# Altera XML Architecture Description File Detailed Design

Version 1.4

May 27, 2005

By

Altera Corporation

# TABLE OF CONTENTS

# 1. Overview

The Altera® XML Architecture Description is a communications standard for describing complex chip architectures to 3rd party users in a comprehensive and digestible format. This document is intended for external use. This document describes in explicit detail the format of the Altera XML Architecture Description file and fills in any gaps in the format that the XML Schema does not explicitly cover. It is intended as a guide for both the future maintenance of the specification and for the third party programmer attempting use the information contained in an XML file that meets the Altera XML Architecture Description specification.

The XML format is a flexible, platform-independent method for sharing information. The Altera XML Architecture Description further constrains this format to provide a flexible and understandable method for sharing Altera architecture information. The standard is set forth in an XML Schema and files validated against this schema are said to be valid Altera XML Architecture Descriptions. A myriad of XML parsers are available for most languages to allow quick and easy digestion of the information in the Altera XML Architecture Description.

# 2. Executive Summary

This document has the following purposes:

- Describe the file formats used to describe Altera devices;
- Explain each part of the file with examples;
- Give an overview of why this file is good;
- Give a reference parser link;
- Give various XML links;
- Provide an example of how to use the XML file.

This file does not:

- Give intra-cell delay file information;
- Give interconnect estimator file information;
- Give an overview of why this information is relevant.

After reading this file and referenced material you should be able to do the following:

- Understand the raw format of an Altera XML Architecture Description file;
- Use the reference parser, or another parser, to read in an XML file;
- Write queries that allow you determine information about any Altera Stratix™ device. Some examples of queries would be:
    - How many LABS are there in a specific device?
    - Where are all the DSP blocks located in a specific device?
    - What is the physical extent of a device?

- Understand the benefits of the XML format.

# 3. Glossary

**XML Document** – A file based on the W3C's eXtensible Markup Language format. All XML documents described in this detailed design document conform to the W3C XML 1.0 standard. For more information on XML see: http://www.xml.com/axml/testaxml.htm

**XML Schema** – Templates for XML documents; they allow you to specify valid arrangements of elements and attributes within an XML document and impose restrictions on their contents. An XML document that is valid according to an XML schema does not break any of the rules laid down in the XML schema.

**Element** – An XML container formed with a start tag (<START_TAG>) and an end tag (</END_TAG>). The start tag can contain some number of attributes (<START_TAG att1=val1 att2=val2>) according the schema for the document. Data that falls between the start and end tags, be it additional elements or text, is said to be contained by the element. This allows for a parent-child tree structure in XML data to be formed. If an element contains no children the start/end tag notation can be abbreviated in to a form known as an empty tag (<START_TAG att1=val1 att2=val2 />).

**Block** – The elements upon which a chip is built. They are hierarchical representations of basic pieces of the architecture. For example: a logic element block, or LE; a LAB block, which contains a number of LE blocks.

**Architecture** – A top-level container for devices. The architecture describes common blocks that are then instantiated in devices of various sizes and configurations.

**Device** – A group of blocks instantiated in a particular configuration.

# 4. The Philosophy of Describing Programmable Logic Devices

Describing the physical implementation of a programmable logic device is not a trivial task. There is a desire to be as verbose as possible; to make details and periphery information available throughout the description. Yet terseness is also desirable; avoiding repetition and grouping together common traits whenever possible improves clarity. The Altera approach to describing a physical implementation of a programmable logic device attempts to find the happy median between compact notation and detailed information by grouping common descriptive elements in a hierarchy.

## 4.1 The hierarchy of a programmable logic device

The basic hierarchy used to describe programmable logic devices conforming to the Altera XML Architecture Description is that of a top-level architecture which encompasses a variety of blocks and devices. The blocks represent detailed descriptions of various components that can be instantiated in conjunction with one another to produce a physical device. A more detailed description of each component is given below.

### 4.1.1 Architectures

At the top of the hierarchy is the notion of an architecture. It contains a common set of building blocks that can be assembled together in many different ways to form different physical devices. These devices are also a part of the top-level architecture.

An architecture is completely self-contained. Any block or device within the hierarchy of the architecture uses only other blocks in the architecture to build its physical implementation. A new, completely contained, architecture must be built if a device that uses blocks in two different existing architecture hierarchies is to be built; a device cannot belong to more than one architecture.

### 4.1.2 Blocks

At the block level, information is divided into global data that affects any block of this type and sub-type information that allows for variance in the physical implementations of a block. This gives a functional-like grouping to blocks. I/O blocks, for example, are one block type, but there are several I/O block sub-types for particular I/O methods. Each sub-type may possibly have a different physical size or number of ports from another sub-type, but the basic function – in the case of our example providing input and output methods to a device – remains the same. Sub-types can also define a varying number of instantiated sub-blocks. A larger block can be built out of several smaller blocks. Blocks are typically a contained block or a container block – there are currently no architectures which have a block that is both a contained block and a container. Note that while a block itself is not an actual instance, and has no location associated with it, sub-blocks represent actual instances of blocks and have a fixed location relative to the parent block. More details about specifying locations are provided in the following sections.

### 4.1.3 Devices

The device level encapsulates several possible packaging types, pin counts and speed grades and one possible layout of blocks. Recall that Blocks in our hierarchy are simply descriptions of components used to create a device within an architecture. A device can be created by instantiating blocks and assigning them to particular locations. Blocks are positioned on a two dimensional Cartesian co-ordinate system and referenced with three co-ordinates: an (X,Y) location and a zero-indexed sub-location. More information on the coordinate system used for block placement can be found in Section 4.2.

This hierarchy of information allows for an architecture to contain a complete description of itself: all the blocks that make up the architecture and all the devices that use the blocks in the architecture. Detailed block information is contained at the architecture level. Blocks are then instantiated in a device with a terse notation that describes only the block type, sub-type and location on the device. This representation of a programmable logic device marries detailed amounts of information in a compact format.

## 4.2 Positioning blocks in space to form a device

As mentioned above blocks are positioned in a device on a two dimensional Cartesian coordinate system using three numbers: (X,Y) coordinates and a zero-valued index coordinate. The (X,Y) coordinates provide the location of the block in a two dimensional grid that overlays the device. The origin of the coordinate system is always the bottom left corner of the device. If a block contains sub-blocks their physical locations are noted relative to the bottom left corner of the block. The sub-location coordinate is used to distinguish between multiple sub-blocks that occupy the same (X,Y) co-ordinate. Figure 4-1 shows the coordinate system in use.

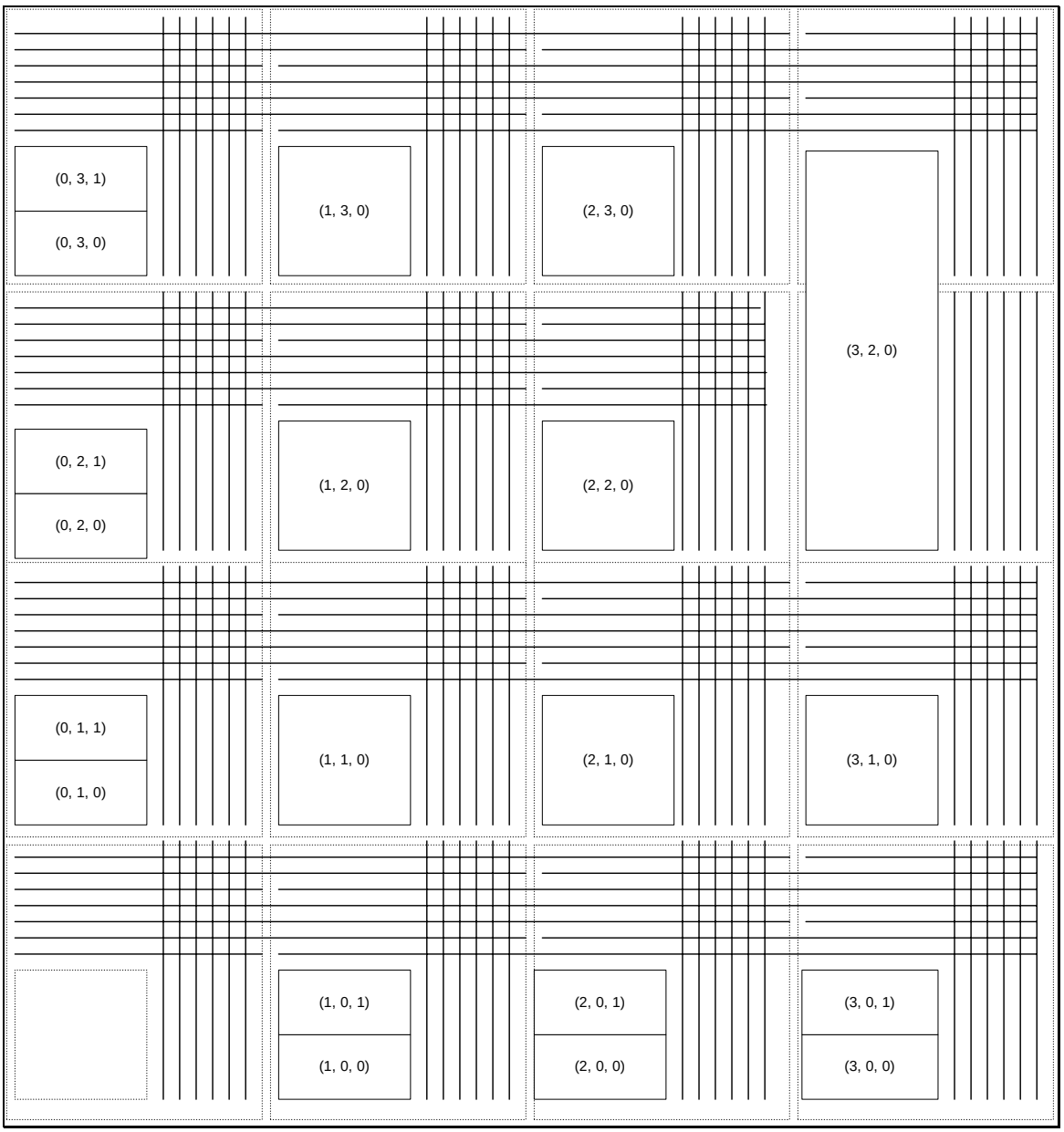


**Figure 4-1: A diagram of a device showing the placement of different blocks and sub-blocks in the Cartesian coordinate system of the device.**

The sixteen dashed-line boxes represent the physical space of a coordinate in the grid. Blocks and sub-blocks placed in a dashed-line box must share the same (X,Y) coordinate of the dashed line box, and they must have unique, contiguous sub-location indices.

