Robotics Society of Southern California

The Robot Builder

Volume Twelve Number One January 2000

Notices

· Practical Mobile Robotics Class - 11:00AM - 12:00PM

· Business Meeting - 12:30 - 1:00

· General Meeting - 1:00 - 3:00

Distribution

If you would like to receive The Robot Builder via e-mail, contact the editor at:

apendragn@earthlink.net

Inside this Issue

Crusoe: Improved Microprocessors for Mobile Applications

By Arthur Ed LeBouthillier

Mobile computing applications have different demands than in the desktop realm. Because a desktop computer is fed from a wall socket, it can be designed to use large amounts of power as if it were an infinite resource. Desktop computers can also be built to operate with IC's that generate lots of heat because designers can add large fans. In fact, some recent desktop computers have two fans inside: one for the power supply and one fan dedicated to cooling off the processor. Mobile computers, however, do not have the luxuries that desktop machines have. They must be efficient in their power use because they usually run from batteries and they can't generate too much heat because to be portable, they must be small. Small handheld or laptop computers are generally not designed with fans yet they cannot get so hot that they damage components or become uncomfortable for their users. Mobile robots fall under the category of mobile computing and it is worth examining new developments for mobile computing that might show their way in mobile robots. One very recent development in mobile processors has been the Crusoe microprocessor by Transmeta. It offers a glimpse into mobile current computing trends.

The Crusoe Family of Mobile Processors

In the past few years, we have seen a trend towards reduced instruction set computers (RISC). These processors use simple instructions which run in one computer clock cycle.

Transmeta is a new microprocessor company which has entered the mobile processor market. In a recently much-hyped product release, they announced a new family of microprocessors specifically targeted for mobile computing applications. This family evidences a wide array of newer computing technologies packaged together to create lower-power demands in a very fast processor. The unique features of these processors are worth examining. First off, the Crusoe family of processors implements a kind of parallelism. Second, the Crusoe family of processors is designed to be low power, making them more suitable for mobile processing applications. Third, the Crusoe family of processors is meant to execute the x86 instruction set as used Pentium-class processors.

Crusoe Parallelism

The Crusoe family of microprocessors breaks the trend in using RISC processors. Rather than using simple, RISC-like features as we have seen among a number of current processors, the Crusoe family of processors utilizes Very Long Instruction Word (VLIW) techniques. VLIW processors have large instruction words which contain a number of smaller instructions which can be made to run in parallel. A typical Crusoe instruction consists of four 32-bit instruction words which run in parallel. The advantage of this is obvious: four operations can be performed simultaneously.

See ***Crusoe***, Page 2

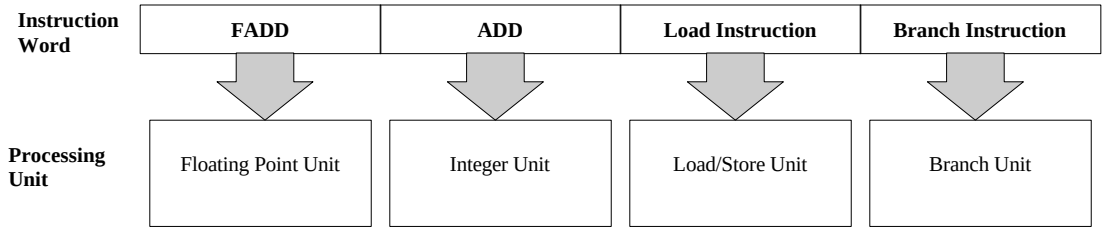

Figure 1 - How Crusoe implements parallelism

A single 128-bit multi-instruction is known as a molecule and it consists of four atomic instructions. The 128-bit instruction's four 32-bit words are each instructions for different processing elements within the processor: the Floating Point Unit (FPU) which performs math operations on decimal numbers, the Integer Unit which performs math operations on integers, the Load/Store Unit which takes information to or from memory and the Branch Unit which controls program flow. Therefore, at one moment, a Crusoe processor can add a floating point number, add integers, save something to memory and jump to another location in the program.

Crusoe's Low Power Technology

The Crusoe processor utilizes several techniques to lower power consumption substantially from that of Pentium-class machines. The most significant way that this is done is by not trying to implement the Pentium design at all. The structure and design of the Crusoe processor are actually very different from that of the Pentium's. Crusoe processors have been designed from the beginning to be extremely power efficient by having a very low number of transistors in their design. A similar-powered Crusoe processor has about half of the die area of the equivalent Pentium processor.

