Direct compilation of high level languages for Multi-media instruction-sets

Paul Cockshott

November 29, 2000

Contents

1

Multi-media instruction-sets

A number of widely used contemporary processors have instructionset extensions for improved performance in multi-media applications. The aim is to allow operations to proceed on multiple pixels each clock cycle. Such instructionsets have been incorporated both in specialist DSP chips like the Texas C62xx[16] and in general purpose CPU chips like the Intel IA32[6][7] or the AMD K6[1].

1.1 The SIMD model

These instructionset extensions are typically based on the Single Instruction-stream Multiple Data-stream (SIMD) model in which a single instruction causes the same mathematical operation to be carried out on many operands, or pairs of operands at the same time. The SIMD model was originally developed in the context of large scale parallel machines such as the ICL Distributed Array Processor or the Connection Machine. In these systems a single control processor broadcast an instruction to thousands of single bit wide dataprocessors causing each to perform the same action in lock-step. These early SIMD processors exhibited massive data-parallelism, but, with each data processor having its own private memory and data-bus they were bulky machines involving multiple boards each carrying multiple memory chip, data-processor chip pairs. Whilst they used single bit processors, the SIMD model is not dependent on this. It can also be implemented with multiple 8-bit, 16-bit or 32-bit data processors.

The incorporation of SIMD technology onto modern general purpose microprocessors is on a more modest scale than were the pioneering efforts. For reasons of economy the SIMD engine has to be included on the same die as the rest of the CPU. This immediately constrains the degree of parallelism that can be obtained. The constraint does not arise from the difficulties of incorporating large numbers of simple processing units. With contemporary feature sizes, one could fit more than a thousand 1-bit processors on a die. Instead the degree of parallelism is constrained by the width of the CPU to memory data path.

The SIMD model provides for all data-processors to simultaneously transfer words of data between internal registers and corresponding locations in their memory banks. Thus with n data-processors each using

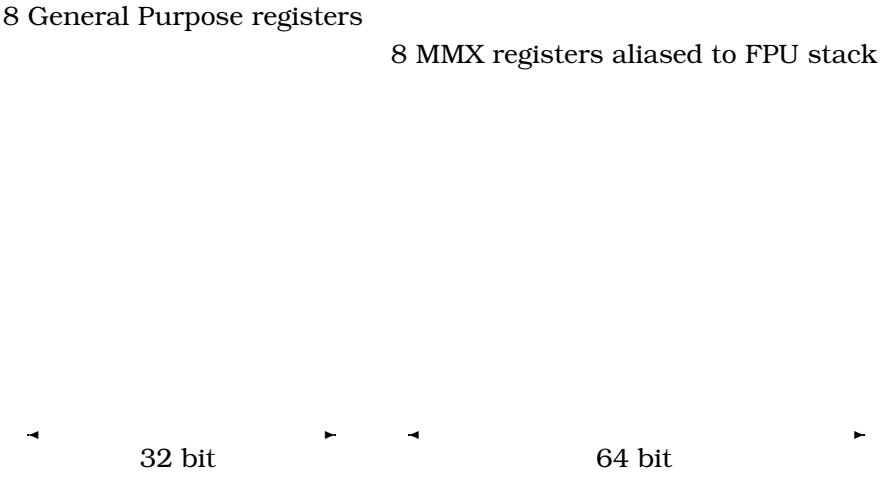

Figure 1.1: The Intel IA32 with MMX register architecture

w-bit words one needs a path to memory of nw bits. If a CPU chip has a 64-bit memory bus then it could support 64 1-bit SIMD data-processors, or 8 8-bit data-processors, 2 32-bit processors, etc.

For bulk data operations, such as those involved in image processing, the relevant memory bus is the off chip bus. For algorithms that can exploit some degree of data locality the relevant bus would be that linking the CPU to the on-chip cache, and the degree of parallelism possible would be constrained by the width of the cache lines used.

Whilst memory access paths constrain the degree of parallelism possible, the large numbers of logic gates available on modern dies, allows the complexity of the individual data-processors to be raised. Instead of performing simple 1-bit arithmetic, they do arithmetic on multi-bit integers and floating point numbers.

As a combined result of these altered constraints we find that SIMD instructions for multi-media applications have parallelism levels of between 32 bits (Texas C62xx) and 128 bits (Intel PIII), and the supported data types range from 8-bit integers to 32-bit floating point numbers.

1.2 The MMX register architecture

The MMX architectural extensions were introduced by Intel in late models of the Pentium processor. They have subsequently been incorporated in the PII and PIII processors from Intel and in compatible chips produced by AMD, Cyrix and others. They can now be considered part of the baseline architecture of any contemporary PC.

The data registers available for computational purposes on processors incorporating the MMX architecture are shown in Figure 1.1. The original IA32 architecture had eight general purpose registers and an eight deep stack of floating point registers. When designing the multi-media extensions to the instructionset, Intel wanted to ensure that no new state bits were added to the process model. Adding new state bits would have made CPU's with the extensions incompatible with existing operating systems, as these would not have saved the additional state on a task switch. Instead, Intel added 8 new virtual 64-bit registers which are aliased onto the existing floating point stack. These new multimedia registers, `mm0..mm7`,

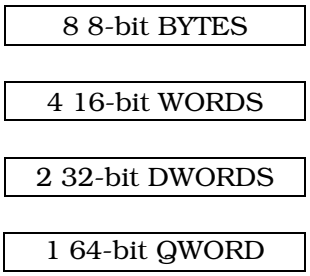

Figure 1.2: The MMX data formats

Table 1.1: MMX data-types

Format	signed	unsigned	signed saturated	unsigned saturated
BYTE	yes	yes	yes	yes
WORD	yes	yes	yes	yes
DWORD	yes	yes	no	no

use state bits already allocated to the Floating Point Unit(FPU), and are thus saved when an operating system saves the state of the FPU.

The MMX instructions share addressing mode formats with the instructions used for the general purpose registers. The 3-bit register identification fields inherited from the previous instructions are now used to index the eight multi-media rather than the eight general purpose registers. The existing addressing modes for memory operands are also carried over allowing the full gamut of base and index address modes to be applied to the loading and storing of MMX operands.

1.3 MMX data-types

The MMX registers support 4 data formats as shown in Figure 1.2. A register can hold a single 64-bit tt QWORD, a pair of 32-bit `DWORDS`, 4 16-bit `WORDS` or 8 `BYTES`. Within these formats the data-types shown in Table 1.1 are supported. The saturated data-types require special comment. They are designed to handle a circumstance that arises frequently in image processing when using pixels represented as integers: that of constraining the result of some arithmetic operation to be within the meaningful bounds of the integer representation.

Suppose we are adding two images represented as arrays of bytes in the range 0..255 with 0 representing black and 255 white. It is possible that the results may be greater than 255. For example 200+175 = 375 but in 8 bit binary
```
    11001000
+   10101111
= 1 01110111
```
droping the leading 1 we get $01110111 = 119$, which is dimmer than either of the original pixels. The only sensible answer in this case would have been 255, representing white.

To avoid such errors, image processing code using 8 bit values has to put in tests to check if values are going out of range, and force all out of range values to the appropriate extremes of the ranges. This inevitably slows down the computation of inner loops. Besides introducing additional

instructions, the tests involve conditional branches and pipeline stalls.

The MMX seeks to obviate this by providing packed saturated data types with appropriate arithmetic operations over them. These use hardware to ensure that the numbers remain in range.

The combined effect of the use of packed data and saturated types can be to produce a significant increase in code density and performance.

Consider the C code in figure 1.3 to add 2 images pointed to by `p1` and `p2`, storing the result in the image pointed to by `p3`. The code includes a check to prevent overflow. Compiled into assembler code by the Visual C++ compiler the resulting assembler code has 18 instructions in the inner loop. The potential acceleration due to the MMX can be seen by comparing it with the following hand coded assembler inner loop:

```
l1: movq mm0,[esi+ebp-LEN]    ; load 8 bytes
    paddusb mm0,[esi+ebp-2*LEN] ; packed unsigned add bytes
    movq [esi+ebp-3*LEN],mm0    ; store 8 byte result
    add esi,8                 ; inc dest pntr
    loop l1                     ; repeat for the rest of the array
```

The example assumes that `p1,p2,p3` are stored in registers `esi,edx,edi` for the duration of the loop. Only 5 instructions are used in the whole loop, compared to 18 for the compiled C code. Furthermore, the MMX code processes 8 times as much data per iteration, thus requiring only 0.625 instructions per byte processed. The compiled code thus executes 29 times as many instructions to perform the same task. Whilst some of this can be put down to the superiority of hand assembled versus automatically compiled code, the combination of the SIMD model and the saturated arithmetic are obviously a major factor.

1.4 3D Now!

The original MMX instructions introduced by Intel were targeted at increasing the performance of 2D image processing, giving their biggest performance boost for images of byte-pixels. The typical operations in 3D graphics, perspective transformations, ray tracing, rendering etc, tend to rely upon floating point data representation. Certain high 2D image processing operations requiring high accuracy such as high precision stereo matching can also be implemented using floating point data. Both Intel and AMD have seen the need to provide for these data representations. AMD responded first with the 3D Now! instructions, then Intel introduced the Steaming SIMD instructions which we discuss in the next section.

The basic IA32 architecture already provides support for 32-bit and 64-bit IEEE floating point instructions using the FPU stack. However, 64-bit floating point numbers are poor candidates for parallelism in view of the datapath limitations described in section 1.1.

AMD provided a straight forward extension of the MMX whereby an additional data-type, the pair of 32 bit floats shown in figure 1.4, could be operated on. Type conversion operations are provided to convert between pairs of 32-bit integers and 32-bit floats.

