7

Synthesis Tools for FPGAs

7.1 Introduction

In the 1980s, the VHSIC program was launched which was a major initiative to identify the need for high-level tools for the next generation of integrated circuits. From this the VHDL language and associated synthesis tools were developed; these were seen as a step function for the design of integrated circuits and, until recently, represented the key design entry mechanism for FPGAs. To avoid the high license costs which would act to prevent FPGA users from using their technology, each of the main vendors developed their own tools.

To a great extent, though, this has strengthened the view of FPGAs as a hardware technology, something that is viewed as difficult to program. For this reason, we have seen a major interest in developing high-level synthesis (HLS) tools to make this a more easily programmable technology for software developers. Xilinx has launched the Vivado tools which allow users to undertake C-based synthesis using Xilinx FPGA technology. Altera have developed an approach based on OpenCL, called SDK for Open Computing Language, which allows users to exploit the parallel version of C, developed for GPUs.

A number of other tools have been developed, including C-based synthesis tools both in the commercial (Catapult® and Impulse-C) and academic (GAUT, CAL and LegUp) domain with some focus on FPGA, and higher-level tools such as dataflow-based synthesis tools. The tools that have been chosen and described in this chapter are the ones specifically linked to FPGAs.

The purpose of the chapter is not to provide a detailed description of the tools as these are described much better and in much more detail in their literature, but to give some insight into the tools and their capabilities from a synthesis point of view and also because they would appear to be closely linked to FPGAs. In particular, there is an interest in how much of the architectural mapping outlined in Chapter 8 has now been automated. In particular, the approaches that are of interest are those which allow the algorithmic parallelism to be exploited and also those that permit pipelining to be introduced.

Section 7.2 discusses HLS and describes the problems of using C to model hardware. A C-based approach, specifically Xilinx Vivado HLS tools, is described in Sections 7.3

FPGA-based Implementation of Signal Processing Systems,
Second Edition. Roger Woods, John McAllister, Gaye Lightbody and Ying Yi.

and 7.4, followed by an OpenCL alternative, Altera's SDK for OpenCL, in Section 7.5. Other HLS approaches are briefly reviewed in Section 7.5, including open source C-based tools and those based on a dataflow approach.

7.2 High-Level Synthesis

As the number of transistors on a chip increases, the level of abstraction should increase, as should the design productivity, otherwise design time/effort will increase. The argument is that HLS tools will play an important role in raising productivity. Their main role is to transform an algorithmic description of the behavior of an algorithm into a desired digital hardware solution that implements that behavior. For FPGA designs, the existence of logic synthesis tools from a register transfer level (RTL) description of the design meant that this was a suitable output.

Gajski and Kuhn's Y-chart (Gajski *et al.* 1992; Gajski and Kuhn 1983) describes the HLS design flow (Figure 7.1) A design can be viewed from three perspectives:

- *behavioral*, which describes what the design does, expressed at levels such as transistor functions, Boolean expressions, register transfers, flowcharts/algorithms;
- *structural*, which shows how the design is built using different levels of components such as transistors, gates/flip-flops, registers/ALUs/muxes, processors/memories/buses;
- *physical*, which outlines how the design is physically implemented using different units such as transistor layouts, cells, chips, boards/MCMs

In traditional RTL design flow (Figure 7.1(a)), the behavioral system specifications of the design down to RT level are handled manually by the designer. RTL synthesis and place and route tools is automatically performed, whereas the verification within the automated parts is necessary to match the design against the top-level specifications.