Figure 4-1 also illustrates the notion of block extent. Extent of a block indicates how many coordinates that each instance of this block occupies. The block in the upper right corner of the device covers two vertical and one horizontal coordinate position in the grid. The block's location is still noted with its bottom left corner (3,2,0) but the vertical or y-extent of the block would be 2 and the x-extent would be 1 in the physical description of this block. The upper right hand corner of the block is at (3,3,0).

# 5. The Altera XML Architecture Schema

The Altera XML Architecture schema describes the format for any XML-based architecture description. A tree-view of this schema is seen in . The <PORT>, <SUB_BLOCK> and <BLOCK_INSTANCE> elements have not been expanded in the diagram for clarity. For more information on each specific element in the diagram please see Section 6.

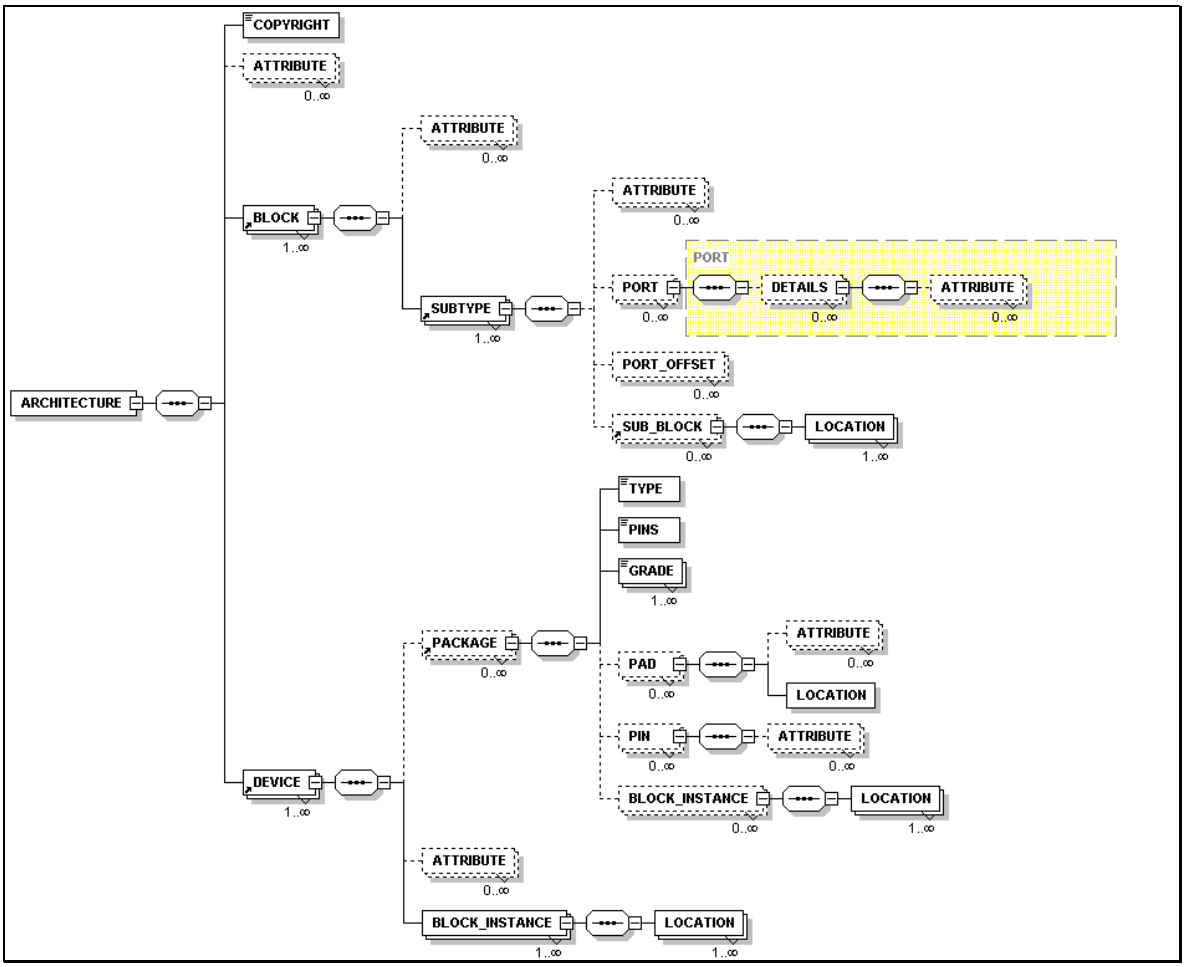


**Figure 5-1: Tree View of the Altera XML Architecture Schema**

The top-level container, or root element, is the <ARCHITECTURE> which describes global <BLOCK>s that are then instantiated at the next level in the specific <DEVICE> sections of the file. <BLOCK>s described in the <ARCHITECTURE> section, in addition to having a name, have a <SUBTYPE> that allows a general block to take on several different physical implementations and instantiate <SUB_BLOCK>s. An XML file must be validated against this schema to be called an Altera XML Architecture Description.

The dashed lines/boxes in Figure 5-1 represent optional elements. The numbers of times an element can occur are listed directly below the element boxes. Order is important in XML. For example, you can have from one to an infinite number of <BLOCK> elements inside an <ARCHITECTURE> element, but they must all occur before any <DEVICE> element in the <ARCHITECTURE> element and after any <ATTRIBUTE> elements of the <ARCHITECTURE>. Once you see a <DEVICE> element, based on this schema, you can assume you will never see a <BLOCK> element again, or an <ATTRIBUTE> element associated with the architecture. Solid lines/boxes represent elements that must occur at least once. According to Figure 5-1 an <ARCHITECTURE> element must have at least one <BLOCK> element, and a <BLOCK> element

must have at least one <SUBTYPE> element. The <SUBTYPE> elements however can have any number of the child elements shown, including zero.

# 6. Detailed Altera XML Architecture File Element Information

This section has detailed information for every possible element that can exist, according to the schema, in an Altera XML Architecture File. It can be used as a reference when creating as well as parsing an XML file that is validated against this schema. A list of elements and their attributes can be found in Table 1. Detailed descriptions of each element can be found in the subsections below or by clicking on the Element tag in the table below.

**Table 1: Element and attributes found in an Altera XML Architecture Description File**

| Element | Attribute | Type | Use | Description |
|---|---|---|---|---|
| <ARCHITECTURE> | version | string | required | The version of the architecture file. |
| | name | string | required | The name of the architecture. |
| <COPYRIGHT> | *Has no attributes* | | | |
| <DEVICE> | name | string | required | The name of the device. |
| | blk_list_id | string | required | Internal ID |
| | pin_table_version | float | required | Internal version number |
| <ATTRIBUTE> | name | string | required | The key in the (key, value) pair. |
| | value | string | required | The value in the (key, value) pair. |
| <BLOCK> | type | string | required | The unique string that identifies the type of block being described. |
| <BLOCK_INSTANCE> | type | string | required | The type of block being instantiated. |
| | subtype | string | optional | The subtype of the block being instantiated. If omitted DEFAULT is assumed. |
| <DETAILS> | *Has no attributes* | | | |
| <LOCATION> | x | integer | optional | The horizontal position of the coordinate. |
| | y | integer | optional | The vertical position of the coordinate. |
| | subloc | integer | optional | The sub-location index of the coordinate. |

| | | | | |
|---|---|---|---|---|
| | name | string | optional | The friendly name of this location. |
| <GRADE> | *Has no attributes* | | | |
| <PACKAGE> | *Has no attributes* | | | |
| <PAD> | id | integer | required | Integer ID of the PAD |
| | name | string | required | String name of the PAD. |
| <PIN> | id | integer | required | Integer ID of the PIN. |
| | pads | string | required | Comma-separated list of PAD IDs to which this PIN is bonded. |
| | name | string | required | String name of the PIN. |
| <PINS> | *Has no attributes* | | | |
| <PORT> | name | string | required | A unique name identifying the port. |
| | type | string | required | A restricted string that can take on one of the values: INPUT or OUTPUT. |
| | width | integer | optional | The width of the port in bits. The default value is 1 if this attribute is omitted. |
| <PORTOFFSET> | name | string | required | The name of the port being offset. |
| | x | integer | optional | The horizontal offset of the port. The default is 0 if this attribute is omitted. |
| | y | integer | optional | The vertical offset of the port. The default is 0 if this attribute is omitted. |
| <SUBTYPE> | name | string | required | The unique name for this sub-type instance. |
| <SUB_BLOCK> | type | string | required | The type of sub-block. |
| | subtype | string | optional | The sub-type of the sub-block. |
| <TYPE> | *Has no attributes* | | | |

## 6.1 <ARCHITECTURE>

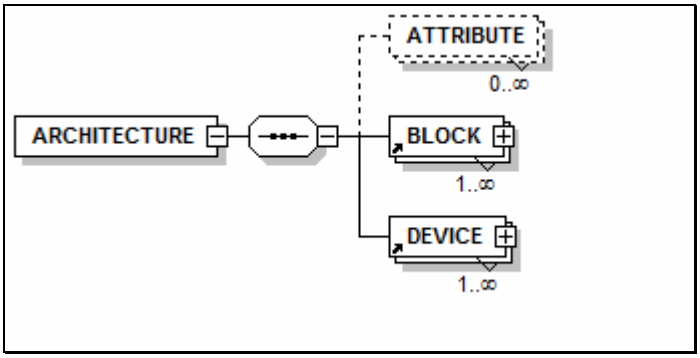


**Figure 6-1: The <ARCHITECTURE> root element and its children**

The <ARCHITECTURE> element is the root of all Altera XML Architecture files. This is the first element encountered as it encapsulates <BLOCK>s and <DEVICE>s that are common to this architecture. The attributes provide a version for the architecture description and a name for the architecture.

The <ARCHITECTURE> tag may have an optional number of <ATTRIBUTE> elements followed by at least one <BLOCK>, but most likely more, and then at least one <DEVICE>. An expanded view of the tree below the <ARCHITECTURE> element is available in .

## 6.2 <COPYRIGHT>

This element has no attributes and simply contains a copyright statement between its start and end tags.

## 6.3 <ATTRIBUTE>

The <ATTRIBUTE> element allows for simple (name, value) paired information to be expressed. It provides additional information about its parent or containing element that is possibly non-essential, but perhaps useful. An example of some common <ATTRIBUTE> elements for the <SUBTYPE> element is:

```
<SUBTYPE>
<ATTRIBUTE name=”X_EXTENT” value=”1” />
<ATTRIBUTE name=”Y_EXTENT” value=”2” />
</SUBTYPE>
```

These <ATTRIBUTE> elements tell us that their parent <SUBTYPE> element has a width (X_EXTENT) of one coordinate block and a height (Y_EXTENT) of 2 coordinate blocks.

Common <ATTRIBUTE> elements for <BLOCK>s and <DEVICE>s are given in Sections 6.4 and 6.7 respectively.