The range of operators supported includes instructions for the rapid computation of reciprocals and square roots - relevant to the computation of Euclidean norms in 3D-space.

```
main()
{
unsigned char v1[LEN],v2[LEN],v3[LEN];
int i,j,t;
for (j=0;j<LEN;j++){
t=v2[j]+v1[j];
v3[j]=(unsigned char)(t>255?255:t);
}

}
ASSEMBLER
        xor       edx, edx                                      ; 9.8
                                                                ;
                                ; LOE edx ebx ebp esi edi esp dl dh bl bh
$B1$3:                          ; Preds $B1$5 $B1$2
        mov       eax, edx                                      ; 10.9
        lea       ecx, DWORD PTR [esp]                          ; 10.6
        movzx     ecx, BYTE PTR [eax+ecx]                       ; 10.6
        mov       DWORD PTR [esp+19200], edi                    ;
        lea       edi, DWORD PTR [esp+6400]                     ; 10.12
        movzx     edi, BYTE PTR [eax+edi]                       ; 10.12
        add       ecx, edi                                      ; 10.12
        cmp       ecx, 255                                      ; 11.26
        mov       edi, DWORD PTR [esp+19200]                    ;
        jle       $B1$5         ; Prob 16%                      ; 11.26
                                ; LOE eax edx ecx ebx ebp esi edi esp al ah dl dh
$B1$4:                          ; Preds $B1$3
        mov       ecx, 255                                      ; 11.26
                                ; LOE eax edx ecx ebx ebp esi edi esp al ah dl dh
$B1$5:                          ; Preds $B1$3 $B1$4
        inc       edx                                           ; 9.18
        cmp       edx, 6400                                     ; 9.3
        mov       DWORD PTR [esp+19200], edi                    ;
        lea       edi, DWORD PTR [esp+12800]                    ; 11.4
        mov       BYTE PTR [eax+edi], cl                        ; 11.4
        mov       edi, DWORD PTR [esp+19200]                    ;
        jl        $B1$3         ; Prob 80%                      ; 9.3
```

Figure 1.3: C code to add two images and corresponding assembler for the inner loop. Code compiled on the Intel C compiler version 4.0.

Figure 1.4: The AMD 3DNOW! extensions add 32 bit floating point data to the types that can be handled in MMX registers

4 × 32-bit floating point fields

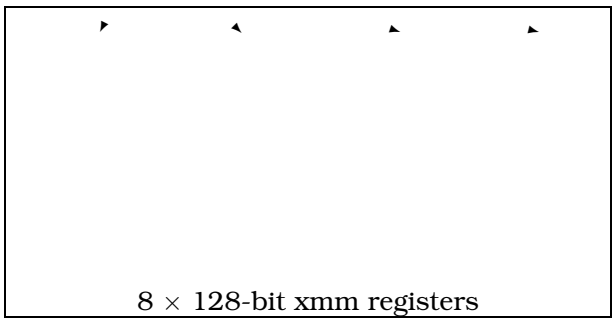

Figure 1.5: The Streaming SIMD extensions add additional 32 bit floating point vector registers

Another significant extension with 3D Now, copied in the Streaming SIMD extensions is the ability to prefetch data into the cache prior to its use. This is potentially useful in any loop operating on an array of data. For instance the loop in the previous section could be acclerated by inserting the marked prefetch instructions.

```
l1: movq mm0,[esi]    ; load 8 bytes
    add esi,8         ; inc src pntr
    prefetch [esi]    ; ensure that the next 8 bytes worth are in the cache
    paddusb mm0,[edx] ; packed unsigned add bytes
    add edx,8         ; inc src pntr
    prefetch [edx]    ; preload 8 bytes from other array
    movq [edi],mm0    ; store 8 byte result
    add edi,8         ; inc dest pntr
    prefetchw [edi]   ; set up the cache ready for writing 8 bytes of data
```

The instruction count rises, despite this performance goes up since loads into the cache are initiated 6 instructions prior to the data being needed. This allows the loading of the cache to be overlapped with useful instructions rather than forcing calculations to stall whilst the load takes place.

1.5 Streaming SIMD

Intel produced their own functional equivalent to AMD's 3DNOW instructionset with the Pentium III processor. They called the new instructions Streaming SIMD. As with 3DNOW, the Streaming SIMD instructions combine cache prefetching techniques with parallel operations on short vectors of 32 bit floating point operands.

The most significant difference is to the model of machine state. Whilst the original MMX instructions and 3DNOW add no new state to the machine architecture, Streaming SIMD introduces additional registers. Eight new 128-bit registers (XMM0..7) are introduced. The addition of new state means that operating systems have to be modified to ensure that xmm registers are saved during context switches. Intel provided a driver to do

Table 1.2: The XMM registers support both scalar and vector arithmetic

Vector addition						
	xmm0	1.2	1.3	1.4	1.5	
ADDPS xmm0,xmm1	xmm1	2.0	4.0	6.0	8.0	+
	xmm0	3.2	5.3	7.4	9.5	
Scalar addition						
	xmm0	1.2	1.3	1.4	1.5	
ADDSS xmm0,xmm1	xmm1	2.0	4.0	6.0	8.0	+
	xmm0	1.2	1.3	1.4	9.5	

this for Microsoft Windows NT 4.0, Windows 98 and subsequent Windows releases have this support built in.

The default format for the XMM registers is a 4-tuple of 32 bit floating point numbers. Instructions are provided to perform parallel addition, multiplication, subtraction and division on these 4-tuples. Two very useful extensions to this format are provided.

1. A set of boolean operations are provided that treat the registers as 128 bit words, useful for operations on bitmps.

2. Scalar operations are provided that operate only on the lower 32 bits of the register. This allows the XMM registers to be used for conventional single precision floating point arithmetic. Whilst the pre-existing Intel FPU instructions support single precision arithmetic, the original FPU is based on a reverse Polish stack architecture. This scheme does not fit well with the register allocation schemes used in some compilers. The existence of what are effectively eight scalar floating point registers can lead to more efficient floating point code.

The scalar and vector uses of the XMM registers are contrasted in table 1.2. A special move instruction (MOVSS) is provided to load or store the least significant 32 bits of an XMM register.

1.5.1 Cache optimisation

The Streaming side of the Streaming SIMD extensions is concerned with optimising the use of the cache. The extensions will typically be used with large collections of data, too large to fit into the cache. If an application were adding two vectors of a million floating point registers using standard instructions, the 4MB of results would pollute the cache. This cache pollution can be avoided using the *non-temporal* store instructions, MOVNTPS and MOVNTQ operating on the XMM and MM registers respectively.

A PREFETCH instruction is provided to preload data into the cache. This is more sophisticated than the equivalent 3DNOW! instruction described above. The AMD instruction applies to all cache levels. The Intel variant allows the programmer to specify which levels of cache are to be preloaded.

Whereas all previous IA32 load and store instructions had operated equally well on aligned and unaligned data. The Streaming SIMD extensions introduces special load and store instructions to operate on aligned 128 bit words. General purpose load and store instructions capable of handling unaligned data are retained.

2

Possible programming techniques for MMX and Streaming SIMD

There is little exploitation of these extensions on specific architectures because of the lack of appropriate compiler support. For example,Intel supply a C compiler that has low level extensions allowing the extended instructions to be used. Intel terms these extensions 'assembler intrinsics'. Syntactically these look like C functions but they are translated one for one into equivalent assembler instructions. The use of assembler intrinsics simplifies the process of developing MMX code, in that programmers use a single tool - the C compiler, and do not need to concern themselves with low level linkage issues. However, the other disadvantages of assembler coding remain. The Intel C compiler comes with a set of C++ classes that correspond to the fundamental types supported by the MMX and SIMD instruction sets. The SIMD classes do a good job of presenting the underlying capabilities of the architecture within the context of the C language. The code produced is also efficient. However, whilst the C++ code has a higher level of expression than assembler intrinsics, it is not portable to other processors. Similarly, SUN's Hotspot Java Optimiser is targeted at the old Intel 486 instructions rather than the newer MMX instructions available since the Pentium.

There are many disadvantages to these approaches. First of all, programmers must have deep knowledge both of low level architectural behaviour and of architecture specific compiler behaviour to integrate assembly language with high level code. Secondly, effective use of libraries depends on there being a close correspondence between the intended semantics of the application program and the semantics of the library routines. Finally, use of architecture specific libraries inhibits program portability across operating systems and CPU platforms.

There has been sustained research within the parallel programming community into the exploitation of SIMD parallelism on multi-processor architectures. Most work in this field has been driven by the needs of high-performance scientific processing, from finite element analysis to meteorology. In particular, there has been considerable interest in exploiting data parallelism in FORTRAN array processing, culminating in High Performance Fortran and F[12]. Typically this involves two approaches. First of all, operators may be overloaded to allow array-valued expressions, sim-

ilar to APL. Secondly, loops may be analysed to establish where it is possible to unroll loop bodies for parallel evaluation. Compilers embodying these techniques tend to be architecture specific to maximise performance and they have been aimed primarily at specialised super-computer architectures, even though contemporary general purpose microprocessors provide similar features, albeit on a far smaller scale.

There has been recent interest in the application of vectorisation techniques to instruction level parallelism. Thus, Cheong and Lam [4] discuss the use of the Stanford University SUIF parallelising compiler to exploit the SUN VIS extensions for the UltraSparc from C programs. They report speedups of around 4 on byte integer parallel addition. Krall and Lelait's compiler [10] also exploits the VIS extensions on the Sun UltraSPARC processor from C using the CoSy compiler framework. They compare classic vectorisation techniques to unrolling, concluding that both are equally effective, and report speedups of 2.7 to 4.7. Sreraman and Govindarajan [14] exploit Intel MMX parallelism from C with SUIF, using a variety of vectorisation techniques to generates inline assembly language, achieving speedups from 2 to 6.5. All of these groups target specific architectures. Finally, Leupers [11] has reported a C compiler that uses vectorising optimisation techniques for compiling code for the multimedia instruction sets of some signal processors, but this is not generalised to the types of processors used in desktop computers.

