An intermediate language for code generation

SICSA workshop on Domain Specific Languages ,

W P Cockshott

Glasgow University

May 1 2014

Machine description languages

VHDL - input to chip synthesis software - not limited to computers but any digital chip

APL - (Iverson) invented as machine description language for IBM 360

Register transfer languages - used to describe other hadware - typically used in simulators when designing processors

ISP - (Bell and Newall) used to describe the instructionset semantics of an arbitrary machine at the binary level

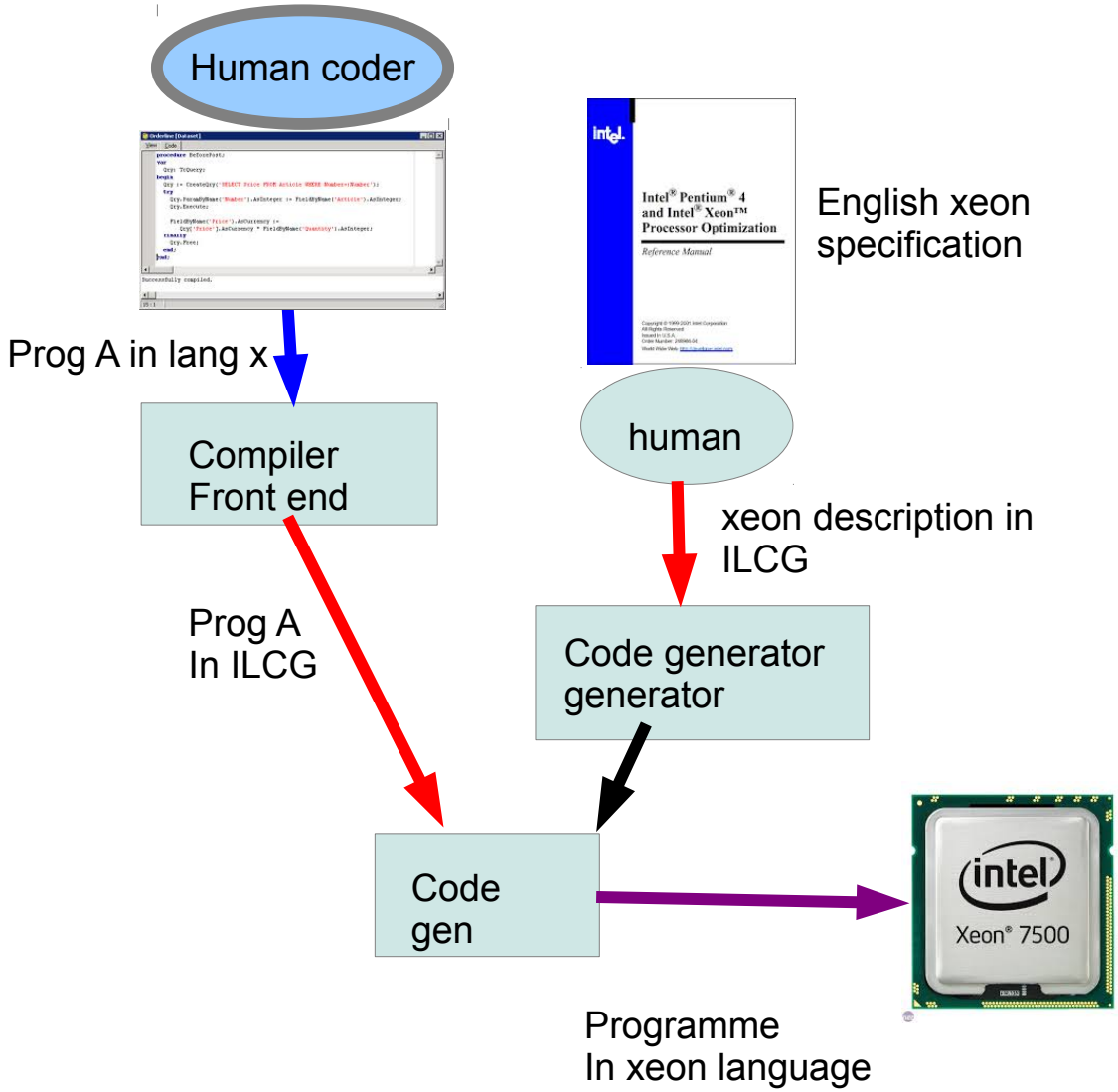
ILCG - designed to describe the instructionset semantics at the assembler level, abstracts from binary code

