Something Cool

- RFID is an exciting and growing technology
- This reader from Parallax is $40 and has a serial interface
Lab

- Lab 1 due next Tues
- Seemed to go pretty well on Tues?
- Questions?
Quiz Results

- **Problem 1**: About 50% of class got it totally right
- **Problem 2**:  
  - Most everyone got the first 4 parts correct  
  - Remaining 3 parts were about 60%
- **Problem 3**: 40%
- **Problem 4**: 50%
- **Problem 5**: 90% was close, about 20% was totally correct
Last Time

- Low-level parts of the toolchain for embedded systems
  - Linkers
  - Programmers
  - Booting an embedded CPU
  - Debuggers
    - JTAG
- Any weak link in the toolchain will hinder development
Today: Intro to Embedded C

- We are not learning C
- We are leaning “advanced embedded C”
  - Issues that frequently come up when developing embedded software
  - Seldom care about these when writing general-purpose apps
Embedded Compilers

- **Today:**
  - General capabilities
  - Specific issues part 1

- **First:** Almost all compilers for embedded systems are cross-compilers
  - Compiler runs on an architecture other than its target
  - Does this matter at all?
Compiler Requirements

◆ Be correct
  ➢ Embedded compilers are notoriously buggy
    ➢ Relatively few copies sold
    ➢ Diverse hardware impedes thorough testing

◆ Produce small, fast code
  ➢ Speed and size are conflicting goals
    ➢ Oops!
  ➢ Take advantage of platform-specific features

◆ Produce code that’s easy to debug
  ➢ Conflicts with optimization
  ➢ Whole-program optimization particularly problematic
Want To Tell the Compiler...

◆ There are only 32 KB of RAM
  ➢ Program must fit, but there’s no point reducing RAM consumption further

◆ There are only 256 KB of ROM
  ➢ Again: Program must fit but there’s no point reducing ROM consumption further

◆ Interrupt handler 7 is time critical
  ➢ So make it very fast, even if this bloats code

◆ Threads 8-13 are background threads
  ➢ Performance is unimportant so focus on reducing code size
What We Get To Tell It

◆ A few compiler flags:
  - -O2, -Os, Etc.
  - May or may not do what you want
  - Typically no flags for controlling RAM usage

◆ Therefore…
  - Meeting resource constraints is 100% your problem
  - Shouldn’t assume compiler did the right thing
  - Shouldn’t assume code you reuse does the right thing
    - Including the C library
  - Figure out which resources matter and focus on dealing with them
  - Changing or upgrading compiler mid-project is usually very bad
I have a 1982 book on 6502 assembly programming:

- `strcmp()`: compare two strings
  - Registers used: all
  - Execution time: \( 93 + 19 \times \text{length of shorter string} \)
  - Code size: 52 bytes
  - Data size:
    - 4 bytes on page 0
    - 4 bytes to hold the string pointers

Try to find this information for current C libraries!
Why use C?

◆ “Mid-level” language
  - Some high-level features
  - Good low-level control
  - Static types
  - Type system is easily subverted

◆ C is popular and well-understood
  - Plenty of good developers exist
  - Plenty of good compilers exist
  - Plenty of good books and web pages exist

◆ In many cases there’s no obviously superior language
Why not use C?

- Hard to write portable code
  - For example “int” does not have a fixed size

- Hard to write correct code
  - Very hard to tell when your code does something bad
  - E.g. out-of-bounds array reference
  - This is Microsoft’s major problem...

- Language standard is weak in some areas
  - Means there is plenty of diversity in implementations

- Linking model is unsafe

- Preprocessor is poorly designed
CPP – the C Preprocessor

- CPP runs as a separate pass before the compiler
- Basic usage:
  - `#define FOO 32`
  - `int y = FOO;`
- Compiler sees:
  - `int y = 32;`
- CPP operates by lexical substitution
- Important: The compiler never sees `FOO`
  - So of course the debugger, linker, etc. do not know about it either
Some Interesting Macros

#define PLUS_ONE(x) x+1
int a = PLUS_ONE(y)*3

#define TIMES_TWO(x) (x*2)
int a = TIMES_TWO(1+1)

#define MAX(x,y) ((x)>(y)?(x):(y))
void f () { int m = MAX(a++,b); }  
#define INT_SYMBOL int * INT_SYMBOL x, y;
Macro Problems

- Root of the problem:
  - C preprocessor is highly error-prone
  - Avoid it except to do very simple things
  - Fully parenthesize macro definitions
  - Make macro usage conventions clear

- Entertaining macros:

```
#define DISABLE_INTS asm volatile ("cli"); {
#define ENABLE_INTS asm volatile ("sei"); }
```

- Is this good or bad macro usage?
◆ Old conventional wisdom:
  ➢ Careful use of CPP is good

◆ New conventional wisdom:
  ➢ Most uses of CPP can be avoided
  ➢ Trust the optimizer
Macro Avoidance

- **Constants**
  - Instead of
    - `#define X 10`
  - Use
    - `const int X = 10;`

- **Functions**
  - Instead of
    - `#define INC_X x++`
  - Use
    - `inline void INC_X(void) { x++ }`
More Macro Avoidance

◆ Conditional compilation
  ➢ Instead of
    ➢ #if FOO ... #endif
  ➢ Use
    ➢ if (FOO) { ... }
  ➢ Instead of
    ➢ #ifdef X86 ... #endif
  ➢ Put x86 code into a separate file

◆ However: Design of C makes it impossible to avoid macros entirely
  ➢ C++ much better in this respect
Bit Manipulation without Macros

- Something like this is good:

```c
void set_bit (int *a, int bit) {
    *a |= (1<<bit);
}
void clear_bit (int *a, int bit) {
    *a &= ~(1<<bit);
}
```
CPP in Action

