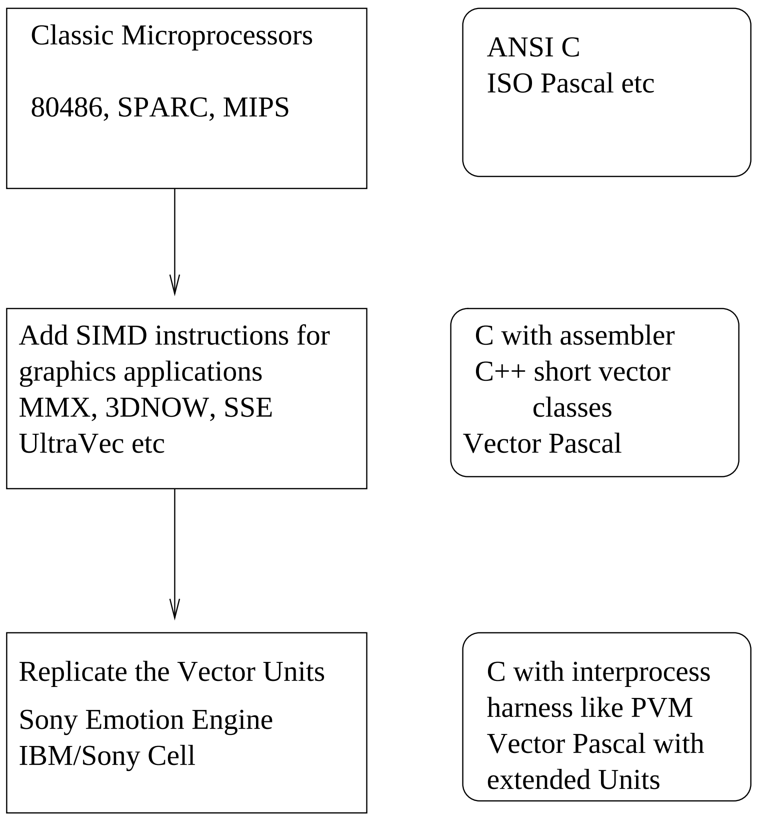
SIMD code generation

Paul Cockshott


Classic Microprocessors
80486, SPARC, MIPS
ANSI C
ISO Pascal etc
Add SIMD instructions for graphics applications
MMX, 3DNOW, SSE
UltraVec etc
C with assembler
C++ short vector classes
Vector Pascal
Replicate the Vector Units
Sony Emotion Engine
IBM/Sony Cell
C with interprocess harness like PVM
Vector Pascal with extended Units

Does Vector Pascal differ from Vector extensions to C for SIMD microprocessors?

Feature	C extension	Vector Pascal
Vector types	small set of fixed size vectors	arrays of any size or rank
Vector Ops	procedural syntax	all operators extended including user defined
Saturated math	procedural syntax	+: -: or Pixel type
slicing	not supported	supported

Summary of new Features in VP

Overloading of all operators to array types
Functions map over arrays
Matrix transpose operator
Array permutation operators
Array slices
Generalised reduction operations
Saturated arithmetic operators `+:`, `-:`
Operators `MIN`, `MAX`
Conditional expressions
Operator overloading
Polymorphic functions
Dimensioned types
Pixels as fixed point type
Input and output of scalar types
Input and output of arrays
Optional garbage collection
Literate programming support
Sets of arbitrary size
 and non ordinal type

An example program listing

```
program tables ;
var
    Let $\alpha$, b, c, d $\in$ array[1..5] of integer;
    Let t $\in$ array[1..5]of array [1..5] of in-
    teger;
begin
    $\alpha \leftarrow \iota_0$;
    $t \leftarrow \alpha \times \alpha^T$;
    write(t);                          times tables
    $b \leftarrow \sum t^T$ ;
    writeln(b);                        sum of columns

    $c \leftarrow 2^{\iota_0}$;                  powers of two

    $d \leftarrow diag\ t$ ;                squares up to 25
    $t \leftarrow c \times d^T$;
    writeln(c, d);
    write(t);
    output table of $i^2 * 2^i$
end .
```

Output produced

```
  1     2     3     4     5
  2     4     6     8    10
  3     6     9    12    15
  4     8    12    16    20
  5    10    15    20    25

 15    45    90   150   225

  2     4     8    16    32
  1     4     9    16    25

  2     4     8    16    32
  8    16    32    64   128
 18    36    72   144   288
 32    64   128   256   512
 50   100   200   400   800
```

And the same as a source program

```
program tables;
var alpha,b,c,d:array[1..5] of integer;
 t:array[1..5]of array [1..5] of integer;
begin
  alpha:= iota 0;
  t:= alpha*trans alpha;
  write(t);{times tables}
  b:=\+ trans t;
  writeln(b);{ sum of columns}
  c:= 2 pow iota 0; { powers of two }
  d:= diag t; { squares up to 25}
  t:= c * trans d;
  writeln(c,d);
  write(t);
(*! output table  of $i^2*2^i$ *)
end.
```

../ilcg/system.eps not found!

