BASM for Beginners
Lessons

Dennis Christensen

From news://forums.borland.com

borland.public.Delphi.languages.basm

BASM 初学者入门

翻译 SkyJacker(HeMiaoYu@gmail.com)

http://bbs.cnpack.org

http://www.cnpack.org

© Dennis Chistensen, 2003

目 录

Introduction to BASM for Beginners BASM 入门教程简介.....2
Lesson 1 第 1 课(整数运算).....4
Lesson 2 第 2 课(符点数运算).....8
Lesson 3 第 3 课(64 位运算,MMX,SSE2 指令).....19
Lesson 4 第 4 课(分支语句if-else).....26
Lesson 5 第 5 课(循环语句).....37
Lesion 6 第 6 课(CharPos).....60
Lesson 7 第 7 课(高级符点数运算).....92
译者注.....142

Introduction to BASM for Beginners BASM 入门教程简介

The series of articles named “BASM for beginners” currently consists of 7 articles and no. 8 and 9 are in progress. Common for the articles, and coming articles, is that they explain some BASM issues by use of an example function. Most often this function is first implemented in Pascal and then the compiler generated assembler code is copied from the CPU view in Delphi and then analyzed and optimized. Sometimes optimization involves the usage of MMX, SSE or SSE2 instructions.

“BASM 入门教程”系列文章当前包含 7 课。第 8 课和第 9 课正在完成中。（译者注：从已获得的资料看，第 8，9 早在 2003 年已经完成，第 8，9 主要介绍了 64 位数据操作与 MMX 指令，暂不打算翻译）。
这些文章通过函数实例来介绍 BASM。通常，这些函数首先使用 Pascal 编写，然后从 Delphi 编译器的 CPU 窗口中复制汇编代码，之后进行分析和优化。也包括使用 MMX, SSE 或 SSE2 指令的优化。

By taking the code made by the compiler from a Pascal function the most commonly used instructions from the big 32-bit Intel Architecture instruction set are introduced to the beginner first. Seeing which code the compiler generates is leading to a valuable insight in the effectiveness of compiler generated code in general and into the Delphi compiler specifically.

通过借用编译器编译 Pascal 函数产生的汇编代码，为初学者介绍了大 32 位 Intel 体系结构指令集。研究编译器产生的代码将使我们更清楚编译器产生的代码的效率，同时有助于了解 Delphi 编译器的特点。

As specific assembly code optimizations are introduced generalizations will be introduced when suitable. These general optimizations are suitable for implementation in compilers and most compilers including Delphi have them. At

some point in the future a tool that automatically optimizes assembler code will be developed.

本教程将介绍一些通用的汇编代码优化技术。这些通用的优化技术已经被大部分编译器使用，包括 Delphi。在某些方面，一些汇编代码自动优化工具也将被开发。

Knowledge about the target processor is often needed when optimizing code and therefore are a lot of CPU details, such as pipelines, explained in the series too.As far as I know there is only little literature available that explains all these issues on a level where beginners can follow it. I hope this series will help fill this void.

优化代码时经常会用到相关处理器特性信息等资料，比如管道线技术，也将在此系列教程中介绍。据我所知,只有很少的文献资料是针对初学者来解释这些问题的，我希望这些文章能够填补这方面的空白。

Best regards
Dennis Kjaer Christensen.

Lesson 1 第 1 课(整数运算)

The first little example gets us started. It is a simple function in Pascal with multiplies an integer with the constant 2.

用一个小例子开始我们的 BASM 之旅。这是一个用常量 2 来乘以一个整数的小函数。

```
function MulInt2(I : Integer) : Integer;
begin
  Result := I * 2;
end;
```

Lets steal the BASM from the CPU view. I compiled with optimizations turned on.
从 CPU VIEW 中获取汇编代码。编译时打开了优化选项。

```
function MulInt2_BASM(I : Integer) : Integer;
begin
  Result := I * 2;
  {
  add eax,eax
  ret
  }
end;
```

From this we see that I am transferred to the function in eax and that the result is transferred back to the caller in eax too. This is the convention for the register calling convention, which is the default in Delphi. The actual code is very simple, the times 2 multiplication is obtained by adding I to itself, I+I = 2I. The ret instruction returns execution to the line after the one which called the function.

从上面可以看出参数值从 eax 传入，并从 eax 返回。这是 Delphi 默认的调用约定。
实际代码非常简单，就是某数乘以 2 等于其自身相加，I + I = 2I。
Ret 指令返回到调用此函数处的下一条指令位置。

Lets code the function as a pure asm function.
其纯汇编函数如下：

```
function MulInt2_BASM2(I : Integer) : Integer;
asm
  //Result := I * 2;
  add eax,eax
  //ret
```

end;
Observe that the ret function is supplied by the inline assembler.
显然，ret 由内嵌汇编器本身提供。

Let us take a look at the calling code.This is the Pascal code
让我们看看调用此函数的代码。Pascal 代码如下：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I, J : Integer;
begin
  I := StrToInt(IEdit.Text);
  J := MulInt2_BASM2(I);
  JEdit.Text := IntToStr(J);
end;
```

The important line is
重要的行是：

```
J := MulInt2_BASM2(I);
```

From the cpu view
在 CpuView 中可以看到

```
call StrToInt
call MulInt2_BASM2
mov esi,eax
```

After the call to StrToInt from the line before the one, which calls our function, I am in eax. (StrToInt is also following the register calling convention). MulInt2_BASM2 is called and returns the result in eax, which is copied, to esi in the next line.

在调用函数 MulInt2_BASM2 之前首先调用了 StrToInt，StrToInt 返回结果 I 存放在 eax 中(StrToInt 也遵循寄存存器调用约定)。MulInt2_BASM2 调用后，结果存入 eax，并且被下一行代码复制到 esi 中。

Optimization issues: Multiplication by 2 can be done in two more ways. Use the mul instruction or shifting left by one. In the Intel IA32 SW developers manual 2 page 536 mul is described. It multiplies the value in eax by another register and the result is returned in the register pair edx:eax. A register pair is needed because a multiplication of two 32 bit numbers results in a 64 bit result, just like 9*9=81 - two one digit numbers (can) result in a two digit result.

优化方式：被 2 乘有多种方法。使用 mul 乘法指令或者左移一位都可以完成此功能。在 IA32 软件开发手册 2 的第 536 页介绍了 mul。被乘数存放在 Eax，乘数存放在其他的寄存器，结果存入寄存器对 edx: eax 中。使用寄存器对的原因是两个 32 位数相乘结果是 64 位，就像 9*9=81，两个一位数相乘得到了一个两位数。

This raises the issue of which registers must be preserved by a function and which can be used freely. This is explained in the Delphi help."An asm statement must preserve the EDI, ESI, ESP, EBP, and EBX registers, but can freely modify the EAX, ECX, and EDX registers."

在这种情况下，必须保存寄存器的原始值后才能使用。在 Delphi Help 中有解释：“在汇编语法中，必须保存 EDI, ESI, ESP, EBP, EBX 寄存器，但是 EAX, ECX, EDX 可以自由使用。“

We can conclude that it is no problem that edx is modified by the mul instruction and our function can also be implemented like this.

因此，我们可以确定乘法指令 mul 修改 edx 不会有任何问题。我们的函数也可以修改为如下：

```
function MulInt2_BASM3(I : Integer) : Integer;
asm
  //Result := I * 2;
  mov ecx, 2
  mul ecx
end;
```

ecx is used also but this is also ok. As long as the result is less than the range of integer it is returned correctly in eax. If I am bigger than half the range of integer overflow will occur and the result is incorrect.

Ecx 也可以被任意使用。只要结果值在 Integer 范围内，eax 存放的就是正确的结果值。如果 I 大于整数的一半，那么相乘后结果会溢出，eax 值也就不准确了。

Implementation with shift
使用移位指令

```
function MulInt2_BASM4(I : Integer) : Integer;
asm
  //Result := I * 2;
  shl eax,1
end;
```

Timing can reveal which implementation is fastest. We can also consult Intel or AMD documents with latency and throughput tables. Add & mov is 0.5 cycles latency and throughput, mul is 14-18 cycles latency and 5 cycles throughput. shl is 4

cycles latency and 1 cycle throughput. The version chosen by Delphi is the most efficient on P4 and this will probably also be the case on Athlon and P3.

这个执行在时间上是高效的。我们也可以参考关于介绍潜伏期和吞吐量的 Intel 或 AMD 文档。Add 或 Mov 指令是 0.5 个时钟周期和吞吐量，Mul 是 14~18 个周期和 5 个吞吐量，Shl 是 4 个时钟周期和 1 个吞吐量。这个版本的 Delphi 代码在 P4 上是最有效的，在 Athlon 和 P3 也可能是。

Issues not covered: mul versus imul and range checking, other calling conventions, benchmarking, clock count on other processors, clock count for call + ret, location of return address for ret etc..

这个版本没有考虑以下情况：mul 与 imul 的不同，范围检查，其他调用约定，基准，时钟周期，调用与返回指令的时钟周期，返回本地地址 ret 等等。

译者附：

1、Cpu 扩展指令全称
MMX(Multi Media eXtension，多媒体扩展指令集)，
SSE(Streaming SIMD Extensions，单指令多数据流扩展)
SSE2(Streaming SIMD Extensions 2，Intel 官方称为 SIMD 流技术扩展 2 或数据流单指令多数据扩展指令集 2)。

2、Latency and throughput
http://www.yesky.com/120/1645120.shtml

Latency：潜伏期，从字面上了解其含义是比较困难的，实际上，它表示完全执行一个指令所需的时钟周期，潜伏期越少越好。严格来说，潜伏期包括一个指令从接收到发送的全过程现今的大多数 x86 指令都需要约 5 个时钟周期，但这些周期之中有部分是与其它指令交迭在一起的（并行处理），因此 CPU 制造商宣传的潜伏期要比实际的时间长。

Throughput：吞吐量，它包括两种含义：
第一种：执行一条指令所需的最少时钟周期数，越少越好。执行的速度越快，下一条指令和它抢占资源的机率也越少。
第二种：在一定时间内可以执行的最多指令数，当然是越大越好。

3、第 1 课其实由 1A, 1B, 1C 构成。这属于 1A。

Lesson 2 第 2 课(符点数运算)

This is the second chapter of the introduction to BASM programming with Delphi. The first chapter was a short introduction to integer code and this second one is about floating point code.Our example functions can evaluate a second order polynomial. The parameters A, B and C that defines the polynomial is coded as local constants. Input to the function is the variable X of type double and the result is also of type double. The function looks like this.

这是 Delphi BASM 的第二章。第一章简单的介绍了整数指令，第二章将介绍关于浮点数指令。我们的例子是求一个 2 次多项式，常数 A, B, C 是多项式的系数。函数的参数是浮点数 X，其返回值也是浮点数。函数如下所示：

```
function SecondOrderPolynomial1(X : Double) : Double;
const
  A : Double = 1;
  B : Double = 2;
  C : Double = 3;
begin
  Result := A*X*X + B*X + C;
end;
```

Copying the assembler code from the CPU view gives us this.
复制 CPU view 中的汇编代码如下：

```
function SecondOrderPolynomial2(X : Double) : Double;
const
  A : Double = 1;
  B : Double = 2;
  C : Double = 3;
begin
  {
  push    ebp
  mov      ebp,esp
  add      esp,-$08
  }
  Result := A*X*X + B*X + C;
  {
  fld      qword ptr [A]
  fmul    qword ptr [ebp+$08]
  fmul    qword ptr [ebp+$08]
  fld      qword ptr [B]
```

```
  fmul    qword ptr [ebp+$08]
  faddp st(1)
  fadd    qword ptr [C]
  fstp    qword ptr [ebp-$08]
  wait
  fld     qword ptr [ebp-$08]
  }
  {
  pop     ecx
  pop     ecx
  pop     ebp
  }
end;
```

Lets explain the asm code line by line.
接下来一行行来解释代码.

The begin results in this code.
begin 指令产生下面的代码

```
begin
  {
  push    ebp
  mov     ebp,esp
  add     esp,-$08
  }
```

which sets up a stack frame for the function.
A stack frame is just a piece of memory that is reserved for the stack.A stack frame is accessed through two pointers, the base pointer and the stack pointer. The base pointer is in ebp and the stack pointer is in esp. These two registers are reserved for use by these pointers only.
The line push ebp backsup the base pointer. The line mov ebp, esp sets up a new base pointer,which is pointing to the top of the stack. The line add esp, -$08 moves the stack pointer 8 bytes down.As a curiosity the stack grows downward and the last line could more intuitively have been sub esp,8.The new stack frame that was created by these three lines is standing on top of, or actually hanging under, the last stack frame, which was probably allocated by the function that called our SecondOrderPolynomial function.

这部分代码是用来设置函数的堆栈。
堆栈是内存中保留的一块区域，堆栈是通过两个指针来访问的，基址指针和栈顶指针。
基址指针存放在 ebp 中，堆栈指针存放在 esp 中。这两个寄存器是专门用来访问堆栈的。
Push esb 保存 ebp 指针，
Mov ebp, esp 将基址指针指向当前的栈顶,

add esp, -$08 将栈顶指针往下移 8 个字节。
堆栈的增长方向是往下的，最后一行 sub esp, 8 直观的表示了这个意思。
新的堆栈空间被最顶部的这 3 行代码创建。实际上是在调用 SecondOrderPolynomial 之后执行。

The next line of Pascal was compiled into no less than 9 lines of ASM.
下一行 pascal 代码被编译成至少 9 行的 ASM 代码。

```
Result := A*X*X + B*X + C;
 {
 fld     qword ptr [A]            // 将 A 装入 st(0)
 fmul    qword ptr [ebp+$08] // st(0) * X ->st(0)
 fmul    qword ptr [ebp+$08] // st(0) * X ->st(0) 完成 A * X * X
 fld     qword ptr [B]          // st(0)不为空，因此执行一次入栈操作：将 st(0)->st(1)
                                                        //      B -> st(0)。St(0) 总是为栈顶。
 fmul    qword ptr [ebp+$08] // st(0) * X ->st(0) = B * X
 faddp st(1)                        // st(1) + st(0) -> st(1)，执行出栈，因此 st(1)->st(0)
 fadd    qword ptr [C]            // st(0) + C -> st(0)
 fstp    qword ptr [ebp-$08] // 结果 st(0) 存入 [ebp-$08]。
 wait                          // 同步 FPU 与 CPU：停止 CPU 的运行，直到 FPU 完成当前操作
码
 fld     qword ptr [ebp-$08] // 结果存入 st(0)
}
```

For those that is used to HP calculators floating point code is very easy to understand. The first line,fld qword ptr [A], loads the constant A onto the floating-point register stack. The line,fmul qword ptr [ebp+$08], multiplies A with X. This makes sense by watching the Pascal code,but what means "qword ptr [ebp+$08]". qword ptr says "pointer to a quad word,which is the size of a double. (64 bit).

这些浮点数指令很容易理解。第一行 fld qword ptr [A]，装载常量 A 到浮点数寄存器堆栈。这行 fmul qword ptr [ebp+$08]，用 X 乘 A。这些可以通过 pascal 代码直接理解其含义,但是"qword ptr [ebp+$08]"表示什么呢。qword ptr 表示指向一个四个字的指针，它是一个浮点数的大小(64 位)。

The value of the pointer is between the brackets in [ebp+$08].
ebp is the base pointer and $08 is - well just 8.
Because the stack grows down the memory location 8 bytes above the base pointer is in the previous stack frame.Here X was placed by the function, which called our function.The register calling convention decides this placement of a double variable.

指针的值是括号之间的数值 ebp+$8。

ebp 是基址，偏移量是 8。因为在上一个堆栈中，堆栈从基址向下增长了 8 个字节。
上一层调用函数把 X 放在这里。
寄存器调用约定决定了一个 Double 变量的位置。

A double variable does not fit into the 32 bit integer registers,but it fits perfectly onto the floating-point registers. Borland decided to pass double variables via the stack,but passing them in floating point registers would have been more efficient.The next 3 lines need no further explanation, but the line, faddp st(1), needs some.All floating-point instructions starts with an f. add is addition. st(1) is floating point register 1,which is the second because st(0) is the first!

一个浮点数变量用 32 位的整型寄存器装不下，但是它正好可以装入浮点数寄存器。
Borland 通过堆栈传输 Double 变量，其实通过浮点数寄存器传输它们应该更有效率。
接下来的三行不需要更多的解释，只有这一行 faddp st(1) 需要说明一下。所有的浮点数指令都用一个 f 开头。 add 表示加法。 st(1) 是 1 号浮点数，但是它是第二个寄存器，因为 st(0) 是第一个！

The floating point registers are combined into a stack and instructions implicitly works on the top of the stack, which is st(0).faddp st(1) is the same as faddp st(0), st(1) and it adds register st(0) to register st(1) and place the result in st(1).The p in faddp means pop st(0) of the stack. This way the result ends up in st(0).The line fadd qword ptr [C] completes the calculations and the only thing left is to place the result in st(0).
It is actually already there and the two lines are redundant.

浮点数寄存器被组合为一个堆栈，指令隐含在栈顶 st(0) 工作。
faddp st(1) 与 faddp st(1), st(0) 相同，它将寄存器 st(0) 加到寄存器 st(1)，结果在 st(1) 中。faddp 的 p 表示将 st(0) 弹出堆栈。这样结果就从 st(1) 移到 st(0) 中。
fadd qword ptr [C] 完成计算，并将结果存入 st(0)。
因为结果已经在 st(0) 中了，因此这两行是冗余的：

```
fstp    qword ptr [ebp-$08]
fld     qword ptr [ebp-$08]
```

They just copy the result into the stack frame and loads it back in again.
Such a waste of precious time and energy :-).
The line wait makes sure that any exceptions that might have been raised by one of the floating-point instructions are checked. See Intel SW Developers Manual Volume 2 page 822 for the full explanation.

它们是将 st(0) 复制到堆栈，然后再从堆栈装回 st(0)。
居然如此浪费珍贵的时间和精力 :)。
wait 指令检查浮点数指令是否产生的意外。可以参考 Intel 软件开发手册卷 2 第 822

页的详细解释。

Then there are only three lines of asm back to explain.
这三行返回代码需要解释一下：

```
 {
 pop    ecx
 pop    ecx
 pop    ebp
 }
end;
```

These are removing the stack frame, by restoring the values of esp and ebp back to the values they had when the function was entered. This code is much more intuitive and does the same thing

```
add esp, 4
pop ebp
```

it is also more effective and I do not know why the compiler is incrementing the stack pointer in this cumbersome way. Remember that ecx can be used for free and assigning values to it is just like pouring them into a waste bucket.

它们弹出堆栈，恢复进入函数时保存的 esp、ebp 的值。更形象的表示如下

```
add esp, 4
pop ebp
```

这也是更有效率的。我不知道为什么编译器用那样麻烦的方法增加堆栈指针。记得 ecx 可以被自由使用和复制，pop ecx 看起来像把数据放到了垃圾桶。

Now we only need to investigate what is hiding behind the [A] in the line fld qword ptr [A].We know that A must be a pointer to the place where A is placed in memory.The address of A is coded in the instruction. This is the full line from the cpu view.

```
00451E40 DD05803C4500        fld qword ptr [B]
```

00451E40 is the address of this instruction in the exe file.
DD05803C4500 is the machine code for the line and fld qword ptr [B] is the more human readable format of it.By consulting the Intel SW Developers Manual Volume 2 on page 280 we see that the opcode for fld is D9, DD,DB or D9C0 depending on the type of data it should load. We recognize DD that is the opcode for fld double. What is left is 05803C4500. 05 is (Somebody help me !). 803C4500 is the 32-bit address of A.

现在我们只需要研究代码 fld qword ptr [A] 中隐藏在 [A] 后面的东西。
我们知道 A 肯定是一个指向存放 A 的内存位置的指针。
A 的地址在指令中被编码。cpu view 中的完整代码如下：

```
00451E40 DD05803C4500        fld qword ptr [B]
```

00451E40 是 exe 中指令的地址。

DD05803C4500 是机器代码，fld qword ptr [B] 也是我们可以读懂的格式。
通过查看 Intel 软件开发手册卷 2 的第 280 页，我们知道 fld 的操作码是 D9,DD,DB 或者 D9C0，它们依赖于装入的数据类型。我们认定 05803C4500 左边的 DD 表示 fld 浮点数。05 是什么呢？(谁能告诉我!)。803C4500 是 A 的 32 位地址。

Let us convert the function into a pure BASM function now that we have finished analyzing it.

我们已经完成了分析，现在将函数转为纯 BASM 的：

```
function SecondOrderPolynomial3(X : Double) : Double;
const
  A : Double = 1;
  B : Double = 2;
  C : Double = 3;
asm
  push    ebp
  mov      ebp,esp
  add      esp,-$08
  //Result := A*X*X + B*X + C;
  fld      qword ptr [A]
  fmul    qword ptr [ebp+$08]
  fmul    qword ptr [ebp+$08]
  fld      qword ptr [B]
  fmul    qword ptr [ebp+$08]
  faddp //st(1)
  fadd    qword ptr [C]
  fstp    qword ptr [ebp-$08]
  wait
  fld      qword ptr [ebp-$08]
  pop      ecx
  pop      ecx
  pop      ebp
end;
```

Now come a few surprises. First the function will not compile. faddp st(1) is not recognized as a valid combination of opcode and operands. By again consulting the Intel manual we learn that faddp comes in one version only. It operates on st(0), st(1) and it is not necessary to write faddp st(0),st(1) and the short form faddp is the only valid one. We comment out st(1) and it compiles now.

现在出现了一些问题。
首先，函数不能编译。faddp st(1) 不是合法的操作码和操作数组合。再次参考 Intel 手册，faddp 只有一个版本，它操作 st(0), st(1) 并且不需要这样写 faddp st(0), st(1)。
简短的 faddp 是唯一合法的一个。我们注释掉 st(1)，现在编译通过了。

Second surprise.
Calling the function with X = 2 yields the calculation Y = 2^2+2*2+3 = 11.
SecondOrderPolynomial3 returns 3!
We must open the FPU view as well as the CPU view and trace through the code and watch what is happening.It is seen that A=1 is correctly loaded into st(0) by the 4 line,but the 5 line that should multiplicate A by X, 1 by 2, is resulting in st(0) being a very small number,in effect 0.This tells us that X is near zero instead of 2.Two things can be wrong. The calling code is transferring a wrong value of X or we are addressing X incorrectly.By comparing the calling code when calling function SecondOrderPolynomial3 and SecondOrderPolynomial1 we see that it is the same and this is not the bug.It would also be quite surprising if Delphi were suddenly getting this wrong! Try to step through the calling code while watching the memory pane in the CPU view.
The little green arrow is the position of the stack pointer.

另一个问题是给函数传传入形参 X =2，应计算 Y = 2^2+2*2+3 = 11，可是 SecondOrderPolynomial3 结果居然是 3。
我们必须打开 FPU 窗口像跟踪代码一样观察发生了什么。
我们看到，第四行将 A=1 装入 st(0) 是正确的, 但是第 5 行是 A *X 即 1*2，它的结果在 st(0) 中变成了一个非常小的数，接近 0。 这告诉我们 X 是接近零而不是 2。两个地方可能出错。调用代码传入了一个错误的 X 值或者我们寻址 X 错误。
通过比较 SecondOrderPolynomial3 与 SecondOrderPolynomial1 的调用代码，我们看到它们是相同的，没有任何错误。如果 Delphi 产生这个错误，它也是相当令人惊讶。
再试着在 CPU view 中跟踪代码，观察内存面板。那个小的绿色箭头是栈顶的位置。

The calling code looks like this
调用代码看起来是这个：

```
push dword ptr [ebp-$0c]
push dword ptr [ebp-$10]
```

call SecondOrderPolynomial1 two pointers are pushed onto the stack. One of them is a pointer to X.I do not what the other one is.
By watching the memory pane we see that the first one is the pointer to X and the second one is a nil pointer.When we trace into the function we see that the first two lines are repeated.The compiler automatically inserted the push ebp and mov ebp, esp lines. Because push decrements the stack pointer by 4,the reference to X went wrong. These two first lines are removed and everything is ok again.
Now we have finished analyzing the code and know what it does, we can begin optimizing it.Let us first change the two fstp/fld lines that we already have seen is redundant.

调用 SecondOrderPolynomial1 时的两个指针指向的内容被压入堆栈。

它们其中的一个指向 X。我不知道另一个是什么。
通过观察内存窗口我们发现第一个是指向 X 的指针，第二个是一个 nil 指针。
当我们跟踪进函数时我们看到前两行是重复的。编译器自动的插入 push ebp 和 mov ebp, esp。
因为 push 使栈顶减 4，因此引用 X 时发生了错误。删除这两个，函数工作正常了.
现在，我们完全分析了代码，并且知道它做了些什么，我们开始优化它.
我们先更改已经发现的冗余代码 fstp 与 fld 这两行。

```
function SecondOrderPolynomial4(X : Double) : Double;
const
  A : Double = 1;
  B : Double = 2;
  C : Double = 3;
asm
  //push    ebp
  //mov      ebp,esp
  add      esp,-$08
  //Result := A*X*X + B*X + C;
  fld      qword ptr [A]
  fmul     qword ptr [ebp+$08]
  fmul     qword ptr [ebp+$08]
  fld      qword ptr [B]
  fmul     qword ptr [ebp+$08]
  faddp //st(1)
  fadd     qword ptr [C]
  //fstp    qword ptr [ebp-$08]
  wait
  //fld      qword ptr [ebp-$08]
  pop      ecx
  pop      ecx
  pop      ebp
end;
```

This was the only reference to the stack frame, which is not needed now.
与堆栈相关的也不需要：

```
function SecondOrderPolynomial5(X : Double) : Double;
const
  A : Double = 1;
  B : Double = 2;
  C : Double = 3;
asm
  //push    ebp
  //mov      ebp,esp
  //add      esp,-$08
```

```
  //Result := A*X*X + B*X + C;
  fld      qword ptr [A]
  fmul     qword ptr [ebp+$08]
  fmul     qword ptr [ebp+$08]
  fld      qword ptr [B]
  fmul     qword ptr [ebp+$08]
  faddp //st(1)
  fadd     qword ptr [C]

  wait

  //pop      ecx
  //pop      ecx
  //pop      ebp
end;
```

That removed another 6 lines and reduces the function to this.
删除这 6 行，代码减为：

```
function SecondOrderPolynomial6(X : Double) : Double;
const
  A : Double = 1;
  B : Double = 2;
  C : Double = 3;
asm
  //Result := A*X*X + B*X + C;
  fld      qword ptr [A]
  fmul     qword ptr [ebp+$08]
  fmul     qword ptr [ebp+$08]
  fld      qword ptr [B]
  fmul     qword ptr [ebp+$08]
  faddp
  fadd     qword ptr [C]
  wait
end;
```

X is loaded from memory into the FPU 3 times. It would be more effective to load it once and then reuse it.
X 从内存装入 FPU 3 次。其实装入一次更有效，修改如下：

```
function SecondOrderPolynomial7(X : Double) : Double;
const
  A : Double = 1;
  B : Double = 2;
  C : Double = 3;
asm
```