## 6.4 <BLOCK>

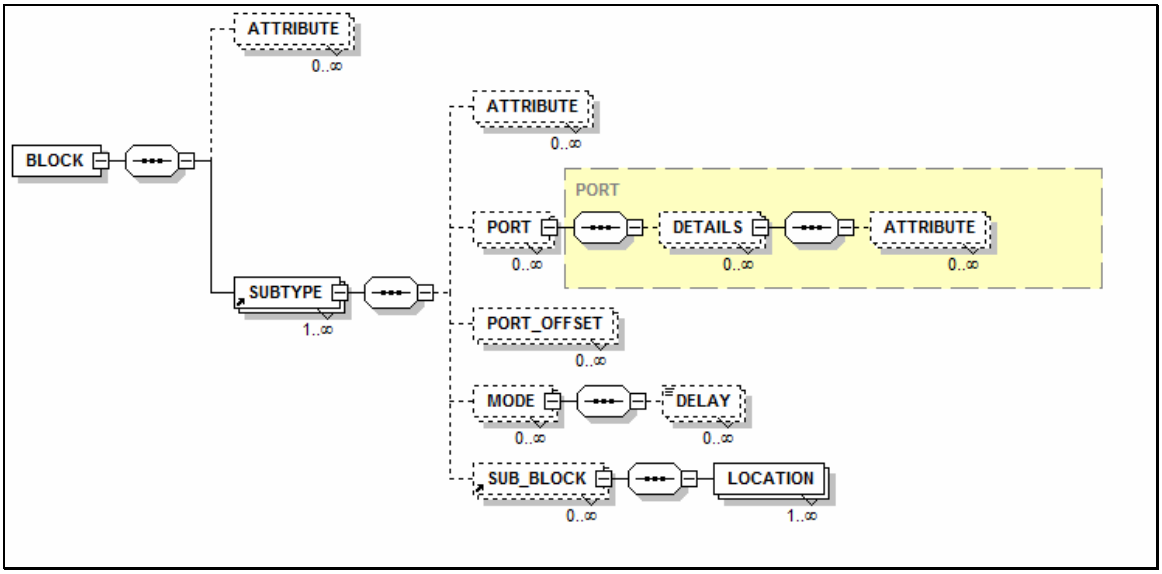


**Figure 6-2: The <BLOCK> element and its children**

The <BLOCK> element describes one of the building blocks of a <DEVICE> in this <ARCHITECTURE>. A block has any number of <ATTRIBUTE> elements followed by one or more <SUBTYPE> elements. The <SUBTYPE> elements provide additional information about a block and allow it to take on possibly different configurations. When a block is instantiated in a device it is done so by name then by subtype name.

Some common <ATTRIBUTE>s of a <BLOCK> include:

- "Can Be Instantiated". This attribute describes whether or not this particular type of <BLOCK> can be instantiated in a device. A value of "TRUE" indicates that it can be, while a value of "FALSE" shows that it cannot.
- "Associated Block Type Name". This is an alias for the block that is sometimes used in other Altera documentation.
- "IS_COMPOSITE". This attribute describes whether or not this block contains other blocks. A value of "TRUE" indicates that at least one other block exists in this block, while "FALSE" indicates that no sub-blocks exist.
- "IS_CONTAINED". This attribute describes whether or not this block is found within other blocks. A value of "TRUE" indicates that this block can be found within at least one other type of block while "FALSE" indicates that it is never another block's sub-block.

Blocks obey several universal properties:

1. Blocks will never overlap their peers. Two blocks on the same level will never share common space in the coordinate system. These situations are illustrated in Figure 6-3.

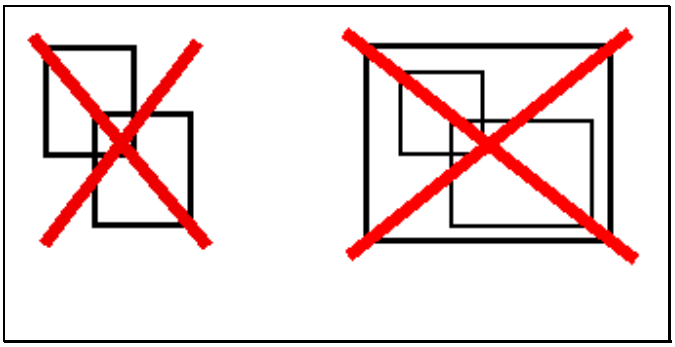

**Figure 6-3: Blocks and their peers will never overlap**

2. Sub-blocks will never extend beyond the extent of their parent block. A contained block, in addition to not overlapping any other contained block, is also completely contained.
3. Blocks will always be rectangular (or square) in shape, never an arbitrary polygon.

## 6.5 <BLOCK_INSTANCE>

The <BLOCK_INSTANCE> element is strictly a child of the <DEVICE> element (see Figure 6-4). It is used to create a physical instance of an architecture block. The block instantiated is picked by the name attribute of the element and the subtype attribute of the tag picks the block's physical implementation (if there are multiple implementations to choose from). The subtype attribute may be omitted, if so the DEFAULT block subtype is assumed.

The only children of the <BLOCK_INSTANCE> element are one or more <LOCATION> elements that specify physical locations of the block relative to the origin of the <DEVICE>.

## 6.6 <DETAILS>

The <DETAILS> element provides additional information about a sub-set of input or output wires on a <PORT> element. If a <PORT> were to have 128 input wires then two separate <DETAILS> sections for that <PORT> could tell us that wires [0-63] had a particular x and y offset and that wires [64-127] had an alternate offset, maybe even on the other side of the <PORT> block.

## 6.7 <DEVICE>

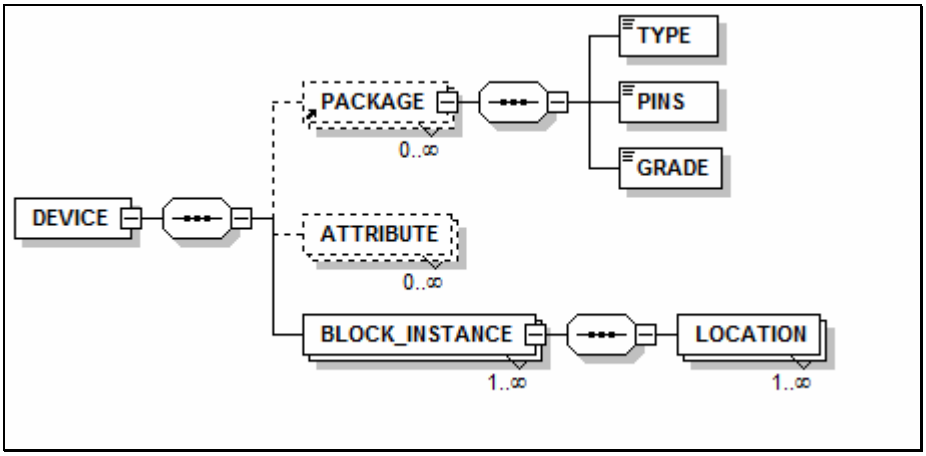


**Figure 6-4: The <DEVICE> element and its children**

The <DEVICE> element defines the physical instances of the architecture building blocks. There can be any number of <PACKAGE> elements that correspond to the different physical device packages available for this is device, any number of <ATTRIBUTE> elements for this device and at least one, but most likely many more, <BLOCK_INSTANCE>s of architecture blocks.

Some common <ATTRIBUTE>s of a <DEVICE> include:

- "X_ORIGIN". This specifies the minimum x co-ordinate available in the Cartesian space used to describe this particular device. No blocks may be instantiated if their left side extends beyond this limit.
- "Y_ORIGIN". This specifies the minimum y co-ordinate available in the Cartesian space used to describe this particular device. No blocks may be instantiated if their bottom extends below this limit.
- "X_EXTENT". This specifies the maximum x co-ordinate available in the Cartesian space used to describe this particular device. No blocks may be instantiated if their right side extends farther than this from the X_ORIGIN.
- "Y_EXTENT". This specifies the maximum y co-ordinate available in the Cartesian space used to describe this particular device. No blocks may be instantiated if the top extends higher than this above the Y_ORIGIN.

Note that the ordered pair (X_ORIGIN, Y_ORIGIN) is always the bottom left corner of the device and the ordered pair (X_EXTENT, Y_EXTENT) is always the top right corner of the device.

## 6.8 <GRADE>

The <GRADE> element defines a speed grade available for a device. Using the lowest <GRADE> number when referencing timing models for a device will give you the fastest possible performance metrics when doing a timing analysis for the device.

## 6.9 <LOCATION>

A simple element, the <LOCATION> has four optional attributes: *x*, *y, subloc* and *name*. The *x* attribute is an integer value representing a horizontal location. The *y* attribute is an integer value representing a vertical location. The *subloc* attribute represents an indexed location in a 1 x 1 block. The name value represents a 'friendly' name for this location. Typically the name value is the name of the location one would see reported if you moused over the location in an Altera architecture view like the Quartus® II Floorplan Editor™ (FLED) or Altera Chip Editor™ (ACE).

For example: a LAB in a Stratix device occupies a 1 x 1 region but contains 10 LE sub-blocks [1]. The LAB has a coordinate with *x* and *y* attributes whereas each LE sub-block has a coordinate with a zero valued *x* and *y* attribute and an indexed *subloc* attribute value in the range [0,9]. No two LABs share the same coordinate, and no two LEs in a LAB share the same *subloc*. This is done to allow sub-block, or clustered block, access without defining a rigid sub-location scheme. The unique *subloc* value is enough to indicate that the LE has a unique, non-overlapping, location at the same coordinate as its containing LAB element.

