CoGenT Language Manual

Version 0.1

1 Types

1.1 Built-in Types

The CoGenT description language supports two forms of built-in types:

- *bit arrays*
- *references to bit arrays*

1.1.1 bit arrays

A *bit array* is a type that can be used to store a set of bit values. Individual bits are accessed by their index into the array surrounded by brackets. A bit array can be set by assigning a value to an array variable and similarly a value representation of the bits contained in a bit array may be retrieved by referring to that array. A bit array is declared using the **bit** keyword and a size contained within square brackets. For example, we can define a variable named *inst* that is a bit array of size 32,

```
bit[32] inst;
```

It is also possible to have multidimensional bit arrays. The following declares a bit array variable *regs* that is 32 rows of 32 bits in each row.

```
bit[32,32] regs;
```

We can retrieve values using the following syntax,

`inst`	refers to the value represented by the 32 bits
`inst[31]`	refers to single bit 31
`regs[4]`	refers to the value of all 32 bits at index 4
`regs[4,5]`	refers to single bit 5 of the 32 bits at index 4

Assignment to bit arrays is allowed provided that the size and number of the dimensions are the same. For instance, the following are valid:

```
bit[30, 29] uneven;
bit[30, 30] even;
bit[30, 29] uneven2;
bit[32] uint32;
bit[32] integer;

uneven = uneven2;
uint32 = integer;
```

But the following are invalid:

```
uneven = even;
even = uneven;
even = uint32;
```

1.1.2 references

A *reference* to a bit array allows us to treat a range of bits within a previously defined array as another bit array. This reference or alias can then be operated on as if it were a bit array itself. References are useful when we want to view a range of bits as an individual unit within a larger set of bits. The following shows a 6 bit reference of the *inst* bit[32] array declared above,

```
bit[6] op @ inst[31];
```

What we are doing here is also known as providing an *overlay* on a previously defined bit array. It is also possible to recursively provide overlays on references, we discuss this more when we talk about type definitions in a later section. It is important to note here that any modifications to the bits that represent a reference in fact modify the array on which it refers.

1.2 Type Modifiers

Type modifiers allow us to provide more information about a type in order to constrain that type to specific values of which we are concerned. This extra information restricts the interpretation of the kind of values we are allowing a particular type to refer to. The following type modifiers are currently defined,

unsigned	
signed	interpreted as *twos-complement* binary encoding
big	interpreted as a big endian value. (array indexing depends on this)
little	interpreted as a little endian value. (array indexing depends on this)
overlay	allows references to be *overlayed* on the type (*default*).
field	disallows references to be *overlayed* on the type.

The following is an example of a bit array declaration using type modifiers,

```
unsigned big bit[32] inst;
```

Notice here that this is also implicitly an *overlay*. We could have also declared the above as,

```
overlay unsigned big bit[32] inst;
```

Both of these declarations are equivalent. Type modifiers may be specified multiple times and conflicting type specifiers may also be listed. In these situations the left-most specifier has precedence, however, the parsing program will report informational messages noting the conlicts. So in the following examples `b` is an unsigned big-endian array, and `l` is a signed little-endian array.

```
big little little unsigned big bit[32] b;
signed signed little big unsigned little signed bit[32] l;
```

1.2.1 Endianness

Endianness determines the meaning of indexing. A little endian array assigns an index of 0 to the least significant element in an array. A big endian array assigns an index of 0 to the most significant element of an array. Compared to a little endian indexing scheme, the big endian one accesses the same array elements by starting at array index size - 1 and decreasing to zero. Sub-elements each having differing endiannesses can be created by declaring arrays of elements of a different endianness and combining the types. For instance, a little endian array of 32-bit big endian values (`uint32`) would be declared thus:

```
little uint32[1024] memory;
```

This is not an array of 1024 `uint32`s whose endianness has been converted to little endianness. Rather this is an array that is indexed in a little endian fashion, but whose elements are indexed in a big endian manner. The following code accesses the most and least significant bits in the `uint32` stored at 512:

```
bit msb = memory[512][0];
bit lsb = memory[512][31];
```

This code simulates the memory as seen by Intel processors, with memory word-indexed as little endian, each word accessed as four little-endian bytes, and the bytes within a word accessed as big-endian:

```
big bit[8] byte;
little byte[4] word;
little word[1024] memory;
```

Another desired operation may be to convert a value from one endianness to another. To change the indexing, but preserving the value is accomplished through type-casting to the other endianness. For example, after the following code is done, the value of `l` will be the same as the value of `b`.

```
little bit[32] l = 0xDEADBEEF;
big bit[32] b = (big)l;
```