In this chapter I look in detail at possible techniques for programming the MMX hardware. In order to illustrate the techniques I will use a sample program that adds two arrays of bytes storing the result in a third. The arrays will be 6400 bytes long, and the operation will be repeated 100,000 times.

Timings for the program are given for a 233Mhz Sony Vaio. The test program therefore requires 640,000,000 add operations.

The program is used purely for illustrative purposes, it is not realistic as a benchmark of general performance on the MMX.

2.1 Direct use of assembler code

With instructionsets as complex as those incorporated into the latest Intel and AMD processors, careful hand-written assembler language routines produce the highest quality machine code.

Microsoft's MASM assembler supports the extended instructionset as does the free assembler NASM. The latter has the advantage of running on both Linux and Windows, and provides support for MMX, 3DNOW! and SIMD instructions.

In the absence of better tools, direct coding of inner loops as assembler routines is the obvious course to pursue where run-time performance requirements demand it. Disadvantages of using assembler are well known:

1. It is not portable between processors. A program written in assembler to use the AMD extensions will not run on an Intel processor nor, *a-fortiori*, on an Alpha.

2. It requires the programmer to have an in-depth knowledge of the underlying machine architecture, which only a small proportion of programmers now have.

```
;

SECTION .text ;
global _main
LEN equ 6400
_main: enter  LEN*3,0
    mov ebx,100000     ; per-
form test 100000 times for timing
l0:
    mov esi,0     ; set esi registers to index the elements
    mov ecx,LEN/8     ; set up the count byte
l1: movq mm0,[esi+ebp-LEN]    ; load 8 bytes
    paddb mm0,[esi+ebp-2*LEN] ; packed unsigned add bytes
    movq [esi+ebp-3*LEN],mm0    ; store 8 byte result
    add esi,8       ; inc dest pntr
    loop l1           ; repeat for the rest of the array
    dec ebx
    jnz l0
    mov eax,0
    leave
    ret
;
```

Figure 2.1: Assembler version of the test program

3. Productivity in terms of programmer time spent to implement a given algorithm is lower than in high level languages.

4. The programmer must further master the low level linkage and procedure call conventions of the high level language used for the rest of the application.

5. Programmers have to master additional program development tools.

All of these militate against widespread use. The fact that the AMD extensions can only be programmed in assembler may have been a factor limiting their practical use.

2.1.1 The example program

The assembler version of the example program is shown in figure 2.1. It runs in 4.01 seconds on the test machine, a 233Mhz Pentium II, a throughput of 160 million byte arithetic operations per second.

The C version is shown in figure 2.3. It compiled with the Intel C compiler it runs in 82 seconds on the test machine, a performance of around 8 million arithmetic operations per second. Thus the assembler version using MMX is about 20 times as fast as the C version.

2.2 Use of assembler intrinsics

Intel supply a C compiler that has low level extensions allowing the extended instructions to be used. Intel terms these extensions 'assembler intrinsics'. For example the ADDPS instruction which adds 4 packed single

```
/*
*/
#define LEN 6400
#define CNT 100000
main()
{
  unsigned char v1[LEN],v2[LEN],v3[LEN];
  int i,j,t;
  for(i=0;i<CNT;i++)
     for (j=0;j<LEN;j++) v3[j]=v2[j]+v1[j];
}
/*
*/
```

Figure 2.2: C version of the test program

precision floating point numbers is mirrored by the Intel C/C++ Compiler Intrinsic Equivalent

```
__m128 _mm_add_ps(__m128 a, __m128 b)
```

which adds the four SP FP values of a and b.

Syntactically these look like C functions but they are translated one for one into equivalent assembler instructions. The use of assembler intrinsics simplifies the process of developing MMX code, in that programmers use a single tool - the C compiler, and do not need to concern themselves with low level linkage issues. However, the other disadvantages of assembler coding remain.

1. It is still not portable between processors.
2. It still requires the programmer to have an in-depth knowledge of the underlying machine architecture.
3. Productivity is unlikely to be higher than with assembler.

2.3 Use of C++ classes

The Intel C compiler comes with a set of C++ classes that correspond to the fundamental types supported by the MMX and SIMD instructionsets. For instance, type Iu8vec8 is a vector of 8 unsigned 8 bit integers, Is32vec2 a vector of 2 signed 32 bit integers, etc. The basic arithmetic operators for addition, subtraction, multiplication and division are then overloaded to support these vector types.

Figure 2.3 shows the example program implemented in C++ using the Intel SIMD class Iu8vec8. The SIMD classes do a good job of presenting the underlying capabilities of the architecture within the context of the C language. The code produced is also efficient, the example program in C++ runs in 4.56 seconds on the test machine, a performance of 140 million byte operations per second. However, it has to be born in mind that the C++ code is not portable to other processors. The compiler always generates MMX or SIMD instructions for the classes. If run on a 486 processor these would be illegal. The C++ code build around these classes,

```
//

#define LEN 800
#define CNT 100000
#include "ivec.h"
main()
{
Iu8vec8 v1[LEN],v2[LEN],v3[LEN];
int i,j,t;
for(i=0;i<CNT;i++)
  for (j=0;j<LEN;j++)
   v3[j]=v2[j]+v1[j];
}//
```

Figure 2.3: C++ version of the test program

```
program vecadd;
type byte=0..255;
var v1,v2,v3:array[0..6399]of byte;
    i:integer;
begin
     for i:= 1 to 100000 do v3:=v1 + v2;
end.
```

Figure 2.4: Example program in High Performance Pascal

whilst it has a higher level of expression than assembler intrinsics, is no more portable.

2.4 High level language expressions of data parallelism

High level language notations for expressing data parallelism have a long history, dating back to APL in the early '60s[8]. The key concept here is the systematic overloading of all scalar operators to work on arrays. Given some type t let us denote an array whose elements are of type t as having type $t[]$. Then if we have a binary operator $\omega : (t \otimes t) \rightarrow t$ defined on type t, in languages derived from APL we automatically have an operator $\omega : (t[] \otimes t[]) \rightarrow t[]$.

Thus if x, y are arrays of integers $k = x + y$ is the array of integers where

Table 2.1: Comparative performance of different versions of example program

Version	Time in seconds
Assembler using MMX	4.95
C++ using MMX classes	5.00
High Performance Pascal to MMX instructionset	9.88
High Performance Pascal to 486 instructionset	75.41
C	87.88

$k_i = x_i + y_i$.

The basic concept is simple, there are complications to do with the semantics of operations between arrays of different lengths and different dimensions, but Iverson provides a consistent treatment of these.

The most recent languages to be built round this model are J, an interpretive language[9], and F[12] a modernised Fortran. In principle though any language with array types can be extended in a similar way. Unlike the SIMD classes provided by the Intel C++ compiler, the APL type extensions are machine independent. They can be implemented using scalar instructions or using the SIMD model. The only difference is speed.

I have chosen this model as the basis for our experiments in compiling high level language code to SIMD instructionsets. Our objectives in these experiments were to:

1. Provide a means of expressing data parallel operations that was independent of the instructionset available.
2. To provide a mechanism for automatically generating good machine code for data parallel operations.
3. To provide a means by which the code generator can be automatically retargeted to different SIMD and non-SIMD instructionsets based on a formal description of these instructionsets.
4. To provide a toolkit that was operating system and machine independent.
5. To produce at least one demonstration compiler for at least one language with data-parallelism to evaluate the technique.
6. To produce at least two code generators one for a scalar instructionset and one for a SIMD instructionset.

For initial experiment I have chosen the base 486 and the Pentium with MMX as the two instructionsets to be targeted. An extended version of pascal with operators overloaded in the APL style is used as the source language. This is provisionally called High Performance Pascal.

The structure of the experimental system is shown in figure 2.5. It is complex. Consider first the path followed from a source file `foo.pas`, shown in the upper right of the diagram, to an executable file `foo.exe`, in the lower center of the diagram. The phases that it goes through are

i. The source file is parsed by a java class Hppc.class and results in an internal data structure, an ilcg tree, which is basically a semantic tree for the program. The classes from which this is built are from package `ilcg.tree`.

ii. The resulting ilcg tree is walked over by a class that encapsulates the semantics of the target machine's instructionset; in this case Pentium.class. The output of this process is an assembler file `foo.asm`.

iii. This is then fed through an appropriate assembler and linker, assumed to be externally provided to generate an executable.

The key components on this path are Hppc.class and Pentium.class Hppc is a simple, hand writen, recursive descent parser, nothing much interesting about this. Pentium.class has a much more complicated genesis. It is produced from a file Pentium.ilc, in a notation ILCG, which gives a