Another power-saving technique used is that processor speed and voltages are adjusted dynamically to give optimum power usage at any instant. Since power demands are linear with respect to processor speed and by the square of the voltage

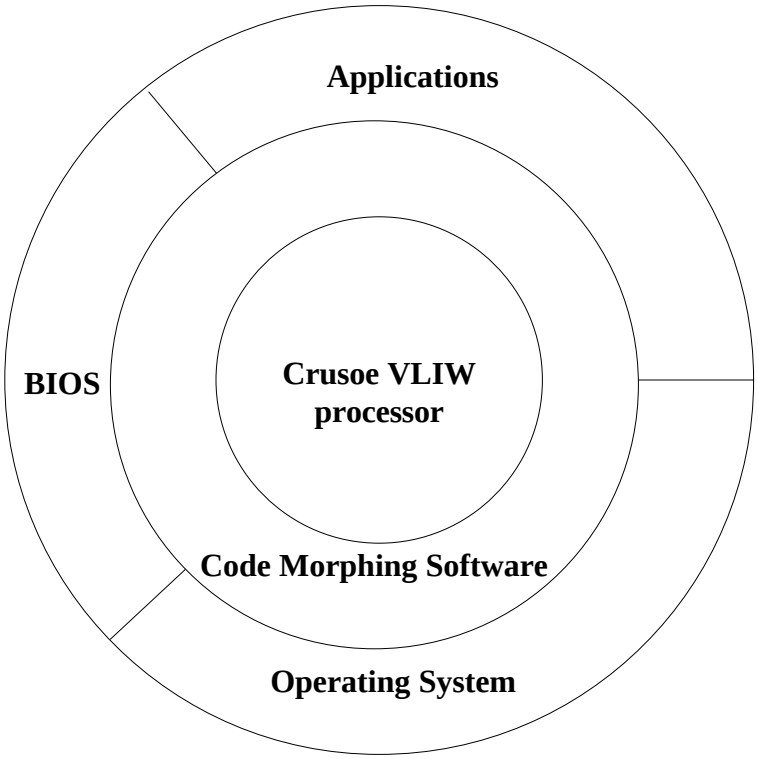

Figure 2 - Code Morphing sits between the BIOS, operating system and applications and the hardware

used, there can be a cubic reduction in power by selecting the proper voltage and processor speed.

Implementing Pentium Instructions

Rather than trying to duplicate all of the instructions of the Pentium processor, the Crusoe chips utilize an ingenious software technique called Code Morphing to implement a Pentium in software. Code Morphing provides a dynamic run-time compiler which converts x86 instructions into Crusoe instructions on the fly. This would normally be an extremely slow operation of analyzing and recompiling a program on the fly, but the Crusoe processors save the compiled code and operate from a different code storage area. Therefore, when an x86 program is run, each block is compiled into Crusoe instructions as it is called. Once a block of code has been compiled, the Crusoe processor can use that pre-compiled block in future calls to that code section. This means that once a code section has been compiled, it can be run at full Crusoe speed without recompiling.

This ingenious technique allows a Crusoe processor to run x86 code at high speed but without actually implementing all hardware of a Pentium. Of course, there is a penalty for translating the x86 instructions the first time, but once compiled, the program runs at competitive rates but with much less power. Based on benchmarks provided by Transmeta, it appears as if a Pentium-class Crusoe runs with about ¼ the power and heat.

Trends Evident in the Crusoe

By itself, the Crusoe is a nice processor for embedded applications. It also seems to illustrate a trend in modern processor design. The first aspect of this trend is the parallelism evidenced by the VLIW design. Several processors have used techniques similar to this. The Pentium itself breaks an x86 instruction into a number of micro-operations that can run in parallel. Texas Instrument's C6000 DSP used the VLIW technique allowing it to perform substantially faster than other DSP's. Now, the Crusoe is using VLIW techniques to allow more parallelism.

Another technology trend that seems obvious is the importance of smart compiler technology. A number of DSP companies have begun producing their own compilers rather than letting third-party developers do it because they are stressing compiler technology as vital to exploiting the full capabilities of their processors. Java was also part of this trend by prompting a desire for JIT (just-in-time) compilers which converted Java code into machine code at run time. Now with the Crusoe processors, we're seeing a smart compiler performing the function of code conversion and optimization in order to replace hardware. If anything, the Crusoe processors could not enter into the x86-compatible market without the existence of the run-time compiling used in Code Morphing.

