Report Number: WUCSE-2005-47

2005-09-27

X Language Specification

Eric Tyson

Language X provides a formal and intuitive way to describe a series of interconnected processing fiblocks.fl Users of Language X may enter in a logical arrangement of blocks that describes the interconnection of their inputs and outputs. Language X also provides syntax for specifying implementation details for processing blocks and for targeting the entire architecture onto arbitrary sets of devices. Formally, Language X is a structure-only dataflow programming language (DFPL) that is heavily dependent on its library of functions.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2005-47

X Language Specification

Authors: Tyson, Eric

September 27, 2005

Abstract: Language X provides a formal and intuitive way to describe a series of interconnected processing fiblocks.fl Users of Language X may enter in a logical arrangement of blocks that describes the interconnection of their inputs and outputs. Language X also provides syntax for specifying implementation details for processing blocks and for targeting the entire architecture onto arbitrary sets of devices.

Formally, Language X is a structure-only dataflow programming language (DFPL) that is heavily dependent on its library of functions.

Language X Specification Draft

November 12, 2005

Eric Tyson, etyson@wustl.edu
Washington University Storage Based Supercomputing

History:

7/10/05	ejt	Initial language spec. finished, almost-draft status
9/27/05	ejt	Minor revisions to syntax, semantics; draft status Corresponds with X compiler version 0.2
10/30/05	ejt	Added "Constant evaluation" section clarifying the 'delayed' evaluation of constants.
11/9/05	ejt	Corrected float96 datatype (not float128)
11/12/05	ejt	Updated to new and improved split/merge syntax

1 Introduction

1.1 Overview

Language X provides a formal and intuitive way to describe a series of interconnected processing "blocks." Users of *Language X* may enter in a logical arrangement of blocks that describes the interconnection of their inputs and outputs. *Language X* also provides syntax for specifying implementation details for processing blocks and for targeting the entire architecture onto arbitrary sets of devices.

Formally, *Language X* is a structure-only dataflow programming language (DFPL) that is heavily dependent on its library of functions.

Associated with *Language X* is the *X Compiler*. The *X Compiler* is capable of forming compileable and synthesizable code for multiple platforms and devices, using a library of native inter-block and inter-device connection routines.

Language X can also be used in conjunction with the *Auto-Pipe* design flow for pipelined algorithms. Architectures created in *Language X* may be analyzed using *Auto-Pipe* to better understand and optimize the system. Additionally, hardware and software created by the *X Compiler* provide the capability to capture statistical information about executed architectures, which can be used to refine the allocation of devices and further optimize the system.

This document describes the syntax and semantics necessary to create a document compliant with *Language X*. To create processing blocks or to use prefabricated blocks, see additional documentation (e.g. *The Language X C API*, *The Language X VHDL API*).

1.2 Notation

The syntax notation used in section 2 is a form of Extended Backus-Naur Formalism (EBNF). Some additional style changes have been made for clarity. To summarize:

- Symbols, on the left hand side, are defined by their substitution, on the right.
- Symbol replacement choices are delineated by the pipe `|` character.
- Direct strings of *Language X* code are contained within single quotes `‘’`.
- Symbols may be grouped into a single logical symbol with parentheses `()`.
- Optional symbols may be grouped using square brackets `[]`.
- Symbols may be replicated zero or more times when suffixed with an asterisk `*`, or one or more times when suffixed with a plus sign +.
- Non-terminal symbols, excluding those found under the heading “Lexical conventions,” begin with an upper-case letter.
- White space has been ignored to make the notation is this document more readable.

2 Syntax

2.1 Lexical conventions

```
[1-1] char        ::= nextline | digit | letter | somepunctuation
[1-2] comment     ::= ‘//’ (char)* nextline | ‘/*’ (char)* ‘*/’
[1-3] natural     ::= (digit – ‘0’) (digit)*
[1-4] integerC    ::= [‘–’|‘+’] (natural | (‘0’)*)
[1-5] floatC      ::= (digit)* [‘.’] (digit)* [‘e’ (digit)+]
[1-6] stringC     ::= ‘"’ (char)* ‘"’
[1-7] filenameC   ::= ‘"’ (letter|digit|punctuation)* ‘"’
[1-8] identifier  ::= (letter|‘@’) (letter|digit|‘_’|‘@’)*
```

“C++ style” comments are allowed. The characters `//` begin a comment, which is only ended by the end-of-line character. Additionally, the characters `/*` begin a comment which is only ended by the characters `*/`. Comments may be inserted between any two distinct symbols in the input file.