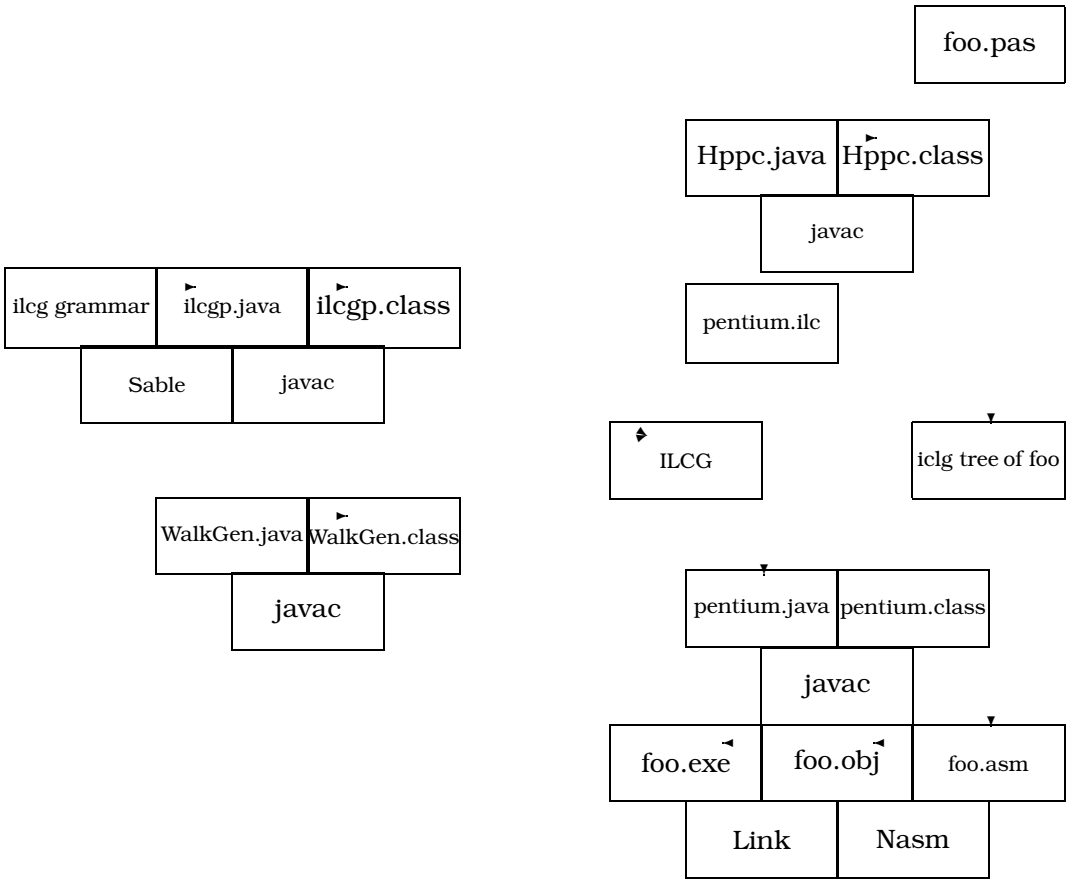

Figure 2.5: System overview

semantic description of the Pentium's instructionset. This is processed by an application ILCG which builds the source file Pentium.java.

The application ILCG is built out of two main components: a collection of classes that comprise a parser for the language ILCG, and a class which walks parse trees for ILCG programs and outputs code generators. The parser classes are automatically generated from a definition of the grammar of ILCG using the parser generator Sable[15]. The walker WalkGen walks a parse tree for an ILCG file and builds a walker for semantic trees.

In what follows I provide a description of the notation ILCG, a description of the semantic trees taken as input to the code generators and then describe the strategy used in the generated code generators to translate these trees to assembler code.

3

ILCG as a notation for describing instuctionsets

The purpose of ILCG to mediate between CPU instruction sets and high level language programs. It poth provides a representation to which compilers can translate a variety of source level programming languages and also a notation for defining the semantics of CPU instructions.

Its purpose is to act as an input to two types of programs:

1. ILCG structures produced by a HLL compiler are input to an automatically constructed code generator, working on the syntax matching principles described in [5]. This then generates equivalent sequences of assembler statements.

2. Its ascii source form acts as input to a code generator generator, that produces the code generators as described above.

It is assumed that all store allocation appart from register spillage has already been accomplished.

It is itself intended to be machine independent in that it abstracts from the instruction model of the machine on which it is being executed. However, it will assume certain things about the machine. It will assume that certain basic types are supported and that the machine is addressed at the byte level.

We further assume that all type conversions, dereferences etc have to be made absolutely explicit. Since the notation is intended primarily to be machine generated we are not particularly concerned with human readability, and thus use a prefix notation.

In what followsIwill designate terminals of the language in bold thus **octet** and nonterminal in sloping font thus *word8*.

3.1 Supported types

3.1.1 Data formats

The data in a memory can be distinguished initially in terms of the number of bits in the individually addressable chunks. The addressable chunks are assumed to be the powers of two from 3 to 7, so we thus have as allowed formats:*word8, word16, word32, word64, word128*. These are treated as non terminals in the grammar of ILCG.

Table 3.1: Range expanding coercions

INPUT	OUTPUT
byte	integer
ubyte	integer
short	integer
ushort	integer
uinteger	double
integer	double
float	double
long	double

When data is being explicitly operated on without regard to its type, we have terminals which stand for these formats: **octet, halfword, word, doubleword, quadword**.

3.1.2 Typed formats

Each of these underlying formats can contain information of different types, either signed or unsigned integers, floats etc.

We thus allow the following integer types as terminals in the language:**int8, uint8, int16, uint16, int32, uint32, int64, uint64** to stand for signed and unsigned integers of the appropriate lengths.

The integers are logically grouped into *signed* and *unsigned*. As non terminal types they are represented as *byte, short, integer, long* and *ubyte, ushort, uinteger, ulong*.

Floating point numbers are either assumed to be 32 bit or 64 bit with 32 bit numbers given the nonterminal symbols *float,double*. If we wish to specify a particular representation of floats of doubles we can use the terminals **ieee32, ieee64**.

3.1.3 Ref types

A value can be a reference to another type. Thus an integer when used as an address of a 64 bit floating point number would be a **ref ieee64** . Ref types include registers. An integer register would be a **ref int32** when holding an integer, a **ref ref int32** when holding the address of an integer etc.

3.2 Supported operations

3.2.1 Type coercions

In order to prevent the n^2 problem associated with conversions between n different types, ILCG provides a limited subset of allowed expansions, which, by composition allow any conversion requested without minimal loss of accuracy. The compressions necessarily involve a loss of accuracy.

The allowed conversions are summarised in the tables 3.1 and 3.2.

The syntax for the type conversions is C style so we have for example `(ieee64) int32` to represent a conversion of an 32 bit integer to a 64 bit real.

Table 3.2: Range contracting coercions

INPUT	OUTPUT
integer	byte
integer	ubyte
integer	short
integer	ushort
double	uinteger
double	integer
double	float
double	long

3.2.2 Arithmetic

The allowed arithmetic operations are addition, multiplication, subtraction, division and remainder with operator symboles **+,*,-,div , mod**.. The arithmetic is assumed to take place at the precisions of 32 bit integer and 64 bit reals.

The syntax is prefix with bracketing. Thus the infix operation $3 + 5 \div 7$ would be represented as **+(3 div (5 7))**.

3.2.3 Memory

Memory is explicitly represented. All accesses to memory are represented by array operations on a predefined array **mem**. Thus location 100 in memory is represented as **mem(100)**. The type of such an expression is *address*. It can be cast to a reference type of a given format. Thus we could have

(ref int32)mem(100)

3.2.4 Assignment

We have a set of storage operators corresponding to the word lengths supported. These have the form of infix operators. The size of the store being performed depends on the size of the right hand side. A valid storage statement might be

(ref octet)mem(299) :=(int8) 99

The first argument is always a reference and the second argument a value of the appropriate format.

If the left hand side is a format the right hand side must be a value of the appropriate size. If the left hand side is an explicit type rather than a format, the right hand side must have the same type.

3.2.5 Dereferencing

Dereferencing is done explicitly when a value other than a literal is required. There is a dereference operator, which converts a reference into the value that it references. A valid load expression might be:

(octet)↑ ((ref octet)mem(99))

The argument to the load operator must be a reference.

3.3 Machine description

Ilcg can be used to describe the semantics of machine instructions. A machine description typically consists of a set of register declarations followed by a set of instruction formats and a set of operations. This approach works well only with machines that have an orthogonal instruction set, ie, those that allow addressing modes and operators to be combined in an independent manner.

3.3.1 Registers

When entering machine descriptions in ilcg registers can be declared along with their type hence

register word EBX assembles['ebx'] ;
reserved register word ESP assembles['esp'];
would declare **EBX** to be of type **ref word**.

Aliasing

A register can be declared to be a sub-field of another register, hence we could write

alias register octet AL = EAX(0:7) assembles['al'];
alias register octet BL = EBX(0:7) assembles['bl'];
to indicate that **BL** occupies the bottom 8 bits of register **EBX**. In this notation bit zero is taken to be the least significant bit of a value. There are assumed to be two pregiven registers **FP, GP** that are used by compilers to point to areas of memory. These can be aliased to a particular real register.

register word EBP assembles['ebp'] ;
alias register word FP = EBP(0:31) assembles ['ebp'];
Additional registers may be reserved, indicating that the code generator must not use them to hold temporary values:

reserved register word ESP assembles['esp'];

3.3.2 Register sets

A set of registers that are used in the same way by the instructionset can be defined.

pattern reg means [$EBP|EBX|ESI|EDI|ECX|EAX|EDX|ESP$**] ;**
pattern breg means[$AL|AH|BL|BH|CL|CH|DL|DH$**];**
All registers in an register set should be of the same length.

3.3.3 Register Arrays

Some machine designs have regular arrays of registers. Rather than have these exhaustively enumerated it is convenient to have a means of providing an array of registers. This can be declared as:

register vector(8)doubleword MM assembles['MM'i] ;
This declares the symbol MMX to stand for the entire MMX register set. It implicitly defines how the register names are to be printed in the assembly language by defining an indexing variable i that is used in the assembly language definition.

We also need a syntax for explicitly identifying individual registers in the set. This is done by using the dyadic subscript operator:

subscript(MM,2)
which would be of type **ref doubleword**.

3.3.4 Register Stacks

Whilst some machines have registers organised as an array, another class of machines, those oriented around postfix instructionsets, have register stacks.

The ilcg syntax allows register stacks to be declared:
register stack (8)ieee64 FP assembles[' '] ;
Two access operations are supported on stacks:

PUSH is a void dyadic operator taking a stack of type ref t as first argument and a value of type t as the second argument. Thus we might have:
PUSH(FP,↑mem(20))

POP is a monadic operator returning t on stacks of type t. So we might have
mem(20):=POP(FP)

3.3.5 Instruction formats

An instruction format is an abstraction over a class of concrete instructions. It abstracts over particular operations and types thereof whilst specifying how arguments can be combined.

instruction pattern
RR(operator op, anyreg r1, anyreg r2, int t)
means[r1:=(t) op(↑((ref t) r1),↑((ref t) r2))]
assembles[op ' ' r1 ',' r2];

In the above example, we specify a register to register instruction format that uses the first register as a source and a destination whilst the second register is only a destination. The result is returned in register r1.

