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

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>+ (double)</td>
<td>5</td>
</tr>
<tr>
<td>* (int)</td>
<td>5</td>
</tr>
<tr>
<td>* (double)</td>
<td>5</td>
</tr>
<tr>
<td>/ (int)</td>
<td>12</td>
</tr>
<tr>
<td>/ (double)</td>
<td>10</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 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>+ (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
- Negate (one's complement)
- Right-shift
- Left-shift
Plus assignment versions of each.
Bit-manipulation basics

```
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

```
/* Set b to the rightmost 1 in a */
b = 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

```
101011
1101
101011
10101100
101011000
+1000101111
= 559
```

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
```

Microblaze code for if-then-else

```
lwi r3,r19,44  # fetch "a" from stack
addik r18,r0,1 # load constant 1
cmp r18,r18,r3 # compare with "a"
bnei r18,SL3  # skip if not equal
brlid r15,foo # call foo
bri $L4     # branch to end

$L3:
lwi r3,r19,44  # fetch "a" from stack
addik r18,r0,2 # load constant 2
cmp r18,r18,r3 # compare with "a"
bnei r18,SL5  # skip if not equal
brlid r15,bar # call bar
bri $L4      # branch to end
```

Microblaze code for switch (1)

```
.addik r3,r22,-1
xor r18,r3,5
beq r18,r10
bli r3,SL14 # Skip if less than 1
bri $L1

.rsubik r18,r3,5
bli r18,SL14 # Skip if greater than 6
$L1:
addk r3,r3,r3
addk r3,r3,r3
lwi r3,r3,SL21 # Fetch address from table
bra r3 # Branch to a case label.sdata2

.L21: # Branch table
    .gpword $L15
    .gpword $L16
    .gpword $L17
    .gpword $L18
    .gpword $L19
    .gpword $L20
```

Microblaze code for switch (2)

```
.text
$L15: # case 1:
brlid r15,foo
bri $L14

$L16: # case 2:
brlid r15,bar
bri $L14

$L17: # case 3:
brlid r15,baz
bri $L14

$L18: # case 4:
brlid r15,qux
bri $L14

$L19: # case 5:
brlid r15,quux
bri $L14
```

Faking Multiplication

```
if (a == 1)    switch (a) {
    foo();        case 1:
    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 (a == 5) case 5:
    quux();
    break;
else if (a == 6) case 6:
    corge();
    break;
}
```
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 */

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 */

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

/* 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 */

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 */

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

/* 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 */

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

How much time does the following loop take?
for (i = 0; i < 1024; ++i) a += b[i];

Operation Cycles per iteration
Memory read 2 or 7
Addition 1
Loop overhead \( \approx 4 \)
Total 6–12

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

\[ 6 \times 1024 \times \frac{1}{50 \text{MHz}} = 0.12 \text{µs} \text{ or } 12 \times 1024 \times \frac{1}{50 \text{MHz}} = 0.24 \text{µs} \]

Double-checking

GCC generates great code with -O7:

```c
sumarray:
  addik r1, r1, -24  # create frame
  add r4, r0, r0  # a = 0
  addik r6, r5, 4092  # end of array
  lw r3, r0, r5  # b[i] 2-7
  addik r5, r5, 4  # ++i 1
  add r4, r4, r3  # a += b[i] 1
  cmp r18, r6, r5  # i < 1024 1
  blee r18, $L6 3
  add r3, r4, r0  # return a
  rtsd r15, 8
  addik r1, r1, 24  # release frame
$L6:
  # cycles
  lw r3, r0, r5
  add r4, r4, r3
  cmp r18, r6, r5
  blee r18, $L6
```

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_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

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 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
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

“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

With the Microblaze

Xilinx supplies a library of I/O operations:
```c
#include "xbasic_types.h"
#include "xio.h"
XIo_In8(XIo_Address address)
XIo_In16(XIo_Address address)
XIo_In32(XIo_Address address)
void XIo_Out8(XIo_Address address, Xuint8 data)
void XIo_Out16(XIo_Address address, Xuint16 data)
void XIo_Out32(XIo_Address address, Xuint32 data)
```
Each is a simple macro, e.g.:
```c
#define XIo_Out32(Addr, Value) 
{ (*(volatile Xuint32 *)(Addr) = Value); }
```
volatile warns compiler not to optimize it

An example

```c
#include "xbasic_types.h"
#include "xio.h"

int main()
{
  int i, j;
  printf("Hello World!\r\n");
  for(j=0;j<256;j++)
  {
    XIo_Out32(0xFEFF0200, j<<24);
    XIo_Out32(0xFEFF0204, j<<24);
    XIo_Out32(0xFEFF0208, j<<24);
    XIo_Out32(0xFEFF020C, j<<24);
  }
  print("Goodbye\r\n");
  return 0;
}
```

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”

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");
  /* some variants require this */
  signal(SIGINT, handleint);
}

int main() {
  /* Register signal handler */
  signal(SIGINT, handleint);
  /* Do nothing forever */
  for (;;) { }
  return 0;
}
```

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.