- Sometimes you need to look at the CPP output
  - That is, see what the C compiler really sees
  - There’s always a way to do this
  - In CodeWarrior, do this using the IDE
  - For gcc: “gcc –E foo.c”
Intrinsics

- “Intrinsic” functions are built in to the compiler
  - As opposed to living in a library somewhere
- Why do compilers support intrinsics?
  - Efficiency – can perform interesting optimizations
  - Ease of use
    - Compiler can add function calls where they do not exist in your code
    - Compiler can eliminate “library calls” in your code
- Need to be careful when compiler inserts function calls for you!
# Integer Division Intrinsics

- **On ARM7**

```assembly
sdiv:
  str   lr, [sp, #-4]!
  bl    __divsi3
  ldr   pc, [sp], #4
```

```c
int sdiv (int x, int y)
{
    return x/y;
}
```

- **On AVR**

```assembly
sdiv:
  rcall  __divmodhi4
  mov    r25,r23
  mov    r24,r22
  ret
```
Copy Intrinsic

```c
struct foo {
    int x, y[3];
    double z;
};

void struct_copy2 (struct foo *a, struct foo *b)
{
    *a = *b;
}
```

ColdFire code:

```
struct_copy2:
    link     a6,#0
    moveq    #6,d1
    move.w   (a1),(a0)
    move.w   2(a1),2(a0)
    addq.l   #4,a1
    addq.l   #4,a0
    subq.l   #1,d1
    bne.s    *-14
    unlink   a6
    rts
```
More Copy

- On ARM7

```assembly
struct_copy2:
    str    lr, [sp, #-4]!
    mov    lr, r1
    mov    ip, r0
    ldmia  lr!, {r0, r1, r2, r3}
    stmia  ip!, {r0, r1, r2, r3}
    ldmia  lr, {r0, r1}
    stmia  ip, {r0, r1}
    ldr    pc, [sp], #4
```
Copy on x86-64

➡️ From Intel CC (but copying a larger struct):

```assembly
struct_copy:
pushq     %rsi
movl      $4000, %edx
call      _intel_fast_mempcpy
popq      %rcx
ret
```
String Length

int len_hello1 (void)
{
    return strlen("hello");
}

◆ ColdFire code:

len_hello1:
0x00000000 link a6,#0
0x00000004 lea @_@71,a0
0x0000000A jsr _strlen
0x00000010 unlk a6
0x00000012 rts
Another String Length

- ARM7

```
len_hello1:
    mov r0, #5
    bx lr
```
So What?

- Compiler can add function calls where you didn’t have one
- Compiler can take out function calls that you put in

- How will you understand the resource usage of the resulting code?
  - What resources are we even talking about?
30-Second Interrupt Review

- Interrupts are a kind of asynchronous exception
- When some external condition becomes true, CPU jumps to the interrupt vector
- When an interrupt returns, previously executing code resumes as if nothing happened
  - Unless the interrupt handler is buggy
  - Also, the state of memory and/or devices has probably changed
- With appropriate compiler support, interrupts look just like regular functions
  - Don’t be fooled – there are major differences between interrupts and functions
void __attribute__ ((interrupt("IRQ")))
tc0_cmp (void);
{
    timeval++;
    VICVectAddr = 0;
}

- All embedded compilers provide similar extensions
- C language has no support for interrupts
Example CF Interrupt

◆ You write:

```c
__declspec(interrupt) void rtc_handler(void) {
    MCF_GPIO_PORTTC ^= 0xf;
}
```

◆ After CPP:

```c
__declspec(interrupt) void rtc_handler(void) {
    (*(vuint8 *) (0x4010000F)) ^= 0xf;
}
```
Assembly for CF Interrupt

rtc_handler:
    strldsr    #0x2700
    link      a6,#0
    lea       -16(a7),a7
    movem.l   d0-d1/a0,4(a7)
    movea.l   #1074790415,a0
    moveq     #0,d1
    move.b    (a0),d1
    moveq     #15,d0
    eor.l     d0,d1
    move.b    d1,(a0)
    movem.l   4(a7),d0-d1/a0
    unlk      a6
    addq.l    #4,a7
    rte
Inline Assembly

- Two reasons to add assembly into a C program:
  1. Need to say something that can’t be said in C
  2. Need higher performance than the C compiler provides

- In both cases
  - Write most of a function in C and then throw in a few instructions of assembly where needed
  - Let the compiler do the grunt work of respecting the calling convention

- When writing asm to increase performance:
  - Be absolutely sure you identified the culprit
  - First try to write faster C
long square (short a) {
    long result=0;
    asm {
        move.w a,d0 // fetch function argument ‘a’
        mulu.w d0,d0 // multiply
        move.l d0,result // store in local ‘result’
    }
    return result;
}

- Compiler generates glue code integrating the assembler and C code
- What if it can’t?
Inline Assembly Example

square:

  link      a6,#0
  subq.l   #8,a7
  move.w   d0,-8(a6)
  clr.l    -6(a6)
  move.w   -8(a6),d0
  mulu.w   d0,d0
  move.l   d0,-6(a6)
  move.l   -6(a6),d0
  unlk      a6
  rts
GCC Inline Assembly

Formatting:

```c
asm volatile (code : outputs : inputs : clobbers );
```
- **Code** – instructions
- **Outputs** – maps results of instructions into C variables
- **Inputs** – maps C variables to inputs of instructions
- **Clobbers** – tells the compiler to forget the contents of registers that were invalidated by the assembly code

- This syntax is much more difficult to use than CodeWarrior’s!
Important From Today

- Embedded C
  - Pros and cons
- Macros and how to avoid them
- Intrinsics
- Interrupt syntax
- Inline assembly