```
  //Result := A*X*X + B*X + C;
  fld      qword ptr [ebp+$08]
  fld      qword ptr [A]
  fmul    st(0), st(1)
  fmul    st(0), st(1)
  fld      qword ptr [B]
  fmul    st(0), st(2)
  ffree st(2)
  faddp
  fadd    qword ptr [C]
  wait
end;
```

I magically came up with this code. The first line loads X. The second line loads A.The third line multiplies A with X.
The 4. line multiplies a*X know in st(0) with X.
Then we have calculated the first term. Loading B and multiplication it with X does calculating the second term.
This was the last time we needed X in we free the register, st(2), holding it.
Now adding term 1 and 2 and popping term 2 of the stack. The only thing left to do is adding C.
The result is now in st(0) and the other registers are empty.
Then we check for exceptions with wait and are done.
It is seen that no redundant work is done and this implementation is near optimal.

我魔法般的处理了这些代码。第一行转入 X，第二行装入 A，第三行执行 A * X。
第 4 行将 A*X 的结果 st(0) 再乘以 X。
我们已经计算完第一段。装入 B 并且乘以 B 来计算第二段。
这是我们最后一次使用 X，因此释放寄存器 st(2)。
现在将第 1 段和第 2 段相加，弹出第 2 段。剩下的事情就是加 C。
结果现在在 st(0) 中，其他寄存器都是空的。
然后我们用 wait 检查意外。
现在代码看起来没有冗余了，同时它运行的也很好。

There exits seven instructions for loading often used constants into the FPU.
One of these constants is 1, which can be loaded with the instruction fld1.
Using it saves one load from memory, which can be costly in terms of clock cycles if data are not properly aligned.

有 7 条指令经常用于装载常量到 FPU。这些常量之一是 1，它能被指令 fld1 装入。使用它节省了从内存转装入的一次。如果数据不是完全对齐，原来的指令会耗费不少时钟周期。

```
function SecondOrderPolynomial8(X : Double) : Double;
const
```

```
  //A : Double = 1;
  B : Double = 2;
  C : Double = 3;
asm
  //Result := A*X*X + B*X + C;
  fld      qword ptr [ebp+$08]
  //fld      qword ptr [A]
  fld1
  fmul    st(0), st(1)
  fmul    st(0), st(1)
  fld      qword ptr [B]
  fmul    st(0), st(2)
  ffree st(2)
  faddp
  fadd    qword ptr [C]
  wait
end;
```

This ended the second lesson. Stay tuned for more.

第二课结束了。稍微休息一下，接下来还有。

Lesson 3 第 3 课(64 位运算,MMX,SSE2 指令)

In this third lesson topic such as MMX and SSE2 will be introduced together with Int64 arithmetic's.
This is the first time we will see processor dependent optimisations.

第 3 课的主题是介绍 Int64 的算术运算，同时也会介绍 MMX,SSE2 指令。
我们将第一次看到依赖处理器的优化。

The example looks like this
例子如下：

```
function AddInt64_1(A, B : Int64) : Int64;
begin
  Result := A + B;
end;
```

Let us jump straight into the asm code.
我们直接来看汇编代码：

```
function AddInt64_2(A, B : Int64) : Int64;
begin
  {
  push ebp
  mov ebp,esp
  add esp,-$08 //  分配临时空间
  }
  Result := A + B;
  {
  mov eax,[ebp+$10]
  mov edx,[ebp+$14]
  add eax,[ebp+$08] //  低  32  位相加
  adc edx,[ebp+$0c] //  带进位高  32  位相加
  mov [ebp-$08],eax //  低  32  位
  mov [ebp-$04],edx //  高  32  位
  mov eax,[ebp-$08]
  mov edx,[ebp-$04]
  }
  {
  pop ecx
  pop ecx
  pop ebp
  //ret
  }
```

end;
The first three lines of code are recognized as setting up a stack frame like in the previous lessons.
This time we know that the compiler might add the first two for us.
The last three lines are also a well-known pattern. Again the compiler might add pop ebp for us.
This brings us into the meat, which are these 8 lines Result := A + B;

开始的前三行汇编代码用来设置堆栈，就像前面课程讲的一样。
我们知道编译器为我们增加了前两行。
最后三行代码也是我们熟悉的形式。此外，编译器为我们增加了 pop ebp。
我们来分析 Result := A + B 产生的这 8 行汇编代码：

```
{
mov eax,[ebp+$10]
mov edx,[ebp+$14]
add eax,[ebp+$08]
adc edx,[ebp+$0c]
mov [ebp-$08],eax
mov [ebp-$04],edx
mov eax,[ebp-$08]
mov edx,[ebp-$04]
}
```

They can be analysed in pairs because they work together in tandem doing 64-bit math by splitting the problem up into 32 bit pieces. The first two lines load A into the register pair eax:edx.
They are loading a contiguous 64-bit block of data from the previous stack frame,
showing us that A was transferred on the stack.

我们来成对分析汇编代码，因为 64 位运算是通过把一个 64 位数分为低 32 位和高 32 位来运算。前两行将形参 A 存入寄存器对 eax: edx。它们将之前堆栈中连续的 64 位数据装入，显然 A 是通过栈传输。

The two load pointers are separated by 4 bytes.
One of them is pointing to the beginning of A and the other one is pointing into the middle of A.
Then comes two add instructions. The first is a normal add and the second one is add with carry.
The pointers in these two lines are pointing to B in the same fashion as the two previous were pointing at A.
The first add adds the lower 32 bits of B to the lower 32 bits of A.
This might lead to a carry if the sum is too big to fit into 32 bits.
This carry is included in the addition of the higher 32 bits.
这两个数据指针之间，有 4 个字节的间隔，
其中一个指向 A 的开始，另外一个指向 A 的中间。
之后，是两个加法指令，第一个是普通相加，第二个是带进位相加，
这两行中的数据指针和前面指向 A 的数据指针是同样的方式。
第一个 add 指令是将 B 的低 32 位与 A 的低 32 位相加。
如果得到的和太大以至于用 32 位装不下它，会产生一个进位，

这个进位将包含在高 32 位的相加中。
To make things totally clear lets do a simple example on decimal numbers.
We have the addition 1+2 = 3.
Our imaginary data types for this in our brain CPU as two digits wide.
This means that the addition is actually looking like this 01+02=03.
There is no carry from the addition of the lower digits into the higher ones, which are zero.

为了清楚明了的解释它，我用一个简单的 10 进制相加的例子来说明。
第一个例子是 1 + 2 = 3 。我们假设我们的大脑 CPU 是 2 个数字的宽度。
这就意味着这个加法其实是 01 + 02 = 03 这种形式，它的低位相加没有进位，进位是零。

Decimal example two. 13+38=?. First we add 3+8=11.
This results in a carry and a 1 in the lower half of the result.
Then we add Carry+1+3=1+1+3=5.
The result is 51.

第二个例子是 13 + 38 = ? 。首先我们计算 3 + 8 = 11，产生了一个进位。
然后，我们计算 进位 + 1 + 3 = 1 + 1 + 3 =5，结果是 51。

In the third example we provoke an overflow. 50+51=101.
101 is too big to fit in two digits and our brain CPU cannot perform the calculation.
There was a carry on the addition of the two higher digits. Back to code.
Two things can happen now.
If we have compiled without range check the result wraps around. With range check an exception will be thrown.
We see that there is now range check code in our listing and wraparound will occur.

第三个例子，我们制造一个溢出。50 + 51 = 101。
101 太大，用 2 位数装不下，并且我们的大脑 CPU 也不能执行计算了。因为两个高位相加也产生了一个进位。
回头看代码，可能有两种情况：
1、编译时没有使用范围检查结果的边界。
2、若使用范围检查将抛出一个意外。我们会看到进行范围检查的代码列表提示，同时产生了一个边界溢出。

The next two lines save the result into the current stack frame.
The last two lines load the result from the stack frame into eax and edx where it already was.
These 4 lines are redundant. They can be removed and this also removes the need for a stack frame.
it so easy to be an optimizer ;-)

接下来的两行代码是将结果 EAX, EBX 保存到堆栈。
最后两行再将结果从堆栈还原到 EAX,EDX 中。
显然，这 4 行是多余的。他们可以被删除，同时也删除堆栈相关的代码。

```
function AddInt64_6(A, B : Int64) : Int64;
asm
  mov eax,[ebp+$10]
  mov edx,[ebp+$14]
  add eax,[ebp+$08]
  adc edx,[ebp+$0c]
end;
```

This is a nice small function.
The compiler generated code consisted of 16 lines and we came down to 4 with only little effort.
Today Delphi was really sleepy.
Now we think like this: If we had 64 bit registers the addition could be done with two lines of code.
But the MMX registers are 64 bits wide and this might be worth taking advantage of.

这是一个非常简洁的函数。
编译器产生了代码包含了 16 行，而我们只是通过一点努力就将代码精简为 4 行。
Delphi 真是有点懒惰啊。
现在我们想，如果我们有 64 位的寄存器，那么 64 位加法用两行代码就能实现。
其实, MMX 寄存器就是 64 位宽，它们可能会有利用价值。

In the Intel SW Developers Manual instructions are not marked as belonging to IA32, MMX, SSE or SSE2.
This information would be nice to have, but we have to look elsewhere for it.
I normally use three small programs from Intel. The so called computer based tutorials on MMX, SSE & SSE2.
I do not know where to find them on the Intel website now, but mail me if you want them.
They are simple and nice - very illustrative.
In these I find that a mov for 64 bits from memory into an MMX register is movq.
Q stands for quad word. The mmx registers are named mm0, mm1....mm7.
They are not arranged as a stack, as the FP registers are, and we can pick which one we like.

在 Intel 软件开发手册中，没有显著的描述 IA32, MMX, SSE 或 SSE2 等指令。
如果手册中有这些指令那就太好了，但是我们不得不到其他地方寻找这些指令。
我利用了 Intel 提供的 3 个小程序，它们被称为关于 MMX, SSE & SSE2 计算机的基本教程。
我不知道现在如何从 Intel 网站找到它们，如果你想要可以给我发 Email。
它们是简单的，精巧的，非常有说明性.
在这些资料中，我发现了一个 64 位的移动指令 movq, 它将数据从内存移到一个 MMX 寄存器.
Q 表示四倍字。mmx 寄存器组被命名为 mm0, mm1....mm7。
它们不能像 FP 寄存器那样作为堆栈使用，我们可以随意使用任意一个。

Lets pick mm0. The first instruction looks like this
movq mm0, [ebp+$10]
There is to ways two go now. We can load B into a register too.
This makes it easy to see what is going on by using the FPU window.
The MMX registers are aliased onto the FP registers and the FPU view can show both sets.
Switch between FP and MMX view by select "Display as words/Display as extendeds" in the shortcut menu.
The second way to go is to use the pattern from the IA32 implementation and perform the addition with
the memory location of B as source.
以 mm0 为例。第一个指令像下面的形式：
movq mm0, [ebp+$10]

现在有两种方式可以选择。
第一种方式是我们也可以将 B 装载到一个寄存器。
这种方式很容易通过 FPU 寄存器看到。
MMX 寄存器作为 FP 寄存器的别名，FPU 窗口可以显示两种指令集。
转换 FP 和 MMX 显示可以通过选择右击快捷菜单上的 "显示字/显示扩展的"

第二种方式是使用 IA32 执行模式，把内存中的 B 作为源操作数。

The two solutions is expected to perform identically because the CPU needs to load B into registers
before doing the addition and whether it is done explicitly with mov or explicitly with the add instruction,
the number of micro instructions will be the same. We use the more illustrative first way.

这两种方式有相同的执行效果。因为 CPU 在执行加法指令之前，需要将 B 装载到寄存器。
不管它是否明确的使用 mov 指令或者 add 指令，所使用的微指令个数是相同的。
我们可以使用更多的命令来说明它。

The next line is then a movq again
还是一个 movq 指令
movq mm1, [ebp+$08]

Then we have to go look for an add instruction which would be something like this- paddq.
P for MMX, add for addition and q for quad word.
Now we get disappointed because there is no such MMX instruction. What about SSE.
One more disappointment.

然后我们需要寻找一个加法指令，像 paddq 形式。
P 表示 MMX，add 表示加法，q 表示四个字大小
当然，我们很失望，因为没有这样的 MMX 指令。
SSE 呢？ 也令人失望。

Finally SSE2 got it and we are happy or are we? If we use it the code will be targeting P4 and not run P3 or Athlon.
Like the P4 lovers we are we proceed anyway.
paddq mm0, mm1
This line is very intuitive. Adding mm1 to mm0.
Only thing left is to copy the result from mm0 into eax:edx.
To do this we need a double word mov instruction that can take 32 bits from a MMX register as source and
an IA32 register as destination.
movd eax, mm0

This MMX instruction does the job. It copies the lower 32 bits of mm0 to eax.
最后我们在 SSE2 中发现了它，我们该不该为之高兴呢？
如果我们使用它，那么代码只能运行在 P4 上，不能运行在 P3 或 Athlon 上。
P4 爱好者们可以继续看下去

paddq mm0, mm1

这一行非常直观，就是累加 mm1 到 mm0。
接下来的事情是将结果从 mm0 复制到 eax:edx。
为了实现它，我们需要一个双字 mov 指令，它可以从作为源操作数的 MMX 寄存器中复制 32 位,
存入作为目标操作数的 IA32 寄存器中。

movd eax, mm0

MMX 指令可以完成这个工作，movd 复制 mm0 的低 32 位到 EAX 中。

Then we need to copy the upper 32 bits of the result to edx. I could not find an instruction for that and insteadI shift the upper 32 bits down into the lower 32 bit using a 64-bit MMX right shift instruction.

然后，我们需要复制 mm0 的高 32 位到 eax 中。我没有发现实现这个功能的指令。
一个替代的方法是使用一个 64 位的 MMX 右移指令，把高 32 位移到低 32 位中。

psrlq mm0, 32
Then we copy
然后我们复制
movd edx, mm0
Then we are done? Unfortunately we have to issue the emms instruction because we have used MMX instructions.

现在结束了吗？很不幸，因为我们使用了 MMX 指令集，因此需要发布 emms 指令。

It cleans up the FP stack and leaves in a well-defined empty state. Emms bums 23 cycles on a P4. Together with the shift which is also ineffective (2 cycles throughput and latency) on P4 our solution is not
especially fast and it will only run on P4 and this AMD thing nobody has yet:-(

它清理 FP 堆栈，即清空堆栈。
Emms 在 P4 上需要 23 个时钟周期。
加上移位指令也不是很有效率(2 cycles throughput and latency 两个时钟周期的吞吐量和潜伏期), 因此我们的 P4 解决方案不是特别的快,
并且它只能运行在 P4 上, AMD 还没有人实验过:-(

This ended the 3. lesson. We left the ball hanging in the air. Can we come up with a more efficient solution?
Moving data between MMX register and IA32 registers is expensive. The calling convention is no good, because data
were transferred on the stack and not in registers. eax->mm0 is 2 cycles.
The other way is 5 cycles. emms is 23 cycles. Addition is only 2 cycles. Overhead is plenty.

第 3 课的总结。
我们留个思考题，我们能找出一个更有效率的解决方案吗？
在 MMX 寄存器和 IA32 寄存器中传输数据的代价很高。调用约定不是很好，因为数据在堆栈上传输而不是在寄存器中传输。
eax -> mm0 是 2 个周期，其他方式是 5 个周期， emms 是 23 个周期，加法只是 2 个周期，整个时间的耗费是很大的。

Lesson 4 第 4 课(分支语句if-else)

In this lesson we will learn about branching, illustrated by an if-else construct.
Conditional floating point move will also be introduced.
The example function of this lesson is the Min function from the Delphi Math unit.

在这一课中,我们将学习分支语句，通过一个 if-else 结构来举例说明。
同时介绍浮点数条件转移。
这一课的例子函数是 Delphi Math 单元中的 Min 函数。

```
function Min1(const A, B: Single) : Single;
begin
  if A < B then
    Result := A
  else
    Result := B;
end;
```

The compiler generated asm for this function looks like this
汇编代码如下：

```
function Min2(const A, B: Single) : Single;
begin
  {
  00452458 55                    push ebp
  00452459 8BEC                  mov ebp,esp
  0045245B 51                    push ecx
  }
  if A < B then
  {
  0045245C D9450C                fld dword ptr [ebp+$0c]
  0045245F D85D08                fcomp dword ptr [ebp+$08]
  00452462 DFE0                  fstsw ax
  00452464 9E                    sahf
  00452465 7308                  jnb +$08
  }
    Result := A
    {
    00452467 8B450C              mov eax,[ebp+$0c]
    0045246A 8945FC              mov [ebp-$04],eax
    0045246D EB06                jmp +$06
    }
  else
    Result := B;
```

```
  {
  0045246F 8B4508          mov eax,[ebp+$08]
  00452472 8945FC          mov [ebp-$04],eax
  }
  {
  00452475 D945FC          fld dword ptr [ebp-$04]
  00452478 59              pop ecx
  00452479 5D              pop ebp
  }
end;
```

This time I included the address and opcode columns, because we need them later.
Lets analyze it line by line, like we always do.
这一次我复制了地址和机器码列，因为后面会用到。
像之前一样，我们来一行行分析。

```
function Min3(const A, B: Single) : Single;
begin
 {
 push ebp                    // Save ebp on stack    保存 ebp
 mov ebp,esp                 // New basepointer is the old stackpointer 堆栈指针赋给基
址指针
 push ecx                    // subtract 4 from esp    esp 减 4
 }
 if A < B then
 {
 fld dword ptr [ebp+$0c]     // Load A on FP stack    A 到浮点数寄存器堆栈
 fcomp dword ptr [ebp+$08]   // FP compare A to B and pop A from stack A 与 B 比较之后，
将 A 从 FP 中弹出
 fstsw ax                    // Store FP statusword in ax 将 FP 的状态字存入 ax
 sahf                        // Store ah into EFlags register 存储 ah 到标志寄存器
 jnb +$08                    // If not below jump 8 bytes forward 如果不小于，则向前跳
8 个字节
 }
  Result := A
  {
  mov eax,[ebp+$0c]          // Copy A into eax    A 到 eax
  mov [ebp-$04],eax          // Copy A into stackframe A 到堆栈
  jmp +$06                   // Jmp 6 bytes forward 向前跳 6 个字节
  }
 else
  Result := B;
  {
  mov eax,[ebp+$08]          // Copy B into eax B 到 eax
  mov [ebp-$04],eax          // Copy B into stackframe    B 到堆栈
```

```
  }
  {
  fld dword ptr [ebp-$04]        // Load A or B from stackframe onto FP stack 装载 A 或 B 到
FP 堆栈
  pop ecx                        // Add 4 to esp   esp 加 4
  pop ebp                        // Restore ebp 恢复 ebp
  }
end;
```

This time I commented every line of BASM. Read the details there.
The first new instruction introduced by this example is fcomp. F says floating-point instruction as always.Com says compare and p says pop FP stack. Fcom compares two floating-point values and sets the condition code flags of the floating-point unit, named C0, C1, C2 and C3.

这一次我注释了每一行 BASM。从这可以看到详细说明。
这个例子中第一个被介绍的新指令是 fcomp。
F 表示浮点数指令，Com 表示比较，p 表示从浮点数寄存器堆栈弹出。
FCom 比较两个浮点数，并且设置被命名为 C0, C1, C2, C3 的浮点数单元的条件代码标志。

These flags are the equivalents of the EFlags register of the CPU.
The flags are checked by conditional jump instructions, which jump or not depending on the type of jump and the status of the flags.
这些标志等同于 CPU 的标志寄存器。
条件跳转指令检查这些标志，跳不跳转，依赖于跳转的类型和这些标志的状态。

Conditional jump instructions check the CPU flags and not the FPU flags and therefore it is necessary to copy the flags from the FPU to the CPU.
条件跳转指令检查 CPU 的标志而不是 FPU 的标志，因此需要将标志从 FPU 拷贝到 CPU。

This is done by the next two instructions.
fstsw stores the FP flags into the ax register and sahf copy the 8 bits from ah to the EFlags register.
This is a long way for the flags to travel before they can be used by the jnb instruction.
Jnb is short for jump not below. In the Intel SW Developers Manual Vol 2 at page 394 there is a table with all the different jump instructions and a description of the flags they test.

下两行指令实现这个功能。
fstsw 存放 FP 标志到 ax 寄存器，sahf 从 ah 复制 8 位到状态标志寄存器。
在它们被 jnb 指令使用之前，经历了一个比较长的步骤。
Jnb 是一个短跳转，如果不低于则跳转。在 Intel 软件开发手册卷 2 的第 394 页中有一张表描述了
所有跳转指令的不同以及它们所检测的标志寄存器。

Here it is seen that jnb jumps if CF=1 and ZF=1.

Try tracing through the code with the FPU view and the CPU view open.
Watch how the FPU flags are set, their values are copied into ah and then copied to the CPU EFlags register.
If the jnb jump is not taken execution continues in the line after it.
This is the if part of the if-else construct.
If the jump is taken execution is continued at the line 8 bytes ahead.
This is at the start of the else part. The if and the else part is very much the same.

这儿看起来如果 CF =1 并且 ZF =1 时 jnb 跳转。
通过 FPU 窗口和 CPU 窗口跟踪代码。
观察 FPU 状态怎样被设置，它们的值如何拷贝到 ah，并且再拷贝到 CPU 的状态寄存器。
如果 jnb 跳转没有执行，那么将继续执行它之后的指令。这是 if-else 结构的 if 部分。
如果 jnb 跳转执行了，那么将从这一行前面的 8 个字节处执行。
这是 else 部分的开始。if 和 else 部分非常相似。

As seen in the Pascal code A or B is copied to the Result variable depending on the if condition.
Instead of copying A or B directly on the top of the FP stack, which is the place of a FP Result from a function due to the register calling convention, the Delphi compiler chose to use the stack as a temporary location.
The fld dword ptr [ebp-$04] instruction copies the result in the right place.

从 Pascal 代码中可以看到，A 或者 B 被拷贝到结果变量依赖于 if 条件。
不是从 FP 堆栈的顶部直接复制 A 或 B，其栈顶存放由寄存器调用预定产生的结果，Delphi 编译器选择使用堆栈作为临时本地变量。
fld dword ptr [ebp-$04] 装载指令右面的结果。

Observe that an unconditional jump is needed at the end of the if block such that execution does not continue in the else block.
This way there is a jump no matter with branch is taken. A short word on branch prediction.
Branch prediction comes in two flavours, static and dynamic. The first time a branch is executed the CPU can have no knowledge about the probability it will be taken or not. In this situation it uses the static predictor, which says that forward jumps are not taken and backward branches are taken.

显然，如果不执行 else 块，则在 if 块的结尾需要一个无条件跳转指令。
不论是否进入分支都有一个跳转。是分支预测中的短字。
分支预测有两种方式，静态预测和动态预测。
分支一开始执行，CPU 不知道分支是否能够执行。
这种情况下，它使用静态预测，表示向前的跳转不被执行，向后的跳转分支被执行。

Our example jump no 1 is a forward branch and it will be predicted not taken the first time through.
If we have knowledge about the values A and B, we can use it to code the if-else such that the if part is the most often executed and static prediction is optimized.

An unconditional jump needs no prediction - it is always taken ;-)
A backward jump is often part of a loop and most loops will run more than once (why else have a loop?).
This way it makes sense to statically predict backward jumps taken.
Dynamic branch prediction tries to accumulate knowledge about the probability that a certain jump is taken or not and then predict it correctly as often as possible.

例子中的跳转 1 是向前分支，第一次不会进入它。
如果我们知道 A 和 B 的值，我们可以这样使用 if-else 编码，if 部分放置最经常被执行的代码，静态预测被优化了 。
一个无条件跳转不需要预测 - 它总是被执行 :)
一个向后的跳转常常是循环中的一部分，大部分循环都将运行多次(为什么 else 有一个循环)。
这种方式对于静态预测向后跳转比较有意义。
动态分支预测累计某种跳转是否执行，然后尽可能的正确预测它。

Now it is time to transform the function into a pure asm one.
是把函数转换成纯 asm 代码的时候了。

```
function Min4(const A, B: Single) : Single;
asm
  //push    ebp
  //mov      ebp,esp
  push    ecx
  //if A < B then
  fld      dword ptr [ebp+$0c]
  fcomp dword ptr [ebp+$08]
  fstsw ax
  sahf
  jnb      @ElseBegin
  //Result := A
  mov      eax,[ebp+$0c]
  mov      [ebp-$04],eax
  jmp      @ElseEnd
  //else
  //Result := B;
  @ElseBegin :
  mov      eax,[ebp+$08]
  mov      [ebp-$04],eax
  @ElseEnd :
  fld      dword ptr [ebp-$04]
  pop      ecx
  //pop      ebp
end;
```

The new thing this time is that we need some labels.

Our analyse of the function made it clear where we jumped to when,I hope.
In general when placing labels it is a good thing to use the knowledge we have about the structure of the code.You can open the FPU view and just trace through the code and see where you go when jumps are taken.If you want to place labels without having to think, then use some math. Example follows We have this jump

这一次我们需要加上一些标签。
我希望已经清楚地分析了这个函数什么时候跳转到哪里。
通常，放置标签的时候，使用我们已知的代码结构命名是个好习惯。
你可以打开 FPU 窗口，并跟踪代码，查看跳转执行的过程。
如果你想放置没有任何意义的标签，可以使用数学的方法。
比如下面的跳转例子：

```
00452465 7308                    jnb +$08
```
This is the line after it 这一行之后
```
00452467 8B450C                  mov eax,[ebp+$0c]
```
This is the line 8 byte after that line 判断行之后的 8 个字节处
```
0045246F 8B4508                  mov eax,[ebp+$08]
```
Take the address of the line after the jump and add the jump offset to it and you have the jump target address.
跳转行之后的地址加上跳转的偏移量等于要跳转到的目标地址。

This is the math: 00452467 + 8 = 0045246F.
公式为 00452467 + 8 = 0045246F.
Why do we add the jump offset to the address of the line after the jump and not to the address of the line with the jump?
为什么我们用跳转指令之后的地址来加偏移量，而不是用跳转指令这一行的地址呢？

It's time to optimise. 现在开始优化：
```
function Min5(const A, B: Single) : Single;
asm
  push    ecx
  //if A < B then
  fld      dword ptr [ebp+$0c]
  fcomp dword ptr [ebp+$08]
  fstsw ax
  sahf
  jnb      @ElseBegin
  //Result := A
  mov      eax,[ebp+$0c]
  mov      [ebp-$04],eax
  jmp      @ElseEnd
  //else
  //Result := B;
```

```
  @ElseBegin :
  mov      eax,[ebp+$08]
  mov      [ebp-$04],eax
  @ElseEnd :
  fld      dword ptr [ebp-$04]
  pop      ecx
end;
```

This is just a cleaned up version of the function. Lets change the push ecx,
pop ecx instructions with instructions that manipulate the esp register directly and do not move data between ecx and the stack.

这只是整理之后的代码。
让我们来优化掉 push ecx, pop ecx 这些直接操作 esp 寄存器的指令， ecx 和堆栈之间每有传递数据。

```
function Min6(const A, B: Single) : Single;
asm
  //push    ecx
  sub      esp, 4
  //if A < B then
  fld      dword ptr [ebp+$0c]
  fcomp dword ptr [ebp+$08]
  fstsw ax
  sahf
  jnb      @ElseBegin
  //Result := A
  mov      eax,[ebp+$0c]
  mov      [ebp-$04],eax
  jmp      @ElseEnd
  //else
  //Result := B;
  @ElseBegin :
  mov      eax,[ebp+$08]
  mov      [ebp-$04],eax
  @ElseEnd :
  fld      dword ptr [ebp-$04]
  //pop      ecx
  add esp, 4
end;
```

When the code was analyzed we saw that the flags were traveling a long way and 2 instructions were needed for this.
What about a floating point compare instruction that sets the EFlags directly?
This is the fcomi instruction, which was introduced in the P6 architecture.
Lets use it and abandon those CPUs older then the Pro. It would be fair to believe that these lines

当我们分析传输标志指令时，发现是用两个指令来实现的。
一个浮点数指令怎样来直接设置 CPU 的标志寄存器呢?
通过 fcomi 指令，它在 P6 结构中被介绍.

