
</tr>
</tbody>
</table>

C Arithmetic Trivia

Operations on char, short, int, and long probably run at the same speed (same ALU).
Same for unsigned variants
Int or long slower when they exceed machine's word size.
Operations on floats performed in double precision. float only useful for reducing memory.

Arithmetic Lessons

Try to use integer addition/subtraction
Avoid multiplication unless you have hardware
Avoid division
Avoid floating-point, unless you have hardware
Really avoid math library functions

Bit Manipulation

C has many bit-manipulation operators.
- & Bit-wise AND
- | Bit-wise OR
- ^ Bit-wise XOR
- ~ Negate (one's complement)
- >> Right-shift
- << Left-shift
Plus assignment versions of each.
Bit-manipulation basics

```c
a |= 0x4; /* Set bit 2 */
b &= ~0x4; /* Clear bit 2 */
c &= ~(1 << 3); /* Clear bit 3 */
d ^= (1 << 5); /* Toggle bit 5 */
e >>= 2; /* Divide e by 4 */
```

Advanced bit manipulation

```c
/* Set b to the rightmost 1 in a */
b = a & (a ^ (a - 1));
/* Set d to the number of 1's in c */
char c, d;
d = (c % 0x55) + ((c & 0xaa) >> 1);
d = (d & 0x33) + ((d & 0xcc) >> 2);
d = (d & 0x0f) + ((d & 0xf0) >> 4);
```

Faking Multiplication

Addition, subtraction, and shifting are fast. Can sometimes supplant multiplication.
Like floating-point, not all processors have a dedicated hardware multiplier.
Recall the multiplication algorithm from elementary school, but think binary:

```
101011
× 1101
10101100 = 43 + 43 << 2 + 43 << 3
101011000 = 43 << 1 + 43 << 2 + 43 << 3
1000101111 = 602
```

Faking Division

Division is a much more complicated algorithm that generally involves decisions.
However, division by a power of two is just a shift:

```
a / 2 = a >> 1
a / 4 = a >> 2
a / 8 = a >> 3
```

There is no general shift-and-add replacement for division, but sometimes you can turn it into multiplication:

```
a / 1.33333333
= a * 0.75
= a * 0.5 + a * 0.25
= a >> 1 + a >> 2
```

Faking Multiplication

Even more clever if you include subtraction:

```
101011
× 1110
1010110 = 43 << 1 + 43 << 2 + 43 << 3
10101100 = 43 << 4 − 43 << 2
10001011 = 559
```

Only useful
- for multiplication by a constant
- for "simple" multipliers
- when hardware multiplier not available

Multi-way branches

```
if (a == 1)
    foo();
else if (a == 2)
    bar();
else if (a == 3)
    baz();
else if (a == 4)
    qux();
else if (a == 5)
    quux();
else if (a == 6)
    corge();
```

```
switch (a) {
    case 1:
        foo();
        break;
    case 2:
        bar();
        break;
    case 3:
        baz();
        break;
    case 4:
        qux();
        break;
    case 5:
        quux();
        break;
    case 6:
        corge();
        break;
}
```

Nios code for if-then-else

```
ldw r2, 0(fp) # Fetch a from stack
cmpnei r2, r2, 1 # Compare with 1
bne r2, zero, .L2 # If not 1, jump to L2
call foo # Call foo()
br .L3 # branch out
.L2:
```

Nios code for switch (1)

```
ldw r2, 0(fp) # Fetch a
cmpnei r2, r2, 7 # Compare with 7
bne r2, zero, .L2 # If not 1, jump to L2
ldw r2, 0(fp) # Fetch a
muli r3, r2, 4 # Multiply by 4
movh r2, %hiadr,.L9 # Load address .L9
add r2, r2, %lo,.L9
add r2, r3, r2 # = a * 4 + .L9
ldw r2, 0(r2) # Fetch from jump table
jmp r2 # Jump to label
```

Nios code for switch (2)

```
.section .text
.L3:
call foo
br .L2
.L4:
call bar
br .L2
.L5:
call baz
br .L2
.L6:
call qux
br .L2
.L7:
call quux
br .L2
.L8:
call corge
.br .L2:
```

Nios code for switch (2)

```
.section .rodata.align 2
.align 2
```

```
.L9:
.long .L2 # Branch table
.long .L3
.long .L4
.long .L5
.long .L6
.long .L7
.long .L8
```