We might however wish to have a more powerful abstraction, which was capable of taking more abstract apecifications for its arguments. For example, many machines allow arguments to instructions to be addressing modes that can be either registers or memory references. For us to be able to specify this in an instruction format we need to be able to provide grammer non-terminals as arguments to the instruction formats.

For example we might want to be able to say
instruction pattern
RRM(operator op, reg r1, maddrmode rm, int t)
means [r1:=(t) op(↑((ref t)r1),↑((ref t) rm))]
assembles[op ' ' r1 ',' rm] ;

This implies that addrmode and reg must be non terminals. Since the non terminals required by different machines will vary, there must be a means of declaring such non-terminals in ilcg.

An example would be:
pattern regindirf(reg r)
means[↑(r)] assembles[r];
pattern baseplusoffsetf(reg r, signed s)
means[+(↑(r) ,const s)] assembles[r '+' s];
pattern addrform means[baseplusoffsetf| regindirf];

pattern maddrmode(addrform f)
means[mem(f)] assembles['[' f ']'];

This gives us a way of including non terminals as parameters to patterns.

4

Implementation strategy

4.1 The package ilcg.tree

The package ilcg.tree provides java classes to support the code generation process. These fall into two broad categories, classes which implement the interface `ilcg.tree.Node`, and classes which are descended from `ilcg.tree.Walker`.

4.1.1 Nodes

The classes implementing the `Node` interface are used to build ILCG trees. These classes are in 1 to 1 correspondance with the constructs of the ILCG grammar. Thus for any expression defining the semantics of a machine instruction in ILCG there is a corresponding tree structure which matches it.

These trees can either be directly constructed by a Java program, or can be loaded from a `.tree` file, which is basically a postfix flattened binary encoding of the tree. Such `.tree` files can be generated by front ends written in languages other than Java. Each class implementing the interface `Node` must have a method

`toBinary(java.io.DataOutputStream out)`

which saves a node in postfix binary notation on the output stream. A utility class `ilcg.tree.Loader` can be used to load postfix binary trees stored in files into memory.

4.1.2 Walkers

The other classes in the package are tree walkers which travese ilcg trees emitting assembler code as they do so.

The base class `ilcg.tree.Walker` provides strategic methods to carry out

- register assignment and spilling,

Implementing Classes for Node:
`Assign, Block, Cast, CharLit, Dyad, For, Format, Goto, If, Integer, Label, Location, Memref, Monad, Op, Real, Register, RegisterStack, Statement, Type`

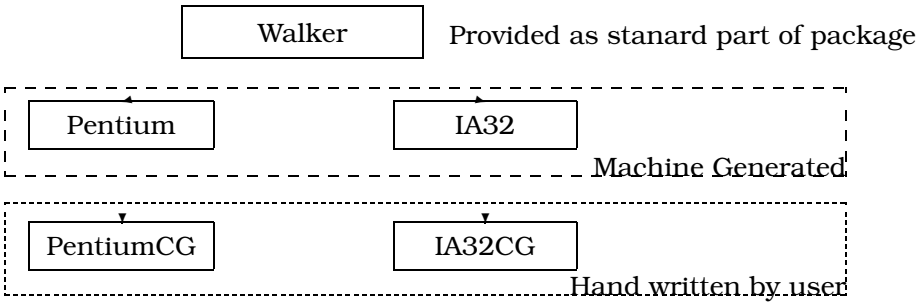

Figure 4.1: How machine generated and hand written walkers descend from `ilcg.tree.Walker`

- constant expression evaluation,
- transformation of high level to low level control structures
- pre-evaluation of repeated sub-expressions within statements
- transformation of array arithmetic to sematically equivalent for loops

It also includes declarations of the variables used in the code generation process. Each time `Walker` encounters a statement it invokes the `match` method. This is an abstract method that must be provided in subsidiary classes, which are machine generated by the program ILCG from CPU specification files. They implement the `match` method in a form appropriate for that CPU.

Finally, the machine generated walker classes can be further specialised by hand written classes which override the behaviour of the standard tree walker. Typically these specialisations will be to handle things like procedure call mechanisms which may be specific to a given processor and a given language.

4.1.3 Generation of array arithmetic code

The algorithm for effectively using SIMD instructions for array arithmetic depends upon the decoration of ILCG trees with appropriate type information. The parsermust produce a syntax tree in which casts are applied to memory referencing subtrees to specify the types that they point to, as illustrated in figure 4.2. The statement is passed to the instruction matcher which fails, as there is no built in instruction capble of adding operands of type `uint8 vector(6400)`.

4.2 The ILCG code-generator generator

The code-generator generator ILCG is invoked by the command line

`java ilcg.ILCG` *src*`.ilc` *dest*`.java` *classname*

It parses the `.ilc` file and for each pattern p in the file generates a recognising method. These methods take a single parameter n of type ilcg.tree.Node and return true if n matches p.

Figures 4.3 and 4.4 illustrate an instruction pattern and the recogniser method that is generated for it. It can be seen that the recogniser for the assignment statement invokes subsidiary recognisers for the left and right hand side of the assignment.

```
program vecadd;
type byte=0..255;
var v1,v2,v3:array[0..6399]of byte;
 i:integer;
begin
  v3:=v1 + v2;
end.
```

```
(ref uint8 vector ( 6400 ))mem(+(((ref int32)ebp),-19200)):=
 +(^((ref uint8 vector ( 6400 ))mem(+(^((ref int32)ebp),-6400))),
  ^((ref uint8 vector ( 6400 ))mem(+(((ref int32)ebp),-12800)))
 )
```

Figure 4.2: Translation of an array expression into ILCG

```
instruction pat-
tern RMR(operator op,addrmode rm,immedreg r1,int t)
    means[ (ref t) rm :=op( ^((ref t) rm),( t) r1)]
    assembles[op ’  ’   t ’ ’ rm ’,’ r1];
```

Figure 4.3: The specification of the register to memory form of x86 instruction

```
/** recog-
nises ( ref t ) rm := op ( ^ ( ( ref t ) rm ) , ( t ) r1 )  */
public boolean pAAssign1126(Node n){
       dlog("( ref t ) rm := op ( ^ ( ( ref t ) rm ) , ( t ) r1 ) ",n);
       if(!(n instanceof Assign))return false;
       Assign a=(Assign)n;
       enterRhs();
         boolean rhsok= pADyadicValue1132(a.src);
       leaveRhs();
       if(!rhsok)return false;
       enterLhs();
         boolean lhsok=pARefcastRefval1127(a.dest);
       leaveLhs();
       return lhsok;
}
```

Figure 4.4: The recogniser method for the meaning part of the instruction pattern shown in figure 4.3

Where a pattern is defined as a set of alternatives, the alternatives are tried in the order in which they are listed in the `.ilc` file. The matching functions for alternative patterns are implemented as memofns. That is to say, each such function has a hash table in which it stores the trees that it has successfully matched and which alternative they matched. This hash table is consulted whenever a new tree is encountered.

5

The grammer for processor definition

```
/*
```

5.1 ILCG grammar

This is a definition of the grammer of ILCG using the Sable grammar specification lanaguage. It is input to Sable to generate a parser for machine descriptions in ilcg

```
*/



Package ilcg;
/*
```

5.1.1 Helpers

Helpers are regular expressions macros used in the definition of terminal symbols of the grammar.

```
*/
Helpers
  letter = [['A'..'Z']+['a'..'z']];
  digit = ['0'..'9'];
  alphanum = [letter+['0'..'9']];
  cr = 13;
  lf = 10;
  tab = 9;
    digit_sequence = digit+;
    fractional_constant = digit_sequence? '.' digit_sequence | digit_sequence '.';
    sign = '+' | '-';
    exponent_part = ('e' | 'E') sign? digit_sequence;
    floating_suffix = 'f' | 'F' | 'l' | 'L';
  eol = cr lf | cr | lf;        // This takes care of different platforms
  not_cr_lf = [[32..127] - [cr + lf]];
  exponent = ('e'|'E');
  quote = ''';
  all =[0..127];
  schar = [all-'''];
```

```
  not_star = [all - '*'];
  not_star_slash = [not_star - '/'];
/*
```