Purpose

The notation must:

- **Describe** the state of the machine
- **Describe** the possible state transitions of the machine
- **Describe** the corresponding assembly language semantics
- **Optionally** it may describe code transformations that preserve semantics
- **Act** as an intermediate format that a compiler can use to describe a sequence of machine specific state changes
- **Act** as an input to an automatic code generator generator

Overview of ILCG use



Range of machines

State descriptions depend on how machines organise their state, the groups of bits that they can store or manipulate in some coherent way.
There have been a large range of ways of doing this, different word sizes, different number representations etc.
But these have converged into a narrower range of representations : byte addressing, twos complement, IEEE.
There are still issues like register/vs stack organisations etc.
The problem is that the ways of manipulating state keep growing. I had designed the system to have the state manipulation facilities needed in 2000. Since then a whole bunch of new machine facilities have arisen: how to handle this?

Supported types

Programming languages imply that we should organise state in terms of types - typically not higher order since the machines do not support this.

The data in a memory can be distinguished initially in terms of the number of bits in the individually addressable chunks.
In ILCG the addressable chunks are assumed to be the powers of two from 3 to 7, so we thus have as allowed formats words of lengths 8, 16, 32, 64, 128 bits.
When data is being explicitly operated on without regard to its type, we have reserved words which stand for these word lengths: `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. 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*.
Floating point numbers are either assumed to be 32 bit or 64 bit numbers.
Since the IEEE representation has become standard we call these types `ieee32, ieee64`.
Note that probably we should now extend the language to include `ieee16` which has recently been introduced.

Vector Types

Earlier generations of computers operated on scalar values. Most new machines operate on vectors of primitive types. The ILCG type system thus allows the declaration of bounded vector types.

So we can denote the type of a vector of 4 $\times$ 32 bit floats as

```
ieee32 vector ( 4 )
```

Storage locations

Machine instructions can themselves contain numbers, and they can contain references to storage slots that can contain numbers.
We have to be able to describe the different semantics of the assembler instructions

```
ADD R8D, 1000 ; add 1000 to the bottom 32 bits
              ; of register 8
```

and

```
ADD R8D,dword [1000]; add the contents of the 32 b
                    ;at location 1000 to the botto
                    ;bits of register 8
```

Ref types

We borrow the idea from Frege via Algol68 that the type of a variable that can contain a type t is a `ref t` .
So the type of a memory location holding an `ieee32` is a `ref ieee32`.
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.

Examples

```
ADD r8d, 1000
```

≡

```
(ref int32)r8d := +((int32) ^(r8d),(int32) 1000)
```

versus

```
ADD r8d, dword[1000]
```

≡

```
(ref int32)r8d := +((nt32)^(r8d),^((ref int32)mem(1000)))
```

The syntax for the type casts is C style, we explicitly describe the type of the data being assumed in registers and memory.
^ means fetch from a location
:= means store in a location If the left hand side is a format (ie `octet, word` etc) 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

Arithmetic

The allowed binary operations are : `+,*,-,div , mod, +:,-:, *:, MAX, MIN, AND, OR, XOR, =, <=, >=, >, <, <>`. The arithmetic is assumed to take place at the precisions of the types specified in the arguments. Arithmetic between different types is not allowed.

Saturated arithmetic `+:, -:, *:`, is defined on integer types so that there is no wrap round in the case of overflow. So `+:((int8)126,(int8)2)` $\rightarrow$`(int8)127`

In the case of saturated multiplication of 8 bit integers the numbers are treated as signed binary fractions.

These operations are required to describe certain image processing primitives. Comparision operations return -1 if true, 0 if false. Again this is chosen to be consistent with the semantics of MMX and SSE instructions.