## 6.10 <PACKAGE>

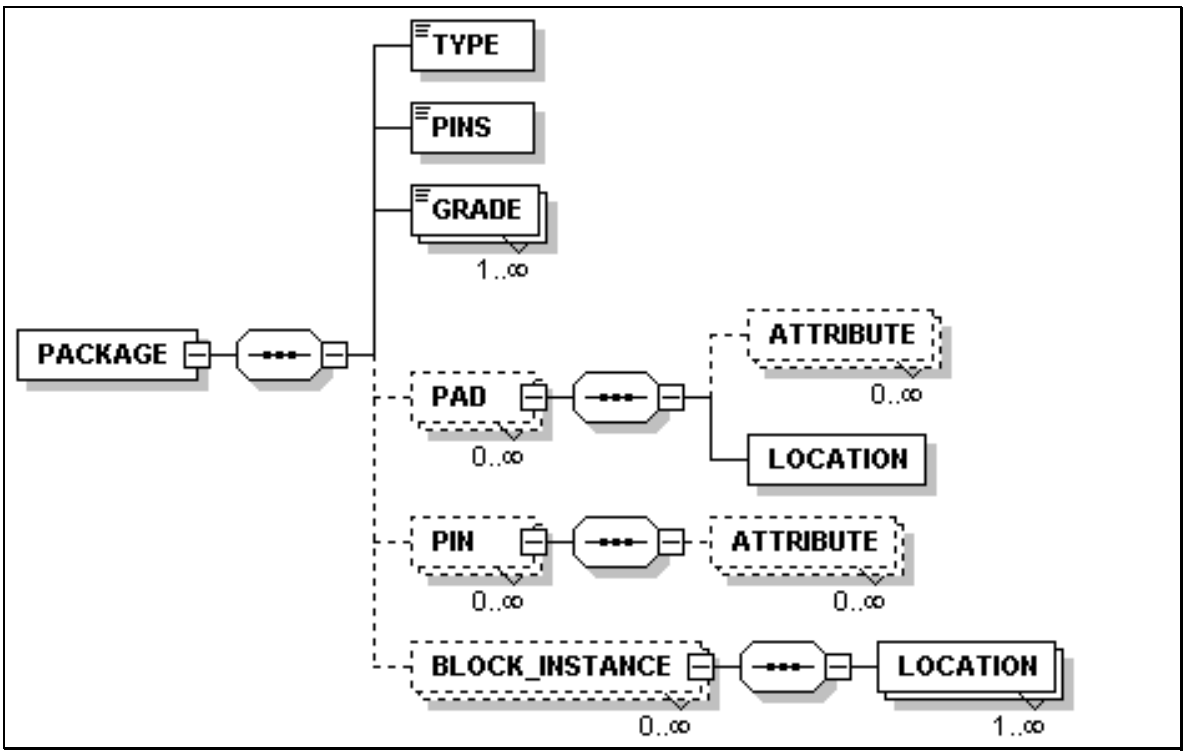


**Figure 6-5: The <PACKAGE> element and its children**

The <PACKAGE> element defines a physical packaging method available for a device. There are five child elements to a <PACKAGE>: the <TYPE> of packaging, the <PINS> available on the package, the speed <GRADE>s available in this packaging, and optionally the <PAD> and <PINS> bonded together for this package configuration. Each <DEVICE> can have multiple <PACKAGE> instances.

## 6.11 <PAD>

The <PAD> element represents one I/O pad on the silicon wafer that is the physical Altera device. Pads are bonded to pins on packing containing the silicon wafer. The <PAD> element represents each pad on the device with a unique ID integer and a <LOCATION>. The <LOCATION> of the <PAD> determines the I/O bank to which the <PAD> belongs. Only pads bonded to physical pins have <PAD> information in the architecture file.

## 6.12 <PIN>

The <PIN> element represents a physical pin on the exterior of the package that is bonded to a <PAD> on the silicon wafer. A <PIN> element has three attributes: the *id* attributes gives a unique integer identifying this pin, the *pads* attribute is a comma-separated list of pad IDs to which this pin is bonded and the *name* attribute is the string name of the pin. Only bonded pins are listed so the total count of <PIN> elements may be less than the number reported by the <PINS> element. Certainly it will never be greater than this number.

## 6.13 <PINS>

The <PINS> element defines the number of physical connection pins available on a specific <PACKAGE> for a <DEVICE>. Not all package pins are bonded to pads on the silicon.

## 6.14 <PORT>

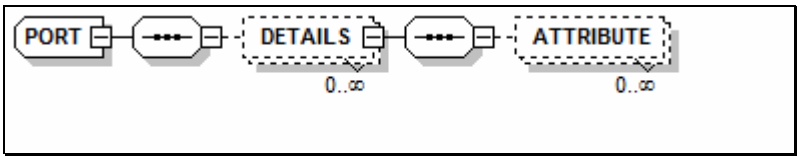


**Figure 6-6: The <PORT> element and its children**

The <PORT> element appears (optionally) below a <SUBTYPE> element and defines a uniquely named port, the direction of the port, and the bus width of the port. It optionally provides additional <DETAILS> about the port. Please be aware that the port names, directions, and bus widths can change from <SUBTYPE> to <SUBTYPE> for a <BLOCK>. For example: the IO block for the Stratix architecture has no less then 8 different <SUBTYPES> for the different IO standards supported by the architecture. Each <SUBTYPE> of the IO block has a set of <PORT>s that provide an appropriate interface for the standard being implemented.

## 6.15 <PORTOFFSET>

The <PORTOFFSET> element defines a physical (x, y) location offset for a port relative to the (0, 0), or the bottom left corner, of the containing <BLOCK>. For blocks larger then 1 x 1 on the device coordinate system this allows you to pin point the location of ports on the block. For example: a 5 x 1 block has three similar input ports, each located at (0,0), (2,0), and (4,0) respectively. If you were hoping to connect another 1 x 1 block located near the top of the 5 x 5 block it would be beneficial to know that third input port at (4, 0) is physically closer to the top right corner of the 5 x 5 block and therefore closer to the 1 x 1 block we are looking to make the connection with.

## 6.16 <SUBTYPE>

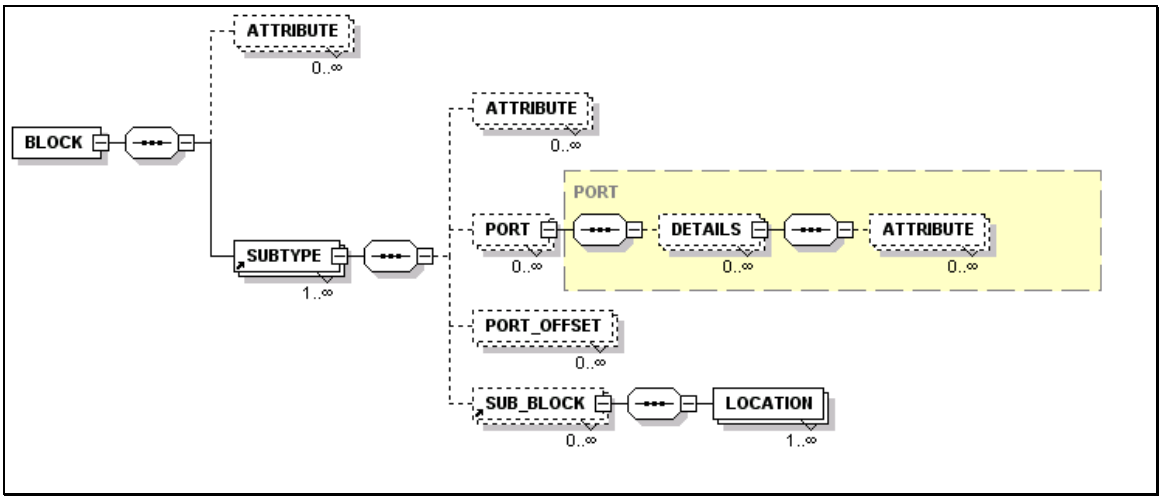


**Figure 6-7: The <SUBTYPE> element and its children**

The <SUBTYPE> element can be thought of as the physical manifestation of the abstract <BLOCK> on the <DEVICE>. It is a particular arrangement of electrical elements that perform a specific set of functions. For example the HIO subtype of the IO block in the Stratix architecture is an arrangement of electrical elements specifically designed to perform high-speed I/O functions.

A <SUBTYPE> can have any number of <ATTRIBUTE>s. Typically you will see an X_EXTENT and Y_EXTENT attribute defined so the physical size of the this subtype instance of the block can be known. For more information on the <ATTRIBUTE> element see Section 6.3.

A <SUBTYPE> can have any number of <PORT>s, each must have a unique name can only be of type INPUT or OUTPUT. These <PORT>s may additionally have <DETAILS> about them. For more information see Section 6.14.

The <PORTOFFSET> provides physical locality information for a <PORT> relative to the bottom left corner of a physical instance of this block's subtype. For more information see Section 6.15.

The <SUB_BLOCK> element defines instances of other blocks within this subtype instance of a block. For more information see Section 6.17.

## 6.17 <SUB_BLOCK>

The <SUB_BLOCK> element defines instances of other blocks within this subtype instance of a block. These instantiations are guaranteed to never be circular. That is, a block will never have a <SUB_BLOCK> of type itself. The <LOCATION>s for the sub-blocks are given as either unique subloc indexes, or as physical locations relative to the bottom left corner of a physical instance of their parent block. For example: a LAB block in the Stratix architecture, with its small 1 x 1 extent, has 10 LE sub-blocks with only unique subloc indices from 0 through 9 to identify their locations. However a MAC block in the same architecture has MAC_MULT sub-blocks with various relative locations and subloc indices because of the larger, 2 x 8, extent of the MAC block [2].

### 6.18 <TYPE>

The <TYPE> element refers to specific type of physical packaging used to house a <DEVICE>. For example: a ball grid array package type.

# 7. Architecture Example

The goal of this example is to provide the user with a visual representation of an architecture described in the Altera XML Architecture Description format. The following figures pertain to the Stratix family of FPGAs and all XML source is taken from the file /xml_descriptions/architectures/source/stratix.xml. We will start by looking at some of the fundamental building blocks of the Stratix family and then see how they can be pieced together to fully describe a physical device. After reviewing the process of creating an XML description for a device, the reader should be comfortable with interpreting the provided architecture XML files and using the information contained within.

## 7.1 Fundamental Blocks

#### 7.1.1 Logic Element

The basic building block for the Stratix family of FPGAs is referred to as a logic element (LE) or a logic cell (LC or LCELL). Pictured below in Figure 7-1 is a black-box representation of the Stratix LE showing all input and output ports (for details on the functionality of the LE and its ports please consult [1]).