```
fcomp dword ptr [ebp+$08]
fstsw ax
sahf
```

could be substituted by this 可以被下面的代码替代

```
fcomip dword ptr [ebp+$08]
```

The fcomi instruction does however not accept a memory reference as operand.
Therefore it is necessary to load data before issuing it.
fcomi 指令不能引用内存操作数。因此使用它之前需要装载数据：

```
fld       dword ptr [ebp+$0c]
fcomip st(0), st(1)
```

Then because we loaded data we have to remove them again with the ffree instruction.
An fcomipp instruction would have been nice to have.
之后，因为我们装入了数据，因此必须用 ffree 指令移除。
fcomipp 指令可能更适合做这个。

```
fld       dword ptr [ebp+$0c]
fcomip st(0), st(1)
ffree    st(0)
```

This is a hell of an optimization, substituting three lines with 3 other.
It is necessary to time the two versions to know whether it was an optimization.
Now the function looks like this.

这是一个魔鬼般的优化，用另外 3 行代替原来的 3 行。
需要来测试这两个版本，看看哪个是比较有效率的。
此函数如下：

```
function Min7(const A, B: Single) : Single;
asm
    sub       esp, 4
    //if A < B then
    fld       dword ptr [ebp+$08]
    fld       dword ptr [ebp+$0c]
    fcomip st(0), st(1)
    ffree    st(0)
    //fstsw ax
```

```
    //sahf
    jnb       @ElseBegin
    //Result := A
    mov       eax,[ebp+$0c]
    mov       [ebp-$04],eax
    jmp       @ElseEnd
    //else
    //Result := B;
  @ElseBegin :
    mov       eax,[ebp+$08]
    mov       [ebp-$04],eax
  @ElseEnd :
    fld       dword ptr [ebp-$04]
    add       esp, 4
end;
```

Now we apply some logic thinking. Why copy the result around? Both A & B is on the stack for use by the comparison by fcom and the result should be delivered on the stack. The only thing needed is to delete either A or B and leave the smallest one of them on the stack

现在我们进行一些逻辑上的思考。为什么要反复复制结果呢？
A 和 B 通过堆栈传递进行 fcom 比较，结果也在堆栈上传输。
只需要删除 A 或者 B，将最小的留在堆栈上。

```
function Min8(const A, B: Single) : Single;
asm
    sub       esp, 4
    //if A < B then
    fld       dword ptr [ebp+$08]
    fld       dword ptr [ebp+$0c]
    //fcomip    st(0), st(1)
    fcomi    st(0), st(1)
    //ffree    st(0)
    jnb       @ElseBegin
    //Result := A
    //mov       eax,[ebp+$0c]
    //mov       [ebp-$04],eax
    ffree st(1)
    jmp       @ElseEnd
    //else
    //Result := B;
  @ElseBegin :
    //mov       eax,[ebp+$08]
    //mov       [ebp-$04],eax
    fxch
```

```
    ffree st(1)
  @ElseEnd :
    //fld       dword ptr [ebp-$04]
    add       esp, 4
end;
```

Fcomip is replaced with fcomi because we do not want to remove B from the stack at this point.
Ffree is deleted because it deleted A. Then all the lines that copied the result around are removed.
In the if block A is the result and B should be deleted. B is in st(1) and ffree st(1) does the job.
In the else block we want to delete A and leave B in st(0). Swapping A and B do this with the fxch instruction and then delete A in st(1) with ffree. Fxch is nearly free (takes 0 cycles),
because if works by register renaming instead of actually copying data.

Fcomip 被 fcomi 替代，因为我们不想从堆栈上移除 B。Ffree 被删除，因为它释放了 A。所有的重复的复制都被删除。
在 if 块中， A 是结果，那么 B 会被删除。B 是在 st(1) 中, ffree st(1) 将其删除。
在 else 块中，我们想删除 A, 将 B 留在 st(0) 中。用 fxch 指令交换 A 和 B 的值，然后用 ffree 删除 st(1) 中的 A。
Fxch 是很节省的(花费 0 个周期)，因为它是重命名寄存器而不是复制数据。

```
function Min9(const A, B: Single) : Single;
asm
    //sub       esp, 4
    //if A < B then
    fld       dword ptr [ebp+$08]
    fld       dword ptr [ebp+$0c]
    fcomi     st(0), st(1)
    jnb       @ElseBegin
    //Result := A
    ffree st(1)
    jmp       @ElseEnd
    //else
    //Result := B;
  @ElseBegin :
    fxch
    ffree st(1)
  @ElseEnd :
    //add       esp, 4
end;
```

Now the stack frame is not needed and we delete the code that set it up.
操作堆栈的代码也不需要:

```
function Min10(const A, B: Single) : Single;
asm
```

```
    //if A < B then
    fld       dword ptr [ebp+$08]
    fld       dword ptr [ebp+$0c]
    fcomi    st(0), st(1)
    jnb       @ElseBegin
    //Result := A
    ffree st(1)
    jmp       @ElseEnd
    //else
    //Result := B;
  @ElseBegin :
    fxch
    ffree st(1)
  @ElseEnd :
end;
```

This is a pretty nice function, but somebody in the newsgroup has told us about conditional moves. fcmovnb is such a thing - floating point conditional move not below. It copies data from st(1)-st(7) to st(0) if the condition is fulfilled. The Eflags are tested for the condition. Fcmov was introduced in the P6 architecture as well fcomi was.

现在是一个非常简洁的函数了，但是在新闻组里有人告诉我关于条件移动指令。
fcmovnb 做这样一件事 - 不低于则浮点数条件移动。
如果条件满足，它从 st(1)-st(7) 拷贝数据到 st(0)。
Eflags 用于条件测试。 Fcmov 和 fcomi 一样在 P6 体系结构中被介绍。

```
function Min11(const A, B: Single) : Single;
asm
    fld         dword ptr [ebp+$08]
    fld         dword ptr [ebp+$0c]
    fcomi      st(0), st(1)
    fcmovnb st(0), st(1)
    ffree      st(1)
end;
```

Instead of all the jumping around we copy A to the top of the stack where B is,
but only if A is the smaller of the two.Delete B and done.
This is a nice clean function with only one piece of redundancy left.
The compiler still push/ pop ebp even though it is not modified in the function.

如果 A 是两者中最小的，则不需要将 B 移到栈顶，删除 B，结束。
这是一个简洁的函数，只有一点点冗余。
因为，编译器仍然会加入 push/ pop ebp，尽管函数中没有改变 ebp 寄存器的值。

Lesson 5 第 5 课(循环语句)

Welcome to lesson number 5. Today topic is loops.
We will take a look on how the compiler implements for loops, and which optimizations it applies on them. We will evaluate the efficiency of these optimizations.
欢迎来到第 5 课。今天的主题是循环。
我们将看一下编译器如何执行循环，同时做了哪些优化。
我们将评价这些优化的效率。

```
function ForLoop(Start, Stop : Integer) : Integer;
var
  I : Integer;
begin
  Result := 0;
  for I := Start to Stop do
    begin
        Result := Result + I;
    end;
end;
```

This example function does nothing useful except giving us an example for loop to examine.
Lets see what the compiler translates the function into.
In this lesson we try something new and compile with optimizations off.

这个例子除了表示一个循环的例子之外没有什么用处。我们来看看编译器是怎样编译的。
在这一课中我们试试一些新东西，把编译器优化选项关了。

```
function ForLoopNonOpt(Start, Stop : Integer) : Integer;
var
  I : Integer;
begin
{
push ebp
mov ebp,esp
add esp,-$14
mov [ebp-$08],edx
mov [ebp-$04],eax
}
  Result := 0;
  {
  xor eax,eax
  mov [ebp-$0c],eax
```

```
  }
  for I := Start to Stop do
  {
  mov eax,[ebp-$04]
  mov edx,[ebp-$08]
  sub edx,eax
  jl +$15
  inc edx
  mov [ebp-$14],edx
  mov [ebp-$10],eax
  }
    begin
        Result := Result + I;
        {
        mov eax,[ebp-$10]
        add [ebp-$0c],eax
        inc dword ptr [ebp-$10]
        }
    end;
    {
    dec dword ptr [ebp-$14]
    jnz -$0e
    mov eax,[ebp-$0c]
    }
  {
  mov esp,ebp
  pop ebp
  ret
  }
end;
```

It is seen that the compiler generates a lot of code when no or few optimizations are applied.
As usual the first 3 lines set up a stack frame. This time it is 20 byte big (16 hex).
The two next lines copy the Start and Stop variables unto the stack. Start is transferred in eax and Stop is transferred in edx to the function due to the register calling convention.
The next two lines create a zero in eax and copy it to the stack frame at [ebp-$0c],
which is the place of the result variable.

在没有进行优化时，编译器产生了许多代码。
前三行设置堆栈。这一次开辟了 20 个字节的大小(16 hex)
接下来的两行复制 Start 和 Stop 变量到堆栈中。根据寄存器调用约定，Start 通过 eax 传递，Stop 通过 edx 传递。
然后下面的两行代码将 eax 清零并复制到堆栈 [ebp-$0c] 中，用于存放结果变量。

Then we are ready to enter the for loop.
Before entering a for loop it is necessary to test whether the loop will iterate at all.
If Stop is bigger than Start this is the case.
Start and Stop is fetched from their stack locations into eax & edx.
We compute Stop-Start and if this is negative the loop should not be entered
and execution is transferred past the end of the loop by the jl (jump low) instruction.

然后准备进入循环。
在进入循环之前需要检测一下循环是否需要进入。如果 Stop 大于 Start 则需要。
Start 和 Stop 的值是通过将本地堆栈中数值复制到 eax & edx 中获得。
计算 Stop-Start，如果是负数，则不进入循环，并且通过 jl(低则跳转) 指令到达循环的末尾处执行指令。

The next line increments Stop and then it is copied to location [ebp-$14].
We have no name for the local variable at this location.
The purpose of it also needs some further explanations.
It is a variable introduced by the compiler as an optimization and this is weird because
we compiled with optimizations off.

下一行 Stop 加 1，然后将它复制到局部变量 [ebp-$14]。
我们没有声明这个局部变量。
关于它的目的需要更深入的解释。
编译器将它作为优化使用，真是令人不可思议，因为我们已经关闭了编译选项。

There is only one more line accessing the NoName variable and this is the dec dword ptr [ebp-$14] line.
dec dword ptr [ebp-$14] 也访问了这个未命名的局部变量(NoName)，

This line decrements NoName by one at the end of each loop iteration
and the loop closing test is a test that it has not reached zero.
在每次循环的末尾它将未命名的变量(NoName)减 1，通过检查它是否是零来判断循环是否结束。

The dec instruction sets the flags and the jnz jumps back to the top of the loop if NoName <> 0.
We would expect that I was used as the loop counter and that it was running from Start to Stop.
It is in fact doing so, but it is not used for control of the loop.
This is the sole purpose of NoName.

dec 指令设置标志，如果 NoName <> 0，jnz 将跳转回循环的开始。
我们期望它用于循环计数，从 Start to Stop。
事实上，它确实是这样做的，但是它不用于循环的控制。这是 NoName 唯一的目的。

The advantage of this is that it saves an instruction for comparing I to Stop.
There is also a cost and this is the dec NoName instruction.

On P4 the latency/throughput of cmp is 0.5/0.5 clock cycles and for dec they are 1/0.5.
Therefore we would expect this optimization to be a "deoptimization".
The latency and throughput numbers for P4 is found in the
"Intel Pentium 4 and Xeon Processor Optimization" manual from Intel.

这样做的优点是，它节省了一个 I 与 Stop 的比较指令。
在 P4 上, cmp 的潜伏期/吞吐量是 0.5/0.5 时钟周期，dec 指令是 1/0.5。
因此，我们预期的优化变成了一个 “非优化”
在 Intel 的 "Intel Pentium 4 和 Xeon 处理器优化"手册上有 P4 指令的潜伏期和吞吐量。

Back to the mov [ebp-$10], eax line. It copies I to the stack.
The loop body consist of only one line of Pascal Result := Result + I;.
This is translated into 3 lines of ASM.
The first two lines are loading I into eax and then adding it to Result at the stack at [ebp-$0c].
The third line increments I.
This ended the explanation of the loop code and only two things are left.
The Result must be copied to eax, which is the register to use for returning the result from
a register calling convention function.
The last three lines remove the stack frame and return execution to the line after the line that called the function.

回到 mov [ebp-$10], eax 这一行，它复制 I 到堆栈。
循环体只由一行 Pascal 代码 Result := Result + I;
它被编译成了 3 行 ASM 代码。
前两行将 I 装入 eax，然后将它累加到结果堆栈 [ebp-$0c]。
第三行是 I 加 1。
只剩下下面两个问题就解释完循环代码了：
结果必须被复制到 eax，eax 是被用来存放返回结果。
最后三行用于移除堆栈，并且返回到调用此函数的代码的下一行。

As an exercise let us change the ASM code such that it follows the Pascal code and our understanding of a for loop.
We start by turning the function into a pure BASM one.
This is done by out commenting the Pascal code and "in commenting" the ASM code.
Defining the two labels LoopEnd and LoopStart is also necessary.
The two jumps are edited such that they jump to the labels.

作为一个练习，我们将自己理解的 pascal 循环代码转为 ASM 代码。
首先将函数转为纯 BASM。
注释掉 pascal 代码，使用 ASM 代码。
也需要定义两个标签 LoopEnd 和 LoopStart。这两个标签是根据它们的跳转意义来命名的。

```
function ForLoopBASM1(Start, Stop : Integer) : Integer;
asm
```

```
  push ebp
  mov ebp,esp
  add esp,-$14
  mov [ebp-$08],edx
  mov [ebp-$04],eax
    //Result := 0;
    xor eax,eax
    mov [ebp-$0c],eax
    //for I := Start to Stop do
    mov eax,[ebp-$04]
    mov edx,[ebp-$08]
    sub edx,eax
    jl @LoopEnd
    inc edx
    mov [ebp-$14],edx
    mov [ebp-$10],eax
      //begin
      @LoopStart :
          //Result := Result + I;
          mov eax,[ebp-$10]
          add [ebp-$0c],eax
          inc dword ptr [ebp-$10]
      //end;
      dec dword ptr [ebp-$14]
      jnz @LoopStart
    @LoopEnd :
      mov eax,[ebp-$0c]
    mov esp,ebp
    pop ebp
    //ret
end;
```

The first thing we do is removing the NoName variable.
首先，我们移除 NoName 变量

```
function ForLoopBASM2(Start, Stop : Integer) : Integer;
asm
  push ebp
  push ebx                                  //New 新
  mov ebp,esp
  add esp,-$14
  mov [ebp-$08],edx
  mov [ebp-$04],eax
    //Result := 0;
```

```
  xor eax,eax
  mov [ebp-$0c],eax
  //for I := Start to Stop do
  mov eax,[ebp-$04]
  mov edx,[ebp-$08]
  sub edx,eax
  jl @LoopEnd
  //inc edx                          //NoName intialize NoName 初始化
  //mov [ebp-$14],edx                //NoName intialize NoName 初始化
  mov [ebp-$10],eax
   //begin
   @LoopStart :
      //Result := Result + I;
      mov eax,[ebp-$10]
      add [ebp-$0c],eax
      inc dword ptr [ebp-$10]
   //end;
   //dec dword ptr [ebp-$14]   //NoName decrement NoName 减一
   mov ebx, [ebp-$10]          //New
   mov ecx, [ebp-$08]          //New
   cmp ebx, ecx                //New
   //jnz @LoopStart
   jbe @LoopStart              //New
  @LoopEnd :
   mov eax,[ebp-$0c]
  mov esp,ebp
  pop ebx                      //New
  pop ebp
  //ret
end;
```

The lines marked "New" are introduced to make I the loop control variable.
The mov ebx, [ebp-$10] line copies I into ebx. The next line copies Stop into ecx.
Then the line cmp ebx, ecx compare them and jbe @LoopStart transfer execution to the start of the loop if I is below Stop or equal to it. Because we use ebx and it is not free for use we remember to push and pop it.
We expect a for loop to evaluate the loop condition at the top.
This test is split in two by the compiler implementation.
Before entering the loop it is tested that it will execute at least once and the actual loop closing test is done at the bottom. This is an optimization technique called loop inversion. Now we change the loop such that this optimization is removed. Then we see what was the benefit of the optimization.

被标记为 "New" 的行是用来制造 I 循环控制变量。
mov ebx, [ebp-$10] 将 I 复制到 ebx。下一行将 Stop 复制到 ecx。

cmp ebx, ecx 进行比较，如果 I 小于等于 Stop ，则 jbe @LoopStart 执行到循环的开始。
因为我们使用了 ebx，它不能被自由的使用，因此我们要记得 push 和 pop 它。
一个循环是在开始就计算循环条件的。
可以将编译器执行过程被分为两步。
在进入循环之前测试，它将至少被执行一次，然后在结尾判断是否循环结束。这种优化技术叫做循环倒置。
现在我们开始修改这个没被优化的循环，来看看优化的好处。

```
function ForLoopBASM4(Start, Stop : Integer) : Integer;
asm
  push ebp
  push ebx
  mov ebp,esp
  add esp,-$14
  mov [ebp-$08],edx
  mov [ebp-$04],eax
    //Result := 0;
    xor eax,eax
    mov [ebp-$0c],eax
    //for I := Start to Stop do
    mov eax,[ebp-$04]
    mov edx,[ebp-$08]
    //sub edx,eax
    //jl @LoopEnd
    mov [ebp-$10],eax
      //begin
      @LoopStart :
          mov ebx, [ebp-$10]
          mov ecx, [ebp-$08]
          cmp ebx, ecx
          ja    @LoopEnd
          //Result := Result + I;
          mov eax,[ebp-$10]
          add [ebp-$0c],eax
          inc dword ptr [ebp-$10]
      //end;
      //mov ebx, [ebp-$10]
      //mov ecx, [ebp-$08]
      //cmp ebx, ecx
      //jbe @LoopStart
      jmp @LoopStart
    @LoopEnd :
      mov eax,[ebp-$0c]
    mov esp,ebp
```

```
   pop ebx
   pop ebp
end;
```

The loop-closing test has been moved to the top and the test has been inverted.
At the place of the test there now is an unconditional jump to the top.
This jump is what the loop inversion optimization is all about.
In the unoptimized for loop there is two jumps and only one in the optimized one.
The test at the top that tested whether Start was above Stop is now redundant and is removed.
Before making any timing to evaluate the effect of the two optimisations it is a good idea to optimise away some,
or all if possible, of the stack to register and register to stack moves.

循环结束的检测被移到循环开始，做了一个反转。
在原先检测的位置用了一个无条件跳转到循环开始。
这个跳转说明了整个循环倒置优化。
在没有优化的循环里有两个跳转，在优化的循环里只有一个跳转。
在开头的测试已经测试了是否 Start，Stop 现在是冗余的因此被移除。
在比较这两个优化的效率之前，最好能尽可能的优化掉比如堆栈到寄存器，寄存器到堆栈的数据移动。

This process is called register allocation and is one of the most important optimisations on all architectures,
but it is even more important on the Intel architecture because of the low number of available registers.
If there is not a register available to all variables it is crucial which variables get a register.
The mov instructions inside the loop body are the most important ones to get rid of.
They are executed as many times as the number of loop iterations.
The instructions outside the loop are only executed once.
The variables used inside the loop should be allocated to registers first. This is I, Stop and Result.
At this point we could be smart and take a look at the use of registers as temporaries.
If a variable were always copied into the same temp register it would be smart to allocate this register for the variable.
Stop is in the edx register when we enter the function and this register is also used as temporary register for it in all but two lines. This is the two lines of the loop test that we added.

这个处理叫做寄存器分配，它是所有体系结构中最重要的优化之一。
因为寄存器数量很少，因此在 Intel 体系结构中它显得很重要。
如果寄存器不能满足所有的变量，因此哪个变量获得寄存器是相当重要的。
在循环体内的 mov 指令是最重要的一个，需要用到一个寄存器。
因为在循环迭代中，它们需要执行很多次。
循环以外的指令只执行一次。
用在循环内部的变量应该首先被分配寄存器。这有 I, Stop 和 Result。
在这一点上，我们可以动动脑筋，观察一下寄存器作为临时变量的使用。

如果一个变量总是被复制到相同的临时寄存器，它将被聪明的分配一个寄存器。
当我们进入函数时，Stop 在 edx 中，这个寄存器也总是作为临时寄存器使用，但是只有两行。
下面是我们为循环测试增加的两行：

Let us change 让我们把
mov ecx, [ebp-$08]
cmp ebx, ecx
to 改为
mov edx, [ebp-$08]
cmp ebx, edx

Eax is used by Start in the top of the function and by Result in the rest of the function.
If there is no overlap in usage we can allocate eax for Result as soon as Start has finished using it.
After Start is assigned to I (mov [ebp-$10], eax) it is not used any more and eax is free to use by Result,
if it was not for those lines where eax is used as temp for I.

Eax 在函数开始被作为 Start ，在函数结尾作为返回结果。
一旦 Start 用完它,如果再没有覆盖 eax 的操作了,那么可以将 eax 作为返回结果了。
在 Start 被赋值为 I (mov [ebp-$10], eax) 之后，如果没有那些将 eax 作为 I 的临时变量，则可以将 eax 作为返回结果了。

```
mov eax,[ebp-$10]
add [ebp-$0c],eax
inc dword ptr [ebp-$10]
```

After ecx got out of use by the last change we can use that as temp for I instead of eax.
在 ecx 不被使用之后，我们可以用它作为临时变量 I 来代替 eax 。

```
mov ecx,[ebp-$10]
add [ebp-$0c],ecx
inc dword ptr [ebp-$10]
```

The summary of the first part of the register allocation is: Result in eax, I in ecx and Stop in edx.
Lets first change the lines with Stop. [ebp-$08] is replaced by edx everywhere.
寄存器分配的第一部分的主题是： 结果在 eax 中, I 在 ecx 中, Stop 在 edx 中。
首先修改 Stop 部分，用 edx 替换所有的 [ebp-$08]。

```
function ForLoopBASM6(Start, Stop : Integer) : Integer;
asm
  push ebp
  push ebx
  mov ebp,esp
```

```
  add esp,-$14
  //mov [ebp-$08],edx
  mov edx,edx
  mov [ebp-$04],eax
    //Result := 0;
    xor eax,eax
    mov [ebp-$0c],eax
    //for I := Start to Stop do
    mov eax,[ebp-$04]
    //mov edx,[ebp-$08]
    mov edx,edx
    mov [ebp-$10],eax
      //begin
      @LoopStart :
          mov ebx, [ebp-$10]
          //mov edx, [ebp-$08]
          mov edx, edx
          cmp ebx, edx
          ja    @LoopEnd
          //Result := Result + I;
          mov ecx,[ebp-$10]
          add [ebp-$0c],ecx
          inc dword ptr [ebp-$10]
      //end;
      jmp @LoopStart
    @LoopEnd :
      mov eax,[ebp-$0c]
    mov esp,ebp
    pop ebx
    pop ebp
end;
```

Then allocate ecx for I by replacing [ebp-$10] by ecx.

用 ecx 作为 I 来替换 [ebp-$10]

```
function ForLoopBASM7(Start, Stop : Integer) : Integer;
asm
  push ebp
  push ebx
  mov ebp,esp
  add esp,-$14
  mov edx,edx
  mov [ebp-$04],eax
    //Result := 0;
    xor eax,eax
    mov [ebp-$0c],eax
```

```
  //for I := Start to Stop do
  mov eax,[ebp-$04]
  mov edx,edx
  //mov [ebp-$10],eax
  mov ecx,eax
    //begin
    @LoopStart :
      //mov ebx, [ebp-$10]
      mov ebx, ecx
      mov edx, edx
      cmp ebx, edx
      ja   @LoopEnd
      //Result := Result + I;
      //mov ecx,[ebp-$10]
      mov ecx,ecx
      add [ebp-$0c],ecx
      //inc dword ptr [ebp-$10]
      inc ecx
    //end;
    jmp @LoopStart
  @LoopEnd :
    mov eax,[ebp-$0c]
  mov esp,ebp
  pop ebx
  pop ebp
end;
```

最后，让 eax 存放结果。
因为 eax 在函数的开始用来传递了参数 Start，因此需要将它初始化为零。
如果 eax 没有再被用于其他目的，我们需要增加一行来将结果复制到 eax。

```
function ForLoopBASM8(Start, Stop : Integer) : Integer;
asm
 push ebp
 push ebx
 mov ebp,esp
 add esp,-$14
 mov edx,edx
 mov [ebp-$04],eax
  //Result := 0;
  xor eax,eax
  mov [ebp-$0c],eax
  //for I := Start to Stop do
  mov eax,[ebp-$04]
```

```
  mov edx,edx
  mov ecx,eax
  mov eax, [ebp-$0c]                    //New
    //begin
    @LoopStart :
      mov ebx, ecx
      mov edx, edx
      cmp ebx, edx
      ja   @LoopEnd
      //Result := Result + I;
      mov ecx,ecx
      //add [ebp-$0c],ecx
      add eax,ecx
      inc ecx
    //end;
    jmp @LoopStart
  @LoopEnd :
    //mov eax,[ebp-$0c]
    mov eax,eax
  mov esp,ebp
  pop ebx
  pop ebp
end;
```

Because we were pretty smart when we chose the registers there is a lot of lines like mov eax, eax. It is easy to see how redundant they are ;-). Let us remove them.
我们应该灵活的选择像 mov eax, eax 这样的寄存器操作行。
很容易发现他们是否冗余。移除它们：

```
function ForLoopBASM9(Start, Stop : Integer) : Integer;
asm
 push ebp
 push ebx
 mov ebp,esp
 add esp,-$14
 //mov edx,edx
 mov [ebp-$04],eax
  //Result := 0;
  xor eax,eax
  mov [ebp-$0c],eax
  //for I := Start to Stop do
  mov eax,[ebp-$04]
  //mov edx,edx
  mov ecx,eax
  mov eax, [ebp-$0c]
```

```
    //begin
    @LoopStart :
      mov ebx, ecx
      //mov edx, edx
      cmp ebx, edx
      ja   @LoopEnd
      //Result := Result + I;
      //mov ecx,ecx
      add eax,ecx
      inc ecx
    //end;
    jmp @LoopStart
  @LoopEnd :
    //mov eax,eax
  mov esp,ebp
  pop ebx
  pop ebp
end;
```

When optimizing ASM code there is generally two lines of thinking we can choose to follow.
We can think as the human beings we are and try to be smart, use whatever information we can get from the code.
We did some of this when we selected the registers here.
Another line of thought is trying to do it systematically as an optimizer/compiler has to.
This way we develop algorithms that can be coded in a tool.
This tool can later take over the most boring of optimizations, these we do over and over again.
Removal of the most obviously redundant line of code of all, the mov eax, eax, was an example of dead code removal,
which is basic to any optimiser.

当我们优化 ASM 代码是通常有以下两种考虑。
我们可以发挥我们的聪明才智尽可能的从这些代码中获取优化信息，就像我们上面所做的用寄存器处理。
另一个考虑是去实现一个系统的优化器或编译器。开发一个工具来定义优化规则。
这个工具可能代替大部分繁琐的优化，就是我们一而再，再而三做得这些人工优化。
一个优化器需要实现的基本功能：删除代码中所有的明显的冗余，mov eax, eax 是一行典型的无效代码。

In the top of the function we still have some references to the stack.
To get rid of these we allocate registers for those variables too.
This time we just pick edi and esi, which are not used elsewhere.
Allocate esi for [ebp-$04] and edi for [ebp-$0c].
Because esi and edi must be preserved by the function we must push and pop them.

在这个函数的开始我们仍然有一些对堆栈的引用。

为了去除这些，我们也为这些变量分配寄存器。
这时候我们可以使用 edi, esi ，他们还没有在别的地方被使用。
为 [ebp-$04] 分配 esi，为 [ebp-$0c] 分配 edi。
因为 esi, edi 是函数所保留的，因此必须 push 和 pop 它们。

```
function ForLoopBASM10(Start, Stop : Integer) : Integer;
asm
  push ebp
  push ebx
  push esi
  push edi
  mov ebp,esp
  add esp,-$14
  //mov [ebp-$04],eax
  mov esi,eax
    //Result := 0;
    xor eax,eax
    //mov [ebp-$0c],eax
    mov edi,eax
    //for I := Start to Stop do
    //mov eax,[ebp-$04]
    mov eax,esi
    mov ecx,eax
    //mov eax, [ebp-$0c]
    mov eax, edi
      //begin
      @LoopStart :
          mov ebx, ecx
          cmp ebx, edx
          ja    @LoopEnd
          //Result := Result + I;
          add eax,ecx
          inc ecx
      //end;
      jmp @LoopStart
    @LoopEnd :
    mov esp,ebp
    pop edi
    pop esi
    pop ebx
    pop ebp
end;
```

The stack frame is not used anymore and there is no need to set it up.
This removes 4 instructions.

Then we observe that the two lines
堆栈不再被使用，不需要设置它。删除这 4 行指令。
接下来，我们观察下面两行.

