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

+, −                        Faster
×                         Slower
sqrt, sin, log, etc.        Slower

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>« (int)</td>
<td>2</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>

Simple benchmarks

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>« (int)</td>
<td>1</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

Even more clever if you include subtraction:

```
101011
× 1110
101011010
101011000
+101011000
1000101111
```

```
= 43 + 43 × 2 + 43 × 3 = 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
```

Multi-way branches

```c
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();
```

Microblaze code for if-then-else

```
lwi r3,r19,44 # fetch "a" from stack
addik r18,r0,1 # load constant 1
cmpeq r18,r18,r3 # compare with "a"
bnep r18,$L3 # skip if not equal
brLeod r15,foo # call foobar
nop # delay slot
bri $L4 # branch to end

$L3:
lwi r3,r19,44 # fetch "a" from stack
addik r18,r0,2 # load constant 2
cmpeq r18,r18,r3 # compare with "a"
bnep r18,$L5 # skip if not equal
brLeod r15,bar # call bar
nop # delay slot
bri $L4 # branch to end

$L5:
```

Microblaze code for switch (1)

```
addik r3,r22,-1
xori r18,r3,5
beq r18,$L10
bti r3,$L14 # Skip if less than 1
bri $L1

$L1:
rneq r18,r18,r3 # skip if not equal
bti r18,$L14 # Skip if greater than 6
bri $L1

$L1:
addk r3,r3,3 # Multiply by four
lwi r3,r3,$L21 # Fetch address from table
bra r3 # Branch to a case label
.sdata2

$L21:
    .gpword $L15
    .gpword $L16
    .gpword $L17
    .gpword $L18
    .gpword $L19
    .gpword $L20
    .branch $L1
```

Microblaze code for switch (2)

```
.text

$L15: # case 1:
    brLeod r15,foo
    bri $L14

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

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

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

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

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
101011000
+101011000
1000101111
```

```
= 43 + 43 × 2 + 43 × 3 = 559
```
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 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
\[
6 \times 1024 \times \frac{1}{50 \text{MHz}} = 0.12 \text{µs} \quad \text{or} \quad 12 \times 1024 \times \frac{1}{50 \text{MHz}} = 0.24 \text{µs}
\]

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

$L6$: # cycles
    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
    bnei r18, $L6 3
    add r3, r4, r0 # return a
    rtsd r15, 8
    addik r1, r1, 24 # release frame
```

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

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

hello.c from the first lab

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

int main()
{
  int i, j;
  printf("Hello World!\n");
  for(j=0;j<256;j++)
    for(i=0;i<100000;i++)
      { XIo_Out32(0xFEFF0200, j<<24); XIo_Out32(0xFEFF0204, j<<24); XIo_Out32(0xFEFF0208, j<<24); XIo_Out32(0xFEFF020C, j<<24); }
  printf("Goodbye\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.”

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

```c
int main()
{
  int i, j;
  printf("Hello World!\n");
  for(j=0;j<256;j++)
    for(i=0;i<100000;i++)
      { XIo_Out32(0xFEFF0200, j<<24); XIo_Out32(0xFEFF0204, j<<24); XIo_Out32(0xFEFF0208, j<<24); XIo_Out32(0xFEFF020C, j<<24); }
  printf("Goodbye\n");
  return 0;
}
```

UART interrupts on the Microblaze

```c
#include "xbasic_types.h"
#include "xio.h"
#include "xintc_l.h"
#include "xuartlite_l.h"
#include "xparameters.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_mEnableIntr( XPAR_INTC_BASEADDR + XPAR_MYUART_BASEADDR + XPAR_RF_IRQ_OFFSET);
  XUartLite_mEnableIntr(XPAR_MYUART_BASEADDR);
  microblaze_print("uart init done\n");
}
```

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

void uart_handler(void *callback)
{
  Xuint32 IsrStatus;
  Xuint32 incoming_character;
  Xuint32 outgoing_character = XPAR_MYUART_BASEADDR + XPAR_RF_IRQ_OFFSET;

  if (IsrStatus & (XUL_SR_RF_IRQ_FULL | XPAR_RF_IRQ_VALID_DATA)) != 0)
    if (incoming_character)
      microblaze_print((char *)incoming_character);
  outgoing_character = XPAR_MYUART_BASEADDR + XPAR_RF_IRQ_OFFSET;
  if (IsrStatus & XPAR_RF_IRQ_EMPTY) != 0)
    microblaze_print((char *)XPAR_RF_IRQ_EMPTY);
}
```

```c
UART interrupts on the Microblaze

```
Debugging Skills

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

The .mhs File

Xilinx platgen uses this to piece together the netlist from library components. Excerpt:

PORT VIDOUT_GY = VIDOUT_GY, DIR = OUT, VCC = [9:0]
PORT VIDOUT_BCB = VIDOUT_BCB, DIR = OUT, VCC = [9:0]
PORT FPGA_CLK1 = FPGA_CLK1, DIR = IN
PORT RS232_TD = RS232_TD, DIR=OUT

BEGIN microblaze
PARAMETER INSTANCE = mymicroblaze
PARAMETER HW_INSTANCE = 2.00.a
PARAMETER C_USE_BARREL = 1
END

BEGIN uartlite
PARAMETER INSTANCE = myuart
PARAMETER C_BASEADDR = 0x07F0100
PARAMETER C_HIGHADDR = 0x07F01FF
END

The .mss File

Used by Xilinx libgen to link software. Excerpt:

BEGIN PROCESSOR
PARAMETER HW_INSTANCE = mymicroblaze
PARAMETER DRIVER_NAME = cpu
PARAMETER DRIVER_VER = 1.00.a
PARAMETER EXECUTABLE = hello_world.elf
PARAMETER COMPILER = mb-gcc
PARAMETER ARCHIVER = mb-ar
PARAMETER DEFAULT_INIT = EXECUTABLE
PARAMETER STDIN = myuart
PARAMETER STDOUT = myuart
END

PARAMETER HW_INSTANCE = myuart
PARAMETER DRIVER_NAME = uartlite
PARAMETER DRIVER_VER = 1.00.b
PARAMETER LEVEL = 1
END

The .ucf file

Pin assignments and other global chip information.

Lab 1
Write and execute a C program that counts in decimal on the two 7-segment displays on the XSB-300E.
We supply
- A hardware configuration consisting of a processor, UART, and
- A simple memory-mapped peripheral that latches and displays a byte controlling each segment of the displays.
- A skeleton project that compiles, downloads, and prints “Hello World” through the serial debugging cable.

Your Job
Write and test C code that
- Counts
- Converts the number into arabic numerals on the display
- Transmits this to the display
Goal: Learn basics of the tools, low-level C coding, and memory-mapped I/O.

Debugging Lab 1
- Examine build error messages for hints
- “make clean” sometimes necessary
- Call print to send data back to the host
- Run Minicom on /dev/ttyS0 (9600 8n1) to observe output