Martin and Smith (2009) provide an outline of the various stages of HLS and suggest that we are now in the third generation after several previous attempts. The first generation was motivated by the observation over two decades ago that RTL-based design

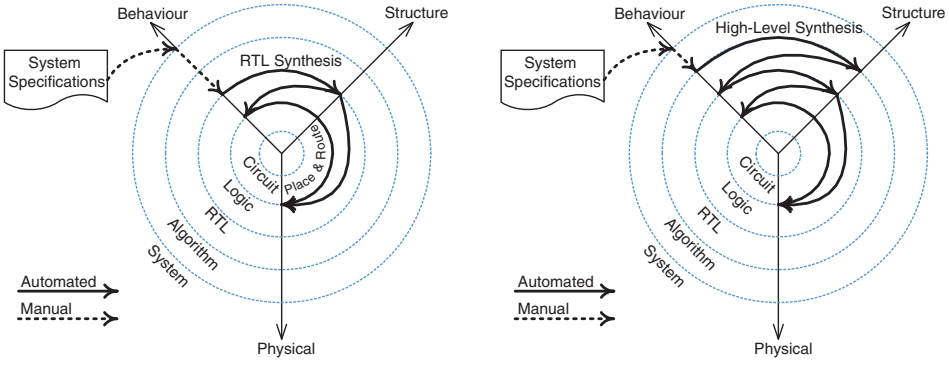

Figure 7.1 High-level synthesis in Gajski and Kuhn's Y-chart

was becoming tedious, complicated, and inefficient as such designs exploded to tens of thousands of lines. To solve this, "behavioral synthesis" was introduced where the detailed description of the architecture was to be replaced with abstract code specifying the design. This required research into sophisticated electronic design automation tools and resulted in the production of multiple behavioral synthesis tools. However, this first attempt failed as the tools were limited and only very small designs were synthesizable; a key example was Synopsys's "Behavioral Compiler." For the second generation, behavioral synthesis was again pursued by academia to fix its problems, and HLS was pursued with improvements such as:

- synthesizing more complex and diverse architectures not just simple data paths;
- synthesizing design I/Os to realize units supporting interface standards and protocols;
- dividing the processing elements into multiple pipelines;
- changing the source language from VHDL or Verilog to the more popular C and C++ languages used by embedded systems designers;
- finding a way to show the trade-offs considered to the designer, namely, speed versus area versus power consumption, latency versus throughput, and memory versus logic.

Since the design space to be explored was vast, thousands of solutions could be created by the tool to find the best one, so it took a long time to develop appropriate HLS-based tools.

While the HLS idea was being developed, another method of raising the abstraction layer was introduced to the market: schematic design to create complicated systems through inserting large reusable IPs, such as processor cores and memory structures, comprising about 80–90% of the design, and 10–20% RTL design including differentiating features. Despite the wide adoption of IP reuse through schematic design, HLS tools reached a level of quality to produce high-performance results for complex algorithms, in particular DSP problems. HLS tools were used to design high-performance IP blocks at high-level languages rather than create the whole system.

With the inclusion of fast processors on chips in SoCs, some designers started to code the 10–20% differentiating part of their design, as software on the processors, hence avoiding any RTL coding in the schematic method. Altera's OpenCL was designed for software engineers who needed software code running on large parallel processor architectures, and also FPGA hardware, as it requires little understanding of hardware.

Given that the previous attempts were viewed to have failed, Martin and Smith (2009) suggest several reasons why this generation of tools would succeed:

- A focus on applications where the tools are being applied in domains where they are expected to have a higher probability of success.
- Algorithm and system designers with the right input languages, allowing them to use languages with which they are comfortable (e.g. C variants, MATLAB®), thus avoiding the learning of special languages.
- Use of compiler-based optimizations which has enabled designers to achieve improved design outputs.
- Performance requirements need significant amounts of signal and multimedia processing and thus need hardware acceleration.
- With FPGAs, the measurement criteria are different than for ASIC as the design has to "fit" into a discrete FPGA size and has to work fast enough, but within the FPGA

speed and size capacity; thus HLS synthesis with FPGA targets is a perfect way of quickly getting an algorithm into hardware.

Overall, the tools appear to be based on C, C++ or C-like tool entry which, as the authors have outlined above, is a major advantage.

7.2.1 HLS from C-Based Languages

Thus there has always been a challenge in using C to model hardware. Given the sequential nature of C, a lot of C-based synthesis tools will translate the C description into some internal data model and then use a series of functionalities to extract the processes, ports and interconnections. This is central to a lot of the design approaches, and a lot of the classical synthesis tools tend to lean on their major investment in classical synthesis tools to achieve the mapping.