UART interrupts on the Microblaze

```c
#include "xbasic_types.h"
#include "xio.h"
#include "xparameters.h"
#include "xuartlite_l.h"

int main()
{
  XIntc_RegisterHandler( XPAR_INTC_BASEADDR, XPAR_MYUART_DEVICE_ID, (XinterruptHandler)uart_handler, (void *)0);
  XIntc_mEnableIntr(XPAR_INTC_BASEADDR, XPAR_MYUART_INTERRUPT_MASK);
  XIntc_Out32(XPAR_INTC_BASEADDR + XPAR_MYUART_INTERRUPT_MASK);
  microblaze_enable_interrupts();
  XUartLite_mEnableIntrs(XPAR_MYUART_BASEADDR);
}
```

```c
void uart_handler(void *callback)
{
  Xuint32 IsrStatus;
  Xuint8 incoming_character;
  IsrStatus = XPAR_INTC_BASEADDR + XPAR_MYUART_INTERRUPT_REG;
  if ( ((IsrStatus & XPAR_SR_RX_FIFO_FULL) || (IsrStatus & XPAR_SR_RX_FIFO_VALID_DATA)) != 0 )
  {
    incoming_character = XPAR_SR_RX_FIFO_OFFSET;
    XUartLite_mDisableIntrs(XPAR_MYUART_BASEADDR);
  }
  if ( ((IsrStatus & XPAR_SR_TX_FIFO_EMPTY) != 0)
  { /* output FIFO empty: can send next char */
```
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

The Edwards Way to Debug

Xilinx \textit{platgen} uses this to piece together the netlist from library components. Excerpt:

- \texttt{PORT VIDOUT\_GY = VIDOUT\_GY, DIR = OUT, VEC = \{9:0\}}
- \texttt{PORT VIDOUT\_BCB = VIDOUT\_BCB, DIR = OUT, VEC = \{9:0\}}
- \texttt{PORT RS232\_TD = RS232\_TD, DIR = IN}

\texttt{BEGIN microblaze}
\texttt{PARAMETER INSTANCE = nymicroblaze}
\texttt{PARAMETER HW\_VER = 2.00.a}
\texttt{PARAMETER C\_USE\_BARREL = 1}
\texttt{END}

\texttt{BEGIN opb\_uartlite}
\texttt{PARAMETER INSTANCE = myuart}
\texttt{PARAMETER C\_CIR\_FREQ = 50.000.000}
\texttt{PARAMETER C\_BASEADDR = 0xFEFF0100}
\texttt{PARAMETER C\_HIGHADDR = 0xFEFF01FF}
\texttt{END}

\texttt{The Xilinx Tool Chain}

The \textit{libgen} uses this to link software. Excerpt:

\texttt{BEGIN PROCESSOR}
\texttt{PARAMETER HW\_INSTANCE = mymicroblaze}
\texttt{PARAMETER DRIVER\_NAME = cpu}
\texttt{PARAMETER DRIVER\_VER = 1.00.a}
\texttt{PARAMETER EXECUTABLE = hello\_world.elf}
\texttt{PARAMETER COMPILER = mb-gcc}
\texttt{PARAMETER ARCHIVER = mb-ar}
\texttt{PARAMETER DEFAULT\_INIT = EXECUTABLE}
\texttt{PARAMETER STDIN = myuart}
\texttt{PARAMETER STDOUT = myuart}
\texttt{END}

\texttt{BEGIN DRIVER}
\texttt{PARAMETER HW\_INSTANCE = mymicroblaze}
\texttt{PARAMETER DRIVER\_NAME = uartlite}
\texttt{PARAMETER DRIVER\_VER = 1.00.a}
\texttt{PARAMETER LEVEL = 1}
\texttt{END}

The \textit{ucf} file

Pin assignments and other global chip information.

- \texttt{net sys\_clk period = 18.000;}
- \texttt{net pixel\_clock period = 36.000;}
- \texttt{net VIDOUT\_GY<0> loc="p9" ;}
- \texttt{net VIDOUT\_GY<1> loc="p10" ;}
- \texttt{net VIDOUT\_GY<2> loc="p11" ;}
- \texttt{net VIDOUT\_BCB<0> loc="p42" ;}
- \texttt{net VIDOUT\_BCB<1> loc="p43" ;}
- \texttt{net VIDOUT\_BCB<2> loc="p44" ;}
- \texttt{net RS232\_TD loc="p71" ;}
- \texttt{net RS232\_RD loc="p72" ;}