```
mov eax,esi
mov ecx,eax
```

Can be replaced by one.
可以被一行代替：

```
mov ecx, esi
```

This is an example of a simple copy propagation followed by dead code removal.
Any other lines do not use the value in eax than the next line that copies it back to ecx.
It is in fact immediately overwritten by the line mov eax, edi.
Therefore we can replace the second line by mov ecx,esi and remove the first one, which becomes dead.

这时一个关于移除无效代码复制繁殖的例子。
没有任何一行使用 eax 的值，在第二行又被拷贝回 ecx。
事实上，eax 马上就被 mov eax, edi 重写。
因此，我们用 mov ecx,esi 替换第二行，删除无效的第一行。

```
function ForLoopBASM11(Start, Stop : Integer) : Integer;
asm
  //push ebp
  push ebx
  push esi
  push edi
  //mov ebp,esp
  //add esp,-$14
  mov esi,eax
    //Result := 0;
    xor eax,eax
    mov edi,eax
    //for I := Start to Stop do
    //mov eax,esi
    //mov ecx,eax
    mov ecx, esi
    mov eax, edi
      //begin
      @LoopStart :
          //mov ebx, ecx
          //cmp ebx, edx
          cmp ecx, edx
```

```
        ja    @LoopEnd
        //Result := Result + I;
        add eax,ecx
        inc ecx
      //end;
      jmp @LoopStart
    @LoopEnd :
    //mov esp,ebp
    pop edi
    pop esi
    pop ebx
    //pop ebp
end;
```

The line xor eax, eax that initialises Result to zero can together with the line right after it be moved some lines down closer to the place where eax is used for the first time. It should not be moved into the loop that would change the logic of the function, but just before the line before loopStart.
This removes the need for copying eax into edi and back into eax again in the line just before the comment line //for I := Start to Stop do, and in the line before the out commented begin.

xor eax, eax 将结果初始化为零。它应当被放在第一次使用 eax 的地方。
它不应该放入循环中，因为会改变函数的逻辑，仅用于 loopStart 之前。
需要移除注释行 //for I := Start to Stop do 之前的复制 eax 到 edi ，然后再复制回 eax 的代码。

```
function ForLoopBASM12(Start, Stop : Integer) : Integer;
asm
  push ebx
  push esi
  push edi
  mov esi,eax
    //for I := Start to Stop do
    mov ecx, esi
    //Result := 0;
    xor eax,eax
    //mov edi,eax
    //mov eax, edi
      //begin
      @LoopStart :
        cmp ecx, edx
        ja    @LoopEnd
        //Result := Result + I;
        add eax,ecx
        inc ecx
```

```
    //end;
    jmp @LoopStart
  @LoopEnd :
  pop edi
  pop esi
  pop ebx
end;
```

After having cleaned up we spot two lines of mov which together copy eax to ecx via esi.
This leaves a copy of eax in esi, which is not used. Therefore one that moves eax directly into ecx can replace
these two lines. This is also copy propagation + dead code removal.
清除之后我们发现两行 mov 指令，是将 eax 通过 esi 复制到 ecx。
这样在 esi 中留下了一个 eax 的拷贝，它没有被使用。
因此，用直接将 eax 移入 ecx 的方法替换这两行。
这也是一个复制繁殖 + 无效代码移除。

```
function ForLoopBASM13(Start, Stop : Integer) : Integer;
asm
 push ebx
 //push esi
 push edi
 //mov esi,eax
  //for I := Start to Stop do
  //mov ecx, esi
  mov ecx, eax
  //Result := 0;
  xor eax,eax
    //begin
    @LoopStart :
       cmp ecx, edx
       ja   @LoopEnd
       //Result := Result + I;
       add eax,ecx
       inc ecx
    //end;
    jmp @LoopStart
  @LoopEnd :
  pop edi
  //pop esi
  pop ebx
end;
```

After having removed the only use of esi there is no need to push and pop it.
移除之后只需要使用 esi，不需要 push 和 pop 它。

```
function ForLoopBASM14(Start, Stop : Integer) : Integer;
asm
  //push ebx
  //push edi
    //for I := Start to Stop do
    mov ecx, eax
    //Result := 0;
    xor eax,eax
      //begin
      @LoopStart :
          cmp ecx, edx
          ja    @LoopEnd
          //Result := Result + I;
          add eax,ecx
          inc ecx
      //end;
      jmp @LoopStart
    @LoopEnd :
    //pop edi
    //pop ebx
end;
```

We also, a little late perhaps, observe that ebx and edi is not used either. After having cleaned up and relocated the comments a little, there is a nice clean function as a result.

还有一个可能，就是我们发现 ebx 和 edi 也没有被使用。

整理代码，并进行少量注释之后，一个简洁的函数就产生了：

```
function ForLoopBASM15(Start, Stop : Integer) : Integer;
asm
    mov ecx, eax
    //Result := 0;
    xor eax,eax
    //for I := Start to Stop do
  @LoopStart :
    cmp ecx, edx
    ja    @LoopEnd
    //Result := Result + I;
    add eax,ecx
    inc ecx
    jmp @LoopStart
  @LoopEnd :
end;
```

It took a long time and a lot of optimisations to get here because we started with the non-optimised output from

the compiler. This long process illustrated the amount of work the compiler leaves to the optimizer.
Sometimes we did not use the algorithmic approach to optimization,
but we could have achieved the same result by doing it.
Instead of taking the same long road again with the function with loop inversion present,
we can cheat and compile the Pascal function with optimizations on.
The compiler might do all the optimisation we did.
经过了长时间和大量的优化我们终于得到这个函数。当然，这是因为我们在开始的时候没有选择优化选项。
这个长过程展示了编译器将大量的工作留给了优化器。
有时，我们不使用优化选项，但是我们可以自己优化它，从而得到相同优化结果。
我们可以打开编译器优化选项来优化代码，编译器内部可能也使用了我们用的循环倒置来进行这么长的优化步骤。

```
function ForLoopOpt(Start, Stop : Integer) : Integer;
var
  I : Integer;

begin
{
}
  Result := 0;
  {
  xor ecx,ecx
  }
  for I := Start to Stop do
  {
  sub edx,eax
  jl +$08
  inc edx
  xchg eax,edx
  }
    begin
      Result := Result + I;
      {
      add ecx,edx
      inc edx
      }
    end;
    {
    dec eax
    jnz -$06
    }
{
```

```
mov eax,ecx
}
end;
```

This time Delphi did a really nice job. Only two lines jump into our eyes as possibly redundant.
xchg eax, edx simply exchange the values in eax and edx and mov eax, ecx copy the Result into eax.
Both lines are outside the loop and make little harm.
这次 Delphi 做的真实非常的好。在我们看来，只有两行冗余的跳转代码。
xchg 简单的交换 eax, edx 的值，mov eax, ecx 将结果复制到 eax 中。
这两行在循环的外面，会有一点效率上的损害。

We now have two functions - one with no loop optimizations and one with two.
To make things complete we need two more functions, one with loop inversion only and one with the NoName variable optimization only.
In the beginning of the lesson we saw how to remove the two optimisations and this is what
I have done to get to the last 2 functions.
In the Delphi optimized function above, I optimized away the xchg instruction by swapping the use of
the two registers it exchanged.
Because we want to se the maximum effect of the loop optimizations I have removed the loop body code doing
the operation Result := Result + I;
Here are the four final functions

现在，我们有两个函数 - 没有循环优化有一个，循环优化的有两个。
为了完成我们要讲的，我们还需要两个函数。一个是只有循环倒置,一个是只有 NoName 变量优化.
在这课的开始我们已经讲了如何进行这两种优化，并写了两个函数.
在 Delphi 优化函数的基础上，我通过使用两个寄存器进行交换来优化掉 xchg 指令.
因为我们想看看循环体 Result := Result + I 的最高循环效率。
下面是四个最终的函数：

```
function ForLoopNoLoopInverNoNoName(Start, Stop : Integer) : Integer;
asm
   mov ecx, eax
   //Result := 0;
   xor eax,eax
   //for I := Start to Stop do
  @LoopStart :
   cmp ecx, edx
   ja    @LoopEnd
   inc ecx
   jmp @LoopStart
  @LoopEnd :
```

end;
循环由 4 个指令组成。cmp, ja, inc 和 jmp。
在 P4 上这些指令的潜伏期和吞吐量是：cmp 0.5/0.5, ja X/0.5, inc 0.5/1 and jmp X/0.5 。
X 表示 "潜伏期不能应用于这条指令"。
累加潜伏期，我们得到：0.5 + X + 0.5 + X = ？ 时钟周期

```
function ForLoopLoopInverNoNoName(Start, Stop : Integer) : Integer;
asm
   mov ecx, eax
   //Result := 0;
   xor eax,eax
   //for I := Start to Stop do
   cmp ecx, edx
   ja    @LoopEnd
  @LoopStart :
   inc ecx
   cmp ecx, edx
   jbe @LoopStart
  @LoopEnd :
end;
```

This loop consists of 3 instructions also with unknown sum of latency.
这个循环由三条指令组成，也不知道潜伏期的和。

```
function ForLoopNoLoopInverNoName(Start, Stop : Integer) : Integer;
asm
   //Result := 0;
   xor ecx,ecx
   //for I := Start to Stop do
   sub edx,eax
   cmp edx, 0
  @LoopStart :
   jz    @LoopEnd
   inc eax
   dec edx
   jmp @LoopStart
  @LoopEnd :
   mov eax,ecx
end;
```

This loop consists of 4 instructions also with unknown sum of latency.
We observe that the two inc/dec instructions are able to execute in parallel.
Because the dec NoName instruction is not followed by the conditional jmp it would look like we throw away the benefit of not needing a cmp or test instruction to set the flags, but the jmp instruction does not change the flags and they are valid when we reach the jz instruction at the top of the loop.

Only at the first iteration a cmp edx,0 instruction is needed.
这个循环由 4 条指令组成，也不知道潜伏期的和。
我们发现 inc/dec 指令能够并行执行。
因为 dec NoName 指令后面不是一个条件 jmp，因此不需要一个 cmp 或 test 指令来设置标志。
又因为 jmp 指令不能改变标志，因此当到达循环的开始 jz 指令时，它们都是正确的。
只是在迭代的开始需要一个 cmp edx,0 指令。

```
function ForLoopLoopInverNoName(Start, Stop : Integer) : Integer;
asm
   //Result := 0;
   xor ecx,ecx
   //for I := Start to Stop do
   sub edx,eax
   jl @LoopEnd
   inc edx
  @LoopStart :
   inc eax
   dec edx
   jnz @LoopStart
  @LoopEnd :
   mov eax,ecx
end;
```

这个循环由 3 条指令组成，也不知道潜伏期的和。
这也有一对独立的 inc/dec。

This is the simple benchmark I have used to find the performance of the 4 functions
我使用下面的基准测试方法来检测 4 函数的性能。

```
const
  LOOPSTART : Integer = 1;
  LOOPEND : Integer = 2000000000;
  LOOPEND2 : Integer = 20;
  CLOCK : Double = 1920E6;

var
  Starttime, Endtime, Runtime : TDateTime;
  I, LoopResult : Integer;
  RunTimeSec, NoOfLoopsPerSec, NoOfLoops, ClockCount, LoopEnd2Float, LoopEndFloat,
LoopStartFloat : Double;

begin
  Starttime := Time;
  for I := 1 to LOOPEND2 do
```

```
    begin
      LoopResult := ForLoopNoLoopInverNoName(LOOPSTART, LOOPEND);
    end;
  Endtime := Time;
  Runtime := Endtime - Starttime;
  CEdit.Text := IntToStr(LoopResult);
  RuntimeEdit4.Text := TimeToStr(Runtime);
  RunTimeSec := RunTime*24*60*60;
  LoopEnd2Float := LOOPEND2;
  LoopEndFloat := LOOPEND;
  LoopStartFloat := LOOPSTART;
  NoOfLoops := LoopEnd2Float * (LoopEndFloat - LoopStartFloat);
  NoOfLoopsPerSec := NoOfLoops / RunTimeSec;
  ClockCount := CLOCK / NoOfLoopsPerSec;
  ClockCountEdit4.Text := FloatToStrf(ClockCount, ffFixed, 9, 1);
end;
```

Results on P4 1920 are 在 P4 1920 上的结果
No Loop Inversion and No NoName variable 00:00:55 (2.7 Clock cycles)
无循环倒置，无未命名变量
Loop Inversion but No NoName variable 00:00:39 (1.9 Clock cycles)
循环倒置，无未命名变量
No Loop Inversion but NoName variable 00:01:02 (3.0 Clock cycles)
无循环倒置，有未命名变量
Loop Inversion + NoName 00:00:41 (2.0 Clock cycles)
循环倒置，有未命名变量

Results on P3 1400 are 在 P3 1400 上的结果
No Loop Inversion and No NoName variable 00:01:26 (3.0 Clock cycles)
无循环倒置，无未命名变量
Loop Inversion but No NoName variable 00:01:26 (3.0 Clock cycles)
循环倒置，无未命名变量
No Loop Inversion but NoName variable 00:01:55 (4.0 Clock cycles)
无循环倒置，有未命名变量
Loop Inversion + NoName 00:01:26 (3.0 Clock cycles)
循环倒置，有未命名变量

Of course the clock count numbers should be integers.
On P4 half cycles are possible due to the double-clocked ALU.
Our timings are not as good as we could wish, but for comparing the performance of the loops they are OK.
Conclusion on P4. Apply loop inversion only or loop inversion with NoName variable optimization.
Conclusion on P3. Do not apply the NoName variable optimization alone.

Conclusion on both targets. Apply both optimizations as Delphi does.
Also observe how efficient P4 is on this code.
当然，时间计数值应该是整数。
由于双时钟的 ALU, 因此在 P4 上可能有半个时钟周期。
我们的测试没有我们想象的那样好，但是比较循环的执行效率足够了。
在 P4 上，只应用了循环倒置和使用未命名(NoName)变量的循环倒置优化。
在 P3 上，没有单独的应用未命名变量优化。
结果是两个平台上的。Delphi 做了这两种优化。用这个代码也可以观察在 P4 上运行的效率。

Lesion 6 第 6 课(CharPos)

This is lesson number 6 and the topic is CharPos.
第 6 课的主题是 CharPos。

This function search for the first occurrence of a character in a string,and returns the position of it when found.
If nothing is found it returns zero. The Delphi library function does basically the same thing,
but with the difference that it search for a substring in a string.
Passing a character to Pos as the substring is possible and this way use Pos as a CharPos.
In this lesson we will develop a CharPos that is nearly 4 times faster than Delphi's Pos.
As usual we start with a Pascal implementation of the algorithm.

这个函数搜索一个字符串中的字符，并且返回发现的位置，如果没有发现则返回零。
Delphi 库也有类似的函数，但是不同的是，Delphi 是在一个串中搜索一个子串。
给 Pos 传输一个字符作为子串也可以像 CharPos 一样使用 Pos。
在这一课中我们将编写一个 CharPos, 它比 Delphi 的 Pos 快将近 4 倍。
照常，我们从一个 pascal 算法讲起。

```
function CharPos2(Chr : Char; const Str : AnsiString) : Cardinal;
var
  I : Integer;
begin
  if (Str <> '') then
    begin
      I := 0;
      repeat
        Inc(I);
      until((Str[I] = Chr) or (Str[I] = #0));
      if (Str[I] <> #0) then
        Result := I
      else
```

```
      Result := 0;
    end
  else
    Result := 0;
end;
```

Input to the function is a Char and a string. The string is passed as const. The functions Result are a Cardinal.
First the function checks that the string is not empty and if is empty it returns zero.
If there is a string we iterate over it with a repeat until loop searching for a match to the input char.
If an occurrence of the char is not found before we reach the zero terminator this will also terminate the loop.
Because the loop can terminate on either of the two conditions it is necessary to check
what terminated the search before we know what to return as result.
If the loop was ended by a successful search we return the value of the loop counter, and if the search was unsuccessful the result is zero.It is also possible to use the length of the string as a condition for terminating the loop on an unsuccessful search.
Then the code looks like this.

函数的输入是一个 Char 和 string，string 是常量，函数返回一个 Cardinal。
首先，函数检查一个字符串是否为空，如果空，则返回零。
使用一个 repeat until 循环在字符串中搜索匹配的输入字符，直到迭代结束。
在到达零终止字符之前，如果没有发现这个字符，那么循环也将被终止。
因为循环可以被两个条件中的任何一个终止，所以在返回结果之前需要检查是谁终止了搜索。
如果成功搜索到此字符，将返回循环计数，如果没找到，则返回零。

也可以使用字符串的长度来结束一个没有成功搜索到字符的循环，代码如下：

```
function CharPos1(Chr : Char; const Str : AnsiString) : Cardinal;
var
  StrLenght, I : Integer;
begin
  StrLenght := Length(Str);
  if StrLenght > 0 then
    begin
      I := 0;
      repeat
        Inc(I);
      until((Str[I] = Chr) or (I > StrLenght));
      if I <= StrLenght then
        Result := I
      else
        Result := 0;
    end
```

```
  else
    Result := 0;
end;
```

Before we start jumping into ASM it is a good idea to see which of the Pascal implementations is the fastest.
For doing this benchmark must be established.
在我们开始进入 asm 之前，最好先看看哪个 pascal 代码执行的最快。
建立如下测试基准：

```
const
  NOOFLOOPS : Cardinal = 200000;
  SCALE : Cardinal = 1000;

procedure Benchmark;
var
  lpPerformanceCount, StartCount, EndCount : TLargeInteger;
  Succes : Boolean;
  Str, Str1, FunctionName : AnsiString;
  Chr1, Chr2 : Char;
  I, CharPos, J, K, Bench, SumBench : Cardinal;
  StringArray : array[1..255] of AnsiString;

begin
  Series1.Clear;
  Str1 := 'T';
  for J := 1 to 255 do
    begin
        StringArray[J] := Str1;
        Str1 := 'A' + Str1;
    end;
  SumBench := 0;
  Chr1 := 'T';
  Chr2 := 'X';
  for K := 1 to 255 do
    begin
      Str := StringArray[K];
      Succes := QueryPerformanceCounter(lpPerformanceCount);
      if Succes then
        StartCount := lpPerformanceCount
      else
        raise Exception.Create('QueryPerformanceCounter failed');
      for I := 1 to NOOFLOOPS do
        begin
          CharPos := CharPosFunction(Chr1, Str);
```

```
      end;
    for I := 1 to NOOFLOOPS do
      begin
        CharPos := CharPosFunction(Chr2, Str);
      end;
    Succes := QueryPerformanceCounter(lpPerformanceCount);
    if Succes then
      EndCount := lpPerformanceCount
    else
      raise Exception.Create('QueryPerformanceCounter failed');
    Bench := Round((EndCount - StartCount) / SCALE);
    Series1.AddXY(K, Bench, '', clBlue);
    Bench := Round(Bench / K);
    SumBench := SumBench + Bench;
    Update;
  end;
 FunctionName := FunctionSelectRadioGroup.Items[FunctionSelectRadioGroup.ItemIndex];
 ReportRichEdit.Lines.Add(FunctionName + #9 + IntToStr(SumBench));
end;
```

The benchmark builds an array of 255 AnsiStrings. The first string is 'T'. 'T' is also the character we use as the search character. String number 1 is therefore the shortest possible successful match. The next strings are 'AT', 'AAT' and 'AAAT'. I guess the pattern is clear. It is equally important to measure the performance on unsuccessful search. For this purpose we pass 'X' as search character. The benchmark does NOOFLOOPS searches on each string and measures the time used on each string. Because we want results from each string length to contribute approximately the same to the benchmark each timing is divided by the length of the string.
On this benchmark CharPos1 obtains the score 767 on a P4 1600A clocked at 1920 and CharPos2 obtains the score 791.For comparison Delphi Pos scores 2637.
Because CharPos1 is slightly better than CharPos2 we select this as basis for optimisation.
This is the ASM code Delphi 6 compiled it into with optimizations turned on.

这个基准测试创建了一个大小为 255 的字符串数组。
第一个串是 'T'。'T' 也是我们要搜索的字符。第一个串成功匹配的最短的串。
接下来的串是 'AT', 'AAT' 和 'AAAT'。
我想这个形式已经是清楚的。
测量不成功的搜索性能也是一样重要的。为了这个目的，我们传入 'X'，作为要搜索的串。
基准循环搜索每个串，并测量在每个串上使用的时间。
(Q:)因为我们想返回每一个字符串的长度去近似的表示基本每次都被字符串的长度相除。
在 P4 1600A 主频是 1920 上进行基准测试，CharPos1 获得了 767 分，CharPos2 获得 791 分，
Delphi 中的 Pos 是 2637 分。
因为 CharPos1 比 CharPos2 效率高些，因此我们选择 Charpos1 进行优化。
打开 Delphi6 的优化选项编译，ASM 代码如下：

```
function CharPos14(Chr : Char; const Str : AnsiString) : Cardinal;
var
  StrLenght, I : Integer;

begin
  {
  push ebx
  push esi
  mov    esi,edx
  mov    ebx,eax
  }
  StrLenght := Length(Str);
  {
  mov    eax,esi
  call @LStrLen
  mov    edx,eax
  }
  if StrLenght > 0 then
  {
  test edx,edx
  jle    @Else1Begin
  }
    begin
      I := 0;
      {
      xor eax,eax
      }
      repeat
        {
        @RepeatBegin :
        }
        Inc(I);
        {
        inc eax
        }
      until((Str[I] = Chr) or (I > StrLenght));
      {
      cmp bl,[esi+eax-$01]
      jz    @If2
      cmp edx,eax
      jnl @RepeatBegin :
      }
      if I <= StrLenght then
```

```
    {
    @If2 :
    cmp edx,eax
    jnl @Exit
    }
      Result := I
      {
      }
    else
      Result := 0;
      {
      xor eax,eax
      pop esi
      pop ebx
      ret
      }
  end
 else
  Result := 0;
  {
  @Else1Begin :
  xor eax,eax
  }
 {
 @Exit :
 pop esi
 pop ebx
 }
end;
```

This time there is no stack frame. Ebx and esi is used and needs to be backed up and restored by pushing and
popping them on the stack.
Because the function does not have its own stack frame they are simply pushed on the stack on the top of the
calling functions frame.

这一次没有分配堆栈。Ebx 和 esi 被使用，需要通过 push 和 pop 来保存和恢复它们。
因此这个函数没有自己的堆栈，因此在此函数的开始，它们被简单的压入堆栈。

The input parameters are received in eax an edx and they are as first thing copied
into esi and ebx. The function Length has a secret internal name, which is LStrLen. This function is called on Str, which is passed in eax. We see that LStrLen is following the register calling convention.

输入参数在 eax 和 edx 中，它们首先被复制到 esi 和 ebx。
Length 函数有个秘密的内部名字 LStrLen。这个函数操作 Str 时用到，通过 eax 传递。
LStrLen 遵循如下寄存器调用约定。

Str was received in edx copied to esi and then to eax. LStrLen delivers its result, StrLenght, in eax. This result is copied into edx and compared to 0. Test edx, edx is the same as cmp edx,0 and it sets the flags. The jle instruction checks the flags and pass execution to the else part of the if-else code if StrLenght is lower than or equal to zero.

Str 通过 edx 接收后复制到 esi，然后再到 eax 中。LStrLen 通过 EAX 返回它的结果 StrLenght。
结果复制到 edx 中与 0 比较。Test edx, edx 与 cmp edx,0 相同都设置状态寄存器标志。
jle 检测标志，如果 StrLenght 小于等于零, 则执行 if-else 的 else 部分。

In this else code we have one line of Pascal, which is Result := 0;.
Because our function must return its result in eax we create a zero here by x-oring eax by itself.
If the string is longer than zero execution is continued in the if code.

在 else 代码中有一行 Pascal 代码 Result := 0。
因为我们的函数必须在 eax 中返回结果，因此我们通过异或 eax 自己来制造一个零。
如果字符串长度大于零，则继续执行 if 代码。

The first line here initialises the loop counter I to zero.
Again a xor instruction is used for this.
The loop body has one line only and this is easy to understand Inc(I); = inc eax.
Pascal and ASM is nearly the same thing ;-)
The implementation of the loop-closing test is where the real meat of this function is.
It is made up of these four lines of ASM.

第一行初始化循环计数为 I 为零。又使用了一个 xor 指令。
循环体只有一行，很容易明白 Inc(I); = inc eax。
Pascal 和 ASM 代码是相当类似。
循环结束时测试函数是在哪里结束的。它由 4 行 ASM 代码组成。

```
cmp bl,[esi+eax-$01]
jz   @If2
cmp edx,eax
jnl @RepeatBegin :
```

We see that there is two jump instructions.
The last one jumps to the start of the loop and the first one jmps out of the loop.
There are two cmp instructions to set the flags. Number two is the easiest to understand.
It compares eax with edx. A fast look at the Pascal code tells us that
StrLenght and I must be in these two registers.

In fact we have just seen that eax is I and we also saw in the top of the function that StrLenght was in edx.

我们看到有两个跳转指令。
最后一个跳到循环的开始，第一个跳出循环。
通过两个 cmp 指令设置标志。 两个数容易理解。
比较 eax 和 edx。Pascal 代码告诉我们 StrLenght 和 I 必须在这两个寄存器。
事实上，我们已经看到 eax 是 I，在函数的开始我们能看到 StrLenght 在 edx 中。

In line number 4 Chr was copied into ebx, but a char is only one byte.
Therefore the first cmp instruction compares something to bl, which is the lowest byte of the four bytes in ebx.We expect that the search character - Chr - be compared to character number 1, 2, 3.. of the input string.
[esi+eax-$01] must therefore be a pointer into this string.
eax is the loop counter I, which is 1 at the first iteration. esi must be the address of the Str parameter that was received in edx and immediately copied into esi.
-$01 is a constant offset, which is needed because the first character in an AnsiString is located at position 0.
The position where Str points to.

在第 4 行 Char 被复制到 ebx，但是一个字符只有一个字节。
因此，第一个比较指令与 bl 比较，它是 ebx 4 个字节中最低的字节。
我们希望通过比较输入字符串的第 1,2,3 ...个字符来搜索 Chr。
[esi+eax-$01] 是指向这个字符串的指针。
eax 是循环计数 I，在迭代的开始置为 1。esi 是字符串参数的地址，它是通过 edx 接收后立即复制到了 esi。
-$01 是一个常量偏移，它是必需的，因为一个 AnsiString 的第一个字符是在位置 0。
Str 指向这个位置。

Where has the or from the Pascal code gone?
To understand this we need to talk about the optimisation called incomplete Boolean evaluation.
This optimisation can be applied to the Boolean operator and, and to the Boolean operator or.
Let us take a look at the truth table for and first.

Pascal 代码将执行到哪里呢？
为了明白这个，需要讨论不完全布尔计算优化。
这个优化可以被应用到布尔操作符 and 或 or。
首先，我们来看一下 and 的真值表

false and false is false
false and true is false
true and false is false
true and true is true

The and operator is only returning true if both operands are true.
This is taken advantage of by the incomplete Boolean evaluation optimisation.
If one operand is found false there is no need to check the second one, because the result of and will be false no matter what it is.

只有两个操作数都是 true, And 操作符才会返回 true。
不完全布尔计算的优点是，如果有一个操作数为 false，则不需要检测第二个，因为无论它是什么结果都是 false。

The truth table for or is
or 的真值表如下：
false or false is false
false or true is true
true or false is true
true or true is true
The result of an or is true if one or both of the operands are true.
If one is seen to be true there is no need to check the second operand.

如果有一个或者两个都为 true，or 的结果就是 true。
如果有一个是 true，不需要检测第二个操作数。

Our loop-closing test takes advantage of this by jumping out of the loop if the first compare is successful.
This is the case if we found a match for the search character in the string.
If it was a match it does not make sense to see if it was the zero terminator too!
The last compare is only executed on unsuccessful matches.
If we knew something about the strings and characters our function was called upon most often we could take advantage
of it by changing the order of the tests such that the most often true one is located first.

在循环结尾我们可以利用这个特点检测，如果第一个比较是成功的，那么就跳出循环。
这是在字符串中找到了匹配字符的情况。
如果它也是零终止的，那么去匹配是没有意义。
最后一个比较只在没有找到匹配字符时执行。
如果我们知道调用函数时经常传入的字符串和字符，我可以利用这个特点改变比较的顺序，将经常是 true 的比较放在前面。

Try switching on the compiler switch "complete Boolean evaluation", in project options,
and see what code is created then.
The rest of the code resample what we have seen in earlier lessons and
I will skip the explanation and go get a cup of coffee instead ;-)
Now it is time to do some optimizations. At first the function is made pure BASM.
The labels were introduced in the previous code listing.
Here it is seen that they follow the Pascal code in an intuitive way.

在工程选项中打开 “完全布尔检测”编译开关，看看代码发生了什么变化。
这部分代码我们已经在前面的课程中见到了，因此我不再解释它，我要去拿一杯咖啡了 ;-)
让我们来做一些优化。首先将函数整理为纯 BASM。
在前面的代码列表中已经介绍标签。
我们用直观的方式定义了这些标签。