Of course, the alternative approach is to adopt C-based languages such as OpenCL, which allows parallelism to be captured. In addition to the ability to capture algorithmic representation, OpenCL defines an application program interface (API) that allows programs running on the host to launch kernels on the compute platform. The language has been driven by the major developments in GPUs which initially developed for processing graphics, but are now applied to a wide range of applications.

The next two sections describe these two varying approaches, one from Xilinx which is based upon C-based synthesis called Vivado and the other from Altera called SDK for OpenCL. While initially Xilinx and Altera created HLS tools which were complementary and created for different groups of users, they have started to add similar functionality to each other's tools.

7.3 Xilinx Vivado

The Xilinx Vivado HLS tool converts a C specification into an RTL implementation synthesizable into a Xilinx FPGA (Feist 2012). C specifications can be written in C, C++, SystemC, or as an OpenCL API C kernel. The company argues that it saves development time, provides quick functional verification, and offers users controlled synthesis and portability.

Figure 7.2 shows that the algorithm can be specified C, C++, or SystemC. These functions are synthesized into RTL blocks and the top-level function arguments are synthesized into RTL I/O ports. Each loop iteration is scheduled to be executed in programmable logic and loops can be unrolled using directives to allow for all iterations to run in parallel. C code arrays are synthesized into block RAM.

Hardware optimized C libraries are available including arbitrary precision data types (to allow optimization to FPGA libraries), HLS stream library, math functions, linear algebra functions, DSP functions, video functions and an IP library. During the synthesis process, a microarchitecture is explored. The tool allows IP system integration and provides RTL generation in VHDL or Verilog.

Vivado creates the optimal implementation based on the default behavior constraints and the user-specified directives. It uses the classical stages of *scheduling* (determining which operations occur when), *binding* (allocating the hardware resource for each

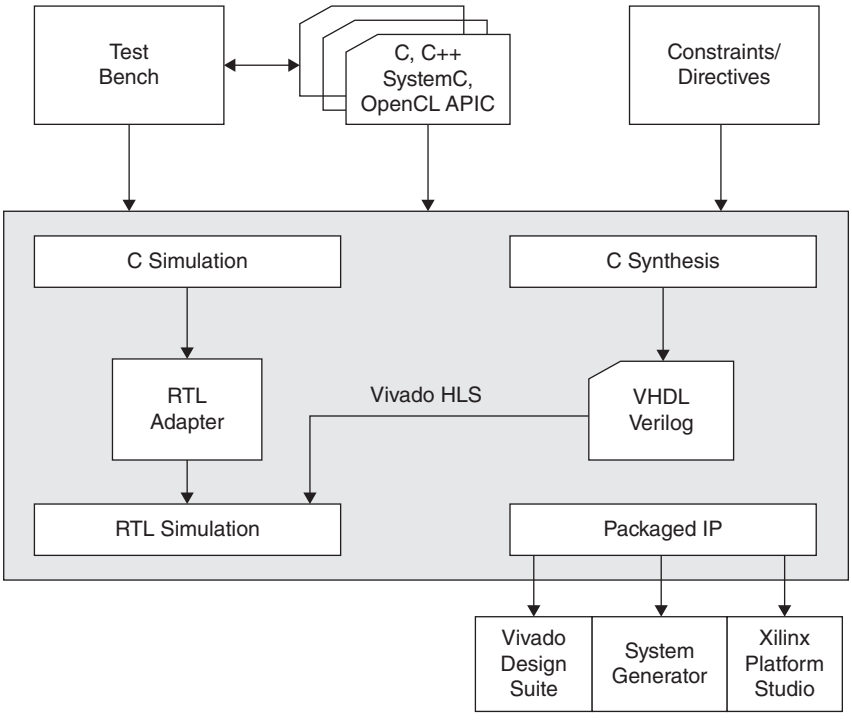

Figure 7.2 Vivado HLS IP creation and integration into a system

scheduled operation) and *control-logic extraction* (which allows the creation of an FSM that sequences the operations).