Sometimes it may be necessary to change the indexing but not the actual order of the bits themselves. This is accomplished through assignment to a variable with a different endianness. For example, the result of the following code will be that `b` will have the value `0xDEADBEEF` and `l` will have the value `0xF77DB57B`.

```
big bit[32] b = 0xDEADBEEF;
little bit[32] l = b;
```

1.3 Sign

Cogent allows two kinds of sign modifiers, `signed` and `unsigned`. The primary purpose of these modifiers is to determine how sign-sensitive operators should interpret a value. For instance if a signed value 3 bits wide is extended to 10 bit width, the value of the new bits is dependant upon the sign (0 for unsigned values, and MSB for signed values). Because signed values are interpreted as being in 2's complement form, signed values may require 1 more bit to represent them than the value may suggest. For instance, the unsigned value 3 can be represented in 2 bits ($(11)_2$), but the signed value -3 requires 3 bits ($(101)_2$).

1.4 Constants

CoGenT allows for the specification of constant data. CoGenT follows the C/Java style for such specifications. Strings of digits are assumed to be in decimal, hex values are prefixed with “0x”, octal values are prefixed by “0o”, and binary values are prefixed with “0b”. The follwing declarations all specify the same constant value.

```
b = 10;
b = 0xA;
b = 0o12;
b = 0b1010;
```

Note that constants have no particular size. Although a constant value may require a certain minimum number of bits to represent it, there is no reason why it can’t be stored in a larger number of bits. The size of values is important for many operators and so CoGenT requires that the programmer specify the type of a constant value. Note that this is not as burdensome as it may seem. A type declaration for a type `const` can be made and all constants declared therewith. For example:

```
type const = unsigned big bit[32];
const b = 0b1010;
```

The preceding code effectively causes all constants to default to 32bit representation (useful for a 32-bit architecture, no?). It also has the valuable side-effect of annotating which values in the code are constants.

1.5 Type Declarations

Now that we understand the simple built-in types provided by the CoGenT language, it is important to have the ability to build more complex types. The CoGenT language provides this in the form of *type declarations*. A type declaration allows us to incrementally build types which may also be used in the definition of other type declarations. This gives us the ability to represent collections of bit arrays in arbitrarily complex ways. For example, it may be the case that all the items that we want to talk about are unsigned big-endian 32-bit integers. We do not want to specify this information for every variable declaration. Instead, we prefer to create a new type which represents those traits we are concerned about,

```
type uint = big unsigned bit[32];
```

We can now declare variables to be of type *uint*,

```
uint inst;
uint[32] regs;
```

Notice here that we can also specify a size on the type as we did for `uint[32]`. It should be clear that what we are declaring is 32 rows of 32 bits and each of those 32 bit rows are big-endian and unsigned.

1.6 Subranges

Another kind of type declaration is called *subranging*. Subranges are to types as references are to bit arrays. Just as references are used to refer to a portion of a particular bit array, subranges allow us to refer to portions of another type

by name. These subranges can be thought of as a union type in the programming language C. The following example shows a type declaration followed by several subrange declarations,

```
type uint = bit[32];
subrange fb = bit[1] @ uint[0];
subrange sb = bit[1] @ uint[1];
subrange slb = bit[2] @ uint[30];
```

We could then declare a variable of type *uint* and refer to its subranges,

```
uint r1;
r1.fb = 1;
r1.sb = 0;
r1.slb = 3;
```

If we were to translate this into an equivalent C like union, we would have something like the following,

```
struct subranges {
    fb : 1;
    sb : 1;
    pad : 28;
    slb : 2;
};

union {
    unsigned int r1;
    subranges s;
} ur1;

ur1.s.fb = 1;
ur1.s.sb = 1;
ur1.s.slb = 3;
```

It is also possible to have subranges of subranges. For example, we could have the following declarations,

```
type footype = bit[4];
subrange mid = bit[2] @ footype[1];
subrange left = bit[1] @ mid[0];
subrange right = bit[1] @ mid[1];

footype bar;
bar.mid = 3;
bar.mid.left = 0;
bar.mid.right = 1;
```

It is not possible to have recursive subranges.

It is possible to have subranges of a different endianness and signedness from the type it is subranging. The following example illustrates such a use:

```
type ieee32BitFloatingPoint = big unsigned bit[32];
subrange sign = bit[1] @ ieee32BitFloatingPoint[31];
subrange exp = bit[8] @ ieee32BitFloatingPoint[30];
subrange mantissa = little bit[23] @ ieee32BitFloatingPoint[22];
```

In this example, the mantissa portion of the `ieee32BitFloatingPoint` type is little-endian even though the encapsulating type is big-endian.