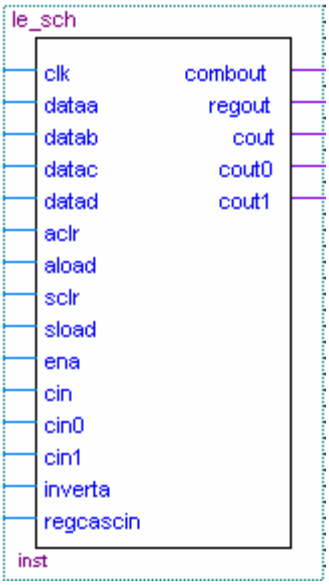


**Figure 7-1: Stratix Logic Element Block**

To describe such a block in an XML file that conforms to the Altera XML Architecture Description standard, one would produce the following (line numbers are added for reference only and would *not* appear in the XML file):

```
1  <BLOCK type="LCELL">
2    <ATTRIBUTE name="CAN_BE_INSTANTIATED" value="FALSE" />
3    <ATTRIBUTE name="ASSOCIATED_BLOCK_TYPE_NAME" value="YEAGER_LCELL" />
4    <ATTRIBUTE name="IS_COMPOSITE" value="FALSE" />
5    <ATTRIBUTE name="IS_CONTAINED" value="TRUE" />
6    <SUBTYPE name="DEFAULT">
7     <ATTRIBUTE name="X_EXTENT" value="1" />
8     <ATTRIBUTE name="Y_EXTENT" value="1" />
9     <PORT name="clk" type="INPUT" width="1"></PORT>
10    <PORT name="dataa" type="INPUT" width="1"></PORT>
11    <PORT name="datab" type="INPUT" width="1"></PORT>
12    <PORT name="datac" type="INPUT" width="1"></PORT>
13    <PORT name="datad" type="INPUT" width="1"></PORT>
14    <PORT name="aclr" type="INPUT" width="1"></PORT>
15    <PORT name="aload" type="INPUT" width="1"></PORT>
16    <PORT name="sclr" type="INPUT" width="1"></PORT>
17    <PORT name="sload" type="INPUT" width="1"></PORT>
18    <PORT name="ena" type="INPUT" width="1"></PORT>
19    <PORT name="cin" type="INPUT" width="1"></PORT>
20    <PORT name="cin0" type="INPUT" width="1"></PORT>
21    <PORT name="cin1" type="INPUT" width="1"></PORT>
22    <PORT name="inverta" type="INPUT" width="1"></PORT>
23    <PORT name="regcascin" type="INPUT" width="1"></PORT>
24    <PORT name="combout" type="OUTPUT" width="1"></PORT>
25    <PORT name="regout" type="OUTPUT" width="1"></PORT>
26    <PORT name="cout" type="OUTPUT" width="1"></PORT>
27    <PORT name="cout0" type="OUTPUT" width="1"></PORT>
28    <PORT name="cout1" type="OUTPUT" width="1"></PORT>
29   </SUBTYPE>
30  </BLOCK>
```

After the start tag that defines the block as an “LCELL” (line 1), some key attributes of the block are listed. These attributes are not mandatory according to the schema (see Figure 5-1) but are generally provided for each block when relevant. They show that the block cannot be instantiated

on its own in a physical device (line 2), is also referred to as a YEAGER_LCELL (line 3), does not contain any blocks within it (line 4) and that it is contained in at least one other block (line 5).

Once these block-level attributes are specified, the first and only subtype of an LCELL block is described. The DEFAULT subtype is a version of the block that fits in a 1 X 1 area on a device (lines 7-8) and has the input and output ports specified (lines 9-28). Note that while the block fits in a 1 X 1 region, it does not occupy the entire 1 X 1 region (you can actually fit 10 LCELLs in a 1 X 1 area). Since there is only one variation of an LCELL, the description of this block is now complete and Figure 7-1 has been specified fully according to the Altera XML Architecture Description. If another type of LCELL existed, its description would follow that of the DEFAULT subtype and be contained within another set of start / end <SUBTYPE> tags.

### 7.1.2 Logic Array Block

As mentioned above, LEs are not instantiated on their own within a physical device. Instead, they are grouped together inside of another type of block, a logic array block (LAB), and these LABs are instantiated in a physical device. Essentially, each lab occupies a full 1 X 1 section of a device and has ten unique sublocations within it, each of which contains an LE. From a high level, Figure 7-2 represents a LAB.

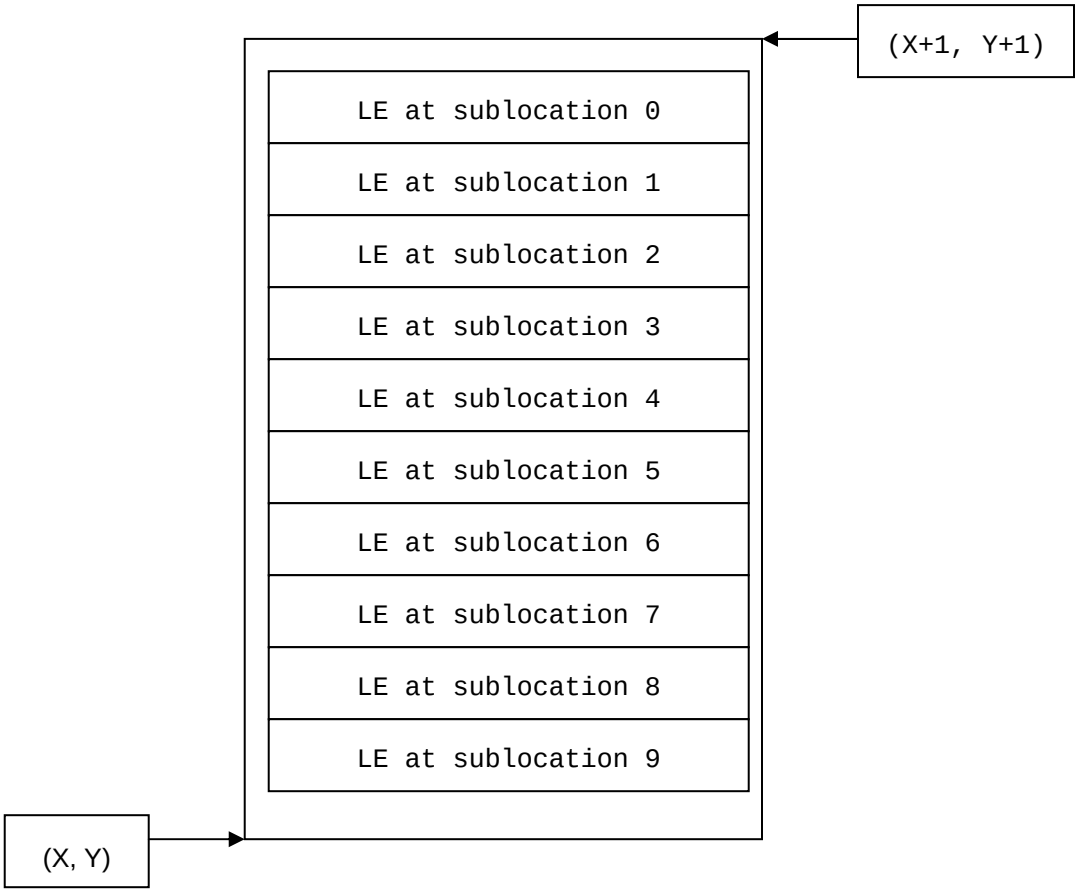


**Figure 7-2: Logic Array Block**

It is clear that there is a parent-child relationship between LABs and LEs. Such a relationship is expressed in an XML description as a BLOCK-SUB_BLOCK relationship. The full XML description of a LAB block is as follows (once again line numbers have been added only for reference):

```
1  <BLOCK type="LAB">
2    <ATTRIBUTE name="CAN_BE_INSTANTIATED" value="TRUE" />
3    <ATTRIBUTE name="IS_COMPOSITE" value="TRUE" />
4    <ATTRIBUTE name="IS_CONTAINED" value="FALSE" />
5    <SUBTYPE name="DEFAULT">
6     <ATTRIBUTE name="X_EXTENT" value="1" />
7     <ATTRIBUTE name="Y_EXTENT" value="1" />
8     <SUB_BLOCK type="LCELL" subtype="DEFAULT">
9      <LOCATION x="0" y="0" subloc="0" />
10     <LOCATION x="0" y="0" subloc="1" />
11     <LOCATION x="0" y="0" subloc="2" />
12     <LOCATION x="0" y="0" subloc="3" />
13     <LOCATION x="0" y="0" subloc="4" />
14     <LOCATION x="0" y="0" subloc="5" />
15     <LOCATION x="0" y="0" subloc="6" />
16     <LOCATION x="0" y="0" subloc="7" />
17     <LOCATION x="0" y="0" subloc="8" />
18     <LOCATION x="0" y="0" subloc="9" />
19    </SUB_BLOCK>
20   </SUBTYPE>
21  </BLOCK>
```

The description starts with the definition of some common attributes for the LAB block. Unlike the LCELL, it can be instantiated in a physical device (line 2), it does contain other blocks within it (line 3) and it is not contained in any other blocks (line 4). There is only one variation of a LAB, noted as the DEFAULT subtype (line 5). This block requires a 1 X 1 region on a physical device (lines 6-7) and contains instances of a sub-block. The sub-block is the DEFAULT variant of an LCELL (line 8). There are multiple instances of this block (lines 9-18), each of which has its bottom left corner at a zero offset from the container block (the DEFAULT variation of a LAB is the container). Since this type of LAB occupies a 1 X 1 region on a given device's plane, any non-zero offset would have been invalid because such an offset would imply that the contained blocks extend outside of their container, which is not allowed. After specifying the DEFAULT variation of the LAB, the block description is complete. Again, more subtypes would appear if there were other types of LABs.

### 7.1.3 Other Basic Blocks

Though the LAB and the contained LEs constitute a large percentage of a device's area, there are several other types of fundamental blocks used by the Stratix family. These include DSP blocks, I/O blocks, RAM blocks and other more sophisticated blocks. For details on the functionality of such blocks please consult [1] [2] [3] and for XML descriptions please view block definitions in the /xml_descriptions/architectures/source/stratix.xml file.

## 7.2 Creating A Device

Now that we have seen how to describe the basic building blocks of a physical device, we can begin to examine how to describe the manner in which these blocks come together to form a device. A full specification requires the placement of hundreds or thousands of blocks on a Cartesian plane that represents the device's surface. One example in the Stratix architecture is the EP1S10F484C5 device. Pictured below in Figure 7-3 is the entire physical device with all blocks as specified in the XML source file.