The user can change the default behavior and allow multiple implementations to be created targeted at reduced area, high-speed, etc. The important performance metrics reported by the HLS are area, latency, loop iteration latency (clock cycles to complete one loop iteration) and loop latency (clock cycles to execute all loop iterations). Generally, the designer can make decisions on functionality, performance including pipeline register allocation, interfaces, storage, design exploration and partitioning into modules.

7.4 Control Logic Extraction Phase Example

The following example shows the control logic extraction and I/O port implementation phase of Vivado. A data computation is placed inside a for loop and two of the function arguments are arrays. The HLS extracts the control logic from the C code and creates an FSM to sequence the operations. The control structure of the C code for loop and the FSM are the same:

$$
\begin{array}{r}
C0 \\
C1, C2, C3 \\
C1, C2, C3 \\
C1, C2, C3 \\
C0, \ldots
\end{array}
$$

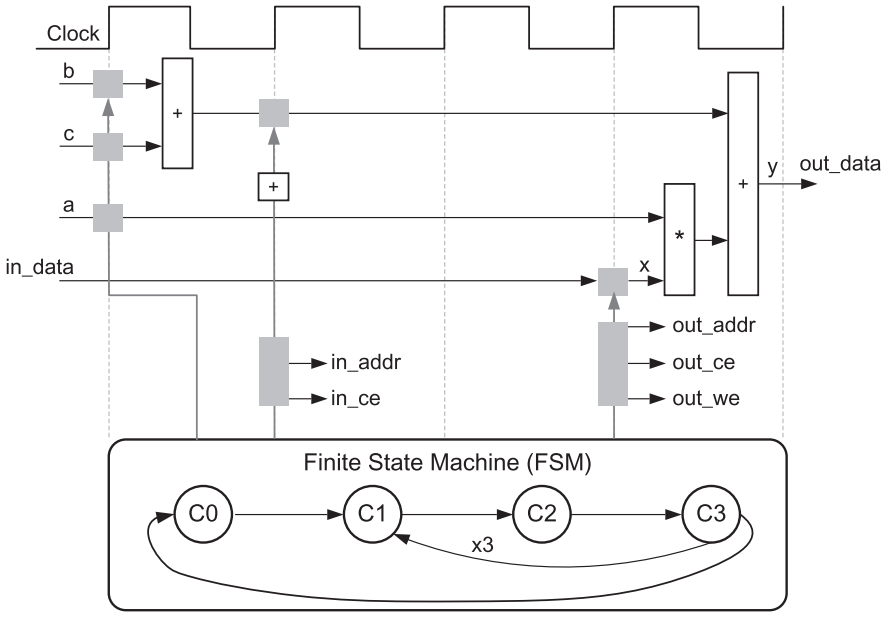

Figure 7.3 Control logic extraction and I/O port implementation example

This is much more easily seen in the timing diagram in Figure 7.3

FSM controls when the registers store data and controls the state of any I/O control signals. Addition of *b* and *c* is moved outside the for loop and into state *C*0. The FSM generates the address for an element in *C*1 and an adder increments to count how many times that the design iterates around *C*1, *C*2, and *C*3. In *C*2, the block RAM returns the data for *in_data* and stores as *x*. By default, arrays are synthesized into block RAMs. In *C*3, the calculations are performed and output is generated. Also the address and control signals are generated to store the value outside the block.

```
void F (int in [3], char a, char b, char c, int out[3]){
int x, y;
for (int i = 0; i ¡ 3; i++) {
x = int [i];
y = a * x + b + c;
out [i] = y;
}
}
```

7.5 Altera SDK for OpenCL

Altera makes the case that the OpenCL standard inherently offers the ability to describe parallel algorithms to be implemented on FPGAs at a much higher level of abstraction than HDLs. The company argues that the OpenCL standard more naturally matches

the highly parallel nature of FPGAs than do sequential C programs. OpenCL allows the programmer to explicitly specify and control the thread-level parallelism, allowing it to be exploited on FPGAs as the technology offers very high levels of parallelism.