1.7 Operators

CoGenT provides several primitive operations on bits and bit arrays. Boolean operations return a bit, 1 for true, 0 for false. The following table illustrates the operations, in this table the string "<value>" is the value being operated on. Anything between < > is to be provided by the user. In the following table references to the front of a variable mean the MSB of the variable, while refrences to the back of a variable mean the LSB of the variable.

Boolean Operations

comparison	==
<,>,≤,≥	<,>,<=,>=
AND	&&
OR	||
XOR	^^
NOT	~~

operation	**syntax**	**notes**
front resize	#(<value>, <new size>, <padding>)	resizes <value> to <new size>, adding <padding> to front, if necessary
back resize	##(<value>, <new size>, <padding>)	resizes <value> to <new size>, adding <padding> to back, if necessary
sign extend	!(<value>, <new size>)	≡ #(<value>, <new size>,((little)<value>)[0])
zero extend	!0(<value>, <new size>)	equivalent to #(<value>, <new size>, 0)
2's complement	-<value>	standard 2's complement
concatenation	<value1> :: <value2>	appends <value2> to <value1>
bitwise AND	&	pads front of short argument with zeros
bitwise OR	|	pads front of short argument with zeros
bitwise XOR	^	pads front of short argument with zeros
bitwise NOT	~	
shift	<<,>>	left shift, right shift, padding depends on sign (signed → sign-preserving padding, unsigned → zero padding)
rotate	<<<,>>>	Rotation of a value
population count	$1(<value>) , $0(<value>)	counts number of 1s and 0s, respectively
$+, -, \div, \cdot$	+,-,/,*	short argument is padded with zeros (WHAT ABOUT OVERFLOW??)
modulo	%	Mathematical modulus (e.g. $-1 \equiv 3(4)$)
increment, decrement	<value>++, <value>–	increments and decrements <value>
assignment	<value1> = <value2>	<value1> and <value2> must have the same number of dimensions and same size, just copies <value2> into <value1> bit by bit
type cast	(<type>)<value>	produces a new variable of type <type> with value derived from <value>.

2 Classes

The CoGenT specification language allows for the use of classes. These classes are similar to classes from general-purpose object oriented languages. However, the CoGenT classes have several unique restrictions. CoGenT classes can contain data and methods. CoGenT classes support single inheritance. There are no interface types such as in Java. Another important distinction is the inclusion of a class within another class. In a general dynamic O-O language, if a class A includes another class B, the data referred to as B can be an object of type B, or an object of a class descended from B. What this means is that it is difficult (sometimes impossible) to determine the size of a class that includes other classes. Because this is such a potentially useful feature, CoGenT allows a restricted form of inclusion. In CoGenT, when a class contains a member that is another class it means that the included member can only be from that class – not from a descendent. Trying to assign an instance of a descendant class to that member is therefore an error. In the following example, class B contains an instance of A. What this means is that instances of B will contain two bit arrays, `half` and `byte` (from `A`). Suppose we create a B named `ab` and a C named `ac`, the following code fragment would be an error: `ab.foo = ac;`.

```
class A
{
  little bit[8] byte;
}

class B
{
  big bit[16] half:
  A foo;
}

class C extends A
{
  bit bar;
}
```

As a consequence of this, when a CoGenT class is declared to include some data, that means that the class literally contains that data at that point. In short, CoGenT classes have no references.

2.1 Attributes

Unlike class definitions in object-oriented languages such as Java, a CoGenT class can define what is called an *attribute* instead of a method. An attribute has the look and feel of a method but is meant to convey properties of a class rather than executable code. These attributes are dependent upon the particular description we are composing. For example, if we are writing a description for instruction set syntax we may be required to define an attribute *effect* and an attribute *asm*. They would look somewhat like the following,

```
class inst_add
{
  big unsigned bit[6] ra;
  big unsigned bit[6] rb;
  big unsigned bit[6] rd;
```

```
  effect() {
    addi()
  }

  asm(ra,rb,rd) {
    ''addi''
  }
}
```

The *effect* attribute requires a "call" to a *mixin* function (more on these later) which defines the semantic meaning of an instruction. The *asm* attribute requires a format string which defines the assembly string output of an instruction. It may seem that attributes are exactly the same type of thing as methods. They look the same and they appear to convey the same type of meaning. They are, however, quite different semantically and are meant as an expressive form of conveying descriptive information of the type of object the class is attempting to describe. The meaning of an attribute may be interpreted quite differently between different descriptions. We defer what this means exactly to our discussion later on the construction and interpretation of CoGenT machine descriptions.