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

Monadic operations

There are a set of monadic arithmetic operations

NOT	bitwise inversion of its inputs
SIN,	COS, TAN computes trig functions arguments must be ieee32 or ieee64
TRUNC	give the non fractional part of a floating point value
ROUND	converts a real to the nearest integer
FLOAT	convert an integer to a floating point
EXTEND	higher precision representation. If unsigned, zero extended. If signed, sign extended.
ABS	the absolute value of a number
SQRT	the square root of a floating point value, precision same as input
SQR	the square of a number, output type the same as the input
LN	the natural logarithm of a floating point value.

Control

The following control mechanisms are *meanings* in ILCG

if statements which take the form `if(value)meaning`

statement chains group meanings together `statement(meaning, meaning)` note that a statement is itself a *meaning*

goto statements which take the form `goto value`

fail statements which denote an interrupt mechanism of the form `fail value`

labels are possible values which can be part of a sequence or of an expression

for loops of the form `for refval := value to value step value do meaning` these allow semantic description of forms like the Intel block move instructions

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)
```

.

Machine description

llcg can be used to describe the semantics of machine instructions. A machine description typically consists of a set of register or stack declarations followed by a set of instruction formats and a set of operations and a set of address modes.

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'];
```

Additionally registers may be reserved, indicating that the code generator must not use them to hold temporary values:

```
reserved register word ESP assembles['esp'];
```

Patterns

ILCG allows the definition of general patterns of three sorts

- Alternative patterns
- Parameterised patterns
- Transformer patterns

Register sets

Machines often have sets of registers that have a common property. These may be defined using alternative patterns. A set of registers 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 a register set declared this way should be of the same type. This uses the general alternative pattern facility of ILCG. The names given in a list of alternatives must all have been predeclared.

Where recursive definitions are required you can predeclare a pattern thus

```
pattern mypat;
```

in which case a full definition must be given lower down.

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)`

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.

Instruction formats continued

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 `maddrmode` and `reg` must be pre declared patterns.

Declaring maddrmode

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 patterns as parameters to patterns.

Unification

Pattern matching proceeds by attempting to match the `means` part to an ilcg tree.
Matching goes left to right and fails as soon as a token in the pattern is not matched in the tree. If a token in the pattern is a parameter, the matching rule for the type of the parameter is invoked.
Associated with each parameter are two slots in the pattern frame. The first slot will, on a successful match of the parameter, hold the tree it matched. The second slot will hold the assembly code output by the parameter pattern.
If a parameter occurs twice in the `means` part, the subtree corresoponding to the second occurence must be identical to the one matched on the first occurence.

Unification continued

The following pattern describes a memory increment instruction. It is intended to match any occurence of a tree that adds one to a memory location. Thus the location (`rm` in the example) occurs twice in the pattern. The parameter t is there to check that the type of the memory location is an integer, since this instruction does not work with reals.

```
instruction pattern INCm(maddrmode rm,int t)
 means[(ref t)rm:= + (^(rm),1)]
 assembles['inc  ' t ' ' rm];
```

When a parameter occurs in the assembles part, the assembler string associated with the previous match of the parameter is substituted. The assembles part thus concatenates the explicit strings with the evaluated parameters. If the pattern is an `instruction pattern`, a trailing newline is output.

Commuting operators

If an operator in a `means` part commutes, then the matcher will attempt to match the swapped version of a pattern if the first attempt fails.
Thus the previous example would match either

```
(ref int32) mem(100):= +(1,^(mem(100)))
```

or

```
(ref int32) mem(100):= +(^(mem(100)),1)
```

Register substitution

Suppose now that we have a more sophisticated instruction, that will perform arithmetic between a byte register and a memory location:

```
instruction pattern RMRB( nonmultoperator op,breg
 means[(ref t)rm :=op((t)^( rm),(t)^( r1))]

 assembles[op ' ' 't ' rm ',' r1];
```

And suppose we try to match it to
`(ref int8)mem(100):= -((int8)^(mem(100),^((ref int8)mem(102)))`
everything matches until we get to the mem(102) which we are attempting to match to a `breg`. Now suppose that we define breg as

```
pattern breg means[
BL|DL|AL|CL|R8B|R9B|R10B|R11B|R12B|R13B|R14B|R15B];
```

What happens?