```
function CharPos15(Chr : Char; const Str : AnsiString) : Cardinal;
//var
  //StrLenght, I : Integer;
asm
  push ebx
  push esi
  mov    esi,edx
  mov    ebx,eax
  //StrLenght := Length(Str);
  mov    eax,esi
  call System.@LStrLen
  mov    edx,eax
  //if StrLenght > 0 then
  test edx,edx
  jle    @Else1Begin
  //I := 0;
  xor    eax,eax
  //repeat
  @RepeatBegin :
  //Inc(I);
  inc    eax
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    bl,[esi+eax-$01]
  jz      @If2
  cmp    edx,eax
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  cmp    edx,eax
  jnl    @Exit
  //Result := I
  //else
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
```

```
  //else
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  @Exit :
  pop    esi
  pop    ebx
end;
```

The call to LStrLen has been prefixed with System, otherwise the compiler will not recognize it.
LStrLen is implemented in the System unit.
The var section is removed because we do not reference any variables by name.

函数 LStrLen 加上了 System 前缀，否则编译器不认识它。
LStrLen 在 System 单元中实现。
var 区被删除了，因为我们没有引用任何变量。

```
function CharPos16(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push esi
  mov    esi,edx
  mov    ebx,eax
  //StrLenght := Length(Str);
  mov    eax,esi
  //call System.@LStrLen
  //**************
  test eax,eax
  jz    @LStrLenExit
  mov eax,[eax-$04]
  @LStrLenExit :
  //**************
  mov    edx,eax
  //if StrLenght > 0 then
  test edx,edx
  jle @Else1Begin
  //I := 0;
  xor    eax,eax
  //repeat
  @RepeatBegin :
  //Inc(I);
  inc    eax
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    bl,[esi+eax-$01]
  jz      @If2
  cmp    edx,eax
```

```
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  cmp    edx,eax
  jnl    @Exit
  //Result := I
  //else
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  //else
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  @Exit :
  pop    esi
  pop    ebx
end;
```

The first thing we do is to inline LStrLen. This is done by tracing into it and copying the body of the function
from the cpu view. It is made up of these four lines.

首先我们来介绍内联函数 LStrLen。
我们在 cpu 窗口中跟进此函数，复制这个函数内部实现。它是由下面四行组成：

```
test eax,eax
jz +$03
mov eax,[eax-$04]
ret
```

If a nil pointer is passed to LStrLen in eax nothing is done but returning. If the pointer is valid the length of the string is found at the 4 bytes preceding the start of the string. These 4 bytes are returned in eax. To inline it we replace the call with the 4 lines.

如果给 LStrLen 的 eax 传入了一个 nil 指针，除了返回之外，什么也不做。
如果指针是合法，在这个字符串起始的前面 4 个字节中有这个字符串的长度。
这 4 个字节用 eax 返回。使用它来代替这 4 行代码。

Jz is transferring execution to the ret instruction.
Instead of this ret instruction we place a label called LStrLenExit.
The ret returned execution to the line after the call of the function.
This ret has to be removed, otherwise it would pass execution to the function that called CharPos,

not exactly what we want.

Jz 调到 ret 指令。我们用一个叫做 LStrLenExit 的标签来代替 ret。
ret 返回到调用本函数的函数的下一行。
ret 必须被删除，否则它将使函数 CharPos 返回，这不是我们想要的。

This is what the inlined LStrLen ended up looking like
内联 LStrLen 修改为如下：

```
test eax,eax
jz    @LStrLenExit
mov eax,[eax-$04]
@LStrLenExit :
```

```
function CharPos17(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push esi
  mov    esi,edx
  mov    ebx,eax
  //StrLenght := Length(Str);
  mov    eax,esi
  //**************
  test eax,eax
  //jz    @LStrLenExit
  jz    @Else1Begin
  mov eax,[eax-$04]
  //@LStrLenExit :
  //**************
  mov    edx,eax
  //if StrLenght > 0 then
  //test edx,edx
  //jle @Else1Begin
  //I := 0;
  xor    eax,eax
  //repeat
  @RepeatBegin :
  //Inc(I);
  inc    eax
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    bl,[esi+eax-$01]
  jz      @If2
  cmp    edx,eax
  jnl    @RepeatBegin
  //if I <= StrLenght then
```

```
  @If2 :
  cmp    edx,eax
  jnl    @Exit
  //Result := I
  //else
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  //else
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  @Exit :
  pop    esi
  pop    ebx
end;
```

现在我们来看 pascal 代码 if Strlenght>0，它和内联 LStrLen 第一行检测相同的东西。
必须检测是一次 Str 是 nil 的情况。
删除第二行，将第一行修改为调到 @Else1Begin，来替换 StrLen 函数中如果 Str 是 nil 则跳出。
LStrLenExit 标签不需要了。

```
function CharPos18(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push esi
  mov    esi,edx
  mov    ebx,eax
  //StrLenght := Length(Str);
  //mov    eax,esi
  //if StrLenght > 0 then
  //test eax,eax
  test esi,esi
  jz    @Else1Begin
  //mov eax,[eax-$04]
  mov eax,[esi-$04]
  mov    edx,eax
  //I := 0;
  xor    eax,eax
  //repeat
  @RepeatBegin :
```

```
  //Inc(I);
  inc    eax
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    bl,[esi+eax-$01]
  jz     @If2
  cmp    edx,eax
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  cmp    edx,eax
  jnl    @Exit
  //Result := I
  //else
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  //else
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  @Exit :
  pop    esi
  pop    ebx
end;
```

We moved the test of StrLenght = 0 and the comment //if StrLenght > 0 then must be moved too.
After the inlining it becomes possible to copy propagate esi in these lines.
我们移动了 StrLenght = 0 的测试，注释 if StrLenght > 0 也必须被移动。
在内联之后，下面这些行变成了重复复制 esi。

```
mov    eax,esi
//*************
test eax,eax
jz    @Else1Begin
mov eax,[eax-$04]
```

The last of the lines overwrites eax and is the last use of the value in eax that was copied from esi.
代码的最后重写了 eax，这是从 esi 复制之后最后一次使用 eax 的值。

```
mov    eax,esi
//*************
//test eax,eax
test esi,esi
jz    @Else1Begin
```

```
//mov eax,[eax-$04]
mov eax,[esi-$04]
```

Actually we must also follow the possible branch to Else1Begin and see whether the value in eax is used here.
实际上，我们也必须看看分支 Else1Begin 处是否使用了 eax 的值。

We see that eax is zeroed out in the line immediately after the label and therefore is not used.
This way the first line is seen to be dead and can be removed.
我们看到 eax 在这个标签之后被清零，因此没有再被用。
第一行是无用的，可以删除掉

```
//mov    eax,esi
test esi,esi
jz    @Else1Begin
mov eax,[esi-$04]
```

```
function CharPos19(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push esi
  mov    esi,edx
  mov    ebx,eax
  //if StrLenght > 0 then
  test esi,esi
  jz    @Else1Begin
  //StrLenght := Length(Str);
  //mov eax,[esi-$04]
  mov edx,[esi-$04]
  //mov    edx,eax
  //I := 0;
  xor    eax,eax
  //repeat
  @RepeatBegin :
  //Inc(I);
  inc    eax
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    bl,[esi+eax-$01]
  jz      @If2
  cmp    edx,eax
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  cmp    edx,eax
  jnl    @Exit
```

```
  //Result := I
  //else
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  //else
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  @Exit :
  pop    esi
  pop    ebx
end;
```

Also as a result of inlining LStrLen we can remove one more mov.
LStrLen returned its result in eax and then it was copied into edx.
mov eax, [esi-$04] can then be changed to mov edx, [esi-$04] and the mov edx, eax can be removed.After this we change focus to the loop. This is far more important because it is executed many more times,depending on the length of the string or the position of a match.

内联了 LStrLen 之后我们可以删除更多的 mov 指令。
LStrLen 用 eax 返回结果，然后又被复制到 edx。
mov eax, [esi-$04] 可以被改为 mov edx, [esi-$04]，mov edx, eax 可以被删除了。

之后我们将焦点转向循环。这是更重要的，因为循环会执行会很多次，它依赖于字符串的长度和那个匹配字符的位置。

```
function CharPos20(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push esi
  mov    esi,edx
  mov    ebx,eax
  //if StrLenght > 0 then
  test esi,esi
  jz    @Else1Begin
  //StrLenght := Length(Str);
  mov edx,[esi-$04]
  //I := 0;
  xor    eax,eax
  dec    esi
  @RepeatBegin :
  //Inc(I);
```

```
  inc    eax
  //until((Str[I] = Chr) or (I > StrLenght));
  //cmp    bl,[esi+eax-$01]
  cmp    bl,[esi+eax]
  jz      @If2
  cmp    edx,eax
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  cmp    edx,eax
  jnl    @Exit
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  @Exit :
  pop    esi
  pop    ebx
end;
```

When we analyzed the code we saw that there was an offset of -1 in the addressing into the string. There is no need to subtract this offset at each iteration. It is a better idea to decrement the pointer to Str in esi once before entering the loop. We could also have decremented the loop counter in eax, but then we would have to add 1 again before returning the result.
At the very top of the function the two input parameters are copied to new registers.
This is redundant and we would like to copy propagate both, but eax is used as loop counter and we must first find another register for this purpose.

我们分析代码发现，在索引字符串的地址中有个 -1 的偏移量。
没有必要在每次迭代中都减去这个偏移。
一个比较好的办法是，在进入循环之前将 str 的指针减 1。
我们也可以在循环中将 eax 减 1，但是在返回结果之前又不得不再加 1。
在函数的最上面，两个形参被复制到新的寄存器。
这是冗余的，我们想复制传播两个，但是 eax 被作为循环计数，我们必须首先寻找另一个寄存器。

```
function CharPos22(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push esi
  mov    esi,edx
```

```
  mov    ebx,eax
  //if StrLenght > 0 then
  test esi,esi
  jz    @Else1Begin
  //StrLenght := Length(Str);
  mov edx,[esi-$04]
  //I := 0;
  //xor    eax,eax
  xor    ecx,ecx
  dec    esi
  @RepeatBegin :
  //Inc(I);
  //inc    eax
  inc    ecx
  //until((Str[I] = Chr) or (I > StrLenght));
  //cmp    bl,[esi+eax]
  cmp    bl,[esi+ecx]
  jz      @If2
  //cmp    edx,eax
  cmp    edx,ecx
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  //cmp    edx,eax
  cmp    edx,ecx
  jnl    @Exit
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  pop    esi            //New
  pop    ebx            //New
  ret                   //New
  @Exit :
  mov    eax, ecx
  pop    esi
  pop    ebx
end;
```

All lines where eax is in use for I, eax is changed to ecx.

Because I is the return value of the function on a match and the return value is in eax,
we have to copy ecx into eax just beneath the @Exit label.
This introduces a little problem, because a jump to Else1Begin also brings us here,
and in this situation we copy a value from ecx into eax, which we have just cleared.
The fix is to add the lines marked new.

所有使用 eax 都是为了 I，将 eax 改为 ecx。
因为当有匹配字符时，I 是函数的返回值，并且用 eax 返回，因此我们需要在 @Exit 标签下面复制 ecx 到 eax。说明一个小问题，因为 Else1Begin 也会到这里，在这种情况写刚刚被清空的 eax 会被 ecx 重写。增加的行用 new 标识

Then we are ready to copy propagate eax.
Only one line uses ebx. This is cmp bl, [esi+ecx], which is changed to cmp al, [esi+ecx].
Then the copy mov ebx, eax becomes dead and is removed.
This was copy propagation followed by dead code removal once more and
we begin realising how important this optimization is.

接下来我们要实现复制传播 eax。
只有 cmp bl, [esi+ecx] 这一行使用 ebx, 将它改为 cmp al, [esi+ecx]。
mov ebx, eax 这一行没用了，可以删除。
复制传播可以移除更多的无效代码，现在我们明白了优化有多么重要了吧。

```
function CharPos23(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push esi
  mov    esi,edx
  //mov    ebx,eax
  //if StrLenght > 0 then
  test esi,esi
  jz    @Else1Begin
  //StrLenght := Length(Str);
  mov edx,[esi-$04]
  //I := 0;
  xor    ecx,ecx
  dec    esi
  @RepeatBegin :
  //Inc(I);
  inc    ecx
  //until((Str[I] = Chr) or (I > StrLenght));
  //cmp    bl,[esi+ecx]
  cmp    al,[esi+ecx]
  jz      @If2
  cmp    edx,ecx
```

```
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  cmp    edx,ecx
  jnl    @Exit
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  @Exit :
  mov    eax, ecx
  pop    esi
  pop    ebx
end;
```

Before we can copy propagate edx (holding the pointer to Str), we must free edx from other uses. It is used for StrLenght and we allocate ebx for this instead of edx.
在我们复制传播 edx (指向 str) 之前，我们必须将 edx 从其他使用它的地方解放出来。它被用来存放 StrLenght，我们为它分配 ebx 来代替 edx。

```
function CharPos24(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push esi
  mov    esi,edx
  //if StrLenght > 0 then
  test esi,esi
  jz    @Else1Begin
  //StrLenght := Length(Str);
  //mov edx,[esi-$04]
  mov ebx,[esi-$04]
  //I := 0;
  xor    ecx,ecx
  dec    esi
  @RepeatBegin :
  //Inc(I);
  inc    ecx
```

```
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    al,[esi+ecx]
  jz     @If2
  //cmp    edx,ecx
  cmp    ebx,ecx
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  //cmp    edx,ecx
  cmp    ebx,ecx
  jnl    @Exit
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  @Exit :
  mov    eax, ecx
  pop    esi
  pop    ebx
end;
```

Then edx is copy propagated and the mov esi, edx becomes dead.
edx 被复制传播，mov esi, edx 无效了。

```
function CharPos25(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push esi
  //mov    esi,edx
  //if StrLenght > 0 then
  //test esi,esi
  test edx,edx
  jz    @Else1Begin
  //StrLenght := Length(Str);
  //mov ebx,[esi-$04]
  mov ebx,[edx-$04]
  //I := 0;
  xor    ecx,ecx
```

```
  //dec    esi
  dec    edx
  @RepeatBegin :
  //Inc(I);
  inc    ecx
  //until((Str[I] = Chr) or (I > StrLenght));
  //cmp    al,[esi+ecx]
  cmp    al,[edx+ecx]
  jz     @If2
  cmp    ebx,ecx
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  cmp    ebx,ecx
  jnl    @Exit
  //Result := 0;
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  pop    esi
  pop    ebx
  ret
  @Exit :
  mov    eax, ecx
  pop    esi
  pop    ebx
end;
```

This removed the use of esi and it does not need to be saved and restored any more. When removing the pop esi, we remember that there are 3 exit paths all with each own pop esi.
移除 esi 的使用，它不再需要保存和恢复了。
当我们移除 pop esi 时，要记着有 3 个退出路径都是自己 pop esi。

```
function CharPos26(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  //push esi
  //if StrLenght > 0 then
  test edx,edx
  jz    @Else1Begin
```

```
  //StrLenght := Length(Str);
  mov ebx,[edx-$04]
  //I := 0;
  xor    ecx,ecx
  dec    edx
  @RepeatBegin :
  //Inc(I);
  inc    ecx
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    al,[edx+ecx]
  jz      @If2
  cmp    ebx,ecx
  jnl    @RepeatBegin
  //if I <= StrLenght then
  @If2 :
  cmp    ebx,ecx
  jnl    @Exit
  //Result := 0;
  xor    eax,eax
  //pop    esi
  pop    ebx
  ret
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  //pop    esi
  pop    ebx
  ret
  @Exit :
  mov    eax, ecx
  //pop    esi
  pop    ebx
end;
```

In the line after the If2 label there is a line, which is identical to the second compare in the loop-closing test. In Pascal it was necessary to have the line if I <= StrLenght after the loop because we could not know which condition the loop terminated on. This line generated the extra cmp ebx, ecx instruction, which looks a little redundant now. It is in fact not redundant because two paths of execution lead to the If2 Label and only one of them has the test. If we split the two exit paths such that only one of them goes to If2 we can remove the extra check. Instead of jumping to If2 on a match we can jump directly to Exit.

在标签 If2 之后有一行是是否循环结束的第二次检测。
在 Pascal 代码的循环之后有 if I <= StrLenght, 因为我们不知道是被哪个条件终止的。
这一行产生额外的指令 cmp ebx, ecx，看起来有一点冗余。

实际上它不冗余，虽然有两路会执行到 If2 标签，但是只会测试一个。

如果我们分成两个退出路径，只让一个到达 If2，我们可以移除额外的检查。
将匹配时调到 If2 改为直接调到 Exit。

```
function CharPos27(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  //if StrLenght > 0 then
  test edx,edx
  jz    @Else1Begin
  //StrLenght := Length(Str);
  mov ebx,[edx-$04]
  //I := 0;
  xor    ecx,ecx
  dec    edx
  @RepeatBegin :
  //Inc(I);
  inc    ecx
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    al,[edx+ecx]
  //jz      @If2
  jz      @Exit
  cmp    ebx,ecx
  jnl    @RepeatBegin
  //if I <= StrLenght then
  //@If2 :
  //cmp    ebx,ecx
  //jnl    @Exit
  //Result := 0;
  xor    eax,eax
  pop    ebx
  ret
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  pop    ebx
  ret
  @Exit :
  mov    eax,ecx
  pop    ebx
end;
```

Then the If2 label goes out of use and when we reach the code at this position we know that the zero terminator was reached and there is no need to test for it again.
There are two sections of identical code just before the Else1Begin label and just after it. The

upper section can be deleted.

If2 标签没用了，当代码到达这个位置，我们已经知道到了零终止符，因此不需要再次检测了。 在 Else1Begin 的前后有两块相同的代码，上面的那块可以删除掉。

```
function CharPos28(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  //if StrLenght > 0 then
  test edx,edx
  jz      @Else1Begin
  //StrLenght := Length(Str);
  mov    ebx,[edx-$04]
  //I := 0;
  xor    ecx,ecx
  dec    edx
  @RepeatBegin :
  //Inc(I);
  inc    ecx
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    al,[edx+ecx]
  jz      @Exit
  cmp    ebx,ecx
  jnl    @RepeatBegin
  //Result := 0;
  //xor    eax,eax
  //pop    ebx
  //ret
  //Result := 0;
  @Else1Begin :
  xor    eax,eax
  pop    ebx
  ret
  @Exit :
  mov    eax,ecx
  pop    ebx
end;
```

This ended our search for redundant code to remove. The cleaned up function looks like this.
最终我们删除了发现的冗余代码，整理为如下函数：

```
function CharPos29(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  //if StrLenght > 0 then
  test edx,edx
```

```
  jz      @Else1Begin
  //StrLenght := Length(Str);
  mov    ebx,[edx-$04]
  //I := 0;
  xor    ecx,ecx
  dec    edx
  @RepeatBegin :
  //Inc(I);
  inc    ecx
  //until((Str[I] = Chr) or (I > StrLenght));
  cmp    al,[edx+ecx]
  jz      @Exit
  cmp    ebx,ecx
  jnl    @RepeatBegin
  @Else1Begin :
  //Result := 0;
  xor    eax,eax
  pop    ebx
  ret
  @Exit :
  mov    eax,ecx
  pop    ebx
end;
```

When the loop is iterating in search for a match or the end of the string, these lines of code are executed over and over again

当循环搜索匹配字符或者直到字符串的结束时，这些代码被一次又一次的执行：

```
inc    ecx
cmp    al,[edx+ecx]
jz      @Exit
cmp    ebx,ecx
jnl    @RepeatBegin
```

Let us make some variants of them and see how they perform. The most exiting line is

```
cmp    al,[edx+ecx]
```

It generates two microinstructions.

让我们来进行一些变化，看看他们怎样执行最重要的行是

```
cmp    al,[edx+ecx]
```

它产生了两个微指令。

One for loading a byte from the address [edx+ecx] and one to compare it against al.

一个是从地址 [edx+ecx] 装入一个字节，一个是再和 al 比较。

This line could also be coded as

这行也可以这样写

```
mov ah, byte ptr [edx+ecx]
cmp al, ah
```

This variant also generates two microinstructions, but it needs one more register (ah).
这个变化也产生了两条微指令，但它还需要一个寄存器 ah。

If we allocate an extra register it could also be coded as
如果我们分配额外的寄存器，也可以这样编码

```
movzx efx, byte ptr [edx+ecx]
cmp     al, fh
```

movzx is mov with zero extension.
It loads one byte into the lowest of efx and fills the three remaining bytes with zeroes.
Of course there is no such thing as an efx register, but the two unused registers esi & edi cannot be accessed byte wise. Therefore it is necessary to free eax, ebx, ecx or edx, by substituting it by edi or esi and then use erg.
ebx instead of "efx".
This function demonstrates the first variant.

movzx 是 mov 的扩展版本。
它将一个字节装入 efx 的最低字节，将剩余的 3 个字节用零填充。
当然，没有 efx 这样的寄存器，寄存器 esi&edi 虽然没被使用，但是不能被作为字节访问。因此，可以用 edi 或 esi 来取代 eax, ebx, ecx 或 edx，让它们自由。我们用 ebx 来替代 "efx"。

```
function CharPos30(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  //if StrLenght > 0 then
  test edx,edx
  jz     @Else1Begin
  //StrLenght := Length(Str);
  mov    ebx,[edx-$04]
  //I := 0;
  xor    ecx,ecx
  dec    edx
  @RepeatBegin :
  //Inc(I);
  inc    ecx
  //until((Str[I] = Chr) or (I > StrLenght));
  mov    ah, [edx+ecx]
  //cmp    al,[edx+ecx]
```

```
  cmp    al,ah
  jz     @Exit
  cmp    ebx,ecx
  jnl    @RepeatBegin
  @Else1Begin :
  //Result := 0;
  xor    eax,eax
  pop    ebx
  ret
  @Exit :
  mov    eax,ecx
  pop    ebx
end;
```

This function demonstrates the second variant.

第二个变化的函数演示

```
function CharPos31(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push ebx
  push edi
  //if StrLenght > 0 then
  test edx,edx
  jz     @Else1Begin
  //StrLenght := Length(Str);
  mov    edi,[edx-$04]
  //I := 0;
  xor    ecx,ecx
  dec    edx
  @RepeatBegin :
  //Inc(I);
  inc    ecx
  //until((Str[I] = Chr) or (I > StrLenght));
  movzx ebx, byte ptr [edx+ecx]
  //cmp    al,[edx+ecx]
  cmp    al, bl
  jz     @Exit
  cmp    edi,ecx
  jnl    @RepeatBegin
  @Else1Begin :
  //Result := 0;
  xor    eax,eax
  pop    edi
  pop    ebx
  ret
  @Exit :
```

```
  mov    eax,ecx
  pop    edi
  pop    ebx
end;
```

Instead of adding edx and ecx in the address calculation at every iteration,
we could add them prior to entering the loop.
Then it is necessary to subtract them from each other again to extract the loop counter value for the result.This is done with the sub instruction in line two after the Exit label.

为了替代在迭代中累加 edx 和 ecx 计算地址，我们可以在进入循环之前累加它们，
然后需要再分别减去他们来得到准确的循环计数结果。
在 Exit 标签后有两行代码来实现这个运算。

```
function CharPos32(Chr : Char; const Str : AnsiString) : Cardinal;
asm
  push    ebx
  push    edi
  //if StrLenght > 0 then
  test    edx,edx
  jz       @Else1Begin
  //StrLenght := Length(Str);
  mov      edi,[edx-$04]
  //I := 0;
  xor      ecx,ecx
  //dec    edx
  add      ecx,edx
  @RepeatBegin :
  //Inc(I);
  //until((Str[I] = Chr) or (I > StrLenght));
  movzx ebx, byte ptr [ecx]
  inc      ecx
  cmp      al, bl
  jz       @Exit
  //cmp    edi,ecx
  test    bl, bl
  jnz      @RepeatBegin
  @Else1Begin :
  //Result := 0;
  xor      eax,eax
  pop      edi
  pop      ebx
  ret
  @Exit :
  mov      eax,ecx
```

```
  sub     eax,edx
  pop     edi
  pop     ebx
end;
```

Then there are 4 functions to compare the performance of; CharPos29, CharPos30, CharPos31 and CharPos32.

有 4 个比较性能的函数 CharPos29, CharPos30, CharPos31 and CharPos32

Results on P4 1920 are 在 P4 1920 上

CharPos29 716

CharPos30 973

CharPos31 710

CharPos32 702

Winner is CharPos32 CharPos32 赢了

Results on P3 1400 are 在 P4 1400 上

CharPos29 949

CharPos30 921

CharPos31 950

CharPos32 1403

Winner is CharPos30 CharPos30 赢了

Summed time 总计时间

CharPos29 716 + 949 = 1665

CharPos30 973 + 921 = 1894

CharPos31 710 + 950 = 1660

CharPos32 702 + 1403 = 2105

Winner is CharPos31 CharPos31 赢了

On P4 the winning loop is 在 P4 上胜利的循环是

```
@RepeatBegin :
movzx ebx, byte ptr [ecx]
inc     ecx
cmp     al, bl
jz      @Exit
test    bl, bl
jnz     @RepeatBegin
```

On P3 the winning loop is 在 P3 上胜利的循环是

```
@RepeatBegin :
inc    ecx
mov    ah, [edx+ecx]
cmp    al,ah
jz     @Exit
cmp    ebx,ecx
jnl    @RepeatBegin
```

on a blend of targets the winning loop is 在混合平台上胜利的是