Low-Level C Programming – p. 10/49

Low-Level C Programming – p. 11/49

Low-Level C Programming – p. 12/49

Low-Level C Programming – p. 13/49

Low-Level C Programming – p. 14/49

Low-Level C Programming – p. 15/49

Low-Level C Programming – p. 16/49

Low-Level C Programming – p. 17/49

Low-Level C Programming – p. 18/49
Computing Discrete Functions

There are many ways to compute a “random” function of one variable:
/* OK, especially for sparse domain */
if (a == 0) x = 0;
else if (a == 1) x = 4;
else if (a == 2) x = 7;
else if (a == 3) x = 2;
else if (a == 4) x = 8;
else if (a == 5) x = 9;

/* Better for large, dense domains */
switch (a) {
  case 0: x = 0; break;
  case 1: x = 4; break;
  case 2: x = 7; break;
  case 3: x = 2; break;
  case 4: x = 8; break;
  case 5: x = 9; break;
}
/* Best: constant-time lookup table */
int f[] = {0, 4, 7, 2, 8, 9};
x = f[a]; /* assumes 0 <= a <= 5 */

Function calls

Modern processors, especially RISC, strive to make this cheap. Arguments passed through registers. Still has noticeable overhead.
Calling, entering, and returning:
int foo(int a, int b) {
  int c = bar(b, a);
  return c;
}

Strength Reduction

Why multiply when you can add?

Strength Reduction

Why multiply when you can add?

Why multiply when you can add?

Why multiply when you can add?

Why multiply when you can add?

Why multiply when you can add?

Why multiply when you can add?

Why multiply when you can add?

Why multiply when you can add?
**Optimized (−O2) pointer code**

```
optimized_code
```

**How Rapid is Rapid?**

How much time does the following loop take?

```
for ( i = 0 ; i < 1024 ; ++i) a += b[i];
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles per Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory read</td>
<td>2 or 7</td>
</tr>
<tr>
<td>Addition</td>
<td>1</td>
</tr>
<tr>
<td>Loop overhead</td>
<td>≈4</td>
</tr>
</tbody>
</table>

Total: 6–12

The Nios runs at 50 MHz, one instruction per cycle, so this takes

\[
\frac{6 \cdot 1024}{50\text{MHz}} = 0.12\mu s \text{ or } 12 \cdot 1024 \cdot \frac{1}{50\text{MHz}} = 0.24\mu s
\]

**Double-checking**

GCC generates good code with −O7:

```
GCC_code
```

**Features in order of increasing cost**

1. Integer arithmetic
2. Pointer access
3. Simple conditionals and loops
4. Static and automatic variable access
5. Array access
6. Floating-point with hardware support
7. Switch statements
8. Function calls
9. Floating-point emulation in software
10. malloc() and free()
11. Library functions (sin, log, printf, etc.)
12. Operating system calls (open, sbrk, etc.)

**Storage Classes in C**

```
/* fixed address: visible to other files */
int global_static;

/* fixed address: only visible within file */
static int file_static;

/* parameters always stacked */
int foo(int auto_param){
    /* fixed address: only visible to function */
    static int func_static;
    /* stacked: only visible to function */
    int auto_i, auto_a[10];
    /* array explicitly allocated on heap */
    double *auto_d = malloc(sizeof(double)*5);
    /* return value in register or stacked */
    return auto_i;
}
```

**Dynamic Storage Allocation**

Rules:
- Each allocated block contiguous (no holes)
- Blocks stay fixed once allocated
- malloc()
  - Find an area large enough for requested block
  - Mark memory as allocated
- free()
  - Mark the block as unallocated

```
Dynamic_allocation
```

**Simple Dynamic Storage Allocation**

Maintaining information about free memory
- Simplest: Linked list
- The algorithm for locating a suitable block
  - Simplest: First-fit
- The algorithm for freeing an allocated block
  - Simplest: Coalesce adjacent free blocks