The matcher associates with each register a free flag. It will scan the register set and find the first free register (BL) , clears its free flag, and attempts to match the pattern

```
BL:=^((ref int8) mem(102))
```

Searching through all the instruction patterns it finds

```
instruction pattern LOAD(maddrmode rm,anyreg r1,
 means[ (ref t) r1:= (t)^(rm )]
 assembles['mov ' r1 ',' t ' ' rm];
```

This matches and outputs

mov BL, BYTE [102]

and the RMRB pattern now matches to give

sub BYTE[100],BL

after which the free flag for BL is again set. The two instructions correctly implement the original tree.

Transformer patterns

These patterns produce new IICG trees as output rather than assembly code. They can perform complex code transformations - in this example vectorising for loops to run on the Intel XeonPhi

```
transformer pattern vectorisablefor ( any i, any start, any finish,
                                      vecdest lhs,vectorisablalternatives rhs
 means[for(ref int32)i:=start to finish step 1
       do lhs[i]:= rhs[i]]
 returns[ statement(
  for i.in:= +(+(-(*(div(+(1,-(finish.in,start.in)),16),16),1),star
   to finish.in step 1
   do lhs.in := rhs.in ,
/* the above is a scalar remainder loop */
  for i.in:= start.in
   to +(-(*(div(+(1,-(finish.in,start.in)), 16), 16 ),1), start.in)
   step 16
   do lhs.out:=rhs.out )/* this is the vectorised loop */
];
```

Preconditions

A transformer pattern may have preconditions associated with it:

```
transformer pattern vecdest(any i,any r, scaledindex j)
 /* this rule recognises addresses that can
          be readily converted to vector addresses */
   means[(ref ieee32)mem((int64)+(r, j[i]))]
   returns[mem(+(j.in,r.in),ieee32 vector(16))]
   precondition[NOT(CONTAINS(r.in,i.in))];
```

The example above checks that a tree corresponds to a destination address that can be vectorised. The address must be made up of two components `r` and `j[i]` where `r` can be any arithmetic expression and `j[i]` must be a scaled index expression: one that multiplies an integer by 4 (the size of an `ieee32`). But `r` must not contain `i` since this would mean that the successive vector destinations were not adjacent and so could not be packed in a SIMD register.

Constructing a complete compiler

There is an extension to Ilcg, unimaginatively called Ilcg2, that I have used in teaching compiler courses. Ilcg automates code generator construction. Ilcg2 automates the front end, and the main driver programme of the compiler as well, and leaves the students to just implement a syntax tree to Ilcg translator. Using Ilcg2 I was able to build from scratch a compiler for Greg's new language Haggis, to Pentium machine code in just under a week of evening work.
The Ilcg2 file specifying the Haggis compiler consists of:

```
compiler haggisc /* haggisc.java is the compiler produced */
parser haggis    /* haggis.sab is the syntax spec */
cpu gnuPentium   /* gnuPentium.ilc is the Ilcg machine
                    spec */
```

Syntax specification

Ilcg is iteself defined in terms of Sable - the McGill compiler compiler. Ilcg2 allows you to define a front end (for Haggis in this case) using a Sable grammar spec haggis.sab.
This results in a lexical analyser and syntax analyser being written in java that can translate Haggis into a syntax tree. It also provides a visitor class that can walk over the syntax tree. The compiler writer now just has to
a) write down the grammar of the source language in sable notation
b) write an extension of the visitor class that will translate the resulting Haggis syntax tree into equivalent Ilcg.
c) supply the machine specification file (gnuPentium.ilc is one of many such machine specs in the library)
Using this technique 3rd year students have been able to construct a compiler for a simple language of the Haggis level in 6 weeks of labs.

Downloads

You can obtain the Ilcg systems (both 1 and 2) by downloading the source for the Vector Pascal Compiler from Source Forge, where they are used as part of the compiler.
You get with this a library of machine descriptions that covers many recent processors, including a pretty comprehensive coverage of the AMD and Intel processors.
You can get a cut down ILCG system as part of the GHaggis compiler sources also from Source Forge.