Integer and floating-point constants are permitted in certain circumstances. All values are entered in decimal format unless otherwise specified, with optional exponentiation using the `e` character (e.g., `11.2e2` is equivalent to `1120`). Entering floating-point constants when integers are expected produces undefined behavior.

String constants consist of a series of one-byte ASCII characters, surrounded by the double-quote character `"`. Only printing characters (ASCII values above decimal 31 and below decimal 127) excluding the double quote character are supported, unless otherwise specified.

Identifiers consist of a sequence of letters, numbers, underscores ‘_’, and at ‘@’ characters. An identifier may not begin with a number or underscore character. Identifiers,

keywords, and numerical constants are case-insensitive. String constants may be case-sensitive in some cases, such as filenames. This behavior is specified where relevant.

The following are keywords that may not be used as identifiers:

```
array                   input                   target
block                   output                  typedef
constant                platform                @blockrank
config                  use                     @devicerank
device                  source                  @starttime
impl                    struct                  @unique
```

2.2 Pre-processing directives

The *Language X* parser uses the C Preprocessor (*cpp*) before parsing the file directly. Any directives supported by the native *cpp* are available to the user. Of particular note are the *cpp* statements `#include`, `#if[def]/#else/#endif`, and `#define`. Use of these directives is recommended to improve readability and reduce redundancy in the *Language X* code. *cpp* identifiers may also be passed in on the command line to facilitate automated construction of systems using *Language X*.

2.3 Statements

```
[2-1] XLanguage   ::= (ToplevelS)* UseS (ToplevelS)* EOF
[2-2] ToplevelS   ::= BlockS | ConstantS | DeviceS | PlatformS |
                      TargetS | TypedefS | UseS
```

Language X files are created from the top-level declarations and statements enumerated in grammar [2-2]. Synthesizable X files contain at least one architecture and one `use` statement (see section 0) indicating an architecture to implement. The following sections examine the top-level statements in further detail.

2.3.1 Data type syntax

```
[3-1] DataType    ::= 'array' '<' DataType '>' '[' (natural | '*') ']' |
                      identifier | BasicType |
                      'struct' '<' DataType (',' DataType)* '>'
[3-2] BasicType   ::= 'signed8' |'unsigned8' |'signed16'|'unsigned16'|
                      'signed32'|'unsigned32'|'signed64'|'unsigned64'|
                      'float32' |'float64'   |'float96' |'string'
```

Language X native data types are constructed from a set of basic data types, including floating-point numbers and signed and unsigned integers, each of various bit widths.

Homogenous arrays of data types are constructed by using the `array` keyword. `array` requires a non-negative length to be specified when creating the data type, or `*` indicating variable length. Variable length arrays are of indeterminate length and their handling by the system is platform-specific.

To create heterogeneous data structures (i.e., consisting of multiple data types), the `struct` keyword is used. This data type is provided with an *ordered* and *unnamed* list of types that are contained in the structure.

`array` and `struct` data types may be nested to create arbitrary data types of larger dimension.

```
[3-3] TypedefS    ::= 'typedef' identifier DataType ';'
```

User-named data types are created using the `typedef` statement. The parser does not distinguish these named types from their fully expanded contents.

2.3.2 Configuration statements

```
[4-2] ConstantS   ::= 'constant' DataType identifier '=' Constant ';'
[4-3] Constant    ::= (integerC|floatC|stringC) |
                      '{' Constant (',' Constant)* '}'
```

`constant` creates a named constant of a specified type, and sets it to an integer or float value, or an array of such values. Array constants are surrounded with braces `{}` and their elements delimited with commas. Array constants may be nested (e.g. `{{1,2},{3,4}}`).

```
[4-4] PlatformS   ::= 'platform' stringC [':' stringC] ['{'
                      (ImplPS | ConfigPS)* '}']
[4-5] ImplPS      ::= 'impl' identifier FuncIdent ';'
[4-6] ConfigPS    ::= 'config' identifier ';'
[4-7] FuncIdent   ::= letter (letter | digit | '_')*
                      [ '<' (DataType | Constant)
                       (',' (DataType | Constant))* '>' ]
```

The `platform` statement associates library functions with blocks for use in device implementations (see section 0). The first string constant specifies the name of the platform. The platform can optionally derive the content of another platform by specifying a second string constant.