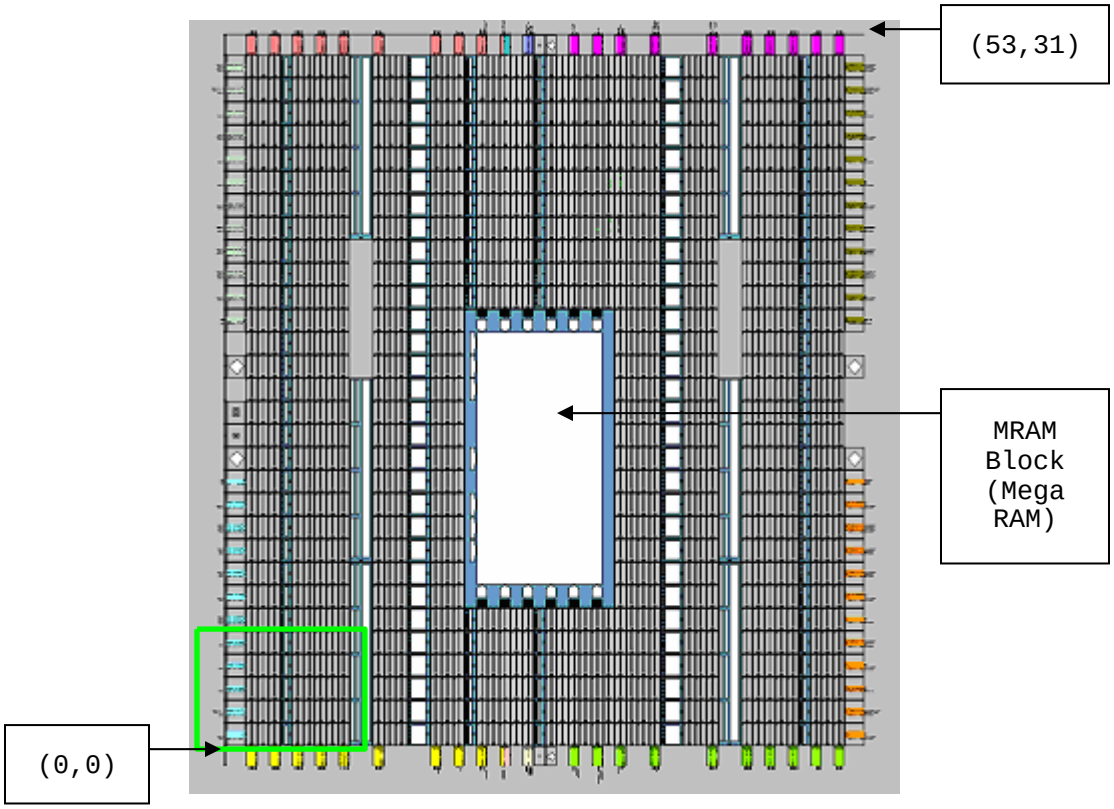


**Figure 7-3: EP1S10F484C5 Device Floorplan**

This device-level view covers all possible co-ordinates on the chip from (0,0) to (53,31) and contains hundreds of blocks distributed across the grid. From this view, the only block that is clearly visible is the large MRAM block in the centre of the device. A more detailed view of the highlighted region in the bottom left corner of the chip is shown below in Figure 7-4.

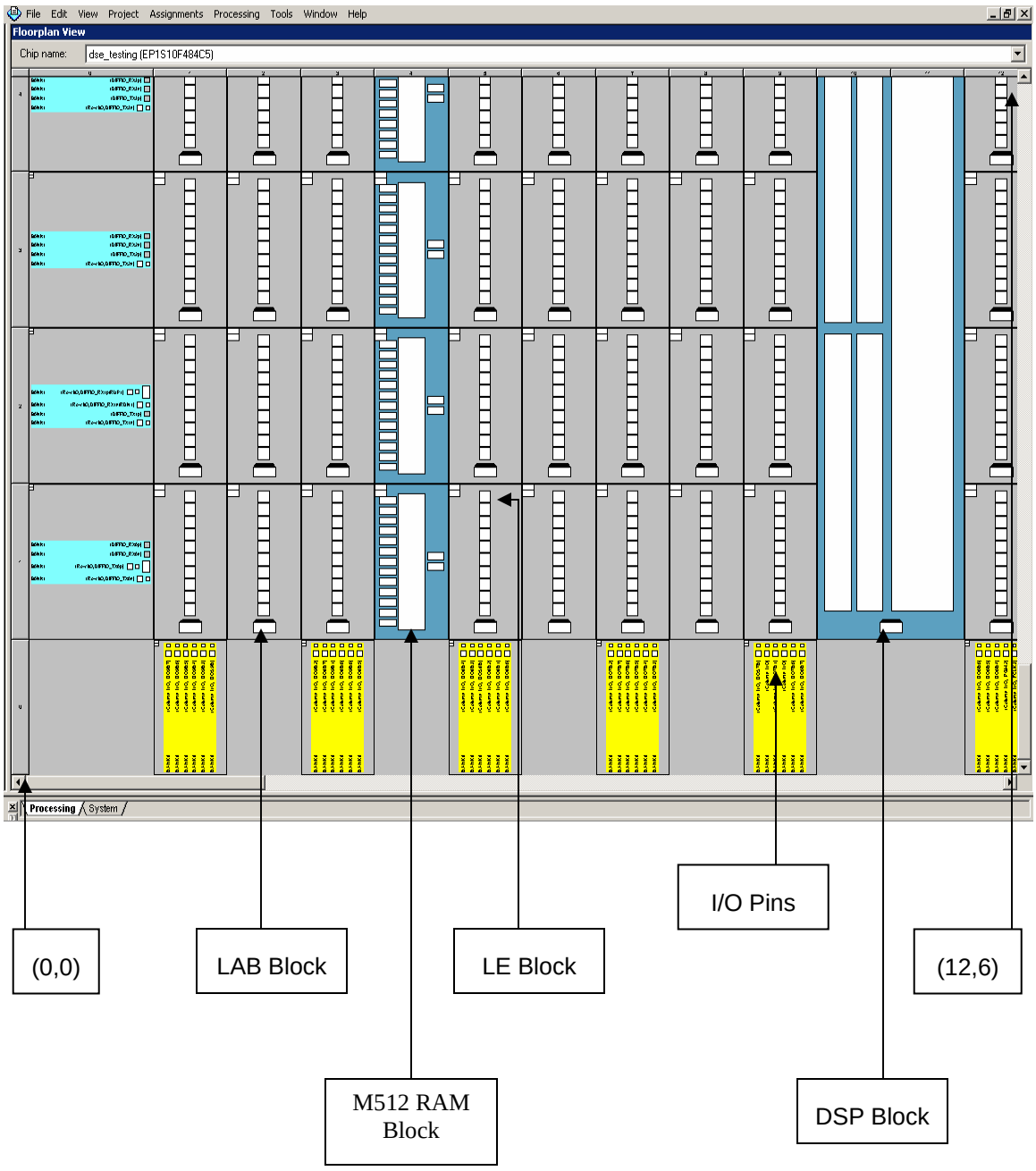


**Figure 7-4: Detailed View of EP1S10F484C5 Device Floorplan**

This offers a better view of how the various fundamental blocks and device components are arranged on the Cartesian plane used to describe the chip's surface.

To describe the entire device requires more than just specifying the blocks and their positions. We would first specify the physical package encompassing the device. The XML required to do this is very straightforward and has the following form:

```
1  <DEVICE name="EP1S10" blk_list_id="20021025_162214_4101958926"
pin_table_version="1.31">
2    <PACKAGE>
3     <TYPE>BGA</TYPE>
4     <PINS>672</PINS>
5     <GRADE>6</GRADE>
6     <GRADE>7</GRADE>
7     <PAD id="0" name="X0Y30SUB_LOC2">
8      <LOCATION x="0" y="30" subloc="2" />
9     </PAD>
                  .
                  .
      (One entry for each pad)
                  .
                  .
10    <PIN id="0" pads="0" name="C1" />
                  .
                  .
      (One entry for each pin bonded to a pad)
                  .
                  .
11 </PACKAGE>
```

This specifies a BGA (ball-grid array) package with 672 pins available in two speed grades (lines 3-6). This is followed by a list of pads and the pins bonded to those pads. Note that not all pins are bonded to pads, so the number of pin entries is necessarily less than or equal to 672 in this case.

After specifying a package, attributes of the device are specified and instances of blocks can now be placed on the device's surface. This is accomplished with XML similar to the following:

```
1   <ATTRIBUTE name="X_EXTENT" value="53" />
2   <ATTRIBUTE name="Y_EXTENT" value="31" />
3   <ATTRIBUTE name="X_ORIGIN" value="0" />
4   <ATTRIBUTE name="Y_ORIGIN" value="0" />
5   <BLOCK_INSTANCE type="LAB" subtype="DEFAULT">
6    <LOCATION x="1" y="1" subloc="0" />
7    <LOCATION x="2" y="1" subloc="0" />
8    <LOCATION x="3" y="1" subloc="0" />
```

```
9   </BLOCK_INSTANCE>
10   <BLOCK_INSTANCE type="M512" subtype="DEFAULT">
11   <LOCATION x="4" y="1" subloc="0" />
12  </BLOCK_INSTANCE>
                  .
                  .
      (Other block instances)
                  .
                  .
13  </DEVICE>
```

The first two lines designate the top-right corner of the device as the ordered pair (53, 31) and the next two lines specify the bottom-left corner as (0,0). These points are labeled in Figure 7-3. Now that the package has been specified and the attributes set, the actual blocks are instantiated in their proper locations. Lines 6-8 place three LABs at (1,1), (2,1) and (3,1) respectively. These LABs are of the subtype DEFAULT and are located next to an M512 RAM block at (4,1). This placement is consistent with Figure 7-4. After all of the other blocks that make up the device are specified (which will certainly include more LABs and RAM blocks), the device has been fully described.

## 7.3 The Stratix Architecture

Up to this point there has been no mention of the top-level entity in this schema; an architecture. The EP1S10F484C5 represents one device in the Stratix architecture, but several others are also defined. The sum total of all blocks used to create devices and all devices created with those blocks makes up an architecture. As a result, to form an architecture definition one would simply use the following XML:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>

2  <ARCHITECTURE xmlns="http://www.altera.com"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.altera.com architecture.xsd"
5   version="0.16.0.1.0a" name="Stratix">
6   <COPYRIGHT>
7    Copyright (C) 2003 Altera Corporation. All rights reserved.
8    This information and code is highly confidential and proprietary
9    information and code of Altera and is being provided in accordance
10   with and subject to the protections of a non-disclosure agreement
11   which governs its use and disclosure.  Altera products and services
```

```
12   are protected under numerous U.S. and foreign patents, maskwork
13   rights, copyrights and other intellectual property laws.  Altera
14   assumes no responsibility or liability arising out of the
15   application or use of this information and code. This notice must be
16   retained and reprinted on any copies of this information and code
17   that are permitted to be made.
18   </COPYRIGHT>

   → Insert all block definitions
   → Insert all device definitions

19  </ARCHITECTURE>
```

This design is self-explanatory, but some of the XML specific attributes warrant some further explanation.

Line 1 is simply an XML declaration indicating that this document uses version 1.0 of XML, has text encoded in ASCII and requires validation against a schema to verify its contents. Line 2 specifies a namespace for operation as http://www.altera.com. Lines 3 and 4 indicate that the schema used to verify this file is called 'architecture.xsd'. Finally a version number is given and the architecture is named "Stratix". This start tag is followed by a copyright statement, which precedes the block and device definitions that complete the architecture definition.

# 8. Parsing the Altera XML Architecture Description File

The XML data exchange format is a well-documented standard with a number of parsers available for most popular languages. For the C++ platform there is an open-source XML parser from the Apache Group called Xerces that implements both the SAX and DOM models. This a robust parser that has undergone rigorous tuning. It can be found at: http://xml.apache.org/ -- There is also a tuned Java implementation that makes an excellent replacement for the Java 2 standard XML parser implementation. For C++ examples that follow the Xerces DOM XML parser is assumed. For more parsers please see Section 9.

## 8.1 Choose a parser that implements the complete XML 1.0 standard

When choosing a parser implementation pay close attention to the XML standard implemented by the parser. Choose a parser that implements the complete XML 1.0 standard. A validating parser will also perform schema validation on the fly for you and is a useful feature to have. Altera recommends the Apache Group's Xerces XML parser for C++ and Java.

## 8.2 Assumptions made are valid for any XML file that validates against the schema

Assumptions made about the format of the data based on the schema for the Altera XML Architecture Description format are always valid from XML document to XML document that can be validated with the schema. This allows you to make querying simplifications based on assumptions about the format and content of the data. For example:

> You can always assume that a <DEVICE> element has at least one <BLOCK_INSTANCE> element with at least one <LOCATION> element. Furthermore the <BLOCK_INSTANCE> element will always have a type and subtype attribute that you can query and match with a <BLOCK> and <SUBTYPE> element that exists under the <ARCHITECTURE> element of the document.

This allows you to write querying routines or even custom classes to represent an Altera XML Architecture Description based on the schema and maintain reliable parsing from XML document to XML document.

## 8.3 Downcasting to an element can always be done

The schema for the Altera XML Architecture Description format allows you to simplify your parsing by ensuring that elements in the <ARCHITECTURE> tree are *not* mixed-mode. That is elements do not contain text and other elements they only contain other elements. This allows you to treat any node you encounter during parsing as an element and to downcast in either Java or C++ from the parser's implementation of a node to the parser's implementation of an element. Downcasting to an element allows for refined querying of attributes and children, not available at the node level because there always exists the possibility that the node being worked with is pure text parsed from a file and not an element.