```
@RepeatBegin :
inc     ecx
```

```
movzx ebx, byte ptr [edx+ecx]
cmp     al, bl
jz      @Exit
cmp     edi,ecx
jnl     @RepeatBegin
```

The P4 winner performs very bad on P3 and could not be used in a library targeting more targets than P4,such as the Delphi RTL. The winner on P3 is performing quite bad on P4 and this one should not be used in a blended target library either. The winner on a blend of the two targets is CharPos31, which performs near to optimal on P4 and also performs optimal within a few percent on P3.

在 P4 上的赢家在 P3 上没效率，不能用在目标不是 P4 处理器的库中，比如 Delphi RTL。P3 上的赢家在 P4 上执行效率非常的低，这个也不能用在混合平台的库里。在两个处理器上，综合效率的赢家是 CharPos31，在 P4 上接近最佳，在 P3 上也没差多少。

This function would make a perfect choice for the Delphi RTL, where I have missed a CharPos function. It is a relief to see that it is possible to optimise for both processors at the same time without sacrificing more than a few percent of performance.

这个函数是 Delphi RTL 最好的选择，在 RTL 中我已经失去了一个 CharPos 函数。
我们欣慰的看到，它可以同时作为两个处理器的优化版本，并且几乎没有牺牲一点性能。

Comparing performance of P3 and P4 on a clock-by-clock basis is always a hit.
There is broad tendency to think at the P4 as having an inferior design,but this is not proved by this code.Taking the blended winner's performance and scaling it to a 1400 MHz processor it is 950 on P3 and 710 * (1920/1400) = 973 on P4.
The processors are performing very much the same at the same clock.

在 P3 和 P4 上比较时钟周期的性能也是一个打击。
有一个大胆的猜测就是 P4 也有一个低级的设计，但是不能被这个代码证明。
在两种平台上比较综合赢家的性能，放在 P3 1400HZ 的处理器上是 950，在 P4 上是 710 * (1920/1400) = 973。
这两种处理器几乎花费了同样的时钟周期。

Lesson 7 第 7 课(高级符点数运算)

Welcome to lesson number 7. Today's subject is floating point BASM.
This was also the subject of an earlier lesson, but this lesson will add new information.
We will look at how to code scalar SSE2 code and how instructions are scheduled in the fp pipelines.
Today's example function is evaluating a third order polynomial.

欢迎来到第 7 课。今天的主题是 BASM 中的浮点数。
这也是我们之前课程的一个主题，但是这一课将增加一些新内容。
我们将着眼于怎样编写标量 SS2 代码和在 fp 管道线中如何调度这些指令。
今天的例子是评测一个三次多项式。

```
function ArcSinApprox1a(X, A, B, C, D : Double) : Double;
begin
  Result := A*X*X*X + B*X*X + C*X + D;
end;
```

Instead of just analyzing and optimizing this function we will see an actual use of it. A third order polynomial can approximate the ArcSin function on its entire interval, [-1;1], with a maximal absolute error of 0.086.
This is not impressive but what we develop in this lesson will extend to higher order polynomials in a straightforward manner, yielding higher precision.

我们先看看它的实际使用，而不是马上分析和优化。
一个三次多项式在闭区间 [-1, 1] 可以近似为 ArcSin 函数，它们的最大绝对值误差是 0.086。这不令人印象深刻，但是我们将在这一课中延伸到更高次幂的多项式，让它更精确。

The parameters A, B, C and D define the shape of the curve for the function and the values for a fit to theArcSin approximation with minimal error must be found.
参数 A, B, C 和 D 决定了函数曲线的形状。在近似的 ArcSin 中，在最小误差允许范围内，肯定存在这些值。

For this purpose we develop an optimizer, which is also used as the benchmark.
Because ArcSin(0) = 0 we immediately see that D=0 and D can be left out of optimization.
为了这个目的我们开发一个优化器，它也可以作为基准测试。
因为 ArcSin(0) =0 ，我们可以马上知道 D=0，所以 D 可以被优化掉。

We also know that ArcSin is an odd function and therefore the second order term B*X*X is of no use in the approximation.
我们也知道 ArcSin 是一个奇函数，因此二次项 B*X*X 在近似中没有用。

This is because a second order term is even and has symmetry around the Y-axis.
Odd order functions have antisymmetry around the Y-axis with f(X) = -f(-X).
All this means that our function could be reduced to Result := A*X*X*X + C*X;
We do however not does that because it is more illustrative to use the full function.
ArcSin is a special case and we want to be as general as possible.

这是因此二次项是一个偶特性，沿着 Y 轴对称。
奇次函数具有沿着 Y 反对称特性，f(X) = -f(-X)。
根据这些特性，函数可以被减为 Result := A*X*X*X + C*X;
但我们却不能这样做，因为使用这个完整的函数更有说明性。
ArcSin 是一个特例，我们尽可能让它通用。

Function number 1a has 6 multiplications and 3 additions. Writing it on Horner form
函数有 6 个乘和 3 个加。将它写成括号形式
Result := ((A*X + B)*X + C)*X + D;

reduces this to 3 multiplications and 3 additions.
减为 3 个乘和 3 个加。

Another form is 另一种形式是
Result := (A*X + B)*(X*X)+(C*X + D);

which has 4 multiplications and 3 additions.
这一个有 4 个乘和 3 个加。

On modern processors it is as important how much parallelism can be extracted from the formula as it is how many multiplications and additions it has.
Modern processors such as AMD Athlon, Intel P4 and P3 have pipelines.
Pipelines are a necessity on processors running at high frequencies, because the amount of work for an addition, subtraction, multiplication or division cannot be done in a single clock cycle.

在现代的处理器中，一个公式进行了多少次计算和它有多少个乘法、加法一样重要。
现代处理器像 Amd Athlon，Intel P4 和 P3 都有管道线。
管道线在处理器上是必需的，运行频率相当高，因为大量的工作是处理加、减、乘、除，它们在单个时钟周期里运行不完。

On P4 there is a pipeline called FP_ADD that handles addition and subtraction.
This pipeline has 5 stages, which means that the process of performing an addition or subtraction is broken down into 5 sub tasks.
Therefore addition and subtraction is done in 5 clock cycles.

在 P4 有一个叫做 FP_ADD 的管道线，它操作加法和减法。
管道线有 5 步，意思是将一个加法或减法分成 5 个子任务执行。
因此加法和减法是用 5 个时钟周期来完成。

The advantage of having a pipeline is that even though it takes 5 cycles to complete an addition a new one can be started in every clock cycle. This is because the first add in a series leaves the first stage of the pipeline in the second clock cycle and then this stage can accept add number 2.
If we have this series of adds the first one will leave the pipeline in cycle 5, number 2 will leave in cycle 6 etc.Throughput is one add per cycle.

管道线的优点是，尽管完成一个加法花费 5 个周期，但是在每个时钟周期都可以开始一个新的。这是因为第一个 add 进入管道线的第一步，在第二个时钟周期时管道线的第一步则可以接收第 2 个 add。如果我们有这一系列的加法，第一个加法将在第 5 个时钟周期离开，第二个加法将在第 6 个时钟周期离开，依次类推。吞吐量每个周期加一个。

Parallelism is achieved by having up to 5 additions/subtractions in flight in the pipeline at the same time. The drawback is that if a second of two following additions need the output from the first addition it has to wait for the first one to complete. We say that there is a data dependency between the two instructions and we see the full latency of the additions, which is 2 times 5 cycles.

使用管道线，同时可以并行完成 5 个加法或减法。
有一个缺点，就是如果两个相邻的加法，第二个加法需要第一个的输出结果，那么它就不得不等待第一个完成。
我们将它称为两个指令之间的数据依赖，我们看到这两个加法的潜伏期是 5 个时钟周期的 2 倍。

Let’s use our function as an example to see how pipelines work.
用我们的函数来看看管道线如何工作

Result := A*X*X*X + B*X*X + C*X + D;
It is seen that the 4 terms can be evaluated in parallel and then be added as the final act Of course term number 4 is not "evaluated". A*X is the first operation ready to be scheduled to run in the F_MUL pipeline. The latency for FMUL on P4 is 7 cycles and A*X will be ready in cycle 7.

这 4 项可以被并行的计算，然后加结果相加，当然第 4 项不用 "计算" 了。
A*X 是被调度进 F_MUL 管道线的第一个操作。在 P4 上 FMUL 的潜伏期是 7 个时钟周期，A*X 会被安排 7 个时钟周期。

FMUL has a throughput of 2 cycles. From this we see that FMUL is not fully pipelined.
The pipeline will accept a new instruction in cycle 3 and not in cycle 2.
B*X is the second instruction ready to execute and it will start in cycle 3.
In cycle 5 the pipeline is again ready to receive a new instruction and this is C*X.
In cycle 7 A*X is complete and (A*X)*X can start in cycle 8.
In cycle 10 B*X is finished and (B*X)*X will start.

FMUL 有 2 个周期的吞吐量。因此 FMUL 没有占满管道线。

管道线将在第 3 个周期接收一个新的指令，而不是在第 2 个周期。
B*X 是要执行的第二个指令，它将在第 3 个周期开始。
在第 5 个周期，管道线将再接收一个新指令，它是 C*X。
在第 7 个周期， A*X 完成了，(A*X)*X 在第 8 个周期开始。
在第 10 个周期 B*X 完成，并且 (B*X)*X 将开始执行。

In cycle 12 C*X is ready to go to the F_ADD pipeline to be added to D.
In cycle 15 is (A*X)*X finished and (A*X*X)*X can start.
In cycle 17 are (B*X)*X and (C*X) + D complete and they can start in the F_ADD pipeline.
This addition completes in cycle 21, where (A*X*X)*X is also ready Then the last addition can start in cycle 22.
Now there is only one operation in flight and we must lean back and wait for the full latency of FADD, which is 5 cycles. In clock 27 the last addition is finished and the job is done.

在第 12 周期 C*X 准备进入 F_ADD 管道线与 D 相加。
在第 15 周期 (A*X)*X 完成，(A*X*X)*X 开始。
在第 17 周期 (B*X)*X 和 (C*X) + D 完成，然后开始进入 F_ADD 管道线。
这个加法在第 21 周期完成，第 22 周期 (A*X*X)*X 进入最后的加法。
现在，只有一个操作在运行，我们必须等待 FADD 的完成，它用了 5 个周期。
在第 27 个周期最后的加法结束，整个工作结束了。

These tables give the details. The left column symbolizes the F_MUL pipeline with 7 stages and the right column symbolizes the F_ADD pipeline with 5 stages.
下面的表格详细的描述了这个过程。
左边的列表示 F_MUL 管道线用了 7 步，右边的列表示 F_ADD 用了 5 步。

F_MUL	F_ADD
A*X	

Cycle 1

F_MUL	F_ADD
A*X	

Cycle 2

F_MUL	F_ADD
B*X	
A*X	

Cycle 3

F_MUL	F_ADD
B*X	
A*X	

Cycle 4

F_MUL	F_ADD
C*X	
B*X	
A*X	

Cycle 5

F_MUL	F_ADD
C*X	
B*X	

A*X

Cycle 6

F_MUL	F_ADD
C*X	
B*X	
A*X	

Cycle 7

F_MUL	F_ADD
(A*X)*X	
C*X	
B*X	

Cycle 8

F_MUL	F_ADD
(A*X)*X	
C*X	
B*X	

Cycle 9

F_MUL	F_ADD
(B*X)*X	
(A*X)*X	
C*X	

Cycle 10

F_MUL	F_ADD

(B*X)*X	
(A*X)*X	
C*X	

Cycle 11

F_MUL	F_ADD
	(C*X)+D
(B*X)*X	
(A*X)*X	

Cycle 12

F_MUL	F_ADD
	(C*X)+D
(B*X)*X	
(A*X)*X	

Cycle 13

F_MUL	F_ADD
	(C*X)+D
(B*X)*X	
(A*X)*X	

Cycle 14

F_MUL	F_ADD
(A*X*X)*X	

	(C*X)+D
(B*X)*X	

Cycle 15

F_MUL	F_ADD
(A*X*X)*X	
	(C*X)+D
(B*X)*X	

Cycle 16

F_MUL	F_ADD
	(B*X*X)+(C*X+D)
(A*X*X)*X	

Cycle 17

F_MUL	F_ADD
	(B*X*X)+(C*X+D)
(A*X*X)*X	

Cycle 18

F_MUL	F_ADD
	(B*X*X)+(C*X+D)

(A*X*X)*X	

Cycle 19

F_MUL	F_ADD
	(B*X*X)+(C*X+D)
(A*X*X)*X	

Cycle 20

F_MUL	F_ADD
	(B*X*X)+(C*X+D)
(A*X*X)*X	

Cycle 21

F_MUL	F_ADD
	(A*X*X*X)+ (B*X*X+C*X+D)

Cycle 22

F_MUL	F_ADD
	(A*X*X*X)+ (B*X*X+C*X+D)

Cycle 23

F_MUL	F_ADD
	(A*X*X*X)+ (B*X*X+C*X+D)

Cycle 24

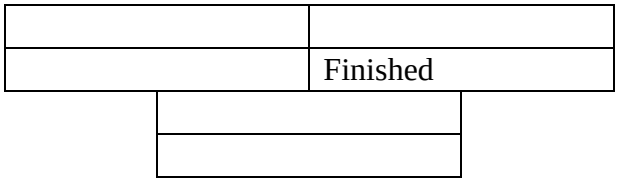
F_MUL	F_ADD
	(A*X*X*X)+ (B*X*X+C*X+D)

Cycle 25

F_MUL	F_ADD
	(A*X*X*X)+ (B*X*X+C*X+D)

Cycle 26

F_MUL	F_ADD



Cycle 27

An out of order core schedules instructions to run as soon as data and resources are ready.
Resources are registers and execution pipelines. I do not know for sure, but I think that instructions are scheduled to run in program order except when an instruction stalls.
In this situation the next ready instruction in program order is scheduled.
The stalled instruction will be started as soon as the stall reason is removed.
It can stall because of missing resources or not ready data.

非顺序执行的内核调度指令只要数据和资源准备好就运行。
资源就是寄存器和执行管道线。
我不能确定，但是我想指令会按照程序的顺序运行，除非一个指令停转。
在这种情况下，下一个指令将被调度。这个停转的指令在停转原因消除了之后会开始执行。
它停转的原因是因为没有资源或者没有准备好数据。

After having seen how instructions are scheduled to run in the pipelines of a P4 we establish the benchmark.
The benchmark is an optimizer that search for the best possible fit of our polynomial to ArcSin. It is based on the most simple of all optimization algorithms, which is the exhaustive search. We simply try a lot of combinations of parameter values and remember the set of parameters, which give the best fit.
A and C are run in the intervals [AStart;AEnd] and [CStart; CEnd], with the step sizes AStepSize and CStepsize.
This is done in two nested while loops.

我们已经学习了 P4 中的管道线是如何调度指令运行的，接下来我们建立一个基准测试。
基准是一个优化器，它能够尽可能的找到满足 ArcSin 的多项式。
它基于最简单的优化算法，就是穷举搜索。我们简单的尝试参数值的多种组合，并且记录最满足条件的参数值。
A 和 C 分别在闭区间 [AStart;AEnd] 和 [CStart; CEnd]中，步进大小是 AStepSize 和 CStepsize。
用两个 while 嵌套循环实现。

```
StartA     := 0;
StartC     := -1;
EndA       := 1;
EndC       := 1;
AStepSize := 1E-2;
CStepSize := 1E-3;
OptA       := 9999;
```

```
  OptC          := 9999;
  A             := StartA;
  while A <= EndA do
    begin
      C := StartC;
      while C <= EndC do
        begin
          Inc(NoOfIterations);
          MaxAbsError := CalculateMaxAbsError(A,C, ArcSinArray);
          if MaxAbsError <= MinMaxAbsError then
            begin
              MinMaxAbsError := MaxAbsError;
              OptA := A;
              OptC := C;
            end;
          C := C + CStepSize;
        end;
      A := A + AStepSize;
    end;
```

The CalculateMaxAbsError function calculates a number of points on the X interval [-1;1], which is the definition interval of the ArcSin function

CalculateMaxAbsError 函数计算 X 的闭区间 [-1;1] 中的数，这个区间是 ArcSin 函数的区间。

```
TMainForm.CalculateMaxAbsError(A, C : Double; ArcSinArray : TArcSinArray) : Double;
var
  X, Y, D, B, Yref, Error, AbsError, MaxAbsError : Double;

begin
  B := 0;
  D := 0;
  MaxAbsError := 0;
  X := -1;
  repeat
    Yref := ArcSin (X);
    Y := ArcSinApproxFunction(X, A, B, C, D);
    Error := Yref-Y;
    AbsError := Abs(Error);
    MaxAbsError := Max(MaxAbsError, AbsError);
    X := X + XSTEPSIZE;
  until(X > 1);
  Result := MaxAbsError;
end;
```

At each point we calculate the error by subtracting the Y value from our approximation function

from the reference
Y value obtained from a call to the Delphi RTL ArcSin function.

我们使用自己的近似 ArcSin 的函数求一个 Y 值，使用 Delhi RTL 中的 ArcSin 函数求一个 Y 参考值，然后用 Y 参考值减 Y 值来计算在每个点上的误差。

The error can be positive or negative, but we are interested in the absolute value.
We remember the biggest absolute error by taking the maximum of the two values MaxAbsError and AbsError assigning it to MaxAbsError.
MaxAbsError is initialized to zero and in the first evaluation it will get the value of the first error (if it is bigger than zero).
The MaxAbsError is returned as the result from the function after a full sweep has been completed.
In the optimizer function the two values of A and C that gave the smallest maximum error are remembered together with the actual MinMaxAbsError.

误差可能是正数或负数，但我们只对绝对值感兴趣。
我们通过将 MaxAbsError 和 AbsError 两个值的最大值赋给 MaxAbsError 来记录最大的绝对值误差。
MaxAbsError 被初始化为零，在计算时，首先获得第一个误差值(如果误差大于零)。
在计算完成之后 MaxAbsError 作为函数的返回值。
在优化器函数中，将记录最小的最大误差时的 A，C 值。

All that matters in an optimizer of this kind is to be able to evaluate as many parameter combinations as possible. For this purpose we must optimize the optimizer ;-) and the functions we evaluate. In this lesson the purpose is slightly different because all we want is some valid benchmarks for the functions we optimize.

这种类型的优化器将尽可能的计算更多的组合参数。
因此，我们将优化这个优化器和我们进行计算的函数。
这一课的目的稍微有点不同，因为我们想的是优化这些正确的基准测试函数。

The means are however the same, the code of the optimizer must take as few cycles as possible such that the cycles the functions use is the biggest part of the total number of cycles used. The first optimizer optimization that is done is to realize that there is no need to evaluate the reference function over and over again.

意思也相似，优化器代码也必须尽能的花费很小的时钟周期，循环周期函数在总的周期内占了最大的一部分。首先优化优化器中并不需要一次次计算的参考函数。

It returns, of course, the same values no matter which values A and C have.
Sweeping the reference function once and storing the Yref values in an array do this.
The next optimization is to compact the lines that evaluate the MaxAbsError Long version

它有返回值，当然不管 A 和 C 是否相同都返回。消除参考函数，只需计算一次并将结果存入数组中。接下来，优化计算 MaxAbsError 的冗长代码

```
Yref := ArcSinArray[I];
Error := Yref-Y;
AbsError := Abs(Error);
```

Short version 短版本

```
AbsError := Abs(ArcSinArray[I]-Y);
```

This helps because Delphi creates a lot of redundant code, when compiling FP code.
The long version compiles into this
这是有益的, 因为当编译 FP 代码时, Delphi 会产生许多冗余代码,
长版本编译为以下

```
Yref := ArcSinArray[I];

mov eax,[ebp-$14]
mov edx,[eax+ebx*8]
mov [ebp-$48],edx
mov edx,[eax+ebx*8+$04]
mov [ebp-$44],edx

Error := Yref-Y;

fld      qword ptr [ebp-$48]
fsub qword ptr [ebp-$30]
fstp    qword ptr [ebp-$50]
wait

AbsError := Abs(Error);

fld qword ptr [ebp-$50]
fabs
fstp qword ptr [ebp-$10]
wait
```

There are a lot of redundancies in this code and we must conclude that Delphi is doing a bad job on optimizing floating point code.
在代码里有大量的冗余, 我们可以断定 Delphi 没有对浮点数代码进行有效的优化.

Let us add some explanations to the code.
我们来解释一下代码.

The first line of Pascal is an assignment of one double variable to another.

This is done by to pairs of mov, one for the lower 4 byte of the variables and one for the upper 4 byte.The first line of asm loads the address of the array into eax, which is used as base for addressing into the array. Ebx is I and it is multiplied by 8 because each entry in the array is 8 byte.

第一行 Pascal 代码是将一个 Double 变量赋给另一个.
这一行由一对 mov 实现, 一个用于变量的低 4 个字节, 一个用于高 4 个字节。
第一行 asm 代码 将数组的地址装入 eax，它指向数据的基地址。
Ebx 是 I，它乘以 8 是因为每个数组数据是 8 个字节。

The 4 byte offset in the last two lines (in the last line it is hidden!) is moving
the reference into the middle of element I.
最后两行中(最后一行是隐藏的) 4 个字节的偏移量，用于指向元素 I 的中间。

Yref is located in the stack frame at [ebp-$48] and is loaded by the first line of FP code.
Y is located at [ebp-$30] and is subtracted from Yref by fsub.
The result is Error and this is stored at [ebp-$50].

Yref 在堆栈 [ebp-$48] 中，由第一行 FP 代码装入。
Y 在 [ebp-$30] 中，fsub 用 Yref 减去 Y。
结果是误差，存储在 [ebp-$50]。

The last line of Pascal compiles into four lines of asm of which the first starts loading Error.
Saving and then loading Error is redundant and the optimizer should have removed it.
Fabs is the Abs function and is probably one of the shortest function implementations seen ;-)
The Delphi compiler does not have the inlining optimization, but it applies “compiler magic” to a few functions,one of which is Abs. The last line stores AbsError on the stack.
The short version compiles into this

最后一行 Pascal 被编译成四行 asm，从装入误差开始。
保存然后再装入是冗余的，优化器应该删除它。
Fabs 是 Abs 函数，看来来可能是最短的执行函数了。
Delphi 编译器没有内联优化，但是它可以对一些函数应用 “编译器魔法”，Abs 是其中的一个。最后一行将 AbsError 存入堆栈。

简短版本被编译成如下

```
mov eax,[ebp-$14]
fld qword ptr [eax+ebx*8]
fsub qword ptr [ebp-$30]
fabs
fstp qword ptr [ebp-$10]
wait
```

Here there is no redundancy and the compiler should have emitted this code as result of the long Pascal version as well. All lines of code are present in the long version, but all redundant lines have been removed. The first line loads the base address of the array into eax. The second line loads element I, I is in ebx,
unto the fp stack. The third line subtracts Y from Yref. The fourth line is the Abs function.
The fifth line stores the result in the AbsError variable.

这时没有冗余，编译器也应该将长版本的 Pascal 代码优化成它。
这些行也存在于长版本中，但是所有的冗余行都被删除了。
第一行将数组的基址装入 eax。第二行装入元素 I 到 fp 堆栈，I 在 ebx 中。
第三行从 Yref 减去 Y。第四行是 Abs 函数。
第五行将结果存入 AbsError 变量。

There is a peculiarity with this benchmark that I have no explanation of.The benchmark values are heavily influenced by the way it is activated. If the keyboard is used to hit the button we get a different score from the one we get by clicking the button with the mouse!
The one who can explain this will probably get the Nobel Prize in Delphi;-)

这个基准测试有一个我还没有解释的特性。
基准值被它的激活方式严重的影响。
用键盘点击按钮得到的值与用鼠标点击按钮得到的值是不相同的!
谁能解释它将获得 Delphi 诺贝尔奖 ;-)

Another irritating thing is that Delphi does not align double variables properly.
They shall be 8 byte aligned but Delphi only 4 byte aligns them.
The penalty we can get from this is a level 1 cache line split (and L2 cache line splits as well).
Loading a variable that splits over two cache lines takes the double time of loading one that does not.

另一个令人气愤地事情是 Delhpi 不能对齐 double 变量。
它们应该是 8 字节对齐，但是 Delphi 只是 4 字节对齐它们。
我们将得到一个处罚，因为一级 cache 将分开它们(二级 cache 也将分开它们)
装载分为两部分的变量到高速缓冲将花费双倍的装入时间，因为不能一次装入。

(Q:)
Because double variables are 8 byte and the L1 cache line of P4 is 64 byte at most 1 of 8 variables can have a split. On P3 that has 32 byte L1 cache lines it is 1 of 4.
因为 double 变量是 8 个字节，P4 的 L1 cache 是 64 个字节，最多 8 个变量中有 1 个被分开。在 P3 上有 32 个字节，L1 cache 最多 4 个变量中有 1 个被分开。

The ideal situation is that variables are aligned at 4 byte if they are 4 byte big, 8 if 8 byte etc. To make things simple let us imagine that the first line in the level one cache is where our variables are loaded. The first line starts at address 0, that is - memory from address 0 is loaded into it. Our

first variable is aligned and occupies the first 8 bytes at line 1.
理想的情况是变量大小是 4 个字节则按 4 个字节对齐，是 8 个字节则按 8 个字节对齐。
为了使事情简单化，我们假设第一行将变量装入一级 cache。
第一行地址起始于 0，就是说，从地址是 0 的内存装入。
第一个变量是对齐的，在第 1 行占用 8 个字节。

Variable number two occupies byte 9-16 ….Variable number 8 occupies byte 57-64 and does not cross the line boundary.If variables are only 4 byte aligned the first one could start at byte 4 and number 8 could start at byte 61.The first 4 byte of it will be in line 1 and the next 4 bytes will be in line 2.The processor loads it by first loading the lower part and then loading the higher part instead of loading it all in one go.

第 2 个变量占用 9-16 字节。第 8 个变量占用 57-64 字节。不能交叉行边界。
如果变量只有 4 个字节，第一个可以对齐，之后从第 4 个字节开始，第 8 个变量将开始于第 61 个字节。它的前面 4 个字节在第 1 行，后面的字节在第 2 行。
处理器首先装入低位，然后再装入高位，而不是一次性装入。

Because of this misalignment of double variables in Delphi our benchmark will not be as stable as we could wish. Alignment can change when we recompile especially if code is changed. I have chosen (a bad choice) to not include code to align variables in the benchmark,
but will give an example of it in a later lesson.
Let us dive into the first function optimization.
We start with the function that uses the naive formula in textbook format.

因为在 Delphi 中 double 变量不能字节对齐，因此我们的基准测试不会像我们想象的那么稳定。通过修改代码，重新编译，可以改变对齐。
我有一个选择(一个坏的选择)，在基准测试中，通过不包含代码来对其变量。
让我们开始优化第一个函数。
我们使用教科书般的公式格式来讲解第一个函数。

```
function ArcSinApprox1a(X, A, B, C, D : Double) : Double;
begin
  Result := A*X*X*X + B*X*X + C*X + D;
end;
```

This function implementation scores the benchmark 43243 on my P4 1600 MHz clocked at 1920 MHz Delphi compiled it like this
在我的 P4 1600 MHz 时钟频率是 1920 MHz 的机器上，函数的基准测试分数是 43243。
编译后的代码如下：

```
function ArcSinApprox1b(X, A, B, C, D : Double) : Double;
begin
  {
  push    ebp
```

```
  mov      ebp,esp
  add      esp,-$08
  }
  Result := A*X*X*X + B*X*X + C*X + D;
  {
  fld      qword ptr [ebp+$20]
  fmul     qword ptr [ebp+$28]
  fmul     qword ptr [ebp+$28]
  fmul     qword ptr [ebp+$28]
  fld      qword ptr [ebp+$18]
  fmul     qword ptr [ebp+$28]
  fmul     qword ptr [ebp+$28]
  faddp st(1)
  fld      qword ptr [ebp+$10]
  fmul     qword ptr [ebp+$28]
  faddp st(1)
  fadd     qword ptr [ebp+$08]
  fstp     qword ptr [ebp-$08]
  wait
  fld      qword ptr [ebp-$08]
  }
  {
  pop      ecx
  pop      ecx
  pop      ebp
  }
end;
```

The code from the CPU view will not compile because of the instruction faddp st(1) and we remove st(1). As default the faddp instruction operates on st(0), st(1) and there is no need to write it out.

来自 CPU 窗口中的代码不能被编译，因此指令 faddp st(1)，我们移除 st(1)。

faddp 指令默认是操作 st(0), st(1)，不需要将它们写上。