5.1.2 Tokens

The tokens section defines the terminal symbols of the grammar.

```
*/
Tokens
  floating_constant = fractional_constant exponent_part? floating_suffix? |
        digit_sequence exponent_part floating_suffix?;
 /*
```

terminals specifying data formats

```
*/
 void ='void';
 octet = 'octet'; int8 = 'int8'; uint8 = 'uint8';
 halfword = 'halfword'; int16 = 'int16' ; uint16 = 'uint16' ;
 word = 'word'; int32 = 'int32' ;
 uint32 = 'uint32' ; ieee32 = 'ieee32';
 doubleword = 'doubleword'; int64 = 'int64' ;
 uint64 = 'uint64'; ieee64 = 'ieee64';
 quadword = 'quadword';
/*
```

terminals describing reserved words

```
*/
 function= 'function';
 flag = 'flag';
 location = 'loc';
 procedure='instruction';
 returns ='returns';
 label = 'label';
 goto='goto';
 for ='for';
 to='to';
 step='step';
 do ='do';
 ref='ref';
 const='const';
 reg= 'register';
 operation = 'operation';
 alias = 'alias';
 instruction = 'instruction';
 address = 'address';
 vector ='vector';
 stack = 'stack';
 sideeffect='sideeffect';
 if ='if';
 reserved='reserved';
 precondition ='precondition';

 instructionset='instructionset';
/*
```

terminals for describing new patterns

```
 */
 pattern = 'pattern';
 means = 'means';
 assembles = 'assembles';

/*
```

terminals specifying operators

```
*/
 colon = ':';
 semicolon= ';';
 comma = ',';
 dot = '.' ;
 bra ='(';

 ket =')';
 plus = '+';
 satplus = '+:';
 satminus = '-:';
 map='->';
 equals = '=';
 le = '<=';
 ge='>=';
 ne='<>';
 lt='<';
 gt='>';
 minus = '-';
 times = '*';
 exponentiate = '**';
 divide = 'div';
 and = 'AND';
 or = 'OR' ;
 xor = 'XOR';
 not = 'NOT';
 sin='SIN';
 cos='COS';
 abs='ABS';
 tan='TAN';
 remainder = 'MOD';
 store = ':=';
 deref = '^';
 push ='PUSH';
 pop ='POP';
 call='APPLY';
 full='FULL';
 empty='EMPTY';
 subscript='SUBSCRIPT';
 intlit = digit+;

 vbar = '|';
 sket=']';
 sbra='[';
 end='end';
 typetoken='type';
 mem='mem';
 string = quote schar+ quote;
/*
```

identifiers come after reserved words in the grammar

```
*/
 identifier = letter alphanum*;
 blank = (' '|cr|lf|tab)+;
 comment = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';

Ignored Tokens
blank,comment;
/*
```

5.1.3 Non terminal symbols

```
*/
Productions
 program = statementlist instructionlist;
 instructionlist =instructionset sbra alternatives sket;
/*
```

non terminals specifying data formats

```
*/
 format = {octet} octet| {halfword} halfword |
          {word} word | {doubleword} doubleword |
          {quadword} quadword|{tformat}tformat;


/*
```

non terminals corresponding to type descriptions

```
*/
 reference =   ref type ;
 array = vector bra number ket;
 aggregate={stack} stack bra number ket |{vector}array |{non};

 type= {format} format |  {typeid}  typeid|{array}type array|
 {cartesian}sbra type cartesian* sket|
{map}bra [arg]:type map [result]:type ket;
 cartesian =  comma type;

 tformat = {signed} signed|{unsigned}unsigned|{ieee32}ieee32|
           {ieee63}ieee64;
 signed = int32 | {int8} int8 | {int16} int16 | {int64} int64;
 unsigned = uint32 | {uint8} uint8 | {uint16} uint16 |
            {uint64} uint64;


/*
```

non terminals corresponding to typed values

```
*/
 value = {refval}refval |{rhs}rhs|{void}void|{cartval}cartval|
{dyadic} dyadic bra [left]:value comma [right]:value ket|
{monadic}monadic bra value ket;
/*
```

value corresponding to a cartesian product type e.g. record initialisers

```
*/
 cartval=sbra value carttail* sket;
 carttail = comma value;
/*
```

```
conditions used in defining control structures

*/
condition={dyadic} dyadic bra[left]:condition comma[right]:condition ket|
{monadic}monadic bra condition ket |
{id}identifier|
{number}number;
 rhs= {number}number| {cast}bra type ket value|{const}const identi-
fier |
         {deref}deref  bra  refval ket;

 refval = loc| {refcast} bra reference ket loc;
 loc = {id}identifier|{memory}mem bra value ket ;


/*predeclaredregister = {fp}fp|{gp}gp;*/
 number =  {reallit} optionalsign reallit|
           {integer} optionalsign intlit;
 optionalsign = |{plus}plus|{minus}minus;
 reallit= floating_constant;
/*

operators

*/
dyadic  = {plus} plus| {minus} minus |
{identifier} identifier|{exp}exponentiate|
{times} times | {divide}  divide|
{lt}lt|{gt}gt|{call}call|
{le}le|{ge}ge|{eq}equals|{ne}ne|{push}push|{subscript}subscript|
{satplus}satplus|{satminus}satminus|
  {remainder} remainder|{or}or|{and}and|{xor}xor;
monadic ={not}not|{full}full|{empty}empty|{pop}pop|
{sin}sin|{cos}cos|{tan}tan|{abs}abs ;
/*

register declaration

*/
registerdecl =  reservation reg aggregate format identifier
assembles sbra string sket  ;
reservation = {reserved}reserved|{unreserved};
aliasdecl =
          alias reg aggregate format [child]:identifier equals
          [parent]:identifier bra [lowbit]:intlit colon [highbit]:intlit ket
          assembles sbra string sket;

opdecl = operation identifier means operator assembles sbra string sket;
operator = {plus}plus|{minus}minus|{times}times|{lt}lt|{gt}gt|
{le}le|{ge}ge|{eq}equals|{ne}ne|{divide} divide|
{remainder}remainder|{or}or|{and}and|{xor}xor;

/*

pattern declarations

*/
assign =  refval store value ;
meaning = {value}value|{assign}assign|
  {goto}goto value|
  {if}if bra value ket meaning|
```

```
{for} for refval store [start]:value
      to [stop]:value step [increment]:value do meaning|
   {loc}location value;
patterndecl = pattern identifier paramlist means sbra meaning sket
assemblesto
sideeffects
precond
 |{alternatives} pattern identifier means sbra alternatives sket;

paramlist = bra param paramtail* ket|{nullparam}bra ket;
param = typeid identifier|
{typeparam} typetoken identifier|
{label}label identifier;
typeid = identifier;
paramtail = comma param;
alternatives = type alts*;
alts = vbar type;
precond = precondition sbra condition sket|{unconditional};
asideeffect=   sideeffect  returnval;
sideeffects = asideeffect*;
assemblesto=assembles sbra assemblypattern sket;
assemblypattern = assemblertoken*;
assemblertoken = {string} string | {identifier} identifier;
returnval = returns identifier;
/*
```

statements

```
*/
statement = {aliasdecl} aliasdecl|
            {registerdecl} registerdecl |
{addressmode} address patterndecl|
{instructionformat}procedure patterndecl|
{opdecl}opdecl|
{flag} flag identifier equals intlit|
{typerename}typetoken format equals identifier|
{patterndecl} patterndecl;
statementlist = statement semicolon statements*;
statements =  statement  semicolon;


//
```

6

An example machine specification

```
/*
```

6.1 Pentium machine description

```
*/ /*
```

Basic ia32 processor description int ilcg copyright(c) Paul Cockshott, University of Glasgow Feb 2000

6.1.1 Declare types to correspond to internal ilcg types

```
*/

type word=DWORD;
type uint32=DWORD;
type int32=DWORD;
type ieee64=QWORD;
type octet=BYTE;
type int16=WORD;
type int8=BYTE;
type octet=BYTE;
type uint8=BYTE;
type halfword=WORD;

/*
```

6.1.2 compiler configuration flags

```
*/
flag realLitSupported = 0;
/*
```

6.1.3 Register declarations

```
*/
register word EAX  assembles['eax'] ;
register word ECX  assembles['ecx'] ;
register word EBX  assembles['ebx'] ;
reserved  register word EBP  assembles['ebp'] ;
```

```
alias register word FP = EBP(0:31)  assembles ['ebp'];
reserved register word ESP  assembles['esp'];
alias register word SP = ESP(0:31)  assembles ['esp'];
register word ESI  assembles['esi'] ;
register word EDI  assembles['edi'] ;
register word EDX  assembles['edx'];


 alias register octet AL = EAX(0:7)  assembles['al'];
 alias register octet BL = EBX(0:7)  assembles['bl'];
 alias register octet CL = ECX(0:7)  assembles['cl'];
 alias register octet DL = EDX(0:7)  assembles['dl'];

 alias register octet AH = EAX(8:15)  assembles['ah'];
 alias register octet BH = EBX(8:15)  assembles['bh'];
 alias register octet CH = ECX(8:15)  assembles['ch'];
 alias register octet DH = EDX(8:15)  assembles['dh'];


pattern reg  means [ EBP| EBX|ESI|EDI|ECX |EAX|EDX|ESP]  ;
pattern areg(reg r)  means[(ref int32)r] assembles[ r ];
pattern breg  means[ AL|AH|BL|BH|CL|CH|DL|DH];
pattern abreg(breg b)  means[(ref int8)b] assembles[b];
pattern anyreg  means[ breg|reg];
pattern acc  means[EAX];
pattern creg  means [ECX];
pattern sreg  means [ESI];
pattern dreg  means [EDI];
pattern modreg  means [EDX];
pattern bacc  means [AL];
/*
```

6.1.4 Operator definition

This section defines operations that can be used to parameterise functions.

```
*/
operation add  means  +   assembles [ 'add'];
/* */operation and  means  AND   assembles[ 'and'];
operation or  means OR  assembles['or'];
operation xor  means XOR  assembles['xor'];/* */
operation sub  means  -   assembles [ 'sub'];
operation mul  means *  assembles ['mul'];
operation imul  means *  assembles ['imul '];
operation lt  means <  assembles ['l'];
operation gt  means >  assembles ['g'];
operation eq  means =  assembles ['z'];
operation le  means <=  assembles ['le'];
operation ge  means >=  assembles ['ge'];
operation ne  means <>  assembles ['nz'];

pattern condition  means[ne|ge|le|eq|gt|lt];
pattern operator  means[add | sub|imul|and|or|xor];
pattern commuteop  means[add|and|or|xor];
/*
```

6.1.5 Data formats

Here we define ilcg symbols for the types that can be used as part of instructions.

```
*/
pattern unsigned  means[uint32|uint8|uint16];
pattern signed  means[int32 |   int8 |   int16 ];
pattern int  means[signed|unsigned];
pattern double  means[ieee64] ;
pattern float   means[ieee32];
pattern real  means [ieee64|float];

pattern dataformat  means[octet|word];
pattern longint  means [int32|uint32];
pattern two(type t) means[2]  assembles['2'];
pattern four(type t) means[4]  assembles['4'];
pattern eight(type t) means[8]  assembles['8'];

pattern scale  means[two|four|eight];

/**
```