2.2 Constraints

An additional feature that seperates a CoGenT class from a standard type declaration is that a class can include constraints on the value of a member. A constraint has the following form: `<field name> = <contrained value>`. This is useful for generating instruction parsers and emitters. For parsing, a constraint implies that the field will have the constant value. For emitting, a constraint implies that the emitter needs to write out that constant value to the emitted instruction. The following code describes a class that contains an unsigned byte, the first two bits of which must be 1.

```
class A
{
  big unsigned bit[8] byte;
  big unsigned bit[2] hiBits @ byte(7);

  hiBits = 0x3;
}
```

2.3 Mixins

As mentioned previously, CoGenT classes may contain *attribute* definitions. These are only meant to convey specific pieces of information regarding the class we are trying to describe and are not meant as executable code in any way (as demonstrated by the definition of the class *addi*). It may be, of course, that we do want to attach a form of executable-like code to a class. This is clear when we are describing the semantic actions of instruction set architectures. We would like to express the function of a particular instruction I_i as one with the following signature, $I_i * S \Rightarrow S'$, where S is some machine state and S' is a new machine state. In other words, we would like to define the meaning of an instruction in terms of some machine state and a new resulting machine state.

The mechanism used for this is mixins. A mixin is basically a class containing only methods These methods may contain free variables which refer to the members of some class such that, when the methods of a mixin are actually used within a given class, all the free variables will be bound within that object (and therefore all the types will be known). This is useful for specifying the semantics of instructions, as different implementations of a method could

co-exist and be chosen by mixing in one rather than the other. This obviates the need to have method overloading and of specifying 'dummy' placeholder methods high in the inheritance tree in order for all the children to have access to the method. With mixins, any class can use any defined mixin. Such uses may cause errors (mixing in multiple conflicting implementations), but those can be detected statically. While CoGenT classes are a means for composing interelated forms of data (as in instruction set syntax or machine state), mixins are a powerful tool for composing interelated forms of semantic meanings of architectural components. We can then use attributes as a way to bridge the syntactical details with the intended meaning.

Here is a simple example of a mixin,

```
mixin arith requires(ra,rb,rd)
{
  addi() { gp[rd] = gp[ra] + gp[rb] }
}
```

The above defines a mixin named `arith`. This mixin *requires* that the class using the mixin must have defined variables named ra, rb, and rd. The body of the mixin defines a function, `addi`, for the addition of two registers and the result of which is stored into a third destination register. *The actual meaning of* `gp` *is explained in a later section on the machine state description, those details are not necessary for understanding the use of mixins.* We showed in the section on *attributes* an attribute name *effect* which used a mixin by the name of `addi`. That example was missing some details, we complete it here:

```
class inst_add uses arith
{
  big unsigned bit[6] ra;
  big unsigned bit[6] rb;
  big unsigned bit[6] rd;

  effect() {
    addi()
  }

  ...
}
```

Previously, we left out the `uses arith` declaration of the class. It is, of course, necessary to specify which mixin you are using. We could have also defined that class as,

```
class inst_add uses arith(ra=rx,rb=ry,rd=rz)
{
  big unsigned bit[6] rx;
  big unsigned bit[6] ry;
  big unsigned bit[6] rz;

  effect() {
    addi()
  }

  ...
}
```

In the above example, we are simply assigning aliases to variables defined in the class `inst_add` to be used in the `arith` mixin. It is also possible to use constants which give mixins a "macro" like quality,

```
class inst_add uses arith(ra=1,rb=2,rd=1)
{
  effect() {
    addi()
  }

  ...
}
```

Note in the above example that we in fact changed the "meaning" of the use of the `addi` function by making the destination register, *rd*, the same as one of the source registers, *ra*.

It is important to note that it is also possible to have mixins that "use" other mixins. These more advanced forms of composing mixins are demonstrated in the discussion of specific machine descriptions later in this manual.

3 Descriptions

The goal of Cogent is to provide a collection of tools for which it is possible, given partial or total descriptions of a machine, instruction set, instruction semantics, architecture pipelines, calling conventions, and others, to generate useful and necessary components for compilers and simulators. The purpose of the CoGenT language is to allow the description writer to do this easily, consistently, incrementally, and in a modular fashion. The CoGenT language also aims for simplicity and expressiveness without sacrificing completeness. The following sections describe several types of descriptions or *specifications* that are possible with the CoGenT language.