Specific functions are attached to blocks using the `impl` statement. The block name is specified, followed by the function identifier. Function identifiers are specific to their native language – see implementation-specific documentation for details. The `source` statement indicates which files are searched for these implementations.

`config` works similarly by indicating possible configuration options; these are only meaningful to the library implementations of certain blocks.

2.3.3 Blocks

```
[5-1] BlockS      ::= 'block' identifier '{'
                       (PortBS | ConfigBS | BlockInst | TypedefS |
                        ConstantS | EdgeBS | SplitBS | MergeBS)* '}' ';'
```

Blocks are the abstract processing elements with which *Language X* creates processing architectures. The `block` statement encloses a description of a single block. Describing a

block does not create the block; only an instantiation from a used architecture or subcomponent thereof will do so.

```
[5-2] PortBS      ::= ('input' | 'output' ) DataType identifier ';'
[5-3] ConfigBS    ::= 'config' DataType identifier ['=' ConstArg] ';'
[5-4] ConstArg    ::= Constant | identifier ('[' natural ']')*
```

Within a `block` statement are port declarations containing the type and name of input and output ports, and one-time configuration inputs. These are indicated by the `input`, `output`, and `config` keywords, respectively. Each declaration is followed by the expected data type, and a unique identifier. `config` ports may optionally include a default value, provided by a constant value or a named constant with optional indexing.

```
[5-5] BlockInst   ::= identifier BlockIdent [BlockOpts]
                      (',' BlockIdent [BlockOpts])* ';'
[5-6] BlockOpts   ::= '(' identifier '=' ConstArg
                      (',' identifier '=' ConstArg)* ')'
```

Blocks may also contain other blocks by declaring an instance using the block's identifier and an instance name. Configuration may be provided by following the identifier with arguments in the form `SomeBlock(config1=constant1, conf2=const2)`. Same-type blocks with different names may be declared by separating the identifiers with commas. Same-type arrays of *n* blocks may be declared by immediately following the identifier with `[n]`.

```
[5-7] EdgeBS      ::= (Port '->' (DefaultEdge '->')* Port) ';'
[5-8] DefaultEdge ::= DefaultPort ('->' DefaultPort )*
[5-9] Port        ::= DefaultPort| BlockIdent '.' identifier | identifier
[5-10]DefaultPort ::= BlockIdent
[5-11]BlockIdent  ::= identifier ['[' natural ']']
[5-12]SplitBS     ::= (DefaultEdge '->')* Port '=<' CompoundPort ';'
[5-13]MergeBS     ::= CompoundPort '>=' (DefaultEdge '->')* Port ';'
[5-14]CompoundPort::= Port | '{' Port (',' Port)* '}'
```

Edges may be created between subcomponent blocks after all block declarations have been made. The most explicit form for an edge is of the form:

```
blockA.outportX -> blockB.inportY;
```

However, for convenience and readability, "default" ports may be used when the input or output is unambiguous. For example, if the only output of `blockA` is `outportX`, and the only input of `blockB` is `inportY`, then the above example can be simplified to:

```
blockA -> blockB;
```

Additionally, unambiguous strings of blocks may be connected in one statement by using their default ports. Note that this is only possible if all interior blocks in the statement satisfy the requirement of having only one input and one output. Referring again to the above example, if `blockA` also has only one input and `blockB` has only one output, then a legal string would be:

```
blockC.outportW -> blockA -> blockB -> blockD.inportZ;
```

The =< split operator allows `array` and `struct` data types to be divided into parts and distributed to multiple ports. A port list, usually a list of ports surrounded by braces `{}`

and separated by commas, is provided in place of a single port for the destination. The number and order of elements in the port list must completely agree with the data type being split.