```
function ArcSinApprox1c(X, A, B, C, D : Double) : Double;
asm
  //push    ebp    //Added by compiler 由编译器添加
  //mov      ebp,esp      //Added by compiler 由编译器添加
  add      esp,-$08
  //Result := A*X*X*X + B*X*X + C*X + D;
  fld      qword ptr [ebp+$20]
  fmul     qword ptr [ebp+$28]
  fmul     qword ptr [ebp+$28]
  fmul     qword ptr [ebp+$28]
  fld      qword ptr [ebp+$18]
```

```
  fmul    qword ptr [ebp+$28]
  fmul    qword ptr [ebp+$28]
  faddp //st(1)
  fld     qword ptr [ebp+$10]
  fmul    qword ptr [ebp+$28]
  faddp //st(1)
  fadd    qword ptr [ebp+$08]
  fstp    qword ptr [ebp-$08]
  wait
  fld     qword ptr [ebp-$08]
  pop     ecx
  pop     ecx
  //pop     ebp //Added by compiler    由编译器添加
end;
```

First we observe that there is no need to set up a stack frame. The stack is actually used for storing the result temporarily and reloading it again in the lines
首先我们观察是否需要设置堆栈。
堆栈实际上用来存储临时结果变量，只有又被重新装入浮点数寄存器

```
fstp    qword ptr [ebp-$08]
wait
fld     qword ptr [ebp-$08]
```

but the base pointer and not the stack pointer are used for this.
The lines that use ebp plus a value are accessing the parameters, which are located above the base pointer,which is in the calling functions stack frame. The stack pointer is not used at all in the function and changing its value is meaningless. The mov ebp, esp instruction added by the compiler together with the line add esp, -$08 creates an 8-byte stack frame.
Because these lines change the ebp register it is necessary to back it up by pushing it to the stack.
Unfortunately we can only remove the add esp, 8 line and the two pop ecx lines that has the purpose of subtracting 8 bytes from the stack pointer, esp.

但是这是基址指针，堆栈指针不能这样使用，基址 ebp 加上一个值可以访问由基址指针定位的参数。堆栈指针根本不能这样使用，改变它的值是无意义的。
编译器增加了 mov ebp, esp 指令，还有在堆栈上分配 8 个字节的 add esp, -$08。
因为这些行改变了 ebp 寄存器，因此需要通过压入堆栈来备份它。
遗憾的是，我们只能移除 add esp, 8 ，还有两行 pop ecx，它们是为了恢复前面已经将堆栈指针 esp 减 8 的操作。

```
function ArcSinApprox1d(X, A, B, C, D : Double) : Double;
asm
  //add     esp,-$08
  //Result := A*X*X*X + B*X*X + C*X + D;
```

```
  fld      qword ptr [ebp+$20]
  fmul     qword ptr [ebp+$28]
  fmul     qword ptr [ebp+$28]
  fmul     qword ptr [ebp+$28]
  fld      qword ptr [ebp+$18]
  fmul     qword ptr [ebp+$28]
  fmul     qword ptr [ebp+$28]
  faddp
  fld      qword ptr [ebp+$10]
  fmul     qword ptr [ebp+$28]
  faddp
  fadd     qword ptr [ebp+$08]
  fstp     qword ptr [ebp-$08]
  wait
  fld      qword ptr [ebp-$08]
  //pop      ecx
  //pop      ecx
end;
```

This function implementation scores the benchmark 42391 and performance actually dipped a little.The compiler inserts the line mov ebp, esp and we can make it redundant by using esp instead of ebp.

这个函数的基准测试分数是 42391，居然减少了一点。

编译器插入了 mov ebp, esp，我们可以使用 esp 来代替冗余的 ebp。

```
function ArcSinApprox1e(X, A, B, C, D : Double) : Double;
asm
  //Result := A*X*X*X + B*X*X + C*X + D;
  //fld      qword ptr [ebp+$20]
  fld      qword ptr [esp+$20]
  //fmul     qword ptr [ebp+$28]
  fmul     qword ptr [esp+$28]
  //fmul     qword ptr [ebp+$28]
  fmul     qword ptr [esp+$28]
  //fmul     qword ptr [ebp+$28]
  fmul     qword ptr [esp+$28]
  //fld      qword ptr [ebp+$18]
  fld      qword ptr [esp+$18]
  //fmul     qword ptr [ebp+$28]
  fmul     qword ptr [esp+$28]
  //fmul     qword ptr [ebp+$28]
  fmul     qword ptr [esp+$28]
  faddp
  //fld      qword ptr [ebp+$10]
  fld      qword ptr [esp+$10]
```

```
  //fmul    qword ptr [ebp+$28]
  fmul    qword ptr [esp+$28]
  faddp
  //fadd    qword ptr [ebp+$08]
  fadd    qword ptr [esp+$08]
  //fstp    qword ptr [ebp-$08]
  fstp    qword ptr [esp-$08]
  wait
  //fld     qword ptr [ebp-$08]
  fld     qword ptr [esp-$08]
end;
```

Unfortunately the compiler still inserts the mov instruction and we performed a copy propagation that gave no optimization because it is not followed by a dead code removal.
Therefore performance is almost the same – 43094. Without investigating whether the result stored on the stack is used we can optimize the lines coping it there and reloading it. The result of them is that there is a copy of Result left on the stack.
They redundantly pop the result of the FP stack and reload Result from the stack.

不幸的是，编译器仍然会插入 mov 指令。我们执行无优化的复制传播，因为它不能作为无效代码被移除。如果我们能够确定结果是存在堆栈上，则可以优化复制和装载这一步。
其中的一个结果是结果存放在堆栈上。
把结果从 FP 堆栈 pop 出，再从堆栈装载是冗余的。

This single line has the same effect, but redundancy is removed.
fst qword ptr [ebp-$08]
This optimization is very often possible on Delphi generated code and is important to remember.

这一行有相同的效果，但是冗余被删除。
这中优化在 Delphi 产生的代码中经常使用，应该记住它。

```
function ArcSinApprox1f(X, A, B, C, D : Double) : Double;
asm
  //Result := A*X*X*X + B*X*X + C*X + D;
  fld     qword ptr [esp+$20]
  fmul    qword ptr [esp+$28]
  fmul    qword ptr [esp+$28]
  fmul    qword ptr [esp+$28]
  fld     qword ptr [esp+$18]
  fmul    qword ptr [esp+$28]
  fmul    qword ptr [esp+$28]
  faddp
  fld     qword ptr [esp+$10]
  fmul    qword ptr [esp+$28]
```

```
  faddp
  fadd    qword ptr [esp+$08]
  //fstp    qword ptr [esp-$08]
  fst    qword ptr [esp-$08]
  wait
  //fld      qword ptr [esp-$08]
end;
```

This function implementation scores the benchmark 47939 and the improvement is 11 %.
The next question to ask is: Is the copy of the Result on the stack ever used?
To answer it we must inspect the code at the location of the call to the function.

这个函数的基准测试是 47939，进步了 11%。
接下来的问题是，在堆栈上的结果如何使用呢？
为了回答它，我们需要看调用这个函数的本地代码：

```
Y := ArcSinApproxFunction(X, A, B, C, D);

call dword ptr [ArcSinApproxFunction]
fstp qword ptr [ebp-$30]
wait
```

The first line after the call stores the result in Y and pops the stack.
Seeing this we assume that the result on the stack is not used,but to be sure we must scan through the rest of the code too. If the rule for the Register calling convention is that FP results are transferred on the FP stack it is weird that a copy is also placed on the stack. We conclude that it is redundant to copy the Result to the stack and remove the line doing it.

第一行将函数结果存入 Y，弹出堆栈。
我们假设在堆栈上的结果没有被使用，同时必须确定剩下的代码也没有使用。
如果 FP 的结果通过 FP 堆栈传输，并且在将结果复制到堆栈，这样的寄存器调用约定是不可思议的。
我们断定将结果复制到堆栈是冗余的，删除它。

```
function ArcSinApprox1g(X, A, B, C, D : Double) : Double;
asm
  //Result := A*X*X*X + B*X*X + C*X + D;
  fld      qword ptr [esp+$20]
  fmul    qword ptr [esp+$28]
  fmul    qword ptr [esp+$28]
  fmul    qword ptr [esp+$28]
  fld      qword ptr [esp+$18]
  fmul    qword ptr [esp+$28]
  fmul    qword ptr [esp+$28]
  faddp
```

```
  fld      qword ptr [esp+$10]
  fmul    qword ptr [esp+$28]
  faddp
  fadd    qword ptr [esp+$08]
  //fst    qword ptr [esp-$08]
  wait
end;
```

This function implementation scores the benchmark 47405 Instead of writing all the qword ptr [esp+$xx] lines we can write the names of the variables and let the compiler translate them into addresses.

This actually makes the code more robust.If the location of the variables should change then the code breaks if we use hard coded addressing. This will however only happen if the calling convention is changed and this is not likely to happen very often.

这个函数的基准测试分数是 47405， 使用命名的变量替换掉所有的 [esp+$xx] 行，让编译器将他们翻译成地址。

实际上，这使代码更健壮。如果我们使用硬编码地址，会受到本地的变量改变的影响。

使用最新代码，如果调用约定改变，也不会对它产生影响。当然，这几乎不可能发生。

```
function ArcSinApprox1g_2(X, A, B, C, D : Double) : Double;
asm
  //Result := A*X*X*X + B*X*X + C*X + D;
  //fld      qword ptr [esp+$20]
  fld      A
  //fmul    qword ptr [esp+$28]
  fmul    X
  //fmul    qword ptr [esp+$28]
  fmul    X
  //fmul    qword ptr [esp+$28]
  fmul    X
  //fld      qword ptr [esp+$18]
  fld      B
  //fmul    qword ptr [esp+$28]
  fmul    X
  //fmul    qword ptr [esp+$28]
  fmul    X
  faddp
  //fld      qword ptr [esp+$10]
  fld      C
  //fmul    qword ptr [esp+$28]
  fmul    X
  faddp
  //fadd    qword ptr [esp+$08]
  fadd    D
```

```
  wait
end;
```

Try having both types of lines enabled
试试着两种类型

```
fld     qword ptr [esp+$20]
fld     A
```

and see in the CPU view how the compiler generated exactly the same code for both versions.
X is used in a lot of lines and it is referenced on the stack.
Therefore it is loaded from the stack into the internal FP registers every time.
It should be faster to load it once into the FP stack and let all uses reference the FP stack.

通过观察 CPU 窗口发现，编译器为这两个版本产生了相同的代码。
堆栈上的 X 被许多行使用。
每次使用都是从堆栈中装入 FP 寄存器。
将它只装入 FP 堆栈一次，所有使用它的都参考 FP 堆栈，这应该是比较快的。

```
function ArcSinApprox1h(X, A, B, C, D : Double) : Double;
asm
  //Result := A*X*X*X + B*X*X + C*X + D;
  fld     qword ptr [esp+$20]
  fld     qword ptr [esp+$28] //New
  fxch
  //fmul qword ptr [esp+$28]
  fmul    st(0),st(1)
  //fmul qword ptr [esp+$28]
  fmul    st(0),st(1)
  //fmul qword ptr [esp+$28]
  fmul    st(0),st(1)
  fld     qword ptr [esp+$18]
  //fmul qword ptr [esp+$28]
  fmul    st(0),st(2)
  //fmul qword ptr [esp+$28]
  fmul    st(0),st(2)
  faddp
  fld     qword ptr [esp+$10]
  //fmul qword ptr [esp+$28]
  fmul    st(0),st(2)
  ffree st(2)
  faddp
  fadd    qword ptr [esp+$08]
  fst     qword ptr [esp-$08]
  wait
end;
```

The second line is one we added and it loads X once and for all.
Because it places X on the top of the stack in st(0) and this position is needed as temp for further operations we exchange st(0) with st(1) with the fxch instruction.
We could of course have changed the position of line 1 and 2 and obtained the same effect.
All the lines multiplying

我们增加第二行来一次性装入 X。
由于 X 在栈顶 st(0)，这个位置需要作为其他操作的临时位置，所以我们用 fxch 指令交换 st(0), st(1)。
我们当然可以通过改变第 1，2 行的位置来获得同样的效果。
所有的乘法都用
st(0) with X st(0) 乘 X
fmul qword ptr [esp+$28]
are changed to 改为
fmul st(0),st(1)
after the FP copy of X has been used for the last time we remove it with the instruction ffree.
This function implementation scores the benchmark 46882 and performance is decreased by 1 %.、
This was a surprise. Fxch is claimed by Intel to be nearly free, because it works by renaming the internal registers.
Let us check that by removing it

在最后一次使用完 X 在 FP 上的拷贝之后，用 free 指令删除它。
这个函数的基准测试分数是 46882，性能减少了 1 个百分点。
真令人惊奇。Fxch 是在 Intel 上几乎不耗费时间，因为它只是重命名内部寄存器。

```
function ArcSinApprox1i(X, A, B, C, D : Double) : Double;
asm
  //Result := A*X*X*X + B*X*X + C*X + D;
  fld      qword ptr [esp+$28]
  fld      qword ptr [esp+$20]
  //fld      qword ptr [esp+$28]
  //fxch
  fmul    st(0),st(1)
  fmul    st(0),st(1)
  fmul    st(0),st(1)
  fld      qword ptr [esp+$18]
  fmul    st(0),st(2)
  fmul    st(0),st(2)
  faddp
  fld      qword ptr [esp+$10]
  fmul    st(0),st(2)
  ffree st(2)
  faddp
  fadd    qword ptr [esp+$08]
```

```
  wait
end;
```

This function implementation scores the benchmark 45393 and performance is decreased by 3 %.
Fxch is surely not to blame because performance once more went down. What is going on?
The wait instruction was discussed in an earlier lesson and this time we just remove it.

这个函数的基准测试分数是 45393，性能减少了 3 个百分点。
真的不能责备 Fxch，性能再一次往下降。
接下来干什么？
Wait 指令在前面的课程中讨论过了，这一次我们移除它。

```
function ArcSinApprox1j(X, A, B, C, D : Double) : Double;
asm
  //Result := A*X*X*X + B*X*X + C*X + D;
  fld      qword ptr [esp+$28]
  fld      qword ptr [esp+$20]
  fmul     st(0),st(1)
  fmul     st(0),st(1)
  fmul     st(0),st(1)
  fld      qword ptr [esp+$18]
  fmul     st(0),st(2)
  fmul     st(0),st(2)
  faddp
  fld      qword ptr [esp+$10]
  fmul     st(0),st(2)
  ffree st(2)
  faddp
  fadd     qword ptr [esp+$08]
  //wait
end;
```

Performance went down to 44140.
Let us crosscheck these surprising results by running the functions on a P3.
性能下降到 44140。
我们在一个 P3 上反复测试这些令人惊讶的结果。

ArcSinApprox1a	63613
ArcSinApprox1b	64412
ArcSinApprox1c	64433
ArcSinApprox1d	65062
ArcSinApprox1e	64830
ArcSinApprox1f	62598
ArcSinApprox1g	79586
ArcSinApprox1h	85361

ArcSinApprox1i 80515
ArcSinApprox1j 80192
First of all we see that ArcSinApprox1h is the fastest function on P3.
Thereby it is seen that loading data from the level 1 data cache is more expensive on P3 than on P4,because changing the code such that X is loaded only once improved performance on P3 and not on P4.
On the other hand we could also say that it must always be slower to get data from the cache than from internal registers if the architecture is sound and this is only true for the P6 architecture here.
P4 has a fast L1 data cache, which can be read in only 2 clock cycles, but an internal register read should have a latency of only one cycle. It however looks like reads from registers are 2 clock cycles.
Then we see that a P3 at 1400 nearly 80 % faster than a P4 at 1920 on this code. We know that latencies on P3 are shorter, but this is not enough to explain the huge difference.

首先我们看到在 P3 上 ArcSinApprox1h 是最快的。
看起来，从一级数据 cache 装载数据 P3 比 P4 更珍贵，因为只装入 X 一次的代码在 P3 上可以增加性能，在 P4 上不能。
另一方面，我们可以看到从 cache 中获得数据比从内部寄存器中慢，条件是体系结构是完整的，这只在 P6 上有效。
P4 有一个快速的一级数据 cache，读时间之后 2 个时钟周期，但读内部寄存器只有一个周期的潜伏期。然而，它看起来读寄存器使用了 2 个时钟周期。我们看到这个代码，在 P3 1400 上比 P4 1920 上快了将近 80%。
我们知道在 P3 上的潜伏期时短的，但是不能解释为什么会有这么大的差别。

The latencies and throughput of the used instructions are on P3
这些指令在 P3 上的潜伏期和吞吐量是
Fadd latency is 3 clock cycles and throughput is 1
Fmul latency is 5 clock cycles and throughput is 1
Fadd 潜伏期是 3 个时钟周期，吞吐量是 1
Fmul 潜伏期是 5 个时钟周期，吞吐量是 1
On P4 在 P4 上
Fadd 潜伏期是 5 个时钟周期，吞吐量是 1
Fmul 潜伏期是 7 个时钟周期，吞吐量是 2

I could not find numbers for fld The explanation for the very bad performance of P4 on this code must be the 2-cycle throughput on fmul together with the slow FP register access.
The fmul pipeline only accepts a new instruction on every second cycle where the P3 pipeline accepts one every cycle.

我没有发现 fld 的数据，这个代码在 P4 上性能很差的解释是，乘法和慢的 FP 寄存器访问是 2 个周期的吞吐量。
fmul 管道线在 P4 上每两个周期接收一个新指令，在 P3 管道线上每个周期接收一个。

Scaling the results by clock frequency

结果与主频的比率是
47939 / 1920 = 25
85361 / 1400 = 61

reveals that clock by clock on the fastest function version for each processor P3 is nearly 2.5 times faster than P4.This is truly astonishing. If P4 should have a slight chance against a P3 we must remove some of those multiplications.
This leads us to the Horner version of our function.

这说明了，在 P3 处理器上最快的函数版本比 P4 快 2.5 倍。
这真是令人惊讶。如果在 P4 上要做一些改变，我们必须移除那些乘法操作。
现在，进入我们的括号版本

```
function ArcSinApprox3a(X, A, B, C, D : Double) : Double;
begin
  Result := ((A*X + B)*X + C)*X + D;
end;
```

Which is compiled into 被编译成

```
function ArcSinApprox3b(X, A, B, C, D : Double) : Double;
begin
{
push ebp
mov    ebp,esp
add    esp,-$08
}
  Result := ((A*X + B)*X + C)*X + D;
  {
  fld    qword ptr [ebp+$20]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$18]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$10]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$08]
  fstp qword ptr [ebp-$08]
  wait
  fld    qword ptr [ebp-$08]
  }
{
pop    ecx
pop    ecx
pop    ebp
}
end;
```

The first three versions of this function are identical and they surprisingly score the same benchmark.Our benchmark is not perfect but it was precise this time ;-)
前三个版本的函数的基准测试是相同的，真令人惊奇。
我们的基准测试不完美，但是他可以精确时间：

ArcSinApprox3a 45076
ArcSinApprox3b 45076
ArcSinApprox3c 45076

Optimization follows the same pattern as on the first function.
Here is the first BASM version with no optimizations.
The out commented the compiler supplies code.

像优化第一个函数一样优化它。
这是没有优化的第一个 BASM 版本，注释掉编译器产生的代码

```
function ArcSinApprox3c(X, A, B, C, D : Double) : Double;
asm
  //push ebp
  //mov ebp,esp
  add    esp,-$08
  //Result := ((A*X + B)*X + C)*X + D;
  fld    qword ptr [ebp+$20]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$18]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$10]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$08]
  fstp qword ptr [ebp-$08]
  wait
  fld    qword ptr [ebp-$08]
  pop    ecx
  pop    ecx
  //pop ebp
end;
```

First thing is to remove the add esp, -$08 line and the two pop ecx.
They are setting up a stack frame and do nothing but manipulate the stack pointer, which is not used at all.

首先移除 add esp, -$08 和两行 pop ecx。
他们设置一个堆栈，除了操作堆栈指针没有做任何事情，没有任何用处。

```
function ArcSinApprox3d(X, A, B, C, D : Double) : Double;
asm
  //add    esp,-$08
  //Result := ((A*X + B)*X + C)*X + D;
  fld    qword ptr [ebp+$20]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$18]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$10]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$08]
  fstp qword ptr [ebp-$08]
  wait
  fld    qword ptr [ebp-$08]
  //pop    ecx
  //pop    ecx
end;
```

This function implementation scores the benchmark 43535.
Both of the redundant lines copying the result to stack and back are removed at the same time.

这个函数的基准测试分数是 43535.
删除复制到堆栈,再从堆栈复制的冗余代码

```
function ArcSinApprox3e(X, A, B, C, D : Double) : Double;
asm
  //Result := ((A*X + B)*X + C)*X + D;
  fld    qword ptr [ebp+$20]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$18]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$10]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$08]
  //fstp qword ptr [ebp-$08]
  wait
  //fld    qword ptr [ebp-$08]
end;
```

This function implementation scores the benchmark 47237 and the improvement is 8.5 %
Then we change the code such that X is loaded only once.

这个函数的基准测试性能是 47237, 提高了 8.5%.
然后,我们修改成只装载一次 X 的的代码.

```
function ArcSinApprox3f(X, A, B, C, D : Double) : Double;
```

```
asm
  //Result := ((A*X + B)*X + C)*X + D;
  fld      qword ptr [ebp+$20]
  fld      qword ptr [ebp+$28]
  fxch
  //fmul qword ptr [ebp+$28]
  fmul    st(0),st(1)
  fadd    qword ptr [ebp+$18]
  //fmul qword ptr [ebp+$28]
  fmul    st(0),st(1)
  fadd    qword ptr [ebp+$10]
  //fmul qword ptr [ebp+$28]
  fmul    st(0),st(1)
  ffree st(1)
  fadd qword ptr [ebp+$08]
  wait
end;
```

This function implementation scores the benchmark 47226 and performance is unchanged.

The ffree instruction can be removed by using fmulp instead of fmul, but to do this we must interchange the two registers used. Only these two registers are in use and A*B = B*A so there is no problem doing that.

We are not removing any redundancy by this and the two ways of coding the same thing should perform identically.

这个函数的基准测试分数是 47226，性能没有改变。

用 fmulp 指令代替 fmul 之后，ffree 可以删除，使用它之后必须在内部交换用到的两个寄存器。这两个寄存器只用来执行 A*B = B*A，因此交换之后对结果没有任何影响。

用这种方法我们不能消除任何冗余，这两种编码方法执行相同的操作。

```
function ArcSinApprox3g(X, A, B, C, D : Double) : Double;
asm
  //Result := ((A*X + B)*X + C)*X + D;
  fld      qword ptr [ebp+$20]
  fld      qword ptr [ebp+$28]
  fxch    st(1)
  fmul    st(0),st(1)
  fadd    qword ptr [ebp+$18]
  fmul    st(0),st(1)
  fadd    qword ptr [ebp+$10]
  //fmul    st(0),st(1)
  fmulp st(1),st(0)
  //ffree st(1)
  fadd qword ptr [ebp+$08]
  wait
```

```
end;
```

This function implementation scores the benchmark 47416.
Then we remove the wait instruction.

这个函数的基准测试分数是 47416。
我们移除 wait 指令。

```
function ArcSinApprox3h(X, A, B, C, D : Double) : Double;
asm
  //Result := ((A*X + B)*X + C)*X + D;
  fld      qword ptr [ebp+$20]
  fld      qword ptr [ebp+$28]
  fxch     st(1)
  fmul     st(0),st(1)
  fadd     qword ptr [ebp+$18]
  fmul     st(0),st(1)
  fadd     qword ptr [ebp+$10]
  fmulp st(1),st(0)
  fadd qword ptr [ebp+$08]
  //wait
end;
```

This function implementation scores the benchmark 47059.
The last thing to do is interchanging the lines that load X and A, and remove the fxch instruction.
这个函数的基准测试分数是 47059。
最后一件事是交换 load X 和 A，删除 fxch 指令。

```
function ArcSinApprox3i(X, A, B, C, D : Double) : Double;
asm
  //Result := ((A*X + B)*X + C)*X + D;
  fld      qword ptr [ebp+$28]
  fld      qword ptr [ebp+$20]
  //fld      qword ptr [ebp+$28]
  //fxch     st(1)
  fmul     st(0),st(1)
  fadd     qword ptr [ebp+$18]
  fmul     st(0),st(1)
  fadd     qword ptr [ebp+$10]
  fmulp st(1),st(0)
  fadd qword ptr [ebp+$08]
end;
```

This function implementation scores the benchmark 46544 and performance went down!
Let us compare the performance of the Horner style function with the naive one by picking the fastest implementations of both on P4.
这个函数的基准测试分数是 46544，性能下降了。

我们在 P4 上比较括号类型的函数和执行最快的函数：
ArcSinApprox1g 47939
ArcSinApprox3g 47416

On P3 在 P3 上
ArcSinApprox1h 85361
ArcSinApprox3h 87604
There difference is not big, but the naive approach is a little faster on P4 and slower on P3.
The naive approach has more calculations, but parallelism makes up for it. The Horner way has very little parallelism and latencies are fully exposed.
This is especially bad on P4.
Having this in mind we continue to the third possible approach, which looks like this.

结果差别不是很大，但是原始函数在 P4 上快点，在 P3 上慢。
原始函数计算多，但是被并行执行。
括号函数几乎不并行执行，潜伏期被浪费。
在 P4 上特别的差。
我们特意进行第三种可能的考虑，像这样：

```
function ArcSinApprox4b(X, A, B, C, D : Double) : Double;
begin
{
push    ebp
mov    ebp,esp
add      esp,-$08
}
  Result := (A*X + B)*(X*X)+(C*X + D);
  {
  fld          qword ptr [ebp+$20]
  fmul    qword ptr [ebp+$28]
  fadd    qword ptr [ebp+$18]
  fld          qword ptr [ebp+$28]
  fmul    qword ptr [ebp+$28]
  fmulp st(1)
  fld          qword ptr [ebp+$10]
  fmul    qword ptr [ebp+$28]
  fadd      qword ptr [ebp+$08]
  faddp st(1)
  fstp      qword ptr [ebp-$08]
  wait
  fld      qword ptr [ebp-$08]
  }
{
pop ecx
```

```
pop ecx
pop ebp
}
end;
```

Experienced as we are now optimizing this function is going to be easy and fast ;-)
This version is as Delphi made it
有经验的我们将很轻快的优化这个函数。
这是 Delphi 的版本：

```
function ArcSinApprox4c(X, A, B, C, D : Double) : Double;
asm
  //push ebp
  //mov ebp,esp
  add      esp,-$08
  //Result := (A*X + B)*(X*X)+(C*X + D);
  fld          qword ptr [ebp+$20]
  fmul    qword ptr [ebp+$28]
  fadd    qword ptr [ebp+$18]
  fld          qword ptr [ebp+$28]
  fmul    qword ptr [ebp+$28]
  fmulp //st(1)
  fld          qword ptr [ebp+$10]
  fmul    qword ptr [ebp+$28]
  fadd    qword ptr [ebp+$08]
  faddp //st(1)
  fstp      qword ptr [ebp-$08]
  wait
  fld      qword ptr [ebp-$08]
  pop      ecx
  pop      ecx
//pop    ebp
end;
```

Removing the stack frame and the two lines that store the result in the stack frame
删除与堆栈相关的代码：

```
function ArcSinApprox4d(X, A, B, C, D : Double) : Double;
asm
  //add    esp,-$08
  //Result := (A*X + B)*(X*X)+(C*X + D);
  fld    qword ptr [ebp+$20]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$18]
  fld    qword ptr [ebp+$28]
  fmul qword ptr [ebp+$28]
```

```
  fmulp //st(1)
  fld    qword ptr [ebp+$10]
  fmul qword ptr [ebp+$28]
  fadd qword ptr [ebp+$08]
  faddp //st(1)
  //fstp qword ptr [ebp-$08]
  wait
  //fld    qword ptr [ebp-$08]
  //pop    ecx
  //pop    ecx
end;
```

Load X once

只装载一次 X