ILCG

The purpose of ILCG (Intermediate Language for Code Generation) is 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 a notation for 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 [**?**]. This then generates equivalent sequences of assembler statements.

2. Machine descriptions written as ILCG source files are input to code-generator-generators which produce java programs which perform function (1) above.

So far one HLL compiler producing ILCG structures as output exists: the Vector Pascal compiler. There also exists two code-generator-generators which produces code generators that use a top-down pattern matching technique analogous to Prolog unification.

ILCG is intended to be flexible enough to describe a wide variety of machine architectures. In particular it can specify both SISD and SIMD instructions and either stack-based or register-based machines. However, it does assume certain things about the machine: that certain basic types are supported and that the machine is addressed at the byte level.

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*.

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**.

Typed formats

Each of these underlying formats can contain information of different types, either signed or unsigned integers, floats etc. ILCG allows the following integer types as terminals :**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**.

Ref types

ILCG uses a simplified version of the Algol-68 reference typing model. 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.

Type casts

The syntax for the type casts is C style so we have for example `(ieee64) int32` to represent a conversion of an 32 bit integer to a 64 bit real. These type casts act as constraints on the pattern matcher during code generation. They do not perform any data transformation. They are inserted into machine descritions to constrain the types of the arguments that will be matched for an instruction. They are also used by compilers to decorate ILCG trees in order both to enforce, and to allow limited breaking of, the type rules.

Arithmetic

The allowed dyadic arithmetic operations are addition, saturated addition, multiplication, saturated multiplication, subtraction, saturated subtraction, division and remainder with operator symboles **+, +:, *, *:, -, -:, div , mod**..

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

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)**

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.

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)**$\uparrow$ **((ref octet)mem(99))**

The argument to the load operator must be a reference.

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.

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'];

Register sets

A set of registers that are used in the same way by the instructionset can be defined. **pattern reg means [**$EBP|EBX|ESI|EDI|ECX|EAX|E$
;

pattern breg means[$AL|AH|BL|BH|CL|CH|DL|DH$

All registers in an register set should be of the same length.

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)**

Vector registers

```
register quadword XMM0
 assembles[ 'XMM0'];

alias register ieee32 XMM00=XMM0(0:31)
 assembles['xmm0'] ;

alias register ieee64 XMM0R64=XMM0(0:63)
 assembles['xmm0'] ;

alias register ieee32 vector (4) XMM0R324=XMM0(0:127)
 assembles['XMM0'];

alias register ieee64 vector (2) XMM0R642=XMM0(0:127)
 assembles['XMM0'];
```

Use of M4 macro processor to define short-cuts for type casts

```
define(singlequad, (ieee32 vector(4))$1)

define(refsinglequad,(ref ieee32 vector(4))$1)

define(i8x16, int8 vector(16))

define(u8x16, uint8 vector(16))

define(i16x8, int16 vector(8))

define(i32x4, int32 vector(4))

define(r64x2, ieee64 vector(2))

define(i8x16, int8 vector(16))

define(i16x8, int16 vector(8))

define(i32x4, int32 vector(4))
```

Declare operators

```
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’];
```

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.

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.

Vector instructions

```
instruction pattern
OPPD(soperator op, xmmr64 r2,xmmr64 r1)
 means[ (ref r64x2) r1 :=
  op( (r64x2) ^( r1), (r64x2)^( r2))]
 assembles[op 'pd ' r1 ',' r2];
```

Control structures

Ilcg provides **goto** and **if then** as control structures along with explicit labels - it is thus close to machine language.

```
instruction pattern GOTO(jumpmode l)
means[goto l]
assembles['jmp ' l];
instruction pattern
 IFLITGOTO(label l,addrmode r1,
          signed r2,condition c,
          signed t,int b)
means[if((b)c((t) ^(r1),const r2))goto l]
assembles[' cmp 't' ' r1 ',  '  r2
          '\n j' c ' near  '  l];
```

It also allows **for** loops.

```
instruction pattern
        REPMOVSD(countreg s,maddrmode m1,
                sourcereg si, destreg di)

means[for (ref int32)m1:=0 to ^(s) step 1 do

(ref int32)mem(+(^(di),*(^((ref int32)m1),4))):=
  ^((ref int32)mem(+(^(si),*(^((ref int32)m1),4))))
        ]

assembles[' inc ecx\n rep movsd'];
```

This is useful both for block moves and optimizing vector operations. Code generator automatically vectorises for-loops looking for vector intsructions to do this.