OpenCL is a low-level programming language created by Apple derived from standard ANSI C. It has a lot of the functionality of C but does not have certain headers, function pointers, recursion, variable-length arrays or bit fields. However, it has a number of extensions to extract parallelism and also includes an API which allows the host to communicate with the FPGA-based hardware accelerators, either from one accelerator to another or to the host over PCI Express. In addition, an I/O channel API is needed to stream data into a kernel directly from a streaming I/O interface such as 10Gb Ethernet.

The Altera SDK for OpenCL tools provides the designer with a range of functionality to implement OpenCL on heterogeneous platforms including an emulator to step through the code on an x86, a detailed optimization report to highlight loop dependencies, a profiler and a compiler capable of performing over 300 optimizations on the kernel code and producing the entire FPGA image in one step.

In standard OpenCL, the OpenCL host program is a pure software routine written in standard C/C++. The computationally expensive function which will benefit from acceleration on FPGA is referred to as an OpenCL kernel. Whilst these kernels are written in standard C, they are annotated with constructs to specify parallelism and memory hierarchy. Take, for example, a vector addition of two arrays, *a* and *b*, which produces an output array. Parallel threads will operate on the each element of the vector. If this can be accelerated with a dedicated processing block, then an FPGA offers massive amounts of fine-grained parallelism. The host program has access to standard OpenCL APIs that allow data to be transferred to the FPGA, invoking of the kernel on the FPGA and return of the resulting data.

A pipelined circuit to implement this functionality is given in Figure 7.4. For simplicity, assume the compiler has created three pipeline stages for the kernel. On the first clock cycle, thread 0 is clocked into the two load units and indicates that the first elements of data from arrays *a* and *b* should be fetched. On the second clock cycle, thread 1 is clocked in at the same time that thread 0 has completed its read from memory and stored the results in the registers following the load units. On cycle 3, thread 2 is clocked in, thread 1 captures its returned data, and thread 0 stores the sum of the two values that it loaded. Eventually the pipeline will be filled and numerous computations will be carried out in parallel (Altera 2013).

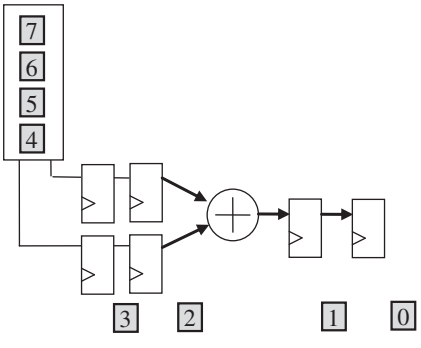

Figure 7.4 Pipelined processor implementation

In addition to the kernel pipeline, Altera's OpenCL compiler creates interfaces to external and internal memory. This can include the connections to external memory via a global interconnect structure that arbitrates multiple requests to a group of external DDR memories and also through a specialized interconnect structure to on-chip RAMs. These specialized interconnect structures ensure high performance and best organization of requests to memory.

Altera SDK for OpenCL is in full production release and supports a variety of host CPUs, including the embedded ARM® Cortex®-A9 processor cores. It supports scalable solutions on multiple FPGAs and multiple boards as well as a variety of memory targets, such as DDR SDRAM, QDR SRAM and internal FPGA memory. Half-precision as well as single- and double-precision floating-point is also supported.

7.6 Other HLS Tools

Obviously, FPGA vendors have a key role in producing place and route tools as this is an essential stage in implementing users' designs in their technologies. More recently, this has been extended to HLS tools particularly as the user base of their technology expands to more general applications of computing. Whilst the previous two sections have highlighted the two major FPGA vendors' approach to HLS, there have also been a number of other approaches, both commercial and academic. A sample of such tools is included in this section in no particular order of importance or relevance.

7.6.1 Catapult

Mentor Graphics offers the Catapult® HLS platform which allows users to enter their design in C, SystemC or C++ and produce RTL code to allow targeting to FPGA. The key focus is the tool's ability to produce correct-by-construction, error-free, power-optimized RTL (Mentor Graphics 2014). The tool derives an optimal hardware microarchitecture and uses this to explore multiple options for optimal power, performance and area. This allows design iteration and faster reuse.