Finally, one other trend that seems to be obvious is recognized by the term "post-PC era." We are now entering an era where powerful portable computers are becoming very important. This contrasts with the last era where desktop computers reigned without peer and the era prior to that where mainframes were the most important computing environment. In the post-PC environment, capable mobile computers will become more widely used. The arrival of processors like the Crusoe signals this trend where capable, low-power mobile processors are in demand. Of course this demand did not immediately arrive, but now a whole company has arrived into the PC class processor market whose sole-emphasis has been on getting lower-power with higher processor speed.

Basic C Programming (Part 2)
By Arthur Ed LeBouthillier

Last month, we reviewed two important ideas about the C language: that C is a typed language and that C is a functional language. This month, let's look at C types.

Table 1 shows the variety of C types available in almost any implementation. It is important to realize that not all implementations of C will have all of these types. Additionally different implementations may have different bit sizes assigned for each of the types.

As you can see, types ***char***, ***unsigned char***, ***short***, ***unsigned short***, ***int***, ***unsigned int***, ***long*** and ***unsigned long*** can represent only whole numbers (numbers without a decimal point). Types ***float*** and ***double*** can be used to represent decimal numbers quite accurately. Type void refers to an unspecified type and is often used to specify that no value is returned from a function; it also has other important uses. Notice that each of the types has limits on the range that it can represent; you must select the proper type to represent the number you want.

Variables

A variable is a named place to store a value of a specific type. If you think of those hotel mailboxes which are rows and columns of boxes with a name on each box, that is kind of the basic idea. With these, you could put a name on them and put in things like letters. Small mailboxes can only accept letters and bigger mailboxes can store packages. This is similar to the idea of variables: they have a name and they can store certain kinds of things. Let's assume we wanted to create an integer variable able to hold a number between -32768 and 32767. A type ***int*** would do the job; this is how we would create an integer variable a in C:

```
int a;
```

Notice the word ***int*** followed by the name of the variable, ***a***, and finally we end the statement with a semicolon.

Now that we have created the variable, or ***declared*** it, we can give it a value:

```
a = -5;
```

The process of giving a variable a value is called ***assigning*** a value to it. C also allows us to perform mathematical operations. We could add a couple of numbers together and then assign them to our

See *Basic C*, Page 5

Type	Size in bits	Minimum	Maximum
char	8	-128	127
unsigned char	8	0	255
short	8	-128	127
unsigned short	8	0	255
int	16	-32768	32767
unsigned int	16	0	65535
long	32	-2,147,483,648	2,147,483,647
unsigned long	32	0	4,294,967,295
float	32	-3.40282 X 10 ^ 38	3.40282 X 10 ^ 38
double	64	-1.79769 X 10 ^ 308	1.79769 X 10 ^ 308
pointer or void	system dependent	?	?

Table 1 - basic C types

recently declared variable like this:

```
a = 5 + 3;
```

You can also use variables in mathematical formulas like this:

```
int a;
int b;

b = 3;
a = b +5;
```

In this example, we declared two variables, ***a*** and ***b***, assigned the number **3** to ***b*** and then assigned ***b*** plus ***5*** to ***a***. Obviously, after doing these operations, ***b*** would contain the value **3** and ***a*** would contain the value **8**.

We could also declare other types, such as ***char***'s which could accept characters. It is worth mentioning how C stores characters. A character is stored as an 8-bit whole number which takes the value according to what is known as ASCII. ASCII is a code system which assigns a number to each of the letters, numbers and other characters used. We could declare a variable ***c*** and assign it a character ***a*** in this manner:

```
char c;

c = 'a';
```

A character is denoted by putting the character in single quotes as above. We could declare a floating point variable ***f*** and assign it a value like this:

```
float f;

f=3.9;
```

The most important thing to realize about C is that you MUST declare a variable before you use.

Simple Programs

We have talked about the idea that C is a functional language meaning that you write a program by declaring functions. Let's look at what a simple program might look like.

All C programs consist of a function called ***main*** which calls the other functions you define. Therefore, the simplest program consists merely of a ***main*** function. Here is one of the most common simple programs:

```
#include "stdio.h"
void main()
{
        printf("Hello world\n");
}
```

This program has only one function, main, and inside that function is only one statement:

```
printf("Hello world\n");
```

The result of running this program is that the line:

Hello world

would appear on the computer screen.

The first part of this program is the line:

```
#include "stdio.h"
```

This statement, known as an include statement, tells C that you want to use previously defined functions which are declared in the file ***stdio.h***. C allows you to put parts of your program in many different files so that you can reuse often used functions. From a programming standpoint, this allows modularization of your code because you can create files which can be used in a bunch of different programs. ***Stdio.h*** is a standard input/output ***library*** which defines a whole host of functions which you can use for input and output. ***printf*** is a function which is used to print things on the screen.