Define the address forms used in lea instructions these differ from the address forms used in other instructions as the semantics includes no memory reference. Also of course register and immediate modes are not present.

```
 */
pattern regindirf(reg r)
 means[^(r) ]
 assembles[ r ];
pattern baseplusoffsetf(reg r, signed s)
 means[+(  ^(r) ,const s)]
 assembles[  r '+' s ];
pattern scaledIndexPlusOffsetf(reg r1, scale s, signed offs)
 means[+(*(^(r1),s),const offs)]
 assembles[r1 '*' s '+' offs];
address pattern basePlusScaledIndexf(reg r1,reg r2,scale s)
 means[+(^(r1),*(^(r2),s))]
         assembles[  r1 '+' r2 '*'  s   ];
address pattern basePlusScaledIndexPlusOffsetf(reg r1,reg r2,scale s,signed off)
 means[+(+(^(r1),const off),*(^(r2),s))]
         assembles[  r1 '+' r2 '*'  s '+'off  ];
address pattern basePlusIndexPlusOffsetf(reg r1,reg r2,signed off)
 means[+(+(^(r2),const off),^(r1))]
         assembles[  r1 '+' r2 ' +'off  ];

address pattern basePlusIndexf(reg r1,reg r2)
 means [+(^(r1),^(r2))]
 assembles[  r1 '+' r2   ];
pattern directf(unsigned s)
 means[const s]
 assembles[  s  ];
pattern immediate(signed s)
 means[const s]
 assembles[  s  ];
pattern labelf(label l)
 means [l]
 assembles[l];

/*
```

6.1.6 Choice of effective address

This contains the useful formats for the load effective address instruction. The pattern regindirf is excluded here as it adds nothing we do not have already from mov instructions.

```
*/
pattern eaform  means[directf  |
labelf|
        scaledIndexPlusOffsetf|
basePlusScaledIndexPlusOffsetf|
basePlusIndexPlusOffsetf|
basePlusScaledIndexf|
baseplusoffsetf |
basePlusIndexf];
/*
```

6.1.7 Formats for all memory addresses

```
*/
pattern addrform  means[eaform| regindirf];


/**

define the address patterns used in other instructions


*/



pattern maddrmode(addrform f)  means[mem(f) ]  assembles[ '[' f ']' ];
pattern addrmode  means[maddrmode|anyreg];
pattern regderef(anyreg r) means[^(r)] assembles[r];
pattern derefmaddrmode (maddrmode a,type t) means[(t)^(a)]  assembles [t ' ' a];
pattern immedmem  means[immediate|derefmaddrmode];
pattern immedaddrmode  means[immediate|derefmaddrmode|regderef];
pattern immedreg  means[immediate|regderef];
/*
```

6.1.8 Instruction patterns for the 386

```
*/

instruction pattern STORE(addrmode rm, immedreg r1, type t)
         means[  (ref t) rm:=(t)r1 ]
 assembles['mov 't rm ',' r1];

instruction  pattern NULMOV(reg r3)
 means[r3:=^(r3)]
         assembles[';nulmov ' r3  r3];

instruction  pattern INC(addrmode rm,type t)
 means[(ref t)rm:=   + (^((ref t)rm),1)]
 assembles['inc ' t ' ' rm];

instruction  pattern DEC(addrmode rm,type t)
 means[(ref t)rm:=   + (^((ref t)rm),-1)]
```

```
 assembles['dec ' t ' ' rm];

instruction  pattern LOAD(immedmem  rm, anyreg r1, type t)
         means[  r1:= ( t)rm ]
 assembles['mov ' r1 ',' ' ' rm];
instruction  pattern MOVE(immedaddrmode rm, anyreg r1, type t)
         means[  r1:= ( t)rm ]
 assembles['mov ' r1 ',' ' ' rm];

instruction pattern MOVZXB(reg r1, addrmode rm)
 means[  r1:=(int32)^((ref octet)rm)]
 assembles['movzx ' r1 ', BYTE 'rm];

instruction pattern MOVZXW(reg r1, addrmode rm)
 means[  r1:=(int32)^((ref halfword)rm)]
 assembles['movzx ' r1 ', WORD 'rm];

instruction pattern LEA(reg r1, eaform ea)
 means [r1:=ea]
         assembles ['lea ' r1 ',[' ea ']' ];

instruction pattern Negate(reg r1,type t)
 means[r1:= -((t)0,^((ref t)r1))]
 assembles ['neg ' r1];

instruction pattern NEG(reg r1)
 means[r1:= *(^(r1),-1)]
 assembles['neg 'r1];

instruction  pattern RMR( operator op,addrmode rm,immedreg r1,int t)
         means[ (ref t) rm :=op( ^((ref t) rm),( t) r1)]
      assembles[op '  '   t ' ' rm ',' r1];
/* commuting version of the rmr form */
instruction  pattern ComRMR( commuteop op,addrmode rm,immedreg r1,int t)
         means[ (ref t) rm :=op( (t)r1,^((ref t) rm))]
      assembles[op '  '   t ' ' rm ',' r1];

instruction  pattern RRM(operator op, anyreg r1, immedaddrmode rm, int t)
         means [r1:=(t) op( ^((ref t)r1),(t) rm)]
         assembles[op ' ' r1 ',' rm ] ;

instruction pattern PLANT(label l)
 means[l]
 assembles[l ':'];

instruction pattern PLANTRCONST( double r)
 means[loc r]
 assembles[ 'SECTION .data\n dq ' r '\n SECTION .text'];

instruction pattern GOTO(label l)
 means[goto l]
 assembles['jmp ' l];
/*
```

6.1.9 Multi Instruction Templates

These are sequences of instructions that are designed to look like single instructions

```
 */

pattern dacc  means[DH|DL];
instruction pattern ADDUSB(bacc r1,dacc r2 )
 means[ r1 := +:(^(r1),^(r2))]
 assembles[' push ebx\n push ecx\n mov bl,al\n mov cl,' r2
           '\n xor ch,ch\n xor bh,bh\n add bx,cx\n cmp cx,255'
           '\n jl $+5\n mov cl,255\n nop\n mov al,cl\n pop ecx\n pop ebx'];
instruction pattern IDIV(reg r1,reg r2)
         means[r1:= div(^(r1),^( r2)) ]
 assembles['push edx\nxchg eax,' r1
           '\ncdq\ndiv ' r2
           '\npop edx\nxchg eax,' r1];


instruction pattern MOVSD(eaform i,creg cx, sreg si, dreg di,type t)
        means[for (ref t)mem(i) :=0 to ^(cx) step 1 do
        mem(+(^(di),*(^(mem(i)),4))):= (word)^(mem(+(^(si),*(^(mem(i)),4))))]
        assembles['cld \n mov ' t'['i'],ecx\n rep\n movsd'];

instruction pattern
        MOVSB(eaform i,creg cx, sreg si, dreg di,type t)
   means[for (ref t)mem(i):=0 to ^(cx) step 1 do
   mem(+(^(di),^(mem(i)))):=(octet)^(mem(+(^(si),^(mem(i)))))]
 assembles['cld \n mov ' t'['i'],ecx\n rep\n movsb'];

instruction pattern
         SETCC(addrmode d,reg r1,immedaddrmode r2,condition c)
 means[(ref octet)d:=(octet) c(^(r1),r2)]
 assembles['cmp ' r1 ',' r2 '\n set' c ' BYTE ' d];

instruction pattern
         IFGOTO(label l,reg r1,immedaddrmode r2,condition c)
 means[if(c(^(r1),^(r2)))goto l]
 assembles['cmp ' r1 ',' r2 '\n j' c ' NEAR ' l];

instruction pattern
         IFLITGOTO(label l,addrform a,signed r2,condition c,type t)
 means[if(c( ^((ref t) mem(a)),const r2))goto l]
 assembles['cmp ' t '[' a ']'   ',' r2 '\n j' c ' NEAR ' l];

instruction pattern CALL(label l)
 means[APPLY( l,void)]
 assembles['call ' l];

instruction pattern IMOD( modreg r1,reg r2)
         means[r1:= MOD(^(r1),^( r2)) ]
 assembles['push eax\nmov eax,' r1 '\ncdq\ndiv   ' r2 '\npop eax'];


/*

*/
   /*
```

6.1.10 Intel fpu instructions

```
*/
```

```
register stack(8)ieee64 ST  assembles[ 'ST'];

operation fdiv  means div  assembles['div'];
pattern foperator  means[add | sub|mul|fdiv];


instruction pattern FSTR(maddrmode a,real t)
 means[(ref t) a:= (t)POP(ST)]
 assembles['fstp  't a];
pattern fint  means[int16|int32|int64];
instruction pattern FIOP(maddrmode a,fint t,foperator op)
 means[PUSH(ST,op(POP(ST),(ieee64)^((ref t)a)))]
 assembles['fi'op' 't a];
instruction pattern FIDIVR(maddrmode a,fint t)
 means[PUSH(ST,div((ieee64)^((ref t)a),POP(ST)))]
 assembles['fidivr 't a];
instruction pattern FLDONE()
 means[PUSH(ST,1.0)]
 assembles['fld1'];
instruction pattern FLDZ()
 means[PUSH(ST,0.0)]
 assembles['fldz'];
instruction pattern FLDPI()
 means[PUSH(ST,3.14159265358979)]
 assembles['fldpi'];
instruction pattern FILD(maddrmode a,fint t)
 means[PUSH(ST,(ieee64)^((ref t)a))]
 assembles['fild  't a];
instruction pattern FISTP(maddrmode a,fint t)
 means[(ref t)a:=(t)POP(ST)]
 assembles ['fistp 't a];
instruction pattern FLD(maddrmode a, real t)
 means[PUSH(ST,(ieee64)^((ref t) a))]
 assembles['fld ' t a];
instruction pattern FNEG(real t)
 means[PUSH(ST, *((t)POP(ST),-1.0))]
 assembles['fchs '];
instruction pattern FOP(maddrmode a,foperator op)
 means[PUSH(ST,op(POP(ST),^((ref ieee64)a)  )) ]
 assembles['f'op '  qword  ' a];
instruction pattern ReversePolishFOP(foperator op)
 means[PUSH(ST,op(POP(ST),POP(ST)))]
 assembles['f'op'p' ' st1'];
instruction pattern FRNDINT()
 means[PUSH(ST,(ieee64)(int64)POP(ST))]
 assembles['frndint'];
instruction pattern FCOS(real t)
 means[PUSH(ST,COS((t)POP(ST)))]
 assembles['fcos'];
instruction pattern FSIN(real t)
 means[PUSH(ST,SIN((t)POP(ST)))]
 assembles['fsin'];
instruction pattern FTAN(real t)
 means[PUSH(ST,TAN((t)POP(ST)))]
 assembles['fptan'];
instruction pattern FABS(real t)
 means[PUSH(ST,ABS((t)POP(ST)))]
 assembles['fabs'];
```

```
instruction pattern FXM(real t)
 means[PUSH(ST,-((t)**(2,(t)POP(ST)),1))]
 assembles['f2xm1'];
instruction pattern FSCALE()
 means[PUSH(ST,*(POP(ST),**(2,(int32)POP(ST))))]
 assembles['fscale'];



/*

*/
   /*
```