Whilst the tool was initially aimed at a wide range of applications, it has now been focused towards power optimization. In addition, the goal of reduction in design and verification has been achieved by having a fixed architecture and optimizing on it. Indeed, the presentation material indicates that the approach of mapping to a fixed architecture has to be undertaken at the design capture stage. This is interesting, given the recent trend of creating soft processors on FPGA as outlined in Chapter 12.

The PowerPro® product allows fast generation of fully verified, power-optimized RTL for ASIC, SOC and FPGA designs. The company argues that this allows analysis of both static and dynamic RTL power usage and allows the user to automatically or manually create power-optimized RTL. From an FPGA perspective, there is clearly a route for trading off parallelism and pipelining as outlined in Chapter 8 as this can be used to alter the power profile as outlined in Chapter 13. Whilst the latter description talks about this from a generated architecture, this approach uses a fixed architecture.

7.6.2 Impulse-C

Impulse-C is available from from Impulse Accelerated Technologies (see Impulse Accelerated Technologies 2011) and provides a C-to-gates workflow. Algorithms are

represented using using a communicating sequential process and a library of specific functions, and then described in Impulse-C, which is a standard ANSI C language. The communication between processes is performed mainly by data streams or shared memories, which translates into physical wires or memory storage. Signals can also be transferred to other processes as flags for non-continuous communication. The key is that this allows capture of process parallelization and communication.

The code can be compiled using a standard C compiler and then translated into VHDL or Verilog. As well as the code for the functional blocks, the tools can generate the controller functionality from the communication channels and synchronization mechanisms. Pragma directives are used to control the hardware generation throughout the C code, for example, and to allow loop unrolling, pipelining or primitive instantiation. Also existing IP cores in the form of VHDL code can be incorporated.

Each defined processes is translated to a software thread allowing the algorithm to be debugged and profiled using using standard tools. The co-design environment includes tools for co-simulation and co-execution of the algorithms. The approach is targeted at heterogeneous platforms allowing compilation onto processors and optimization onto a programmable logic platform. Impulse-C code can be implemented in a growing number of hardware platforms and, specifically, Altera and Xilinx FPGA technologies.

The tools have been applied to core DSP functions, many of which were described in Chapter 2, but also more complex systems, specifically, filters and many image processing algorithms including edge enhancement, object recognition, video compression and decompression and hyperspectral imaging. There have also been some financial applications, specifically to high-frequency trading.

7.6.3 GAUT

GAUT is an academic HLS tool (http://www.gaut.fr/) that has been targeted at DSP applications (Coussy *et al.* 2008). It allows the user to start from a C/C++ description of the algorithm and supports fixed-point representation to allow more efficient implementation on FPGAs. The user can set the throughput rate and the clock period as well as other features such as memory mapping (Corre *et al.* 2004) and I/O timing diagrams (Coussy *et al.* 2006). The tool synthesizes an architecture consisting of a processing unit, a memory unit and a communication and interface block. The processing unit is composed of logic and arithmetic operators, storage elements, steering logic and an FSM controller.

The flow for the tool is given in Figure 7.5. It starts with the design of the architecture, which involves selecting the arithmetic operators, then the memory registers and memory banks involving memory optimization, followed by the communication paths such as memory address generators and the communication interfaces. It generates not only VHDL models but also the testbenches and scripts necessary for the Modelsim simulator. It has been applied to a Viterbi decoder and a number of FIR and LMS filters, giving a reduction in code of around two orders of magnitude.

