Low-Level C Programming

CSEE W4840

Prof. Stephen A. Edwards

Columbia University

Spring 2007

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

$+, -$

$\times$

$\div$

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,

Operator	Time	Operator	Time
+ (int)	1	+ (double)	5
* (int)	5	* (double)	5
/ (int)	12	/ (double)	10
« (int)	2	`sqrt`	28
		`sin`	48
		`pow`	275

Simple benchmarks

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

Operator	Time		
+ (int)	1	+ (double)	140
* (int)	1	* (double)	110
/ (int)	7	/ (double)	220
« (int)	1	`sqrt`	500
		`sin`	3300
		`pow`	820

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 `float`s 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

```
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 ^ (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:

$$\begin{array}{r} 101011 \\ \times\ 1101 \\ \hline 101011 \\ 10101100 \\ +101011000 \\ \hline 1000101111 \end{array} = 43 \ll 0 + 43 \ll 2 + 43 \ll 3 = 559$$

Faking Multiplication

Even more clever if you include subtraction:

$$\begin{array}{r} 101011 \\ \times\ 1110 \\ \hline 1010110 \\ 10101100 \\ +101011000 \\ \hline 1001011010 \end{array} \quad \begin{array}{l} = 43 \ll 1 + 43 \ll 2 + 43 \ll 3 \\ = 43 \ll 4 - 43 \ll 2 \\ = 602 \end{array}$$

Only useful

- for multiplication by a constant
- for "simple" multiplicands
- when hardware multiplier not available

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)
  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:
  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      .L3            # 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
    muli    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
    .section        .rodata
    .align  2
.L9:
    .long   .L2             # Branch table
    .long   .L3
    .long   .L4
    .long   .L5
    .long   .L6
    .long   .L7
    .long   .L8
```

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
.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;
```


Computing Discrete Functions

```
/* 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 noticable overhead.

Calling, entering, and returning:

```
int foo(int a, int b) {
  int c = bar(b, a);
  return c;
}
```


Code for foo() (unoptimized)

```
foo:
  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

  ldw  r4, 4(fp)    # Fetch b
  ldw  r5, 0(fp)    # Fetch a
  call bar          # Call bar()
  stw  r2, 8(fp)    # Store result in c

  ldw  r2, 8(fp)    # Return value in r2 = c
  ldw  ra, 16(sp)   # Restore return address
  ldw  fp, 12(sp)   # Restore frame pointer
  addi sp, sp, 20   # Release stack space
  ret               # Return from subroutine
```


Code for foo() (optimized)

```
foo:
  addi sp, sp, -4  # Allocate stack space
  stw  ra, 0(sp)   # Store return address
  mov  r2, r4      # Swap arguments (r4, r5)
  mov  r4, r5      #   using r2 as temporary
  mov  r5, r2
  call bar         # Call bar() (return in r2)
  ldw  ra, 0(sp)   # Restore return address
  addi sp, sp, 4   # Release stack space
  ret              # Return from subroutine
```


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;
}
```

```
struct {
  int a;
  char b;
  int c;
} *fp, *fe, foo[10];

fe = foo + 10;
for (fp = foo ; fp != fe ; ++fp) {
  fp->a = 77;
  fp->b = 88;
  fp->c = 99;
}
```

Good optimizing compilers do this automatically.

Unoptimized array code (fragment)

```
.L2:
  ldw    r2, 0(fp)        # Fetch i
  cmpgei r2, r2, 10          # i >= 10?
  bne    r2, zero, .L1          # exit if true
  movhi  r3, %hiadj(foo)  # Get address of foo array
  addi   r3, r3, %lo(foo)
  ldw    r2, 0(fp)          # Fetch i
  muli   r2, r2, 12       # i * 12
  add    r3, r2, r3       # foo[i]
  movi   r2, 77
  stw    r2, 0(r3)        # foo[i].a = 77
  movhi  r3, %hiadj(foo)
  addi   r3, r3, %lo(foo)
  ldw    r2, 0(fp)
  muli   r2, r2, 12
  add    r2, r2, r3       # compute &foo[i]
  addi   r3, r2, 4        # offset for b field
  movi   r2, 88
  stb    r2, 0(r3)        # foo[i].b = 88
```

Unoptimized pointer code (fragment)