The `>=` merge operator is similar to split, but in the opposite direction. A port list is merged into a single port of the compound data type. If the merged port data types are different, then a `struct` will be formed. If the types are the same, then either a `struct` or `array` will be formed, depending on the destination data type. As with split, the number order of the data types must completely agree.

Block array members must be referred to using an index subscript (i.e. `[n]`), unless they are used in a split or merge operation. In the case of a split, a non-indexed array block identifier may be used as the destination block. Similarly, in the case of a merge, a non-indexed array block identifier may be used as the source block. For instance, if `E` is an array of five blocks with a scalar data type output `y`, and `F` is a single block with one input that is an array of five elements of the same data type, then the following statements perform the same operation:

```
{ E[1].y, E[2].y, E[3].y, E[4].y, E[5].y } >= F;
E.y >= F;
```

Inputs and outputs declared by the block may be used as output ports and input ports, respectively.

See section 3.7 for more details on the behavior of split and merge.

2.3.4 Generation statements

```
[7-1] UseS        ::= 'use architecture' BlockInst [identifier] ';'
```

The `use architecture` statement is used to indicate all block hierarchies that are to be actually synthesized by the code generator. Only a single instance of the highest-level block containing all subcomponents should be used to create a single architecture. Multiple architectures may be used; see section 3.4. An optional name may be given to the architecture; otherwise, the name will be the same as the instantiated block's.

```
[7-2] DeviceS     ::= 'device' DeviceIdent ':' stringC
                      [ DevConfig | '{' DevConfig (',' DevConfig)* '}']
                      ';'
[7-3] DevConfig   ::= '(' (identifier '=' ConstArg ',')* ')'
[7-4] DeviceIdent ::= identifier ['[' natural ']']
```

Actual devices available to the code generator are specified using the `device` statement. This statement identifies a name for the device or devices, and the type of device (from `platform`, syntax 4-4). In this statement, the user may also provide configuration information. For scalar (non-array) devices, only a single configuration of the form `( configA = valueA, configB = valueB, ... )` is accepted. Array devices may be configured identically using a single configuration, or differently by nesting configurations within an additional pair of enclosing braces. An empty `{ }` configuration is also accepted in all cases. Device configuration must specify all configuration items from `platform` that lack a default value.

```
[7-5] TargetS     ::= 'target' DeviceIdent '='
                        '{' [FullIdent (',' FullIdent)* ] '}' ';'
[7-6] FullIdent   ::= (BlockIdent '.')* BlockIdent
```

Blocks in the used architectures are bound to devices using the `target` statement. Using `target`, a device identifier is connected to one block or a set of blocks. Blocks are identified as a fully specified child of an architecture identifier provided by the `use` statement. Multiple `target` statements for the same device identifier are permitted and will be merged. If a non-terminal block (i.e., a block containing other blocks) is specified, then all contained blocks will be assigned to that device as well, unless overridden by a later `target` statement.

3 Behavior

3.1 Type checking

Language X is strongly, statically typed with respect to the data types of blocks' ports. Types are checked during compilation and may not necessarily be re-checked when synthesizing block implementations in the native languages.

Any named types (corresponding to `typedef` statements) with the same fully expanded type may be used interchangeably. For instance, in following example, types `T` and `U` are identical:

```
typedef array<unsigned8>[8] S;
typedef array<S>[4] T;
typedef array<array<unsigned8>[8] >[4] U;
```

However, simply having the same number of elements of the same type will not guarantee type compatibility. In the below statement, the type `V` is not compatible with either type `T` or `U`, above, even though they all contain 32 `unsigned8` elements:

```
typedef array<array<unsigned8>[4] >[8] V;
```

An exception to the type checking is allowed in the case of variable-length array outputs connected to static-length array inputs. An output port of type `array<T>[`*n*`]` may be connected to an input port of type `array<T>[*]`, but the reverse is not permitted.

3.2 Declaration ordering

Similar to other languages, most identifiers must be declared and/or defined before they are used. add: typedefs, blocks, etc.

3.3 Special constants

The following constants are known to the *Language X* parser and will be filled in at or before runtime:

`@BUILDTIME` (unsigned32)

This takes the value of the UNIX timestamp (seconds since the beginning of Jan. 1, 1970) on the build machine at some time during the parsing of the X Language file.