```
// An Example of Downcasting from a Node to an Element in C++
DOM_Node nodeDevice = someFunctionReturningNodeThatRepresentsADevice();
DOM_Element device = (DOM_Element &) nodeDevice;
// Now you can do more useful things with the device as an element like getting
// all the child elements of this element that have a particular name. Something that
// can't be done on a DOM_Node object:
DOM_NodeList block_instances_list =
        device.getElementsByTagName("BLOCK_INSTANCE");
```

## 8.4 Use XSLT to transform data before you parse

If the data contained in an Altera XML Architecture Description is not exactly as you would like it presented it is easy to use the Extensible Style Sheet Language (XSLT) to transform the data. Base your XSLT transformations on the schema and you can be assured complete and successful transformations when applying the XSLT to any document that validates against the schema. This is a powerful feature of XML and is much faster than doing transformations on data that has already been parsed. For more XSLT references see Section 9.

# 9. Additional References

http://www.w3.org/XML/ -- The World Wide Web Consortium is in charge of the XML standard. This is a good place to start for general standard information on XML, XSLT, XML Schema, DOM and other XML-related standards.

http://www.xml.com/ -- The O'Reilly Group's XML resource site. Lots of tutorials and examples are available under the 'Programming' section. There is also a great annotated version of the XML 1.0 specification, complete with in-line notes and pointers, at http://www.xml.com/axml/testaxml.com.

http://xml.apache.org/ -- Home of the Apache Group's Xerces XML parser. Their goal is "to provide commercial-quality standards-based XML solutions" within the open source framework. In addition to Xerces for C++, Java, Perl and COM you will Xalan, an XSLT stylesheet processor for transforming XML in Java and C++.

CodeNotes for XML, Edited by Gregory Brill – An excellent and invaluable reference for any developer new to XML.

# 10. Referenced Documents

1. “WYSIWYG Device Primitives User Guide For Stratix.”
2. “Stratix RAM WYSIWYG User Guide.”
3. “Stratix MAC WYSIWYG Description.”

# Appendix A: Counting Device Resources – An Example

This Perl script illustrates how to use a tree-parsed version of an Architecture Description File to gather general resource information about the available devices in a family. General resource information gathering is fairly the straight forward with the Architecture Description File format. The exception to this is counting available I/O resources for given device/package/pin arrangement. The script is completely data-driven and makes no assumptions about the parsed data tree save that the format of the parsed file adheres to the Altera Architecture Description File schema.

To use this script cut and paste the Perl code below into a Perl script with the name `resourcecounter.pl`. The instructions for running the script are contained in the comments at the top and detailed comments through out the script offer insight and instruction into building your parsers that consume the data found in the Architecture Description Files.

```
#! perl -w

######################################################################
#                                                                    #
# SYNOPSIS                                                           #
#     perl resourcecounter.pl --xml-file <fname>                     #
#                                                                    #
# ARGUMENTS                                                          #
#     --xml-file=<fname>                                             #
#     Specifies the names of Altera Architecture Description         #
#     files on disk that the script should parse for device          #
#     information. At least one --xml-file argument is required      #
#     and multiple files can be passed with multiple --xml-file      #
#     flags.                                                         #
#                                                                    #
#     --dump-file=<fname>                                            #
#     Optional argument, it can appear only once if at all and       #
#     gives the name of a file on disk to dump debug information     #
#     into as the parsing is performed. The file will be             #
#     overwritten.                                                   #
#                                                                    #
# DESCRIPTION                                                        #
#     Counts resources for devices using information found in        #
#     one or more Altera Architecture Description files in a         #
#     manner similar to the ::quartus::device::get_part_info         #
#     command. Illustrates parsing the XML architecture files        #
#     with Perl. The script makes the very big leap of faith         #
#     that the XML files being passed with the --xml-file            #
#     argument are valid Altera Architecture Description files.      #
#     If they are not the behaviour of this script is                #
#     completely undefined.                                          #
#                                                                    #
# PERL MODULES REQUIRED                                              #
#     The command-line option parsing module Getopt::Long is         #
#     required. This module general pre-installed with most          #
#     Perl 5.+ installation. You can get the module at CPAN if       #
#     your Perl installation does not include it:                    #
#                                                                    #
#        http://search.cpan.org/~jv/Getopt-Long-2.34/                #
#                                                                    #
#     This script use the excellent XML::Twig parser -- a module     #
#     designed for parsing huge XML documents in a tree format.      #
```

```
#     You can get the module at CPAN here:                        #
#                                                                 #
#        http://search.cpan.org/~mirod/XML-Twig-3.11/             #
#                                                                 #
#     The XML::Dumper module may be omitted. The script requires  #
#     a few small modifications to work without this module.      #
#     Removal only affects the script when the --dump-file        #
#     option is used for debugging. The module can be obtained    #
#     at CPAN if you would like to use it:                        #
#                                                                 #
#        http://search.cpan.org/~mikewong/XML-Dumper-0.67/        #
#                                                                 #
###################################################################

#####################################################################
# USE/REQUIRE STATEMENTS                                            #
#####################################################################
use 5.6.1;
use Getopt::Long;
use XML::Dumper;
use XML::Twig;
use strict;

# Arguments:
# @ - Array of strings to print to dump file
#
# Description:
# Dumps data to the dump file.
#
# If the --dump-file flag was not passed to the script then this
# subroutine does nothing at all. The $dump_file variable is
# global to the script.
#
# Returns true if it did dump something; return false if
# nothing was dumped.
sub debug_dump ( @ ) {
   our $dump_file;
   if ($dump_file ne '') {
      unless ( open (DFH, ">>$dump_file") ) {
         return 0;
      }
      print DFH @_;
      close DFH;
   }
   return 1;
}

# Arguments:
# $ - The XML::Twig object
#
# Description:
# Retrieves the name of the architecture described by the XML
# loaded into the XML::Twig object.
#
# Returns a scalar.
sub find_architecture_name ( $ ) {
   my $twig = shift @_;
```

```
   my $root = $twig->root;
   return $root->att('name');
}

# Arguments:
# $ - The XML::Twig object
#
# Description:
# Gathers the following resources for each device it can find the
# XML::Twig tree:
#
#    Family Name <fname>
#    Device Name <dname>
#    Package <pkg>
#    Pin Count <pins>
#    Speed Grades <speeds>
#    I/O Count <ios>
#    Lcell Count <lcells>
#
# Returns an array where each element in the array contains the
# information above organized as a string with :: as a field
# separator. You can extract the data for each device with a
# short bit of code like this:
#
#    foreach (gather_device_resources ($twig)) {
#       my ($fname, $dname, $pkg, $pins,
#           $speeds, $ios, $lcells) = split(/::/);
#    }
#
# If the array is empty no devices were found in the $twig object.
sub gather_device_resources {
   my $twig         = shift @_;
   my @return_array = ();
   my $root         = $twig->root;
   my $family       = $root->att('name');
   my @devices      = get_devices ( $twig );

   # For each device now
   foreach my $device (@devices) {
      my $logic_cells  = get_lcell_count ( $twig, $device );
      my @pkg_and_pins =
            get_package_types_and_pin_counts ( $twig, $device );
      foreach my $pnp (@pkg_and_pins) {
         my $speed_grades =
         join ( ',',
               get_speed_grades ( $twig, $device, $pnp ) );
         my $io_count     = get_io_count ( $twig, $device, $pnp );
         my $data_string  = join ('::',
                                  $family, $device, $pnp,
                                  $speed_grades, $io_count,
                                  $logic_cells);
         push ( @return_array, $data_string );
      }
   }
   return @return_array;
}
```

```
# Arguments:
# $ - The XML::Twig object
#
# Description:
# Finds all the devices for this family in an XML::Twig object.
#
# Returns a list of them. Returns an empty list if no devices are
# found.
sub get_devices ( $ ) {
   my $twig        = shift @_;
   my $root        = $twig->root;
   my @device_list = ();
   foreach my $block ($root->children('DEVICE')) {
      push ( @device_list, $block->att('name') );
   }
   return @device_list;
}

# Arguments:
# $ - The XML::Twig object
# $ - Device name
#
# Description:
# Counts the Logic Cell instances for a device found in an
# XML::Twig object.
#
# Returns the count. Returns 0 if no Logic Cell instances can
# be found.
sub get_lcell_count ( $ $ ) {
   my $twig        = shift @_;
   my $device_name = shift @_;
   my $root        = $twig->root;

   # Figure out how many Logic Cells there are in a default LAB
   # for this particular family. For improved performance you
   # could cache this number some place instead of looking it up
   # for every single device.
   my %lcells_per_lab = ();
   foreach my $block ($root->children('BLOCK')) {
      my $type = $block->att('type');
      if ($type =~ m/LAB/i) {
         foreach my $subtype ($block->children('SUBTYPE')) {
            my $subtype_name = $subtype->att('name');
            $lcells_per_lab{$subtype_name} = 0;
            foreach my $sub_block (
                  $subtype->children('SUB_BLOCK')) {
               my $sub_block_name = $sub_block->att('type');
               if ( $sub_block_name =~ m/^LE_COMB$/ or
                    $sub_block_name =~ m/^LE$/      or
                    $sub_block_name =~ m/^LCELL$/ ) {
                  # Count the number of LOCATIONs
                  $lcells_per_lab{$subtype_name} +=
                     scalar $sub_block->children('LOCATION');
               }
            }
         }
         # The search is over, we have our answer
```

