Many-Core Levels Language Descriptions

Pieter Hijma

VU University Amsterdam
Department of Computer Science
pieter@cs.vu.nl

September 2014

1 Introduction

This document is an extension of [1] and discusses more details of the syntax and semantics of the two languages of Many-Core Levels (MCL). Section 2 discusses the hardware description language and is an extension of Sec. 4.2 of [1]. Section 3 extends Sec. 4.3 of [1] and provides the syntax of the programming language.

In the syntax descriptions we use EBNF in a slightly modified form for readability. A capitalized identifier is a non-terminal, = defines a rule of the syntax, | an option and concatenation is implied using whitespace. A string enclosed in "" is a literal in the syntax and we will use three suffixes for terms: ? means zero or one ([ ] in EBNF), * means zero or more ({ } in EBNF), and + means one or more. We will use ( ) for grouping, and { } for interspersing terms: for example, to allow a term A interspersed with B zero or more times, allowing A, A B A or A B A B A, etc., we can use {A B}*, which is equivalent to $\lambda | A(BA)*$ where λ is the empty string.

2 Hardware Description Language HDL

A hardware description is defined in a .hdl file, separate from .mcl files which contain programming modules. The top-level structure of HDL is very simple:

```
HWDescription    =   "hardware_description" Identifier
                     Specializes?
                     Block*
Specializes      =   "specializes" Identifier ";"
```

The start symbol of HDL is HWDescription which has the keyword hardware_description. In HDL each keyword is reserved. The Identifier indicates which hardware description it represents. Following this declaration, a hardware description has an optional Specializes clause which specifies on which hardware description the current hardware description specializes. Finally, a hardware description has zero or more Blocks.

The Block is the main construct in HDL. Its semantics are determined by the specific Block-Keyword with which a Block is declared. The syntax is listed below:

```
Block           =  BlockKeyword Identifier "{" Statement* "}"
BlockKeyword    =  "parallelism"   |  "memory_space"   |  "par_unit"
                |  "par_group"   |  "device"   |  "memory"   |  "interconnect"
                |  "device_group"   |  "device_unit"   |  "execution_group"
                |  "execution_unit"   |  "instructions"   |  "cache"
                |  "simd_group"   |  "simd_unit"   |  "load_store_group"
                |  "load_store_unit"
```

A Statement can have several forms, listed below:

```
Statement           =  Block
                    |  PropertyStatement
PropertyStatement   =  PropertyKeyword "=" Expression PrefixUnit? ";"
                    |  Expression ";"
                    |  ArgPropKeyword "(" {Expression ","}+ ")" ";"
                    |  StatementKeyword ";"
```

The first production rule shows that Blocks can be either a nested Block or a PropertyStatement. A PropertyStatement has the following forms: some property that can be assigned expressions with optionally a PrefixUnit, expressions can be used as statements, there are properties that take expressions as arguments, and finally, there are statements that are just keywords. The keywords that can be used in statements are listed below:

```
PropertyKeyword    =  "nr_units"   |  "max_nr_units"   |  "capacity"   |  "latency"
                   |  "bandwidth"   |  "nr_banks"   |  "clock_frequency"
                   |  "addressable"   |  "cache_line_size"   |  "width"
ArgPropKeyword     =  "slots"   |  "connects"   |  "space"   |  "op"
                   |  "performance_feedback"
StatementKeyword   =  "default" | "read_only"
```

A property can be assigned an expression having an optional PrefixUnit. This allows us to define units for some expressions, for example GB/s:

```
PrefixUnit   =  Prefix? Unit
Prefix       =  "G" | "M" | "k"
Unit         =  BasicUnit "/" BasicUnit
             |  BasicUnit
BasicUnit    =  "B" | "bit" | "bits" | "cycle" | "cycles" | "s" | "Hz"
```

The grammar for an expression is:

```
Expression   =  IntExp
             |  QualIdentifier "[*]"?
             |  QualIdentifier "(" {Expression ","}+ ")"
             |  Operation
             |  ExpressionKeyword
```