`@BLOCKRANK` (unsigned16)

This takes the value of the array index to the block in which this constant is referenced. For instance, if there is an array of 32 `MyBlock` blocks, then `@BLOCKRANK` will be assigned numbers from 1 to 32 whenever it is used from *within* the MyBlock block. If there is no array index available, `@BLOCKRANK` takes the value 0 (zero).

`@DEVICERANK` (unsigned16)

This takes the value of the array index to the device to which the referring block is targeted. For instance, if `MyBlock[3]` is targeted to device `PC[1]`, then any use of `@DEVICERANK` within `MyBlock[3]` will be given the value 1. If there is no array index available, `@DEVICERANK` takes the value 0 (zero).

`@UNIQUE` (unsigned32)

This constant is replaced with a unique value for every instance in which it is referenced (up to 2^{32} references).

3.4 Evaluation of constants

To increase the flexibility of the *Language X* and enable more complex structures to be created, all constants are evaluated after the initial parsing of the language file. This includes but is not limited to the special constants in section 3.3, block and constant array indices, and `CONSTANT` assignments.

The utility of this is apparent in the block instantiation expression:

```
CONSTANT ARRAY<UNSIGNED8>[4] c_array = {5, 3, 1, 4, 2};
Block myBlock[4] (c = c_array[@BLOCKRANK]);
```

where `myBlock[1]` will be configured with `c=5`, `myBlock[2]` with `c=3`, and so on.

3.5 Multiple architectures

Multiple architectures may be created through multiple `use` statements. Identical statements will create multiple distinct architectures. As with a single `use` statement, additional resources will be consumed for each additional architecture.

3.6 Stacking block bindings

An important feature in the behavior of the `platform` statement is that implementation bindings for a specific platform may be distributed across any number of equally identified `platform` statements. For instance, if a function for a new block Foo has been written for platform Bar, then a new `platform "Bar" { impl Foo … }` binding may be written without modifying any earlier `platform "Bar" { … }` blocks.

3.7 Split and merge

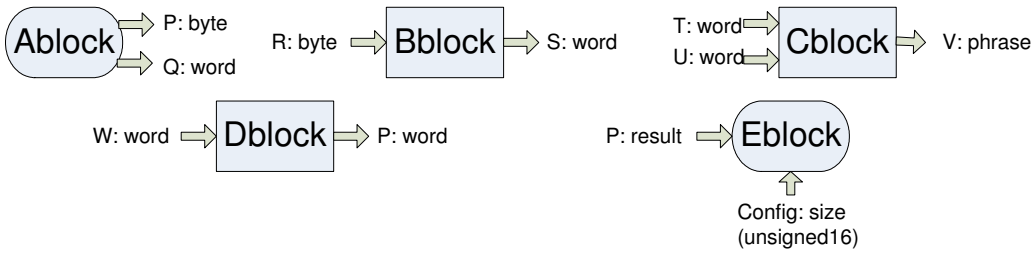
The split operator =< (syntax 5-12) and merge operator >= (syntax 5-13) are used to trivially separate and collate compound data types. Splits allow an edge to connect an output of type `array<`*T*`>[`*n*`]` on a source block to an input of type *T* on an array of *n* blocks. Similarly, merges allow outputs of type *T* on an array of *n* blocks to connect to a

single input of type `array<T>[n]`. When a split or merge operate on a `struct<>` data type, the semantics are the same over the heterogeneous data types.

The behavior of splits and merges on edges is intended to be as transparent as possible. Once data is transmitted on the source of the edge, it is to be ready on the destination of the edge without preference to any single element, in the same manner as an equal number of simple one-to-one edges. Often, a more complex method of distributing compound data types to multiple destinations is necessary. In these cases, it is necessary to create a block implementation to perform the more complicated operation.