7.6.4 CAL

Sequential software programming depends mostly on HLS tools to automatically extract the parallelism of the code. Other than the automatic detection, to cover the language concurrency-support limitations, some libraries are also introduced or features such as

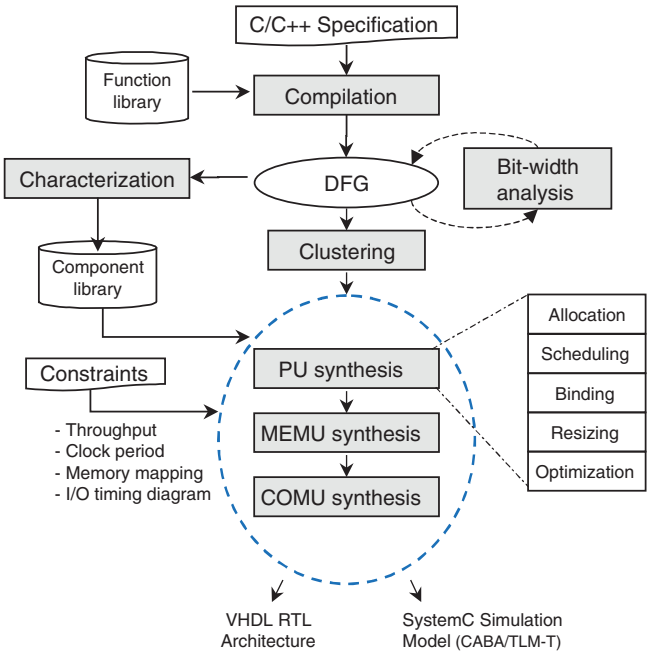

Figure 7.5 GAUT synthesis flow

pragmas are included. These changes in a sequential language to make it executable on hardware have led to different implementations of that language. Development of languages based on the computation model which reflects the hardware specifics and parallel programming seems a better approach than adapting sequential C-like languages to support hardware design.

A language designed to support both parallel and sequential coding constructs and expression of applications as network processes is CAL (Eker and Janneck 2003). The CAL actor language was developed by Eker and Janneck in 2001 as part of the Ptolemy project (Eker *et al.* 2003) at the University of California at Berkeley. CAL is a high-level programming language of the form of a dataflow graph (DFG), for writing actors which transform input streams into output streams. CAL is an ideal language for use as a single behavioral description for software and hardware processing elements.

A subset of the CAL language which has been standardized by the ISO MPEG committee is reconfigurable video coding or RVC-CAL. The main reason for the introduction of RVC is to provide reuse of commonalities among various MPEG standards, and their extension through system-level specifications (Eker *et al.* 2003). This provides a more flexible and faster path to introducing new MPEG standards. The RVC framework is being developed by the MPEG to provide a unified high-level specification of current and future MPEG video coding technologies using dataflow models. In this framework, a decoder is generated by configuring video coding modules which are standard MPEG toolbox libraries or propriety libraries. RVC-CAL is used to write the reference software of library elements. A decoder configuration is defined in XML language by connecting a set of RVC-CAL modules.

The RVC-CAL language limits the advanced features of the CAL language. There are some tools available for development of applications based on RVC-CAL language as summarized in the following.

The Open RVC-CAL Compiler (ORCC) is an open source dataflow development environment and compiler framework which uses RVC-CAL, allows the transcompilation of actors and generates equivalent codes depending on the chosen back-ends (Eker *et al.* 2003). ORCC is developed within the Eclipse-based IDE as a plug-in with graphical interfaces to ease the design of dataflow applications.

CAL2HDL (Janneck *et al.* 2008) was the first implementation of a direct hardware code generation from CAL dataflow programs. A CAL actor language is first converted into an XML language independent model from which Verilog code can be generated using an open source tool. The tool supports a limited subset of CAL actor language such that complex applications cannot be easily expressed or synthesized using this tool. In addition, Xronos (Bezati *et al.* 2013) is an evolution of CAL2HDL and TURNUS (Brunei *et al.* 2013) is a framework used for iterative design space exploration of RVC-CAL programs to find design solutions which satisfy performance constraints or optimize parts of the system.

