**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

**Arithmetic**

<table>
<thead>
<tr>
<th>Integer Arithmetic</th>
<th>Fastest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point arithmetic in hardware</td>
<td>Slower</td>
</tr>
<tr>
<td>Floating-point arithmetic in software</td>
<td>Very slow</td>
</tr>
</tbody>
</table>

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.

**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 (int)</td>
<td>2</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 Xscale (ARM) processor:

<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>140</td>
</tr>
<tr>
<td>/ (int)</td>
<td>110</td>
</tr>
<tr>
<td>sqrt (int)</td>
<td>28</td>
</tr>
<tr>
<td>sin</td>
<td>300</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.

**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

**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.

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

Faking Multiplication

```
/* 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 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
```

Multi-way branches

```
if (a == 1)
  case 1:
    foo();
    break;
  else if (a == 2)
    case 2:
      bar();
      break;
    else if (a == 3)
      case 3:
        baz();
        break;
    else if (a == 4)
      case 4:
        qux();
        break;
    else if (a == 5)
      case 5:
        quxx();
        break;
    else if (a == 6)
      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:
ldw r2, 0(fp)  # Fetch a from stack (again!)
cmpnei r2, r2, 2  # Compare with 2
bne r2, zero, .l4  # If not 1, jump to L4
call bar  # Call bar()
br .l5  # branch out
.l4:
```

Nios code for switch (1)

```
ldw r2, 0(fp)  # Fetch a
cmpgeui r2, r2, 7  # Compare with 7
bne r2, zero, .l2  # Branch if greater or equal
ldw r2, 0(fp)  # Fetch a
mul r3, r2, 4  # Multiply by 4
movhi r2, %hiadj(.l9)  # Load address .l9
addi 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  # Branch table
.br .l2
.L4:
call bar  # Branch table
.br .l2
.L5:
call baz  # Branch table
.br .l2
.L6:
call qux  # Branch table
.br .l2
.L7:
call quxx  # Branch table
.br .l2
.L8:
call corge  # Branch table
.br .l2
```
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;
}

Code for foo() (unoptimized)
f:

addi sp, sp, -20  # Allocate space on stack
stw ra, 16(sp)  # Store return address
stw fp, 12(sp)  # Store frame pointer
mov fp, sp  # Frame pointer is new SP
stw r4, 0(fp)  # Save a on stack
stw r5, 4(fp)  # Save b on stack
lw r4, 4(fp)  # Fetch b
lw r5, 0(fp)  # Fetch a
call bar  # Call bar()
stw r2, 8(fp)  # Store result in c
lw r2, 8(fp)  # Return value in r2 = c
lw ra, 16(sp)  # Restore return address
lw fp, 12(sp)  # Restore frame pointer
add sp, sp, 20  # Release stack space
ret  # Return from subroutine

Unoptimized array code (fragment)

.L2:
lw r2, 0(fp)  # Fetch i
cmpeq r2, r2, 10  # i >= 10?
bne r2, zero, .L1  # exit if true
mov r3, @hiadj(foo)  # Get address of foo array
add r3, r3, @lo(foo)
lw r2, 0(fp)  # Fetch i
mul r2, r2, 12  # i * 12
add r2, r2, r2  # + foo[i]
add r2, 77
stw r2, 0(r3)  # foo[i] = 77
mov r3, @hiadj(foo)
add r3, r3, @lo(foo)
lw r2, 0(fp)
muli r2, r2, 12
add r2, r2, r3  # compute &foo[i]
add r2, r2, 4  # offset for b field
mov r2, 88
stb r2, 0(r3)  # foo[i].b = 88

Unoptimized pointer code (fragment)

.L2:
lw r2, 0(fp)  # Fetch i
cmpeq r2, r2, 10  # i >= 10?
bne r2, zero, .L1  # exit if true
mov r3, @hiadj(foo)  # Get address of foo array
add r3, r3, @lo(foo)
lw r2, 0(fp)
lw r3, 0(fp)  # Fetch i
mul r3, r3, 12  # i * 12
add r3, r3, r3  # + foo[i]
add r3, 77
stw r3, 0(r2)  # foo[a] = 77
mov r2, 0(fp)
mov r2, r2, 4  # offset for b field
mov r2, 88
stb r2, 0(r3)  # foo[i].b = 88

Optimized (-O2) array code

movi r6, 77  # Load constants
movi r5, 88
movi r4, 99
movi r2, @hiadj(foo)  # Load address of array
addi r2, r2, @lo(foo)
movi r3, 10  # iteration count
.L5:
lw r3, 0(fp)  # foo[a] = 77
mov r3, r3, r3  # i * 12
add r3, r3, r3  # + foo[i]
add r3, 77
stw r3, 0(r2)  # foo[a] = 77
mov r2, 0(fp)
mov r2, r2, 4  # offset for b field
mov r2, 88
stb r2, 0(r3)  # foo[i].b = 88
movi r2, r2, 12  # go to next array element
bne r3, zero, .L5  # if there are more to do
ret

Strength Reduction

Why multiply when you can add?

struct {
    int a;
    char b;
    int c;
} foo[10];

int i;
for (i = 0; i < 10; ++i) {
    foo[i].a = 77;
    foo[i].b = 88;
    foo[i].c = 99;
}

Good optimizing compilers do this automatically.
Optimized (–O2) pointer 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

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

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
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

Low-Level C Programming – p. 37

Memory-Mapped I/O

“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

Low-Level C Programming – p. 38

Memory-Mapped I/O Access in C

#define SWITCHES \ ((volatile char *) 0x1800)
#define LEDS \ ((volatile char *) 0x1810)
void main() {
    for (;;) {
        *LEDS = *SWITCHES;
    }
}

Low-Level C Programming – p. 39

What’s With the Volatile?

#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:

foo:
    movi r2, 6144
    ldbu r2, 0(r2)
    add r2, r2, r2
    andi r2, r2, 0xff
    ret

bar:
    movi r3, 6144
    ldbu r2, 0(r3)
    ldbu r3, 0(r3)
    add r2, r2, r3
    andi r2, r2, 0xff
    ret

Low-Level C Programming – p. 40

Altera I/O

/* Definitions of alt_u8, etc. */
#include “alt_types.h”

/* IORD_ALTERA_AVALON... for the ‘PIO’ device */
#include “altera_avalon_pio_regs.h”

/* Auto-generated addresses for all peripherals */
#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);
    }
}

(From the Nios II Software Developer’s Handbook)

Low-Level C Programming – p. 41

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.

#include <stdio.h>
#include <signal.h>
void handleint() {
    printf(“Got an INT\n”);
    /* some variants require this */
    signal(SIGINT, handleint);
}

int main() {
    signal(SIGINT, handleint);
    /* Do nothing forever */
    for (;;) {
    }
    return 0;
}
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,
        (void *) &captured_edges,
        button_isr );
}
```

Debugging Skills

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