```
function ArcSinApprox4e(X, A, B, C, D : Double) : Double;
asm
  //Result := (A*X + B)*(X*X)+(C*X + D);
  fld      qword ptr [ebp+$20]
  fld      qword ptr [ebp+$28]
  //fmul qword ptr [ebp+$28]
  fxch
  fmul    st(0),st(1)
  fadd    qword ptr [ebp+$18]
  //fld    qword ptr [ebp+$28]
  fld      st(1)
  //fmul    qword ptr [ebp+$28]
  fmul    st(0),st(2)
  fmulp
  fld      qword ptr [ebp+$10]
  //fmul    qword ptr [ebp+$28]
  fmul    st(0),st(2)
  fadd    qword ptr [ebp+$08]
  faddp
  ffree st(1)
  wait
end;
```

Remove fxch and wait.

移除 fxch 和 wait

```
function ArcSinApprox4f(X, A, B, C, D : Double) : Double;
asm
  //Result := (A*X + B)*(X*X)+(C*X + D);
  fld      qword ptr [ebp+$28]
  fld      qword ptr [ebp+$20]
```

```
  //fxch
  fmul    st(0),st(1)
  fadd    qword ptr [ebp+$18]
  fld     st(1)
  fmul    st(0),st(2)
  fmulp
  fld     qword ptr [ebp+$10]
  fmul    st(0),st(2)
  fadd    qword ptr [ebp+$08]
  faddp
  ffree st(1)
  //wait
end;
```

Reschedule ffree st(1)

重调整 ffree string(1)

```
function ArcSinApprox4g(X, A, B, C, D : Double) : Double;
asm
  //Result := (A*X + B)*(X*X)+(C*X + D);
  fld     qword ptr [ebp+$28]
  fld     qword ptr [ebp+$20]
  fmul    st(0),st(1)
  fadd    qword ptr [ebp+$18]
  fld     st(1)
  fmul    st(0),st(2)
  fmulp
  fld     qword ptr [ebp+$10]
  fmul    st(0),st(2)
  ffree st(2)
  fadd    qword ptr [ebp+$08]
  faddp
  //ffree st(1)
end;
```

Replace fmul/ffree by fmulp

用 fmulp 代替 fmul/ffree

```
function ArcSinApprox4h(X, A, B, C, D : Double) : Double;
asm
  //Result := (A*X + B)*(X*X)+(C*X + D);
  fld     qword ptr [ebp+$28]
  fld     qword ptr [ebp+$20]
  fmul    st(0),st(1)
  fadd    qword ptr [ebp+$18]
  fld     st(1)
  fmul    st(0),st(2)
```

```
  fmulp
  fld      qword ptr [ebp+$10]
  //fmul    st(0),st(2)
  fmulp st(2),st(0)
  //ffree st(2)
  fadd    qword ptr [ebp+$08]
  faddp
end;
```

Cleaning up and observing that the compiler still backs up ebp and modifies esp redundantly.
整理并且注意到编译器仍将多余的备份 ebp 和修改 esp。

```
function ArcSinApprox4i(X, A, B, C, D : Double) : Double;
asm
  //Result := (A*X + B)*(X*X)+(C*X + D);
  fld      qword ptr [ebp+$28]
  fld      qword ptr [ebp+$20]
  fmul    st(0),st(1)
  fadd    qword ptr [ebp+$18]
  fld      st(1)
  fmul    st(0),st(2)
  fmulp
  fld      qword ptr [ebp+$10]
  fmulp st(2),st(0)
  fadd    qword ptr [ebp+$08]
  faddp
end;
```

The big question is now how well this function version performs.
现在最大的问题时怎样适当的执行这些函数：

ArcSinApprox4a	45228
ArcSinApprox4b	45239
ArcSinApprox4c	45228
ArcSinApprox4d	51813
ArcSinApprox4e	49044
ArcSinApprox4f	48674
ArcSinApprox4g	48852
ArcSinApprox4h	44914
ArcSinApprox4i	44914

We see that “optimizations” from function d to i are “deoptimizations” on P4 except for g.
我们看到从 d 到 i，除了 g 之外在 P4 上是“非优化”的。

On P3 在 P3 上

ArcSinApprox4a	68871
ArcSinApprox4b	68871

ArcSinApprox4c 68634
ArcSinApprox4d 86806
ArcSinApprox4e 85727
ArcSinApprox4f 83542
ArcSinApprox4g 80548
ArcSinApprox4h 88378
ArcSinApprox4i 85324
We see that optimizations d and h are very good and optimizations e, f g and I are bad.
It is quite possible that the optimal function implementation is none of the ones we have made.
We could pick version h and remove the bad optimizations or simply make some more variants and this way get a faster implementation.
我们看到优化的 d 和 h 是非常的好，优化的 e,f,g 和 I 效果很差。
很有可能我们已经执行的函数中没有最佳的函数。
我们改变一下版本 h，删除差的优化，做一些简单的变化，得到一个更快的执行。

Which function approach is the winner? To find out we pick the fastest implementation of each approach On P4
哪个函数是胜利者呢？
我们找出在 P4 上执行最快的函数

ArcSinApprox1f 47939
ArcSinApprox3g 47416
ArcSinApprox4d 51813
The last version is the fastest.
Parallelisation is very important on a modern processor and version 4 beats the others by 9 %.

最后的版本最快。
并行处理在现代处理器上非常重要，版本 4 比其他版本快 9%。

On P3 在 P3 上
ArcSinApprox1h 85361
ArcSinApprox3h 87604
ArcSinApprox4h 88378
The function version 4 is a winner on P3 as well, but with a less margin.
The P4 has an SSE2 instruction set, which contains instructions for double precision floating-point calculations.
The main idea with this instruction set is SIMD calculations.

版本 4 在 P3 上也是冠军，但是只有微弱的优势。
P4 有 SSE2 指令集，它包含双精度浮点数指令计算。
这些指令主要用于 SIMD 计算。

SIMD is an acronym for Single Instruction Multiple Data.
SIMD 是单指令多数据 (Single Instruction Multiple Data) 的缩写。

Multiple data is here two FP double precision variables (64 bit) and two sets of these data can be added, subtracted, multiplied or divided with one instruction.
SSE2 also have some instructions for scalar calculations, which are calculations on
one pair of data just like ordinary FP math in the FPU.

多数据在这里是指两个 FP 浮点数精度变量(64 位)，两个这样的数据可以用一条指令来执行加、减、乘、除。SSE2 也有普通的数量计算，它们也可以像在 FPU 中执行普通的算术运行一样，计算一对数据。

The biggest difference between ordinary FP math and SSE2 scalar math is that FP math is performed on extended precision and results are rounded to double precision when copied to double precision variables in RAM/cache.
SSE2 math is double precision calculation and double precision registers.

普通 FP 算术和 SSE2 分级算术的最大不同是，FP 算术使用扩展的精度执行，当用 RAM/cache 复制到双精度变量时，结果也被
扩展为双精度。SSE2 算术是双精度计算和双精度寄存器。

The code examples of this lesson have relatively few calculations and precision on the FPU will be double. If we load data, perform all calculations and store the result, the result will only bee a little less than extended precision when still on the FPU stack, and will be rounded to exact double precision when copied to a memory location.

这一课的代码例子中，在 FPU 上与计算精度有关，也有几个是关于双精度的。
如果装载数据，执行所有的计算，存储结果，则结果只比在 FPU 堆栈上的扩展精度差一点。当复制到本地内存时，将被转为精确的双精度。

SSE2 calculations on the other hand are a little less than double precision and the result
in a register is a little less than double precision too. If there is only one calculation the result will be in double precision, but when performing additional calculations the error from each will sum up. Because the FPU does all calculations in extended precision and can hold temporary results in registers,there can be done many calculations before precision declines below double.
We have seen that the drawback of SSE2 is that precision is double or less versus the double precision of IA32 FP.

SSE2 计算在另一方面也比双精度差，那就是在寄存器中结果也比双精度差。
如果只有一个计算结果则将用双精度，但是当执行累加计算时，将产生错误。
因为 FPU 用扩展的精度进行所有的计算，可以将临时结果存入寄存器，在双精度计算以前也可以做许多计算。
我们也看到 SSE2 的缺点是精度是双倍的，其小于 IA32 FP 的双精度。

What is the advantage? There are two advantages. Registers are not ordered as a stack, which makes it easier to arrange code and secondly calculations

in double precision are faster than calculations in extended precision.
SSE2 的优点是什么呢？它有两个优点。
寄存器没有作为堆栈的顺序，可以更方便的安排代码，第二，用双精度的计算比用扩展精度的计算更快。

We must expect scalar SSE2 instructions to have shorter latencies than their IA32 FP counterparts.
我们也希望分级 SSE2 指令的潜伏期也比 IA32 FP 配对的指令短些.

Fadd latency is 5
Fsub latency is 5
Fmul latency is 7
Fdiv latency is 38

Addsd latency is 4
Subsd latency is 4

Mulsd Divsd latency is 35

Fadd 的潜伏期是 5
Fsub 的潜伏期是 5
Fmul 的潜伏期是 7
Fdiv 的潜伏期是 38

Addsd 的潜伏期是 4
Subsd 的潜伏期是 4

Mulsd Divsd 的潜伏期是 35。

The P4 optimizations reference manual has no latency and throughput information for the Mulsd instruction!
We see that latencies are one cycle less for scalar SSE2 in general, but 3 cycles less for division.

在 P4 的优化参考手册中没有 Mulsd 指令的潜伏和吞吐量信息。

Throughput is 吞吐量是
Fadd throughput is 1
Fsub throughput is 1
Fmul throughput is 2
Fdiv throughput is 38

Addsd throughput is 2
Subsd throughput is 2
Mulsd
Divsd latency is 35

Fadd 的吞吐量是 1
Fsub 的吞吐量是 1
Fmul 的吞吐量是 2
Fdiv 的吞吐量是 38

Addsd 的吞吐量是 2
Subsd 的吞吐量是 2

Mulsd Divsd 的吞吐量是 35。

We see that throughput for addsd and subsd surprisingly are the double of fadd and fsub.
All that think SSE2 has dedicated hardware and that SIMD is calculation on two sets of data in parallel raise your hands!
我们惊奇的看到 addsd 和 subsd 的吞吐量是 fadd 和 fsub 的两倍。

From the manual “Optimizations for Intel P4 and Intel Xeon” latency and throughput tables at page C-1 it is seen that all SSE2 FP instructions are executed in the same pipelines as old time FP instructions.
This eans that an SIMD addition as example is generating two microinstructions that execute in the F_ADD pipeline. At clock cycle one the first one enters the pipeline, at the second cycle number 2 enters the pipeline.

在 "Intel P4 和 Intel Xeon 优化" 手册的 C-1 页的潜伏期和吞吐量表中，可以看到所有的 SSE2 FP 指令都是执行在相同的
管道线中，就像以前的 FP 指令一样。
这表示 SIMD 加法就是例子中的一样在 F_ADD 管道线中产生了两个微指令。
第一个指令在第一个时钟周期进入管道线，第二个指令在第二个周期进入管道线。

Because latency is 4 cycles the first one leaves the pipeline at clock cycle 3 and the second one leaves at cycle four. This leads us to expect that a scalar SSE2 add should generate one microinstruction of the same type and have a latency of 3 cycles and a throughput of 1.

因为潜伏期是 4 个周期,第一个指令在时钟周期 3 离开管道线,第二个在周期 2 离开管道线.这使我们也期望分级 SSE2 加法也产生一个相同类型的微指令,有 3 个周期的潜伏期和 1 个吞吐量.

From the tables it is seen that the SIMD version of add, addpd, has the same latency and throughput as the scalar version, addsd. Either there is an error in the tables or the scalar instruction also generates two microinstructions of which one is “blind”, that is have no effect.

从表中可以看到 SIMD 版本的 add, addpd 和分级版本的版本有相同的潜伏期和吞吐量.
可能在表中有错误 或者 分级指令产生的两个微指令中有一个是无效的。

Come on Intel!
To verify the numbers from the table we create some dedicated code and time the instructions.

Intel 加油。
为了检验表中的数值，我们创建一些专门的代码记录指令的时间。

```
procedure TMainForm.BenchmarkADDSDLatency;
var
  RunNo, ClockFrequency : Cardinal;
  StartTime, EndTime, RunTime : TDateTime;
  NoOfClocksPerRun, RunTimeSec : Double;
const
  ONE : Double = 1;
  NOOFINSTRUCTIONS : Cardinal = 895;

begin
  ADDSDThroughputEdit.Text := 'Running';
  ADDSDThroughputEdit.Color := clBlue;
  Update;
  StartTime := Time;
  for RunNo := 1 to MAXNOOFRUNS do
    begin
      asm
        movsd xmm0, ONE
        movsd xmm1, xmm0
        movsd xmm2, xmm0
        movsd xmm3, xmm0
        movsd xmm4, xmm0
        movsd xmm5, xmm0
        movsd xmm6, xmm0
        movsd xmm7, xmm0

        addsd xmm0, xmm1
        addsd xmm0, xmm1
        addsd xmm0, xmm1
        addsd xmm0, xmm1
        addsd xmm0, xmm1
        addsd xmm0, xmm1
        addsd xmm0, xmm1

        //Repeat the addsd block of code such that there are 128 blocks
        // 重复 addsd 代码块，像这样会有 128 个时钟周期
      end;
```

```
  end;
  EndTime := Time;
  RunTime := EndTime - StartTime;
  RunTimeSec := (24 * 60 *60 * RunTime);
  ClockFrequency := StrToInt(ClockFrequencyEdit.Text);
  NoOfClocksPerRun := (RunTimeSec / MaxNoOfRuns) * ClockFrequency * 1000000 /
NOOFINSTRUCTIONS;
  ADDSDThroughputEdit.Text := FloatToStrF(NoOfClocksPerRun, ffFixed, 9, 1);
  ADDSDThroughputEdit.Color := clLime;
  Update;
end;
```

The addsd instructions all operate on the same two registers and therefore they cannot execute in parallel.The second instruction has to wait for the first to finish and the full latency of the instruction is exposed.

addsd 指令操作两个相同的寄存器，因此不能被并行指令。
第二个指令不得不等待第一个指令的结束，指令的整个潜伏期都将暴露。

For measuring throughput insert this block 128 times
为了测量吞吐量，插入 12 个时钟周期

```
addsd xmm1, xmm0
addsd xmm2, xmm0
addsd xmm3, xmm0
addsd xmm4, xmm0
addsd xmm5, xmm0
addsd xmm6, xmm0
addsd xmm7, xmm0
```

Here there are no data decencies between instructions and they can execute in parallel.
Xmm0 is used as source in every line but this does not create a data dependency.
Results from run of the code show us that latency is 4 cycles and throughput is 2 cycles.
This is in consistency with the table numbers.
Let us code the three functions in scalar SSE2 and perform some benchmarks.

这里两个指令之间没有数据依靠，一次它们可以并行执行。
Xmm0 在每一行被用作源操作数，但是没有产生数据依赖。
代码的运行结果告诉我们，潜伏期是 4 周期，吞吐量是 2 个周期。
这与表中的数值一致。
我们用分级 SSE2 指令来编写这三个函数，执行相同的基准测试。

The 8 SSE2 registers are called xmm0-xmm7 and Delphi has no register view for them.
So we must create our own, by creating a global (or local) variable for each register,
put a watch on them and add code in the function to copy register contents to variables.

SSE2 的 8 个寄存器是 xmm0-xmm7，Delphi 没有寄存器来观察它们。
因此我们必须自己创建全局（或者局部）变量来观察每一个寄存器，将寄存器的内存存入变量。

It is somewhat cumbersome to do all this and I am looking forward to Borland creating an xmm register view.
This code shows how I do it.
这样做稍微有一点麻烦，我期待 Borland 创建一个 xmm 寄存器窗口。

```
var
  XMM0reg, XMM1reg, XMM2reg, XMM3reg, XMM4reg : Double;

function ArcSinApprox3i(X, A, B, C, D : Double) : Double;
asm
  //Result := ((A*X + B)*X + C)*X + D;

  fld      qword ptr [ebp+$20]
  movsd xmm0,qword ptr [ebp+$20]

  movsd XMM0reg,xmm0
  movsd XMM1reg,xmm1
  movsd XMM2reg,xmm2
  movsd XMM3reg,xmm3

  fld              qword ptr [ebp+$28]
  movsd xmm1,qword ptr [ebp+$28]

  movsd XMM0reg,xmm0
  movsd XMM1reg,xmm1
  movsd XMM2reg,xmm2
  movsd XMM3reg,xmm3

  fxch        st(1)
  fmul        st(0),st(1)
  mulsd xmm0,xmm1

  movsd XMM0reg,xmm0
  movsd XMM1reg,xmm1
  movsd XMM2reg,xmm2
  movsd XMM3reg,xmm3

  fadd      qword ptr [ebp+$18]
  addsd xmm0,qword ptr [ebp+$18]
```

```
movsd XMM0reg,xmm0
movsd XMM1reg,xmm1
movsd XMM2reg,xmm2
movsd XMM3reg,xmm3

fmul      st(0),st(1)
mulsd xmm0,xmm1

movsd XMM0reg,xmm0
movsd XMM1reg,xmm1
movsd XMM2reg,xmm2
movsd XMM3reg,xmm3

fadd      qword ptr [ebp+$10]
addsd xmm0,qword ptr [ebp+$10]

movsd XMM0reg,xmm0
movsd XMM1reg,xmm1
movsd XMM2reg,xmm2
movsd XMM3reg,xmm3

fmulp st(1),st(0)
mulsd xmm0,xmm1

movsd XMM0reg,xmm0
movsd XMM1reg,xmm1
movsd XMM2reg,xmm2
movsd XMM3reg,xmm3

fadd      qword ptr [ebp+$08]
addsd xmm0,qword ptr [ebp+$08]

movsd XMM0reg,xmm0
movsd XMM1reg,xmm1
movsd XMM2reg,xmm2
movsd XMM3reg,xmm3

movsd [esp-8],xmm0
fld           qword ptr [esp-8]

movsd XMM0reg,xmm0
movsd XMM1reg,xmm1
movsd XMM2reg,xmm2
movsd XMM3reg,xmm3
```

```
  wait
  end;
```

The code is not using xmm4-xmm7 and there was no need to create a register view for them. There is added xmm view code after each line of SSE2 code. All lines but the last two are the FP code with the SSE2 code added such that every operation is done in FP as well as in SSE2. This way it is possible to trace through the code and control that the SSE2 version is doing the same as the classic version.

代码没有使用 xmm4-xmm7，因此不需要为它们创建寄存器显示。
每一行 SSE2 代码的后面都有一个 xmm 相加显示代码。
除了最后两行是 FP 代码以外，其他所有的 SSE2 代码就像在 FP 相加中一样。
这个方法可以跟踪代码和控制 SSE2 的版本就像一个经典的 SSE 2 版本。

Open the FPU view and see how the FP stack is updated and control that xmm registers are updated in the same way. I developed the SSE2 code simply by adding an SSE2 instruction after each line of FP code.

打开 FPU 窗口，查看 FP 堆栈如何更新，用同样的方法控制 xmm 寄存器的更新。
在每一行 FP 代码之后，我简单的增加了一个 SSE2 指令。

```
fld          qword ptr [ebp+$20]
movsd xmm0,qword ptr [ebp+$20]
```

movsd copy one double precision variable from the memory location at [ebp+$20] into an xmm register. “qword ptr” is not needed but I kept it to emphasise the pattern between SSE2 code and FP code. A big difference between FP code and scalar SSE2 code is that the FP registers are organized as a stack and SSE2 registers are not.

movsd 从内存 [ebp+20] 处复制一个双精度变量到一个 xmm 寄存器。
"qword ptr" 不需要，但是我保留它是为了强调 SSE2 和 FP 代码的样式。
FP 代码和 SSE2 代码的最大不同是 FP 寄存器被组织为一个堆栈, SSE2 寄存器不是.

At first while coding the SSE2 code I just ignored this and then after having made all the SSE2 lines I went back and traced through the lines one by one and corrected them to work on the correct variable/register.

当编写 SSE2 代码时我首先忽略这个,然后回去单步跟踪, 确定它们正确地工作在变量或者寄存器中.

Activate the function with some variable values that are easy to follow in the two views (e.g. X=2, A=3, B=4, C=5, D=6), and see that first “2” is loaded, then “3”, then 2 is multiplied by “3” and “2” is overwritten by “6” etc.
函数的一些变量很容易用两种方式观察(例如: X=2, A=3, C=5, D=6)，首先 2 被装入，然后 2

被 3 乘，并且 2 被 6 重写，等等。
The scalar SSE2 counterpart for fmul is mulsd. The sd prefix means Scalar – Double.
分级 SSE2 的 fmul 副本是 mulsd。sd 前缀表示分级 - 双倍。

```
fxch    st(1)
fmul    st(0),st(1)
mulsd xmm0,xmm1
```

The SSE2 counterpart for fadd is addsd. SSE 的 fadd 副本是 addsd。

```
fadd    qword ptr [ebp+$18]
addsd xmm0,qword ptr [ebp+$18]
```

Continue this way line by line. 继续一行行的观察

The FP code leaves the result in st(0), but the SSE2 code leaves the result in an xmm register. Then the result has to be copied from xmm to st(0) via a memory location on the stack.

FP 代码将结果留在 st(0)，但是 SSE2 指令结果存在一个 xmm 寄存器。
然后，结果不得不通过在堆栈将结果从 xmm 传到 st(0)

```
movsd [esp-8],xmm0
fld      qword ptr [esp-8]
```

These two lines do this.
这两行做实现这个操作。

At esp-8, 8 bytes above the top of the stack, there is some place we can use as the temporary location for the result.The first line copies xmm0 to temp and then the last line loads temp on the FP stack. These two lines are overhead that will make small SSE2 functions less effective than their FP cousins.
在 esp-8 处，使栈顶的 8 个字节，这是我们存放结果的临时位置。
这两行是 SSE2 效率较低的 FP 姊妹版。

After having double-checked the SSE2 code we can remove the instrumentation code as well as the old FP code,leaving a nice scalar SSE2 function with only a little extra overhead.
在仔细检查 SSE2 代码之后，我们再删除 FP 代码，剩下的就是一个精简的分级 SSE2 函数，只有一点额外的操作。

```
function ArcSinApprox3j(X, A, B, C, D : Double) : Double;
asm
  //Result := ((A*X + B)*X + C)*X + D;
  movsd xmm0,qword ptr [ebp+$20]
  movsd xmm1,qword ptr [ebp+$28]
```

```
  mulsd xmm0,xmm1
   addsd xmm0,qword ptr [ebp+$18]
   mulsd xmm0,xmm1
   addsd xmm0,qword ptr [ebp+$10]
   mulsd xmm0,xmm1
   addsd xmm0,qword ptr [ebp+$08]
   movsd [esp-8],xmm0
   fld      qword ptr [esp-8]
  end;
```

It can be even nicer if we remove the not needed “qword ptr” text.
移除不需要的 "qword ptr" 文字，函数将更简洁。

```
function ArcSinApprox3j(X, A, B, C, D : Double) : Double;
asm
  //Result := ((A*X + B)*X + C)*X + D;
  movsd xmm0, [ebp+$20]
  movsd xmm1, [ebp+$28]
  mulsd xmm0,xmm1
  addsd xmm0, [ebp+$18]
  mulsd xmm0,xmm1
  addsd xmm0, [ebp+$10]
  mulsd xmm0,xmm1
  addsd xmm0, [ebp+$08]
  movsd [esp-8],xmm0
  fld      qword ptr [esp-8]
end;
```

Change the pointers with the parameter names
用参数名替换指针

```
function ArcSinApprox3j(X, A, B, C, D : Double) : Double;
asm
  //Result := ((A*X + B)*X + C)*X + D;
  movsd xmm0, A
  movsd xmm1, X
  mulsd xmm0,xmm1
  addsd xmm0, B
  mulsd xmm0,xmm1
  addsd xmm0, C
  mulsd xmm0,xmm1
  addsd xmm0, D
  movsd [esp-8],xmm0
  fld      qword ptr [esp-8]
```

end;
Well how does this version perform?
The benchmark is 45882。
This version is somewhat slower than the FP version, which scored 48292.
We need to investigate what is the reason for this.

这个版本的执行效果如何？
基准测试分数是 45822。
这个版本稍微被分数是 48292 的 FP 版本慢。
我们需要研究具体的原因。

Is it the overhead of the last two lines or is it due to the 2-cycle throughput of addsd and mulsd? The overhead can be removed by transferring the result as an out parameter or we can inline the function. It would be interesting for us to see how big an advantage it is to inline this relatively small function. After all there is a good deal of overhead of copying 5 double precision parameters each 8 byte big. Let us see how much code is actually needed for this.

最后两行是问题还是由于 addsd 和 mulsd 是 2 个周期的吞吐量。
通过输出参数或者内联这个函数我们可以移除 overhead。
将函数内联看起来会有很大的益处。
一个好的办法就通过每次复制 8 个字节大小来传输 5 个双精度参数。
我们看看这样需要多少代码:

```
push dword ptr [ebp+$14]
push dword ptr [ebp+$10]
push dword ptr [ebp+$34]
push dword ptr [ebp+$30]
push dword ptr [ebp+$2c]
push dword ptr [ebp+$28]
push dword ptr [ebp+$24]
push dword ptr [ebp+$20]
push dword ptr [ebp+$1c]
push dword ptr [ebp+$18]
call dword ptr [ArcSinApproxFunction]
fstp qword ptr [ebp+$08]
```

No less than 10 push instructions each pushing a 4 byte half of each parameter onto the stack. Imagine that the register calling convention took its name seriously and transferred the parameters on the FP stack instead. Then we would have 5 fld, which would also remove the need to load parameters from the stack in the function. That is – 5 fld in the function would be replaced by 5 fld at the call place of the function and 10 push instructions would go away. This would lead to a dramatic increase in performance.

不少于 10 个 push 指令，每次压入每个参数的 4 个字节到堆栈。设想用这样的寄存器调

用约定来代替通过 FP 堆栈传输参数。然后我们要有 5 个 fld, 来移除函数中从堆栈装入参数部分.这 5 个 fld 用在函数中的 5 处装载, 10 个 push 指令可以删除了。
这将导致性能上的戏剧性的增长。

Inlining the function would also remove the overhead of the call/ret pair
which is a lot less than the overhead of the mega push, and this would give as
a clue about the performance of the updated register2 calling convention ;-).
内联函数也将移除 call/ret 调用对，同时给了我们更新寄存器调用约定性能的启示。

Inlined ArcSinApprox3i 156006
Inlined ArcSinApprox3j 160000
The improvement is an astonishing 400 %.

内联的 ArcSinApprox3i 156006
内联的 ArcSinApprox3j 160000
The improvement is an astonishing 400 %.
令人惊奇的增长了 400%。

I truly wish Borland introduced a true register calling convention for floating point parameters in the near future.The SSE2 version is only 3 % faster than the IA32 version. This could be more on a decent SSE2 implementation.

我真的希望 Borland 在将来可以为浮点数参数引入一种真寄存器调用规则.
SSE2 版本只是比 IA32 快 3%。这将更好的在 SSE2 上执行。

Lesson 7 has hereby come to an end.
You now know nearly all about floating point programming ;-)

第 7 课这就结束了。
你现在几乎知道了关于所有浮点数的编程。

译者注

1、翻译已获原作者 Dennis Christensen 同意；
2、加了一个文档目录，便于了解文档内容；
3、翻译此文纯属个人兴趣，水平有限，有任何意见或建议请发 Email，谢谢；
4、感谢‘天天吃好’朋友参与部分校对；
5、感谢 www.cnpack.com 上的所有朋友；
6、译者最大收获：Dennis Kjaer Christensen 严谨负责的治学态度令人敬佩；
7、关于作者 Dennis Christensen 的几个连接：
http://www.fastcode.dk/
http://www.fastcodeproject.org
http://dennishomepage.gugs-cats.dk/index.htm