```
.L2:
  ldw  r3, 0(fp)    # fp
  ldw  r2, 4(fp)    # fe
  beq  r3, r2, .L1  # fp == fe?
  ldw  r3, 0(fp)
  movi r2, 77
  stw  r2, 0(r3)    # fp->a = 77
  ldw  r3, 0(fp)
  movi r2, 88
  stb  r2, 4(r3)    # fp->b = 88
  ldw  r3, 0(fp)
  movi r2, 99
  stw  r2, 8(r3)    # fp->c = 99
  ldw  r2, 0(fp)
  addi r2, r2, 12
  stw  r2, 0(fp)    # ++fp
  br   .L2
```

Optimized (–O2) array code

```
  movi  r6, 77          # Load constants
  movi  r5, 88
  movi  r4, 99
  movhi r2, %hiadj(foo) # Load address of array
  addi  r2, r2, %lo(foo)
  movi  r3, 10          # iteration count
.L5:
  addi  r3, r3, -1      # decrement iterations
  stw   r6, 0(r2)       # foo[i].a = 77
  stb   r5, 4(r2)       # foo[i].b = 88
  stw   r4, 8(r2)       # foo[i].c = 99
  addi  r2, r2, 12      # go to next array element
  bne   r3, zero, .L5   # if there are more to do
  ret
```

Optimized (–O2) pointer code

```
movhi r6, %hiadj(foo+120)   # fe = foo + 10
addi  r6, r6, %lo(foo+120)
addi  r2, r6, -120          # fp = foo
movi  r5, 77                # Constants
movi  r4, 88
movi  r3, 99
.L5:
stw   r5, 0(r2)             # fp->a = 77
stb   r4, 4(r2)             # fp->b = 88
stw   r3, 8(r2)             # fp->c = 99
addi  r2, r2, 12            # ++fp
bne   r2, r6, .L5           # fp == fe?
ret
```


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	≈4
Total	6–12

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

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

Double-checking

GCC generates good code with –O7:

```
  movhi r4, %hiadj(b)  # Load &b[0]
  addi  r4, r4, %lo(b)
  movi  r3, 1024       # Iteration count

.L5:                   #                    cycles
  ldw   r2, 0(r4)      # Fetch b[i]           2-7
  addi  r3, r3, -1     # --i                   1
  addi  r4, r4, 4      # next b element        1
  add   r5, r5, r2     # a += b[i]             1
  bne   r3, zero, .L5  # repeat if i > 0       3
  mov   r2, r5         # result
  ret
```


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

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

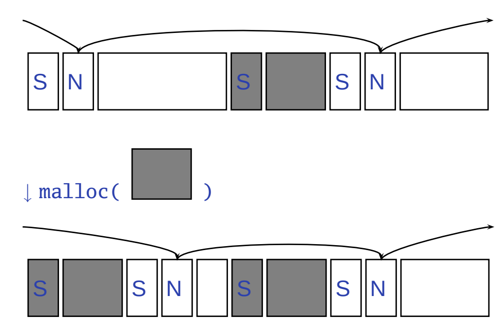
The algorithm for locating a suitable block

- Simplest: First-fit

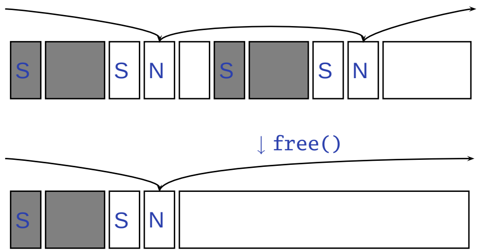
The algorithm for freeing an allocated block

- Simplest: Coalesce adjacent free blocks

Dynamic Storage Allocation



Simple Dynamic Storage Allocation



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

Memory-Mapped I/O Access in C

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

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


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
```


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)

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"

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

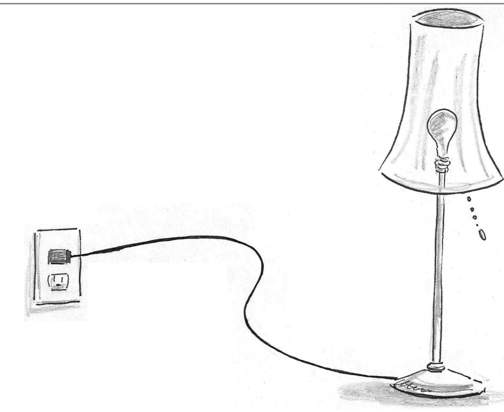
Interrupts under Altera (1)

```
#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)

```
#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



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