3.1 Machines

A machine description describes the abstract state of a machine (for semantic purposes).

3.2 Instructions

An instruction description gives a name to a grouping of instructions (perhaps specified by parent class?).

3.2.1 Syntax

3.2.2 Semantics

4 Formal Grammar

```
<identifier> = [A-z][0-9,A-z]*

<pos integer> = [0-9]+    //decimal
```

```
              | 0b[0,1]+  //binary
              | 0o[0-7]+  //octal
              | 0h[0-9,A,a,B,b,C,c,D,d,E,e,F,f]+  //hex

<neg integer> = - <pos integer>

<integer> = <pos integer> | + <pos integer> | <neg integer>

<float> = //FIXME: fill this in

<constant> = <integer>
           | <float>
           //FIXME: maybe we need strings?

<endianness> = big | little

<sign> = signed | unsigned

<array dims> = <pos integer>
             | <pos integer> , <array dims>

<array spec> = [ <array dims> ]

<type spec> = <endianness> <sign> bit <array spec>
            | <endianness> bit <array spec>        //default sign
            | <sign> bit <array spec>              //default endianness
            | bit <array spec>
            | <identifier>                         //user defined type

<type dec> = type <identifier> = <type spec> ;

<variable dec> = <type spec> <identifier> ;
               | <type spec> <identifier> @ <identifier> <array spec>;

<value> = <identifier>                  //variable ref.
        | ( <type spec> ) <constant>    //assigning a size and sign

<infix op> = == | < | > | <= | >= | && | || | ^^ | ~~ | :: | & | | | ^ | ~
           | << | >> | <<< | >>> | + | - | * | / | % | =

<arg list> = EMPTY
           | <argz>

<argz> = <expr>
       | <expr> , <argz>

<function call> = <identifier> ( <arg list> )

<expr> = <value>
       | <expr> <infix op> <expr>
       | #( <expr>, <pos integer>, <expr> )
```

```
        | ##( <expr>, <pos integer>, <expr> )
        | !( <expr>, <pos integer> )
        | !0( <expr>, <pos integer> )
        | $1( <expr> )
        | $0( <expr> )
        | - <expr>
        | <expr> ++
        | <expr> --
        | ( <type spec> ) <expr>
        | <function call>
        //FIXME: deal with delimiters

<class dec> = class <identifier> { <class members> }

<expr list> = EMPTY
            | <expr> ;
            | <expr> ; <expr list>

<function body> = <expr list>

<attribute dec> = <identifier> ( <arg list> ) { <function body> }

<constriant dec> = <identifier> = <expr> ;

<class members> = EMPTY
                | <variable dec> <class members>
                | <attribute dec> <class members>
                | <constraint dec> <class members>
```

5 Formal Semantics

5.1 Formal Array Numbering

In CoGenT, most things are arrays of one kind or another. In order to assign one array to another or operate on two arrays a kind of equivalence must be demonstrated. In CoGenT, for two arrays to be assignable to one another implies that the two arrays are of the same size. Size, as used here, has the specific meaning of length of the array resulting from collapsing the type to a 1-D array.

For a 1-D array collapsing is simple. Its already 1-D and its formal size is its actual size. For multidimensional arrays, the size is more complicated. In general, for a type `bit[`$d_1, d_2, ..., d_n$`]`, the size of the collapsed array is $d_n * d_{n-1} * ... * d_1$.

To convert an informal location specification, `foo[`$x_1, ..., x_n$`]` into a formal location in the collapsed representation of `foo` (`foo`$_c$) requires the following sum: `foo[`$x_1, ..., x_n$`]` $=$ `foo`$_c$`[`$b_1 * \prod_{i=2}^{n} d_i + b_2 * \prod_{i=3}^{n} d_i + ... + b_n$`]`, where $b_j = x_j$ if x_j is big-endian and where $b_j = (d_j - (1 + x_j))$ if x_j is little-endian. Or, more generally: `foo[`$x_1, ..., x_n$`]` $=$ `foo`$_c$`[`$\sum_{j=1}^{n} b_j * \prod_{i=j+1}^{n} d_i$`]` (where $\prod_{i=j+1}^{n} d_i$ for $i \geq n$ is defined to be 1). This is to say that in CoGenT, arrays are considered to be in row-major order.

5.2 Formal Meaning of Assignment

In CoGenT, assignment requires the operands to have a size.