An Expression can be an integer expression IntExp (not further shown in this grammar) with the usual operations +, -, etc. defined on them. An Expression can also be a qualified Identifier with optionally a "[*]" suffix, or with Expressions as arguments (line 3). A QualIdentifier uses dot notation to indicate identify properties that are nested:

```
QualIdentifier   =   Identifier
                 |   QualIdentifier "." Identifier
```

Finally, an expression can be an operation or an expression keyword:

```
Operation           =   "(+)"  |  "(-)"  |  "(/)"  |  "(%)"  |  "(*)"
                    |   StringExpression
ExpressionKeyword   =   "unlimited"  |  "true"  |  "false"
```

The syntax of an operation allows us to define some properties about the operations +, -, etc. in the programming language. An operation can also be a string (not further explained in this grammar), for example to indicate something about special hardware functionality such as special units that support transcendental functions.

Hardware descriptions define their parents using the specializes declaration which results in a hierarchy (an example is shown in Fig. 4 in [1]). The edges of the hierarchy do not mean that a hardware description on a lower level inherits features of its parent. It means that a program written for hardware description x can be translated to a program adhering to the rules of hardware description y where y is a child of x. For example, it is possible to translate a program written for hardware description *perfect* to a program for hardware description *fermi* in Fig. 4 in [1]. It is possible to reuse parts of other hardware descriptions by using direct references to the blocks in other hardware descriptions.

A valid hardware description with the name *name* has at least two blocks, a parallelism block and a device block with *name* as Identifier. The parallelism block defines a hierarchy of programming abstractions to which programmers map their algorithm, and the device block describes the physical device.

The nesting of Blocks is governed by rules which are summarized in Table 1. The blocks in the second column are allowed to be nested inside the blocks in the first column. The third column specifies whether such a block is required and the fourth column specifies whether more than one of these blocks are allowed.

The first section “Common” shows the rule for grouping blocks. A block ending with _group specifies that a certain number of _unit blocks are grouped together. The _unit blocks are nested in the _group blocks and must have the same prefix. The second and third section show the rules for the blocks for programming abstractions and the physical device respectively.

Inside the blocks, several properties can be specified. Table 2 shows which properties can be used where and what kinds of expressions and units are allowed. The second column specifies the property that can be specified in the block in the first column. The third and fourth column indicate which expressions and units are allowed. The last column indicates whether the property is required inside the block.

A _group block always needs to have a max_nr_units or nr_units with an integer expression or the keyword unlimited, which means that it may be a very large integer number. The sections in this table are the same as in Table 1. In section “Physical Device” the expression (inherited) means that the rules for the properties are inherited from the device parent. The relation between device blocks has been specified in Fig. 5 in [1].

Table 1: Allowed nestings of Blocks

Block	Block	required	multiple
Common			
x_group	x_unit	yes	no
Programming abstractions			
parallelism	par_group	yes	no
	memory_space	yes	yes
par_unit	memory_space	no	yes
	par_group	no	no
Physical device			
device	memory	no	yes
	cache	no	yes
	interconnect	no	yes
	execution_group	no	yes
	device_group	no	yes
execution_unit	memory	no	yes
	cache	no	yes
	execution_group	no	yes
	load_store_group	no	yes
	simd_group	no	yes
	instructions	no	no
device_unit	memory	no	yes
	cache	no	yes
	interconnect	no	yes
	execution_group	no	yes
	device_group	no	yes
load_store_unit	instructions	yes	no
simd_unit	instructions	yes	no

Table 2: Allowed properties of Blocks

Block	property	expression	unit	required
		Common		
*x*_group	(max_)nr_units	IntExp or countable		yes
		Programming abstractions		
memory_space	default			no
	read_only			no
		Physical device		
memory	capacity	IntExp or countable	B, bit, bits	no
	space	Identifier		yes
	nr_banks	IntExp		no
	addressable	false		no
cache	(inherited)			
	cache_line_size	IntExp	B, bit, bits	yes
interconnect	connects	Identifier		yes
	latency	IntExp	cycle(s)	yes
	bandwidth	IntExp	{B, bit, bits}/s	no
	width	IntExp	bit, bits	no
	clock_frequency	IntExp	Hz	no
execution_group	slots	Identifier, IntExp		yes
execution_unit	slots	Identifier, IntExp		yes
	performance_feedback	String		no
simd_unit	(inherited)			
load_store_unit	(inherited)			