7.6.5 LegUp

LegUp is an open source HLS tool that allows a standard C source program to be synthesized onto a hybrid FPGA-based hardware/software system (Canis *et al.* 2013). The authors envisage implementing their designs onto an FPGA-based 32-bit MIPS soft processor and synthesized FPGA accelerators. It has been created using modular C++ and uses the state-of-the-art LLVM compiler framework for high-level language parsing and its standard compiler optimizations (LLVM 2010). It also uses a set of benchmark C programs that can be used to a combined hardware/software system and allows specific functionality to be added (Hara *et al.* 2009).

7.7 Conclusions

The chapter has briefly covered some of the tools used to perform HLS. The purpose of the chapter has been to give a brief overview of some relevant tools which may in some cases cover the types of optimizations covered in Chapter 8. In some cases the tools start with a high-level description in C or C++ and can produce HDL output in the form of VHDL or Verilog, allowing vendors' tools to be used to produce the bit files.

C-based tools are particularly useful as many of the FPGA platforms are SoC platforms comprising processors, memory and high-speed on-chip communications. The ability to explore optimization across such platforms is vital and will become increasingly so as future systems requirements evolve as heterogeneous FPGA platforms become ever more complex. Such issues are addressed in the tools outlined in Chapter 10.

Bibliography

Altera Corp. 2013 Implementing FPGA Design with the OpenCL Standard. White paper WP-01173-3.0. Available from www.altera.com (accessed June 11, 2015).

Bezati E, Mattavelli M, Janneck JW 2013 High-level synthesis of dataflow programs for signal processing systems. In *Proc. Int. Symp. on Image and Signal Processing and Analysis*, pp. 750–754.

Brunei SC, Mattavelli M, Janneck JW 2013 TURNUS: A design exploration framework for dataflow system design. In *Proc. IEEE Int. Symp. on Circuits and Systems*. doi 10.1109/ISCAS.2013.6571927

Canis A, Choi J, Aldham M, Zhang V, Kammoona A, Czajkowski T, Brown SD, Anderson JH. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embedded Computing Systems*, 13(2), article 24.

Corre G, Senn E, Bomel P, Julien N, Martin E 2004 Memory accesses management during high level synthesis. In *Proc. IEEE Int. Conf. on CODES+ISSS*, pp. 42–47.

Coussy P, Casseau E, Bomel P, Baganne A, Martin E 2006 A formal method for hardware IP design and integration under I/O and timing constraints. *ACM Trans. on Embedded Computing Systems*, 5(1), 29–53.

Coussy P, Chavet C, Bomel P, Heller D, Senn E, Martin E 2008 GAUT: A high-level synthesis tool for DSP applications. In Coussy P, Morawiec A (eds) *High-Level Synthesis: From Algorithm to Digital Circuit*, pp. 147–169. Springer, New York.

Eker J, Janneck J 2003 CAL language report. Technical Report UCB/ERL M 3. University of California at Berkeley.

Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y 2003 Taming heterogeneity – the Ptolemy approach. *Proc. IEEE*, 91(1), 127–144.

Feist T 2012 Vivado design suite. White Paper WP416 (v1.1). Available from www.xilinx.com (accessed May 11, 2016).

Gajski DD, Dutt ND, Wu AC, Lin SY-L 1992 *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, Norwell, MA.

Gajski DD, Kuhn RH 1983 Guest editors' introduction: New VLSI tools. *Computer*, 16(2), 11–14.

Hara Y, Tomiyama H, Honda S, Takada H. 2009 Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *J. of Information Processing*, 17, 242–254.

Impulse Accelerated Technologies 2011 Impulse codeveloper C-to-FPGA tools. Available from http:// www.impulse accelerated.com/products_universal. htm (accessed May 11, 2016).

Janneck JW, Miller ID, Parlour DB, Roquier G, Wipliez M, Raulet M 2008 Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In *Proc. Workshop on Signal Processing Systems*, pp. 287–292.

LLVM 2010. The LLVM compiler infrastructure project. Available from http://www.llvm.org (accessed May 11, 2016).

Martin G, Smith G 2009 High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4), 18–25.

Mentor Graphics 2014 High-level synthesis report 2014. Available from http://s3.mentor.com/ public_documents/whitepaper/resources/mentorpaper_94095.pdf (accessed May 11, 2016).