```
            last;
         }
      }

      my %labs_per_device = ();
      foreach my $device ($root->children('DEVICE')) {
         my $temp_device_name = $device->att('name');
         if ($temp_device_name eq $device_name) {
            foreach my $bi ($device->children('BLOCK_INSTANCE')) {
               my $type = $bi->att('type');
               my $subtype = $bi->att('subtype');
               if ($type =~ m/LAB/i) {
                  if (!exists $labs_per_device{$subtype}) {
                     $labs_per_device{$subtype} = 0;
                  }
                  # Count the number of LOCATIONs
                  $labs_per_device{$subtype} +=
                     scalar $bi->children('LOCATION');
                  # Keep searching through all the block instances
                  # because there may be more LAB instantiations
                  # elsewhere in the tree for this device.
               }
            }
            # The search is over, we have our answer
            last;
         }
      }

      # Return # of lcells/lab * # of labs for each LAB subtype
      my $logic_cells = 0;
      foreach my $subtype (keys %labs_per_device) {
         if (exists $lcells_per_lab{$subtype}) {
            $logic_cells +=
               $labs_per_device{$subtype} *
                  $lcells_per_lab{$subtype};
         } else {
            die ("Found a LAB subtype $subtype BLOCK_INSTANCE\n",
                 "that was not found as a subtype of LAB when\n",
                 "the BLOCK for LAB was parsed.\n\n");
         }
      }
      return $logic_cells;
}

# Arguments:
# $ - The XML::Twig object
# $ - Device name
#
# Description:
# Finds all the package and pin count combinations for a specific
# device in an XML::Twig object.
#
# Returns a list with values that are in the form:
#
#    <ptype>::<pincount>
#
# for a specific device. You can use this list to get speed grades
```

```
# and an I/O count for a specific <ptype>::<pincount> combination.
sub get_package_types_and_pin_counts ( $ $ ) {
   my $twig         = shift @_;
   my $device_name  = shift @_;
   my @return_array = ();
   my $root         = $twig->root;

   foreach my $device ($root->children('DEVICE')) {
      my $temp_device_name = $device->att('name');
      if ($temp_device_name eq $device_name) {
         foreach my $package ($device->children('PACKAGE')) {
            my $type = $package->first_child_text('TYPE');
            my $pins = $package->first_child_text('PINS');
            push ( @return_array, "${type}::${pins}" );
         }
         # The search is over, we have our answer
         last;
      }
   }
   return @return_array;
}

# Arguments:
# $ - The XML::Twig object
# $ - Device name
# $ - Package type & pin count in the form <ptype>::<pins>
#
# Description:
# Finds all the speed grades for a specific device with a specific
# package and pin count combination in an XML::Twig object. This
# subroutine expects the package/pin information is in the form
# <ptype>::<pins> -- this is the form the information is in when
# it's returned by the get_package_types_and_pin_counts
# subroutine.
#
# Returns a list of speed grades found. List is empty if the
# device/package/pin combination could not be found or no speed
# grades were found.
sub get_speed_grades ( $ $ $ ) {
   my $twig                 = shift @_;
   my $device_name          = shift @_;
   my ( $ptype, $pincount ) = split ( /::/, shift (@_), 2 );
   my @return_array         = ();
   my $root                 = $twig->root;

   foreach my $device ($root->children('DEVICE')) {
      my $temp_device_name = $device->att('name');
      if ($temp_device_name eq $device_name) {
         foreach my $package ($device->children('PACKAGE')) {
            my $type = $package->first_child_text('TYPE');
            my $pins = $package->first_child_text('PINS');
            if ($type =~ m/$ptype/i and $pins eq $pincount) {
               push ( @return_array,
                      $package->children_text('GRADE') );
            }
         }
         # The search is over, we have our answer
```

```
         last;
      }
   }
   return @return_array;
}

# Arguments:
# $ - The XML::Twig object
# $ - Device name
# $ - Package type & pin count in the form <ptype>::<pins>
#
# Description:
# Counts all the general purpose I/O instances for a specific
# device with a specific package and pin count combination
# found in an XML::Twig object. This subroutine expects the
# package/pin information is in the form <ptype>::<pins> -- this
# is the form the information is in when it's returned by the
# get_package_types_and_pin_counts subroutine.
#
# General purpose I/Os are I/O subtypes that have the
# IS_GENERAL_PURPOSE_IO = TRUE attribute.
#
# Returns the count of general purpose I/O's for the specific
# device/package/pin configuration. Returns 0 if no general
# purpose I/O subtypes exist.
sub get_io_count ( $ $ $ ) {
   my $twig                = shift @_;
   my $device_name         = shift @_;
   my ( $ptype, $pincount ) = split ( /::/, shift (@_), 2 );
   my $root                = $twig->root;

   # Find out what IO subtypes are actually general purpose I/O
   # for this particular family.
   my @gpios = ();
   foreach my $block ($root->children('BLOCK')) {
      my $block_type = $block->att('type');
      if ($block_type =~ m/IO/i) {
         foreach my $subtype ($block->children('SUBTYPE')) {
            my $subtype_name = $subtype->att('name');
            # Assume this is not a global purpose I/O
            my $is_gio = 0;
            # Search attributes to correct this assumption
            foreach my $attribute (
               $subtype->children('ATTRIBUTE')) {
               my $name  = $attribute->att('name');
               my $value = $attribute->att('value');
               if ( $name =~ m/IS_GENERAL_PURPOSE_IO/i and
                    $value =~ m/TRUE/i ) {
                  # Correct initial assumption
                  $is_gio = 1;
               }
            }
            # Save this subtype if it's a general purpose I/O
            push (@gpios, $subtype_name) if $is_gio;
         }
      }
   }
```

```
if (scalar @gpios eq 0) {
   # This should probably be a more serious error
   return 0;
}

my $io_count = 0;

foreach my $device ($root->children('DEVICE')) {
   my $temp_device_name = $device->att('name');
   if ($temp_device_name eq $device_name) {

      # Now build a hash of locations in with keys that have
      # the form: <x>::<y>::<subloc> -- we'll use this later
      # with the package information count I/O's that are
      # actually bonded to pins.
      my %gpio_location_hash = ();
      foreach my $bi ($device->children('BLOCK_INSTANCE')) {
         my $type    = $bi->att('type');
         my $subtype = $bi->att('subtype');
         if ($type =~ m/IO/i) {
            # Is it one of the gpio subtypes we care about?
            foreach my $gpio_subtype (@gpios) {
               if ($subtype =~ m/^$gpio_subtype$/i) {
                  foreach my $l (
                     $bi->children('LOCATION')) {
                     my $x      = $l->att('x');
                     my $y      = $l->att('y');
                     my $subloc = $l->att('subloc');
                     my $key    = $x."::".$y."::".$subloc;
                     $gpio_location_hash{$key} = 1;
                  }
                  # No need to look any further
                  last;
               }
            }
         }
      }

      # Locate the package that has our expected type and
      # pin count for this particular device now.
      foreach my $package ($device->children('PACKAGE')) {
         my $type = $package->first_child_text('TYPE');
         my $pins = $package->first_child_text('PINS');
         if ($type =~ m/$ptype/i and $pins eq $pincount) {
            # Generate a pad id to pin mapping now. This
            # represents all the pads for this device/pkg/pin
            # combination that are bonded to physical pins
            my %pad_to_pin_mapping = ();
            foreach my $pin ($package->children('PIN')) {
               my $pinid = $pin->att('id');
               my $padid = $pin->att('pads');
               $pad_to_pin_mapping{$padid} = $pinid;
            }

            # Now look at each pad. If the pad has the same
            # location as a general purpose I/O in the
```

```
               # %gpio_location_hash AND the pad is bonded to
               # pin (has a key entry in the %pad_to_pin_mapping
               # hash) then we can count this as an I/O for
               # this device/pkg/pin combination.
               foreach my $p ($package->children('PAD')) {
                  my $padid        = $p->att('id');
                  my $location     = $p->first_child('LOCATION');
                  my $x            = $location->att('x');
                  my $y            = $location->att('y');
                  my $subloc       = $location->att('subloc');
                  my $location_key = $x."::".$y."::".$subloc;
                  if (
                      exists $pad_to_pin_mapping{$padid} and
                      exists $gpio_location_hash{$location_key}
                     ) {
                     # Count this as a general purpose I/O
                     ++$io_count;
                  }
               }
               # No need to look any further
               last;
            }
         }
         # The search is over, we have our answer
         last;
      }
   }
   return $io_count;
}

######################################################################
# GLOBAL VARIABLES                                                   #
######################################################################

# If $dump_file is set we dump everything the XML parser returns
# to this file. Really only useful for debugging stuff.
our $dump_file       = '';

# This is an XML::Dumper instance that we can use to translate
# Perl data structures into a nice, easy to read, XML
# representation before dumping the data to the dump file with
# the debug_dump() subroutine.
my $xmldumper        = new XML::Dumper;

# This is a list of the XML files we will be parsing.
my @xml_files        = ();

# This is where we keep all the data we gather
my @device_resources = ();

######################################################################
# START OF SCRIPT                                                    #
######################################################################

print "Resource Counter\n";
print "-------------------------------------------------\n";
print "\n";
```

```
# Get and process the command line options
GetOptions ( 'dump-file=s' => \$dump_file,
             'xml-file=s' => \@xml_files)
   or die ( "Please see script comments for help\n\n");

# Delete the dump file if it already exists
if ($dump_file ne '' and -f $dump_file) {
   unlink ( $dump_file )
      or die ( "Could not delete dump file $dump_file: $!\n" );
}

# Load device resource information for each XML Architecture
# file we were passed via the --xml-file option.
foreach my $xml_file (@xml_files) {
   if (!-f $xml_file) {
      print "Skipping $xml_file -- not a file\n";
      next;
   }

   print "Processing Altera ADF: $xml_file\n";

   # Create a new XML::Twig objec to store data
   my $twig = XML::Twig->new(pretty_print => 'indented');

   # Build twig
   $twig->parsefile($xml_file);

   push ( @device_resources, gather_device_resources ( $twig ) );
   debug_dump ( $xmldumper->pl2xml(\@device_resources) );

   # Purge the twig data structure
   $twig->purge();
}

# Print all the data we've gathered to the screen
print "\n";
foreach my $resource_string (@device_resources) {
   my ($family, $device, $package,
       $pins, $speeds, $io_count, $logic_cells)
         = split ( /::/, $resource_string );
   print "-----------------------------------------------------\n";
   print "Family Name  : ${family}\n";
   print "Device Name  : ${device}\n";
   print "Package Name : ${package}\n";
   print "Pin Count    : ${pins}\n";
   print "Speed Grades : ${speeds}\n";
   print "I/O Count    : ${io_count}\n";
   print "Lcell Count  : ${logic_cells}\n";
   print "-----------------------------------------------------\n";
   print "\n";
}

# All done
exit (0);
```