After we included the functions from the ***stdio.h*** file, we then declared the ***main*** function. We did this by declaring its type, ***void***, followed by the functions name, ***main***, followed by empty

See *Basic C*, Page 6

parenthesis pairs followed by what is known as a ***block***. A block is a sequence composed of a left brace, ***{***, with a number of statements separated by semicolons, followed by a right brace, ***}***. The general rule for declaring a function is:

type name (arguments) block

This means you must declare the type of value returned by the function (or void if there is no return value), followed by the name of the function, followed by a left parenthesis, followed by a declaration for the function's arguments, followed by a right parenthesis, followed by a block. As we said, a block is a left brace, followed by a number of statements, separated by semicolons, followed by a right brace. Taking all of this complicated description of a function declaration, we derive the main function:

```
void main ()
{
        printf("Hello world\n");
}
```

The function's block contains only one statement in it, ***printf("Hello world\n")***. There were no arguments into the function so there is nothing between the parentheses.

We could have done other things within the main function. A different program using the variable declarations above is:

```
#include "stdio.h"

void main()
{
                int a;
                int b;

                a = 3;
                b = a + 10;

                printf("a = %d\n", a);
                printf("b = %d\n", b);
}
```

This program creates variables ***a***, and ***b***, stores some numbers in each of them and then prints their values to the screen.

We could have had our main function call another function that we declared like this:

```
#include "stdio.h"

int sum(int a, int b)
{
        return a + b;
}

void main()
{
        int c;
        int d;
        int e;

        c = 5;
        d = 6;
        e = sum( c, d );
        printf("e = %d\n",e);
}
```

This program declares two functions: ***sum*** which returns an integer type and ***main*** which returns no value. ***Sum*** takes two arguments, ***a*** and ***b*** which are of type ***int***, adds them together and returns their sum as a type ***int***. ***Main*** declares three variables, ***c***, ***d*** and ***e***, assigns values to ***c*** and ***d*** and then calls the function ***sum*** with their values. The result of the ***sum*** function is assigned to ***e*** and then this result is printed on the screen. Although somewhat complicated, this is representative of most programs you'll write: you define one or more auxiliary functions which are called from main. Together, these functions perform the operations you want your program to perform.

Summary

We looked at the types of variables one can declare in C. We also looked at a few complete programs in C that print results on the screen. There's a lot more to learn and we'll look at some new more complicated functions next month.

Robotics Society of Southern California

President	Arthur Ed LeBouthillier
Vice President	Henry Arnold
Secretary	Randy Eubanks
Treasurer	Henry Arnold
Past President	Randy Eubanks
Member-at-Large	Tom Carrol
Member-at-Large	Pete Cresswell
Member-at-Large	Jerry Burton
Faire Coordinator	Joe McCord
Newsletter Editor	Arthur Ed LeBouthillier

The Robot Builder (TRB) is published monthly by the Robotics Society of Southern California. Membership in the Society is $20.00 per annum and includes a subscription to this newsletter.

Membership applications should be directed to:

Robotics Society of Southern California
Post Office Box 26044
Santa Ana, CA 92799-6044

Manuscripts, drawings and other materials submitted for publication that are to be returned must be accompanied by a stamped, self-addressed envelope or container. However, RSSC is not responsible for unsolicited material.

We accept a wide variety of electronic formats but if you are not sure, submit material in ascii or on paper. Electronic copy should be sent to:

apendragn@earthlink.net

Arthur Ed LeBouthillier - editor

The Robotics Society of Southern California was founded in 1989 as a non-profit experimental robotics group. The goal was to establish a cooperative association among related industries, educational institutions, professionals and particularly robot enthusiasts. Membership in the society is open to all with an interest in this exciting field.

The primary goal of the society is to promote public awareness of the field of experimental robotics and encourage the development of personal and home based robots.

We meet the 2nd Saturday of each month at California State University at Fullerton in the electrical engineering building room EE321, from 12:30 until 3:00.

The RSSC publishes this monthly newsletter, The Robot Builder, that discusses various Society activities, robot construction projects, and other information of interest to its members.

Membership/Renewal Application

Name _____

Address _____

City _____

Home Phone ( ) - Work Phone ( ) -

Annual Membership Dues: ($20) Check #
(includes subscription to The Robot Builder)

Return to: RSSC
POB 26044
Santa Ana CA 92799-6044

How did you hear about RSSC? _____

RSSC
POB 26044
Santa Ana CA 92799-6044

Please check your address label to be sure your subscription will not expire!