3 Programming Language

Since many syntactic forms are similar to C, we will only discuss what is different from C. Please note that below, we define a different grammar than the grammar in Sec. 2 that is used in different files. We can therefore safely reuse the names for non-terminals, such as Statement and Block denoting different syntactic forms.

An MCPL module consists of a "module" keyword with an Identifier declaring which module it is, a list of Imports and a list of Functions. An import declares which hardware descriptions should be imported into the module.

```
Module   =   "module" Identifier
             Import*
             Function*
Import   =   "import" Identifier ";"
```

A Function starts with an Identifier that indicates which hardware description this function targets. This means that we can specify per function which hardware we target. It then specifies the return type, an Identifier that indicates the name of the function, and then a list of Declarations for parameter list and a Block:

```
Function   =   Identifier Type Identifier "(" {Declaration ","}* ")" Block
```

A Type is a primitive type or a Type with an ArrayExpression. This results in a multi-dimensional tiled array type. The syntax is listed below.

```
Type              =  "int"  |  "float"  |  "bool"  |  "void"
                  |  Type ArrayExpression
ArrayExpression   =  "[" {Expression ","}+ "]"
```

Having tiled multi-dimensional arrays means that we can have the following forms: The type int[2][3] has 2 tiles of 3 elements. The type int[2, 3] is a two-dimensional array with 2 rows and 3 columns. The type int[2,2][3,3] is a two-dimensional array with 2×2 tiles of 3×3 elements each.

A Declaration has two forms. It can have BasicDeclarations interspersed with an as keyword or it can be an assignment Declaration.

```
Declaration        =  Modifier* {BasicDeclaration "as"}+
                   |  Modifier* BasicDeclaration "=" Expression
BasicDeclaration   =  Type Identifier
Modifier           =  "const"
                   |  Identifier
```

With the as keyword one can declare variables with more than one form, for example: int[2, 3] a as int[6] b. A Modifier indicates whether the variable is constant or it can be an Identifier that has to refer to a memory space in the hardware description of the function.

A Block contains a list of Statements:

```
Block      =  "{" Statement* "}"
Statement  =  Block  |  Declaration ";"  |  Assignment ";"  |  Increment ";"
           |  Call ";"  |  Return ";"  |  If  |  For  |  As ";"  |  Barrier ";"
           |  ForEach
```

A Statement can be a Block and the Declaration that was defined above. The subsequent syntactic forms are not discussed, since they are very similar to any C-like language, except for the last three: As, Barrier, and ForEach.

An As statement is equivalent to a Declaration with keyword as. It declares new forms of a declaration, only not at the declaration itself but at a later point. The syntactic form is:

```
As              =  Variable "as" {BasicDeclaration "as"}+
Variable        =  BasicVariable ("." Variable)?
BasicVariable   =  Identifier ArrayExpression*
```

A Variable can be interspersed with dots and a BasicVariable has zero or more array expressions.

A Barrier statement creates a synchronization point for units of parallelism for a specific memory space. The syntax is:

```
Barrier  =  "barrier" "(" Identifier ")"
```

The Identifier has to refer to a memory space in the hardware description.

In MCPL, the ForEach statement expresses parallelism for a specific parallelism_group. The syntax is:

```
ForEach  =  "foreach" "(" BasicDeclaration "in" Expression Identifier ")" Statement
```

Considering the ForEach as a loop, then the BasicDeclaration declares a variable that will hold a unique value from 0 to Expression for each loop iteration. The Identifier has to refer to a par_group in the hardware description.

References

[1] P. Hijma, R.V. van Nieuwpoort, C.J.H. Jacobs, and H.E. Bal. Under Review. 2014.