Choice of Instructions

Instructions are chosen in the priority order given in a list of instructions at the end of the ILCG machine spec.

```
instructionset[
LDW|LDB|LDBU|LDH|LDHU|LDHH|LDHUH|
 STW|STH|STB|
 RI|RRR|NOR|NOTOP| RRRieee32|RRRBieee32|
IFGOTO|IFBOOL|SET| PLANT|GOTO|GOTOINDIRECT|
PLANTICONST|PLANTSCONST|
PLANTBCONST|PLANTWCONST|
/* make mov last to prevent redundant moves */
MOVIA|MOVI|MOVIU |MOVI32|MOV|

PUSHR]
```

Use of Ilcg

1. Write a standard parser using top down of machine generated parsing techniques

2. Generate a tree of the code in ILCG format

3. Pass the tree to an automatically constructed tree walker which walkes over it to produce assembler

A:

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

↓

C:

```
cmp DWORD[ ebp+-19208],6399
jg NEAR l4847d577
mov ecx, DWORD [ ebp+-19208]
movq MM1, [ ecx+ebp +-6400]
paddb MM1, [ ecx+ebp +-12800]
movq [ ecx+ebp +-19200],MM1
add DWORD [ ebp+-19208], 8
jmp l4843d577
l4847d577:
```

↑

B:

```
(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)))
 )
```

A normal Pascal unit has 4 main parts:

1. A **uses** list which specifies which other units it imports types, data or procedures from.

2. An **interface** part which specifies the identifiers that are to be exported from the unit.

3. An **implementation** part that contains both identifiers that are private to the unit and also the procedure bodies of any procedures exposed in the interface part.

4. Finally there is an **initialisation** block that executes prior to program startup time, to ensure that variables in the block are appropriately intialised.

```
program skyslider;
uses dyanmics, views,cmath;
....
```

If we want to place this on a PS2 we change the program header to:

```
program skyslider;
uses dyanmics[0], views[1],cmath;
....
```

Thus indicating that the unit `dynamics` was to be mounted on attached processor 0 (APU) and `views` was to be mounted on attached processor 1.

The fact that memory is not shared between units imposes restraints on what identifiers may be exported from a unit that is being compiled to an APU. Only constants, types and procedures can be exported from a unit running on an APU.

Units used by units that are mounted on an APU must themselves be compiled and mounted on the APU. If the unit `dynamics` made use of a further unit `cmath` handling complex maths then a copy of `cmath` would be generated in the instructionset of the attached processor. This copy of `cmath` would be distinct from that seen and used by the main program.

The absence of shared memory has other implications for what is exported accross unit boundaries. If a procedure is exported from an APU mounted unit, then that procedure must use call by value semantics and, moreover, the types passed as parameters must not include any pointers.

The absence of `var` parameters prevents procedures from passing any information back to the calling environment.

Functions on the other hand, return results by a copy back mechanism, which is adaptable to the DMA mechanisms used to transfer information between processors on the PS2 or the PS3.

```
UNIT views;

INTERFACE

CONST maxv=40;

TYPE matrix4x4= array[1..4,1..4] of real;
     coordblk = array[1..maxv,1..4] of real;
     PROCEDURE setviewtransform(m:matrix4x4);
     PROCEDURE processvertices(c:coordblk);
     FUNCTION  getvertices:coordblk;

IMPLEMENTATION

VAR t:matrix4x4;
    transformed:coordblk;
    PROCEDURE setviewtransform;
    BEGIN   t:=m  END;
    PROCEDURE processvertices;
    BEGIN
      { do matrix multiply }
      transformed := t . c
    END;
    FUNCTION getvertices;
    BEGIN getvertices:=transformed END;

END.
```

Status

Work on the PS2 compiler reached the stage that allows programs to run in SIMD mode on the MIPS control processor using VPU0, but the DMA transfer of parameters to units has not yet been implemented.

Conclusion

The language Vector Pascal alreay provides a number of the key elements needed to support microgrids. In particular it has

1. Support for the SIMD model of programming supported by machines like the Cell.

2. A readily retargetable backend that allows configuration to new instructionsets.

3. Dynamically loadable code generators so that hetrogenous code can be output.

Further, we have shown above that a relatively simple extension to the Pascal Unit syntax allows both the parameterisation of units to multiple hetrogenous cores, and provides a natural model for coarse grained parallelism without the need to introduce any new control constructs beyond those already provided by procedures and functions.