6.1.11 MMX registers and instructions

Registers

```
*/

register doubleword MM0  assembles[ 'MM0'];
register doubleword MM1  assembles[ 'MM1'];
register doubleword MM2  assembles[ 'MM2'];
register doubleword MM3  assembles[ 'MM3'];
register doubleword MM4  assembles[ 'MM4'];
register doubleword MM5  assembles[ 'MM5'];
register doubleword MM6  assembles[ 'MM6'];
register doubleword MM7  assembles[ 'MM7'];

pattern mreg  means [MM1|MM3|MM4|MM5|MM7|MM6|MM2|MM0];



/*
```

Addressing modes

```
*/
pattern mrmaddrmode  means[maddrmode|mreg];

/*
```

MMX instructions

```
*/
pattern byteint  means[int8|uint8];
pattern wordint  means[int16|uint16];
instruction pattern PMULLW(mreg m, mrmaddrmode ma)
 means[(ref int16 vector(4))m :=
 (int16 vector(4))*((int16 vector(4))^(m),(int16 vector(4))^(ma))]
 assembles['pmullw ' m ',' ma];
instruction pattern PADDD(mreg m, mrmaddrmode ma)
 means[(ref int32 vector(2))m :=
 (int32 vector(2))+((int32 vector(2))^(m),(int32 vector(2))^(ma))]
 assembles ['paddd 'm ',' ma];
instruction pattern PADDW(mreg m, mrmaddrmode ma)
 means[(ref int16 vector(4))m :=
```

```
 (int16 vector(4))+((int16 vector(4))^(m),(int16 vector(4))^(ma))]
 assembles [’paddw ’m ’,’ ma];
instruction pattern PADDB(mreg m, mrmaddrmode ma)
 means[(ref int8 vector(8))m :=
 (int8 vector(8))+((int8 vector(8))^(m),(int8 vector(8))^(ma))]
 assembles [’paddb ’m ’,’ ma];
instruction pattern PADDUB(mreg m, mrmaddrmode ma,byteint t)
 means[(ref uint8 vector(8))m :=
 (uint8 vector(8))+((uint8 vector(8))^(m),(uint8 vector(8))^(ma))]
 assembles [’paddb ’m ’,’ ma];
instruction pattern PADDUSB(mreg m, mrmaddrmode ma)
 means[(ref int8 vector(8))m :=
 (int8 vector(8))+:((int8 vector(8))^(m),(int8 vector(8))^(ma))]
 assembles [’paddusb ’m ’,’ ma];
instruction pattern PSUBUSB(mreg m, mrmaddrmode ma)
 means[(ref int8 vector(8))m :=
 (int8 vector(8))-:((int8 vector(8))^(m),(int8 vector(8))^(ma))]
 assembles [’psubusb ’m ’,’ ma];
instruction pattern PADDUSW(mreg m, mrmaddrmode ma)
     means[(ref int16 vector(4))m :=
     (int16 vector(4))+:((int16 vector(4))^(m),(int16 vector(4))^(ma))]
     assembles [’paddusw ’m ’,’ ma];
instruction pattern PSUBUSW(mreg m, mrmaddrmode ma)
     means[(ref int16 vector(4))m :=
     (int16 vector(4))-:((int16 vector(4))^(m),(int16 vector(4))^(ma))]
     assembles [’psubusw ’m ’,’ ma];
instruction pattern PSUBD(mreg m, mrmaddrmode ma)
 means[(ref int32 vector(2))m :=
 (int32 vector(2))-((int32 vector(2))^(m),(int32 vector(2))^(ma))]
 assembles [’psubd ’m ’,’ ma];
instruction pattern PSUBW(mreg m, mrmaddrmode ma)
 means[(ref int16 vector(4))m :=
 (int16 vector(4))-((int16 vector(4))^(m),(int16 vector(4))^(ma))]
 assembles [’psubw ’m ’,’ ma];
instruction pattern PSUBB(mreg m, mrmaddrmode ma)
 means[(ref int8 vector(8))m :=
 (int8 vector(8))-((int8 vector(8))^(m),(int8 vector(8))^(ma))]
 assembles [’psubb ’m ’,’ ma];
instruction pattern PSUBUB(mreg m, mrmaddrmode ma,byteint t)
 means[(ref uint8 vector(8))m :=
 (uint8 vector(8))-((uint8 vector(8))^(m),(uint8 vector(8))^(ma))]
 assembles [’psubb ’m ’,’ ma];
instruction pattern PAND(mreg m, mrmaddrmode ma)
 means[m := AND(^(m),^(ma))]
 assembles [’pand ’m ’,’ ma];
instruction pattern POR(mreg m, mrmaddrmode ma)
 means[m := OR(^(m),^(ma))]
 assembles [’por ’m ’,’ ma];
instruction pattern MOVDS(addrmode rm, mreg m)
 means[(ref int32)rm:= (int32)^(m)]
 assembles[’movd ’rm ’,’m];
instruction pattern MOVDL(addrmode rm, mreg m)
 means[m := (int64)^((ref int32)rm)]
 assembles[’movd ’m ’,’rm];
instruction pattern MOVQS(maddrmode rm, mreg m,type t)
 means[(ref t)rm:= (t)^(m)]
 assembles[’movq ’rm ’,’m];
instruction pattern MOVQL(addrmode rm, mreg m)
```

```
 means[m := (doubleword)^(rm)]
 assembles['movq ' m ',' rm];
```

```
instruction pattern CMPXCHG(acc a,reg src,addrmode dest)
 means[if(=(^(a),^(dest)))dest:=^(src)]
  assembles['cmpxchg dword' dest ','  src ];

/* \begin{verbatim}*/
instructionset [CMPXCHG|MOVSB|MOVSD|INC|DEC|NEG|Negate|
LEA|LOAD|MOVZXB|MOVZXW|RRM|ADDUSB|
                SETCC|IDIV|IMOD|RMR|ComRMR|STORE|MOVE|
                PLANT|IFLITGOTO |IFGOTO|GOTO|CALL|
                PLANTRCONST|PADDD|PADDW|PADDB|PSUBD|PSUBW|
                PSUBB|PSUBUB|PADDUSB|PADDUB|
                PSUBUSB|PADDUSW|PSUBUSW|POR|PAND|PMULLW
                |MOVDS|MOVDL|MOVQS|MOVQL ]

/*


*/
```

Bibliography

[1] Advanced Micro Devices, 3DNow! Technology Manual, 1999.

[2] Aho, A.V., Ganapathi, M, TJiang S.W.K., Code Generation Using Tree Matching and Dynamic Programming, ACM Trans, Programming Languages and Systems 11, no.4, 1989, pp.491-516.

[3] Cattell R. G. G., Automatic derivation of code generators from machine descriptions, ACM Transactions on Programming Languages and Systems, 2(2), pp. 173-190, April 1980.

[4] Cheong, G., and Lam, M., An Optimizer for Multimedia Instruction Sets, 2nd SUIF Workshop, Stanford University, August 1997.

[5] Susan L. Graham, Table Driven Code Generation, IEEE Computer, Vol 13, No. 8, August 1980, pp 25..37.

[6] Intel, Intel Architecture Software Developers Manual Volumes 1 and 2, 1999.

[7] Intel, Willamette Processor Software Developer's Guide, February, 2000.

[8] K. E. Iverson, A Programming Language, John Wiley & Sons, Inc., New York (1962), p. 16.

[9] Iverson K. E, A personal View of APL, IBM Systems Journal, Vol 30 , No 4, 1991.

[10] Krall, A., and Lelait, S., Compilation Techniques for Multimedia Processors, International Journal of Parallel Programming, Vol. 28, No. 4, pp 347-361, 2000.

[11] Leupers, R., Compiler Optimization for Media Processors, EMMSEC 99/Sweden 1999.

[12] Metcalf, M., and Reid., J., The F Programming Language, OUP, 1996.

[13] Peleg, A., Wilke S., Weiser U., Intel MMX for Multimedia PCs, Comm. ACM, vol 40, no. 1 1997.

[14] Srereman, N., and Govindarajan, G., A Vectorizing Compiler for Multimedia Extensions, International Journal of Parallel Programming, Vol. 28, No. 4, pp 363-400, 2000.

[15] Étienne Gagnon , SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK, School of Computer Science McGill University, Montreal , March 1998.

[16] Texas Instruments, TMS320C62xx CPU and Instruction Set Reference Guide, 1998.