4 Example

4.1 Block declarations

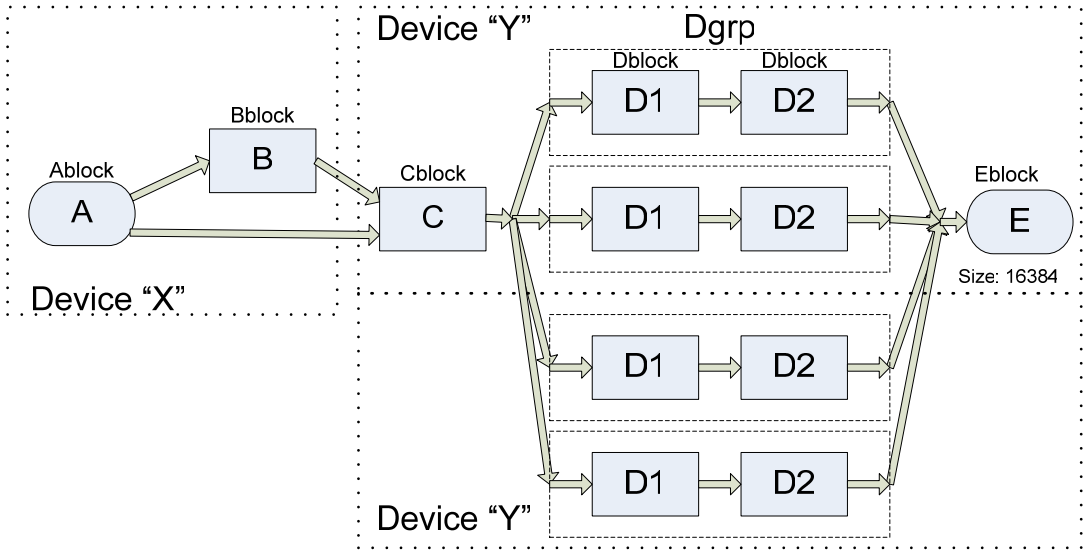



```
// byte, word are defined for convenience
typedef UNSIGNED8 byte, SIGNED16 word;
// two array objects are defined as arrays of words
typedef ARRAY<word>[4] phrase, ARRAY<word>[4] result;


block Ablock {
   output UNSIGNED8 p;       // these can be connected to byte and
   output SIGNED16 q;        // word, respectively
};
block Bblock {
   input byte r;
   output word s;
};
block Cblock {
   input word t;
   input word u;
   output phrase v;
};
block Dblock {
   input word w;
   output word p;
};
block Eblock {
   input result p;
   config UNSIGNED16 size;   // config constants passed in arch
};

platform "C_MPI" {           // append Bar to "C_MPI" platform functions
   impl Bblock Bar<byte,word>; // templates may pass datatypes
};
platform "X" : "C_MPI" {     // "X" derives "C_MPI" functionality
   config debug;
   impl Ablock Foo;
};
platform "Y" {
   config address;
   impl Cblock BAZ_8x8;
   impl Dblock BLIP<1,2,3.4>;
   impl Eblock FLIM<1.0e-14>;
};
```

4.2 Implementation




```
block Dgrp {            // define a composite block with two Dblocks
   input word in;       // composite blocks have inputs and outputs
   output word out;
   Dblock D1, D2;
   in -> D1 -> D2 -> out;    // The default ports are used here
};

block Flow_Arch {       // this is a main architecture block
   Ablock A;
   Bblock B;
   Cblock C;
   Dgrp DD[4];          // an ordered array of components
   Eblock E(size=16384);     // 16384 passed in as configuration to E

   A.p -> B -> C.t;     // unambiguous connections may be grouped
   A.q -> C.u;
   C.v =< DD;           // divide the phrase type into 4 words
   DD  >= E;            // DD refers to the whole array
};

use architecture Flow_Arch flow;

device xdev    : "X" { debug = "true "; };    // a single "X" device
device ydev[2] : "Y" { // a set of "Y" devices
   { address = "0xabc00000"; },    // configuration; apply to ydev[1]
   { address = "0xabc10000"; }     // this applies to ydev[2]
};

/* Blocks are allocated to each device.  DD is split up into two
 * groups of two, specifically selected using DD[n] */
target xdev    = { flow.A, flow.B };
// flow.DD[1] target uses the highest-level blocks, flow.DD[1].D1 and D2
target ydev[1] = { flow.C, flow.DD[1], flow.DD[2], flow.E };
target ydev[2] = { flow.DD[3], flow.DD[4] };
```