```
Simple_allocation
```

**Dynamic Storage Allocation**

```
Dynamic_allocation
```

```
Dynamic_allocation
```
Simple Dynamic Storage Allocation

On most processors, access to automatic (stacked) data and globals is equally fast.
Automatic usually preferable since the memory is reused when function terminates.
Danger of exhausting stack space with recursive algorithms. Not used in most embedded systems.
The heap (malloc) should be avoided if possible:
- Allocation/deallocation is unpredictably slow
- Danger of exhausting memory
- Danger of fragmentation
Best used sparingly in embedded systems

Storage Classes Compared

On most processors, access to automatic (stacked) data and globals is equally fast.
Automatic usually preferable since the memory is reused when function terminates.
Danger of exhausting stack space with recursive algorithms. Not used in most embedded systems.
The heap (malloc) should be avoided if possible:
- Allocation/deallocation is unpredictably slow
- Danger of exhausting memory
- Danger of fragmentation
Best used sparingly in embedded systems

Memory-Mapped I/O

“We magical” memory locations that, when written or read, send or receive data from hardware.
Hardware that looks like memory to the processor, i.e., addressable, bidirectional data transfer, read and write operations.
Does not always behave like memory:
- Act of reading or writing can be a trigger (data irrelevant)
- Often read- or write-only
- Read data often different than last written

Memory-Mapped I/O Access in C

```c
#define SWITCHES \ ((volatile char *) 0x1800)
#define LEDS \ ((volatile char *) 0x1810)

void main() {
  for (;;) {
    *LEDS = *SWITCHES;
  }
}
```

What’s With the Volatile?

```c
#define ADDRESS \ ((char *) 0x1800)
#define VADDRESS \ ((volatile char *) 0x1800)

char foo() {
  char a = *ADDRESS;
  char b = *ADDRESS;
  return a + b;
}

char bar() {
  char a = *VADDRESS;
  char b = *VADDRESS;
  return a + b;
}
```

Compiled with optimization:

```c
char foo() {
  char a = ADDRESS;
  char b = ADDRESS;
  return a + b;
}

char bar() {
  char a = VADDRESS;
  char b = VADDRESS;
  return a + b;
}
```

Altera I/O

```c
/* Definitions of alt_u8, etc. */
#include "alt_types.h"

/* IORD_ALTERA_AVALON P0... for the "PIO" device */
#include "altera_avalon_pio_regs.h"

#include "system.h"

int main() {
  alt_u8 sw;
  for (;;) {
    sw = IORD_ALTERA_AVALON_PIO_DATA(SWITCHES_BASE);
    IOWR_ALTERA_AVALON_PIO_DATA(LEDS_BASE, sw);
  }
}
```

HW/SW Communication Styles

Memory-mapped I/O puts the processor in charge: only it may initiate communication.
Typical operation:
- Check hardware conditions by reading “status registers”
- When ready, send next “command” by writing control and data registers
- Check status registers for completion, waiting if necessary
Waiting for completion: “polling”
“Are we there yet?” “No.” “Are we there yet?” “No.” “Are we there yet?” “No.” “Are we there yet?” “No.”

HW/SW Communication: Interrupts

Idea: have hardware initiate communication when it wants attention.
Processor responds by immediately calling an interrupt handling routine, suspending the currently-running program.

Unix Signals

The Unix environment provides “signals,” which behave like interrupts.

```c
#include <stdio.h>
#include <signal.h>

void handleint() {
  printf("Got an INT\n");
}

int main() {
  signal(SIGINT, handleint);
  for (;;) {
    return 0;
  }
}
```

(From the Nios II Software Developer’s Handbook)
Interrupts under Altera (1)

```c
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

static void button_isr(void* context, alt_u32 id) {
    /* Read and store the edge capture register */
    *(volatile int *) context =
        IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
    /* Write to the edge capture register to reset it */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);
    /* Reset interrupt capability for the Button PIO */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
}
```

Interrupts under Altera (2)

```c
#include "sys/alt_irq.h"
#include "system.h"

volatile int captured_edges;

static void init_button_pio() {
    /* Enable all 4 button interrupts. */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);
    /* Register the ISR. */
    alt_irq_register( BUTTON_PIO_IRQ,
            &captured_edges,
            button_isr );
}
```

The Edwards Way to Debug

1. Identify undesired behavior
2. Construct linear model for desired behavior
3. Pick a point along model
4. Form desired behavior hypothesis for point
5. Test
6. Move point toward failure if point working, away otherwise
7. Repeat #4–#6 until bug is found